GeForce 6 Series OpenGL Extensions

Simon Green
NVIDIA Technical Developer Relations
Overview

- Brief History of Programmability OpenGL
- Why extensions?
- New NVIDIA extensions for GeForce 6 series
  - NV_vertex_program3
  - NV_fragment_program2
  - Multiple draw buffers
  - Floating point filtering and blending
  - Render to vertex array
- Demos!
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
New Extensions

- Two new program extensions
  - `NV_vertex_program3`
  - `NV_fragment_program2`
- Superset of DirectX 9 VS 3.0 and PS 3.0 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 Cg 1.3 and the OpenGL Shading Language
New features:

- Textures lookups in vertex programs!
- Index-able vertex attributes and result arrays
  - MOV R0, vertex.attrib[A0.x+3];
  - MOV result.texcoord[A0.x+7], R0;
- More flexible skinning, animation (blend shapes)
- 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 mip-mapping
  - Need to calculate LOD yourself
  - Use TXL instruction (explicit LOD)
- Currently supports GL_NEAREST filtering only
  - Can do own filtering in shader if necessary
- Multiple vertex texture units
  - `glGetIntegerv(MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB)`
  - 4 units supported on GeForce 6800
- 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
  - Note – not real adaptive displacement mapping
  - Hardware doesn’t tessellate for you
  - Terrain, ocean surfaces

- Render to vertex texture
  - Provides feedback path from fragment program to vertex program

- Particle systems
  - Calculate particle positions using fragment program, read positions from texture in vertex program, render as points

- Character animation
  - Can do arbitrarily complex character animation using fragment programs, read final result as vertex texture
  - Not limited by vertex attributes – can use lots of bones, lots of blend shapes

- Vertex textures are NOT practical for use as extra constant memory
!!ARBvp1.0
OPTION NV_vertex_program3;
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 on GeForce 6800
  - Even if vertices take different branches (MIMD hardware)
  - Use this to avoid unnecessary vertex work (e.g. skinning)

Vertex texture
- Look-ups are not free!
- Only worth using vertex texture if texture coordinates or texture contents are dynamic
  - Otherwise values could be baked into vertex attributes
- Coherency of texture access affects performance
  - If you don’t need random access, may be get better performance using render to vertex array with VBO/PBO

Try to cover texture fetch latency with other non-dependent instructions
Covering Vertex Texture Fetch Latency

```c
!!ARBvp1.0
OPTION NV_vertex_program3;
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
```
New features:

