The Importance of Being Noisy: Fast, High Quality Noise

Natalya Tatarchuk
3D Application Research Group
AMD Graphics Products Group

Outline

- Introduction: procedural techniques and noise
  - Properties of ideal noise primitive
- Lattice Noise Types
- Noise Summation Techniques
- Reducing artifacts
  - General strategies
  - Antialiasing
- Snow accumulation and terrain generation
- Conclusion

Outline

- Introduction: procedural techniques and noise
  - Properties of ideal noise primitive
  - Noise in real-time using Direct3D API
- Lattice Noise Types
- Noise Summation Techniques
- Reducing artifacts
  - General strategies
  - Antialiasing
- Snow accumulation and terrain generation
- Conclusion

The Importance of Being Noisy

- Almost all procedural generation uses some form of noise
  - If image is food, then noise is salt – adds distinct "flavor"
- Break the monotony of patterns!!
  - Natural scenes and textures
  - Terrain / Clouds / fire / marble / wood / fluids
- Noise is often used for not-so-obvious textures to vary the resulting image
  - Even for such structured textures as bricks, we often add noise to make the patterns less distinguishable
    - Ex: ToyShop brick walls and cobblestones

Why Do We Care About Procedural Generation?

- Recent and upcoming games display giant, rich, complex worlds
- Varied art assets (images and geometry) are difficult and time-consuming to generate
  - Procedural generation allows creation of many such assets with subtle tweaks of parameters
- Memory-limited systems can benefit greatly from procedural texturing
  - Smaller distribution size
  - Lots of variation
  - No memory/bandwidth requirements

Images from the upcoming Crytek game using CryEngine2

Why Do We Care About Noise?

- Basic building block of procedural generation
  - “A Function that Launched a Thousand Textures”
  - Flexible and powerful toolbox of techniques
- Cinematic rendering
  - For offline, quality is more important than computation cost
- Artifacts can be avoided by applying noise many times
  - “The Perfect Storm”: the ocean waves procedural shaders combined 200 procedures invoking Perlin noise

The Perfect Storm

Ideal Properties of Noise

- Not every noise is equal!
- Noise *appears* random, but it isn't really
  - We want the noise function to be repeatable
  - Always yield the same value for a given input point
  - *Pseudorandom* with no obvious periodicity
- Noise is a mapping from $R_n \rightarrow R$
  - N-dimensional input returns a real value
  - Animation .. → Textures … → Time-varying solid objects
- Known range [-1; 1] and average (0)
- Noise is *band-limited*
  - Most energy concentrated in a small part of the frequency spectrum
    - Noise spends most of its time in [0.2; 0.6] range
  - Assemble a set of *frequency* and *amplitude* scaled noise functions to build complex functions

Other Desired Noise Properties

- Inexpensive to evaluate
- Visually isotropic: directionally insensitive signal
  - Viewer shouldn't discern patterns or orientation
  - Translation / rotation invariant
- Well-behaved derivatives
  - Should be easy to compute
  - Should be at least 2nd order continuous
  - A lot of implementations require approximation by evaluating neighboring noise samples
    - That can be expensive
  - Analytical evaluation of derivative is desirable
    - Very useful for normal perturbation computations and effects which use the derivative of noise, not its value

Noise Evaluation in Real-Time

- Solid noise: can use object space coordinates to sample
  - Better quality, avoids UV problems (seams, cracks, etc)
- Definitely a huge win with any VS / GS computations
  - Can use solid noise
  - Higher quality and higher precision
  - No concerns about aliasing in this domain
- Highly beneficial for material computation (PS)
  - Unlimited resolution and ability to scale frequencies as necessary depending on closeness
  - Must pay attention to aliasing and adjust

Solid Noise: Rendering to Volume: D3D9

- In D3D9 we could emulate volume texture rendering by rendering to individual 2D slices
- However, filtering had to be done in the shader
  - Expensive!
  - Example: Fluid flow in 3D

[Sander et al 04] on ATI Radeon X800

Solid Noise in D3D10

- We can render and sample to / from 3D textures directly in D3D10
  - Set the volume texture as the render target
  - Draw quads for each slice in the volume texture
  - In GS, send the primitives into the appropriate slices
  - Bind this texture via a sampler and use HW filtering
- Usual performance implications for volume textures usage (read vs. write cache coherency)
- However - This is very advantageous for solid noise

Outline

- Introduction: procedural techniques and noise
  - Properties of ideal noise primitive
- Lattice Noise Types
  - Lattice noise and permutation tables
  - Value noise
  - Gradient (classic Perlin noise)
  - Improved Perlin noise
  - Simplex noise
- Noise Summation Techniques
- Reducing artifacts
  - General strategies
  - Antialiasing
- Snow accumulation and terrain generation
- Conclusion

Lattice Noise

- Simplest method to introduce noise
  - Repeatable
- 1D pseudo-random **permutation table** of length *n*
  - The table is precomputed or can be generated per frame.
  - Contains all integer values from 0 to *n-1* in a random order
  - Uniformly distributed PRN (pseudo-random numbers)
  - Indexed modulo *n*
- Often used on the GPU for fast PRNs
  - Currently there isn't a random number generation primitive on the GPU
  - Efficient – just a look-up without any filtering

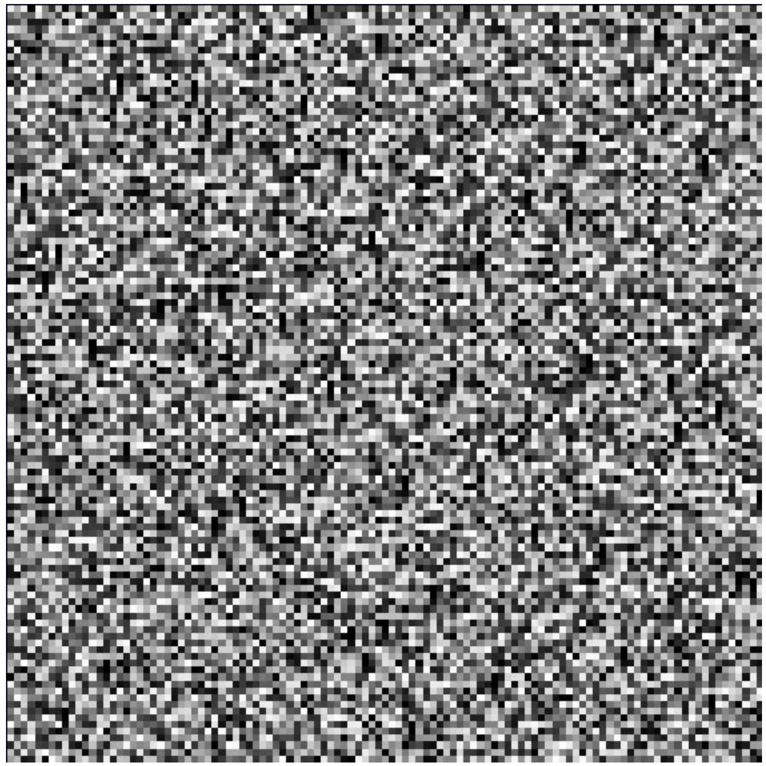
Permutation Table in 2 and Higher Dimensions

- The permutation table entry of a dimension is used to perturb the index into the next dimension
  - As necessary
  - `perm2D(x, y)       = perm(y + perm(x));`
  - `perm3D(x, y, z)    = perm(z + perm2D(x, y));`
  - `perm4D(x, y, z, t) = perm(t + perm3D(x,y,z));`
- Common approach:
  - Bake `perm2D` into a 2D repeatable texture
  - Ideal size of noise functions is >256
  - Very costly for 3D textures
    - Memory and cache performance unfriendly

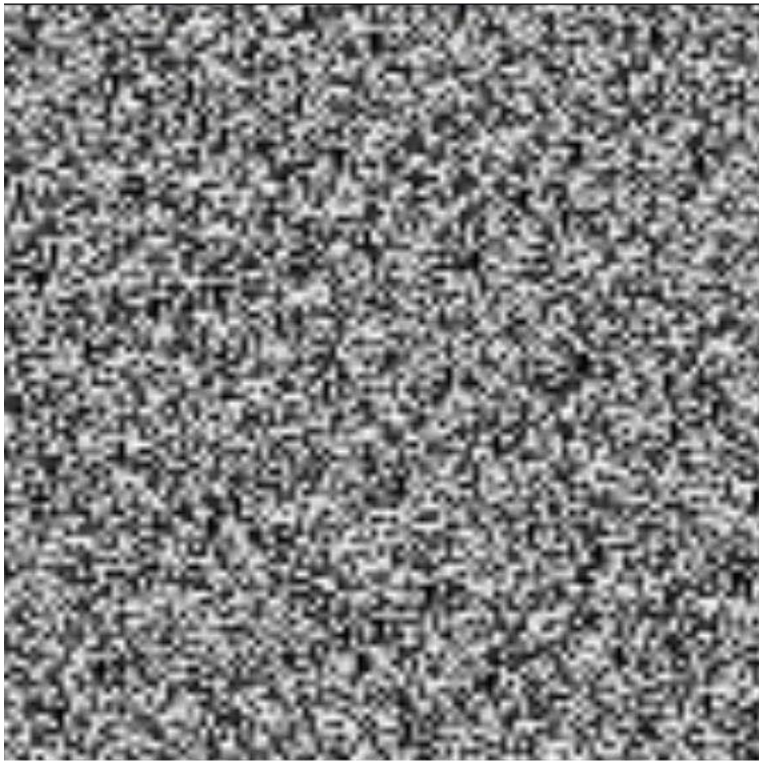
Value Noise

- Simplest method to generate a low-pass filtered stochastic function
- Given a PRN between -1 and 1 at each lattice point, compute a noise function by interpolating these random values
- Key difference is in the interpolation function
  - Linear interpolation produces “boxy” artifacts
    - The derivative of a linearly interpolated value is not continuous
    - Produces visually noticeable sharp changes
  - For smooth noise we can use cubic interpolation instead
    - Catmull-Rom spline
    - Hermite spline
    - Quadratic / cubic B-splines
    - Evaluation can be not cheap

