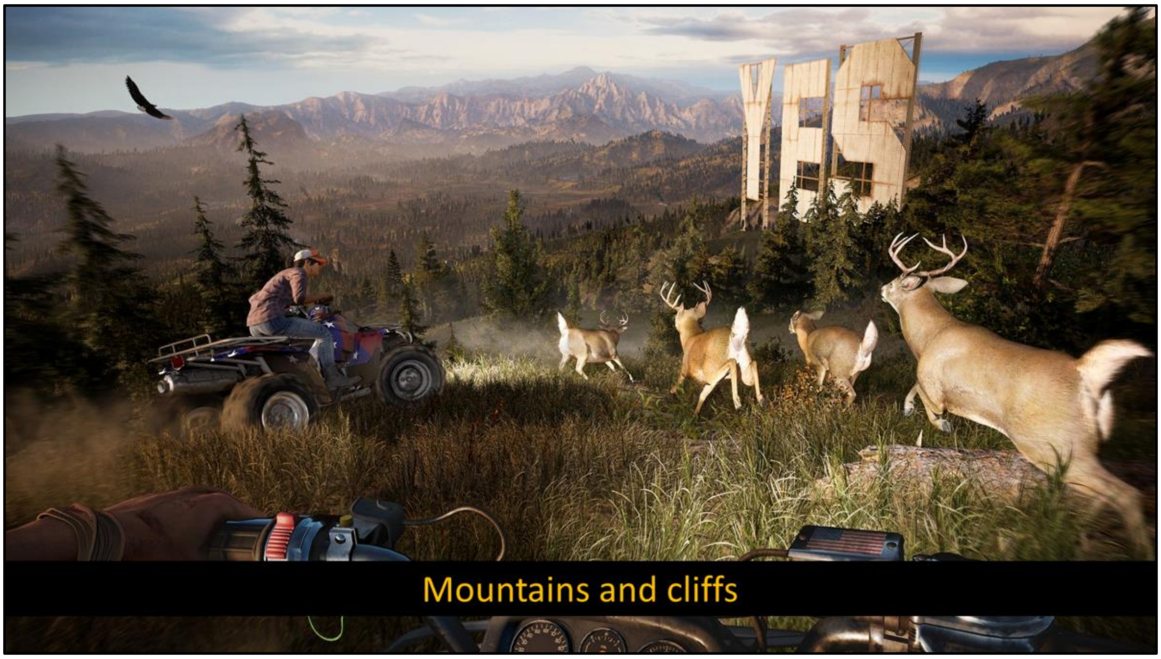
Terrain Rendering in ‘Far Cry 5’

Jeremy Moore
Tech Lead 3D, Ubisoft Montréal

GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18

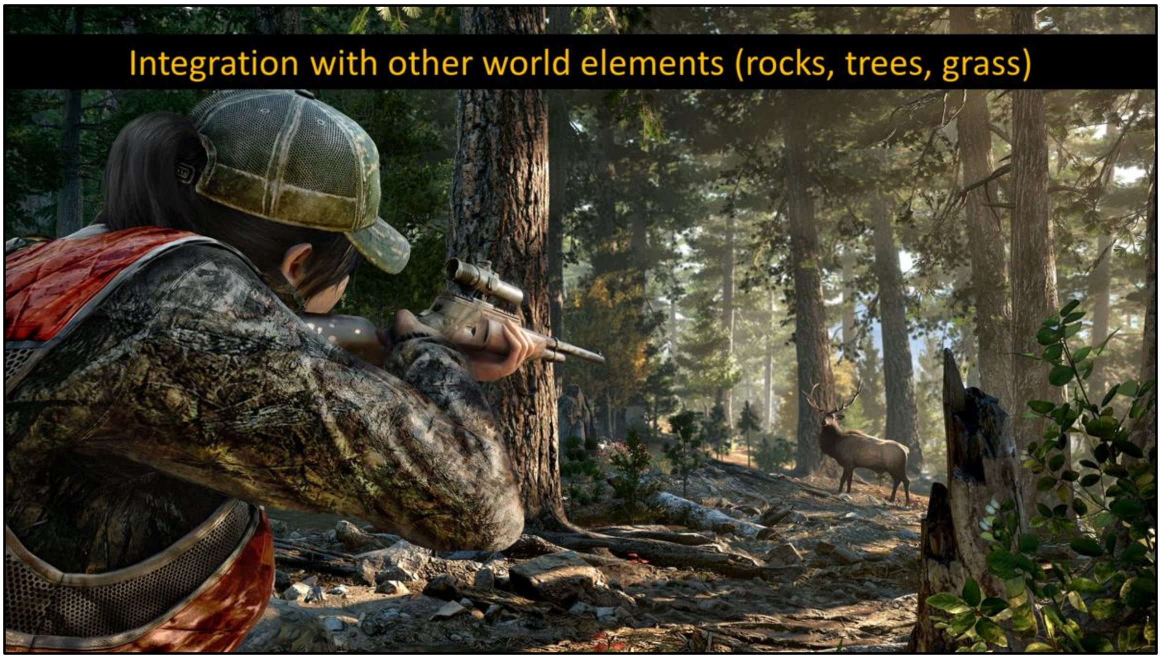


Mountains and cliffs

Far Cry 5 is an open world first person shooter set in the beautiful landscape of Montana.
We need to support rendering big vistas with large areas of mountain and cliff at all levels of detail.

High detail shading and geometry close to camera

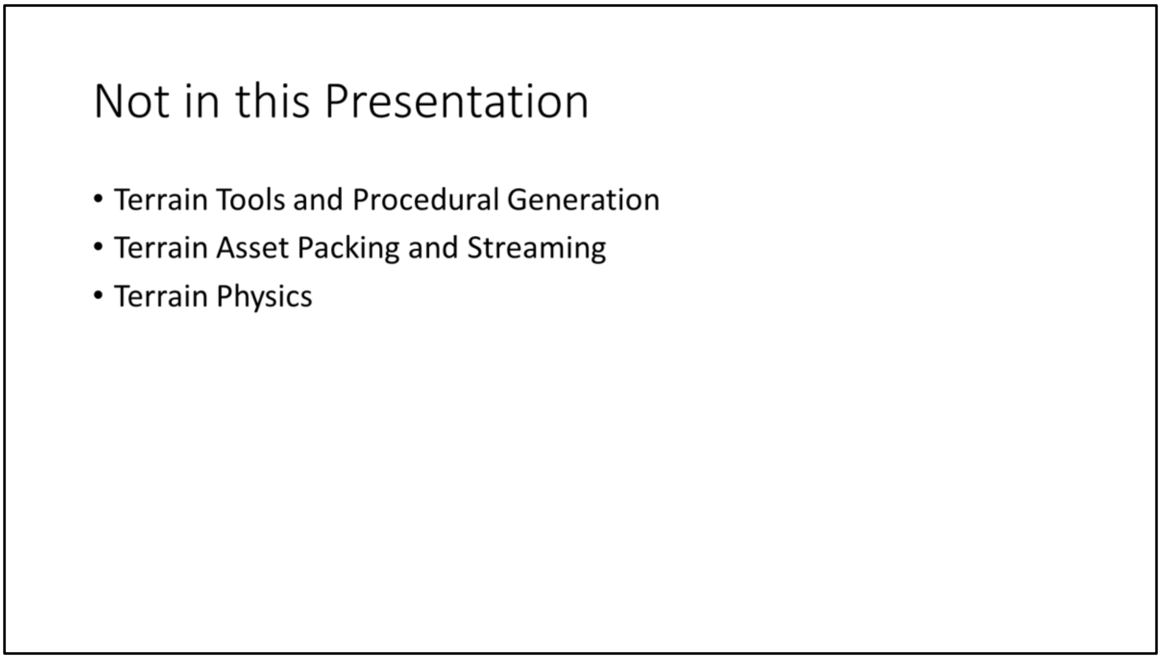
We also need to support a high level of material and geometric detail close to the camera.
Here we see fine geometric and shading detail for rocks and mud in an outpost.



Integration with other world elements (rocks, trees, grass)

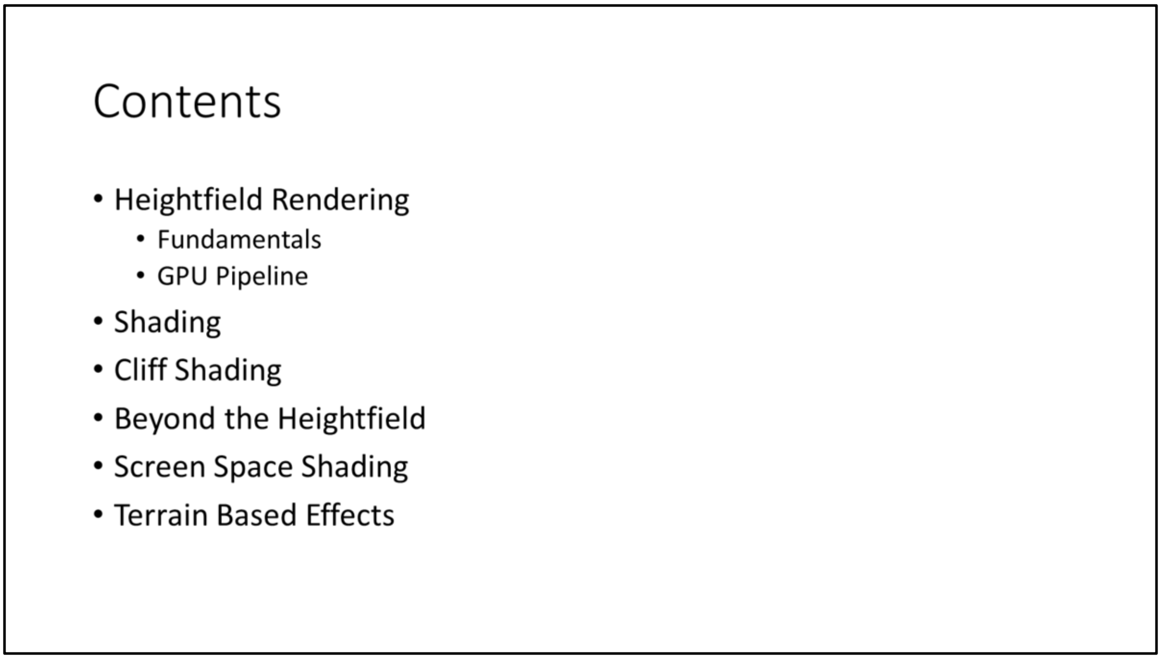
Also we need to make it easy to integrate other world elements with the terrain.
We want rocks, trees and grass to blend with, and be grounded in, the terrain.

We wanted to achieve all of this with a smaller budget for terrain rendering then we had in previous Far Cry games.
Our target budget was around 2ms GPU for full screen terrain.
A large part of this presentation will be focussed on the performance tricks that allowed us to do this.



Not in this Presentation

- Terrain Tools and Procedural Generation
- Terrain Asset Packing and Streaming
- Terrain Physics



Contents

- Heightfield Rendering
  - Fundamentals
  - GPU Pipeline
- Shading
- Cliff Shading
- Beyond the Heightfield
- Screen Space Shading
- Terrain Based Effects

What I am covering will be divided into 6 sections.

First I'll discuss how we render the terrain heightfield geometry.
I'll start with the classic approaches that you may be familiar with.
And then I'll discuss how we ported parts of this to a GPU pipeline.

Second I'll discuss how we shade the resulting terrain mesh.

Then I'll discuss shading specializations and optimizations that we made for cliffs.

After that I'll discuss how we combined our base heightfield with additional unique geometry.

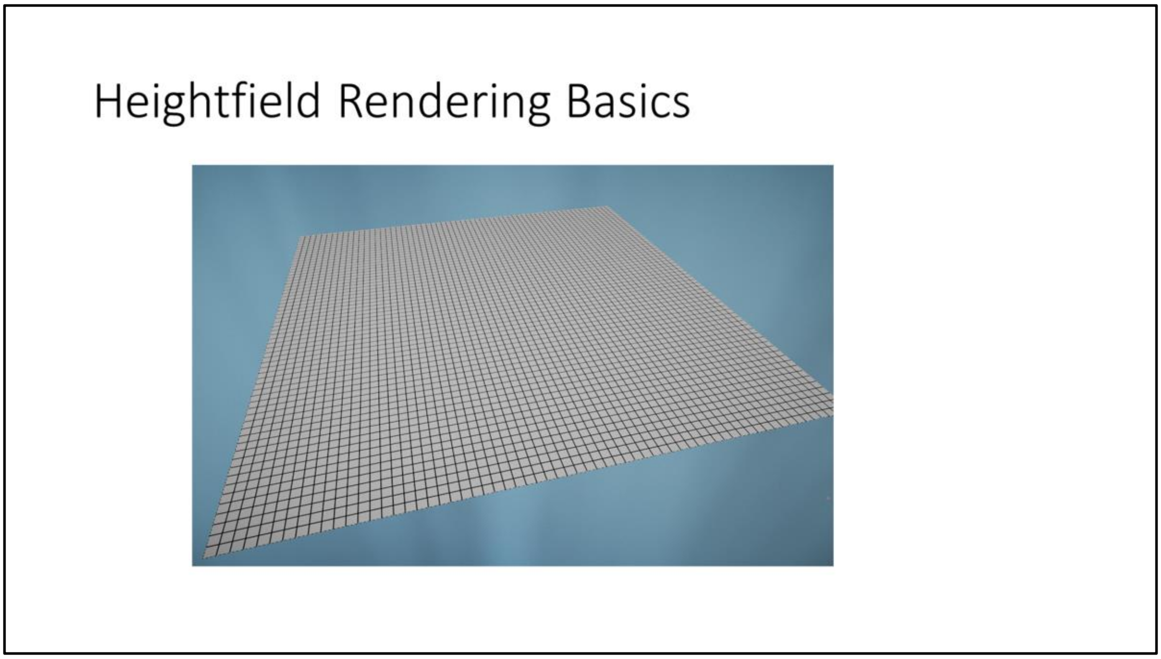
This will lead on to covering how we shade all our terrain geometry inputs in a single screen space pass.

Finally I'll talk about how we used the terrain data on the GPU to enhance the rendering of other assets such as trees, grass and rocks.

GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Heightfield Rendering

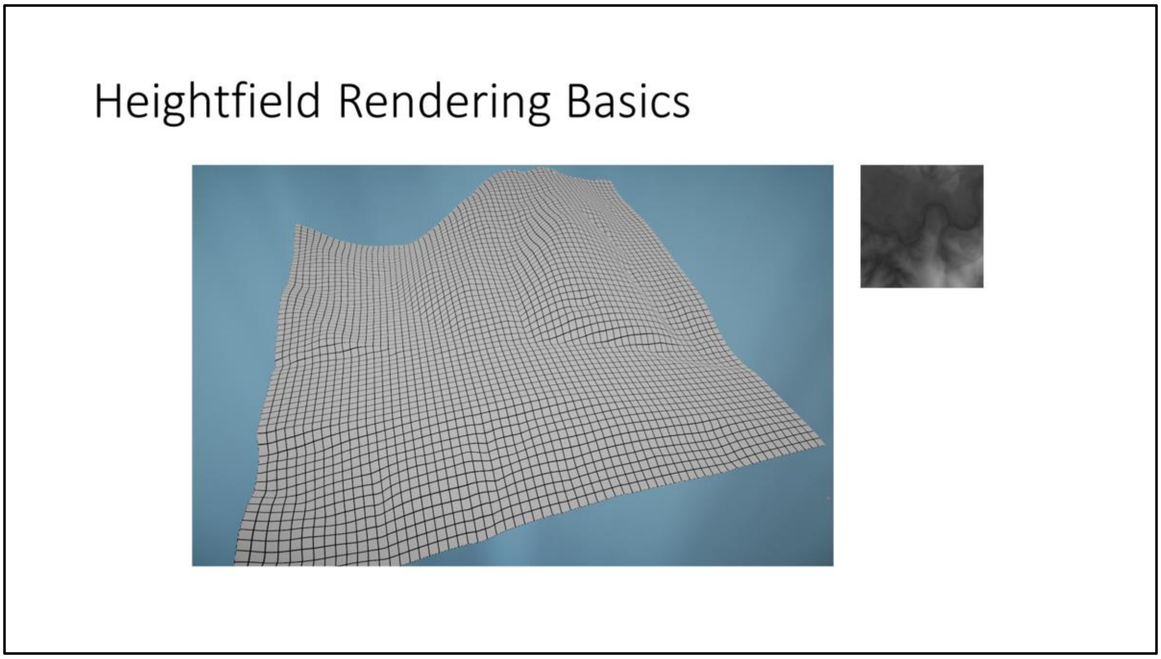
So first we’ll cover the basics of how we store and render terrain.
The introduction will be familiar if you have worked with any terrain rendering system.
But it lays the ground work for the GPU data structures and rendering that we will see later in this section.



Heightfield Rendering Basics

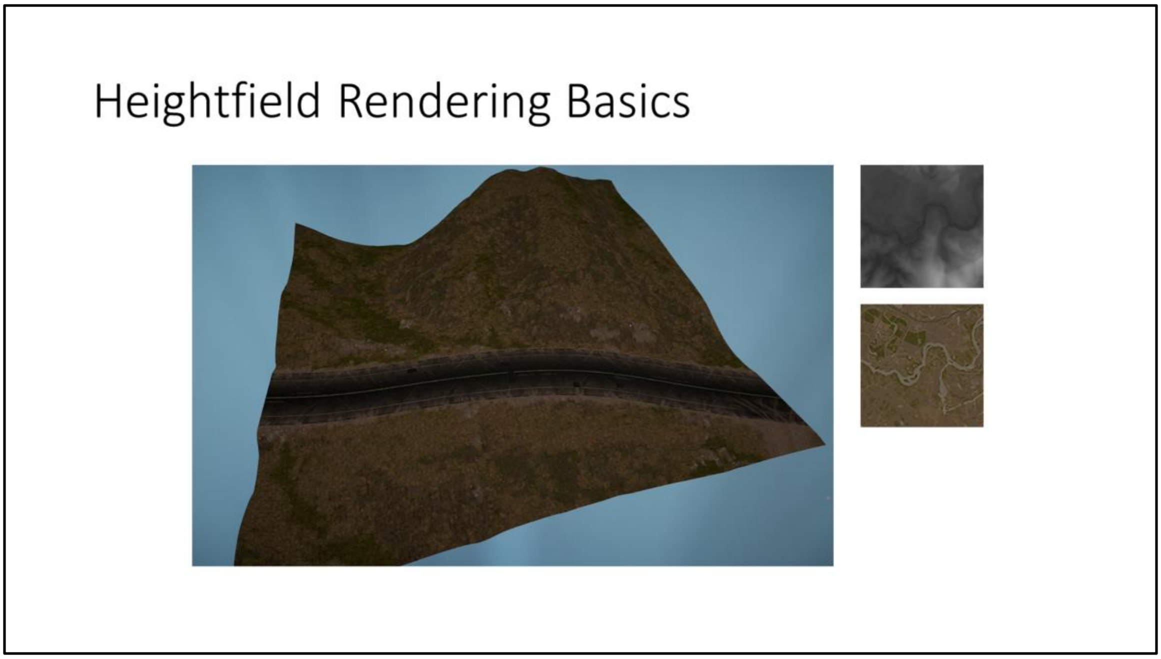
A simple GPU friendly way to render a small area of terrain is with a heightfield using the following steps:

First render a grid patch of geometry.



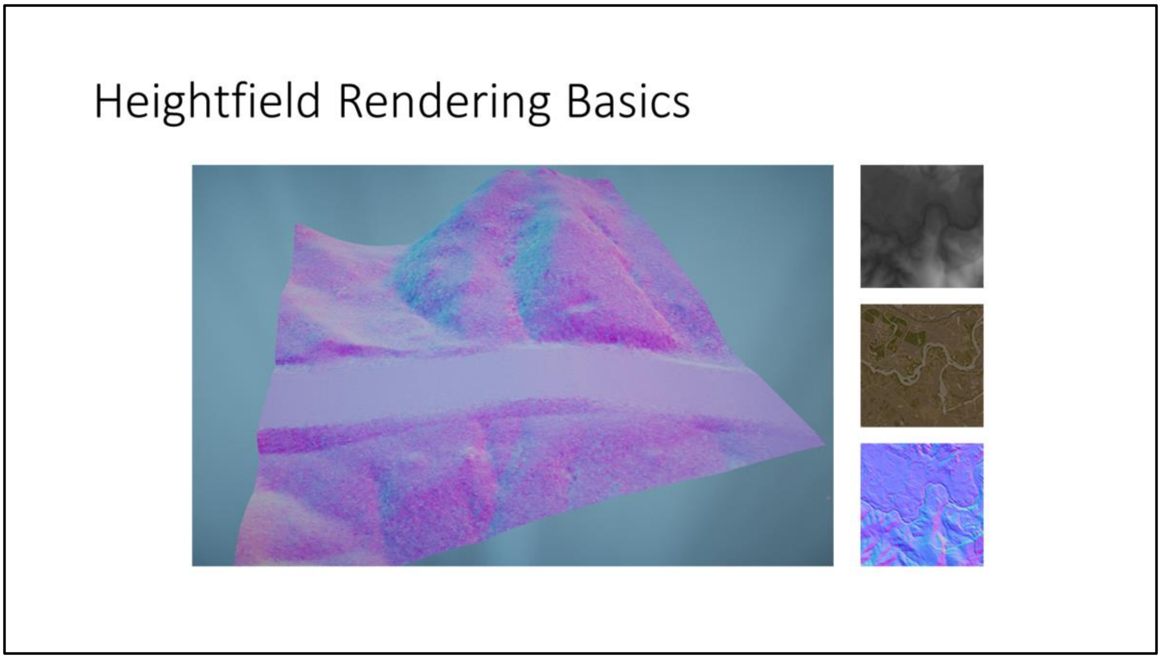
Heightfield Rendering Basics

Displace each of the vertices in the vertex shader with a height map texture.



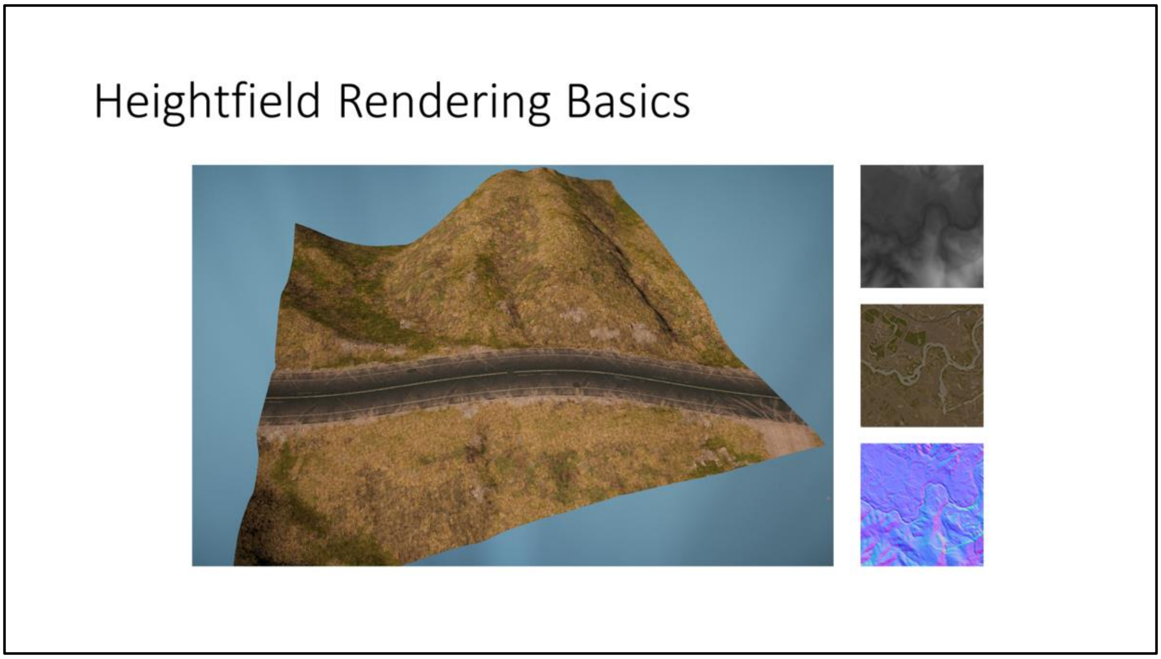
Heightfield Rendering Basics

Apply base color in the pixel shader using an albedo texture.



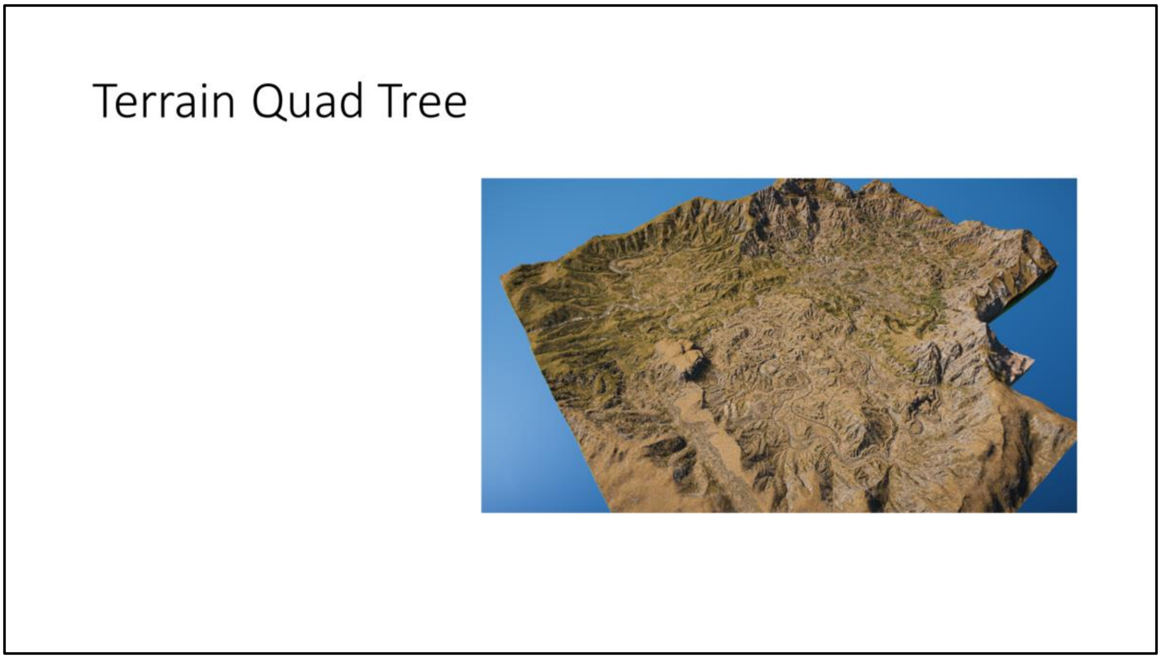
Heightfield Rendering Basics

Light in the pixel shader using a normal map texture with a lighting setup.



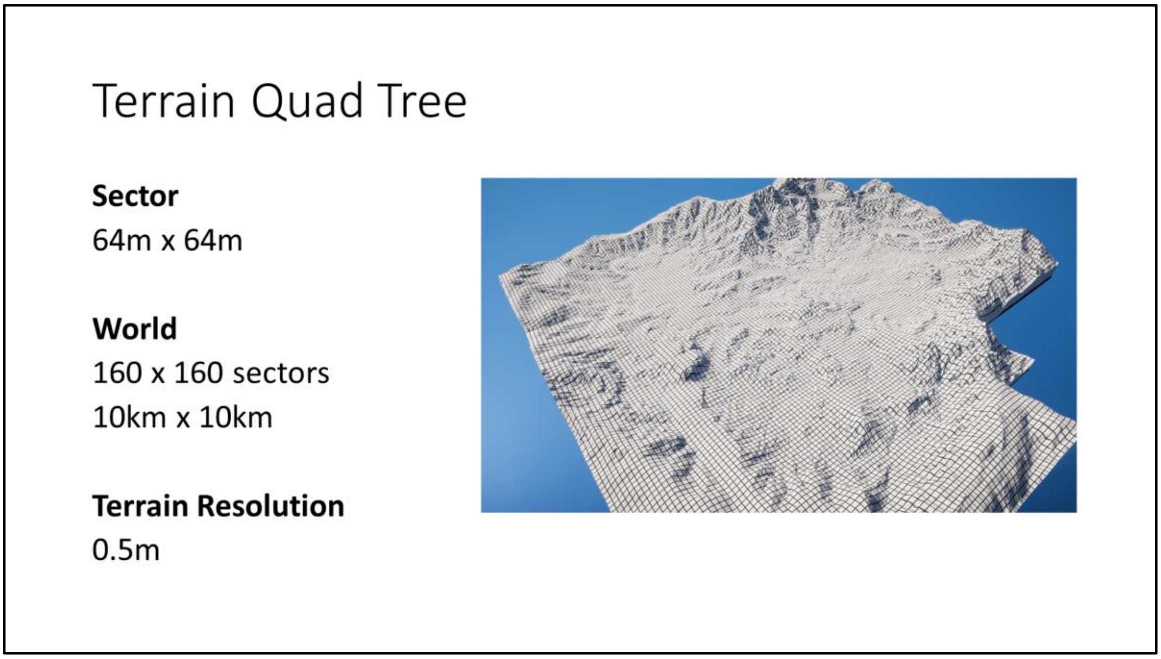
Heightfield Rendering Basics

Light in the pixel shader using a normal map texture with a lighting setup.



Terrain Quad Tree

This is the Far Cry world.



Terrain Quad Tree

Sector
64m x 64m

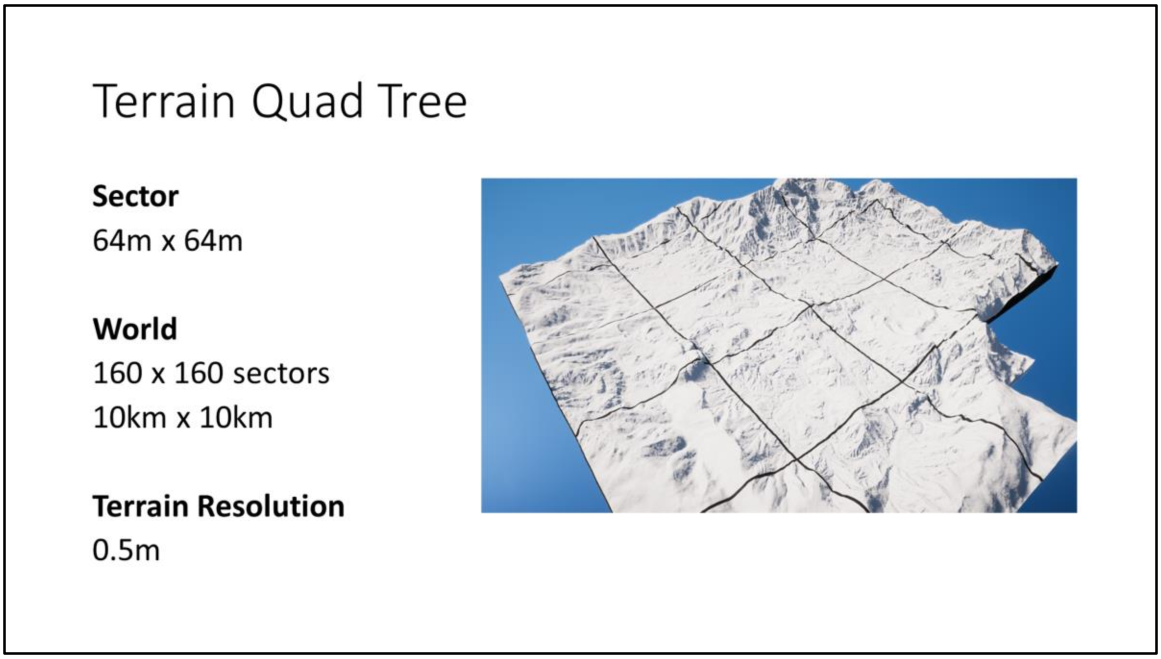
World
160 x 160 sectors
10km x 10km

Terrain Resolution
0.5m

In Far Cry the highest resolution for our terrain tiles is 64 meters square.
In Far Cry we call this size a “sector” and the world is 160x160 sectors.

The Far Cry 5 world is 10km x 10km in size.

We author terrain information at half meter resolution.



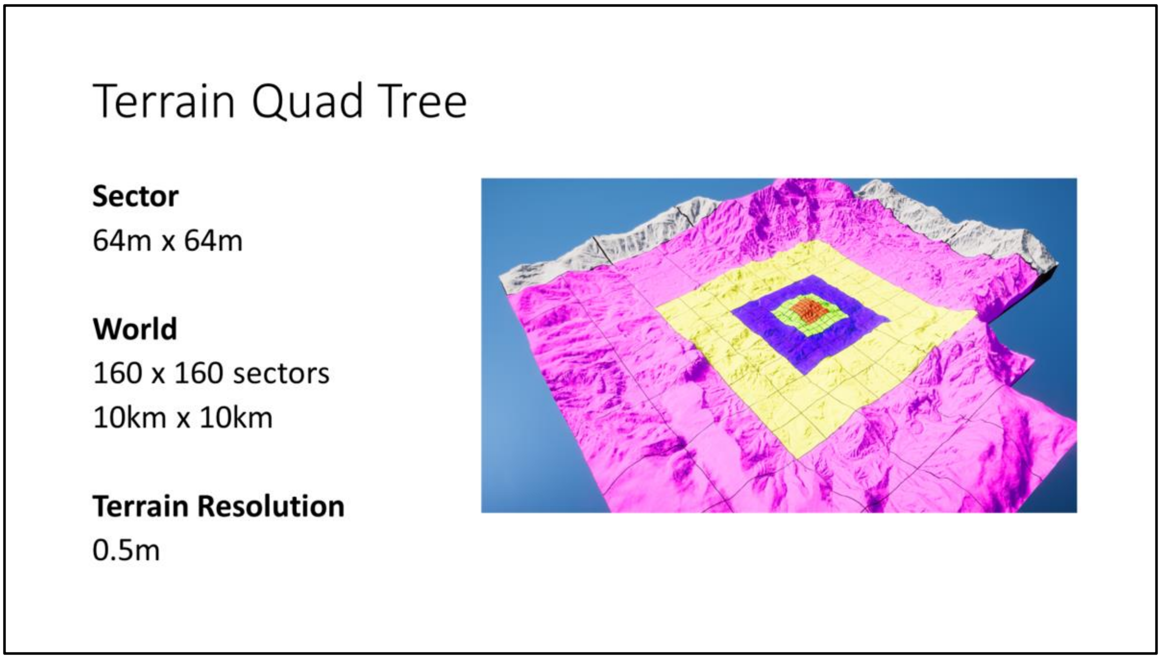
Terrain Quad Tree

Sector
64m x 64m

World
160 x 160 sectors
10km x 10km

Terrain Resolution
0.5m

We can’t render the thousands of high resolution tiles you see here so we need to have a level of detail system.
To support that, the terrain data is stored in a quad tree structure.
At the root the world is divided into a set of 2km square tiles which are always resident.



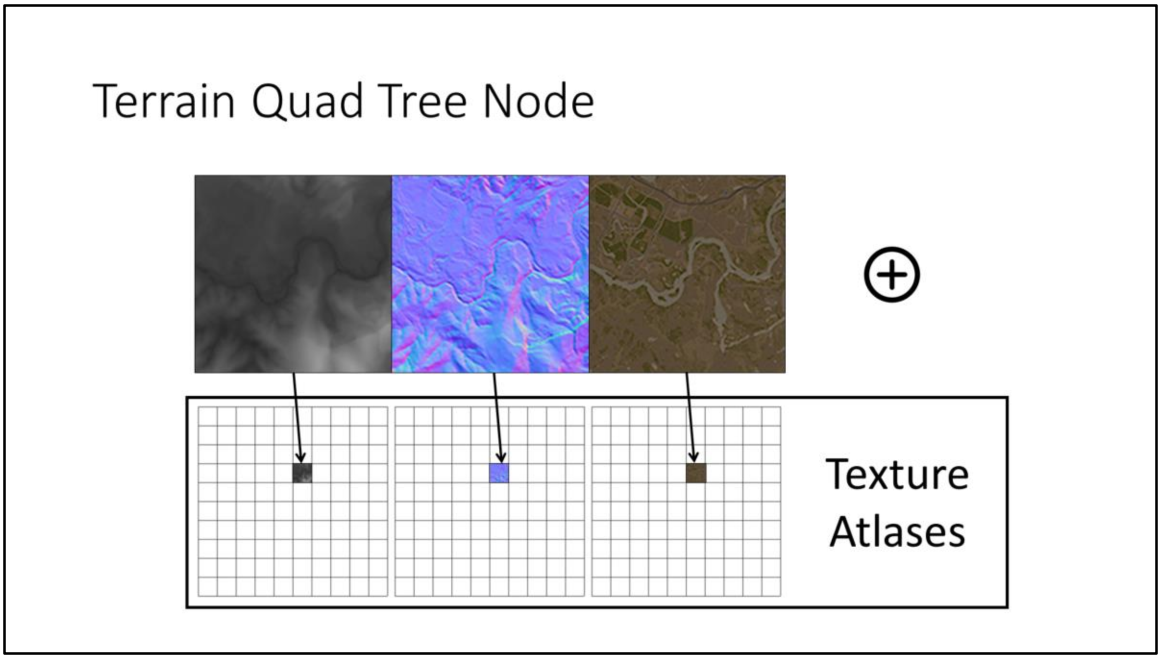
Terrain Quad Tree

Sector
64m x 64m

World
160 x 160 sectors
10km x 10km

Terrain Resolution
0.5m

We subdivide each 2km tile into a quad tree hierarchy of tiles.
We stream tiles according to the LOD and distance from the player.
This means that we have tens of thousands of tiles on disk.
But we only need to allocate memory for, and load, a maximum of about 500 resident tiles.



Terrain Quad Tree Node

⊕

Texture Atlases

So our world is comprised of tens of thousands of these quad tree nodes.

Each quadtree node carries a payload of:

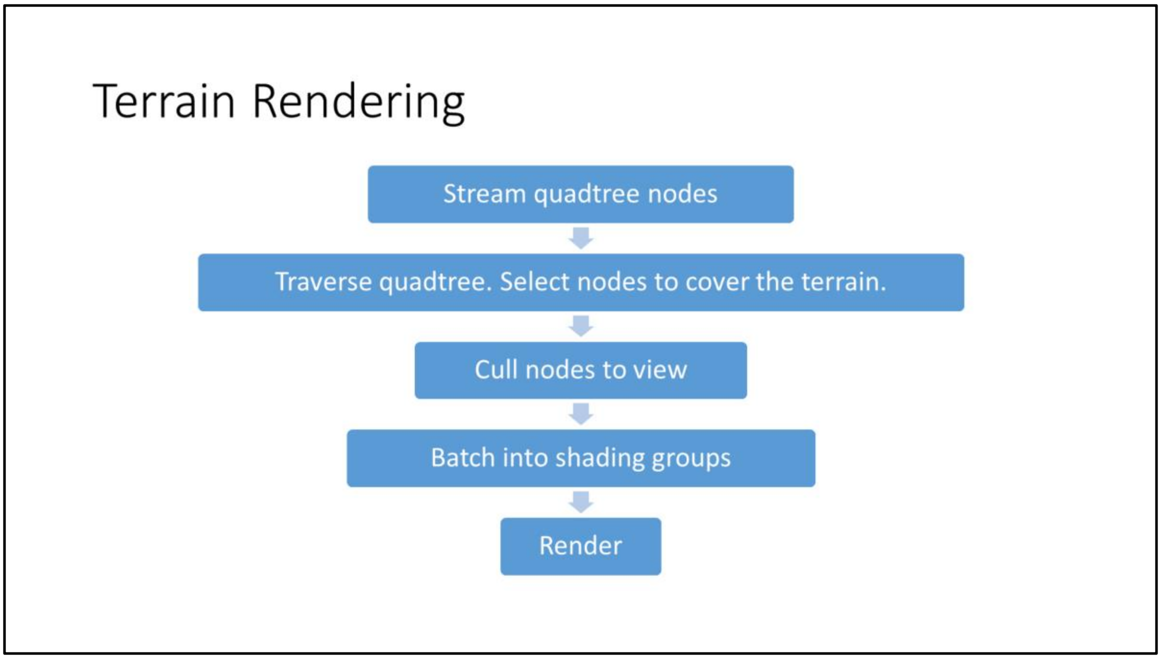
- Heightmap
- World space normal map
- Albedo map

We store some other textures too that we'll see later…

When we load a quad tree node each of these textures is loaded into (synchronized) atlases.
This means that we can reference all of a node's texture data through knowing its location in these atlases.

The texture formats are:
Heightmap (R16_UNORM, 129x129)
World space normal map (we can assume positive z and also pack smoothness and specular occlusion) (BC3, 132x132)
Baked albedo map (BC1, 132x132) (we use 1-bit alpha and store 0 alpha where there

is a hole in the terrain)



Terrain Rendering

Stream quadtree nodes

Traverse quadtree. Select nodes to cover the terrain.

Cull nodes to view

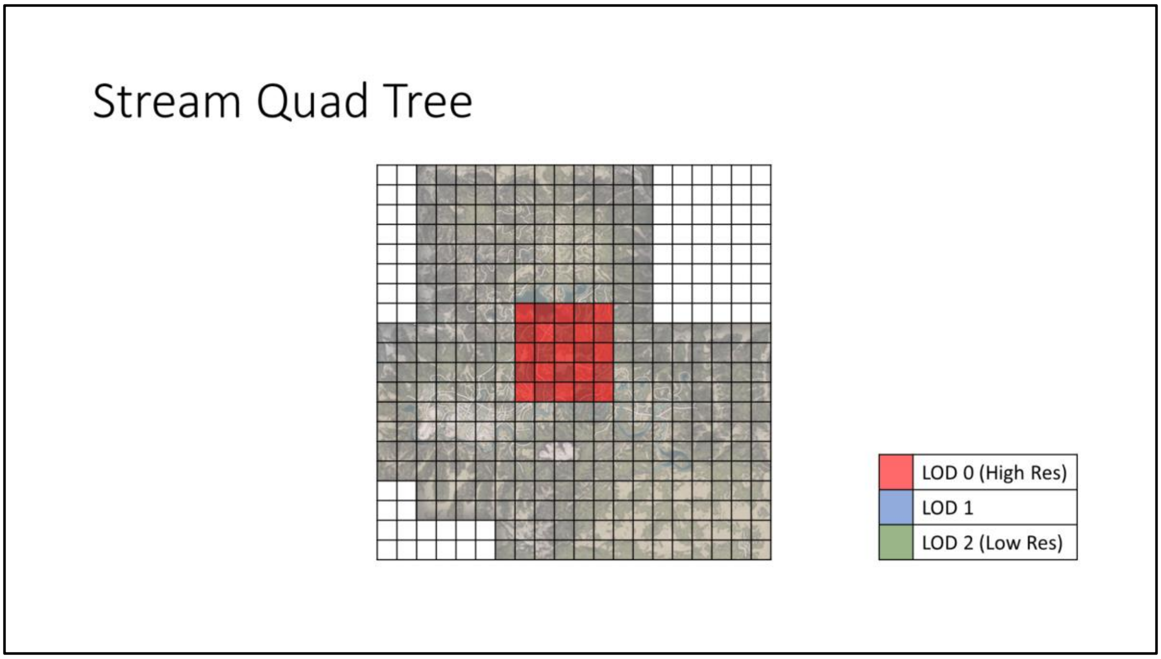
Batch into shading groups

Render

We carry out a number of steps to take the data from disk to displaying terrain on screen

We need to stream in the data.
Then there is a per frame update phase for traversing the quad tree.
And a per view render phase for culling, and batching, before submitting groups of terrain patches to render.

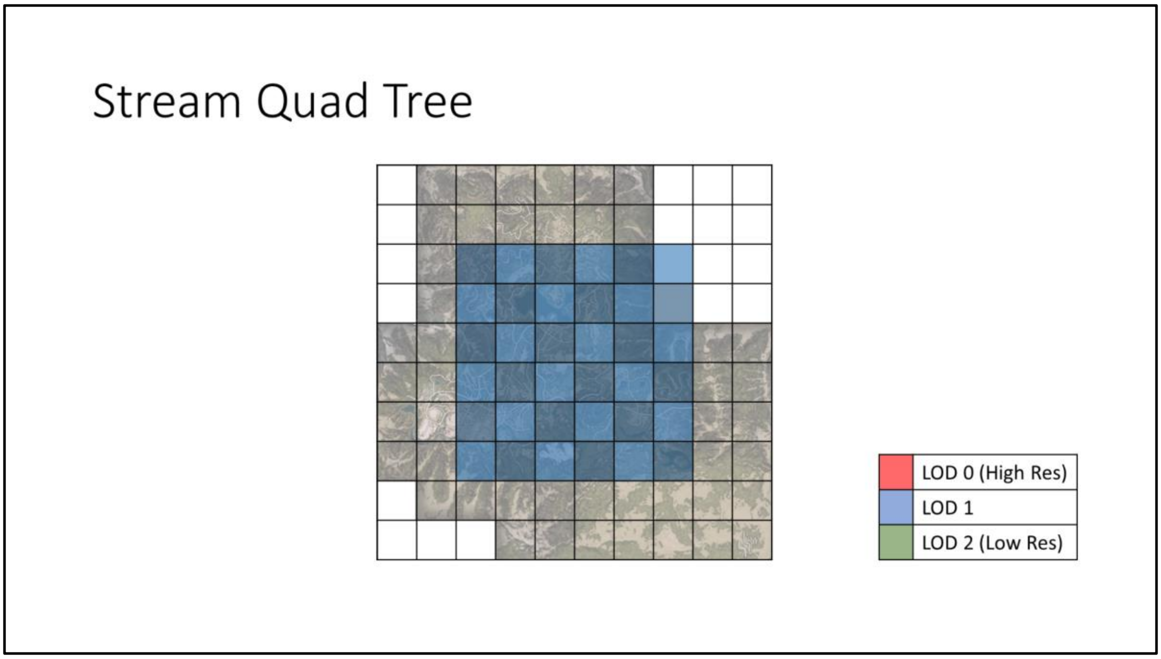
I’ll give a quick overview for each of these steps.



