GPU Shading and Rendering: Introduction & Graphics Hardware

Marc Olano

Computer Science and Electrical Engineering
University of Maryland, Baltimore County

SIGGRAPH 2005

Schedule

Shading Technolgy

8:30	Intro/Hardware (Olano)
9:25	Compilers (Bleiweiss)

Shading Languages

10:30	GLSL (Olano)
10:55	Cg (Kilgard)
11:20	HLSL (Sander)
11:45	Sh (McCool)

GPU Rendering

1:45	Rendering Algorithms (Hart)
2:35	GPU Production Rendering (Gritz)

Hardware Systems

3:45	ATI (Sander)
4:25	NVIDIA (Kilgard)
5:05	Panel Q&A (all)

Part I

Introdution

What is a GPU?

- Graphics Processing Unit
  - Graphics accelerator
  - Parallel processing unit
- We're doing graphics, what is it good for?
  - Better real-time graphics
  - Faster non-real-time graphics

What is a GPU?

- Graphics Processing Unit
  - Graphics accelerator
  - Parallel processing unit
- We're doing graphics, what is it good for?
  - Better real-time graphics
  - Faster non-real-time graphics

What is a GPU?

- Graphics Processing Unit
  - Graphics accelerator
  - Parallel processing unit
- We're doing graphics, what is it good for?
  - Better real-time graphics
  - Faster non-real-time graphics

What is a GPU?

- Graphics Processing Unit
  - Graphics accelerator
  - Parallel processing unit
- We're doing graphics, what is it good for?
  - Better real-time graphics
  - Faster non-real-time graphics

What is Shading?

- What color are the pixels
- Programmable
    - Flexible Appearance
    - Arbitrary computation
- Procedural

What is Shading?

- What color are the pixels
- Programmable
  - Flexible Appearance
  - Arbitrary computation
- Procedural

What is Shading?

- What color are the pixels
- Programmable
    - Flexible Appearance
    - Arbitrary computation
- Procedural
    - Simple procedures

What is Shading?

- What color are the pixels
- Programmable
  - Flexible Appearance
  - Arbitrary computation
- Procedural
  - Simple procedures
  - High-level language

What is Shading?

- What color are the pixels
- Programmable
  - Flexible Appearance
  - Arbitrary computation
- Procedural
  - Simple procedures
  - High-level language

What is Shading?

- What color are the pixels
- Programmable
  - Flexible Appearance
  - Arbitrary computation
- Procedural
  - Simple procedures
  - High-level language

What is Shading?

- What color are the pixels
- Programmable
  - Flexible Appearance
  - Arbitrary computation
- Procedural
  - Simple procedures
  - High-level language

Some examples

- More realistic appearance
  - Bump mapping, Anisotropic, Precomputed radiance transfer, ...
- Non-realistic appearance
  - Cartoon, Sketch, Illustration, ...
- Animated appearance
  - Skinning, Water, Clouds, ...
- Visualization
  - Data on surfaces, Volume rendering, ...

Some examples

- More realistic appearance
  - Bump mapping, Anisotropic, Precomputed radiance transfer, ...
- Non-realistic appearance
  - Cartoon, Sketch, Illustration, ...
- Animated appearance
  - Skinning, Water, Clouds, ...
- Visualization
  - Data on surfaces, Volume rendering, ...

Some examples

- More realistic appearance
  - Bump mapping, Anisotropic, Precomputed radiance transfer, ...
- Non-realistic appearance
  - Cartoon, Sketch, Illustration, ...
- Animated appearance
  - Skinning, Water, Clouds, ...
- Visualization
  - Data on surfaces, Volume rendering, ...

Some examples

- More realistic appearance
  - Bump mapping, Anisotropic, Precomputed radiance transfer, ...
- Non-realistic appearance
  - Cartoon, Sketch, Illustration, ...
- Animated appearance
  - Skinning, Water, Clouds, ...
- Visualization
  - Data on surfaces, Volume rendering, ...

What is Rendering?

- The rest of the problem!
- In our case, using GPU for other than polygon rendering
  - Curved surfaces
  - Ray tracing
  - Point based rendering
  - ...

What is Rendering?

- The rest of the problem!
- In our case, using GPU for other than polygon rendering
  - Curved surfaces
  - Ray tracing
  - Point based rendering
  - ...

What is Rendering?

- The rest of the problem!
- In our case, using GPU for other than polygon rendering
  - Curved surfaces
  - Ray tracing
  - Point based rendering
  - ...

Non-Real Time vs. Real-Time

Real-Time

Tens of frames per second

Thousand instruction shaders

Limited computation, texture, memory, ...

Non-Real-Time

Seconds to hours per frame

Thousands line shaders

"Unlimited" computation, texture, memory, ...

Non-Real Time vs. Real-Time

Real-Time

Tens of frames per second

Thousand instruction shaders

Limited computation, texture, memory, ...

Non-Real-Time

Seconds to hours per frame

Thousands line shaders

"Unlimited" computation, texture, memory, ...

Non-Real Time vs. Real-Time

Real-Time

Tens of frames per second

Thousand instruction shaders

Limited computation, texture, memory, ...

Non-Real-Time

Seconds to hours per frame

Thousands line shaders

"Unlimited" computation, texture, memory, ...

Non-Real Time vs. Real-Time

Real-Time
Tens of frames per second
Thousand instruction shaders
Limited computation, texture, memory, ...

Non-Real-Time
Seconds to hours per frame
Thousands line shaders
"Unlimited" computation, texture, memory, ...

How is this possible?

- GPUs are programmable!
  - Per-vertex programs
  - Per-fragment programs

Research Languages

- Pixel-Planes 5 [Rhoades et al., 1992]
- PixelFlow/pfman [Olano and Lastra, 1998]
- RTSL [Proudfoot et al., 2001]
- Sh [McCool and Toit, 2004]
- (BrookGPU) [Buck et al., 2004]

Research Languages

- Pixel-Planes 5 [Rhoades et al., 1992]
- PixelFlow/pfman [Olano and Lastra, 1998]
- RTSL [Proudfoot et al., 2001]
- Sh [McCool and Toit, 2004]
- (BrookGPU) [Buck et al., 2004]

Research Languages

- Pixel-Planes 5 [Rhoades et al., 1992]
- PixelFlow/pfman [Olano and Lastra, 1998]
- RTSL [Proudfoot et al., 2001]
- Sh [McCool and Toit, 2004]
- (BrookGPU) [Buck et al., 2004]

