GPU Driven Rendering and Virtual Texturing in Trials Rising

Oleksandr Drazhevskyi
Technical Lead
Ubisoft Kiev

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

Agenda

- Trials Rising Technical Overview
- GPU Driven Rendering
- Virtual Texturing Technique
- VT and GPU Pipeline Optimization
- Conclusion and Future

Trials Rising

Latest installment to the brand which features:

- Physically based gameplay
- Fully dynamic world
- Constant 60 Fps gameplay
- Focus on User Generate Content
- Game as a platform
- Advanced customization
- Competitive gameplay

Trials Rising

Latest installment to the brand which features:

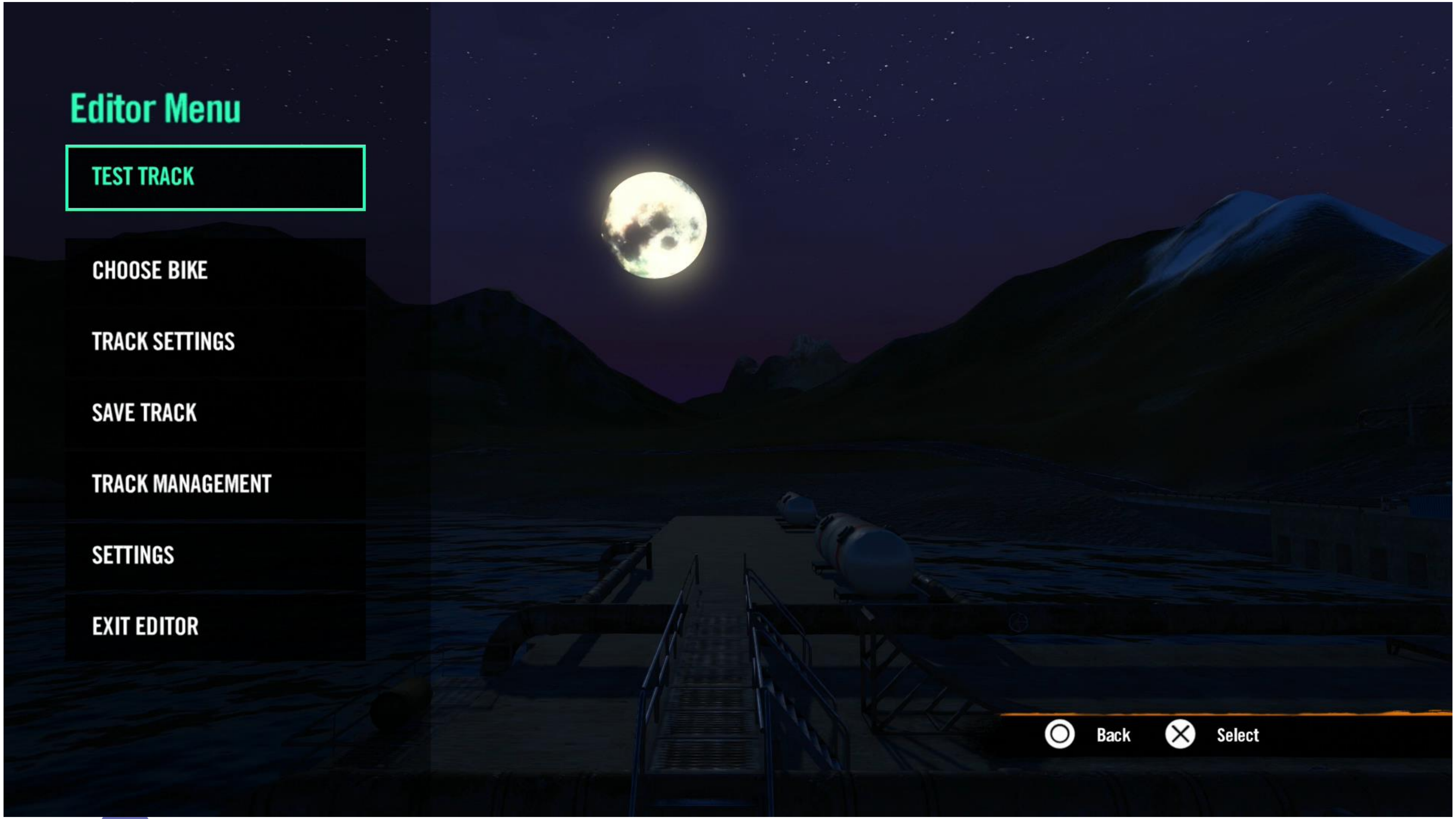
- Physically based gameplay
- Fully dynamic world
- Constant 60 Fps gameplay
- Focus on User Generate Content
- Game as a platform
- Advanced customization
- Competitive gameplay

This features drive rendering engine architecture

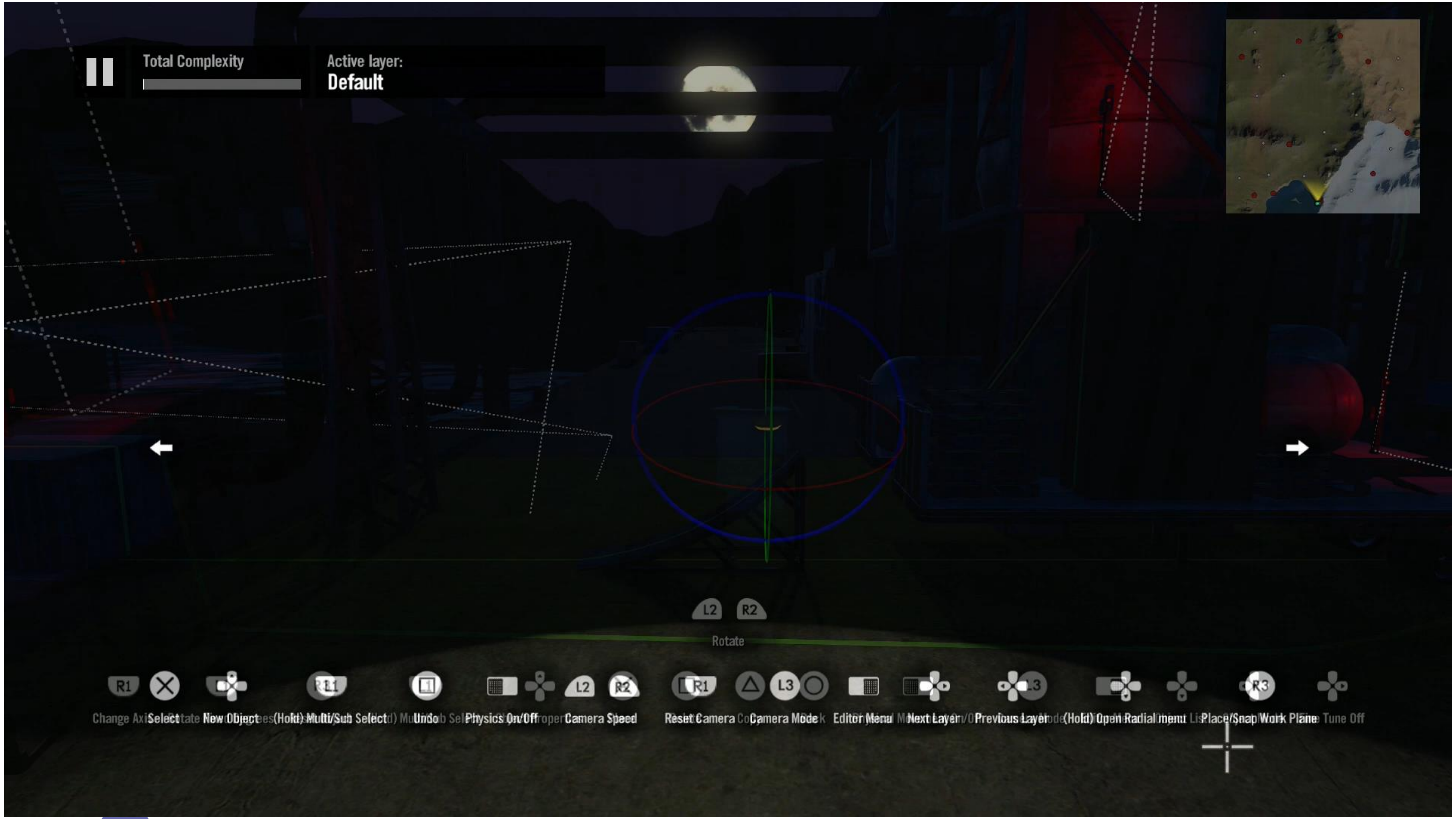
Initial Technical Requirements

- Improved visual quality
- More CPU time for engine and gameplay
- Enhanced 4k rendering
- 60 frames per second
- Improved UGC performance
- Better GPU utilization
- Multi platform scalability

UGC


Editor Menu
TEST TRACK
CHOOSE BIKE
TRACK SETTINGS
SAVE TRACK
TRACK MANAGEMENT
SETTINGS
EXIT EDITOR
Back
Select

UGC


Total Complexity
Active layer:
Default
Rotate
Select
New Object
(Hold) Multi/Sub Select
Undo
Physics On/Off
Camera Speed
Reset Camera
Camera Mode
Editor Menu
Next Layer
Previous Layer
(Hold) Open Radial menu
Place/Snap Work Plane

UGC

https://www.youtube.com/watch?v=wwrC0qNBQcc

World Structure

- Artists author micro prefabs
- Level designers populate world with macro blocks
- Small levels with high geometry complexity
- Players can do the same (UGC game)
- Blocks "density" not balances across the level
- Visual quality highly depends on macro blocks count
- Batching is extremely complicated

