

Re-inventing the wheel for snow rendering

Paolo Surricchio

Senior Staff Rendering Programmer
@rollingshark



Hello everyone, I hope you are having a great day here at GDC, and welcome to my talk, Re-inventing the wheel for snow rendering. Some housekeeping before the talk starts, remember to turn your phones to silent, and please fill out the survey at the end of the presentation. With that out of the way, let's get started! My name is Paolo Surricchio and I'm a Senior Staff Rendering Programmer at Santa Monica Studio.

Hello!

10+ years Engine/Rendering Engineer

Deadpool

Call of Duty: Advanced Warfare

Firewatch

God of War

Apex Legends

God of War: Ragnarök



I've been working as a rendering engineer in the industry for 10 years and going, and these are the games I worked on.
At Santa Monica Studio I work on all parts of the renderer, with a focus on the VFX system.

Goal of talk

How we write technology
to support high quality custom setups
by creating reusable toolset



The goal of this talk is to show how we write technology at SMS to support high-quality custom setups while creating a reusable toolset for the future, and how to apply this approach.

Goal of talk

Look at the re-write of
height field system in
God of War: Ragnarök



We are going to do that by looking at the process behind the re-write of our snow, or more generically, height field system for GoW Ragnarök. No spoilers for this or the previous game.

The height field system is the system responsible for displacing surfaces like snow in real time as characters or objects intersect through it as you can see in the video

Keep in mind

Less about tech details, more about reasoning behind each choice

Take-aways during the talk



The presentation is less about the intricate details of the system, and more about reasons behind why we made the choices we made.

During the talk I'm also going to highlight key take-aways, which I'll then summarize at the end of the presentation.

How we do things at SMS

Art and design have manual control over all details

We work in Maya as our main level editor, run on dev-kit

We use automation in service of creator's control



Some context about how we work at SMS. We have a world class team of artists and designers who have lots of manual control over all the details of the game.

We use Maya as our level editor, where levels are sculpted by hand by the level design and art team. We then run the game directly on the development kit, and that's the only way to see game rendering.

We use plenty of tools like Houdini to automate workflows, but we always give the option to tune by hand at the end of the pipeline.

Engineering challenge

Support creator team for the long term

Details are for our workflows, lessons are for all



This poses a challenge for engineering since we have to balance solutions that are too specific, or too generic, and we'll see how the height field system is an example of that.

While the details apply to our specific limitations, which might be different for your engine or game, the lessons behind the why of our choices are for everyone

Let's paint a scene

April 2018, God of War has shipped

Sequel is PS4/PS5, high quality on both



Let's start from the beginning of production on GoW:Ragnarok. April 2018, GoW has shipped, and we are working on the sequel.

From the beginning we knew we were going to be shipping on both PS4 and PS5, and it was imperative that the game was a great experience on both consoles.

When referring to performance or memory numbers in this presentation, I'm referring specifically about PS4 numbers, since the system must run well on the older console, and should scale up on the faster one.

God of War: Ragnarök requirements

Fimbulvetr (Old Norse) 'awful, mighty winter'

Good part of game is Midgard -> lots of snow



From the end of the last game, we know an event called Fimbulwinter is affecting Midgard by covering most of the world in blanket of snow. Since this is an area where we spend good part of the game, we know we are going to have to deal with an awful lot of snow

We already have that



Santa
Monica
Studio

At first, this might not be a concern. We shipped dynamic snow with displacement last project, GoW 2018, and it looked good, we should be good to go. Just “check the checkbox” on the new game.

Can't we re-use that? *Narrator voice: they couldn't*



Santa
Monica
Studio

You can see it here in motion. It looks great, so we are done! Aren't we?

Unfortunately, it wasn't that easy. To understand why we need to understand how this technology works.

GoW 2018 Snow: Rendering

Surface is rendered with ss-parallax mapping

All manual custom rendering different per material



Snow meshes are rendered with a technique called screen-space parallax mapping which is a variation on regular parallax mapping invented by Florian Strauss, our at-the-time technical director.

If you are familiar with parallax mapping the principle is the same, but instead of raymarching in tangent/texture space, you are raymarching on the screen. This allows various shortcuts and optimizations, but it's not without faults.

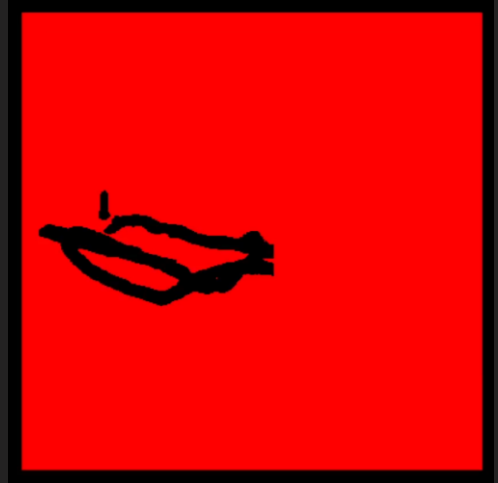
We use meshes and VFX to render displacement information, which is what carves displacement. Let's look at it next:

GoW 2018 Snow: Heights and Details

Top-down

Persistent scrolled

Carving shapes/VFX



We have a persistent top-down render target that scrolls with camera movement every frame in which we render meshes and vfx describing how deep we are carving into the snow.

We can render arbitrary meshes, often capsules and spheres for characters, with a custom shader that encodes information specifically tailored to how the parallax algorithm worked.

Like I said, our parallax mapping works in screen-space, so we need a mesh to project the top-down information on the screen. We render a mesh in the same position as the regular opaque mesh to do this.

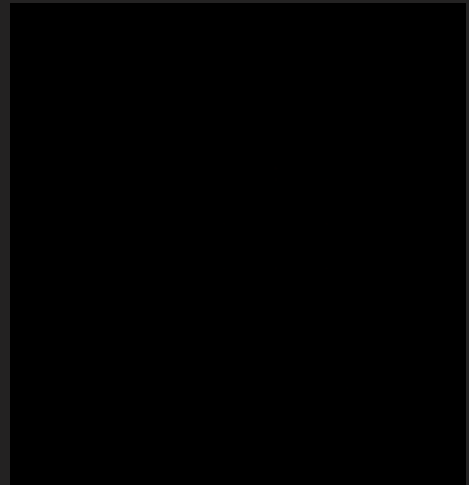
GoW 2018 Snow: Heights and Details

Rendered from camera view

Project top-down to screen

Additional VFX and meshes for details

What screen-space parallax uses

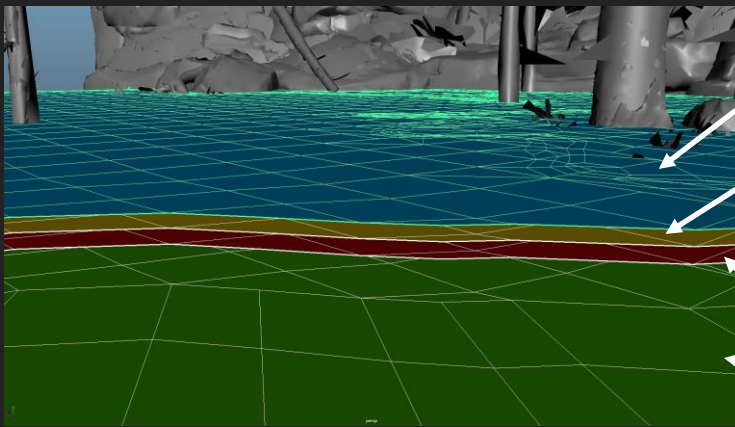


This is what it looks like: the mesh is now rendered from the camera point of view, and it samples the top-down projected texture I showed you before.

