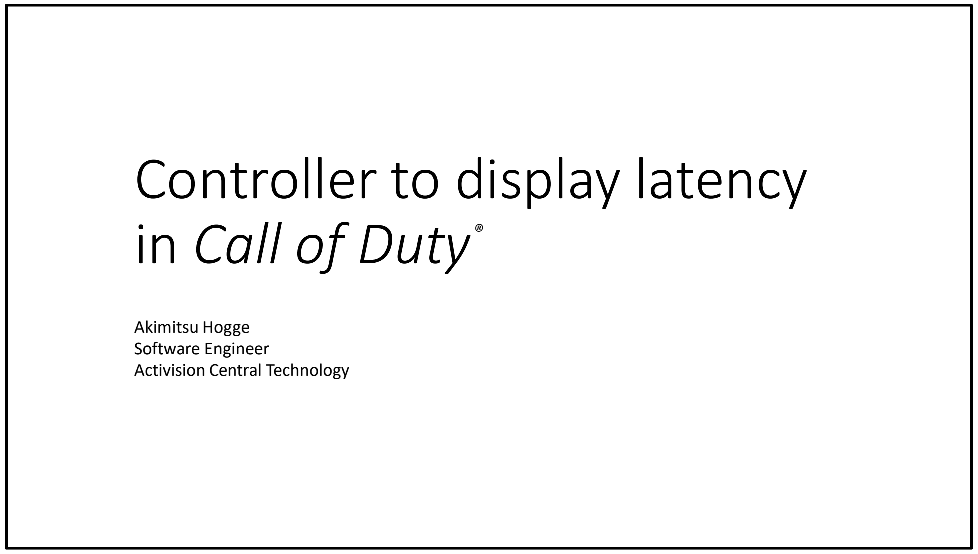


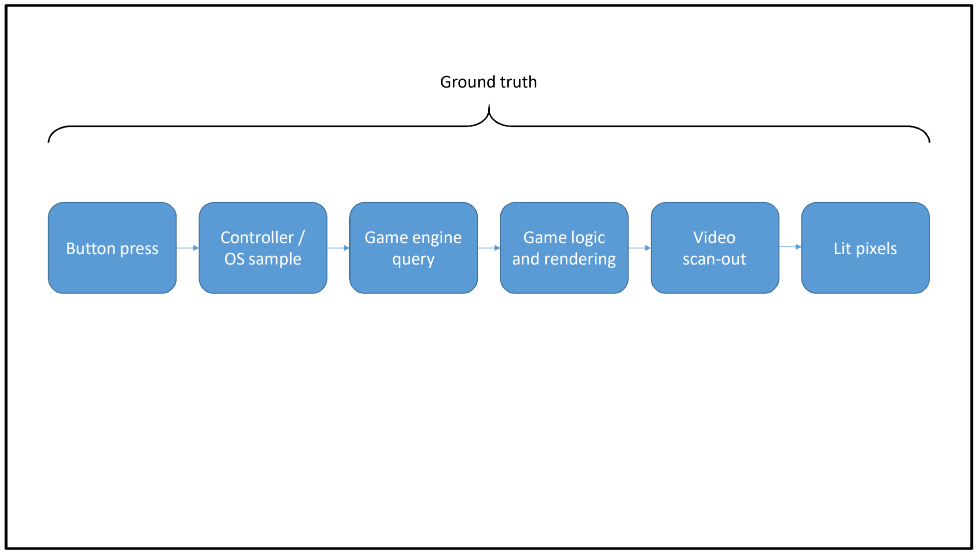
Controller to display latency in *Call of Duty*®

Akimitsu Hogge
Software Engineer
Activision Central Technology

My name is Aki and I'm a programmer for Activision Central Tech. I work with a small team in Portland Maine where we tackle performance problems in Call of Duty titles.

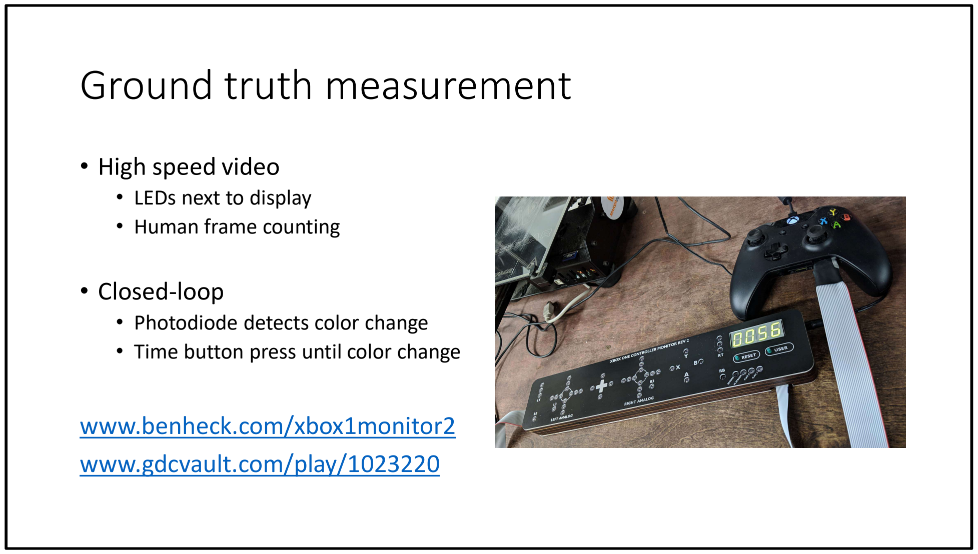
Today we're going to be discussing controller to display latency: that's the shortest duration between a player pressing a button and the player seeing the result of the press on screen. I'm only going to be discussing the game client, and won't be covering anything related to networking or the server.

We're going to go over a dynamic throttling feature that was added to recent Call of Duty games to control the tradeoffs that impact input latency, with the ultimate goal of reducing controller to display latency.
But first we'll go over how we measure latency, and then we'll dive into specific aspects of the engine that had to be factored into the throttle. Finally I'll discuss the implementation of the throttle and its impact on a recent game.



If you split the controller to display latency duration out into steps:
First the player presses a button, and the controller samples the button and the console samples the controller at some frequency. At some point in the game loop the game queries the OS for its controller state, and uses the knowledge of the button press for game logic and rendering. The final rendered image then gets scanned out to the display, and the display lights up pixels for the player to see.

Let’s call this entire duration ‘ground truth latency’



Ground truth measurement

- High speed video
  - LEDs next to display
  - Human frame counting
- Closed-loop
  - Photodiode detects color change
  - Time button press until color change

www.benheck.com/xbox1monitor2
www.gdcvault.com/play/1023220

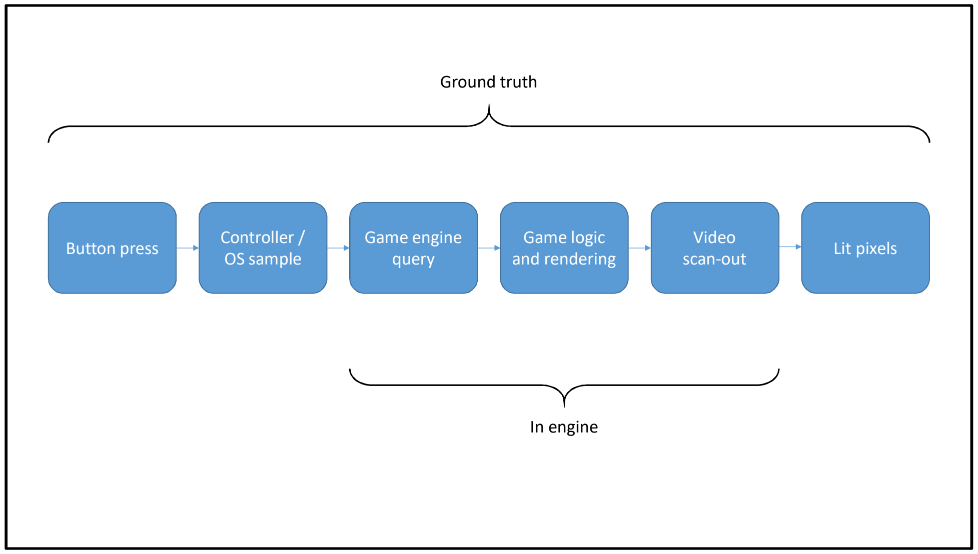
There are a couple ways of measuring ground-truth latency.

One technique is to a use a high speed video camera. You instrument the controller with a device that lights up LEDs whenever a button gets pressed, and record high-speed video of the LEDs next to the display showing the game. You can then calculate total latency by counting the number of video frames between an LED lighting up and an action appearing on screen.

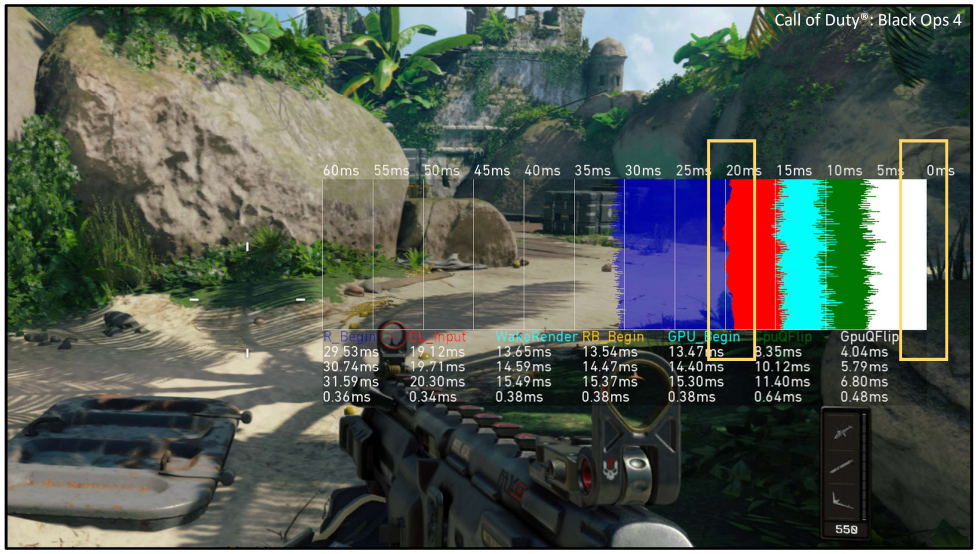
This process can be cumbersome because it relies on human frame-counting, and because individual frame latency measurements vary quite a bit, you have to take many samples for the average to converge. That's why it's preferable to use a closed-loop system.

For that you use device that detects button presses and also detects color changes with a photodiode. You modify the game to toggle the color of an on-screen square based on controller input, and the device then times the button press and times the color change, and subtracts the two to get total latency. This process can be more precise than frame counting if the photodiode is sampled at a faster rate than the video camera, and it's typically much faster to collect enough samples for the average to converge.

The Ben Heck Controller Monitor for Xbox One has both a bar of LEDs that can be used with a high-speed camera and photodiode that can be attached to a display (www.benheck.com/xbox1monitor2)
Also, please check out Benjamin Goyette's talk "Fighting Latency on Call of Duty Black Ops III" from GDC 2016 on how latency measurements have been used in a wider variety of applications. (www.gdcvault.com/play/1023220)



Now let's look at this subset of the latency duration between the game’s input query and the final image scan-out. This segment can be measured totally in software: let’s call it "in-engine latency"

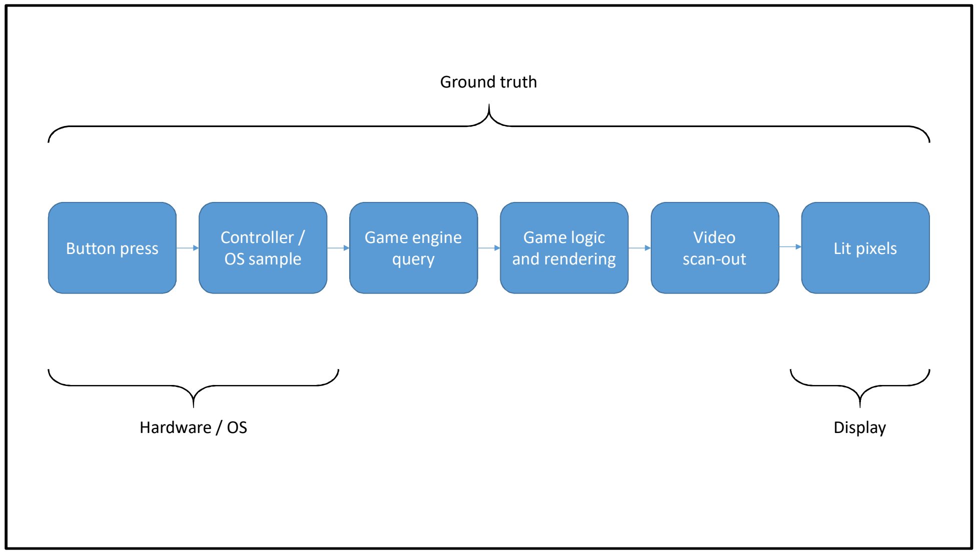


R_Begin	CL_Input	WakeRender	RB_Begin	GPU_Begin	CpuQFlip	GpuQFlip
29.53ms	19.12ms	13.65ms	13.54ms	13.47ms	8.35ms	4.04ms
30.74ms	19.71ms	14.59ms	14.47ms	14.40ms	10.12ms	5.79ms
31.59ms	20.30ms	15.49ms	15.37ms	15.30ms	11.40ms	6.80ms
0.36ms	0.34ms	0.38ms	0.38ms	0.38ms	0.64ms	0.48ms

The engine just has to timestamp the input sample and the start of the video scan-out along with frame indices, the correspond the two to get accurate per-frame in-engine latency measurements.

Here's a latency measurement overlay I added in Call of Duty Black Ops 4. Horizontally we're measuring milliseconds, with the right edge corresponding to the start of the scan-out. Each row of the graph represents a single frame, with frames scrolling up over time.

Each of the colored segments correspond to various engine events that contributed to that particular frame, with the left edge of the bright red bar corresponding to the frame's input sample. In this example, in-engine latency is sitting at around 20ms.



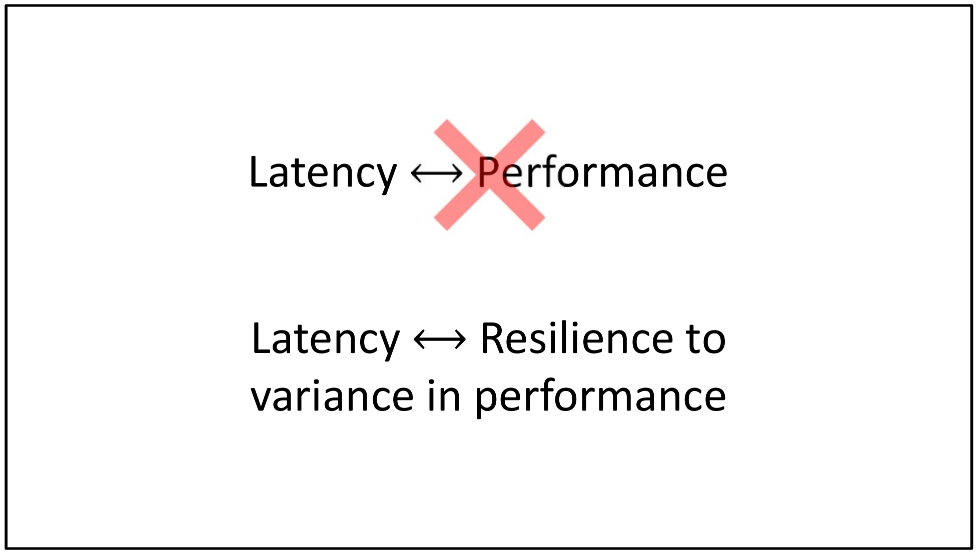
What about the segments that aren't being measured in software?
This first segment is handled by the controller and console hardware and software, and the last segment is handled by the display hardware.
The behavior of the game engine itself shouldn't have any impact on these segments, and given a stable hardware setup, their total latency contribution should average out to a constant.
We can test this by comparing in-engine measurements and ground-truth measurements.

Ground truth	In-engine	Δ
101.4 ms	57.3 ms	44.1 ms
64.7 ms	20.3 ms	44.4 ms

Call of Duty®: Black Ops 4

Here’s a couple of measurement of the game running under different latency parameters. The first column shows ground-truth measurement averages, and the second column shows in-engine measurement averages. The difference is about 44ms, which means that I’m getting around 44ms of latency contribution from factors outside of the game itself.

The fact that the difference is constant and independent from game behavior is really useful: it means we can rely purely on software measurements during development, checking ground-truth only for final confirmation. This significantly speeds up iteration time, and makes it easier to log and automate measurements. Moving forward, we’re only going to be focusing on how to reduce in-engine latency.



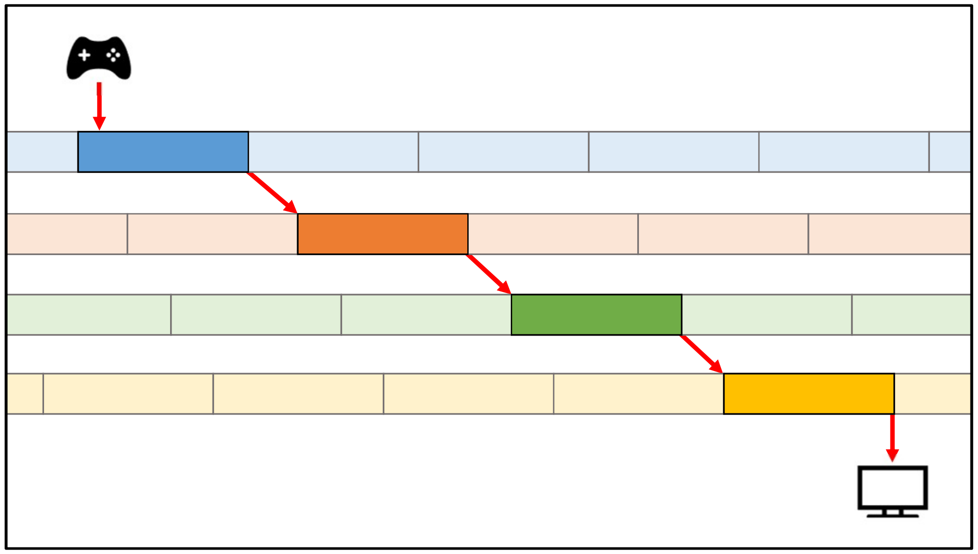
Latency ~~↔~~ Performance

Latency ↔ Resilience to variance in performance

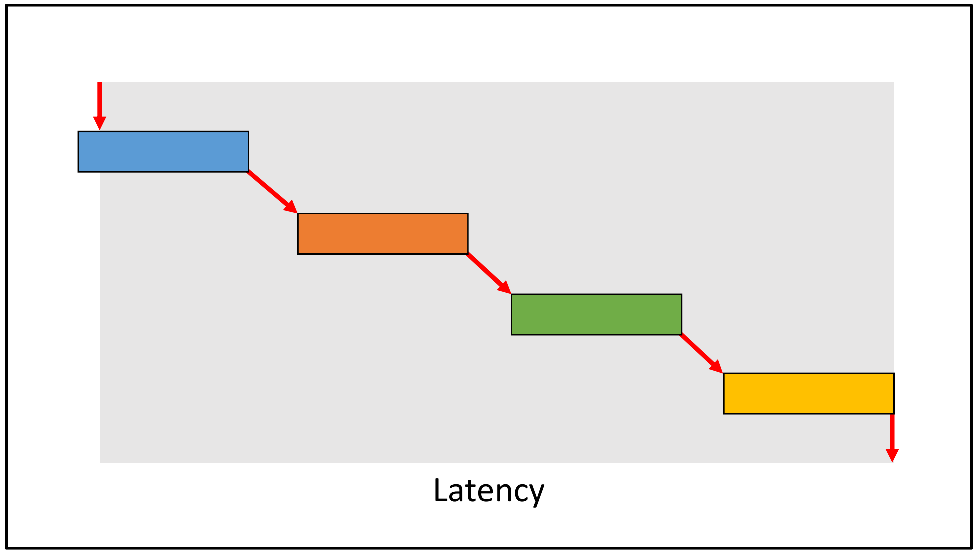
Running at 60 FPS has been a cornerstone of the Call of Duty franchise, and a lot of effort goes into creating as rich of a gaming experience as possible while still holding 60.

But there's an intuition that performance is a tradeoff with input latency. And that's true to a certain extent:
One of the most important ways we fit more content into the game is by making full use of the available hardware. That means directing as many CPU and GPU cycles as possible towards game content: more pixels, more triangles, more physics, and so on. Some of the optimizations we make to improve hardware utilization do introduce latency into the engine.

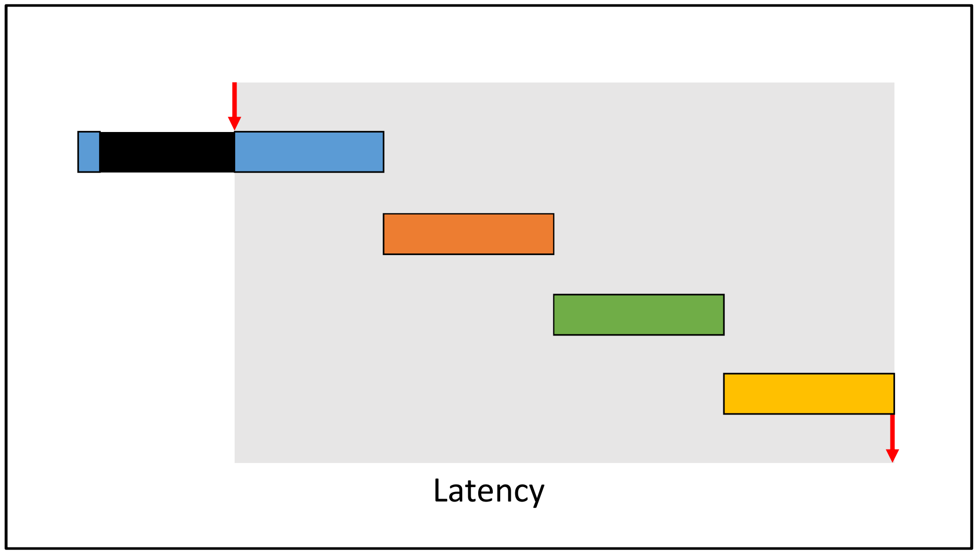
Our goal today is to claw back some of that latency without sacrificing the performance gains from those optimizations. I hope to show that the trade-off is not between latency and performance. Rather, that it's between latency and resilience to variance in performance.



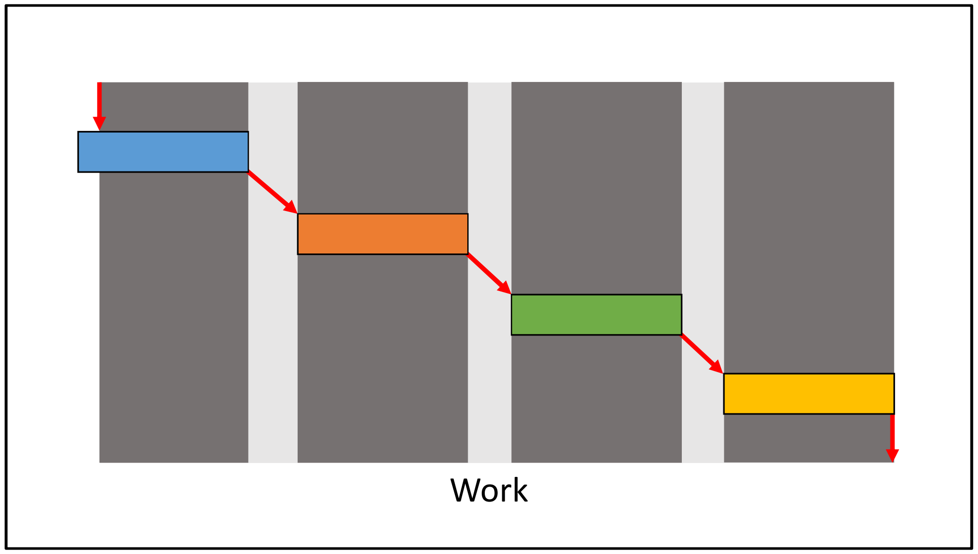
To achieve high hardware utilization, our games are very pipelined, with work being performed in parallel on multiple timelines. The best way to think around latency at a high level is to follow the flow of data down this pipeline. Controller input gets sampled near the beginning of the top timeline. The result gets processed and passed down to the next timeline in a chain of producers and consumers until a final image shows up on the display. The point of pipelining is that you have multiple frames in flight at once, increasing the chance that all CPU and GPU cycles are put to good use: so while work is being performed on this frame, another frame can start and be in flight as well.



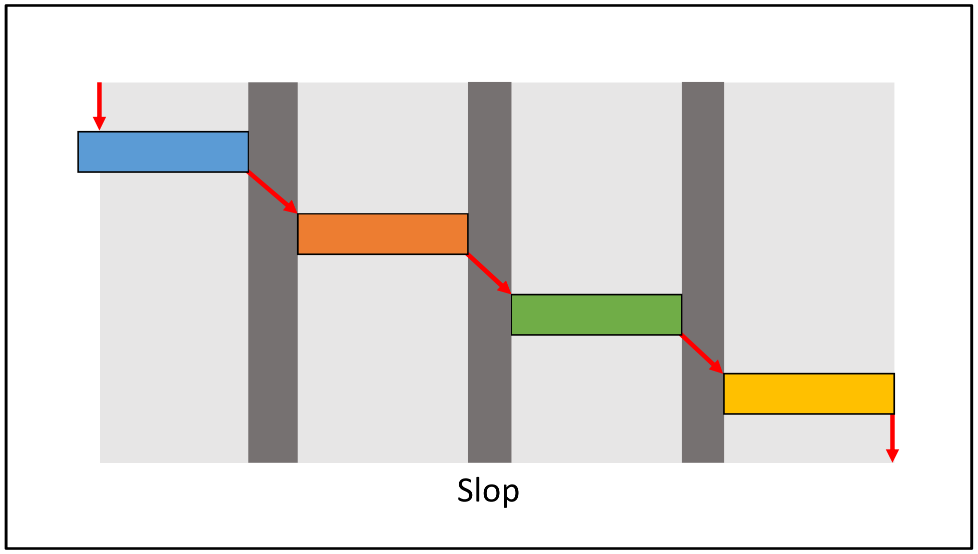
The duration from the input sample to the scan-out is the total in-engine latency for this particular frame.



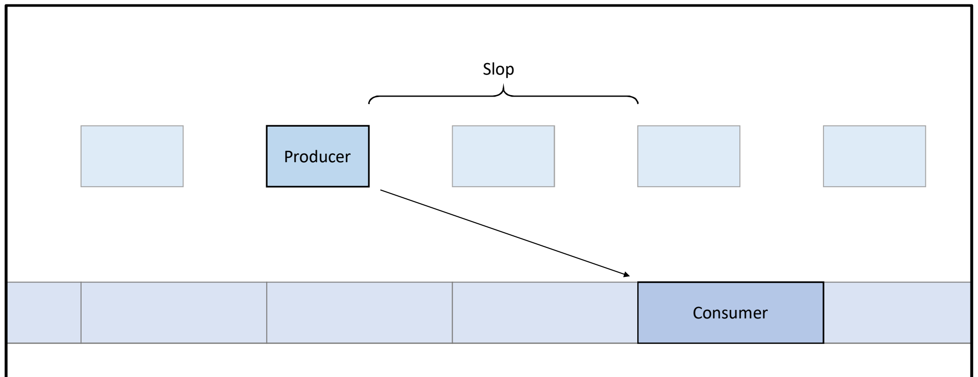
The strategy we’re going to be employing for latency reduction is to introduce a throttle right before the input sample. By adding a delay here, the input sample gets squeezed closer to the end of the frame.



To make thinking about this throttle easier, let's split the latency duration into two categories.
First is work. work is the time spent actively processing something for this specific frame: for example game logic or rendering. It's the time represented inside the colored boxes.



Everything else, let's call "slop".

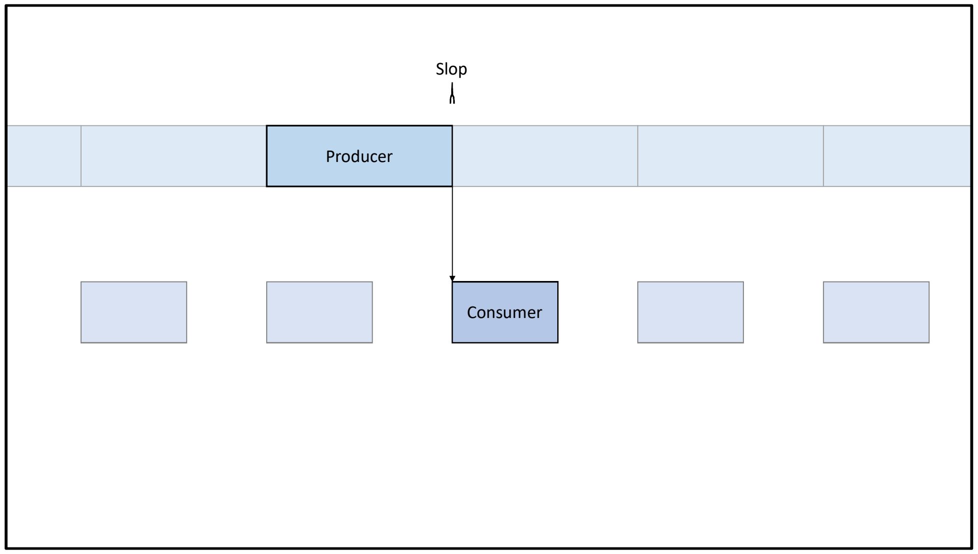


Slop appears when bound by a consumer

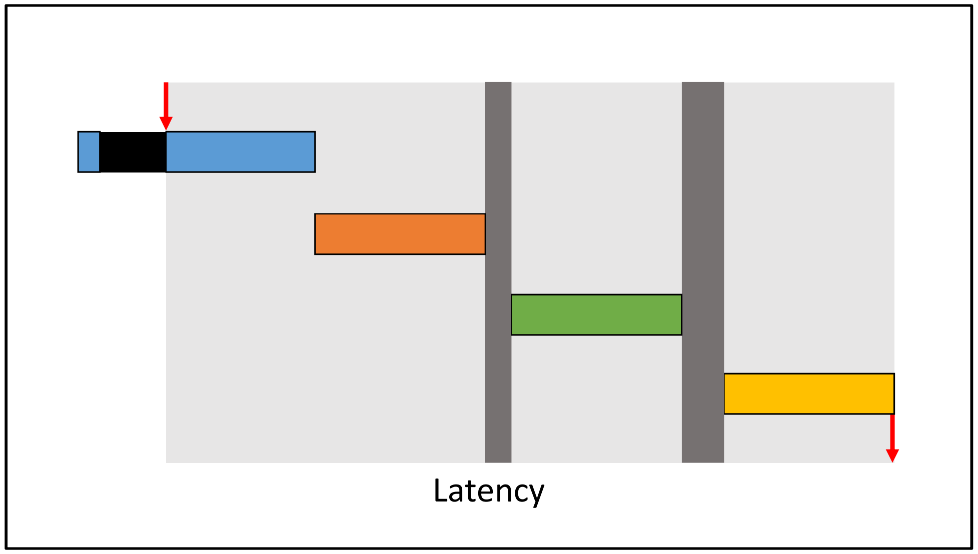
Slop is not necessarily idle time: it's usually segments where one timeline is working on a previous frame, or when one timeline is waiting for another timeline to release a shared resource.

If you look at a generic producer-consumer system, the producer performs a unit of work, passes the result on to the consumer, and then starts producing the next unit of work. If the consumer is consistently slower than the producer, the system becomes “consumer-bound”. The consumer’s timeline stays full, trying to keep up with the producer, but the producer is allowed to get as far ahead as possible.

That’s why slop exists: the producer is ahead of the consumer, so there’s a gap between when the producer finishes and when the consumer starts. How far ahead the producer can get depends on how much buffering is allowed between the producer and consumer, and exactly when they acquire and release access to the buffered data.