Stream Quad Tree

🟥	LOD 0 (High Res)
🟦	LOD 1
🟩	LOD 2 (Low Res)

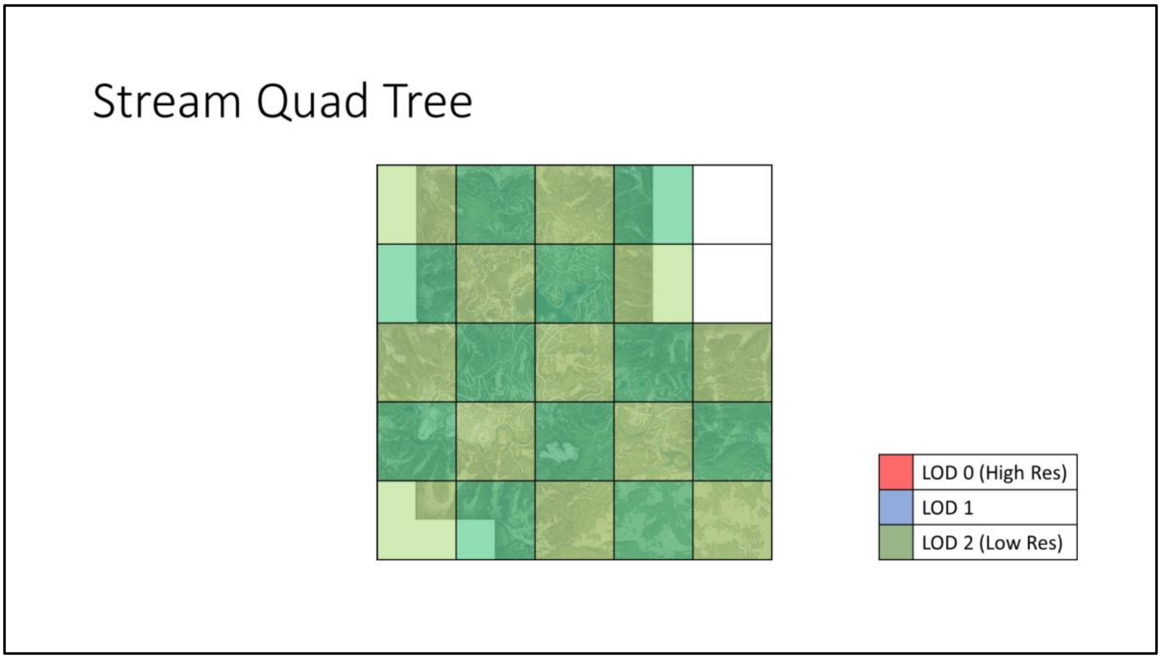
Our general streaming strategy is to load:
First a ring of the highest LOD around the player



Stream Quad Tree

🟥	LOD 0 (High Res)
🟦	LOD 1
🟩	LOD 2 (Low Res)

A ring of the next LOD around the player that will cover roughly double the world distance.

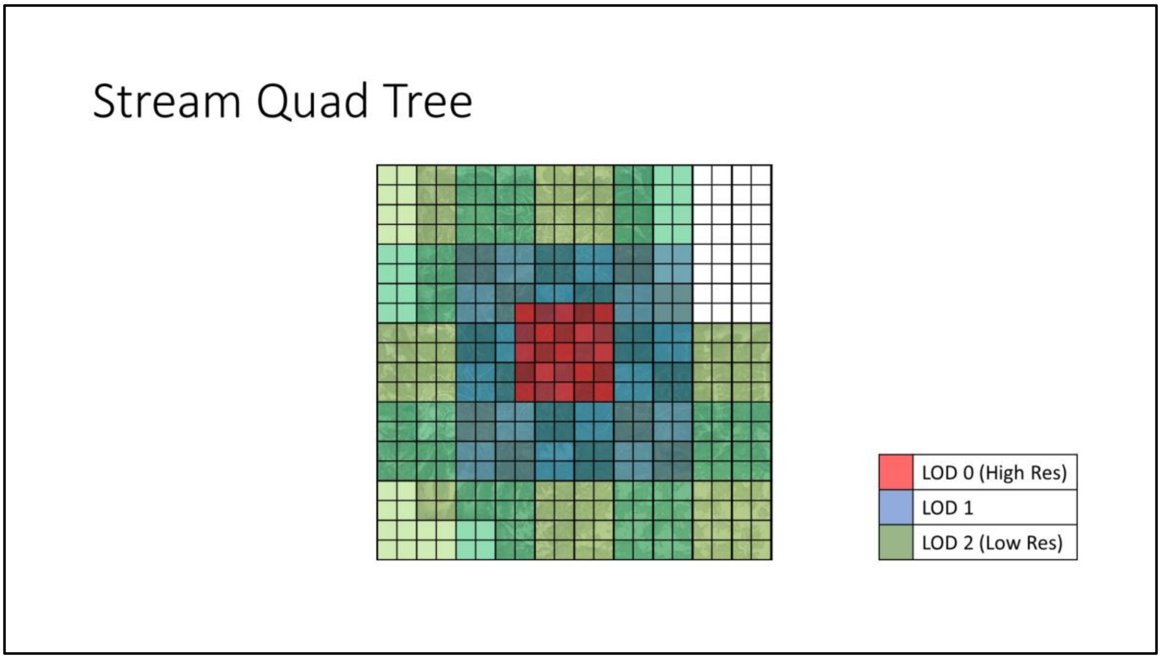


Stream Quad Tree

🟥	LOD 0 (High Res)
🟦	LOD 1
🟩	LOD 2 (Low Res)

And so on…
Until at the lowest LOD we load all of the nodes and keep them always resident.

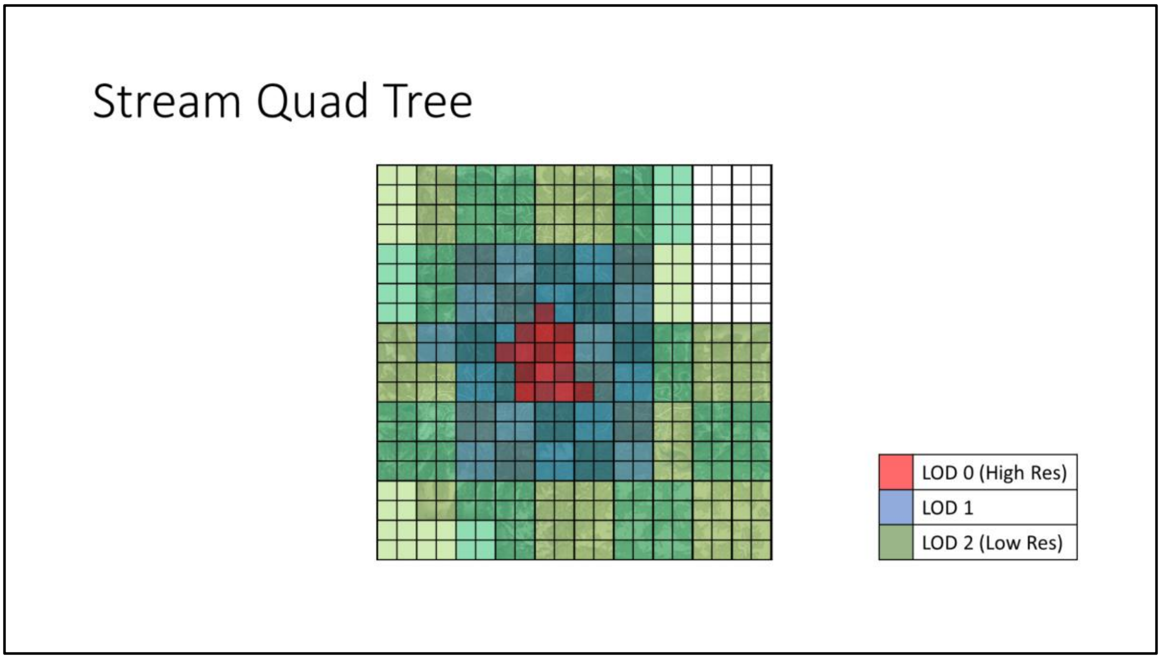
In this and the subsequent diagrams, I will only show 3 LODs for simplicity.
But we use 6 levels in the Far Cry 5 quad tree.



Stream Quad Tree

🟥	LOD 0 (High Res)
🟦	LOD 1
🟩	LOD 2 (Low Res)

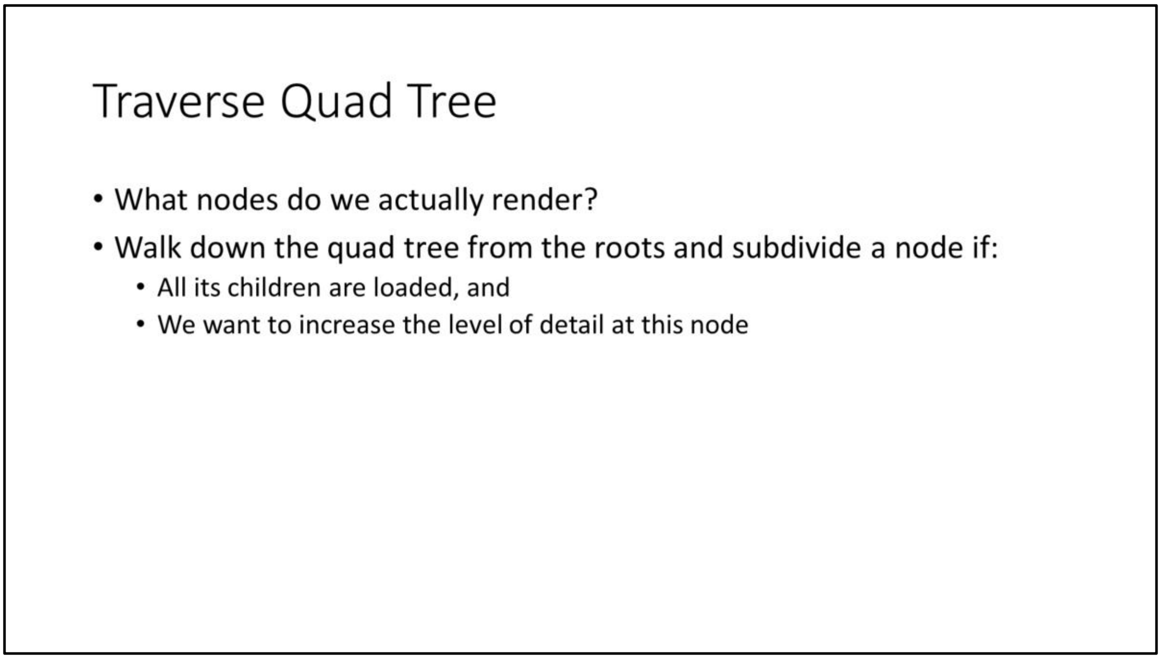
So these are the (overlapping) nodes that we have requested the underlying streaming system to load.
However we can’t make any assumptions about what nodes are actually loaded at any time.
Some streaming requests might be waiting…



Stream Quad Tree

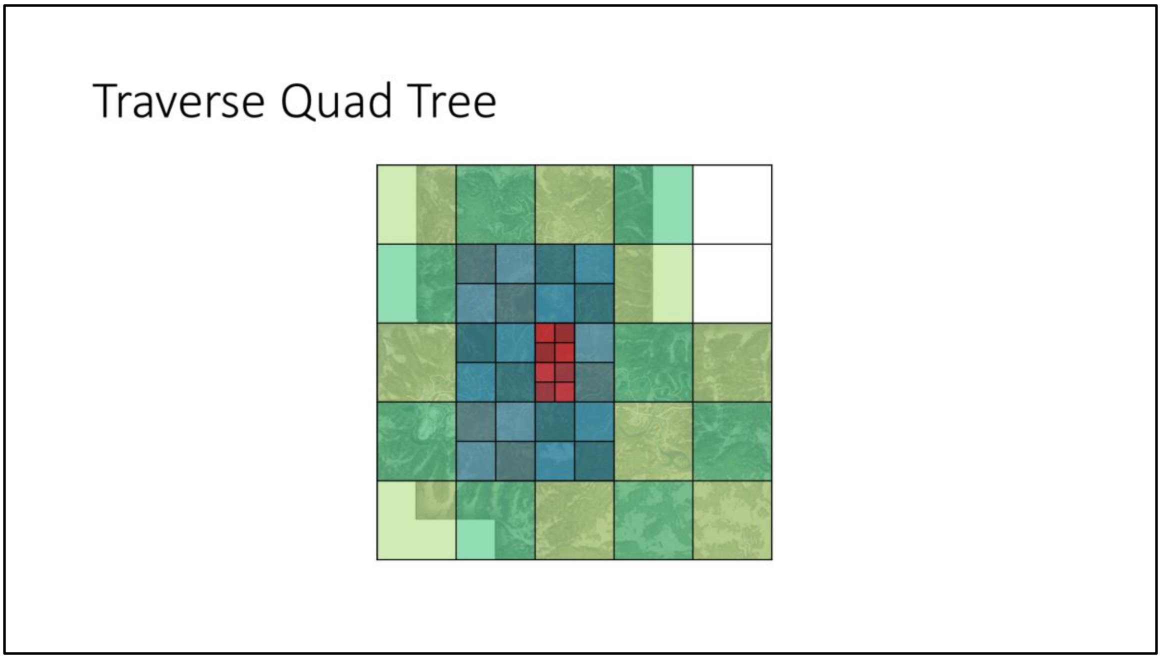
🟥	LOD 0 (High Res)
🟦	LOD 1
🟩	LOD 2 (Low Res)

So our actual state can be different from our requested state.
It's as likely be something like this, or indeed anything else…
We will need to render something valid for any scenario.



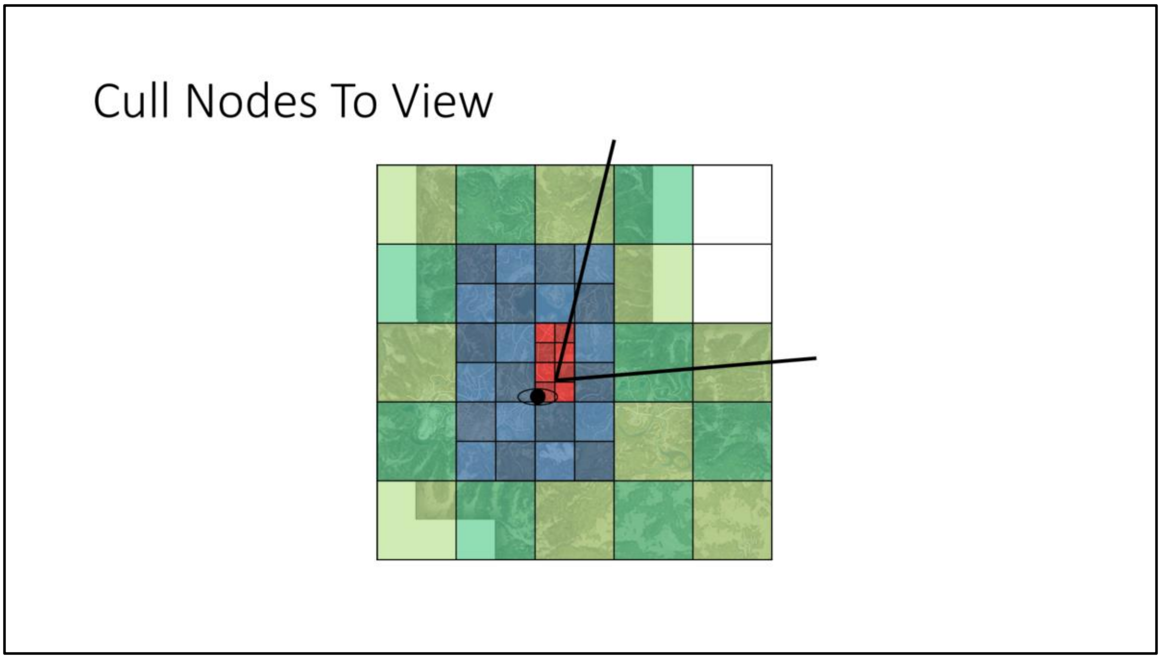
Traverse Quad Tree

- What nodes do we actually render?
- Walk down the quad tree from the roots and subdivide a node if:
  - All its children are loaded, and
  - We want to increase the level of detail at this node



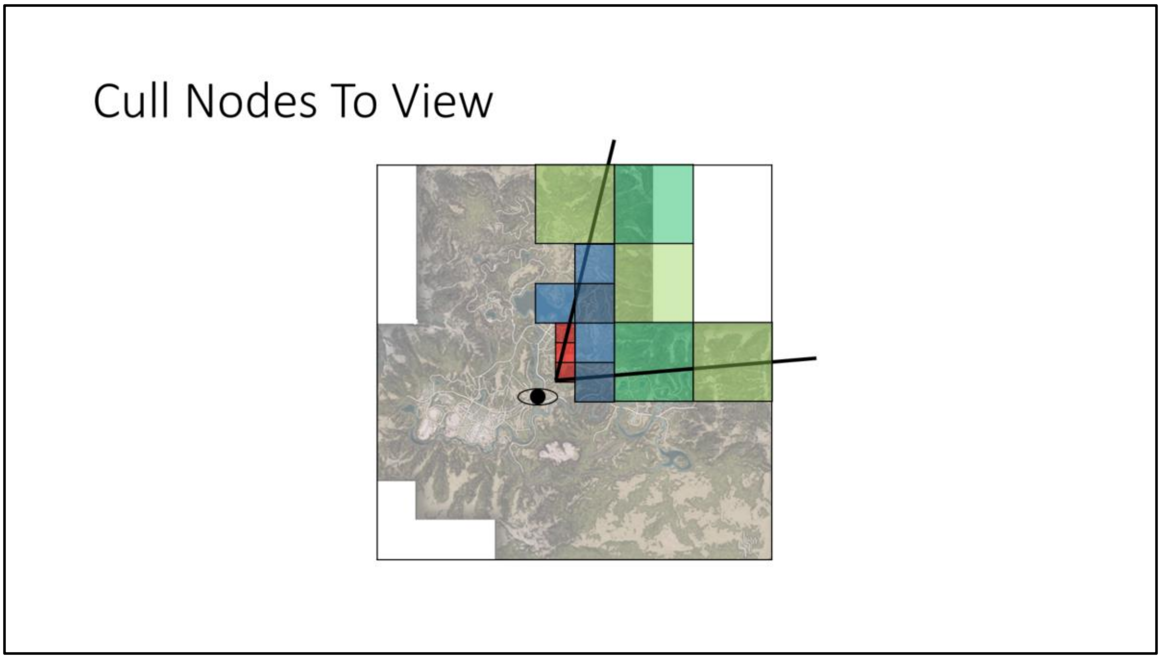
Traverse Quad Tree

If we follow these simple rules we are left with a set of nodes that cover the world and don’t overlap.



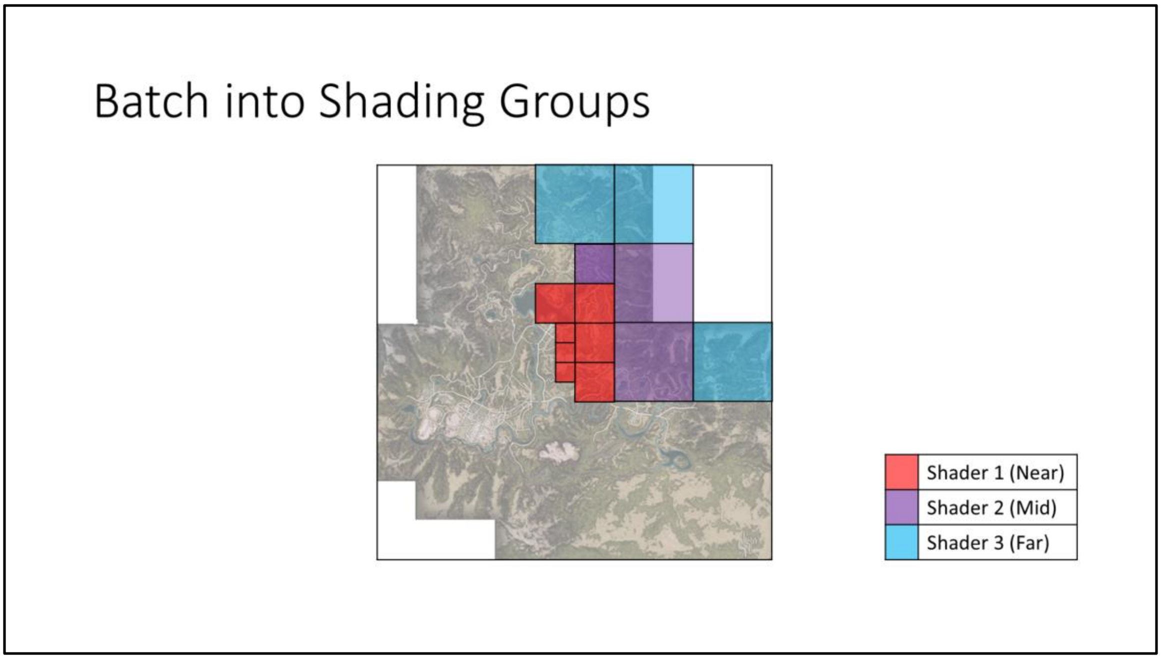
Cull Nodes To View

Also we only want to render the nodes visible from the camera.



Cull Nodes To View

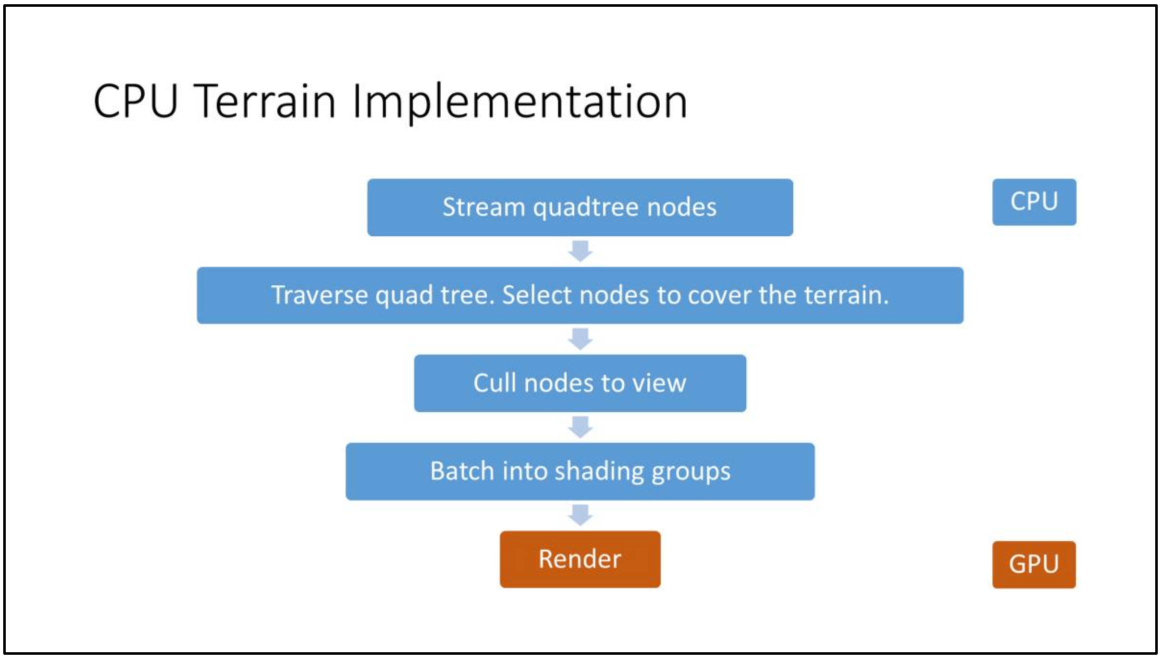
These are the nodes left after culling to the camera.



Batch into Shading Groups

🟥 Shader 1 (Near)
🟪 Shader 2 (Mid)
🟦 Shader 3 (Far)

Finally if we use different shaders we would like to batch our nodes into shading groups.
For example maybe we have different shaders for near and far terrain.
All the nodes in a group can be sent in a single batch to be rendered (even if they are in different LODs).



CPU Terrain Implementation

Stream quadtree nodes

Traverse quad tree. Select nodes to cover the terrain.

Cull nodes to view

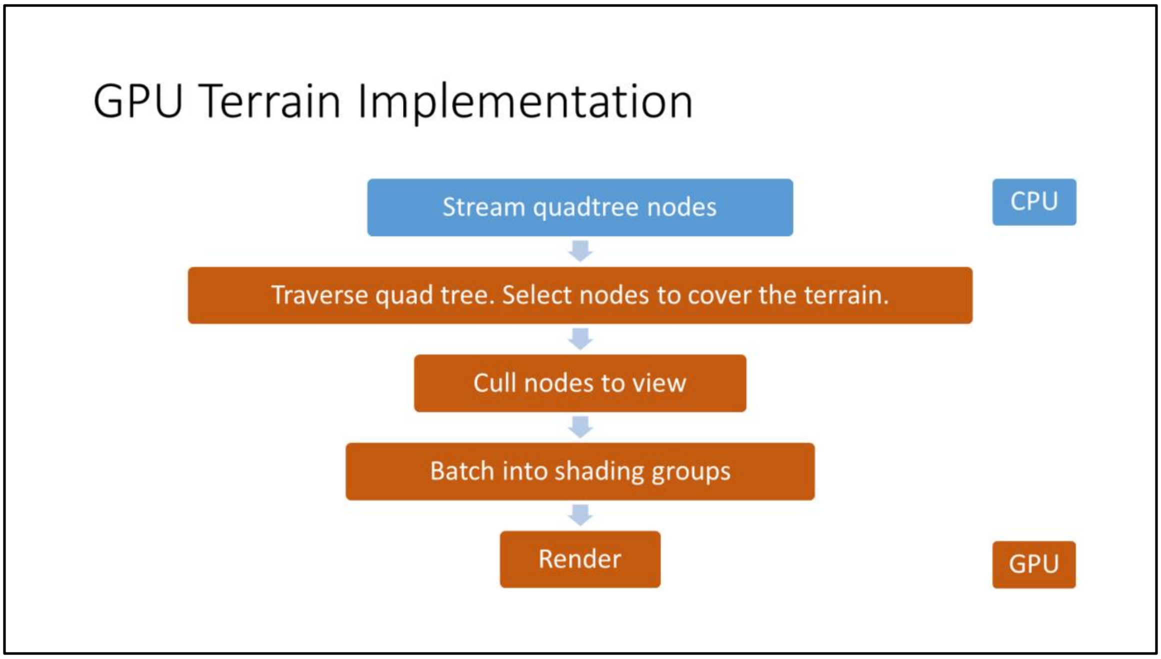
Batch into shading groups

Render

CPU

GPU

Initially we implemented this approach with most parts of the system running on the CPU.
Only the final rendering (in red) made use of the GPU.



GPU Terrain Implementation

Stream quadtree nodes

Traverse quad tree. Select nodes to cover the terrain.

Cull nodes to view

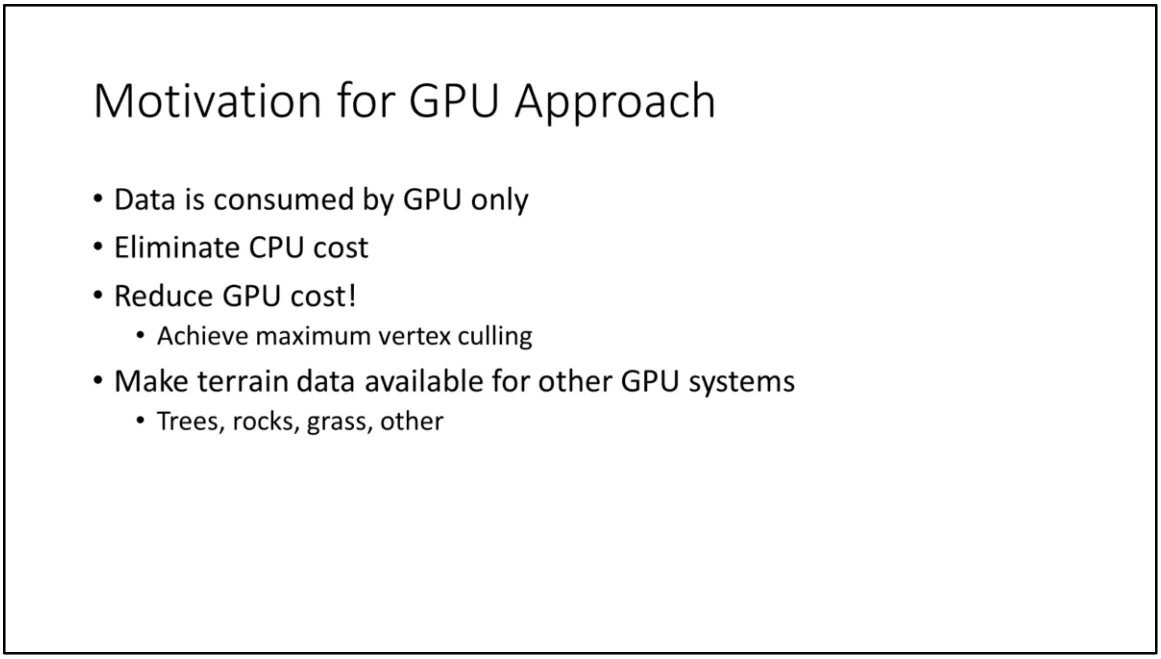
Batch into shading groups

Render

CPU

GPU

But we realized that we could move almost all of the work in this diagram to the GPU. Only the initial streaming logic needs to be handled by the CPU.



Motivation for GPU Approach

- Data is consumed by GPU only
- Eliminate CPU cost
- Reduce GPU cost!
  - Achieve maximum vertex culling
- Make terrain data available for other GPU systems
  - Trees, rocks, grass, other

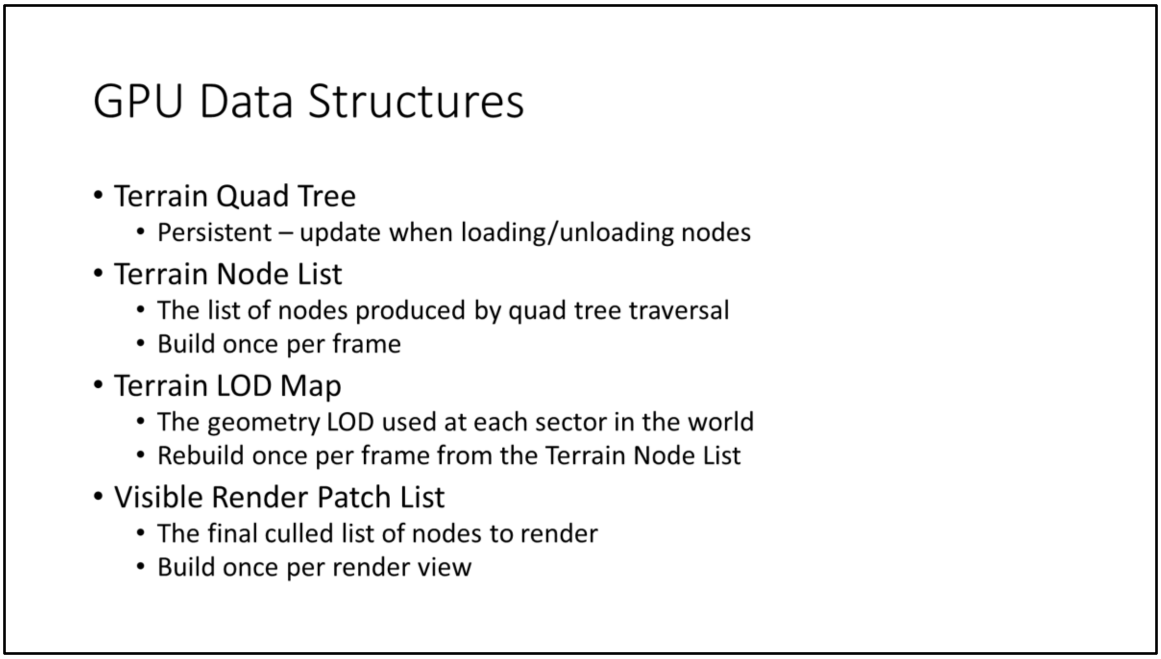
Moving the work to GPU makes sense because the data is only for GPU consumption.

Of course it means that we save CPU time.

But also we reduce GPU time!
We can use data available on the GPU to improve LOD selection and culling.
This can improve GPU terrain performance which can easily become vertex bound.
This means that the technique pays for itself.

Finally having more terrain info in GPU data structures has the advantage that other shaders can access it.

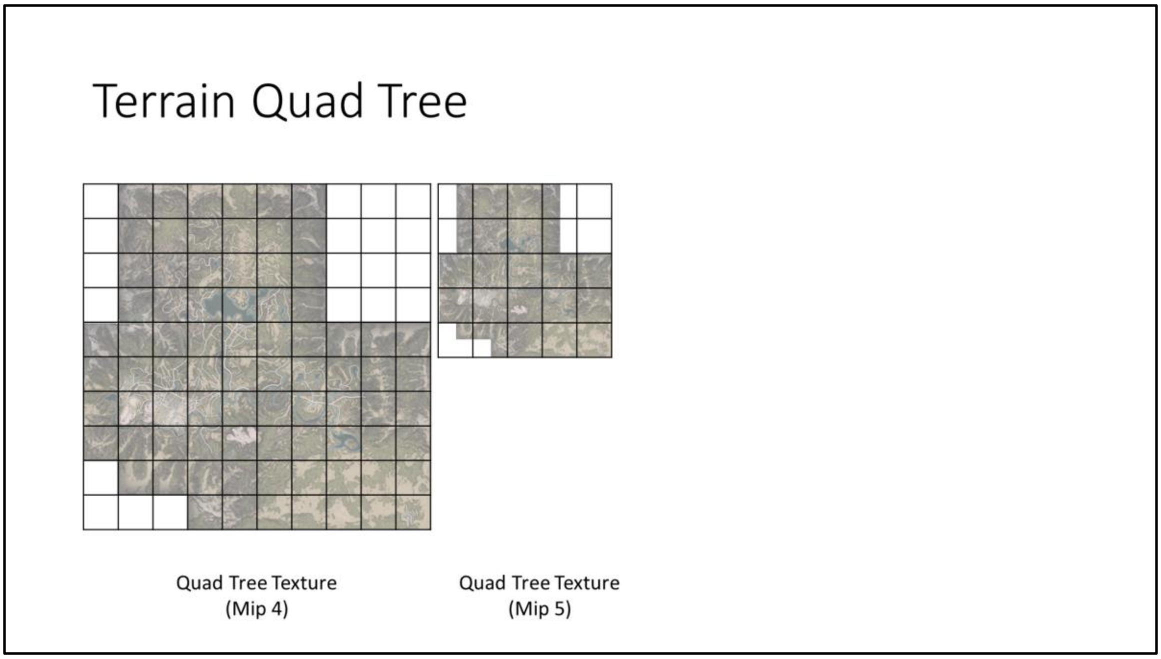
Note that we have a separate CPU data structure for gameplay terrain height and material queries.



GPU Data Structures

- Terrain Quad Tree
  - Persistent – update when loading/unloading nodes
- Terrain Node List
  - The list of nodes produced by quad tree traversal
  - Build once per frame
- Terrain LOD Map
  - The geometry LOD used at each sector in the world
  - Rebuild once per frame from the Terrain Node List
- Visible Render Patch List
  - The final culled list of nodes to render
  - Build once per render view

We will need several data structures on the GPU to store and process our quad tree. The structures need to be in a layout that can be efficiently accessed and transformed by GPU parallel algorithms.
I'll discuss 4 main data structures and how we read and write them.



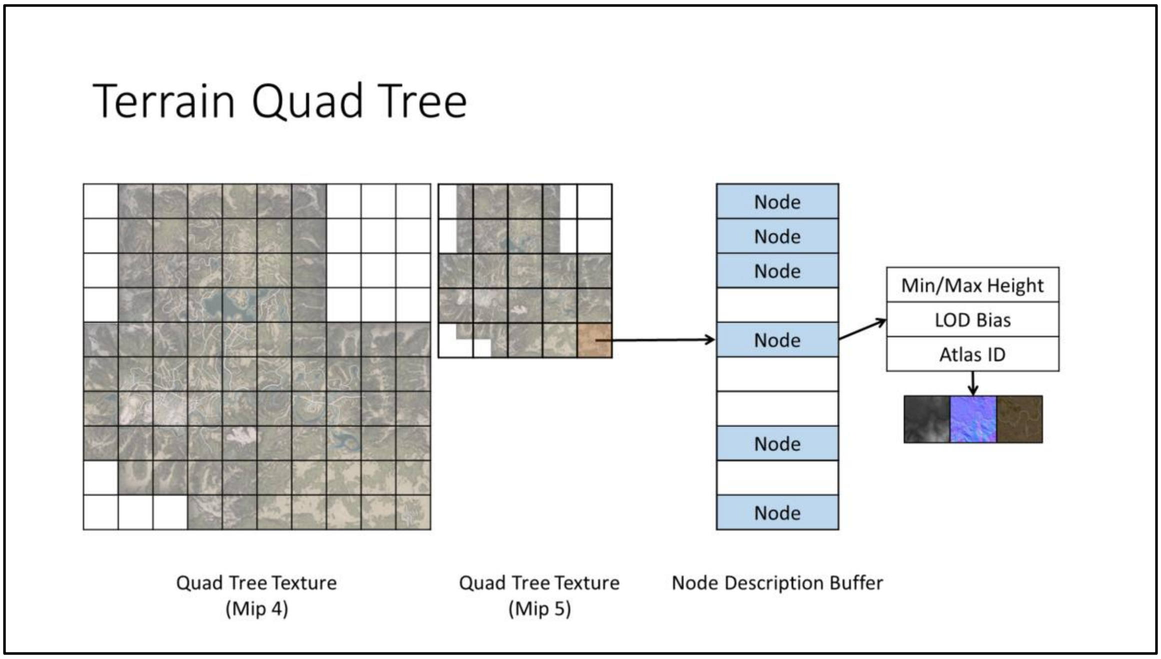
Terrain Quad Tree

Quad Tree Texture
(Mip 4)

Quad Tree Texture
(Mip 5)

The terrain quad tree is implemented using two linked GPU objects.

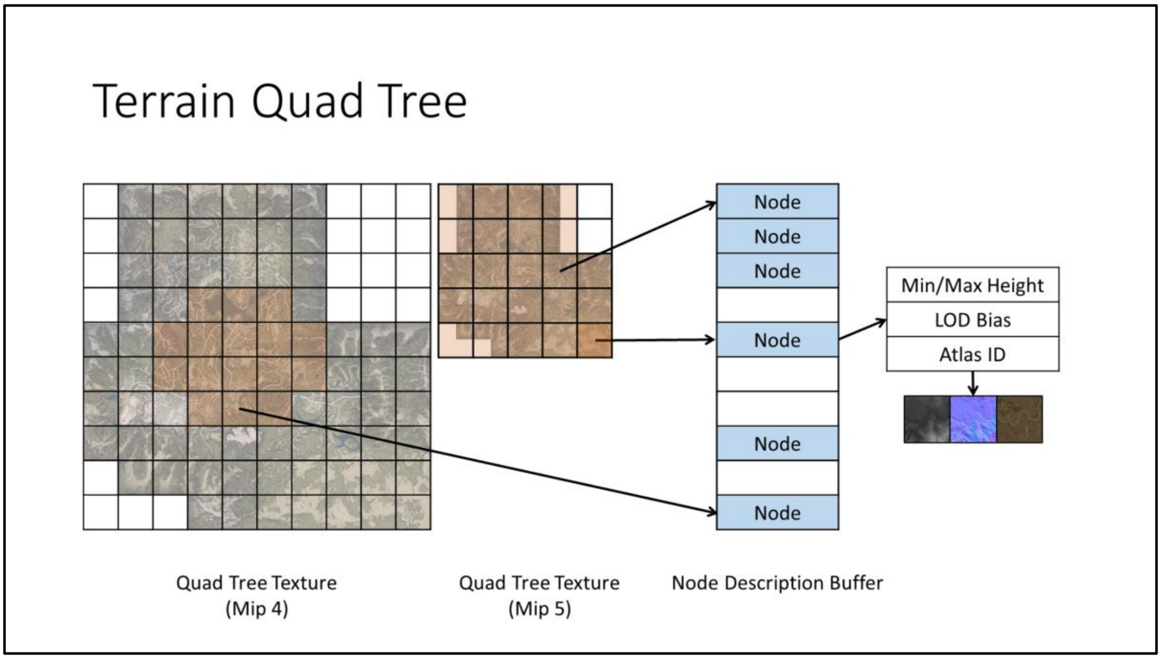
First we have a mip-mapped texture.
It is 160x160 texels in size at Mip0 and then has one mip level for each of the 6 quad tree levels.
So each node in our quad tree is represented by a single texel in this texture.



Terrain Quad Tree

What do we store in this texture?
It’s a 16 bit index into our second GPU object.
This is a Node Description Buffer which is simply an array.

It contains packed description for our loaded nodes.
This includes the minimum and maximum terrain height, and the location of the node’s textures in our texture atlases.



Terrain Quad Tree

Node
Node
Node
Node
Node
Node

Min/Max Height
LOD Bias
Atlas ID

Quad Tree Texture
(Mip 4)

Quad Tree Texture
(Mip 5)

Node Description Buffer

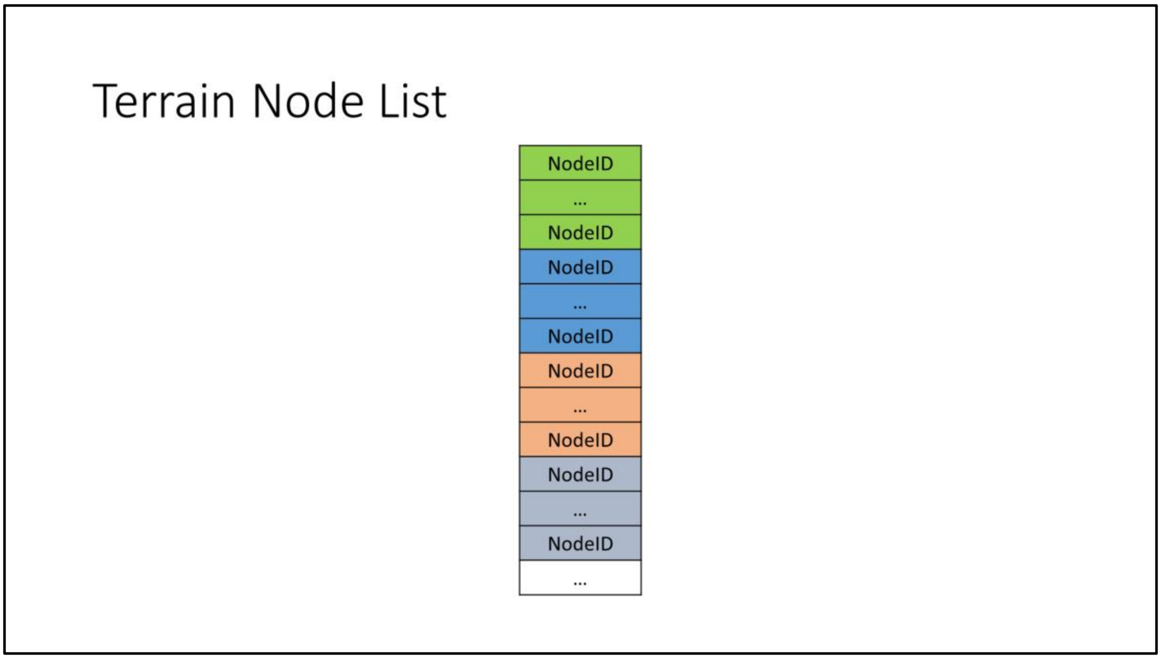
Whenever we load a node we load its node description into an available slot.
We dispatch a compute shader every frame to add and remove nodes as they are streamed in and out.

The texture here is a R16_UINT texture.
The node data is packed in 2 uints.
We keep a shadow copy of the node data on the CPU to track node allocation.

Note that we have 2 special values in the texture:

- The empty value 0xffff (means nothing is loaded for this quad tree node)
- The invalid value 0xfffe (means nothing can ever be loaded – it doesn't exist)

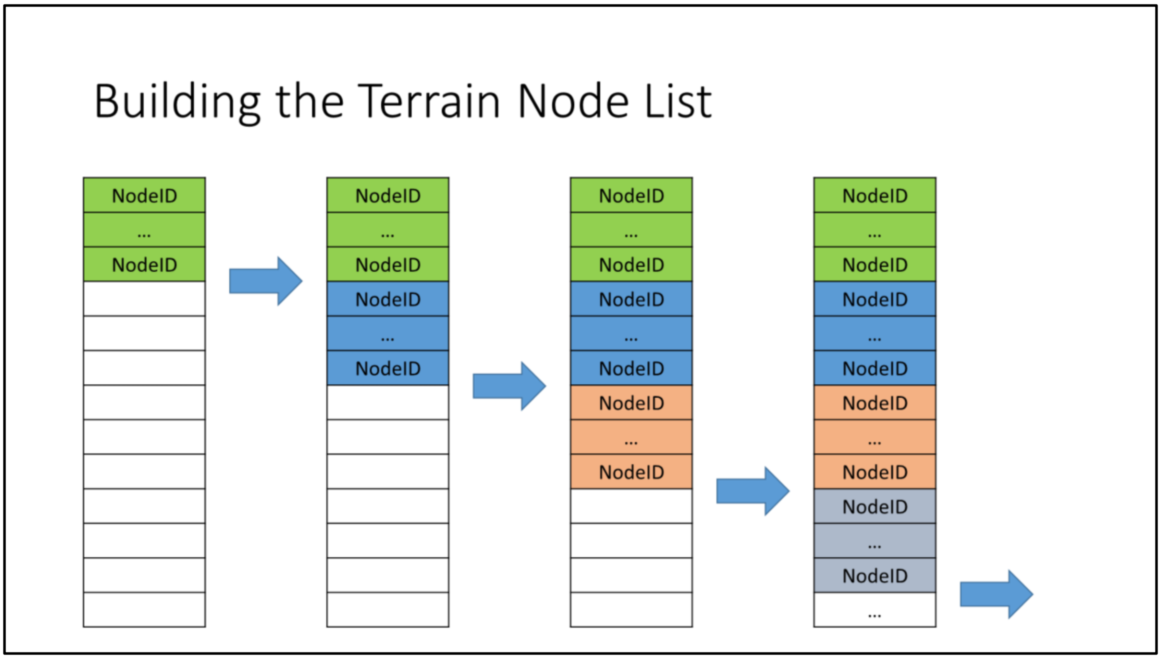
These values allow us to traverse the quad tree correctly (ignoring invalid nodes etc.)