Research Languages

- Pixel-Planes 5 [Rhoades et al., 1992]
- PixelFlow/pfman [Olano and Lastra, 1998]
- RTSL [Proudfoot et al., 2001]
- Sh [McCool and Toit, 2004]
- (BrookGPU) [Buck et al., 2004]

Research Languages

- Pixel-Planes 5 [Rhoades et al., 1992]
- PixelFlow/pfman [Olano and Lastra, 1998]
- RTSL [Proudfoot et al., 2001]
- Sh [McCool and Toit, 2004]
- (BrookGPU) [Buck et al., 2004]

Commercial Languages

- GL or DX low-level
- OpenGL Shading Language
- DirectX HLSL
- NVIDIA Cg

Commercial Languages

- GL or DX low-level
- OpenGL Shading Language
- DirectX HLSL
- NVIDIA Cg

Commercial Languages

- GL or DX low-level
- OpenGL Shading Language
- DirectX HLSL
- NVIDIA Cg

Commercial Languages

- GL or DX low-level
- OpenGL Shading Language
- DirectX HLSL
- NVIDIA Cg

Part II

Graphics Hardware

Outline

Machine Complexity

- Graphics machines are complex
- User does not want to know
  - How machine does what it does
  - Tons of machine-specific differences
- Answer:
  - Simple model of machine

Machine Complexity

- Graphics machines are complex
- User does not want to know
  - How machine does what it does
  - Tons of machine-specific differences
- Answer:
  - Simple model of machine
  - High-level language for procedures
  - Well-defined procedure input & output
  - System connects procedures

Machine Complexity

- Graphics machines are complex
- User does not want to know
  - How machine does what it does
  - Tons of machine-specific differences
- Answer:
  - Simple model of machine
  - High-level language for procedures
  - Well-defined procedure input & output
  - System connects procedures

Machine Complexity

- Graphics machines are complex
- User does not want to know
  - How machine does what it does
  - Tons of machine-specific differences
- Answer:
  - Simple model of machine
  - High-level language for procedures
  - Well-defined procedure input & output
  - System connects procedures

Machine Complexity

- Graphics machines are complex
- User does not want to know
  - How machine does what it does
  - Tons of machine-specific differences
- Answer:
  - Simple model of machine
  - High-level language for procedures
  - Well-defined procedure input & output
  - System connects procedures

Machine Complexity

- Graphics machines are complex
- User does not want to know
  - How machine does what it does
  - Tons of machine-specific differences
- Answer:
  - Simple model of machine
  - High-level language for procedures
  - Well-defined procedure input & output
  - System connects procedures

Machine Complexity

- Graphics machines are complex
- User does not want to know
    - How machine does what it does
    - Tons of machine-specific differences
- Answer:
    - Simple model of machine
    - High-level language for procedures
    - Well-defined procedure input & output
    - System connects procedures

Simplified Machine

- User's mental model
- Hide details
- Device independent
- Procedural stages

Simplified Machine

- User's mental model
- Hide details
- Device independent
- Procedural stages

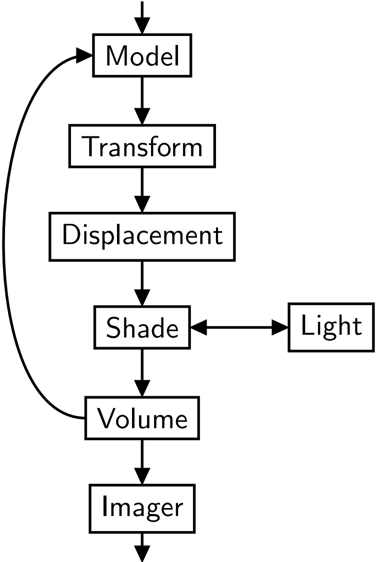
Simplified Machine

- User's mental model
- Hide details
- Device independent
- Procedural stages

Simplified Machine

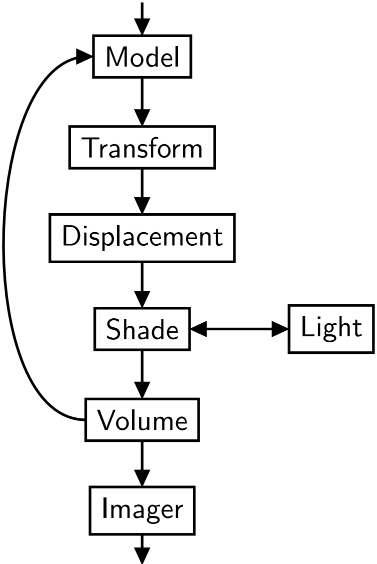
- User's mental model
- Hide details
- Device independent
- Procedural stages

RenderMan Model



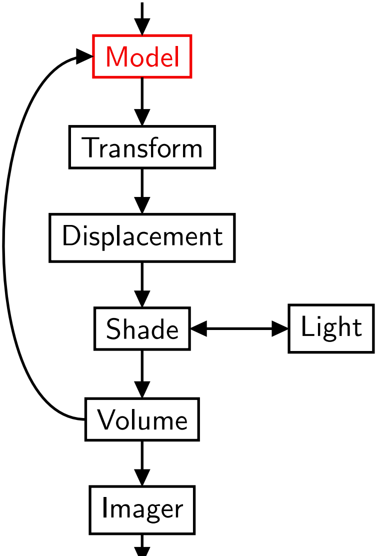
- "Abstract" interface
    - Blocks = procedures
    - Block interfaces well defined
- Connections
    - Inputs & outputs don't have to match
    - System handles conversion

RenderMan Model

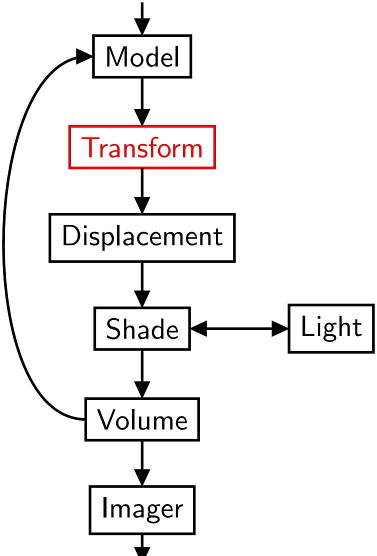


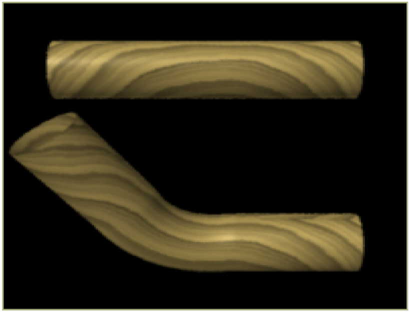
- "Abstract" interface
  - Blocks = procedures
  - Block interfaces well defined
- Connections
  - Inputs & outputs don't have to match
  - System handles conversion

