New NVIDIA OpenGL Extensions

Simon Green
NVIDIA Developer Technology

Overview

- Brief History of OpenGL
- Why extensions?
- New NVIDIA extensions

Brief History of OpenGL

- 1983 IRIS GL ships with SGI IRIS 1000 terminal
- 1987 SGI and Pixar consider joint API development
- 1991 OpenGL ARB created
- 1992 OpenGL 1.0 completed (June 30)
- 1995 OpenGL 1.1 released (vertex array, texture objects, new texenv modes)
- 1997 Fahrenheit agreement between SGI and Microsoft
- 1998 OpenGL 1.2 released (3D textures, separate specular, imaging)
- 1999 OpenGL 1.2.1 released (multi-texture)
- 2001 OpenGL 1.3 released (compressed texture, cube maps, multi-sample, dot3)
- 2002 OpenGL 1.4 (mip-map generation, shadows, point parameters)
- 2003 OpenGL 1.5 (vertex buffer objects, occlusion query)
  ARB extensions: OpenGL Shading language, ARB_vertex_program, ARB_fragment_program
- 2004 OpenGL 2.0 ?

Why Extensions?

- Vendors want to expose as much hardware functionality as possible
- Lets early adopters try new features as soon as possible
- Proven functionality is then incorporated into multi-vendor extensions

Life of an Extension

- GL_NVX_foo – eXperimental
- GL_NV_foo – vendor specific
- GL_EXT_foo – multi-vendor
- GL_ARB_foo
- Core OpenGL

History of Programmability in OpenGL

• EXT_texture_env_combine	
• NV_register_combiners	GeForce 256
• NV_vertex_program	GeForce 3
• NV_texture_shader	GeForce 3
• NV_texture_shader3	GeForce 4
• NV_vertex_program2	GeForce FX
• NV_fragment_program	GeForce FX
• ARB_vertex_program	
• ARB_fragment_program	

GameDevelopers Conference

EVOLVE GDC>04

New Extensions

- Two new program extensions
  - NV_vertex_program3
  - NV_fragment_program2
- Superset of DirectX 9 VS3.0 and PS3.0 functionality
- Exposed as options to ARB_vertex_program / ARB_fragment_program
  - `OPTION NV_vertex_program3;`
  - `OPTION NV_fragment_program2;`
- No new entry points, can use named parameters, temporaries etc.
- Previous program exts. also now available as options
- Functionality will also be exposed in OpenGL Shading Language

NVIDIA.

GL_NV_vertex_program3

- Textures lookups in vertex programs!
- Index-able vertex attributes and result arrays
  – More flexible skinning, animation
  – `MOV R0, vertex.attrib[A0.x+3];`
  – `MOV result.texcoord[A0.x+7], R0;`
- Additional condition code register (2 total)
- Can push/pop address registers on stack
  – For loop nesting, subroutine call / return
  – `PUSHA A0; POPA A0;`
- Up to 512 instructions

Vertex Texture

- Supports GL_NEAREST filtering only (currently)
- Supports mip-mapping
  - Need to calculate LOD yourself
  - TEX, TXP (projective), TXL (explicit LOD)
- Multiple vertex texture units
  - **`glGetIntegerv(MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB)`**
  - 4 units on new NVIDIA hardware
- Uses standard 2D texture targets
  - **`glBindTexture(GL_TEXTURE_2D, displace_tex);`**
- Currently must use **`LUMINANCE_FLOAT32_ATI`** or **`RGBA_FLOAT32_ATI`** texture formats

