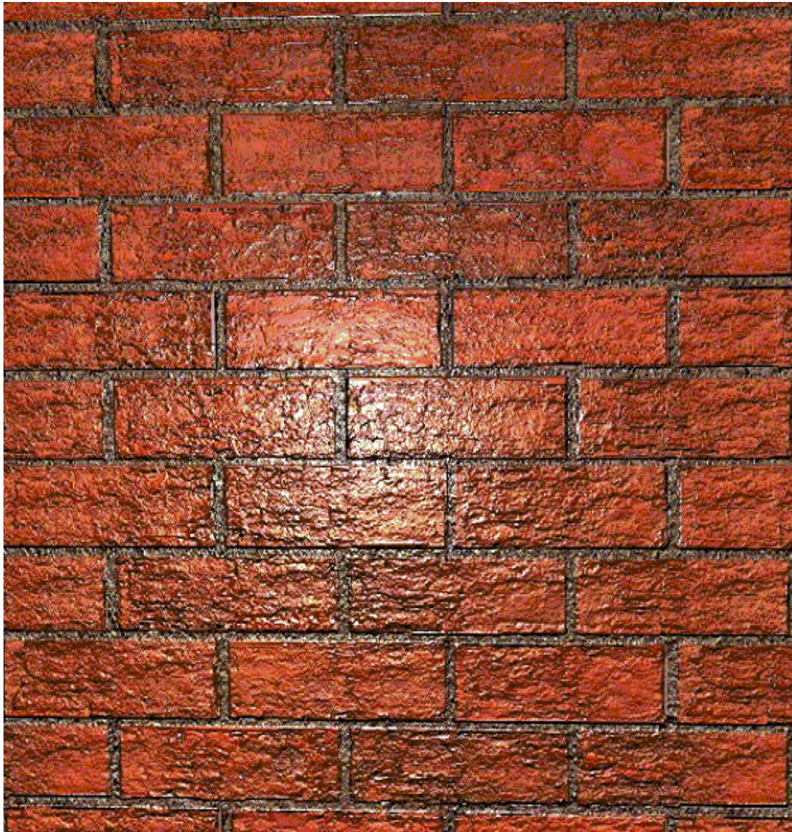
Dx8 Pixel Shaders

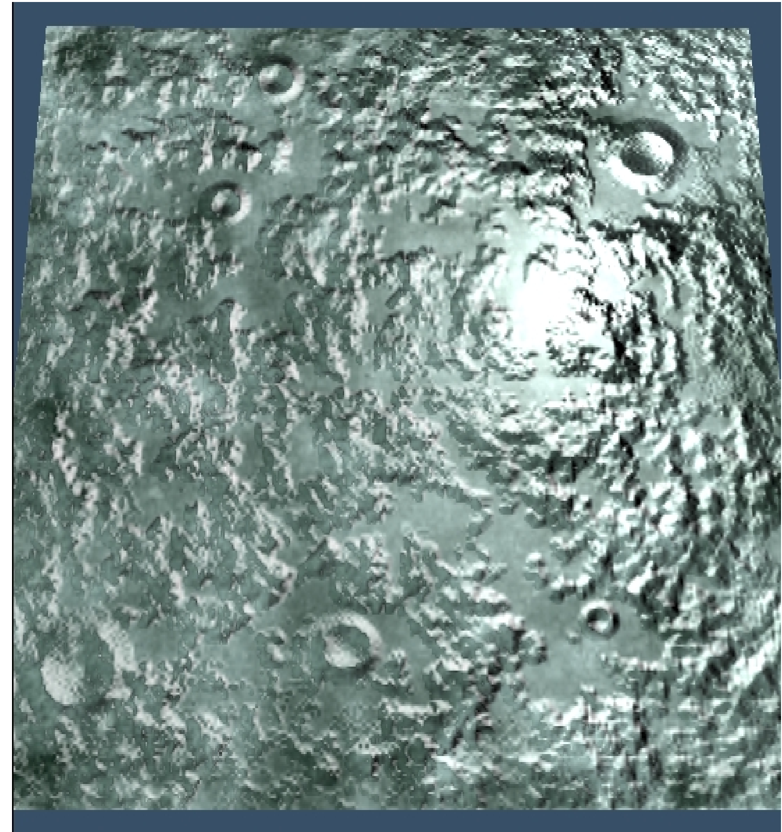
Sim Dietrich

NVIDIA Corporation

sim.dietrich@nvidia.com

Two Quads Lit Per-Pixel

