# NV30 OpenGL Extensions

Mark J. Kilgard

## NV30 OpenGL Driver Essentials

1. Most compliant
   - Complete OpenGL 1.4 support
   - And more ARB extensions than anyone else
   - Unmatched driver quality
2. Backward compatible with previous GeForce generations and OpenGL extensions
   - Your existing OpenGL code "just works"
3. Most advanced 3D rendering functionality
   - Everything NV30 does exposed
   - Surpasses DirectX 9
   - And surpasses every competitor's hardware
4. Fastest hardware, fastest API

NVIDIA Proprietary

## What is OpenGL 1.4 ?

- Raises the base OpenGL functionality bar
  - Integrates proven functionality
    - 17 existing extensions integrated into the core
  - All OpenGL 1.4 functionality already in NVIDIA and Mesa OpenGL drivers
- Hopefully voted & approved by SIGGRAPH 2002
  - OpenGL 1.4 draft specification available now
  - OpenGL 1.3 released just last year
    - OpenGL core is rev'ing faster than DX8 to DX9

NVIDIA Proprietary

## New OpenGL 1.4 Functionality (1)

- Texture-related functionality
  - Automatic mipmap generation – SGIS_generate_mipmap
  - Texture crossbar environment mode – ARB_texture_env_crossbar
  - Shadow mapping – ARB_depth_texture, ARB_shadow
  - Texture level-of-detail (*blur-sharpen*) control – EXT_texture_lod_bias
  - Texture mirror repeat – ARB_texture_mirrored_repeat

NVIDIA Proprietary

## New OpenGL 1.4 Functionality (2)

- Blending & stencil-related functionality
  - DirectX 6 blend modes – NV_blend_square
  - Blend color, minmax, and subtract mandatory, no longer part of ARB_imaging – ARB_blend_color, ARB_blend_minmax, ARB_blend_subtract
  - Separate blend function for RGB and Alpha – EXT_blend_func_separate
  - DirectX 6 wrapping stencil operations – EXT_stencil_wrap

NVIDIA Proprietary

## New OpenGL 1.4 Functionality (3)

- Vertex processing-related functionality
  - Point parameters – ARB_point_parameters
  - Fog coordinate – ARB_fog_coord
  - Secondary (specular) color – ARB_secondary_color
  - Multiple draw arrays – EXT_multi_draw_arrays
  - Window-space raster position – ARB_window_pos

  - Perhaps ARB_vertex_program
    - Depends on Microsoft intellectual property issues

NVIDIA Proprietary

## Backward Compatible OpenGL Extension Support

- Still get all the ARB, EXT, and NV extensions you know and love
  - If you are using existing extensions, new extensions integrate well
  - For example, new NV_texture_shader texture formats work as expected
- One caveat
  - Dropping support for NV_evaluators



## Categories of NV30 Extensions

- Some more per-pixel functionality
- More data & frame buffer formats
- Better vertex arrays
- Better geometry
- Programmability
  - Better vertex programs
  - Amazing new fragment programs
  - *Heart & Soul of NV30*
  - Target for NV30 Cg compilation



## Disclaimer

- Prior to NV30's shipping, OpenGL extension APIs and functionality are subject to change
- That said, today the NV30 OpenGL extensions are fully implemented as described
- NV_vertex_program2 & NV_fragment_program likely to be subject to minor changes



## Per-pixel Functionality (1)

- EXT_blend_func_separate
  - Matches ATI, Creative, IBM, Intergraph, Mesa
  - Provides a separate RGB & Alpha blending state
    - Example: modulate Alpha while blending RGB
  - New glBlendFuncSeparateEXT(*sRGB*,*dRGB*,*sA*,*dA*)



## Per-pixel Functionality (2)

- EXT_stencil_two_side
  - In partnership with Apple & Id Software
  - Provides *front* and *back* stencil state
  - glActiveStencilFaceEXT(*mode*)
    - *Mode* is either GL_FRONT or GL_BACK
    - Affects glStencilOp, glStencilMask, glStencilFunc
  - Uses GL_STENCIL_TEST_TWO_SIDE_EXT enable
    - When enabled, polygons use appropriate state depending on how they face
    - Points and lines always use front-facing state
  - Ideal for stenciled shadow volumes
    - One pass for shadow volumes instead of two
    - See Everitt & Kilgard paper



## New Data and Frame Buffer Formats

- NV_half_float
  - Provides support for 16-bit floating-point representation throughout OpenGL
- NV_float_buffer
  - IEEE 32-bit floating-point components for textures and frame buffers

## NV_half_float (1)

- 16-bit floating point format
  - So-called s10e5 representation
    - 1 bit sign
    - 10 bit mantissa
    - 5 bit exponent, -15 bias
  - Otherwise IEEE 754 floating-point semantics
- Advantages & disadvantages
  - More range than signed shorts
  - Half the space of a 32-bit floating-point value
  - Warning: integer values > 1024 not all representable

NVIDIA Proprietary

## NV_half_float (2)

- Available throughout OpenGL
  - Immediate new commands with “h” suffix
    - Example: glColor4hv
  - Vertex array support
    - New type: GL_HALF_FLOAT_NV
  - Pixel formats
    - Draw & read pixels with GL_HALF_FLOAT_NV type
    - Texture images with GL_HALF_FLOAT_NV type
    - Fragment programs can use half float temporaries & pack/unpack instructions
      - More on this later

