Performance and Memory Post Mortem for Middle-earth: Shadow of War

Piotr Mintus
Technical Director, Monolith Productions

MONOLITH

Shadow of War

MONOLITH

MIDDLE-EARTH
SHADOW OF WAR

Shadow of War

- 3rd person action adventure game
- Retail on October 10th 2017
- Platforms: Xbox One, Xbox One S, Xbox One X, Playstation 4, Playstation 4 Pro, PC Win32, PC UWP.
- Target console frame rate: 30 fps (33.3ms)
- Target PC frame rate: 60 fps (16.6ms)

Shadow of War

- Nemesis System
  - Core design pillar
  - Unique AI
    - Personalities
    - Traits
    - Visuals

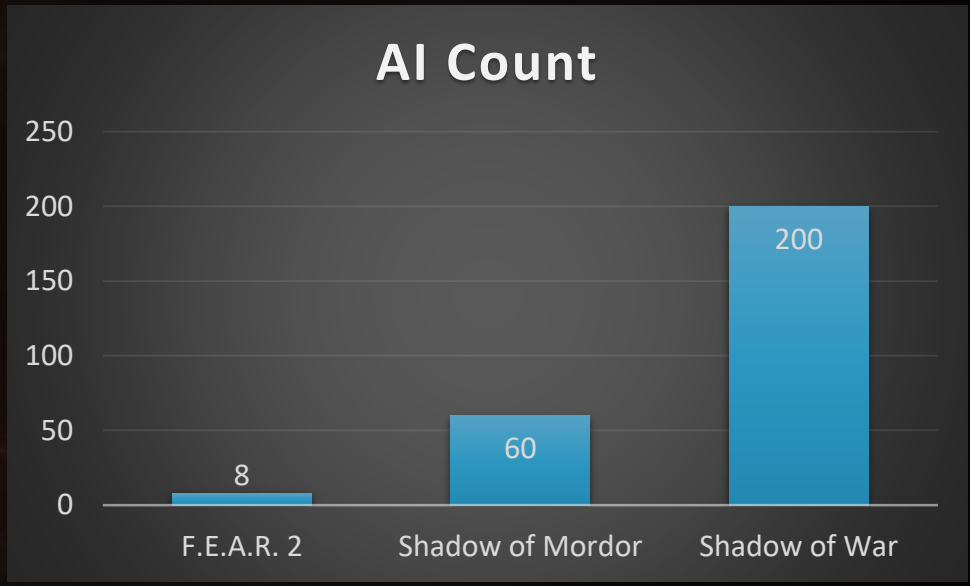
Shadow of War

1. Performance Optimizations
2. Memory Optimizations

Performance Optimizations

The Cost of War

- Shadow of Mordor was a challenge to move from 8 active AI to 60 since F.E.A.R. 2
- Shadow of War's design pushed this challenge even further by moving to 200 active AI
- This becomes both a problem with executing AI logic and rendering 200 unique AI necessitated by the Nemesis System.



The Cost of War

	Shadow of Mordor	Shadow of War
AI Count	60	200
Mesh per AI (LOD0)	4	48
Mesh Density	1x	5x
Bones per Mesh	32	64

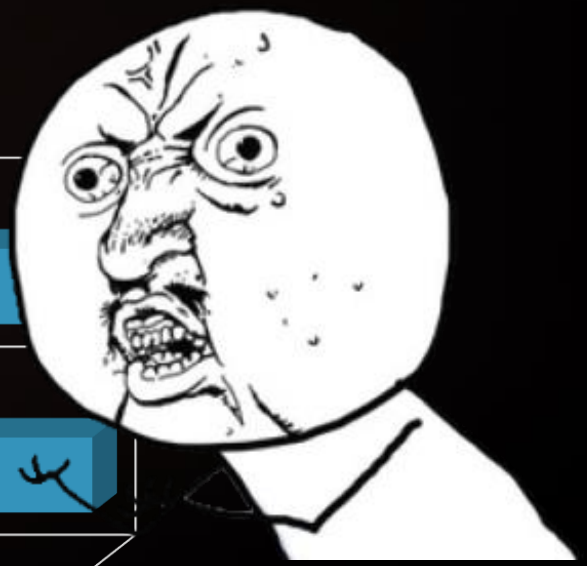
48 pieces X 64 bones = 768 bones per character

The Cost of War

	Shadow of Mordor	Shadow of War
FX	~2ms on CPU	~20ms on CPU
GPU Particle Emitters	10 - 20	5000+
NavMesh	371K triangles	1.1M triangles 76 pathing sets in 8 clearance caches

The Cost of War

- At its worse, the game ran at 90ms per frame on the consoles.



Threading

Threading

- In order to get 200 AI simulating at the same time we had to thread many game systems from Shadow of Mordor.
- This was a major undertaking that consumed a large portion of the engineering team and took most of the project.

Threading

Thread	Description
FX	Background thread to process FX on the CPU
AI	AI updating to support 200 AI
Path Finding	Background path finding thread
Path Region	Background path region updating thread
Fire Simulation	Fire simulation thread
Player and AI Motion	Player and AI motion system

Threading – Synchronization Primitives

- Moved away from using kernel primitives in favor of lightweight atomic spin locks for atomic data access protection.
- When a context switch is desired a kernel primitive is still used.
- The real cost of a context switch is cache eviction.

Threading – Synchronization Primitives

- Introduced a lightweight multiple readers single writer primitive
- Reading the state of the physics simulation from multiple threads is an example.
- Microsoft platforms have a Slim Reader/Writer (SRW) Locks primitive.
- Sony has an equivalent primitive.

Threading – Synchronization Primitives

- Can be implemented using two atomic spin lock instances and an atomic counter to track the number of readers.
  - First atomic spin lock is used as a reader lock
  - Second atomic spin lock is used as a writer lock

Threading – CPU Clusters

- Separated workloads between the two CPU clusters on the consoles to take advantage of the shared L2$.
  - The first CPU cluster executes game play logic.
  - The second CPU cluster executes rendering logic.

Threading – CPU Clusters

- Due to the increased complexity of Shadow of War this resulted in a 10% performance gain.
- Requires that the clusters are not touching the same cache lines at the same time.
  - Black Magic.