- 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 65,536 instructions
### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>absolute value</td>
</tr>
<tr>
<td>ADD</td>
<td>add</td>
</tr>
<tr>
<td>BRK</td>
<td>break out of loop instruction</td>
</tr>
<tr>
<td>CAL</td>
<td>subroutine call</td>
</tr>
<tr>
<td>CMP</td>
<td>compare</td>
</tr>
<tr>
<td>COS</td>
<td>cosine with reduction to ([-\pi, \pi])</td>
</tr>
<tr>
<td>DDX</td>
<td>partial derivative relative to (X)</td>
</tr>
<tr>
<td>DDY</td>
<td>partial derivative relative to (Y)</td>
</tr>
<tr>
<td>DIV</td>
<td>divide vector components by scalar</td>
</tr>
<tr>
<td>DP2</td>
<td>2-component dot product</td>
</tr>
<tr>
<td>DP2A</td>
<td>2-comp. dot product w/scalar add</td>
</tr>
<tr>
<td>DP3</td>
<td>3-component dot product</td>
</tr>
<tr>
<td>DP4</td>
<td>4-component dot product</td>
</tr>
<tr>
<td>DPH</td>
<td>homogeneous dot product</td>
</tr>
<tr>
<td>DST</td>
<td>distance vector</td>
</tr>
<tr>
<td>ELSE</td>
<td>start if test else block</td>
</tr>
<tr>
<td>ENDIF</td>
<td>end if test block</td>
</tr>
<tr>
<td>ENDLOOP</td>
<td>end of loop block</td>
</tr>
<tr>
<td>ENDREP</td>
<td>end of repeat block</td>
</tr>
<tr>
<td>EX2</td>
<td>exponential base 2</td>
</tr>
<tr>
<td>FLR</td>
<td>floor</td>
</tr>
<tr>
<td>FRC</td>
<td>fraction</td>
</tr>
<tr>
<td>IF</td>
<td>start of if test block</td>
</tr>
<tr>
<td>KIL</td>
<td>kill fragment</td>
</tr>
<tr>
<td>LG2</td>
<td>logarithm base 2</td>
</tr>
<tr>
<td>LIT</td>
<td>compute light coefficients</td>
</tr>
<tr>
<td>LOOP</td>
<td>start of loop block</td>
</tr>
<tr>
<td>LRP</td>
<td>linear interpolation</td>
</tr>
<tr>
<td>MAD</td>
<td>multiply and add</td>
</tr>
<tr>
<td>MAX</td>
<td>maximum</td>
</tr>
<tr>
<td>MIN</td>
<td>minimum</td>
</tr>
<tr>
<td>MOV</td>
<td>move</td>
</tr>
<tr>
<td>MUL</td>
<td>multiply</td>
</tr>
<tr>
<td>NRM</td>
<td>normalize 3-component vector</td>
</tr>
<tr>
<td>PK2B</td>
<td>pack two 16-bit floats</td>
</tr>
<tr>
<td>PK2US</td>
<td>pack two unsigned 16-bit scalars</td>
</tr>
<tr>
<td>PK4B</td>
<td>pack four signed 8-bit scalars</td>
</tr>
<tr>
<td>PK4UB</td>
<td>pack four unsigned 8-bit scalars</td>
</tr>
<tr>
<td>POW</td>
<td>exponentiate</td>
</tr>
<tr>
<td>RCP</td>
<td>reciprocal</td>
</tr>
<tr>
<td>REP</td>
<td>start of repeat block</td>
</tr>
<tr>
<td>RET</td>
<td>subroutine return</td>
</tr>
<tr>
<td>RFL</td>
<td>reflection vector</td>
</tr>
<tr>
<td>RSQ</td>
<td>reciprocal square root</td>
</tr>
<tr>
<td>SCS</td>
<td>sine/cosine without reduction</td>
</tr>
<tr>
<td>SEQ</td>
<td>set on equal</td>
</tr>
<tr>
<td>SFL</td>
<td>set on false</td>
</tr>
<tr>
<td>SGE</td>
<td>set on greater than or equal</td>
</tr>
<tr>
<td>SGT</td>
<td>set on greater than</td>
</tr>
<tr>
<td>SIN</td>
<td>sine with reduction to ([-\pi, \pi])</td>
</tr>
<tr>
<td>SLE</td>
<td>set on less than or equal</td>
</tr>
<tr>
<td>SLT</td>
<td>set on less than</td>
</tr>
<tr>
<td>SNE</td>
<td>set on not equal</td>
</tr>
<tr>
<td>STR</td>
<td>set on true</td>
</tr>
<tr>
<td>SUB</td>
<td>subtract</td>
</tr>
<tr>
<td>SWZ</td>
<td>extended swizzle</td>
</tr>
<tr>
<td>TEX</td>
<td>texture sample</td>
</tr>
<tr>
<td>TXB</td>
<td>texture sample with bias</td>
</tr>
<tr>
<td>TXD</td>
<td>texture sample w/partials</td>
</tr>
<tr>
<td>TXL</td>
<td>texture same w/explicit LOD</td>
</tr>
<tr>
<td>TXP</td>
<td>texture sample with projection</td>
</tr>
<tr>
<td>UP2H</td>
<td>unpack two 16-bit floats</td>
</tr>
<tr>
<td>UP2US</td>
<td>unpack two unsigned 16-bit scalars</td>
</tr>
<tr>
<td>UP4B</td>
<td>unpack four signed 8-bit scalars</td>
</tr>
<tr>
<td>UP4UB</td>
<td>unpack four unsigned 8-bit scalars</td>
</tr>
<tr>
<td>X2D</td>
<td>2D coordinate transformation</td>
</tr>
<tr>
<td>XPD</td>
<td>cross product</td>
</tr>
</tbody>
</table>
Fragment Program 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 or exit shader early
- Blocks may be nested
Looping Limitations