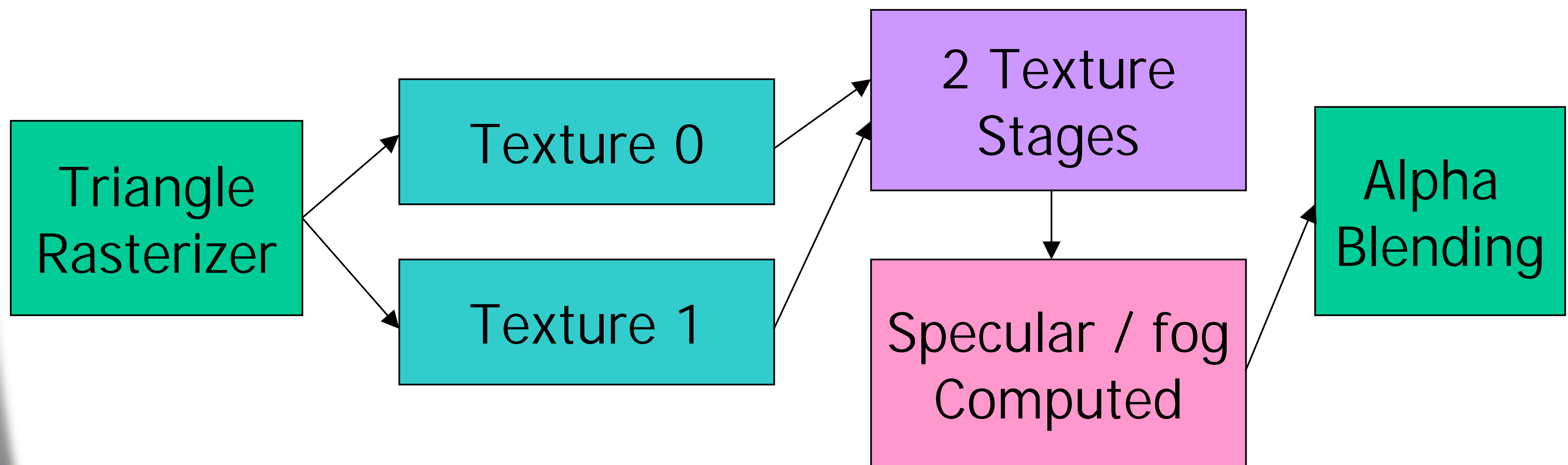




Dx8 Pixel Shading Topics

- Dx7 Pixel Pipeline
- Dx8 Pixel Pipeline
- What is a Pixel Shader?
- Instruction overview