Threading – The Monolith Approach

- Keep jobs large
  - Keep as much code single threaded as possible
  - Lowers the level of entry for junior engineers
  - Lowers the amount of bugs introduced by concurrency
  - Requires less overall synchronization
  - Typically results in better use of the CPU cache

Threading – The Monolith Approach

- Move entire systems over to background threads
- Set hard affinities for large systems

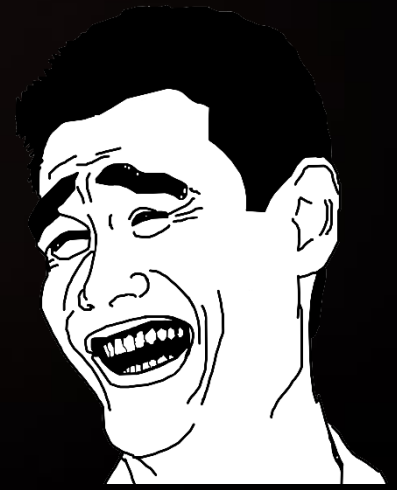
Threading – The Monolith Approach

- We fill the CPU gaps with lower priority threads.
  - Similar to the concept of Async Compute but for a CPU core.
  - File I/O, streaming, and asynchronous ray casts are examples.

Threading – The Monolith Approach

Threading – The Monolith Approach

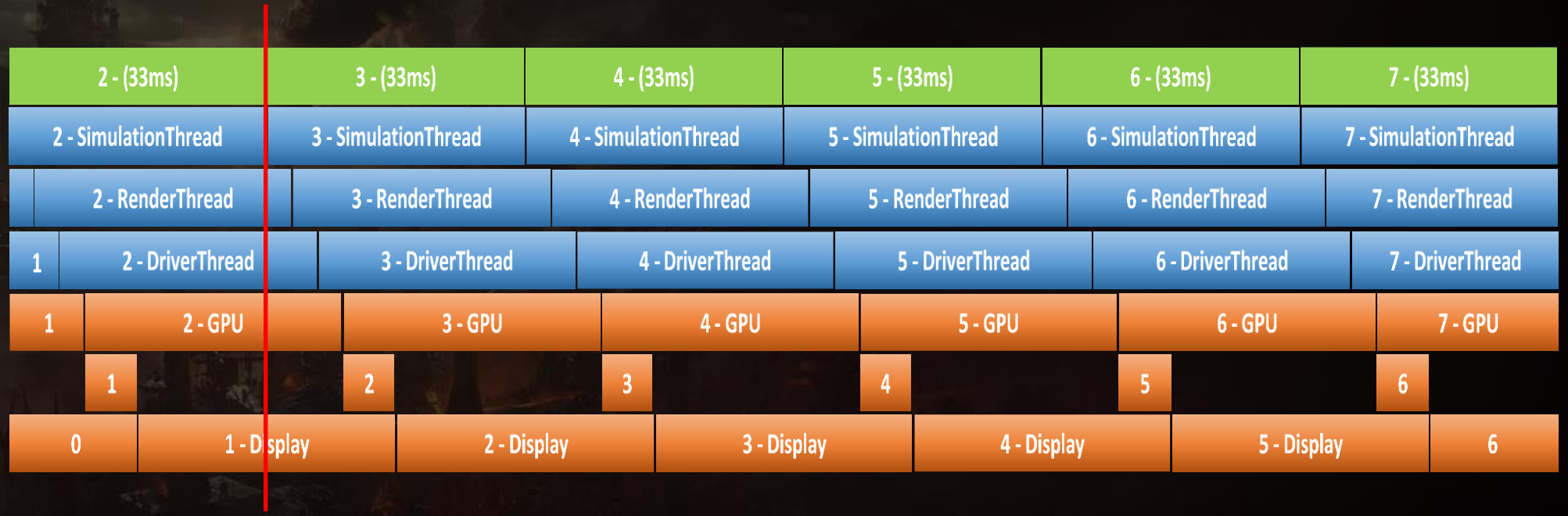
The **Monolithic** Approach



Threading – Pipelining

- Operations are pipelined through the use of circular command buffers.
- The pipeline pushes operations in a single direction.
- Each one of these command buffers are then executed on a dedicated thread that runs on a dedicated core.
- It allows each pipelined stage to get the full 33.3ms per frame

Threading – Pipelining



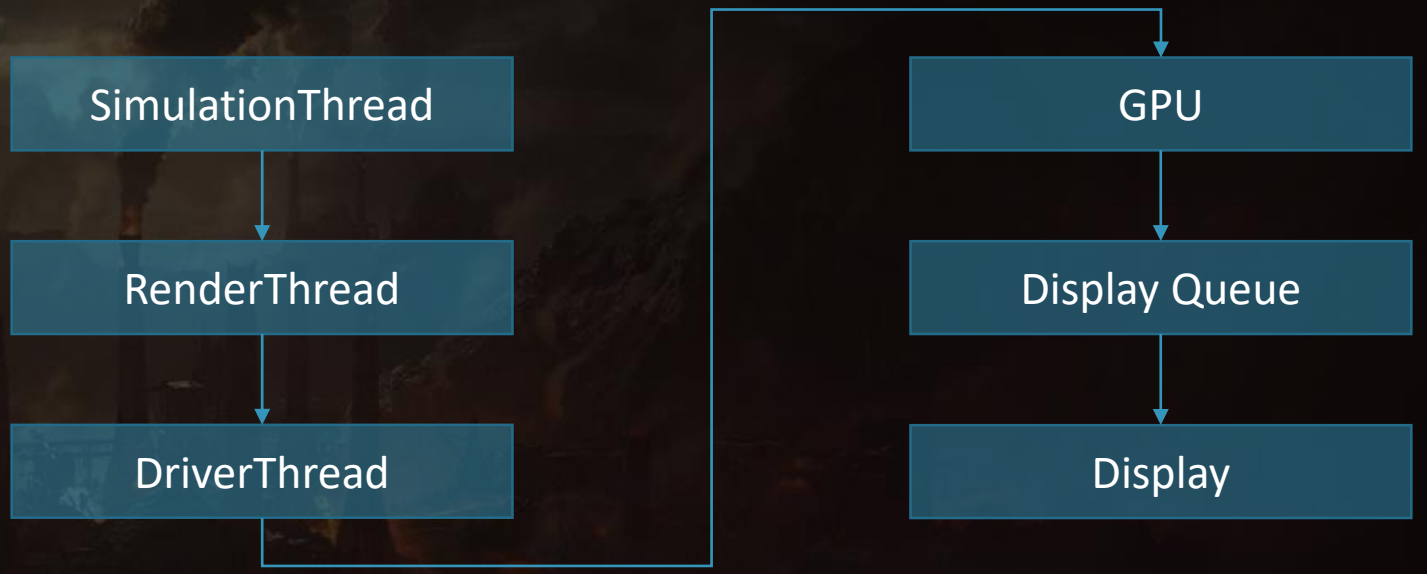
Threading – Dynamic Frame Pacing

- When pipelining threads, we guarantee that the next stage in the pipeline is no more than 1 frame behind (33.3ms).
- Ideally all pipelined stages are running in parallel, with a few milliseconds offset between each.
- This will be true if the entire pipeline is bound by the first stage in the pipeline.

