Modified Noise for Evaluation on Graphics Hardware

Marc Olano

Computer Science and Electrical Engineering
University of Maryland, Baltimore County

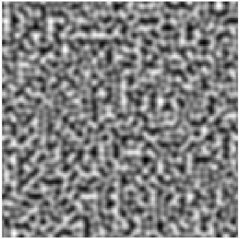
Graphics Hardware 2005

Outline

Outline

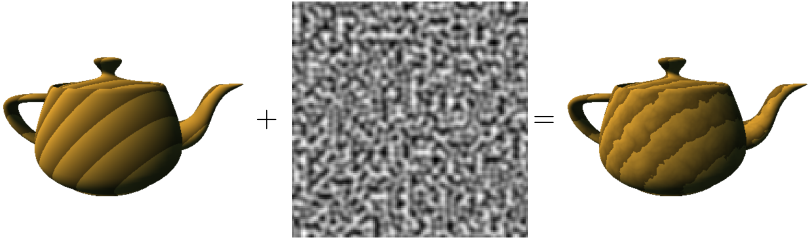
Why Noise?

- Introduced by [Perlin, 1985]
  - Heavily used in production animation
  - Technical Achievement Oscar in 1997
- "Salt," adds spice to shaders



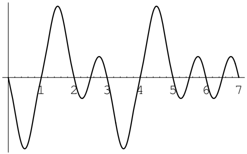
Why Noise?

- Introduced by [Perlin, 1985]
    - Heavily used in production animation
    - Technical Achievement Oscar in 1997
- "Salt," adds spice to shaders



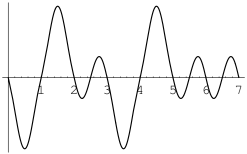
Noise Characteristics

- Random
    - No correlation between distant values
- Repeatable/deterministic
    - Same argument always produces same value
- Band-limited
    - Most energy in one octave (e.g. between f & 2f)



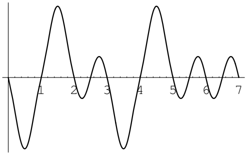
Noise Characteristics

- Random
  - No correlation between distant values
- Repeatable/deterministic
  - Same argument always produces same value
- Band-limited
  - Most energy in one octave (e.g. between f & 2f)



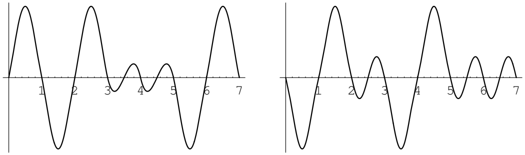
Noise Characteristics

- Random
  - No correlation between distant values
- Repeatable/deterministic
  - Same argument always produces same value
- Band-limited
  - Most energy in one octave (e.g. between f & 2f)



Gradient Noise

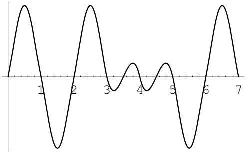
- Original Perlin noise [Perlin, 1985]
- Perlin Improved noise [Perlin, 2002]
- *Lattice* based
    - Value=0 at integer lattice points
    - Gradient defined at integer lattice
    - Interpolate between
- 1/2 to 1 cycle each unit



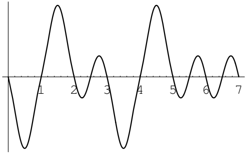
Original   Improved

Gradient Noise

- Original Perlin noise [Perlin, 1985]
- Perlin Improved noise [Perlin, 2002]
- *Lattice* based
  - Value=0 at integer lattice points
  - Gradient defined at integer lattice
  - Interpolate between
- 1/2 to 1 cycle each unit



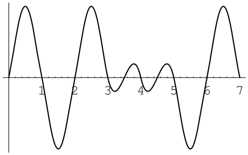
Original



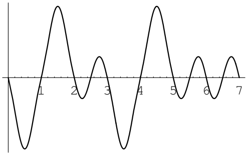
Improved

Gradient Noise

- Original Perlin noise [Perlin, 1985]
- Perlin Improved noise [Perlin, 2002]
- *Lattice* based
  - Value=0 at integer lattice points
  - Gradient defined at integer lattice
  - Interpolate between
- 1/2 to 1 cycle each unit



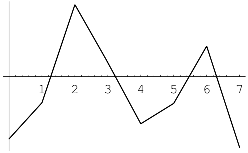
Original



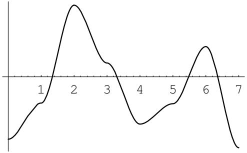
Improved

Value Noise

- Lattice based
  - Value defined at integer lattice points
  - Interpolate between
- At most 1/2 cycle each unit
  - Significant low-frequency content
- Easy hardware implementation with lower quality



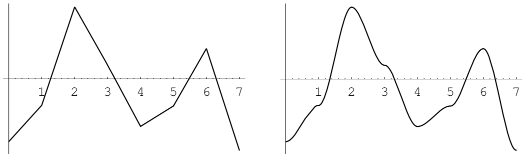
Linear Interp



Cubic Interp

Value Noise

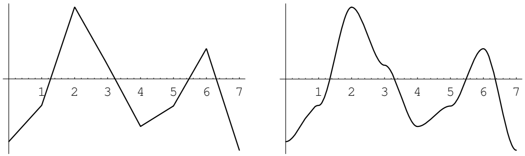
- Lattice based
  - Value defined at integer lattice points
  - Interpolate between
- At most 1/2 cycle each unit
  - Significant low-frequency content
- Easy hardware implementation with lower quality