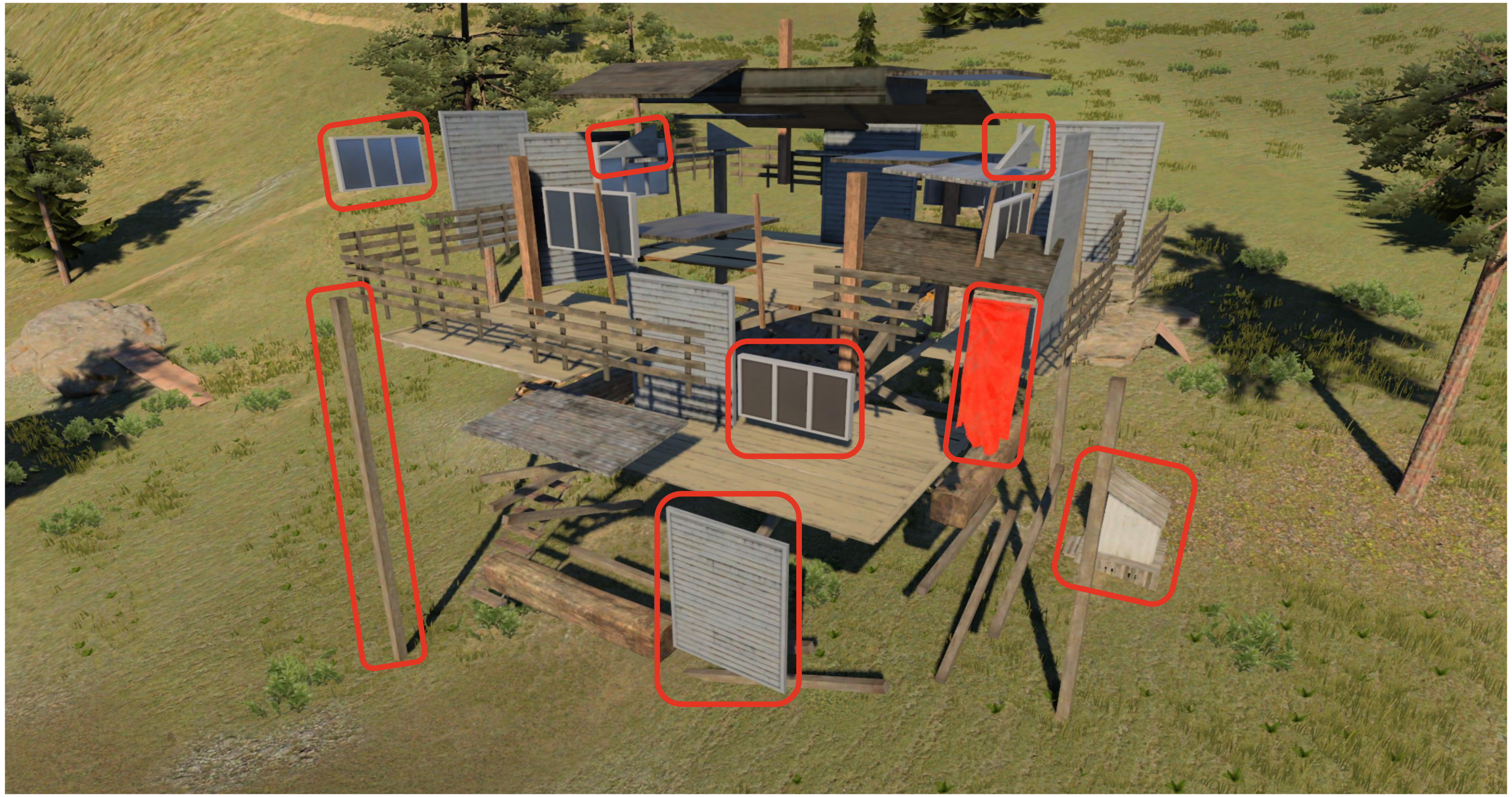
Geometry Complexity

Macro block example

Geometry Complexity

Macro block contains about 65 micro prefabs in average

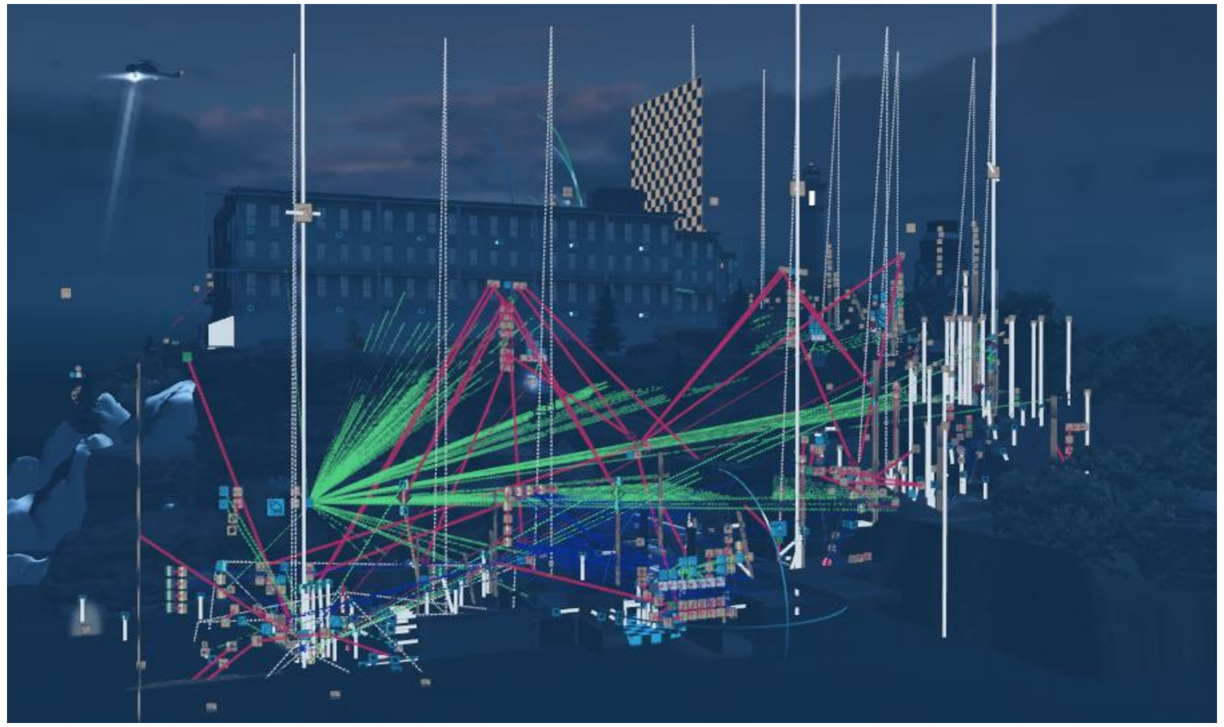
Geometry Complexity



Micro prefabs example

Engine Limitations

- About 2500 visible instances (i.e. micro blocks)
- Two CPU cores allocated for rendering
- Conservative shadows draw distances
- CPU is a huge bottleneck
- UGC levels sharing issues
- No occlusion culling on PC



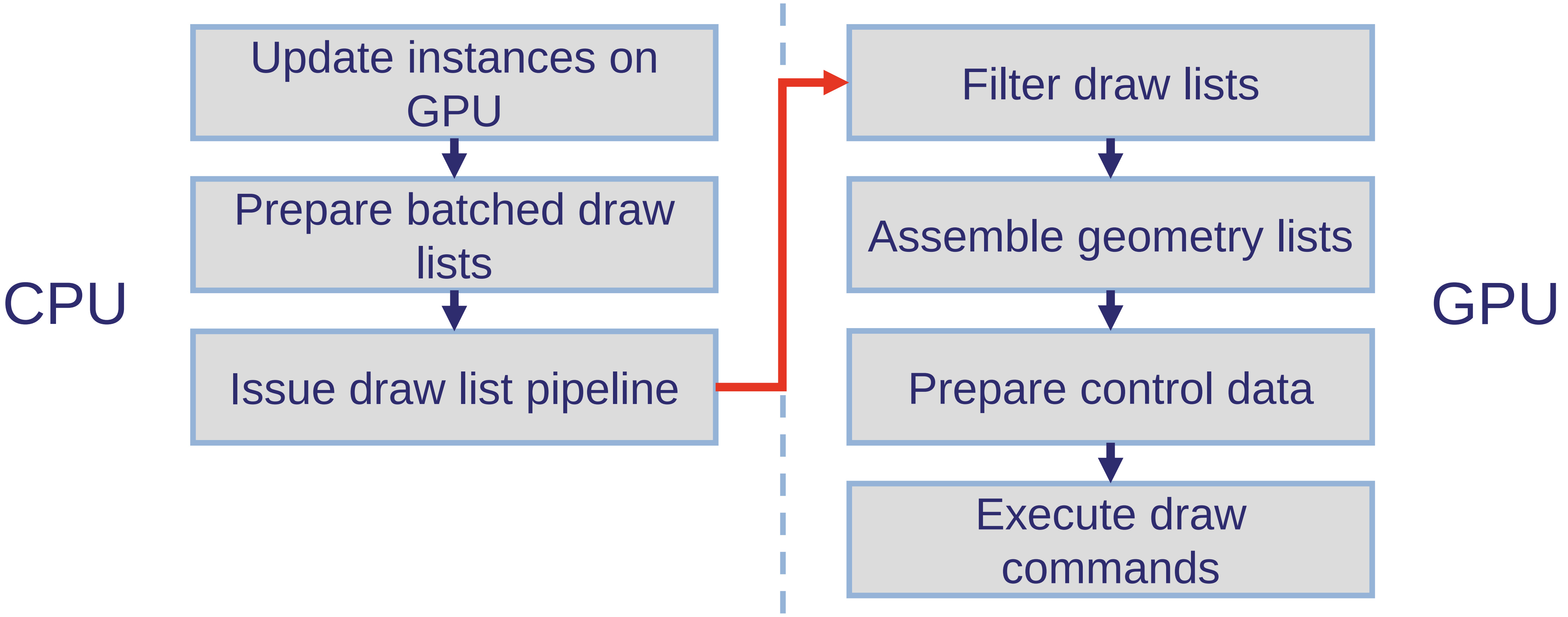

1
00:28.350

500
11
FOX

GPU Driven Rendering

- Move visibility testing to GPU
- Use testing results directly on GPU
- Batch instances on GPU
- Merge instances of different meshes together
- GPU is aware of scene state – not just a portion that passed frustrum test
- “GPU feeds itself with rendering commands”

GPU Driven Rendering



GPU Pipeline Rationale

- Perfectly fits requirements
- Few successfully shipped games
- Investment in future
- Unlocks quite a bit of optimization possibilities
- Doesn’t require “intrusive” changes in content
- Overall optimization benefits for the engine

GPU Pipeline First Iteration Outcome

- Great CPU/GPU boost
  - Not enough to hit target frame rate
  - CPU is struggling to gather and send instance data to GPU
  - A lot of CPU to GPU bandwidth pressure
  - GPU utilization decreased
- **Draw data gathering and batching is a main bottleneck**

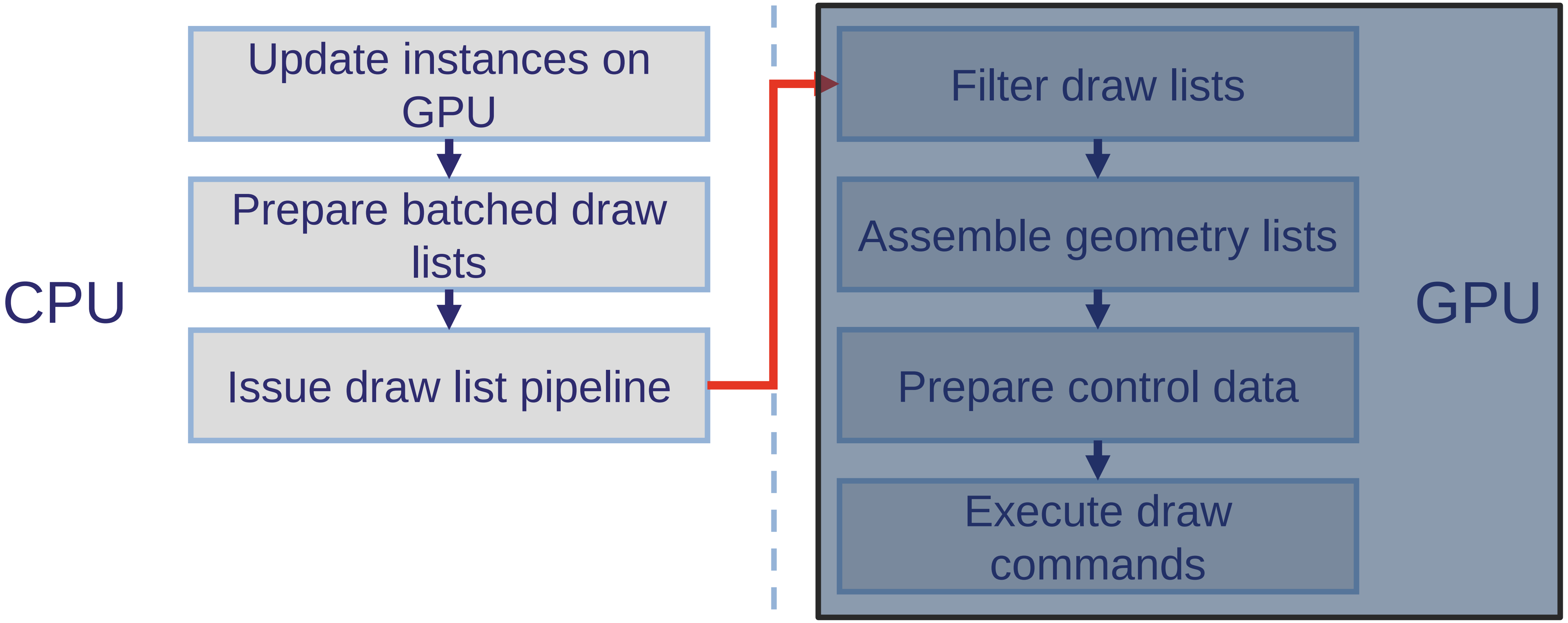
(*Xbox One numbers)

Scene	GPU	CPU
Construction Site	22.17 ms	51.43 ms

Scene	GPU	CPU
Construction Site	17.95 ms (-4.22 ms)	24.09 ms (-27.34 ms)

(*Xbox One numbers)

GPU Driven Rendering



GPU Data Structures

- Two GPU buffers to store geometry (i.e. Vertices and Indices Pool)
- One big GPU buffer to hold instance parameters (i.e. Instance Data Pool)
- Instance represented by Instance Descriptor
- One GPU buffer to store an array of all descriptors in scene
- CPU and GPU mostly operates with indices in this lists

Instance Descriptor

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset
1 byte	Transform mode
1 byte	Skinning size
1 byte	Material size
1 byte	Reserved

- Internal and external data
- Pretty much a pointer table
- Can represent any instance in scene
- Idea is pretty close to descriptor set

Instance Descriptor

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset
1 byte	Transform mode
1 byte	Skinning size
1 byte	Material size
1 byte	Reserved

Vertices pool (X Mb)			
Instance 0 vertices	Instance 1 vertices	…	Instance N vertices

Instance Descriptor

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset
1 byte	Transform mode
1 byte	Skinning size
1 byte	Material size
1 byte	Reserved

Vertices pool (X Mb)			
Instance 0 vertices	Instance 1 vertices	…	Instance N vertices

Indices pool (Y Mb)			
Instance 0 indices	Instance 1 indices	…	Instance N indices

Instance Descriptor

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset
1 byte	Transform mode
1 byte	Skinning size
1 byte	Material size
1 byte	Reserved

Vertices pool (X Mb)			
Instance 0 vertices	Instance 1 vertices	…	Instance N vertices

Indices pool (Y Mb)			
Instance 0 indices	Instance 1 indices	…	Instance N indices

Instance data pool (Z Mb)				
Instance 0 transform/history	Instance 1 transform/history	…	Instance 0 material	…

Instances Table

Instances table (X Mb)			
Instance 0 descriptor	Instance 1 descriptor	…	Instance N descriptor

- Represents simulated scene state
- Hottest data in pipeline
- CPU has own copy of the table
- Main sync point between CPU and GPU
- Used to update instance simulation state

CPU and GPU Instances State Synch

- CPU is responsible for instance state simulation and uploading to the GPU
- GPU waits for signal to read instances data
- Straightforward state regeneration every frame doesn't work
- According to the first implementation results the data traffic is quite high and tends to increase in future

```
struct InstanceDescriptor
{
public:
    // Indirection to unified vertices buffer
    uint32_t vertices;

    // Indirection to unified indices buffer
    uint32_t indices;

    uint16_t indicesSize;
    uint16_t flags;

    uint32_t culling;

    // Indirections to unified parameters buffer
    // Transform and skinning hold both
    // current and history continuously
    uint32_t transform;
    uint32_t skinning;

    uint32_t material;

    // Transform mode in first bit has a switcher
    // between current and history transform/skinning
    // Zero means common order of data:
    // Current Transform | HistoryTransform
    // Current Skinning | History Skinning
    // One means reversed order of data:
    // History Transform | Current Transform
    // History Skinning | Current Skinning
    uint8_t transformMode;
    uint8_t skinningSize;
    uint8_t materialSize;
    uint8_t reserved;
};
```

Memory Budget

Defined maximum per level instances amount is 256k

Instances table CPU	Instances table GPU
256k instances = 8 Mb	256k instances = 8 Mb