Terrain Node List

NodeID
…
NodeID
NodeID
…
NodeID
NodeID
…
NodeID
NodeID
…
NodeID
…

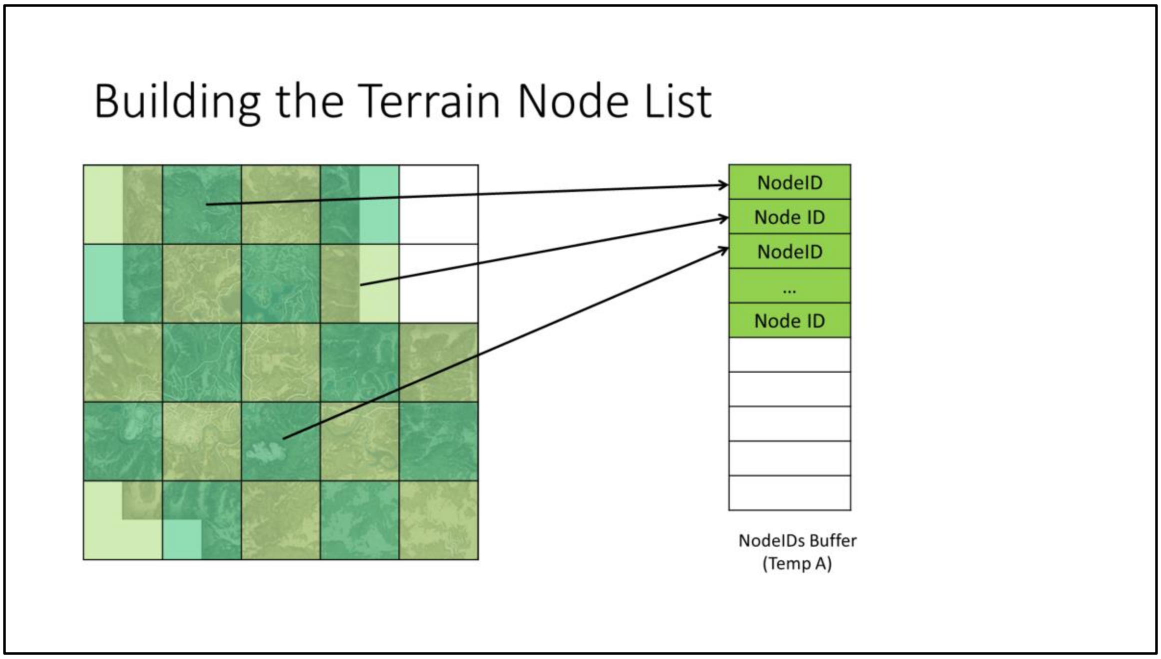
The Terrain Node List is simply the list of nodes that we could potentially render (before any further visibility testing).
It is calculated each frame by traversing the quad tree structure we have just seen.



Building the Terrain Node List

NodeID	NodeID	NodeID	NodeID
…	…	…	…
NodeID	NodeID	NodeID	NodeID
	NodeID	NodeID	NodeID
	…	…	…
	NodeID	NodeID	NodeID
		NodeID	NodeID
		…	…
		NodeID	NodeID
			NodeID
			…
			NodeID
			…

We fill it using a compute shader that iterates through the LOD levels.
At each stage one thread processes each of the nodes at the current LOD and potentially subdivides it into child nodes.



Building the Terrain Node List

NodeID

Node ID

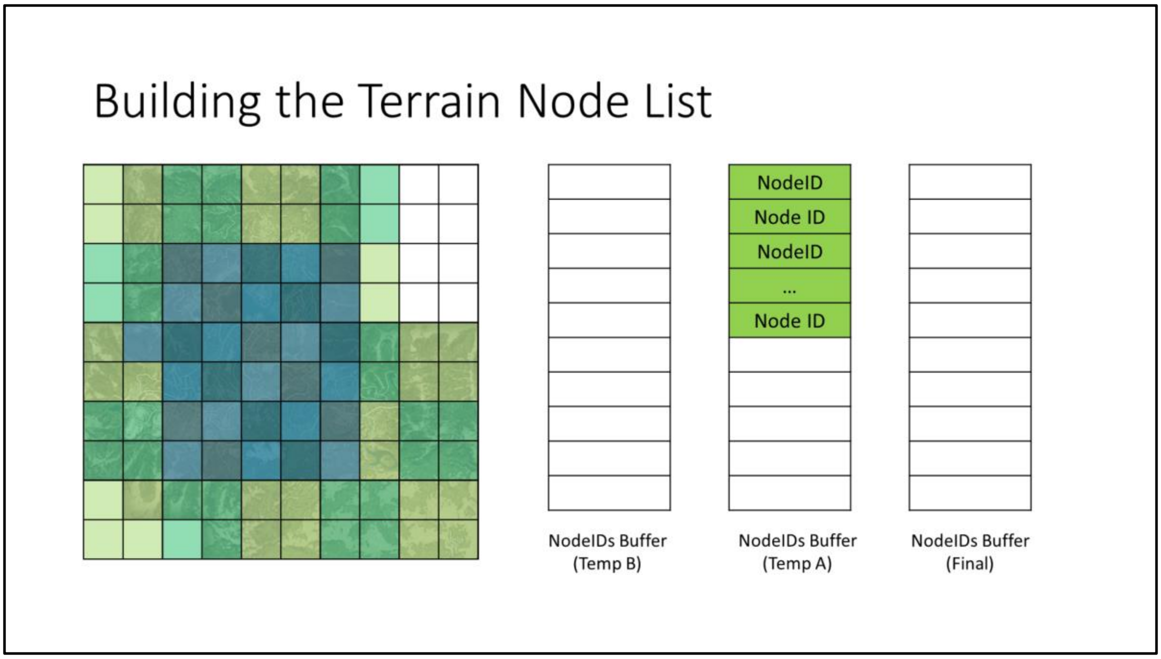
NodeID

…

Node ID

NodeIDs Buffer
(Temp A)

I’ll give you some concrete details on how the processing is done.
We start with the quad tree lowest LOD and fill a temporary buffer (A) with all of the node indexes.



Building the Terrain Node List

NodeID
Node ID
NodeID
…
Node ID

NodeIDs Buffer (Temp B)

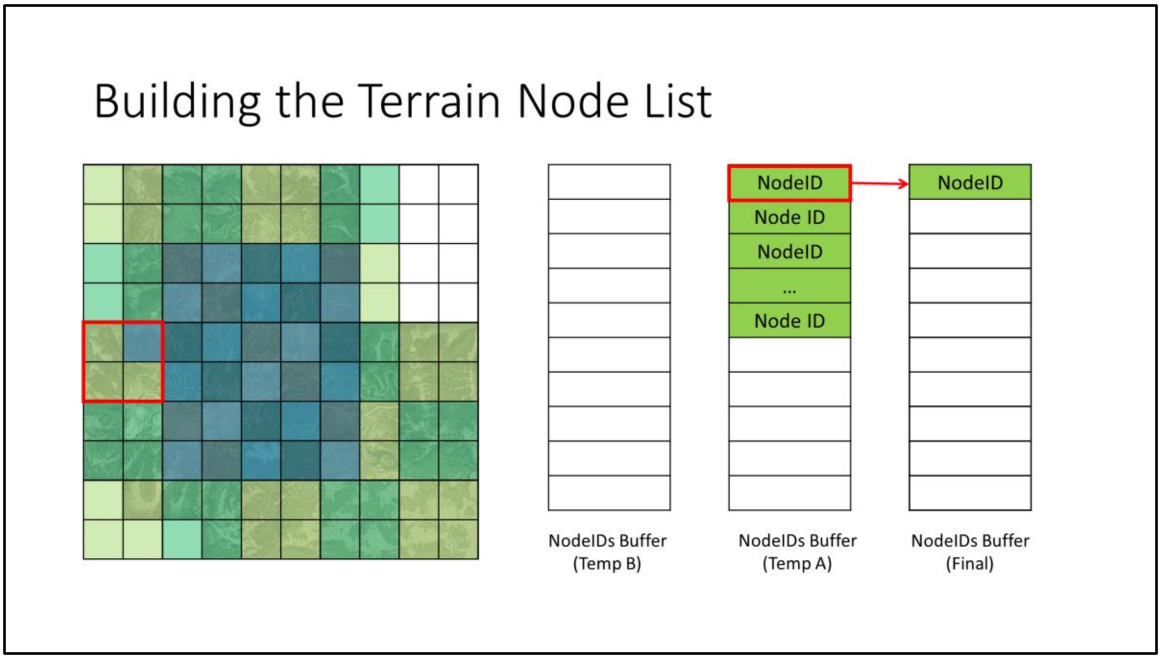
NodeIDs Buffer (Temp A)

NodeIDs Buffer (Final)

Next we want to subdivide these nodes into their children.

We read our temporary buffer A.
And will output two buffers: a second temporary buffer (B) and the final result buffer.

For each node in buffer A we will see if it can be divided into children.
If it can then we append the children in buffer B.
If it can’t then it will be appended to our final buffer for rendering.



Building the Terrain Node List

NodeID

Node ID

NodeID

...

Node ID

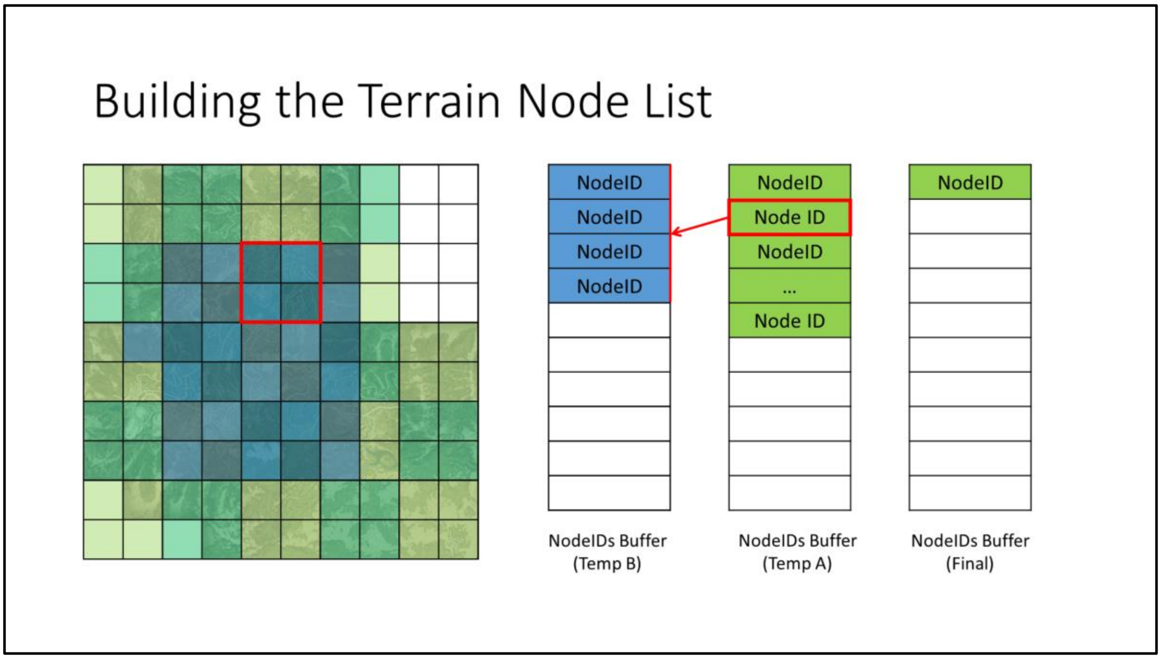
NodeID

NodeIDs Buffer
(Temp B)

NodeIDs Buffer
(Temp A)

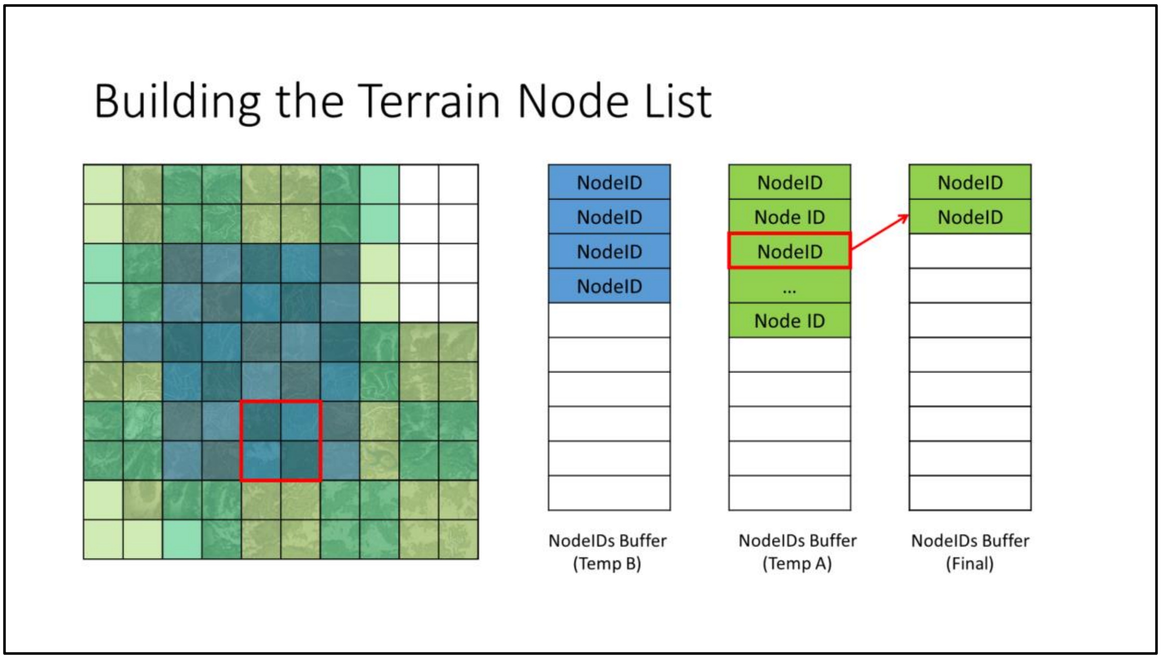
NodeIDs Buffer
(Final)

Running through a concrete example.
This node doesn’t have all 4 children loaded so we append it to the list of final nodes to render.



Building the Terrain Node List

This node has all of its 4 children loaded so we append the child nodes to the temporary buffer B for further processing on the next dispatch.



Building the Terrain Node List

NodeID
NodeID
NodeID
NodeID

NodeIDs Buffer
(Temp B)

NodeID
Node ID
NodeID
...
Node ID

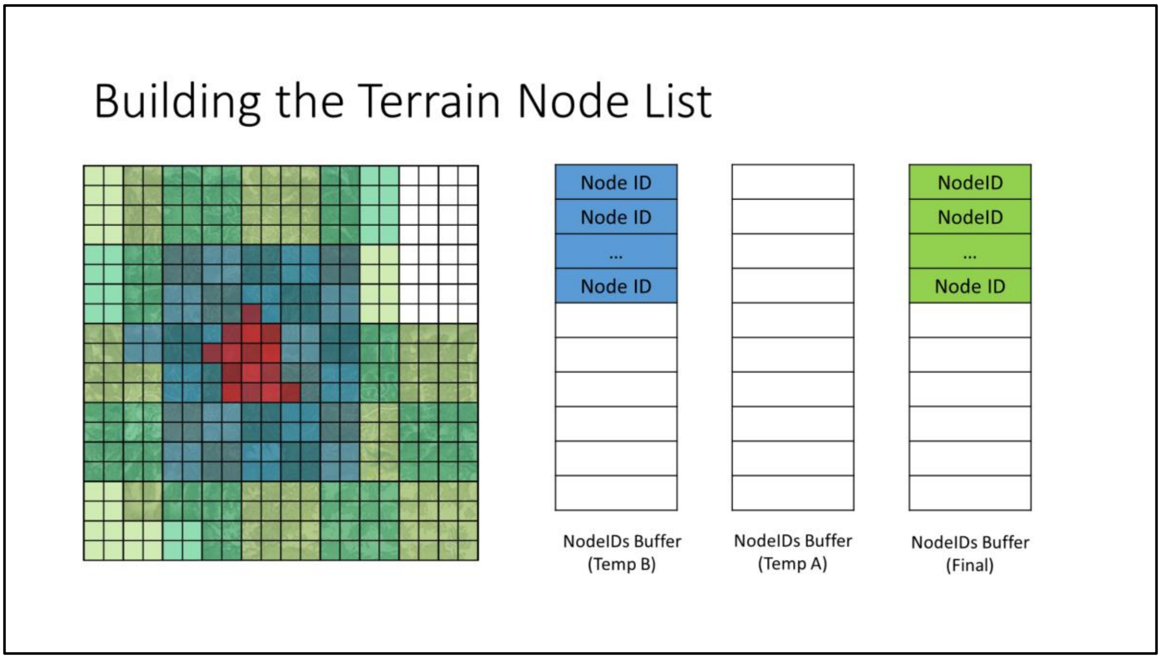
NodeIDs Buffer
(Temp A)

NodeID
NodeID

NodeIDs Buffer
(Final)

This node does have all children loaded but we decide not to divide it because we have already met our level of detail criteria.
We append it to the list of final nodes to render

In Far Cry we do this based on distance from one or more cameras.
We also take into account a per node LOD bias to force some parts of the world to use higher detail.



Building the Terrain Node List

Node ID
Node ID
...
Node ID

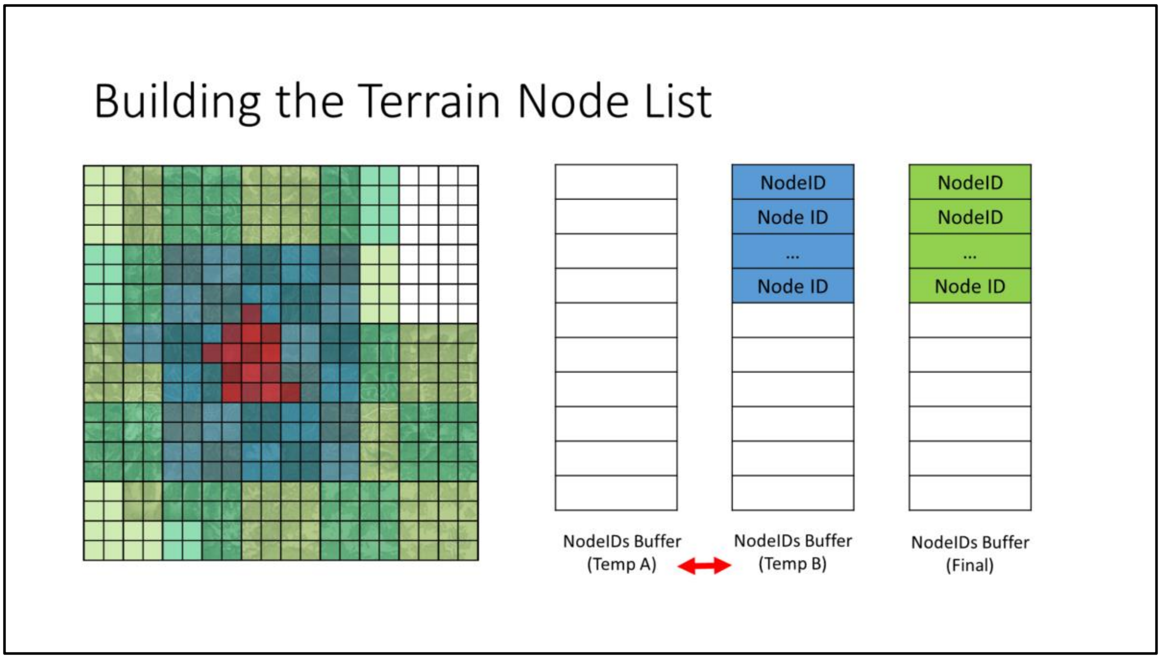
NodeIDs Buffer
(Temp B)

NodeIDs Buffer
(Temp A)

NodeID
NodeID
...
Node ID

NodeIDs Buffer
(Final)

We have consumed temporary buffers A and are ready for the next dispatch which will consume temporary buffer B.



Building the Terrain Node List

NodeIDs Buffer (Temp A)

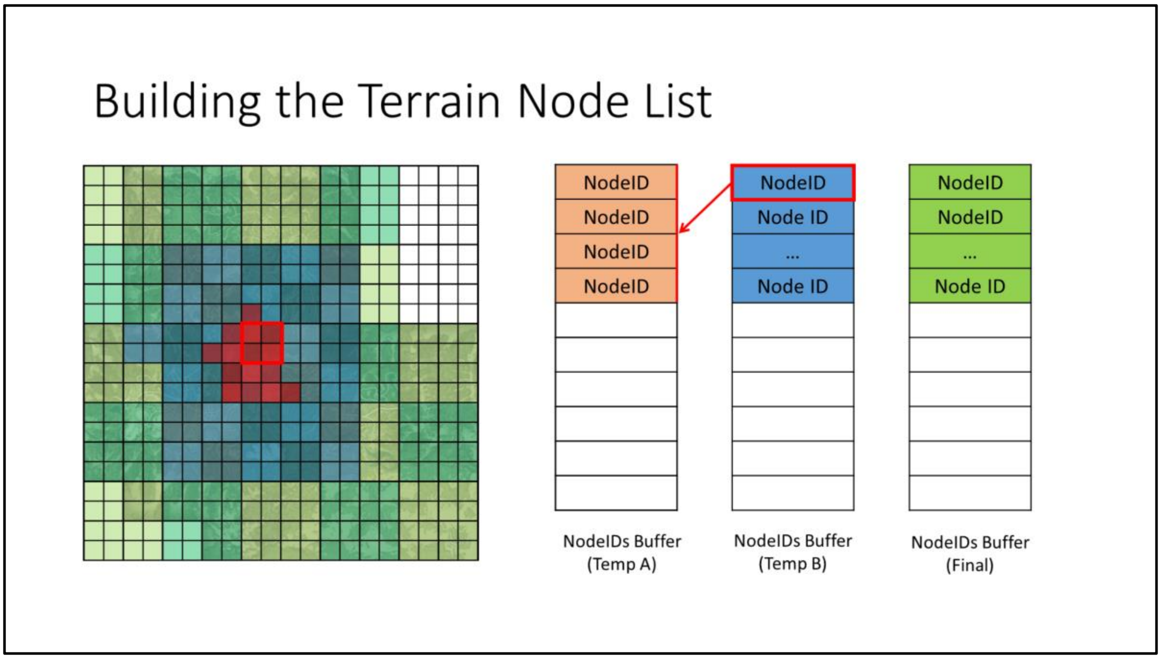
NodeID
Node ID
...
Node ID

NodeIDs Buffer (Temp B)

NodeID
NodeID
...
Node ID

NodeIDs Buffer (Final)

We switch temporary buffers A and B and go to the next LOD.



Building the Terrain Node List

NodeID
NodeID
NodeID
NodeID

NodeIDs Buffer
(Temp A)

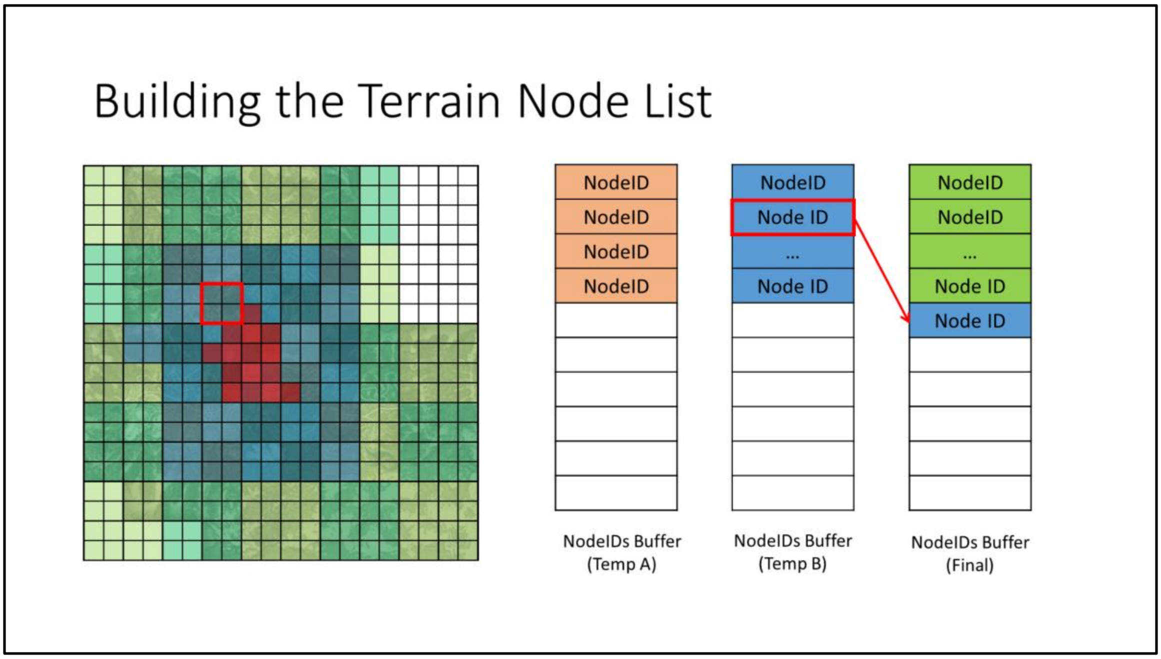
NodeID
Node ID
...
Node ID

NodeIDs Buffer
(Temp B)

NodeID
NodeID
...
Node ID

NodeIDs Buffer
(Final)

Again we process each node.
This node can be divided so its children are appended to temp buffer A.



Building the Terrain Node List

NodeID
NodeID
NodeID
NodeID

NodeIDs Buffer
(Temp A)

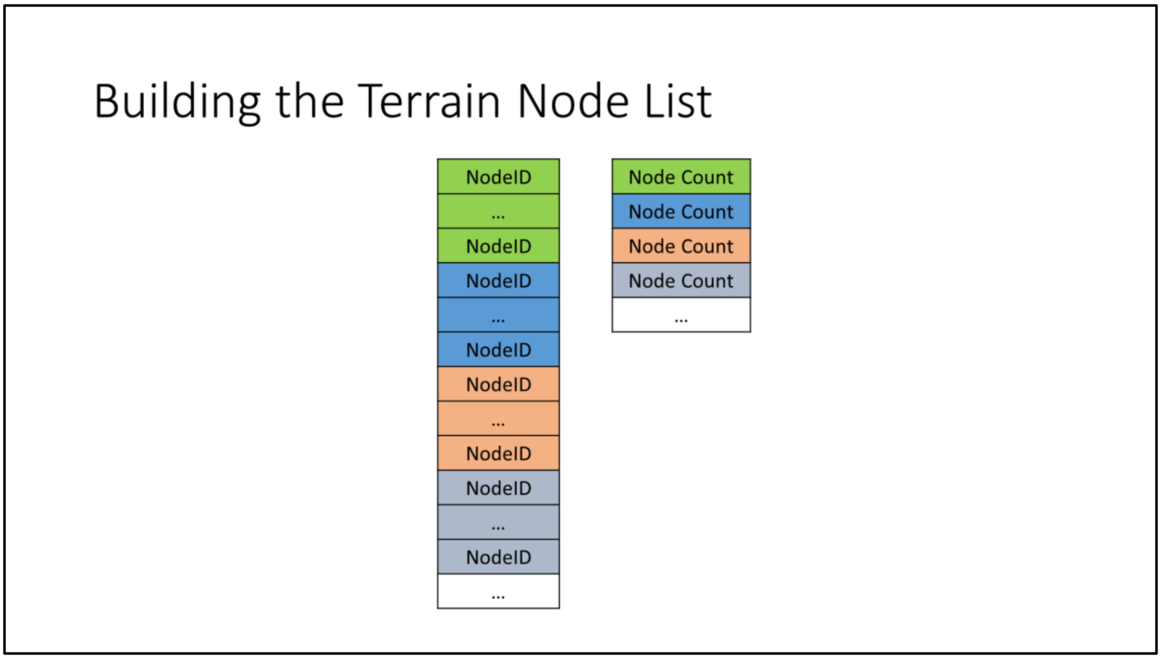
NodeID
Node ID
...
Node ID

NodeIDs Buffer
(Temp B)

NodeID
NodeID
...
Node ID
Node ID

NodeIDs Buffer
(Final)

This node can’t be divided so we append it to our final list of nodes.



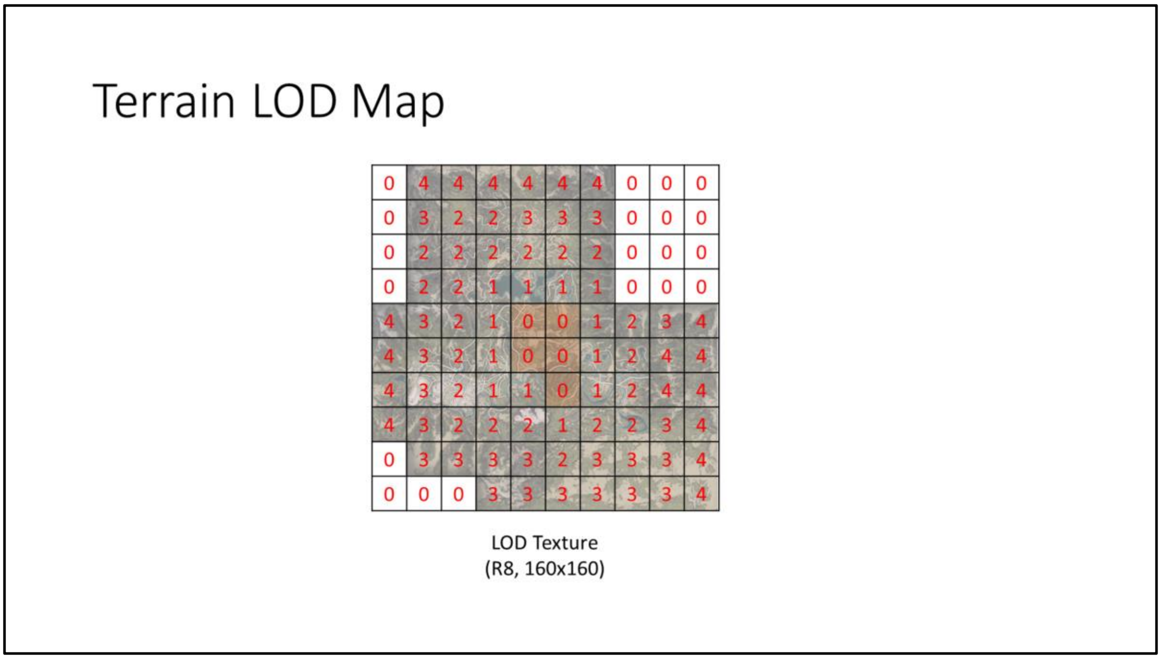
Building the Terrain Node List

NodeID
…
NodeID
NodeID
…
NodeID
NodeID
…
NodeID
NodeID
…
NodeID
…

Node Count
Node Count
Node Count
Node Count
…

We do this for each LOD level and end up with our final result.
While doing this we also stored a buffer of node counts for each LOD.

The process can be done quite elegantly with a compute shader dispatch for each pass.
However that requires two dispatches per pass, because we need one extra dispatch to fill an indirect args buffer to feed the next pass.
And each dispatch involves a small synchronization overhead.
So on Far Cry to keep performance cost to a minimum we do all the passes in a single dispatch, storing the intermediate data in LDS.
We can do this because we have a fixed upper bound for the number of nodes that we will generate, and it is less then the maximum possible compute shader group size.



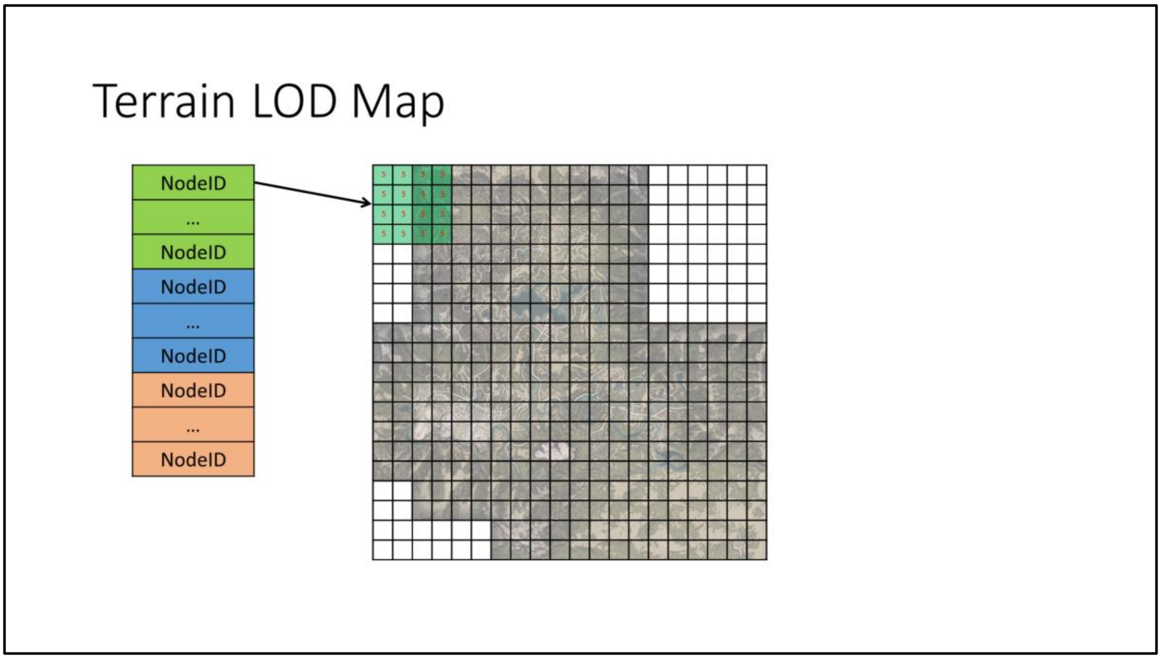
Terrain LOD Map

0	4	4	4	4	4	4	0	0	0
0	3	2	2	3	3	3	0	0	0
0	2	2	2	2	2	2	0	0	0
0	2	2	1	1	1	1	0	0	0
4	3	2	1	0	0	1	2	3	4
4	3	2	1	0	0	1	2	4	4
4	3	2	1	1	0	1	2	4	4
4	3	2	2	2	1	2	2	3	4
0	3	3	3	3	2	3	3	3	4
0	0	0	3	3	3	3	3	3	4

LOD Texture
(R8, 160x160)

Each frame we completely fill an 8-bit texture containing the terrain LOD at each sector of the world.
We will need this information for one thing only, and that is to detect where we need to stitch between patches of with different LODs.

Note we only modify the patch vertices when an adjacent patch has a lower LOD, so by filling the empty areas with 0, we effectively ignore them.



Terrain LOD Map

NodeID

…

NodeID

NodeID

…

NodeID

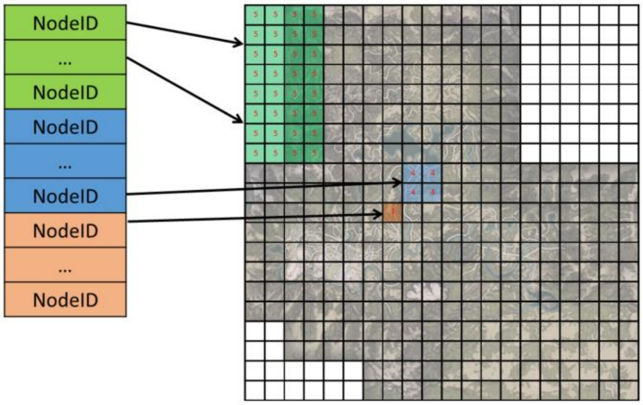
NodeID

…

NodeID

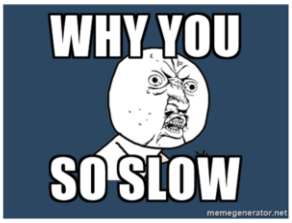
To build the LOD map we process the Terrain Node List that we have just built.
For each node we simply fill in its LOD for each of the sectors that it covers.

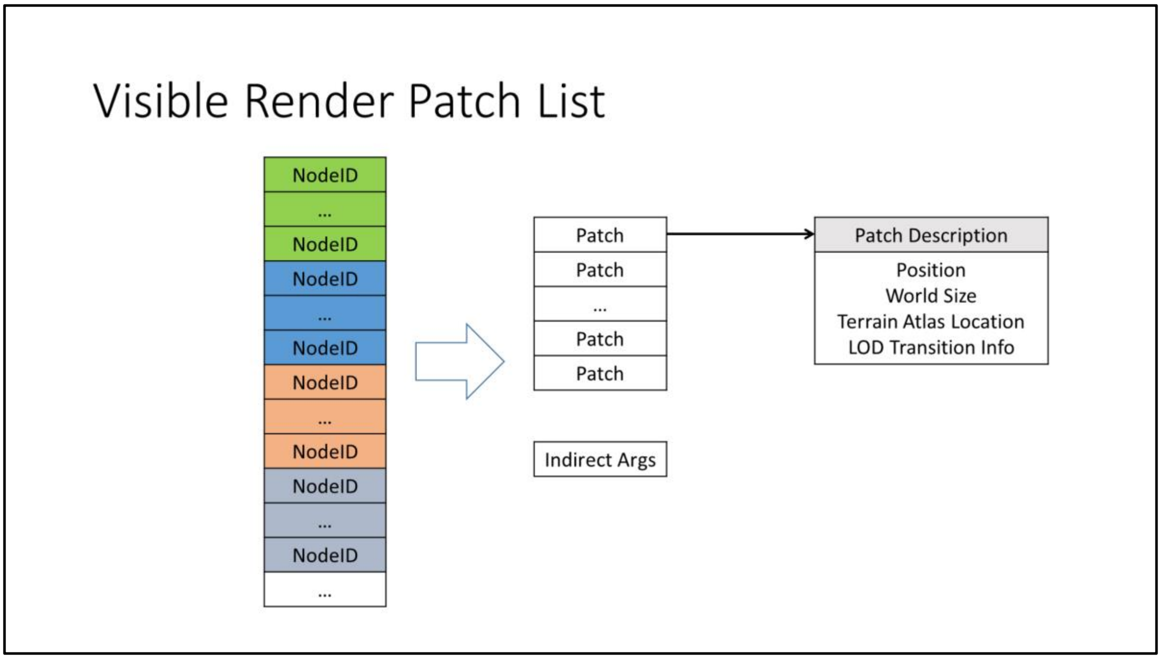
Terrain LOD Map



Filling the Terrain LOD Map

- **Don’t** use NodeCount threads to *push* the values into the Terrain LOD Map
  - In the worst case one thread has to write 32x32 values!
- **Do** use SectorCount threads, and *pull* the values from the Terrain Node List
  - Each thread writes 1 or 0 values





Visible Render Patch List

NodeID
…
NodeID
NodeID
…
NodeID
NodeID
…
NodeID
NodeID
…
NodeID
…

Patch
Patch
…
Patch
Patch

Indirect Args

Patch Description

Position
World Size
Terrain Atlas Location
LOD Transition Info

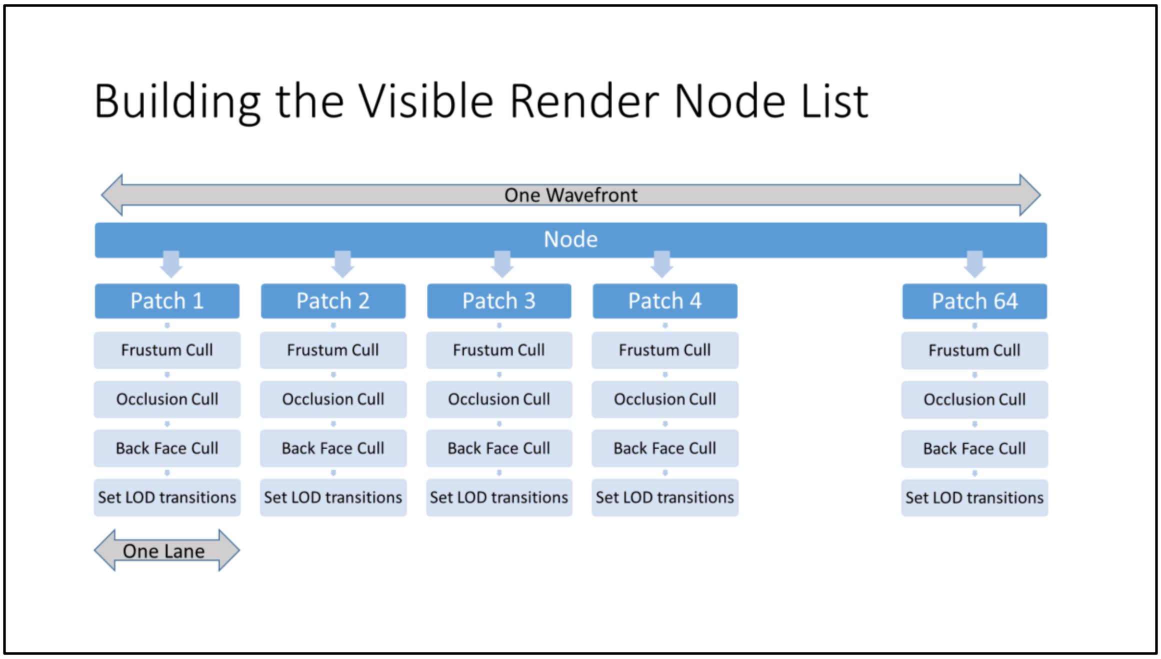
The final data structure is the Visible Render Patch List.

It’s simply a buffer containing a list of terrain patches that we want to actually render in a single instanced draw call.
The buffer is accompanied with an IndirectArgs buffer to drive the draw call.

The patch structure itself contains world position and size, and the atlas location of the terrain quad tree textures that the patch should sample.

Each patch will be rendered as a 16x16 grid.

We will create the patch list by taking all of the quad tree nodes in our Terrain Node List, breaking them into smaller patches and culling ones that we don’t need to render.



Building the Visible Render Node List

One Wavefront

Node

Patch 1	Patch 2	Patch 3	Patch 4	Patch 64
Frustum Cull	Frustum Cull	Frustum Cull	Frustum Cull	Frustum Cull
Occlusion Cull	Occlusion Cull	Occlusion Cull	Occlusion Cull	Occlusion Cull
Back Face Cull	Back Face Cull	Back Face Cull	Back Face Cull	Back Face Cull
Set LOD transitions	Set LOD transitions	Set LOD transitions	Set LOD transitions	Set LOD transitions

One Lane

Each node is divided into 8x8 patches
And as I mentioned the patches can be rendered as 16*16 vertex grids.
This works out nicely as one patch to process per compute shader thread in a wavefront.

For each patch we:

- Frustum cull
- Occlusion buffer cull
- Back face cull
- Calculate and pack LOD transitions into the patch data

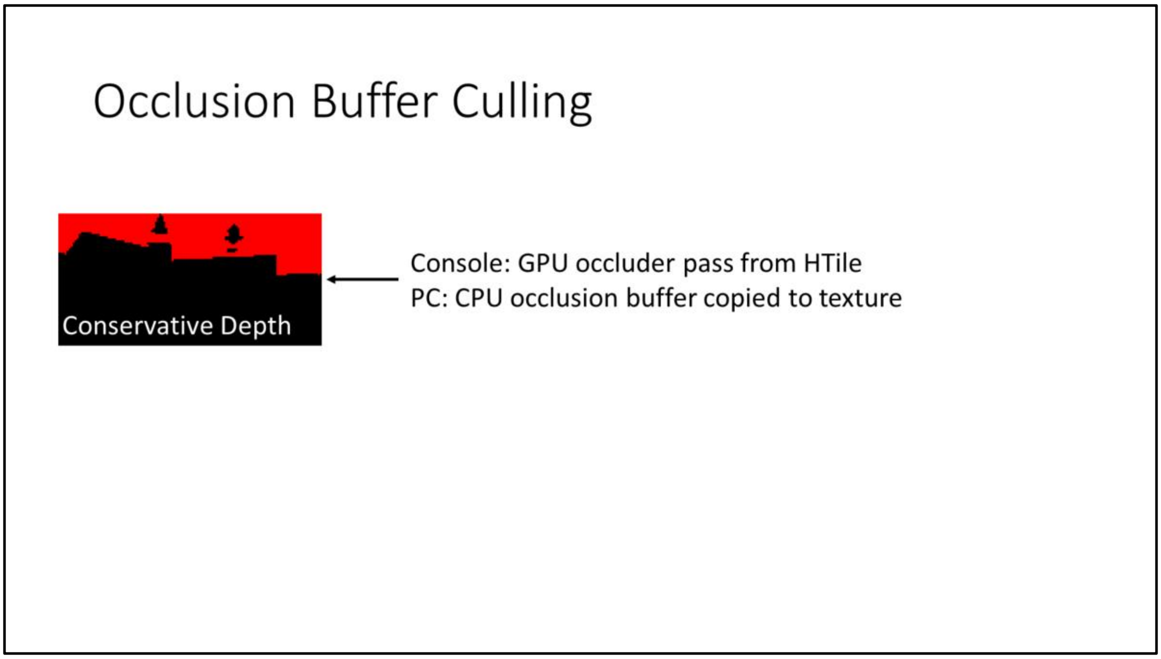
GPU Culling

SIGGRAPH 2015: Advances in Real-Time Rendering in Games

GPU-Driven Rendering Pipelines

Ulrich Haar, Sebastian Aaltonen

The culling steps follow a similar approach to the GPU pipeline for arbitrary meshes first introduced in Siggraph 2015 by Haar and Aaltonen.



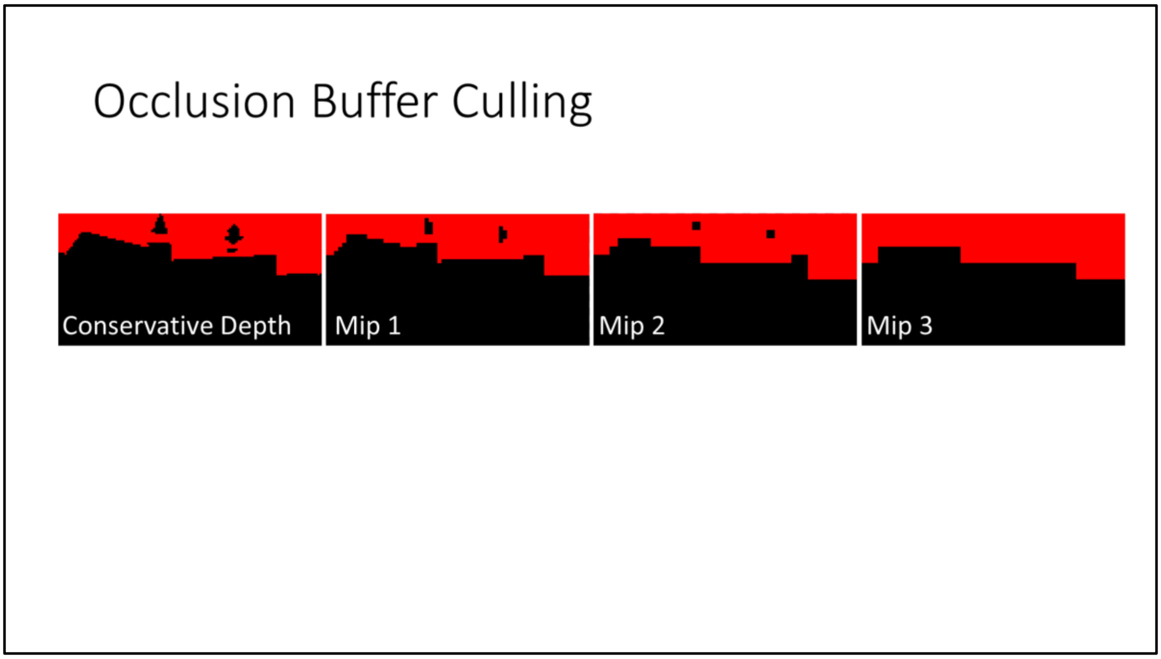
Occlusion Buffer Culling

Conservative Depth

Console: GPU occluder pass from HTile
PC: CPU occlusion buffer copied to texture

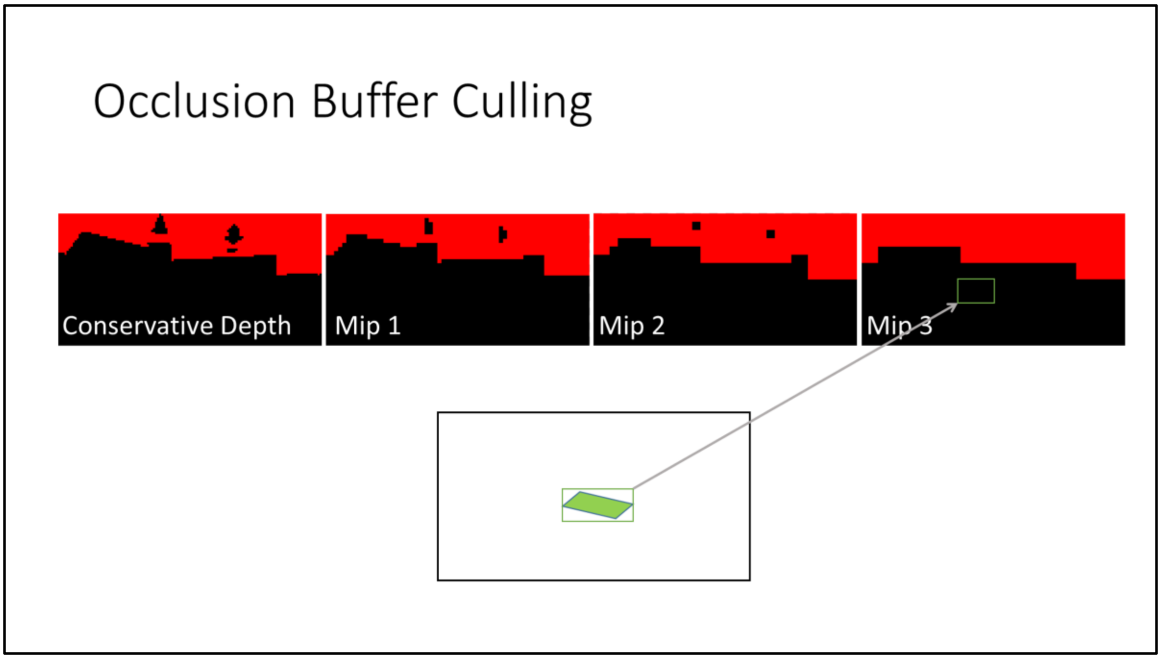
We use a sub resolution conservative depth buffer to cull the terrain geometry.

