Improved Alpha Testing Using Hashed Sampling

Chris Wyman, *Member, IEEE,* and Morgan McGuire

Abstract—
We further describe and analyze the idea of *hashed alpha testing* from Wyman and McGuire [1], which builds on stochastic alpha testing and simplifies stochastic transparency. Typically, *alpha testing* provides a simple mechanism to mask out complex silhouettes using simple proxy geometry with applied alpha textures. While widely used, alpha testing has a long-standing problem: geometry can disappear entirely as alpha mapped polygons recede with distance. As foveated rendering for virtual reality spreads, this problem worsens as peripheral minification and prefiltering introduce this problem on *nearby* objects.

We first introduce the notion of *stochastic alpha testing*, which replaces a fixed alpha threshold of $\alpha_\tau = 0.5$ with a randomly chosen $\alpha_\tau \in [0..1)$. This entirely avoids the problem of disappearing alpha-tested geometry, but introduces temporal noise.

Hashed alpha testing uses a hash function to choose α_τ procedurally. With a good hash function and inputs, hashed alpha testing maintains distant geometry without introducing more temporal flicker than traditional alpha testing. We also describe how hashed alpha interacts with temporal antialiasing and applies to alpha-to-coverage and screen-door transparency. Because hashed alpha testing addresses alpha test aliasing by introducing stable sampling, it has implications in other domains where increased sample stability is desirable. We show how our hashed sampling might apply to other stochastic effects.

Index Terms—anisotropy, alpha map, alpha test, hash, hashed alpha test, stable shading, stochastic sampling.

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 [2] and order-independent transparency [3], 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 [4] 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.

But alpha testing introduces artifacts. Its binary queries alias on alpha boundaries, where $\alpha \approx \alpha_\tau$. Because this occurs in texture space, geometric antialiasing is ineffective and only postprocess antialiasing addresses these artifacts (e.g., Karis [5], Lottes [6], and Yang et al. [7]). 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 [8]). Some scene-specific tuning and restrictions on content creation reduce the problem, but none completely solve it.

To solve both problems, 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 propose a hashed alpha test with similar properties but stable behavior, i.e., using:

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

Hashing techniques have many graphics applications, ranging from packing sparse data [9] to visualizing big data [10]. However, we present the first application of hash functions that enables controllable stability of stochastic samples. In particular, this paper makes the following contributions:

- we introduce the idea of *stochastic alpha testing*, which simplifies stochastic transparency [11] but maintains the expected value of geometric coverage at all viewing distances;
- we show *hashed alpha testing* provides similar benefits and, with well-selected hash inputs, provides spatial and temporal stability comparable to traditional alpha testing;
- we introduce an anisotropic variant of hashed alpha testing that retains these properties even for surfaces viewed at grazing angles; and
- we demonstrate how to robustly incorporate hashed alpha testing into modern rendering engines, including those using temporal antialiasing, multisample alpha-to-coverage, or screen-door transparency.

We also provide insight into how our new stable, hash-based sampling scheme might apply in other stochastic sampling contexts where stable sampling may be desirable.

2 Why Does Geometry Disappear?

Disappearing alpha-tested geometry is poorly covered in academic literature. However, game developers repeatedly

- *C. Wyman is with NVIDIA Corporation in Redmond, WA. E-mail: chris.wyman@acm.org*
- *M. McGuire is with NVIDIA Corporation and Williams College.*

Manuscript received ???; revised ???.

(a) Traditonal alpha test (b) Alpha-to-coverage (c) Hashed alpha test (d) Hashed alpha-to-coverage

Fig. 1: *Hashed alpha testing* avoids disappearing alpha-mapped geometry and reduces correlations in multisample alpha-to-coverage. We compare a bearded man rendered with alpha testing, alpha-to-coverage, hashed alpha testing, and hashed alpha-to-coverage. Insets show the same geometry from further away, enlarged to better depict variations.

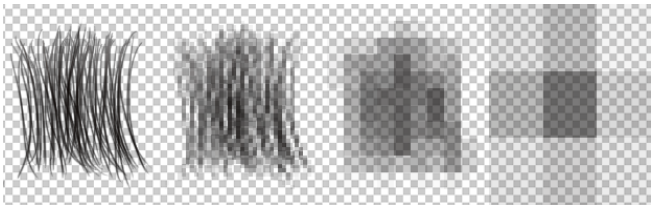


Fig. 2: Coarse texels average alpha over finer mipmapped texels. The hair in Figure 1 has an average alpha $\alpha_{\text{avg}} = 0.32$, partly due to artist-added padding. Accessing coarse mip levels for alpha thresholding gives many samples close to α_{avg}, causing most fragments to fail the alpha test.