Value Noise: Point



Value Noise: Linear Interpolation

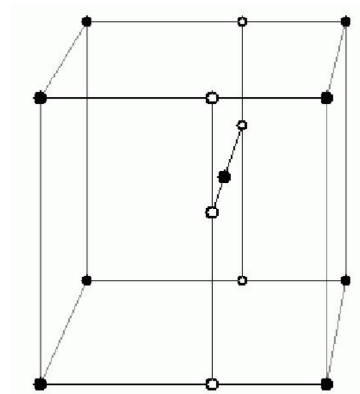


Value Noise Shader

```
float ValueNoiseSmoothstepQuintic2D( float4 texCoord )
{
   float4 texelVals;
   float4 uvFrac;           //fractional component of texture coordinates
   float4 smoothStepFrac;   //smooth step frac
   float4 interpVals;       //interpolation values
   float4 offsetTexCoord;

   offsetTexCoord = texCoord - g_fPermHalfTexelOffset;

   // Assumes 2x2 neighborhood packing for noise function
   texelVals = tex2D( Perm2DSamplerPoint, offsetTexCoord) * 2 – 1.0;

   // Derive fractional position
   uvFrac = frac( offsetTexCoord * g_fPermTextureSize );

   // Quintic smoothstep interpolation function: 6t^5 - 15t^4 + 10t^3
   smoothStepFrac = (((6 * uvFrac) - 15) * uvFrac + 10) * uvFrac * uvFrac *
                      uvFrac;

   // Build weights for 2x2 interpolation grid
   interpVals = float4( 1 - smoothStepFrac.x, smoothStepFrac.x,
                        1 - smoothStepFrac.x, smoothStepFrac.x);
   interpVals *= float4( 1 - smoothStepFrac.y, 1 - smoothStepFrac.y,
                         smoothStepFrac.y, smoothStepFrac.y);

    return( dot( interpVals, texelVals ) );
}
```

Gradient (Classic Perlin) Noise

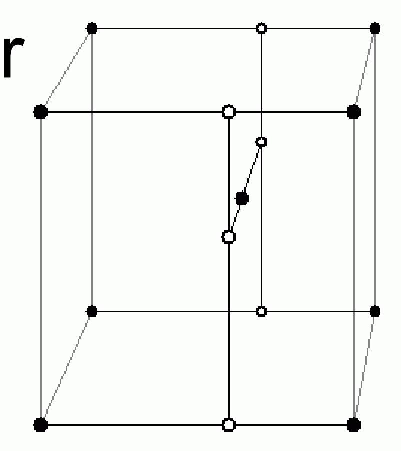
- Generate a pseudorandom gradient vector at each lattice point and then use the gradients to generate the stochastic function
  - Given an input point *P* (*n*-dimensions)
  - For each of its neighboring grid points
    - Pick a “pseudo-random” direction vector
    - Compute linear function (dot product)
  - Combine with a weighted sum
    - Using a cubic ease curve in each dimension



- Smooth function which has a regular pattern of zero crossings
  - Value of a gradient noise is 0 at all the integer lattice points
  - Combine value and gradient noise to have non-zero crossings

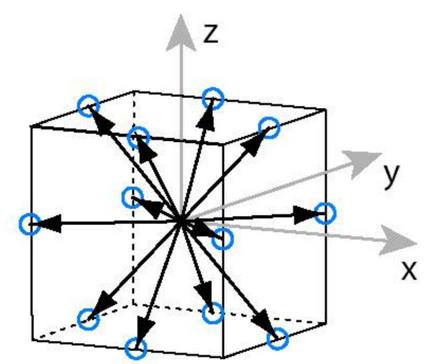
Gradient Noise in Higher Dimensions

- In 3D:
  - The gradients are three-dimensional
  - The interpolation is performed along three axes, one at a time
- Similar generation to 4 and higher dimensions
  - Proportional increase in cost



Gradient Generation

- The gradients determine behavior: must be pseudo-random
  - Need enough variation to conceal that the noise function isn't truly random.
  - However, too much variation can cause unpredictable behavior in the noise function
- Pick gradients of unit length equally distributed in directions
  - 2D: 8 or 16 gradients distributed around the unit circle
  - 3D: use the midpoints of each of the 12 edges of a cube centered on the origin

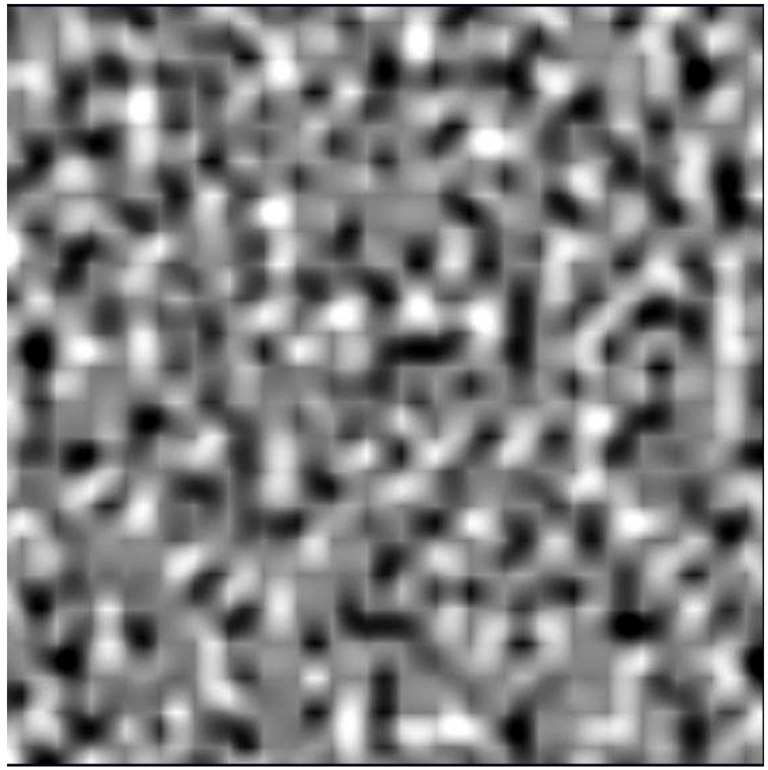


- In reality, as long as we have enough gradients and they're evenly distributed over all directions, that's enough

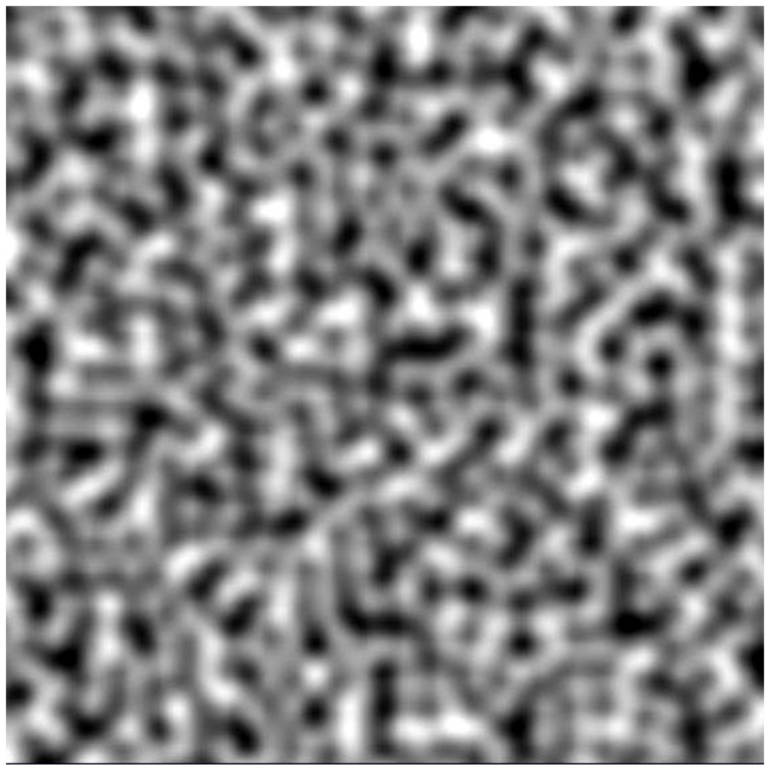
Pack 2x2 Neighborhoods for Noise Optimizations

- We are sampling over the 2x2 lattice neighborhood
  - Fetching values from the permutation & gradient tables for each location
- We can permutation & gradient values to store 2x2 neighborhoods in 4 channels
  - Fetch in a single texture fetch
  - Vectorize noise computations
  - Better bandwidth utilization and memory savings for the tables

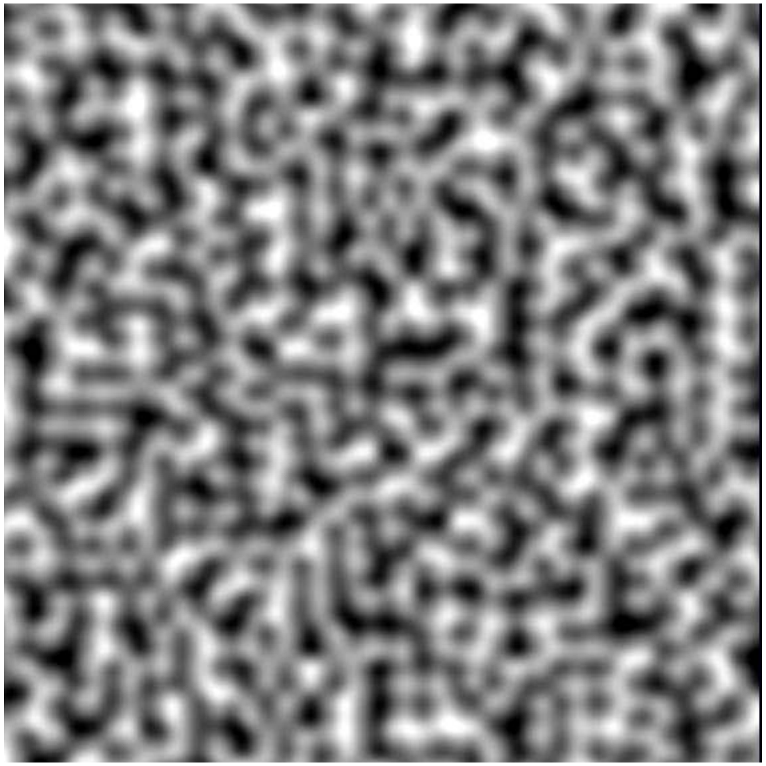
Gradient Noise: Linear Interpolation



Gradient Noise: Cubic Interpolation



Gradient Noise: Quintic Interpolation



2D Gradient Noise Shader

```
float GradientNoiseSmoothstepCubic2D( float4 texCoord )
{
   float4 xGrad, yGrad, xDelta, yDelta, xExtrap, yExtrap, gradDotDist,
          uvFrac, smoothStepFrac, interpVals, offsetTexCoord;

   // Interpolation values
   offsetTexCoord = texCoord - g_fPermHalfTexelOffset;

   // Sample 2x2 neighborhood of gradient values for each dimension
   xGrad = 4 * tex2D( GradientTableXSamplerPoint, offsetTexCoord) - 2;
   yGrad = 4 * tex2D( GradientTableYSamplerPoint, offsetTexCoord) - 2;

   // Derive fractional position
   uvFrac = frac( offsetTexCoord * g_fPermTextureSize );

   // Extrapolate gradients. Distance in X from each vertex.
   xDelta = float4( uvFrac.x, uvFrac.x - 1, uvFrac.x, uvFrac.x - 1 );
   xExtrap = xGrad * xDelta;

   // Distance in Y from each vertex.
   yDelta = float4( uvFrac.y, uvFrac.y, uvFrac.y - 1, uvFrac.y - 1 );
   yExtrap = yGrad * yDelta;
…
```