Linear Interp   Cubic Interp

Value Noise

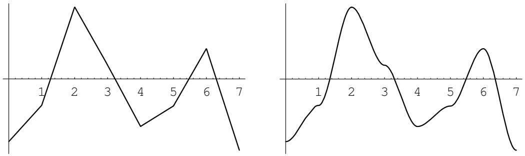
- Lattice based
  - Value defined at integer lattice points
  - Interpolate between
- At most 1/2 cycle each unit
  - Significant low-frequency content
- Easy hardware implementation with lower quality



Linear Interp   Cubic Interp

Value Noise

- Lattice based
  - Value defined at integer lattice points
  - Interpolate between
- At most 1/2 cycle each unit
  - Significant low-frequency content
- Easy hardware implementation with lower quality



Linear Interp   Cubic Interp

Hardware Noise

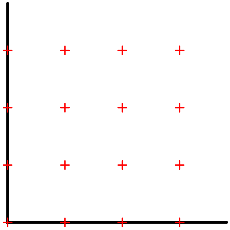
- Value noise
  - PixelFlow [Lastra et al., 1995]
  - *Perlin Noise* Pixel Shaders [Hart, 2001]
  - Noise textures
- Gradient noise
  - Hardware [Perlin, 2001]
  - Complex composition [Perlin, 2004]
  - Shader implementation [Green, 2005]

Noise Details

- Subclass of *gradient noise*
  - Original Perlin
  - Perlin Improved
  - All of our proposed modifications

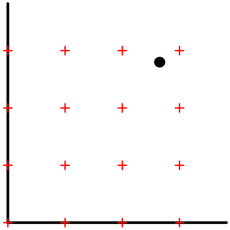
Find the Lattice

- Lattice-based noise: must find nearest lattice points
- Point $\vec{p} = (\vec{p}^x, \vec{p}^y, \vec{p}^z)$
- has integer lattice location
$\vec{p}_i = (\lfloor \vec{p}^x \rfloor, \lfloor \vec{p}^y \rfloor, \lfloor \vec{p}^z \rfloor) = (X, Y, Z)$
- and fractional location in cell
$\vec{p}_f = \vec{p} - \vec{p}_i = (x, y, z)$



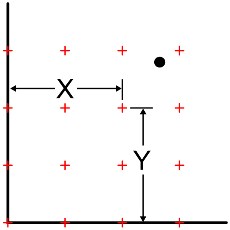
Find the Lattice

- Lattice-based noise: must find nearest lattice points
- Point $\vec{p} = (\vec{p}^x, \vec{p}^y, \vec{p}^z)$
- has integer lattice location
  $\vec{p}_i = (\lfloor \vec{p}^x \rfloor, \lfloor \vec{p}^y \rfloor, \lfloor \vec{p}^z \rfloor) = (X, Y, Z)$
- and fractional location in cell
  $\vec{p}_f = \vec{p} - \vec{p}_i = (x, y, z)$



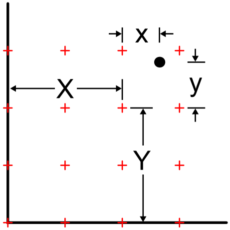
Find the Lattice

- Lattice-based noise: must find nearest lattice points
- Point $\vec{p} = (\vec{p}^x, \vec{p}^y, \vec{p}^z)$
- has integer lattice location
  $\vec{p}_i = (\lfloor \vec{p}^x \rfloor, \lfloor \vec{p}^y \rfloor, \lfloor \vec{p}^z \rfloor) = (X, Y, Z)$
- and fractional location in cell
  $\vec{p}_f = \vec{p} - \vec{p}_i = (x, y, z)$



Find the Lattice

- Lattice-based noise: must find nearest lattice points
- Point $\vec{p} = (\vec{p}^x, \vec{p}^y, \vec{p}^z)$
- has integer lattice location
  $\vec{p}_i = (\lfloor \vec{p}^x \rfloor, \lfloor \vec{p}^y \rfloor, \lfloor \vec{p}^z \rfloor) = (X, Y, Z)$
- and fractional location in cell
  $\vec{p}_f = \vec{p} - \vec{p}_i = (x, y, z)$



Gradient

- Random vector at each lattice point is a function of $\vec{p}_i$

$$g(\vec{p}_i)$$

- A function with that gradient

$$\begin{aligned} grad(\vec{p}) &= g(\vec{p}_i) \bullet \vec{p}_f \\ &= g^x(\vec{p}_i) * x + g^y(\vec{p}_i) * y + g^z(\vec{p}_i) * z \end{aligned}$$

Gradient

- Random vector at each lattice point is a function of $\vec{p}_i$

$$g(\vec{p}_i)$$

- A function with that gradient

$$grad(\vec{p}) = g(\vec{p}_i) \bullet \vec{p}_f$$
$$= g^x(\vec{p}_i) * x + g^y(\vec{p}_i) * y + g^z(\vec{p}_i) * z$$

