Low-Level Pixel Programming

Martin Kraus

Institute of Visualization and Interactive Systems
University of Stuttgart

Overview

before coffee break

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

in this talk

- **Low-Level Pixel Programming**
  - **Applications**
  - **OpenGL Extension: GL_ARB_fragment_program**
  - **DirectX 9: Pixel Shader 2.0**

Applications of Pixel Programming

- customized computation of fragment attributes
- computation of anything that should be computed per pixel
- more specific:
  - normal computations
    (per-pixel interpolation and normalization, bump mapping, ...)
  - color computations
    (per-pixel shading and lighting, ...)
  - texture mapping
    (per-pixel reflection and environment mapping, random memory access, render-to-texture, ...)

Applications of Pixel Programming

- limitations:
  - fragments cannot be generated
  - position of fragments cannot be changed
  - no information about geometric primitive is available

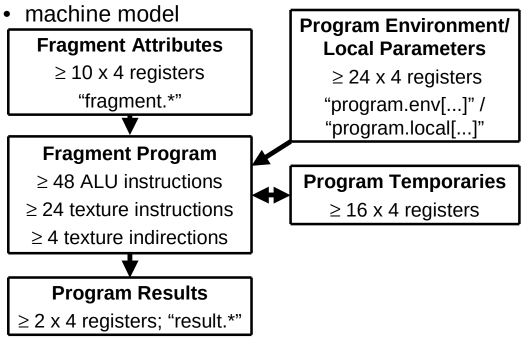
OpenGL Ext.: GL_ARB_fragment_program

- circumvents the traditional fragment pipeline
- what is replaced by a pixel program?
  - texturing
  - color sum
  - fog

  for the rasterization of points, lines, polygons, pixel rectangles, and bitmaps
- what is not replaced?
  - coverage application
  - fragment tests (alpha, stencil, and depth tests)
  - blending

OpenGL Ext.: GL_ARB_fragment_program

- machine model



OpenGL Ext.: GL_ARB_fragment_program

- fragment attributes:

  fragment.color
  fragment.color.primary
  fragment.color.secondary
  fragment.texcoord
  fragment.texcoord[*n*]
  fragment.fogcoord
  fragment.position

  – implicit binding: use "fragment.*" in instruction
  – explicit binding:

```
ATTRIB name = fragment.*;
```

OpenGL Ext.: GL_ARB_fragment_program

- program environment/local parameters
  – environment parameters: for all fragment programs

  program.env[*index*]
  program.env[*index1*..*index2*]

  – local parameters: for one fragment 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_fragment_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_fragment_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.texenv[*n*].color
  state.fog.* (color, params)
  state.depth.range
  state.matrix.* (modelview[*n*], projection, mvp, texture[*n*], palette[*n*], program[*n*])

OpenGL Ext.: GL_ARB_fragment_program

- matrix modifiers:

  name.inverse
  name.transpose
  name.invtrans
  name.row[*index*] (0 ≤ *index* ≤ 3)
  name.row[*index1*..*index2*] (0 ≤ *index1* ≤ *index2* ≤ 3)

  – examples:

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

OpenGL Ext.: GL_ARB_fragment_program

- program temporaries
  – at least 16 four-component vectors
  – declare before use:

```
TEMP name;
```

OpenGL Ext.: GL_ARB_fragment_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.color
    result.depth
- aliases
  - declare before use:

```
ALIAS new_name = old_name;
```

  - just a reference

OpenGL Ext.: GL_ARB_fragment_program

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

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

  - example:

```
MOV result.color, fragment.color;  # sets result.color
```

OpenGL Ext.: GL_ARB_fragment_program

- modifiers
  - all components of sources may be negated

```
-source
```

  - components of sources (x, y, z, w or r, g, b, a) 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
  - _SAT instruction suffix:
    clamping of resulting components to [0,1]

OpenGL Ext.: GL_ARB_fragment_program

- list of instructions with scalar argument(s)

COS ssss, s	cosine with reduction to [-pi, pi]
EX2 ssss, s	exponential base 2
LG2 ssss, s	logarithm base 2
POW ssss, s, s	exponentiate
RCP ssss, s	reciprocal
RSQ ssss, s	reciprocal square root
SCS ss--, s	sine/cosine without reduction
SIN sss, s	sine with reduction

s: scalar, ssss: replicated scalar

OpenGL Ext.: GL_ARB_fragment_program

- list of instructions with one vector source

ABS v, v	absolute value
FLR v, v	floor
FRC v, v	fraction
KIL v, v	kill fragment (counts as texture instruction)
LIT v, v	compute light coefficients
MOV v, v	move
SWZ v, v	extended swizzle

v: vector

OpenGL Ext.: GL_ARB_fragment_program

- list of instructions with multiple vector sources

ADD v, v, v	add
CMP v, v, v	compare
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
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
SLT v, v, v	set on lower than
SUB v, v, v	subtract
XPD v, v, v	cross product

