Modified Noise for Evaluation on Graphics Hardware

Marc Olano

Computer Science and Electrical Engineering
University of Maryland, Baltimore County

Graphics Hardware 2005
Outline

Introduction & Background

Modifications

Conclusion
Outline

Introduction & Background
  Noise?
  Perlin noise

Modifications

Conclusion
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
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 vs Improved Noise](image)
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
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](image1)

![Cubic Interp](image2)
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
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
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](chart1)
![Cubic Interp](chart2)
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 $\vec{p}_i = ([\vec{p}^x], [\vec{p}^y], [\vec{p}^z]) = (X, Y, Z)$
  - 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
    $\vec{p}_i = ([\vec{p}^x], [\vec{p}^y], [\vec{p}^z]) = (X, Y, Z)$
  - 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 $\mathbf{p} = (\mathbf{p}^x, \mathbf{p}^y, \mathbf{p}^z)$
- has integer lattice location $\mathbf{p}_i = (\lfloor \mathbf{p}^x \rfloor, \lfloor \mathbf{p}^y \rfloor, \lfloor \mathbf{p}^z \rfloor) = (X, Y, Z)$
- and fractional location in cell $\mathbf{p}_f = \mathbf{p} - \mathbf{p}_i = (x, y, z)$
Gradient

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

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

• A function with that gradient

  $$\text{grad}(\vec{p}) = g(\vec{p}_i) \cdot \vec{p}_f$$

  $$= g^x(\vec{p}_i) \cdot x + g^y(\vec{p}_i) \cdot y + g^z(\vec{p}_i) \cdot z$$
Gradient

- Random vector at each lattice point is a function of \( \vec{p}_i \)
  \[
g(\vec{p}_i)
  \]
- A function with that gradient
  \[
  \text{grad}(\vec{p}) = g(\vec{p}_i) \cdot \vec{p}_f \\
  = g^x(\vec{p}_i) \cdot x + g^y(\vec{p}_i) \cdot y + g^z(\vec{p}_i) \cdot z
  \]
Gradient

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

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

- A function with that gradient

$$\text{grad}(\vec{p}) = g(\vec{p}_i) \bullet \vec{p}_f$$

$$= g^x(\vec{p}_i) \star x + g^y(\vec{p}_i) \star y + g^z(\vec{p}_i) \star z$$
Interpolate

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

- Interpolate nearest $2^n$ gradient functions
- 2D $\text{noise}(\vec{p})$ 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
  - $\text{lerp}(t, a, b) = (1 - t) \cdot a + t \cdot b$
- Smooth interpolation
Interpolate

- Interpolate nearest $2^n$ gradient functions
- 2D $\text{noise}(\vec{p})$ 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
  - $\text{lerp}(t, a, b) = (1 - t) \ a + t \ b$
- Smooth interpolation
• Interpolate nearest $2^n$ gradient functions
• 2D $\text{noise}(\vec{p})$ 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
  • $\text{lerp}(t, a, b) = (1 - t) \ a + t \ b$
• Smooth interpolation
Interpolate

- Interpolate nearest $2^n$ gradient functions
- 2D $\text{noise}(\vec{p})$ 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
  - $\text{lerp}(t, a, b) = (1 - t) \ a + t \ b$
- Smooth interpolation
Interpolate

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

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

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

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

- Interpolate nearest $2^n$ gradient functions
- 2D $\text{noise}(\vec{p})$ 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
  - $\text{lerp}(t, a, b) = (1 - t) a + t b$
- Smooth interpolation
  - $\text{fade}(t) = \begin{cases} 
  3t^2 - 2t^3 & \text{for original noise} \\
  10t^3 - 15t^4 + 6t^5 & \text{for improved noise} 
\end{cases}$
  - $\text{flerp}(t) = \text{lerp}(\text{fade}(t), a, b)$
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
  \[ g(\text{hash}(Z + \text{hash}(X, Y))) \]

- Both original and improved use a permutation table \text{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

\[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]