RenderMan
Shader types

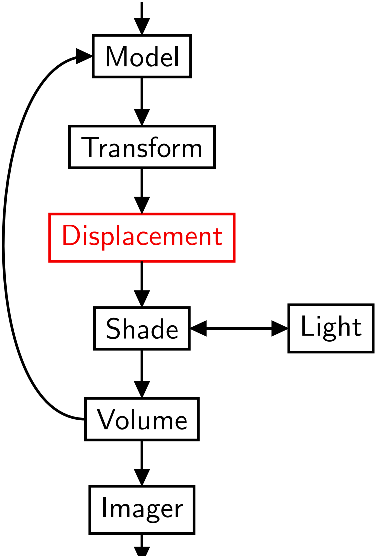


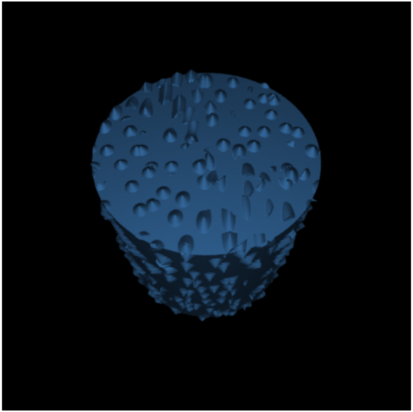
RenderMan
Shader types



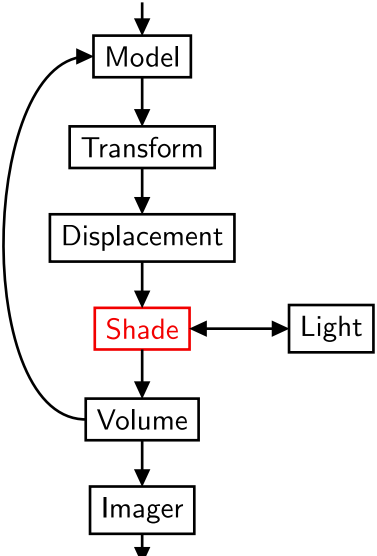


RenderMan
Shader types

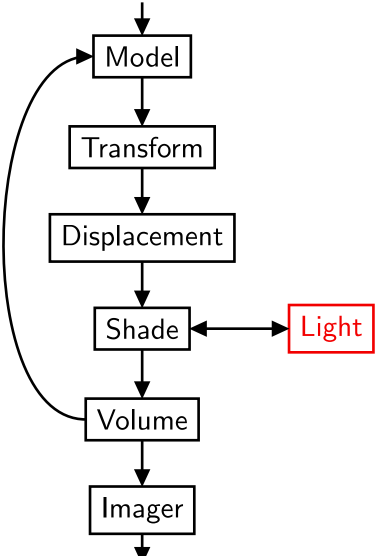


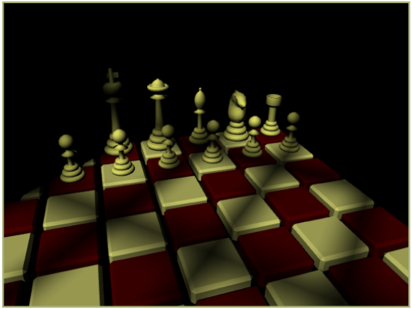


RenderMan
Shader types

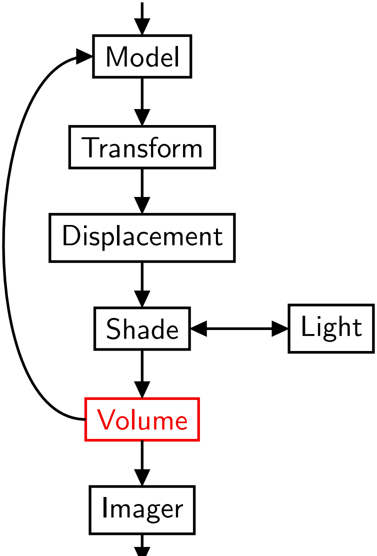


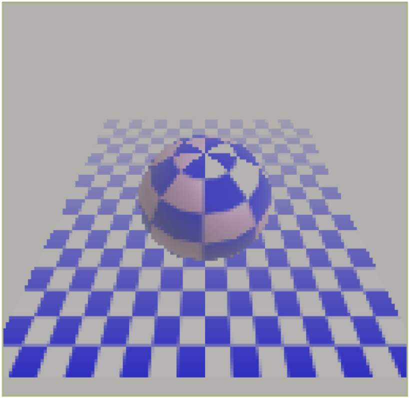
RenderMan
Shader types



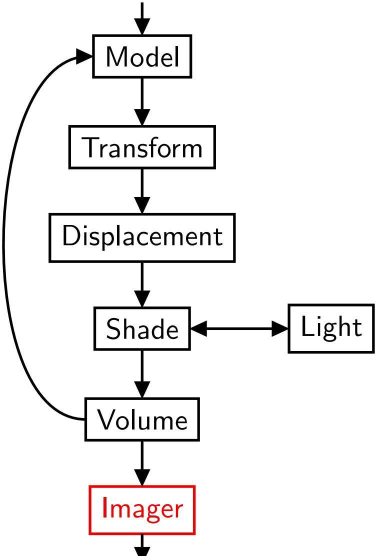


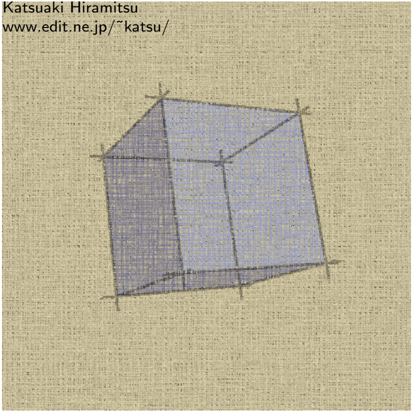
RenderMan
Shader types



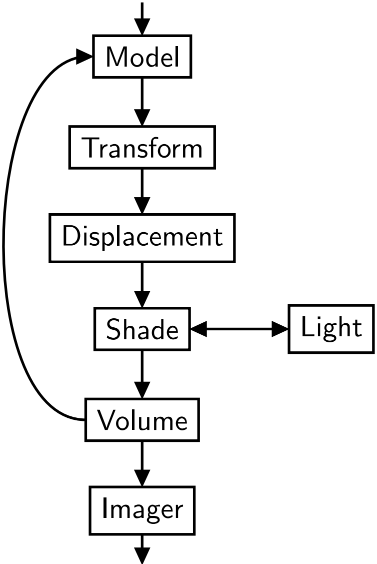


RenderMan
Shader types



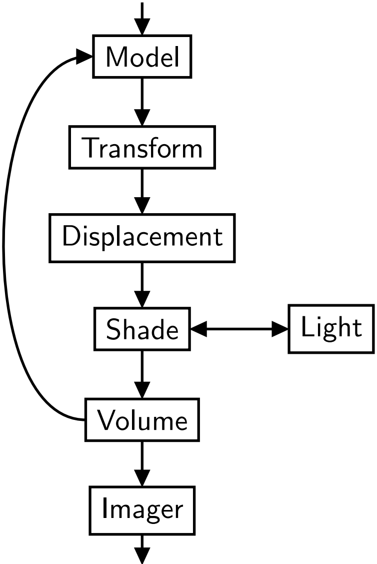


RenderMan Model



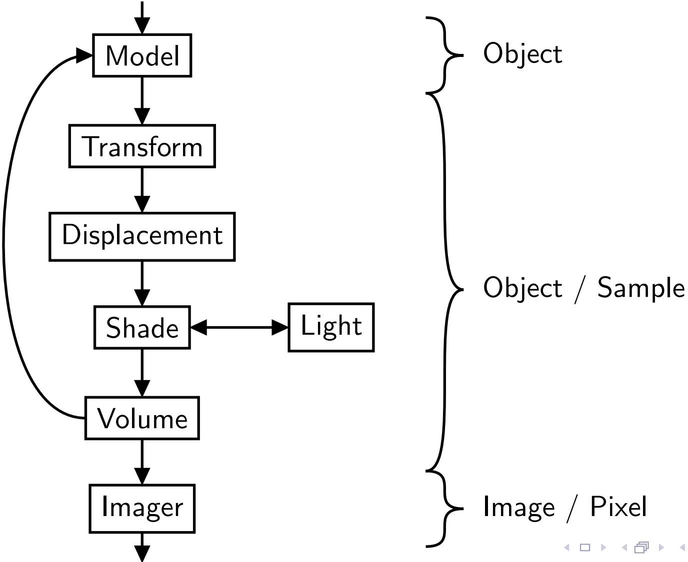
- What it says:
  - Input and output of each block
  - What each block should do
- What it doesn't say:
  - Order or grouping of processing

RenderMan Model

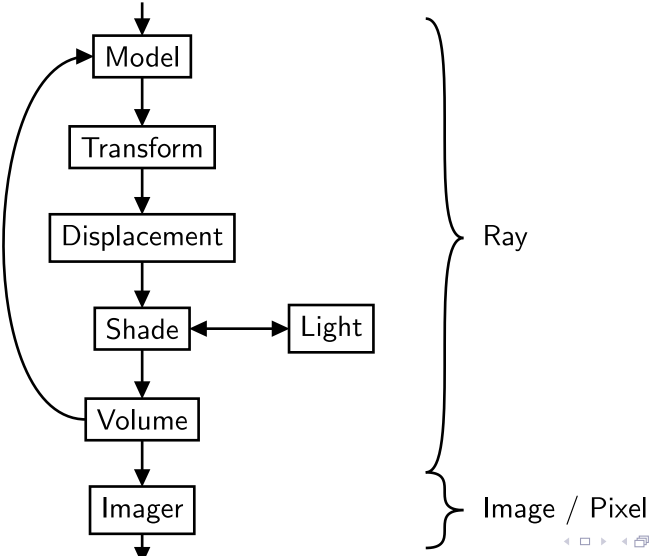


- What it says:
  - Input and output of each block
  - What each block should do
- What it doesn't say:
