Vertex Programs
Texture Shaders

Erik Lindholm
3D Architect
NVIDIA Corp.

Vertex Programs

Vertex Programs Overview

- Fixed Function Pipeline
- Vertex Program Model
- OpenGL support
- Simple OpenGL Examples
- Appendix A: Vertex Program Instructions
- Appendix B: Fixed Function Emulation

Vertex Programs:
Fixed Function Pipeline

Fixed Function Pipeline

- SGI IrisGL (1983) and OpenGL (1992) xform and light (T&L) pipeline
- D3D (1997) xform and light pipeline

- Provide a limited set of effects that were felt to be reasonable for hw
- These effects mostly date from the 1970's

7-5

Fixed Function Pipeline

- Current T&L pipelines are configurable not programmable.
- Yet native hw is often programmable
- Why not expose this programmability?
  - Designs are highly customized for performance, very difficult to program
  - Future compatibility problems

7-6

Fixed Function Pipeline

- Vendor tends to code a general (slow) path for all modes
- Important mode combinations implemented with highly tuned code
- Customers and benchmarks tend to determine what "important" means
- Need a new release to expose improvements

7-7

Fixed Function Pipeline

- Customers sometimes request custom enhancements
- Vendor must decide whether to devote scarce coding resources
- Customer must wait for new code release
- Common requests might become extensions
- Extensions sometimes become core, although it might take years

7-8

Fixed Function Pipeline

Mode explosion in T&L

- User clip planes - 2^6
- Color Material - 36 (or more)
- Fog - 4
- Lights - 4/light (off, infinite, local, spot)
- LightModel - 4
- Texgen - 144/texture (or more)

7-9

Fixed Function Pipeline

- Assuming 1 texture and 8 lights we get about half a trillion combinations
- Assuming 4 textures and 8 lights we easily break 1 quintillion
- Fortunately driver compaction and looping in microcode greatly simplifies this
- Unfortunately this costs performance

7-10

Fixed Function Pipeline

- Performance drives fewer features
  - Which requires less modes
    - making tuning easier
- Features sell new hw
  - which requires more modes
    - making tuning harder
- New modes are expensive to support

7-11

Fixed Function Pipeline

- Odds are that a particular mode a user cares about is not optimized unless it is a standard path that everyone else or an important benchmark is using
- Most users do not care about most of these modes so why pay for the complexity?
- Users can take shortcuts not known to hw for higher performance

7-12

Fixed Function Pipeline

- Pixel shader support greatly complicates xform features
- Much harder to determine exactly what these new xform features would be
- So don't try

Fixed Function Pipeline

- Problem is performance AND flexibility
- Solution is vertex programs

Vertex Programs:
Vertex Program Model

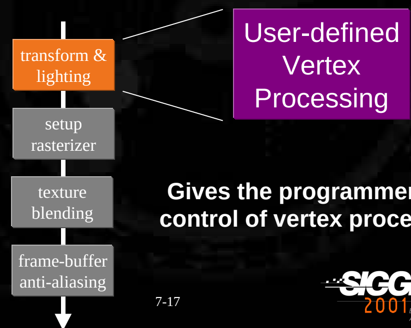
The instruction set is supported in OpenGL through NVIDIA extensions and is native in DX8

Vertex Program Model

- Allow user to program xform engine
- Assembly language interface
- Hw natively supports programming language and also uses it for fixed function mode
- Streaming 1 vertex in and 1 vertex out model
- Cannot create or destroy vertices
- SIMD 4 wide IEEE 32bit fp
- All instructions are equal performance

Vertex Program Model

- Vertex Programming offers programmable T&L unit



Gives the programmer total control of vertex processing.

7-17

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Vertex Program Model

Reads an untransformed, unlit vertex

Creates a transformed vertex

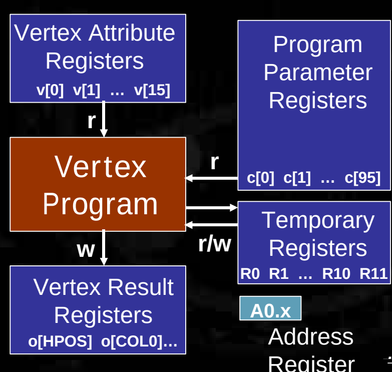
Optionally:

- Lights vertex
- Creates/modifies texture coordinates
- Creates/modifies fog coordinate
- Creates point size

7-18

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Vertex Program Model



7-19

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Program Model

128 instruction program

16 quad-float vertex inputs (single read addr)

96 quad-float constants (single read addr)

12 quad-float registers (triple read addr)

1 address register

15 quad-float vertex outputs. Outputs are initialized to (0,0,0,1) at start of program

7-20

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Program Model: Input

Attribute Register	Conventional per-vertex Attribute	Conventional Command	Conventional Mapping
0	Vertex position	glVertex	x,y,z,w
1	Vertex weights	glVertexWeightEXT	w,0,0,1
2	Normal	glNormal	x,y,z,1
3	Primary color	glColor	r,g,b,a
4	Secondary color	glSecondaryColorEXT	r,g,b,1
5	Fog coord	glFogCoordEXT	f,0,0,1
6	NA		
7	NA		

Program Model: Input (cont'd)

Attribute Register	Conventional per-vertex Attribute	Conventional Command	Conventional Mapping
8	Texcoord 0	glMultiTexCoord	s,t,r,q
9	Texcoord 1	glMultiTexCoord	s,t,r,q
10	Texcoord 2	glMultiTexCoord	s,t,r,q
11	Texcoord 3	glMultiTexCoord	s,t,r,q
12	Texcoord 4	glMultiTexCoord	s,t,r,q
13	Texcoord 5	glMultiTexCoord	s,t,r,q
14	Texcoord 6	glMultiTexCoord	s,t,r,q
15	Texcoord 7	glMultiTexCoord	s,t,r,q

Program Model: Output

Register Name	Description	Component Interpretation
o[HPOS]	Homogeneous clip space position	(x,y,z,w)
o[COL0]	Primary color (front-facing)	(r,g,b,a)
o[COL1]	Secondary color (front-facing)	(r,g,b,a)
o[BFC0]	Back-facing primary color	(r,g,b,a)
o[BFC1]	Back-facing secondary color	(r,g,b,a)
o[FOGC]	Fog coordinate	(f,*,*,*)
o[PSIZ]	Point size	(p,*,*,*)

Program Model: Output (cont'd)

Register Name	Description	Component Interpretation
o[TEX0]	Texture coordinate set 0	(s,t,r,q)
o[TEX1]	Texture coordinate set 1	(s,t,r,q)
o[TEX2]	Texture coordinate set 2	(s,t,r,q)
o[TEX3]	Texture coordinate set 3	(s,t,r,q)
o[TEX4]	Texture coordinate set 4	(s,t,r,q)
o[TEX5]	Texture coordinate set 5	(s,t,r,q)
o[TEX6]	Texture coordinate set 6	(s,t,r,q)
o[TEX7]	Texture coordinate set 7	(s,t,r,q)

Program Model: Constants

- Only loadable outside of the vertex stream (e.g. outside glBegin/glEnd)
- Useful for matrices, positions, vectors, etc…
- Can be addressed with absolute or relative address
- Relative addressing uses address register A0
- A0 is loadable via instruction
- Out of range A0 reads return (0,0,0,0)

7-25

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Program Model: Registers

- There are 12 temporary quad-float registers
- Triple read port and single write port
- Size chosen for simple modular code design
- Registers are initialized to (0,0,0,0) at start of each vertex program

7-26

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Program Model: Instructions

There are 17 instructions in total …

• ARL	• RSQ	• SLT
• MOV	• DP3	• SGE
• MUL	• DP4	• EXP
• ADD	• DST	• LOG
• MAD	• MIN	• LIT
• RCP	• MAX	

7-27

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Input Modifiers

- Any input vector can be negated by the "-" prefix. All vector components are negated.
- Any input vector can be swizzled, i.e. have its components arbitrarily rearranged or replicated via variants of the ".xyzw" postfix.
- Input modifiers are free

7-28

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Source Swizzling

- R0 is the same as R0.xyzw
- R0.x is the same as R0.xxxx
- R0.y is the same as R0.yyyy
- R0.z is the same as R0.zzzz
- R0.w is the same as R0.wwww
- All 256 combinations of the 4 subscripts are legal. Except for above shortcuts, swizzling requires 4 subscripts (from x,y,z,w).

