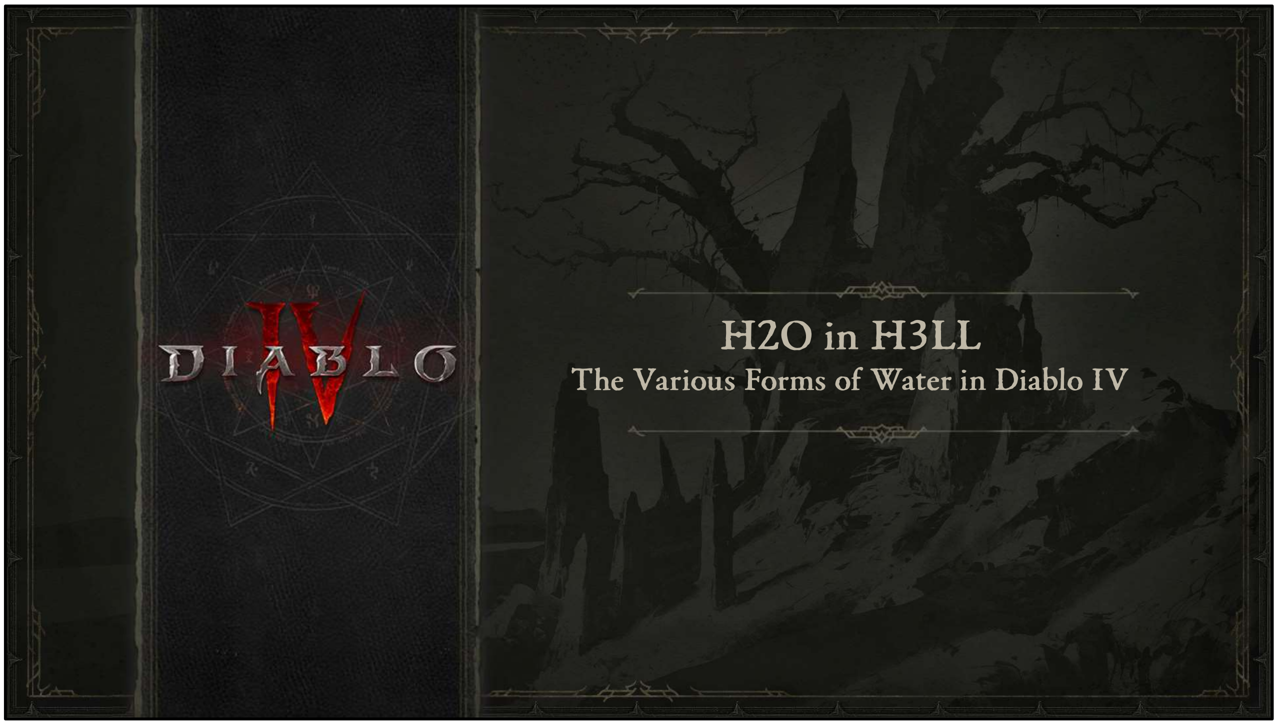
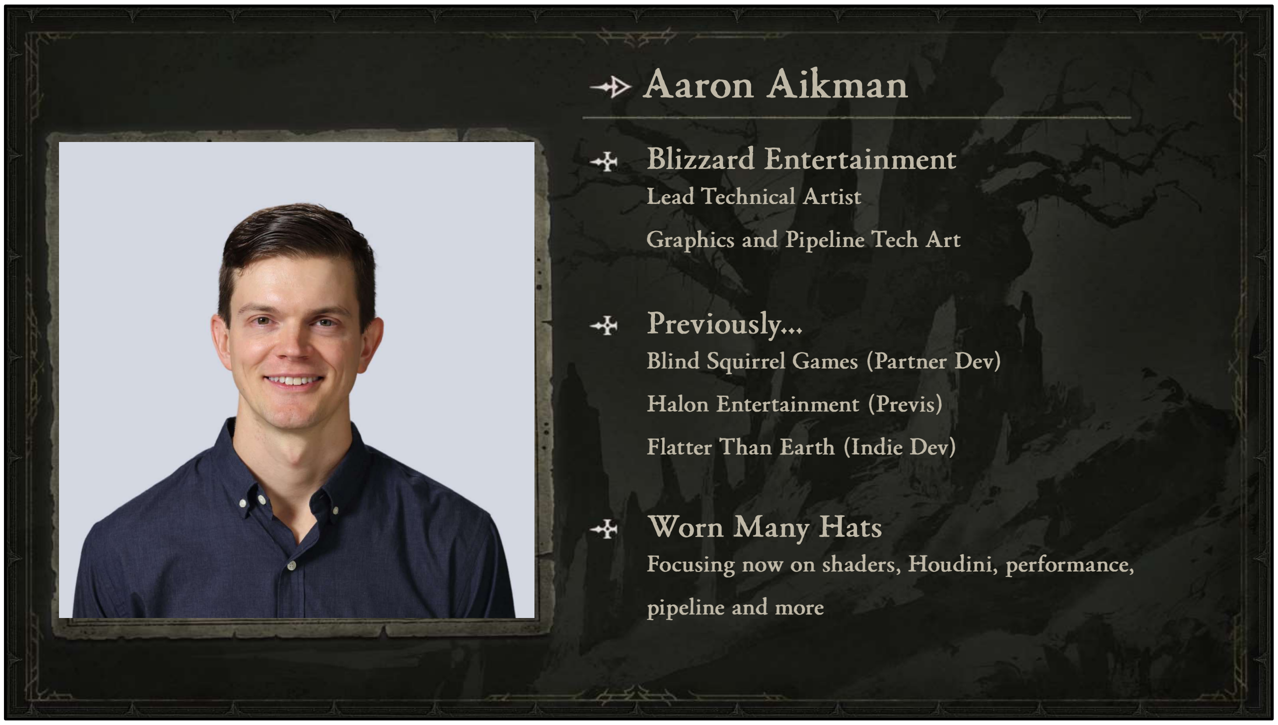


H2O in H3LL
The Various Forms of Water in Diablo IV

Hello! And welcome to H2O in H3LL: The Various Forms of Water in Diablo IV

BTW: This deck has a lot of videos and animations, so I highly recommend looking at the full ppt or presentation recording as well.



Aaron Aikman

Blizzard Entertainment
Lead Technical Artist
Graphics and Pipeline Tech Art

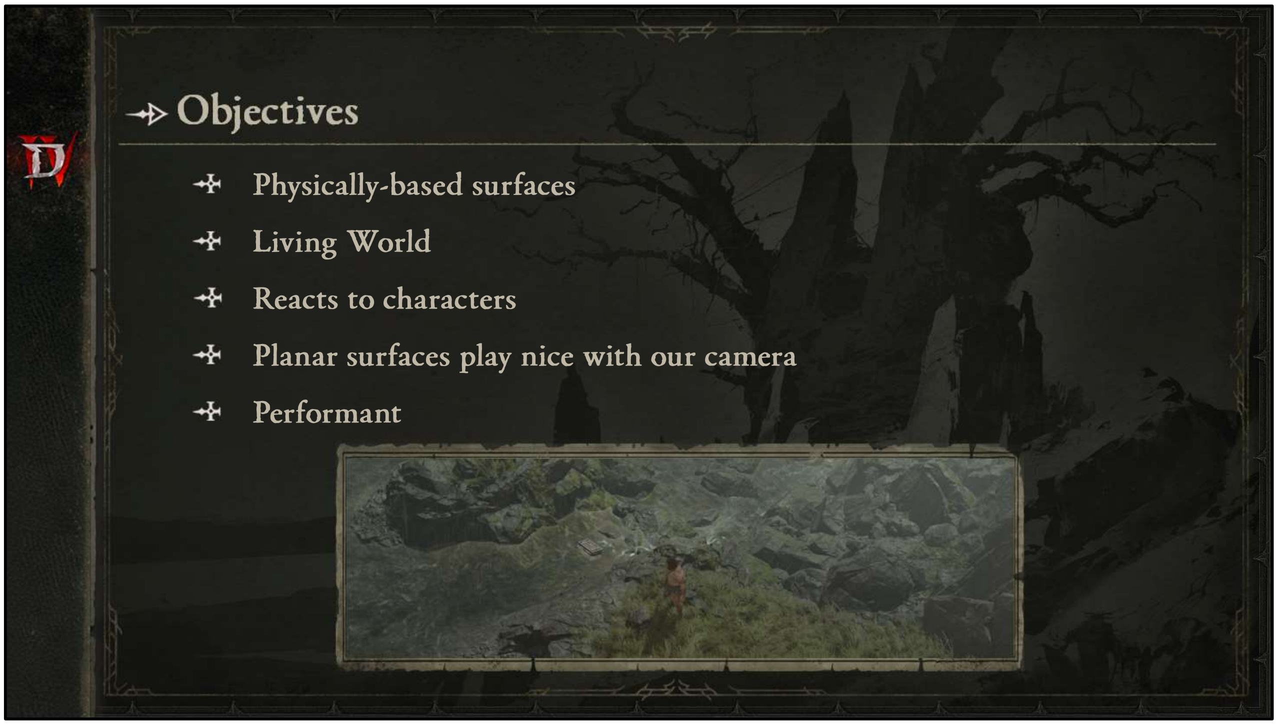
Previously...
Blind Squirrel Games (Partner Dev)
Halon Entertainment (Previs)
Flatter Than Earth (Indie Dev)

Worn Many Hats
Focusing now on shaders, Houdini, performance, pipeline and more

My name is Aaron Aikman and I'm a Lead Technical Artist at Blizzard Entertainment.
I lead the Graphics and Pipeline Tech Art groups on Diablo IV.

I previously worked at Blind Squirrel Games which does a lot of partner development with other studios, among other things.
Before that I worked in Film at Halon Entertainment doing mainly Previs in Unreal, which is where you effectively plan out the most important scenes in the movie so they can be "previsualized" by directors and editors, before being sent off for final pixels.
And before that I was an indie dev at a company called Flatter Than Earth, doing a lot of tech art, but also 3D character and environment art.

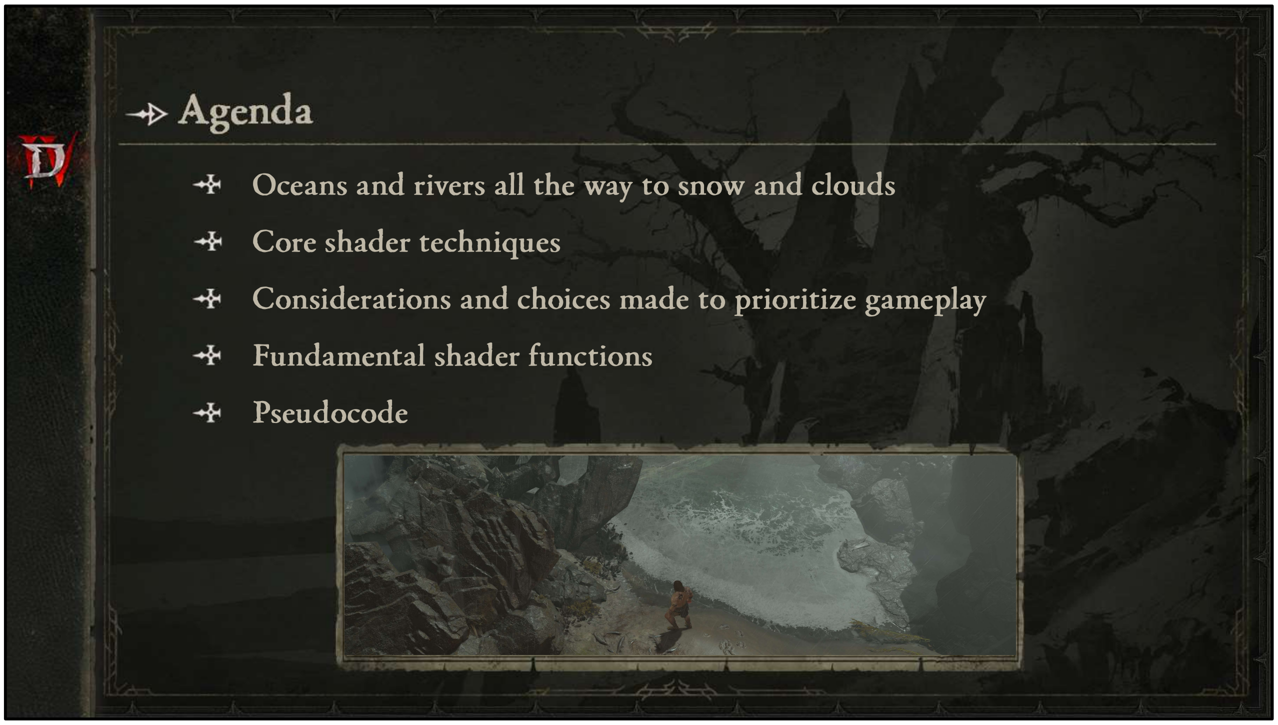
All of that is to say, I've worn many tech art hats at progressively larger studios, but now I'm focusing mostly on shaders, Houdini, and performance, which I like to call the "graphics" hat as well as wearing the "pipeline & tools" hat



Objectives

- Physically-based surfaces
- Living World
- Reacts to characters
- Planar surfaces play nice with our camera
- Performant

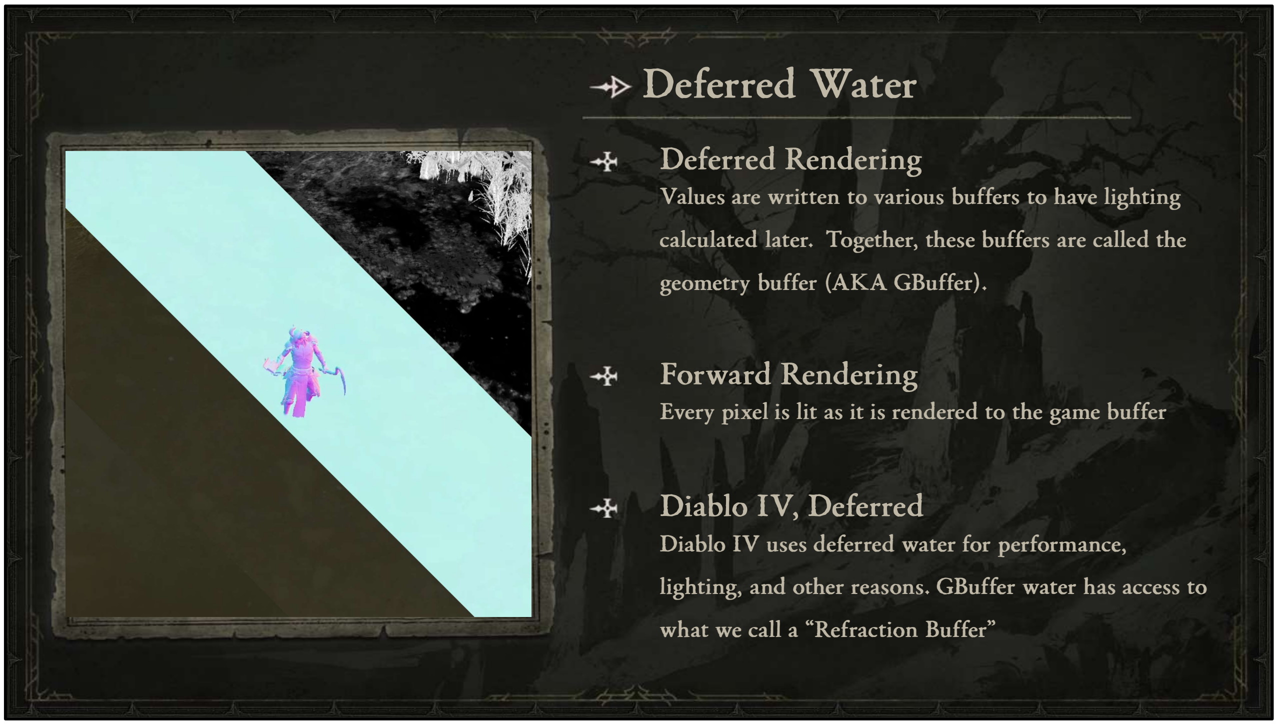
Alright so first let's set our objectives.
For Diablo IV, we wanted Physically-based surfaces.
We wanted the environment to feel like a living natural world
For water, this also means that it should react to characters
Planar surfaces like water and ice should play nice with our camera angle.
And of course, the game has to be performant, meaning that it runs smoothly on a variety of platforms. If you'd like to learn more about performance on our game, I highly recommend Evan Edwards' talk "Embracing Art Performance on 'Diablo IV'".



Agenda

- Oceans and rivers all the way to snow and clouds
- Core shader techniques
- Considerations and choices made to prioritize gameplay
- Fundamental shader functions
- Pseudocode

During this talk, we will be covering everything from oceans and rivers all the way to snow and clouds.
I'm going to go over the core pixel and vertex shader features used to achieve the effects that you can see
I’ll talk about the considerations and choices that we had to make to prioritize gameplay.
For those of you new to shaders or still getting your bearings, I’ll go over a few fundamental shader functions that were used to make these effects possible, breaking them down in a digestible way.
And for those of you who want to see a bit deeper into some nuts and bolts, I'll also go over some pseudocode - not necessarily code that you could plug straight in, but a code-based approach you could to take to implementation



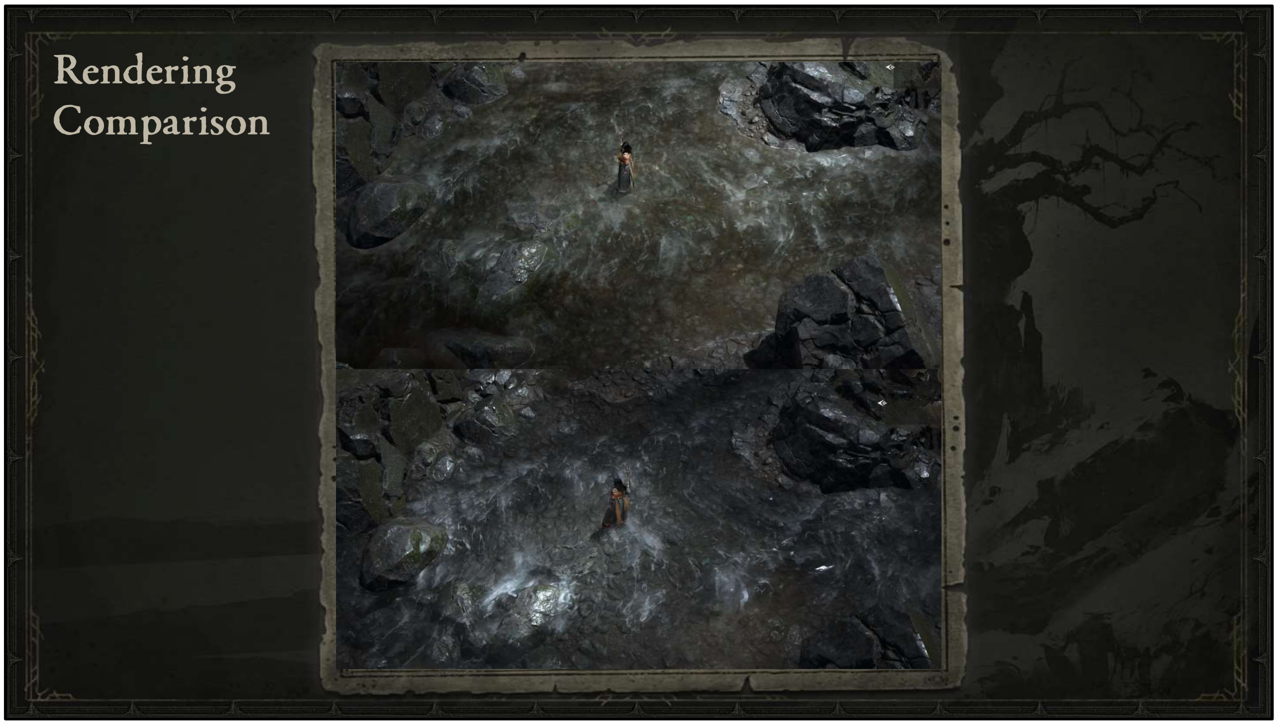
Deferred Water

- **Deferred Rendering**
  Values are written to various buffers to have lighting calculated later. Together, these buffers are called the geometry buffer (AKA GBuffer).
- **Forward Rendering**
  Every pixel is lit as it is rendered to the game buffer
- **Diablo IV, Deferred**
  Diablo IV uses deferred water for performance, lighting, and other reasons. GBuffer water has access to what we call a "Refraction Buffer"

Before anything else, let's talk about deferred rendering.
Deferred rendering is when material properties are written to various buffers (which are like textures) to have lighting calculated later. Together these buffers are called the geometry buffer (AKA GBuffer)
You can see a breakdown of the Gbuffer pictured here on the screen.

This is in contrast with Forward rendering, which is where every pixel is lit as it is rendered to game buffer.
With forward rendering costs for additional lights add up quickly and depending upon render order, you could easily end up paying to calculate lighting for a pixel multiple times.

For Diablo IV, we ultimately decided to use deferred water mainly because of performance and enable higher light counts.
Many skills and powers bring dynamic lighting with them so using deferred lighting help to enable that without running the risk of lights being dropped.
With Gbuffer water, we do have access to what we call a Refraction Buffer, which is effectively the albedo of the geometry buffer layers underneath the water, with the option to distort the buffer based upon normal information and to modify color based upon scene depth. You'll see more of this a bit later.



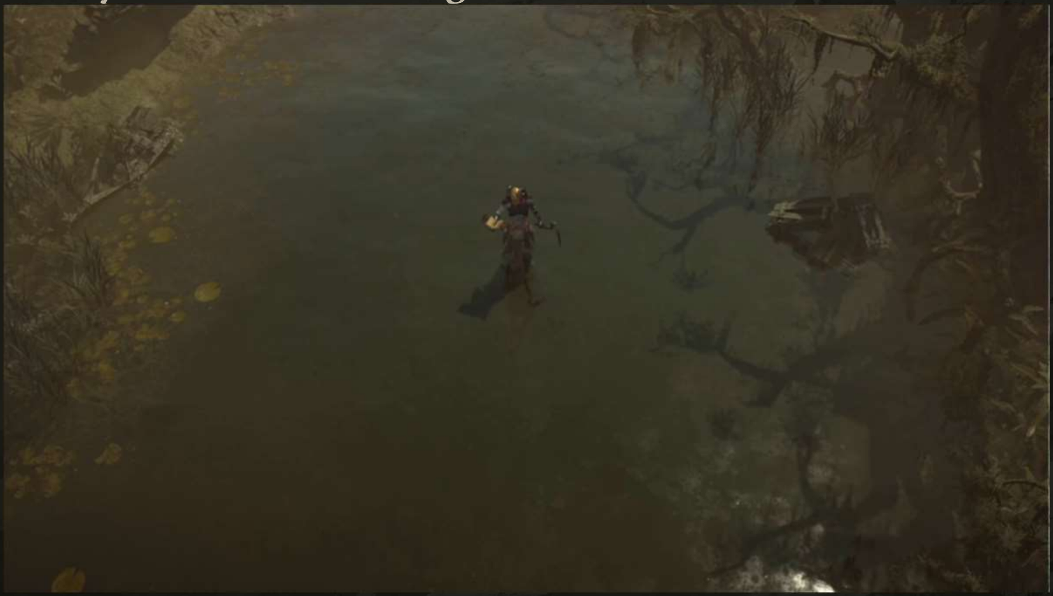
Rendering Comparison

Here you can see a comparison with Gbuffer water on top and forward rendered water on the bottom.
With the deferred water, you may notice that the edges near the rocks and dirt are blending Gbuffer values with layers underneath, which I think actually creates a nice specular hit.

