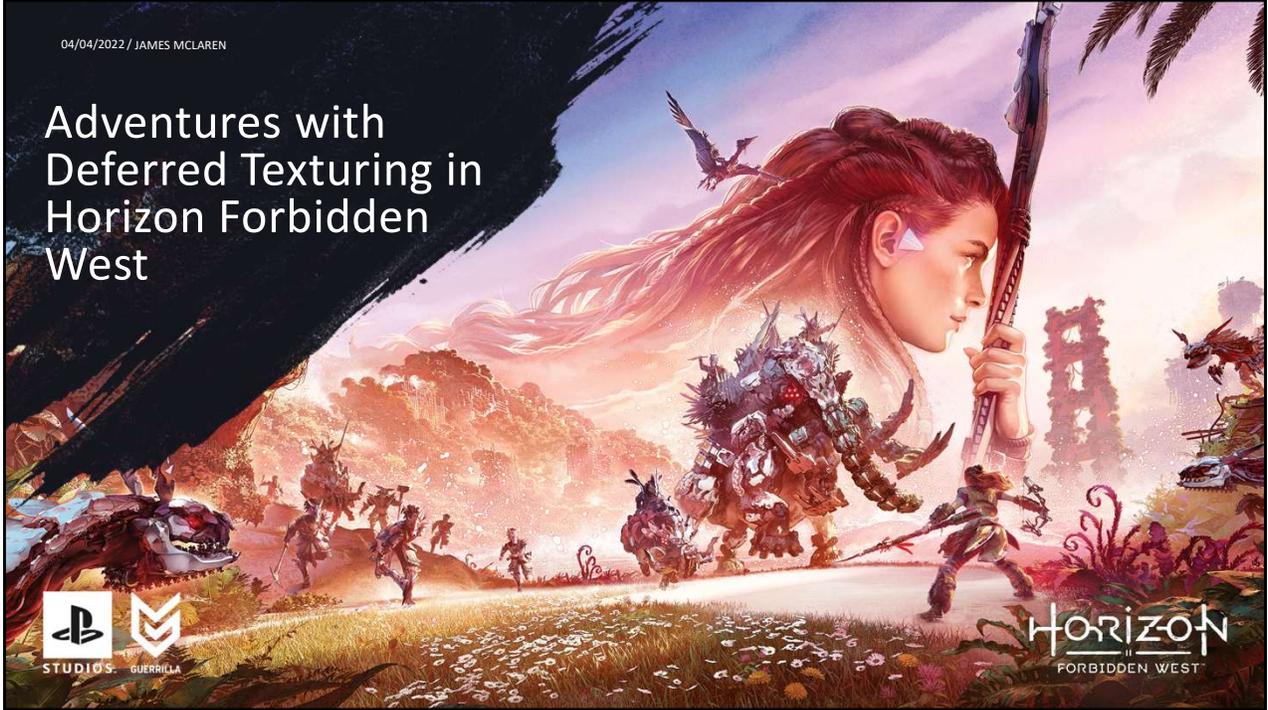


04/04/2022 / JAMES MCLAREN

Adventures with Deferred Texturing in Horizon Forbidden West



“The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.”

— Marcel Proust

In Horizon Forbidden West Aloy goes on a great adventure with many new landscapes in a far off land, but in order to realize some of those we had to look at our rendering pipeline anew, and that’s what I’ll talk about today.



Hello, and thank you for coming.
My name is James McLaren and I'm a Senior Principal Tech programmer at Guerrilla.

In this talk I'm going to be going through some of the details of the Deferred Texturing system we made primarily to accelerate our foliage for Horizon Forbidden West.

Talk Overview

- ▶ Previous Foliage system.
- ▶ What deferred texturing is.
- ▶ Overview of our system.
- ▶ Dive into some details
- ▶ Detour into VRS.
- ▶ Talk about performance



HORIZON
FORBIDDEN WEST



3 3

I'll be starting out by talking about the problems of rendering foliage and reviewing our previous system, after which I'll move on to talk a little bit about what deferred texturing actually is. I'll then give a high level overview of our system, before diving down into some of its details and talking a little bit about our variable rate shading implementation and finally we'll finish off with some performance numbers.

Horizon Forbidden West

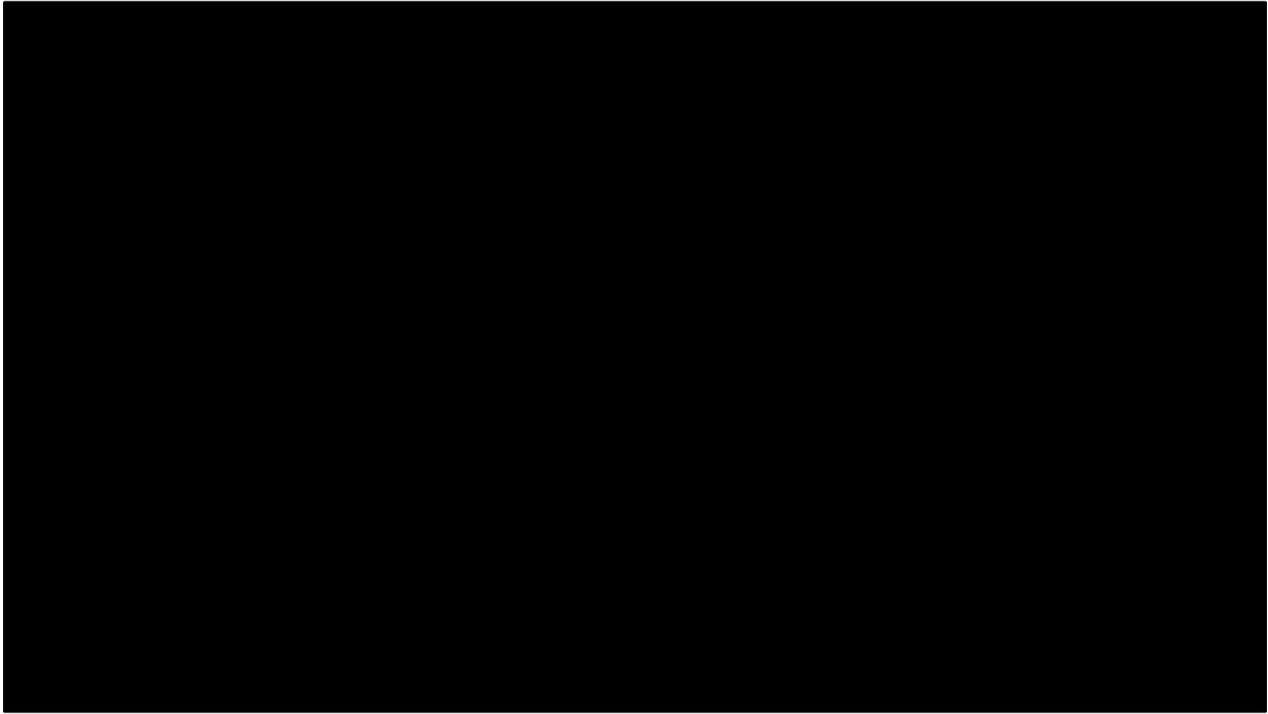


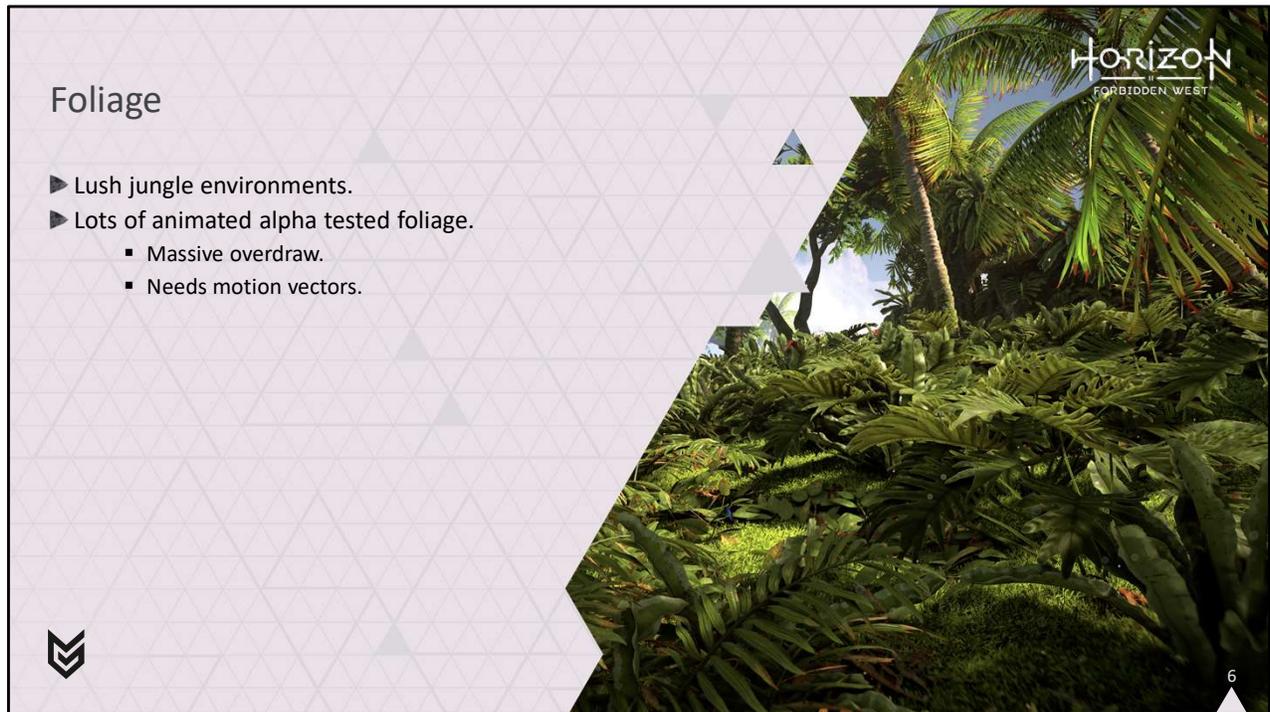
But before that, in case you don't know anything about it, let me introduce Horizon Forbidden West.

Horizon Forbidden West is an open world adventure game for the PS4 & PS5 that launched in February this year as the follow up to Horizon Zero Dawn.

In it you play Aloy, a Nora brave, who has to save the world, mostly by fighting lots and lots of dangerous machines.

Here's a trailer to bring everyone up to speed.

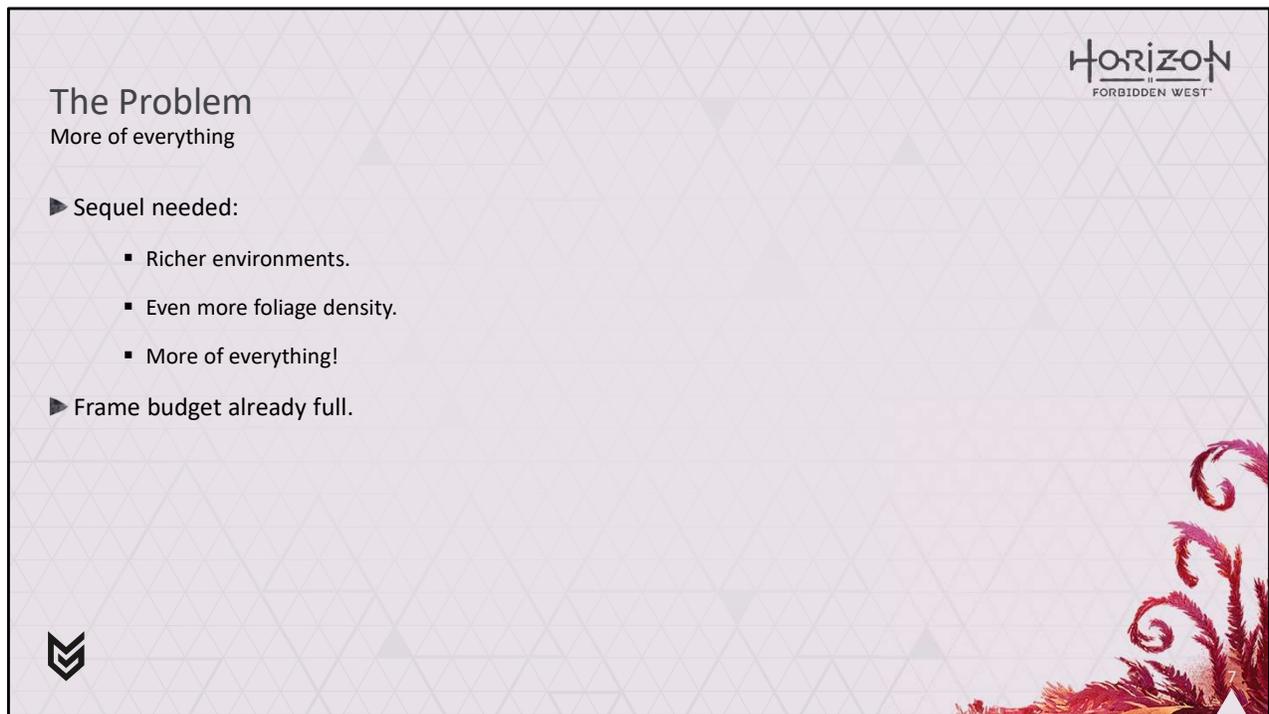




In the world of Horizon we have many diverse environments, including places like forests and lush jungles.

These are typically a challenge to render efficiently. Most of our plants are made up of alpha tested geometry, and there can be a very large amount of overdraw.

The geometry is also animated, and we need to generate correct motion vectors for it to feed into our TAA and motion blur so there's no easy cuts we can make.



In Horizon Zero Dawn we already had a highly optimized foliage system, but for Horizon Forbidden West, the art department wanted an even higher density of assets, leading us to wonder how we were going to find the extra milliseconds this would entail.

This was going to be a really tight squeeze with everything they wanted in the project, especially as we still needed to target PS4.

In case you're wondering how the art department looked to us when asking for more of all the things.

Here's a dramatic re-enactment in game.



So we were pretty motivated to see what could be done.

The Problem

Horizon Zero Dawn's Foliage

- ▶ Depth pre-pass.
 - Simple shader reads alpha texture and discards.
- ▶ Geometry-pass.
 - Main shader run through geometry again with depth equal test.
- ▶ Avoids outputting to the G-Buffer for a pixel more than once.

HORIZON
FORBIDDEN WEST



9

Rendering foliage can be expensive.

There is lots of overdraw and it tends to be alpha tested which can bring with it added expense because it can disable some of the hardware's early Z optimizations.

In Horizon Zero Dawn we worked around this by using a depth pre-pass to do the alpha test, followed by rendering the foliage again in the geometry pass without the alpha test, but using a depth equals test to ensure we only shaded the visible pixels. This is a pretty good solution as it avoids us writing to a foliage pixel in the G-Buffer more than once.

The Problem

Horizon Zero Dawn's solution

HORIZON
FORBIDDEN WEST

► But...

- Transforms all the geometry twice.
- Often difficult for the GPU to keep itself full.
- Suffers from major issues with quad overdraw.



10

However, there are some issues with this approach.

All the geometry must be transformed twice by the GPU.

Due to this it's also often hard for the GPU to keep itself full when rendering everything again, with lots of gaps where vertices are transformed but generate no pixel work in the second pass.

Also, with the fine detail typical in foliage this approach can suffer from large amounts of quad overdraw.

Pixel Quads

- ▶ Pixel shader always shades 2x2 quads.
 - Necessary to calculate derivatives via finite differencing.
 - Helper lanes must be generated if a triangle doesn't cover the whole quad.
 - Also happens with the depth equal test in geometry pass.



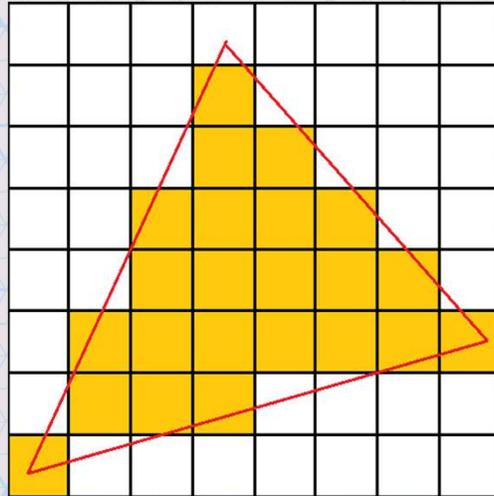
If you aren't familiar with Quad overdraw, then let me give you a very quick overview. The pixel shading hardware in a GPU needs to be able to automatically calculate derivatives so that it can select the right mip-maps and correctly filter textures when they are sampled.

To do this it always shades in 2x2 quads.

This way it can use finite differencing to subtract the UV coordinates in one lane from another and calculate the derivatives it needs for texture sampling.

Quad Overdraw Visualization

HORIZON
FORBIDDEN WEST



12

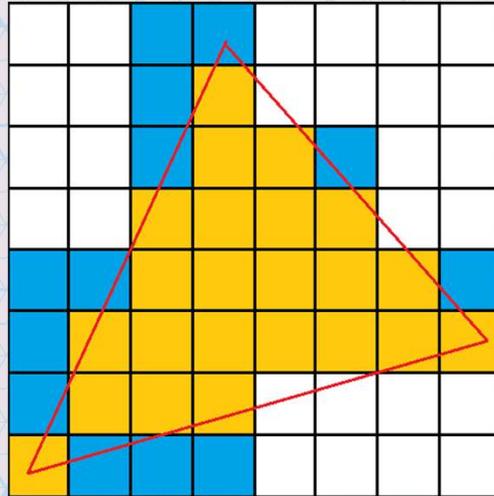
Let's take a moment to visualize this. Here is a triangle that we are going to pretend we're rasterizing.

The yellow pixels show the result of the rasterization.

Unfortunately, the HW doesn't schedule pixels to be shaded it schedules quads

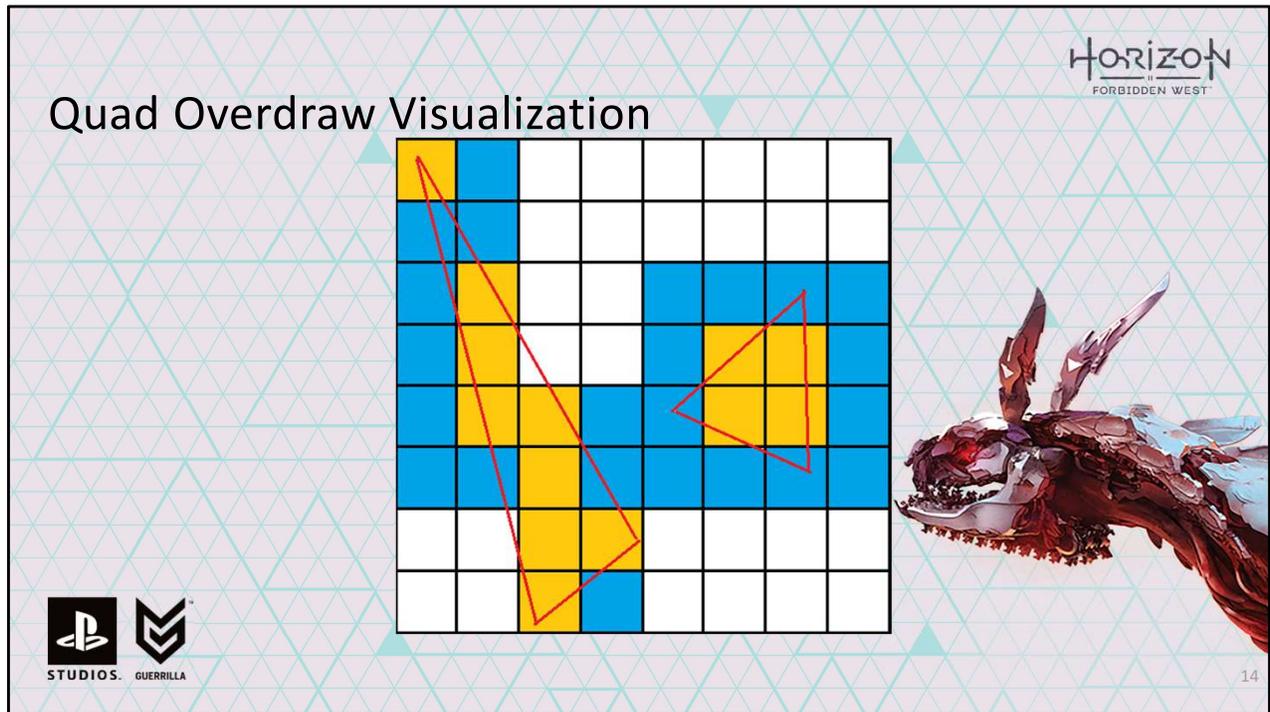
Quad Overdraw Visualization

HORIZON
FORBIDDEN WEST



13

So, we end up doing shading work for all the blue pixels here as well.