This brings all the information onto the screen. As you can see in the video, we then render extra details through meshes and particles from the camera point of view.

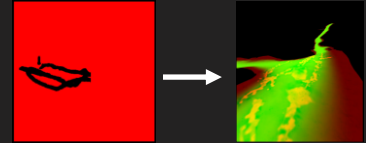
This texture is what the screen space parallax marches through. Let's see what the level setup looks like.

GoW 2018 Snow: Maya Setup



Flat rendering surface

Projection surface



Max and min displacement

Red (top) describes un-perturbed surface

Santa
Monica
Studio

This is a cross-cut of the main meshes needed to setup snow.

The artist creates the main rendering surface, in the image the light blue one. This is the mesh we render with the snow material and displace with ss parallax. Notice how the mesh must be flat to work well with parallax mapping.

Like I mentioned before, we then need a matching mesh to project top-down information onto the screen, in the image it's the yellow one, the projection surface.

After that, a maximum and minimum meshes have to be sculpted under it which are the actual meshes that describe the unperturbed snow surface.

This was often done with height textures, which is why in Maya you can't see any wavy surface resembling a terrain.

Sorry if that was a lot of packed details, it's not important to understand all of them since the point is why we decided to re-write this system. Let's see what the pros and cons of this technique are.

Pros...

Very detailed

Cost proportional to use



Parallax mapping allows us to render displacement with per-pixel accuracy.

Another pro is that parallax mapping cost is proportional to the surface detail and use. If your surface is all "elevated" or "non-displaced" you have an early out and the shader is not as expensive.

Like all techniques it does have cons though, and we'll see how they were a roadblock to scaling this technology.

...and cons...

Variable cost, worst when it looks the best

Tuning is hard, and error prone

Visual artifacts due to rendering technique



The cost being variable is a double-edged sword. Rendering is more expensive when the terrain is fully displaced because we have to ray-march through more pixels.

This is exactly what happens during combat scenarios where we are stressing our performance budgets the most.

Forgetting performance for a second, authoring was a concern. I actually simplified the maya setup, and it didn't include material and texture setup either. I skipped over many small details, but all these moving parts caused the system to be very error prone.

Lastly, parallax mapping has unavoidable artifacts that are just bound to how the technique works, and our version, while faster, made these artifacts worse.

...and cons... (parallax and camera)



Santa
Monica
Studio

GOD OF WAR

Parallax mapping shows artifacts as the view vector becomes parallel to the parallax plane. In screen space this is even worse because the screen space texture now crams all the height information in just a few pixels.

...and cons... (detail shimmering)



Santa
Monica
Studio

Our screen space version added other issues on top of it: since we render our details on the top surface, they shimmer with respect to the displaced final result, as you can see in the video above.

The only way to address this is to use projected decals, but to get this effect we are spawning hundreds, if not thousands of particles, and we can't afford anything close to that with decals.

...and cons

Limitations/workarounds about where and how it's used

Art spending too much time not making art

Not just them: design, collision, camera



This created a lot of design limitations about the spaces this was used in.

To work around the artifacts we had to take many precautions which meant lots of environment art work spent not on not making art. And not just environment art, this is designers, gameplay programmers for camera collision, cinematic animators, ... pretty much the whole team had to be aware of it.

This was the source of a lot of cross department dependencies which slowed down the use of the technique

Consequences: Limited Use



Santa
Monica
Studio

And this is why GoW 2018 only shipped a few areas in the whole game with this technology. All of them needed special care to make this work.

If all you have is a hammer...

Parallax is great

Great on water, still used it in GoW: Ragnarök

Not good for organic terrains

Good enough at the time, given the constraints



I want to make sure I'm not demonizing the tech here. SS-parallax mapping is an incredible technique, but we were pushing it too far, and we were forcing it to do more than it was suitable for.

In fact there are other places in GoW 2018 and Ragnarok where we still use this with a great result. Our water displacement rendering tech uses the same screen-space parallax technique for both games (although the rendering is improved a lot for Ragnarok)

Water is a great use case since it's already a special case in the game due to its nature: it's a flat plane, and both art and gameplay already have to work around it no matter what

It's just not good for large terrains that intersect organically with other meshes and delineate a curvy surface.

At the time it was the best solution, and if I went back in time, I'd probably do the same given the constraints we had.

Summary of Issues

Too much control, not right reasons

Too much because of parallax



Now it's hopefully clear we needed too much control to make this work on our terrain.

All of this was because of the special requirements of parallax mapping, and how everything revolved around feeding it the right data and working around its limitations.

A “Respectful Big Nope”

Art meeting with rendering:
current system won't work



This brings us back to the beginning of the production on GoW: Ragnarok. We had a meeting with art seniors and leads where they told us there was no way we could use or scale the current system to the needs of this new game.

This is a professional team of senior artists, we are talking about a room where the total experience of game development can be counted in tens of years each, with the total beyond the hundred. They wouldn't say this if it wasn't the case.

Why?

Bigger game

Bigger team

More teams



And the main reason for it came down to the size of the development effort. From pre-production it was clear that everything about the new title was bigger:
the world is more than twice the size than the previous game, the team was going to be larger than the one on the previous game, and we had more partners and outsourcing studios helping than before.

Game Requirements

Snow is everywhere

10x from GoW 2018 to Ragnarök 😊 (and more cases)



And as I said before, snow is everywhere.

We actually ended up having snow displacement in an order of magnitude more levels than before.

And, spoiler for future slides, we used the new system on more than snow.

What Didn't Work

Change workflow: more automatic

Change core: geometrical approach



It's clear that the main parts causing scaling issues were how much manual control was needed to make it work, and the rendering limitations of parallax.

We need to improve the workflow, and we know a geometrical approach won't suffer from any of the issues we had before.

Once we addressed those, this was what we needed for the new system:

Requirements

No terrain system; arbitrary meshes

Highly customizable with material features

Minimal setup

Aim for good average perf, no fixed memory cost

Must run well/look good on PS4, scale up on PS5



We don't have a de-facto terrain system in our engine; terrain is sculpted with bespoke meshes created by art, so the tech must work on arbitrary meshes.

On top of that, we want to make sure our system integrates automatically with all our rendering features of our material editor, and it must be easy to setup

With respect to performance, we want less variance between best- and worst-case scenario.

We don't want to pay any cost for memory if we are not using this technique as there are large portions of the game that don't use it.

Lastly, it must run well and look good on the older console, and scale in quality with little to no extra work on the newer one.

There are compromises though,

Compromise

Not same pixel quality

Art team gave the thumbs up

We did it! And we learned a ton, and created new tools



And it's the per pixel quality parallax provided. We knew it was hard to reach that with geometry, especially on PS4.
This was not a concern for the art team.

Allow me to jump to the end and tell you we shipped a system that hit all these requirements, but more importantly, and that's the interpretation key for this talk, we achieved this goal while leaving ourselves open during development, and after, to do much more without compromising reaching the goal.

Let's see how we went about doing that.

The First Step

Establish visual goal: get a spec from art



The first step is to get a spec from art on what the visual goal is, and what features we'd like to have.

Kyle Bromley, one of our Senior Environment Artist got assigned as the environment art point person to work on the snow tech with me.

I mentioned to him it would be cool to get an idea of what the final result could look like. Kyle told me "give me a day or two and I'll write down a few notes".