DX7 Texture Stage Pipeline



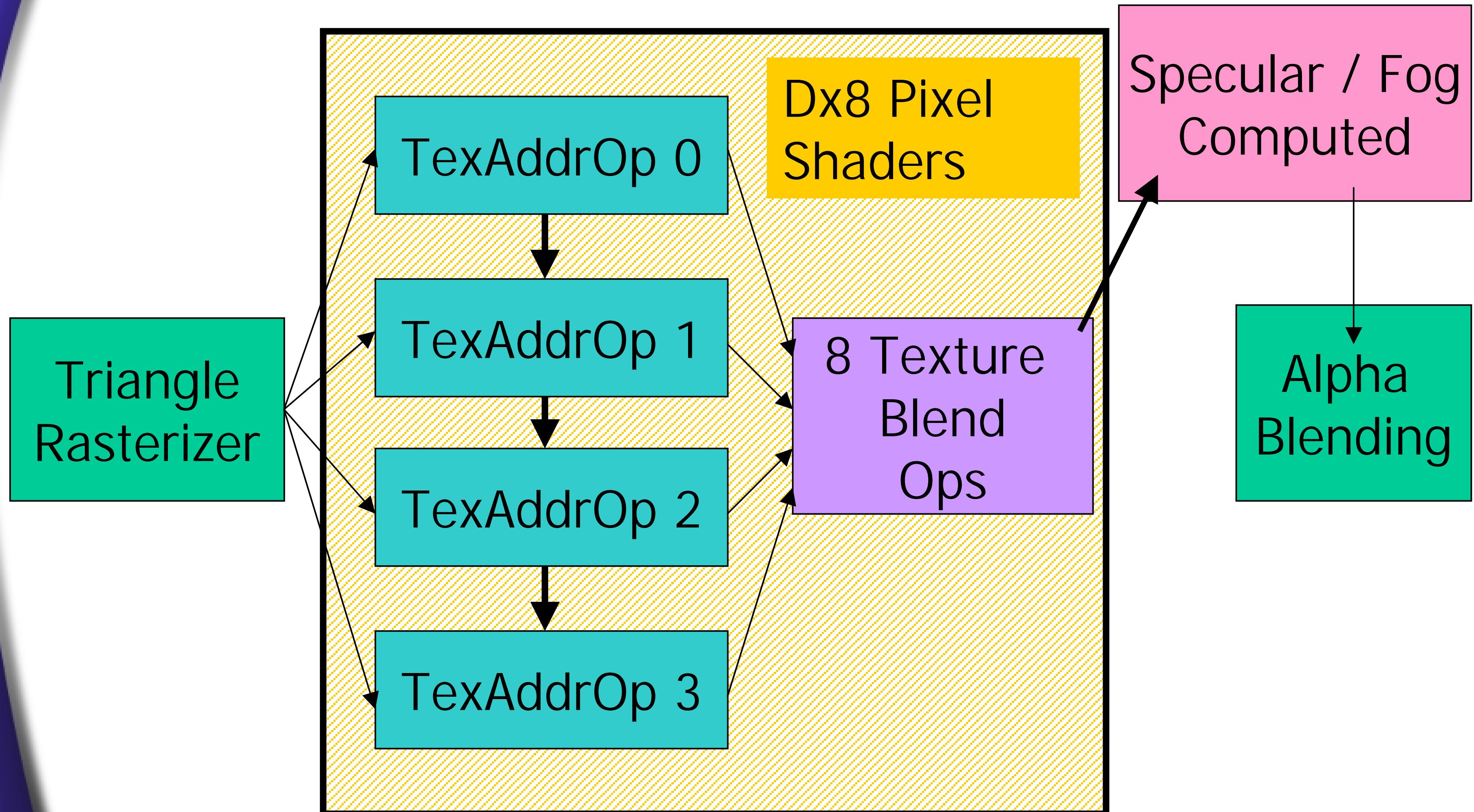
Dx7 Texture Stages

- On Dx7-class hardware, you can’t use the “Pixel Shader” API
- You can use the old dx6 & dx7-style Texture Stage API
- pDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
- But, there are two new new color ops available
  - D3DTOP_MULTIPLYADD
  - D3DTOP_LERP

Dx7-Style Multi-pass effects

- Most interesting effects are enabled through multi-pass techniques
- DOT3 is often multi-pass under D3D
  - Full texture blending functionality of hardware is sometimes hidden for the sake of API compatibility, so more passes
  - You typically end up burning a stage or two trying to control the look of the DOT3 effect anyway
    - For instance, adding ambient or per-vertex colors

DX8 Pixel Shading Pipeline



What is a Pixel Shader?

- A Pixel Shader is a byte stream of instructions compiled from a text file
- You can compile the pixel shader at runtime during development, and then change to pre-compiled byte-streams for release mode
- Assemble a Pixel shader like so :
- D3DXAssembleShader(strParsedProgram.c_str(), // source

```
D3DXAssembleShader( strParsedProgram.c_str(), // source
                    strParsedProgram.size() - 1, //size
                    0,  // no flags
                    &pConstants, // constant floats
                    pCode,           // where to put code
                    &pCompileErrors); // get errors
```

Dx8 Pixel Shaders

- Now ask D3D for a handle to the compiled pixel shader :

```
pD3DDev->CreatePixelShader(  pCode→GetBufferPointer(),
                             &m_dwMyHandle );
```

- Now select this pixel shader program like so:
- pD3DDev->SetPixelShader(m_dwMyHandle);
- Be sure to delete it

```
pD3DDev->SetPixelShader( 0 );
pD3DDev->DeletePixelShader( m_dwMyHandle );
```