About 2.8k macro blocks – more than a target requirement

Memory Budget Per Instance

There are ~20x skinned instances compared to previous game

Instance Data Pool Static Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 0	Current skinning matrices
64 bytes x 0	History skinning matrices
16 bytes x 6	Materials specific data

Instance Data Pool Skinned Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 8	Current skinning matrices
64 bytes x 8	History skinning matrices
16 bytes x 6	Materials specific data

Average Static Instance	
224 bytes	Tolerable update cost

Average Skinned Instance	
1248 bytes	Expensive update cost

Construction Site Scene			
Static instances	93.04%	11716 Ins	~2.5 Mb
Skinned instances	6.96%	877 Ins	~1.04 Mb

Typical Scene	
Static instances	89.76%
Skinned instances	10.24%

History Transforms

Instance Data Pool Static Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 0	Current skinning matrices
64 bytes x 0	History skinning matrices
16 bytes x 6	Materials specific data

Instance Data Pool Skinned Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 8	Current skinning matrices
64 bytes x 8	History skinning matrices
16 bytes x 6	Materials specific data

History Transforms

Instance Data Pool Static Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 0	Current skinning matrices
64 bytes x 0	History skinning matrices
16 bytes x 6	Materials specific data

Instance Data Pool Skinned Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 8	Current skinning matrices
64 bytes x 8	History skinning matrices
16 bytes x 6	Materials specific data

One address + indirection

History Transforms

Instance Data Pool Static Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 0	Current skinning matrices
64 bytes x 0	History skinning matrices
16 bytes x 6	Materials specific data

Instance Data Pool Skinned Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 8	Current skinning matrices
64 bytes x 8	History skinning matrices
16 bytes x 6	Materials specific data

Transform mode controls current and history data sets

```
if (TransformMode == 0)
{
    CurrentTransform = Pool[BaseTransform];
    HistoryTransform = Pool[BaseTransform + sizeof(float4x4)];
}
else
{
    CurrentTransform = Pool[BaseTransform + sizeof(float4x4)];
    HistoryTransform = Pool[BaseTransform];
}
```

Instance Descriptor (32 bytes)	
…	
1 byte	Transform mode
…	

History Transforms

Instance Data Pool Static Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 0	Current skinning matrices
64 bytes x 0	History skinning matrices
16 bytes x 6	Materials specific data

Instance Data Pool Skinned Instance	
64 bytes	Current transform
64 bytes	History transform
64 bytes x 8	Current skinning matrices
64 bytes x 8	History skinning matrices
16 bytes x 6	Materials specific data

Savings

Average Static Instance	
160 bytes	Total: ~71.43%

Average Skinned Instance	
736 bytes	Total: ~58.97%

History Transforms

Types	Relative	Count	Memory	CPU
Static instances	93.04%	11716 Inst	~2.5 Mb	10.81 ms
Skinned instances	6.96%	877 Inst	~1.04 Mb	1.32 ms

⇩

Types	Relative	Count	Memory	CPU
Static instances	93.04%	11716 Inst	~1.79 Mb	10.49 ms
Skinned instances	6.96%	877 Inst	~0.62 Mb	0.89 ms

UpdateSubresource is slow: up to ~4ms

(*Xbox One numbers)

Instance Synchronization Rate

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset
1 byte	Transform mode
1 byte	Skinning size
1 byte	Material size
1 byte	Reserved

- Most instances don’t move at all
- Update rate depends on objects usage
- Looks reasonable to split static and skinned object in smaller groups
- Still keep only one instances table

Instance Synchronization Rate

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset
1 byte	Transform mode
1 byte	Skinning size
1 byte	Material size
1 byte	Reserved

This data changes frequently

Sort Instances by Category

Update rate (i.e. mutation rate) is a good metric for sorting

Construction Site Scene			
Immobile instances	64.21%	8086	~1.23 Mb
Mobile instances	25.09%	3160	~0.51 Mb
Mutable instances	3.74%	470	~0.08 Mb
Skinned instances	6.96%	877	~0.62 Mb

Instance Synchronization Rate

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset
1 byte	Transform mode
1 byte	Skinning size
1 byte	Material size
1 byte	Reserved

- Global transform mode
- Move LOD selection to GPU
- Technically only CB data mutates frequently

Instance Synchronization Rate

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset
1 byte	Transform mode
1 byte	Skinning size
1 byte	Material size
1 byte	Reserved

- Global transform mode
- Move LOD selection to GPU
- Technically only CB data mutates frequently

Type	Relative	Count	Memory	CPU
Mobile instances	70.11%	3160	~0.51 Mb	5.99 ms
Mutable instances	10.43%	470	~0.08 Mb	0.37 ms
Skinned instances	19.46%	877	~0.62 Mb	0.82 ms
From 3.63 Mb to 1.21 Mb				

Data Transfer Improvements

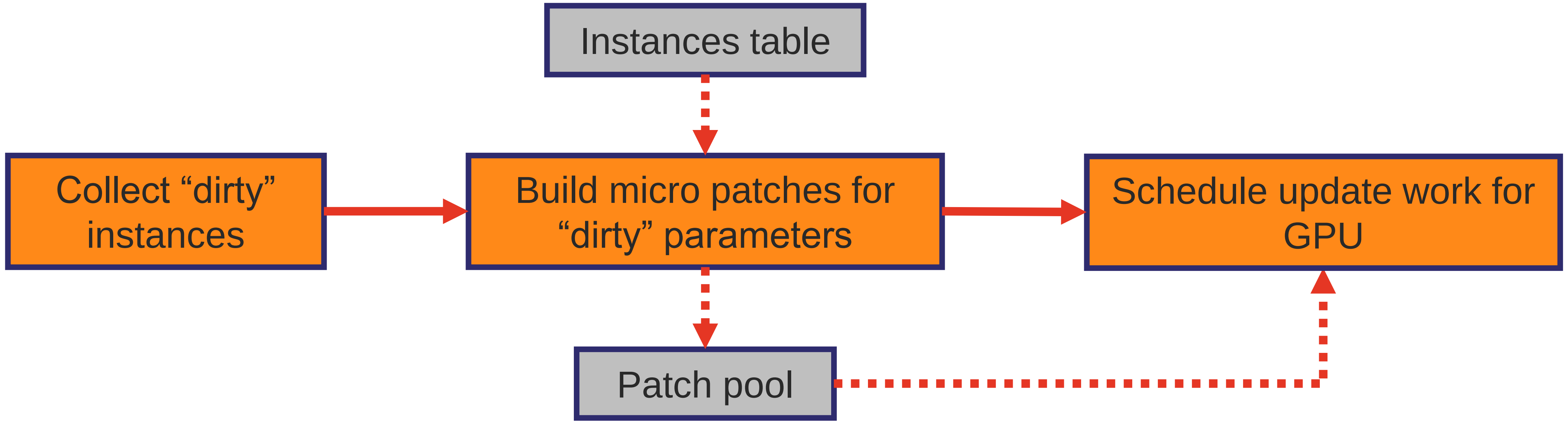
- Instance update rate is not constant
- State parameters average update rate is different and usually stable
- “Immobile” parameters is the case for specific instance categories
- Huge amount of “idling” components
- Synchronize a bare minimum of the state

Instance Descriptor (32 bytes)	
4 bytes	Vertices offset
4 bytes	Indices offset
2 bytes	Indices size
2 bytes	Flags
4 bytes	Culling data
4 bytes	Transform offset
4 bytes	Skinning offset
4 bytes	Material offset

Flags modified only if object state “changed”

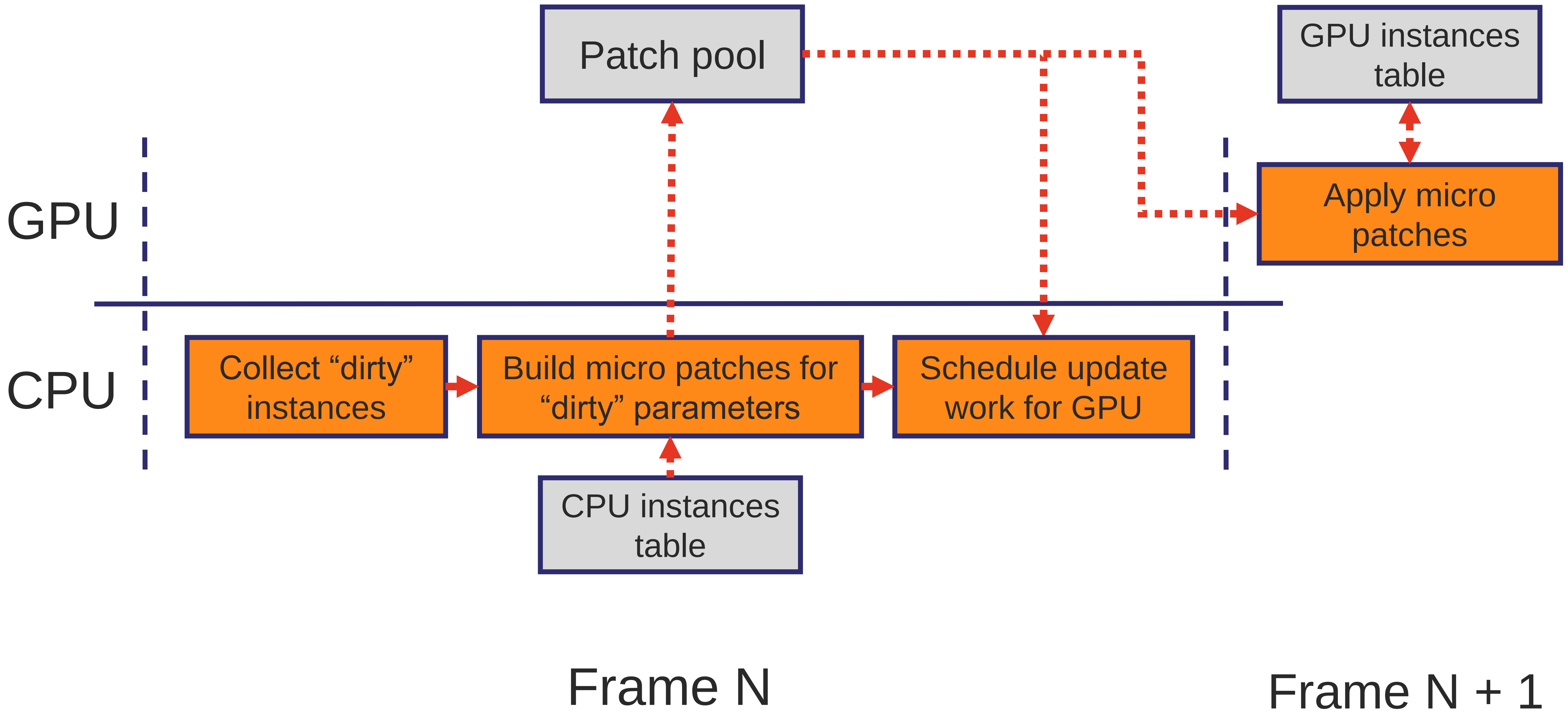
Skinning modified every frame

Micro Patches

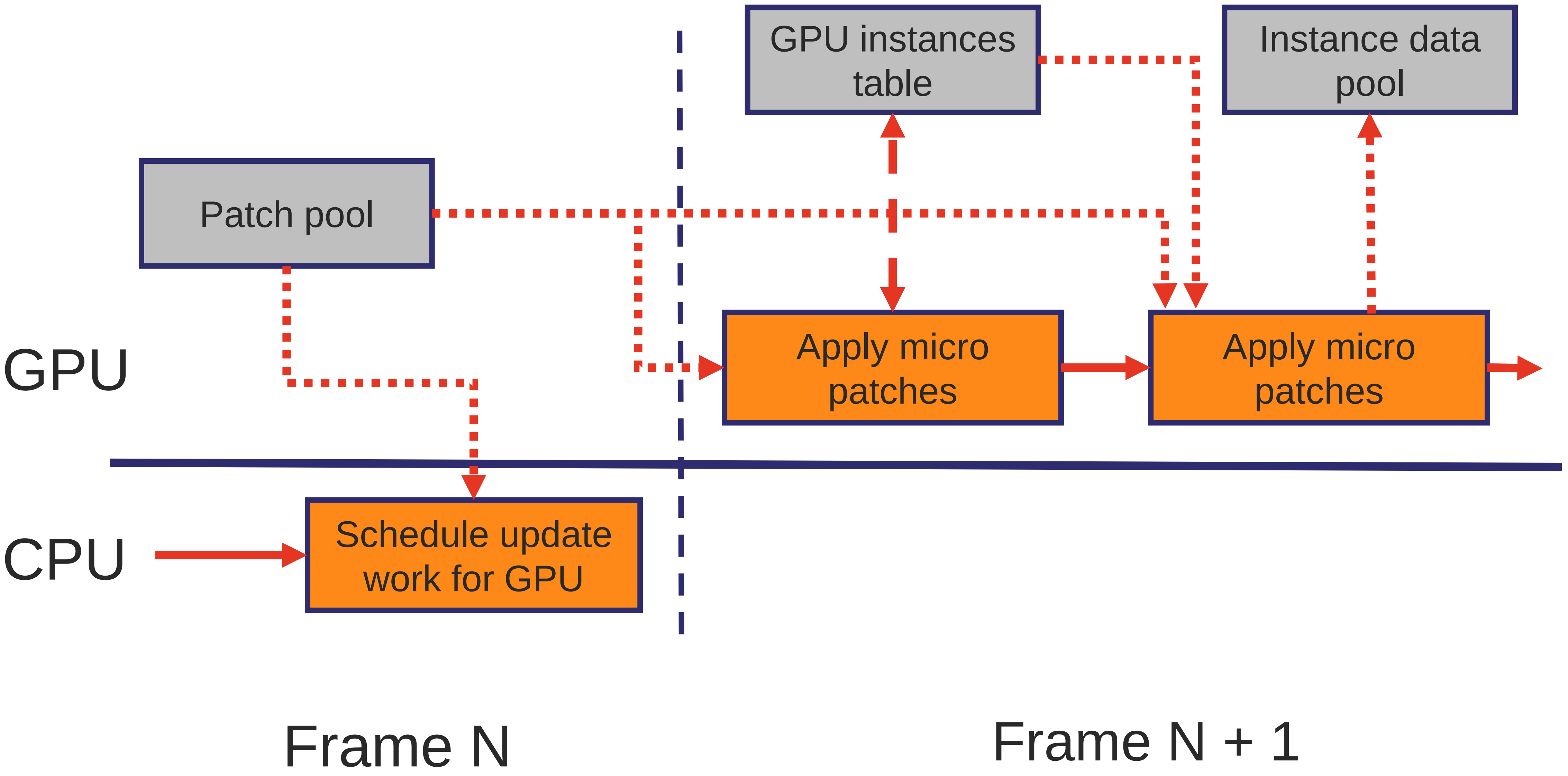


- Write a simple list with buffer offsets and the data to be written
- Patch application order matters