OpenGL Ext.: GL_ARB_fragment_program

- texture sampling
  - syntax:

    OP *destination*, *source*, texture[*index*], *type*;
  - texture instructions (apart from KIL):

TEX	map coordinate to color (no division by q)
TXP	project coordinate and map to color
TXB	map coordinate to color while biasing its LOD

  - texture types: 1D, 2D, 3D, CUBE, RECT
  - example:

```
TEX result.color, fragment.texcoord[1],
texture[0], 2D;
```

    samples 2D texture in unit 0 with texture coordinate set 1 and writes result to result.color.

OpenGL Ext.: GL_ARB_fragment_program

- dependent texture sampling
  - at least 4 levels of indirection are allowed
  - dependent texture samples:
    1. the source coordinate is a temporary that has already been written or
    2. the result is a temporary that as already been written or read.

OpenGL Ext.: GL_ARB_fragment_program

- simple example:

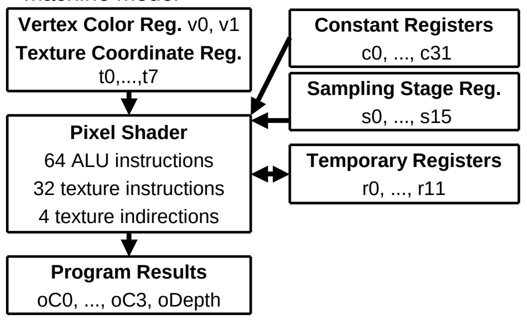
```
!!ARBfp1.0
ATTRIB tex = fragment.texcoord;
ATTRIB col = fragment.color.primary;
OUTPUT outColor = result.color;
TEMP tmp;

TXP tmp, tex, texture[0], 2D;
MUL outColor, tmp, col;
END
```

DirectX 9: Pixel Shader 2.0

- Pixel Shader 2.0 introduced in DirectX 9.0
- similar functionality and limitations as GL_ARB_fragment_program
- similar registers and syntax

DirectX 9: Pixel Shader 2.0

- machine model


Vertex Color Reg. v0, v1
Texture Coordinate Reg. t0,...,t7
Constant Registers c0, ..., c31
Sampling Stage Reg. s0, ..., s15
Pixel Shader
64 ALU instructions
32 texture instructions
4 texture indirections
Temporary Registers r0, ..., r11
Program Results oC0, ..., oC3, oDepth

DirectX 9: Pixel Shader 2.0

- declaration of texture samplers:

    dcl_*type* s*
- examples:

```
dcl_2d s0
dcl_cube s1
dcl_volume s2
```

- declaration of input color and texture coordinate:

```
dcl v*[.mask]
dvl t*[.mask]
```

- example:

```
dcl t0.xy
```

DirectX 9: Pixel Shader 2.0

- definition of constants:

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

- 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 oC0, v0;  // sets resulting color
```

  - “nop”: no operation

DirectX 9: Pixel Shader 2.0

- modifiers:
  - negate source with “-”
  - restricted swizzling (.rgba, .xyzw, .r, .rrrr, .x, .xxxx, .g, .gggg, .y, .yyyy, .b, .bbbb, .z, .zzzz, .a, .aaaa, .w, .wwww, .gbra, .brga, .abgr, .yzxw, .zxyw, .wzyx)
  - any (ordered) mask with r, g, b, a or x, y, z, w
  - “_sat”: clamps result to [0,1] (not with frc, SINCOS, texld*, texkill, o* registers)
  - _pp: partial precision hint

DirectX 9: Pixel Shader 2.0

- list of instructions and macros with scalar argument(s)

exp ssss, s	exponential base 2
log ssss, s	logarithm base 2
POW ssss, s, s	exponentiate
rcp ssss, s	reciprocal
rsq ssss, s	reciprocal square root
SINCOS v, s, v, v	sine, cosine

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

DirectX 9: Pixel Shader 2.0

- list of instructions and macros with one vector argument

ABS v, v	absolute value
frc v, v	fraction
mov v, v	move
texkill v	kill pixel (counts as texture instructions)

v: vector

DirectX 9: Pixel Shader 2.0

- list of instructions and macros with multiple vector sources

add v, v, v	add
CMP v, v, v, v	compare
CRS v, v, v	cross product
dp2add ssss,v,v,s	2-component dot product and add
dp3 ssss, v, v	3-component dot product
dp4 ssss, v, v	4-component dot product
LRP v, v, v, v	linear interpolation
MAX v, v, v	maximum
MIN v, v, v	minimum
MUL v, v, v	multiply

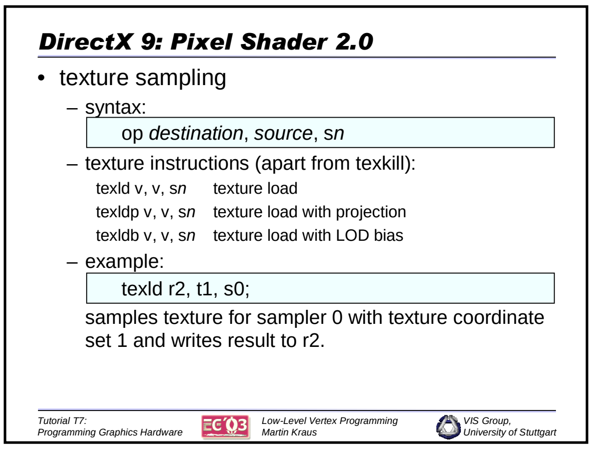
s: scalar, v: vector, ssss: replicated scalar

DirectX 9: Pixel 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

v: vector



DirectX 9: Pixel Shader 2.0

- texture sampling
  - syntax:

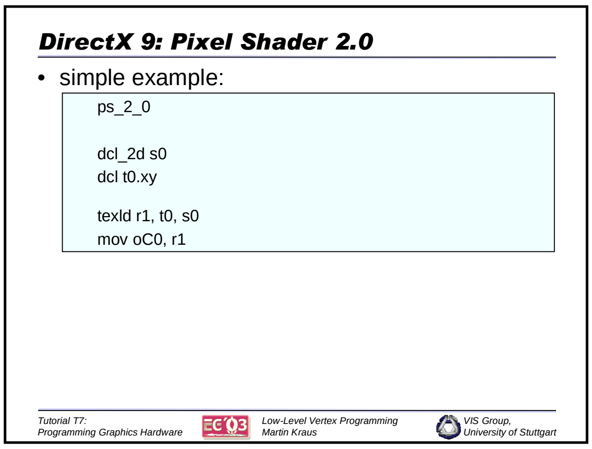
    op *destination*, *source*, s*n*
  - texture instructions (apart from texkill):

texld v, v, s*n*	texture load
texldp v, v, s*n*	texture load with projection
texldb v, v, s*n*	texture load with LOD bias

  - example:

    texld r2, t1, s0;

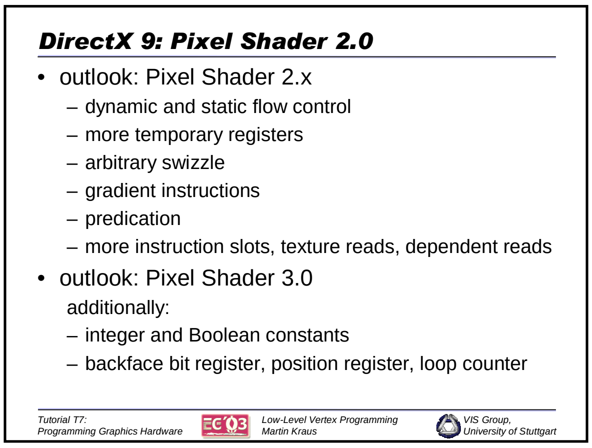
    samples texture for sampler 0 with texture coordinate set 1 and writes result to r2.



DirectX 9: Pixel Shader 2.0

- simple example:

```
ps_2_0

dcl_2d s0
dcl t0.xy

texld r1, t0, s0
mov oC0, r1
```



DirectX 9: Pixel Shader 2.0

- outlook: Pixel Shader 2.x
  - dynamic and static flow control
  - more temporary registers
  - arbitrary swizzle
  - gradient instructions
  - predication
  - more instruction slots, texture reads, dependent reads
- outlook: Pixel Shader 3.0

  additionally:
  - integer and Boolean constants
  - backface bit register, position register, loop counter