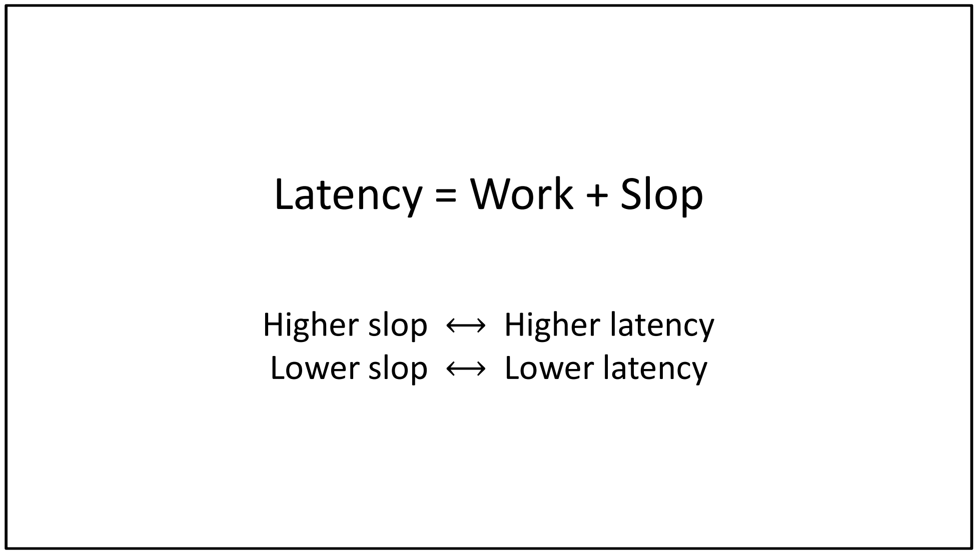


On the flipside, slop usually disappears when we're producer-bound. The consumer's timeline is freed up, so as soon as the producer finishes a frame, the consumer is allowed to start immediately and we don't get a gap.



When you delay the input sample, slop gets squeezed out starting with the first segment until it disappears, moving to the second segment, and so on.
On the other hand work gets shifted forward in time, while the total work duration across the frame ideally stays constant.

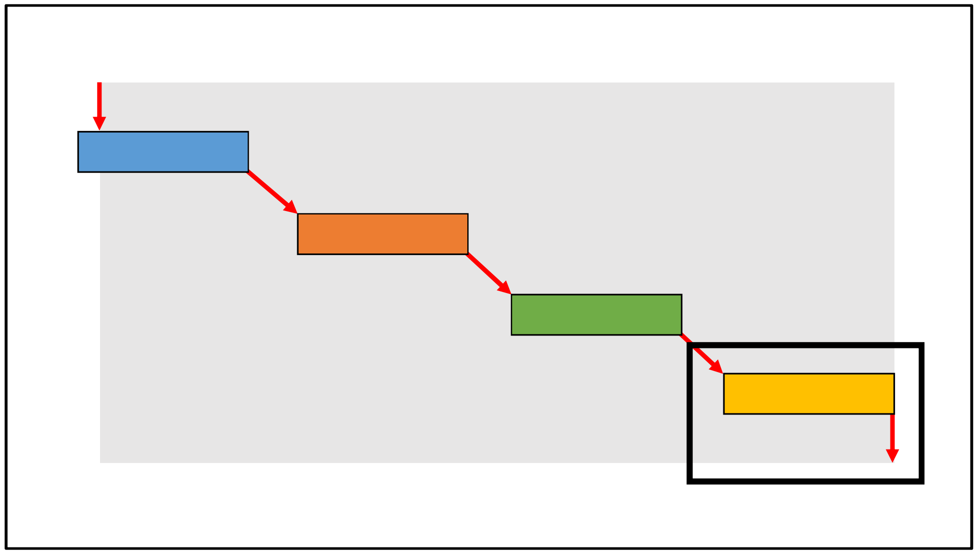
The key behavior of the throttle is that it squeezes slop while preserving work.



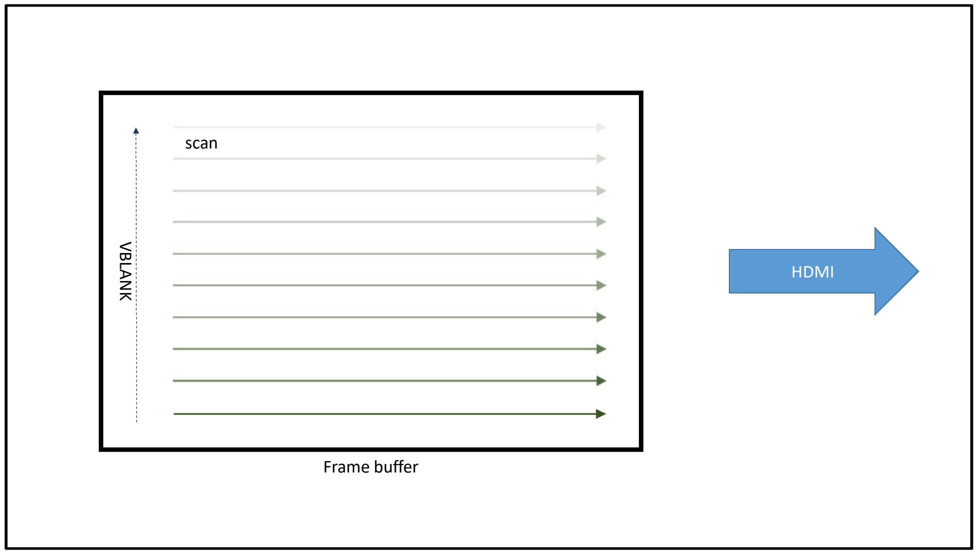
Latency = Work + Slop

Higher slop ⟷ Higher latency
Lower slop ⟷ Lower latency

We’ve split the latency duration up into work and slop.
Our strategy is to throttle the input sample to reduce slop, thereby reducing latency. To figure out how much to throttle, we’re going to examine all the sources of work and slop in the game.



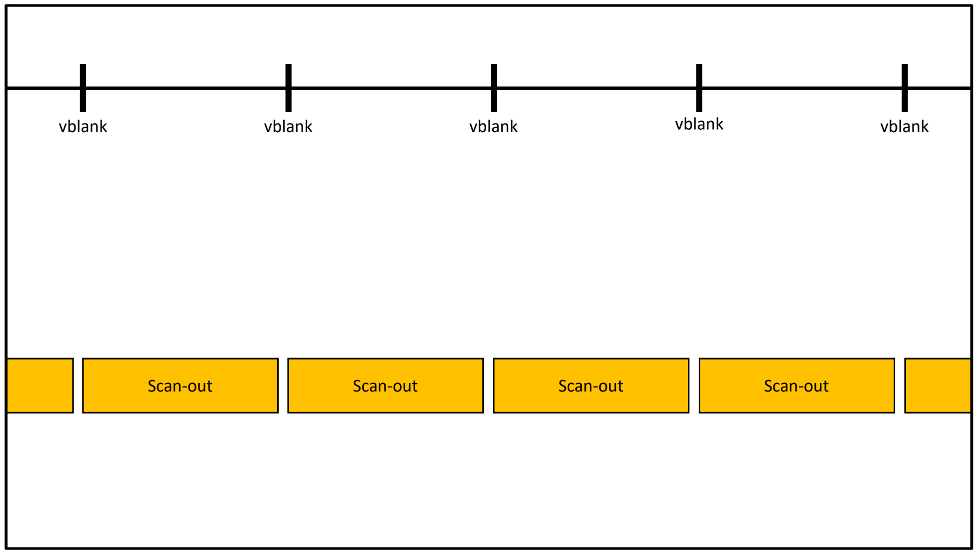
Let’s start by looking out what happens at the end of the latency duration.



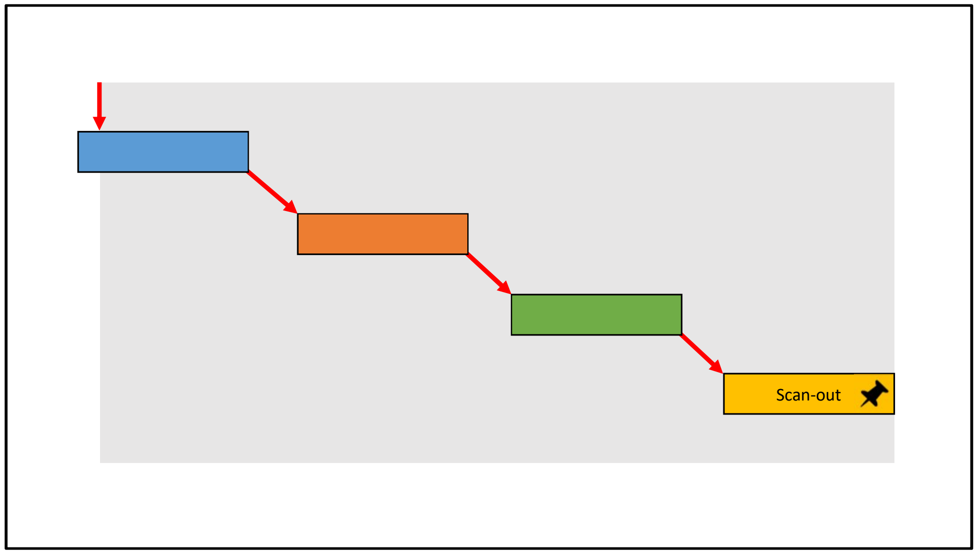
The game renders a final image into a piece of memory called the frame buffer. The video scan-out hardware then reads the framebuffer row by row, top to bottom, and transmits the pixel data across a cable to the display.

The scan-out transmits continuously at a rate matching the refresh rate of the display. Conventional displays today have a refresh rate of 60Hz, so the framebuffer is typically scanned-out at 60Hz as well, or once every ~16.6ms. Most of that 16.6ms period is spent actively transmitting visible pixel data, but there are brief pauses where the transmitted data doesn't correspond to visible pixels.

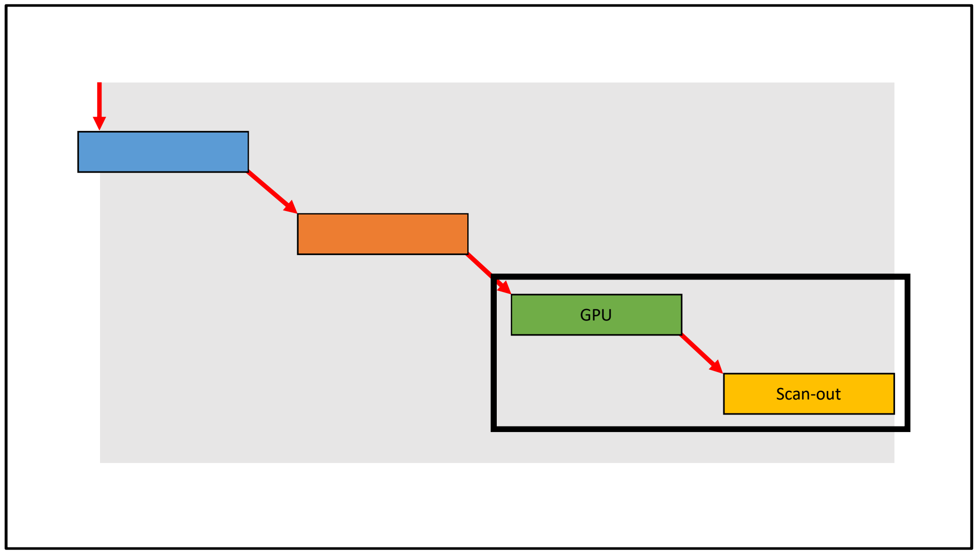
First, at the end of every row is a pause called the horizontal blank or HBLANK, and after the last row there's a pause called the vertical blank or VBLANK. Theses pauses were physically necessary on old CRT monitors, but they still exist today for legacy reasons and for transmitting metadata.



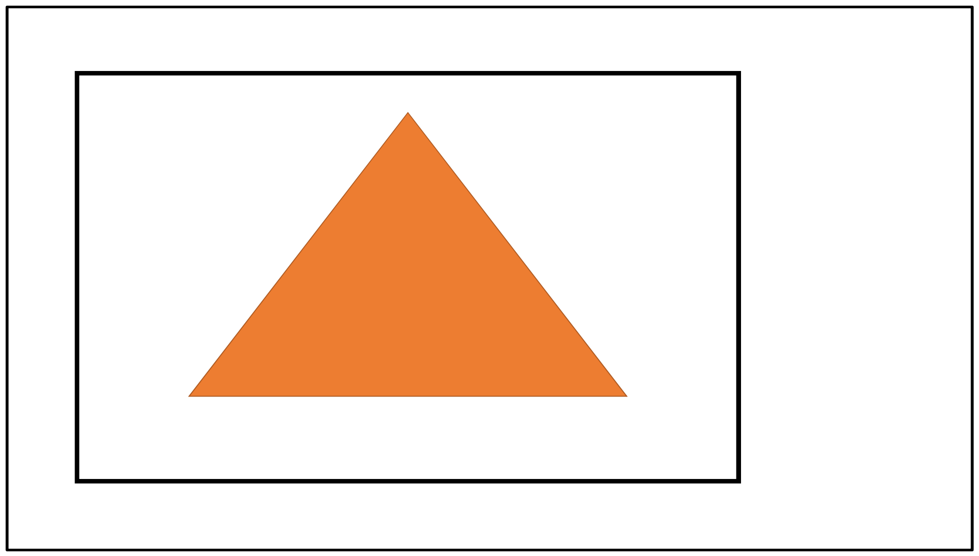
If we put this on a timeline, we get scan-out then vblank, scan-out then vblank, and so on, all happening at a fixed 60Hz.



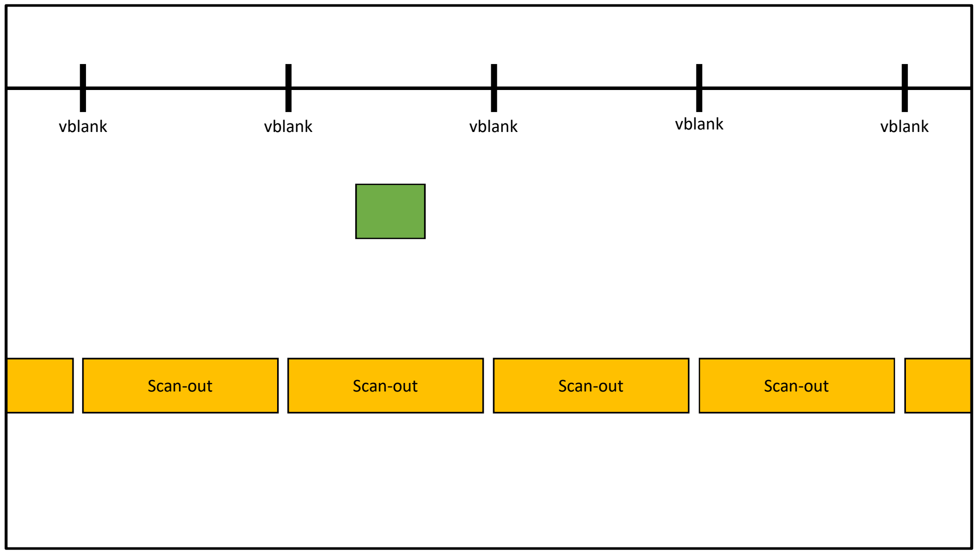
Now we know that the end of the latency duration is fixed: it occurs every ~16.6ms. That means we can predict in advance exactly when the scan-out will begin for this frame.
Our goal now is to figure out where the rest of the timelines land in relation to the fixed scan-out.



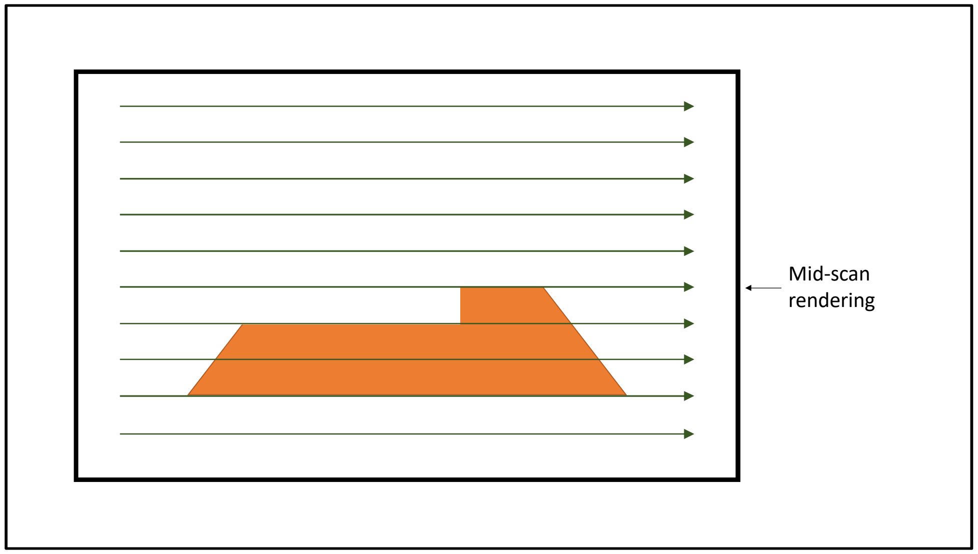
Moving up the chain, the GPU is what renders the image to the framebuffer.



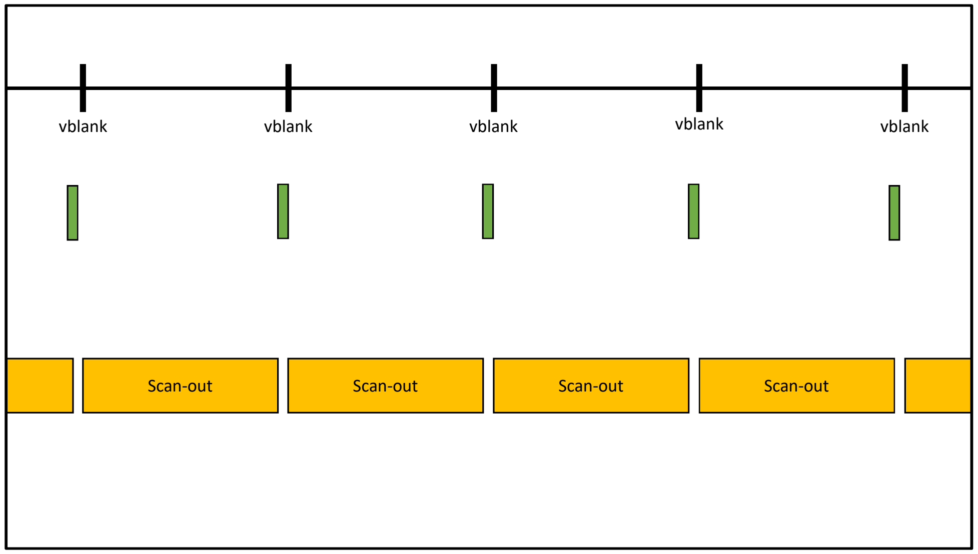
So let’s say we want the GPU to render a big triangle. That means writing a bunch of data into frame buffer memory.



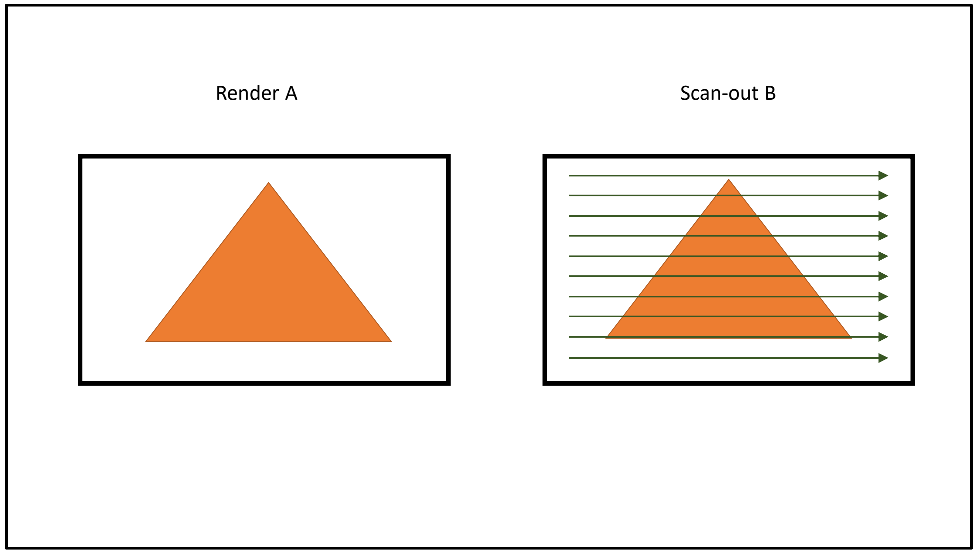
Where does all this rendering show up in the timeline? Let’s say all the rendering happens here.
But remember that the scan-out is continuously reading from the framebuffer and sending data to the display.
That means that we’re simultaneously reading and writing into the same memory: a race condition.



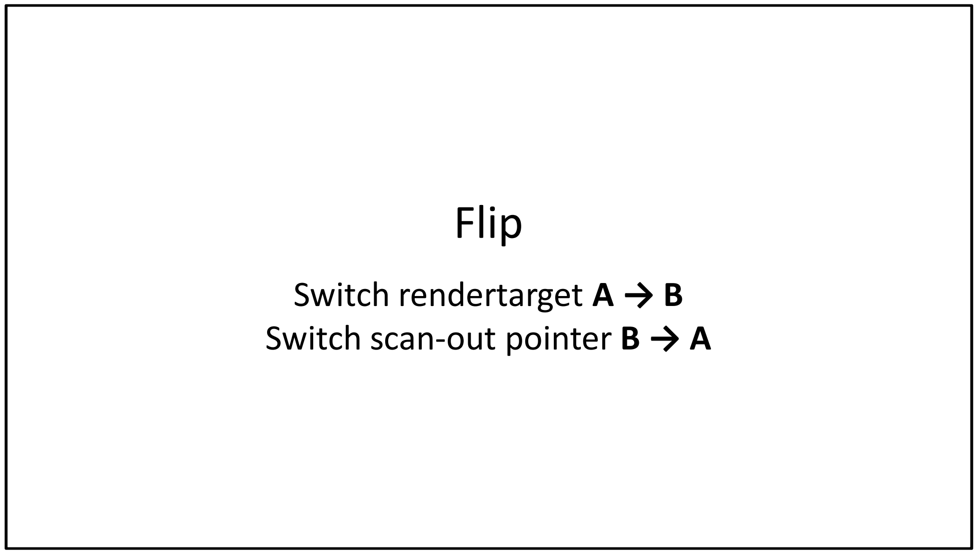
The first part of the scan-out transmits pixels before the triangle rendered. The GPU swoops in and draws the triangle, and scan-out proceeds with triangle pixels leading to pretty bad artifacts. Let’s decide on a rule: don’t render into a framebuffer while it’s being scanned out.



With that rule, is there any time left for the GPU is allowed to render?
One solution is to only render when no visible pixels are being transmitted: during the HBLANK and VBLANK periods. In fact, really old games on systems with a single dedicated framebuffer used to do this.
Of course this isn't practical today.



The conventional solution is double buffering: We allocate two frame buffers, and render to alternate buffers each frame. While rendering into framebuffer A, framebuffer B gets scanned out. Then we render to framebuffer B, and scan-out from framebuffer A.



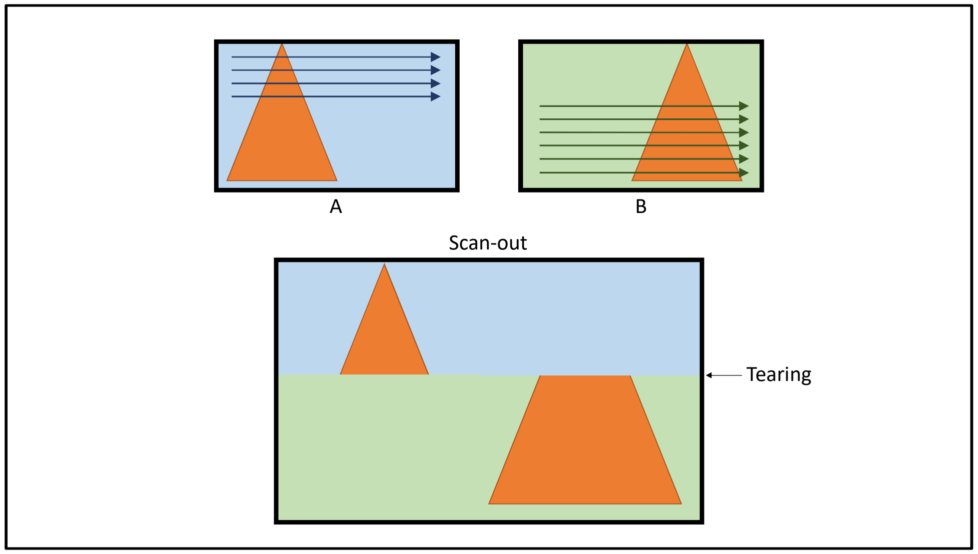
Flip

Switch rendertarget **A → B**
Switch scan-out pointer **B → A**

This switching operation is called the 'flip'.

Two operations are performed during the flip:

- First the rendertarget the GPU is rendering to switches from A to B
- And the address of the framebuffer that the scan-out reads from switches from B to A



It’s not sufficient to flip immediately after the GPU is done with a frame.

Let’s say rendering A finishes. There’s a flip, and A starts to scan-out while rendering begins for B.
B finishes rendering before the scan-out finishes for A. What happens if the flip happens now, right after B finishes rendering?

The scan-out would then switch in the middle of reading from A to reading from B without resetting the scan-line position. The resulting image on screen would have the top half scanned from buffer A, while the bottom half of the screen would be scanned from buffer B.
This artifact is called tearing, and is most visible when there’s large horizontal movement between A and B, like if the game camera is rotating.

Solving this requires another rule: Only flip when the scan-out is done. In other words, only flip during vblanks. This rule is called vertical blank synchronization, or vsync. All of the games we ship on console have vsync enabled.

Mid-scan rendering

Fix: Double buffering

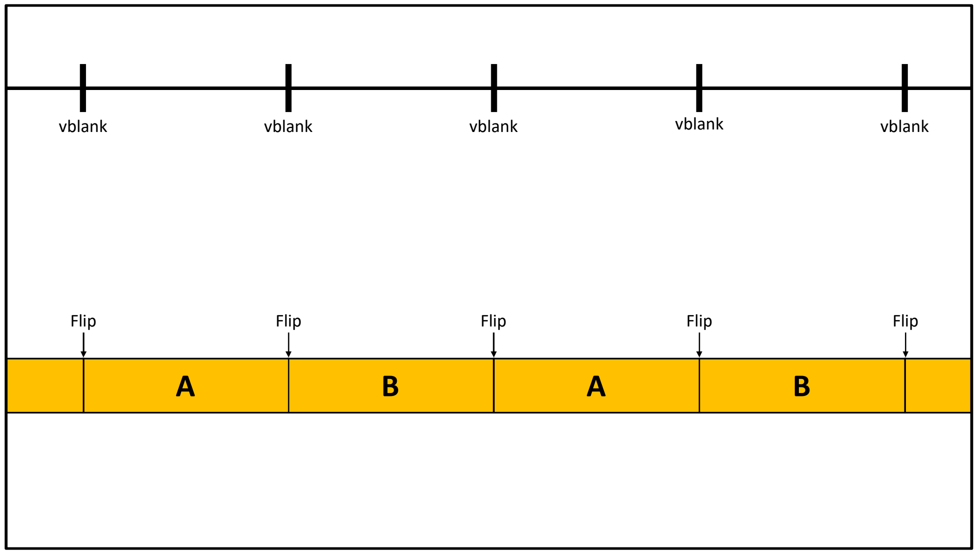
Tearing

Fix: Wait until a vblank to flip

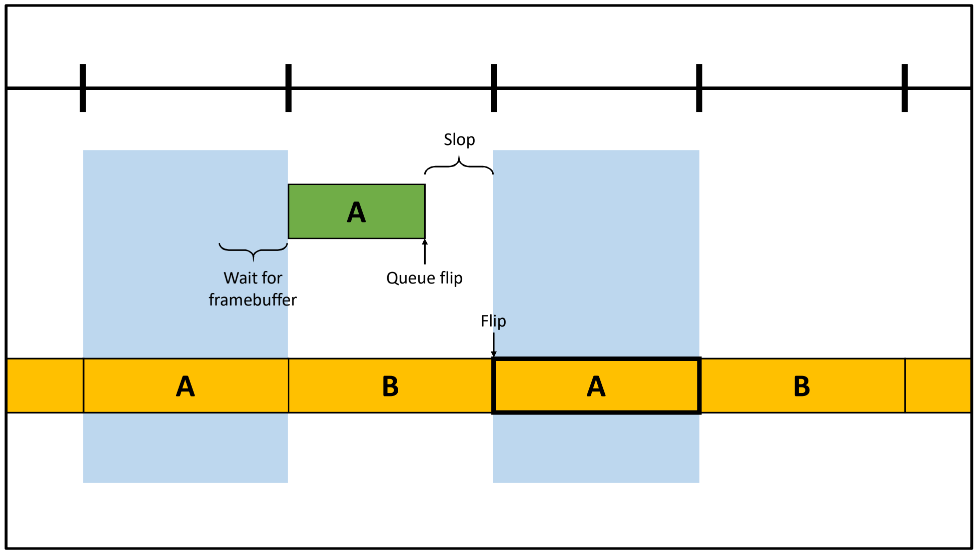
I think we're all already familiar with double buffering and vsync, but it always takes me a minute to work out what they are and exactly why they're necessary.
To review, there are two distinct artifacts we're trying to avoid.

First: if we render to a framebuffer while it's still being scanned out, we get partial renders in the scanned image. This is fixed by double buffering, and never rendering to a buffer that's currently being scanned-out.

Second: if we flip during the middle of a scan-out, we get tearing artifacts between the two framebuffers. This is fixed by only flipping during vblanks, or vsync.

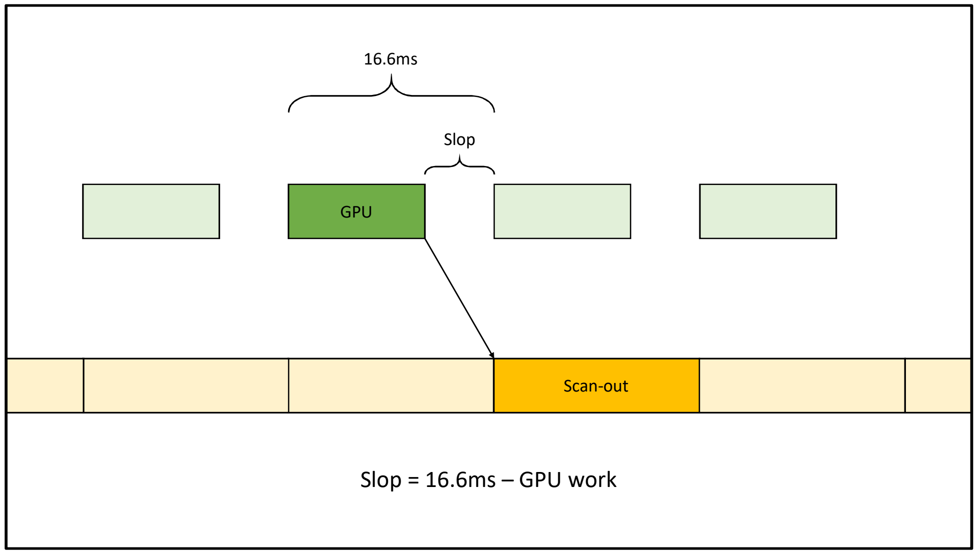


With double buffering at 60Hz and vsync, flips between framebuffer A and B happen after each scan-out during the vblank.



Let’s say the GPU is rendering into frame buffer A.
The periods when the GPU is not allowed to touch A are the regions in the big blue rectangles. During the first period, the GPU has to idle waiting for framebuffer A to become available. Once the scan-out for A is done, rendering to A is allowed to start.
Once the GPU is done rendering, it can’t immediately flip A because that causes tearing. Instead the GPU “queues a flip”: it says that A is ready to be flipped during the next vblank.

