GDC 12

Don’t Throw it all Away: Efficient Buffer Management

John McDonald
Developer Technology, NVIDIA Corporation

GAME DEVELOPERS CONFERENCE 2012
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9

What are we talking about?

- General Performance/Functional Guidance
- CPU-GPU Sync Points
- Buffer Usage Patterns
- Contention-Free Buffers
- Constant Buffers
- Performance Investigation

“Buffers” is really generic…

- Vertex Buffers
- Index Buffers
- Constant Buffers

General Guidance

General Guidance

- D3D11 >> D3D9 **(generally)**
  - It's **much** harder to hit the ultra-slow path (aka CPU-GPU Sync Points)
- Reduce your API calls where possible
  - Batch up buffer updates
- Alignment matters! (16-byte, please)
  - Aligned copies can be ~30x faster

More General Guidance

- D3D11Device will grab a mutex for you, but each DeviceContext can only be called from one thread at a time
  - This is the source of **many** crashes blamed on the driver
- UpdateSubresource requires more CPU time
  - When possible, prefer Map/Unmap
- D3D11 Debug Runtime is awesome!
  - Please use it, ensure you are running clean

CPU-GPU Sync Points

CPU-GPU Sync Points

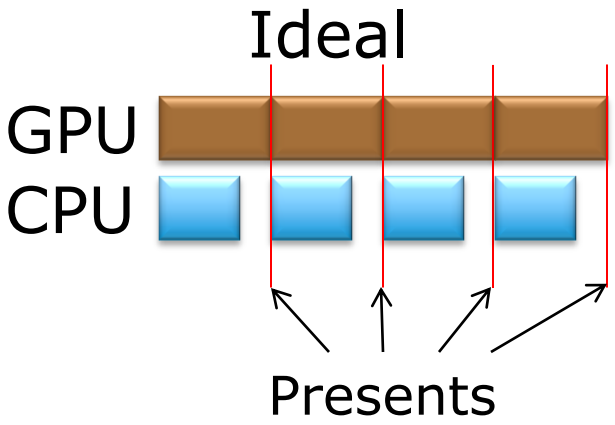
- CPU-GPU Sync Points are caused when the CPU needs the GPU to complete work before an API call can return
- These make us sad

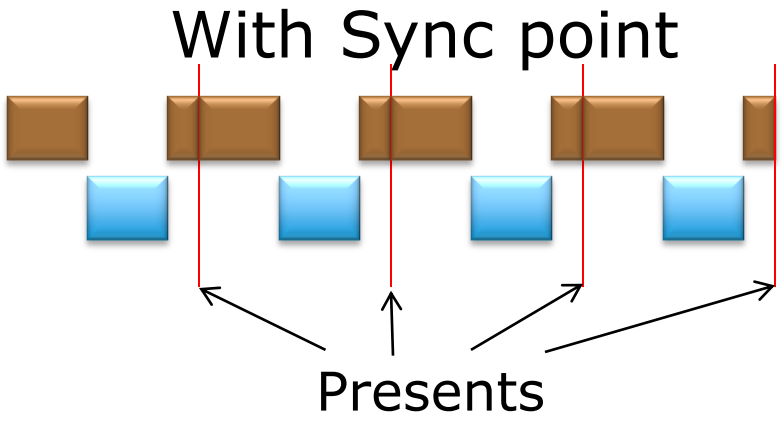
CPU-GPU sync point examples

- Explicit
  - Spin-lock waiting for query results
  - Readback of Framebuffer you just rendered to
- Implicit (potential sync points)
  - GPU Memory Allocation after Deallocation
  - Buffer Rename operation (MAP_DISCARD) after deallocation
  - Immediate update of a buffer still in use

Why are they bad?

- Ideal frame time should be max(CPU time, GPU time)
- CPU-GPU Sync point turns this into CPU Time + GPU Time.





Really? That bad?

- One bad sync point can **halve** your frame rate
- Even worse: the more sync points you have, the harder they are to find.
  - Performance will just seem generally slow
- The badness depends, in part, on where in the frame the sync-point occurs
  - Generally, the later the sync point, the worse it is
  - Early sync-points are also bad if your workload is very lopsided towards either the CPU or the GPU

