6800 LEAGUES UNDER THE SEA

Shader Model 3.0, Best Practices

Phil Scott
Technical Developer Relations, EMEA
Overview

- Short Pipeline Overview
- CPU Bound – new optimization opportunities
- Obscure bits of the pipeline that can trip you up
- Pixel Bound – new optimization opportunities
- 3.0 shader performance characteristics
Bottlenecks

Limits the speed of the pipeline

CPU / Fragment – focus of this talk
Still the two most likely cases these days in modern apps:

- CPU bound
  - Becomes more and more likely the faster GPUs get
- Fragment bound
  - Becomes more and more likely the longer shaders get

Neither of these trends are likely to change soon

Some new weapons for combating these

- Instancing
- HW Shadow Maps
- Shader model 3.0
DirectX 9 Instancing API

What is it?
- Allows you to avoid DIP calls and minimise batching overhead
- Allows a single draw call to draw multiple instances of the same model

What is required to use it?
- Microsoft DirectX 9.0c
- VS/PS 3.0 hardware
Why use instancing?

- Speed. Still the single most common performance suck in most games today is draw calls

- Yeah. Yeah. We all know draw calls are bad
  - But world matrices and other state often force us to separate draw calls

- The instancing API pushes the per instance draw logic down into the driver
  - Saves DIP call overhead in both D3D and Driver
  - Allows the driver to ensure minimal state changes between instances
When to use instancing?

- Scene contains many instances of the same model
  - Forest of Trees, Particles, Sprites

- If you can encode per instance data in 2\textsuperscript{nd} streams.
  I.e instance transforms, model color, indices to textures/constancts.

- Less useful if your batch size is large
  - >1k polygons per draw
  - There is some fixed overhead to using instancing
How does it work?

**DX Instancing API** makes use of an extended vertex stream frequency divider API.

- **Primary stream** is a single copy of the model data.
- **Secondary streams** contain per instance data and stream pointer is advanced each time the primary stream is rendered.

Uses `IDirect3DDevice9::SetStreamSourceFreq` entry point.
Simple Instancing Example

- 100 poly trees
  - Stream 0 contains just the one tree model
  - Stream 1 contains model WVP transforms
    - Possibly calculated per frame based on the instances in the view
  - Vertex Shader is the same as normal, except you use the matrix from the vertex stream instead of the matrix from VS constants

- If you are drawing 10k trees that’s a lot of draw call savings!
  - You could manipulate the VB and pre-transform vertices, but it’s often tricky, and you are replicating a lot of data
Some Test Results

- Test scene that draws 1 million diffuse shaded polys
- Changing the batch size, changes the # of drawn instances
- For small batch sizes, can provide an extreme win as it gives savings PER DRAW CALL.
- There is a fixed overhead from adding the extra data into the vertex stream
- The sweet spot will change based on many factors (CPU Speed, GPU speed, engine overhead, etc)

![Instancing versus Single DIP calls](chart)
Instancing - More test results

Instancing Method Comparison
(Note: % is relative to HW instancing in each group)
[28 poly mesh]

- Single Draw Calls
- Replicated 2 Stream Instancing
- Static 2 Stream Instancing
- Hardware Instancing
- Static Pretransformed VB

# Instances:
- 100
- 1000
- 5000
- 10000
- 20000

FPS (relative to HW Instancing):
- 0.00%
- 20.00%
- 40.00%
- 60.00%
- 80.00%
- 100.00%
- 120.00%
Instancing Demo

- Space scene with 500+ ships, 4000+ rocks
- Complex lighting, post-processing
  - Some simple CPU collision work as well
- Dramatically faster with instancing
Instancing – Caution!

- You can quickly become “attribute bound” due to the extra data that needs to be fetched per instance.
  - This explains the slowdown at the limit in the previous app.
- Make sure you vertex cache optimize.
  - Remember, a hit in the cache saves all previous work, including attribute access.
- Pack input attributes as tightly as possible.
  - Even if it requires a little vshader work to unpack, probably worth it.
  - Be careful of things in the input stream that can be constants or easily derived in the vshader.
What are attributes?

- Bits of vertex data fetched
- Positions
- Normals
- Texture coordinates
- etc...
Obscure Pipeline Bits

For parts of the pipeline like vertex fetching and triangle setup, the old advice was always “don’t worry about it”

No longer true!

This is not because these parts have become slower, everything around them just keeps getting exponentially faster

Vertex fetch (attribute access) bound – Instancing

Setup bound – Stencil Shadow Volumes

Two sided stencil

External triangles for extrusion
Many people developing new engines are already using R32F or R16F shadow maps

- Multiple jittered samples for higher quality / soft edges

NVIDIA Hardware Shadow Maps can just “drop in” to these engines

- Same setup, same pipeline as any shadow map technique
Percentage-closer filtering is “free” on these
- Use \( \frac{1}{4} \) the taps for performance, or get 4x the quality for the same performance!

