Hardware-Accelerated Procedural Texture Animation

Greg James

NVIDIA

Agenda

- Introduce Concepts
- Demos of Effects
- Explanations
  - Basic Effect – Fire and Smoke
  - Effect – Dynamic Bump Maps
  - Effect – Interactive Water Simulation on the GPU!
  - Effect – Large Bodies of Water
    - Special Guest!
- More Ideas
- Q&A

Audience

- Intro & Overview:
  - Everyone!
  - Artists, Programmers, Designers

- Detailed Explanations:
  - Everyone!
  - Programmers
  - Folks that know textures and basic 3D graphics
  - A little DX8.1 code
    - Emphasis on concepts
    - Same things possible in OpenGL

Context

- PC and Xbox Games
  - concepts apply to PS2 (Baldur’s Gate: Dark Alliance)
- Hardware
  - GeForce 3
  - Radeon 8500
- DirectX 8.1
  - Pixel Shaders 1.1
  - Vertex Shaders 1.1
- OpenGL – not discussed today
  - demos → http://developer.nvidia.com
  - NV_vertex_program
  - NV_texture_shader
  - NV_register_combiners
  - Similar extensions from other vendors

Introduction

- Procedural textures are more than wood and marble!
- **Animation** is our goal
  - Dynamic bump maps
  - Animated textures
  - Special effects
  - Interactive effects
- “Procedural” enables real-time control

Elder Scrolls III: Morrowind
Bethesda Softworks

Brief History

- Early techniques for textures and geometry
  - Ken Perlin, D. Peachey, G. Gardner – Noise, marble
  - F.K. Musgrave – Fractal landscapes, geometry
- Storage is expensive, slow
- Non-procedural gives: Fixed resolution, 2D, no animation
- Reference:
  - D. Ebert, F.K. Musgrave, “Texturing & Modeling, A Procedural Approach 2nd ed.,” Academic Press, 1998

Effects for This Talk

- Developed over the last year
- GeForce 3 hardware demos
  - 4 texture samples per pass
  - Vertex and Pixel Shaders
- Matthias Wloka (NVIDIA) began this thread
  - Edge detection
  - Image processing
- Greg James (NVIDIA)
  - Dynamic normal maps
  - Animation effects
- ATI – Image processing

Practical Techniques!

Elder Scrolls III: Morrowind

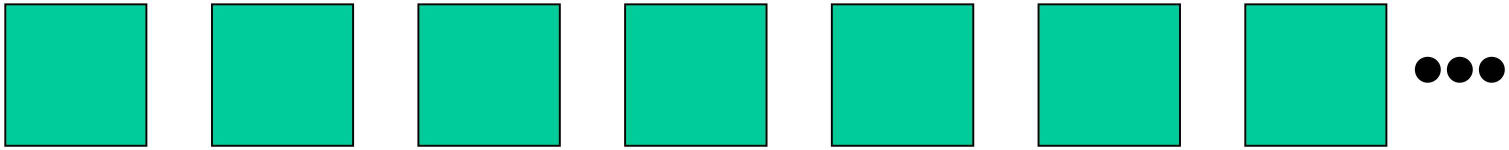
- Designed for real-time games
- Effects run at 150 – 500+ frames per second
  - Shouldn’t kill your frame rate
- Bump Maps, Water, and Fire/Plasma are cool!
- Free modular source code
  - Drop-in classes to run an effect
  - Add working effect in 5 lines of code ☺
- Developer support
  - Just ask!

Define “HW Procedural Texture Animation”!

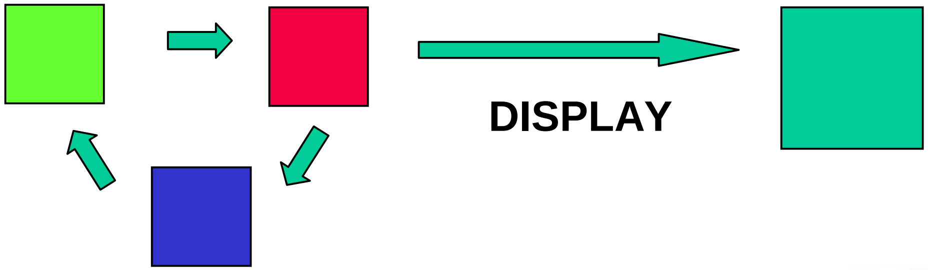
- It Is:
  - Textures created at run-time as needed
  - Rendering operations which create new textures
  - Using textures to create new textures
- It is NOT:
  - An artist painting everything by hand
  - ‘Canned’ animation
  - Consuming disk space for each frame
- It can be:
  - Fast, endless, non-repeating or repeating
  - Interactive

Compare ‘Canned’ to Procedural

- Canned animations store every frame
  - Huge storage requirement, even with compression



- Procedural animations only need to store a few frames or components
- Frames created and displayed as needed



Canned vs. Procedural

- Canned offers complete control
  - Make a movie
  - To change it: Make another movie

- Procedural defines a behavior
  - A system with rules
  - Hopefully it behaves the way you want it to!
  - To change it: Add input or change the rules
    - Change rules as it is running
    - Emergent behavior

Each Has Its Place

- Use Canned for:
  - Short loops
  - Absolute control
    - Photography
    - Movies
- Use Procedural for:
  - Things you can calculate
  - Physical simulation, water, noise, image processing, special effects
  - User interaction
    - Reactive displays and surfaces

The Basic Idea

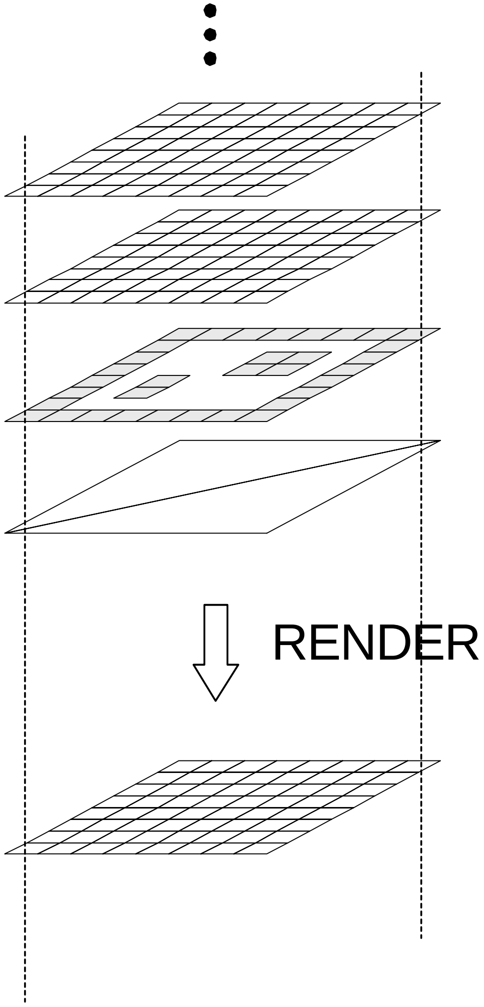
- Render to texture
- Simple geometry drives the processing

- You get several texture samples at each pixel
  - Sample a set of neighbors
- Combine samples into rendered pixels
- Use rendered textures as needed

- Can use advanced Pixel Shader instructions
  - Dependent texture reads
  - Dot-products
  - CND   conditional instruction

Fundamental Operation

- **Render to texture**
- **Then use rendered texture in rendering your scene**



Keep it on the GPU!