2D Gradient Noise Shader (cont.)

```
…
   //This now contains the 2D dot product between the gradient vector
   // and the x, y offsets from lattice point in the current 2x2
   // neighborhood.
   gradDotDist = xExtrap + yExtrap;

   // Use smoothstep based interpolation of extrapolated values.
   smoothStepFrac = ((-2 * uvFrac) + 3) * uvFrac * uvFrac;

   interpVals = float4( 1 - smoothStepFrac.x, smoothStepFrac.x,
                        1 - smoothStepFrac.x, smoothStepFrac.x );

   interpVals *= float4( 1 - smoothStepFrac.y, 1 - smoothStepFrac.y,
                         smoothStepFrac.y, smoothStepFrac.y );

   return dot( gradDotDist, interpVals );
}
```

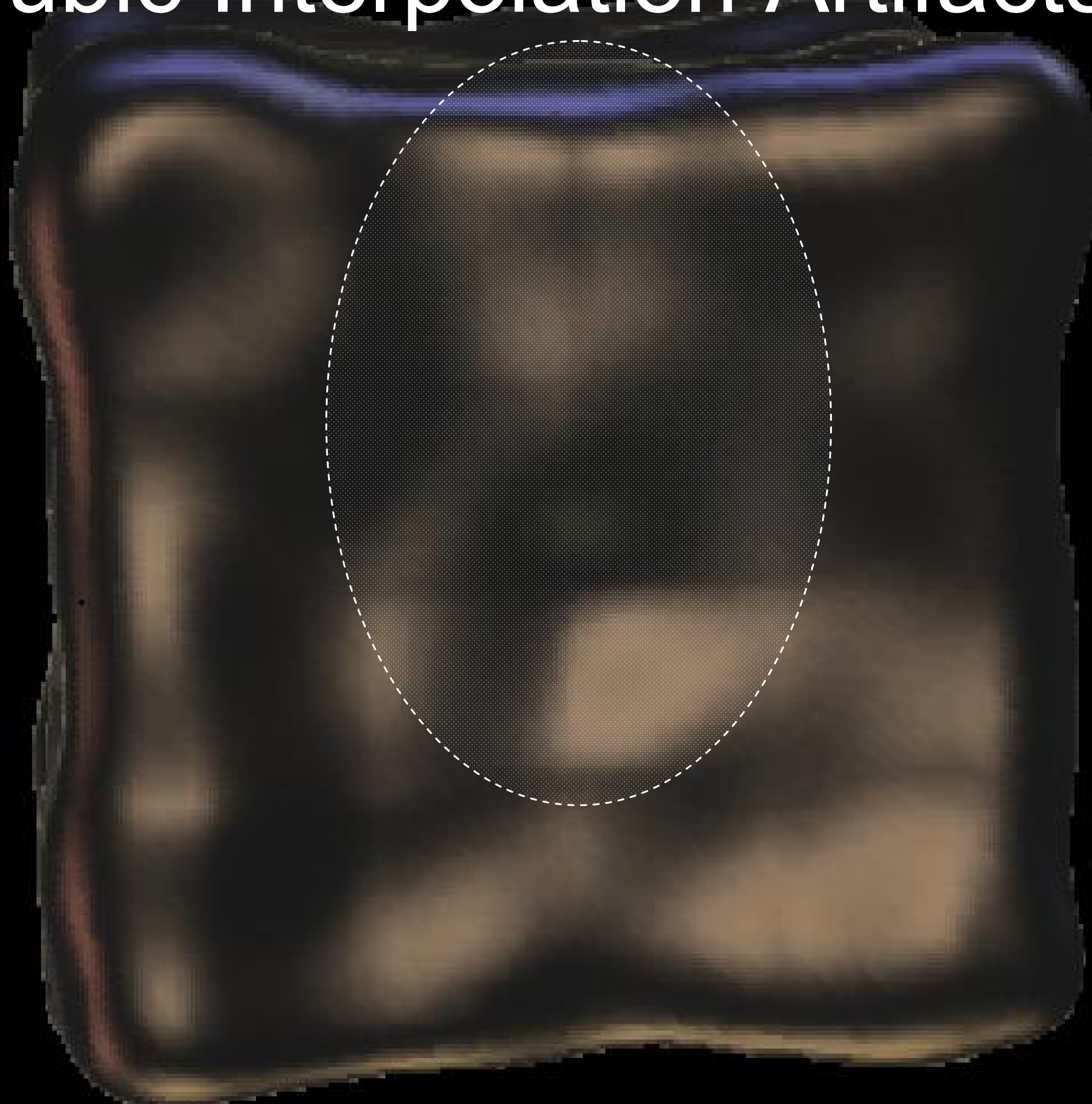
Classic Perlin 2D Noise Shader

```
float3 noise( const in float2 P )
{
  // Integer part, scaled and offset for texture lookup
  float2 Pi = cfTEXEL_SIZE * floor( P ) + cfHALF_TEXEL_SIZE;
  float2 Pf = frac(P); // Fractional part for interpolation

  // Noise contribution from lower left corner:
  float2 grad00 = tex2D( sPermutationTable, Pi ).rg * SCALE + BIAS;
  float n00 = dot( grad00, Pf );

  // Noise contribution from lower right corner
  float2 grad10 = tex2D( sPermutationTable, Pi + float2( cfTEXEL_SIZE, 0.0 )).rg * SCALE + BIAS;
  float n10 = dot( grad10, Pf - float2(1.0, 0.0) );

  // Noise contribution from upper left corner
  float2 grad01 = tex2D (sPermutationTable, Pi + float2( 0.0, cfTEXEL_SIZE )).rg * SCALE + BIAS;
  float n01 = dot( grad01, Pf - float2( 0.0, 1.0 ));

  // Noise contribution from upper right corner
  float2 grad11 = tex2D( sPermutationTable, Pi + float2( cfTEXEL_SIZE, cfTEXEL_SIZE )).rg *
    SCALE + BIAS;
  float n11 = dot( grad11, Pf - float2( 1.0, 1.0 ));

  // Blend contributions along x
  float2 n_x = lerp( float2( n00, n01 ), float2( n10, n11 ), InterpolateC2Continuous( Pf.x ));

  // Blend contributions along y
  float n_xy = lerp(n_x.x, n_x.y, InterpolateC2Continuous(Pf.y));

  // We're done, return the final noise value.
  return float3( n_xy.xxx );
}
```

Limitations of Gradient and Perlin Noise

- Biggest problems is artifacts
- 2nd order derivative discontinuity due to choice of a cubic interpolant
  - Non-zero at lattice grid cells
  - Introduces visual artifacts
  - Especially when bump- or displacement- mapping
- Not rotation-invariant
  - Easy to distinguish grid patterns
  - Even though the gradients were distributed randomly over an *n*-sphere, the cubic grid has directional bias
    - Shortened along the axes
    - Elongated on the diagonals
  - This can produce “clumped” gradients
  - Axis-aligned

Cubic Interpolation Artifacts



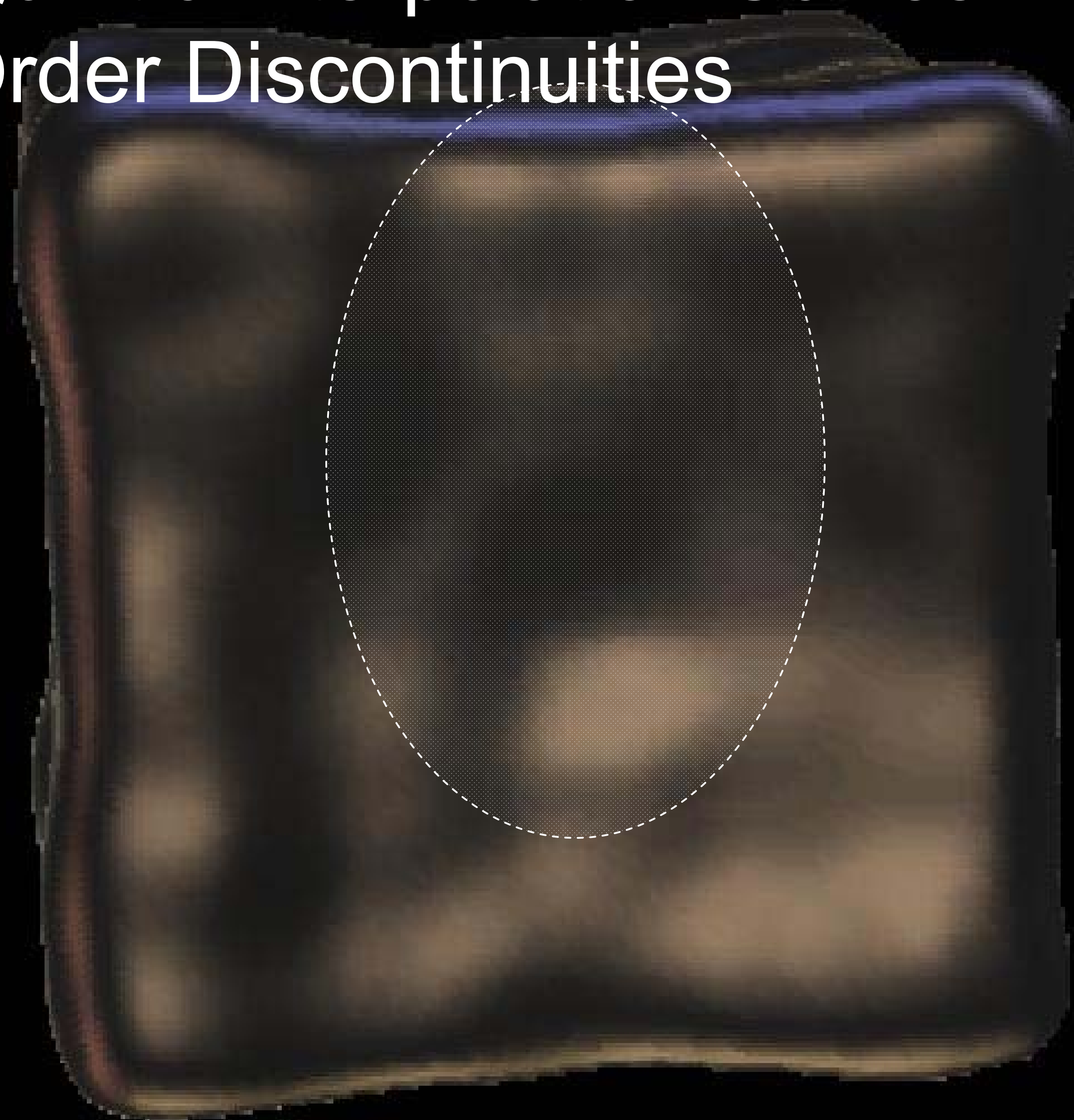
Improved Perlin Noise

- Perlin identified these problem areas of his original noise implementation
- Solution:
  - Quintic polynomial for interpolation
    - Instead of the original Hermite