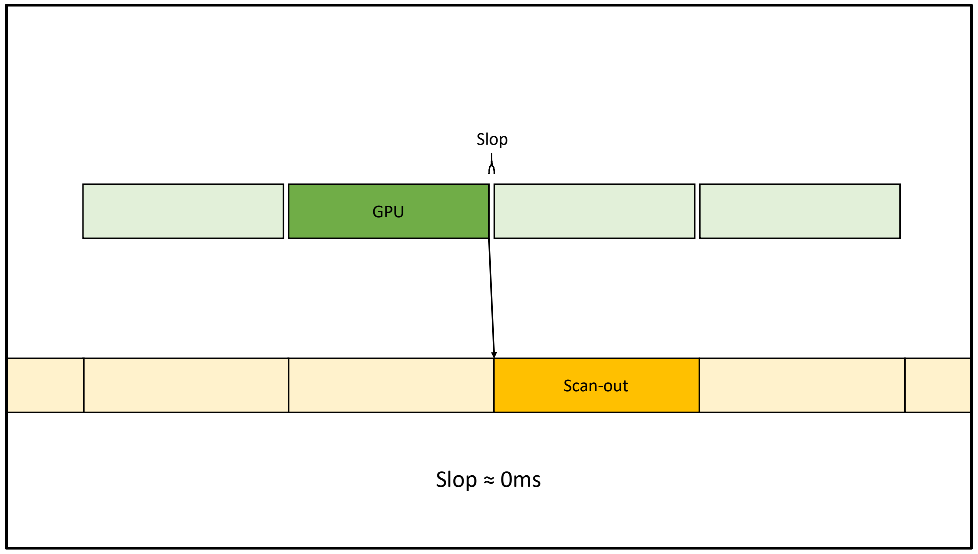
This is the first source of slop: between the queue for flip, and the actual flip.



If you calculate the slop duration, it’s ~16.6ms minus the total GPU Work.

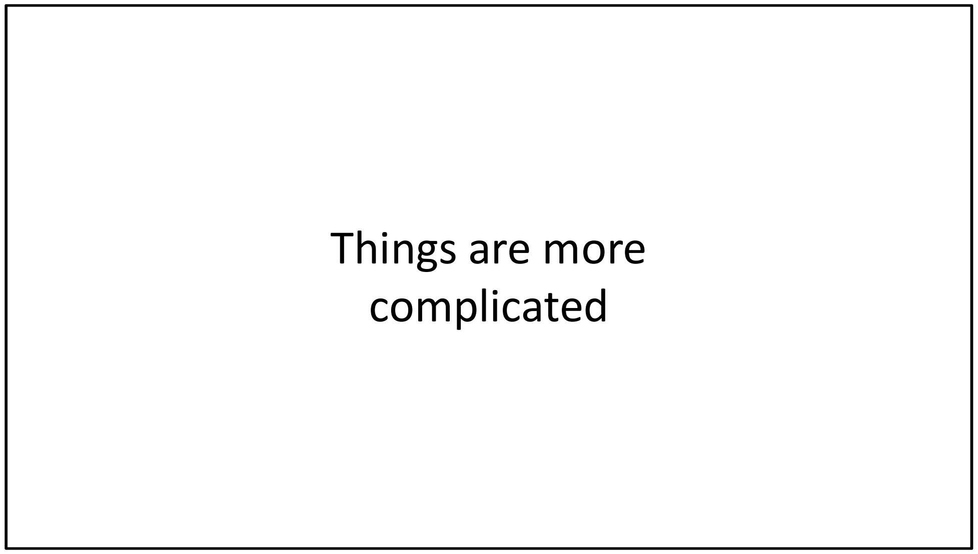
But remember how we want to be using as much of the available hardware as possible. GPU cycles are an especially valuable resource, and we’d be wasting it if the GPU workload were consistently low. If we’re doing our jobs well, we'd be drawing more and at higher resolutions.

What happens to slop if under heavier GPU workloads?



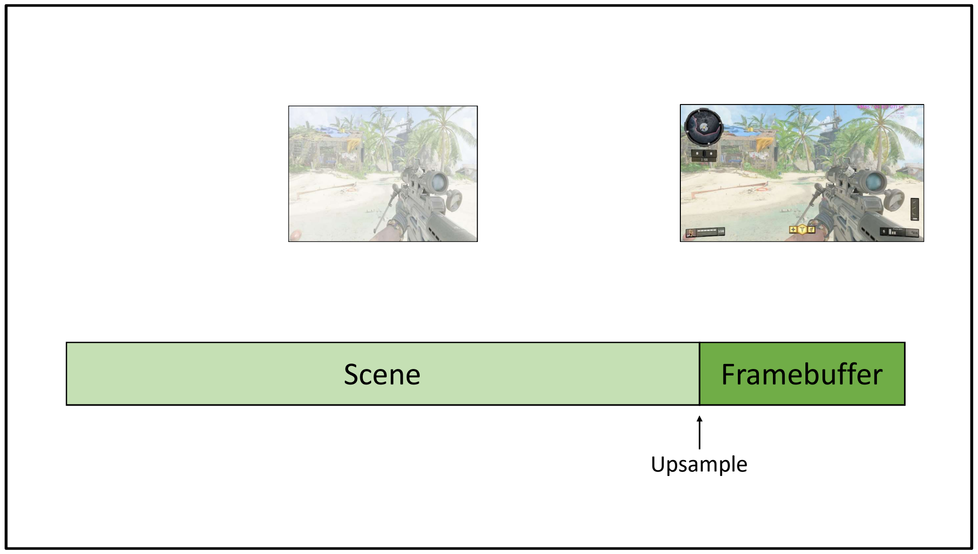
When the GPU workload is full, it approaches ~16.6ms and slop disappears.

But remember that the throttle is supposed to squeeze slop out of the latency duration. If there no slop here, what was the point of looking at the GPU to scan-out in the first place?



Things are more complicated

In reality, things are a little more complicated.
There’s actually a huge source of slop here, even with heavy GPU workloads.

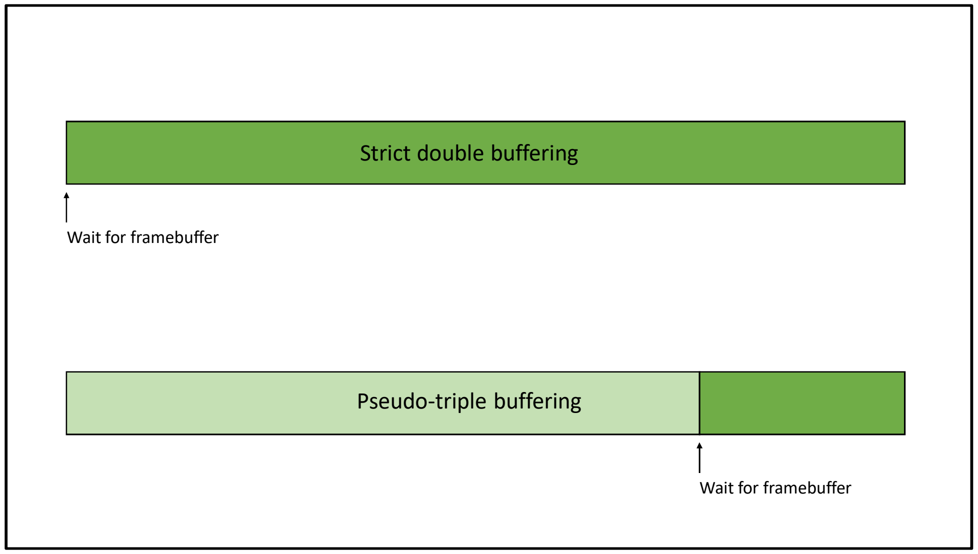


The reason is that most of the draws for our frames are rendered somewhere else, not to a frame buffer.

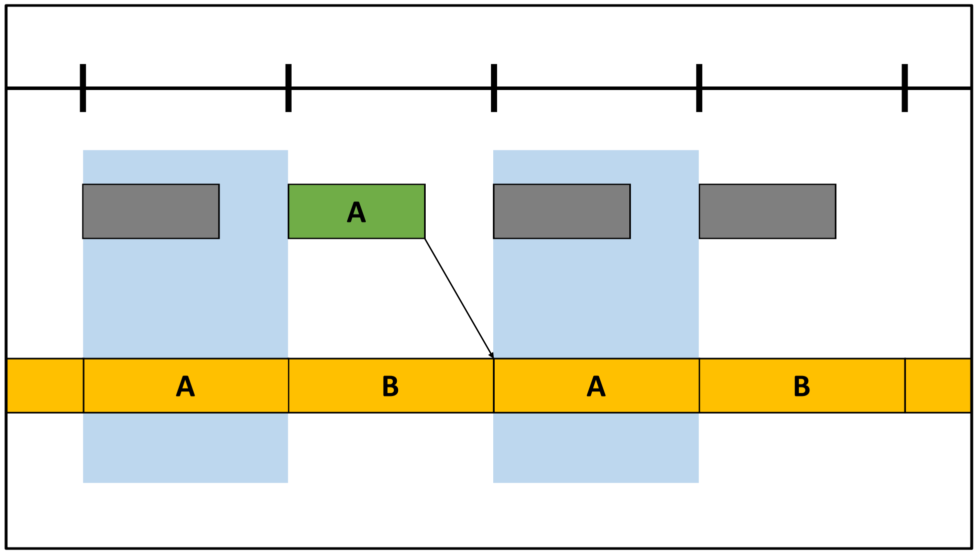
Almost all of our GPU time is spent rendering to off-screen buffers, with the majority of our 3D draws rendered into a third off-screen buffer called the 'scene buffer’
The scene buffer can have a lower resolution than the final display resolution and can have a different color encoding.

When scene rendering is complete, the scene buffer gets upsampled into the frame buffer, typically with temporal anti-aliasing applied during the upsample.
After the upsample, UI elements a rendered into the framebuffer at display resolution, and then the framebuffer is queued for flip.

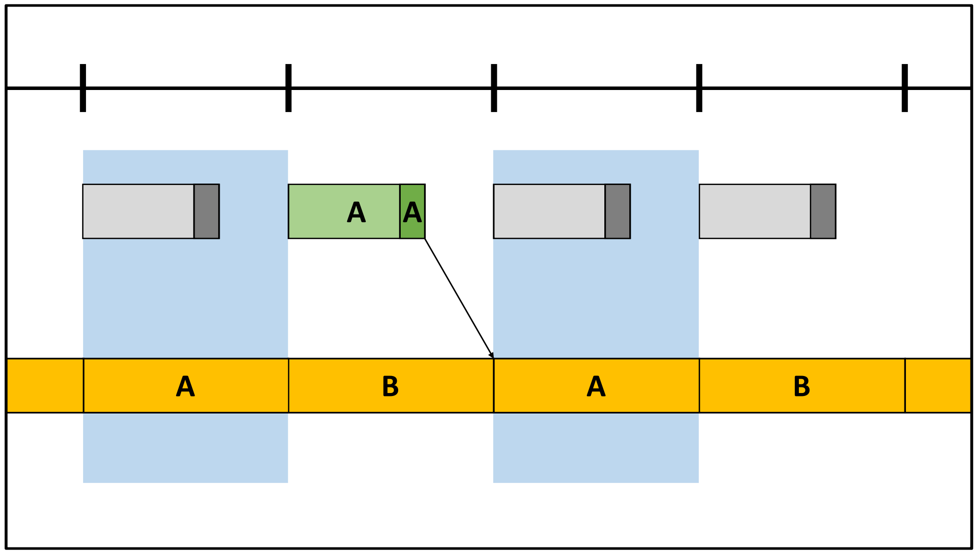
The important point here is that most of the GPU frame time is spent rendering the 3D scene, and the framebuffer doesn’t get touched until the upsample late in the frame. This isn’t exactly triple buffering, because the scene buffer is never scanned out to the display. But the timings work out similarly to triple buffering, so let’s call this ‘pseudo triple buffering’.



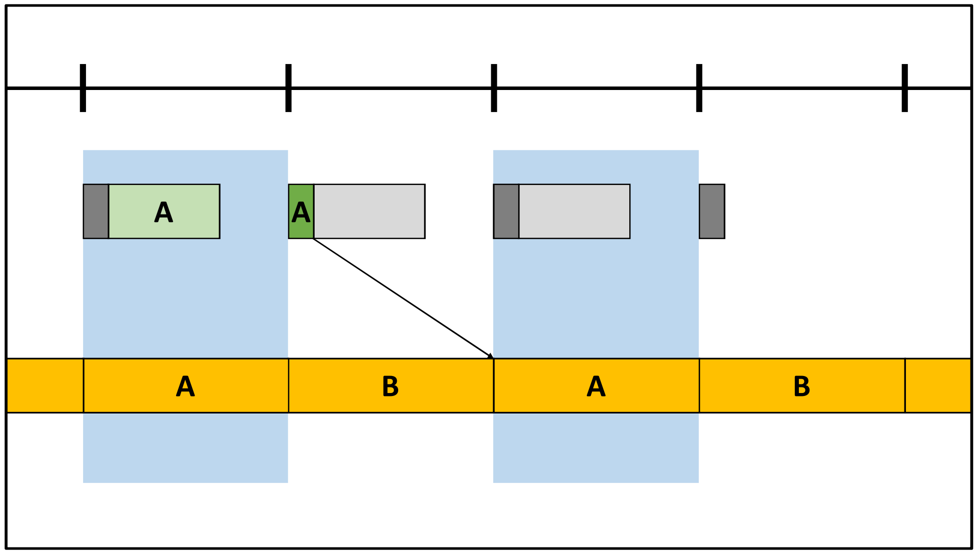
In the original double-buffering setup, the GPU needed to wait for a framebuffer at the very beginning of the frame, syncing the GPU timeline with the scan-out timeline.
Now with pseudo-triple buffering, the GPU only needs to wait for the framebuffer late in the frame, right before the upsample.



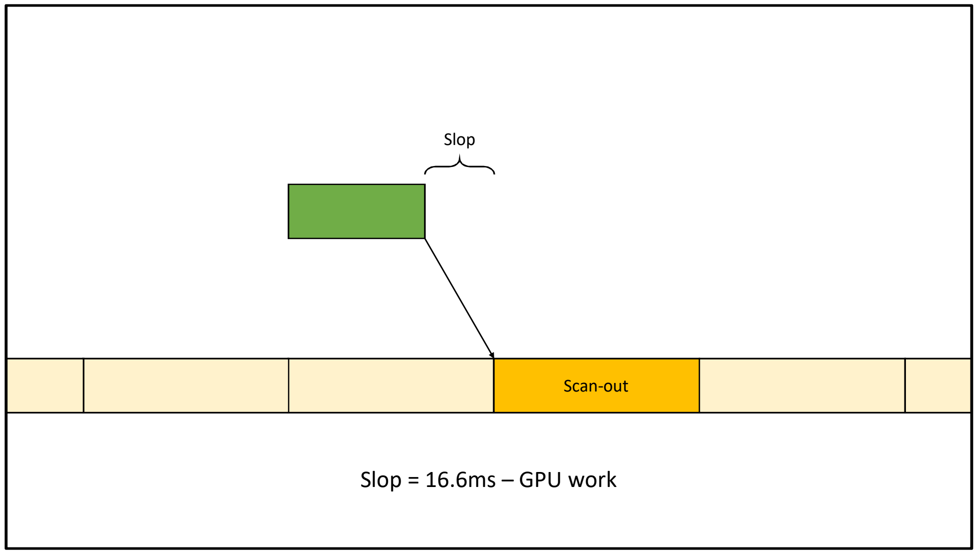
Remember with strict double buffering, rendering to framebuffer A was only allowed in the period between the blue rectangles.



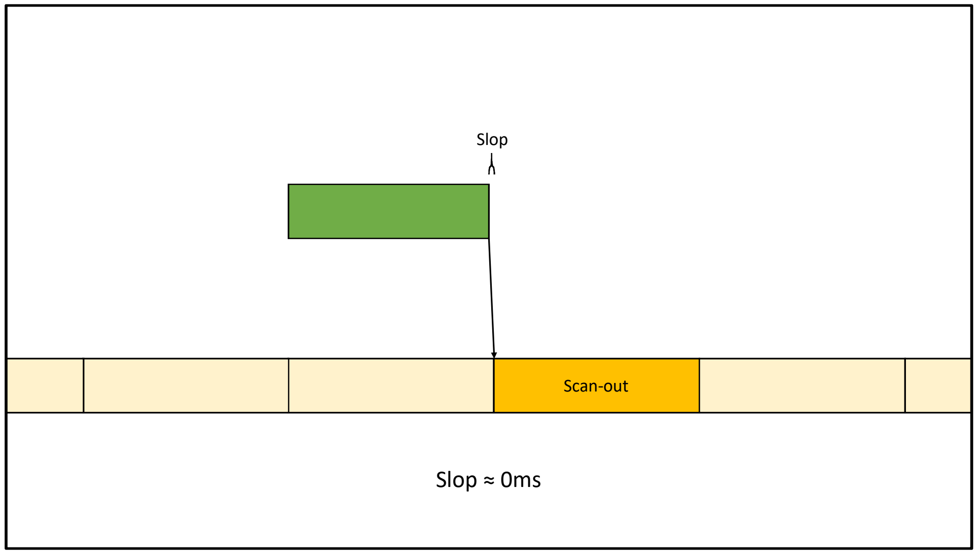
Now the GPU workload is split into two parts: scene rendering, which doesn't use the framebuffer, and framebuffer rendering, which does.
The framebuffer rendering section still has to live between the two blue rectangles to avoid mid-scan rendering, but the scene rendering section is allowed to start whenever it wants to.



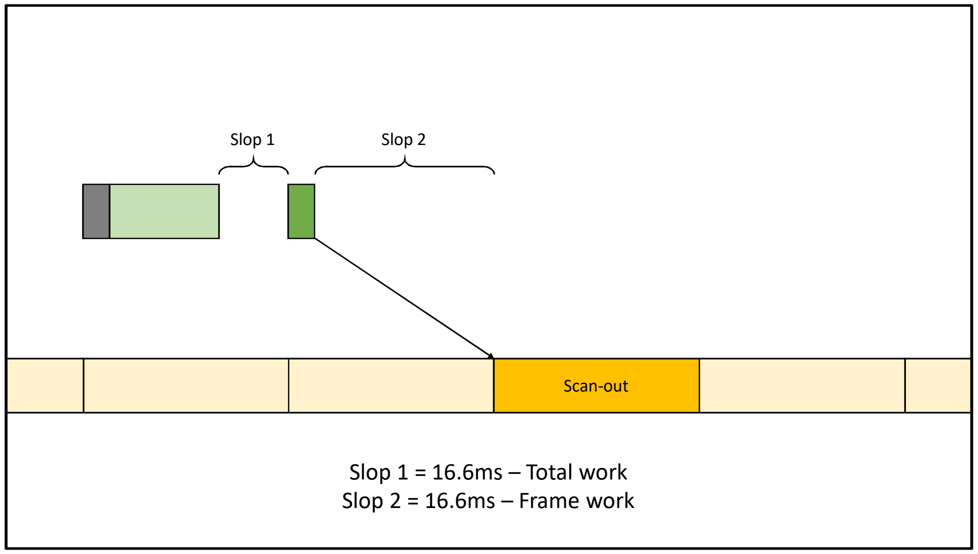
Everything is allowed to shift backwards in time. The GPU starts scene rendering as soon as it’s done with last frame. Framebuffer rendering still has to sync with the scan-out, but it gets shifted back into the space left by scene rendering.



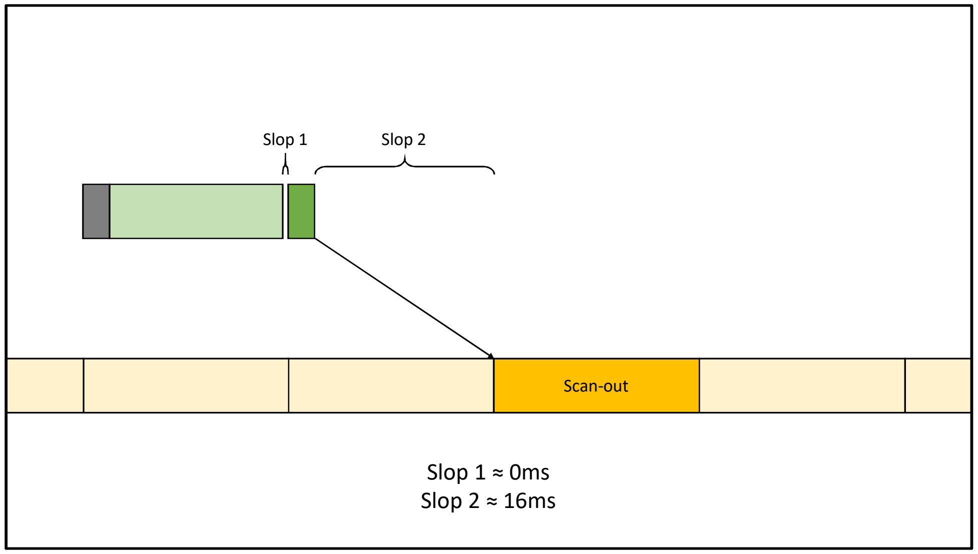
Remember that with strict double buffering, slop was just ~16.6ms minus the total GPU work time,



and that meant with full GPU workloads, slop disappeared.

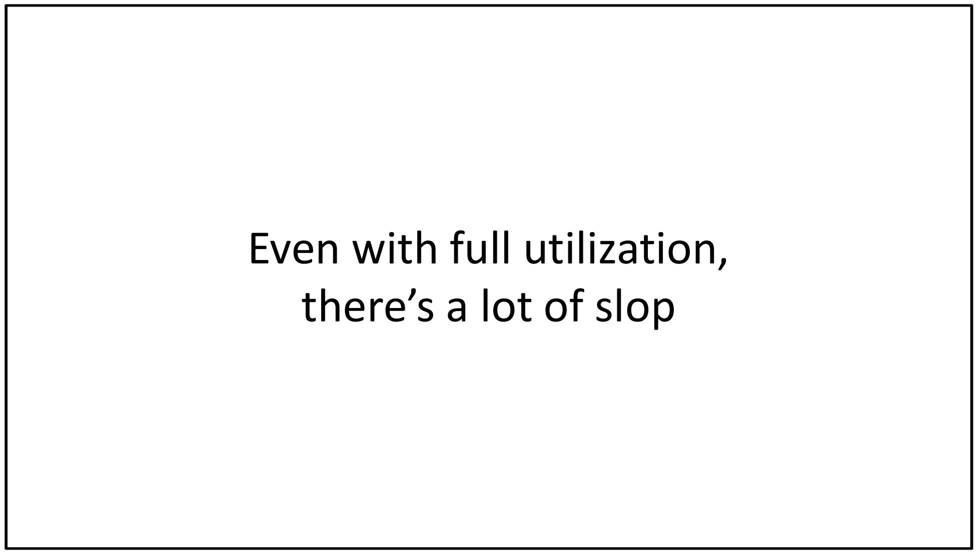


Now with pseudo-triple buffering, slop get's split into two parts.
The first slop duration is the framebuffer wait: between the end of scene rendering and beginning of framebuffer rendering. That comes out to ~16.6ms minus the total GPU workload,
And the second slop duration is between the queue-for-flip and the actual flip, which comes out to ~16.6ms minus just the framebuffer rendering workload.



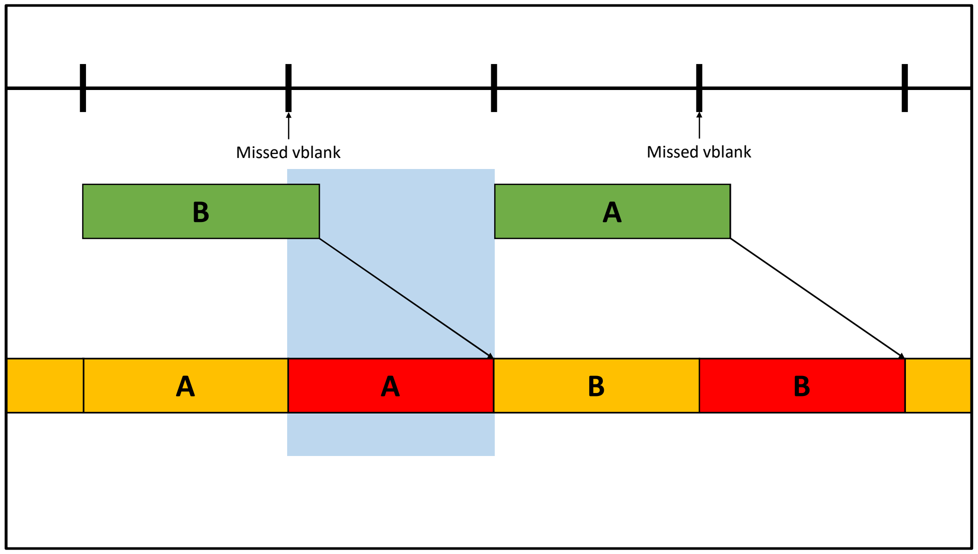
A full GPU workload generally means more scene rendering: more 3D objects and higher scene resolutions. Framebuffer rendering typically stays short.

So when the GPU workload is full, the first slop still disappears while the second slop can stay very long.


Even with full utilization,
there's a lot of slop

That means that even though the GPU is working at nearly full capacity, it's still possible to have pretty huge slop, and that's slop that we can squeeze out by throttling.

I've found that this slop duration was not being explicitly measured or controlled in the games I've worked on. Even when the GPU was being properly fed, if you measured this duration it would either be wandering around or sitting at around 16ms.

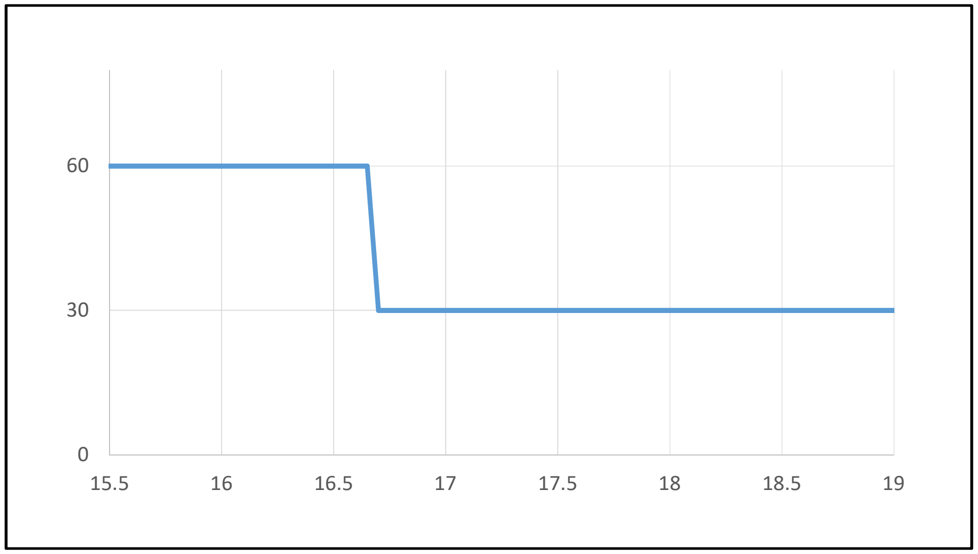


But there’s another consequence to pseudo-triple buffering.

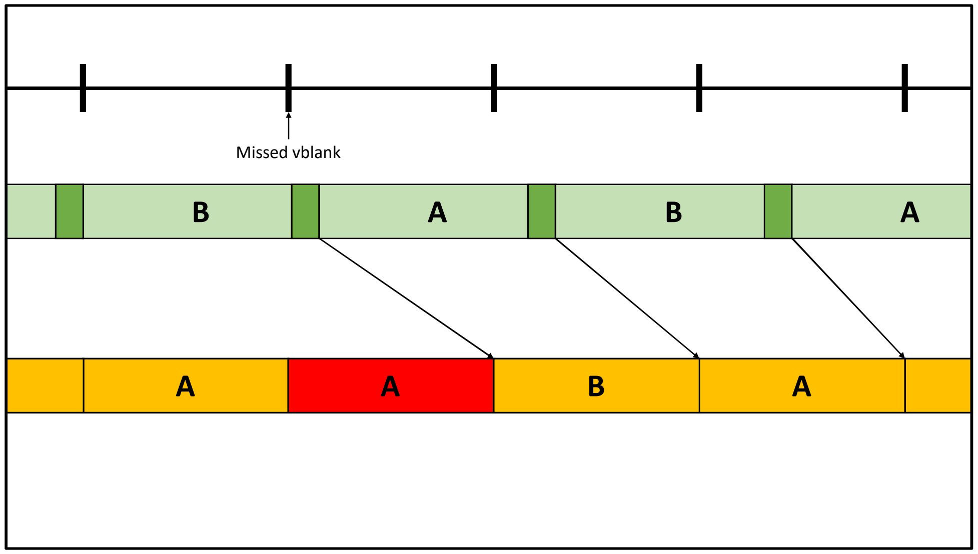
So far we’ve been assuming that the GPU workload was always faster than ~16.6ms. But if you’re trying to keep the GPU as busy as possible, it’s easy to accidentally go over a little and have a slow frame.

Let’s say rendering for frame B was slower than ~16.6ms in the strict double buffering case.
The frame isn’t ready to scan-out by the upcoming vblank interval, so it 'misses the vblank’. No framebuffer is queued for flip, so no flip happens during the next vblank. Instead, the scan-out stays pointed at framebuffer A, and A gets scanned out a second time. The scan-out for B has to wait until the next vblank to start.

Now because A is being scanned out a second time, the GPU has to wait all the way until the next flip to start rendering A. And if rendering frame A is also slow, it misses another vblank, and so on and so on.



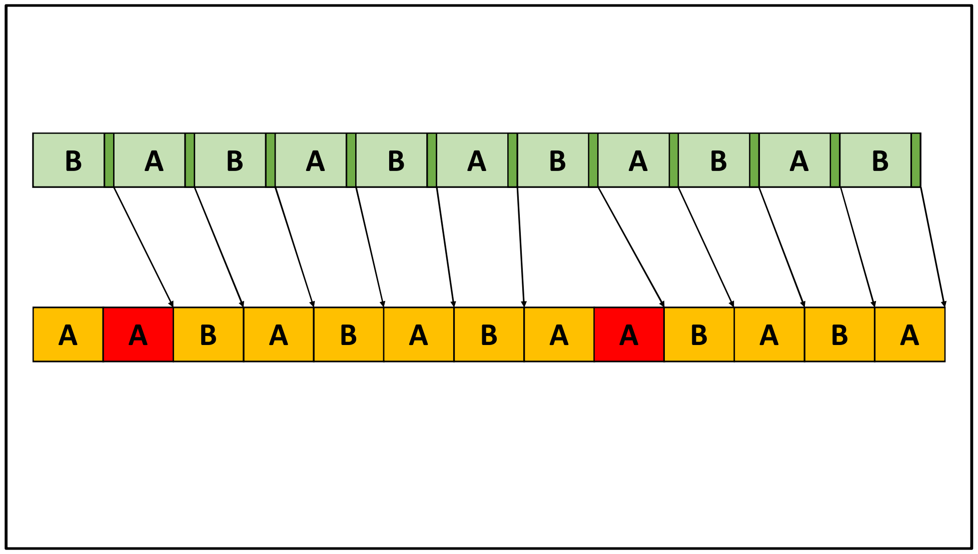
If the GPU render time is consistently slower than the refresh rate, even by a microsecond, it misses every other vblank and the framerate gets pegged to 30Hz instead of 60Hz. This is a pretty disastrous consequence, especially if we’re trying to keep the GPU workload as full as possible.