Threading – Dynamic Frame Pacing

- If the entire system is bound by the last stage, we end up displaying frames that were simulated several frames ago.
- This is caused by the telescoping nature of pipelining
- We are always bound by the last stage when we v-sync.

Threading – Dynamic Frame Pacing



33.3ms + 33.3ms + 33.3ms + 33.3ms + 33.3ms + 33.3ms = 200ms

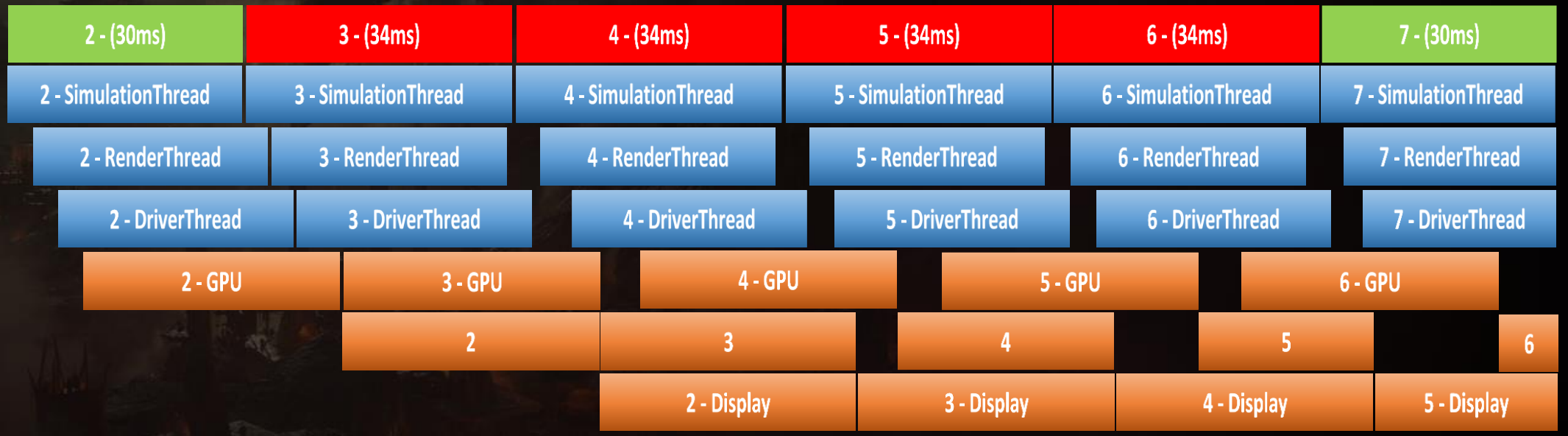
Threading – Dynamic Frame Pacing

- Input latency becomes a problem since input is evaluated on the simulation thread, up to 200ms before the frame displayed on the screen.

Threading – Dynamic Frame Pacing

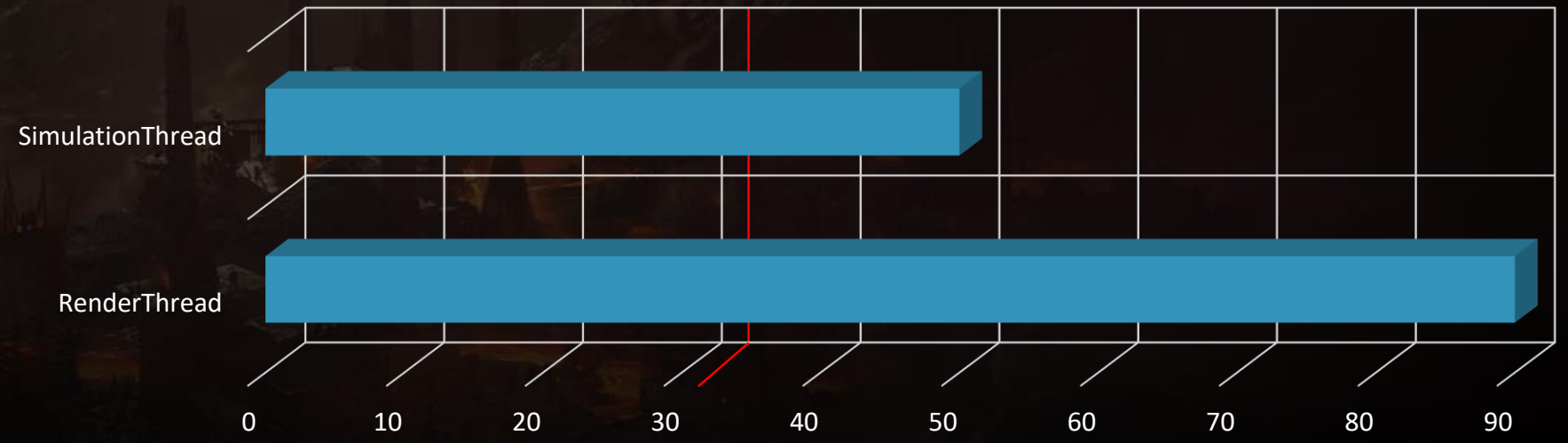
- Our solution is a dynamic frame pacing system.
- This system monitors the display queue on the GPU. If the queue is full then we are bound by the GPU.
- When we are GPU bound we artificially stall the first stage, the SimulationThread, so that it takes slightly more time than it's allotted.
  i.e., Sleep(33.3 – TimeOfSimulationThread + 1)
- This causes the telescope to contract.

Threading – Dynamic Frame Pacing



Threading

- The SimulationThread is now down to 50ms in the worst known case



Renderer

Renderer

	Shadow of Mordor	Shadow of War
Xbox One API	D3D11/D3D11.X Mix	D3D11.X with Fast Semantics
PS4 API	GNM	LCUE
PC API (minimum)	D3D11.0	D3D11.1
PC Hardware (minimum)	11_0	11_0
Console Cores	2	1.5

Renderer – Constant Buffers

- Too many copies were made between the game to engine abstraction layer.
- Each constant was set individually then assembled into a constant buffer.
- Constant buffers were regenerated for every draw call.

Renderer – Constant Buffers

- We broke up our constant buffers based on update frequency.
- Accessing constants in the renderer returns a pointer to the actual constant buffer.
- Exposed constant buffers directly to game through accessors.

Renderer – Constant Buffers

- Tracked through dirty state and frame code.
  - Once sent to the GPU, a new copy is made and the dirty state cleared.
  - Memory is not reused until the frame code is cleared by the GPU.

Renderer – Constant Buffers

- Binding named constant buffers just sets the pointer.
- Material Constant Buffers are baked into assets at cook time.