NVIDIA Proprietary

## NV_float_buffer Textures (1)

- Floating-point texture formats
  - 1, 2, 3, or 4 components per texel
  - 32-bit or 16-bit floating-point internal formats
    - IEEE s23e8 or s10e5
  - Twelve new internal formats
    - GL_FLOAT_R*n*_NV, GL_FLOAT_RG*n*_NV, GL_FLOAT_RGB*n*_NV, and GL_FLOAT_RGBA*n*_NV
    - Where *n* is 16, 32, or nothing
    - For glTexImage2D and glCopyTexImage2D

NVIDIA Proprietary

## NV_float_buffer Textures (2)

- Floating-point texture limitations
  - Only GL_NEAREST filtering
    - Try fragment programs for filtering
      - Hint: summed area tables
    - Corollary: No mipmap filtering
  - Only GL_TEXTURE_RECTANGLE_NV texture target
    - No 1D, 2D, 3D, or cube map floating-point texture targets
  - Requires fragment programs to use
    - Conventional texture environment & register combiners cannot use float textures
    - More on fragment programs later

NVIDIA Proprietary

## NV_float_buffer Frame Buffers (1)

- High-level functionality
  - Create pixel buffers (*pbuffers*) with floating-point frame buffer formats
  - Permits OpenGL rendering to & read-back from floating-point frame pixel buffers
  - Use floating-point pixel buffers with WGL_NV_render_to_texture_rectangle
- Window system dependent related extensions
  - WGL_NV_float_buffer for Windows
  - GLX_NV_float_buffer for Linux/X11

NVIDIA Proprietary

## NV_float_buffer Frame Buffers (2)

- WGL pixel formats for floating-point buffers
  - wglChoosePixelFormatARB parameters
    - WGL_TEXTURE_FLOAT_R_NV, WGL_TEXTURE_FLOAT_RG_NV, WGL_TEXTURE_FLOAT_RGB_NV, and WGL_TEXTURE_FLOAT_RGBA_NV
  - wglGetPixelFormatAttrib*xv*ARB parameters
    - WGL_FLOAT_COMPONENTS_NV, WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_R_NV, WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RG_NV, WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGB_NV, WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV
- Similar GLX interface

NVIDIA Proprietary

## NV_float_buffer Frame Buffers (2)

- Floating-point frame buffer limitations
  - No coverage application multiply of alpha
    - Affects GL_POINT_SMOOTH, GL_LINE_SMOOTH, and GL_POLYGON_SMOOTH
  - No alpha test
    - Use KIL fragment program instruction instead
  - No frame buffer blending, logic-op, or dithering
  - No multi-sampled floating-point frame buffers
    - Use render to texture & fragment programs to "downsample" high-resolution floating-point frame buffers
  - No accumulation buffer support



## NV_float_buffer Frame Buffers (3)

- What does work with floating-point frame buffers
  - All primitive types
    - points, lines, polygons, image rectangles, and bitmaps
  - Fragment programs, of course!
  - Texturing
    - Recall that float textures are consistent only when used with fragment programs
  - Texture environments, fog, & color sum
    - But only generates clamped [0,1] fragment colors so not that interesting
  - Scissoring
  - Depth & stencil testing
  - Color mask



## NV_float_buffer Frame Buffers (3)

- Floating-point frame buffer caveats
  - glClearColor maintains unclamped value for floating-point frame buffer clears
    - GL_FLOAT_CLEAR_COLOR_VALUE_NV state
    - Fixed-point color buffers use [0,1] clamped clear values
  - Floating-point frame buffer writes & reads are not clamped to [0,1]
    - What you want generally
    - Careful, pixel path functionality ***will*** clamp if clamping is part of the functionality
  - No way to display floating-point frame buffers since *pbuffers*



## Better Vertex Arrays

- NV_half_float
  - Half float vertex arrays *(already mentioned)*
- NV_element_array
  - Equivalent to Direct3D "index buffers"
- NV_primitive_restart
  - Magic array element index restarts same primitive



## NV_element_array (1)

- Conventional OpenGL
  - glDrawElements takes array of vertex array element indices
  - Despite such arrays typically being static
    - Since model topology is typically static
- New element array functionality
  - Specify vertex array element indices in their own vertex array
  - Render models by drawing sequential ranges of indices pulled from the element array
  - Best used with NV_vertex_array_range



## NV_element_array (2)

- Setup usage
  - Establish an element array like a conventional vertex array
    - glElementPointerNV(*type*, *pointer*)
    - No *stride* or *count*
    - *Type* should be GL_UNSIGNED_SHORT or GL_UNSIGNED_INT
      - GL_UNSIGNED_BYTE allowed but not advised
  - Best when used in conjunction with NV_vertex_array_range
    - Then GPU reads element array and then reads element index's array data
    - CPU moves almost no data

