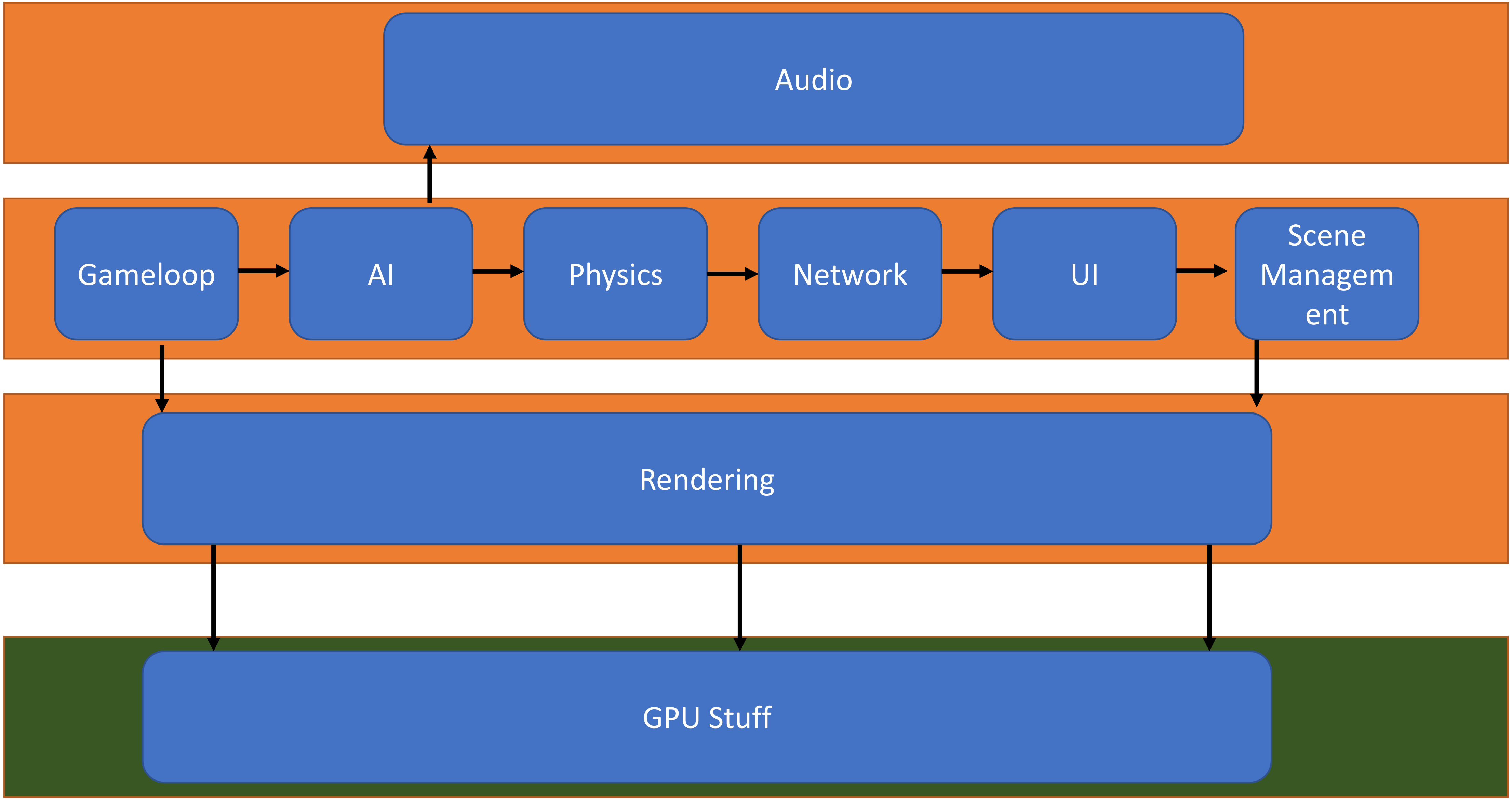
Next generation Engine design

Aka, why did we build these next gen APIs in the first place?