Gradient

- Random vector at each lattice point is a function of $\vec{p}_i$

$$g(\vec{p}_i)$$

- A function with that gradient

$$\begin{aligned} grad(\vec{p}) &= g(\vec{p}_i) \bullet \vec{p}_f \\ &= g^x(\vec{p}_i) * x + g^y(\vec{p}_i) * y + g^z(\vec{p}_i) * z \end{aligned}$$

Interpolate

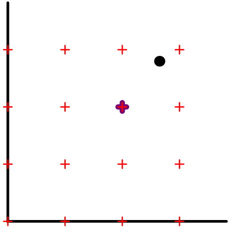
- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t,a,b) = (1-t)\ a + t\ b$
- Smooth interpolation

Interpolate

- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation

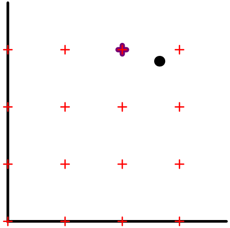
Interpolate

- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation



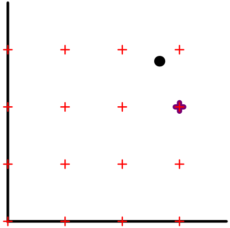
Interpolate

- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation



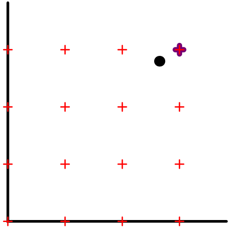
Interpolate

- Interpolate nearest 2^n gradient functions
- 2D *noise*$(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation



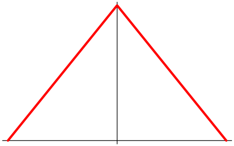
Interpolate

- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t,a,b) = (1-t)\ a + t\ b$
- Smooth interpolation
  - $fade(t) = \{$



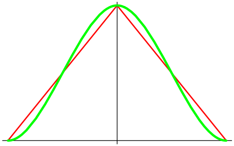
Interpolate

- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation
  - $fade(t) = \begin{cases} 3t^2 - 2t^3 & \text{for original noise} \\ \end{cases}$
  - $flerp(t, a, b) = lerp(fade(t), a, b)$



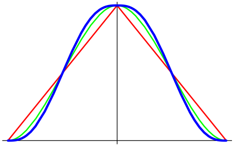
Interpolate

- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation
  - $fade(t) = \begin{cases} 3t^2 - 2t^3 & \text{for original noise} \\ 10t^3 - 15t^4 + 6t^5 & \text{for improved noise} \end{cases}$
  - $flerp(t, a, b) = lerp(fade(t), a, b)$



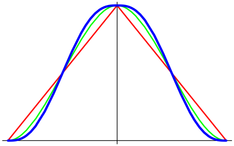
Interpolate

- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation
  - $fade(t) = \begin{cases} 3t^2 - 2t^3 & \text{for original noise} \\ 10t^3 - 15t^4 + 6t^5 & \text{for improved noise} \end{cases}$
  - $flerp(t, a, b) = lerp(fade(t), a, b)$



Interpolate

- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by
  $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation
  - $fade(t) = \begin{cases} 3t^2 - 2t^3 & \text{for original noise} \\ 10t^3 - 15t^4 + 6t^5 & \text{for improved noise} \end{cases}$
  - $flerp(t, a, b) = lerp(fade(t), a, b)$



Hash

- n-D gradient function built from 1D components

$$g(\vec{p}_i)$$

- Both original and improved use a permutation table *hash*
- Original: g is a table of unit vectors
- Improved: g is derived from bits of final hash

Hash

- n-D gradient function built from 1D components

$$g(hash(X, Y, Z))$$

- Both original and improved use a permutation table *hash*
- Original: g is a table of unit vectors
- Improved: g is derived from bits of final hash

Hash

- n-D gradient function built from 1D components

$$g(hash(Z + hash(Y + hash(X))))$$

- Both original and improved use a permutation table *hash*
- Original: g is a table of unit vectors
- Improved: g is derived from bits of final hash

Hash

- n-D gradient function built from 1D components

$$g(hash(Z + hash(Y + hash(X))))$$

- Both original and improved use a permutation table *hash*
- Original: g is a table of unit vectors
- Improved: g is derived from bits of final hash

Hash

- n-D gradient function built from 1D components

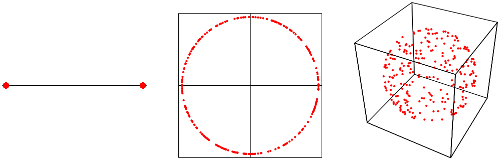
$$g(hash(Z + hash(Y + hash(X))))$$

- Both original and improved use a permutation table *hash*
- Original: g is a table of unit vectors
- Improved: g is derived from bits of final hash

Outline

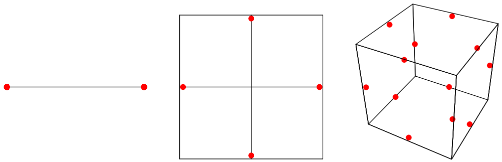
Gradient Vectors of n-D Noise

- Original: on the surface of a n-sphere
  - Found by hash of $\vec{p}_i$ into gradient table
- Improved: at the edges of an n-cube
  - Found by decoding bits of hash of $\vec{p}_i$



