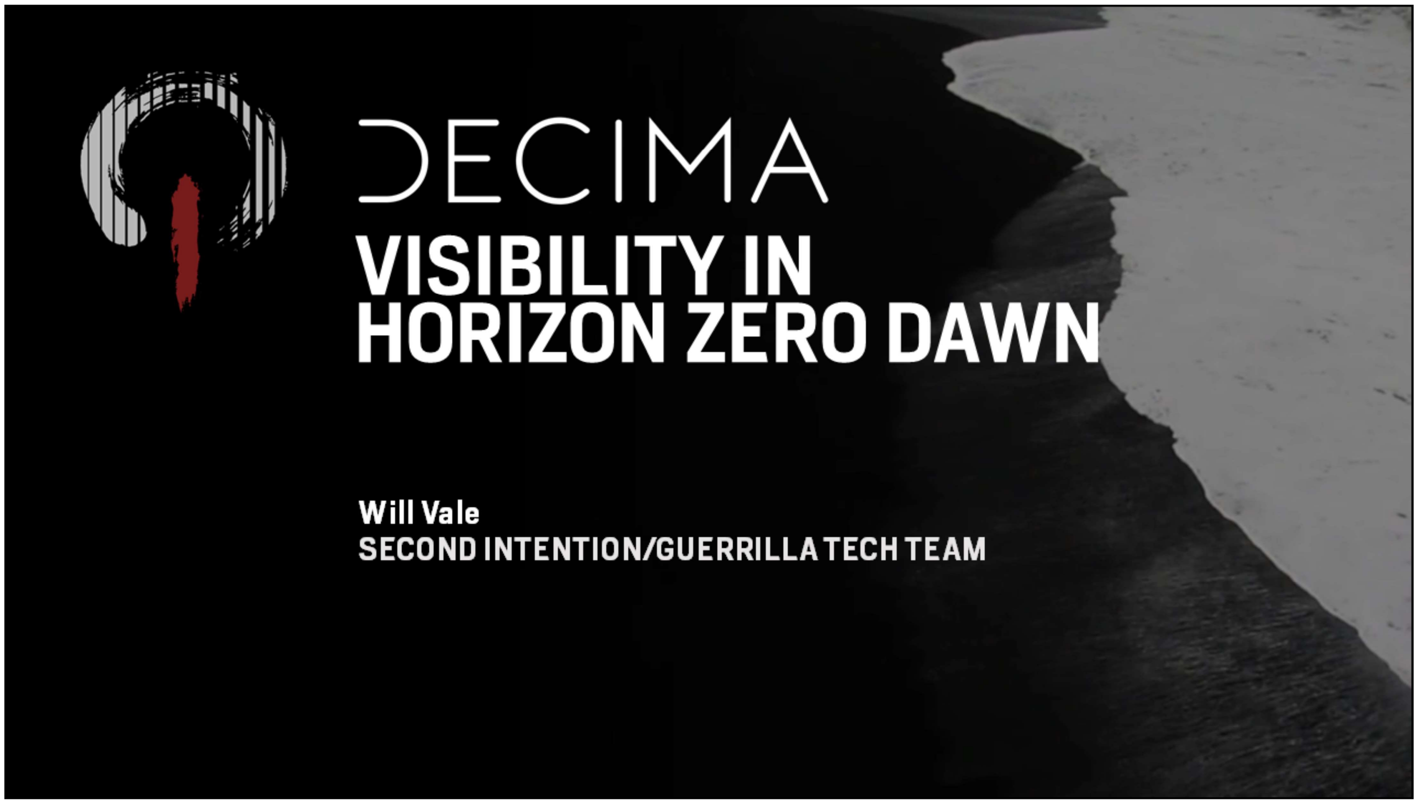


DECIMA

VISIBILITY IN HORIZON ZERO DAWN

Will Vale
SECOND INTENTION/GUERRILLA TECH TEAM

INTRODUCTION

About me

DECIMA

Making games since 19(ZX)81

Getting paid for it since 1998

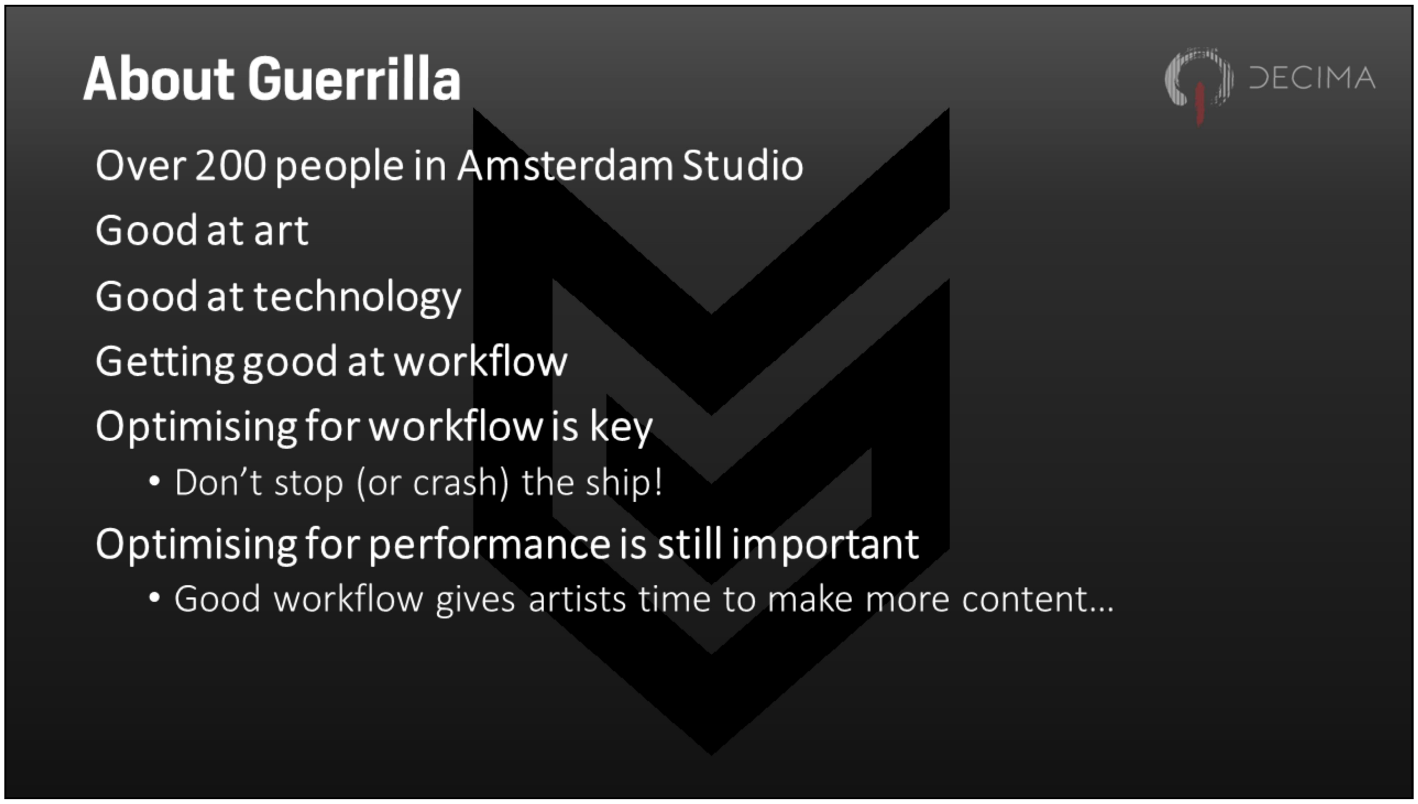
Moved to New Zealand in 2004

Contracting for Guerrilla since 2005, mostly full time

Sinclair ZX81 Personal Computer-
the heart of a system
that grows with you.
Kit: £49.95
Built: £69.95
16K-byte RAM pack for massive add-on memory.
Available now-the ZX Printer for only £49.95
sinclair ZX81

character arc

Hi, I'm Will Vale and I've been making games for a while. I worked in the UK for several years at Particle Systems in Sheffield, and started contracting for Guerrilla fairly soon after emigrating to this hemisphere.



About Guerrilla

Over 200 people in Amsterdam Studio

Good at art

Good at technology

Getting good at workflow

Optimising for workflow is key

- Don't stop (or crash) the ship!

Optimising for performance is still important

- Good workflow gives artists time to make more content…

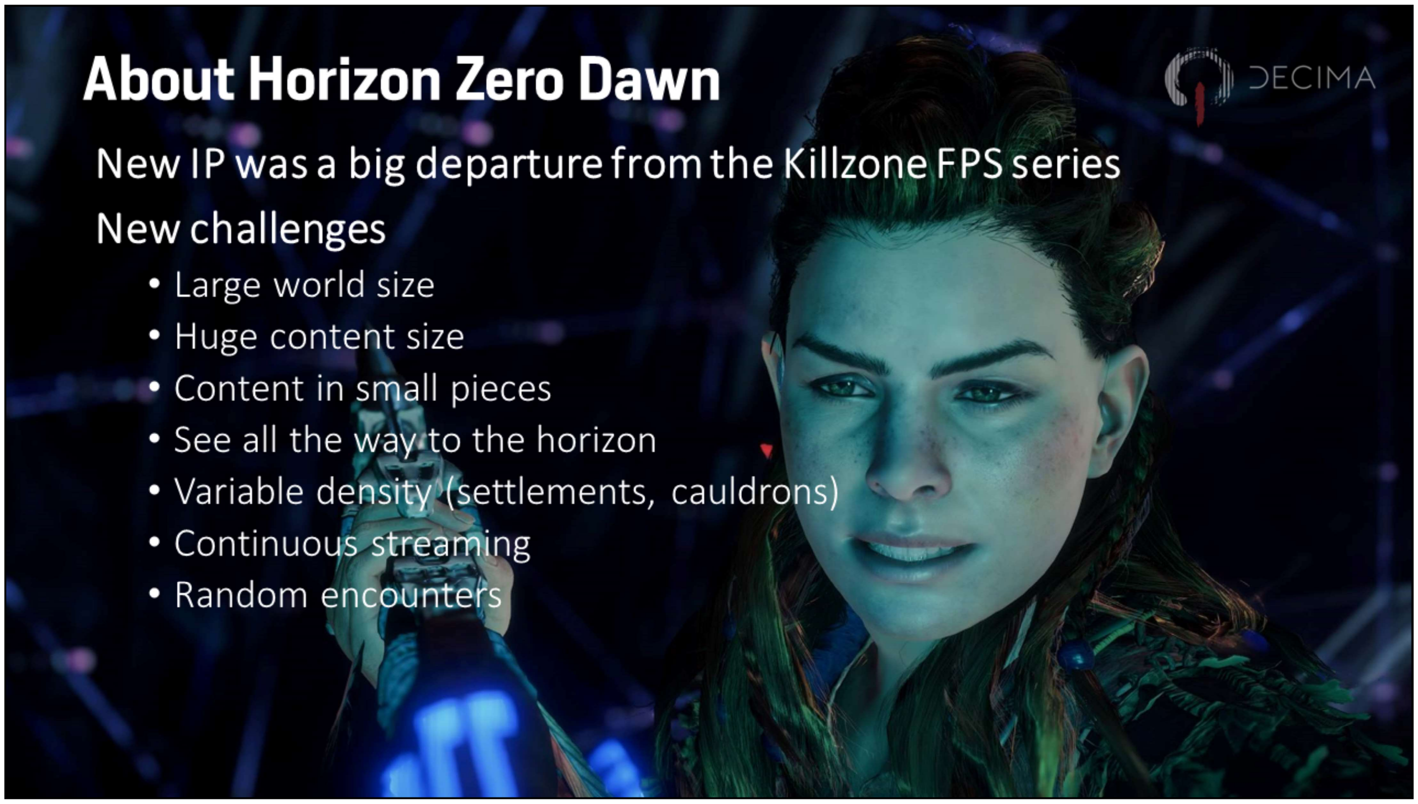
DECIMA

Guerrilla is a first party Sony studio that's been traditionally known for really amazing art and technology. While we haven't always had the most efficient workflow we've paid a lot of attention to that over the development of KZ4 and Horizon and we're getting good at it.

Because the studio is big, workflow is probably the most important optimisation critierion. Any wasted time is scaled up by the large team size.

But performance is important too, since good workflow tends to mean more and better content for the engine to deal with.

Our in-house engine is Decima, it's been used to create the Killzone series of games across multiple platforms, and Horizon and Kojima Productions' Death Stranding on Playstation 4.



About Horizon Zero Dawn

DECIMA

New IP was a big departure from the Killzone FPS series

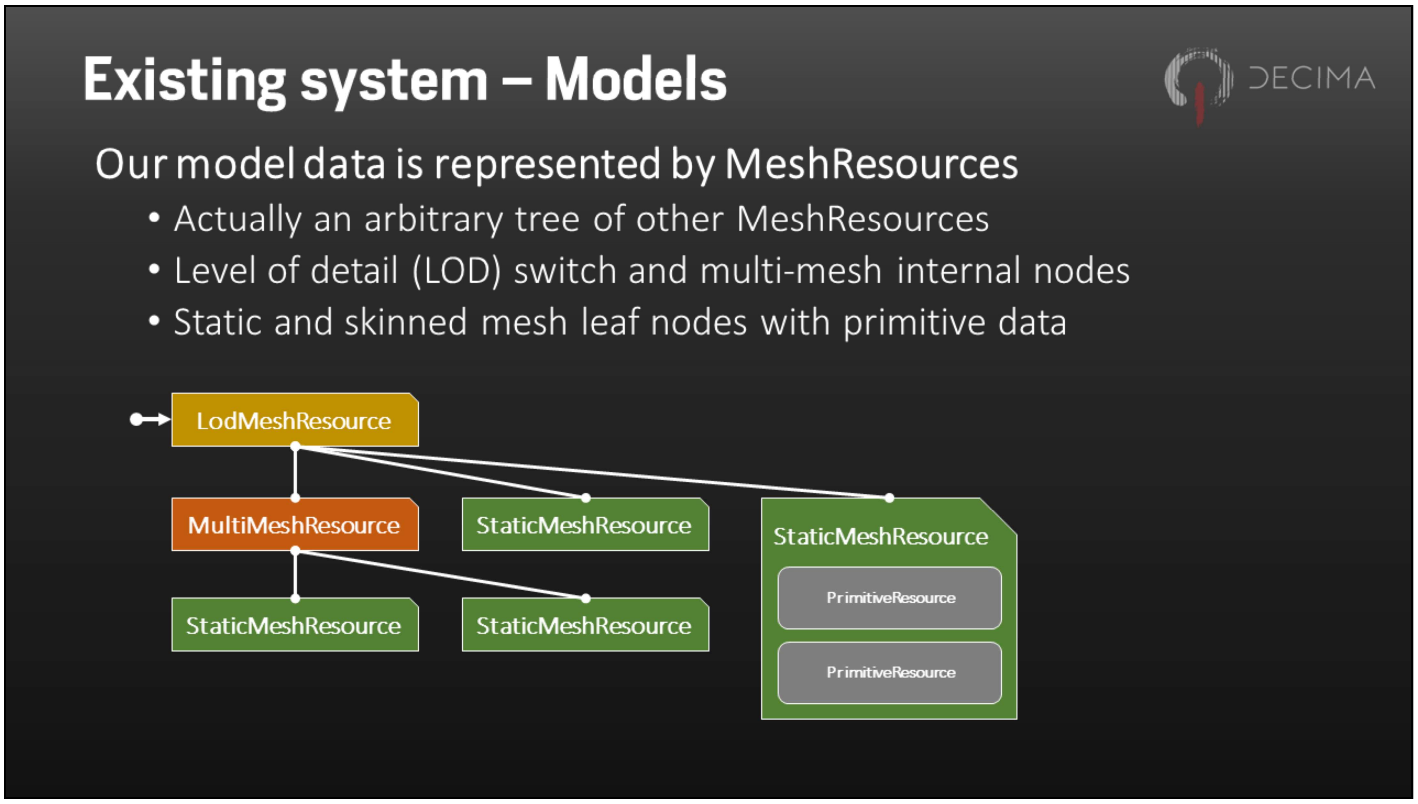
New challenges

- Large world size
- Huge content size
- Content in small pieces
- See all the way to the horizon
- Variable density (settlements, cauldrons)
- Continuous streaming
- Random encounters

The Killzone games are relatively linear first person shooters with narrower areas connecting wider play spaces. Horizon was a huge departure from this – we wanted a large open world full of detail, and this obviously created some challenges for the engine. The variable content density and the need to keep headroom for spontaneous enemy encounters made for an interesting problem.

BACKGROUND

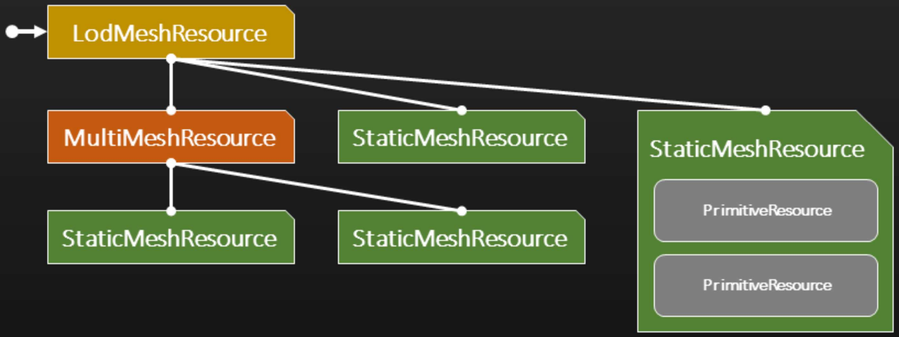
Before talking about what we did, I'll talk a little about where we started from with Horizon's visibility.



Existing system – Models

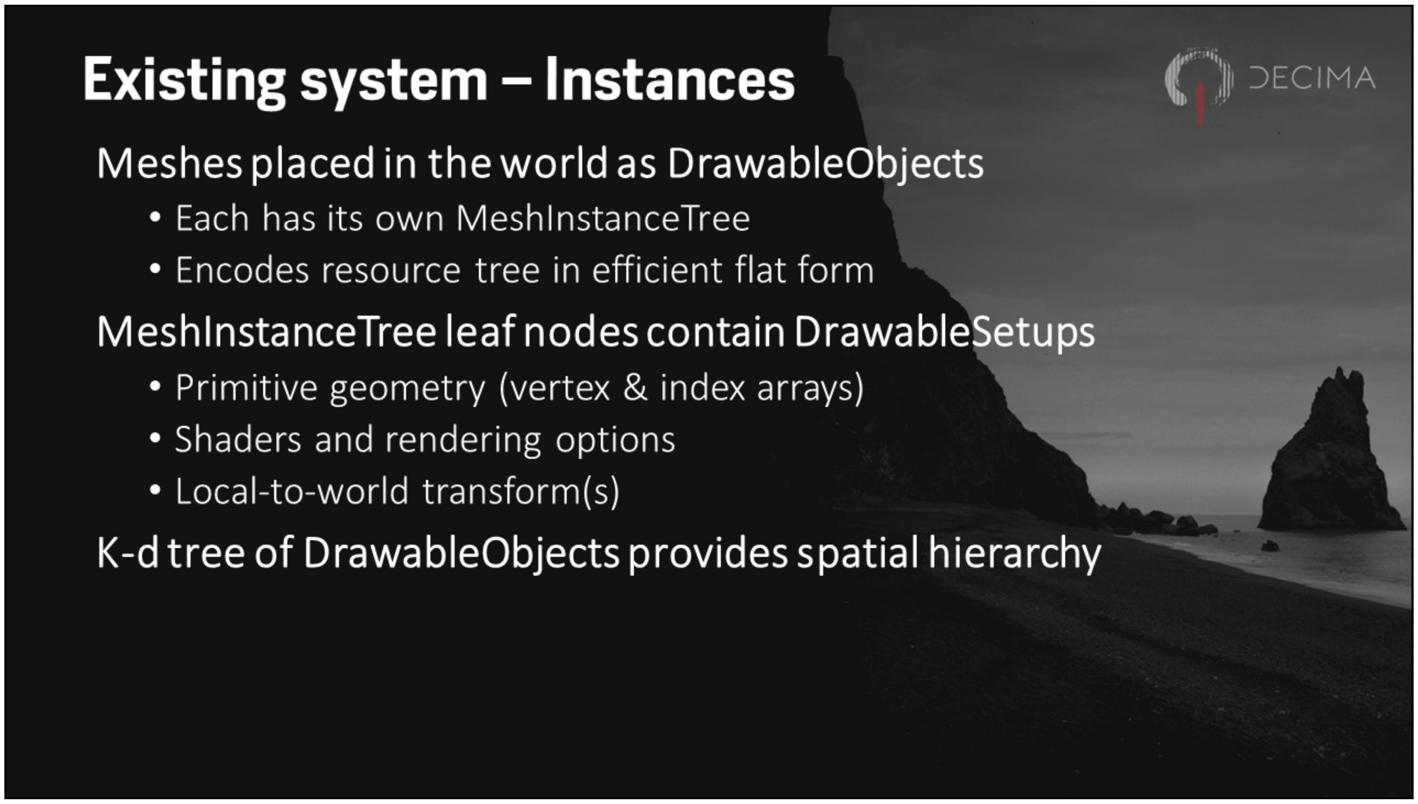
Our model data is represented by MeshResources

- Actually an arbitrary tree of other MeshResources
- Level of detail (LOD) switch and multi-mesh internal nodes
- Static and skinned mesh leaf nodes with primitive data



Our 3D models as seen by the game are a tree of MeshResources which can be arbitrarily complex. We have Lod meshes which select between different branches, and Multimeshes to place branches relative to the root. And at the leaves we have static meshes containing the geometry. There are other leaf node types but they’re not relevant to this talk.

It’s a very flexible system, which is good for building different content workflows, but makes it harder to nail down and extract typical behaviours when optimising.



Existing system – Instances

Meshes placed in the world as DrawableObjects

- Each has its own MeshInstanceTree
- Encodes resource tree in efficient flat form

MeshInstanceTree leaf nodes contain DrawableSetups

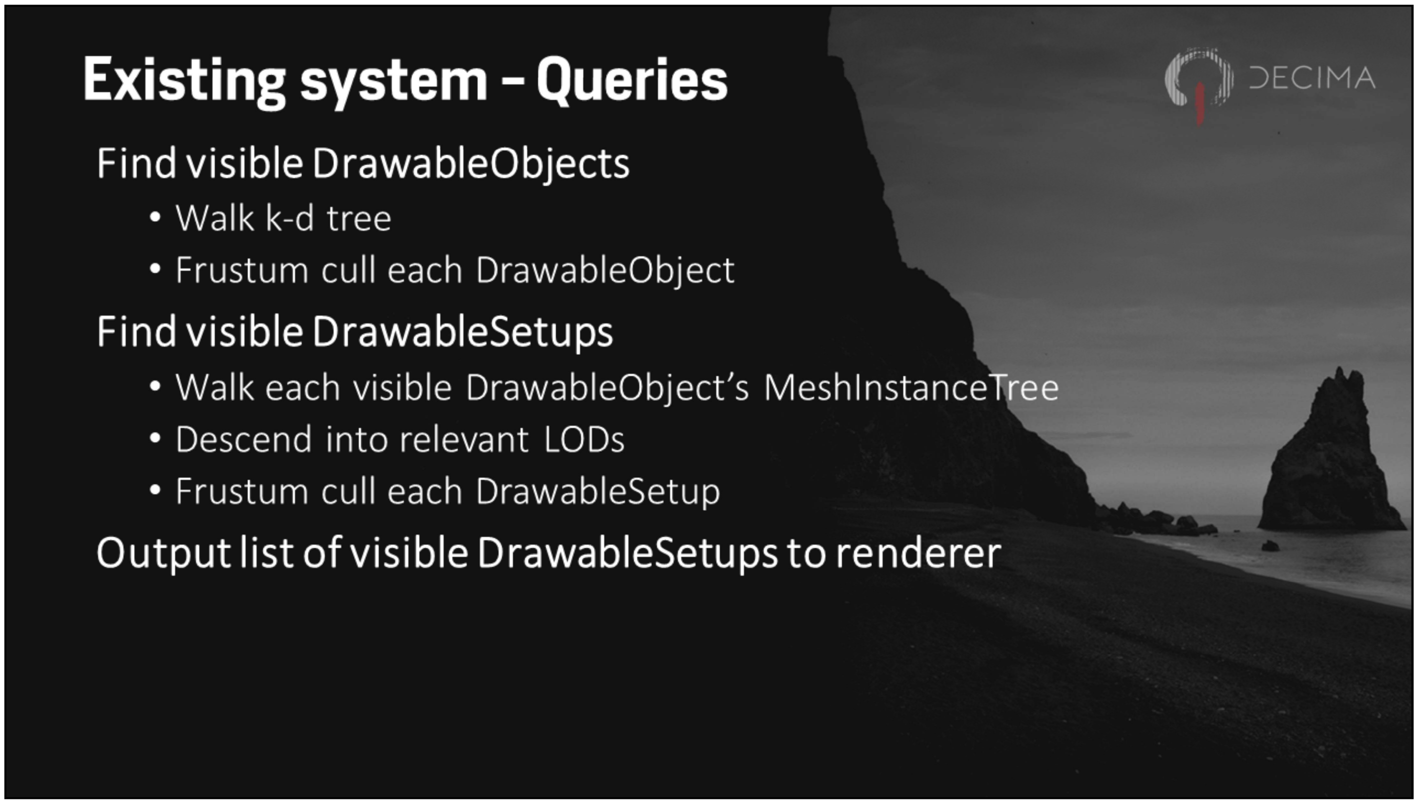
- Primitive geometry (vertex & index arrays)
- Shaders and rendering options
- Local-to-world transform(s)

K-d tree of DrawableObjects provides spatial hierarchy

We place instances of mesh resources in the world as DrawableObjects. These have a more efficient encoding of the MeshResource tree which is used at runtime.