GPU Instance Table Patching



Pools Patching



Micro Patching Results

- Deferred synchronization
- Exactly one explicit sync point for all GPU structures
- No contention with GPU to access/modify buffers
- Patches data gathering improves CPU performance significantly
- Patch pool data scattering hits GPU performance
- Substantially less instances in “dirty” state

Micro Patching Results

- Deferred synchronization
- Exactly one explicit sync point for all GPU structures
- No contention with GPU to access/modify buffers
- Patches data gathering improves CPU performance significantly
- Patch pool data scattering hits GPU performance a bit
- Substantially less instances in “dirty” state

	Memory	CPU	GPU
From	1.21 Mb	7.18 ms	17.95 ms
To	1.01 Mb	5.89 ms	17.41 ms

(*Xbox One numbers)

Variable Synchronization Rate

- Use variable animation rate idea
- Synchronize instances state based on “importance” factor
- Keep it deterministic (for replays, debugging, etc.)
- “Importance” computed automatically whenever possible
- Overall synchronization problem looks pretty similar to network synchronization

Simulated instance		Rendered instance
Transform data	60Hz →	Transform data
Skinned data		Skinned data

Instance Importance

Main character simulated instances Transform data Skinned data	60Hz →	**Main character rendered instances** Transform data Skinned data
Physically simulated decoration instance Transform data Skinned data	30Hz →	**Rendered decoration instance** Transform data Skinned data
Distant simulated instances Transform data Skinned data	15Hz →	**Distant rendered instances** Transform data Skinned data

High Synchronization Rate

Low Synchronization Rate

Variable Synchronization Rate Results

- Tricky to define “importance”
  - Use heuristics and dynamic “importance” adjustment
- Hard to spot quality difference but it’s possible
  - Send data for the “next” simulated state (interpolate on GPU)
- Complexity with current and history parameters
  - Make “frame index” associated with instance
- Worst case simulation cost still high
  - Move as much as possible to GPU

Instance State Generation on GPU

- Bones, animations, and skeletons data moved to GPU
  - Apply developed synchronization tech to keep this data up to date
- Generate skinning data right next to the place of usage
- Significant CPU offload – especially for big crowds
- Hierarchical data structures with interdependencies
  - Pretty low occupancy on pointers chasing – overlap as much as possible
  - Start next frame simulation right after the last geometry pass
- Manageable GPU hit

Animation on GPU Results

Decouple CPU and GPU for mobile, mutual, and skinned instances to minimize data transfer and synchronization points

Type	Relative	Count	Memory	CPU
Mobile instances	70.11%	3160	~0.51 Mb	4.96 ms
Mutual instances	10.43%	470	~0.08 Mb	0.33 ms
Skinned instances	19.46%	877	~0.18 Mb	0.24 ms

	Memory	Offline CPU*	Run-time CPU*	GPU
From	1.01 Mb	14.96 ms	5.89 ms	17.95 ms
To	0.77 Mb	13.49 ms	5.53 ms	18.06 ms

Next steps:

- Revise data layout for animation structures
- Skin geometry only once

(*Xbox One numbers)

Instance State Generation on GPU

- Use Bullet Physics with GPU simulation support
- Split mobile instances further
  - Introduce CPU invisible/visible physical instances
- Explicitly tag CPU invisible instances
  - Code driven solution – not easy to find tagging approach
- Sync CPU visible instances and simulate the rest
- GPU simulation faster than CPU most of the time

Rigid Body Physics on GPU Results

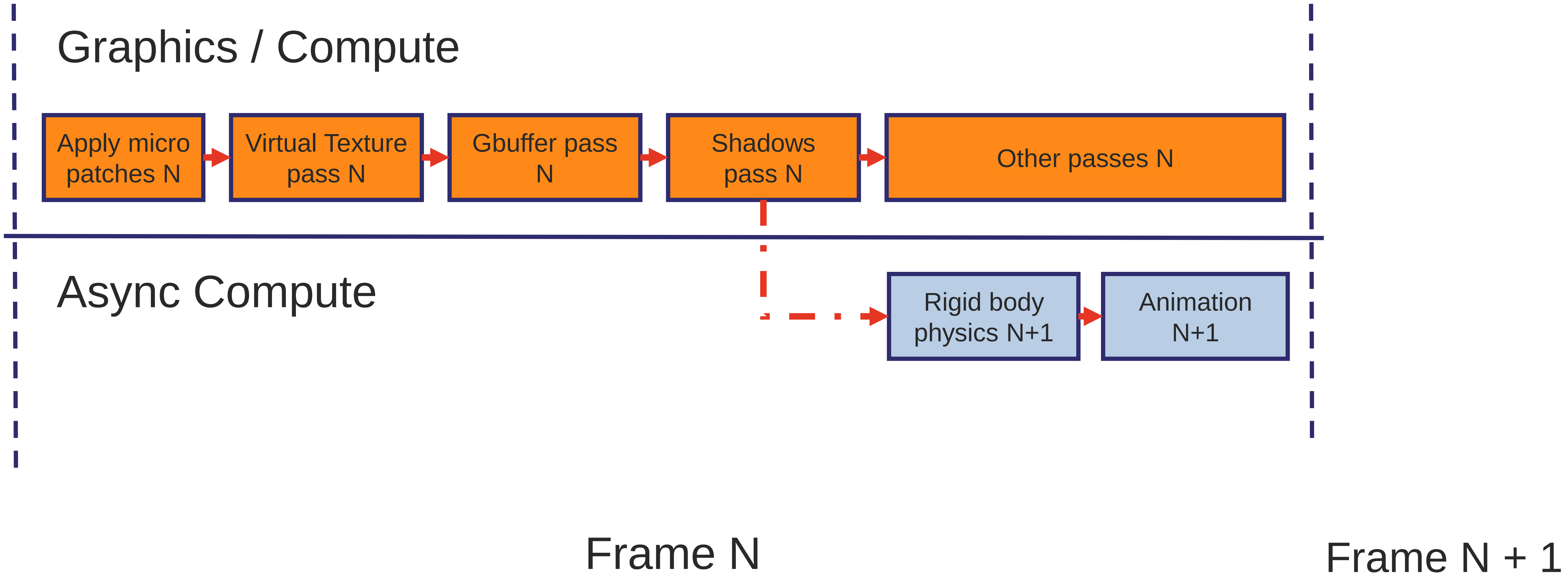
Type	Relative	Count	Memory	CPU
CPU visible mobile instances	53.33%	1539	~0.26 Mb	4.54 ms
Mutual instances	16.29%	470	~0.08 Mb	0.31 ms
Skinned instances	30.38%	877	~0.18 Mb	0.21 ms

	Memory	Offline CPU	Run-time CPU	GPU
From	0.77 Mb	13.49 ms	5.53 ms	18.06 ms
To	0.52 Mb	12.17 ms	5.06 ms	18.25 ms

- Significant CPU offload
- Manageable GPU hit - perfectly overlaps with PP stack
- High code complexity

(*Xbox One numbers)

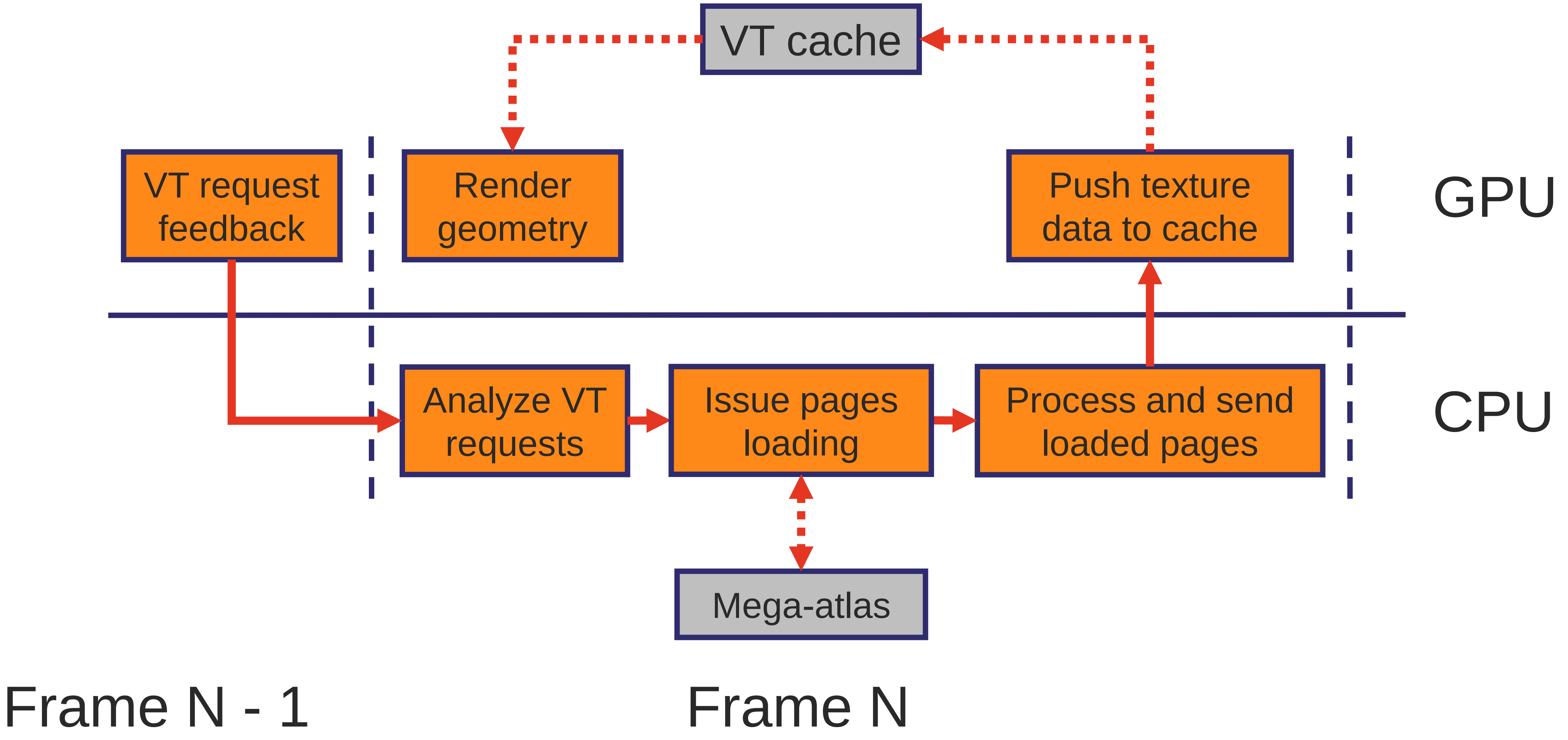
Geometry Rendering Passes Overview



	Memory	CPU	GPU
Base line	1.05 Mb	51.43 ms	22.17 ms
To result	0.57 Mb	16.94 ms	17.02 ms

(*Xbox One numbers)

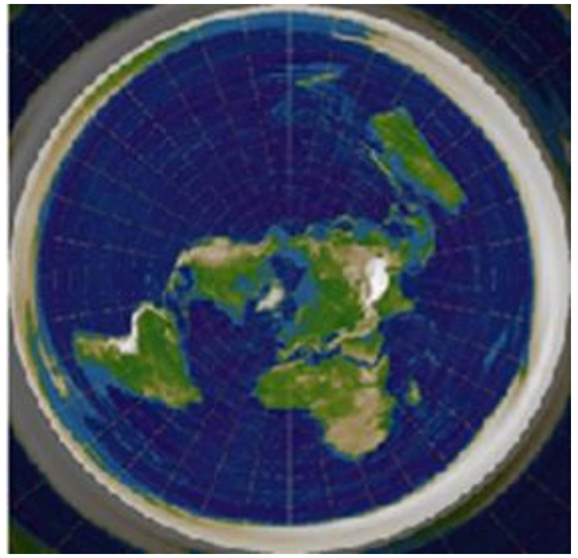
Virtual Texturing



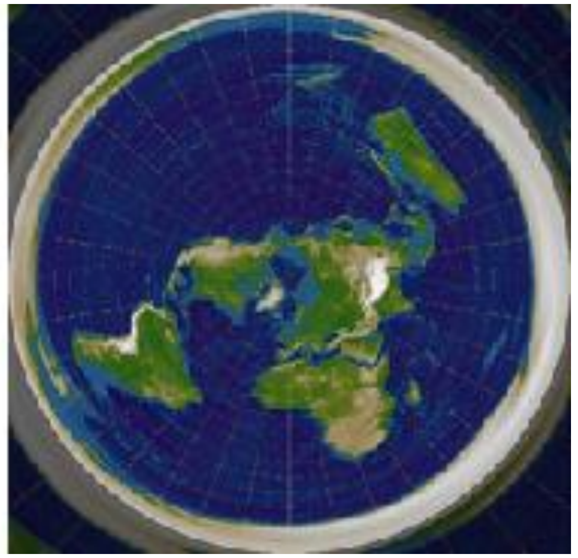
Virtual Texturing

- Access to all the texture data at once
- Generally lower memory usage and data transfer pressure than common texture streaming techs
- Drastically improves “batching” efficiency of the GPU Driven Rendering
- Replacement for the bindless textures available for some platforms/APIs