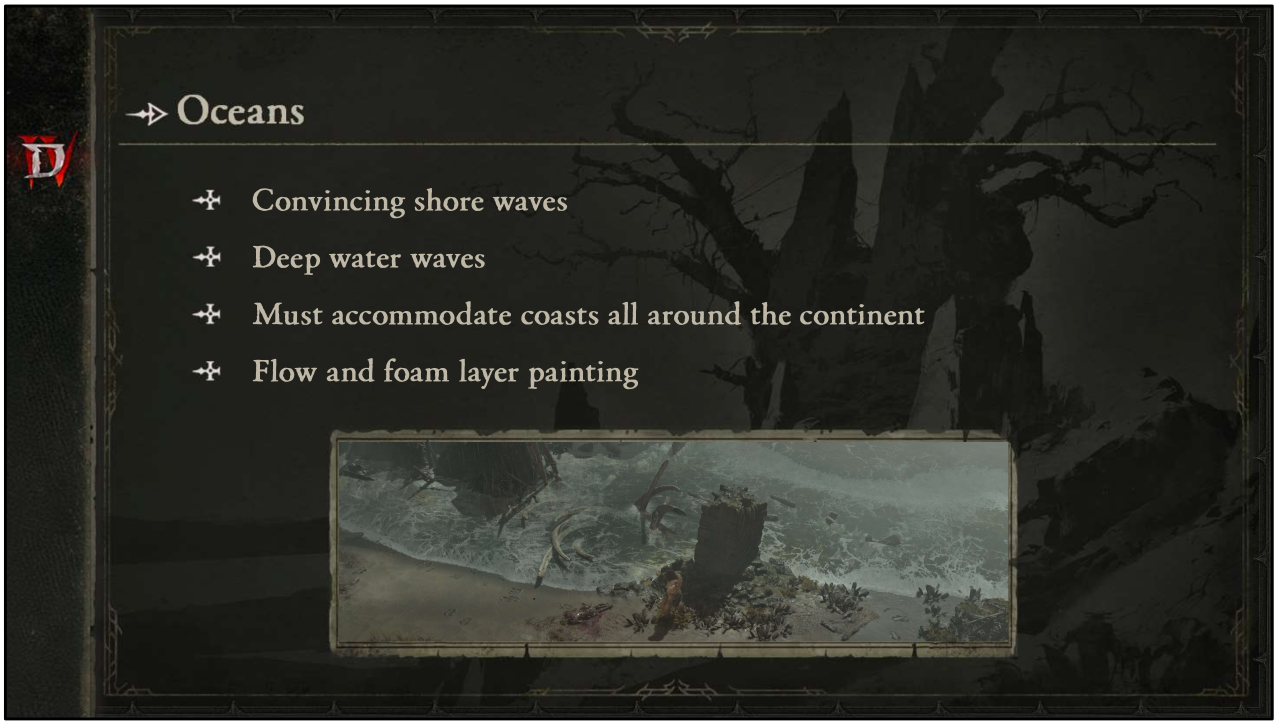
With forward rendered water, especially in this particular example, you will notice how meshes and normal maps are shadowed unlike with deferred water where we're simply sample albedo values.
Many game utilize forward-rendered water since that suit their gameplay purposes, but Diablo IV is very much about fast-paced gameplay in a physically-based living world.

This is purely a visual example, but what this doesn't convey is the non-trivial performance savings from using Gbuffer water.

Geometry Buffer Walkthrough

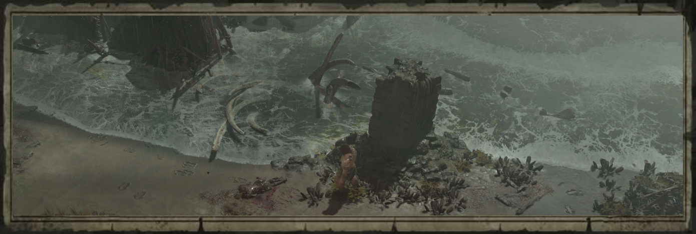


Here's a breakdown of the GBuffer.
The albedo is effectively the color value of each surface. You can also think of this as a measure of what colors are reflects by the surface instead of absorbed.
The normal describes the direction the surface is facing down to a texture level
And roughness further elaborates on that direction by describing how much light is diffused when it bounces off a surface, effectively controlling how broad the highlight is.
And then back to the everything combined, the whole image with the G buffer.



Oceans

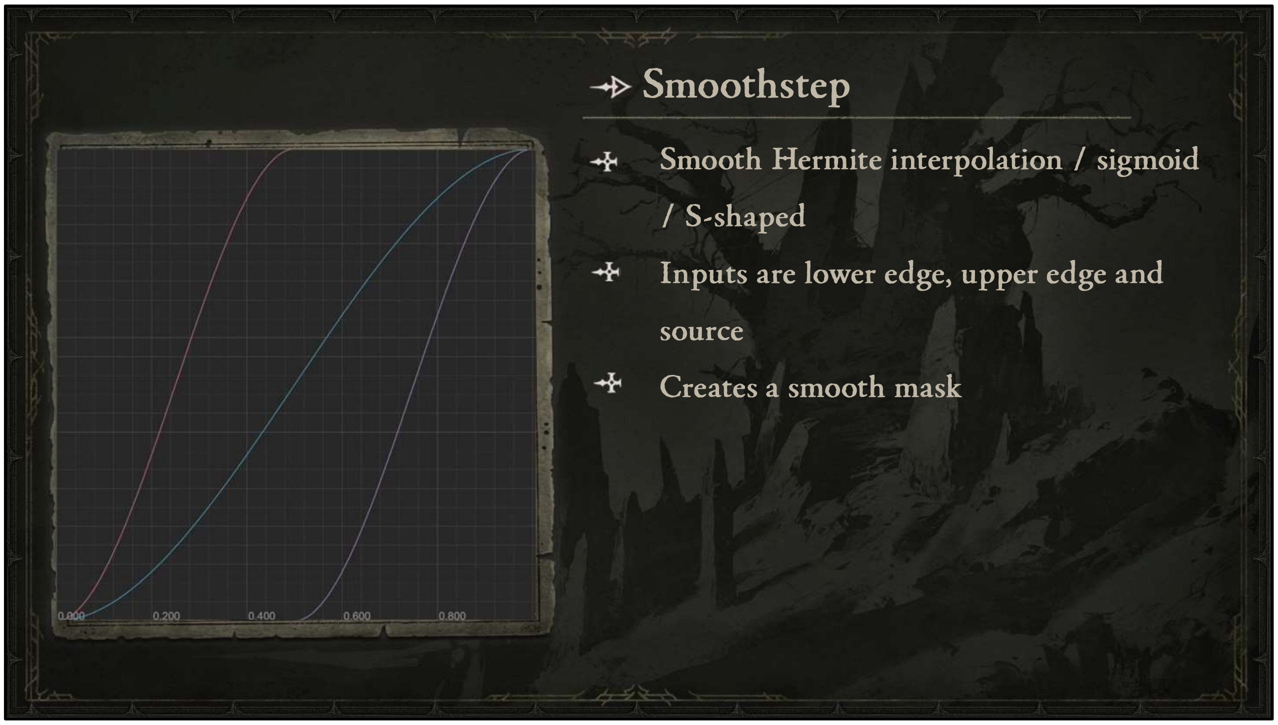
- Convincing shore waves
- Deep water waves
- Must accommodate coasts all around the continent
- Flow and foam layer painting



First stop on our shader tour is the ocean
One of the main things we wanted to achieve with oceans was having convincing shore waves, having the feeling of the water ebbing and flowing convincingly.
But we also wanted to be able to convey a sense of deep and turbulent water in places.
These techniques needed to accommodate coasts all around the continent, looking as good as possible without much effort, but with the option to add fine polish where needed.
A key part of creating a sense of water flow and turbulence was the utilization of flow and foam painting.

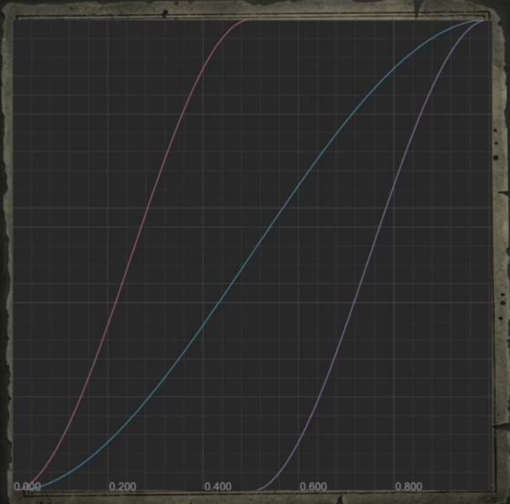
Ocean Showcase

To start things off, here's a brief tour of some of the Coastal areas in Scosglen.
Note that depth fog is disabled here to make the ocean shader more clear.
I'll let the waves do most of the talking here as the video guides us on a journey through sandy shores to rocky coves, and everything in between.

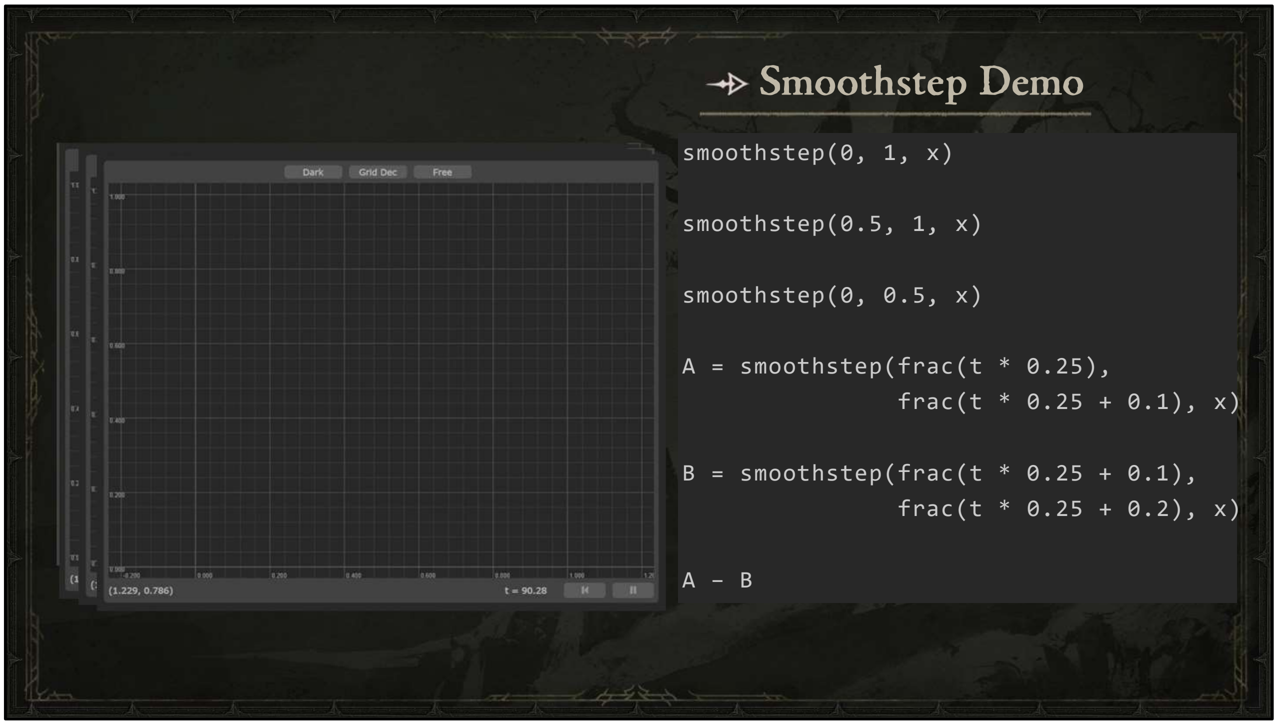


Smoothstep

- Smooth Hermite interpolation / sigmoid / S-shaped
- Inputs are lower edge, upper edge and source
- Creates a smooth mask



Now we've come to our first Function Shader Fundamental.
This is one of my personal favorites and was instrumental for many shaders but especially the ocean.
And that is the Smoothstep.
So smooth step allows you to do smooth Hermite interpolation which can also be referred to as sigmoid which effectively means S-shaped.
Smoothstep inputs include the lower edge, upper edge, and source. What this allows you to do is create a nice smoothly interpolated value based upon those edge inputs.
You can see in this visualization various smoothstepping effects on the same source data.

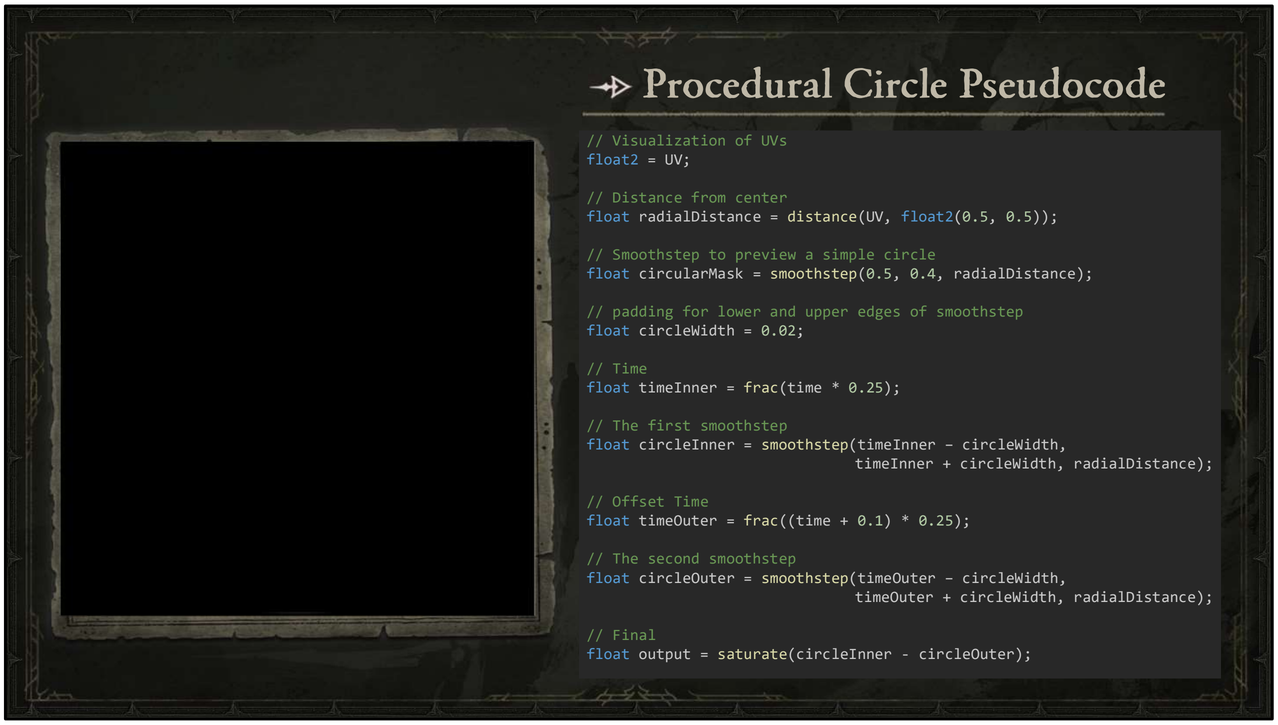


Smoothstep Demo

```
smoothstep(0, 1, x)

smoothstep(0.5, 1, x)

smoothstep(0, 0.5, x)

A = smoothstep(frac(t * 0.25),
               frac(t * 0.25 + 0.1), x)

B = smoothstep(frac(t * 0.25 + 0.1),
               frac(t * 0.25 + 0.2), x)

A - B
```

Alright so if we use GraphToy.com to map out values with smoothstep, the first thing we'll see is a straight yellow line which is increasing linearly from 0 in the bottom left to 1 in the upper right..
This is x. . .You'll also see a blue s-shaped line here which demonstrates the shape created when smoothstepping even if the lower edge is 0 and the upper edge is 1.
If we set the lower edge to 0.5, we'll see a similar shape, but everything below 0.5 is now 0.
And if we go back and set the lower edge to 0 and the upper edge to 0.5, you'll see the same shape a third time, but with every value above 0.5 becoming 1.

Now you can also change the lower and upper edge over time. So if we take only the factional component of t, which is time, multiplied by 0.25 to slow it down, we could use that for the lower edge.
And do the same thing for the upper edge, but offset it by 0.1 to create the "width" of our S shape.
We could call this line A.
Then we could do the same thing again, but offset both values a bit more to move it over.
We could call this line B.
Now if we subtract B from A, we'll get a nice peak that travels across the screen.



Procedural Circle Pseudocode

```
// Visualization of UVs
float2 = UV;

// Distance from center
float radialDistance = distance(UV, float2(0.5, 0.5));

// Smoothstep to preview a simple circle
float circularMask = smoothstep(0.5, 0.4, radialDistance);

// padding for lower and upper edges of smoothstep
float circleWidth = 0.02;

// Time
float timeInner = frac(time * 0.25);

// The first smoothstep
float circleInner = smoothstep(timeInner - circleWidth,
                               timeInner + circleWidth, radialDistance);

// Offset Time
float timeOuter = frac((time + 0.1) * 0.25);

// The second smoothstep
float circleOuter = smoothstep(timeOuter - circleWidth,
                               timeOuter + circleWidth, radialDistance);

// Final
float output = saturate(circleInner - circleOuter);
```

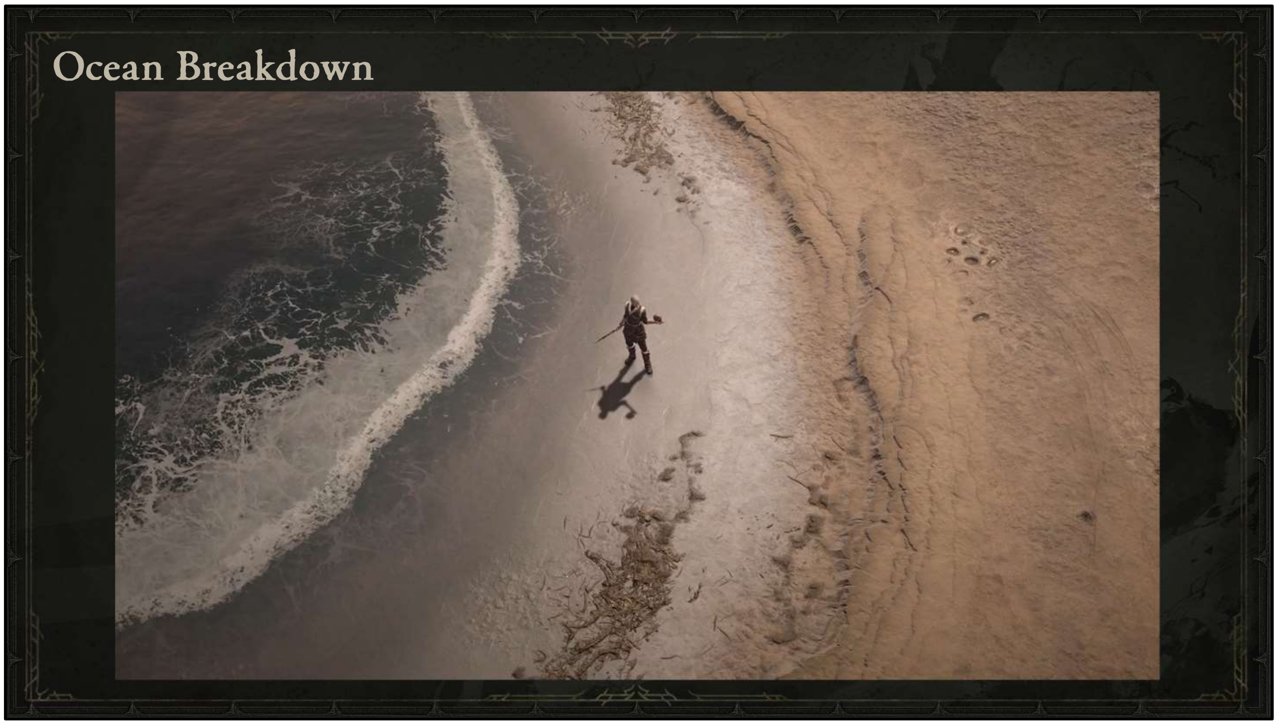
First we'll visualize basic UVs on a square.
If we take the distance from the UV to the center, we can create a radial mask.
We could then modify this mask using smoothstep with 0.5, and 0.4. Note that this is mainly to help visualize this next step.
But first, a bit of setup.

We'll create a variable to define the width of our procedural circle.

Then we'll create a variable that is just a frac of time multiplied by 0.25 to slow it down. We'll use this for our first smoothstep mask which also takes the circleWidth into account, and uses the radialDistance for the source data.

We'll do something similar again, with the key difference being a slight offset to time.

Then if we subtract one from the other, we can a shrinking procedural circle using smoothstep.

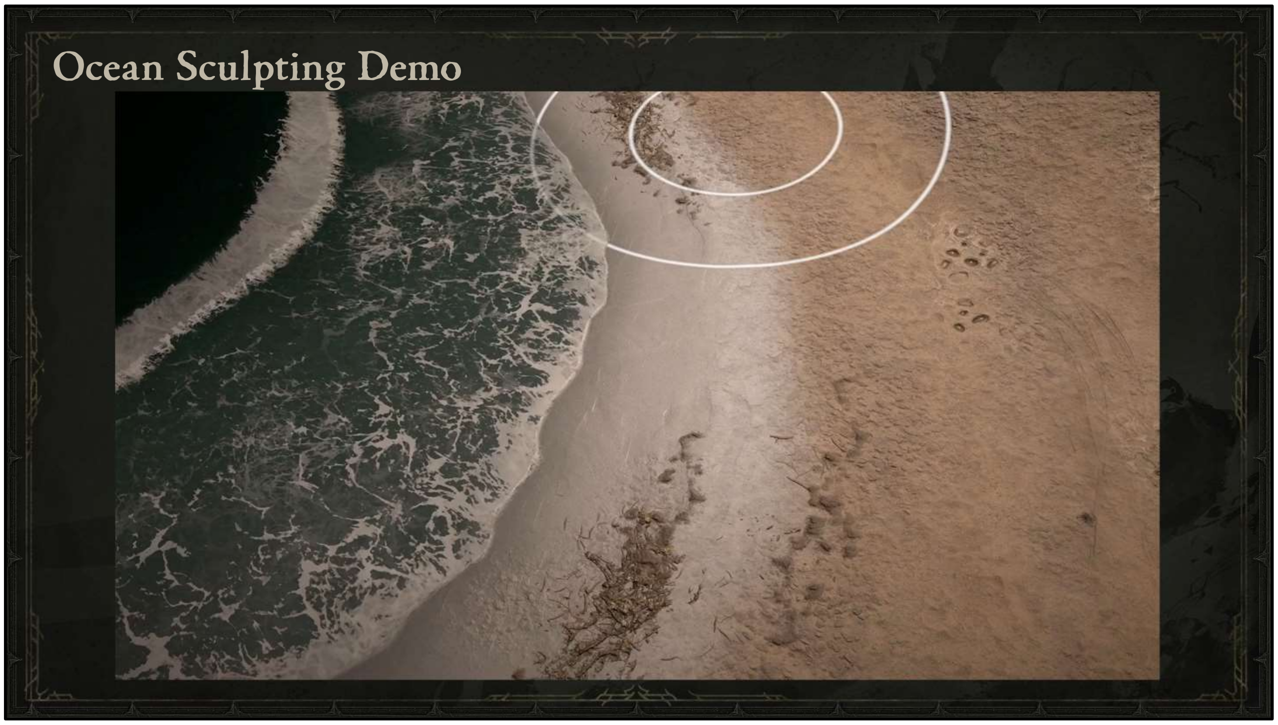


Ocean Breakdown

Here you can see a breakdown of the ocean shader in a very simple area
First, we'll take a look at everything together.
On the previous slides, I demonstrated how you can smoothstep based upon different data to create masks.
For the ocean, that data is the depth of the terrain underneath the water, basically absolute world Z position.
If we take only the fractional component of that height, we can see where one meter is.
. .and then two. . .and so on.
Next we can start to smoothstep through that depth using time…
If we offset that smoothstep slightly. . .and then subtract one from the other, we can get a mask advancing towards the shore.
Now with a bit more polish and some repetition, we can begin to construct some nice "wave lines", as I like to call them. These represent the front of the wave.
Here we have the mask for the back of the wave.
These masks are used to bring in different foam textures, which help to create more convincing feeling of water being agitated as it moves…
Here we can see the foam texture used to emulate water falling over the top of the cresting wave. Notice how important the flow is for the sense of motion.
And here they are combined together again.