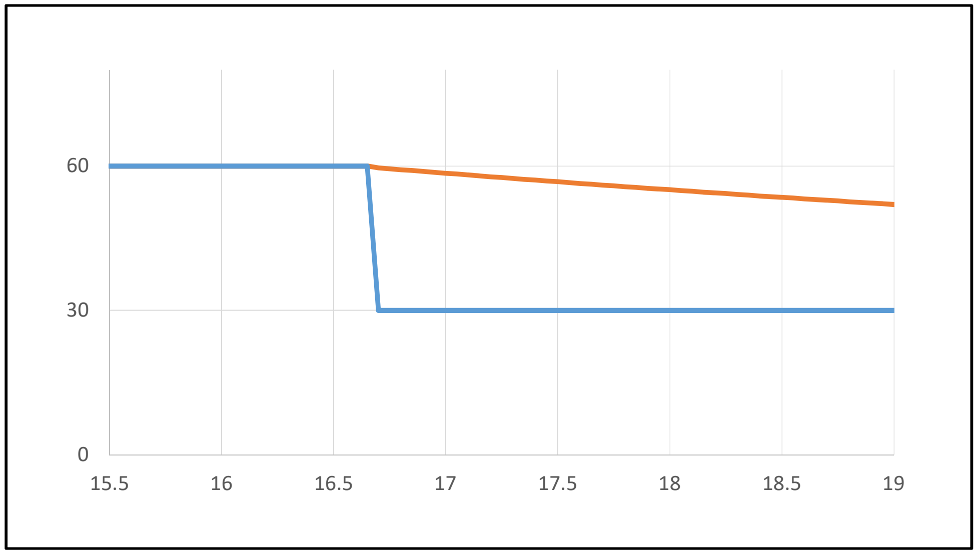
Now in the pseudo-triple buffering case, assume the first vblank is still missed: framebuffer A still has to scan-out twice.

But now scene rendering for A is allowed to start immediately, even though A is still being scanned out. That’s because there’s no restriction on when the GPU is allowed to render to off-screen buffers.

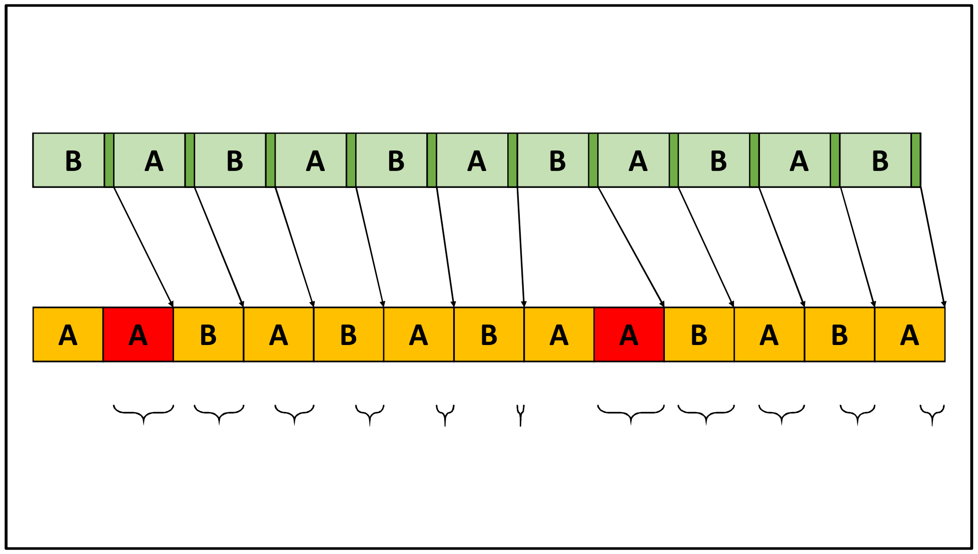
This is a pretty big difference from the double-buffer case, where the GPU had to idle all the way to the next vblank. Even though the GPU workload is still greater than ~16.6ms, A doesn’t miss the next vblank.



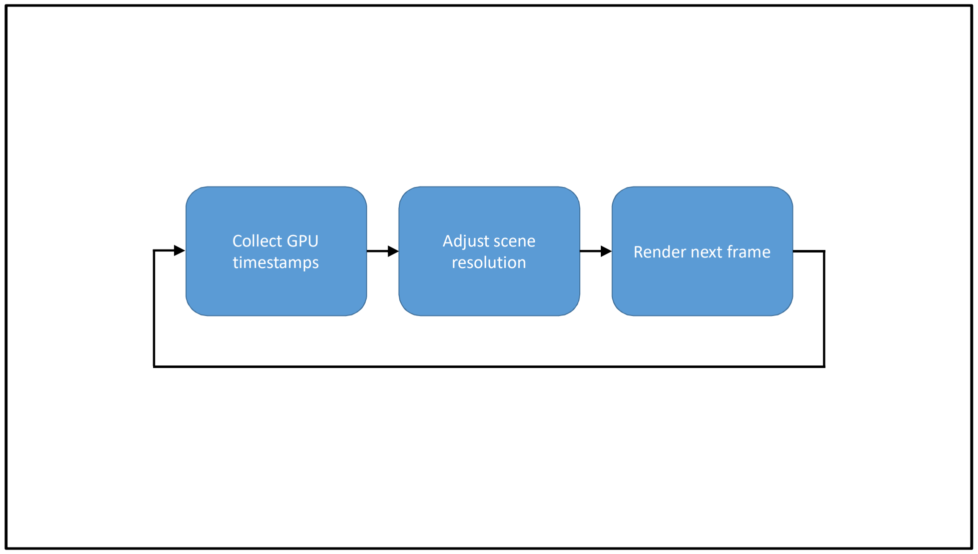
Zooming out and looking at the behavior over multiple frames (assuming a fixed, slower than 16.6ms GPU workload) The very first frame misses the vblank, and A gets scanned out twice. But then the next frame, and the next frame, and the next frame all make their vblanks. Ultimately a vblank does get missed again, but not before six slow frames go through that don't miss their vblanks.



With pseudo triple-buffering, average framerate doesn’t experience the cliff to 30Hz.
Instead average frame rate slowly drops from 60Hz into the 50s.



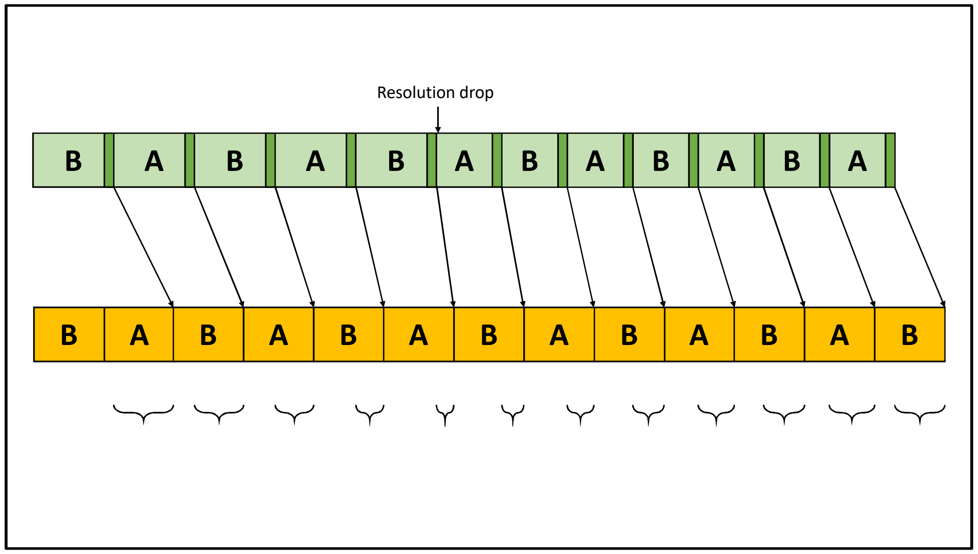
Take a look the braces on the bottom: These are measuring slop between the queue-for-flip and the actual flip for each of these frames.
Notice that after the first missed vblank, slop is high. As slow frames go through, slop gradually decreases. Each slow frames eats into the available slop, until slop finally drops below zero and the vblank is missed.
This illustrates an important point about slop: slop serves as buffer room for slow frames to go by without missing a vblank.



Being able to stay in the 50s is better than dropping to 30, but it still creates an unpleasant effect where a single frame is missed every few frames.
We'd much rather stay at a consistent 60, and that's where dynamic resolution comes in.

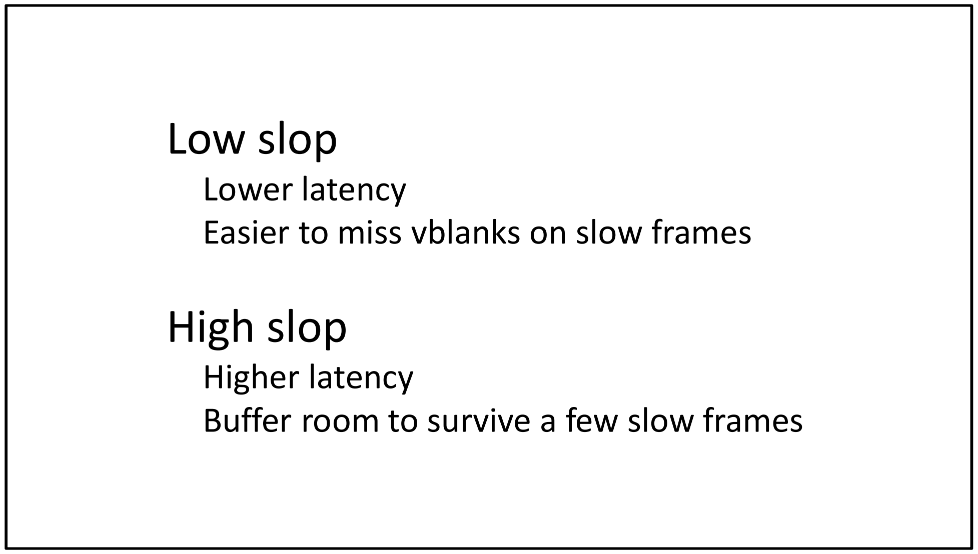
When the game notices GPU times are getting slower than a threshold, the engine cranks down scene resolution to stabilize frame times. No matter what, the game has to experience at least one slow frame before the lower resolution can kick in, so there's an intuition that at least one vblank miss is guaranteed before dynamic resolution can make an impact.

And that would be true with strict double buffering if the slow frame threshold was close to 16.6ms.
To avoid dropped vblanks with strict double buffering, the slow frame threshold would have to be tuned significantly lower than 16.6ms, and no spikes above 16.6ms could be tolerated.



But let's look again at the timeline for pseudo-triple buffering, now that there's dynamic resolution.

A few slow frames go by without missing a vblank. Slop gets dangerously low, but finally the resolution drop kicks in.
Now GPU frame times are faster than 16.6ms.
Notice that as fast frames go by, slop accumulates back to safe levels. The idea is that slow frames eat into slop, while fast frames build it back up.
With the extra slop that comes with pseudo-triple buffering, the slow frame threshold can be turned up closer to 16.6ms, and there's a chance of surviving spikes above that threshold.
This property is absolutely vital for dynamic resolution to be a viable technique.



Low slop

Lower latency

Easier to miss vblanks on slow frames

High slop

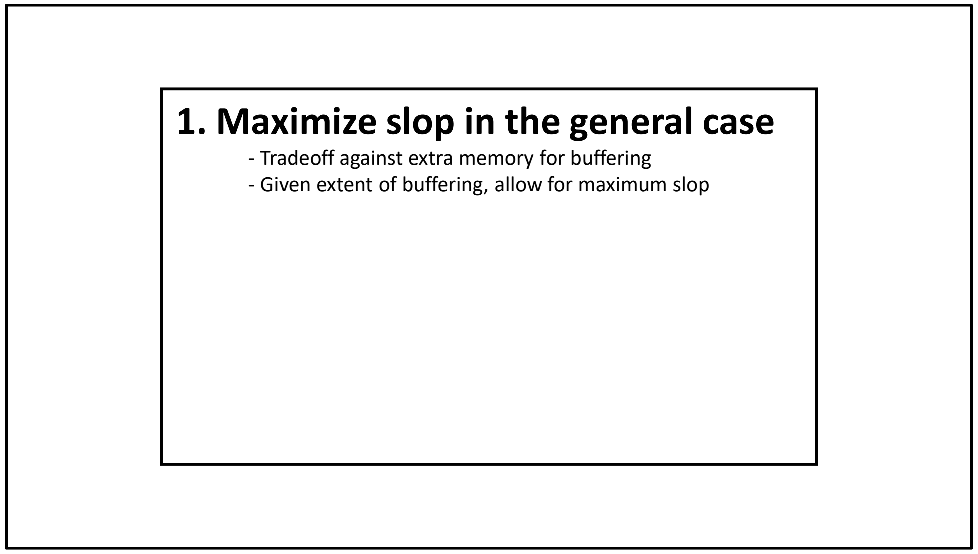
Higher latency

Buffer room to survive a few slow frames

This illustrates the fundamental trade-off of latency throttling:

On one hand, we want to reduce slop because the goal is to reduce latency without reducing work.
On the other hand, slop is necessary for surviving a few slow frames without missing a vblank.

Given this trade-off, how do we proceed?



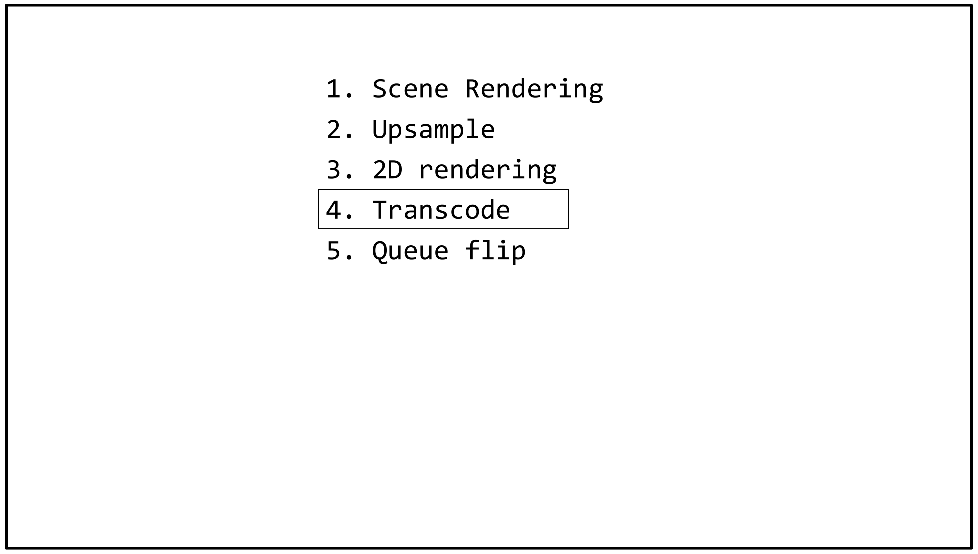
1. Maximize slop in the general case

- Tradeoff against extra memory for buffering
- Given extent of buffering, allow for maximum slop

Here's the plan we went with when looking at the last few Call of Duty games:
In the general case, the game should be coded to maximize slop given the extent of the engine's buffering. That way in the general case things will be as safe as possible. Only at runtime when the coast is clear is the throttle introduced to squeeze out slop and reduce latency.
The extent of the game's buffering is a tradeoff between extra memory and increased maximum slop.
Take pseudo-triple buffering for example: this seems like a good amount of buffering between the GPU and Scan-out for Call of Duty games

- Two framebuffers already need to be allocated to avoid mid-scan rendering
- A scene buffer already needs to be allocated to support dynamic resolution

We could go with true triple buffering or even quadruple buffering, but in practice there are diminishing returns in terms of safety.
Once the extent of buffering is determined, the engine needs to be allowed to achieve this maximum. That means synchronizing buffer access as late as possible.



1. Scene Rendering
2. Upsample
3. 2D rendering
4. Transcode
5. Queue flip

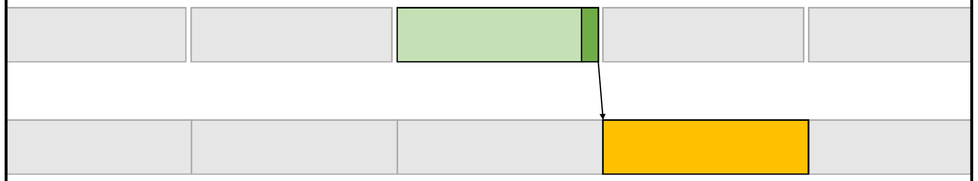
Here’s a really simple example of maximizing slop in practice from a recent Call of Duty game.
When HDR support was added, the scene buffer – frame buffer split changed.

- First scene rendering is performed as usual to the scene buffer,
- The scene buffer is upsampled to display resolution, but this time into another off-screen buffer at display resolution
- UI is rendered at display resolution to this off-screen buffer,
- Finally at the very end of the frame, this buffer is transcoded to the actual framebuffer in it’s final color space

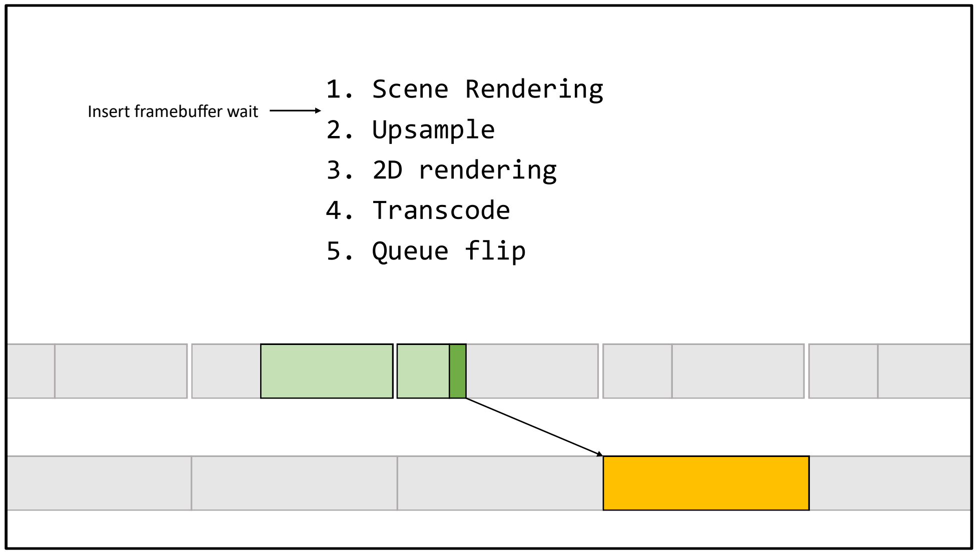
Note that the first time a real framebuffer is touched is right before the transcode.
The framebuffer is only used for a very small portion of the frame time, or about 170 microseconds at 1080p.

Insert framebuffer wait $\longrightarrow$

```
1. Scene Rendering
2. Upsample
3. 2D rendering
4. Transcode
5. Queue flip
```

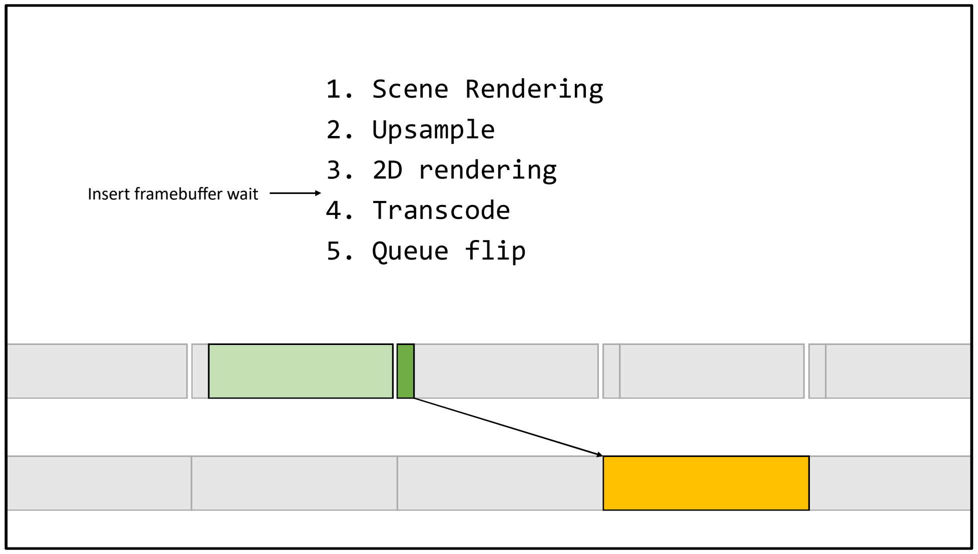


On one platform, the framebuffer was originally acquired at the very beginning of the frame.
This totally counteracts the benefit of pseudo-triple buffering, making it behave identically to strict double buffering. Any slow frame guarantees a missed vblank, making dynamic resolution less effective.

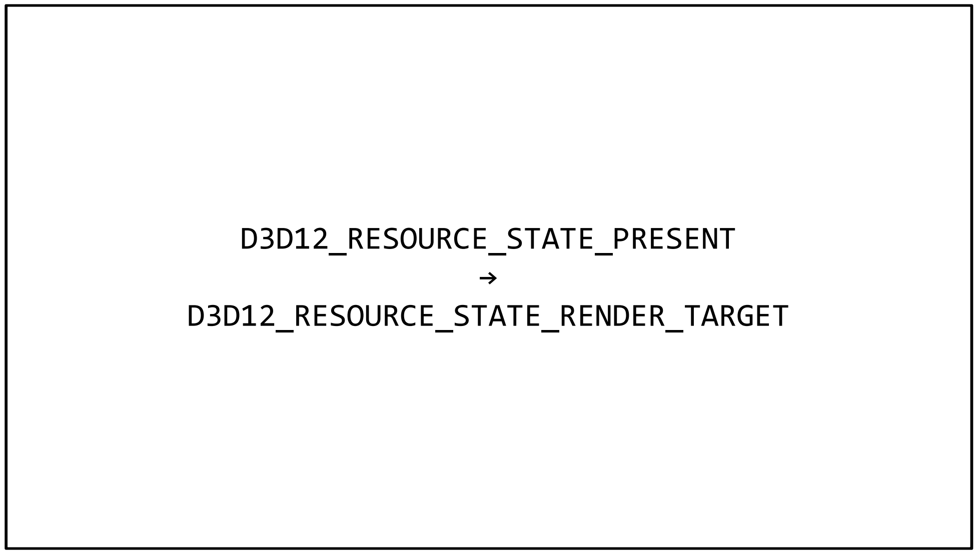


Things were a little better on another other platform:
The wait for frame buffer was inserted after scene rendering, but when HDR support was added, this wait wasn't moved.

This setup behaves better than the previous case, but it still causes the framebuffer portion to be longer than necessary. Even though the frame is allowed to start earlier than with strict-double buffering, slop was not maximized, leading to a higher than optimal chance for missing vblanks.



The correct location to acquire the framebuffer is right before the transcode. This maximizes slop and minimizes the chance of dropping vblanks.

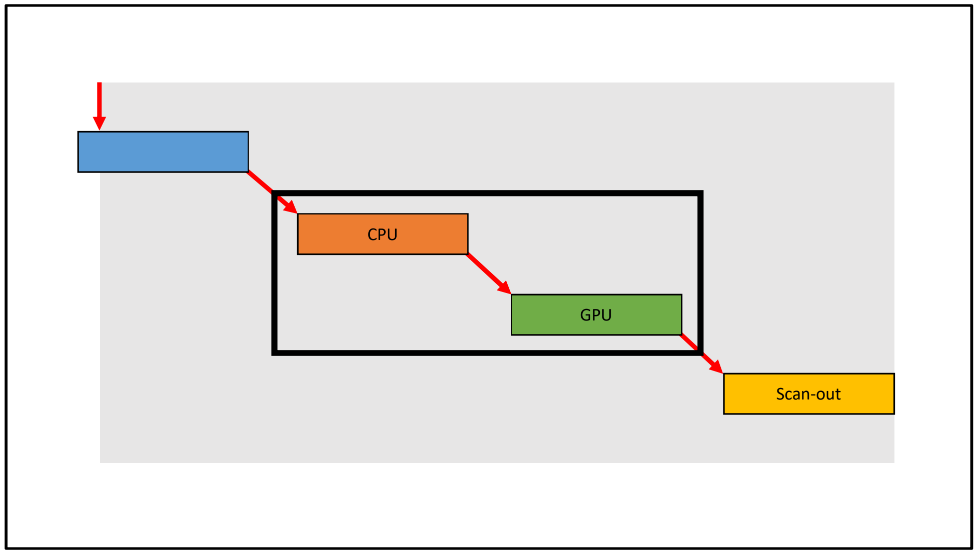


```
D3D12_RESOURCE_STATE_PRESENT
→
D3D12_RESOURCE_STATE_RENDER_TARGET
```

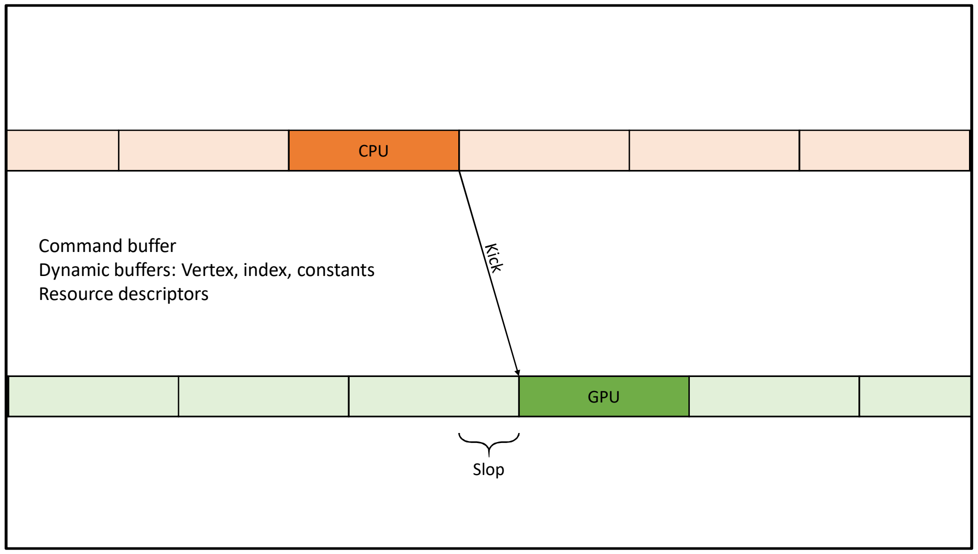
I’d recommend checking your code to make sure the wait for framebuffer happens as late in the frame as possible. It’s typically a single function, but when the structure of a frame changes it’s easy to forget to move it because it has no effect on performance or timing when vsync is off.

With DX12, the wait happens when performing a transition barrier on the framebuffer resource from Present state to the Render Target (or another write) state. There are equivalent functions on other platforms.

I’d also recommend timing these waits. You can stick GPU timestamps around these calls and incorporate the measurements into your GPU timing system. You have to subtract these values from total GPU time to get accurate frame timings for dynamic resolution when vsync is enabled.

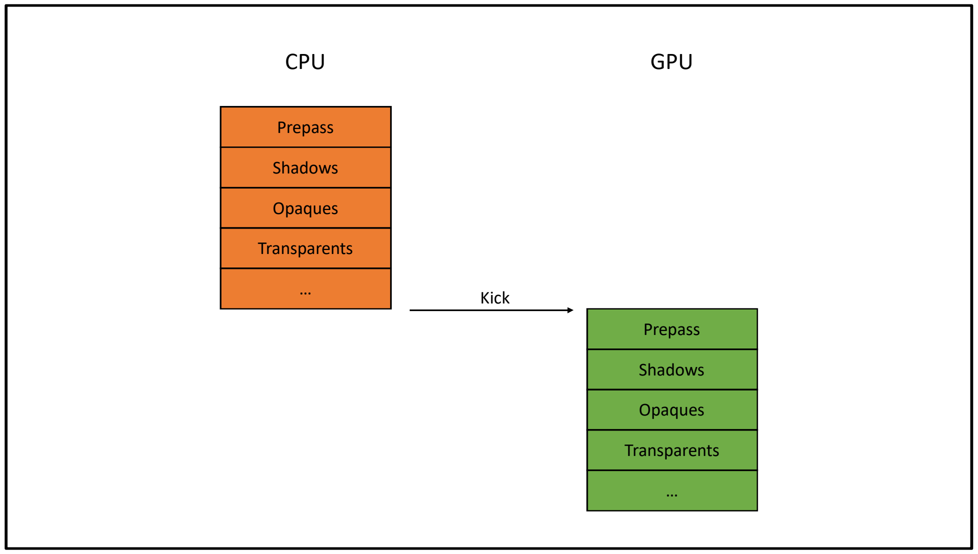


We’ve discussed the GPU to scan-out relationship and identified the potential slop. We’ve concluded that slop can accumulate here even when running with full workloads, but that some slop is necessary to avoid missed vblanks. Let’s move up to the next link in chain: The CPU to GPU relationship.

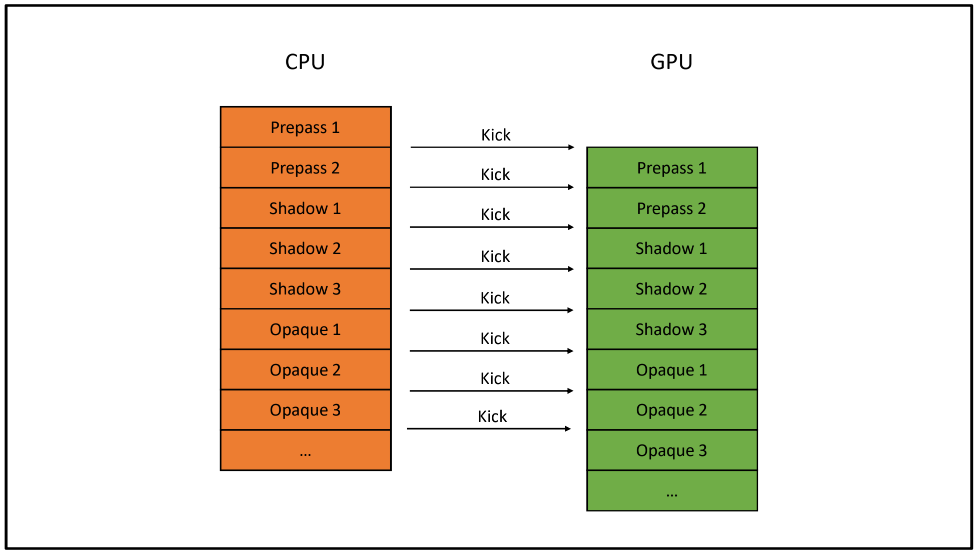


A large part of the CPU's workload is spent telling the GPU what to do. That means recording state changes, draws and dispatches into command buffers for the GPU to consume, as well as generating associated rendering data like dynamic vertex and index buffers. The buffers written by the CPU and read by the GPU are typically either double buffered or allocated from a ring.

After the CPU has built up a command buffer, it tells the GPU to start working on it through a ‘kick’: This kick event is analogous to the framebuffer flip between the GPU and scan-out.
Just like with the GPU to scan-out, slop can accumulate between the CPU’s kick and the GPU actually starting this frame.

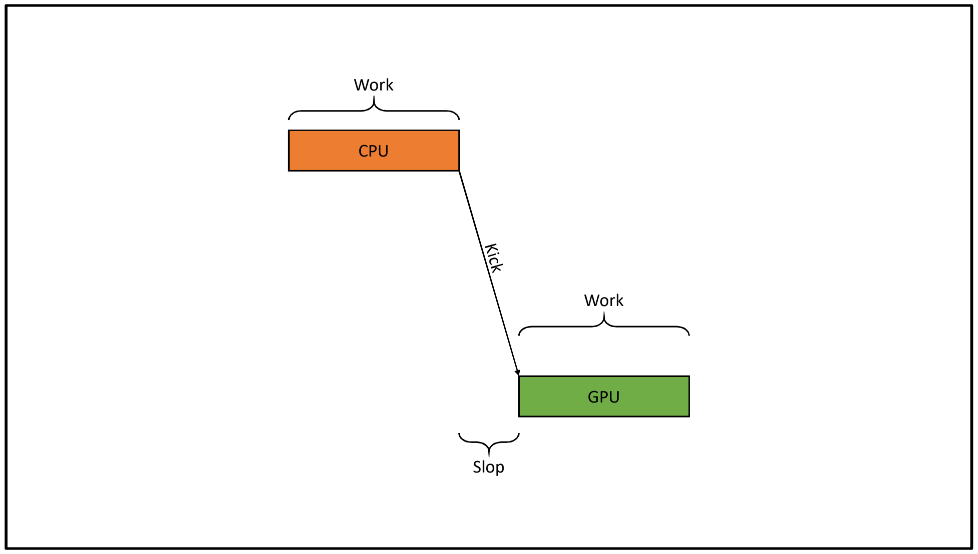


Something that's different from the GPU to Scan-out system though, is that the CPU records commands in approximately the same order that the GPU consumes them. For example, a frame on the CPU might consist of recording commands for a Prepass, then commands for Shadows, then commands for Opaques, and so on. After the kick, the GPU draws in that order too.



