GDC | March 21-25, 2022
San Francisco, CA

One Frame in ‘Halo Infinite’

Daniele Giannetti

#GDC22

Hello everyone! And thank you for coming to this talk about Halo Infinite.

About me & 343

Daniele Giannetti (He/Him)
Principal architect at 343 Industries
(Performance, threading, engine/game architecture, physics, navigation, streaming, etc…)

- 343 Industries == Halo
  - Shooter team:
    - Halo 4 - 2012
    - Halo 5: Guardians - 2015
    - Halo Infinite – 2021

My name is Daniele Giannetti, my pronouns are He and Him, and I work at 343 Industries as an architect.

Over there, I help out with probably too many things, but mainly focused on performance, engine architecture, threading, physics, navigation, streaming and a few other things…

At 343 Industries we work on all things Halo, and today I'll be talking about some of the things we did to build our biggest game ever: Halo Infinite.

Previous titles from our team are Halo 4 and Halo 5: Guardians, although I joined after the launch of Halo 5.

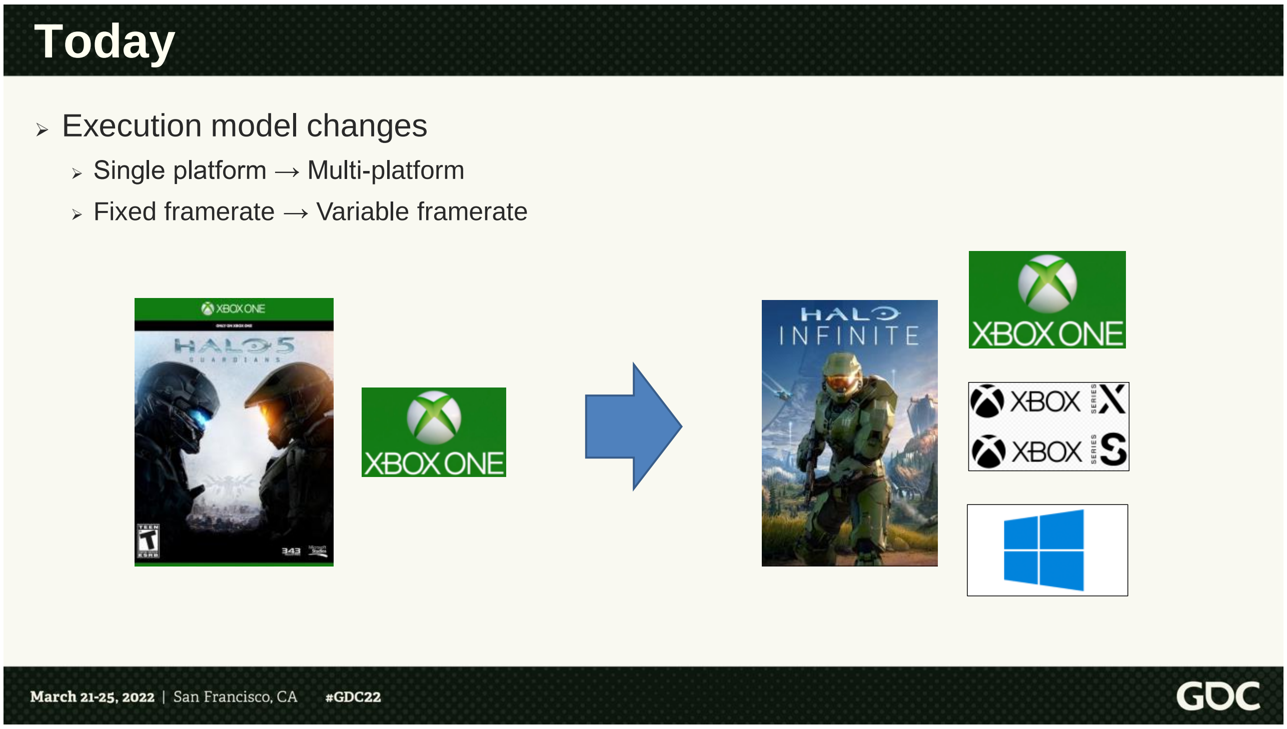
Halo Infinite

Anyway, here’s a short video of Halo infinite, just to get an idea of what the game is. Enjoy.

Halo Infinite

Halo Infinite is a fast-paced online first-person shooter with competitive and more casual multiplayer modes with vehicles, explosions and what not.

At the same time, the Halo Infinite campaign is a narrative-driven experience bringing players into a vast open environments larger than anything we’ve ever built.



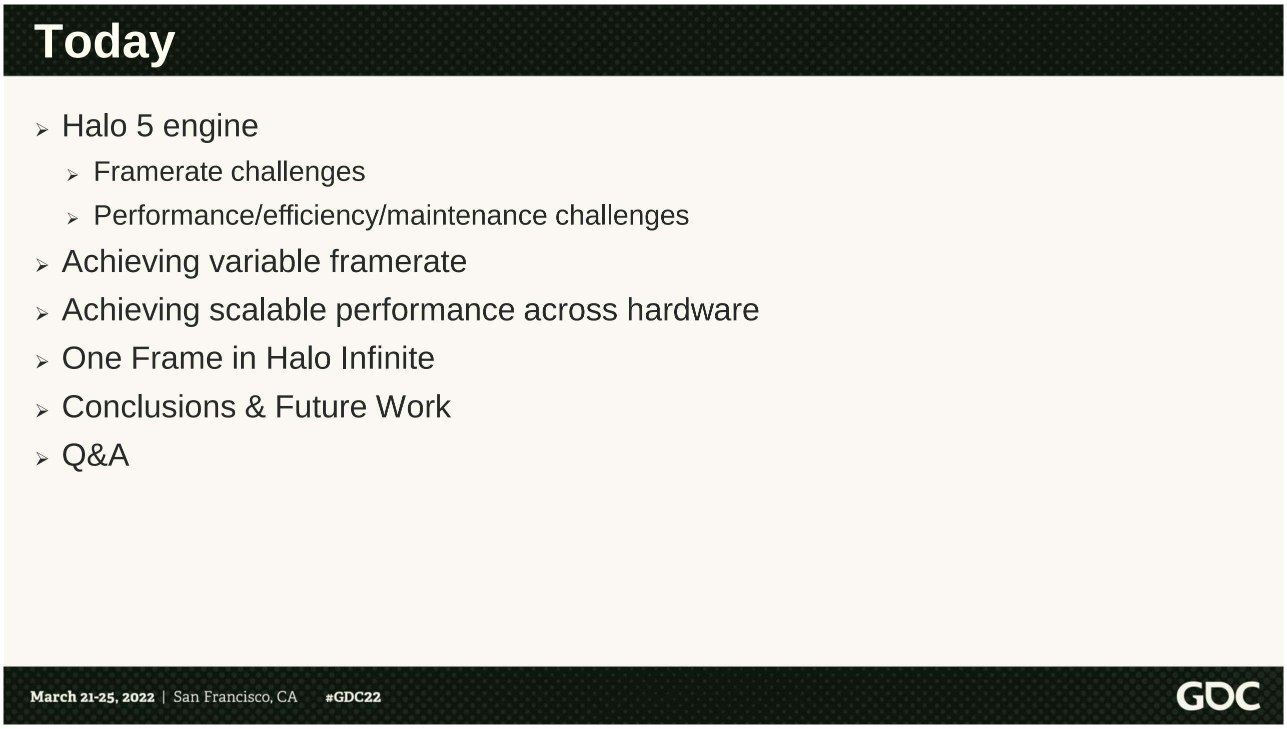
Today

- Execution model changes
  - Single platform → Multi-platform
  - Fixed framerate → Variable framerate

XBOX ONE
HALO 5 GUARDIANS
XBOX ONE
HALO INFINITE
XBOX ONE
XBOX SERIES X
XBOX SERIES S

Today, we are going to be talking about some of the game execution model transformations the Halo engine went through to evolve from a single platform fixed framerate engine, hardwired to Xbox One at 60 fps in Halo 5, to multi-platform and variable framerate for Halo Infinite.

Halo Infinite was still shipping on Xbox One, but we were also targeting the new Xbox Series console generation as well as Windows PCs for the first time. With the massive variability in hardware capabilities on the right, we wanted to allow for gameplay to scale up to 120 fps on Xbox Series X, and more on powerful PCs, and allow PC players to configure their desired framerate for the best possible experience.



Today

- Halo 5 engine
  - Framerate challenges
  - Performance/efficiency/maintenance challenges
- Achieving variable framerate
- Achieving scalable performance across hardware
- One Frame in Halo Infinite
- Conclusions & Future Work
- Q&A

March 21-25, 2022 | San Francisco, CA #GDC22

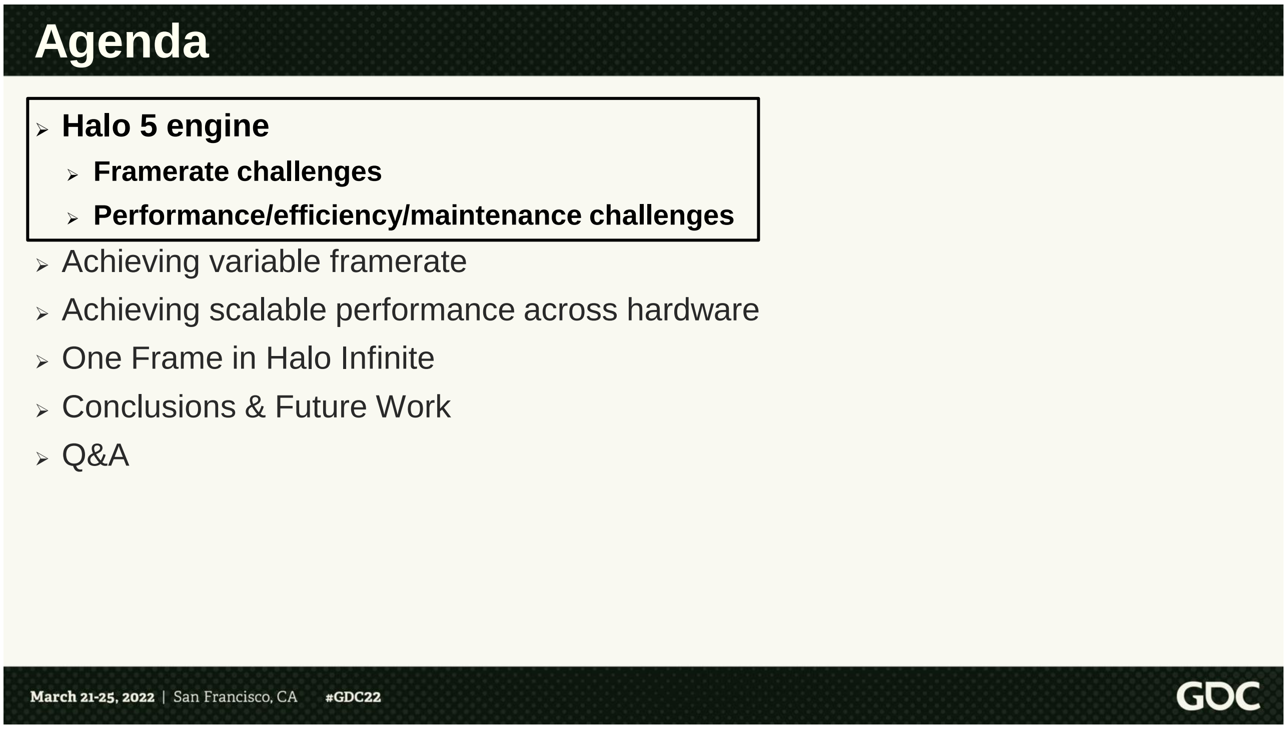
GDC

We are going to start with a quick tour of the past. I'll talk through the execution model of our legacy Halo engine, and focus specifically on two challenges we set out to solve for our new game:

- **How to achieve variable framerate**, so we could support very high FPS on new gen consoles and powerful PCs, without compromising the experience on low end consoles (and also because PC players want accurate control of their framerate).
- How to make our execution model **scale well across heterogenous** hardware without weaving a massive ball of spaghetti code.

This session will be focused on **CPU execution** mainly around simulation workload, it's not a graphics talk!

Towards the end of the talk, I'll break down one frame in Halo Infinite across different platforms to show how everything fits together, and then we'll finish up and move to Q&A.



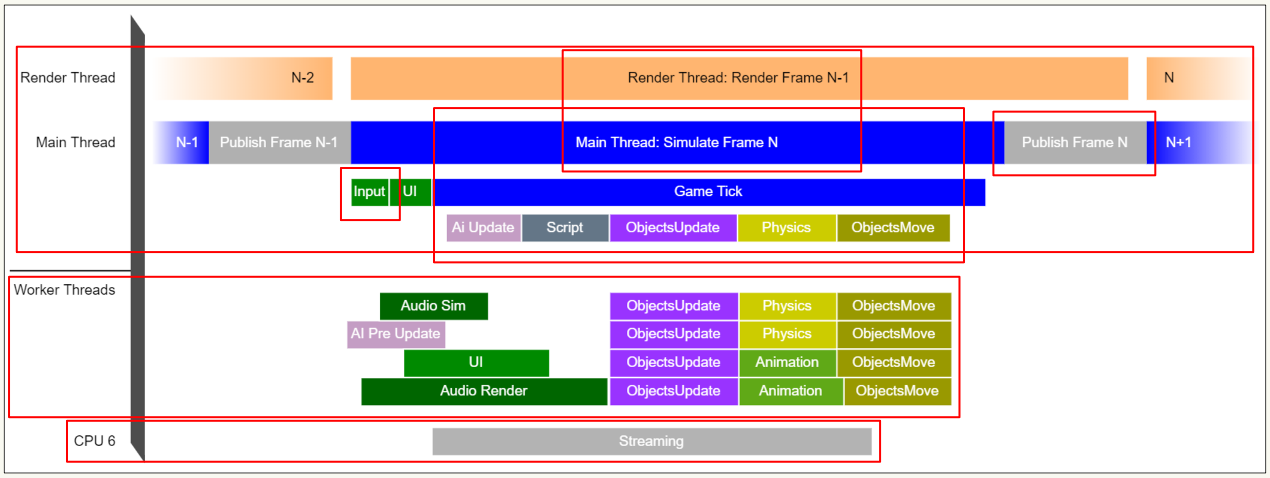
Agenda

- **Halo 5 engine**
  - **Framerate challenges**
  - **Performance/efficiency/maintenance challenges**
- Achieving variable framerate
- Achieving scalable performance across hardware
- One Frame in Halo Infinite
- Conclusions & Future Work
- Q&A

Ok, let’s jump into our legacy!

Halo 5 engine

Legacy engine frame overview: Xbox One



Coming off Halo 5, the engine was tightly optimized for Xbox One.

Xbox One has 8 physical cores and the title has access to 7, CPU-0 through CPU-6, CPU 6 is partially shared with the system and thus ideal for asynchronous latency tolerant workloads (only roughly 50% of its bandwidth is guaranteed to the title).

We had a classic 2-thread architecture with Simulation and Rendering arbitrated by their own thread (which we call main thread and render thread) running on CPU 0 and 1. 4 worker threads spinning on CPU 2 through 5 mostly collaborating on simulation workloads.

Latency tolerant work (like most streaming) relegated to CPU 6.

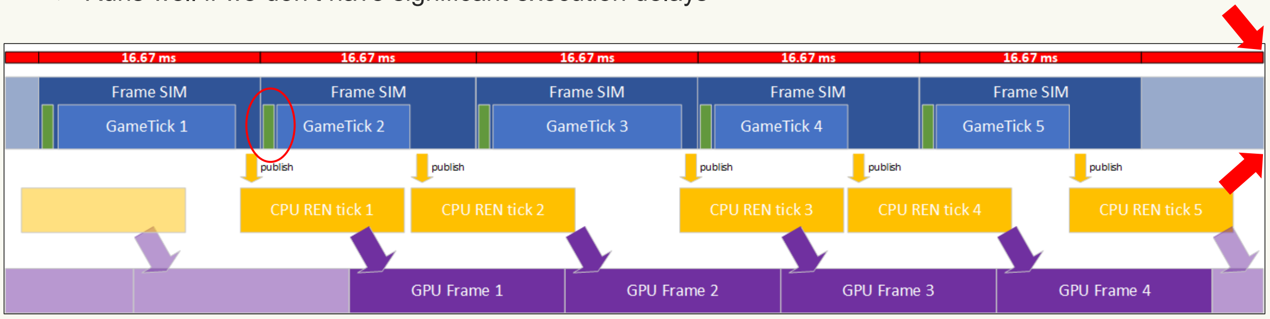
From a CPU standpoint, as we simulate frame N, we render frame N-1 in a classic pipelined model.

We sample input at the beginning of the simulation frame, we do all our object updates, animations, physics and transform updates as part of what we call “game tick”, which is a portion of the simulation frame.

To publish data to the renderer we do a massive copy of game state so that the renderer can consume it while the simulation works on the next frame.

Halo 5 engine: game tick

- Each game tick consumes exactly 16.67 ms of time
  - Hard-wired to run at 60 fps
  - Runs well if we don't have significant execution delays

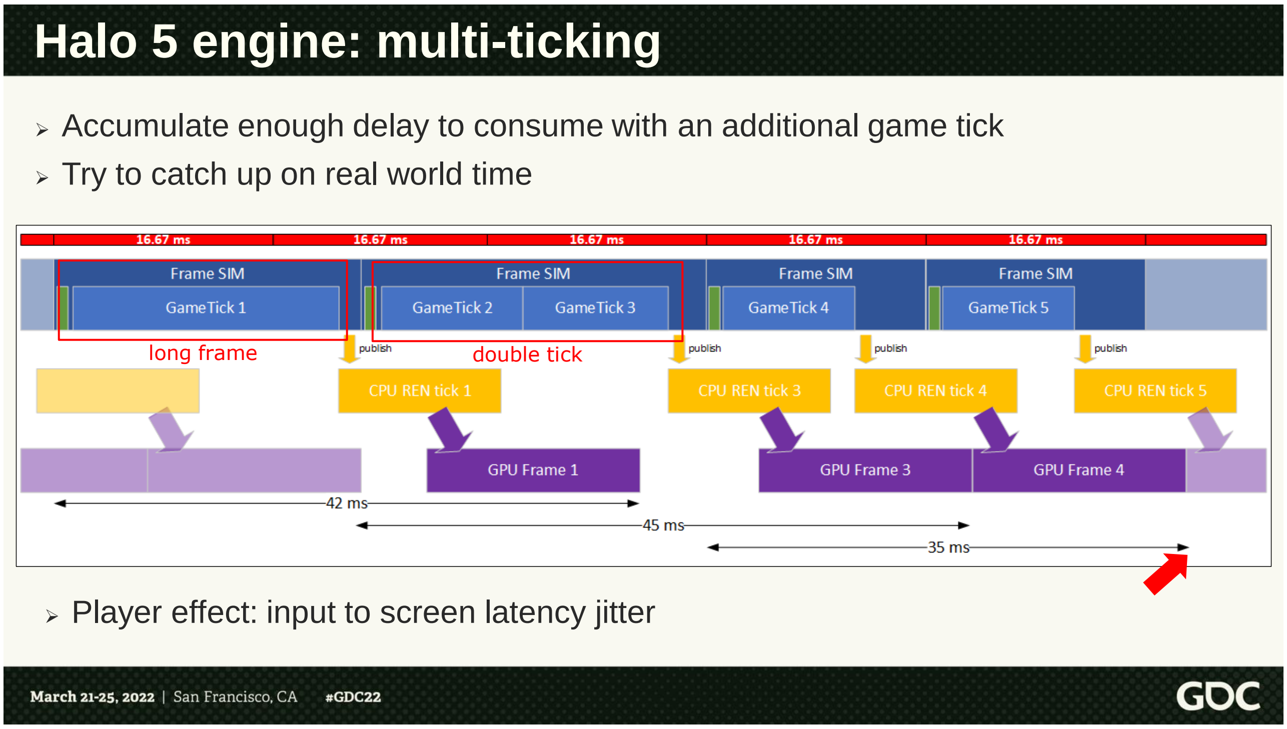


- What happens if we have execution delays?

The legacy Halo engine was hard-wired to run at 60 fps, each game tick would consume (and simulate) exactly 16.67 ms of time. This worked reasonably well if we didn't have any execution delays since Halo 5 was strictly a 60 fps game.

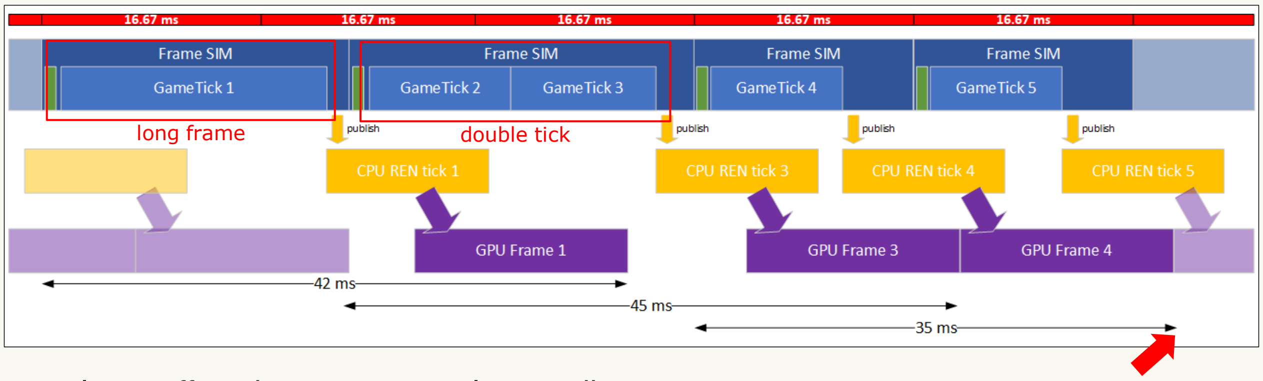
In this diagram I drew the wall clock (real world) time in red, showing 16.67 ms intervals, and I'm showing game tick execution on the hardware in blue. If the game tick executes faster than real time, we can consume the real world time fast enough to never miss a frame. I'm also showing in green the input sampling happening right before we execute the game tick on the CPU. After we are done executing the game tick, we publish the simulation results to the renderer (drawn in yellow for each frame) and as the renderer submits draw commands, the GPU picks up the work to display the frame.

What happened in Halo 5 if the game simulation had some delay? Maybe because of an execution spike?



Halo 5 engine: multi-ticking

- Accumulate enough delay to consume with an additional game tick
- Try to catch up on real world time



- Player effect: input to screen latency jitter

March 21-25, 2022 | San Francisco, CA #GDC22 GDC

We had a mechanism for catching up on lost time that we called multi-ticking.

Basically, if simulation was running long due to a hitch or some abnormally long workload, eventually we would accumulate enough real-world time delay to make a whole 16.67 ms of delay, and we would consume that time by doing an additional game tick within the simulation frame (that is 2 game ticks instead of 1).

Doing 2 game ticks in one frame means effectively doing all the heavyweight workload twice, but we would only pay for input sampling and publishing the simulation once, of course. There is no guarantee that doing multiple game-ticks in one frame would lead to catching up on time. Specifically, if we do 2 ticks, but the SIM frame takes us longer than 33.33 ms, then this approach would accumulate even more delay. So this approach of “catching up” on real world time assumes a performant game with seldom “spikey” workloads.

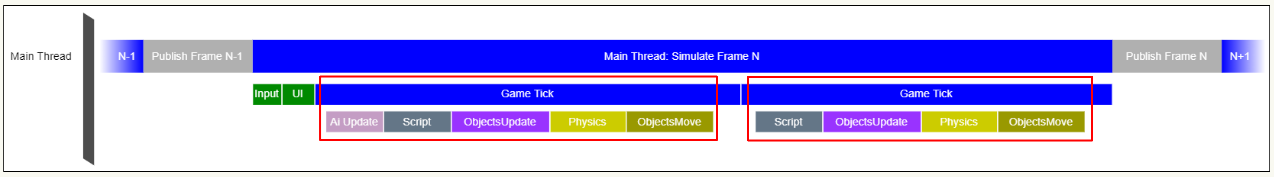
We had a maximum amount of delay that we allowed to accumulate. Specifically, we allowed a maximum of 4 game ticks per frame. In offline gameplay (for instance single-player campaign), if we accumulated more delay than that we would start slowing down the game timeline, in online gameplay instead we'd eventually get corrected by the server timeline.