- Both original and improved use a permutation table \( \text{hash} \)
- 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

\[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]

- Both original and improved use a permutation table \text{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

\[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]

- Both original and improved use a permutation table \text{hash}
- Original: \( g \) is a table of unit vectors
- Improved: \( g \) is derived from bits of final hash
Outline

Introduction & Background

Modifications
  Corner Gradients
  Factorization
  Hash

Conclusion
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 $\bar{p}_i$ into gradient table
- **Improved**: at the edges of an n-cube
  - Found by decoding bits of hash of $\bar{p}_i$
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

  \[ \text{grad} = g^x x + g^y y + g^z z \]

- In any integer plane, fractional \( z = 0 \)

  \[ \text{grad} = g^x x + g^y y + 0 \]

- Any choice keeping *projection* of vectors the same will work

  - Improved noise uses cube edge centers
Solution?

- Observe: use *gradient function*, not vector alone
  
  \[ \text{grad} = g^x \ x + g^y \ y + g^z \ z \]

- In any integer plane, fractional \( z = 0 \)
  
  \[ \text{grad} = g^x \ x + g^y \ y + 0 \]

- 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

\[ \text{grad} = g^x \, x + g^y \, y + g^z \, z \]

- In any integer plane, fractional \( z = 0 \)

\[ \text{grad} = g^x \, x + g^y \, y + 0 \]

- 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

\[ \text{grad} = g^x x + g^y y + g^z z \]

- In any integer plane, fractional \( z = 0 \)

\[ \text{grad} = g^x x + g^y y + 0 \]

- 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

\[ \text{grad} = g^x \times x + g^y \times y + g^z \times z \]

- In any integer plane, fractional \( z = 0 \)

\[ \text{grad} = g^x \times x + g^y \times y + 0 \]

- 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 \( \text{noise}(\vec{p}_x, \vec{p}_y, \vec{p}_z) \) as several 2D terms)
  
  \[
  \text{noise}(\vec{p}_x, \vec{p}_y, \vec{p}_z) = \text{flerp}(z, z + 1) \\
  \text{flerp}(2 + z, 2 + z + 1)
  \]
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 $\text{noise}(\vec{p}_x, \vec{p}_y, \vec{p}_z)$ as several x/y terms)

$$\text{noise}(\vec{p}_x, \vec{p}_y, \vec{p}_z) = \text{flerp}(z, + \ast z + \ast (z - 1))$$
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 $\text{noise}(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)
  - $\text{noise}(\vec{p}^x, \vec{p}^y, \vec{p}^z) = \text{flerp}(z, + \ast z + \ast (z - 1))$
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 $\text{noise}(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several $x/y$ terms)

\[
\text{noise}(\vec{p}^x, \vec{p}^y, \vec{p}^z) = \text{flerp}(z, \text{xyz-term} + \text{xyz-term} \times z \\
\text{xyz-term} + \text{xyz-term} \times (z - 1))
\]
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 $\text{noise}(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)

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

\[
\text{noise}(\vec{p}) = \text{flerp}(z, \text{zconst}(\vec{p}^x, \vec{p}^y, Z_0) + \text{zgrad}(\vec{p}^x, \vec{p}^y, Z_0) \ast z, \text{zconst}(\vec{p}^x, \vec{p}^y, Z_1) + \text{zgrad}(\vec{p}^x, \vec{p}^y, Z_1) \ast (z - 1))
\]

- With nested hash,
  \[
  \text{zconst}(\vec{p}^x, \vec{p}^y, Z_0) = \text{zconst}(\vec{p}^x, \vec{p}^y + \text{hash}(Z_0))
  \]
  \[
  \text{zgrad}(\vec{p}^x, \vec{p}^y, Z_0) = \text{zgrad}(\vec{p}^x, \vec{p}^y + \text{hash}(Z_0))
  \]

