

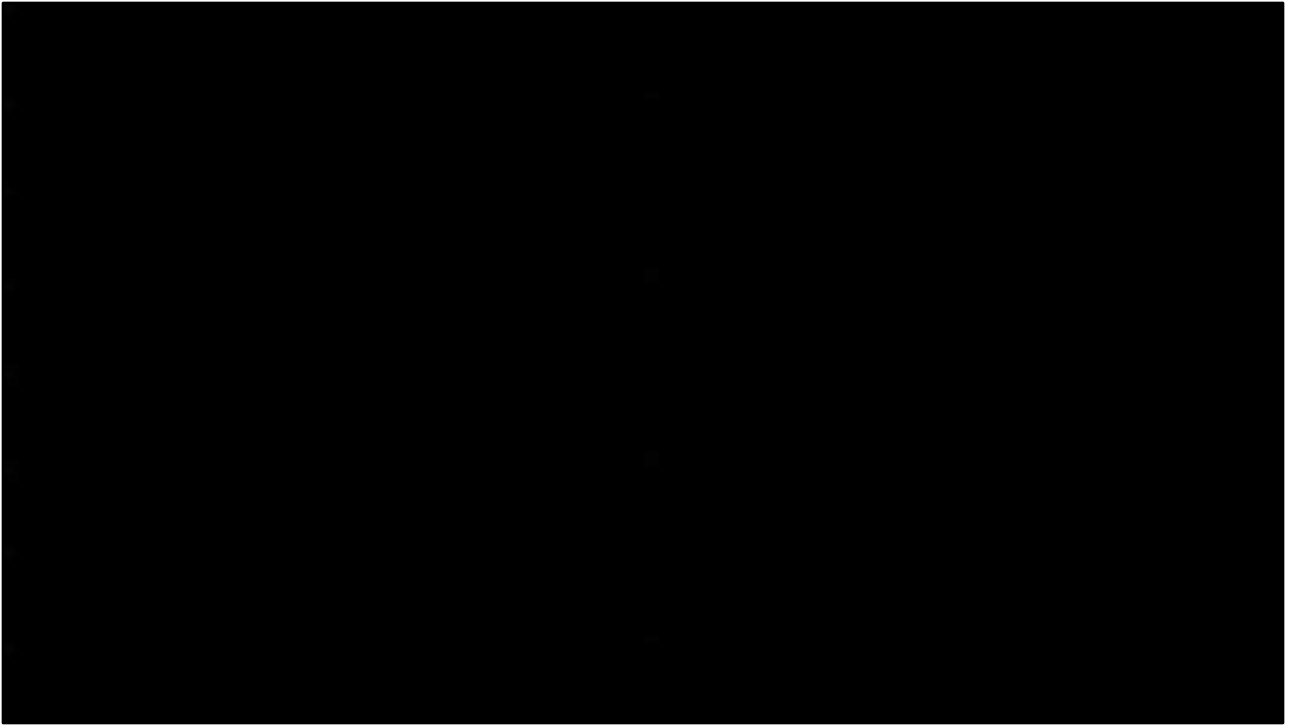
The Technical Art of Uncharted 4

Waylon Brinck – Technical Art Director
Andrew Maximov – Lead Technical Artist



Render the Possibilities
SIGGRAPH2016





Video Introduction



Introduction...

In this talk we'll be presenting features that were implemented by the technical artists at Naughty Dog. This is our first game using a Physically-Based Renderer. We found that following a physically-based rendering philosophy actually made our jobs easier, made cleaner code, and made the artist interface much more intuitive. So we'll look at some features that were built using that philosophy!

Overview



We're going to cover a lot of individual topics here. First Waylon will talk about some rendering features covering everything from post processing, to atmosphere, to pixel shaders...



Then Andrew will talk about simulating washing machines. 45 minutes of washing machine goodness and all the vertex processing tech leading up to it.

