



FrameGraph: Extensible Rendering Architecture in Frostbite

Yuriy O'Donnell
Rendering Engineer
Frostbite

GAME DEVELOPERS CONFERENCE® | FEB 27–MAR 3, 2017 | EXPO: MAR 1–3, 2017 #GDC17

UBM



Outline

- ▶ Introduction and history
- ▶ Frame Graph
- ▶ Transient Resource System
- ▶ Conclusions

Spoilers:

- ✓ Improved engine extensibility
- ✓ Simplified async compute
- ✓ Automated ESRAM aliasing
- ✓ Saved **tons** of GPU memory

Introduction

FROSTBITE EVOLUTION OVER THE LAST DECADE

Frostbite 2007 vs 2017

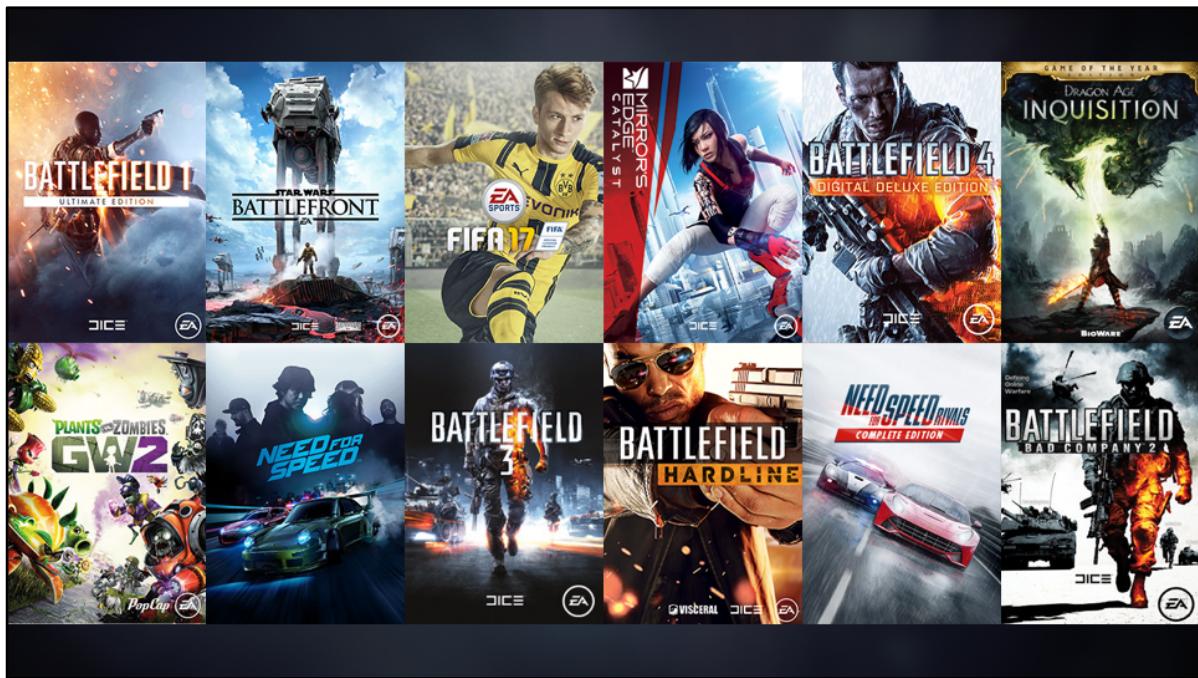
2007

- ▶ DICE next-gen engine
- ▶ Built from the ground up for
 - ▶ Xbox 360
 - ▶ PlayStation 3
 - ▶ Multi-core PCs
 - ▶ DirectX 9 SM3 & Direct3D 10
- ▶ To be used in future DICE games

2017

- ▶ The EA engine
- ▶ Evolved and scaled up for
 - ▶ Xbox One
 - ▶ PlayStation 4
 - ▶ Multi-core PCs
 - ▶ DirectX 12
- ▶ Used in ~15 current and future EA games

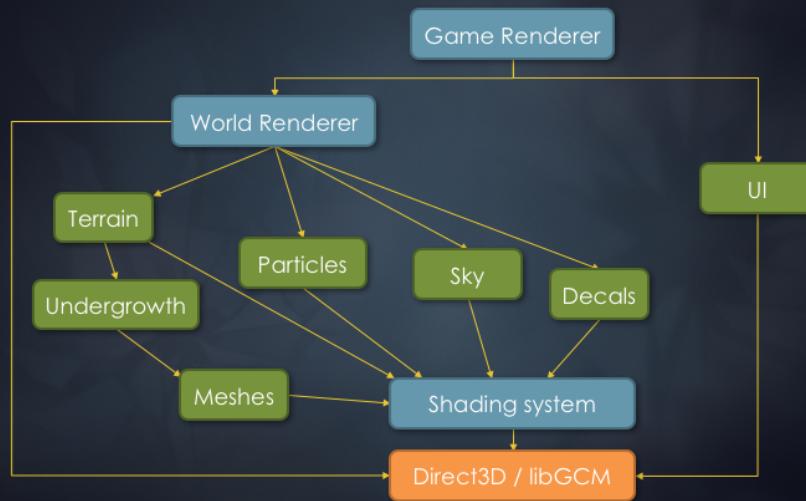
<http://www.frostbite.com/2007/04/frostbite-rendering-architecture-and-real-time-procedural-shading-texturing-techniques>



More diverse games

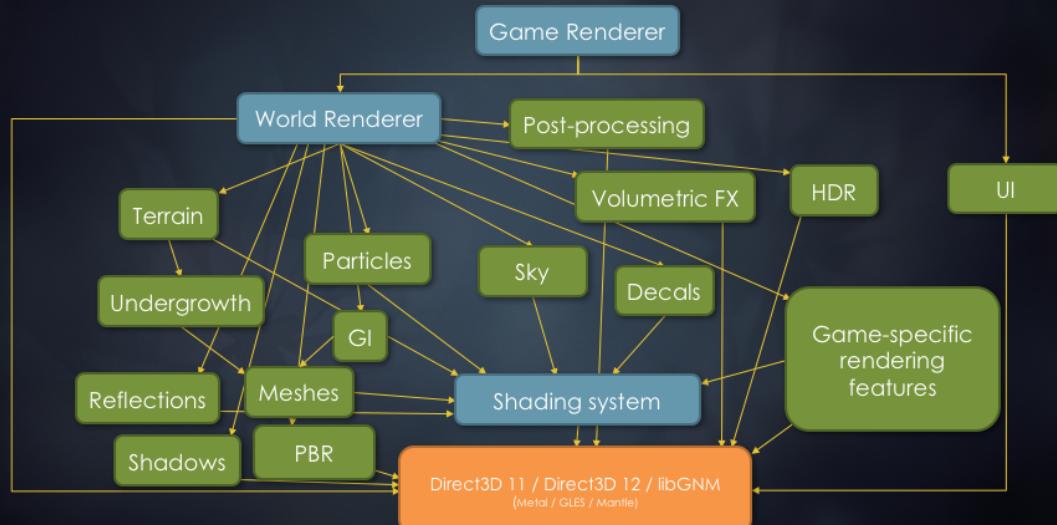
Not just Battlefield engine
RPG, Racing, Sports, Action

Rendering system overview '07



<http://www.frostbite.com/2007/04/frostbite-rendering-architecture-and-real-time-procedural-shading-texturing-techniques> slide 17

Rendering system overview '17



Not meant to be a complete / representative graph, just illustrating the scaling challenges.

Basically the same, except larger number of systems with more complicated coupling.

Mostly unchanged since 2007

Until recently!

Everything scaled up

More features

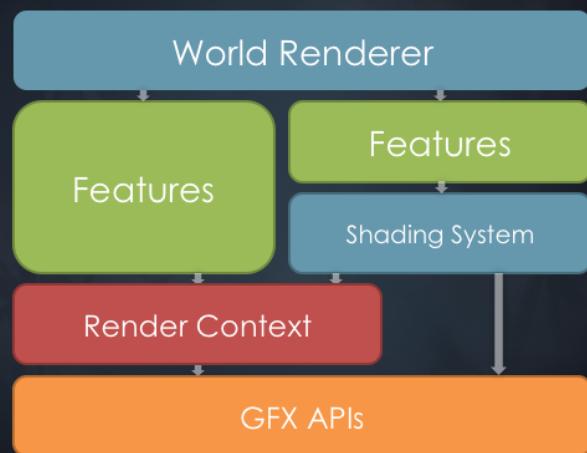
Much larger community

Scaling and maintenance challenges

Shading system described in detail by Johan in 2007.

Rest of this talk will be about World Renderer and rendering features.