- Loop count cannot be computed at runtime
  - Must be a program parameter (i.e. 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
  - If you want to do something else you can maintain your own loop counter
- Can't index into constant memory in fragment programs
  - Can read data from texture instead
  - Think of texture as fragment program’s random access read-only memory
Branching Examples

LOOP {8, 0, 1};  
    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**

```plaintext
!!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;
        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
```
“Uber” shaders
- Avoids writing separate shaders for different numbers, types of lights
- Can help to increase batch size

Image processing
- Variable width filters
- For fixed width, probably faster to unroll loops

Early exit in complex shaders
- Ray tracing
- Volume rendering
  - can stop marching along ray when pixel is opaque
- GP-GPU simulations
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 spatial coherency of branching - which pixels take which branches
More Performance Tips

- Use half-precision where possible
  - OPTION ARB_precision_hint_fastest
  - or
    - 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

- GeForce 6 series has fully-featured support for floating point textures
  - Supports all texture targets, including cube maps, non-power-of-2 textures with mip-maps
  - Texture filtering for 16-bit float formats – including trilinear, anisotropic filtering
  - Blending for 16-bit float formats – all blending modes supported
- Exposed currently using ATI extensions:
  - GL_ATI_texture_float
  - WGL_ATI_pixel_format_float
  - These will be replaced with new ARB float extensions
FP16 Blending Example
FP16 Applications

- High-Dynamic-Range Imagery
  - 16-bit integer texture formats are not enough for very high dynamic ranges – can cause banding
- Multi-pass algorithms
  - e.g. one pass per light
- Interactive HDR paint
  - fp16 Photoshop
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;
  ```

- Outputs are written to GL_AUX buffers
  - Need to request a pixel format with aux buffers
  - All must be same format, share a single depth buffer
  - AUX buffers are allocated lazily to save memory
- Useful for deferred shading, reducing number of passes in general purpose algorithms
- Supported in Cg 1.3, GLslang soon
Draw Buffers Example
Render To Vertex Array

- Allows the GPU to interpret floating point frame buffer data as geometry – data stays resident on GPU

Applications
- GPU-based simulation – cloth, particles, soft bodies

3 possible implementations today:
- VAR / PDR
  - presented at GDC 2003 for cloth simulation, now obsolete
- VBO / PBO
  - uses new vertex / pixel buffer object extensions
  - works on all NV3x hardware
  - fast – 90M vertices / second measured on GeForce 6800!
- Vertex texture (NV_vertex_program3)
  - easy, only works with GeForce 6 series
- Uber/super buffers extension coming soon
Render To Vertex Array Examples
Render To Vertex Array using VBO/PBO

- Create buffer object for each vertex attribute you want to render to
  - use GL_STREAM_COPY usage flag
- Bind buffer object to pixel pack (destination) buffer
- Render vertex data to floating point pbuffer
- Do `glReadPixels` from pbuffer to buffer object
  - Implemented as fast copy in video memory by the driver
- Bind buffer object to vertex array
- Set vertex array pointers
- Draw geometry
- There will be example code in the new SDK
Conclusion

NV_vertex_program3 and NV_fragment_program2 expose the latest in programmable shading in OpenGL
Available on Windows, Linux and MacOS (soon)
Functionality will be available in vendor-independent extensions and OpenGL Shading Language
Start thinking about these features now, future hardware will be even faster and more flexible