DX8 Pixel Shaders

- Three types of Instructions
  - Constant Definitions
    - Similar to Setting the TFACTOR
  - Texture Address Ops
    - Fetching Texels
    - Floating Point Math
  - Texture Blending Ops
    - Combining texels, constant colors and iterated colors to produce SrcColor and SrcAlpha

Dx8 Pixel Shader Versions

- Version 1.1 – GeForce3
  - Up to 8 instructions
  - Texture registers can be read from and written to
  - EMBM takes only a single texture address slot
- Version 1.0 – Unknown Hardware
  - Up to 4 instructions
  - Texture registers are read-only
  - EMBM takes two texture address slots

Pixel Shader Parts

- Using the Pixel Shader API, there are two parts to each program
  - Up to 4 Texture Address Ops – Essentially here is where you say what each set of 4 texture coordinates are doing
    - This controls HOW the texels are fetched
  - Up to 8 Texture Blending Ops – Similar to TextureStageStates
    - This is AFTER the texels are fetched and filtered
    - There is no loopback to the Texture Address Ops

Setting Constants

def c#, x, y, z, w

Sets the Constant, from 0 to 7, with the appropriate floating point value :

def c0, 1.0f, 4.0f, -10.0f, 1.0f

Constants are clamped in the range [-1..1]

A single color/alpha instruction pair can only reference two constants

Texture Address Ops

- Each Texture Address Op represents the use of a particular set of texture coordinates
- Texture Address Ops can be used either to :
  - Look up a filtered texel color
  - Use as a vector
  - Use as the part of a matrix

Simple Texture Lookup

- Could be projective, or a cubemap, or a volume texture
- Just fetches a filtered texel color

- tex t0

```
tex t0
mov r0, t0        // just output color
```

Bump Environment Map

- texbem tDest, tSrc0
  - U += 2x2 matrix(dU)
  - V += 2x2 matrix(dV)
  - Then Sample at (U, V)

```
tex t0              // sample offset map
texbem t1, t0       // perform offset & sample
mov r0, t1          // output perturbed value
```

Bump Environment Map 2x2 Matrix

- **The 2x2 Matrix modifies the direction and scale of the dU and dV terms from the bump map**
- **The 2x2 Matrix is set via SetTextureStageState calls**

  D3DTSS_BUMPENVMAT00

  D3DTSS_BUMPENVMAT01

  D3DTSS_BUMPENVMAT10

  D3DTSS_BUMPENVMAT11

Bump Environment Map Luminance

- texbeml tDest, tSrc0
  - U += 2x2 matrix(dU)
  - V += 2x2 matrix(dV)
  - Then Sample at (U, V) & Apply Luminance

```
tex t0              // sample offset map
texbeml t1, t0      // perform offset & sample
                    // Also apply luminance & offset
mov r0, t1          // output perturbed & scaled value
```

texbeml - continued

- **The Luminance is set via a SetTextureStage State call**
- **D3DTSS_BUMPENVLSCALE**
  - **This is the amount to scale the fetched color by**
- **D3DTSS_BUMPENVLOFFSET**
  - **This is the amount to add to fetched color**

texbem & texbeml

- These instructions implement Environmental Bump Mapping (EMBM)
- EMBM is great for planar surfaces, but breaks down on anything convex or complicated
- EMBM uses an implicit 2D tangent basis, whereas you really need a 3D tangent basis to handle all objects and orientations
- That said, for water and planar surfaces, it’s the way to go

texcoord

- **Clamps the texture coordinates to the range [0.0, 1.0] and output as a color**

- **texcoord tDest**

```
texcoord t0        // pass in texture coordinates as a color
tex t1             // sample a regular texture

mov r0, t1
mul r0, r0, t0     // modulate color and texture
```

texcoord

- Useful for passing in vectors without having to use a cubemap or iterated color

```
texcoord t0      // grab L vector in tangent space
tex t1           // grab N vector in tangent space
mov r1, t1_bx2
dp3_sat r0, r1, t0_bx2 // Compute Clamp( L dot N )
```

- Can be used for 1 – d*d distance calculation for attenuation as well

```
texcoord t0                 // turn into color
dp3 r1, t0_bx2, t0_bx2      // compute d^2
mov r0, 1 - r1              // compute 1-d^2
```