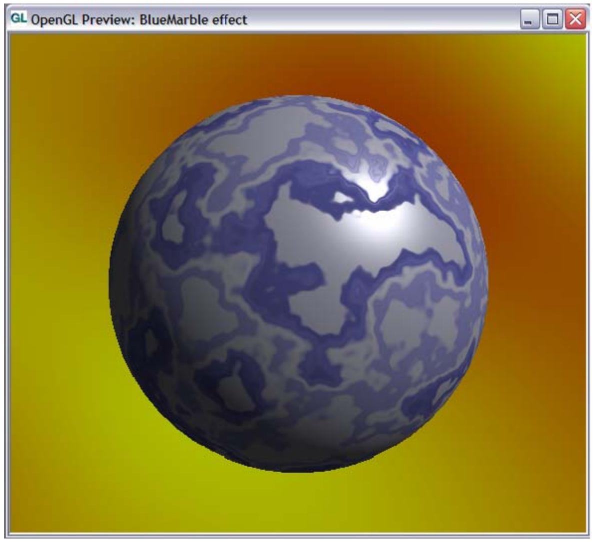
$$f(t) = 6t^5 - 15t^4 + 10t^3$$

  - Simplex grid for gradient selection
- Fixes limitations of previous noise
  - Continuous for 2nd order derivative at zero crossings
  - Removes directional bias and clumping of gradients

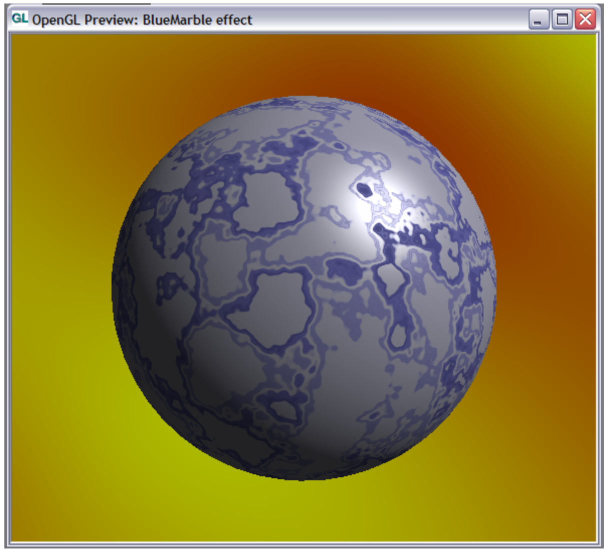
Quintic Interpolation Solves 2nd Order Discontinuities



Simplex vs. Perlin Improved Noise



(a) 3D Perlin Improved Noise

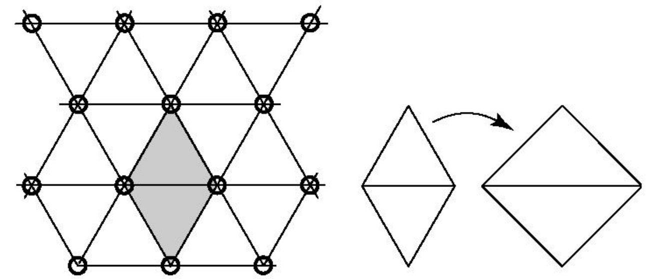


(b) 3D Perlin Simplex Noise

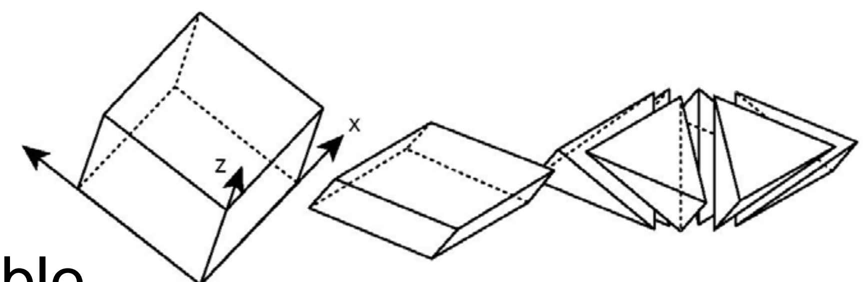
- Both average to the same value
- Perlin Simplex noise has a slightly higher peak range
- Simplex noise is cheaper in higher dimensions (3+)
- Higher quality

Simplex Geometry

- A simplex is the generalization of a tetrahedral region of space to *n* directions
  - Changing the lattice for sampling noise
  - Instead of using an orthogonal cubic lattice
  - Define noise on simplices



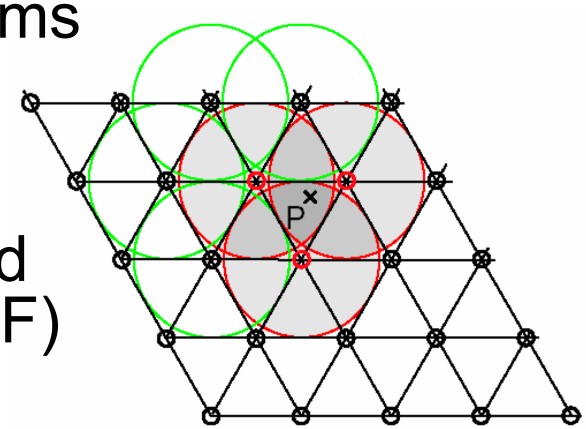
- Simplex: the simplest and most compact shape that can be repeated to fill the entire space
  - 1D: equal length intervals
  - 2D: squished triangles
  - 3D: squished tetrahedrons



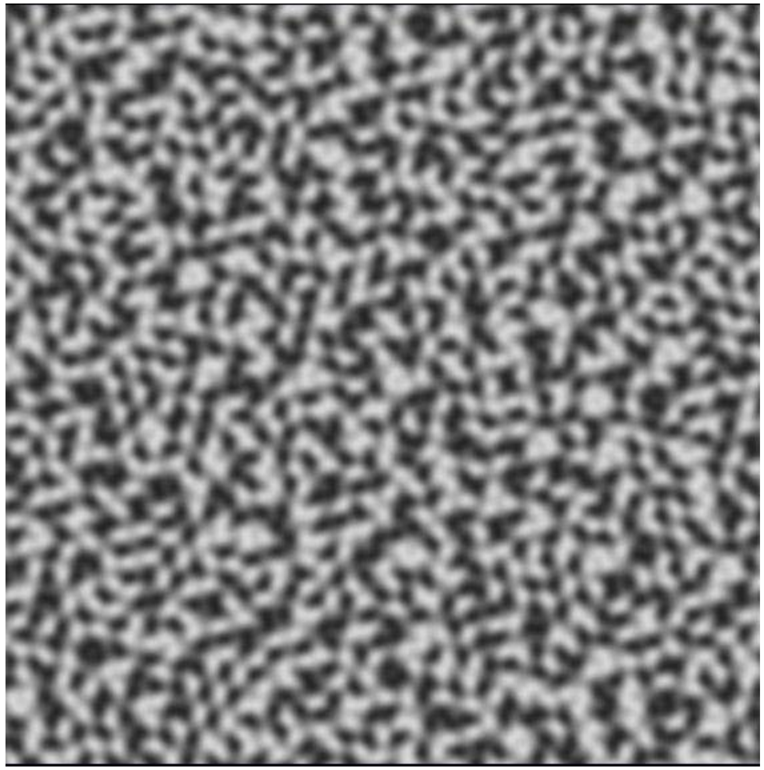
- A simplex shape has as few corners as possible
  - Fewer than a cube
  - Cheaper for interpolation

Interpolation on Simplex Grids

- Simplex noise uses straight summation of corner contributions
- The noise value at each point can be always calculated as a sum of three terms
  - Points inside a simplex are only influenced by the contributions from the corners of that simplex
- Interpolation weights are defined using radial basis functions (RBF) centered around each vertex
  - RBF become extinct when reaching the opposite corner (limited extent)
- n+1 vs. 2^n interpolations of the noise function prior to interpolation



Perlin Simplex Noise



Determining the Simplex

```
void simplex( const in float3 P, out float3 offset1,
              out float3 offset2 )
{
  float3 offset0;

  float2 isX = step( P.yz, P.xx );
  offset0.x  = dot( isX, float2( 1.0, 1.0 ) );
  offset0.yz = 1.0 - isX;

  float isY = step( P.z, P.y );
  offset0.y += isY;
  offset0.z += 1.0 - isY;

  offset2 = clamp(   offset0, 0.0, 1.0 );
  offset1 = clamp( --offset0, 0.0, 1.0 );

}
```

Perlin Simplex Noise Algorithm

- Transform to “sheared” space
- Select grid location (which cell, cube, hypercube)
- Transform cell origin back
- Determine the simplex
- Sum nearest vertex contributions
- Scale

2D Simplex Noise Shader

```
float3 snoise( const in float2 P )
{
  // Skew and unskew factors are a bit hairy for 2D, so define them as constants
  #define F2 0.366025403784            // This is (sqrt(3.0)-1.0)/2.0
  #define G2 0.211324865405            // This is (3.0-sqrt(3.0))/6.0

  // Skew the (x,y) space to determine which cell of 2 simplices we're in
  float  u  = ( P.x + P.y ) * F2;
  float2 Pi = floor( P + u );
  float  v  = ( Pi.x + Pi.y ) * G2;
  float2 P0 = Pi - v; // Unskew the cell origin back to (x,y) space

  Pi = Pi * cfTEXEL_SIZE + cfHALF_TEXEL_SIZE; // Integer part, scaled and offset
    for texture lookup
  float2 Pf0 = P - P0;                             // The x,y distances from the cell
    origin

  // For the 2D case, the simplex shape is an equilateral triangle.
  // Find out whether we are above or below the x = y diagonal to
  // determine which of the two triangles we're in.
  float2 o1;
  if ( Pf0.x > Pf0.y )
     o1 = float2( 1.0, 0.0 );  // +x, +y traversal order
  else
     o1 = float2( 0.0, 1.0 );  // +y, +x traversal order

  float n = 0.0;
 …
```

2D Simplex Noise Shader (cont.)