If we were able to catch up quickly, then the player wouldn't easily feel any overall slow down or speed up of the simulation (real world time).

In the picture here I also drew black lines at the bottom representing the total SIM + REN + GPU duration for the frames, which is indicative of input latency since input is in green at the beginning of the simulation frame, and we're presenting at the end of the GPU frame. As you can see, the longer-than-expected tick in the first frame, as well as the two ticks happening in the second simulation frame, cause longer than expected end to end input latency compared to "normal" frames (we see 42 and 45 ms compared to 35 ms on the last frame depicted above).

The player impact of multi-ticking can then be summed up as a temporary slight variation in input to screen latency (i.e. jitter), but no simulation "slowdown".

Halo 5 engine: multi-ticking

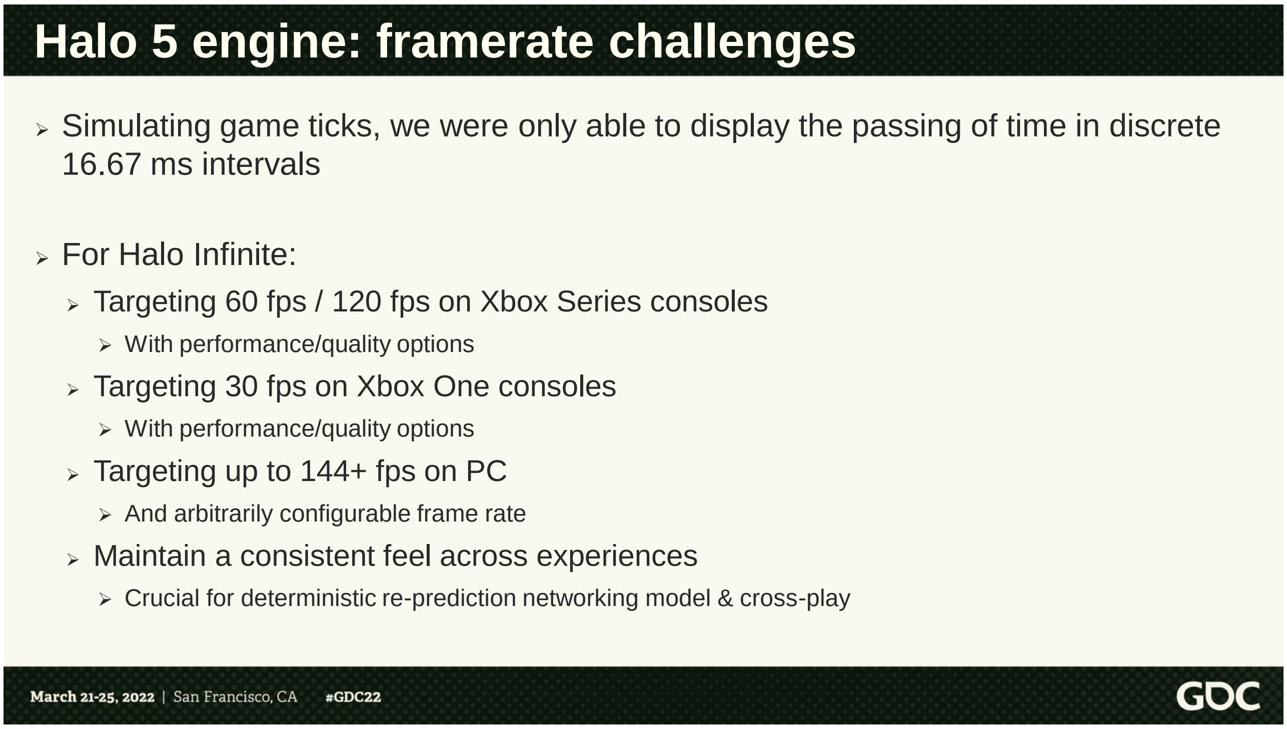


- Repeat core simulation workloads
- Input/publish only once

Here's a more detailed visualization of a simulation frame multi-ticking with 2 ticks.

As you can see, we are doing twice the work for the core of the simulation (objects update, physics update, and objects move), but only sampling inputs at the beginning of the frame and of course only publishing to the renderer once.

Some systems were able to react and help the simulation catch up during multi-ticking. For instance: in case of locally simulated AI, we would avoid most of the workload in the second tick of the frame (in pink above). This was the exception, though, and not the rule.


Halo 5 engine: framerate challenges

- Simulating game ticks, we were only able to display the passing of time in discrete 16.67 ms intervals
- For Halo Infinite:
  - Targeting 60 fps / 120 fps on Xbox Series consoles
    - With performance/quality options
  - Targeting 30 fps on Xbox One consoles
    - With performance/quality options
  - Targeting up to 144+ fps on PC
    - And arbitrarily configurable frame rate
  - Maintain a consistent feel across experiences
    - Crucial for deterministic re-prediction networking model & cross-play

March 21-25, 2022 | San Francisco, CA #GDC22

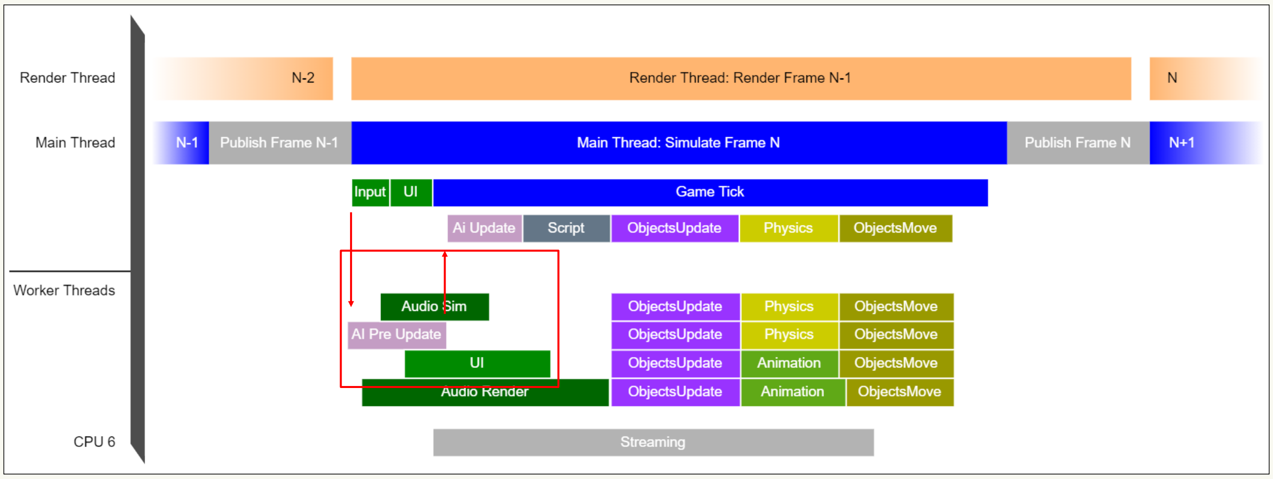
Even with multi-ticking, the fact was that the Halo 5 simulation advanced in discrete chunks of 16.67 ms of time. We were only able to display movement of objects at that rate, that is hard-wired to 60 fps.

But for Infinite we had different framerate targets per hardware, and on PC specifically we wanted to even allow unlocked framerate where the game would present “as fast as possible” always.

All hardware targets had to maintain a very consistent feel for the physics-driven Halo sandbox. Moreover, our core physical simulation needed to be as deterministic as possible across targets to enable cross-platform deterministic re-prediction of the network simulation (which is key to our gameplay networking model for latency compensation), more on this later.

Before we jump in and describe what we did to achieve these goals, though, let’s talk about the second big challenge of our legacy Halo 5 tech

Halo 5 engine: Tetris scheduling



➢ Doesn't scale well to diverse hardware, but there's more…

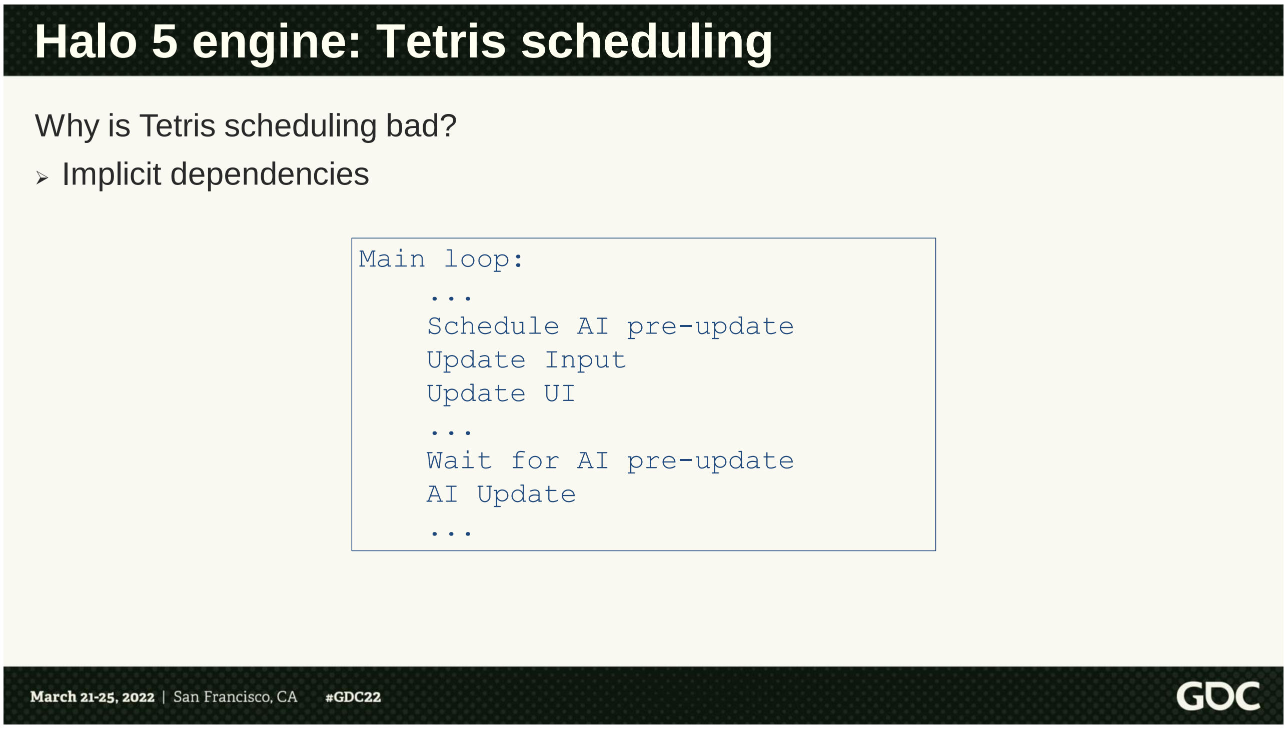
Going back to the structure of a frame in our legacy engine.

Let's look specifically at the systems in the red box, taking AI pre-update as an example.

AI Pre Update workload gets scheduled by the main thread at the beginning of the frame on a specific CPU, and then the main thread waits for that workload to be finished before continuing with the AI update workload. It was scheduled this way because we knew that on Xbox One, when running clients with AI simulation (e.g. offline campaign) we would have idle time there. All CPU workloads were then carefully assigned a place to run within our Xbox One frame on CPU, this is what we call 'Tetris scheduling'.

UI and Audio Sim were scheduled similarly in the picture above.

This type of scheduling is trivially suboptimal across different CPUs because we may not have enough logical cores to run all these simulation threads at the same time on some PCs, and if the PC we are running on has more than 6 or 7 cores, then our utilization would have been pretty bad. But there's actually an even more tricky problem that comes with Tetris scheduling of our work, and that is a maintenance problem, let’s talk through it



Halo 5 engine: Tetris scheduling

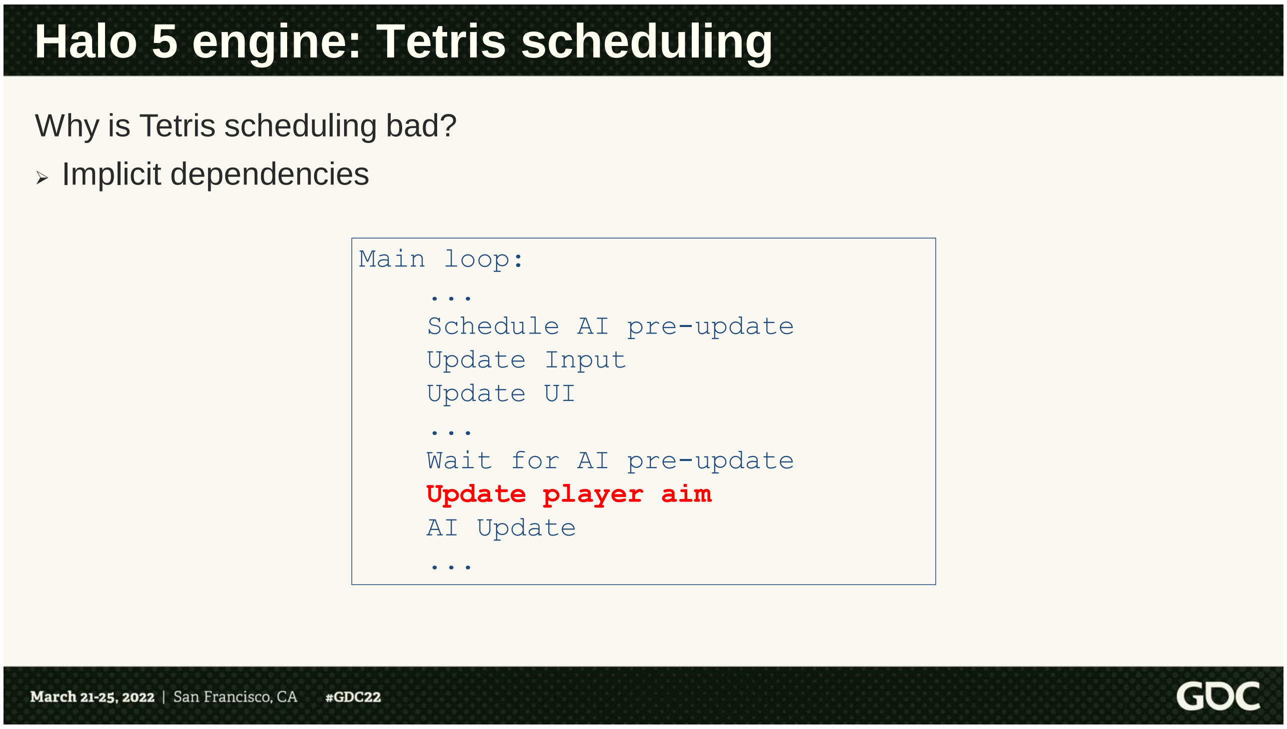
Why is Tetris scheduling bad?

➢ Implicit dependencies

```
Main loop:
    ...
    Schedule AI pre-update
    Update Input
    Update UI
    ...
    Wait for AI pre-update
    AI Update
    ...
```

To understand why Tetris scheduling was particularly challenging for us, let’s look at an example following the frame topology that we just discussed.

Early in the main loop function running on the main thread, the game schedules the AI pre-update job. On the main thread then a few things happen like updating the input and a portion of the UI workload. After a while we decide to wait for the AI pre-update before running the AI Update as discussed above.



Halo 5 engine: Tetris scheduling

Why is Tetris scheduling bad?

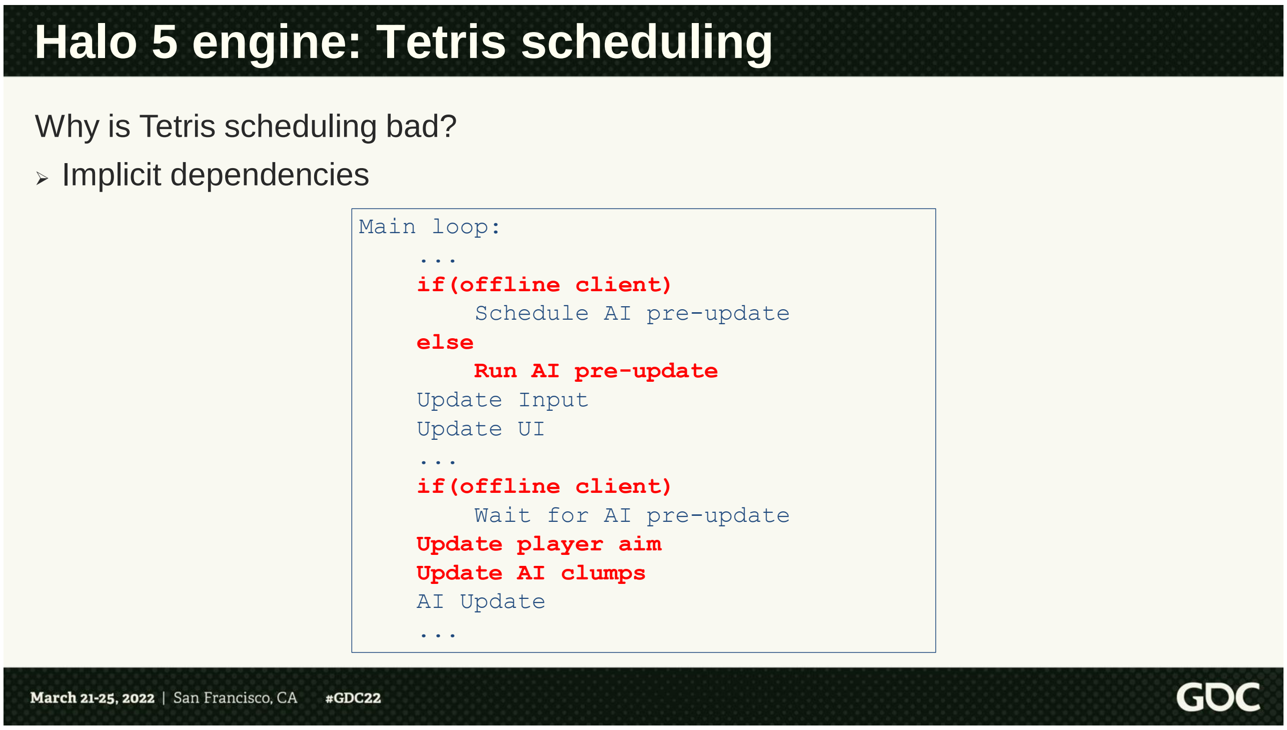
- Implicit dependencies

```
Main loop:
    ...
    Schedule AI pre-update
    Update Input
    Update UI
    ...
    Wait for AI pre-update
    Update player aim
    AI Update
    ...
```

Let’s say now that we are adding a new system to adjust some parameters of the player aim vector, the engineer implementing the system finds a spot in the main loop to update his system, and they leave it there as above. Months or years pass, the engineer moves on to something else, and some time later someone is looking at this code.

Does this aim update system need to update before AI update? Does it need to update after the threaded AI pre-update is completed? That’s what is currently happening, but if we are trying to improve parallelism and efficiency, it is impossible to tell without a deep inspection of all systems involved what the actual execution dependencies are. This is what we call implicit execution dependencies. The execution dependencies between systems are implicit to the order in which they run and not explicitly documented in the code. Also we call this model explicit scheduling, because on the main thread we are explicitly commanding the start of parallel execution of individual workloads like the AI pre-update.

This is a grossly oversimplified overview of the problem, but to add some spice we can see how this can get complicated quickly.



Halo 5 engine: Tetris scheduling

Why is Tetris scheduling bad?

- Implicit dependencies

```
Main loop:
    ...
    if(offline client)
        Schedule AI pre-update
    else
        Run AI pre-update
    Update Input
    Update UI
    ...
    if(offline client)
        Wait for AI pre-update
    Update player aim
    Update AI clumps
    AI Update
    ...
```

Some time later, another engineer added a system to update AI clumps and stuffed it into the main loop as above.

Presumably, AI clumps update must occur before AI update, but did the engineer mean to wait on the update for the player aim, or is that just how it fell into the main loop body function and the engineer didn’t have time to carefully hand schedule this together with the rest of the AI system in a parallel fashion? Also, does the AI clumps system need to wait for the end of the AI pre-update or not?

Moreover, in Halo 5 we would have branching code like above, that changed the topology of execution depending on the runtime state of the game, changing the implicit dependencies as well, were those implicit dependencies unnecessary? Or true dependencies?

Halo 5 engine: Tetris scheduling

Why is Tetris scheduling bad?

- Challenges scaling to diverse hardware
- Hard to maintain code as new systems are added or re-written
- Very hard to onboard new engineers

These maintenance issues made it difficult to scale execution to more powerful hardware like PC CPUs where often we have more available logical cores, and especially execution was particularly suboptimal on more powerful PC CPUs but with less available parallelism (less logical cores but faster).

Maintenance issues also make it really hard to keep execution efficient as new systems were being added for Halo Infinite and those new systems needed to update within the frame.

Finally, new or junior engineers without intimate knowledge of the implicit dependencies in the system would always err on the side of caution, adding to the tech debt, and also would be completely thrown off by the challenges of having to schedule any parallel work themselves due to the amount of boilerplate, so ended up writing slow/serial code “by default”.

This led to a mostly untenable ball of spaghetti code that was hard to maintain and evolve for cross-platform efficiency. I wanted to find a really ugly picture of spaghetti for this slide, but then my Italian nature took over and ended up with a much more inviting plate here… but don’t be fooled, this was actually a tough problem for us.

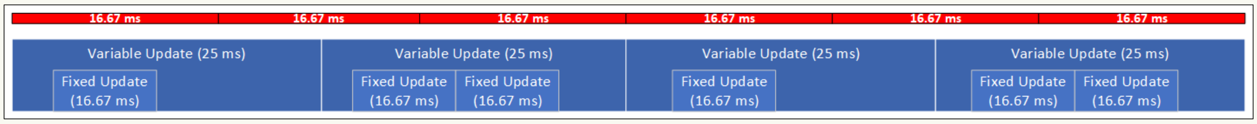
Agenda

- Halo 5 engine
  - Framerate challenges
  - Performance/efficiency/maintenance challenges
- **Achieving variable framerate**
- Achieving scalable performance across hardware
- One Frame in Halo Infinite
- Conclusions & Future Work
- Q&A

Ok, we are ready to move on to Halo Infinite, let's start talking about the modifications we did to achieve our goals of variable framerate while maintaining the same feel for the game sandbox across targets.

Achieving variable framerate

- Split time consumed by the game in two:
  - **Fixed update** time
    - Consumed in 16.67 ms interval inside the game tick
    - Fixed update systems include the physics sim and the deterministic network simulation systems
    - Updated inside the fixed update (or game tick) within the simulation frame
  - **Variable update** time
    - Consumed in variable intervals depending on the duration of the frame
    - Variable update systems include all eye-candy systems or not affecting deterministic simulation
    - Updated outside of the game tick within the simulation frame



While before we were only able to consume time in 16.67 ms intervals, for Infinite we logically subdivided time consumed by the engine in two types.

Fixed update time is consumed always in 16.67 ms chunks, just like in Halo 5. Systems that require a consistent rate to achieve a consistent feel will be stepped in fixed updates like this. This includes our physics systems, for instance, in order to achieve a consistent vehicle/movement “feel”, this also includes other systems that are part of our deterministic network simulation. In this talk I won’t go into details of our gameplay networking model, but we rely on near-determinism for our simulation, which requires all systems contributing to results of deterministic simulation (like gameplay object updates and physics) to produce the same result across hardware targets. This requires stepping forward at a consistent rate, which is our fixed update.