Gradient Vectors of n-D Noise

- Original: on the surface of a n-sphere
  - Found by hash of $\vec{p}_i$ into gradient table
- Improved: at the edges of an n-cube
  - Found by decoding bits of hash of $\vec{p}_i$

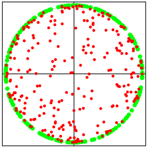


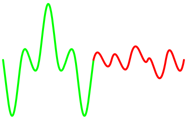
Gradients of noise(x,y,0) or noise(x,0)

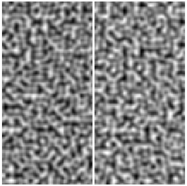
- Why?
  - Cheaper low-D noise matches slice of higher-D
  - Reuse textures (for full noise or partial computation)
- Original: new short gradient vectors
- Improved: gradients in new directions
  - Possibly including 0 gradient vector!

Gradients of noise(x,y,0) or noise(x,0)

- Why?
  - Cheaper low-D noise matches slice of higher-D
  - Reuse textures (for full noise or partial computation)
- Original: new short gradient vectors
- Improved: gradients in new directions
  - Possibly including 0 gradient vector!

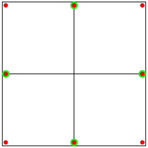


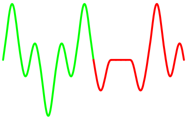


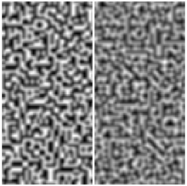


Gradients of noise(x,y,0) or noise(x,0)

- Why?
  - Cheaper low-D noise matches slice of higher-D
  - Reuse textures (for full noise or partial computation)
- Original: new short gradient vectors
- Improved: gradients in new directions
  - Possibly including 0 gradient vector!







Solution?

- Observe: use gradient function, not vector alone

$$grad = g^x\ x + g^y\ y + g^z\ z$$

- In any integer plane, fractional $z = 0$

$$grad = g^x\ x + g^y\ y + 0$$

- Any choice keeping projection of vectors the same will work
  - Improved noise uses cube edge centers

Solution?

- Observe: use gradient function, not vector alone

$$grad = g^x\ x + g^y\ y + g^z\ z$$

- In any integer plane, fractional $z = 0$

$$grad = g^x\ x + g^y\ y + 0$$

- Any choice keeping projection of vectors the same will work
  - Improved noise uses cube edge centers
  - Instead use cube corners!

Solution?

- Observe: use gradient function, not vector alone

$$grad = g^x\ x + g^y\ y + g^z\ z$$

- In any integer plane, fractional $z = 0$

$$grad = g^x\ x + g^y\ y + 0$$

- Any choice keeping projection of vectors the same will work
  - Improved noise uses cube edge centers
  - Instead use cube corners!

Solution?

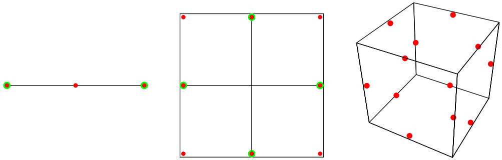
- Observe: use gradient function, not vector alone

$$grad = g^x\ x + g^y\ y + g^z\ z$$

- In any integer plane, fractional $z = 0$

$$grad = g^x\ x + g^y\ y + 0$$

- Any choice keeping projection of vectors the same will work
  - Improved noise uses cube edge centers
  - Instead use cube corners!



Solution?

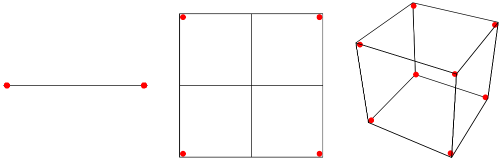
- Observe: use gradient function, not vector alone

$$grad = g^x\ x + g^y\ y + g^z\ z$$

- In any integer plane, fractional $z = 0$

$$grad = g^x\ x + g^y\ y + 0$$

- Any choice keeping projection of vectors the same will work
  - Improved noise uses cube edge centers
  - Instead use cube corners!

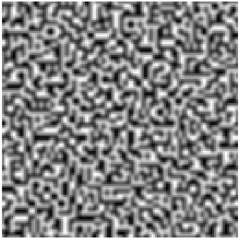


Corner Gradients

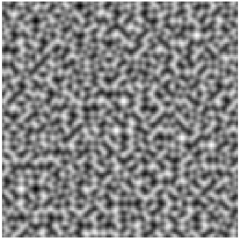
- Simple binary selection from hash bits
  $\pm x, \pm y, \pm z$
- Perlin mentions "clumping" for corner gradient selection
  - Not very noticeable in practice
  - Already happens in any integer plane of improved noise

Corner Gradients

- Simple binary selection from hash bits
  $\pm x, \pm y, \pm z$
- Perlin mentions "clumping" for corner gradient selection
  - Not very noticeable in practice
  - Already happens in any integer plane of improved noise



Edge Centers



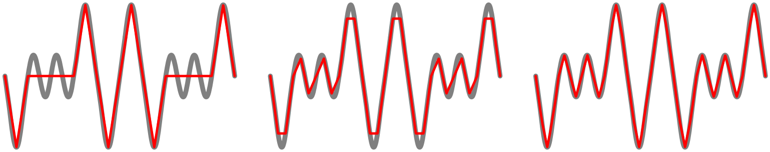
Corner

Separable Computation

- Like to store computation in texture
  - Texture sampling 3-4x highest frequency
  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures

Separable Computation

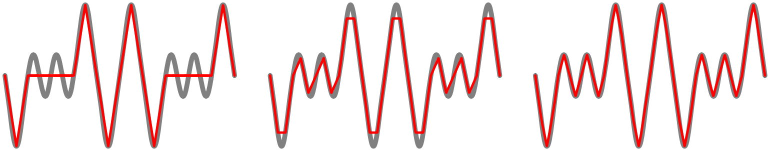
- Like to store computation in texture
  - Texture sampling 3-4x highest frequency



  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures
  - (e.g. write *noise*(p^x, p^y, p^z) as several x/y terms)

Separable Computation

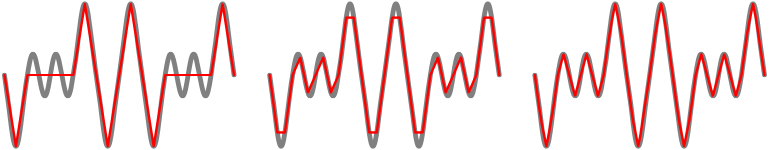
- Like to store computation in texture
  - Texture sampling 3-4x highest frequency



  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures
  - (e.g. write *noise*$(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)

Separable Computation

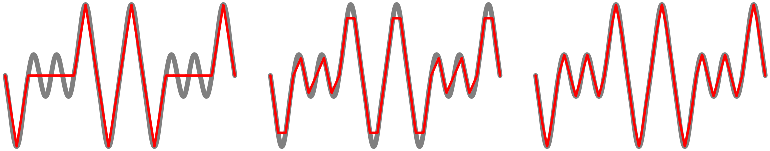
- Like to store computation in texture
  - Texture sampling 3-4x highest frequency



  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures
  - (e.g. write *noise*$(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)

Separable Computation

- Like to store computation in texture
  - Texture sampling 3-4x highest frequency

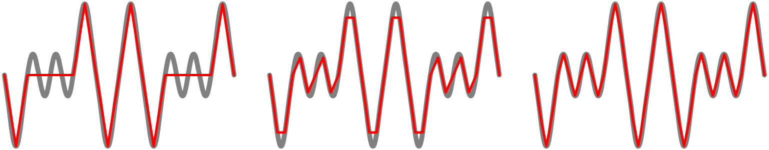


  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures
  - (e.g. write $noise(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)

$$noise(\vec{p}^x, \vec{p}^y, \vec{p}^z) = flerp(z, \text{xyz-term} + \text{xyz-term} * z \\ \text{xyz-term} + \text{xyz-term} * (z-1))$$

Separable Computation

- Like to store computation in texture
  - Texture sampling 3-4x highest frequency



  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures
  - (e.g. write $noise(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)

$$noise(\vec{p}^x, \vec{p}^y, \vec{p}^z) = flerp(z, \text{xy-term}(Z_0) + \text{xy-term}(Z_0) * z$$
$$\text{xy-term}(Z_1) + \text{xy-term}(Z_1) * (z - 1))$$

Factorization Details

$$noise(\vec{p}) = flerp(z, zconst(\vec{p}^x, \vec{p}^y, Z_0) + zgrad(\vec{p}^x, \vec{p}^y, Z_0) * z,$$
$$zconst(\vec{p}^x, \vec{p}^y, Z_1) + zgrad(\vec{p}^x, \vec{p}^y, Z_1) * (z-1))$$

- With nested hash,

$$zconst(\vec{p}^x, \vec{p}^y, Z_0) = zconst(\vec{p}^x, \vec{p}^y + hash(Z_0))$$
$$zgrad\ (\vec{p}^x, \vec{p}^y, Z_0) = zgrad\ (\vec{p}^x, \vec{p}^y + hash(Z_0))$$

- With corner gradients, $zconst = noise$!

Factorization Details

$$noise(\vec{p}) = flerp(z, zconst(\vec{p}^x, \vec{p}^y, Z_0) + zgrad(\vec{p}^x, \vec{p}^y, Z_0) * z, \\ zconst(\vec{p}^x, \vec{p}^y, Z_1) + zgrad(\vec{p}^x, \vec{p}^y, Z_1) * (z - 1))$$

- With nested hash,

$$zconst(\vec{p}^x, \vec{p}^y, Z_0) = zconst(\vec{p}^x, \vec{p}^y + hash(Z_0)) \\ zgrad(\vec{p}^x, \vec{p}^y, Z_0) = zgrad(\vec{p}^x, \vec{p}^y + hash(Z_0))$$

- With corner gradients, $zconst = noise$!

Factorization Details

$$
\begin{aligned}
noise(\vec{p}) = flerp(z, & zconst(\vec{p}^x, \vec{p}^y, Z_0) + zgrad(\vec{p}^x, \vec{p}^y, Z_0) * z, \\
& zconst(\vec{p}^x, \vec{p}^y, Z_1) + zgrad(\vec{p}^x, \vec{p}^y, Z_1) * (z - 1))
\end{aligned}
$$

- With nested hash,

$$
\begin{aligned}
zconst(\vec{p}^x, \vec{p}^y, Z_0) &= zconst(\vec{p}^x, \vec{p}^y + hash(Z_0)) \\
zgrad\,(\vec{p}^x, \vec{p}^y, Z_0) &= zgrad\,(\vec{p}^x, \vec{p}^y + hash(Z_0))
\end{aligned}
$$

- With corner gradients, $zconst = noise$!