Check your middleware

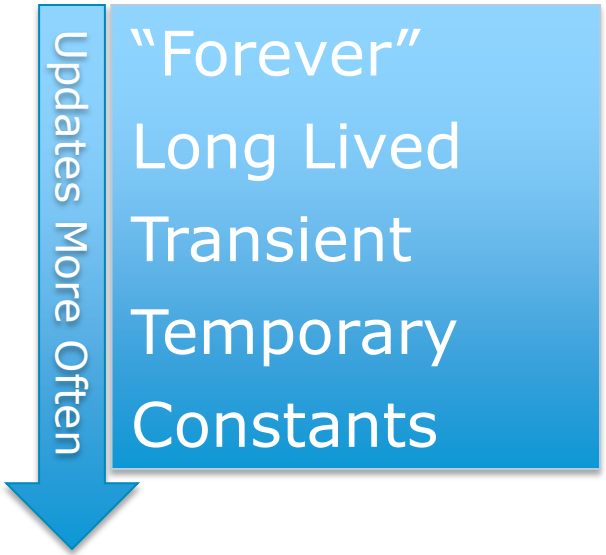
- Middleware is generally written in a vacuum
  - What works best in the small might not scale well
- Especially check for CPU-GPU sync points

A quick D3D9 interlude

- CPU-GPU sync points are *trivial* to introduce in D3D9
- Locking any buffer in D3D9 with flags=0 is a virtually guaranteed CPU-GPU Sync point if that buffer is still in use. ☹☹☹

Buffer Usage Patterns

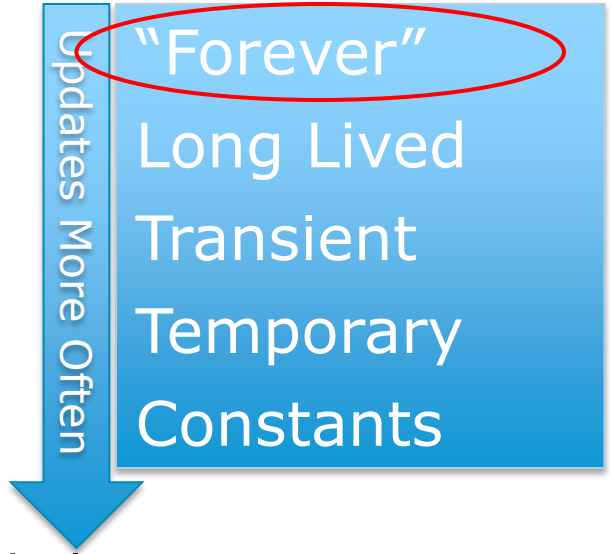
Buffer Usage Patterns



- Level BSPs
- Character Geometry
- UI, Text (New!)
- Particle Systems (Streaming)
- Shader Parameters

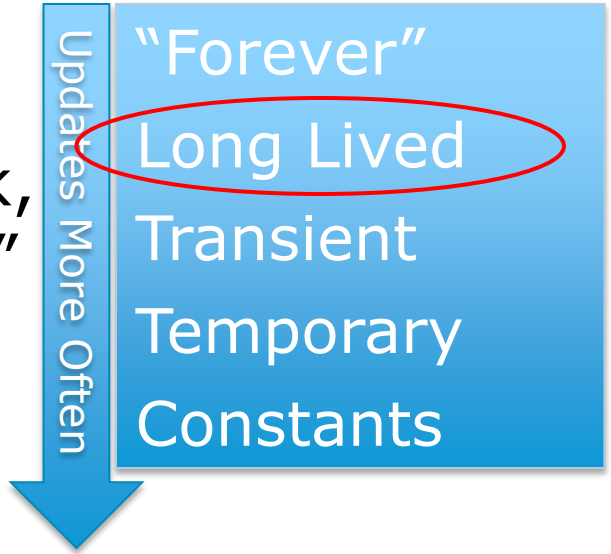
"Forever" Buffers

- Useful for geometry that is loaded once
  - Ex: Level BSPs, loaded behind a load screen
- Don't use this for streaming data
  - Hitching during allocation is possible/likely
- IMMUTABLE flag at creation time
- Cannot update these!



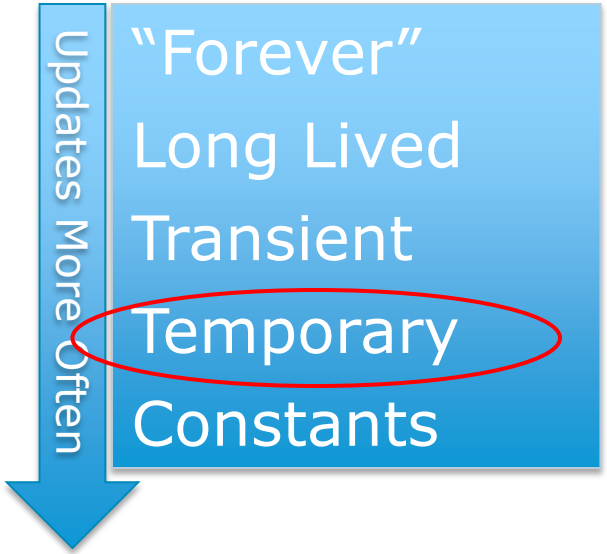
Long Lived Buffers

- Data that is streamed in from disk, but is expected to last for "awhile"
  - Ex: Character geometry
- Reuse these; stream into them
- DEFAULT flag at creation time
- UpdateSubresource to update