  - Order or grouping of processing

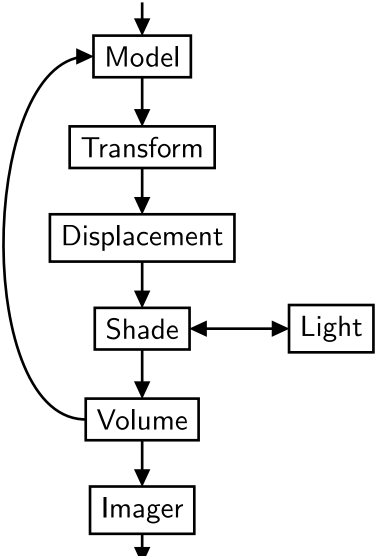
RenderMan
REYES



RenderMan
Ray Tracing



RenderMan
SGI Multi-pass RenderMan

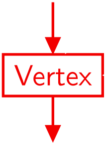


Object

Object / Pass / Pixel

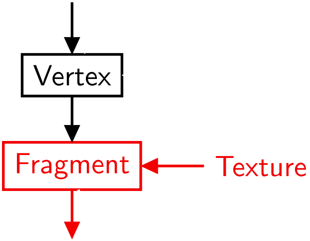
Image / Pixel

Hardware Model



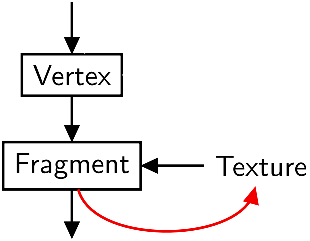
- Vertex shading
  - Transform
    - Procedural transformation
    - Skinning
  - Shade
    - Per-vertex shading
    - Computed texture coordinates

Hardware Model



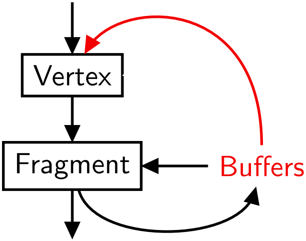
- Fragment shading
  - Per-fragment shading
  - Computed and dependent texture

Hardware Model



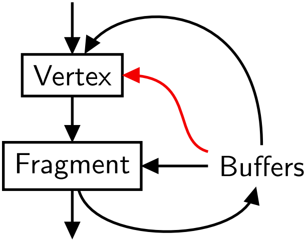
- Render to texture
  - Rendered shadow & environment maps
  - Multi-pass fragment shading [Proudfoot et al., 2001]

Hardware Model



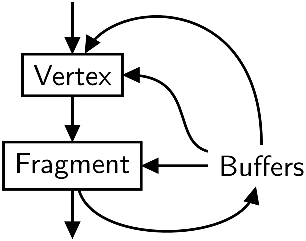
- Render to vertex array / buffer objects
  - Geometry images [Gu et al., 2002]
  - Multi-pass vertex shading
  - Merge vertex & fragment capabilities