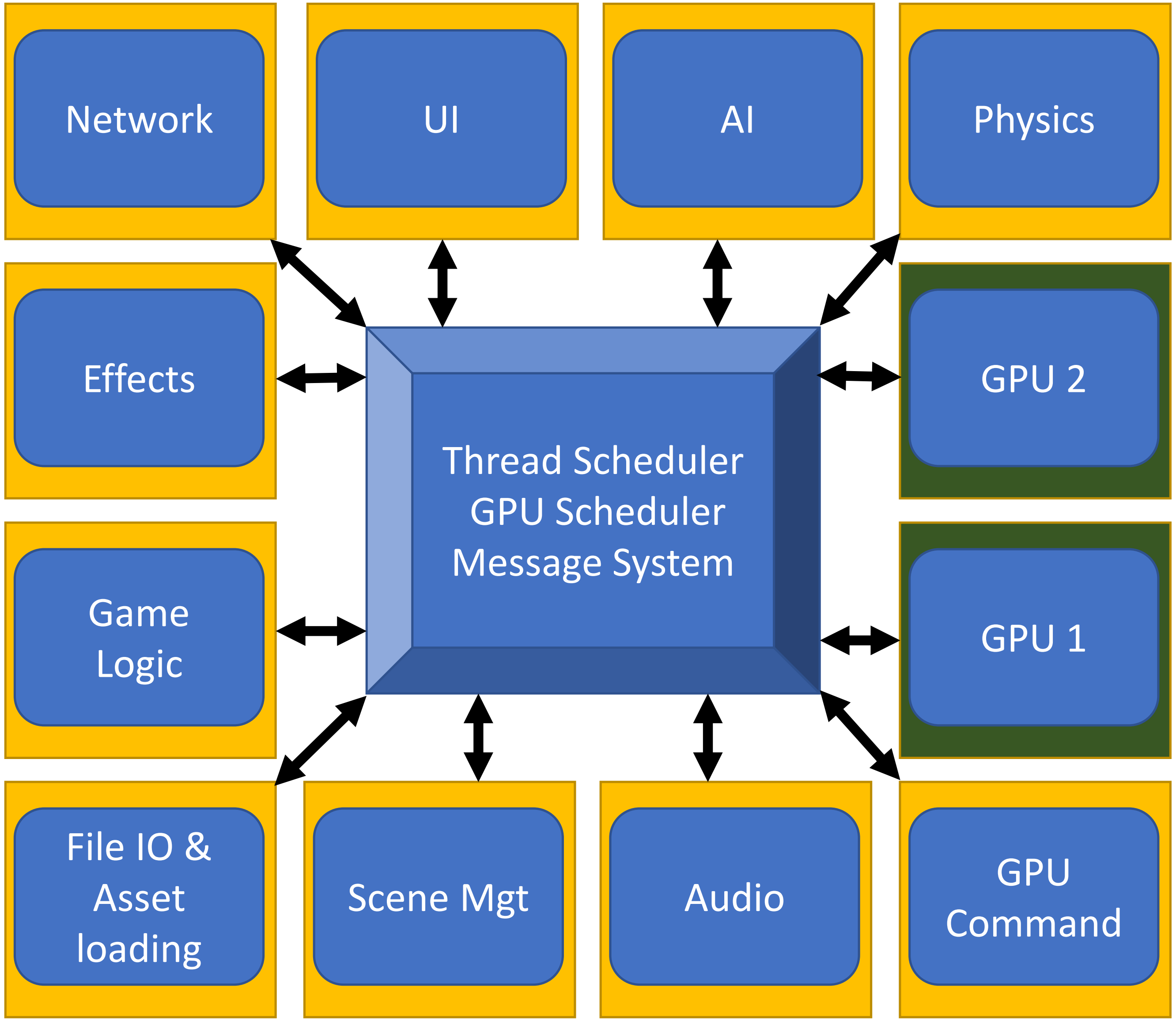
Dan Baker, Graphics Architect, Oxide Games

What’s so hard about next gen APIS?