Oh, you want a spec?



This is what he came back with. I want to take a second to highlight the level of mastery of our art team.

It actually looked prettier, I had to cut it and remove parts to make it fit in this presentation.

We now have a visual bar with guidelines and ideas for what it should achieve. It is also possible to see three distinct sub-systems emerging from the document:

From left to right, first, we have displacement based on where the player has walked, then we want to alter the look of the inner part of the displacement, and lastly we would like to have details around the displaced areas.

Engineering Time

Back of the napkin math for perf: Tri-count and overshading

Same area as previous tech (50x50m)

Tessellated mesh by hand



So now it's on engineering to do our part: the main question we had was how many triangles we could push before tri-count or tiny-triangle overdraw would be an issue.

I took a mesh in maya, sub-divided it a few times until I could represent detail reasonably well.

I ran some more tests with those goals in mind, and in the end we knew we could confidently rely on geometry displacement for what we needed.

There are more details about this in a few slides, but what's more important is our first exciting take-away.

Takeaway #1

Start from product, do back of napkin math

Make your knowledge "better"



Always start from the product and then work your way backwards.
Do an estimate of costs to make sure your solution is in the ballpark of doable to the best of your knowledge
If your knowledge is bad, try everything to make it better.

How, exactly?

How do we go about dynamic geometry displacement?

Standing on shoulders of giants



Well, after our rendering/hardware test, we actually have to make it real and work dynamically in game.

Before doing anything new, we took a look at how other games had solved this problem. Geometry displacement is by far not a new topic.

What's out there

Terrain/fixed meshes/fixed tessellation

T-Junctions

High memory cost (or not flexible enough)



Unfortunately some of our requirements are really strict: most of the techniques we analyzed didn't work with our requirements. Some have artifacts like T-junctions they were able to ignore, or start from pre-processed meshes with a limited set of features. This is a topic worth of its own presentation, but after more research we realized what was out there didn't work for us.

I would recommend looking into the research though because their limitations might not be a problem for your game, or your engine. I left the references at the end of the presentation.

Do It Yourself

Some RnD but production/team constraints

Idea: indirect draws of differently tessellated tris skinned to the original mesh – only where needed

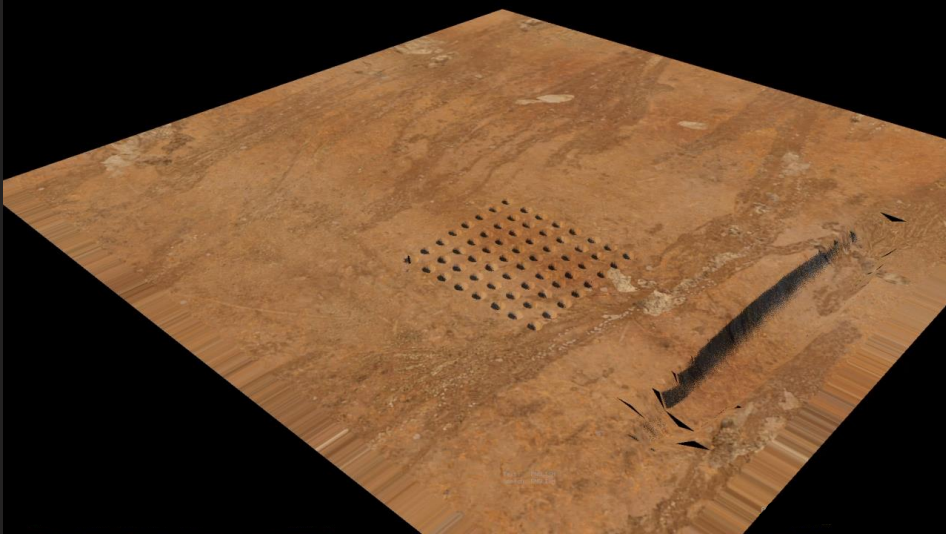
Didn't pan out. Was it wasted? *(narrator voice: it wasn't)*



We did some research ourselves: given the number of rendering engineers and the production schedule at the time we had to heavily timebox this effort.

We tried a few approaches, and I'll just mention the goals: try to only tessellate where needed, but ideally try to reduce tessellation factor as much as possible, where possible.

Do It Yourself: Programmer Art Edition



Santa
Monica
Studio

I don't need to mention that no artists were involved with the making of this video.

This is a proof of concept of how, artifacts aside, we can support a high level of tessellation if we only do it where needed.

You can see in the wireframe that we only tessellate if there are details to tessellate.

Sometimes you get lucky

Another programmer was working on hardware tessellation pipeline
Has issues, but fits all our boxes



Around the same time another rendering programmer, Valerio Guagliumi, was working on adding hardware tessellation to our general shader pipeline.

This was to spear head an effort to get better mesh details for PS5 only.

Hardware tessellation fits all our boxes: no extra memory, can be tuned and can scale dynamically. We knew about its performance woes, but we were confident enough in one way or another we could address that. The real story is a bit more complicated than that.

But lets look at our second, and most important, take-away:

Takeaway #2

Force yourself into a modular approach (*moral of the story*)

No "Core Pillars"



This is a summary of the development philosophy I'm talking about in this presentation: build things in blocks without any of them being a core pillar without which everything collapses
We designed things so that nothing would rely specifically on how the triangles were going to be tessellated, just that they were.

The issue with the previous technology was how much had to be designed around the specific requirements of the parallax algorithm.

How It Works – 1st Pass

Any mesh in Maya: just a checkbox (limitation on topology)

Made setup one checkbox

Reuse what worked before



Let's look at how the new system evolved through its iterations. I'm going to show you an example of what the output of the system was at every step.

Unlike before, artists just sculpt meshes as they normally do. The only limitations we had was that mesh topology had to respect some density rules. This got addressed later.

For a mesh to be displaced, it's a checkbox in our material, everything else is done for you during the build pipeline.

At runtime, we kept the same carving logic as before, although much more simplified since all the requirements of parallax were gone.

How It Works – 1st Pass

Automated heights and carving



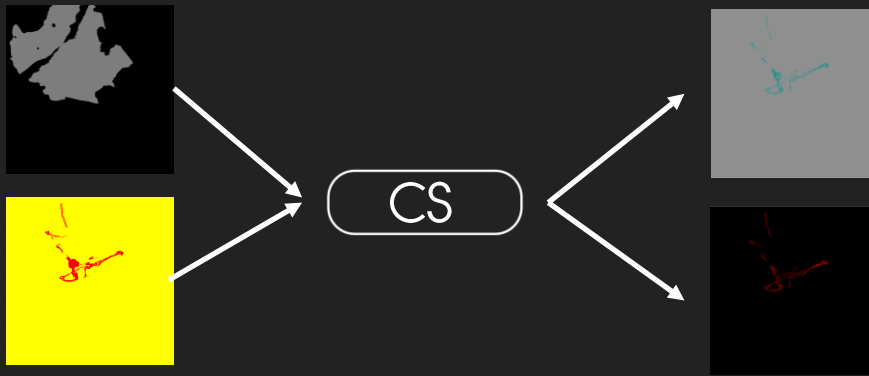
Santa
Monica
Studio

We render an area of the same size as before around the camera from a top-down view of the original un-displaced mesh. We then render carving shapes into a render target. We test them against the un-displaced mesh depth so that shapes that are not intersecting with the snow don't carve anything.