Rendering system overview (simplified)



WorldRenderer

- ▶ Orchestrates all rendering
 - ▶ Code-driven architecture
 - ▶ Main world geometry (via Shading System)
 - ▶ Lighting, Post-processing (via Render Context)
 - ▶ Knows about all views and render passes
- ▶ Marshalls settings and resources between systems
- ▶ Allocates resources (render targets, buffers)



World Renderer architecture is the focus of the presentation from this point.

Battlefield 4 rendering passes (Features)

- ▶ reflectionCapture
- ▶ planarReflections
- ▶ dynamicEnvmap
- ▶ mainZPass
- ▶ mainGBuffer
- ▶ mainGBufferSimple
- ▶ mainGBufferDecal
- ▶ decalVolumes
- ▶ mainGBufferFixup
- ▶ msaaZDown
- ▶ msaaClassify
- ▶ lensFlareOcclusionQueries
- ▶ lightPassBegin
- ▶ cascadedShadowmaps
- ▶ spotlightShadowmaps
- ▶ downsampleZ
- ▶ linearizeZ
- ▶ ssao
- ▶ hbaohalfZ
- ▶ hbaohalfZ
- ▶ ssr
- ▶ halfResZPass
- ▶ halfResTransp
- ▶ mainDistort
- ▶ lightPassEnd
- ▶ mainOpaque
- ▶ linearizeZ
- ▶ mainOpaqueEmissive
- ▶ mainTransDecal
- ▶ fgOpaqueEmissive
- ▶ subsurfaceScattering
- ▶ skyAndFog
- ▶ hairCoverage
- ▶ mainTransDepth
- ▶ linearizeZ
- ▶ mainTransparent
- ▶ halfResUpsample
- ▶ motionBlurDerive
- ▶ motionBlurVelocity
- ▶ motionBlurFilter
- ▶ filmicEffectsEdge
- ▶ spriteDof
- ▶ fgTransparent
- ▶ lensScope
- ▶ filmicEffects
- ▶ bloom
- ▶ luminanceAvg
- ▶ finalPost
- ▶ overlay
- ▶ fxaa
- ▶ smaa
- ▶ resample
- ▶ screenEffect
- ▶ hmdDistortion

This is the rendering passes we had in Frostbite few years ago for Battlefield 4. Our pipeline now with PBR has even more passes and complexity.

WorldRenderer challenges

- ▶ Explicit immediate mode rendering
- ▶ Explicit resource management
 - ▶ Bespoke, artisanal hand-crafted ESRAM management
 - ▶ Multiple implementations by different game teams
- ▶ Tight coupling between rendering systems
- ▶ Limited extensibility
- ▶ Game teams must fork / diverge to customize
- ▶ Organically grew from **4k** to **15k** SLOC
 - ▶ Single functions with over **2k** SLOC
 - ▶ Expensive to maintain, extend and merge/integrate



World Renderer state as it evolved from 2007 to 2016.

Modular WorldRenderer goals

- ▶ High-level knowledge of the full frame
- ▶ Improved **extensibility**
 - ▶ Decoupled and composable code modules
 - ▶ Automatic resource management
- ▶ Better visualizations and diagnostics

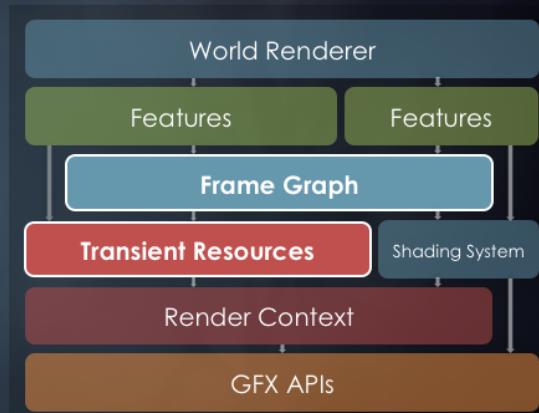


Major World Renderer re-architecture in 2016 to address accumulated technical debt, improve extensibility and maintainability.

Not micro-management of explicit passes and resources
Not hacking monolithic functions inside engine code
Not baby-sitting of memory allocation and aliasing.

New architectural components

- ▶ **Frame Graph**
 - ▶ High-level representation of **render passes** and **resources**
 - ▶ Full knowledge of the frame
- ▶ **Transient Resource System**
 - ▶ Resource allocation
 - ▶ Memory aliasing



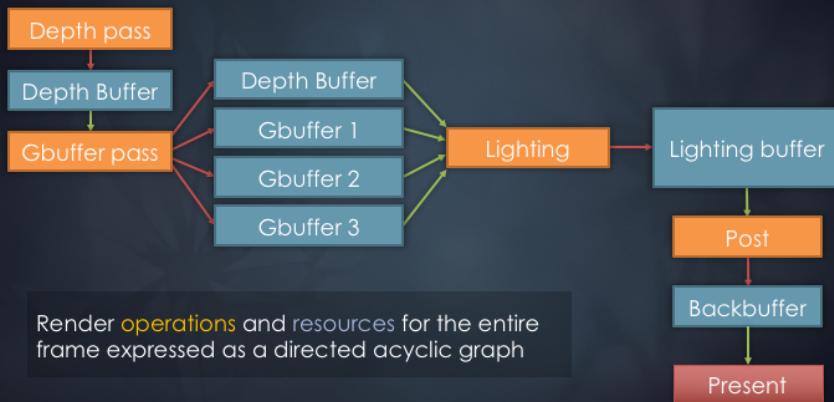


Frame Graph

Frame Graph goals

- ▶ Build **high-level knowledge** of the entire frame
 - ▶ Simplify resource management
 - ▶ Simplify rendering pipeline configuration
 - ▶ Simplify async compute and resource barriers
- ▶ Allow self-contained and **efficient rendering modules**
- ▶ Visualize and debug complex rendering pipelines

Frame Graph example



Toy example of a frame graph that implements a deferred shading pipeline.

The graph contains render passes and resources as nodes.

Access declarations / dependencies are edges.

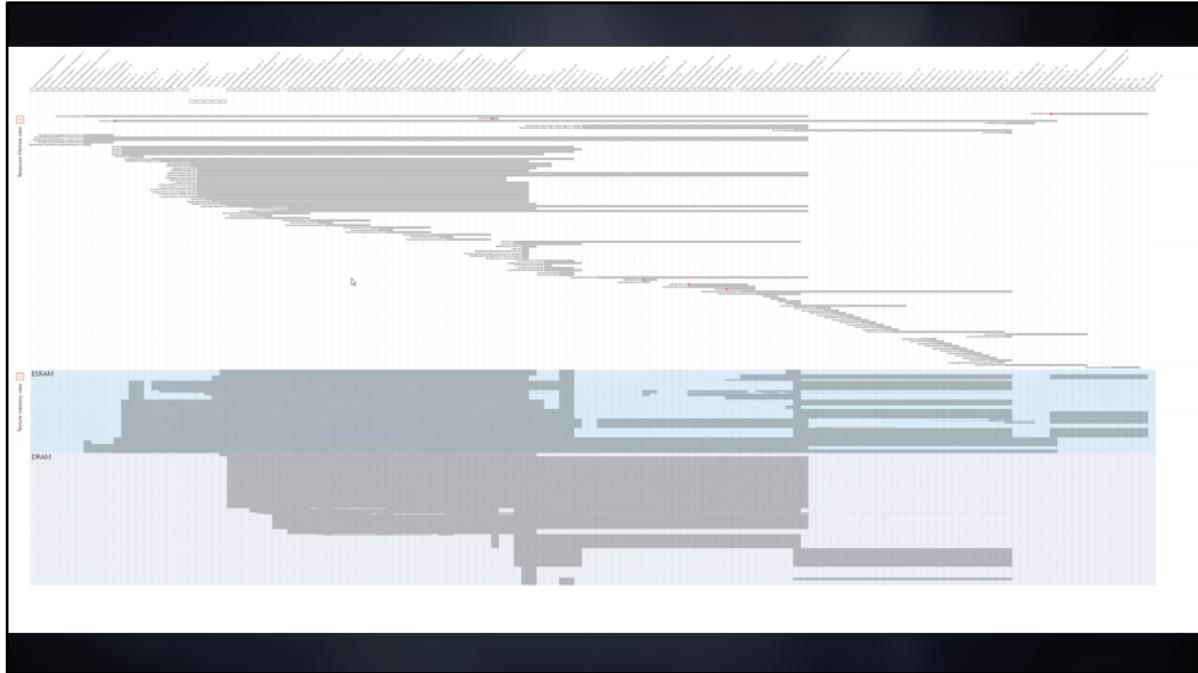
Graph of a Battlefield 4 frame



Debug graph visualization using GraphViz. Output is a **searchable** PDF (not static image).

Graphs can be surprisingly large and complex.

While can be useful in some cases, it's definitely not the primary visualization tool that we ended up using.



We wrote a custom visualization script (HTML+Javascript and JSON data exported from runtime).