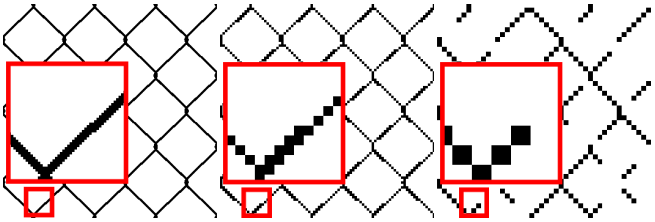
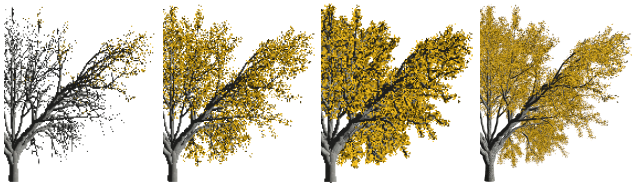


Fig. 3: When nearby, each metal wire in this fence is 5 pixels wide (left). With distance, alpha testing discretely erodes pixel coverage along edges so the wire thins to roughly one pixel (center). Further away wires no longer cover even one pixel, disappearing entirely (right).

encounter this issue. Castano [8] 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), a single texel averages alpha, α_{avg}, over the base texture. Many textures contain $\alpha_{\text{avg}} \ll \alpha_\tau$, 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.



(a) (b) (c) (d)

Fig. 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). Using conservative raster with hashed alpha testing (c) ensures full coverage, but the alpha threshold α_τ must be modified based on sub-pixel coverage to avoid the oversampled billboards shown here.

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 increasing distance (see Figure 3). At some point, 1-pixel wide geometry erodes to 0-pixel wide geometry and disappears.

Coarse raster grid. With enough distance, even proxy billboard geometry becomes sub-pixel. In this case, a relatively coarse rasterization grid (i.e., sampling at pixel centers) poorly captures the geometry. Some billboards may not appear, causing an apparent loss of coverage (see Figure 4). Our work does 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. We expect a consistent percentage of pixels passing the alpha test, i.e.:

$$a_0/A_0 \approx a_i/A_i. \quad (1)$$

One can precompute test thresholds $\alpha_\tau(i)$ that maintain this ratio for each mipmap. But content affects this threshold, so it differs between textures and even within a mip level (i.e., we 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.

Adjust stored α values per mipmap level. Castano [8] proposes a similar, mathematically equivalent, approach. Rather than requiring variable $\alpha_\tau(i)$ that complicates shader code, the alpha values in each mipmap level are premultiplied by $0.5/\alpha_\tau(i)$ at asset creation time. The allows use of a fixed $\alpha_\tau = 0.5$ at run time. Otherwise, this has identical properties to defining variable $\alpha_\tau(i)$.

Adjust stored α values per texel. Castano [8] adjusts alpha for each mipmap level. Separate per-texel adjustments (e.g., by $0.5/\alpha_\tau(i,u,v)$) are possible, but it is unclear how to compute $\alpha_\tau(i,u,v)$ or if independent texels adjustments still just dialate alpha boundaries (e.g., see Figure 11).

Always sampling α from finest mip lod. As filtering reduces alpha variance and changes threshold $\alpha_\tau(i)$, always sampling α from mip level 0 trivially avoids the problem. But this either requires two texel fetches per pixel (one for RGB, one for α) or eliminates color prefiltering (using both RGB and α from mip 0). An alternative limits the maximum mipmap lod to a user-specified $i_{\max}$, using 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 the difficulty of efficient order-independent blending. Naive alpha blending causes halos where 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 [12]. 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, one 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 [13]. Usually these dither patterns are fixed, causing correlations between layers and preventing correct multilayer compositing. Enderton et al. [11] propose selecting random pattern permutations to address this problem.

Screen-door transparency. While fairly uncommon today, screen-door transparency behaves similar to alpha-to-coverage except that dithering occurs over multiple pixels. Various mask selection techniques exist for screen-door transparency [14], but even random mask patterns lead to correlation between composited layers and repeated dither patterns visible over the screen.

(a) (b) (c) (d) (e) (f)

Fig. 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.

4 Stochastic Alpha Testing

To provide a mental framework to help understand hashed alpha testing, we first introduce the idea of *stochastic alpha testing*. Essentially, stochastic alpha testing replaces traditional alpha testing's fixed threshold ($\alpha_\tau = 0.5$) with a stochastic threshold ($\alpha_\tau = drand48()$). This replaces one sample on a regular grid (i.e., sampling $[0..1)$ at 0.5) with one uniform random sample.

A stochastic alpha test is a simplified form of stochastic transparency [11] with only one sample per pixel. While this seems trivial, we observe that replacing a fixed alpha threshold with a random one solves our 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. It introduces significant noise, causing continuous twinkle (see video). Reusing random seeds between frames and using stratification helps stabilize noise for static geometry (see Laine and Karras [15] and Wyman [16]). But whenever motion occurs the high-frequency noise reappears. Supersampling helps, but using just one sample per pixel is a major appeal of alpha testing, given it works in forward and deferred shading, without MSAA, and even on low-end hardware.

5 Hashed Alpha Testing

In *hashed alpha testing* we target visual quality equivalent to stochastic alpha testing with temporal stability equivalent to traditional alpha testing.

Instead of stochastic sampling, we propose using a hash function to generate alpha thresholds. This is not particularly surprising, as hash functions have often formed the basis for common pseudo-random number generators [17]. Our contribution is recognizing that by careful manipulation of hash function inputs, rather than hiding them inside a pseudorandom number generator, we can better control the stability of our random sample.

