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, ...)

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 \( w \))
  - viewpoint transformation
  - depth range transformation
  - front and back color selection
  - clamping colors
  - primitive assembly and per-fragment operations
  - evaluators

Machine Model

- Processor
  - 16 x 4 vertex attributes registers
  - 128 instructions vertex program
  - 8 x 4 registers program results

Program Environment/Local Parameters

- 96 x 4 registers for program.env[...]
- 96 x 4 registers for program.local[...]

Program Temporaries

- 12 x 4 registers

Program Address Registers

- 1 x 1 register
**OpenGL Ext.: GL_ARB_vertex_program**

- **vertex attributes:**
  - `vertex.position`
  - `vertex.fogcoord`
  - `vertex.weight[n]`
  - `vertex.texcoord[n]`
  - `vertex.normal`
  - `vertex.matrixindex`
  - `vertex.color`
  - `vertex.matrixindex[n]`
  - `vertex.attrib[n]`
  - `vertex.color.primary`
  - `vertex.color.secondary`

- **vertex attributes with implicit binding:**
  - `vertex.*`

- **vertex attributes with explicit binding:**
  - `ATTRIB name = vertex.*;`

- **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]`

- **context parameters:**
  - **implicit binding:**
    - `program.env[index]`, ...
  - **explicit binding:**
    - `PARAM name = program.env[index];`
    - `PARAM name[size] = program.env[index1..index2];`

- **constants:**
  - **implicit binding:**
    - `literal numbers` in instructions
  - **explicit binding:**
    - `PARAM name = number;`
    - `PARAM name[size] = {number, ..., number};`

- **state variables:**
  - **implicit binding:**
    - `state.*`
  - **explicit binding:**
    - `PARAM name = state.*;`

- **matrix modifiers:**
  - `name.inverse`
  - `name.transpose`
  - `name.invtrans`
  - `name.row[index][0 ≤ index ≤ 3]`
  - `name.row[index1..index2][0 ≤ index1 ≤ index2 ≤ 3]`

- **state variables with explicit binding:**
  - `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
    ]]

  - 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.yzw

    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
  - `FRC v, v` fraction
  - `LIT v, v` compute light coefficients
  - `MOV v, v` move
  - `SWZ v, v` extended swizzle

  \(a\): address, \(v\): vector

- 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` 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

---

**Example:**

```plaintext
!!ARBvp1.0
MOV result.position, vertex.position;
MOV result.color, vertex.color;
END
```

**Transformation to clip coordinates:**

```plaintext
!!ARBvp1.0
ATTRIB pos = vertex.position;
ATTRIB col = vertex.color;
OUTPUT clippos = result.position;
OUTPUT newcol = result.color;
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
```

---

**Simple lighting:**

```plaintext
!!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;
PARAM ambientCol = state.lightprod[0].ambient;
PARAM diffuseCol = state.lightprod[0].diffuse;
PARAM specularCol = state.lightprod[0].specular;

# continued on next page
```

# Simple lighting continued

```plaintext
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

```gl
TEMP dots;
DP3 dots.x, xNormal, lightDir;
DP3 dots.y, xNormal, 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
```

### 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

#### Machine Model

- **Vertex Input Registers**
  - v0, ..., v16

- **Vertex Shader**
  - 256 instructions

- **Temporary Registers**
  - r0, ..., r11

- **Output Registers**
  - oPos, oPts, oFog, oT0, ..., oT7, oD0, oD1

- **Loop Counter**
  - aL

#### Declaration of Usage of Vertex Input Registers:

```gl
dcl_vuv
```

#### Examples:

```gl
dcl_position v0
dcl_position1 v1
dcl_position2 v2
dcl_color v3
dcl_texcoord v6
```
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:
  \[ \text{op destination} [, \text{source1} [, \text{source2} [, \text{source3}] ] \] //comment
- example:
  \[ \text{mov oPos, v0} \] // sets resulting position
- some modifiers (negate, full swizzle, write masks) as in vertex programs
- no operation: “nop”

- list of instructions with scalar argument(s):
  - exp ssss, s full precision 2 power X
  - exp ssss, s partial precision 2 power X
  - log ssss, s full precision base-2 logarithm of X
  - logf ssss, s partial precision base-2 logarithm of X
  - POWe ssss, s exponentiate
  - rcp ssss, s reciprocal
  - RSQ ssss, s reciprocal square root
  - s: scalar, ssss: replicated scalar, v: vector

- 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
  - mov a v, v move data from floating point to integer register
  - v: vector

- 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 distance vector
  - LRP 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, s sine and cosine
  - slt v, v, v compute sign if less

- 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

- 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