- Avoid texture Locks!
- No AGP texture transfer between CPU and GPU
- No CPU or GPU stalls!
  - Caveat – May flush some GPU pipes, but this is better than a complete stall
- Huge GPU computation power (fill rate)
  - 10s or 100s of millions of animated texels / second
  - Saves a lot of CPU MHz
- Free parallel processing
  - Several pixels per clock

Without Graphics HW

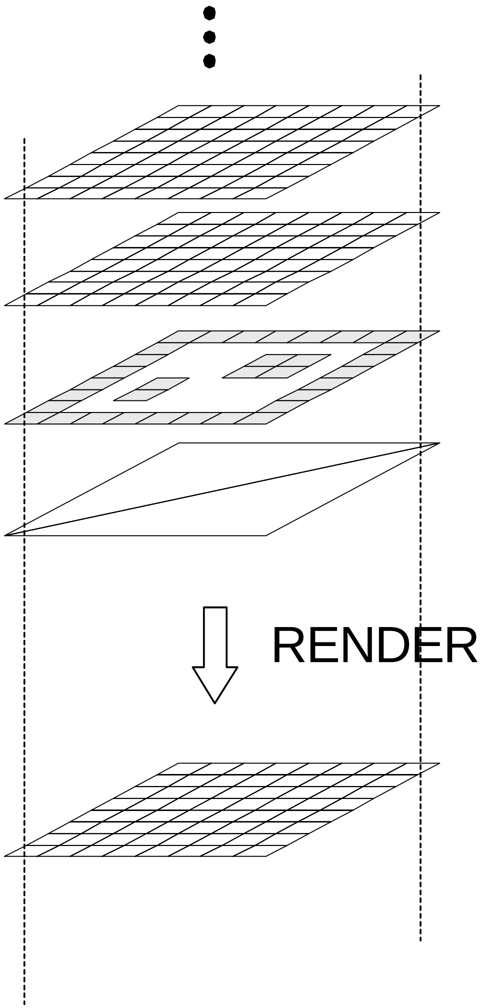
- Heavy CPU load
    - Cycles and memory bandwidth
- Slow transfer to GPU over AGP
- GPU and CPU stall waiting for each other
- Breaks efficient buffering of GPU commands
- Lots of nasty SIMD assembly code
    - Two versions: Intel, AMD
- It’s SLOWER! ☹

Overview of Effects

- Simple fire and smoke
- Dynamic normal maps
- Water
- Cellular automata
  - Noise
  - Patterns

Fundamental Operation

- **Render to texture**



API Calls

- DirectX 8
  - IDirect3DDevice8->SetRenderTarget(color, depth);
  - color = IDirect3DTexture8->GetSurface(..);
  - depth is usually not used
- OpenGL
  - WGL_ARB_render_texture
  - WGL_EXT_pbuffer
  - GL_NV_register_combiners
  - GL_NV_texture_shader

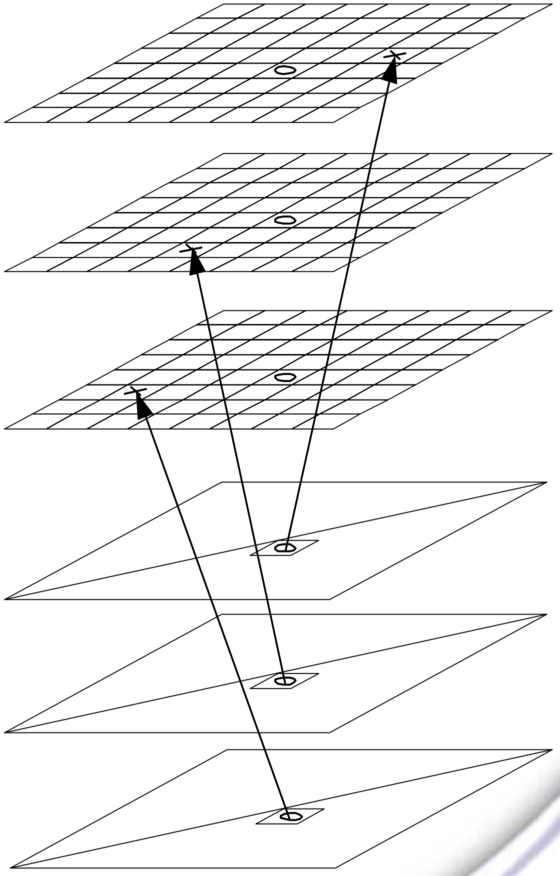
Steps

- Bind input textures
- Establish texture coordinates
- Configure Pixel Shader / Register Combiners
- Set texture render target
- Render simple geometry

- Set ordinary render states
- Set render target to back buffer
- Render scene

Texture Coordinates Determine Sampling

- **Sampling can be**
  - **One-to-One**
  - **Neighbors**
  - **Arbitrary**
- **Coordinates from**
  - **Vertices**
  - **Per-pixel displacements**



Texture 0

Texture 1

Texture 2

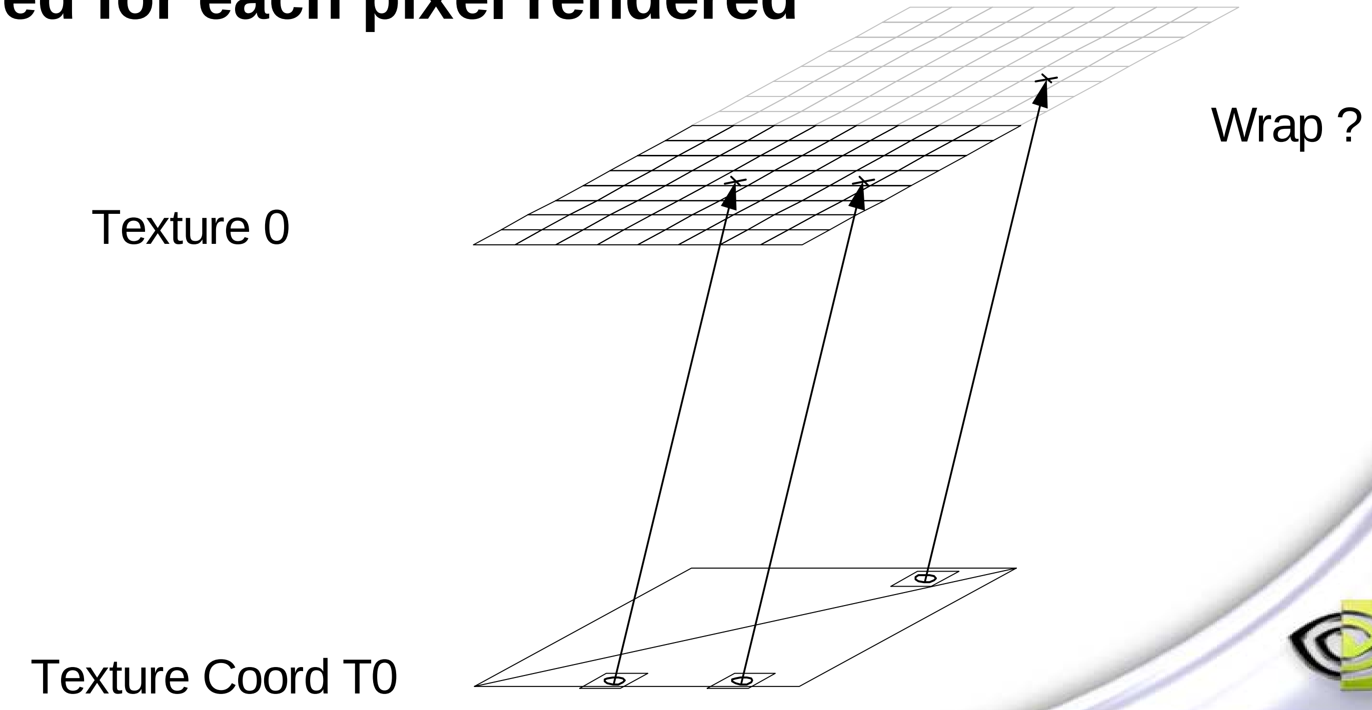
Texture Coord T0

Texture Coord T1

Texture Coord T2

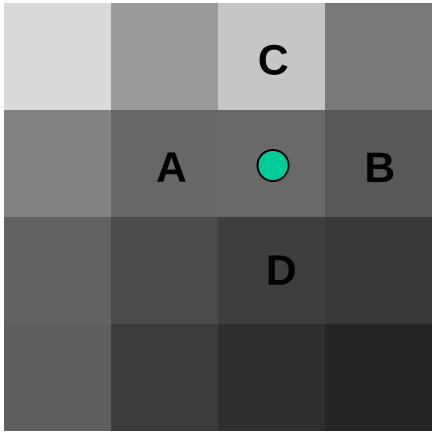
Coordinate Interpolation

- Vertex texture coordinates are interpolated
  - Gives texture coordinates for each pixel rendered
- Interpolation causes same neighbor pattern to be sampled for each pixel rendered

