



DESTINY'S MULTITHREADED RENDERING ARCHITECTURE

NATALYA TATARCHUK
ENGINEERING ARCHITECT



BUNGIE

Hello! Welcome to Destiny's Multi-threaded Renderer Architecture talk.
My name is Natalya Tatarчук, and I'm a graphics engineering architect at Bungie.

This Talk Won't Cover...

- Destiny graphics algorithms
- Destiny shader techniques
- If you want to learn more about them:
 - See SIGGRAPH 2009-2014 talks and I3D 2015 keynote for some of our techniques
 - <http://advances.realtimerendering.com>

DESTINY 

Today's talk is not about specific graphics techniques or shader tricks for Destiny. We covered a fair amount of those in previous years and you can find most of the slides on the advances website.

Assumptions

- Basic familiarity with task-based parallelism, job manager designs and synchronization primitives
- Helpful Destiny engine design details in:
 - *Multithreading the Entire Destiny Engine*
by Barry Genova (Thu, March 5 | 4:00pm - 5:00pm)

DESTINY 

For this talk, I will assume that you have some familiarity with task-based parallelism concepts, job managers and synchronization primitives.

Also, Barry Genova covered the foundations of the threading design for the Destiny engine in an earlier talk today – I hope you had a chance to attend if. If not, you can watch it on the GDC vault later on.

– Today's – SPECIALS

- Destiny core renderer architecture
- Game simulation to GPU data flow
- Jobification and load-balancing considerations
- Latency reduction techniques
 - Input and rendering latency reduction
 - Keeping GPU fully saturated
- Complexity encapsulation

DESTINY 

Today, I will cover the Destiny core renderer architecture, talk about how the engine data flows from game simulation through to the GPU, discuss how we approached renderer jobification, and relevant considerations for task- and data-parallel execution for our renderer workloads, including dynamic load-balancing.

We'll cover topics for reducing game input and rendering latency, and methods of keeping GPU fully saturated at all times.

I'll also touch on the architectural principles that allowed us to encapsulate the complexity behind well-designed abstractions, allowing our graphics engineers to focus on what they do best – writing Destiny graphics features.

If there's anything I want you to walk away with today it is that creating data-driven pipelined architectures can give you huge flexibility for creating graphics features and yet help you optimize the overall performance of your engine across different platforms.

Outline

- Coarse-grained parallelism
- Destiny renderer goals
- Decoupling simulation and rendering
- Jobification for core workloads
- Data-driven render submission
- Advanced optimizations
- Conclusions

DESTINY 

This breaks down into the following sections – we'll do a quick background on coarse-grained parallelism, talk about our goals for Destiny, then dive into the deep details of the architecture in the next four sections before drawing some conclusions about our experience in the end.

Coarse-Grained Parallelism


A 10,000' View of a Frame in a Game



Let's do a high level overview of what happens in a single frame of a game

A 10,000 ' View of a Frame in a Game

SIMULATE
GAME
OBJECTS

DESTINY 

For every game tick, we start by running our game simulation. This is where we run the physics engine, our AI, animation and any code that affects the game play.

A 10,000 ' View of a Frame in a Game

SIMULATE
GAME
OBJECTS

DETERMINE
WHAT TO
RENDER

DESTINY 


Then we use the latest simulation data to determine what elements would be visible

A 10,000 ' View of a Frame in a Game

SIMULATE
GAME
OBJECTS

DETERMINE
WHAT TO
RENDER


GENERATE
GPU
COMMANDS

DESTINY 

Then we convert these elements to a set of GPU commands. We refer to this operation as 'CPU submit'. The result of this is GPU command buffers that we flush to the GPU

A 10,000 ' View of a Frame in a Game




DESTINY 

which processes them to output results to the backbuffer, which we flip to display to the player.

A 10,000 ' View of a Frame in a Game

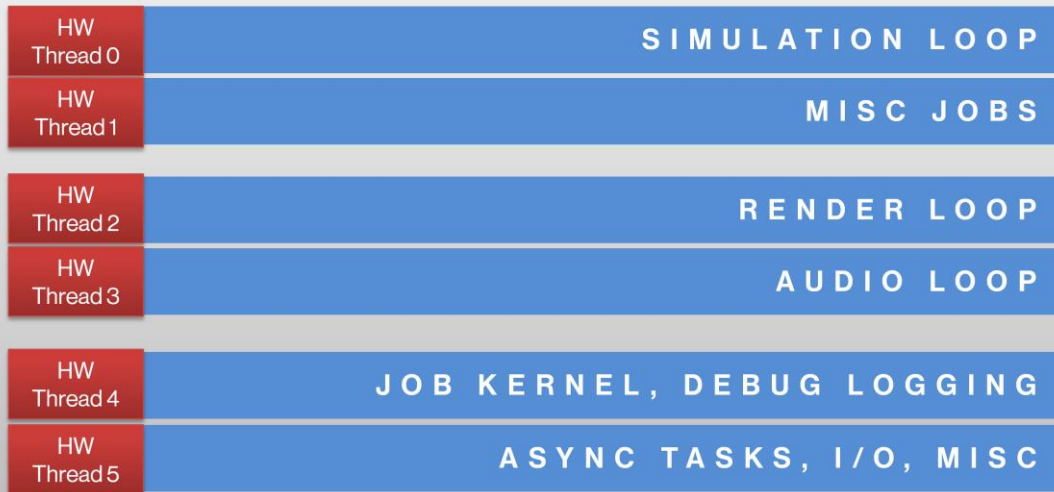


This pipeline maps well to system-on-a-thread design

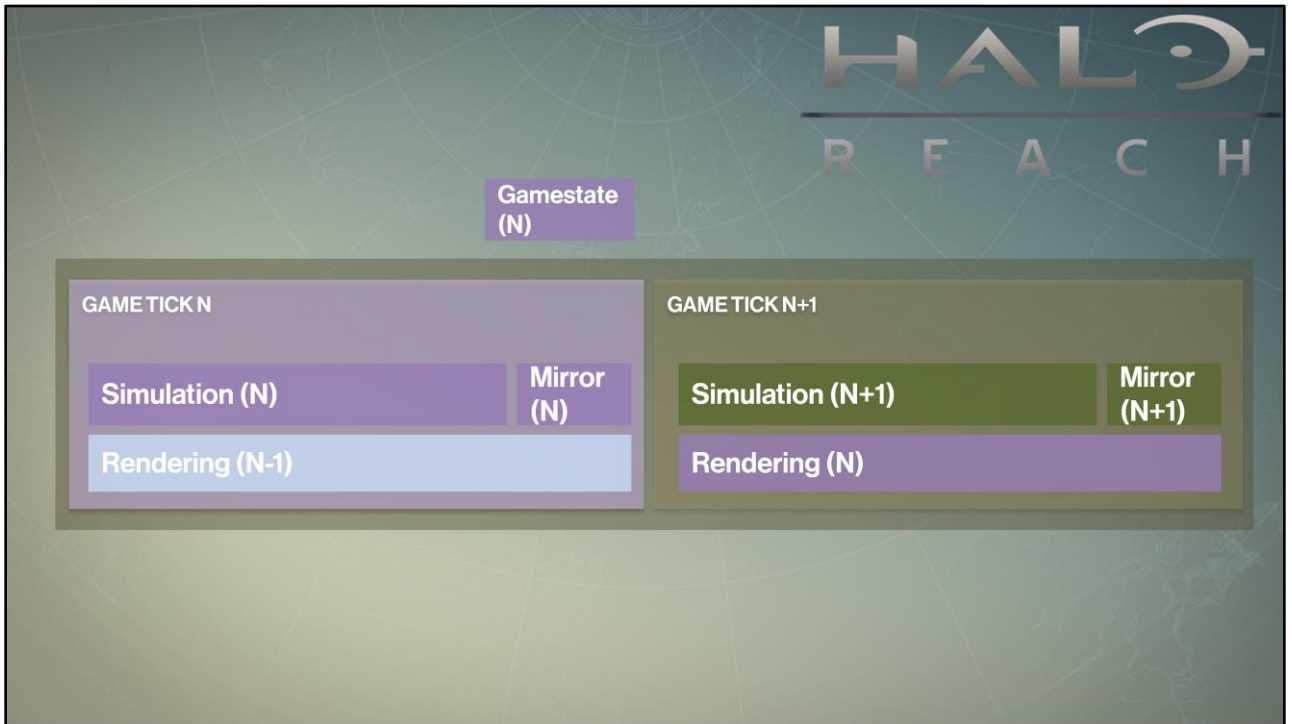
DESTINY 

This pipeline maps pretty well to what's called "system on a thread" type of parallelism. We shipped with that approach in our Halo engine.

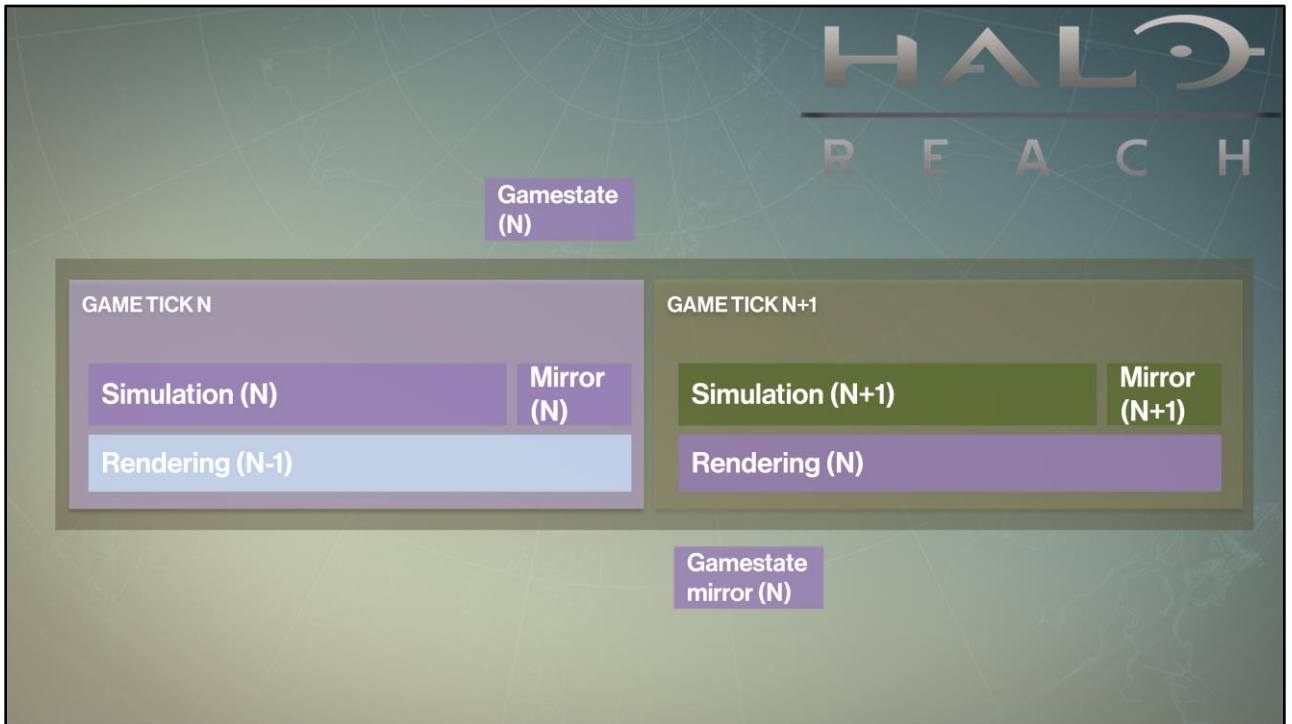
System-on-A-Thread Design



In Halo: Reach we mapped major game functions to a small number of threads, for example, unique threads for rendering, audio, simulation, where each CPU thread was also mapped to a HW thread.

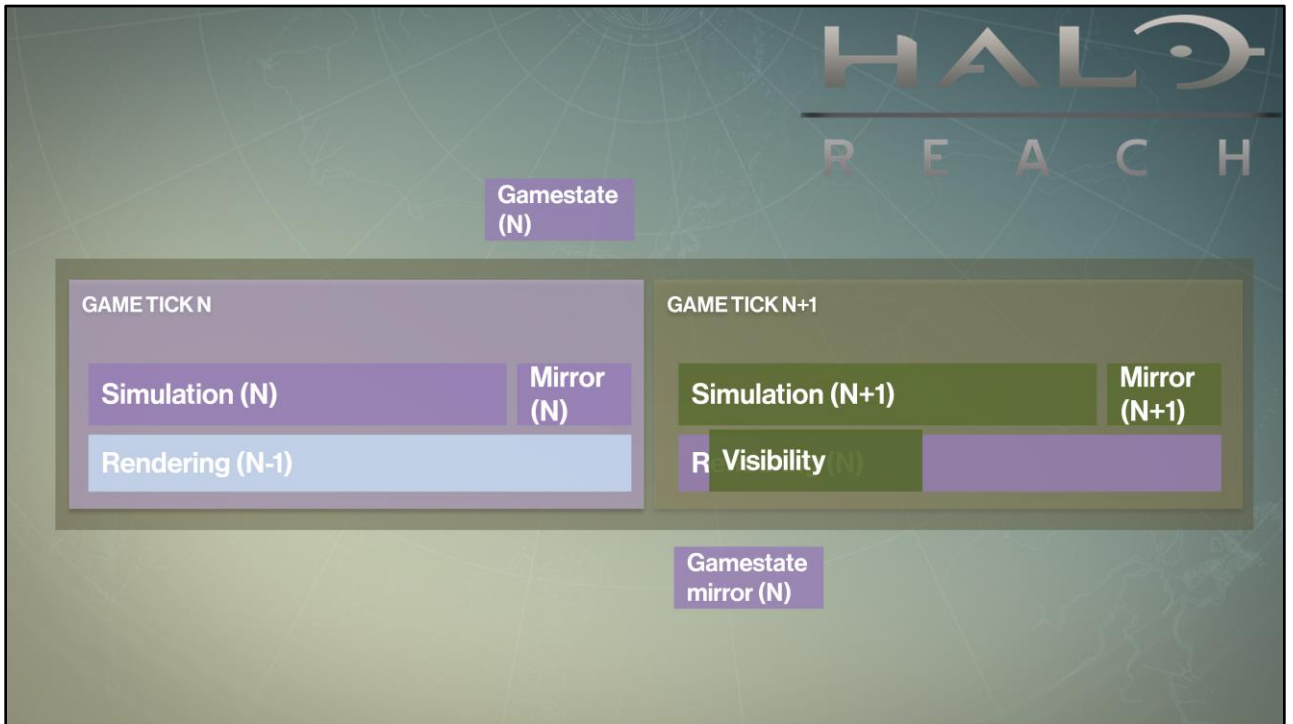


A different way to view execution is by looking at frame tick diagram. Here you see that as we go through Game Tick N, simulation computes <game> tick for frame N, while we are <rendering> the previous frame.



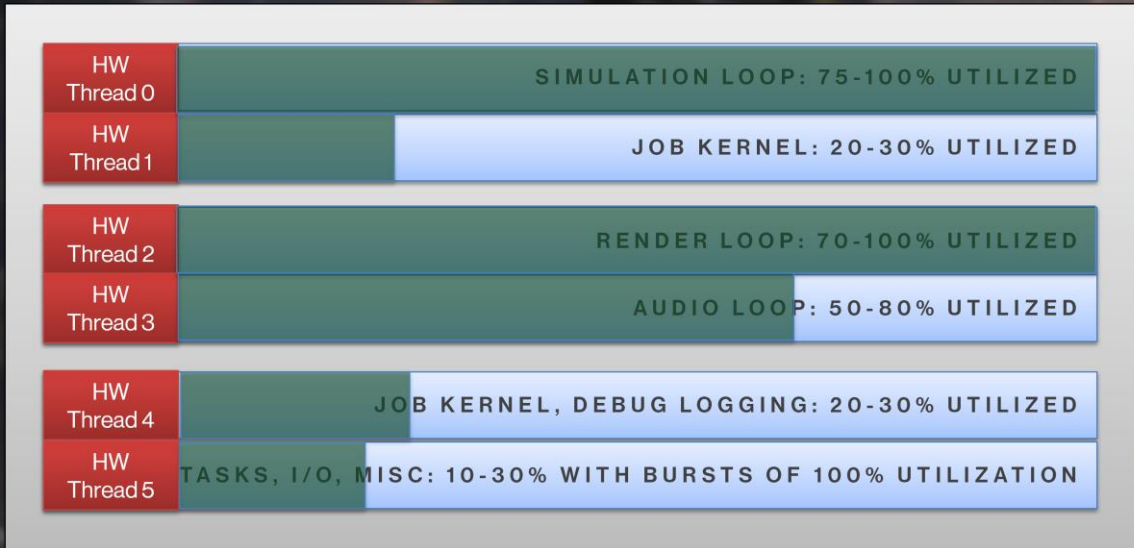
Once the simulation has finished <we copy> the gamestate to start <rendering> this frame during the next gametick.

In Halo games, all output systems' processing began only after simulation thread finished its work for current game tick and we copied the full gamestate.



We ran <serialized> visibility computation at the start of each render loop using the full gamestate copy. Until visibility is done, we can't start generating drawcalls.

System-on-A-Thread Load Balancing



Static per-thread load balancing meant suboptimal workload distribution.

We tended to see heavy utilization on the threads for simulation and render loops (our largest workloads) but the other threads saw plenty of idle time.

S-o-T Observations: Cons

- Difficult to adopt across generations / platforms
 - Does not scale for heterogeneous platforms

DESTINY 

This approach also does not scale well for systems with different core and thread layout, or heterogeneous computing systems (such as PS3). We also wouldn't be able to easily take advantage of additional cores we would have on the latest generation of hardware with this design.

S-o-T Observations: Cons

- Difficult to adopt across generations / platforms
 - Does not scale for heterogeneous platforms
- Synchronization required full double-buffer of game state

DESTINY 

We pay for a full extra copy of the entire game state which includes a fair amount of data that rendering does not care about (physics state, for example, or animation state machines).

S-o-T Observations: Cons

- Difficult to adopt across generations / platforms
 - Does not scale for heterogeneous platforms
- Synchronization required full double-buffer of game state
- Serialized up-front heavy visibility cost
 - Potential GPU idle bubbles

DESTINY 

Serialized visibility computation could mean GPU idle bubbles and potentially longer latency.

S-o-T Observations: Pros

- Convenient data access
- Extensible
- Easy

DESTINY 

But this design is easy to use, extensible and convenient to code for.

S-o-T Observations: Pros

- Convenient data access
- Extensible
- Easy
- Simple threading model

DESTINY 

Nor does it exhibit complex concurrency issues because the threading model is pretty straightforward.

S-o-T Observations: Pros

- Convenient data access
- Extensible
- Easy
- Simple threading model
- Pipelined concurrent execution of simulation and rendering

DESTINY 

Since we overlap simulation and rendering, we could do more work in each phase. When you have complex AI and physics computation and the simulation could take up the entire frame's ms, and heavy rendering workloads, this pipelining is important.

Outline

- Coarse-grained parallelism
- **Destiny renderer goals**
- Decoupling simulation and rendering
- Jobification for core workloads
- Data-driven render submission
- Advanced optimizations
- Conclusions



With all of this in mind, what were the goals we set out to hit for our Destiny engine?

Destiny Renderer Goals

- **Ship a game with great visuals and responsive gameplay**
 - Complex, alive, beautiful worlds
 - Large destinations with diverse environments
 - High-quality lighting, dynamic time of day, real-time shadows, weather elements, high resolution rendering
 - Many graphics features

DESTINY 

Our most important goal was to ship a fun game with great visuals and responsive gameplay.

Destiny worlds are complex, alive and beautiful. Destiny players explore large destinations, with diverse environments, lush vegetation. All of this required a renderer with high-quality lighting with dynamic time of day and real-time shadows, high-resolution rendering, weather elements such as rain, snow, dynamic wind. To make that happen, we implemented variety of complex graphics features.

Destiny Renderer Goals

- Ship a game with great visuals and responsive gameplay
 - Proven renderer architecture
 - 17 million players across 4 console platforms

DESTINY 

Our renderer architecture has been proven by the best test there is – the players. Destiny shipped last year and more than 17 million players have experienced the world of Destiny across four different console platforms.

Destiny Renderer Goals

- Ship a game with great visuals and responsive gameplay
- **Keep it scalable**
 - Cross-generation and cross-platform
 - Rock-solid performance across all platforms

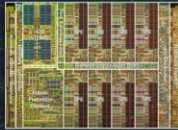
DESTINY 

We needed an engine with rock-solid performance on all of our platforms, but that also would be scalable across several console generations (and ideally beyond what's currently available).

Cross-Platform and Cross-Generation

Last Generation

- *PlayStation 3*
 - 2 HW PPU threads
 - ~6 SPU threads
- *Xbox 360*
 - 6 HW CPU threads
 - 3 3.2 GHz IBM PowerPC processors

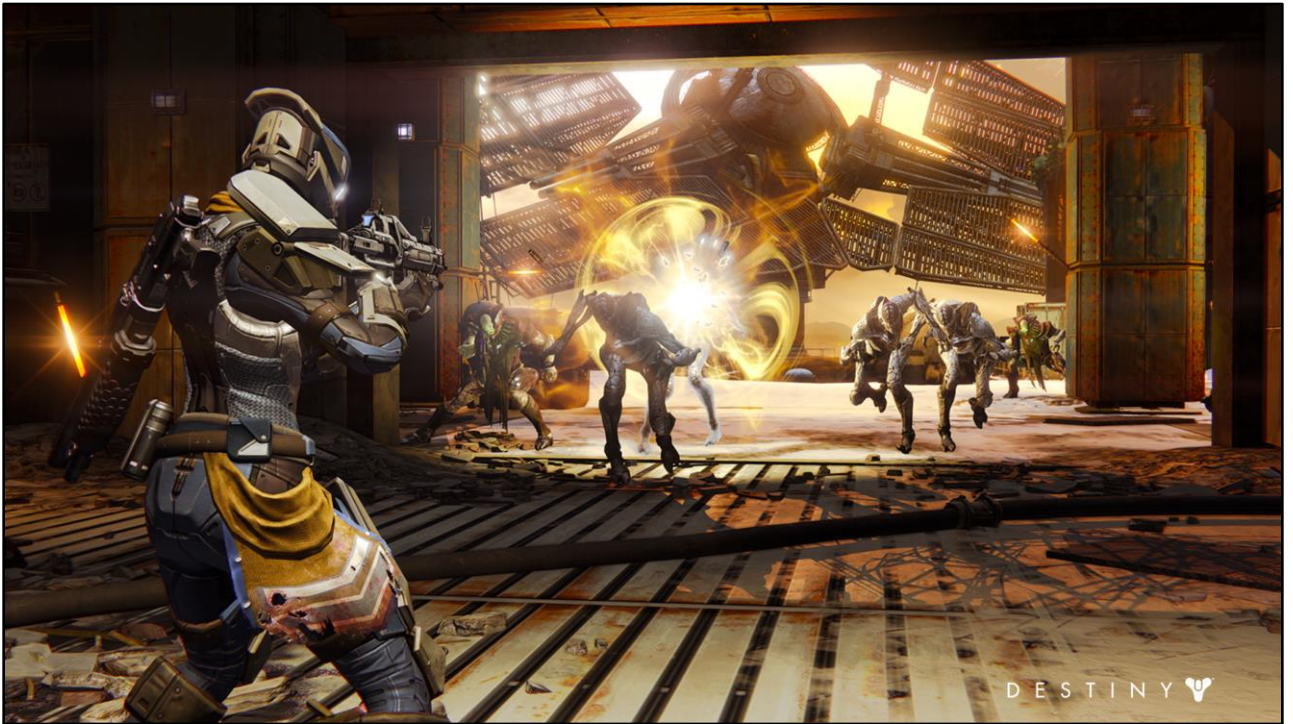


Current Generation

- *PlayStation 4*
 - 2 clusters of 4 x64 Cores
 - 1 HW thread per core
- *Xbox One*
 - 2 clusters of 4 x64 Cores
 - 1 HW thread per core

DESTINY 

We see quite a difference in the makeup of last versus current generation consoles CPUs. And GPU designs are even more varied across these generations. This introduced a fair amount of challenges that our architecture had to cope with.



At the same time, Destiny is a fun, fast-paced sandbox game with highly-responsive gameplay, requiring consistently low input latency - from the time the player pressed a controller button to the time they see the result of their desired action on the screen.

Sandbox == High Workload Variability



But any sandbox game can exhibit high workload variability and bursty workloads on both CPU and GPU.

Frames with little CPU simulation workload but heavy CPU and GPU rendering workloads (like this one) could mix with

Sandbox == High Workload Variability



shots where both GPU and CPU workloads are fairly light such as shot of the player on their sparrow flying through a Mars valley

Sandbox == High Workload Variability



Or, during a big battle scene, both CPU and GPU are brought to their knees and our engine and renderer must keep up which required very efficient load balancing.

Keeping It Efficient

- Maximal CPU and GPU occupancy
 - Keep them occupied fully
 - Avoid GPU idle
- Dynamic load-balancing and smart job batching
- Keep latency low

DESTINY 

To ship on last generation consoles without sacrificing our game play meant that we had to maximize every ms available on every available computational unit.

To do that, we moved everything on the rendering and visibility workloads into tasks, used dynamic load balancing with smart job batching to for best CPU and GPU occupancy to keep our latency low.

We achieved this by fully parallelized execution for best load-balancing, batching jobs smartly for both CPU and GPU and flushing GPU command buffer effectively for optimal GPU saturation, as well as keeping GPU idle to a minimum to avoid GPU bubbles.

Everything on the rendering and visibility workloads was moved into tasks.

Destiny Renderer Goals

- Ship a game with great visuals and responsive gameplay
- Keep it scalable
- Keep it efficient
- **Keep it simple**
 - “How I learned to stop worrying and love multithreading”
 - From 0 to render jobs in < 10 min
 - New features → little or no changes to existing jobs

DESTINY 

A major goal of the rendering architecture was to keep the API simple to use and let graphics engineers do their job and not worry about threading. We had a lot of features to write and we wanted to create new features quickly and automatically jobify them.

The inherent complexity of multi-threading should be encapsulated within the underlying core architecture. By providing the right abstraction layers for common and advanced graphics operations, by focusing on local and coherent data encapsulation we could automatically enforce synchronization and correct data access. This allowed us to hide the vast majority of the threading complexity and write a ton of features!.

Destiny Renderer Goals

- Ship a game with great visuals and responsive gameplay
- Keep it scalable
- Keep it efficient
- Keep it simple
- **Decouple simulation from rendering**

DESTINY 

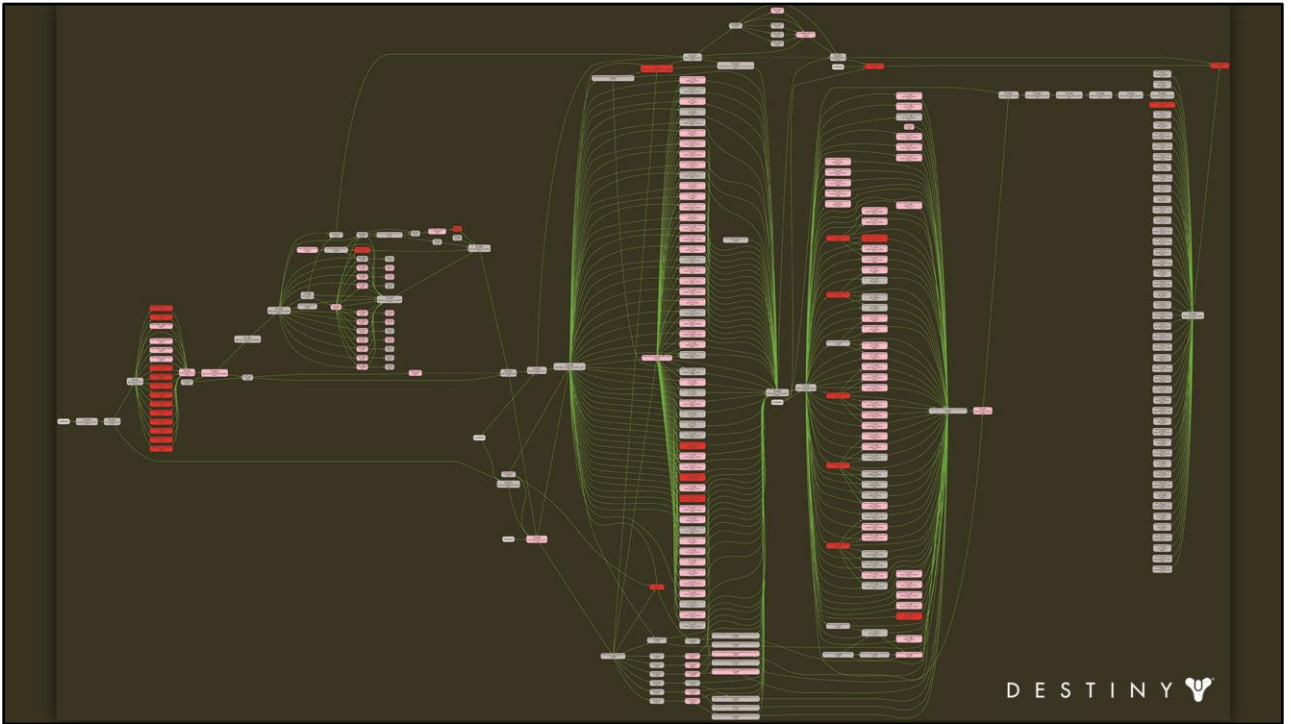
Decoupling game-state traversal from rendering would allow us to improve latency and help make the rendering submission a data-driven streamlined kernel processor.

Destiny Renderer Goals

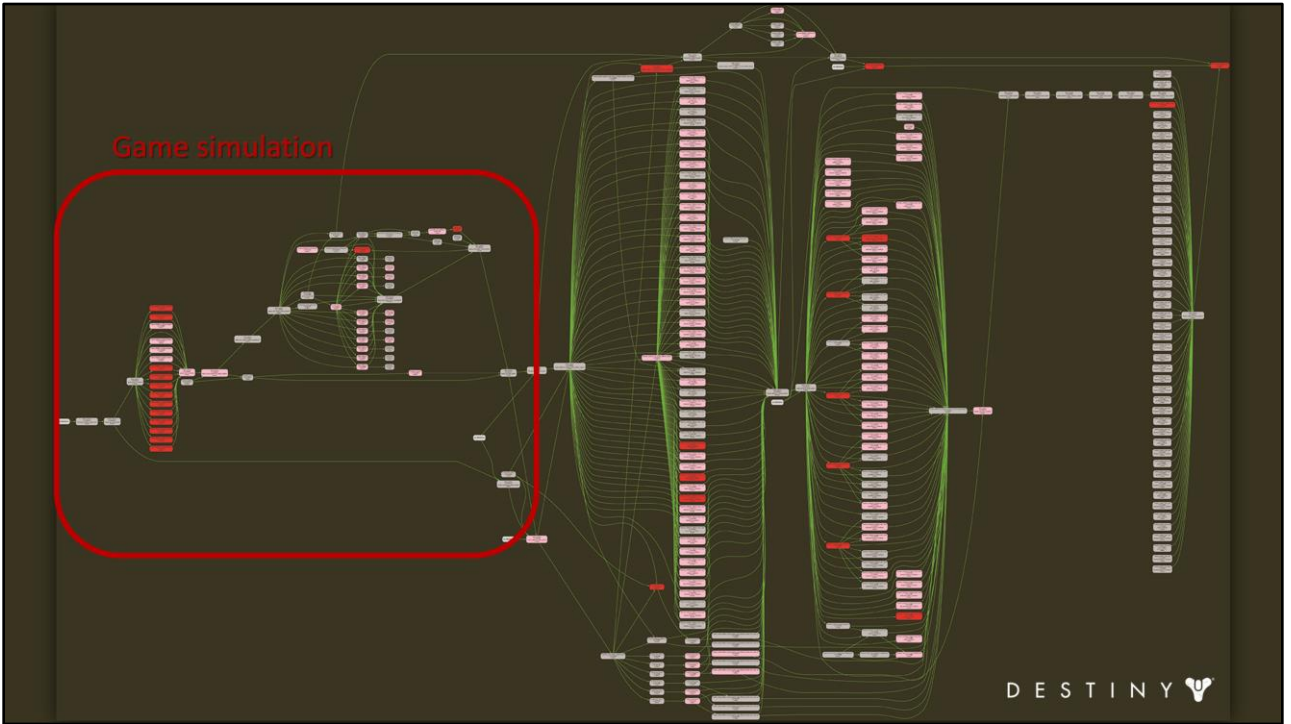
- Ship a game with great visuals and responsive gameplay
- Keep it scalable
- Keep it efficient
- Keep it simple
- Decouple simulation from rendering
- **Fully data-driven rendering pipeline**

DESTINY 

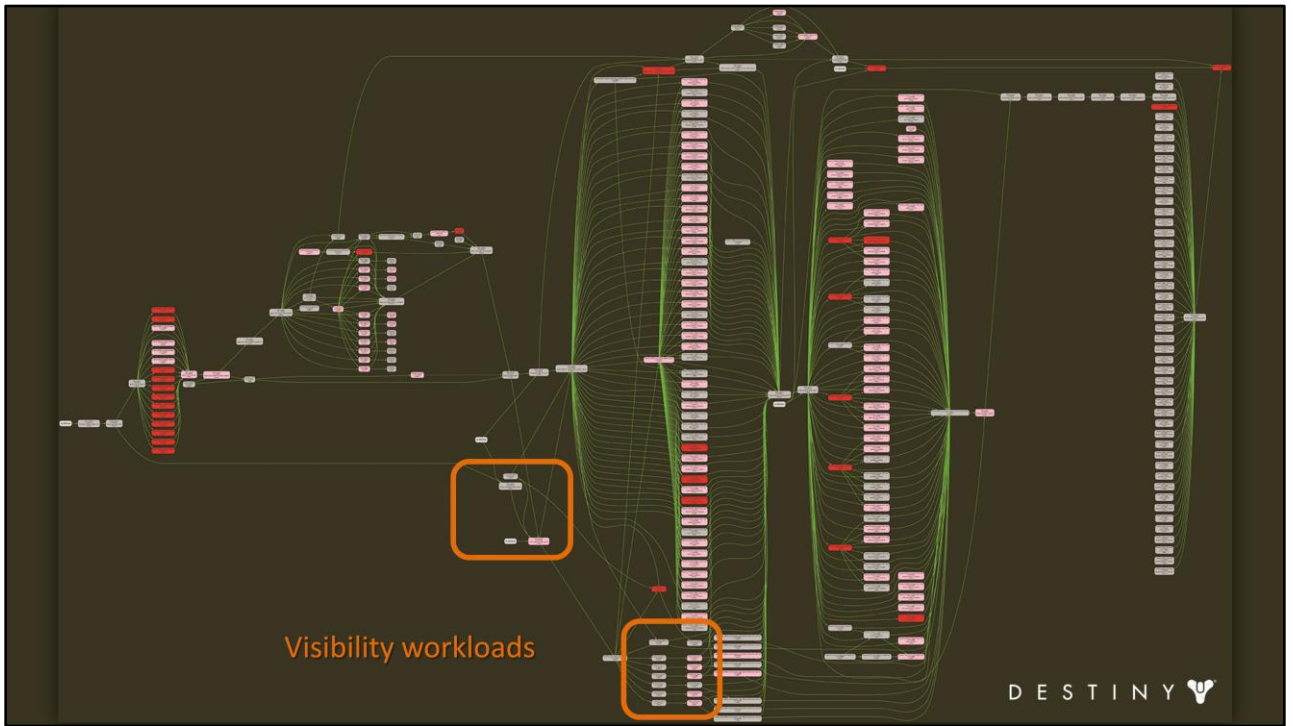
We needed a fully data-driven rendering pipeline, where the rendering passes are executed via jobs operating on individual render elements arranged in coherent caches rather than directly on game objects.



