NVIDIA®

Canonical Shaders for Optimal Performance

Sébastien Dominé

Manager of Developer Technology Tools

Agenda

- Introduction
- FX Composer 1.0
- High Performance Shaders
  - Basics
  - Vertex versus Pixel
  - Talk to your compiler
  - Clever tricks
  - Texture look ups
- Case Study
- Conclusion
- Q&A

NVIDIA.

Introduction

- High Level Shading Languages:
  - Powerful in getting your shader up and running
  - Allows TD to quickly reach targeted look and feel
  - Compilers do a good job at optimizing code
  - Is this heaven or what?

NVIDIA.

Introduction

- Not quite...there are a lot of pitfalls:
  - Too expensive computation at the wrong stage
  - Not enough hints given to the compiler:
    - Complex HW
    - Shader writers know the computation – talk to your compiler
  - Not using clever tricks that help compilers
  - Not always verifying what comes out
    - Look at the assembly – fxc;FX Compiler/Shader Perf
    - Look at the scheduling – FX Compiler/Shader Perf
    - Look at the fps of unique fullscreen shaders

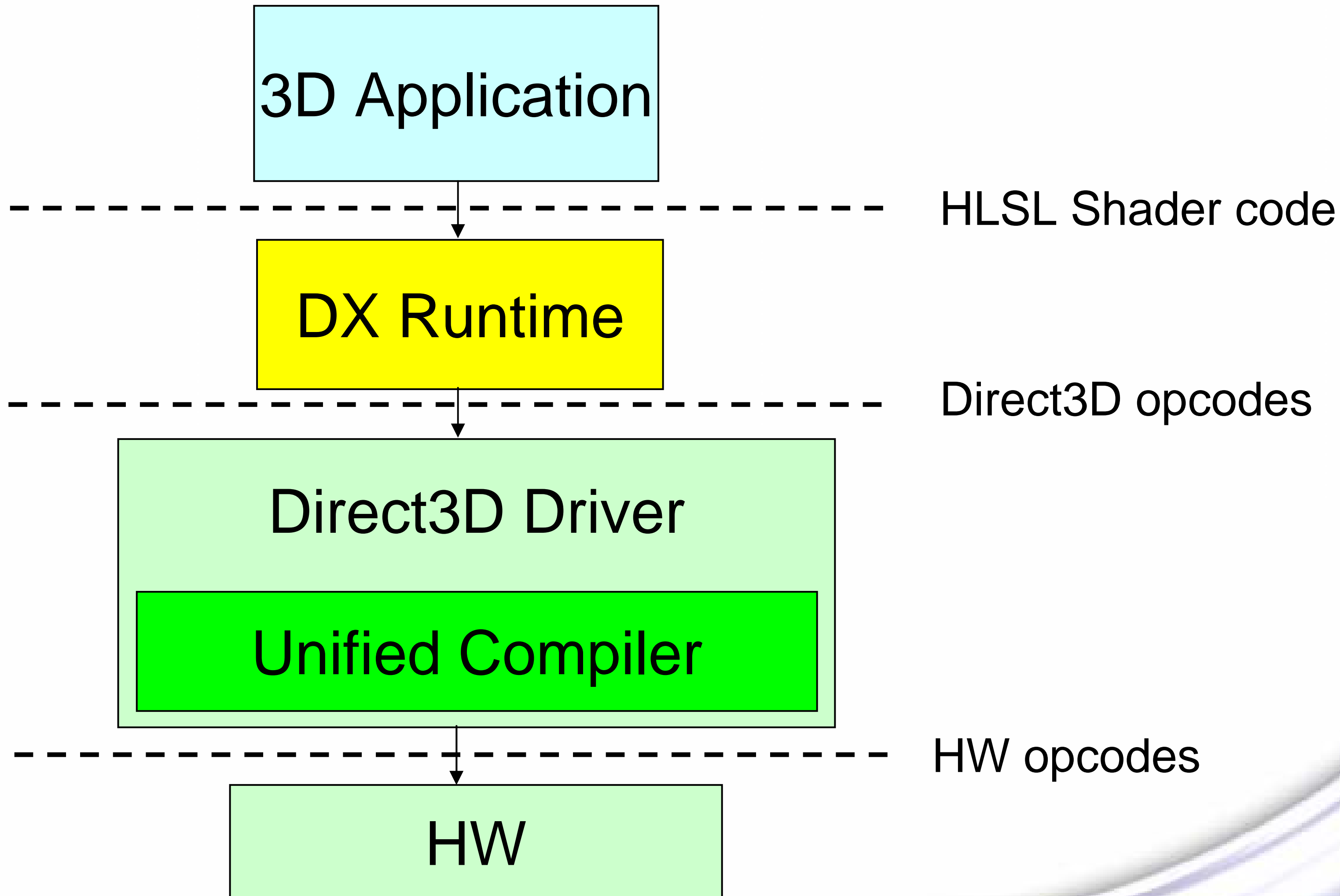
NVIDIA.

