Hashed Alpha Testing

Chris Wyman

GDC 2017
Why Alpha Test?

• Alpha testing has advantages over alpha blend
  • Order independent, cheap, for forward or deferred
  • Extends to MSAA, via alpha-to-coverage
But... Problem

• Alpha testing has advantages over alpha blend
  • Order independent, cheap, for forward or deferred
  • Extends to MSAA, via alpha-to-coverage

• But alpha-tested geom can disappear w/ distance
But... Problem

- Alpha testing has advantages over alpha blend
  - Order independent, cheap, for forward or deferred
  - Extends to MSAA, via alpha-to-coverage

- But alpha-tested geom can disappear w/ distance
  - Why? Cannot prefilter binary queries
New Solution: Hashed Alpha

- Use stochastic sampling to avoid this problem
New Solution: Hashed Alpha

- Use stochastic sampling to avoid this problem
  - Basic idea is replace standard test:
    
    ```c
    if (color.a < \alpha_T) discard;
    ```
  - With this stochastic test:
    ```c
    if (color.a < drand48()) discard;
    ```
New Solution: Hashed Alpha

- Use stochastic sampling to avoid this problem
  - Basic idea is replace standard test:
    ```cpp
    if ( color.a < \alpha \tau ) discard;
    ```
  - With this stochastic test:
    ```cpp
    if ( color.a < drand48() ) discard;
    ```
  - But this flickers like crazy
  - Want temporal stability, esp. under slight motion
  - Use stable, procedural noise:
    ```cpp
    if ( color.a < hash( . . . ) ) discard;
    ```
What Does this Look Like?

Alpha Test  Hashed Alpha  Ground Truth
Talk Takeaways

• *Stochasm* addresses problems with alpha testing

  • Converges to ground truth (OIT) with enough random samples
Talk Takeaways

• *Stochasm* addresses problems with alpha testing
  • Converges to ground truth (OIT) with enough random samples

• *Hashing* can give noise stable over time
Talk Takeaways

- **Stochasm** addresses problems with alpha testing
  - Converges to ground truth (OIT) with enough random samples

- **Hashing** can give noise stable over time

- By *constraining* hash inputs:
  - Control noise behavior
  - Ensure samples remain largely stable between frames
Talk Takeaways

• *Stochasm* addresses problems with alpha testing
  - Converges to ground truth (OIT) with enough random samples

• *Hashing* can give noise stable over time

• By *constraining* hash inputs:
  - Control noise behavior
  - Ensure samples remain largely stable between frames

• Also applies to alpha-to-coverage, screen-door transparency, etc.
Do Stochastic Alpha Thresholds Work?
Stochastic Alpha Thresholds Work

Alpha test
1 sample
4 samples
16 samples
64 samples
OIT with alpha blend
Stochastic Alpha Thresholds Work

Traditional Alpha

if ( color.a < 1/2 ) discard;
1 sample, selected uniformly on interval [0..1]

Alpha-to-Coverage

if ( color.a < 1/8 ) discard;
if ( color.a < 3/8 ) discard;
if ( color.a < 5/8 ) discard;
if ( color.a < 7/8 ) discard;
N samples, selected uniformly on interval [0..1]

Stochastic Alpha (aka stochastic transparency)

if ( color.a < drand48() ) discard;
if ( color.a < drand48() ) discard;
if ( color.a < drand48() ) discard;
if ( color.a < drand48() ) discard;
N samples, selected randomly on interval [0..1]
Stochastic Alpha Thresholds Work

Traditional Alpha

\[
\text{if ( color.a < 1/2 ) discard;}
\]

1 sample, selected uniformly on interval [0..1]

Alpha-to-Coverage

\[
\begin{align*}
\text{if ( color.a < 1/8 ) discard;} \\
\text{if ( color.a < 3/8 ) discard;} \\
\text{if ( color.a < 5/8 ) discard;} \\
\text{if ( color.a < 7/8 ) discard;}
\end{align*}
\]

N samples, selected uniformly on interval [0..1]

Stochastic Alpha (aka stochastic transparency)

\[
\begin{align*}
\text{if ( color.a < drand48() ) discard;} \\
\text{if ( color.a < drand48() ) discard;} \\
\text{if ( color.a < drand48() ) discard;} \\
\text{if ( color.a < drand48() ) discard;}
\end{align*}
\]

N samples, selected randomly on interval [0..1]

But stochastic algorithms change each frame, causing severe temporal flickering!
Why Hashing?

• Hashing long known as a way of ‘randomizing’
  
  • See “Numerical Recipes” for an example PRNG using DES encryption
Why Hashing?

• Hashing long known as a way of ‘randomizing’
  • See “Numerical Recipes” for an example PRNG using DES encryption

• Good hash function properties:
  • **Determinism** given fixed input i.e., gives same value each frame
  • **Defined range** of outputs i.e., in range [0...1)
  • **Uniformity** over output range i.e., uniform outputs in range [0...1)
What Does This Mean?

