Vertex Programs

Chris Wynn
Overview

- What is Vertex Programming?
- Program Specification and Parameters
- Vertex Program Register Set
- Vertex Programming Assembly Language
  - Instruction Set
  - Mini-Examples
- Example Programs
- Performance
- Summary
What is Vertex Programming?

- Traditional Graphics Pipeline

Each unit has specific function (possibly with “modes” of operation)
What is Vertex Programming?

- Vertex Programming offers programmable T&L unit

  transform & lighting
  setup rasterizer
  texture blending
  frame-buffer anti-aliasing

  User-defined Vertex Processing

Gives the programmer total control of vertex processing.
What is Vertex Programming?

- Complete control of transform and lighting HW
- Complex vertex operations accelerated in HW
- Custom vertex lighting
- Custom skinning and blending
- Custom texture coordinate generation
- Custom texture matrix operations
- Custom vertex computations of your choice

- Offloading vertex computations frees up CPU
  - More physics, simulation, and AI possible.
What is Vertex Programming?

- Custom transform, lighting, and skinning
What is Vertex Programming?

- Custom cartoon-style lighting
What is Vertex Programming?

- Per-vertex set up for per-pixel bump mapping
What is Vertex Programming?

- Character morphing & shadow volume projection
What is Vertex Programming?

- Dynamic displacements of surfaces by objects
Demo

- Matrix-Palette Skinning...
What is Vertex Programming?

• Vertex Program
  • Assembly language interface to T&L unit
  • GPU instruction set to perform all vertex math
  • Reads an untransformed, unlit vertex
  • Creates a transformed vertex
  • Optionally …
    • Lights a vertex
    • Creates texture coordinates
    • Creates fog coordinates
    • Creates point sizes
What is Vertex Programming?

- **Vertex Program**
  - Does not create or delete vertices
    - 1 vertex in and 1 vertex out
  - No topological information provided
    - No edge, face, nor neighboring vertex info
  - Dynamically loadable

- Exposed through NV_vertex_program extension
What is Vertex Programming?

```
glEnable( GL_VERTEX_PROGRAM_NV );
```

Bypass the fixed function T&L path…
and do your own thing in HW.
Vertex Programming
Conceptual Overview

Vertex Attributes

Vertex Program

Vertex Output
Vertex Programming
Conceptual Overview

- **Vertex Attributes**: 16x4 registers
  - Position, colors, normal
  - User-defined vertex parameters: densities, velocities, weights, etc.

- **Vertex Program**

- **Vertex Output**

Sixteen 4-component vector floating point registers
Position, colors, normal
User-defined vertex parameters: densities, velocities, weights, etc.
Vertex Programming
Conceptual Overview

Vertex Attributes
16x4 registers

Vertex Program
128 instructions

Up to 128 program instructions (SIMD) (i.e. add, multiply, etc.)
Read vertex attribute registers
Write vertex output registers

Vertex Output
Vertex Programming
Conceptual Overview

Vertex Attributes
16x4 registers

Program Parameters
96x4 registers

Temporary Registers
12x4 registers

Modifiable only outside of glBegin/glEnd pair
Read-only
Read/Write-able
Vertex Programming
Conceptual Overview

Vertex Attributes
16x4 registers

Vertex Program
128 instructions

Program Parameters
96x4 registers

Temporary Registers
12x4 registers

Vertex Output
15x4 registers

Fifteen 4-component floating vectors
Homogeneous clip space position
Primary, secondary colors
Fog coord, point size, texture coords.
Vertex Program Specification and Invocation

- Programs are arrays of GLubyteS ("strings")
- 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 )
- Invoked when glVertex issued
Vertex Programming
Parameter Specification

- Two types
  - Per-Vertex
  - Per-Begin/End block

- Vertex Attributes
- Program Parameters
Vertex Programming
Per-Vertex Parameters

- Up to 16x4 per-vertex attributes
- Values specified with new commands
  - `glVertexAttrib4fNV(index, ...)`
  - `glVertexAttribPointerNV(index, ...)`
- Attributes also specified through conventional per-vertex parameters via aliasing
- Values correspond to 16x4 readable vertex attribute registers
**Vertex Programming**

