DirectX 10 Performance

Ignacio Llamas, NVIDIA
illamas@nvidia.com

Nicolas Thibieroz, AMD
nicolas.thibieroz@amd.com
Outline

- General DX10 API usage
  - Designed for performance
  - Batching and Instancing
  - State Management
  - Constant Buffer Management
  - Resource Updates and Management
  - Reading the Depth Buffer
  - MSAA

- Optimizing your DX10 Game
  - or how to work around GPU bottlenecks

- IHV-specific optimizations
Color Guide For IHV-specific Advice

AMD

NVIDIA
DX10 Runtime and Driver. Designed for Performance

- DX10 validation moved from runtime to creation time
  - Only basic error checking at runtime
- Immutable state objects
  - Can be pre-computed and cached
  - Subset of command buffer at creation time
- Vista driver model delegates scheduling and memory management to OS
  - Pro: more responsive system, GPU sharing across apps
  - Con: harder to guarantee performance if multiple apps share the GPU
    - Fullscreen mode should be fine
Batch Performance

- The truth about DX10 batch performance
- “Simple” porting job will not yield expected performance
- Need to use DX10 features to yield gains:
  - Geometry instancing or batching
  - Intelligent usage of state objects
  - Intelligent usage of constant buffers
  - Texture arrays
Geometry Instancing

- Better instancing support in DX10
- Use “System Values” to vary rendering
  - SV_InstanceID, SV_PrimitiveID, SV_VertexID
  - Additional streams not required
  - Pass these to PS for texture array indexing
  - Highly-varied visual results in a single draw call

Watch out for:
- Texture cache trashing if sampling textures from system values (SV_PrimitiveID)
- Too many attributes passed from VS to PS
- InputAssembly bottlenecks due to instancing
- Solution: Load() per-instance data from Buffer in VS or PS using SV_InstanceID
State Management

- DX10 uses immutable “state objects”
  - Input Layout Object
  - Rasterizer Object
  - DepthStencil Object
  - Blend Object
  - Sampler Object

- DX10 requires a new way to manage states
  - A naïve DX9 to DX10 port will cause problems here
  - **Always create state objects at load-time**
  - Avoid duplicating state objects
  - Recommendation to sort by states still valid in DX10!
Constant Buffer Management (1)

- Probably a major cause of poor performance in initial naïve DX10 ports!
- Constants are declared in buffers in DX10

```c
cbuffer PerFrameConstants cbuffer SkinningMatricesConstants
{
    float4x4 mView;
    float fTime;
    float3 fWindForce;
    // etc.
}
```

- When any constant in a cbuffer is updated, the full cbuffer has to be uploaded to GPU
- Need to strike a good balance between:
  - Amount of constant data to upload
  - Number calls required to do it (== # of cbuffers)
Constant Buffer Management (2)

- Use a pool of constant buffers sorted by frequency of updates
- Don’t go overboard with number of cbuffers! (3-5 is good)

- Sharing cbuffers between shader stages can be a good thing

- Example cbuffers:
  - PerFrameGlobal (time, per-light properties)
  - PerView (main camera xforms, shadowmap xforms)
  - PerObjectStatic (world matrix, static light indices)
  - PerObjectDynamic (skinning matrices, dynamic lightIDs)
Constant Buffer Management (3)

- Group constants by access pattern to help cache reuse due to locality of access

Example:

```c
float4 PS_main(PSInput in)
{
    float4 diffuse = tex2D0.Sample(mipmapSampler, in.Tex0);
    float ndotl = dot(in.Normal, vLightVector.xyz);
    return ndotl * vLightColor * diffuse;
}
```

```c
cbuffer PerFrameConstants
{
    float4 vLightVector;
    float4 vLightColor;
    float4 vOtherStuff[32];
};
```

GOOD

```c
cbuffer PerFrameConstants
{
    float4 vLightVector;
    float4 vOtherStuff[32];
    float4 vLightColor;
};
```

BAD
Constant Buffer Management (4)

- Careless DX9 port results in a single $Globals cbuffer containing all constants, many of them unused

- $Globals cbuffer typically yields bad performance:
  - Wasted CPU cycles updating unused constants
    - Check if used: D3D10_SHADER_VARIABLE_DESC.uFlags
  - cbuffer contention
  - Poor cbuffer cache reuse due to suboptimal layout

- When compiling SM3 shaders for SM4+ target with D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY: use conditional compilation to declare cbuffers (e.g. #ifdef DX10 cbuffer{ #endif )
Consider `tbuffer` if access pattern is more random than sequential