One particular issue is that no mesh resource knows that it's the root of a tree, it's only when placed by a DrawableObject that this is made clear.

The leaf nodes of this tree contain DrawableSetups, which are what we feed the renderer. Each is a chunk of geometry with the shaders (and state) needed to render it. They isolate the renderer from the rest of the content.



Existing system – Queries

Find visible DrawableObjects

- Walk k-d tree
- Frustum cull each DrawableObject

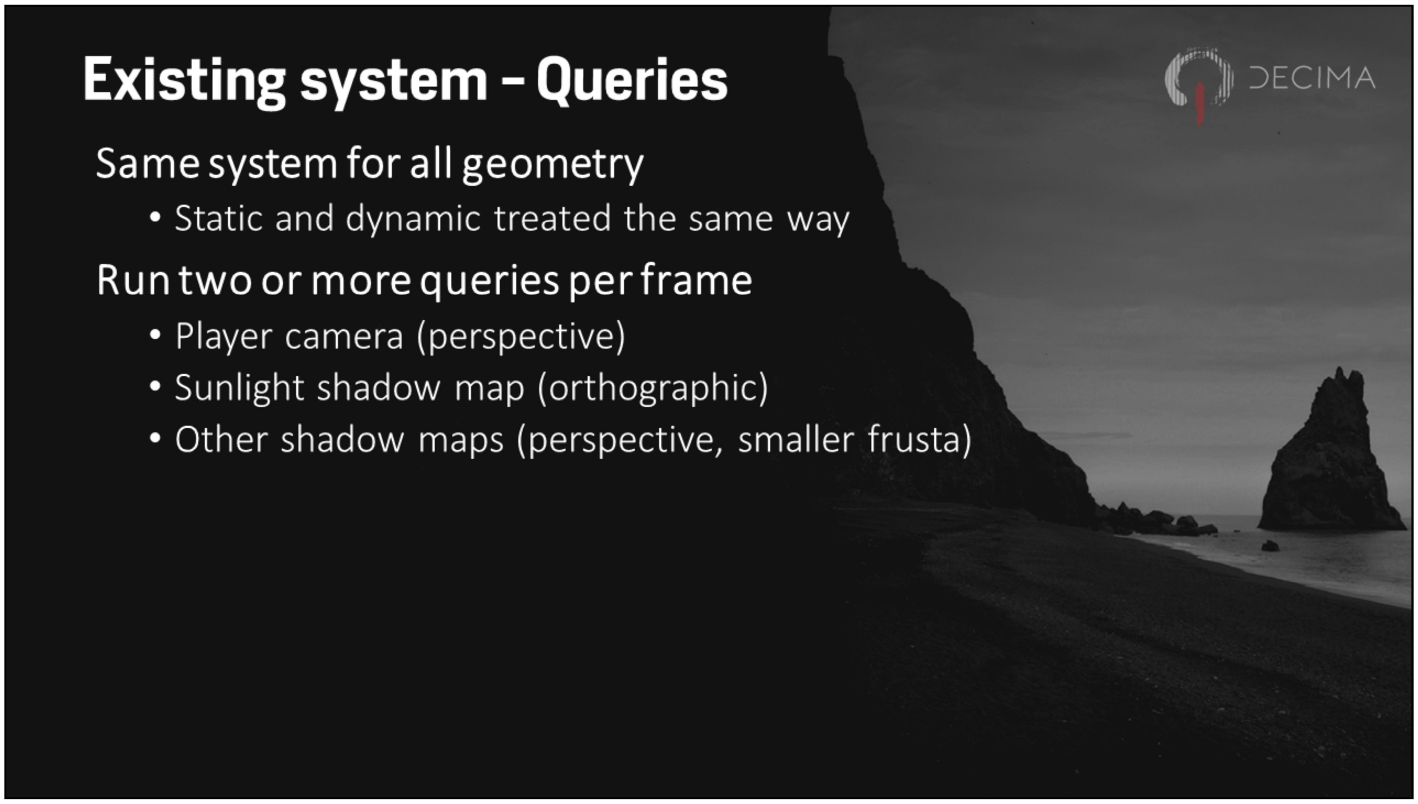
Find visible DrawableSetups

- Walk each visible DrawableObject’s MeshInstanceTree
- Descend into relevant LODs
- Frustum cull each DrawableSetup

Output list of visible DrawableSetups to renderer

In KZ3 we also occlusion-culled each object and setup using software occlusion culling on the PS3 SPUs.

In KZ4 we used middleware for static content, which handled all visibility including occlusion.



Existing system – Queries

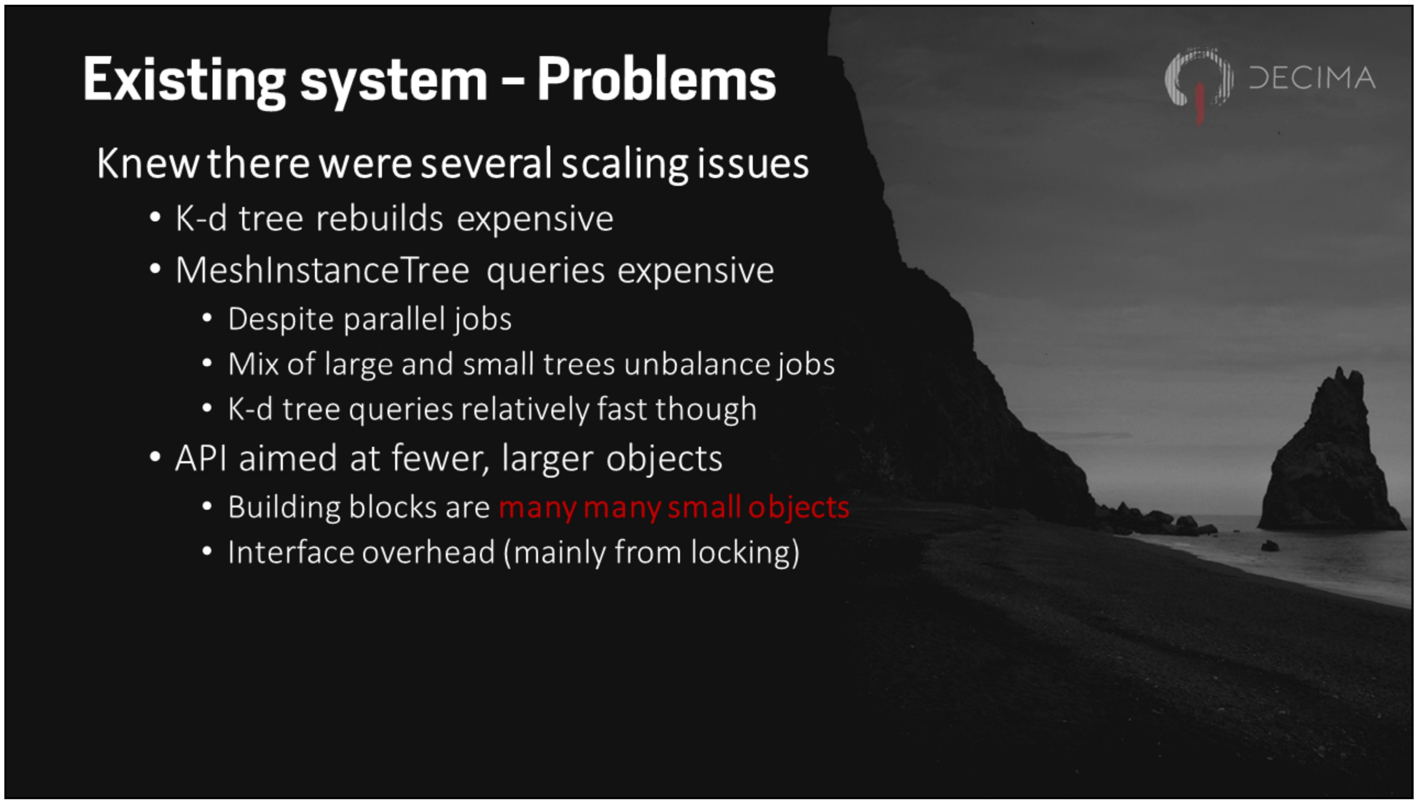
Same system for all geometry

- Static and dynamic treated the same way

Run two or more queries per frame

- Player camera (perspective)
- Sunlight shadow map (orthographic)
- Other shadow maps (perspective, smaller frusta)

Queries are CPU jobs which are part of the general rendering job graph. We have a flexible job architecture on the CPU – most code (and all rendering code) runs in jobs.



Existing system – Problems

Knew there were several scaling issues

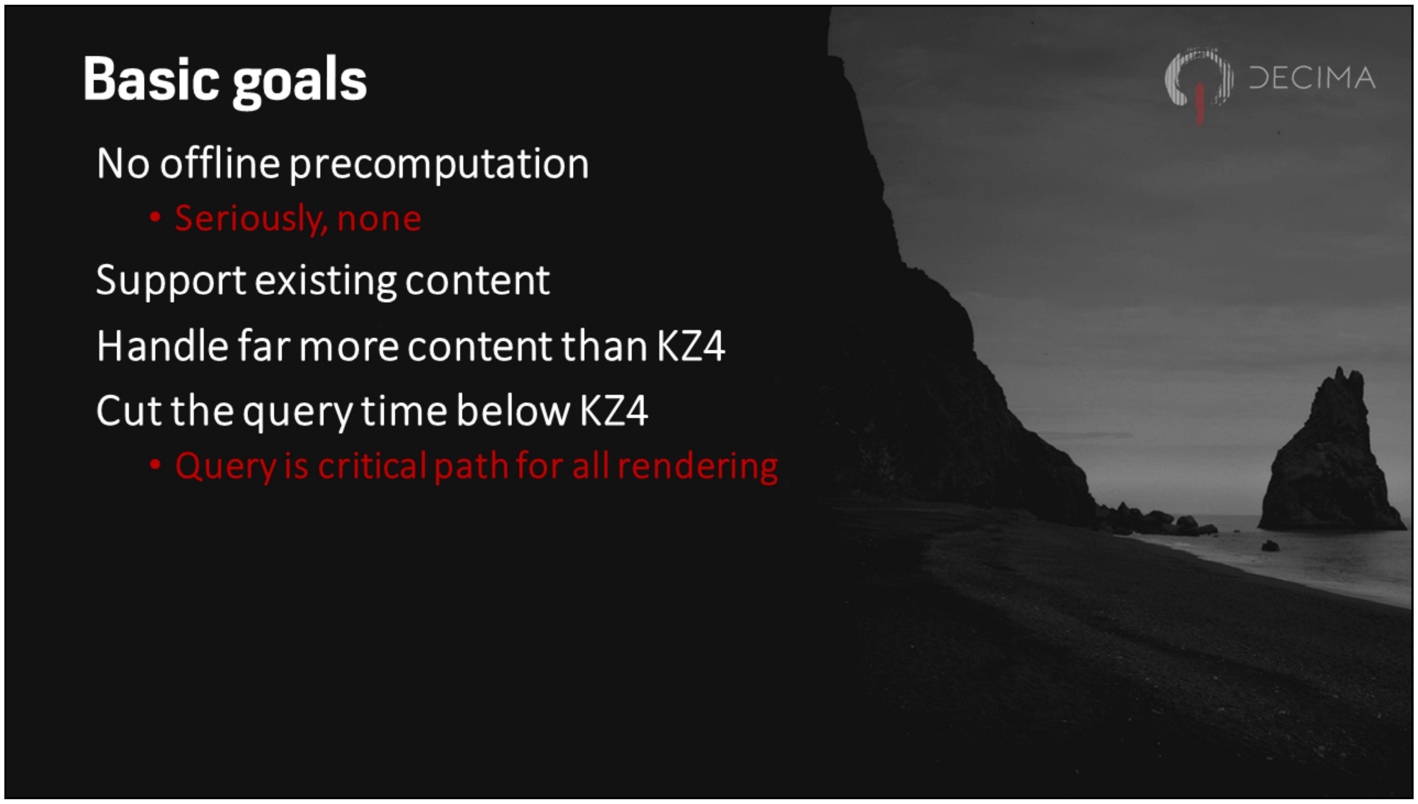
- K-d tree rebuilds expensive
- MeshInstanceTree queries expensive
  - Despite parallel jobs
  - Mix of large and small trees unbalance jobs
  - K-d tree queries relatively fast though
- API aimed at fewer, larger objects
  - Building blocks are many many small objects
  - Interface overhead (mainly from locking)

This system wasn't good enough for the world size of Horizon, we needed to reduce the complexity of the Kd-Tree since we'd be rebuilding it often as content streamed, and we wanted to run fewer MeshInstanceTree queries since these were expensive.

We also had the problem that the scene API was really aimed at few, large objects, and the building blocks which create Horizon's world are a vast number of tiny objects.

NEW SYSTEM

So what did we build instead?



Basic goals

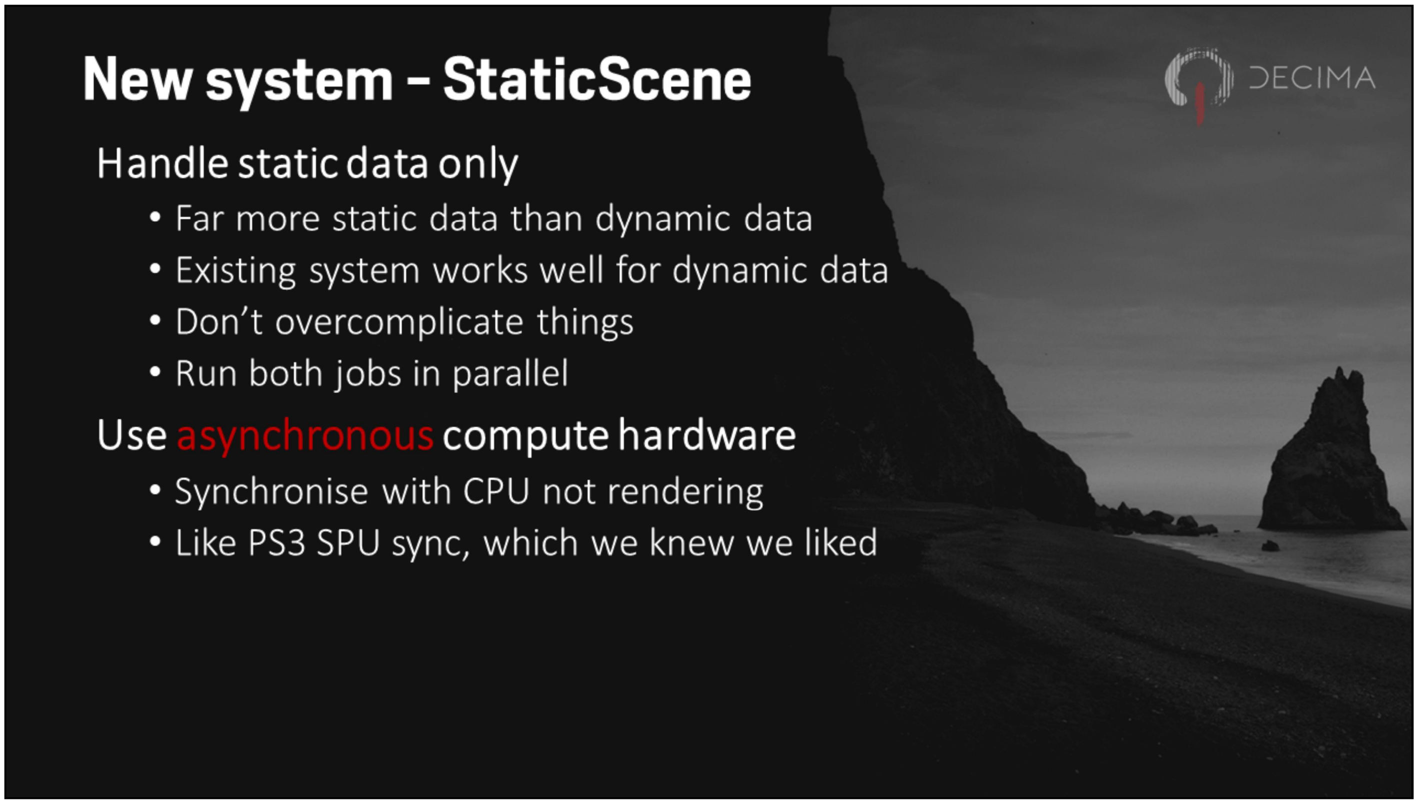
DECIMA

- No offline precomputation
  - Seriously, none
- Support existing content
- Handle far more content than KZ4
- Cut the query time below KZ4
  - Query is critical path for all rendering

We had visibility middleware in KZ4 which caused workflow problems as we needed to precompute visibility data and this just didn't fit into how we were working. It increased the time needed to properly test changes – we could start the game without it, but then artists had to wait for the baking process to finish in the background before they could see things as they should be. We still shipped the game with it, but we didn't think we could fit it into Horizon.

I found it quite hard to give up the idea of precomputation, but it is liberating – if you don't have to wait for baking processes you can test code changes quickly as well.

Obviously we also needed to handle new content and handle it faster.



New system – StaticScene

Handle static data only

- Far more static data than dynamic data
- Existing system works well for dynamic data
- Don't overcomplicate things
- Run both jobs in parallel

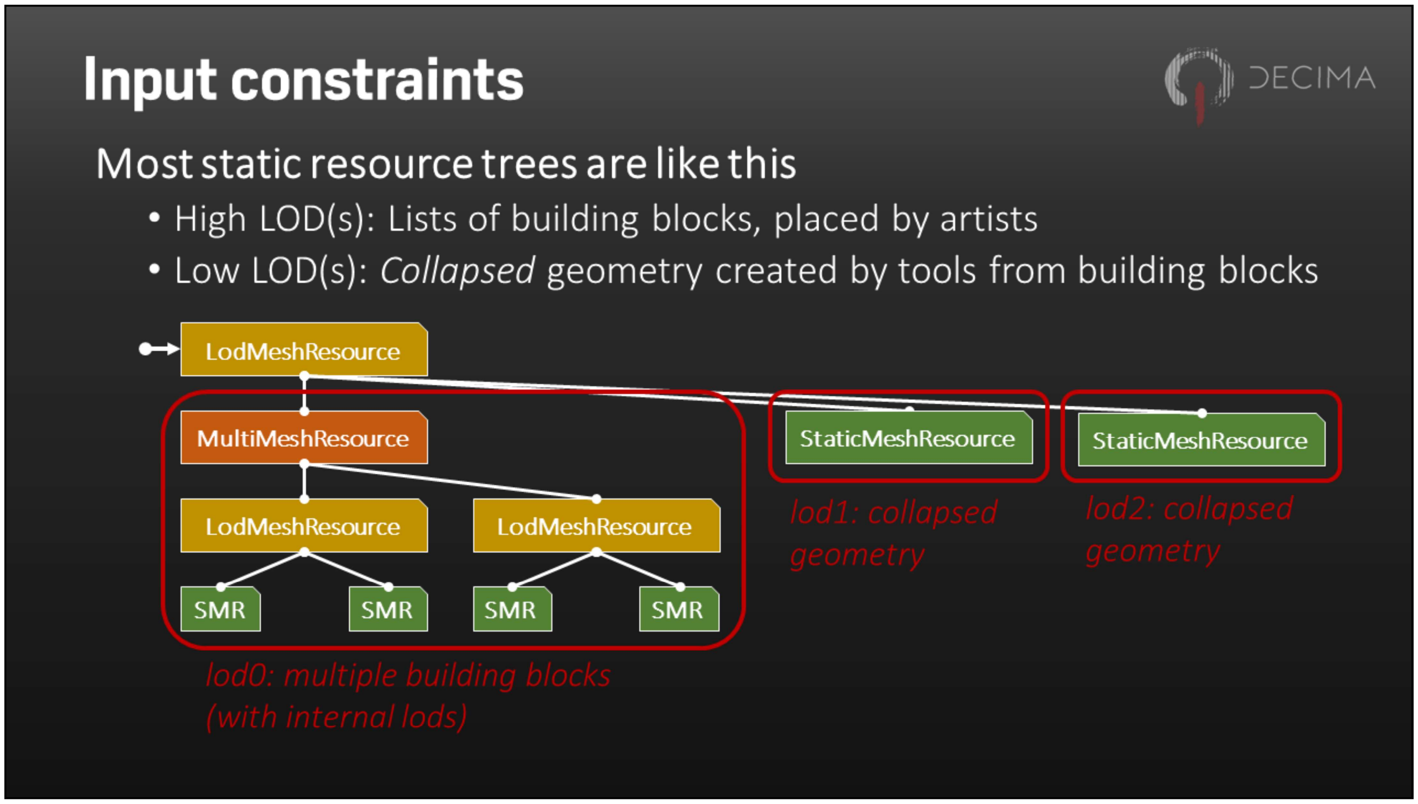
Use asynchronous compute hardware

- Synchronise with CPU not rendering
- Like PS3 SPU sync, which we knew we liked

We built a single-purpose system to ease the load on the existing multi-purpose system. The StaticScene handles just static geometry, since that's what we have vastly more of.

We could use the existing system to handle the remaining dynamic geometry and run the jobs in parallel.

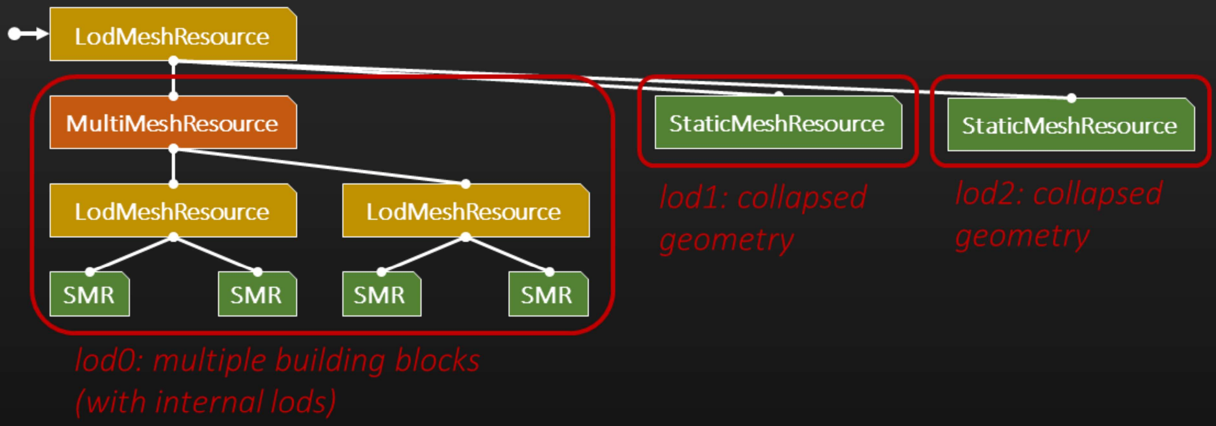
We also aimed to use the PS4's asynchronous compute capability, since we thought that would better handle the amount of content, and would be relatively easy to synchronize with the CPU.



Input constraints

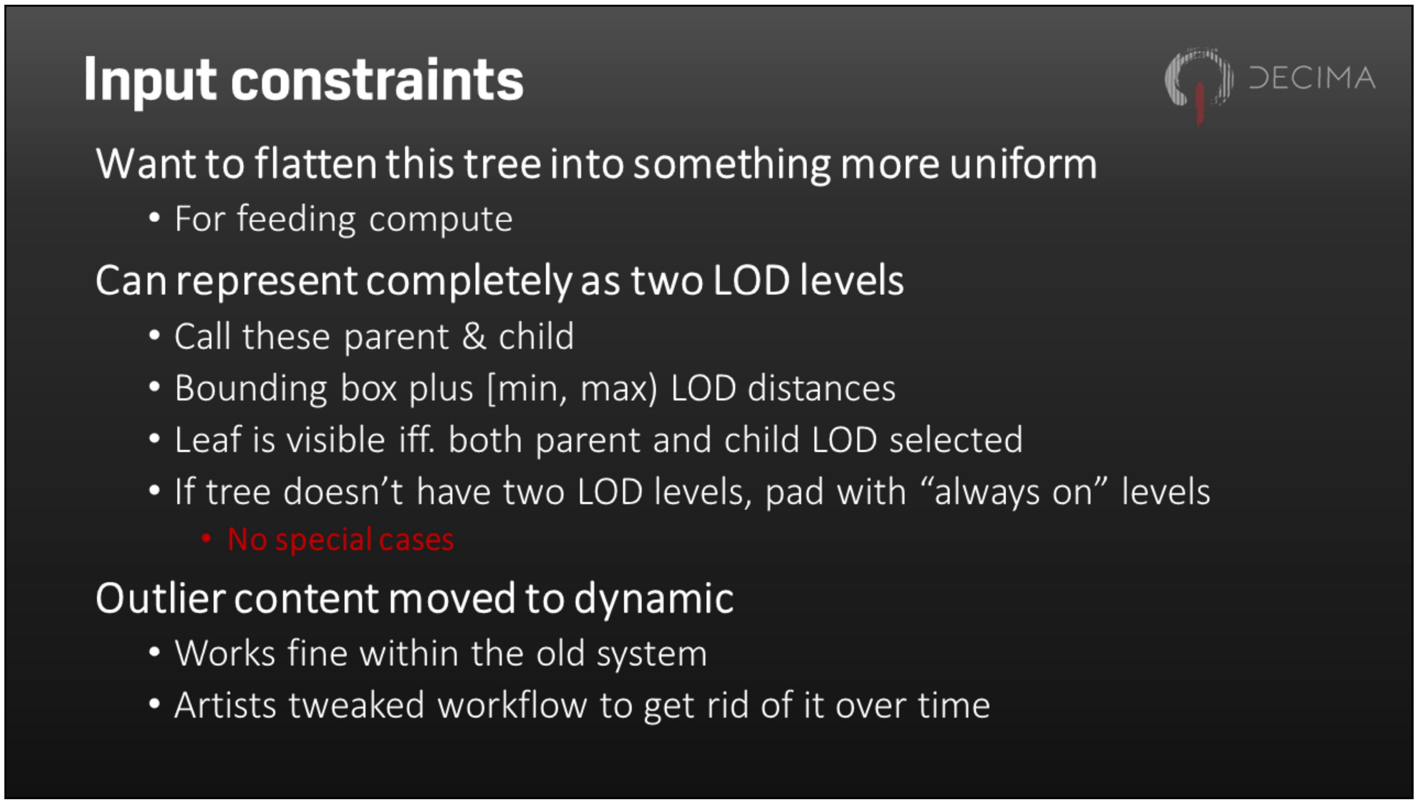
Most static resource trees are like this

- High LOD(s): Lists of building blocks, placed by artists
- Low LOD(s): *Collapsed* geometry created by tools from building blocks



Since we wanted to feed relatively flat data to compute, we had to constrain the complex MeshResource trees.

