Vertex Programs

Chris Wynn

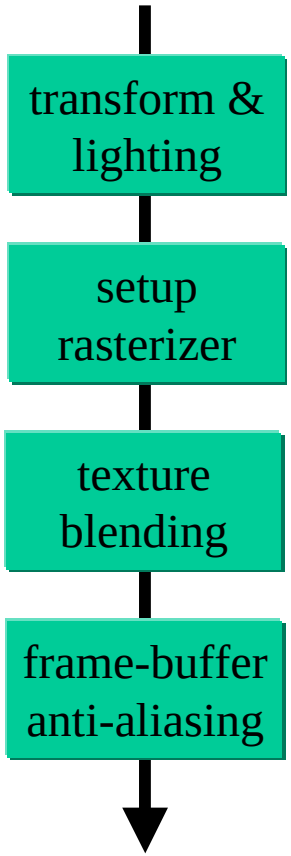
GAMEDevelopers CONFERENCE 2001

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?

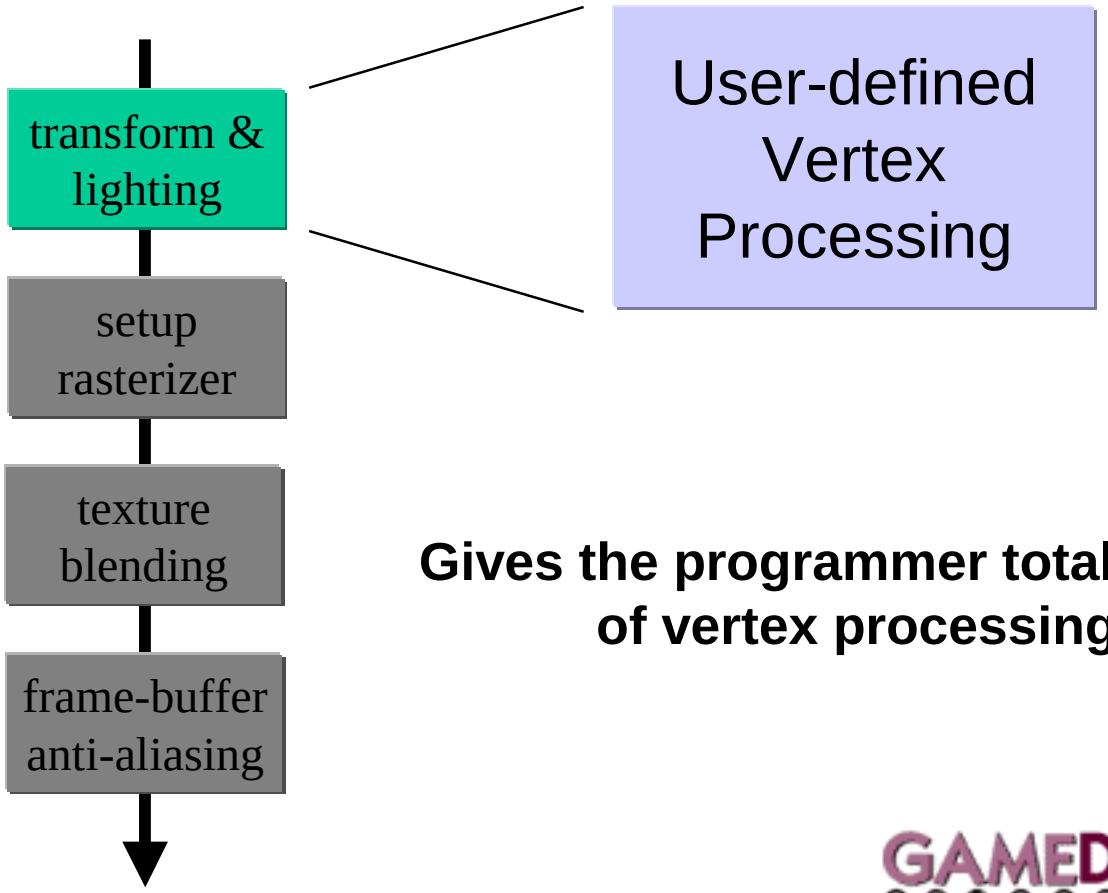
- **Traditional Graphics Pipeline**



Each unit has specific function (possibly with “modes” of operation)

What is Vertex Programming?

- **Vertex Programming offers programmable T&L unit**



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?

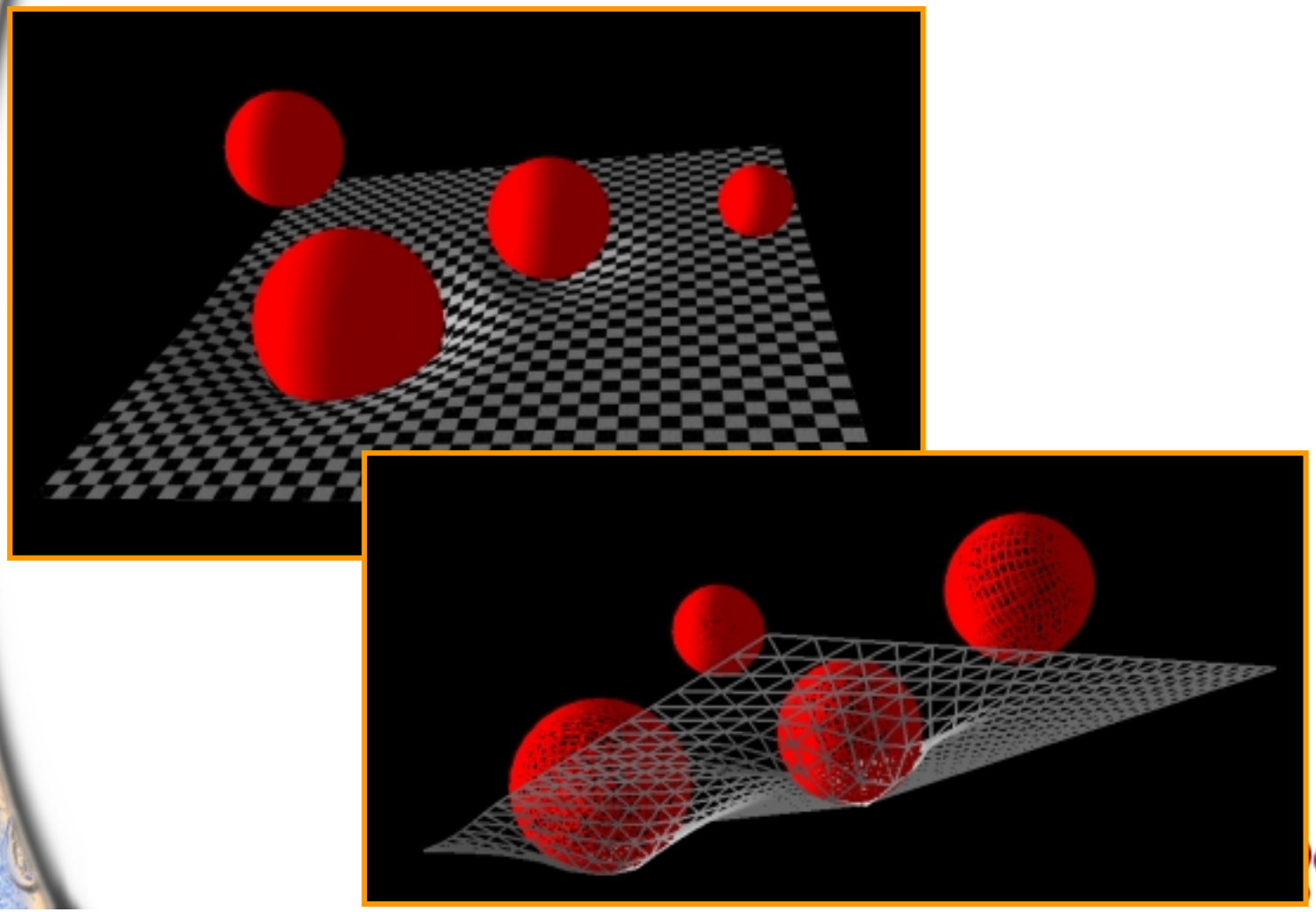
- **Per-vertex set up for per-pixel bump mapping**