Hardware Model



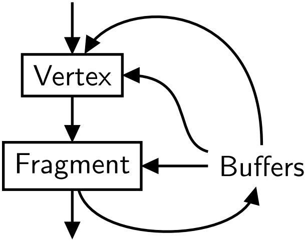
- Vertex texture
  - Texture-based vertex displacement
  - Tabulated functions

Hardware Model



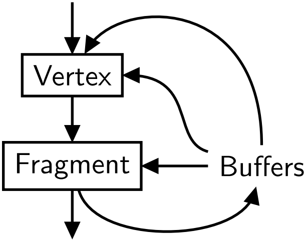
- It's all about the memory
- What it says:
  - Input and output of each block
  - What each block should do
- What it doesn't say:
  - Vertex processing order
  - Fragment processing order
  - Interleaving of vertex and fragment

Hardware Model



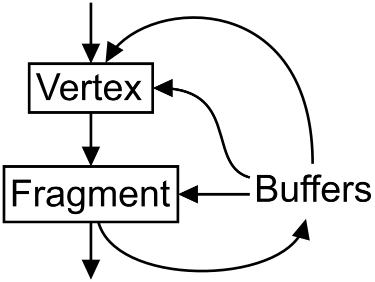
- It's all about the memory
- What it says:
  - Input and output of each block
  - What each block should do
- What it doesn't say:
  - Vertex processing order
  - Fragment processing order
  - Interleaving of vertex and fragment

Hardware Model



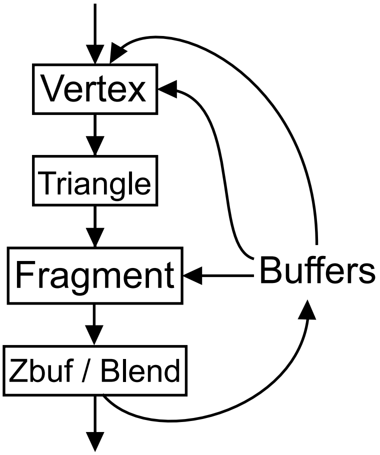
- It's all about the memory
- What it says:
  - Input and output of each block
  - What each block should do
- What it doesn't say:
  - Vertex processing order
  - Fragment processing order
  - Interleaving of vertex and fragment

No, but really, what's in there?



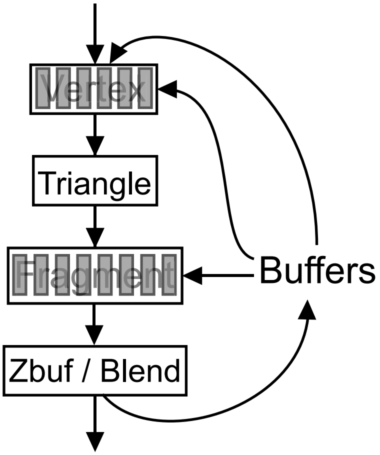
- Some other stuff,
- Parallelism,
- And more parallelism

No, but really, what's in there?



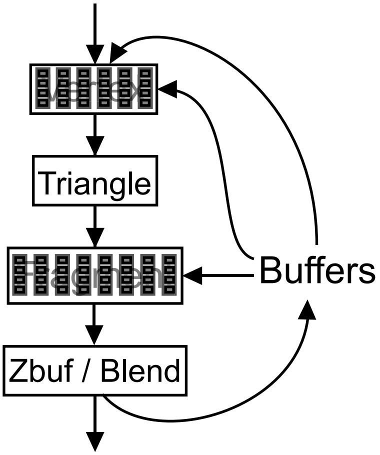
- Some other stuff,
- Parallelism,
- And more parallelism

No, but really, what's in there?



- Some other stuff,
- Parallelism,
- And more parallelism

No, but really, what's in there?



- Some other stuff,
- Parallelism,
- And more parallelism

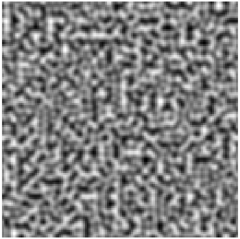
Part III

Noise

Outline

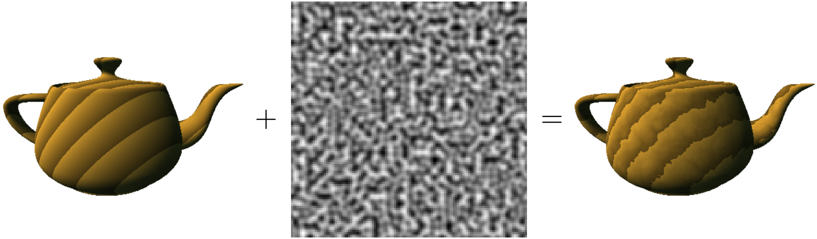
Why Noise?

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



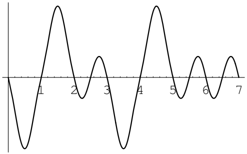
Why Noise?

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



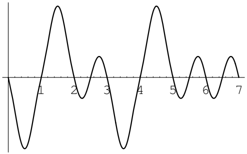
Noise Characteristics

- 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)



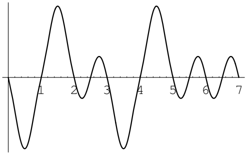
Noise Characteristics

- 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)



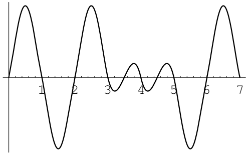
Noise Characteristics

- 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)

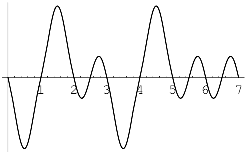


Gradient Noise

- 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



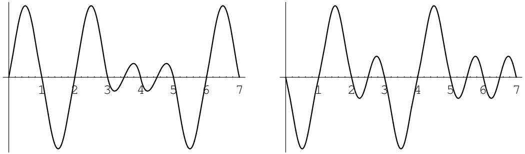
Original



Improved

Gradient Noise

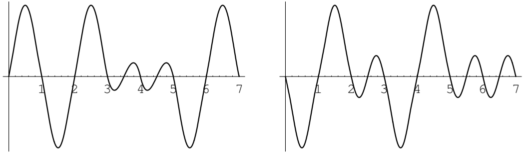
- Original Perlin noise [Perlin, 1985]
- Perlin Improved noise [Perlin, 2002]
- *Lattice* based
  - Value=0 at integer lattice points
  - Gradient defined at integer lattice
  - Interpolate between
- 1/2 to 1 cycle each unit