- The Myth: “Next generation APIs are hard, older APIs are easier”
- The reality: “Multi-core, asynchronous programming is hard. Old APIs make it harder or impossible, new APIs make it doable.”
- To do: Learn multicore, asynchronous programming techniques
- Don’t if it all possible: retro-fit old scalar code with next gen API
- Need to think about how the entire engine should work

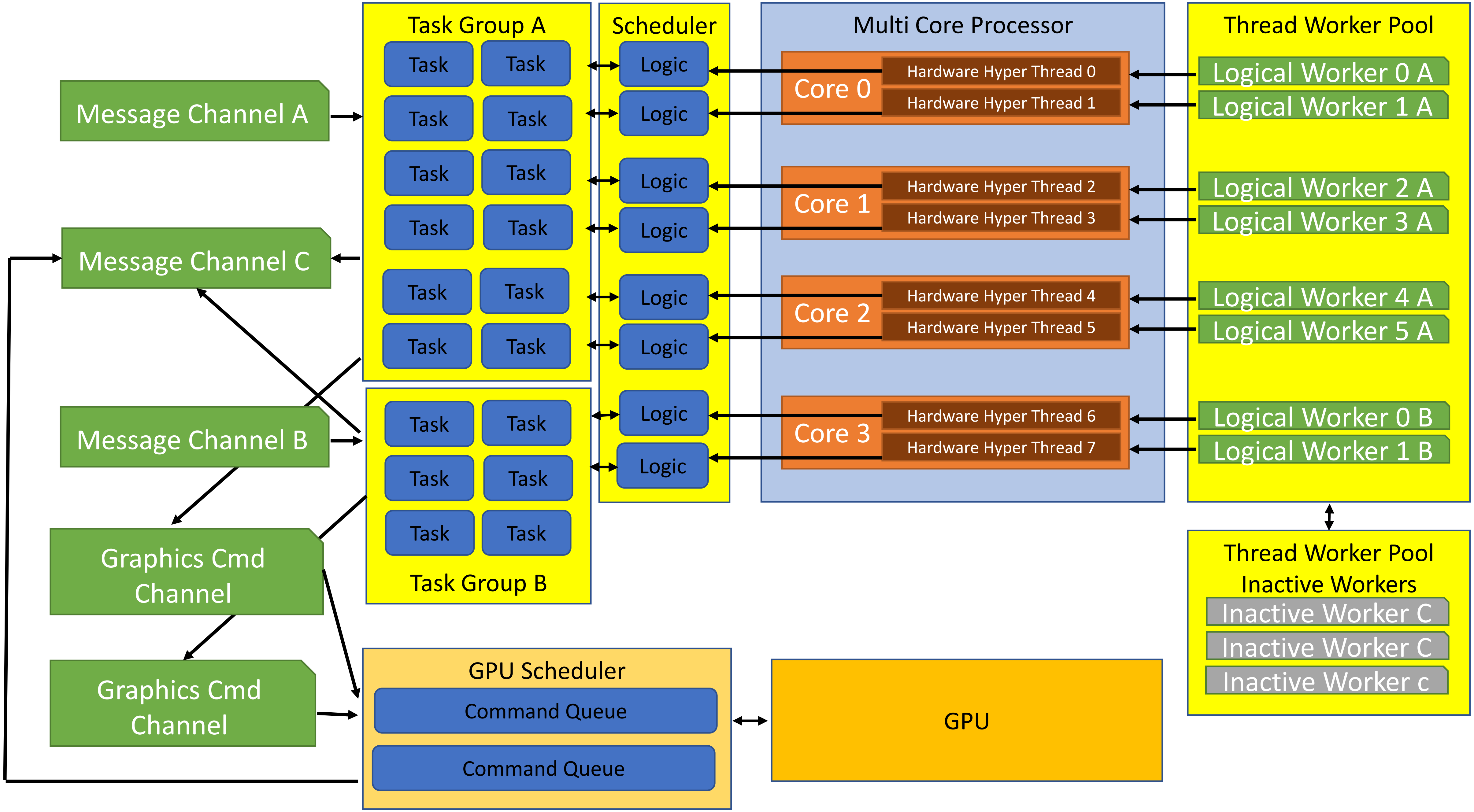

Audio
Gameloop
AI
Physics
Network
UI
Scene Managem ent
Rendering
GPU Stuff

Pre-REQUISITE Motivational SLIDE



Legend
Yellow boxes – CPU clusters
Green Boxes – GPU clusters
Arrows – Message or Command channels