## NV_element_array (3)

- **Rendering usage**
  - **Specify sequential range of element array**
    - **Like glDrawArrays but with a level of indirection**
  - **Commands are glDrawElements variants**
    - **glDrawElementArrayNV(*mode*, *first*, *count*) reads [*first*,*first*+*count*-1] element indices from element array and uses the indices a la glDrawElements**
    - **glDrawRangeElementArrayNV(*mode*, *start*, *end*, *first*, *count*) is like glDrawElementArrayNV**
      - **But guarantees all element array indices are within the range [*start*,*end*]**



## NV_element_array (4)

- **More rendering usage**
  - **Also *Multi* versions of these routines**

    **glMultiDrawElementArrayNV(*mode*, *firstArray*, *countArray*, *primCount*)**

    **glMultiDrawRangeElementArrayNV(*mode*, *start*, *end*, *firstArray*, *countArray*, *primCount*)**



## NV_element_array (5)

- **Element array advantages**
  - **No copying of index arrays to GPU every glDrawElements call when using glDrawElementArrayNV instead**
  - **Model topologies can be cached in element array and re-used by accessing the element array contents**
- **Element array performance requirements**
  - **Must be used in conjunction with vertex array range**
    - **NV_vertex_array_range**
  - **Then GPU does all the heavy lifting**



## NV_primitive_restart (1)

- **If a model is a soup of independent triangles, it can be rendered with a single glDrawElements call**
  - **But lots of redundant element indices are required**
  - **Often more efficient to *strip-ify* a model**
- **A model can *strip-ified* into some number of triangle strips**
  - **Or quad strips**
  - **Or triangle fans**
- ***Strip-ification* can often require almost 66% fewer indices per triangle!**



## NV_primitive_restart (2)

- **Strip-ified geometry requires multiple glDrawElements calls**
  - **Or glMultiDrawElements**
  - **Application still must keep track of different arrays of indices**
- **Primitive restart makes it fast and easy to render a strip-ified model in a single glDrawElements call**
  - **Or single glDrawElementArrayNV call!**



## NV_primitive_restart (3)

- **Primitive restart command**
  - **New immediate mode glPrimitiveRestartNV() call**
  - **Effectively, glEnd(); glBegin(*mode*); sequence**
    - **Assuming already within a glBegin**
    - **Where *mode* is the previous glBegin's *mode***
- **But how do we accomplish a primitive restart when using vertex arrays?**

## NV_primitive_restart (4)

- Using primitive restart for vertex arrays
  - Specify what element index value indicates a primitive restart with glPrimitiveRestartIndexNV(*index*)
    - The initial primitive restart index is zero
  - Enable GL_PRIMITIVE_RESTART_NV
    - Using glEnableClientState
  - Use glDrawElements but use the value of *index* in the *indices* array to indicate when a primitive restart should occur



## NV_primitive_restart (5)

- Primitive restart performance
  - Must use with NV_vertex_array_range for optimal performance
  - Works with NV_element_array extension
    - Helps minimize the storage requirements in the vertex array range for storing element arrays
  - Post-transform vertex cache still optimizes vertex element re-use within a glDrawElements command using primitive restart
    - So arrange strip-ified element arrays to maximize vertex re-use



## Better Geometry

- Displacement mapping & triangle tessellation
- Details still being worked out
- Talk with us about



## Programmability versus Configurability

- Old School OpenGL: *configurable* state machine for high-performance rendering
  - Conventional OpenGL lighting
  - NV_register_combiners & NV_register_combiners2
  - NV_texture_shader
- New School OpenGL: ***programmable*** state machine for high-performance rendering
  - NV20’s NV_vertex_program, a good start
  - NV30’s NV_vertex_program2 & NV_fragment_program



## OpenGL Programming Philosophy

- Program objects
  - Specified as ASCII text strings
    - Specified & parsed with a regular grammar
    - Assembly-like syntax
  - Similar to texture objects
  - Multiple program targets (*vertex* & *fragment*)
- Execution model
  - Enable enters programmable mode
    - Conventional OpenGL processing bypassed
  - Inputs -> Program -> Outputs
    - Each vertex/fragment processed in relative isolation



## Domains for Programmability (1)

- Vertex programs for per-vertex programmability
  - NV20’s NV_vertex_program & NV_vertex_program1_1
  - Standard ARB_vertex_program
    - Comparable to NV_vertex_program
  - NV30’s NV_vertex_program2
- Fragment programs for per-fragment programmability
  - NV30’s NV_fragment_program

## Domains for Programmability (2)

- NV30 per-vertex & per-fragment similarities
  - Similar 4-component floating-point instruction set
    - MAD, DP4, MIN, MAX, SGE, RSQ, COS, etc
  - Swizzling, negation, absolute value, write-masking
  - Very similar IEEE 754-like floating-point semantics
  - Condition code mechanism
  - Same program object model
  - Same basic programming interface

NVIDIA Proprietary

## Domains for Programmability (3)

- NV30 *per-vertex* specific features
  - Branching & subroutines
  - Global program parameters
  - Address register relative addressing
  - Inputs: 16 vertex attributes (aliased) for input
  - Outputs: 1 position, 2 clamped RGBA colors, 8 texture coordinate sets, 1 fog coordinate, 1 point position, and 6 clip coordinates
    - New clip coordinates for clip planes