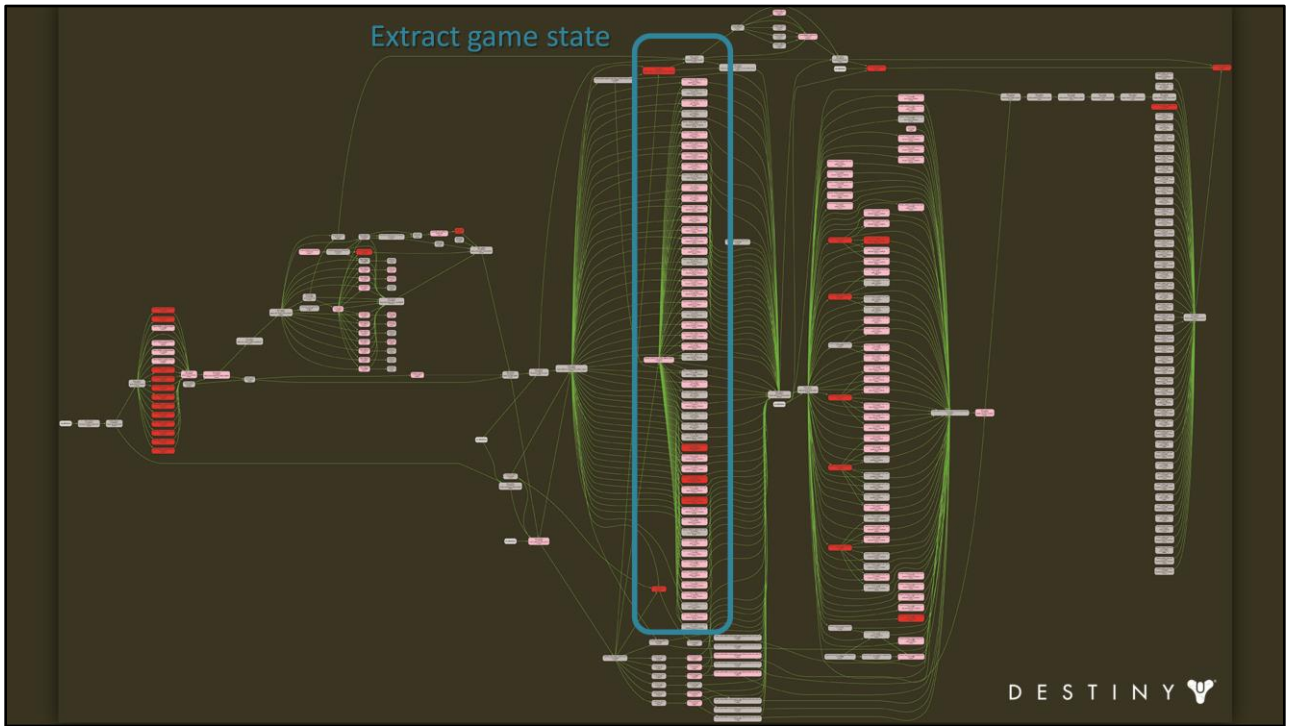
And when it's put together, the execution of our renderer look something like <this>
Each node in this diagram is an actual job (this is a shipping frame from our
Cosmodrome level on Xbox One)



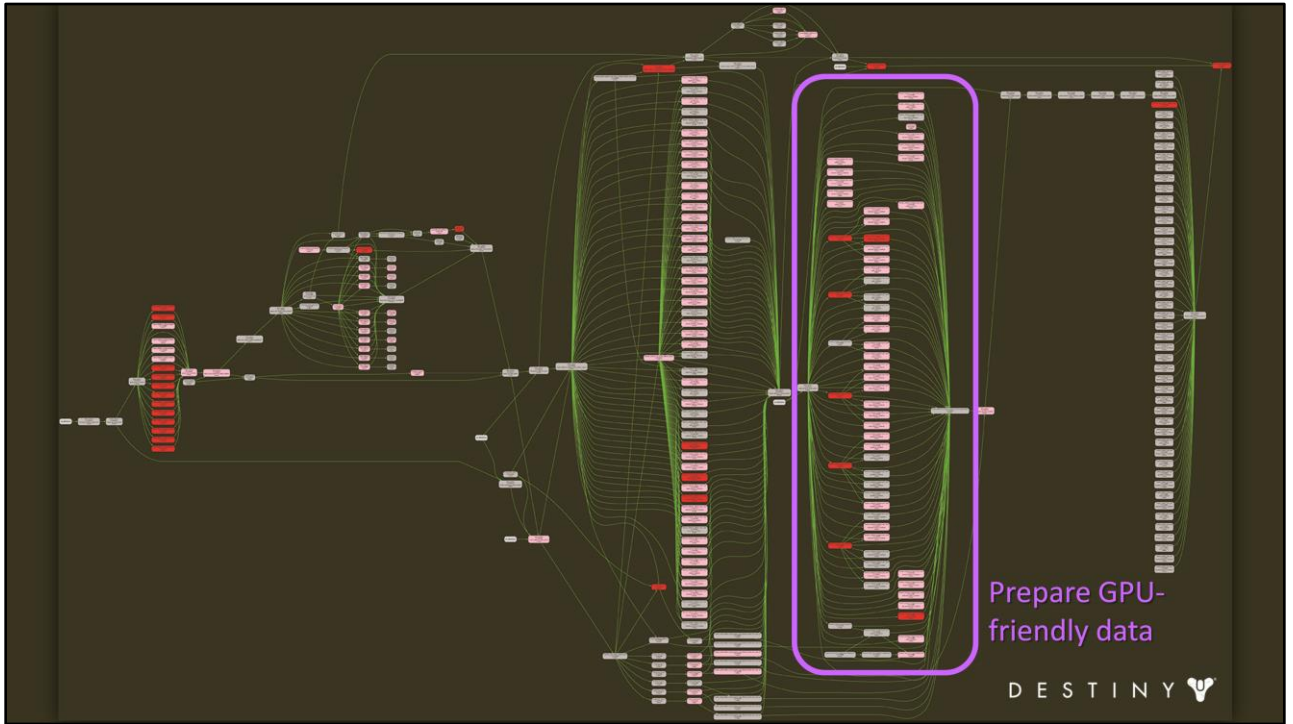
We'll start by simulating the game tick (multi-threaded where possible)



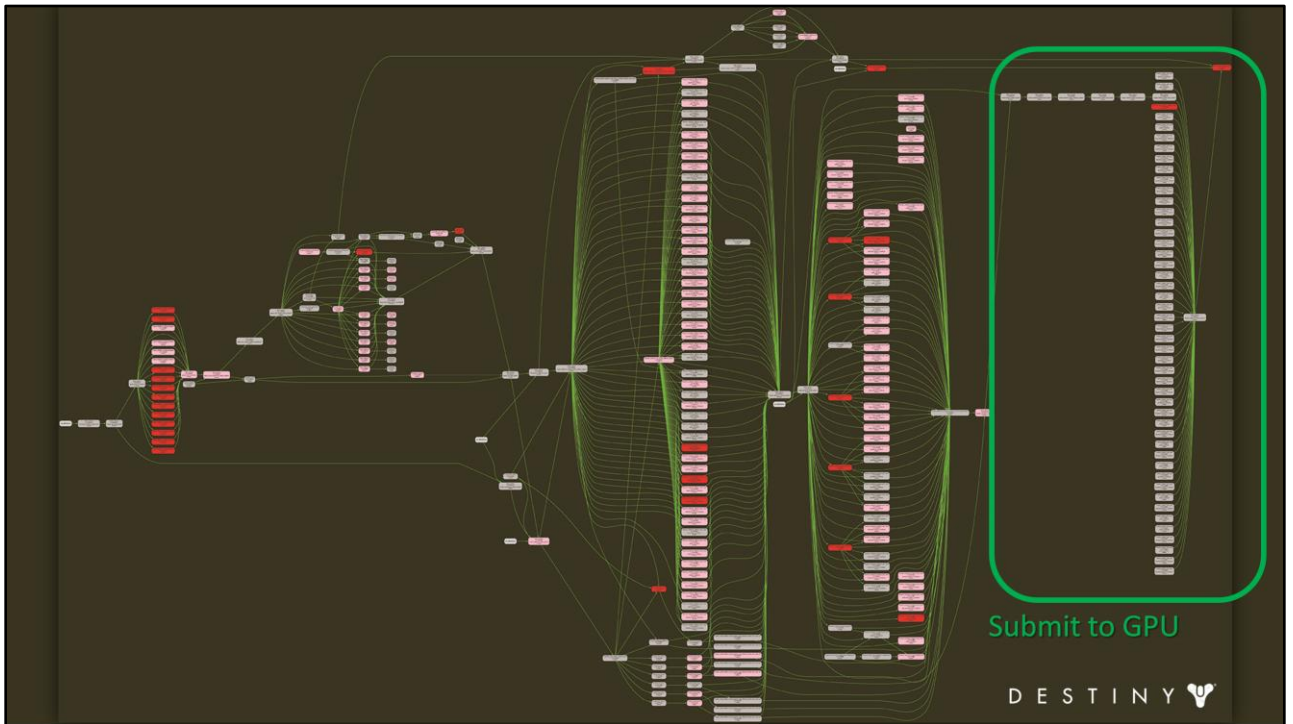
Then we will run our visibility workloads to determine what should be rendered in this frame. We used Umbra visibility in the Destiny engine – and last year at GDC there was an excellent talk covering a fair amount of details of the visibility



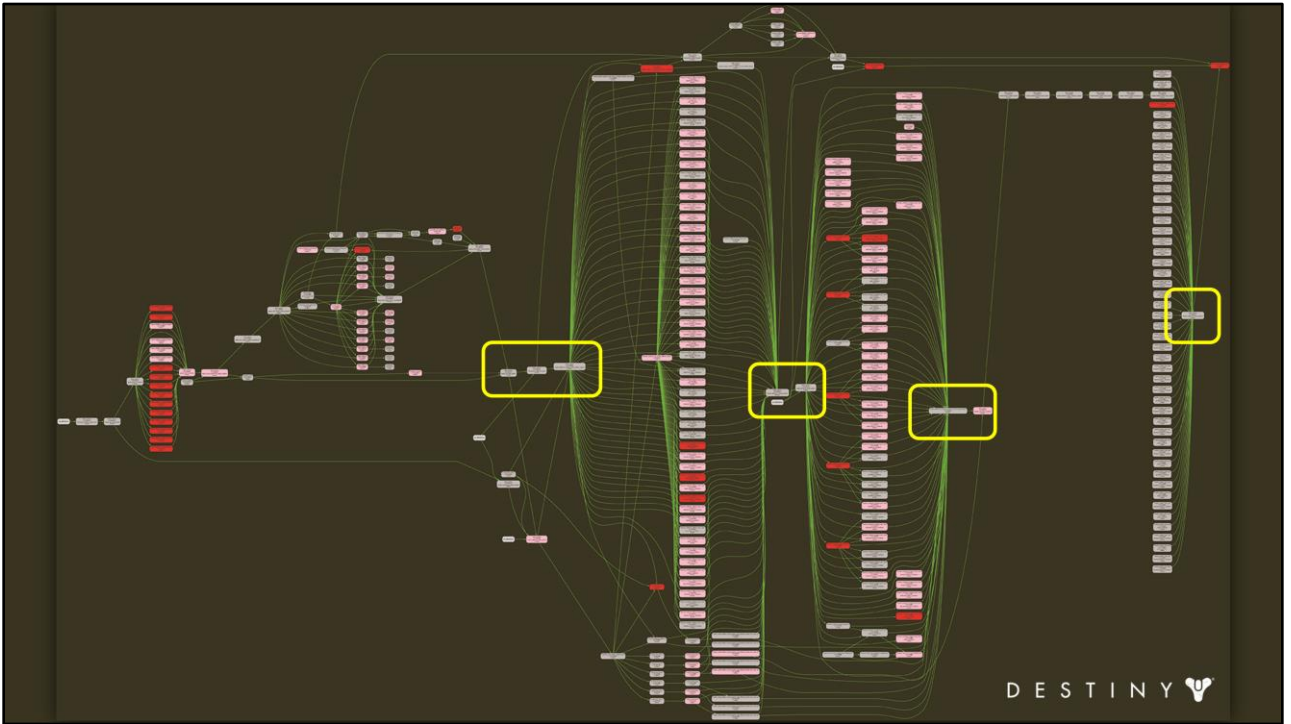
We'll extract data from the current game state for visible elements only into a dynamic data structure we call *frame packet* at every frame.



We'll then convert this data to GPU friendly format ready to be copied into GPU registers during the *render prepare* phase.

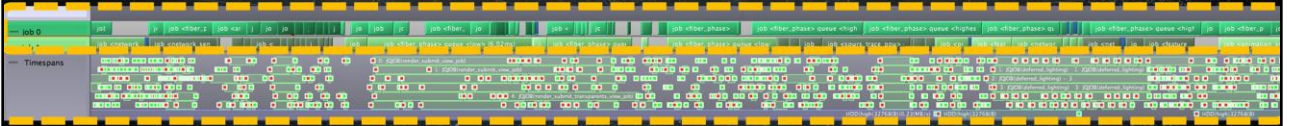


And generate drawcalls to submit to the GPU during the *render submit* phase. We go wide during this phase to reduce latency (by finishing the work faster) and to keep GPU fully saturated without any idle time bubbles.



Our architecture has well-defined synchronization points which are necessary to ensure safe and well-defined concurrent data access across each phase of execution.

In Practice...



DESTINY 

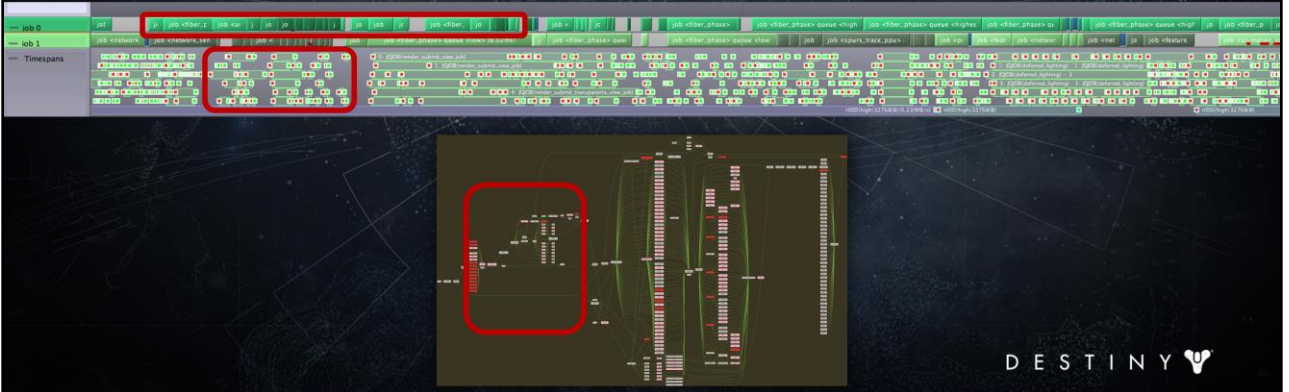
Here is an example from PS3 execution.

<Note> the distribution of jobs on PPU (these are the two PPU threads) – we had a few jobs running on PPU for rendering but mainly during extract and the high-level submit script jobs.

<And> the rest had pretty solid wall of SPU occupancy.

In Practice...

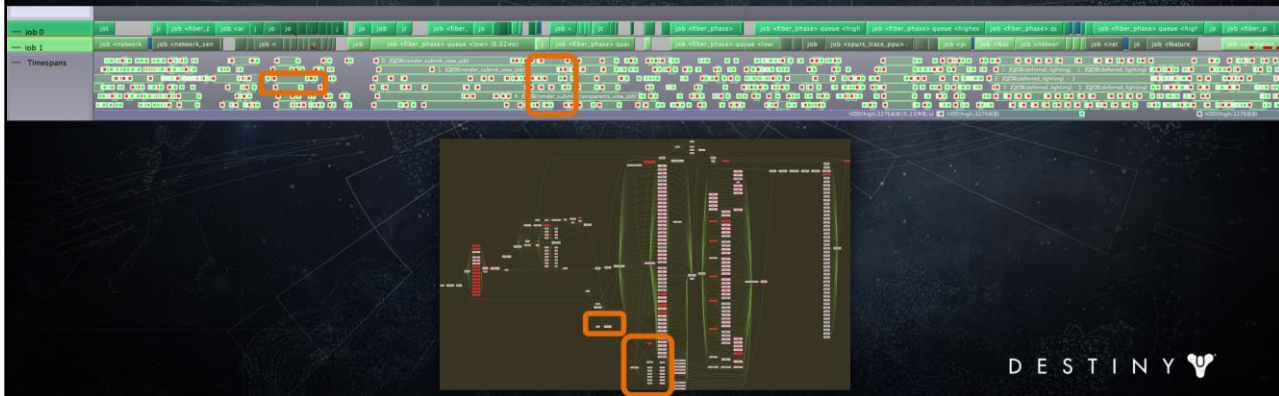
Game simulation



<The first> set of jobs that runs in our frame are simulation jobs.

In Practice...

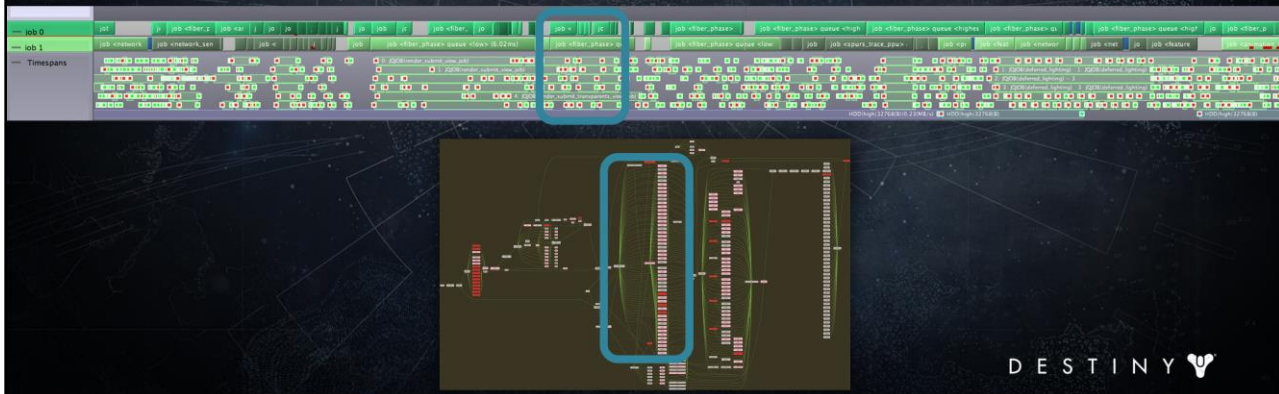
Visibility workloads



Next we have visibility workloads.

In Practice...

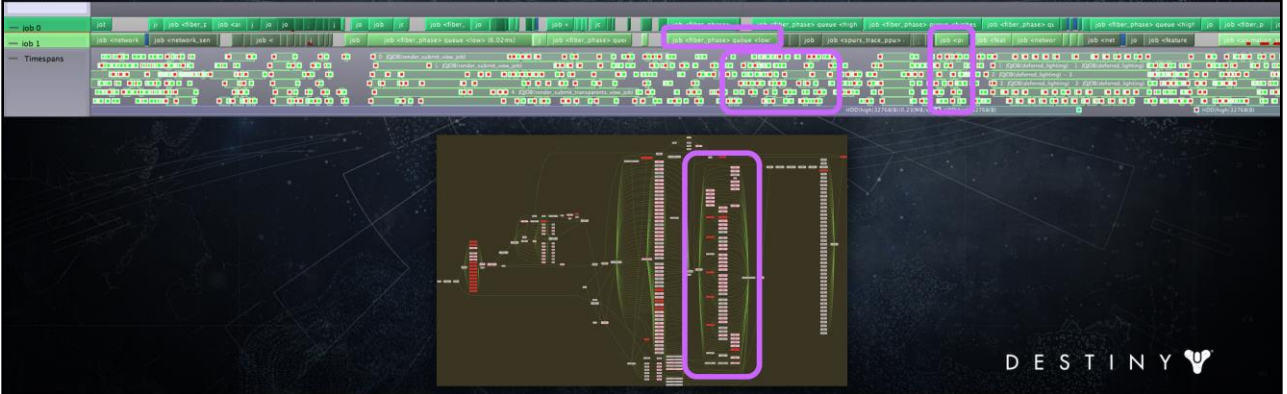
Extract Game State



We extract gamestate for visible elements

In Practice...

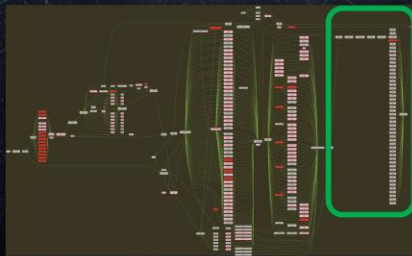
Prepare GPU-Friendly Data



Run render simulations and prepare GPU-friendly data

In Practice...

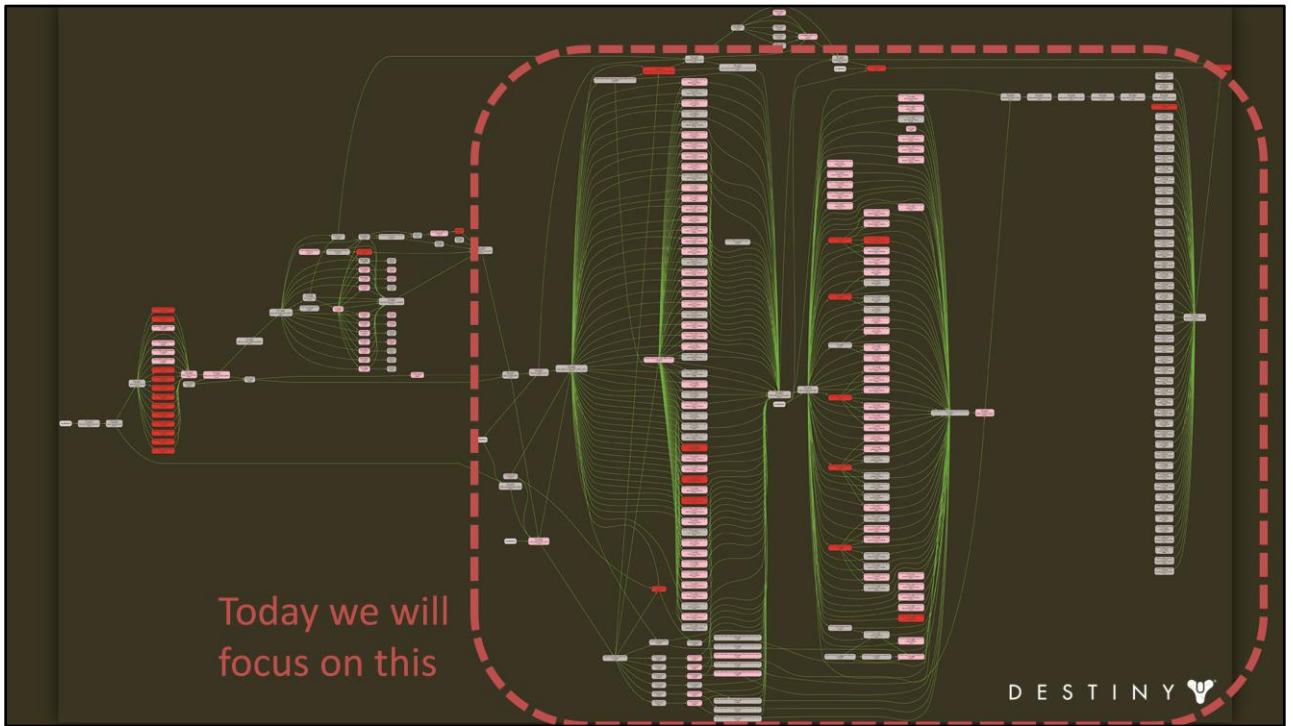
Submit Commands to the GPU



DESTINY 

And finally generate GPU commands to render the frame.

Note that we had a pretty solid wall of jobs to execute the frame.



Today, we will only focus on this part of our engine's job graph. While I won't have time to cover all aspects of our entire design, including getting into proper details for some of the more advanced topics, I hope that I will be able to give you a good taste of a system we have developed and give you ideas about how you could approach multi-threading graphics engines.

Outline

- Coarse-grained parallelism
- Destiny renderer goals
- **Decoupling simulation and rendering**
- Jobification for core workloads
- Data-driven render submission
- Advanced optimizations
- Conclusions

A screenshot from the video game Destiny, showing a character in a blue and white suit in a dark, industrial environment. The text "Decouple Game State Traversal from Rendering" is overlaid in yellow. The Destiny logo is in the bottom right corner.

Decouple Game State Traversal from Rendering

So how did we decouple game state traversal from rendering?

Conscious Uncoupling

- Decouple visibility and game objects
 - Visibility traversal independent of game objects
- Drive render workloads from visibility results
 - Decouple render and game objects

DESTINY 

We drive all of our render workloads by results of visibility to decouple game objects and rendering. We decouple visibility objects and game objects (except for a thin interface between). With that, we don't need to traverse game data to determine what's visible.

Put That Static Data Away

- Simple observations:
 - Rendering data != game object data
 - Bulk of render data is static over object's lifespan
- Cache immutable render data in the renderer
 - On registration
 - Read-only post registration

DESTINY 

What we store in a game object is not necessarily what we need for rendering. A typical game object has a ton of data we don't need for rendering (AI, pathfinding, physics). So let's cache only the data we need in the renderer (maybe its mesh and material references, some dynamic runtime state (where it is in the world, what are its skinning transforms if it's animated, etc.)). Most of that data is static while the game object is in memory and we can limit the access to be read-only.

A typical game object has AI, physics, animation components. But we need a small subset of that data to render that object – maybe its mesh and material references, some dynamic runtime state (where it is in the world, what are its skinning transforms if it's animated, etc.).

For most game objects much of rendering data is static over its lifespan. For example, if we have environment scenery, we have their mesh references, their materials and shaders handles, their transforms – all of that data never changes until we unload that environment instance from memory.

We cache this static render data in the renderer when registering a game object w/ the renderer. Then since this data never changes, its access is always read-only.

Carry On Only!

- Extract per-frame *transient* render-specific data
 - Static data is already cached
 - Extract only dynamic data - each frame

DESTINY 

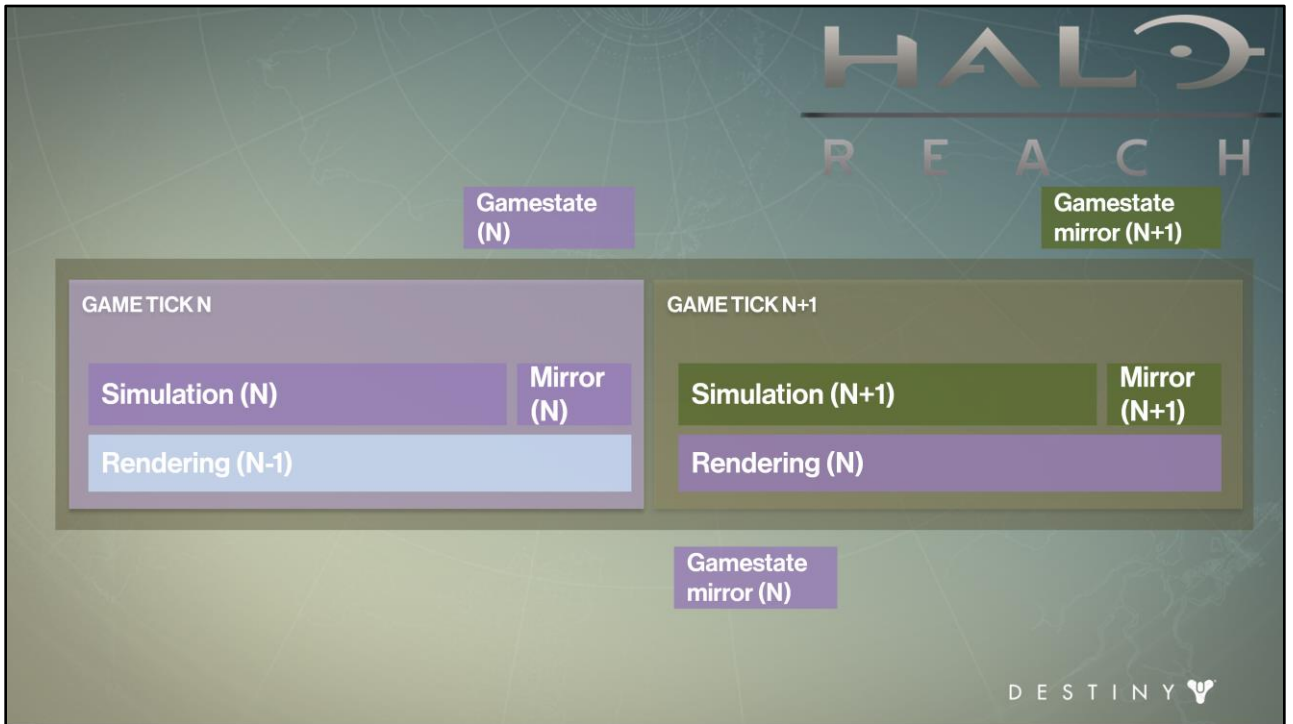
For dynamically generated data we also don't need the full object. For example, in case of a character we only want it's world transform, its skinning transforms, and maybe some small additional state about that object ("am I the player?" "am I damaged?", etc.) for each frame. We already cached this character's mesh and material references in a static cache, so we only need to extract this dynamic data – and only for the frame when it was generated.

Carry On Only!

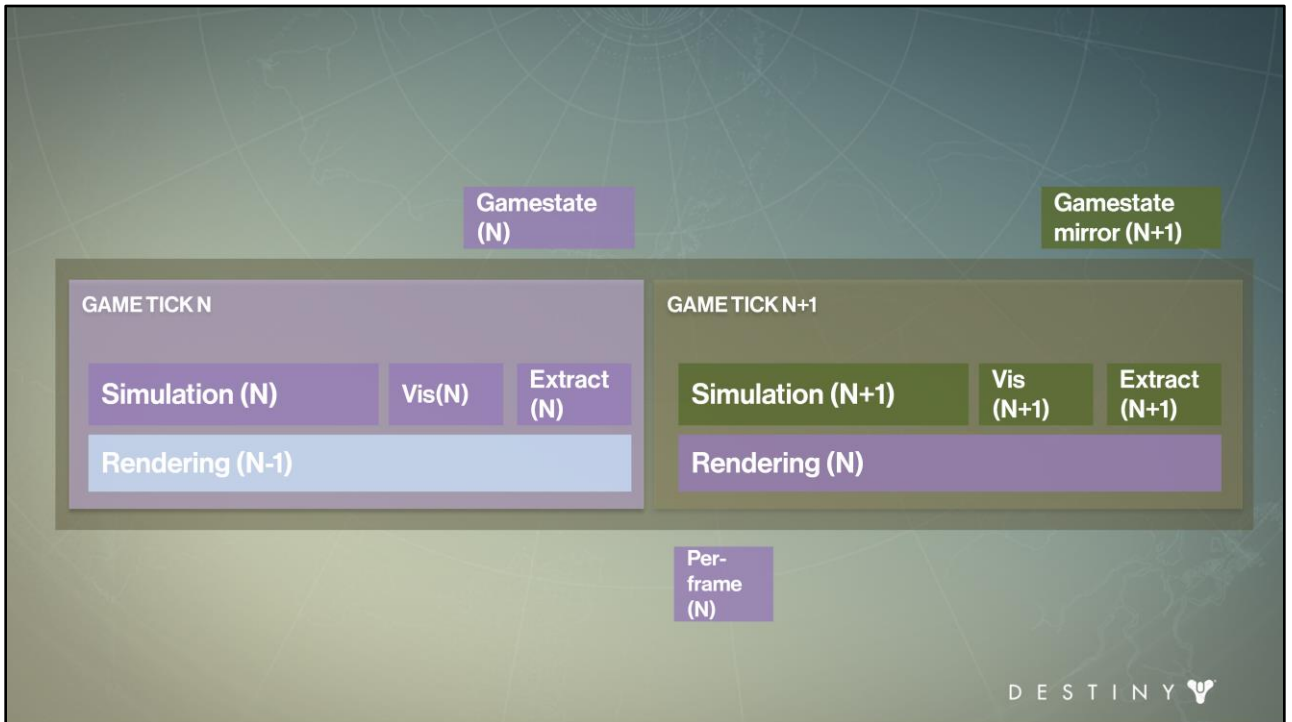
- Extract per-frame *transient* render-specific data
 - Static data is already cached
 - Extract only dynamic data - each frame
- **Visibility results define what gamestate to extract**
 - *Extract data for visible elements only*

DESTINY 

Also, we only care to grab data for what we will actually render. If an object is not visible, don't extract anything. This lets us save the amount of data we need to double-buffer, and reduces the amount of data we need to copy out of game state. Thus, visibility drives *what* data needs to be extracted.



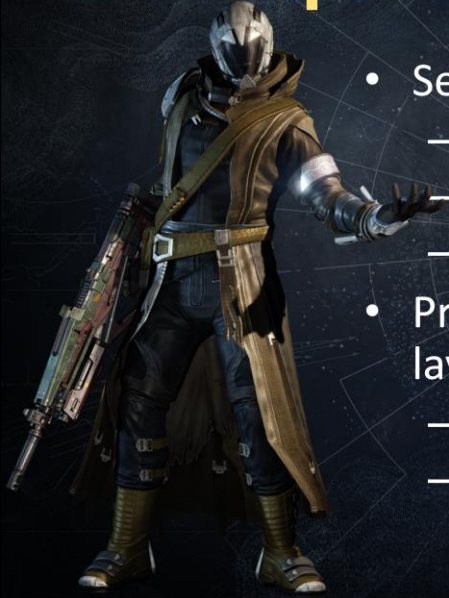
Let's bring back the Halo engine diagram and modify it for this approach.



So in this world we have <simulation>, followed by <visibility>, followed by <extract>, which copies out a <much smaller> per-frame data out of gamestate which is then used by rendering this frame.

(Note that simulation time didn't reduce in this model, I just had to fit the words on the diagram for readability).