While the depth texture is re-drawn every frame, the carving map is persistent, and like before we scroll it every frame depending on the camera movement, keeping the center of the projection where the camera is.

How It Works – 1st Pass

Compositing pass: smoothing heights & calculate tessellation factor



Santa
Monica
Studio

In a compute shader we then composite all the information into a few textures for displacement, and we also smooth out the heights to avoid sharp displacement.

Given for every pixel we are analyzing the neighbors, we calculate the detail variance in an area around every pixel and derive a tessellation factor, which we write in a texture.

How It Works – 1st Pass

Hardware tessellation for mesh



Santa
Monica
Studio

When rendering the mesh, we use hardware tessellation to subdivide the triangle as much as the tessellation texture suggests, and then displace the mesh. This is a look at this first iteration in a test scene.

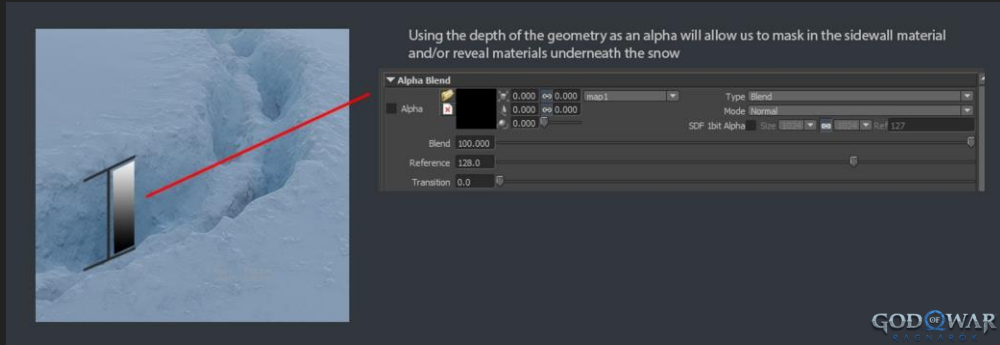
Again, programmer art.

What was cool about this is the fact we could already render a surface that would have been impossible with the previous technique.

Now with this you just get a displaced surface, which doesn't look like much. Let's look at how we improved the look.

How It Works – 2nd Pass

Layers for blending



Santa
Monica
Studio

GOD OF WAR
SANTAMONICA

We built a suite of tools to translate the system-specific height information into the language of our material editor. Heights are transformed into alpha, which means we can blend different layers depending on the displacement. This means that we immediately have the whole blending feature set of our material system available to us, without having to do anything, and that's the bread and butter of our artists' workflow.

How It Works – 2nd Pass



Santa
Monica
Studio

This is what this iteration looked like. Again, very much programmer art here. You can see how when snow is displaced, we reveal a mud layer. It can be any layer the artist desires, or any number of layers.

How It Works – 2nd Pass

Art starts requesting masking feature: red flag?

You *can* use it, but you don't *have* to



After this, art started asking for features like vertex color masking to control the displacement. At first one might think we were going down the same path as the previous system but there's a major, fundamental, difference:

this was not necessary to ship the tech. It was just to have more control.

I find that if a tool gives more verbs to the user, it is additive. If a tool is the only way to control a feature, it's subtractive.

How It Works – 3rd Pass

Detail Mask

Persistent

Used as mask for
detail normal map



Next step was to get some of the high-resolution details we lost when moving away from parallax:
we generate another top-down persistent scrolling low-res mask around where the player steps that is used to blend a detail normal map on the snow mesh to add detail around the displaced areas.

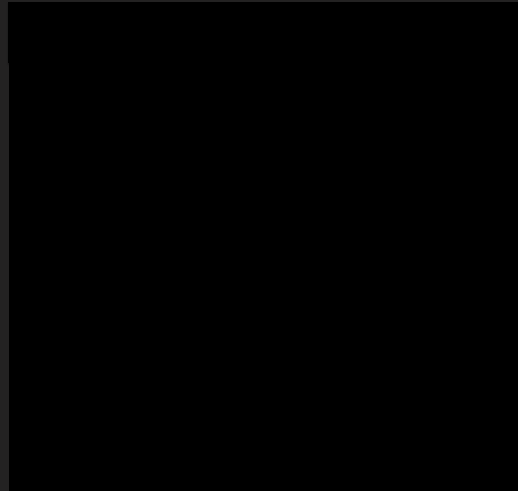
How It Works – 3rd Pass

Dynamic Details:

Non-Persistent

Generate dynamic
normals in shader

Could have done more



On top of that, we also generate a top-down height mask from which we derive every frame a dynamic normal map to simulate finer snow details on the surface.

We used this on all our materials by default, and while this is currently working and shipped in the final game, we didn't push this second part as much as we could because the VFX team at the time didn't have room in the schedule.

This was not a requirement though, so the team was happy to get what we could get.

How It Works – 3rd Pass

Addressed mesh topology limitation



Santa
Monica
Studio

Lastly we addressed the limitation of the mesh topology. You can now throw any reasonable mesh at the feature, and it behaves correctly. And like we had designed, we are only tessellating around displacement, and only as little as we need to.

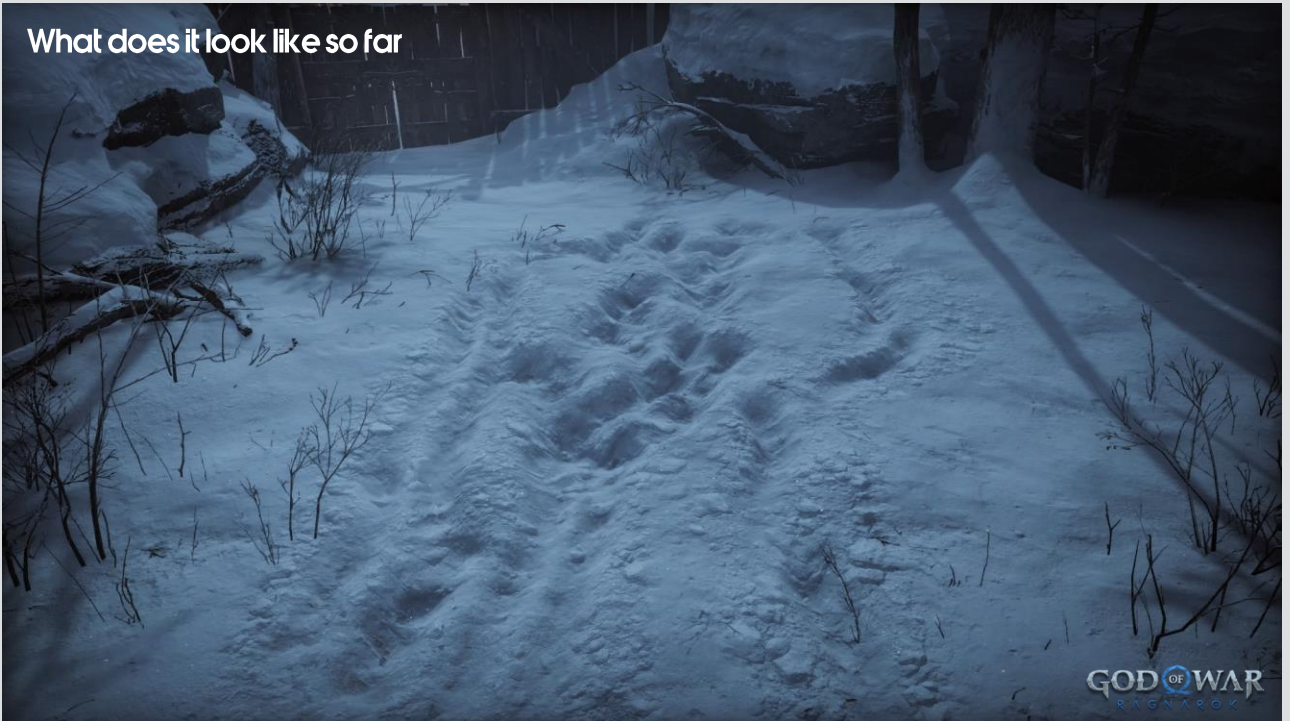
Also in this video you can see a good example of the detail normal map blended around the displacement.