- `tbuffer` access uses texture Loads, so higher latency but higher performance sometimes
- Watch out for texture-bound cases resulting from `tbuffer` usage

Use `tbuffer` if you need more data in a single buffer

- `cbuffer` limited to 4096*128-bit
- `tbuffer` limited to 128 megabytes
Resource Updates

- In-game destruction and creation of Texture and Buffer resources has a significant impact on performance:
  - Memory allocation, validation, driver checks

- Create all resources up-front if possible
  - During level load, cutscenes, or any non-performance critical situations

- At runtime: replace contents of existing resources, rather than destroying/creating new ones
Resource Updates: Textures

- Avoid `UpdateSubresource()` for textures
  - Slow path in DX10
    (think `DrawPrimitiveUP()` in DX9)
  - Especially bad with larger textures!

- Use ring buffer of intermediate
  D3D10_USAGETAGE STAGING textures
  - Call `Map(D3D10_MAP_WRITE,...)` with
    `D3D10_MAP_FLAG_DO_NOT_WAIT` to avoid stalls
  - If Map fails in all buffers: either stall waiting for Map
    or allocate another resource (cache warmup time)
  - Copy to textures in video memory
    (D3D10_USAGE_DEFAULT):
    - `CopyResource()` or `CopySubresourceRegion()`
Resource Updates: Buffers

To update a Constant buffer

- Map(D3D10_MAP_WRITE_DISCARD, ...);
- UpdateSubResource()

Recall full buffer must be updated, but with Map() CPU can skip parts that the shader does not care about. All the data must be uploaded to GPU though.

To update a dynamic Vertex/Index buffer

- Use a *large* shared ring-buffer type; writing to unused portions of buffer using:
  - Map(D3D10_MAP_WRITE_DISCARD,...) when full or if possible the first time it is mapped at every frame
  - Map(D3D10_MAP_WRITE_NO_OVERWRITE, ...) thereafter
- Avoid UpdateSubResource()
- not as good as Map() in this case either
Accessing Depth and Stencil

- DX10 enables the depth buffer to be read back as a texture
- Enables features without requiring a separate depth render
  - Atmosphere pass
  - Soft particles
  - Depth of Field
  - Deferred shadow mapping
  - Screen-space ambient occlusion
  - Etc.
- Popular features in most recent game engines
Accessing Depth and Stencil with MSAA

DX10.0: reading a depth buffer as SRV is only supported in single sample mode
- Requires a separate render path for MSAA

Workarounds:
- Store depth in alpha of main FP16 RT
- Render depth into texture in a depth pre-pass
- Use a secondary render target in main color pass

DX10.1 allows depth buffer access as Shader Resource View in all cases:
- Fewer shaders
- Smaller memory footprint
- Better orthogonality
MultiSampling Anti-Aliasing

- MSAA resolves cost performance
  - Cost varies across GPUs but it is never free
  - Avoid redundant resolves as much as possible
    E.g.: no need to perform most post-process ops on MSAA RT. Resolve once, then apply p.p. effects
- No need to allocate SwapChain as MSAA
  - Apply MSAA only to rendertargets that matter

- Be aware of CSAA on NVIDIA hardware:
  Certain DXGI_SAMPLE_DESC.Quality values will enable higher-quality but slightly costlier MSAA mode
Optimizing your DX10 Game

🔍 Use **PerfHUD** or **GPUPerfStudio** to identify bottlenecks:

🔍 **Step 1:** are you GPU or CPU bound
  🔍 Check GPU idle time
  🔍 If GPU is idle you are probably CPU bound either by other CPU workload on your application or by CPU-GPU synchronization

🔍 **Step 2:** if GPU bound, identify the top buckets and their bottlenecks
  🔍 Use **PIX** or **PerfHUD Frame Profiler** for this

🔍 **Step 3:** try to reduce the top bottleneck/s
If Input Assembly is the bottleneck

- Optimize IB and VB for cache reuse
  - Use ID3DXMesh::Optimize() or other tools
- Reduce number of vector attributes
  - Pack several scalars into single 4-scalar vector
- Reduce vertex size using packing tricks:
  - Pack normals into a float2 or even RGBA8
  - Calculate binormal in VS
  - Use lower-precision formats
- Use reduced set of VB streams in shadow and depth-only passes
  - Separate position and 1 texcoord into a stream
  - Improves cache reuse in pre-transform cache
  - Also use shortest possible shaders
If Vertex Shader is the bottleneck

- Improve culling and LOD (also helps IA):
  - Look at wireframe in debugging tool and see if it’s reasonable
  - Check for percentage of triangles culled:
    - Frustum culling
    - Zero area on screen
  - Use other scene culling algorithms
    - CPU-based culling
    - Occlusion culling