FX Composer

- Integrated IDE for HLSL FX development
- Simulated shader scheduling on nv3x family
- Disassembly of vertex and pixel shaders
- Bakes textures from HLSL code
- Allows render to texture effects
- HLSL Intellisense
- Allows scene import from .x and .nvb files
- Supports animation, lights, skinned meshes, etc...
- Allows pluggable geometry modifiers (fins, ...)
- Project files .fxcomposer
- Fxmapping.xml – custom semantic/annotation mapping

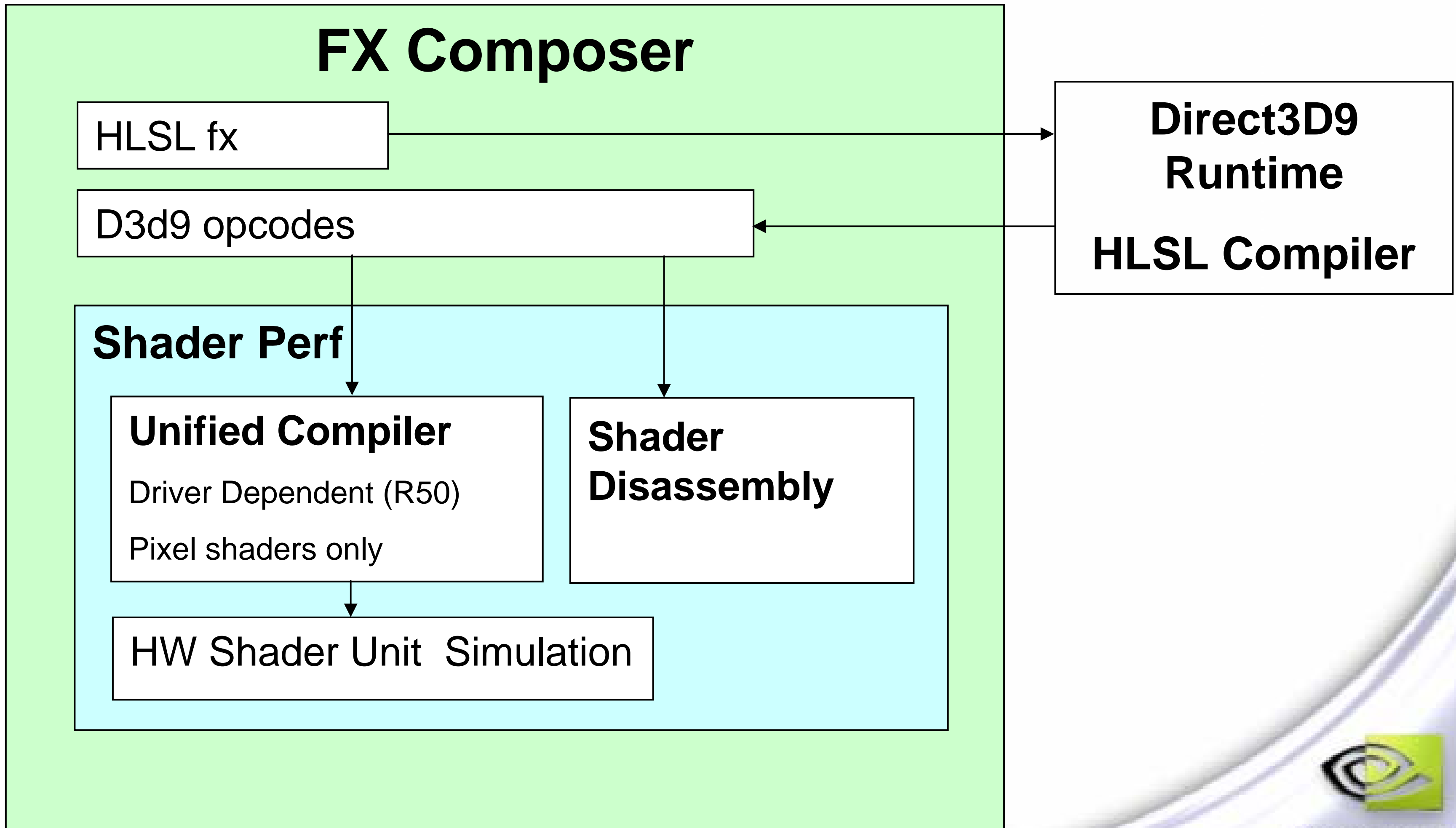
NVIDIA.

Driver Model



NVIDIA.