Virtual Texturing

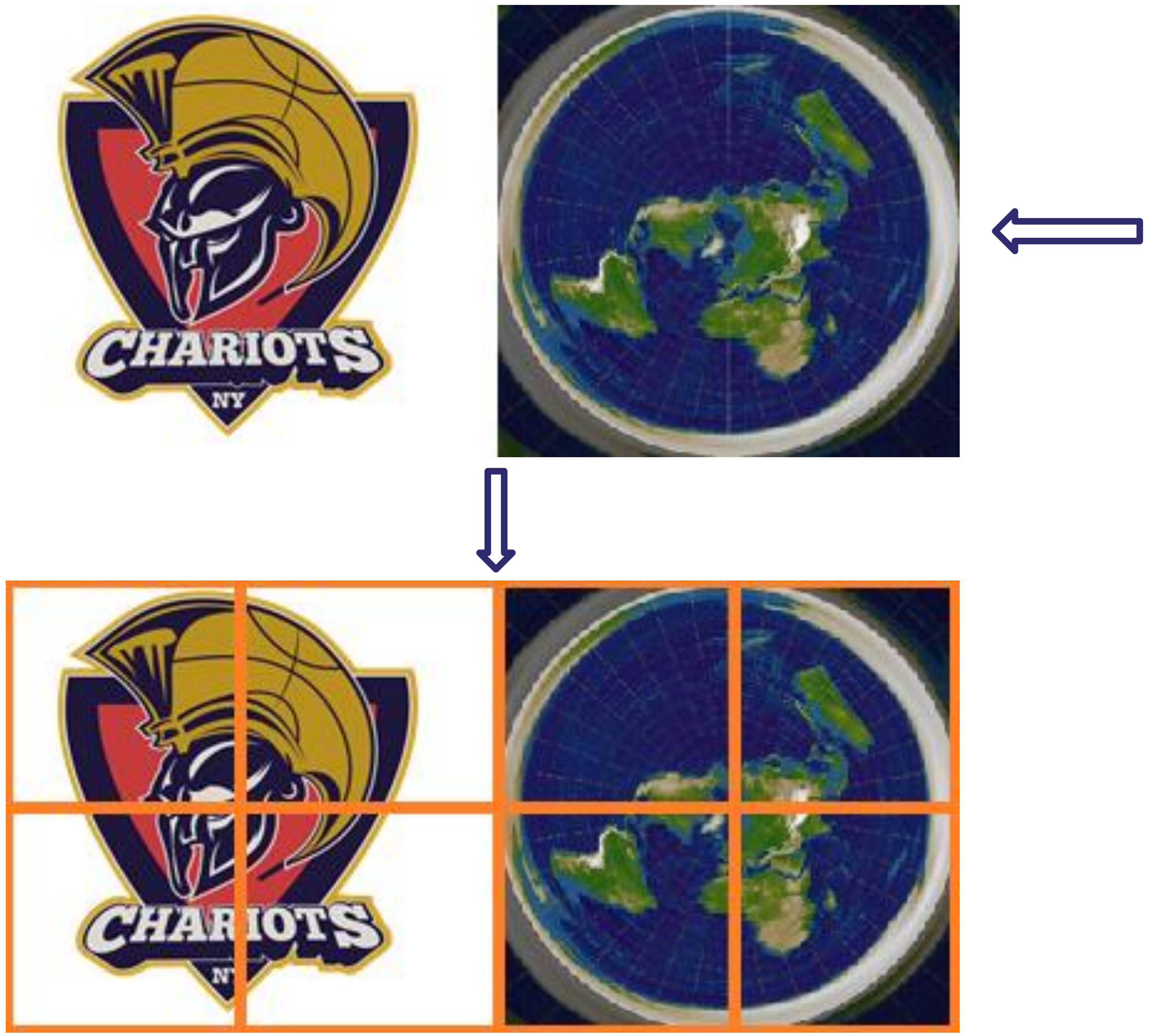


Virtual Texturing



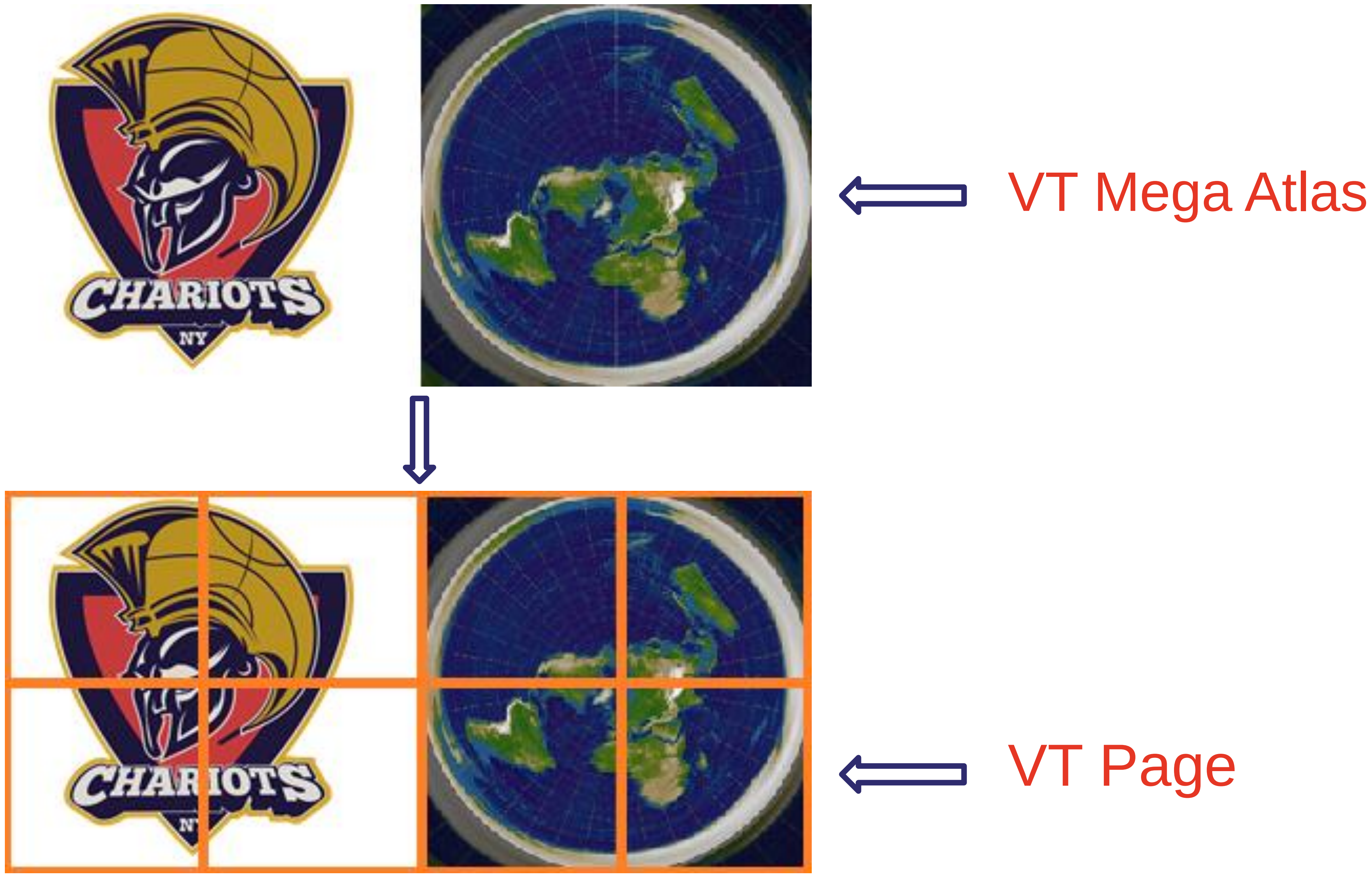
VT Mega Atlas

Virtual Texturing

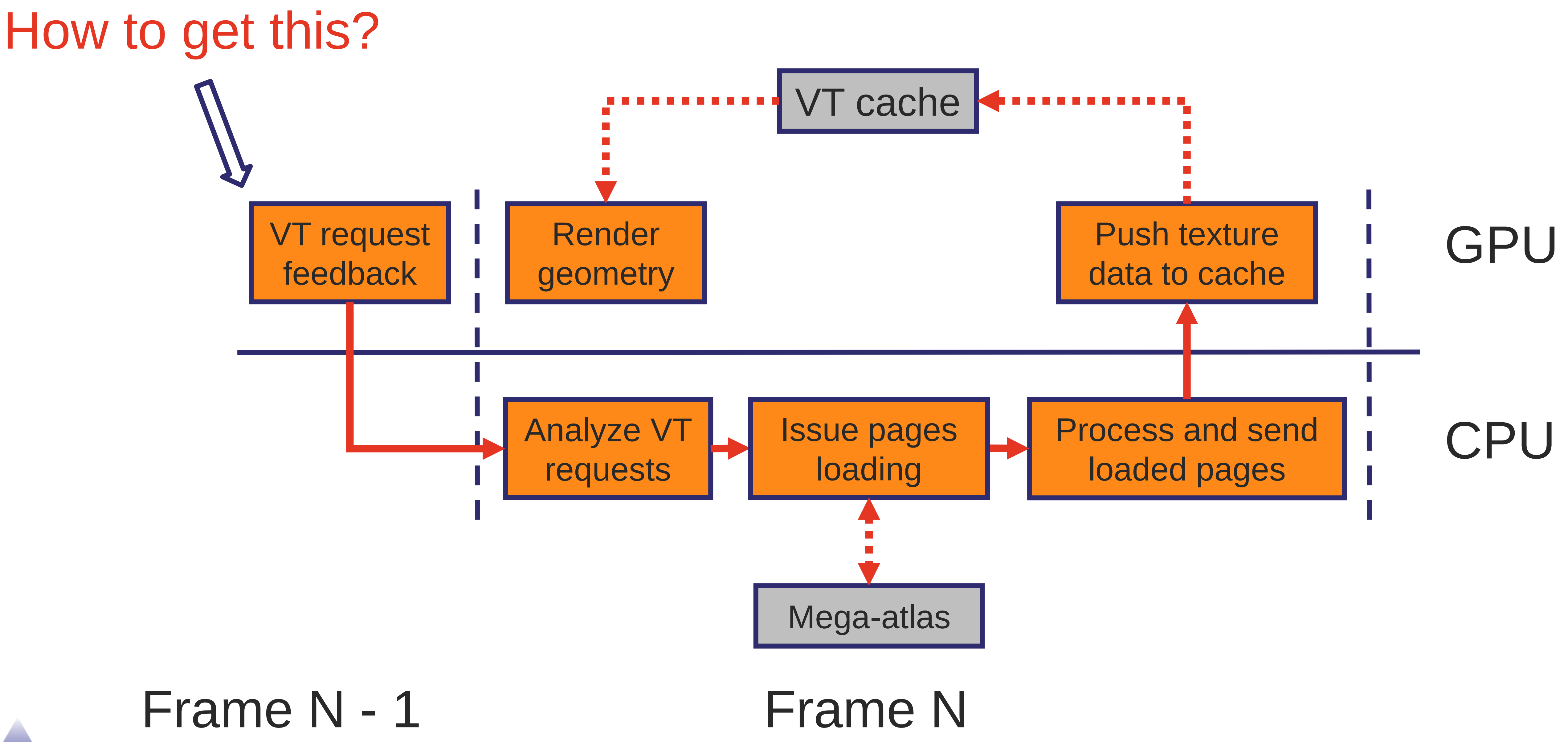


VT Mega Atlas

Virtual Texturing



Virtual Texturing

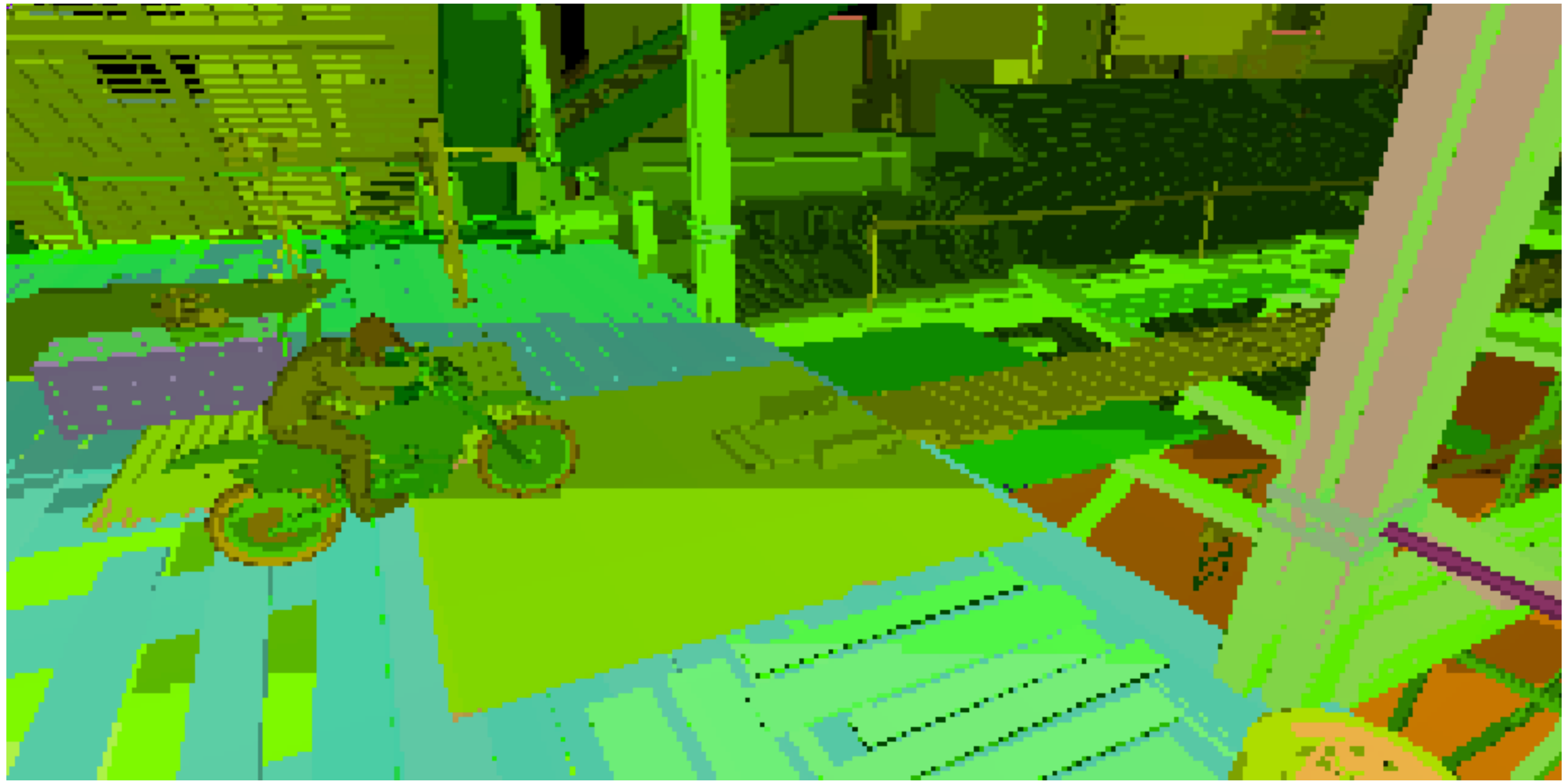


Virtual Texturing – Dedicated Pass

- Use special camera for the VT Page Request pass
  - Streaming prediction, textures preloading, etc.
  - Requires extra viewport culling
- Rasterize geometry twice to generate Page Request RT
  - Smaller render target (1/4 + jitter)
  - Store page requests (x coordinate, y coordinate, mip)
- Analyze on CPU and generate loading requests

Virtual Texturing – Dedicated Pass

Lighting Buffer

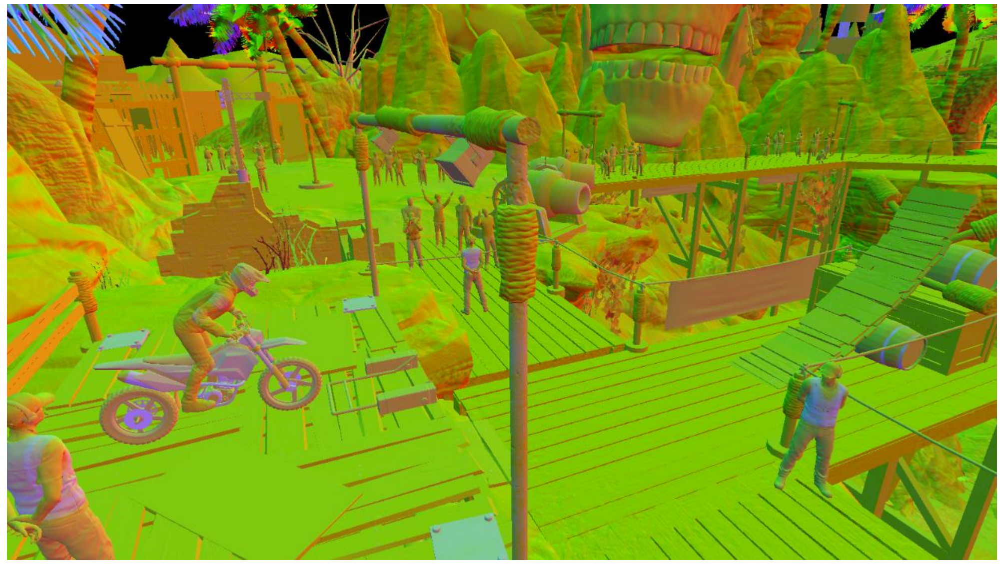


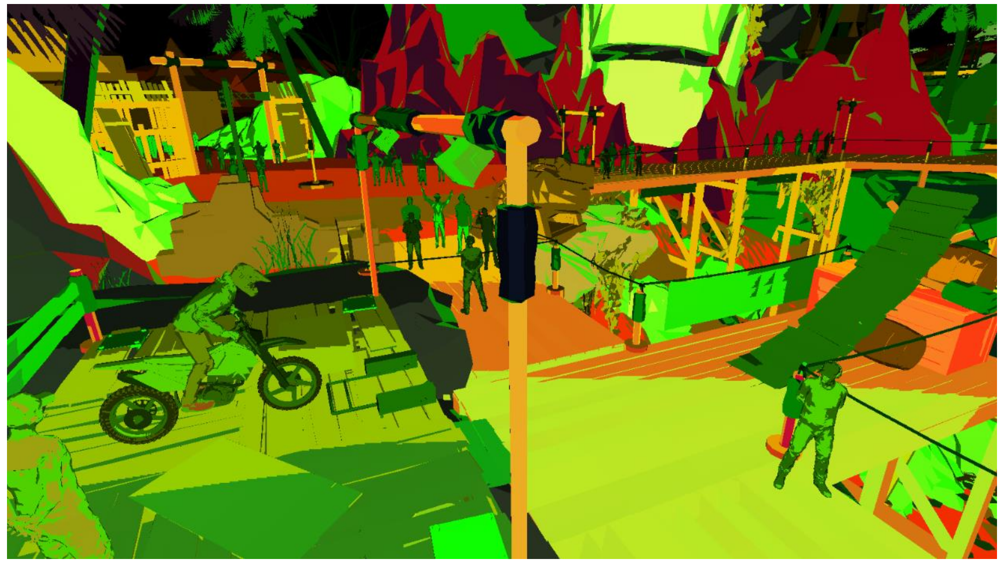
Page Request Buffer

Virtual Texturing – Part of GBuffer

- Use fat GBuffer layout
  - Add extra 32-bits to GBuffer
  - Extra memory bandwidth
- One visible geometry rasterization pass
  - Less shaders, same code-path for page request and page usage
- Compute pass to downscale / filter page request buffer

Virtual Texturing – Part of GBuffer





Page Request Buffer

Virtual Texturing – “In-place PR”

- Replace render target with two UAV buffers
  - Bloom filter buffer
  - Page requests buffer
- Move PR filtering, sorting, etc. to GPU and delegate it to each pixel in GBuffer
  - Allows to shave some time from CPU
- Page requests for transparent passes, alpha blended passes, etc.

Virtual Texturing – “In-place PR”

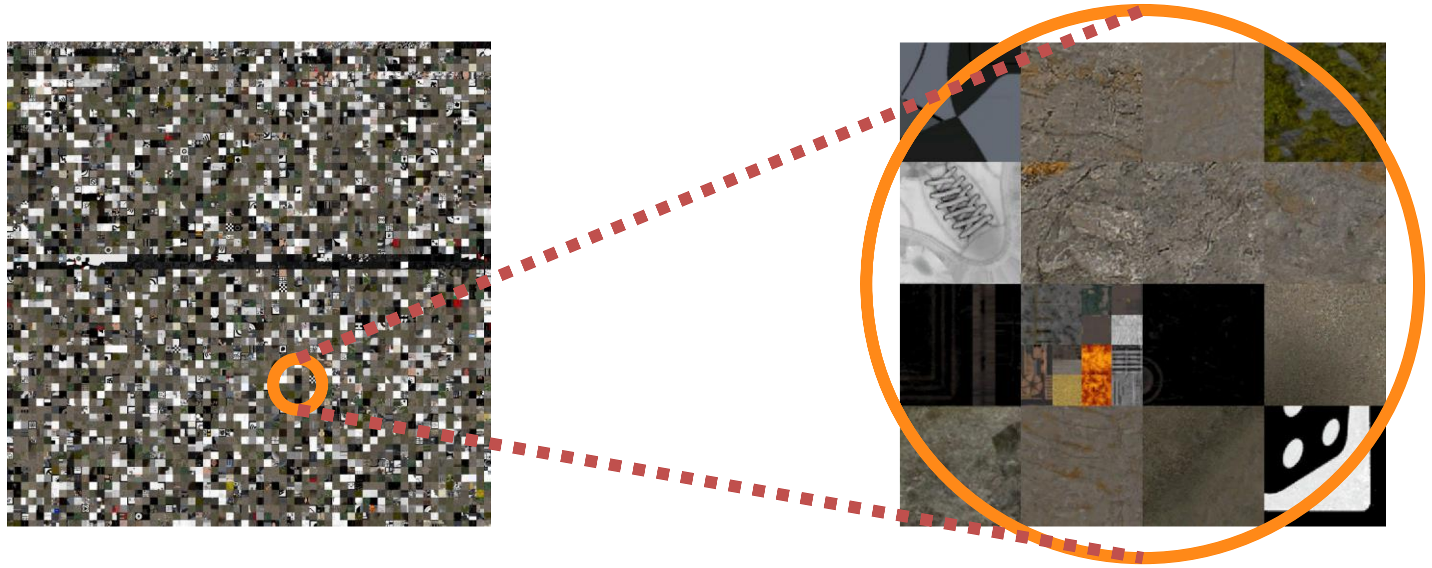
```
uint triplet = encodePageRequestTriplet(pageRequest);
uint tripletHash = fnv1AHash32(triplet, g_vtBloomFilterSeed);

uint bloomBitIndex = tripletHash % g_vtBloomFilterSize;
uint bloomWordIndex = bloomBitIndex / 32;
uint bloomWordMask = 1 << (bloomBitIndex % 32);

if (!(VTBloomFilter[bloomWordIndex] & bloomWordMask)) {
