Low-Level Vertex Programming

Martin Kraus

Institute of Visualization and Interactive Systems
University of Stuttgart

Overview

in this talk:

- **What Are Low-Level APIs?**
- **Low-Level Vertex Programming**

after coffee break:

- Low-Level Pixel Programming

What Are Low-Level APIs?

- similarity to assembler:
  - close to hardware functionality
  - input: vertex/fragment attributes
  - output: new vertex/fragment attributes
  - sequence of instructions on registers
  - very limited control flow (if any)
  - platform-dependent
    BUT there is convergence

What Are Low-Level APIs?

- last year's low-level APIs:
  - OpenGL extensions:
    GL_NV_vertex_program(1_1),
    GL_NV_texture_shader(2,3),
    GL_NV_register_combiners(2),
    GL_NV_fragment_program,
    GL_EXT_vertex_shader,
    GL_ATI_fragment_shader,
    GL_ATI_text_fragment_shader
  - DirectX 8.0 and 8.1:
    Vertex Shader 1.0, 1.1,
    Pixel Shader 1.0, 1.1, 1.2, 1.3, 1.4

What Are Low-Level APIs?

- this year's low-level APIs:
  - OpenGL extensions:
    GL_ARB_vertex_program,
    GL_ARB_fragment_program
  - DirectX 9:
    Vertex Shader 2.0,
    Pixel Shader 2.0

What Are Low-Level APIs?

- last year's and this year's reasons to use them:
  - low-level APIs offer best performance & functionality
  - help to understand the graphics hardware
    (ATI's r300, NVIDIA's nv30, ...)
  - help to understand high-level APIs
    (Cg, HLSL, ...)
- for good reasons not to use low-level APIs:
  see talk on high-level APIs

Overview

- What Are Low-Level APIs?
- **Low-Level Vertex Programming**
  - **Applications**
  - **OpenGL Extension: GL_ARB_vertex_program**
  - **DirectX 9: Vertex Shader 2.0**
- Low-Level Pixel Programming

Applications of Vertex Programming

- customized computation of vertex attributes
- computation of anything that can be interpolated linearly between vertices
- more specific:
  - transformations of position (vertex blending, vertex skinning, displacement, ...)
  - normal computations (procedural bump mapping, ...)
  - color computations (lighting, depth cueing, cool/warm shading, ...)
  - texture coordinate generation (reflection mapping, environment mapping, ...)

Applications of Vertex Programming

- limitations:
  - vertices can neither be generated nor destroyed (but degenerate primitives are not rasterized)
  - no information about topology or ordering of vertices is available

OpenGL Ext.: GL_ARB_vertex_program

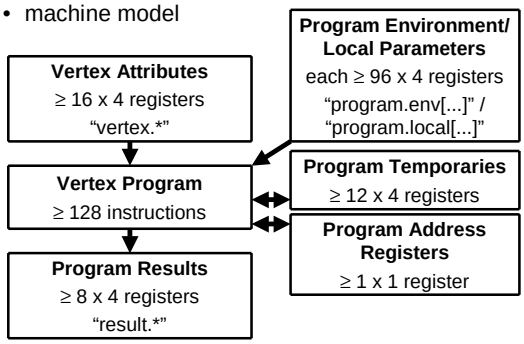
- circumvents the traditional vertex pipeline
- what is replaced by a vertex program?
  - vertex transformations
  - vertex weighting/blending
  - normal transformations
  - color material
  - per-vertex lighting
  - texture coordinate generation
  - texture matrix transformations
  - per-vertex point size computations
  - per-vertex fog coordinate computations
  - client-defined clip planes

OpenGL Ext.: GL_ARB_vertex_program

- what is not replaced? (after a vertex program)
  - clipping to the view frustum
  - perspective divide (division by *w*)
  - viewport transformation
  - depth range transformation
  - front and back color selection
  - clamping colors
  - primitive assembly and per-fragment operations
  - evaluators

OpenGL Ext.: GL_ARB_vertex_program

- machine model



OpenGL Ext.: GL_ARB_vertex_program

- vertex attributes:
  - vertex.position
  - vertex.weight
  - vertex.weight[*n*]
  - vertex.normal
  - vertex.color
  - vertex.color.primary
  - vertex.color.secondary
  - vertex.fogcoord
  - vertex.texcoord
  - vertex.texcoord[*n*]
  - vertex.matrixindex
  - vertex.matrixindex[*n*]
  - vertex.attrib[*n*]
  - implicit binding: use “vertex.*” in instruction
  - explicit binding:

```
ATTRIB name = vertex.*;
```