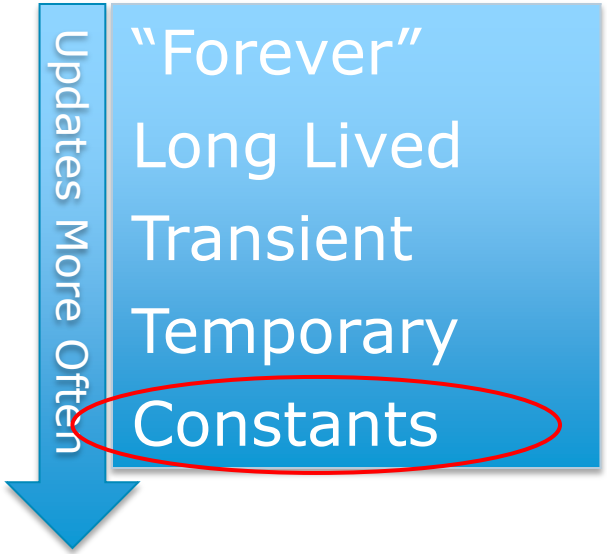
Temporary buffers

- Fire-and-forget data
  - E.g. Particle systems
- Almost certainly lives in system RAM
- DYNAMIC flag at create time
- Prefer Map/Unmap to update these
  - UpdateSubresource involves an extra copy



Constant Buffers

- These are different than other buffers in D3D11.
- The GPU can deal with many of them in flight at once
- Create with DYNAMIC
- Map/DISCARD to Update
- More on these in a bit

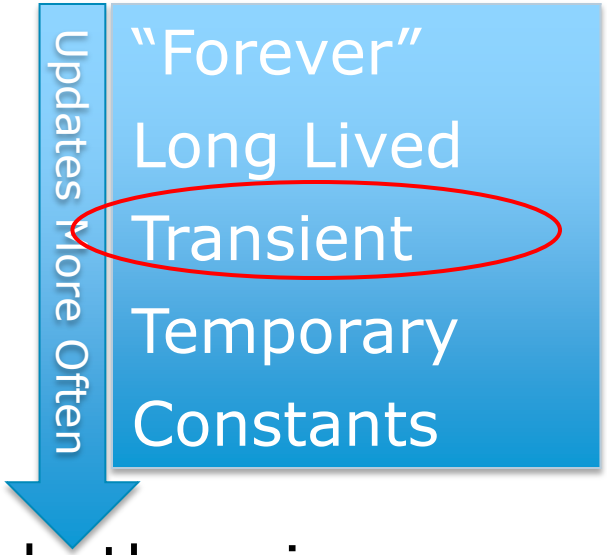


We skipped one…

- Transient Buffers
- New informal class of Buffer
- Used for (e.g.) UI/Text
  - Things that are dynamic, but few vertices each—and may need to be updated on odd schedules
- DYNAMIC flag at creation time
- Transient Buffers are part of a new class of buffer…

Contention-Free Buffers

Transient Buffer Overview



- Treat Buffer as a Memory Heap, with a twist
  - On CPU, Freed memory available now
  - On GPU, Freed memory is available when GPU is finished with it
- Assume memory is in use until told otherwise
- Determine when GPU must be finished with Freed memory, then return to the "really free" list

CTransientBuffer

- On Alloc, walk a Free list looking for best fit
  - Data is updated using Map/NO_OVERWRITE
  - Return opaque, immutable handle
- On Free, record that chunk was freed—into RetiredFrames.back()
- Just after present, an "OnPresent" function is called

```
class CTransientBuffer
{
  ID3D11Buffer* mBuffer;
  UINT mLengthBytes;
  ID3D11Device* mOwner;
  vector<CSubAlloc> mFreeList;
  list<RetiredFrame> mRetiredFrames;

public:
  CSubAlloc* Alloc(UINT, void*,
                   ID3D11DeviceContext*);
  void Free(CSubAlloc*);
  void OnPresent(ID3D11DeviceContext*);
```

CTransientBuffer Guts

```
class CTransientBuffer
{
  ID3D11Buffer* mBuffer;
  UINT mLengthBytes;
  ID3D11Device* mOwner;
  vector<CSubAlloc> mFreeList;
  list<RetiredFrame> mRetiredFrames;

public:
  CSubAlloc* Alloc(UINT, void*,
                   ID3D11DeviceContext*);
  void Free(CSubAlloc*);
  void OnPresent(ID3D11DeviceContext*);
  ...
```

```
struct RetiredFrame
{
  list<CSubAlloc*> mPendingFrees;
  ID3D11Query* mFrameCompleteQuery;
};
```

```
class CSubAlloc
{
  UINT mOffset;
  UINT mLength;

  ...
```