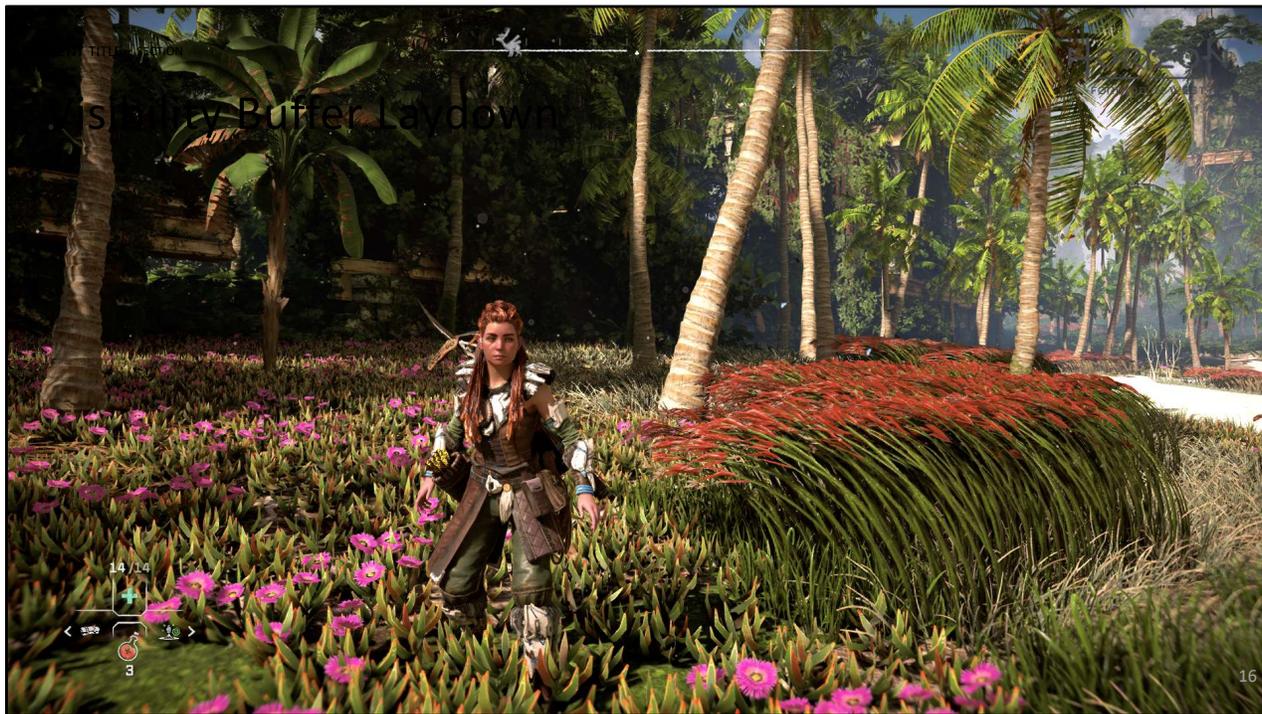
And as you can see as we get down to small triangles, this problem gets worse. The triangle on the right here is particularly sad. It outputs 4 pixels, but we had to do the shading work for 16 pixels in order to calculate our derivatives. This is what we're referring to when we talk about quad overdraw.

With foliage we can sometimes get away with large triangles, but usually accompanied by alpha testing, in order to mimic fine detail. This then ran into exactly the same quad overdraw issues in our old system when doing the depth equal test in the geometry pass.

The Adventure Begins



So with this knowledge in hand we are now ready to head out on our new adventure, searching for a better solution.



This is Aloy ready to head out on her adventure.

Deferred Texturing

- ▶ Render a Visibility Buffer containing Primitive IDs.
- ▶ Analyze its contents.
- ▶ Use the primitive IDs for pixels to fetch their vertex and batch properties to shade them.



The great adventure that we decided to embark on was to try to implement Deferred Texturing.

The basic idea here is that before rendering any opaque geometry we do a pre-pass over the scene to generate a visibility buffer.

This is very similar to a depth pre-pass except that we also write Primitive IDs to an additional render target.

Once done the Visibility Buffer will contain the Primitive ID of the topmost triangle for each pixel.

We can then analyze this buffer and use the Primitive IDs to shade the pixels.

Deferred Texturing

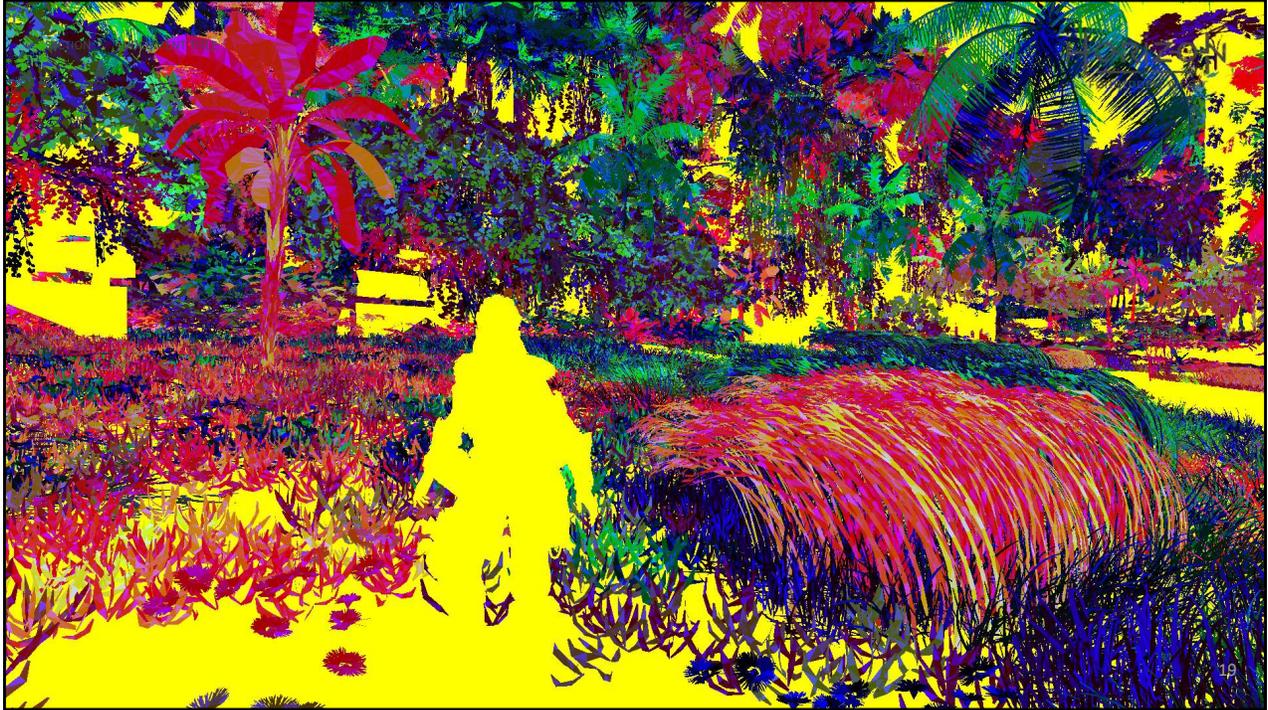
- ▶ Render a Visibility Buffer containing Primitive IDs.
- ▶ Analyze its contents.
- ▶ Use the primitive IDs for pixels to fetch their vertex and batch properties to shade them.
- ▶ Profit!
- ▶ Based on The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading.



This idea was originally described in a paper from Intel: The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading.

This can be applied to both forward and deferred renderers.

In HFW we have a deferred renderer, so the pass to shade the pixels will generate our Gbuffers.



And this is a colour coded visualization of that same scene showing all the primitive IDs that we will put in the visibility buffer.

As you can see for us, only some pixels actually have a valid primitive ID, the yellow pixels here are pixels that are not included in our visibility buffer.

Deferred Texturing

Advantages

- No quads!
 - So, we are (mostly) freed from quad overdraw issues.
- No overshading.
- No second pass through **ALL** the geometry.
- Saves bandwidth.

Disadvantages

- No quads!
 - We need to calculate derivatives ourselves.
- Need to manage efficient dispatch of work to shade pixels.



So why would we do this?

Well, for one thing we can hope to only ever shade the visible pixels so there is no overdraw.

In theory we can also free ourselves from any quad overdraw issues, depending on how we process things.

Also, we don't need to make a pass through transforming all our geometry again, possibly just what's on screen, or possibly none at all, depending on how we implement things.

It also helps save bandwidth, at least in comparison to a naïve deferred solution.

It's not all sunshine and roses though.

Deferred Texturing

Advantages

- No quads!
 - So, we are (mostly) freed from quad overdraw issues.
- No overshading.
- No second pass through **ALL** the geometry.
- Saves bandwidth.

Disadvantages

- No quads!
 - We need to calculate derivatives ourselves.
- Need to manage efficient dispatch of work to shade pixels.



Because the hardware isn't creating quads for us, we can no longer use finite differencing to generate our derivatives and must calculate them ourselves. Also, we now have a significant task ahead of us in figuring out how to efficiently dispatch or draw all the pixels that are in the visibility buffer to resolve our materials.

Like any good technique, this has already started to come in various flavors.

Flavors of Visibility Buffer

Original Paper

- Write thin 32 bit visibility buffer encoding triangle ID and instance ID,
- Build work list of material tiles.
- Dispatch compute shader tiles per material.
- Transform vertices per pixel.
- Barycentrics and Ddx Ddy and tangent frame calculated per pixel.
- Lit pixels directly output.



In the original paper they write out a simple visibility buffer with the triangle ID and instance ID encoded in a single 32 bit uint.

They then do some analysis on this, and build a set of screen space tiles that contain each unique material.

After which they then use indirect dispatch to launch compute shaders to process the set of tiles they've recorded for each material,

The primitive ID is read and used to read vertex information, barycentrics are computed and any vertex transformation work is done along with the pixel work, and is thus repeated for each pixel.

The derivatives for any attributes are also manually calculated in the compute shader that process the tiles along with the tangent frame if required.

At the end the shaded and lit results are written out with no intermediate Gbuffer, which makes it very low bandwidth.

Flavors of Visibility Buffer

Horizon
FORBIDDEN WEST

Dawn Engine

- Fat visibility buffer (4 x 32 bit)
 - UV, ddy/ddy, material ID, tangent space.
- Material ID pass
 - Material ID copied into 16-bit depth buffer.
- Shade materials with pixel shaders via creative abuse of depth test.

23

At the opposite end of the spectrum, we have something like the Dawn engine. Here they take a similar approach, but instead of calculating derivatives, barycentrics and tangent frames in the back end, when they resolve the materials, that work is instead front loaded, so UV, derivatives, material ID and tangent space are packed into a very fat visibility buffer during laydown. The visibility buffer is then analysed and material IDs are copied into a 16 bit depth buffer. They then shade each material by using what I like to refer to as “creative abuse of the depth buffer”, They then use a depth equals test per material ID to select the appropriate pixels and shade them with a pixel shader by drawing a screen space quad that bounds all the objects using that material and setting the depth of the quad to be the same as the material ID.

Flavors of Visibility Buffer

Everything in between

- A deferred material rendering system [h3r2tic]
 - PrimitiveID + Barycentrics.
- Nanite [Karis]
 - Static geometry + GBuffer output.
- Activision's Geom Pipeline [Drobot]
 - 2 UINT's export for implicit geometry (Foliage).

HORIZON
FORBIDDEN WEST

24

And then there are some other variants that are somewhere between those two. Tomasz Stachowiak's deferred material system writes out barycentrics along with the primitive ID and shades in compute shades by generating lists of pixels per material. More recently we have Nanite and the geometry pipeline from Activision. Nanite works with compressed software rasterized geometry clusters and produces a relatively thin visibility buffer with cluster and triangle information packed together with depth in a 64 bit UINT. It then writes out a GBuffer via pixel shaders when resolving it's materials, using a variant of the creative depth buffer abuse trick. Activision's Geometry pipeline, also works with geometry clusters, and writes out a thin visibility buffer for most geometry, but for foliage they write out 64 bits of information across two render targets to encode Normal, texture LOD and UV which keeps the work needed in the material resolve step to a minimum, by limiting the material complexity.

* If you're interested in reading around the subject I would also heartily recommend checking out John Hable's blog post [Visibility Buffer Rendering With Material Graphs](#)

Our Requirements

HORIZON
FORBIDDEN WEST

- ▶ Handles animated geometry.
 - Including motion vectors.
- ▶ Efficiently handles a number of different materials.
- ▶ Had to work on both PS4 & PS5.
- ▶ Can't use a lot of memory.
 - 10s of MB available on base PS4.
- ▶ Fast visibility buffer laydown.
- ▶ Output to G-Buffer.



25

Given the range of possibilities for setting up a deferred texturing system, we started looking at what we needed it to do and tried to figure out what would fit our requirements.

All our foliage is animated, so support for dynamic geometry was a must.

We also have several different materials for different types of foliage, so we would need to efficiently support switching between them.

The approach was also going to have to work on both PS4 and PS5 as we were going to launch on both.

Because we were going to be running on PS4 whatever we did was going to have to be very light on memory. We could afford 10's of Mb not 100s.

Also, foliage typically has lots of overdraw, so the laydown of the visibility buffer would need to be very cheap, preferably just a single 32 bit export.

We were also going to want this to integrate nicely with other non-deferred textured geometry and so we wanted it to output to our G-Buffer.



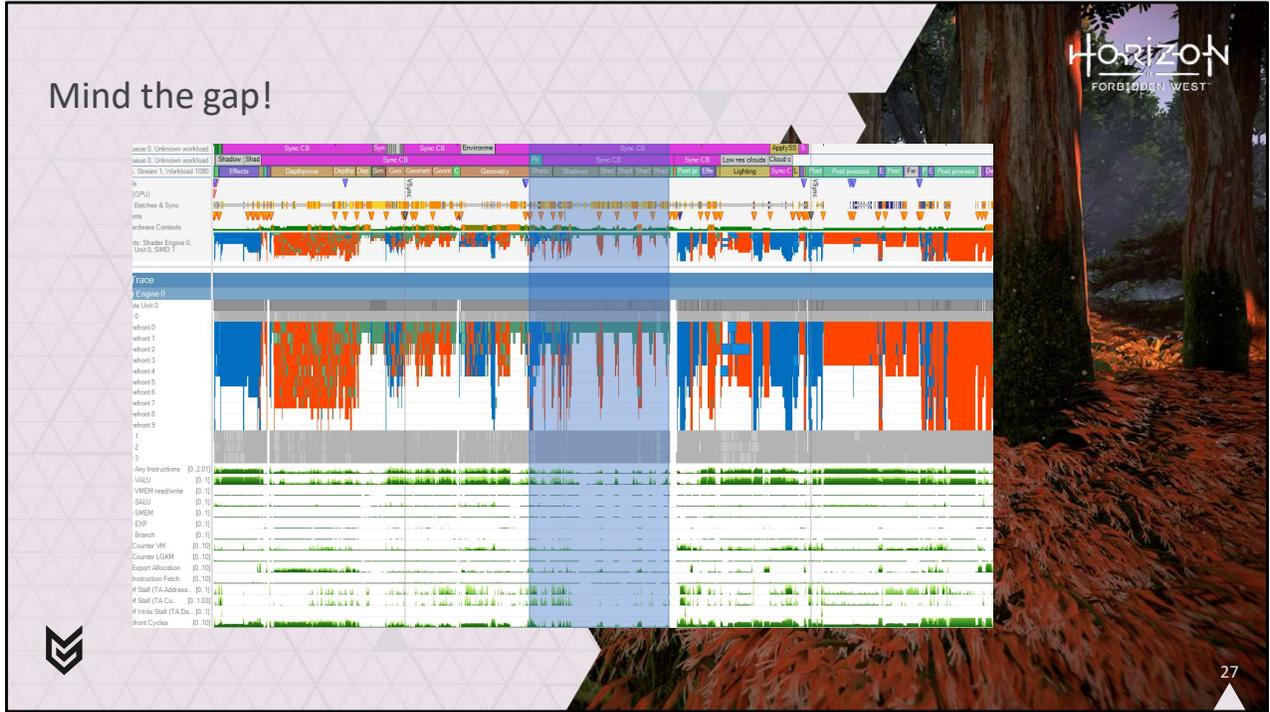
Now during our musings, we also started looking at our frame to see where all of this work was going to fit.

My initial prototype had used pixel shaders.

I started this process by hand modifying some of our generated shaders, to figure out what would need to happen to them if we were to use Deferred Texturing.

It was promising, but I wasn't entirely convinced it was the right way to go.

Anyway, when I started looking at the frame what I saw was this...



Mind the gap!

So, you can see that we had a nice spot where the cascaded shadows are rendered that was typically filled with a lot of vertex work and not a great deal of pixel work. So, I started wondering if we could shade our foliage materials in this gap? However, this meant that we would need to use the Async Compute pipes to do the shading and use compute shaders to fill our G-Buffer.

Pixels with Compute

Possible approaches?

- ▶ Identify 8x8 tiles containing particular materials/batches.
 - DispatchIndirect() call to shade the tiles.
 - Suffers from unfilled tiles.
 - Like unfilled quads but worse!



So, with this in mind I started thinking about how we were going to do our pixel shading with compute

This is not as simple as it first appears.

A simple naïve approach is to try to identify small tiles that at least partially cover the same materials and do sets of dispatch indirects that shade lists of tiles.

This is what the original visibility buffer paper did

However, it's quite obvious that without the HW to identify quads of the same materials and pack them into waves for us that this is going to lead to large amounts of lanes in each tile doing no work.

This is like the unfilled quads problem that pixel shaders can have with small triangles, but actually much worse!

Pixels with Compute

Possible approaches?

- ▶ Identify pixels contributing to particular materials/batches and make lists of pixel commands.
 - DispatchIndirect() call to shade.
 - Possible our list of commands bounces around memory a lot. Bad for L2.
- ▶ Both potentially need a dispatch per batch – problem on Async.
 - Can become CP bound.



The other simple option is to identify pixels of the same materials and form lists of pixels to shade.

This is a much better option however there are still some issues with this approach. Because we're processing materials in an effectively random order we can bounce around a lot in screen space.

This is not too much of a problem if we deal with large groups of pixels that have the same material and batch, but for foliage it's very easy to get a many batches intermingled in the same screen space area.