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Output Modifiers

- Any output can be write-masked via the ".xyzw" postfix. Only enabled components are written.
- Example: a destination of R4.xw only updates the x and w components.
- Valid writemasks list x before y before z before w. No writemask is the same as .xyzw.

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Vertex Programs: OpenGL Support

(For DX8, please refer to Microsoft DX8 documentation)

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Vertex Program Specification

Programs are arrays of Glubytes ("strings")

Invoked when glVertex issued

Created/managed similar to texture objects

- glGenProgramsNV(sizei n, uint *ids)
- glLoadProgramNV(enum target, uint id,
  sizei len, const ubyte *program)
- glBindProgramNV(enum target, uint id)

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Sample Vertex Program

```
static const GLubyte v_pgm[] = "\
   !!VP1.0                    \
   MOV o[HPOS],v[0];      \
   MOV o[COL0],v[3];      \
   END\
";
```

7-33

Vertex Program Attributes

Vertex has up to 16 quad-floats of input

Values specified with the new commands:

- glVertexAttrib4fNV(index,…)
- glVertexAttribPointerNV(index,…)

Attributes also specified through conventional per-vertex parameters via aliasing (e.g. glColor4f)

7-34

Vertex Program Parameters

96 quad-float parameters

Values specified with new commands

- glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, index, x, y, z, w);
- glProgramParameters4fNV(GL_VERTEX_PROGRAM_NV, index, n, params)

Correspond to 96 registers (c[0],…,c[95])

7-35

Vertex Program Matrix Tracking

- Matrixes can be "tracked"
- Makes matrices automatically available in vertex program's parameter registers
- MODELVIEW, PROJECTION, COMPOSITE, TEXTUREi, MATRIXi can be mapped
- Mapping can be IDENTITY, TRANSPOSE, INVERSE, and INVERSE_TRANSPOSE
- Eight new user matrices (MATRIXi)

7-36

Vertex Programs: Simple OpenGL Examples

(DX8 is very similar)

7-37

Absolute Value

```
# |source| = MAX source,-source
MAX R0,R1,-R1;

# -|source| = MIN source,-source
MIN R0,R1,-R1;
```

7-38

Cross Product

```
# Cross product R2 = R0 x R1
MUL R2, R0.zxyw, R1.yzxw;
MAD R2, R0.yzxw, R1.zxyw, -R2;
```

7-39

3x3 Determinant

```
# R3 = | R0.x  R0.y  R0.z |
#      | R1.x  R1.y  R1.z |
#      | R2.x  R2.y  R2.z |
MUL R3, R1.zxyw, R2.yzxw;
MAD R3, R1.yzxw, R2.zxyw, -R3;
DP3 R3, R0, R3;

# good for matrix cofactors
```

7-40

Normalize Vector

```
# normalize R1 = (nx,ny,nz,NA)
DP3 R1.w, R1, R1;
RSQ R1.w, R1.w;
MUL R1.xyz, R1, R1.w;
```

7-41

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Reduce To Period

```
# reduce radian angle to [0,2PI]
# c[0] contains (1/(2PI),2PI,NA,NA)
MUL R0.x,v[0].x,c[0].x;
EXP R0.y,R0.x;                  # fraction
MUL R0.x,R0.y,c[0].y
```

7-42

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Matrix[4][4] * Vector[4]

```
# Assume matrix in constants 0,1,2,3
# Source vector in R5, output in R6
# Vertex data specifies matrix address
ARL A0.x,v[6].x;
DP4 R6.x, R5, c[0+A0.x];
DP4 R6.y, R5, c[1+A0.x];
DP4 R6.z, R5, c[2+A0.x];
DP4 R6.w, R5, c[3+A0.x];
```

7-43

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Matrix[4][4] Inversion

Registers R4,R5,R6,R7 contain the matrix

Invert matrix into registers R8,R9,R10,R11

7-44

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Matrix[4][4] Inversion: Part 1/5

```
# generate first half of matrix
MUL R0,R6.wyzx,R7.zwyx;
MUL R1,R4.wyzx,R5.zwyx;
MUL R2,R6.wxzy,R7.zwxy;
MUL R3,R4.wxzy,R5.zwxy;
MAD R0,R6.zwyx,R7.wyzx,-R0;
MAD R1,R4.zwyx,R5.wyzx,-R1;
MAD R2,R6.zwxy,R7.wxzy,-R2;
MAD R3,R4.zwxy,R5.wxzy,-R3;
```

7-45

Matrix[4][4] Inversion: Part 2/5

```
# generate first half of matrix
DP3 R8.x,R5.yzwx,R0;
DP3 R9.x,R4.yzwx,-R0;
DP3 R10.x,R7.yzwx,R1;
DP3 R11.x,R6.yzwx,-R1;
DP3 R8.y,R5.xzwy,-R2;
DP3 R9.y,R4.xzwy,R2;
DP3 R10.y,R7.xzwy,-R3;
DP3 R11.y,R6.xzwy,R3;
```

7-46

Matrix[4][4] Inversion: Part 3/5

```
# generate second half of matrix
MUL R0,R6.wxyz,R7.ywxz;
MUL R1,R4.wxyz,R5.ywxz;
MUL R2,R6.zxyw,R7.yzxw;
MUL R3,R4.zxyw,R5.yzxw;
MAD R0,R6.ywxz,R7.wxyz,-R0;
MAD R1,R4.ywxz,R5.wxyz,-R1;
MAD R2,R6.yzxw,R7.zxyw,-R2;
MAD R3,R4.yzxw,R5.zxyw,-R3;
```

7-47

Matrix[4][4] Inversion: Part 4/5

```
# generate second half of matrix
DP3 R8.z,R5.xywz,R0;
DP3 R9.z,R4.xywz,-R0;
DP3 R10.z,R7.xywz,R1;
DP3 R11.z,R6.xywz,-R1;
DP3 R8.w,R5.xyzw,-R2;
DP3 R9.w,R4.xyzw,R2;
DP3 R10.w,R7.xyzw,-R3;
DP3 R11.w,R6.xyzw,R3;
```

7-48

Matrix[4][4] Inversion: Part 5/5

```
# calculate and divide by determinant
DP4 R7.w,R8,R4;
RCP R7.w,R7.w;
MUL R8,R8,R7.w;
MUL R9,R9,R7.w;
MUL R10,R10,R7.w;
MUL R11,R11,R7.w;
```


Power Series

```
# 16 term power series with input scalar X
DST R0,X,X;
MUL R0,R0.xzyw,R0.xxxy;   /* 1,x,x^2,x^3 */
DP4 R1.x,R0,c[ABCD];
DP4 R1.y,R0,c[EFGH];
DP4 R1.z,R0,c[IJKL];
DP4 R1.w,R0,c[MNOP];
MUL R0,R0,R0;
MUL R0,R0,R0;             /* 1,x^4,x^8,x^12 */
DP4 R0,R0,R1;
```


Xform/Light Example

```
# c[0-3]  = modelview+projection matrix
# c[4-7]  = modelview inverse transpose
# c[32]   = eye-space directional light
# c[33]   = eye-space half-angle vector
# c[34]   = ambient color
# c[35]   = diffuse color
# c[36]   = specular color
# c[36].w = specular power
```