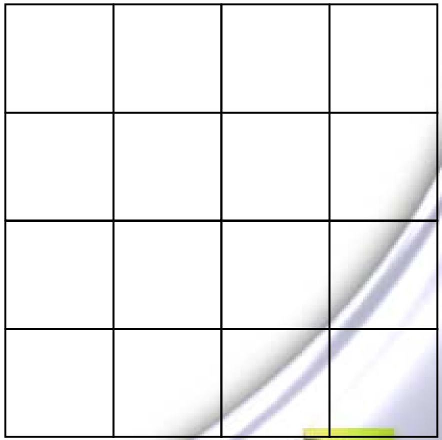


How to Sample Each Texel’s Neighbors

- **Source texture: (x,y) pixels in size**
  - **SetTexture(0..3, pSource);**



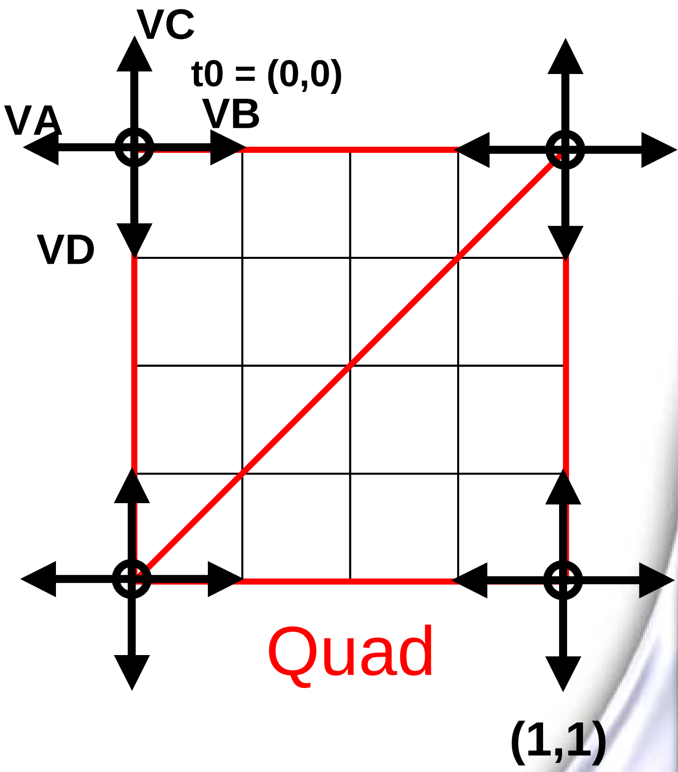
- **Render Target: also (x,y) pixels in size**
  - **SetRenderTarget(pDest, NULL);**
  - **NULL for no depth buffer**



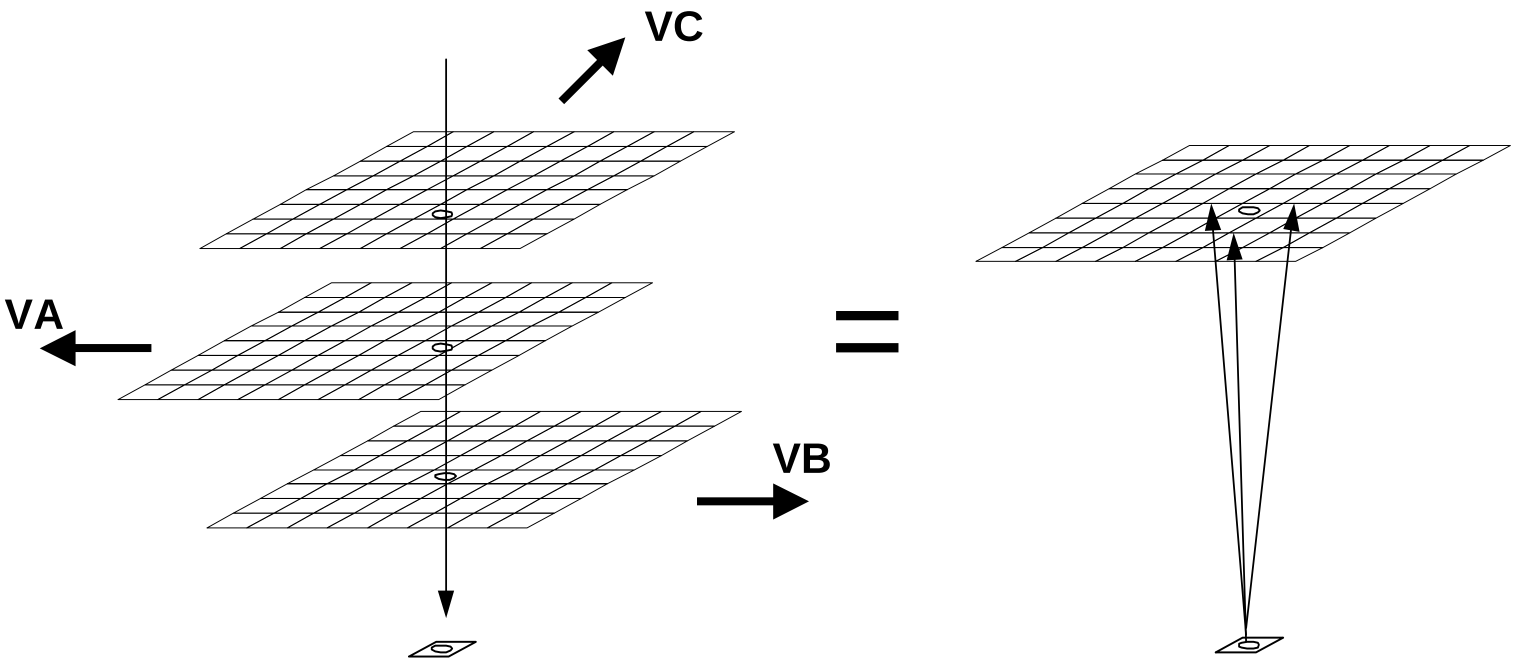
How To Sample Each Texel's Neighbors

- Render a quad over render target
  - Texture coordinates from (0,0) to (1.0, 1.0)
- Vertex Shader writes <u>four</u> different texture coordinates for each vertex
- Each of the four coordinates is offset by a vector VA, VB, VC, or VD

```
oT0 = vertex_tc0 + c[VA]
oT1 = vertex_tc0 + c[VB]
oT2 = vertex_tc0 + c[VC]
oT3 = vertex_tc0 + c[VD]
```

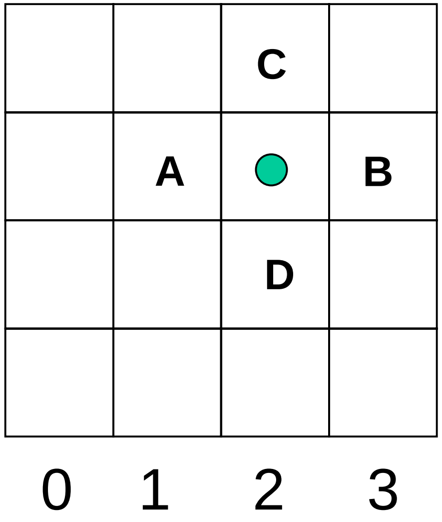


Offset Coordinates Sample Neighbors



- **Or some pattern of other texels**

Sampling From Neighbors



- **t0, t1, t2, t3 samples delivered to Pixel Shader**
- **When destination pixel, ● is rendered, if VA, VB, VC, VD are (0,0) then:**

```
t0 = ● pixel at (2,1)
t1 = ● pixel at (2,1)
t2 = ● pixel at (2,1)
t3 = ● pixel at (2,1)
```