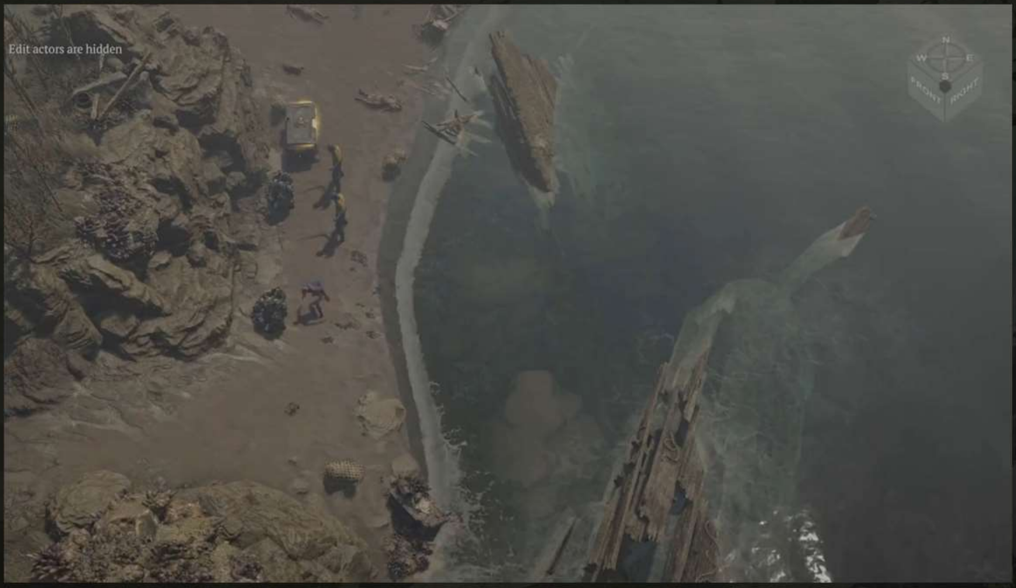
Now if we were to remove the flowmap, you would see how the smoothstep still advances toward the shore, but the textures themselves aren't moving naturally.
With the flowmap back on, we feel a much more natural motion to the water.
Next, we'll take a look at the two foam layers that can be painted in. This foam layer is meant to be more transient and can be painted broadly.
And this layer is for more consistently frothy areas.
Next you can see a very simple emulation of caustics which is just a texture with uv distortion that is masked in based upon terrain depth.
And here's what it looks like if we combine the albedo back together. But that's only part of the equation.
We also want to set roughness, especially for the foam where the highlight would be much more diffused.
Another key aspect is vertex animation, which we've had enabled this whole time.
Here's what it would look like with no vertex animation. Note that the textures move the same, but the water plane doesn't actually flow and recede underneath the player.
If we bring back all of the elements you can see the albedo, roughness, and vertex animation working in concert.

Ocean Sculpting Demo

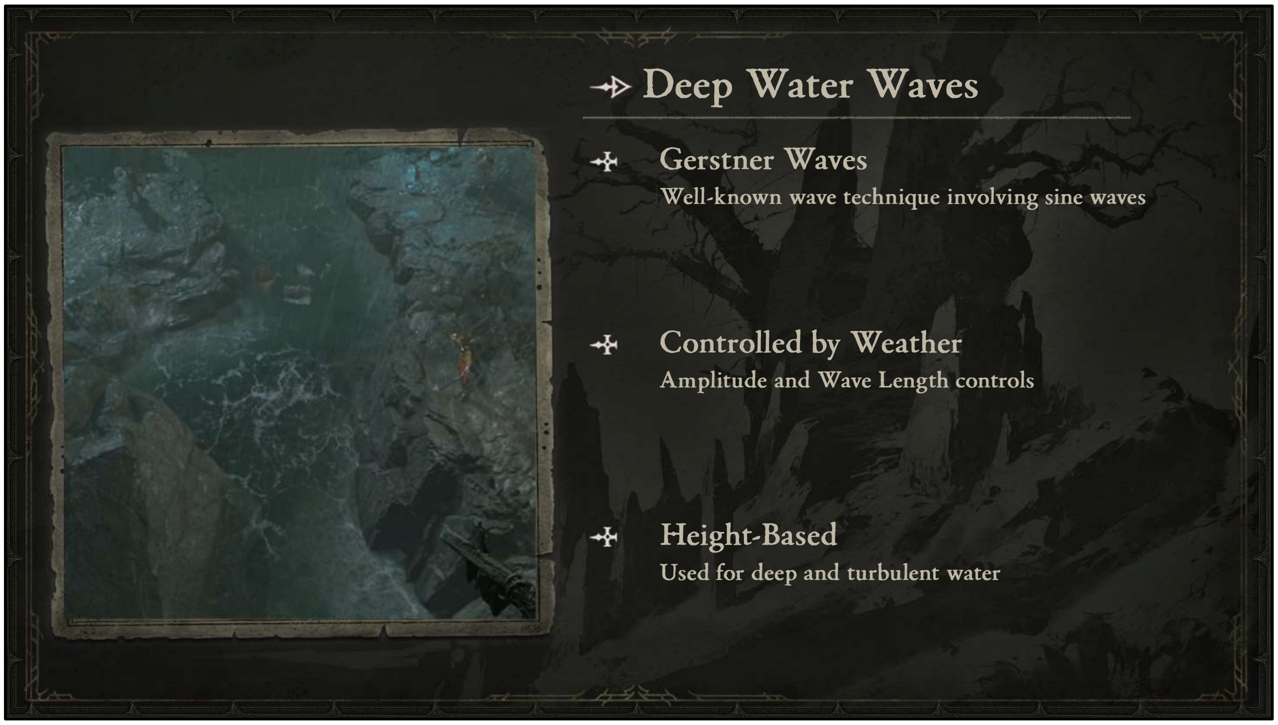


Here I'm going to demonstrate exactly how impactful the height of the terrain is on the timing and shape of the wave.
I'll start by doing some dramatic changes with a ramp brush, which sets a grade from the first click to mouse up.
Next I'll raise a portion of the terrain with the regular sculpting brush. . . And then smooth it out a bit.
You can see how we already have a nice shape forming for the purposes of our demo here.
Next I'll push the terrain down a bit. You'll notice that I smooth things out to make the change in elevation more gradual. This helps to keep the pacing of the wave even, but note that you can effectively control the speed of the wave based upon the rate of change in terrain height.
Another useful technique is to pause the game so that you can shape the wave at a particular moment.
I like to call this whole process "sculpting the waves"

Ocean Sculpting Timelapse



Like I mentioned before the intention with the ocean is that it looks good “out of the box” with the option to add plenty of polish.
So here’s a timelapse of some polish I put into a portion of the Hawezar coast.
You’ll notice right off the bat that another technique I use is to disable the water so I can focus on the shape of the terrain underneath.
You’ll also see me pausing and unpausing a number of times.
Here, I’m pushing and pulling the terrain to give the sense of waves flowing around different objects.
You’ll also notice a lot of smoothing when I want to even the shape of the wave out.
For the rest of the timelapse, I’m going to mostly let the brushstrokes do the talking, though I’ll convey a couple more notes along the way.
Here you can see me enabling a view of our navmesh which determines where the player can walk.
It’s important to be consistent about how deep into the water the player feels like they can go.
And here you can see a brief pan through of the final result.



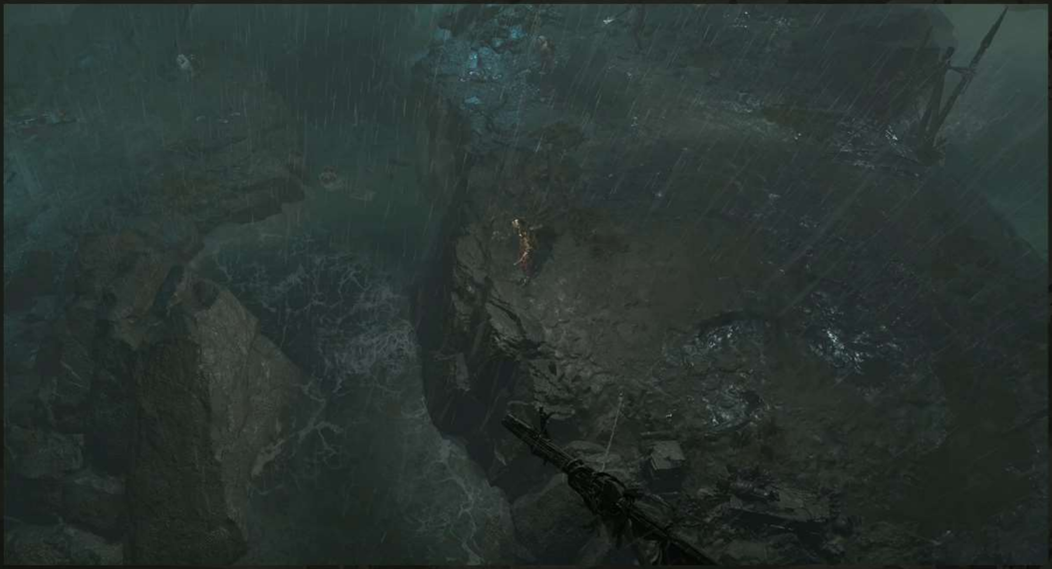
Deep Water Waves

- **Gerstner Waves**
  Well-known wave technique involving sine waves
- **Controlled by Weather**
  Amplitude and Wave Length controls
- **Height-Based**
  Used for deep and turbulent water

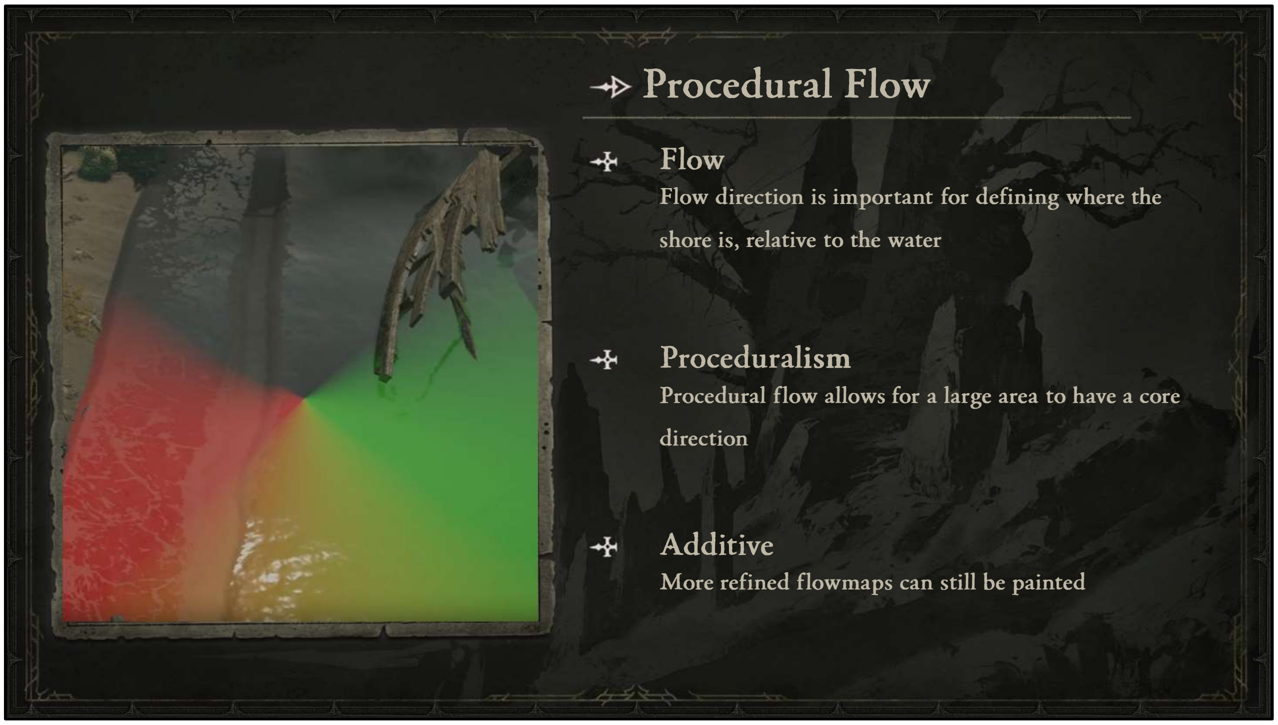
Next, on to Deep Water Waves.

Our Deep water areas use “Gerstner” waves, which is a well known technique that involves combining sine waves to create convincing vertex offsets.
We have control over this in the weather for things like the amplitude and length of the wave
And they are height-based, which means they are used wherever the depth of the terrain is low enough. So this is another case where the height of the terrain drives the behavior of the waves.

Gerstner Showcase

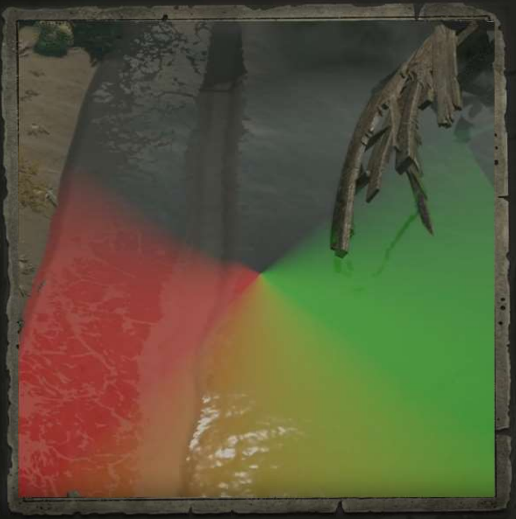


I'm not going to go into too much detail on our Gerstner waves, but here's a fairly pronounced example in game.
You'll also notice that we have buoyant objects utilizing the same displacement there at the top of the screen..

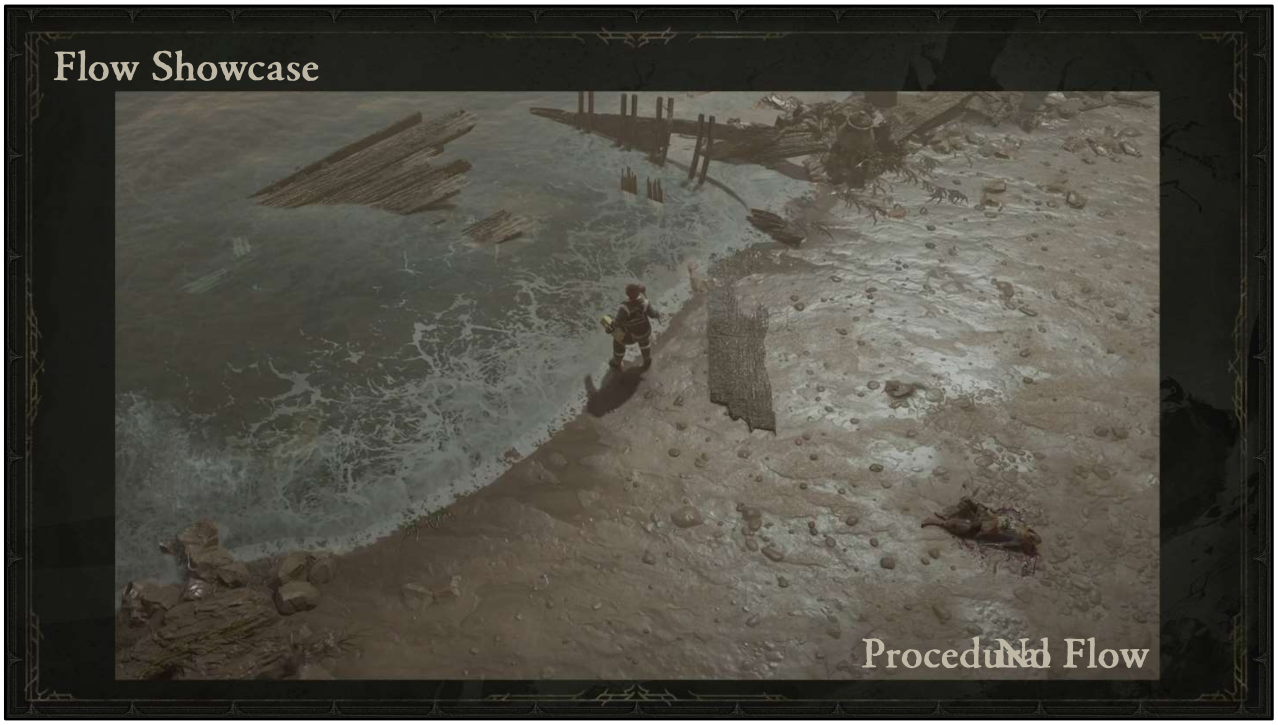


Procedural Flow

- **Flow**
  Flow direction is important for defining where the shore is, relative to the water
- **Proceduralism**
  Procedural flow allows for a large area to have a core direction
- **Additive**
  More refined flowmaps can still be painted

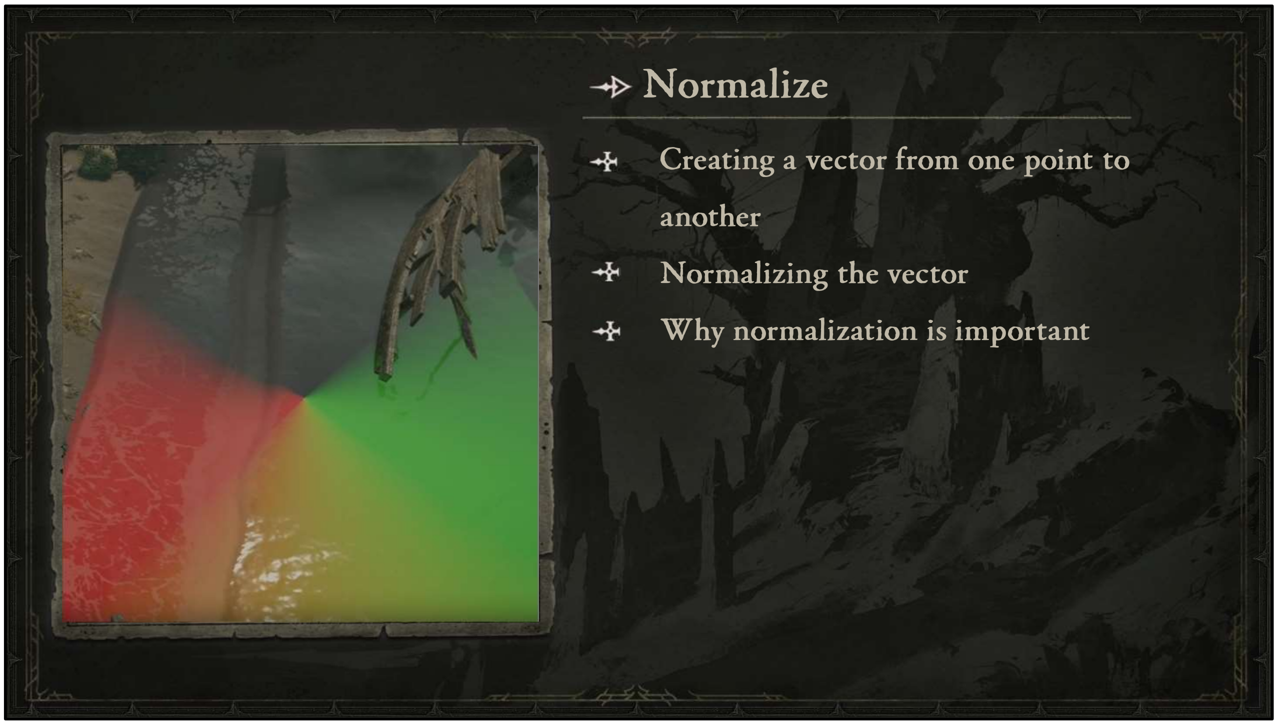


As you saw previously, flow is very important for defining the direction to the shore in order to help move foam textures in a natural way.
Procedural flow allows for a large area to have a core direction defined in the weather.
You can see here the visualization of single point defined in world space and a vector towards that point from each pixel on the water.
This procedural flow is designed to be additive, allowing for the option to paint more refined flowmaps on top.



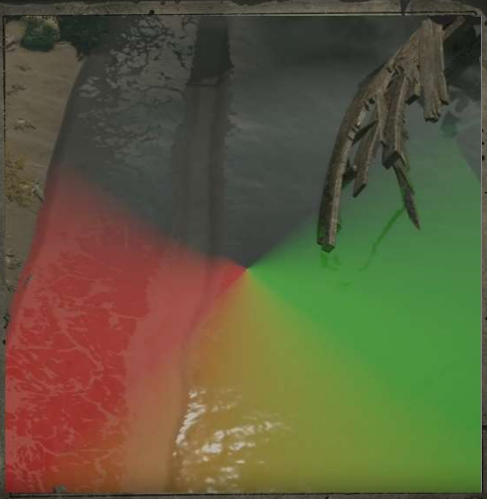
Flow Showcase

If we turn the flow off again in this area, we can see a reinforcement of what we saw earlier where flow is essential for the overall look of the water.
With the procedural flow back on, the foam again feels much more realistic.