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?

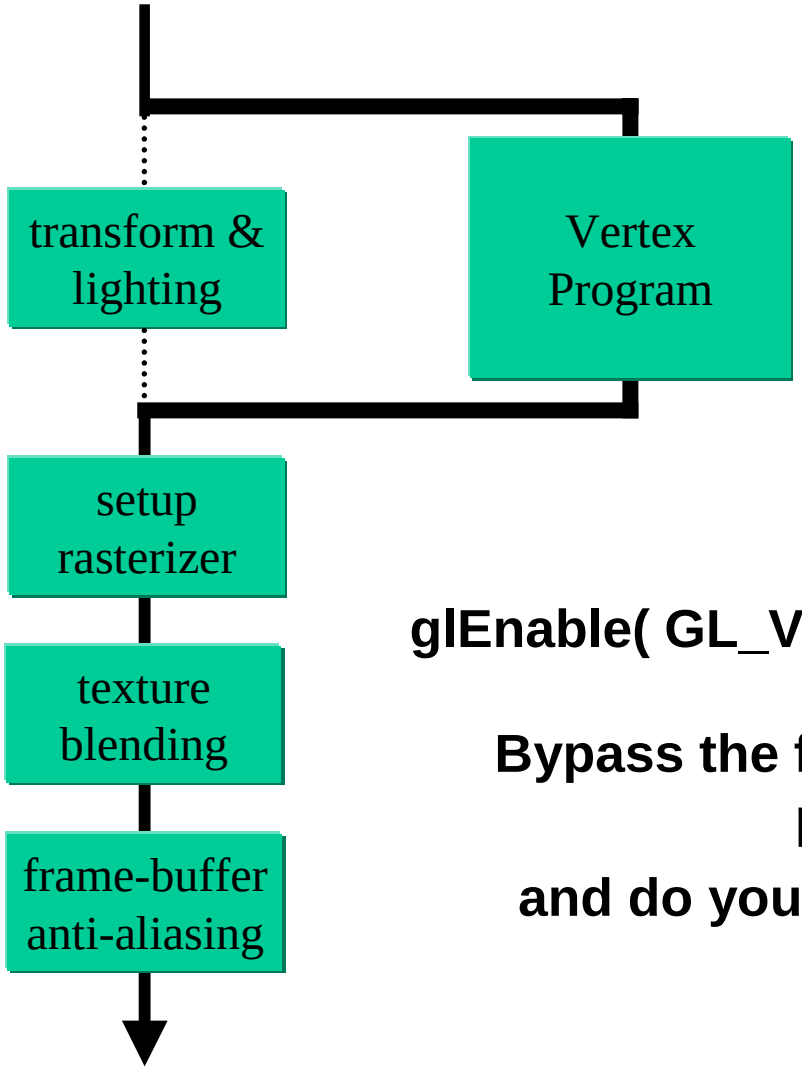
- **Vertex Program**
  - **Assembly language interface to T&L unit**
  - **GPU instruction set to perform all vertex math**
  - **Reads an untransformed, unlit vertex**
  - **Creates a transformed vertex**
  - **Optionally …**
    - **Lights a vertex**
    - **Creates texture coordinates**
    - **Creates 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?

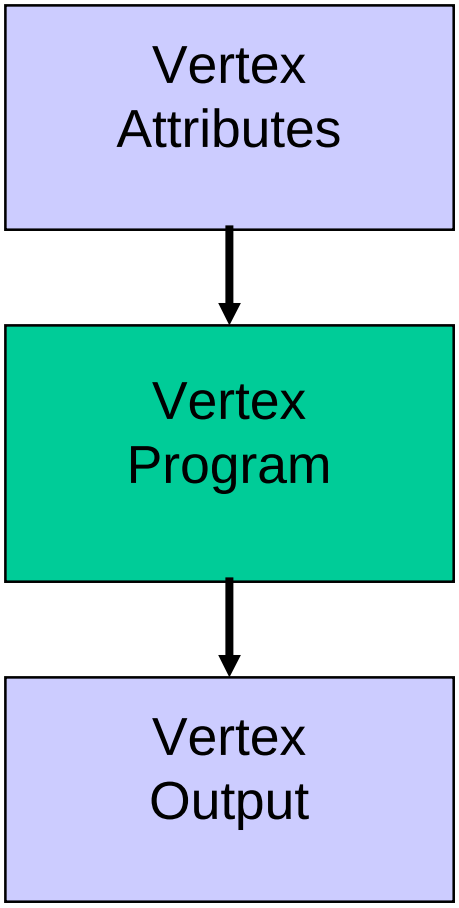


glEnable(GL_VERTEX_PROGRAM_NV);