- Use Stream-Output to cache vertex shader results for multiple uses
  - E.g.: StreamOut skinning results, then render to shadowmap, depth prepass and shading pass
  - StreamOut pass writes point primitives (vertices)
    - Same index buffer used in subsequent passes
If Geometry Shader is the bottleneck

- Make sure `maxvertexcount` is as low as possible
  - `maxvertexcount` is a shader constant declaration ➔ need different shaders for different values
  - Performance drops as output size increases
- Minimize the size of your output and input vertex structures
- GS not designed for large-expansion algorithms like tessellation
  - Due to required ordering and serial execution
  - See Andrei Tatarinov’s talk on Instanced Tessellation
- Consider using instancing in current hardware
- Move some computation to VS to avoid redundancy
- Keep GS shaders short
- Free ALUs in GS because of latency
  - Can be used to cull geometry (backface, frustum)
If Stream-Output is the bottleneck

- Avoid reordering semantics in the output declaration
  - Keep them in same order as in output structure
- You may have hit bandwidth limit
  - SO bandwidth varies by GPU
- Remember you don’t need to use a GS if you are just processing vertices
  - Use ConstructGSWithSO on Vertex Shader
- Rasterization can be used at the same time
  - Only enable it if needed (binding RenderTarget)
If Pixel Shader is the bottleneck (1)

- Verify by replacing with simplest PS
  - Use PerfHUD / GPUPerfStudio
- Move computations to Vertex Shader
- Use pixel shader LOD
- Only use `discard` or `clip()` when required
- `discard` or `clip()` as early as possible
  - GPU can skip remaining instructions if test succeeds
- Use common app-side solutions to maximize pixel culling efficiency:
  - Depth prepass (most common)
  - Render objects front to back
  - Triangle sort to optimize both for post-transform cache and Z culling within a single mesh
  - Stencil/scissor/user clip planes to tag shading areas
  - Deferred shading
If Pixel Shader is the bottleneck (2)

- Shading can be avoided by Z/Stencil culling
  - Coarse (ZCULL / Hi-Z)
  - Fine-grained (EarlyZ)

- Coarse Z culling is transparent, but it may underperform if:
  - If shader writes depth
  - High-frequency information in depth buffer
  - If you don’t clear the depth buffer using a “clear” (avoid clearing using fullscreen quads)
If Pixel Shader is the bottleneck (3)

- Fine-grained Z culling is not always active

- Disabled on current hardware if:
  - PS writes depth (SV_Depth)
  - Z or Stencil writes combined with:
    - Alpha test is enabled (DX9 only)
    - discard / texkill in shaders
    - AlphaToCoverageEnable = true

- Disabled on current NVIDIA HW if:
  - PS reads depth (.z) from SV_Position input
    - Use .w (view-space depth) if possible
  - Z or Stencil writes combined with:
    - Samplemask != 0xffffffff
Any Shader is still the bottleneck (1)

- Use IHV tools:
  - AMD: GPUShaderAnalyzer
  - NVIDIA: ShaderPerf
- Be aware of appropriate ALU to TEX hardware instruction ratios:
  - 4 5D-vector ALU per TEX on AMD
  - 10 scalar ALU per TEX on NVIDIA GeForce 8 series
- Check for excessive register usage
  - > 10 vector registers is high on GeForce 8 series
  - Simplify shader, disable loop unrolling
  - DX compiler behavior may unroll loops so check output
- Use dynamic branching to skip instructions
  - Make sure branching has high coherency
Any Shader is *still* the bottleneck (2)

- Some instructions operate at a slower rate
  - Integer multiplication and division
  - Type conversion (float to int, int to float)

- Too many of those can cause a bottleneck in your code

- In particular watch out for type conversions
  - Remember to declare constants in the same format as the other operands they’re used with!
If Texture is the bottleneck (1)

- Verify by replacing textures with 1x1 texture
  - PerfHUD or GPUPerfStudio can do this

- Basic advice:
  - Enable mipmapping
  - Use compressed textures where possible
    - Block-compressed formats
    - Compressed float formats for HDR
  - Avoid negative LOD bias (aliasing != sharper)

- If multiple texture lookups are done in a loop
  - Unrolling partially may improve batching of texture lookups, reducing overall latency
  - However this may increase register pressure
  - Find the right balance
If Texture is the bottleneck (2)

- DirectX compiler moves texture instructions that compute LOD out of branches
  - Use SampleLevel (no anisotropic filtering)
  - SampleGrad can be used too, but beware of the extra performance cost

