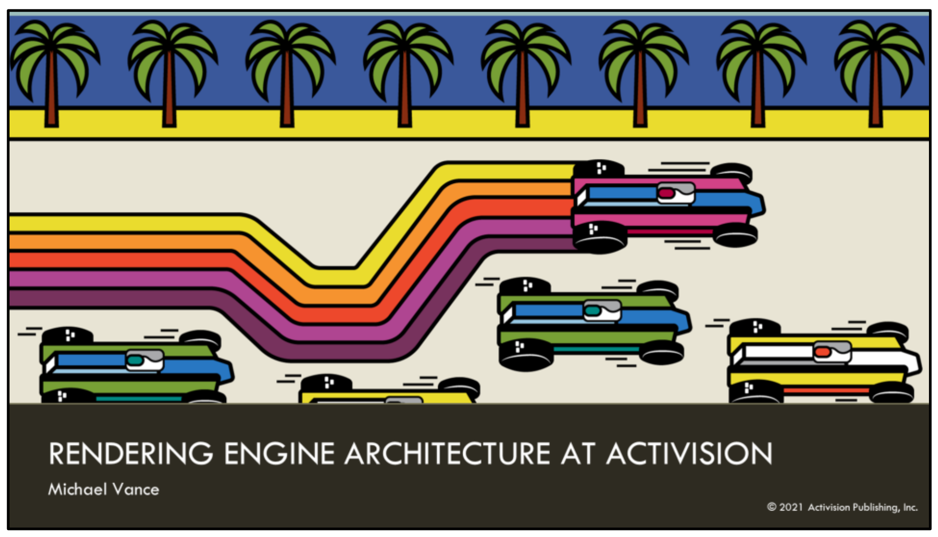
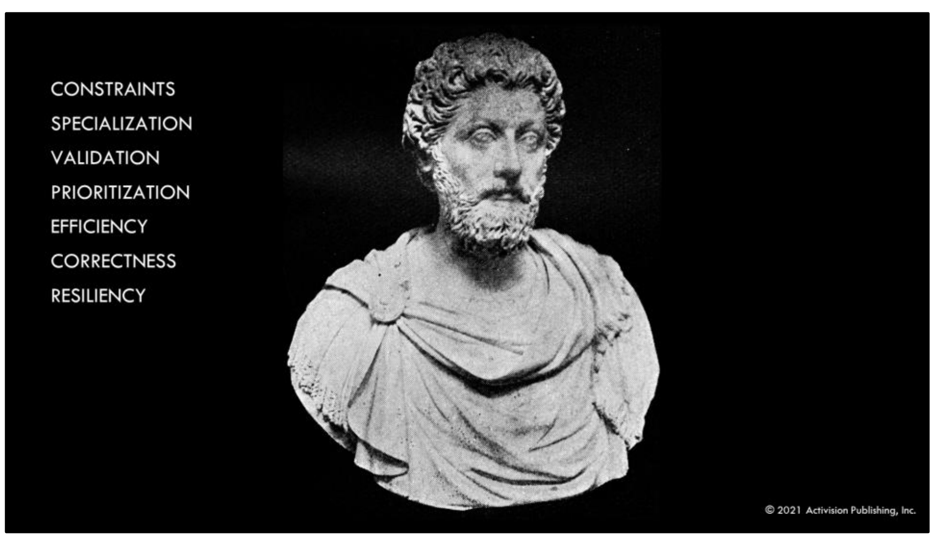
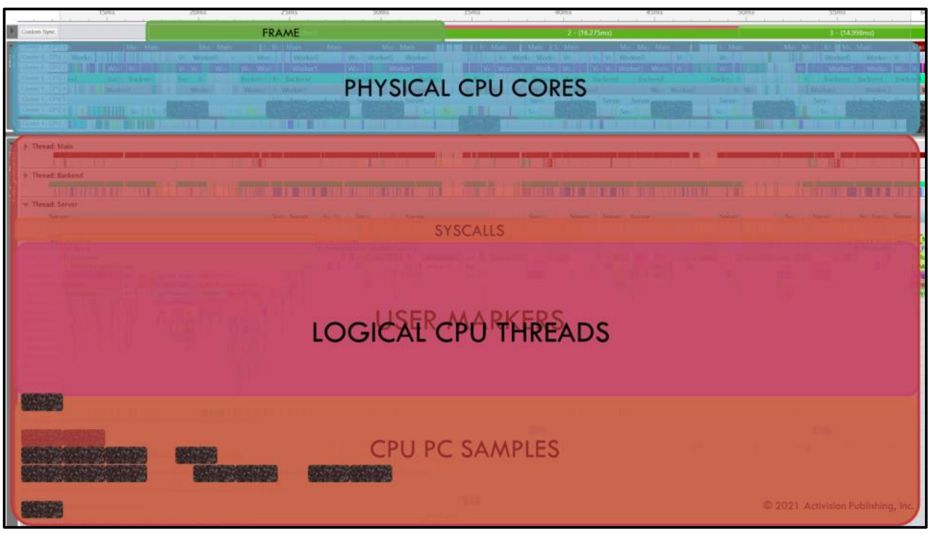


Good morning everyone, my name's Michael Vance.
Thanks for joining us today to talk about something we've been very excited to get going for a while, which is this series on rendering engine architecture.
While I am now the CTO of Activision Publishing, for most of my twenty years in this industry I have been a systems programmer.
Primarily working on rendering code or rendering-adjacent code.



The idea of this series of talks was to discuss the varying approaches to rendering engine architecture in games.
For instance what different decisions do we make if we're a bespoke engine, vs a platform, vs a licensed engine technology?
In our case as a bespoke piece of technology for shipping a specific game, we have some philosophical ideas that are important to us.
Constraints, how to choose them and the specialization they unlock, and the importance of validating them over time.
Prioritization of work based on knowing the technology's use end to end.
How specialization unlocks efficiency.
How we can reason about correctness with large knowledge about the technology.
And how we can engineer for resiliency in our code and why that's valuable.
[MARCUS AURELIUS]



My daughters like to dispel notions of the excitement of working on games by describing it as “staring at a screen for several hours with an annoyed look”.
Which is pretty much spot on.
What I was usually looking at was a performance analysis tool.
And this talk is largely “a tale told in analysis tool captures, with some other pretty pictures to boot.”
At this event there are possibly some people who haven’t worked in these tools that much.
So I’m going to do a quick overview of what we’ll be looking at, since this tool forms the basis of a lot of my slides.
This is Razor CPU, which is our preferred analysis tool for CPU work and understanding of execution.
CLICK
At the very top we have a timeline with a frame marker. Time moves from left to right.
CLICK
Then below that the physical CPU layout showing our named threads.
You’ll notice on this capture that we have two clusters with four cores each: each cluster has its own L1 but they share an L2.
CLICK
This next section is the logical thread view.

CLICK, CLICK
And these are syscalls, you can see the gaps on the logical thread markers correspond to operating system calls like synchronization primitives such as mutexes and condition variables.
CLICK
User markers which are pushed and popped by the client with custom names are shown here.
CLICK
And down here you have PC samples from the executable including callstack walks.

I like to look at things from a ground truth state on the hardware first, so let's begin there, and we'll take a look at it from a higher logical pipeline view later.

CLICK

Our titles operate around three primary threads which unfortunately have a few names each: main/client/frontend for the first, render/backend, and server.

We have multiple names because some have multiple systems responsibility of each, since that can be a little confusing we'll try to use them consistently as frontend/backend/server because this talk is about rendering and frontend/backend is one of the core distinctions.

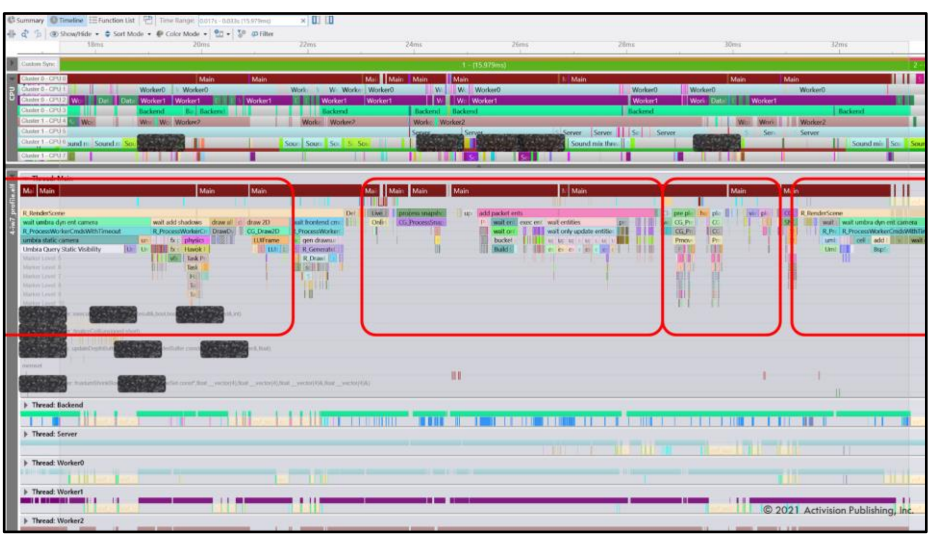
CLICK

We also instantiate worker command threads to cover remaining cores. Database is a special file service thread that cooperatively shares a core with a worker.

CLICK

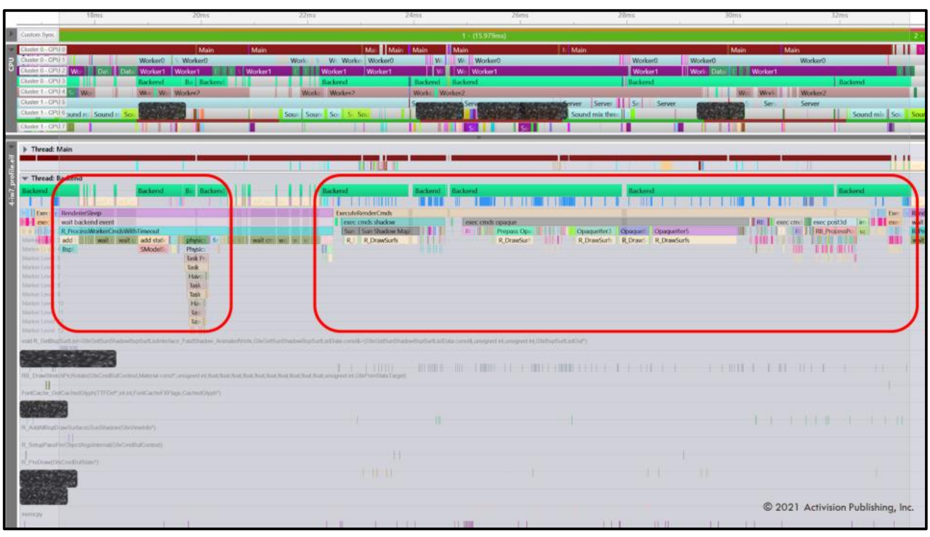
For those unfamiliar, on some consoles the last core is shared partially with the operating system and so you want to schedule work that is robust to latency on there, and audio can mix ahead in time, so we have a large group of audio

threads that are setup for that purpose.
CLICK
Last, there are lightweight task threads that exist with looser affinity to cover low frequency operations such as save game processing, cinematics, etc.



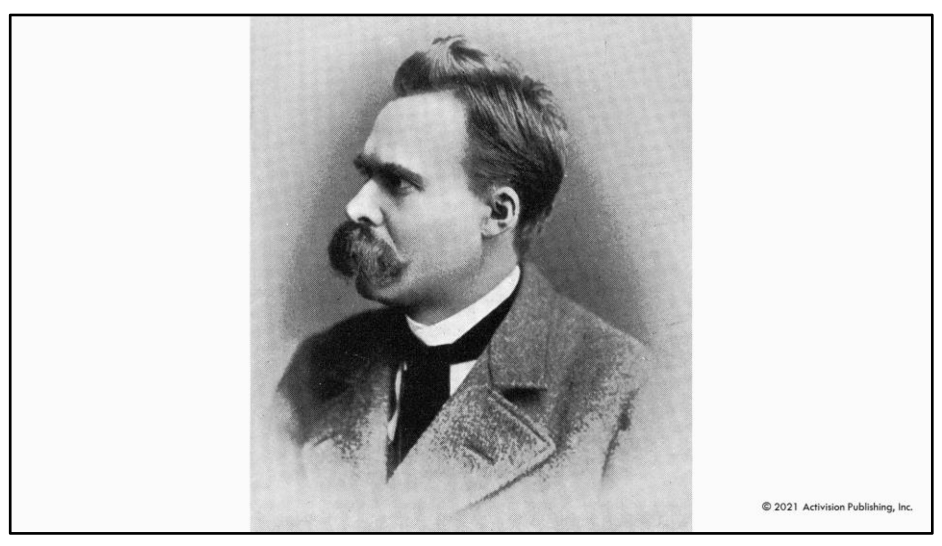
Let’s look at the frontend thread first.
It’s responsible for network packet gathering, input sampling, and scene graph traversal, which is our frontend rendering work.
Most of these operations will spawn worker commands that operate in a fork/join model in order to utilize all available CPU bandwidth.
CLICK
So here I’m highlighting first the “early” part of our main thread, which involves pumping the network on the client side and processing the snapshots as they come in from a raw dispatch perspective.
And then we have the actual work interpretation of the snapshot updates, which you can see here are marked by many waits on forked work.
Those waits then turn into assists to perform actual work to ensure we’re making forward progress.
CLICK
Then our player handling is processed, we do this as late as possible so that input sampling is more closely aligned with kicking graphics work.

CLICK
Last, the frontend scene rendering which is the scene graph traversal phase for the world state.