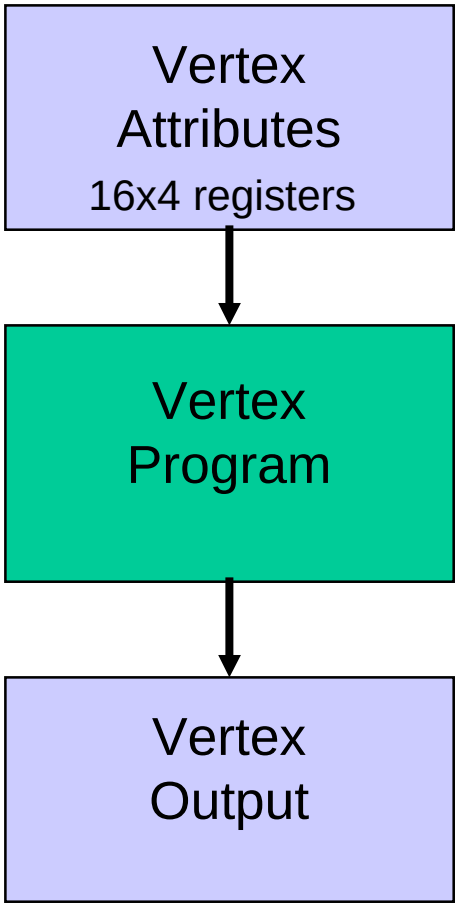
Bypass the fixed function T&L path…

and do your own thing in HW.

Vertex Programming Conceptual Overview



Vertex Programming Conceptual Overview

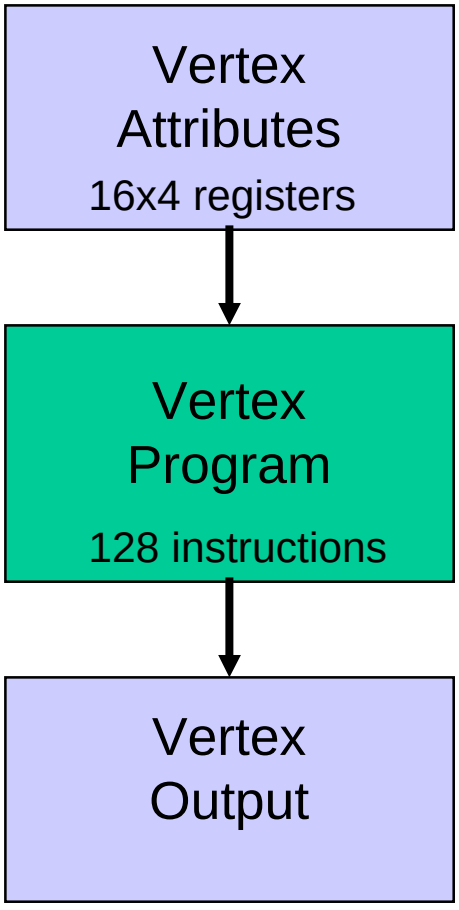


Sixteen 4-component vector floating point registers

Position, colors, normal

User-defined vertex parameters:
densities, velocities, weights, etc.

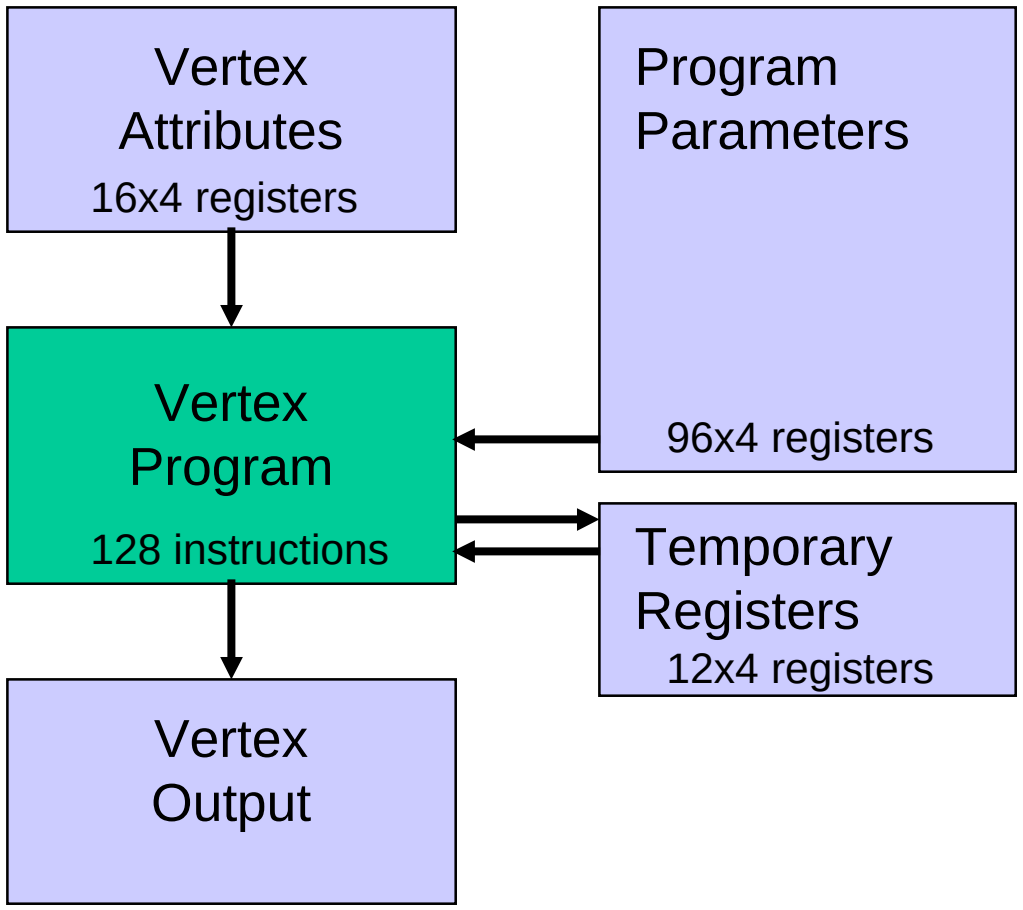
Vertex Programming Conceptual Overview



Up to 128 program instructions (SIMD)
(i.e. add, multiply, etc.)