It turns out that many of our static resources looked like this, with a top-level LodMeshResource selecting between one or more LODs of building blocks, and one or more LODs of collapsed geometry created by export tools. The building blocks themselves have LOD nodes, but in the collapsed geometry that's simplified away.



Input constraints

DECIMA

Want to flatten this tree into something more uniform

- For feeding compute

Can represent completely as two LOD levels

- Call these parent & child
- Bounding box plus [min, max) LOD distances
- Leaf is visible iff. both parent and child LOD selected
- If tree doesn't have two LOD levels, pad with "always on" levels
  - No special cases

Outlier content moved to dynamic

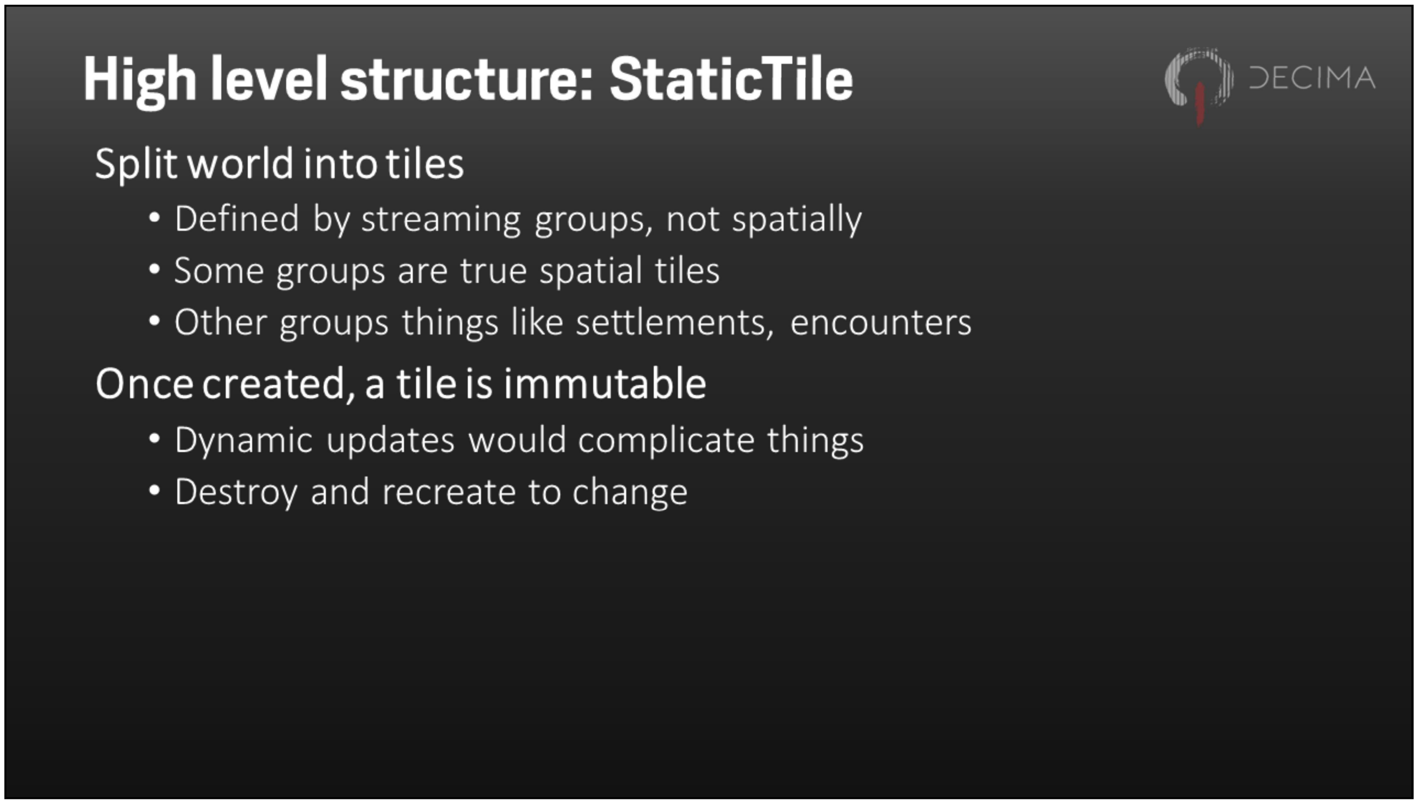
- Works fine within the old system
- Artists tweaked workflow to get rid of it over time

The key observation was that two LOD levels were enough to represent all those trees. We call these parent and child, and they have the bounds of the LodMeshResource and the LOD bracket for one of the branches.

A leaf is visible if and only if both parent and child LODs are selected.

To avoid special cases, we add "empty" LOD levels so everything has two.

Any content that didn't fit this moved to the dynamic system and the artists removed it over time.



High level structure: StaticTile

Split world into tiles

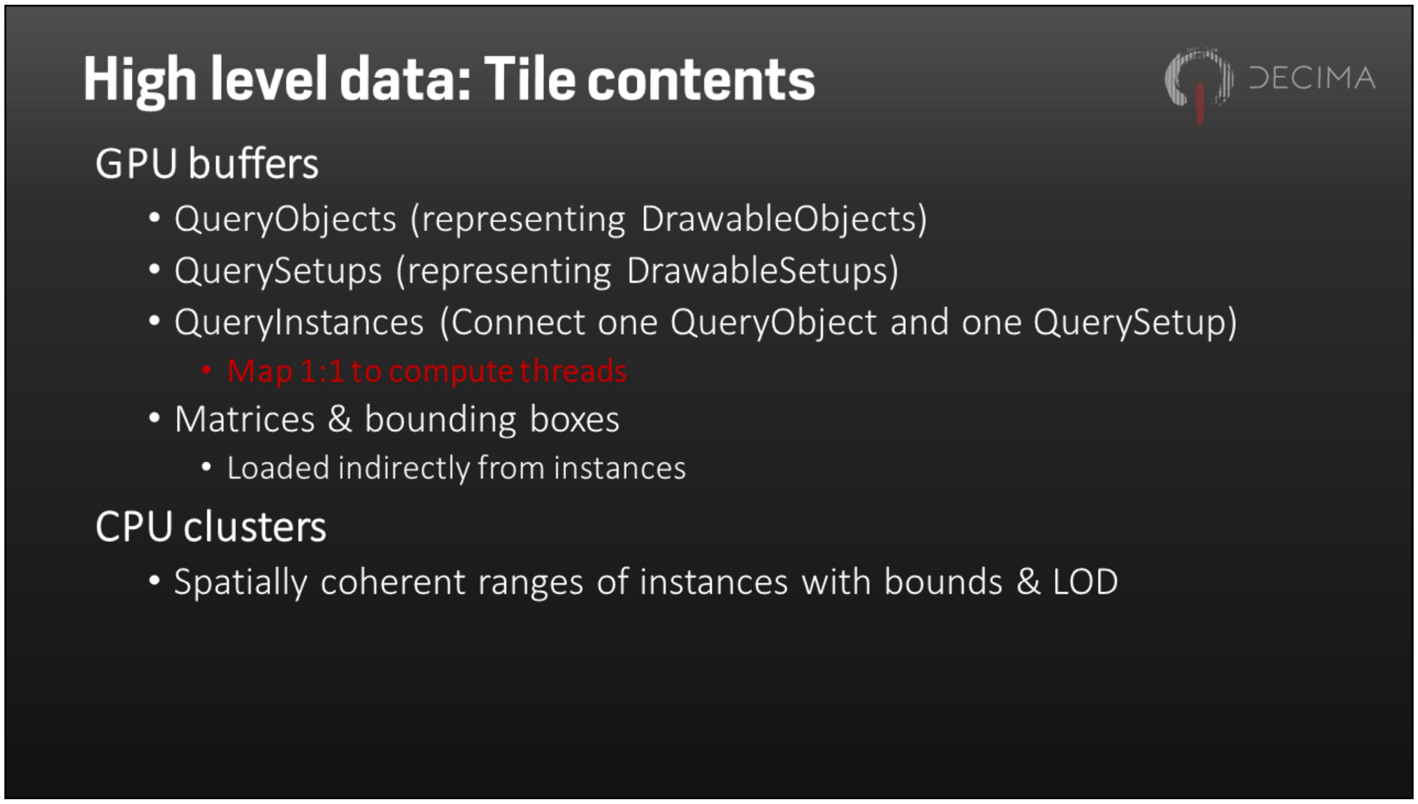
- Defined by streaming groups, not spatially
- Some groups are true spatial tiles
- Other groups things like settlements, encounters

Once created, a tile is immutable

- Dynamic updates would complicate things
- Destroy and recreate to change

To cull all this data we needed some spatial structure. At the top level we split the world into StaticTiles, which are sometimes true spatial tiles, and sometimes other things like settlements or encounters. These are defined by the streaming system, and the StaticScene doesn't get to choose.

The tiles are created immutable and can't be updated, which simplifies things.



High level data: Tile contents

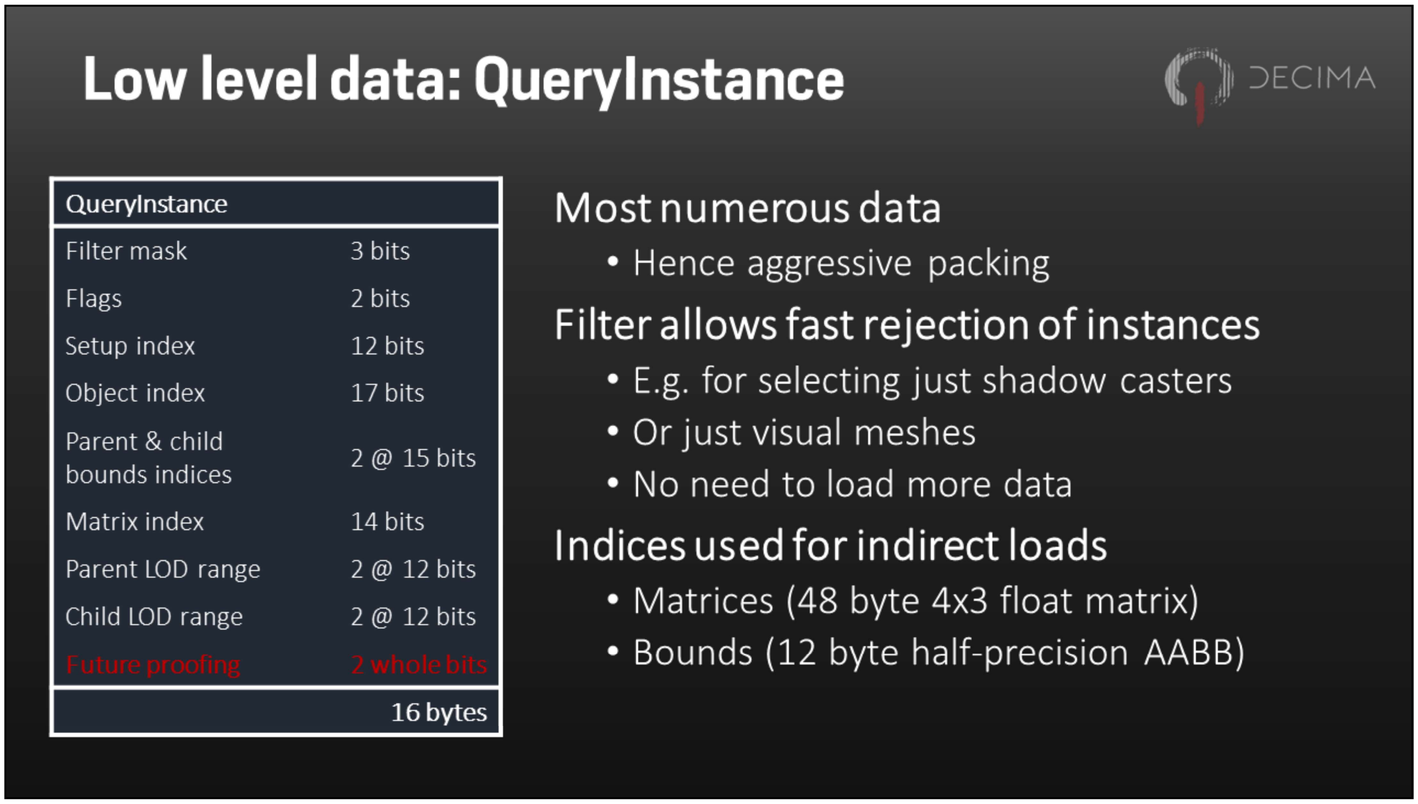
GPU buffers

- QueryObjects (representing DrawableObjects)
- QuerySetups (representing DrawableSetups)
- QueryInstances (Connect one QueryObject and one QuerySetup)
  - Map 1:1 to compute threads
- Matrices & bounding boxes
  - Loaded indirectly from instances

CPU clusters

- Spatially coherent ranges of instances with bounds & LOD

A tile is mostly flat buffers of GPU data, with a list of clusters used on the CPU for first-pass culling.



Low level data: QueryInstance

DECIMA

QueryInstance	
Filter mask	3 bits
Flags	2 bits
Setup index	12 bits
Object index	17 bits
Parent & child bounds indices	2 @ 15 bits
Matrix index	14 bits
Parent LOD range	2 @ 12 bits
Child LOD range	2 @ 12 bits
Future proofing	2 whole bits
	16 bytes

Most numerous data
- Hence aggressive packing

Filter allows fast rejection of instances
- E.g. for selecting just shadow casters
- Or just visual meshes
- No need to load more data

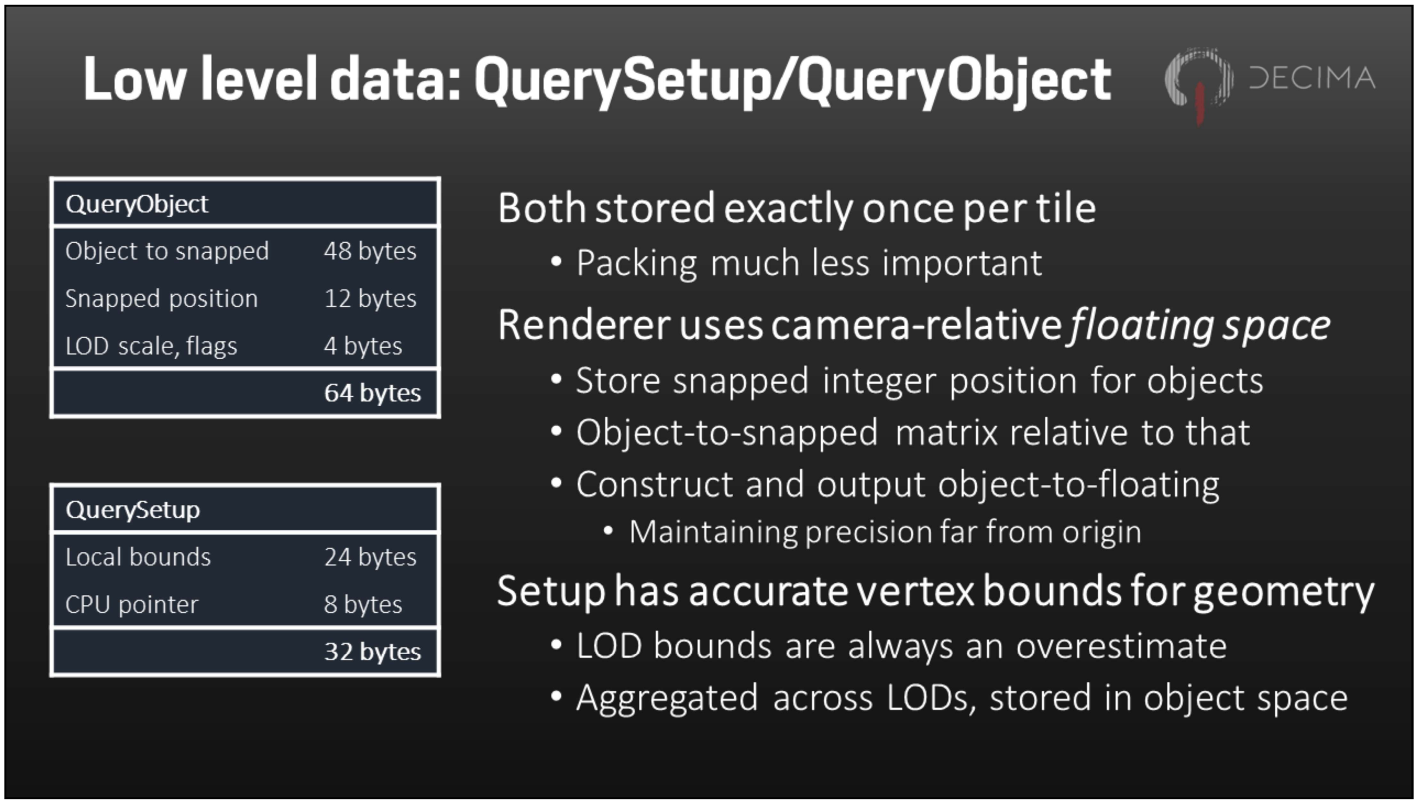
Indices used for indirect loads
- Matrices (48 byte 4x3 float matrix)
- Bounds (12 byte half-precision AABB)

Since shipping Horizon we've made some changes to support batch rendering, which I'll talk about later. The data formats I'm describing are the new ones since I didn't think there'd be time to cover both. Briefly, we used to have smaller QueryInstances and larger QuerySetups since each DrawableSetup had its own local-to-object space transform. For batching we remove the transforms from the DrawableSetups and instead use the DrawableSetup's location in the MeshResourceTree to define a local-to-object space transform.

To save space, all the elements other than the QueryInstance are hashed and stored exactly once per tile, with the instances looking them up by index. Again, this is a post-shipping change.

The data is mostly indices, some filters bits to indicate visual meshes and different types of shadowcaster, and the parent and child LOD ranges for the leaf the instance is in.

As it stands now there's not much room to grow!



Low level data: QuerySetup/QueryObject

DECIMA

QueryObject	
Object to snapped	48 bytes
Snapped position	12 bytes
LOD scale, flags	4 bytes
	64 bytes

QuerySetup	
Local bounds	24 bytes
CPU pointer	8 bytes
	32 bytes

Both stored exactly once per tile
- Packing much less important

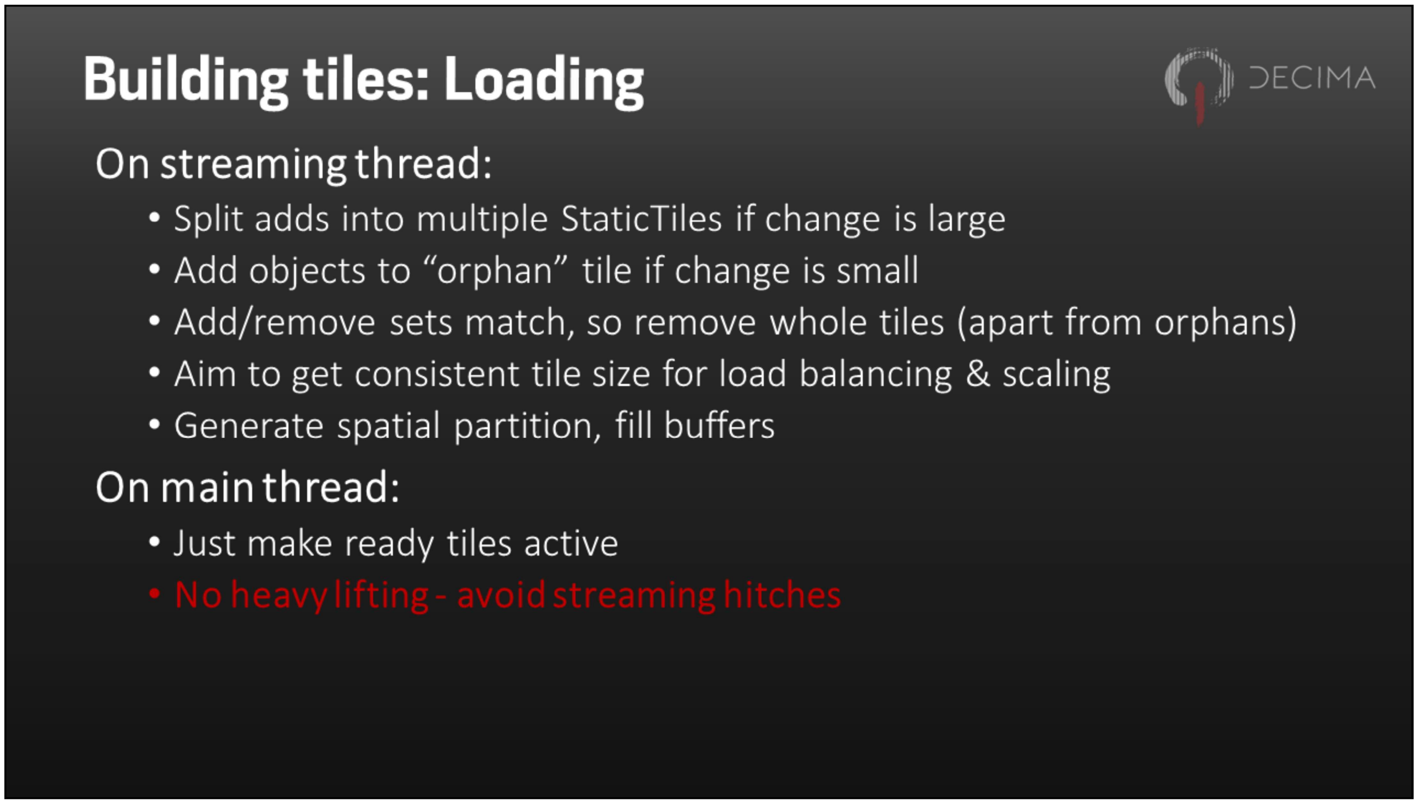
Renderer uses camera-relative *floating space*
- Store snapped integer position for objects
- Object-to-snapped matrix relative to that
- Construct and output object-to-floating
  - Maintaining precision far from origin

Setup has accurate vertex bounds for geometry
- LOD bounds are always an overestimate
- Aggregated across LODs, stored in object space

Each unique QuerySetup and QueryObject is stored once per tile, so packing these structures isn’t as important.

The game world uses a high-precision coordinate system to avoid problems caused by floating point precision loss away from origin, and the renderer works in a floating space which follows the camera. We pass in a high precision object transform based on a 1m integer grid and the query shader subtracts the snapped camera origin and outputs floating space transforms to the renderer.

The QuerySetup has accurate local bounding information which we use for frustum culling. We could use the LOD bounds but they’re always an overestimate.



Building tiles: Loading

DECIMA

On streaming thread:

- Split adds into multiple StaticTiles if change is large
- Add objects to “orphan” tile if change is small
- Add/remove sets match, so remove whole tiles (apart from orphans)
- Aim to get consistent tile size for load balancing & scaling
- Generate spatial partition, fill buffers

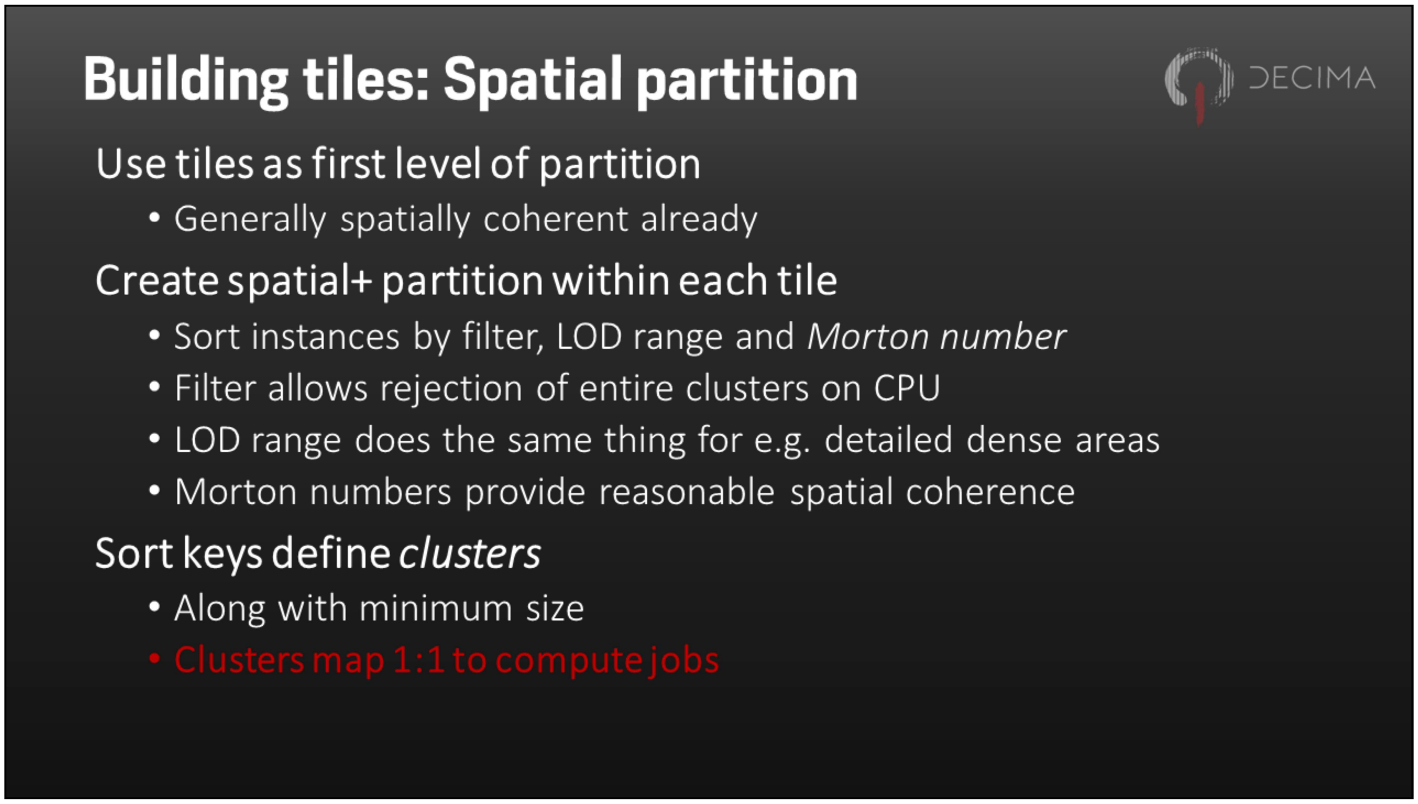
On main thread:

- Just make ready tiles active
- No heavy lifting - avoid streaming hitches

Our streaming system loads and unloads objects on a background thread, since loading can happen over several frames. The StaticScene receives sets of added and removed objects, with the adds and removes guaranteed to match up so we don’t have to deal with partial unloads.