On console platforms we use a GPU best occluder pass to prime the depth buffer and extract the HTile.
On PC we have a software rasterized occlusion buffer that we already use for CPU visibility which we upload to a texture.



Occlusion Buffer Culling

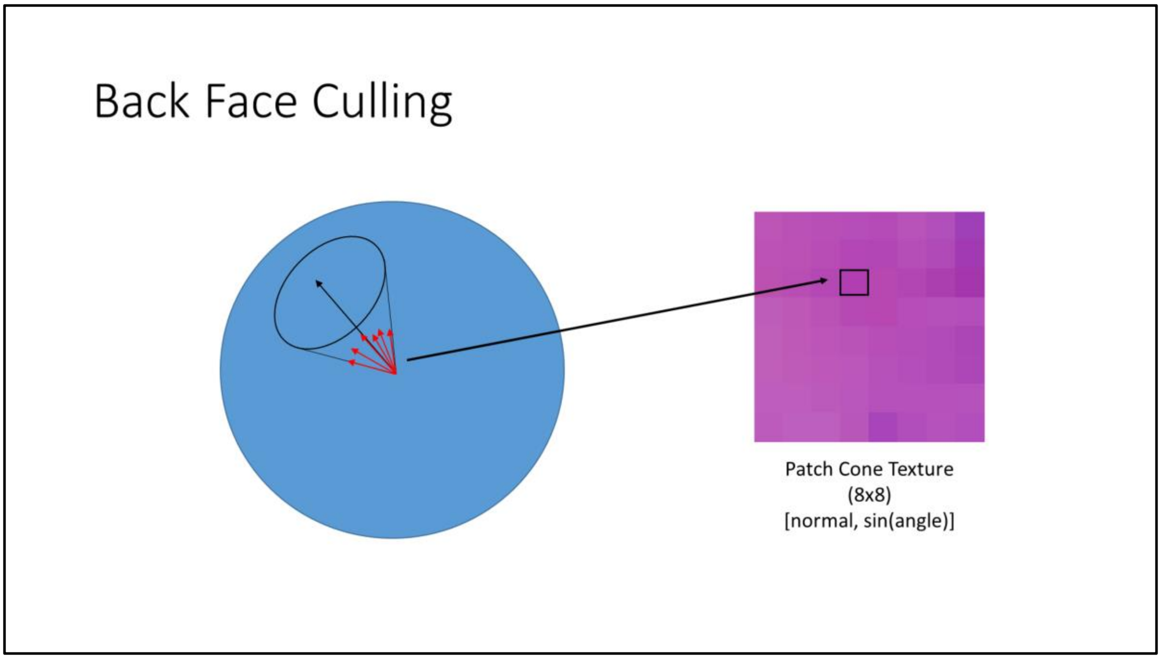
We then generate mip levels so that we can efficiently compare depth at multiple object scales.



Occlusion Buffer Culling

In the culling pass we:

- Get the terrain patch bounding volume.
- Find its projected screen extents.
- Find one or more sample in mip hierarchy that covers the extents.
- Test and conservatively occlude.



Back Face Culling

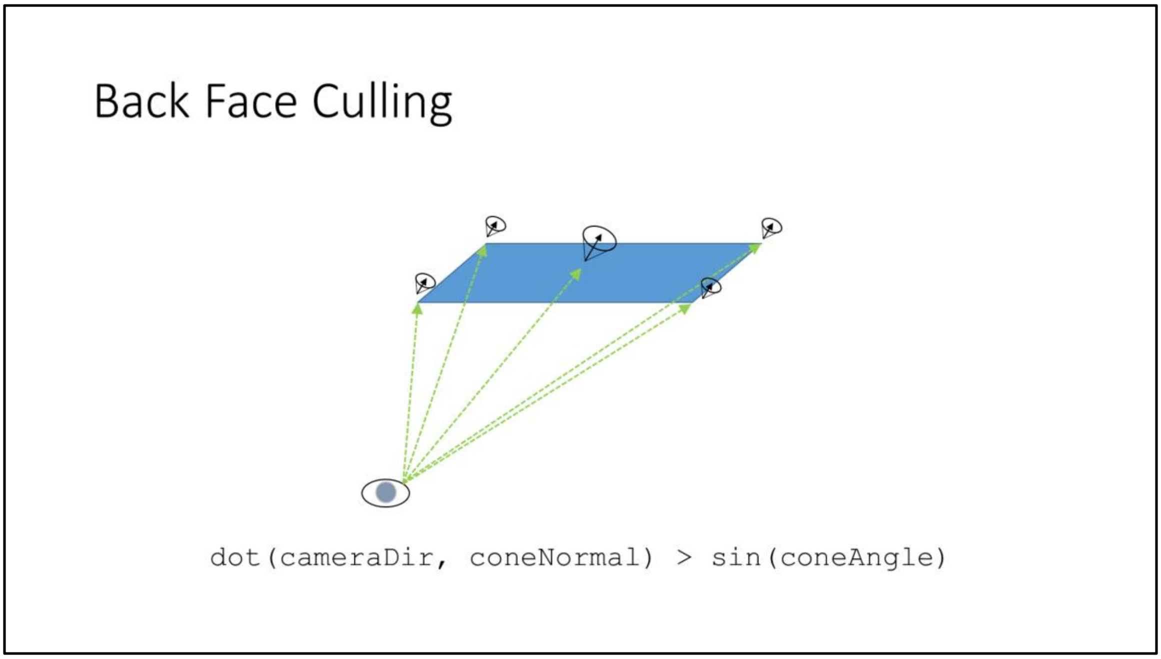
Patch Cone Texture
(8x8)
[normal, sin(angle)]

Next we will remove any patches that can be back faced culled because all of their faces point away from the camera.

This requires us to build and store information about each patch in an offline data build step.

In this offline step we find the world space normals for every triangle in a patch.
Then we find the minimum circle on a sphere that contains all of these normal, which we can visualize as a cone.
We store the cone for each of the 8x8 patches in a node as a texel in an 8x8 texture.
The texel holds the normal at the centre of the cone, and the half angle subtended by the cone.

See [Ericson] for finding minimal bounding sphere of normals.
We use a BC3 texture to store the normal and angle.
The normal is stored in two color channels, and sin(angle) is stored in the alpha channel.
This could lead to precision issues so we have to allow some fudge for conservative tests.

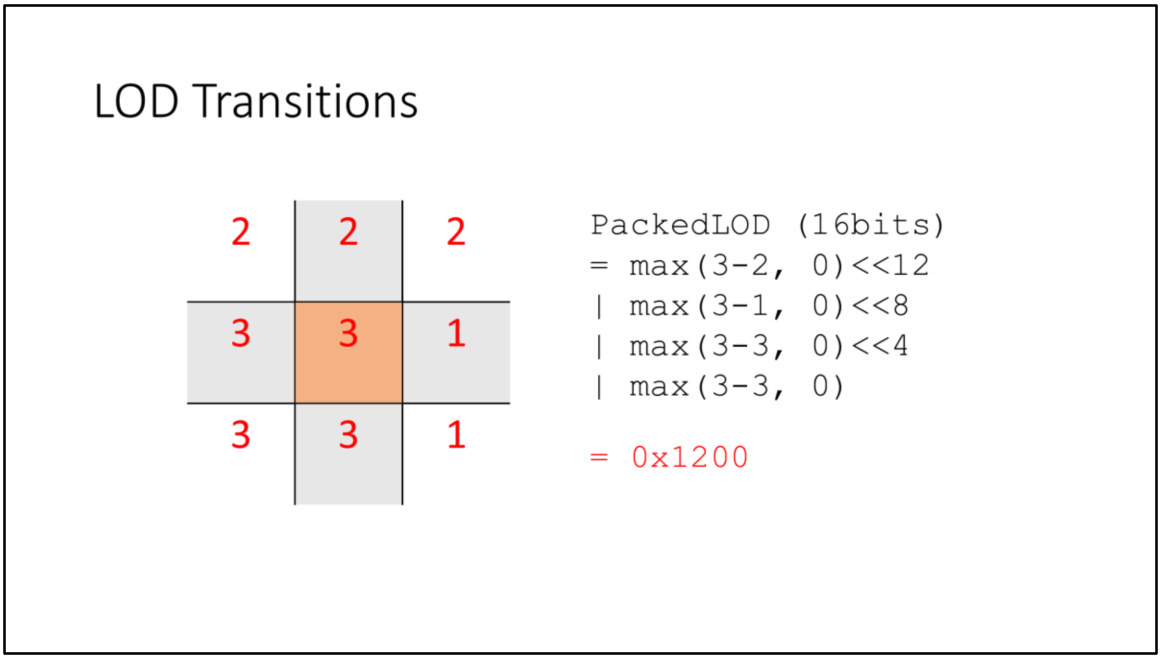


Back Face Culling

```
dot(cameraDir, coneNormal) > sin(coneAngle)
```

At runtime we sample the patch cone texture map and test against the camera direction to see if we can cull the patch.
See [Shirmun] for original description of this technique.

Unfortunately it’s not as simple as checking a single dot product because the camera direction varies across the patch.
To be conservative, we need to check against the camera direction to each corner of the patch’s bounding box.



LOD Transitions

2	2	2
3	3	1
3	3	1

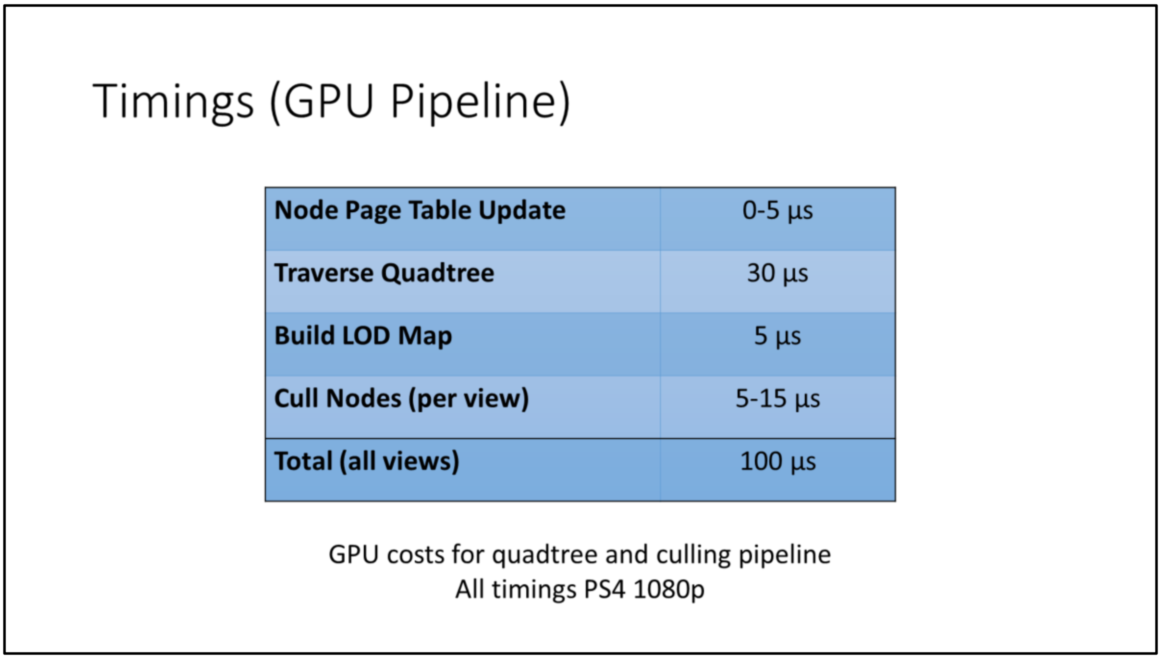
```
PackedLOD (16bits)
= max(3-2, 0)<<12
| max(3-1, 0)<<8
| max(3-3, 0)<<4
| max(3-3, 0)

= 0x1200
```

Each output patch description stores a packed LOD description.
This is 16 bits: 4 bits each for the 4 sides describing the (clamped positive) LOD delta.

We fill this value during the culling pass by sampling our LOD map around each patch.
This value will be read later in the vertex shader to stitch the meshes.

For example here we see a patch that is at the corner or a sector where the LOD is changing.
To the north we have a LOD delta of 1.
To the east we have a LOD delta of 2.
And south and west are probably in the same sector so there is no LOD delta.

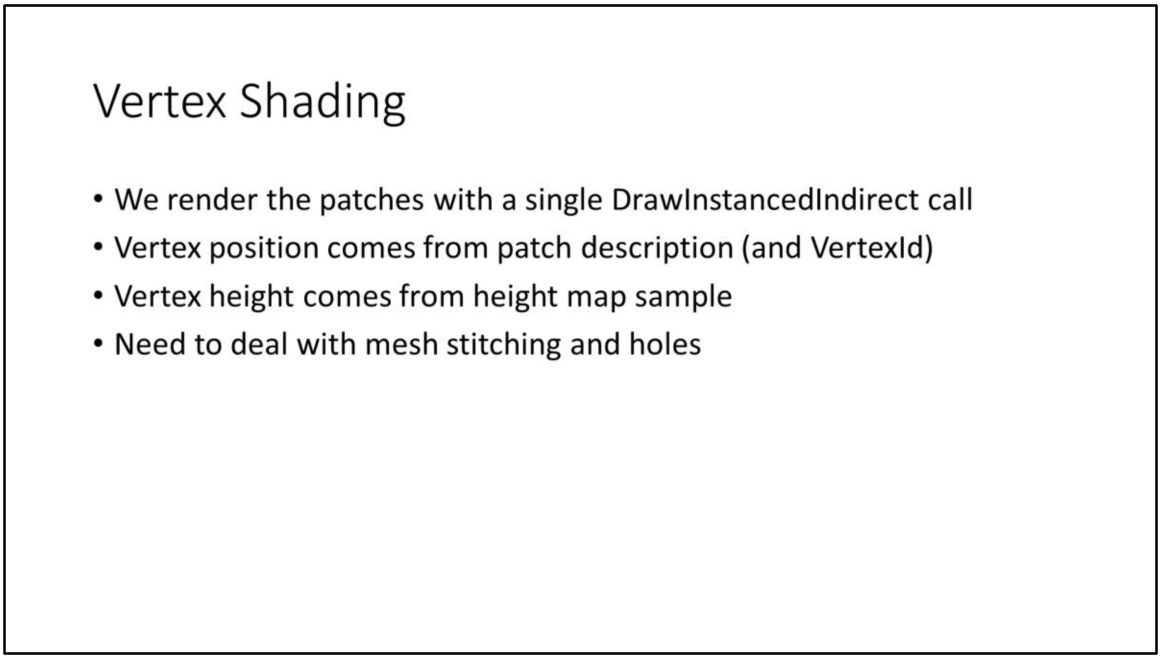


Timings (GPU Pipeline)

Node Page Table Update	0-5 µs
Traverse Quadtree	30 µs
Build LOD Map	5 µs
Cull Nodes (per view)	5-15 µs
Total (all views)	100 µs

GPU costs for quadtree and culling pipeline
All timings PS4 1080p

After this we have our final output patch description buffer ready for rendering.
Total cost of GPU pipeline processing without rendering is ~0.1ms each frame.
Most of this can run in asynchronous compute early in the frame making it effectively free.

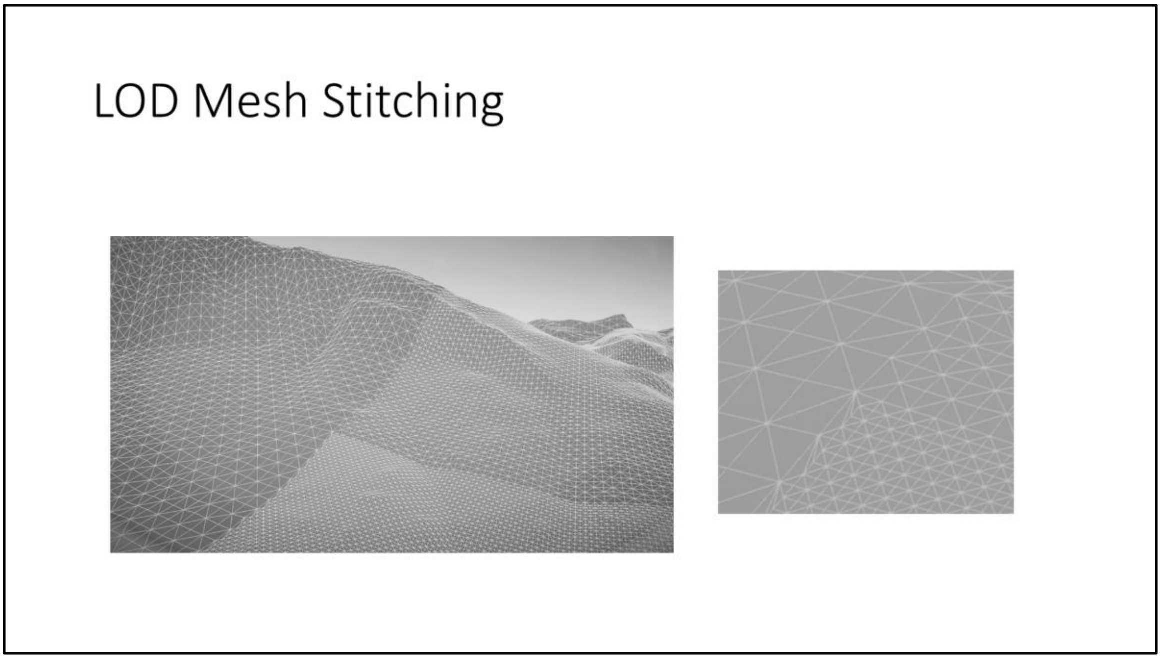


Vertex Shading

- We render the patches with a single DrawInstancedIndirect call
- Vertex position comes from patch description (and VertexId)
- Vertex height comes from height map sample
- Need to deal with mesh stitching and holes

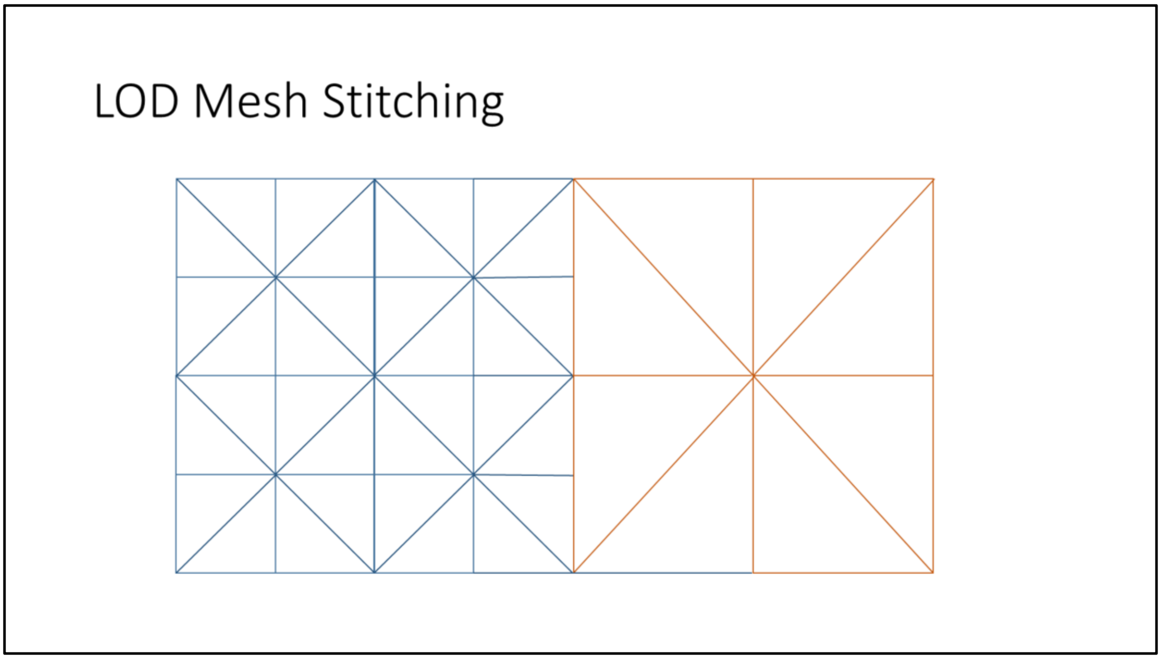
We take our patch list and render the patches themselves

There’s an interesting related post about index buffer layouts for grid rendering in [Kapoulkine]



LOD Mesh Stitching

Where two patches from different quad tree LODs we need to ensure that all of the vertices are stitched so that we can’t see seams between the patches.



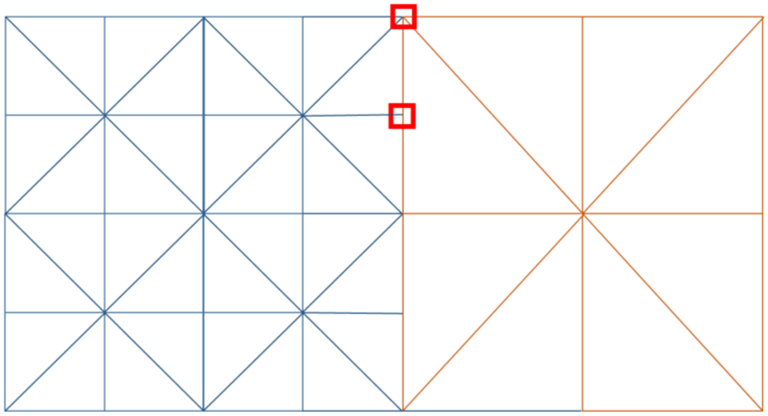
LOD Mesh Stitching

We know when a patch connects to a patch of lower LOD by reading the LOD packed data created in the culling pass.

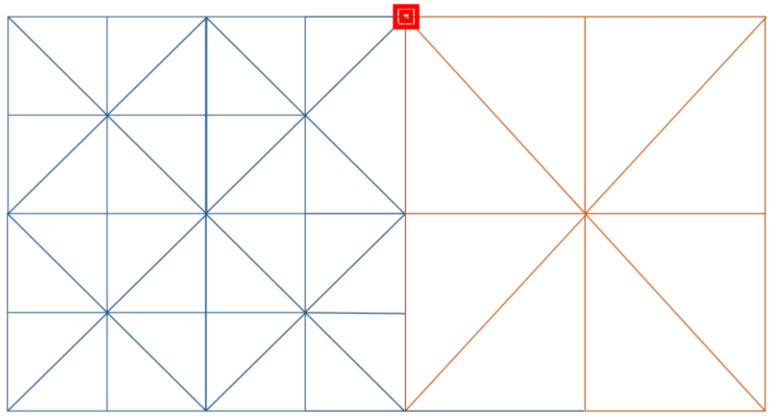
In the vertex shader we can simply morph the vertices at the edge of a patch to the nearest vertex at the required subdivision level.

Here we see a drop of a single LOD, so we weld each pair of vertices.

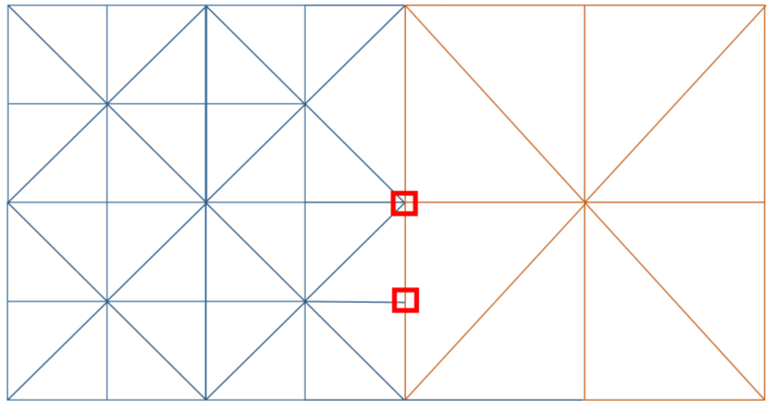
LOD Mesh Stitching



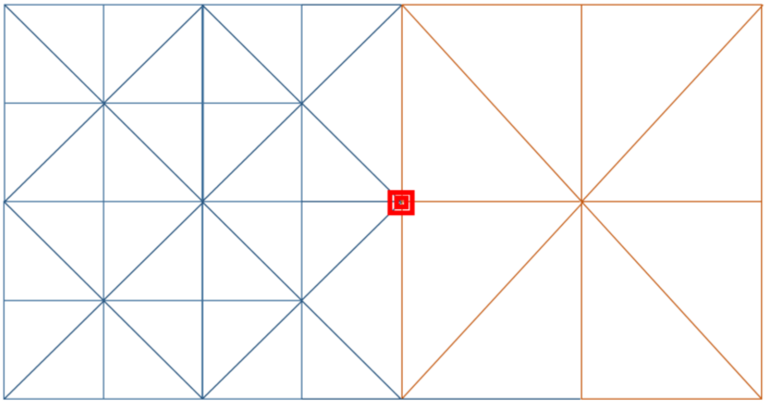
LOD Mesh Stitching

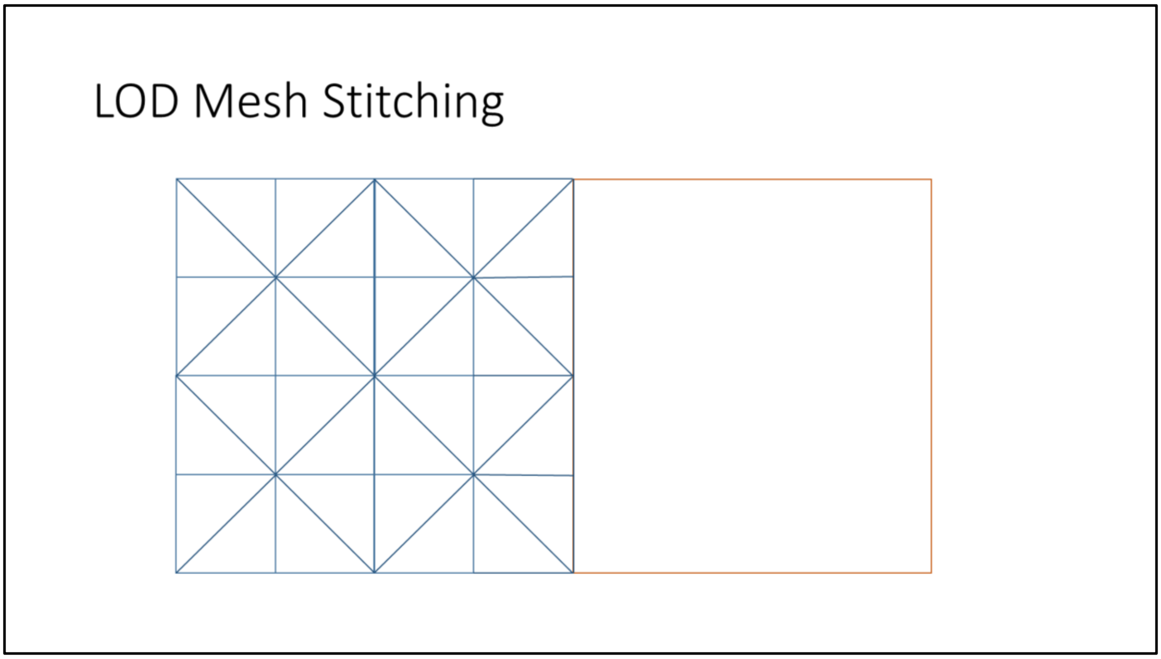


LOD Mesh Stitching



LOD Mesh Stitching

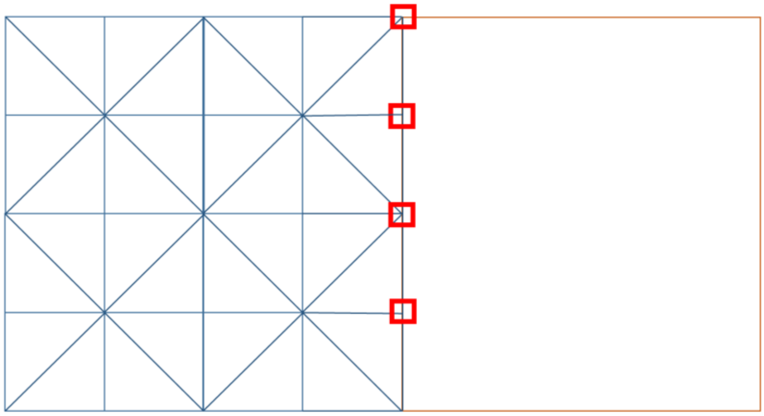


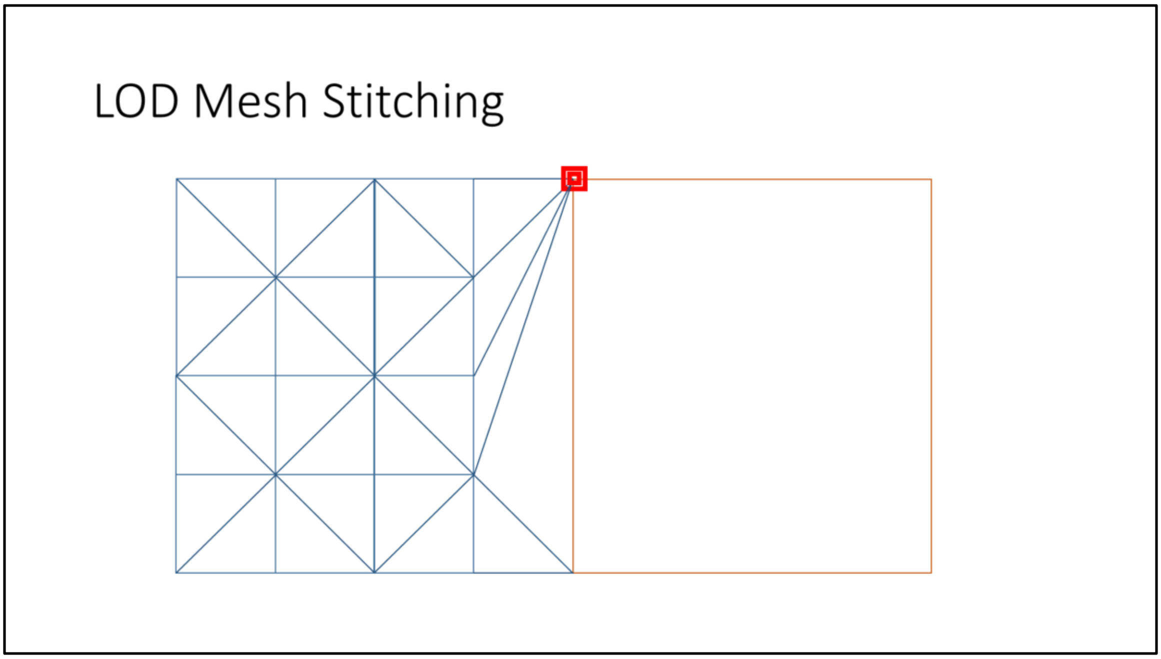


LOD Mesh Stitching

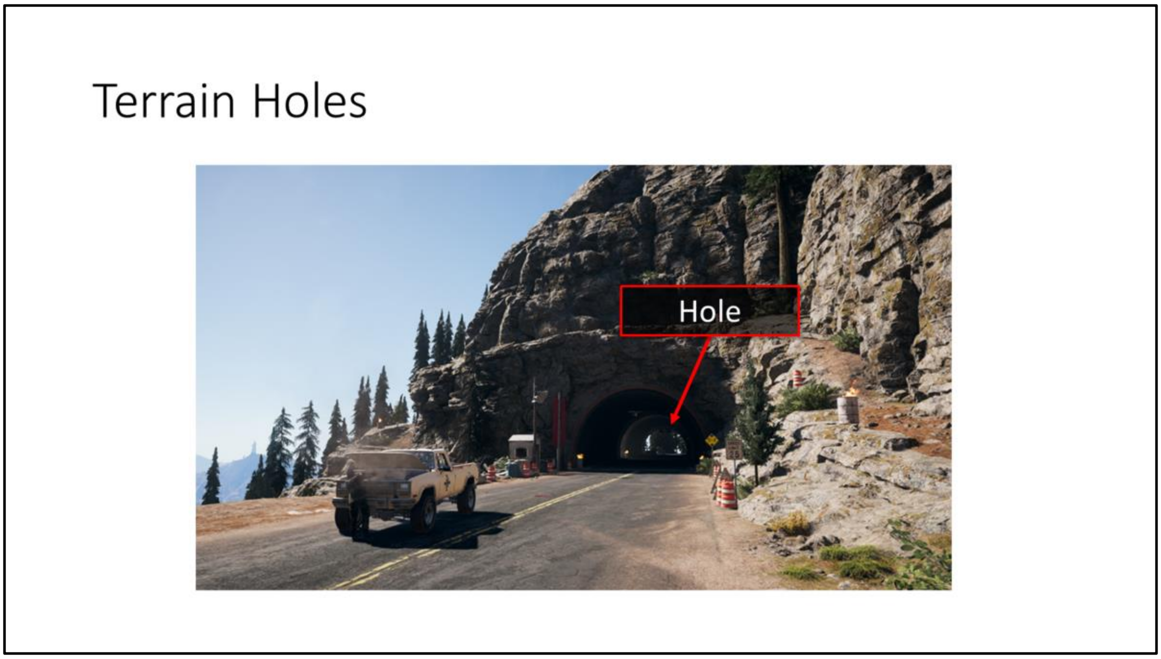
Here we see a drop of a two LODs, so we weld four vertices at a time.

LOD Mesh Stitching



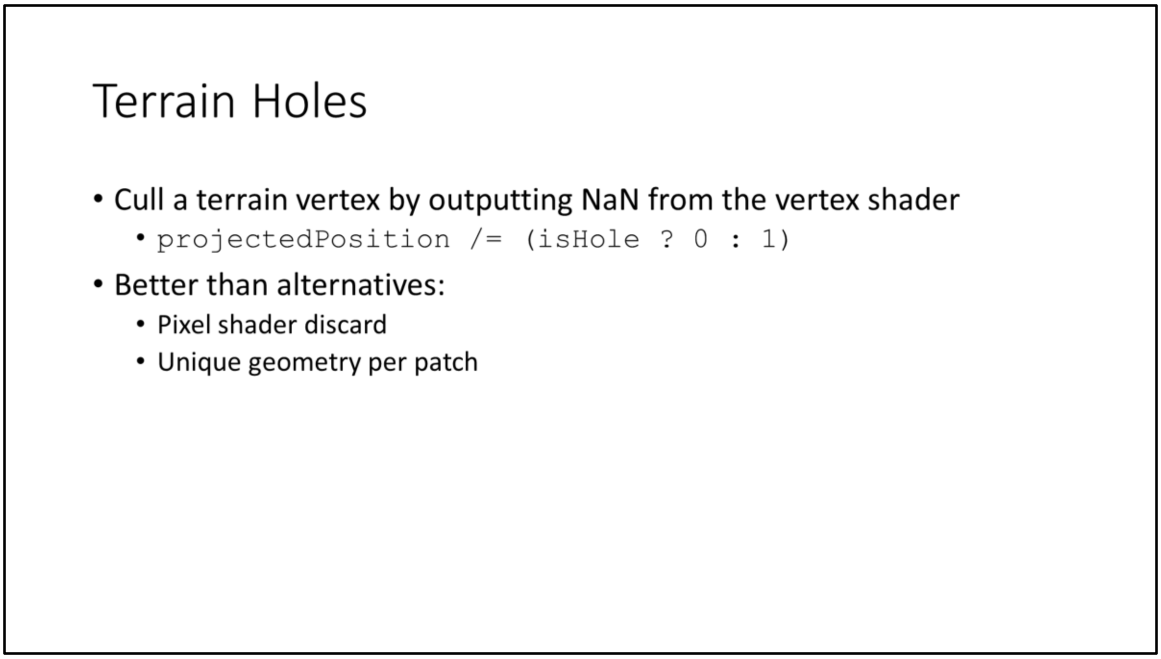


LOD Mesh Stitching



Terrain Holes

Holes are needed around opening to caves/bunkers.
We store terrain holes in 1 bit alpha in our BC1 atlas albedo map.

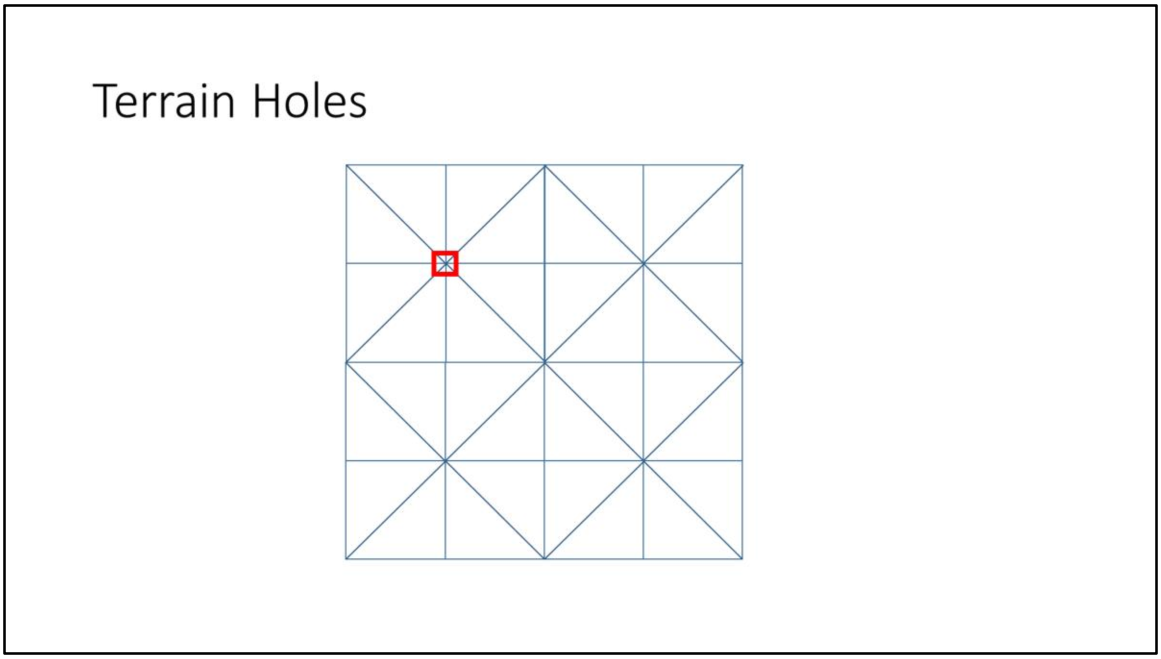


Terrain Holes

- Cull a terrain vertex by outputting NaN from the vertex shader
  - `projectedPosition /= (isHole ? 0 : 1)`
- Better than alternatives:
  - Pixel shader discard
  - Unique geometry per patch

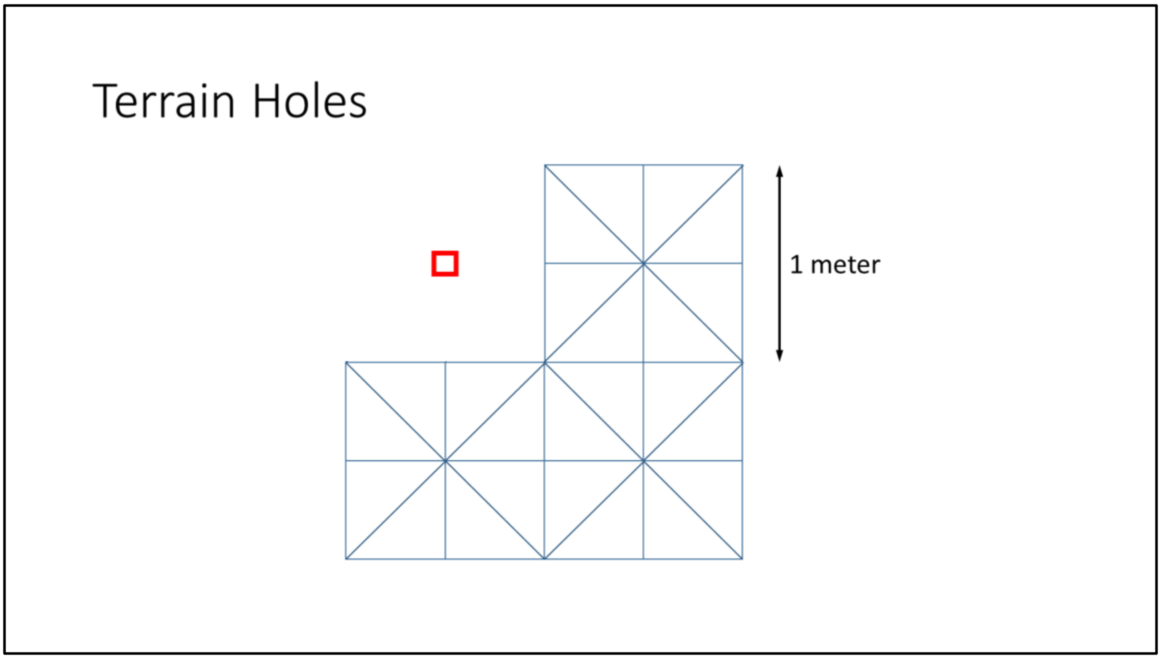
We can cull vertices in the vertex shader using the trick of returning NaN in the projected position.
Outputting NaN for projection position is not specifically documented in graphics APIs but all GPU hardware seems to support this.

This is wonderfully simple and cheap compared to the alternatives of either using pixel shader discard or building unique geometry for patches with holes.



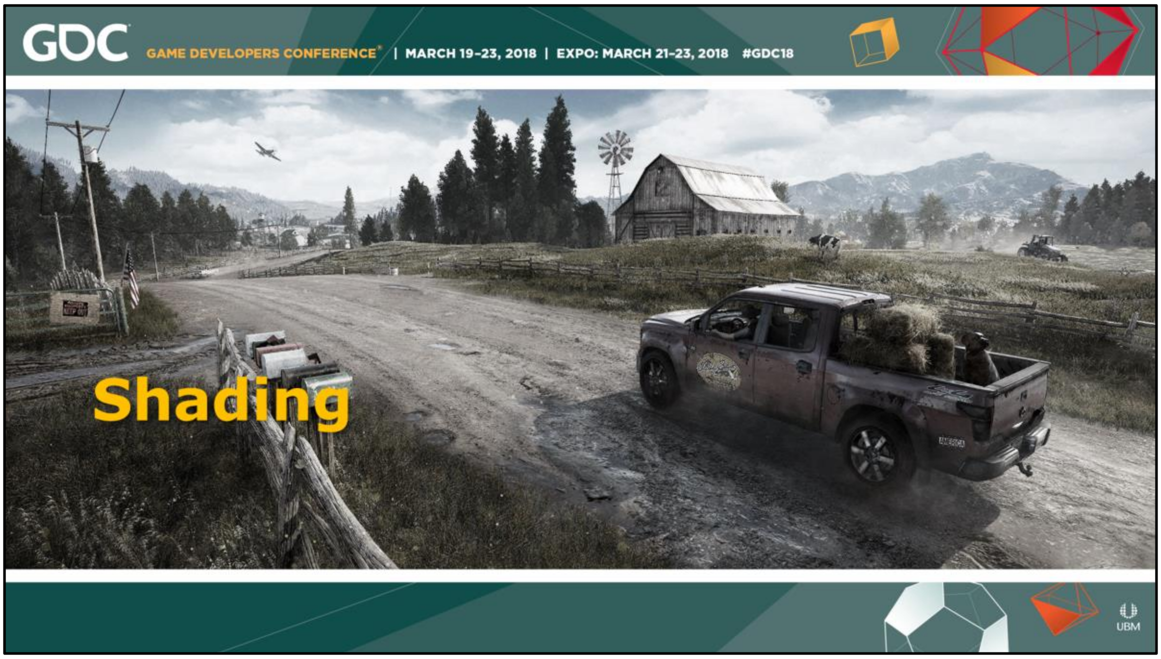
Terrain Holes

A culled vertex will cull all connected triangles.
So culling the central vertex in our terrain topology kills 8 triangles.



Terrain Holes

This makes for a hole resolution that can only be half of the terrain resolution.
But our level designers are OK with 1 meter holes in their half meter terrain.