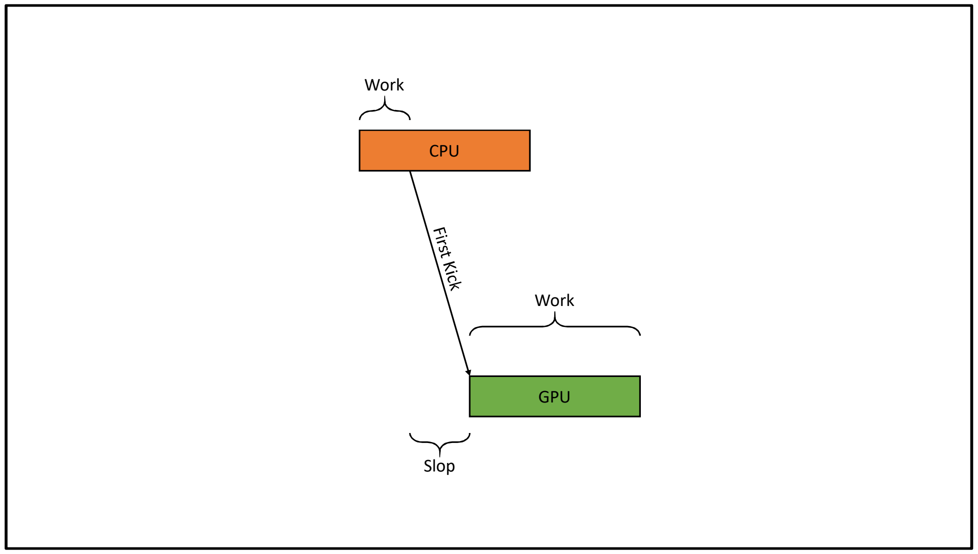
We take advantage of this fact to split the buffers between the CPU and GPU at a finer granularity than one frame. Instead of the CPU generating all of the data for an entire frame upfront and kicking the whole thing once, the CPU can instead make multiple smaller kicks per frame.

For example, the CPU can kick immediately after recording the data for the first half of the prepass. The GPU starts working on the prepass, while the CPU simultaneously records the data for the second half of the prepass, and so-on. This allows for significant overlap between the CPU and GPU frames. To correctly account for how latency works in this overlap situation, the definitions of slop and work need to be modified.



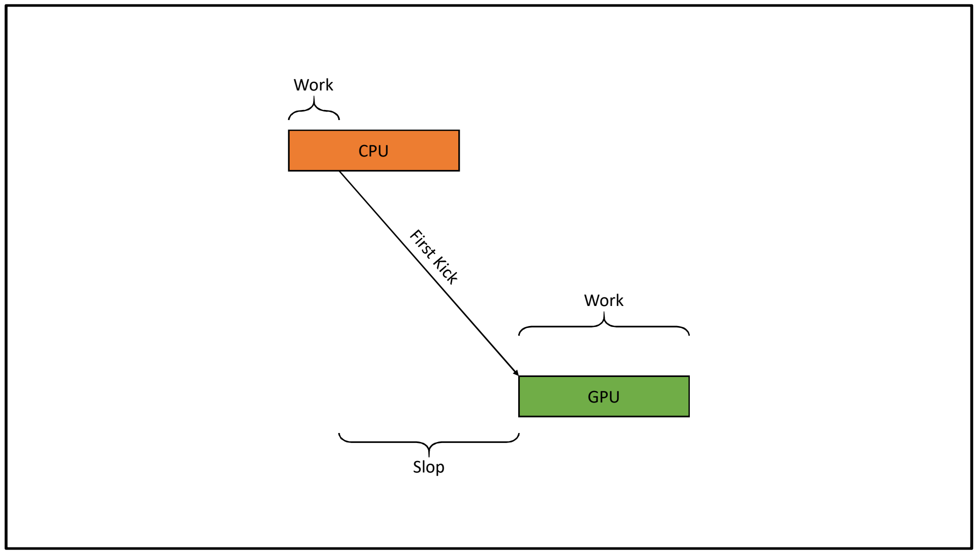
In the non overlapping case, slop was the segment from the end of the CPU frame to the beginning of the GPU frame.

And remember why slop and work were defined this way: When throttling
- Work is the part of the latency duration that moves forward in time, but doesn’t shrink
- Slop is the part of the latency duration that does shrink

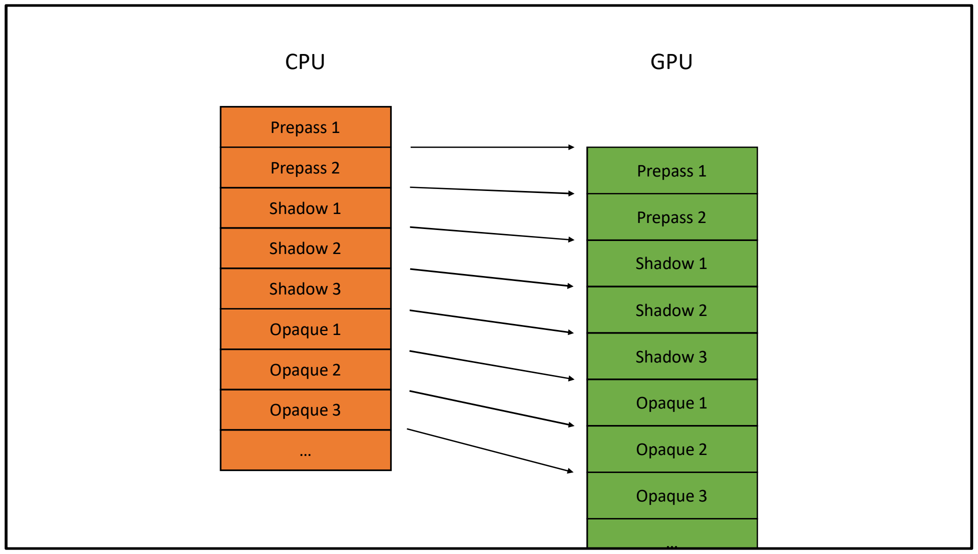


Now with the overlapping case, the GPU is allowed to start working immediately after the CPU makes its first kick.
Slop needs to be defined as the range between the CPU's first kick, and the GPU starting the frame.
And work would be everything else: the duration between the beginning of the CPU frame and its first kick, plus the entirety of the GPU frame time.

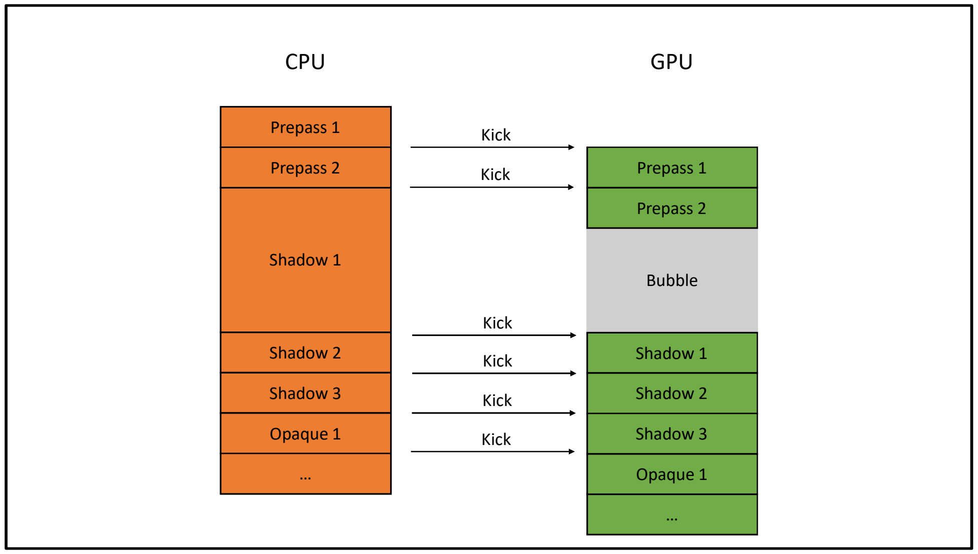
Notice that just by allowing overlap, the measured work has be significantly reduced.



That’s not to say that the CPU and GPU are forced to overlap.
There could still be a delay between the end of the CPU frame, and the beginning of the GPU frames. The measurement for slop would just be much higher, because the GPU could have started earlier. By allowing overlap, all that’s been done is allow the throttle to squeeze further than it could have in the non-overlapping case.

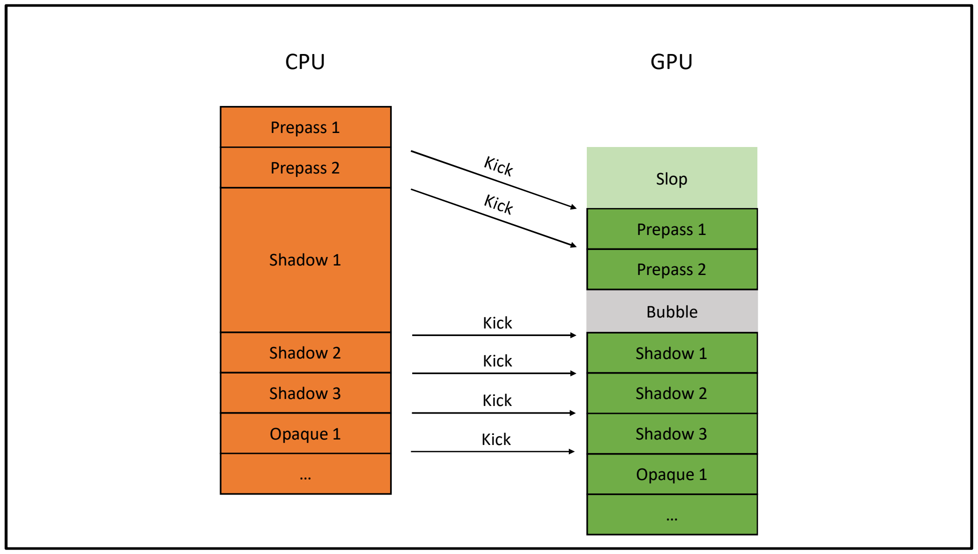


Let’s talk about performance.
Remember that GPU cycles are typically the most precious hardware resource: In Call of Duty we’re more likely heavier on the GPU rather than the CPU. In the ideal case, the CPU would be consistently faster at recording commands than GPU is at consuming it. That way the GPU would never be starved and idle.

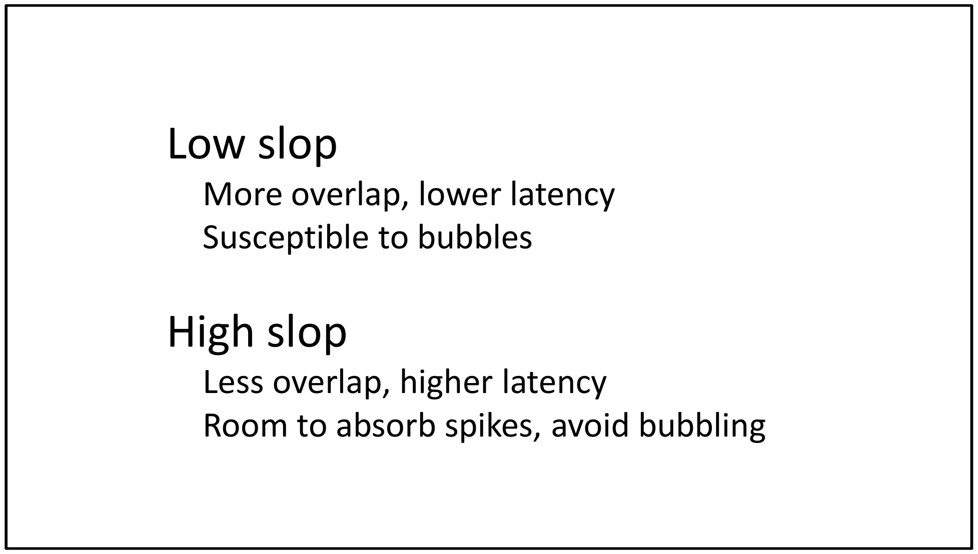


But if one of the CPU segments gets kicked too late, the GPU can go idle: this is called a GPU bubble.

When the GPU and CPU are significantly overlapped and the GPU is only slightly behind the CPU, the risk of bubbling is much higher, especially early in the frame. Bubbling is inefficient because it means idle GPU cycles, but it can also bias the GPU frame time measurements used for dynamic resolution, causing unnecessary resolution drops



But if there were some slop between the GPU and the CPU, that slop can serve as buffer room to minimize the effect of CPU spikes, yielding a smaller bubble or avoiding bubbles all together.



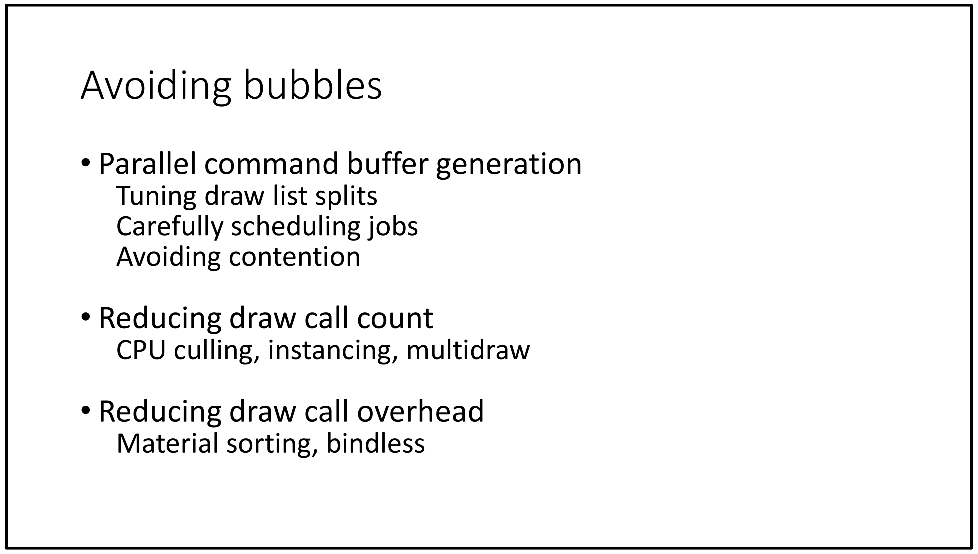
Low slop

- More overlap, lower latency
- Susceptible to bubbles

High slop

- Less overlap, higher latency
- Room to absorb spikes, avoid bubbling

This leads back to the fundamental tradeoff:
Low slop means more overlap, so lower latency. But it also means bigger bubbles on the GPU if the CPU spikes.
Higher slop means less overlap and higher latency, but CPU spikes can be more easily absorbed, avoiding GPU bubbles



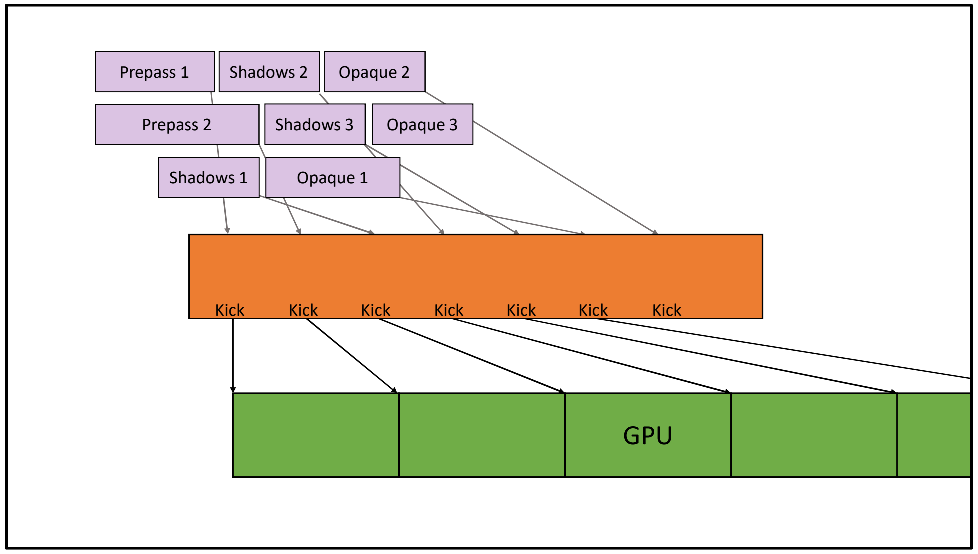
Avoiding bubbles

- Parallel command buffer generation
  Tuning draw list splits
  Carefully scheduling jobs
  Avoiding contention
- Reducing draw call count
  CPU culling, instancing, multidraw
- Reducing draw call overhead
  Material sorting, bindless

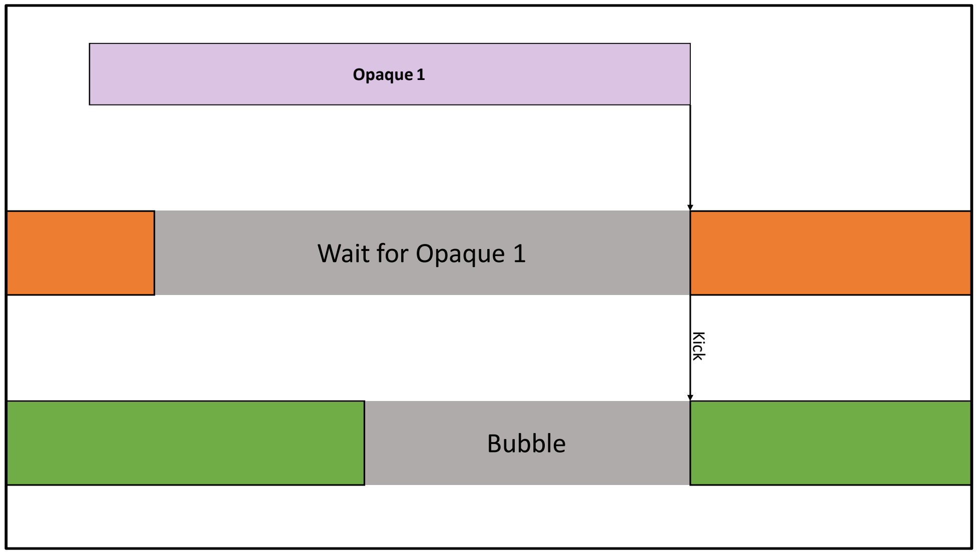
Much effort over multiple generations has gone into making sure command buffer recording finishes as fast as possible.

To name a few techniques:
The most important is running wide over all available cores instead of recording on a single thread. We do this by recording the individually kicked segments as separate jobs in our job system. To get these jobs to run as fast as possible, special care has to be taken to tune the granularity that the frame is split, and to reduce contention between jobs over shared resources.

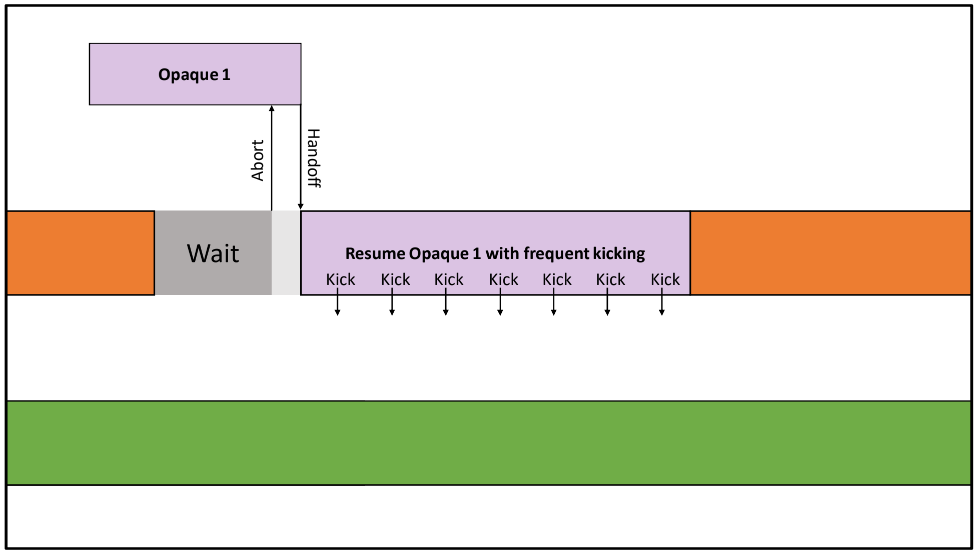
Other techniques to optimize command buffer generation include reducing draw calls counts through instancing and a variety of CPU culling techniques, and reducing the overhead of draw calls by minimizing state changes through material sorting and bindless.



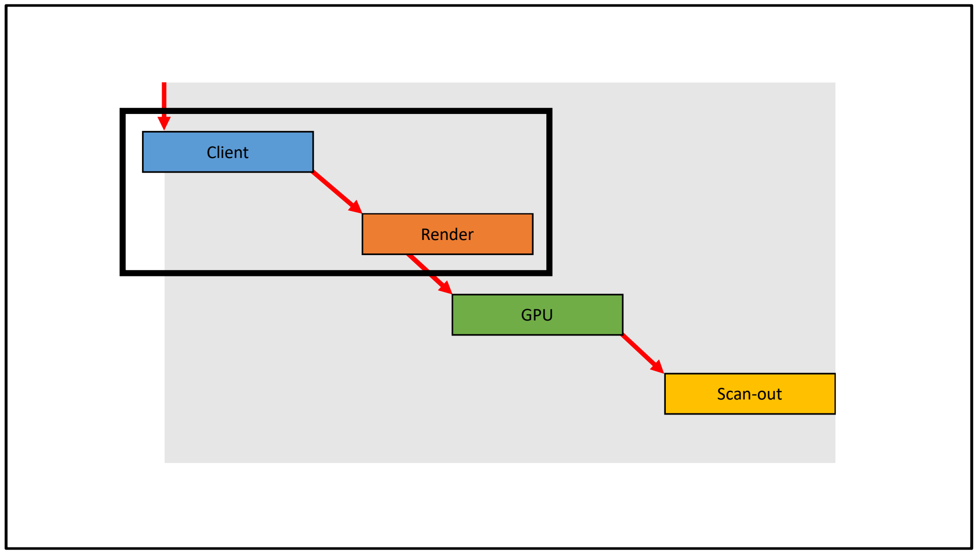
Now we have super fast command buffer generation that runs wide over multiple cores.
One thread picks up the finished command buffers and then kicks them to the GPU, and in the ideal case, the GPU is running slower than the CPU, so it stays solid and fed for the whole frame.



Unfortunately, ideal case still doesn't always happen. Maybe one of the draw jobs started late, or maybe it got kicked off a core by a system thread, or maybe it just had too much work to do.
The submission thread has to wait for the job to finish before it can kick the command buffer, and if that wait is too long, the GPU goes idle and bubbles.

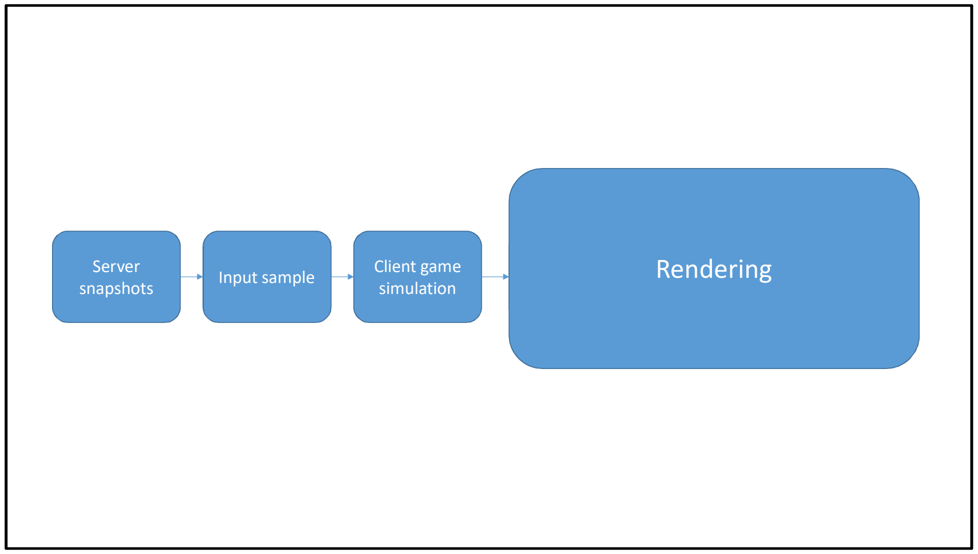


But there’s a cool thing that happens in one code base to avoid bubbles.
The submission thread’s wait for the draw job has a timeout, and if the job doesn’t finish in time, an abort gets signaled telling the job to stop. The job periodically checks for the signal, and if it detects an abort, it stops iterating and ties off its command buffer. Then the submission thread takes over the job and performs all of the work that the job would have done. But this time it kicks more than once, instead just once at the very end of the job.
By kicking more frequently, the GPU gets spoon-fed bits of work a little sooner than it would have had the submission thread waited for the job to finish. This comes at the cost of extra CPU and CP overhead from more frequent kicking, but it potentially reduces the impact of bubbles or avoids them altogether.



We’ve gone over the CPU to GPU relationship: they’re allowed to overlap, which can significantly reduce latency. We’ve tried very hard to speed up the CPU workload to avoid GPU bubbles, but there’s still a chance for bubbles.

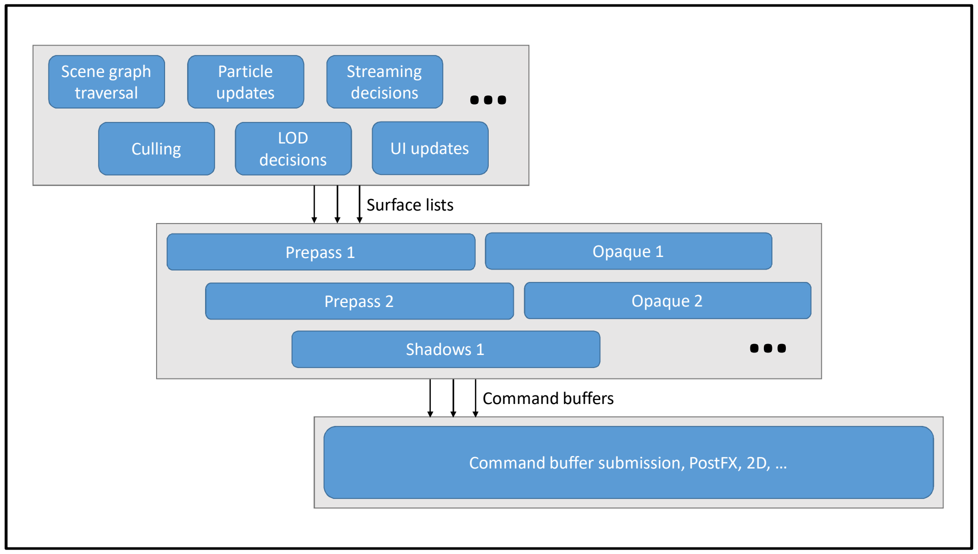
Let’s go up to the final link in the latency chain. I oversimplified by treating the CPU frame as a single monolithic block of work. There can be sources of slop between threads on the CPU. To look for those sources, I’ll introduce the client frame and render frame split.



A single frame of CPU work involves the interaction of many systems, but here's a very simplified summary.

First we get an authoritative game state from the server to update the client game state.
Then we sample for input, and factor that input into client-side game simulation.
Finally we do all the work necessary for rendering the result of the game simulation

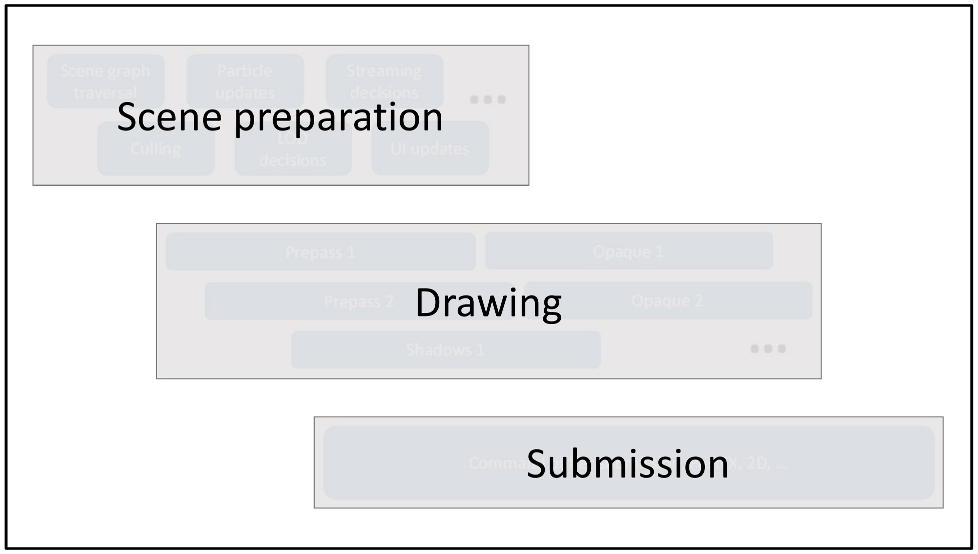
Let's zoom in on the Rendering part.



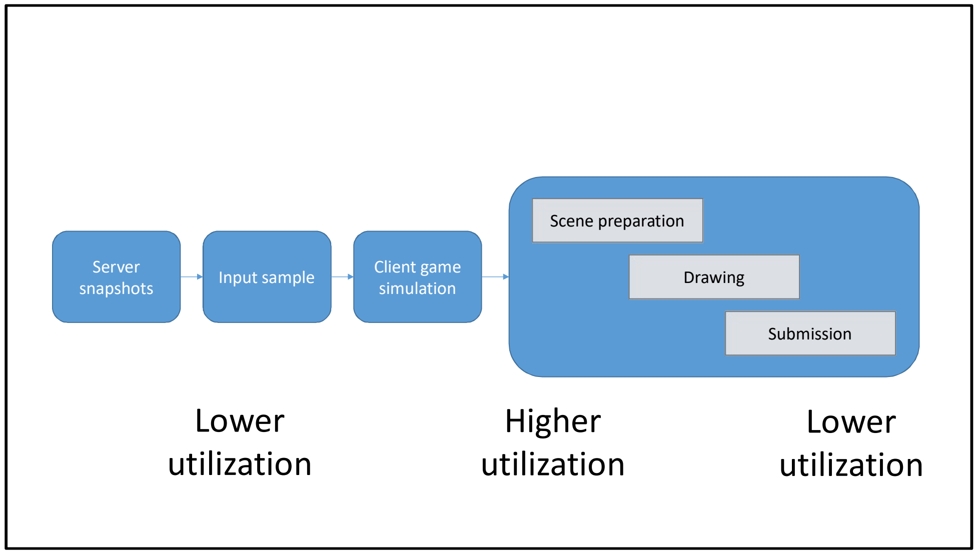
Rendering is split into many little jobs. This includes traversing the scene graph, culling, picking LODs for models, and so on.

The final output of all of these jobs is a set of visible surface lists. Jobs are then assigned to iterating each individual list, and these are the jobs that generate command buffer and associated rendering data.