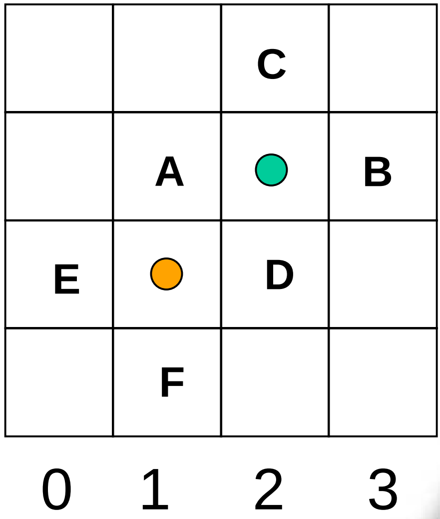
If VA = (-1,0), VB = (1,0), VC = (0,-1), VD = (0,1) then:

```
t0 = pixel A at (1,1)
t1 = pixel B at (3,1)
t2 = pixel C at (2,0)
t3 = pixel D at (2,2)
```

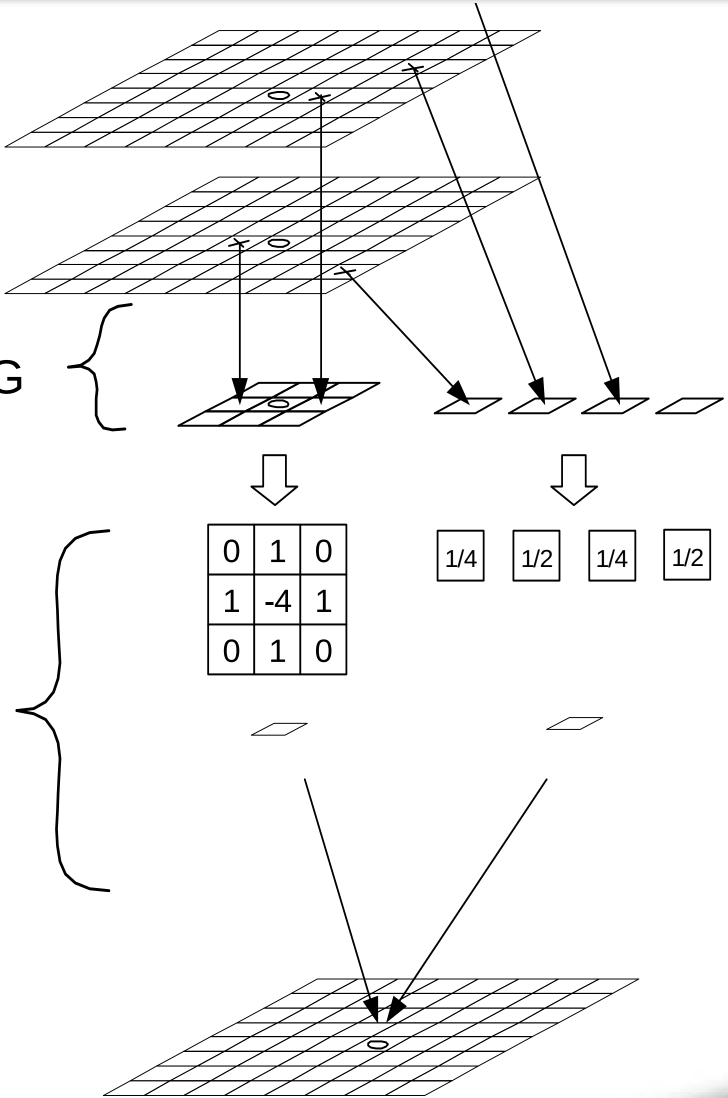
Sampling From Neighbors

- **Same pattern is sampled for each pixel rendered to the destination**
- **When pixel ● is rendered, it samples from:**

```
t0 = pixel E
t1 = pixel D
t2 = pixel A
t3 = pixel F
```



Sample Local Area or Not



ADDRESSING

TEXTURE COORDS

FRAGMENT COMBINE

PIXEL SHADER

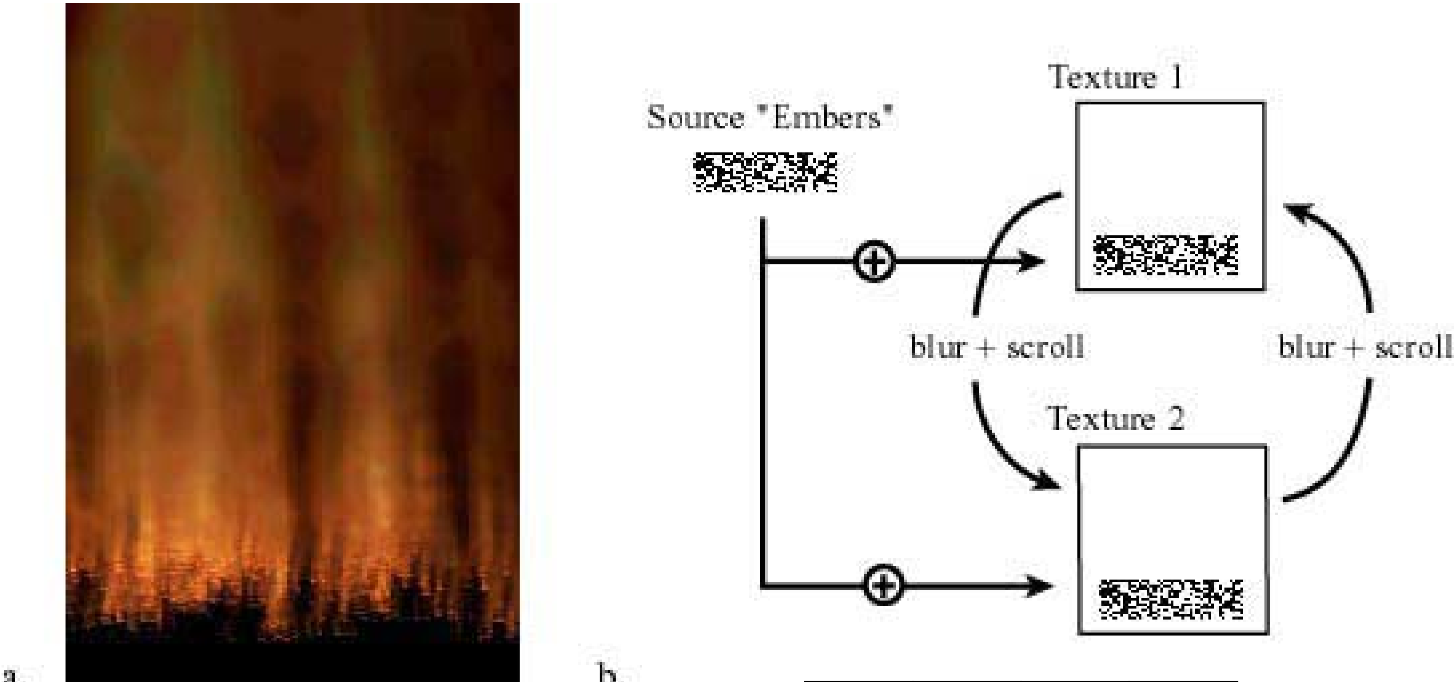
RENDER TARGET

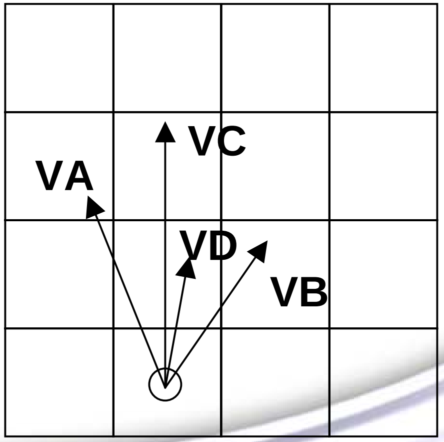
Samples Delivered to Pixel Shader

- Process them however you like
  - Average to blur
  - Difference to sharpen or compute gradients
- Example DirectX 8 Pixel Shader

```
ps.1.1
tex  t0             // t0 = -s,  0     neighbor 1
tex  t1             // t1 = +s,  0     neighbor 2
tex  t2             // t2 =  0, +t
tex  t3             // t3 =  0, -t

sub_x4  r0, t0, t1                      // (t0 - t1)*4  : 4 for higher scale
mul     t0, r0, c[PCN_RED_MASK]         // t0 = s result in red only
sub_x4  r1, t3, t2                      // r1 = t result in green
mad     r0, r1, c[PCN_GREEN_MASK], t0   // r0 = red,green for s and t result
mul_x2  t1, r0, r0                      // t1 = ( 2 * s^2, 2 * t^2, 0.0)
dp3_d2  r1, 1-t1, c1                    // blue = = 1 - s^2 - t^2
add     r0, r0, c2                      // bias red,green to 0.5
mad     r0, r1, c4, r0                  // RGB = (r0.r+0, r0.g+0, 0 + r1.blue )
```