Appropriate hash functions include those that generate outputs uniformly distributed in $[0..1)$, allowing direct substitution for uniform random number generators. To obtain well distributed noise with spatial and temporal stability, we sought to control our hash function inputs to achieve the following properties:

- noise anchored to surface, to avoid appearance of swimming;
- no sample correlations between overlapping alpha-mapped layers; and
- discretization of α_τ at roughly pixel scale, so subpixel 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 [18]:

```
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. To hash 3D coordinates, a hash $f : \mathbb{R}^3 \to [0..1)$ may provide more control. We found repeatedly applying our 2D hash worked just as well:

```
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 threshold α_τ. It is vital to use coordinates consistent over an entire aggregate surface (e.g., a tree) rather than a part of the object (e.g., each leaf).

5.3 Avoiding Correlations Between Layers

For overlapping alpha-mapped surfaces, reusing 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 α_τ thresholds between frames, as this 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, allowing reuse of α_τ for subpixel 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 For Screen-Space Translations in X and Y

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

```
// Find the derivatives of our object-space 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 (see Section 7.2.1), using noise with subpixel scale can allow for temporal averaging.

5.4.2 Stability For Screen-Space Translations in Z

That approach gives stable noise under small vertical and horizontal translations. But moving along the camera's z-axis continuously changes derivatives `dFdx()` and `dFdy()`, thus changing hash inputs. 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 );
```

```
// 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 linearly interpolate 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 ατ = (x < (1-a)) ?
               ((x < a) ? cases.x : cases.y) :
               cases.z;

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

Listing 1: Code for our (isotropic) hashed alpha threshold.

```
// Interpolate alpha threshold from noise at two scales
float ατ = (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, akin to the logarithmic steps in a texture's mipmap chain, 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) );
```

A trickier problem arises during interpolation. A well-designed hash function $f : \mathbb{R}^2 \to [0..1)$ produces uniformly distributed output values. Interpolating between two uniformly distributed values 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:

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

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

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

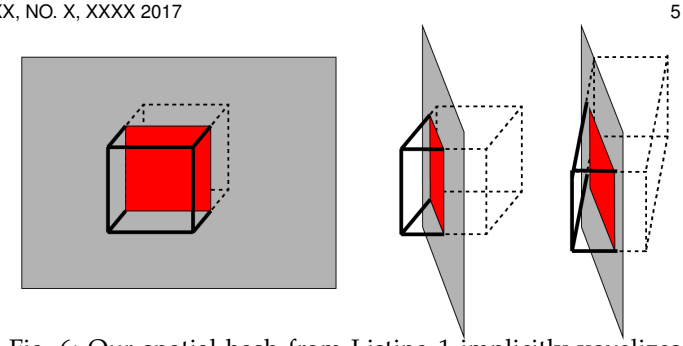


Fig. 6: Our spatial hash from Listing 1 implicitly voxelizes space (boxes). Samples in a voxel hash to the same value. Depending on viewing angle, voxels intersect triangles (red regions) either (left) isotropically or (center) anisotropically. Section 6 modifies hash inputs to independently scale voxel dimensions, (right) allowing us to resize intersection regions to remain roughly uniform in x and y.

6 Anisotropic Hashed Alpha Testing

Listing 1 provides stable hashing, but using the length of object-space derivatives provides a uniform noise scale irrespective of surface orientation. Unfortunately, this creates anisotropy if viewing surfaces obliquely, as projected noise scales differ along screen-space x- and y-axes (see Figure 6).

6.1 Difficulties Removing Anisotropy

We tested numerous methods to remove anisotropy, before realizing the impossibility given our desire for stable, uniform hash values varying on a discrete object-space grid.

Traditional anisotropic texture accesses repeatedly sample along a texture space vector. In hashed alpha testing, this breaks stability by introducing temporal variance between multiple subpixel samples. This approach fails to meet hashed alpha's goal: generation of a single pseudorandom threshold rather than producing a nicely filtered value.

Discretizing hash inputs on a coordinate frame aligned with the axes of anisotropy seems a compelling alternative. But this discretization reintroduces temporal instability as the axes of anisotropy change under most types of motion.

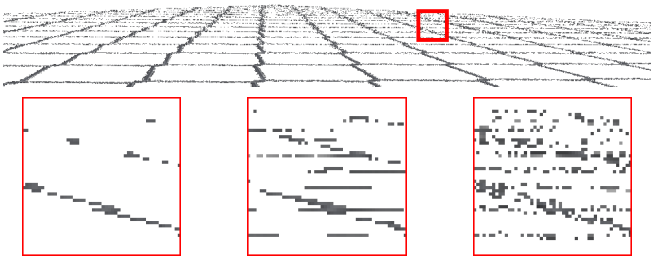
Continuing to discretize in object-space prevents a complete removal of anisotropy, as mismatches between eye- and object-space sampling grids introduce small, but variable, amounts of anisotropy over the image.

6.2 Mitigating Anisotropy

Given our inability to completely avoid anisotropy, we designed a simple algorithm that significantly reduces apparent anisotropy yet maintains a approach similar to Listing 1.

Anisotropy arises when voxels of constant hash project into screen space with different x- and y-extents (see Figure 6). With uniform discretization, we can generate pixel-scale noise only along one axis. Noise along the other axis will either be subpixel or too large, reintroducing temporal flicker or generating elongated regions of constant hash (see Figure 7).

To mitigate this problem, we select a separate discretization scale along each of the three object-space axes, rather than picking a single scale based on the maximum projected derivative. Changing these scales independently significantly reduces anisotropy:



(a) Traditional (b) Isotropic (c) Anisotropic

Fig. 7: An alpha-tested chain fence at extreme grazing angle. (a) Traditional alpha tests make chains disappear. (b) Isotropic hash samples get elongated along one axis. (c) Our anisotropic hash samples have much better shape.