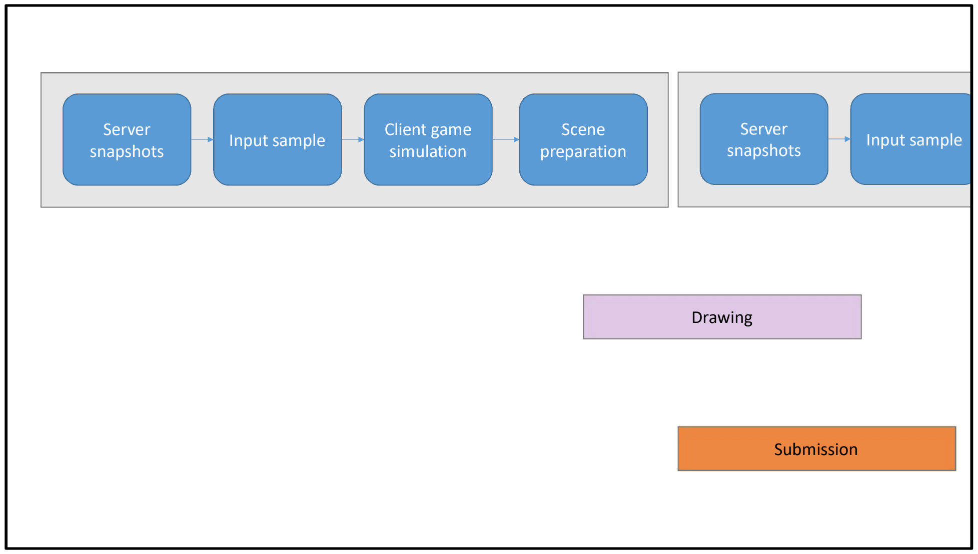
Finally one job collects all the completed command segments and kicks them to the GPU.
This job also records commands not included in the surface lists: for example decompressing surfaces, PostFX, and 2D drawing.



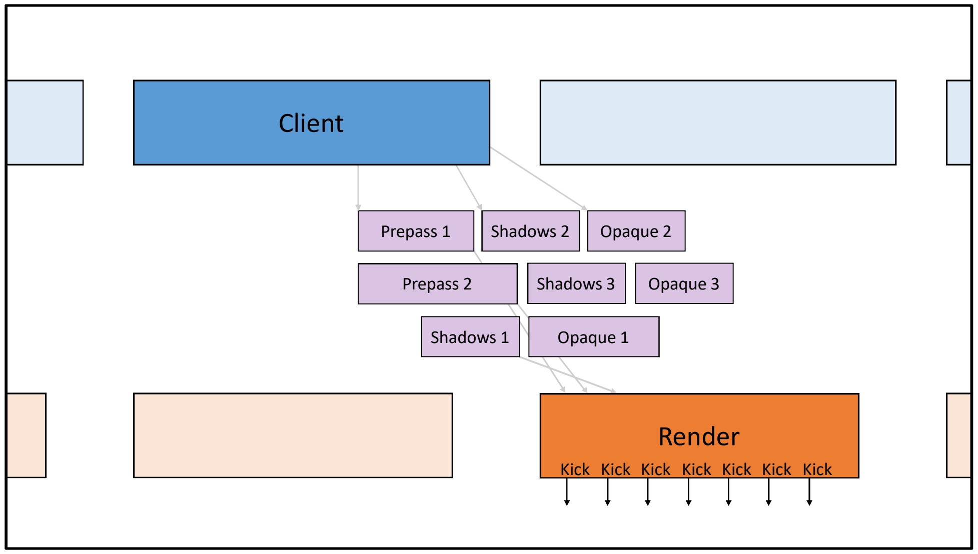
Conceptually, all of this work can be categorized into three groups: Scene preparation, Drawing, and GPU submission.



All of the work over a frame is heavily multithreaded and runs as wide as possible. On average, though, scene preparation and drawing are the parts that benefit the most from running wide. The beginning and ends of the frame achieve lower utilization over the available cores, leaving more idle CPU cycles on the table.



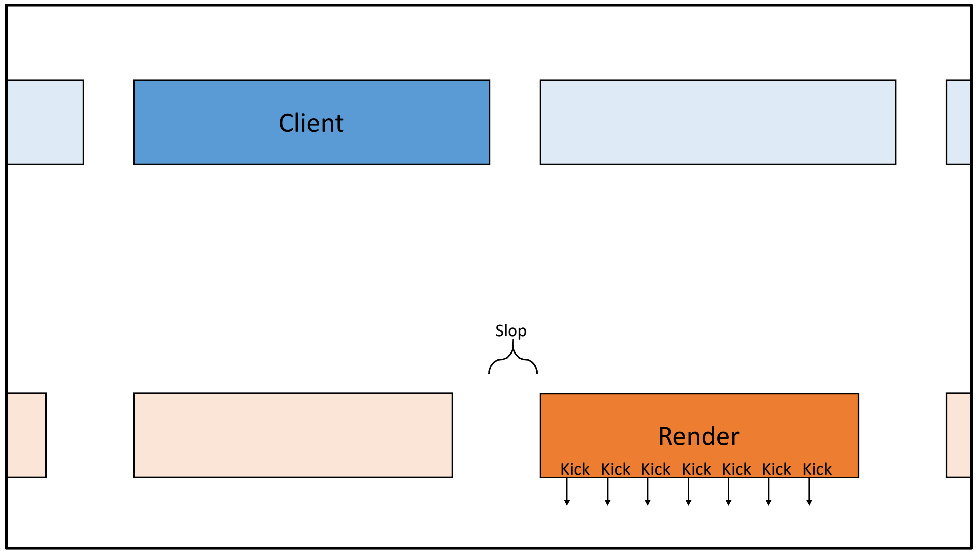
To keep the CPU cores more consistently saturated, drawing and submission work are split out. Then after this frame's scene preparation, the next frame is allowed to begin immediately. That way the beginning of the next frame overlaps with Drawing and Submission for this frame. This yields a pretty big speedup.



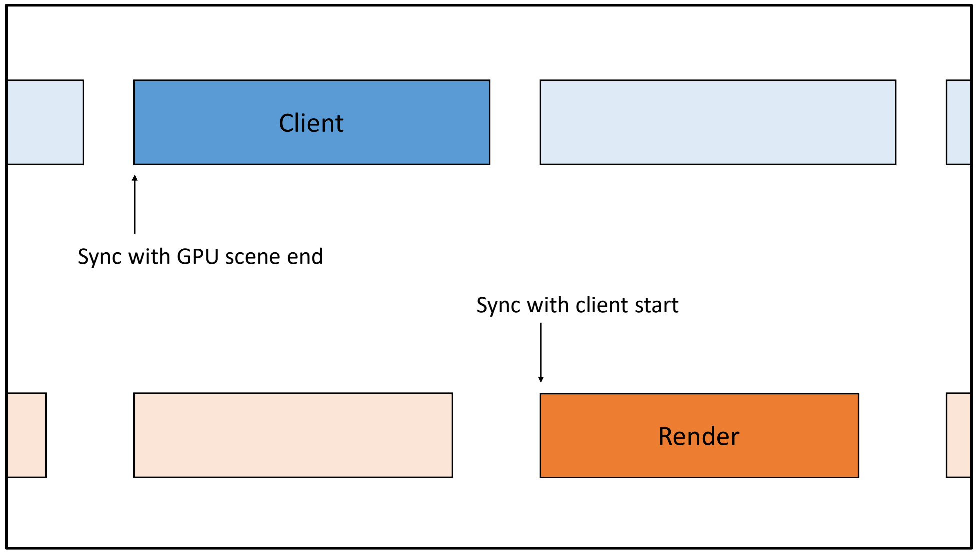
Conceptually, everything up through scene preparation can be called “the client frame” with a dedicated client thread coordinating all of the associated jobs.

A second render thread performs all of the command buffer submission, PostFX, 2D drawing, etc, and comprises a second parallel “render frame.”

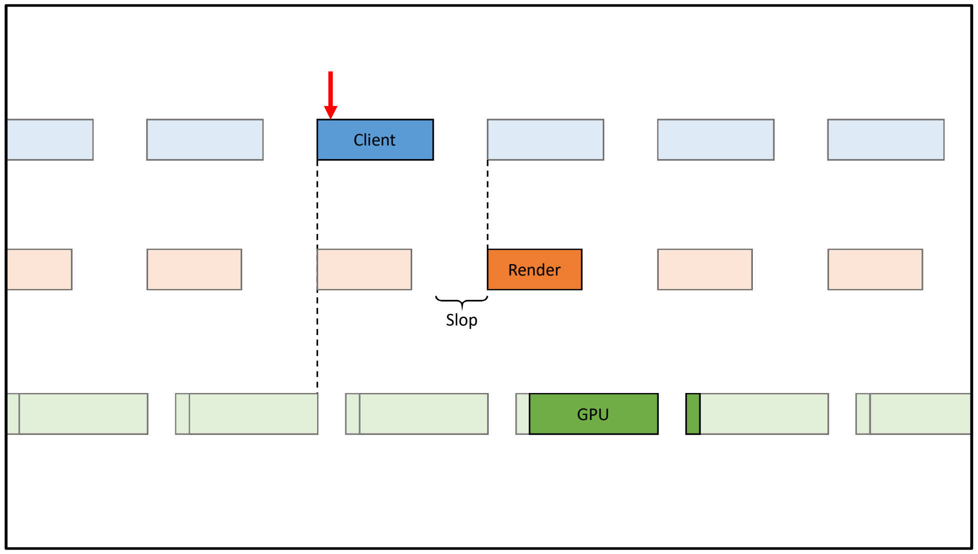
In between the two timelines are all of the drawing jobs that generate command buffer. The client launches these jobs, and the render thread waits for them and kicks their result.



Just like with the other producer consumer pairs, slop can accumulate between the client and render frames.

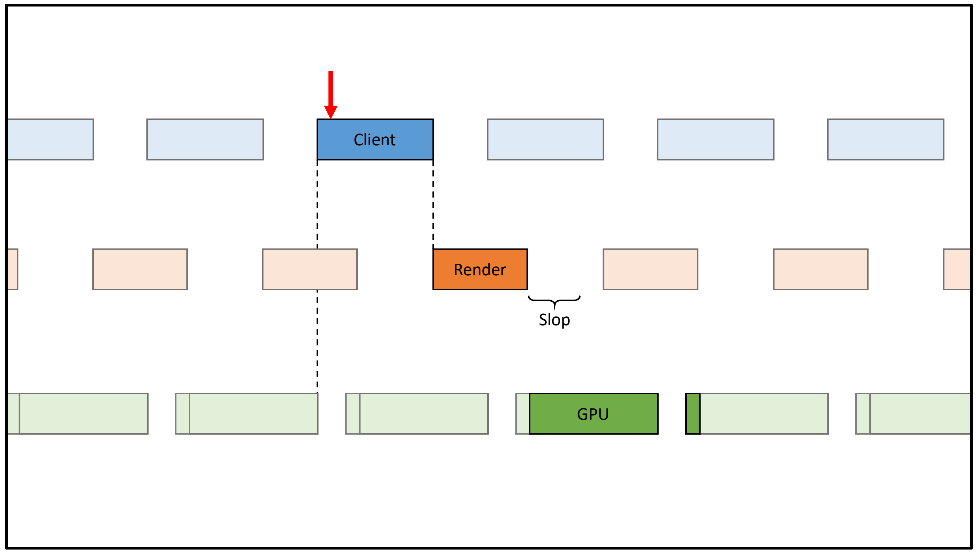


But let’s focus a bit on synchronization between these two timelines and how it affects slop.
First, the client has to synchronize with the GPU because it writes into buffers shared with scene rendering.
These resources are double buffered, so before the client can start its frame, it has to wait for the GPU scene from two frames ago to finish.
The client and render threads also share this data. Originally, the render thread wasn’t allowed to start a frame until the beginning of the next client frame.



Zooming out, on the bottom is the GPU timeline
The beginning of the client frame synchronizes with the end of GPU scene rendering from two frames ago. The beginning of the render frame then synchronizes with the beginning of the next client frame.

This creates some slop: between the end of the client frame and the beginning of the render frame. Why does this slop actually exist, if the pipeline is not bound on the render frame?

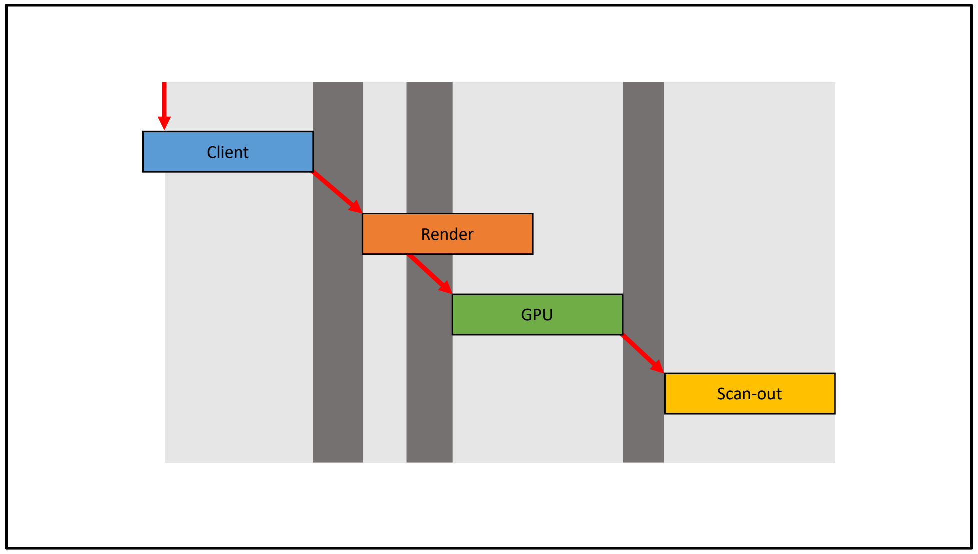


There's no correctness reason why the render frame can't start here instead: right at the end of the current client frame. Making this change has no effect on total latency: the total slop across the frame stays the same, it just moves from the left to the right of the render frame.

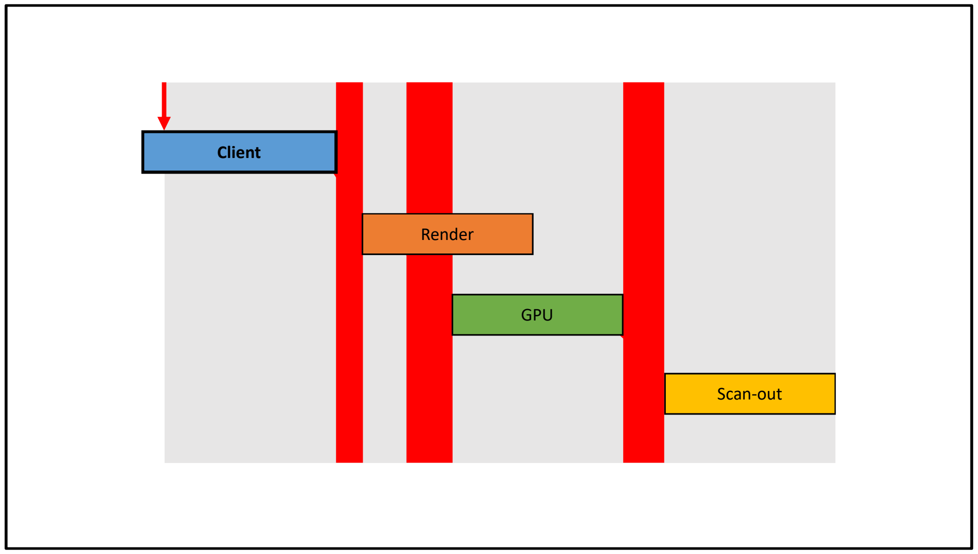
1. Maximize slop in the general case

2. Move slop later in the frame

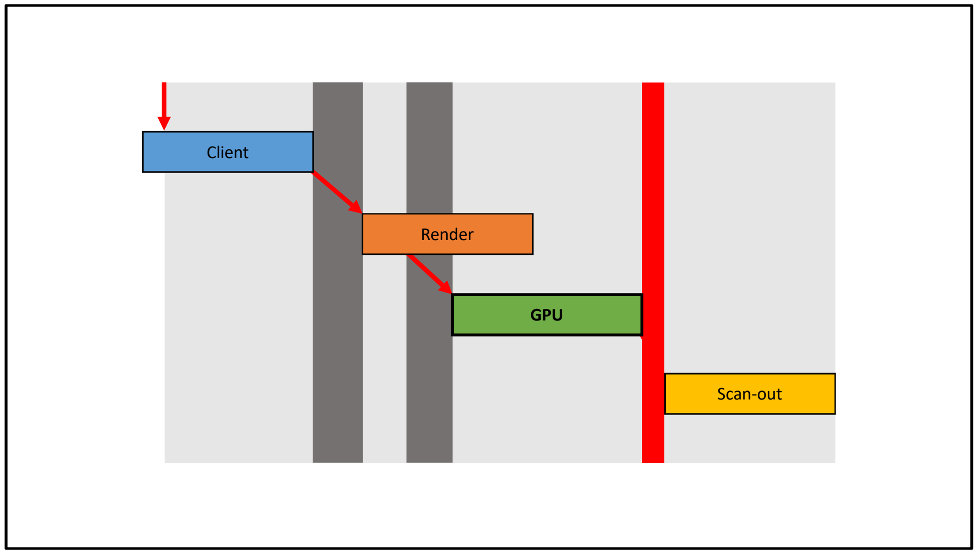
This illustrates a second goal, after maximizing slop. For safety, it’s a good idea to move slop forward in time as much as possible.



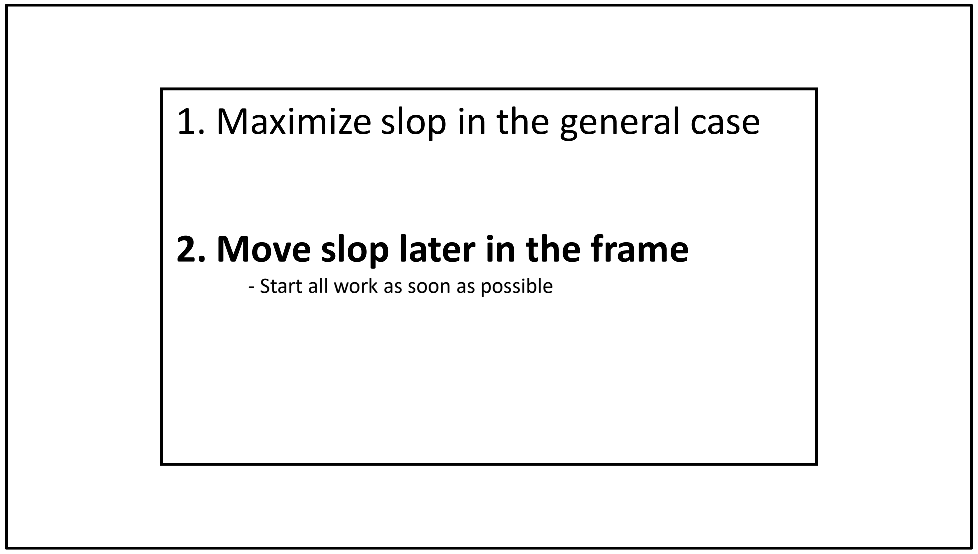
Remember that slop is what protects against spikes. But not all slop in the frame protects equally against all possible spikes.



For example if the client frame spikes, all sources of slop later in the frame are available to absorb the spike and avoid a dropped frame.



But if the GPU spikes, only the slop after the GPU frame can absorb the spike.

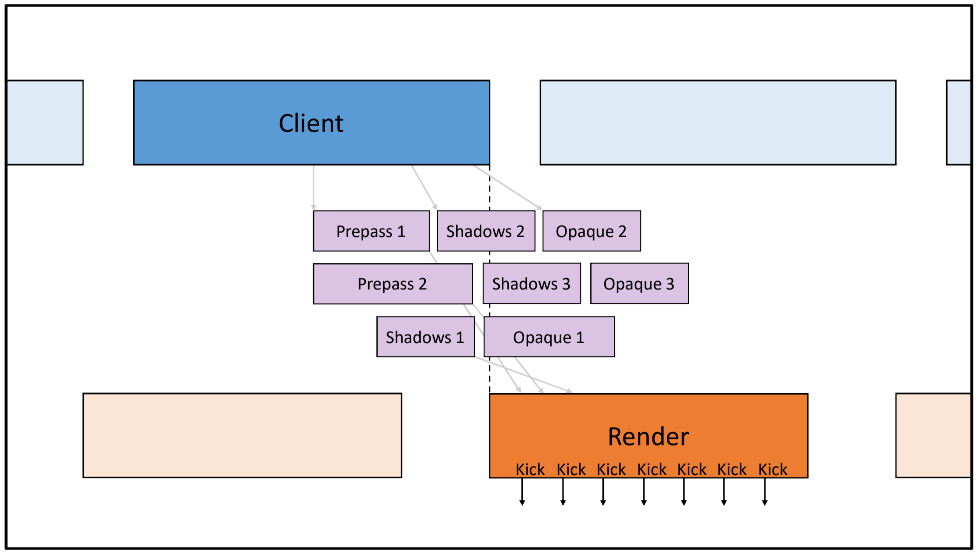


1. Maximize slop in the general case

2. Move slop later in the frame

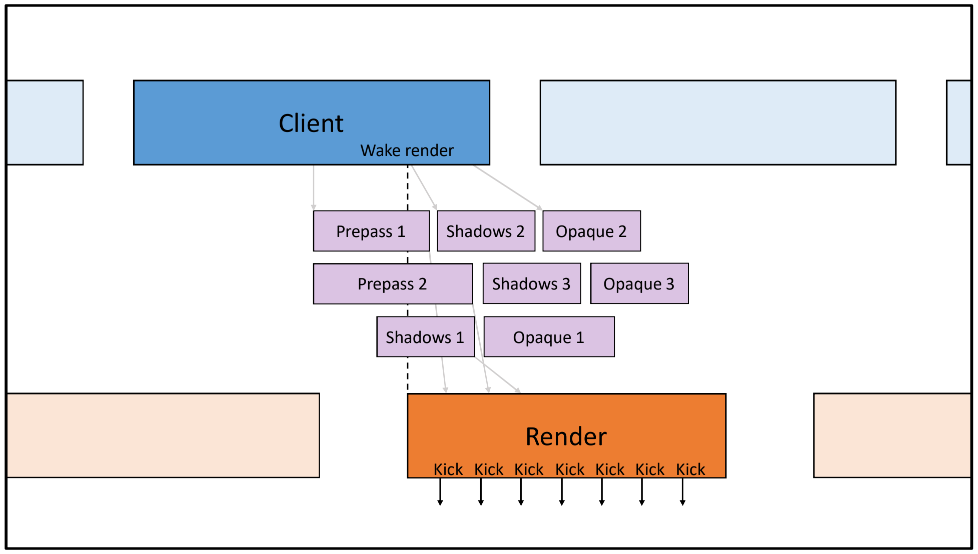
- Start all work as soon as possible

Because slop later in the frame can absorb any work spikes before it, it's always better to keep slop as late in the frame as possible. The way to accomplish this is to make sure that synchronization lets work start as soon as its legal to do so: for example, letting the render frame start at the end of the client frame, instead of at the beginning of the next client frame.



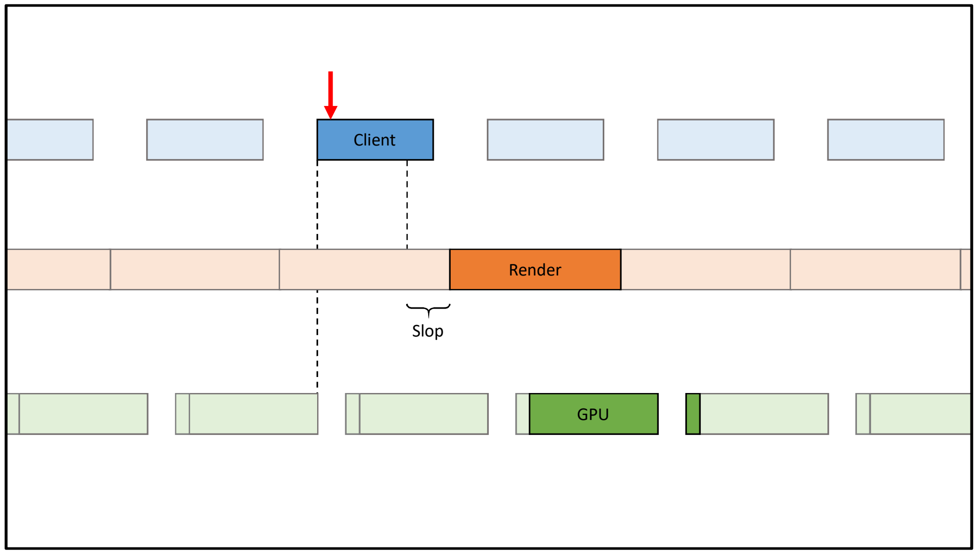
Synchronization has been changed to allow the render frame to start immediately after the client frame finishes.

Now remember why the render and client frames weren't allowed to run concurrently in the first place: because the client writes into buffers that are read by the render thread.

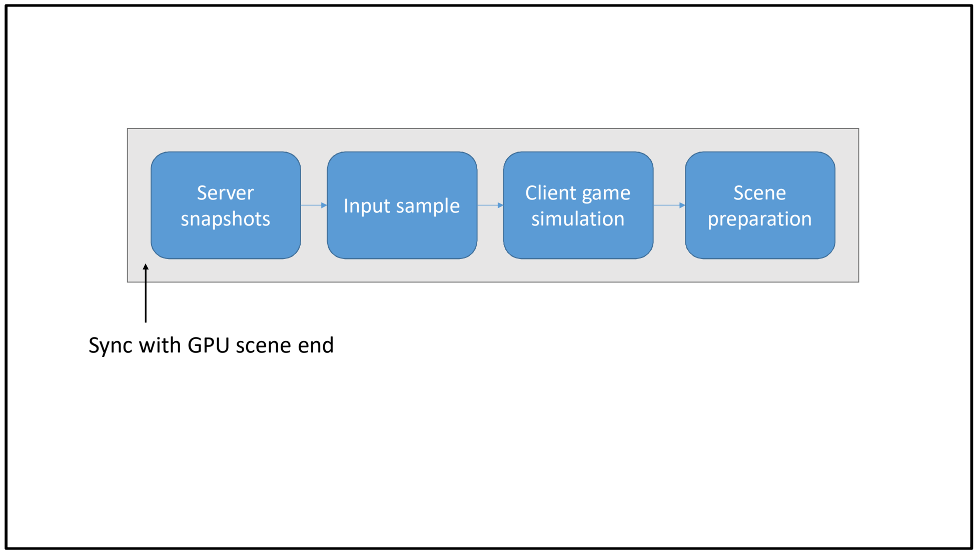


In fact, the render frame can start even earlier, allowing it to begin work before the end of the client frame.
This overlap is implemented in one code branch. It doesn’t yield as dramatic latency savings as the CPU-GPU overlap, but it can still claw back a few milliseconds.

This overlap is achieved by splitting the shared data between the client and render and minimizing dependencies between splits. The extent of overlap depends on how far back in the client frame dependencies the data split can be enforced.

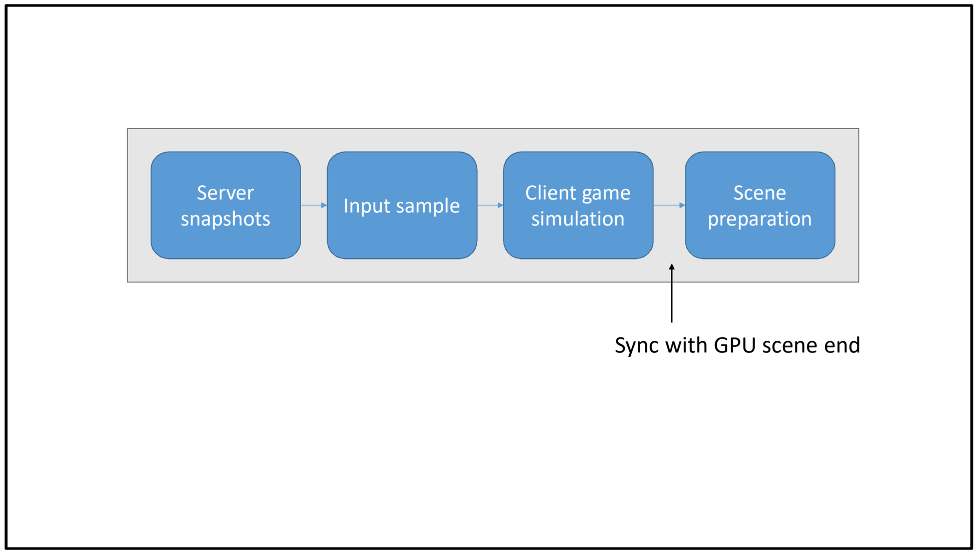


The definition of slop between the client and render frames needs to be altered to account for this overlap: now it's the delay between waking the render thread earlier in the client and the render frame actually starting.

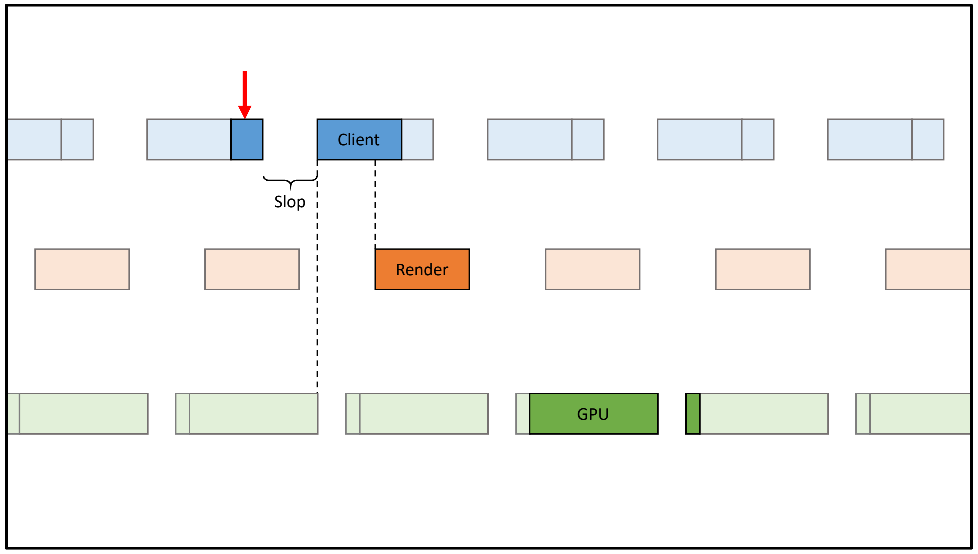


One last thing about synchronization. Remember that the client can't start until the GPU finishes the scene from 2 frames ago. This protects access to double buffered data shared between the CPU and GPU.

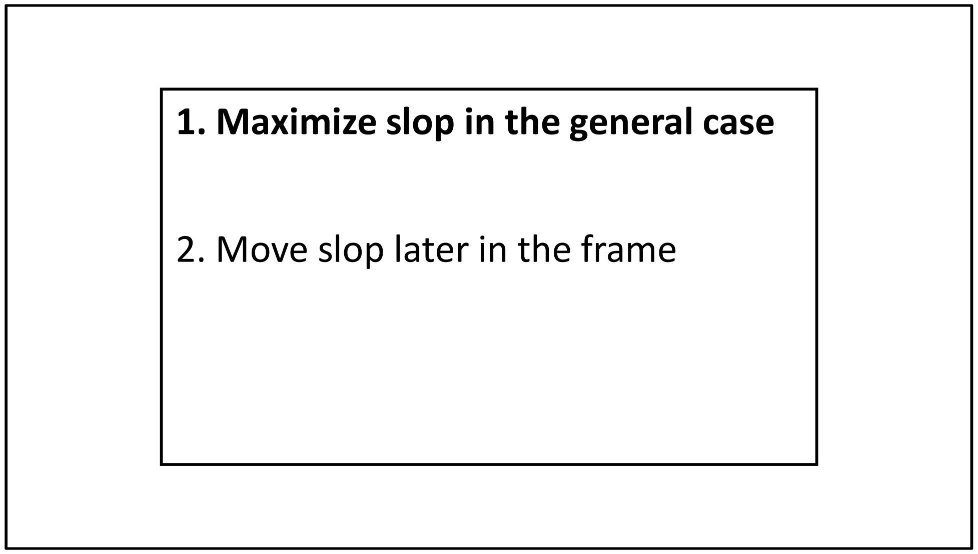
However, those buffers are only ever written to by the CPU during rendering related operations: scene preparation, drawing, and submission. All the client simulation work in the first half of the frame doesn't touch GPU visible buffers.