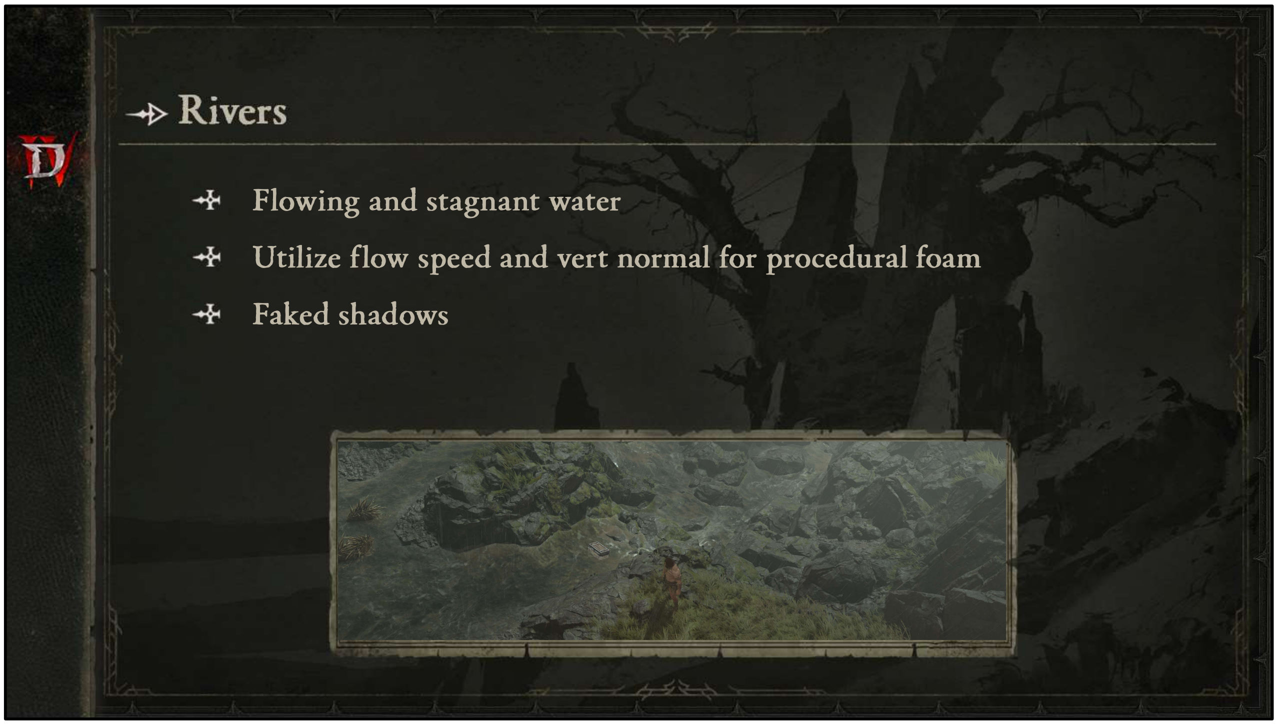


Normalize

- Creating a vector from one point to another
- Normalizing the vector
- Why normalization is important

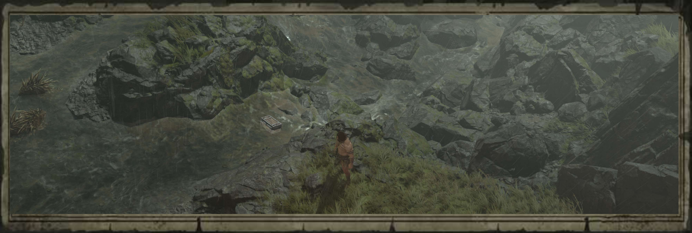


So let's look at how this procedural flow can be achieved.
This brings us to another fundamental shader function, which is Normalize.
We'll be dipping into some basic vector math with this one...
If we imagine these points A and B at arbitrary 2d locations, we could describe the vector from A to B as AB.
We can get this vector by doing B, which is the terminal, minus A, which is the start.
One trick for remembering this is that "if you're going to travel on an airplane, you have to go to the terminal before you can start your trip"
This vector has length, but for the purposes of our procedural flow, all we want is direction. That's what a "normal" (or normalized vector) is, a vector with a length of one.
To get that normalized vector, we want to divide AB by the length of AB, which will give us the "unit vector", denoted by the little hat on the left, which is a vector with a length of one.
You don't have to remember the specifics here if this is new to you, but the main takeaway is that we now have a direction, which in the case of peridural flow could mean that B is this point in the middle of the image, while A is different for every pixel on the screen.



Rivers

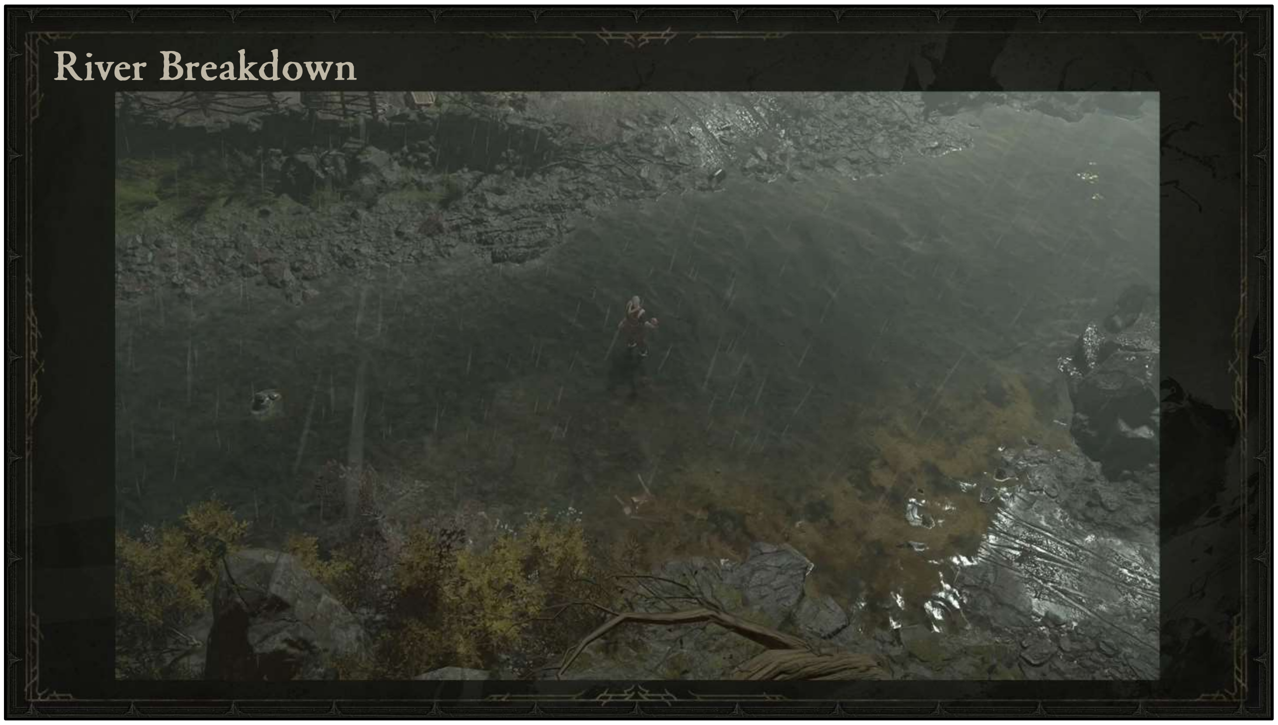
- Flowing and stagnant water
- Utilize flow speed and vert normal for procedural foam
- Faked shadows



Next, we'll talk about rivers.
Rivers vary from flowing to stagnant in different areas
We utilize flow speed and vert normal for procedural foam
And since we don't have lighting under Gbuffer water, we faked shadows for areas with stagnant foam layers.

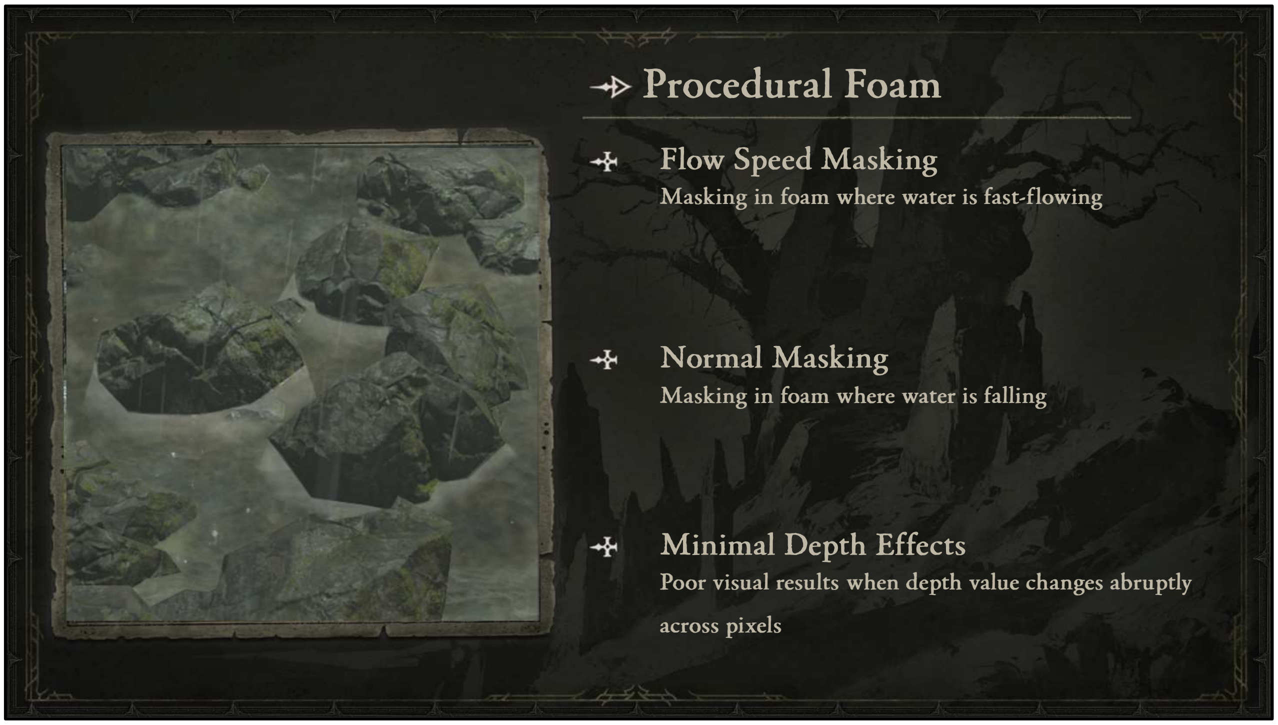
River Showcase

Here we can see a walkthrough of a river that has many elements combined, from stagnant foam to more turbulent foam and flow, as well as some vertex animation downstream.



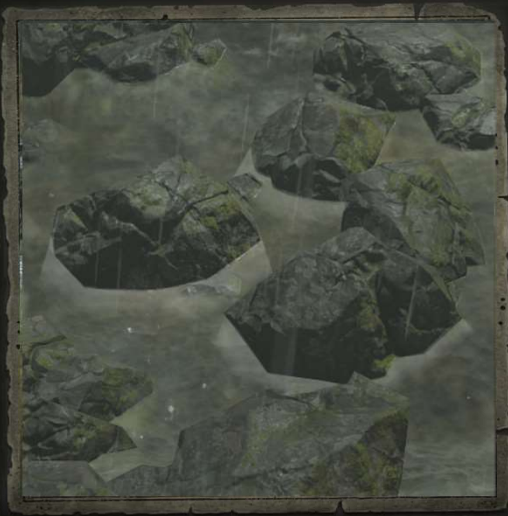
River Breakdown

Here we have a full breakdown of one of our river shaders.
First we’ll take a look at the normal.
Then if we turn the rain particles back on, we’ll see ripple normals being added in.
Next we’ll break down the albedo.
A core element as you might guess is the flow map, which we’re visualizing here.
We also have the option to blend some curl noise into the flow, which is made extreme here for visualization purposes.
If we combine those together, we get some nice versatility.
Here’s what the albedo might look like if we used only our “stagnant” foam layer.
And here’s our more turbulent foam layer.
Here is again with foam masking applied.
Next, we’ll take a look at the Refraction Buffer, which has some fairly subtle uv distortion, but we could amplify that a bit for emphasis.
If we add the foam back in. . .and maybe some caustics like we saw on the ocean. . .
We can then see the total effect including some rain here.

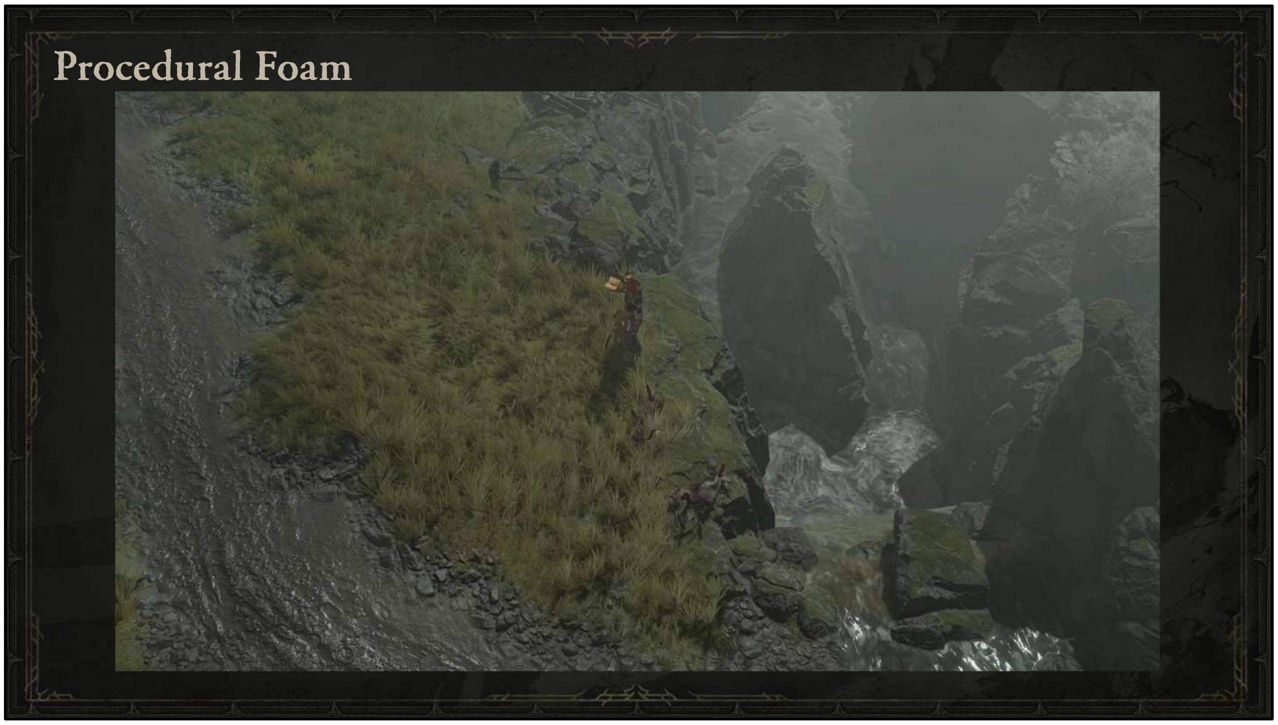


Procedural Foam

- **Flow Speed Masking**
  Masking in foam where water is fast-flowing
- **Normal Masking**
  Masking in foam where water is falling
- **Minimal Depth Effects**
  Poor visual results when depth value changes abruptly across pixels



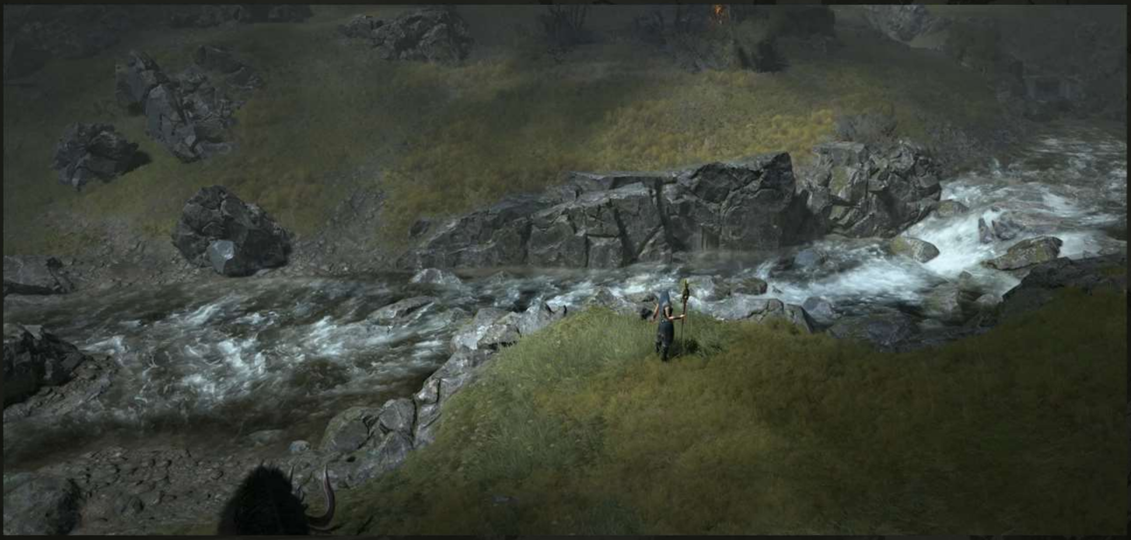
As you may have gathered, level artists can paint in foam where desired, but we also wanted to do some procedural foam to make flowing rivers look good right out of the box.
We do this primarily through two methods.
Flow Speed Masking is where we can take the flowmap information and smoothstep it to mask in foam where the flow vectors are the longest, meaning they are moving the fastest.
We also do normal masking meaning that we bring in foam wherever the normal of the water heightfield is pointed away from up, indicating that water should be falling.
And while we do modify color a bit using scene depth when sampling the refraction buffer, we don't do any depth-based foam because that can easily lead to poor visual results when depth values change abruptly across pixels. I've emphasized that using photoshop in this image and I'm sure you've seen similar results in various games. This is something we chose not to do for our procedural foam.



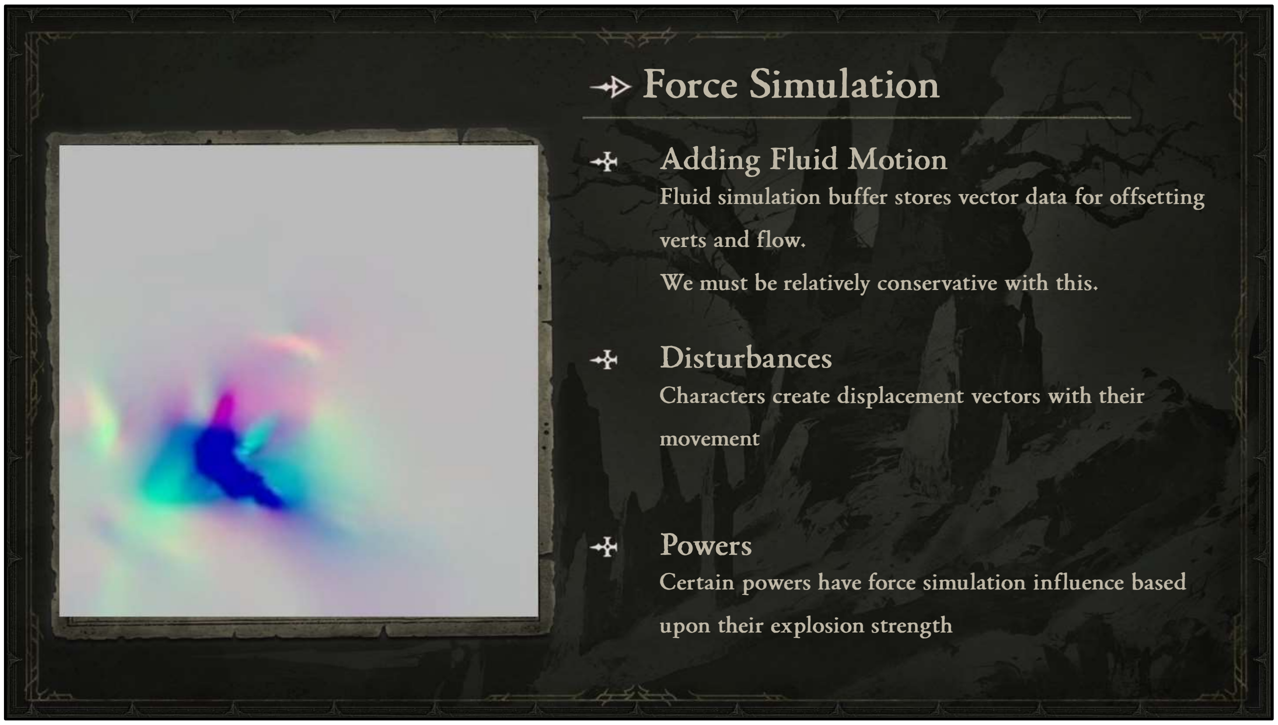
Procedural Foam

If we look at this small waterfall where I’ve disabled particles, we can focus on just the shader.
He we could visualize the flow speed mask which will be white wherever the flow is fast enough as defined by the material parameters
And here we can see a mask for the verts pointed away from world up.
With those two combined, we see how even without manually painting foam layers, we get some nice procedural foam.

River Foam Showcase



Now, during actual gameplay, we have to balance the values in the playable space so that we don’t distract from gameplay, but here you can see a more cinematic demonstration of procedural and hand-painted foam used together.



Force Simulation

Adding Fluid Motion
Fluid simulation buffer stores vector data for offsetting verts and flow.
We must be relatively conservative with this.

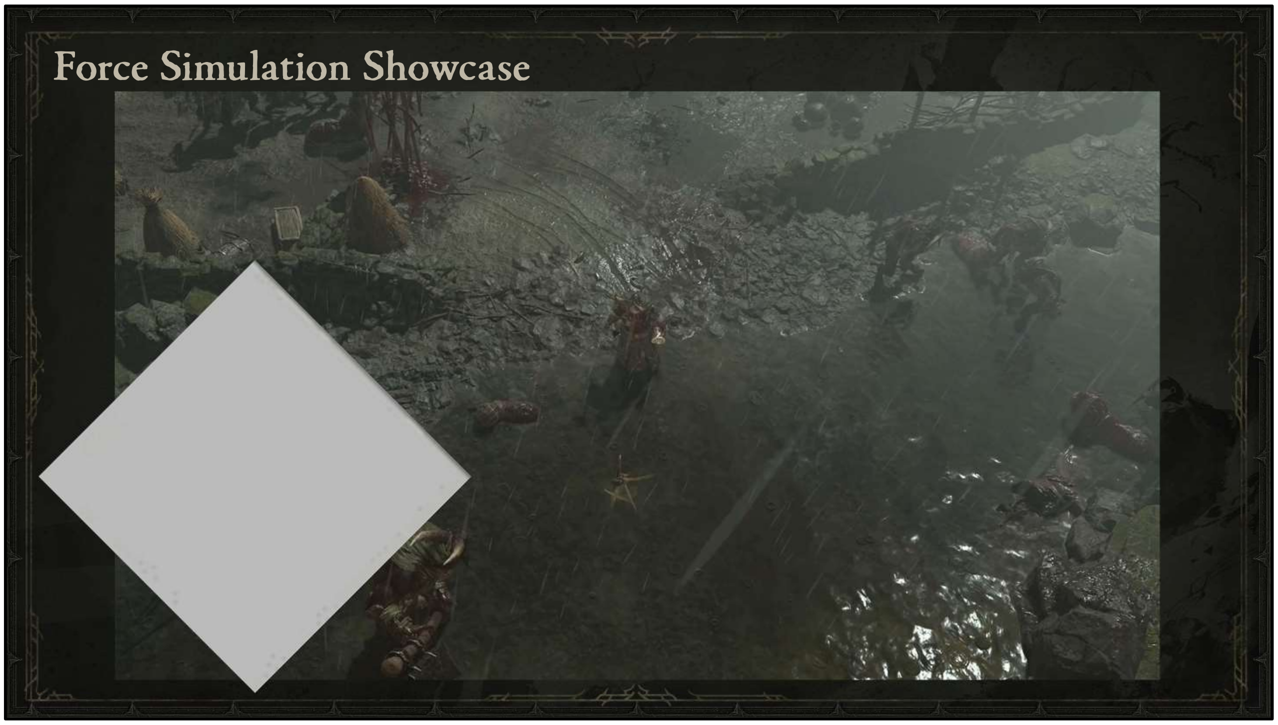
Disturbances
Characters create displacement vectors with their movement

Powers
Certain powers have force simulation influence based upon their explosion strength

Let’s talk about simulation.
Our game uses what we call a Force Simulation buffer (or “ForceSim” for short).
We use this to add fluid motion by storing vector data for offsetting verts and flow. It’s comparable to a flow map in that it stores vector information, but the unique feature is that it simulates fluid movement at runtime. It’s a fairly low resolution buffer and you can see a preview of it here.
We have to be relatively conservative with this, though, and you’ll see why in a moment.