**Vertex Attributes**

<table>
<thead>
<tr>
<th>Attribute Register</th>
<th>Conventional per-vertex Attribute</th>
<th>Conventional Command</th>
<th>Conventional Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>vertex position</td>
<td>glVertex</td>
<td>x, y, z, w</td>
</tr>
<tr>
<td>1</td>
<td>vertex weights</td>
<td>glVertexWeightEXT</td>
<td>w, 0, 0, 1</td>
</tr>
<tr>
<td>2</td>
<td>normal</td>
<td>glNormal</td>
<td>x, y, z, 1</td>
</tr>
<tr>
<td>3</td>
<td>Primary color</td>
<td>glColor</td>
<td>r, g, b, a</td>
</tr>
<tr>
<td>4</td>
<td>secondary color</td>
<td>glSecondaryColorEXT</td>
<td>r, g, b, 1</td>
</tr>
<tr>
<td>5</td>
<td>Fog coordinate</td>
<td>glFogCoordEXT</td>
<td>fc, 0, 0, 1</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>Texture coord 0</td>
<td>glMultiTexCoord</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>9</td>
<td>Texture coord 1</td>
<td>glMultiTexCoord</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>10</td>
<td>Texture coord 2</td>
<td>glMultiTexCoord</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>11</td>
<td>Texture coord 3</td>
<td>glMultiTexCoord</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>12</td>
<td>Texture coord 4</td>
<td>glMultiTexCoord</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>13</td>
<td>Texture coord 5</td>
<td>glMultiTexCoord</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>14</td>
<td>Texture coord 6</td>
<td>glMultiTexCoord</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>15</td>
<td>Texture coord 7</td>
<td>glMultiTexCoord</td>
<td>s, t, r, q</td>
</tr>
</tbody>
</table>

Semantics defined by program NOT parameter name!
Vertex Programming
Program Parameters

- Up to 96x4 per-block parameters
- Store parameters such as matrices, lighting params, and constants required by vertex programs.

- Values specified with new commands
  - glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, index, x, y, z, w )
  - glProgramParameter4fvNV( GL_VERTEX_PROGRAM_NV, index, n, params )

- Correspond to 96 registers (c[0], ... , c[95])
Vertex Programming
Program Parameters

- Matrices can be “tracked”.
  - Makes matrices automatically available in vertex program’s parameter registers

- MODELVIEW, PERSPECTIVE, TEXTUREi, and others can each be mapped to 4 program parameter registers

- Mapping can be IDENTITY, TRANSPOSE, INVERSE, or INVERSE_TRANSPOSE
26

Vertex Programming
Program Parameters

• Matrix “Tracking”

```c
glTrackMatrixNV( GL_VERTEX_PROGRAM_NV, 4,
                 GL_MODELVIEW, GL.IDENTITY_NV );

glTrackMatrixNV( GL_VERTEX_PROGRAM_NV, 20,
                 GL_MODELVIEW, GL.INVERSE_NV );
```

c[4], c[5], c[6], c[7] correspond to the modelview
c[20], c[21], c[22], c[23] correspond to inverse modelview

Eliminates the need to compute inverses and transposes.
Vertex Programming

Program Parameters

- Values also modifiable by “Vertex State Programs”

- Vertex State Programs are a special kind of vertex program
  - NOT invoked by glVertex
  - Explicitly executed, only outside of a glBegin/glEnd pair.
  - Used to modify program parameters.
  - Uses same instructions/register set but can read AND write c[0], ..., c[95].
Vertex Programming
Program Parameters

- All parameters specified through the API appear as registers to the vertex program

- Read/Write privileges depend on the type of program
  - Vertex State Programs have different read/write access than regular Vertex Programs

- A quick look at the register set...
## The Register Set

<table>
<thead>
<tr>
<th>Vertex Attribute Registers</th>
<th>Program Parameter Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>v[0] v[1] ... v[15]</td>
<td>c[0] c[1] ... c[95]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex Program</th>
<th>Temporary Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R0  R1  ...  R10  R11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex Result Registers</th>
<th>Address Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>o[HPOS] o[COL0] ...</td>
<td>A0.x</td>
</tr>
</tbody>
</table>
The Register Set: Vertex Attribute Registers

<table>
<thead>
<tr>
<th>Attribute Register</th>
<th>Mnemonic Name</th>
<th>Typical Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>v[0]</td>
<td>v[OPOS]</td>
<td>object position</td>
</tr>
<tr>
<td>v[1]</td>
<td>v[WGHT]</td>
<td>vertex weight</td>
</tr>
<tr>
<td>v[3]</td>
<td>v[COL0]</td>
<td>primary color</td>
</tr>
<tr>
<td>v[6]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v[7]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v[8]</td>
<td>v[TEX0]</td>
<td>texture coordinate 0</td>
</tr>
<tr>
<td>v[9]</td>
<td>v[TEX1]</td>
<td>texture coordinate 1</td>
</tr>
</tbody>
</table>