That means the wait can be moved later in the frame, right before writing into any of the shared buffers.



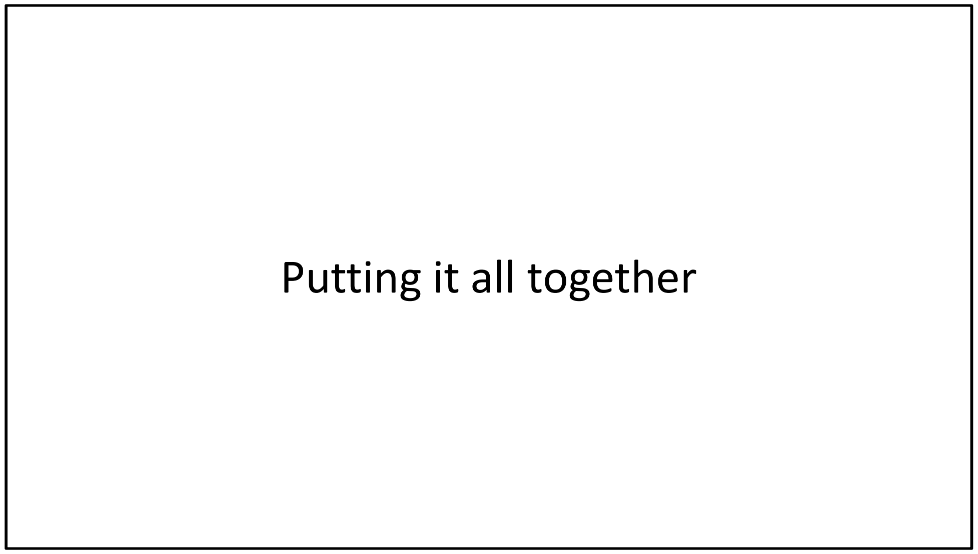
With this change, the client frame gets split up into two parts: one segment before syncing with the GPU, and one segment after syncing with the GPU.
The GPU wait is now after the input sample, so this introduces a new slop in the latency path.



1. Maximize slop in the general case

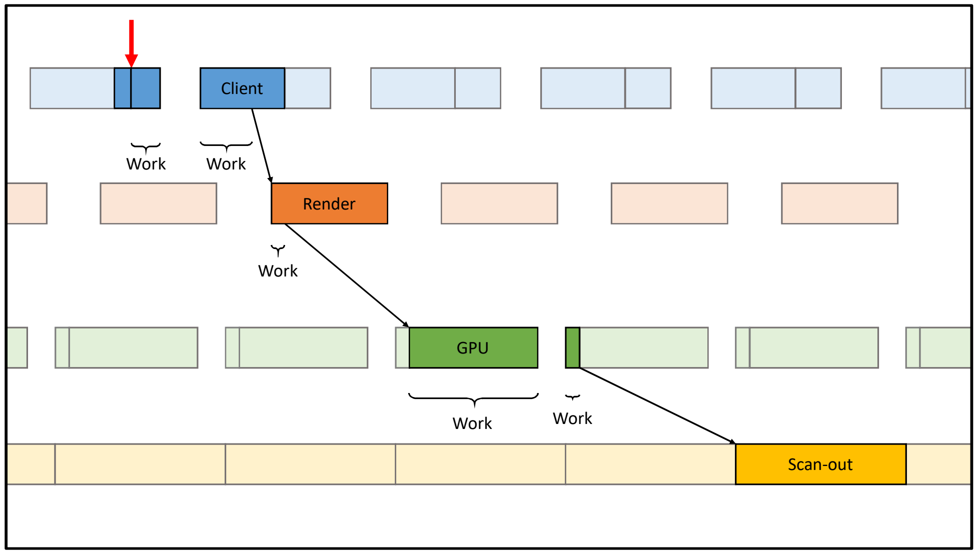
2. Move slop later in the frame

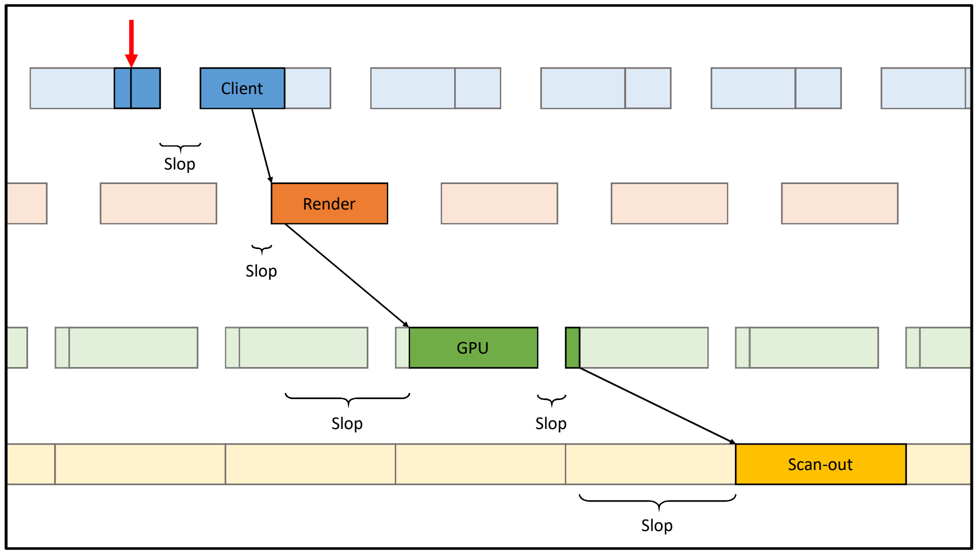
This is another example of maximizing slop in the general case for safety.



Putting it all together

Now we’ve looked at all the major timelines in the engine and identified all the segments of work and slop in the latency path.


Client
Work
Work
Render
Work
GPU
Work
Work
Scan-out

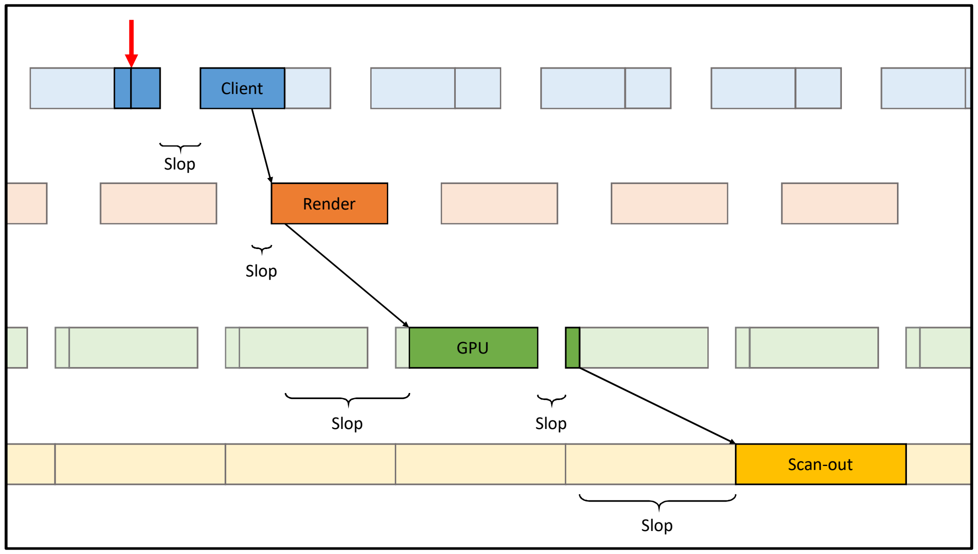

Client
Slop
Render
Slop
GPU
Slop
Slop
Scan-out
Slop

1. Maximize slop in the general case

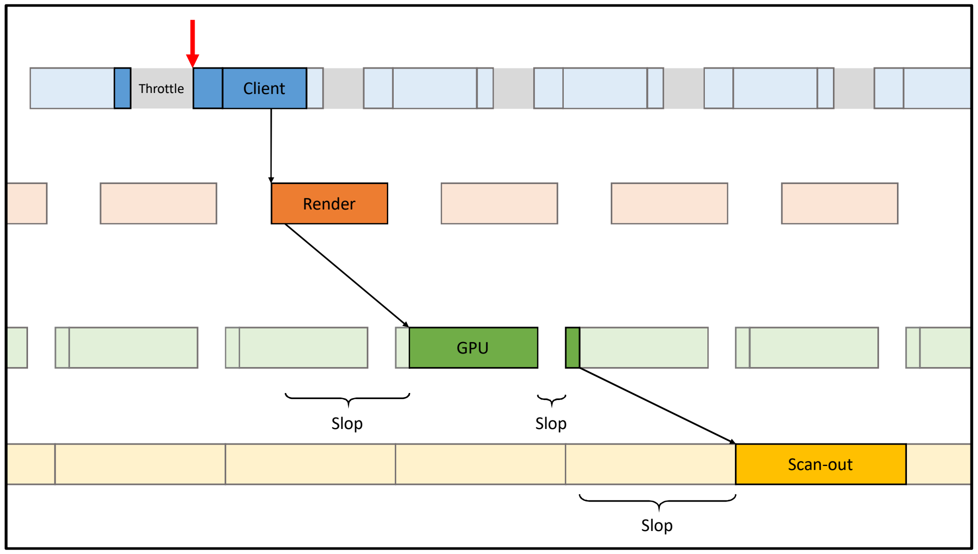
2. Move slop later in the frame

3. Throttle slop in the safe case

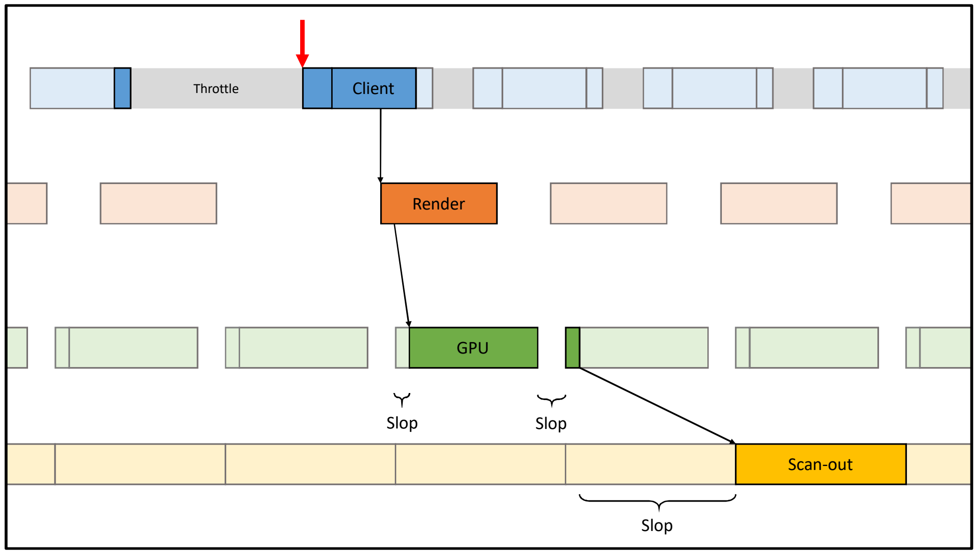
Now let’s incorporate all of this into our throttle implementation



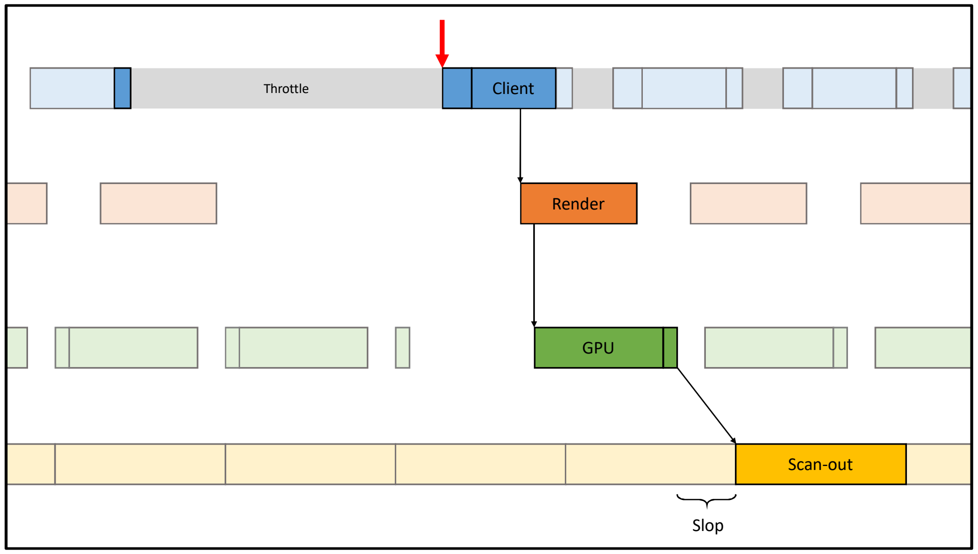
We’re introducing the throttle right before the input sample, so that the sample and subsequent work are delayed and shift to the right.



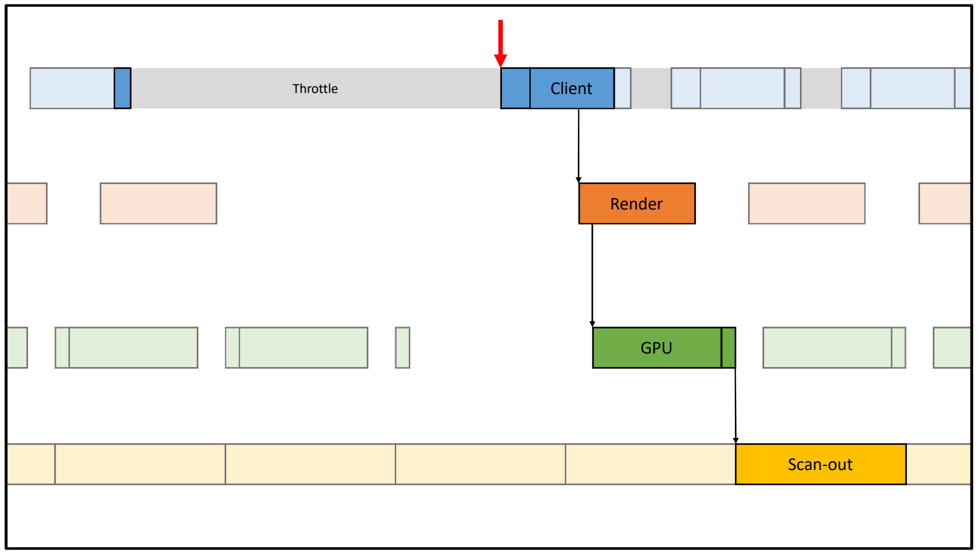
Notice how slop gets squeezed out starting from the left, but the total work stays the same.



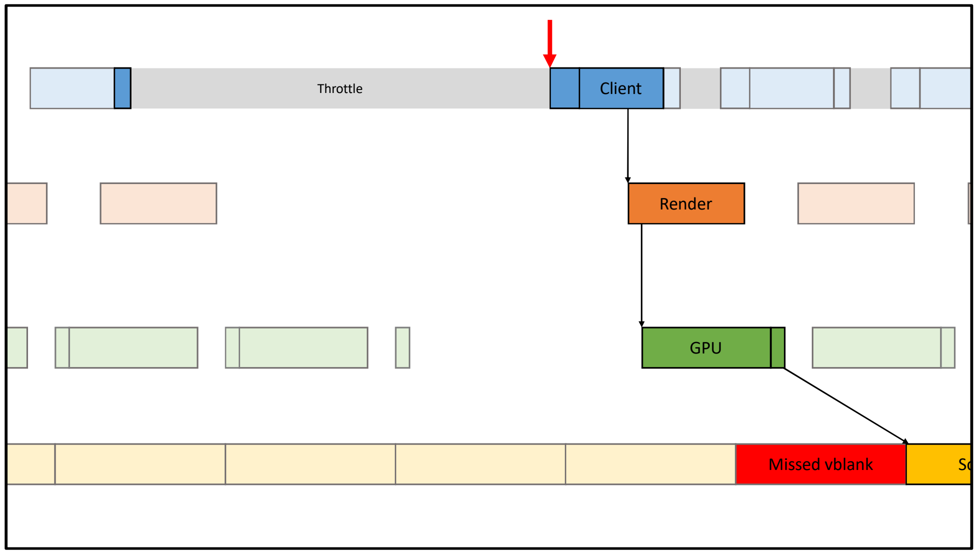
Throttle
Client
Render
GPU
Slop
Slop
Scan-out
Slop



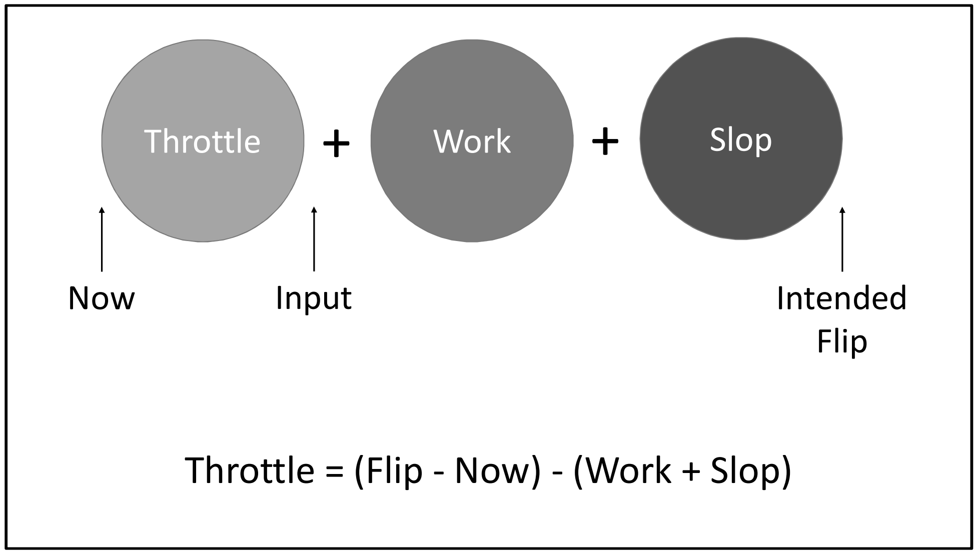
The throttle is long enough now that there's no longer any slop between the client, render, and GPU. The only remaining slop is between the GPU and Scan out.



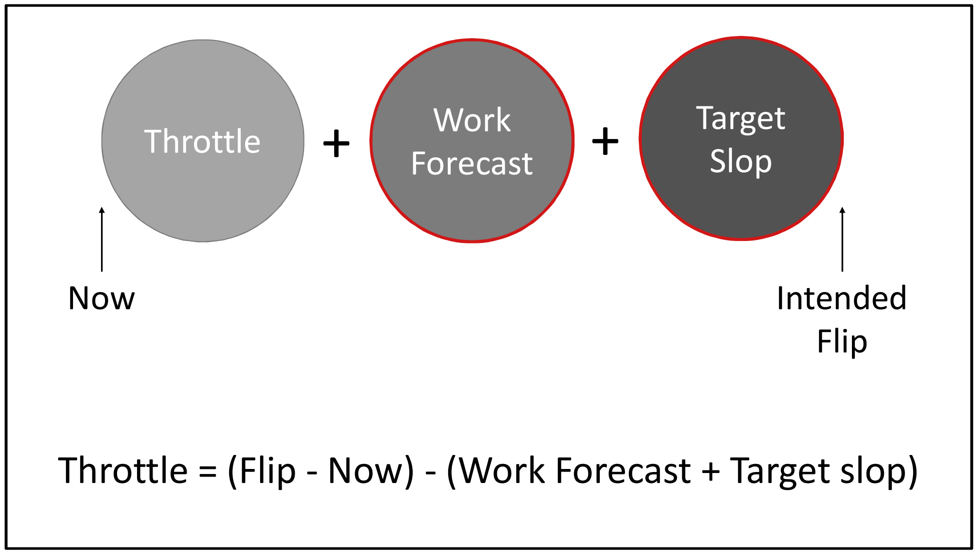
This is the maximum throttle we can introduce, leaving zero slop in the frame.



If we delay just a bit further, the end of the GPU work gets pushed past the vblank and miss a frame.

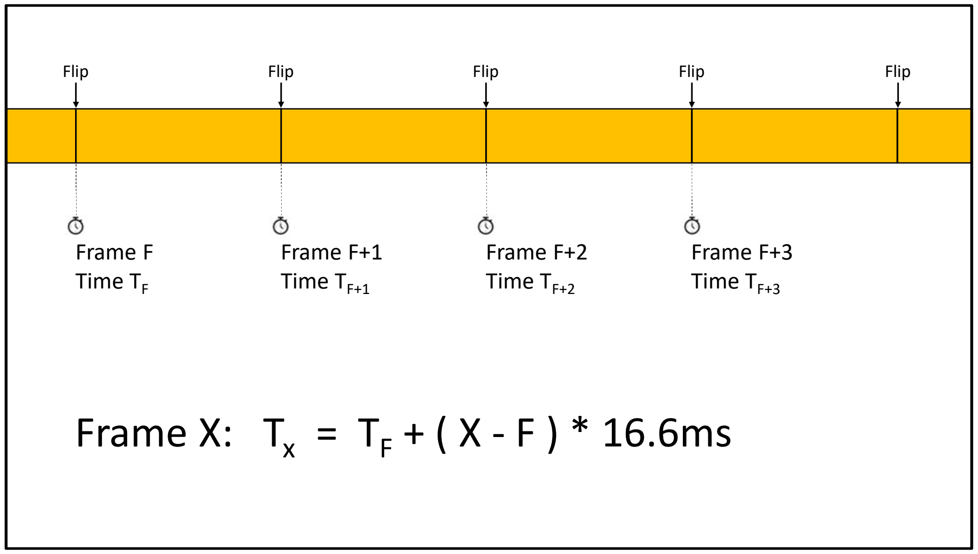


So the question is: how long should the throttle wait?
The throttle is in the middle of the client frame, right before the input sample. The end of the latency duration is during a future vblank, when this frame is intended to be flipped. The time between now and that intended flip is the throttle that we're solving for plus the work and slop for the rest of the frame.
Solving for the throttle, we get the duration between now and the vblank minus the total work and slop. But remember that the throttle is near the beginning of the frame, and before it runs, we have no idea how much work and slop there's going to be.

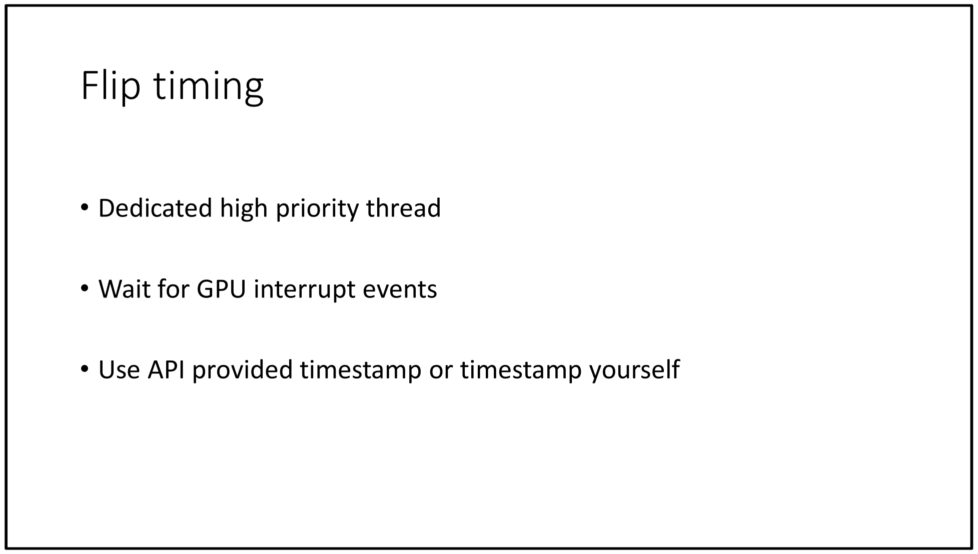


Throttle = (Flip - Now) - (Work Forecast + Target slop)

Instead, we have to estimate how much work there will be this frame, relying on previous frame data. And, we have to decide beforehand on a target slop value that we want to achieve this frame.
The problem then becomes: given a target amount of slop, how long should the throttle sleep?



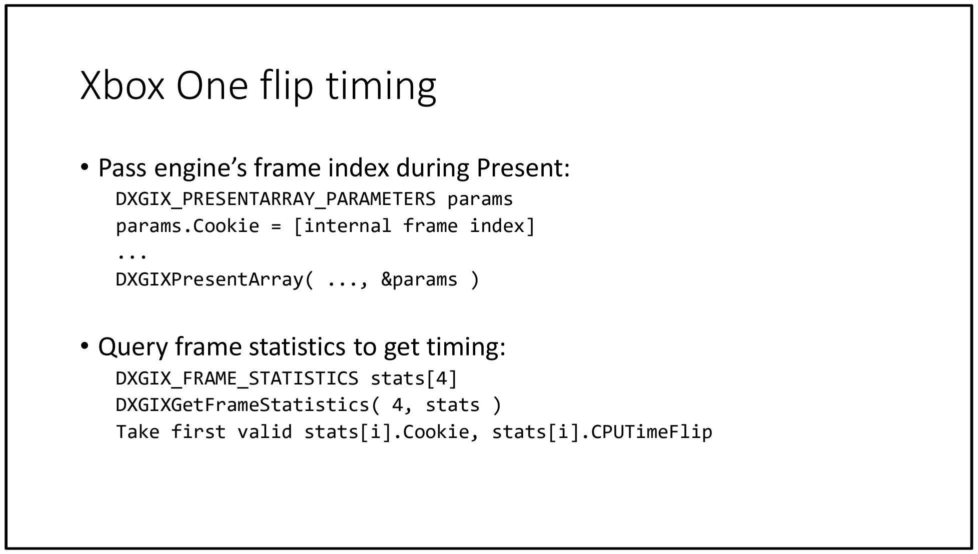
Let’s start by looking at how to find the time of the intended flip. We need to calculate the absolute time of the vblank when we're aiming to flip. Because vblanks come every 16.6ms, the future frame’s vblank can be extrapolated from previous frame timings.



Flip timing

- Dedicated high priority thread
- Wait for GPU interrupt events
- Use API provided timestamp or timestamp yourself

Typically dedicated high priority thread can listen for GPU interrupts, like flips and vblanks. Sometimes the API exposes the display driver's internal timestamps for these events, or you can timestamp them yourself. I haven't noticed a huge difference in timings between these two methods.



Xbox One flip timing

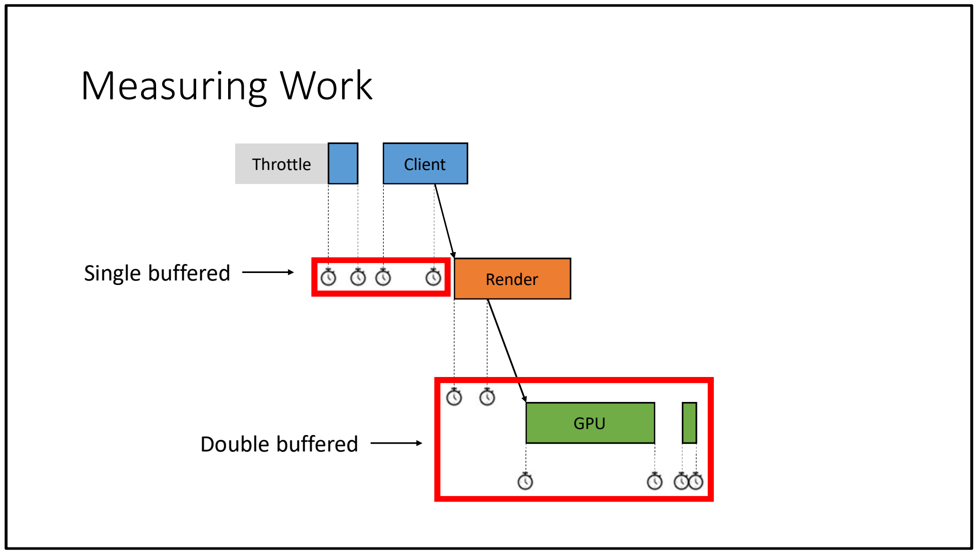
- Pass engine’s frame index during Present:

```
DXGIX_PRESENTARRAY_PARAMETERS params
params.Cookie = [internal frame index]
...
DXGIXPresentArray( ..., &params )
```

- Query frame statistics to get timing:

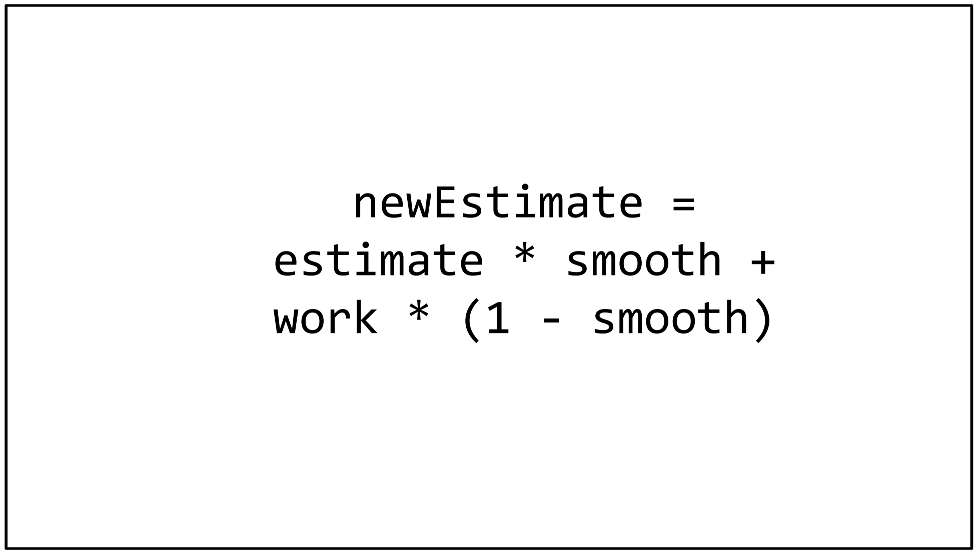
```
DXGIX_FRAME_STATISTICS stats[4]
DXGIXGetFrameStatistics( 4, stats )
Take first valid stats[i].Cookie, stats[i].CPUTimeFlip
```

On Xbox the dedicated thread method will work, but Microsoft also provides a convenient helper function for getting comprehensive frame timings.
The engine’s internal frame index can be to Present as the “Cookie” parameter. Then DXGIXGetFrameStatistics returns an array of timing structs that include vblank, flip, and queue flip timings, with timestamps in both CPU and GPU ticks. Each of these structs also has a Cookie field parameter that can be used to correspond engine frames with timings.