Xform/Light Example: Part 1/2

```
# transform normal
DP3  R0.x, c[4], v[NRML];
DP3  R0.y, c[5], v[NRML];
DP3  R0.z, c[6], v[NRML];
# transform position
DP4  o[HPOS].x, c[0], v[OPOS];
DP4  o[HPOS].y, c[1], v[OPOS];
DP4  o[HPOS].z, c[2], v[OPOS];
DP4  o[HPOS].w, c[3], v[OPOS];
```

Xform/Light Example: Part 2/2

```
# light
DP3   R1.x, c[32], R0;             # n*l
DP3   R1.y, c[33], R0;             # n*h
MOV   R1.w, c[36].w;               # power
LIT   R2, R1;                      # lighting
MOV   R3, c[34];                   # ambient
MAD   R3, c[35], R2.y,R3;          # diffuse
MAD   o[COL0].xyz, c[36], R2.z, R3; # sp
```

7-53

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Texture Shaders

7-54

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Texture Shaders Overview

- Pipeline
- Conventional texture shaders
- Special mode texture shaders
- Simple dependent textures
- Dot product dependent textures
- Reflective Bump Map Demo

7-55

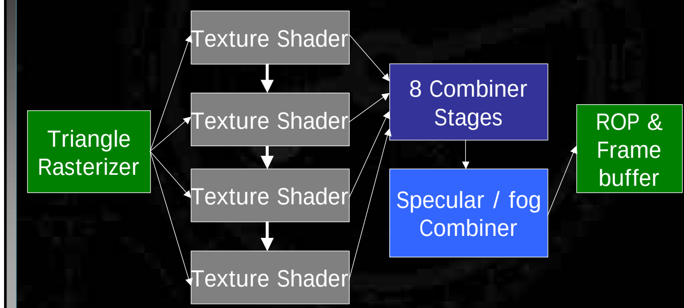
SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Texture Shaders: Pipeline

7-56

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

GeForce3 Shader Pipeline



Units

Texture shader

- 4 texture units
- 21 different texture shader operations
  - conventional (1D, 2D, texture rectangle, cubemap)
  - special case (none, pass through, cull fragment)
  - dependent texture fetches (result of one texture lookup affects texcoords for subsequent unit)
  - dependent textures fetches with dot product (and optional reflection) calculations

Register combiners

- 8 stages (general combiners)

Considerations

- When texture shaders are enabled, they are *ALL* enabled
- Select GL_NONE shader for unused stages
- Several texture shader operations return (0,0,0,0) - if not using register combiners, ensure TEX_ENV_MODE is GL_NONE
- Shader operations implicitly determine which texture is accessed (if any) as opposed to unextended OpenGL, where enabled texture targets have pre-set precedence

Texture Shaders:
Conventional Texture Shaders

Conventional Texture Shaders

- Texture 1D
- Texture 2D
- Texture rectangle
- Texture cube map

Texture 1D

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
				1D Any Format	
i	(S_i,T_i,R_i,Q_i) →	Texture 1D	$(\frac{S_i}{Q_i})$ →		→ (R,G,B,A)

Texture 2D

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
				2D Any Format	
i	(S_i,T_i,R_i,Q_i) →	Texture 2D	$(\frac{S_i}{Q_i},\frac{T_i}{Q_i})$ →		→ (R,G,B,A)

Texture Rectangle

- New texture target defined via new extension - NV_texture_rectangle
- Allows for non-power-of-2 width and height (e.g. 640x480)
- S and T texcoords address [0,width] and [0,height] respectively, instead of [0,1] as in Texture 2D
- Can be used independently from texture shader
- No mipmaps

Texture Rectangle

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
i	(S_i,T_i,R_i,Q_i) →	Texture Rectangle	$(\frac{S_i}{Q_i}, \frac{T_i}{Q_i})$ →	Texture Rectangle Any Format	(R,G,B,A)

Texture Cube Map

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
i	(S_i,T_i,R_i) →	Texture Cube Map	$U=(S_i, T_i, R_i)$ →	Cube Map Any Format (U)	$R_3G_3B_3A_3$

Texture Shaders:
Special Mode Texture Shaders

Special Mode Texture Shaders

- None
- Pass through
- Cull fragment

None

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
i	Ignored	None	None	None	→ (0,0,0,0)

Pass Through

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
i	(S_i,T_i,R_i,Q_i) →	R = Clamp0to1(S_i) G = Clamp0to1(T_i) B = Clamp0to1(R_i) A = Clamp0to1(Q_i)	None	None	→ (R,G,B,A)

Cull Fragment

Cull the fragment based upon sign of texcoords
- each texcoord (STRQ) has its own settable condition
- each of 4 conditions is set to one of the following:
  - GL_GEQUAL (texcoord   0) – pass iff positive or 0
  - GL_LESS (texcoord < 0) – pass iff negative
- all four texcoords are tested
- if any of the four fail, the fragment is rejected

No texture accesses, outputs (0,0,0,0)

Very useful for per-pixel user-defined clipping
– up to 4 per texture unit (16 total!)

Texture Shaders:
Simple Dependent Shaders

Simple Dependent Texture Shaders

- Dependent alpha-red
- Dependent green-blue
- Offset texture 2D
- Offset texture 2D scaled

7-73

Simple Dependent Texture Shaders

- Take results of one texture, use them for addressing subsequent texture
- Single stage, not including source texture
- Simple dependent textures (single stage)
- All diagrams from here on out start at texture unit 0 and use a contiguous series of texture units
  - This is an artificial restriction to ease in explaining the concepts of these texture shaders

7-74

Dependent Alpha-Red Texturing

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	App specific →	Texture specific	Texture specific	Any type Unsigned RGBA	→ $R_0G_0B_0A_0$
1	Ignored	None	(A_0,R_0)	2D RGBA	→ $R_1G_1B_1A_1$

7-75

Dependent Green-Blue Texturing

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	App specific →	Texture specific	Texture specific	Any type Unsigned RGB[A]	→ $R_0G_0B_0A_0$
1	Ignored	None	(G_0,B_0)	2D RGBA	→ $R_1G_1B_1A_1$

7-76

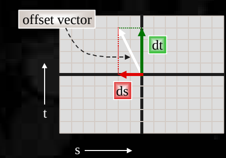
Offset Texture 2D

- Use previous lookup (a signed 2D offset) to perturb the texcoords of a subsequent (non-projective) 2D texture lookup
- Signed 2D offset is transformed by user-defined 2x2 matrix (shown in the following diagrams as constants k_0-k_3)
- This 2x2 constant matrix allows for arbitrary rotation/scaling of offset vector
- This shader operation can be used for what is called EMBM in DirectX 6 lingo
- Offset defined in DS/DT texture

What is a DS/DT Texture?

Format encodes texture space 2D offset vector

- **ds** and **dt** are mapped to the range [-1,1]



MAG and MAG/Intensity flavors use the third and fourth component to optionally include scaling and luminance

Offset Texture 2D

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	(S_0,T_0,R_0,Q_0) →	Texture 2D	$(\frac{S_0}{Q_0}, \frac{T_0}{Q_0})$	2D DSDT →	(0,0,0,0)
1	(S_1,T_1) →	$S_1' = S_1 + k_0{*}ds + k_2{*}dt$ $T_1' = T_1 + k_1{*}ds + k_3{*}dt$	(S_1', T_1')	2D Any Format →	$R_1G_1B_1A_1$

k_0, k_1, k_2 and k_3 define a constant 2x2 matrix

Offset Texture 2D Scale

- Same as Offset Texture 2D, except that subsequent (non-projective) 2D texture RGB output is scaled
- Scaling factor is the MAG component (from previous texture) scaled/biased by user-defined constants (k_{scale} and k_{bias})
- Alpha component is NOT scaled
- For GL_DSDT_MAG_INTENSITY_NV, the previous texture output is the intensity component, else it is (0,0,0,0)