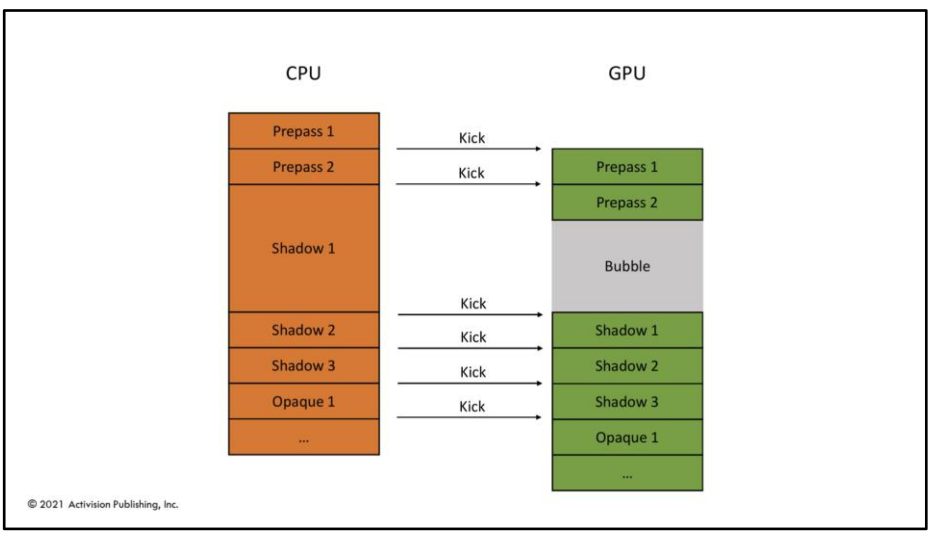


The second major thread, the backend thread.
It is historically responsible for command buffer generation.
But we'll see later that it is more of a simple "graphics device ownership" thread given that so much of the actual command buffer generation happens in parallel on other cores.
And is kicked as early as possible from the frontend.
CLICK
Here I've highlighted what is our command buffer submission on the backend.
You can see a few chunks blow up, this is where the backend itself has picked up jobs it needs immediately and started work on them.
The very heavy hashing in between those is it just submitting command buffer that has already been finished by completed jobs.
CLICK
You can also see earlier in the frame where the backend thread is idle that it is picking up some random jobs like processing frontend command and physics.

We're sensitive to the idea that it might pick up a long job which causes bubbling so we guard against that with custom job dependency rules.



Speaking of jobs going long and latency, let’s have a brief philosophical interlude.
[NIETZSCHE]



We think in terms of the GPU always being our critical path.
That is, the GPU should never be idle, it should be presenting images as fast as possible to the user, and never starved for work.
If the GPU is idle, we feel we have made a mistake in our work scheduling.
A bubble is the term we have for that blank space that shows up and no work is being done, pushing the frame farther out.
When the frame is pushed farther out it increase the odds we will miss our vsync/flip scheduling.
And that will cause a frame to persist, cause latency to spike, and create that feeling of “judder” you get with occasional repeated frames.
Thus we are very careful to avoid bubbling on the GPU, scheduling jobs at enough of a granularity to ensure they are able to feed the GPU command buffer at a sustained rate with no gaps.
And we are willing to trade non-trivial amounts of CPU to improve GPU performance in many cases.
You can think of trying to move such bubbles back through time.
From the GPU to the backend thread which submits work to the frontend thread which creates the input to the command buffer generation, all the way back to the best place to be idle.
Which is right before input sampling because all your work is done ahead of time and you want to schedule input

sampling as late in the pipeline as possible to reduce input to display latency.

Getting back to our final major thread, the server thread.
It simulates game logic decoupled from update rate of visuals, and communicates via loopback/shared memory in single player.
The server is actually run on a dedicated server instance in most situations in multi-player, although we also support running it locally on one of the clients as well.
These are the so called “listen servers”.
The fact that the server runs at a separate tick rate provides us wide latitude in scheduling work on the server--we can afford expensive operations such as pathfinding in single player without affecting the client frame rate.
It also allows us to increase the server tick rate in multiplayer to ensure smooth, low latency play.
CLICK
Here I’ve highlighted how the server “frame” is actually around 50ms, although we’re not server limited here so there is a variety of work its picking up to assist other jobs with.
CLICK CLICK CLICK
You can see how essentially three client frames fit up against the server frame.

CLICK

We have the ability to interpolate between arbitrary tick rates but as you can imagine there is a sort of “state latency” that can occur where the interpolation fidelity is under stress at lower tick rates, or when the server goes long and the client state starts to drift from the server.

Because the server’s view is authoritative the client will eventually “correct” to it which can result in non-linearity/warping in the client’s view.

This is really correction of the client prediction’s failure to reconstruct signal due to a lack of information.

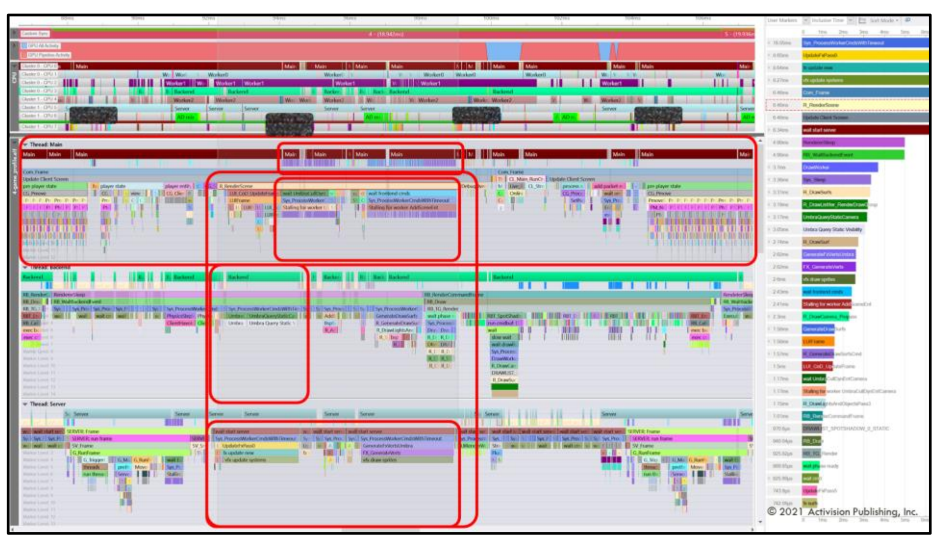
There is complexity to managing this separation between client server--both have their own views of the state of the world, and communicating and synchronizing them requires a non-trivial amount of work.

It can be tempting when programming client functionality in single-player, for instance, to want to “peek through the wall”, and directly examine server state, however this creates significant problems when memory access is not properly synchronized.

This separation is an imposed constraint that allows us to leverage this decoupling, and the payoff is worth the effort, and this idea of copying for memory separation for concurrency will be a recurring one in this discussion.

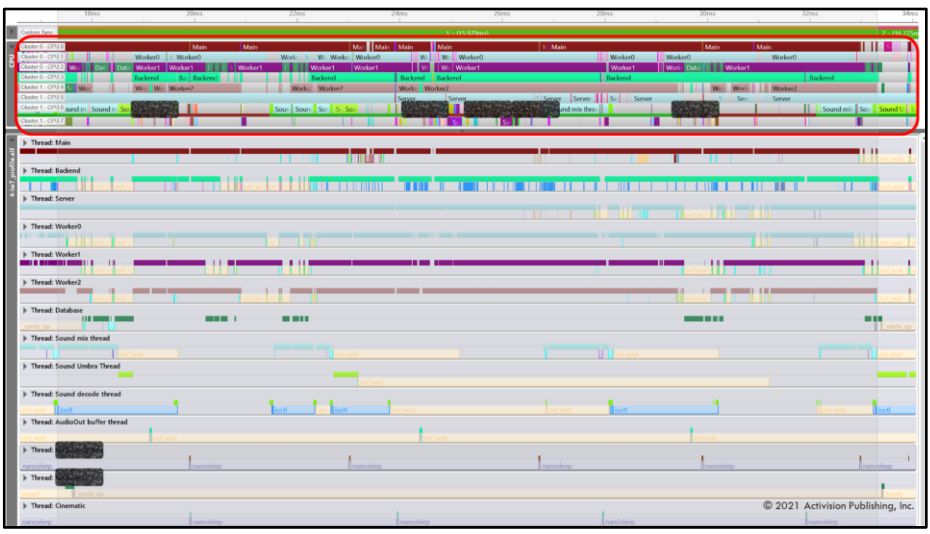
Even if it does make it a pain to move an entity render flag all the way over the network so that the renderer can see the “correct” state for it at some point in time.

This is an important point and so I'll reinforce it again.
We often think of constraints as restrictive of art or design.
But carefully chosen constraints are one of the most important way to unlock large-scale transformations and optimizations in our systems.
Those constraints may have non-trivial engineering requirements but they are what will unlock the quality and performance that deliver value to content creators and players.
The most common examples of these are pre-computed lighting in static environments, but the client-server split is in a similar vein.



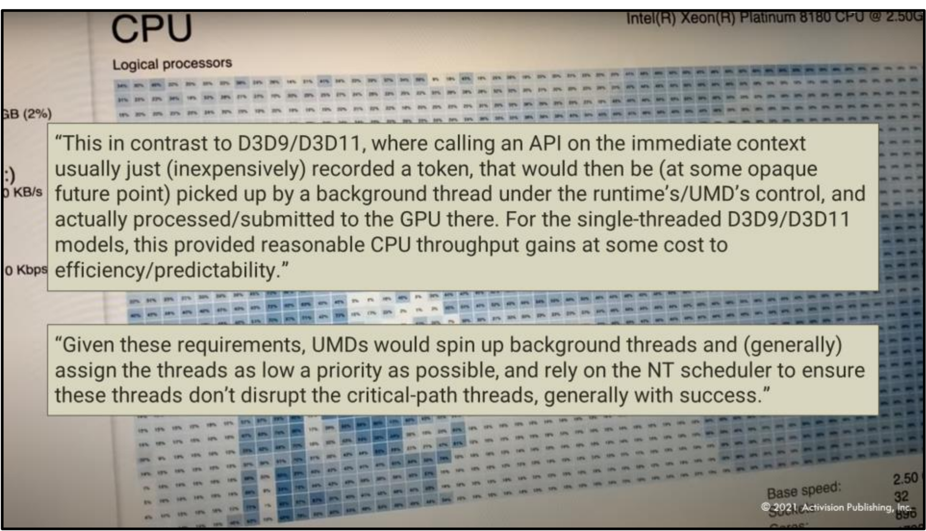
Walking back to our general threading and job model.
CLICK
Focusing on the frontend thread here.
All of these work on these “major” threads are generally executed as a series of jobs in a fork/join model with work spun out to the worker threads.
CLICK
Here you can see our main thread invoke R_RenderScene which is essentially the scene graph traversal.
Including like 1.5ms of UI code which fills me with a deep and abiding sadness.
CLICK
On some of the other cores we start executing visibility work.
CLICK
We’re also executing visual effects system simulation and draw data generation.
CLICK
You can see we’re actually not getting good granularity and balance of jobs here, either, we’re spamming our work

queues on the main thread to see if the jobs are done, instead of helping out with any of them.
CLICK
And those of you familiar with the sight of many syscalls in Razor know that's not the greatest sign when you see that many tick marks.



Last point from our “ground up” view.
CLICK
All of our primary threads and worker command threads have fixed processor affinity on console.
This is in order to ensure optimal memory coherency for execution due to the Jaguar SOC per-cluster cache arrangement.
We execute work on the second cluster that has the most memory orthogonality--the server thread and audio processing are specifically targeted for this.
As mentioned the server thread has its own memory representation of entities distinct from the client’s, so we run the server on the second cluster.
And similarly audio data has very little interaction with the main client thread outside of some shared data for systems like notetrack events.
You will notice that there is a worker thread on the second cluster, and also that we discussed previously that the server thread will pick up work when idle.
In the past we attempted various heuristics to optimally arrange which jobs got picked up by the second cluster workers.

We found that in real world scenarios it was always at least marginally advantageous to suffer the cache pollution in exchange for the extra processing throughput.
We've also worked to ensure that OS threads like network drivers, are affinitized in a way that makes the most sense for data delivery, e.g. the main thread processing snapshots coming in via dedicated servers.



We *have* attempted similar thread affinity work on PC in the past but run up against difficulty in finding a robust solution that wasn’t in conflict with the OS scheduler.

CLICK

This was especially true in the D3D11 era when PC GPU drivers would perform non-trivial work, and in general we were almost always bound on CPU by the primary driver thread.

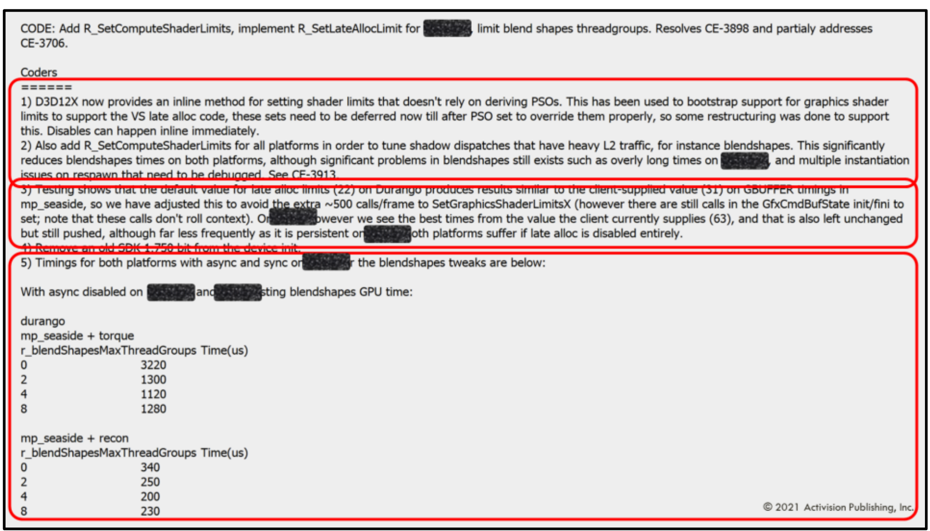
On ARM we do respect BIG.little configurations and schedule heavyweight threads as appropriate and as you would expect based on what I just explained.

CLICK

As a side note, this slide is from the D3D12 presentation on why they were restoring background worker threads as a concept in D3D12.

I’m not sure we’ve gone back and re-tested our thesis on processor affinity on PC recently which is definitely something we should do.

That’s one of the important things about picking a set of constraints--every now and then you have to go back and re-examine them to make sure the assumptions you decided on are still valid.



CODE: Add R_SetComputeShaderLimits, implement R_SetLateAllocLimit for [illegible], limit blend shapes threadgroups. Resolves CE-3898 and partialy addresses CE-3706.