Perlin's Hash

- 256-element *permutation array*
    - Turns each integer 0-255 into a different integer 0-255
- Chained lookups

$$g(hash(Z + hash(Y + hash(X))))$$

- Must compute for each lattice point around $\vec{p}$
- Even with a 2D $hash(Y + hash(X))$ texture, that's

Perlin's Hash

- 256-element *permutation array*
  - Turns each integer 0-255 into a different integer 0-255
- Chained lookups

$$g(hash(Z + hash(Y + hash(X))))$$

- Must compute for each lattice point around $\vec{p}$
- Even with a 2D $hash(Y + hash(X))$ texture, that's
  - 2 hash lookups for 1D noise

Perlin's Hash

- 256-element *permutation array*
  - Turns each integer 0-255 into a different integer 0-255
- Chained lookups

$$g(hash(Z + hash(Y + hash(X))))$$

- Must compute for each lattice point around $\vec{p}$
- Even with a 2D $hash(Y + hash(X))$ texture, that's
  - 2 hash lookups for 1D noise
  - 4 hash lookups for 2D noise
  - 12 hash lookups for 3D noise
  - 20 hash lookups for 4D noise

Perlin's Hash

- 256-element *permutation array*
  - Turns each integer 0-255 into a different integer 0-255
- Chained lookups

$$g(hash(Z + hash(Y + hash(X))))$$

- Must compute for each lattice point around $\vec{p}$
- Even with a 2D $hash(Y + hash(X))$ texture, that's
  - 2 hash lookups for 1D noise
  - 4 hash lookups for 2D noise
  - 12 hash lookups for 3D noise
  - 20 hash lookups for 4D noise

Perlin's Hash

- 256-element *permutation array*
  - Turns each integer 0-255 into a different integer 0-255
- Chained lookups

$$g(hash(Z + hash(Y + hash(X))))$$

- Must compute for each lattice point around $\vec{p}$
- Even with a 2D $hash(Y + hash(X))$ texture, that's
  - 2 hash lookups for 1D noise
  - 4 hash lookups for 2D noise
  - 12 hash lookups for 3D noise
  - 20 hash lookups for 4D noise

Perlin's Hash

- 256-element *permutation array*
  - Turns each integer 0-255 into a different integer 0-255
- Chained lookups

$$g(hash(Z + hash(Y + hash(X))))$$

- Must compute for each lattice point around $\vec{p}$
- Even with a 2D $hash(Y + hash(X))$ texture, that's
  - 2 hash lookups for 1D noise
  - 4 hash lookups for 2D noise
  - 12 hash lookups for 3D noise
  - 20 hash lookups for 4D noise

Perlin's Hash

- 256-element *permutation array*
  - Turns each integer 0-255 into a different integer 0-255
- Chained lookups

$$g(hash(Z + hash(Y + hash(X))))$$

- Must compute for each lattice point around $\vec{p}$
- Even with a 2D $hash(Y + hash(X))$ texture, that's
  - 2 hash lookups for 1D noise
  - 4 hash lookups for 2D noise
  - 12 hash lookups for 3D noise
  - 20 hash lookups for 4D noise

Perlin's Hash

- 256-element *permutation array*
  - Turns each integer 0-255 into a different integer 0-255
- Chained lookups

$$g(hash(Z + hash(Y + hash(X))))$$

- Must compute for each lattice point around $\vec{p}$
- Even with a 2D $hash(Y + hash(X))$ texture, that's
  - 2 hash lookups for 1D noise
  - 4 hash lookups for 2D noise
  - 12 hash lookups for 3D noise
  - 20 hash lookups for 4D noise

Alternative Hash

- Many choices; I kept 1D chaining
- Desired features
  - Low correlation of hash output for nearby inputs
  - Computable without lookup
- Use a random number generator?
  - Seed
  - Successive calls give uncorrelated values

Alternative Hash

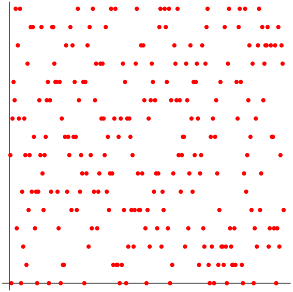
- Many choices; I kept 1D chaining
- Desired features
  - Low correlation of hash output for nearby inputs
  - Computable without lookup
- Use a random number generator?
  - Seed
  - Successive calls give uncorrelated values

Alternative Hash

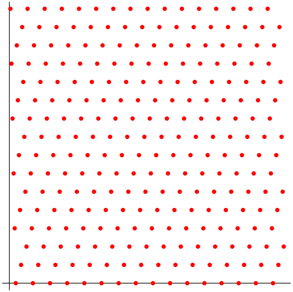
- Many choices; I kept 1D chaining
- Desired features
  - Low correlation of hash output for nearby inputs
  - Computable without lookup
- Use a random number generator?
  - Seed
  - Successive calls give uncorrelated values

Random Number Generator Hash

- Hash argument is seed
  - Most RNG are highly correlated for nearby seeds
- Hash argument is number of times to call
  - Most RNG are expensive (or require n calls) to get n^{th} number
  - Should noise(30) be 30 times slower than noise(1)?