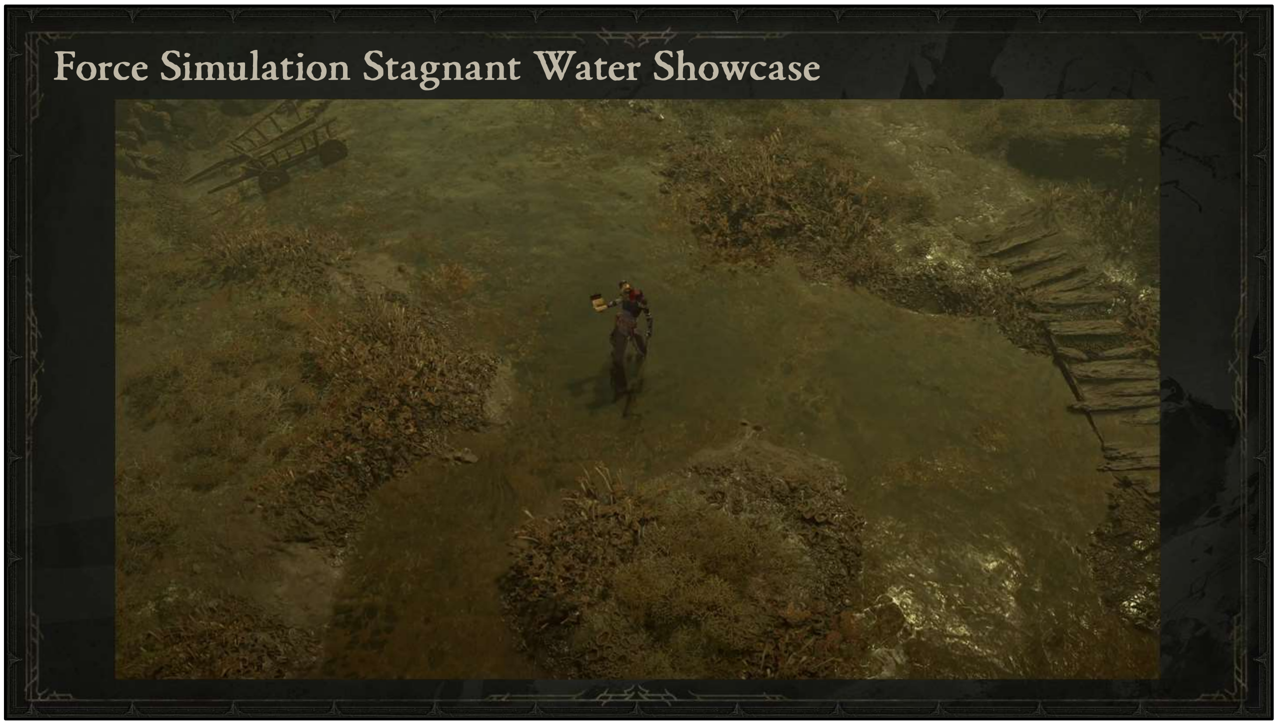
There are two main contributors to the force sim.
Disturbances are where characters create displacement vectors when they move

Certain powers have force simulation influence based upon their “explosion” strength



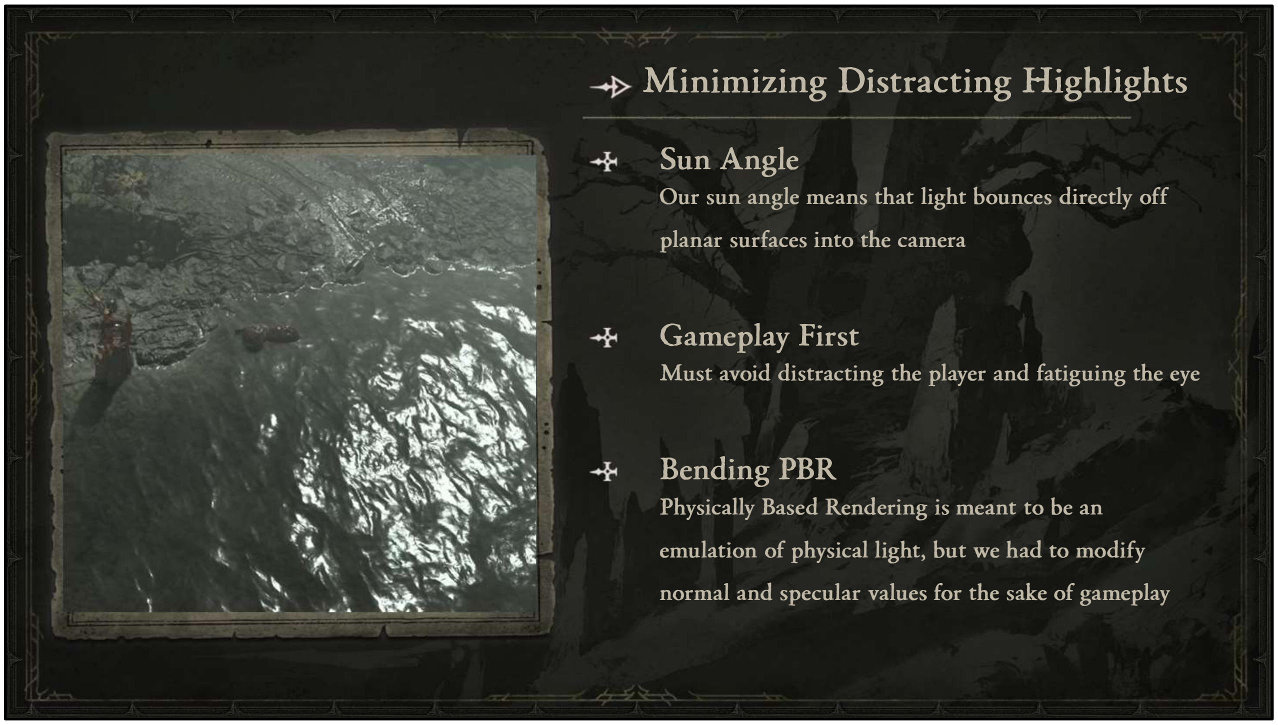
Force Simulation Showcase

Here we can see a druid using pulverize.
You'll notice that even with the particles disabled, the effect displacing the verts and adding in foam is fairly subtle.
Let's look at this again with the force sim debug enabled in the upper left.
By default the debug is fairly small so I'm going to scale it up and rotate it so it's easy to understand what you're looking at relative to the camera angle.
If we turn the monster AI on, you'll notice that of all of the character movement and attacks quickly fill the force sim with vector information, which is why the overall effect on the water must be kept fairly conservative.

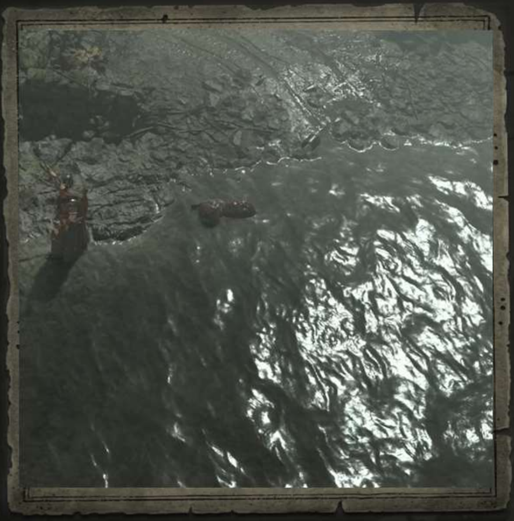


Force Simulation Stagnant Water Showcase

We can see the same force simulation being used in the more murky swamps of Hawezar to cut away at stagnant foam. As you might expect the foam gradually returns as the vectors in the force sim return to normal.



Minimizing Distracting Highlights



Sun Angle
Our sun angle means that light bounces directly off planar surfaces into the camera

Gameplay First
Must avoid distracting the player and fatiguing the eye

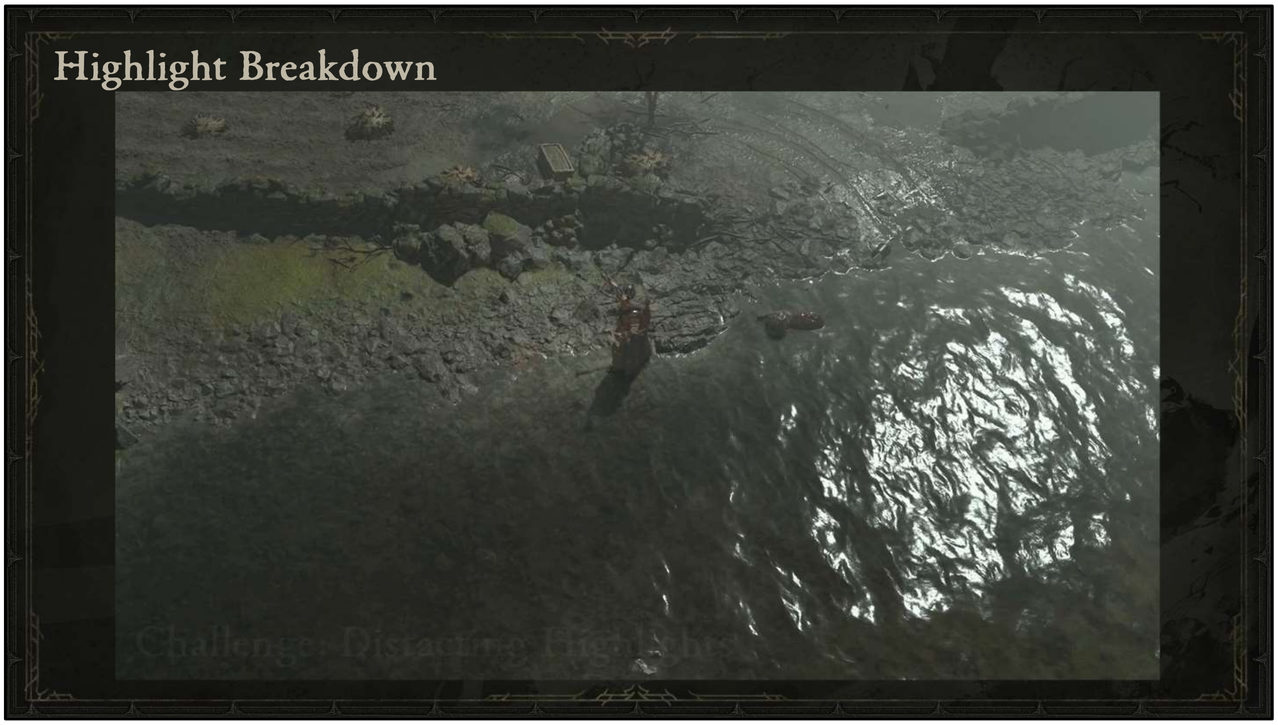
Bending PBR
Physically Based Rendering is meant to be an emulation of physical light, but we had to modify normal and specular values for the sake of gameplay

Next we'll talk about a very important approach we took to minimizing distracting highlights.

Our sun angle being behind the player relative to the camera means that light bounces directly off planar surface into the camera, creating a large highlight in the playable space.

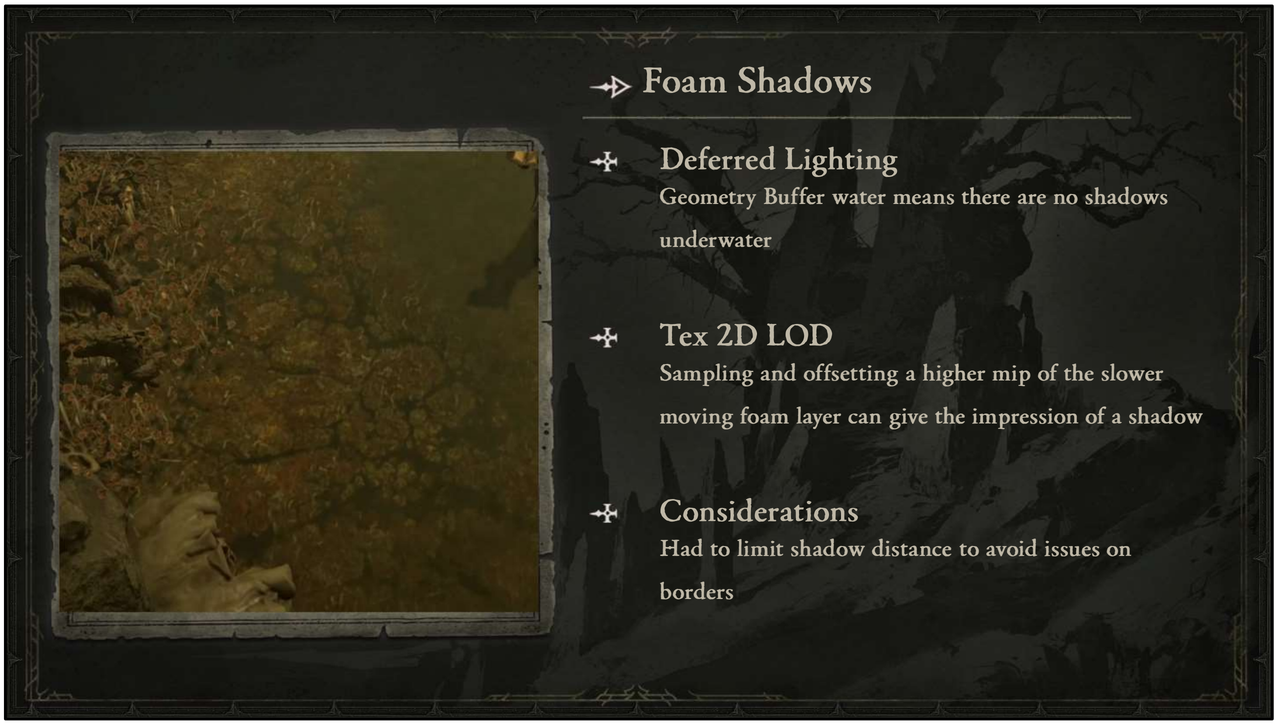
In order to put Gameplay First, we must avoid this potentially very distracting highlight that could be fatiguing to the player's eye

For us, this meant "bending" PBR. Physically Based Rendering is meant to be an emulation of physical light, but we had to modify normal and specular values for the sake of gameplay.

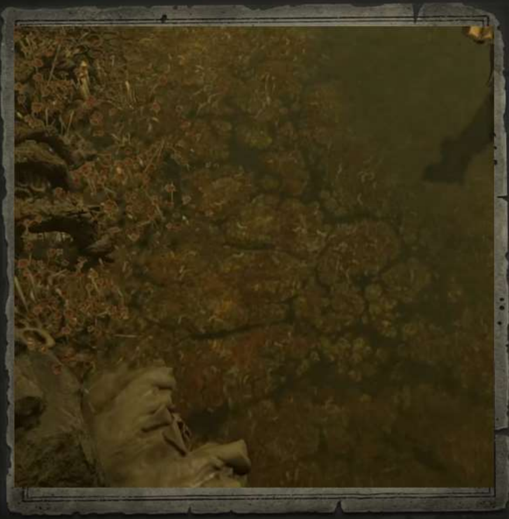


Highlight Breakdown

Here we can see what the water highlight would look like without any modifications. It takes up a large portion of the right-center of the screen and is very bright.
So the first thing we'll do is reduce the specular reflectance to lower the amount of light reflected off the water.
And then we will rotate the viewspace normal of the water to push the highlight from the center to the lower right portion of the screen, thereby maintaining an important specular hit, but moving it out of the primary gameplay space.



Foam Shadows



Deferred Lighting
Geometry Buffer water means there are no shadows underwater

Tex 2D LOD
Sampling and offsetting a higher mip of the slower moving foam layer can give the impression of a shadow

Considerations
Had to limit shadow distance to avoid issues on borders

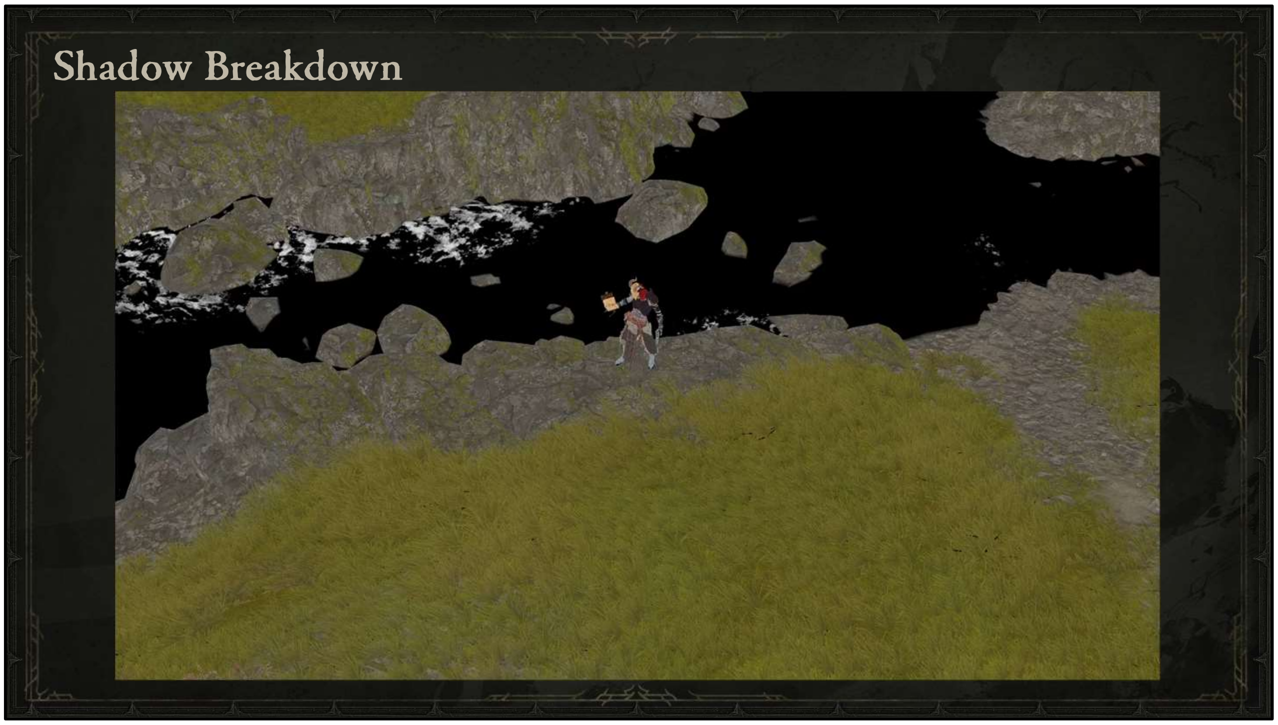
Let’s talk about Foam Shadows.
Now as I’m sure you’ve gathered, with gbuffer water, we don’t get shadows under the water.

In comes, Tex 2D LOD.
With Tex 2D LOD, we can sample and offset a higher (meaning “blurrier”) mip of the slow moving foam layer alpha to give the impression of a shadow.

We did however have to limit shadow distance in order to avoid issues on scene borders

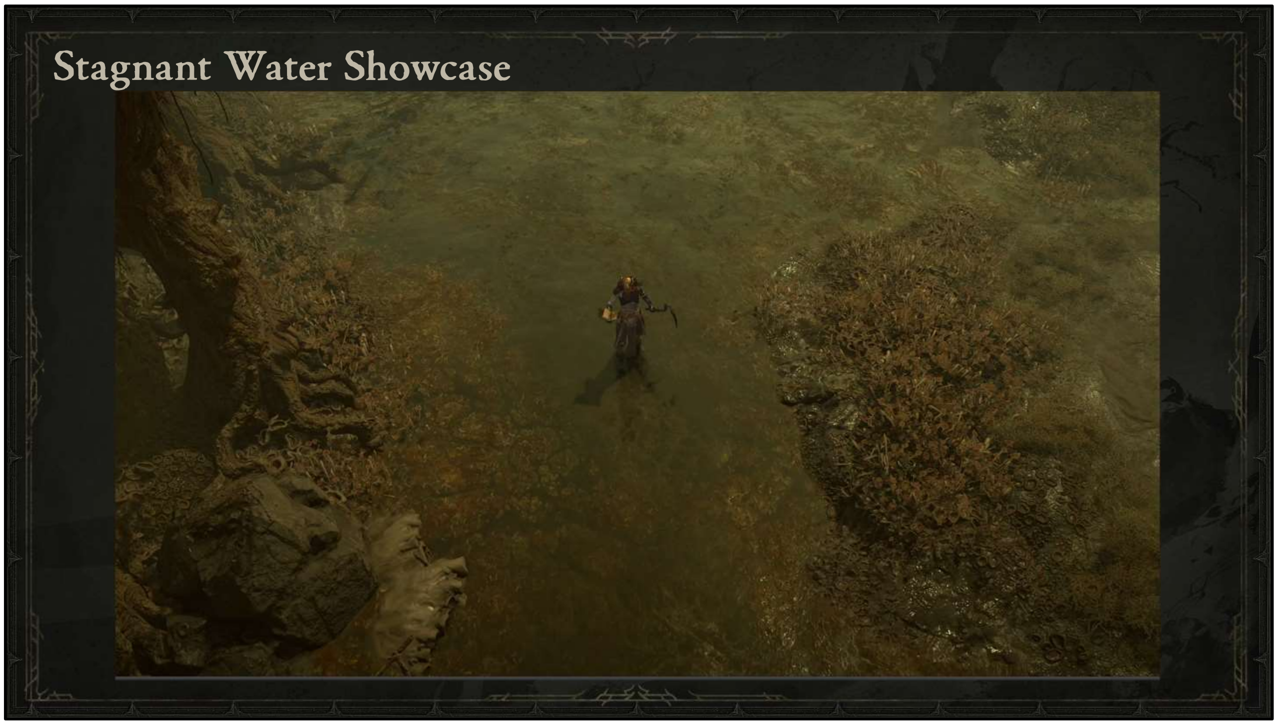
Shadow Comparison

Here you can see a fairly subtle example of this in the river we saw earlier.
In the upper left there’s a slow-moving algae layer with foam shadows toggling on and off.