About Our Renderer

- Singleplayer 30fps 1080p, Multiplayer 60fps 900p
- Physically-Based Renderer based on Disney BRDF (with plenty of our own tweaks) [Burley2012]
- Deferred Renderer
- One uber-shader written in PSSL that's used on all environments and characters



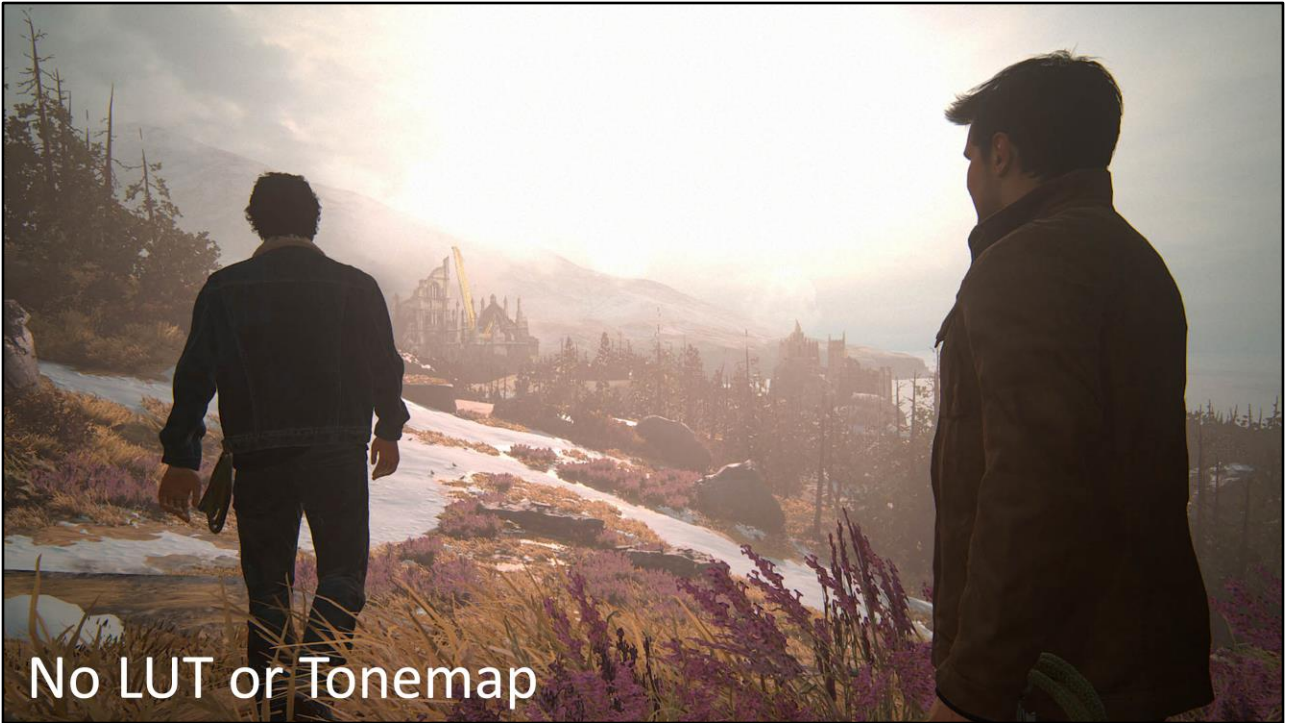
First, a few key things to know about our renderer. Note, we don't do any node-graph-based editing for our environment shaders – when you get this complicated, it's much easier to manage in traditional code!