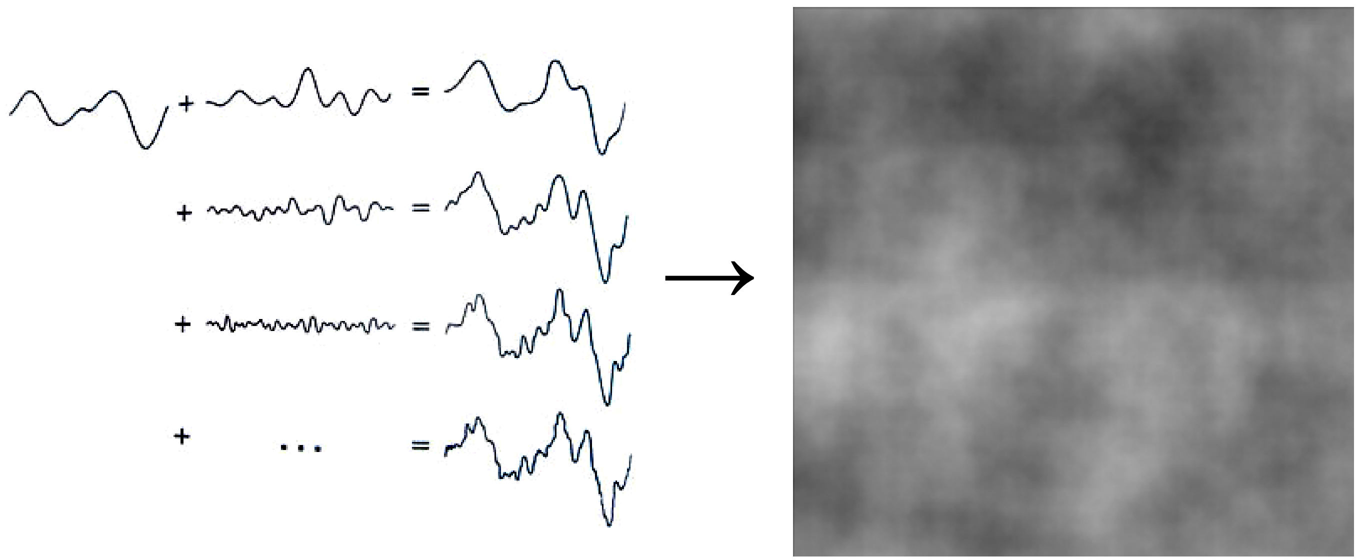
```
…
  // Noise contribution from simplex origin
  float2 grad0 = tex2D( sPermutationTable, Pi ).rg * SCALE + BIAS;
  float  t0    = 0.5 - dot( Pf0, Pf0);
  if ( t0 > 0.0 )
  {
    t0 *= t0;     n  += t0 * t0 * dot( grad0, Pf0 );
  }
  // Noise contribution from middle corner
  float2 Pf1   = Pf0 - o1 + G2;
  float2 grad1 = tex2D( sPermutationTable, Pi + o1 * cfTEXEL_SIZE ).rg * SCALE +
   BIAS;
  float  t1    = 0.5 - dot( Pf1, Pf1 );
  if ( t1 > 0.0)
  {
    t1 *= t1;     n  += t1 * t1 * dot( grad1, Pf1 );
  }
  // Noise contribution from last corner
  float2 Pf2   = Pf0 - float2( (1.0 - 2.0 * G2).xx );
  float2 grad2 = tex2D( sPermutationTable, Pi + float2( cfTEXEL_SIZE,
                                            cfTEXEL_SIZE)).rg * SCALE + BIAS;
  float  t2    = 0.5 - dot(Pf2, Pf2);
  if ( t2 > 0.0 )
  {
    t2 *= t2;    n  += t2 * t2 * dot( grad2, Pf2 );
  }

  // Sum up and scale the result to cover the range [-1,1]
  return float3( 70.0 * n.xxx );
}
```

Outline

- Introduction: procedural techniques and noise
  - Properties of ideal noise primitive
- Lattice Noise Types
- Noise Summation Techniques
  - Fractional Brownian motion
  - Turbulence
- Reducing artifacts
  - General strategies
  - Antialiasing
- Snow accumulation and terrain generation
- Conclusion

fBm: Fractional Brownian Motion

- What happens when we add several octaves of noise together?
- fBm adds several copies of noise() together
  - Each copy w/ different amplitude & frequency



fBm

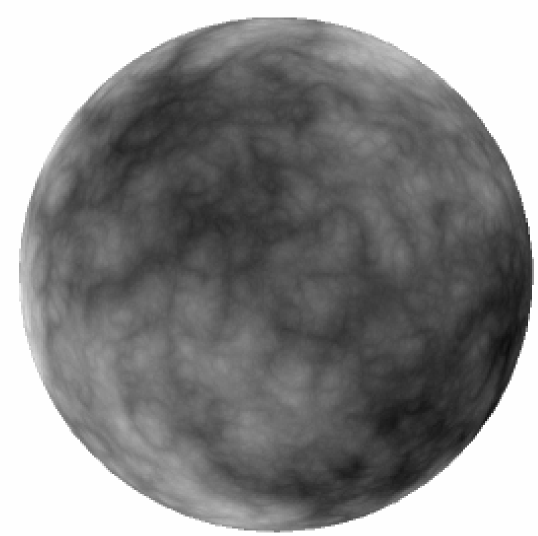
- The frequencies and amplitudes are related by **`lacunarity`** & **`gain`** respectively
  - **`Lacunarity`** controls frequency change between each band
  - **`Gain`** controls amplitude change between each band
  - Typically: **`lacunarity`** = 2, **`gain`** = 0.5
  - Any time that **`gain = 1/ lacunarity`** => “1/f” noise
- fBm is self-similar
  - Summing up different copies of itself at different scales

fBm Shader Code

```
float fBm( float3 vInputCoords, float nNumOctaves, float fLacunarity,
           float  fGain )
{
   float fNoiseSum     = 0;
   float fAmplitude    = 1;
   float fAmplitudeSum = 0;

   float3 vSampleCoords = vInputCoords;

   for ( int i = 0; i < nNumOctaves; i+= 1 )
   {
      fNoiseSum     += fAmplitude * noise( vSampleCoords );
      fAmplitudeSum += fAmplitude;

      fAmplitude    *= fGain;
      vSampleCoords *= fLacunarity;
   }

   fNoiseSum /= fAmplitudeSum;

   return fNoiseSum;
}
```

Turbulence

- Same as fBm, but add **`abs(noise)`**
  - Roughly doubles the effective frequency
  - Makes everything positive
  - More “billowy” appearance
- *Beware*: abs() can introduce high frequencies
  - May increase the amount of aliasing

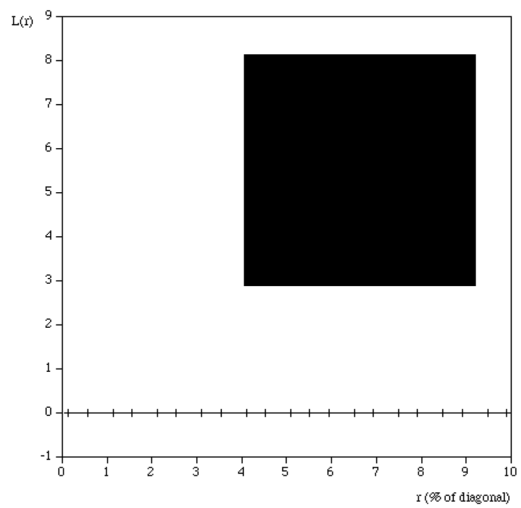


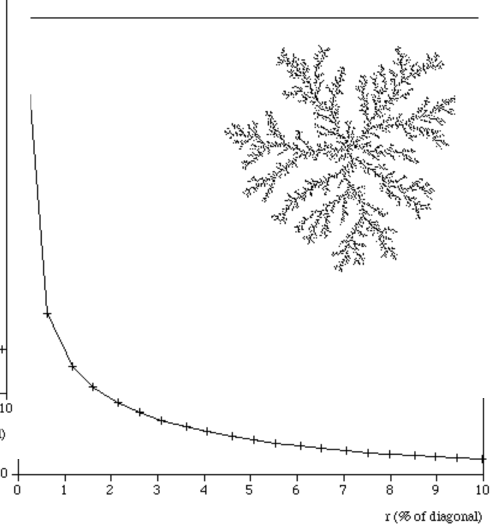
Turbulence Shader code

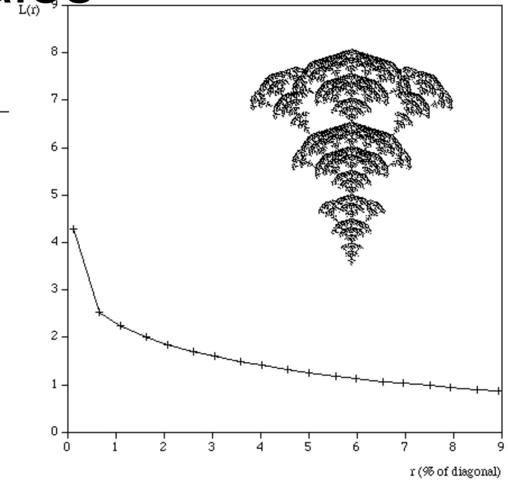
```
float Turbulence( float3 vInputCoords, float nNumOctaves,
                  float  fLacunarity, float fGain)
{
   float fNoiseSum     = 0;
   float fAmplitude    = 1;
   float fAmplitudeSum = 0;

   float3 vSampleCoords = vInputCoords;

   for ( int i = 0; i < nNumOctaves; i+= 1 )
   {
      fNoiseSum += fAmplitude * abs( noise( vSampleCoords ).x);
      fAmplitudeSum += fAmplitude;

      fAmplitude    *= fGain;
      vSampleCoords *= fLacunarity;
   }

   fNoiseSum /= fAmplitudeSum;
   return fNoiseSum;
}
```