• Consider the following, with good hash function hash( ):

  float hashSample = hash( myPosition );

• Gives noisy hashSample like RNG per sample, but stays fixed between frames
What Does This Mean?

• Consider the following, with good hash function hash( ):

  float hashSample = hash( myPosition );

• Gives noisy hashSample like RNG per sample, but stays fixed between frames

• Important caveat! Tiny camera or object motions change hash sample, as

  hash( myPosition ) ≠ hash( myPosition + Δ )

• So these give different random values → flicker under motion
What Does “Stability” Look Like?

Unstable

Stable
Achieving Hash Stability

- **Key:** Discretize hash inputs in appropriate coordinate frame
  - For small $\Delta$:
    
    $$\text{hash}( \text{floor}( \text{myPosition} ) ) = \text{hash}( \text{floor}( \text{myPosition} + \Delta ) )$$
  - Tweaking this, allows us to control the *scale* of the stable noise, e.g.:
    
    $$\text{hash}( \text{floor}( \text{myPosition} / \text{scale} ) * \text{scale} )$$
What Does A Hash Scale Look Like?

1x1 pixel scale

3x3 pixel scale

9x9 pixel scale
Key: Coordinate Choice For Hash Input

• Need to discretize in appropriate coordinates

• Same geometry should yield same hash, under:
  • Camera translation or rotation
  • Object translation or rotation
  • Ideally object skinning and deformation
## Key: Coordinate Choice For Hash Input

<table>
<thead>
<tr>
<th>Coordinate Type</th>
<th>Camera Translate</th>
<th>Camera Rotate</th>
<th>Object Translate</th>
<th>Object Rotate</th>
<th>Object Skinning</th>
<th>Object Deform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen-Space Coords</td>
<td>✓</td>
<td>*</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Eye-Space Coords</td>
<td>✓</td>
<td>*</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Norm. Device Coords</td>
<td>✓</td>
<td>*</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Texture-Space Coords</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>World-Space Coords</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Object-Space Coords</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

*’s work for deformation (and skinning) assuming hashing of pre-deformed coordinates

* = being somewhat generous
### Key: Coordinate Choice For Hash Input

<table>
<thead>
<tr>
<th></th>
<th>Camera Translate</th>
<th>Camera Rotate</th>
<th>Object Translate</th>
<th>Object Rotate</th>
<th>Object Skinning</th>
<th>Object Deform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen-Space Coords</td>
<td>✓</td>
<td>*</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Eye-Space Coords</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Norm. Device Coords</td>
<td>✓</td>
<td>*</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Texture-Space Coords</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>World-Space Coords</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Object-Space Coords</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Clear choices:

- Screen-Space Coords
- Eye-Space Coords
- Norm. Device Coords

- Texture-Space Coords
- World-Space Coords
- Object-Space Coords

?’s work for deformation (and skinning) assuming hashing of pre-deformed coordinates

* = being somewhat generous
Key: Coordinate Choice For Hash Input

<table>
<thead>
<tr>
<th></th>
<th>Camera Translate</th>
<th>Camera Rotate</th>
<th>Object Translate</th>
<th>Object Rotate</th>
<th>Object Skinning</th>
<th>Object Deform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen-Space Coords</td>
<td>✅ *</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Eye-Space Coords</td>
<td>✅ *</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Norm. Device Coords</td>
<td>✅ *</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Texture-Space Coords</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>World-Space Coords</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Object-Space Coords</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

*We selected object coordinates; can discuss why, offline*

’’s work for deformation (and skinning) assuming hashing of pre-deformed coordinates

* = being somewhat generous
// 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 αₜ = (x < (1-a)) ? ((x < a) ? cases.x : cases.y) : cases.z;

// Avoids αₜ == 0. Could also do αₜ=1-αₜ
αₜ = clamp( αₜ, 1.0e-6, 1.0 );
// 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 αₜ = (x < (1-a)) ?
    ((x < a) ? cases.x : cases.y) :
    cases.z;

// Avoids αₜ == 0. Could also do αₜ = 1-αₜ
αₜ = clamp( αₜ, 1.0e-6, 1.0 );
Hashed Alpha Test Code

// 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);

// Compute ideal scale to get 1-pixel sized noise
float idealSize = 1.0/pixScale;

// 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 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 );

Compute ideal scale to get 1-pixel sized noise

Use 2 scales, one smaller & one larger than desired scale; akin to mipmapping
Hashed Alpha Test Code

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

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

Compute ideal scale to get 1-pixel sized noise

Use 2 scales, one smaller & one larger than desired scale; akin to mipmapping

Compute hashed noise samples at 2 discrete scales
Hashed Alpha Test Code

// 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 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 );