Generally a group of added objects creates a single StaticTile, but if the tile would have a lot of QueryInstances (more than 24K) we split it up. Likewise if it would have very few (under 1K) we add these to a special “orphan” tile.

All the heavy lifting (spatial partitioning, memory allocation etc.) happens on the streaming thread, the main thread just has to make tiles active when they’re available.



Building tiles: Spatial partition

DECIMA

Use tiles as first level of partition

- Generally spatially coherent already

Create spatial+ partition within each tile

- Sort instances by filter, LOD range and *Morton number*
- Filter allows rejection of entire clusters on CPU
- LOD range does the same thing for e.g. detailed dense areas
- Morton numbers provide reasonable spatial coherence

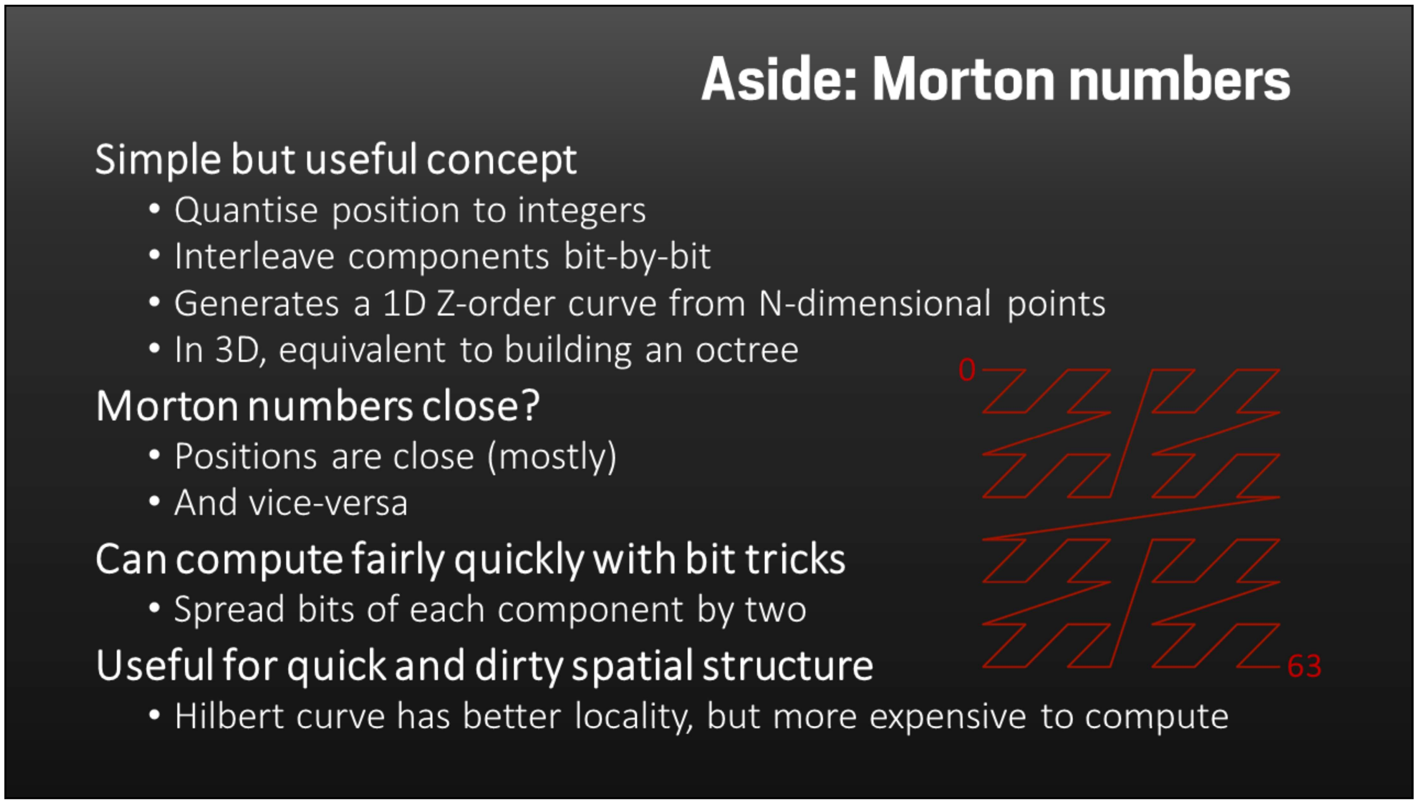
Sort keys define *clusters*

- Along with minimum size
- Clusters map 1:1 to compute jobs

We need a spatial partition to break up the data, and we have a couple of levels for this. The StaticTiles provide the first level since they're generally coherent already, and we create a partially spatial partition within each tile to define clusters.

We generate a sort key for each QueryInstance using filter bits, maxmimum LOD range, and Morton number. The filter allows us to quickly discard clusters that aren't relevant to a query, and the LOD range does a similar thing for dense areas of content. Below this the Morton numbers give a degreee of spatial coherence.

After sorting, we use the changes in sort keys to define clusters, again with a minimum size of 4K instances to help with load balancing. Each cluster is a potential compute job.



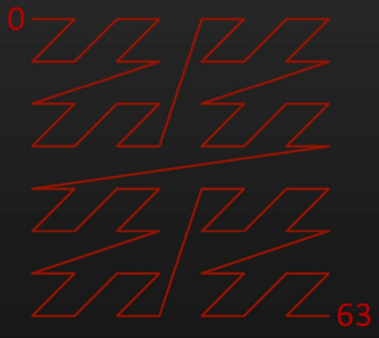
Aside: Morton numbers

Simple but useful concept
- Quantise position to integers
- Interleave components bit-by-bit
- Generates a 1D Z-order curve from N-dimensional points
- In 3D, equivalent to building an octree

Morton numbers close?
- Positions are close (mostly)
- And vice-versa

Can compute fairly quickly with bit tricks
- Spread bits of each component by two

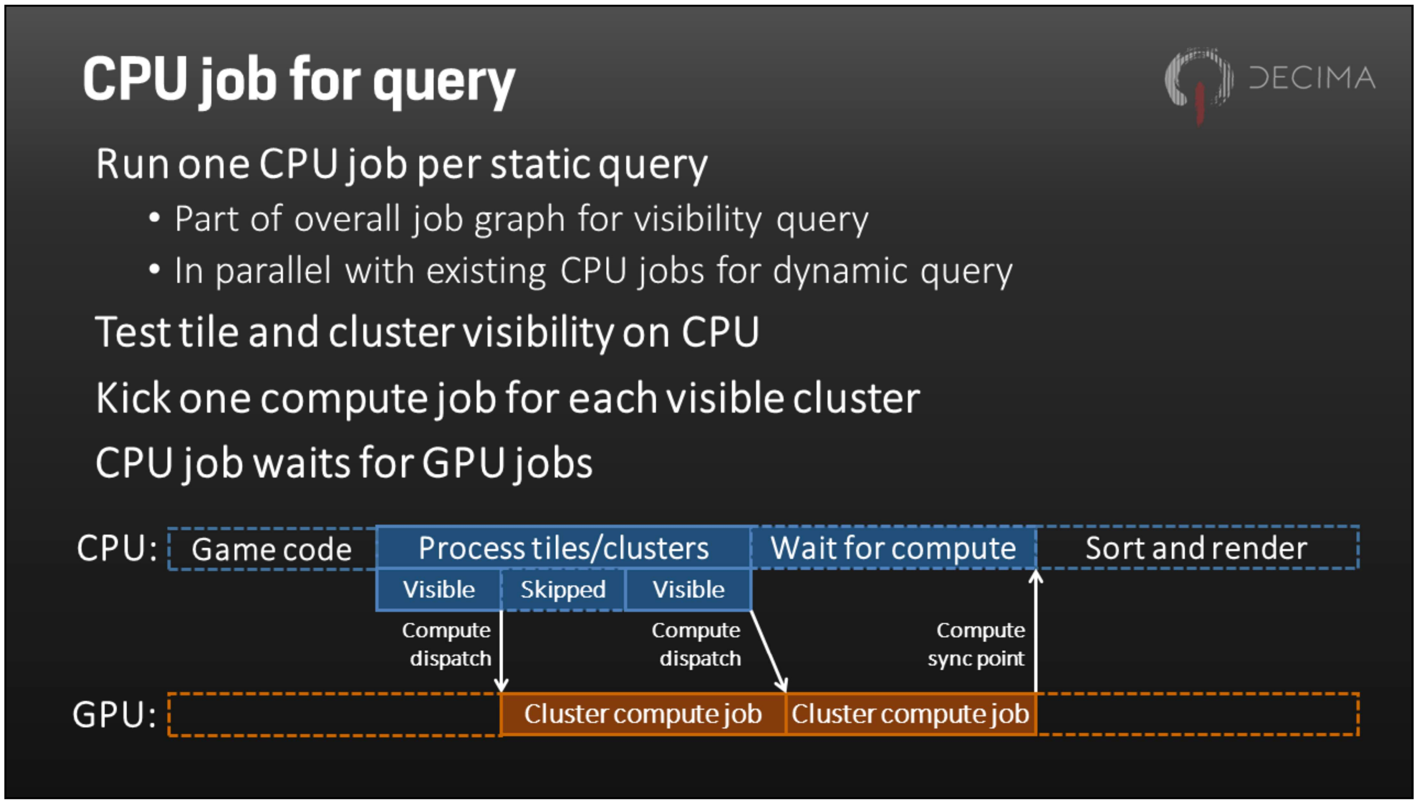
Useful for quick and dirty spatial structure
- Hilbert curve has better locality, but more expensive to compute



Morton numbers, introduced by Guy Macdonald Morton, are a way of mapping between N-dimensional coordinates and a one-dimensional value.

We start by quantising a position to some integer grid, and interleave the components bit-by-bit to produce a single number. In 3D this is like building an octree, and the increasing Morton numbers follow a Z-order curve like the one in the picture.

They're pretty easy to compute with bit tricks, and if the Morton numbers are close together, then the positions are generally close together too so these are useful for quick and dirty spatial structuring. There are better but more expensive curves available, such as the Hilbert curve.



CPU job for query

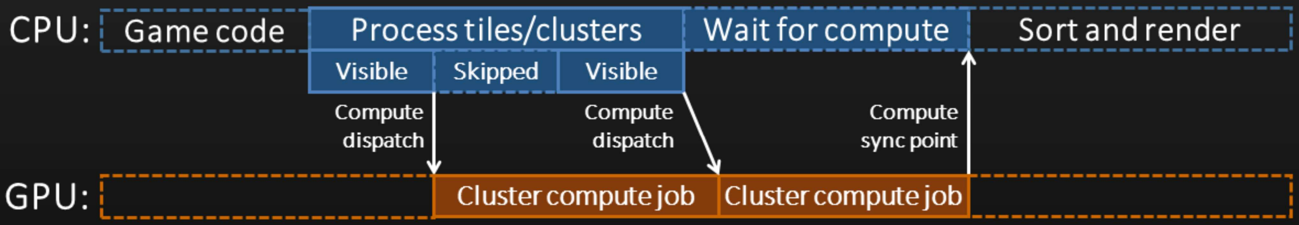
Run one CPU job per static query

- Part of overall job graph for visibility query
- In parallel with existing CPU jobs for dynamic query

Test tile and cluster visibility on CPU

Kick one compute job for each visible cluster

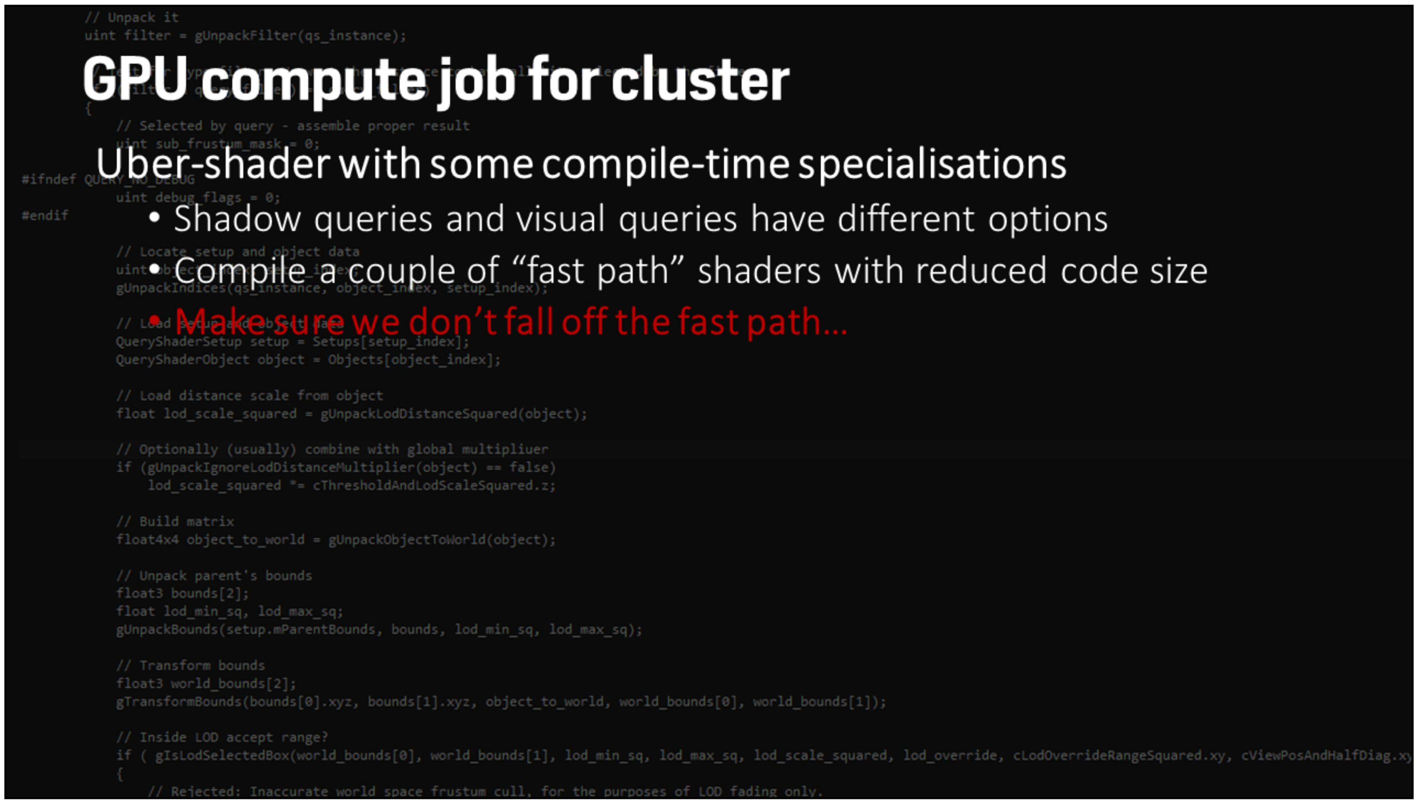
CPU job waits for GPU jobs



Having built our data we want to execute queries on it, and like the old system we create a CPU job to do this within the render job graph. This can run in parallel with the dynamic query job so the latencies don’t add up.

We use the tile and cluster hierarchy to skip as many clusters as possible, and dispatch one compute job for each visible cluster. Then the CPU job waits for compute to finish. This is nice and simple and avoids having our CPU job scheduler synchronise mixed job types.

Usually when waiting for a result like this, we’d pick up more work from the scheduler to avoid idling, but in this case we opt not to since the wait is generally short and we’re really after minimal latency.



GPU compute job for cluster

Uber-shader with some compile-time specialisations

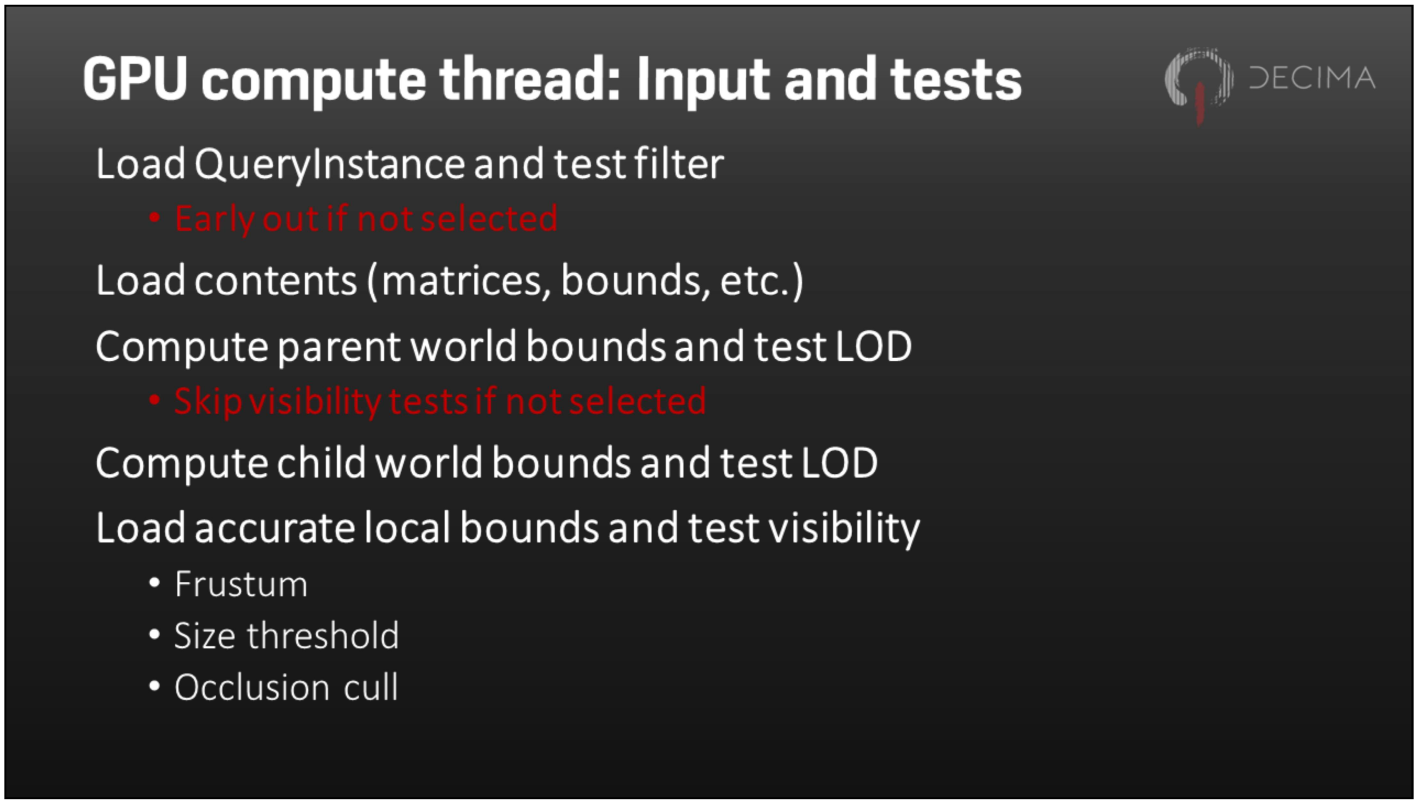
- Shadow queries and visual queries have different options
- Compile a couple of “fast path” shaders with reduced code size
- Make sure we don’t fall off the fast path…

```
        // Unpack it
        uint filter = gUnpackFilter(qs_instance);

        [illegible]
        [illegible]
        {
            // Selected by query - assemble proper result
            uint sub_frustum_mask = 0;

#ifndef QUERY_NO_DEBUG
            uint debug_flags = 0;

#endif

            // Locate setup and object data
            uint object_index, setup_index;
            gUnpackIndices(qs_instance, object_index, setup_index);

            // Load setup and object data
            QueryShaderSetup setup = Setups[setup_index];
            QueryShaderObject object = Objects[object_index];

            // Load distance scale from object
            float lod_scale_squared = gUnpackLodDistanceSquared(object);

            // Optionally (usually) combine with global multipliuer
            if (gUnpackIgnoreLodDistanceMultiplier(object) == false)
                lod_scale_squared *= cThresholdAndLodScaleSquared.z;

            // Build matrix
            float4x4 object_to_world = gUnpackObjectToWorld(object);

            // Unpack parent's bounds
            float3 bounds[2];
            float lod_min_sq, lod_max_sq;
            gUnpackBounds(setup.mParentBounds, bounds, lod_min_sq, lod_max_sq);

            // Transform bounds
            float3 world_bounds[2];
            gTransformBounds(bounds[0].xyz, bounds[1].xyz, object_to_world, world_bounds[0], world_bounds[1]);

            // Inside LOD accept range?
            if ( gIsLodSelectedBox(world_bounds[0], world_bounds[1], lod_min_sq, lod_max_sq, lod_scale_squared, lod_override, cLodOverrideRangeSquared.xy, cViewPosAndHalfDiag.xy
            {
                // Rejected: Inaccurate world space frustum cull, for the purposes of LOD fading only.
```

The GPU jobs we dispatch use a single uber shader which is specialised into a handful of compile time variants. These all have reduced instruction and register count versus the generic shader, which helps the GPU compute units run more wavefronts at a time.

We only use the generic shader when query options are switched from their default settings, which is a debug only thing.

The shader is pretty long, around 1500 instructions in most cases.



GPU compute thread: Input and tests

DECIMA

Load QueryInstance and test filter
- Early out if not selected

Load contents (matrices, bounds, etc.)

Compute parent world bounds and test LOD
- Skip visibility tests if not selected

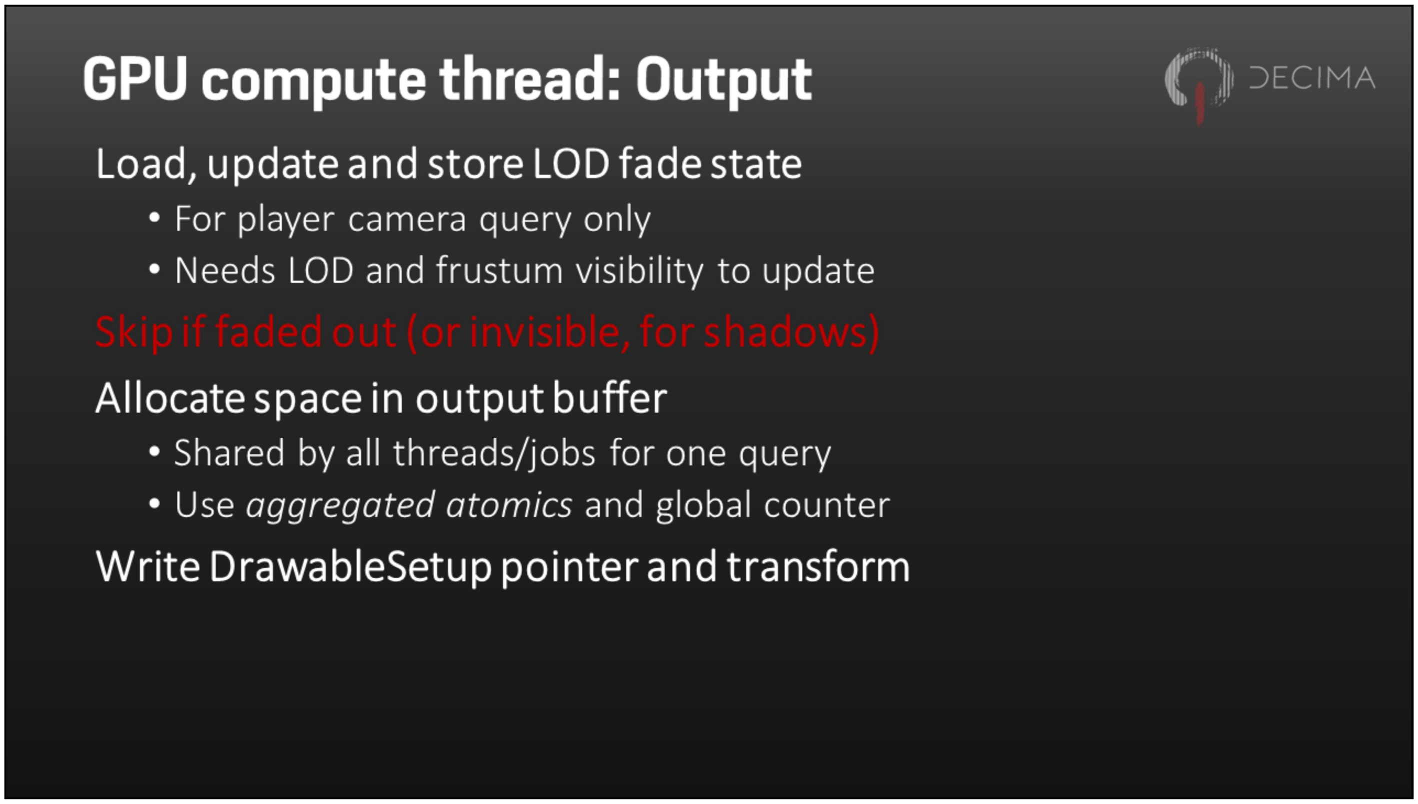
Compute child world bounds and test LOD

Load accurate local bounds and test visibility
- Frustum
- Size threshold
- Occlusion cull

Within the compute job, each thread loads a QueryInstance and tests the filter bits to see if it can skip all the work.