Renderer – Constant Buffers

Shadow of Mordor	Shadow of War
$Global	$Global
	Frame
	View
	Material
	RenderTarget
	CurrentBones
	PreviousBones
	Vegetation
	Wind
	TiledLighting

Renderer – Constant Buffers

48 pieces X 64 bones X 2 previous frame X 200 AI

= 307,200 bone transforms per frame

X 4 render stages

= 1,228,800 bone transforms per frame

Renderer – Constant Buffers

- Each constant buffer constant was hand sorted by acquiring heuristics on how often a constant is used.
- Rarely used constants and constants that were typically set to 0 were sorted to the bottom of the constant buffer.

Renderer – Constant Buffers

- Allocated constant buffers only big enough to hold used constants and constants that were not 0.
  - This drastically lowered the amount of memory accessed.
  - Reading past the end of a buffer on the GPU just returns 0.
  - Causes Graphics API error spam but it can be suppressed.

Renderer – Constant Buffers

- Cached constant buffers with render nodes to reuse at different stages if they haven’t changed.
  - Bones for G-Buffer stage and CSM stages

Renderer – General Optimizations

- Switched to faster 1st party Graphics APIs
  - D3D11.X with Fast Semantics / LCUE
- Removed all frame-to-frame reference counters.
- Lifetimes managed using frame codes only.
- Cached entire Graphics API state.
  - Removed redundant state changes to 1st party Graphics APIs.

Renderer – General Optimizations

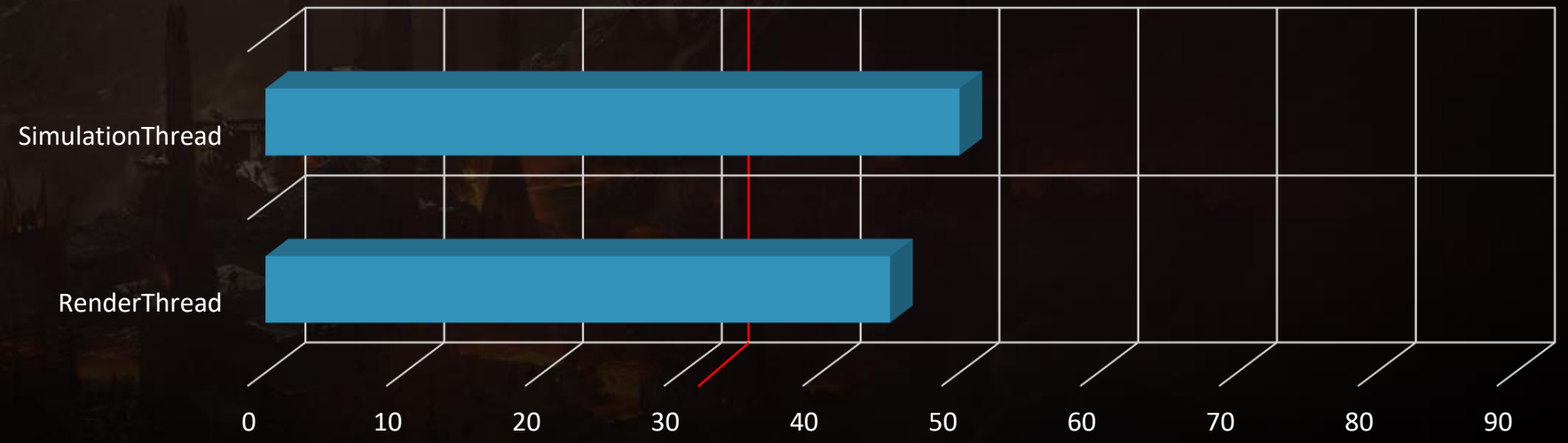
- Reduced CPU and GPU cache flushing by allocating dynamic GPU memory on cache line boundaries and padding to the end of cache lines then tracking this memory with a frame code for CPU access.

Renderer – General Optimizations

- Dynamic CPU load scaling
  - Pushing out LODs based on CPU load to lower mesh counts
  - Pausing high mip streaming to lower CPU usage, memory pressure and the cost of physical page mapping.