Decouple Objects and Rendering



- Separate representation in each system
 - Object system (game side)
 - Rendering (render objects)
 - Visibility (visibility objects)
- Provide interfaces to communicate cross layer
 - Strict access rules
 - Only during specific game phases

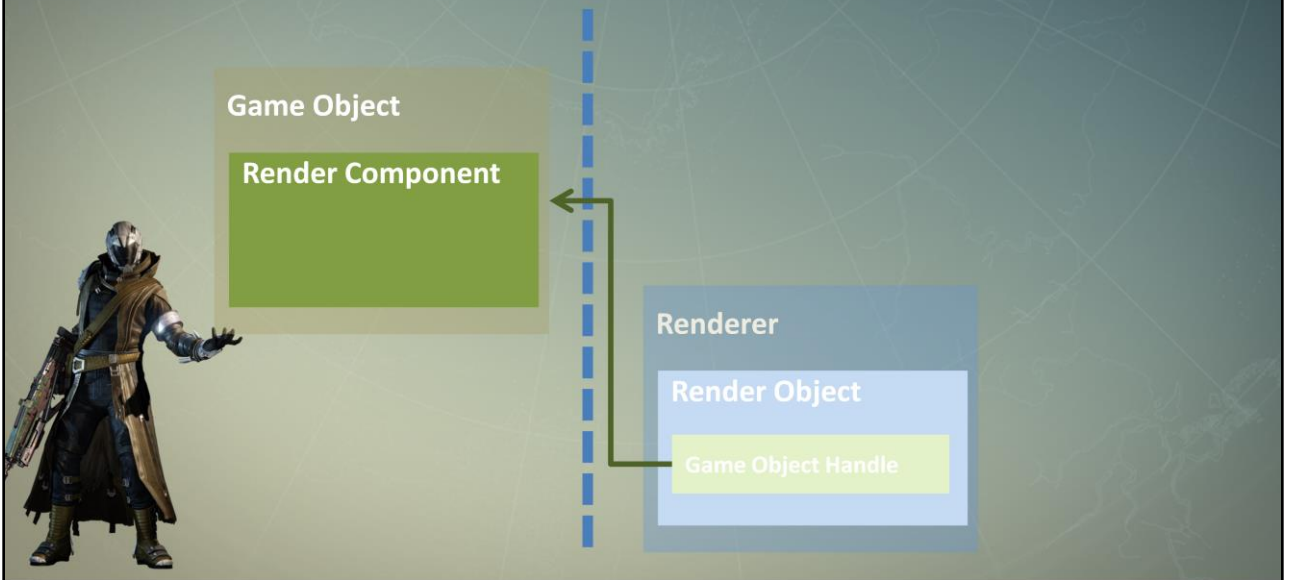
DESTINY 

To decouple game systems and visibility and rendering, we first separate data representations into three separate categories:

- Object system (this is our components on the game side)
- Render objects
- Visibility objects

We provide interfaces for each layer to communicate across the boundaries. And we enforce strict access rules to allow access only during specific phases to guarantee correct threading synchronization.

Data Flow

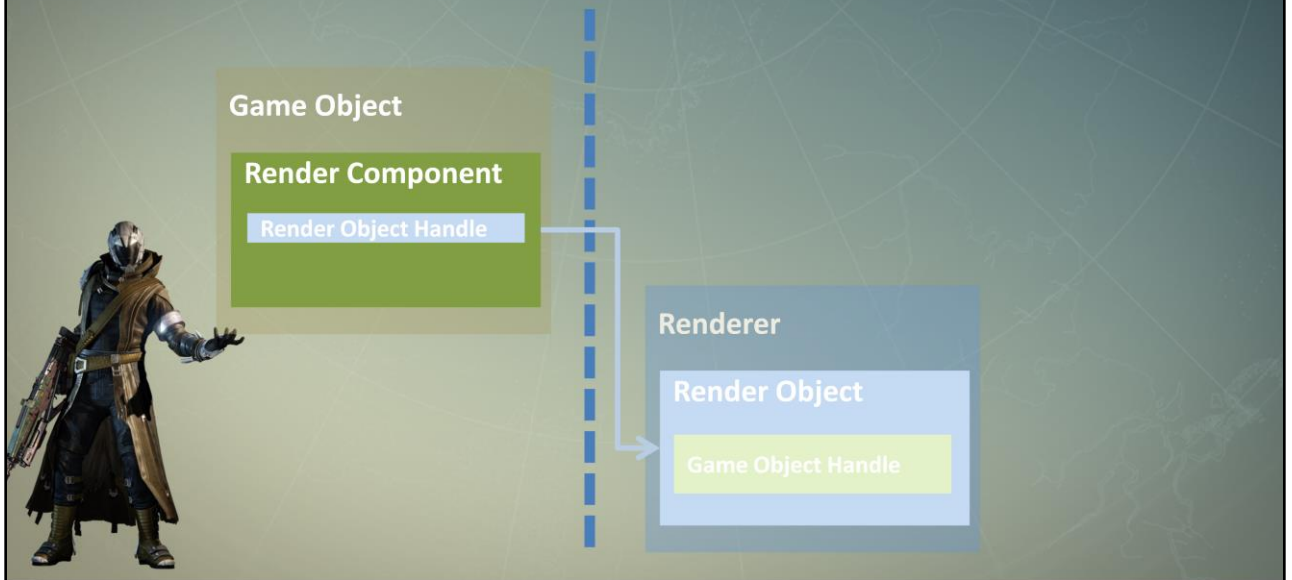


We start with a game object. <Each> component will map to a particular render object on the renderer side.

<When> a new game object is added to the world, it registers itself with the renderer. This caches the static render data for that game object inside the renderer.

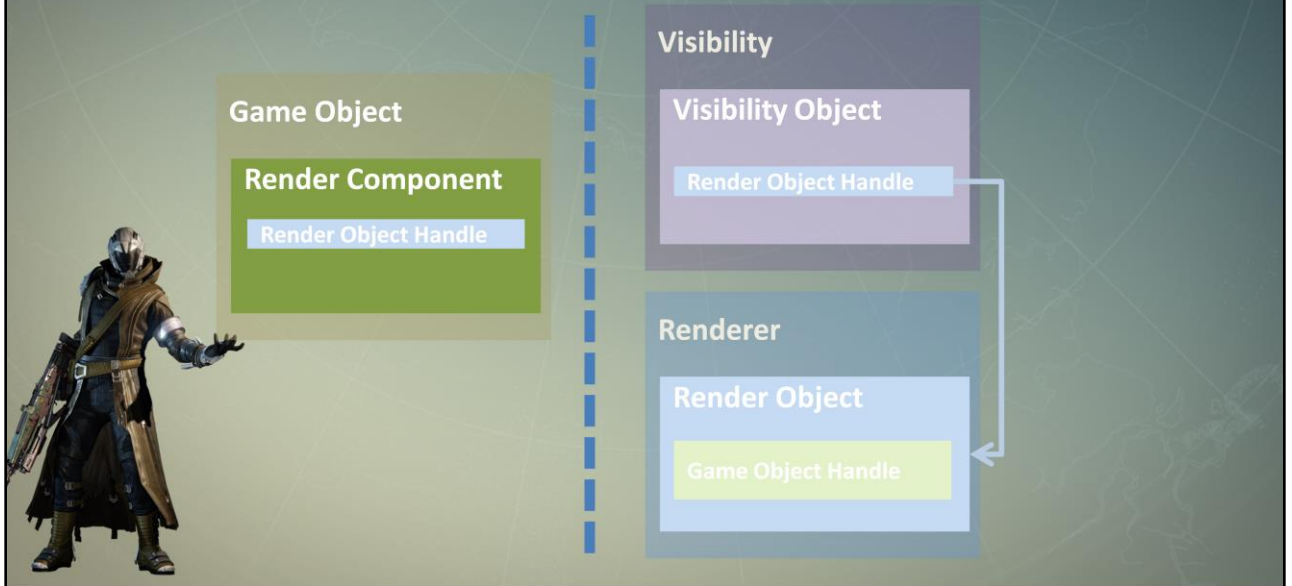
<The render object> may cache a game object handle (for dynamic objects) for later use.

Data Flow



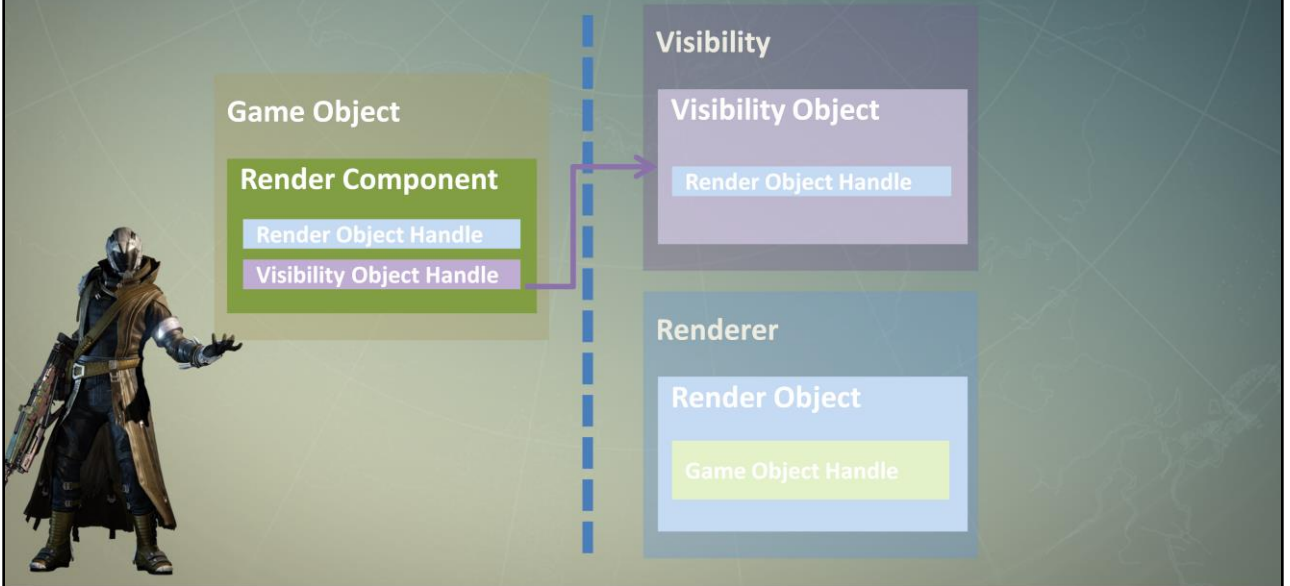
The renderer then returns a handle to the render object back to the game object which caches it in the corresponding render component.

Data Flow



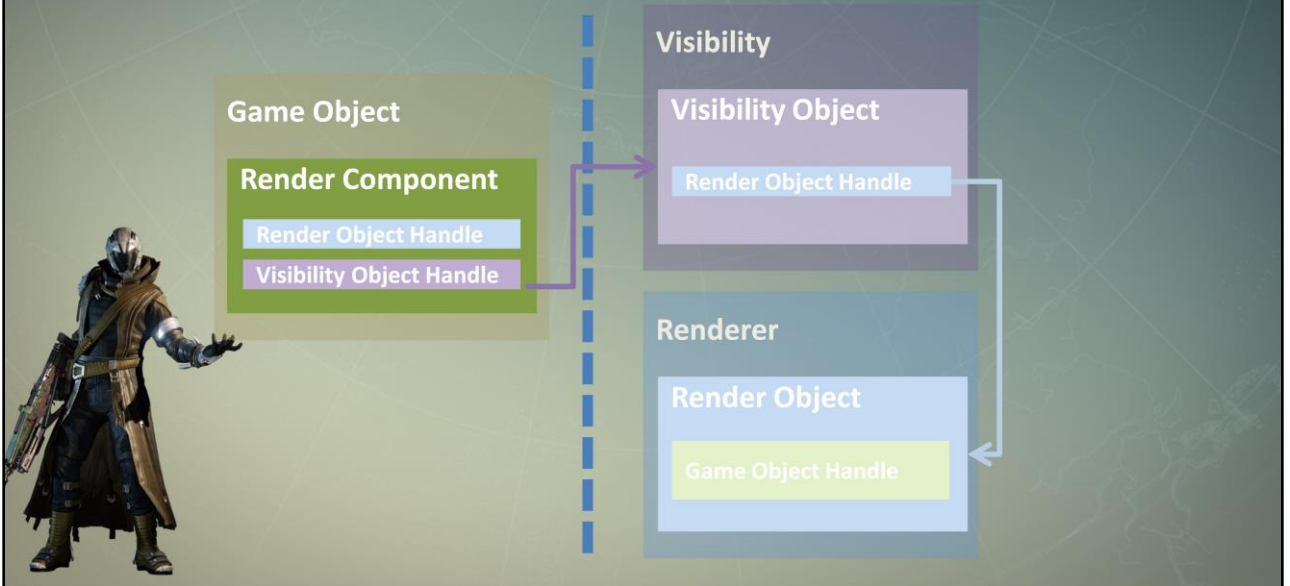
Using this render object handle, the game object registers with the visibility system to create a visibility object which caches render object handle inside. This render object handle points to the original render object

Data Flow



Which then returns a visibility object handle back to the object.
To hide the game object now, we then just have to unregister this object with the visibility, and it will stop rendering.

Data Flow



Note that render object does not know anything about visibility; only visibility <knows> about the render object

Thus we've decoupled game object system, visibility and render layers and can now drive rendering system from the results of visibility without having to access game objects.

What about the dynamic data extraction?

Frame Ring Buffer

Frame Packet

Frame Packet

Frame Packet

- Dynamic per-frame render data storage
 - All feature renderer dynamic data stored there
 - Fully stateless; thrown away after every frame
- Small total footprint
 - 1 MB per frame on PS3 | Xbox360
 - 9% of total gamestate

We store all dynamic data (extracted or generated) in a double-buffered frame packet ring buffer. This multi-thread-allocation-safe structure is fully stateless - it is generated each frame and thrown away after we submitted each frame.

<Each>frame's worth of data is referred to as a frame packet.

It has a fairly small footprint – each frame is ~1 MB per frame on last gen consoles which is about 9% of the total game state on those consoles. For comparison in Reach we copied ~10 MB of gamestate every frame to a mirror.

Outline

- Coarse-grained parallelism
- Destiny renderer goals
- Decoupling simulation and rendering
- **Jobification for core workloads**
- Data-driven render submission
- Advanced optimizations
- Conclusions

DESTINY 

Next, let's talk about how we generate jobs for the renderer workloads

Processing Views

- View == { frustum | camera }

DESTINY 

Our render pipeline operates on concept of views.
A view is defined by the usual frustum & camera parameters.

Processing Views

- View == { frustum | camera }

– Examples:



PLAYER VIEW



SHADOW VIEW



OVERHEAD MAP VIEW

DESTINY 

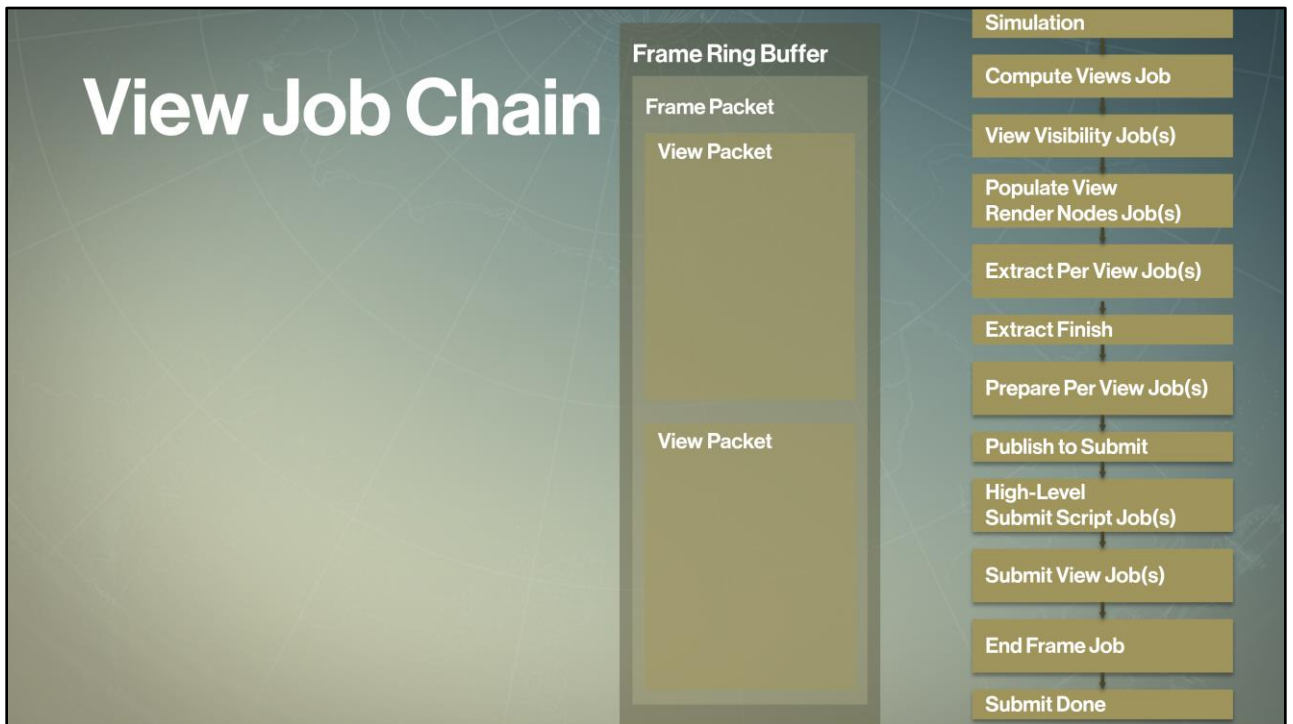
for example – we have <player view>, <shadow views>, <overhead> map views, etc.

Processing Views

- **View == { frustum | camera }**
 - Ex: Player, shadows, overhead view, etc.
- **View == render job chain unit**
 - Visibility .. Extract .. Prepare .. | Submit

DESTINY 

A view also defines end-to-end renderer job chain, so it becomes our jobification unit.



Let's take a look in a simplified form at how the architecture handles views to jobify the renderer workloads.

We run <game simulation>, then <determine the views> for current frame and <generate> the frame packet data structure for this frame, <figure out what's visible> in each view,, <populate> internal data structures for each view for efficient iteration in later jobs, <extract> game object data into the frame packet for each view's visible list, and <unblock simulation> for next game tick. Then we <prepare> GPU-friendly data and run render-specific simulation workloads, <tell> the core system we're ready to submit it to the GPU, <run> high level submit script job, <generate> drawcalls for each view, <present> the frame, and <tell> the core system that we've completed this frame's submit.

View Job Chain

- Core system jobs
- Always run
 - Independent of frame contents

Frame Ring Buffer

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done

We run a few <global core renderer system jobs> which are executed regardless of what data we have in our frame. These act as the main synchronization points for our system.

View Job Chain

- Data-driven view job chains
- Created dynamically
- Based on content in a given frame

Frame Ring Buffer

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

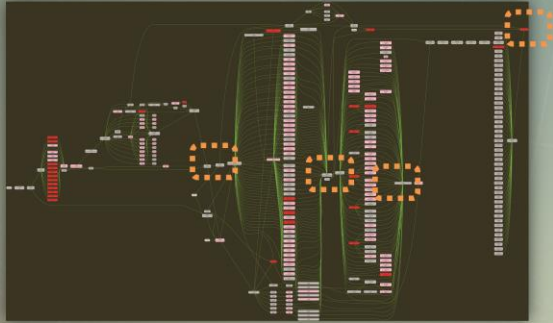
Submit View Job(s)

End Frame Job

Submit Done

However, for every view that we have in a given frame, we create a separate <view job chain>. These job chains are data-driven - they are only created if needed. For example, if a shadow view does not appear in our frame, we do not create jobs for that view.

View Job Chain



Frame Ring Buffer

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done

Each engine phase completes by the <global synchronization> jobs. These synchronization jobs are used by the core renderer and simulation system to control access patterns to the underlying data containers (render and visibility objects, game state data).

We use the execution environment mechanism that Barry Genova covered in his talk to help us identify any breaks in the expected data access patterns.

View Job Chain

- Determine views
 - Player view?
 - Shadow view?
 - Etc.



Frame Ring Buffer

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done

The <first job> we execute is a global job for the frame to compute which views we're going to render in this frame.

View Job Chain

- Determine views
- Reserve frame and view packets
- No heavy-weight allocations, heaps, etc.

Frame Ring Buffer

Frame Packet

View Packet



View Packet



Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

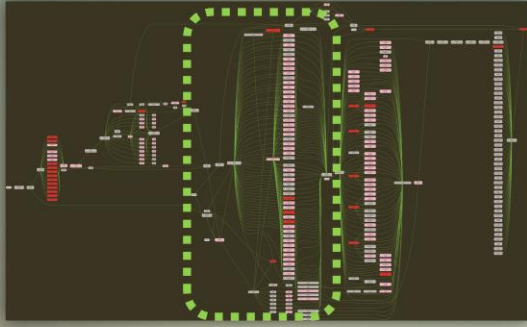
Submit View Job(s)

End Frame Job

Submit Done

This job also reserves the frame packet and <view packets> within it. Each view packet is just encapsulation for all data necessary to operate on this view. <Note> that for all the memory management in the frame packet within the view job chains, we don't perform any runtime dynamic memory allocation in the true sense of that word. There are also no heavyweight heaps here. We simply allocate already frame entries in a lock-free frame ring buffer. Once the views are determined, this job sets up job chains for executing all operations for each view.

Extract



Frame Ring Buffer

Frame Packet

View Packet

View Packet

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done

Next we begin the <extract phase> to copy data out of the game state.

Extract

- Compute a list of **visible** render objects for each view
 - Stored in visibility ring buffer (temp)

Frame Ring Buffer

Frame Packet

View Packet

View Packet

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

Submit View Job(s)

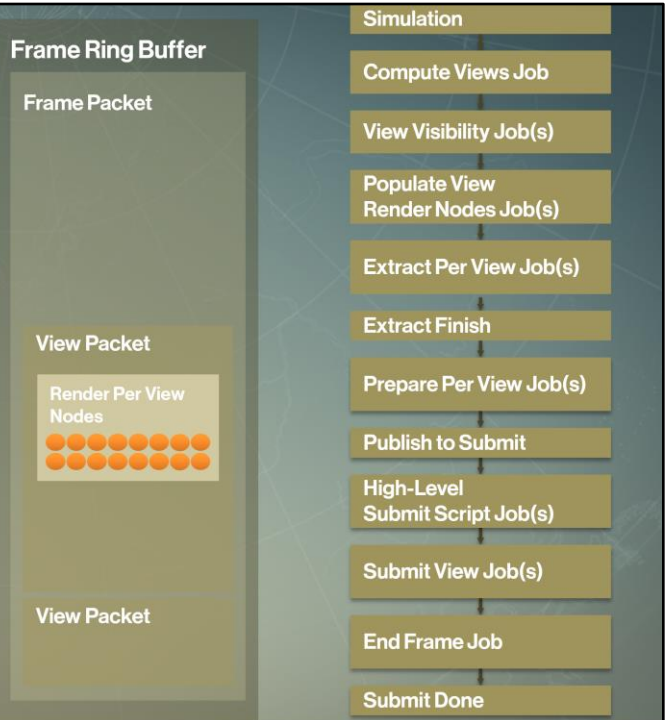
End Frame Job

Submit Done

So now that we have views defined, we run <our visibility job> for that view. This job operates on visibility objects that we just talked about and returns a list of visible render objects. We only need the visible list until the next job we run (populate render nodes for this view). So we store the <visible list> in a temporary visibility ring buffer just until extract is done.

Extract

- View chain jobs operate on render nodes
 - Render nodes are efficient, über-compact, cache-coherent representations
- Populate coherent arrays of render nodes



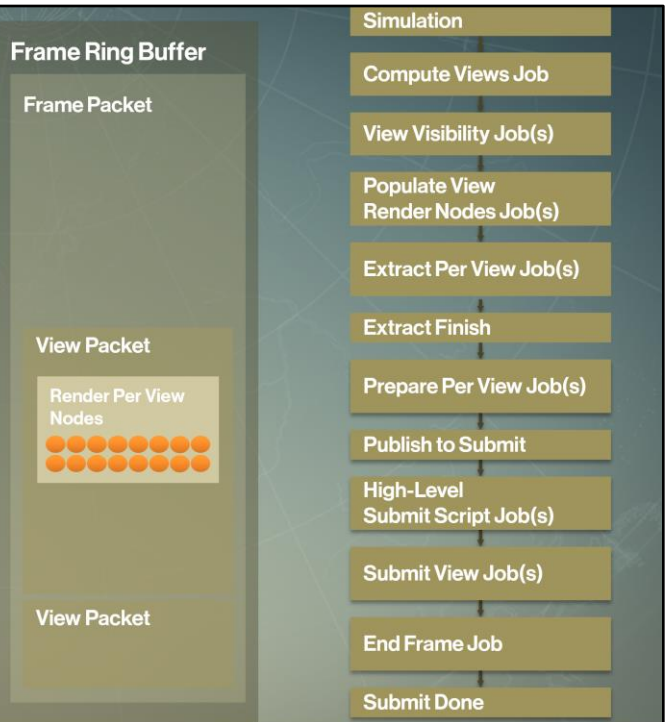
The next job we're going to run for this view is the <populate render nodes> job. This is the job that sets up the cache coherent arrays of data we call render nodes that the rest of the job chain will operate on.

From this point onward, rendering jobs be driven from an array of render nodes.

Each render nodes maps to a unique render object visible in each view. This data structure is defined to be very compact for iteration efficiency.

Extract

- Sort visible list by render object type
- For later execution coherency
- Allocate render nodes in view packet

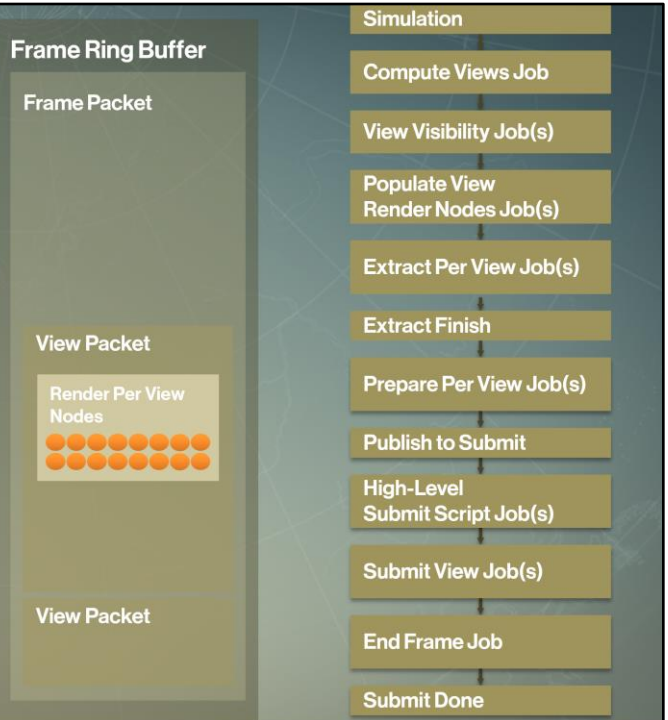


The populate render nodes job will run through the visible elements for that view, and sort them by render object type. This sorting enables coherent execution in later jobs.

It will then <reserve and populate> render nodes in each corresponding view packet for the view we're working on using the lock-free frame ring buffer.

Extract

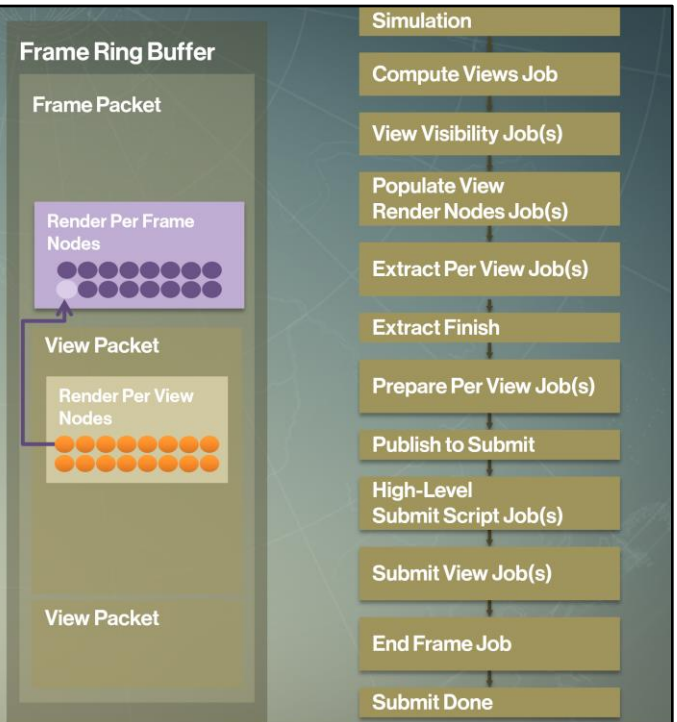
- Render view node stores view-dependent data
- Can allocate custom frame packet data
 - Based on render object type



Each view node stores some view information (bounding sphere, distance from view camera). This data is stored directly in render node for cache coherency for core renderer workloads which run tight iteration loops over the render nodes during extract, prepare and submit operations. A small amount of redundancy for such elements is more than made up by resulting performance improvements, and we selectively choose what data to store in the render nodes by type. The view node can also allocate data in the frame packet if needed. This allocation details are defined uniquely by each render object type (which is stored in the view node).

Extract

- Share data across views with frame nodes
 - Data sharing within frame packet
 - Avoids redundant data



Each view node <maps uniquely to a frame render node>. This allows our system to share data across multiple views. For example, we might have multiple views that have a skinned character visible. All of those views would need to extract and prepare the skinning matrices. By putting those in the frame node we only do this work once, and we only have one copy of the data. We take this even further by allowing game objects to share data too, for different render objects.

Extract

Frame Ring Buffer

Frame Packet

Simulation

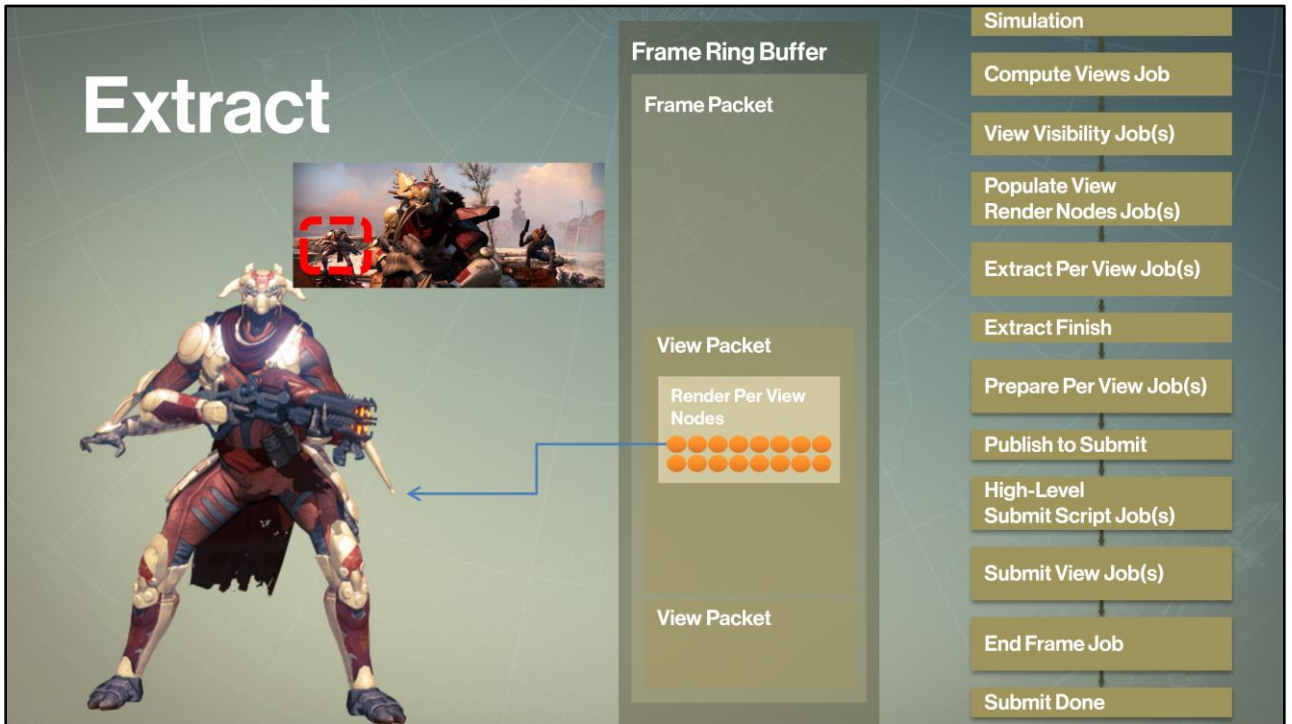
Compute Views Job

View Visibility Job(s)

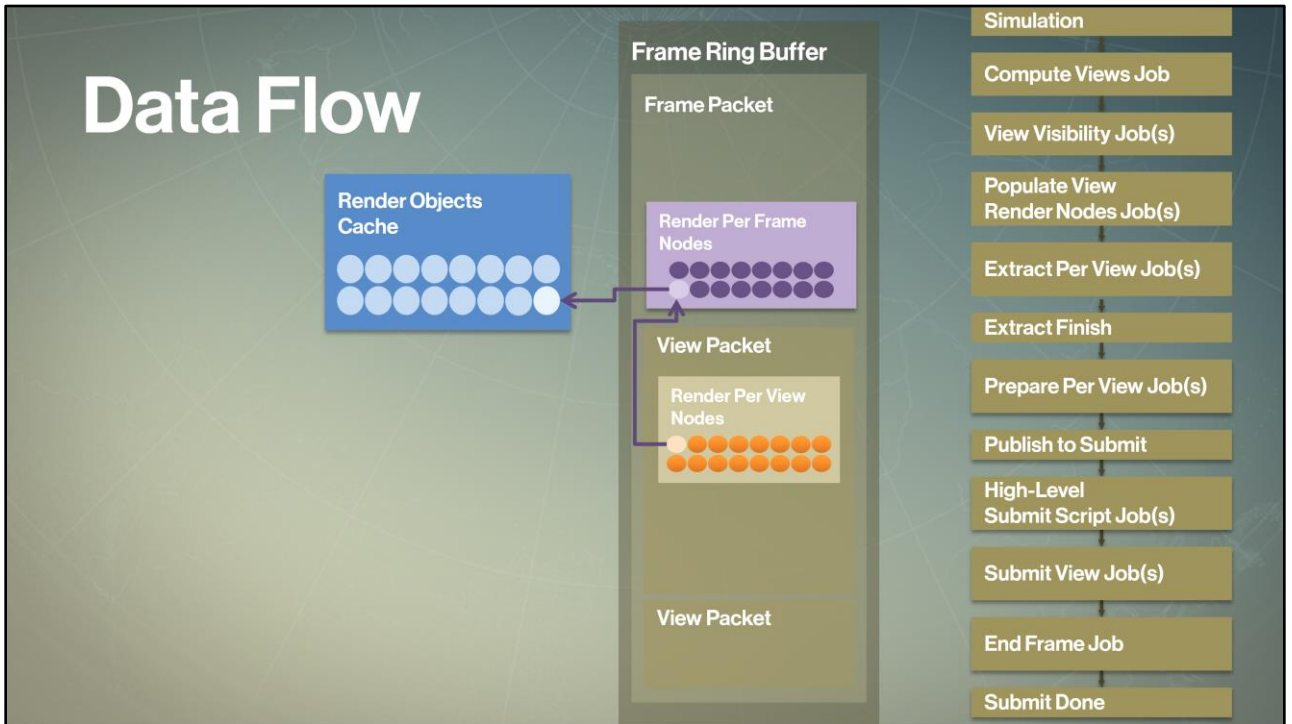
Populate View



For example, if we were rendering this frame with a <raider> character

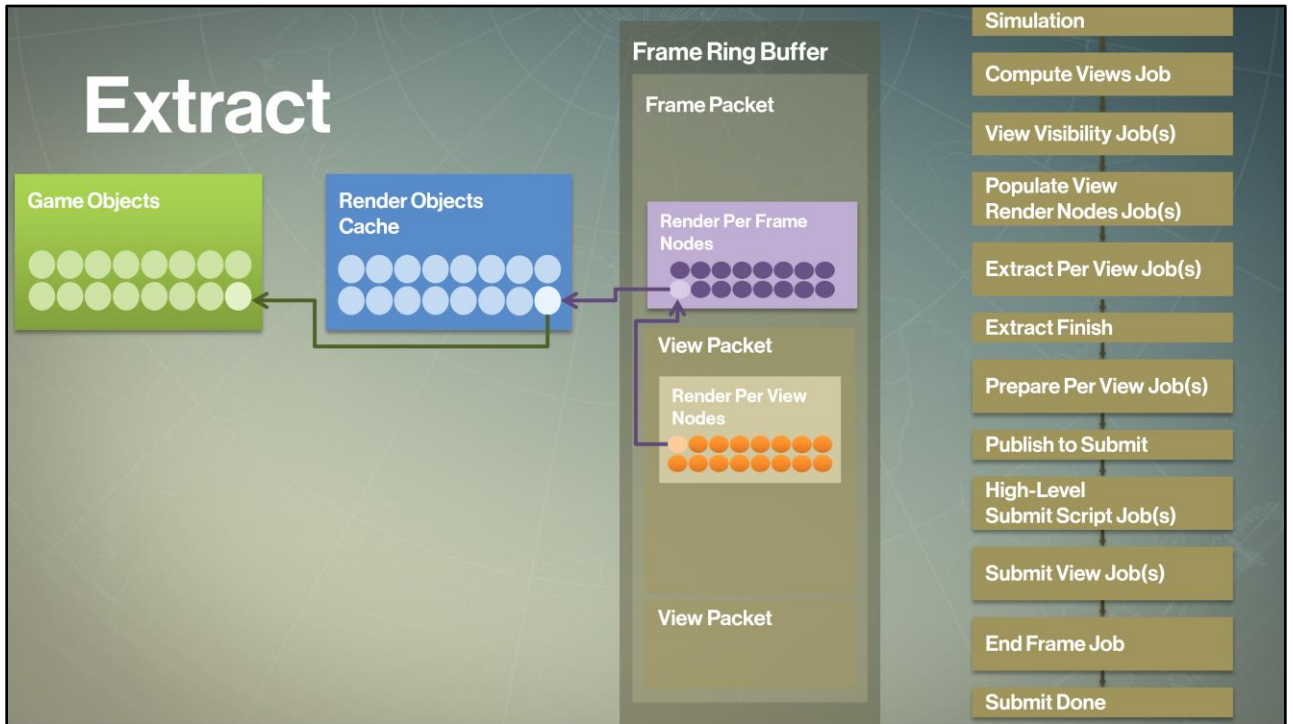


Then if this raider is visible in the player view, then the <view> node would be our <reference> to the raider render object



Each <frame node> links to the render object, which allows us to access its statically cached data.