This means that when we'll potentially ending up pulling the same memory into and out of L2 again and again as we shade.

The other problem is that if we have many batches each with their own combination of shader, uniforms and textures they will each need a separate dispatch (unless of course we want to allow divergence in our waves).

This can also become prohibitively expensive if we are using the compute pipes, as each dispatch can take significantly longer for the CP to process than it would on the GFX pipe so this can lead to us becoming quite CP bound.



Idea: Loose Tiling?

- ▶ Sort pixels so that pixels within a tile with the same shader are processed together.
- ▶ Use large 128x128 pixel tiles.
- ▶ Sort the pixels in a tile into complete waves (64 threads each) of the same batch.
 - Store a command for each wave denoting the batch.



So in the spirit of adventure...

We decided to try to combine both approaches.

We wanted the good spatial locality of tiles, and we wanted the packing flexibility of lists of pixels.

What we came up with was what I'm calling a "loose tiling" approach.

The basic idea is to work with large tiles of 128x128 pixels.

Within a tile we identify pixels that use the same shader, but not necessarily the same constants and textures.

This means that they can come from different batches.

We can then take these pixels and sort them into complete waves with the same constants and textures.

In addition to commands for each pixel, for each wave we store a "wave command" that tells us the tile we are processing, the batch we are processing and where the pixel commands start.

Idea: Loose Tiling?

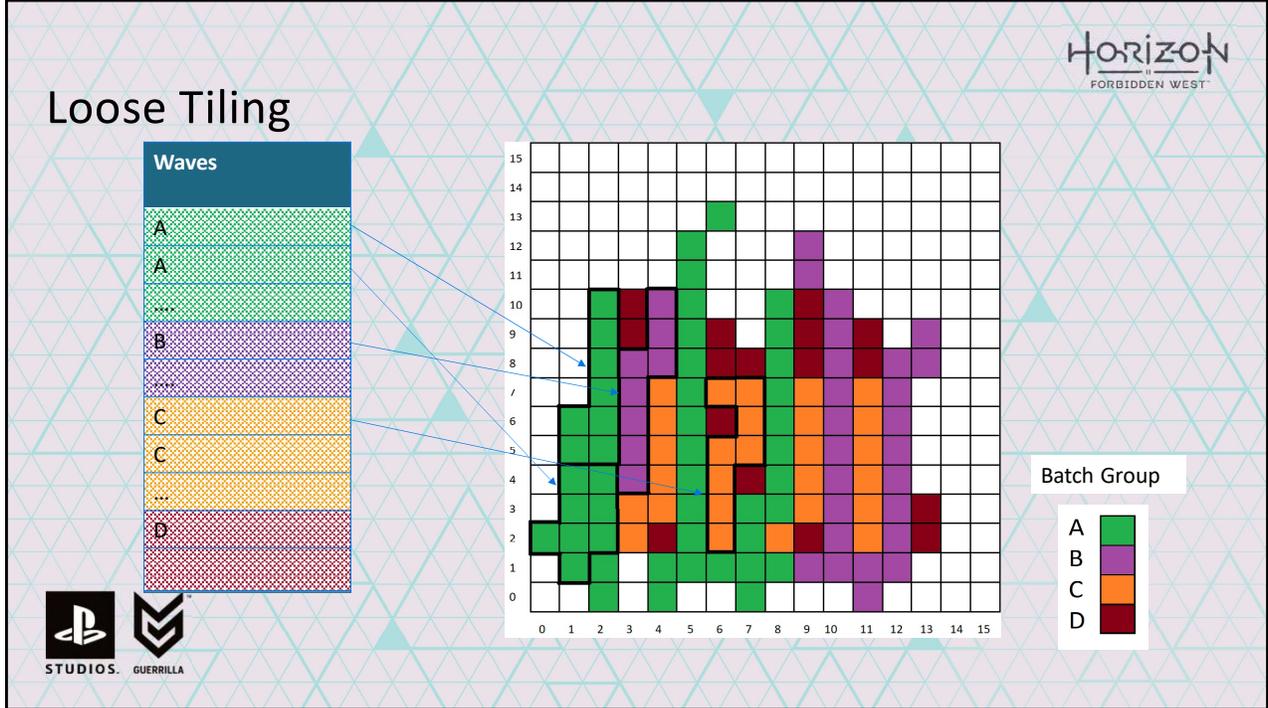
HORIZON
FORBIDDEN WEST

- ▶ Same shader can be used for many different batches. May have:
 - Different per batch constants.
 - Different textures.
 - No divergence in the wave ☺.
- ▶ Single `dispatchIndirect()` can then process all the commands for a single shader for all the tiles.
- ▶ Good chance of L2 cache hits.



32

If we do this for all the tiles on the screen then for each shader we can end up with a single `dispatchIndirect()` call that process many batches at once. And because we ensure that we the pixel commands for each tile are stored together they are likely to be processed in close temporal locality to each other which should mean that when they do memory requests to fill in the Gbuffer there is a much better chance that the memory is still in L2. Also, because we are processing complete waves with the same constants and textures, we don't have to worry about divergence. There is a slight caveat here in that we can end up with "unfilled waves" as we must round the pixel commands to fit in a particular tile. However, in practice this tends to be a very minor issue.



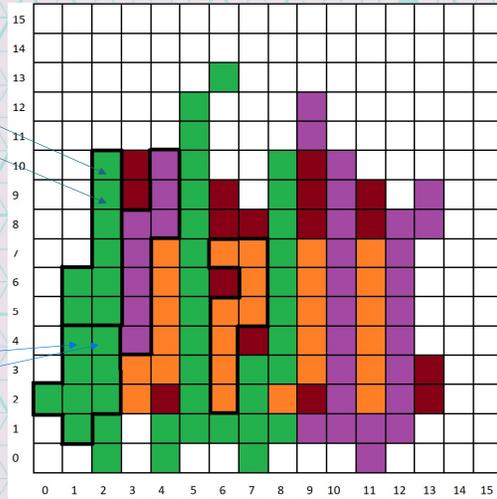
Here is a simplified example of the idea of this loose tiling in practice.
 Let's assume that this is a tile in a scene we're rendering.
 It's only 16x16 rather than 128x128 but it will do the job.
 All the coloured pixels use the same shader.
 The different colours represent pixels from what we will call different batch groups.

A batch group for our purpose is a set of batches with the same shader, textures, constants and geometry, but the batches in that group will have different sets of instances.

.....
 For the sake of this example, let's assume that we're on a GPU where the size of a wave is just 8 threads.
 What we're going to do is to group together pixels that are from the same batch group, into waves.

Loose Tiling

Pixel Commands	
2	10
2	9
2	8
2	7
1	6
2	6
1	5
2	5
1	4
2	4
...

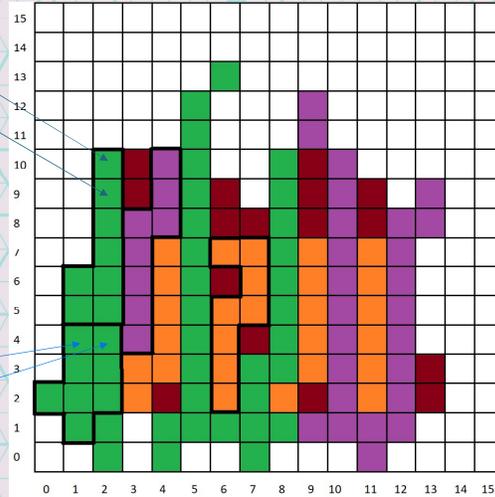


We will also build up a list of pixel commands, that describe what pixels to shade.

Loose Tiling

Wave Commands	
A	0
A	8
...	...
B	38
...	...
C	117
...	...
...	...
...	...

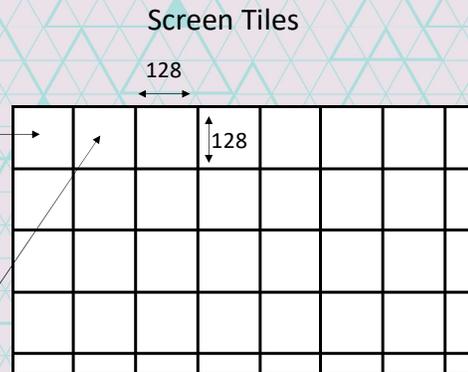
Pixel Commands	
2	10
2	9
2	8
2	7
1	6
2	6
1	5
2	5
1	4
2	4
...	...



And then produce a set of wave commands, that define what batch group a wave will work on, and will also point to it's set of pixel commands.

Loose Tiling

Batch Group	Pixel Command Start	Num Pixel Commands	Tile
A	8	8	0,0
A	16	8	0,0
...			
B	38	8	0,0
...			
C	117	5	0,0
...			
A	237	7	1,0
...			



Lets have a closer look at the wave commands

The wave commands will also record which tile they are in and the number of pixel commands the wave should process.

So, once we have produced all the wave and pixel commands for a particular shader on this tile, we can then start appending commands for the next tile.

This way for a single shader, we can build up a set of wave and pixel commands that will shade many batches, while avoiding any divergence, and attempting to preserve spatial locality for good L2 cache performance.

One nice thing about this scheme is that as we are working with 128x128 tiles, and so our buffer to hold pixel commands only needs to be 16 bit.

Also, the 128x128 tile size is carefully chosen so that if we want to we can use 12 of those 16 bits to encode pixel quads, and 4 bits to encode what pixels are set within those quads, which will be important later when we talk about variable rate shading.

Vertices with Compute

- ▶ What about transforming vertices?
- ▶ Could do this as we shade pixels?
 - Each pixel then needs to transform 3 vertices.
 - Triangle sizes still relatively large on PS4.
 - Would cause lots of redundant vertex transforms.
- ▶ Global vertex cache
 - Not big enough for all situations.



So now we know roughly how we can process pixels on compute. What about vertices?

In theory we could transform as we shade pixels, as the original Intel paper does. However especially on PS4 where we are still dealing with triangles that are relatively large this would possibly lead to a large amount of redundant work as each pixel would have to shade 3 vertices.

From Horizon Zero Dawn we had a caching system for object space vertex positions that help us avoid a lot of the redundant transformation work in the depth equals pass, and also allowed easy access to vertex positions from last frame for motion vectors. We could have tried to build on this, however it was of limited size and could overflow and we had to be able to gracefully handle situations where we would run out of space in the cache.

Vertices with Compute

- ▶ Transform vertices into a ring buffer on a separate compute pipe.
 - Break batches into passes.
 - Transform vertices for a pass.
 - Using vertex wave commands
 - Consume vertices for previous passes at the same time.
 - Limit a pass to $\frac{1}{2}$ the size of the ring buffer for sanity.
- ▶ 12mb on PS4, 24mb on PS5.



So, what we do is we transform the vertices into a ring buffer on a separate compute pipe, whilst still making use of the vertex cache.

In order to make this approach work we need to sort batches into passes that have the same shader and transform vertices for the pass in one go.

In order to drive this and pass information about what vertices to shade, we'll also need to produce vertex wave commands, that tell us what chunks of vertices we need to shade.

While we are doing this our pixels compute shaders can be consuming the vertices for previous passes.

Because we always want to be able to have things overlapped in this way, we limit the maximum number of vertices we process in a pass to $\frac{1}{2}$ the size of the ring buffer. This ring buffer will be 12mb on PS4 and 24mb on PS5.

The system in one slide

HORIZON
FORBIDDEN WEST

- ▶ List of batches sort passes per material on CPU.
- ▶ 32 bit UINT Visibility Buffer.
- ▶ GPU Classification of Visibility Buffer to feed:
 - Loose Tiling to process pixels.
 - 16 bits per pixel for pixel commands.
 - 32 bits per wave for pixel wave commands.
 - Vertices transformed on separate compute pipe into a ring buffer.
 - 32 bit vertex wave commands.
- ▶ Variable rate shading support.



39

So, lets just quickly run down how the system looks.

First off, we have some CPU work that's needed to sort batches into passes per material.

Then we're going to fill a thin 32 bit visibility buffer to encode info about the triangles we want to shade.

After which we're going to analyze the visibility buffer and do some classification on it, in order to produce pixel and pixel wave commands that will drive our shading on the pixel side

, along with vertex wave commands to drive the transformation of vertices on a separate compute pipe that will bounce off memory using a ring buffer before being consumed by our pixel work.

Finally, we'll add a little bit of variable rate shading magic, to try to squeeze even more out of things.

CPU Setup

HORIZON
FORBIDDEN WEST



40

Decima Rendering 101

HORIZON
FORBIDDEN WEST

Batch:

- Single draw call.
- Many instances with the same shader, geometry, textures and batch constants.
- Varying constants per instance.
- Output from our culling system.
- Not guaranteed to be consistent across frames.



41

Before I go any further, I want to briefly describe the inputs we have from the Decima engine.

In Decima, we query a highly optimized scene graph to determine what needs to be drawn each frame.

The scene graph perform occlusion culling for the current frame and spits out a visibility list that contains instances of visible objects packed into batches, with each batch containing 1 or more instances.

A batch from the scene graph will typically be rendered as a single draw call, so all the instances in it will share the same shaders, and geometry and textures, but some per instance parameters may vary.

The batches are not necessarily consistent from frame to frame as we move around, as we can end up with different instances being culled, and batches can also get merged internal to the scene graph query.

This may seem like simple obvious stuff, but I think it's worth me calling out here, before I get into any more of the details of what we did to avoid confusion, as we'll come back to talking about batches a little later.

Pass Sorting

- ▶ Sort batches into passes based on:
 - Shader.
 - Per batch data.
 - Quantized Morton code of their position.
- ▶ BUT: Automatic foliage placement system can create batches with hundreds of instances.
 - Can lead to individual batches that are too big for the ring buffer!



So, before we begin rendering on the GPU, we sort batches on the CPU based on a hash of their shader, per batch data and a quantized Morton code of their position and split them into passes.

We did have some issues with this approach though.

The main one being that we have a procedural foliage placement system that places most of the foliage in the world.

This can often create batches with 100s of instances.

If each of these instances uses a large number of vertices, then we are in trouble as we may not be able to fit all the vertices in our ring buffer, let alone half of it!

Micro Batches

- ▶ Limit ourselves to 16-bit indices.
- ▶ Split batches into “micro batches”.
 - Up to 64k triangles.
 - Up to 192k vertices.
 - From 1 to 64k instances.
- ▶ Multiple micro batches in a pass.
- ▶ A batch can be split over passes.
- ▶ Make a table of micro batches to look up info about them on the GPU.



To combat this, we first set some sane limit on the size of our geometry by making all our individual bits of geometry use 16 bit indices and up to 64k primitives. This stops individual bits of geometry needing too much ring buffer space. Then we have to split up the batches we get from the placement system into manageable pieces that we call micro batches. Each micro batch has up to 64k triangles and contains from 1 to 64k instances. We then make passes out of micro batches instead of batches. So if a batch is contains too many vertices to fit then it will be broken down into multiple micro batches. This means that an original batch that we are given can end up being processed over multiple passes.

Micro Batches

- ▶ Limit ourselves to 16-bit indices.
- ▶ Split batches into “micro batches”.
 - Up to 64k triangles.
 - Up to 192k vertices.
 - From 1 to 64k instances.
- ▶ Multiple micro batches in a pass.
- ▶ A batch can be split over passes.
- ▶ Make a table of micro batches to look up info about them on the GPU.



For each micro batch that we produce we fill in an entry in what we call the MicroBatchInfoTable Table.

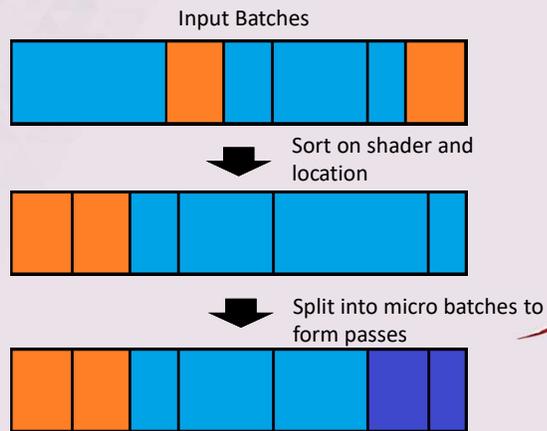
This contains info about the micro batch such as how many verts it's geometry uses, which instance of the batch it starts on and which pass it belongs to.

This is placed in a GPU accessible buffer and can then be used during shading to recover information about the micro batch we're shading.

<Perhaps go take a look at Bonus slide 108 before continuing...>

Vertex/Batch Flow

HORIZON
FORBIDDEN WEST



45

So here you can see the rough flow of the batches.
They get sorted on shader,
And then split into micro batches to form passes.

Vertex/Batch Flow

Micro batches

Vertex Ring Buffer

Transformed by vertex compute shader dispatches

Consumed by pixel compute shader dispatches

Wrap around

Pass 0

Pass 1

Pass 2



The micro batches will then be transformed by vertex compute shader into a ring buffer
And those transformed vertices will be consumed by pixel compute shader associated with each pass.



Ok, so now you hopefully have a high level view of our setup work on the CPU. Let me dive into how we lay down our Visibility Buffer.

Visibility Buffer Laydown

- ▶ DepthAndVisibility pass at the start of the frame.
- ▶ For each pixel in this pass, we write a single 32 bit UINT encoding:
 - Triangle.
 - Instance.
 - Batch.
- ▶ Could use dedicated HW on PS4 Pro and PS5 to accelerate
 - No support on base ☹️



Decima already had a depth prepass, so we augmented this with an additional depth and visibility pass that wrote to the Visibility Buffer as well as the Depth Buffer. For each pixel this pass needs to write out information about the triangle, instance and batch as a 32 bit primitive ID. In theory we could use the dedicated hardware that's in PS4 Pro and PS5 for this, but unfortunately this would leave PS4 Base out in the cold, as it doesn't have a proper way to get a primitive Id without using a geometry shader.

PRIMITIVE ID WITH XOR

HORIZON
FORBIDDEN WEST

- ▶ Initial plan was to encode primitive ID in provoking vertex index.
 - Issues with HW rotating vertices.
- ▶ Pass data to the PS with data extracted from 3 vertex indices.
- ▶ Combine that data with XOR.
- ▶ $\text{PrimitiveID} = \text{Data0} \wedge \text{Data1} \wedge \text{Data2}$
- ▶ Same number of additional indices as provoking vertex.



49

Initial plan was to encode the primitive ID in the top 16 bits of our triangle indices and just read the provoking vertex in the pixel shader to pass through the primitive ID from the vertex shader, but early in development we had issues with the hardware rotating the vertices.

I believe there is now a workaround for this, however, we ended up using a solution that avoided this issue entirely.

If we use XOR we can uniquely reconstruct a primitive ID from 3 values that we encode in each of the 3 vertices.

With this scheme we don't care about the order of the vertices, just the combination of their payloads.

*This also requires us to introduce additional indices, but no more than we would have had to with the provoking vertex scheme.

So now when we preprocess our mesh, we determine what data to store in the provoking vertices index based on the primitive ID we want to encode and the data we already have encoded in the other indices.

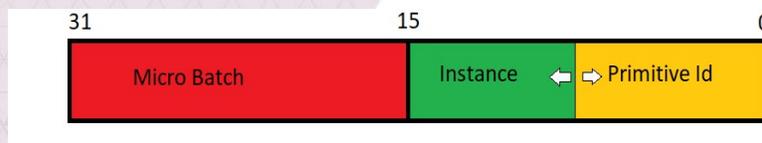
If all indices already have their data determined, then we must add a new one.