Within a single game frame of simulation, we may perform 0 or more fixed updates.
We still use the legacy terminology of “game tick” to refer to a fixed update in the new engine, so in the rest of this presentation I’ll be using them interchangeably.

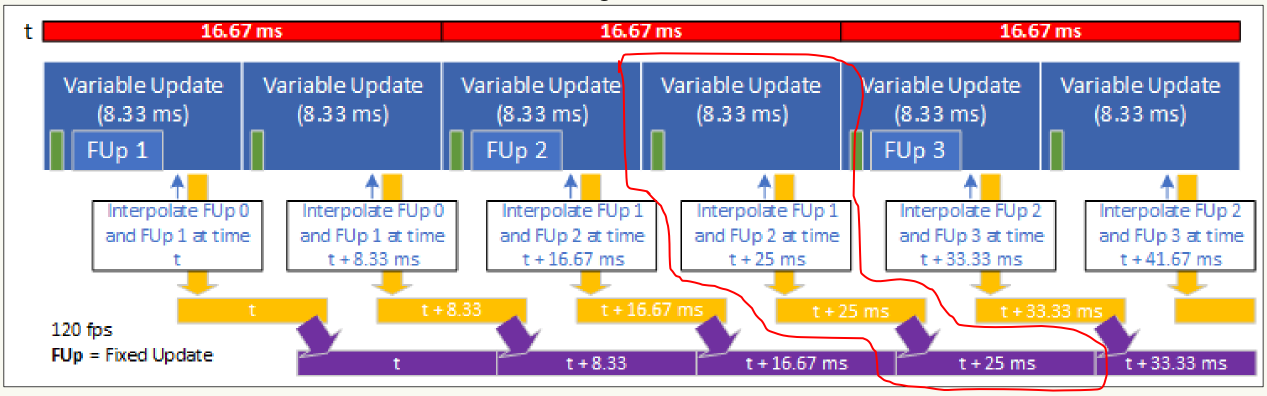
Variable update time is effectively wall clock time elapsed since the previous frame. This is consumed in variable intervals. System that aren’t sensitive to behavioral changes due to variable framerate, or systems that aren’t needed to achieve deterministic networked simulation are then stepped at variable frame rate outside of the “fixed updates” or “game ticks”.

See an example on the slide of how the game simulation might run at 40 Hz in this new model.

Assuming we are running at exactly 40 Hz, each frame is exactly 25 ms, and variable time is consumed at that rate of 25 ms per frame. Fixed update time is only consumed by game ticks at a rate of 16.67 ms for each game tick, and thus we’ll have 3 game ticks for every 2 frames, leading to an execution pattern like in the slide, with simulation frames alternating 1 game tick and 2 game ticks to consume the right amount of time.

Halo Infinite "tweening"

- Fixed update results are interpolated to match the variable update timeline (aka "tweening")
  - Objects/player transforms and poses
  - Gives the illusion of smooth movement regardless of framerate



t
16.67 ms
16.67 ms
16.67 ms
Variable Update (8.33 ms)
FUp 1
Variable Update (8.33 ms)
Variable Update (8.33 ms)
FUp 2
Variable Update (8.33 ms)
Variable Update (8.33 ms)
FUp 3
Variable Update (8.33 ms)
Interpolate FUp 0 and FUp 1 at time t
Interpolate FUp 0 and FUp 1 at time t + 8.33 ms
Interpolate FUp 1 and FUp 2 at time t + 16.67 ms
Interpolate FUp 1 and FUp 2 at time t + 25 ms
Interpolate FUp 2 and FUp 3 at time t + 33.33 ms
Interpolate FUp 2 and FUp 3 at time t + 41.67 ms
t
t + 8.33
t + 16.67 ms
t + 25 ms
t + 33.33 ms
120 fps
FUp = Fixed Update
t
t + 8.33
t + 16.67 ms
t + 25 ms
t + 33.33 ms

For cases where our target framerate isn't a divisor of 60 fps, the variable time consumed every frame will never consistently match the amount of time consumed by our fixed update systems. However, we want to show on screen the results of fixed update systems (like the physics simulation), so we must bring fixed update systems onto the variable update timeline before rendering the results (or else they will appear "stuttery" as we won't display the result at the right rate). We do this through a system responsible for interpolation of fixed update system results, and we call this interpolation "tweening".

Let's look at an example of how this works when the game is running at 120 fps, for instance on Xbox Series X in performance mode.

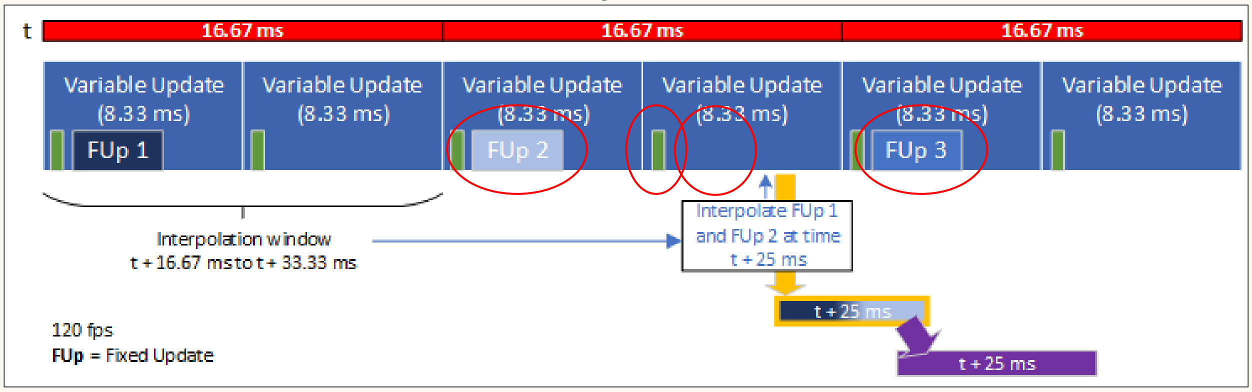
If the game is running at a steady 120 fps (no performance issues), each frame will consume exactly 8.33 ms of variable update time, and since each fixed update consumes exactly 16.67 ms of fixed update time, we will only have one fixed update every 2 frames. The result for each 8.33 ms frame will be shown on screen as normal. For fixed update systems, we interpolate the results of the 2 most recent game ticks to show a smooth result on screen.

Looking at a random frame, let’s focus on the highlighted one here in red.

This simulation frame doesn’t contain any fixed update. Input sampling proceeds as normal. Variable framerate systems step forward for the variable delta time (8.33 ms in this case). Fixed update systems like the deterministic physics sim are not updated at all, however we still have the results for FU 1 and FU 2 stored in memory, so we interpolate between FU1 and FU2 to show smooth movement.

Halo Infinite “tweening”

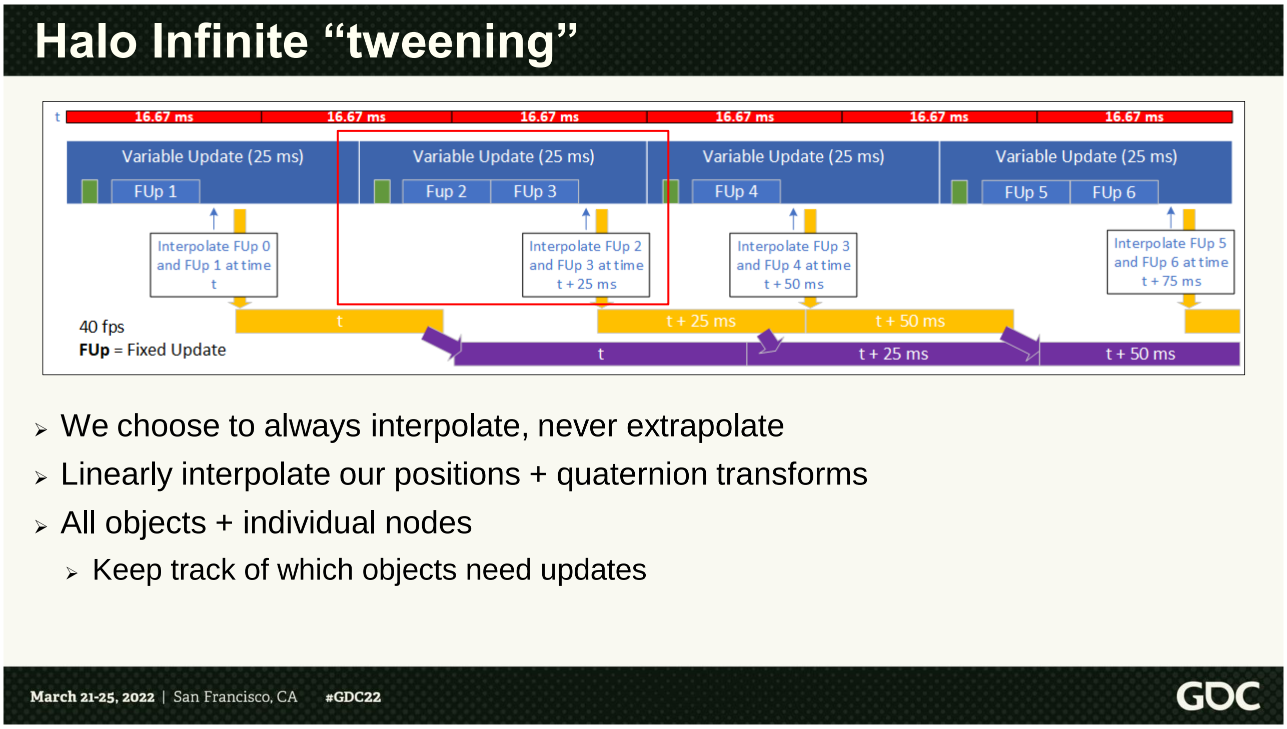
- Fixed update results are interpolated to match the variable update timeline (aka “tweening”)
  - Objects/player transforms and poses
  - Gives the illusion of smooth movement regardless of framerate


t
16.67 ms
16.67 ms
16.67 ms
Variable Update (8.33 ms)
Variable Update (8.33 ms)
Variable Update (8.33 ms)
Variable Update (8.33 ms)
Variable Update (8.33 ms)
Variable Update (8.33 ms)
FUp 1
FUp 2
FUp 3
Interpolation window t + 16.67 ms to t + 33.33 ms
Interpolate FUp 1 and FUp 2 at time t + 25 ms
t + 25 ms
t + 25 ms
120 fps
FUp = Fixed Update

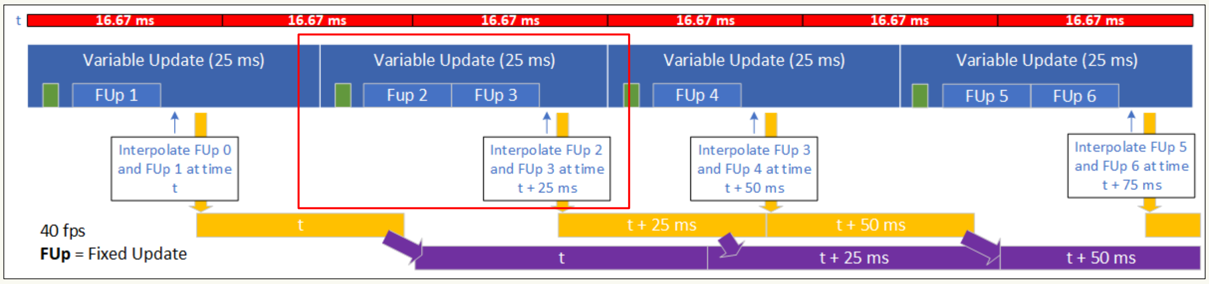
Here’s a color-coded visualization of the fixed update results that we’d be showing in that frame’s rendering pass (fixed update 1 and fixed update 2 in different shades of blue), as you can see in this frame we are drawing an interpolation of the results from FU 1 and FU 2.

Focusing on where FU 1 and FU 2 are on the timeline, please note that the results of FU 2 are not affected by the input sampled in this frame (because they were calculated last frame), and neither are the older results of FU 1. With this interpolation approach we can see then that while variable framerate systems achieve minimal input to screen latency (as they always consume latest input), inputs affecting fixed update systems can incur additional latency, and their minimum latency is bound by the fixed quantum of time we consume in each fixed update, regardless of framerate (16.67 ms).

Also note that because some simulation frames contain a fixed update, and some don’t, we can expect significantly different amounts of CPU workloads between simulation frames. In practice, though, this was not a concern for us on platforms able of running at > 60 fps, because at large we were GPU bound on those platforms, and usually our target framerate timer was the thing (intentionally) slowing down the simulation on the CPU, and not lack of CPU resources.



Halo Infinite "tweening"



- We choose to always interpolate, never extrapolate
- Linearly interpolate our positions + quaternion transforms
- All objects + individual nodes
  - Keep track of which objects need updates

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

Let's look at another example where the engine is configured to run at a 40 fps target.

In this case, each frame consumes 25 ms of time exactly, we step variable update systems forward by that amount, and we will perform 1 or two fixed updates each frame as we consume fixed update time in 16.67 ms chunks.

Even in this case, the results on screen make sense on the 40 fps real world timeline, even if fixed update simulation is advancing in 16.67 ms chunks.

We chose to interpolate between fixed update, but a different choice could have been to extrapolate. Extrapolating means in this case to project the past fixed update results forward in time using some prediction criteria. They cause different challenges:

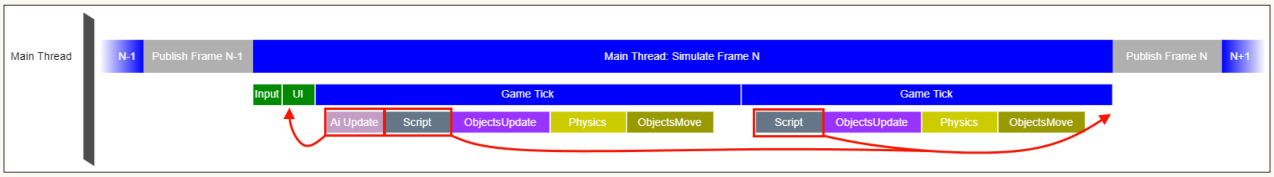
- Interpolation causes 16.67 ms worth of delay in response when dealing with fixed update system, as we must overstep the fixed update simulation in order to interpolate.
- Extrapolation can cause artefacts like incorrect extrapolation of transforms, and lead to subsequent corrections.

Our interpolation is linear across the board as we are effectively always interpolating between samples at 16.67 ms from one another, so a more expensive slerp wasn’t necessary. We interpolate all object transforms and all nodes within objects for stuff like character poses, those are the main outputs of fixed update systems. Of course, none of this treatment is needed for any simulation system updating at variable framerate as their results will already be on the right timeline.

Also, to minimize tweening workload, we keep track of which objects may need tweened state updates, and only process those.

Fixed updates on Xbox One

- At 30 fps, we must perform 2 fixed updates per frame
- Optimize on slow CPU platform (like Xbox One)
  - Turn as many systems as possible into variable framerate and bring them out of the fixed update.
    - AI, some player updates, design scripts, gameplay audio/lights/FX, object garbage collection, etc…
  - Optimize remaining fixed update systems as much as possible



When we target 30 fps (which was our target for the slower CPUs of Xbox One) we are then doing 2 fixed updates (or game ticks) per frame, and there's no need for tweening.

To maintain framerate on those low-end targets, at least from a simulation standpoint, we focused our efforts on moving first the most expensive systems that didn't need to be fixed update out into the variable update (converting them to variable framerate systems). For instance, in the picture below we show how we moved the AI update out of the game tick into the variable framerate update, and similarly for our script update in gray we moved that system to after all game ticks are finished for the frame.

Converting systems from fixed framerate to variable framerate can be complex, so we mainly focused on the heavy hitters from a simulation workload standpoint, and there's more to be done. Examples of systems that were converted to variable framerate:

- Design script logic update (as above)
- AI brains update (e.g. action selection)

- Gameplay audio processing
- Gameplay effects updates
- Light simulation updates
- Object recycling/garbage collection
- Etc…

And a bunch of new systems that were added for Halo Infinite (e.g. a lot of streaming systems) were built to be variable framerate from the ground up!

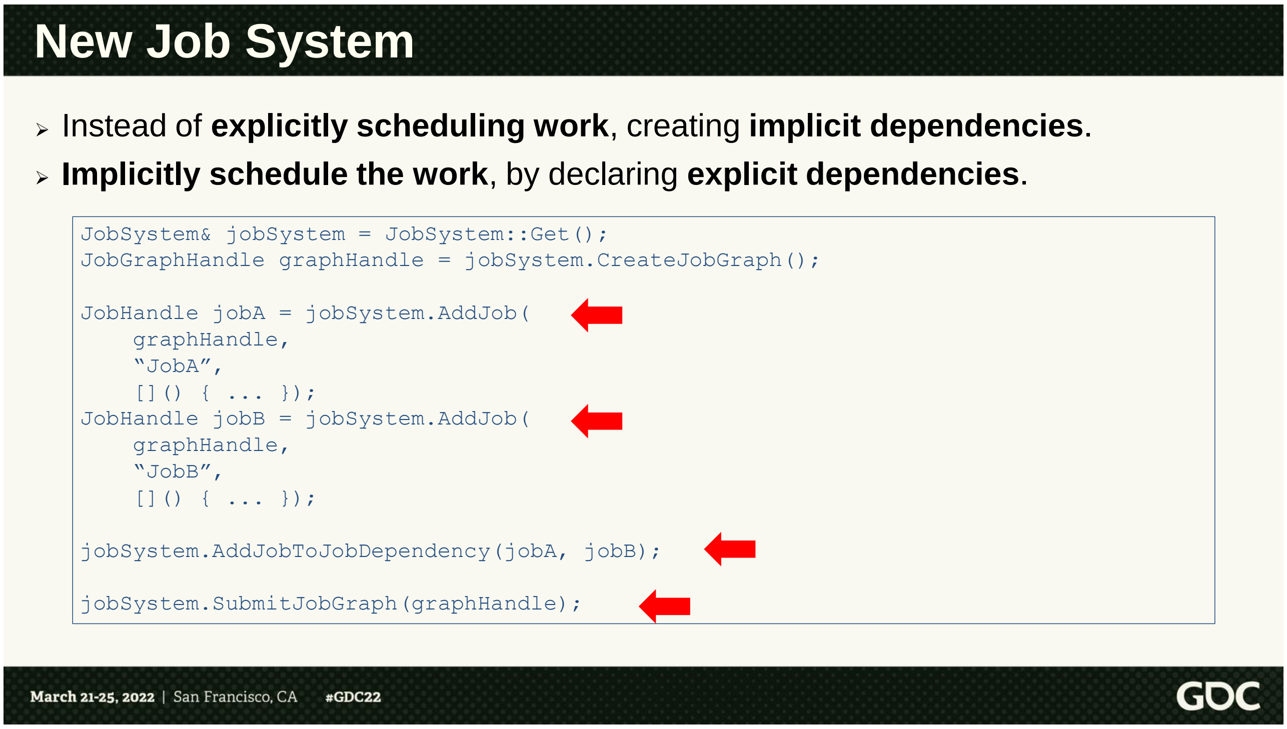
One of the most important systems to step at variable framerate is player aiming and some aspects of camera control like aim assist (this is crucial for good game feel at high fps). This is something that previous Halo games already used to do (looking directly at wall clock time) so it extended naturally to our new model.

For the systems that remained in the fixed update (because crucial to the physics-based deterministic simulation model of the game) we focused on optimizing those workloads as much as possible.

Agenda

- Halo 5 engine
  - Framerate challenges
  - Performance/efficiency/maintenance challenges
- Achieving variable framerate
- **Achieving scalable performance across hardware**
- One Frame in Halo Infinite
- Conclusions & Future Work
- Q&A

Now that we understand how variable framerate works in Halo Infinite, let’s talk about scaling naturally to various CPU sizes.



New Job System

- Instead of **explicitly scheduling work**, creating **implicit dependencies**.
- **Implicitly schedule the work**, by declaring **explicit dependencies**.

```
JobSystem& jobSystem = JobSystem::Get();
JobGraphHandle graphHandle = jobSystem.CreateJobGraph();

JobHandle jobA = jobSystem.AddJob(
    graphHandle,
    "JobA",
    []() { ... });
JobHandle jobB = jobSystem.AddJob(
    graphHandle,
    "JobB",
    []() { ... });

jobSystem.AddJobToJobDependency(jobA, jobB);

jobSystem.SubmitJobGraph(graphHandle);
```

March 21-25, 2022 | San Francisco, CA #GDC22 GDC

To front the maintenance and scalability challenges caused by our legacy “Tetris scheduling” model, we developed a new Job System.

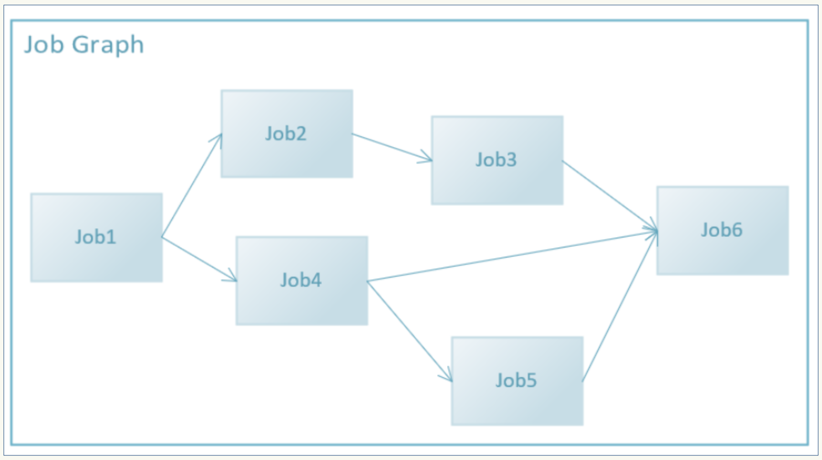
Legacy work in Halo 5 was scheduled explicitly, and its position in the main loop would enforce some implicit dependencies. With the new Job System instead items of work (jobs) were declared upfront with their dependencies, and execution would proceed according to these dependencies through automatic scheduling of jobs on the available execution resources.

Here’s a snippet of code showing a trivial interaction with this system, we are creating two jobs (job A and job B), and we are adding a dependency between these two jobs specifying that job A needs to complete before job B starts. And then the graph is submitted for execution.

For the rest of the presentation, when we talk about an item of work as “Job” we explicitly refer to work that is scheduled through the Job system, in general, for all cooperatively scheduled work in the engine, we use the term “Workload”.

New Job System

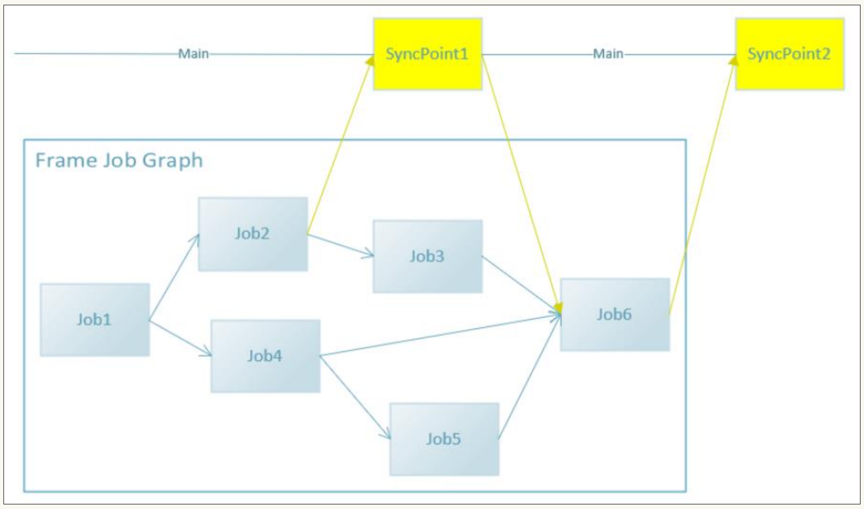
- The system schedules work according to the dependencies declared



The system is in charge of scheduling the work as efficiently as possible on the available execution resources. For instance, if we are given the graph in the slide here, the system would schedule job1 to finish before either job2 or job4 starts, job2 and job4 could then run in parallel with one another, as could job3 and job4 since there's no dependency between them, and so on.