JSON data contains information about all render passes and resources.

For each render pass we know which resources were created, read or written.

For each resource we know its complete memory layout and various metadata (debug name, size, format, etc.).

The visualization is interactive and provides a much more useful overview of what's going on in a frame, similar to what you'd find in PIX.

Frame Graph design

- ▶ Moving away from immediate mode rendering
- ▶ Rendering code split into **passes**
- ▶ Multi-phase retained mode rendering API
 1. Setup phase
 2. Compile phase
 3. Execute phase
- ▶ Built from scratch every frame
- ▶ Code-driven architecture

FrameGraph is a step away from immediate mode rendering towards retained. We build the graph every frame from scratch, since rendering configuration may change dynamically based on player actions, cut-scenes, etc. The big assumption is that setup phase is relatively cheap, as we're only dealing with a relatively small number of render passes and resources.

Frame Graph setup phase

- ▶ Define render / compute passes
- ▶ Define **inputs** and **output** resources for each pass
- ▶ Code flow is similar to immediate mode rendering



Flow is similar to IM rendering, but we are not generating any GPU commands during this phase. Just building up the information about rendering operations for the frame. All resources are **virtual** during graph building. Render pass inputs and outputs are declared using virtual resource handles.

Frame Graph resources

- ▶ Render passes must declare all used resources
 - ▶ Read
 - ▶ Write
 - ▶ Create
- ▶ External permanent resources are **imported** to Frame Graph
 - ▶ History buffer for TAA
 - ▶ Backbuffer
 - ▶ etc.



Some render passes may have effects that are not visible to FrameGraph (for example data read-back from GPU). Such passes are explicitly marked as having side-effects.

Some persistent render targets are still required (TAA, SSR, etc.). They can be imported into FrameGraph.

Writing to imported resource counts as side-effect of a render pass, which ensures that it is not culled during the compilation phase.

Frame Graph resource example

```
RenderPass::RenderPass(FrameGraphBuilder& builder)
{
    // Declare new transient resource
    FrameGraphTextureDesc desc;
    desc.width = 1280;
    desc.height = 720;
    desc.format = RenderFormat_D32_FLOAT;
    desc.initialState = FrameGraphTextureDesc::Clear;
    m_renderTarget = builder.createTexture(desc);
}
```



Simple dummy render pass that produces a render target resource.
Very similar to creating a regular texture, except we also specify initial resource state
(clear or discard/undefined)

Frame Graph setup example

```
RenderPass::RenderPass(FrameGraphBuilder& builder,
    FrameGraphResource input,
    FrameGraphMutableResource renderTarget)
{
    // Declare resource dependencies
    m_input = builder.read(input, readFlags);
    m_renderTarget = builder.write(renderTarget, writeFlags);
}
```



Render pass that reads from one texture and writes to another.

Writing to a texture produces a renamed handle. This allows us to catch errors when resources are modified in undefined order (when same resource is written by different passes).

Renaming resources enforces a specific execution order of the render passes.

Advanced FrameGraph operations

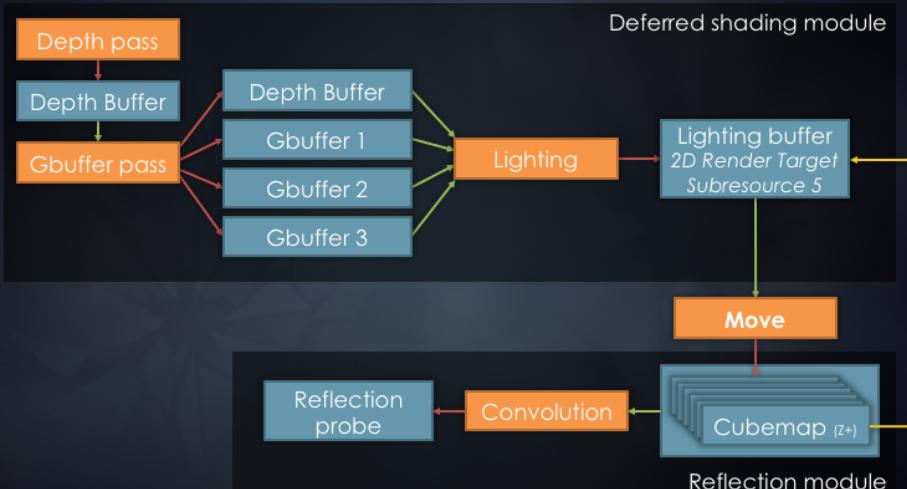
- ▶ Deferred-created resources
 - ▶ Declare resource early, allocate on first actual use
 - ▶ Automatic resource bind flags, based on usage
- ▶ Derived resource parameters
 - ▶ Create render pass output based on input size / format
 - ▶ Derive bind flags based on usage
- ▶ **MoveSubresource**
 - ▶ Forward one resource to another
 - ▶ Automatically creates sub-resource views / aliases
 - ▶ Allows "time travel"

Abstract / virtualized resources allow some convenient tricks. Resources may be declared early, but their memory will be allocated only on first use. An example use case is a depth buffer resource. We know that we will need one to do 3D rendering, but we don't necessarily know (or care) if our rendering pipeline is using depth pre-pass. Depth pre-pass, gbuffer pass and forward-shaded geometry passes all simply write to the resource that they require to be declared early.

FrameGraph resource handles have metadata attached to them that can be queried during setup phase.

This allows some render passes to create derived resources. For example, a generic down-sample render pass can create an output resource that shares all properties of the input, but overrides width/height. Resource bind flags can be also automatically computed based on how the resource is used. The pass that creates a render target resource does not need to know that this resource is going to be used as a UAV, etc. One of the more magical operations that are possible on virtualized resources is **MoveSubresource**. **Create aliases of resources that will be created by the future render passes.**

MoveSubresource example



A generic rendering pipeline can be implemented that creates an output texture, which is a simple 2D image resource.

It can be combined with a reflection probe filtering pipeline that takes a cubemap input.

Move operation can be used to assign "Lighting buffer" resource to one of the cubemap faces.

This causes the deferred shading module to write directly to the cubemap face, instead of creating a separate render target.

The same deferred shading module can be used in a different context, without move operation. In this case FrameGraph will allocate a transient render target for the output.

Frame Graph compilation phase

- ▶ Cull unreferenced resources and passes
 - ▶ Can be a bit more sloppy during declaration phase
 - ▶ Aim to reduce configuration complexity
 - ▶ Simplifies conditional passes, debug rendering, etc.
- ▶ Calculate **resource lifetimes**
- ▶ Allocate concrete GPU resources based on usage
 - ▶ Simple greedy allocation algorithm
 - ▶ Acquire right before first use, release after last use
 - ▶ Extend lifetimes for async compute
 - ▶ Derive resource bind flags based on usage



FrameGraph data structures

Flat array of used resource handles per RenderPass

Flat array of RenderPasses in FrameGraph

Flat array of resources in ResourceRegistry

Resource handles are just indices into this array

Compilation phase linearly walks through all RenderPasses

Computes reference counts for resources

Computes first and last users for resources

Computes async wait points and resource barriers

RenderPass execution order is defined by setup order

No re-ordering during compilation

Culling algorithm

Simple graph flood-fill from unreferenced resources.

Compute initial resource and pass reference counts

renderPass.refCount++ for every resource write
resource.refCount++ for every resource read