Tonemapping and HDR Color Grading

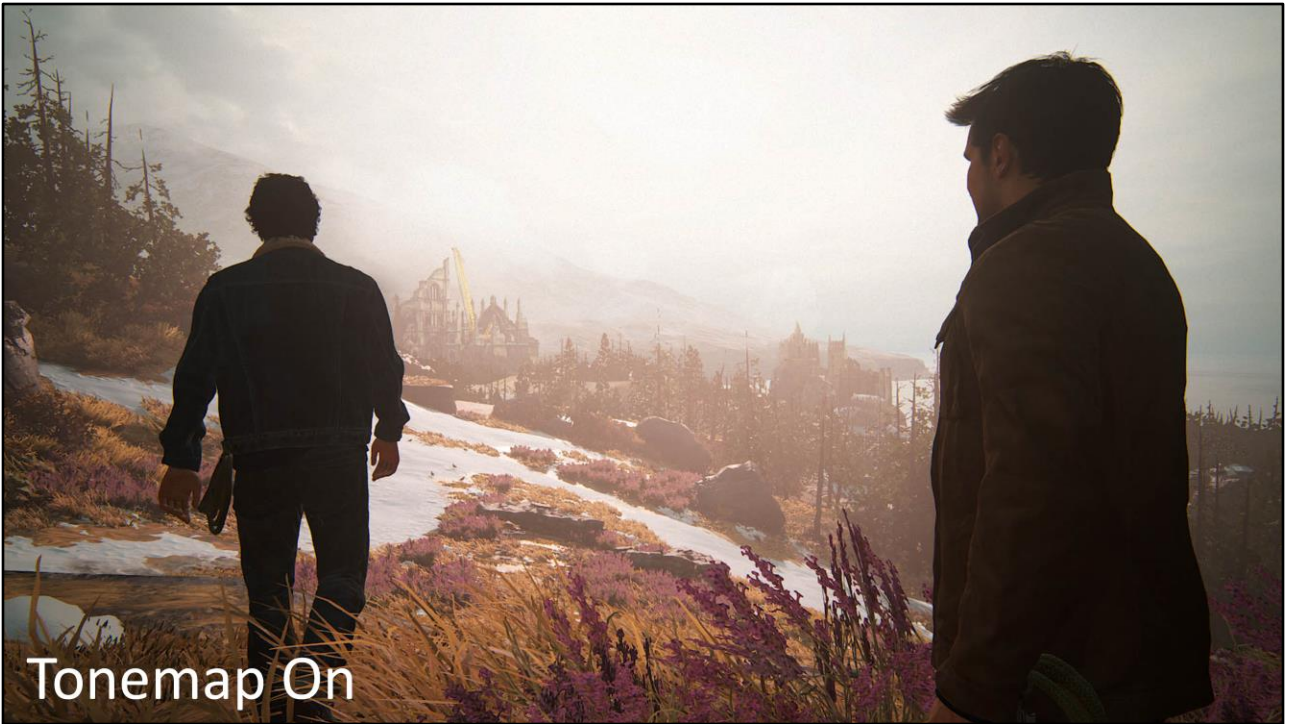
HDR Lookup Table (LUT)



Let's start the talk with a new twist on some old features.



First, here's a screenshot with the HDR LUT turned off. There's no artistic color grading and no tonemapping.



Here we turn on tonemapping. Note it adds a little contrast in the shadows, but importantly, it prevents the yellow/pink color banding when the clouds get overexposed.



And here we turn on the Color LUT. You can see we're artistically applying a cool tint. This is the final shot from the game, but so far, nothing we couldn't do before.



First question – why do we want to color grade in HDR? It allows us to treat the color grading more like a filter on the lens, rather than a post process effect. Let's do an extreme example. Here's a shot from the final game.



Same shot, but with a strong cyan tint in “multiply” mode. Note that it completely saturates the whites in the clouds.

