Clever Shader Tricks

Nick Thibieroz
European Developer Relations
MrT@ati.com
Introduction

- Harnessing the power of dynamic flow control
- The power of maths
- Skyboxes
- Understanding Z
Harnessing the Power of Dynamic Flow Control
Dynamic Flow Control is the essential feature of Shader Model 3.0 (and 4.0) & GLSL

- Ability to loop or branch within a pixel shader
  - Branch comes from temporary register
  - Loop iteration comes from integer constant

- Dynamic vs static branching
  - Different beasts, leading to different results

- DFC enables a different code path to be executed
  - Allows more intuitive shader writing
  - E.g. two-sided lighting, LOD-based branching, etc.

- DFC allows skipping of instructions
  - This is where the performance savings are!
  - But there are conditions...
• GPUs process operations in parallel
• Multiple threads are shared across shader cores
  • For best efficiency don’t use too many or too small branches
• Thread size is 16 pixels on X1800, 48 pixels on X1900
  • Fine grain parallelism
• Shaders whose pixels take different paths within the thread size (branching granularity) execute both parts of the branch
  • This can be costly if it happens too often!
• DFC is best used on spatially-coherent branches

• Good examples
  • Chunky shadow areas
  • Light falloff
  • Terrain material blending
  • Screen-space shader LOD branching

• Bad examples
  • Sparse shadow areas (e.g. grid)
  • Specular lighting with high-frequency normal maps
• DFC allows the skipping of texture instructions
• Leads to performance increase in texture-limited cases
• Potentially allows huge memory bandwidth savings

• Examples:
  • Terrain material blending
  • Shadow map samples
  • Materials when pixel is not facing the light (or in shadow)
  • Detail textures
  • Etc.
Shader LOD branching revolves around two ideas:
- Run complex shader operations for pixels at close range
- Reverting to simpler operations for pixels further away

Helps performance, but also with shader aliasing.

Examples:
- Parallax occlusion mapping / normal mapping
- Detail maps / no detail map
- Complex lighting model / simple lighting model
- Shadows (fading, self-shadowing)
- Etc.

Optimal spatial coherency for efficient branching.
Use a custom MIPMap LOD calculation to determine code path to apply

- Calculate maximum distance in texels of adjacent pixels
  
  ```
  float2 dxSize = ddx(i.texCoord * vTextureDims);
  float2 dySize = ddy(i.texCoord * vTextureDims);
  float2 Delta = sqrt(dxSize*dxSize + dySize*dySize);
  ```

- Use maximum distance between the two generated values
  
  ```
  float minDelta = max(Delta.x, Delta.y);
  ```

- Translate this distance into MIPMap LOD
  
  ```
  float fMipLevel = log2(minDelta);
  ```

- Compare with desired MIPMap LOD for branching
  
  ```
  if (fMipLevel <= (float)nLODThreshold)
  {
    ComplexColor = RunComplexShader();
  }
  ```

- Blend between complex and simple shader LOD to smooth transition, e.g.:

  ```
  if (fMipLevel > ((float)nLODThreshold - 1.0))
  {
    SimpleColor = RunSimpleShader();
  }
  FinalColor = lerp(ComplexColor, SimpleColor, saturate(fMipLevel-((float)nLODThreshold-1.0)));
  ```
Why am I not getting good DFC performance?

• Potential reasons...

• You’re not pixel-shader limited
  • Make sure you are before profiling pixel shader performance!

• Branching granularity is poor
  • Test this by outputting a plain colour where branches occur

• You have too many or too small branches
  • Threads are limited so apply branching efficiently

• You’re running on old nVidia hardware 😊
  • 6800 series is embarrassingly slow, 7800 series is ~ OK’ish
The Power Of Maths
The Power of Maths

- 48 pixel shader cores in X1900
  - 16 textures per clock
- ALU:TEX Ratio will only go up from here on
- Use more maths than texture instructions in shader
  - POM, atmospheric fog, PTM, procedural textures, complex lighting models, etc.
- Get rid of:
  - Cube normalization maps
  - Specular/falloff lookups
  - Any texture lookup that can be done with maths
• Understand the impact of texture filtering
• Each 32-bits texture instruction will take a number of clocks
  • Bilinear filtering = 1 clock
  • Trilinear filtering = 2 clocks
  • Anisotropic filtering = X clocks (depends on number of taps)
• 64-bits format and volume textures take twice as many clocks
  • This is cumulative
  • E.g. a 3D texture in 16161616 format with trilinear filtering takes 8 clocks
• Take this into consideration when writing shaders for an ideal ALU:TEX ratio
  • Target a minimum of 6:1 (higher is better)
  • Compiler will optimize maths instructions but not textures!
“Easy to say – I still have older GPUs to take care of!”
  • You’re right!