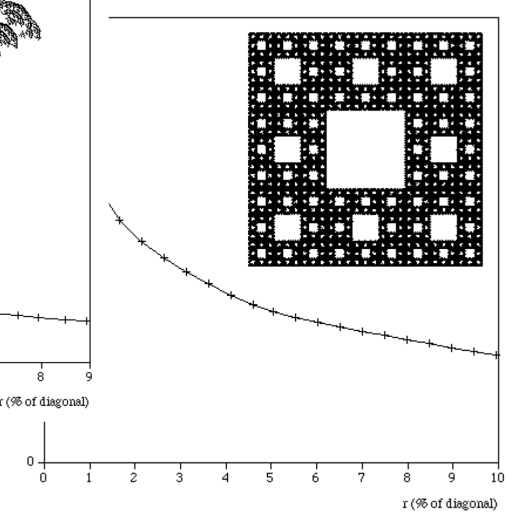
Lacunarity

- A measure how a fractal curve fills space
  - If the fractal is dense, lacunarity is small
  - Lacunarity increases with coarseness
- In our context it is the ratio between the sizes of successive octave scales





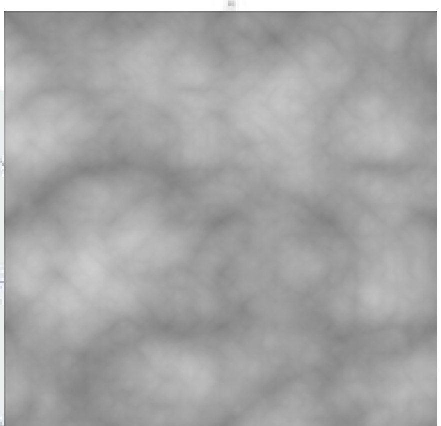




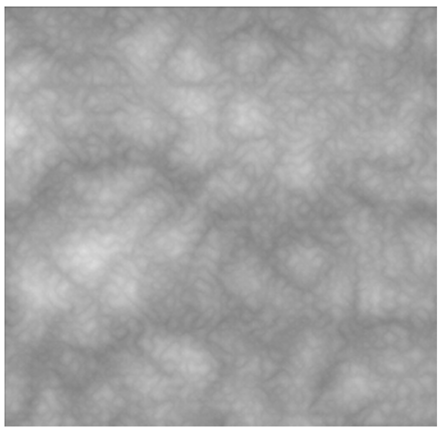
Lacunarity and Noise

- The same applies to noise
  - For fBm and Turbulence or other noise sums

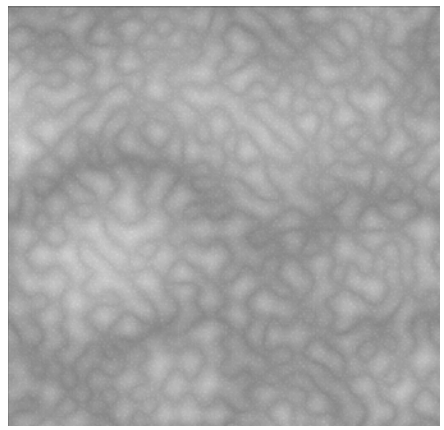
Turbulence:



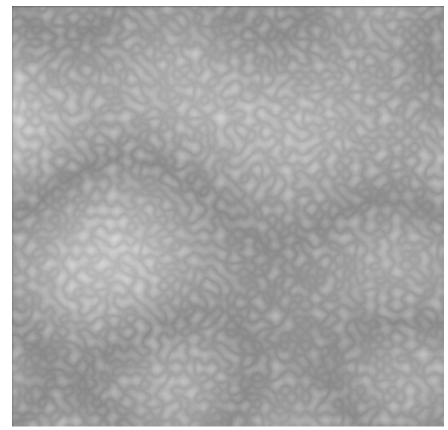
Lacunarity = 2



Lacunarity = 4



Lacunarity = 8



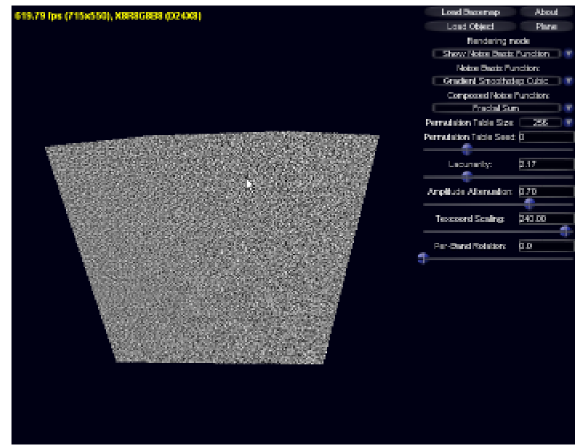
Lacunarity = 16

Outline

- Introduction: procedural techniques and noise
  - Properties of ideal noise primitive
- Lattice Noise Types
- Noise Summation Techniques
- Reducing artifacts
  - General strategies
  - Antialiasing
- Snow accumulation and terrain generation
- Conclusion

Aliasing: The Bane of Shader-Writer's Existence

- Everyone has experienced it…
  - The sharp jaggies
  - Pixellated image
  - Swimming pixels
  - Shimmering pixels
  - Horrible twinkling
  - Or just bizarre artifacts
  - And motion only makes it worse
- This is aliasing, and it's a fact of life when writing shaders



Strategies for Reducing Artifacts

- Sum up more frequency bands
  - Higher quality result, more control
  - That's why the film folks use up to 200+ octaves
- Rotate each frequency band to align to a (precomputed) random orientation
  - The lattices of different scales won't line up
  - Makes the artifacts less noticeable
  - For derivatives (for bump mapping) have to multiply the derivative with an inverse rotation matrix before you sum them
    - Otherwise will see artifacts
- Using non-power-of-two lacunarity values with rotated frequency band also helps reducing artifacts
  - This also lets you use smaller permutation tables

Strategies for Reducing Artifacts: Lacunarity

- Using non-power-of-two lacunarity values with rotated frequency band also helps reducing artifacts
  - This also lets you use smaller permutation tables
- Don't use exact values for lacunarity
  - Use 1.93485736 or 2.18387276 instead of 2.0
  - An exact ratio makes the different bands "align"
    - The next smaller scale repeats exactly twice on top of the larger scale
    - Artifacts can appear periodically
  - This periodicity is broken by using a number that's not a simple ratio

Aliasing: Quick Recap

- Want each pixel to represent some weighted average measure of the image function in the area “behind” the entire pixel
- Sampling Theorem
  - Signal reconstruction is only guaranteed to work when signal bandwidth ≤ the information captured by samples
  - The latter depends on sampling rate
- Signal bandwidth > sampling rate → Aliasing
  - Threshold: Nyquist frequency
- High frequency energy doesn’t disappear!
  - Energy from high frequencies components converted to wrong low-frequency energy
  - This is *alias* of the high-frequency energy in the original signal

Aliasing: Causes

- Two sources of aliasing: screen-space and the shading samples aliasing
  - First is traditionally solved with MSAA or super sampling or stochastic sampling
  - Second is trickier

Prefiltering

- Often, by the time the shader is executed, it is too late – aliasing has been introduced
- We want to “prefilter” the shading
  - A weighted average of the shader function in the neighborhood of the point being shaded.
- Another weight to think about it: convolve the shader function with a filter for the sample
  - Some kind of average value underneath the rendered pixel
- Key difficulty: estimating the filter width and weights

SuperSampling: Poor man's Antialiasing

- One method is to use brute force
  - Pont sample multiple points under the filter kernel
  - Average
- Replaces one point sample with many
  - One possible solution – only recompute the portion that is causing the aliasing
- The error (aliasing) decreases only as the *square root* of the number of samples, *n*
  - Yet the cost of shading is x *n:* need to run the full shader computation for all samples
- Have to do a huge amount of extra shading to eliminate the aliasing

Stochastic Sampling for Antialiasing

- Sample the signal at irregularly spaced points
  - Energy from frequencies above the Nyquist frequency would then appear as *random noise*
    - Rather than a structured low-frequency alias
    - People are less likely to notice this noise in the final image than they are to noise the low-frequency alias pattern
- Expensive
    - Requires evaluating the procedural shader many times
  - Alternatively separate shader sampling from pixel sampling

Removing Aliasing

- One solution: increase sampling rate
  - Supersamping / multisampling
- Not always possible
  - Resolution / memory footprint / speed of evaluation issues
  - Some signals have unlimited bandwidth (Ex: step function)
  - For those we can't rid of high frequencies regardless of how high the sampling rate is
- Ideal goal: take out the excessive high frequencies out of the original signal *before* sampling
  - In the context of procedural texturing, designing antialiasing *into* texture evaluation

Pregenerate the Texture (Prebake or On the Fly)

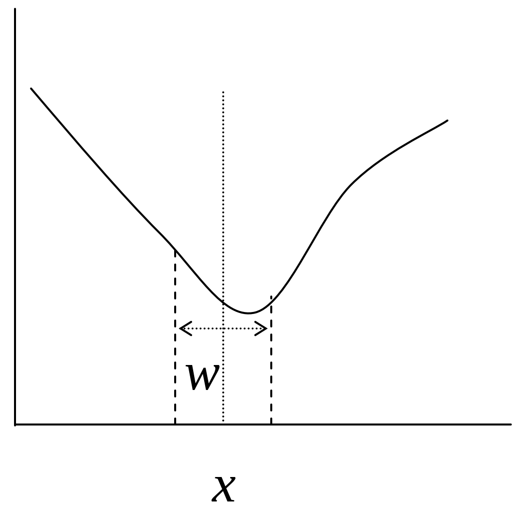
- We can also take advantage of hardware mip mapping
  - Generate the shading as a texture – on the fly per frame
  - When fetching to apply to the surface
  - HW will filter and remove the aliasing
- Pros
  - We can vary the resolution as necessary or change the frustum
  - This lets us have unlimited detail for the object
    - Zoom in / zoom out
- Con: extra draw calls and texture memory
- This works if the original shader doesn't have aliasing artifacts

Antialiasing Procedural Shaders

- Must prefilter to get rid of aliasing
  - If we don't want to pay the cost for supersampling
- Two prefiltering strategies:
  - Analytic solutions to the integral
  - Frequency clamping methods