Next, let's look at our extract jobs execution

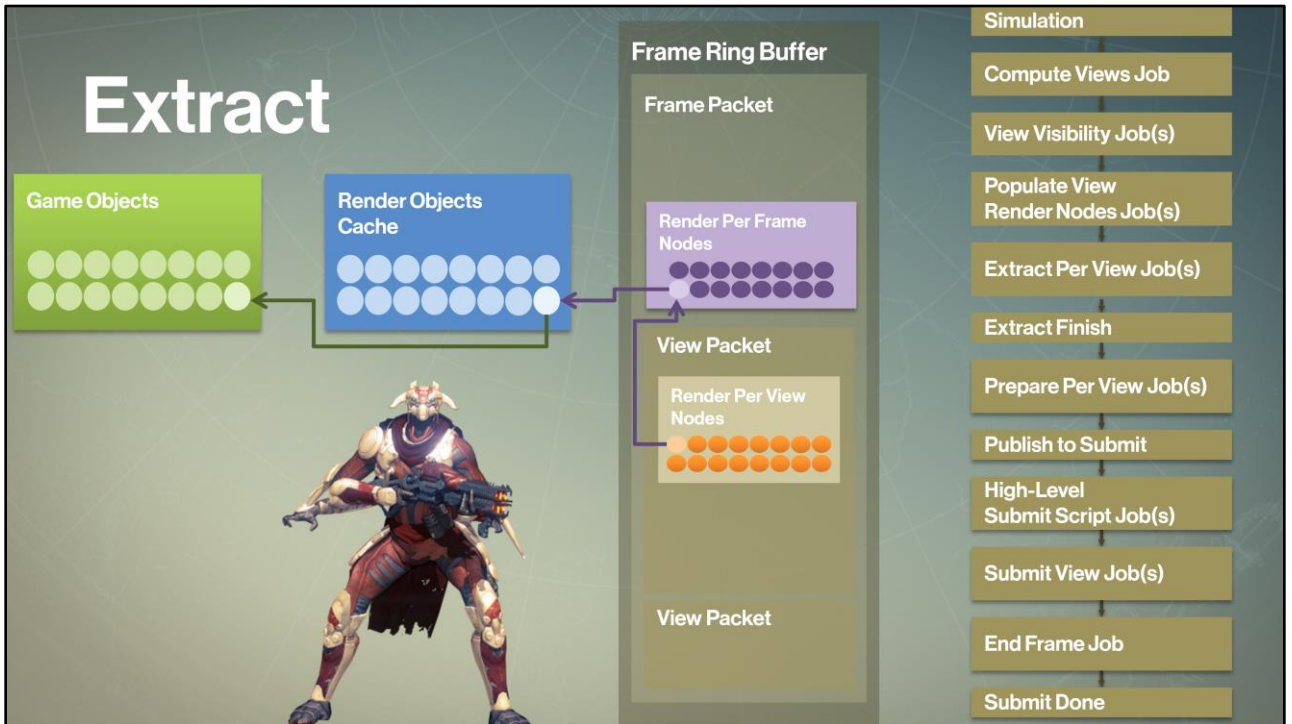


For each view node in the view packet, we're going to extract data out of the game state object into the frame packet.

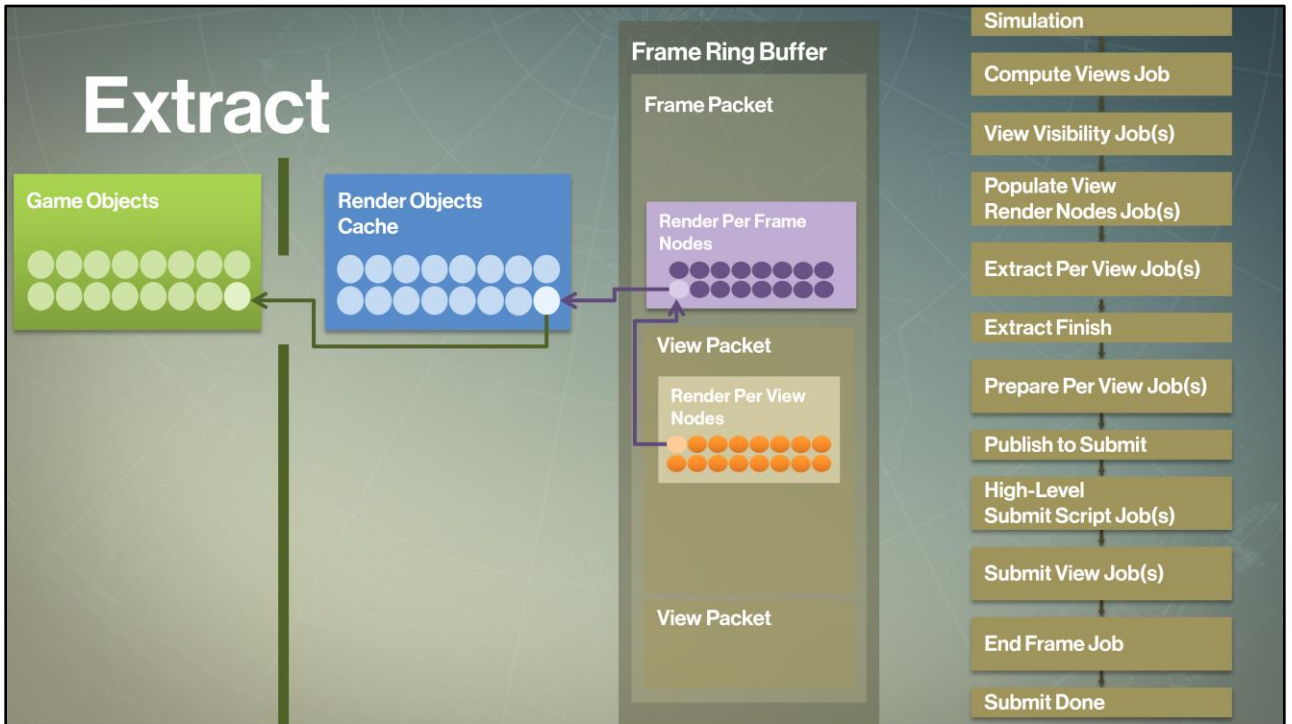
The core renderer will iterate coherently over the view nodes sorted by render object type and execute extract job entry point for each render object type. Those jobs will operate on only this data as inputs:

- Individual frame and view nodes for that visible element
- Statically cached data from the corresponding render object

During extract, we also reach out to the game object using the handle cached in the render object.



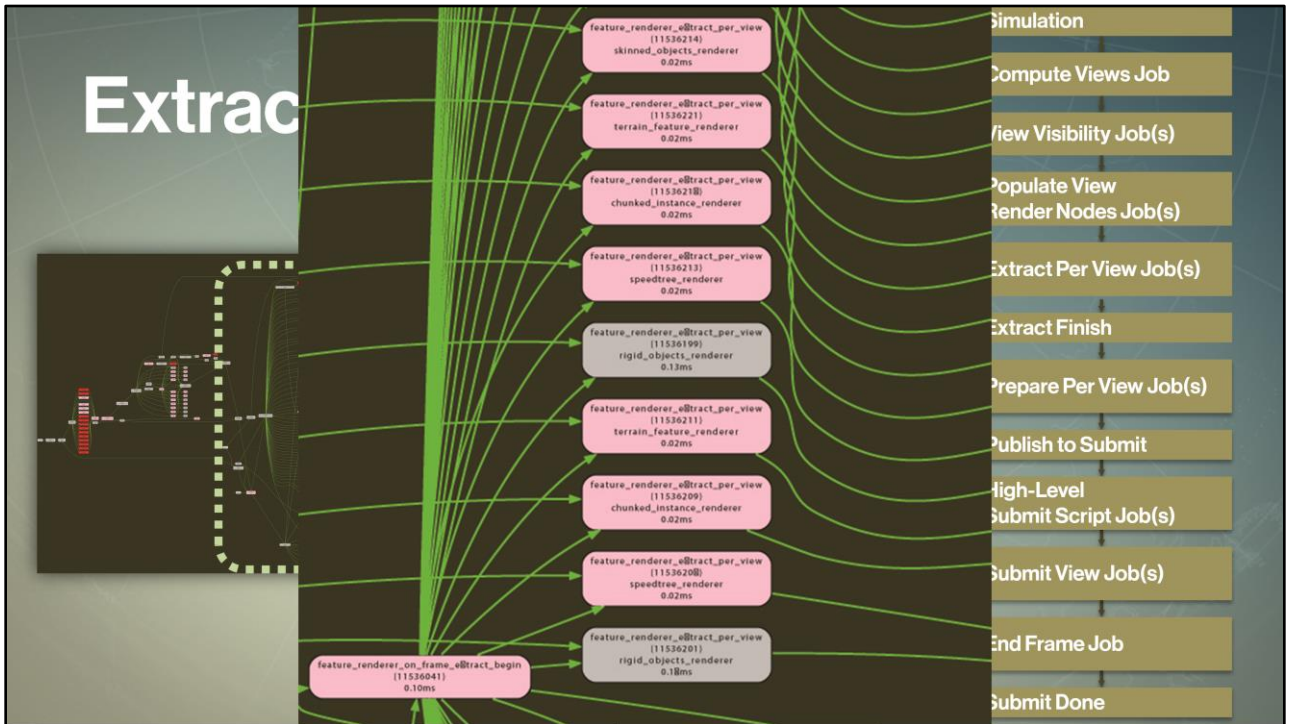
Coming back to our friend the raider here, we would extract and store his per-view attributes (distance from camera, etc.) in the <view node>, we would extract and store his bounding sphere and skinning transforms in the frame node. We would get all this data by reaching out to the <game objects> using the cached <render object>. While this all sounds pretty basic, I'll explain in a minute how we wrapped this up in convenient interfaces to make sure that we could go wide for these operations with safe threaded access.



Note that extract phase is the <only> time we allow crossing the object system and renderer wall.

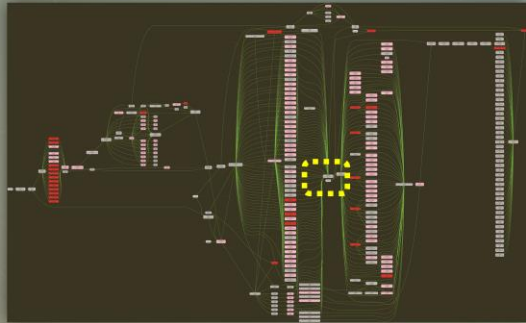


It's akin to <this>



During extract, we actually generate separate jobs to go <wide> during extract. We generate multiple extract jobs for each view. The jobs are generated with smart batching for different render objects types.

Extract



Frame Ring Buffer

Frame Packet

Render Per Frame Nodes



View Packet

Render Per View Nodes



View Packet

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

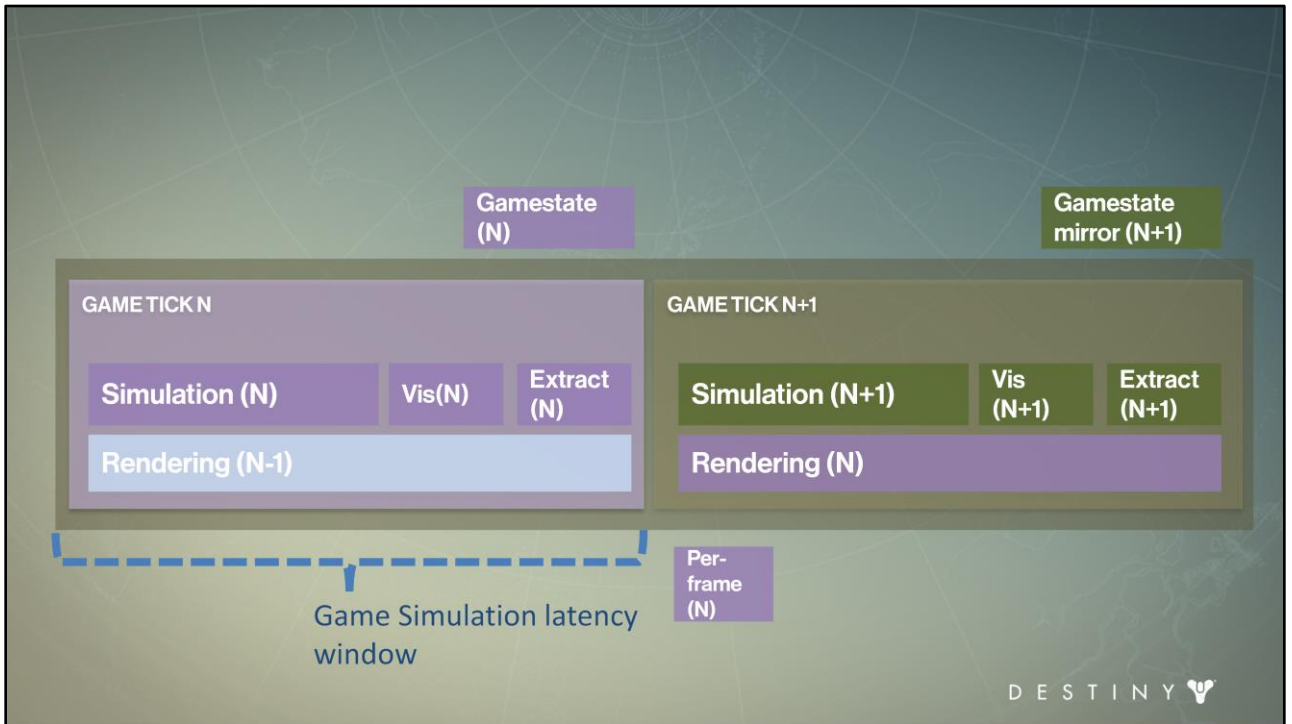
High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done

Once extract jobs are finished, we tell simulation that <extract is complete> – this signals that simulation can proceed computing the next game tick. Before we go much further, there's an important point to touch on – namely game simulation latency.



Coming back to an earlier diagram, we see that <extract> phase is required to complete before we can <unblock> the next frame's game tick, thus increasing <game simulation latency>

Extract Window and Latency

- Extract blocks simulation from proceeding
 - Gamestate is locked for read
- Making extract most latency-impacting workload
- Going wide is crucial
 - But not enough

DESTINY 

During the extract phase, all of the output systems (rendering, networking, UI, audio) are *reading* gamestate.

Which means it can't be modified during that phase thus we cannot start next frame's simulation.

So we have to run extract as quickly as possible to unblock simulation.

During Destiny development, extract was our most latency-sensitive workload that we constantly fought to reduce.

Optimizing Extract Window

- Extract window consists of:
 - Visibility computations for all views
 - Game state data extraction
- Visibility computation is CPU work
 - Can we move visibility out of extract?
 - Player view's static environment visibility is heaviest workload

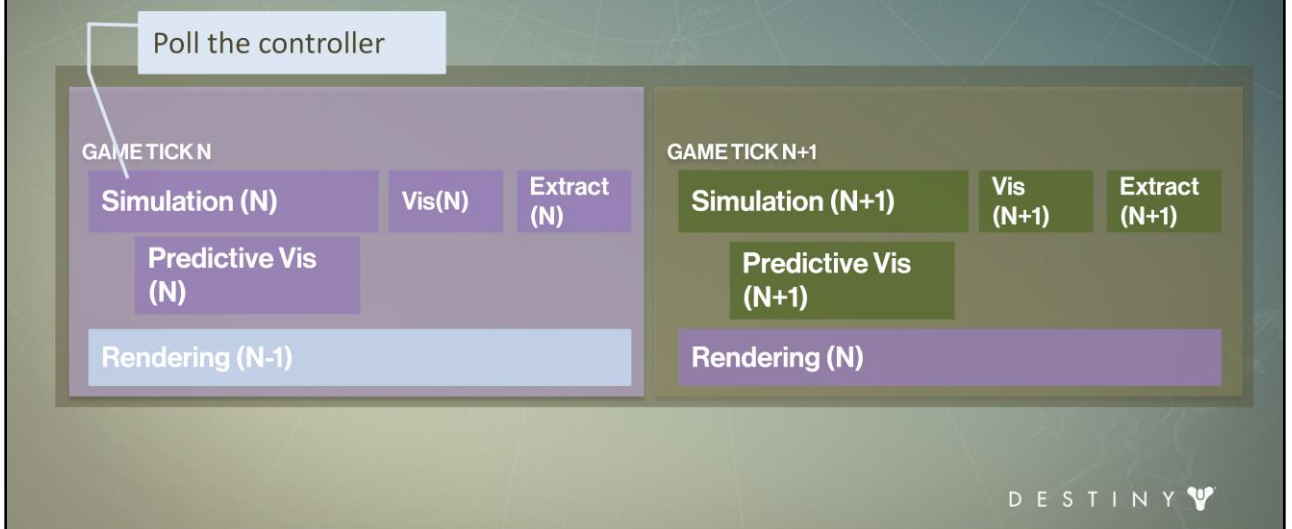
DESTINY 

Extract window consists of visibility computations for all views and gamestate data extraction.

Visibility computation is CPU work. Can we move visibility out of extract?

Observation: the main player view's static environment visibility is the heaviest workload in visibility computation. Let's move that out of the latency-sensitive extract window

Predictive Visibility



If we stagger visibility with simulation we can achieve that. We start by <polling> the controller during simulation (as late as possible). As soon as we have the controller information, we determine what views are present in our frame.

Predictive Visibility

Poll the control

GAMETICK N

Simulation (N)

Predictive Vis
(N)

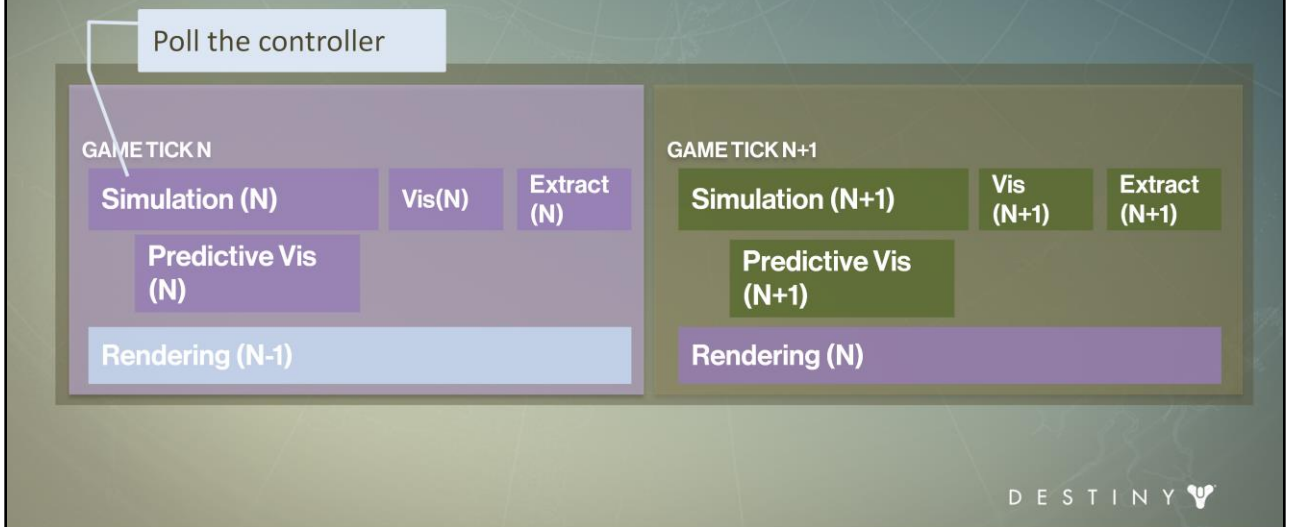
Rendering (N-1)



DESTINY 

Then we are going to run <static visibility> computations *predictively* for our main – heaviest – <player view>. It's the view where we see all of the environment objects for our gameplay.

Predictive Visibility



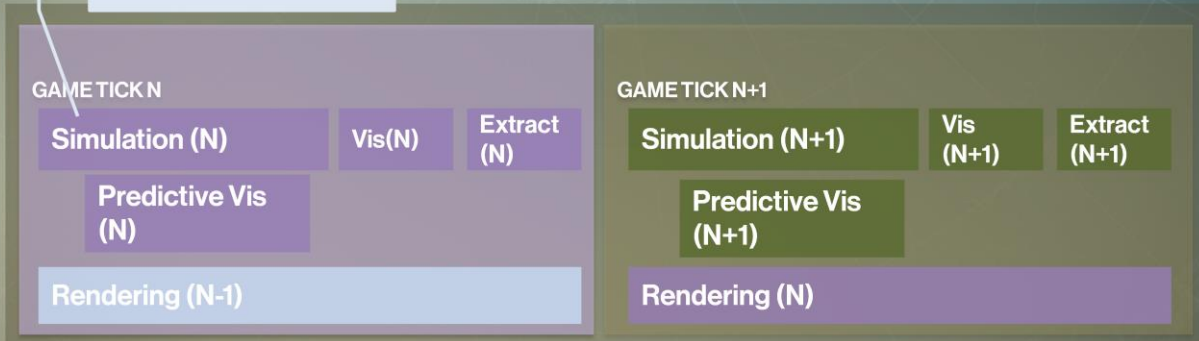
There, we are going to compute visibility for static environment objects, which is the bulk of computations for the player view. Then <we will run> visibility computations for any dynamic views or dynamic objects.

Because simulation might move the dynamic objects (physics or AI decisions), we can't compute their visibility until the simulation is complete. Similarly, we create dynamic views (this includes, for example, dynamic shadow views) when we're done with static environment visibility (when we know that a shadowing light is visible in the player view, and we'll need its shadow).

Predictive Visibility

Reduced game latency by
~4 ms last gen and ~2.5 ms current gen

Poll the controller



DESTINY 

This <reduces> game latency by 3-4 ms on last generation consoles and by ~2-3 ms on current generation consoles.

Extract versus Prepare

- Extract:
 - Copy out the dynamic data with minimal work
 - Don't run any complex data transformations
- Finish and unlock gamestate / simulation for next game tick

DESTINY 

Next, we separated the work to *extract* data out of game state in its raw form from *prepare* (where we'll transform that data to render-ready form).

During extract, we are going to simply copy out the game state data with minimal work and not do any complex transformations on that data.

And then we will unlock game state to unblock simulation for the next game tick.

Prepare

- Run **prepare** jobs for **visible elements only**
- Skip computations based on LOD if possible
 - Don't compute what won't be perceivable

DESTINY 

We're going to run prepare jobs for *visible elements only*.

By running them *after* visibility computation we also avoid extraneous work for elements that won't be seen.

We also can use LOD to skip any computations that aren't going to be perceivable



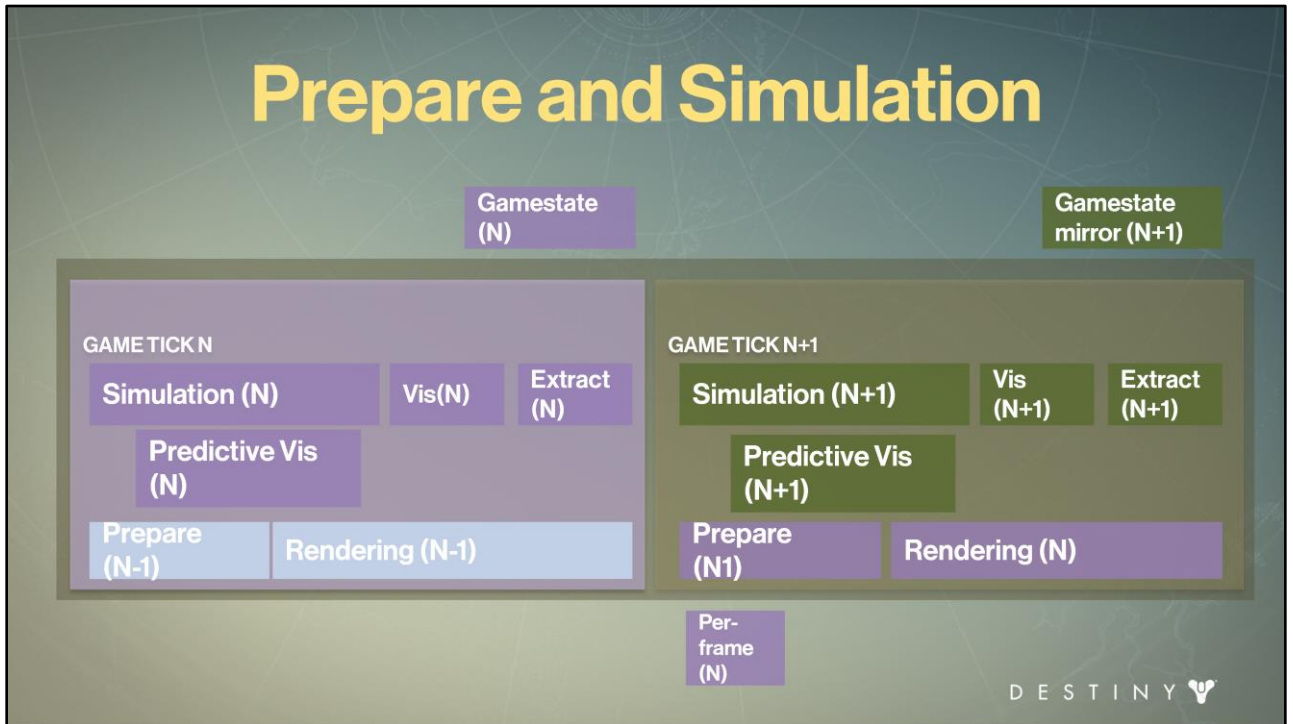
<video>

This video can show a good amount of examples for the work we do in prepare phase. We convert the extracted data to GPU-friendly data formats. For example, for characters that you see, we convert local-space animation transforms to world-space dual quaternion vectors or linear blend skinning matrices ready to be copied directly to GPU registers during prepare.

We also run non-game-affecting simulation workloads in prepare such as cloth and particle simulations.

Another really important set of workloads we compute in prepare is for maintaining predictable performance budgets. We have designed our system to maintain predictable performance for various parts of our pipeline (for example, for cloth simulation, or for GPU cost of skinned elements). During prepare execute work to help us distribute visible elements into LOD buckets, for example, to maintain a consistent skinning budget per frame, or for trees instance, etc.

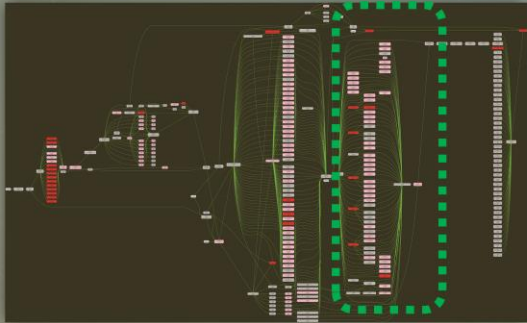
Prepare and Simulation



Then you can <prepare> the render-ready data in subsequent jobs while <simulation> for next game tick is running.

This allows us to pipeline the work much better.

Prepare



Frame Ring Buffer

Frame Packet

Render Per Frame Nodes



View Packet

Render Per View Nodes



View Packet

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

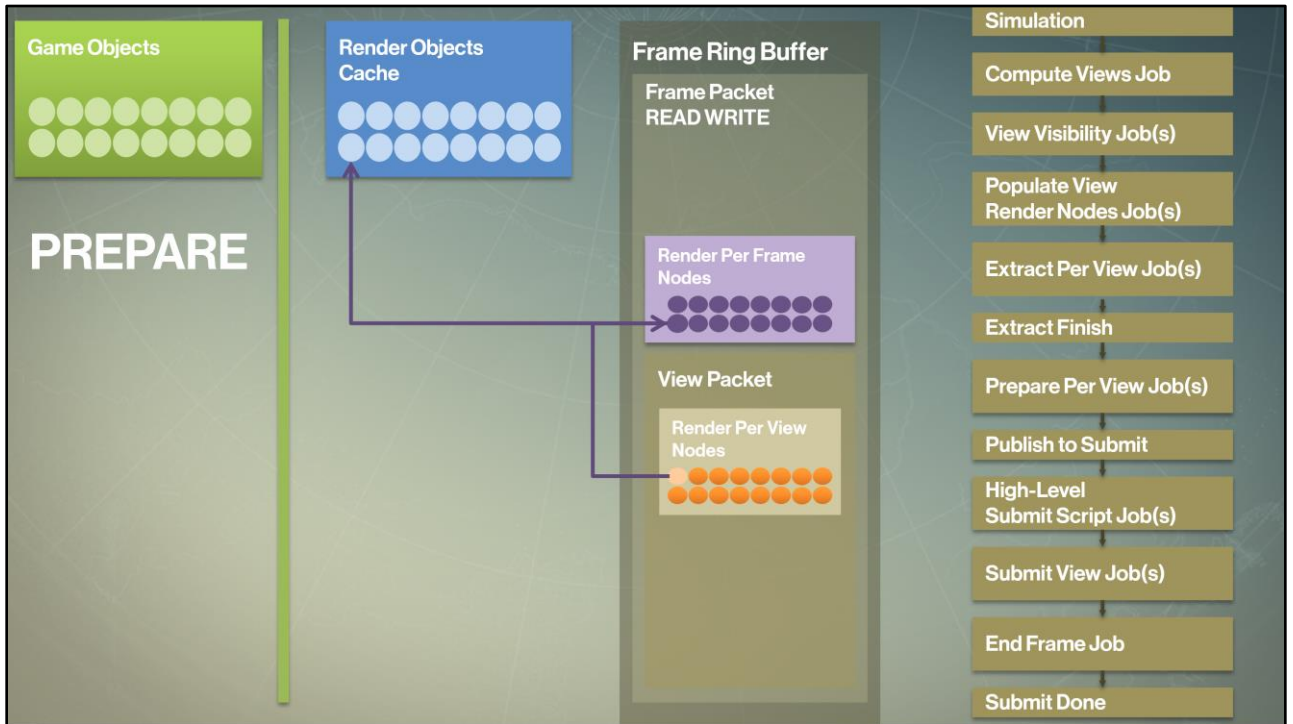
High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done

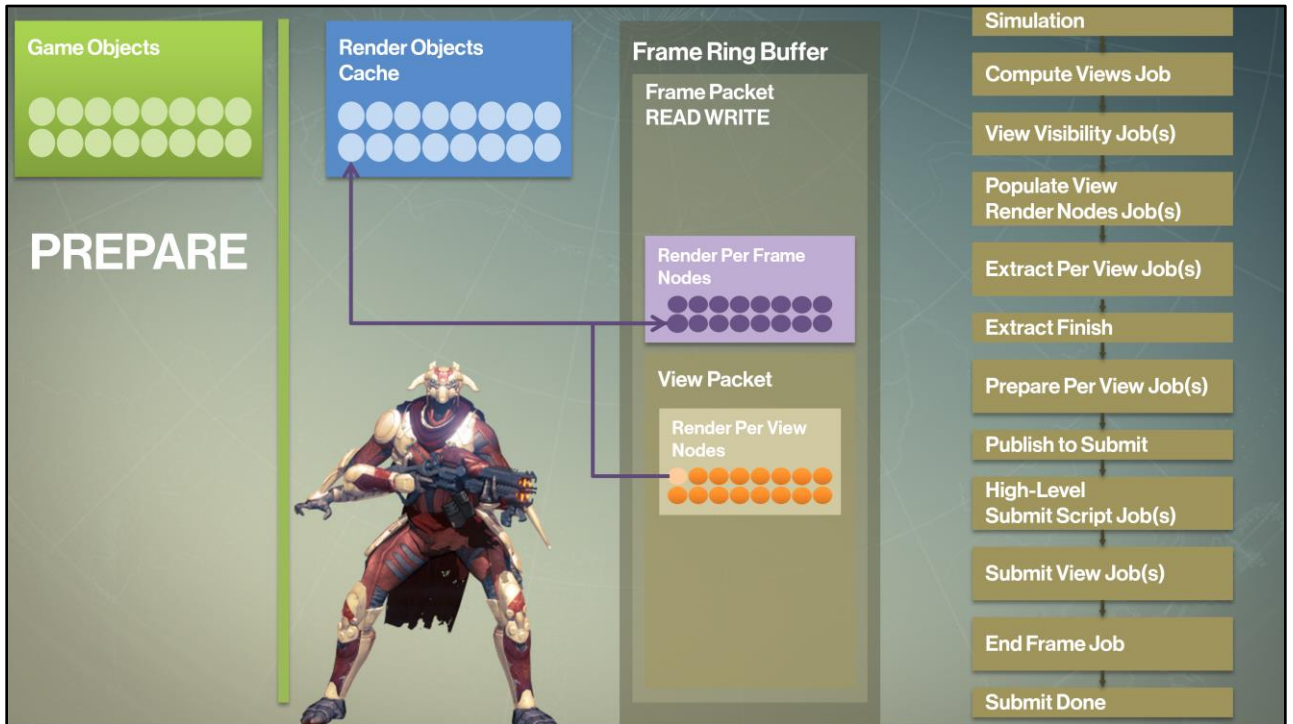
So now while simulation is running in parallel, we are going to execute the <prepare> phase



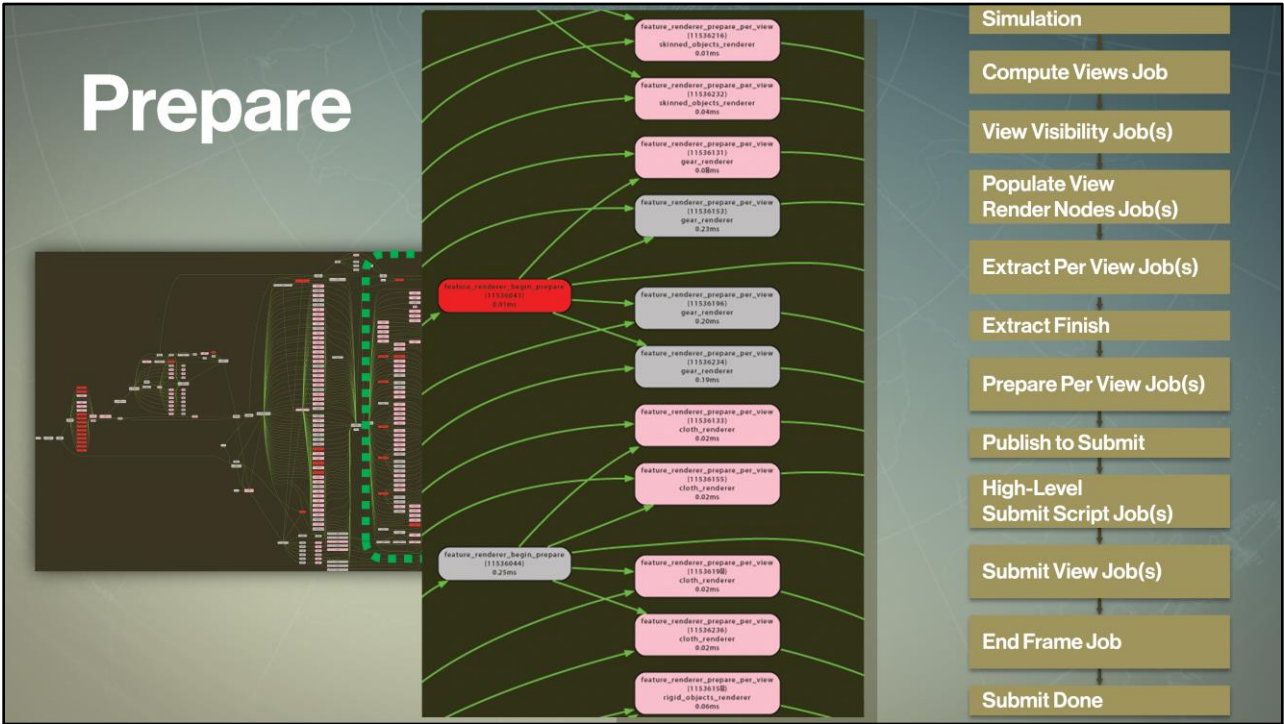
Similar to extract, prepare also operates by iterating through the view and frame nodes and cached render objects data, executing prepare entry points for different render objects.