Coders
======
1) D3D12X now provides an inline method for setting shader limits that doesn't rely on deriving PSOs. This has been used to bootstrap support for graphics shader limits to support the VS late alloc code, these sets need to be deferred now till after PSO set to override them properly, so some restructuring was done to support this. Disables can happen inline immediately.
2) Also add R_SetComputeShaderLimits for all platforms in order to tune shadow dispatches that have heavy L2 traffic, for instance blendshapes. This significantly reduces blendshapes times on both platforms, although significant problems in blendshapes still exists such as overly long times on [illegible], and multiple instantiation issues on respawn that need to be debugged. See CE-3913.
3) Testing shows that the default value for late alloc limits (22) on Durango produces results similar to the client-supplied value (31) on GBUFFER timings in mp_seaside, so we have adjusted this to avoid the extra ~500 calls/frame to SetGraphicsShaderLimitsX (however there are still calls in the GfxCmdBufState init/fini to set; note that these calls don't roll context). O[illegible]owever we see the best times from the value the client currently supplies (63), and that is also left unchanged but still pushed, although far less frequently as it is persistent o[illegible]oth platforms suffer if late alloc is disabled entirely.
4) Remove an old SDK 1.750 bit from the device init.
5) Timings for both platforms with async and sync o[illegible]r the blendshapes tweaks are below:

With async disabled on [illegible] and [illegible]sting blendshapes GPU time:

durango
mp_seaside + torque

r_blendShapesMaxThreadGroups	Time(us)
0	3220
2	1300
4	1120
8	1280

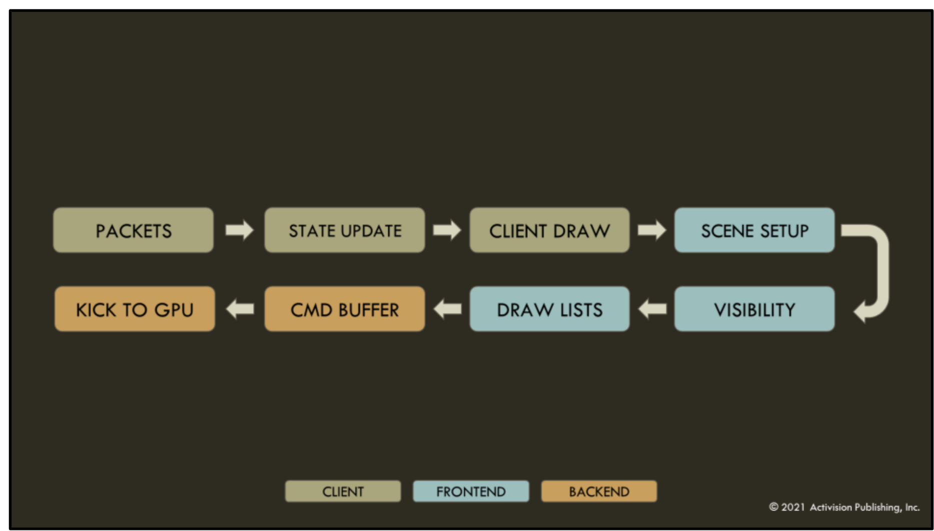
mp_seaside + recon

r_blendShapesMaxThreadGroups	Time(us)
0	340
2	250
4	200
8	230

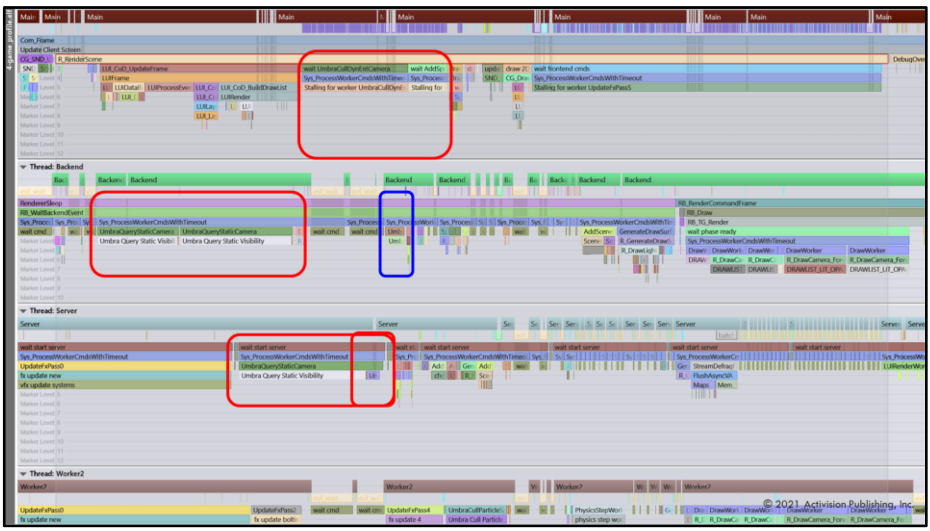
© 2021 Activision Publishing, Inc.

This reminds me of another digression, which is our approach to changelists.
This may sound a bit banal, but much advanced work that we do on the renderer is not just about looking forward, but also understanding why changes were made historically.
Good changelist discipline is critical for understanding this, and stands, in my mind, in stark contrast to the relative uselessness of code comments and documentation.
Most comments and documentation are a risk in my mind, because they can quickly decouple from the underlying system implementation.
Changelist comments, however, are (for the most part) immutable in time and directly linked to the textual changes present in the diffs applied in that change.
Our approach is that we don't want them to be descriptive or duplicative of the textual diffs in the change--what's the point of restating that you renamed a function when the diff tells you that?
Instead we want to know *why* you made the change you did, CLICK what testing you did to validate it CLICK, and what were the observed results?
With that information, we can go back and test an assumption that was made earlier and validate the results/benefits are the same as they once were.
Which is what I should have someone do later this week on PC!

This is one of my changes from a couple of years back when I still did engineering work, and is based on a style popularized by Robert Field at IW.



Back to our render work submission.
In comparison to the physical CPU and logical thread setup, we can also think of the renderer from the top-down as a pipeline of data and tasks moving across threads and time.
The front of the pipeline is the state updates to the world simulation.
The middle piece is the frontend scene traversal, including scene setup, which includes the view/camera data, light state, game mapping between semantic image resources and actual texture data, etc.
It also kicks the visibility jobs which are then the input to the draw list generation.
The last piece is the command buffer generation and work submission to the GPU, which is owned by the backend.
The draw lists are traversed and turned into command buffer which are then kicked to the GPU in scene submission order.

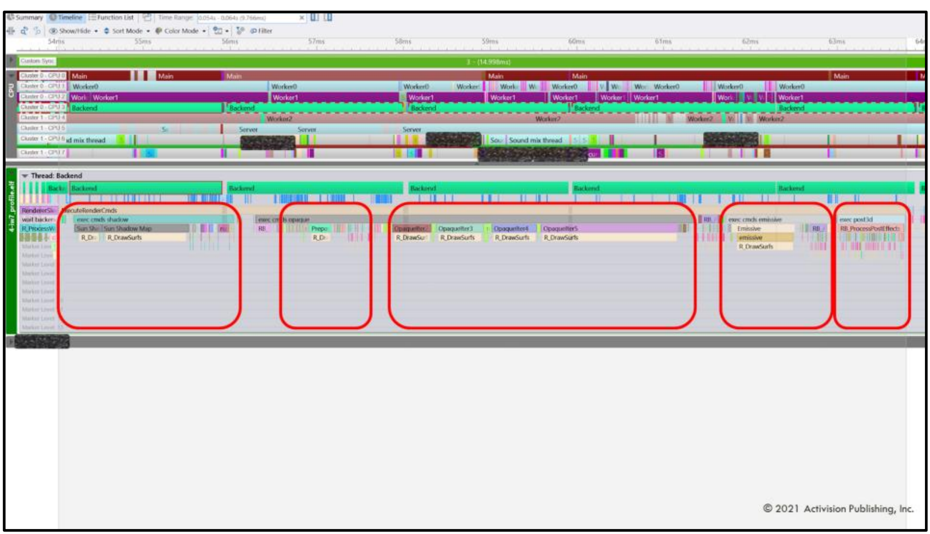


Zooming in looking at the early part of the frontend, during the visibility phase jobs are kicked per camera (main scene, sun, spots, etc.) and then per object type which determine visibility of these disparate elements for each camera.
We have used a variety of visibility determination approaches through time, from classical portals with narrowed traversal, to bespoke software rasterization of likely occluders, to middleware solutions like Umbra.
All of these approaches have varying tradeoffs which we don't have time to explore in detail, although harkening back to my earlier point about re-testing assumptions, we did do a fulsome evaluation of these again on a recent title.
For the sake of today's discussion we'll assume that we're using Umbra for visibility determination.
In our case we don't use Umbra merely to analyze the presence of renderable objects in the world, but we also use it to determine visibility of lights, reflection probe volumes, and decals.
Culling inactive "scene meta objects" is a valuable way for us to constrain overall processing as part of our forward+ renderer.
Here you can see the majority of our time is spent processing static object visibility.
CLICK
And then a much smaller section here where we're processing what we call "scene entities".

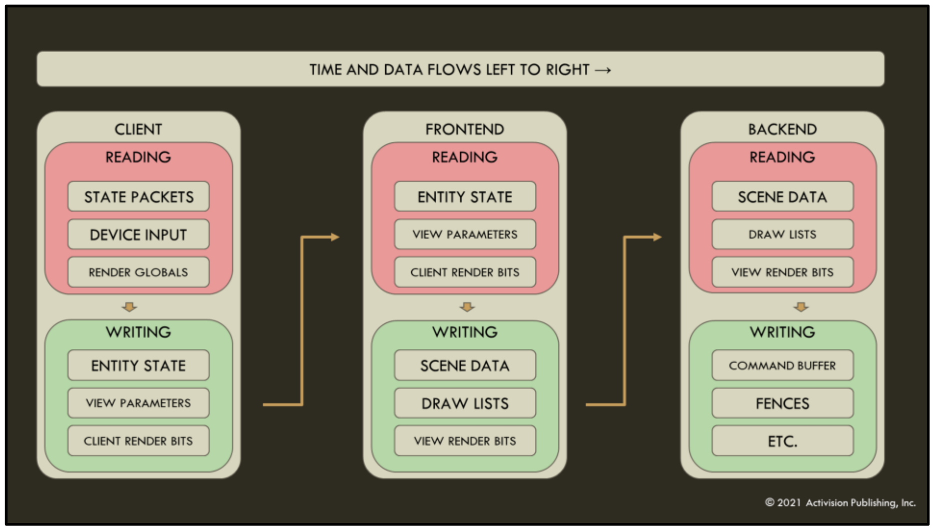
CLICK
Two other things worth noting here.
CLICK
Here you can see a small job embedded at the end of the last static visibility query job.
This is a coalesce job that unifies all the per-job buffers into the final list.
We run it inline rather than have another job kicked and waiting in order to keep the memory hot.
CLICK
And here you'll notice a non-trivial inefficiency.
We're spending a lot of time spinning here because our wait is on a job with a long dependency chain and we don't have much else to do, and we're afraid of missing end of the visibility jobs, so we're not picking up other work.
To some extent this is a legacy of the capture I made which is an empty MP match, a normal game would have more work to hide here, but it's interesting to observe, I may yell at someone about this.
But this goes back to the earlier point about concern for bubbles, etc.

In the next phase, we populate draw lists which correspond to various passes in the scene, and which map back to camera setups and visibility lists from the first phase.
These population jobs are launched according to the specific underlying object type.
CLICK
Brushes
CLICK
static models
CLICK
dynamic models
CLICK
and particle systems.
These draw lists include typical passes you'd be familiar with like shadows, prepass, viewmodel, opaque, but also special buckets like subsurface scattering skin materials, etc.
Each of these draw lists also corresponds to a command buffer chunk which will later be generated based on the draw

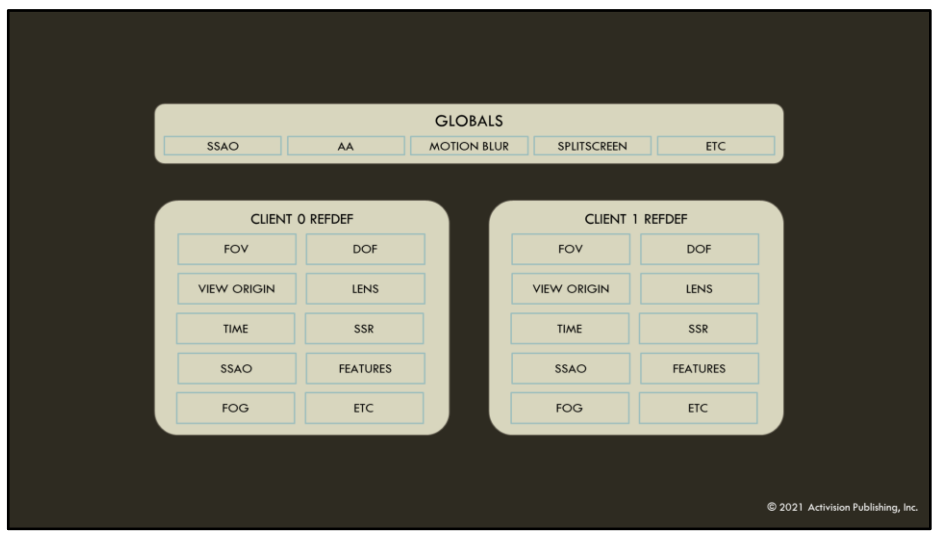
list assuming it’s not empty.
CLICK
You can also see some of the meta objects I mentioned previously like the frustum lights here, which we draw as objects into the bit-vector with atomics.