        uint previous = 0;
        InterlockedOr(VTBloomFilter[bloomWordIndex], bloomWordMask, previous);

        if (!(previous & bloomWordMask)) {
                uint index = 0;
                InterlockedAdd(VTFilteredPages[0], 1, index);
                VTFilteredPages[index + 1] = triplet;
        }
}
```

Virtual Texturing

- Trials Rising ended up using “In-place PR”
- This change allowed to shave about 3-5 ms of CPU time from every odd frame (Xbox One timings)
- Can cause extra texture loading latency that must be taken into account
- Semi-transparencies can be placed in Mega-Texture
- Great shaders code simplification

Virtual Texture Scalability

- Page requests per frame dictates ”performance” of the system
  - Highly depends on rendering resolution and VT pages cache size
  - Both parameters can be controlled in “online” and “offline” settings



Virtual Texture Mip Bias

VT mip bias can be controlled by CPU and GPU

```
float computeMipBias()
{
#if defined(USE_GLOBAL_MIP_BIAS_ONLY)
		// Directly fetch global mip bias
		float mipBias = g_mipBias;
#else
		// Combine global mip bias and per-instance mip bias
		float mipBias = g_mipBias + gp_mipBias;
#endif
		// Restrict mip bias
		mipBias = max(mipBias, g_mipBiasMinimum);

		return mipBias;
}
```

Virtual Texture Mip Bias

VT mip bias can be controlled on CPU and GPU

```
float mipMapLevel(float2 coordinates, float2 size, float bias)
{
        // Hacky anisotropic emulation
        const float anisotropicSize = size.x * 0.7071;

        const float2 dx = ddx(coordinates * anisotropicSize);
        const float2 dy = ddy(coordinates * anisotropicSize);

        const float d = dot(dx, dx) + dot(dy, dy);

        return 0.5 * log2(d) + bias;
}
```

Virtual Texture Page Cache Size

- Static configuration per platform
  - Can be of arbitrary size and even changed dynamically