Texture Kill (Clip Plane)

texkill tDest

Kill the pixel if at least one of s,t,r,q is < 0

```
tex t0              // sample a normal texture
tex t1              // sample a normal texture

texcoord t2         // clip out per-pixel based on
                    // s,t,r & q

mov r1, t1
mul r0, r1, t0      // this will get skipped if the
                    // pixel is killed
```

texm3x2pad

- Texm3x2pad t1, t0
  - “padding” instruction as part of the texm3x2tex instruction – performs a dot product of t0’s color with these texture coordinates
  - S coordinate of next stage’s texture = t1 DOT t0

texm3x2tex : Dependent Texture

- Take previous dot product from “pad” instruction as the S coordinate
- Perform dot product of t0’s color with this texture coordinate and use as T
- Sample from a 2D texture using (S, T)

```
tex t0              // sample normal map
texm3x2pad t1, t0_bx2 // t2.s = t1.texcoord dot t0.rgb
texm3x2tex t2, t0_bx2 // t2.t = t2.texcoord dot t0.rgb
```

- `mov r0, t2    // output result of lookup`

texm3x2tex

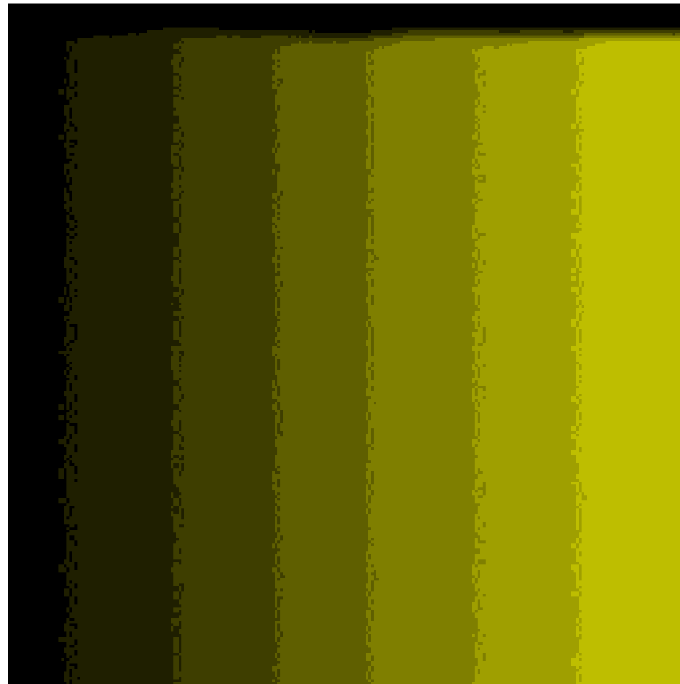
- This is possibly the most useful new instruction
- You pass in two vectors as texture coordinates (usually L and H), and sample another vector (usually N)
- The 3rd texture is sampled using
  - (L dot N) as the S texture coordinate
  - (H dot N) as the T texture coordinate
- This gives you BRDF-like anisotropic lighting on a per-pixel basis
- Great for velvet, brushed metal or any material that has intensity or hue shifts that vary with angle
- Of course, if your surface is not bumpy, then just do this per-vertex instead

texm3x2tex

- Here is a N.L N.H texture used for Anisotropic Lighting

texm3x2tex

- **You can also use it for toon shading. Rather than H dot N for the vertical dimension, perform E dot N instead. This allows the silhouette detection to pick up even the silhouette edges of bump maps**