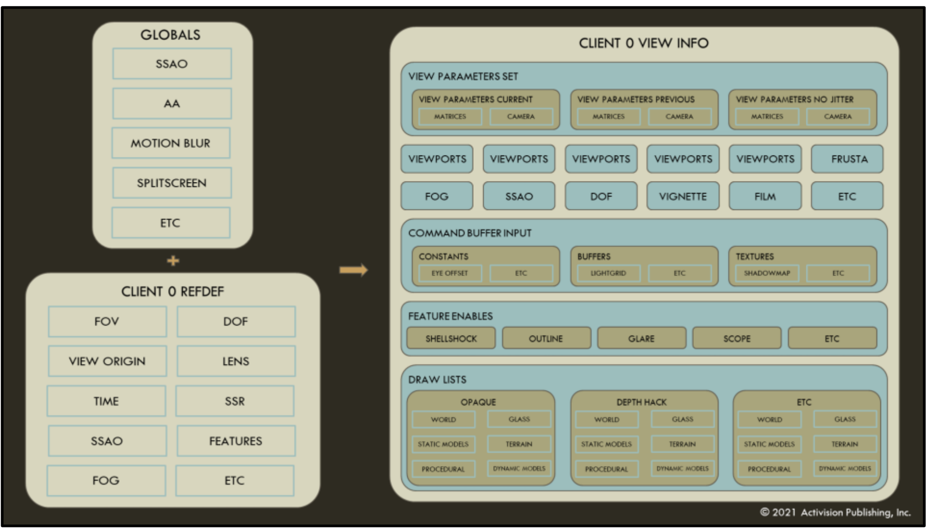
The object draw lists are split up by semantics of which part of the frame they belong to: shadows, viewmodel, opaque, skin surfaces, distortion effects, transparencies, UI, etc.
Rough categories called out here show us submitting CLICK shadows first, CLICK then prepass, CLICK then opaque, broken up into several segments.
CLICK
Then what we call emissive which is really a deprecated term for transparencies and particle effects.
CLICK
Last, the post processing chain is submitted.
So that's a sort of more granular view of the job flow of each of the pieces of the pipeline but next I want to talk a little bit about data flow.



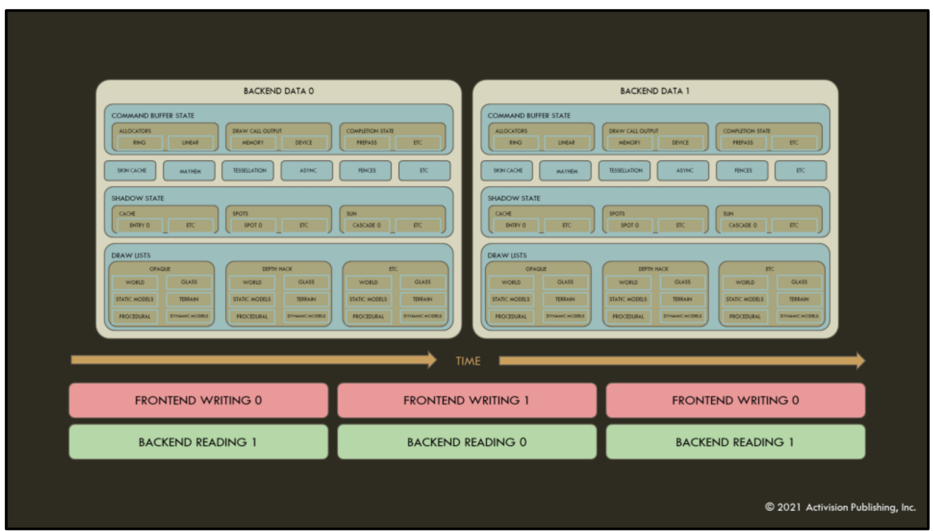
What I'd like to talk about next is data mutability.
The draw lists are mutable from the frontend's perspective, and immutable from the backend's perspective, modulo many years of small subtle and shameful hacks here and there.
There is a general flow of feeding data forward that is present in a lot of the different stages in the renderer.
We're transforming or copying data, or we're transitioning it from writable to read-only as it flows through the pipe.
The split between the frontend and the backend is the major inflection point, with the largest gathered data bundle, the 'backend data', but there are multiple other variations of this throughout the frontend itself as it is forking and joining jobs.



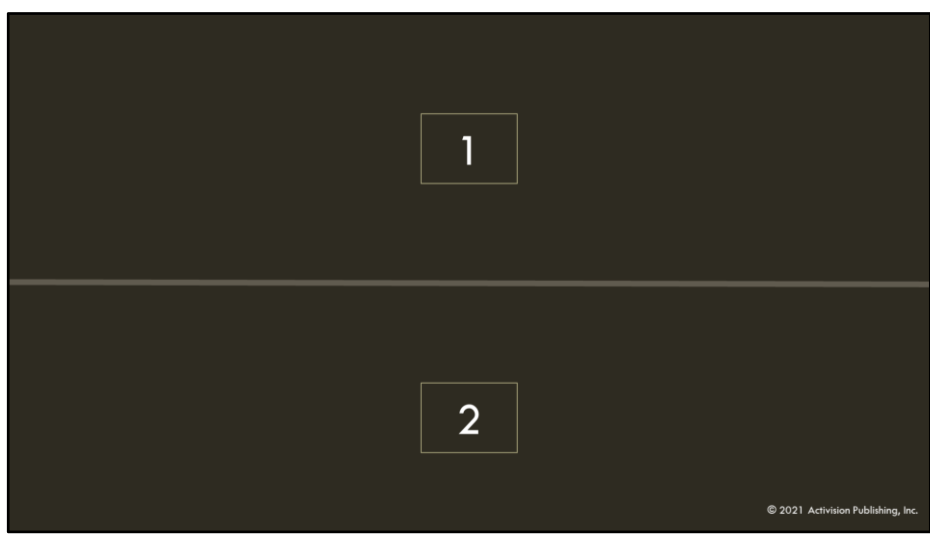
I'll try to illustrate the data flow in more detail and talk about why it is structured the way that it is at each point.
First, there is global render data stored in the appropriately named render globals.
Then each client has a set of data that belongs to it specifically, including what is known as a reference definition or refdef.
In the case of splitscreen this will mean two locally connected clients, for instance, and thus two refdef.
Incoming packets from the network tell us how to modulate the client state and then in turn its refdef is updated, as an example the current fog state for the player.
This will be a variety of parameters like Rayleigh (RAYLEE) scattering, etc., that would be interpolated from vision sets that the server informs us are now in effect, etc.



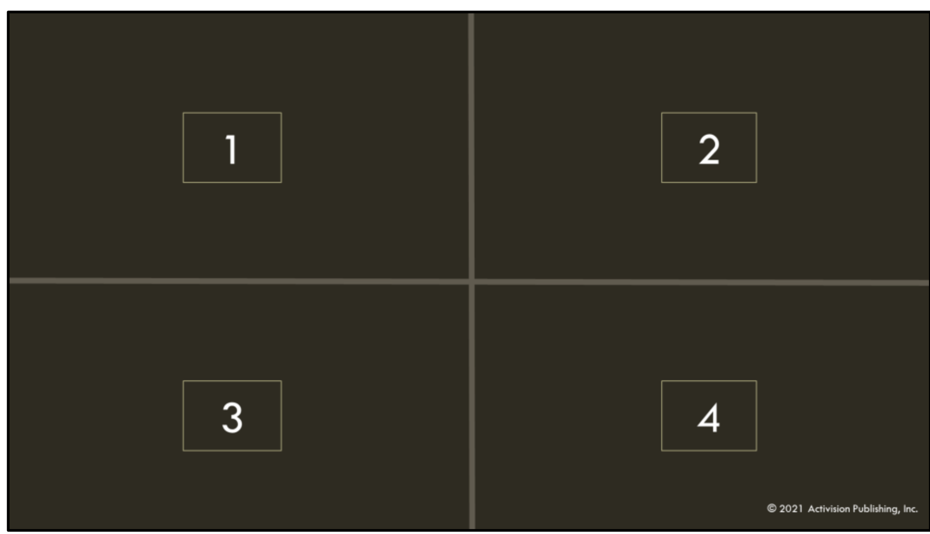
Something we call the viewport features, will be passed down into the renderer frontend which will make a copy and modulate it based on certain renderer limitations that the client need not know about, most of which are debug tunables but some of which represent PC optional features, etc.
We then begin splitting up in the input refdef data into pieces based on frequency of specification, data that needs to be saved as the previous view parameters for temporal effects will be split from others which have no cross-frame references, etc.
We now also begin to setup the “view info” which will be one of the major first class objects visible to the command buffer generation as an input.
As the “view info” is setup, so is something we call the “command buffer input” which contains a lot of input data specific to command buffer generation.
For instance you want to have a system to map render targets as inputs to later passes and you can use this system to specify these.
They’re decoupled in this way so you can override them with e.g. black/white/gray images for various purposes, mostly debug but sometimes for real effect in the game.
Similar mechanisms exist for constant buffer management in the “old push style” although we’re moving more and more towards arbitrary packed blobs of data in pull style these days.



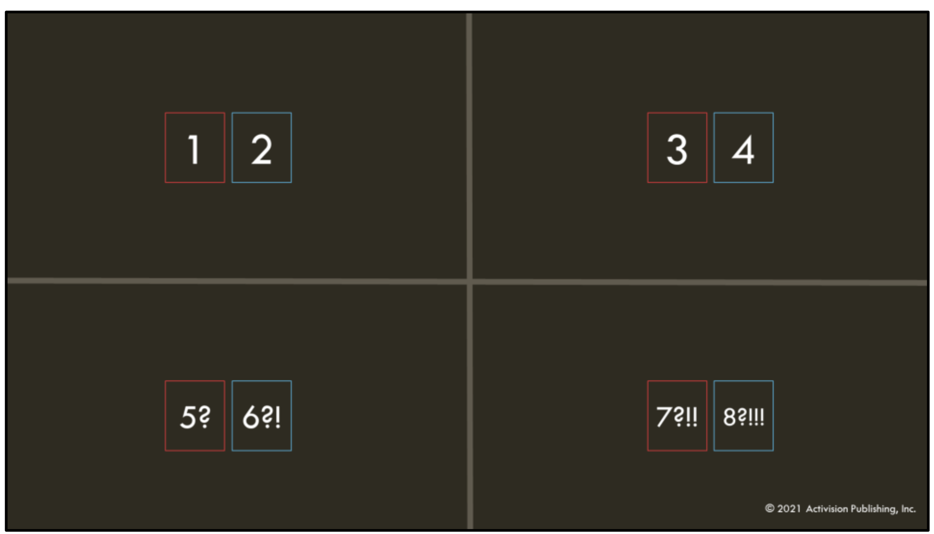
We'll also begin setting up "backend data".
The backend data represents the interchange between the frontend and the backend of the renderer, namely the difference between scene traversal and command buffer generation.
There are two of these "backend data" structures, and as soon as the frontend thread is done writing it will be handed off to the backend as transfer of ownership and then the frontend thread can immediately begin writing the next frame's worth of data.
This double-buffered pipelining allows us to amortize and overlap work at the cost of some carefully managed latency.
Another thing worth mentioning is that we try to avoid is any linkage in saved state between the frontend and backend.
There is little caching of work in the backend, everything is feed forward to avoid consistency issues in such designs, and the requisite synchronization.



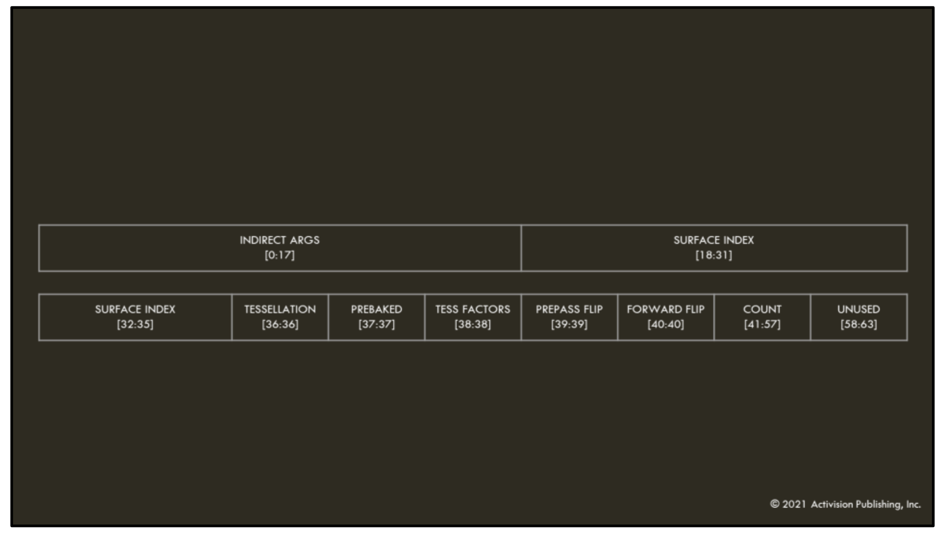
We discussed the duplicated backend data for pipelining previously.
In the case of splitscreen we'll also utilize this double-buffered mechanism to begin processing the next "backend data" for the second viewport.
CLICK
This reduces our pipelining benefit of course but given the additional stress on systems to support splitscreen it works out alright in balance, and it is most important that we generally have a fully independent set of resources available for the second viewport when possible.
CLICK
Unfortunately our support is not as elegant as one might hope, but of course elegance and efficiency aren't necessarily bedfellows.
It's not uncommon to find various rendering systems disabled to improve performance in splitscreen such as tessellation (even worse quad efficiency at sub-viewport!),
Moreover there are some systems whose benefits simply don't exist, for instance early asynchronous compute jobs which expect "end of frame" work to overlap can collide with full cost end of viewport draw work from the first client.
Or, due to memory lifetime management, would need to stall for longer than would be desired in order to have correct CPU/GPU synchronization.