Semantics defined by program NOT parameter name!
### Vertex Programming

#### Vertex Result Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
<th>Component Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>o[HPOS]</td>
<td>Homogeneous clip space position</td>
<td>(x,y,z,w)</td>
</tr>
<tr>
<td>o[COL0]</td>
<td>Primary color (front-facing)</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>o[COL1]</td>
<td>Secondary color (front-facing)</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>o[BFC0]</td>
<td>Back-facing primary color</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>o[BFC1]</td>
<td>Back-facing secondary color</td>
<td>(r,g,b,a)</td>
</tr>
<tr>
<td>o[FOGC]</td>
<td>Fog coordinate</td>
<td>(f,<em>,</em>,*)</td>
</tr>
<tr>
<td>o[PSIZ]</td>
<td>Point size</td>
<td>(p,<em>,</em>,*)</td>
</tr>
<tr>
<td>o[TEX0]</td>
<td>Texture coordinate set 0</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>o[TEX1]</td>
<td>Texture coordinate set 1</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>o[TEX2]</td>
<td>Texture coordinate set 2</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>o[TEX3]</td>
<td>Texture coordinate set 3</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>o[TEX4]</td>
<td>Texture coordinate set 4</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>o[TEX5]</td>
<td>Texture coordinate set 5</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>o[TEX6]</td>
<td>Texture coordinate set 6</td>
<td>(s,t,r,q)</td>
</tr>
<tr>
<td>o[TEX7]</td>
<td>Texture coordinate set 7</td>
<td>(s,t,r,q)</td>
</tr>
</tbody>
</table>

Semantics defined by down-stream pipeline stages.
Vertex Program Register Access

- Vertex Attribute Registers: v[0], v[1], ..., v[15]
- Program Parameter Registers: c[0], c[1], ..., c[95]
- Temporary Registers: R0, R1, ..., R10, R11
- Address Register: A0.x
Vertex State Program
Register Access

- **Vertex Attribute Registers**
  - v[0] v[1] ... v[15]

- **Vertex Program**
  - r (v[0] only)

- **Program Parameter Registers**
  - c[0] c[1] ... c[95]

- **Temporary Registers**
  - R0 R1 ... R10 R11

- **Vertex Result Registers**
  - o[HPOS] o[COL0]...

VSPs used to modify program parameter state.
Demo

- Spline Evaluation...
Vertex Programming
Assembly Language

- Powerful SIMD instruction set
- Four operations simultaneously
- 17 instructions
- Operate on scalar or 4-vector input
- Result in a vector or replicated scalar output
Vertex Programming
Assembly Language

Instruction Format:

 Opcode dst, [-]s0 [,[-]s1 [,[-]s2]];  #comment

<table>
<thead>
<tr>
<th>Instruction name</th>
<th>Destination Register</th>
<th>Source0 Register</th>
<th>Source1 Register</th>
<th>Source2 Register</th>
</tr>
</thead>
</table>

Vertex Programming
Assembly Language

Instruction Format:

Opcode   dst, [-]s0 [,[-]s1 [,[-]s2]];   #comment

<table>
<thead>
<tr>
<th>Instruction name</th>
<th>Destination Register</th>
<th>Source0 Register</th>
<th>Source1 Register</th>
<th>Source2 Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>dst</td>
<td>[-]s0</td>
<td>[,[-]s1</td>
<td>[,[-]s2]</td>
</tr>
</tbody>
</table>

Example:

MOV  r1, r2

R1  R2
x   x
y   y
z   z
w   w
Vertex Programming
Assembly Language

Simple Example:

```
MOV    R1, R2;
```

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>0.0</td>
<td>7.0</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>x</td>
<td>7.0</td>
<td>x</td>
<td>3.0</td>
<td>y</td>
</tr>
<tr>
<td>y</td>
<td>3.0</td>
<td>y</td>
<td>3.0</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>6.0</td>
<td>z</td>
<td>6.0</td>
<td>z</td>
</tr>
<tr>
<td>w</td>
<td>2.0</td>
<td>w</td>
<td>2.0</td>
<td>w</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>after</th>
<th>0.0</th>
<th>7.0</th>
<th>7.0</th>
<th>7.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>7.0</td>
<td>x</td>
<td>3.0</td>
<td>y</td>
</tr>
<tr>
<td>y</td>
<td>3.0</td>
<td>y</td>
<td>3.0</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>6.0</td>
<td>z</td>
<td>6.0</td>
<td>z</td>
</tr>
<tr>
<td>w</td>
<td>2.0</td>
<td>w</td>
<td>2.0</td>
<td>w</td>
</tr>
</tbody>
</table>
Source registers undergo an input mapping before operation occurs…