New Job System: sync points

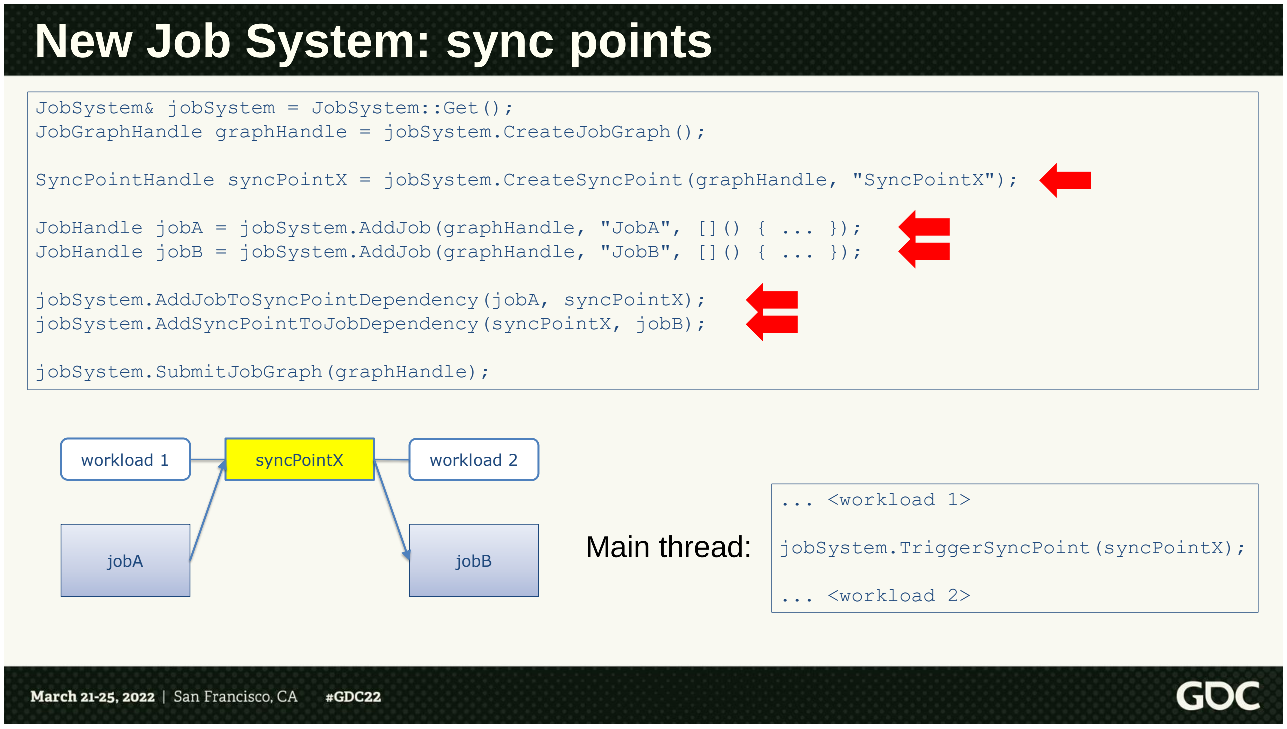
- Synchronizing with legacy code



Since we were adding this system on top of a legacy codebase, and we didn’t have time to re-write all of our legacy system, we introduced a way to synchronize execution of systems leveraging the job system with legacy systems updated by the main thread (or other legacy threads), this was achieved through the use of Sync Points.

Basically, jobs can take predecessor or successor dependencies with sync points on top of other jobs, and job execution will be fired off accordingly as main proceeds with execution at the same time as the job graph is running, and the main thread would join execution of jobs if work was needed to complete before execution could move beyond a given sync point. This approach can be used for main thread or other threads where we mix legacy work with new style jobs, and it was crucial to avoid having to wholly migrate execution of all legacy systems.

The vision for sync points is as a stop-gap solution as we are converting legacy work to only run through jobs.



New Job System: sync points

```
JobSystem& jobSystem = JobSystem::Get();
JobGraphHandle graphHandle = jobSystem.CreateJobGraph();

SyncPointHandle syncPointX = jobSystem.CreateSyncPoint(graphHandle, "SyncPointX");

JobHandle jobA = jobSystem.AddJob(graphHandle, "JobA", []() { ... });
JobHandle jobB = jobSystem.AddJob(graphHandle, "JobB", []() { ... });

jobSystem.AddJobToSyncPointDependency(jobA, syncPointX);
jobSystem.AddSyncPointToJobDependency(syncPointX, jobB);

jobSystem.SubmitJobGraph(graphHandle);
```

Main thread:

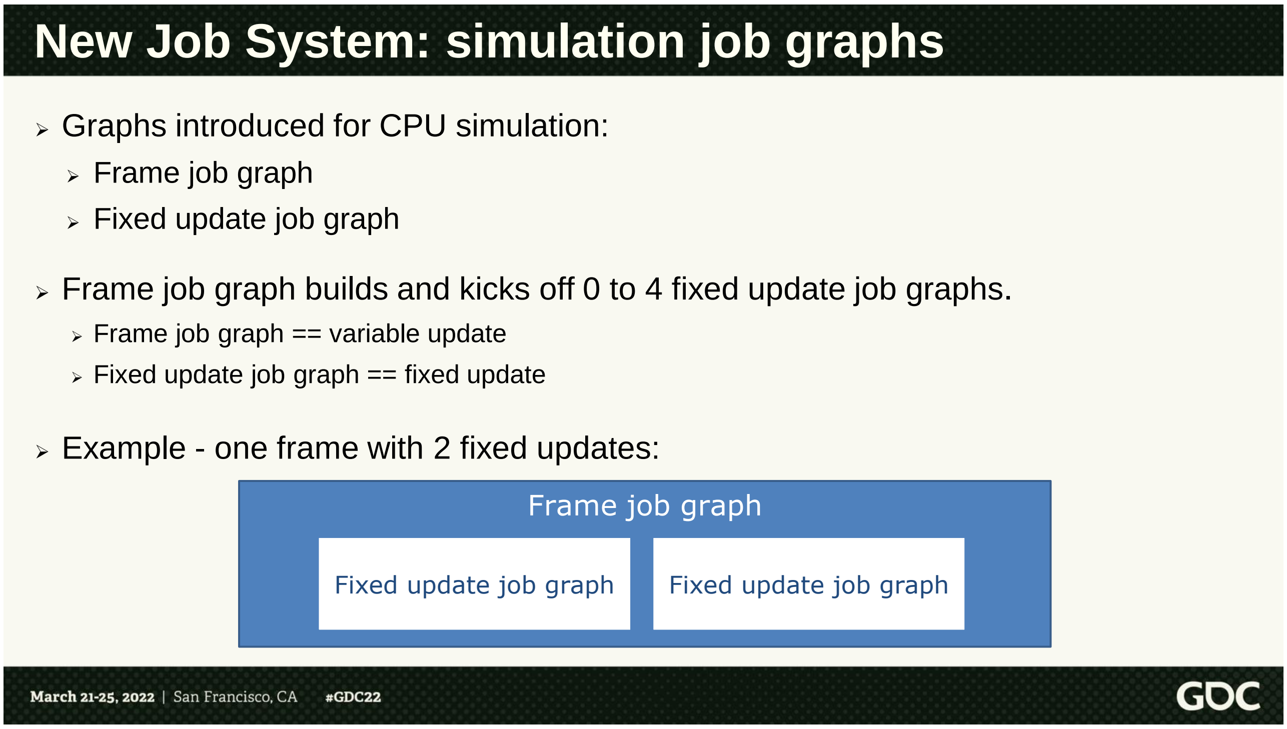
```
... <workload 1>

jobSystem.TriggerSyncPoint(syncPointX);

... <workload 2>
```

To better understand sync points, let’s look at a simple code example.

We again create a job graph with 2 jobs, this time we also add a sync point called “SyncPointX”.

We specify 2 dependencies during construction of the job graph, specifically that job A must terminate before triggering sync point X, and that sync point X must trigger before job B can start, and then we submit the job graph.

In this case, Job A will be able to start immediately on the available execution resources, and at the same time we will have a legacy thread executing non-jobified workloads (for instance main thread executing independently). When that legacy thread triggers the sync point, we will then be able to start job B. Effectively, this means that in this example we are specifying that workload 1 on the legacy thread must be complete before we start job B, and that job A must finish before we can start legacy workload 2.



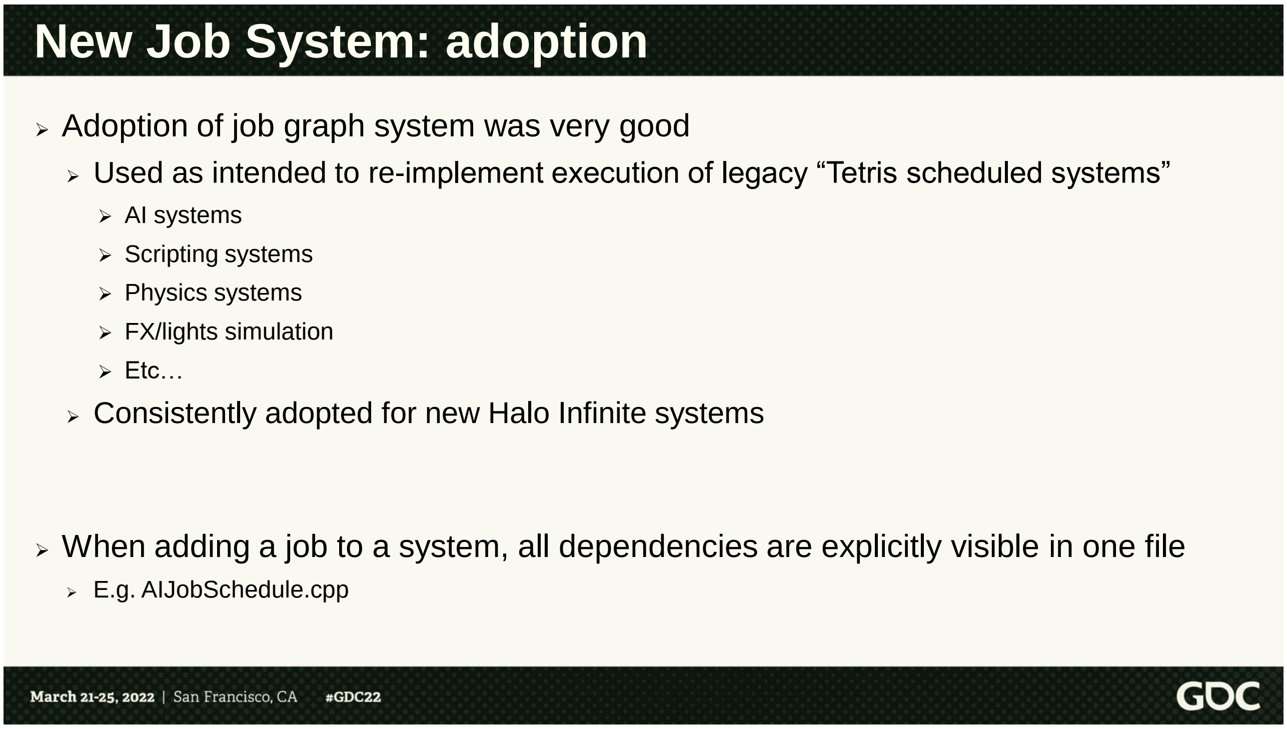
New Job System: simulation job graphs

- Graphs introduced for CPU simulation:
  - Frame job graph
  - Fixed update job graph
- Frame job graph builds and kicks off 0 to 4 fixed update job graphs.
  - Frame job graph == variable update
  - Fixed update job graph == fixed update
- Example - one frame with 2 fixed updates:

Frame job graph
Fixed update job graph
Fixed update job graph

For simulation, we mainly introduced:

- Frame job graph, that gets built at the beginning of the frame with all jobs representing a variable framerate update, and submit for execution as main thread proceeds and synchronizes with this graph through sync points.
- Fixed update job graph, that executes within the frame job graph 0 to 4 times depending on how the game is running (how many game ticks we want this frame).



New Job System: adoption

- Adoption of job graph system was very good
  - Used as intended to re-implement execution of legacy “Tetris scheduled systems”
    - AI systems
    - Scripting systems
    - Physics systems
    - FX/lights simulation
    - Etc…
  - Consistently adopted for new Halo Infinite systems
- When adding a job to a system, all dependencies are explicitly visible in one file
  - E.g. AIJobSchedule.cpp

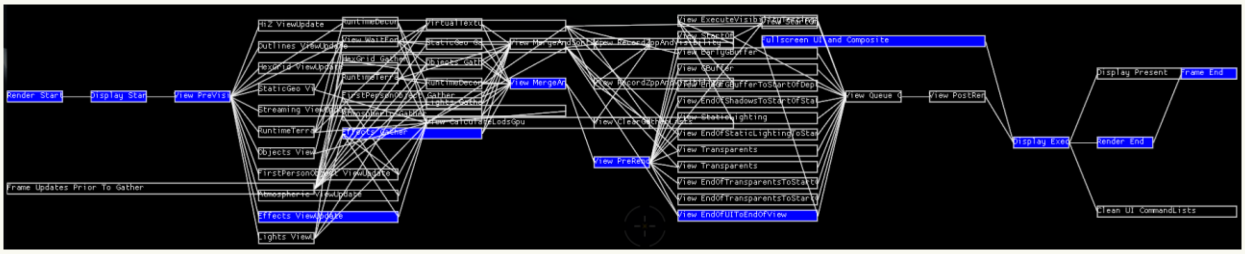
Overall, we found that adoption of the new job system was very good, and it was used as intended to re-implement parallel execution of some legacy systems like AI, FX/Lights simulation, etc… But even better, it was consistently adopted for new Halo Infinite systems added during production of the game.

Other than the advantages of being able to understand execution dependencies much better, another great perk of this approach was that for most systems the code scheduling their job graph workload was all contained in a single file, so any engineer could crack open that file to see all workloads belonging to a high-level game system, and the execution dependencies between each job.

New Job System: multi-threaded rendering

- Multi-threaded rendering through the job system
  - Job system was used as the backbone for our new multi-threaded renderer
  - Halo Infinite renderer required more CPU bandwidth
- Halo Infinite has **no render thread**

In-game render job graph visualization



A great result of introducing the new job system is that this was used as the backbone for our new multi-threaded renderer. This is not a graphics talk, but I want to mention this because it is important as it was a major scalability problem in the legacy Halo engine.

The legacy Halo engine had a mostly single-threaded renderer that used lots of console-specific tricks to achieve high efficiency with very low CPU bandwidth requirements. However, Halo Infinite moved away from console-only, and PC D3D code has much higher CPU overhead, we also brought back split-screen, and introduced a slew of new rendering techniques. It became very apparent that we had to lean heavily into multi-threaded rendering to achieve good performance.

Using the job system, the render thread was fully converted into jobs and a large amount of parallelism was achieved. Specifically, Halo Infinite has no render thread, and all rendering happens as part of jobs running in the render job graph.

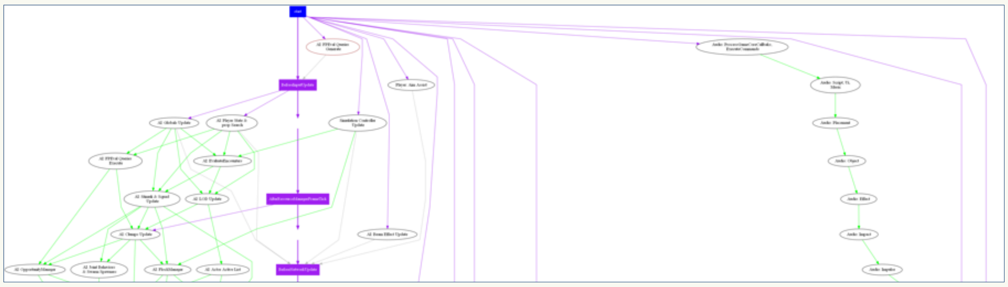
We have in-game visualization tools for job graphs, for instance in the slide you can see a screenshot of the rendering job graph for a sample frame, to get an idea of the jobs we have running in there and the amount of parallelism that we allow.

Also interesting is to note how this debug shows in blue jobs that are part of the critical path of render execution.

New Job System: optimization notes

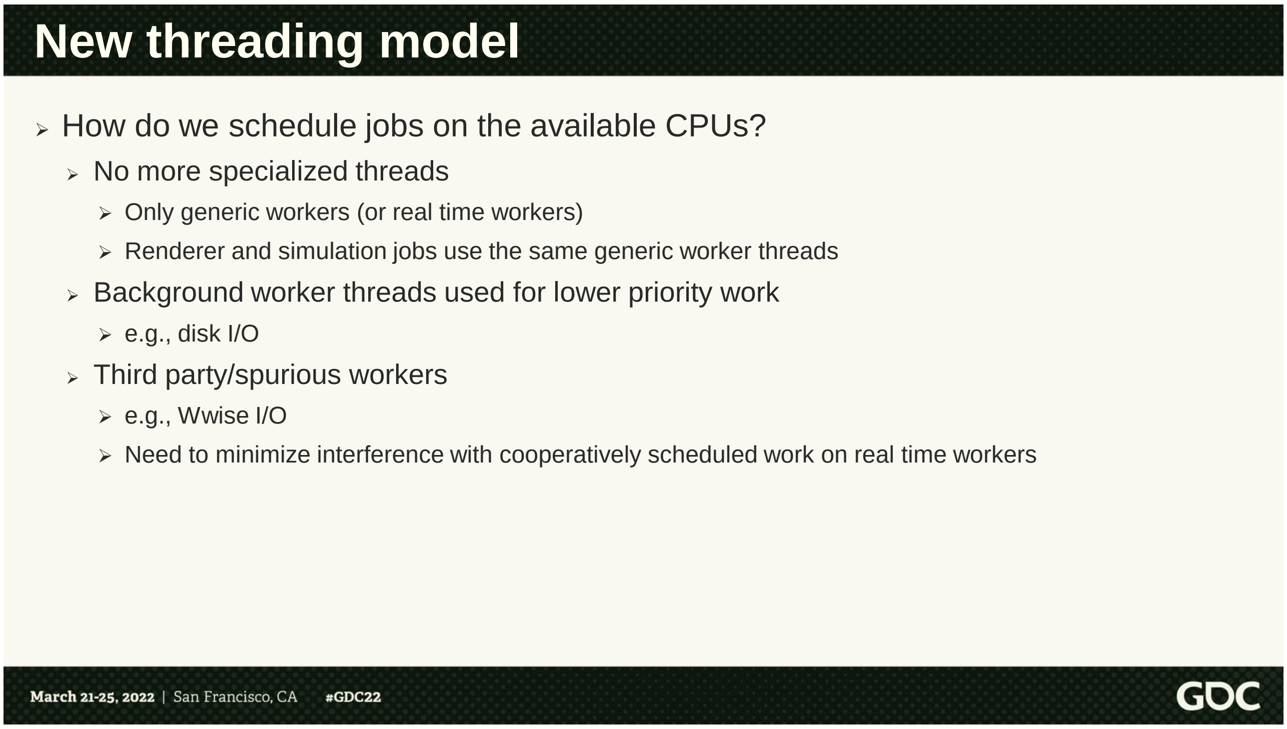
- Optimizing CPU execution
  - Optimizing jobs along the critical path
  - Relaxing dependencies between jobs to achieve greater parallelism
  - Reorder jobs by shuffling dependencies

Offline inspection of job graph topology



Optimizing CPU execution of a job graph becomes a targeted optimization effort for jobs along the critical path (like in the screenshot in the previous slide). Also, relaxing dependencies between jobs in order to achieve greater parallelism, and finally reordering jobs can be achieved simply by shuffling execution dependencies.

Other than in-game visualization tools like the one shown in the previous slide, we also have ways of dumping out the job graph for offline inspection.



New threading model

- How do we schedule jobs on the available CPUs?
  - No more specialized threads
    - Only generic workers (or real time workers)
    - Renderer and simulation jobs use the same generic worker threads
  - Background worker threads used for lower priority work
    - e.g., disk I/O
  - Third party/spurious workers
    - e.g., Wwise I/O
    - Need to minimize interference with cooperatively scheduled work on real time workers

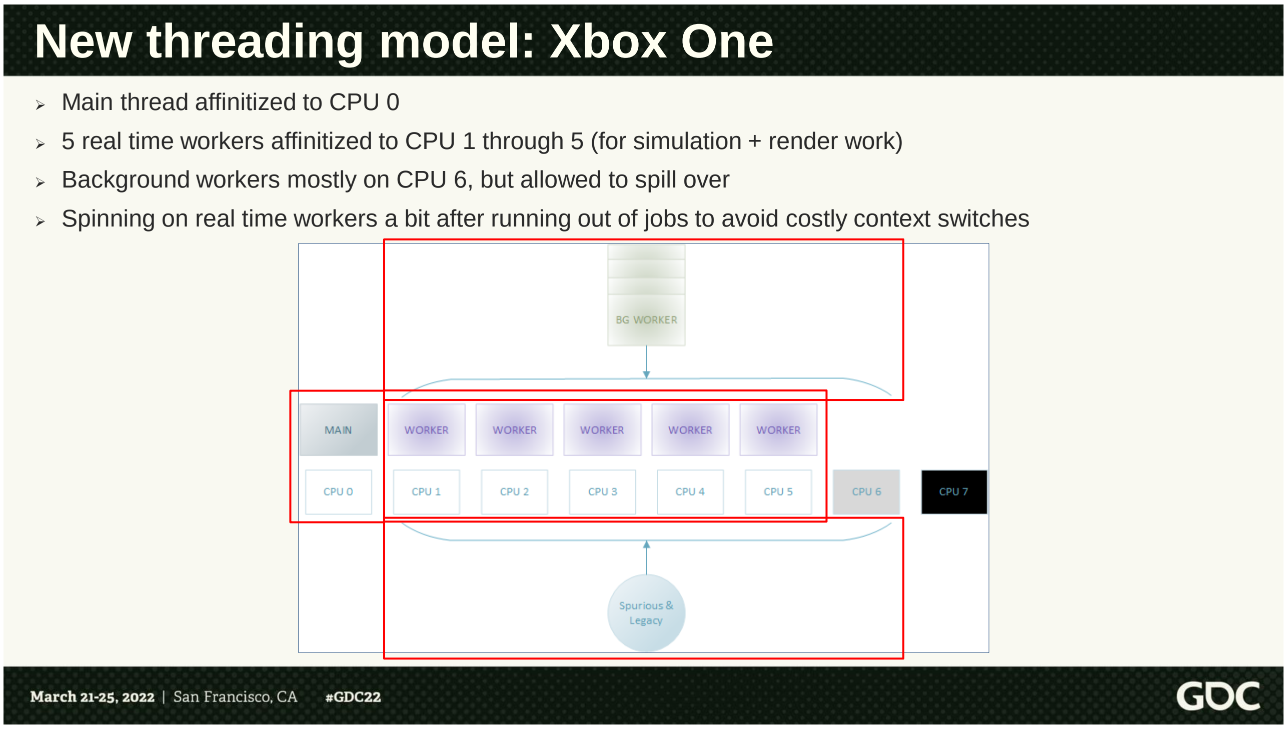
March 21-25, 2022 | San Francisco, CA #GDC22

We've said before that the job system tries to schedule jobs according to their execution dependencies on any available execution resource. But what are those resources in practice?

In Halo Infinite, those are mainly generic worker threads. While previously we had specialized worker threads for various workloads (for instance AI pre-update must run on worker 2), in Halo Infinite we only have generic workers (which we also call real-time workers). All rendering and simulation can run on any of these threads, and those are the execution resources that the job system uses to schedule jobs.