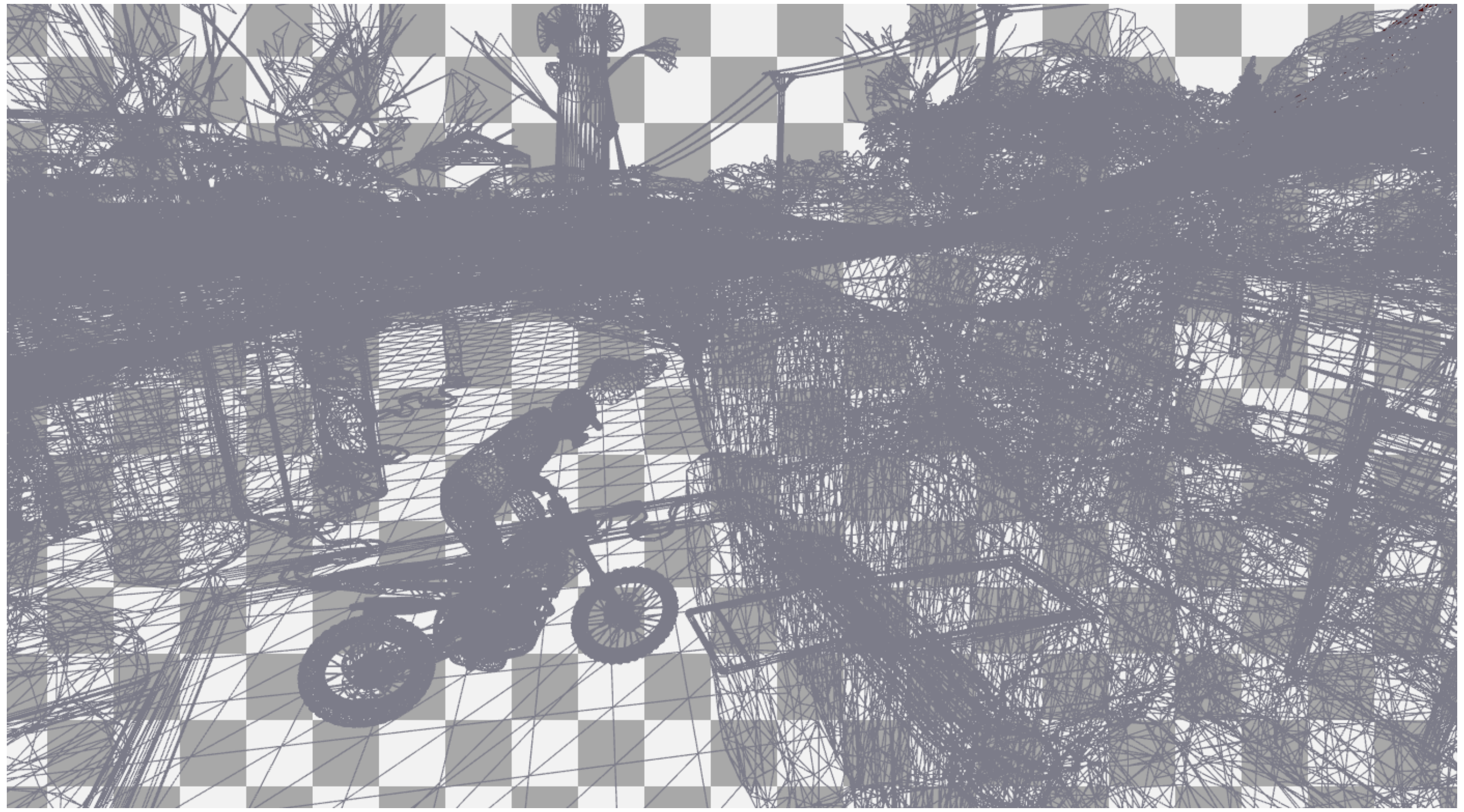
Platform	Page Cache Size	Memory Consumption
Xbox One Base	8k x 8k	3 x 64 Mb = 192 Mb
Xbox One S	8k x 8k	3 x 64 Mb = 192Mb
Xbox One X	16k x 8k	3 x 128 Mb = 384 Mb
PS4 Base	8k x 8k	3 x 64 Mb = 192 Mb
PS4 Pro	16k x 8k	3 x 128 Mb = 384 Mb
Switch	8k x 4k	3 x 32 Mb = 96 Mb

GPU Driven Rendering Scalability

- Computation part of the GPU pipeline very fast
  - Culling and geometry assemble scales with GPU clock and memory bandwidth/latency
  - Rasterization is based on resolution and scales pretty well
- GPU pipeline requires specific data path for CPU-GPU
  - Memory bandwidth hit from instances synchronization
  - Batches with small amount of instances overhead is higher than the actual work

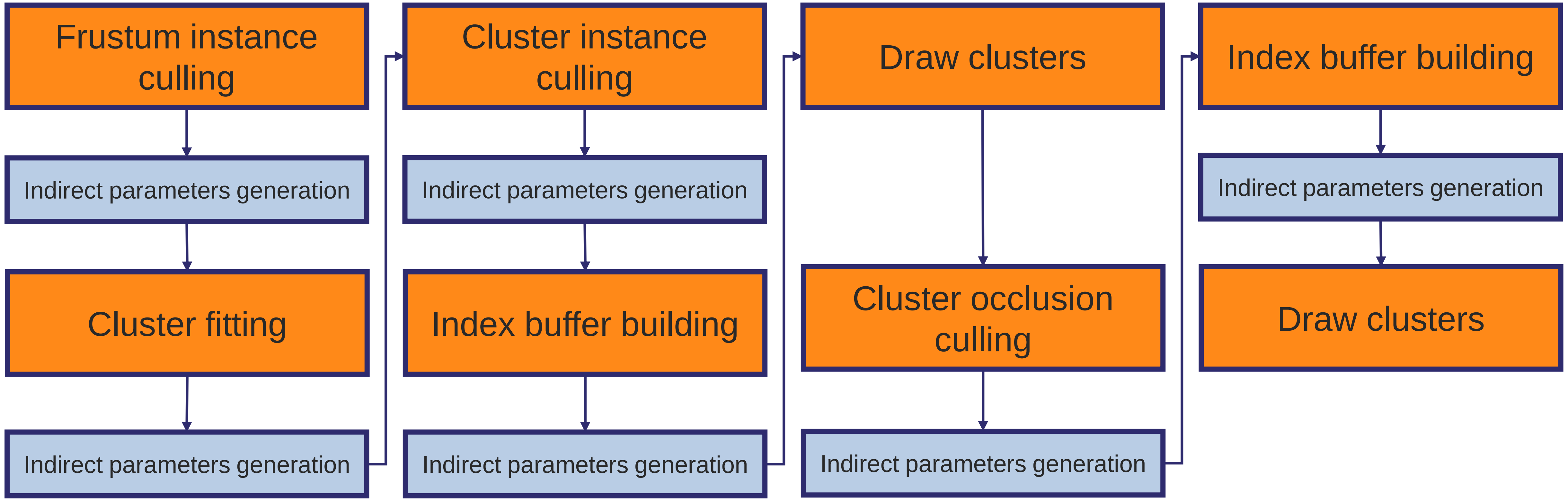
GPU Driven Rendering Scalability

- Use more aggressive culling settings for distant objects for low end platforms
- Adjust level of details in runtime depending on CPU/GPU load



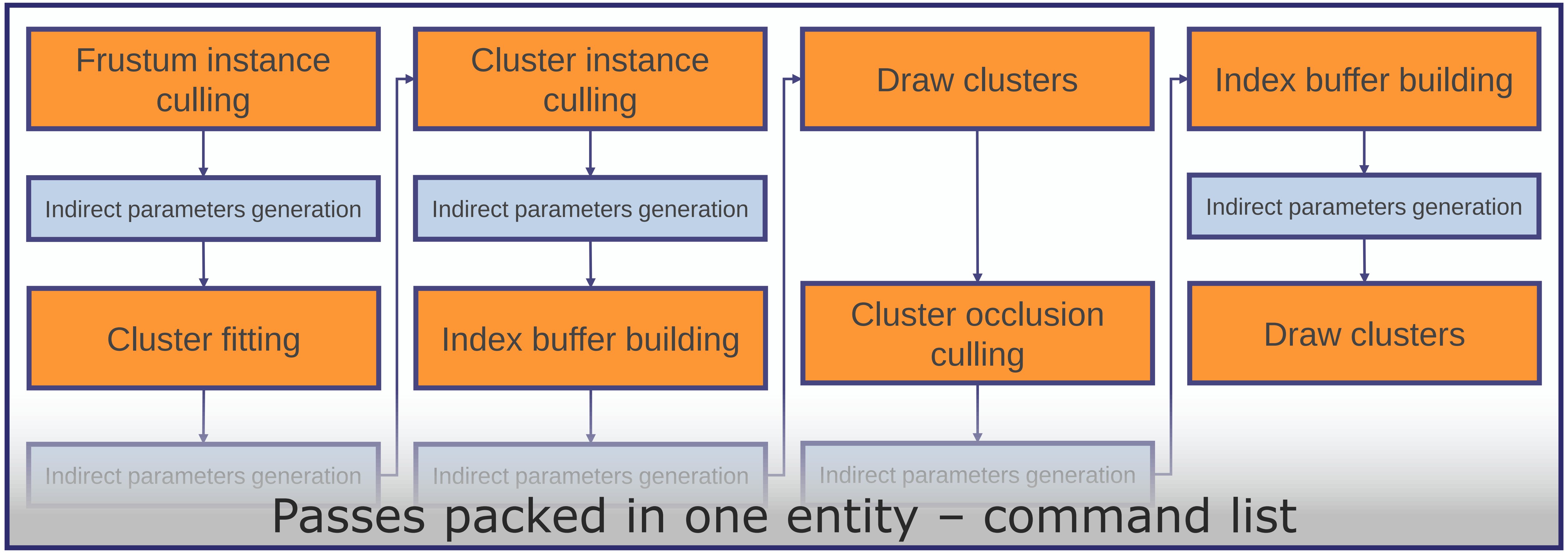
Future - High-level Batch Rendering

- Many indirect parameters generation passes
- This pipeline repeated for each batch



Future - High-level Batch Rendering

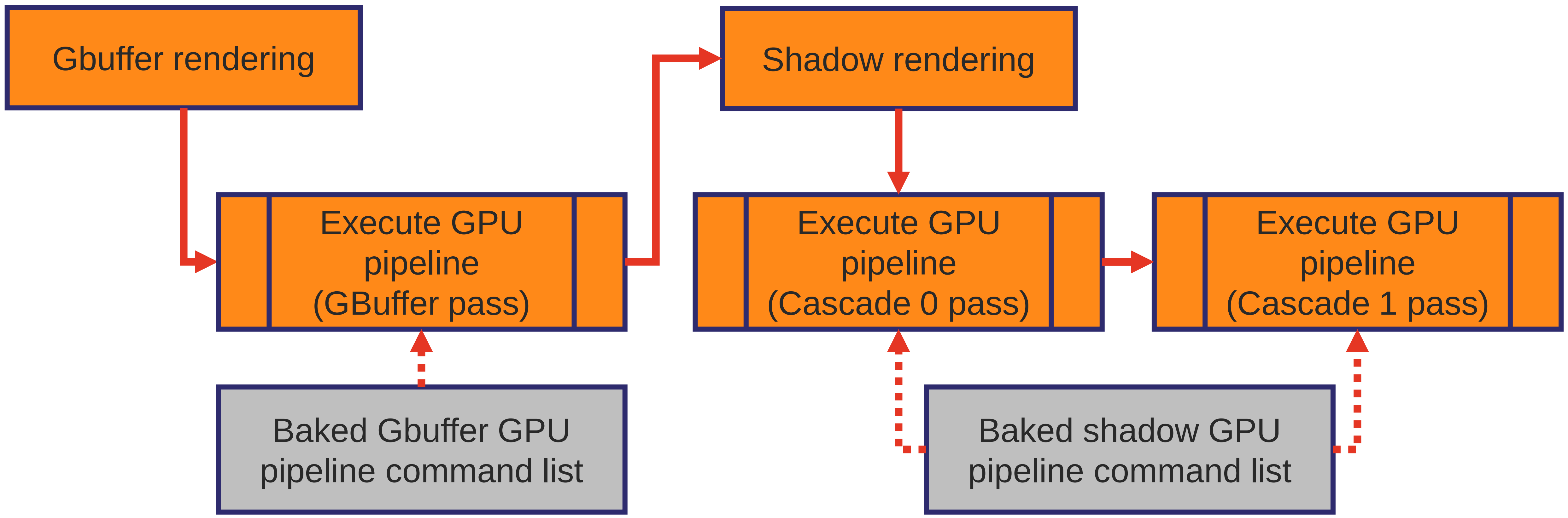
Idea is to bake passes in command list and use it for each batch



Future - Command List to Render a Scene

- GPU pipeline commands sequence almost constant every frame
- Commands do not change based on scene structure
- Draw calls count predictable and reasonably low
- Manageable maximum index buffer size
- Worst measured memory overhead is about 6%
- Record GPU pipeline commands for all possible butches in nested command list