Offset Texture 2D Scale

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	(S_0,T_0,R_0,Q_0) →	Texture 2D	$(\frac{S_0}{Q_0}, \frac{T_0}{Q_0})$	2D DSDT_Mag → (ds, dt, mag)	→ (0,0,0,0)
1	(S_1,T_1) →	$S_1' = S_1 + k_0 \cdot ds + k_2 \cdot dt$ $T_1' = T_1 + k_1 \cdot ds + k_3 \cdot dt$	(S_1', T_1')	2D RGBA	→ $(R \cdot M, G \cdot M, B \cdot M, A)$

$M = k_{scale} \cdot mag + k_{bias}$

k_0, k_1, k_2 and k_3 define a constant 2x2 matrix

k_{scale} and k_{bias} define constant scale/bias

7-81

Offset Texture Issues

Offset texturing also available for texture rectangles, in addition to 2D textures

Limited precision in DSDT formats (max 8-bits per component)

- Don't scale DS/DT values by more than 8 so as to preserve sub-texel precision
- Limits texcoord perturbation to [-8,8] (or so)
- Applications needing to perturb texcoords by more than this should use Dot Product Texture 2D (explained in next section) with HILO textures

7-82

Texture Shaders: Dot Product Dependent Shaders

7-83

Dot Product Dependent Shaders

- dp texture 2D
- dp texture rectangle
- dp texture cube map
- dp constant eye reflect cube map
- dp reflect cube map
- dp diffuse cube map
- dp depth replace

7-84

DP Dependent Texture Shaders

- Take results of one texture, perform 2 or 3 dot products with it and incoming texcoords, then use results for addressing subsequent texture(s)
- Multiple contiguous stages, not including source texture

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Dot Product

- Calculates a high-precision dot product
- All dot product operations can be considered to perform this operation, the others just do something with the resulting scalars
- Source (previous) texture can have one of the following internal formats:
  - Signed RGBA (used in all the diagrams)
  - Unsigned RGBA (expandable to [-1,1])
  - Signed HILO
  - Unsigned HILO

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

RGBA texture formats

- Very useful for arbitrary vector encoding
- Signed RGB[A]
  - New formats (GL_SIGNED_RGB_NV, etc.)
  - Three (or four) 8-bit signed components in [-1,1]
- Unsigned RGB[A]
  - Three (or four) 8-bit unsigned components in [0,1]
  - All components can be expanded to [-1,1] range prior to any dot product shader operation

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

HILO texture formats

- Two 16-bit channels (high and low)
- Signed HILO (GL_SIGNED_HILO_NV)
  - Both components are [-1,1]
  - Useful for encoding normals with high precision
  - Third channel is hemispherical projection of first 2

$$\left(HI, LO, \sqrt{1 - HI^2 - LO^2}\right)$$

- Unsigned HILO (GL_HILO_NV)
  - Both components are [0,1]
  - Useful for encoding 32-bit values, like depth
  - Third channel is set to 1

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

HILO Advantages

- Filtering for each component done in 16-bits
- Hemispherical projection *after* filtering
- Always results in unit length vector

Single bump mapped quad

Dot Product Texture 2D

- Previous stage must be Dot Product
- Two dot products as 3x2 matrix/vector mult:

$$\begin{bmatrix} S' \\ T' \end{bmatrix} = \mathbf{M}\vec{n} = \begin{bmatrix} S_0 & T_0 & R_0 \\ S_1 & T_1 & R_1 \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$$

- Matrix is "Texel Matrix", and transforms previous texture result (e.g. a normal) from R^3 to R^2, then uses transformed 2D vector to access a 2D texture

Dot Product Texture 2D

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	App specific →	Texture specific	Texture specific	Any type Signed RGB[A]	→ $R_0G_0B_0$
1	(S_1, T_1, R_1) →	$U_x = [S_1, T_1, R_1] \bullet [R_0, G_0, B_0]$	None	None	→ (0,0,0,0)
2	(S_2, T_2, R_2) →	$U_y = [S_2, T_2, R_2] \bullet [R_0, G_0, B_0]$	(U_x, U_y)	2D RGBA	→ $R_2G_2B_2A_2$

Dot Product Texture Rectangle

- Previous stage must be Dot Product
- Similar to Dot Product Texture 2D, except that subsequent texture target is a texture rectangle, instead of a 2D texture

Dot Product Texture Rectangle

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	App specific →	Texture specific	Texture specific	Any type Signed RGB[A]	→ $R_0G_0B_0$
1	(S_1, T_1, R_1) →	$U_x=[S_1,T_1,R_1] \bullet [R_0,G_0,B_0]$	None	None	→ (0,0,0,0)
2	(S_2, T_2, R_2) →	$U_y=[S_2,T_2,R_2] \bullet [R_0,G_0,B_0]$	(U_x, U_y)	Texture Rectangle RGBA	→ $R_2G_2B_2A_2$

7-93

Dot Product Texture Cube Map

- Previous two stages must be Dot Product
- Three dot products as 3x3 matrix/vector mult:

$$\vec{n}' = \mathbf{M}\vec{n} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$$

- "Texel Matrix" transforms previous texture result (e.g. normal), then accesses cube map
- Matrix above moves normal map vector from tangent to modelview space

7-94

Dot Product Texture Cube Map

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	App specific →	Texture specific	Texture specific	Any type Signed RGB[A]	→ $R_0G_0B_0$
1	(T_x, B_x, N_x) →	$U_x=[T_x,B_x,N_x] \bullet [R_0,G_0,B_0]$	None	None	→ (0,0,0,0)
2	(T_y, B_y, N_y) →	$U_y=[T_y,B_y,N_y] \bullet [R_0,G_0,B_0]$	None	None	→ (0,0,0,0)
3	(T_z, B_z, N_z) →	$U_z=[T_z,B_z,N_z] \bullet [R_0,G_0,B_0]$ $\mathbf{U} = (U_x, U_y, U_z)$		Cubemap RGBA	→ $R_3G_3B_3A_3$

7-95

DP Constant Eye Reflect Cube Map

- Similar to Dot Product Texture Cube Map, except that vector accessing the cube map (R) is computed as the reflection of the eye vector about the transformed normal
- The eye vector is passed in as constants (i.e. an infinite viewer)

$$\mathbf{E} = (E_x, E_y, E_z)$$

$$\vec{n}' = \mathbf{M}\vec{n} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \qquad \mathbf{R} = \frac{2n'(n' \bullet \mathbf{E})}{(n' \bullet n')} - \mathbf{E}$$

7-96

DP Constant Eye Reflect Cube Map

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	App specific →	Texture specific	Texture specific	Any type Signed RGB[A]	→ $R_0G_0B_0$
1	(T_x, B_x, N_x) →	$U_x = [T_x,B_x,N_x] \bullet [R_0,G_0,B_0]$	None	None	→ (0,0,0,0)
2	(T_y, B_y, N_y) →	$U_y = [T_y,B_y,N_y] \bullet [R_0,G_0,B_0]$	None	None	→ (0,0,0,0)
3	(T_z, B_z, N_z) →	$U_z = [T_z,B_z,N_z] \bullet [R_0,G_0,B_0]$ $\mathbf{U} = (U_x, U_y, U_z)$ $\mathbf{E} = (E_x, E_y, E_z)$ $\mathbf{R} = \frac{2\mathbf{U}(\mathbf{U}\bullet\mathbf{E})}{(\mathbf{U}\bullet\mathbf{U})} - \mathbf{E}$	$\mathbf{R}=(R_x, R_y, R_z)$	Cubemap RGBA	→ $R_3G_3B_3A_3$

7-97

Dot Product Reflect Cube Map

Same as Constant Eye Reflect Cube Map, except that the eye vector is passed in through the Q coordinate of the three dot product stages

