D3D11 Deferred Contexts
Primer & Best Practices

Bryan Dudash
Developer Technology, NVIDIA
Agenda

- Discussions on bottlenecks
- What are these “deferred contexts”?
- Best Practices
- Anecdotes
- Final Thoughts
Bottlenecks
Game Engines are Complex

- Many possible bottlenecks
  - CPU
    - Game code bottleneck
    - D3D11 Runtime bottleneck
    - Driver code bottleneck
  - GPU
    - Shading, Texture, etc etc
    - Blending
  - Bandwidth
    - Texture and Buffer updates
CPU bottleneck

- This talk is about CPU bottlenecks
  - Specifically code around rendering

- Other bottlenecks well covered by previous talks
  - “DirectX11 Performance Reloaded”
  - Nick Thibiero, AMD
  - Holger Gruen, NVIDIA
Our target case

- Not feeding draw commands to driver fast enough
- Not ideal way to drive performance

* Cool diagram blatantly borrowed from "DirectX11 Performance Reloaded Talk"
What is a “Deferred Context”

- ID3D11DeviceContext that does not immediately issue commands invoked on it
  - Called a “deferred context” or “DC”
  - All commands are deferred until later
- “Finished“ into a ID3D11CommandList
  - ID3D11CommandList is executed later on immediate context (“IC”)
- Supported on all D3D11 hardware
  - Possibly through emulation in D3D11 runtime

- Check direct driver support with:

```c
struct D3D11_FEATURE_DATA_THREADING {
    BOOL DriverConcurrentCreates;
    BOOL DriverCommandLists;
} D3D11_FEATURE_DATA_THREADING
```
Simple Pseudo code example*

**IC Render Thread**

```c
ID3D11Device* pd3dDevice;
ID3D11DeviceContext* pd3dImmediateContext;
ID3D11DeviceContext* pd3dDeferredContext = NULL;
ID3D11CommandList* pd3dCommandList = NULL;

// Make ourselves a shiny new DC
pd3dDevice->CreateDeferredContext(0, &pd3dDeferredContext);

loop {  // our frame loop
    // Some IC rendering or other setup
    // Indicate to other thread to start rendering to DC
    // with Event or other threading construct
    // Possibly do some unrelated IC work
    // Wait for completion of DC thread(s)
    // Execute all deferred commands
    pd3dImmediateContext->ExecuteCommandList(pd3dCommandList, FALSE);
    // More IC rendering and back buffer swap
}
```

**Some Worker Thread**

```c
// Traverse scene graph and
// render some stuff to deferred context
// Create a command list with
// all commands since previous finish call
pd3dDeferredContext->FinishCommandList(FALSE, &pd3dCommandList);
```

* Don’t write an implementation that looks like this. This is just meant to show you the D3D11 interfaces used.
Another Simple Example – Jobs

DC Pool

Allocate DC to Group “A”
Allocate DC to Group “B”

Traverse Scene Rendering to Group “A”
Traverse Scene Rendering to Group “B”

Render Object to “A”
Render Object to “B”

Dependency

Finalize Command List “A”
Finalize Command List “B”
What does using a DC enable?

- Lower CPU bottleneck*
  - By de-serializing app render, D3D runtime and driver work

- Thread out runtime D3D calls onto as many threads as you like.
  - Can simplify a jobs solution
  - Reduced app/driver sync time

- The Good/Bad
  - +Facilitates parallelization of scene traversal
  - +Parallelizes runtime API calls
  - +Parallelizes buffer updates
  - -Redundant state overhead
    - Avoidable depending on grouping

* There are tons of caveats we’ll cover in the Best Practices section
What can’t I do with DCs?

- You knew this was coming, right?

- DCs are a “fire & forget” model
  - Deferred Contexts cannot get any feedback from the GPU
  - Query data cannot be retrieved.