Otherwise it goes ahead and indirectly loads the bounds and matrices in order to perform the LOD and visibility tests. If at least the parent LOD test passes, we do both child LOD and accurate visibility tests since we need both results to update the LOD fade states.

We also check mesh streaming availability at this point – if vertex and index data is streamed out we can't draw the object.


GPU compute thread: Output

Load, update and store LOD fade state
- For player camera query only
- Needs LOD and frustum visibility to update

Skip if faded out (or invisible, for shadows)

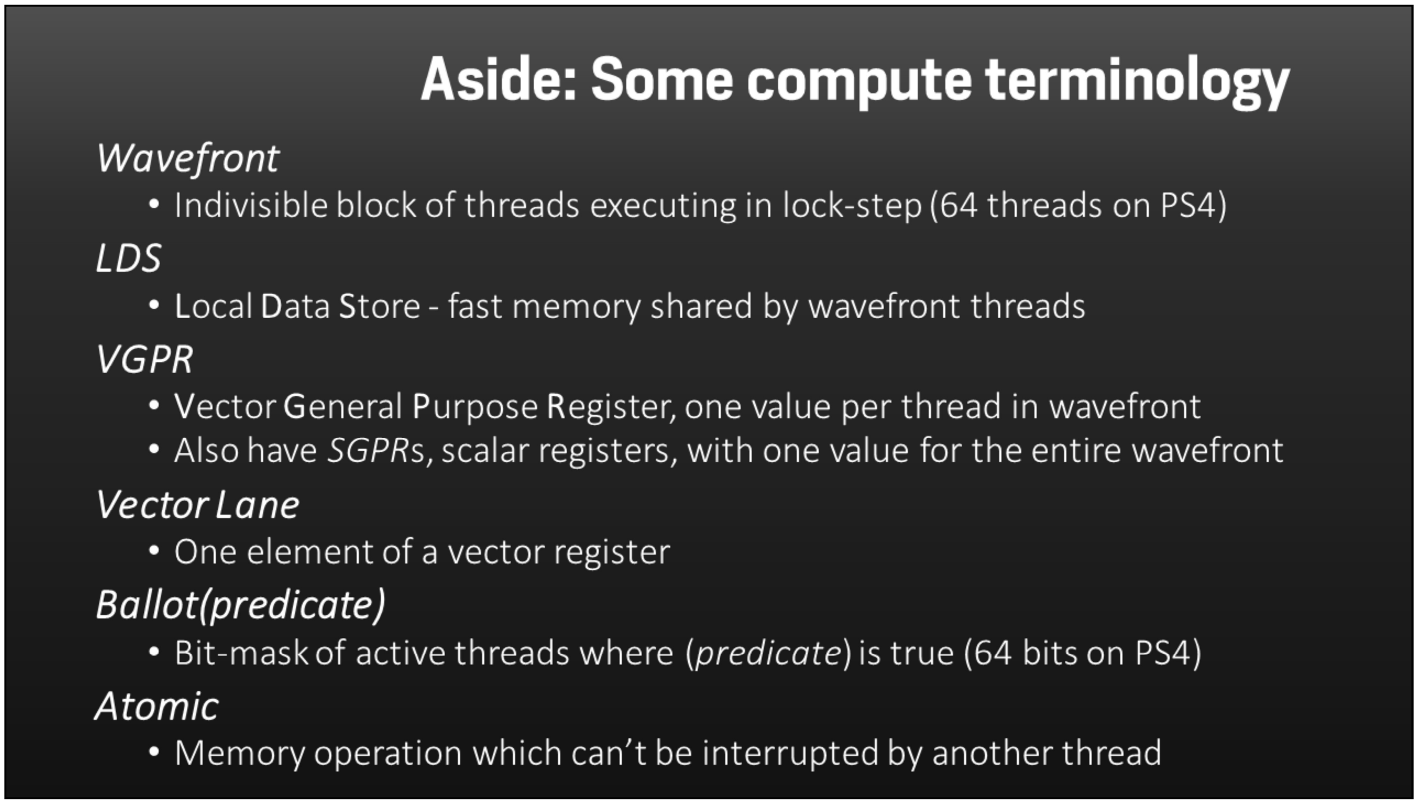
Allocate space in output buffer
- Shared by all threads/jobs for one query
- Use *aggregated atomics* and global counter

Write DrawableSetup pointer and transform

Once the tests are complete, we can update the LOD fade state using the results and the previous frame’s value. This only happens for the player camera queries at the moment.

If the instance is invisible, we stop, otherwise we allocate space for it in the output buffer. Since this is shared by all jobs and threads in the query, we have to synchronise access using a global counter updated atomically.

Once we have an address, each thead can write the DrawableSetup and transform to the output buffer.



Aside: Some compute terminology

Wavefront

- Indivisible block of threads executing in lock-step (64 threads on PS4)

LDS

- Local Data Store - fast memory shared by wavefront threads

VGPR

- Vector General Purpose Register, one value per thread in wavefront
- Also have *SGPR*s, scalar registers, with one value for the entire wavefront

Vector Lane

- One element of a vector register

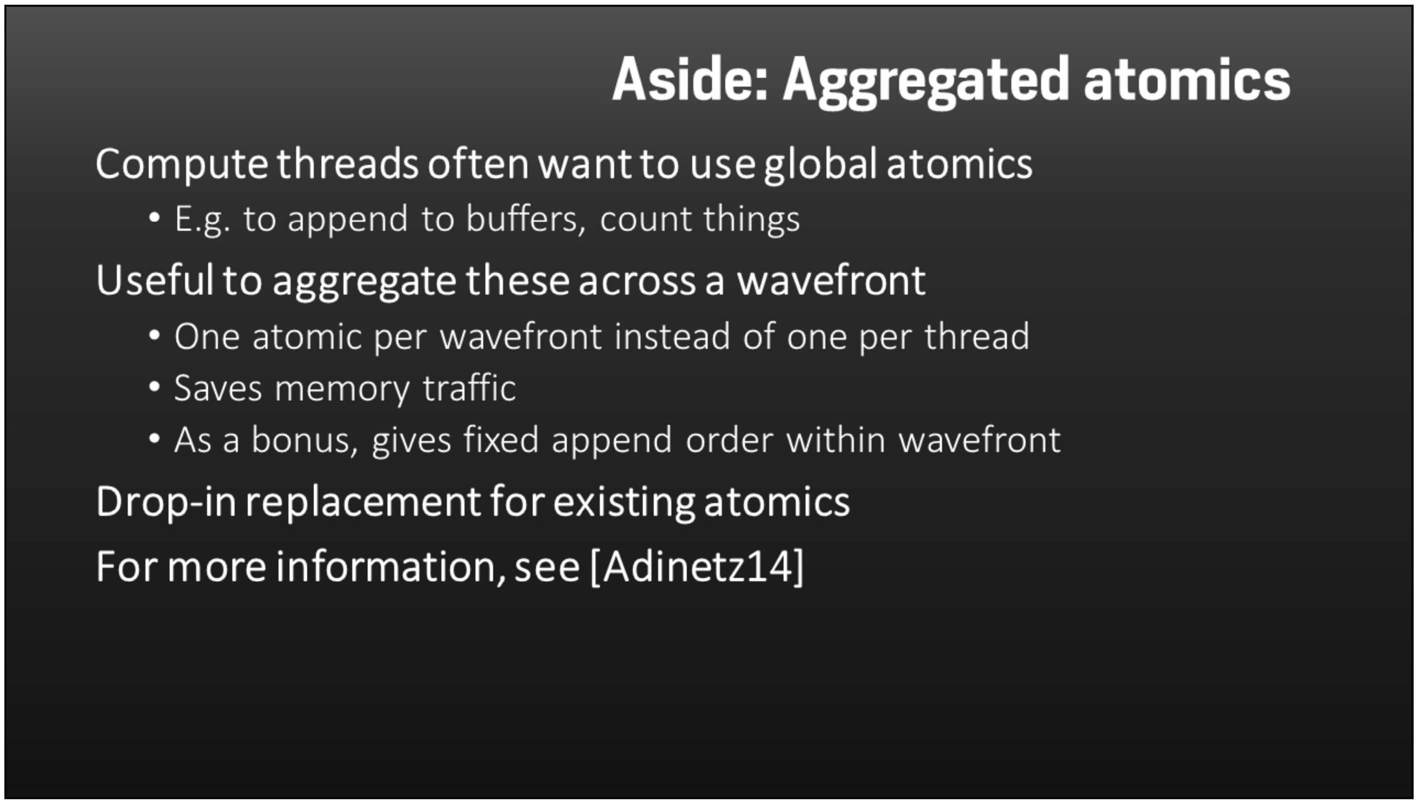
Ballot(predicate)

- Bit-mask of active threads where (*predicate*) is true (64 bits on PS4)

Atomic

- Memory operation which can't be interrupted by another thread

I'm going to talk in more detail about our compute work now, so I'll just go over some basic terms.



Aside: Aggregated atomics

Compute threads often want to use global atomics

- E.g. to append to buffers, count things

Useful to aggregate these across a wavefront

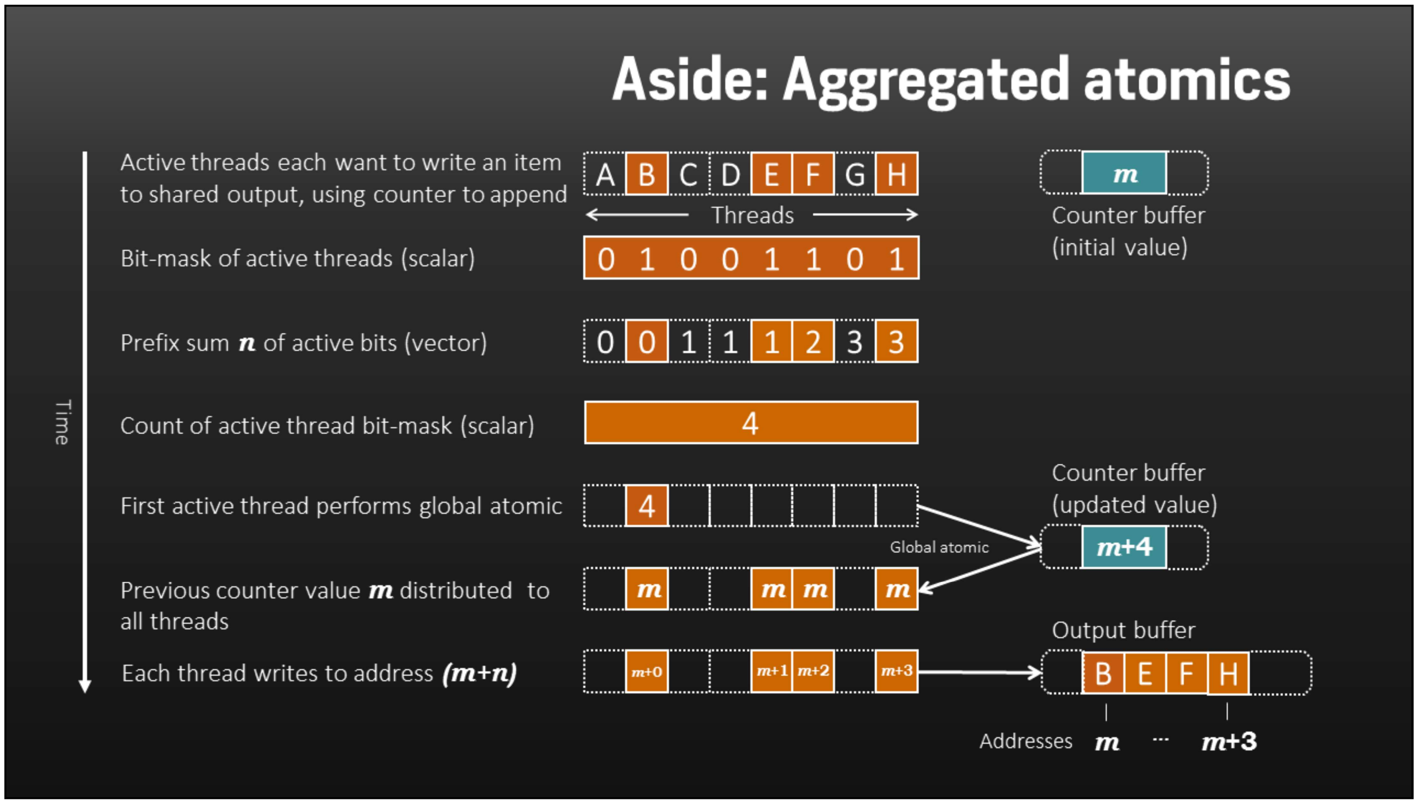
- One atomic per wavefront instead of one per thread
- Saves memory traffic
- As a bonus, gives fixed append order within wavefront

Drop-in replacement for existing atomics

For more information, see [Adinetz14]

All our atomic operations are aggregated across a wavefront's threads. This means that instead of using one atomic per thread we use one per wavefront.

This saves a lot of memory traffic, and is a drop-in replacement for standard atomics.



Aside: Aggregated atomics

Active threads each want to write an item to shared output, using counter to append

Bit-mask of active threads (scalar)

Prefix sum n of active bits (vector)

Count of active thread bit-mask (scalar)

First active thread performs global atomic

Previous counter value m distributed to all threads

Each thread writes to address $(m+n)$

In this diagram, the rows of boxes show threads within a single simplified eight-thread wavefront, and the contents show what that thread is looking at during a particular stage of the process. So each column represents values in a single thread. This is obviously simplified since each thread will have several live registers at any time rather than just the one-per-row shown here.

The rounded boxes on the right show global buffers elsewhere in GPU memory, which are being operated on by this wavefront and many others at the same time.

We start with some active threads (coloured boxes) which want to write to the output buffer.

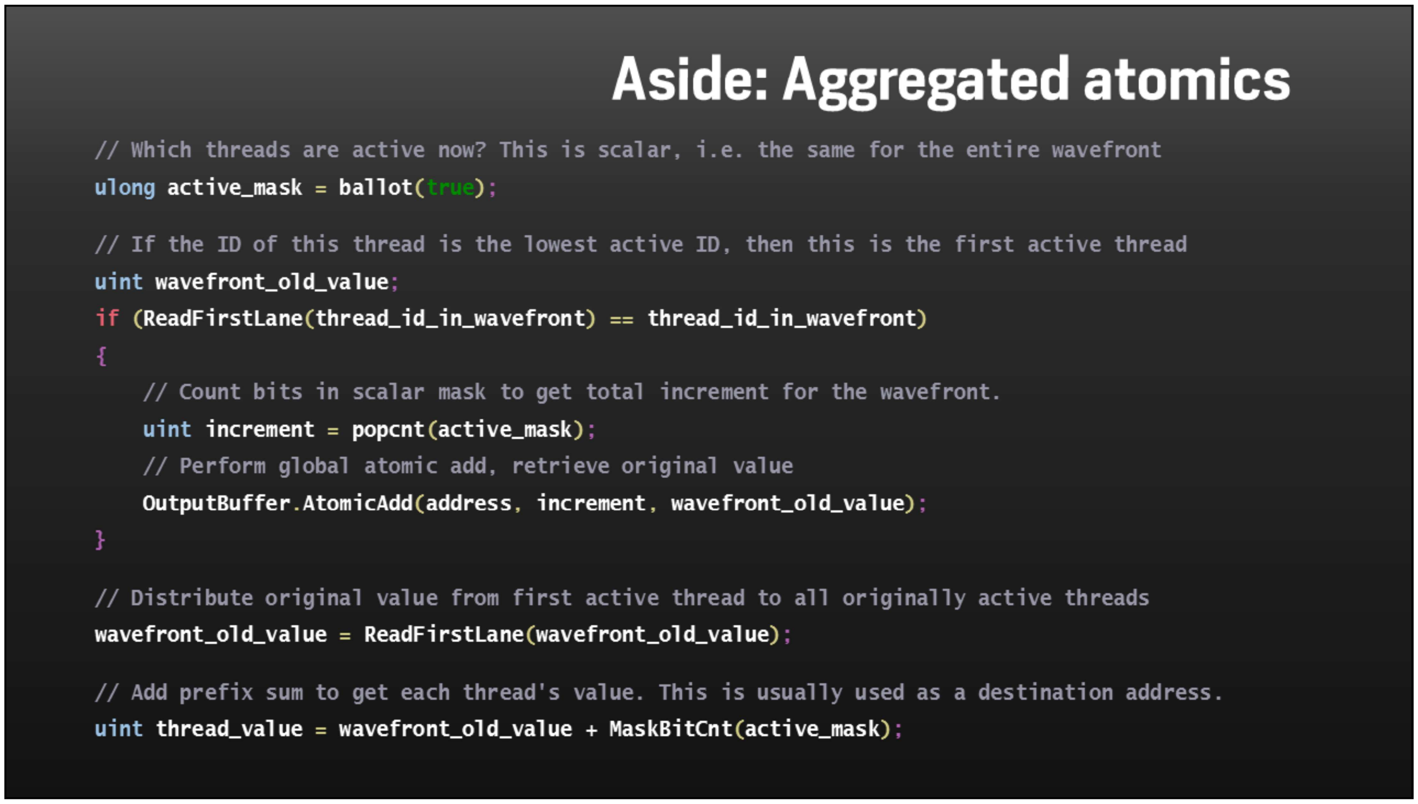
We create a bitmask of the active threads using ballot(), which is a scalar value, i..e the same for each thread.

We also create a prefix sum over the bitmask, by adding up all the bit values to the left of the thread in question. This is different for each thread as you can see.

Lastly we sum the bits in the bitmask, which is scalar again.

We then select the first thread and have it perform the global atomic, adding the sum of bits to the global counter value.

We get back the original value and distribute it to each thread, and finally each thread adds the prefix sum and writes to that address. As you can see the writes are compact and ordered by the prefix sum.



Aside: Aggregated atomics

```
// Which threads are active now? This is scalar, i.e. the same for the entire wavefront
ulong active_mask = ballot(true);

// If the ID of this thread is the lowest active ID, then this is the first active thread
uint wavefront_old_value;
if (ReadFirstLane(thread_id_in_wavefront) == thread_id_in_wavefront)
{
    // Count bits in scalar mask to get total increment for the wavefront.
    uint increment = popcnt(active_mask);
    // Perform global atomic add, retrieve original value
    OutputBuffer.AtomicAdd(address, increment, wavefront_old_value);
}

// Distribute original value from first active thread to all originally active threads
wavefront_old_value = ReadFirstLane(wavefront_old_value);

// Add prefix sum to get each thread's value. This is usually used as a destination address.
uint thread_value = wavefront_old_value + MaskBitCnt(active_mask);
```

And here’s the code for that. ballot(true) gives us a mask for active threads.

We select the first active thread by comparing the current thread’s ID to the first active thread’s value for that ID.

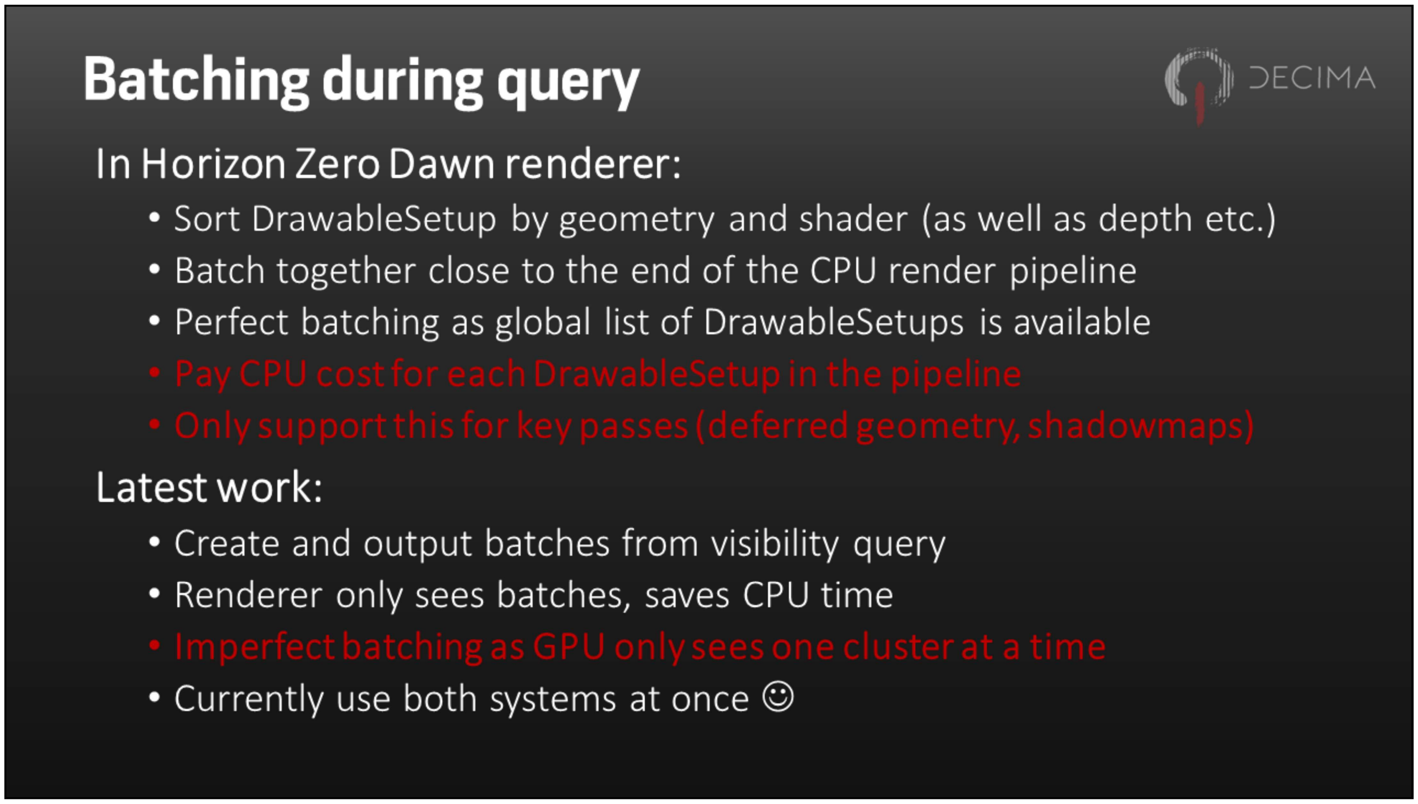
Then the first thread counts the bits in the mask and adds that to the global counter.

Each thread then retrieve’s the first active thread’s result, and finally adds the prefix sum to get its own counter value.

LATEST WORK

So that covers how the system is put together,

I’m now going to talk about some brand new work which didn’t ship in Horizon.



Batching during query

In Horizon Zero Dawn renderer:

- Sort DrawableSetup by geometry and shader (as well as depth etc.)
- Batch together close to the end of the CPU render pipeline
- Perfect batching as global list of DrawableSetups is available
- Pay CPU cost for each DrawableSetup in the pipeline
- Only support this for key passes (deferred geometry, shadowmaps)

Latest work:

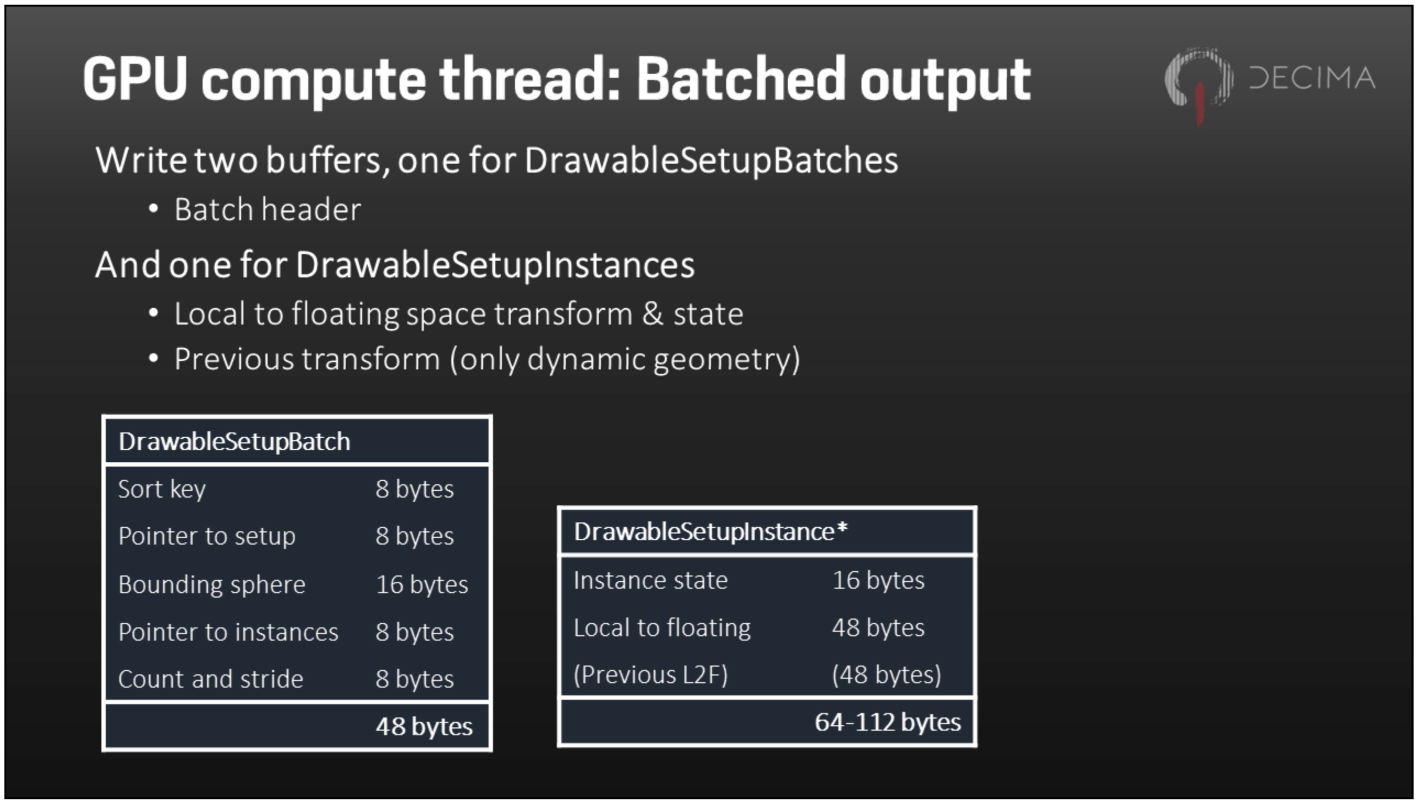
- Create and output batches from visibility query
- Renderer only sees batches, saves CPU time
- Imperfect batching as GPU only sees one cluster at a time
- Currently use both systems at once ☺

In the Horizon renderer, after the query is complete we sort the visible DrawableSetups by their geometry and shader (as well as the usual sort criteria like depth)

At the back end of the renderer, we collect batches with consistent geometry and shader, and draw those as a single draw call. This gives us good batching because we can see all the DrawableSetups, but means we paid the CPU cost for all the individual DrawableSetups in the render pipeline.