Prepare writes results of each prepare jobs into frame packet as well.

But there's an important difference from extract: <render jobs no longer can access game objects>.

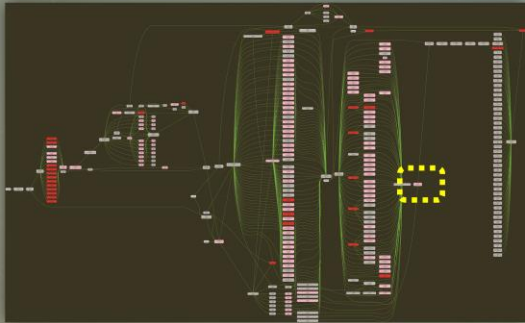


Bringing back our friend the raider, during prepare we will run cloth simulation jobs for its tattered cape, we will also compute non-deterministic animation for his fingers and toes bones if the raider is close enough to the player (or skip that work if he's not). We'll also bucket him based on his vert count into a skinning LOD bucket to maintain a consistent LOD footprint for our combat frame.



During prepare phase the core renderer architecture generates per-render-object type jobs automatically, so that the phase also goes wide.

Publish



Frame Ring Buffer

Frame Packet

Render Per Frame Nodes



View Packet

Render Per View Nodes



View Packet

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

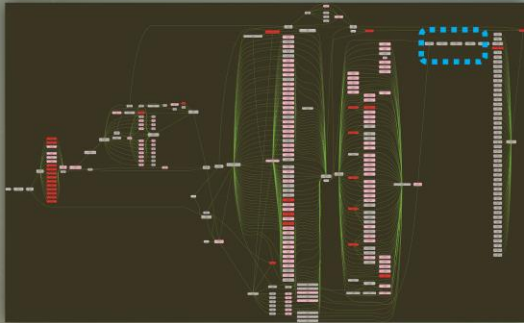
Submit View Job(s)

End Frame Job

Submit Done

Once prepare jobs completed, the core architecture closes the frame packet for write and signals the system that we're now ready to start submitting to the GPU. The reason why we need this explicit synchronization is because we may have multiple jobs generating GPU-friendly data, and we need to complete all that work before we start piping it to the GPU. Past the end of prepare, no jobs are allowed to write to the frame packet for this frame.

Submit



Frame Ring Buffer

Frame Packet

Render Per Frame Nodes



View Packet

Render Per View Nodes



View Packet

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

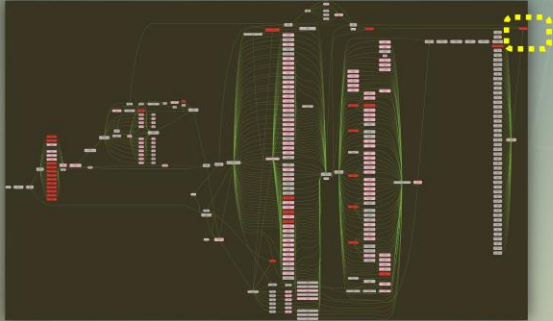
Submit View Job(s)

End Frame Job

Submit Done

Next we will execute jobs to generate GPU commands for this frame. We'll talk about submit in details in a few minutes.

End Frame



Frame Ring Buffer

Frame Packet

Render Per Frame Nodes



View Packet

Render Per View Nodes



View Packet

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

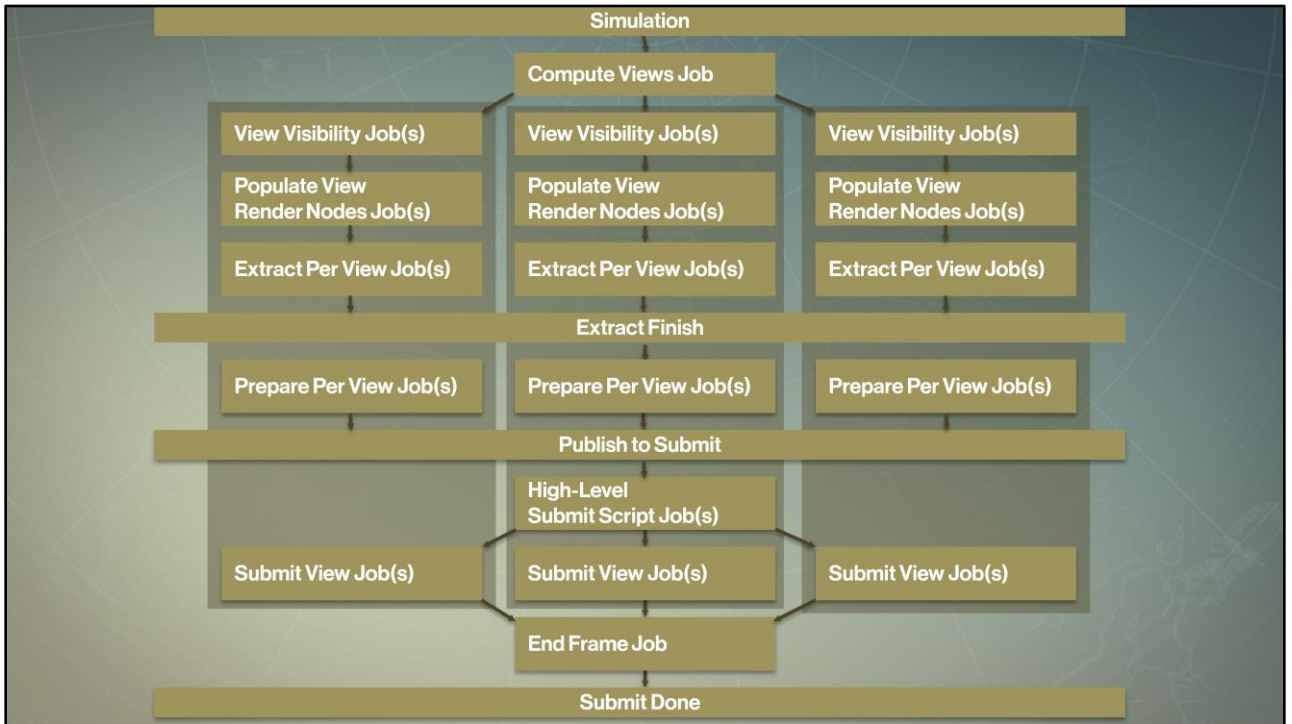
High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done

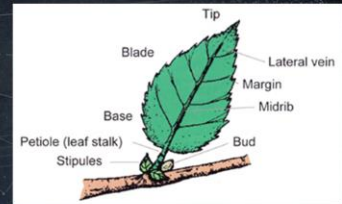
When all submit view jobs are done, we execute the <final present> for the frame in end frame job. This synchronization point indicates to the core system that we are ready to flip any time GPU gives us back the signal that it processed all workloads. At the same time, we are ready to start generating the next frame's worth of GPU commands.



We can repeat this jobification process for additional views (for example, if we had three views total in a frame) – and we have the following job chains for each view

Keep It Simple, ...!

- Simplify graphics feature development
- Transparent jobification for rendering workloads
- Guarantee cache coherency and proper synchronization for core workloads
- **Solution: feature abstraction and encapsulation**

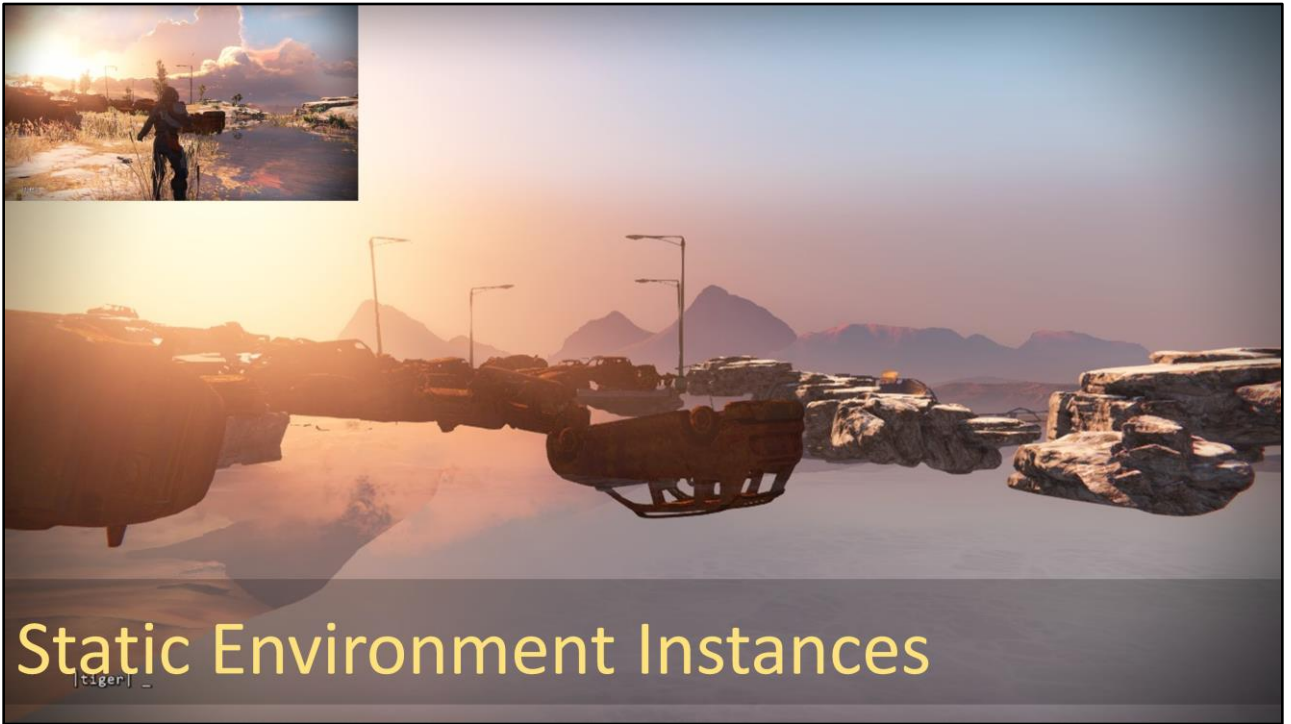


DESTINY 

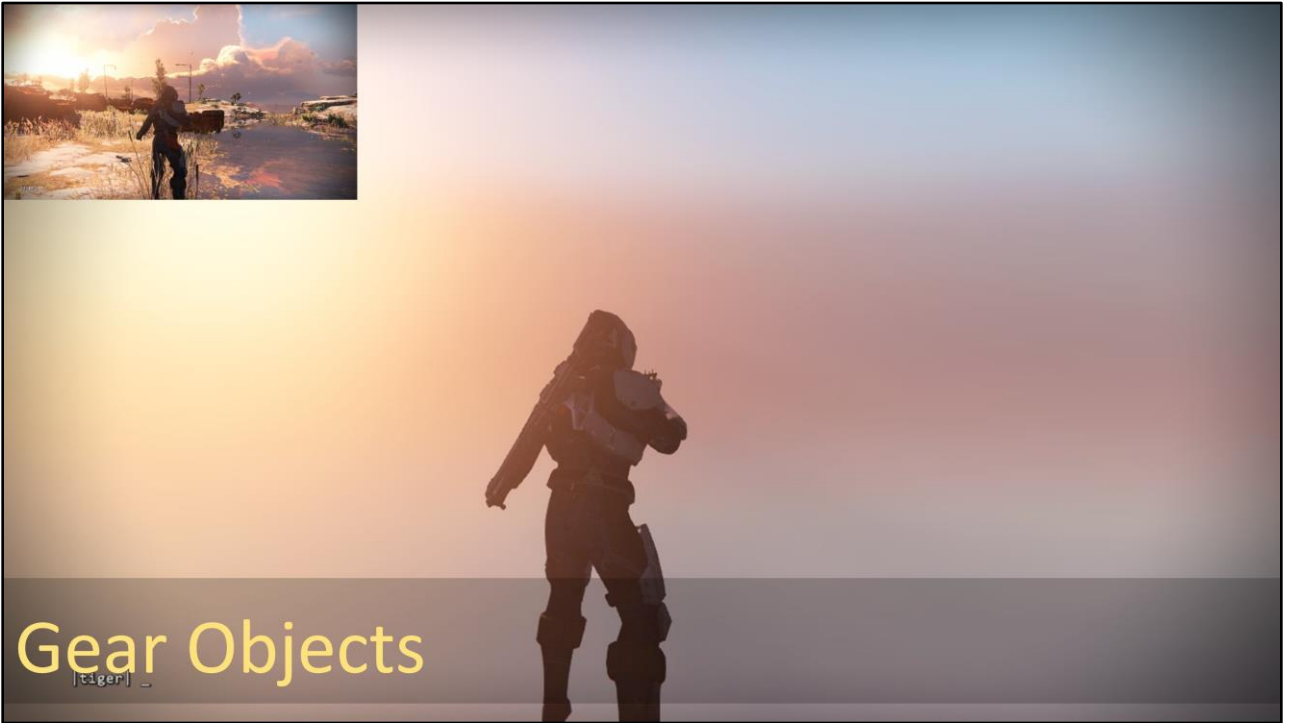
Next, let's talk about what all this complex job chaining meant for writing leaf features – i.e. the bread and butter of game graphics code. Threading should be transparent and automatic for the leaf features. Every one of our graphics engineers wrote graphics features (including myself), and we didn't want to have to change job dependencies or job occupancy, or synchronization any time a new feature was added. We also didn't want to do this for every new platform we support. So jobification had to become transparent and automatic and guarantee cache coherency and proper synchronization for core rendering workloads. <The way> we have implemented this is via the feature abstraction.



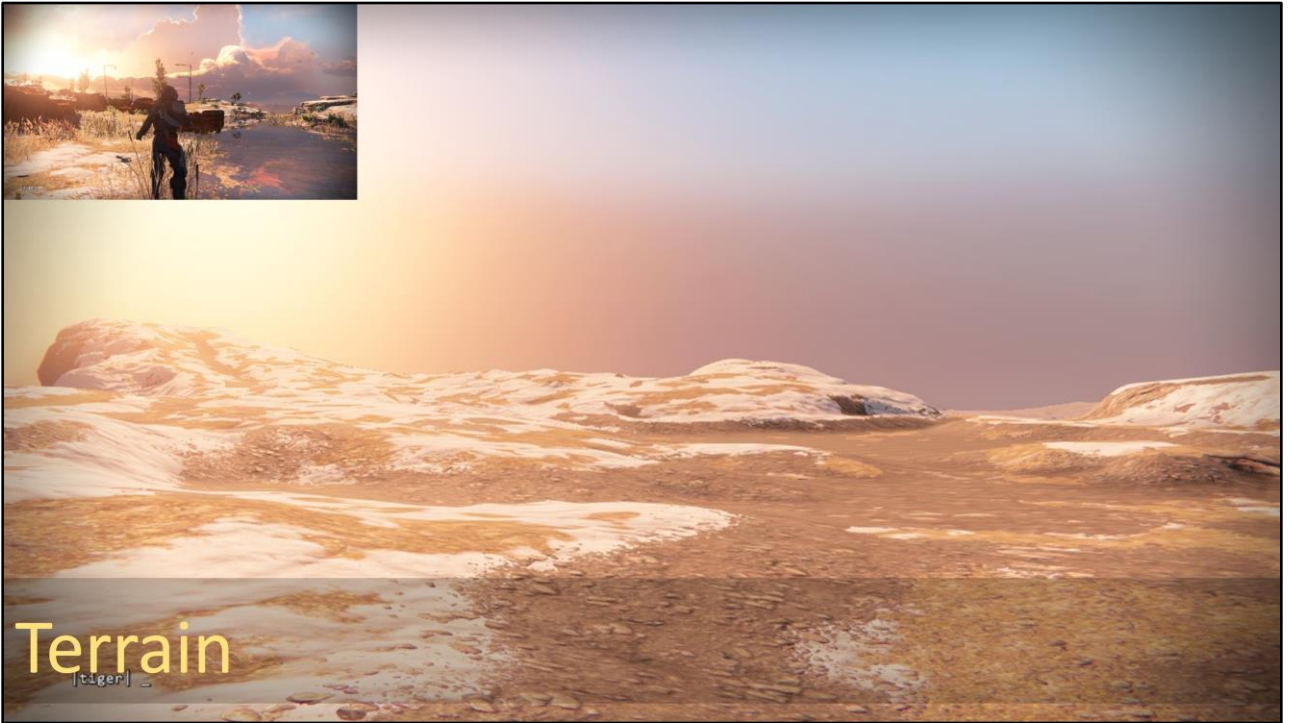
So far we talked about our frame rendering as if we have the same types of render objects. But our game has many different render object types. For example, this shot is made of ...



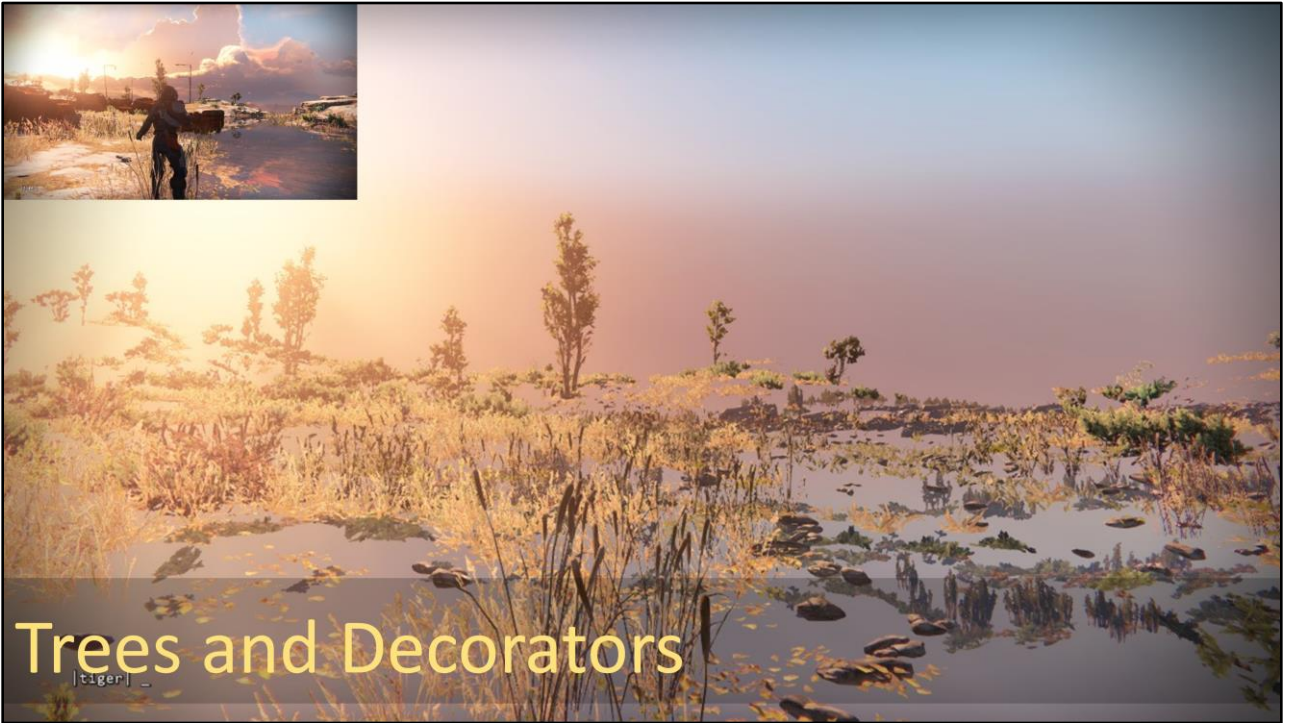
Static environment instances render objects



Gear render objects

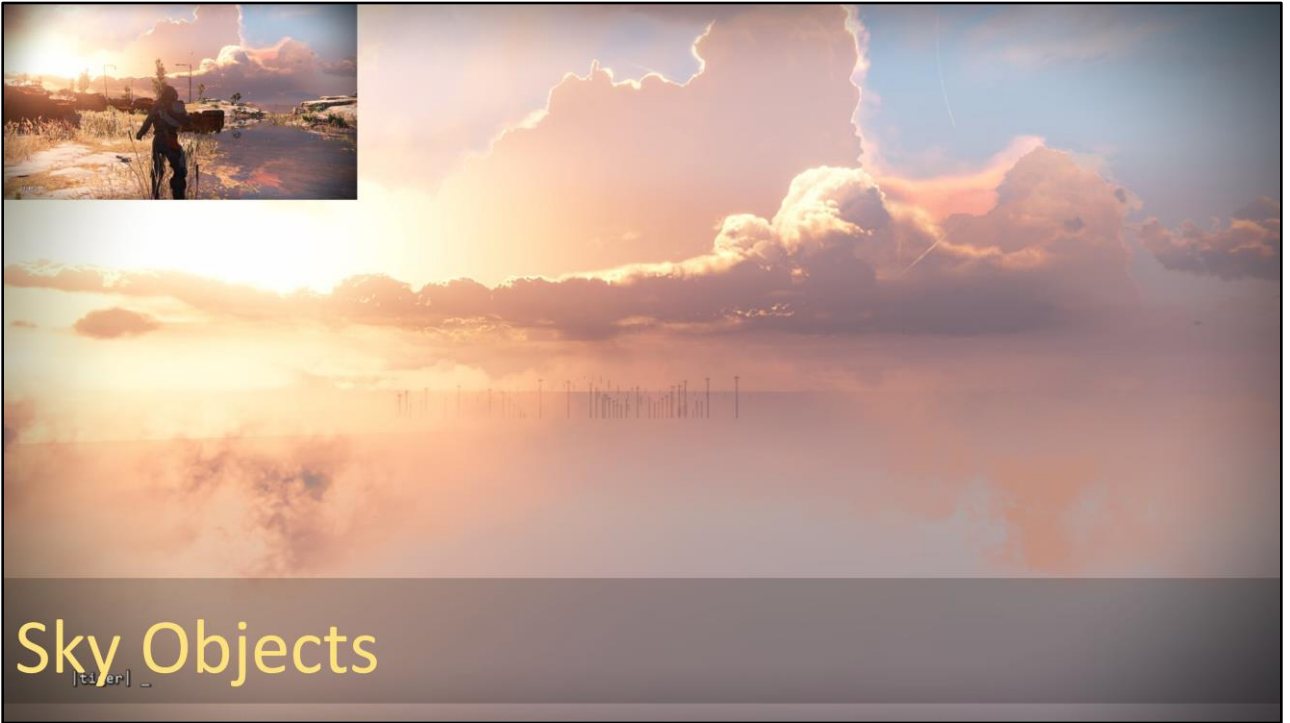


Terrain



Trees and Decorators

Trees and Decorator render objects populating our environment




Sky dome objects



And many other features which together builds up this frame.
How do we express this taxonomy in our core renderer architecture?

Render Features


- A *render feature* is a unit of graphics rendering
 - Defined by:
 - Cache-coherent data representation
 - Code paths for data management and rendering
- Maps to a *feature renderer instance*

DESTINY 

Each render object type maps to a *render feature* which is the basic encapsulation unit in our architecture. We group render functionality by same data representation and code paths. For example, most of the skinned characters need dynamic transforms for skinned data, shaders and meshes, and iterate over the mesh containers to generate drawcalls, uploading skinned matrices to the GPU. A particle system would have a different data representation and have very different draw call generation logic.

Destiny Feature Renderer

- Defines how graphics features map to
 - Render objects with cached data representation
 - Frame packet render node representation
 - Job entry points for each phase

DESTINY 

Each render feature is implemented with a unique *feature renderer* responsible for all work for that graphics feature. Feature renderers are the main interface for implementing graphics features in the Destiny engine. This interface defines how graphics features represent their cached render data; how we extract dynamic data from game objects into frame packet render nodes, how we convert that data to GPU-friendly formats, and, most importantly, the code path to render these objects. Feature renderer interface exposes entry points for each of our engine's phases which the core renderer architecture converts into jobs. This interface also provides data encapsulation to allow safe multi-threaded data access.

Game Objects and Features



- Game objects 1 : n render objects
 - Static or dynamic mapping

DESTINY 

A game object can register with multiple feature renderers – that can be statically defined based on imported components of the object or dynamic (if we add new components to the game object at runtime).
For example, this Hunter can register with ...

Game Objects and Features



- Game objects 1 : n render objects
 - Static or dynamic mapping
- Example: Hunter
 - Cloth feature renderer



DESTINY 

simulated cloth, ...

Game Objects and Features



- Game objects 1 : n render objects
 - Static or dynamic mapping
- Example: Hunter
 - Cloth feature renderer
 - Skinned objects feature renderer

DESTINY 

skinned objects, ...

Game Objects and Features



- Game objects 1 : n render objects
 - Static or dynamic mapping
- Example: Hunter
 - Cloth feature renderer
 - Skinned objects feature renderer
 - Gear feature renderer

DESTINY 

and customizable gear feature renderers.

Feature Renderer Jobification

- Each feature renderer exposes a set of entry points for each phase
 - Constrained data access:
 - Only reads corresponding node state and feature render object data
 - Outputs *only* to frame packet data
 - Double-buffering was automatic for all output

DESTINY 

We provided an interface for feature renderers entry points with strict rules for data they are allowed to read or write at each entry point. Feature renderers could only read the render node data and statically cached render object data. They are also only allowed to output to frame packet data. The latter was done to automatically ensure synchronization – double-buffering of dynamic data was automatic for feature writers as long as they wrote to the frame packet.

Feature Renderer Jobification

- Core renderer jobifies for each phase using these entry points
- Without affecting leaf features
 - Leaf features are not affected when core architecture restructured job dependencies or load-balancing rules.

DESTINY 

The core renderer architecture generates jobs for each phase by batching across multiple visible render objects of the same feature type for several entry points (for example, batching all extract entry points into one extract job). This jobification is done transparently to feature writers. This was very important while shipping Destiny because as we added new platforms, or had to significantly restructure job dependencies or load balancing granularity for existing platforms (which happened quite a few times throughout our development cycle), the leaf features' code was not affected.

Feature Renderer Entry Points

ON FRAME
BEGIN EXTRACT

EXTRACT PER
FRAME

EXTRACT PER
VIEW

EXTRACT PER
VIEW FINALIZE

ON FRAME
EXTRACT
FINALIZE

ON FRAME
BEGIN PREPARE

PREPARE PER
FRAME

PREPARE PER
VIEW

PREPARE PER
VIEW FINALIZE

ON FRAME
PREPARE
FINALIZE

ON SUBMIT NODE
BLOCK BEGIN

SUBMIT NODE

ON SUBMIT NODE
BLOCK END

Here is the set of all entry points for feature renderers that our interface provides. You probably can't read their names but don't worry about that, they are just to give you an idea how our system worked.

Feature Renderer Entry Points

ON FRAME
BEGIN EXTRACT

EXTRACT PER
FRAME

EXTRACT PER
VIEW

EXTRACT PER
VIEW FINALIZE

ON FRAME
EXTRACT
FINALIZE

ON FRAME
BEGIN PREPARE

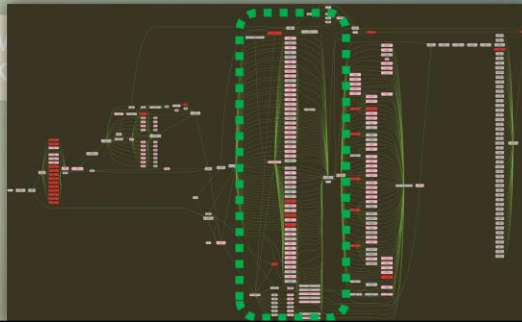
PREPARE PER
FRAME

PREPARE PER
VIEW

PREPARE PER
VIEW FINALIZE

ON FRAME
PREPARE
FINALIZE

ON SUBM
BLOCK



EXIT NODE
END

These entry points mapped to each phase: extract ...

Feature Renderer Entry Points

ON FRAME
BEGIN EXTRACT

EXTRACT PER
FRAME

EXTRACT PER
VIEW

EXTRACT PER
VIEW FINALIZE

ON FRAME
EXTRACT
FINALIZE

ON FRAME
BEGIN PREPARE

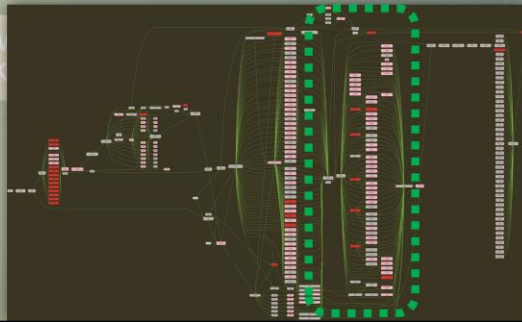
PREPARE PER
FRAME

PREPARE PER
VIEW

PREPARE PER
VIEW FINALIZE

ON FRAME
PREPARE
FINALIZE

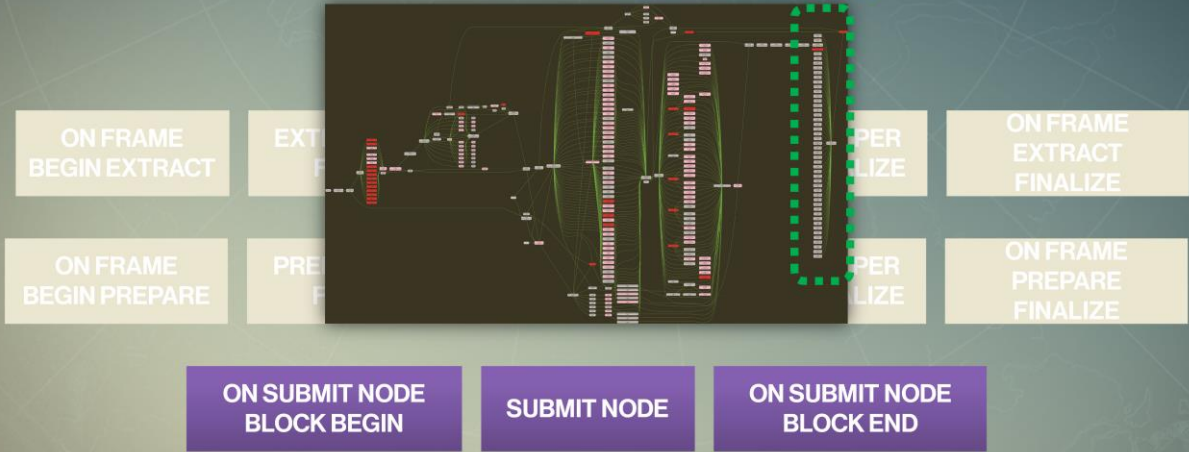
ON SUBM
BLOCK



EXIT NODE
END

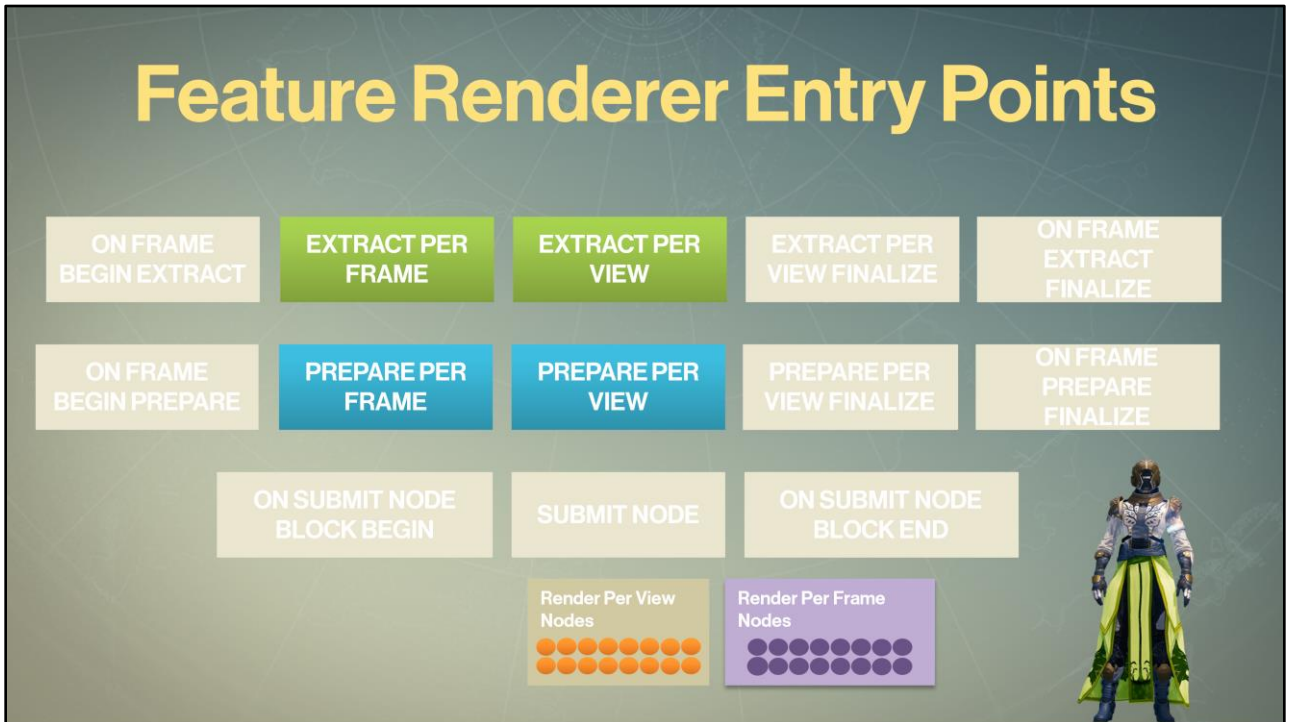
Prepare

Feature Renderer Entry Points



And submit.