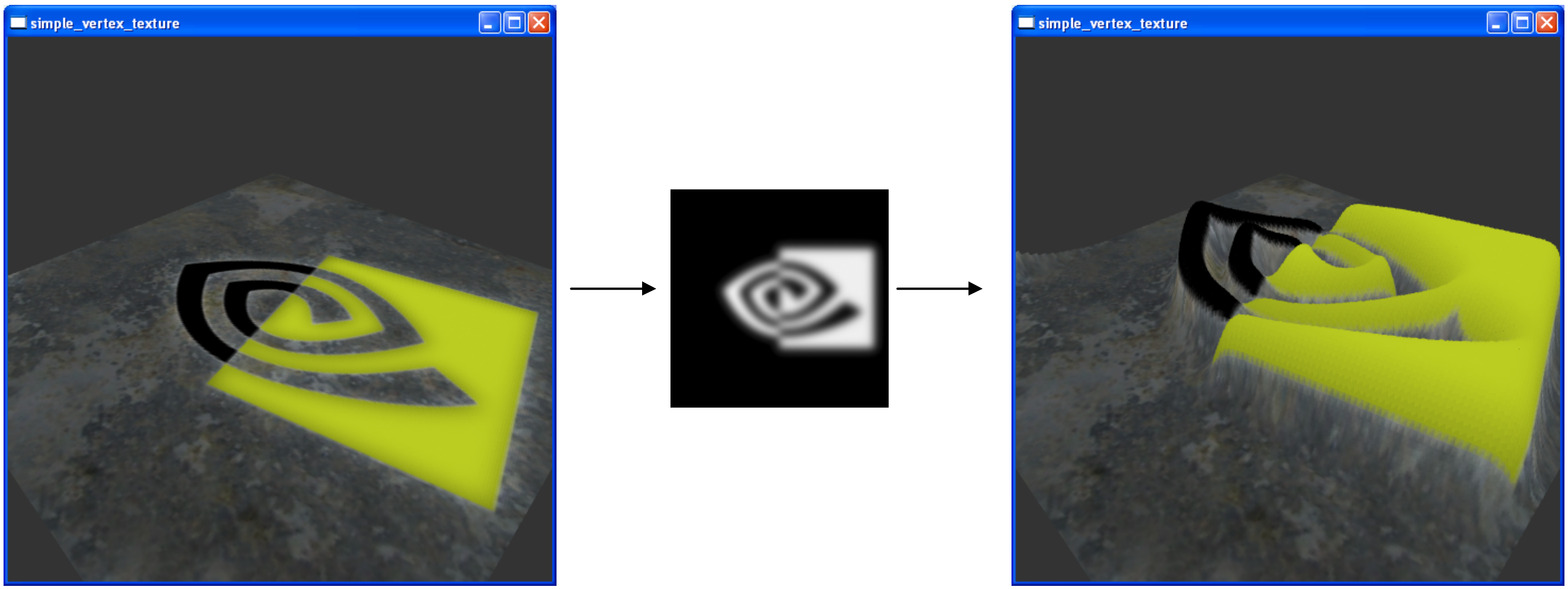
Vertex Texture Applications

- Simple displacement mapping
  - Not adaptive, hardware doesn't tessellate for you
  - Only worth it if texture coordinates or texture contents are dynamic, otherwise displacement could be baked into vertex data
  - Terrain, ocean rendering
- Render to vertex texture
  - Provides feedback path from fragment program to vertex program
  - Water simulation
  - Particle systems
  - Arbitrarily complex character animation

Vertex Texture Example

```
!!ARBvp1.0
OPTION NV_vertex_program3;
PARAM mvp[4] = { state.matrix.mvp };
PARAM scale = program.local[0];
TEMP pos, displace;
# vertex texture lookup
TEX displace, vertex.texcoord, texture[0], 2D;
MUL displace.x, displace.x, scale;
# displace along normal
MAD pos.xyz, vertex.normal, displace.x, vertex.position;
MOV pos.w, 1.0;
# transform to clip space
DP4 result.position.x, mvp[0], pos;
DP4 result.position.y, mvp[1], pos;
DP4 result.position.z, mvp[2], pos;
DP4 result.position.w, mvp[3], pos;
MOV result.color, vertex.color;
MOV result.texcoord[0], texcoord;
END
```

Vertex Texture Demo



GL_NV_vertex_program3 Performance

- Branching
  - Dynamic branches only have ~2 cycle overhead
    - Even if vertices take different branches
    - Use this to avoid unnecessary vertex work (e.g. skinning)
- Vertex texture
  - Look-ups are not free
  - Try to cover texture fetch latency with other non-dependent instructions
  - NOT practical for use as extra constant memory

GL_NV_fragment_program2

- Branching
  - Limited static and data-dependent branching
  - Fixed iteration-count loops
- Subroutine calls: CAL, RET
- New instructions: NRM, DIV, DP2
- Texture lookup with explicit LOD (TXL)
- Indexed input attributes
- Facing register (front / back)
  - can be used for two-sided lighting
- Up to 16K instructions