All major systems in Nitrous communicate asynchronously when possible, each system could run on a different physical computer, with relatively high tolerance for latency



Task Group A
Task
Scheduler
Logic
Multi Core Processor
Core 0
Hardware Hyper Thread 0
Hardware Hyper Thread 1
Core 1
Hardware Hyper Thread 2
Hardware Hyper Thread 3
Core 2
Hardware Hyper Thread 4
Hardware Hyper Thread 5
Core 3
Hardware Hyper Thread 6
Hardware Hyper Thread 7
Thread Worker Pool
Logical Worker 0 A
Logical Worker 1 A
Logical Worker 2 A
Logical Worker 3 A
Logical Worker 4 A
Logical Worker 5 A
Logical Worker 0 B
Logical Worker 1 B
Message Channel A
Message Channel C
Message Channel B
Graphics Cmd Channel
Graphics Cmd Channel
Task Group B
Thread Worker Pool Inactive Workers
Inactive Worker C
Inactive Worker C
Inactive Worker c
GPU Scheduler
Command Queue
Command Queue
GPU

Pre-REQUISITE Motivational SLIDE

Problems with previous gen APIs

- Lots of little things add up
- 2 major problems require rearchitecture
  - Functional threading model throws a wrench into task based systems
  - Implicit Hazard tracking and synchronization
    - API tries to hide the async nature of GPU

Can't RETRO fit old APIS

- Lots of little things, memory model, binding model, etc
- Analysis of features like instancing indicate that it is unreliable and tends to speed up only the fastest frames, correlation between batches and driver perf is casual

Multi-core CPU Basics

Be Wary, There Is A Lot Of Very Bad Advice In The Wild

- Spawning threads to handle tasks
- Relying OS preemptive scheduler, heavy weight OS synchronization primitives
- Functional threading in general