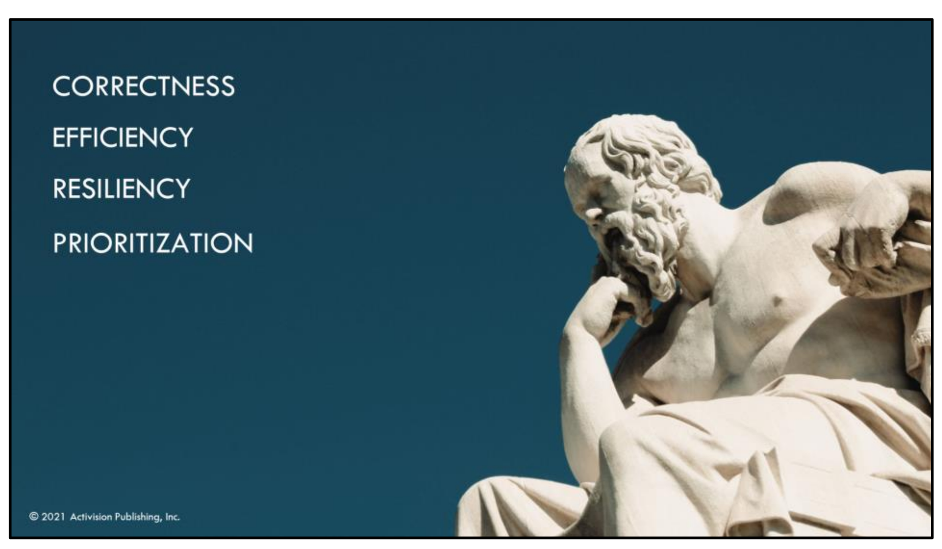
Last, some systems can be optimized based on previous frame hints, such as our lightgrid spatial walk, and this is more complicated to manage in splitscreen.
CLICK
Some work is amortized--we'll often do tricks like only render one set of sun shadow cascades for splitscreen.
CLICK
A lot of the functionality that is currently encoded in splitscreen is a mixture of past legacies as well--picture-in-picture, stereoscopic rendering, etc.
CLICK
But splitscreen is still important to the franchise and players and we do the work to support it even if it's not internally as beautiful as we'd like.
CLICK



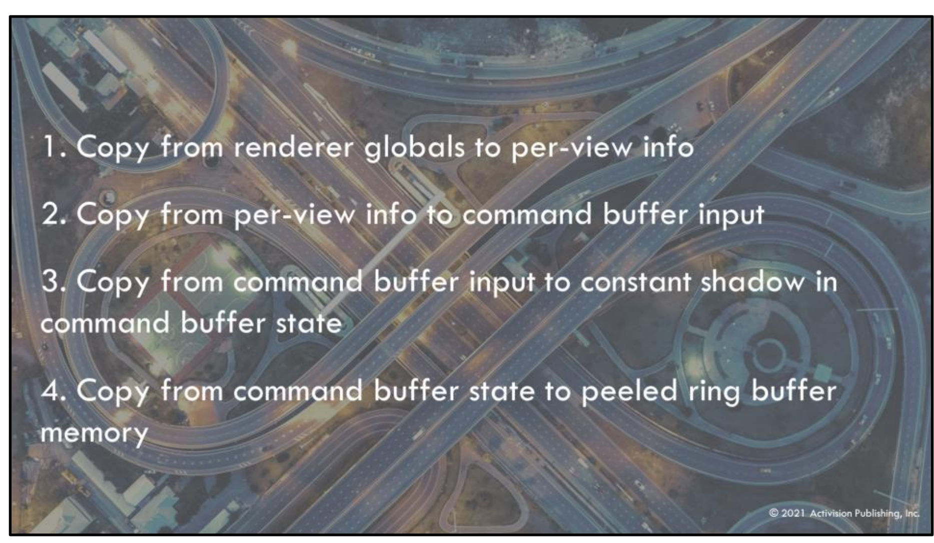
In the case of past features like stereoscopic 3D...
CLICK x 8
… we managed to get up to 8 backend data swaps per frame, which, uh, mostly held 20Hz for four player co-op stereoscopic 3D.
I'm guessing probably a dozen people ever ran the game in that configuration.



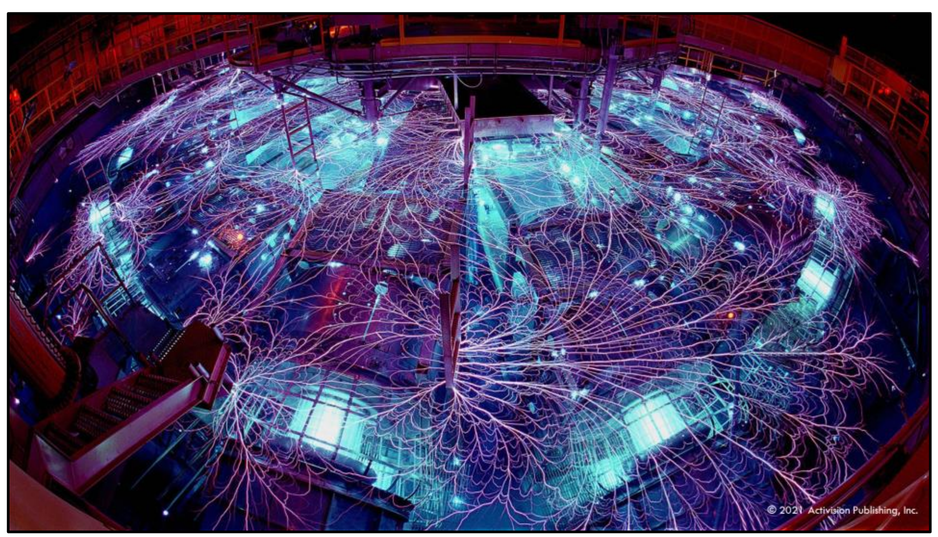
Another benefit for mutability analysis is that if you can freeze the data early in the pipeline you can avoid copying overhead and just refer to it.
And copy overheard is something to pay attention to, and to make decisions about bulk vs fine-grained, data structure size, etc.
Which is why for example our drawable interchange data is very referential, and tightly packed.
Take for example our world-geometry drawable surface data.
It’s encoded as a 64-bit integer.
That surface index refers to other arrays of data in true SOA spirit, but that data is similarly packed, albeit it into 32-bits, and another chunk which is 32 bytes, etc.



Let's pause for another philosophical interlude on what I consider some of the most important framing ideas of our renderer design, which are correctness, efficiency, resiliency and prioritization.
[SOCRATES]



First of these, minimize explicit synchronization needs to ensure correctness.
This implies a fair amount of copying, and one of the criticisms of our internal engineering team is that it can take a lot of effort to move things through the render pipeline.
CLICK CLICK CLICK CLICK CLICK
But this data stays immutable as it flows through the pipe other than the copy points and thus minimizes synchronization needs, complex mutex systems, etc.
As well as the tendency for bugs to creep in due to referencing memory we shouldn't, similar to what we spoke about earlier with the client vs server fence.
This isn't a new or novel insight per se, functional languages have talked about this for years, but we have seen a lot of messes from state mutation, so discipline here really pays off.



The downside of these multiple stages is of course their cost.
Thus our second goal, which is to minimize the amount of data that needs to flow through the rendering pipeline as much as possible for the sake of efficiency.
A more general way we would sometimes say this is that our goal with code and data flow is to minimize total data transformation energy.
Each transform you are required to do, whether a copy or calculation, represents the spending of some finite resource in the system such as memory bandwidth or ALU.
Minimizing this keeps the renderer efficient.

```
	assertstring( DObjSkelAreBonesUpToDate( obj, &skinCmd.surfacePartBits ), DObjGetName( obj ) );

	if ( !totalSurfaceCount )
	{
		PROF_END();
		return 0;
	}

#if USING( MAYHEM_CUSTOM_CHANNELS )
	if ( skinningContext.numMayhemChannelVerts )
	{
		if ( !R_AddMayhemChannelsComputeCmds( frontEndDataOut->compute.cmdList, skinningContext.numMayhemChannelVerts, surfBuf, truncate_cast<uint>( surfP
		{
			PROF_END();
			return 0;
		}
	}
#endif // #if USING( MAYHEM_CUSTOM_CHANNELS )

	if ( skinningContext.numSkinnedVerts )
	{
		if ( !R_AllocDObjSurfsSkinnedCacheVerts( surfBuf, surfPos, totalSurfaceCount, skinningContext.numSkinnedVerts, motionblurBuildInfo ) )
		{
			PROF_END();
			return 0;
		}

#if USING( TENSION_MAPPING_RUNTIME )
		if ( skinningContext.numTensionMappingVerts > 0 )
		{
			if ( !R_AllocDObjSurfsTensionMappingData( surfBuf, surfPos, totalSurfaceCount, skinningContext.numTensionMappingVerts ) )
			{
				PROF_END();
				return 0;
			}
		}
#endif // #if USING( TENSION_MAPPING_RUNTIME )
```

© 2021 Activision Publishing, Inc.

I also discussed that we valued resiliency.
We want our designs to be resilient to unknown inputs at a level that is appropriate for the constraints we have chosen.
Tools should never assert on malformed input, for instance, but handle errors gracefully and report as much context as possible to the user.
Similarly if we are unable to determine a static limit a priori, we want runtime systems to handle that gracefully.
As a simple example, one thing we try to do in our traversal that is worth highlighting is that we have a system whereby the frontend is responsible for resource allocation used by the subsequent workers.
This includes determining the buffers necessary to be reserved in the interchange data structure between the frontend and backend.
At the top level these are surface descriptors, but they can also include reserving space in the skinning output buffer or other graphics related structures that are used strictly by the backend.
If we are unable to reserve this space we simply elide the draw at the top level--it's a goal of the renderer that it is resilient to heavy load without crashing.
This is especially helpful in the early days when most gameplay was 6v6 DM and then a killstreak would come in and try

to draw the entire map at once and things would still be handled gracefully.
This is our top level “dynamic model” draw code, which if it has skinned verts will attempt to allocate them and simply skip rendering if it can’t.

```
NO_SANITIZE( "alignment" ) // Misaligned structs used in surfBuf[R_DOBJ_SURFBUF_SIZE]
static bool R_AllocDObjSurfsSkinnedCacheVerts( byte *RESTRICT surfBuf, byte *RESTRICT surfBufEnd, uint totalSurfaceCount, uint numSkinnedVerts, GfxMoti
{
	int				skinnedCachedOffset;
	GfxModelBaseSurface	*RESTRICT baseSurf;
	byte				*RESTRICT surfPos;
	uint				totalSurfaceIndex;
	int				surfSkinnedVertOffset;
#if USING( VELOCITY_RENDERING )
	GfxSkinCacheEntry	*skinCacheEntry;
#endif // #if USING( VELOCITY_RENDERING )

	skinnedCachedOffset = R_AllocSkinnedCachedVerts( numSkinnedVerts );
	if ( skinnedCachedOffset < 0 )
		return false;

#if USING( VELOCITY_RENDERING )
	int oldSkinnedCachedOffset = -1;
	if ( mbBuildInfo->useVertexCaches )
	{
		skinCacheEntry						= mbBuildInfo->skinCacheEntry;
		oldSkinnedCachedOffset				=( gfxBuf.skinnedCacheFrameCount - skinCacheEntry->frameCount == 1 ) && ( skinCacheEntry->numSkinnedVer
		skinCacheEntry->numSkinnedVerts		= numSkinnedVerts;
		skinCacheEntry->skinnedCachedOffset	= skinnedCachedOffset;
	}
#endif // #if USING( VELOCITY_RENDERING )

	surfPos = surfBuf;

	for ( totalSurfaceIndex = 0; totalSurfaceIndex < totalSurfaceCount; totalSurfaceIndex++ )
	{
		baseSurf = reinterpret_cast< GfxModelBaseSurface * >( surfPos );

		if ( R_IsRigidSurfaceType( baseSurf->skinnedCachedOffset ) )
		{
			surfPos += R_GetModelRigidSurfaceSize( reinterpret_cast< GfxModelRigidSurface * >( surfPos ) );
			continue;
		}
```

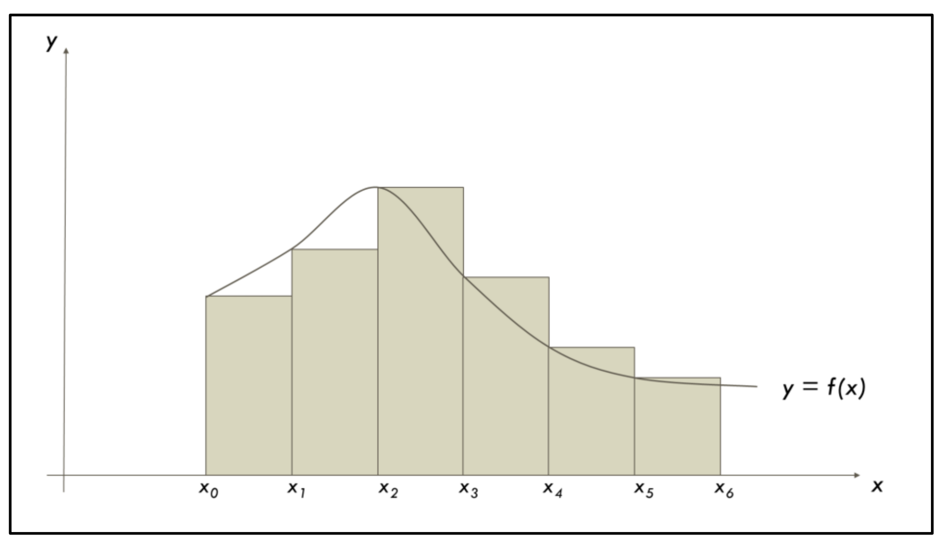
© 2021 Activision Publishing, Inc.

Inside that code a call to another sub function to get the offset into the skinned cache buffer, and skip if unable to allocate.

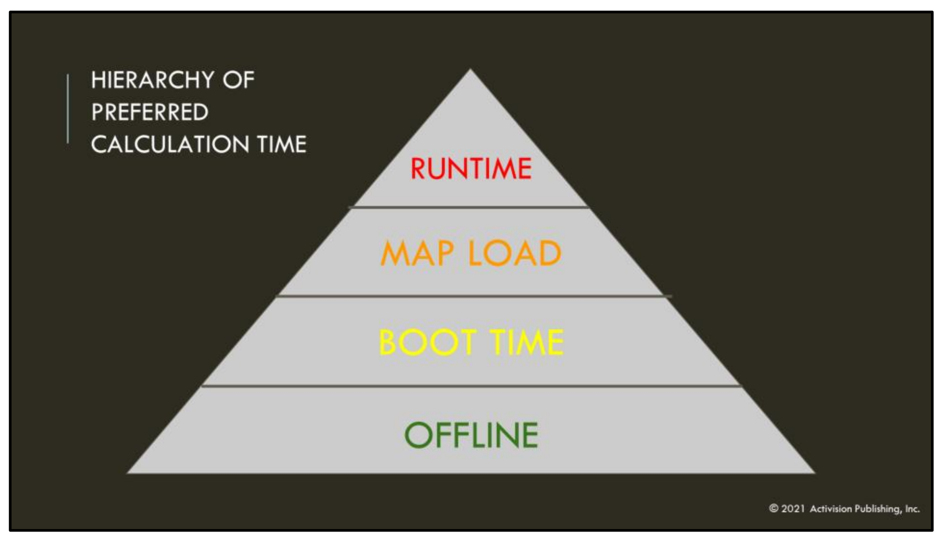
```
static int R_AllocSkinnedCachedVerts( int vertCount )
{
	uint offset;
	uint skinCacheSize;

	assert( vertCount > 0 );

	assert( frontEndDataOut );
	assert( frontEndDataOut->skinnedCacheVb );

	offset = Sys_InterlockedExchangeAdd( &frontEndDataOut->skinnedCacheVb->used, vertCount );

	skinCacheSize = frontEndDataOut->skinnedCacheVb->total / R_SKINNED_VERT_SIZE;

	if ( offset + vertCount <= skinCacheSize )
	{
		return offset;
	}

	R_WarnOncePerFrame( R_WARN_MAX_SKINNED_CACHE_VERTICES, skinCacheSize );
	return -1;
}

#if USING( SUBDIV_SURFACES )
bool R_AllocSubdivSkinnedCachedVerts( uint cacheSize, int *cacheOffset )
{
	uint offset;

	assert( cacheSize > 0 );
	assert( frontEndDataOut );
	assert( frontEndDataOut->subdivCacheVb );

	cacheSize = (cacheSize + 15) & ~15;

	offset = Sys_InterlockedExchangeAdd( &frontEndDataOut->subdivCacheVb->used, cacheSize );
```

© 2021 Activision Publishing, Inc.