Instruction Set

ABS	absolute value
ADD	add
BRK	break out of loop instruction
CAL	subroutine call
CMP	compare
COS	cosine with reduction to [-PI,PI]
DDX	partial derivative relative to X
DDY	partial derivative relative to Y
DIV	divide vector components by scalar
DP2	2-component dot product
DP2A	2-comp. dot product w/scalar add
DP3	3-component dot product
DP4	4-component dot product
DPH	homogeneous dot product
DST	distance vector
ELSE	start if test else block
ENDIF	end if test block
ENDLOOP	end of loop block
ENDREP	end of repeat block
EX2	exponential base 2
FLR	floor
FRC	fraction
IF	start of if test block
KIL	kill fragment
LG2	logarithm base 2
LIT	compute light coefficients
LOOP	start of loop block
LRP	linear interpolation
MAD	multiply and add
MAX	maximum
MIN	minimum
MOV	move
MUL	multiply
NRM	normalize 3-component vector
PK2H	pack two 16-bit floats
PK2US	pack two unsigned 16-bit scalars
PK4B	pack four signed 8-bit scalars
PK4UB	pack four unsigned 8-bit scalars
POW	exponentiate
RCP	reciprocal
REP	start of repeat block
RET	subroutine return
RFL	reflection vector
RSQ	reciprocal square root
SCS	sine/cosine without reduction
SEQ	set on equal
SFL	set on false
SGE	set on greater than or equal
SGT	set on greater than
SIN	sine with reduction to [-PI,PI]
SLE	set on less than or equal
SLT	set on less than
SNE	set on not equal
STR	set on true
SUB	subtract
SWZ	extended swizzle
TEX	texture sample
TXB	texture sample with bias
TXD	texture sample w/partials
TXL	texture same w/explicit LOD
TXP	texture sample with projection
UP2H	unpack two 16-bit floats
UP2US	unpack two unsigned 16-bit scalars
UP4B	unpack four signed 8-bit scalars
UP4UB	unpack four unsigned 8-bit scalars
X2D	2D coordinate transormation
XPD	cross product

Branching

- Three types of instruction blocks
  - LOOP / ENDLOOP
    - Uses loop index register A0.x
  - REP / ENDREP
    - Repeats a fixed number of times
  - IF / ELSE / ENDIF
    - Conditional execution based on condition codes
- BRK instruction can be used to conditionally exit loops
- Blocks may be nested

Looping Limitations

- Loop count cannot be computed at runtime
  – Must be program parameter (constant)
- Number of iterations & nesting depth are limited
- Loop index register A0.x only available inside current loop
  – can only be used to index vertex attributes
- Can't index into constant memory
  – Can read data from texture instead

Branching Examples

```
LOOP {8, 0, 1};      # loop count, initial, increment
ADD R0, R0, fragment.texcoord[A0.x];
ENDLOOP;

REP repCount;
ADD R0, R0, R1;
ENDREP;

MOVC RC, R0;
IF GT.x;
  MOV R0, R1;        # executes if R0.x > 0
ELSE;
  MOV R0, R2;        # executes if R0.x <= 0
ENDIF;
```

Subroutine Calls

- CAL
  - Call subroutine, pushes return address on stack
- RET
  - address is popped off stack, execution continues at return address
  - execution stops if stack is empty, or overflows
  - can use as early exit from top level
- Note – no data stack – no recursion!
- Labels
  - Name followed by colon
  - Execution will start at “main:” if present