OpenGL Ext.: GL_ARB_vertex_program

- generic vs. conventional attributes
  - vertex.attrib[0] vertex.position
  - vertex.attrib[1] vertex.weight, vertex.weight[0]
  - vertex.attrib[2] vertex.normal
  - vertex.attrib[3] vertex.color, vertex.color.primary
  - vertex.attrib[4] vertex.color.secondary
  - vertex.attrib[5] vertex.fogcolor
  - vertex.attrib[8] vertex.texcoord
  - vertex.attrib[8+*n*] vertex.texcoord[*n*]
  - set with **`glColor`**, **`glNormal`**, ... or **`glVertexAttrib`**
  - setting position executes vertex program

OpenGL Ext.: GL_ARB_vertex_program

- program environment/local parameters
  - environment parameters: for all vertex programs
    - program.env[*index*]
    - program.env[*index1..index2*]
  - local parameters: for one vertex program
    - program.local[*index*]
    - program.local[*index1..index2*]
  - implicit binding: use “program.env[*index*]”, ...
  - explicit binding:

```
PARAM name = program.env[index];
PARAM name[size] = program.env[index1..index2];
```


OpenGL Ext.: GL_ARB_vertex_program

- constants:
  - implicit binding: use literal numbers in instructions
  - explicit binding:

```
PARAM name = number;
PARAM name = {number, number, number, number};
PARAM name[size] = {{number, ...}, ...};
```

- state variables:
  - implicit binding: use “state.*” in instructions
  - explicit binding:

```
PARAM name = state.*;
```


OpenGL Ext.: GL_ARB_vertex_program

- incomplete list of state variables:
  - state.material.* (ambient, diffuse, specular, ...)
  - state.light[*n*].* (ambient, diffuse, position, ...)
  - state.lightmodel.* (ambient, scenecolor, ...)
  - state.lightprod[*n*].* (ambient, diffuse, ...)
  - state.texgen[*n*].* (eye.s, eye.t, eye.r, eye.q, object.s,...)
  - state.fog.* (color, params)
  - state.clip[*n*].plane
  - state.point.* (size, attenuation)
  - state.matrix.* (modelview[*n*], projection, mvp, texture[*n*], palette[*n*], program[*n*])

OpenGL Ext.: GL_ARB_vertex_program

- matrix modifiers:
  - *name*.inverse
  - *name*.transpose
  - *name*.invtrans
  - *name*.row[*index*] ($0 \le index \le 3$)
  - *name*.row[*index1..index2*] ($0 \le index1 \le index2 \le 3$)
  - examples:

```
PARAM mm[ ] = { state.matrix.program[0].transpose };
PARAM m[ ] = { state.matrix.program[0].row[1..2] };
```

OpenGL Ext.: GL_ARB_vertex_program

- program temporaries
  - at least 12 four-component vectors
  - declare before use:

```
TEMP name;
```

- address register
  - at least 1 four-component vector
  - declare before use:

```
ADDRESS name;
```

  - use to access elements of arrays:

```
array_name[name.x + n]
```


OpenGL Ext.: GL_ARB_vertex_program

- program results and output variables
  - write-only registers
  - implicit binding: use “result.*” in instructions
  - explicit binding to output variables:

```
OUTPUT name = result.*;
```

  - program results:
    - result.position
    - result.color.* (primary, secondary, front.primary, front.secondary, back.primary, back.secondary)
    - result.fogcoord
    - result.pointsize
    - result.texcoord
    - result.texcoord[n]

OpenGL Ext.: GL_ARB_vertex_program

- aliases
  - declare before use:

```
ALIAS new_name = old_name;
```

  - just a reference

OpenGL Ext.: GL_ARB_vertex_program

- instruction set:
  - 27 instructions
  - operate on floating-point scalars or 4-vectors
  - basic syntax:

```
OP destination [, source1 [, source2 [, source3]]]; # comm.
```

  - example:

```
MOV result.position, vertex.position;  # sets result.position
```


OpenGL Ext.: GL_ARB_vertex_program

- modifiers
  - all components of sources may be negated

```
-source
```

  - components of sources may be swizzled, e.g.

```
source.yxzw
```

    exchanges x and y component for this operation
  - components of destination may be masked, e.g.

```
destination.zw
```

    writes only z and w component

OpenGL Ext.: GL_ARB_vertex_program

- list of instructions with scalar argument(s)