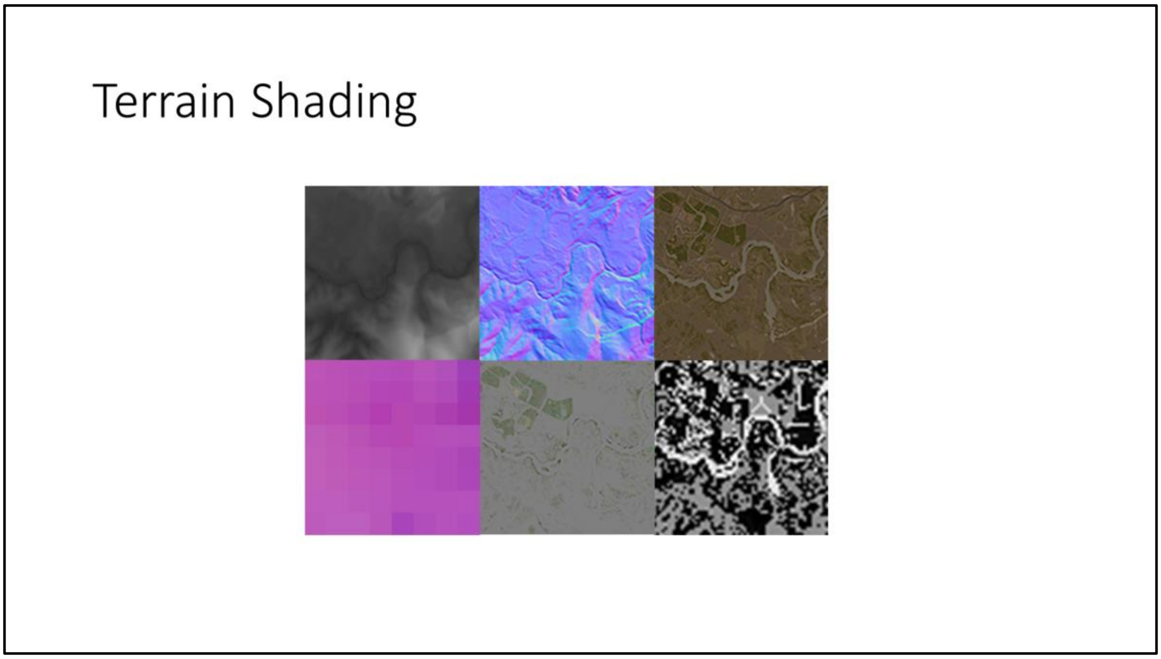
GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Shading

Now that we’ve generated a terrain mesh I’ll describe how we shade it.

Terrain Shading

Our shading approach is taken from the 2017 GDC talk on Ghost Recon Wildlands. Check that talk if you want more details, but I'll give a brief summary here.



Terrain Shading

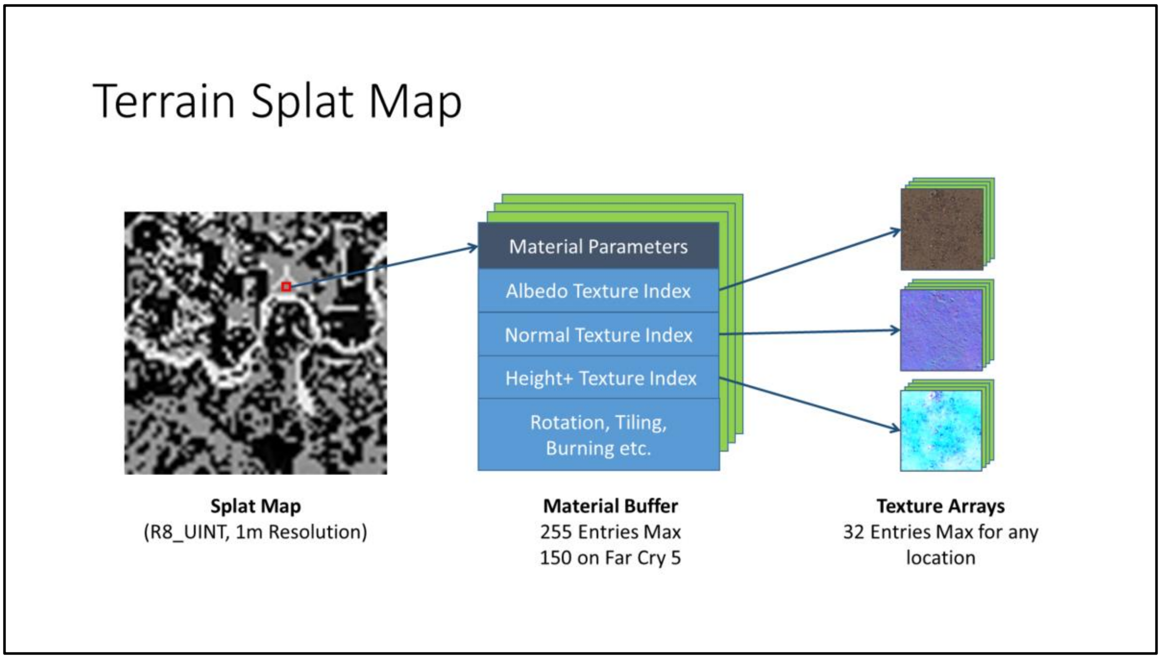
Remember that each quad tree node carries a texture payload.

We’ve already seen the Height, Normal, Albedo textures. Also there is:

- Patch cone map (for back face culling we saw earlier)
- Color modulation map
- Splat map

As I mentioned these textures are each loaded into atlases at runtime.

Full format details are:
Heightmap (R16_UNORM, 129x129)
World space normal map (we can assume positive z and also pack smoothness and specular occlusion) (BC3, 132x132)
Albedo map (BC1, 132x132)
Color modulation map (BC1, 132x132)
Splat map (R8_UNORM, 65x65)
Patch cone texture (BC3, 8x8)



Terrain Splat Map

Material Parameters

Albedo Texture Index

Normal Texture Index

Height+ Texture Index

Rotation, Tiling, Burning etc.

Splat Map
(R8_UINT, 1m Resolution)

Material Buffer
255 Entries Max
150 on Far Cry 5

Texture Arrays
32 Entries Max for any location

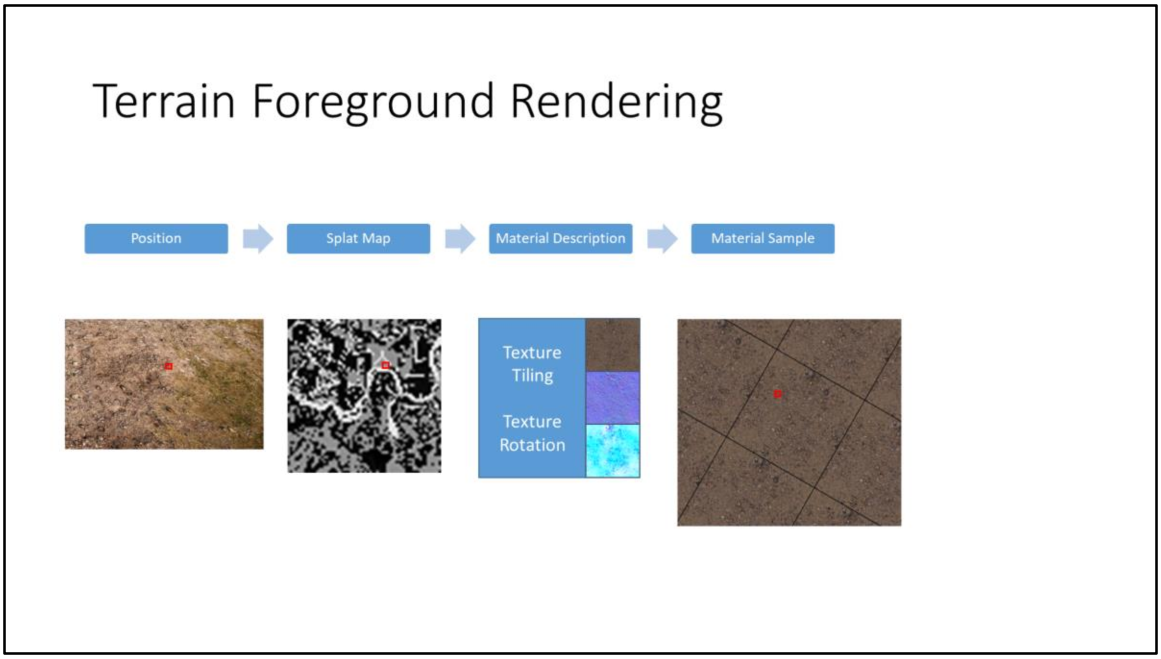
Far shading can be handled directly and cheaply using the albedo/normal of the quad tree payload.
For near shading we need to use the splat map.

This splat map is an 8-bit format texture into which we paint the terrain materials onto the world.
The splat map uses a palette of 256 materials, though on Far Cry 5 we actually have ~150 terrain materials.

A material contains indices to a set of detail textures stored in texture arrays.
We have 3 arrays for a typical set of PBR textures types, and textures are streamed in dynamically as they are referenced by splat maps.
The material also contains a set of parameters defining orientation/tiling/whether the terrain can burn etc.

Resolutions for the detail textures:
Albedo (BC1, 1024x1024)
Normal/Smoothness/Specular Occlusion (BC3, 1024x1024)
Height/Color Mask (BC1, 512x512)

Note that due to memory constraints only 32 of each detail texture type can be loaded into memory at once (for ~70MB of memory).
The streaming of the textures referenced by the quad tree is handled by our streaming dependency system.
But the world building team have to be careful to manage the materials for each biome to stay within their 32 texture budget.



Terrain Foreground Rendering

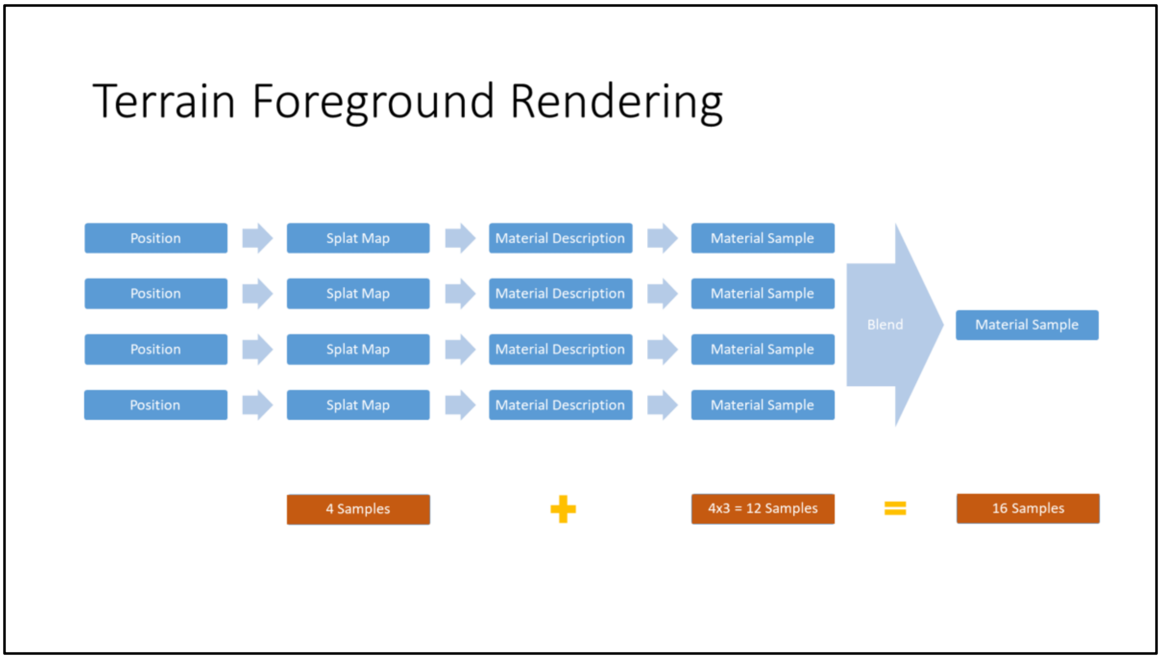
Position → Splat Map → Material Description → Material Sample

Texture Tiling

Texture Rotation

When we want to render a terrain pixel:

- Get the pixel's local position in the tile.
- Use it to look up into the splat map to get our material ID.
- Use the material ID to get the material description (which includes the detail textures to sample).
- We sample the associated detail textures using UVs based on world position and material tiling and rotation.
- This gives us the final material sample data that we can use for shading.



Terrain Foreground Rendering

Position → Splat Map → Material Description → Material Sample

Position → Splat Map → Material Description → Material Sample

Position → Splat Map → Material Description → Material Sample

Position → Splat Map → Material Description → Material Sample

Blend

Material Sample

4 Samples + 4x3 = 12 Samples = 16 Samples

However our terrain location maps to some point between 4 splat map texels.
So we need to read not 1, but 4 surrounding material samples.
We blend the 4 results in the shader using bilinear coefficients modified by the material heights.
This means that each terrain pixel needs to sample 16 textures to calculate the final material result.

Virtual Texturing

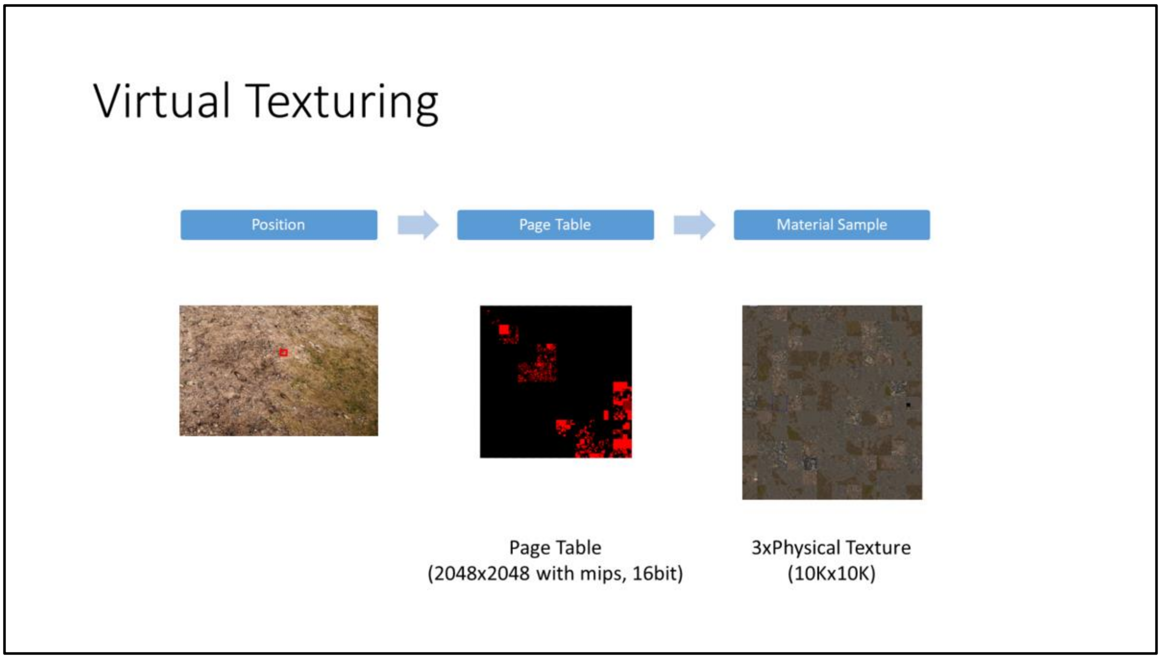
ADAPTIVE VIRTUAL TEXTURE RENDERING IN FARCRY4

Ka Chen
Technique Architect Ubisoft

GAME DEVELOPERS CONFERENCE
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

We can see that the generic sampling from the splat map is expensive.
We save some performance by caching the results into a virtual texture.

Our virtual texture system for terrain is described in the Far Cry 4 Adaptive Virtual Texture presentation at GDC 2015.



Virtual Texturing

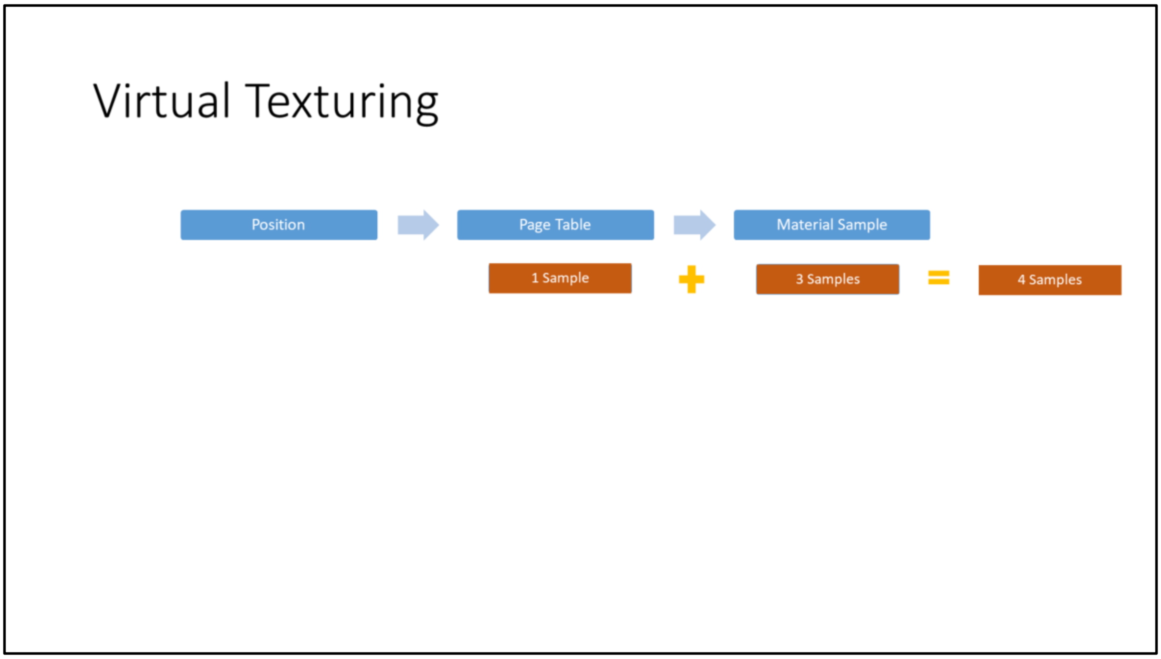
Position → Page Table → Material Sample

Page Table
(2048x2048 with mips, 16bit)

3xPhysical Texture
(10Kx10K)

Virtual texturing allows us to simulate having an extremely large texture covering the terrain.
We only physically store the parts of the texture that we need to sample at the current time.
The cached texture data is stored as pages in a Physical Texture.

If we want to get the terrain material sample at a location we first look up a corresponding location in a Page Table texture.
This gives us the page in the Physical Texture to sample.
We then sample the Physical Texture for our final result.

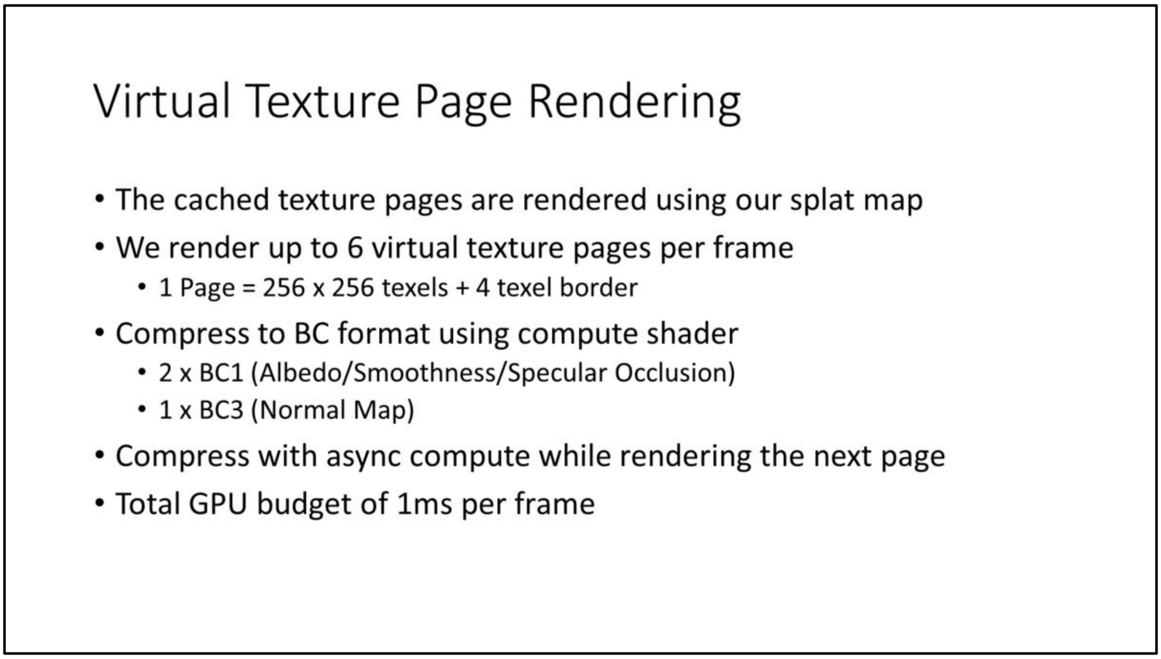


Virtual Texturing

Position → Page Table → Material Sample

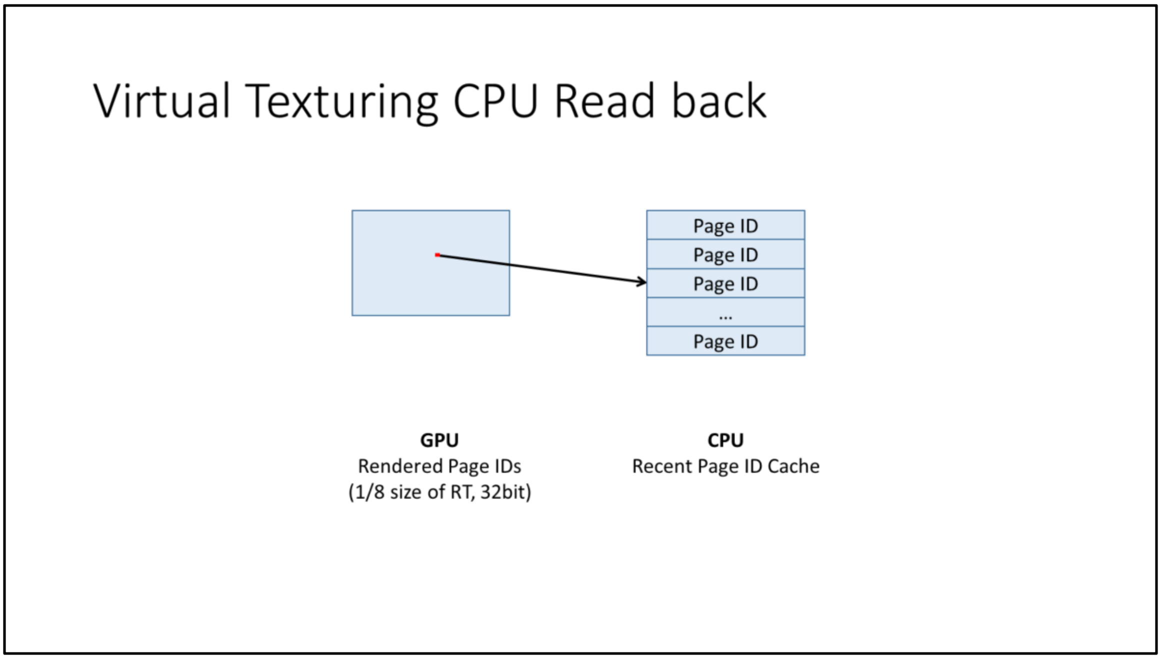
1 Sample + 3 Samples = 4 Samples

This gives us the full terrain material sample in 4 texture samples.



Virtual Texture Page Rendering

- The cached texture pages are rendered using our splat map
- We render up to 6 virtual texture pages per frame
  - 1 Page = 256 x 256 texels + 4 texel border
- Compress to BC format using compute shader
  - 2 x BC1 (Albedo/Smoothness/Specular Occlusion)
  - 1 x BC3 (Normal Map)
- Compress with async compute while rendering the next page
- Total GPU budget of 1ms per frame



Virtual Texturing CPU Read back

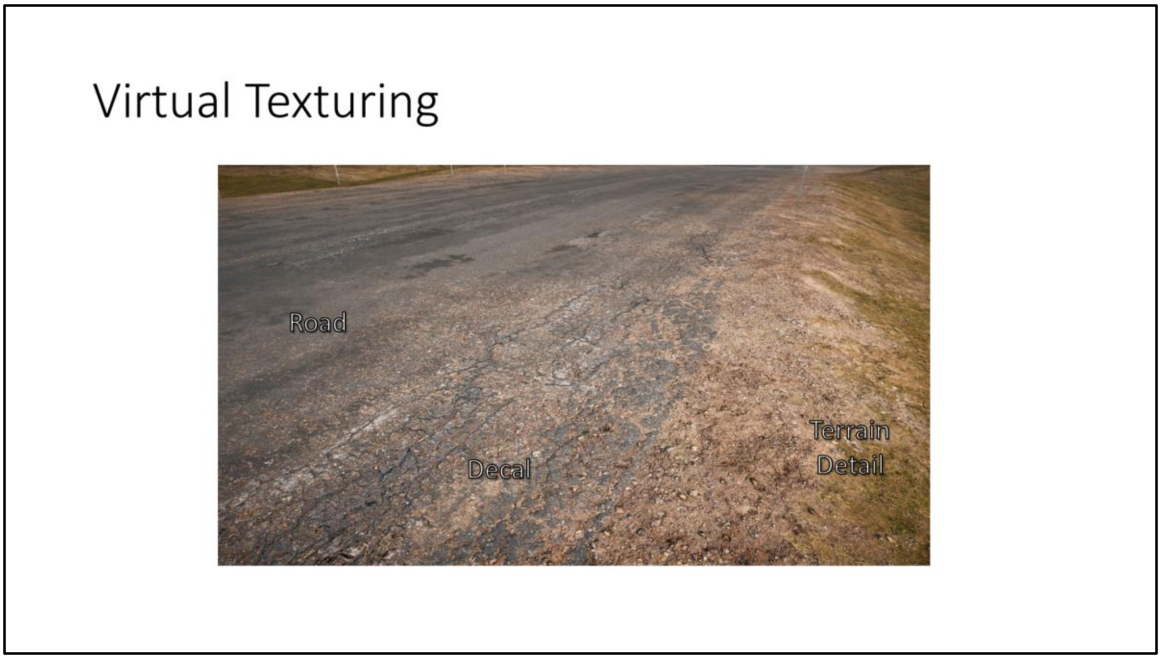
Page ID
Page ID
Page ID
...
Page ID

GPU
Rendered Page IDs
(1/8 size of RT, 32bit)

CPU
Recent Page ID Cache

We always need to keep track of which texture pages need to be updated.
So when we render the terrain we also write the requested pages to a bound UAV.
We then read back the requested pages on the CPU to prioritize which pages to render next.

The UAV is 1/8 the size of the full render target to keep the CPU read back cost low.
We jitter the screen space sample position over 64 frames to ensure that we collect all of the rendered page IDs over time.

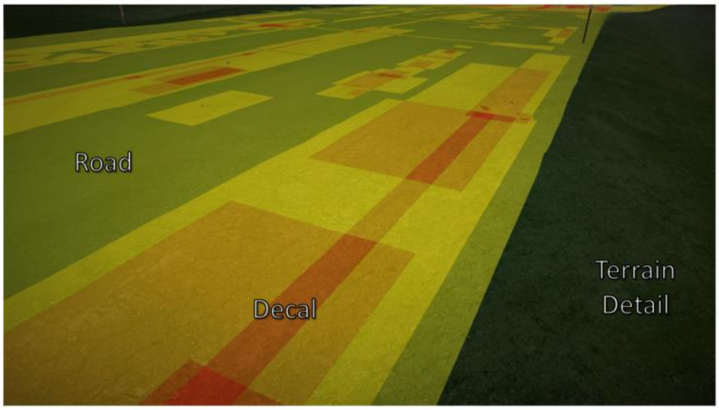


Virtual Texturing

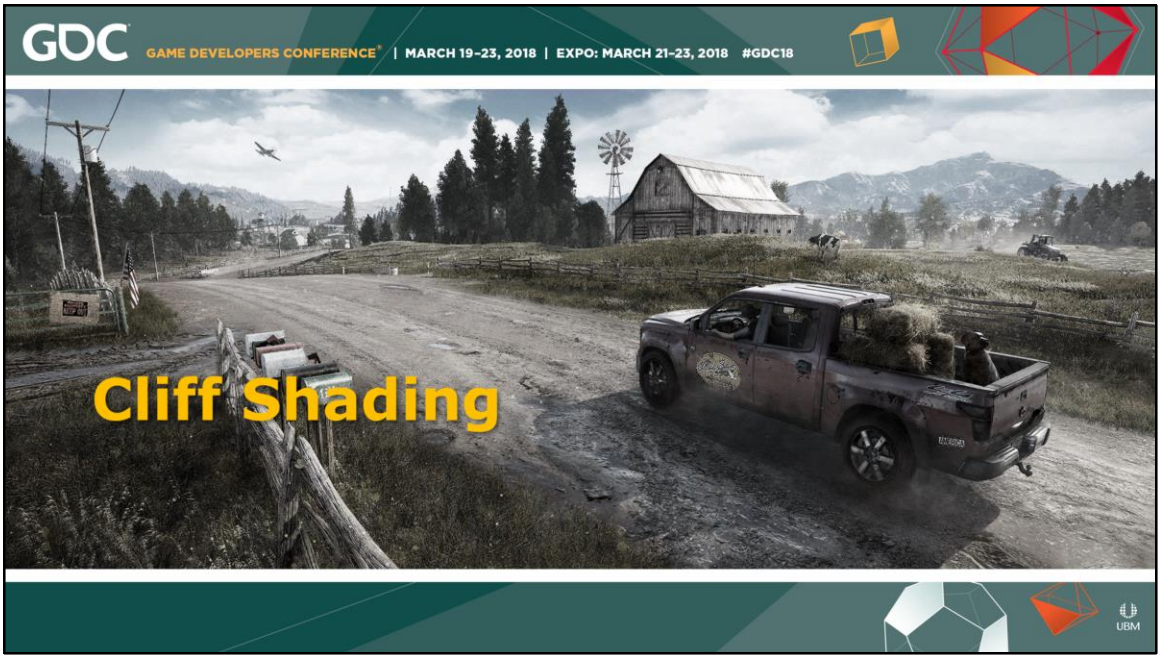
Since we use virtual texturing we can composite multiple layers of road and decal at little extra cost.

You can see the visual results in this screenshot.
A road spline decal is rendered over the terrain detail texture rendering.
And multiple decals are rendered over that.

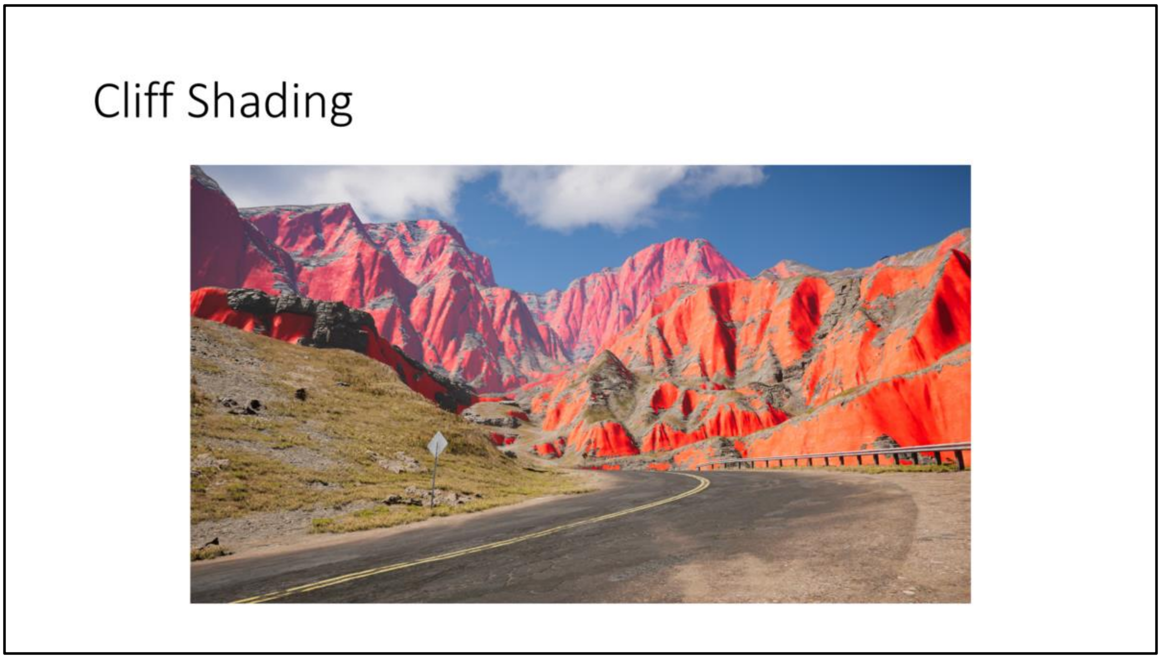
Virtual Texturing



Here is a visualization of the decal overdraw that we are getting practically for free.

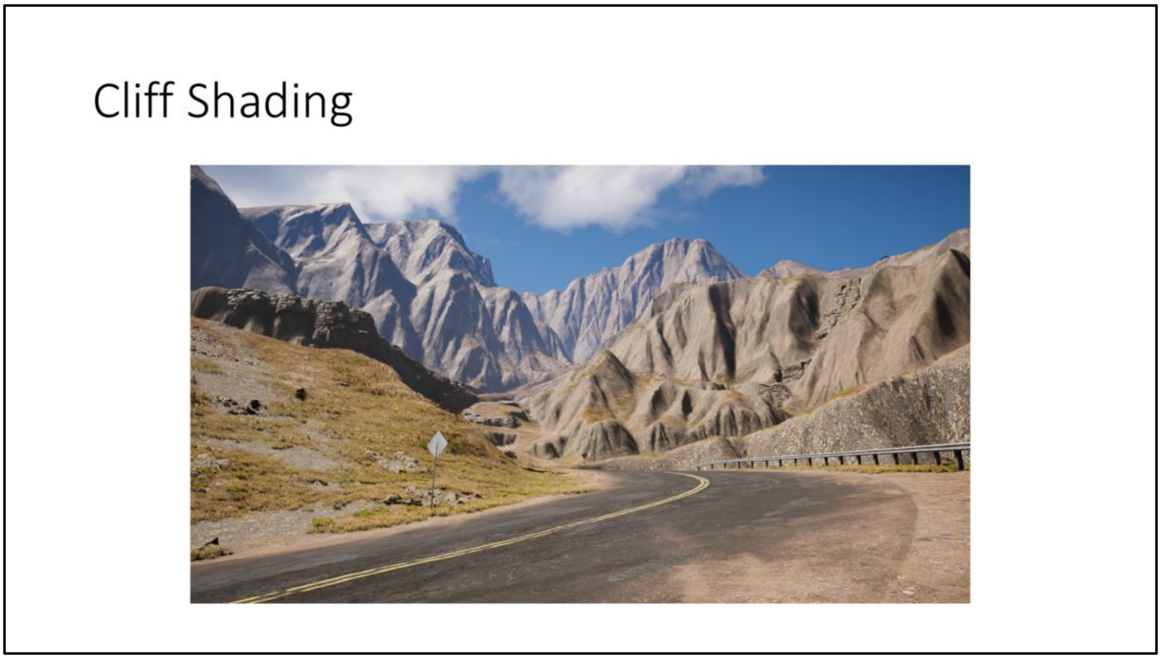


Cliff Shading



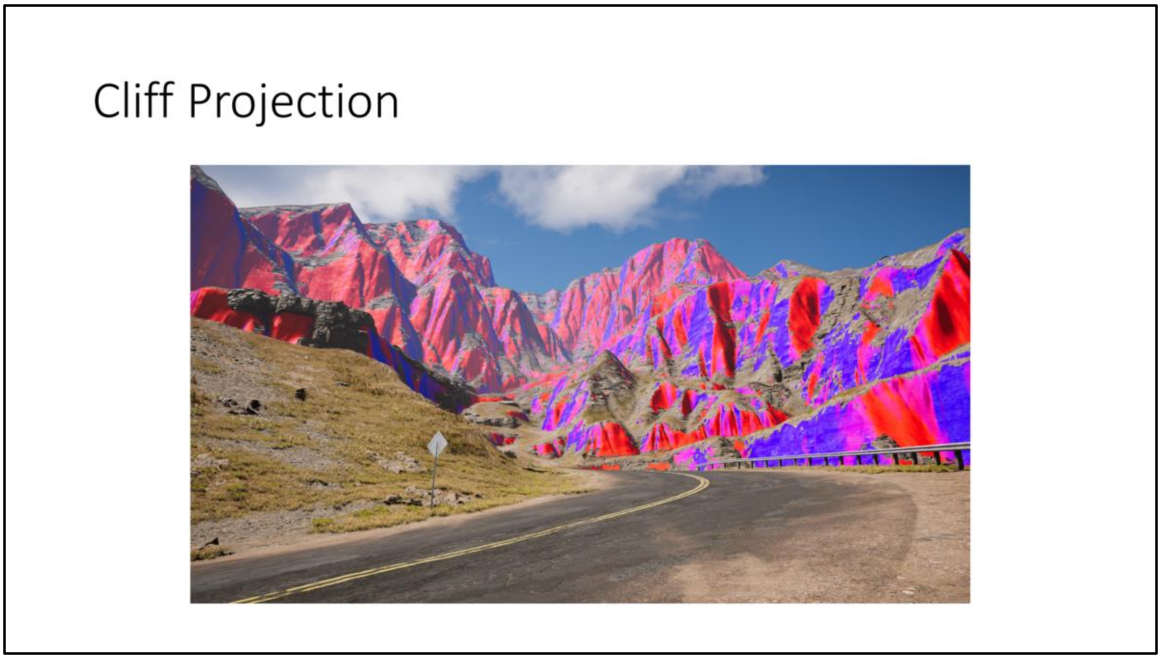
Cliff Shading

All of the terrain texture sampling we have seen so far uses UV coordinates that are derived the (x, y) world position.
This is what we call the top down or ‘z’ projection.
That means that where we have steep slopes the texel resolution will become too low.
Here I’ve marked those areas in red.



Cliff Shading

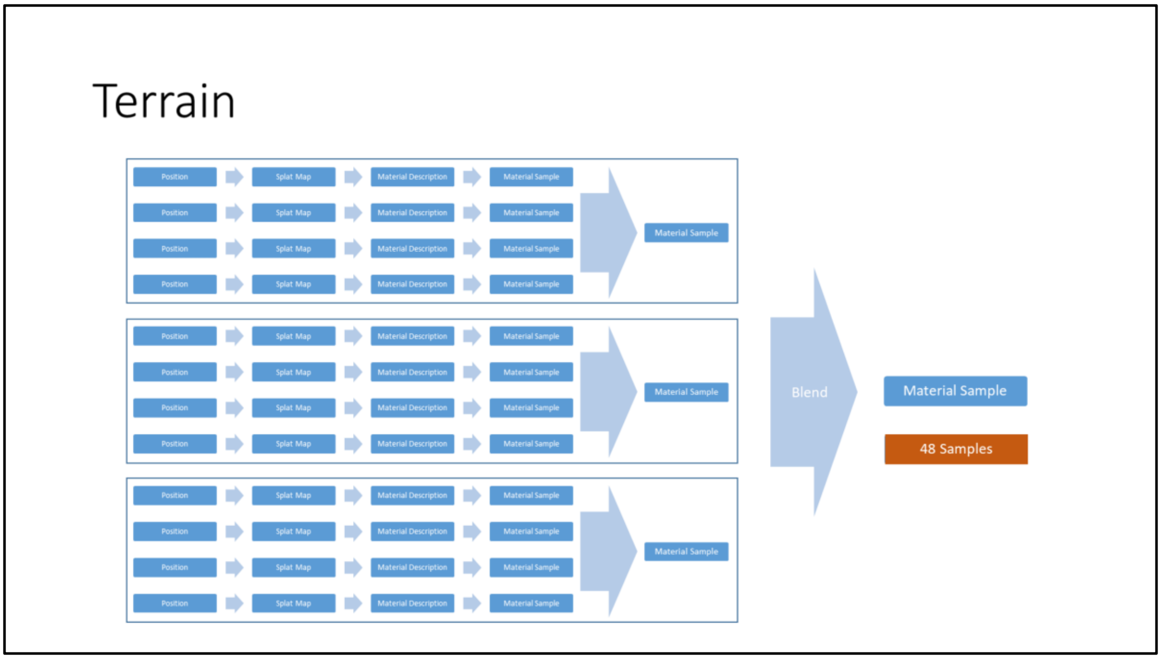
And removing the highlight we can see the problem on cliffs of blurry textures.



Cliff Projection

A classic approach would be to use tri-planar mapping.
Depending on the angle of our normal, we blend 3 projections of the world position aligned to each of the x, y, z axis.

Here on the cliff I’ve marked the where the x and y projections would dominate in red and blue.
We can see where they blend in purple.



Terrain

Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Material Sample

Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Material Sample

Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Position → Splat Map → Material Description → Material Sample
Material Sample

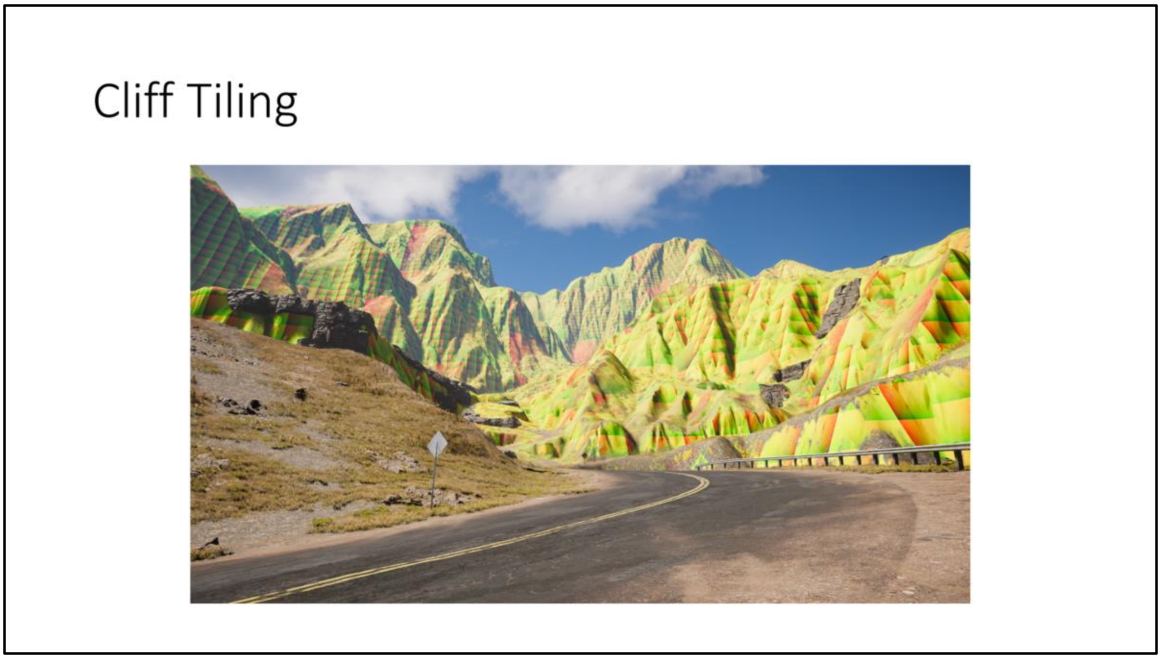
Blend

Material Sample

48 Samples

When we’re shading the cliff we would need to blend our 3 projections for our tri-planar mapping.
So we need to pay the full splat map calculation cost not once, but three times, once for each projection.
Naively this would take 48 texture samples.

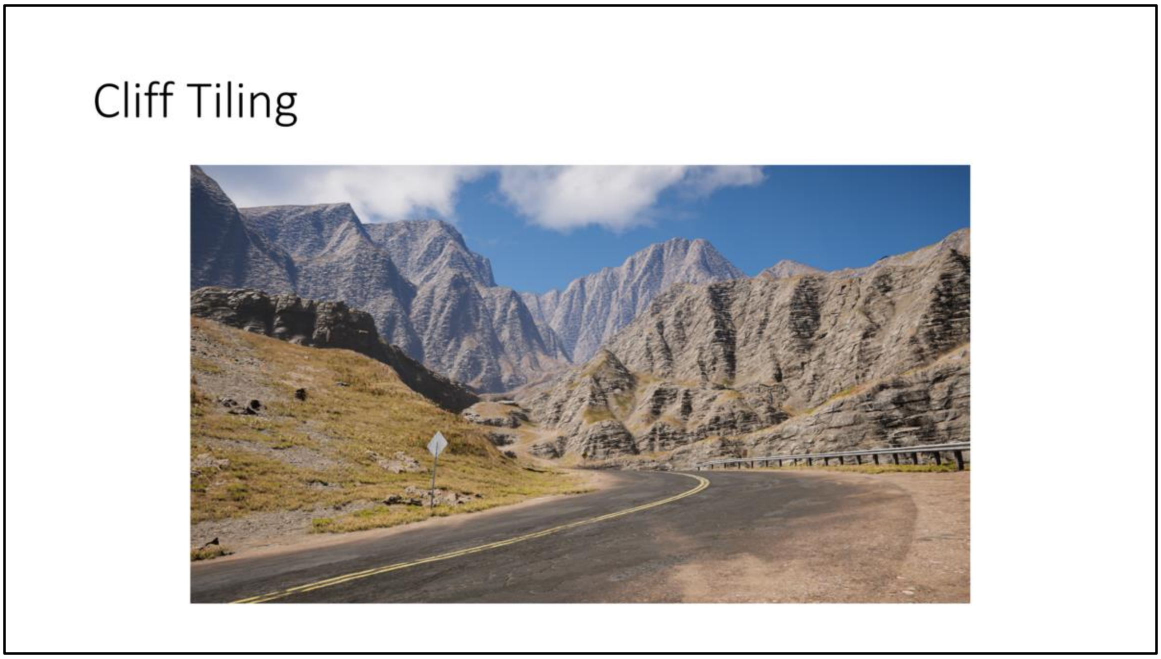
We can actually use virtual texture instead of the z projection to save a few samples here, but we can see that it is still very expensive.