- Texture cache misses may be high due to poor coherence
  - In particular in post-processing effects
  - Modify access pattern

- Not all textures are equal in sample performance
  - Filtering mode
  - Volume textures
  - Fat formats (128 bits)
  - 64-bit integer textures
If ROP is the bottleneck: Causes

- Pixel shader is too cheap 😊
- Large pixel formats
- High resolution
- Blending
- MSAA
- MRT
- Rendering to system memory over PCIe (parts with no video memory)

- Typical problem with particle effects: little geometry, cheap shading, but high overdraw using blending
If ROP is the bottleneck: Solutions

- Render particle effects to lower resolution offscreen texture
  - See GPUGems 3 chapter by Iain Cantlay

- Disable blending when not needed, especially in larger formats (R32G32B32A32_FLOAT)

- Unbind render targets that are not needed
  - Multiple Render Targets
  - Depth-only passes

- Use R11G11B10 float format for HDR (if you don't need alpha)
If performance is *hitchy* or irregular

- Make sure you are not creating/destroying critical resources and shaders at runtime
  - Remember to warm caches prior to rendering

- Excessive paging when the amount of required video memory is more than available

- Could be other engine component like audio, networking, CPU thread synchronization etc.
OTHER IHV-SPECIFIC RECOMMENDATIONS
AMD: Clears

- Always clear Z buffer to enable HiZ

- Clearing of color render targets is *not* free on Radeon HD 2000 and 3000 series
  
  Cost is proportional to number of pixels to clear
  
  The less pixels to clear the better!

- Here the rule about minimum work applies:
  
  Only clear render targets that need to be cleared!
  
  Exception for MSAA RTs: need clearing every frame

- RT clears are *not* required for optimal multi-GPU usage
AMD: Depth Buffer Formats

- Avoid `DXGI_FORMAT_D24_UNORM_S8_UINT` for depth shadow maps
  Reading back a 24-bit format is a slow path
  Usually no need for stencil in shadow maps anyway

- Recommended depth shadow map formats:
  
  `DXGI_FORMAT_D16_UNORM`
  Fastest shadow map format
  Precision is enough in most situations
  Just need to set your projection matrix optimally

  `DXGI_FORMAT_D32_FLOAT`
  High-precision but slower than the 16-bit format
NVIDIA: Clears

- Always Clear Z buffer to enable ZCULL
- Always prefer Clears vs. fullscreen quad draw calls
- Avoid partial Clears
  - Note there are no scissored Clears in DX10, they are only possible via draw calls
- Use Clear at the beginning of a frame on any render target or depth stencil buffer
  - In SLI mode, driver uses Clears as hint that no inter-frame dependency exist. It can then avoid synchronization and transfer between GPUs
NVIDIA: Depth Buffer Formats

- Use `DXGI_FORMAT_D24_UNORM_S8_UINT`

- `DXGI_FORMAT_D32_FLOAT` should offer very similar performance, but may have lower ZCULL efficiency

- Avoid `DXGI_FORMAT_D16_UNORM`
  - will not save memory or increase performance

- CSAA will increase memory footprint
NVIDIA: Attribute Boundedness

- Interleave data when possible into a less VB streams:
  - at least 8 scalars per stream
- Use Load() from Buffer or Texture instead
- Dynamic VBs/IBs might be on system memory accessed over PCIe:
  - maybe CopyResource to USAGE_DEFAULT before using (especially if used multiple times in several passes)
- Passing too many attributes from VS to PS may also be a bottleneck
  - packing and Load() also apply in this case
NVIDIA: ZCULL Considerations

- Coarse Z culling is transparent, but it **may underperform** if:
  - If depth test changes direction while writing depth (== no Z culling!)
  - Depth buffer was written using different depth test direction than the one used for testing (testing is less efficient)
  - If stencil writes are enabled while testing (it avoids stencil clear, but may kill performance)
  - If DepthStencilView has Texture2D[MS]Array dimension (on GeForce 8 series)
  - Using MSAA (less efficient)
  - Allocating too many large depth buffers (it’s harder for the driver to manage)
Conclusion

- DX10 is a well-designed and powerful API
- With great power comes great responsibility!
  Develop applications with a “DX10” state of mind
  A naïve port from DX9 will not yield expected gains

- Use performance tools available
  AMD GPUPerfStudio
  AMD GPUShaderAnalyzer
  NVIDIA PerfHUD
  NVIDIA ShaderPerf

- Talk to us
Questions

Ignacio Llamas, NVIDIA
illamas@nvidia.com

Nicolas Thibieroz, AMD
nicolas.thibieroz@amd.com