EX2 ssss, s	exponential base 2
EXP v, s	exponential base 2 (approximate)
LG2 ssss, s	logarithm base 2
LOG v, s	logarithm base 2 (approximate)
POW ssss, s, s	exponentiate
RCP ssss, s	reciprocal
RSQ ssss, s	reciprocal square root

s: scalar, ssss: replicated scalar, v: vector

OpenGL Ext.: GL_ARB_vertex_program

- list of instructions with one vector source

ABS v, v	absolute value
ARL a, v	address register load
FLR v, v	floor
FRC v, v	fraction
LIT v, v	compute light coefficients
MOV v, v	move
SWZ v, v	extended swizzle

a: address, v: vector

OpenGL Ext.: GL_ARB_vertex_program

- list of instructions with multiple vector sources

ADD v, v, v	add
DP3 ssss, v, v	3-component dot product
DP4 ssss, v, v	4-component dot product
DPH ssss, v, v	homogeneous dot product
DST v, v, v	distance vector
MAD v, v, v, v	multiply and add
MAX v, v, v	maximum
MIN v, v, v	minimum
MUL v, v, v	multiply
SGE v, v, v	set on greater than or equal
SLT v, v, v	set on lower than
SUB v, v, v	subtract
XPD v, v, v	cross product

OpenGL Ext.: GL_ARB_vertex_program

- simple example:

```
!!ARBvp1.0

MOV result.position, vertex.position;
MOV result.color, vertex.color;

END
```


OpenGL Ext.: GL_ARB_vertex_program

- transformation to clip coordinates:

```
!!ARBvp1.0
ATTRIB pos = vertex.position;
ATTRIB col = vertex.color;
OUTPUT clippos = result.position;
OUTPUT newcol = result.color;
PARAM modelviewproj[4] = { state.matrix.mvp };
DP4 clippos.x, modelviewproj[0], pos;
DP4 clippos.y, modelviewproj[1], pos;
DP4 clippos.z, modelviewproj[2], pos;
DP4 clippos.w, modelviewproj[3], pos;
MOV newcol, col;
END
```


OpenGL Ext.: GL_ARB_vertex_program

- simple lighting:

```
!!ARBvp1.0
ATTRIB iPos = vertex.position;
ATTRIB iNormal = vertex.normal;
PARAM mvinv[4] = { state.matrix.modelview.invtrans };
PARAM mvp[4] = { state.matrix.mvp };
PARAM lightDir = state.light[0].position;
PARAM halfDir = state.light[0].half;
PARAM specExp = state.material;
PARAM ambientCol = state.lightprod[0].ambient;
PARAM diffuseCol = state.lightprod[0].diffuse;
PARAM specularCol = state.lightprod[0].specular;

# continued on next page
```


OpenGL Ext.: GL_ARB_vertex_program

```
# simple lighting continued

OUTPUT oPos = result.position;
DP4 oPos.x, mvp[0], iPos;
DP4 oPos.y, mvp[1], iPos;
DP4 oPos.z, mvp[2], iPos;
DP4 oPos.w, mvp[3], iPos;

TEMP xfNormal;
DP3 xfNormal.x, mvinv[0], iNormal;
DP3 xfNormal.y, mvinv[1], iNormal;
DP3 xfNormal.z, mvinv[2], iNormal;

# continued on next page
```

OpenGL Ext.: GL_ARB_vertex_program

```
# simple lighting continued

TEMP dots;
DP3 dots.x, xfNormal, lightDir;
DP3 dots.y, xfNormal, halfDir;
MOV dots.w, specExp.x;
LIT dots, dots;

TEMP temp;
OUTPUT oColor = result.color;
MAD temp, dots.y, diffuseCol, ambientCol;
MAD oColor.xyz, dots.z, specularCol, temp;
MOV oColor.w, diffuseCol.w;
END
```


OpenGL Ext.: GL_ARB_vertex_program

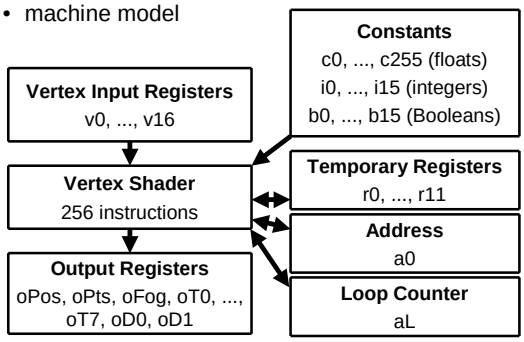
- outlook:
  - OPTION mechanism for extensions (currently only OPTION ARB_position_invariant)
  - next version of assembly language with flow control (!!ARBvp2.0)