Older GPUs may not benefit from a high ALU:TEX ratio
  • But all future GPUs will

High-end versions of your shaders should always use ALU-heavy shaders

Shaders for lower-end GPUs could use a lesser ratio
  • Use hardware detection to detect lower-end GPUs (or let the user choose)
  • Don’t do the opposite as newer GPUs won’t be in your list

For practical shader management use #ifdef in shader code
  • E.g. replace cube normalization map lookup with normalize()
  • ...and specular lookups with power functions etc.
Skyboxes
• MrT’s rant: “I pity the fool who renders their skybox first!” 😊

• For best performance **RENDER YOUR SKYBOX LAST**
  • Not really “last”, but after all other opaque primitives
  • Skyboxes are not the way to clear the colour buffer
    • Main colour buffer should always be explicitly Clear()ed
  • This is what happens when the skybox is rendered first:
    • The pixel shader is executed for all skybox pixels
    • All skybox pixels are then written to the colour buffer
  • This is what happens when the skybox is rendered last:
    • The pixel shader is executed for skybox pixels that pass the Z test
    • Only pixels passing the Z test are written to the colour buffer
  • Only a fraction of the skybox will be visible after the rest of scene
    • The sky may not even be visible at all!
• Concerned about clipping issues?
  • Several methods to ensure a skybox is correctly rendered last

• Depth partitioning
  • Part of the D3DVIEWPORT9 structure in DX9
  • Part of the D3D10_VIEWPORT structure in DX10
  • glDepthRange() in OpenGL
  • Set a Z range of \([1, 1]\) for skybox rendering

• Vertex shader trick
  • Set output Z to \(W\)
  • Perspective divide will give you a Z screen of 1.0
  • Can also be done in projection matrix

• Depth clamping (DX10)
Understanding Z

Knowledge is power...
• Understanding hardware Z optimizations is key to efficient rendering

• **Fast Z Clears**
  • Z and stencil buffer are contained in the same surface
  • Z can only be fast-Clear()ed if cleared in conjunction with stencil
  • Otherwise a read/modify/write operation occurs

• **Compressed Z buffer**
  • Z buffer values are block-compressed to save Z bandwidth
  • Lossless compression
  • Main Z buffer automatically compressed for performance
  • Depth textures are not compressed (DF16, DF24)
Understanding Z - Hierarchical Z Culling

• Keeps the max or min Z value per block in on-chip memory
  • Depends on depth compare more

• Incoming triangles are split in blocks

• Depth compare is performed per-block using the Z stored for this block
  • If incoming Z value is greater/smaller than block Z then the triangle portion is hidden
  • Else triangle is split into smaller blocks

• Allows fast Z culling of whole (or portions of) triangles
  • Works with everything except when pixel shader outputs depth!
  • Can break if Z compare modes are reversed

• Render alpha-tested/texkill primitives after opaque ones
  • This increases the chance of those being rejected by HiZ culling

• Consider depth-only pass
Hierarchical Z already performs gross Z culling using on-chip HiZ values.

Early Z testing occurs as well on the full-resolution Z buffer:
- Allows rejection of pixels failing the depth test prior to shading them.

Early Z doesn’t work in three cases:
- When shader outputs depth
- When alpha-test or texkill is used
- When alpha to coverage is used

Disable those whenever possible.

Front-to-back sorting helps.

---

D3D pixel pipeline:
```
PS | Depth Read/Write
```

GPU pixel pipeline:
```
Depth Read/Write | PS
```

GPU pixel pipeline when AT/Texkill/A2C/Depth output is used:
```
Depth Read/Write | PS | Depth Read/Write
```
Case Study: Depth-only Pass

- Lay out the contents of the depth buffer in an initial pass
  - Ensures an overdraw of 1 for subsequent opaque pixels
    - Helps with high-overdraw situations
    - ...especially when longer shaders are involved
  - Requires an extra geometry pass
- Can be optimized heavily though!
  - Color writes disabled = faster depth-only throughput
  - Use a reduced vertex structure (no need for binormal, tangent, color, texcoords)
  - Use a reduced vertex shader (transform only - including skinning when needed)
- Alpha testing optimizations
  - Use a reduced pixel shader for this: e.g. only fetch cut-out texture
  - Disable alpha test after depth pass (change Z compare mode to EQUAL for those)
• Dynamic Flow Control is a very powerful feature
  • Convenience
  • Performance
  • Use it!
• Ensure your shaders predominantly use maths instructions
  • Target 6:1 ALU:TEX minimum
• Z recommendations
  • AT/Texkill/depth output with moderation
  • AT/Texkill after opaque
  • Avoid inverting Z compare mode whenever possible
  • Depth pass?
  • Front-to-back sorting
• Did I mention to render your skybox last? 😊