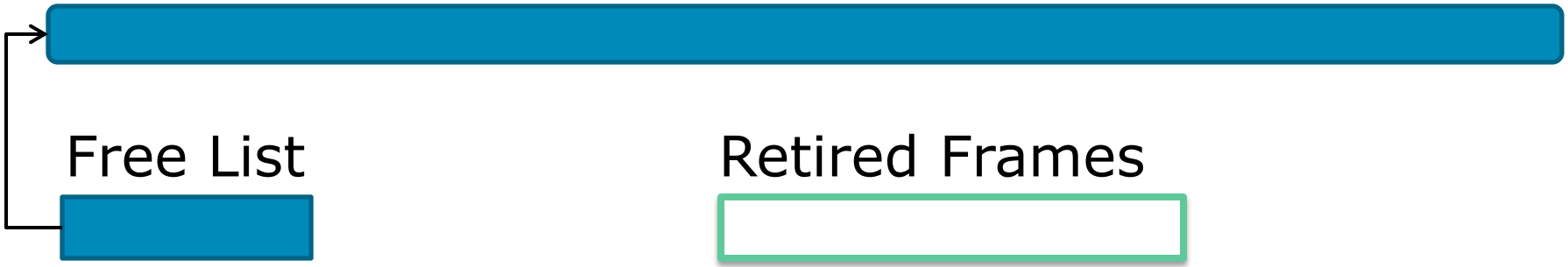
CTransientBuffer::OnPresent

```
void CTransientBuffer::OnPresent(ID3D11DeviceContext* _dc)
{
  // First, deal with deletes from this frame
  RetiredFrame& retFrame = mRetiredFrames.back();
  if (!retFrame.mPendingFrees.empty()) {
    retFrame.mFrameCompleteQuery = CreateAndIssueEventQuery(_dc);

    // Append a new (empty) RetiredFrame to mRetiredFrames
    mRetiredFrames.push_back(RetiredFrame());
  }
  // Second, return pending frees to mFreeList
```

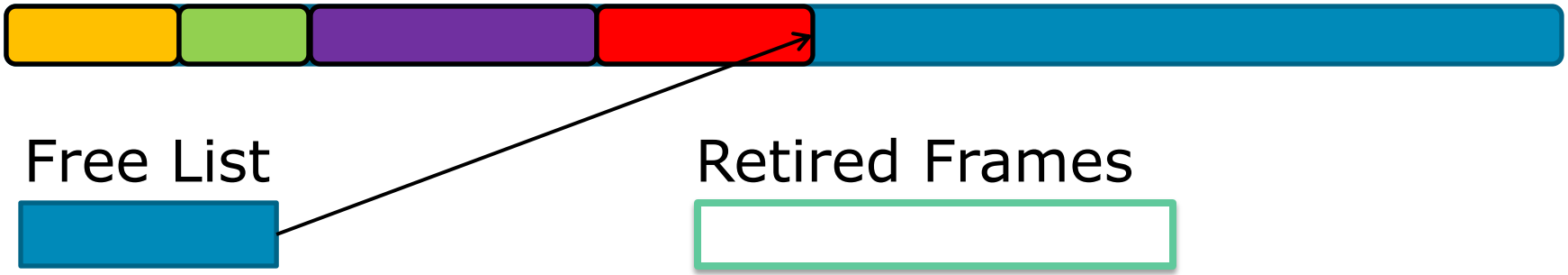
CTransientBuffer::OnPresent

```
    // Second, return pending frees to mFreeList
    FOREACH(frameIt, mRetiredFrames) {
    auto query = frameIt->mFrameCompleteQuery;
    if (!(query && IsQueryComplete(query)))
      break;

    FOREACH(suballocIt, frameIt->mPendingFrees) {
      ReallyFree(*subAllocIt);
    }
  }
}
```