Visibility Buffer Format

► Everything packed into 32 bits in a flexible format:

- Micro Batch : 16 bits
- PrimitiveID : $\text{Ceil}(\text{Log}_2(\text{num primitives in geometry}))$ bits
 - Only support 64k prims per geo.
- InstanceID : remaining bits.

► Running out of instance bits wraps to the next Micro Batch.

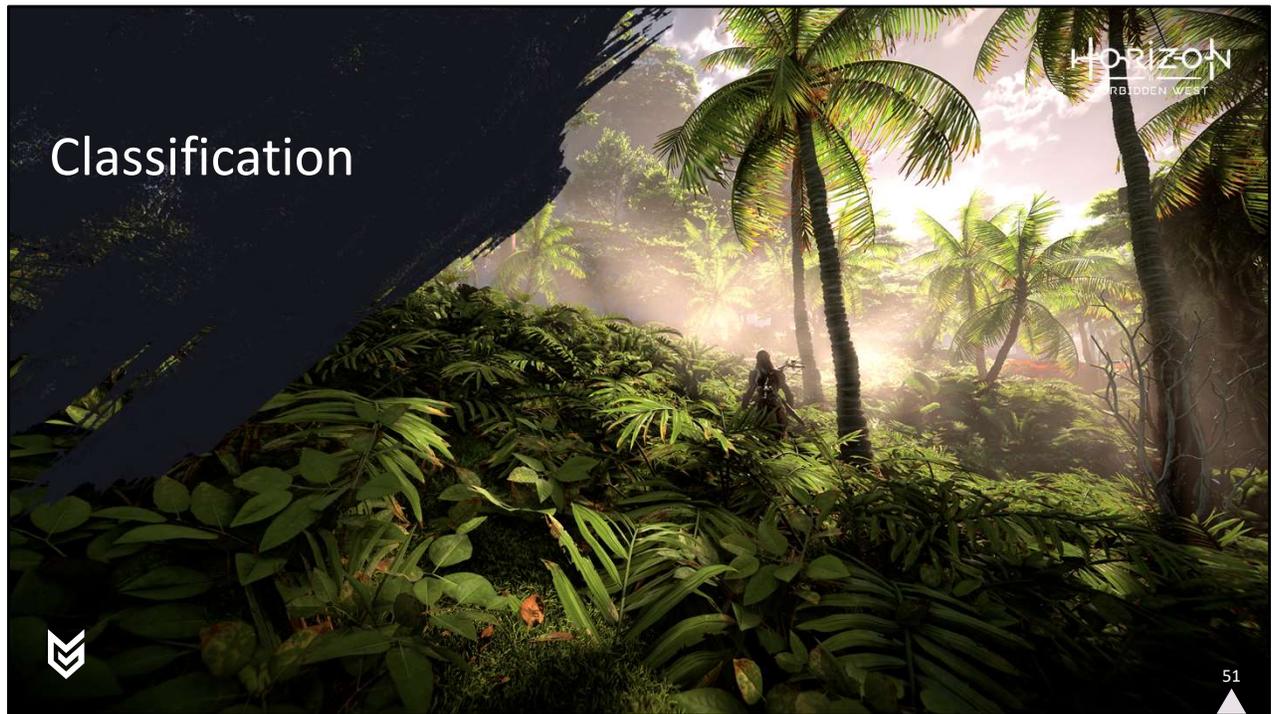


In the pixel shader we can then combine the primitive Id we've reconstructed with the Micro Batch ID and the Instance ID to produce our visibility buffer.

The top 16 bits encode the micro batch ID.

We use a variable number of bits to encode the Primitive ID depending how many primitives are in the geometry we are rendering.

The remaining bits are used for the instance ID.



OK, so now we've managed to lay down our Visibility Buffer.
Next, we're going to need to do some classification on that to figure out what
vertices and pixels we need to shade.

Building the Data for the Passes

- ▶ Now we have a visibility buffer.
- ▶ How can we use that to dispatch the compute shaders for our passes?
- ▶ Need to classify.
- ▶ Split into three phases.
 - Laydown finalization
 - Mid classify
 - Classify output

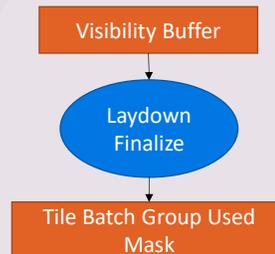


We split our classification into to 3 phases.
Laydown finalization, mid classify and classify output

Visibility Buffer Classification

Laydown finalization

- ▶ Writes out masks of which batch groups are used per tile per pass.
- ▶ Not all geometry writes visibility.
- ▶ Stencil encodes valid visibility.



First, we have what we call laydown finalization

This reads the visibility buffer and then writes out a mask of which batch groups are used in each tile per pass

One thing to note here is that not everything writes into the Visibility Buffer.

We have a geometry pass that runs afterwards that fills our Gbuffers with regular deferred geometry.

We don't want to have to add another export to write the Visibility Buffer to all these shaders.

So, we make use of a bit in the stencil buffer to indicate if the contents of the Visibility Buffer are valid.

If we write this stencil bit when we are laying down the Visibility Buffer then it incurs a none insignificant cost, due to the amount of overdraw.

So instead, we write this via a UAV when we run our laydown finalize shader.

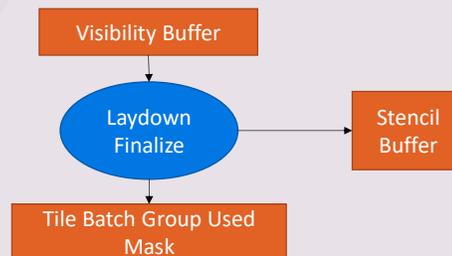
Batches in the geometry pass then overwrite this stencil bit.

Once the geometry pass is done, only pixels with the stencil bit set contain valid visibility buffer information.

Visibility Buffer Classification

Laydown finalization

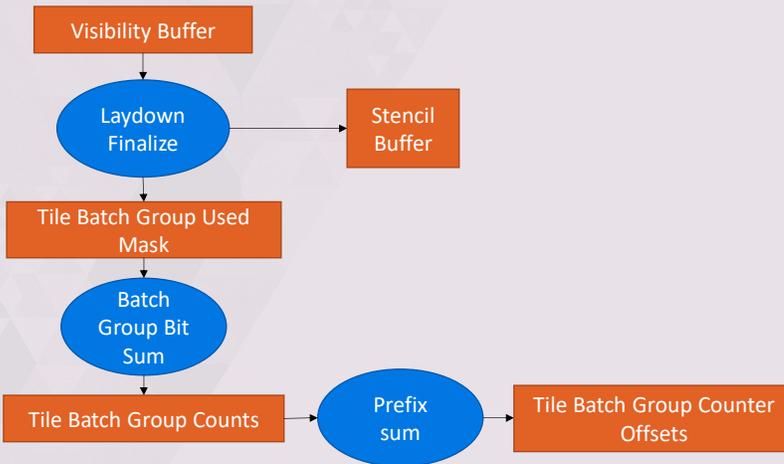
- ▶ Writes out masks of which batch groups are used per tile per pass.
- ▶ Not all geometry writes visibility.
- ▶ Stencil encodes valid visibility.
- ▶ Directly write the stencil buffer via a UAV to indicate valid Visibility buffer pixels in Laydown Finalization.
- ▶ Geometry pass doesn't write to the Visibility Buffer, just the stencil.



So instead, we write this via a UAV when we run our laydown finalize shader. Batches in the geometry pass then overwrite this stencil bit. Once the geometry pass is done, only pixels with the stencil bit set contain valid visibility buffer information.

Visibility Buffer Classification

Mid Classify



<Note, it may be worth reading bonus slide 108 before this slide to understand the limit of 32 batch groups per pass>

We then have the Mid classify phase.

This can run alongside the geometry pass.

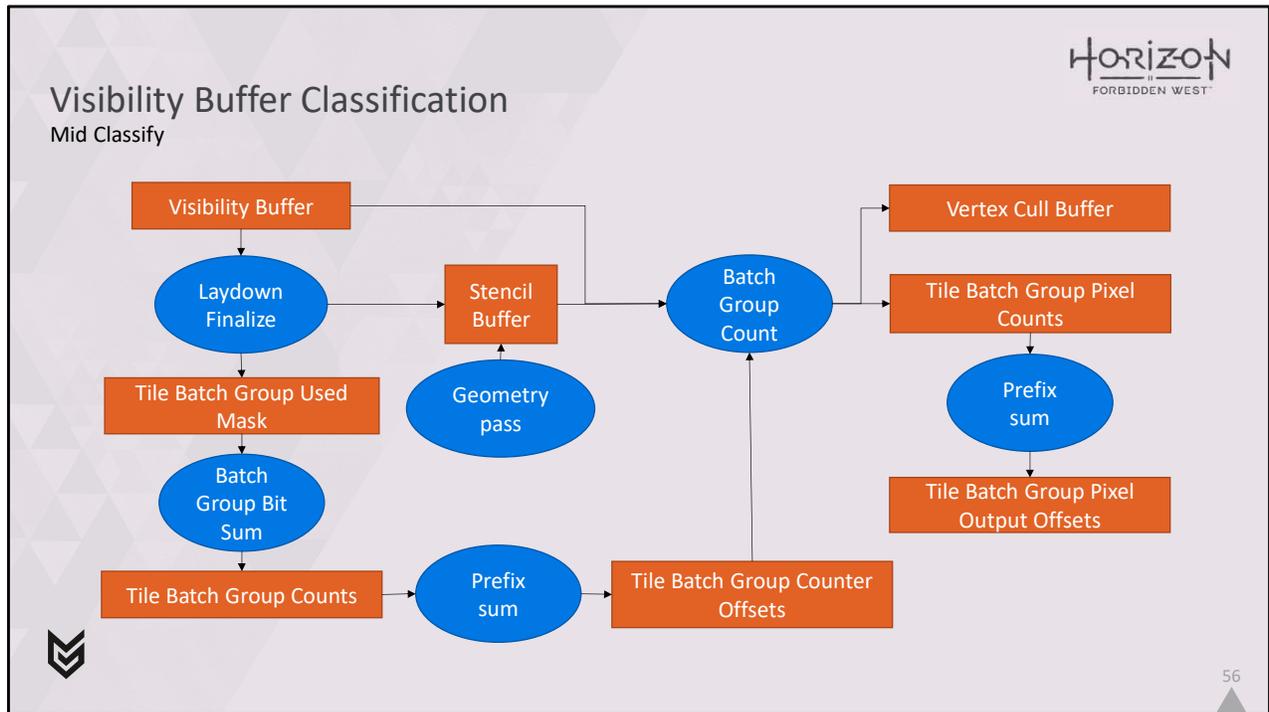
We already have our batch group used mask and have filled the stencil buffer in the laydown finalization.

Next we take the used mask and use popcount to sum the bits to get a count of how many batch groups are used for each tile in each pass.

And we do a prefix sum on that to get the offsets of our counters.

This is all done so that we don't have to have a full set of 32 batch group counters for each tile for each pass, as to do so with 256 passes at 4K (which is the maximum we currently support) would need us to have 16mb of space for our counters.

However, as most batch groups are not used in all tiles, we can get away with considerably less than that by calculating the set of counters that will actually be used and storing offsets to them in a global counter buffer.



While all of this is happening the geometry pass is still running, and will be updating the stencil buffer.

We then read the stencil buffer, the visibility buffer and the tile batch group counter offsets and count up how many pixels we have in each batch group in each tile in each pass.

At the same time we also use the primitive ID to decode which vertices are used and record that in the vertex cull buffer.

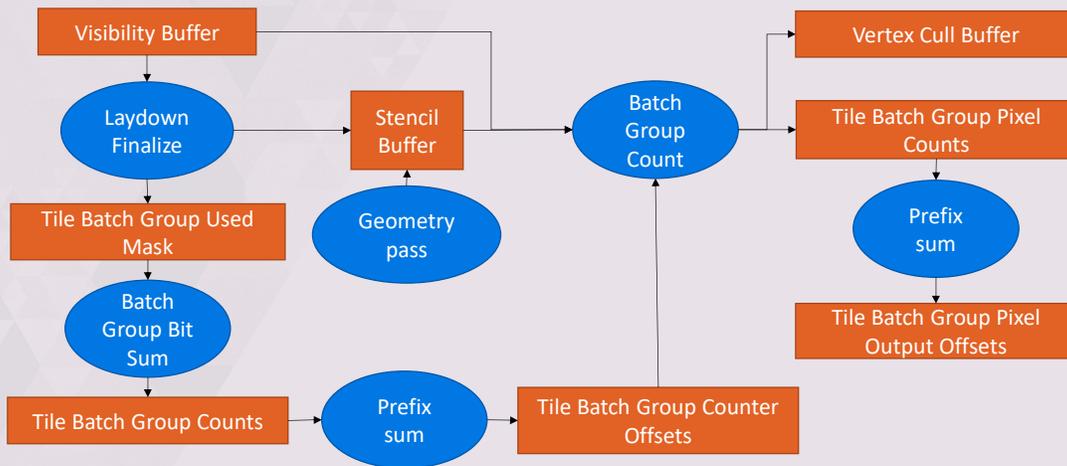
We only currently do this at a granularity of 64 vertex chunks, to keep things simple and to save space, as we can have 10's of millions of verts in a scene in some circumstances.

Then we take the tile batch group pixel counts, perform a prefix sum on them, and spit out a set of offsets for where we will start recording our pixel command for each batch group per tile per pass.

It's important to remember that this can all be running while the geometry pass is still in flight, so we can over estimate how many pixel commands each batch group will need, but this isn't a problem as we will only ever reduce the number of valid visibility pixels the further into the geometry pass we get.

Visibility Buffer Classification

Mid Classify



It's important to remember that this can all be running while the geometry pass is still in flight, so we can over estimate how many pixel commands each batch group will need, but this isn't a problem as we will only ever reduce the number of valid visibility pixels the further into the geometry pass we get.

Visibility Buffer Classification

Mid classify

HORIZON
FORBIDDEN WEST

- ▶ Identifies visible Vertex Chunks.
 - Groups of 64 verts in our ring buffer that we would need to transform.
 - Output Vertex Wave commands (32 bits)
 - Micro batch & vertex chunk.



58

Now during this mid classify step we also need to deal with producing vertex wave commands that will tell us what vertex work we need to do in each pass. The main input to this calculation is the vertex cull buffer that we previously produced.

This has a byte set per vertex chunk to indicate if it is visible or not.

I'm not going to cover how the vertex commands are created in this presentation for time reasons, but if you're interested you can find some slides about it in the bonus slides at the end of this presentation.

Visibility Buffer Classification

Classification output

► Uses final Visibility Buffer after Geometry pass.

- Stencil buffer identifies valid pixels.

► Outputs

- Pixel Wave Commands (64 bits)
 - Tile Coords, Batch Group Id, Pixel command offset, Number of pixel commands.
- Pixel Commands (16 bits)
 - Pixel X,Y in the tile.



HORIZON
FORBIDDEN WEST



59

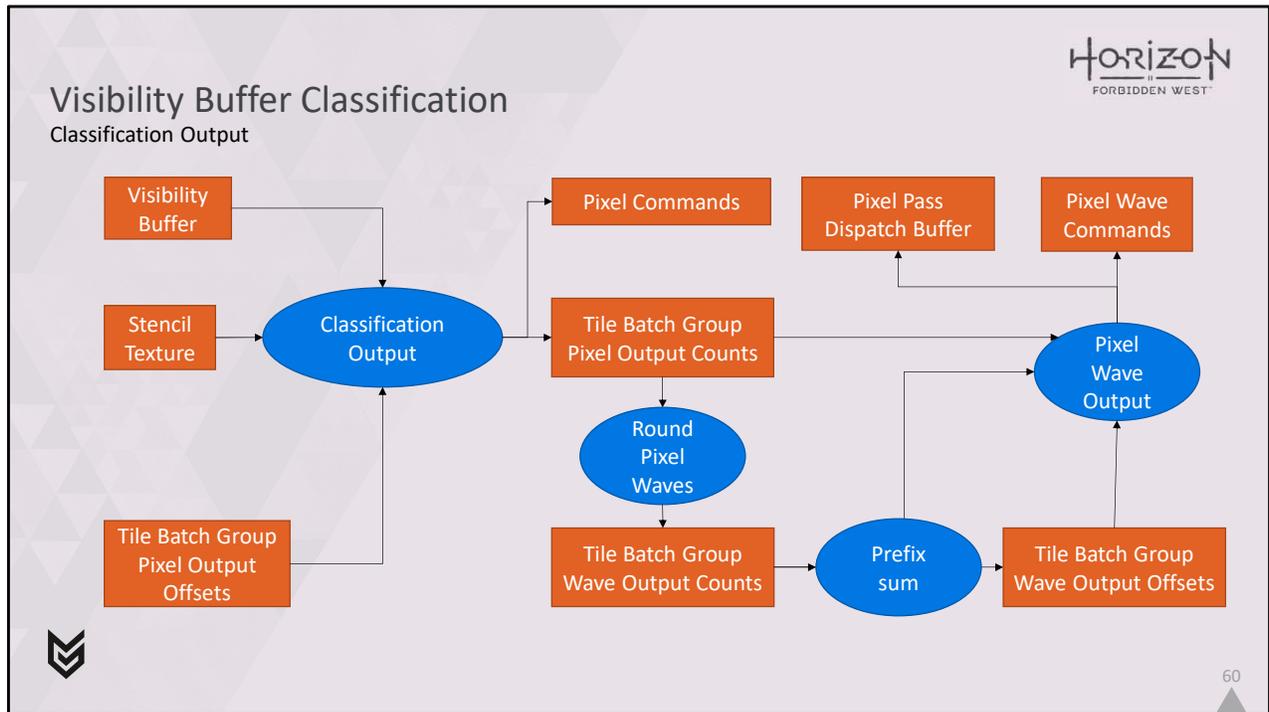
So, with the mid classify done we have the classification output phase that runs on the Visibility Buffer after the Geometry pass is finished.

The stencil buffer which gets updated in the geometry pass is used to tell us which entries in the Visibility Buffer are actually valid.

We can then use this to write out our various commands.

These are the pixel commands that contain the Pixel X and Y within in the tile.

And also, we write the pixel wave commands, that encode the tile coordinates, the batch group ID, and the offset to and number of the pixel commands.



So lets take a look at this step in more detail.