Simple 2D Dependent Texture

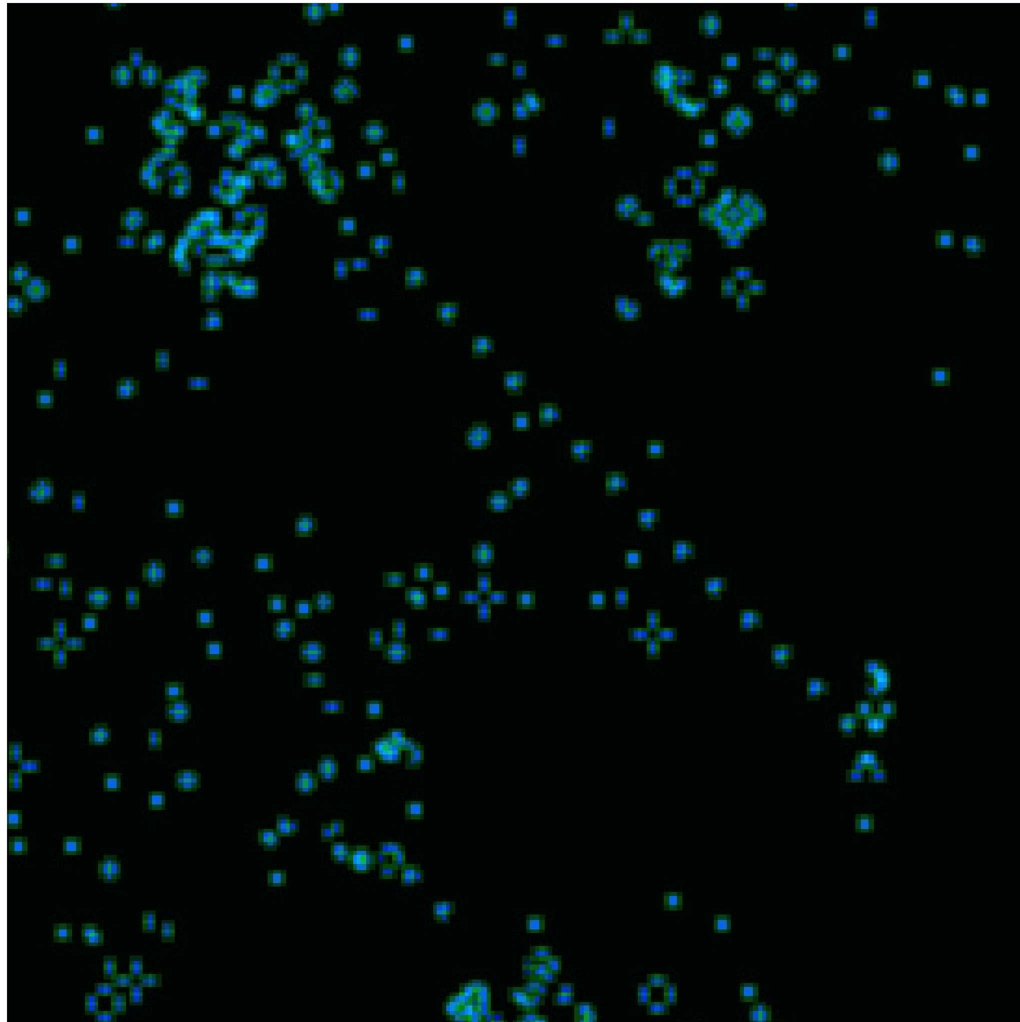
- texreg2ar tDest, tSrc
  - Sample from (tSrc.A, tSrc.R)

- texreg2gb tDest, tSrc
  - Sample from (tSrc.G, tSrc.B)

```
tex t0                  // sample regular texture
texreg2ar t1, t0        // t1.S = t0.Alpha, t1.T = t0.Red
texreg2gb t2, t0        // t2.S = t0.Green, t2.T = t0.Blue

dp3_sat r0, t1_bx2, t2_bx2
```

Game Of Life Using texreg2gb



3x3 Texture Address Ops

- Texm3x3pad
  - Padding for 3x3 matrix operation
  - Uses the 3D texture coordinate as a row of the matrix
- Texm3x3spec
  - Compute Non-Local Viewer Specular reflection about Normal from Normal Map
    - tex t0 ; Normal Map
    - texm3x3pad t1, t0 ; 1st row of matrix
    - texm3x3pad t2, t0 ; 2nd row of matrix
    - texm3x3spec t3, t0, c0 ; 3rd row, reflect & sample
    - mov r0, t3

Local Viewer Reflection

- Texm3x3vspec
  - Compute Local Viewer Specular reflection about Normal from Normal Map
  - Eye vector comes from q coordinates of the 3 sets of 4D textures