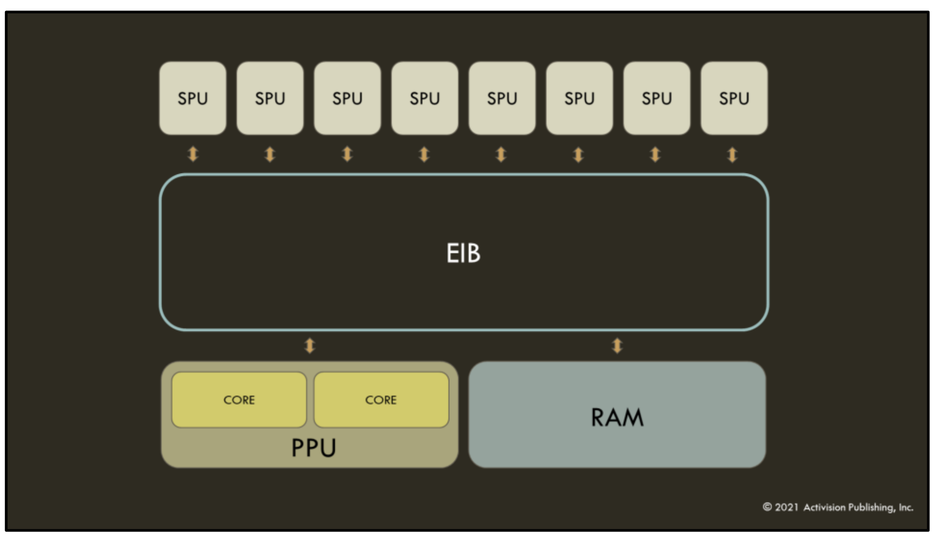
CLICK
The actual code to alloc from the buffer.
CLICK
The atomics.
CLICK
And if unable to allocate, issue a warning once per frame, and then return an error.
Another important part of this resiliency is validating it--one thing we encourage programmers to do as part of their testing for checkin is to artificially reduce their limits and ensure system works correctly.
In this case we'd ask them to knock down the size of the skinned vertex cache vertex buffer to something trivially small and see what happens.



Last, prioritization.
This is the idea that you should gauge the value of a change by the area under the curve of its exposure to the players over time.
If you're working hard to optimize a part of the code that can be very slow but isn't seen very often, an occasional spike, the area under that curve of the player's experience is pretty small.
However if you're talking about the tradeoff in a lighter needs to wait an extra 45s for the radiosity solver to finish but the new output will result in a savings of 130us each frame and multiply that out by the number of play hours… you get the picture.

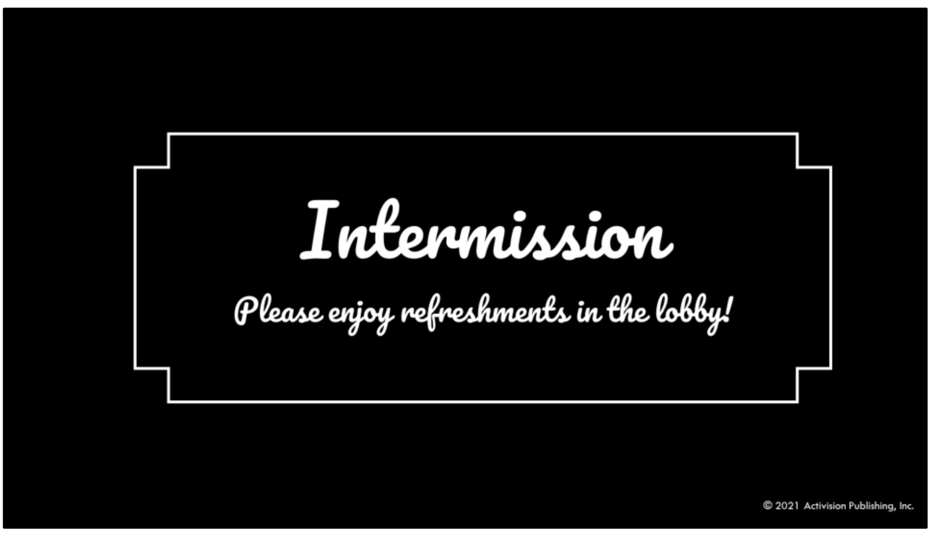


This also leads to a sort of corollary which is the “hierarchy of preferred calculation time”, which Angelo so viciously slandered.
Let’s consider resource allocation within the frame.
Because we value correctness and efficiency we try to disallow anything that looks like dynamic memory allocation during the frame--we want to frontload calculation of necessary resources to as early as possible.
In the best case we can do that offline by reserving memory for some known upper bound across all of our levels.
At boot time we may make decisions such as which piece of hardware we’re running on, resolution of the output device, etc.
Then at map load time we can make a sized allocation per specific level based on its contents or game mode.
And then at runtime we want our allocators to be as simple as possible, which means general some form of linear allocator with a per-frame reset.

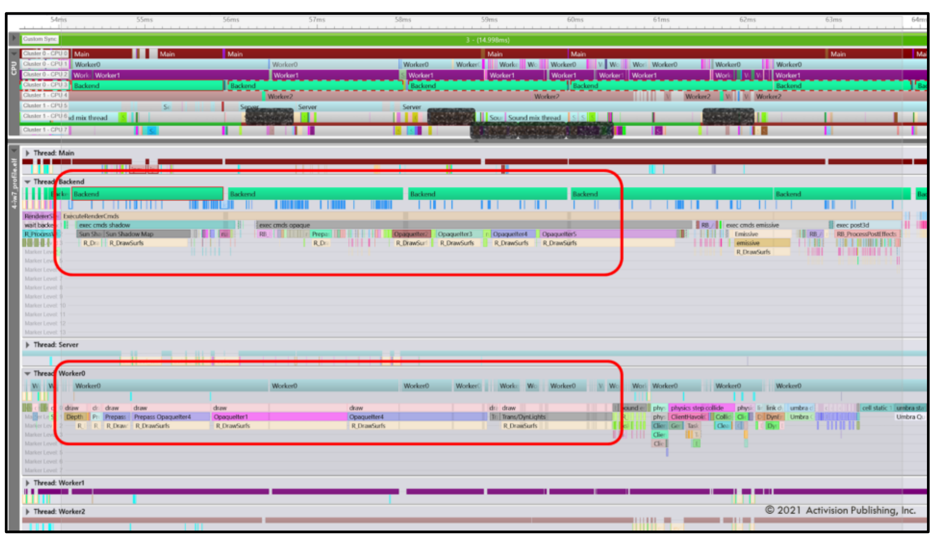


Another thing I would be remiss if I didn't bring up.
These philosophical positions exist not just because of some first principles formulation that they were considered good and valuable.
Some of them exist because of very strict platform limitations imposed in previous generations, such as the PS3.
For those who don't know, the PS3 had many small processors with very little memory and you had to be very careful in how you explicitly copied memory around, and how small your programs working set was.
But ironically this approach didn't just make the PS3 versions of our games possible.
It also made pretty much everything faster.
That's because data locality and organization matters.

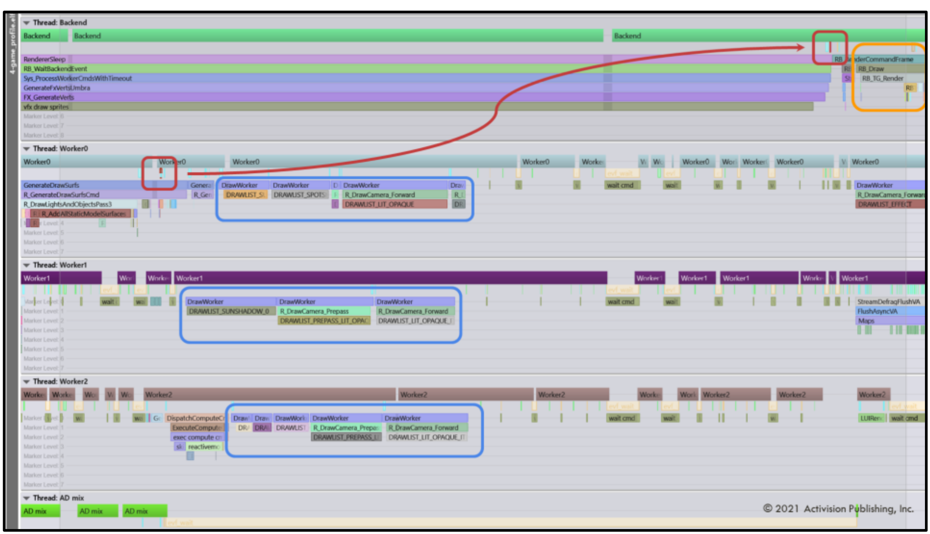
There's a quote from Frank Herbert's Dune that I sometimes think about when considering the impact Cell had on how we think about programming, at least how it shaped my approach.



And that's the end of our philosophical intermission, let's get back to the frame…

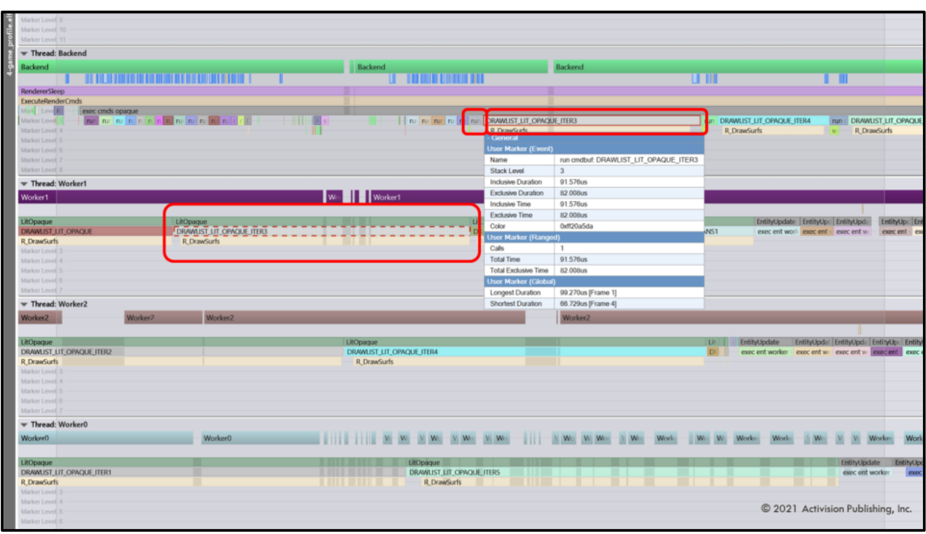


So getting back to the frame.
I mentioned before that originally the backend didn’t just submit the command buffer but was also responsible for generating it.
However in the seventh generation of consoles (Xbox 360, PS3) this became multithreaded parallel command buffer generation.
CLICK
Here you can see the “Worker0” thread picking up command buffer generation work
CLICK
While the backend is also helping and kicking.



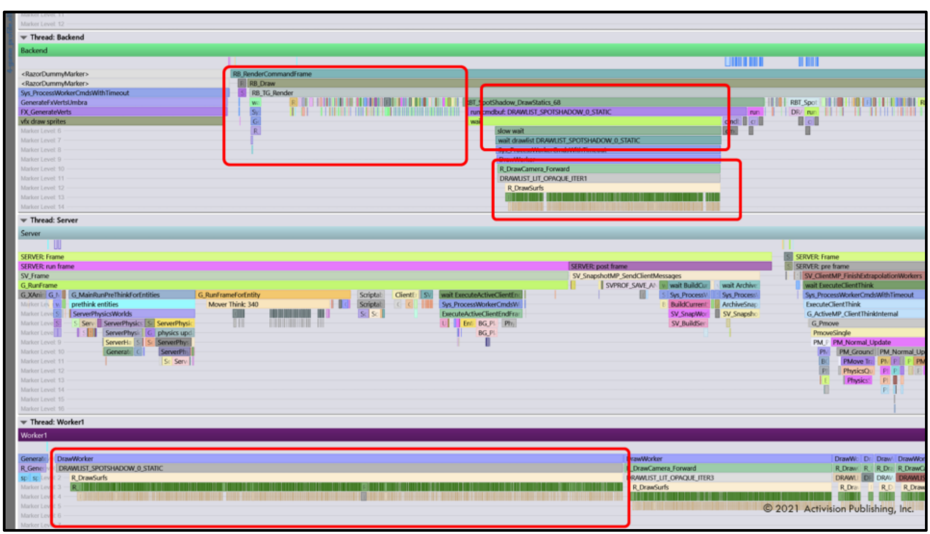
Because the scene graph traversal is divided into jobs and they can complete at a variety of times, but the backend submits work in a specific order, we want to minimize latency and bubbling by ordering our work submission and kick-off as strictly as possible.
To accomplish this the frontend/backend interchange data is actually partitioned into subsections which are fenced so that they we can see when the previous frame's backend is done reading from it, and beginning writing to it from this frame's frontend.
The worker commands that generate the command buffer packets are actually launched from the frontend as their respective scene graph traversal completes, and then these are waited on by the backend before they are kicked to the GPU.
As such, the backend does not wait for the frontend to finish processing the entire scene graph traversal, it instead kicks work opportunistically.
CLICK
So here we see that one of the frontend jobs is signalling that we have made sufficient progress in the scene graph traversal to wake up the backend, which unfortunately picked up long job, some work for us there to improve scheduling

heuristic.
CLICK
And here we can see a variety of worker commands picking up draw workers before the backend has even woken up.
CLICK
And now we’re actually getting around to kicking command buffers to the GPU on the backend, and those jobs are already done so it’s just rapid fire kicks one after the other.