In D3D, simply create a depth format texture (like D3DFMT_D24X8) and render to it
- When sampled, the shadow map comparison happens automatically

In OpenGL, use TEXTURE_COMPARE_MODE_ARB with COMPARE_R_TO_TEXTURE
3.0 Shaders Overview

- 3.0 shaders can help with both CPU boundedness and GPU boundedness
  - Improved batching / fewer passes
  - Early-outs with dynamic branching

- Gory performance details of 3.0 features
  - Vertex and Pixel
Many engines have a primary lighting shader that does something like this:

```glsl
half3 diffuseTex = tex2D(DiffuseSampler);
half3 normalTex = tex2D(NormalSampler);
half shadow = tex2D(ShadowMap);
// do complex lighting
// output result
```
ps.3.0 – Better Batching / Fewer Passes

- A few possible perf pitfalls

  - One pass per light – means more DrawPrimitive() calls, worse batching
  - You have to refetch the diffuse map and normal map for every pass
    - With 16X aniso, this can be very expensive
  - Memory bandwidth / transform required for each pass
ps.3.0 – Better Batching / Fewer Passes

Solution: branching in the pixel shader!
Loop over a number of lights, accumulate lighting in the shader
- Fetches from textures only once
- Fewer batches
- Less transform / attribute fetching, less bandwidth
ps.3.0 – Potential Gotchas

- May require more interpolators
  - Good thing ps.3.0 has 10 high-precision interpolators

- May require more samplers
  - A shadow map per light

- Doesn’t really work with stencil shadow volumes
Early out is when you do a dynamic branch in the shader to reduce computation.

Some obvious examples:
- If in shadow, don’t do lighting computations.
- If out of range (attenuation zero), don’t light.
- Obviously these apply to vs.3.0 as well.

Next – a novel example for soft shadows.
ps.3.0 – Soft Shadows

fps = 37.7  mode = NV4X
jitter = yes  fwidth = 8.0
ps.3.0 – Soft Shadows

- Works by taking 8 “test” samples from the shadowmap
  - If all 8 are in shadow or all 8 are in the light we’re done
  - If we’re on the edge (some are in shadow some are in light), do 56 more samples for additional quality
- 64 samples at much lower cost
ps.3.0 – Soft Shadows

- On GeForce6 GPUs, this demo runs more than twice as fast using dynamic branching vs. doing all 64 samples all the time.

- Combined with hardware shadow maps, makes real-time cinematic shadows a reality.
3.0 Shaders Perf – Pixel Nitty-Gritty

- Pixel shader flow control instruction costs:
  - Not free, but certainly usable

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cost (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if / endif</td>
<td>4</td>
</tr>
<tr>
<td>if / else / endif</td>
<td>6</td>
</tr>
<tr>
<td>call</td>
<td>2</td>
</tr>
<tr>
<td>ret</td>
<td>2</td>
</tr>
<tr>
<td>loop / endloop</td>
<td>4</td>
</tr>
</tbody>
</table>

- Additional cost associated with divergent branches
3.0 Shaders Perf - Pixel

GeForce 6 series LOD texture instructions:
- texldb – full perf
- texldl – full perf
- texldd – much lower perf
  - Factor of 10

- texldl has the additional benefit of not requiring the hw to calculate derivatives for LOD
  - Means you can branch over them dynamically

- With GeForceFX, all of these are lower perf
3.0 Shaders Perf - Pixel

Question: Does _pp (fp16) still matter in the pixel shader?

Answer: YES
- Critical for GeForceFX performance
- Even helps GeForce6:
  - Less register pressure, better hiding of texture latency
  - Fast fp16 normalize (nrm_pp)
3.0 Shaders Perf - Vertex

- Vertex flow control behaves a little differently
  - Branch instructions have a fixed cost of ~1 cycle
  - Divergence doesn’t matter (MIMD)

- The one big gotcha with vertex is VTF...
3.0 Shaders Perf - VTF

- Vertex Texture Fetch has potentially large latency
  - Equivalent to ~20 instructions

- So multiple dependent texture fetches will be slow
  - Using VTF to emulate a larger constant RAM is a bad idea in this generation of hw

- But, this is per-vertex, so certainly usable for many effects
  - See dynamic water displacement demo in NVSDK
Conclusion

**Complex pipeline**

- Some stages that used to be overlooked can bite you now that shading power has been increased so dramatically

**Most popular culprits still shading and CPU, however**

- A combination of instancing and 3.0 shaders can overcome these bottlenecks
Questions?

Phil Scott (pscott@nvidia.com)