- Negation
- Swizzling
- Smearing
Vertex Programming
Assembly Language

Source registers can be negated:

```
MOV R1, -R2;
```

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R1</td>
</tr>
<tr>
<td>0.0</td>
<td>x</td>
<td>-7.0</td>
</tr>
<tr>
<td>0.0</td>
<td>y</td>
<td>-3.0</td>
</tr>
<tr>
<td>0.0</td>
<td>z</td>
<td>-6.0</td>
</tr>
<tr>
<td>0.0</td>
<td>w</td>
<td>-2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.0</td>
<td>3.0</td>
<td>6.0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>7.0</td>
<td>3.0</td>
<td>6.0</td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>
Vertex Programming
Assembly Language

Source registers can be “swizzled”:

```
MOV    R1, R2.yzwx;
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>0.0</td>
<td>7.0</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.0</td>
<td>3.0</td>
<td>6.0</td>
<td>3.0</td>
</tr>
<tr>
<td>0.0</td>
<td>6.0</td>
<td>2.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.0</td>
<td>2.0</td>
<td>7.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**before**  

**after**
Source registers can be negated and “swizzled”:

```
MOV R1, -R2.yzzx;
```
Vertex Programming
Assembly Language

Source registers can be swizzled by “smearing”:

```assembly
MOV     R1, R2.w;          # alternative to
# using R2.wwww
```

### before

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.0</td>
<td>3.0</td>
</tr>
<tr>
<td>0.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

### after

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>7.0</td>
</tr>
<tr>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2.0</td>
<td>6.0</td>
</tr>
<tr>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>
Vertex Programming
Assembly Language

Destination register can mask which components are written to...

R1 \[\Rightarrow\] write all components
R1.x \[\Rightarrow\] write only x component
R1.xw \[\Rightarrow\] write only x, w components
### Vertex Programming

**Assembly Language**

**Destination register masking:**

\[ \text{MOV} \quad R1.xw, \ -R2; \]

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0 x</td>
<td>7.0 x</td>
<td>-7.0 x</td>
</tr>
<tr>
<td>0.0 y</td>
<td>3.0 y</td>
<td>0.0 y</td>
</tr>
<tr>
<td>0.0 z</td>
<td>6.0 z</td>
<td>0.0 z</td>
</tr>
<tr>
<td>0.0 w</td>
<td>2.0 w</td>
<td>-2.0 w</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.0 x</td>
<td>3.0 y</td>
<td>6.0 z</td>
</tr>
<tr>
<td>3.0 y</td>
<td>7.0 x</td>
<td>2.0 w</td>
</tr>
</tbody>
</table>
Vertex Programming
Assembly Language

There are 17 instructions in total ...

- ARL
- MOV
- MUL
- ADD
- MAD
- RCP
- RSQ
- DP3
- DP4
- DST
- MIN
- MAX
- SLT
- SGE
- EXP
- LOG
- LIT
The Instruction Set

MOV: Move

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

Syntax:
MOV dest, src0;
The Instruction Set

MUL: Multiply

Function:
Performs a component-wise multiply on two vectors.

Syntax:
MUL dest, src0, src1;
## The Instruction Set

### MUL Example:

```plaintext
MUL R1.xyz, R2, R3;
```

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0.0</td>
<td>14.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>6.3</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>30.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>y</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>z</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>w</td>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R3</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>y</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>z</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>w</td>
<td>7.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>
The Instruction Set

ADD: Add

Function:
Performs a component-wise addition on two vectors.

Syntax:
ADD dest, src0, src1;
The Instruction Set

ADD Example:

```
ADD  R1, R2, -R3;
```

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th></th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R1</td>
</tr>
<tr>
<td>x</td>
<td>0.0</td>
<td>7.0</td>
<td>2.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>3.0</td>
<td>2.1</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>6.0</td>
<td>5.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>2.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>
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;
The Instruction Set

MAD Example:

```
MAD    R1.xyz, R2, R3, R4;
```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>R2</td>
<td>7.0</td>
<td>3.0</td>
<td>6.0</td>
<td>2.0</td>
</tr>
<tr>
<td>R3</td>
<td>2.0</td>
<td>2.1</td>
<td>5.0</td>
<td>7.0</td>
</tr>
<tr>
<td>R4</td>
<td>1.0</td>
<td>3.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>15.0</td>
<td>9.3</td>
<td>32.0</td>
<td>0.0</td>
</tr>
<tr>
<td>R2</td>
<td>7.0</td>
<td>3.0</td>
<td>6.0</td>
<td>7.0</td>
</tr>
<tr>
<td>R3</td>
<td>2.0</td>
<td>2.1</td>
<td>5.0</td>
<td>7.0</td>
</tr>
<tr>
<td>R4</td>
<td>1.0</td>
<td>3.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
The Instruction Set

RCP: Reciprocal

Function:

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

Syntax:

RCP dest, src0.C;

where ‘C’ is x, y, z, or w
### The Instruction Set

**RCP Example:**

\[
\text{RCP } \text{R1, R2}.w;
\]

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>R1</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>R2</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>R1</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>R2</td>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>
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
The Instruction Set

RSQ Example:

```
RSQ R1.x, R5.x;
```

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R5</td>
<td>R1</td>
</tr>
<tr>
<td>x</td>
<td>0.0</td>
<td>-4.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>3.0</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>7.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>9.0</td>
</tr>
</tbody>
</table>
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;
The Instruction Set

DP3 Example:

```
DP3  R1, R6, R6;
```

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R6</td>
<td>R1</td>
</tr>
<tr>
<td>x</td>
<td>0.0</td>
<td>14.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>14.0</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>14.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>14.0</td>
</tr>
</tbody>
</table>
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:
\[
\text{DP4 dest, src0, src1;}
\]
The Instruction Set

DP4 Example:

\[
\text{DP4} \quad R1, \; R6, \; R6;
\]

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.0</td>
<td>15.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>15.0</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>15.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>15.0</td>
</tr>
<tr>
<td>R1</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>R6</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>R1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>R6</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
The Instruction Set

MIN: Minimum

Function:
Computes a component-wise minimum on two vectors.

Syntax:
MIN dest, src0, src1;
**The Instruction Set**

**MIN Example:**

```
MIN   R1, R2, R3;
```

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
</tr>
<tr>
<td>x</td>
<td>0.0</td>
<td>2.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>2.1</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>5.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>

The values in `R1`, `R2`, and `R3` are updated after the `MIN` instruction is executed.
The Instruction Set

MAX: Maximum

Function:
Computes a component-wise maximum on two vectors.

Syntax:
MAX dest, src0, src1;
The Instruction Set

MAX Example:

```
MAX R1, R2, R3;
```

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.0</td>
<td>7.0</td>
<td>2.0</td>
<td></td>
<td>7.0</td>
<td>7.0</td>
<td>2.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>3.0</td>
<td>2.1</td>
<td></td>
<td>3.0</td>
<td>3.0</td>
<td>2.1</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>6.0</td>
<td>5.0</td>
<td></td>
<td>6.0</td>
<td>6.0</td>
<td>5.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>2.0</td>
<td>7.0</td>
<td></td>
<td>7.0</td>
<td>2.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>
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;
The Instruction Set

SLT Example:

```
SLT R1, R2, R3;
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>7.0</td>
<td>3.0</td>
<td>6.0</td>
<td>2.0</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>2.0</td>
<td>2.1</td>
<td>5.0</td>
<td>7.0</td>
<td>R3</td>
<td>R3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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;
**The Instruction Set**

**SGE Example:**

```
SGE    R1, R2, R3;
```

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.0</td>
<td>7.0</td>
<td>2.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>3.0</td>
<td>2.1</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>6.0</td>
<td>5.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>2.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1.0</td>
<td>7.0</td>
<td>2.0</td>
</tr>
<tr>
<td>y</td>
<td>1.0</td>
<td>3.0</td>
<td>2.1</td>
</tr>
<tr>
<td>z</td>
<td>1.0</td>
<td>6.0</td>
<td>5.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>2.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>
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 that can be used to compute a more accurate result using additional instructions.)