Read vertex attribute registers
Write vertex output registers

Vertex Programming Conceptual Overview

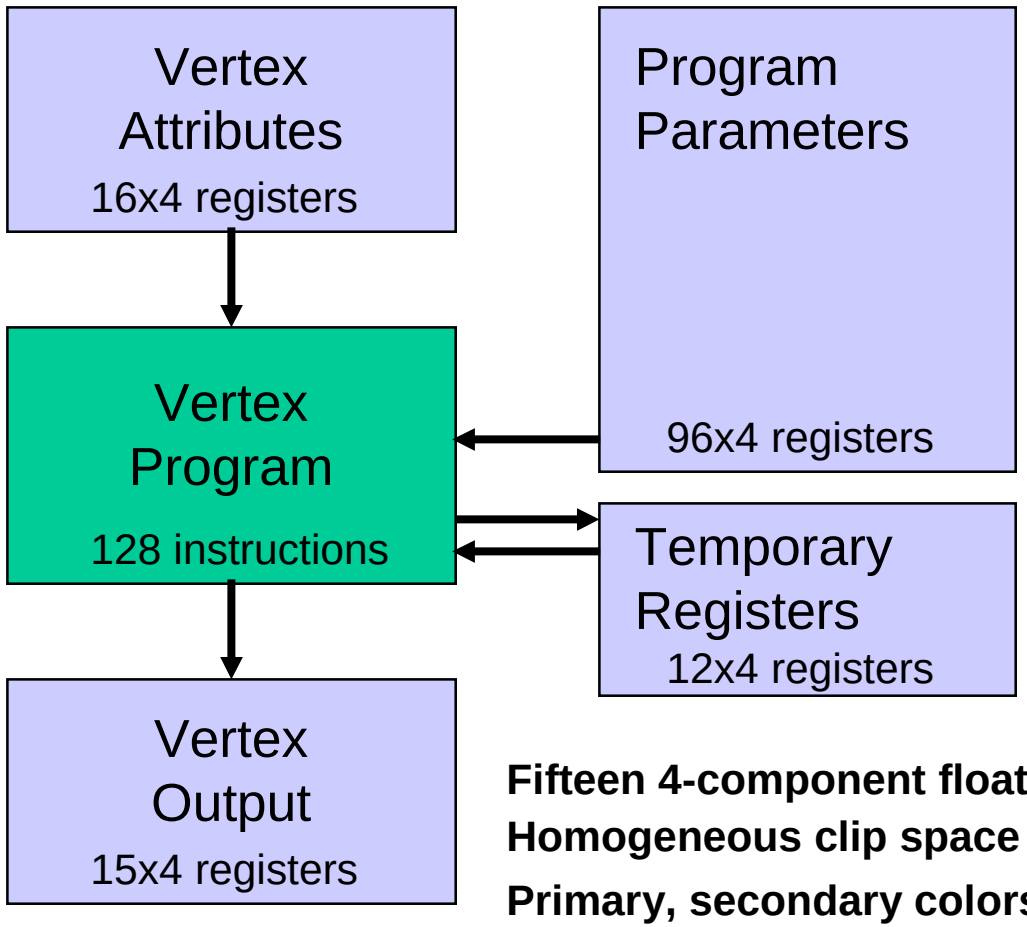


Modifiable only outside of glBegin/glEnd pair

Read-only

Read/Write-able

Vertex Programming Conceptual Overview



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

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 coordinate	glFogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	Texture coord 0	glMultiTexCoord	s,t,r,q
9	Texture coord 1	glMultiTexCoord	s,t,r,q
10	Texture coord 2	glMultiTexCoord	s,t,r,q
11	Texture coord 3	glMultiTexCoord	s,t,r,q
12	Texture coord 4	glMultiTexCoord	s,t,r,q
13	Texture coord 5	glMultiTexCoord	s,t,r,q
14	Texture coord 6	glMultiTexCoord	s,t,r,q
15	Texture coord 7	glMultiTexCoord	s,t,r,q