NVIDIA Proprietary

## Domains for Programmability (4)

- NV30 *per-fragment* specific features
  - Texture fetch instructions
  - Screen-space partial derivatives
  - Local program parameters
  - Embedded vector constants
  - Half-precision & X-precision operations and temporaries
  - Saturation to [0,1] range for outputs
  - KIL instruction terminates fragment
  - Pack & unpack instructions

NVIDIA Proprietary

## Domains for Programmability (5)

- NV30 *per-fragment* specific features
  - Inputs: 1 window position (x,y,z,1/w), 2 colors, 8 texture coordinate sets, 1 fog coordinate
  - Outputs:
    - Either fragment color, optional fragment depth
    - Or 4 texture RGBA results, optional fragment depth
  - 4 texture results feed NV_register_combiners
    - Special "fragment combiner" program

NVIDIA Proprietary

## ARB_vertex_program

- Recently standardized ARB extension
  - Similar to NV_vertex_program
  - Multi-vendor support: ATI, Creative, NVIDIA, Matrox
- NV30 has complete ARB_vertex_program support
  - However NV30's NV_vertex_program2 has far more functionality than ARB_vertex_program
- More on ARB_vertex_program in a later presentation

NVIDIA Proprietary

## NV_vertex_program Overview

1. Condition codes
2. Branching & subroutines
3. Even faster performance
4. Nineteen new instructions
5. New source modifiers
6. Clip plane support
7. More registers & instructions

NVIDIA Proprietary

## NV_vertex_program2 API & Usage

- Uses the same API established by NV_vertex_program
  - No new commands
  - No new tokens
- NV_vertex_program2 program text begins with !!VP2.0 start token
  - Older !!VP1.0 and !!VP1.1 grammars retain the limitations of NV2x
  - Using !!VP1.0 and !!VP1.1 guarantees accepted programs will run on older GPUs



## NV_vertex_program2 Resource Limits

- 256 vertex program parameters
  - Up from 96
- 16 temporary registers
  - Up from 12
- Two 4-component address registers
  - Up from one single-component address register
- 256 static instructions per program
  - Up from 128
  - Given branching, 65536 dynamic instructions can execute before termination to avoid infinite loops



## NV_vertex_program2 Performance

- Vertex program performance
  - Clock for clock
    - Over 3x faster than NV20
    - Over 1.5x faster than NV25
  - Less overhead launching vertices
  - Older VP1.0 and VP1.1 programs go faster too
- Straightforward performance model
  - Vertex program execution rate is essentially inversely proportional to number of instructions executed per vertex
  - Branching makes this dynamic



## NV_vertex_program2 Source Modifiers

- Source operand absolute value
  - Example: MOV R0, |R1|;
  - In addition to source negation & swizzling
    - Example: MAD R0, -|R1|.yzwy, |R2|, -R3,w;
  - Swizzle, negate, & absolute value operations are "free" source modifiers



## NV_vertex_program2 Condition Codes (1)

- Condition code state
  - 4-component register stores condition code values
  - Four possible values
    - LT – less than zero
    - EQ – equal to zero
    - GT – greater than zero
    - UN – unordered, for comparisons involving NaN
- Most instructions optionally update condition code state
  - Indicated with "C" suffix: DP4C, MOVC, etc
  - "CC" pseudo-register used to just update condition codes



## NV_vertex_program2 Condition Codes (2)

- Optional condition code based destination masking
  - Example: MOV R1.xy (NE.z), R0;
    - Copy R0 components to R1's X & Y components *__except__* when condition code's Z component is EQ
  - Condition code rules: EQ, equal; GE, greater or equal; GT, greater than; LE, less or equal; LT, less than; NE, not equal; FL, false; and TR, true
  - Note that condition code masking rule can swizzle condition code components

## NV_vertex_program2 Branching

- First-class branching support, BRA instruction
  - Unconditional and conditional branches
    - Conditional branches based on condition codes
    - Example: BRA label (LE.xyww);
      - Branches to label if any of the X, Y, or W condition code components are LT or EQ
    - Label syntax example: label: MOV R0, R1;
  - Computed branches
    - Example:
      JMPTABLE = { a, b, c, d };
      BRA [A1.z] (GT.x);
      - If A1.z is 2 and the the condition code's X component is greater than, branch c



## NV_vertex_program2 Subroutines

- Call & return for subroutines
  - CAL & RET instructions
  - Branch conditions apply
    - Example: CAL label (GE.y); RET (LT.xxyy);
  - Four levels of subroutine execution
  - No parameter stack



## NV_vertex_program2 Clipping

- Six new output registers for clip codes
  - Named o[CLP0] .. o[CLP5]
- When GL_CLIP_PLANE*n* is enabled
  - Clip coordinate *n* is interpolated across the primitive
  - Only the portion of the primitive where the clip coordinate is greater than zero is rasterized
  - Hardware performs fast trivial reject if all clip coordinates of a primitive are negative



## New NV_vertex_program2 Instructions (1)

- ARL – supports loading 4-component A0 and A1 integer registers now
  - Rather than just A0.x
- ARR – like ARL except rounds rather than truncates before storing integer result in an address register
- BRA, CAL, RET – branching instructions, discussed earlier
- COS, SIN – high-precision trigonometric functions
- FLR, FRC – floor and fraction of floating-point values