Feature Renderer Entry Points



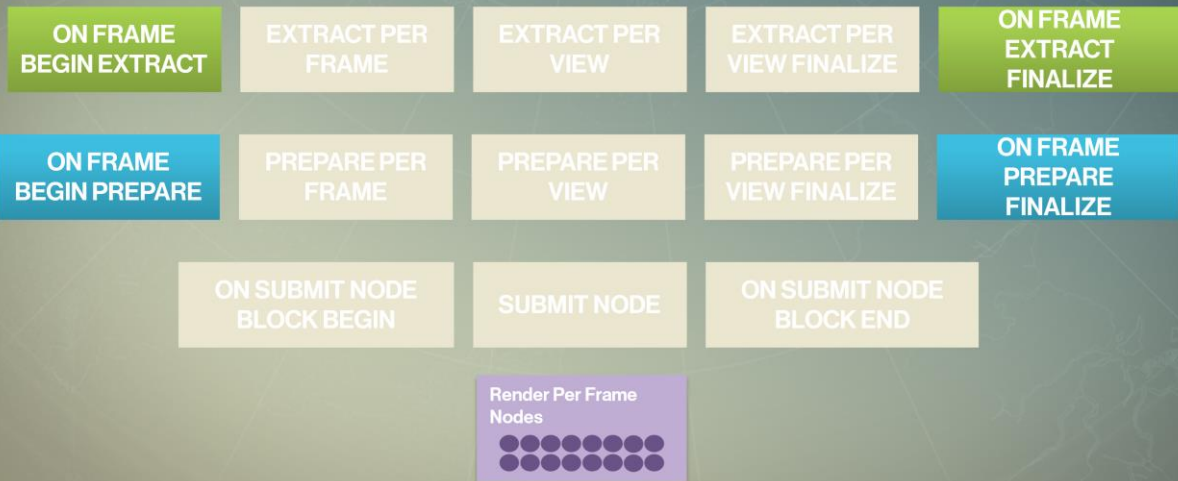
For extract and prepare we also had entry points that operated *only* on the <individual local render nodes> (per view or per frame) – these were the bread-n-butter extract and prepare entry points for regular render object operations (extracting and preparing skinning bones, for example). In other words, these entries only operate on one render object at a time – like <this piece of simulated cloth> here

Feature Renderer Entry Points



But there were cases where you had to do operations on all <visible nodes> of the same feature type in a given view. We added extract and prepare finalize entry points to the feature renderer for that. By these entry points, all previous per-object operations were complete and we could do view-global operations, for example, running computations for all visible <<cloth elements>> in our player view. We used that also for sorting lights into priority queue to generate dynamic local light shadows, or LOD bucketing all skinned objects to maintain a constant skinning vert count per frame (we've implemented a number of techniques that maintained constant budgets for individual features within a frame to maintain consistent performance and latency).

Feature Renderer Entry Points



And you may have wanted to do some global operations per frame – at the start and end of each phase (extract and prepare). So those entry points were also available to feature renderers. There, they could go through every view’s visible nodes (in other words traverse all elements visible in a frame) or do other frame-global operation *for that specific feature*.

As you can see we had a number of entry points. When you multiply those per view, there could be a lot of jobs that we are running at once. How did we deal with this concurrency on the feature renderer level?

Perf & Memory Optimizations

- Organize data and workloads **by frequency of access**
 - Across views, across render objects
 - Share data: save frame packet memory
 - Save perf: perform costly computations once per frequency

DESTINY 

Extract and prepare computations and data are split up by frequency (view, frame, object). This allowed us to share data across different views, across different render objects to save memory in frame packet (ex: only need one copy of skinning transforms for any render objects using the same game object), and performance (only compute skinning transforms once for all render objects using them).

Perf & Memory Optimizations

- Core architecture synchronized access and execution
 - Multiple threads access to shared elements

DESTINY 

The core architecture sets up synchronization primitives to ensure safe multi-threaded access. When feature renderers writers write code for 'extract_per_frame' for example, they don't need to worry that this entry point *will* be executed from different jobs and may write to the same data in frame packet. However, it is important to use a performant synchronization method for this operation since it will be a high-frequency operation per frame.

Perf & Memory Optimizations

- Brute-force locking in tight inner loops murders performance



Early development telemetry

Since thread access has to be controlled per render node based on frequency, we can have tens of jobs vying for the same node (for example, trying to extract skinning matrices for a cloth element on a warlock from player view in one job, at the same time as from a skinned object in the same view in a different job, and maybe from a skinned object in a shadow view). If we just used a regular heavy-weight synchronization primitive we would kill all performance gains from running these operations per-frame. <here> is an example. It is an example of our jobs in an early development build where we used locks to control this synchronization. The <wall of red> that you see is locks. This is awful (it was only there for convenience for a brief moment in time).

Perf & Memory Optimizations

- Brute-force locking in tight inner loops murders performance
- Developed custom lockless primitives for fast shared render node operations



DESTINY 

Instead we developed custom lockless primitives for render node synchronization control. We used an interlocked bitvector and then hashed a render node to a key to use for the synchronization. The wall of red is now gone. One word of caution is that although lockless primitives are good for performance, they are notoriously finicky. When developing these, beware of <schrodinger's bugs> – now they are here, and now they aren't. Timing related bugs due to getting lockless prim logic wrong can be quite challenging to debug and get correct, so patience is required.

Perf & Memory Optimizations

- **Extract | Prepare Per frame operations**
 - Once per frame as long as object is visible in *any* view
 - Lockless synchronization primitives on per-frame node index
 - Unique to all objects registered with the renderer

DESTINY 

For example, we run extract and prepare per frame operations to perform expensive computations that have to happen only once per entire frame as long as this render object is visible in *any* view. For controlling lockless access we used per-frame node index (which was unique to all objects registered with the renderer)

Perf & Memory Optimizations

- **Extract | Prepare Per *Game* object operations**
 - Shared workloads for multiple render objects
 - Skinned extract and prepare/ dynamic AO computation / forward-light probe computation / non-deterministic animation solves
 - Lockless synchronization primitive based on object skeleton hash

DESTINY 

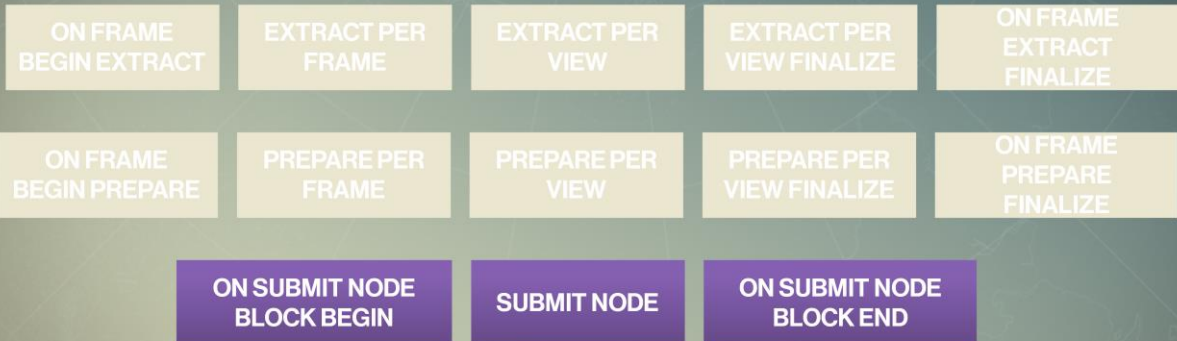
We also run per-game object extract and prepare computations to do workloads that need to be shared across multiple render objects that belong to the same game object. For example, we might have a skinned render object, a cloth and a gear render object all share computations to:

- Extract and transform skinning data
- Compute dynamic AO
- Compute forward-light probe
- Run non-deterministic animation solves (fingers, face bones)

The core renderer provides lockless synchronization primitives for this operation based on skeleton hash

Now let's look at a couple of feature renderer examples.

Feature Example: Static Decals

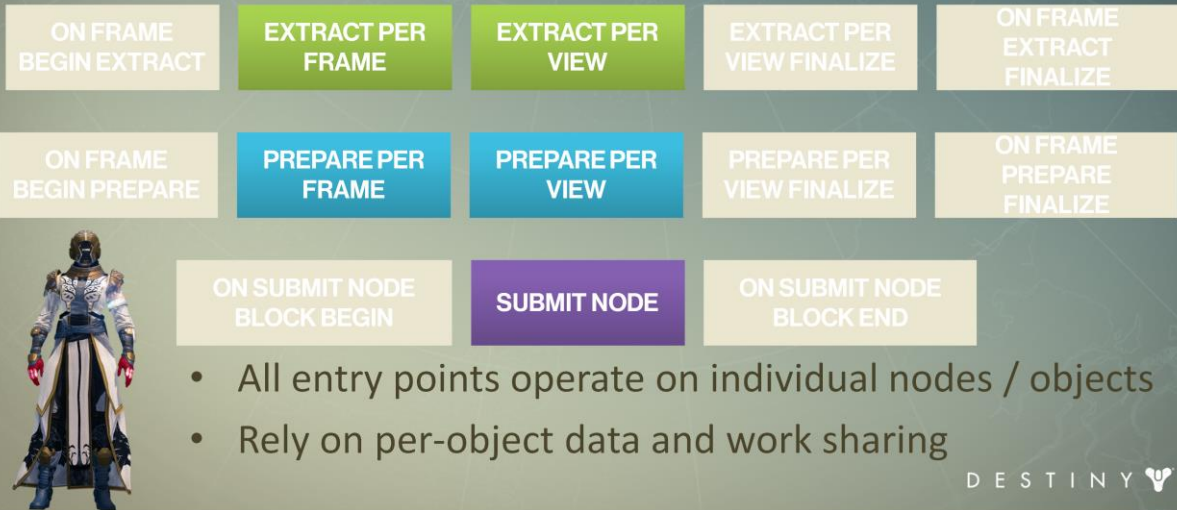


- Submit node works on render node alone
- State block optimization for begin / end blocks

DESTINY 

First a simple one – static projective decals feature renderer `<a><a>`. Static decal renderer does not implement any extract or prepare entry points. It doesn't need to. But it does implement the submit entry points. In the `<submit node>` entry point we execute decal drawcall generation logic. But we render a bunch of decal render nodes at once since they render to G-Buffer stage and we can sort them to execute together. So as an optimization we can setup render states they need in `<"on submit node block begin">` entry point and then we clear these states in `<on submit node block end>` entry points.

Feature Example: Skinned



The next level is the skinned feature renderer. This feature needs to <extract> game data (object properties, skinning transforms) during extract phase. In <prepare>, this feature renderer executes animation solve workloads for non-gameplay affecting bones (fingers, face bones, etc.) based on the object's LOD. This is also where we generate the GPU-registers data for bones (dual quaternions or otherwise) and compute per-object dynamic AO and light probe data. <Both extract> and prepare use render node entry points and rely on <per-object data> and work-sharing. And of course we <render> our skinned objects using submit node entry point.

Advanced Feature Example: Cloth

ON FRAME
BEGIN EXTRACT

EXTRACT PER
FRAME

EXTRACT PER
VIEW

EXTRACT PER
VIEW FINALIZE

ON FRAME
EXTRACT
FINALIZE

ON FRAME
BEGIN PREPARE

PREPARE PER
FRAME

PREPARE PER
VIEW

PREPARE PER
VIEW FINALIZE

ON FRAME
PREPARE
FINALIZE



ON SUBMIT NODE
BLOCK BEGIN

SUBMIT NODE

ON SUBMIT NODE
BLOCK END

- Performs global (all cloth objects) operations
- Rely on per-object data and work sharing

DESTINY 

One of the most complex feature examples we have is cloth feature renderer. This feature renderer implements nearly all entry points our architecture provides. It <iterates> over all possible cloth objects to resets cloth simulation LOD on extract begin since we recompute simulation state every frame, in <extract per-frame and per-view entry> points we extract skinning data, object transforms, and cloth collider data from game state for individual render objects. In <extract per-frame finalize>, we run LOD bucketing to sort all visible cloth instances into high-, low- simulation and GPU skinned buckets to maintain a consistent performance budget per frame, we also determine cloth transition state for each cloth element (are we transitioning to or from GPU skinned state?) and do a number of other cross-object / cross-frame computations.

Advanced Feature Example: Cloth

ON FRAME
BEGIN EXTRACT

EXTRACT PER
FRAME

EXTRACT PER
VIEW

EXTRACT PER
VIEW FINALIZE

ON FRAME
EXTRACT
FINALIZE

ON FRAME
BEGIN PREPARE

PREPARE PER
FRAME

PREPARE PER
VIEW

PREPARE PER
VIEW FINALIZE

ON FRAME
PREPARE
FINALIZE



ON SUBMIT NODE
BLOCK BEGIN

SUBMIT NODE

ON SUBMIT NODE
BLOCK END

- Launches and manages Havok cloth simulation jobs

DESTINY 

In <prepare> entry points we update cloth colliders, gather static and dynamic physics shapes to collide cloths with, launch and run Havok cloth solver jobs, synchronize on final cloth world updates for the whole frame, updated global cloth vertex buffers, and so forth.. And of course we render individual cloth objects in <submit> node entry point.

Adding a New Feature Renderer

1. New feature type enum `my_new_feature`
2. Create `s_my_render_entity`
3. Call `register / unregister` from external systems (object system, etc.) for `my_new_feature`
4. Convert `example_feature_renderer.inl` to use `my_render_entity` (find and replace)
 - Fill in entry points (~5 for basic, ~10 advanced)

DESTINY 

Adding a new feature is easy (and very quick to code). And very important, it is cross-platform.

<Create> a new render feature type (enum)

<Create> render object data structure for your feature type. Typically a simple struct with a few helper methods.

<Add> `register / unregister` calls to your game-side object for that feature

<Create> a new feature renderer by copying an example FR file; convert to use your render object data structure.

Note that for FR we did not use virtual functions for better performance.

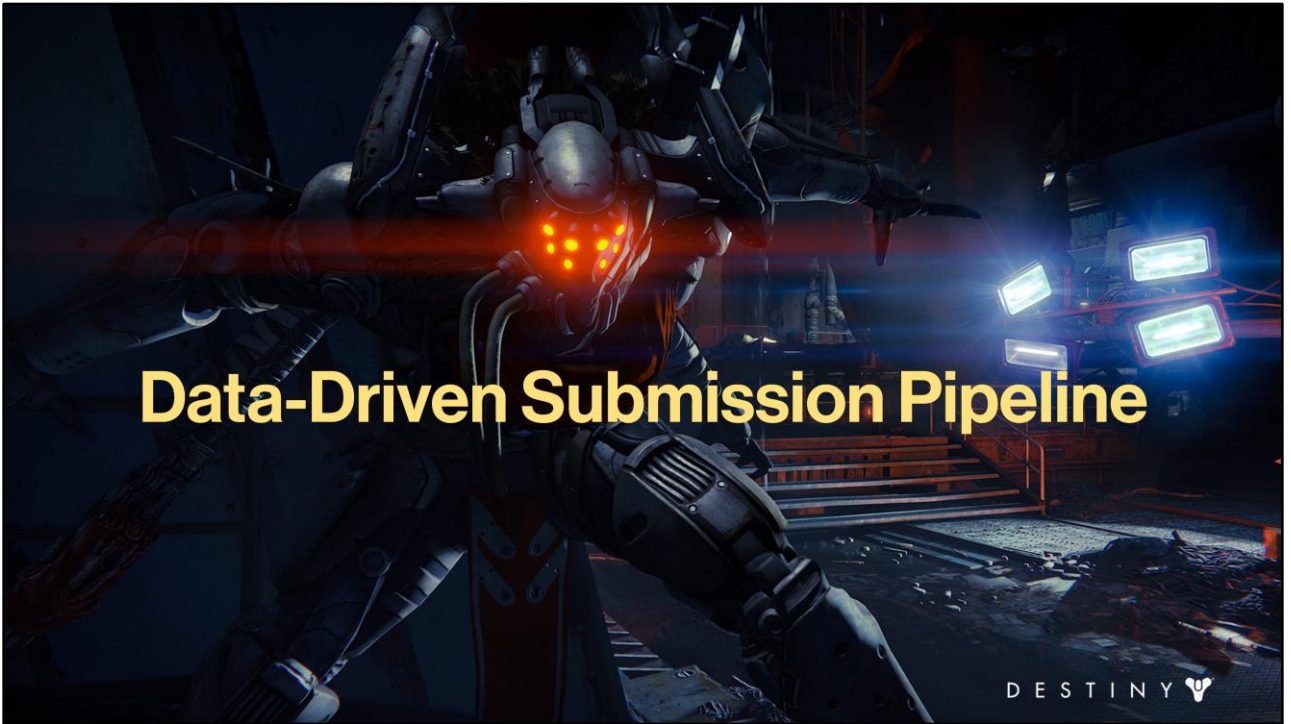
<Implement> entry points for each phase (extract / prepare / submit)

Voila – now you'll have jobified feature

In practice for a new feature renderer (without the actual rendering code) this is about 10 mins of coding. And then you can focus on your data layout, and the submit code.

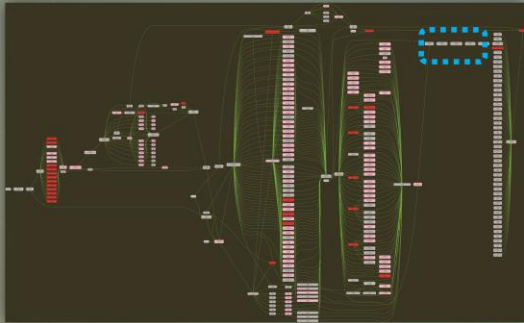
Outline

- Coarse-grained parallelism
- Destiny renderer goals
- Decoupling simulation and rendering
- Jobification for core workloads
- **Data-driven render submission**
- Advanced optimizations
- Conclusions



So far we have a job pipeline that can submit general views. Which would work great if we always rendered with the same shader and the same render target in that view. But what we really want is actually selectively render some objects into render passes automatically selecting the right shader techniques when we want to render that pass. And we want to make this process automatic and super extensible.

Submit



Frame Ring Buffer

Frame Packet

Render Per Frame Nodes



View Packet

Render Per View Nodes



View Packet

Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done

The point of submit is to generate GPU commands for this frame. The first workload we start processing is <the player frame submit script >job. I'll often refer to this workload as "high-level submit script" – but that's a conceptual, name, we are not running an actual script (Lua, etc.) at the moment.

Before we breakdown how that works, let's take a look at how the GPU frame is structured.



If we look at breakdown of a frame, you'll quickly see that we build the frame from <a number of passes> (as you see in this list on the right). Note that these passes are different depending on the state of the game and the content visible.



Atmosphere

We typically start computing GPU updates for particles and other GPU-driven elements.

Then we render the atmosphere pass.



Followed by Gbuffer passes (depth prepass opaque, decals, etc.)



Then shadows – this includes cascade shadow view rendering and application, or local dynamic light shadows and application.



Followed by the lighting pass



Then we render additive decals



And transparents



We render geometry to light shafts occlusion buffer



We punch everything up by rendering lens flares



And that, ladies and gentlemen, is our frame.
So what does it take to submit this frame?

Frame Submission

- High-level submit code executes passes – Branchy, based on state of the game

GPU updates
Atmosphere
Gbuffer Depth Prepass
Gbuffer Opaque
Gbuffer Decals
Gbuffer Decals Additive
Lens Flares Occlusion
Shadow views generate and apply
Light Probe Lighting
Deferred Lights Pass
Subsurface Scattering
HDAO Pass
Shading Pass
Water and Transparents
Lens Flare and Gunk Render
Postprocess

DESTINY

The high-level submission of the overall frame is branchy, complex code with dynamically modified execution based on the state of the game.

We need to figure out how to setup our player view based on whether the game is in first-person or third-person mode. We may have special rendering paths for cinematics, with additional subsurface scattering passes, or higher quality post-processing effects.

Frame Submission

- High-level submit code executes passes
- Sets up infrequent but heavy-cost global state
 - Render targets operations
{ Bind | Clear | Resolve | Decompress |... }
 - Global states (frame / view registers)

GPU updates
Atmosphere
Gbuffer Depth Prepass
Gbuffer Opaque
Gbuffer Decals
Gbuffer Decals Additive
Lens Flares
Occlusion
Shadow views generate and apply
Light Probe Lighting
Deferred Lights Pass
Subsurface Scattering
HDAO Pass
Shading Pass
Water and Transparents
Lens Flare and Gunk Render
Postprocess

DESTINY


High level frame submit 'script' executes high level rendering directives. It interleaves global state with high-level submit directives to submit the frame. Global state setting generates direct GPU commands into the main GPU command ring buffer; inserted directly. This includes:

- Render target operations (alias / bind, clear / resolve / decompress)
- Fframe registers: engine time, etc. and view state (view projection matrix, camera location, etc.)

Frame Submission

- GPU updates
- Atmosphere
- Gbuffer Depth Prepass
- Gbuffer Opaque
- Gbuffer Decals
- Gbuffer Decals Additive
- Lens Flares
- Occlusion
- Shadow views generate and apply
- Light Probe Lighting
- Deferred Lights Pass
- Subsurface Scattering
- HDAO Pass
- Shading Pass
- Water and Transparents
- Lens Flare and Gunk Render
- Postprocess

- High-level submit code executes passes
- Sets up infrequent but heavy-cost global state
- Each pass calls **submit view directive**
 - Simple, cross-platform API
 - Generate GPU commands for that view

DESTINY 

During each high-level submit script pass, we issue “submit view” directives. This is a call to the core architecture which is designed to generate GPU commands for this view. From the perspective of folks writing the high-level submit script, this is all they need to do. They don’t worry about jobifying submit directives manually. Another benefit is the submit view directives are cross platform. Which means that as we had to adjust job granularity, batching rules, and occupancy behavior for each platform, the high level feature code was not affected.

Frame Submission

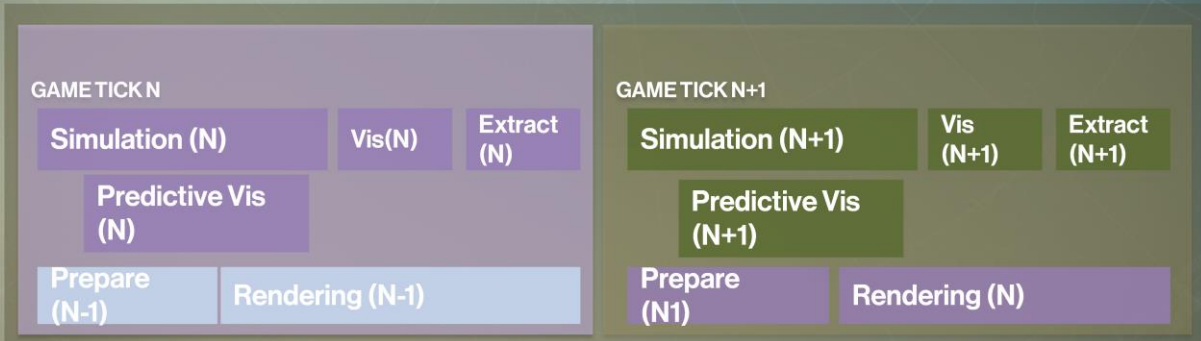
- GPU updates
- Atmosphere
- Gbuffer Depth Prepass
- Gbuffer Opaque
- Gbuffer Decals
- Gbuffer Decals Additive
- Lens Flares Occlusion
- Shadow views generate and apply
- Light Probe Lighting
- Deferred Lights Pass
- Subsurface Scattering
- HDAO Pass
- Shading Pass
- Water and Transparents
- Lens Flare and Gunk Render
- Postprocess

- High-level submit code executes passes
- Sets up infrequent but heavy-cost global state
- Each pass calls submit view directive
- Each submit view directive sets up **submit view job(s)**
 - Jobification rules varied per platform

DESTINY

Under the hood, the core renderer architecture converts the submit view directive to a set of submit view jobs. The rules for converting to jobs varied per platform due to our desire to have optimal GPU occupancy and keep game latency low. I will talk about that in a few minutes.

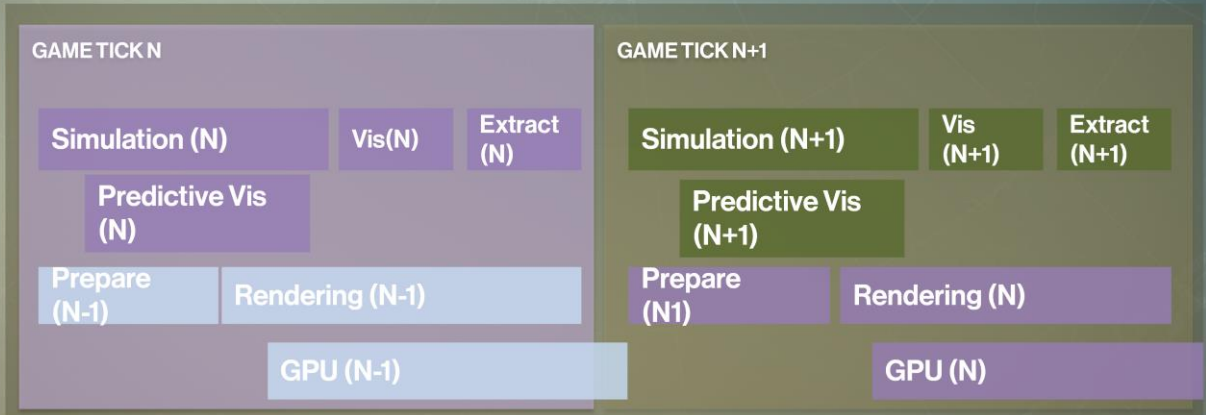
Pipelining GPU Work



DESTINY 

If we come back to our previous diagram for our pipelined execution, where does GPU fit into the picture?

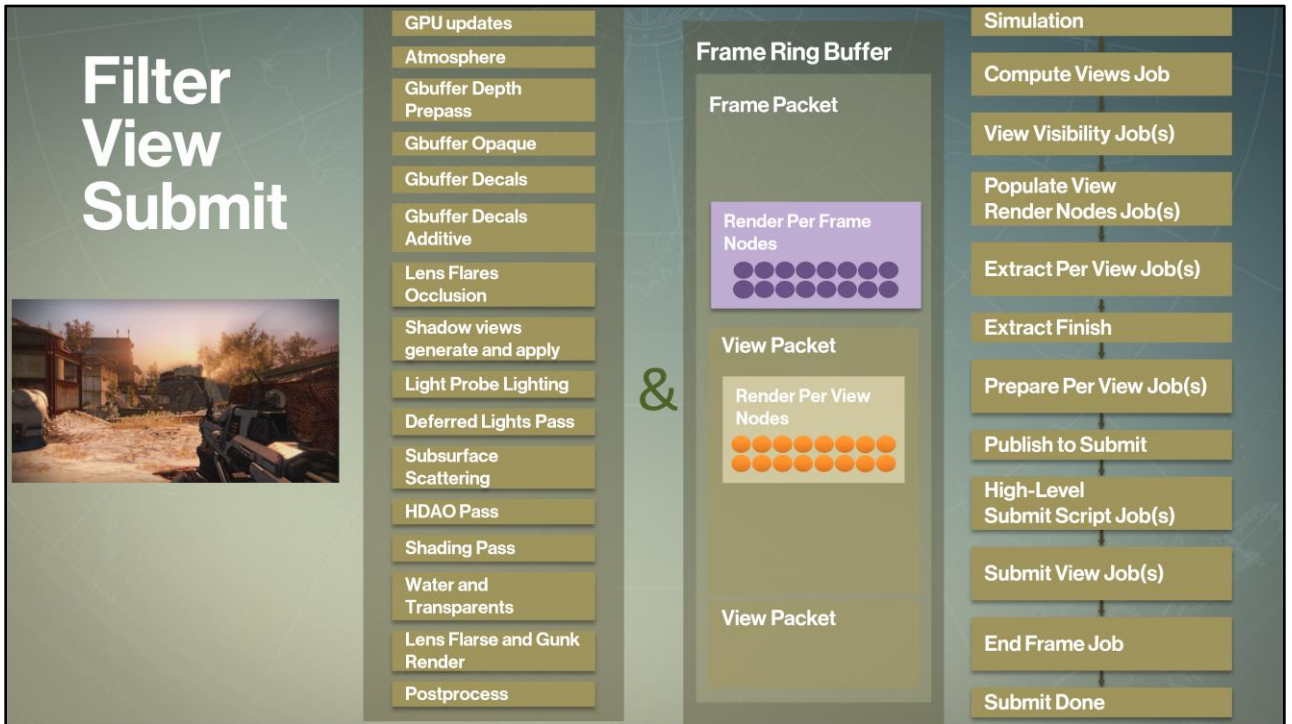
Pipelining GPU Work



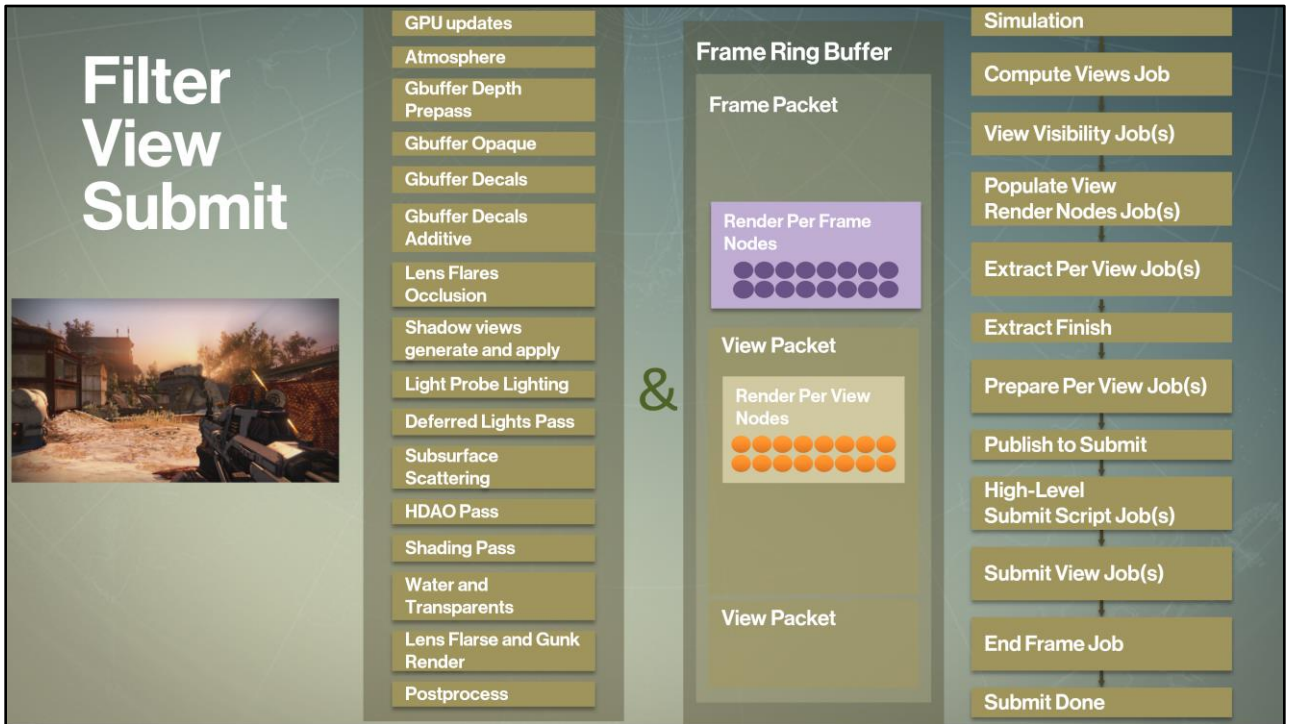
What we want to do is thread the <rendering> execution to start feeding the <GPU> right away to have GPU crunch of the work as quickly as possible so that we can be ready to flip at the very next vblank event. I'll mention a few things we did for that in a later section.