Eye in this case is "local", resulting in better, more realistic, images as it is interpolated across all polygons

$$\vec{n}' = \mathbf{M}\vec{n} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$$

$$\mathbf{E} = (q_0, q_1, q_2)$$

$$\mathbf{R} = \frac{2n'(n'\bullet\mathbf{E})}{(n'\bullet n')} - \mathbf{E}$$

7-98

Dot Product Reflect Cube Map

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	App specific →	Texture specific	Texture specific	Any type Signed RGB[A]	→ $R_0G_0B_0$
1	$([T_x, B_x, N_x], E_x)$ →	$U_x = [T_x,B_x,N_x] \bullet [R_0,G_0,B_0]$	None	None	→ (0,0,0,0)
2	$([T_y, B_y, N_y], E_y)$ →	$U_y = [T_y,B_y,N_y] \bullet [R_0,G_0,B_0]$	None	None	→ (0,0,0,0)
3	$([T_z, B_z, N_z], E_z)$ →	$U_z = [T_z,B_z,N_z] \bullet [R_0,G_0,B_0]$ $\mathbf{U} = (U_x, U_y, U_z)$ $\mathbf{E} = (E_x, E_y, E_z)$ $\mathbf{R} = \frac{2\mathbf{U}(\mathbf{U}\bullet\mathbf{E})}{(\mathbf{U}\bullet\mathbf{U})} - \mathbf{E}$	$\mathbf{R}=(R_x, R_y, R_z)$	Cubemap RGBA	→ $R_3G_3B_3A_3$

7-99

DP Diffuse/Reflect Cube Map

- Texture output for second-to-last stage is transformed normal lookup (i.e. diffuse)
- Texture output for last stage is reflection vector lookup (i.e. specular)
- Cube map targets for these stages may hold identity (or normalization) cube maps allowing further computation with register combiners

7-100

DP Diffuse/Reflect Cube Map

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	App specific →	Texture specific	Texture specific	Any type Signed RGB[A]	→ $R_0G_0B_0$
1	$([T_x, B_x, N_x], E_x)$ →	$U_x = [T_x, B_x, N_x] \bullet [R_0, G_0, B_0]$	None	None	→ (0,0,0,0)
2	$([T_y, B_y, N_y], E_y)$ →	$U_y = [T_y, B_y, N_y] \bullet [R_0, G_0, B_0]$	$\mathbf{U} = (U_x, U_y, U_z)$	Cubemap RGBA	→ $R_2G_2B_2A_2$
3	$([T_z, B_z, N_z], E_z)$ →	$U_z = [T_z, B_z, N_z] \bullet [R_0, G_0, B_0]$ $\mathbf{U} = (U_x, U_y, U_z)$ $\mathbf{E} = (E_x, E_y, E_z)$ $\mathbf{R} = \frac{2\mathbf{U}(\mathbf{U}\bullet\mathbf{E})}{(\mathbf{U}\bullet\mathbf{U})} - \mathbf{E}$	$\mathbf{R} = (R_x, R_y, R_z)$	Cubemap RGBA	→ $R_3G_3B_3A_3$

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Dot Product Depth Replace

- Used for "depth sprites", where a screen aligned image can also have correct depth
- Previous stage must be Dot Product program
- Best precision if source texture is unsigned HILO, though other formats may be used
- Calculates two dot products and replaces fragment depth with their quotient
- New depth value clipped to near/far planes
- Output color is (0,0,0,0)

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Dot Product Depth Replace

TEX #	Texture Coords (S,T,R,Q)	Shader Operations	Texture Fetch	Bound Texture Target/Format	Output Color
0	(S_0, T_0, R_0, Q_0) →	Texture 2D	$(\frac{S_0}{Q_0}, \frac{T_0}{Q_0})$ →	2D Unsigned HILO	→ (0,0,0,0)
1	$(Z_{scale}, \frac{Z_{scale}}{2^{16}}, Z_{bias})$ →	$Z = (Z_{scale}, \frac{Z_{scale}}{2^{16}}, Z_{bias}) \bullet [H, L, 1]$	None	None	→ (0,0,0,0)
2	$(W_{scale}, \frac{W_{scale}}{2^{16}}, W_{bias})$ →	$W = (W_{scale}, \frac{W_{scale}}{2^{16}}, W_{bias}) \bullet [H, L, 1]$	None	None	→ (0,0,0,0)

$Z_{window} = \frac{Z}{W}$ → Replaces current fragment's depth

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Texture Shaders
Reflective Bump Map Demo

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Reflection Mapping

Normals are transformed into eye-space

Eye vector is calculated as negated eye-space vertex position

Reflection vector is calculated in eye-space

Reflection vector is transformed into cubemap- space with the texture matrix

- Since the cubemap represents the environment, cubemap-space is typically the same as world-space
- OpenGL does not have an explicit world-space, but the application usually does

Basic shader configuration

The normal map can be HILO or RGB

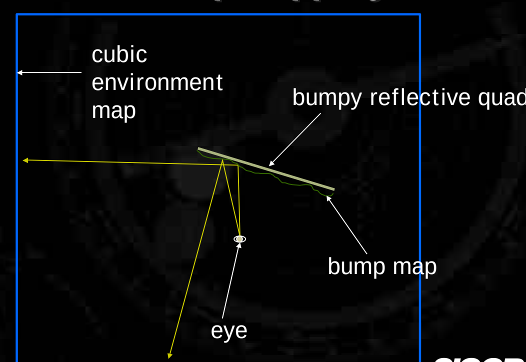
- -stage0: TEXTURE_2D
  - texture image is normal map
- -stage1: DOT_PRODUCT
  - no texture image
- -stage2: DOT_PRODUCT
  - no texture image
- -stage3: DOT_PRODUCT_REFLECT_CUBE_MAP
  - texture image is cubic environment map

Object-Space Reflective Bump Map

Demo renders a single bumpy, reflective quad

- Normal map defined in *object-space*
- Cubic environment map space is same as eye-space in this example
- Reflection vector is calculated per-pixel

Reflective Bump Mapping



Rendering

The normal vector and eye vector must be transformed into cubemap-space (which is the same as eye space in this example)

- Normal vector is multiplied by the upper 3x3 of the inverse transpose of the MODELVIEW matrix, the same as object-space per-vertex normals are treated for per-vertex lighting in OpenGL
- The eye vector is calculated per-vertex, and because the eye is defined to be at (0,0,0) in eye-space, it is simply the negative of the eye-space vertex position

7-109

Rendering (2)

Given the normal vector (n') and the eye vector (e) both defined in cubemap-space, the reflection vector (r) is calculated as

$$\mathbf{r} = \frac{2\mathbf{n}'(\mathbf{n}' \bullet \mathbf{e})}{(\mathbf{n}' \bullet \mathbf{n}')} - \mathbf{e}$$

- **The reflection vector is used to look into a cubic environment map**
- **This is the same as per-vertex cubic environment mapping except that the reflection calculation *must* happen in cubemap-space**

7-110

Details

The per-vertex data is passed in as texture coords of texture shader stages 1, 2, 3

- The upper-left 3x3 of the inverse transpose modelview matrix (M^{-t}) is passed in s, t, r coordinates
  - note: $M^{-t} \equiv M$ for rotation-only matrices
- The (unnormalized) eye vector (e_x, e_y, e_z) is specified per-vertex in the q coordinate.

$$(s_1, t_1, r_1, q_1) = (M^{-t}_{00}, M^{-t}_{01}, M^{-t}_{02}, e_x)$$
$$(s_2, t_2, r_2, q_2) = (M^{-t}_{10}, M^{-t}_{11}, M^{-t}_{12}, e_y)$$
$$(s_3, t_3, r_3, q_3) = (M^{-t}_{20}, M^{-t}_{21}, M^{-t}_{22}, e_z)$$

7-111

True Reflective Bump Mapping

Unlike DX6 EMBM technique, this method performs 3D vector calculations per-pixel!