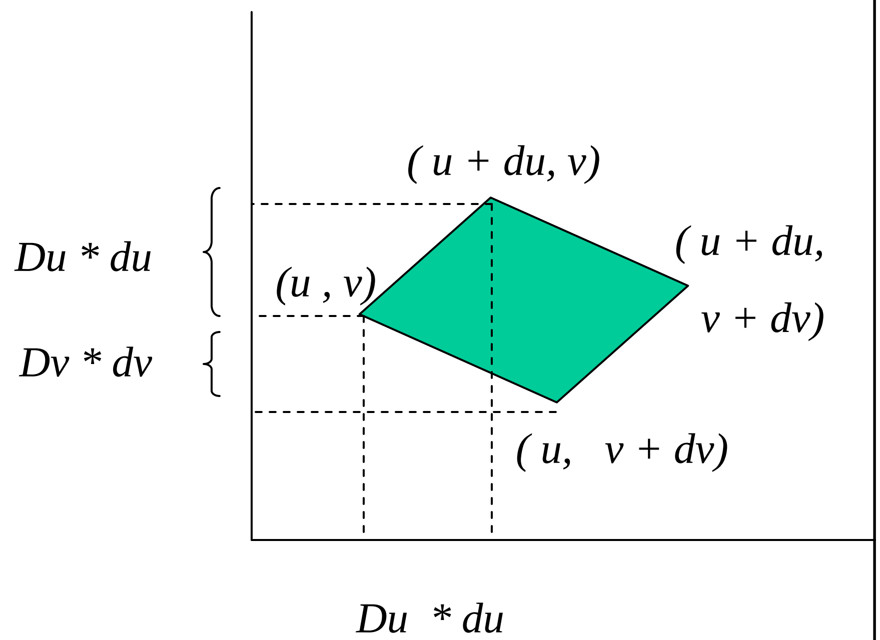
Filter Estimation

- How big should *w* be in order to cover the pixel?
- Use derivatives to estimate the change for *x* (ddx / ddy)
  - Take the derivatives of the sampling coordinates for the procedural shader
  - We can use this to estimate the area covered by the pixel or the current mip map
  - Example: see the Parallax Occlusion Mapping: Educational sample in the upcoming release of RenderMonkey for these computations



Filter Estimation with Derivatives

- Square root of area is a decent estimate of the amount that *p(x)* changes between adjacent pixels
  - This assumes that u ⊥ v
- This is the estimate for the filter width



Analytic Prefiltering

- Use knowledge of the sampling function *f* derive an analytic formula for prefiltering
- Remember that we are convolving filter kernel *k* with our procedural function *f*

$$F(x) = (f \otimes k)(x) = \int_{-\infty}^{\infty} f(\delta)k(x-\delta)d\delta$$

- Consider the simple case of averaging over the interval [x – w/2, x + w/2]
  - Equivalent to convolving the input signal with a box filter
  - We can assume this for our convolution kernel
- If we really need to, we can also compute summed-area tables in real-time to compute this integral
  - See [Hensley05] for reference of real-time SAT computation

Example: Analytic Prefiltering of the Step Function

- Replacing a step function with *filteredstep*

```
float filteredstep( float fEdge, float x, float w)
{
   return clamp( (x + w/2 – fEdge)/w, 0, 1 )
}
```

- This convolves the step function with a box filter

Antialiasing by Frequency Clamping

- But often we can't derive an analytic formula for prefiltering many procedural functions (including noise)
  - They often simply don't have an analytic solution
- The next best thing: frequency clamping
  - Decompose your shader into composite functions with known frequencies
  - Only use the frequencies that are low enough to be below your sampling rate
  - This is ideal for antialiasing noise
- We need to know the filter size in order to determine which frequencies to keep

Frequency Clamping Strategy

- We want to antialias our procedural function *f(x)* and we know the filter width *w*
- Suppose we know the following:
  - Function *f* has no features smaller than w_f
  - The average value of *f(x)* is *a*
- Then *f* won't alias when $w \ll w_f/2$, and will alias when $w \gg w_f/2$
- But we know *the average!*
- Why not substitute it when the filter is too wide compared to the feature size?
  - Use smoothstep to fade between the true function and its average between those extremes

```
#define fadeout( f, fAverage, fFeatureSize, fWidth )\
 lerp( f, fAverage, smoothstep( 0.2, 0.6, \
    fWidth / fFeatureSize)
```

Noise Frequency Clamping Strategy

- We know the average value for noise: 0 for signed, and 0.5 for unsigned
- Then we can easily add the following macros to our noise functions
  - Turbulence, fBm

```
#define filterednoise(x, w) \
    fadeout(noise(x), 0.5, 1, w)
```

Filtered fBm Shader Code

```
float fBm( float3 vInputCoords, float nNumOctaves, float fLacunarity,
           float  fInGain, float fFilterWidth )
{
   float fNoiseSum           = 0;
   float fAmplitude          = 1;
   float fAmplitudeSum       = 0;
   float fFilterWidthPerBand = fFilterWidth;

   float3 vSampleCoords = vInputCoords;

   for ( int i = 0; i < nNumOctaves; i+= 1 )
   {
      fNoiseSum += fAmplitude * filterednoise( vSampleCoords,
                   fFilterWidthPerBand );
      fAmplitudeSum += fAmplitude;

      fFilterWidthPerBand *= fLacunarity;
      fAmplitude          *= fInGain;
      vSampleCoords       *= fLacunarity;
   }

   fNoiseSum /= fAmplitudeSum;

   return fNoiseSum;
}
```

Frequency Clamping: Cons

- Not really low-pass filtering
  - Each octave fades to the average as the frequency gets high enough
- When the noise frequency is twice the Nyquist limit, it will be attenuated severely by *fadeout(..)*.
  - But the noise octave has power at all frequencies!
  - Real low-pass filtering would completely eliminate the high frequencies
  - Leave the low frequencies intact
- Frequency clamping may not be enough
  - It attenuates all frequencies equally, leaving to much of highs and removing too much of lows
  - This can cause artifacts when the filter width is too large
  - Just something to be aware of

Outline

- Introduction: procedural techniques and noise
  - Properties of ideal noise primitive
- Lattice Noise Types
- Noise Summation Techniques
- Reducing artifacts
  - General strategies
  - Antialiasing
- Snow accumulation and terrain generation
- Conclusion

Practical Example: Mountains Generation and Realistic Snow Accumulation

Use fBm to Generate Mountain Terrain

- Compute multiple octaves (10-50) of fBm noise to use as displacement
  - Vertex texture-based displacement
- Variety of options
  - Compute displacement directly in the shader per frame
    - Great for animating earthquakes
  - Stream out and reuse as necessary
  - Precompute for static geometry
- Use masks to vary noise computation / parameters as needed

Mountains: Wireframe

Snow: The Old Way

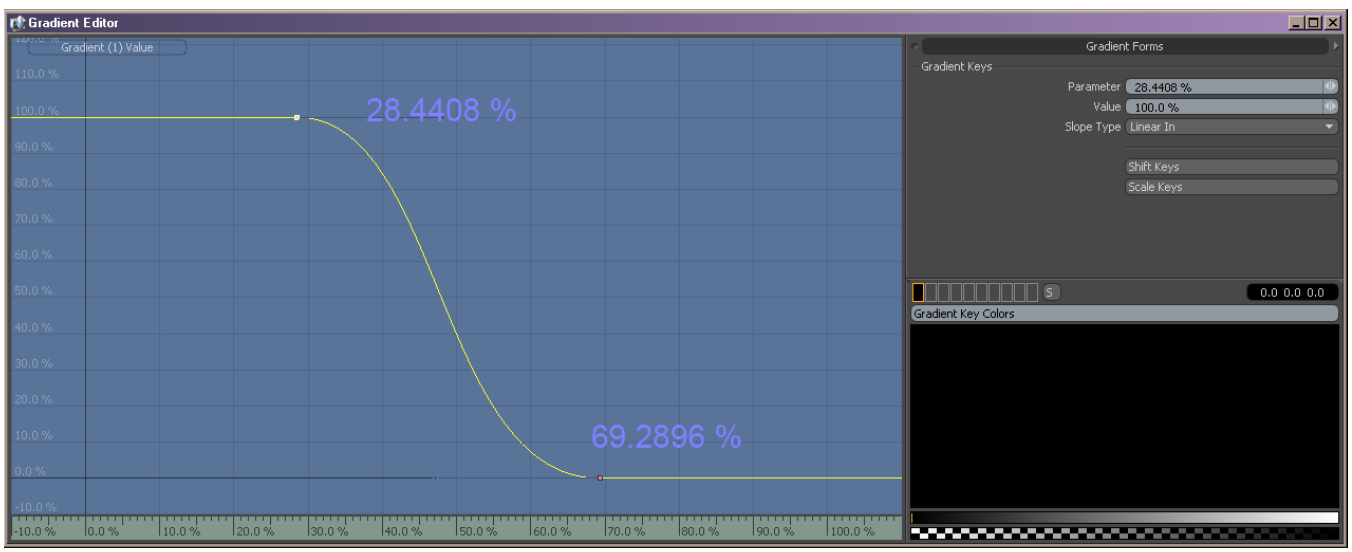
- Traditionally snow coverage was controlled via “coverage” textures
  - Placement textures controlling blending between snow and terrain textures
  - Cumbersome to author
  - Additional memory footprint
  - Not easily modifiable
  - Hard to adjust for dynamically generated geometry

Controlling Snow Accumulation

- Want snow accumulation to correlate to the objects - automatically
- Determine snow coverage procedurally
- Idea: use the combination of the geometric normal and the bump map normal to control snow coverage
  - With blending factors which control how we "accumulate" or "melt" snow
  - i.e. its appearance on the geometry (Eg: Mountain)
  - Depending on the geometric normal orientation