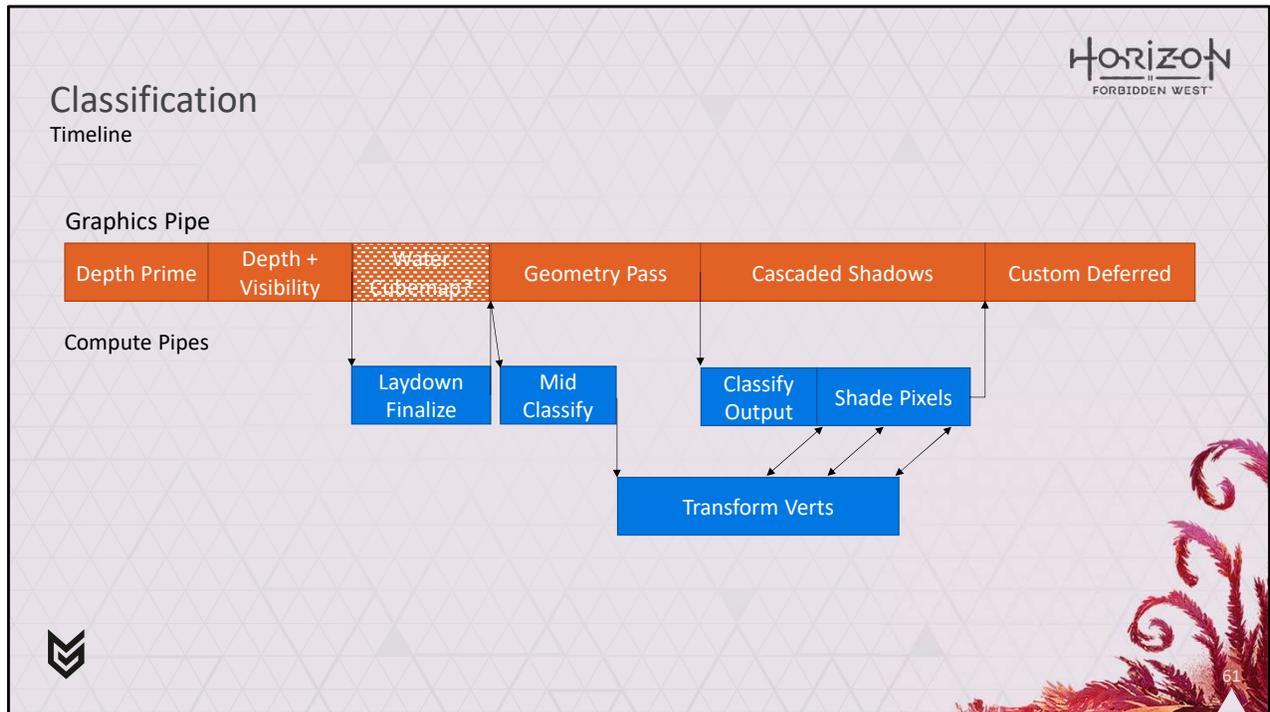
So here you can see we take the visibility buffer, stencil buffer and the tile batch group pixel output offsets we calculated previously and process each pixel with the classification output shader.

This will output our pixel commands, along with a the final counts of how many pixels are used in each batch group, for each tile for each pass.

We can then round these counts up to wave alignment and use them to calculate then number of waves we will need for each batch group per tile per pass.

We can then perform a prefix sum to figure out where we need to output our pixel wave commands.

Finally we can do a dispatch over all the passes and tiles and output both the wave commands and the dispatch buffer to use for the pixel passes.



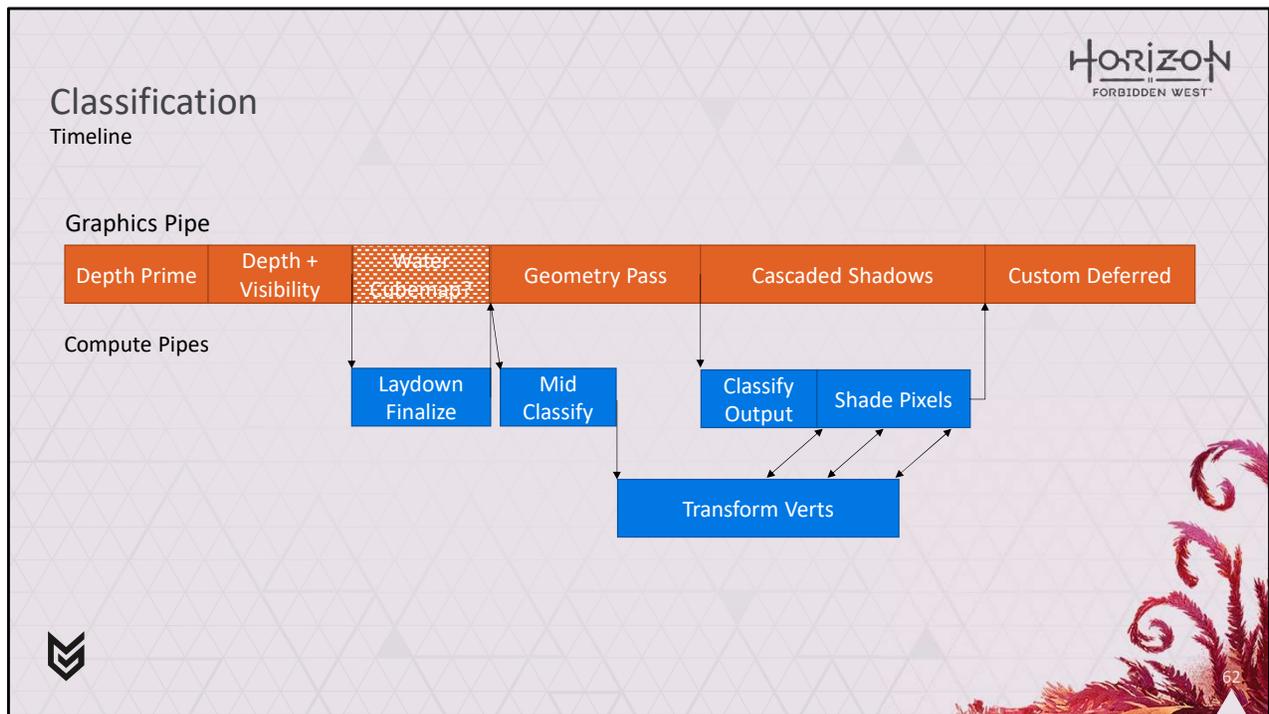
Here's a look at how this all fits in our GPU timeline.

As you can see, we start our laydown finalization right after the Depth and Visibility pass.

If we're lucky then this can run in parallel with rendering a face of our water cube map.

After that the geometry pass starts and we can begin the mid classification steps.

These get us ready for the final classify output step, but also cull vertices and output vertex wave commands.



Because the mid classification happens while the geometry pass is running on the graphics pipe, some of the results we get are conservative.

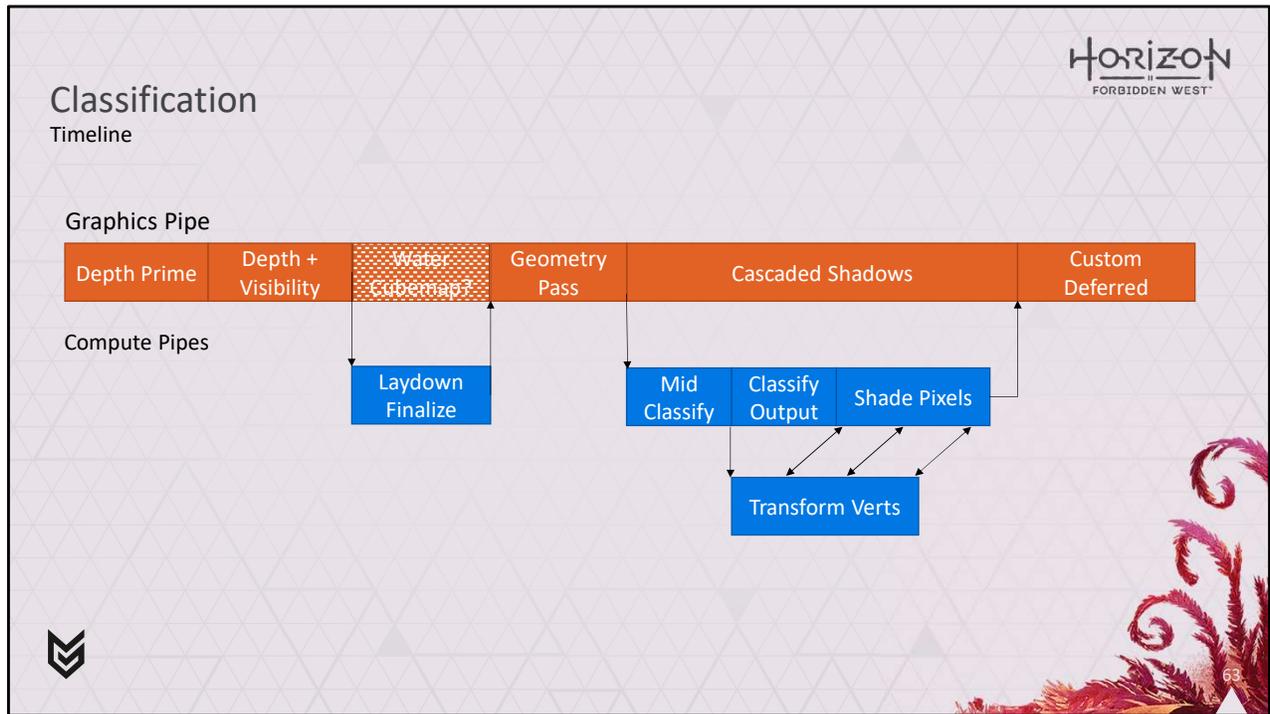
So we might end up transforming some vertices that may be occluded.

However as most of our major occluders are in the depth prime pass we generally find that it is better to run this in parallel with the geometry pass than to wait for accurate results after it.

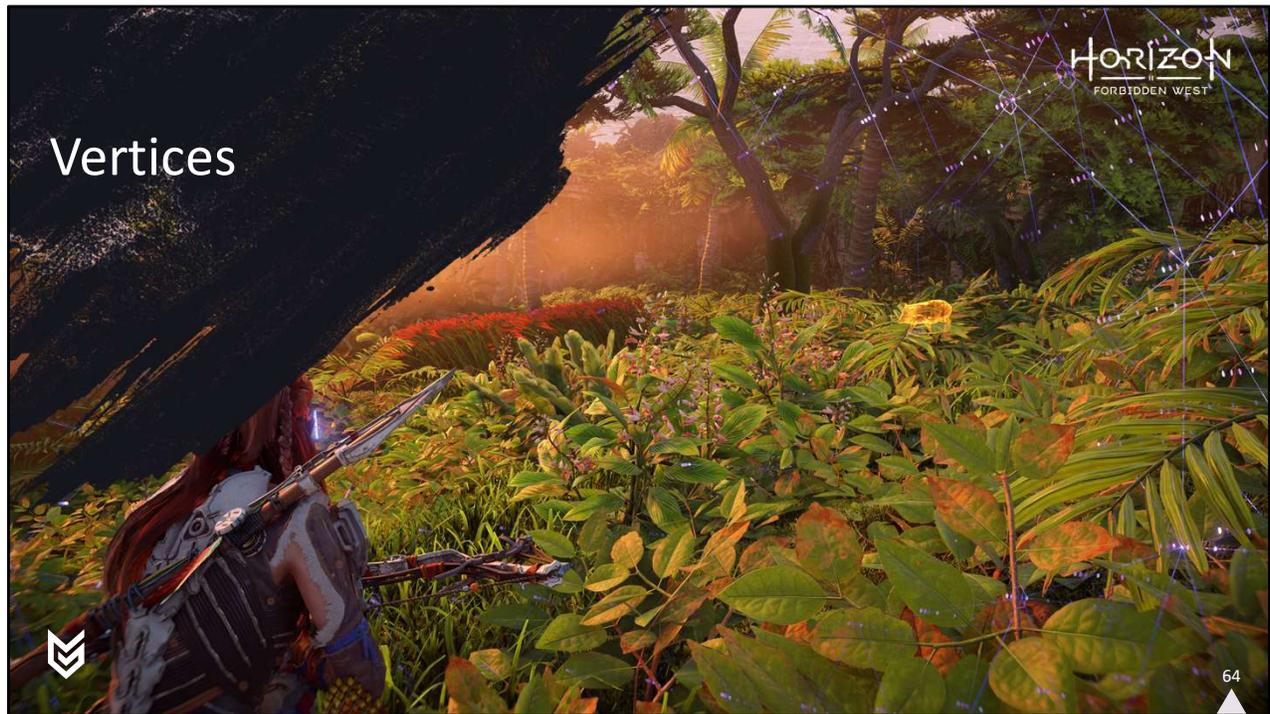
It also allows us to start transforming vertices for our deferred texturing passes before the geometry pass has finished.

Once the geometry pass is done, we do our final classification, output our pixel and wave commands and start shading pixels while we render shadows.

We then finish up our Gbuffer laydown with our custom deferred pass, which is used for decals and the like that can modify already written Gbuffer values.



It's also possible to run the system in an alternate mode where the mid classify is moved to the end of the geometry pass. This trades less work in parallel with the geometry pass against higher vertex culling rates, and we're experimenting with this on PS5 where the geometric density is that much higher.



So a few words about how the vertices in our ring buffer is encoded

Vertex Format

Member	Format	Optional
HPOS (xyw)	Half2 + float	No
UV	16:16 scaled down by 8 to allow wrapping	Yes
Vertex Color	RGBA8888	Yes
Normal	10:10:10:2	No
Tangent	10:10:10:2	No
Previous HPOS (xyw)	Half3	No
Unused	16 bits (Available for additional interpolants)	N/A



This is the format what we use. It's 32 bytes long in total.

For HPOS we encode x & y as half floats but found that for accuracy reasons we needed to keep w as a float, and z doesn't need to be sent as it can be easily recovered from w.

UVs we store as fixed point 16:16 but scaled down by 8 to allow support for some UV wrapping.

We also have space for vertex color, normal, tangent, and the previous HPOS so that we can construct motion vectors, and you'll see we also have 16 bits unused.

Extra Interpolants

HORIZON
FORBIDDEN WEST

- ▶ Some shaders need extra interpolants.
 - Allow UV and Vertex Color to be read separately.
 - Reuse their space + unused 16 bits.
 - Can support up to 5 x 16-bit interpolant slots.
 - Half or 2 channel sqrt 8 bit UNorm for colors.



66

So, this isn't a fully fixed format, for some shaders we need extra interpolants, especially if we want to try to move some pixel work to the vertex shader for optimization purposes.

We don't animate UV and vertex color, however it's usually more efficient for them to be read in the vertex program and placed in the vertex format to help reduce the number of buffer reads per pixel.

To account for extra interpolants however, we allow the UV and vertex color to be kicked out of the fix format and their data buffers to be read directly. We then reuse their space for our additional interpolants.

Using this scheme, we can support up to 5x16 bit extra interpolant slots.

At the moment, a slot can be filled with either a half float, or a 2-channel square rooted 8 bit unorm for color values, which you can think of as like a poor man's sRGB.

OK, so, that's all the information I'm going to present on the core of the system.

Hopefully you now have some inkling of how this all works,

I'm aware that I've gone through some of this pretty fast and introduced a bunch of new concepts

so if you're thinking man that was a lot, then trust me.



.... I know! I can't keep it all in my head half the time either..



So, lets just chill out with Aloy the forest for a few seconds and appreciate what it's all for
.... Ok everyone got their breath?



So now I'm going to talk a little bit about the variable rate shading support that we added towards the end of the project to try to squeeze just a little bit more performance out of the system.

Variable Rate Shading



- ▶ Not all HW has native support for VRS.
- ▶ We are doing manual pixel export.
- ▶ Can do it ourselves!
- ▶ Top 2 bits of the visibility buffer used for shading rate.
- ▶ Driven by the vertex shader in the DepthAndVisibility pass.
- ▶ 1x1, 1x2, 2x1 and 2x2 rates supported.
- ▶ Gains ~ 0.2 – 0.6ms on PS4 Base



What I've described so far can be pretty efficient, but we wanted to see if we could get things even faster.

A lot of the foliage we render ends up being various shades of green, and so we don't always benefit from shading at full rate.

Because we are effectively managing pixel export ourselves rather than going via the ROPs we were able to modify our scheme to

support variable rate shading even on hardware that doesn't natively support it.

We steal the top 2 bits of the visibility buffer from the micro batch offset, and instead use it to encode the shading rate.

In theory we could support driving this via a screen space shading rate texture, but for the moment we have chosen to just drive this from the vertex shader.

We support all the Direct X Tier 1 VRS shading rates.

We only enable this on PS4 Base, and the gains are heavily scene dependent.



Here we can see a nice forest scene



And this is our standard setting for VRS when you are stationary, which blends through 1x1 and 2x1 to 2x2 depending on distance. On this scene when the deferred texturing is not overlapped this gets us about 0.2ms back



However, we use the screen space speed of vertices in the vertex shader to drive the shading rate

So, when we are moving at pace, everything blends towards using 2x2, which in this scene gets us almost 0.5ms, and so is much more worth our time.

Variable Rate Shading

Classification

- ▶ Classification Output reads this and uses QuadSwizzle() (yes quads ☺!)
 - Identifies pixels that can be shaded together.
- ▶ Pixel commands modified so that 6 + 6 bits identify a quad (64x64)
 - 4 bits identify pixels within that quad to broadcast to.

HORIZON
FORBIDDEN WEST



74

Our classification output stage can read this shading rate info and use QuadSwizzle to quickly determine pixels that can be shaded together.

As I alluded to earlier, we also change our pixel commands so that 12 bits identify a particular quad to shade in a 128x128 tile.

The top 4 bits of the command are then used to identify which pixels in the quad we need to broadcast to.

Variable Rate Shading

Output Broadcast

- ▶ Calculated sample results must be broadcast.
 - Output order is important.
 - Single thread should not output adjacent pixels.
 - Bad for bandwidth.
 - Expand work within the wave into LDS.
 - Loop over expanded work in the wave to output.



HORIZON
FORBIDDEN WEST



75

In the generated shader that does the output for a material, we will then need to broadcast the results for any variable rate samples in a wave.

It's possible to naively do this by just making each thread loop over the samples that it should broadcast to, but this is not very memory friendly and gives less than stellar performance.

What we need to do instead is some work expansion within each wave.

We take all the samples and build a list of expanded pixel commands in LDS.

This is done at the start of the shader, if any of the pixel commands in the wave need to broadcast.

At the end of the shader for each UAV in our G-Buffer we will then use this expanded list of work to do the broadcast.

```

thread_group_memory uint sIntermediateSpace[64]; // Space to store the pre converted output per lane
thread_group_memory unsigned char sIntermediatePixelX[64]; // X Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sIntermediatePixelY[64]; // Y Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sPixelCommands[256]; // Packed lane and offset from original position

void BroadcastOutput0(uint inNumPixels, uint inGroupIndex, uint2 inScreenTileCoords, float4 inColor)
{
    uint4 tlo = __get_tsharplo(SRT_DEFERREDTEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);
    uint4 thi = __get_tsharpri(SRT_DEFERREDTEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);

    sIntermediateSpace[inGroupIndex] = packUnorm4x8(inColor); // Cache manually converted results into LDS

    uint num_active_lanes = gWaveActiveLaneCount();

    for(int i=inGroupIndex; i< inNumPixels; i+=num_active_lanes) // Loop over the all the broadcast work
    {
        uint cmd = sPixelCommands[i]; // Read the work expanded command
        uint pixel_lane = BitFieldExtract(cmd,0,6); // What lane calculated this result
        uint pixel_offset_x = BitFieldExtract(cmd,6,1); // Where should it be offset from its original position
        uint pixel_offset_y = BitFieldExtract(cmd,7,1);

        uint col_out = sIntermediateSpace[pixel_lane]; // Read the manually converted result we stashed
        uint2 out_pixel = uint2(sIntermediatePixelX[pixel_lane], sIntermediatePixelY[pixel_lane]) + // Get its origin
            uint2(pixel_offset_x, pixel_offset_y) + // Add the 1 bit offset in x&y
            inScreenTileCoords; // Place in the right tile
        __image_store_pck(col_out, uint4(out_pixel,0,0), tlo, thi, __kImage_texture2d); // Output!
    }
}

```



76

The code to broadcast the output to a single UAV looks something like this.

You can see that we start <CLICK> by manually converting our shaded result into what will be its final raw form in memory and caching the results in LDS.

<CLICK>We can then loop through the expanded set of commands, <CLICK>decode them and <CLICK> grab the result we stashed in LDS related to each of them.