Identify resources with refCount == 0 and push them on a stack

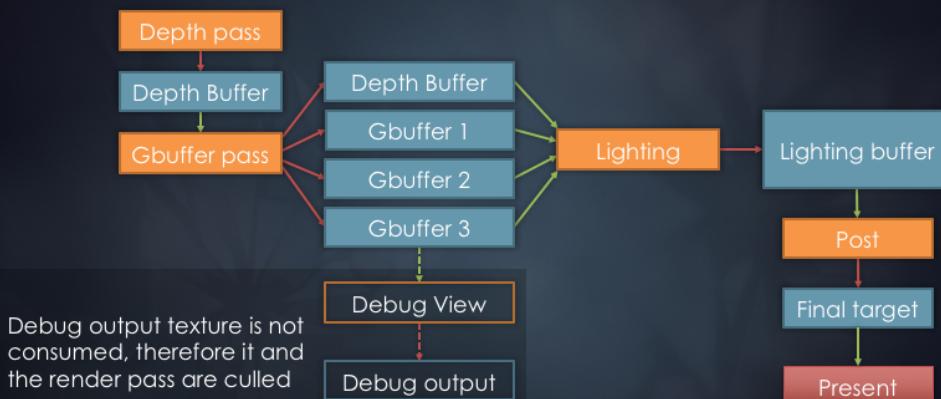
While stack is non-empty

Pop a resource and decrement ref count of its producer

If producer.refCount == 0, decrement ref counts of resources that it reads

Add them to the stack when their refCount == 0

Sub-graph culling example

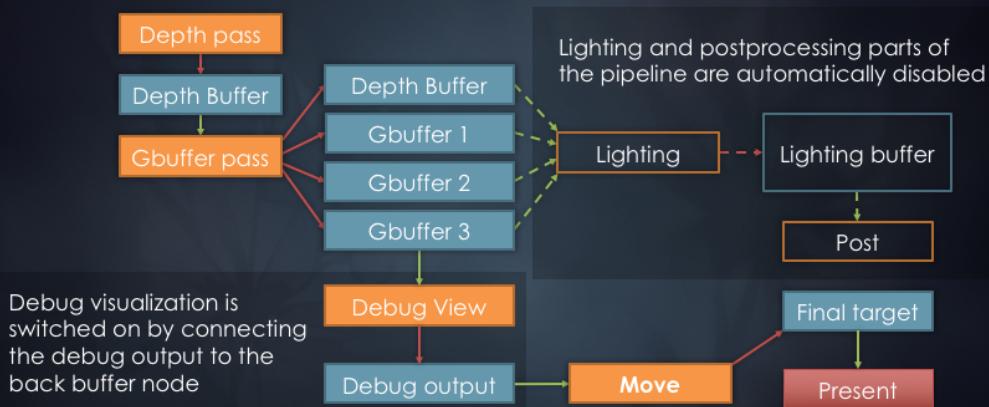


It is sometimes convenient to add render passes and resources to the graph without checking if they are needed first.

For example, we can always add certain debug visualizations or specialized passes, such as depth buffer linearization.

This cuts down on the rendering pipeline configuration complexity a bit.

Sub-graph culling example



The engine contains many features and deciding whether to execute a certain render pass can be a chore. It also introduces some coupling between passes.

Lighting passes don't need to know anything about the debug output. When debug is enabled, it overrides the lighting output, which will cull it. This leads to more decoupled / modular code.

Frame Graph execution phase



- ▶ Execute callback functions for each render pass
- ▶ Immediate mode rendering code
 - ▶ Using familiar RenderContext API
 - ▶ Set state, resources, shaders
 - ▶ Draw, Dispatch
- ▶ Get **real** GPU resources from handles generated in setup phase

Execution phase is quite simple. Iterate over render passes that are not culled and call their execution callback function.

This phase is almost identical to how rendering was done before FrameGraph. Just use RenderContext API, except must de-virtualize FrameGraph resources first.

Async compute

- ▶ Could derive from dependency graph automatically
- ▶ Manual control desired
 - ▶ Great potential for **performance** savings, but...
 - ▶ Memory increase
 - ▶ Can hurt performance if misused
- ▶ Opt-in per render pass
- ▶ **Kicked off** on main timeline
- ▶ Sync point at first use of output resource on another queue
- ▶ Resource lifetimes automatically extended to sync point

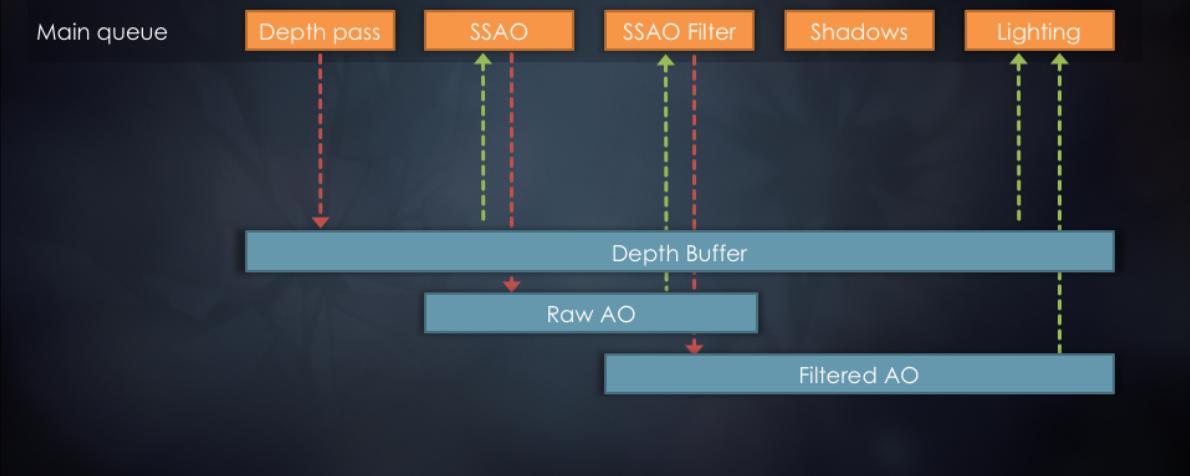
Efficient async compute requires some hand-holding today. While we do have all the render pass and resource dependencies in the graph, we don't know what bottlenecks will exist on the GPU during execution. Don't want bandwidth-heavy compute passes to run with bandwidth-heavy graphics work (shouldn't be news to anyone).

Async operations will increase memory water mark, because resource lifetimes are extended (more resources are alive simultaneously). Need to be a bit careful.

Ended up with a manual opt-in mechanism for render passes. Async passes are kicked off on the main timeline at the point where they'd execute serially (we don't re-order passes).

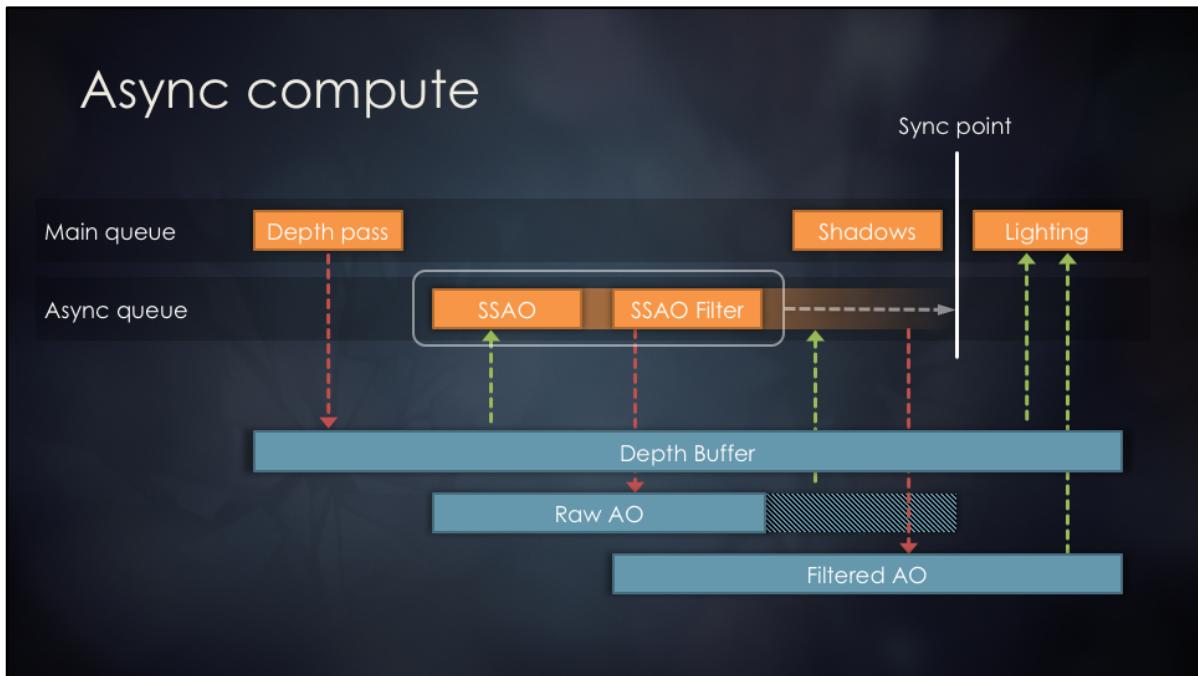
Synchronization point is automatically added on the main pipeline before the first render pass that consumes the output of async pass.

Async compute



Example compute-based AO filtering pipeline.

Async compute



AO buffer generation and filtering can be moved to async queue, but resource lifetimes must be extended a bit (up to the sync point).

Frame Graph async setup example

```
AmbientOcclusionPass::AmbientOcclusionPass(FrameGraphBuilder& builder)
{
    // The only change required to make this pass
    // and all its child passes run on async queue
    builder.asyncComputeEnable(true);

    // Rest of the setup code is unaffected
    // ...
}
```

This is all you have to do in high-level code. Super simple to answer questions like “what would happen if we ran this async?”.

In the future we’d like to explore automatic render pass re-ordering, perhaps with profile-guided optimization step.