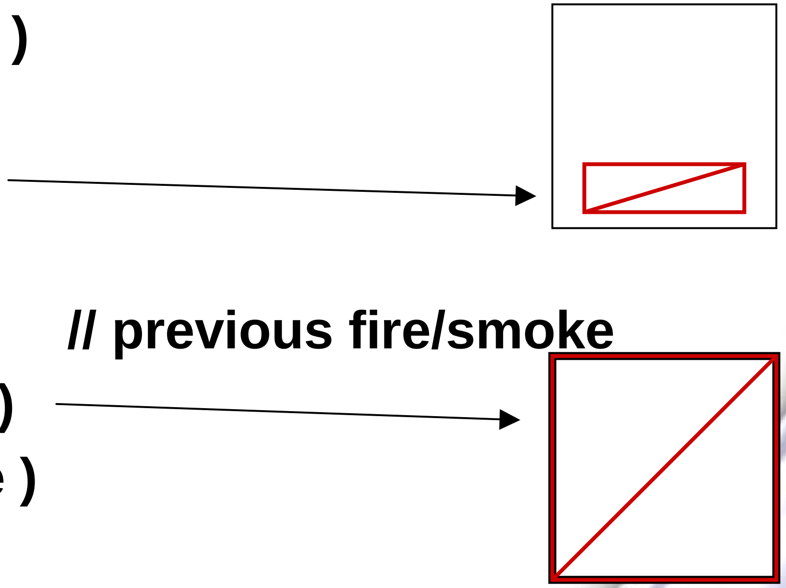
Fire Effect





- Blur and scroll upward
- Trails of blur emerge from bright source ‘embers’ at the bottom

Fire Effect Pseudo-Code



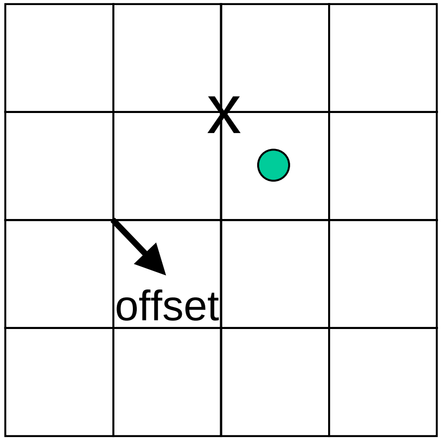
```
Clear( F1_texture );  Clear( F2_texture );
while( not( done ))
    Jitter(  VA, VB, VC, VD,  full_coverage_quad  )
    SetVertexConsts(  VA, VB, VC, VD  )
    SetRenderTarget( F1_texture )
    SetTexture( embers_texture )
    Render( embers_object )

    SetTexture( F2_texture )          // previous fire/smoke
    Render( full_coverage_quad )
    Swap( F1_texture, F2_texture )

    SetRenderTarget( backbuffer, depth )
    RenderScene( using F2_texture )
```

Fire Effect

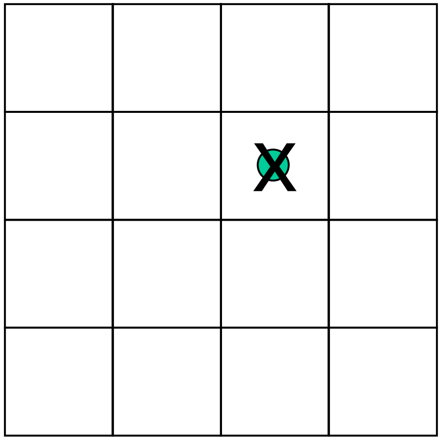
- Jitter texture sampling
  - Vary scroll direction for a wind effect
  - Turbulence: Tessellate geometry with jittered texture coords or positions
- Change color averaging multiplier
  - Brighten or extinguish the smoke
- How to improve:
  - Better jitter patterns (not random jumps)
  - Re-map colors
    - Dependent texture read

Sample Placement

- D3D and OpenGL sample differently
  - D3D samples from texel corner
  - OpenGL samples from texel center
- Can cause problems with bilinear sampling
- Solution: Add half-texel sized offset with D3D



D3D



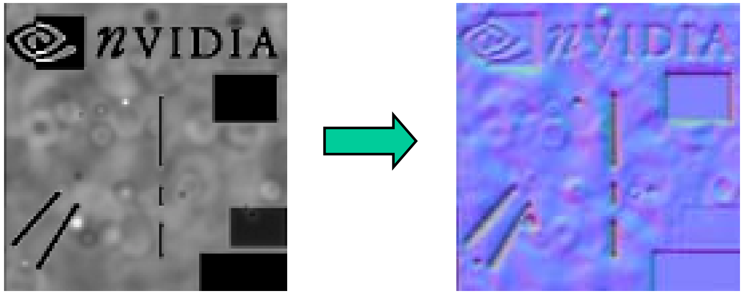
OpenGL

O = pixel rendered
X = sample placement

Dynamic Normal Maps

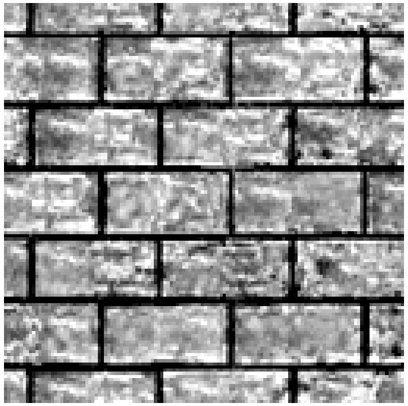
- **Create and update surface normal maps as needed**
- **MOST POWERFULL TECHNIQUE**
- **Normal map from Height map in single pass**

- **Quick review of surface normal maps**
    - **Represent surface geometry**



Review: Surface Normal Maps

- Height maps are popular (3DS Max, Maya, ..)
  - RGBA color represents height of a surface
  - Usually limited to 8 bits of precision
- Normal maps are better
  - RGB color represents XYZ coordinates of surface normal
  - 8 or 16 bits per coordinate axis (more precise!)