Finally, <CLICK>, each lane can then figure out exactly where it's output should go and <CLICK> output it via a packed image store intrinsic.

```

thread_group_memory uint sIntermediateSpace[64]; // Space to store the pre converted output per lane
thread_group_memory unsigned char sIntermediatePixelX[64]; // X Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sIntermediatePixelY[64]; // Y Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sPixelCommands[256]; // Packed lane and offset from original position

void BroadcastOutput0(uint inNumPixels, uint inGroupIndex, uint2 inScreenTileCoords, float4 inColor)
{
    uint4 tlo = __get_tsharplo(SRT_DEFERREDEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);
    uint4 thi = __get_tsharpri(SRT_DEFERREDEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);

    sIntermediateSpace[inGroupIndex] = packUnorm4x8(inColor); // Cache manually converted results into LDS

    uint num_active_lanes = gWaveActiveLaneCount();

    for(int i=inGroupIndex; i< inNumPixels; i+=num_active_lanes) // Loop over the all the broadcast work
    {
        uint cmd = sPixelCommands[i]; // Read the work expanded command
        uint pixel_lane = BitFieldExtract(cmd,0,6); // What lane calculated this result
        uint pixel_offset_x = BitFieldExtract(cmd,6,1); // Where should it be offset from its original position
        uint pixel_offset_y = BitFieldExtract(cmd,7,1);

        uint col_out = sIntermediateSpace[pixel_lane]; // Read the manually converted result we stashed
        uint2 out_pixel = uint2(sIntermediatePixelX[pixel_lane], sIntermediatePixelY[pixel_lane]) + // Get its origin
            uint2(pixel_offset_x, pixel_offset_y) + // Add the 1 bit offset in x&y
            inScreenTileCoords; // Place in the right tile
        __image_store_pck(col_out, uint4(out_pixel,0,0), tlo, thi, __kImage_texture2d); // Output!
    }
}

```



The code to broadcast the output to a single UAV looks something like this.

You can see that we start <CLICK> by manually converting our shaded result into what will be its final raw form in memory and caching the results in LDS.

<CLICK>We can then loop through the expanded set of commands, <CLICK>decode them and <CLICK> grab the result we stashed in LDS related to each of them.

Finally, <CLICK>, each lane can then figure out exactly where it's output should go and <CLICK> output it via a packed image store intrinsic.

```

thread_group_memory uint sIntermediateSpace[64]; // Space to store the pre converted output per lane
thread_group_memory unsigned char sIntermediatePixelX[64]; // X Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sIntermediatePixelY[64]; // Y Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sPixelCommands[256]; // Packed lane and offset from original position

void BroadcastOutput0(uint inNumPixels, uint inGroupIndex, uint2 inScreenTileCoords, float4 inColor)
{
    uint4 tlo = __get_tsharplo(SRT_DEFERREDTEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);
    uint4 thi = __get_tsharpri(SRT_DEFERREDTEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);

    sIntermediateSpace[inGroupIndex] = packUnorm4x8(inColor); // Cache manually converted results into LDS

    uint num_active_lanes = gWaveActiveLaneCount();

    for(int i=inGroupIndex; i< inNumPixels; i+=num_active_lanes) // Loop over the all the broadcast work
    {
        uint cmd = sPixelCommands[i]; // Read the work expanded command
        uint pixel_lane = BitFieldExtract(cmd,0,6); // What lane calculated this result
        uint pixel_offset_x = BitFieldExtract(cmd,6,1); // Where should it be offset from its original position
        uint pixel_offset_y = BitFieldExtract(cmd,7,1);

        uint col_out = sIntermediateSpace[pixel_lane]; // Read the manually converted result we stashed
        uint2 out_pixel = uint2(sIntermediatePixelX[pixel_lane], sIntermediatePixelY[pixel_lane]) + // Get its origin
            uint2(pixel_offset_x, pixel_offset_y) + // Add the 1 bit offset in x&y
            inScreenTileCoords; // Place in the right tile
        __image_store_pck(col_out, uint4(out_pixel,0,0), tlo, thi, __kImage_texture2d); // Output!
    }
}

```



78

The code to broadcast the output to a single UAV looks something like this.

You can see that we start <CLICK> by manually converting our shaded result into what will be its final raw form in memory and caching the results in LDS.

<CLICK>We can then loop through the expanded set of commands, <CLICK>decode them and <CLICK> grab the result we stashed in LDS related to each of them.

Finally, <CLICK>, each lane can then figure out exactly where it's output should go and <CLICK> output it via a packed image store intrinsic.

```

thread_group_memory uint sIntermediateSpace[64]; // Space to store the pre converted output per lane
thread_group_memory unsigned char sIntermediatePixelX[64]; // X Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sIntermediatePixelY[64]; // Y Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sPixelCommands[256]; // Packed lane and offset from original position

void BroadcastOutput0(uint inNumPixels, uint inGroupIndex, uint2 inScreenTileCoords, float4 inColor)
{
    uint4 tlo = __get_tsharplo(SRT_DEFERREDEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);
    uint4 thi = __get_tsharpri(SRT_DEFERREDEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);

    sIntermediateSpace[inGroupIndex] = packUnorm4x8(inColor); // Cache manually converted results into LDS

    uint num_active_lanes = gWaveActiveLaneCount();

    for(int i=inGroupIndex; i< inNumPixels; i+=num_active_lanes) // Loop over the all the broadcast work
    {
        uint cmd = sPixelCommands[i]; // Read the work expanded command
        uint pixel_lane = BitFieldExtract(cmd,0,6); // What lane calculated this result
        uint pixel_offset_x = BitFieldExtract(cmd,6,1); // Where should it be offset from its original position
        uint pixel_offset_y = BitFieldExtract(cmd,7,1);

        uint col_out = sIntermediateSpace[pixel_lane]; // Read the manually converted result we stashed
        uint2 out_pixel = uint2(sIntermediatePixelX[pixel_lane], sIntermediatePixelY[pixel_lane]) + // Get its origin
            uint2(pixel_offset_x, pixel_offset_y) + // Add the 1 bit offset in x&y
            inScreenTileCoords; // Place in the right tile
        __image_store_pck(col_out, uint4(out_pixel,0,0), tlo, thi, __kImage_texture2d); // Output!
    }
}

```



79

The code to broadcast the output to a single UAV looks something like this.

You can see that we start <CLICK> by manually converting our shaded result into what will be its final raw form in memory and caching the results in LDS.

<CLICK>We can then loop through the expanded set of commands, <CLICK>decode them and <CLICK> grab the result we stashed in LDS related to each of them.

Finally, <CLICK>, each lane can then figure out exactly where it's output should go and <CLICK> output it via a packed image store intrinsic.

```

thread_group_memory uint sIntermediateSpace[64]; // Space to store the pre converted output per lane
thread_group_memory unsigned char sIntermediatePixelX[64]; // X Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sIntermediatePixelY[64]; // Y Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sPixelCommands[256]; // Packed lane and offset from original position

void BroadcastOutput0(uint inNumPixels, uint inGroupIndex, uint2 inScreenTileCoords, float4 inColor)
{
    uint4 tlo = __get_tsharplo(SRT_DEFERREDTEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);
    uint4 thi = __get_tsharpri(SRT_DEFERREDTEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);

    sIntermediateSpace[inGroupIndex] = packUnorm4x8(inColor); // Cache manually converted results into LDS

    uint num_active_lanes = gWaveActiveLaneCount();

    for(int i=inGroupIndex; i< inNumPixels; i+=num_active_lanes) // Loop over the all the broadcast work
    {
        uint cmd = sPixelCommands[i]; // Read the work expanded command
        uint pixel_lane = BitFieldExtract(cmd,0,6); // What lane calculated this result
        uint pixel_offset_x = BitFieldExtract(cmd,6,1); // Where should it be offset from its original position
        uint pixel_offset_y = BitFieldExtract(cmd,7,1);

        uint col_out = sIntermediateSpace[pixel_lane]; // Read the manually converted result we stashed
        uint2 out_pixel = uint2(sIntermediatePixelX[pixel_lane], sIntermediatePixelY[pixel_lane]) + // Get its origin
            uint2(pixel_offset_x, pixel_offset_y) + // Add the 1 bit offset in x&y
            inScreenTileCoords; // Place in the right tile
        __image_store_pck(col_out, uint4(out_pixel,0,0), tlo, thi, __kImage_texture2d); // Output!
    }
}

```



The code to broadcast the output to a single UAV looks something like this.

You can see that we start <CLICK> by manually converting our shaded result into what will be its final raw form in memory and caching the results in LDS.

<CLICK>We can then loop through the expanded set of commands, <CLICK>decode them and <CLICK> grab the result we stashed in LDS related to each of them.

Finally, <CLICK>, each lane can then figure out exactly where it's output should go and <CLICK> output it via a packed image store intrinsic.

```

thread_group_memory uint sIntermediateSpace[64]; // Space to store the pre converted output per lane
thread_group_memory unsigned char sIntermediatePixelX[64]; // X Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sIntermediatePixelY[64]; // Y Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sPixelCommands[256]; // Packed lane and offset from original position

void BroadcastOutput0(uint inNumPixels, uint inGroupIndex, uint2 inScreenTileCoords, float4 inColor)
{
    uint4 tlo = __get_tsharplo(SRT_DEFERREDTEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);
    uint4 thi = __get_tsharpri(SRT_DEFERREDTEXTURINGCOMPUTEPASSPARAMS_COLOR0TEX);

    sIntermediateSpace[inGroupIndex] = packUnorm4x8(inColor); // Cache manually converted results into LDS

    uint num_active_lanes = gWaveActiveLaneCount();

    for(int i=inGroupIndex; i< inNumPixels; i+=num_active_lanes) // Loop over the all the broadcast work
    {
        uint cmd = sPixelCommands[i]; // Read the work expanded command
        uint pixel_lane = BitFieldExtract(cmd,0,6); // What lane calculated this result
        uint pixel_offset_x = BitFieldExtract(cmd,6,1); // Where should it be offset from its original position
        uint pixel_offset_y = BitFieldExtract(cmd,7,1);

        uint col_out = sIntermediateSpace[pixel_lane]; // Read the manually converted result we stashed
        uint2 out_pixel = uint2(sIntermediatePixelX[pixel_lane], sIntermediatePixelY[pixel_lane]) + // Get its origin
            uint2(pixel_offset_x, pixel_offset_y) + // Add the 1 bit offset in x&y
            inScreenTileCoords; // Place in the right tile
        __image_store_pck(col_out, uint4(out_pixel,0,0), tlo, thi, __kImage_texture2d); // Output!
    }
}

```



81

The code to broadcast the output to a single UAV looks something like this.

You can see that we start <CLICK> by manually converting our shaded result into what will be its final raw form in memory and caching the results in LDS.

<CLICK>We can then loop through the expanded set of commands, <CLICK>decode them and <CLICK> grab the result we stashed in LDS related to each of them.

Finally, <CLICK>, each lane can then figure out exactly where it's output should go and <CLICK> output it via a packed image store intrinsic.

```

thread_group_memory uint sIntermediateSpace[64]; // Space to store the pre converted output per lane
thread_group_memory unsigned char sIntermediatePixelX[64]; // X Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sIntermediatePixelY[64]; // Y Coord of the intermediate pixel within the tile
thread_group_memory unsigned char sPixelCommands[256]; // Packed lane and offset from original position

void BroadcastOutput0(uint inNumPixels, uint inGroupIndex, uint2 inScreenTileCoords, float4 inColor)
{
    uint4 tlo = get_tsharplo(SRT_DEFERREDETEXTURINGCOMPUTEASSPARAMS_COLOR0TEX);
    uint4 thi = get_tsharpri(SRT_DEFERREDETEXTURINGCOMPUTEASSPARAMS_COLOR0TEX);

    sIntermediateSpace[inGroupIndex] = packUnorm4x8(inColor); // Cache manually converted results into LDS

    uint num_active_lanes = gWaveActiveLaneCount();

    for(int i=inGroupIndex; i< inNumPixels; i+=num_active_lanes) // Loop over the all the broadcast work
    {
        uint cmd = sPixelCommands[i]; // Read the work expanded command
        uint pixel_lane = BitFieldExtract(cmd,0,6); // What lane calculated this result
        uint pixel_offset_x = BitFieldExtract(cmd,6,1); // Where should it be offset from its original position
        uint pixel_offset_y = BitFieldExtract(cmd,7,1);

        uint col_out = sIntermediateSpace[pixel_lane]; // Read the manually converted result we stashed
        uint2 out_pixel = uint2(sIntermediatePixelX[pixel_lane], sIntermediatePixelY[pixel_lane]) + // Get is its origin
            uint2(pixel_offset_x, pixel_offset_y) + // Add the 1 bit offset in x&y
            inScreenTileCoords; // Place in the right tile
        image store pck(col out, uint4(out_pixel,0,0), tlo, thi, kImage texture2d); // Output!
    }
}

```



82

The code to broadcast the output to a single UAV looks something like this.

You can see that we start <CLICK> by manually converting our shaded result into what will be its final raw form in memory and caching the results in LDS.

<CLICK>We can then loop through the expanded set of commands, <CLICK>decode them and <CLICK> grab the result we stashed in LDS related to each of them.

Finally, <CLICK>, each lane can then figure out exactly where it's output should go and <CLICK> output it via a packed image store intrinsic.

```

uint num_pixels_to_write = 0;
bool broadcast_divergent = pixels_to_shade > 1; // Does this thread need to export more than one pixel?
bool broadcast_conservative_and_uniform = !gWaveMaskIsEmpty(gWaveMaskActiveBallot(broadcast_divergent)); // Do any threads in the
wave need to export more than one pixel?

if(broadcast_conservative_and_uniform)
{
    pixel = pixel &~0x1; // Take top left pixel in the quad
    float pixels_right = popcnt(vrs_mask&0xa);
    float pixels_bottom = popcnt(vrs_mask&0xc);
    wpos_offset = float2(pixels_right, pixels_bottom); //figure out where to shade in the quad
    // pixels_to_shade contains the total number of pixels this lane needs to write
    uint command_pos = gWaveActiveLanePrefixCount(pixels_to_shade&0x1) + // Calculate where our command's will go
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x2)<<1) + // using prefix count magic
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x4)<<2);

    for(int i=0; i<4;++i)
    {
        if(vrs_mask&(1<<i))
            sPixelCommands[command_pos++] = inGroupIndex | (i << 6); // Write a command for every broadcast pixel
    }

    sIntermediatePixelX[inGroupIndex] = pixel.x-screen_tile_coords.x; // Stash the tile local coords of the
    sIntermediatePixelY[inGroupIndex] = pixel.y-screen_tile_coords.y; // pixel this lane will shade

    num_pixels_to_write = gWaveActiveCountBits(pixels_to_shade&0x1) + // Count up how many pixels in total
        (gWaveActiveCountBits(pixels_to_shade&0x2)<<1) + // this wave will write
        (gWaveActiveCountBits(pixels_to_shade&0x4)<<2);
}

```



83

Here is a snippet of the code we need to insert near the top of our generated shader in order to create this expanded list of pixel commands in LDS.

<CLICK> We need to generate these expanded command if any lane in our wave needs to broadcast

For each lane <CLICK> we then figure out the top left pixel in the quad, and where <CLICK> we should calculate our shaded sample.

After which we can figure out where <CLICK> the commands for this lane should start in LDS, <CLICK> output them, <CLICK> and record the relative position of the quad in the tile.

<CLICK> Finally we calculate how many pixels this wave needs to output in total.

```

uint num_pixels_to_write = 0;
bool broadcast_divergent = pixels_to_shade > 1; // Does this thread need to export more than one pixel?
bool broadcast_conservative_and_uniform = !gWaveMaskIsEmpty(gWaveMaskActiveBallot(broadcast_divergent)); // Do any threads in the
wave need to export more than one pixel?

if(broadcast_conservative_and_uniform)
{
    pixel = pixel &~0x1; // Take top left pixel in the quad
    float pixels_right = popcnt(vrs_mask&0xa);
    float pixels_bottom = popcnt(vrs_mask&0xc);
    wpos_offset = float2(pixels_right, pixels_bottom); //figure out where to shade in the quad
    // pixels_to_shade contains the total number of pixels this lane needs to write
    uint command_pos = gWaveActiveLanePrefixCount(pixels_to_shade&0x1) + // Calculate where our command's will go
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x2)<<1) + // using prefix count magic
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x4)<<2);

    for(int i=0; i<4;++i)
    {
        if(vrs_mask&(1<<i))
            sPixelCommands[command_pos++] = inGroupIndex | (i << 6); // Write a command for every broadcast pixel
    }

    sIntermediatePixelX[inGroupIndex] = pixel.x-screen_tile_coords.x; // Stash the tile local coords of the
    sIntermediatePixelY[inGroupIndex] = pixel.y-screen_tile_coords.y; // pixel this lane will shade

    num_pixels_to_write = gWaveActiveCountBits(pixels_to_shade&0x1) + // Count up how many pixels in total
        (gWaveActiveCountBits(pixels_to_shade&0x2)<<1) + // this wave will write
        (gWaveActiveCountBits(pixels_to_shade&0x4)<<2);
}

```



84

Here is a snippet of the code we need to insert near the top of our generated shader in order to create this expanded list of pixel commands in LDS.

<CLICK> We need to generate these expanded command if any lane in our wave needs to broadcast

For each lane <CLICK> we then figure out the top left pixel in the quad, and where <CLICK> we should calculate our shaded sample.

After which we can figure out where <CLICK> the commands for this lane should start in LDS, <CLICK> output them, <CLICK> and record the relative position of the quad in the tile.

<CLICK> Finally we calculate how many pixels this wave needs to output in total.

```

uint num_pixels_to_write = 0;
bool broadcast_divergent = pixels_to_shade > 1; // Does this thread need to export more than one pixel?
bool broadcast_conservative_and_uniform = !gWaveMaskIsEmpty(gWaveMaskActiveBallot(broadcast_divergent)); // Do any threads in the
wave need to export more than one pixel?

if(broadcast_conservative_and_uniform)
{
    pixel = pixel &~0x1; // Take top left pixel in the quad
    float pixels_right = popcnt(vrs_mask&0xa);
    float pixels_bottom = popcnt(vrs_mask&0xc);
    wpos_offset = float2(pixels_right, pixels_bottom); //figure out where to shade in the quad
    // pixels_to_shade contains the total number of pixels this lane needs to write
    uint command_pos = gWaveActiveLanePrefixCount(pixels_to_shade&0x1) + // Calculate where our command's will go
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x2)<<1) + // using prefix count magic
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x4)<<2);

    for(int i=0; i<4;++i)
    {
        if(vrs_mask&(1<<i))
            sPixelCommands[command_pos++] = inGroupIndex | (i << 6); // Write a command for every broadcast pixel
    }

    sIntermediatePixelX[inGroupIndex] = pixel.x-screen_tile_coords.x; // Stash the tile local coords of the
    sIntermediatePixelY[inGroupIndex] = pixel.y-screen_tile_coords.y; // pixel this lane will shade

    num_pixels_to_write = gWaveActiveCountBits(pixels_to_shade&0x1) + // Count up how many pixels in total
        (gWaveActiveCountBits(pixels_to_shade&0x2)<<1) + // this wave will write
        (gWaveActiveCountBits(pixels_to_shade&0x4)<<2);
}

```



85

Here is a snippet of the code we need to insert near the top of our generated shader. In order to create this expanded list of pixel commands in LDS.

<CLICK> We need to generate these expanded command if any lane in our wave needs to broadcast

For each lane <CLICK> we then figure out the top left pixel in the quad, and where <CLICK> we should calculate our shaded sample.

After which we can figure out where <CLICK> the commands for this lane should start in LDS, <CLICK> output them, <CLICK> and record the relative position of the quad in the tile.