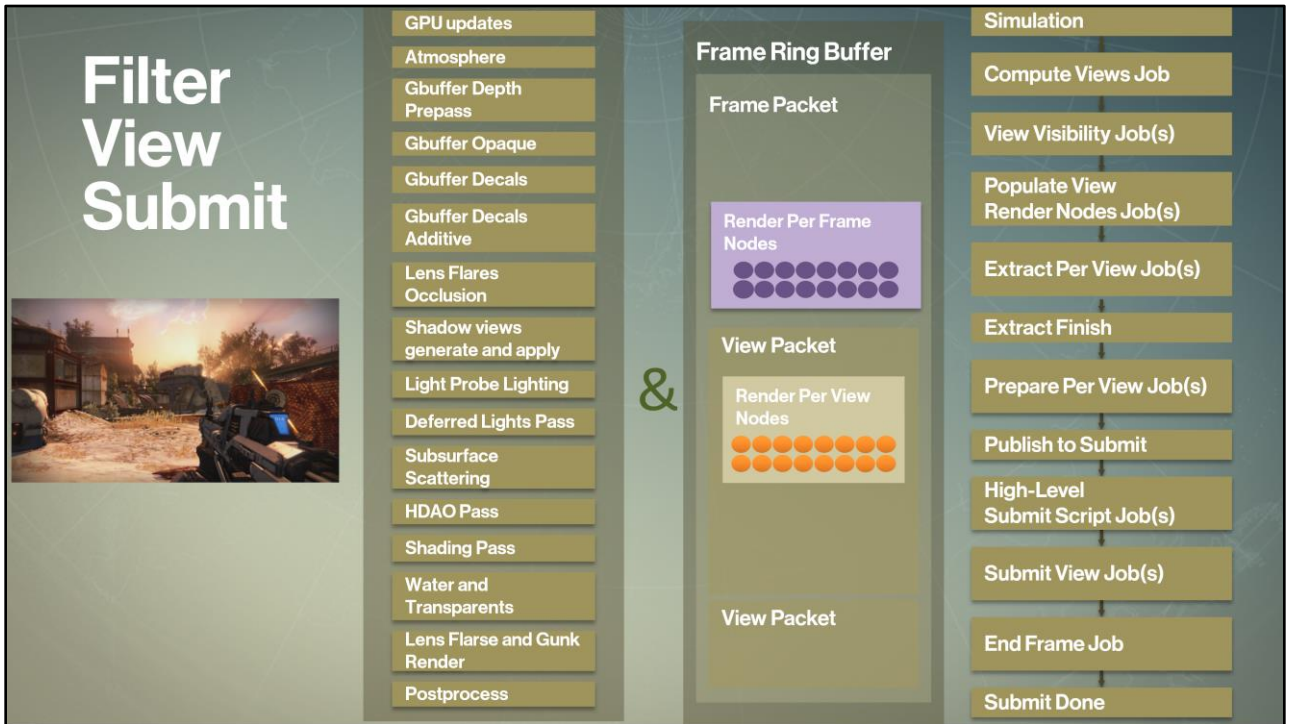
Ultimately, what we want to do is figure out how to fire up <submit view jobs> and feature submit entry points, from the high-level submit pipeline. What we need to do is distinguish which of the visible elements need to be submitted to a decals stage or which need to be submitted to transparents stage



For example, we already know that some of our main player view's visible elements render to G-Buffer pass



Other render as additive decals



And yet other elements for the same view render as transparents.

How do we filter the visible nodes in the view packet for our streamlined feature renderer kernel submission?

High-Level Submit Control

- Cross-platform code path
- Organized in phases

GPU updates
Atmosphere
Gbuffer Depth Prepass
Gbuffer Opaque
Gbuffer Decals
Gbuffer Decals Additive
Lens Flares Occlusion
Shadow views generate and apply
Light Probe Lighting
Deferred Lights Pass
Subsurface Scattering
HDAO Pass
Shading Pass
Water and Transparents
Lens Flare and Gunk Render
Postprocess

DESTINY

In Destiny's renderer, the high-level submit control code is cross-platform (with some exceptions for dealing with surfaces due to ESRAM / EDRAM / main / GPU memory particulars).

The high-level submit code is organized in phases, where each phase is a sequential combo of render passes and render stage submission directives in our system. The distinction between the two is whether it is required or data-driven – i.e. only executed if the right data is present in the pipeline.

High-Level Submit Control

- Cross-platform code path
- Organized in phases
 1. Required passes
 - Shading pass
 - Tone mapping / resolve
 - ...
 2. Render stage submission directives

GPU updates
Atmosphere
Gbuffer Depth Prepass
Gbuffer Opaque
Gbuffer Decals
Gbuffer Decals Additive
Lens Flares Occlusion
Shadow views generate and apply
Light Probe Lighting
Deferred Lights Pass
Subsurface Scattering
HDAO Pass
Shading Pass
Water and Transparents
Lens Flare and Gunk Render
Postprocess

DESTINY

Certain render passes are required to render regardless of any data present in the pipeline – for example, the shading pass, the tone mapping and resolve pass, etc. Those are not considered render stages in our system.

Render stage directives are a high-level command to execute submission for a specific view for a specific stage of the frame pipeline where we might have content-driven data (for example, G-buffer or transparents, or shadows elements).

High-Level Submit Control

```
generate_gbuffer_pass()  
  set_render_targets(depth_stencil, gbuffer_surfaces)  
  setup_viewport_parameters()  
  ...  
  clear_viewport()  
  submit_render_stage_for_view(first_person_view, _render_stage_gbuffer_opaque);
```


DESTINY 

Here's an example of pseudocode (fairly close to the original) of what our high level submit code looks like.

What we will talk about here is <this> part

Render Stage Mechanism

- Filtering runtime submission passes
- Provides runtime shader technique management
 - Select the right technique at the right time
- Allows filtering visible list
 - Selects right sets of meshes to submit to the right pass using the right technique
- Built for cache coherency and fast iteration

DESTINY 

Render stage is our mechanism for filtering runtime submission for passes and views. At its core it is about shader management (selecting the right techniques at the right time) and filtering the visible list in a given view (which comes down to mesh filtering for actual drawcall generation using the right shader we've just selected). Of course we want it to be automatic and transparent to leaf feature writers. Render stage has also been design to improve cache coherency and enable fast iteration for core renderer workloads.

How do we go about it?

Render Stage

- Each render object can *subscribe* to render stages at any time
 - By registering with the core renderer for stages
 - At registration or any time at runtime

DESTINY 


Each render object can subscribe to a render stage. To do that, it registers with specific render stages when upon the registration with the renderer. This subscription can either be computed during offline importing or ...



... registered dynamically at runtime. For example <when we engage a super>, we spawn effects on the character. We register the new character's transparent FX elements for the character render object w/ the transparent render stage to render these new dynamic effects with the character's feature renderers when we invoke transparent render stage high-level submission passes.

Render Stage

- Each view can subscribe to render stages at runtime
 - Visible list is filtered to only subscribed render stages

DESTINY 

How do we filter the list of visible elements in a view for each stage?

A view can specify whether it subscribes to a render stage or not. If the view doesn't subscribe to the stage, it won't render any visible elements for that stage.

Render Stage

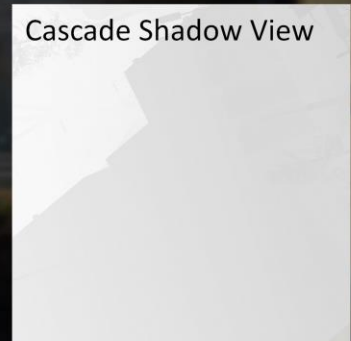
- Each view can subscribe to render stages at runtime

Shadow Generate
Render Stage

Cascade Shadow View



Cascade Shadow View



DESTINY 

For example, a shadow view only supports shadow generate stage and thus renders only the render objects that have subscribed to shadow generate render stage (i.e. shadow casters) that are visible in each view (respectively)

Render Stage

- Each view can subscribe to render stages at runtime

Gbuffer Opaque
Render Stage



but the main player view supports variety of render stages (gbuffer...

Render Stage

- Each view can subscribe to render stages at runtime

Gbuffer Decals
Render Stage



...decals...

Render Stage

- Each view can subscribe to render stages at runtime




Transparents
Render Stage

DESTINY 

...transparents, etc.).

Filter Views by Render Stages

- Render object submit will only be called if both are true:
 - The view subscribes to the given stage
 - And*
 - The view contains visible nodes that subscribe to that render stage

DESTINY 

When we execute the submit view directives in our high level submit script, the core renderer architecture figures out for any given view whether we should execute anything for that stage by checking:

- <Does this view support this render stage?>
- <and>
- <Do any of the elements visible in this view support this render stage?>


I Want to Be Transparent!

- Shader metadata specifies render stage in shader code

```
technique default
{
  @render_stage(generate_gbuffer);
  ...
}

technique default_shadow_generate
{
  @render_stage(shadow_generate);
  ...
}

technique depth_prepass
{
  @render_stage(depth_prepass);
  ...
}
```

DESTINY 

How do we get information about render stage subscription for render entities? Each shader specifies render stage property in its metadata. Here is an example of how we do that in source code. (We wrote a custom shader language format for Destiny).

I Want to Be Transparent!

- For each object during offline importing
 - Iterate over all meshes
 - Find all shaders
 - Extract render stages assigned
 - Build lookup containers
- Render stage mesh runtime lookup in $O(1)$
 - Fast filtering of drawcalls for render stage at runtime

DESTINY 

We iterate through all shaders offline when processing meshes for each platform, and create containers that allow constant time look up at runtime to get all mesh parts for a given render stage.

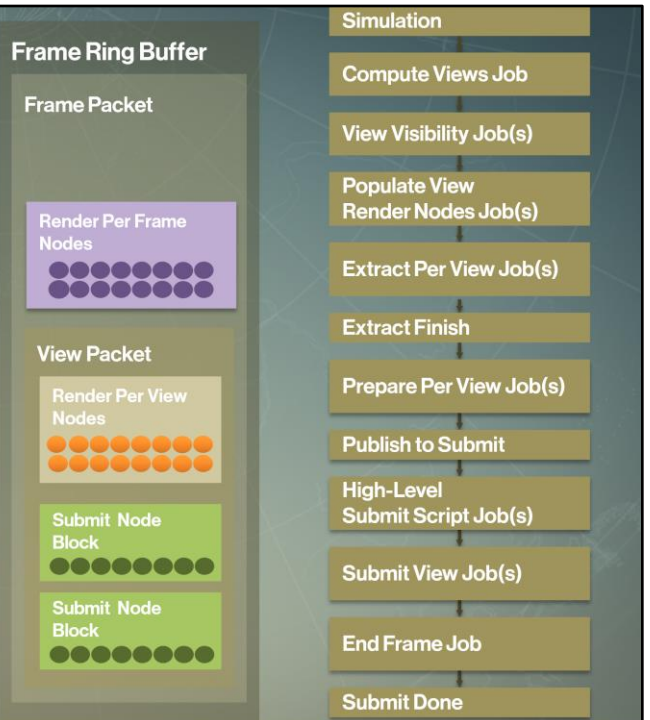
This let's us have super quick submit filtering for drawcalls when processing a submit node based for a render stage.

Each render entity subscribes to stages at runtime by either using the static offline-generated data or by using dynamic runtime state.

Next, let's look at how render stage filtering flows through our pipeline in an efficient manner.

Populate Submit Nodes per Stage

- Filter visible list by render stages

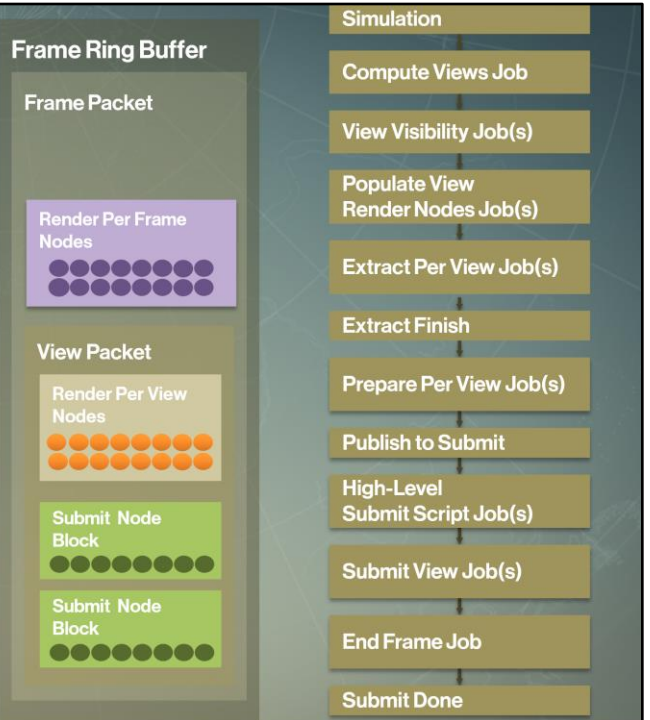


After visibility returned us a visible list, the <populate view render nodes> job will run through the list of visible nodes for that view and sort them by render stages that they support (as well as render feature type). We actually cache the render stage subscription in core renderer objects and during visibility when we add objects to the visible list we automatically build counts for each render stage (better cache coherency). Effectively this filters visible nodes by stages they support for each submit node block

<For each render stage>, the core architecture then creates a <submit node block>. A submit node block is just a block of render nodes that should render in that view | render stage pass.

Sort Submit Nodes Per Stage

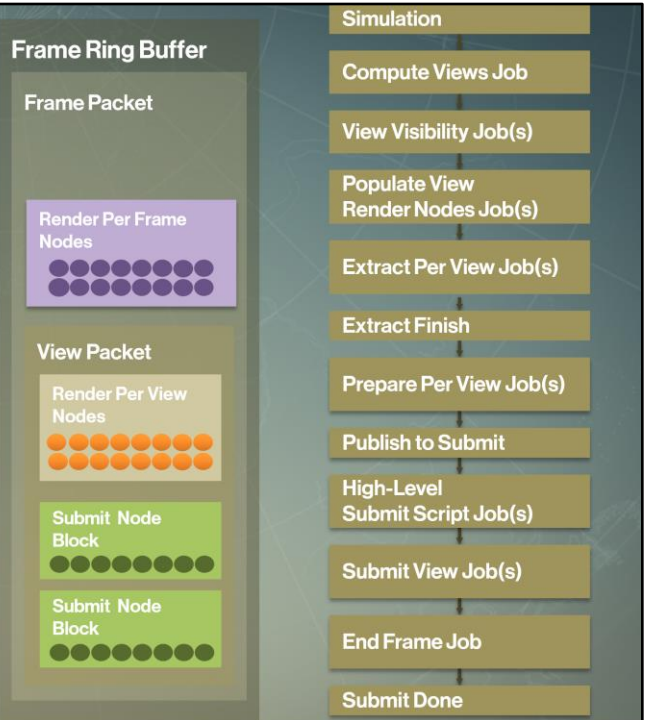
- Certain passes required sorting prior to rendering
 - Transparents, etc.
- Sort during prepare per stage
- Use configurable callbacks per render stage
- Each render node computes custom per-stage heuristic at runtime



When we submit certain passes, we may want to render objects in certain order. For example, transparents need to be rendered back to front. Or we may want to render G-buffer objects in groups of the same type, to have coherent execution. We do this sorting using submit nodes. Each render object writes custom per-stage sort heuristic key into the submit node. Each render stage has custom sort callbacks. Sorting is executed during prepare.

Sort Submit Nodes Per Stage

- Each render object can have one or multiple submit nodes
- Inlined fast sort – based on tiny sort keys



It's important to note that each render object can create as many submit nodes per stage as it wants. This is very handy if you want to sort individual drawcalls for transparents for a given object, instead of just sorting by object. If you want to sort by individual drawcalls, go ahead.

The sort calls themselves are extremely fast since they are operating on tiny elements in cache-coherent submit node blocks.

Cache Coherency Note

- All core renderer jobs operate on the render nodes
 - Tiny data structures
- **Fast, cache-coherent** sorts, allocations, traversal for all core workloads
 - Small data duplication for faster cached access during tight iteration loops

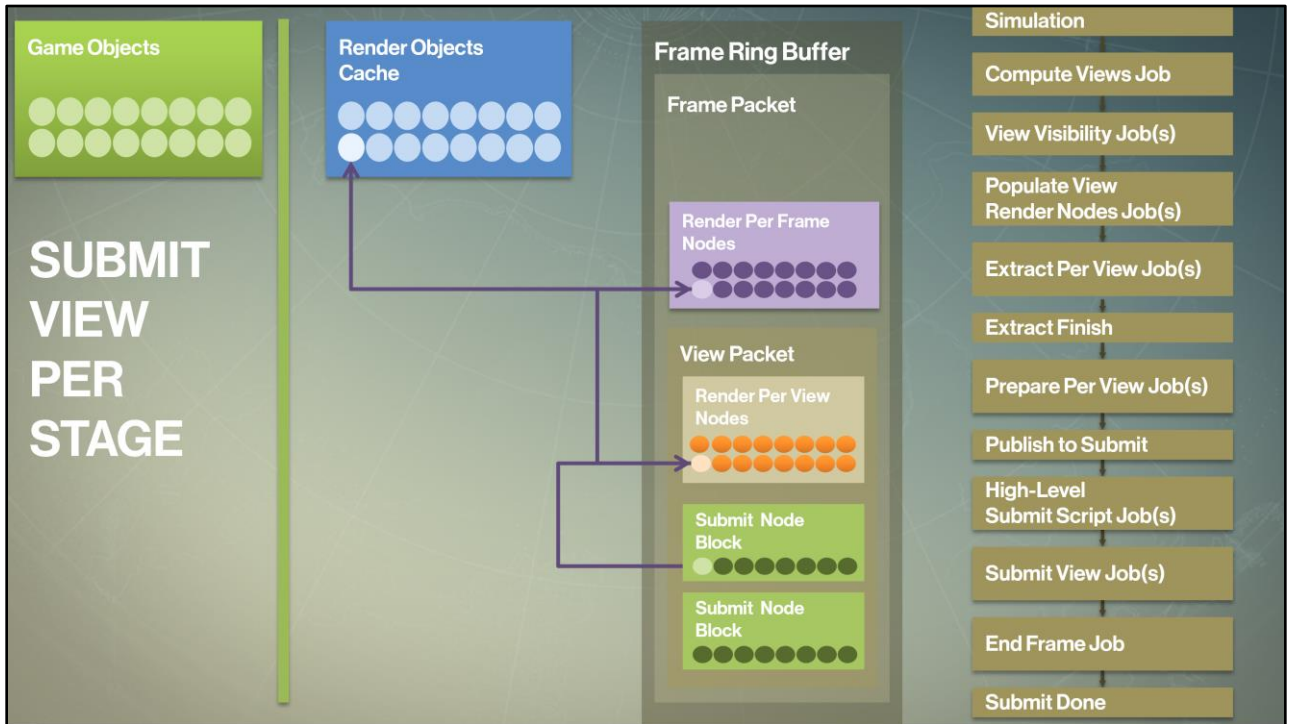
```
sizeof(s_submit_node)      24
sizeof(c_render_per_view_node) 48
sizeof(c_render_per_frame_node) 24
```

DESTINY 

All core renderer jobs operate on the render nodes – <which> are tiny data structures (here are the sizes in bytes for each node type)

This ensures that all core renderer workloads are fast, cache-coherent operations (sorts, allocations, iteration for each stage by the core feature renderer interfaces).

Note that a small percentage of the data is duplicated in each node type for faster cached access during tight iteration in the core workloads.



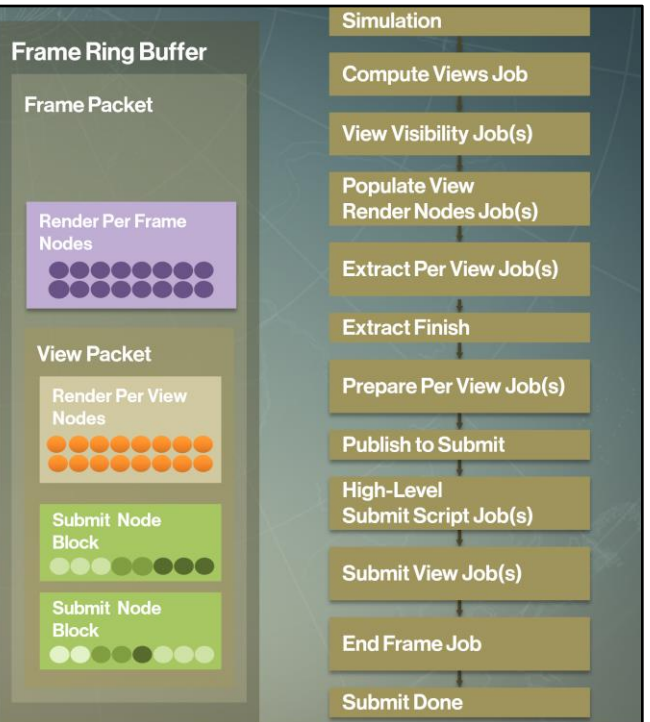
Now when we say “render first cascade shadow view” the core system interprets that as “generate <submit jobs> to render first cascade shadow view packet’s shadow generate render stage submit nodes block”.

Internally, the renderer grabs the appropriate <submit node block> for the view and render stage we’re submitting. Then for each submit node (i.e. a visible node for that stage), we execute the corresponding render object’s submit entry point, using <the view node>, <the frame> and the <render object’s> cached data to generate the drawcalls.

<And, like prepare, submit> also cannot access game state because simulation is running at the same time.

Submit View Per Stage Jobs

- Split up into separate submit leaf jobs
 - Per stage | view | feature | set of nodes
- Dynamically load balance the split



The core system actually splits up the high level directive into separate jobs, for example, grouping one set of nodes together into a single job, while another grouping gets batched into a different job (I used different colors for submit nodes here to indicate example batching). Within each job, we still execute feature renderer's submit kernels for individual render nodes. We'll talk about how the batching works in a few minutes in a later section on dynamic load balancing.

Submit View Per Stage

- For each feature submit entry points, access only local and render object data
 - Render nodes
 - Feature render object cache



Frame Ring Buffer

Frame Packet

Render Per Frame Nodes



View Packet

Render Per View Nodes



Submit Node Block



Submit Node Block



Simulation

Compute Views Job

View Visibility Job(s)

Populate View
Render Nodes Job(s)

Extract Per View Job(s)

Extract Finish

Prepare Per View Job(s)

Publish to Submit

High-Level
Submit Script Job(s)

Submit View Job(s)

End Frame Job

Submit Done


This way, feature renderer submit abstracted inputs to each submit kernel to restrict knowledge only to the specific render entity's local state for this submission:

- The render object itself, its locally cached data and it's dynamic data in the frame packet
- The specific submit parameters (view, stage, etc.)
- But not the global state (bound surfaces, etc.)

Leave the state in the same way as when you can in.

Feature Renderer Submit Kernel

- Coherent code paths for each feature
- Can be sorted by feature type
 - For passes that allow it
- **Submit view jobs are streamlined kernel processors**

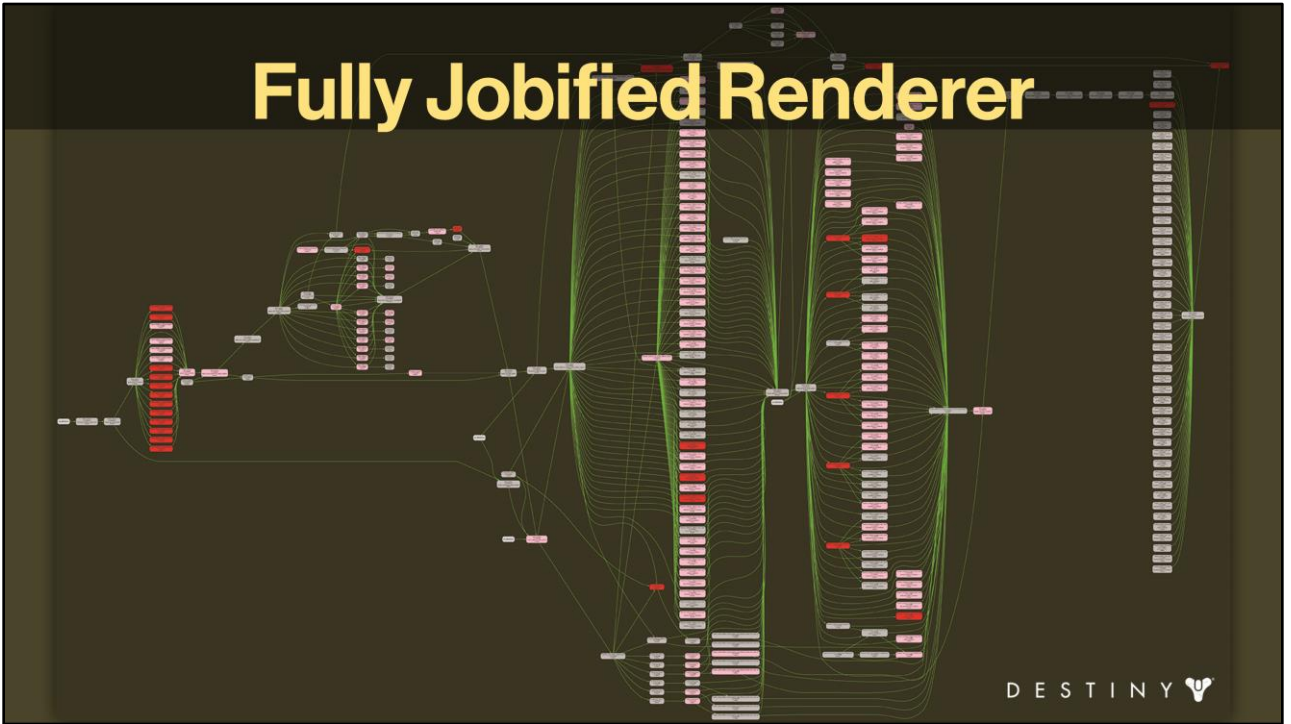
DESTINY 

So feature submits are already grouped by same code paths. We can then sort the submit node blocks by feature type if the render stage allows it (ex: G-Buffer or shadows). We are only executing using local state using specific stage's kernels. This sure looks like GPU pipeline, right? All of this allows us run coherent execution in that stage's job. In a way, we have so much to thank SPU for, don't we? 😊

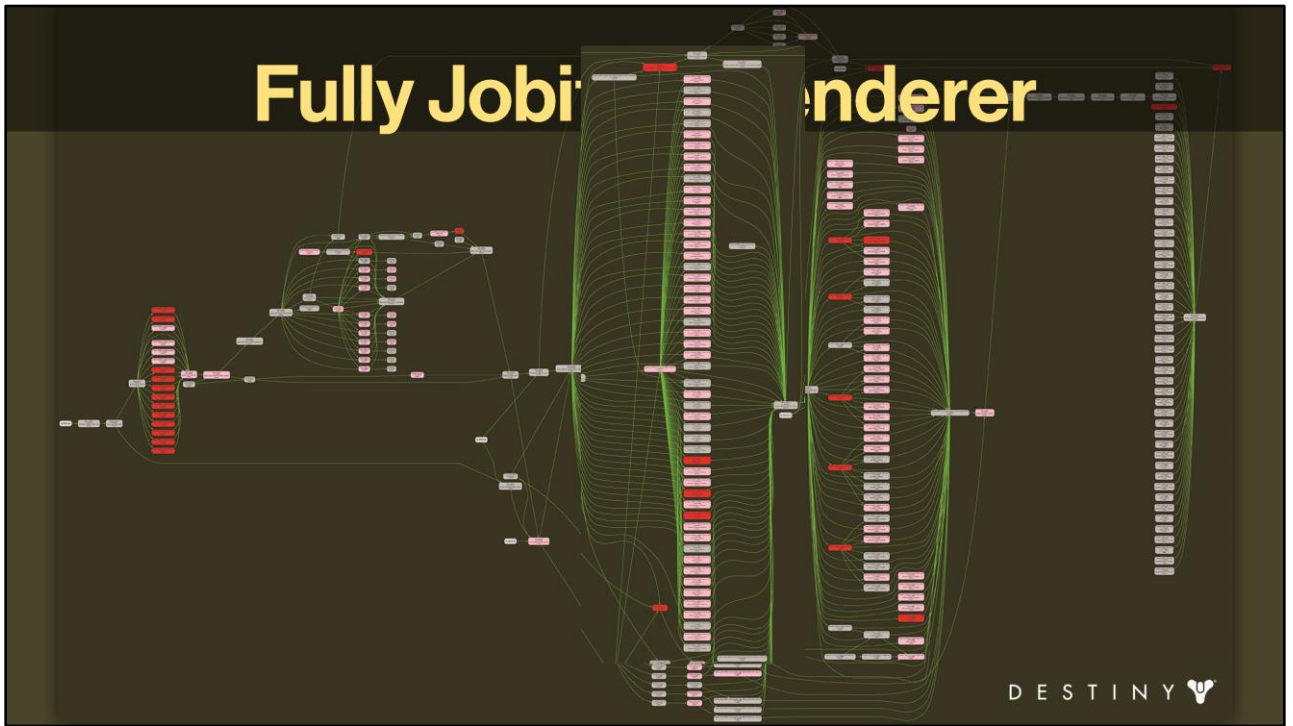
You can see that this maps well to data-parallel processing.

Our submit jobs became fully streamlined kernel processors.

Fully Jobified Renderer

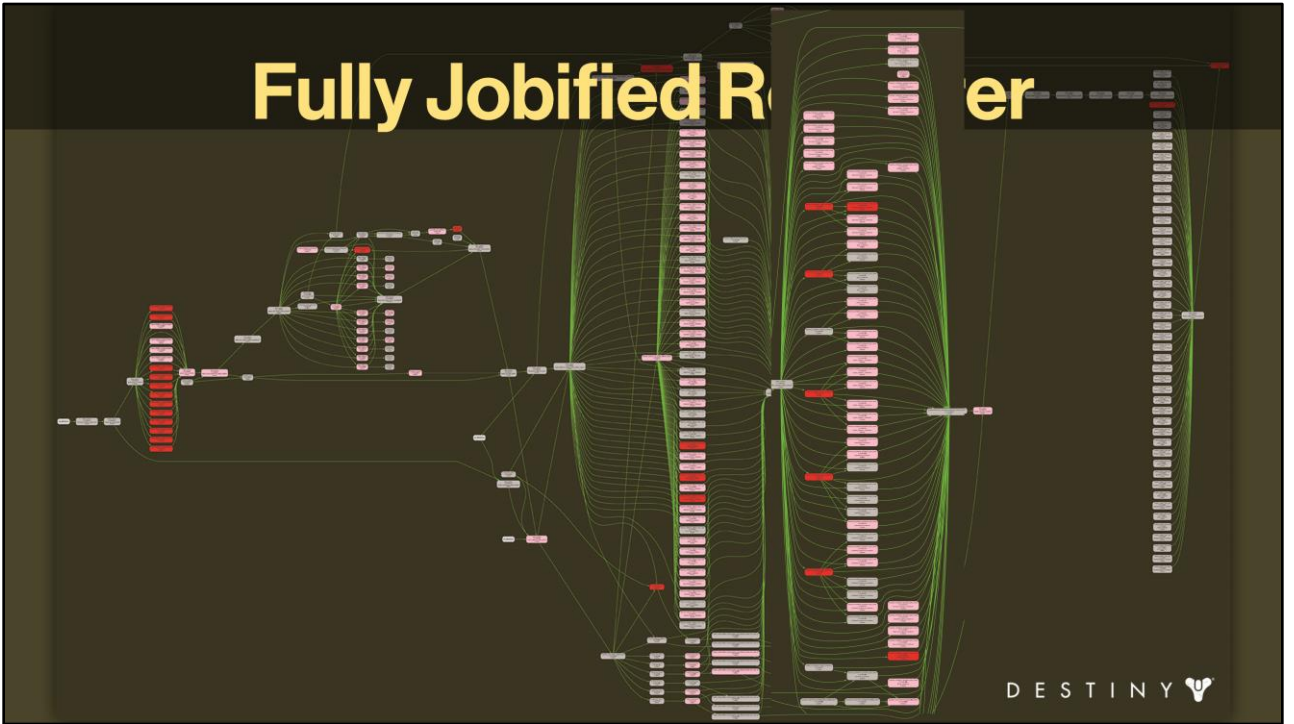


When you put this together across all views, across all render stages and render features, here you have it - our full jobified renderer pipeline.



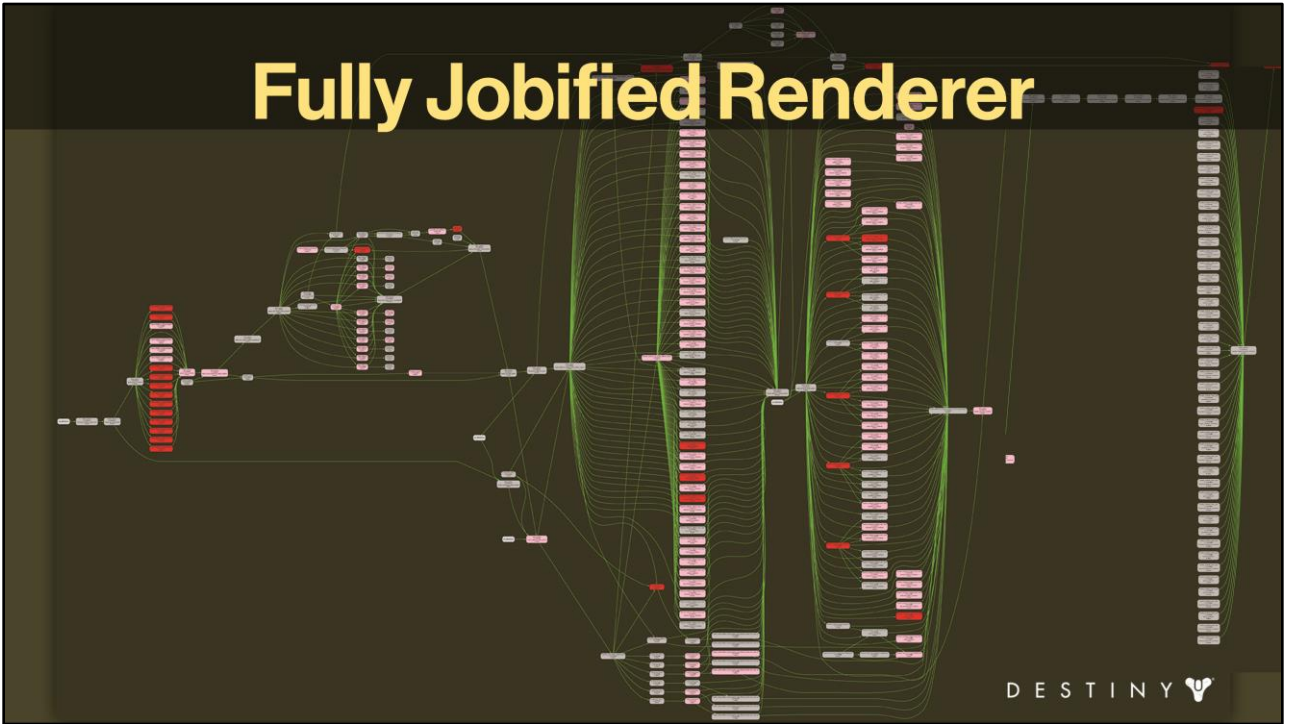
All the render features' jobs are going wide across <extract> ..

Fully Jobified Rer



<prepare> ..

Fully Jobified Renderer



<and submit phases> ..

Outline

- Coarse-grained parallelism
- Destiny renderer goals
- Decoupling simulation and rendering
- Jobification for core workloads
- Data-driven render submission
- **Advanced optimizations**
- Conclusions

DESTINY 

And now that we have the core architecture established, we can start adding bells and whistles to improve it. And the nice part about getting to this point is from now onward – **no feature code needs to change**. Throughout the project, we had tweaked, pruned, batched, and otherwise massaged the job dependencies, synchronization, occupancy in the frame. And yet none of the leaf render feature code was affected and needed to be touched. And that's the way I like it. 😊

So, what other cool stuff can we do to make our architecture even more effective?