Renderer

- The RenderThread is now down to 45ms in the worst known case

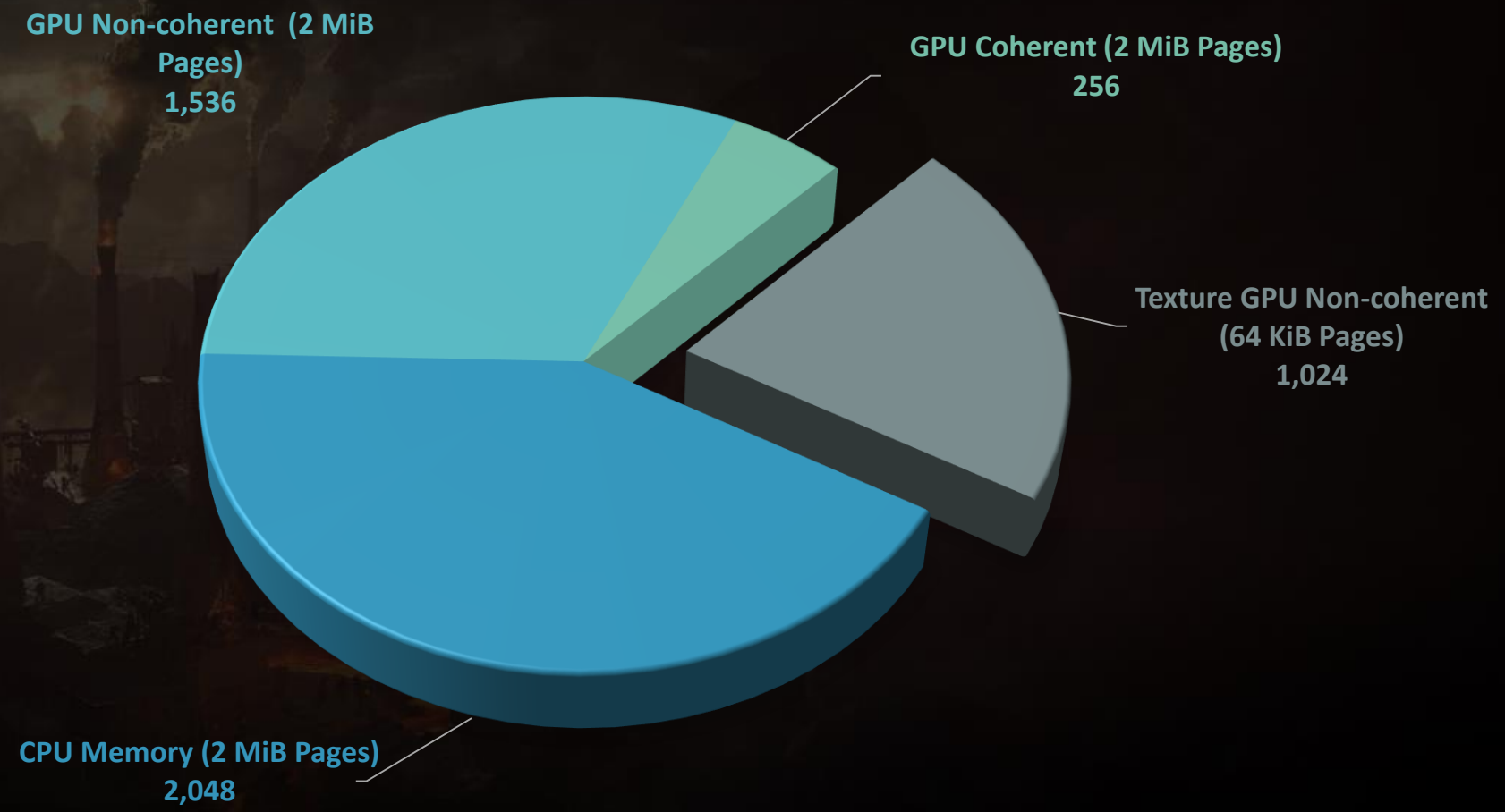


Performance – Memory

Performance – Memory

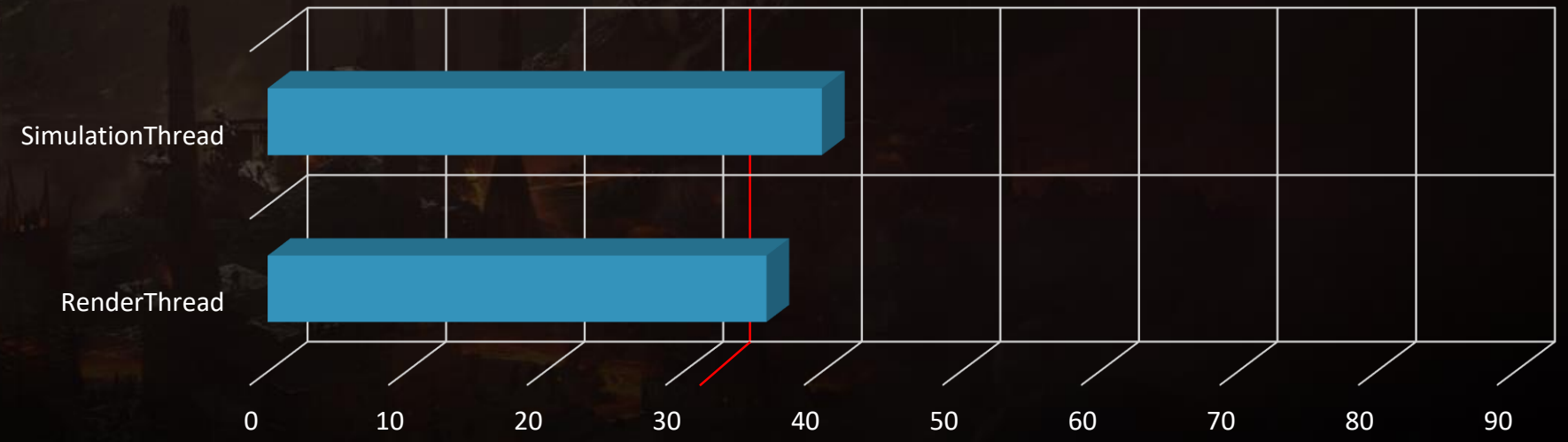
- A huge performance win for Shadow of War was switching all allocations to large 2MiB pages.
  - A 20% performance gain over 64KiB pages.
  - Large pages reduce Translation Lookaside Buffers (TLBs) misses which are very costly.
- We preallocate all large pages at process creation.
- Implemented this for PC too, although PC is almost always GPU bound.

Performance – Memory



Performance – Memory

- The SimulationThread is now down to 40ms and the RenderThread is now down to 36ms in the worst known case



Performance – The Final Blow

Performance – The Final Blow

- Development Builds
  - DLLs are used for improved engineering iteration
  - Incremental linking is enabled on all projects
  - Debug:FastLink is available and used by some engineers
  - The executable is a tiny stub that loads and runs these DLLs

Performance – The Final Blow

- Retail Builds
  - DLLs now compile as LIBs
  - Incremental linking is disabled
  - The executable is still a tiny stub that now links in the LIBs.
  - Improves runtime performance by about 10%

Performance – The Final Blow

- Retail Builds
  - LTCG is enabled on all platforms and all projects, including all middleware.
  - LTCG on Microsoft platforms gave us another 10% improvement and about a 5% improvement on the PS4.
  - PGO is enabled on all platforms.
  - PGO improved performance by about 5% on all platforms.
  - Make sure to disable COMDAT folding on Microsoft platforms.
    - It will counter the gains from PGO.

Performance – The Final Blow

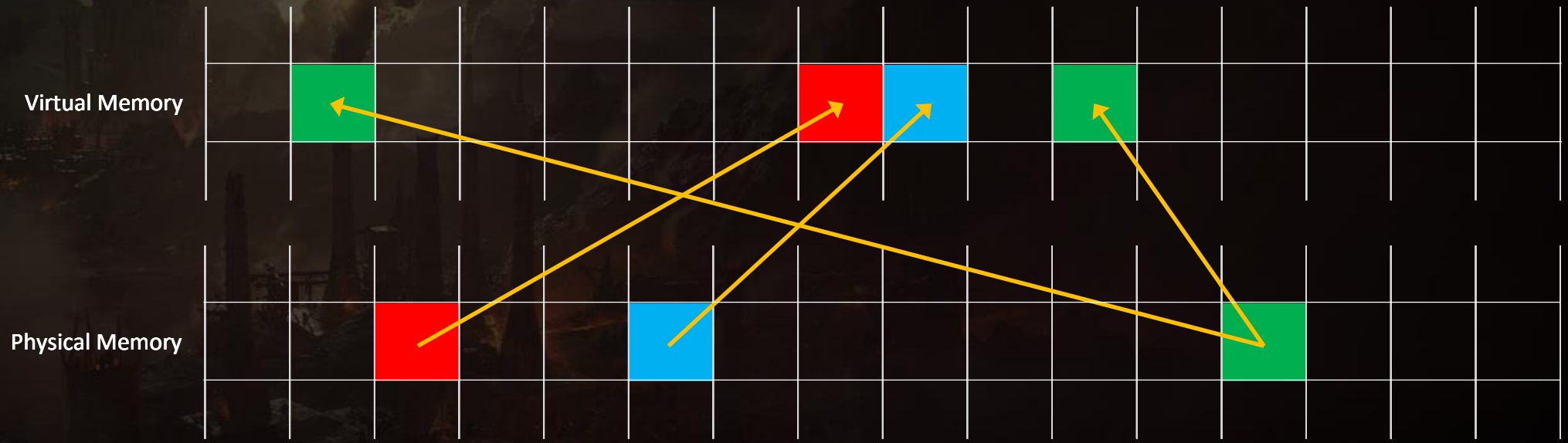
- The SimulationThread is now down to 33ms and the RenderThread is now down to 30ms in the worst known case

Memory Optimizations

Memory – GPU Virtual Memory

- Modern GPUs use virtual memory just like CPUs.
- Physical memory does not have to be contiguous.
- Physical memory can be mapped and unmapped at page granularity.
- A single physical memory page can be mapped to multiple virtual memory addresses.