Adaptive tessellation detail



This is a zoomed in screenshot to show how we tessellate only around displacement. I turned off all the other features to make it clearer to see.

What does it look like so far



This is what it looks like up until this point.

How does it run?

As we start using it everywhere, cost creeps up

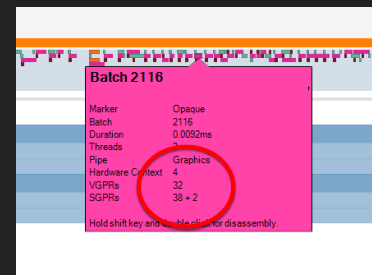


While we knew tessellation was slow, it still surprised us by how slow it was in some areas.
Let me show you what we had:

Performance: PS4 Capture



Occupancy 😞



This is a capture on base PS4 of a large terrain mesh using hardware tessellation. You can see all the waves I have highlighted: those are all hull and domain shader invocations.

As you can see at the bottom our occupancy goes to almost 0. That's because we are waiting on the tessellator to do its thing and return. I highlighted how simple the hull shader is. It doesn't matter. We are bound by how fast the tessellator comes back to us.

We don't have too much time to go into the nitty-gritty details, but the summary is that hardware tessellation has a high entry fixed cost. If you take a mesh and turn on tessellation with a passthrough hull shader, without any subdivision, it's going to be about 30% to 40% slower than a regular draw call, all other things equals. Again, this is a rough estimate just to give a high-level idea.

This cost is higher on older generation hardware than newer one, so keep that in mind. The rule of thumb for tessellation is if you use it, really use it and subdivide those triangles. Big fewer triangles that are tessellated more are better than more smaller triangles tessellated less. There's more tradeoffs to consider, so I left a reference to a great presentation on the topic which goes more in details in the bonus section of this presentation.

Also keep in mind that I'm referring to AMD hardware specifically.

Let's continue the development story with our last takeaway:

Takeaway #3

Be prepared



There is a huge difference between a hiccup like this if you know it's coming or not. As much as the total cost did surprise us a bit, we were aware of this since we started using tessellation, so we had been thinking of a few solutions to address it since day one.

How did we address it?

Problem is tessellated tri count

Our use case is worst case scenario

Artist are nice people, tried to offer help



We know we need to run hardware tessellation on as few triangles as possible.

Unfortunately, terrain is the worst-case scenario for this: you have large, oddly shaped meshes, which are hard to cull, and it's easy to have quite a few active ones at the same time even though only a limited area really needs tessellation.

Art offered to manually split the meshes to help with this in their LOD pass, but we knew we could automate this for them.

Solution Overview - Summary

Inspired by GPU driven rendering

Partition the mesh and only render what's needed

Saved a lot, thanks to research done before

Solved a specific problem, gained knowledge



I'll give you a quick summary of the solution first to finish the story, and then we'll go into details.

Inspired by GPU driven rendering techniques, we thought about automatically splitting the mesh in smaller pieces and render as few as possible with tessellation.

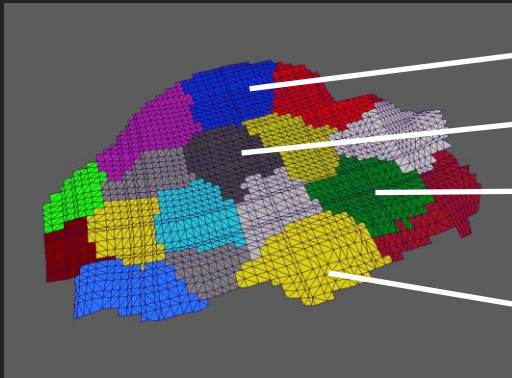
This made a huge difference and was the major part in shipping this whole system on PS4.

All of this was possible because of the RnD we had done at the beginning of the project.

This is another example of technology we built for a specific case, but the lessons will keep paying off for the future.

How: Data

Divide mesh in chunks/meshlets
Create offset and culling info buffers



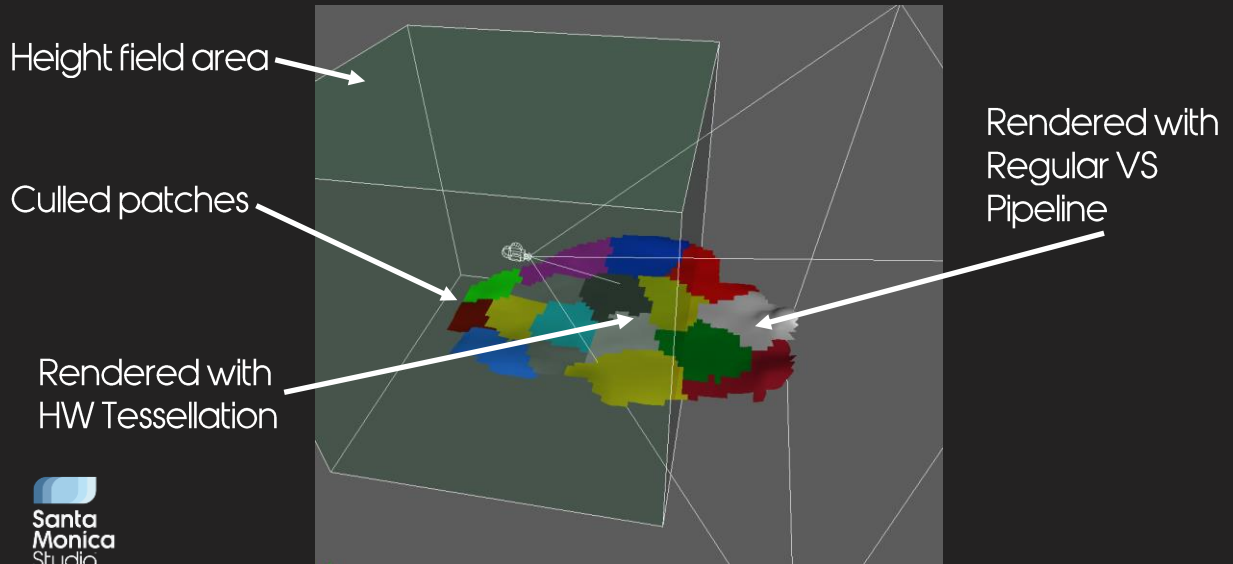
Santa
Monica
Studio

maxMeshletIndexCount = 315

```
[  
  {  
    indexOffset = 0;  
    indexCount = 315;  
  },  
  {  
    indexOffset = 800;  
    indexCount = 300;  
  },  
  {  
    indexOffset = 1400;  
    indexCount = 303;  
  },  
  ...  
  {  
    indexOffset = 15000;  
    indexCount = 297;  
  }  
]
```

We divide the index buffer into sections of geometrically cohesive triangles. For this specific technique try having patches of the same triangle count, or as close as possible to each other. We save the max index count of our meshlets for each mesh. While doing that, generate a buffer of the index offset and index count of each meshlet, and whatever culling info you need. For our culling scenario spheres were good enough.