Original Improved

Gradient Noise

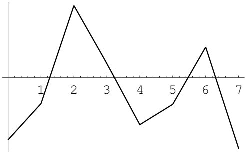
- Original Perlin noise [Perlin, 1985]
- Perlin Improved noise [Perlin, 2002]
- *Lattice* based
    - Value=0 at integer lattice points
    - Gradient defined at integer lattice
    - Interpolate between
- 1/2 to 1 cycle each unit



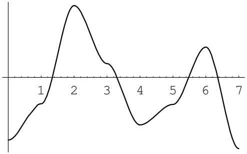
Original Improved

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



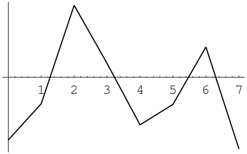
Linear Interp



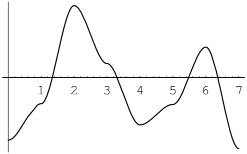
Cubic Interp

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



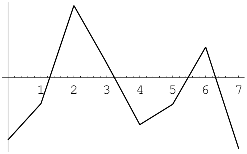
Linear Interp



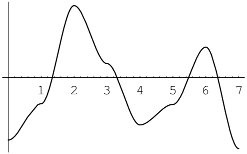
Cubic Interp

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



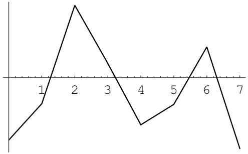
Linear Interp



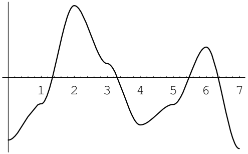
Cubic Interp

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]

Outline

What is this Noise?

Perlin noise

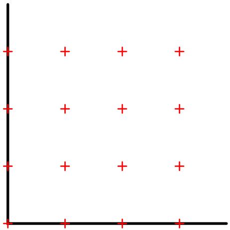
Modifications

Noise Details

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

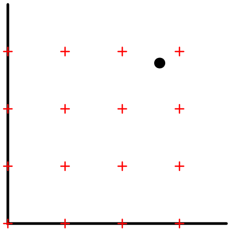
Find the Lattice

- 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
  $\vec{p}_f = \vec{p} - \vec{p}_i = (x, y, z)$



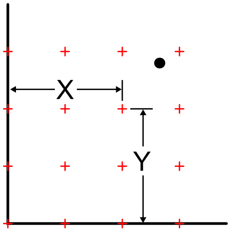
Find the Lattice

- 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
  $\vec{p}_f = \vec{p} - \vec{p}_i = (x, y, z)$



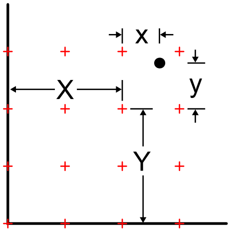
Find the Lattice

- 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

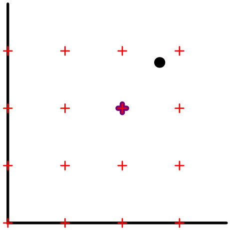
- Interpolate nearest 2^n gradient functions
- 2D $noise(\vec{p})$ is influenced by $\vec{p}_i + (0,0)$; $\vec{p}_i + (0,1)$; $\vec{p}_i + (1,0)$; $\vec{p}_i + (1,1)$
- Linear interpolation
  - $lerp(t, a, b) = (1 - t)\ a + t\ b$
- Smooth interpolation

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

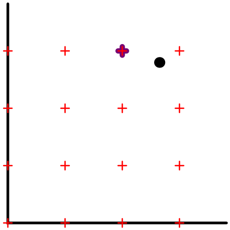
Interpolate

- 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



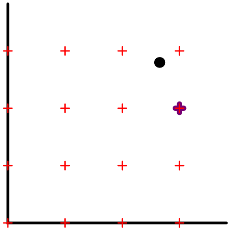
Interpolate

- 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



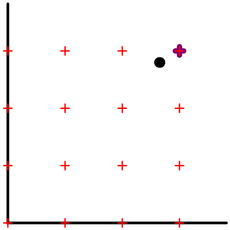
Interpolate

- 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



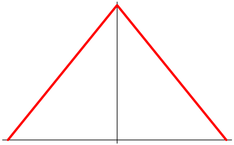
Interpolate

- 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
  - $fade(t) = \{$



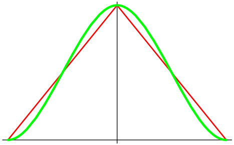
Interpolate

- 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
  - $flerp(t) = lerp(fade(t), a, b)$



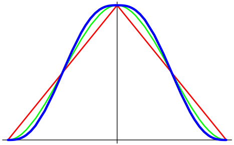
Interpolate

- 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
  - $fade(t) = \begin{cases} 3t^2 - 2t^3 & \text{for original noise} \\ 10t^3 - 15t^4 + 6t^5 & \text{for improved noise} \end{cases}$
  - $flerp(t) = lerp(fade(t), a, b)$



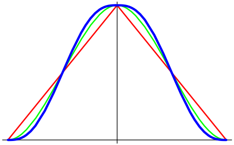
Interpolate

- 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
  - $fade(t) = \begin{cases} 3t^2 - 2t^3 & \text{for original noise} \\ 10t^3 - 15t^4 + 6t^5 & \text{for improved noise} \end{cases}$
  - $flerp(t) = lerp(fade(t), a, b)$



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
  - $fade(t) = \begin{cases} 3t^2 - 2t^3 & \text{for original noise} \\ 10t^3 - 15t^4 + 6t^5 & \text{for improved noise} \end{cases}$
  - $flerp(t) = lerp(fade(t), a, b)$



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

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

- 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

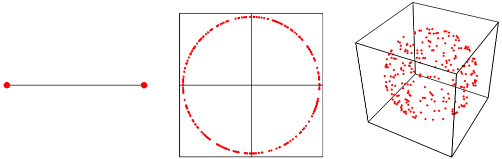
$$\textcolor{red}{g}(hash(Z + hash(Y + hash(X))))$$

- Both original and improved use a permutation table *hash*
- Original: $\textcolor{red}{g}$ is a table of unit vectors
- Improved: $\textcolor{red}{g}$ is derived from bits of final hash

Outline

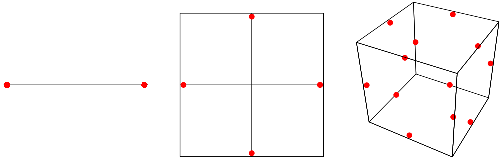
Gradient Vectors of n-D Noise

- 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 $\vec{p}_i$

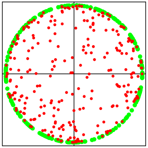


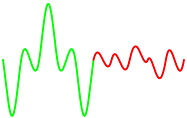
Gradients of noise(x,y,0) or noise(x,0)

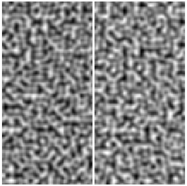
- Why?
  - Cheaper low-D noise matches slice of higher-D
  - Reuse textures (for full noise or partial computation)
- Original: new short gradient vectors
- Improved: gradients in new directions
  - Possibly including 0 gradient vector!