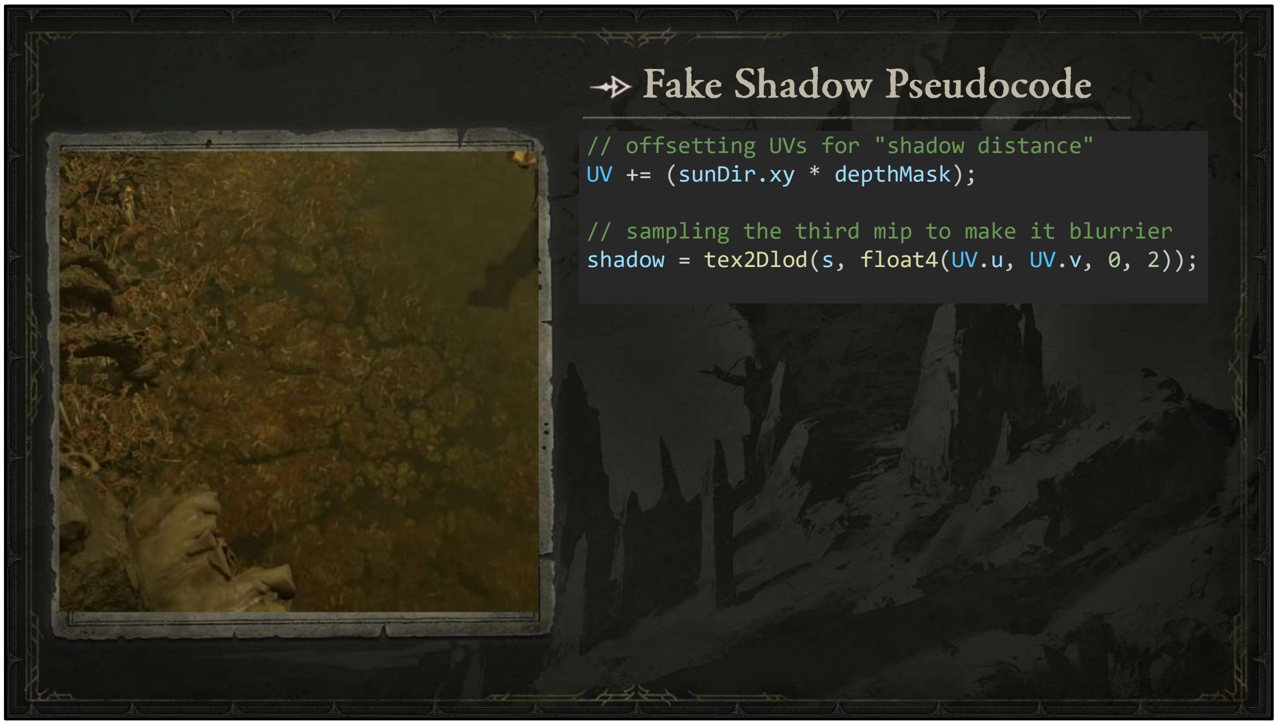
Shadow Breakdown

Here, you can see a black and white mask of where that shadow comes in.



Stagnant Water Showcase

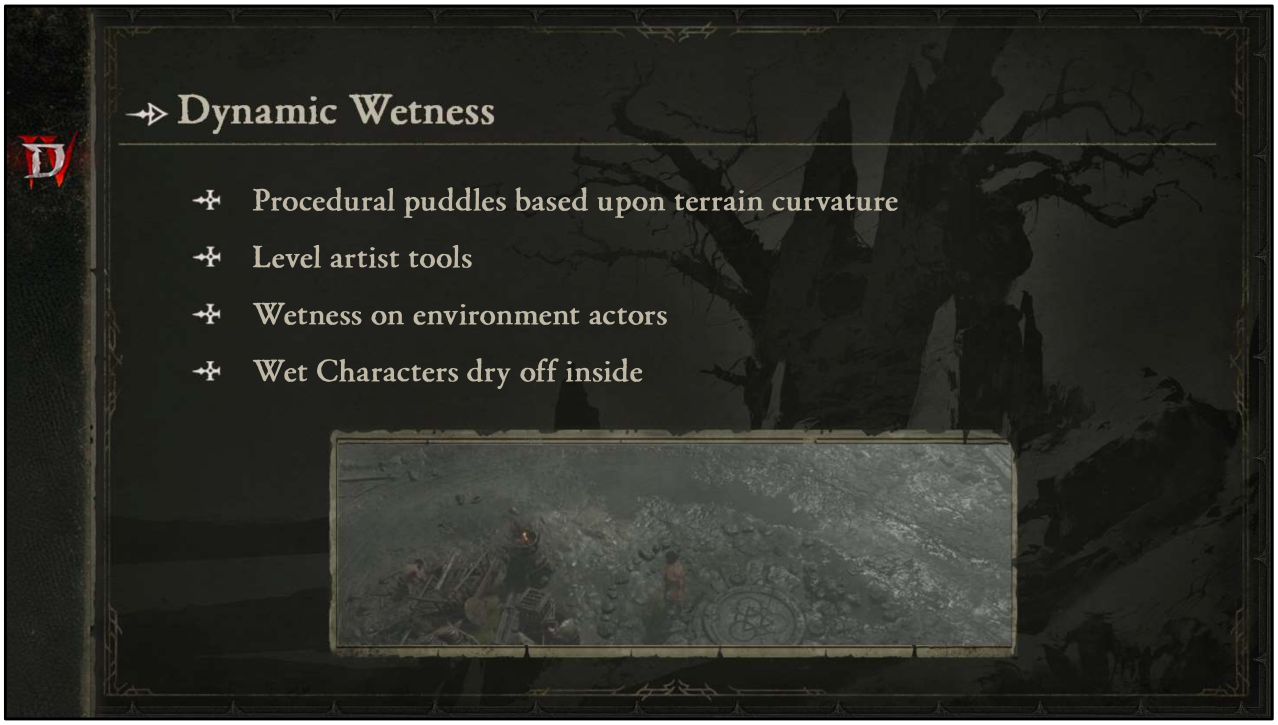
But the effect is more prevalent here in the swamps of Hawezar again where we can see the shadows toggling on the algae on the left.



Fake Shadow Pseudocode

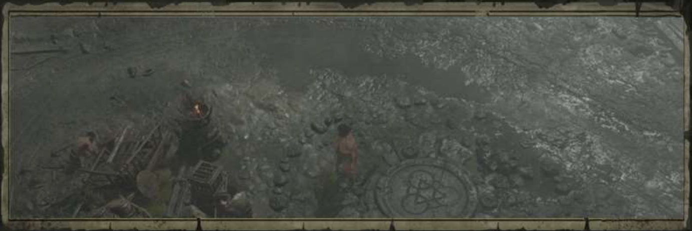
```
// offsetting UVs for "shadow distance"
UV += (sunDir.xy * depthMask);

// sampling the third mip to make it blurrier
shadow = tex2Dlod(s, float4(UV.u, UV.v, 0, 2));
```

Now if we look at some pseudocode for how this is generally achieved,
We want to start with offsetting our UVs based upon the sun direction in x and y, and multiplying that offset by the scene depth comparison with the terrain below, so that the deeper the water, the greater the offset, up to a limit.
Then we'll use Tex 2D LOD to sample the texture at the third mip, that's number 2 because it's zero-based, which will result in a softer mask for the shadows.



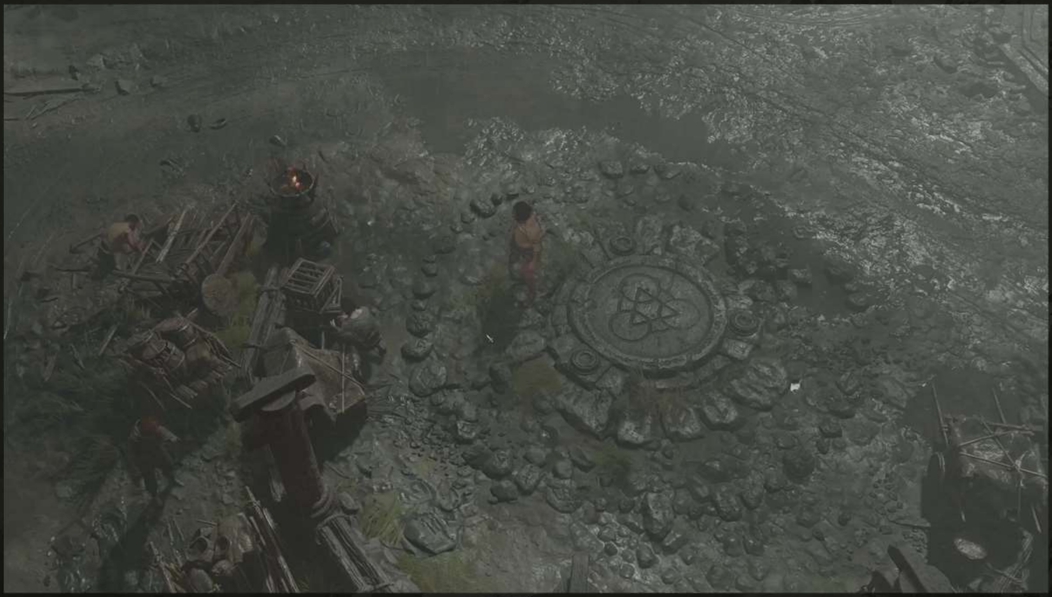
Dynamic Wetness

- Procedural puddles based upon terrain curvature
- Level artist tools
- Wetness on environment actors
- Wet Characters dry off inside



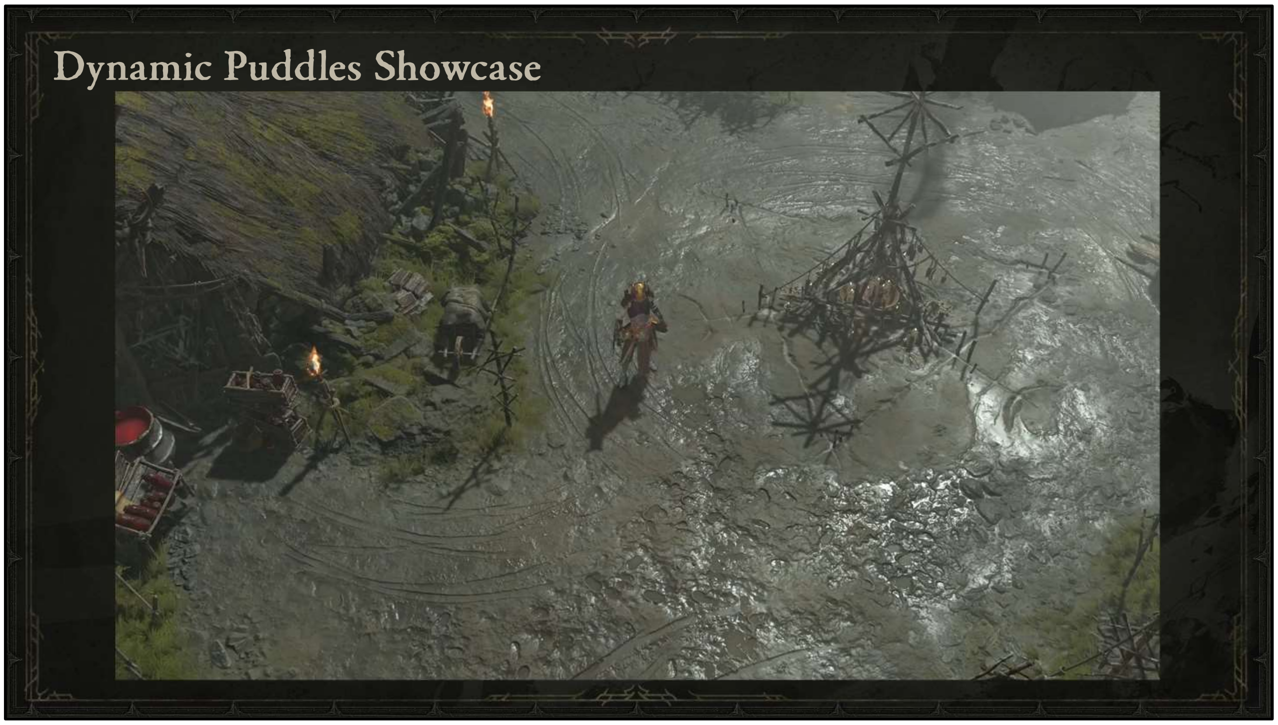
Let’s talk about Dynamic Wetness.
Procedural puddles accumulate in the terrain based upon curvature
Level artists have tools for masking out these puddles.
Environment actors also get wet when it rains, modifying their albedo and roughness values.
And Characters get wet as well, but as a nice touch, they dry off inside buildings, caves, and the like.

Dynamic Wetness Showcase

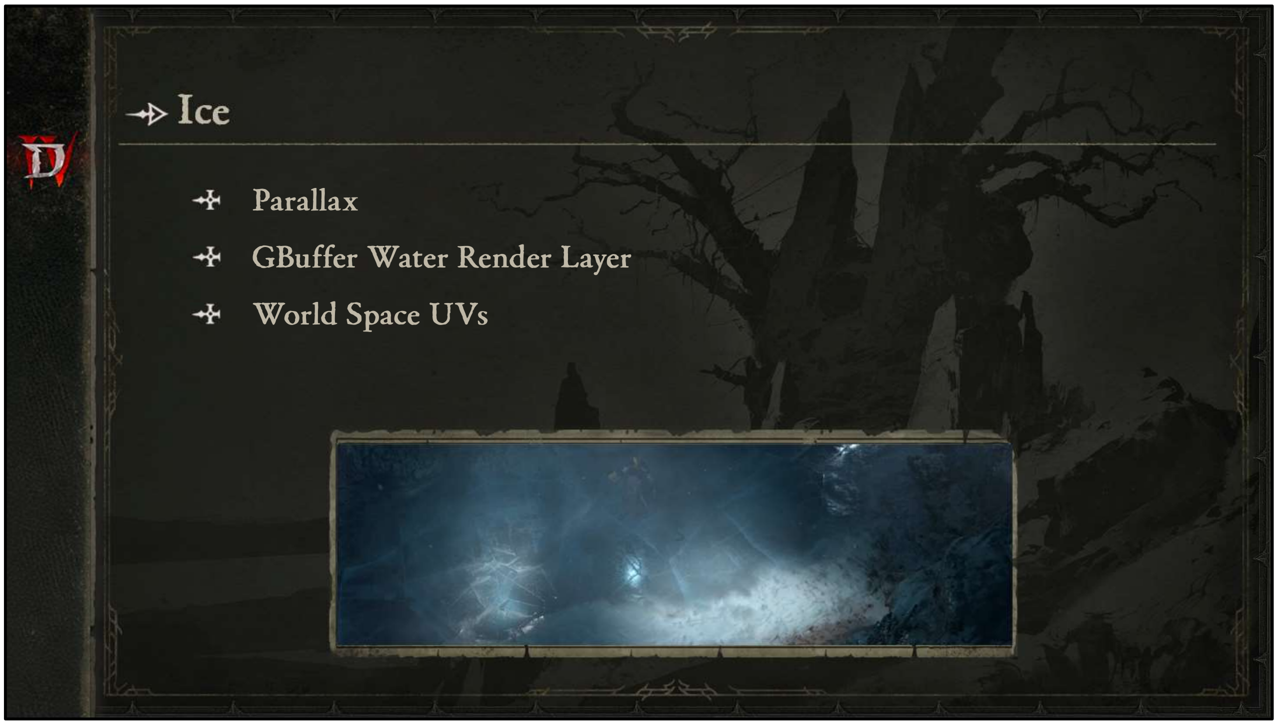


Here's a quick run-through of a town in Scosglen where you can see a lot of dynamic wetness, especially in the environment and in the terrain puddles.
Environment actors have darker albedo to emulate water on their surface causing less light to be reflected and they also become less rough, emulating that smooth water collecting on the surface making the highlight smaller.
These changes are modified based upon material parameters for how porous the surface is intended to be, as well as a world space noise mask to add some variety.

Dynamic Puddles Showcase



Here’s a demonstration of wetness and puddles accumulating over time, where I’ve markedly increased the rate of accumulation.
Some terrain materials are authored to be wet all the time, but even with these, the wetness is amplified and you may also notice ripples from the rain showing up in the footprints in the lower right.
For puddles, the curvature data is limited to the vert density of the heightfield, so I had to get creative with how we add noise edges of the mask to break up angular shapes.
You’ll also see here how puddles are first represented by the darkening of albedo, followed by the roughness getting lower.
To make these puddles look like water, the normal is pointing in world up, so in order to minimize mirror-like reflections of the sun, there’s always a small amount of noise added to the normal to break up the highlight.

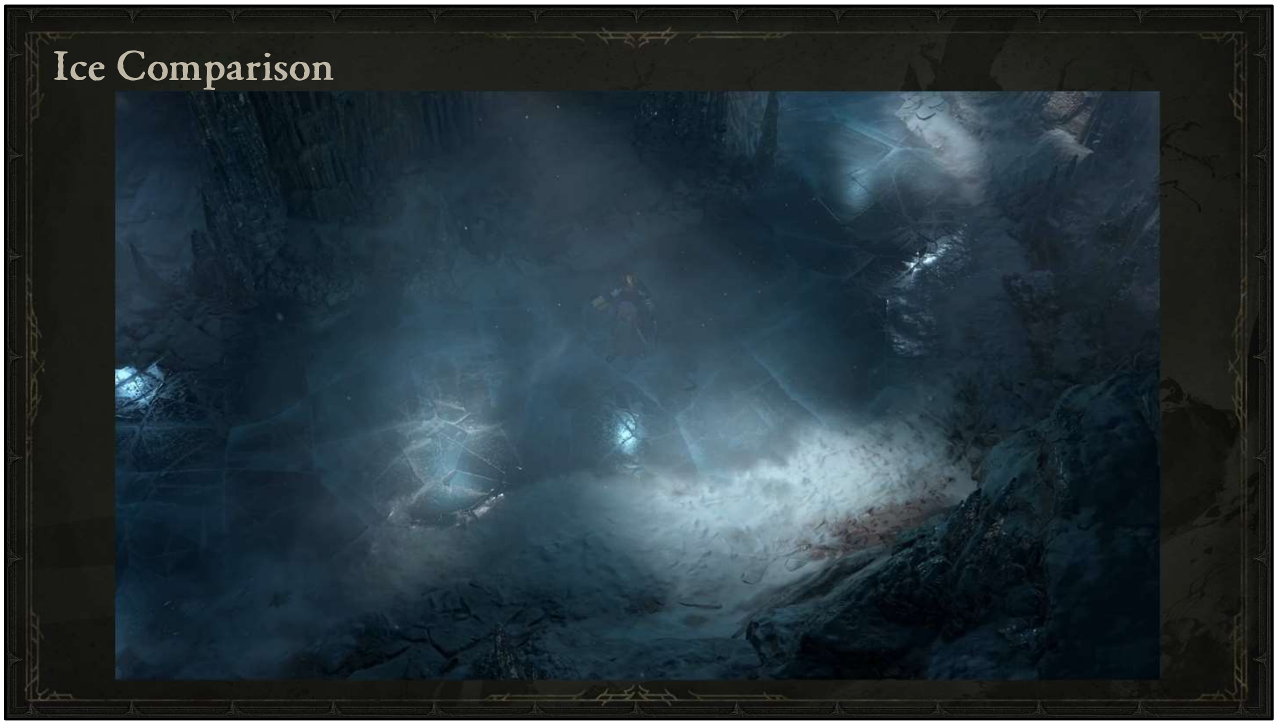


Ice

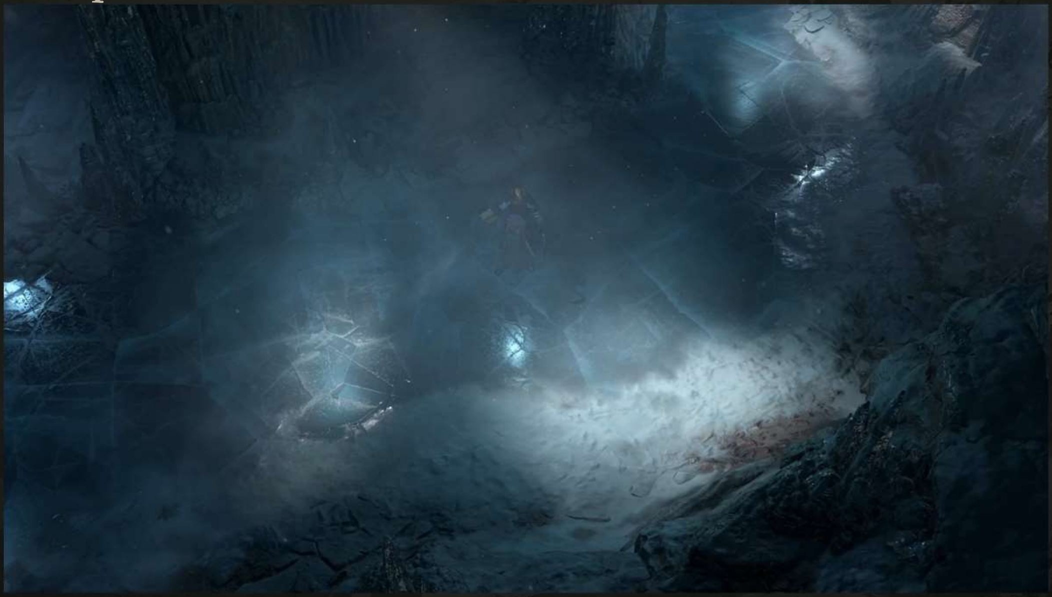
- Parallax
- GBuffer Water Render Layer
- World Space UVs

Now let's talk about ice.