How: Culling and Draw Args



For each mesh we have a compute shader dispatch where we go through each individual meshlet, and decide whether it is not in the frustum and should be culled. If it's not culled, the second step is to decide whether it's far enough from the displacement area that it should be a non tessellated meshlet.

We enqueue their indices in 2 buffers, one for tessellated meshlets, one for regular ones.

How: Draw Call



Lastly, instead of rendering the mesh with one draw call, we render the mesh with 2 indirect draws, one with hardware tessellation, one with regular vertex pipeline.

The compute shader that does the culling fills the `vertexCountPerInstance` of the two indirect draw arguments assuming the worst-case index count for all meshlets, times the number of patches that passed the test for each type.

If we rendered the mesh with a regular shader, the indices we'd get wouldn't match the mesh ones.

How: Mesh VS

```
// Returns true if the given Input Assembler Index ID in a meshlet draw corresponds to a valid
// meshlet index, and in that case replaces it with the meshlet index ID that can be used
// to read from the index buffer.
bool FetchMeshletIndexID(..., const bool isTessellatedDraw, inout uint indexID)
{
    uint maxMeshletIndexCount = constantBuffer.heightField@MaxMeshletIndexCount;

    const uint indexIDInMeshlet = indexID % maxMeshletIndexCount;
    const uint meshletIDInDraw = indexID / maxMeshletIndexCount;

    // Get meshlet ID, and base index offset
    uint meshletID, baseIndexOffset;
    if (isTessellatedDraw)
    {
        meshletID = meshletIDInDraw;
        baseIndexOffset = constantBuffer.tessellatedIndexOffset;
    }
    else
    {
        meshletID = constantBuffer.heightField@MeshletCount - 1 - meshletIDInDraw;
        baseIndexOffset = 0;
    }

    const uint meshletIndex = constantBuffer.heightField@MeshletIndices[meshletID];
    const uint2 fetchedMeshletInfo = constantBuffer.heightField@MeshletInfo[meshletIndex];

    HeightField@MeshletInfo meshletInfo;
    meshletInfo.indexOffset = fetchedMeshletInfo.x;
    meshletInfo.indexCount = fetchedMeshletInfo.y;

    // We use the maximum meshlet index count as index stride between meshlets
    // while the vertex buffers are compacted so the surplus indices must be discarded
    if (indexIDInMeshlet >= meshletInfo.indexCount)
        return false;

    indexID = indexIDInMeshlet + meshletInfo.indexOffset + baseIndexOffset;
    return true;
}
```

Draw index count is
(0, meshletIdxCountMax * validChunksCount)

Not "real" index

Clip invalid vertices

```
// We use the maximum meshlet index count as index stride between meshlets
// while the vertex buffers are compacted so the surplus indices must be discarded
if (indexIDInMeshlet >= meshletInfo.indexCount)
    return false;

indexID = indexIDInMeshlet + meshletInfo.indexOffset + baseIndexOffset;

return true;
```

Must have manual vertex fetching!



The vertex shader is where it all ties together. This is simplified pseudocode.

In the vertex shader we use the index we get from the hardware to grab our meshlet information which allows us to know how many indices we actually have in the meshlet this vertex belongs to.

We clip the extra vertices that are not part of our meshlet by collapsing them to 0. This is how we achieved rendering the meshlets with different index count, and why it's important to try to divide the mesh in meshlets with as close to the same index count as as possible.

Side note here: for this technique we rely on not using automatic vertex fetching. On consoles we don't do that since it's the same on that GCN hardware.

Result



We end up with exactly what we wanted: as few triangles as possible are getting tessellated. In the video you can see red patches appearing as we move forward. Red is tessellated patches, while green is non-tessellated.

If I lock the frustum with debug commands and look behind me, we see how effective the culling is.

It's great to see how much of the mesh is being culled.

Art is happy

Does anyone need ~1ms?

No need of user optimization -> better game



Art was very happy with this. We came back with an optimization worth from 0.5 ms to 2 ms in some scenarios. This was great for the team. It saved lot of optimization work, which can now be dedicated to making the game better.

Be ready for more

New lead, more people: more wiggle room in schedule

Core is shippable. Didn't plan for more. It happened

Go back to visual goal: add more details

Can do that thanks to composite pass



But wait, there's more!

During the production of Ragnarök Stephen McAuley joined Santa Monica Studio as lead of the rendering team, and with more people joining we had more breathing room.

This gave us room to experiment and improve the system. While we were happy with the result, we knew we could push it further.

Before I continue it's important to point out: we could have stopped here, we didn't plan on having more time. But also, when we did have time, we had the tool to expand the system, and use that extra precious time in the best way possible.

Looking at the original pitch made from Kyle we decided to add more fine details to the system.

Thanks to our composite pass already running we could generate more textures and information really cheaply.

We ended up adding two more pieces to the system.

More Polish

Opaque particles/geo collision



Santa
Monica
Studio

Our particle system runs the simulation particles fully on the GPU. Among other features we support depth collision.

Unfortunately, depth collision is not supported for our opaque particles, and for our mini-models, which are opaque models rendered for each particle instead of sprites.

This is because both opaque particles, and mini-models write to depth, which means they collide with themselves and each other if you turn depth collision on.

Now though we have a representation of the surface from the top down, and we have all the information we need to do accurate collision. We added height field collision to our particles systems, and in the game you can see pieces of snow being thrown around as you walk, colliding realistically with the terrain.

This is also incredibly cheap, which is great when you are adding a feature toward the end of production.

We can now create hundreds of small models and have them collide without any issue. This is even better than depth collision since we are not bound to the screen, and instead we can do it anywhere we have snow in a 25 meter radius from the camera.

More polish

Added dynamic model spawning around displacement



Santa
Monica
Studio

Another feature we added is permanent models spawned around the displacement in real time around the camera. This is not that much different than your regular vegetation system spawning meshes on a terrain reading a mask, but instead of a painted mask we read the displacement information generated by the height field compositing pass. We can then tweak the parameters and see their effect in real time as you can see in the video.

Offscreen I'm changing the system parameters and you can see its effects here.

Art even happier

Very simple parameter driven system around displacement

Art gets taste of in-game no-authoring pipeline



This was tunable per area, and art got a taste at the power of procedural model placement in real time, and how it can adapt situationally to information that we have in our pipeline.

This not only looked great, but also showed what is possible in the future if we invest in this type of technology.

Our artists are aware of the power of procedural model placement, we use it plenty in our level editor. Seeing it running in real time, with close to no authoring cost, on a dynamically changing mesh, for an irrisory run-time cost is another thing though.

This was an invaluable lesson that I know will inform what technology we'll invest in the future.

Production Wins

Came online late. Wasn't planned

All thanks to reusable components

Real lesson: shipping a system is the ultimate test



Both of these features came online late, but we were able to squeeze them in both in terms of production schedule and in terms of performance budgets because of how malleable the system and its components were.

The information we needed was easily accessible, and it was cheap to do so.

Had we hardcoded the composite pass to be too specific or not malleable enough, we would have had to change it later in production, or we would have had to add it when we didn't have the budget.

I have mentioned we learned many lessons through this, and a key component here is that we shipped every one of them in the final game. There is a big difference between RnDing some systems for the future, and bringing it all the way through production in a final product.

Height field tech, not just snow



And we didn't just use this on snow: we used this same tech on the sands of Alfheim as well.