- No device state inheritance or transmission
  - Always starts with default device state
  - Always leaves with default device state
  - However global state (textures, buffers, etc) persists
    - Across IC/Execute

- Only addresses CPU bottlenecks
Inherited Object State

- Global state of objects is inherited between contexts
  - Texture data, constant data, queries
- Display lists
  - Fill once, use multiple times

<table>
<thead>
<tr>
<th>Operation</th>
<th>VB data</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC: write(A)</td>
<td>A</td>
</tr>
<tr>
<td>CL execute (next 4 operations)</td>
<td>A</td>
</tr>
<tr>
<td>-- CL Map(discard) – write(B)</td>
<td>B</td>
</tr>
<tr>
<td>-- CL Map(discard) – write(C)</td>
<td>C</td>
</tr>
<tr>
<td>-- CL Draw</td>
<td>C</td>
</tr>
<tr>
<td>-- CL Map(discard) – write(D)</td>
<td>D</td>
</tr>
<tr>
<td>IC Draw</td>
<td>D</td>
</tr>
<tr>
<td>IC Map(discard) – write(E)</td>
<td>E</td>
</tr>
</tbody>
</table>
Manual Command Lists

- Application custom threaded command lists
  - Manually capture all data required to issue D3D11 calls
  - Replay on IC

- Token+Replay is what D3D11 emulation does
  - If driver doesn’t support command lists directly
  - Be careful of I$ thrashing from branchy replays
    - Branch mispredicts

- The good/bad
  - + Allows you to parallelize scene traversal
  - + Allows more efficient render state reuse
  - + Can be lock-free and guarantee no allocations/deallocations during replay
  - - Does not parallelize runtime API calls
  - - Does not parallelize buffer updates on app thread
    - Driver still able to parallelize these
  - - Watch out for thread sync issues
Some Numbers

All numbers run on:
- In house DC test application
- Notebook Core i7 2670QM @ 2.2GHz
- 16GB RAM,
- GeForce GTX560M
Batched DC versus IC

* Run with custom DC test application
Scene vs Per-Draw vs Batched DC performance

* Run with custom DC test application

Too many thread events causing efficiency issues
Best Practices
Test Test Test

- Always test using the latest drivers
  - Remember to test on equivalent systems
  - Or the same system for best results
- Two complete render paths
  - Initial render path (non-DC)
  - DC command lists threaded path
  - At least during internal dev to make sure you are gaining perf
- Try to test on different:
  - CPUs – clock speed and cores affect CPU perf and bottlenecks
  - GPUs
    - Multiple generations
    - Multiple IHVs – different drivers have different implementations
  - Motherboards – PCIE bandwidth may affect CPU waiting
Be a Good Buffer Management Citizen™

- John McDonald’s “Efficient Buffer Management”
  - GDC2012 talk

- NEVER readback from the GPU
  - I.e. Never use staging resource on a DC
  - Will result in the map being forced onto IC
    - when command list is executed
  - And thus serialized
  - And anything dependent on that will also be serialized
NEVER set Restore Context State

- 2\textsuperscript{nd} parameter to \texttt{ExecuteCommandList}
- If set to TRUE
  - Will save and restore \texttt{ALL} d3d state
  - Set \texttt{*tons*} of redundant state
  - Added CPU overhead
- If set to FALSE
  - Application is responsible to set what state it needs
  - Likely you are already setting proper state
Load Balance List Size

- Don’t make a new DC/Commandlist for every draw call
  - Really, just don’t
- Don’t make your command lists too short
  - Should have at least a few hundred API calls
    - At least dozen draws or so
    - A “standard” mix of buffer updates, state setting and draws.
- Don’t make your command lists too long
  - Execute of long lists may interfere with other IC calls
  - Chop into multiple as some tweak-able limit
- Dependent on engine implementation
  - State per call, etc
  - See “TEST TEST TEST” best practice
Operations to Avoid

- Doing these inside a DC will affect performance adversely
- Queries
  - Subsequent `getData()` on IC will (potentially) stall until DC exec reaches `endQuery`
- Readbacks/blit to staging resources
  - Subsequent `map()` on IC will potentially stall until DC exec reaches the blit
- Any really large one time updates
  - Do these on IC
Don’t hog the CPU

- I know you want to get to 100% utilization but...
  - If the driver has no headroom to process commands then your worker threads will just be waiting...