- Transform of the normal map normal (n) by the texel matrix (T) to yield (n')
- Evaluation of reflection equation using n' and e

The resulting 3D reflection vector is looked up in a cubic environment map

This IS true reflective bump mapping

7-112

Results

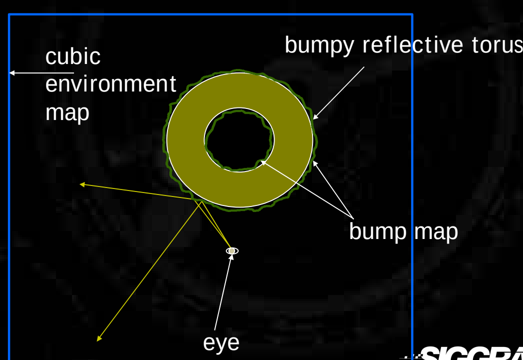
A screen shot from the running demo

Tangent-Space Reflective Bump Map

Demo renders a bumpy, reflective torus

- Normal map defined in *tangent-space*
- Cubemap-space is same as eye-space
- Reflection vector is calculated per-pixel

Reflective Bump Mapping



Rendering

The texture coordinates are the same in this example as in previous demo, with the notable exception that the surface-local transform (S) must also be applied to the normals in the normal map

- Normal vector is multiplied by the product of the upper 3x3 of the inverse transpose of the MODELVIEW matrix (M^{-t}) and the matrix (S) whose columns are the tangent, binormal, and normal surface-local basis vectors

Rendering (2)

The texel matrix (T) is defined as the product of the upper-left 3x3 of the inverse transpose of the modelview matrix (M^{-t}) and the surface-local-space to object-space matrix (S)

Details

The texel matrix (T) and eye vector (e) are specified in the texture coordinates of stages 1, 2, and 3

$(s_1, t_1, r_1, q_1) = (T_{00}, T_{01}, T_{02}, e_x)$

$(s_2, t_2, r_2, q_2) = (T_{10}, T_{11}, T_{12}, e_y)$

$(s_3, t_3, r_3, q_3) = (T_{20}, T_{21}, T_{22}, e_z)$

Results

A screen shot from the running demo

For More Information

- **NVIDIA OpenGL Extension Specification**
- **NVIDIA OpenGL SDK**
- **Available at NVIDIA Developer Website:**
  http://www.nvidia.com/developer
- **Microsoft DX8 Documentation:**
  http://www.microsoft.com/directx

Vertex Programs: Appendix A

The NV_vertex_program Instruction Set

(DX8 is very similar)

The Instruction Set

MOV: Move

Function:

Moves the value of the source vector into the destination register.

Syntax:

```
MOV   dest, src0;
```


The Instruction Set

ARL: Address Register Load

Function:

Loads the floor(s) into the address register for some scalar s. The address register is a signed integer scalar.

Syntax:

```
ARL   A0.x, src0.C
```

where ‘C’ is x, y, z, or w

The Instruction Set

MUL: Multiply

Function:

Performs a component-wise multiply on two vectors.

Syntax:

```
MUL   dest, src0, src1;
```

The Instruction Set

ADD: Add

Function:

Performs a component-wise addition on two vectors.

Syntax:

```
ADD   dest, src0, src1;
```

7-125

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Instruction Set

MAD: Multiply and Add

Function:

Adds the value of the third source vector to the product of the values of the first and second source vectors.

Syntax:

```
MAD   dest, src0, src1, src2;
```

7-126

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Instruction Set

RCP: Reciprocal

Function:

Inverts the value of the source scalar and replicates the result across the destination register.

Syntax:

```
RCP   dest, src0.C;
```

where ‘C’ is x, y, z, or w

7-127

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Instruction Set

RSQ: Reciprocal Square Root

Function:

Computes the inverse square root of the absolute value of the source scalar and replicates the result across the destination register.

Syntax:

```
RSQ   dest, src0.C;
```

where ‘C’ is x, y, z, or w

7-128

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Instruction Set

DP3: Three-Component Dot Product

Function:

Computes the three-component (x,y,z) dot product of two source vectors and replicates the result across the destination register.

Syntax:

```
DP3    dest, src0, src1;
```

7-129

The Instruction Set

DP4: Four-Component Dot Product

Function:

Computes the four-component dot product (x,y,z,w) of two source vectors and replicates the result across the destination register.

Syntax:

```
DP4    dest, src0, src1;
```

7-130

The Instruction Set

MIN: Minimum

Function:

Computes a component-wise minimum on two vectors.

Syntax:

```
MIN    dest, src0, src1;
```

7-131

The Instruction Set

MAX: Maximum

Function:

Computes a component-wise maximum on two vectors.

Syntax:

```
MAX   dest, src0, src1;
```

7-132

The Instruction Set

SLT: Set On Less Than

Function:

Performs a component-wise assignment of either 1.0 or 0.0. 1.0 is assigned if the value of the first source is less than the value of the second. Otherwise, 0.0 is assigned.

Syntax:

SLT dest, src0, src1;

7-133

The Instruction Set

SGE: Set On Greater Than or Equal Than

Function:

Performs a component-wise assignment of either 1.0 or 0.0. 1.0 is assigned if the value of the first source is greater than or equal the value of the second. Otherwise, 0.0 is assigned.

Syntax:

SGE dest, src0, src1;

7-134

The Instruction Set

EXP: Exponential Base 2

Function:

Generates an approximation of 2^P for some scalar P. (accurate to 11 bits)
(Also generates intermediate terms used to compute more accurate result using extra instructions.)

Syntax:

EXP dest, src0.C

where 'C' is x, y, z, or w

7-135

The Instruction Set

EXP: Exponential Base 2

Result:

z contains the 2^P result
x and y contain intermediate results
w set to 1

dest.x = $2^{floor(src0.C)}$
dest.y = src0.C – floor(src0.C)
dest.z ~= $2^{(src0.C)}$
dest.w = 1

7-136

The Instruction Set

LOG: Logarithm Base 2

Function:

Generates an approximation of log_2(|s|) for some scalar s. (accurate to 11 bits)
(Also generates intermediate terms used to compute more accurate result using additional instructions.)

Syntax:

LOG   dest, src0.C

where 'C' is x, y, z, or w

7-137

The Instruction Set

LOG: Logarithm Base 2

Result:

z contains the log_2(|s|) result
x and y just contain intermediate results
w set to 1

dest.x = Exponent(src0.C)   in range [-126.0, 127.0]
dest.y = Mantissa(src0.C)   in range [1.0, 2.0)
dest.z ~= log_2(|src0.C|)
dest.w = 1

7-138

The Instruction Set

EXP and LOG – Increasing the precision

- EXP approximated by:

  EXP(s) = $2^{floor(s)}$ ´ APPX(s-floor(s))
  where APPX is an approximation of 2^t for t in [0.0, 1.0)

- LOG approximated by:

  LOG(|s|) = Exponent(s) + APPX(Mantissa(s))
  where APPX is an approximation of log2(t) for t in [1.0, 2.0)

7-139

The Instruction Set

LIT: Light Coefficients

Function:

Computes ambient, diffuse, and specular lighting coefficients from a diffuse dot product, a specular dot product, and a specular power.