We also have background worker threads used for lower priority work that is latency-tolerant, for example disk I/O. These are also a group of generic workers.

On top of that, we have what we call spurious workers. These are threads that are usually created by third party software for specialized workloads, and on which we don't have a good amount of control. For instance, the Wwise audio package will create a thread for certain I/O operations. We carefully configure spurious threads to try and minimize interference with the real time workers. Because the rest of the work (jobs) are scheduled cooperatively on the real time worker threads, interference from spurious threads pre-empting that work can drastically reduce execution efficiency.



New threading model: Xbox One

- Main thread affinitized to CPU 0
- 5 real time workers affinitized to CPU 1 through 5 (for simulation + render work)
- Background workers mostly on CPU 6, but allowed to spill over
- Spinning on real time workers a bit after running out of jobs to avoid costly context switches

BG WORKER
MAIN
WORKER
WORKER
WORKER
WORKER
WORKER
CPU 0
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7
Spurious & Legacy

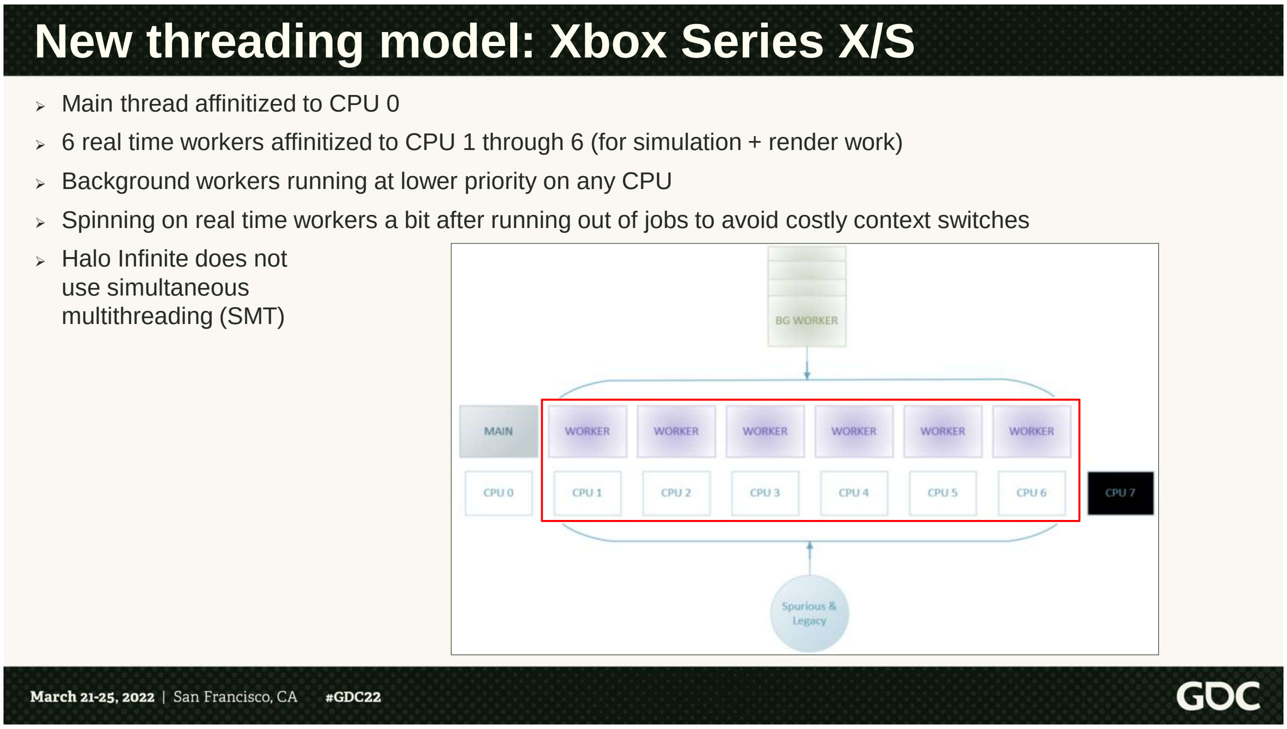
Let’s take a look at the threading model for various hardware targets.

On Xbox One, we have the main thread affinitized to CPU 0, and 5 generic workers affinitized to CPU 1 through 5. These are responsible to run all of the simulation and rendering work. These threads also spin for a bit after running out of jobs to execute to see if a new job is will show up for execution. We do this to try and avoid expensive context switches as much as possible, and it greatly improves our efficiency on Xbox One (slow CPU), where we know we have dedicated CPU resources to the game.

Background worker threads are allowed to run on CPU 1 through 6 at lower priority, they will never pre-empt any of the real time workers and often will decide to run on CPU 6 as that’s more often than not the only CPU available for them.

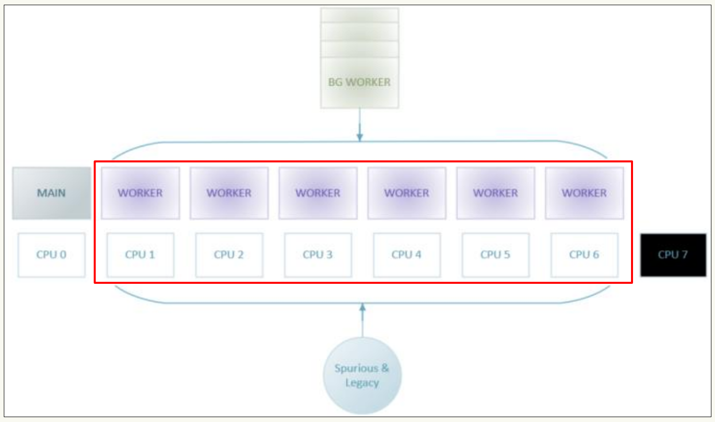
CPU 7 is reserved for the system.

Spurious threads will run also on CPU 6 or CPU 1 through 5, depending on the case, with the right priority.



New threading model: Xbox Series X/S

- Main thread affinitized to CPU 0
- 6 real time workers affinitized to CPU 1 through 6 (for simulation + render work)
- Background workers running at lower priority on any CPU
- Spinning on real time workers a bit after running out of jobs to avoid costly context switches
- Halo Infinite does not use simultaneous multithreading (SMT)



Xbox Series consoles also have 8 physical cores, and the title gets access to 7, this time CPU 6 is also exclusive to the title!
The hardware also gives the choice of enabling simultaneous multithreading or SMT for the title, this configuration allows the game to leverage more logical cores for execution but at a slightly reduced clock rate (3.6 GHz vs 3.8 GHz). More threads can lead to more throughput depending on your engine even at the reduced clock rate. For Halo infinite we tested both modes and ultimately decided to not leverage SMT, that is we run at the higher clock rate with fewer threads, because even without SMT the hardware exposes 7 dedicated very fast cores, and this worked better for us.

On Xbox Series X we affinitize Main thread to CPU 0 like on Xbox One, but this time we create 6 real time workers for CPU 1 through 6. Like on Xbox One, we spin a little bit on real time worker after each job to try and avoid needless context switches (because the hardware is dedicated to the title). This leads to great execution efficiency on Xbox Series consoles.

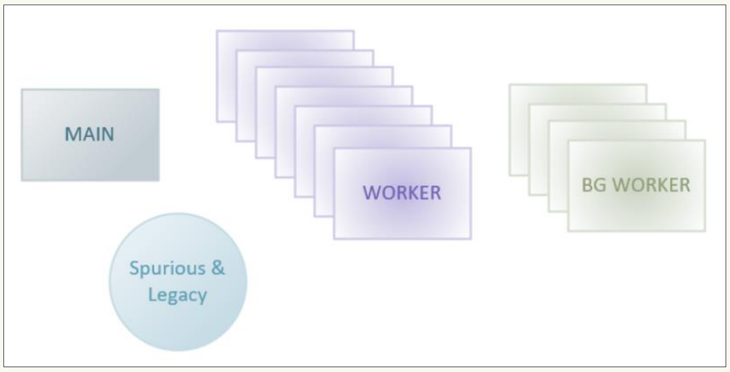
Like on Xbox One, background workers run at lower priority on any CPU, and CPU 7 is reserved to the system and inaccessible by the title.

Overall, then, a model very similar to our Xbox One threading model, which simplified maintenance too.

New threading model: Windows PC

- Main thread
- One real time worker per available logical core up to 9
- No affinitization
- No spinning

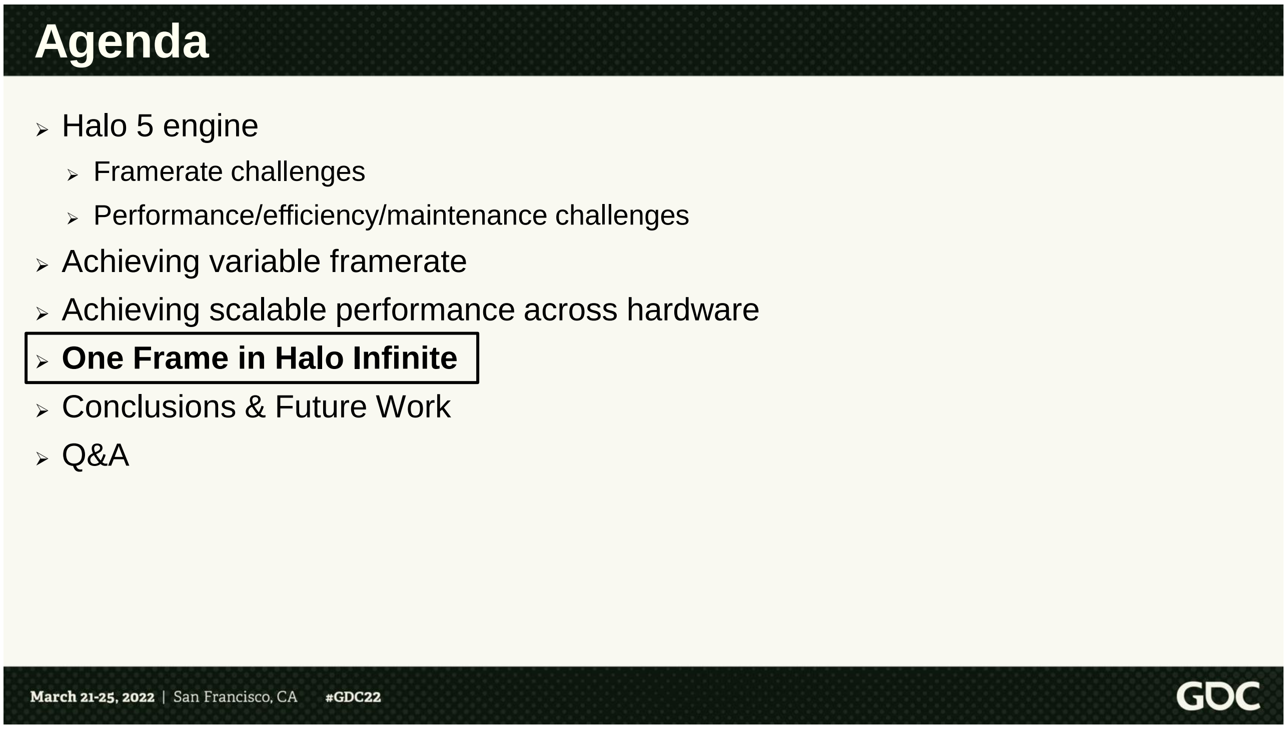
Example for 8 logical cores



On PC, the situation is a bit different.

First of all, a Windows PC is a shared platform, and a game is never guaranteed any execution resources. So, we do not affinitize (we let any thread run on any available logical core). And we typically do not spin on any worker thread (if we did, from the perspective of the OS, we'd be eating part of our quantum, doing useless work), so we intentionally yield a core any time a thread runs out of jobs to execute immediately.

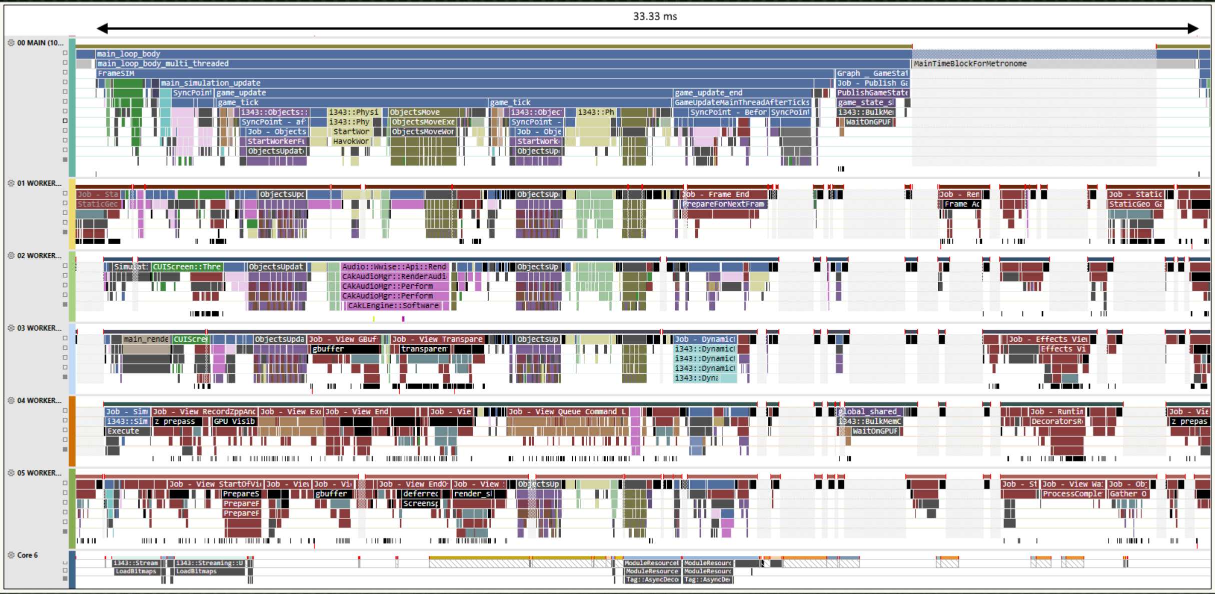
Other than the main thread, we create one real time worker for each available logical core up to 9. 9 was the magic number for us and we picked it empirically as we profiled that going beyond that started producing lower returns due to overhead intrinsic to our scheduling systems. So for example, in the picture we have an 8 logical core machine, we'll have main thread + 7 real time workers, a few lower priority background worker threads, and finally the legacy/spurious threads.


Agenda

- Halo 5 engine
  - Framerate challenges
  - Performance/efficiency/maintenance challenges
- Achieving variable framerate
- Achieving scalable performance across hardware
- **One Frame in Halo Infinite**
- Conclusions & Future Work
- Q&A

Ok, now we have all the tools to dissect together one frame in Halo infinite, so let’s check out how it looks across some sample of hardware.

One Frame in Halo Infinite – Xbox One



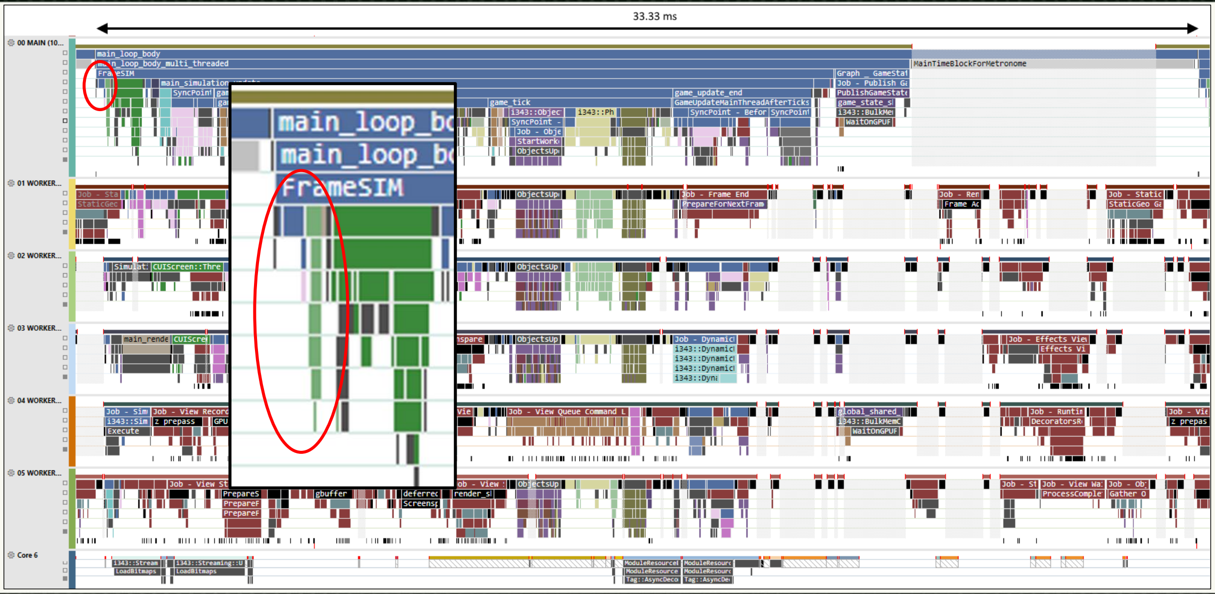
Starting with Xbox One:

This is a screenshot from PIX, which is a profiling tool that comes with the Xbox GDK. There's also a Windows version for PC, and we ended up using this tool across the board for all our console and PC profiling, highly recommend it.

First of all, let's note that we see the main thread and our 5 real time worker threads, affinitized to each fully dedicated CPU. Core 6 is the half-core and there are a couple of threads running on it in this screenshot.

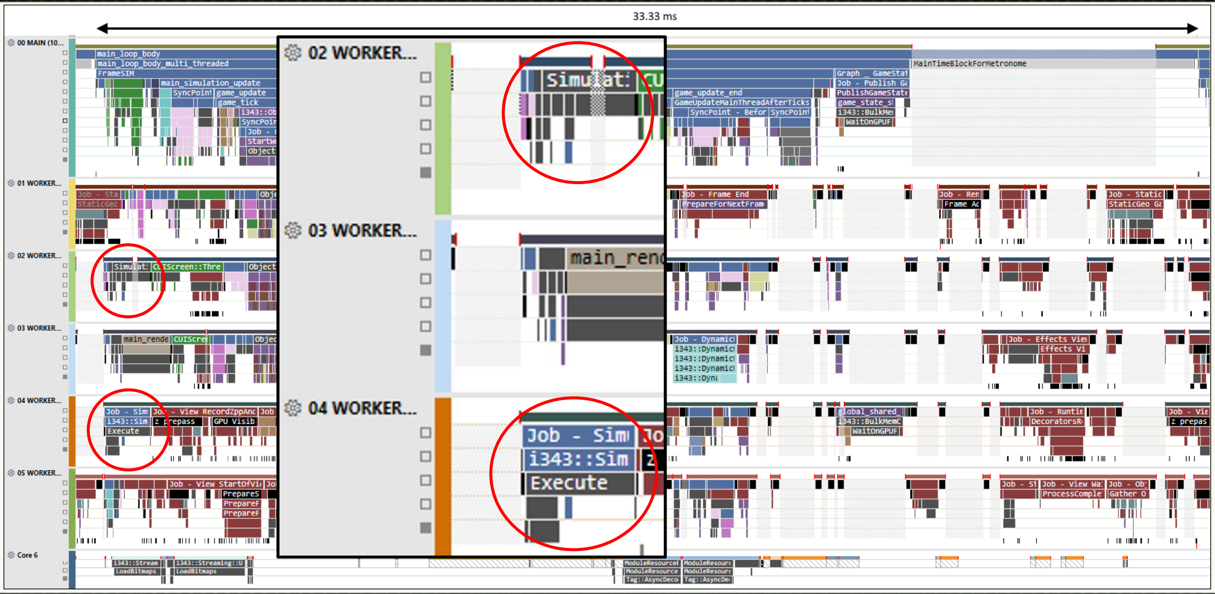
There is no render thread, in fact if you squint you can probably see some rendering jobs sprinkled throughout the frame.

One Frame in Halo Infinite – Xbox One



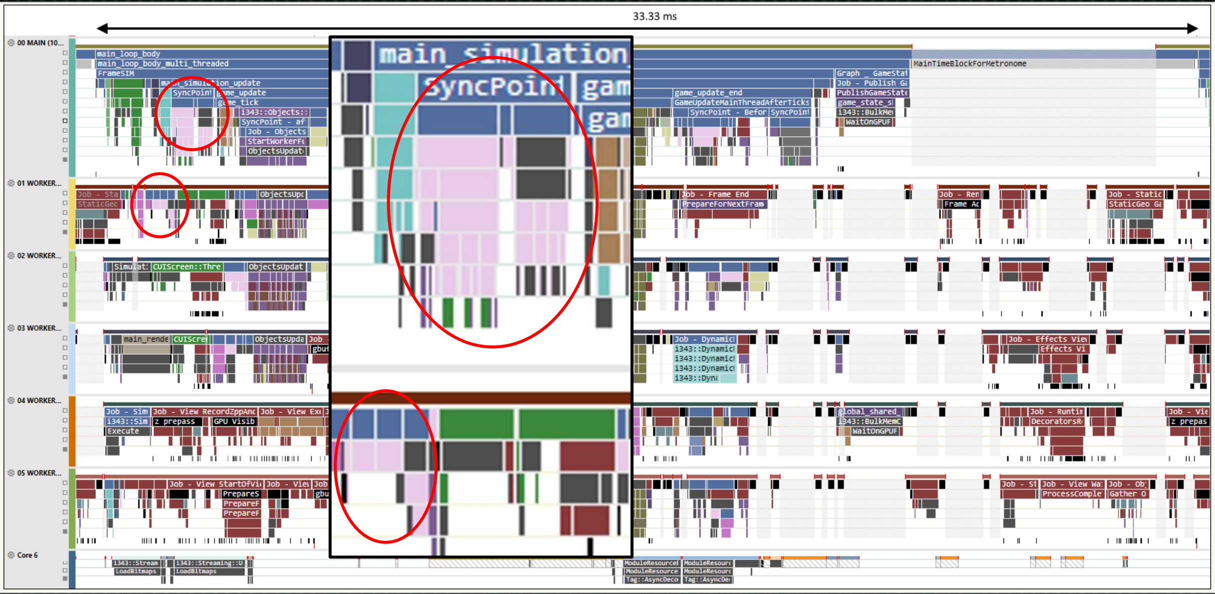
Starting at the beginning of the simulation, the very first thing we do is build the frame job graph with all our frame-level simulation jobs. After we build it, we quickly submit it for execution. The tiny green workload on the main thread inside this circle is input sampling.