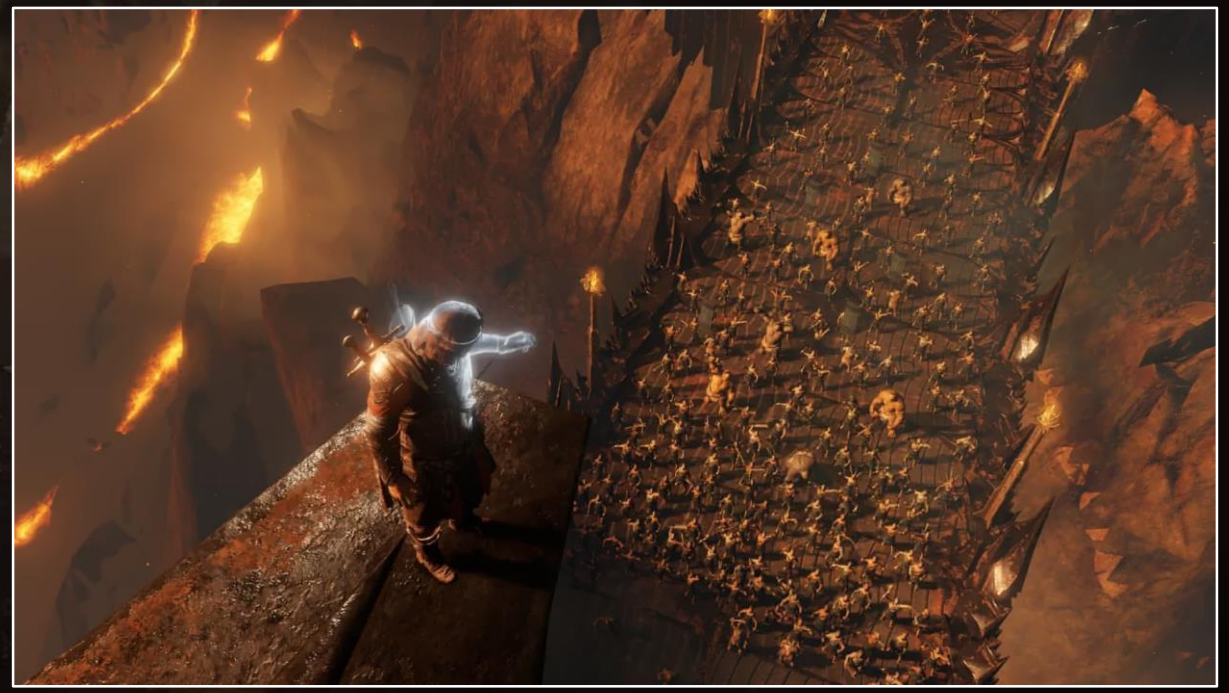
Memory – GPU Virtual Memory



Memory – 64KiB Pages

- The advantage of using 64KiB pages is that they are smaller than larger pages and allow for greater sharing and reuse.
- The disadvantage to using 64KiB pages is that they are slower to access due to increased TLB misses.