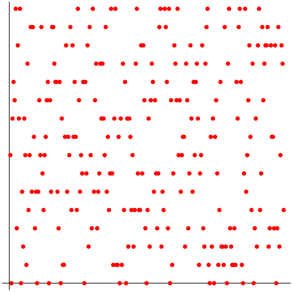
permute table



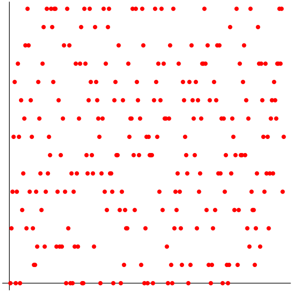
hash using seed=X

Random Number Generator Hash

- Hash argument is seed
  - Most RNG are highly correlated for nearby seeds
- Hash argument is number of times to call
  - Most RNG are expensive (or require n calls) to get n^{th} number
  - Should noise(30) be 30 times slower than noise(1)?



permute table

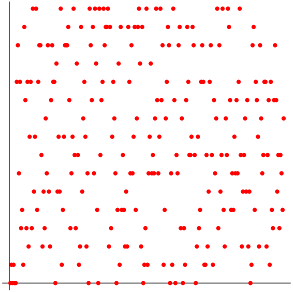


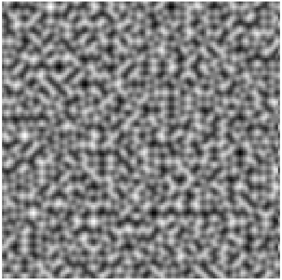
hash using X^{th} random number

Blum-Blum Shub

$$x_{n+1} = x_i^2 \text{ mod } M$$
$$M = \text{product of two large primes}$$

- Uncorrelated for nearby seeds...
- But large M is bad for hardware...
- But reasonable results for smaller M...
- And square and mod is simple to compute!



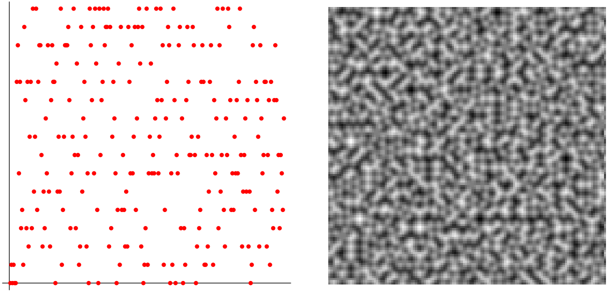


523*527

Blum-Blum Shub

$$x_{n+1} = x_i^2 \bmod M$$
$$M = \text{product of two large primes}$$

- Uncorrelated for nearby seeds...
- But large M is bad for hardware...
- But reasonable results for smaller M...
- And square and mod is simple to compute!

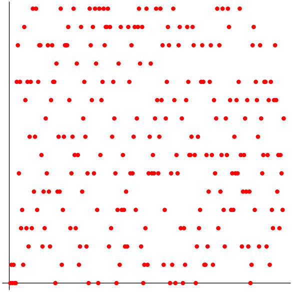
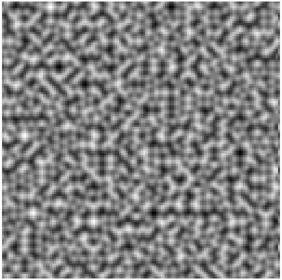


523*527

Blum-Blum Shub

$$x_{n+1} = x_i^2 \bmod M$$
$$M = \text{product of two large primes}$$

- Uncorrelated for nearby seeds...
- But large M is bad for hardware...
- But reasonable results for smaller M...
- And square and mod is simple to compute!

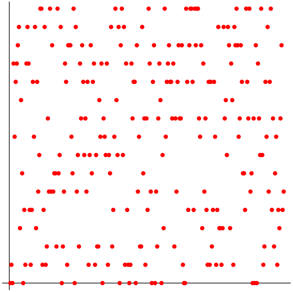



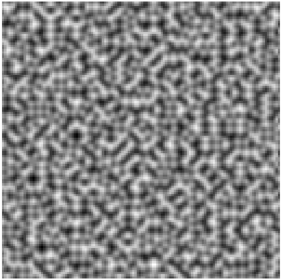
523*527

Blum-Blum Shub

$$x_{n+1} = x_i^2 \bmod M$$
$$M = \text{product of two large primes}$$

- Uncorrelated for nearby seeds...
- But large M is bad for hardware...
- But reasonable results for smaller M...
- And square and mod is simple to compute!



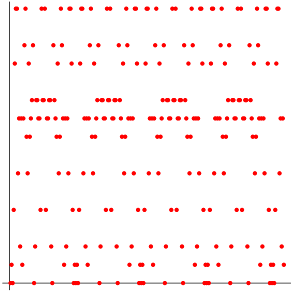


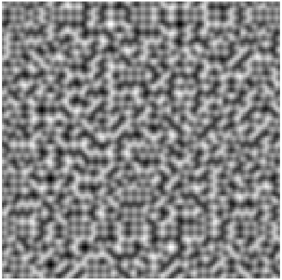
29*31

Blum-Blum Shub

$$x_{n+1} = x_i^2 \bmod M$$
$$M = \text{product of two large primes}$$

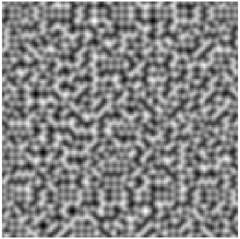
- Uncorrelated for nearby seeds...
- But large M is bad for hardware...
- But reasonable results for smaller M...
- And square and mod is simple to compute!