We also only supported this for key passes like deferred geometry and shadowmap rendering, since it needed more machinery in the back end.

We wanted to remove this extra CPU and scratch memory cost by doing the batching in the query instead, at the very front of the pipeline. The batch quality is lower since the GPU only sees a cluster of objects at a time, but it's still pretty good. Currently we're using both systems but I'm hoping we can turn off the last minute batching in future.



GPU compute thread: Batched output

Write two buffers, one for DrawableSetupBatches

- Batch header

And one for DrawableSetupInstances

- Local to floating space transform & state
- Previous transform (only dynamic geometry)

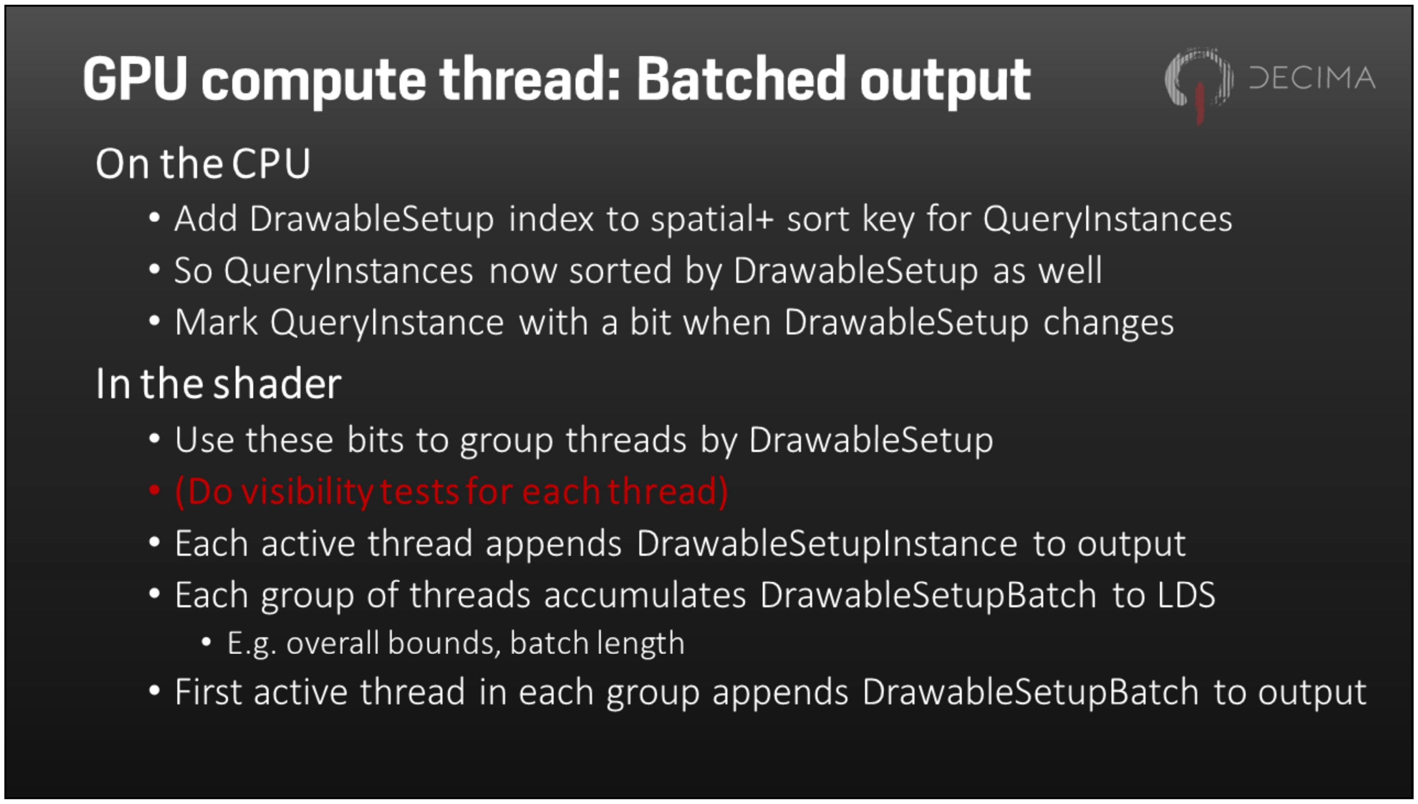
DrawableSetupBatch	
Sort key	8 bytes
Pointer to setup	8 bytes
Bounding sphere	16 bytes
Pointer to instances	8 bytes
Count and stride	8 bytes
	48 bytes

DrawableSetupInstance*	
Instance state	16 bytes
Local to floating	48 bytes
(Previous L2F)	(48 bytes)
	64-112 bytes

To do batching well, we removed transforms from the DrawableSetups entirely so they could be shared more effectively between instances and resources. Without this we didn’t have a good enough sharing rate.

Instead of writing DrawableSetups and transforms, we write DrawableSetupBatches, which are effectively heads for batch lists of DrawableSetupInstances. The batches contain bounds for all the instances, batch lengths etc. and the instances have the transforms and some other renderer state.

These go in scratch memory and they’re only written for what we can see, so again dense packing isn’t super critical



GPU compute thread: Batched output

DECIMA

On the CPU

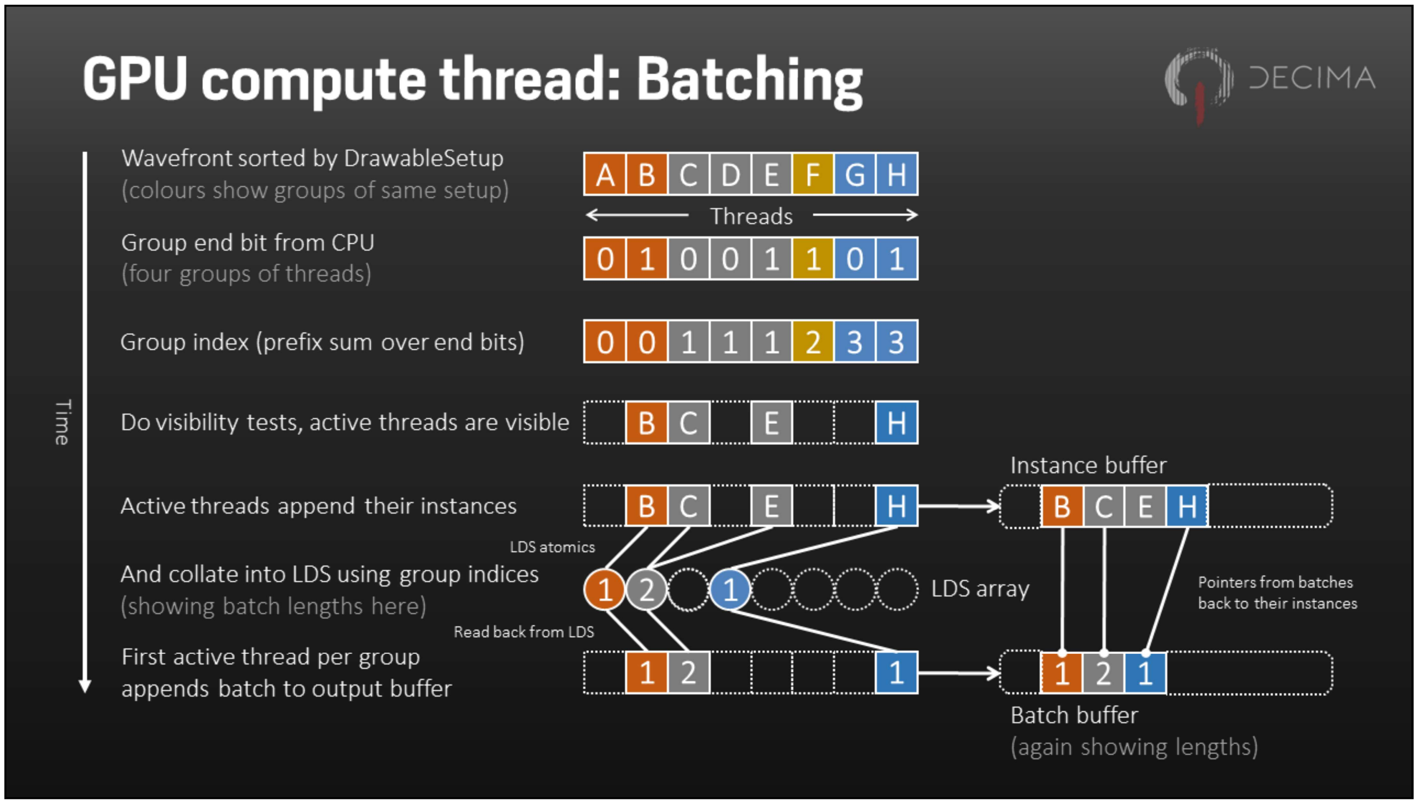
- Add DrawableSetup index to spatial+ sort key for QueryInstances
- So QueryInstances now sorted by DrawableSetup as well
- Mark QueryInstance with a bit when DrawableSetup changes

In the shader

- Use these bits to group threads by DrawableSetup
- (Do visibility tests for each thread)
- Each active thread appends DrawableSetupInstance to output
- Each group of threads accumulates DrawableSetupBatch to LDS
  - E.g. overall bounds, batch length
- First active thread in each group appends DrawableSetupBatch to output

On the CPU, we extend the QueryInstance sort key to sort by DrawableSetup, and flag QueryInstances with a bit when this changes. So now we only have one bit free of our two!

On the GPU, we uses this bits to group threads by DrawableSetup, do the normal visibility tests, have each active thread write a DrawableSetupInstance and then have the first active thread in each group about a DrawableSetupBatch to the output.



GPU compute thread: Batching

DECIMA
Time
Wavefront sorted by DrawableSetup
(colours show groups of same setup)
A B C D E F G H
Threads
Group end bit from CPU
(four groups of threads)
0 1 0 0 1 1 0 1
Group index (prefix sum over end bits)
0 0 1 1 1 2 3 3
Do visibility tests, active threads are visible
B C E H
Instance buffer
Active threads append their instances
B C E H
B C E H
LDS atomics
And collate into LDS using group indices
(showing batch lengths here)
1 2 1
LDS array
Pointers from batches back to their instances
Read back from LDS
First active thread per group appends batch to output buffer
1 2 1
1 2 1
Batch buffer
(again showing lengths)

Again we're looking at a simplified wavefront here. The letters show the QueryInstance being processed by each thread, and the colours the groups of DrawableSetups.

We start by loading the group end bit from the QueryInstance, and taking a prefix sum over this to get a group index. You can see that each coloured group has been identified by a consistent index.

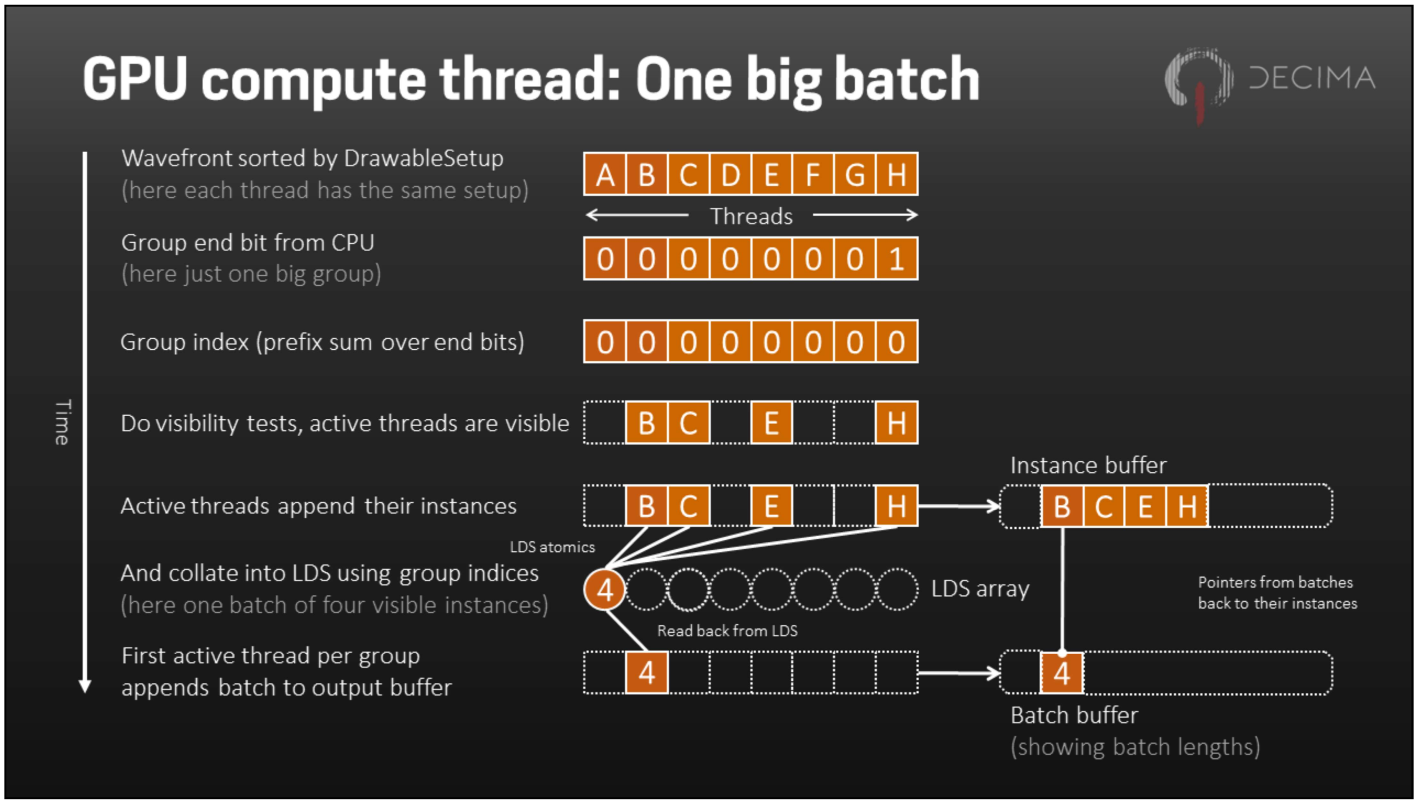
We then do the normal visibility testing and end up with a subset of active threads representing visible instances.

The active threads then write their instances to the first output buffer, using aggregated atomics as before.

They also collate their batch properties in LDS arrays shared by the entire wavefront. These are the circles, and here I'm showing each thread adding the value one to the group's array slot to create the batch lengths. In the real shader we also collate the bounds and other per-batch information.

The first active thread in each group reads back the batch data and writes the batch to the output buffer. It knows how to point to the correct instance since it was the thread that wrote the first instance in the batch.

I know this is a bit confusing, here are a couple more examples that might help.



GPU compute thread: One big batch

DECIMA
Time
Wavefront sorted by DrawableSetup
(here each thread has the same setup)
A B C D E F G H
Threads
Group end bit from CPU
(here just one big group)
0 0 0 0 0 0 0 1
Group index (prefix sum over end bits)
0 0 0 0 0 0 0 0
Do visibility tests, active threads are visible
B C E H
Instance buffer
Active threads append their instances
B C E H
LDS atomics
And collate into LDS using group indices
(here one batch of four visible instances)
4
LDS array
Pointers from batches back to their instances
Read back from LDS
First active thread per group appends batch to output buffer
4
Batch buffer
(showing batch lengths)

This example shows what happens if there's one big batch. Everything gets collated to group index zero and we output one batch for the whole wavefront.

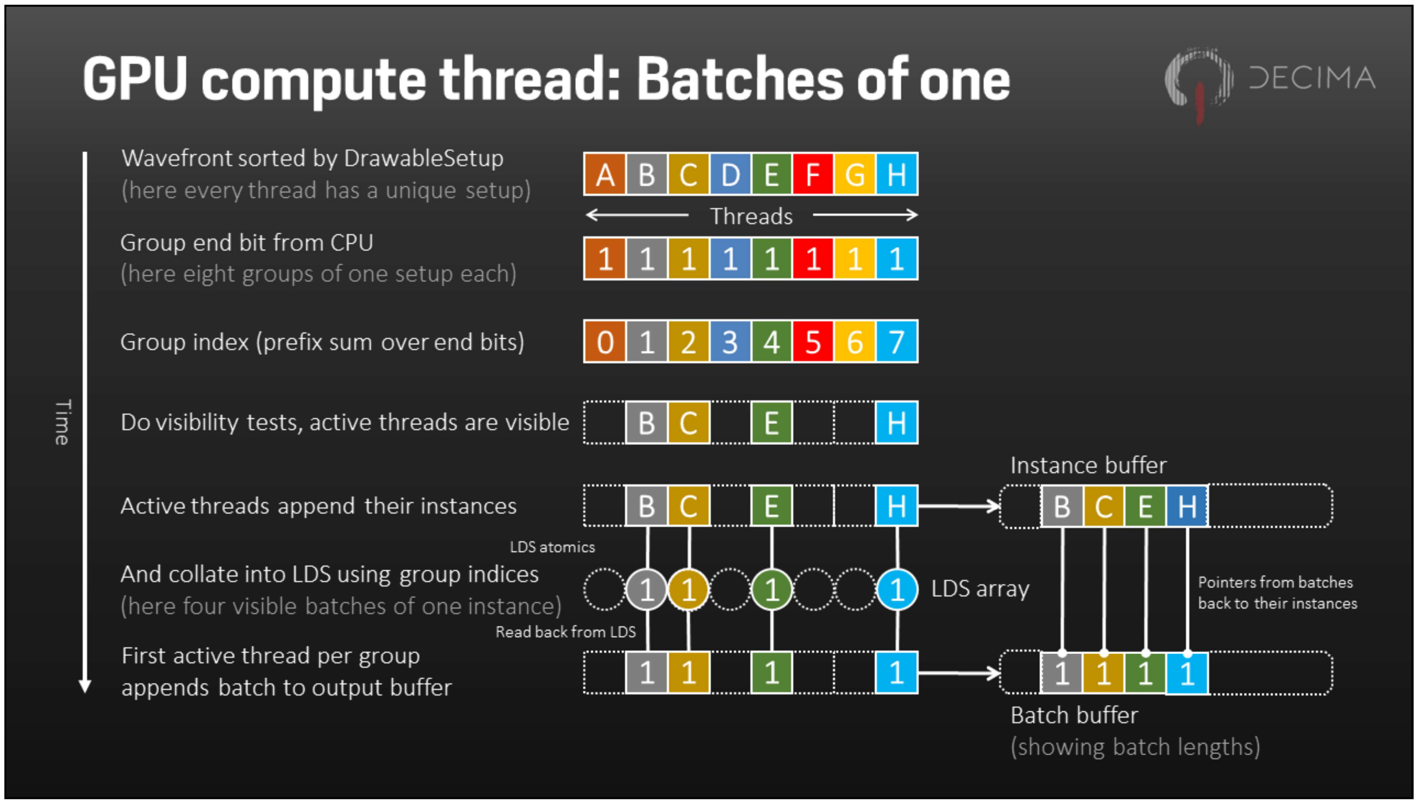
The group index is zero for each thread, since they're all in the same group.

The instances written are the same

All threads collate to group zero's entry so we get a count of four instances in the batch

And finally there's one group so we only write one entry to the batch buffer, pointing again to the start of the list of four instances.

GPU compute thread: Batches of one


DECIMA
Time
Wavefront sorted by DrawableSetup
(here every thread has a unique setup)
A B C D E F G H
Threads
Group end bit from CPU
(here eight groups of one setup each)
1 1 1 1 1 1 1 1
Group index (prefix sum over end bits)
0 1 2 3 4 5 6 7
Do visibility tests, active threads are visible
B C E H
Active threads append their instances
Instance buffer
B C E H
LDS atomics
And collate into LDS using group indices
(here four visible batches of one instance)
1 1 1 1
LDS array
Pointers from batches back to their instances
Read back from LDS
First active thread per group appends batch to output buffer
1 1 1 1
Batch buffer
(showing batch lengths)

And this is when all the DrawableSetups are different. Here the group indices are the same as the thread indices. This is basically the setup before we had batching in the shader.

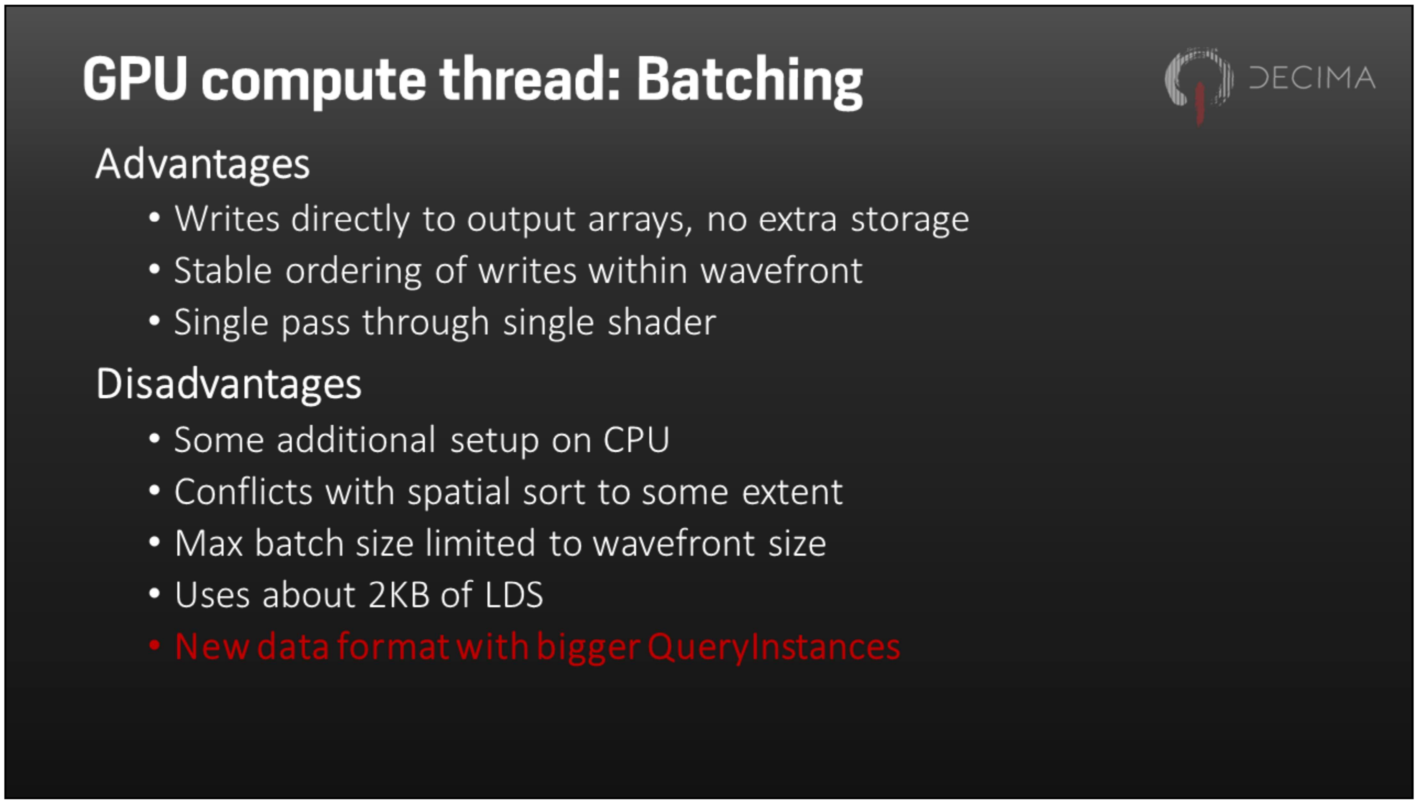
Here all the group end bits are set, so each thread gets a different group index.

We write the same list of instances as before.