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

Vertex Programming Program Parameters

- **Matrix “Tracking”**

```
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.

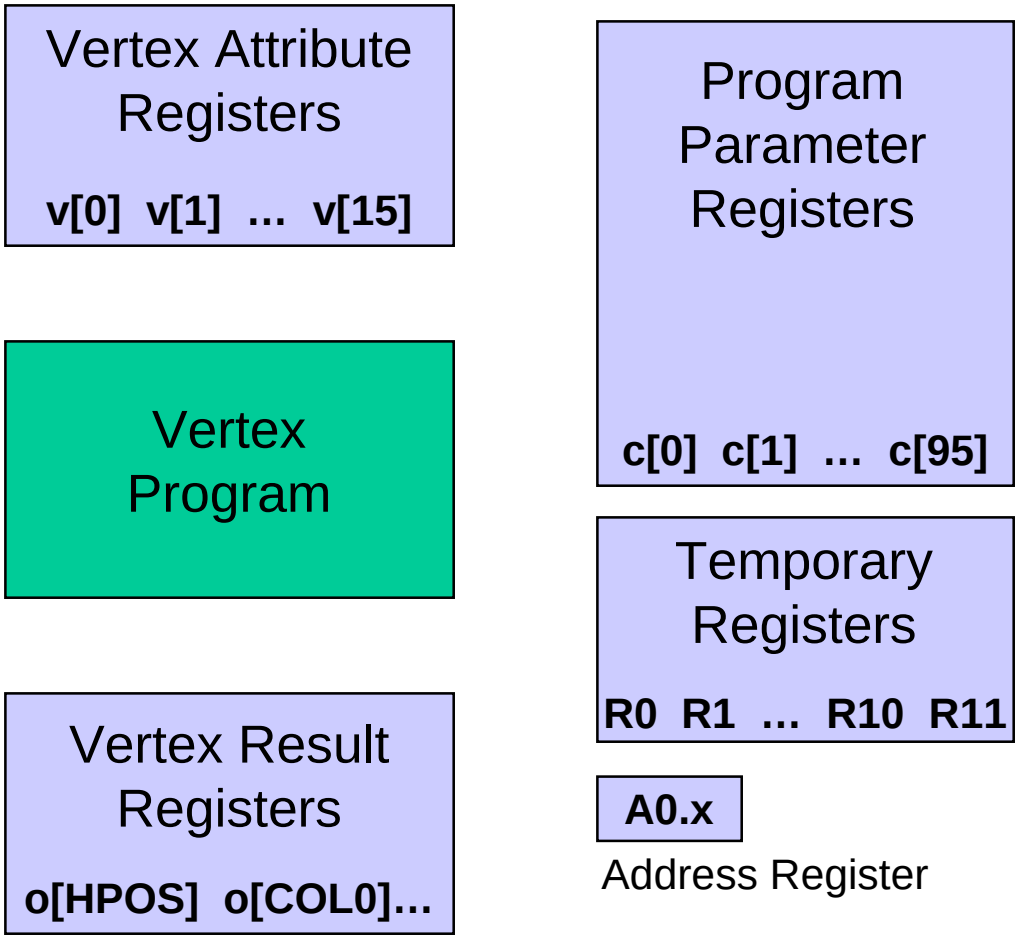
Vertex Programming
Program Parameters

- 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



The Register Set: Vertex Attribute Registers

Attribute Register	Mnemonic Name	Typical Meaning
v[0]	v[OPOS]	object position
v[1]	v[WGHT]	vertex weight
v[2]	v[NRML]	normal
v[3]	v[COL0]	primary color
v[4]	v[COL1]	secondary color
v[5]	v[FOGC]	fog coordinate
v[6]	-	-
v[7]	-	-
v[8]	v[TEX0]	texture coordinate 0
v[9]	v[TEX1]	texture coordinate 1
v[10]	v[TEX2]	texture coordinate 2
v[11]	v[TEX3]	texture coordinate 3
v[12]	v[TEX4]	texture coordinate 4
v[13]	v[TEX5]	texture coordinate 5
v[14]	v[TEX6]	texture coordinate 6
v[15]	v[TEX7]	texture coordinate 7

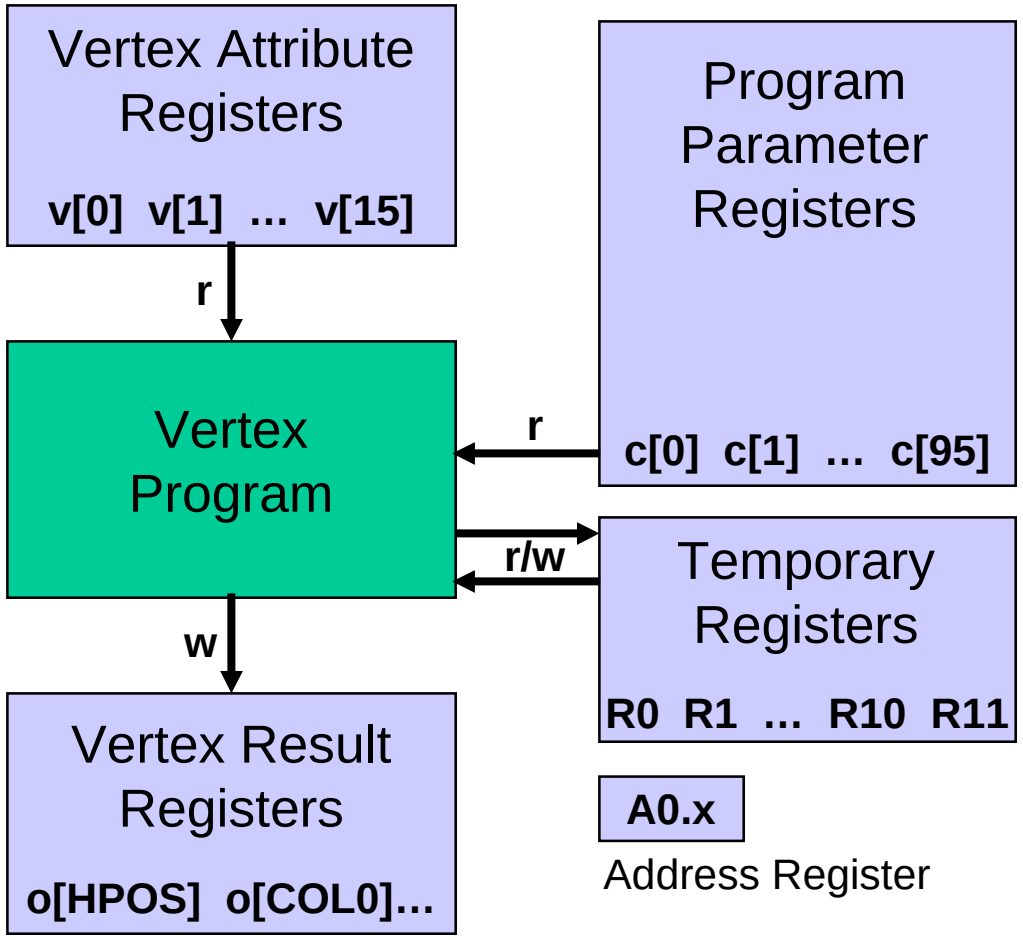
Semantics defined by program NOT parameter name!

Vertex Programming
Vertex Result Registers

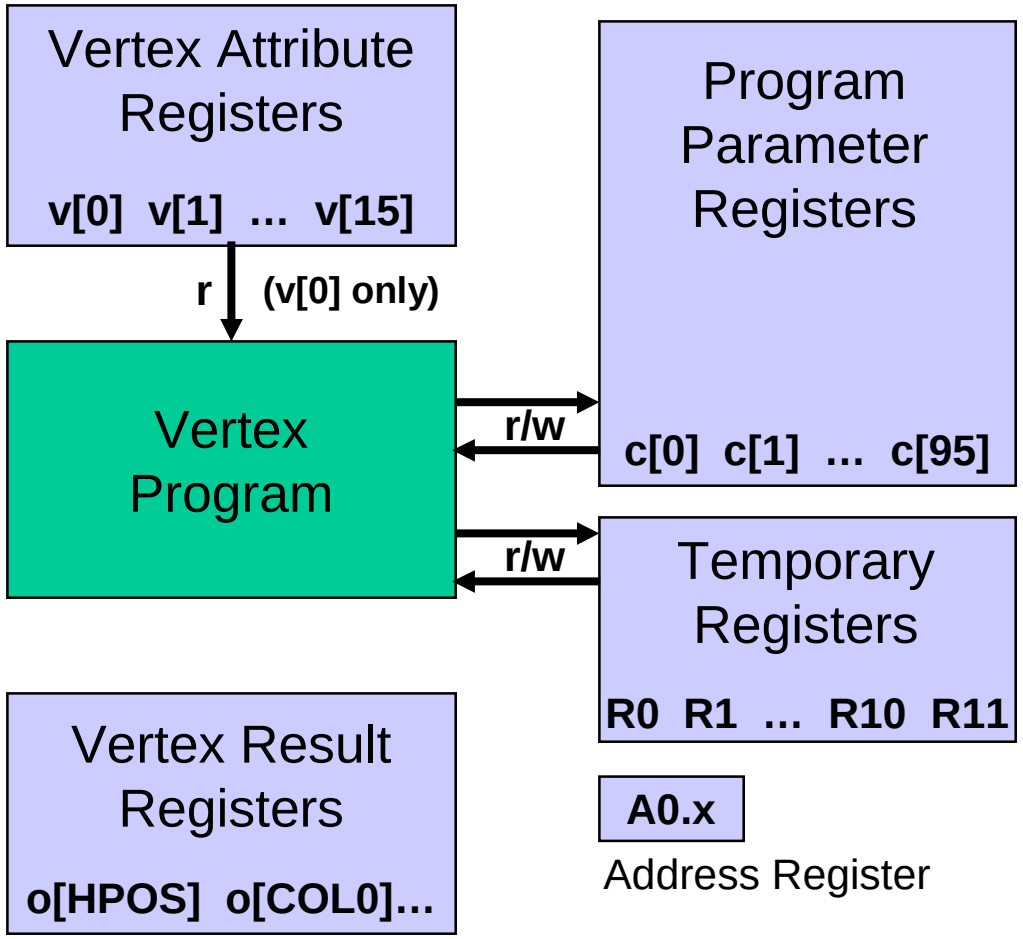