<CLICK> Finally we calculate how many pixels this wave needs to output in total.

```

uint num_pixels_to_write = 0;
bool broadcast_divergent = pixels_to_shade > 1; // Does this thread need to export more than one pixel?
bool broadcast_conservative_and_uniform = !gWaveMaskIsEmpty(gWaveMaskActiveBallot(broadcast_divergent)); // Do any threads in the
wave need to export more than one pixel?

if(broadcast_conservative_and_uniform)
{
    pixel = pixel &~0x1; // Take top left pixel in the quad
    float pixels_right = popcnt(vrs_mask&0xa);
    float pixels_bottom = popcnt(vrs_mask&0xc);
    wpos_offset = float2(pixels_right, pixels_bottom); //figure out where to shade in the quad
    // pixels_to_shade contains the total number of pixels this lane needs to write
    uint command_pos = gWaveActiveLanePrefixCount(pixels_to_shade&0x1) + // Calculate where our command's will go
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x2)<<1) + // using prefix count magic
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x4)<<2);

    for(int i=0; i<4;++i)
    {
        if(vrs_mask&(1<<i))
            sPixelCommands[command_pos++] = inGroupIndex | (i << 6); // Write a command for every broadcast pixel
    }

    sIntermediatePixelX[inGroupIndex] = pixel.x-screen_tile_coords.x; // Stash the tile local coords of the
    sIntermediatePixelY[inGroupIndex] = pixel.y-screen_tile_coords.y; // pixel this lane will shade

    num_pixels_to_write = gWaveActiveCountBits(pixels_to_shade&0x1) + // Count up how many pixels in total
        (gWaveActiveCountBits(pixels_to_shade&0x2)<<1) + // this wave will write
        (gWaveActiveCountBits(pixels_to_shade&0x4)<<2);
}

```



86

Here is a snippet of the code we need to insert near the top of our generated shader. In order to create this expanded list of pixel commands in LDS.

<CLICK> We need to generate these expanded command if any lane in our wave needs to broadcast

For each lane <CLICK> we then figure out the top left pixel in the quad, and where <CLICK> we should calculate our shaded sample.

After which we can figure out where <CLICK> the commands for this lane should start in LDS, <CLICK> output them, <CLICK> and record the relative position of the quad in the tile.

<CLICK> Finally we calculate how many pixels this wave needs to output in total.

```

uint num_pixels_to_write = 0;
bool broadcast_divergent = pixels_to_shade > 1; // Does this thread need to export more than one pixel?
bool broadcast_conservative_and_uniform = !gWaveMaskIsEmpty(gWaveMaskActiveBallot(broadcast_divergent)); // Do any threads in the
wave need to export more than one pixel?

if(broadcast_conservative_and_uniform)
{
    pixel = pixel &~0x1; // Take top left pixel in the quad
    float pixels_right = popcnt(vrs_mask&0xa);
    float pixels_bottom = popcnt(vrs_mask&0xc);
    wpos_offset = float2(pixels_right, pixels_bottom); //figure out where to shade in the quad
    // pixels_to_shade contains the total number of pixels this lane needs to write
    uint command_pos = gWaveActiveLanePrefixCount(pixels_to_shade&0x1) + // Calculate where our command's will go
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x2)<<1) + // using prefix count magic
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x4)<<2);

    for(int i=0; i<4;++i)
    {
        if(vrs_mask&(1<<i))
            sPixelCommands[command_pos++] = inGroupIndex | (i << 6); // Write a command for every broadcast pixel
    }

    sIntermediatePixelX[inGroupIndex] = pixel.x-screen_tile_coords.x; // Stash the tile local coords of the
    sIntermediatePixelY[inGroupIndex] = pixel.y-screen_tile_coords.y; // pixel this lane will shade

    num_pixels_to_write = gWaveActiveCountBits(pixels_to_shade&0x1) + // Count up how many pixels in total
        (gWaveActiveCountBits(pixels_to_shade&0x2)<<1) + // this wave will write
        (gWaveActiveCountBits(pixels_to_shade&0x4)<<2);
}

```



87

Here is a snippet of the code we need to insert near the top of our generated shader. In order to create this expanded list of pixel commands in LDS.

<CLICK> We need to generate these expanded commands if any lane in our wave needs to broadcast.

For each lane <CLICK> we then figure out the top left pixel in the quad, and where <CLICK> we should calculate our shaded sample.

After which we can figure out where <CLICK> the commands for this lane should start in LDS, <CLICK> output them, <CLICK> and record the relative position of the quad in the tile.

<CLICK> Finally we calculate how many pixels this wave needs to output in total.

```

uint num_pixels_to_write = 0;
bool broadcast_divergent = pixels_to_shade > 1; // Does this thread need to export more than one pixel?
bool broadcast_conservative_and_uniform = !gWaveMaskIsEmpty(gWaveMaskActiveBallot(broadcast_divergent)); // Do any threads in the
wave need to export more than one pixel?

if(broadcast_conservative_and_uniform)
{
    pixel = pixel &~0x1; // Take top left pixel in the quad
    float pixels_right = popcnt(vrs_mask&0xa);
    float pixels_bottom = popcnt(vrs_mask&0xc);
    wpos_offset = float2(pixels_right, pixels_bottom); //figure out where to shade in the quad
    // pixels_to_shade contains the total number of pixels this lane needs to write
    uint command_pos = gWaveActiveLanePrefixCount(pixels_to_shade&0x1) + // Calculate where our command's will go
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x2)<<1) + // using prefix count magic
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x4)<<2);

    for(int i=0; i<4;++i)
    {
        if(vrs_mask&(1<<i))
            sPixelCommands[command_pos++] = inGroupIndex | (i << 6); // Write a command for every broadcast pixel
    }

    sIntermediatePixelX[inGroupIndex] = pixel.x-screen_tile_coords.x; // Stash the tile local coords of the
    sIntermediatePixelY[inGroupIndex] = pixel.y-screen_tile_coords.y; // pixel this lane will shade

    num_pixels_to_write = gWaveActiveCountBits(pixels_to_shade&0x1) + // Count up how many pixels in total
        (gWaveActiveCountBits(pixels_to_shade&0x2)<<1) + // this wave will write
        (gWaveActiveCountBits(pixels_to_shade&0x4)<<2);
}

```



88

Here is a snippet of the code we need to insert near the top of our generated shader
 In order to create this expanded list of pixel commands in LDS.

<CLICK> We need to generate these expanded command if any lane in our wave
 needs to broadcast

For each lane <CLICK> we then figure out the top left pixel in the quad, and where
 <CLICK> we should calculate our shaded sample.

After which we can figure out where <CLICK> the commands for this lane should start
 in LDS, <CLICK> output them, <CLICK> and record the relative position of the quad in
 the tile.

<CLICK> Finally we calculate how many pixels this wave needs to output in total.

```

uint num_pixels_to_write = 0;
bool broadcast_divergent = pixels_to_shade > 1; // Does this thread need to export more than one pixel?
bool broadcast_conservative_and_uniform = !gWaveMaskIsEmpty(gWaveMaskActiveBallot(broadcast_divergent )); // Do any threads in the
wave need to export more than one pixel?

if(broadcast_conservative_and_uniform)
{
    pixel = pixel &~0x1; // Take top left pixel in the quad
    float pixels_right = popcnt(vrs_mask&0xa);
    float pixels_bottom = popcnt(vrs_mask&0xc);
    wpos_offset = float2(pixels_right, pixels_bottom); //figure out where to shade in the quad
    // pixels_to_shade contains the total number of pixels this lane needs to write
    uint command_pos = gWaveActiveLanePrefixCount(pixels_to_shade&0x1) + // Calculate where our command's will go
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x2)<<1) + // using prefix count magic
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x4)<<2);

    for(int i=0; i<4;++i)
    {
        if(vrs_mask&(1<<i))
            sPixelCommands[command_pos++] = inGroupIndex | (i << 6); // Write a command for every broadcast pixel
    }

    sIntermediatePixelX[inGroupIndex] = pixel.x-screen tile coords.x; // Stash the tile local coords of the
    sIntermediatePixelY[inGroupIndex] = pixel.y-screen tile coords.y; // pixel this lane will shade

    num_pixels_to_write = gWaveActiveCountBits(pixels_to_shade&0x1) + // Count up how many pixels in total
        (gWaveActiveCountBits(pixels_to_shade&0x2)<<1) + // this wave will write
        (gWaveActiveCountBits(pixels_to_shade&0x4)<<2);
}

```



89

Here is a snippet of the code we need to insert near the top of our generated shader in order to create this expanded list of pixel commands in LDS.

<CLICK> We need to generate these expanded command if any lane in our wave needs to broadcast

For each lane <CLICK> we then figure out the top left pixel in the quad, and where <CLICK> we should calculate our shaded sample.

After which we can figure out where <CLICK> the commands for this lane should start in LDS, <CLICK> output them, <CLICK> and record the relative position of the quad in the tile.

<CLICK> Finally we calculate how many pixels this wave needs to output in total.

```

uint num_pixels_to_write = 0;
bool broadcast_divergent = pixels_to_shade > 1; // Does this thread need to export more than one pixel?
bool broadcast_conservative_and_uniform = !gWaveMaskIsEmpty(gWaveMaskActiveBallot(broadcast_divergent)); // Do any threads in the
wave need to export more than one pixel?

if(broadcast_conservative_and_uniform)
{
    pixel = pixel &~0x1; // Take top left pixel in the quad
    float pixels_right = popcnt(vrs_mask&0xa);
    float pixels_bottom = popcnt(vrs_mask&0xc);
    wpos_offset = float2(pixels_right, pixels_bottom); //figure out where to shade in the quad
    // pixels_to_shade contains the total number of pixels this lane needs to write
    uint command_pos = gWaveActiveLanePrefixCount(pixels_to_shade&0x1) + // Calculate where our command's will go
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x2)<1) + // using prefix count magic
        (gWaveActiveLanePrefixCount(pixels_to_shade&0x4)<2);

    for(int i=0; i<4;++i)
    {
        if(vrs_mask&(1<<i))
            sPixelCommands[command_pos++] = inGroupIndex | (i << 6); // Write a command for every broadcast pixel
    }

    sIntermediatePixelX[inGroupIndex] = pixel.x-screen_tile_coords.x; // Stash the tile local coords of the
    sIntermediatePixelY[inGroupIndex] = pixel.y-screen_tile_coords.y; // pixel this lane will shade

    num_pixels_to_write = gWaveActiveCountBits(pixels_to_shade&0x1) + // Count up how many pixels in total
        (gWaveActiveCountBits(pixels_to_shade&0x2)<1) + // this wave will write
        (gWaveActiveCountBits(pixels_to_shade&0x4)<2);
}

```



90

Here is a snippet of the code we need to insert near the top of our generated shader. In order to create this expanded list of pixel commands in LDS.

<CLICK> We need to generate these expanded commands if any lane in our wave needs to broadcast.

For each lane <CLICK> we then figure out the top left pixel in the quad, and where <CLICK> we should calculate our shaded sample.

After which we can figure out where <CLICK> the commands for this lane should start in LDS, <CLICK> output them, <CLICK> and record the relative position of the quad in the tile.

<CLICK> Finally we calculate how many pixels this wave needs to output in total.

```

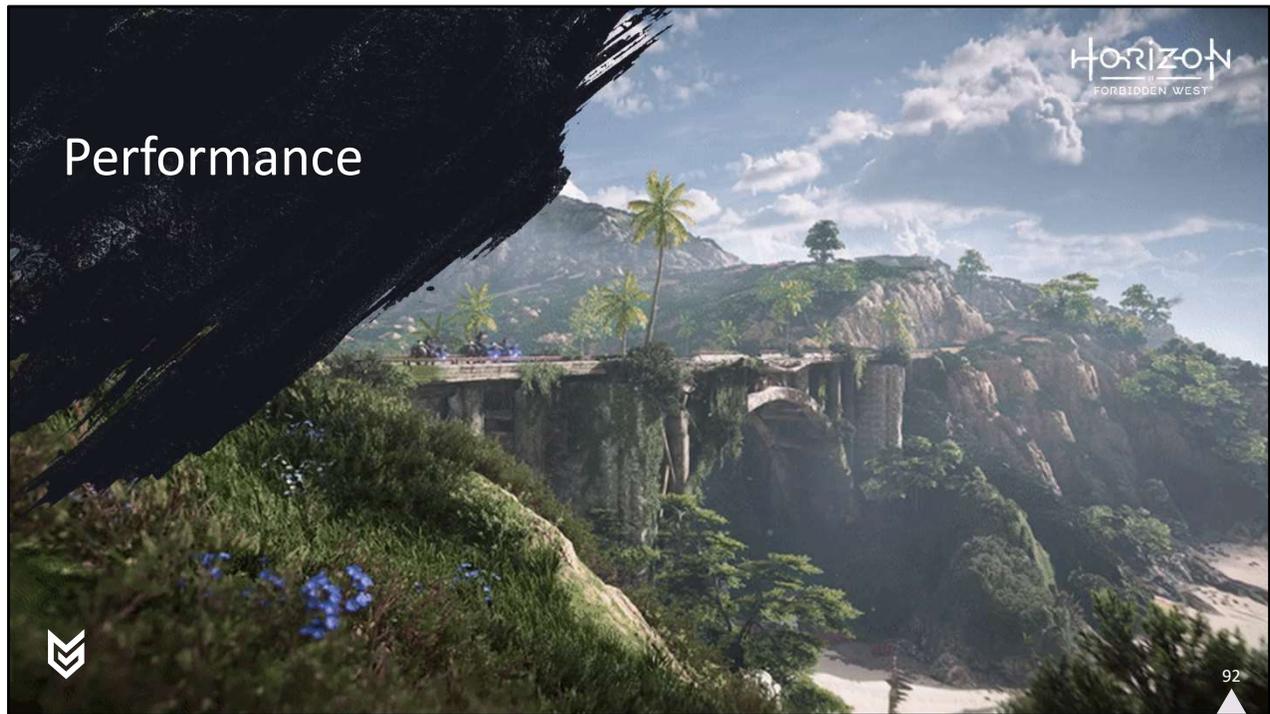
// Body of generated shader code

if(broadcast_conservative_and_uniform) // Determine if we need to broadcast
{
    // Whole wave uses broadcast functions
    BroadcastOutput0(num_pixels_to_write, inGroupIndex, screen_tile_coords, out_color0_srgb);
    BroadcastOutput2(num_pixels_to_write, inGroupIndex, screen_tile_coords, o_outColor2);
    BroadcastOutput3(num_pixels_to_write, inGroupIndex, screen_tile_coords, o_outColor3);
    BroadcastOutput4(num_pixels_to_write, inGroupIndex, screen_tile_coords, o_outColor4);
    BroadcastOutput5(num_pixels_to_write, inGroupIndex, screen_tile_coords, o_outColor5);
}
else
{
    // Whole wave outputs directly to UAVs
    SRT_DEFERREDTEXTURINGCOMPUTE_PASSPARAMS_COLOR0TEX[pixel] = out_color0_srgb;
    SRT_DEFERREDTEXTURINGCOMPUTE_PASSPARAMS_COLOR2TEX[pixel] = o_outColor2;
    SRT_DEFERREDTEXTURINGCOMPUTE_PASSPARAMS_COLOR3TEX[pixel] = o_outColor3;
    SRT_DEFERREDTEXTURINGCOMPUTE_PASSPARAMS_COLOR4TEX[pixel] = o_outColor4;
    SRT_DEFERREDTEXTURINGCOMPUTE_PASSPARAMS_COLOR5TEX[pixel] = o_outColor5;
}
}

```

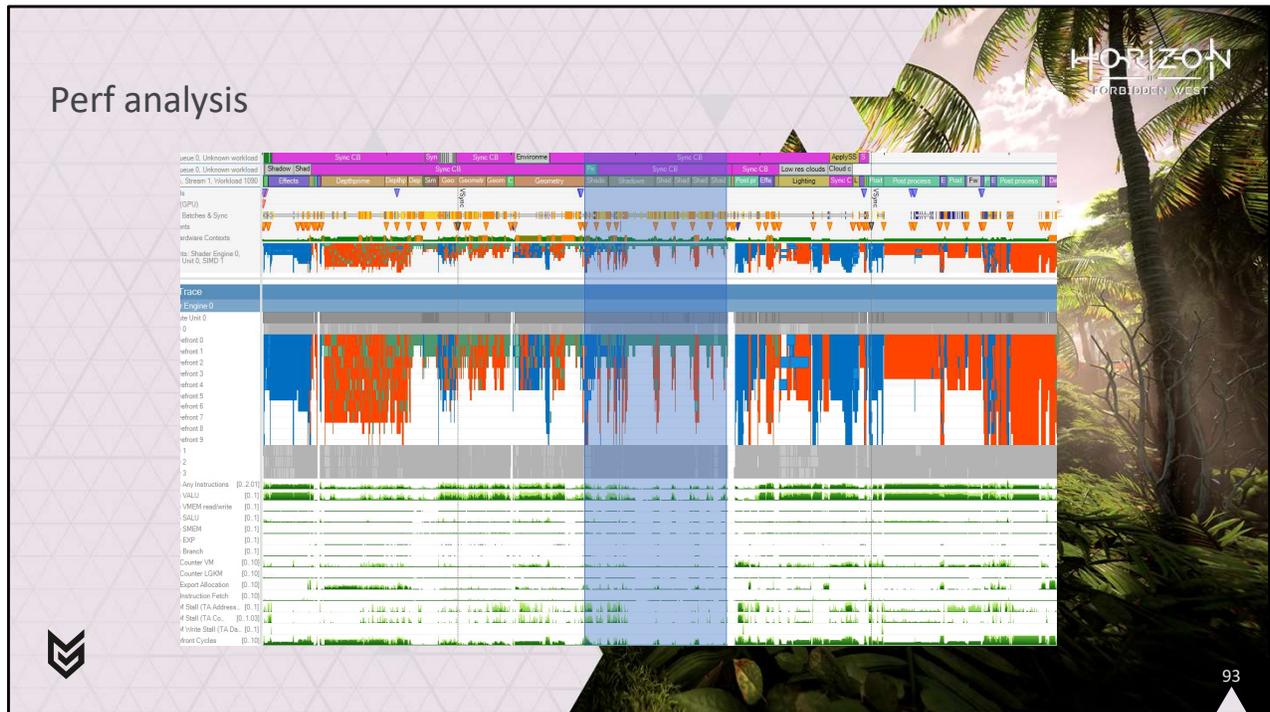


Then at the end of the shader we select between a fast path for normal output and the broadcast output path.



OK, that's all I have time for on the VRS. Let's go and have a little look at how the whole system performs.

Perf analysis

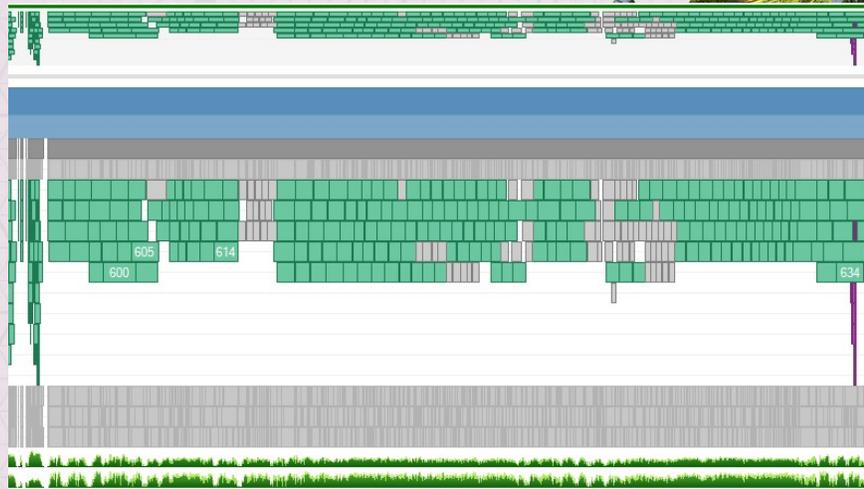


Let's take a look at our frame with the large shadow cap from the start of the presentation.