But when we collate, each thread has its own group so we get batches of length one.

And each active thread writes out its batch, giving four batches of one instance each.

Phew.



GPU compute thread: Batching

Advantages

- Writes directly to output arrays, no extra storage
- Stable ordering of writes within wavefront
- Single pass through single shader

Disadvantages

- Some additional setup on CPU
- Conflicts with spatial sort to some extent
- Max batch size limited to wavefront size
- Uses about 2KB of LDS
- New data format with bigger QueryInstances

This scheme is pretty simple, and that gives us some big advantages – we don’t need any extra machinery, storage, or passes. For example if we generated linked lists of setups in each batch, we’d need another pass to flatten those into something the renderer can consume. Here we don’t.

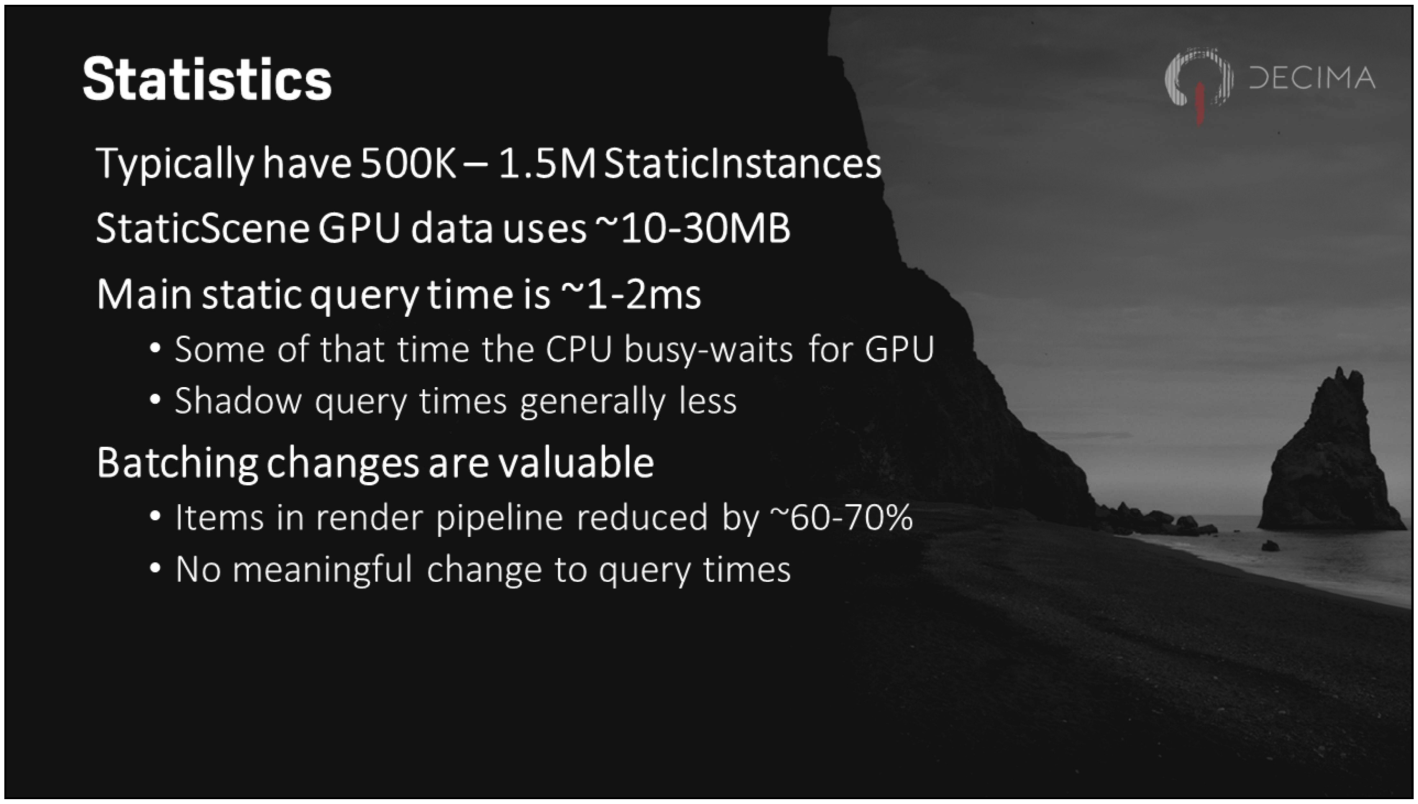
We do lose some of the spatial coherence since we have to sort by setup first, but that only affects the efficiency of the culling hierarchy, not the renderer.

The downside is really that the max batch size is limited to the wavefront size, but this doesn’t hurt much in practice – it’s still a 64-fold saving.

We also use a chunk of LDS, but since our shader uses a lot of registers the LDS isn’t hurting occupancy – we can still run as many wavefronts.

SUMMING UP

So that’s where we are now.



Statistics

DECIMA

Typically have 500K – 1.5M StaticInstances

StaticScene GPU data uses ~10-30MB

Main static query time is ~1-2ms

- Some of that time the CPU busy-waits for GPU
- Shadow query times generally less

Batching changes are valuable

- Items in render pipeline reduced by ~60-70%
- No meaningful change to query times

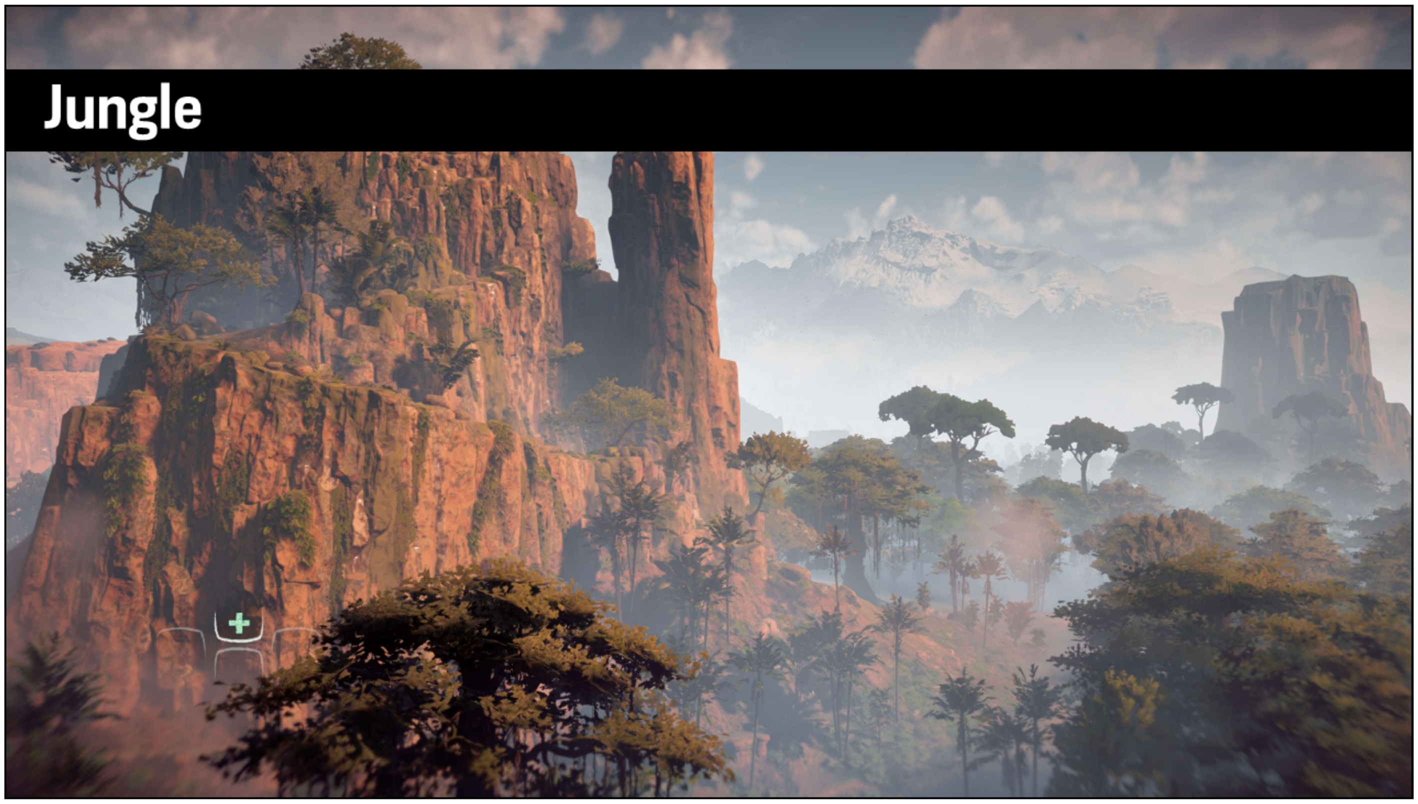
The static scene handles a lot of data – usually 500K up to 1.5 million instances which could potentially be visible to a single camera position.

The GPU data to represent this isn't too big, but as always we'd like it to be smaller.

The query time is also pretty consistent, the shadow queries are usually faster since they have different frusta and somewhat relaxed LOD criteria.

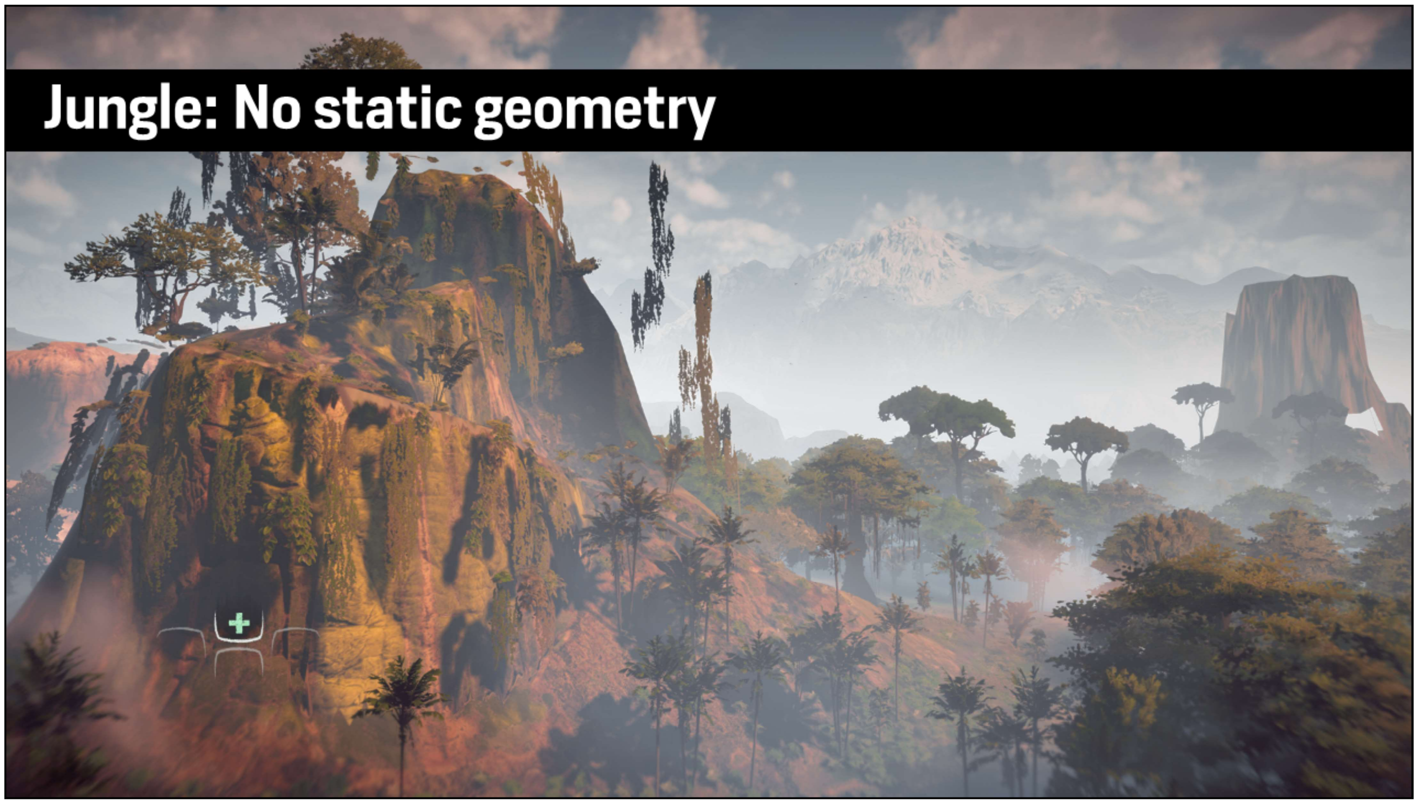
Changes we made for batching were very valuable, removing about two thirds of the work from the render pipeline without changing query times much if at all.

Jungle



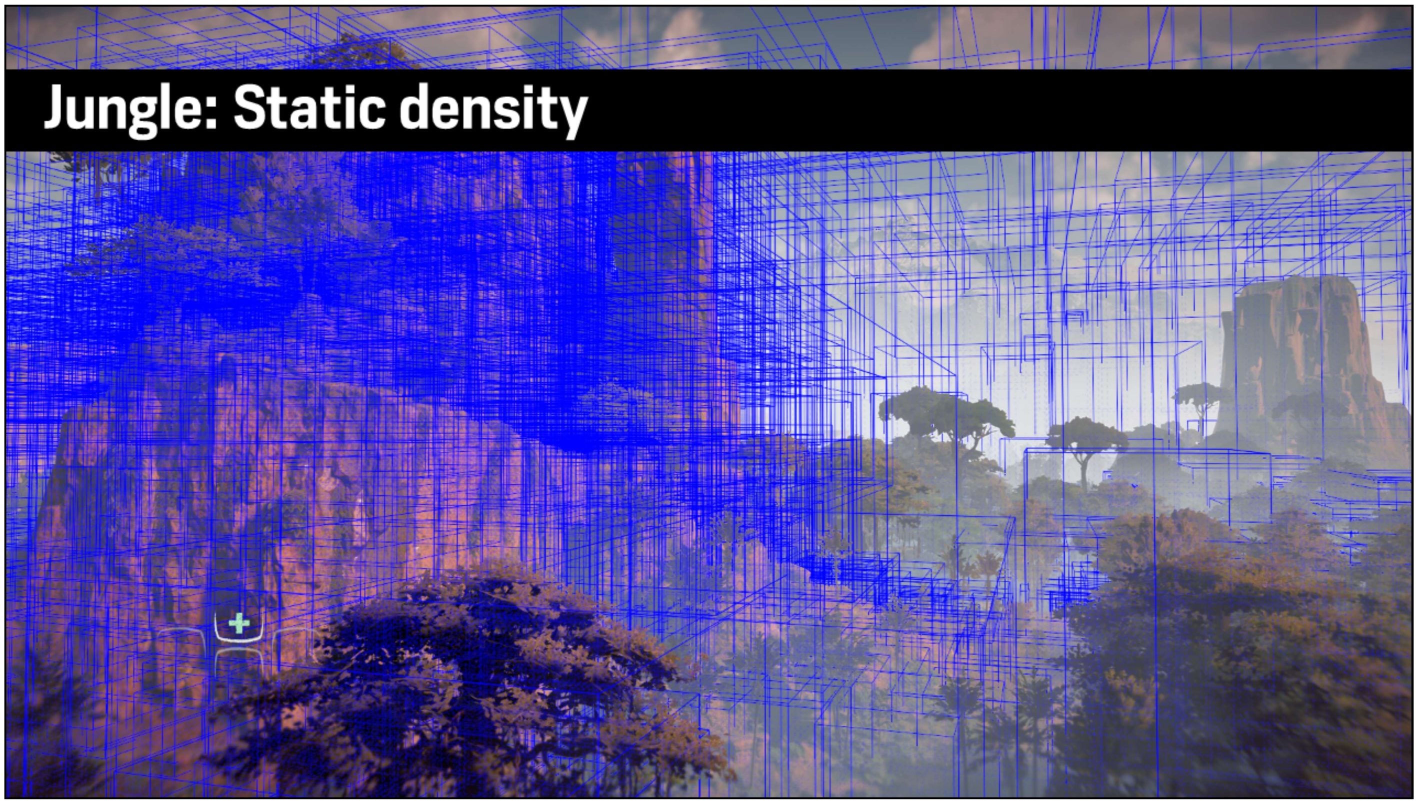
I thought I'd show some pics to finish up, to give an idea of what the static content actually looks like. Here's a chunk of random jungle, which doesn't have a lot of static content.

It's mostly terrain and vegetation and those are both dynamic systems with a lot of regeneration at runtime.



Jungle: No static geometry

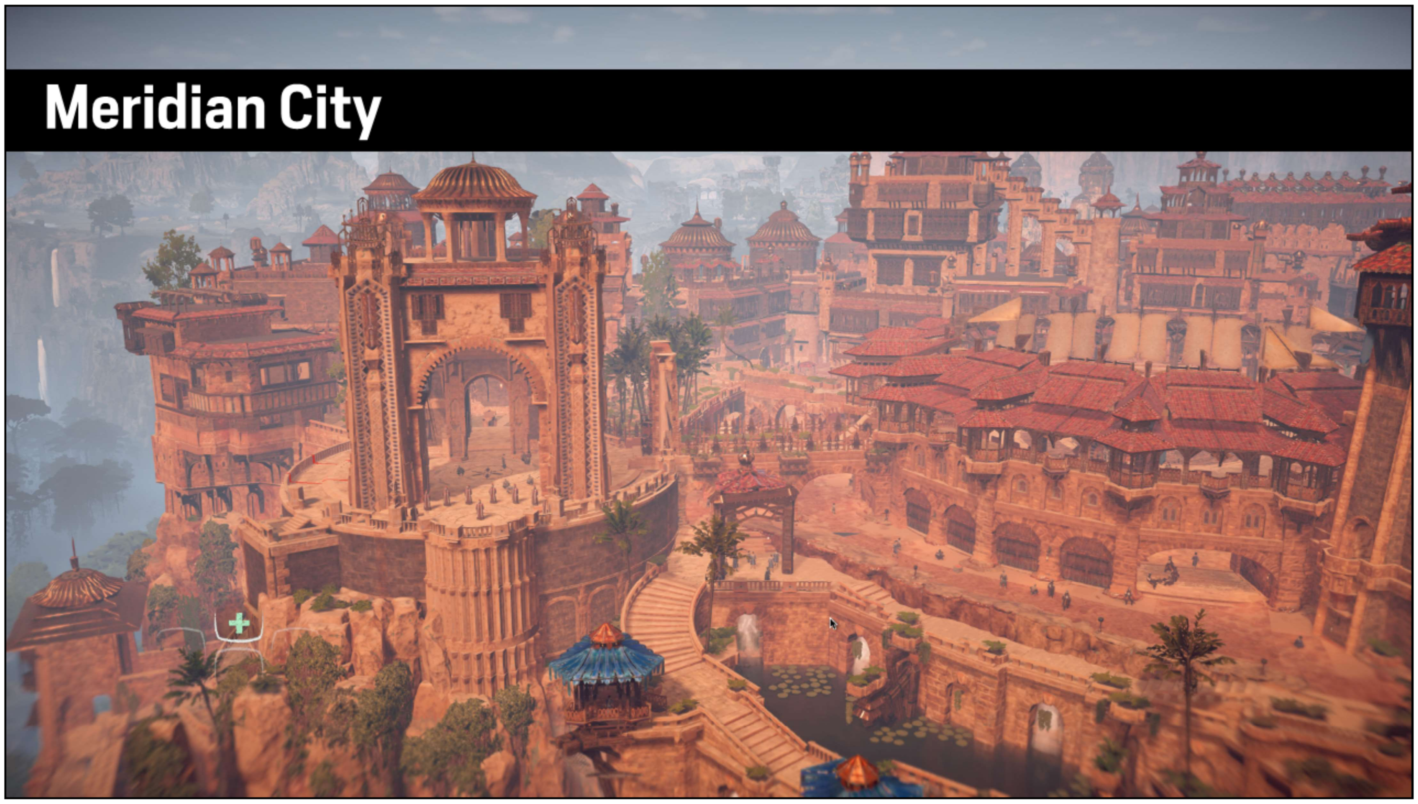
If we remove the static geometry, we can see the rocks and some hand-placed trees vanish.



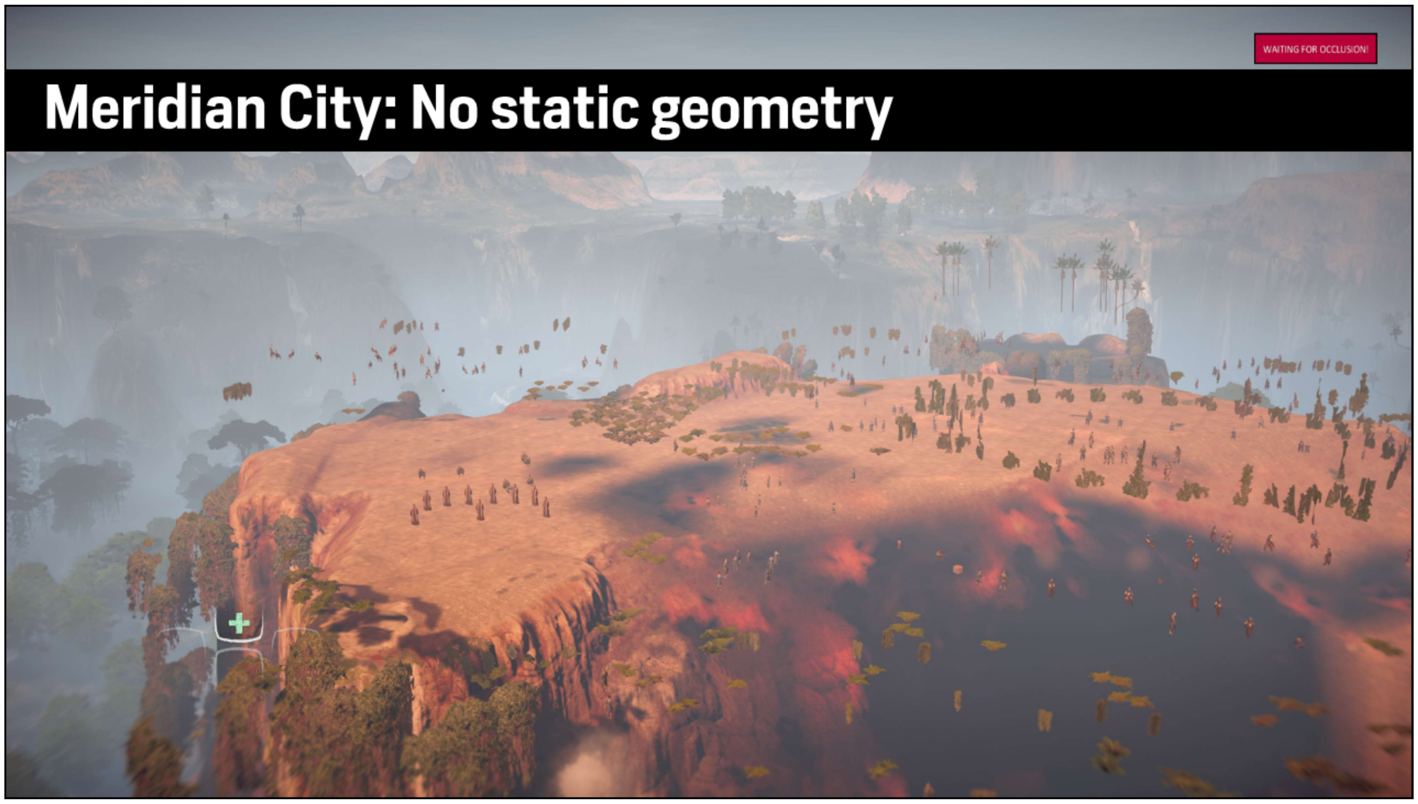
Jungle: Static density

But if we visualise the static DrawableSetups there are still an awful lot of them!