Assumes:

src0.x = diffuse dot product   (N • L)
src0.y = specular dot product   (N • H)
src0.w = power   (m)

7-140

The Instruction Set

LIT: Light Coefficients

Syntax:

LIT dest, src0

Result:

dest.x = 1.0 (ambient coeff.)

dest.y = CLAMP(src0.x, 0, 1)
= CLAMP(N • L, 0, 1) (diffuse coeff.)

dest.z = (see next slide…) (specular coeff.)

dest.w = 1.0

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

The Instruction Set

LIT: Light Coefficients

Result: (Recall: src0.x ° N • L, src0.y ° N • H, src0.w ° m)

if (src0.x > 0.0)
dest.z = $(MAX(src0.y,0))^{(ECLAMP(src0.w,-128,128))}$
= $(MAX(N • H,0))^{m}$ where m in (-128,128)

otherwise,
dest.z = 0.0

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Vertex Programs: Appendix B

OpenGL Fixed Function Emulation

(DX8 is very similar)

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Fixed Function Emulation

Master register mapping
Master template
Master constant map

Extract code of interest
Compress (no re-arrangement required)
Optional performance tuning
Load and run

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Registers

- Most common data stored at top of registers
- Feel free to reorganize as you see fit
- You have 12 registers

7-145

SIGGRAPH 2001

Master register mapping

R0	scratch
R1	scratch
R2	scratch
R3 (Rd)	scratch/light vector
R4 (Rr,Rf,Rl)	scratch
R5 (Rs,Rv)	scratch/sphere vector/eye vector
R6 (Rx)	scratch/specular color
R7 (Rc)	scratch/diffuse color
R8 (RH)	scratch/half angle vector
R9 (Rh)	eye position homogeneous
R10(Re)	eye position non-homogeneous
R11(Rn)	eye normal

7-146

SIGGRAPH 2001

Program

- Code is organized into four main blocks
- 1. Transform/skinning
- 2. Fog/Point parameters
- 3. Lighting
- 4. Texture
- Replicate blocks for more lights or textures

7-147

SIGGRAPH 2001

Modelview0 Transform

```
/* eye space normal */
DP3 Rn.x,v[NRML],c[NORMAL0_MATRIX_X];
DP3 Rn.y,v[NRML],c[NORMAL0_MATRIX_Y];
DP3 Rn.z,v[NRML],c[NORMAL0_MATRIX_Z];

/* eye space position homogeneous */
DP4 Rh.x,v[OPOS],c[MODELVIEW0_MATRIX_X];
DP4 Rh.y,v[OPOS],c[MODELVIEW0_MATRIX_Y];
DP4 Rh.z,v[OPOS],c[MODELVIEW0_MATRIX_Z];
DP4 Rh.w,v[OPOS],c[MODELVIEW0_MATRIX_W];
```

7-148

SIGGRAPH 2001

Modelview1 Transform (skin)

```
/* eye space normal */
DP3 R2.x,v[NRML],c[NORMAL1_MATRIX_X];
DP3 R2.y,v[NRML],c[NORMAL1_MATRIX_Y];
DP3 R2.z,v[NRML],c[NORMAL1_MATRIX_Z];

/* eye space position homogeneous */
DP4 R3.x,v[OPOS],c[MODELVIEW1_MATRIX_X];
DP4 R3.y,v[OPOS],c[MODELVIEW1_MATRIX_Y];
DP4 R3.z,v[OPOS],c[MODELVIEW1_MATRIX_Z];
DP4 R3.w,v[OPOS],c[MODELVIEW1_MATRIX_W];
```

7-149

Modelview2 Transform (skin)

```
/* eye space normal */
DP3 R4.x,v[NRML],c[NORMAL2_MATRIX_X];
DP3 R4.y,v[NRML],c[NORMAL2_MATRIX_Y];
DP3 R4.z,v[NRML],c[NORMAL2_MATRIX_Z];

/* eye space position homogeneous */
DP4 R5.x,v[OPOS],c[MODELVIEW2_MATRIX_X];
DP4 R5.y,v[OPOS],c[MODELVIEW2_MATRIX_Y];
DP4 R5.z,v[OPOS],c[MODELVIEW2_MATRIX_Z];
DP4 R5.w,v[OPOS],c[MODELVIEW2_MATRIX_W];
```

7-150

Modelview3 Transform (skin)

```
/* eye space normal */
DP3 R6.x,v[NRML],c[NORMAL3_MATRIX_X];
DP3 R6.y,v[NRML],c[NORMAL3_MATRIX_Y];
DP3 R6.z,v[NRML],c[NORMAL3_MATRIX_Z];

/* eye space position homogeneous */
DP4 R7.x,v[OPOS],c[MODELVIEW3_MATRIX_X];
DP4 R7.y,v[OPOS],c[MODELVIEW3_MATRIX_Y];
DP4 R7.z,v[OPOS],c[MODELVIEW3_MATRIX_Z];
DP4 R7.w,v[OPOS],c[MODELVIEW3_MATRIX_W];
```

7-151

Skinning: Blend Weights

```
MOV R0,v[WGHT];
MOV R0.w,c[CONSTANT0].z;          /* if need new */
DP4 R0.y,R0,c[CONSTANT0].xyyz;    /* new 2nd */
DP4 R0.z,R0,c[CONSTANT0].xxyz;    /* or new 3rd */
DP4 R0.w,R0,c[CONSTANT0].xxxz;    /* or new 4th */

MUL Rh,R0.x,Rh;      /* position */
MUL Rn,R0.x,Rn;      /* normal */
MAD Rh,R0.y,R3,Rh;   /* if 2+ matrix skin */
MAD Rn,R0.y,R2,Rn;   /* if 2+ matrix skin */
MAD Rh,R0.z,R5,Rh;   /* if 3+ matrix skin */
MAD Rn,R0.z,R4,Rn;   /* if 3+ matrix skin */
MAD Rh,R0.w,R7,Rh;   /* if 4  matrix skin */
MAD Rn,R0.w,R6,Rn;   /* if 4  matrix skin */
```

7-152

Position Output

```
/* skinning output */
DP4 o[HPOS].x,Rh,c[PROJECTION_MATRIX_X];
DP4 o[HPOS].y,Rh,c[PROJECTION_MATRIX_Y];
DP4 o[HPOS].z,Rh,c[PROJECTION_MATRIX_Z];
DP4 o[HPOS].w,Rh,c[PROJECTION_MATRIX_W];

/* non-skinning output */
DP4 o[HPOS].x,v[OPOS],c[COMPOSITE_MATRIX_X];
DP4 o[HPOS].y,v[OPOS],c[COMPOSITE_MATRIX_Y];
DP4 o[HPOS].z,v[OPOS],c[COMPOSITE_MATRIX_Z];
DP4 o[HPOS].w,v[OPOS],c[COMPOSITE_MATRIX_W];
```


Normalize Eye Normal

```
DP3 Rn.w,Rn,Rn;
RSQ Rn.w,Rn.w;
MUL Rn,Rn,Rn.w;
```


Vertex Eye Space Position

```
/* Vertex position in eye space */
RCP R0.w,Rh.w;
MUL Re,Rh,R0.w;

/* Eye vector */
ADD R0,-Re,c[EYE_POSITION];
DP3 R0.w,R0,R0;
RSQ R1.w,R0.w;
MUL Rv,R0,R1.w;        /* direction vector */
DST Rf,R0.w,R1.w;      /* distance vector */
```


Fog/Point Parameters

```
/* radial fog */
MOV o[FOGC].x,Rf.y;

/* z linear fog */
MOV o[FOGC].x,-Re.z;

/* Point parameters */
DP3 R0.w,Rf,c[POINT_PARAM_ATTENUATION];
RSQ R0.w,R0.w;
MUL R0.w,R0.w,c[POINT_PARAM].x;
MIN R0.w,R0.w,c[POINT_PARAM].y;
MAX o[PSIZ].x,R0.w,c[POINT_PARAM].z;
```

Lighting Initialization

```
/* diffuse only */
MOV Rc,c[GLOBAL_ILLUMINATION];

/* diffuse and specular */
MOV Rc,c[GLOBAL_ILLUMINATION];
MOV Rx,c[CONSTANT0].y;
```


Infinite Light/Infinite Viewer

```
DP3 R0.x,Rn,c[LIGHT_POSITION];
DP3 R0.y,Rn,c[LIGHT_HALF_ANGLE_VECTOR];
MOV R0.w,c[LIGHT_SPECULAR].w;
LIT R0,R0;
MAD Rc.xyz,R0.x,c[LIGHT_AMBIENT],Rc;
MAD Rc.xyz,R0.y,c[LIGHT_DIFFUSE],Rc;
MAD Rx.xyz,R0.z,c[LIGHT_SPECULAR],Rx;
/* use Rc above for single color */
```