CTransientBuffer Visualized

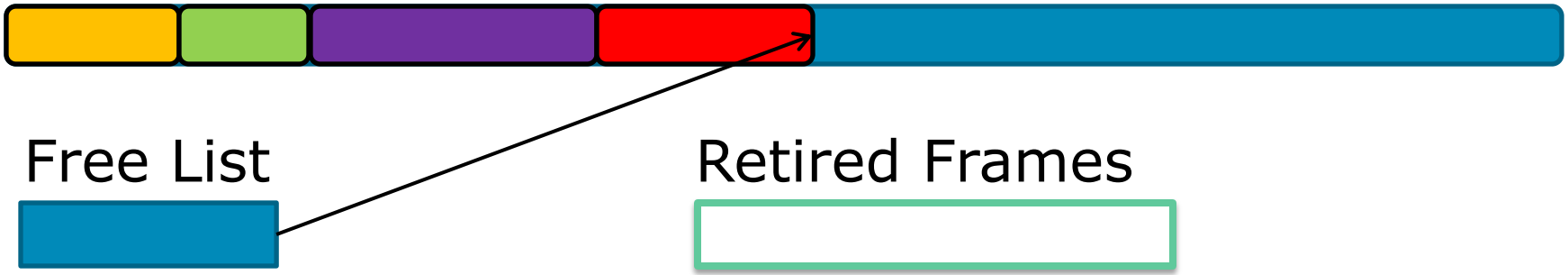


CTransientBuffer Visualized



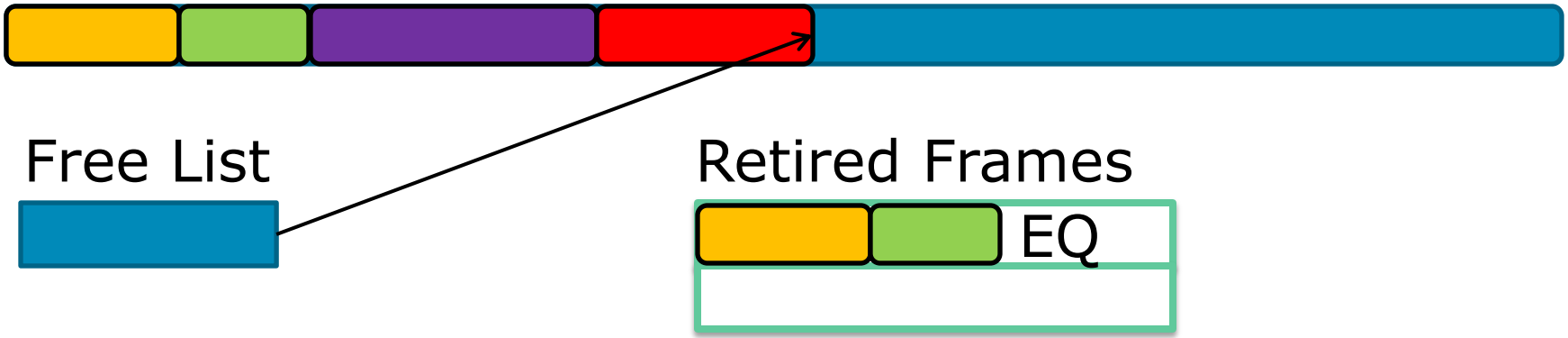
Allocating four Buffers

CTransientBuffer Visualized



Nothing

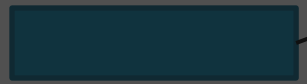
CTransientBuffer Visualized



Deallocating
Yellow and Green

CTransientBuffer Visualized

Free List

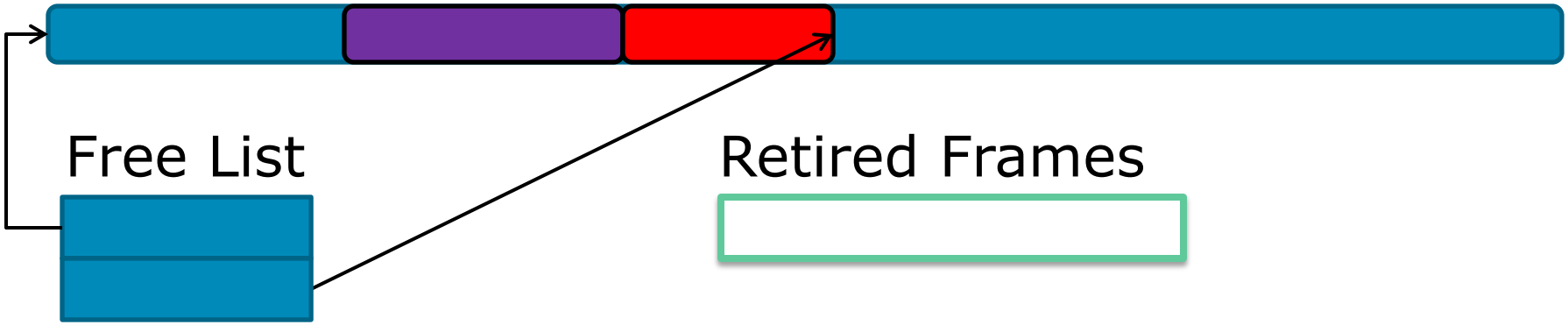


Retired Frames

A Few Frames Later

Deallocating Yellow and Green

CTransientBuffer Visualized



EQ Returns for Retired Frame

CTransientBuffer: Handling OOM

- Ways to handle Out of Memory on Alloc:
  - Spin-lock waiting for RetiredFrame Queries to return
  - Allocate a new, larger buffer
    - Release current buffer
    - Requires a system memory copy to initially fill new buffer
- These will (probably) stall
  - But in your code
    - can be easily logged -and/or-
    - Recorded to adjust and avoid for subsequent runs