Cliff Tiling

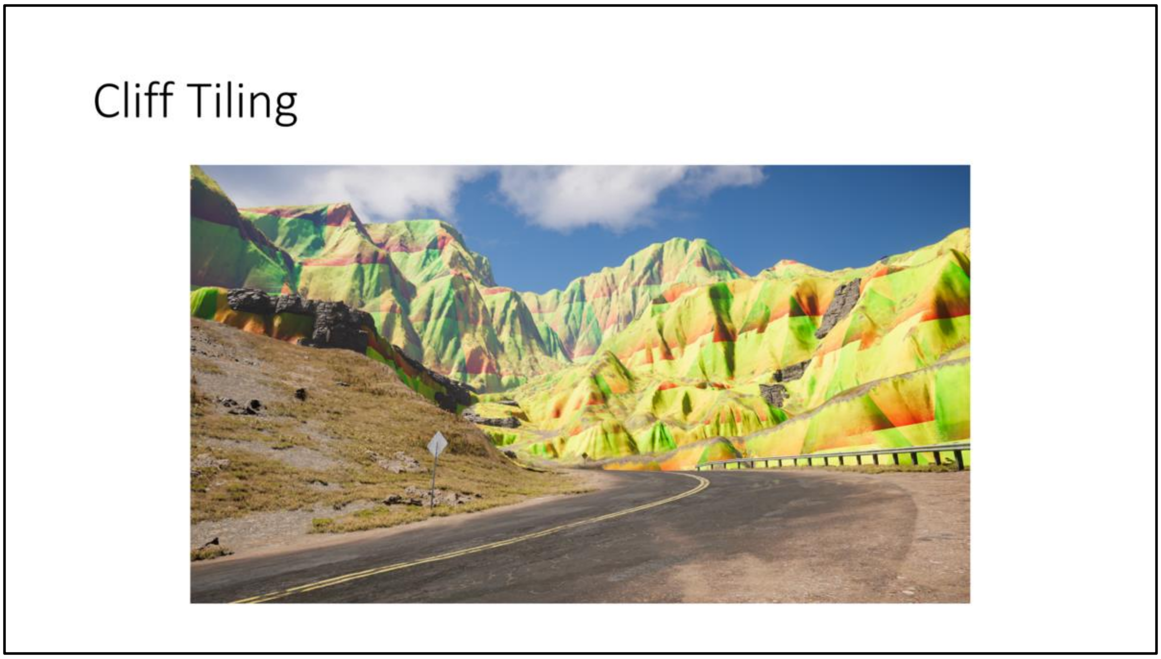
With triplane projection enabled, another issue with cliff shading is that cliffs can be seen a long way from the player.
That can make texture tiling is obvious.

If we use a fixed value for our texture tiling scale, and tune it for the near terrain our UV mapping will look like this.



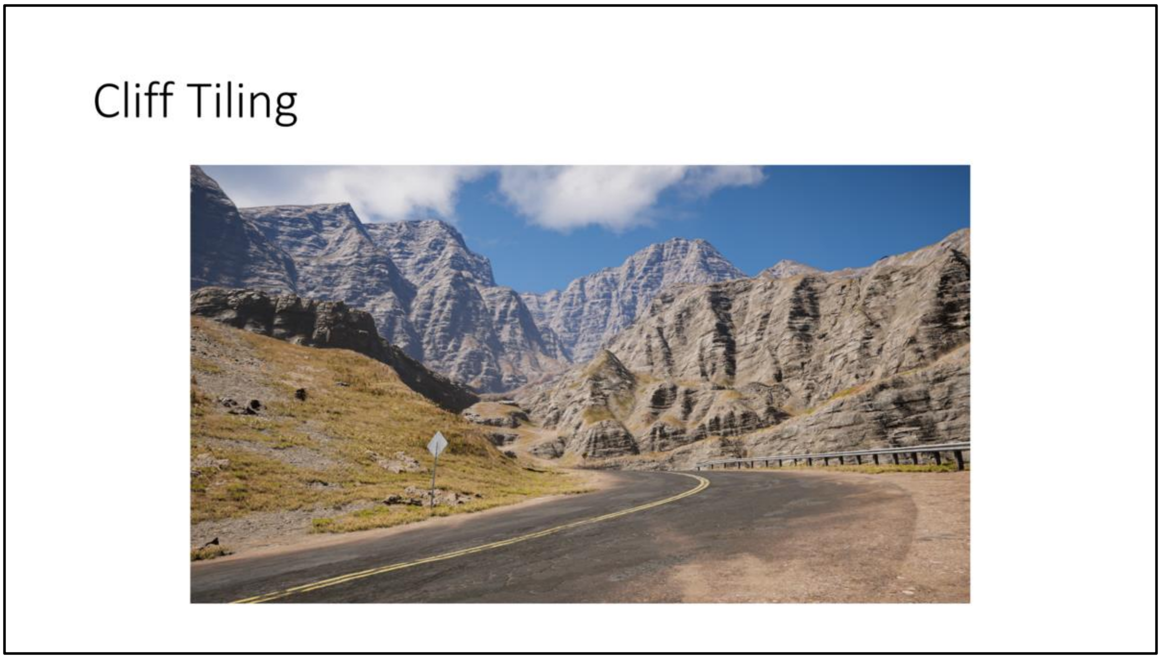
Cliff Tiling

Which gives this result.



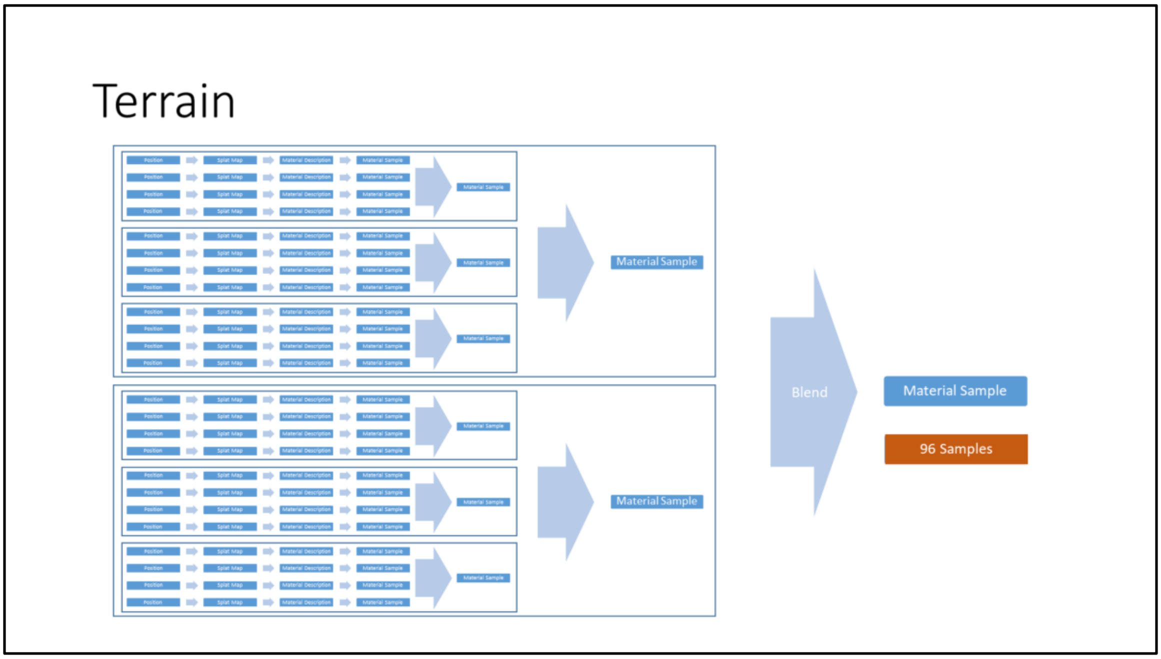
Cliff Tiling

We would prefer to have our UV space being more constant in screen space like this.



Cliff Tiling

Which gives this result.
This appears plausible because some terrain and cliff textures can be quite fractal and work at multiple scales.

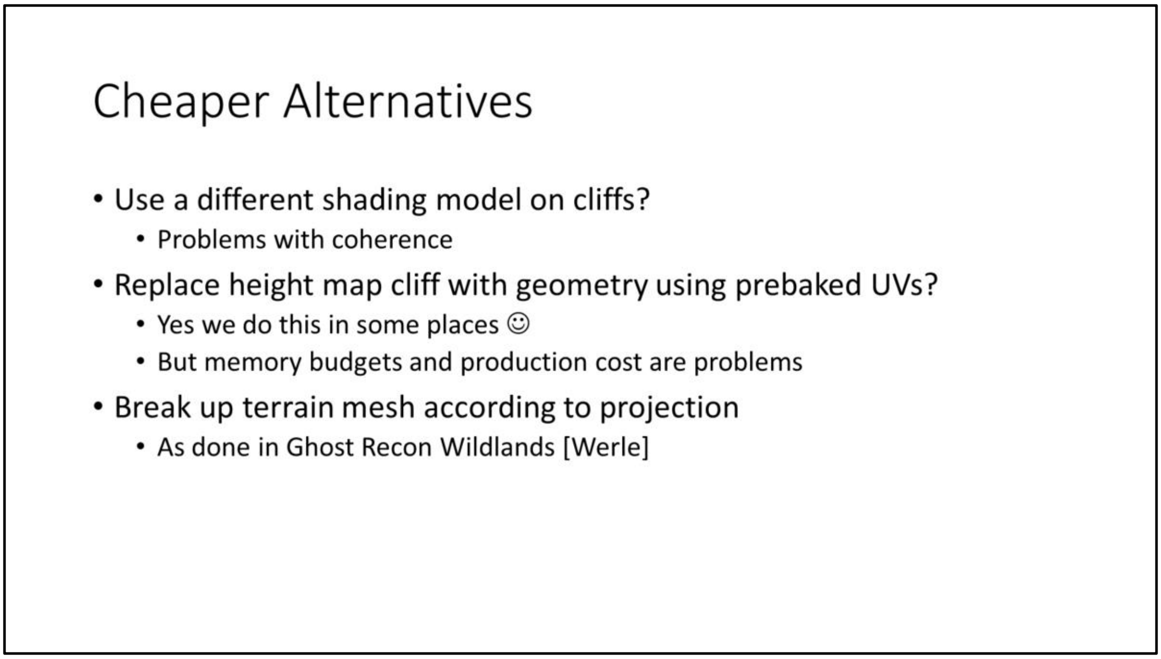


Terrain

However changing the tiling over distance would cause texture discontinuities.
We would need to blending the results of 2 different scales.

So when we’re shading cliffs like this we still need to blend our 3 projections for our tri-planar mapping.
But now we also have to do this for two tiling scales at each pixel

That’s 96 texture samples!

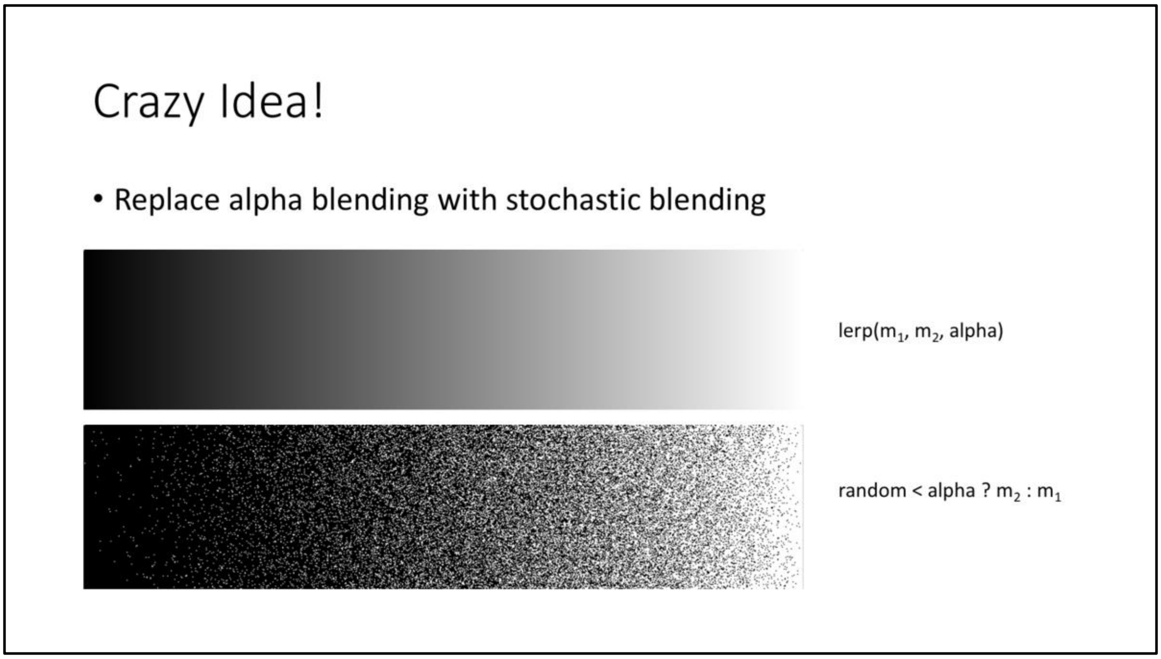


Cheaper Alternatives

- Use a different shading model on cliffs?
  - Problems with coherence
- Replace height map cliff with geometry using prebaked UVs?
  - Yes we do this in some places ☺
  - But memory budgets and production cost are problems
- Break up terrain mesh according to projection
  - As done in Ghost Recon Wildlands [Werle]

We considered a number of cheaper approaches…

Another solution that we want investigate is to generate a virtual texture atlas that can accommodate cliffs.



Crazy Idea!

- Replace alpha blending with stochastic blending

lerp(m_1, m_2, alpha)

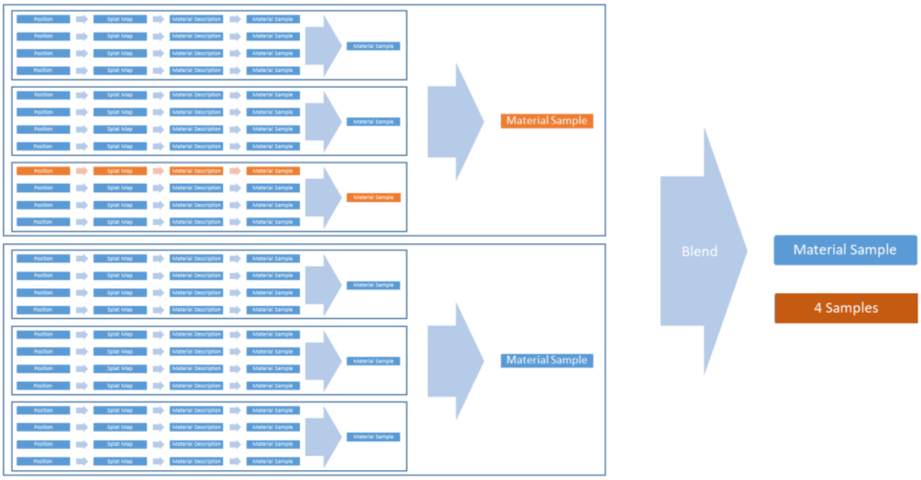
random < alpha ? m_2 : m_1

One slightly insane idea is to blend our different shading inputs using a stochastic approach.

You can see here a blend between black and white.
On the top we use a standard lerp alpha blend.
On the bottom we use stochastic blending.

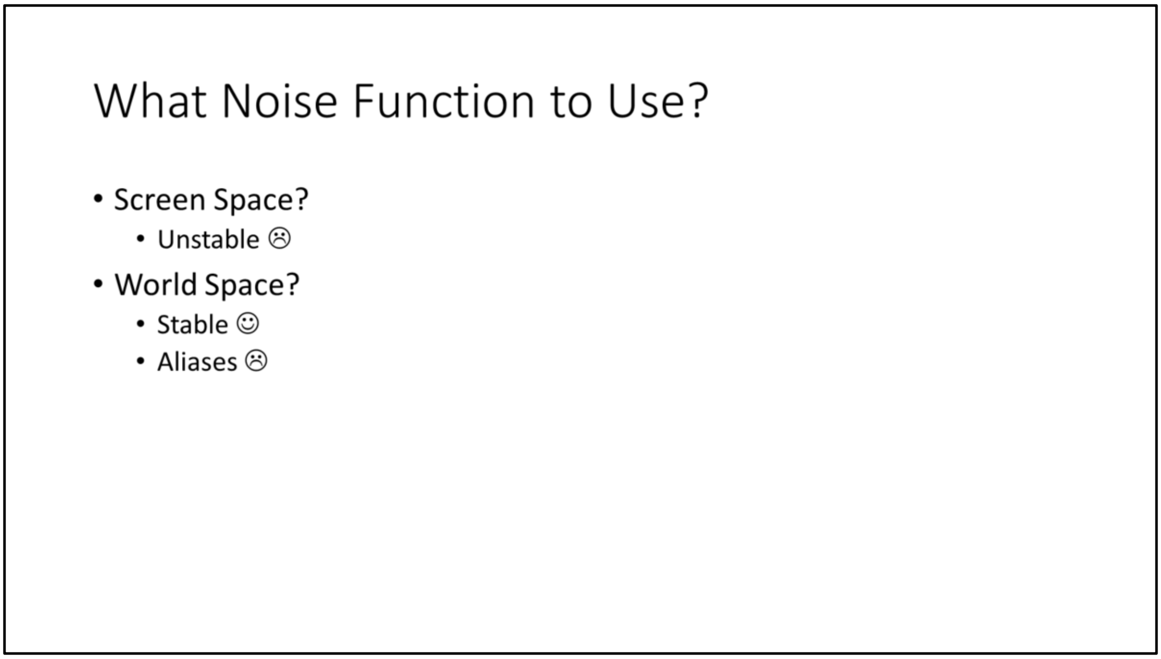
The stochastic results are incorrect at every pixel but are correct “on average”.
We are trading maths for noise.

Stochastic Cliff Shading



For terrain instead of sampling multiple splat map samples per pixel and blending, we could chose one splat map sample per pixel.
Which one we chose from our list of inputs here can change randomly from pixel to pixel.
We just need weight our random choice, according to the blend factors so that we choose the dominant weighted samples more often.

So we tried that!
And… it didn't look very good…
It was noisy and unstable.
But it wasn't quite *terrible* enough to give up!



What Noise Function to Use?

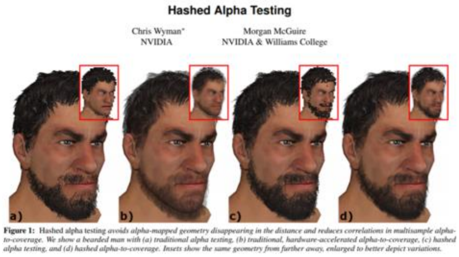
- Screen Space?
  - Unstable ☹
- World Space?
  - Stable ☺
  - Aliases ☹

One of the ways to change the quality of the output is to change the noise function that is providing our per pixel random numbers.
We wanted a good noise function.

Screen space noise was unstable under camera motion.
World space noise was stable, but aliased.

What Noise Function to Use?

- “Hashed Alpha Testing” Paper from I3D 2017 by Wyman and McGuire
  - Quite stable☺
  - Per-pixel noise resolution☺



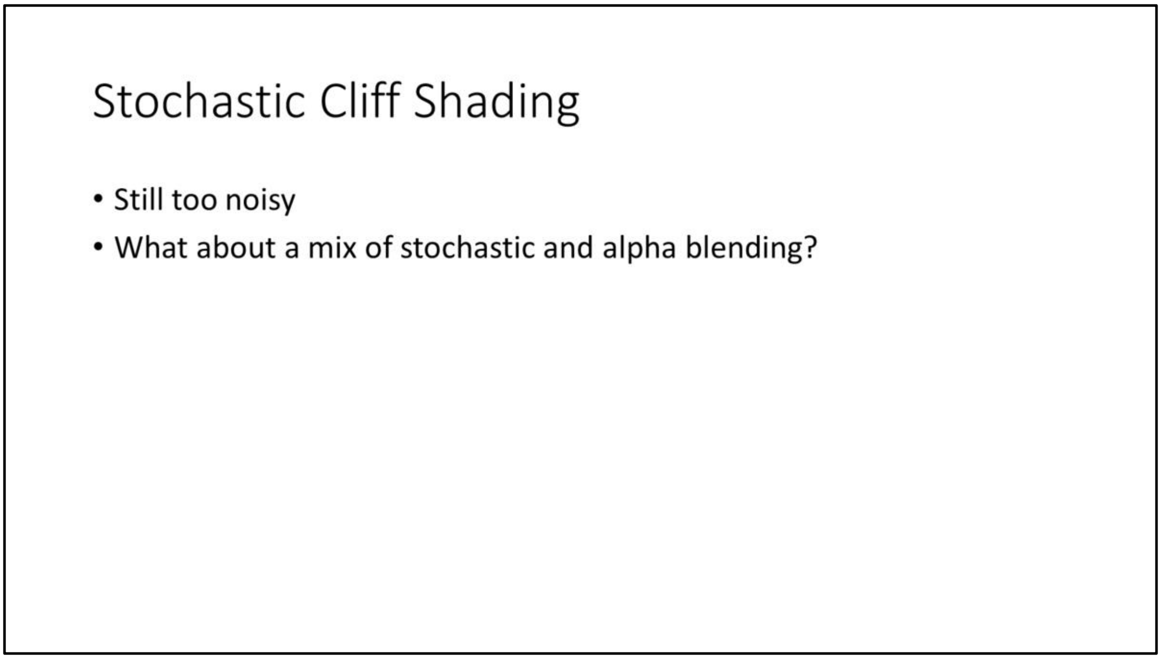
Hashed Alpha Testing

Chris Wyman*
NVIDIA

Morgan McGuire
NVIDIA & Williams College

Figure 1: Hashed alpha testing *avoids alpha-mapped geometry disappearing in the distance and reduces correlations in multisample alpha-to-coverage. We show a bearded man with (a) traditional alpha testing, (b) traditional, hardware-accelerated alpha-to-coverage, (c) hashed alpha testing, and (d) hashed alpha-to-coverage. Insets show the same geometry from further away, enlarged to better depict variations.*

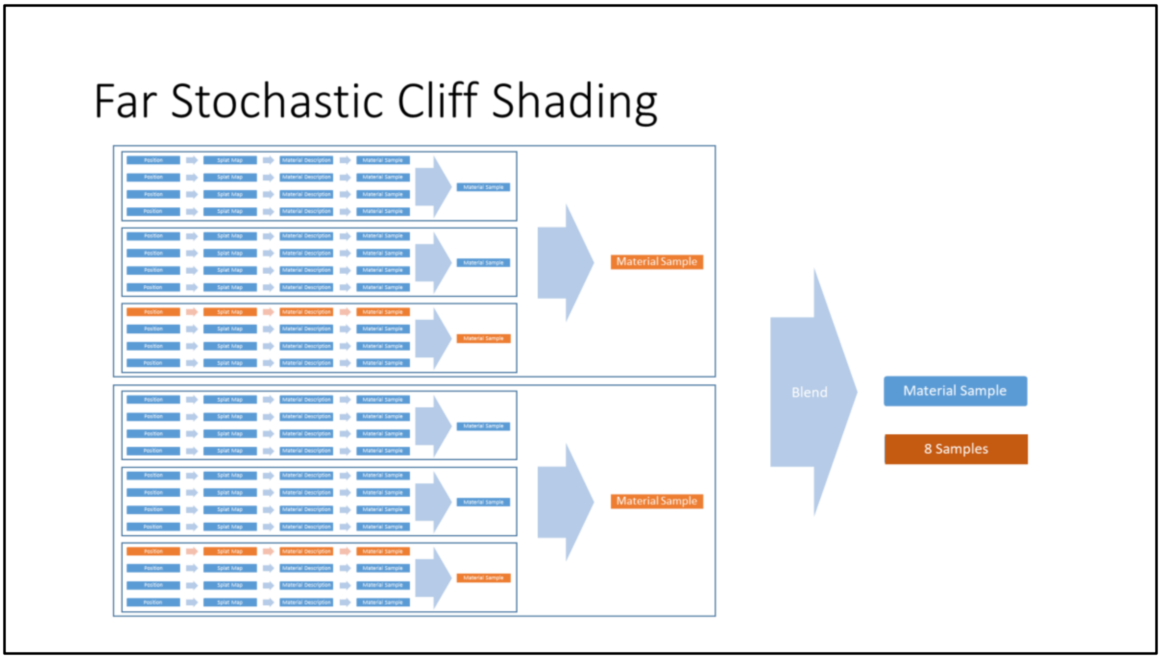
Then we read the “Hashed Alpha Testing” paper from Wyman and McGuire.
It outlines a method of stochastic alpha testing.
It has an insight into how to stabilize the noise from stochastic sampling by using a position hash combined with screen gradients.
It is quite stable AND it doesn’t alias.



Stochastic Cliff Shading

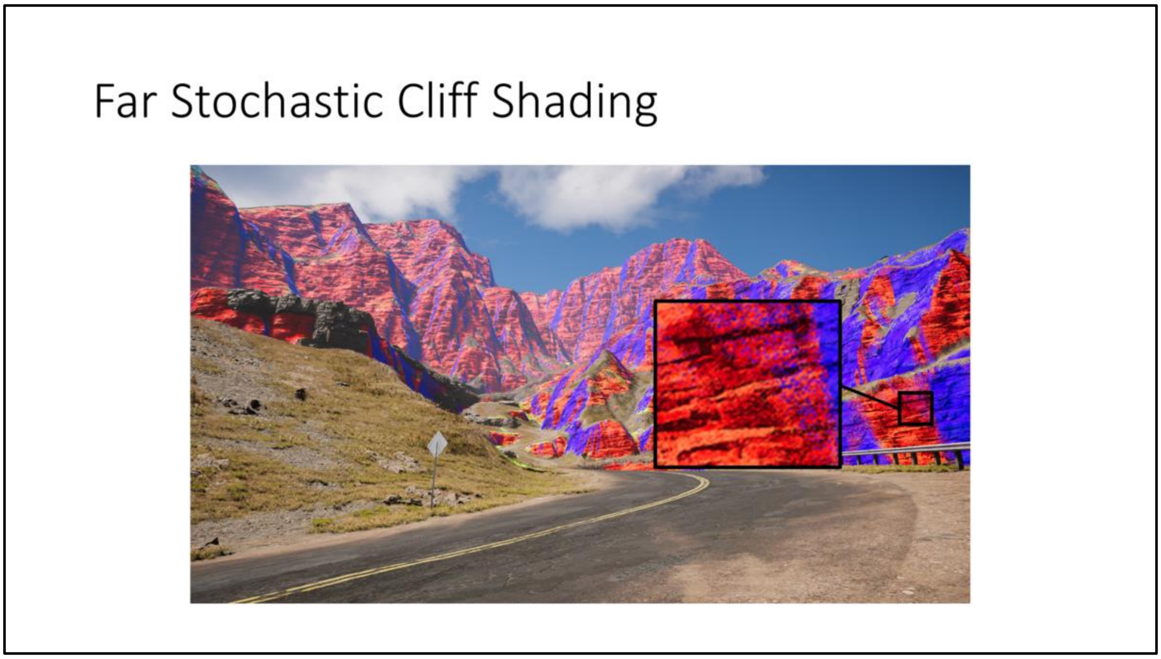
- Still too noisy
- What about a mix of stochastic and alpha blending?

Still the result was too noisy.
So we looked at mixing stochastic and alpha blending approaches.
We came up with two scenarios…



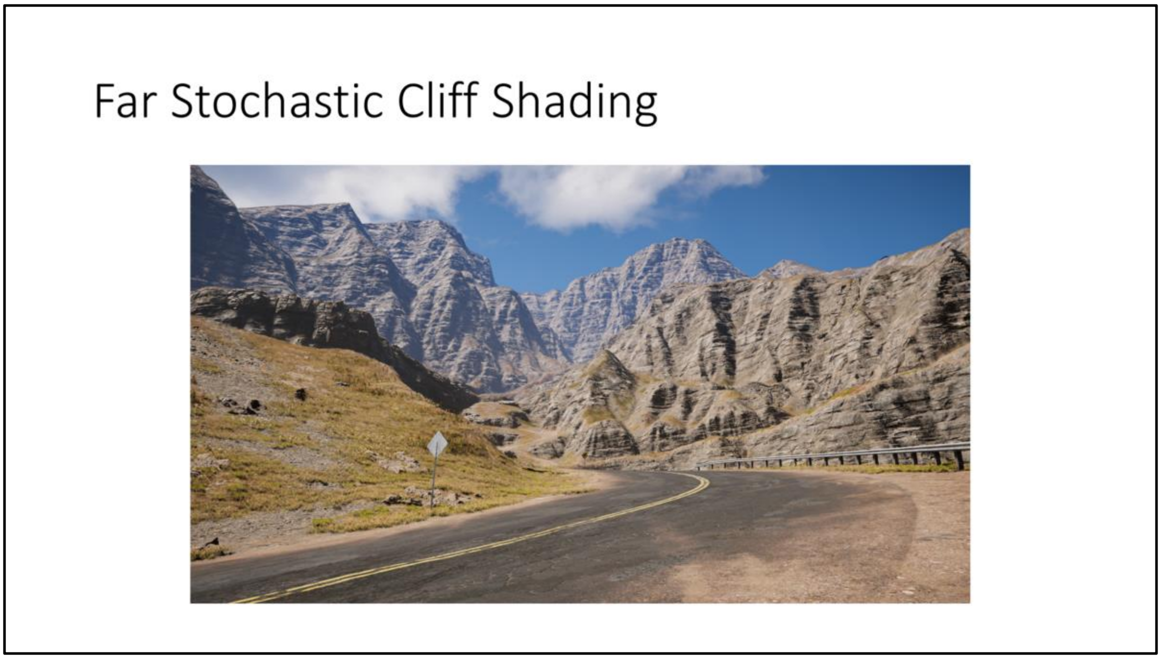
Far Stochastic Cliff Shading

First consider cliff shading at some large-ish distance from the camera.
We use two tiling scales that are fully alpha blended.
But each is a single stochastic sample based on 3 projections * 4 materials.
This reduces the number of texture samples to 8.



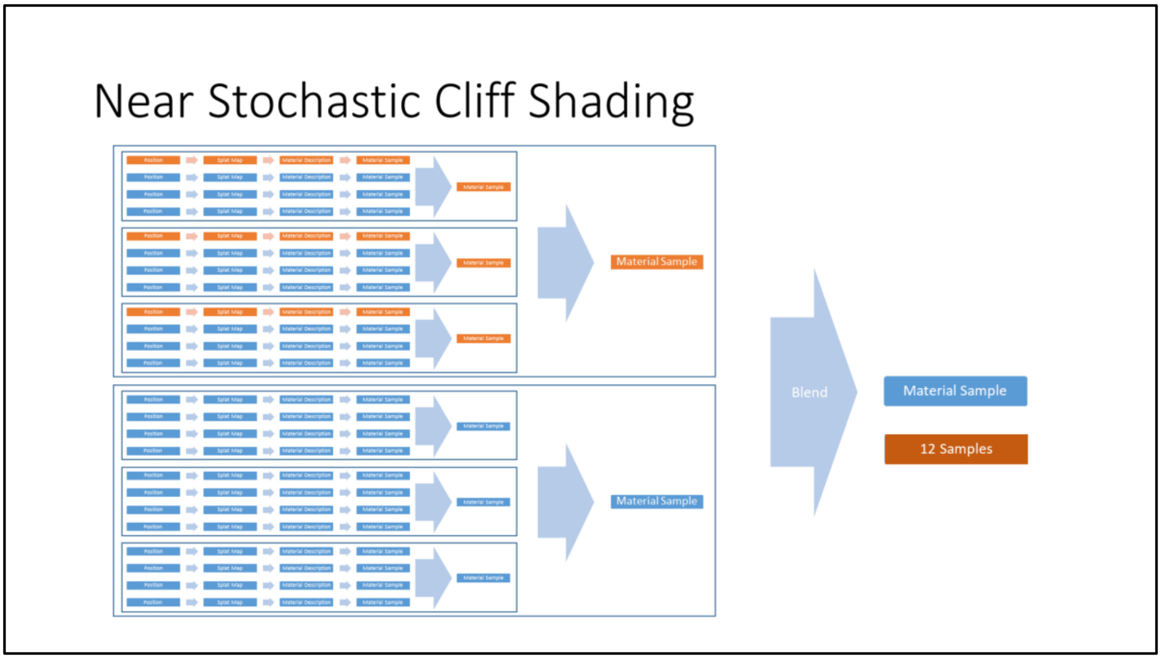
Far Stochastic Cliff Shading

This image shows a the scene being rendered with far stochastic cliff shading but using a flat color to replace the projection axis.
If we don’t look closely the colors appear to blend, but zooming in shows that we have a noisy pattern.



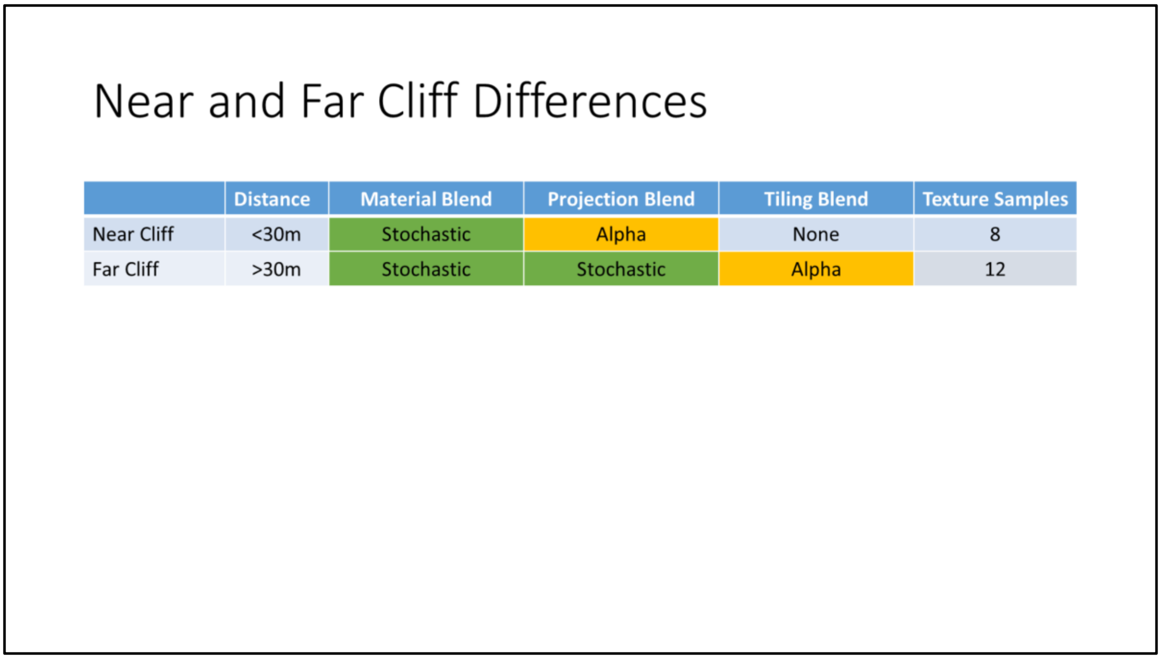
Far Stochastic Cliff Shading

But because in our color data we have very low contrast we don’t feel the noise in the final image.



Near Stochastic Cliff Shading

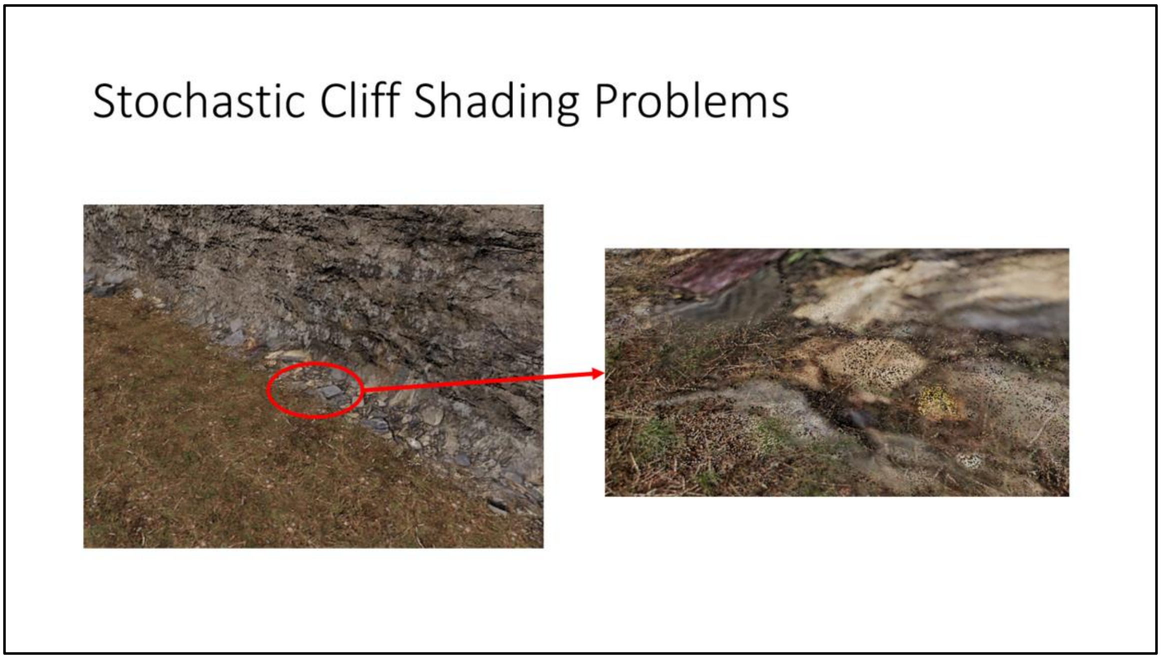
Now consider cliffs close to the camera.
At this distance we don’t need a second tiling sample, but the noise from using a stochastic tri-planar blend is too obvious.
So we use a stochastic material sample for 3 projections and then alpha blended.
This reduces the number of texture samples to 12.



Near and Far Cliff Differences

	Distance	Material Blend	Projection Blend	Tiling Blend	Texture Samples
Near Cliff	<30m	Stochastic	Alpha	None	8
Far Cliff	>30m	Stochastic	Stochastic	Alpha	12

So this table summarizes the different approaches for near and far cliff.
For the near cliff we only use the stochastic blend for the 4 material samples, but still do full tri-planar blending.
For the far cliff we also use stochastic blending between projections.

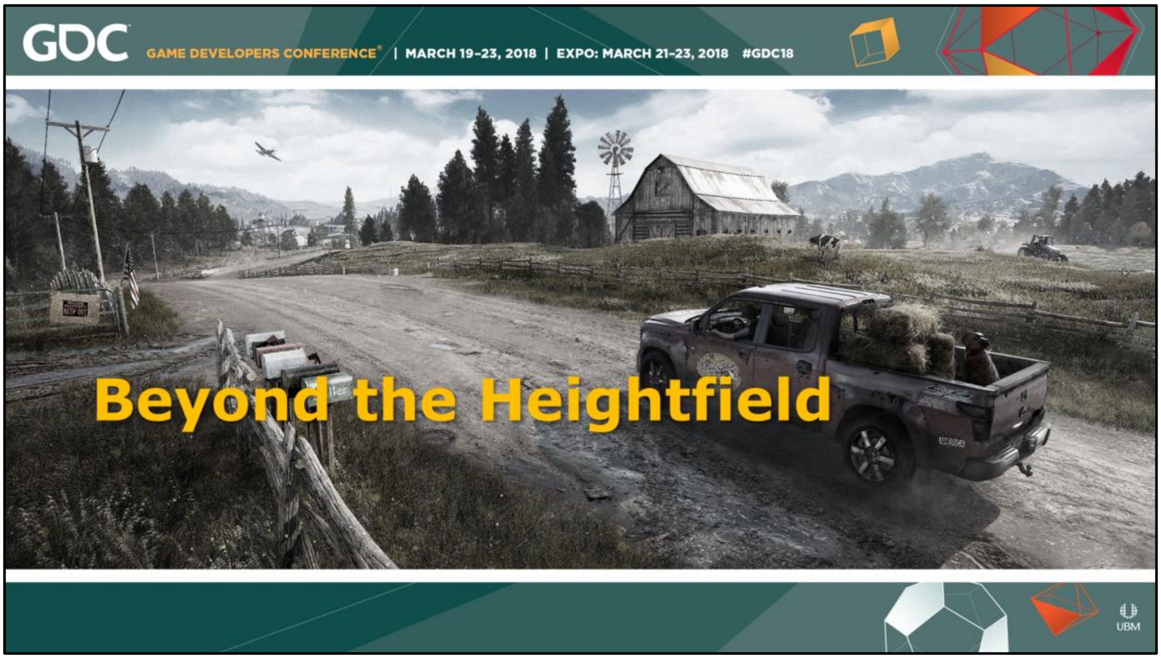


Stochastic Cliff Shading Problems

Generally we don’t see problems but it’s not perfect.

Any noise we see for this technique is concentrated in areas where materials change. If we really zoom in to inspect a high contrast texture transition (here from rock to grass at a cliff border) we can see it.

On Far Cry 5 the artists are in control over where the technique is used, and can avoid the worst cases.

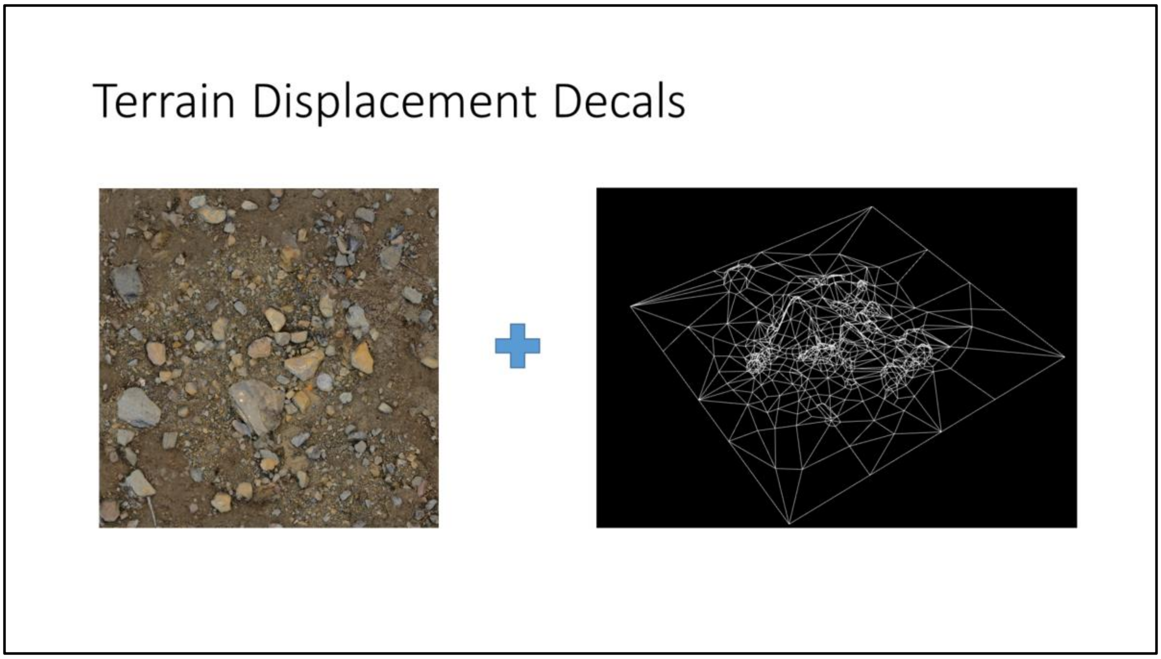


GDC GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18

Beyond the Heightfield

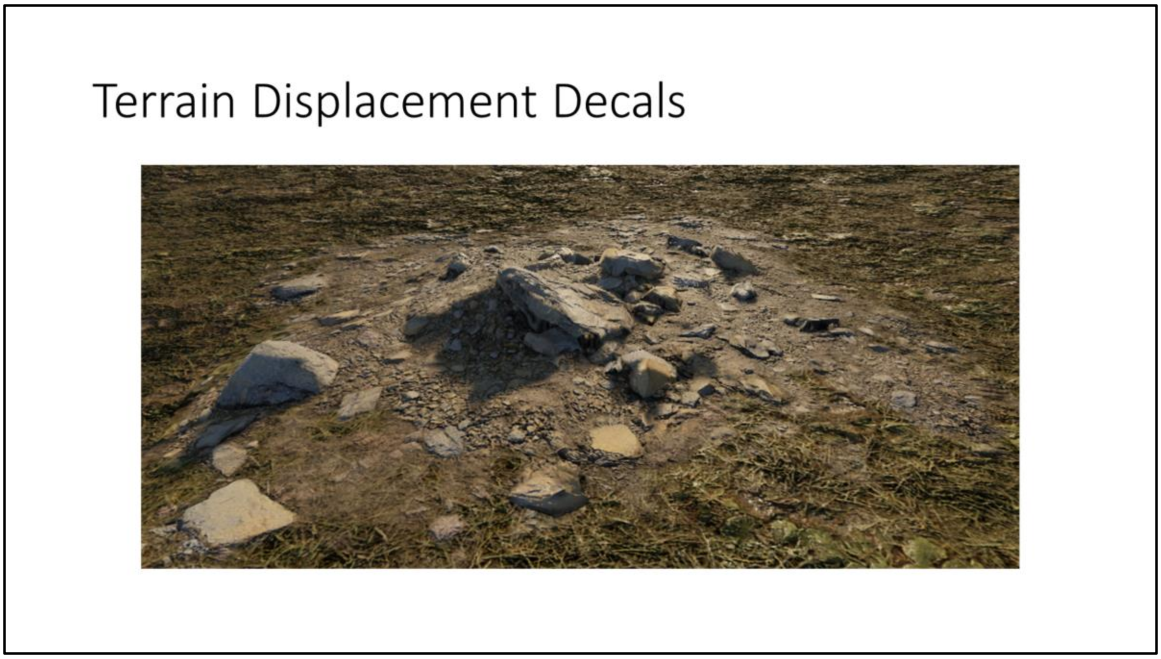
UBM

In the next section I`ll describe how we augmented the heightfield geometry with other art created or procedural geometry.



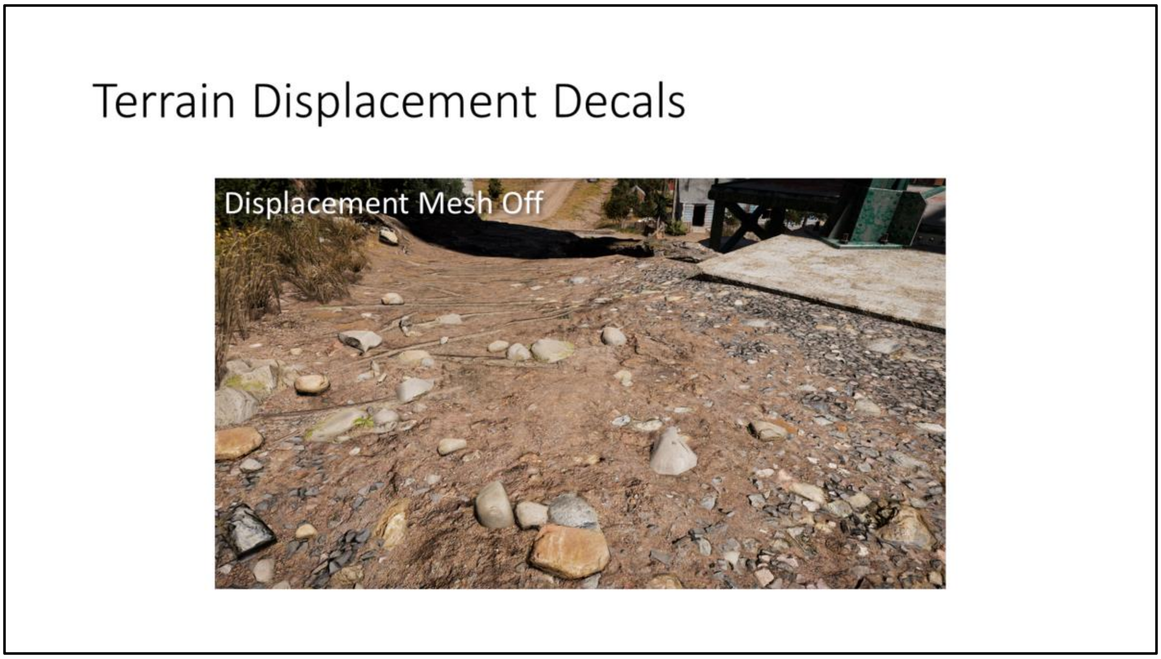
Terrain Displacement Decals

On Far Cry 5 we extended our virtual texture decal system with what we called Terrain Displacement Decals.
These are ordinary decals that render into the virtual texture.
But they also have a matching simple mesh that we will use to give shape to the terrain.