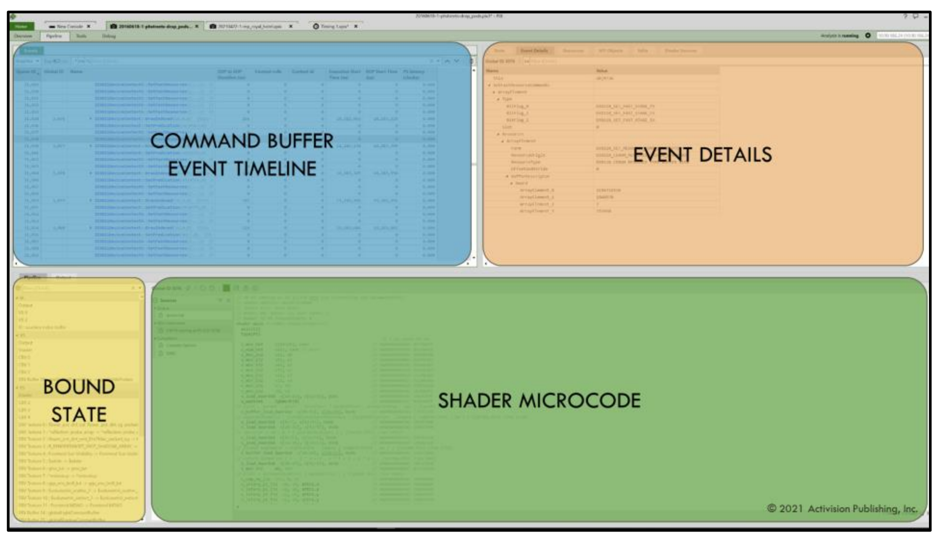
In addition, if the backend notices that draw work is coming in late, possibly due to poor scheduling behavior of worker commands, it can perform something we call handoff.
This is where we interrupt that worker command and kick what work it has completed, and then continue with the draw list on the backend itself and submit it to the GPU as quickly as possible.
This reduces GPU bubbling, that critical path I mentioned previously, and drives down overall latency.
That said, our double-buffered pipeline scheme itself introduces latency, and we have a variety of systems that attempt to control this.
CLICK
Going back to handoff and showing it on this frame.
CLICK
Here you can see worker 1 was processing the fourth segment of our opaque draw list.
Because our opaque draw list generally tends to be the longest and heaviest we break it and the corresponding prepass list into segments for better job granularity.
CLICK

This small job here is where we try to run it, or kick it to the GPU, but we find it is not ready to be submitted so we get impatient, flag it, and the worker command stops processing.
CLICK
And now here you can see that same segment picked up by the backend and then submitted directly.

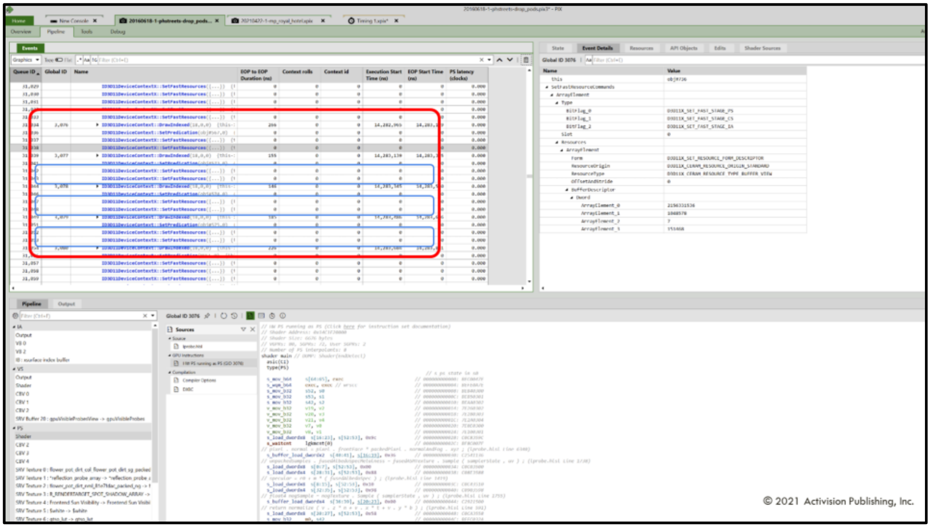


In what is now becoming a theme, as I was putting together this presentation I noticed an inefficiency in the handoff system.
CLICK
Here you can see a very nice tight set of kicks of command buffer.
CLICK
Then we get to our cached spot shadow static draws.
And we start waiting… and waiting.
CLICK
Specifically on this worker down here which is taking its time.
And so we initiate the handoff mechanism.
But interestingly it can't get the draw on the worker to interrupt!
CLICK
Instead the backend picks up a random opaque iterator so as to do some work.
I think this is due to the granularity at which we exit our tight inner draw loops, which are highly optimized for minimal

possibility of state change in the inner draw loop.
As we’ve moved to more of a bindless and multi-draw indirect approach with unified shaders, we have continued to coalesce the granularity of our inner draw loop.
We’re simply not breaking out of it frequently enough to check if we should pause our work.
Something else to have people look at later this week.

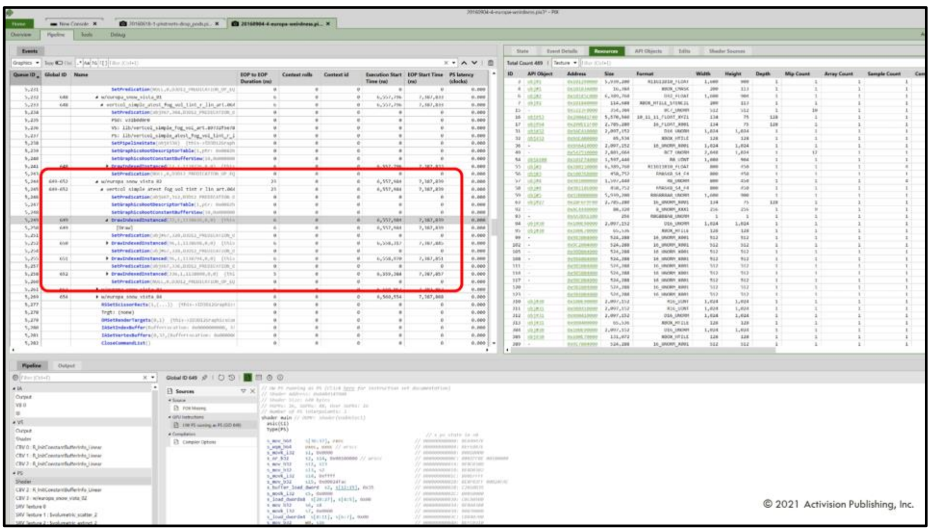


A new analysis tool appears!
For some of this command buffer generation, a quick introduction to PIX3 for those who haven't used it before.
CLICK
On the top left, a vertical list of events recorded in the command buffer such as draws, dispatches, resource sets, state sets, etc.
CLICK
On the top right, any details for that event, you can see this resource set is for a buffer object, namely an updated constant buffer set.
CLICK
Bottom left, details of what is bound on the device at the time of this event.
CLICK
And then what I used to spend a lot of time looking at, shader microcode.

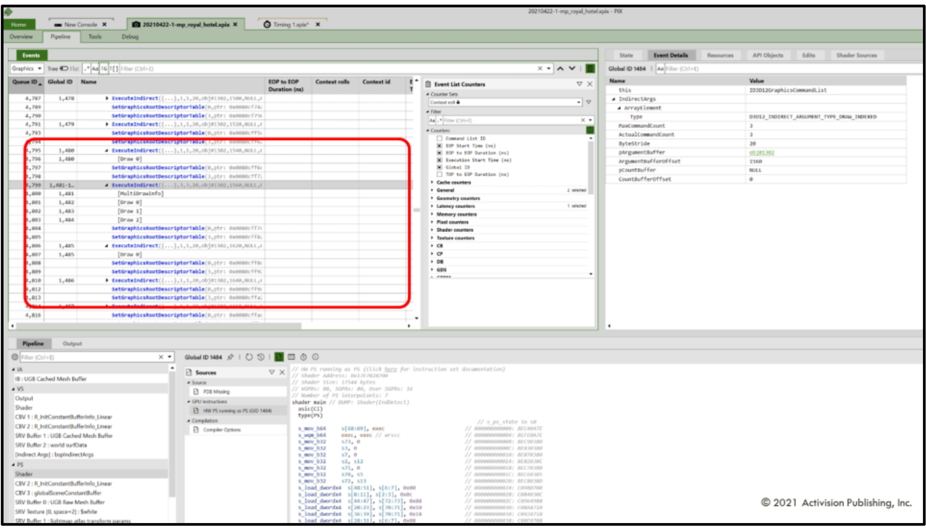


Back to our analysis tools.
I mentioned earlier that we have customized inner draw loops.
This sort of specialization has long been a feature of the renderer, the idea of having these very specific types of drawable objects.
Roughly speaking these are world, static models, dynamic models, procedural geometries such as particles, glass, and in newer titles, terrain geometries.
Each of these types has specific constraints, and also has specific draw code that goes alongside them.
In the seventh generation of consoles, the Xbox 360 and the PS3, that draw code was setup to minimize state changes, which in that era meant also just minimizing the surface area exposure to the rendering API.
We accomplished this with gross material/asset types, and then at data serialization time we would calculate a much more fine-grained key based on the various draw state requested for each material.
This key allowed us to quickly sort assets to minimize those state changes, and was a part of the frontend surface emit phase.
On the backend then this resulted in very optimized command buffer generation for those platforms.

CLICK
Here you can see just two simple constant buffer updates between some draws.
CLICK
This setup does impose a lot of rigidity, though.
For instance within the static model draw loop you couldn’t change anything except the reflection probe index and secondary lighting data index, along with the actual instance data itself.
When we went to later add vertex-based baked lighting, and then lightmapped static models, this meant plumbing through a lot of code for those specializations.



This is a similar setup for our world draws, albeit in D3D12 now, to give you a slightly different API flavor.
CLICK
Here you can see there is nothing between the draws but predication--our world materials are in, well, world space, so they don't even need transforms from local.
The supported modulation of reflection probe and lightmap index don't come into play so it's just pumping out draws.



Over the previous, eighth generation of hardware we had to adjust our notion of sorting to reflect the underlying hardware realities of the consoles.
For the GCN GPUs this meant understanding the importance of context rolls and how for instance vertex and pixel shader state was not part of a draw context and now shouldn’t be a primary part of the sort key.
Although there are values to minimizing those roles--I$ was a surprising impact to performance, around 250us per frame once we had precaching enabled.
But the larger point here of course is how we reconsider things through the years as hardware and software approaches change.
CLICK
For instance this is how a chunk of our draw loop looks now, which is all indirect execution with multi-draw.

PRIMARY / RING ALLOCATOR, OWNED BY BACKEND THREAD

CHUNK 0 CHUNK 1 CHUNK 2 FENCE 0 CHUNK 3 CHUNK 4 CHUNK 5 FENCE 1

LINEAR ALLOCATORS, OWNED BY PER FRAME BACKEND DATA / MULTIPLE THREADS

CHUNK 0 CHUNK 1 CHUNK 2 CHUNK 3 CHUNK 4 CHUNK 5 CHUNK 6 CHUNK 7 STOP

WORKER 0

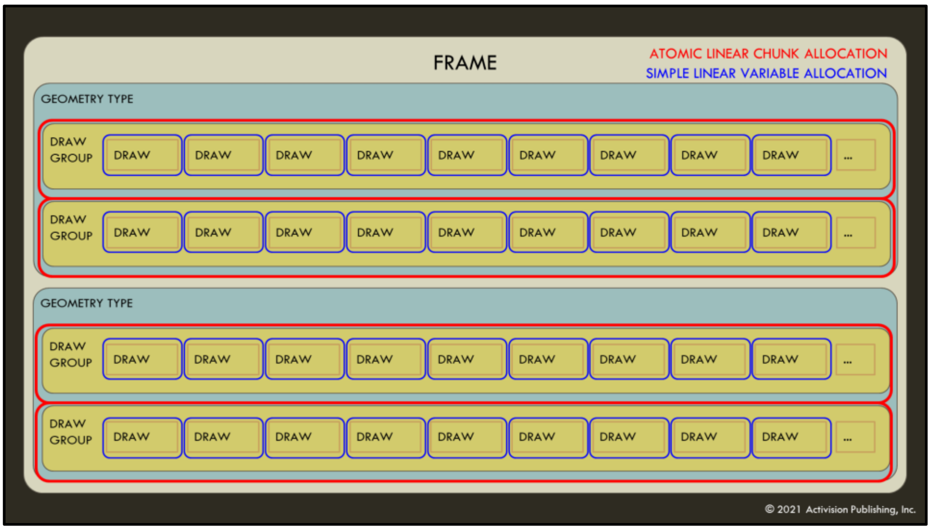
CHUNK 0 CHUNK 2 CHUNK 6

WORKER 1

CHUNK 1 CHUNK 3 CHUNK 4 CHUNK 5 CHUNK 7

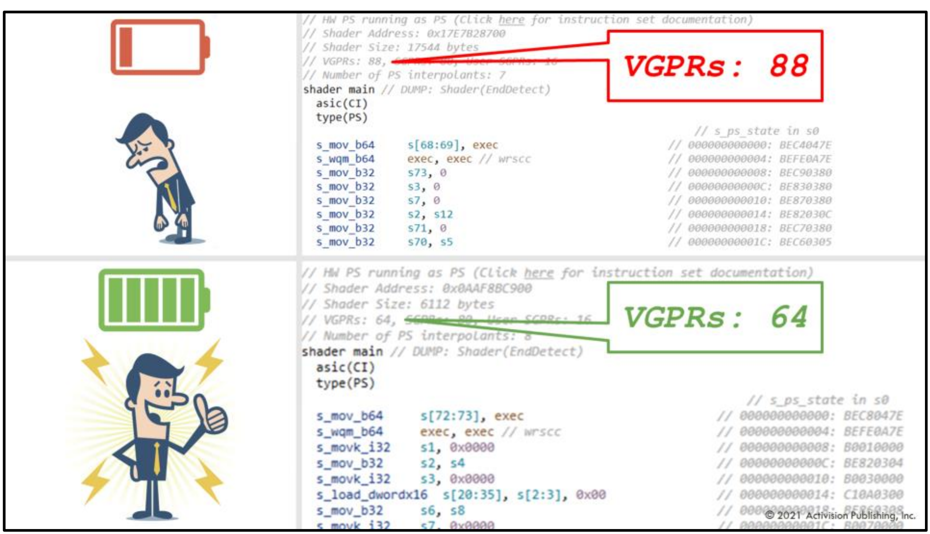
© 2021 Activision Publishing, Inc.