One Frame in Halo Infinite – Xbox One



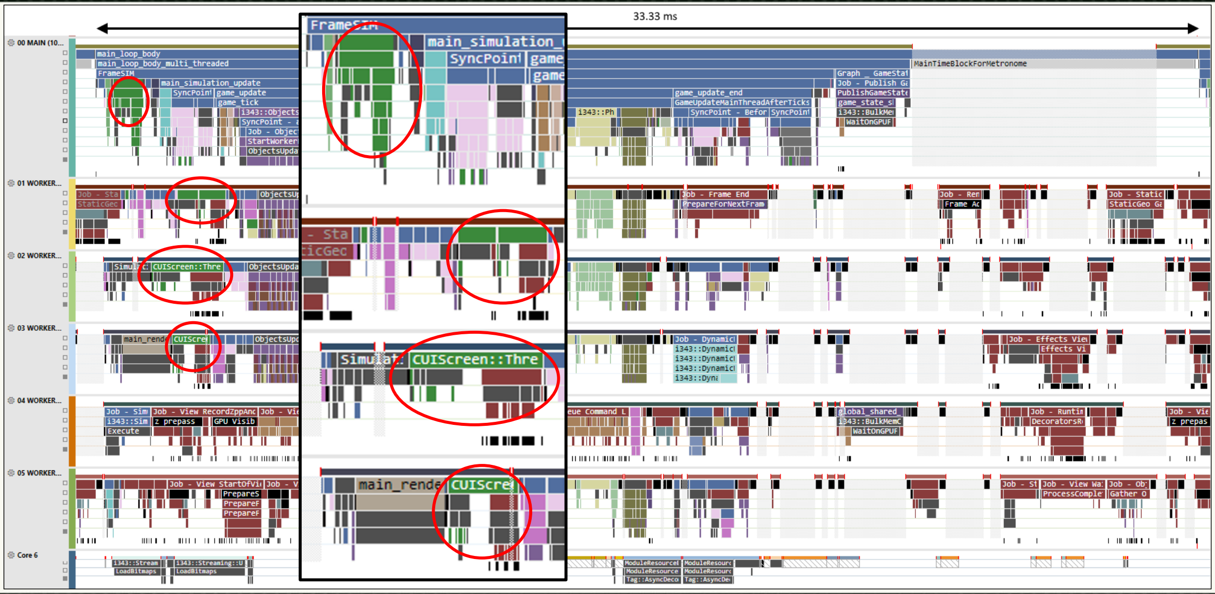
Here's an example of jobs scheduled early on worker 04 and 02, these may run on any worker, but the job system found that worker 04 and 02 were available to start executing the work so it scheduled them there. This specific workload is one of our streaming systems, establishing relevancy of a bunch of "entities" in the game to figure out what can be streamed out and what needs to start streaming in.

One Frame in Halo Infinite – Xbox One



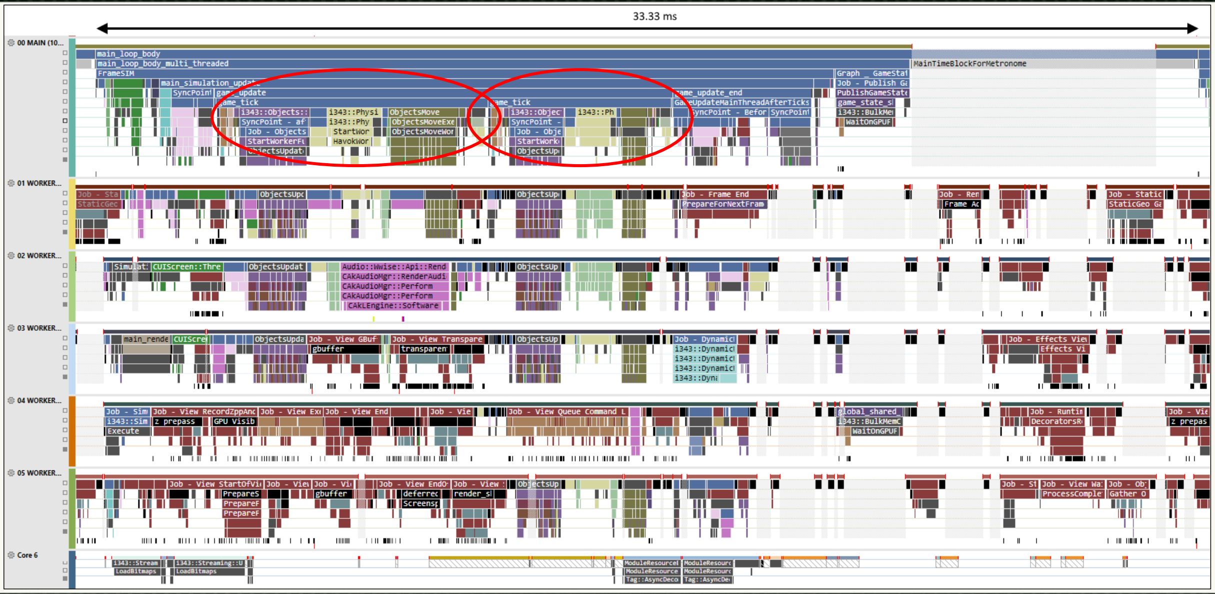
These ones highlighted here in pink show a couple of AI jobs that before would have been explicitly scheduled on the specialized worker thread. We also show that the main thread itself can pick up jobs (like it does in this case) if execution reaches a point that jobs need to have finished to satisfy a sync point.

One Frame in Halo Infinite – Xbox One



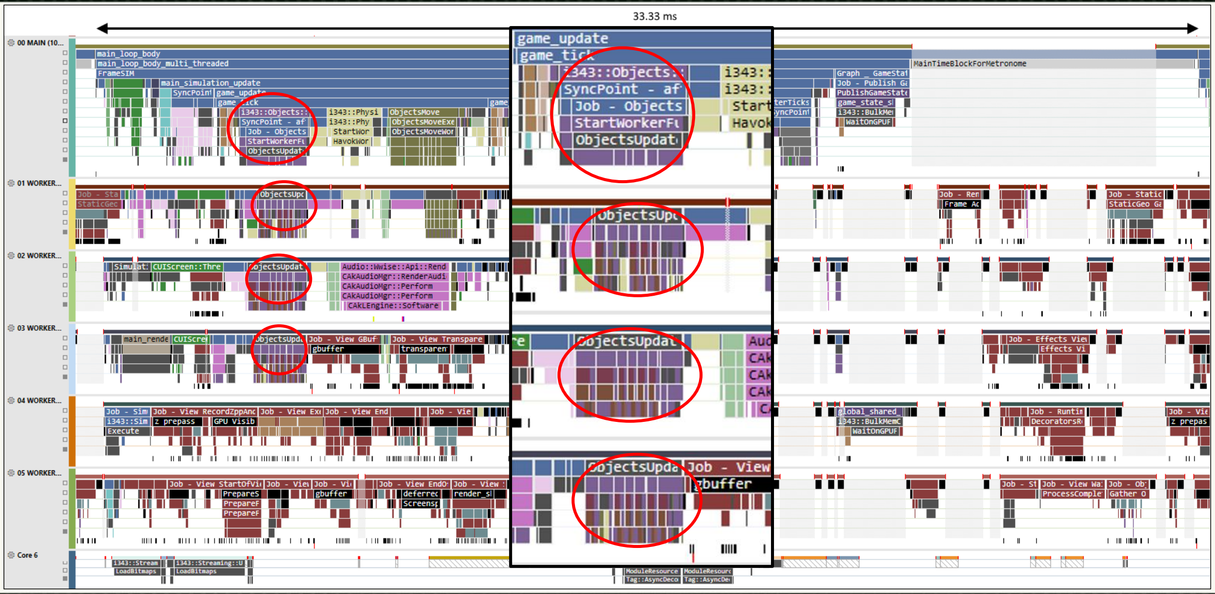
We also see in green a good amount of UI work early on. These jobs are responsible for updating our UI/Hud and build the command lists needed for their rendering.

One Frame in Halo Infinite – Xbox One



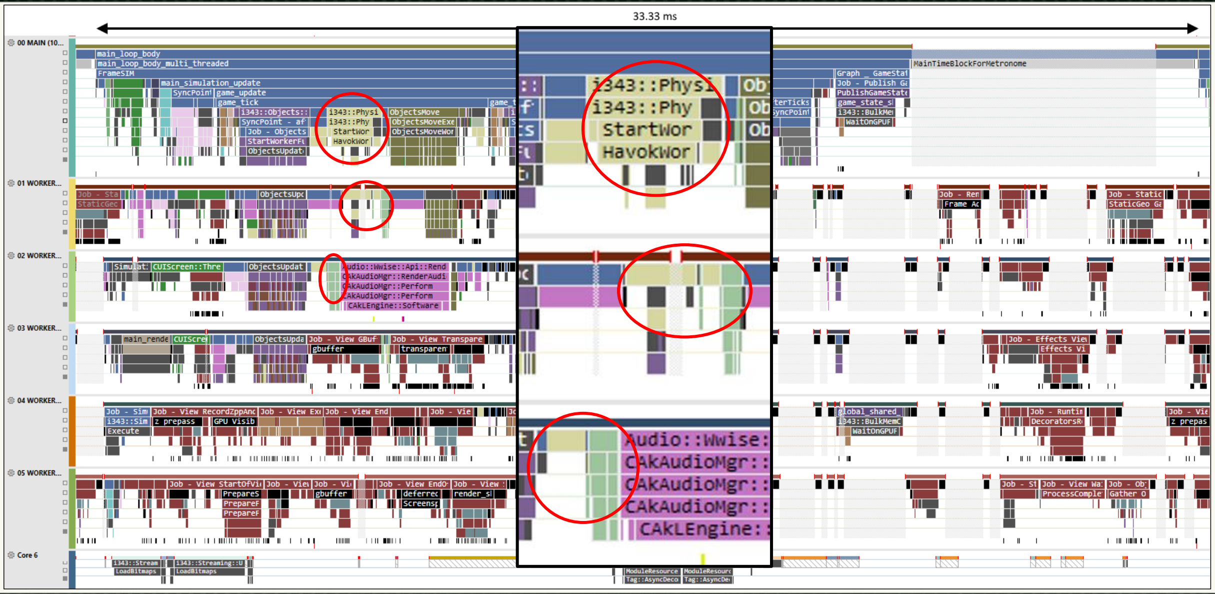
Moving on to the core of the simulation, we can see two fixed updates (or game ticks) in this frame, which makes sense since we are executing at 30 fps on Xbox One.

One Frame in Halo Infinite – Xbox One



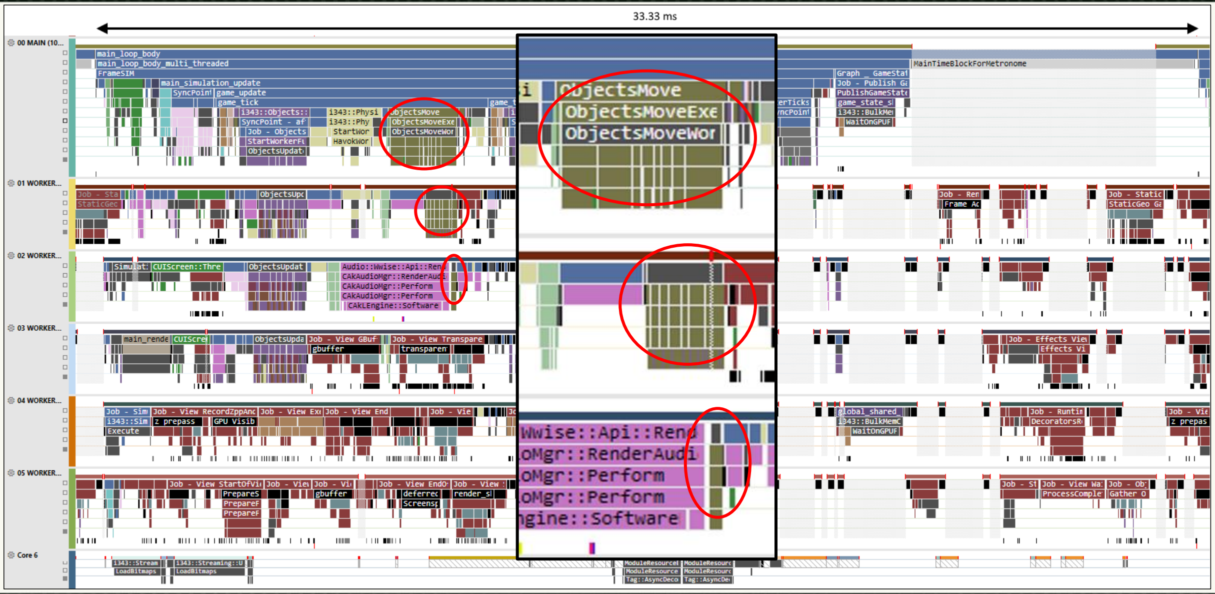
Inside a fixed update, the main phases are our parallel object update (updating the state of all active simulating objects in the scene)….

One Frame in Halo Infinite – Xbox One



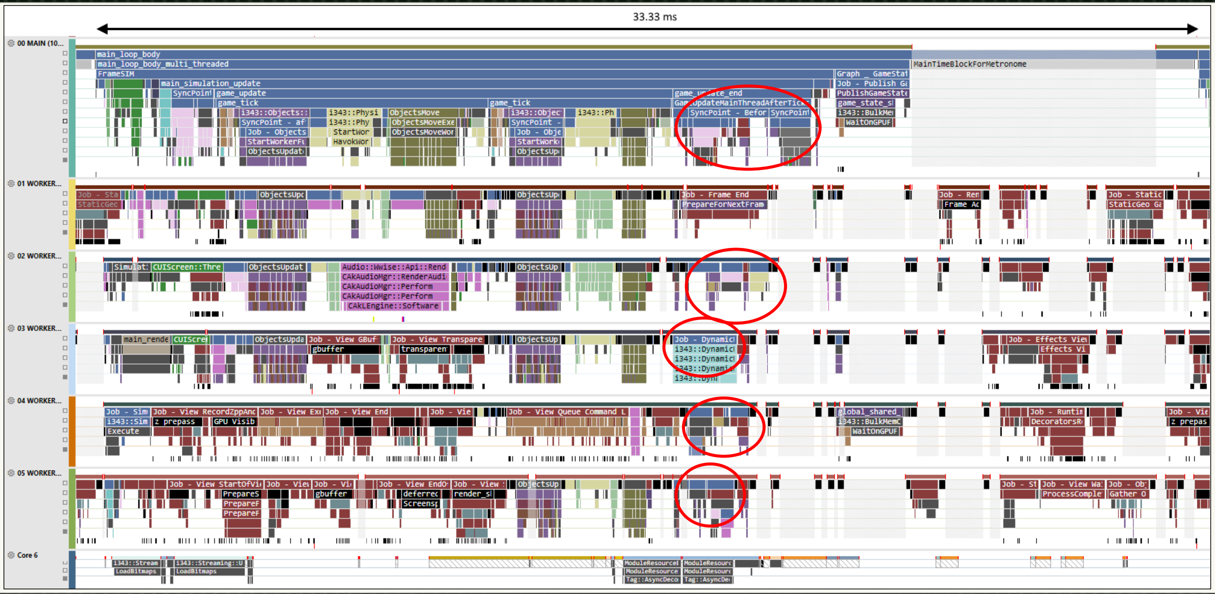
… followed by the physics update in yellow and in parallel our animation update in green; physics is the core of our deterministic fixed rate simulation.

One Frame in Halo Infinite – Xbox One



… followed by the “move” operation on all moving objects that updates their transforms and poses.

One Frame in Halo Infinite – Xbox One

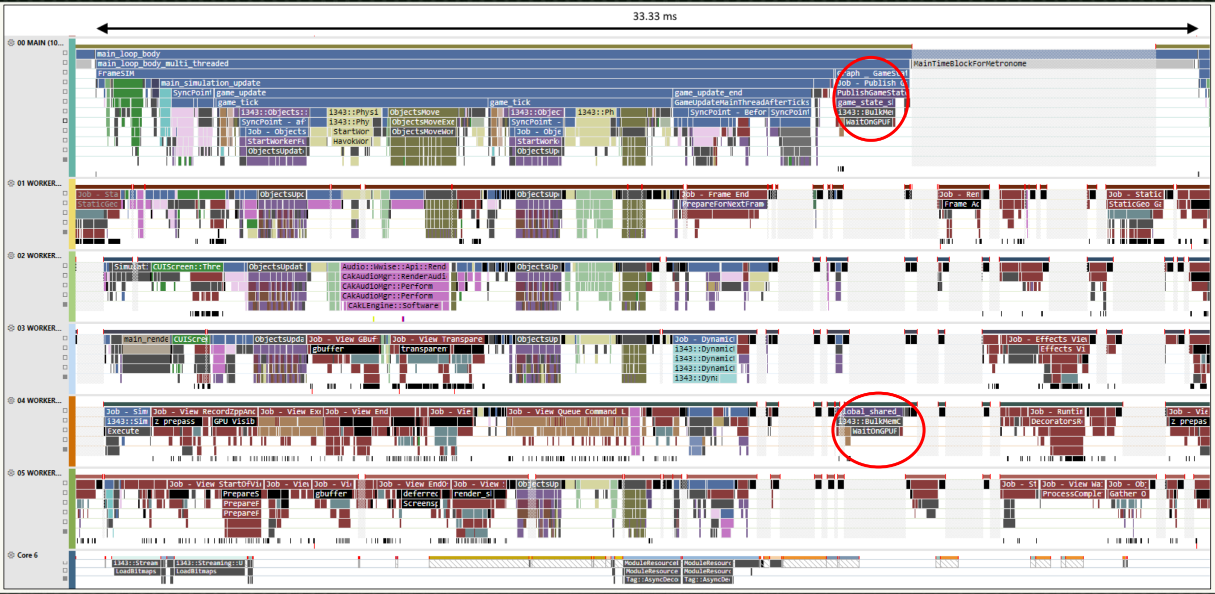


After the fixed updates for the frame, we have a few more variable framerate systems running as part of the frame job graph, in there we spot:
- More AI workloads in pink

- In light blue, the update for our dynamic world state system responsible for stuff like time of day.

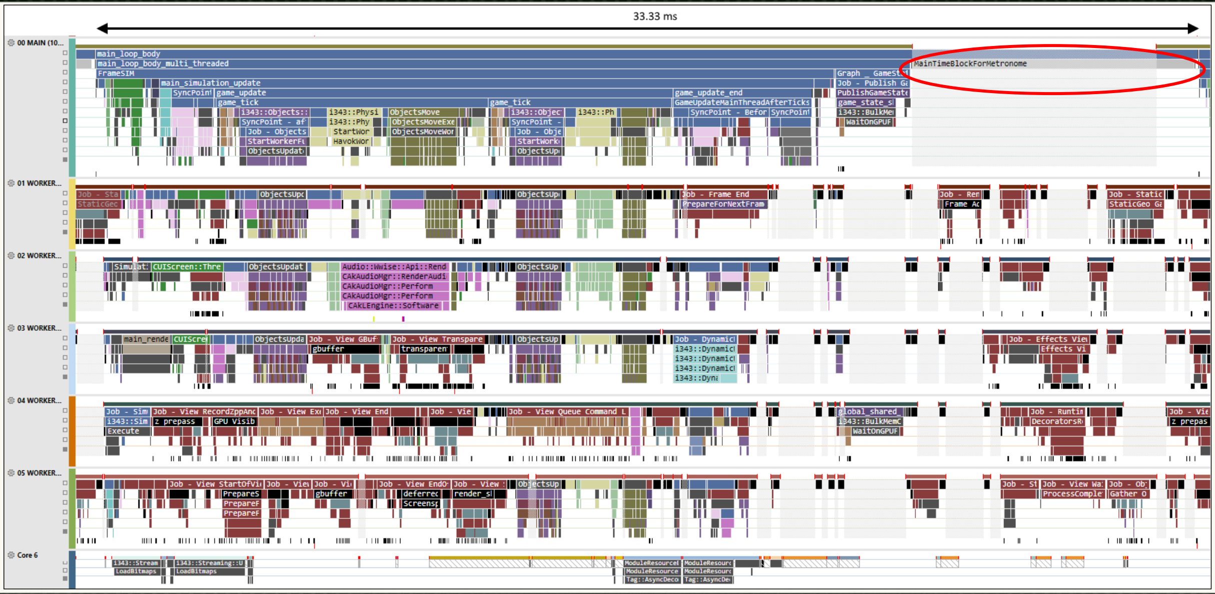
- In gray on the main thread the script update, which used to be part of the game tick in Halo 5, updating design logic.

One Frame in Halo Infinite – Xbox One



After the simulation for the frame is complete, we do our publish operation that copies the results to the renderer. We do this in bulk using a DMA accelerated copy (on console).

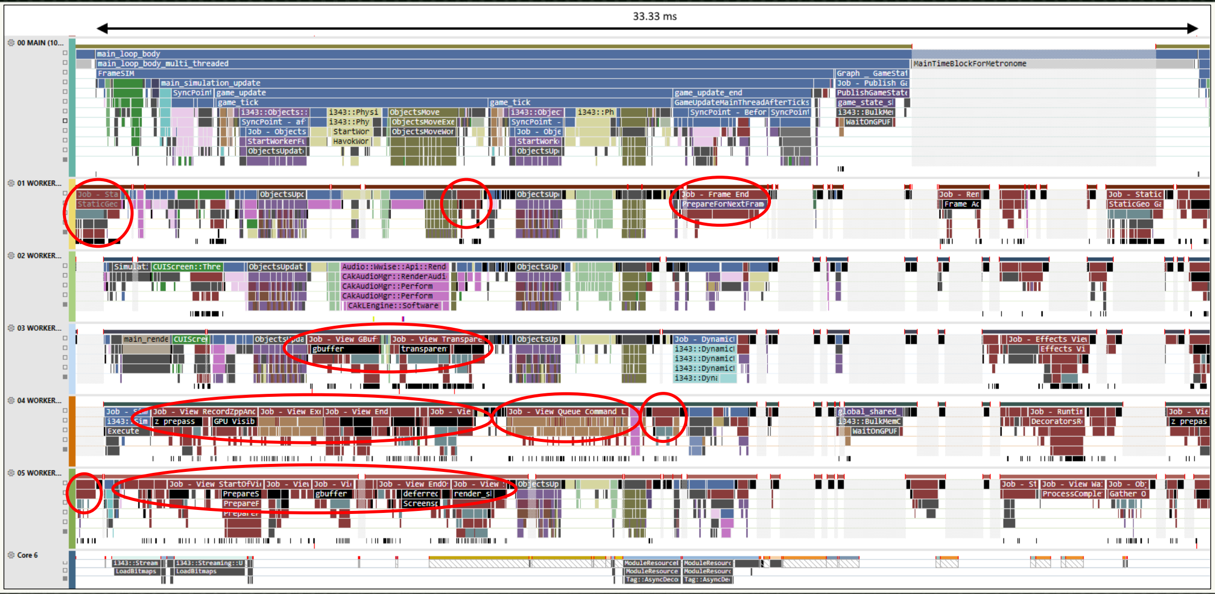
One Frame in Halo Infinite – Xbox One



March 21-25, 2022 | San Francisco, CA #GDC22

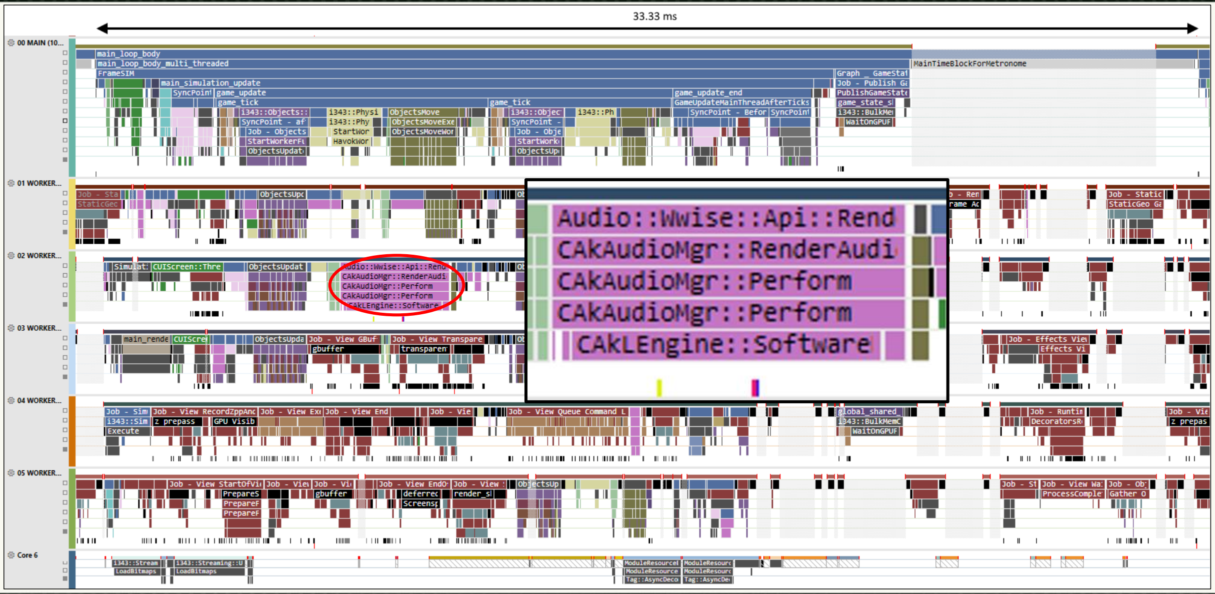
After publishing the results, then, if we were fast enough to stick with our framerate target of 33.33 ms, we wait for the next beat of the timer (aka. “metronome”) to start the next simulation frame.