FX Composer



FX Composer

- **Demo**

High Performance Shaders

- Basics
- Vertex versus Pixel
- Talk to your compiler
- Clever tricks
- Texture look ups

High Performance Shaders

- How many cycles does *normalize* take?
  - Using ps2.0 math: 2-3 cycles
    - High Level: `normalize(V);`
  - Using texture fetch (signed normalization cube map): 1 cycle
    - High Level: `texCube(S, V);`
  - Using ps1.x assembly: 1 cycle

```
dp3 r0, v0_bx2, v0_bx2
mad r0, v0_bias, 1-r0, v0_bx2
```

NVIDIA.

The Basics 1/2

- Develop your code in the easiest way that makes it work
  - Then optimize it
- Make a budget
  - Target a reference platform:
    - Target resolution @ *x* Hz, *y* XAA, Aniso.
  - Identify key points of interest
  - Use more complex shaders for them

NVIDIA.

The Basics 2/2

- Move constant computations off of the GPU and on to the CPU
- Example:
  - Computing Material Color * Light Color at each vertex
  - Computing 4x3 matrix transpose in the pixel shader
    - Watch out for row vs. column major matrices

Vertex versus Pixel 1/1

- Move linear computations out of the pixel shader and into the vertex shader

- Example:
  - Move from world space to light space for attenuation
    - (Light.pos – vertex.pos) / Light.Range

  - Moving into tangent space usually belongs per-vertex
    - Unless you are doing per-pixel reflection into a cubemap – cubemap is defined in world space

Talk to your compiler 1/3

- Use lowest possible precision that will do the job
- Fixed, half ,float (24 or 32 bit) in that order
- Use _pp modifiers in ps_2_0 asm or half type in HLSL
- Example: tex2d * diffuse_color
  - Can be done in fixed
  - No advantage to higher precision

Talk to your compiler 2/3

- Pick the right profile for the right complexity
- Use lowest possible pixel shader version that will do the job
- Remember that DirectX 9 *still* supports ps1.x
  - If you have a shader that fits in 1.x, use it
  - Try compiling HLSL for ps 1.x
  - Use ps2.0 when it can't be achieved at the level of quality you are requiring

- Example:
  - Simple diffuse lighting * texture

NVIDIA.

Talk to your compiler 3/3

- Compile with ps_2_a target – alternate model for compilation
  - Closely matches GeForce FX hardware
- Example: Dependent texture fetches under ps_2_0 target generate sub-optimal assembly for ps_2_a hardware
- Look at assembly output of fxc or FX Composer – does the number of instructions match your expectations? What about the scheduling?

Clever Tricks 1/6
Reflection Vector Optimizations

- There's no substitute for algebraic savings
- If a reflection vector is used to index into a cube map, its length doesn't matter
- Standard reflect function can be optimized from:
  - R = 2*N*[dot(N,V)/dot(N,N)] – V

  to

  - R = 2*N*dot(N,V) – dot(N,N)*V
- If N is already normalized, you can remove the dot(N,N):
  - R = 2 * N * dot(N,V) – V

nVIDIA.

Clever Tricks 2/6
Other Normalization Optimizations

- Other examples:
  - Normal map lookups can be safely left un-normalized
  - Try leaving L vector un-normalized per-pixel for diffuse bump mapping

Clever Tricks 3/6
Computing Normalize Efficiently

- *Normalize(N)* dot *Normalize(L)* can be computed more efficiently:
- The usual way: (N/|N|) dot (L/|L|)
- Two rsq computations!
- Better:

  = (N dot L) / (|N| * |L|)

  = (N dot L)/ (sqrt((N dot N) * (L dot L))

  = (N dot L) * rsq ((N dot N) * (L dot L))

Clever Tricks
Clamping Efficiently

- C/C++ don’t have native min() and max() functions
- But GPU high-level languages do
- Use max() and min() instead of if-then-else
  - max(A,B) != (A>=B?A:B)
- Why?
- Because in this case, fxc uses a cmp instruction to handle the if-then-else
- cmp is not a native NV3X instruction
- But min and max are native NV3X instructions

Clever Tricks 5/6
Lerps

- lrp instruction is more expensive than other blending operations on NV3X
- Look for opportunities to simplify, or remove
- Note that:
  - lerp(a,b, 1-f) == lerp(b, a, f)
  - Compiler should find this and optimize, but may not yet do that

Clever Tricks 6/6
Vectorize Scalar Constants

- Make sure scalar constants don’t end up misleading the compiler to use extra temporary registers
  - Look at FX Composer/Shader Perf for a report on how many temporary registers end up being used

Texture Lookups 1/3

- Texture lookups can be faster than math in many cases
- Free clamping, free filtering
- Examples :
  - Free [0..1] clamping for N.L and N.H and self-shadow term
  - A8R8G8B8 Normalization Cubemap
  - L.N, N.H map in A8L8 2D Texture
  - L.N, N.H, $N.H^k$ in A8L8 3D Texture
  - Attenuation via 1 or 2 Texture lookups

NVIDIA.