Pass declaration with C++

- ▶ Could just make a C++ class per RenderPass
 - ▶ Breaks code flow
 - ▶ Requires plenty of boilerplate
 - ▶ Expensive to port existing code
- ▶ Settled on **C++ lambdas**
 - ▶ Preserves code flow!
 - ▶ Minimal changes to legacy code
 - ▶ Wrap legacy code in a lambda
 - ▶ Add a resource usage declarations

Programmer convenience is very important. Don't want to introduce too much boilerplate code or break the code flow.

Started with a C++ class with virtual execute(), but quickly realized that such approach requires moving quite a lot of code around.

It also requires a bit of plumbing to pass data between setup and execution phases.

Implemented a lambda-based API to improve the convenience. This also greatly simplified the effort of porting legacy rendering code to the new system.

Started by simply wrapping huge chunks of code in lambdas. Gradually replaced raw resources with transients and sub-divided monolithic lambdas into smaller ones.

Eventually moved out final small code blocks into stand-alone functions. Spaghetti is mostly untangled ☺

The price that we have to pay is a template-heavy FrameGraph setup API.

Pass declaration with C++ lambdas

```
FrameGraphResource addMyPass(FrameGraph& frameGraph,
                             FrameGraphResource input, FrameGraphMutableResource output)
{
    struct PassData
    {
        FrameGraphResource input;
        FrameGraphMutableResource output;
    };

    auto& renderPass = frameGraph.addCallbackPass<PassData>("MyRenderPass",
        [&](RenderPassBuilder& builder, PassData& data)
    {
        // Declare all resource accesses during setup phase
        data.input = builder.read(input);
        data.output = builder.useRenderTarget(output).targetTextures[0];
    },
    [=](const PassData& data, const RenderPassResources& resources, IRenderingContext* renderContext)
    {
        // Render stuff during execution phase
        drawTexture2d(renderContext, resources.getTexture(data.input));
    });
}

return renderPass.output;
}
```

`addCallbackPass()` is a template function that creates a render pass class behind the scenes that's parametrized by the `PassData` and the execution lambda.

Setup lambda is inlined in `addMyPass()`, but execute lambda is deferred. Setup lambda may capture everything by reference, but execute must capture by value. Capturing data by value is a little bit dangerous since it's possible to accidentally capture a pointer that's released before execution phase. It's also possible to accidentally capture huge structures by value. Luckily, we can enforce that the size of execution lambda is below a certain size at compile time (we settled on 1KB limit).

Render modules

- ▶ Two types of render modules:
 1. Free-standing **stateless** functions
 - ▶ Inputs and outputs are Frame Graph resource handles
 - ▶ May create nested render passes
 - ▶ Most common module type in Frostbite
 2. Persistent render modules
 - ▶ May have some persistent resources (LUTs, history buffers, etc.)
- ▶ WorldRenderer still orchestrates high-level rendering
 - ▶ Does not allocate any GPU resources
 - ▶ Just kicks off rendering modules at the high level
 - ▶ Much easier to extend
 - ▶ Code size reduced from 15K to 5K SLOC

Communication between modules

- ▶ Modules may communicate through a **blackboard**
 - ▶ Hash table of components
 - ▶ Accessed via component Type ID
 - ▶ Allows **controlled coupling**

```
#include "BlurModule.h"
void BlurModule::renderBlurPyramid(
    FrameGraph& frameGraph,
    FrameGraphBlackboard& blackboard)
{
    // Produce blur pyramid in the blur module
    auto& blurData = blackboard.add<BlurPyramidData>();
    addBlurPyramidPass(frameGraph, blurData);
}

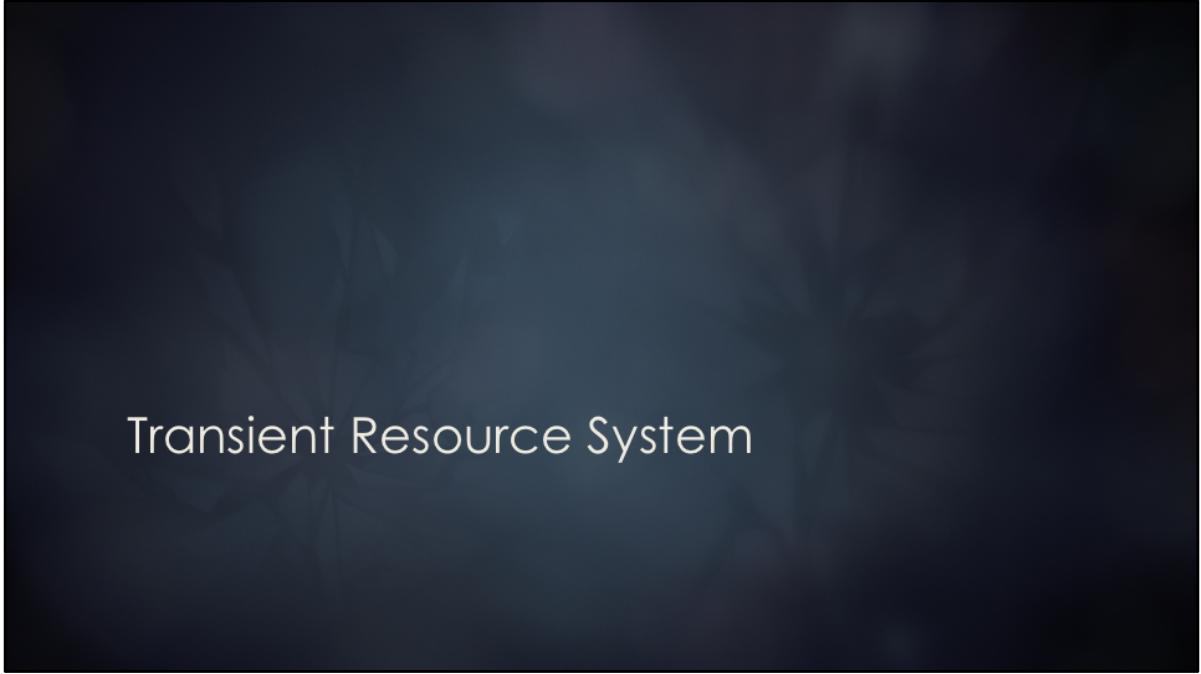
#include "TonemapModule.h"
void TonemapModule::createBlurPyramid(
    FrameGraph& frameGraph,
    const FrameGraphBlackboard& blackboard)
{
    // Consume blur pyramid in a different module
    const auto& blurData = blackboard.get<BlurPyramidData>();
    addTonemapPass(frameGraph, blurData);
```

Not necessarily need a single global blackboard. Modules may create their own blackboard within their setup scope, propagate some data from the parent into it and then copy results into parent blackboard at the end.

While blackboard is great for decoupling, it does make the code harder to understand. If a module takes a blackboard as a parameter, it's not possible to tell at the call site which resources will actually be accessed.
The module code itself must be viewed to answer this.

Invalid blackboard access can only be validated at run-time.

On balance, we believe that the benefits outweigh the drawbacks.



Transient Resource System

The back-bone of FrameGraph.

Transient resource system

- ▶ **Transient** /'trænzɪənt/ adjective
Lasting only for a short time; impermanent.
- ▶ Resources that are alive for no longer than one frame
 - ▶ Buffers, depth and color targets, UAVs
 - ▶ Strive to minimize resource life times **within** a frame
- ▶ Allocate resources where they are used
 - ▶ Directly in *leaf* rendering systems
 - ▶ Deallocate as soon as possible
 - ▶ Make it easier to write self-contained features
- ▶ **Critical component of Frame Graph**

Transient resource system back-end

- ▶ Implementation depends on platform capabilities
 - ▶ Aliasing in physical memory (XB1)
 - ▶ Aliasing in virtual memory (DX12 PS4)
 - ▶ Object pools (DX11)
- ▶ Atomic linear allocator for buffers
 - ▶ No aliasing, just blast through memory
 - ▶ Mostly used for sending data to GPU
- ▶ Memory pools for textures



Transient textures on PlayStation 4



- Reserve a single large **virtual memory** pool
- Allocate texture virtual memory block on first use
 - Use a general purpose non-local memory allocator
 - Patch or allocate GNM resource descriptors as needed
- Return virtual memory block after last use
- Commit physical memory to cover VA range used in current frame
 - Grow the physical memory pool on demand
 - Shrink down to the high water mark of last N frames
- Resources overlap in **virtual address** space
 - Understood natively by PS4 graphics debugging tools (Razor)

Transient textures on DirectX 12 PC



A bit similar to PS4, except many disjoint address ranges instead of just one.
Can't use a single range, as it's impossible to shrink it without stalling the GPU or temporarily increasing memory usage.
Despite these shortcomings, we're still able to re-use memory sometimes and see a significant overall water mark reduction.

Frostbite does not currently perform global memory allocation optimization, but it could theoretically be implemented. A global optimization pass would allow merging Heap 2 into Heap 6. This would bring down the overall number of heaps and the memory water mark.