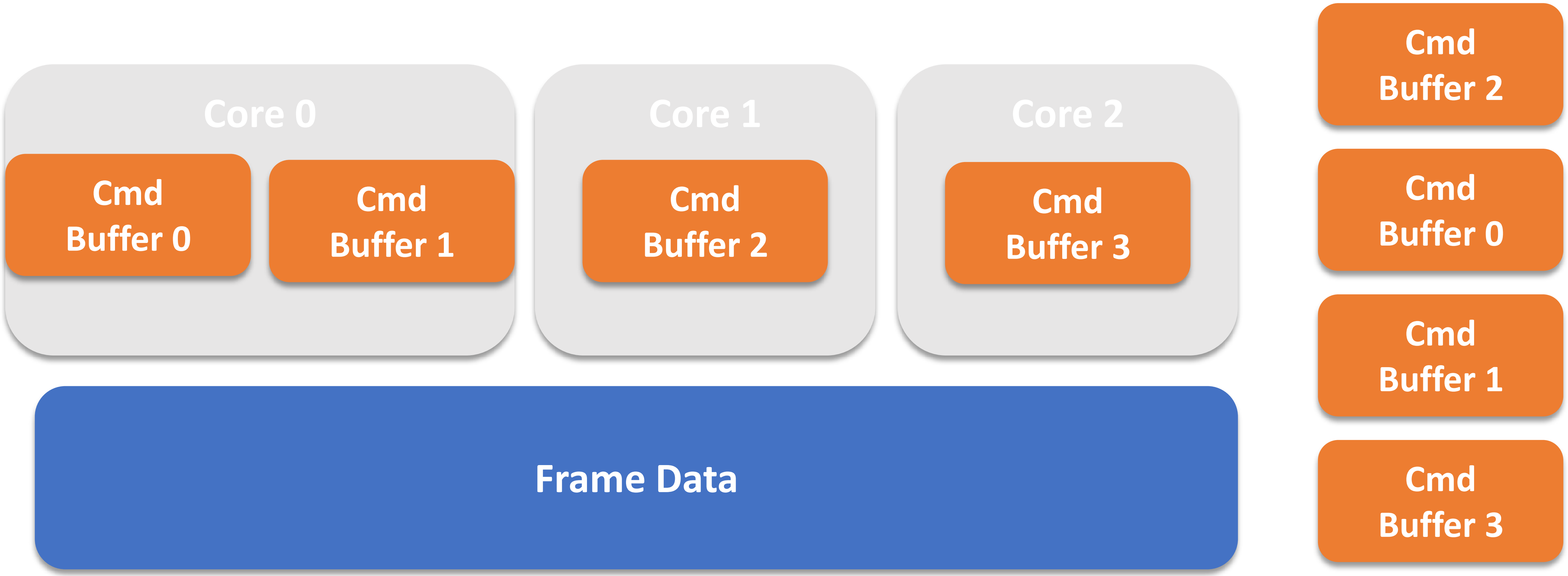
Your Survival Guide

- OK: Multi-thread read of same location
- OK: Multi-thread write to different locations
- OK: Multi-thread write to same location in 'stamp' mode
- CAUTION: Atomic instructions
- STOP: Multi-thread read/write to same location
- STOP: Multi-thread write to same CACHE line

Task based system

- Idea is that work load is a constructed graph of much smaller nuggets
- Many advantages
  - Scales well, 32+ cores
  - Easy to balance workload
  - More power efficient – more slower cores just as good
    - Already seeing CPUs dynamically slowing clock speed
  - If enough similar work items queued, can execute same code on cores
    - Cache hit rate much higher
  - End up generating a larger number of command buffers to prevent thread serialization