Compute ideal scale to get 1-pixel sized noise

Use 2 scales, one smaller & one larger than desired scale; akin to mipmapping

Compute hashed noise samples at 2 discrete scales

Linearly interpolate between noise scales
Hashed Alpha Test Code

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

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

Compute ideal scale to get 1-pixel sized noise

Use 2 scales, one smaller & one larger than desired scale; akin to mipmapping

Compute hashed noise samples at 2 discrete scales

Linearly interpolate between noise scales

Interpolating between two uniform distributions does not give a new uniform distribution

This math corrects for this non-uniformity
Hashed Alpha Test Code

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

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

Compute ideal scale to get 1-pixel sized noise

Use 2 scales, one smaller & one larger than desired scale; akin to mipmapping

Compute hashed noise samples at 2 discrete scales

Linearly interpolate between noise scales

Interpolating between two uniform distributions does not give a new uniform distribution

This math corrects for this non-uniformity

Clamp; alpha threshold of 0 makes no sense
Hashed Alpha Test Code

```c
// 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 \( \alpha_T \) = (x < (1-a)) ?
  ((x < a) ? cases.x : cases.y) :
  cases.z;

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

Specific hash not important, if uniform and takes \( \mathbb{R}^3 \rightarrow [0...1) \)

We use:

```c
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 ) ) )
                );
}

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

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

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

Alpha test $\alpha_t = 0.5$

Hashed alpha test

Ground truth OIT

0.06 ms @ $1024^2$

0.08 ms @ $1024^2$

1.3 ms @ $1024^2$

Performances not optimized; only demonstrative of relative costs
Results

Adjust alpha [Castaño 2010]

Hashed alpha test

Ground truth OIT

Performances not optimized; only demonstrative of relative costs
But Not Just Alpha!
Hashed Sampling Has Other Uses

Shows there’s a problem with anisotropic hashing!
Hashed Sampling Has Other Uses

Can be addressed, mostly, by updating hash inputs
Hashing At Grazing Angles

Alpha test
Hashed alpha
Anisotropic hashed alpha
Results

Alpha test $\alpha_t = 0.5$

Hashed alpha test

Ground truth OIT
Results

Alpha test $\alpha_t = 0.5$

Hashed alpha test

Ground truth OIT

Might ask:

Do I want noise nearby?

(Alpha test is OK here)
Results

Adjust alpha [Castaño 2010]

Hashed alpha test
(Faded in with LOD)

Ground truth OIT

www.gameworks.nvidia.com
Results

Per-LOD threshold adjust
(Manual, “artist” driven)

Hashed alpha test
(Faded in with LOD)

Ground truth OIT
Helps With MSAA and A2C

8x alpha-to-coverage
8x hashed alpha-to-coverage
Ground truth OIT
Temporal Antialiasing

No Temporal Antialiasing
Alpha  Hashed Alpha  Ground Truth

With Temporal Antialiasing
Alpha  Hashed Alpha  Ground Truth
Temporal Antialiasing

- Might think, “based on stochastic sampling; of course TAA works well!”
  - But hashed alpha designed for *stability* under tiny camera motions, e.g. TAA jitter
Temporal Antialiasing

• Might think, “based on stochastic sampling; of course TAA works well!”
  • But hashed alpha designed for *stability* under tiny camera motions, e.g. TAA jitter

• A couple approaches:
  • Reduce global noise scale to < 1 pixel
    • TAA integrates sub pixel noise samples
  • Jitter offset in hash space;
    • Hash on objPos+offset[i] rather than objPos for relatively large, uncorrelated offset[i]
  • Jitter the alpha threshold
    • Compute $\alpha_T = \text{hash}(\text{objPos})$, use thresholds $\text{fract}(\alpha_T + i/N)$ for i in [0...N)
Summary

• Introduce new approach for alpha testing:
  • Alpha threshold cheaply & procedurally selected via a hash function
  • Gives stable noise; roughly as stable as traditional alpha test
Summary

• Introduce new approach for alpha testing:
  • Alpha threshold cheaply & procedurally selected via a hash function
  • Gives stable noise; roughly as stable as traditional alpha test

• Still use one alpha test per pixel, allowing:
  • Use in both deferred and forward pipelines
  • Run on older hardware; no new (or recent) features required
Summary

• Introduce new approach for alpha testing:
  • Alpha threshold cheaply & procedurally selected via a hash function
  • Gives stable noise; roughly as stable as traditional alpha test

• Still use one alpha test per pixel, allowing:
  • Use in both deferred and forward pipelines
  • Run on older hardware; no new (or recent) features required

• Requires nothing in asset pipeline
  • Directly uses mip-chain’s alpha channels representing pre-filtered visibility
For Questions:

Email: cwyman@nvidia.com  Twitter: @_cwyman_


Thanks!