I mentioned our approach to memory allocation previously and these tight draw loops touch on this as well.
In the core draw loops we may need to peel some memory for constant buffer allocation (and command buffer as well, but that's another story).
However we don't want to do anything expensive at per-draw granularity.
And we don't want complicated individual allocation schemes, either, such as synchronized multi-threaded heaps.
For our approach we have two underlying allocation idioms.
We maintain a primary ring buffer for constant allocation which is fenced to retire chunks and always allow forward progress by the GPU.
However the vast majority of our allocations go through linear allocators that are contained within each of the 'backend data' structures we discussed earlier.
CLICK CLICK
Having that ownership aligned with the 'backend data' allows us to avoid any sort of complicated synchronization with the GPU as we already fence on 'backend data' granularity.



In order to avoid expensive per draw costs we reserve chunks from the allocator at a higher level using standard atomics.
CLICK
These chunks are assigned to each of the parallel command buffer generation threads as they need them.
And because that ownership is then per-thread we can simply increment non-atomically within that chunk to each draw as needed.
CLICK
The size of the chunk is based on a calculation of the maximum per-draw reservation multiplied by a limit on the number of surfaces we'll iterate on before breaking the loop.
We can validate this value offline during shader serialization by noting the maximum size of constant buffer references.
So we know that per-draw allocations will never fail, but that they may fail at a higher level if we exhaust the linear allocator.
However at that level we can also safely exit our draw loops, which leads to improved resiliency in the renderer.
And if we're super clever, we can use the handoff mechanism we discussed previously to continuing drawing on the backend thread using the ring allocator which can always advance.

One of the downsides of this approach is internal waste/fragmentation if the surface count within the group is low. But that's also indicative of other problems as well in terms of draw grouping efficiency.

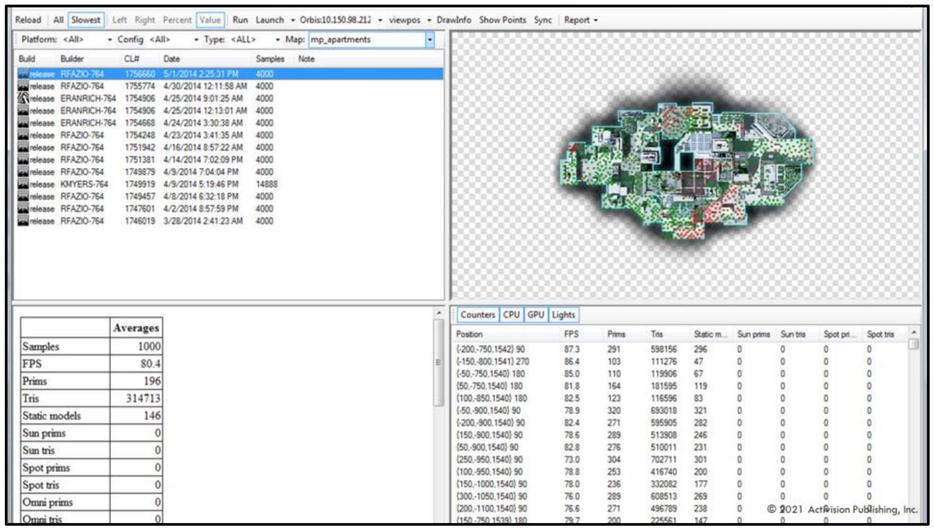


I wanted to go back to shader microcode and talk about our approach to building shaders and why it is the way it is.
This slide is basically the shared trauma for those of us who spent the past ten years doing microcode optimization.
The GPU architecture in the eighth generation of consoles were sensitive to what we call occupancy.
Which is essentially the number of concurrent workloads that could be processed in-flight at any point in time.
The underlying hardware had a register file which was split between varying and scalar registers depending on whether values were uniform or not across a wavefront.
The shader compilers didn’t always make optimizing for this easy, we spent enormous amounts of time analyzing how to improve register usage.

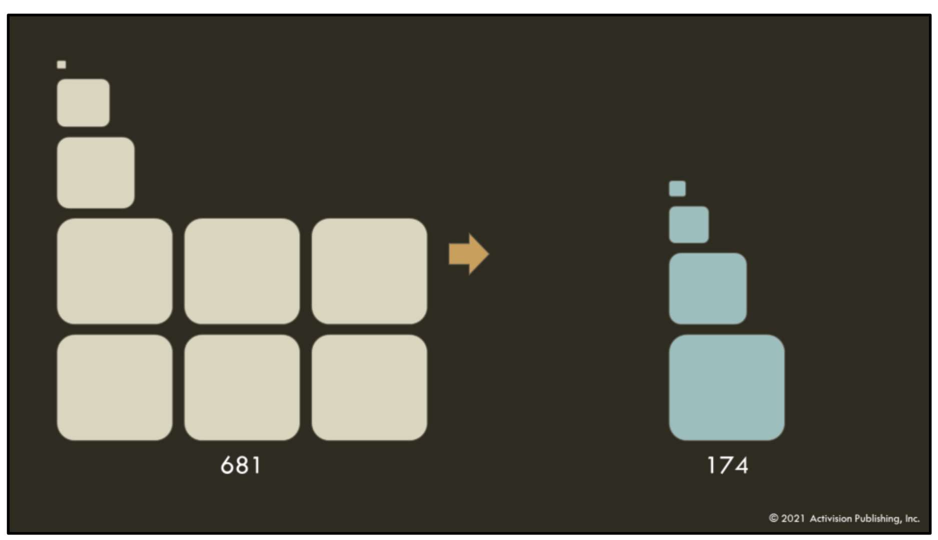
```
void LIGHTS_ITERATE( const LocalAttributes localAttributes, const LIGHT_ITERATOR_INPUT_TYPE lightingInput, inout LIGHT_ITERATOR_OUTPUT_TYPE lightingOutput
{
#if TOOLSGFX
#if USE_DIR_LIGHTING
    Lights_Iterate_SingleSun( localAttributes, float3( 0.0f, 0.0f, 0.0f ), lightingInput, lightingOutput );
#endif // #if USE_DIR_LIGHTING
#if USE_PRIMARY_LIGHTING
    uint3 cullIndex = CullResultBaseIndex( localAttributes.svPosition.xy, localAttributes.svPosition.w );
    for ( uint lightBatchIndex = 0; lightBatchIndex < gScene.numLights; lightBatchIndex += GFX_LIGHT_CULL_WORD_BITS )
    {
        GFX_LIGHT_CULL_WORD_TYPE bits = gLightCullResults[cullIndex + uint3( 0, 0, lightBatchIndex / GFX_LIGHT_CULL_WORD_BITS )];
        FOR_EACH_BIT_UNIFORMLY( bits, uniformBit )
        {
            uint lightIndex = lightBatchIndex + uniformBit;

            PackedLightConstants light = LoadPackedLightProperties( lightIndex );
            LocalLightAttributes lightAttributes = PhysicallyBased_GetSpotOrOmniLightAttributes_PackedLight( localAttributes, light );

            float3 shadowLookupPos = lightAttributes.samplePosition;

            #if USE_SHADOW_OFFSET_LIGHT_DIR
                float3 receiverNormal = -lightAttributes.lightDir;
            #else // #if USE_SHADOW_OFFSET_LIGHT_DIR
                float3 receiverNormal = localAttributes.vertexNormal;
            #endif // #else // #if USE_SHADOW_OFFSET_LIGHT_DIR

            float2 unusedTraceParams;
            float visibility = SpotShadowMapLookup( localAttributes.svPosition.xy, false, shadowLookupPos, receiverNormal, light.shadowIndex, lightAttribu
            lightAttributes.lightVis *= lerp( 1.0f, visibility, lightAttributes.shdFalloff );

            [branch]
            if ( lightAttributes.lightVis > 0.0f )
            {
                LIGHT_ITERATOR_CALLBACK( localAttributes, lightAttributes, lightingInput, lightingOutput );
            }
        }
        FOR_EACH_END
    }
```

© 2021 Activision Publishing, Inc.

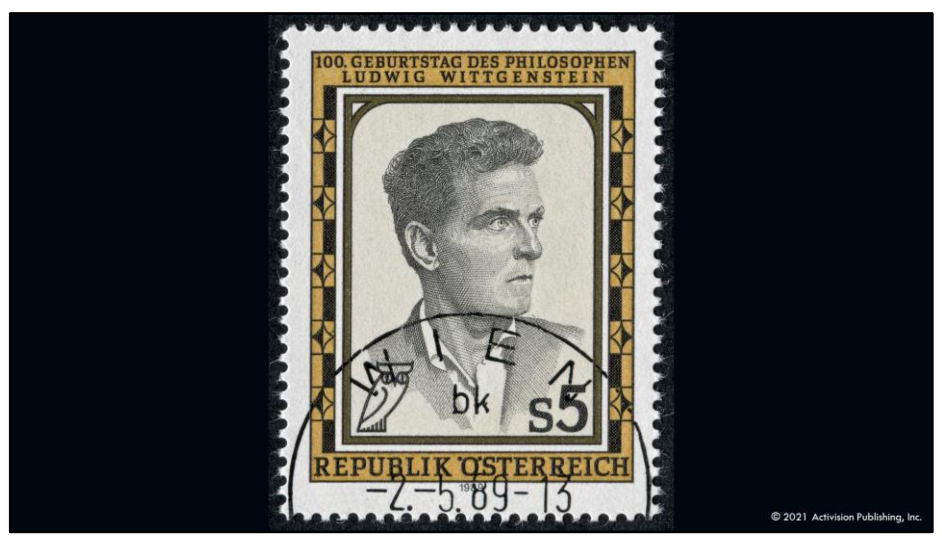
Hitting that magical four-wave occupancy at 64 VGPRs was a holy grail for us for our forward+ opaque shaders.
CLICK
To achieve this we spent a significant amount of time hand-building and structuring our lighting loops, decal loops, etc., with bespoke scalarization tricks.
But we also have material composition like any moderately complex rendering system.
CLICK
Many people pursue graph systems but we still rely on essentially textual processing with preprocessor directives to build all of our shaders.
Our tight rein on performance has never made us comfortable with permitting the full level of arbitrary expressiveness in a graph system, even at the cost of iteration.
And because we are shipping games and not engines we can spend the time on this level of specialization and optimization.



But any permutation system becomes susceptible to combinatoric explosion at some point.
On a previous title for instance we peaked at around 400k unique pixel shaders for a given map.
Tracking performance across all of those is impossible, and we pursued several approaches.
Automation. Telemetry system where shader statistics are recorded per CI run and then diffs are produced, can examine these from data builds, but also for local changes you can queue preflight checks and view reports
This is our tool, fpsTool, that takes thousands of samples in real game levels and builds aggregate performance data.
In what is a massive defeat for our industry's dignity and self-respect, we also brought back infrastructure to explore shader compiler tunable space via exhaustive compilation and custom performance evaluation heuristics.
Some of you will have gone through this with nvshaderperf back in the PS3 era and random scheduler seed permutation.



We also aggressively pursued static reductions of our techniques which subsequently led to reductions in permutations.
We did this via careful folding of functionality into static/scalar branches in our base forward+ shaders.
On some olders titles we had 681 separate techniques, and on more recent ones we got that down to 174.



© 2021 Activision Publishing, Inc.

Our final philosophical interlude.
In the seventh generation of consoles, the Xbox 360 and the PS3, I felt like, as a systems and rendering programmer, our primary work was to learn to maximize CPU parallelism.
And this especially manifested, at least for us, in parallel command buffer generation.
In the eighth generation of consoles, it felt like our primary work was in understanding how to generate efficient GPU microcode to exploit GPU parallelism.
We spent a lot of time working with shader compilers and learning how to coax them to generate optimal microcode.
What will be our primary legacy for the ninth generation of consoles?
Will it be a solution to the practical problems of raytracing? That feels sort of obvious and banal.
Maybe something else entirely? I'm curious as to what you think.
[WITTGENSTEIN]

To wrap up. When I think of the benefits we try to exploit because of our global knowledge of the use of our technology, they are that our systems are:
CONSTRAINED.
Many optimization opportunities come from the careful choice of constraints and thus can be SPECIALIZED.
Much of our lighting can be pre-calculated offline because our world is largely static.
Our game logic can operate at a decoupled rate because we always operate in a client-server model.
Choosing the right constraints is one of the most important things for you to do architecturally.
VALIDATED.
As workloads and underlying platforms change you need to revisit your choices and validate they're still the right ones.
Good historical record-keeping helps inform that validation and the work you're doing today.
PRIORITIZED.
Prioritize your work by its exposure to the user, and make the effort to napkin math the impact.
EFFICIENT.
Think of optimization as a minimization exercise for energy through a system.
CORRECT.

Engineer systems for simplicity of ownership of memory to better reason about synchronization.
RESILIENT.
Encode your assumptions at the appropriate level of operation and do the work to ensure robustness.

Thanks everyone, I hope you enjoyed the talk.
Also some call outs here from folks at Activision as well as at Sony and Microsoft who helped permit this presentation to be as good as I wanted it to be.

REFERENCES
[HARGREAVES19] D3D12 Background Processing - Spec, Shawn Hargreaves, 2019
© 2021 Activision Publishing, Inc.