Concrete problems with resource heaps in current D3D12:

Tier 1 heaps have restrictions on types of resources that can be placed in them. Only buffers **or** only textures **or** only render targets and depth buffers. Must create separate heaps for different resource types. Most transient resources that we alias are RT or DS, so it's not too bad. We force the RT flag on a transient texture even if

user did not specifically request it.

Tier 2 heaps are better, as all types of resources can be aliased. They are still not ideal, as we must allocate many heaps and sub-allocate within them. This leads to more fragmentation compared to allocating from a single large address range. We can't allocate a single huge heap, as we can't shrink it. Compromise is to create one large-ish persistent transient resource heap and then create smaller *overflow* heaps.

Once a resource is created, it can't be moved. This means that if memory allocation "schedule" changes a bit, some objects will change their placements and will have to be re-created. It's possible to work around this issue to some degree by caching placed D3D objects (resources and various views) and re-using them when possible (potentially many frames later, when allocation schedule changes again to a compatible one). Resource allocation schedule may change based on player actions, cut-scenes, UI, etc. However, there is typically only a handful of unique schedules, so it's possible to use an LRU cache.

These problems simply don't exist on consoles. Tiled resources could be quite convenient in the future (almost the same level of efficiency as XB1 memory aliasing), however as of October 2016 there are significant CPU and GPU overheads to using them as RTVs / DSVs. Additionally, resource heap tier restrictions prevent efficient tile mapping updates via CopyTileMappings. We sometimes want to use multiple heaps as page sources to back a single resource. Current UpdateTileMappings API can only take a single heap pointer, therefore multiple API calls are required.

Transient textures on Xbox One



Fragmentation-free dynamic memory allocation and aliasing

Close to optimal ESRAM utilization automatically

- Don't need contiguous memory blocks

- Resources may be fully or partially in ESRAM

- Overflow to DRAM when every ESRAM page is in use

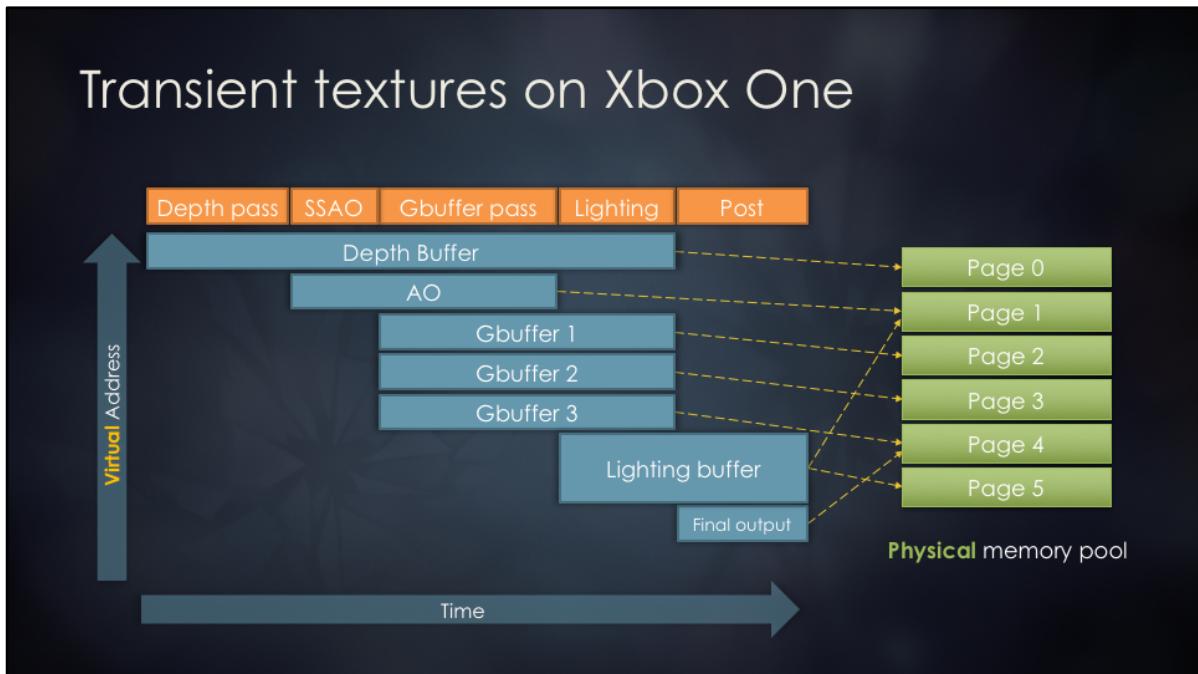
Hand-tune memory allocation based on profiling

- Deny ESRAM for some resources

- Allocate ESRAM top-down or bottom-up

- Restrict ESRAM to % of the resource, place rest in DRAM

Transient textures on Xbox One



Use a **physical memory** pool of ESRAM and DRAM pages

Allocate all resources at unique virtual addresses

`VirtualAlloc, CreatePlacedResourceX`

Allocate physical memory pages from pool on first use

ESRAM pages first, overflow to DRAM

Extend DRAM pool on demand

Shrink DRAM pool based on high water mark of last N frames

Return physical pages to the pool after last use

Update GPU page table before executing other commands

XB1-specific `ID3D12CommandQueue` API

Conceptually similar to `CopyTileMappings`

Page table update happens on GPU timeline

Memory aliasing considerations

- ▶ Must be **very careful**
- ▶ Ensure valid resource metadata state (FMASK, CMASK, DCC, etc.)
 - ▶ Perform fast clears **or** discard / over-write resources **or** disable metadata
- ▶ Ensure resource lifetimes are correct
 - ▶ Harder than it sounds
 - ▶ Account for compute and graphics pipelining
 - ▶ Account for async compute
 - ▶ Ensure that physical pages are written to memory before reuse

DiscardResource & Clear

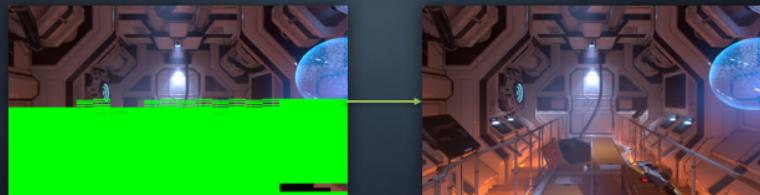
- ▶ Must be the first operation on a newly allocated resource
- ▶ Requires resource to be in the **render target** or **depth write** state
- ▶ Initializes resource metadata (HTILE, CMASK, FMASK, DCC, etc.)
 - ▶ Similar to performing a fast-clear
 - ▶ Resource contents remains undefined (not actually cleared)
- ▶ Prefer **DiscardResource** over **Clear** when possible

Aliasing barriers



Aliasing barriers

- ▶ Add synchronization between work on GPU
- ▶ Add necessary cache flushes
- ▶ Use precise barriers to minimize performance cost
- ▶ Can use wildcard barriers for difficult cases (*but expect IHV tears*)
- ▶ Batch with all your other resource barriers in DirectX 12!

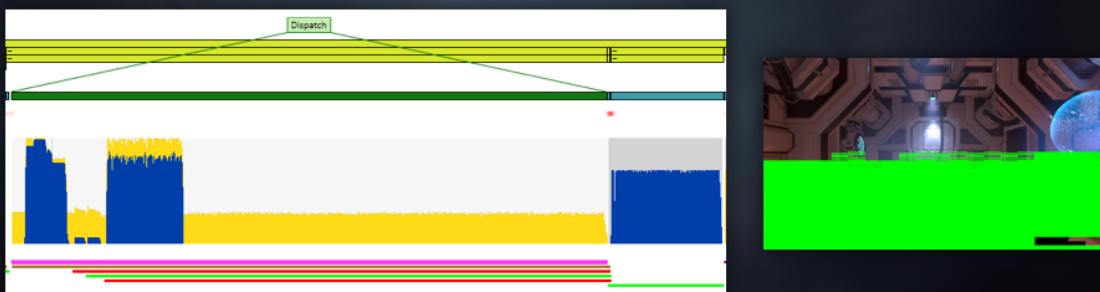


Since different rendering passes may use the same physical memory, we need to add synchronization point between them to make sure that they don't run in parallel and overwrite each others memory.

We do this by using aliasing barriers, which will add the necessary pipeline and cache flushes.