Syntax:

`EXP dest, src0.C`

where ‘C’ is x, y, z, or w
The Instruction Set

EXP: Exponential Base 2

Result:

- \( z \) contains the \( 2^p \) result
- \( x \) and \( y \) contain intermediate results
- \( w \) set to 1

\[
\begin{align*}
\text{dest}.x &= 2^{\text{\text{floor}}(\text{src0.C})} \\
\text{dest}.y &= \text{src0.C} - \text{\text{floor}}(\text{src0.C}) \\
\text{dest}.z &= 2^{(\text{src0.C})} \\
\text{dest}.w &= 1
\end{align*}
\]
The Instruction Set

EXP Example:

```
EXP R1, R3.y;
```

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td>0.0</td>
<td>2.0</td>
</tr>
<tr>
<td><strong>y</strong></td>
<td>0.0</td>
<td>2.1</td>
</tr>
<tr>
<td><strong>z</strong></td>
<td>0.0</td>
<td>5.0</td>
</tr>
<tr>
<td><strong>w</strong></td>
<td>0.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td><strong>y</strong></td>
<td>0.1</td>
<td>2.1</td>
</tr>
<tr>
<td><strong>z</strong></td>
<td>4.287..</td>
<td>5.0</td>
</tr>
<tr>
<td><strong>w</strong></td>
<td>1.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>

(Good to 11 bits)
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 that can be used to compute a more accurate result using additional instructions.)

Syntax:
\[
\text{LOG dest, src0.C}
\]
where ‘C’ is x, y, z, or w
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

\[
\begin{align*}
\text{dest}.x &= \text{Exponent}(\text{src0}.C) \quad \text{in range} \ [-126.0, 127.0] \\
\text{dest}.y &= \text{Mantissa}(\text{src0}.C) \quad \text{in range} \ [1.0, 2.0) \\
\text{dest}.z &\sim= \log_2(|\text{src0}.C|) \\
\text{dest}.w &= 1
\end{align*}
\]
The Instruction Set

LOG Example:

\texttt{LOG R1, R3.y;}

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R3</td>
<td>R1</td>
</tr>
<tr>
<td>x</td>
<td>0.0</td>
<td>2.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>2.1</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>5.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>

(Good to 11 bits)
The Instruction Set

EXP and LOG – Increasing the precision

- EXP approximated by:

\[ \text{EXP}(s) = 2^{\text{floor}(s)} \times \text{APPX}(s - \text{floor}(s)) \]

where APPX is an approximation of \(2^t\) for \(t\) in \([0.0, 1.0)\)

- LOG approximated by:

\[ \text{LOG}(|s|) = \text{Exponent}(s) + \text{APPX}(\text{Mantissa}(s)) \]

where APPX is an approximation of \(\log_2(t)\) for \(t\) in \([1.0, 2.0)\)

If necessary, better results can be computed by implementing more accurate APPX functions.
The Instruction Set

ARL: Address Register Load

Background:

96 program parameters accessed through “c” registers.

Direct addressing:

i.e.  c[0], c[7], c[4]

Relative addressing:

only via “address register” A0.x
i.e. c[A0.x + offset]
The Instruction Set

ARL: Address Register Load

Function:
Loads the floor(s) into the address register for some scalar s.

Syntax:
ARL A0.x, src0.C

where ‘C’ is x, y, z, or w
## The Instruction Set

### ARL Example:

```plaintext
ARL   A0.x, R8.y;
MOV   R9, c[A0.x + 2];
```

<table>
<thead>
<tr>
<th></th>
<th>A0</th>
<th>R8</th>
<th>R9</th>
<th>c[7]</th>
<th></th>
<th>A0</th>
<th>R8</th>
<th>R9</th>
<th>c[7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>2.0</td>
<td>0.0</td>
<td>17.0</td>
<td>x</td>
<td>5</td>
<td>2.0</td>
<td>17.0</td>
<td>17.0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>5.9</td>
<td>0.0</td>
<td>22.0</td>
<td>y</td>
<td>0</td>
<td>5.9</td>
<td>22.0</td>
<td>22.0</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>4.2</td>
<td>0.0</td>
<td>3.0</td>
<td>z</td>
<td>0</td>
<td>4.2</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>w</td>
<td>0</td>
<td>7.0</td>
<td>0.0</td>
<td>3.0</td>
<td>w</td>
<td>0</td>
<td>7.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>
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:

\[ \text{src0.x} = \text{diffuse dot product (N} \cdot \text{L)} \]
\[ \text{src0.y} = \text{specular dot product (N} \cdot \text{H)} \]
\[ \text{src0.w} = \text{power (m)} \]
The Instruction Set

LIT: Light Coefficients

Syntax:

\[ \text{LIT} \quad \text{dest, src0} \]

Result:

\[ \begin{align*}
\text{dest}.x &= 1.0 \\
\text{dest}.y &= \text{CLAMP}(\text{src0}.x, 0, 1) \\
&= \text{CLAMP}(N \cdot L, 0, 1) \\
\text{dest}.z &= (\text{see next slide...}) \\
\text{dest}.w &= 1.0
\end{align*} \]

(ambient coeff.)