One Frame in Halo Infinite – Xbox One



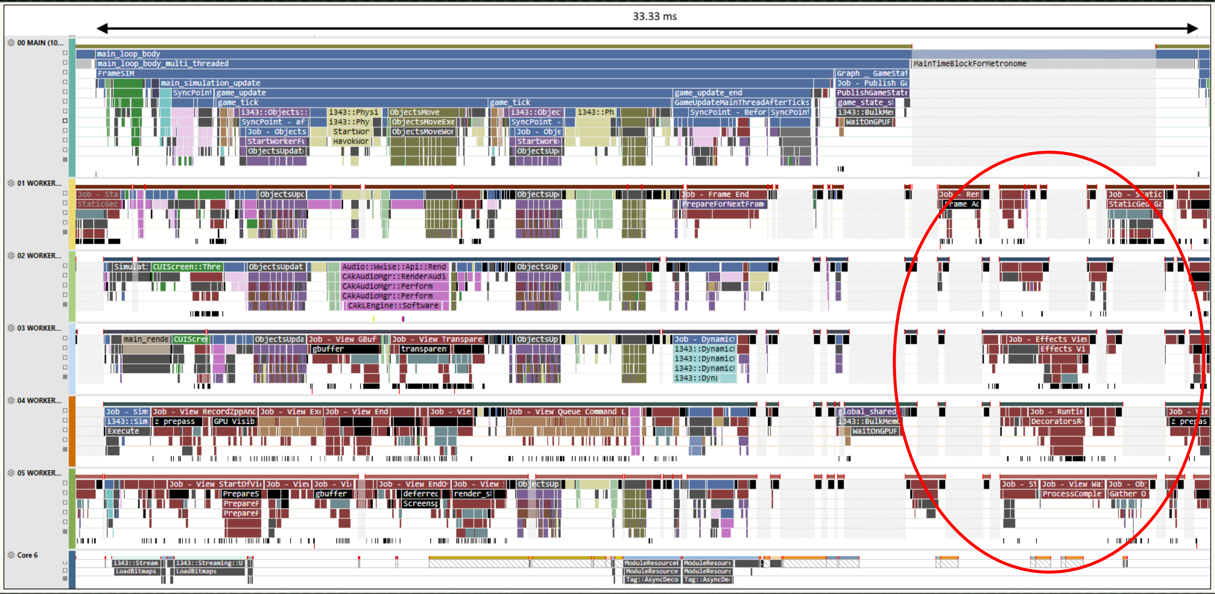
Note how we have all these rendering jobs going on during simulation as well, these are rendering jobs from the job graph of our renderer, rendering the previous frame as we simulate this frame.

One Frame in Halo Infinite – Xbox One



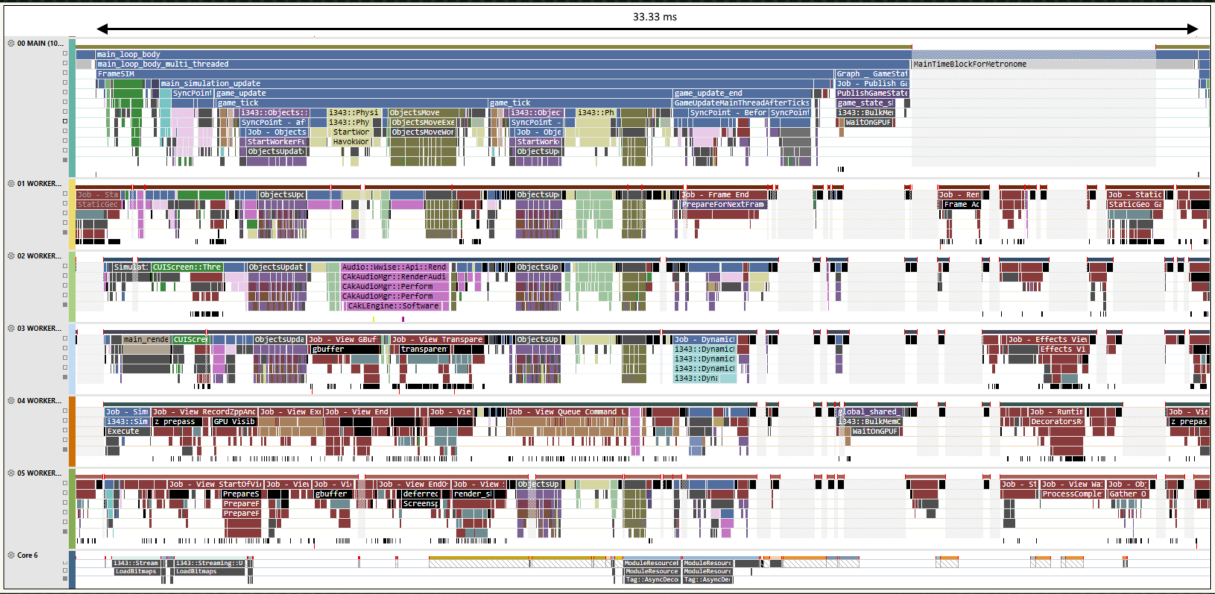
We also see audio rendering here in hot pink, note how this is scheduled cooperatively like any other job without switching to other threads.

One Frame in Halo Infinite – Xbox One



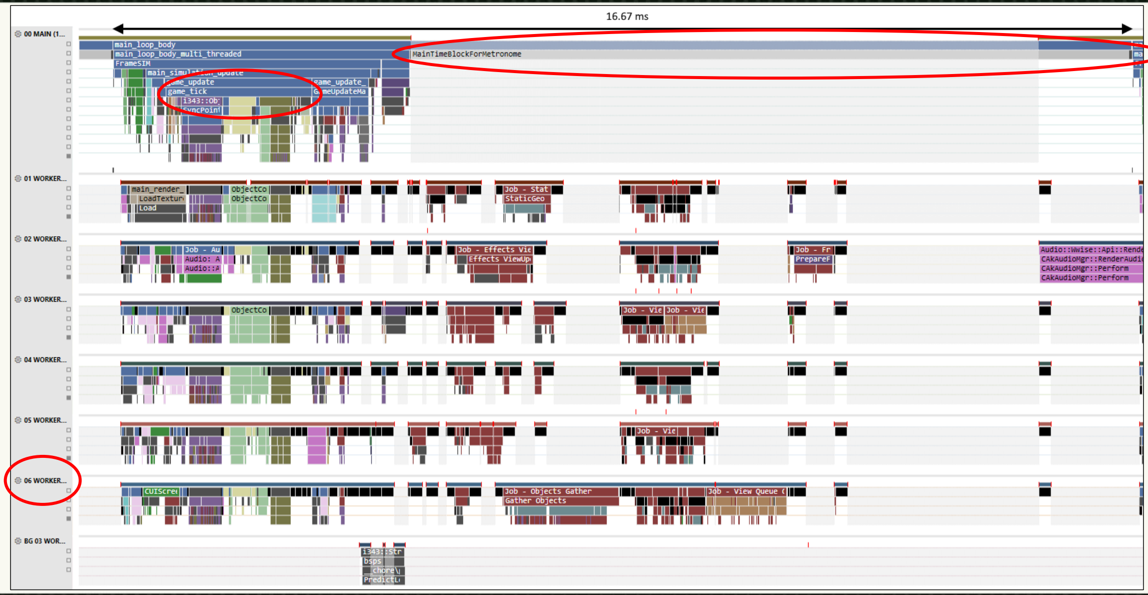
And here’s where the renderer starts execution of rendering after the frame has been published. I won’t be breaking down the layout of the rendering job graph in this talk, but here you can see the beginning of that job graph running without any simulation workload, and you can see that there’s pretty good parallelism in our renderer.

One Frame in Halo Infinite – Xbox One



Overall, we achieve pretty good utilization when simulation and rendering are going at the same time (that is the hardware is soaked), like on the left portion of this screenshot.

One Frame in Halo Infinite – Xbox Series X



One Xbox Series X we support a performance mode at 120 Hz but reduced resolution. By default, though, we run at 60 fps targeting 4K resolution (screen-permitting).

We recognize a lot of our familiar workloads from the Xbox One case here, looking at the frame, however there are a couple of things I'd like to point out:

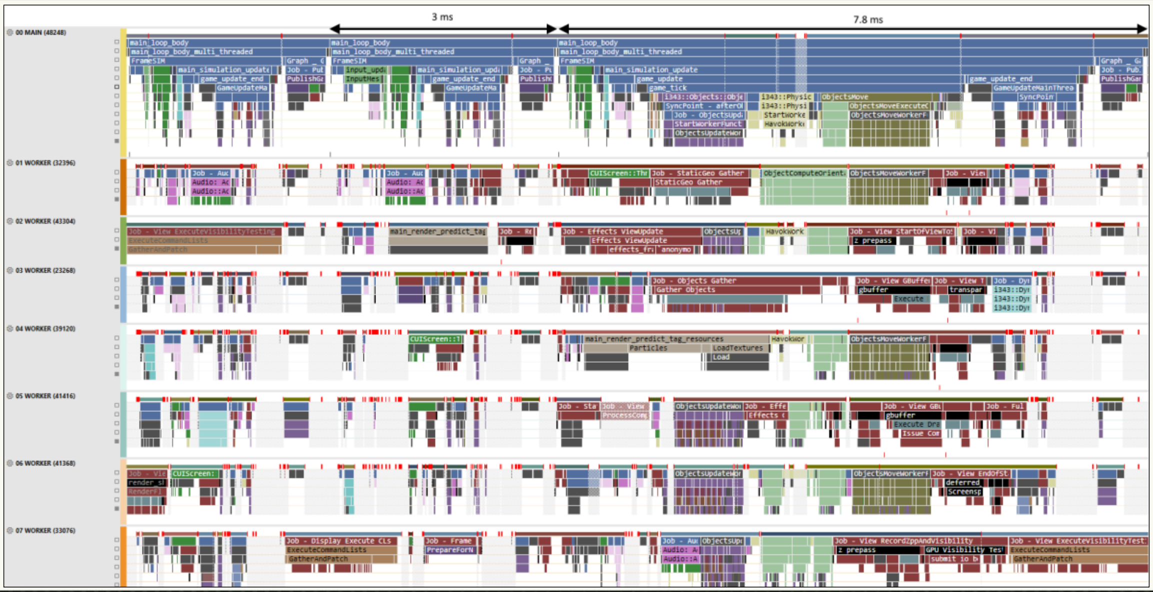
1) We have a new real time worker that also collaborates with rendering and simulation just like all the others. This is because CPU 6 is actually exclusive to the game on Xbox Series consoles.

2) Also note how we have a single game tick or fixed update here. That is of course by design. We are targeting 60 fps in this screenshot and thus we will perform one fixed update each frame.

3) In this screenshot, we are done with the simulation _way_ ahead of our metronome beat, just because of how fast the CPUs are in the new console, and because we execute so efficiently on consoles with collaborative scheduling and very few context switches. We do scale up certain CPU workloads on Xbox Series X, like texture/geo

streaming, a lot of rendering workloads drawing more objects more often, and more simulation eye-candy stuff like higher fidelity animation, critters and details throughout the world. This slack is very important for us to achieve 120 fps simulation on the console when in performance mode, but it also allows us to ingest occasional spikes more easily, without “missing a beat”, or dropping a frame.

One Frame in Halo Infinite – unlocked framerate PC

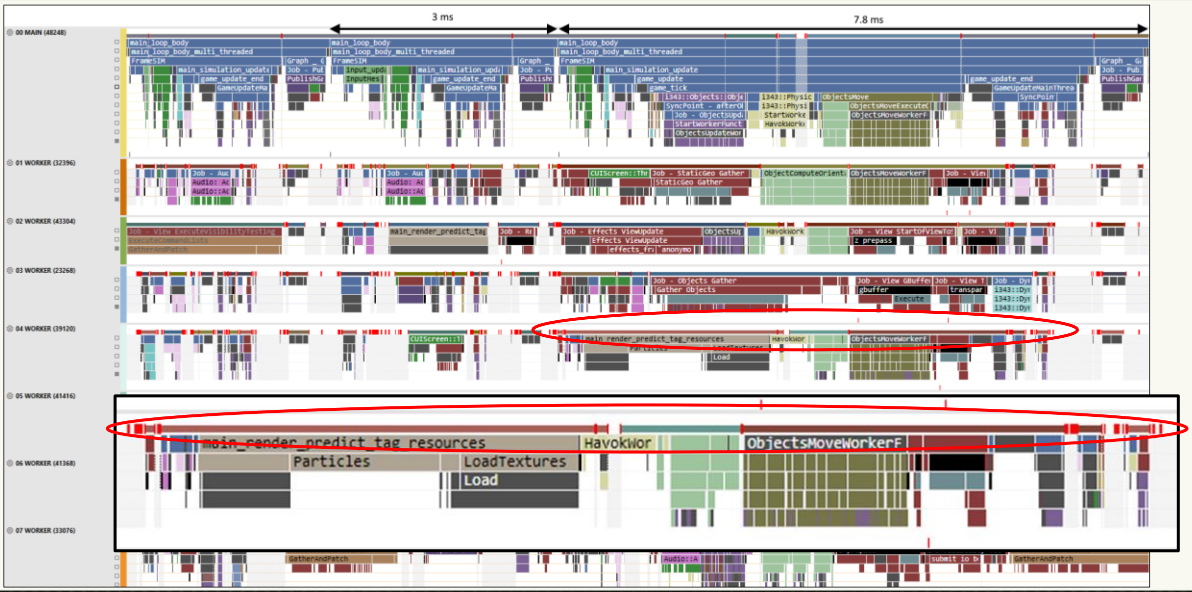


March 21-25, 2022 | San Francisco, CA #GDC22

Let’s now look at a PC frame. The way a PC frame looks depends wildly on how the player configured the game to run, they can select arbitrary framerates, enable/disable vsync, and choose to run at “unlocked framerate”. Unlocked framerate means that basically we want no cap on the maximum fps, and the game always runs as fast as it can without ever waiting for a metronome before starting the next frame.

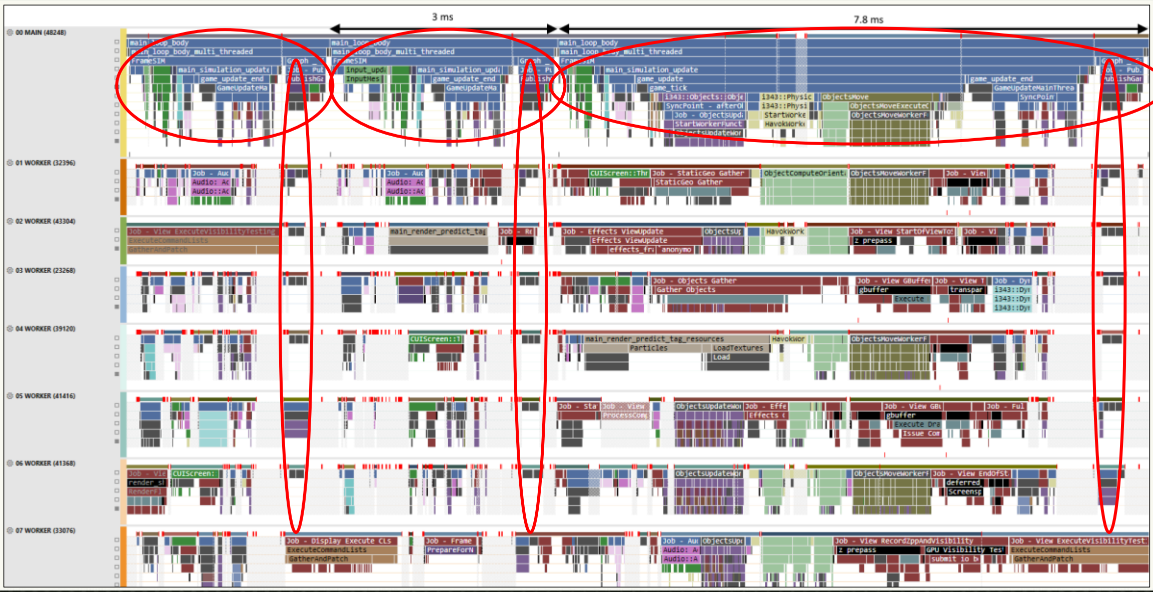
For input latency consistency, this is not often how most players will want to run, but let’s look at what the engine does here as an exercise.

One Frame in Halo Infinite – unlocked framerate PC



Note first of all that all threads seem to be changing CPUs pretty consistently, represented by the thin bars highlighted here, that’s normal as we don’t affinitize any thread on PC.

One Frame in Halo Infinite – unlocked framerate PC



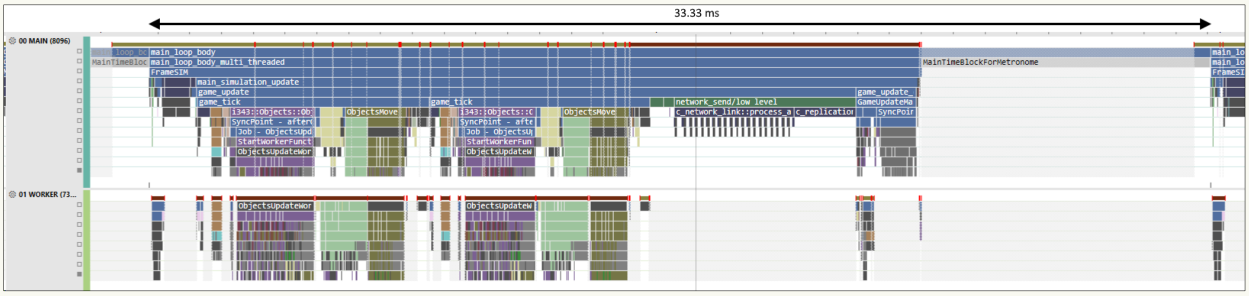
1) We can see that we have 2 frames at the beginning of this screenshot where we do _no_ fixed update (or game tick). That’s because we haven’t accumulated enough delay to perform another 16.67 ms worth of fixed update time and the game is running at >> 60 fps. Even without a fixed update, we will use our tweening system to interpolate between the last two fixed update and show smooth motion on screen as we publish the results at the end of these frames.

2) When we have accumulated enough time, see how a frame with a fixed update actually looks. We recognize our usual workloads in there.

3) Note one important difference on PC: because we don’t have DMA hardware available to the game to efficiently perform the bulk copy of the game state to the renderer, we actually use a parallel memcpy strategy on PC for maximum memory bandwidth utilization.

One Frame in Halo Infinite – 24 player server

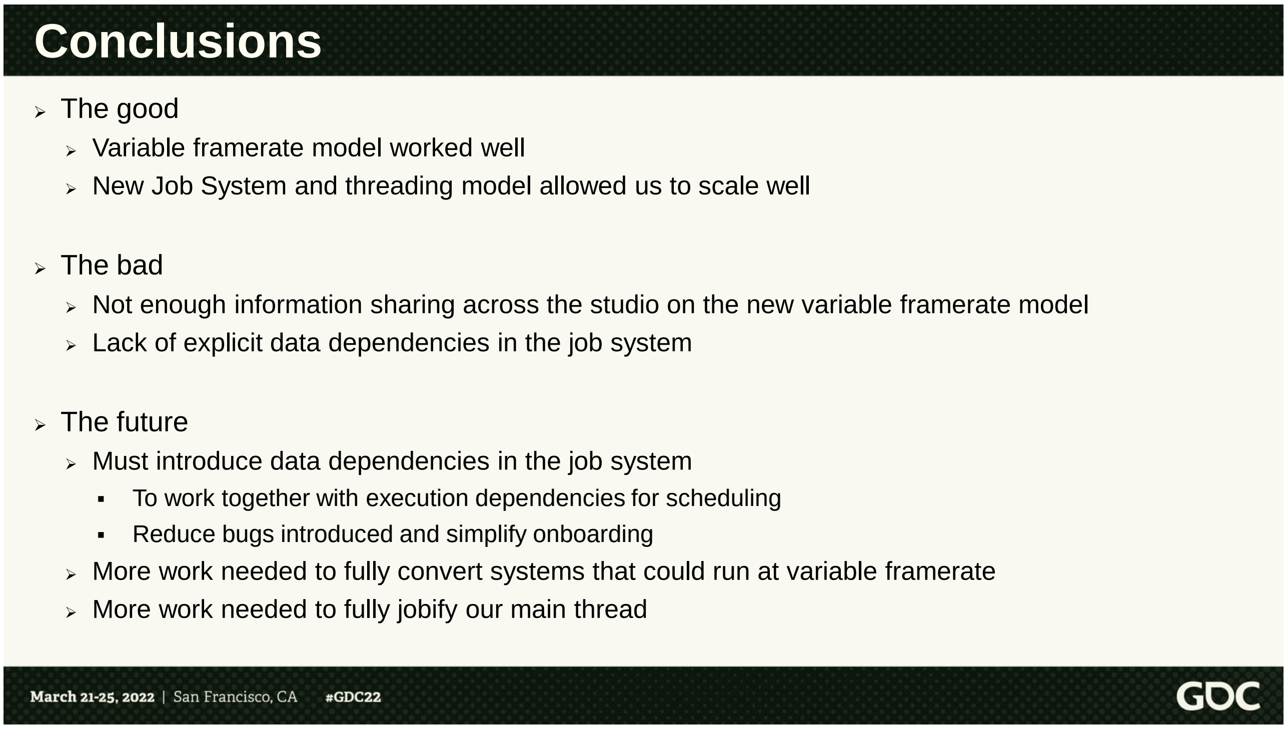
- Execution on cloud dedicated servers is the same as any other PC
  - Arena 4v4 servers are 60 Hz
  - Big Team Battle 12 v 12 servers are 30 Hz



For Halo Infinite our server VMs have different “framerate“ targets depending on the experience, for instance Arena 4v4 servers run at 60 Hz (which is effectively the network send rate), while “big team battle” 24 player servers run at 30 Hz since that is a more social mode.

Here’s a screenshot again from PIX for windows of a big team battle server “frame” (2 core server VM).

There are some difference compared to a regular PC execution since there is no rendering, audio or simulation state publish, and obviously additional networking workloads, but most of the workloads should look familiar.



Conclusions

- The good
  - Variable framerate model worked well
  - New Job System and threading model allowed us to scale well
- The bad
  - Not enough information sharing across the studio on the new variable framerate model
  - Lack of explicit data dependencies in the job system
- The future
  - Must introduce data dependencies in the job system
    - To work together with execution dependencies for scheduling
    - Reduce bugs introduced and simplify onboarding
  - More work needed to fully convert systems that could run at variable framerate
  - More work needed to fully jobify our main thread

In conclusion, we did accomplish our goals: we feel like our variable framerate model worked pretty well, and we also feel like our Job system and threading model changes were a big win, we fully removed our render thread and achieved good execution efficiency across hardware targets.

It wasn't without some pain: we should have done more evangelization on our variable framerate model to avoid confusion from owners of various systems… And one key component missing from our job system scheduling approach is data dependencies, in other words what data is each job accessing, and is there potential of race condition between these jobs using the same data. Lacking this tool created a good number of race condition bugs that we had to hunt down and resolve.