Transient Buffer Pattern

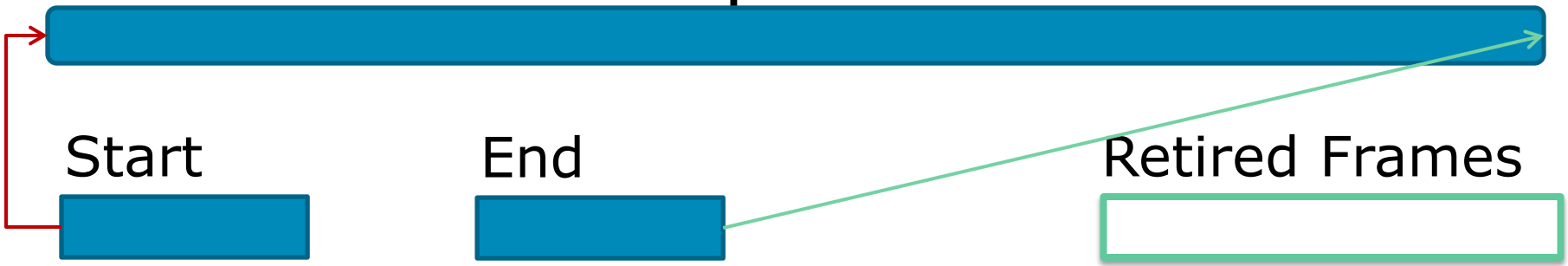
- Works in D3D9 as well
- Can be extended and simplified to contention-free Temporary Buffers, too!
  - Let’s take a quick look at that.

Discard-Free Temporary Buffers

- Allocate out of Buffer as a circular buffer
- No opaque handle needed
- Remember ending address of the last allocation
- Per frame: Assuming any allocations, issue query
- Later: When query returns, move the end pointer to indicate additional available space

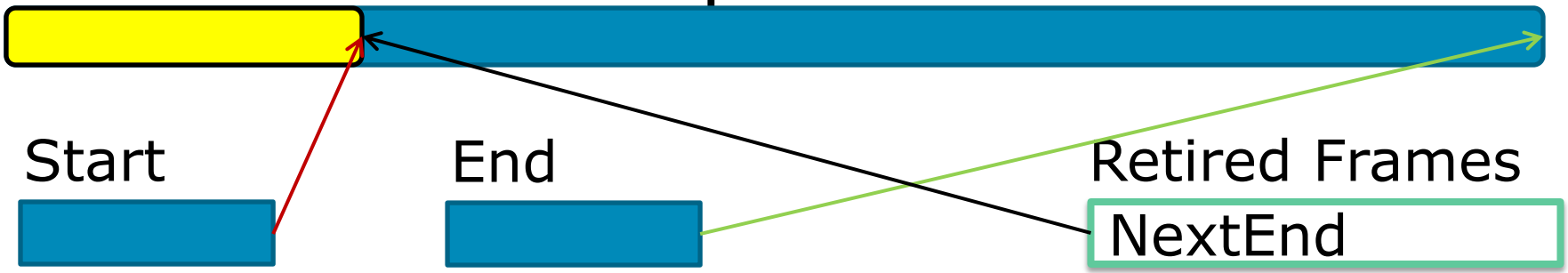
- Credit: Blizzard's StarCraft 2 Team (thanks!)