Aliasing barrier example

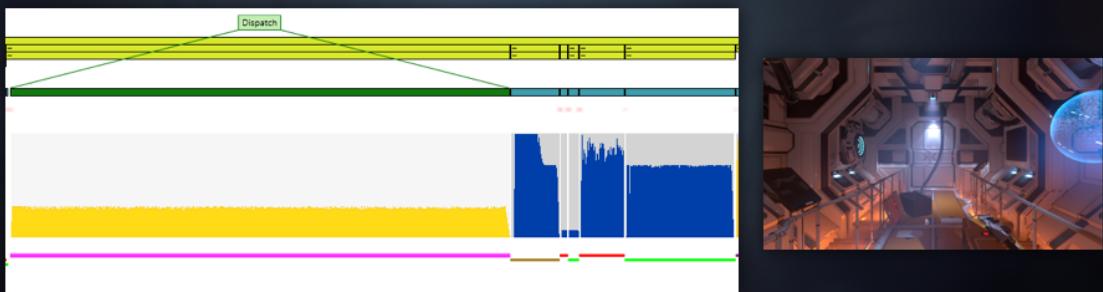
- ▶ Potential aliasing hazard due to pipelined **CS** and **PS** work
 - ▶ **CS** and **PS** use different D3D sources, so transition barriers aren't enough
- ▶ Must flush **CS** before **PS** or extend **CS** resource lifetimes



Graphics and compute passes don't overlap logically, but run in parallel because they are independent (they don't have any producer-consumer relationship or any shared logical resources).

Aliasing barrier example

- ▶ Serialized compute work ensures correctness when memory aliasing
 - ▶ May hurt performance in some cases
- ▶ Use explicit async compute when overlap is critical for performance

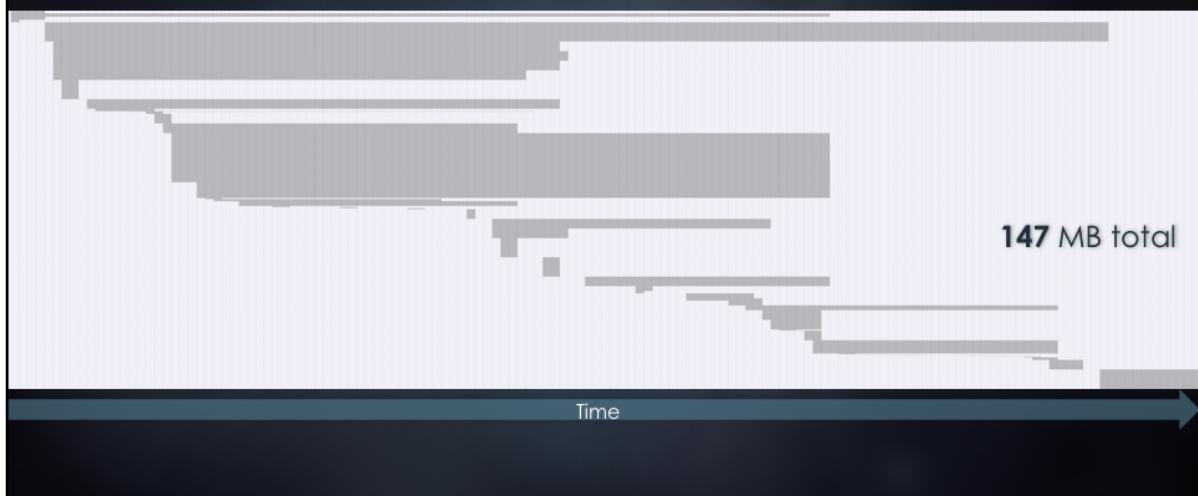


Using explicit async compute allows us to ensure that resource memory isn't released until compute chain is done.

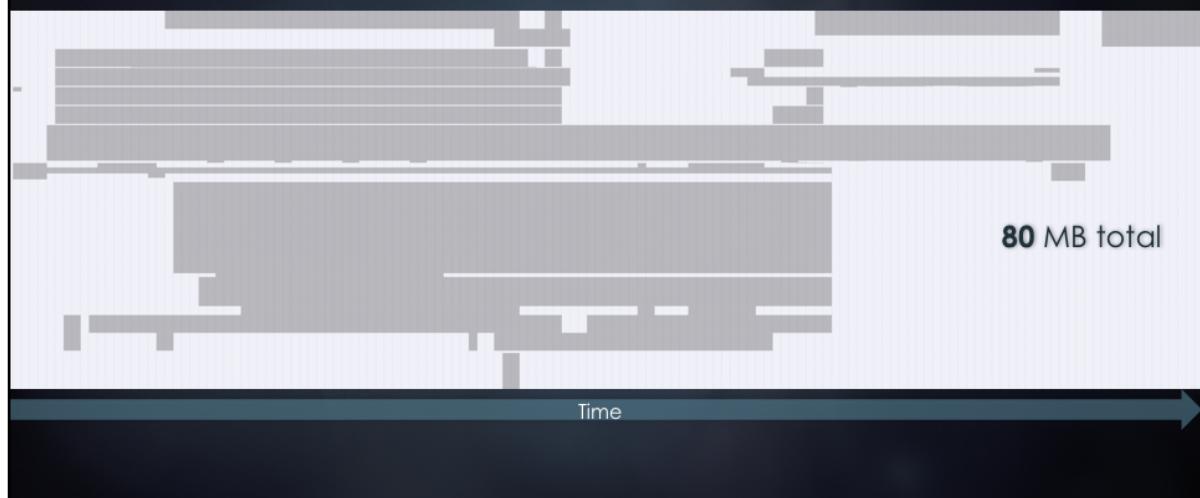
In this particular example, overlapped and non-overlapped versions have exactly the same performance characteristics. Overlap does not *always* improve perf. In fact, it may sometimes hurt it.



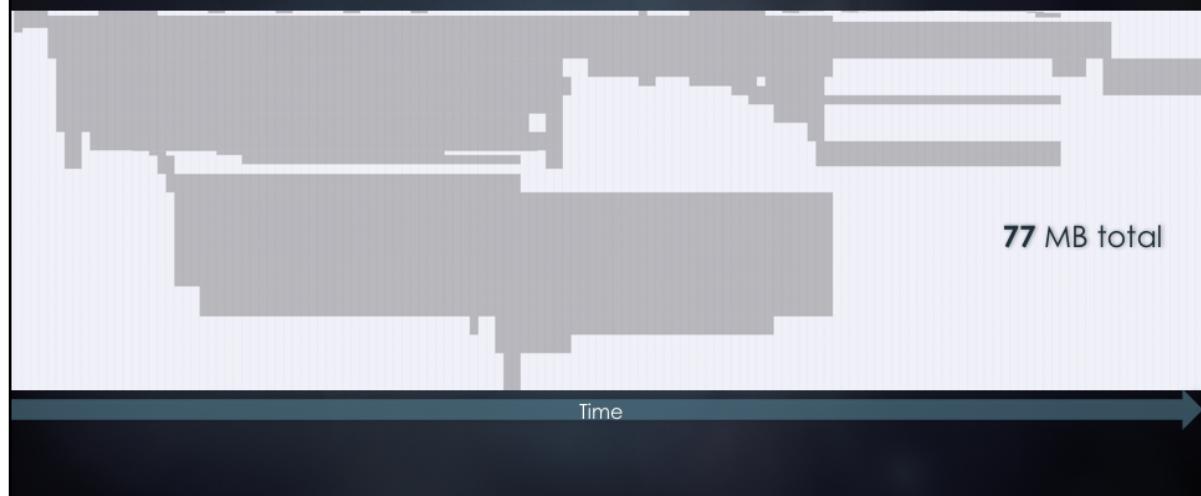
Non-aliasing memory layout (720p)



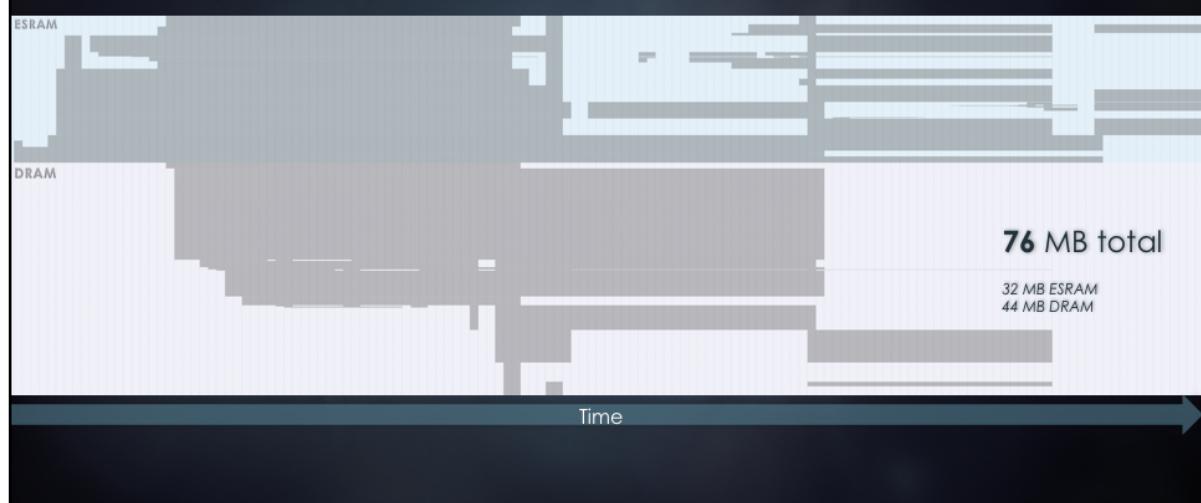
DirectX 12 PC memory layout (720p)