Meridian City



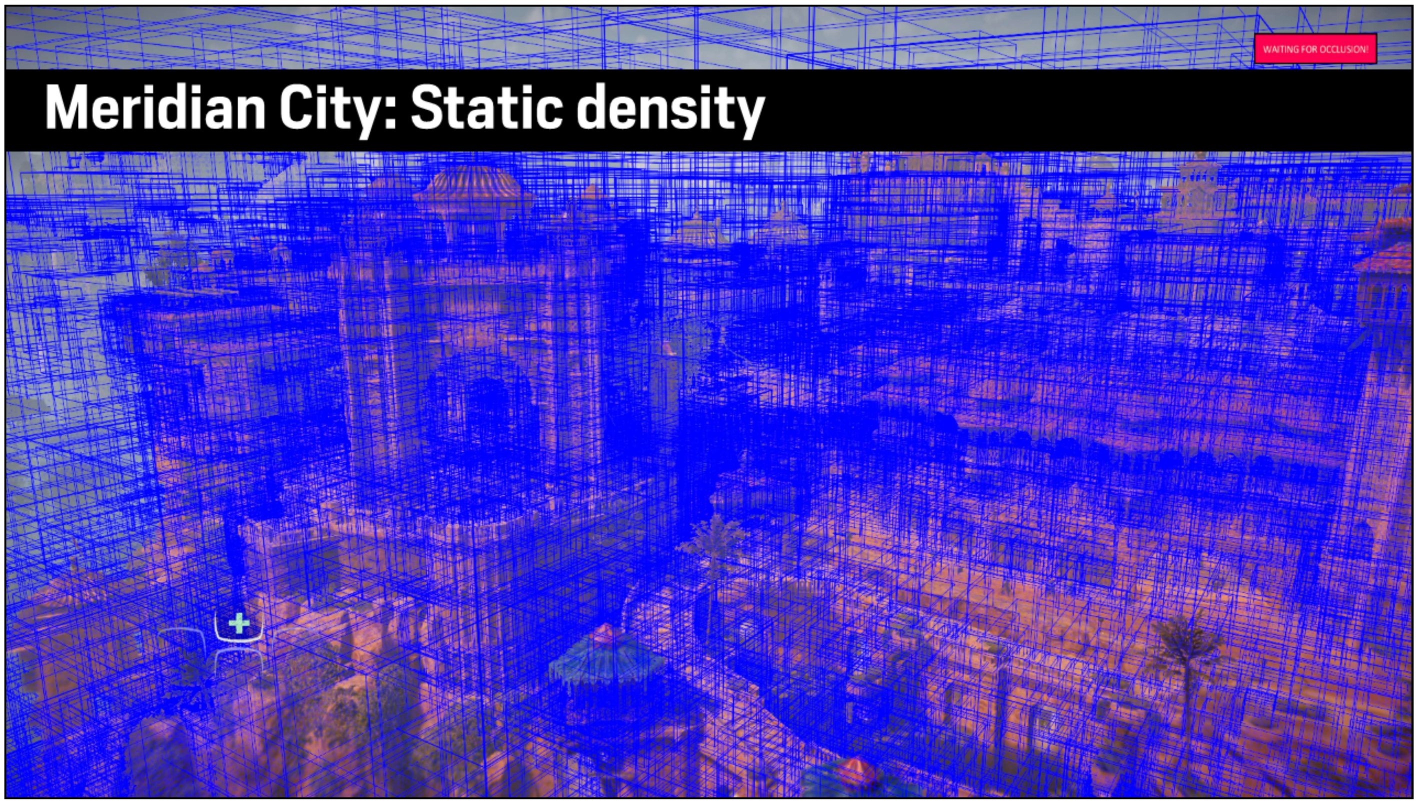
Meridian is the opposite, it’s nearly all static geometry



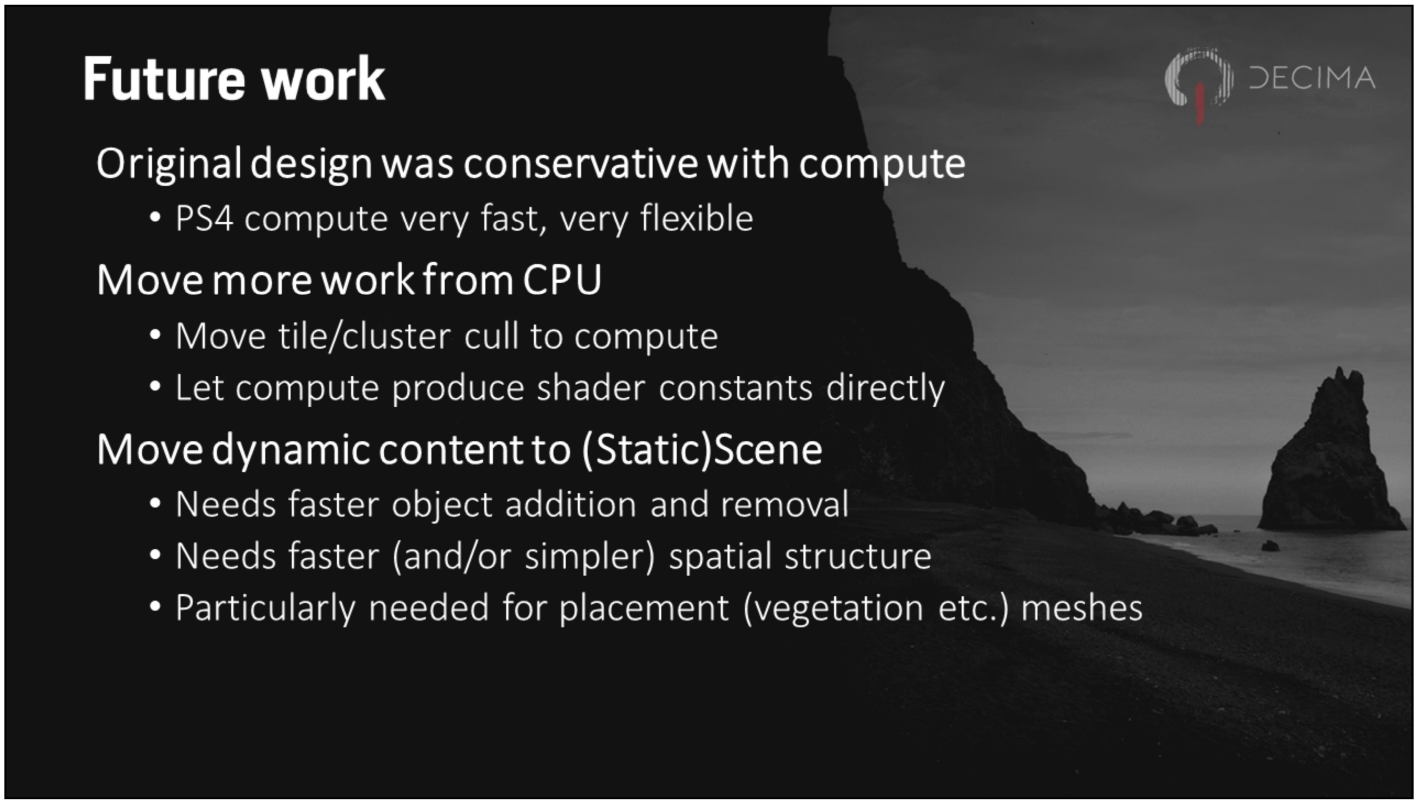
Meridian City: No static geometry

As you can see, there's just terrain, a bit of vegetation, and the animated characters left.

Meridian City: Static density



And the static scene density is so high that you can’t really make it out!



Future work

Original design was conservative with compute

- PS4 compute very fast, very flexible

Move more work from CPU

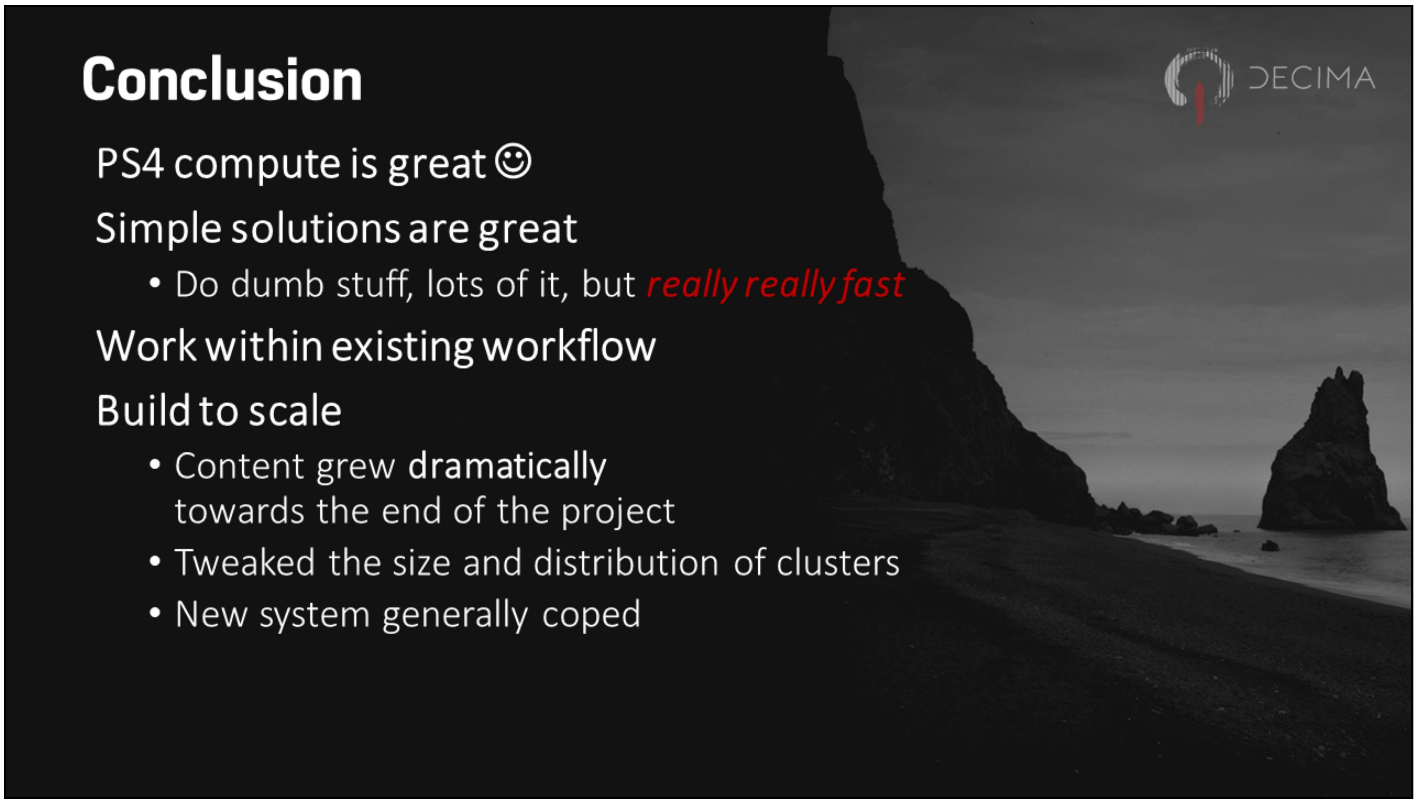
- Move tile/cluster cull to compute
- Let compute produce shader constants directly

Move dynamic content to (Static)Scene

- Needs faster object addition and removal
- Needs faster (and/or simpler) spatial structure
- Particularly needed for placement (vegetation etc.) meshes

Where are we going from here? Well, we were pretty cautious with compute since this was one of the first complicated things we did with it. So we want to move more of the system from the CPU to the GPU, and ideally much of the query output on the GPU so the renderer doesn't need to touch it, just feed it forward to the shaders later.

More importantly, we want to have the static scene take on more dynamic content, in particular the placed meshes, which don't change all the time but often enough that the current tile creation wouldn't be quick enough.



Conclusion

PS4 compute is great ☺

Simple solutions are great

- Do dumb stuff, lots of it, but *really really fast*

Work within existing workflow

Build to scale

- Content grew **dramatically** towards the end of the project
- Tweaked the size and distribution of clusters
- New system generally coped

To conclude, compute is great on PS4 and well worth making use of.

I also think that simple solutions are great – do things that aren't complicated, but do lots of them fast.

Work within the workflow to avoid slowing down or annoying your content creators, and build to scale for all the content they're going to create. For Horizon the content grew very fast at the end of the project but by tweaking our minimum sizes we could generally cope with this quite well.

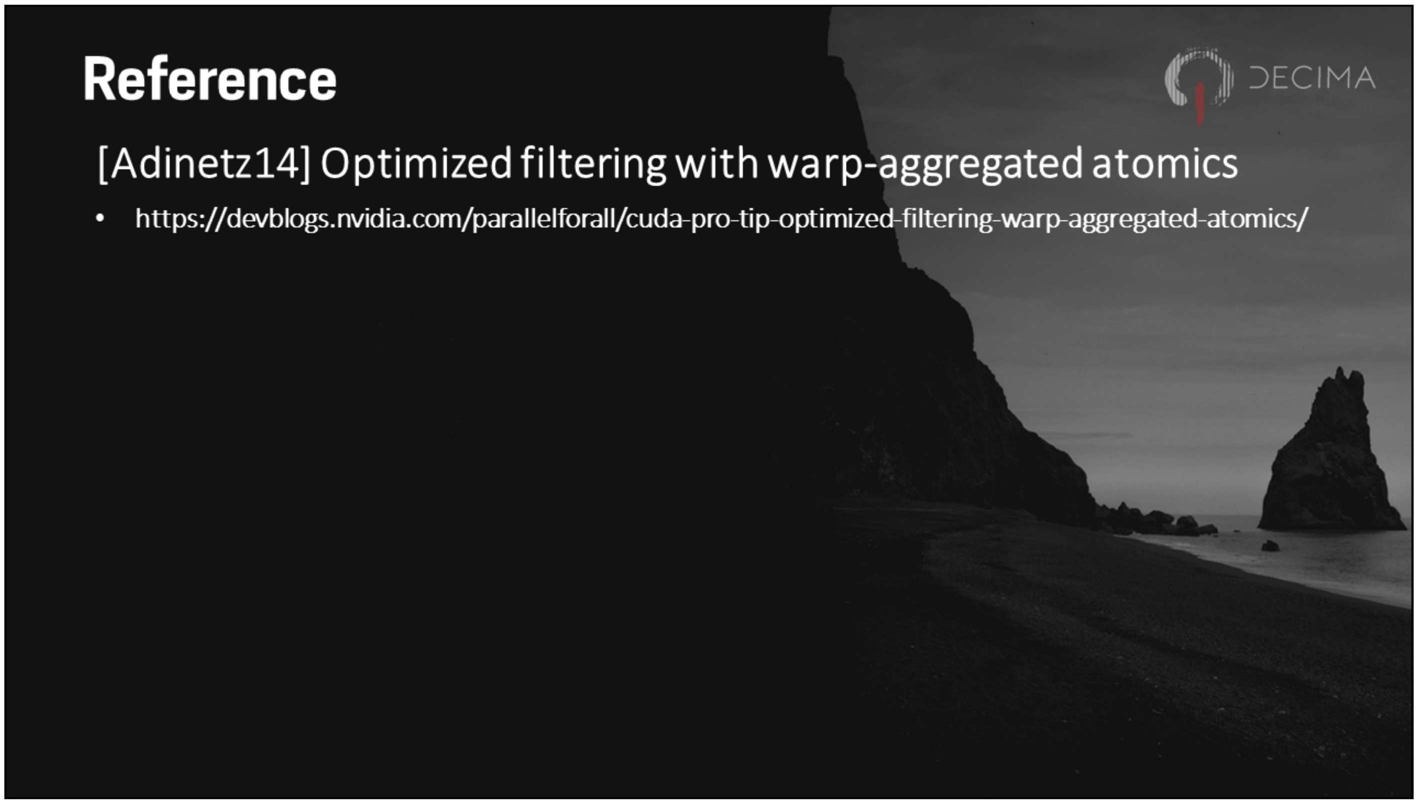
DECIMA

QUESTIONS?

Special thanks to Michiel, Jeroen, Roland and the Guerrilla Tech Team for making this talk possible today!

will@secondintention.com

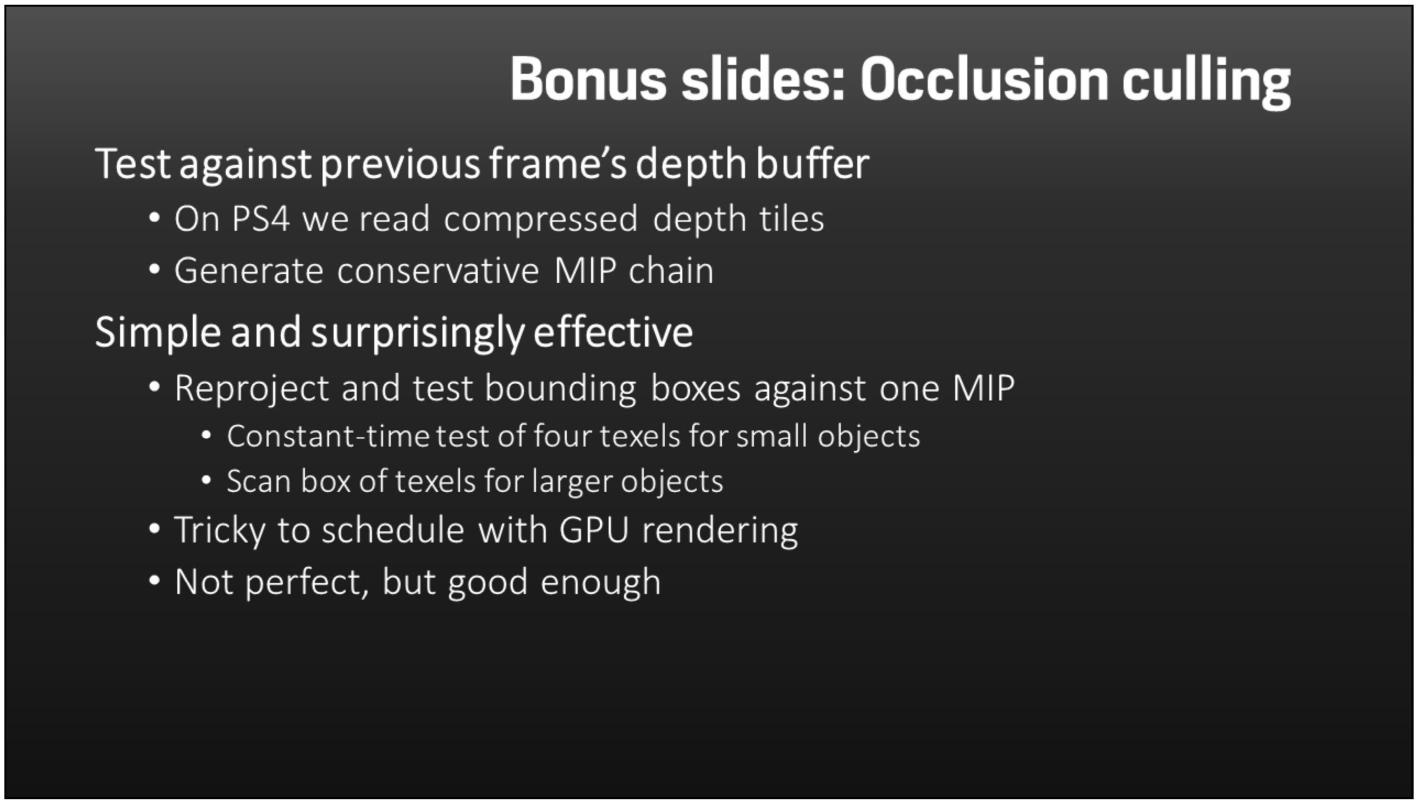
Thanks to my colleages at Guerrilla, particularly, Michiel, Jeroen and the tech team for help and support, and Roland for the awesome slide designs.



Reference

[Adinetz14] Optimized filtering with warp-aggregated atomics

- https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/



Bonus slides: Occlusion culling

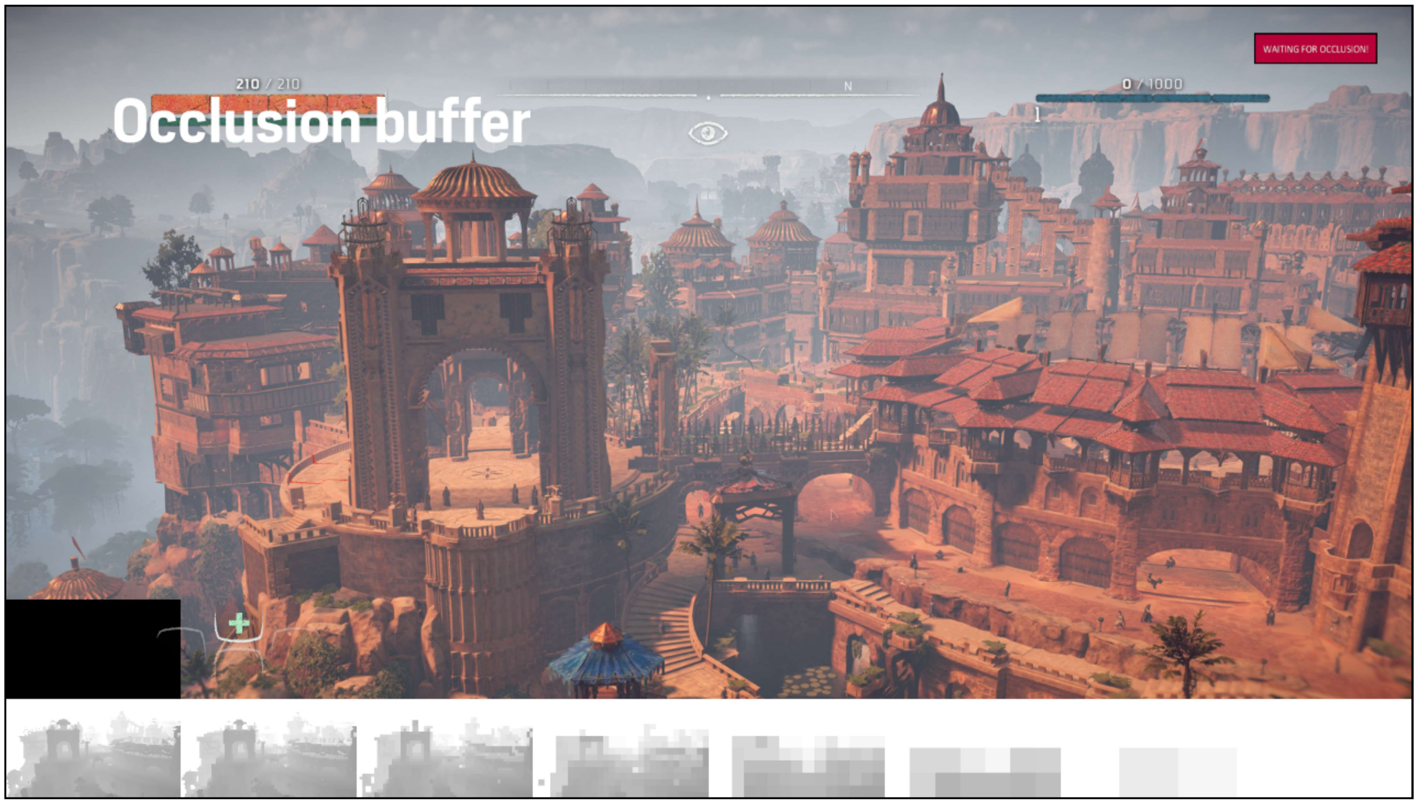
Test against previous frame's depth buffer

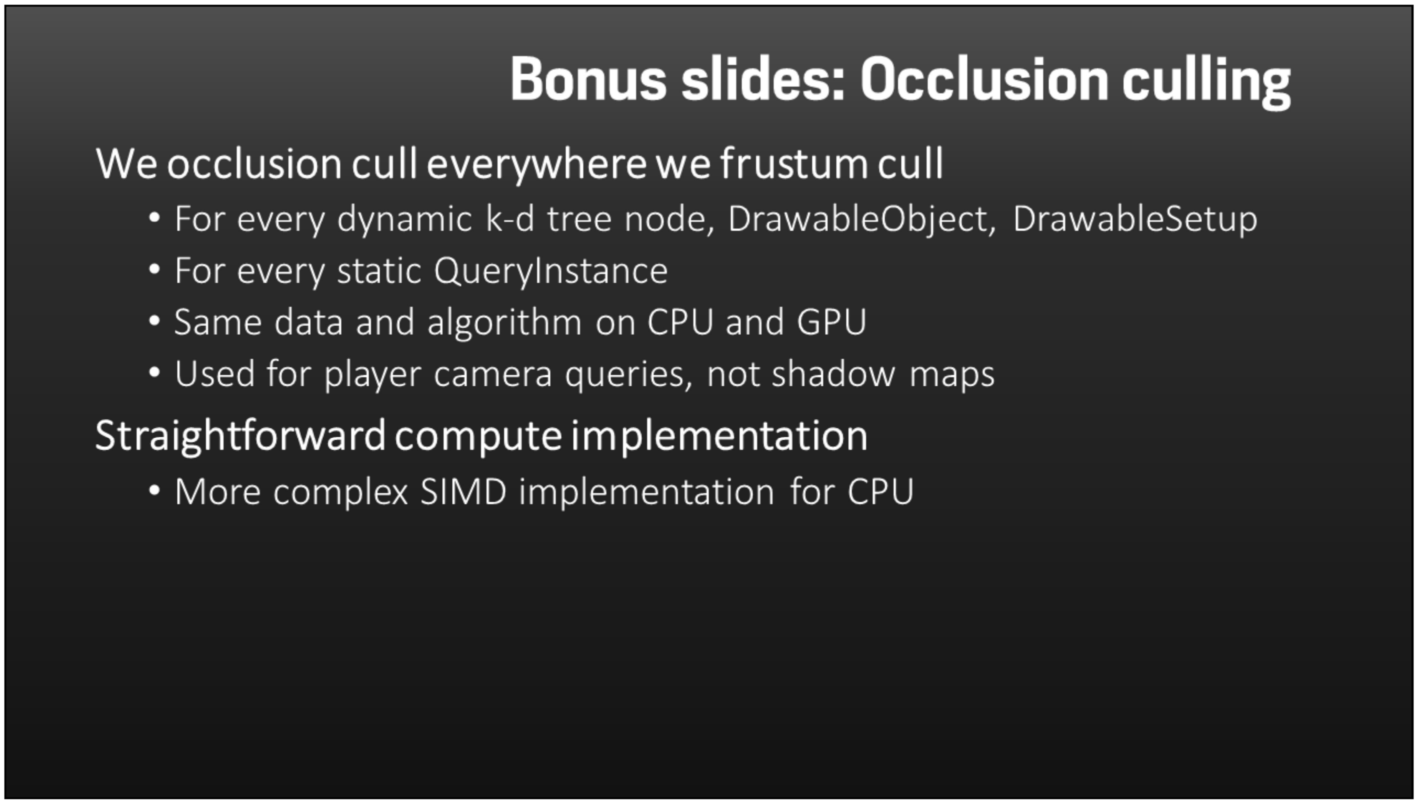
- On PS4 we read compressed depth tiles
- Generate conservative MIP chain

Simple and surprisingly effective

- Reproject and test bounding boxes against one MIP
  - Constant-time test of four texels for small objects
  - Scan box of texels for larger objects
- Tricky to schedule with GPU rendering
- Not perfect, but good enough

Occlusion buffer





Bonus slides: Occlusion culling

We occlusion cull everywhere we frustum cull

- For every dynamic k-d tree node, DrawableObject, DrawableSetup
- For every static QueryInstance
- Same data and algorithm on CPU and GPU
- Used for player camera queries, not shadow maps

Straightforward compute implementation

- More complex SIMD implementation for CPU