Register Name	Description	Component Interpretation
o[HPOS]	Homogeneous clip space position	(x,y,z,w)
o[COL0]	Primary color (front-facing)	(r,g,b,a)
o[COL1]	Secondary color (front-facing)	(r,g,b,a)
o[BFC0]	Back-facing primary color	(r,g,b,a)
o[BFC1]	Back-facing secondary color	(r,g,b,a)
o[FOGC]	Fog coordinate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

Semantics defined by down-stream pipeline stages.

Vertex Program Register Access



Vertex State Program Register Access



VSPs used to modify program parameter state.

Demo

- **Spline Evaluation…**

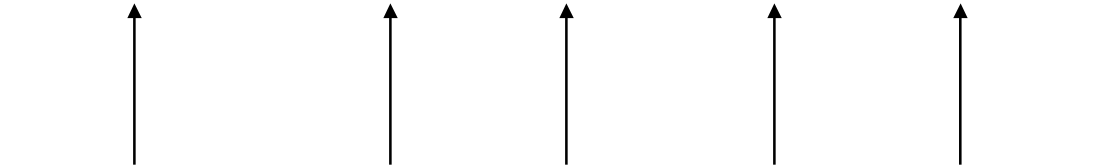
Vertex Programming Assembly Language

- 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:

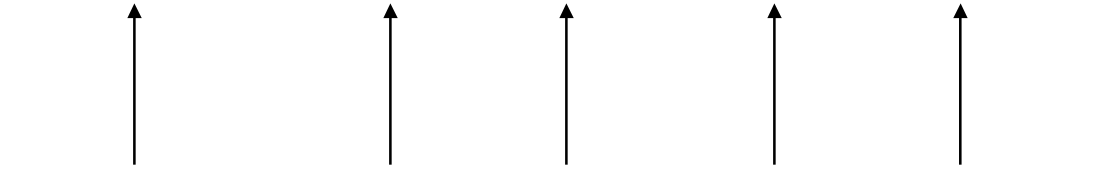
Opcode dst, [-]s0 [,[-]s1 [,[-]s2]]; #comment



Vertex Programming Assembly Language

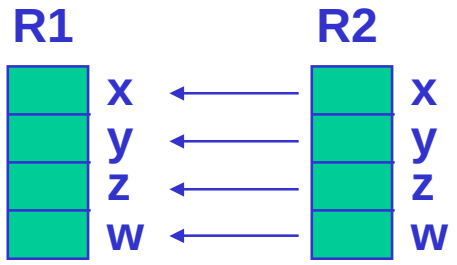
Instruction Format:

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



Example:

MOV r1, r2



Vertex Programming Assembly Language

Simple Example:

```
MOV     R1, R2;
```

before

R1		R2	
0.0	x	7.0	x
0.0	y	3.0	y
0.0	z	6.0	z
0.0	w	2.0	w

after

R1		R2	
7.0	x	7.0	x
3.0	y	3.0	y
6.0	z	6.0	z
2.0	w	2.0	w

Vertex Programming Assembly Language

Source registers undergo an input mapping before operation occurs…

- **Negation**
- **Swizzling**
- **Smearing**

Vertex Programming Assembly Language

Source registers can be negated:

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

before

R1		R2	
0.0	x	7.0	x
0.0	y	3.0	y
0.0	z	6.0	z
0.0	w	2.0	w

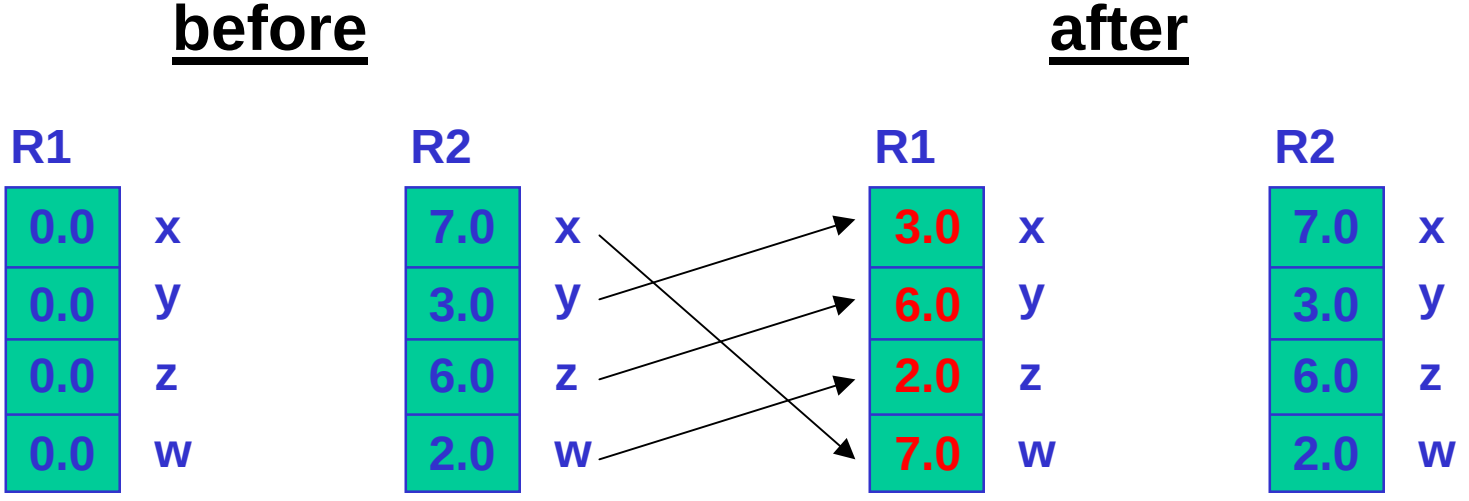
after

R1		R2	
-7.0	x	7.0	x
-3.0	y	3.0	y
-6.0	z	6.0	z
-2.0	w	2.0	w

Vertex Programming Assembly Language

Source registers can be “swizzled":

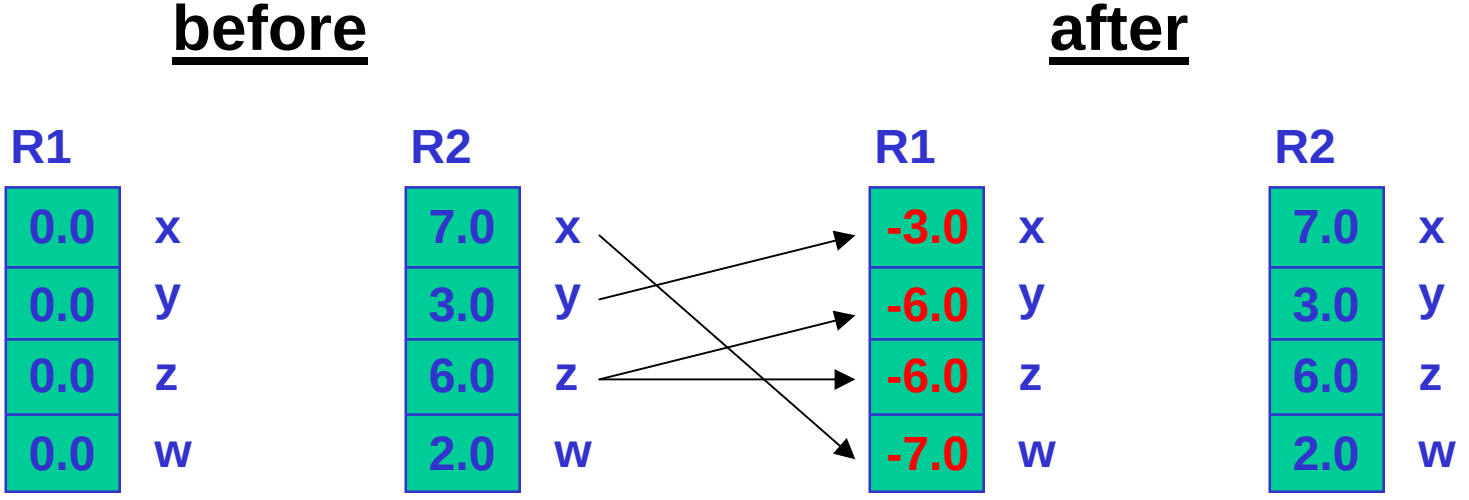
```
MOV     R1, R2.yzwx;
```



Vertex Programming Assembly Language

Source registers can be negated and “swizzled":

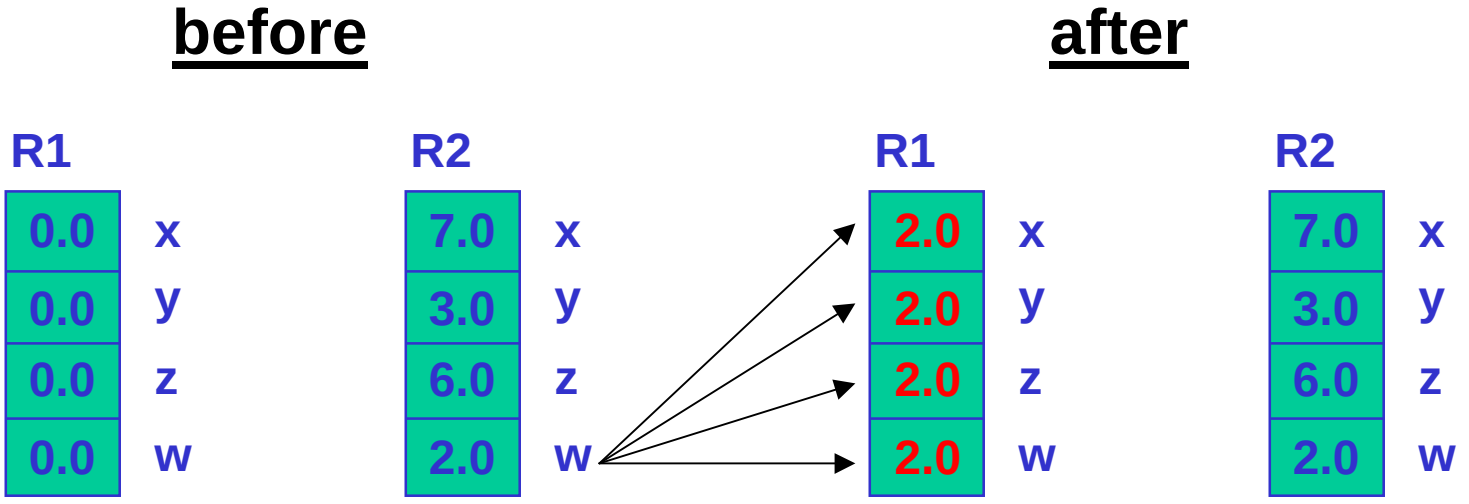
```
MOV    R1, -R2.yzzx;
```



Vertex Programming Assembly Language

Source registers can be swizzled by “smearing":

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



Vertex Programming Assembly Language

Destination register can mask which components are written to…

R1	⇒	**write all components**
R1.x	⇒	**write only x component**
R1.xw	⇒	**write only x, w components**

Vertex Programming Assembly Language

Destination register masking:

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

before

R1		R2	
0.0	x	7.0	x
0.0	y	3.0	y
0.0	z	6.0	z
0.0	w	2.0	w

after

R1		R2	
-7.0	x	7.0	x
0.0	y	3.0	y
0.0	z	6.0	z
-2.0	w	2.0	w

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:

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

before

	R1	R2	R3
x	0.0	7.0	2.0
y	0.0	3.0	2.1
z	0.0	6.0	5.0
w	0.0	2.0	7.0

after

	R1	R2	R3
x	14.0	7.0	2.0
y	6.3	3.0	2.1
z	30.0	6.0	5.0
w	0.0	2.0	7.0

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;
```