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!

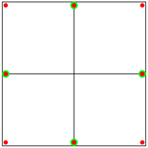


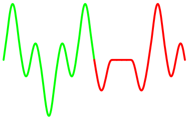


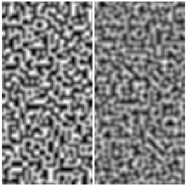


Gradients of noise(x,y,0) or noise(x,0)

- 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?

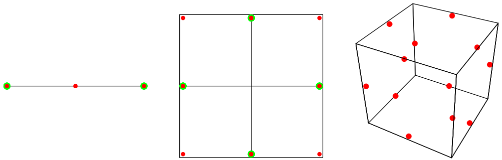
- Observe: use gradient function, not vector alone

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

- 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?

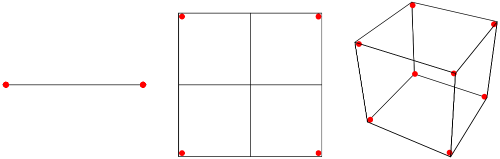
- Observe: use gradient function, not vector alone

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

- 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!

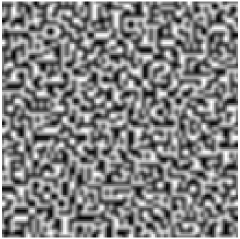


Corner Gradients

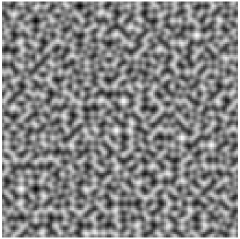
- Simple binary selection from hash bits
  $\pm x, \pm y, \pm z$
- Perlin mentions "clumping" for corner gradient selection
  - Not very noticeable in practice
  - Already happens in any integer plane of improved noise

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



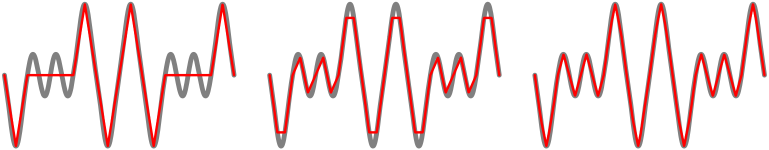
Corner

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

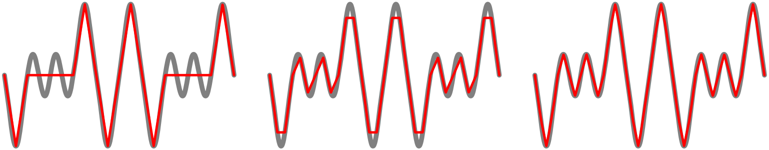
- Like to store computation in texture
    - Texture sampling 3-4x highest frequency



    - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures

Separable Computation

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

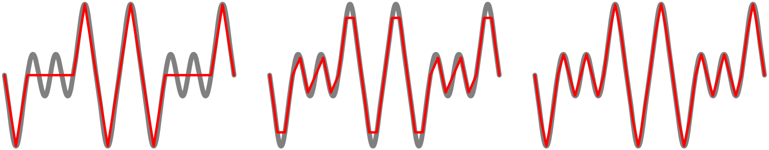


  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures
  - (e.g. write $noise(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)

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

Separable Computation

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

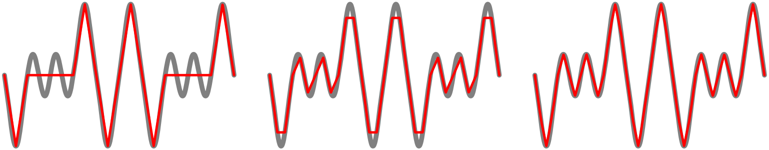


  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures
  - (e.g. write $noise(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)

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

Separable Computation

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

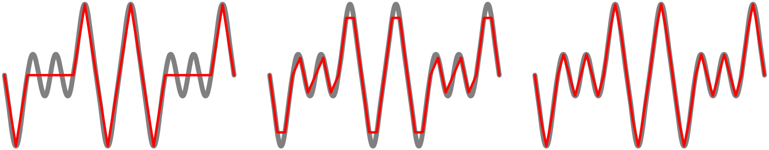


  - 1D & 2D OK size, 3D gets **big**, 4D impossible
- Factor into lower-D textures
  - (e.g. write $noise(\vec{p}^x, \vec{p}^y, \vec{p}^z)$ as several x/y terms)

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

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

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

- 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, $zconst = noise$!

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 3D 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

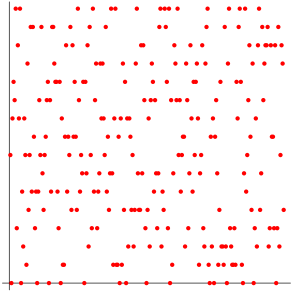
- Many choices; I kept 1D chaining
- Desired features
  - Low correlation of hash output for nearby inputs
  - Computable without lookup
- Use a random number generator?
  - Seed
  - Successive calls give uncorrelated values

Alternative Hash

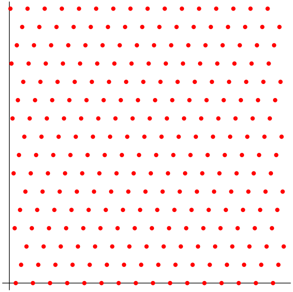
- Many choices; I kept 1D chaining
- Desired features
  - Low correlation of hash output for nearby inputs
  - Computable without lookup
- Use a random number generator?
  - Seed
  - Successive calls give uncorrelated values

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)?