## New NV_vertex_program2 Instructions (2)

- EX2, LG2 – high-precision exponentiation and logarithm functions
- ARA – adds pairs of components of an address register; useful for looping and other operations
- SEQ, SFL, SGT, SLE, SNE, STR – add six “set on” instructions similar to SLT and SGE
- SSG – “set sign” operation generates a vector holding –1.0 for negative operand components, 0 for zero components, and +1.0 for positive components



## Complete NV_vertex_program2 Instruction List (1)

- Add & multiply instructions
  ADD, DP3, DP4, DPH, MAD, MOV, SUB
- Math functions
  ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, RCP, RSQ, SIN
- Set on instructions
  SEQ, SFL, SGE, SGT, SLE, SLT, SNE, STR
- Branching instructions
  BRA, CAL, RET
- Address register instructions
  ARL, ARA

## Complete NV_vertex_program2 Instruction List (2)

- Graphics-oriented instructions
  DST, LIT, RCC, SSG
- Minimum / maximum instructions
  MAX, MIN



## NV_fragment_program Overview

1. Similar to NV_vertex_program2, but per-fragment
2. 32-bit & 16-bit floating-point and X precision & storage
3. Texture lookup instructions for 16 texture images
4. Local program parameters, rather than global
5. Immediate constants
6. Condition codes
7. Screen-space partial derivatives
8. Pack & unpack instructions
9. KIL pixel kill
10. LRP and X2D math instructions
11. Saturate result to [0,1] instruction modifier
12. Fragment combiner programs (!!FCP1.0)
13. Depth replace

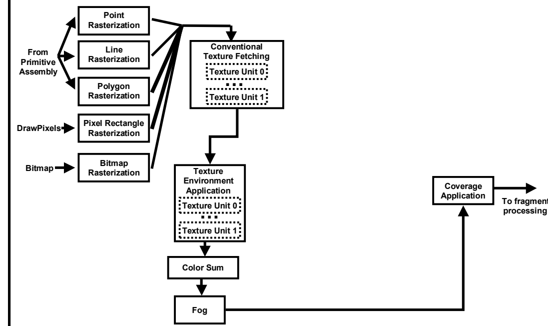


## OpenGL Fragment Dataflow

- OpenGL has evolved over time
  - Single textured OpenGL 1.1
  - ARB_multitexture (TNT)
  - NV_register_combiners (GeForce 256)
  - NV_texture_shader (GeForce3)
  - And now NV_fragment_program (NV30)
- So where does NV_fragment_program fit?

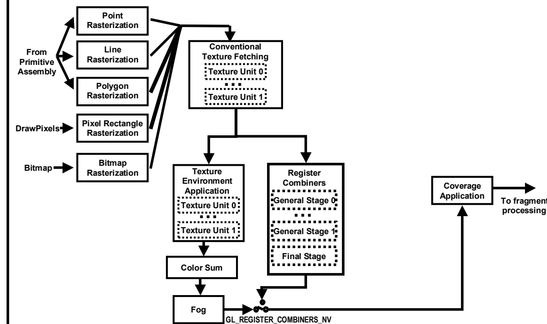


## Core OpenGL Fragment Texturing & Coloring



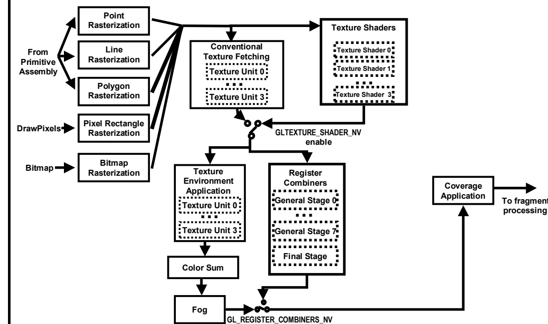



## NV10 OpenGL Fragment Texturing & Coloring

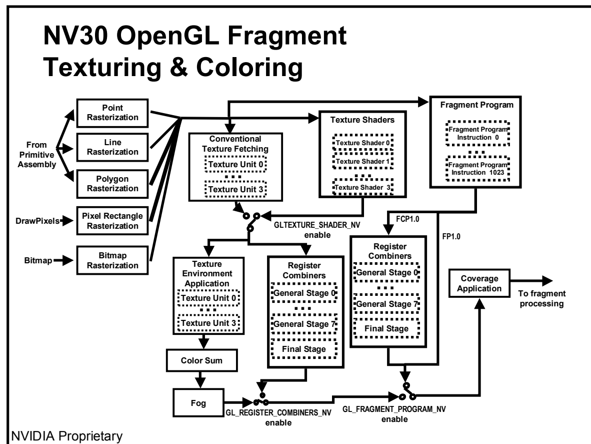





## NV20 OpenGL Fragment Texturing & Coloring



## NV30 OpenGL Fragment Texturing & Coloring




NVIDIA Proprietary

## NV_fragment_program API & Usage

- Uses same basic API established by NV_vertex_program
  - New target & enable: GL_FRAGMENT_PROGRAM_NV
  - Adds glProgramLocalParameter4fNV and similar commands
- NV_fragment_program program text begins with !!FP1.0 start token
  - Also provides fragment combiner program using the !!FCP1.0 start token, more on this later