before

	R1	R2	R3
x	0.0	7.0	2.0
y	0.0	3.0	2.1
z	0.0	6.0	5.0
w	0.0	2.0	7.0

after

	R1	R2	R3
x	5.0	7.0	2.0
y	0.9	3.0	2.1
z	1.0	6.0	5.0
w	-5.0	2.0	7.0

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;
```

before

	R1	R2	R3	R4
x	0.0	7.0	2.0	1.0
y	0.0	3.0	2.1	3.0
z	0.0	6.0	5.0	2.0
w	0.0	2.0	7.0	1.0

after

	R1	R2	R3	R4
x	15.0	7.0	2.0	1.0
y	9.3	3.0	2.1	3.0
z	32.0	6.0	5.0	2.0
w	0.0	2.0	7.0	1.0

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:

```
RCP     R1, R2.w;
```

before

	R1	R2
x	0.0	7.0
y	0.0	3.0
z	0.0	6.0
w	0.0	2.0

after

	R1	R2
x	0.5	7.0
y	0.5	3.0
z	0.5	6.0
w	0.5	2.0

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;
```

before

	R1	R5
x	0.0	-4.0
y	0.0	3.0
z	0.0	7.0
w	0.0	9.0

after

	R1	R5
x	0.5	-4.0
y	0.0	3.0
z	0.0	7.0
w	0.0	9.0

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;
```

before

	R1	R6
x	0.0	3.0
y	0.0	2.0
z	0.0	1.0
w	0.0	1.0

after

	R1	R6
x	14.0	3.0
y	14.0	2.0
z	14.0	1.0
w	14.0	1.0

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;
```

The Instruction Set

DP4 Example:

```
DP4     R1, R6, R6;
```

before

	R1	R6
x	0.0	3.0
y	0.0	2.0
z	0.0	1.0
w	0.0	1.0

after

	R1	R6
x	15.0	3.0
y	15.0	2.0
z	15.0	1.0
w	15.0	1.0

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;
```

before

	R1	R2	R3
x	0.0	7.0	2.0
y	0.0	3.0	2.1
z	0.0	6.0	5.0
w	0.0	2.0	7.0

after

	R1	R2	R3
x	2.0	7.0	2.0
y	2.1	3.0	2.1
z	5.0	6.0	5.0
w	2.0	2.0	7.0

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;
```

before

	R1	R2	R3
x	0.0	7.0	2.0
y	0.0	3.0	2.1
z	0.0	6.0	5.0
w	0.0	2.0	7.0

after

	R1	R2	R3
x	7.0	7.0	2.0
y	3.0	3.0	2.1
z	6.0	6.0	5.0
w	7.0	2.0	7.0

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;
```

before

	R1	R2	R3
x	0.0	7.0	2.0
y	0.0	3.0	2.1
z	0.0	6.0	5.0
w	0.0	2.0	7.0

after

	R1	R2	R3
x	0.0	7.0	2.0
y	0.0	3.0	2.1
z	0.0	6.0	5.0
w	1.0	2.0	7.0

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;
```

before

	R1	R2	R3
x	0.0	7.0	2.0
y	0.0	3.0	2.1
z	0.0	6.0	5.0
w	0.0	2.0	7.0

after

	R1	R2	R3
x	1.0	7.0	2.0
y	1.0	3.0	2.1
z	1.0	6.0	5.0
w	0.0	2.0	7.0

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

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

The Instruction Set

EXP Example:

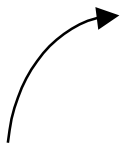
```
EXP     R1, R3.y;
```

before

	R1	R3
x	0.0	2.0
y	0.0	2.1
z	0.0	5.0
w	0.0	7.0

after

	R1	R3
x	4.0	2.0
y	0.1	2.1
z	4.287..	5.0
w	1.0	7.0



(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:

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

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

The Instruction Set

LOG Example:

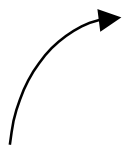
```
LOG     R1, R3.y;
```

before

	R1	R3
x	0.0	2.0
y	0.0	2.1
z	0.0	5.0
w	0.0	7.0

after

	R1	R3
x	1.0	2.0
y	1.05	2.1
z	1.07...	5.0
w	1.0	7.0



(Good to 11 bits)

The Instruction Set

EXP and LOG – Increasing the precision

- **EXP approximated by:**

  $EXP(s) = 2^{floor(s)} \times APPX(s-floor(s))$

  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)

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:

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

before

	A0	R8	R9	c[7]
x	0	2.0	0.0	17.0
y	0	5.9	0.0	22.0
z	0	4.2	0.0	3.0
w	0	7.0	0.0	3.0

after

	A0	R8	R9	c[7]
x	5	2.0	17.0	17.0
y	0	5.9	22.0	22.0
z	0	4.2	3.0	3.0
w	0	7.0	3.0	3.0

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)**

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

The Instruction Set

LIT: Light Coefficients

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

if (src0.x > 0.0)

$$dest.z = (MAX(src0.y,0))^{(ECLAMP(src0.w,-128,128))}$$

$$= (MAX(N \bullet H,0))^{m}$$ where m in (-128,128)

otherwise,

dest.z = 0.0

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

The Instruction Set

LIT Example:

```
LIT     R1, R7;
```

before

	R1	R7
x	0.0	0.3
y	0.0	0.8
z	0.0	0.0
w	0.0	2.0

after

		R1	R7
(ambient)	x	1.0	0.3
(diffuse)	y	0.3	0.8
(specular)	z	0.64	0.0
	w	1.0	2.0

(Good to 8+ bits)

The Instruction Set

DST: Distance Vector

Function:

Efficiently computes a "distance attenuation" vector (1, d, d^2, 1/d) from two source scalars.

Assumes:

$src0.C_1 = d^2$ **(where "c_1" is x, y, z, or w)**

$src1.C_2 = 1.0/d$ **(where "c_2" is x, y, z, or w)**

"d" is some distance

d = | light pos. – vertex pos. |

d = | eye pos. – vertex pos. |

The Instruction Set

DST: Distance Vector

Syntax:

DST dest, src0.C_1, src1.C_2

Result:

dest.x = 1

dest.y = src0.C_1 * src1.C_2

= d

dest.z = src0.C_1

= d^2

dest.w = src1.C_2

= 1/d

The Instruction Set

DST: Utility exemplified through an example…

Lighting example with distance attenuation:

modulate by $1 / (k_0 + k_1 d + k_2 d^2)$
where d = | light pos. – vertex pos. |

Suppose vector R5 = light pos. – vertex pos.
= unnormalized light vector (L)

Likely need to normalize L for N • L computation.

The Instruction Set

DST: Distance attenuation example…

Normalize L by:

```
DP3   R0.w, R5, R5;          # R0.w is d²
RSQ   R1.w, R0.w;            # R1.w is 1/d
MUL   R5.xyz, R5, R1.w;      # R5 is normalized
```

Now get attenuation vector:

```
DST   R6, R0.w, R1.w;        # R6 is (1,d,d²,1/d)
```

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

```
DP3   R7.w, R6, c[0];     # R7.w is k0+k1d+k2d^2
RCP   R1.w, R0.w;         # R1.w is attenuation
```

Same task would require additional instructions w/o DST.

The Instruction Set

DST Example:

```
DST     R1, R2.w, R3.w;
```

before

	R1	R2	R3
x	0.0	7.0	2.0
y	0.0	3.0	2.1
z	0.0	6.0	5.0
w	0.0	4.0	0.5

after

	R1	R2	R3
x	1.0	7.0	2.0
y	2.0	3.0	2.1
z	4.0	6.0	5.0
w	0.5	4.0	0.5

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;
```

Example Programs

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”.**

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:**

  http://www.nvidia.com/developer

Questions, comments, feedback

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