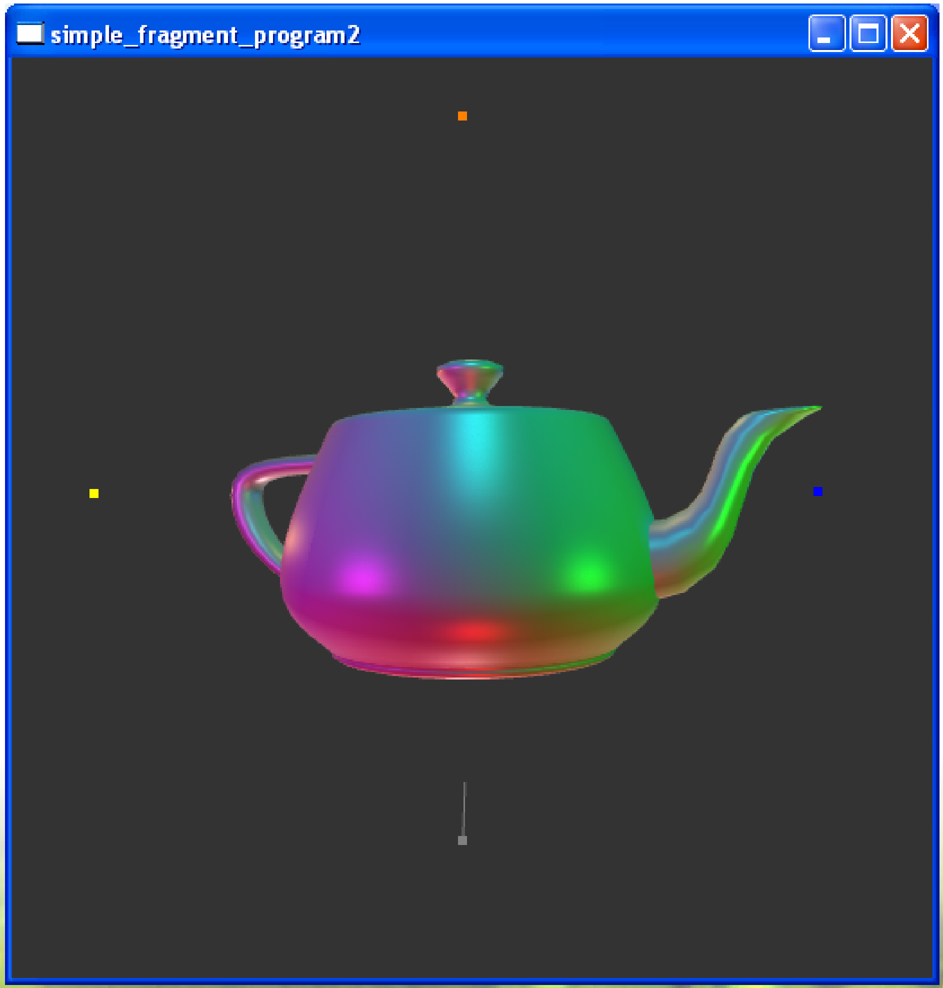
Looping Example

```
!!ARBfp1.0
OPTION NV_fragment_program2;
...
# loop over lights
MOV lightIndex.x, 0.0;
REP nlights;
    TEXC lightPos, lightIndex, texture[0], RECT;   # read light pos from texture
    TEX lightColor, lightIndex, texture[1], RECT;  # read light color from texture
    IF EQ.w;                                       # lightPos.w == 0
        CAL dirlight;
    ELSE;
        CAL pointlight;
    END
    ADD lightIndex.x, lightIndex, 1.0;               # increment loop counter
ENDREP;
MOV result.color, color;
RET;

pointlight:
…
RET;

dirlight:
…
RET
```

Fragment Program Branching Applications

- “Uber” shaders
  - Avoids writing separate shaders for different numbers, types of lights
- Image processing
- Early exit in complex shaders
  - Ray tracing
  - Volume rendering
  - Simulation

Multiple Lights Demo



Fragment Program Branching Performance

- Static branching is fast
  - But still may not be worth it for short branches (less than ~5 instructions)
  - Can use conditional execution instead
- Divergent (data-dependent) branching is more expensive
  - Depends on which pixels take which branches

More Performance Tips

- Use half-precision where possible
  - `SHORT TEMP normal;`
- Use NRM instruction for normalizing vectors, rather than DP3/RSQ/MUL
  - Very fast for half-precision data
- Always use write masks
  - `mul r0.x, r0.x, r2.w` (not `mul r0, r0.x, r2.w`)

Floating Point Filtering and Blending

- New NVIDIA hardware has fully-featured support for floating point textures
  - FP16 texture filtering – including tri-linear, aniso.
  - FP16 blending
  - Supports all texture targets, including cube maps, non-power-of-2 textures with mip-maps
- Exposed currently using ATI extensions:
  - GL_ATI_texture_float
  - WGL_ATI_pixel_format_float

FP16 Blending Example

FP16 Applications

- HDR imagery
  - 16-bit integer texture formats are not enough for very high dynamic ranges
- Image based lighting
- Interactive HDR paint
- Multi-pass algorithms

HDR With Int 16 Format

Dynamic range: 200,000:1

HDR With FP 16 Format

Dynamic range: 200,000:1

Multiple Draw Buffers

- Equivalent to Direct3D Multiple Render Targets (MRT)
- Exposed via ATI_draw_buffers extension
- Allows outputting up to 4 colors from a fragment program in a single pass:

```
MOV result.color[0], color;
MOV result.color[1], N;
MOV result.color[2], pos;
MOV result.color[3], H;
```

- Useful for deferred shading algorithms
- Will be exposed in GLslang also

Draw Buffers Example

Conclusion

- NV_vertex_program3 and NV_fragment_program2 expose the latest in programmable shading
- Functionality will be available in vendor-independent extensions and OpenGL Shading Language soon
- Start thinking about these features now, future hardware will be even faster