Texture Lookups 2/3
Vector Normalization

- Normalize using math or normalization cube map lookup?
  - For 1.x shaders, can use cube map or math (Newton-Raphson approximation)

```
dp3 r0, v0_bx2, v0_bx2
mad r0, v0_bias, 1-r0, v0_bx2
```

  - For higher quality, use 2 16-bit signed cube maps for (x,y) and (z). 8-bit cubemaps had artifacts
  - In general, signed textures are either the same speed or faster than unsigned
    - ps.2.0 HW doesn't always have a free _bx2

NVIDIA.

Texture Lookups 3/3
Lighting - Specular Power

- Exponentiation: math or texture lookup?
  - 1d texture: Replace power function with texture lookup – for fixed exponent. Less flexible, but fast
  - 2d texture: Second coordinate is exponent
  - 3d texture: (u,v,w) = (L dot N, H dot N, spec exponent). Clamping on L dot N for free.
- Texture lookup, in general, faster than math
- Texture lookups less flexible than math
  - Changing exponent per pixel for 3d texture approach will be more expensive – texture cache thrashing
- Texture lookups are more flexible for artists
  - They can paint them well!

NVIDIA.

Unique NP2 Texture Coordinates

- All non-power-of-two (NP2) textures and most NP2 render targets are addressed using image mode coordinates (0...width, 0...height)
- Texture coordinates need to be scaled by texture dimensions
- Driver will try to insert this scale into vertex shader

NVIDIA.

Unique NP2 Texture Coordinates

- Problem:
  - If you have one set of texture coordinates used for multiple textures (allowed by ps_2_*)
  - Driver must insert scale into pixel shader!
  - Even if the textures are the same size
- Solution:
  - Use separate texture coordinates for each NP2 texture
  - Driver will correctly detect that they can be scaled by the vertex shader

nVIDIA.

Per-Pixel Lighting Case Study

- Well understood set of solutions
- Different approaches for achieving the same look
- Diffuse Lighting: k_d * (L dot N)
- Specular Lighting:
  - k_s * (L dot R)^{spec} [Phong] where R is reflected E
    - Cheaper for many lights
  - k_s * (H dot N)^{spec} [Blinn]
    - Cheaper for one light

NVIDIA.

How to compute L, N and H?

- All *should* be normalized
- But, N will be close to normalized if from a tangent space normal map
- L can be normalized per-vertex
  - Use vertex shader to put into light space for attenuation
  - Optionally re-normalize per-pixel
- H can be computed per-vertex
  - But can’t be linearly interpolated per-pixel
  - But on a well-tessellated mesh will often work fine anyway
  - Renormalizing per-pixel not the correct vector direction but still visually plausible

NVIDIA.

Per-Pixel Lighting Experimental Results

- float to half gave a performance increase without quality loss
- normalize() -> 16-bit cubemaps gave further perf increase without quality loss. 8-bit cubemaps had artifacts
- Clamp with a texture even faster
  - Both for N.L and self-shadowing term for N.H
- If light is not too close to a triangle, H per-vertex looks good
  - Otherwise, compute per-pixel

Per-Pixel Lighting Summary

- There are many variations of even simple diffuse and specular lighting techniques
  - R.L vs N.H
  - Per-object, Per-Vertex or Per-Pixel Exponent
  - Gloss Factors/Specular maps
  - Math, Texture or Per-Vertex Normalization
  - Per-Vertex, Per-Pixel or Texture-based attenuation
- The ideal method is very app-dependent
  - But more flexibility costs performance
  - The more you do per-vertex, the more quantity and complexity of lighting you can afford

NVIDIA.

Case Study

- FX Composer tutorial
  - Implement classic lighting model: “A Reflectance Model for Computer Graphics” by Cook & Torrance, *SIGGRAPH ‘81*.
  - Bake textures for faster computation
  - Use half precision
  - Give more hints to the compiler

Conclusion

- Select appropriate unit for ops: CPU, Vertex, Pixel
  - CPU for constants, Vertex for linear calculations
- Select appropriate precision
- Use lowest pixel shader version that works
- You only get so many pixel shader cycles per frame
  - Use them for visually interesting effects
    - Per-Pixel Bump Maps
    - Per-Pixel Reflections
  - Give yourself more cycles for effects by not spending them on unneeded precision and calculation

NVIDIA.

Questions?

- If you see un-optimal code generated by fxc or FX Composer/Shader Perf, send us e-mail
  - We'll get it fixed
  - Compilers are still evolving
- Send mail to:
  - devrelfeedback@nvidia.com
  - fxcomposer@nvidia.com

NVIDIA.