NVIDIA Proprietary

## NV_fragment_program Resources (1)

- Up to 1024 instructions
  - No branching permitted
- Fragment programs themselves stored in video memory
  - Unlike vertex programs that are stored internally
  - Makes managing lots of fragment programs cheap

NVIDIA Proprietary

## NV_fragment_program Resources (2)

- Inputs
  - 1 position (x,y,z,1/w), 2 colors, 8 texture coordinate sets, 1 fog coordinate
  - Names: f[WPOS], f[COL0], f[COL1], f[TEX0], … f[TEX7], f[FOGC] respectively
  - Interpolated perspective correctly
- Temporaries
  - Up to 64 16-bit float temporaries overlapped with 32 32-bit float temporaries
  - 4-component values, initialized to (0,0,0,0)
  - Names: R0-R31 (32-bit) and H0-H63 (16-bit)

NVIDIA Proprietary

## NV_fragment_program Resources (3)

- Outputs
  - Fragment's RGBA color
    - Can be floating-point when using NV_float_buffer
    - Name: o[COLR] (32-bit) or o[COLH] (16-bit)
  - Optionally, fragment's computed depth
    - Name: o[DEPR]
  - FCP1.0 programs output 4 RGBA texture results instead of a single color
    - Names: o[TEX0] … o[TEX3]
    - Feeds NV_register_combiners functionality
  - Output registers overlap temporaries

NVIDIA Proprietary

## Fragment Program Temporary & Output Register Usage

- Programs must have a *register count* less than or equal to 64 to load
  - Depends on temporaries & output registers
- Computing the *register count*
  - Each R# temporary register counts as 2 points
  - Each H# temporary register counts as 1 point
  - o[TEX0], o[TEX1], o[TEX2], & o[TEX3] each count as 1 point
  - o[COLR] & o[DEPR] each count as 2 points
  - o[COLH] count as 1 point

NVIDIA Proprietary

## NV_fragment_program Similarities

- Same basic instruction features as NV_vertex_program2
  - Swizzling, negation, absolute value, write-masking
  - One destination, 0 to 3 sources
  - Condition code support
    - But now branch & subroutine instructions
  - 4-component vector instruction set
    - Familiar instructions, plus some



## NV_fragment_program Differences

- Two storage formats
  - 32-bit floating-point 4-component vectors (128 bits)
  - 16-bit floating-point 4-component vectors (64 bits)
- Three computation precisions
  - R precision: 32-bit floating-point
  - H precision: 16-bit floating-point
  - X precision: [-2,2) fixed-point
  - More math units available to compute X precision
  - Instructions uses these precision suffixes
    - Example: MULX H0, H1, R2; LG2R R0, H1;
    - No suffix defaults to destination precision



## NV_fragment_program Texturing (1)

- Three texture lookup instructions
  - TEX – non-projective texturing
    - Example: TEX H0.x, f[TEX3].yzxx, 2D,3;
    - 2D,3 indicates access the 2D texture target for unit 3
    - Uses (y,z) components of f[TEX3] to access texture
  - TXP – projective texturing
    - Example: TXP H1, R5, 3D,0;
    - Uses (x/w,y/w) of R5 to access texture
  - TXD – non-projective texturing with explicit partials
    - Program controlled filtering, including anisotropy
    - Example: TXD H1, f[TEX0], R3, R4;
    - R3 and R4 supply partial derivatives



## NV_fragment_program Texturing (2)

- Texture image units
  - 16 distinct texture images available
    - Note: only 8 texture coordinate sets
    - Texture accesses can be made with interpolated or arbitrarily computed coordinates
  - 5 texture targets
    - Names: 1D, 2D, 3D, CUBE, RECT
- Texture object usage
  - Bind texture object to give image unit & target
- Conventional texture enables ***ignored***



## NV_fragment_program Texturing (3)

- Texture filtering
  - Hardware automatically computes mipmapping level-of-detail
    - Even for dependent & computed texture coordinates
  - Texture object bound to particular image unit/target pair determines filtering parameters



## OpenGL Texture Resources (1)

- Three distinct limits now
  - OpenGL 1.3's GL_MAX_TEXTURE_UNITS
    - For conventional OpenGL texturing & register combiners
    - NV30's value is 4
  - NV_fragment_program's GL_MAX_TEXTURE_COORDS_NV
    - Number of texture coordinate sets
    - NV30's value is 8
  - NV_fragment_program's GL_MAX_TEXTURE_IMAGE_UNITS_NV
    - Number of texture images for fetching texels
    - NV30's value is 16

## OpenGL Texture Resources (2)

- GL_MAX_TEXTURE_UNITS applies to
  - glTexEnv calls for texture environment application & NV_texture_shader
  - Number of NV_register_combiners texture registers
- GL_MAX_TEXTURE_COORDS_NV applies to
  - glTexGen calls
  - glActiveClientTexture, glTexCoordPointer, glEnableClientState & glDisableClientState for GL_TEXTURE_COORD_ARRAY
  - Number of vertex program texture input and output registers