(diffuse coeff.)

(specular coeff.)
The Instruction Set

LIT: Light Coefficients

Result: (Recall: \( src0.x \equiv N \cdot L, \ src0.y \equiv N \cdot H, \ src0.w \equiv m \))

\[
\text{if ( src0.x } > \ 0.0) \\
\quad \text{dest.z } = \ (\text{MAX}(src0.y,0))(\text{ECLAMP}(src0.w,-128,128)) \\
\quad = \ (\text{MAX}(N \cdot H,0))^m \quad \text{where m in (-128,128)}
\]

otherwise,
\[
\quad \text{dest.z } = \ 0.0
\]

(dest.z is specular coeff. as defined by OpenGL)
The Instruction Set

LIT Example:

\[
\text{LIT } R1, R7;
\]

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>R1</td>
<td>(ambient)</td>
</tr>
<tr>
<td></td>
<td>R7</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>(diffuse)</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>y</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>(specular)</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w</td>
</tr>
</tbody>
</table>

(ambient) 1.0
(diffuse) 0.3
(specular) 0.64

(Good to 8+ bits)
The Instruction Set

DST: Distance Vector

Function:

Efficiently computes a “distance attenuation” vector (1, d, d², 1/d) from two source scalars.

Assumes:

\[
\begin{align*}
\text{src0.C}_1 &= d^2 \\
\text{src1.C}_2 &= 1.0/d
\end{align*}
\]

(where “c₁” is x, y, z, or w)

“d” is some distance

\[
\begin{align*}
d &= | \text{light pos. – vertex pos.} | \\
d &= | \text{eye pos. – vertex pos.} |
\end{align*}
\]
The Instruction Set

DST: Distance Vector

Syntax:

```
DST dest, src0.C₁, src1.C₂
```

Result:

```
dest.x = 1
dest.y = src0.C₁ * src1.C₂
    = d
dest.z = src0.C₁
    = d²
dest.w = src1.C₂
    = 1/d
```
The Instruction Set

DST: Utility exemplified through an example...

Lighting example with distance attenuation:

modulate by \( \frac{1}{k_0 + k_1d + k_2d^2} \)
where \( d = |\text{light pos.} - \text{vertex pos.}| \)

Suppose vector \( R5 = \text{light pos.} - \text{vertex pos.} \)

= unnormalized light vector \( (L) \)

Likely need to normalize \( L \) for \( N \cdot L \) computation.
The Instruction Set

DST: Distance attenuation example...

Normalize L by:

\[
\begin{align*}
\text{DP3} & \quad \text{R0.w, R5, R5;} \quad \# \text{R0.w is } d^2 \\
\text{RSQ} & \quad \text{R1.w, R0.w;} \quad \# \text{R1.w is } 1/d \\
\text{MUL} & \quad \text{R5.xyz, R5, R1.w;} \quad \# \text{R5 is normalized}
\end{align*}
\]

Now get attenuation vector:

\[
\begin{align*}
\text{DST} & \quad \text{R6, R0.w, R1.w;} \quad \# \text{R6 is } (1,d,d^2,1/d)
\end{align*}
\]
The Instruction Set

DST: Distance attenuation example...

If program parameter register has attenuation coefficients (i.e. \( c[0] = (k_0, k_1, k_2, *) \)) ...

Get attenuation factor with 2 more instructions

\[
\begin{align*}
\text{DP3} & \quad R7.w, R6, c[0]; \quad \# \text{R7.w is } k_0 + k_1 d + k_2 d^2 \\
\text{RCP} & \quad R1.w, R0.w; \quad \# \text{R1.w is attenuation}
\end{align*}
\]

Same task would require additional instructions w/o DST.
The Instruction Set

DST Example:

\[
\text{DST} \quad \text{R1}, \text{R2.w}, \text{R3.w};
\]

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>2.0</td>
</tr>
<tr>
<td>z</td>
<td>0.0</td>
<td>4.0</td>
</tr>
<tr>
<td>w</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>R1 7.0 2.0 2.1</td>
<td>R2 3.0 6.0 5.0</td>
</tr>
<tr>
<td></td>
<td>R3 6.0 5.0 0.5</td>
<td>R3 4.0 6.0 5.0</td>
</tr>
</tbody>
</table>
The Instruction Set

What about more complex instructions?