How Nitrous generates commands

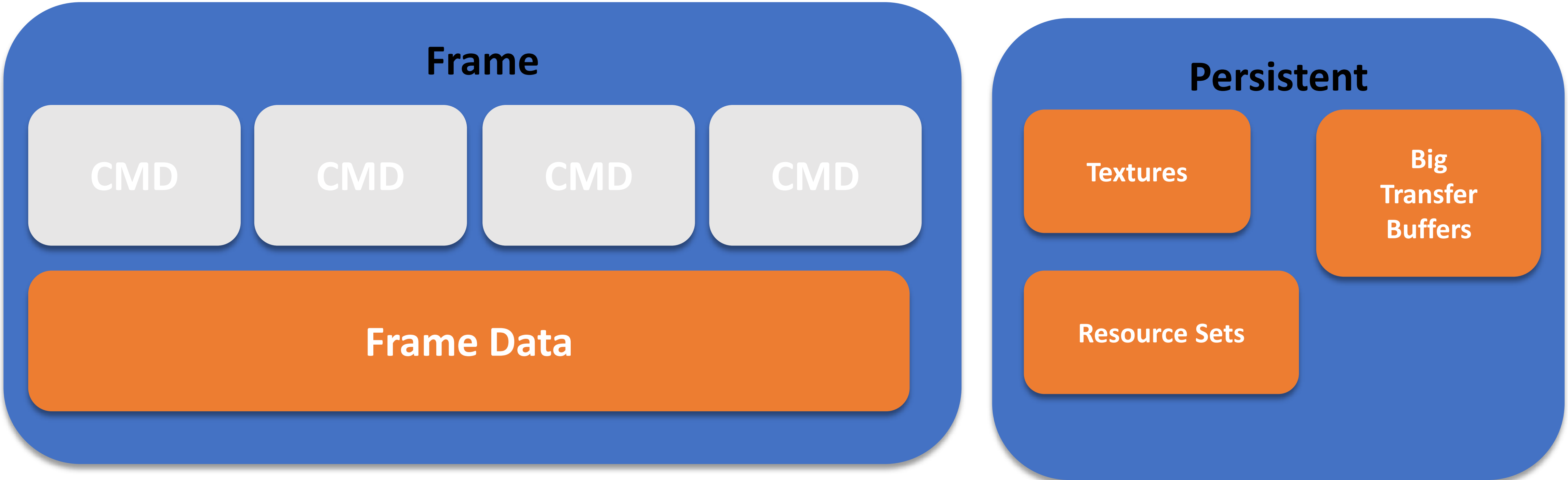


Nitrous command formats

- In reality, diagram is over simplified
- Nitrous has it’s own internal command format
  - Small, efficient commands
  - Stateless, each command contains references to all needed state
  - Inheritance unneeded
  - Separates internal graphics system from any particular API
- Being Stateless, can be generated completely out of order
- Entire Frame is queued up in internal command format
- Frame is translated to GPU commands via Vulkan
- Gets more optimal use out of instruction cache and data cache

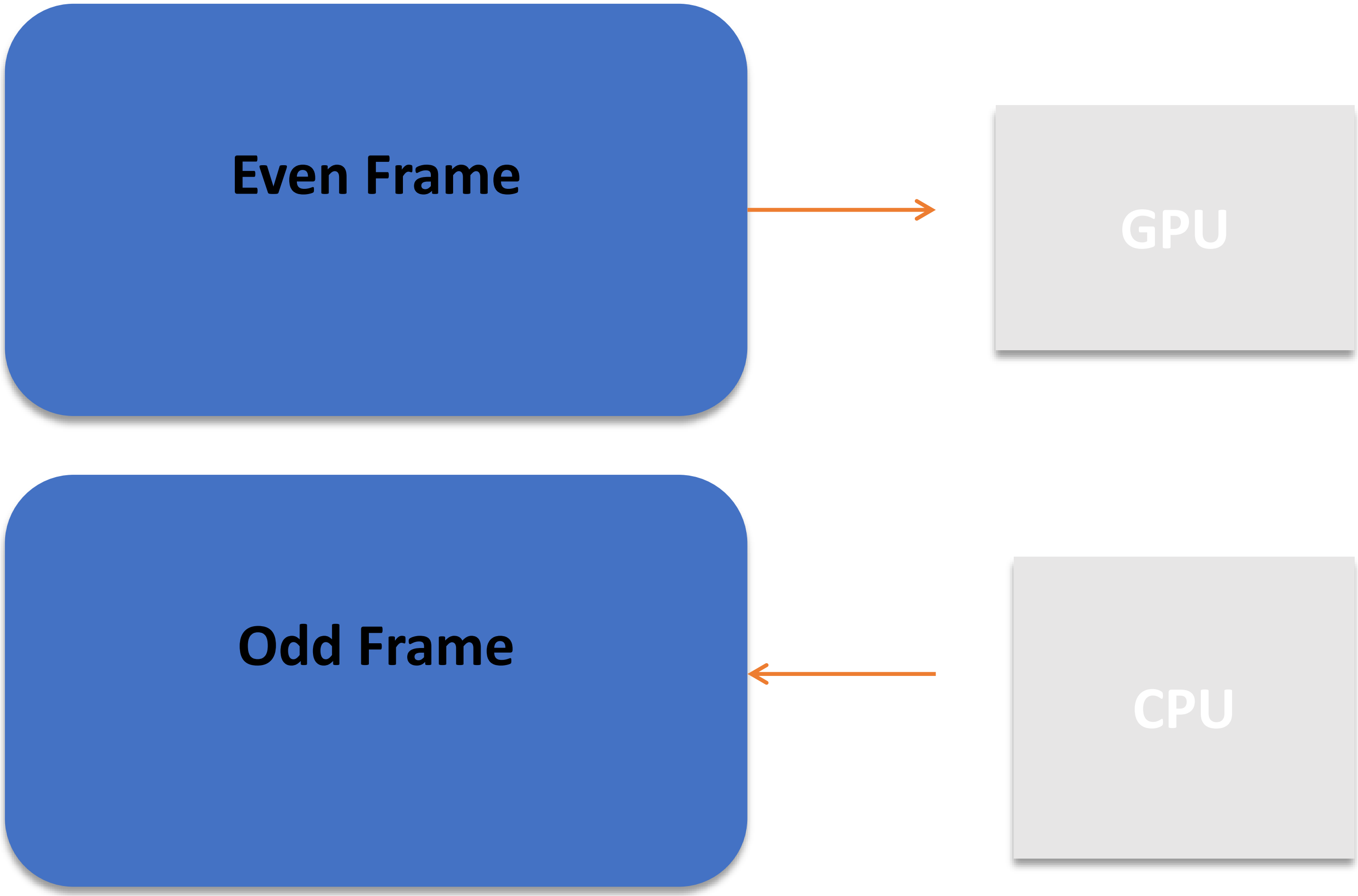
Building around Asyncronisity:

- Entire app should be exposed to concept of asyncronisity
- The concept of a frame:
  - A set of commands which will be executed on the GPU
  - A set of data which will be read by the GPU
  - This concept is fundamental in Nitrous, regardless of API



Creating a Frame, using frame data

- Create 2 copies of our frame data
- One will be read by GPU, while other is being written to by the CPU
- Must use fence to make sure CPU doesn't get ahead
- More complex situations could be explored
- Frame data includes
  - Constant Data
  - Small texture updates



Some extra stuff we will need

- Because we track hazards, we will want a few more buffers
- A delete queue – objects are not deleted, but placed in the delete queue
  - One queue per frame, once that frame is complete, items will be deleted
- A state transition queue
  - Used only when a resource is created, to transition it to the desired initial state
- An Unordered Command Queue
  - Gets flushed before main frames command queue
  - Useful for preparing resources for first time use (e.g. initialization)

Internal command format

- Nitrous has it’s own internal command format, ~20 different types of commands
- Persistent state:
  - Resource Sets
  - Shader Blocks
  - Various pipeline state
- Frame State, primary construct is a draw set
  - Contains primitives, batches and shader sets
  - Batches which reference
    - Primitives
    - Shader Sets
  - Constant references are made into our frame memory
- Each one of these has a different, natural change frequency

Resource Sets

- In real world, textures are grouped
- Nitrous has 5 bind points
  - 2 for batch
  - 2 for shader
  - 1 for primitive
- VB is just a resource set
- Nitrous does not allow binding of individual textures
- Clearly, maps 1:1 to a descriptor

Space Fighter 1
(0) Albiedo
(1)Material Mask
(2) Ambient Occlusion
(3) Normal Map
(4) Weathering Map

Vertex Buffers

- Nitrous does not use Vertex Buffers
- Instead, Resource Set acts as VB, but with more programmatic control
- Vastly simplifies engine side management
  - VBs can be saved as DDS files
  - Do not require a huge amount of loading code for slightly different Vertex Formats
  - Can fold Displacement maps and other geometry modifiers into Primitive Resource Set
- Not seen strong evidence on any hardware that this causes a performance issue

Constant BUFFERs

- Nitrous does not have concept of constant buffers
- Instead, all constant data is thrown out every frame
  - When we render an object, CPU will generate the constants needed for that frame
  - Grab a piece of the Frame Memory and write to it
- Constant bindings are just references into our frame memory
- But… be careful! CPU could be writing straight to GPU memory. Do NOT read it back!
- Evidence suggests no performance advantage of persisting constants across frames, regenerating every frame is ample fast. 100k+ batches not a problem