Height Map

Normal Map

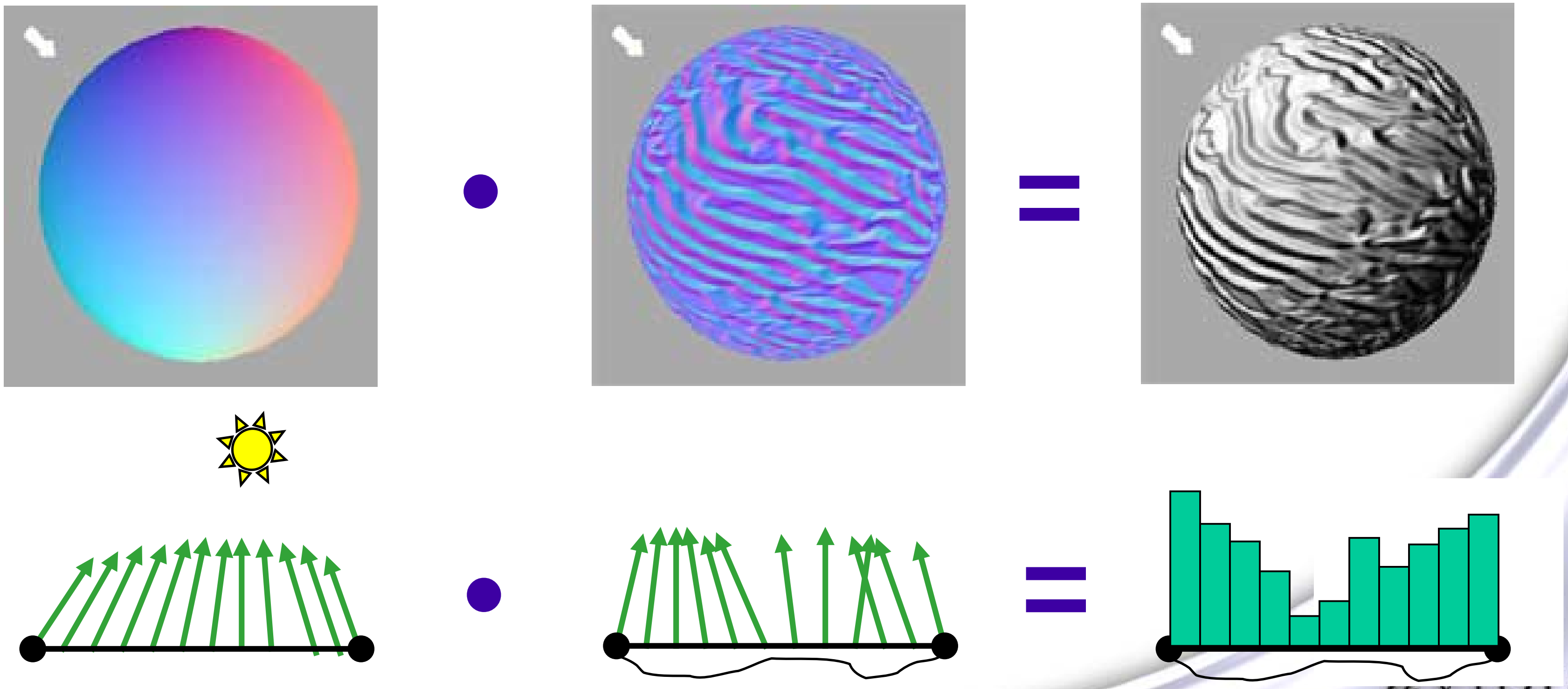
Review: Per-Pixel Lighting

- Lighting equation per-pixel instead of per-vertex
- Visualize light vector and normal map as RGB color

Light Vector, L

Normal map

Per-Pixel Lighting

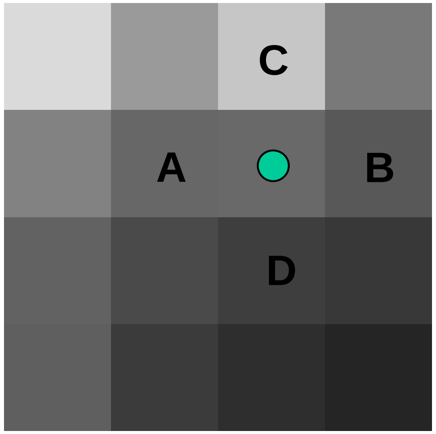


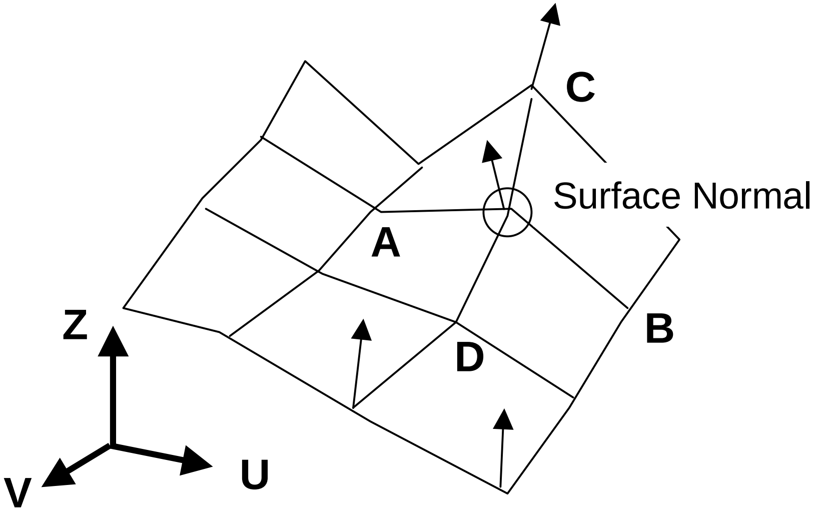
Per-Pixel Reflection Using Surface Normal Map

Cass Everitt

Creating Normal Maps From Height Maps

- Simple: Use 4 nearest neighbors





- dz/du = (B.z - A.z) / 2.0f // U gradient
- dz/dv = (D.z - C.z) / 2.0f // V gradient
- Normal = Normalize((dz/du) × (dz/dv))

× denotes cross-product

Creating Normal Maps in HW

- Can render a normal map from a height map source in a single rendering pass
  - Approximate normalization
  - if A is small then sqrt(1 - A) ~= 1 – ½ A
- Could do exact normalization in 2 passes
  - This isn’t needed. Approximation is good enough!

- Update height maps
  - Render features into height map
- Create normal maps
- Keeps all data on graphics HW

Normal Map Creation Shader

```
ps.1.1
def c2, 0.5, 0.5, 0.0, 0.0
def c1, 1.0, 1.0, 0.0, 0.0
def c4, 0.0, 0.0, 1.0, 1.0

tex  t0        // t0 = -u,  0     neighbor A  t0..t3 are same texture
tex  t1        // t1 = +u,  0     neighbor B
tex  t2        // t2 =  0, +v     neighbor D
tex  t3        // t3 =  0, -v     neighbor C

sub_x4  r0, t0, t1                     //(t0 - t1)*4  for higher scale
mul     t0, r0, c[PCN_RED_MASK]        // t0 = s result in red only
sub_x4  r1, t3, t2                     // r1 = t result in green
mad     r0, r1, c[PCN_GREEN_MASK], t0  // r0= r,g for s and t result
mul_x2  t1, r0, r0                     // t1 =( 2 *s^2, 2 * t^2, 0.0)
dp3_d2  r1, 1-t1, c1                   // blue = = 1 - s^2 - t^2
add     r0, r0, c2                     // bias red,green to 0.5
mad     r0, r1, c4, r0                 // RGB=(r0.r+0,r0.g+0,0+r1.b )
```

Normal Map – Src Height in Blue, Alpha

```
def c1, 1.0, 1.0, 0.0, 0.0
def c2, 0.5, 0.5, 0.0, 0.0
def c4, 0.0, 0.0, 1.0, 1.0

tex t0              // -u,0     t0, t1, t2, t3 are same height texture
tex t1              // +u,0
tex t2              // 0, +v
tex t3              // 0, -v

sub_x4  r0.a, t0, t1                      // (t0 - t1)*4  : 4 for higher scale
mul     t0.rgb, r0.a, c[PCN_RED_MASK]     // t0 = s result in red only
+sub_x4  r1.a, t3, t2                     // r1 = t result in green
mad     r0, r1.a, c[PCN_GREEN_MASK], t0   // r0 = red,green for s and t result
mul_x2  t1, r0, r0                        // t1 = ( 2 * s^2, 2 * t^2, 0.0)
dp3_d2  r1, 1-t1, c1                      // ( 1-2s^2 + 1-2t^2 )/2 = 1 - s^2 - t^2
add     r0, r0, c2                        // bias red,green to 0.5
mad     r0, r1, c4, r0                    // RGB = (r+0, g+0, 0+blue )
```