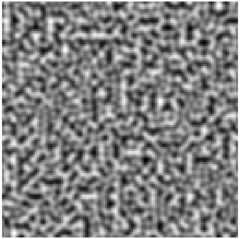


Modified Noise

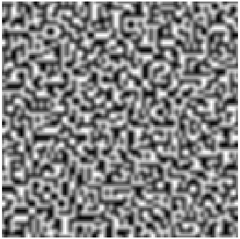
- Square and mod hash
  - $M = 61$
- Corner gradient selection
  - One 2D texture for both 1D and 2D
- Factor
  - Construct 3D and 4D from 2 or 4 2D texture lookups



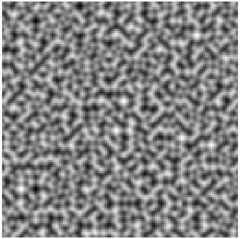
Comparison



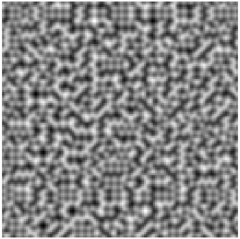
Perlin original



Perlin improved



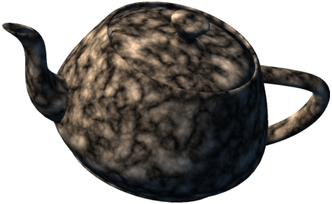
Corner gradients



Corner+Hash

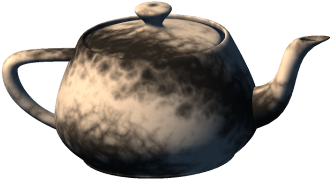
Using Noise

3D noise



3D turbulence

Wood



Marble

Outline

Introduction & Background

Modifications

Conclusion

Conclusions

- Three (mostly) independent modifications to Perlin noise
  - Corner gradient: can subset noise
    - noise(x) = noise(x,0)
    - noise(x,y) = noise(x,y,0)
  - Factorization: can superset noise
    - build 3D noise out of 2D
    - build 4D noise out of 3D
  - Computed hash
    - lookup-free noise
    - avoid potentially costly chained lookups
- Admit a range of choices for texture vs. compute

Conclusions

- Three (mostly) independent modifications to Perlin noise
  - Corner gradient: can subset noise
    - noise(x) = noise(x,0)
    - noise(x,y) = noise(x,y,0)
  - Factorization: can superset noise
    - build 3D noise out of 2D
    - build 4D noise out of 3D
  - Computed hash
    - lookup-free noise
    - avoid potentially costly chained lookups
- Admit a range of choices for texture vs. compute

Conclusions

- Three (mostly) independent modifications to Perlin noise
  - Corner gradient: can subset noise
    - noise(x) = noise(x,0)
    - noise(x,y) = noise(x,y,0)
  - Factorization: can superset noise
    - build 3D noise out of 2D
    - build 4D noise out of 3D
  - Computed hash
    - lookup-free noise
    - avoid potentially costly chained lookups
- Admit a range of choices for texture vs. compute

Conclusions

- Three (mostly) independent modifications to Perlin noise
  - Corner gradient: can subset noise
    - noise(x) = noise(x,0)
    - noise(x,y) = noise(x,y,0)
  - Factorization: can superset noise
    - build 3D noise out of 2D
    - build 4D noise out of 3D
  - Computed hash
    - lookup-free noise
    - avoid potentially costly chained lookups
- Admit a range of choices for texture vs. compute

Future Work

- Other computed hash functions?
- Extend to simplex noise
- Extend to other hash-based primitives
  - Tiled texture
  - Worley cellular textures
- Further explore turbulence & fBm
  - Can we pre-bake the octaves together?

Questions?

www.umbc.edu/~olano/noise

Green, S. (2005).
Implementing improved Perlin noise.
In Pharr, M., editor, *GPU Gems 2*, chapter 26.
Addison-Wesley.

Hart, J. C. (2001).
Perlin noise pixel shaders.
In Akeley, K. and Neumann, U., editors, *Graphics Hardware 2001*, pages 87–94, Los Angeles, CA.
SIGGRAPH/EUROGRAPHICS, ACM, New York.

Lastra, A., Molnar, S., Olano, M., and Wang, Y. (1995).
Real-time programmable shading.
In *I3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*. ACM Press.

Perlin, K. (1985).
An image synthesizer.

In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press.

Perlin, K. (2001).
Noise hardware.
In Olano, M., editor, *Real-Time Shading SIGGRAPH Course Notes*.

Perlin, K. (2002).
Improving noise.
In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682. ACM Press.

Perlin, K. (2004).
Implementing improved Perlin noise.
In Fernando, R., editor, *GPU Gems*, chapter 5. Addison-Wesley.