What's notable is that the sand was created by the artists at Bluepoint Games, our partner studios. They were able to use this technology in a different way while achieving a different look with very little input.

Greater than the sum of its parts



Santa
Monica
Studio

GOD OF WAR
SANTAMONICA

Let's look at the breakdown of all the parts of the system one last time:
this is a scene from behind Kratos' house.



This is just the base mesh with no features turned on.

Displacement



We add displacement from Kratos walking around

Height based layers



We then blend different material layers depending on height displacement

Detail normal map



We reveal the detail normal map created by the player walking around

Dynamic VFX



We then rendering particles, both opaque and transparent ones

Final: Procedural models



And finally, we add procedurally placed models

Concept vs In-Game



Santa
Monica
Studio

So this is where we started and what we shipped. I am really proud we were able to match the original vision as much as we did, and arguably, we even went beyond it.

It's worth considering that the mesh on the left is a bespoke sculpt, while the one on the right is dynamically generated just by the player running around.

We didn't forget about PS5



Santa
Monica
Studio

So far, all assets I showed you were captured on PS4. Lastly, I want to show how the system scales on PS5. I turned off the dynamic models and extra details to show how much we achieved with just good textures and tessellation.

All you see here is just the base mesh with tessellation and displacement.

In Closing - Requirements

No terrain system; arbitrary meshes ✓

Highly customizable with material features ✓

Minimal setup ✓

Aim for good average perf, no fixed memory cost ✓

Must run well/look good on PS4, scale up on PS5 ✓



We are approaching the end of the presentation. Let's summarize where we got.

First of all, we achieved all our goals.

The system is easy to use, and it's fully integrated in our standard workflow. We used it everywhere we wanted to use it.

We shipped the technology on multiple platforms with way to scale quality in between them without art having to modify any content.

We hit our performance target on both consoles, while shipping a product that lives up to the standard of the Santa Monica Studio art team.

In Closing – The Process

We shipped on features on time

We were able to expand and reuse

No major re-writes/issues



Thanks to the planning and to our approach, we were able to ship the needed features first, and then expand the system when we had time/resources to do that.

In Closing – The Future

We have tech that solved a problem, but can/will reuse easily

We won't re-invent the wheel since the wheel
is the friends we made
is the knowledge we gained along the way



We created new technologies that solved the specific need of the game, and looking at the future we have a set of tools we can, and will, easily reuse.

More importantly we have built a knowledge base and explored new avenues that have inspired us for what's to come. And those lessons are invaluable

Let's look at those takeaways all together one last time:

In Closing – Takeaways

Start from product, back of napkin math

Modular approach

Be prepared



Always start from the end goal, and while working always make sure you are in the ballpark of doable.

Force yourself into a modular approach so that no parts of your technology dictate everything around it.

Plan and be prepared for what's to come, so nothing will surprise you and you'll always have a plan.

These takeaways are high level, so you will have to break them. You and your team have to grow an intuition about when and how to do that. But if you keep on building your toolset and your knowledge with these principles, you'll tackle more problems faster, and more easily than before.

Thank you!

paolo.surricchio@sony.com

twitter: @rollingshark

mastodon: @therollingshark@mastodon.social



We are at the actual end! First of all, I want to thank you all for coming to my talk.

I want to thank the incredible team at Santa Monica Studio, and in particular the people who helped putting this presentation together: my lead, tech director, manager, Kevin, Kyle, Chris, and Valerio, and my advisor Julien.

We have quite a few presentations on God of War: Ragnarök this year at GDC. Go see them all!

I'm going to leave my contact in the next slide up, feel free to reach out with questions or just to say hi!

We have some time for questions, please line up to the mics and ask.

JOIN US AT GDC 2023



BUILD YOUR GOD OF WAR GDC AT:
SCHEDULE.GDCONF.COM



BRUNO VELAZQUEZ • ANIMATION DIRECTOR
DAVID GIBSON • ANIMATION DIRECTOR
ERICA PINTO • LEAD NARRATIVE ANIMATOR
MENDI YSFER • LEAD GAMEPLAY ANIMATOR

Animation in 'God of War Ragnarök' • Animation Summit
MONDAY, MARCH 20 • 9:30AM – 10:30AM • ROOM 303, SOUTH HALL

SUE PACETE • SR USER RESEARCHER

Playtesting 'God of War Ragnarök' Accessibility Options • UX Summit
MONDAY, MARCH 20 • 1:20PM – 1:50PM • ROOM 2001, WEST HALL



PAOLO SURRICCHIO • SR STAFF PROGRAMMER

Reinventing the Wheel for Snow Rendering • Advanced Graphics Summit
MONDAY, MARCH 20 • 1:20PM – 2:20PM • ROOM 301, SOUTH HALL



BEN HINES • SR STAFF DEVOPS ENGINEER

Automated Testing at Santa Monica Studio • Tools Summit
MONDAY, MARCH 20 • 4:40PM – 5:10PM • ROOM 3004, WEST HALL



XUANYI ZHOU • PROGRAMMER

Real-time Neural Texture Upsampling in 'God of War Ragnarök' •
Machine Learning Summit
TUESDAY, MARCH 21 • 2:10PM – 2:40PM • ROOM 2010, WEST HALL



ETHAN AYER • SR ENVIRONMENT ARTIST

The Art of Making Vistas • Art Summit
TUESDAY, MARCH 21 • 3:00PM – 3:30PM • ROOM 3007, WEST HALL



GÖKSU UĞUR • AI LEAD

Preparing AI Systems For 'God of War Ragnarök' • Programming
WEDNESDAY, MARCH 22 • 9:00AM – 10:00AM • ROOM 303, SOUTH HALL



VICKI SMITH • SR STAFF LEVEL DESIGNER

The Final Battle of 'God of War Ragnarök': Techniques For Delivering
High-stakes Sequences • Design
WEDNESDAY, MARCH 22 • 10:30AM – 11:00AM • ROOM 2002, WEST HALL



STEPHEN MCAULEY • LEAD RENDERING PROGRAMMER

Rendering 'God of War Ragnarök' • Programming
WEDNESDAY, MARCH 22 • 2:00PM – 3:00PM • ROOM 303, SOUTH HALL



ERIC GOTTESMAN • SR STAFF DEVOPS ENGINEER

Modernizing multiplayer services for 'God of War: Ascension' (PS3) •
Production & Team Leadership • Presented by Amazon Web Services
WEDNESDAY, MARCH 22 • 2:00PM – 3:00PM • GDC PARTNER STAGE, EXPO FLOOR, NORTH HALL



SAM STERNKLAR • SR PROGRAMMER

'God of War Ragnarök': Visual Scripting Solution • Programming
THURSDAY, MARCH 23 • 10:00AM – 11:00AM • ROOM 2006, WEST HALL



ADAM OLIVER • SR COMBAT DESIGNER

Breaking Barriers: Combat Accessibility in 'God of War Ragnarök' • Design
THURSDAY, MARCH 23 • 2:00PM – 2:30PM • ROOM 2002, WEST HALL



GÖKSU UĞUR • AI LEAD

Practical Tools for Transitioning Into Leadership Roles • Leadership
THURSDAY, MARCH 23 • 2:00PM – 2:30PM • ROOM 303, SOUTH HALL



ZACH BOHN • SR STAFF TECHNICAL UI DESIGNER

'God of War Ragnarök': Building The UI For a AAA Title • Design
THURSDAY, MARCH 23 • 4:00PM – 5:00PM • ROOM 303, SOUTH HALL



SALAZAR KOHARI • PROGRAMMER

Companion Traversal in 'God of War Ragnarök' • Programming
FRIDAY, MARCH 24 • 10:00AM – 11:00AM • ROOM 2002, WEST HALL



TENGHAO WANG • SR PROGRAMMER

Joint-based Skin Deformation in 'God of War Ragnarök' • Programming
FRIDAY, MARCH 24 • 1:30PM – 2:30PM • ROOM 2006, WEST HALL



HARLEIGH AOWNER • TECHNICAL NARRATIVE DESIGNER

How to Build a Home: Designing Narrative For Sindri's House in 'God of War
Ragnarök' • Design
FRIDAY, MARCH 24 • 3:00PM – 3:30PM • ROOM 2001, WEST HALL

Santa Monica Studio GDC

We have a big presence here at GDC. Go watch all these talks, they are all really interesting!