As you can see, we now have what appears to be a fairly solid block of compute work overlapping with the shadows.

This is in fact 10's of passes transforming vertices into our ring buffer and then consuming that to shade pixels on the screen.

Perf analysis



95

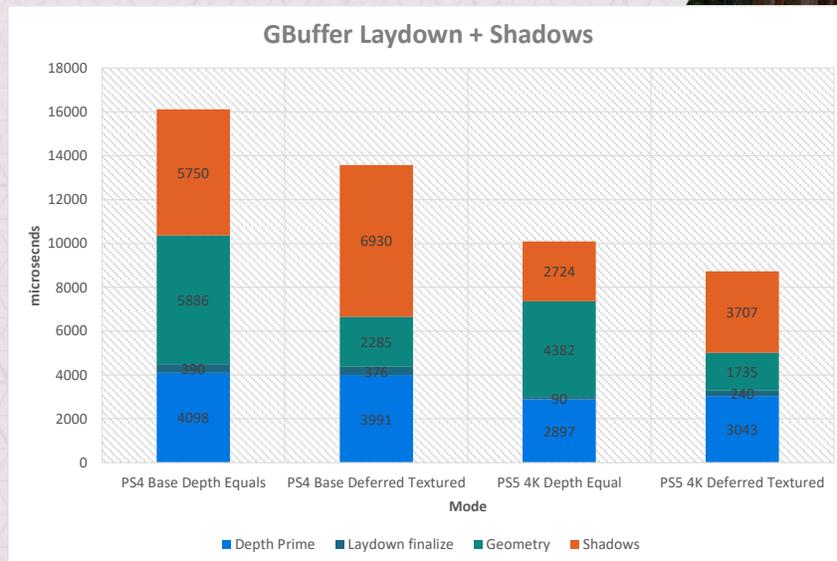
If we don't overlap with the shadows and run the pixel shading work on the graphics pipe while still keeping the vertex shading on the compute pipe, you can see how everything is nicely interleaved. In this picture the green wavefronts are pixel work and the grey wavefronts are the vertex work.

Perf analysis



Switching to a colouring based on the batches, you can see how all the different batches on the same pipes also interleave nicely.

Forest



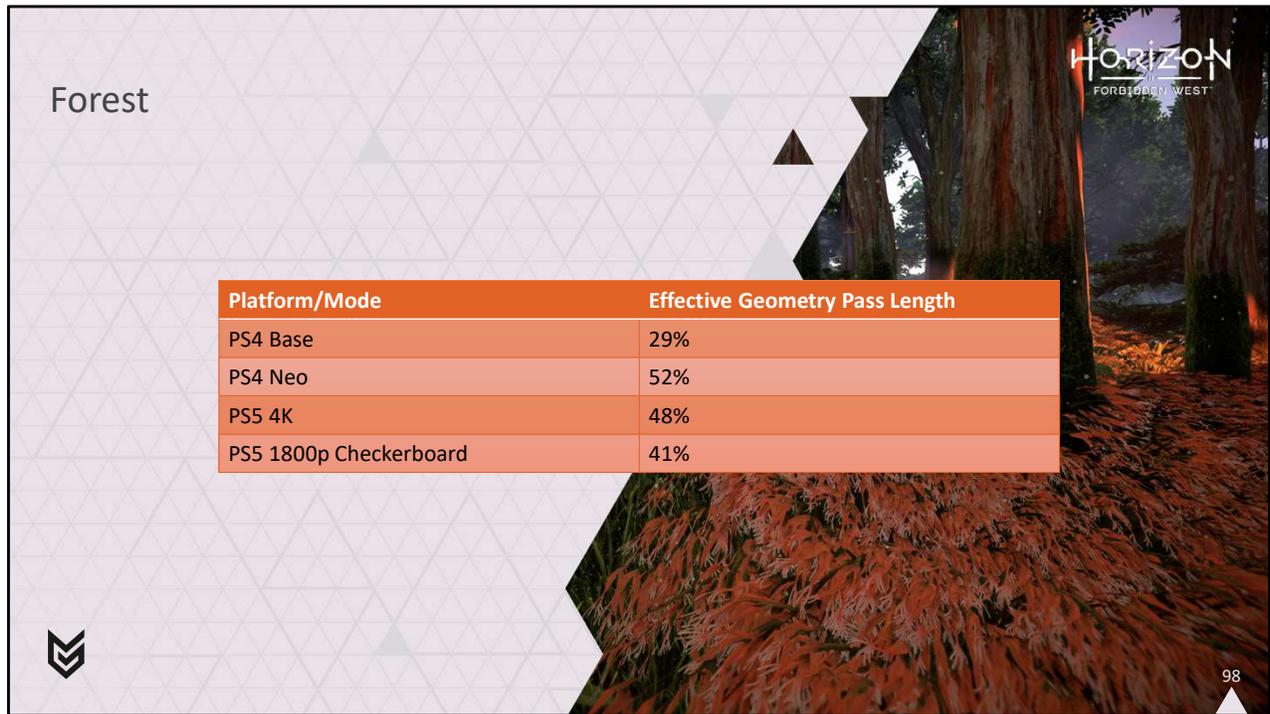
So, here are some numbers to give you an idea of how this performs on a couple of scenes.

First off, we have a forest scene.

Here you can see that we get reasonable gains of about 1.4ms on PS5 at 4K, but the real winner here is PS4.

On base PS4 due to overlapping with the shadows we get back almost 2.5 ms which is pretty useful.

Forest

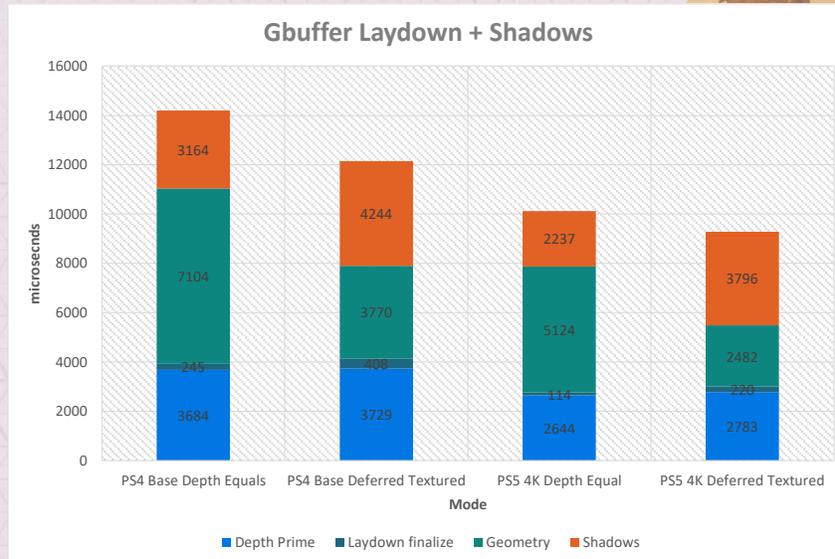


Platform/Mode	Effective Geometry Pass Length
PS4 Base	29%
PS4 Neo	52%
PS5 4K	48%
PS5 1800p Checkerboard	41%

98

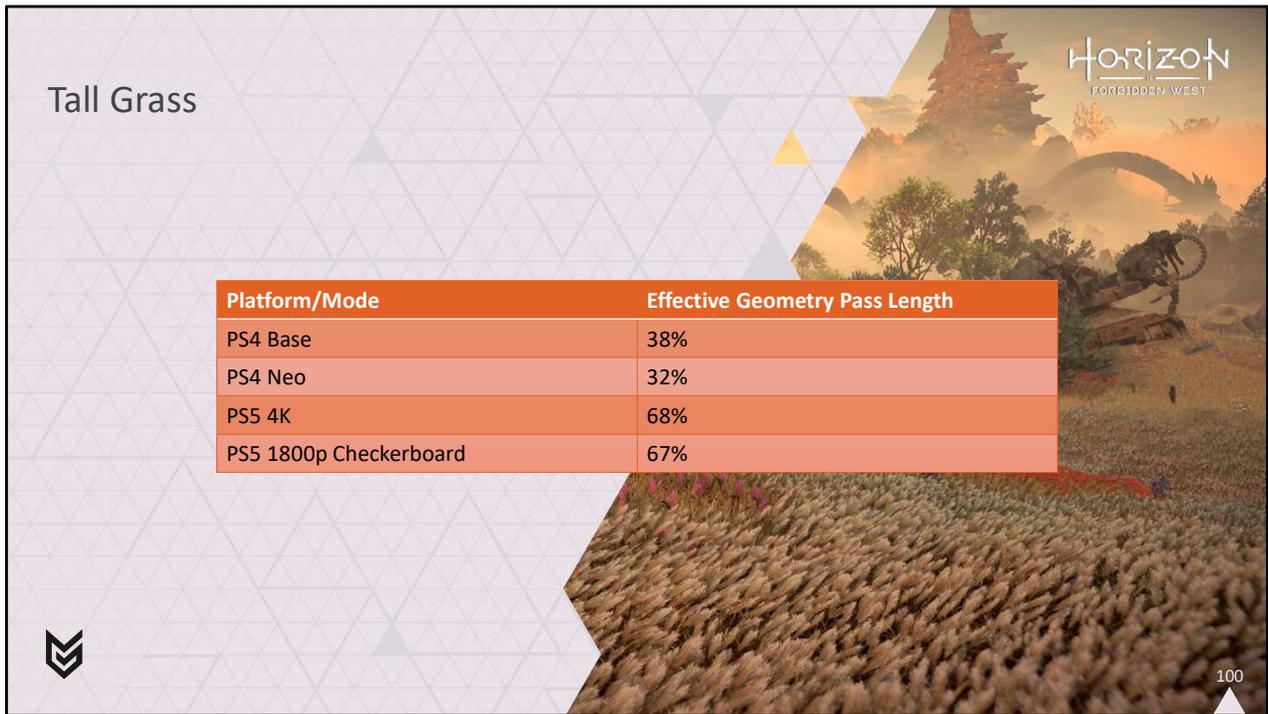
And if we isolate the work we used to have in the geometry pass for shading foliage and look at how much time that work is now effectively taking in the frame now that it's resolved by deferred texturing and overlapped with the shadows, you can see that we are getting some pretty healthy gains.

Tall Grass



In this grassland scene, you can see that we're also getting some good gains but not quite as much as for the forest.
PS4 is still the big winner here, but PS5 is also benefitting.
This scene is slightly more challenging because there is so much vertex work, and the culling is less effective.

Tall Grass



The image shows a screenshot from the game Horizon Forbidden West. The scene depicts a vast landscape with tall, golden-brown grass in the foreground. In the background, there are mountains, a large stone archway, and a sunset sky. The game's title 'HORIZON FORBIDDEN WEST' is visible in the top right corner. A table is overlaid on the scene, providing performance data for different platforms and modes. The table has two columns: 'Platform/Mode' and 'Effective Geometry Pass Length'. The data shows that PS5 4K has the highest effective geometry pass length at 68%, followed by PS5 1800p Checkerboard at 67%, PS4 Base at 38%, and PS4 Neo at 32%. A small '100' with a triangle icon is visible in the bottom right corner of the screenshot.

Platform/Mode	Effective Geometry Pass Length
PS4 Base	38%
PS4 Neo	32%
PS5 4K	68%
PS5 1800p Checkerboard	67%

And again if you look at the effective length of the work that we are now doing in deferred texturing that used to be done in the geometry pass, you can see that we're still making some good gains.

Pixel Work Reduction

Scene	Pixel shading waves with depth equal	Pixel shading waves with deferred texturing	Ratio
Forest 4K PS5	276500	111180	40%
Tall Grass 4K PS5	330786	85528	25%



101

Finally, we can also see how much the system is managing to cut down on overshading with pixel work

Vertex Work Reduction

Scene	Vertices in the scene	Vertices after culling	Ratio
Forest 4K PS5	3867456	1241664	32%
Tall Grass 4K PS5	5077248	2774784	55%



102

And also how much work the vertex culling we're doing is managing to cut out.



Thank you!

Also many thanks to all of the Guerrilla Tech Team, especially Hugh Malan and Jonas Meyer for suffering through code reviews of this system.

A big thanks too to Gilbert Sanders and Team Green for being such great collaborators and making such awesome foliage.

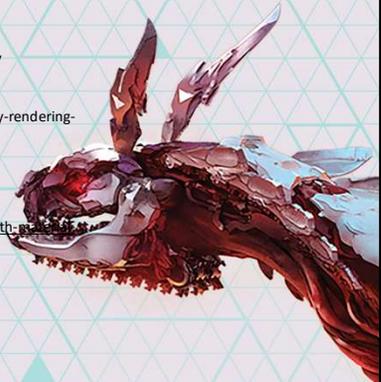
Thanks also to Maarten van der Gaag for assistance and suggestions getting this working in our shader graph!

And that brings me to the end of my talk.

I hope you've enjoyed journeying on this adventure with me. We've looked at our foliage shading with new eyes, and though the pixels look almost exactly the same as when we started, I hope you'll agree that the grass now looks a just a little bit greener.... at least from a framerate perspective.

References

1. Sanders 2018 "Between Texh and Art: The Vegetation of Horizon Zero Dawn" <https://youtu.be/wavnKZNSyqU>
2. Burns and Hunt 2013, "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading" <http://jcgf.org/published/0002/02/04/>
3. Stachowiak 2015, "A Deferred Material Rendering System" <https://onedrive.live.com/view.aspx?resid=EBE7DEDA70D06DA01115&app=PowerPoint&authkey=IAP-pDh4IMUug6vs>
4. Doghramachi and Bucci 2017, "Deferred+: Next-Gen Culling and Rendering for Dawn Engine" <http://guzen.blogspot.com/>
<https://www.eidosmontreal.com/news/deferred-next-gen-culling-and-rendering-for-dawn-engine/>
5. Drobot 2021 "Geometry Rendering Pipeline Architecture" <https://research.activision.com/publications/2021/09/geometry-rendering-pipeline-architecture>
6. Karis 2021 "A Deep Dive into Nanite Virtualized Geometry" https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf
7. Hable 2021, "Visibility Buffer Rendering With Material Graphs" <http://filmicworlds.com/blog/visibility-buffer-rendering-with-material-graphs/>



04/04/2022

HORIZON
FORBIDDEN WEST

Questions?

Twitter: @selfresonating



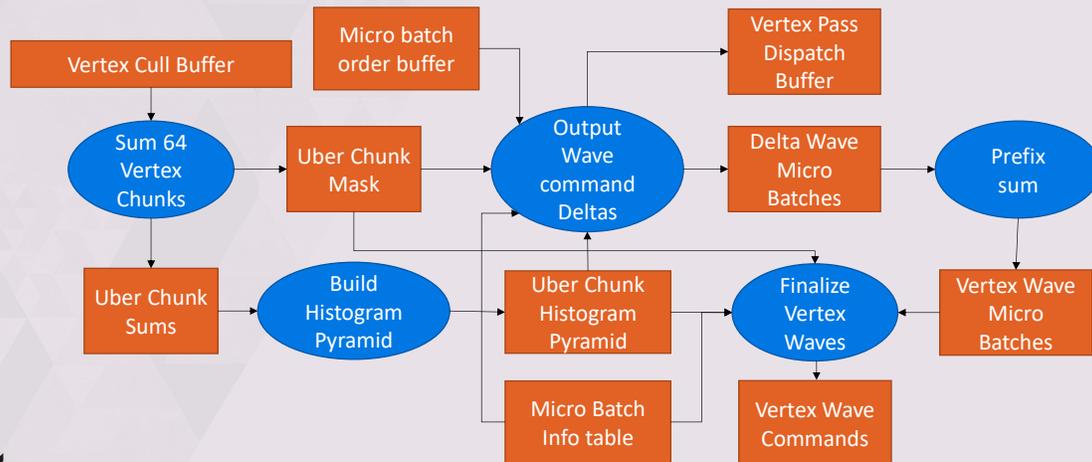
105

Bonus Slides



Visibility Buffer Classification

Mid Classify - Vertices



107

<This is a brief explanation of how we generate the vertex wave commands>

So we take the vertex cull buffer, and sum up how many vertex chunks are visible in what we call an uber chunk, which is a set of 64 chunks of 64 vertices.

We also spit out a 64 bit mask per uber chunk that tells us which chunks are set.

With the uber chunk sums we can then construct a histogram pyramid, that will allow us to easily map from a linear index of a work item to an uber chunk and the sub index of the child vertex chunk within it.

Using the 64 bit mask that we constructed per uber chunk, we can then easily transform this child vertex chunk index back into the global index of a vertex chunk.

We do a dispatch over all the micro batches, read the microbatch info table and use the mapping we just described to figure out how many vertex waves we will need for each pass.

We read a micro batch order buffer that we setup on the CPU describing the order in which micro batches will be processed.

This is used to atomically write a delta of the micro batch ID at each position where in the vertex wave command stream where the micro batch ID will change.

A prefix sum is then used to transform from the deltas to the actual micro batch IDs.

Then we finish off by doing an indirect dispatch over all the vertex wave commands

we will need,

Reading the histogram pyramid and mask along with the microbatch info table to look up which vertex chunk each wave command should be transforming and add this info into the command along with the already recorded micro batch.

Batch Groups

- ▶ Allow 32 Batch Groups per pass.
 - A Batch Group is N batches with the same shader and per batch data and geometry.
 - Batches within a batch group differ only by their set of instances.



<This slide was (perhaps foolishly) cut, it used to live directly after slide 44, this limit of 32 makes a sense of some of the mask/counter stuff in later diagrams>

For each pass we support up to 32 batch groups.

A batch group is a set of batches that have the same shader and per batch data, but different instances.

Ideally, we'd like to not have this 32 limit, but it's currently a consequence of how some of our Visibility Buffer classification work is structured.

Load Balancing – CUT Maybe mention in perf section

- ▶ Record the time of Shadows and Deferred Texturing with performance counters.
- ▶ Use this to the difference in time to drive changes in compute wave limits.
- ▶ SetComputeShaderControl() on PS4/5.
- ▶ Target the Deferred Texturing work to take ~90% of the shadow time.
- ▶ Ensures optimal mix of work.



Our shading work is going to run in parallel with the shadows, and ideally, we want the time the compute work runs to roughly match the time the shadows take.

This should hopefully ensure an optimal mixing of the two workloads.

Having either run significantly longer than the other would lead to inefficiency.

To achieve this ideal balance, we use performance counters to understand how long each took in the last frame.

Then we use this to gradually tune the parameters for SetComputeShaderControl() so that we can change the rate at which wavefronts of compute work are generated.

We target the compute work to take around 90% of the shadow time as there is usually a depth decompression at the end of the shadow pass that tends to not overlap very nicely.

Also, we want to avoid the situation where the compute work is tuned to not be using all the wavefronts that it can but is running long. This is usually a much worse situation than the shadows running longer.

Shader Graph Integration

- ▶ Generates compute shader pairs rather than Pixel and Vertex shader pairs.
 - Support for skinning etc. just works out of the box.
- ▶ Derivatives for Deferred Texturing provided by a special node.
 - Worked for this project but would like to invest in automatic differentiation in future.