```
vec3 anisoDeriv = max( abs(dFdx(objCoord.xyz)),
                       abs(dFdy(objCoord.xyz)) );
vec3 anisoScales = vec3( 1.0/(g_HashScale*ansioDeriv.x),
                         1.0/(g_HashScale*anisoDeriv.y),
                         1.0/(g_HashScale*anisoDeriv.z) );
```

But using separate scales along our axes changes the average size of our implicit voxels. The length `length(dFdx(objCoord))` is longer than the individual components of `dFdx(objCoord)`, so anisotropic noise appears smaller for a given hash scale. To maintain a consistent scale noise, we needed to scale our lengths by $\sqrt{2}$:

```
vec3 anisoScales = vec3( 0.707/(g_HashScale*ansioDeriv.x),
                         0.707/(g_HashScale*anisoDeriv.y),
                         0.707/(g_HashScale*anisoDeriv.z) );
```

We then compute log-discretized scales independently for each of our axes:

```
vec3 scaleFloor = vec3( exp2(floor(log2( anisoScales.x ))),
                        exp2(floor(log2( anisoScales.y ))),
                        exp2(floor(log2( anisoScales.z ))) );
vec3 scaleCeil = vec3( exp2(ceil(log2( anisoScales.x ))),
                       exp2(ceil(log2( anisoScales.y ))),
                       exp2(ceil(log2( anisoScales.z ))) );
```

And then we compute two hash values at opposite corners of this anisotropic box:

```
vec2 alpha = vec2(hash3D(floor(scaleFloor*objCoord.xyz)),
                  hash3D(floor(scaleCeil*objCoord.xyz)));
```

We tried computing hash values at the 8 corners of each anisotropic voxel, using trilinear interpolation with an appropriate CDF to correct the interpolated distribution. But this adds significant computation cost without improving hash sample stability. Instead, we linearly interpolate between two hash values using the following factor representing a pixel's fractional pre-discretized location in log-space:

```
vec3 fractLoc = vec3( fract(log2( anisoScale.x )),
                      fract(log2( anisoScale.y )),
                      fract(log2( anisoScale.z )) );
vec2 toCorners = vec2( length(fractLoc),
                       length(vec3(1.0f)-fractLoc) );
float lerpFactor = toCorners.x / (toCorners.x+toCorners.y);
```

Given this interpolation factor, the rest of the hashed alpha code remains the same, correcting for the CDF of interpolation between two uniform hashed samples to give a final uniform threshold. This code is shown in Listing 2.

7 Implementation Considerations

When adding a simple modification like hashed alpha testing into an existing renderer, a developer's goal is improving quality without changing the rendering workflow.

```
// Find the discretized derivatives of our coordinates
vec3 anisoDeriv = max( abs(dFdx(objCoord.xyz)),
                       abs(dFdy(objCoord.xyz)) );
vec3 anisoScales = vec3(
          0.707/(g_HashScale*ansioDeriv.x),
          0.707/(g_HashScale*anisoDeriv.y),
          0.707/(g_HashScale*anisoDeriv.z) );

// Find log-discretized noise scales
vec3 scaleFlr = vec3( exp2(floor(log2(anisoScales.x))),
                      exp2(floor(log2(anisoScales.y))),
                      exp2(floor(log2(anisoScales.z))) );
vec3 scaleCeil = vec3( exp2(ceil(log2(anisoScales.x))),
                       exp2(ceil(log2(anisoScales.y))),
                       exp2(ceil(log2(anisoScales.z))) );

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

// Factor to linearly interpolate with
vec3 fractLoc = vec3( fract(log2( anisoScale.x )),
                      fract(log2( anisoScale.y )),
                      fract(log2( anisoScale.z )) );
vec2 toCorners = vec2( length(fractLoc),
                       length(vec3(1.0f)-fractLoc) );
float lerpFactor = toCorners.x/(toCorners.x+toCorners.y);

// 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 ατ = (x < (1-a)) ?
               ((x < a) ? cases.x : cases.y) :
               cases.z;

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

Listing 2: Code for our anistropic hashed alpha threshold.

This requires working nicely with other common algorithms without introducing new artifacts. This section looks at some implementation considerations for using hashed alpha tests in existing engines.

7.1 Fading in Noise with Distance

Because of hashed alpha testing's basis in stochastic sampling, it introduces (stable) noise everywhere alpha-tested geometry is used. Developers may want to avoid introducing apparent randomness near the viewer, where existing alpha testing works fairly well, and focus on improving visual quality in the distance.