## OpenGL Texture Resources (3)

- GL_MAX_TEXTURE_IMAGE_UNITS_NV applies to
  - glTexImage*N*D, glBindTexture, glTexParameter, glEnable for texture targets, glTexEnv for LOD bias
  - Without using NV_fragment_program, only 4 texture units are available
- Rationale
  - Does not make sense to scale texture machinery in every direction
  - Old-style texture environment too limited a model



## Fragment Program Constants & Local Parameters

- NV_vertex_program model provides global program parameter registers
  - Named c[0] … c[255]
  - Numbered registers, rather than named
  - Per-context (*global*) state
  - NV_fragment_program does not provide such numbered, global program parameters
- NV_fragment_program model provides
  - Program constants
    - Immutable
  - Local program parameters
    - Per-program, mutable, named



## Fragment Program Constants

- Two methods
  - Literally specify floating-point constants in program
    - Example: MOV H0, { 1, 2.5, -4, +8.0 };
  - Otherwise, use DEFINE construct
    - Example: DEFINE Pi = 3.14159; MUL H0, H1, Pi;
    - Example: DEFINE Vector = { 1, 2, 4, 8 };<br>DP3 H0, H2, Vector.xxwy;
    - DEFINE namespace local to each fragment program
    - Allows symbolic names for constants
- Storage
  - Constants embedded in programs
  - No limit to the number of constants



## Fragment Program Local Parameters

- Named rather than numbered
  - Example: “lightPosition”
- Local to a given fragment program object
- Use DECLARE rather than DEFINE
  - Example:<br>DECLARE lightPosition = { 3.2, 8.5, -9.1, 1 };<br>DP3 H0, f[TEX0], lightPosition;
- Mutable by name
  - glProgramLocalParameter*NV calls
  - Example
    - glProgramLocalParameter4fv(*progID*, strlen(“lightPosition”), “lightPosition”, -3.5, 8.6, -9.2, 1);



## Fragment Program Partial Derivatives (1)

- Partial derivative approximation instructions
  - DDX – computes vector derivative of a register in term of screen-space X
  - DDY – computes vector derivative of a register in term of screen-space Y
- Applications
  - Anti-aliasing procedural shaders
  - Height-field bump mapping
  - Computing parameters for the TXD texture lookup with partial derivatives

## Fragment Program Partial Derivatives (2)

- How DDX and DDY work
  - Finite differencing with adjacent fragments
    - Difference with left or right fragment for DDX
    - Difference with above or below fragment for DDY
  - Just an approximation
- No second derivatives possible
  - DDX of a register value computed by DDX is zero
  - Same with DDY
- Derivatives in terms of values other than screen-space x & y requires extra math
  - Involves solving linear system of equations



## Fragment Program Pack and Unpack Instructions (1)

- Two component packing & unpacking
  - PK2H – converts X & Y components to 16-bit floating-point and packs the two values into a 32-bit floating-point value
    - Reversed by UP2H unpack instruction
  - PK2US – saturates to [0,1] and then converts X & Y components to 16-bit fixed-point fractions and packs the two values into a 32-bit floating-point value
    - Reversed by UP2US



## Fragment Program Pack and Unpack Instructions (2)

- Four component packing & unpacking
  - PK4B – saturates to [-1,1] and then converts X, Y, Z, & W components to 8-bit signed fixed-point fractions and packs the four values into a 32-bit floating-point value
    - Reversed by UP4B unpack instruction
  - PK4UB– saturates to [0,1] and then converts X, Y, Z, & W components to 8-bit unsigned fixed-point fractions and packs the four values into a 32-bit floating-point value
    - Reversed by UP4UB



## Fragment Program Pack and Unpack Instructions (3)

- Gamma-corrected four component packing & unpacking
  - PK4UBG – like PK4UB but gamma corrects each component after saturation and before packing
    - Reversed by UP4UBG
- Applications
  - Provides a means to pack multiple values into floating-point frame buffer components
    - When using NV_float_buffer
  - RGBA float buffer could contain
    - R=32-bit float, G=2 packed 16-bit floats, etc



## Fragment Program KIL Instruction

- KIL instruction conditionally discards a fragment
  - Used in conjunction with condition codes
  - Example: KIL ( LE.xyzz );
  - Example: KIL ( GT.w );
- Applications
  - Arbitrarily programmed alpha test



## Extra Math Instructions (1)

- LRP – linear interpolation
  - Computes vector A * B + (1-A) * C
  - Faster for X fixed-point precision that floating-point
    - Example: LRPX H0, H1, H2, H3;
- X2D – 2D coordinate transformation
  - Computes
    ( A.x * B.x + B.y * C.y ,
    A.y * B.x + B.y * C.w ,
    A.x * B.x + B.y * C.y,
    A.y * B.x + B.y * C.w )
  - Example: X2D H0, H1, H2, H3;
- Not vertex program instructions