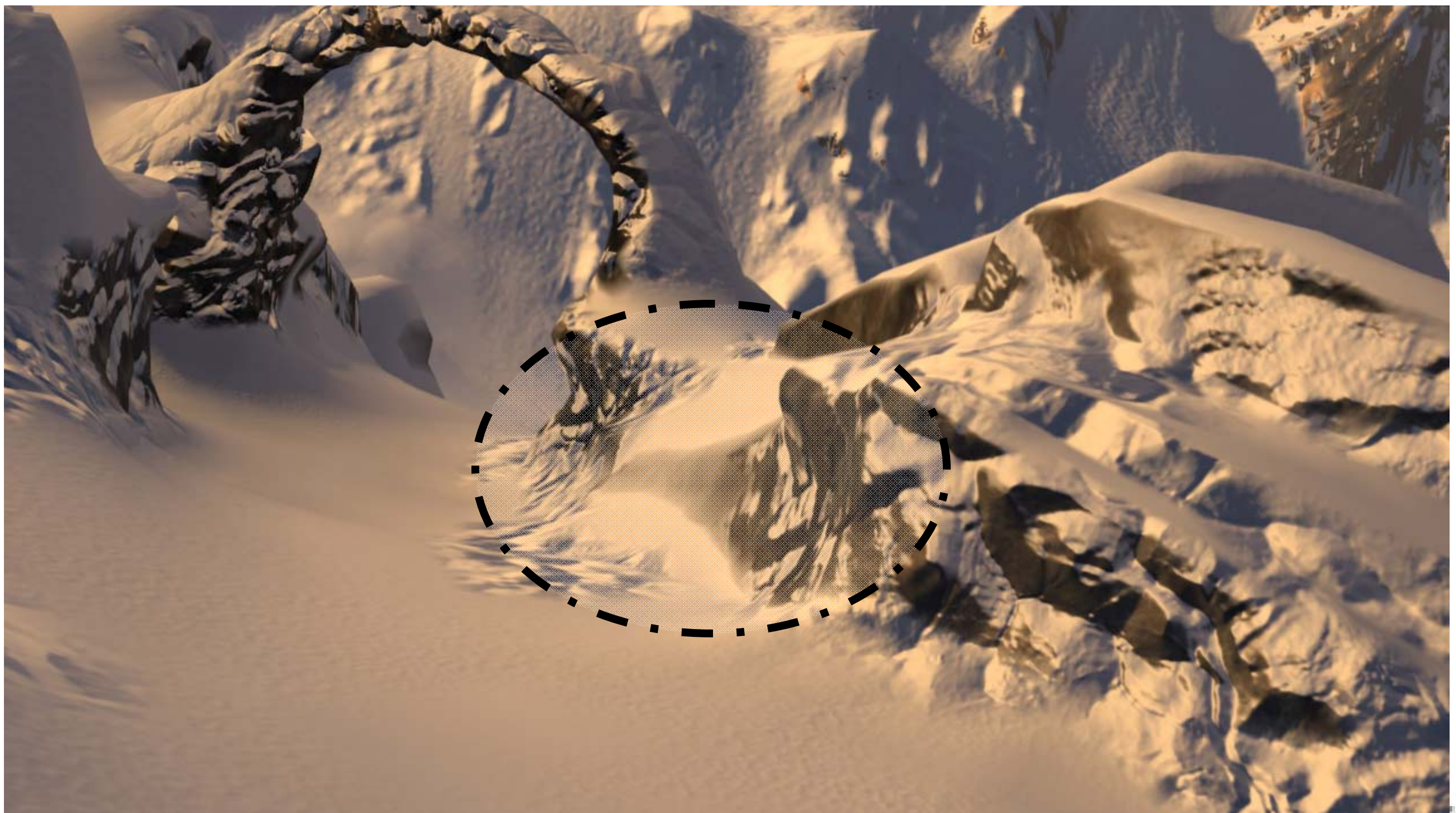
Snow Coverage

- Snow coverage determined procedurally
- Artists paint slope's control points as vertex colors
- Slope used for smoothstep to define snow masks
  - Mask 1: smoothstep based on *geometric normal's **y*** component
  - Mask 2: smoothstep based on *normal map's **y** component*
  - Final snow coverage mask = Mask1 * Mask2



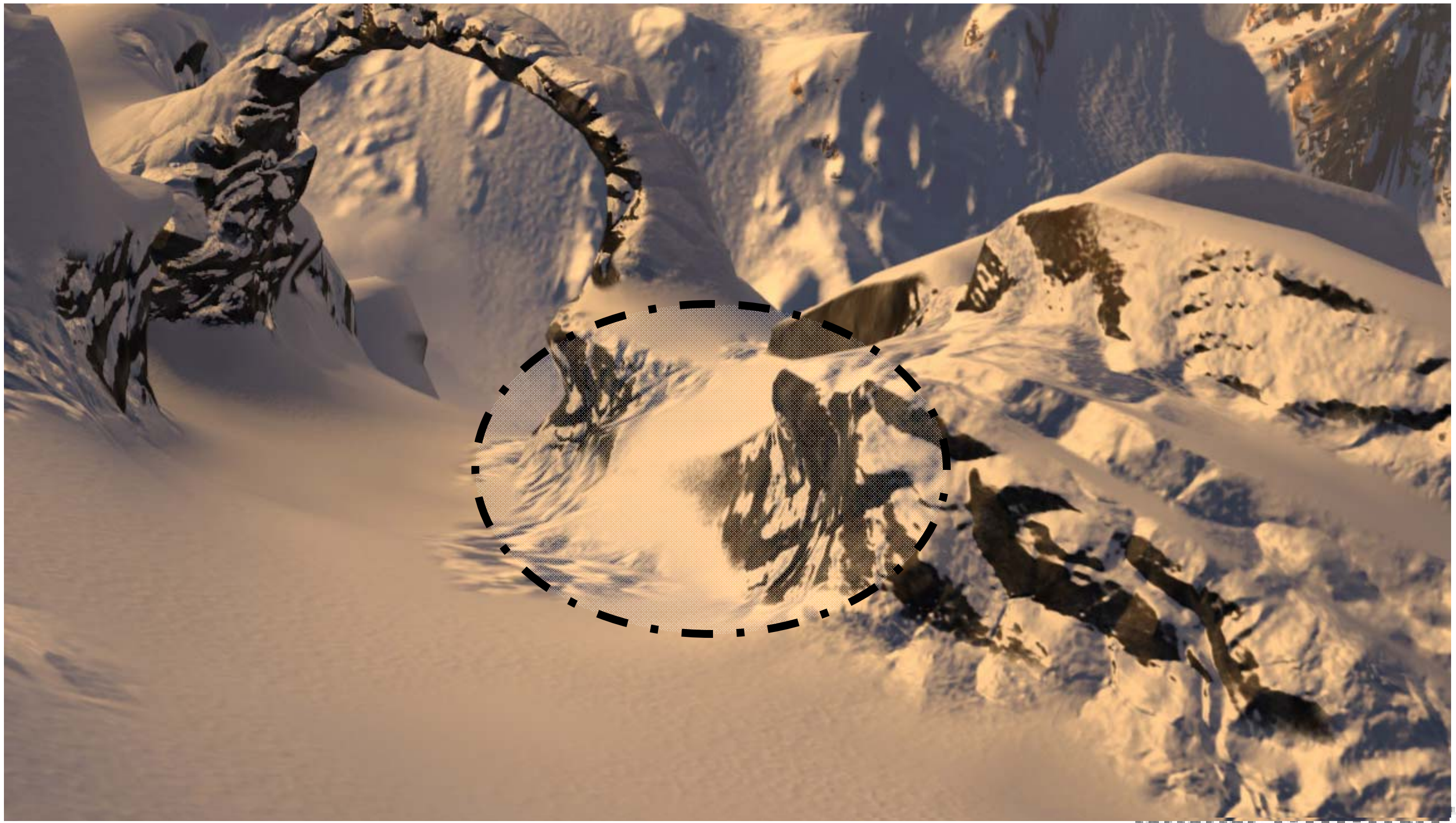
What If We Don't Use Noise?

- Straight-forward blend creates a sharp crease between snow and ground



Break Up the Monotony

- Use noise to adjust the blend between snow and rock for a natural transition



Demo

Conclusions

- Noise is crucial to interesting, high quality rendered images
  - A function that launched a thousand textures!
- Procedural computation of noise in real-time yields better quality
  - Fast on current hardware, even including earlier generations (PS 2.0 and beyond)
  - Particularly fast on latest hardware
- Must pay close attention to antialiasing
  - Use analytic prefiltering or frequency clamping whenever possible

A Round of Applause for These Folks!

- John Isidoro (some noise shaders and many valuable discussions, NoiseTexGen app)
- Chris Oat & Abe Wiley (snowy mountains)
- Thorsten Scheuermann, Jeremy Shopf, Dan Gessel-Abrahams & Josh Barczak for many fun discussions on any *random* topics ☺
- Bill Licea-Kane (OpenGL noise shaders)

References

[Ebert03]: David S. Ebert, F. Kenton Musgrave, Darwyn Peachy, Ken Perlin, and Steven Worley "Texturing and Modeling, A Procedural Approach" Third Edition, Morgan Kaufman Publishers.

[Gustafson04]: Stefan Gustavson, "Simplex Noise Demystified", Technical Report, Linkoping University, Sweden, December 6, 2004

[Hart01]: John C. Hart, "Perlin Noise Pixel Shaders", Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware, 2001, pp 87-94

[Lewis89]: J. P. Lewis . "Algorithms for Solid Noise Synthesis", SIGGRAPH 1989

[Perlin01]: Ken Perlin, "Noise Hardware", SIGGRAPH 2001 Course Notes, Real-Time Shading

[Perlin02]: Ken Perlin, "Improving Noise", Computer Graphics; Vol. 35 No. 3, 2001

[Perlin03]: Ken Perlin, "Implementing Improved Perlin Noise"; GPU Gems, pp 73-85

[Green05] Simon Green**, "**Implementing Improved Perlin Noise", GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley 2005

[Hensley05] Hensley, J. (UNC), Scheuermann, T., Coombe G. (UNC), Singh, M. (UNC), Lastra, A. (UNC) 2005. Fast Summed-Area Table Generation and its Applications. In *Proceedings of Eurographics '05*. SAT Bibtex [PDF] [Video]

[Apodaca99] **Advanced RenderMan: Creating CGI for Motion Pictures (The Morgan Kaufmann Series in Computer Graphics)** by Anthony A. Apodaca, Larry Gritz

[Ebert03] **Texturing & Modeling: A Procedural Approach, Third Edition (The Morgan Kaufmann Series in Computer Graphics)** by David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley

[Sander04] Sander, P. V., Tatarchuk, N., Mitchell, J. L. 2004. Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering. ATI Research Technical Report. August 2nd, 2004. Flow BibTex [PDF]

AMD Tools

- New release of AMD RenderMonkey:
  - www.ati.com/developer/rendermonkey.html
  - Parallax Occlusion mapping sample in the *Advanced* folder

Questions?

- natalya.tatarchuk@amd.com
- devrel@amd.com