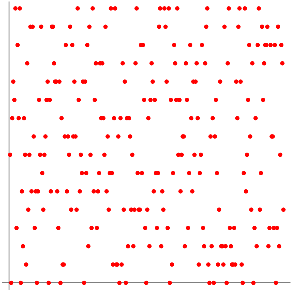
permute table



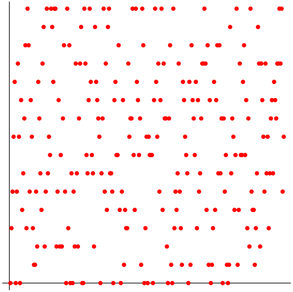
hash using seed=X

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

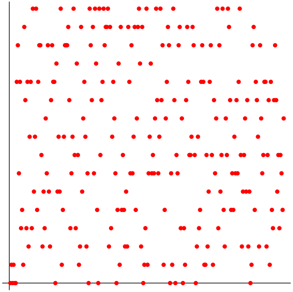


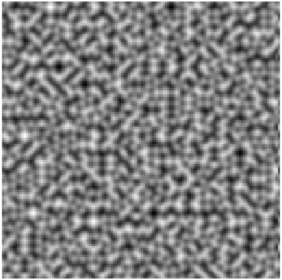
hash using X^{th} random number

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!



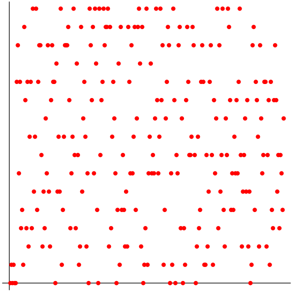


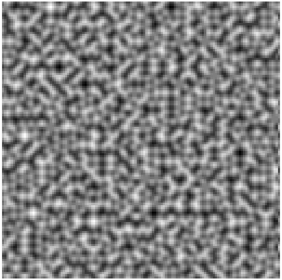
523*527

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!



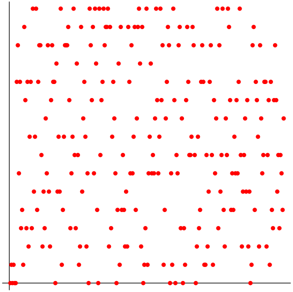


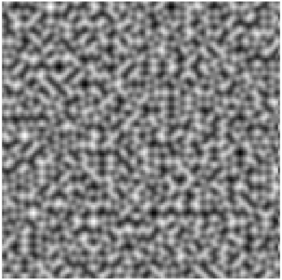
523*527

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!



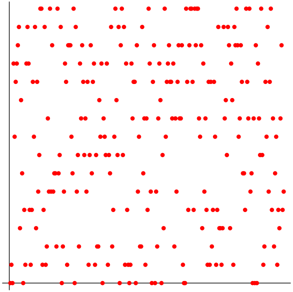
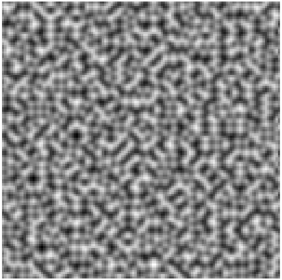


523*527

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!

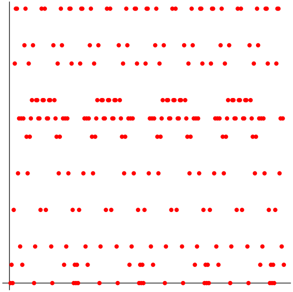



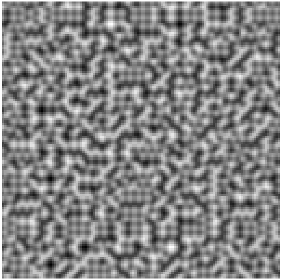
29*31

Blum-Blum Shub

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

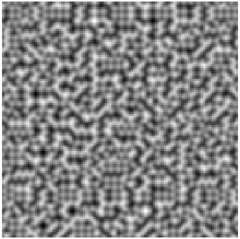
- Uncorrelated for nearby seeds...
- But large M is bad for hardware...
- But reasonable results for smaller M...
- And square and mod is simple to compute!



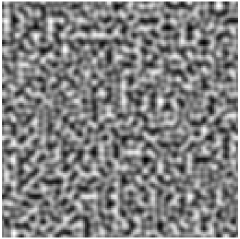


Modified Noise

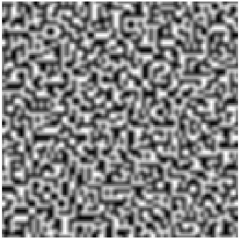
- Square and mod hash
  - $M = 61$
- Corner gradient selection
  - One 2D texture for both 1D and 2D
- Factor
  - Construct 3D and 4D from 2 or 4 2D texture lookups



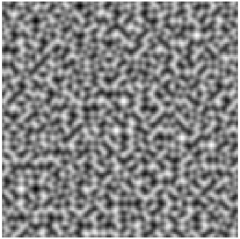
Comparison



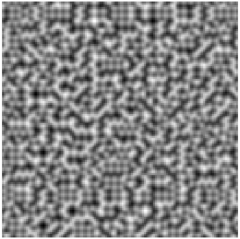
Perlin original



Perlin improved



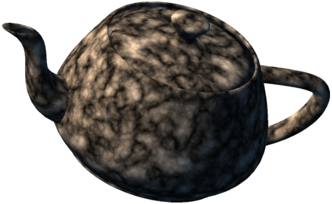
Corner gradients



Corner+Hash

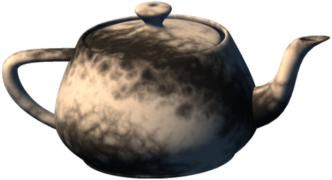
Using Noise

3D noise



3D turbulence

Wood



Marble

Buck, I., Foley, T., Horn, D., Sugerman, J., and pat Hanrahan (2004).
Brook for GPUs: Stream computing on graphics hardware.
ACM Transactions on Graphics, 23(3).

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

Gu, X., Gortler, S. J., and Hoppe, H. (2002).
Geometry images.
ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002), 21(3):355–361.

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.

McCool, M. and Toit, S. D. (2004).
Metaprogramming GPUs with Sh.
AK Peters.

Olano, M. and Lastra, A. (1998).
A shading language on graphics hardware: The pixelflow shading system.
In *Proc. SIGGRAPH*, pages 159–168.

Perlin, K. (1985).
An image synthesizer.
Computer Graphics (Proceedings of SIGGRAPH 85), 19(3):287–296.

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.

Proudfoot, K., Mark, W. R., Hanrahan, P., and Tzvetkov, S. (2001).
A real-time procedural shading system for programmable graphics hardware.
In *Proc. ACM SIGGRAPH*.

Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U., and Varshney, A. (1992).
Real-time procedural textures.
In Zeltzer, D., editor, *1992 Symposium on Interactive 3D Graphics*, pages 95–100. ACM SIGGRAPH.