Execute GPU Pipeline Command List



Baking Results

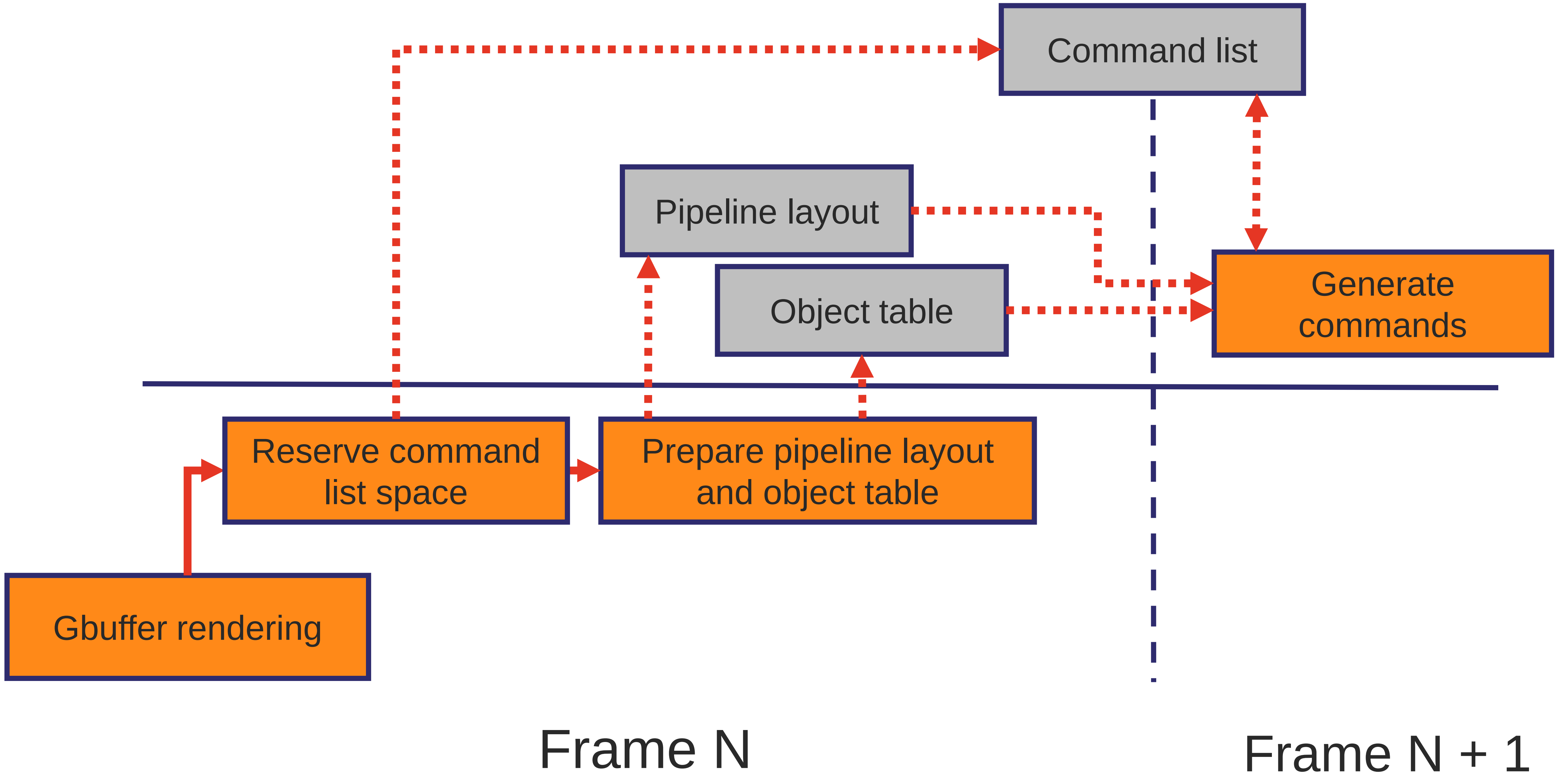
Test scene	Memory	GPU	CPU
Construction Site Scene	+2.14%	-0.05 ms	-0.21 ms
Scene with small differences by instances types	+0.42%	-0.08 ms	-0.13 ms
Scene with high differences by instances types	+5.13%	+0.02 ms	-0.34 ms
First scene with medium differences by instances types	+1.46%	-0.06 ms	-0.19 ms
Second scene with medium differences by instances types	+3.58%	+0.01 ms	-0.26 ms

(*Xbox One numbers)

Future – Device Generated Commands

- NVidia extension for Vulkan to populate command list on GPU
  - https://developer.nvidia.com/device-generated-commands-vulkan
- Possibility to compose GPU pipeline commands conditionally
- More flexibility with PSO selection on GPU
- Possibility to switch PSO with compile time optimization based on runtime condition
- Possibility to avoid empty draw chains
- Decreased reserved memory size

Generate GPU Pipeline Command List on GPU

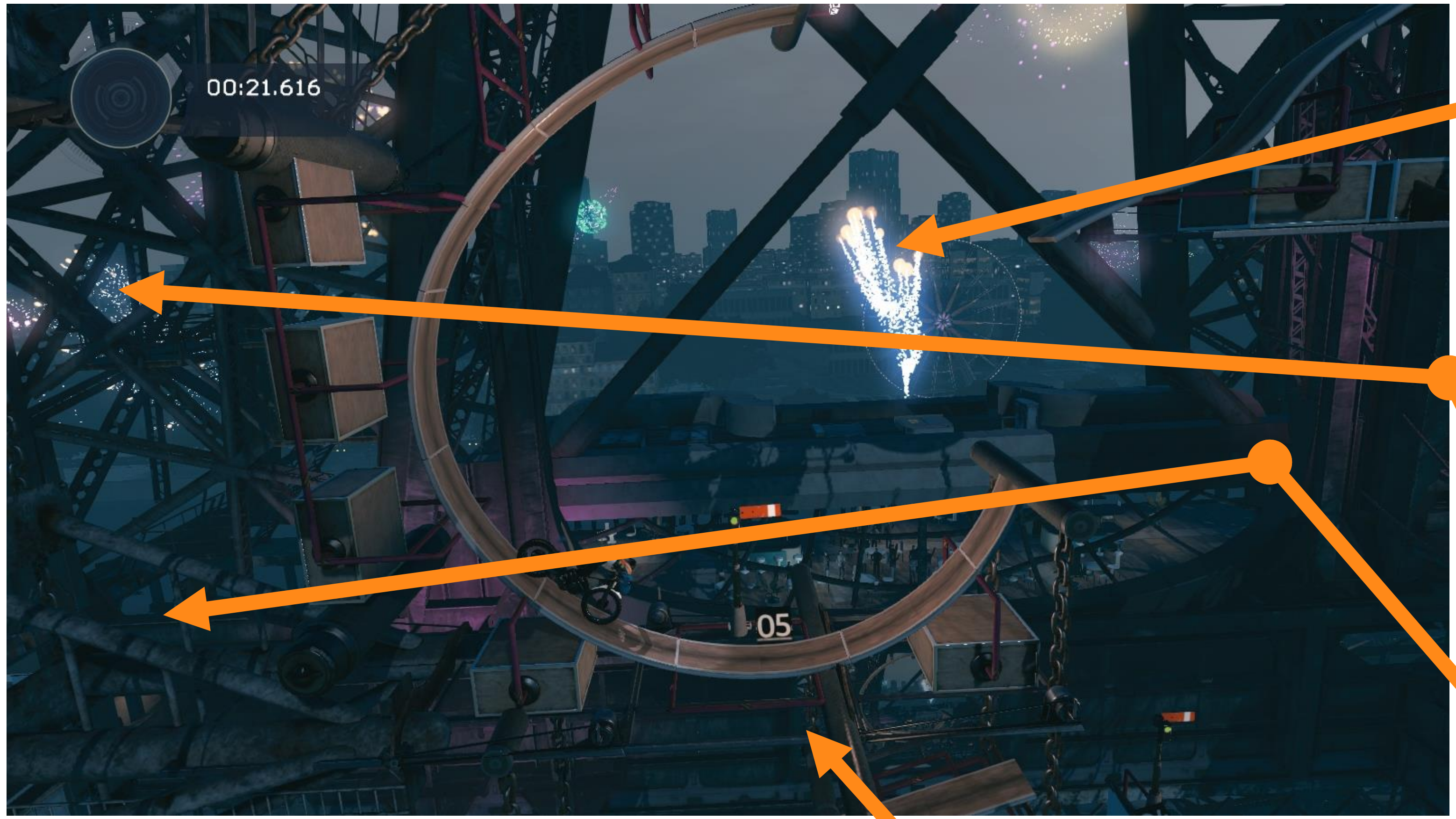


GPU Side Generation Results

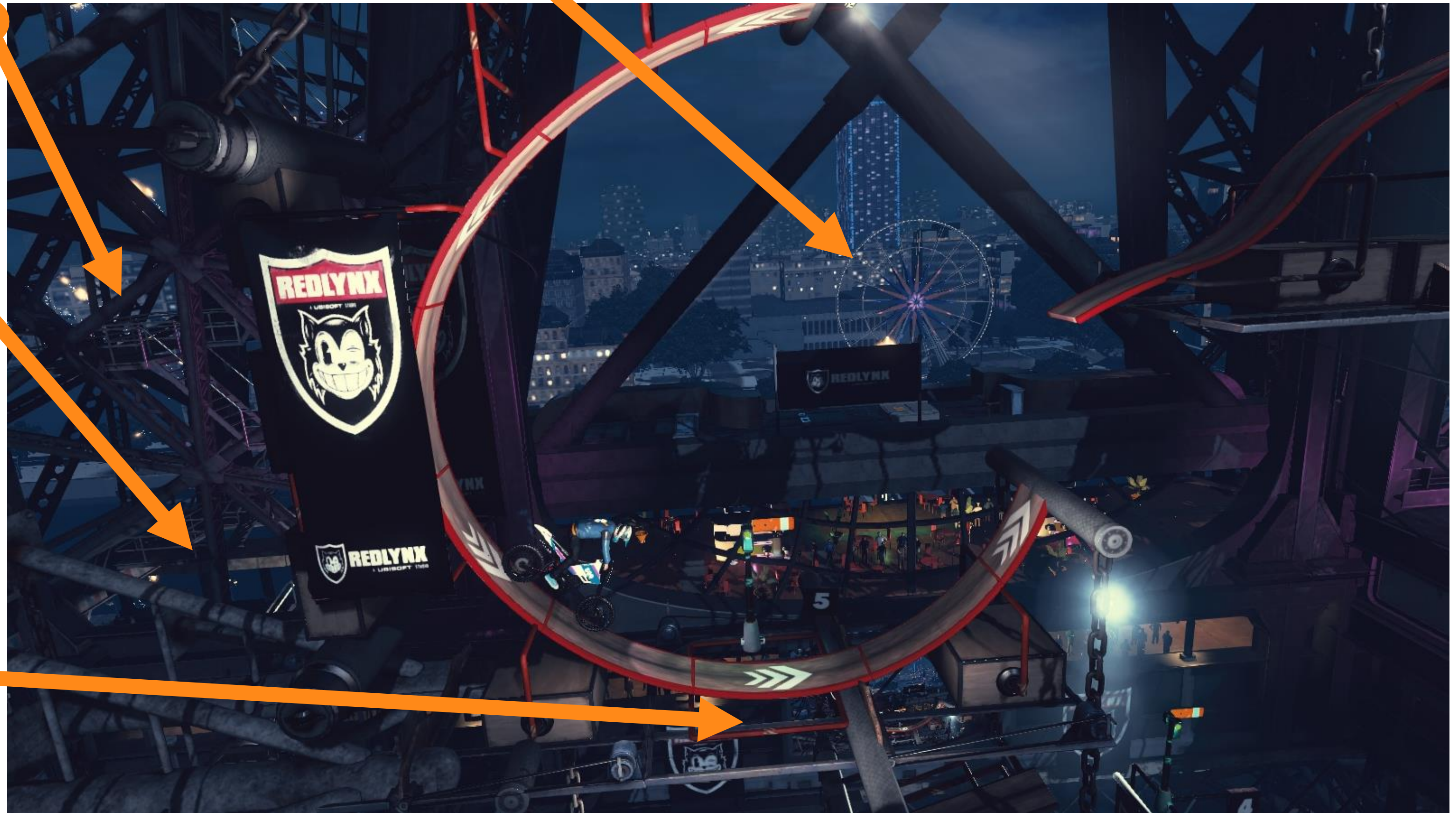
Test scene	Memory	GPU	CPU
Construction Site Scene	+0.76%	-0.03 ms	-0.32 ms
Scene with small differences by instances types	+0.11%	-0.05 ms	-0.18 ms
Scene with high differences by instances types	+1.22%	+0.03 ms	-0.39 ms
First scene with medium differences by instances types	+0.45%	-0.07 ms	-0.21 ms
Second scene with medium differences by instances types	+1.08%	+0.02 ms	-0.33 ms

(*Xbox One numbers)

Visual Fidelity Comparison



Visual quality unlocked by Virtual Texturing and GPU Driven Rendering techs



Visual Fidelity Comparison

Visual quality unlocked by Virtual Texturing and GPU Driven Rendering techs

Conclusions

- GPU Driven Rendering and Virtual Texturing work great together
- It’s easy to implement GPU Pipeline but hard to make it fast with a specific engine
- Debugging utilities are must have
- Virtual Texturing might be painful for the art pipeline
- GPU Pipeline surprisingly aligned with Ray Tracing

Special Thanks

- Code
  - Jussi Knuuttila, Anton Remezenko, Milos Tosic
- Presentation
  - Christina Coffin, Stephen McAuley
- Oleksandr Shvyrlo, Mickael Godard, Michael Ockenden
- Entire Trials Rising team


ARE YOU READY TO
CREATE THE UNKNOWN?
jobs.ubisoft.com
LEARN MORE AT OUR BOOTH!
WEST HALL - FLOOR 2
UBISOFT

Thank you! Дякую!
Questions?

oleksandr.drazhevskyi@ubisoft.com

References

- “GPU-Driven Rendering Pipelines”, Haar & Aaltonen, SIGGRAPH 2015
- “Optimizing the Graphics Pipeline with Compute”, Wihlidal, GDC 2016
- “Deferred+: Next-Gen Culling and Rendering for Dawn Engine”, Bucci & Doghramachi, Eidos Montreal
- “GPU-Based Scene Management for Rendering Large Crowds”, Barczak at al., AMD

References

- “March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU”, Shopf at al., SIGGRAPH 2008
- “Large-Scale Physics-Based World Simulation Using Adaptive Tiles on the GPU”, Vanek, IEEE Computer Graphics and Applications 2011
- “GCN Shader Extensions for Direct3D and Vulkan”, Matthaeus Chajdas, GPUOpen.com, 2016