## Extra Math Instructions (2)

- RFL – computes reflection of the 2nd vector operand (the *direction* vector) about the vector specified by the 1st vector operand (the *axis* vector)
  - Both operands are treated as 3D vectors
  - The w components is ignored
  - The length of the result, ignoring rounding errors, should equal that of the second operand
- POW – exponentiation
  - Approximates A^B
  - Assumes A is positive, otherwise generates *NaN*
- Again, not vertex program instructions



## Fragment Program Saturation Output Modifier

- Nearly all fragment program instructions support a saturation output modifier
  - When specified, clamps output to the range [0,1]
  - Indicated by the _SAT suffix
    - Example: ADD_SAT H0, f[COL0], f[COL1];
  - Exceptions
    - KIL and pack instructions



## Fragment Combiner Programs (1)

- NV_register_combiners provides an efficient end-game for a fragment program
  - Advantages of NV_register_combiners
    - Free input mappings
    - Up to 6 operations per general combiner stage
    - Free final combiner math
    - Per-context (rather than local) parameters
    - Free scale & bias
- Fragment combiner programs feed register combiners texture registers
  - Indicated by !!FCP1.0 start token
  - Output names: o[TEX0] … o[TEX3]



## Fragment Combiner Programs (2)

- Fragment combiner program outputs
  - Instead than a single RGBA color output as with a standard fragment program
  - Four RGBA texture results are output
  - These output values initialize the four register combiners texture registers
  - NV_register_combiners operates as in NV20
- Useful when your final fragment program operations could be more efficiently performed using the register combiners



## Fragment Program Depth Replace

- Optionally, a fragment program can replace the interpolated depth with a program-computed depth
  - Output a new depth to o[DEPR]
- Similar to NV_texture_shader's GL_DOT_PRODUCT_DEPTH_REPLACE_NV operation
  - But fragment program clamps to depth range rather than clipping against it
- Performance notes
  - Computing depth values pre-empts early depth buffer reject optimizations
    - Since depth depends of program



## Complete Fragment Program Instruction List (1)

- Add & multiply instructions
  ADD, DP3, DP4, LRP, MAD, MOV, SUB, X2D
- Texturing instructions
  TEX, TXD, TXP
- Partial derivative instructions
  DDX, DDY
- Math functions
  COS, EX2, FLR, FRC, LG2, POW, RCP, RSQ, SIN
- Set on instructions
  SEQ, SFL, SGE, SGT, SLE, SLT, SNE, STR

## Complete Fragment Program Instruction List (2)

- **Graphics-oriented instructions**
  **DST, LIT, RFL**
- **Minimum / maximum instructions**
  **MAX, MIN**
- **Pack instructions**
  **PK2H, PK2US, PK4B, PK4UB, PK4UBG**
- **Unpack instructions**
  **UP2H, UP2US, UP4B, UP4UB, UP4UBG**
- **Kill instruction**
  **KIL**

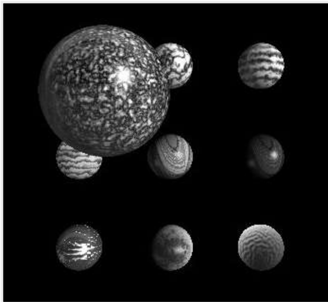


## NV_fragment_program Examples (1)

Utah teapot with procedural generated star pattern.
Credit: Jacopo Pantaleoni



## NV_fragment_program Examples (2)

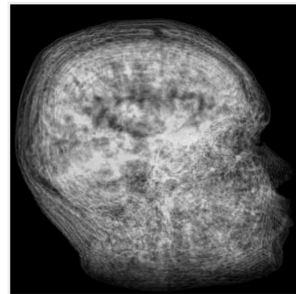


Various procedural shaders (marble, wood, bumpy velvet, brushed anisotropic metal).

Credit: Jacopo Pantaleoni



## NV_fragment_program Examples (3)



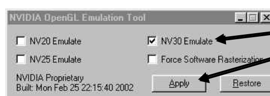
“Single-pass” volume rendering – the above image is rendered as a single quad; a lengthy fragment program computes the volume rendering integral for each fragment representing a ray through the volume (stored as a 3D texture)

Credit: Jacopo Pantaleoni



## Developing for NV30 *Today*

- **NVIDIA has an “NV30 Emulation” driver**
  - **This driver emulates all major NV30 OpenGL extensions, including**
    - **NV_vertex_program2**
    - **NV_fragment_program**
    - **NV_stencil_two_side**
    - **EXT_blend_func_separate, etc**
  - **Available for NV30 partners**




1. Click this on
2. Then Apply



## Cg and NV30 OpenGL Extensions

- **Cg compiler will target NV30 profiles**
  - **Will generate NV_vertex_program2 & NV_fragment_program code for you**
- **Cg is the most productive and efficient to make use of NV30**

## Conclusions

1. NV30 is the fastest & most functional OpenGL implementation ever
   - OpenGL exposes ***all*** NV30 features
   - Well beyond even what DirectX 9 exposes
2. Cg automatically targets this functionality
   - Makes optimal use of NV30 programmability a snap
3. Standards oriented
   - NV30 adopts ARB/EXT functionality when available
   - NV extensions are used when NV30 is far ahead of what other vendors offer
   - Working for OpenGL 1.X & 2.X inclusion

NVIDIA Proprietary