Fortunately, we can fade in hashed noise with distance. Consider the following formulation of our alpha threshold:

$$\alpha_\tau = 0.5 + \delta, \tag{3}$$

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

$$\alpha_\tau = 0.5 + \delta \cdot b(\mathrm{lod}), \tag{4}$$

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

We found that linearly ramping b still kept visible noise too close to the camera. A quadratic ramp gave better results, perhaps because apparent noise depends on solid angle, which changes with the square of distance. We used the following function to transition between traditional and hashed alpha testing:

$$b(x) = \begin{cases} 0 & : x \leq 0 \\ (x/n)^2 & : 0 < x < n \\ 1 & : x \geq n. \end{cases} \quad (5)$$

7.1.1 Fading in Noise With Anisotropic Texture Sampling

Equation 5 fails for alpha-tested surfaces viewed at a grazing angle. This occurs since anisotropic filtering repeatedly accesses finer mip levels, causing alpha geometry to disappear even at relatively low mip levels. This means the transition to a hashed alpha test needs to occur at lower mip levels than in regions sampled isotropically. 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 4, avoiding alpha maps disappearing at grazing angles.

7.2 Hashed Alpha Testing With Temporal Antialiasing

Given widespread use of temporal antialiasing (TAA) in games [5], hashed alpha testing needs to work seamlessly with temporal accumulation techniques. Since hashed testing derives from stochastic sampling, one might assume it works trivially with TAA. But changing hash inputs to induce stability explicitly creates pixel-sized noise. Naive application of TAA gives smooth-edged, pixel-sized blocks rather than accumulating stochastic coverage temporally.

We see three simple approaches to integrate TAA with hashed alpha testing: reduce the global noise scale below one pixel, use temporally independent hash samples, or temporally stratify the hash samples.

These techniques all reduce noise, but as in stochastic transparency [11], the eight jittered samples TAA typically uses is insufficient for converged results. This can introduce temporal flicker. We found temporally stratifying hash samples gave the most stable results (see Section 7.2.3).

7.2.1 Temporal Antialiasing by Reducing Noise Scale

Reducing the global noise scale `g_HashScale` to a value below 1.0 gives subpixel noise. TAA's temporal camera jitter averages this subpixel noise over multiple frames. For n temporal samples, using a `g_HashScale` of $\sqrt{1/n}$ gives the ideal number of n unique subpixel hash values.

Unfortunately, reducing noise scale below 1.0 reduces hash sample stability under motion. Until sampling sufficiently for a converged solution, this effectively trades off spatial stability for temporal stability. With the 8 TAA samples commonly used today, this approach does not provide a particularly compelling solution.

7.2.2 Temporal Antialiasing by Independent Hashing

Instead of introducing subpixel noise and trusting to TAA jitter to accumulate the results, another approach keeps the pixel-scale hash grid but chooses n different hash values over the n frames in a n-sample TAA.

A naive solution uses multiple different hash functions, using $\alpha_\tau = \text{hash}[i](\text{hashInput.xyz})$ for frame i, repeating every n frames. However for 8-sample TAA, this requires designing 8 good hash functions and dynamically selecting which to execute each frame.

In theory, separate regions of hash space are also independent. This means $\text{hash}(\text{input})$ and $\text{hash}(\text{input} + \text{offset})$ give independent random thresholds, allowing us to select $\alpha_\tau = \text{hash}(\text{hashInput.xyz} + \text{offset}[i])$ for frame i. In this case, rather than designing separate hash functions, we just need to provide a list of n unique vector offsets.

While this approach gives a stable set of n hash thresholds α_τ over any set of n frames, it still gives temporally unstable results. Since TAA effectively averages via an exponentially weighted moving average, the highest weighted threshold α_τ has an n-frame cycle. When these n samples are independent, this introduces temporal flicker.

7.2.3 Temporal Antialiasing by Temporal Stratification

Rather than using independent hash samples, we can select n dependent alpha thresholds, stratifying them to uniformly sample $[0...1)$ and distributing them temporally to reduce the flicker introduced by an exponentially weighted moving average. Given a hash sample χ, computed via the code in Listings 1 or 2, we define alpha threshold α_τ over an n-frame TAA sequence:

$$\alpha_\tau = \text{fract}\left(\chi + \frac{i}{n}\right), \quad \forall\, i \in [0...n-1] \quad (6)$$

As with the independent hash samples in Section 7.2.2, this gives spatial stability but temporally flickers as the TAA cycles between heavily weighing low and high thresholds.

An exponential moving average weighs one of our n stratified samples higher than others, but we can reorder to ensure good distribution of our two highest weighted samples. With a stratified sample set, we know half of our samples will have $\alpha_\tau < \frac{1}{2}$ and half will have $\alpha_\tau \geq \frac{1}{2}$. We should alternate so every other sample is below $\frac{1}{2}$, e.g.:

$$j = \left\lfloor \frac{i}{2} \right\rfloor + (i \bmod 2) * \frac{n}{2}$$
$$\alpha_\tau = \text{fract}\left(\chi + \frac{j}{n}\right)$$

This provided sufficient stability for us over 8 temporal samples, though for an even better, low discrepancy sample distribution we could define j using a radical-inverse sequence like the binary van der Corput sequence.

7.3 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 texture regions.

Using premultiplied alpha allows correct mipmaping and avoids this problem (e.g., see Glassner [19] for further discussion). However premultipled alpha textures store $(\alpha R, \alpha G, \alpha B, \alpha)$, so we need to divide by alpha before returning our alpha tested color (R, G, B) to maintain convergence to ground truth when increasing sample counts.

Hashed alpha testing works with non-premultiplied diffuse textures, but we frequently found that when our hash returned $\alpha_\tau < 0.5$, colors bled from transparent texels and introduced arbitrary colored halos at alpha boundaries.