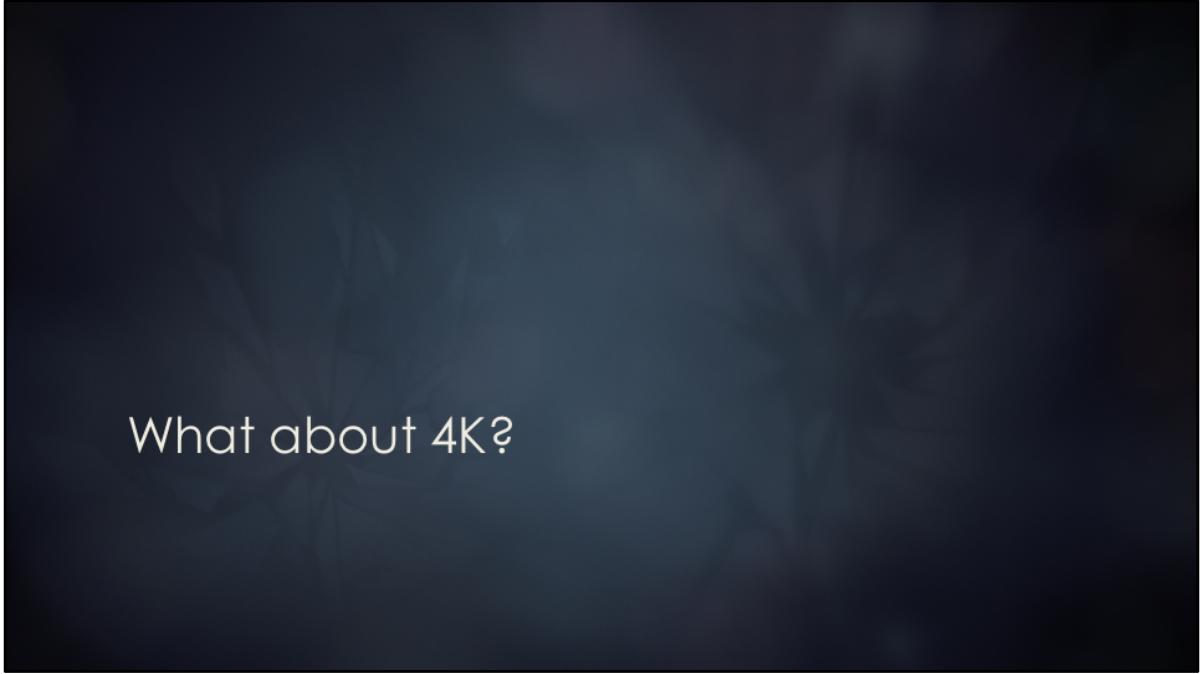


PlayStation 4 memory layout (720p)



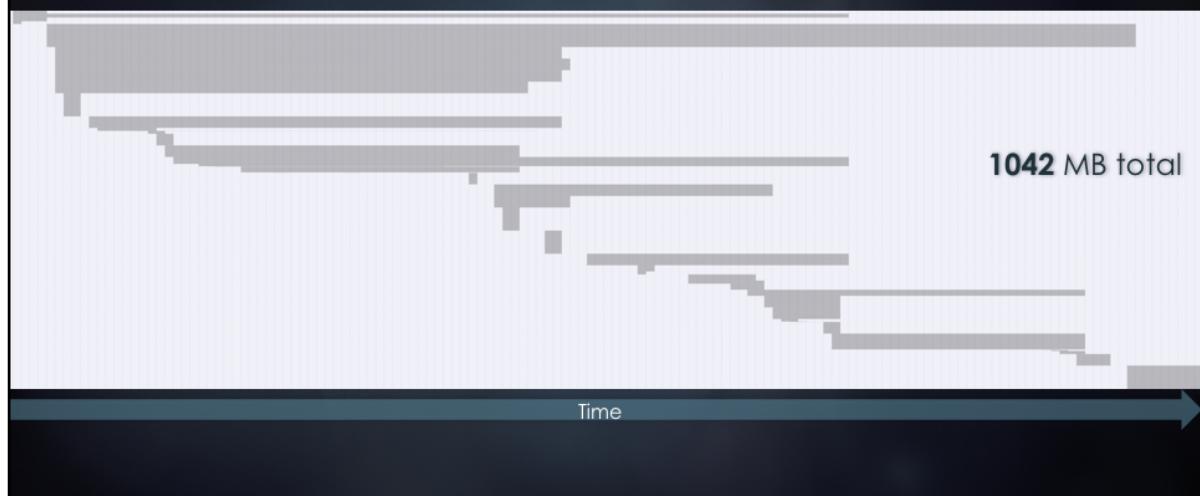
Xbox One memory layout (720p)



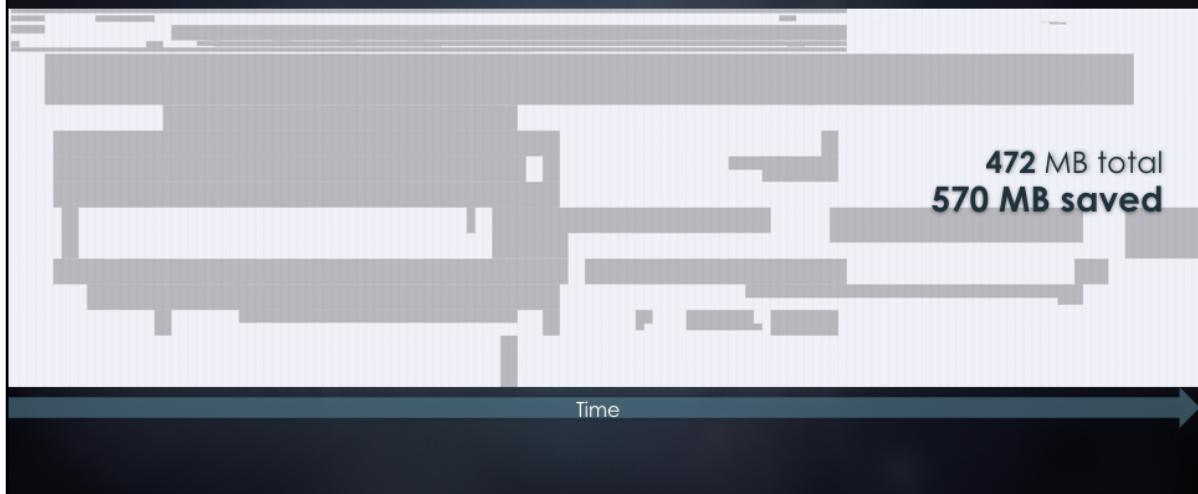


What about 4K?

Non-aliasing memory layout (4K, DX12 PC)



Aliasing memory layout (4K, DX12 PC)



... now we finally have space for those 16k² eyeball textures!

Conclusion

Summary

- ▶ Many benefits from **full frame knowledge**
 - ▶ Huge memory savings from resource aliasing
 - ▶ Semi-automatic async compute
 - ▶ Simplified rendering pipeline configuration
 - ▶ Nice visualization and diagnostic tools
- ▶ **Graphs** are an attractive representation of rendering pipelines
 - ▶ Intuitive and familiar concept
 - ▶ Similar to CPU job graphs or shader graphs
- ▶ Modern C++ features ease the pain of retained mode API

Full frame knowledge and **visualization** is an awesome tool that allowed us to spot inefficiencies in resource allocation, possibilities for async compute, etc.

Future work

- ▶ **Global optimization of resource barriers**
- ▶ Async compute bookmarks
- ▶ Profile-guided optimization
 - ▶ Async compute
 - ▶ Memory allocation
 - ▶ ESRAM allocation

We're only starting to scratch the surface of what's possible with the modern rendering engine architecture and APIs. We expect to see more engines in the future moving to a similar design (high-level frame setup), since that appears to be the most optimal way to drive DX12 and Vulkan style renderers.

Special thanks

- ▶ Johan Andersson (Frostbite Labs)
- ▶ Charles de Rousiers (Frostbite)
- ▶ Tomasz Stachowiak (Frostbite)
- ▶ Simon Taylor (Frostbite)
- ▶ Jon Valdes (Frostbite)
- ▶ Ivan Nevraev (Microsoft)
- ▶ Matt Lee (Microsoft)
- ▶ Matthäus G. Chajdas (AMD)
- ▶ Christina Coffin (Light & Dark Arts)
- ▶ Julien Merceron (Bandai Namco)

Questions?

YURIY@FROSTBITE.COM



@YURIYODONNELL

The End