Spotlight/Local Viewer 1/3

```
/* light direction/distance vectors */
ADD R0,-Re,c[LIGHT_POSITION];
DP3 R0.w,R0,R0;
RSQ R1.w,R0.w;
MUL Rl,R0,R1.w;        /* direction */
DST Rd,R0.w,R1.w;      /* distance */

/* half-angle vector */
ADD RH,Rv,Rl;
DP3 RH.w,RH,RH;
RSQ RH.w,RH.w;
MUL RH,RH,RH.w;
```


Spotlight/Local Viewer 2/3

```
/* distance attenuation */
DP3 Rd.w,Rd,c[LIGHT_ATTENUATION];
RCP Rd.w,Rd.w;

/* spotlight cone attenuation */
DP3 R0.y,Rl,-c[LIGHT_SPOT_DIRECTION];
ADD R0.x,R0.y,-c[LIGHT_SPOT_DIRECTION].w;
MOV R0.w,c[LIGHT_ATTENUATION].w;
LIT R0,R0;
MUL Rd,Rd.w,R0.z;
```

Spotlight/Local Viewer 3/3

```
DP3 R0.x,Rn,Rl;
DP3 R0.y,Rn,RH;
MOV R0.w,c[LIGHT_SPECULAR].w;
LIT R0,R0;
MUL R0,R0,Rd.w;
MAD Rc.xyz,R0.x,c[LIGHT_AMBIENT],Rc;
MAD Rc.xyz,R0.y,c[LIGHT_DIFFUSE],Rc;
MAD Rx.xyz,R0.z,c[LIGHT_SPECULAR],Rx;
/* use Rc above for single color */
```

7-161

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Lighting Output

```
/* diffuse only */
MOV o[COL0],Rc;

/* diffuse and specular */
MOV o[COL0],Rc;
MOV o[COL1],Rx;
```

7-162

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Global Texture Generation State

```
/* reflection vector */
DP3 Rr.w,Rn,Rv;
MUL R0,Rn,c[EYE_POSITION].w;
MAD Rr,Rr.w,R0,-Rv;

/* sphere map vector */
ADD R0,c[CONSTANT0].yyzy,Rr;
DP3 R0.w,R0,R0;
RSQ R0.w,R0.w;
MUL R0.xyz,R0,c[CONSTANT0].wwyy;
MAD Rs,R0.w,R0,c[CONSTANT0].wwyy;
```

7-163

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Per Texture 1/2

```
MOV R0,v[TEX0];                          /* initialize R0 */

DP4 R0.x,v[OPOS],c[TEXTURE_OBJECT_PLANE_X]; /*obj */
DP4 R0.y,v[OPOS],c[TEXTURE_OBJECT_PLANE_Y];
DP4 R0.z,v[OPOS],c[TEXTURE_OBJECT_PLANE_Z];
DP4 R0.w,v[OPOS],c[TEXTURE_OBJECT_PLANE_W];

DP4 R0.x,Rh,c[TEXTURE_EYE_PLANE_X];   /* eye space */
DP4 R0.y,Rh,c[TEXTURE_EYE_PLANE_Y];
DP4 R0.z,Rh,c[TEXTURE_EYE_PLANE_Z];
DP4 R0.w,Rh,c[TEXTURE_EYE_PLANE_W];
```

7-164

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Per Texture 2/2

```
MOV R0.xy,Rs;          /* sphere map */

MOV R0.xyz,Rn;         /* normal vector */

MOV R0.xyz,Rr;         /* reflection vector */

/* texture matrix and output */
DP4 o[TEX0].x,R0,c[TEXTURE_MATRIX_X];
DP4 o[TEX0].y,R0,c[TEXTURE_MATRIX_Y];
DP4 o[TEX0].z,R0,c[TEXTURE_MATRIX_Z];
DP4 o[TEX0].w,R0,c[TEXTURE_MATRIX_W];
```

7-165

Constants

- **Constants required by the template code is organized into blocks by functionality.**
- **Feel free to reorganize as you see fit**
- **You have 96 locations**
- **Use remaining space for more lights, more textures, more matrices, more ???**

7-166

Constant Map 0-11

```
/* inverse transpose modelview matrices */
c[NORMAL0_MATRIX_X]
c[NORMAL0_MATRIX_Y]
c[NORMAL0_MATRIX_Z]
c[NORMAL1_MATRIX_X]
c[NORMAL1_MATRIX_Y]
c[NORMAL1_MATRIX_Z]
c[NORMAL2_MATRIX_X]
c[NORMAL2_MATRIX_Y]
c[NORMAL2_MATRIX_Z]
c[NORMAL3_MATRIX_X]
c[NORMAL3_MATRIX_Y]
c[NORMAL3_MATRIX_Z]
```

7-167

Constant Map 12-27

```
/* modelview matrices */
c[MODELVIEW0_MATRIX_X]
c[MODELVIEW0_MATRIX_Y]
c[MODELVIEW0_MATRIX_Z]
c[MODELVIEW0_MATRIX_W]
c[MODELVIEW1_MATRIX_X]
c[MODELVIEW1_MATRIX_Y]
c[MODELVIEW1_MATRIX_Z]
c[MODELVIEW1_MATRIX_W]
c[MODELVIEW2_MATRIX_X]
c[MODELVIEW2_MATRIX_Y]
c[MODELVIEW2_MATRIX_Z]
c[MODELVIEW2_MATRIX_W]
c[MODELVIEW3_MATRIX_X]
c[MODELVIEW3_MATRIX_Y]
c[MODELVIEW3_MATRIX_Z]
c[MODELVIEW3_MATRIX_W]
```

7-168

Constant Map 28-35

```
/* Projection Matrix */
c[PROJECTION_MATRIX_X]
c[PROJECTION_MATRIX_Y]
c[PROJECTION_MATRIX_Z]
c[PROJECTION_MATRIX_W]

/* Projection*Modelview Matrix */
c[COMPOSITE_MATRIX_X]
c[COMPOSITE_MATRIX_Y]
c[COMPOSITE_MATRIX_Z]
c[COMPOSITE_MATRIX_W]
```


Constant Map 36-43

```
/* texture matrix */
c[TEXTURE_MATRIX_X]
c[TEXTURE_MATRIX_Y]
c[TEXTURE_MATRIX_Z]
c[TEXTURE_MATRIX_W]

/* texgen planes (object or eye) */
c[TEXTURE_PLANE_X]
c[TEXTURE_PLANE_Y]
c[TEXTURE_PLANE_Z]
c[TEXTURE_PLANE_W]
```


Constant Map 44-51

```
/* light state */
c[GLOBAL_ILLUMINATION]        /* R,G,B,A (emission+global ambient) */
c[LIGHT_POSITION]             /* X,Y,Z,NA */
c[LIGHT_HALF_ANGLE_VECTOR]    /* X,Y,Z,NA for infinite light+viewer */
c[LIGHT_AMBIENT]              /* R,G,B,NA (light*mat) */
c[LIGHT_DIFFUSE]              /* R,G,B,NA (light*mat) */
c[LIGHT_SPECULAR]             /* R,G,B,SPECULAR POWER (light*mat) */
c[LIGHT_ATTENUATION]          /* K0,K1,K2,SPOT POWER */
c[LIGHT_SPOT_DIRECTION]       /* X,Y,Z,cos(CUTOFF) */
```


Constant Map 52-55

```
/* point parameters */
c[POINT_PARAM]                /* PSIZE,MAX,MIN,NA */
c[POINT_PARAM_ATTENUATION]    /* K0,K1,K2,NA */

/* miscellaneous */
c[EYE_POSITION]               /*  0.0,0.0,0.0,2.0 */
c[CONSTANT0]                  /* -1.0,0.0,1.0,0.5 */
```