Santa Monica Studio

Our journey
Your story

We're hiring for what's next!

We're expanding our family across disciplines and would love to meet you. Please visit sms.playstation.com/careers for all openings or drop us a line at sms.recruiting@sony.com

 @santamonicastudio

 @SonySantaMonica

 @santamonicastudio

And we are hiring!



References

GPU-Driven Rendering Pipelines: Ulrich Haar, Sebastian Aaltonen

<https://advances.realtimerendering.com/s2015/>

Boots on the Ground: The Terrain of Call of Duty: J.T. Hooker

<https://research.activision.com/publications/2021/09/boots-on-the-ground-the-terrain-of-call-of-duty>

Optimizing the Graphics Pipeline With Compute: Graham Wihlidal

<https://www.gdcvault.com/play/1023109/Optimizing-the-Graphics-Pipeline-With>

Lifting the Fog: Geometry & Lighting in 'Demon's Souls': Bruce Woodard

<https://gdcvault.com/play/1027011/Advanced-Graphics-Summit-Lifting-the>

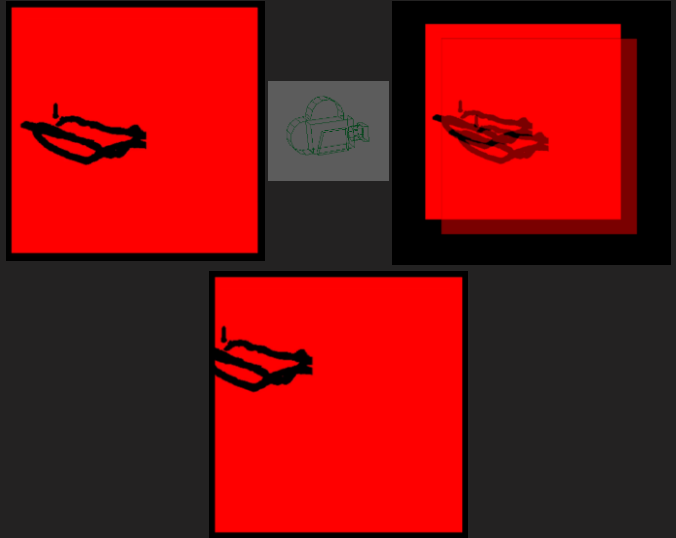
Bonus – Copy Scroll

Persistent copy scroll
Ping-pong 2 render targets

As camera move, lock to
pixel increment

Copy previous to current
with a -X/-Y offset

Clear to expected default color



As the camera moves, we lock it to pixel movement. We then “scroll” the texture in the opposite direction, and clear the new area to the default color that makes sense with our technique

Bonus – Shadows

Screen space shadows

No budget on PS4 to do tessellation in shadows

Meshlet culling would have to be aware of shadow influence

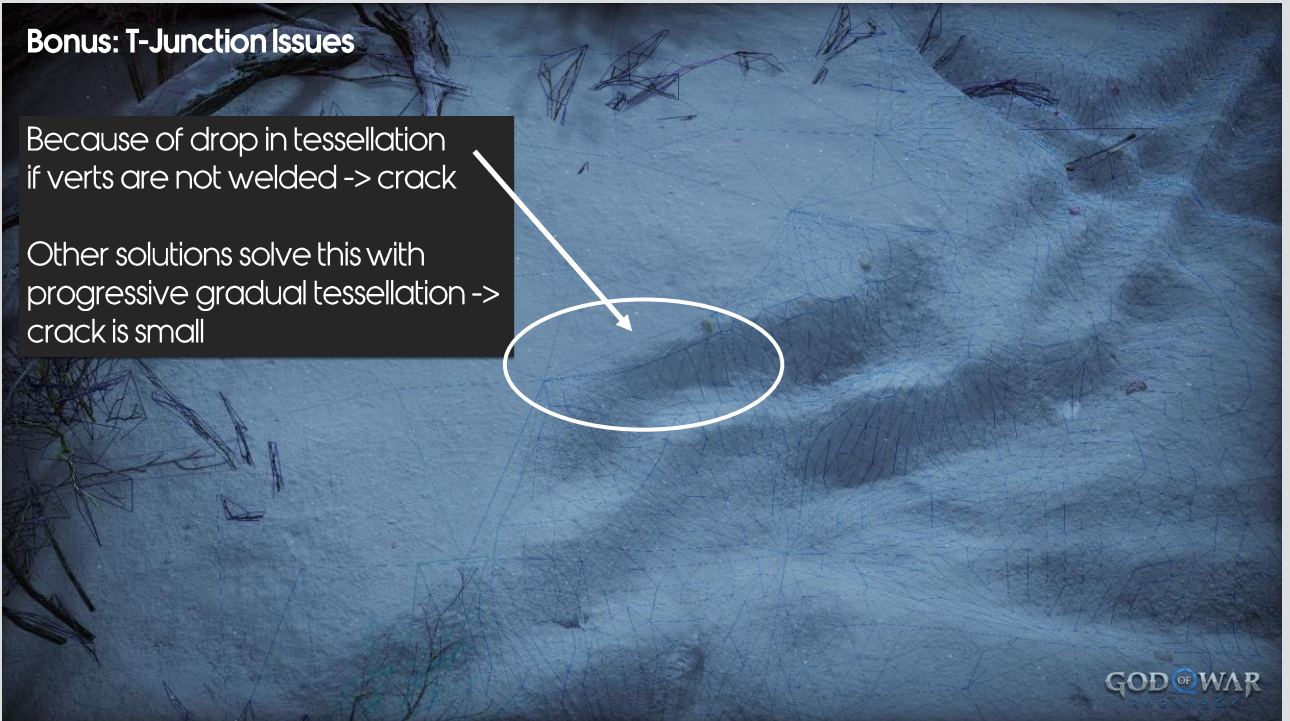


We only used screen space shadows since re-rendering the mesh in the shadow map with tessellation was too expensive

Bonus: T-Junction Issues

Because of drop in tessellation
if verts are not welded -> crack

Other solutions solve this with
progressive gradual tessellation ->
crack is small



Since we don't tessellate gradually, and instead tessellate only where we need to, we might have edges that have a big subdivision difference on the two sides.

If we have a t-junction, it's likely we'll have a big gap

Other solutions addressed this concern by having a more progressive-gradual tessellation, so the difference between 2 sides of the same edge is not that big.

In that case, the gap is less likely to happen, and when it happens it's smaller

Bonus – Partial carving

Avoid knife-through-butter effect

When writing, you know displacement you want to have

Write only proportion by reading previous frame information