DirectX 9: Vertex Shader 2.0

- Vertex Shader 2.0 introduced in DirectX 9.0
- similar functionality and limitations as GL_ARB_vertex_program
- additional functionality: static flow control
- similar registers and syntax

DirectX 9: Vertex Shader 2.0

- machine model



DirectX 9: Vertex Shader 2.0

- declaration of usage of vertex input registers:

```
dcl_u v*
```

- examples:

```
dcl_position v0
dcl_position1 v1
dcl_position2 v2
dcl_color v3
dcl_normal v4
dcl_texcoord v5
```


DirectX 9: Vertex Shader 2.0

- definition of float constants:

```
def c*, number, number, number, number
```

- definition of integer constants:

```
defi i*, number, number, number, number
```

- definition of Boolean constants:

```
defb b*, true
```

```
defb b*, false
```

DirectX 9: Vertex Shader 2.0

- instruction set:
  - instructions (lower case) and macros (upper case)
  - operate on floating-point scalars or 4-vectors
  - basic syntax:

    op destination [, *source1* [, *source2* [, *source3*]]] *//comment*

  - example:

```
mov oPos, v0        // sets resulting position
```

  - same modifiers (negate, full swizzle, write masks) as in vertex programs
  - no operation: “nop”

DirectX 9: Vertex Shader 2.0

- list of instructions with scalar argument(s):

exp ssss, s	full precision 2 power X
expp ssss, s	partial precision 2 power X
log ssss, s	full precision base-2 logarithm of X
logp ssss, s	partial precision base-2 logarithm of X
POW ssss, s, s	exponentiate
rcp ssss, s	reciprocal
RSQ ssss, s	reciprocal square root

s: scalar, ssss: replicated scalar, v: vector

DirectX 9: Vertex Shader 2.0

- list of instructions with one vector source:

ABS v, v	absolute value
frc v, v	fraction
lit v, v	partial lighting calculation
mov v, v	move floating point data between registers
mova v, v	move data from floating point to integer register

v: vector

DirectX 9: Vertex Shader 2.0

- list of instructions with multiple vector sources:

add v, v, v	add
CRS v, v, v	cross product macro
dp3 ssss, v, v	3-component dot product
dp4 ssss, v, v	4-component dot product
dst v, v, v	distance vector
LRP v, v, v, v	linear interpolation
mad v, v, v, v	multiply and add
max v, v, v	maximum
min v, v, v	minimum
mul v, v, v	multiply
sge v, v, v	set on greater than or equal
SGN v, v, v, v	compute sign
SINCOS v, s, v, v	sine and cosine
slt v, v, v	compute sign if less

DirectX 9: Vertex Shader 2.0

- more vector macros:

M4x4 v, v, v	four dot products of 4-component vectors
M4x3 v, v, v	three dot products of 4-component vectors
M3x4 v, v, v	four dot products of 3-component vectors
M3x3 v, v, v	three dot products of 3-component vectors
M3x2 v, v, v	two dot products of 3-component vectors
NRM v, v	normalize

DirectX 9: Vertex Shader 2.0

- static flow control:
  - control of flow determined by constants (not by per-vertex attributes!)
  - conditional blocks (if ... else endif)
  - repetition (loop ... endloop, rep ... endrep)
  - subroutines (call, callnz, label, ret)

DirectX 9: Vertex Shader 2.0

- simple example

```
vs_2_0

dcl_position v0
dcl_color v1

mov oPos, v0
mov oD0, v1
```


DirectX 9: Vertex Shader 2.0

- outlook: Vertex Shader 2.x
  - dynamic flow control (ifc, break, breakc)
  - more temporary registers
  - deeper static flow control nesting
  - predication (conditional execution of instructions)
- outlook: Vertex Shader 3.0
  - all of Vertex Shader 2.x
  - indexing registers (not only c* but also v*, oT*)
  - vertex textures
  - vertex stream frequency divider

Overview

in this talk:

- What Are Low-Level APIs?
- Low-Level Vertex Programming
  - Applications
  - OpenGL Extension: GL_ARB_vertex_program
  - DirectX 9: Vertex Shader 2.0

after coffee break:

- **Low-Level Pixel Programming**