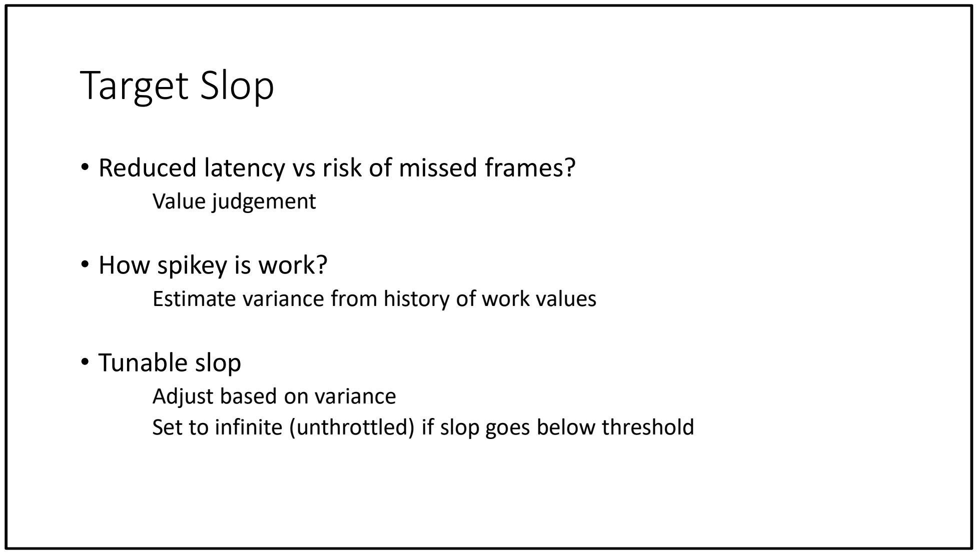
Measuring Work

To forecast how much work there's going to be this frame, timestamps need to be collected for work segments from previous frames. Before the throttle, the most recent measurements need to be collected and summed up to get a total work estimate from the previous frame.
Because the throttle is in the client timeline, the client work measurements don't need to be double buffered and can instead just be read straight from last frame, because they're guaranteed to be complete.
But because the client frame runs concurrently with the render and GPU frame, there's a chance that the work duration will be in flight when timestamps are collected, causing a begin timestamp to be after an end timestamp. Instead the timestamps on non-client timelines can be double-buffered. At least one of each pair of timestamps should be complete: by comparing the begin and end times of both pairs, the most recent valid pair can be determined.



```
newEstimate =
estimate * smooth +
work * (1 - smooth)
```

We use the previous frame work measurements to make a forecast for how much total work there will be this frame. Work is unavoidably noisy: content changes between frames, and the performance of some systems can be spiky or exhibit cyclic behavior. To reduce the variance of when input gets sampled and to lessen the impact of spikes, we exponentially smooth previous work measurements to get a forecast for this frame.



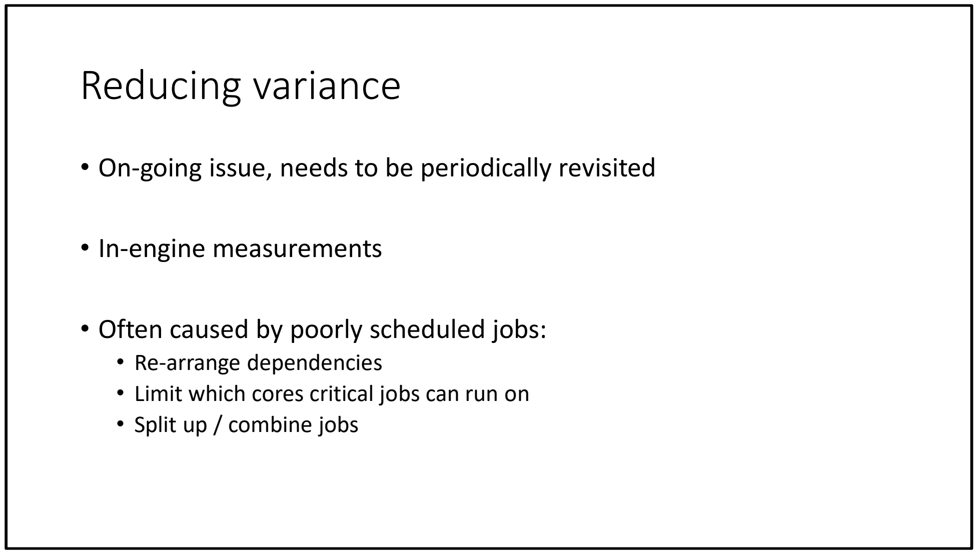
Target Slop

- Reduced latency vs risk of missed frames?
  - Value judgement
- How spikey is work?
  - Estimate variance from history of work values
- Tunable slop
  - Adjust based on variance
  - Set to infinite (unthrottled) if slop goes below threshold

Finally, we need to decide on a target value for the total slop accumulation throughout the frame.
This depends on two questions:

First: How much do we value reduced latency versus the risk of missed vblanks?
Second: How spiky is the amount of Work?

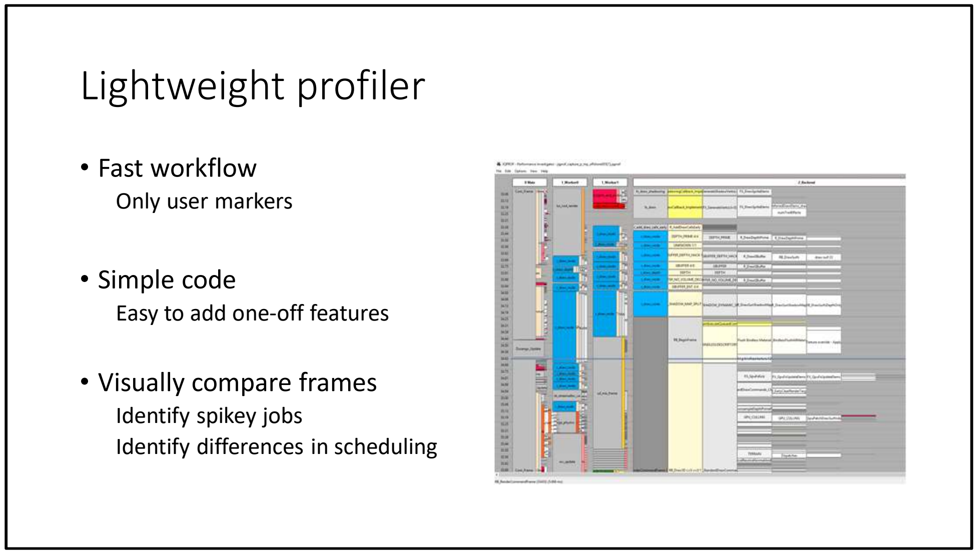
The first question is a value judgement that depends on your level of risk aversion and the genre of game you're working on.
The second question, however, can be quantified by measuring how much forecasted work differs from actual work, for example by estimating variance.
We settled on a tunable slop target dependent on observed variance. If measured slop goes under a safety threshold, we turn throttling off completely to give the biggest chance for the engine to recover without missing frames.



Reducing variance

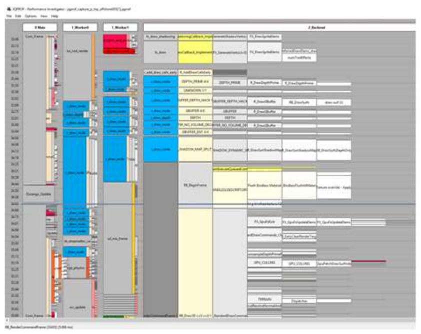
- On-going issue, needs to be periodically revisited
- In-engine measurements
- Often caused by poorly scheduled jobs:
  - Re-arrange dependencies
  - Limit which cores critical jobs can run on
  - Split up / combine jobs

We've now directly tied the problem of input latency with the problem of frame variance. Reducing frame variance can be tricky and requires a team effort.
I don't have any clear-cut directives for how to reduce frame variance in general, but a few practices can help. First, adding in-engine variance measurements for high-level workloads was helpful in identifying where the major sources were coming from.
Many sources of spikiness were due to poor job scheduling, where critical jobs would be picked up early in some frames or too late in others, or would be picked up on cores with more frequent system thread intrusion.



Lightweight profiler

- Fast workflow
  Only user markers
- Simple code
  Easy to add one-off features
- Visually compare frames
  Identify spikey jobs
  Identify differences in scheduling

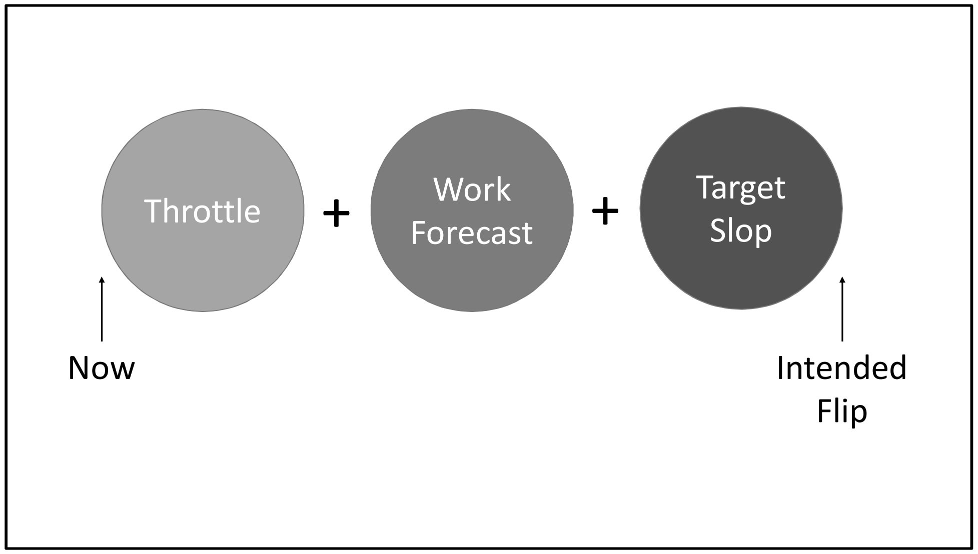


A tool that was invaluable when identifying spikiness problems was a lightweight marker based profiler.

The workflow is super simple and non-invasive: timestamps are dumped to a file and the tool renders simple boxes for markers.
While more advanced commercial or IHV profilers are crucial for deep dives into particular systems, the capture and analyze loop for lightweight tools is much faster.

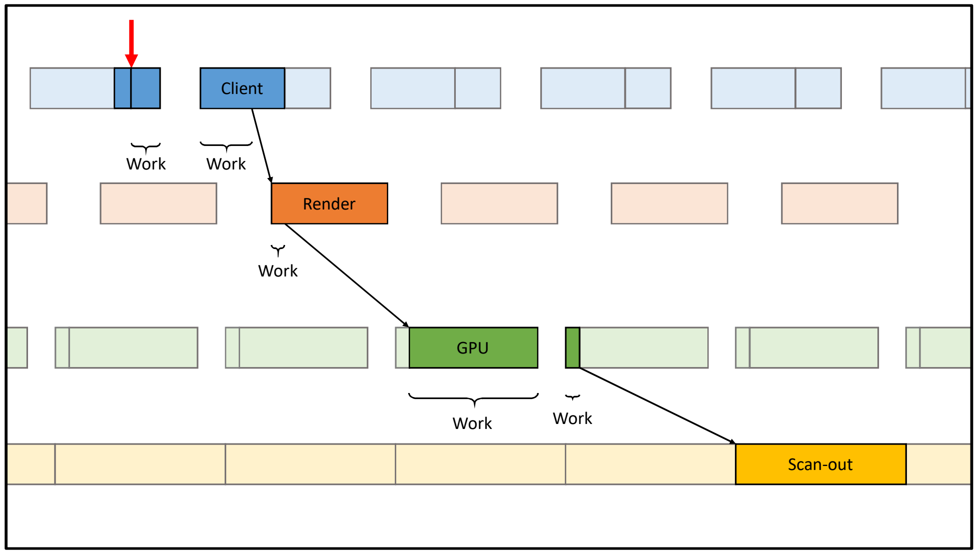
Also the code for the tool is dead simple and can be modified to hack in one-off features as needed.
One of those of one-off features was a mode to quickly scroll between frames while keeping the beginning or end of an arbitrary per-frame marker fixed on screen. That lets you quickly eyeball shape differences between frames and identify which jobs are starting early or late certain frames, and then go up the chain of dependencies to figure out why.

It’s always good to get numbers, especially for validation, but I think visual inspection can be more effective for some kinds of search-and-destroy work.

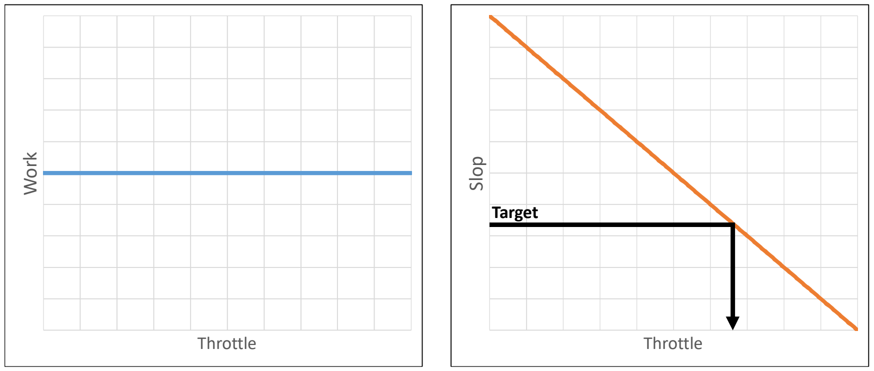


Ok, we’ve now discussed each of the terms needed to calculate the throttle:
The time that the upcoming frame is supposed to be flipped can be figured out by extrapolating from previous flips
This frame’s total work can be forecasted by smoothing measurements from previous frames,
And the amount of slop to allow across this frame can be decided with a heuristic based on observed variance.

Using those terms, we can then calculate how long the Client frame needs to sleep before sampling for input

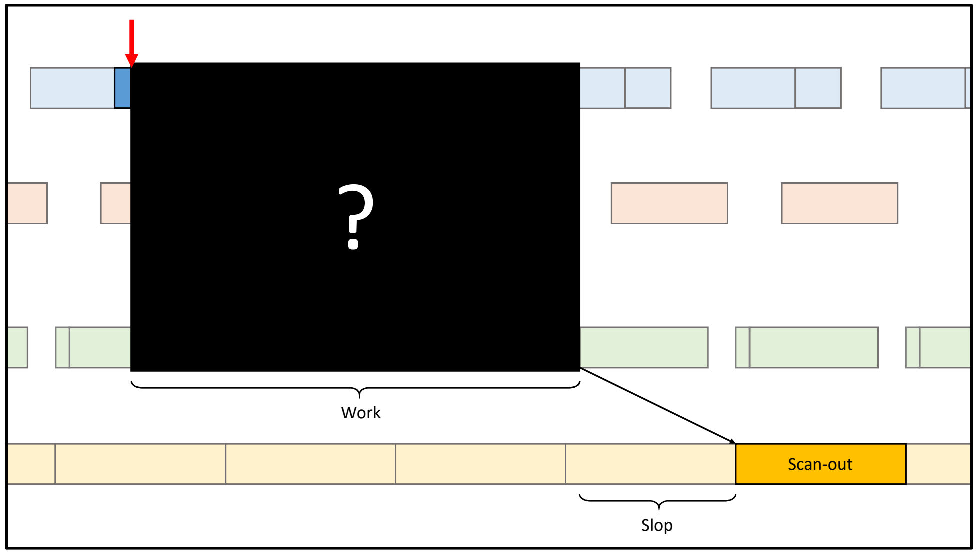


Stepping back, let’s go over why we had to go through the process of identifying every source of work and slop in the engine.

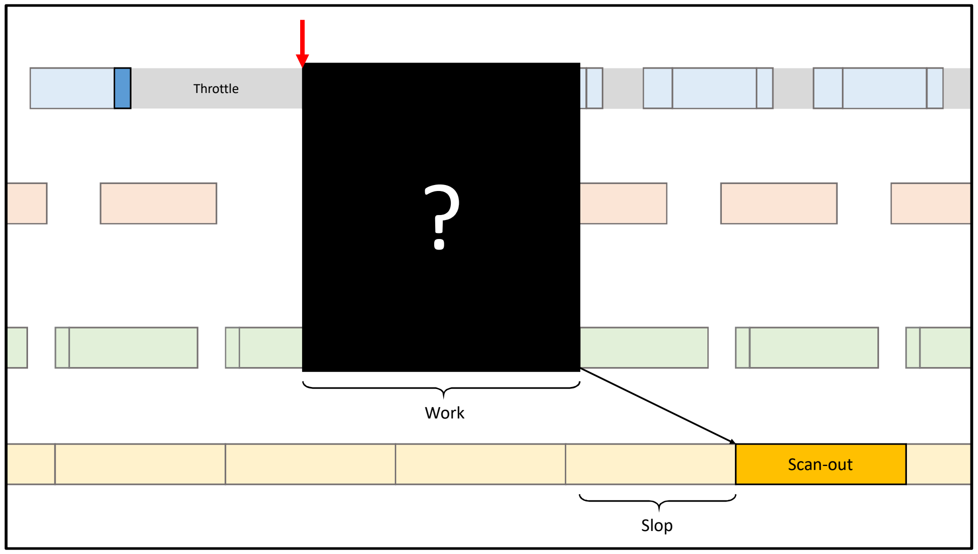


Increase Throttle ⟷ Same Work
Increase Throttle ⟷ Decrease Slop

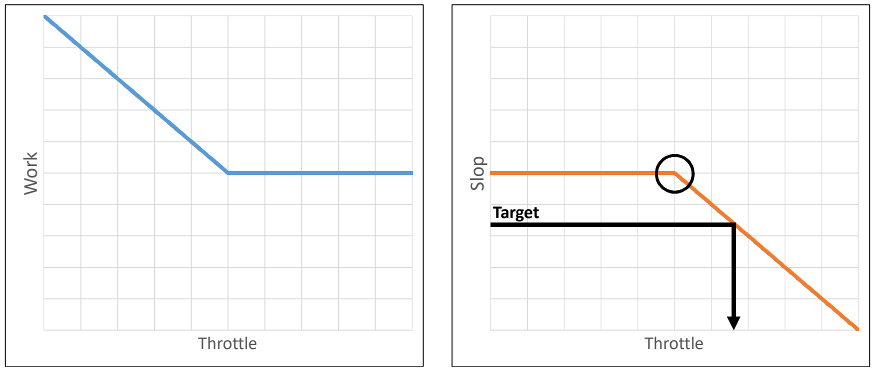
With correct definitions for work and slop, the effect of the throttle on these values becomes very predictable.
This makes calculating an appropriate throttle easy given a target slop. The measured slop then quickly converges to the near the target in one frame.



When I was first looking at latency, I treated the engine as a big black box. I called everything inside the big black box work, and the only slop was the duration from the Queue Flip to the Flip. This ignores all of the extra sources of slop inside the engine, but it makes work measurement a lot easier.
I expected the throttle to still work.



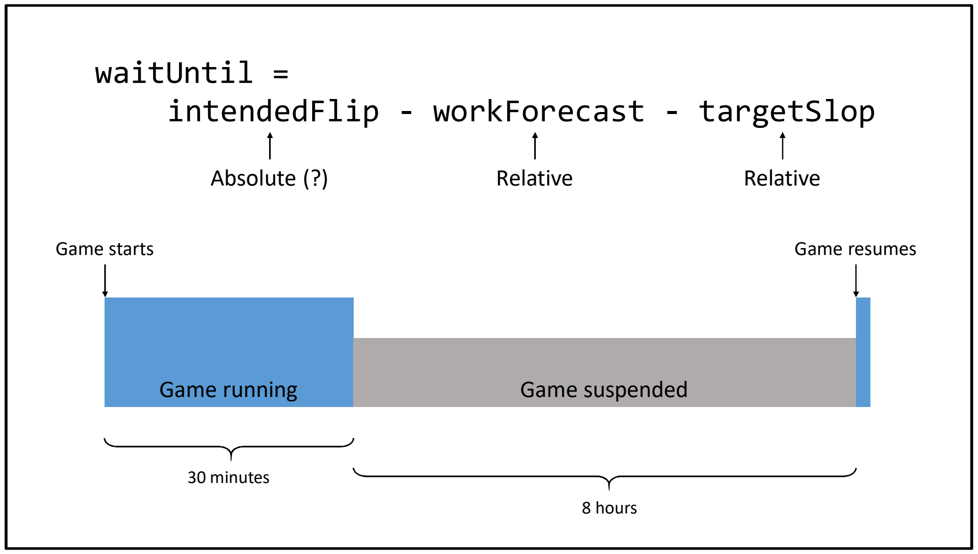
The problem is that throttling is no longer predictable. Some initial throttling squeezes work but doesn’t affect slop.



No longer linear
Requires searching over multiple frames

As you increase the Throttle, work decreases for a little while, then flattens out. Slop stays flat for a little while, then gets squeezed. You lose the linear relationship between throttling and slop that makes it so simple to calculate the proper throttle to achieve a given slop target.
That's because you don't know where this corner lives up front. Instead you have to search the throttle space to find the correct throttle.

Searching instead of solving is doable on static workloads, but convergence to the target slop takes at least two frames. In practice, that corner moves around depending on properties of the workload, making searching less reliable. I found that slop would wander too far from the target, causing either huge amounts of latency or too many missed vblanks. I had to go back and unpack the big black box to arrive at definitions of work and slop that converged more properly.

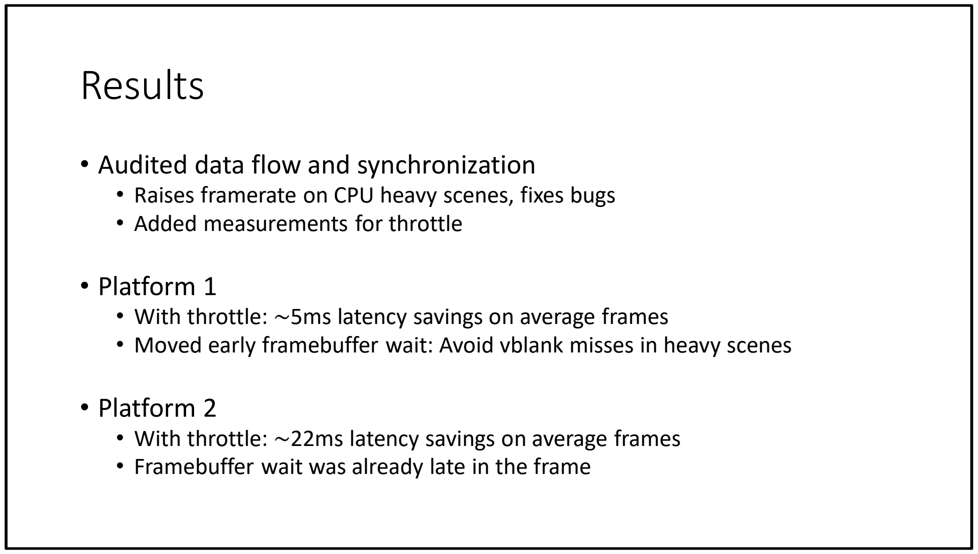


There aren't many systems in our games that dynamically alter behavior based on performance measurements from the running, shipped game. The only other example I can think of is dynamic resolution. Feedback systems like this need to be especially robust to weird inputs and edge cases.

Here's a bug in the throttle that wasn't discovered until quite late:
The throttle subtracts a work forecast and target slop value from the intended flip time. The work forecast and target slop values are relative durations: in the ballpark of multiple milliseconds. I assumed that extrapolated flips were huge, measured since the epoch, and huge minus small should be safe right?
Unfortunately on one platform the flip timings were actually relative to the time that the game was launched, not from an epoch. And that usually didn't matter, because the game doesn't render a frame for a few seconds after launch, so the extrapolated flip timings would be in the range of seconds and higher.

And that was all fine, except when process lifetime management gets involved. Take this scenario: you play the game for 30 minutes, then suspend the game for 8 hours, then resume the game.
The first resumed frame's intended flip would be extrapolated from 8 hours ago, so it would come out to 30 minutes, plus a couple vblanks. The game would have been suspended in the middle of a work duration, so you'd get a work measurement around 8 hours. Even with smoothing, you could end up with a huge work forecast longer than 30 minutes, causing the calculated "wait until" value to wrap around, causing an infinite sleep.

This bug only shows up after suspending the game for longer than it had been originally running, and that's not a case I typically test for.



Results

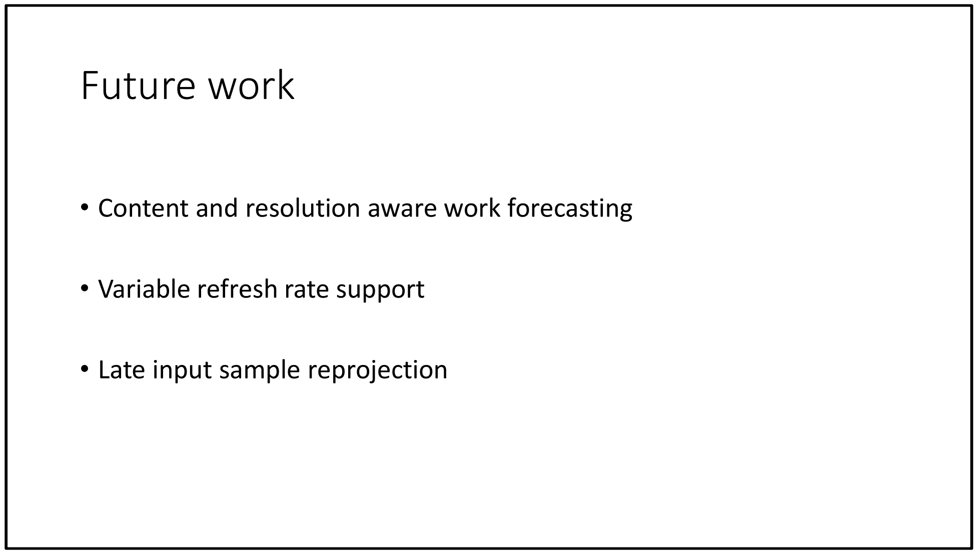
- Audited data flow and synchronization
  - Raises framerate on CPU heavy scenes, fixes bugs
  - Added measurements for throttle
- Platform 1
  - With throttle: ~5ms latency savings on average frames
  - Moved early framebuffer wait: Avoid vblank misses in heavy scenes
- Platform 2
  - With throttle: ~22ms latency savings on average frames
  - Framebuffer wait was already late in the frame

On recent Call of Duty games, the throttle was introduced to unify the measurement and control over the tradeoff between latency and safety.
Before implementing the latency throttle, the engine's data flow had to be audited to identify which segments to measure so that work and slop would behave correctly when throttled. This process uncovered performance issues that impacted frame times in certain CPU bound scenarios, and solved some bugs and rare crashes due to improper synchronization.

On one platform, the throttle clawed back about 5ms of latency on average multiplayer frames. On this platform the wait for framebuffer was originally very early in the frame, causing the slop between the GPU and scan-out to already be tight. Moving this wait later in the frame allowed for higher average framerates when running below target framerate. On typical frames above target, the throttle then brought latency back down to a little lower than previous levels.

On another platform, the wait for framebuffer was already late in the frame. The loosening was not as major, so average framerates when running below target were not dramatically impacted. The slop between the GPU and scan-out was consistently high during typical workloads, and all of this slop allowed the throttle to reduce latency by more dramatic amounts on typical frames.

As described before, this latency tightening decreased the threshold for spikes to cause frame drops. A lot of work has gone into reducing these spikes during gameplay, and that work is still ongoing.



Future work

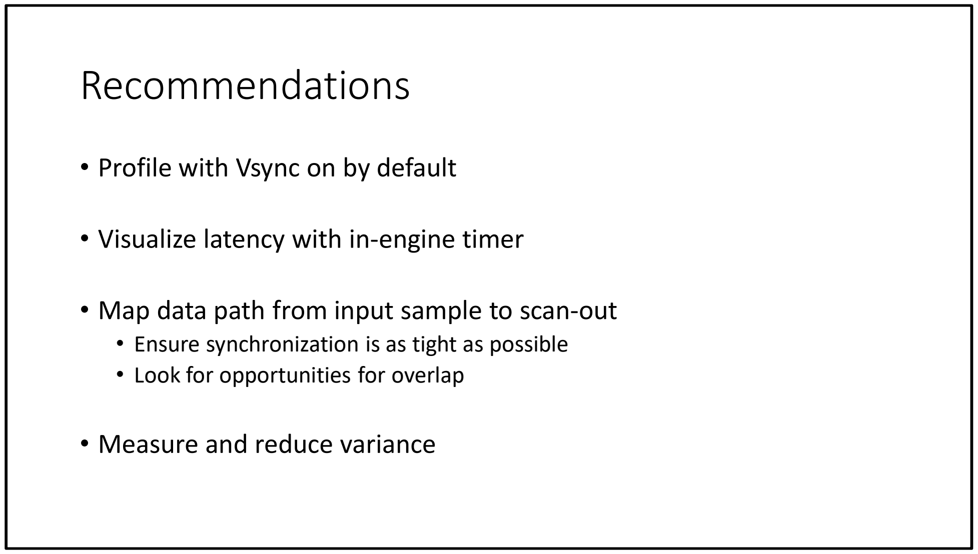
- Content and resolution aware work forecasting
- Variable refresh rate support
- Late input sample reprojection

Moving forward, it might be possible to significantly improve forecasting for work. More intelligent solutions were tried, but currently the shipped estimation takes no account for the content that’s planned in the upcoming frame. A simple improvement would be for the estimate to be aware of dynamic resolution: when the game decides to increase or decrease resolution, the work forecast should account for the slower or faster GPU time. More informed estimation could include marking certain game sections as not requiring tight latency (dramatic single player sections, level transitions, multiplayer lobbies, etc) so the engine could dial back the throttle before encountering heavy or spikey workloads.

The throttle currently doesn’t support variable refresh rate displays. Variable refresh rate is interesting, because if the game is running slower than the maximum refresh rate supported by the display, the game no longer stays bound by the scan-out, and slop no longer exists between the GPU and scan-out. This already significantly reduces latency, but there could still be milliseconds available to throttle. The throttle would need to figure out which timeline is the current bottleneck, estimate the end of that bottlenecked timeline for each frame, and then throttle the input to achieve target slop between now and that estimated end.

It might make sense in the future to look at some of the reprojection techniques from VR. Instead of using the same input sample across the entire duration of a frame, a later input sample could be used for last minute reprojection. This wouldn’t improve the latency of all input, but it could improve the perceived input latency of the camera controls.

User research into the effect of the variance of latency itself would be useful. Instead of only valuing absolute latency values, there could be a tradeoff between consistent latency and low latency that would be more appealing to the player.



Recommendations

- Profile with Vsync on by default
- Visualize latency with in-engine timer
- Map data path from input sample to scan-out
  - Ensure synchronization is as tight as possible
  - Look for opportunities for overlap
- Measure and reduce variance

Even if latency is not a top priority, every game makes the described tradeoffs in one direction or the other, and its valuable to be able to measure and understand those tradeoffs.
Here are some recommendations for looking at latency and performance in games in general.

I would recommend keeping vsync on by default during profiling. I've noticed an organizational intuition that vsync should be kept off during development out of fear that it skews performance measurements. But we ship with vsync on, so it makes sense to figure out whether or not vsync is skewing performance and if so, why. Keeping it on also uncovers vsync related problems sooner.

I would recommend implementing in-engine latency measurements and an on-screen timer. We've always had very detailed on-screen GPU timers and CPU timers, and these are great when identifying specific slowdowns. But for getting a global sense for performance across the frame, I found that an on-screen timer that unified the CPU, GPU, and scan-out timelines to be very useful.

I would recommend periodically auditing all of the buffering and synchronization involved in the engine's creation of one frame. As developers we spend most of our time diving deep into particular systems, assuming that all of the surrounding scaffolding is sound. This scaffolding is often old and can accumulate bugs as new platforms or new APIs are added. While diving into this this code, I would also look for opportunities to tighten mutual exclusion periods and opportunities to overlap work that is currently mutually exclusive due to overly-coarse buffering granularity.

Finally I would recommend thinking about frame time variance as a first class performance metric and periodically looking for ways to reduce it, because it's directly responsible for how far you can safely tighten your game's input latency.

Thank you

Activision Maine

Wade Brainerd

Michael Vance

Infinity Ward

Sledgehammer Games

Treyarch

I’d like to thank Wade Brainerd and Michael Vance at Activision Maine for their generous guidance and mentorship.
I’d also like to thank the teams behind the Call of Duty games at Infinity Ward, Sledgehammer Games, and Treyarch.

Stories / Questions

akimitsu.hogge@activision.com

Please feel free to email me at akimitsu.hogge@activision.com with questions or comments
I’m also curious what other games are doing in this space, so feel free to share if you have stories from your games.

https://danluu.com/latency-mitigation/
http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/
https://docs.microsoft.com/en-us/windows/uwp/gaming/reduce-latency-with-dxgi-1-3-swap-chains