OpenGL ARB Vertex Program

Cass Everitt
cass@nvidia.com
Overview

- ARB Vertex Programming Overview
- Loading Vertex Programs
- Register Set
- Variables and Variable “Binding”
- Assembly Instruction Set
- Example Programs
- Wrap-Up
ARB Vertex Programming

Overview

• Traditional Graphics Pipeline

Each unit has specific function (usually with configurable “modes” of operation)
ARB Vertex Programming

Overview

- Vertex Programming offers programmable T&L unit

Transform, lighting

Setup

Rasterizer

Texture fetch, fragment shading

Frame-buffer ops

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
What is Vertex Programming?

• **Vertex Program**
  - Assembly language interface to T&L unit
  - GPU instruction set to perform all vertex math
  - **Input**: arbitrary vertex attributes
  - **Output**: a transformed vertex attributes
    - homogeneous clip space position (required)
    - colors (front/back, primary/secondary)
    - fog coord
    - texture coordinates
    - point size
What is Vertex Programming?

- **Vertex Program**
  - Does not generate or destroy vertexes
    - 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 and EXT_vertex_shader extensions
- and now ARB_vertex_program
What is ARB_vertex_program?

- ARB_vertex_program is similar to NV_vertex_program with the addition of:
  - variables
  - local parameters
  - access to GL state
  - some extra instructions
  - implementation-specific resource limits
What is Vertex Programming?

glEnable(GL_VERTEX_PROGRAM_ARB);

Switch from conventional T&L model to Programmable mode
Specifically, what gets bypassed?

- Modelview and projection vertex transformations
- Vertex weighting/blending
- Normal transformation, rescaling, normalization
- Color material
- Per-vertex lighting
- Texture coordinate generation and texture matrix transformations
- Per-vertex point size and fog coordinate computations
- User-clip planes
What does NOT get bypassed?

- Evaluators
- Clipping to the view frustum
- Perspective divide
- Viewport transformation
- Depth range transformation
- Front and back color selection (for two-sided)
- Clamping of primary and secondary colors to [0,1]
- Primitive assembly, setup, rasterization, blending
Vertex Programming
Conceptual Overview

Program Environment Parameters
Ex4 registers (E ≥ 96)

Address Variables
Ax4 variables (A ≥ 1)

Vertex Attributes
Nx4 registers (N ≥ 16)

Program Local Parameters
Lx4 registers (L ≥ 96)

Temporary Variables
Tx4 variables (T ≥ 12)

Vertex Result Registers
Rx4 registers (R ≥ 8)

M instructions (M ≥ 128)
Creating a Vertex Program

- Programs are arrays of GLubyte ("strings")
- Created/managed similar to texture objects
  - notion of a *program object*
  - `glGenProgramsARB(sizei n, uint *ids)`
  - `glBindProgramARB(enum target, uint id)`
  - `glProgramStringARB(enum target, enum format, sizei len, const ubyte *program)`
Creating a Vertex Program

Gluint progid;

// Generate a program object handle.
glGenProgramsARB( 1, &progid );

// Make the “current” program object progid.
glBindProgramARB( GL_VERTEX_PROGRAM_ARB, progid );

// Specify the program for the current object.
glProgramStringARB( GL_VERTEX_PROGRAM_ARB,
    GL_PROGRAM_FORMAT_ASCII_ARB,
    strlen(myString), myString );

// Check for errors and warnings...
Creating a Vertex Program

```c
// Check for errors and warnings...
if ( GL_INVALID_OPERATION == glGetError() ) {

    // Find the error position
    GLint errPos;
    glGetIntegerv( GL_PROGRAM_ERROR_POSITION_ARB, &errPos);

    // Print implementation-dependent program
    // errors and warnings string.
    Glubyte *errString;
    glGetString( GL_PROGRAM_ERROR_STRING_ARB, &errString);

    fprintf( stderr, “error at position: %d\n%s\n”, errPos, errString );
}
```
Creating a Vertex Program

• When finished with a program object, delete it

```c
// Delete the program object.
glDeleteProgramsARB( 1, &progid );
```
Specifying Program Parameters

- Three types
  - Vertex Attributes – specifiable per-vertex
  - Program Local Parameters
  - Program Environment Parameters

Program Parameters modifiable outside of a Begin/End block
Specifying Vertex Attributes