Render Jitter & Latency Reduction

- Multi-threaded submit
- CPU / GPU load-balancing for reducing GPU bubbles
- Efficient command buffer flushing
- Asynchronous swapping
- Jitter reduction with latency auto-recovery

DESTINY 

We had to implement a number of techniques to reduce render jitter and latency reduction. We won't have time to go into deep details for these, but I wanted to at least mention them as they were crucial for robust performance. To keep the GPU maximally occupied and our game latency minimal, we multi-threaded our submit, implemented custom dynamic CPU & GPU load balancing to get rid of GPU bubbles. We implemented efficient command buffer flushing to get the generated commands to the GPU as soon as possible. We also used asynchronous swapping on all platforms that allowed it, and implemented jitter reduction technique with latency auto-recovery which worked great and reduced the overall latency significantly.

But with all of this work happening under the hood, none of the leaf features were affected!.

GPU Occupancy Factors

- Ultimate goal: keep GPU fully saturated
 - Never idle
- Important factors
 - Time when flush GPU commands to the GPU
 - Drawcalls
 - Execute command lists directives
 - Render target state
 - Previous frame's work on GPU complete and flip

DESTINY 

We need to keep GPU fully saturated with work. This means we had to mind the following:

- When did the CPU generate the GPU commands and flush them to the GPU to work on? If we take a long time on CPU to generate and flush GPU commands to the GPU, we can starve the GPU.
- When did GPU finish working on our previous frame? We need to make sure to time our submission correctly. This also helps us avoid problems with running out of space for GPU command ring buffer (on more memory constrained platforms).

Tracking GPU Idle

- Counters are spotty but available on some platforms
- Built that our engine to help track
 - For all performance and activity testing
- GPU idle == starved GPU
 - Helps pinpoint where latency is higher because GPU is starved

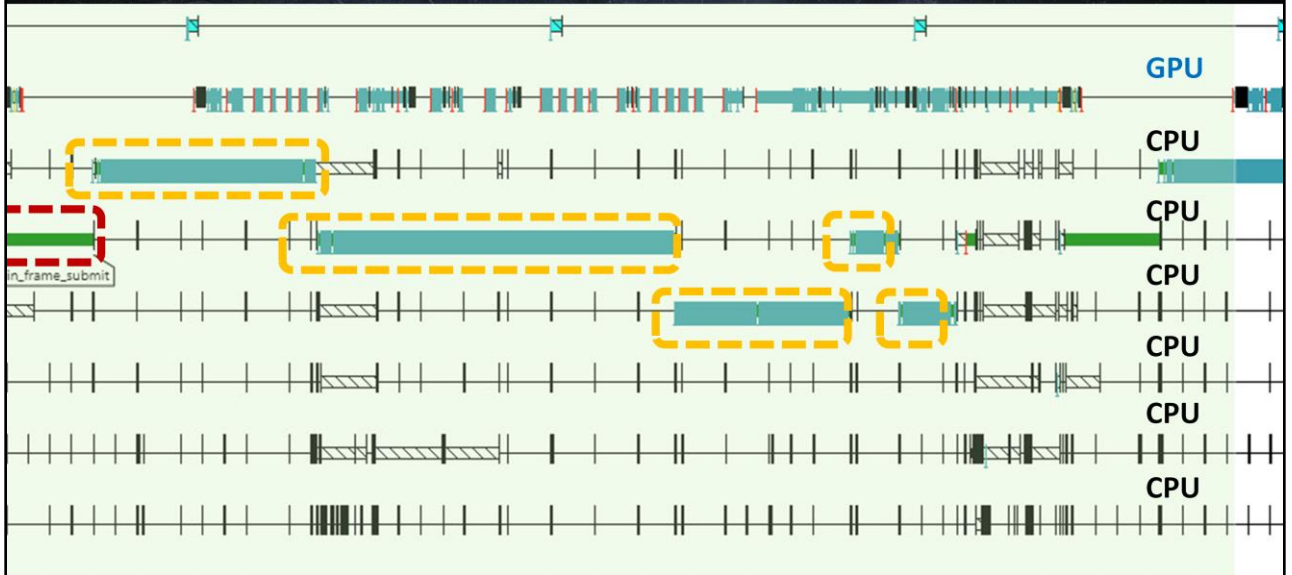
DESTINY 

To help answer a lot of these questions, we implemented custom tracking of GPU idle in our engine.

While exposed counters for querying GPU idle state on various platforms are spotty and not super reliable, we still used what we could get our handles on.

We tracked GPU idle information for all of our performance runs and used this data to attack latency reduction. Whenever you saw GPU idle rear its head strongly, you know that you are starving your GPU.

GPU Starvation Example

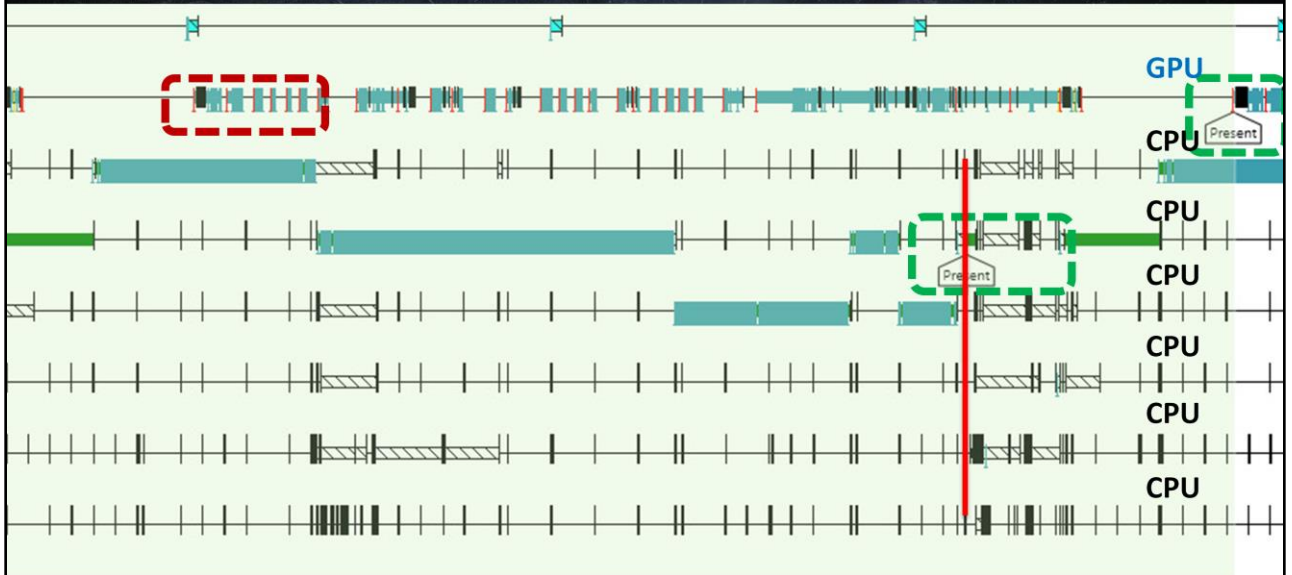


Here is a timing capture from our profile Xbox One build using Microsoft PIX using in-development mode.

In this case, we are submitting the GPU command buffers using serialized submit. Now in our world this still fell into a small set of jobs but everything is submitted using a single threaded device.

We begin CPU submission <here>. Then <we execute> serialized CPU submit jobs next.

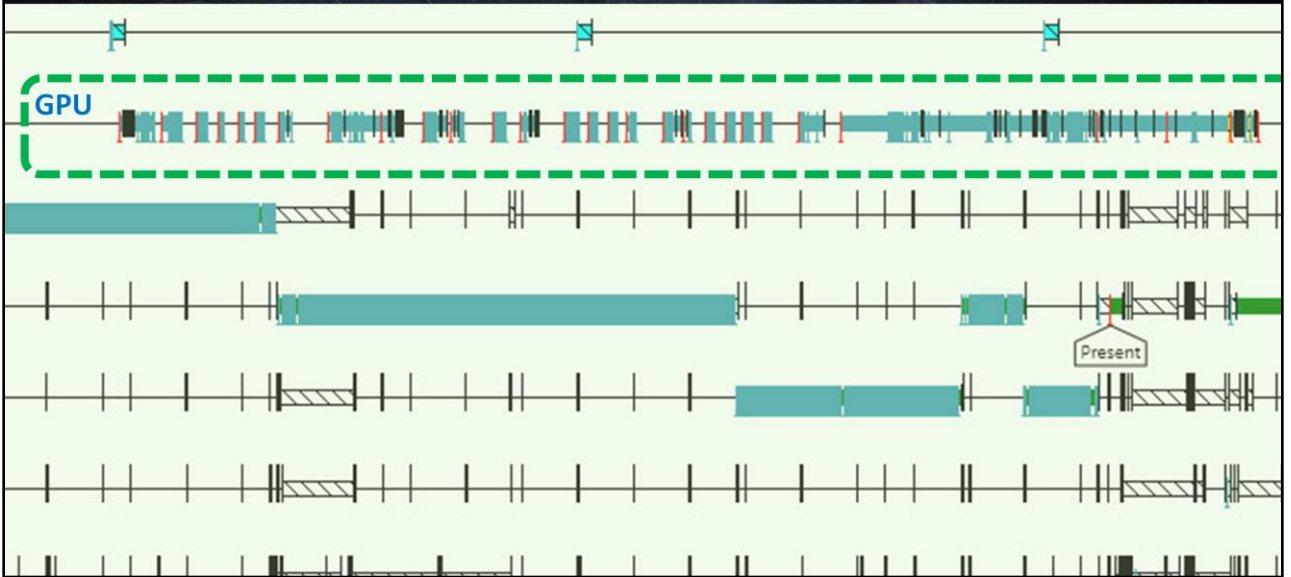
GPU Starvation Example



You can see that GPU starts working on submitted drawcalls <here>. The plus side of serialized execution is that you can flush immediately to the GPU.

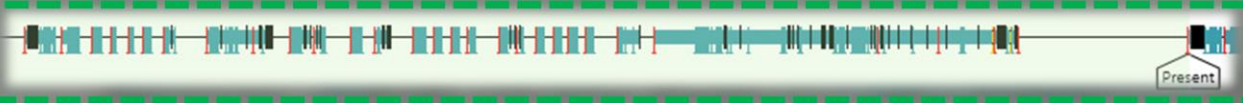
<Here> is we are done with all of our drawcall generation and we submitted present and swap directives. We're done with all the submit jobs <here>. Single-threaded submit takes a very long time to execute on CPU [bad for game latency]. And finally, the present call gets processed on the GPU <here>.

GPU Starvation Example



But what you can also clearly see here is that there are a <TON of GPU> idle bubbles in the GPU timeline.

GPU Starvation Example



- Lots of GPU idle bubbles
- Indicates GPU starvation
- Increases overall rendering latency

BAD.

which indicates that the GPU was starved and did not have work to execute. This means that our overall time to process the rendering of the frame has been increased, negatively affecting the rendering latency. We need to fix that to ship.

Keeping GPU Occupied

- Multi-threaded submit
- Efficient generated command buffer flushing
- Submit job granularity load balancing
- Vblank synchronization

DESTINY 

While optimal GPU occupancy is a very complex subject, these four pieces were crucial to our shipping plan:

- Multi-threaded submit
- Efficient generated command buffer flushing
- Submit job granularity load balancing
- Vblank synchronization

Synchronize to VBlank

- Offset your computations based on vblank
 - In order to get GPU work complete when we're ready to flip without missing an interval
- Some consoles allow raw vblank event tracking
 - Great for complete control
- Other consoles / PC do not and require manual work
 - Spin up a separate thread, wait for Vblank event
 - Beware of Present deadlock when resize / fullscreen event posted

DESTINY 

Offset your computations based on vblank

In order to get GPU work complete when we're ready to flip without missing an interval

Some consoles allow raw vblank event tracking

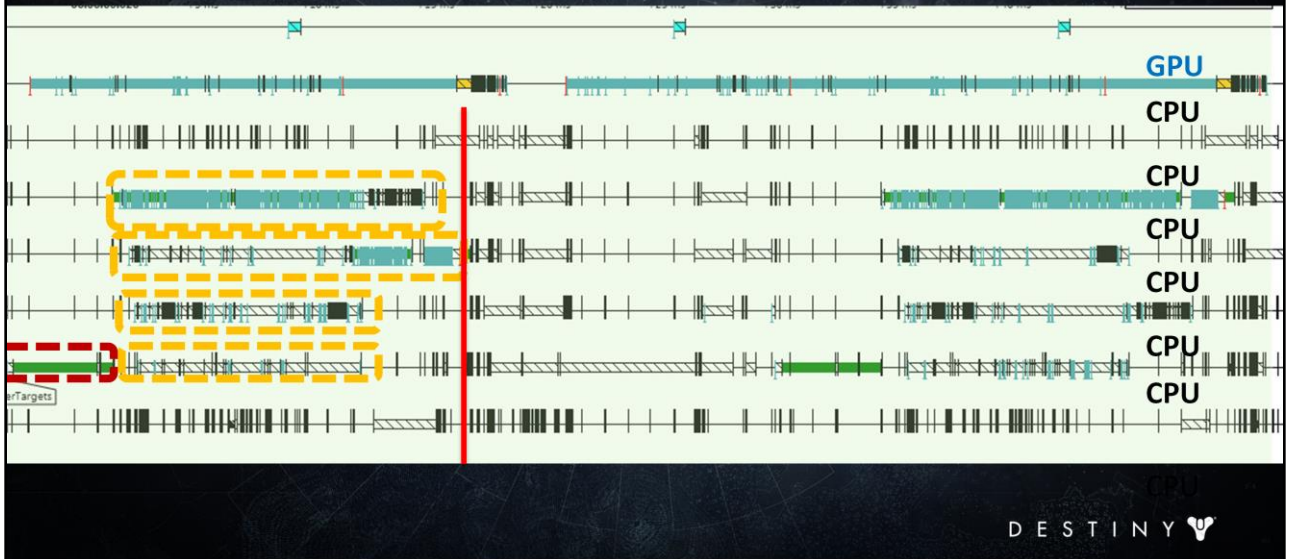
Great for complete control

Other consoles / PC do not and require manual work

Spin up a separate thread, wait for Vblank event

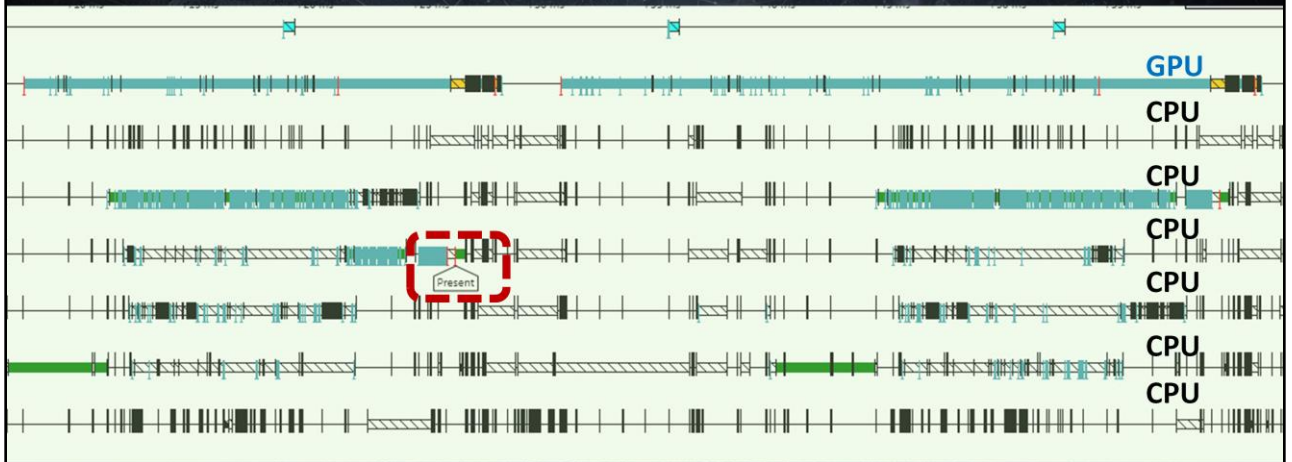
Beware of Present deadlock when resize / fullscreen event posted

Optimal GPU Occupancy



We want to multi-thread our submission to keep the overall game latency low and keep GPU idle time minimal. The quicker we can get commands to the GPU, the more it's occupied. In MT submit case, we start submitting the frame <here>. <Then>we run a bunch of threaded commands to generate deferred command buffers. Multi-threaded submit means we can flush commands to the GPU tighter, keeping it constantly occupied <good for latency>

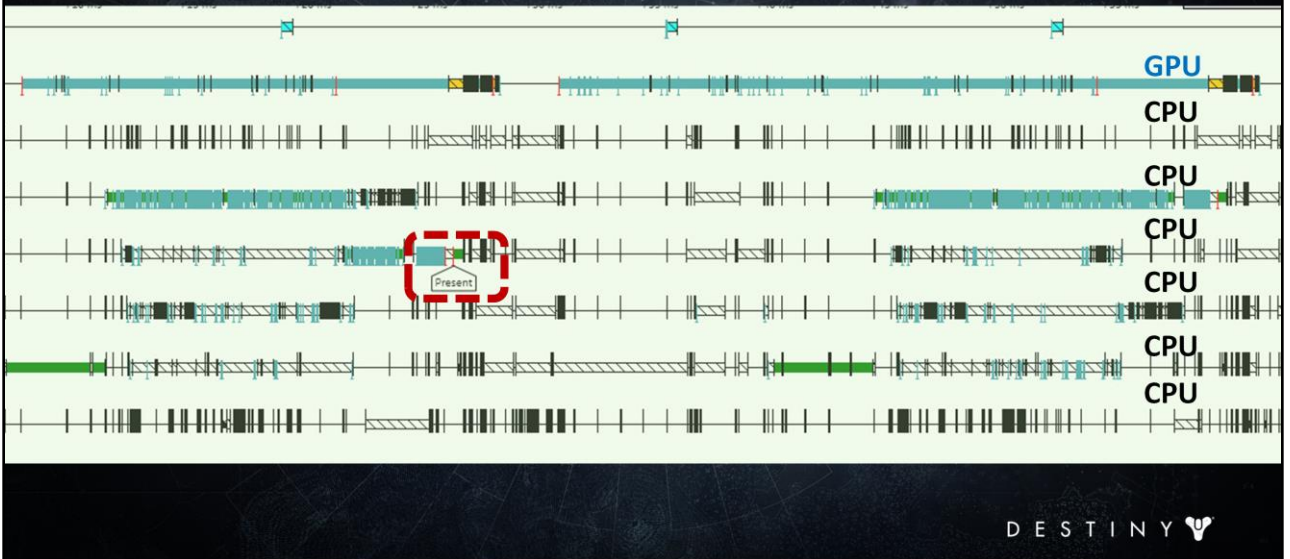
Optimal GPU Occupancy



DESTINY 

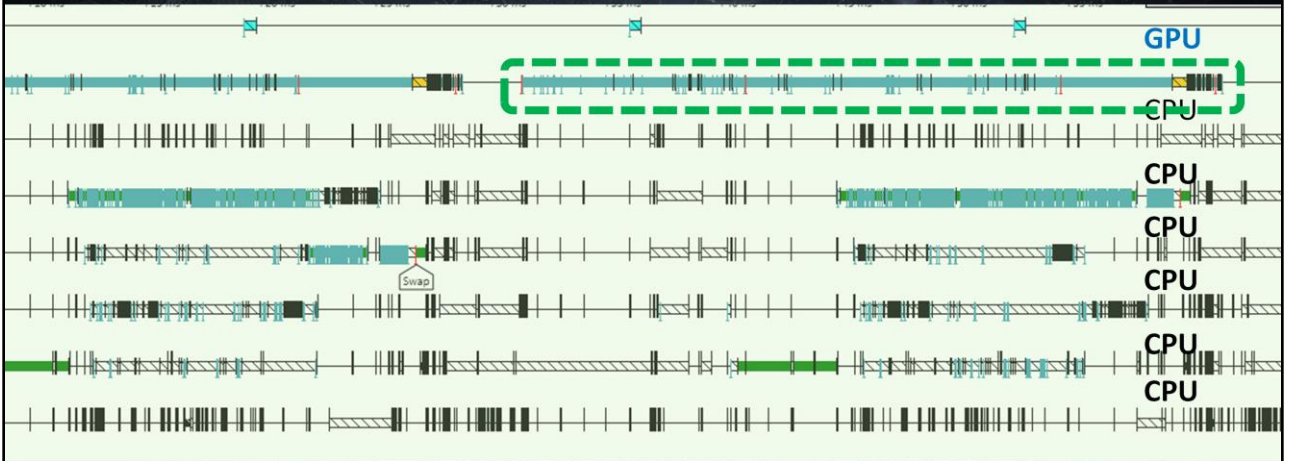
You'll notice that we <present> a lot faster

Optimal GPU Occupancy



You'll notice that we <present> a lot faster

Optimal GPU Occupancy



DESTINY 

And we see that the GPU execution is one solid wall of work – this is what we want to see.

Multi-Threaded Submit

- Complex topic, won't be able to cover all aspects
- High-level submit script generates command buffers for each per-view, per-stage submit
- GPU must execute commands in order
 - Important when we *flush* command buffers to GPU
 - Not when we generate them

DESTINY 

Next, let's take a deeper look at our threaded GPU command generation on the high-level (that complex topic can fill another two hours easily).

We convert the high-level submit directives to submit jobs executing feature renderers submit kernels.

But GPU must execute commands in a very specific order. Ex: Can't run transparent before G-buffer or else...

So we need to establish and maintain this order via explicit command buffer *flush* dependency. It doesn't matter when the command buffers are generated but only when we flush them to the GPU.

So we must allocate GPU command buffers, execute submit jobs and flush the command buffers using this synchronization.

High Level Render Submission

- Issue high-level, high-cost render state
 - Target bind / clear / resolve
- Issue high-level submission directives
 - “Render <view> for <render stage> {optional: further feature filters}”

DESTINY 

Remember our high-level render submission which issued directives to submit render per view per stage directives?

High Level Render Submission

- Issue high-level, high-cost render state
 - Target bind / clear / resolve
- Issue high-level submission directives
 - “Render <view> for <render stage> {optional: further feature filters}”
 - Jobify under this layer
 - Generate command buffers for this directive

DESTINY 

Each such request maps to a { asynchronous command buffer | deferred command list }, generated as the result of a number of submit jobs and inserted into the sequence of submission command

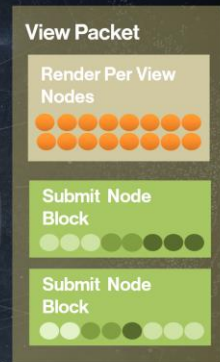
These command buffers are either directly inserted into the main GPU command ring buffer (if platform allowed) or use custom indirection to insert later (for platforms with more limiting API)

Each request generates submit jobs to generate this asynchronous command list in concurrent jobs

*Lots of trickery here, I'm avoiding that all together... (allocation callbacks versus allocation upfront versus “jump-to-self” versus “flush from watchdog thread” – each platform plays by its own rules)

Generating Submit Jobs

- Jobify N submit nodes together
 - Group by feature and render stage
- Splice command buffers in order of submit job *creation*
 - From high-level submit script job
 - For GPU order execution
- Each per-feature | per-stage | per-view submit job == 1 command buffer



DESTINY 

Thus we can jobify submit execution by blocks of submit nodes together.

We can start by grouping by feature and render stage – the feature kernels are already in place.

It's important to know that these jobs will be flushed in order of creation so that we have the right GPU execution order.

Each submit node block can map to a single submit job / command buffer { view / render stage }

Submit Jobs & GPU Occupancy

- Need optimal balance of CPU and GPU loads
 - Light CPU to flush to GPU quickly and get it going
 - Medium CPU with heavy GPU workloads next
 - Etc.
- Sort submit jobs by render feature and render stage cost
 - Tuned per platform
 - Core renderer provides this mechanism

DESTINY 

Need optimal cadence of CPU and GPU loads

Light CPU to flush to GPU quickly and get it going

Medium CPU with heavy GPU workloads next

Etc.

Sort submit jobs by render feature and render stage cost

ex: submit jobs for depth prepass stage for environment and terrain features which have very light CPU cost, but will crunch for a bit on GPU while we have the CPU work on submitting Gbuffer character jobs next, which will occupy the GPU for a while, and then continue submitting your workloads.

Since GPU strengths varies per platform, this sort heuristic needs to be balanced per platform.



Dynamic Load-Balancing

That in itself was pretty good, but we still had to attack one more axis. That is –in a data-driven pipeline, we can easily be doomed to high job overhead cost. So how did we attack dynamic load-balancing in our architecture?

Load Balancing Render Jobs

- Naïve approach can over-jobify
 - Initial implementation: one job per feature per phase (extract / prepare / submit)
 - Workloads varied drastically
 - Ex: Environment / terrain – nothing to do in extract | prepare
 - Characters { Gear / Skinned / Cloth } – heavy work
 - Too much job overhead cost

DESTINY 

A naïve approach to this type of design can result in a huge explosion of jobs. In fact, when we started, we had one job per feature renderer per entry point per phase. That resulted in pretty heavy job overhead. Additionally, workloads varied drastically from job to job – some features had heavy CPU workloads in extract, for example, but none in prepare, while others may have had almost no workload in either phases. This was not a shippable approach.

Dynamic Load Balancing

- Each feature provides per phase cost function
 - Requires stand-alone job?
- Core system batches based on entities and feature cost per phase { extract | prepare | submit}
 - Automatically load balances based on data in the frame

DESTINY 

So we extended the architecture to allow dynamic load balancing by the number of visible objects *and* feature cost per each object type. Each feature renderer can specify cost per phase. On the coarsest level you can think of this cost as “should I run in a stand-alone job?”. This allowed the core architecture to batch execution of multiple render objects within a phase based on this cost. If you’re a feature with light extract, you will be batched with other light extract features. But if you’re a heavy extract user, then you get your own job. This automatically load balanced based on visible objects data in any frame.

Heterogeneous Job Execution

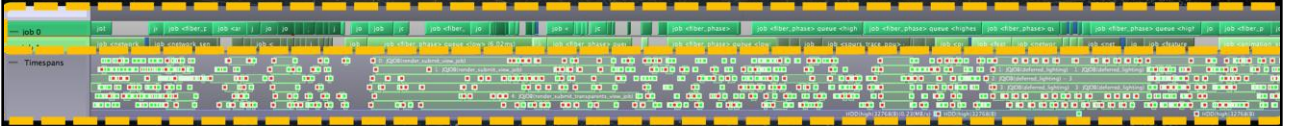
- Each feature specifies what execution units it supports
- Core renderer automatically schedules based on availability
 - Ex: SPU / PPU scheduling
- Mostly used for extract
 - Other phases had heavy workloads on PPU
- Can extend to other systems in the future

DESTINY 

Next, for heterogeneous systems, we allowed each feature to specify which execution unit they are able to run on – for example, on PPU or SPU. Then the core renderer architecture will automatically schedule based on phase and available to run on either. We mostly used it for extract phase because for both prepare and submit, simulation occupied the vast majority of PPU and we had none to spare. But during extract, which also had some of the heaviest SPU occupancy, this allowed us to balance more efficiently based on load.

We could consider extending this to future systems, for example, compute on the GPU (if we ever had GPU to spare).

Load Balancing Example

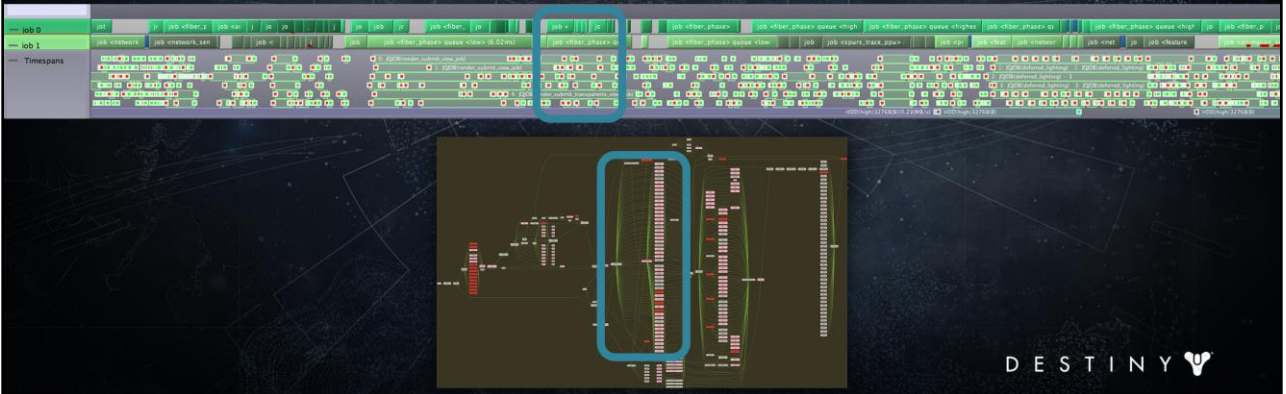


DESTINY 

Combing back to the example from PS3 execution we looked at earlier
<Note> the distribution of jobs on PPU (these are the two PPU threads) – we had a few jobs running on PPU for rendering but mainly during extract and the high-level submit script jobs.
<And> the rest had pretty solid wall of SPU occupancy.

In Practice...

Extract Game State



If we look at the extract phase, you'll see that we spread the work pretty heavily across all units – this is the example of dynamic heterogeneous load-balancing I just told you about.

Outline

- Coarse-grained parallelism
- Destiny renderer goals
- Decoupling simulation and rendering
- Jobification for core workloads
- Data-driven render submission
- Advanced optimizations
- Conclusions



Well, this describes a fair amount of details about our system. Of course we barely touched on the more complex aspects of our system, but you should now have a good idea of the overall design. How did we do with respect to our goals though?

Results

- Delivered low-latency, efficient, highly scalable execution across multiple platforms
 - PlayStation 4, PlayStation 3, Xbox One and Xbox 360
- Feature code was cross-platform
 - Was not affected by architecture tuning



DESTINY 

Our architecture delivered low-latency, efficient highly scalable execution across multiple platforms and multiple console generations. The Destiny renderer architecture enabled us to ship the game with steady performance on all shipping platforms.

The vast majority of our feature code and high-level submit code was cross-platform and cross-generation. The vast majority of optimizations done for the core architecture did not affect any of the leaf feature code. This was no small feat.

I'd say "ship it!" but it's too late. We already have. 😊

Required to Ship


- Multithreading the renderer was key to shipping Destiny
 - Scalability and dynamic load balancing
 - Cache-coherent data access
 - Heterogeneous support
 - Achieved low latency despite heavy workloads

DESTINY 

Multithreading the renderer was key to shipping Destiny. We simply would not be able to support some of the more challenging platforms otherwise as the kernel-based processing for our render workload jobs mapped well to SPU. Our architecture's scalability and dynamic load balancing, combined with cache-coherent data access and heterogeneous support allowed us to achieve low latency despite heavy workloads across the board.

Takeaway

- Data-driven pipelined architecture provides great flexibility for creating graphics features
- Solid encapsulation allows much optimization to happen throughout development

DESTINY 

If there's anything I want you to walk away with today it is that creating data-driven pipelined architectures can give you huge flexibility for creating graphics features and yet help you optimize the overall performance of your engine across different platforms.

Achieved Our Goals

- Successfully decoupled game-state traversal from output
- Achieved better CPU and GPU utilization for shipping game
 - Breaking up rendering algorithms into task- and data-parallel workloads via a data-driven architecture

DESTINY 

We successfully decoupled game-state traversal from output systems (in our case, rendering). We moved visibility computation out of the render thread and multi-thread visibility computation from the get-go. We achieved better CPU and GPU utilization for shipping game by breaking up rendering algorithms into task and data-parallel workloads is a data-driven process based on visible elements in the frame and the underlying data (shader and mesh properties)

Lego-Like Extensibility

- Adding new features was straightforward
 - Simple, quick steps
 - Core architecture provides data management and jobification
- Adding new render stages was easy



DESTINY 

Our jobification is accomplished in a transparent manner for almost entire graphics team. Our graphics engineers focused on developing rendering algorithms instead of the challenges of breaking every feature up into multithreaded workloads. The <lego-like> functionality of feature renderers paid off tremendously for Destiny and allowed us to deliver a huge amount of varied render features. And the architecture proved to be extensible - many new render features and stages were added throughout the development and the architecture held up well. Of course we also learned and extended the core architecture as we went.

Don't Rinse and Repeat

- Cross-platform feature code meant we only have to write graphics features once
 - Relied on cross-platform surface and shader layer
- **Write features and port to platforms easily:**
 - Port low-level layers (device, surfaces, shader layer) and one feature renderer (Ex: 5 weeks)
 - Once that's done, the rest of features are converted in < 1 week
 - Exception: advanced API (HW instancing, ...) – small effort to port
- **Provided huge overall engineering savings**

DESTINY 

Cross-platform feature code meant we only have to write graphics features once in our cross-platform API (including the cross-platform device, surfaces and our shader layer). This meant that typical graphics features (ex: terrain, environment, characters, etc.) only had to be written once and automatically worked on new development platforms as we added them (with notable exception of SPU-fication). Some of the more advanced features needed special care, for example, for HW instancing (example: tree rendering) where we had to modify some of the feature code for new platforms (Example: Playstation 4).

This meant tremendous savings to the graphics team coding efforts.

Once we had a couple of platforms (for example, Xbox 360 and Playstation 3), adding a new platform meant: port low-level layers (device, surfaces, shader layer) and one representative feature renderer – let's say, terrain. This is the bulk of the porting work and may take several weeks to do (depending on the platform). Once that's complete, it took us less than one week to port and test the rest of the graphics features (with a couple of additional touches for HW instancing and advanced surface usage).

Challenges

- Each platform had custom requirements for
 - Submit job granularity
 - Command buffer generation
- Continuous job overhead cost optimization throughout the project

DESTINY 

We encountered some challenges along the way, of course. Unfortunately, we don't have a sufficient amount of time to cover these in details, but I wanted to mention a few.

Submit job granularity and GPU command buffer generation / flushing had to be revisited for each platform with each throwing custom requirements into the mix. We have plans on how to improve that with our next engine's iteration by making this a more automatic, data-driven process.

As we added more features and threw more jobs with more complex dependencies, job overhead cost optimization was a periodic task for the core architecture optimization, both for existing platforms and for platforms we added throughout development.

W E ' R E H I R I N G



WWW.BUNGIE.NET/CAREERS
CAREERS@BUNGIE.COM

So if you liked what you just heard and were excited about the work we do at Bungie – come join us! We are hiring (and in fact for the graphics team). Many people contributed to the awesome graphics of Destiny and we have an incredible team of engineers and artists that make this fun game.

Stop by and talk to our recruiters – they are in the audience, or grab us afterwards, we'll follow up post conference.



Slides


- <http://advances.realtimerendering.com>
- Email: natalya.tatarchuk@yahoo.com
- Twitter: [@mirror2mask](https://twitter.com/mirror2mask)

DESTINY 

Note that we will post the slides on the Advances website. I listed my email and twitter handle – if you have some questions, let me know.

Thank You!



DESTINY 



BUNGiE®