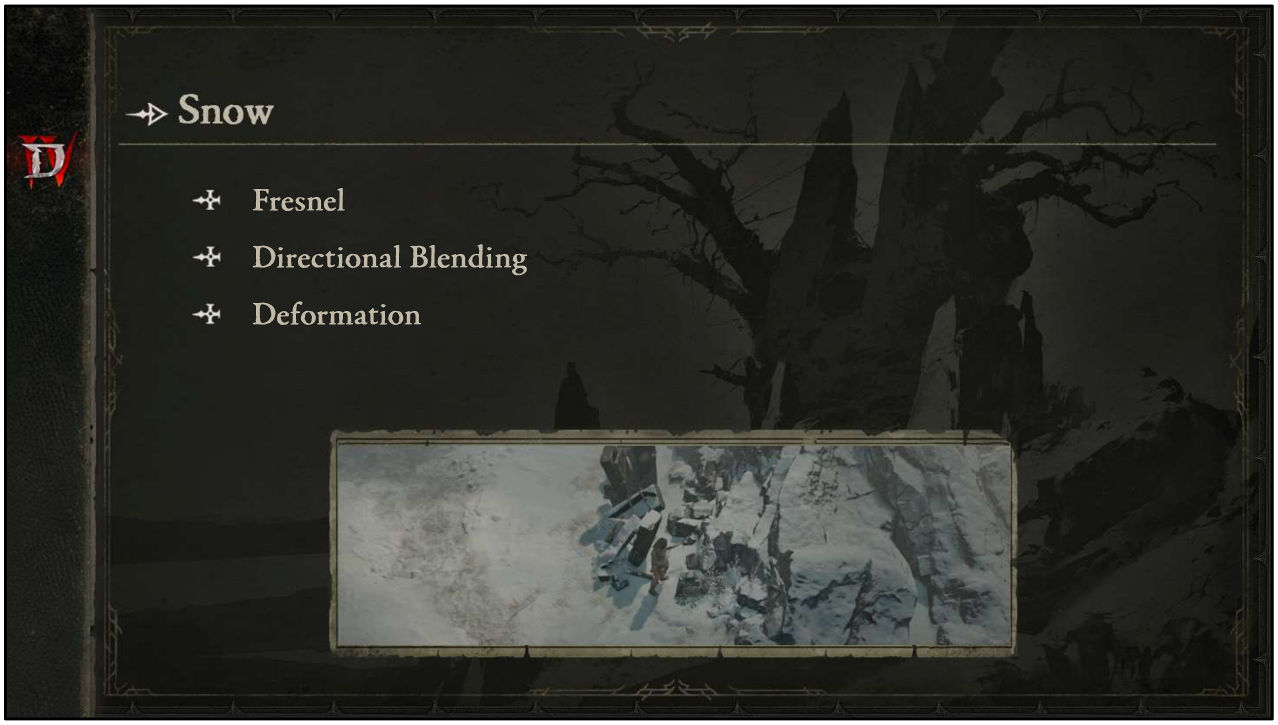
For ice, we mainly lean on parallax, as you might expect, where we offset UVs based upon the camera angle relative to the surface normal.
Our Ice is also in the Gbuffer Water Render Layer, meaning it relies upon the same access to the refraction buffer for sampling the albedo underneath it.
And we frequently use World Space UVs with ice.



Ice Comparison



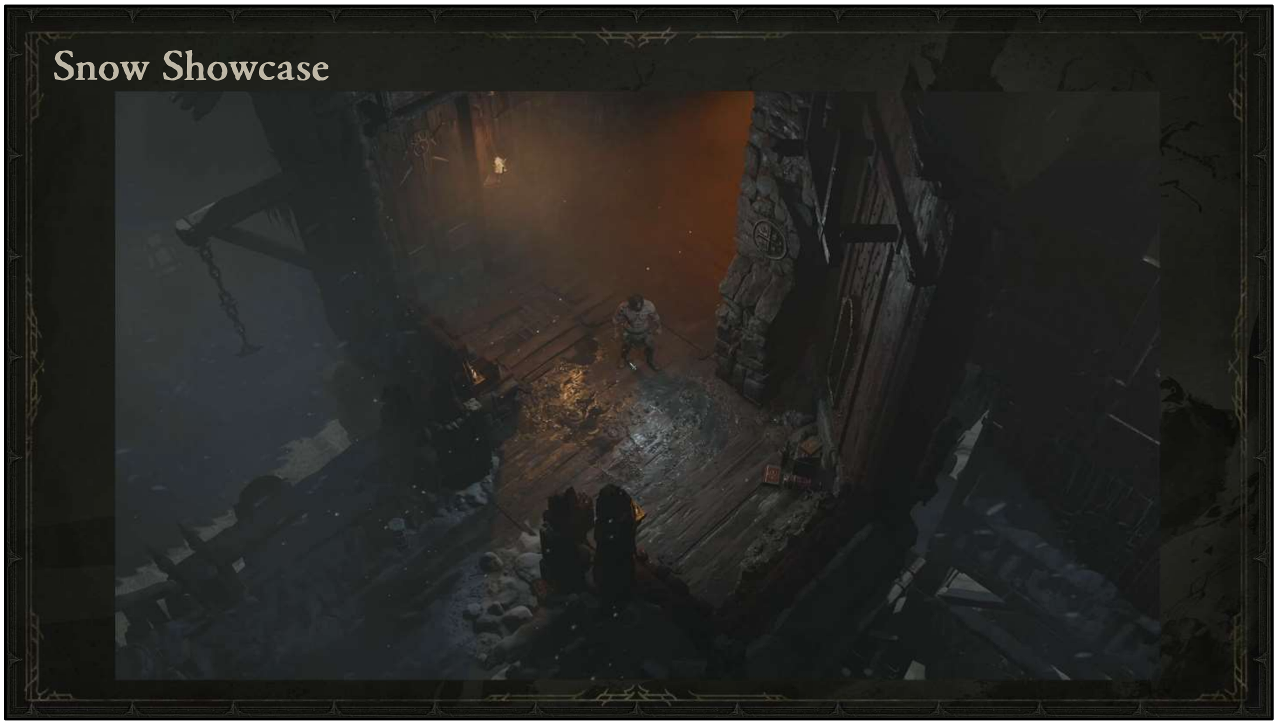
Let’s take a look at this dungeon with an ice plane underneath the player.
First we’ll walk back and forth with regular lighting so you can see the in-game effect.
Next, we’ll look at just the albedo. You may notice how the cracks underneath the surface are offset to sell the idea that there’s actual volume under this plane.
Now we’ll take a look at the effect again while toggling it on and off, first with regular lighting…
And again with only the albedo. All of this is achieved mainly through world space UVs and Parallax UV distortion.



Snow

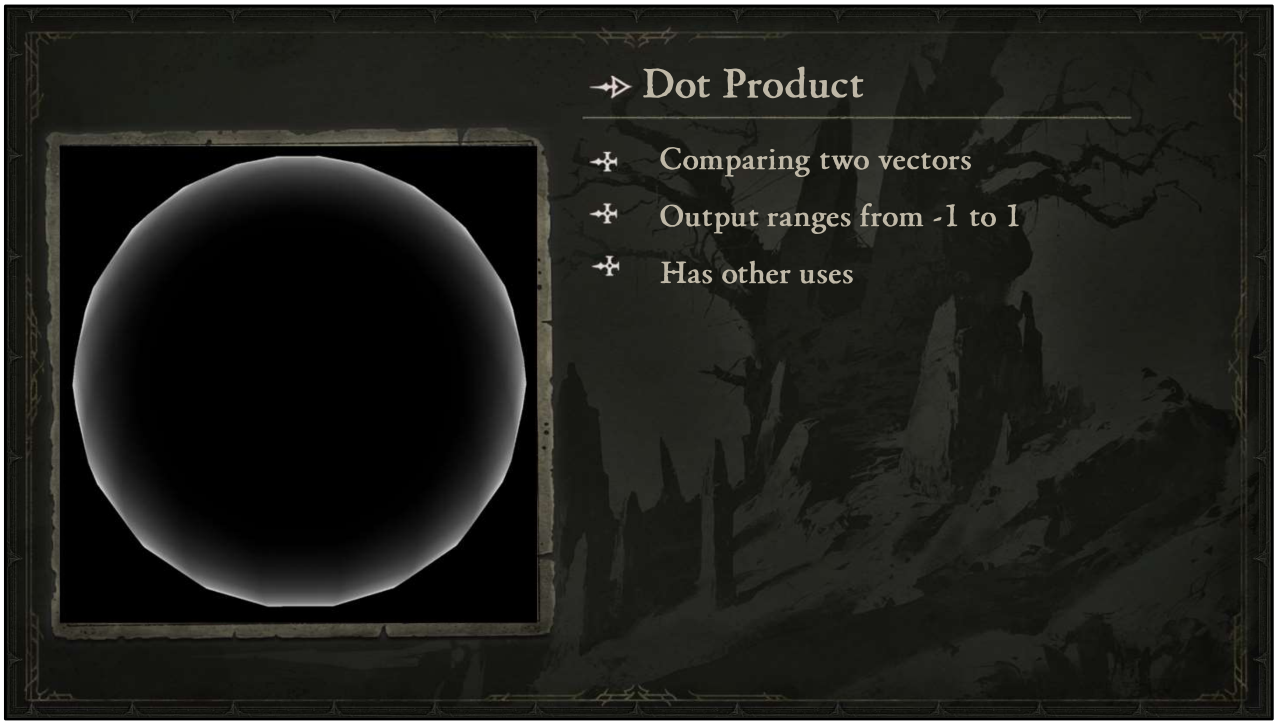
- Fresnel
- Directional Blending
- Deformation

Now let's talk about Snow.
Snow shaders tend to use Fresnel effects where the edges are brighter than the center to convey a sense of light bouncing off of the glancing angles.
The other primary effect you'll see with snow is directional blending, namely for snow-topped surfaces.
And our terrain does dynamic deformation in certain areas.



Snow Showcase

First, we'll run briefly through this dungeon.
It worth noting that even in dramatic lighting conditions, effects like Fresnel that make the edges look brighter can help add to the realism of the scene.
That's most apparent here on the snow-topped railings.

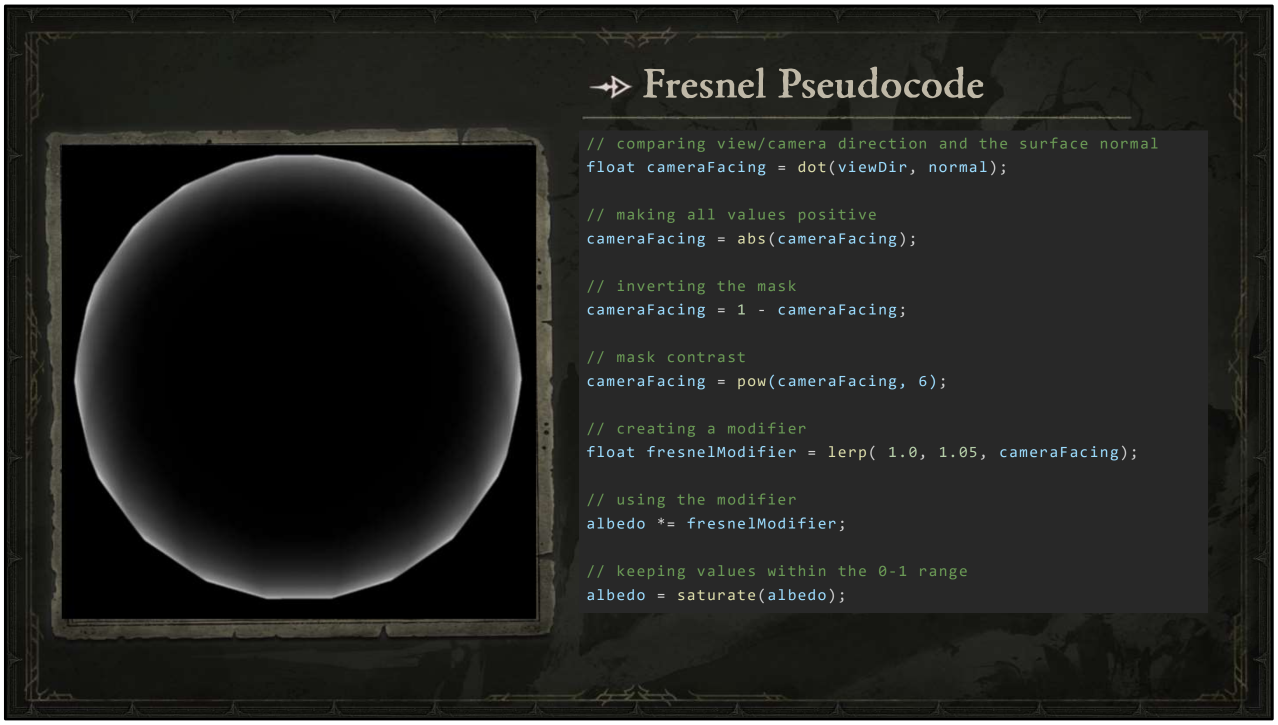


Dot Product

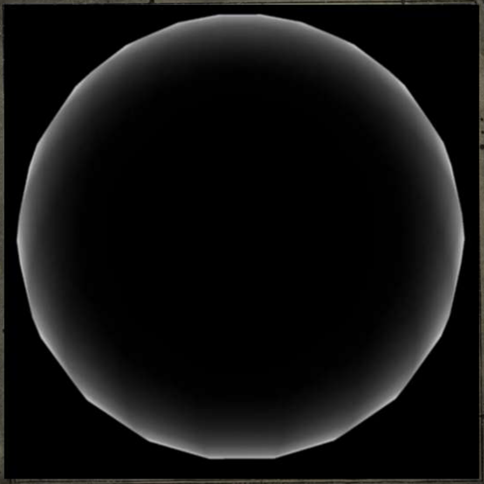
- Comparing two vectors
- Output ranges from -1 to 1
- Has other uses

Let’s talk about how the Fresnel is achieved with our last shader function fundamental, the dot product.

The dot product is used to compare two vectors.
It outputs a range from -1, when the vectors are pointed opposite directions, to 1, when the vectors are pointing the same direction
It also has some other interesting uses when used algebraically, like for instance, it can be used to create grayscale from a color image.

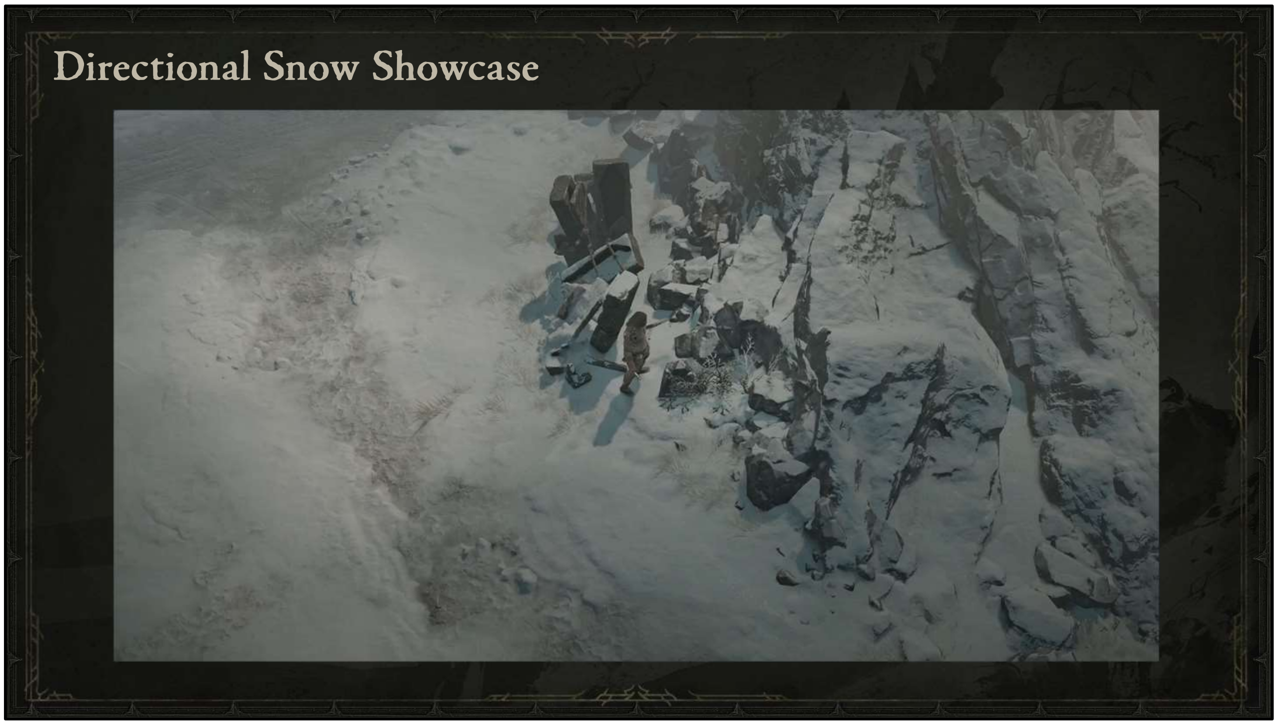


Fresnel Pseudocode



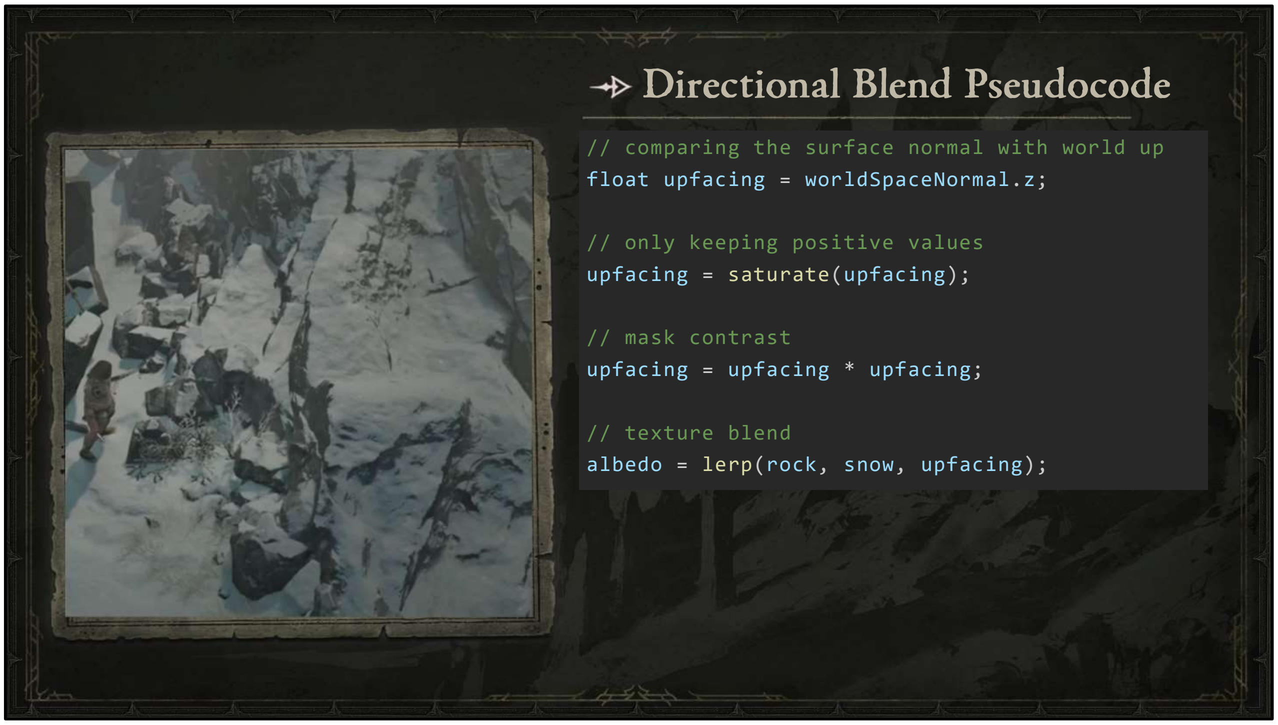
```
// comparing view/camera direction and the surface normal
float cameraFacing = dot(viewDir, normal);

// making all values positive
cameraFacing = abs(cameraFacing);

// inverting the mask
cameraFacing = 1 - cameraFacing;

// mask contrast
cameraFacing = pow(cameraFacing, 6);

// creating a modifier
float fresnelModifier = lerp( 1.0, 1.05, cameraFacing);

// using the modifier
albedo *= fresnelModifier;

// keeping values within the 0-1 range
albedo = saturate(albedo);
```

For creating the Fresnel effect, we want to compare the view direction (which is where your camera is pointing, and the surface normal. So to begin, we're looking at a visualization of the normal.
We will take the dot product of the viewDir and normal, like I mentioned.
Which can be a negative result so for our purposes, we will take the absolute value.
Then we'll want to invert it so that we can make modification to the edges rather than the center.
There are a number of ways to increase contrast including smoothstep, but for this example, I'm simply doing a power with an exponent of 6.
From there, you have your mask and it really comes down to however you want to use it.
For this pseudocode example I've just lerped between 1 and 1.05 and multiplied my albedo by that value, making sure the saturate the result so that we clamp values to the 0-1 range.

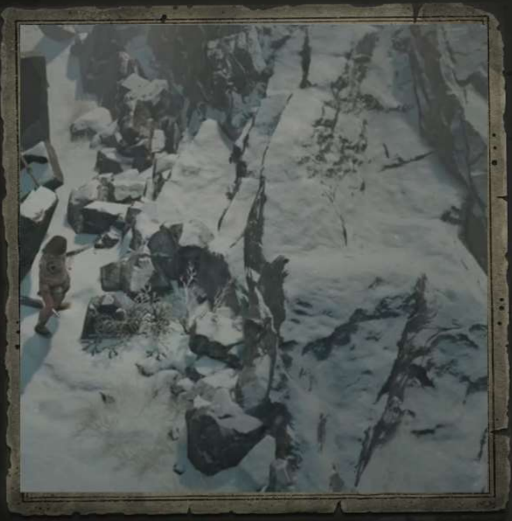


Directional Snow Showcase

Snow-topped surfaces are seen throughout our Fractured Peaks region.
Here's you can see our snow terrain blending nicely with snow-topped rocks that can be rotated in any direction and will always have snow on top.