8 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:

$$\alpha_\tau = \frac{0.5}{n}, \frac{1.5}{n}, ..., \frac{n-0.5}{n}. \quad (7)$$

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:

$$\alpha_\tau = \frac{\chi_1}{n}, \frac{1+\chi_2}{n}, ..., \frac{n-1+\chi_n}{n}, \quad (8)$$

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:

$$\alpha_\tau = \frac{\chi_i + ((\alpha_o + i\alpha_+) \bmod n)}{n}, \quad \forall\, i \in [0...n-1] \quad (9)$$

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.

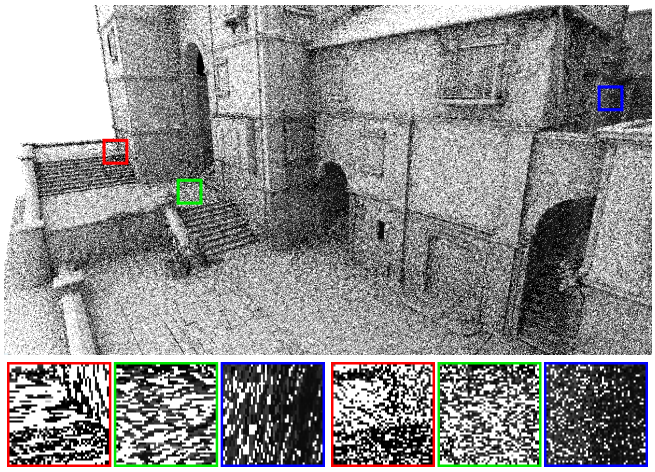
8.1 Applications to Screen Door Transparency

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.

9 Other Applications of Hashed Sampling

Until this point, we focused our description on how stable hashed samples apply to disappearing alpha-tested geometry. However, interactive applications strive for temporal stability in various domains, and stable hashing may prove useful either for directly generating samples or improving the quality of reprojection caching [20].

For example, light transport algorithms often use Monte Carlo sampling to approximate the rendering equation [21].



(a) Isotropic samples (b) Anisotropic samples

Fig. 8: Using hashed pseudorandom samples for ambient occlusion, with one visibility ray per pixel. Noise remains largely stable under motion and stays fixed to the geometry. The top image uses ansiotropic sampling (see Section 6), but insets compare (a) isotropic and (b) anisotropic sampling.

Real-time constraints can prevent the increased ray counts needed to avoid temporal noise.

A common approach defines per-pixel, fixed pseudorandom seeds, but this induces a screen-door effect visible as geometry moves relative to the screen. We instead generate samples with our stable, pixel-sized grid. This produces largely stable noise that stays fixed to the geometry, even under motion. We demonstrate this for single ray per pixel ambient occlusion (see Figure 8). This is simple enough to visually understand the hash samples, yet remains representative of more complex light transport.

For our simple test, rather than computing α_τ via Listing 2, we generate two hash samples $\chi_1, \chi_2 \in [0...1)$ with the same code, but using two different `hash3D()` functions for χ_1 and χ_2:

```
float hash3D_χ1(vec3 in)
    { return hash( vec2( hash( in.xy ), in.z ) ); }
float hash3D_χ2(vec3 in)
    { return hash( vec2( in.x, hash( in.yz ) ) ); }
```

As described in Section 7.2.2, independent samples can also be generated using different offsets, e.g.,:

```
float hash3D_χ1(vec3 in) { return hash3D( in+offset[0] ); }
float hash3D_χ2(vec3 in) { return hash3D( in+offset[1] ); }
```

We convert these hashed values into a cosine-sampled hemisphere in the standard way:

```
vec3 direction = vec3( sqrt( χ1 ) * cos( 2πχ2 ),
                       sqrt( χ1 ) * sin( 2πχ2 ),
                       sqrt( max( 0.0, 1.0 - χ1 ) ) );
```

Extending to more general light transport algorithms, which require many stable samples each frame, simply requires using additional $\mathrm{offset}[i]$ values. Alternatively, a hash returning an integer could seed a more traditional pseudorandom number generator.

10 Stochastic Transparency Comparison

Stochastic alpha testing essentially simplifies stochastic transparency to one sample per pixel. Hence, when using

TABLE 1: Cost comparisons for traditional, hashed, and stochastic alpha tests at 1920×1080 on a GeForce GTX 1080. Performance of isotopic and anisotropic hashing were equal, given our measurement error; the single column in our table represents both.

Scene	# tris	Trad. α test	Hashed α test	Stoch. α 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

(a) Traditional (b) Hashed (c) Castano [8]

Fig. 10: A comparison between traditional and hashed alpha testing with Castano's precomputed per-mip modifications to texture alpha, which tends to enlarge thin geometry.

(a) Traditional (b) Hashed (c) Fade in hash

Fig. 12: With alpha testing, tree leaves disappear with distance. Hashed alpha testing keeps these leaves, but nearby leaves have noisy edges and, due to alpha of 0.99, some internal noise. Fading in the hash contribution, per Section 7.1, keeps distant leaves without nearby noise.

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 7, alpha-to-coverage is stochastic transparency with regular instead of random samples. Using randomized thresholds, as in Equation 8, 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.

11 Results

We prototyped our hashed and stochastic alpha test in an OpenGL-based renderer using the Falcor prototyping library [22]. 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.