Terrain Displacement Decals

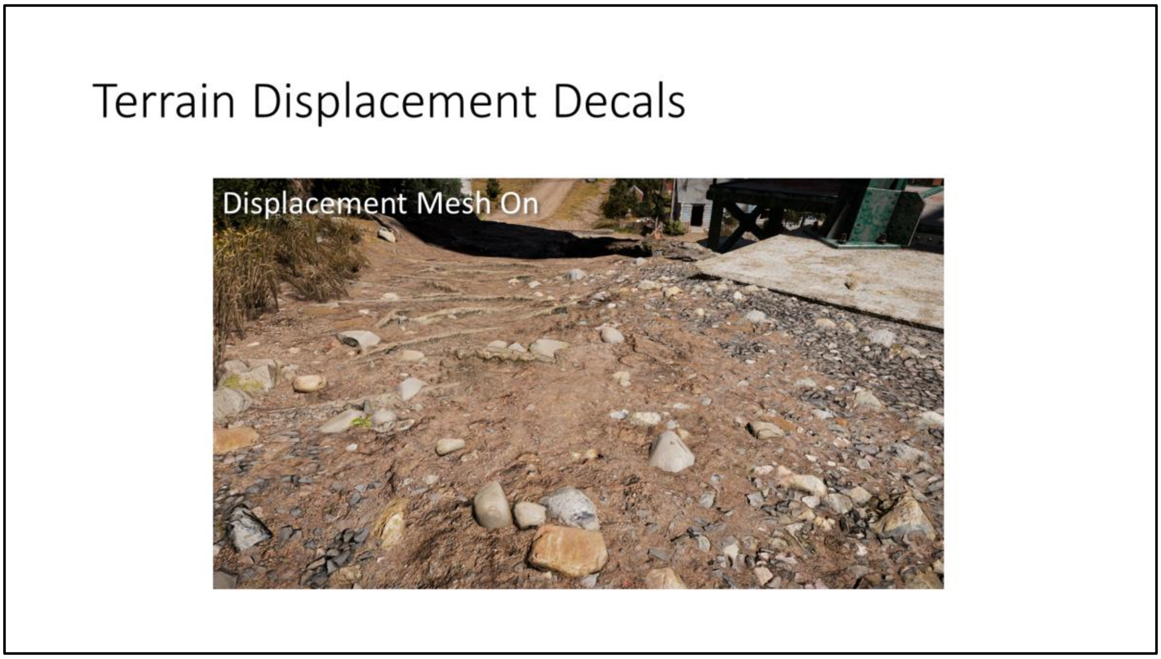
The mesh is placed on the terrain to align with the matching decal.
Then the pixel shader samples the terrain virtual texture which contains the decal.
The result blends seamlessly into the terrain because the decal is already alpha blended into the virtual texture.



Terrain Displacement Decals

We use this everywhere in the world for Far Cry 5 for rock decoration, tyre tracks, etc.

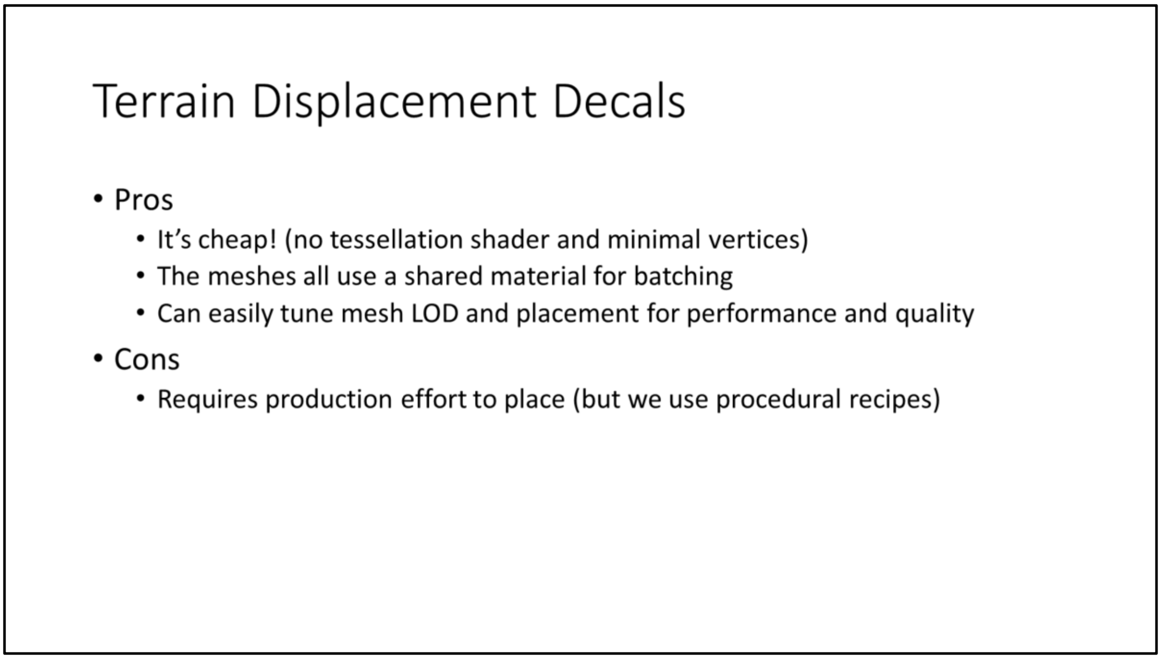
Here is a typical example scene.
With the displacement meshes off, we can already see the rocks and tree roots that have been rendered into the virtual texture.



Terrain Displacement Decals

But with the meshes on we can now see the shape of these objects more clearly.

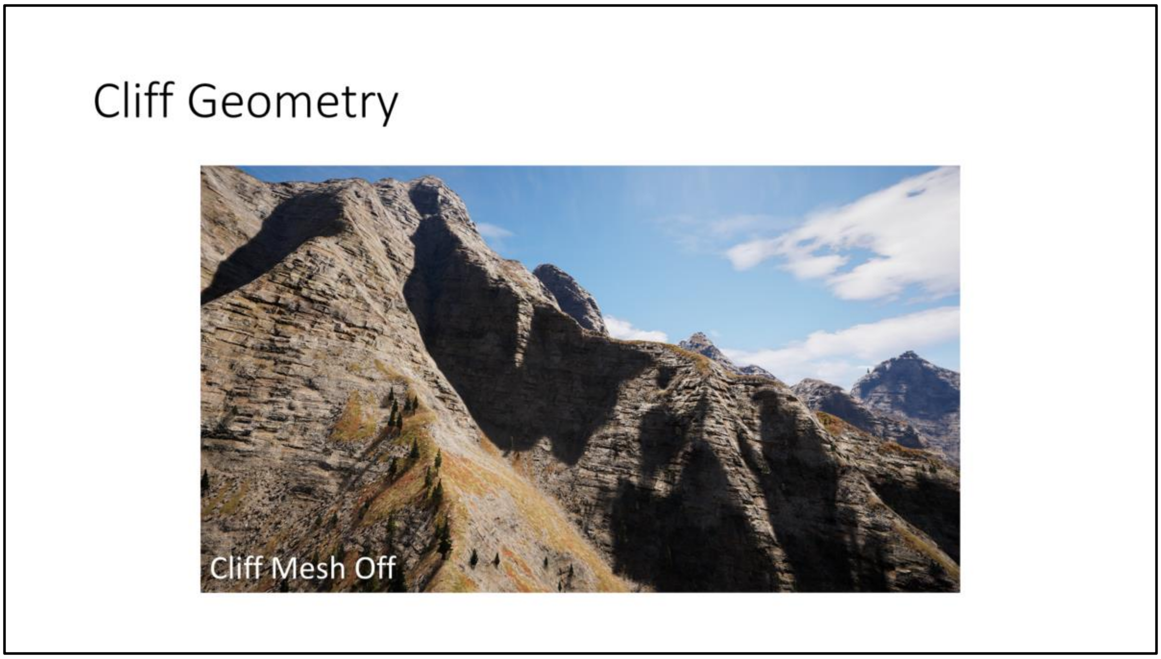
This approach is a substitute for terrain tessellation displacement mapping.
We implemented that approach too but rejected it due to cost.



Terrain Displacement Decals

- Pros
  - It’s cheap! (no tessellation shader and minimal vertices)
  - The meshes all use a shared material for batching
  - Can easily tune mesh LOD and placement for performance and quality
- Cons
  - Requires production effort to place (but we use procedural recipes)

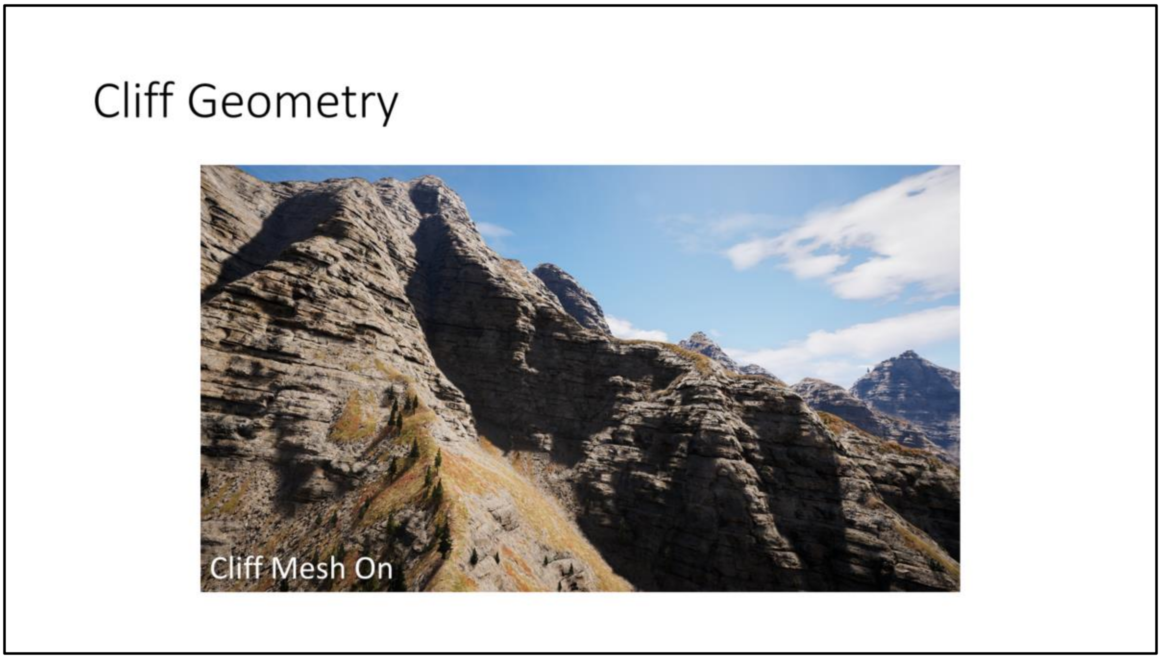
There are pros and cons to using displacement decals instead of terrain tessellation…



Cliff Geometry

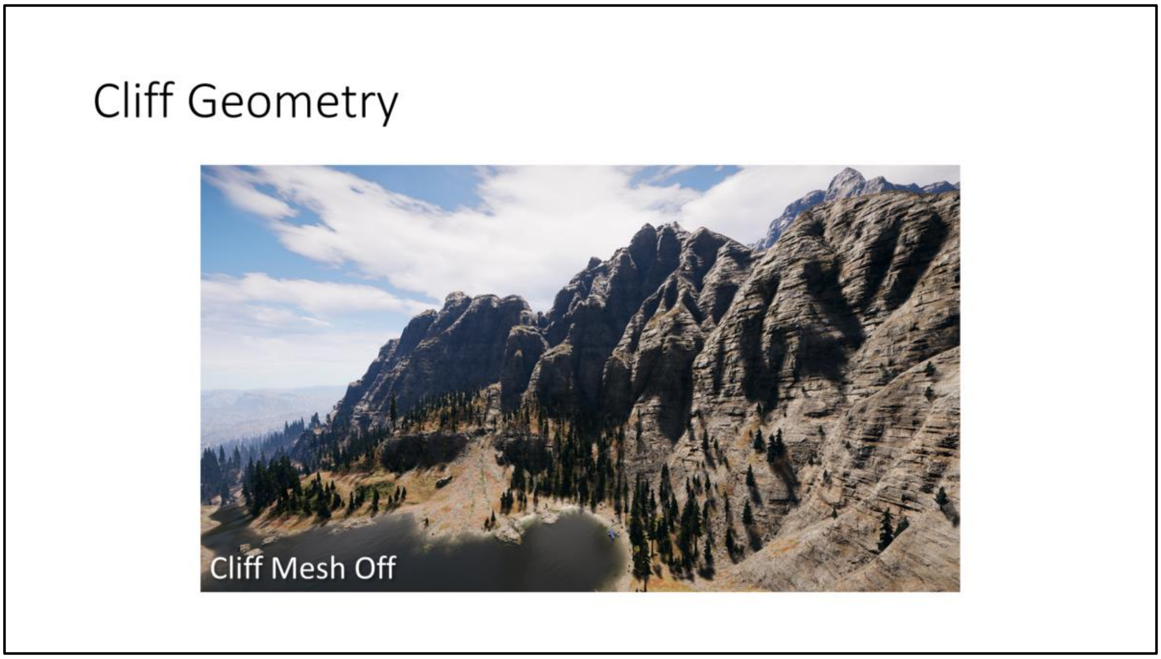
Cliff Mesh Off

Far Cry 5 added a procedural cliff pipeline that adds unique cliff geometry to the world.
The pipeline creates mesh in areas that it detects as cliff, and we render these meshes.
It adds shape and variety and helps reduce a little the “height map” feel.
This is with the geometry off.



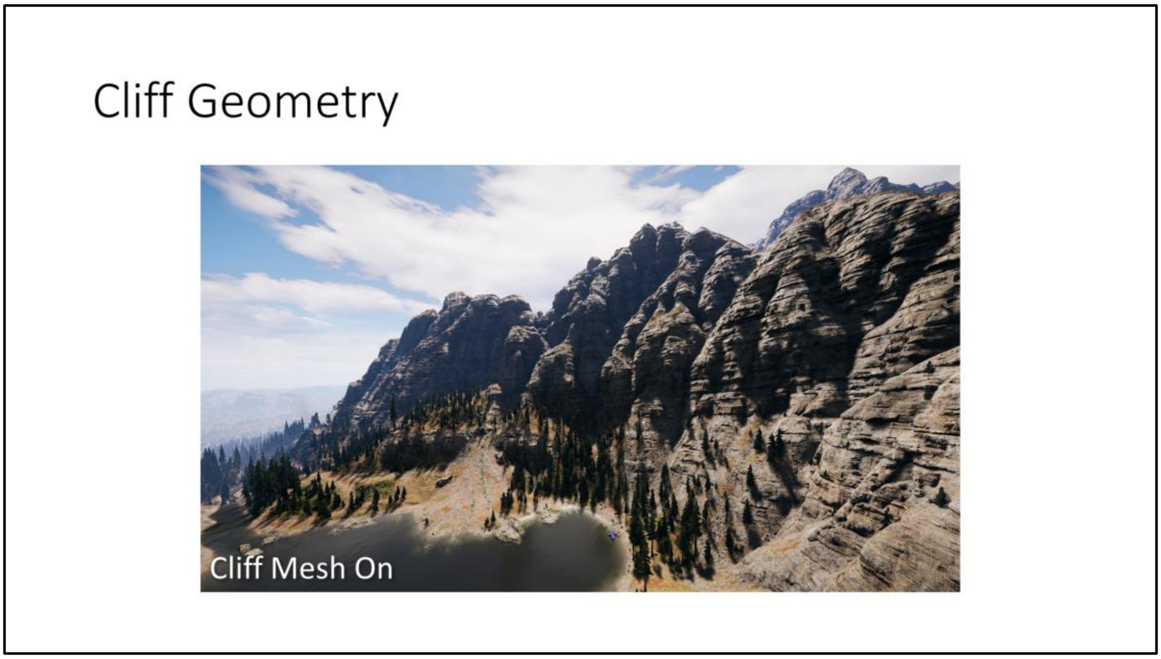
Cliff Geometry

This is with the geometry on.
The unique geometry has only vertex position and normal and so is quite lightweight.



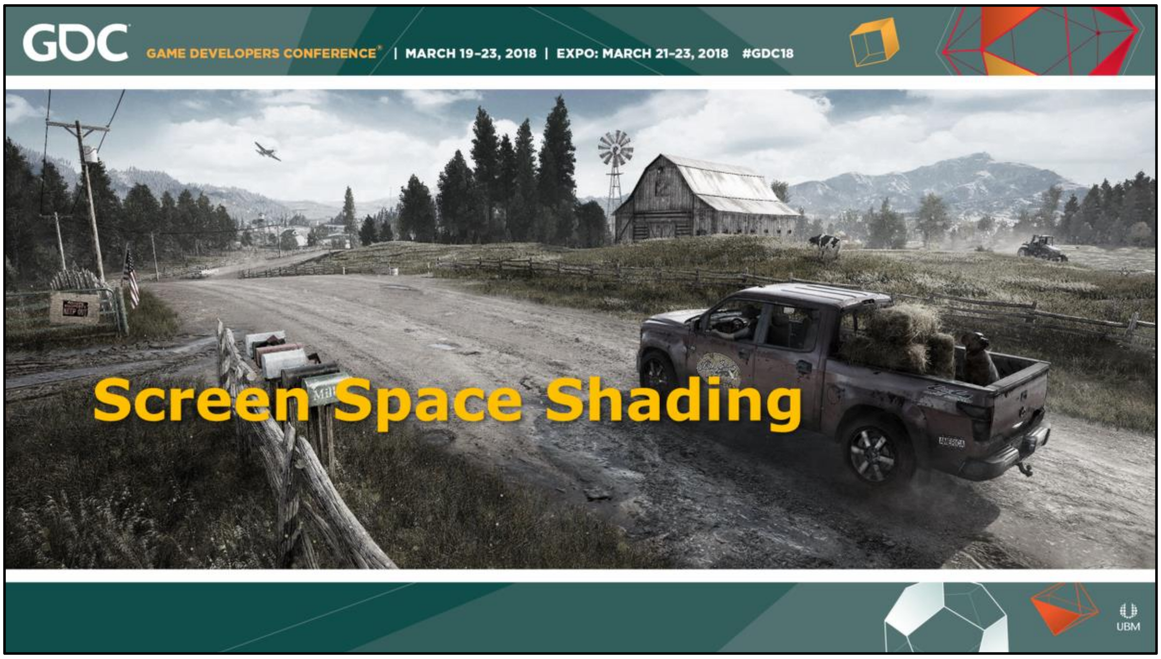
Cliff Geometry

Here is another example with and without the extra cliff geometry.



Cliff Geometry

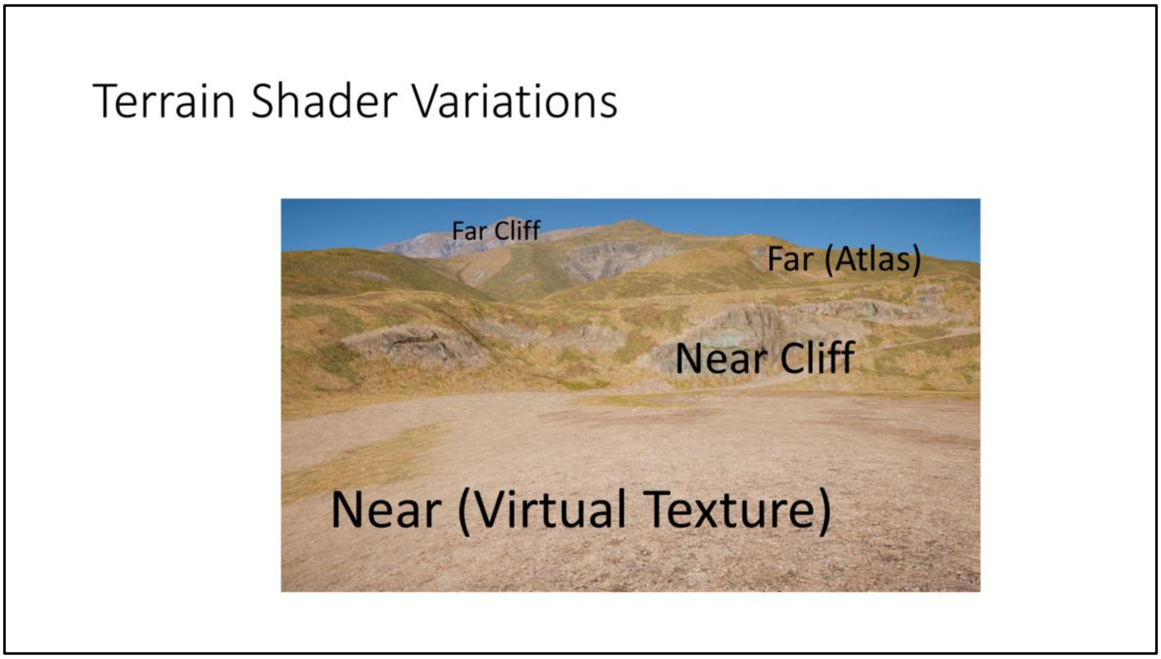
The crucial thing is that we can shade the geometry exactly as we do for the terrain heightmap.
So we use the same shader/terrain textures etc.
That means that we can guarantee that the shading is coherent between heightmap and cliff mesh.



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Screen Space Shading

UBM



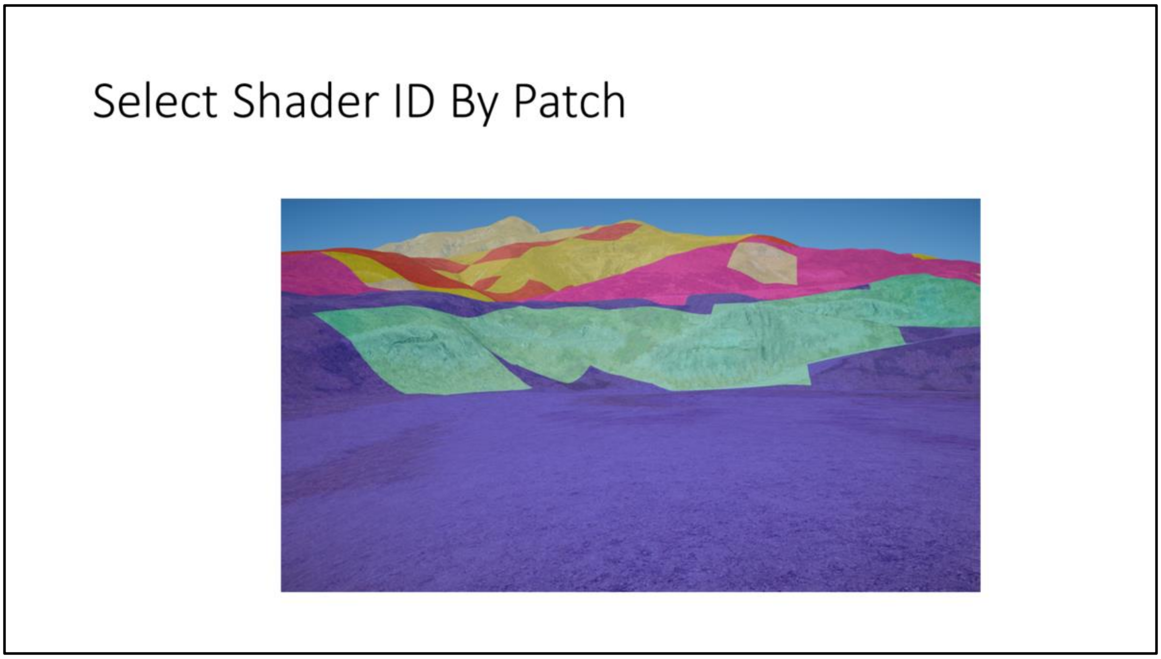
Terrain Shader Variations

So we have seen that we use 5 main shading flavors:

- Virtual texture shading
- Far atlas shading
- Near cliff
- Far cliff
- And also the full tri-planar splat map rendering (for anything that we don’t use an optimized path for)

Because we need to blend between these approaches this give us a total of 31 shader variations.

Some of these shader variations are more expensive then others.
To keep within our GPU budgets we need to make sure that we’re always using the cheapest shader variation (which we call the “Shader ID”) for any given part of the terrain.



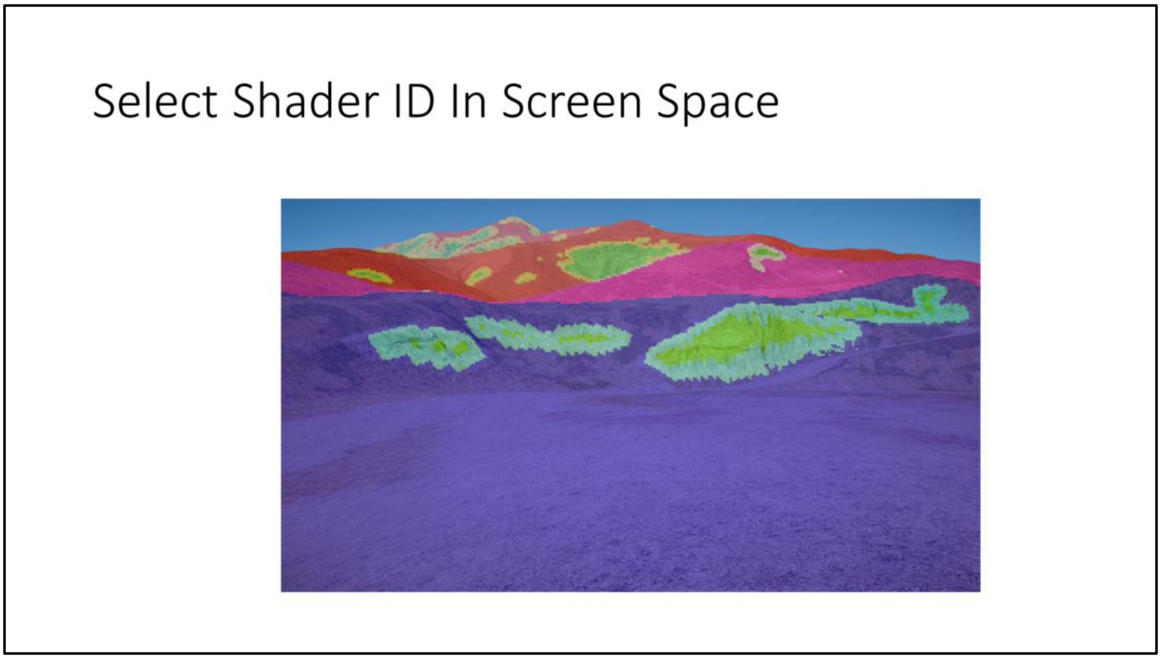
Select Shader ID By Patch

The simplest way to select an optimal Shader ID is by patch.
During the culling pass we determine what Shader ID to use and add each patch to its Shader ID bucket.

This is simple and practically free.
But it’s not always optimal because patches are big enough to have significant variation within the patch.

Also this approach only works with our heightmap geometry.
If we also want to use the terrain shaders for procedural cliff geometry then we would like to optimize shader selection for that geometry too.

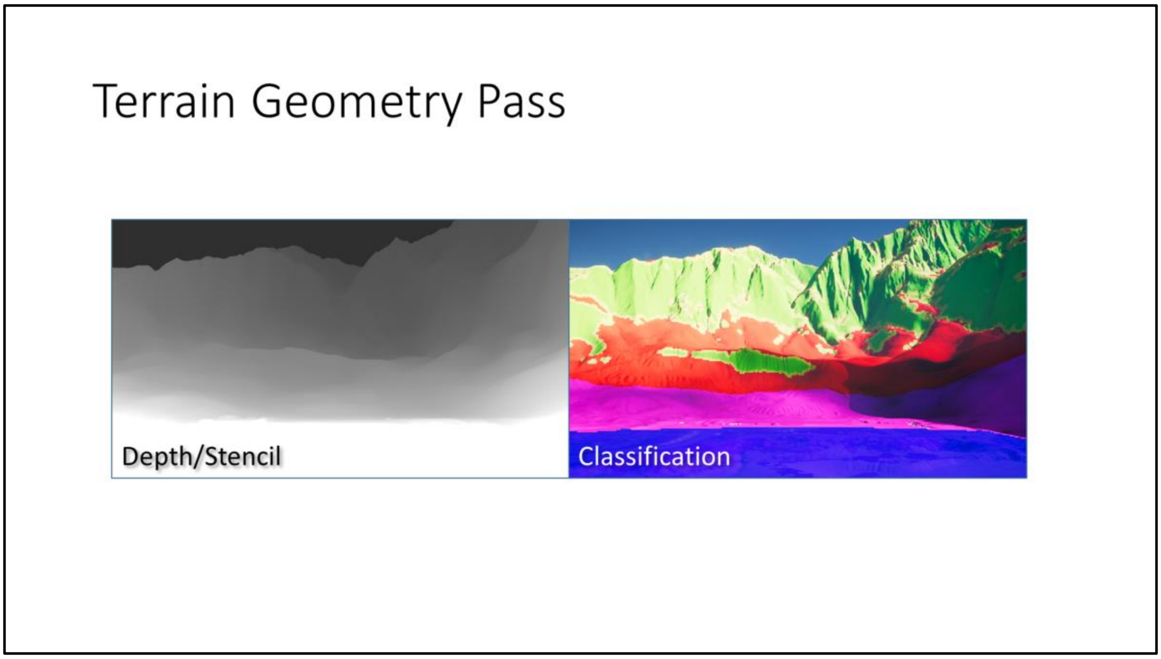
It’s also possible to use a compute shader to analyse each patch and generate a individual triangle lists for each Shader ID.
This is close to what Ghost Recon suggested in their GDC 2017 talk.



Select Shader ID In Screen Space

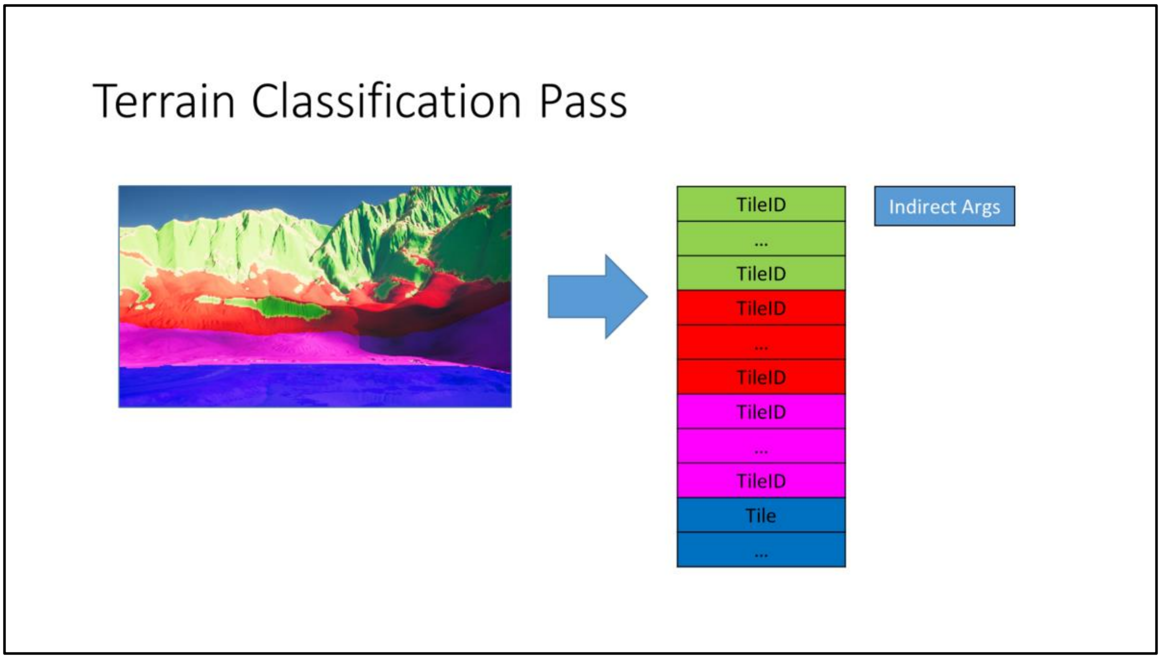
An alternative approach would be to select a Shader ID per screen space tile.
This would allow us to have much tighter bounds for the expensive shaders as we can see in this comparison.
This works for procedural cliff geometry too!

But it means that we need to do the actual shading in screen space!
That’s what we do, and I’ll explain how we do it…



Terrain Geometry Pass

First we do a terrain geometry pass that renders terrain depth and a terrain stencil bit along with a classification render target.
The 8bit classification render target contains the optimal Shader ID per pixel as a bitmask of our 5 rendering flavours.
Here I recolored the classification target to make it easier to see.



Terrain Classification Pass

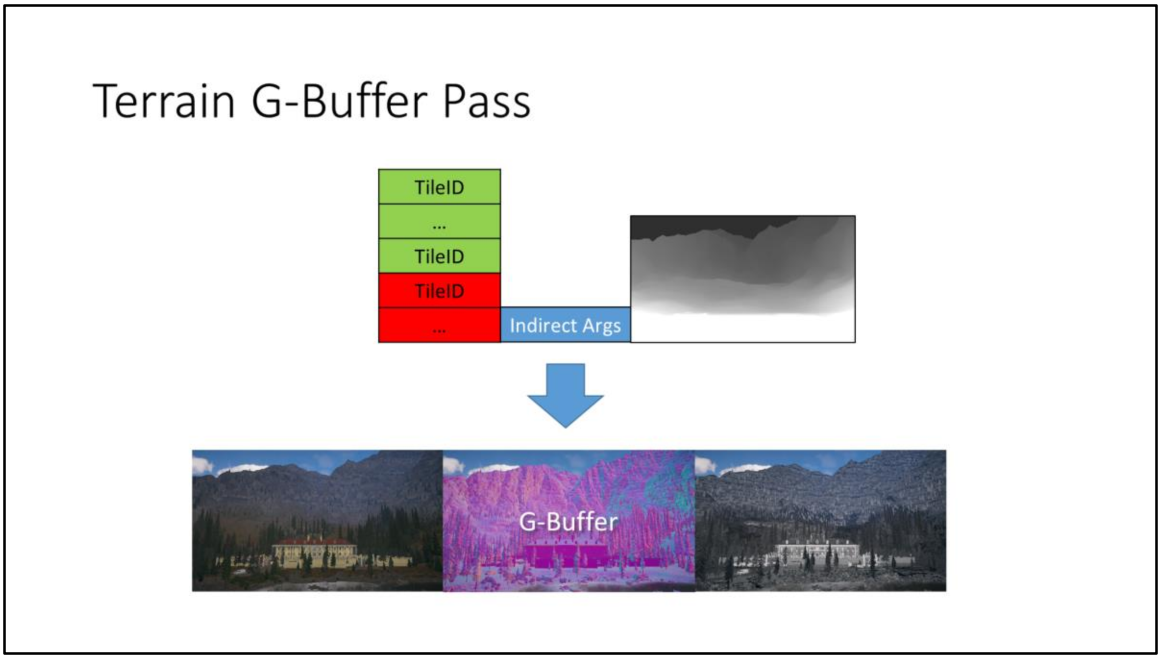
TileID
...
TileID
TileID
...
TileID
TileID
...
TileID
Tile
...

Indirect Args

Next we do a full screen compute shader classification pass that reads the classification buffer and ORs the Shader ID across a tile to get the combined Shader ID for the tile.
It outputs a final list of tiles for each possible shader variation along with an IndirectArgs to drive the next pass.

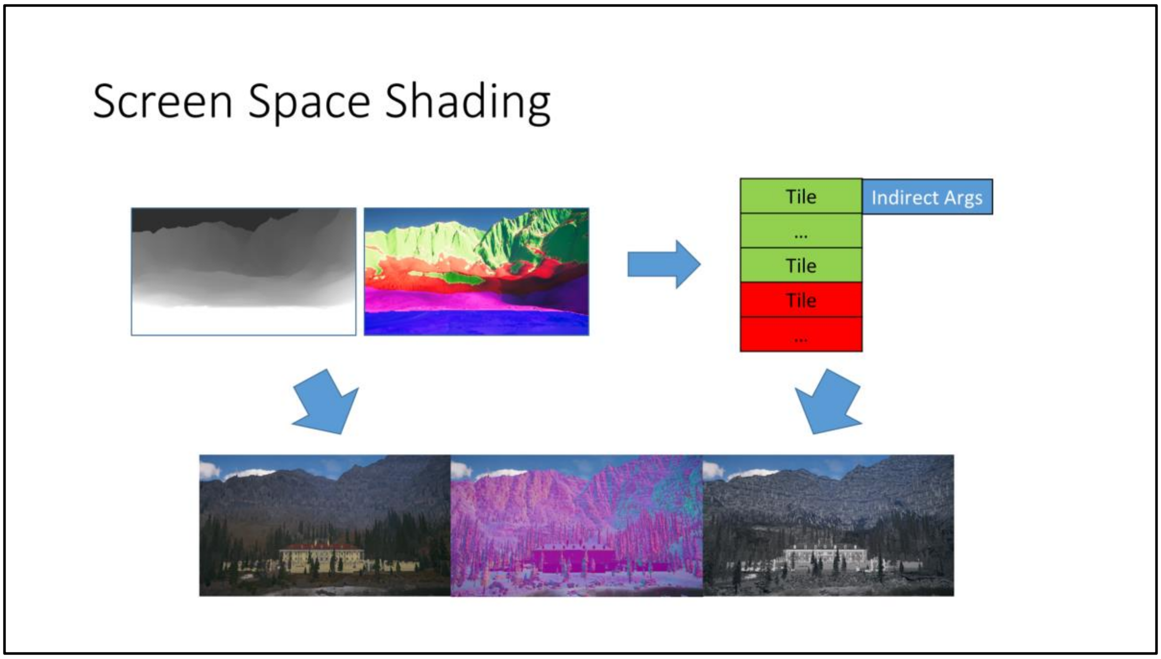
We found that a tile size of 8x8 was optimal.

We can read stencil to reject tiles that contain no terrain (so we don’t need to clear the classification target first).
On console we can use swizzles to OR the Shader ID, and ballot to reduce Interlocked operations when appending to the list.



Terrain G-Buffer Pass

Finally we do submit one tiled pixel shader draw call for each Shader ID.
The pass uses stencil to reject non-terrain pixels.
It reads the depth pass and renders the terrain to our actual G-Buffer.
I’ll cover how the pass does that in a moment.

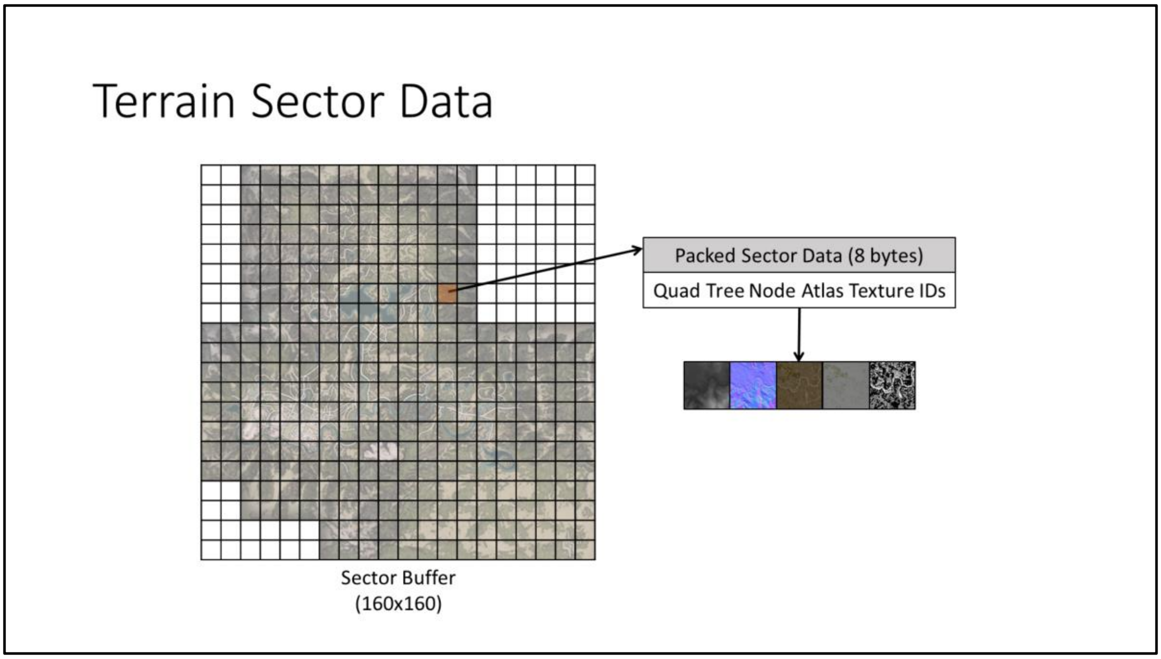


Screen Space Shading

This is the high level flow of data between the 3 passes

Sampling Terrain Textures

- How do we get terrain material data for each pixel in screen space?
  - First we get world position from the depth buffer
- How do we get terrain material data from world position?
  - The Terrain Quad Tree contains everything we need
  - It’s not optimal to traverse it per pixel though!
  - Build a map from world sector to terrain data



Terrain Sector Data

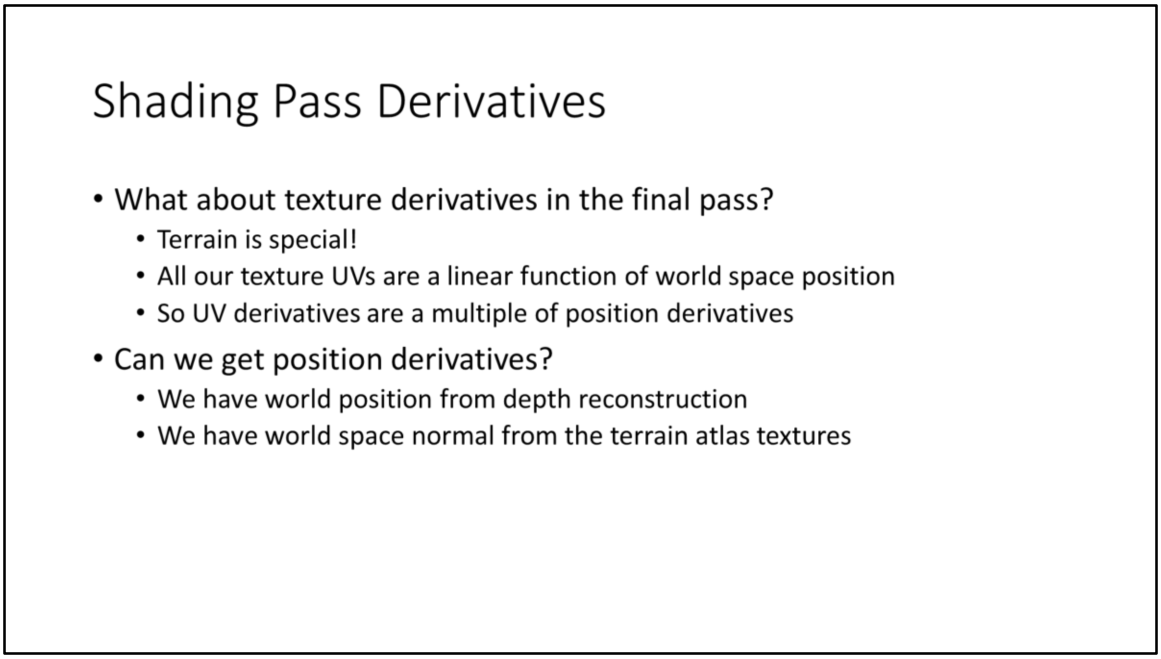
Packed Sector Data (8 bytes)

Quad Tree Node Atlas Texture IDs

Sector Buffer
(160x160)

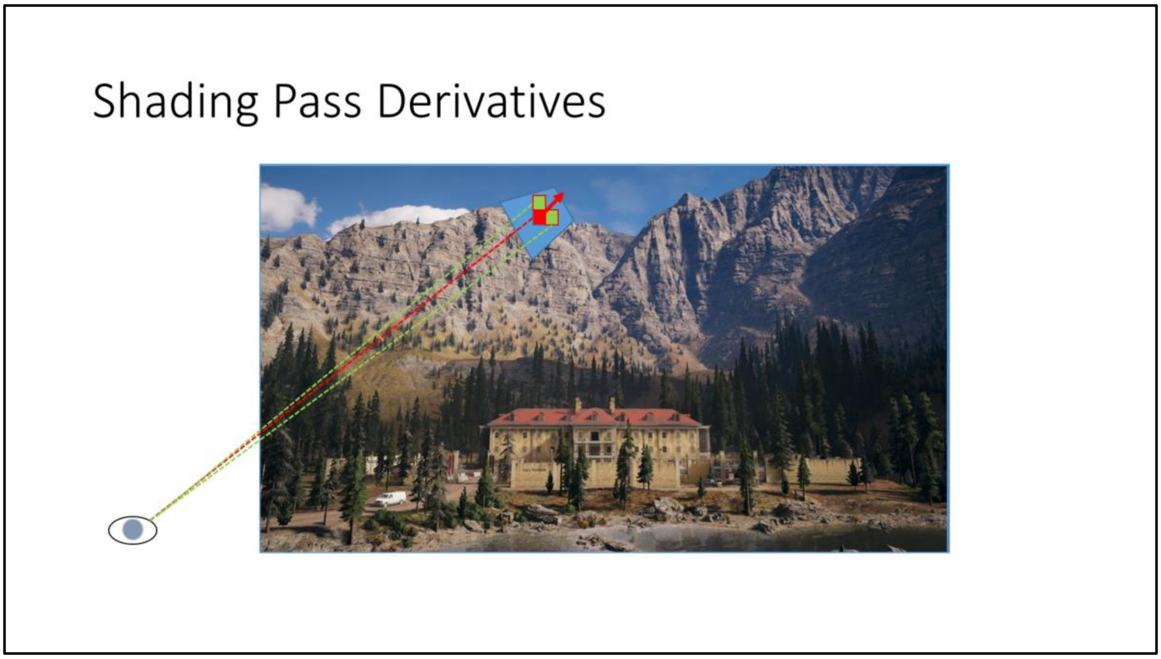
This is all of the information we need to be able to sample from the correct quadtree textures for any position in the world.
We build it every frame from the Terrain Quad Tree when we’re doing our other GPU pipeline work.