- Driver cannot fully transform to hardware commands on DC
  - Some work remains to be done on IC during command list execute
  - If all cores are dominated by application, driver is starved.
  - Try 2*(N-1) as well as (2*N)-1 application threads
    - i.e. 6-7 on a quad core. For *all* game threads.
    - Driver may or may not need a full physical core
    - Test test test
Don’t muck with CPU affinity

- Will almost never offer a speedup
- Will interfere with driver’s efficiency
  - Can quickly become bottleneck
Don’t pre-clear state

- DCs provide a default state context for you!
- Clearing state is just extra busy work
  - But may happen as a result of your engine’s state management code
- Examples
  - Setting shaders to NULL
  - Setting SRVs to NULL
  - Etc...
Manage Redundant State

- A general best practice
- Spend time on threads to determine which state can be reused
  - May not be true for single threaded IC
Maintain a DC pool

- Initialize DCs pool with threads
- Reuse these
  - DC state resets after finalize
- DCs hold memory while commands lists are “in flight”
  - Or longer if you don’t release the command list!
  - ~10-30MB/list/frame assuming balanced lists
    - Constant buffers, state, etc
  - Plus dynamic buffer updates sizes
  - 32bit applications may run into address space issues for large command lists
UpdateSubResource bug

• On drivers that don’t support command lists

If your application calls `UpdateSubresource` on a deferred context with a destination box—to which `pDstBox` points—that has a non-`(0,0,0)` offset, where the driver does not support command lists, `UpdateSubresource` inappropriately applies that destination-box offset to the `pSrcData` parameter.

• There is workaround code listed in the MC D3D11 documentation for `UpdateSubResource`
Anecdotes
Civilization V

- Watch Dan Baker’s GDC2010 presentation.
  - “Firaxis’ Civilization V: A Case Study in Scalable Performance”
- Large multi-threaded engine
  - Sometimes >10k draws per frame (w/ lots of state)
- “n wide” render buffers
  - Threaded out to # of cores
  - Cognizant of command list sizes
    - Load balance to homogenize # of calls
- DCs versus serialized execution of render commands initially gained ~50% performance
  - Later non-DC path optimizations closed that gap a bit
- Saw major benefits from parallel buffer updates
Other Anecdotes

- **Assassin's Creed 3**
  - Conservatively ~24% gain from using DCs in CPU bottleneck situations
  - >> in some situations
  - i.e. 37 FPS -> 46 FPS
    - 2.93GHZ Nehalem, GTX680, 720p

- **Other engines**
  - DC command lists quicker to implement than manual threading with IC
    - Simpler than rolling your own token+replay
  - Be careful with too many command lists
    - Extra state require to set up draws
    - Lint on your state calls to avoid redundant sets
      - Important in non-DC case as well
  - Watch out for over utilizing CPU in game code
    - Driver needs some time too

* Covers common cases on various engines, so just call ‘em general anecdotes*
Final Thoughts

- Threading your engine == good
  - Jobs/Work system == better
- Driver DC command lists
  - Parallelize API calls and buffer updates
  - May add overhead from extra state sets
    - Amortize by grouping and state change filters
- **Always test performance continuously**
  - To make sure you have the right solution for your game
  - Test on both AMD and NVIDIA
Final Thoughts(2)

- Work with your IHV
  - Only you can prevent CPU bottlenecks™
  - Constantly tuning driver performance for game engine workloads
    - Improved directly as a result from working with Civ5 and AC3
- DC use may(should) shift bottleneck
  - GPU may become bottleneck
  - Driver may become bottleneck
Questions?

Bdudash at nvidia.com