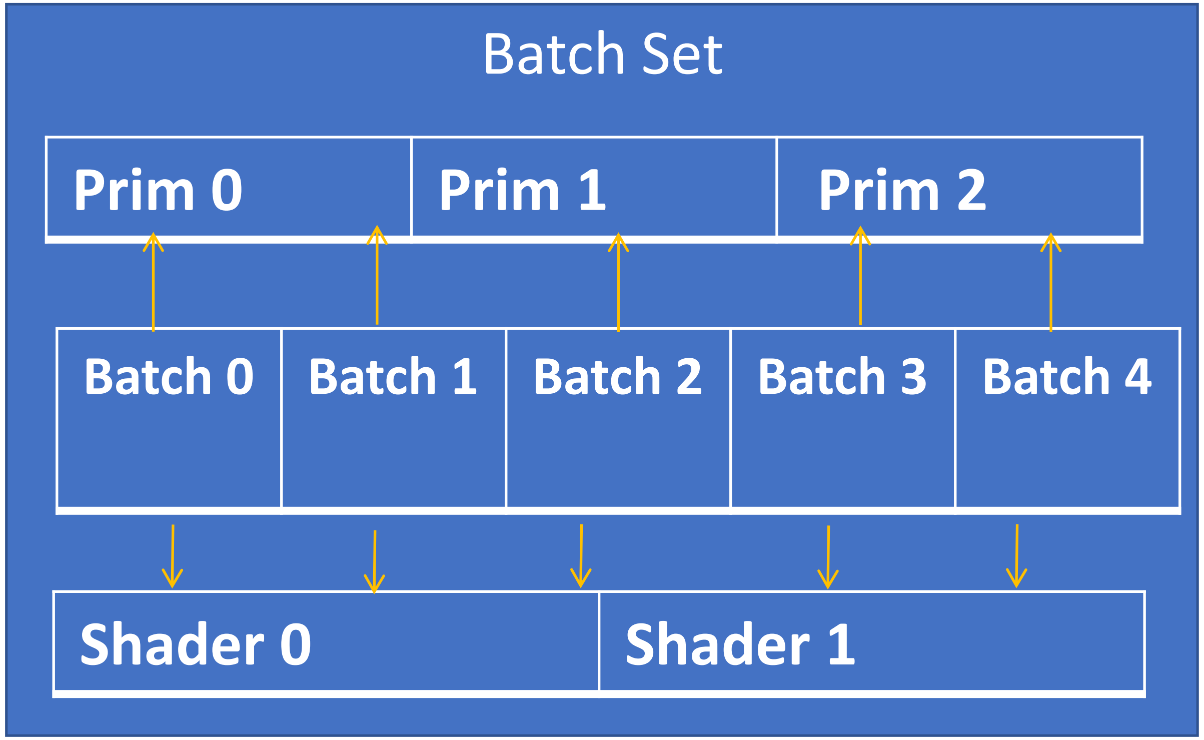
Draw call in Nitrous consists of 4 parts

Primitive
IB
Resources
Tri info

Batch Set
Batches
Primitives
Shaders
RTs
Blend State

Batch
Primitive
Shader
Resources (2)
Constants (2)

Shader
Resources (2)
Constants (2)
Shader Block



Descriptor 0
*Batch Resource Set 0
*Batch Resource Set 1
Batch Constants 1
Batch Constants 2
*Shader Resource Set 0
*Shader Resource Set 1
Shader Constants 0
Shader Constants 1
*UAV
*Samplers (only 1 global bank)

Descriptor 1
*Primitive VB

Dynamic Const
Batch Constants 0

What our frame submission looks like

1) Block on last frames present's job (e.g. NOT the fence, the actual job we spawned)
2) Process and pending resource transitions from newly created resources
3) Generate all pending unordered commands, by generating into 1 or more cmd buffers
4) Send signals to the issuers of unordered commands, to notify them the commands are submiitted
5) Begin translation of Nitrous cmds into Vulkan cmds – usually 100-500 jobs across all cores
6) Flush the deletion queues for this frame (likely a few frames old at this point)
7) Any item in our master deletion queue, add to the now empty deletion queue for this frame
8) Handle memory readbacks
9) Spawn Present job

Ashes of the Singularity

- Ashes of the Singularity: Escalation in Vulkan

Conclusion

- Vulkan is ready for primetime for desktop
- D3D11 to Vulkan is about the same complexity of D3D11 to D3D12
- For us, application/engine level makes no distinction between APIs
    - Only base level graphics layer knows what API is being used
- If moving from D3D11 to next gen APIs, both could be supported simultaneously without massive extra effort