    - tex t0 ; Normal Map
    - texm3x3pad t1, t0 ; 1st matrix row, x of eyevector
    - texm3x3pad t2, t0 ; 2nd matrix row, y of eyevector
    - texm3x3vspec t3, t0 ; 3rd row & eye z, reflect & sample
    - mov r0, t3

texm3x3vspec

- This instruction is the one to use for bumpy reflective surfaces
- It is the most complex, and slowest instruction, but arguably also the most visually stunning

Procedural Normal Maps using tex3x3vspec

3x3 Per-Pixel Vector Rotation

- texm3x3tex
  - Rotate vector through 3x3 matrix, then sample a CubeMap or 3D texture

    - tex t0 ; Normal Map
    - texm3x3pad t1, t0 ; 1st matrix row
    - texm3x3pad t2, t0 ; 2nd matrix row
    - texm3x3tex t3, t0 ; 3rd matrix row & sample
    - mov r0, t3

Texture Blending Ops

- After all Texture Address Ops, you can have up to 8 texture blending instruction slots
- Each slot can hold a color and an alpha operation to be executed simultaneously
- These are analogous to the old TextureStageState COLOROP and ALPHAOPs

Texture Blending Ops

add dest, src1, src2

dest = src1 + sr2

sub dest, src1, src2

dest = src1 – src2

lrp dest, factor, src1, src2

dest = (factor)src1 + (1-factor)src2

dp3 dest, src1, src2

dest = (src1.x * src2.x + src1.y * src2.y …)

dp3

- This is the workhorse instruction
  - It is used for all per-pixel lighting calculations in the texture blending unit
  - Typically you want to _sat your dot3 in lighting calculations to prevent lights behind a surface from showing up, so mostly you will use something like :

```
dp3_sat r0, t0_bx2, r1
```

  - The bx2 modifier is there to take an 8 bit unsigned value and expand it to 1.8 signed format
  - Note that the registers keep their sign bit, so be sure to use _bx2 only once after storing the signed value

Texture Blending Ops

```
mul    dest, src0, src1
   dest = src0 * src1

mad dest, src0, src1, src2
   dest = ( src0 * src1 + src2 )

mov dest, src
   dest = src

cnd dest, r0.a, src1, src2
   if ( r0.a > 0.5 ) { dest = src1;  }
 else  { dest = src2; }
```

Argument Modifiers

- Alpha Replicate
  - r0.a
- Invert
  - 1 – r0
- Negate
  - -r0
- Bias – subtract 0.5
  - r0_bias
- Signed Scale – 2 * (x – 0.5f)
  - r0_bx2

Instruction Modifiers

_x2 // double result

_x4 // quadruple result

_d2 // halve result

_sat // clamp < 0 to 0 and > 1 to 1

You can use _sat together with scaling :

For instance :

```
add_x2_sat r0, r1, t2
```

Common Example :

```
dp3_sat   r1, r0_bx2, t0_bx2
```

Simultaneous Color/Alpha

- **You can dual issue color and alpha instructions**
- **Only applies for blending ops, not for address ops**
- **Use the '+' plus sign to indicate simultaneous execution**
- **mul r0.rgb, c1.rgb, c2.rgb**

+ add r1.a, t0.a, t1.a

Example Pixel Shader

- ps.1.1 ; DirectX8 Version
- tex t0 ; sample normal map
- texm3x2pad t1, t0_bx2 ; N dot L
- texm3x2tex t2, t0_bx2 ; N dot H and sample
- add r0, t2, c0 ; add in ambient
- mov r0.a, t0.a ; normal map alpha into r0

Additional Pixel Shader Notes

- The result of a pixel shader is always r0
  - SRCCOLOR = r0.rgb
  - SRCALPHA = r0.a
- You can combine rgb & alpha operations to occur concurrently with "+"
  - add r0.rgb, t1.rgb t0.rgb

  \+ mul r0.a, r1.a, t1.a
- You are in charge of your own specular add, SPECULAR_ENABLE is ignored
- Fog is still automatic

Vertex & Pixel Shaders

- It is common to write Vertex Shaders that use the legacy TextureStageState pipeline to do the texture blending
- However, almost all Pixel Shaders have a Vertex Shader to set things up properly
  - Calculating & Packing L and H vectors
  - Passing in Texture Space vectors
  - Calculating and/or setting up Attenuation

Questions…

Sim Dietrich

Sim.Dietrich@nvidia.com

www.nvidia.com/Developer.nsf