Memory – Mipmap Streaming



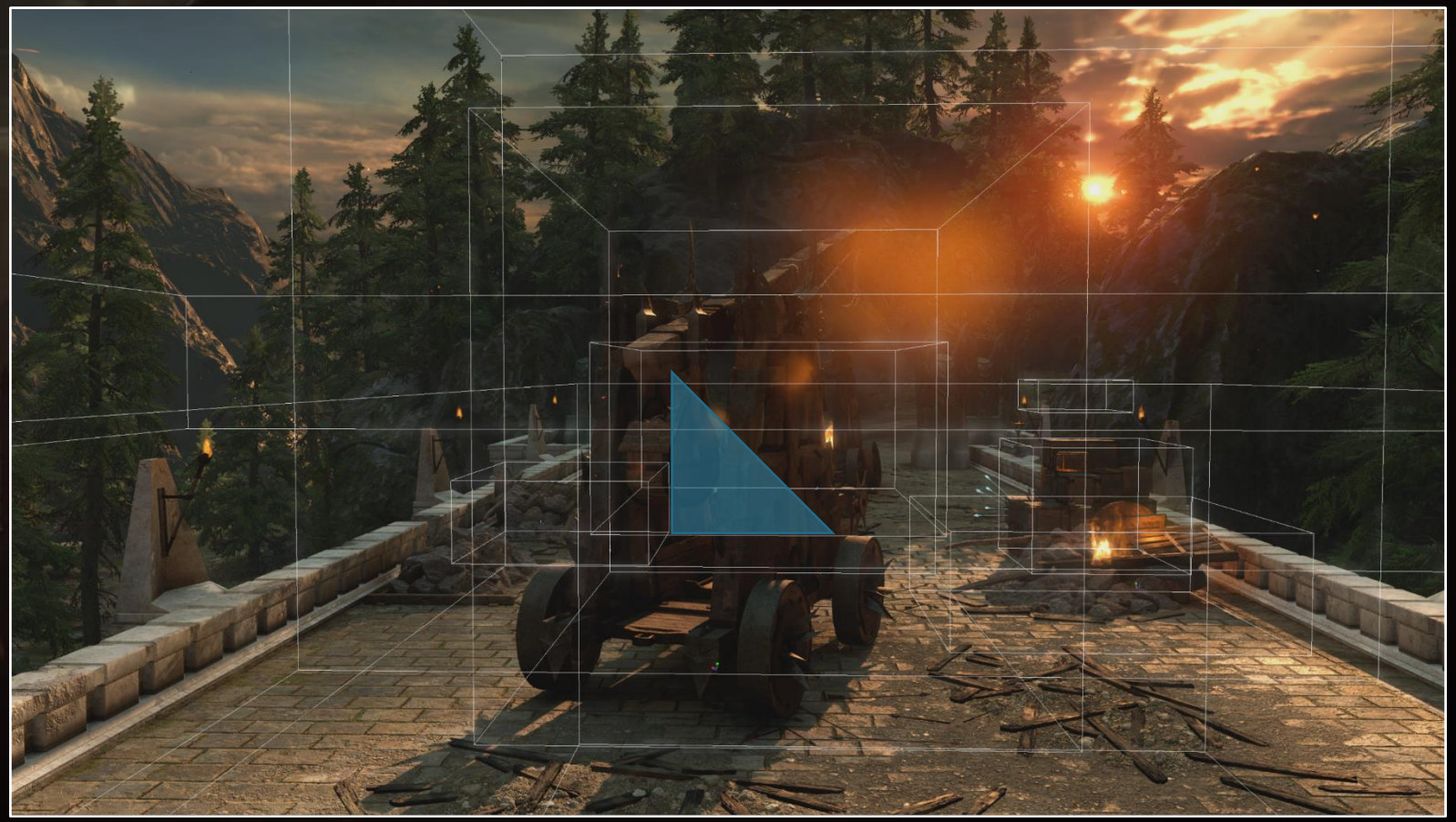
Memory – Mipmap Streaming

- On Shadow of Mordor we streamed mipmaps in but never unloaded them.
  - This was mostly done to lower load times.
  - If a texture was used, its high mip was streamed in
- On Shadow of War we needed to stream high mipmaps in and out to save on memory.
  - We used a fixed mipmap memory pool.
  - The high mip is 66% of the memory for a Texture2D.

Memory – Mipmap Streaming

- At cook time we analyze every mesh and determine the largest triangle with the largest texel density. We save it off.
- At runtime, when rendering a mesh, we project this triangle into screen space using the CPU and calculate the approximate mipmap value.

Memory – Mipmap Streaming



Memory – Mipmap Streaming

- The CPU system is throttled by a number of meshes/materials that can be analyzed per frame (64), by a frame code per mesh, and by a dampener to avoid thrashing.
  - 1920 meshes tested every second
- The CPU cost of mipmap analysis is fixed at 0.1ms per frame.

Memory – Mipmap Streaming

- The high mips use a pool of 64KiB pages. This pool of pages are preallocated at process creation.
- High mips are loaded until the pool is depleted.
- All textures that support high mip streaming are created without physical memory backing for their high mip.

Memory – Mipmap Streaming

Memory – Mipmap Streaming

- Conclusion
  - Using a pool of memory for our high mips saved about 1.0GiB of memory.

Memory – Texture2DArray

MONOLITH

MIDDLE-EARTH™ SHADOW OF WAR

Memory – Texture2DArray

- Shadow of War uses a fair amount of Texture2DArrays.
  - Terrain
  - Character models
  - Most structures
  - FX flipbooks

Memory – Texture2DArray

- Benefits
  - Slices within Texture2DArrays are all sampled at the same level which is great for blending.
  - It is typically easier to avoid branches in the shader to sample a Texture2DArray than it is when sampling several Texture2Ds.

Memory – Texture2DArray

- Issues
  - Padding - Texture2DArrays on AMD hardware are stored per mip, per slice. This ends up padding slice counts to a power of 2. If a texture array contains 3 slices, the memory layout contains 4 slices.

Memory – Texture2DArray

- Issues
  - Duplication - If two Texture2DArrays use the same slice, that slice ends up getting duplicated in memory.
    - For example, these arrays are typically used for blending. In a snowy region everything will get blended with snow. The snow texture slice will be duplicated in many texture arrays.

Memory – Texture2DArray

- In order to get around padding we manually bind 64KiB pages to the portions of the texture that are actually read by the GPU.
- We do not back layout padding with physical memory.
- Packed Mips are mips that are smaller than a 64KiB page and share a 64KiB page. They are always backed by physical memory pages.