Directional Blend Pseudocode

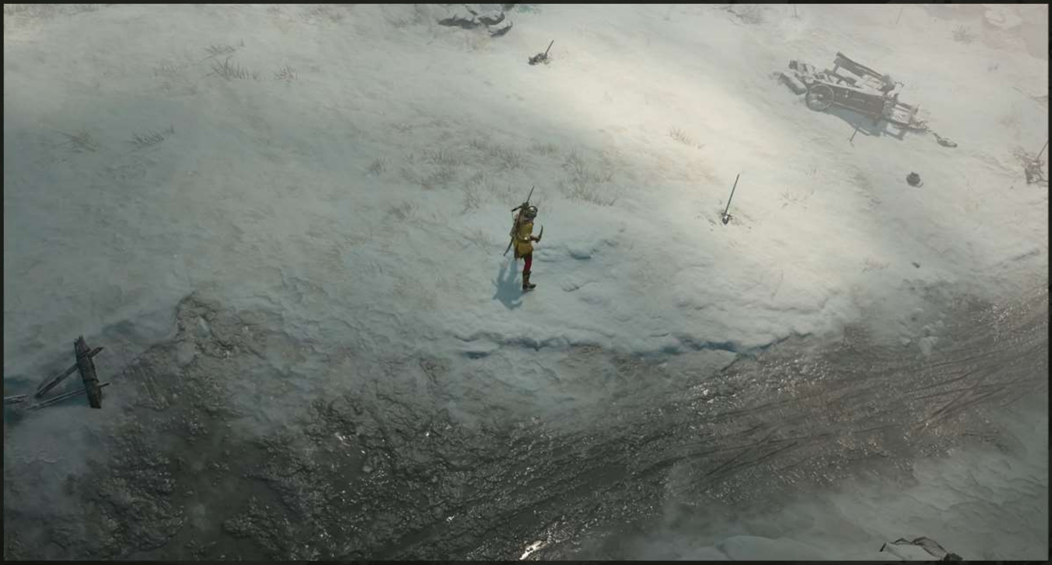


```
// comparing the surface normal with world up
float upfacing = worldSpaceNormal.z;

// only keeping positive values
upfacing = saturate(upfacing);

// mask contrast
upfacing = upfacing * upfacing;

// texture blend
albedo = lerp(rock, snow, upfacing);
```

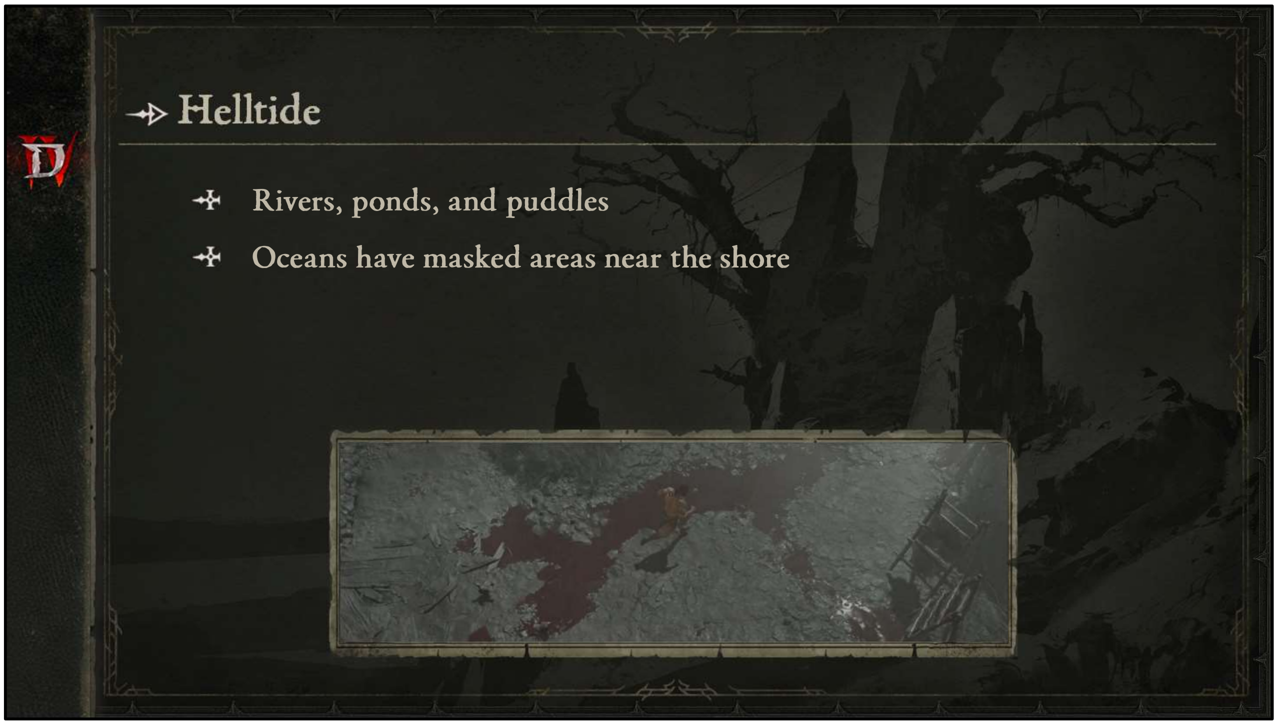
To achieve this, you can actually avoid using a dot product, and simply use your world up as a mask, which is Z in our case.

We want to only keep the positive component, so I'll use saturate to clamp it to 0-1. Then I'll demonstrate a cheaper means of creating contrast by simply multiplying the mask by itself, which is a more efficient way of doing a power with an exponent of 2. You can then do whatever you like with that mask. For this example, I'll blend between two imagine rock and snow textures using the upfacing mask with a lerp.

Snow Deformation Showcase



And here we can see an example of Terrain Deformation.
Our terrain allows for tessellation so we can get more triangles to work with for deformation, which occurs when characters pass through it or powers are used on top of it, similar to the force simulation we saw earlier.
Fun fact: This can be used on any terrain material depending upon the settings and terrain painting, so you may also notice it on mud and sand materials as well.

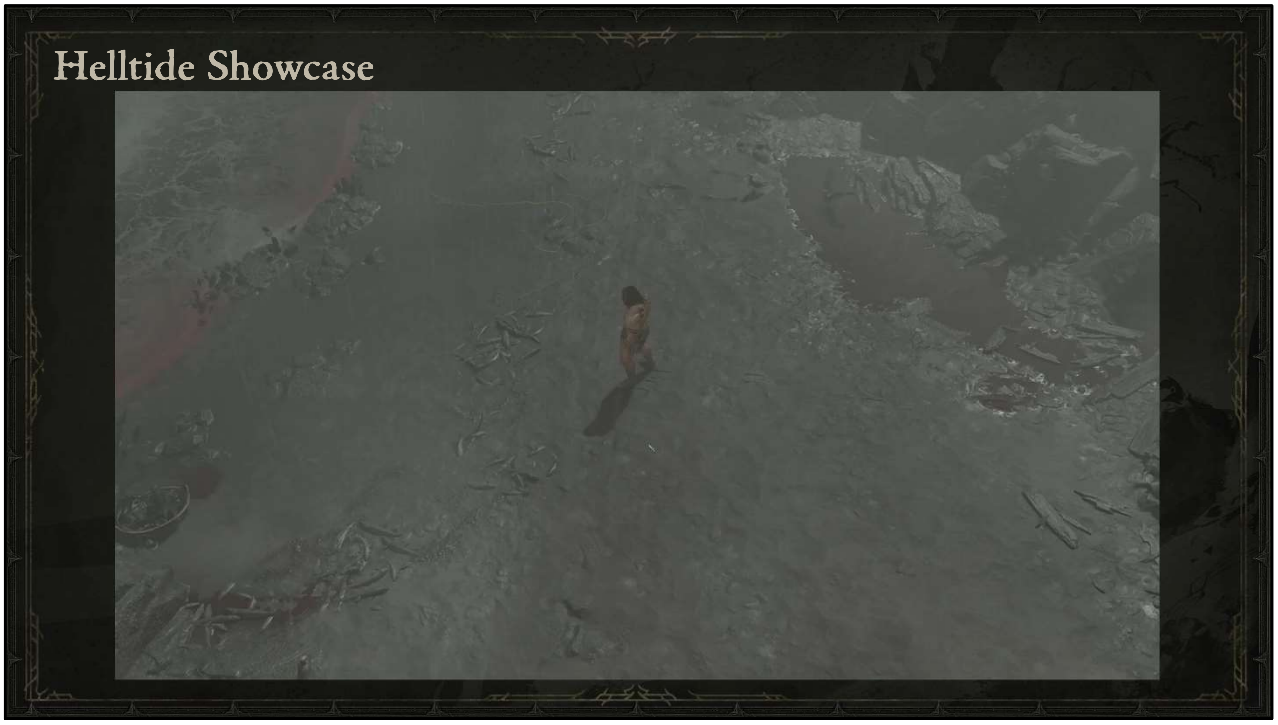


Helltide

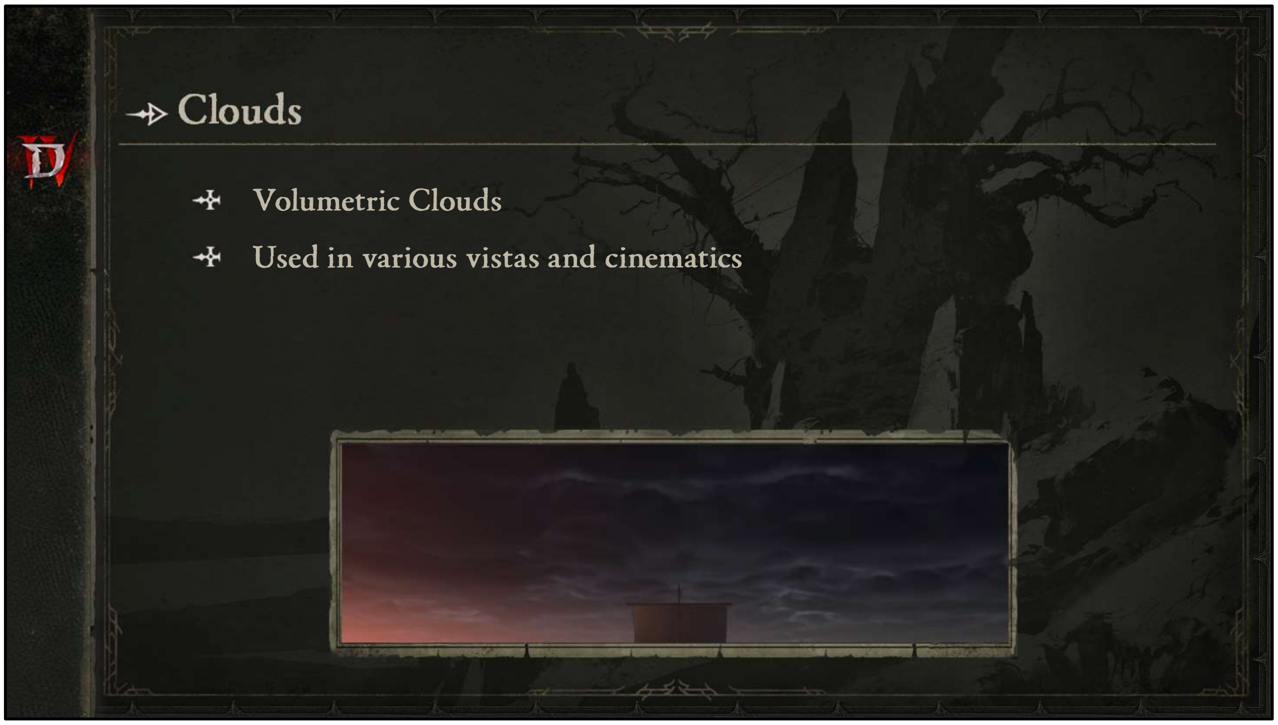
- Rivers, ponds, and puddles
- Oceans have masked areas near the shore

Here we come to Helltide, which is when different areas of the game dramatically change their look and challenge for better rewards.

When helltide is active, rivers, ponds, and puddles, turn a deep red with darker values around the edges.
Oceans have use some very simple world space sine math to achieve a very efficient, yet comparable effect.



And here you can see that represented of the left with the ocean where areas near the shore use sines with world x and y for efficient, variegated coloring.
As we move into the town, you see the dynamic puddles we saw earlier have been taken over by the ominous nature of helltide, conveying an added sense of danger to the player.



Clouds

- Volumetric Clouds
- Used in various vistas and cinematics

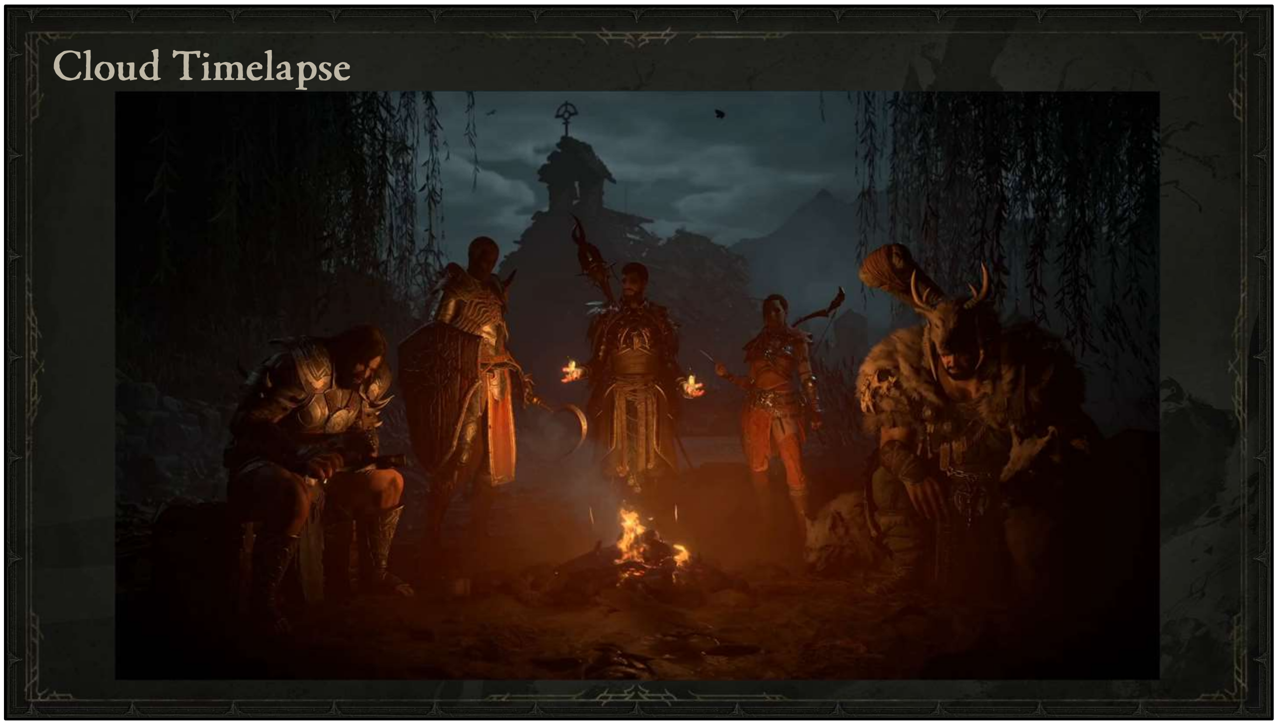
On to clouds.

We actually have volumetric clouds in our game. This effectively means that cloud shapes are represented by combining 3D volume textures, which are then ray marched to calculate lighting and shadows.
To keep this affordable, we don't update every pixel at once, but accumulate them over time, similar to other talks you may have seen on volumetric clouds.

As far as Diablo IV goes, they're used in various vistas and cinematics. You may have noticed them in the very last scene of the campaign which is partially pictured here, but another key place you might have noticed them is...

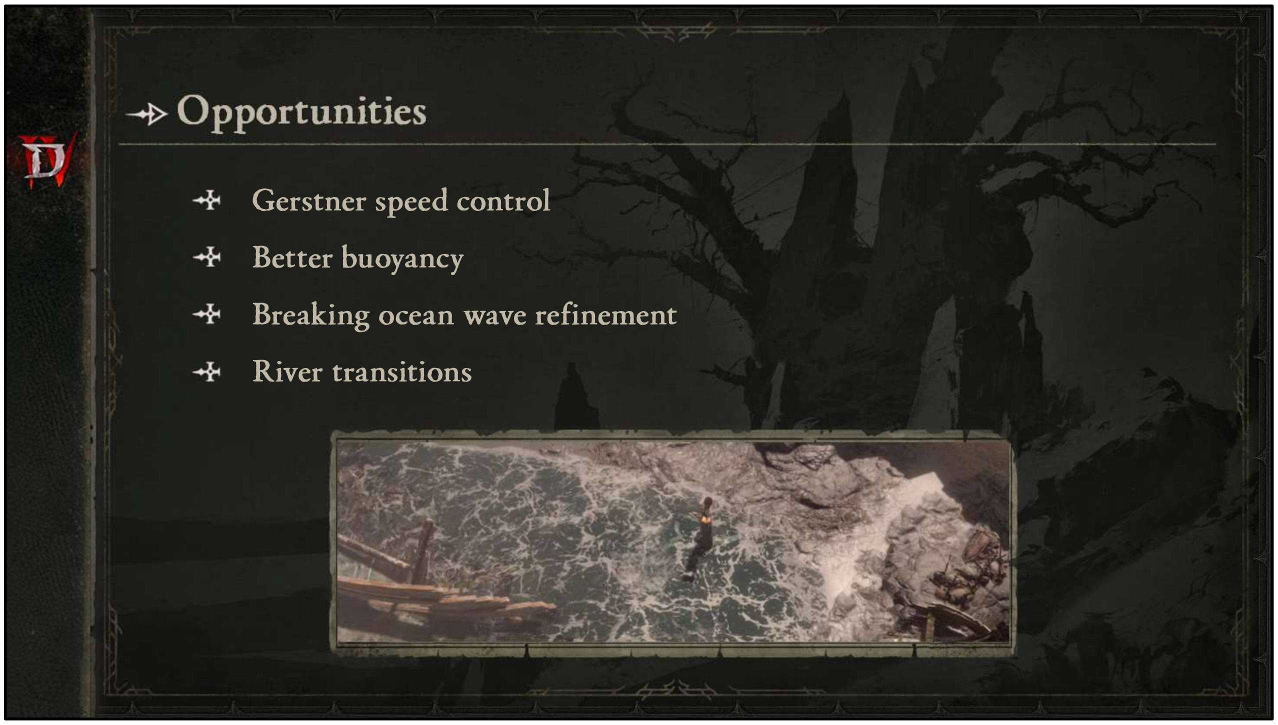
Cloud Showcase

...here at the very start of the game in the campfire scene where you select your character.
If you look close enough, you'll notice the clouds changing shape over time.



Cloud Timelapse

That’s even more apparent here if we speed up time. You’ll see the clouds progressing across the volume of the sky and evolving their shapes as they go.



Opportunities

- Gerstner speed control
- Better buoyancy
- Breaking ocean wave refinement
- River transitions

Alright. So what are some opportunities for the future?
We'd like to refine our control over Gerstner waves and their speed.
I mentioned earlier how we have the ability to make props buoyant in the ocean, but I think that's also an opportunity where we could make the result more versatile.
The breaking waves on the ocean are something I'd like to continue to polish over time.
And here's a noteworthy one. Rivers don't have an easy way to transition into the ocean, so level artists had to be creative whenever a river or other body of water was letting out near the ocean, so this is something I think we could explore more of in the future.

Final Thoughts

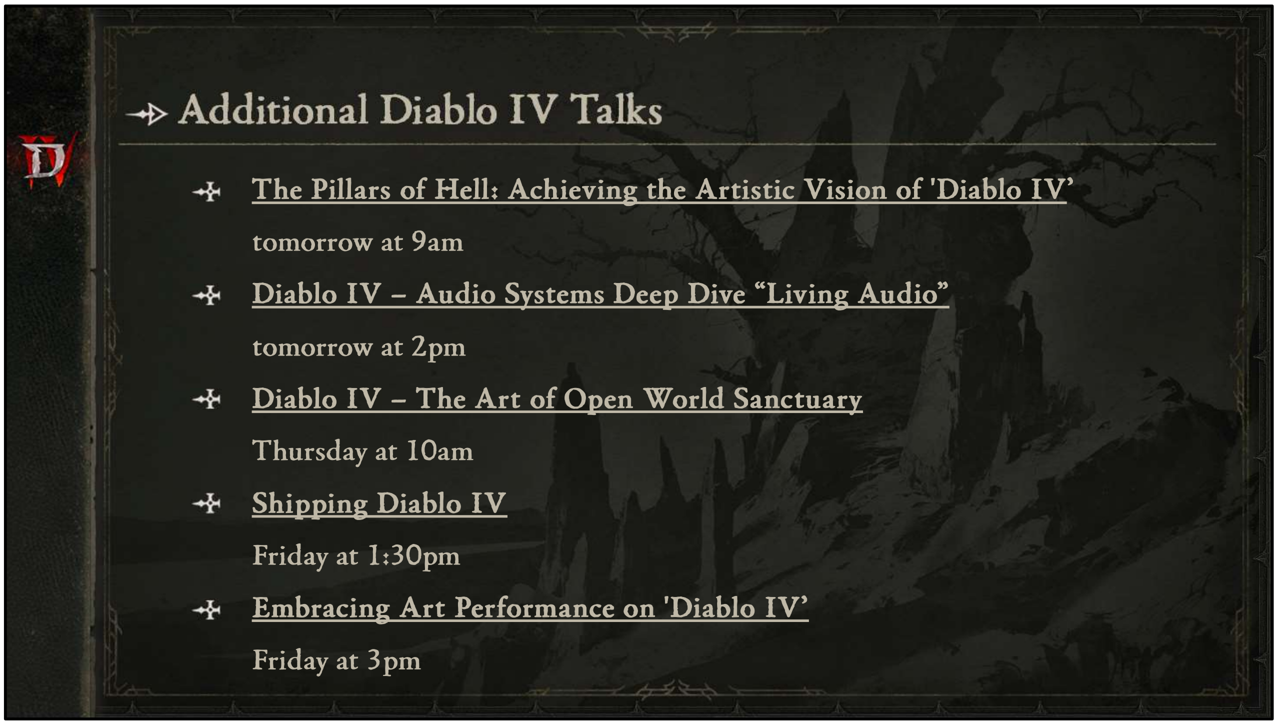
- Emulate reality, but bend it to suit the game
- Building up from the tried-and-true shader techniques can get you a long way
- Shout out to our artists and engineers for all their help

So here are my final thoughts.
As you will have seen, emulating reality is key to creating a believable living world, but you may need to bend aspects of that physically-based approach to suit the gameplay needs of your game.
Like you may have gathered from the various shader function fundamentals and pseudocode I showed, you can get a long way with tried-and-true shader techniques and building upon them.
I also wanted to give a shout out and thank you to all of the highly talented artists and engineers who supported the visuals presented here today through everything from the tech that makes it possible to the textures and actual assets that create this living world.

Acknowledgements

- John Buckley
- Sebastien Poirier
- Milan Singh
- Alain Bellehumeur
- Eric Wiley
- Genesis Prado
- Phil Williams
- Matt McDaid
- Zach Vinless
- Kevin Klein
- Dustin Biser
- Chris Donelson
- Alice Gionchetta
- Boyd McKenzie
- John Mueller
- Leadership, art, tech art, engineering, QA, production, and more

I'd like to specifically call out a few folks who influenced the visuals we've seen here today, but this is not an exhaustive list.
On the engineering side, John Buckley for the Gerstner waves, force simulation, and initial pass at procedural wetness.
Zach Vinless for his work on water reflections
Sebastien Poirier for access to the terrain depth for waves, and terrain curvature for puddles, and for terrain tessellation and deformation.
Kevin Klein for continued work on the snow deformation pixel shader.
Milan Singh and later Dustin Biser for taking volumetric cloud prototypes to a reality.
Alain Bellehumeur for transferring my hlsl logic to the cpu-side so we could dynamically spawn particles based upon the wave height.
On the art side, shout out to Eric Wiley for all of the textures used on water materials and terrain materials. On the dungeon side, shout out to Genesis Prado for ice textures, and Alice Gionchetta for snow textures.
Thank you to Phil Williams and Boyd McKenzie on the level art side for embracing and amplifying the ocean sculpting techniques you saw here today.
Thank you to Matt McDaid for feedback and to the Art Director, John Mueller, who is a self-described "wave nerd" and gave me a lot of valuable feedback as I was working on the ocean
And there are many more talented folks, as we like to say "It takes a Blizzard"



Additional Diablo IV Talks

- The Pillars of Hell: Achieving the Artistic Vision of 'Diablo IV'
  tomorrow at 9am
- Diablo IV – Audio Systems Deep Dive "Living Audio"
  tomorrow at 2pm
- Diablo IV – The Art of Open World Sanctuary
  Thursday at 10am
- Shipping Diablo IV
  Friday at 1:30pm
- Embracing Art Performance on 'Diablo IV'
  Friday at 3pm

The Pillars of Hell: Achieving the Artistic Vision of 'Diablo IV
John Mueller

Diablo IV – Audio Systems Deep Dive "Living Audio"
Kris Giampa & Michael Bartnett

Diablo IV – The Art of Open World Sanctuary
Mahreen Fatima, Sam Gao, & Justin Whitehead

Shipping Diablo IV
Tiffany K. Watt & Michael Bybee

Embracing Art Performance on 'Diablo IV'
Evan Edwards

Thank you very much for going on that water shader journey with me.