Table 1 shows performance relative to traditional alpha testing, rendered at 1920×1080 and using one shader for all surfaces, transparent and opaque. Our added overhead for hashed alpha testing is all computation, without additional texture or global memory accesses. Anistopic hashed alpha testing increases costs, perhaps 10–20% over the isotropic variant, but with our limited timing precision and our test's small overhead the timing runs were identical for both variants. 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 9 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.

In Figures 10 and 11, we compare hashed alpha against the widely used approach of Castano [8], which computes a per-mipmap alpha threshold and bakes this into the texture as a preprocess. This approach dilates distant geometry to keep it visible, giving an overly dense appearance on the thin leaves in Figure 10.

Figure 11 compares hashed alpha testing, $8\times$ MSAA using hashed alpha-to-coverage, and both hashed techniques with temporal antialiasing. Interestingly using hashed alpha testing with TAA gives results largely similar to alpha-to-coverage, and temporally antialiased alpha-to-coverage is nearly indistinguishable from the ground truth.

Figure 12 shows a more complex example where hashed noise may be undesirable nearby and the fade-in from Section 7.1 maintains crisp edges near the viewer.

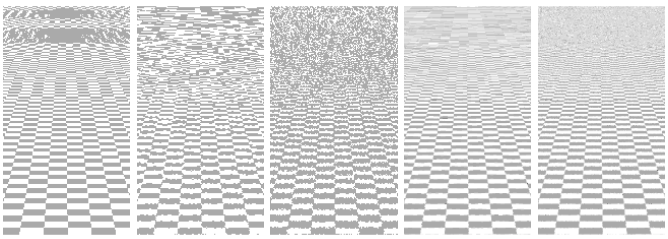
Fig. 9: Four plant models whose alpha-mapped polygons disappear with distance. This also happens when rendering at lower resolution (left four columns), which allows for better comparisons to our supersampled ground truth (right column). Notice how alpha testing loses alpha-mapped details and alpha-to-coverage 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.

(a) Ground truth (b) Traditional (c) Castano [8] (d) Hashed (e) Hashed, TAA (f) Hash A2C (g) A2C w/TAA

Fig. 11: Comparing distant renderings of the bearded man using various techniques, including a ground truth sorted blend, traditional alpha testing, Castano's tweaks to texture alpha values, as well as hashed alpha testing and hashed alpha-to-coverage (both with and without temporal antialiasing).

(a) (b) (c)

Fig. 13: Difference images from before and after a sub-pixel translation along the x-axis using a (a) traditional alpha test, (b) hashed alpha test, and (c) stochastic alpha test. Difference images inverted for better visibility.



(a) (b) (c) (d) (e)

Fig. 14: A synthetic example, with a checkerboard texture where half the texels have $\alpha = 0$ and half have $\alpha = 1$. We compare (a) traditional alpha testing, (b) isotropic hashed alpha, (c) anisotropic hashed alpha, (d) $8\times$ isotropic alpha-to-coverage, and (e) $8\times$ anisotropic alpha-to-coverage.

Figure 13 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.

Figure 14 shows a synthetic example of a planar checkerboard texture containing half opaque and half transparent texels. In this case, alpha testing does not disappear with distance but aliases. Hashed alpha testing replaces this aliasing with noise, and using MSAA-based alpha-to-coverage converges to a desired uniform gray in the distance.

(a) Traditional alpha test (b) Hashed alpha test

Fig. 15: Two foveated images, as per Patney et al. [23]. In both, the viewer looks towards the curtains (yellow circle). Regions outside the circle use progressively lower quality. Low resolution shading uses coarser mipmap levels, causing traditional alpha tests to fail. With hashed alpha testing, the foliage maintains its aggregate appearance.

Beyond use for distant or low-resolution alpha-mapped geometry, other applications exist for hashed alpha testing. In head-mounted displays for virtual reality, rendering at full resolution in the user's periphery is wasteful, especially as display resolutions increase. Instead, foveated rendering [24] uses at lower resolution away from a user's gaze. Patney et al. [23] suggest prefiltering all rendering terms, but they were unable 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 15). With temporal antialiasing, hashed alpha testing enables use of alpha mapped geometry in foveated renderers.

Figure 16 shows hashed alpha testing in a more complex environment. Results are more subtle as scene scale is small enough to minimize the pixels accessing coarse mipmaps.

12 Conclusions

We introduced the idea of *stochastic* or *hashed alpha testing* to address the problem of alpha-mapped geometry disappearing with distance. Stochastic alpha testing uses a random α_τ rather than a fixed threshold of 0.5. To address the temporal noise stochasm introduces, we proposed a procedural hash to provide a stable noisy threshold.

We obtained stable, pixel scale noise by hashing on discretized object-space coordinates at two scales. We showed

(a) Traditional alpha test (b) Hashed alpha test

Fig. 16: Disappearing geometry in San Miguel.

how to ensure the interpolated hash value maintains a uniform distribution, how to maintain sample quality in the presence of anisotropy, and how hashed alpha testing works with temporal antialiasing. We demonstrate temporal stability both in Figure 13 and the accompanying video. And we suggested how this stable sampling scheme could extend to other stochastic light transport effects. We provided inline code to replicate our hashed test.