- Up to Nx4 per-vertex “generic” attributes
- Values specified with (several) new commands
  
  \[ \text{glVertexAttrib4fARB}(\text{index}, x, y, z, w) \]
  \[ \text{glVertexAttribs4fvARB}(\text{index}, \text{values}) \]

- Some entry points allow component-wise linear re-mapping to [0,1] or [-1,1]
  
  \[ \text{glVertexAttrib4NubARB}(\text{index}, x, y, z, w) \]
  \[ \text{glVertexAttrib4NbvARB}(\text{index}, \text{values}) \]

  similar to \text{glColor4ub()} and \text{glColor4b()}
## Specifying Vertex Attributes

Component-wise linear re-mapping

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Data Type</th>
<th>Min Value</th>
<th>Min Value</th>
<th>Maps to</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1-byte integer</td>
<td>-128</td>
<td>-1.0</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>2-byte integer</td>
<td>-32,768</td>
<td>-1.0</td>
<td></td>
</tr>
<tr>
<td>i4</td>
<td>4-byte integer</td>
<td>-2,147,483,648</td>
<td>-1.0</td>
<td></td>
</tr>
<tr>
<td>ub</td>
<td>unsigned 1-byte integer</td>
<td>0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>us</td>
<td>unsigned 2-byte integer</td>
<td>0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>ui</td>
<td>unsigned 4-byte integer</td>
<td>0</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Data Type</th>
<th>Max Value</th>
<th>Max Value</th>
<th>Maps to</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1-byte integer</td>
<td>127</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>2-byte integer</td>
<td>32,767</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>i4</td>
<td>4-byte integer</td>
<td>2,147,483,647</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>ub</td>
<td>unsigned 1-byte integer</td>
<td>255</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>us</td>
<td>unsigned 2-byte integer</td>
<td>65,535</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>ui</td>
<td>unsigned 4-byte integer</td>
<td>4,294,967,295</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
Specifying Vertex Attributes

- Vertex Array support
- `glVertexAttribPointerARB(
  uint index,
  int size,
  enum type,
  boolean normalize,
  sizei stride,
  const void *pointer )`

- “normalize” flag indicates if values should be linearly remapped
Specifying Vertex Attributes

• Setting vertex attribute 0 provokes vertex program execution
• Setting any other vertex attribute updates the current values of the attribute register

• Conventional attributes may be specified with conventional per-vertex calls
  • glColor, glNormal, glWeightARB, etc.
• Not strict aliasing (like NV_vertex_program)
  • More on this later…
Specifying Program Local Parameters

- Each program object has an array of \((N \geq 96)\) four-component floating point vectors
  - Store program-specific parameters required by the program

- Values specified with new commands
  - `glProgramLocalParameter4fARB(GL_VERTEX_PROGRAM_ARB, index, x, y, z, w)`
  - `glProgramLocalParameter4fvARB(GL_VERTEX_PROGRAM_ARB, index, params)`

- Correspond to 96+ local parameter registers
Specifying Program Environment Parameters

- Shared array of \((N \geq 96)\) four-component registers accessible by any vertex program
  - Store parameters common to a set of program objects (i.e. Modelview matrix, MVP matrix)

- Values specified with new commands
  - `glProgramEnvParameter4fARB(GL_VERTEX_PROGRAM_ARB, index, x, y, z, w)`
  - `glProgramEnvParameter4fvARB(GL_VERTEX_PROGRAM_ARB, index, params)`

- Correspond to 96+ environment registers
The Register Set

Program Environment Parameters
- program.env[0]
- ... program.env[N-1]

Address Variables
- User defined

Vertex Attributes
- vertex.*

Program Local Parameters
- program.local[0]
- ... program.local[N-1]

Temporary Variables
- User defined

Vertex Program

Vertex Result Registers
- result.*
Program Environment and Program Local Registers

- Program environment registers
  access using: `program.env[i]`
  \[ i \text{ in } [0, \text{GL_MAX_PROGRAM_ENV_PARAMETERS_ARB}-1] \]

- Program local registers
  access using: `program.local[i]`
  \[ i \text{ in } [0, \text{GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB}-1] \]
## Vertex Attribute Registers