Absolute Value: MAX R1, -R1;
Division: RCP; MUL
Matrix Transform: DP4; DP4; DP4; DP4
Cross-Product: MUL; MAD

Others...

NVIDIA will provide examples and programs.
The Instruction Set

What about branches?

No branching, no early exit

Why?
  • Execution Dependencies
  • Performance Implications

Can multiply by zero and accumulate.
Example Programs

3-Component Normalize

# R1 = (nx,ny,nz)
#
# R0.xyz = normalize(R1)
# R0.w = 1/sqrt(nx*nx + ny*ny + nz*nz)
#
DP3 R0.w, R1, R1;
RSQ R0.w, R0.w;
MUL R0.xyz, R1, R0.w;
Example Programs

3-Component Cross Product

# Cross product
# | i   j   k | into R2.
# | R0.x R0.y R0.z |
# | R1.x R1.y R1.z |
#
MUL R2, R0.zxyw, R1.yzxw;
MAD R2, R0.yzxw, R1.zxyw, -R2;
Determinant of a 3x3 Matrix

# Determinant of | R0.x  R0.y  R0.z | into R3
# | 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;
Example Programs

Simple Specular and Diffuse Lighting

!!VP1.0
#
# c[0-3] = modelview projection (composite) matrix
# c[4-7] = modelview inverse transpose
# c[32] = eye-space light direction
# c[33] = constant eye-space half-angle vector (infinite viewer)
# c[35].x = pre-multiplied monochromatic diffuse light color & diffuse mat.
# c[35].y = pre-multiplied monochromatic ambient light color & diffuse mat.
# c[36] = specular color
# c[38].x = specular power
# outputs homogenous position and color
#
DP4   o[HPOS].x, c[0], v[OPOS];      # Compute position.
DP4   o[HPOS].y, c[1], v[OPOS];
DP4   o[HPOS].z, c[2], v[OPOS];
DP4   o[HPOS].w, c[3], v[OPOS];
DP3   R0.x, c[4], v[NRML];           # Compute normal.
DP3   R0.y, c[5], v[NRML];
DP3   R0.z, c[6], v[NRML];           # R0 = N' = transformed normal
DP3   R1.x, c[32], R0;               # R1.x = Ldir DOT N'
DP3   R1.y, c[33], R0;               # R1.y = H DOT N'
MOV   R1.w, c[38].x;                 # R1.w = specular power
LIT   R2, R1;                        # Compute lighting values
MAD   R3, c[35].x, R2.y, c[35].y;   # diffuse + ambient
MAD   o[COL0].xyz, c[36], R2.z, R3;  # + specular
END
Performance

- Programs managed similar to texture objects
  - Switching between small number of programs is fast!
  - Switching between large number of programs is slower.
  - Use glRequestProgramsResidentNV() to define a small set of programs which can be switched quickly.
Performance

- Use vertex programming when required
- Use conventional OpenGL T&L mode when not

  - There is no penalty for switching in and out of vertex program mode.

- Vertex Program execution time
  ~ proportional to length of program

shorter programs → faster execution
Performance

For Optimal performance …

- Be clever!
- Exploit vector parallelism
  - (Ex. 4 scalar adds with a vector add)
- Swizzle and negate away
  - (no performance penalty for doing so)
- Use LIT and DST effectively
- Use Vertex State Programs for “pre-processing”.

98
Summary – Vertex Programs ROCK!

- Increased programmability
  - Customizable engine for transform, lighting, texture coordinate generation, and more.
  - Facilitates setup for per-fragment shading.
  - Allows animation/deformation through key-frame interpolation and skinning.

- Accelerated in Future Generation GPUs!
  - Offloads CPU tasks to GPU yielding higher performance.
For More Information…

- **NVIDIA OpenGL Extension Specification**
  - Explains exactly how the NV_vertex_program extension works “in detail”

- **NVIDIA OpenGL SDK**
  - Technical Demos
  - Lab Exercises of varying difficulty
  - Additional Documentation

- **Available at NVIDIA Developer Website:**
Questions, comments, feedback

- Chris Wynn, cwynn@nvidia.com
- www.nvidia.com/developer