Discard-Free Temp Buffer Visualized



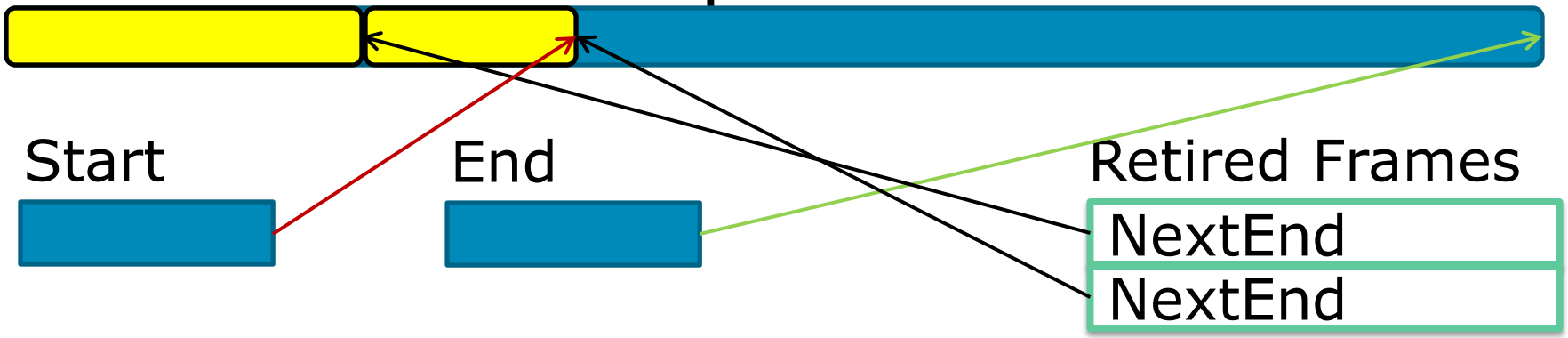
Start State

Discard-Free Temp Buffer Visualized



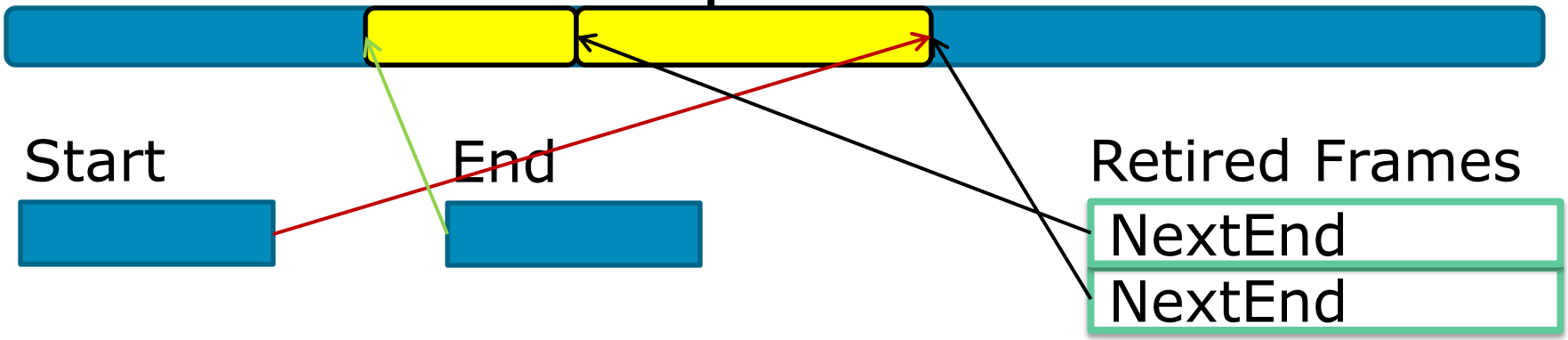
Allocate some stuff

Discard-Free Temp Buffer Visualized



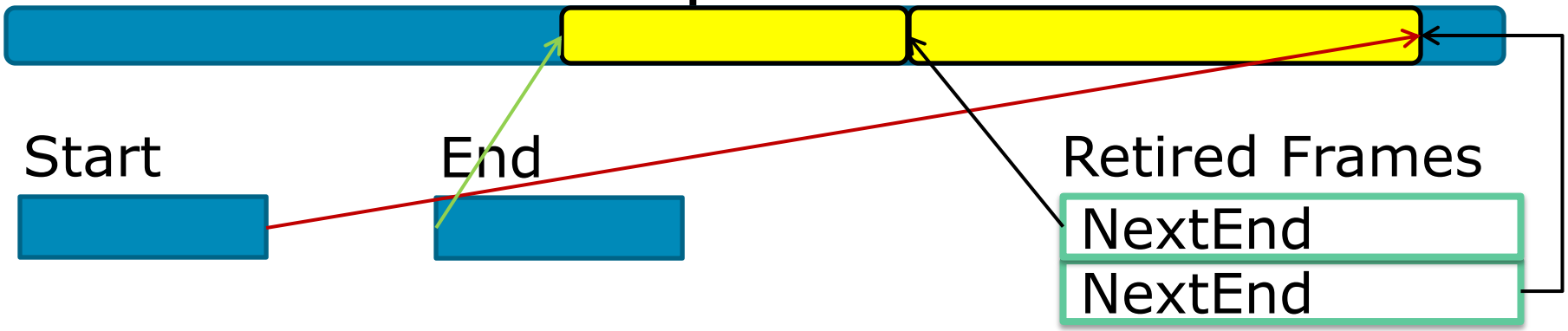
Go on…

Discard-Free Temp Buffer Visualized



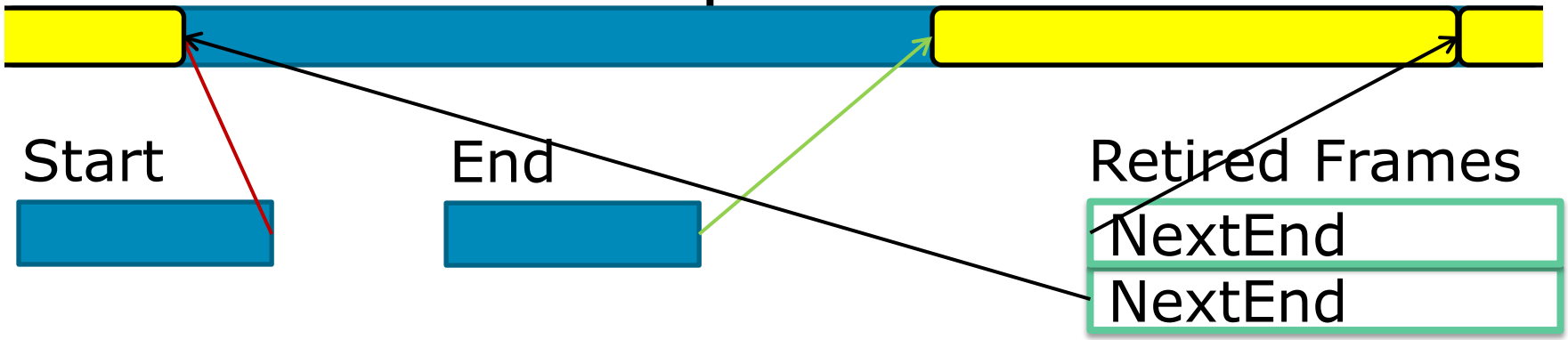
Queries start to return…

Discard-Free Temp Buffer Visualized



etc…

Discard-Free Temp Buffer Visualized



etc…

Constant Buffers

Constant Buffer Organization

- Group by frequency of update
- The cheapest buffers are the ones you never update
- You can bind multiple buffers in one call (Reduce those API calls!)

Proposed Buffer Grouping

- Assuming you are not vertex shading limited
  - Don’t solve the travelling salesman in your VS
  - Seriously: this isn’t common

Multiple Constant Buffers

- One for per-frame constants (GI values, lights)
- One for per-camera constants (ViewProj matrix, camera position in world, RT dimensions)

Old HLSL

```
oPos = in.Position
     * cWorldViewPos;
```

New HLSL

```
oPos = in.Position
     * cWorld
     * cViewPos;
     ^
```

One extra 3x3 matrix multiply in the VS. No biggie.

Multiple Constant Buffers cont'd

- One for per-object constants (World matrix, dynamic material properties, etc)
- One for per-material constants (if these are shared—if not then drop them in with per-object constants)
- Splitting constants this way eliminates constant updates for static objects.

Constant Buffer Tricks

- Use shared structs to update when possible
  - Struct can be included from both hlsl and C++
  - Makes buffer updates trivial!
- Assign them to slots by convention:
  - b0: Per-Frame, b1: Per-Camera, etc
  - Slot assignment can live in shared header, too.

Performance Investigations

Performance Investigation

- Scene from a Typical D3D11 Application (unreleased)
  - 115 Dynamic Vertex Buffer Updates (particles) per frame
  - Total Time: 4.36ms / frame

Per-	Call	Frame
Map/Unmap	0.036 ms	3.79 ms
Memcpy	~0.004 ms	0.4 ms

Let's buffer the updates

- All Dynamic Updates during one update
  - 1 Map per frame (using MAP_DISCARD)
  - Still 115 memcpys (I'm lazy)
  - Total Time: 0.267ms / frame (savings: 4.1ms!)

Per-	Call	Frame
Map/Unmap	0.036 ms	0.036 ms
Memcpy	~0.002 ms	0.231 ms

Buffered update, no discards

- One update into a triple buffer
  - 1 Map per frame (using MAP_NOOVERWRITE)
  - Still 115 memcpys (I'm *still* lazy)
  - Total Time: 0.217ms / frame (savings: 4.15ms)
  - Bonus: No hitching ever
  - Downside: 3x the memory

Per-	Call	Frame
Map/Unmap	0.031 ms	0.031 ms
Memcpy	~0.002 ms	0.231 ms

Performance Results

- Reducing API usage was a huge CPU-side savings (4.09 ms). GPU Perf Unaffected
- Discard-Free updates were marginally faster still—but would never hitch.

	Total Frame Time
Original	4.360 ms
Buffered Updates	0.267 ms
Discard-Free	0.217 ms

GPUView

- Covered by Jon Story earlier today
  - Hopefully you caught it!
- Great for finding CPU-GPU sync points

Questions?

- jmcdonald at nvidia dot com

Nifty Buffer Summary Table

Type	Usage (e.g)	Create Flag	Update Method
“Forever”	Level BSPs	IMMUTABLE	Cannot Update
Long-Lived	Characters	DEFAULT	UpdateSubResource
Transient	UI/Text	DYNAMIC	CTransientBuffer
Temporary	Particles	DYNAMIC	Map/NO_OVERWRITE
Constant	Material Props	DYNAMIC	Map/DISCARD