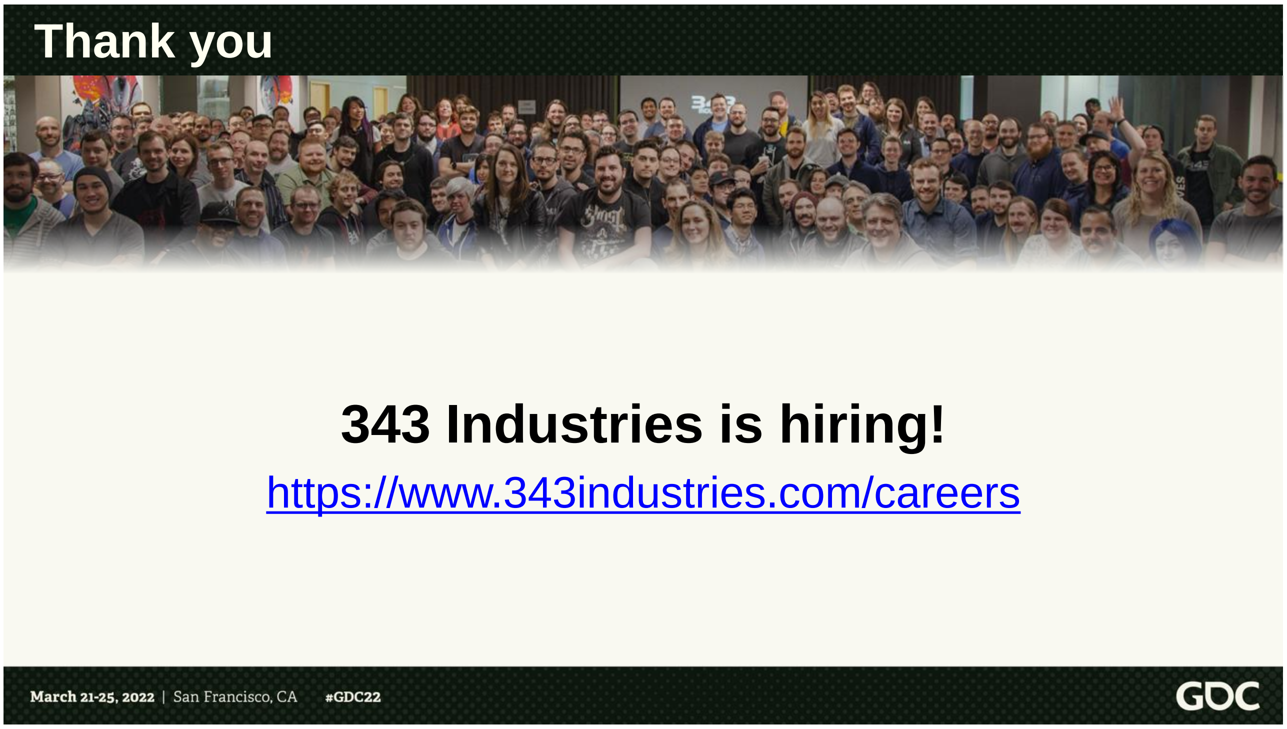
In terms of future work, we really want to address data dependencies by adding them as new functionality in the job system, our vision is that we'll introduce data dependencies to work together with execution dependencies for scheduling, which should reduce bugs created during development and further simplify onboarding.

On top of that, more work is needed to finish conversion of engine systems that could run at variable framerate. And finally, just like we did with our render thread, we want to get rid of our main thread by converting it to just jobs, and remove the concept of Job System “sync points”

Thank you

Jon Adams
Zully Barrientos
David Berger
Nick Bygrave
Gabe Castro
Danny Chen
Simon Craddick
Alexandre David
Reza Elghazi
Seamus Epp
Jason Fleming
Bill Fowler
Nicholas Frechette
Jun Fu
David Garza
Andrew Gleeson
Paul Haban
Steve Heijster
Chris Herrman
Tim Hinds
Tom Holmes
Chris Howard
Dean Johnson
Gohar Kanungo
Hironobu Kikuchi
Robert Kingsley
Matthew Koch
Jon Koelzer
Patrick Laukaitis
Nicholas LaCroix
Stephane LeBrun
Gareth Lough
Jeff MacDermot
Logan Marshall-Medlock
Josh Mattoon
Jon Olson
David Pashute
Tyler Staples
Adam Sturge
Chase Thompson
Richard Watson
Dan White
Ivy Wu

Here are some people from the Halo Infinite team I want to thank because they were either a big part of making these changes happen, or because they had to deal with me chasing them down to do optimization work. And of course, all of 343 Industries!



Thank you

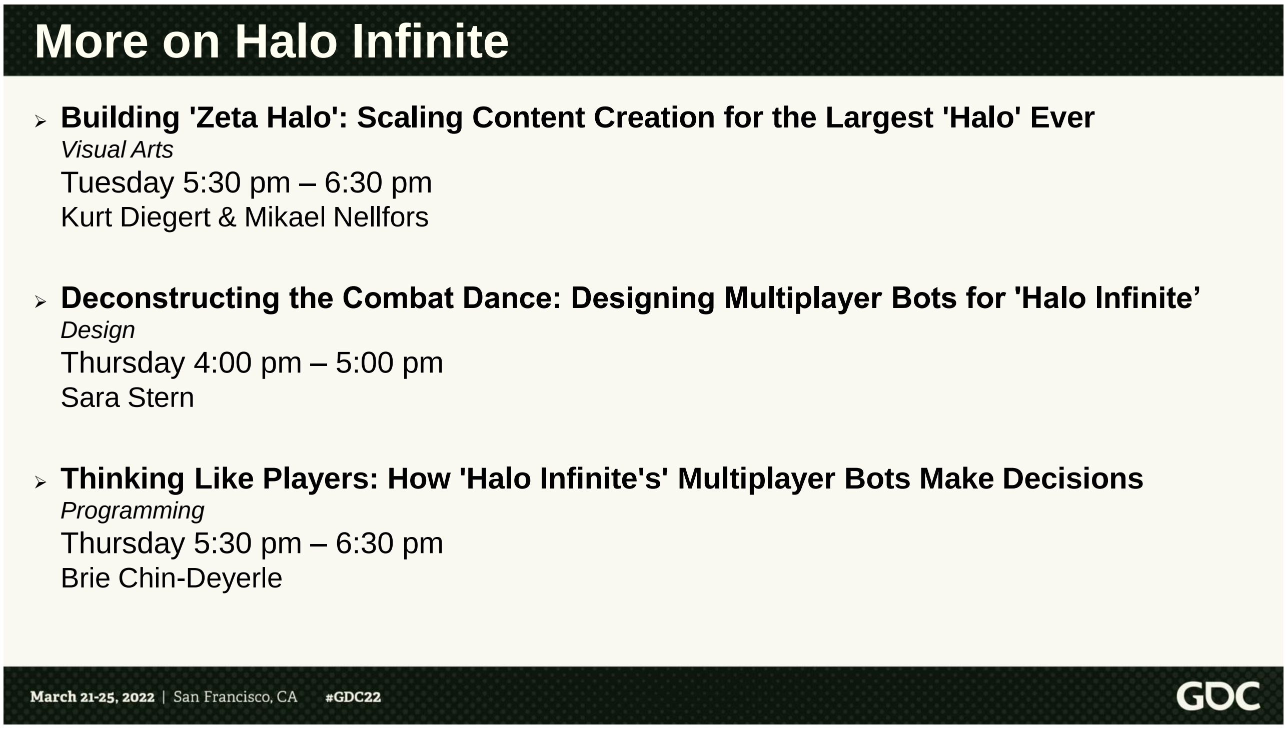
343 Industries is hiring!

https://www.343industries.com/careers

March 21-25, 2022 | San Francisco, CA #GDC22

GDC

343 industries is hiring! Check out our website if you want to come be part of whatever’s next in Halo Infinite!



More on Halo Infinite

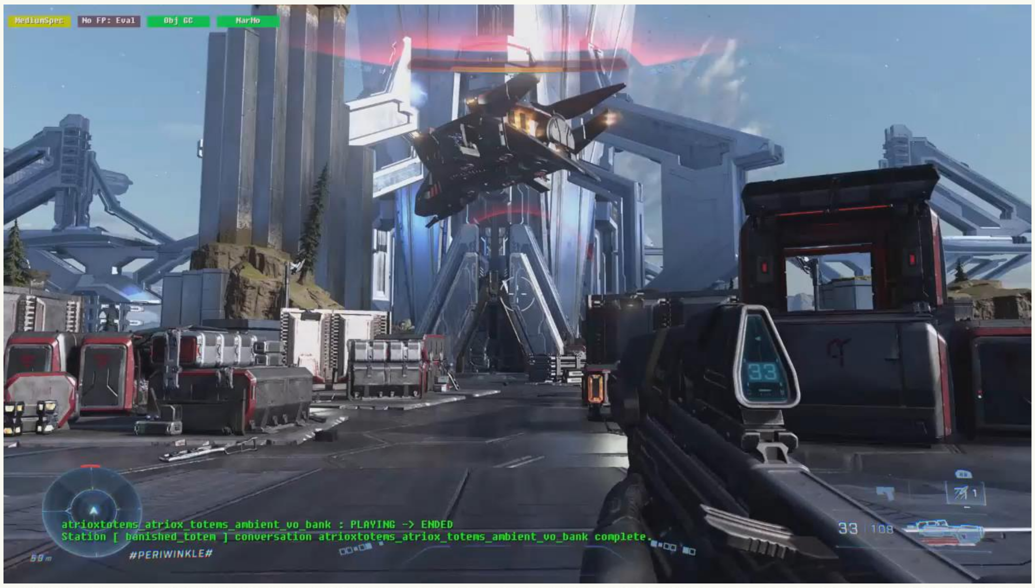
- **Building 'Zeta Halo': Scaling Content Creation for the Largest 'Halo' Ever**
  Visual Arts
  Tuesday 5:30 pm – 6:30 pm
  Kurt Diegert & Mikael Nellfors

- **Deconstructing the Combat Dance: Designing Multiplayer Bots for 'Halo Infinite'**
  Design
  Thursday 4:00 pm – 5:00 pm
  Sara Stern

- **Thinking Like Players: How 'Halo Infinite's' Multiplayer Bots Make Decisions**
  Programming
  Thursday 5:30 pm – 6:30 pm
  Brie Chin-Deyerle

If you want to learn more about Halo Infinite and Halo Infinite tech, here are some more sessions we have going on this year at GDC!

Q&A



```
atrioxtotems_atriox_totems_ambient_vo_bank : PLAYING -> ENDED
Station [ banished_totem ] conversation atrioxtotems_atriox_totems_ambient_vo_bank complete.
```

And now let’s do some Q&A. I’m going to let this video reel run as I take some questions so we can enjoy some bloopers!

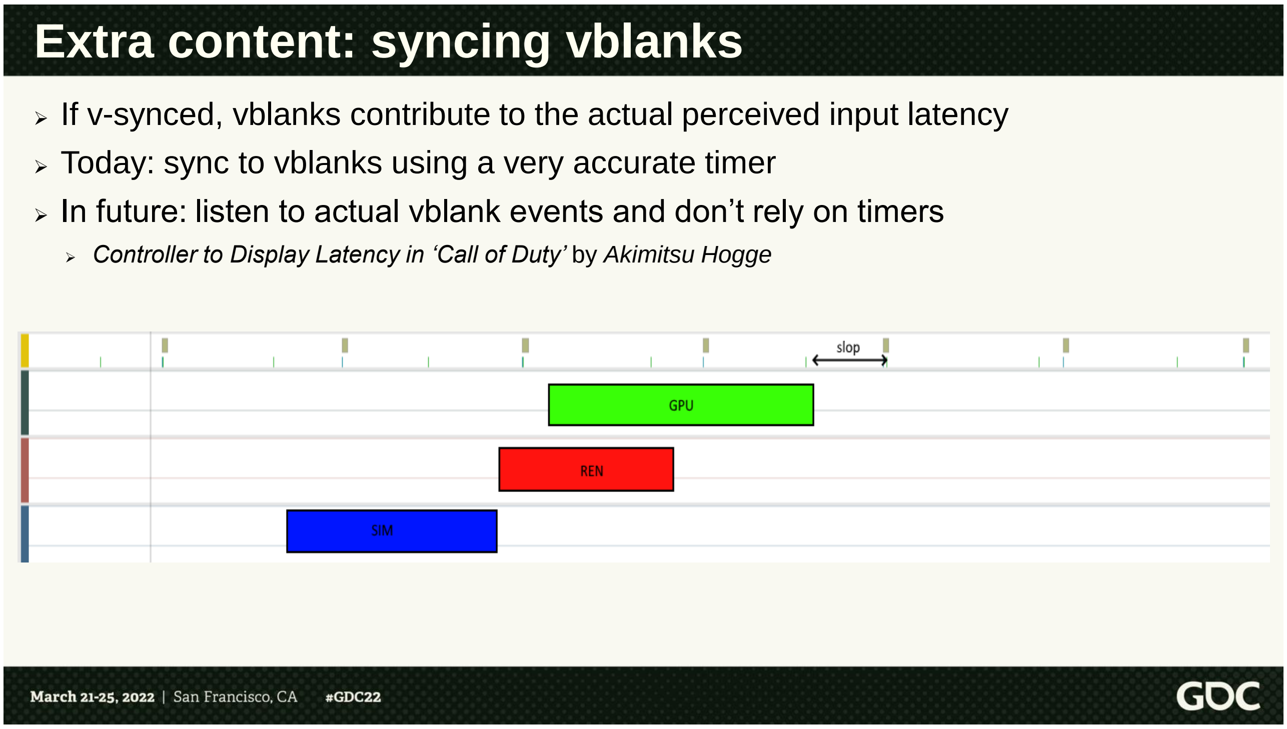
March 21-25, 2022
San Francisco, CA

One Frame in ‘Halo Infinite’

Extra Content

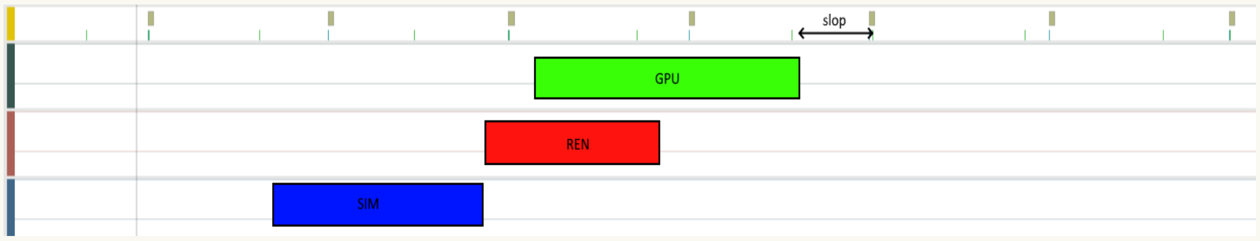
#GDC22

Extra content



Extra content: syncing vblanks

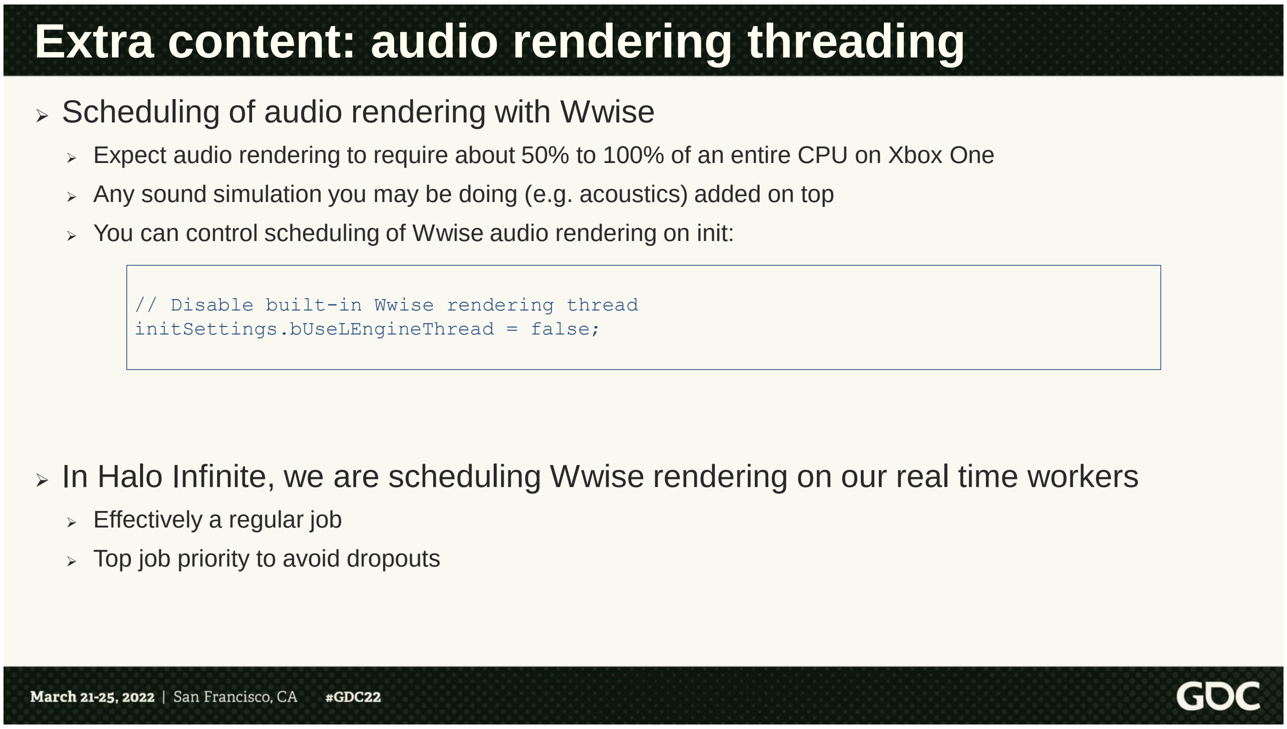
- If v-synced, vblanks contribute to the actual perceived input latency
- Today: sync to vblanks using a very accurate timer
- In future: listen to actual vblank events and don't rely on timers
  - *Controller to Display Latency in 'Call of Duty'* by *Akimitsu Hogge*



When vsynced (which is also how all our console targets will display the image on screen) we also must take into account the wait for the vblank when considering total input to display latency.

For Infinite, we are using a very high accuracy timer to synchronize the beginning of a frame's simulation CPU workload to the same rhythm of the display vblanks. This approach works reasonably well, however it can be improved by accurately measuring the available slop (as in the slide above) and taking that into account when deciding when to start the simulation workload to minimize the latency while still keeping some headroom to absorb small spikes without missing a vblank. We plan on investing in this area soon.

There's a very good GDC 2019 talk about this specific subject by Akimitsu Hogge I left on the slide.



Extra content: audio rendering threading

- Scheduling of audio rendering with Wwise
  - Expect audio rendering to require about 50% to 100% of an entire CPU on Xbox One
  - Any sound simulation you may be doing (e.g. acoustics) added on top
  - You can control scheduling of Wwise audio rendering on init:

```
// Disable built-in Wwise rendering thread
initSettings.bUseLEngineThread = false;
```

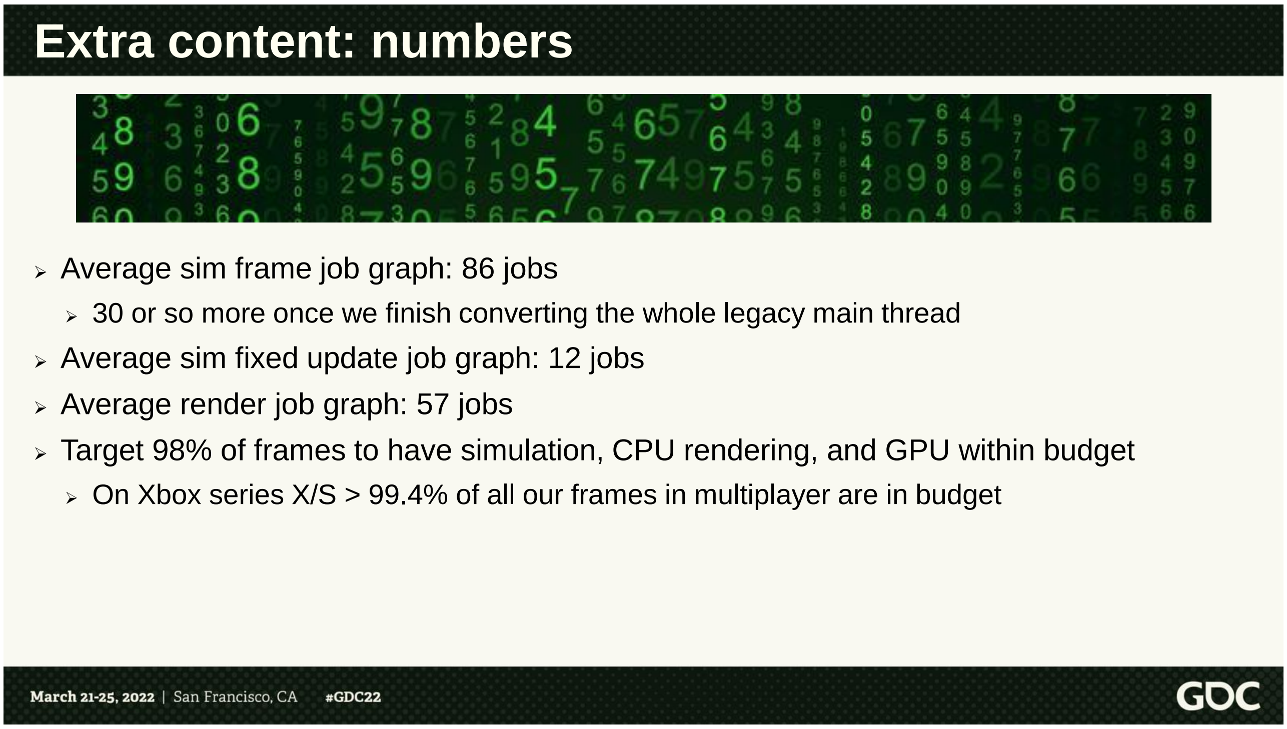
- In Halo Infinite, we are scheduling Wwise rendering on our real time workers
  - Effectively a regular job
  - Top job priority to avoid dropouts

One interesting digression into Wwise.

On Xbox One, audio rendering is kind-of expected to require about 50 to 100% of an entire CPU for rendering consistently and avoiding drop-outs. Any sound simulation that we might have (like acoustics/occlusion) is added on top.

This puts audio rendering as one of our top workloads together with actual graphics rendering and the core game simulations.

For Infinite, since we schedule all our real-time work cooperatively using jobs, we are configuring Wwise to allow us to schedule our own audio rendering, and we do this using this specific flag on init. I’d recommend this configuration for any game using cooperative scheduling and Wwise, or else the Wwise rendering thread will cause heavy interference and reduce efficiency.



Extra content: numbers

- Average sim frame job graph: 86 jobs
  - 30 or so more once we finish converting the whole legacy main thread
- Average sim fixed update job graph: 12 jobs
- Average render job graph: 57 jobs
- Target 98% of frames to have simulation, CPU rendering, and GPU within budget
  - On Xbox series X/S > 99.4% of all our frames in multiplayer are in budget

I wanted to share some numbers for the curious…

An average frame simulation job graph for us contained 86 jobs every frame, and considering that we haven't finished converting a lot of legacy workloads on the main thread, we are probably missing 30 or so for a full jobification.

An average fixed update job graph contained 12 jobs. Jobs are pretty chunky here, for instance the workload executing the parallel object update across worker threads for all objects is a single job.

An average renderer job graph is 57 jobs, this includes all CPU rendering workloads.

For performance, we set ourselves a target of 98% of all frames being within our SIM/REN/GPU budgets, we hit this largely across platforms and experiences in our automation. Gathering clean playtest data was challenging for us on PC through, as there are many knobs the player can toy with to "shoot themselves in the foot".

On Console, and specifically Series X/S, more than 99.4% of all our frames across multiplayer experiences are within budget, we are pretty happy with how smooth the experience is.