- With corner gradients, \( \text{zconst} = \text{noise} \! \)
Factorization Details

\[ \text{noise}(\vec{p}) = \text{flerp}(z,z\text{const}(\vec{p}^x, \vec{p}^y, Z_0) + z\text{grad}(\vec{p}^x, \vec{p}^y, Z_0) \ast z, \]
\[ z\text{const}(\vec{p}^x, \vec{p}^y, Z_1) + z\text{grad}(\vec{p}^x, \vec{p}^y, Z_1) \ast (z - 1)) \]

- With nested hash,

\[ z\text{const}(\vec{p}^x, \vec{p}^y, Z_0) = z\text{const}(\vec{p}^x, \vec{p}^y + \text{hash}(Z_0)) \]
\[ z\text{grad}(\vec{p}^x, \vec{p}^y, Z_0) = z\text{grad}(\vec{p}^x, \vec{p}^y + \text{hash}(Z_0)) \]

- With corner gradients, \( z\text{const} = \text{noise}! \)
Factorization Details

\[ \text{noise}(\vec{p}) = \text{flerp}(z, \text{zconst}(\vec{p}^x, \vec{p}^y, Z_0) + \text{zgrad}(\vec{p}^x, \vec{p}^y, Z_0) \ast z, \]
\[ \text{zconst}(\vec{p}^x, \vec{p}^y, Z_1) + \text{zgrad}(\vec{p}^x, \vec{p}^y, Z_1) \ast (z - 1)) \]

- With nested hash,

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

- With corner gradients, \( \text{zconst} = \text{noise}! \)
Perlin's Hash

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

• Chained lookups
  \[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]

• Must compute for each lattice point around \( \vec{p} \)
• Even with a 2D \( \text{hash}(Y + \text{hash}(X)) \) texture, that’s
Perlin’s Hash

- 256-element permutation array
  - Turns each integer 0-255 into a different integer 0-255
- Chained lookups
  \[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]
- Must compute for each lattice point around \( \vec{p} \)
- Even with a 2D \( \text{hash}(Y + \text{hash}(X)) \) 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
  \[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]
- Must compute for each lattice point around \( \vec{p} \)
- Even with a 2D \( \text{hash}(Y + \text{hash}(X)) \) 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
  \[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]
- Must compute for each lattice point around \( \vec{p} \)
- Even with a 2D \( \text{hash}(Y + \text{hash}(X)) \) 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
  \[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]
• Must compute for each lattice point around \( \vec{p} \)
• Even with a 2D \( \text{hash}(Y + \text{hash}(X)) \) 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
  \[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) ) \]
- Must compute for each lattice point around \( \vec{p} \)
- Even with a 2D \( \text{hash}(Y + \text{hash}(X)) \) 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
  
  \[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]

- Must compute for each lattice point around \( \vec{p} \)
- Even with a 2D \( \text{hash}(Y + \text{hash}(X)) \) 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
  \[ g(\text{hash}(Z + \text{hash}(Y + \text{hash}(X)))) \]
- Must compute for each lattice point around \( \vec{p} \)
- Even with a 2D \( \text{hash}(Y + \text{hash}(X)) \) 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

\[ x_{n+1} = x_i^2 \mod M \]

\[ M = \text{product of two large primes} \]

- 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

\[ x_{n+1} = x_i^2 \mod M \]

\[ M = \text{product of two large primes} \]

- 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!
Blum-Blum Shub

\[ x_{n+1} = x_i^2 \mod M \]

\[ M = \text{product of two large primes} \]

- 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!
Blum-Blum Shub

\[ x_{n+1} = x_i^2 \mod M \]

\[ M = \text{product of two large primes} \]

- 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!
Blum-Blum Shub

\[ x_{n+1} = x_i^2 \mod M \]

\[ M = \text{product of two large primes} \]

- 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!
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
    - $\text{noise}(x) = \text{noise}(x,0)$
    - $\text{noise}(x,y) = \text{noise}(x,y,0)$
  - 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
    - \( \text{noise}(x) = \text{noise}(x,0) \)
    - \( \text{noise}(x,y) = \text{noise}(x,y,0) \)
  - 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