Animate Normal Map Geometry

- Change surfaces in subtle or drastic ways
- Render damage into surfaces
- Animate cracks, wear
- Character aging
- X-Files inspired skin crawlers
- Fluid surfaces
- Warping, melting surfaces

Height-Based Water Simulation

- Physics on the GPU
  - In glorious 8-bit precision
  - 8 bits is enough, barely!
- Each texel is one point on water surface
- Each texel holds
  - Water height H
  - Velocity V
  - Force F - computed from height of neighbors
- Damped + Driven system
  - For “stability”

It Just So Happens That…

- Discretizing a 2D wave equation to a uniform grid gives equations which sample neighbors

- Derivatives (slopes) in partial differential equations (PDEs) turn into neighbor sampling on a grid

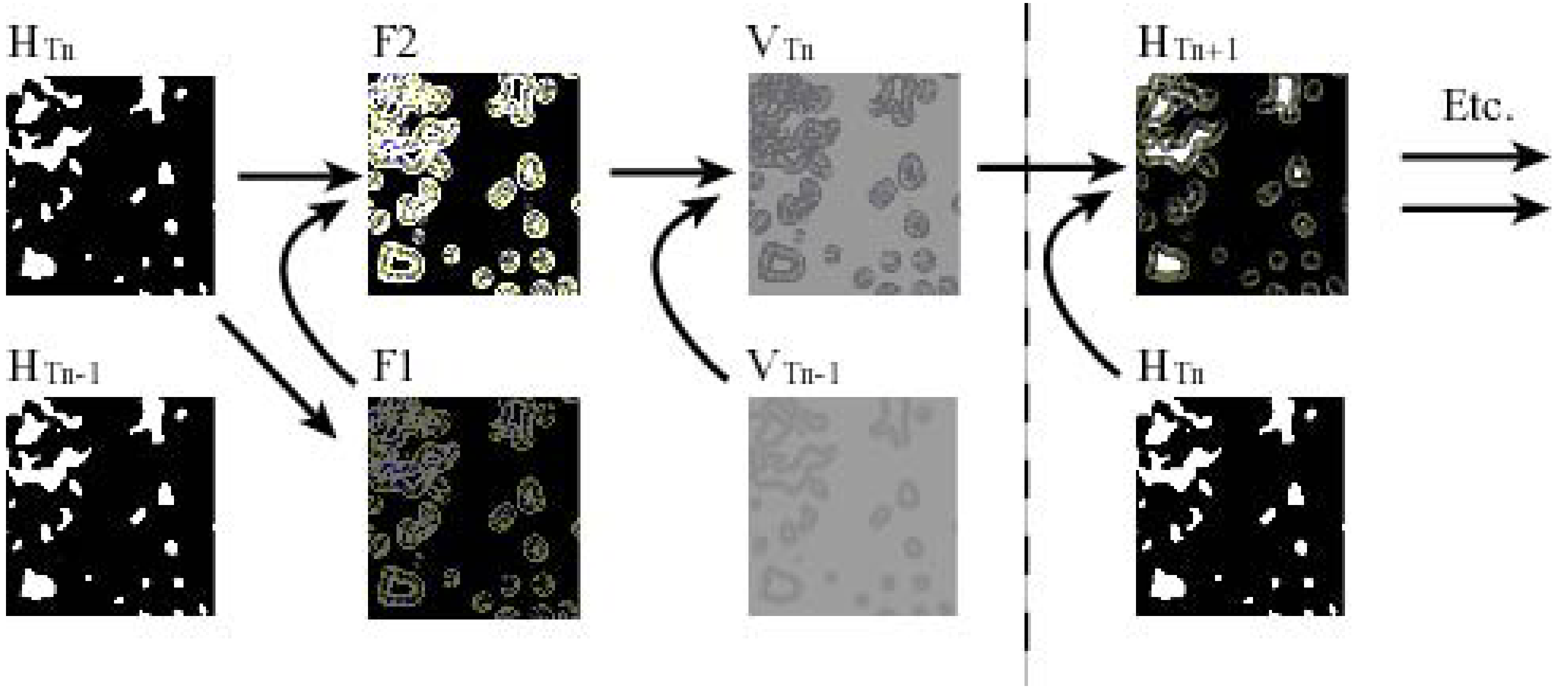
- See [Lengyel] or [Gomez] for great derivations
- Textures + Neighbor Sampling are all we need!
- Forget the math – Use Intuition!
  - And a spring-mass system
  - Math near identical to PDE derivation

The Math

- Height texels are connected to neighbors with springs
- Force acting on H0 from spring connecting H0 to H1
  - $= k * (H1 - H0)$
  - k = spring strength constant
  - Always pulls H0 toward H1
  - H0, H1 are 8-bit color values
- $F = k * (H1 + H2 + H3 + H4 - 4*H0)$
- $V = V + c1 * F$
- $H0 = H0 + c2 * V$
  - c1, c2 are constants (mass,time)

		H1	
	H4	H0	H2
		H3	

Height-Based Water Simulation



- Height current (HTn), previous (HTn-1)
- Force partial (F1), force total (F2)
- Velocity current (VTn), previous (VTn-1)
- Use 1 color channel for each
  - F = red; V = green; H = blue and alpha

Newtonian Physics in Pixel Hardware

F = k * (H1 + H2 + H3 + H4 – 4*H0)

V = V + c1 * F

H0 = H0 + c2 * V

- Repeat, generating new H & V values at each point
- New set of heights is next time step
- Pixel Shader
  - 1) Reads H0..H4, V from texture
  - 2) Calculates new H & V
  - 3) Renders new H & V to texture, to be read back again at step 1
- Will it work? Not quite!

Stability Issues

- High frequency oscillation
  - Checkerboard patterns amplify
  - Solution: Add blur step to smooth H and/or V
- Values hit 0 or 1 saturation
  - Numerical error in 8-bit values
  - Solution: Add gentle force pulling height to 0.5
  - Option: Move heights slightly toward 0.5 at each step
- Blur and Dampening make waves fade to nothing
  - Solution: Add subtle excitations to keep it going
  - Render blobs additively into H or V values

Final Approach

- **Pick c1, c2, k, k2, d1 to match [0,1] color value range**
  - **c1 = 0.4; c2 = 0.48; k = 1; k2 = 0.15; d1 = 0.9875**
  - **Change them to change water behavior!**

F = k * (H1 + H2 + H3 + H4 – 4*H0) + k2 * (0.5 - H0)

V = V + c1 * F

H0 = H0 * d1 + c2 * V

H0 = blur (H1, H2, H3, H4, or other neighbors)

Repeat!

- **Works great!**

How Many Passes?

- Passes at texture resolution – Not screen resolution
- GeForce 3 or 4:
  - Calculate F, V, H: 2 passes
  - Blur H: 1 pass
  - Normal map from H: 1 pass
  - Possible to do it all in 3 passes
  - Mipmapping requires more passes. Not used