Memory – Texture2DArray

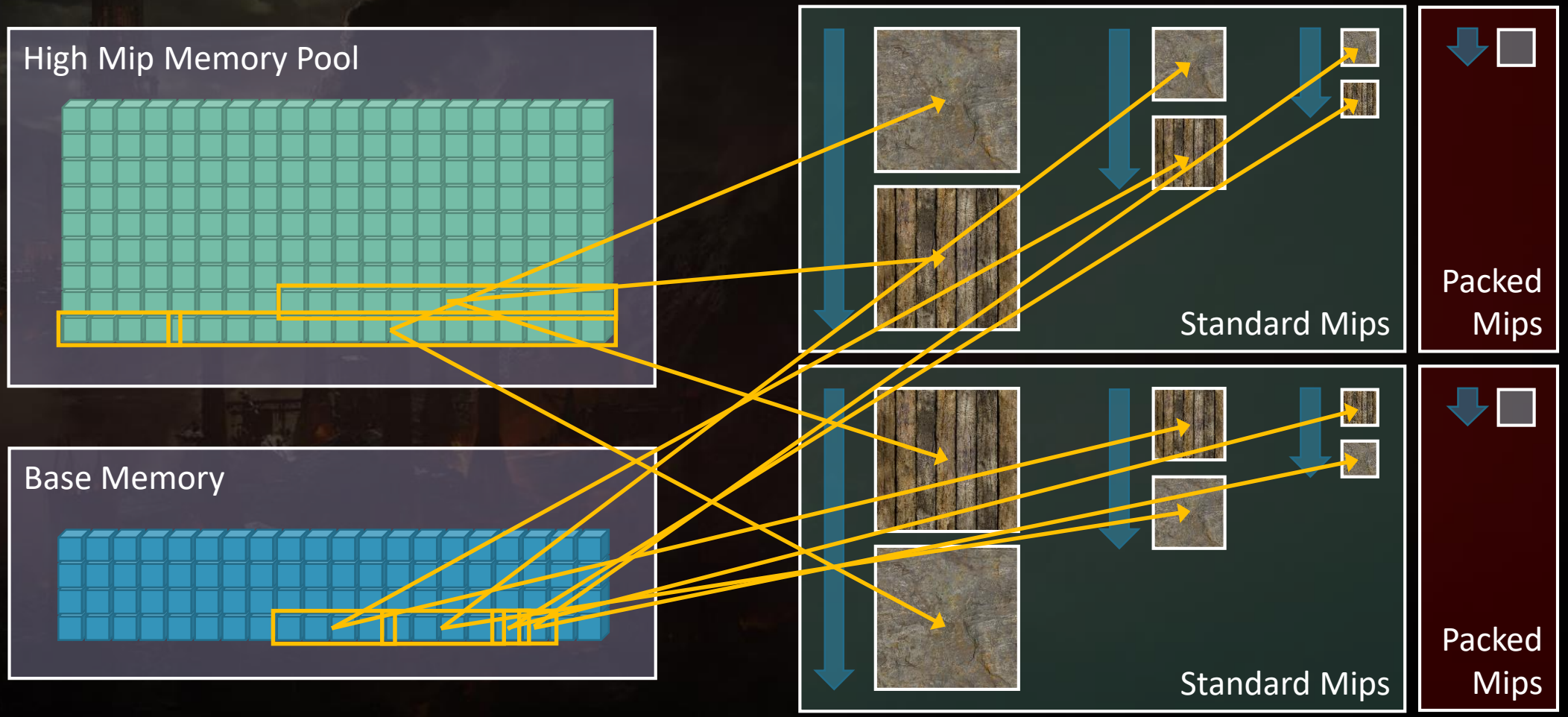
Memory – Texture2DArray

- We solved duplication by sharing 64KiB physical memory pages between the slices.
- Texture2DArrays only contain references to Texture2D slices.
- At runtime the physical pages of the slices are mapped onto the Texture2DArrays.
- Each slice is reference counted. Physical memory for the slice is not freed until all references are gone.

Memory – Texture2DArray

- Packed Mips are smaller than a single 64KiB page, and remain duplicated.
- This solution also allows us to use the slice as a regular Texture2D and share it with a Texture2DArray in memory.
- Cook times were also reduced as we only need to cook one slice at a time.

Memory – Texture2DArray



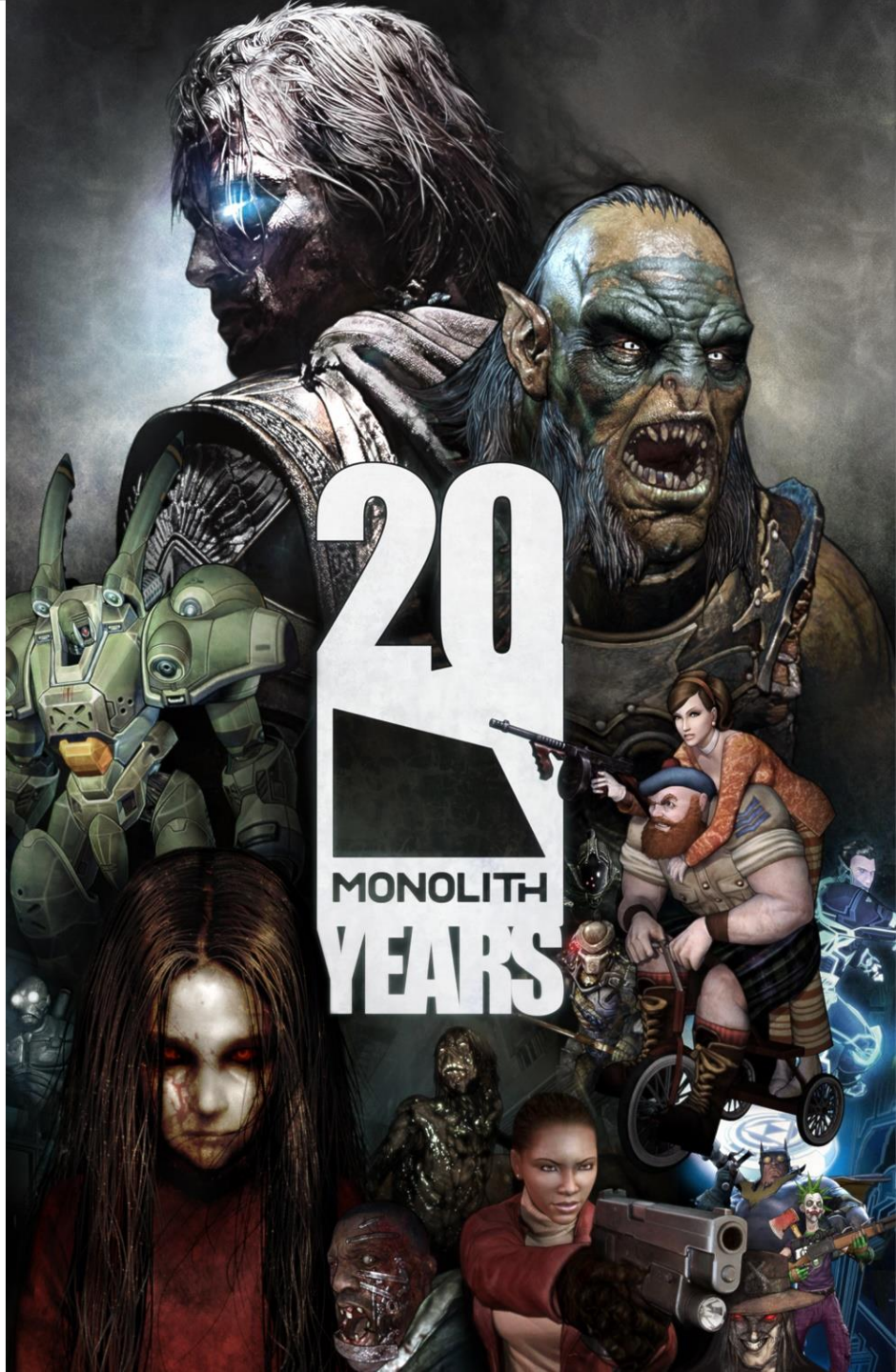
Memory – Texture2DArray

- Conclusion
  - Avoiding duplication saved us around 300MiB on average.
    - Scene dependent.
  - Avoiding padding allowed artists to create non-power of 2 arrays.

Join us!

We currently have multiple open positions across all of our departments at all levels.

https://www.lith.com/careers



MONOLITH

ANIMATION BOOTCAMP: 'MIDDLE-EARTH: SHADOW OF WAR': CREATING MULTI-CHARACTER COMBAT ANIMATIONS

John Piel, Camille Chu

Monday 11:20am - Room 2010, West Hall

SCORING 'MIDDLE-EARTH: SHADOW OF WAR': A POSTMORTEM

Nathan Grigg, Garry Schyman

Wednesday 3:30pm - Room 3006, West Hall

HELPING PLAYERS HATE (OR LOVE) THEIR NEMESIS

Chris Hoge

Thursday 4:00pm - Room 2014, West Hall

MIDDLE-EARTH: SHADOW OF WAR: ASSET PIPELINE GROWING PAINS

Doug Heimer

Friday 10:00am - Room 2010, West Hall

Thank You!

Questions?

Piotr Mintus (piotr@lith.com)
Technical Director, Monolith Productions

MONOLITH

MIDDLE-EARTH™ SHADOW OF WAR