<table>
<thead>
<tr>
<th>Attribute Register</th>
<th>Components</th>
<th>Underlying State</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex.position</td>
<td>(x,y,z,w)</td>
<td>object position</td>
</tr>
<tr>
<td>vertex.weight</td>
<td>(w,w,w,w)</td>
<td>vertex weights 0-3</td>
</tr>
<tr>
<td>vertex.weight[n]</td>
<td>(w,w,w,w)</td>
<td>vertex weights n-n+3</td>
</tr>
<tr>
<td>vertex.normal</td>
<td>(x,y,z,1)</td>
<td>normal</td>
</tr>
<tr>
<td>vertex.color</td>
<td>(r,g,b,a)</td>
<td>primary color</td>
</tr>
<tr>
<td>vertex.color.primary</td>
<td>(r,g,b,a)</td>
<td>primary color</td>
</tr>
<tr>
<td>vertex.color.secondary</td>
<td>(r,g,b,a)</td>
<td>secondary color</td>
</tr>
<tr>
<td>vertex.fogcoord</td>
<td>(f,0,0,1)</td>
<td>fog coordinate</td>
</tr>
<tr>
<td>vertex.texcoord</td>
<td>(s,t,r,q)</td>
<td>texture coordinate, unit 0</td>
</tr>
<tr>
<td>vertex.texcoord[n]</td>
<td>(s,t,r,q)</td>
<td>texture coordinate, unit n</td>
</tr>
<tr>
<td>vertex.matrixindex</td>
<td>(i,i,i,i)</td>
<td>vertex matrix indices 0-3</td>
</tr>
<tr>
<td>vertex.matrixindex[n]</td>
<td>(i,i,i,i)</td>
<td>vertex matrix indices n-n+3</td>
</tr>
<tr>
<td>vertex.attrib[n]</td>
<td>(x,y,z,w)</td>
<td>generic vertex attribute n</td>
</tr>
</tbody>
</table>

Semantics defined by program, NOT parameter name.
# Vertex Result Registers

<table>
<thead>
<tr>
<th>Result Register</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result.position</td>
<td>(x,y,z,w)</td>
<td>position in clip coordinates</td>
</tr>
<tr>
<td>result.color</td>
<td>(r,g,b,a)</td>
<td>front-facing, primary color</td>
</tr>
<tr>
<td>result.color.primary</td>
<td>(r,g,b,a)</td>
<td>front-facing, primary color</td>
</tr>
<tr>
<td>result.color.secondary</td>
<td>(r,g,b,a)</td>
<td>front-facing, secondary color</td>
</tr>
<tr>
<td>result.color.front</td>
<td>(r,g,b,a)</td>
<td>front-facing, primary color</td>
</tr>
<tr>
<td>result.color.front.primary</td>
<td>(r,g,b,a)</td>
<td>front-facing, primary color</td>
</tr>
<tr>
<td>result.color.front.secondary</td>
<td>(r,g,b,a)</td>
<td>front-facing, secondary color</td>
</tr>
<tr>
<td>result.color.back</td>
<td>(r,g,b,a)</td>
<td>back-facing, primary color</td>
</tr>
<tr>
<td>result.color.back.primary</td>
<td>(r,g,b,a)</td>
<td>back-facing, primary color</td>
</tr>
<tr>
<td>result.color.back.secondary</td>
<td>(r,g,b,a)</td>
<td>back-facing, secondary color</td>
</tr>
<tr>
<td>result.fogcoord</td>
<td>(f,<em>,</em>,*)</td>
<td>fog coordinate</td>
</tr>
<tr>
<td>result.pointsize</td>
<td>(s,<em>,</em>,*)</td>
<td>point size</td>
</tr>
<tr>
<td>result.texcoord</td>
<td>(s,t,r,q)</td>
<td>texture coordinate, unit 0</td>
</tr>
<tr>
<td>result.texcoord[n]</td>
<td>(s,t,r,q)</td>
<td>texture coordinate, unit n</td>
</tr>
</tbody>
</table>

Semantics defined by down-stream pipeline stages
Address Register Variables

- four-component signed integer vectors where only the ‘x’ component is addressable.