Thinking about alpha-to-coverage and screen door transparency in 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, using more sophisticated hash inputs than object-space coordinates may generalize over a larger variety of highly instanced scenes. Both areas seem fruitful for future work.

Acknowledgments

Our work evolved from a larger research project, so many researchers contributed directly and indirectly. Thanks to Pete Shirley for keeping our hash uniform by deriving the cdf in Equation 2, Cyril Crassin for suggesting alpha maps as a simplified domain worth studying, Anton Kaplanyan for discussing stable noise, Anjul Patney for trying hashed alpha in foveated VR, and Dave Luebke, Aaron Lefohn, and Marco Salvi for additional discussions. Also, thanks to the insightful I3D and TVCG reviewers and numerous helpful comments from the game developer community on Twitter.

References

[1] C. Wyman and M. McGuire, "Hashed alpha testing," in *Symposium on Interactive 3D Graphics and Games*, February 2017, pp. 7:1–7:9.
[2] T. Porter and T. Duff, "Compositing digital images," in *Proceedings of SIGGRAPH*, 1984, pp. 253–259.
[3] C. Wyman, "Exploring and expanding the continuum of OIT algorithms," in *High Performance Graphics*, June 2016, pp. 1–11.
[4] T. Saito and T. Takahashi, "Comprehensible rendering of 3-d shapes," *Proceedings of SIGGRAPH*, pp. 197–206, 1990.
[5] B. Karis, "High-quality temporal supersampling," in *SIGGRAPH Course Notes: Advances in Real-Time Rendering in Games.*, 2014.
[6] T. Lottes, "FXAA," NVIDIA, http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf, Tech. Rep., 2009.
[7] L. Yang, D. Nehab, P. V. Sander, P. Sitthi-amorn, J. Lawrence, and H. Hoppe, "Amortized supersampling," *ACM Transactions on Graphics*, vol. 28, no. 5, p. 135, 2009.
[8] I. Castano, "Computing alpha mipmaps," http://the-witness.net/news/2010/09/computing-alpha-mipmaps/, 2010.
[9] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 579–588, Jul. 2006.
[10] C. A. L. Pahins, S. A. Stephens, C. Scheidegger, and J. L. D. Comba, "Hashedcubes: Simple, low memory, real-time visual exploration of big data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 671–680, Jan 2017.
[11] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency," in *Symposium on Interactive 3D Graphics and Games*, 2010, pp. 157–164.
[12] J. Moore and D. Jefferies, "Rendering technology at Black Rock Studios," in *SIGGRAPH Course Notes: Advances in Real-Time Rendering in Games.*, 2009.
[13] A. Kharlamov, I. Cantlay, and Y. Stepanenko, *GPU Gems 3*. Addison-Wesley, 2008, ch. Next-Generation SpeedTree Rendering, pp. 69–92.
[14] J. Mulder, F. Groen, and J. van Wijk, "Pixel masks for screen-door transparency," in *Proceedings of Visualization*, 1998, pp. 351–358.
[15] S. Laine and T. Karras, "Stratified sampling for stochastic transparency," *Computer Graphics Forum*, vol. 30, no. 4, pp. 1197–1204, 2011.
[16] C. Wyman, "Stochastic layered alpha blending," in *ACM SIGGRAPH Talks*, 2016.
[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1992.
[18] M. McGuire, *The Graphics Codex*, 2nd ed. Casual Effects, 2016.
[19] A. Glassner, "Interpreting alpha," *Journal of Computer Graphics Techniques*, vol. 4, no. 2, pp. 30–44, 2015.
[20] D. Nehab, P. V. Sander, J. Lawrence, N. Tatarchuk, and J. R. Isidoro, "Accelerating real-time shading with reverse reprojection caching," in *Proceedings of Graphics Hardware*, 2007, pp. 25–35.
[21] J. Kajiya, "The rendering equation," in *Proceedings of SIGGRAPH*, 1986, pp. 143–150.
[22] N. Benty, K.-H. Yao, A. S. Kaplanyan, C. Lavelle, and C. Wyman, "Falcor real-time rendering framework," https://github.com/NVIDIA/Falcor, 2016.
[23] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N. Benty, D. Luebke, and A. Lefohn, "Towards foveated rendering for gaze-tracked virtual reality," *ACM Transactions on Graphics*, vol. 35, no. 6, pp. 179:1–12, 2016.
[24] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder, "Foveated 3d graphics," *ACM Transactions on Graphics*, vol. 31, no. 6, pp. 164:1–10, 2012.

Chris Wyman is a Principal Research Scientist in NVIDIA's real-time rendering research group located in Redmond, WA. Before moving to Redmond, he was a Visiting Professor at NVIDIA and an Associate Professor of Computer Science at the University of Iowa. He earned a PhD in computer science from the University of Utah and a BS in math and computer science from the University of Minnesota. Recent research interests include efficient shadows, antialiasing, and participating media, but his interests span the range of real-time rendering problems from lighting to global illumination, materials to color spaces, and optimization to hardware improvements.

Morgan McGuire is a professor of Computer Science at Williams College and a researcher at NVIDIA. He is the author or coauthor of *The Graphics Codex*, *Computer Graphics: Principles and Practice, 3rd Edition*, and *Creating Games*. He received his degrees at Brown University and M.I.T.