- Future HW:
  - Everything in 1 pass
  - Sometimes better to use 2 passes

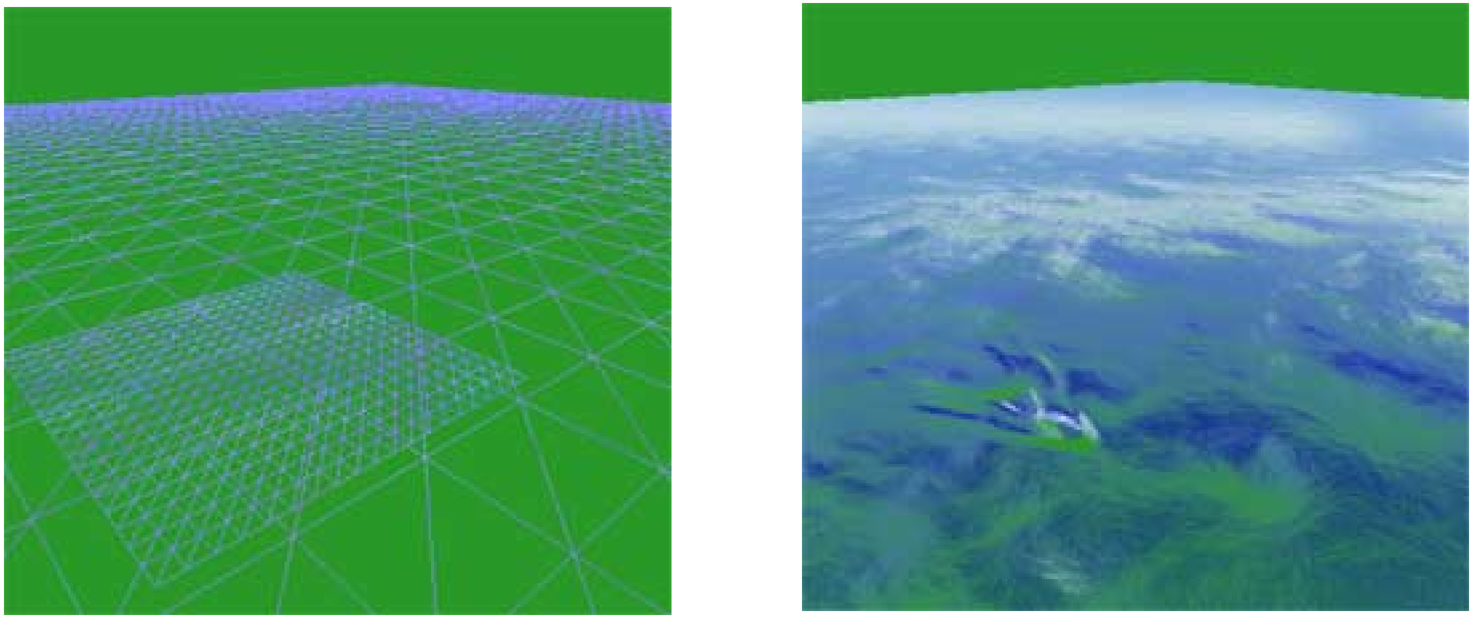
Make It React

- Character moving through
  - Render small blob into H, V, or F (blue, green, or red color channels) at character location
  - Best to render into H Height
  - Additively or alpha blend
  - Physics makes waves spread naturally
- Barriers in water
  - Texture with barrier height in one channel, and barrier ‘strength’ in alpha
  - Alpha blend into H after the physics
  - Alpha = 0 has no effect. Alpha = 1 has full effect of solid barrier

Large Bodies of Water

- Texture border wrap makes water tile seamlessly
- Problem: Character displacements shouldn’t tile
- Answer: Two water simulations
  - One for tiled water
  - One for localized unique water with waves from character
- Couple tiled water into border of localized water
  - Match texture coords as the local water moves
  - Render tiled texture to outer edge of local water
- Tiled and Local will match seamlessly
- See public demos for specifics

Coupled Water



- Used in “Elder Scrolls III: Morrowind”

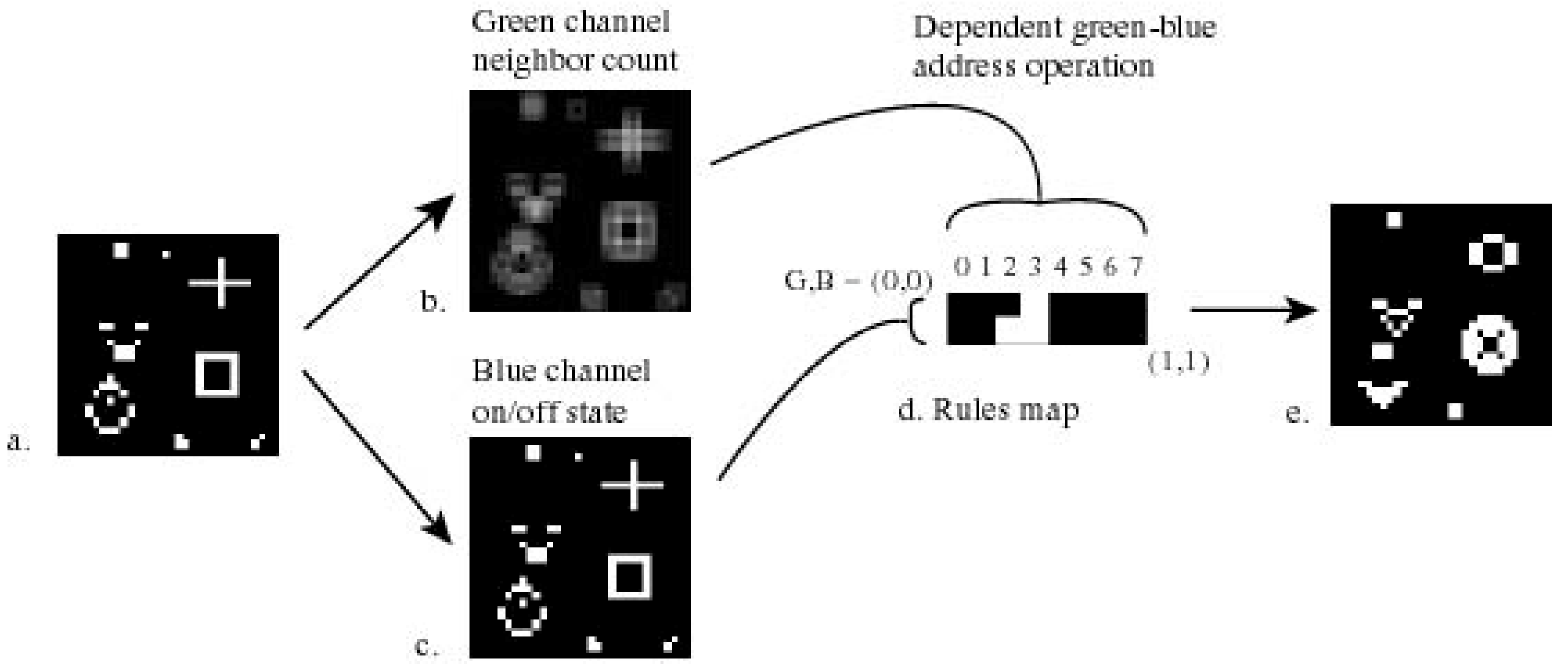
Special Guest

- **Todd Howard, Bethesda Softworks**

More Ideas

- Cellular Automata: patterns, noise, tiles, life!
- Image Processing: edges, bad TVs
  - XBox game “Wreckless”
- Advanced fluids
  - Use texture distortions for flow
  - Simulate temperature, density, pressure, 2D velocity, heat flow
- Future hardware will make it easier, faster, more powerfull

Cellular Automata



- GREAT for generating noise and other animated patterns to use in blending
- Game of Life in a Pixel Shader
  - Three render-to-texture passes per generation
  - Dependent texture read with rules in a texture

Questions?

- **Greg James - gjames@nvidia.com**
  devsupport@nvidia.com

References & Source Code

- Height-based fluid simulation
  - Gomez, Miguel, “Interactive Simulation of Water Surfaces” in “Game Programming Gems,” Charles River Media, 2000, p 187
  - Lengyel, Eric, “Mathematics for 3D Game Programming & Computer Graphics,” Charles River Media, 2002, Chapter 12, p 327
- Game Gems II Article
  - James, Greg, “Operations for Hardware-Accelerated Procedural Texture Animation,” in “Game Programming Gems II,” Charles River Media, 2001, p 497
- Demos -- NVIDIA Effects Browser
  - http://developer.nvidia.com
  - (OLD) http://developer.nvidia.com/view.asp?IO=dynamic_bump_reflection
  - (NEW) http://developer.nvidia.com/view.asp?IO=water_interaction
  - http://developer.nvidia.com/view.asp?IO=cellular_automata_fire
  - http://developer.nvidia.com/view.asp?IO=game_of_life