- Must be “declared” before use – address register variables
  
  ```
  ADDRESS Areg;
  ADDRESS A0;
  ADDRESS A1, Areg;
  ```

- Number of variables limited to
  
  `GL_MAX_PROGRAM_ADDRESS_REGISTERS_ARB`
Temporary Variables

- Four-component floating-point vectors used to store intermediate computations

- Temporary variables declared before first use

  ```
  TEMP flag;
  TEMP tmp, ndotl, keenval;
  ```

- Number of temporary variables limited to `GL_MAX_PROGRAM_TEMPORARIES_ARB`
Identifiers and Variable Names

- Any sequence of one or more
  - letters (A to Z, a to z),
  - digits ("0" to "9")
  - underscores ("_")
  - dollar signs "$"
- First character may not be a digit
- Case sensitive
- Legal: A, b, _ab, $_ab, a$b, $
- Not Legal: 9A, ADDRESS, TEMP
  (other reserved words)
Program Constants

- Floating-point constants may be used in programs
- Standard format

\(<\text{integer portion}> . \ <\text{fraction portion}> \ \{\text{“e”}<\text{integer}>| \text{“E”}<\text{integer}>\}\)

One (not both) may be omitted
Decimal or exponent (not both) may be omitted

- Some Legal examples
  
  4.3, 4., .3, 4.3e3, 4.3e-3, 4.e3, 4e3, 4.e-3, .3e3
Program Parameter Variables

- Set of four-component floating point vectors used as constants during program execution

- May be single four-vector or array of four-vectors

- Bound either
  - Explicitly (declaration of “param” variables)
  - Implicitly (inline usage of constants)
Program Parameter Variable Bindings

• Explicit Constant Binding

• Single Declaration

 PARAM a = {1.0, 2.0, 3.0, 4.0};  (1.0, 2.0, 3.0, 4.0)
 PARAM b = {3.0};              (3.0, 0.0, 0.0, 1.0)
 PARAM c = {1.0, 2.0};         (1.0, 2.0, 0.0, 1.0)
 PARAM d = {1.0, 2.0, 3.0 };   (1.0, 2.0, 3.0, 1.0)
 PARAM e = 3.0;                (3.0, 3.0, 3.0, 3.0)

• Array Declaration

 PARAM arr[2] = { {1.0, 2.0, 3.0, 4.0},
                  {5.0, 6.0, 7.0, 8.0} };
Program Parameter Variable Bindings

• Implicit Constant Binding

ADD a, b, {1.0, 2.0, 3.0, 4.0}; (1.0, 2.0, 3.0, 4.0)
ADD a, b, {3.0}; (3.0, 0.0, 0.0, 1.0)
ADD a, b, {1.0, 2.0}; (1.0, 2.0, 0.0, 1.0)
ADD a, b, {1.0, 2.0, 3.0}; (1.0, 2.0, 3.0, 1.0)
ADD a, b, 3.0; (3.0, 3.0, 3.0, 3.0)

• Number of program parameter variables (explicit+implicit) limited to GL_MAX_PROGRAM_PARAMETERS_ARB
Program Parameter Variable Bindings

- Program Environment/Local Parameter Binding

```c
PARAM a = program.local[8];
PARAM b = program.env[9];
PARAM arr[2] = program.local[4..5];
PARAM mat[4] = program.env[0..3];
```

- Essentially creates a “Reference”
Program Parameter Variable Bindings

- Material Property Binding
  - Bind to current GL material properties

PARAM ambient = state.material.ambient;
PARAM diffuse = state.material.diffuse;

- Additional material state to bind to...
### Program Parameter Variable Bindings

<table>
<thead>
<tr>
<th>Binding</th>
<th>Components</th>
<th>Underlying GL state</th>
</tr>
</thead>
<tbody>
<tr>
<td>state.material.ambient</td>
<td>(r,g,b,a)</td>
<td>front ambient material color</td>
</tr>
<tr>
<td>state.material.diffuse</td>
<td>(r,g,b,a)</td>
<td>front diffuse material color</td>
</tr>
<tr>
<td>state.material.specular</td>
<td>(r,g,b,a)</td>
<td>front specular material color</td>
</tr>
<tr>
<td>state.material.emission</td>
<td>(r,g,b,a)</td>
<td>front emissive material color</td>
</tr>
<tr>
<td>state.material.shininess</td>
<td>(s,0,0,1)</td>
<td>front material shininess</td>
</tr>
<tr>
<td>state.material.front.ambient</td>
<td>(r,g,b,a)</td>
<td>front ambient material color</td>
</tr>
<tr>
<td>state.material.front.diffuse</td>
<td>(r,g,b,a)</td>
<td>front diffuse material color</td>
</tr>
<tr>
<td>state.material.front.specular</td>
<td>(r,g,b,a)</td>
<td>front specular material color</td>
</tr>
<tr>
<td>state.material.front.emission</td>
<td>(r,g,b,a)</td>
<td>front emissive material color</td>
</tr>
<tr>
<td>state.material.front.shininess</td>
<td>(s,0,0,1)</td>
<td>front material shininess</td>
</tr>
<tr>
<td>state.material.back.ambient</td>
<td>(r,g,b,a)</td>
<td>back ambient material color</td>
</tr>
<tr>
<td>state.material.back.diffuse</td>
<td>(r,g,b,a)</td>
<td>back diffuse material color</td>
</tr>
<tr>
<td>state.material.back.specular</td>
<td>(r,g,b,a)</td>
<td>back specular material color</td>
</tr>
<tr>
<td>state.material.back.emission</td>
<td>(r,g,b,a)</td>
<td>back emissive material color</td>
</tr>
<tr>
<td>state.material.back.shininess</td>
<td>(s,0,0,1)</td>
<td>back material shininess</td>
</tr>
</tbody>
</table>
Program Parameter Variable Bindings