This is a linear buffer of uint2, not a texture (even though the diagram makes it look that way).
For each sector we store the quad tree node atlas texture ids for each loaded LOD (or zero if LOD isn’t loaded).
This is 10 bits per LOD packed into 64 bits.



Shading Pass Derivatives

- What about texture derivatives in the final pass?
  - Terrain is special!
  - All our texture UVs are a linear function of world space position
  - So UV derivatives are a multiple of position derivatives
- Can we get position derivatives?
  - We have world position from depth reconstruction
  - We have world space normal from the terrain atlas textures



Shading Pass Derivatives

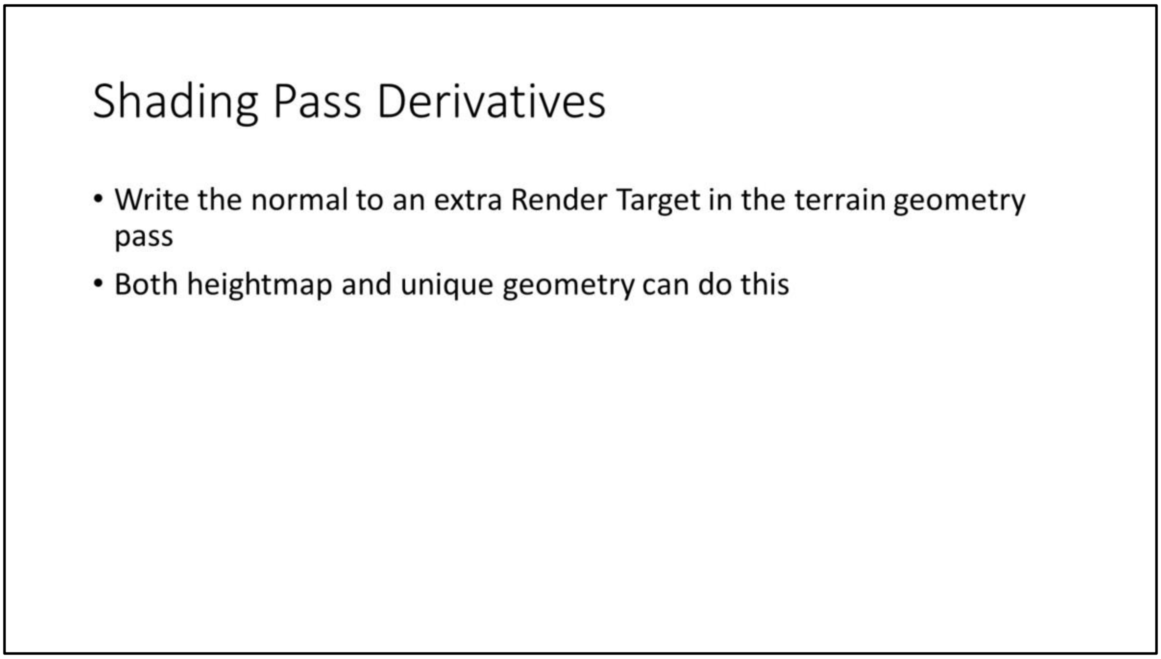
To get the world space derivatives for a point on the terrain at pixel (x,y) we do the following:

- Reconstruct the world position from depth.
- Get the terrain world space normal.
- Use this to create the plane that is normal to the terrain surface.
- Intersect this plane with a ray from the eye through the pixel at (x,y+1). This can give us the world space position derivative in the y screen space direction.
- Intersect this plane with a ray from the eye through the pixel at (x+1,y). This can give us the world space position derivative in the x screen space direction.

Now that we have ddx and ddy for world space position we can use it to create all of our texture space derivatives.
With the terrain sampling system and these derivatives we have everything we need to shade our terrain.

Shading Pass Derivatives

- Wait! This only works for terrain heightmap
  - Don't we want to unify with other unique geometry?

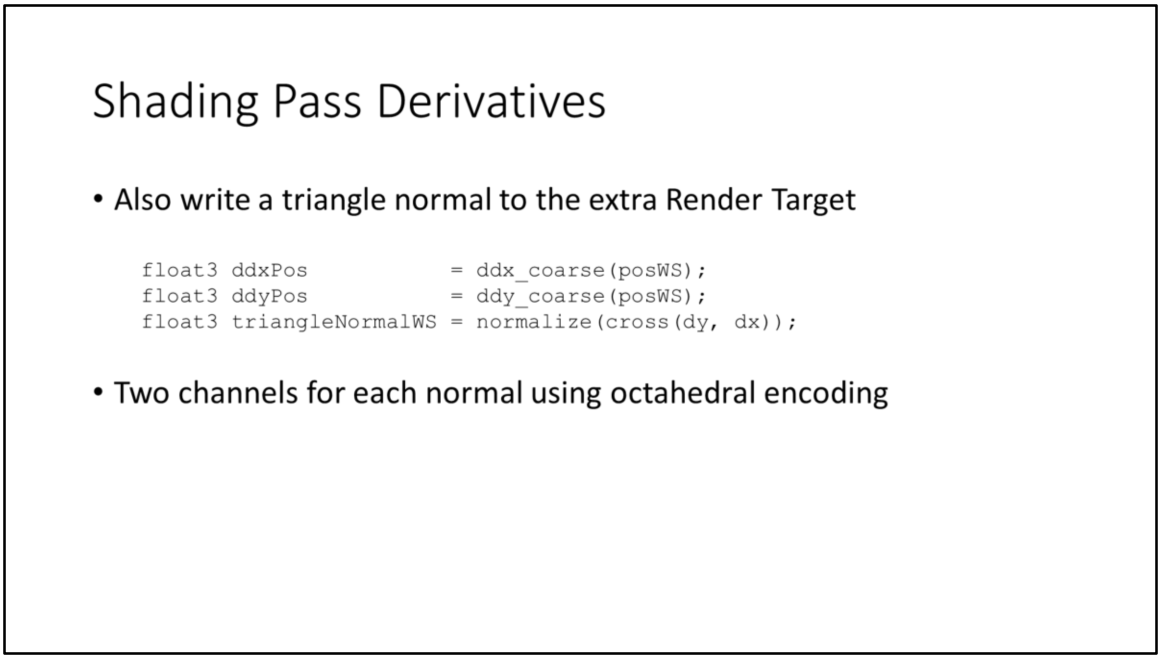


Shading Pass Derivatives

- Write the normal to an extra Render Target in the terrain geometry pass
- Both heightmap and unique geometry can do this

Shading Pass Derivatives

- Wait! This isn't the true derivative
  - The interpolated shading normal doesn't match the actual triangle geometry



Shading Pass Derivatives

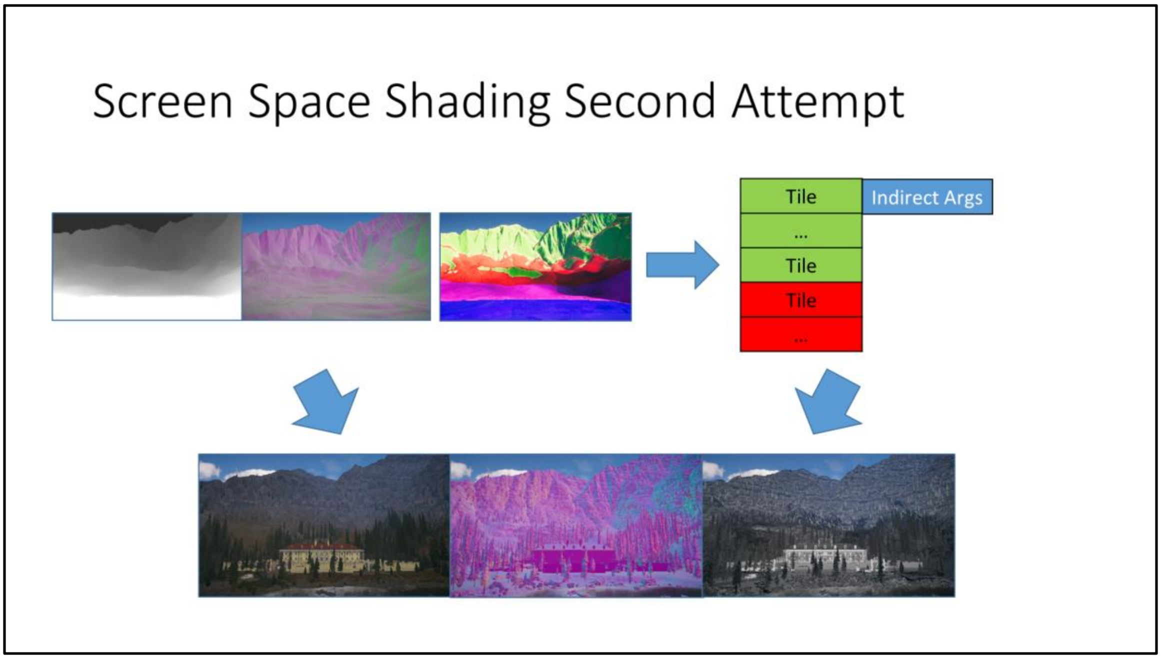
- Also write a triangle normal to the extra Render Target

```
float3 ddxPos            = ddx_coarse(posWS);
float3 ddyPos            = ddy_coarse(posWS);
float3 triangleNormalWS = normalize(cross(dy, dx));
```

- Two channels for each normal using octahedral encoding

For octahedral encoding see [Cigolle].

The storage of the triangle normal like this can be thought of as a compression format for ddx/ddy (6 floats->2floats).
Though of course it depends on other information being available in the depth buffer and projection matrices!



Screen Space Shading Second Attempt

Tile

…

Tile

Tile

…

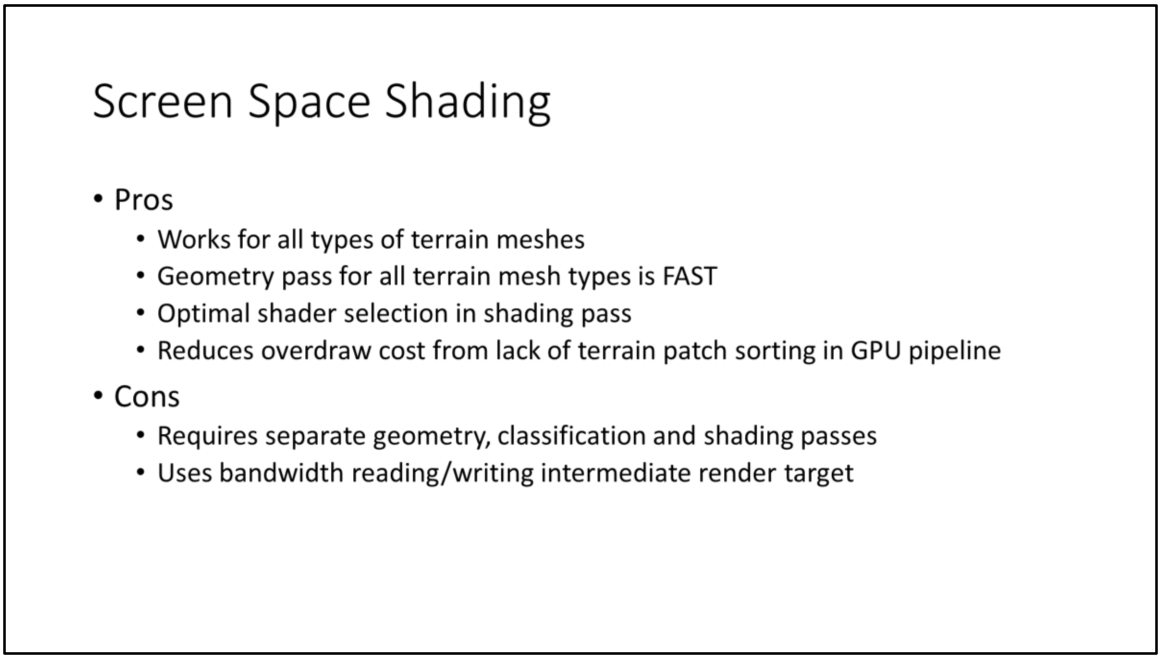
Indirect Args

We modify the first attempt but add a render target output from the terrain geometry pass

This RGBA8 target contains:

- 2 channels for world space geometry normal (octahedral encoding)
- 2 channels for world space triangle normal (octahedral encoding)

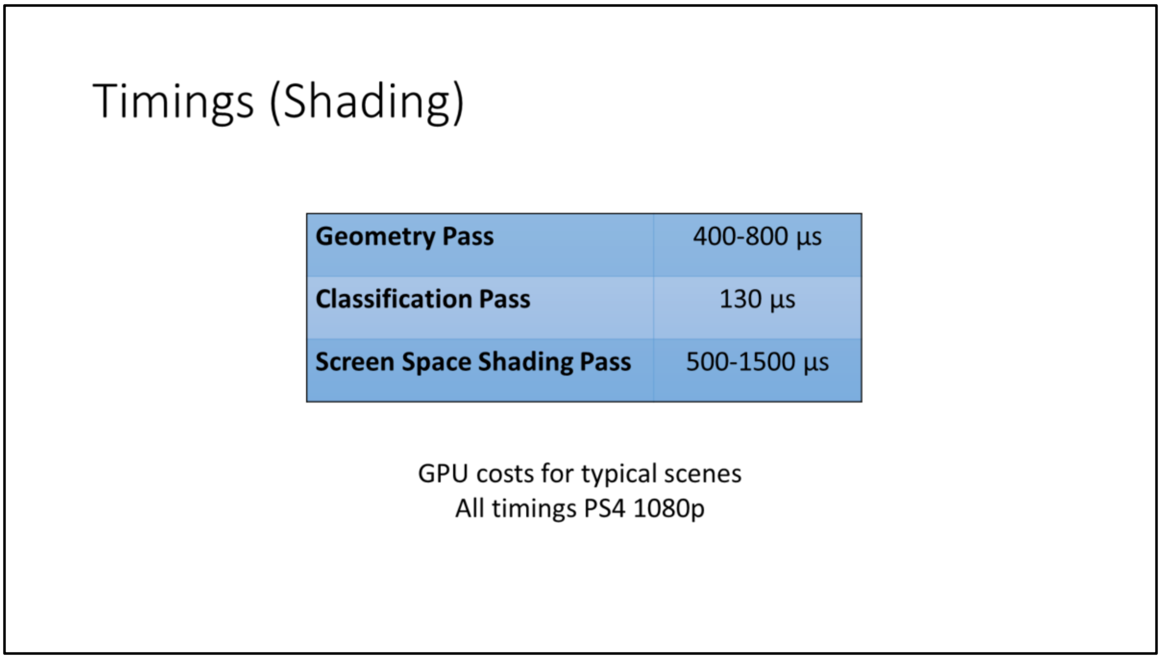
Then we proceed as before but we read the new target in the G-Buffer pass and derive the world space derivatives and the shading normal



Screen Space Shading

- Pros
  - Works for all types of terrain meshes
  - Geometry pass for all terrain mesh types is FAST
  - Optimal shader selection in shading pass
  - Reduces overdraw cost from lack of terrain patch sorting in GPU pipeline
- Cons
  - Requires separate geometry, classification and shading passes
  - Uses bandwidth reading/writing intermediate render target

Overall on Far Cry 5 we see better performance using screen space shading.
It’s good to be able to treat all types of terrain geometry uniformly, and get the same optimizations for everything.

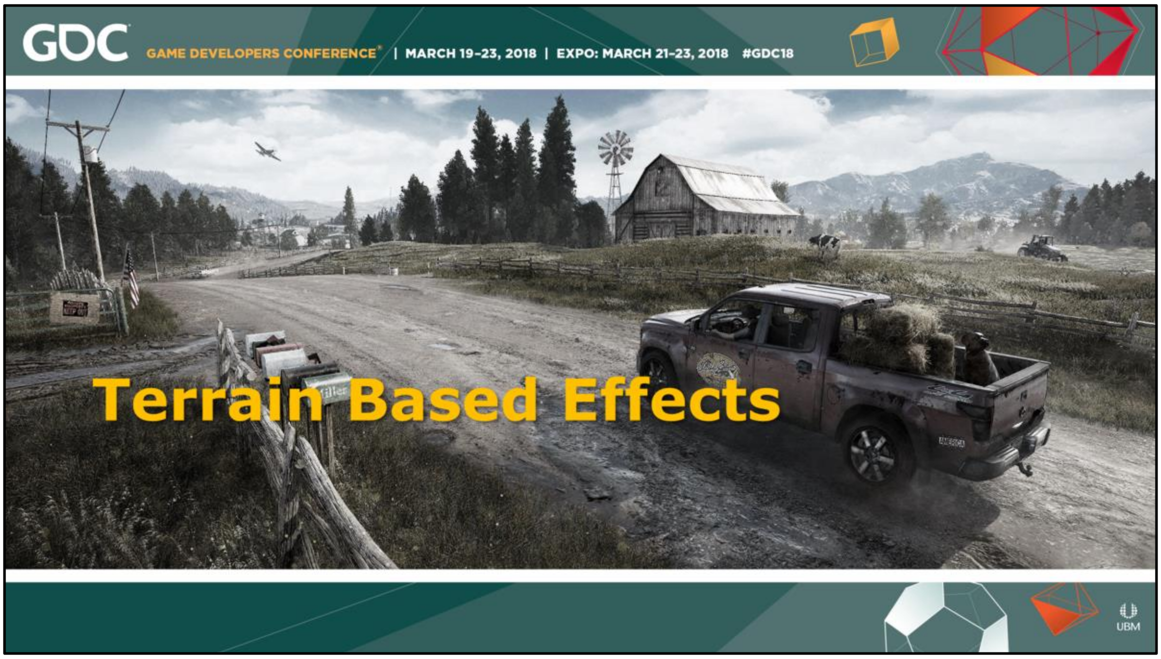


Timings (Shading)

Geometry Pass	400-800 µs
Classification Pass	130 µs
Screen Space Shading Pass	500-1500 µs

GPU costs for typical scenes
All timings PS4 1080p

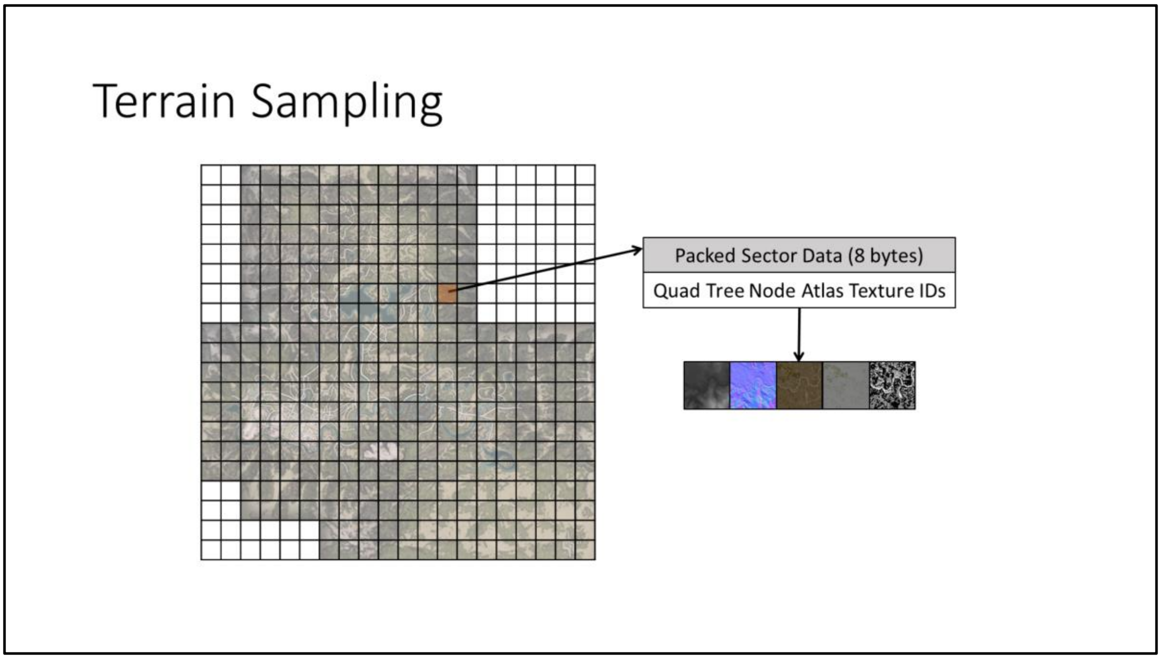
Our typical frame cost for G-Buffer terrain is around 1.5-2ms
The geometry pass is most expensive when we have a lot of additional mesh geometry that doesn’t flow through our heightmap culling pass.
The shading pass is most expensive when the screen is full with one of our more expensive shading techniques.



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Terrain Based Effects

UBM



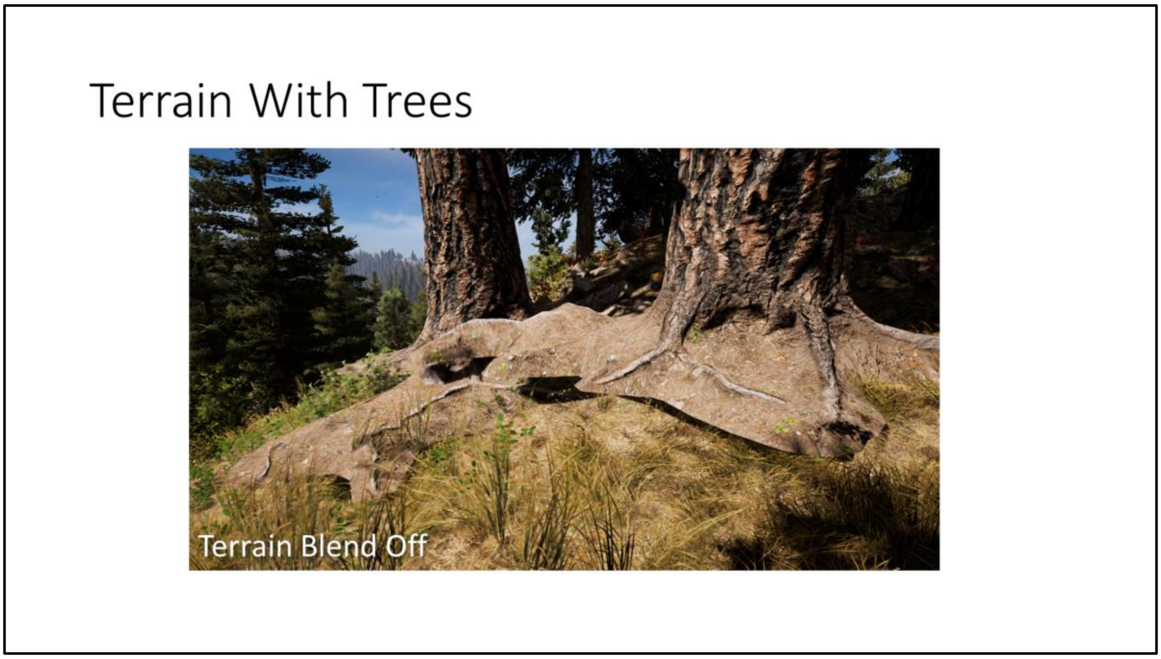
Terrain Sampling

Packed Sector Data (8 bytes)

Quad Tree Node Atlas Texture IDs

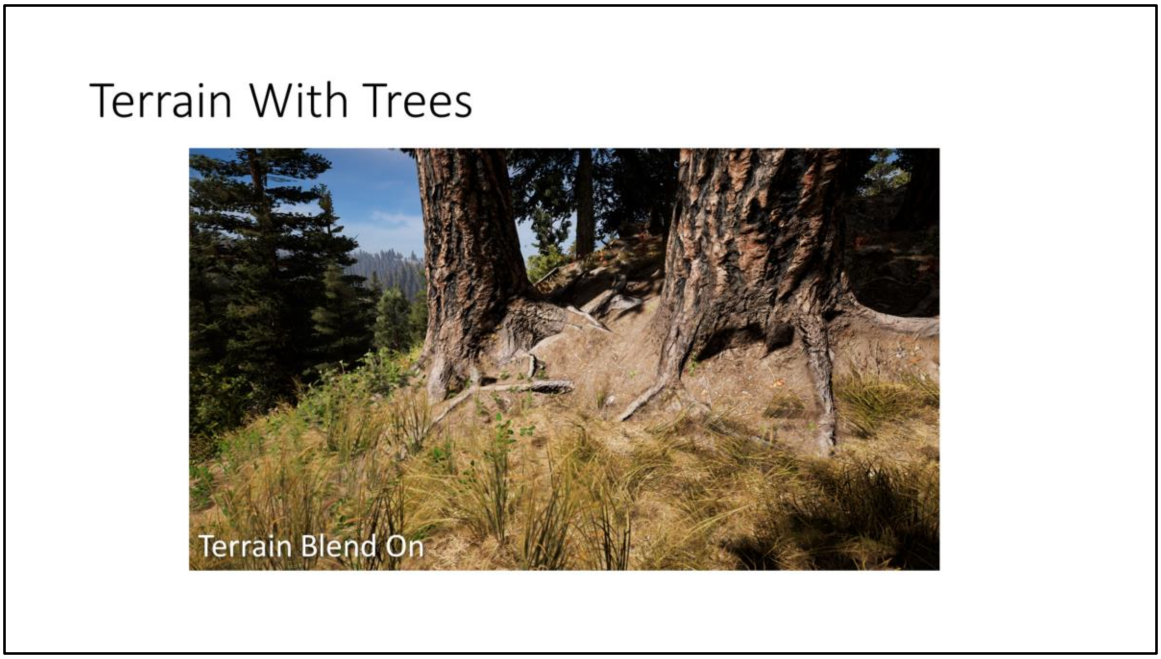
Remember our terrain sector data?
Using this for any point in the world we can access from any shader:

- What is the terrain height?
- What is the terrain albedo/normal/smoothness/etc.?
- What is the terrain material (the splat map material ID)?



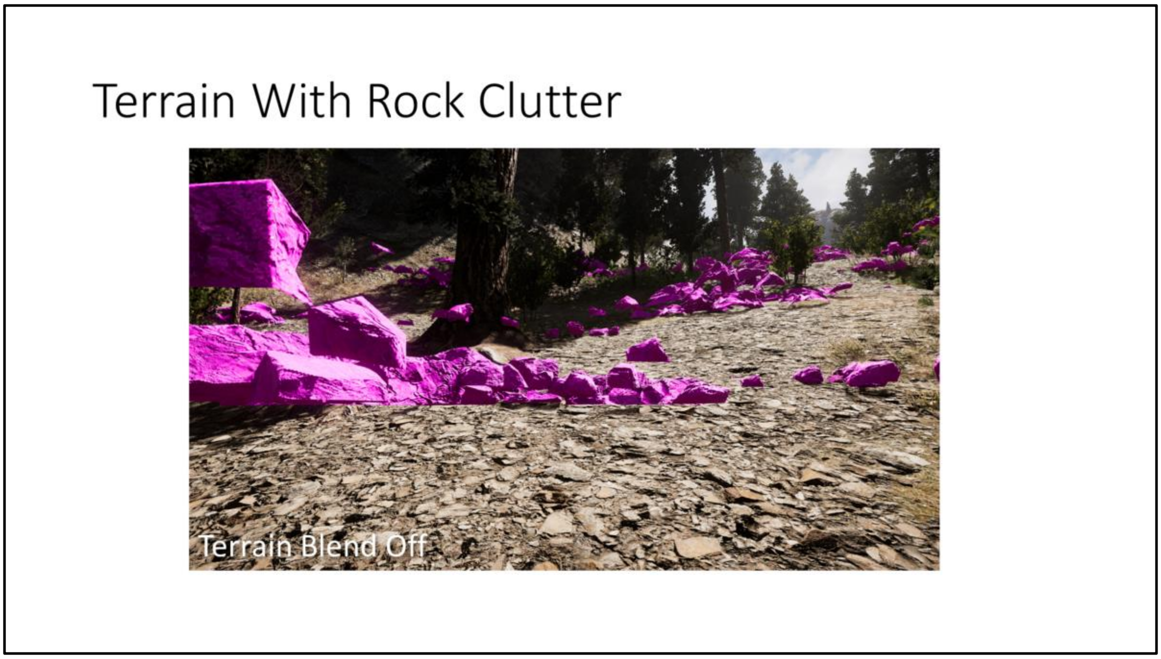
Terrain With Trees

One use case of this is to sample the terrain height map in the tree trunk vertex shader.
This allows us to blend tree roots with the terrain.
It gives natural variation to our forest areas but without the cost of having to generate multiple assets to fit terrain.
This is with the effect off.



Terrain With Trees

This is with the effect on.



Terrain With Rock Clutter

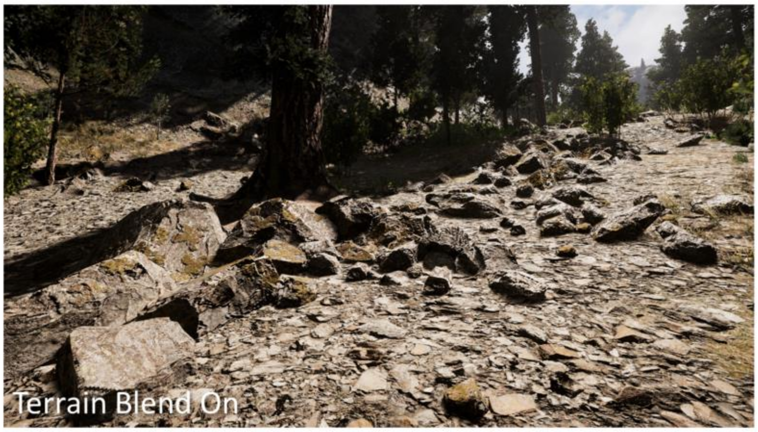
We can apply the same technique to other elements of our biomes.

For example here are some groups of rock clutter.
It’s more efficient to build these assets as large groups of rocks and debris.
But that can makes integrating with terrain difficult.
Normally we would simply orientate a rock cluster to try and fit to terrain, but if the terrain is bumpy then the rocks won’t fit well.

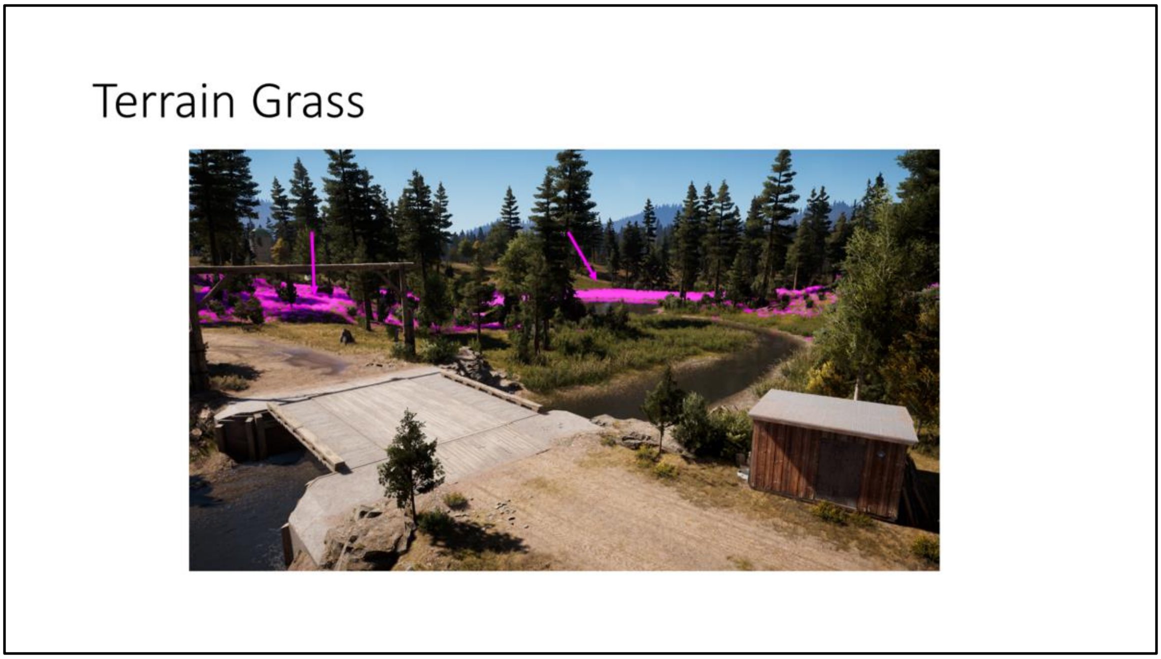
Using our terrain sampling we can blend the vertices to the terrain height.
We can also blend to the terrain color to help embed the rocks better into the scene.

This is with the effect off (highlighting the rocks in purple).

Terrain With Rock Clutter

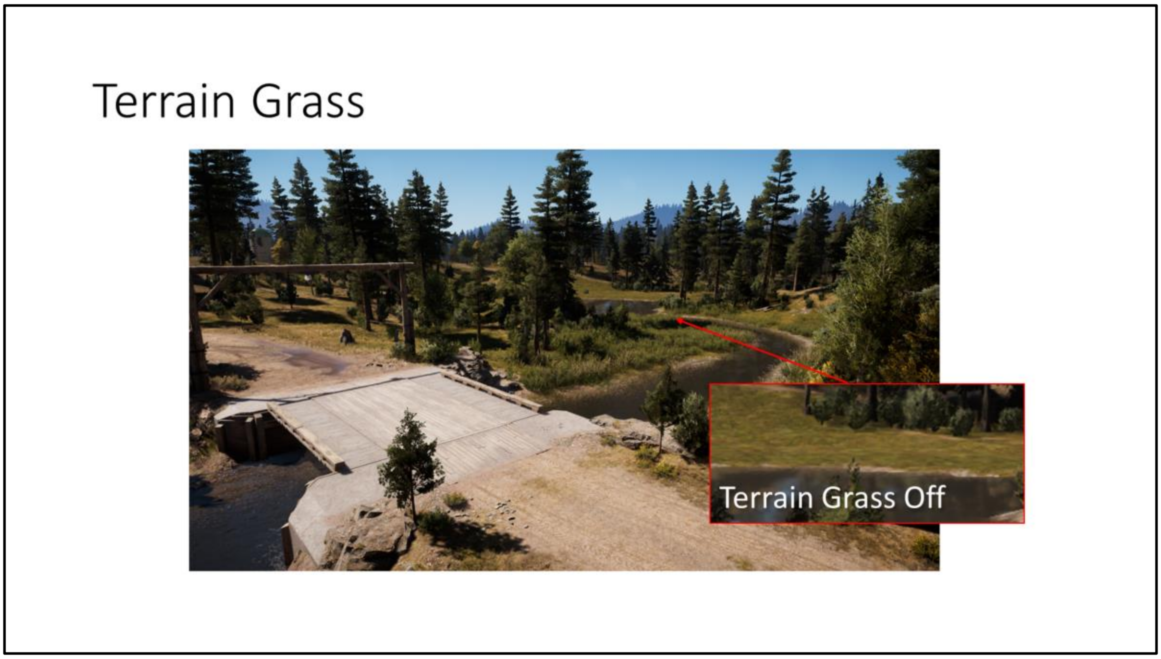


This is with the effect on.



Terrain Grass

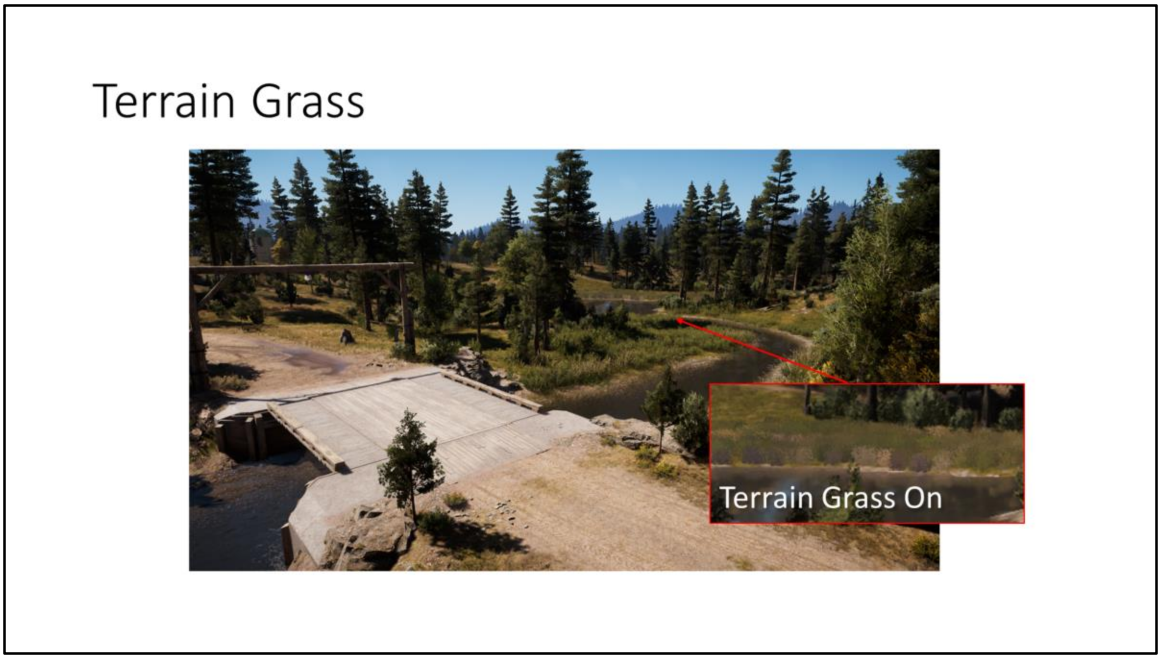
We also use the terrain data to help render the last LOD of our grass.
Near the camera we need to render high detail grass.
The last LOD highlighted here only needs to have much simpler grass quads that match the terrain height and color.



Terrain Grass

We place grass instances each frame with a compute shader that samples the terrain material type, height, color and normal.
It produces a buffer containing a list of grass instances that can be rendered very quickly in a single indirect draw call.

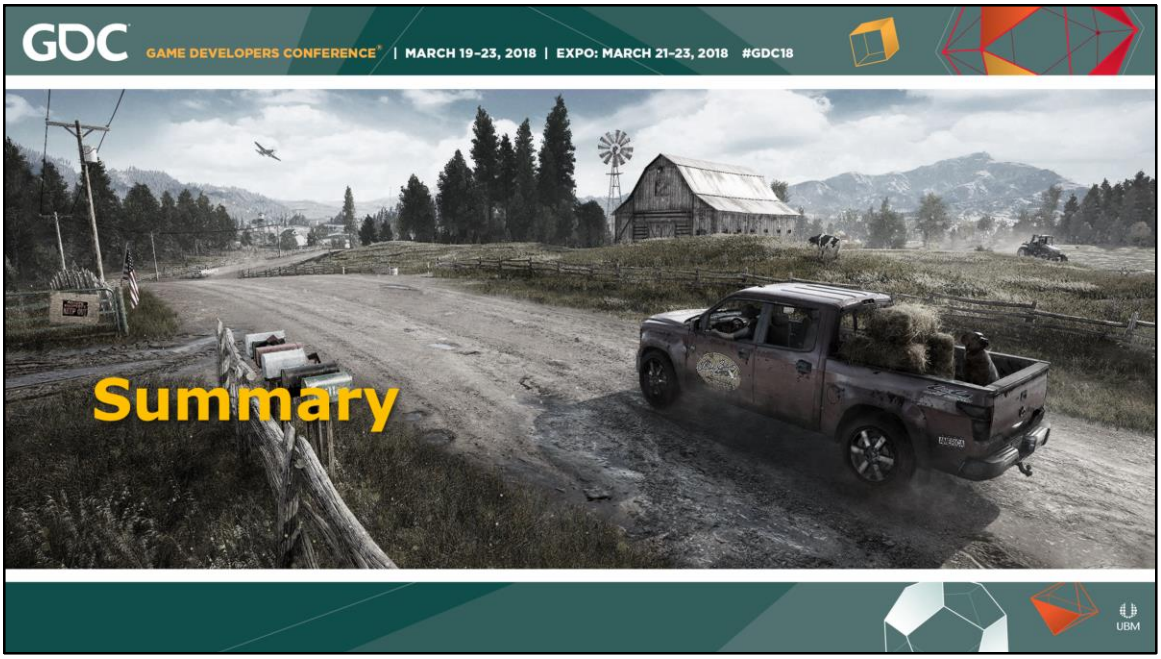
Here we can see the world without that last grass LOD.



Terrain Grass

Terrain Grass On

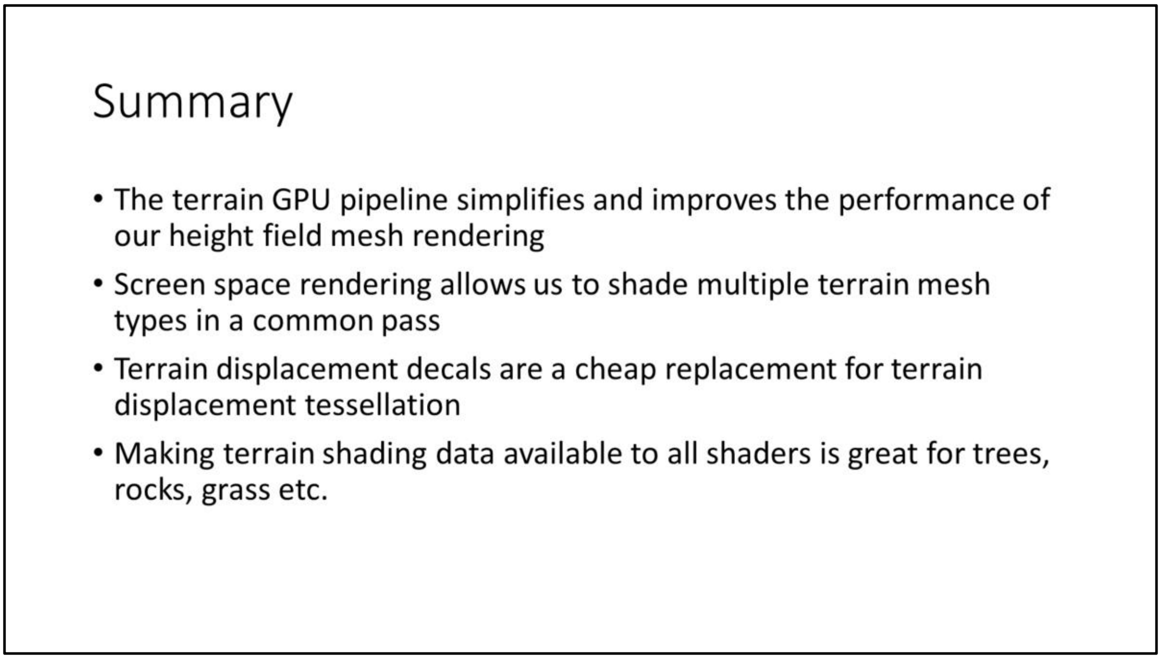
Here we can see the world with that last grass LOD.



GDC GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18

Summary

UBM



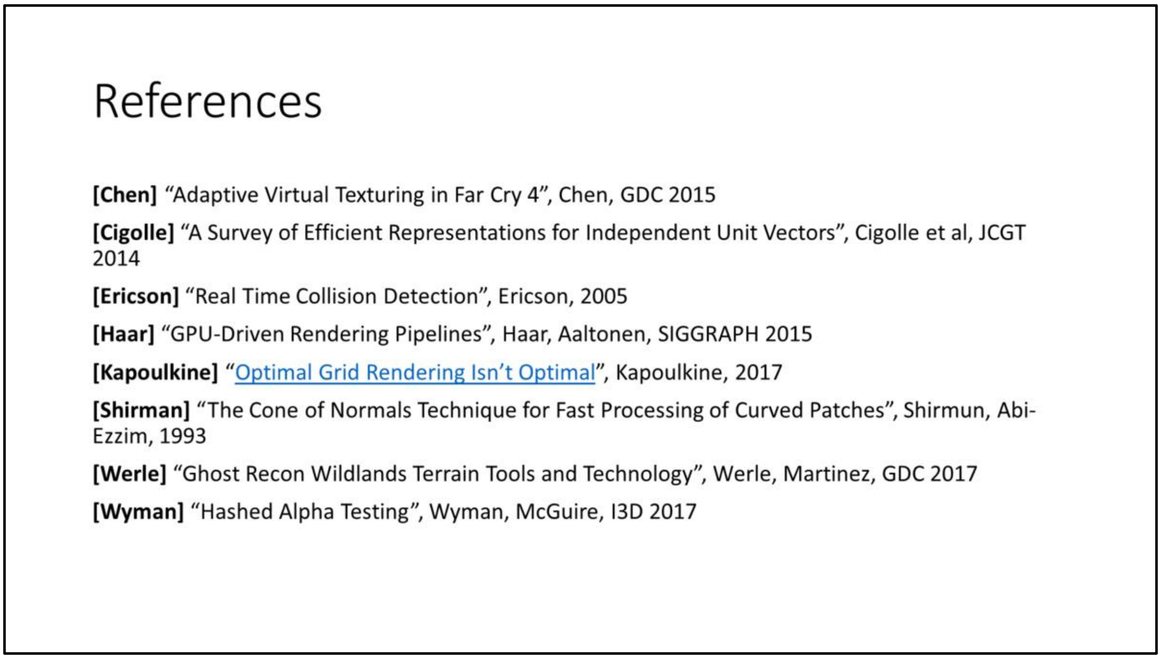
Summary

- The terrain GPU pipeline simplifies and improves the performance of our height field mesh rendering
- Screen space rendering allows us to shade multiple terrain mesh types in a common pass
- Terrain displacement decals are a cheap replacement for terrain displacement tessellation
- Making terrain shading data available to all shaders is great for trees, rocks, grass etc.

Appendix

Thanks

- Stephen McAuley
- Mickael Gilabert
- Jean-Sebastien Guay
- Nathalie Dubois
- Jean-Francois Tremblay
- Jendrik Illner
- Olivier Painnot
- Aurora Huang
- Alex Ribard
- Pavlo Turchyn
- Bane Grujic
- Mao Zhen Yu
- Guillaume Werle
- Etienne Carrier
- Doug Clayton
- Sebastien Viard
- Mark Cerny



References

[Chen] “Adaptive Virtual Texturing in Far Cry 4”, Chen, GDC 2015

[Cigolle] “A Survey of Efficient Representations for Independent Unit Vectors”, Cigolle et al, JCGT 2014

[Ericson] “Real Time Collision Detection”, Ericson, 2005

[Haar] “GPU-Driven Rendering Pipelines”, Haar, Aaltonen, SIGGRAPH 2015

[Kapoulkine] “Optimal Grid Rendering Isn’t Optimal”, Kapoulkine, 2017

[Shirman] “The Cone of Normals Technique for Fast Processing of Curved Patches”, Shirmun, Abi-Ezzim, 1993

[Werle] “Ghost Recon Wildlands Terrain Tools and Technology”, Werle, Martinez, GDC 2017

[Wyman] “Hashed Alpha Testing”, Wyman, McGuire, I3D 2017