- **Light Property Binding**
  ```c
  PARAM ambient = state.light[0].ambient;
  PARAM diffuse = state.light[0].diffuse;
  ```

- **Additional light state to bind to...**
- **Also bind to**
  - Texture coord generation state
  - Fog property state
  - Clip plane state
  - Matrix state
Output Variables

- Variables that are declared bound to any vertex result register

  ```
  OUTPUT ocol = result.color.primary;
  OUTPUT opos = result.position;
  ```

- Write-only, essentially a “reference”
Aliasing of Variables

- Allows multiple variable names to refer to a single underlying variable
  
  ALIAS var2 = var1;

- Do not count against resource limits
Additional Notes on Variables

• May be declared anywhere prior to first usage

• ARB spec. details specific rules with regards to resource consumption
  • Rule of thumb – generally minimize/remove unessential variables to keep resource counts
  • Can always load a program then query resource counts if desired
Vertex Programming Assembly Language

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

Instruction Format:

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

Examples:

MOV     R1, R2;
MAD     R1, R2, R3, -R4;

‘[’ and ‘]’ indicate optional modifiers
Assembly Language

Source registers can be negated:

```
MOV R1, –R2;
```
Assembly Language

Source registers can be “swizzled”:

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

```
before

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

after

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

Note: MOV R1, R2.xxxx; ↔ MOV R1, R2.x;
Assembly Language

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

R1 \implies \text{write all components}

R1.x \implies \text{write only x component}

R1.xw \implies \text{write only x, w components}
Vertex Programming
Assembly Language

Destination register masking:

```
MOV R1.xw, -R2;
```

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th></th>
<th>R2</th>
<th></th>
<th>R1</th>
<th></th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>0.0</td>
<td>7.0</td>
<td>x</td>
<td>y</td>
<td>6.0</td>
<td>z</td>
<td>w</td>
</tr>
<tr>
<td>after</td>
<td>-7.0</td>
<td>7.0</td>
<td>x</td>
<td>y</td>
<td>3.0</td>
<td>z</td>
<td>w</td>
</tr>
</tbody>
</table>
There are 27 instructions in total …

- ABS
- ADD
- ARL
- DP3
- DP4
- DPH
- DST
- EX2
- EXP
- FLR
- FRC
- LG2
- LIT
- MAD
- MAX
- MIN
- MOV
- MUL
- POW
- RSQ
- SGE
- SLT
- SUB
- SWZ
- XPD
- RCP
Example Program #1

Simple Transform to CLIP space

!!ARBvp1.0

ATTRIB pos = vertex.position;
PARAM mat[4] = { state.matrix.mvp };

# Transform by concatenation of the
# MODELVIEW and PROJECTION matrices.
DP4  result.position.x, mat[0], pos;
DP4  result.position.y, mat[1], pos;
DP4  result.position.z, mat[2], pos;
DP4  result.position.w, mat[3], pos;

# Pass the primary color through w/o lighting.
MOV  result.color, vertex.color;

END
Example Program #2

Simple ambient, specular, and diffuse lighting
(single, infinite light, local viewer)

!!ARBvp1.0

ATTRIB iPos = vertex.position;
ATTRIB iNormal = vertex.normal;
PARAM mvinv[4] = { state.matrix.modelview.invtrans };
PARAM lightDir = state.light[0].position;
PARAM halfDir = state.light[0].half;
PARAM specExp = state.material.shininess;
PARAM ambientCol = state.lightprod[0].ambient;
PARAM diffuseCol = state.lightprod[0].diffuse;
PARAM specularCol = state.lightprod[0].specular;
TEMP eyeNormal, temp, dots, lightcoeffs;
OUTPUT oPos = result.position;
OUTPUT oColor = result.color;
Example Program #2

# Transform the vertex to clip coordinates.
DP4 oPos.x, mvp[0], iPos;
DP4 oPos.y, mvp[1], iPos;
DP4 oPos.z, mvp[2], iPos;
DP4 oPos.w, mvp[3], iPos;

# Transform the normal into eye space.
DP3 eyeNormal.x, mvinv[0], iNormal;
DP3 eyeNormal.y, mvinv[1], iNormal;
DP3 eyeNormal.z, mvinv[2], iNormal;

# Compute diffuse and specular dot products
# and use LIT to compute lighting coefficients.
DP3 dots.x, eyeNormal, lightDir;
DP3 dots.y, eyeNormal, halfDir;
MOV dots.w, specExp.x;
LIT lightcoefs, dots;

# Accumulate color contributions.
MAD temp, lightcoefs.y, diffuseCol, ambientCol;
MAD oColor.xyz, lightcoefs.z, specularCol, temp;
MOV oColor.w, diffuseCol.w;
Program Options

- OPTION mechanism for future extensibility
- Only one option: ARB_position_invariant
  - Guarantees position of vertex is same as what it would be if vertex program mode is disabled
  - User clipping also performed
  - Useful for “mixed-mode multi-pass”
- At start of program
  - OPTION ARB_position_invariant
- Error if program attempts to write to result.position
Querying Implementation-specific Limits

- **Max number of instructions**
  ```c
  glGetProgramivARB( GL_VERTEX_PROGRAM_ARB,
                    GL_MAX_PROGRAM_INSTRUCTIONS, &maxInsts );
  ```

- **Max number of temporaries**
  ```c
  glGetProgramivARB( GL_VERTEX_PROGRAM_ARB,
                    GL_MAX_PROGRAM_INSTRUCTIONS, &maxTemps );
  ```

- **Max number of program parameter bindings**
  ```c
  glGetProgramivARB( GL_VERTEX_PROGRAM_ARB,
                    GL_MAX_PROGRAM_PARAMETERS, &maxParams );
  ```

- **Others (including native limits)**
  Query current program resource usage by removing “MAX_”
Generic vs. Conventional Vertex Attributes

• ARB_vertex_program spec allows for “fast and loose” storage requirements for generic and conventional attributes...

• Mapping between Generic Attributes and Conventional ones

• When a generic attribute is specified using glVertexAttrib*(), the current value for the corresponding conventional attribute becomes undefined
  • Also true for the converse
Generic vs. Conventional Vertex Attributes

- This allows implementations flexibility
- Mapping defined in the spec.
- Single programs may not access both
  A generic attribute register
  AND
  Its corresponding conventional attribute register
- Error if it attempts to
## Generic and Conventional Attribute Mappings

<table>
<thead>
<tr>
<th>Conventional Attribute</th>
<th>Generic Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex.position</td>
<td>vertex.attrib[0]</td>
</tr>
<tr>
<td>vertex.weight</td>
<td>vertex.attrib[1]</td>
</tr>
<tr>
<td>vertex.weight[0]</td>
<td>vertex.attrib[1]</td>
</tr>
<tr>
<td>vertex.normal</td>
<td>vertex.attrib[2]</td>
</tr>
<tr>
<td>vertex.color</td>
<td>vertex.attrib[3]</td>
</tr>
<tr>
<td>vertex.color.primary</td>
<td>vertex.attrib[3]</td>
</tr>
<tr>
<td>vertex.color.secondary</td>
<td>vertex.attrib[4]</td>
</tr>
<tr>
<td>vertex.fogcoord</td>
<td>vertex.attrib[5]</td>
</tr>
<tr>
<td>vertex.texcoord</td>
<td>vertex.attrib[8]</td>
</tr>
<tr>
<td>vertex.texcoord[0]</td>
<td>vertex.attrib[8]</td>
</tr>
</tbody>
</table>

In practice, probably use either conventional or generic not both.
Wrap-Up

- Increased programmability
  - Customizable engine for transform, lighting, texture coordinate generation, and more.

- Widely available!
  - great, portable target for higher level abstractions

- Vendor extensions available for dynamic branching
  - will roll those into an ARBvp2 spec soon.
Questions?

• cass@nvidia.com