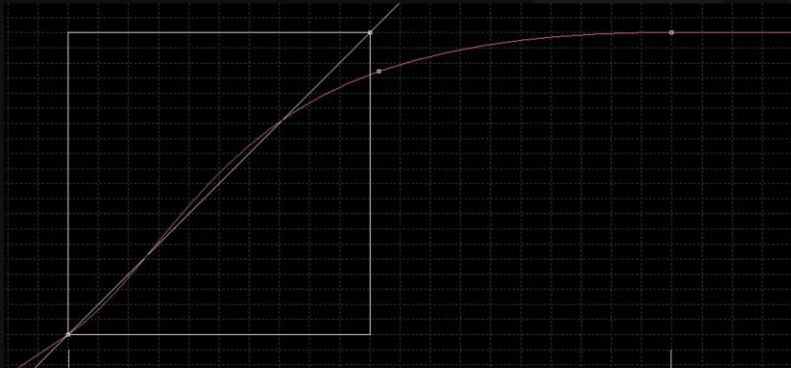


HDR Tint

...Versus in HDR space, even with the cyan multiply, some of the white from the clouds still shows through. Yes, this can be achieved in LDR with masking or advanced blend modes, but in HDR it happens automatically.

```
tonemap-filmic-white-point          0.0000
tonemap-filmic-shoulder-strength  -2586.3655
tonemap-filmic-linear-strength     0.6900
tonemap-filmic-linear-angle        -767.6706
tonemap-filmic-toe-strength        -8.5706
tonemap-filmic-toe-numerator        2.8784
tonemap-filmic-toe-denominator     107.4683
tonemap-filmic-exposure-bias       0.0000
```

[Hable2010]



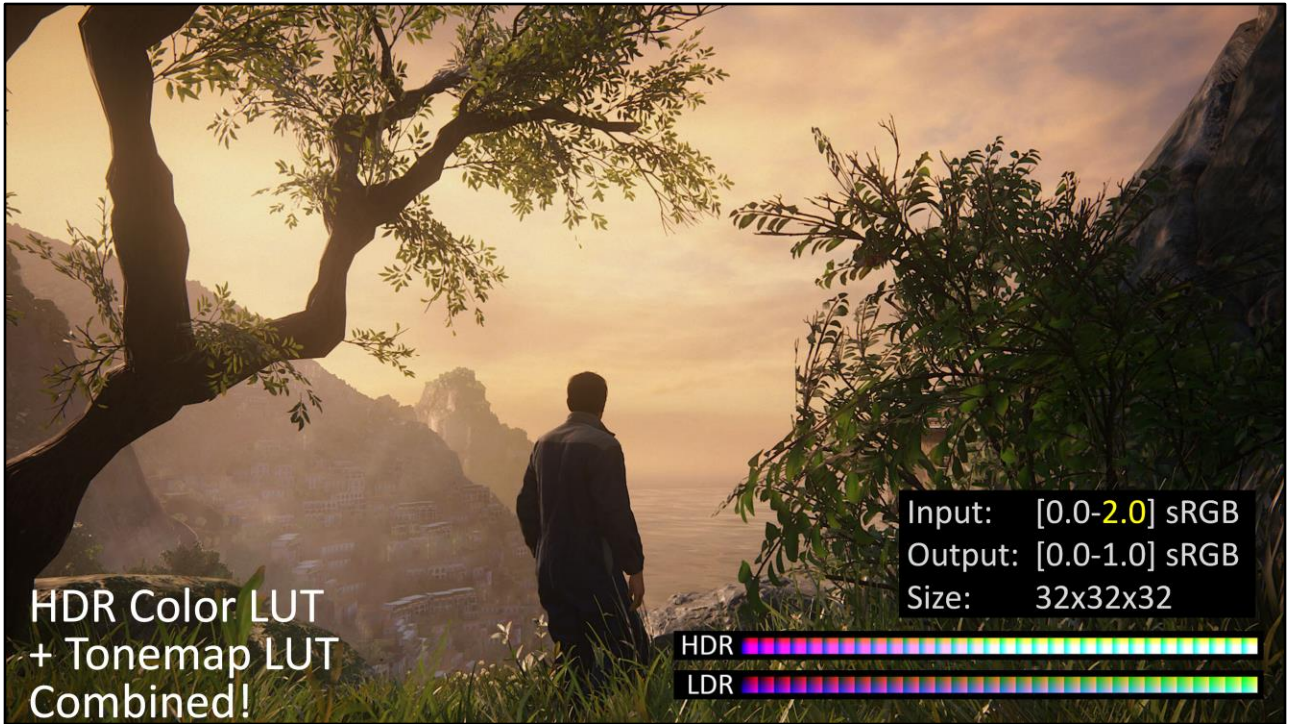
Input [0.0-2.0]

Output [0.0-1.0]

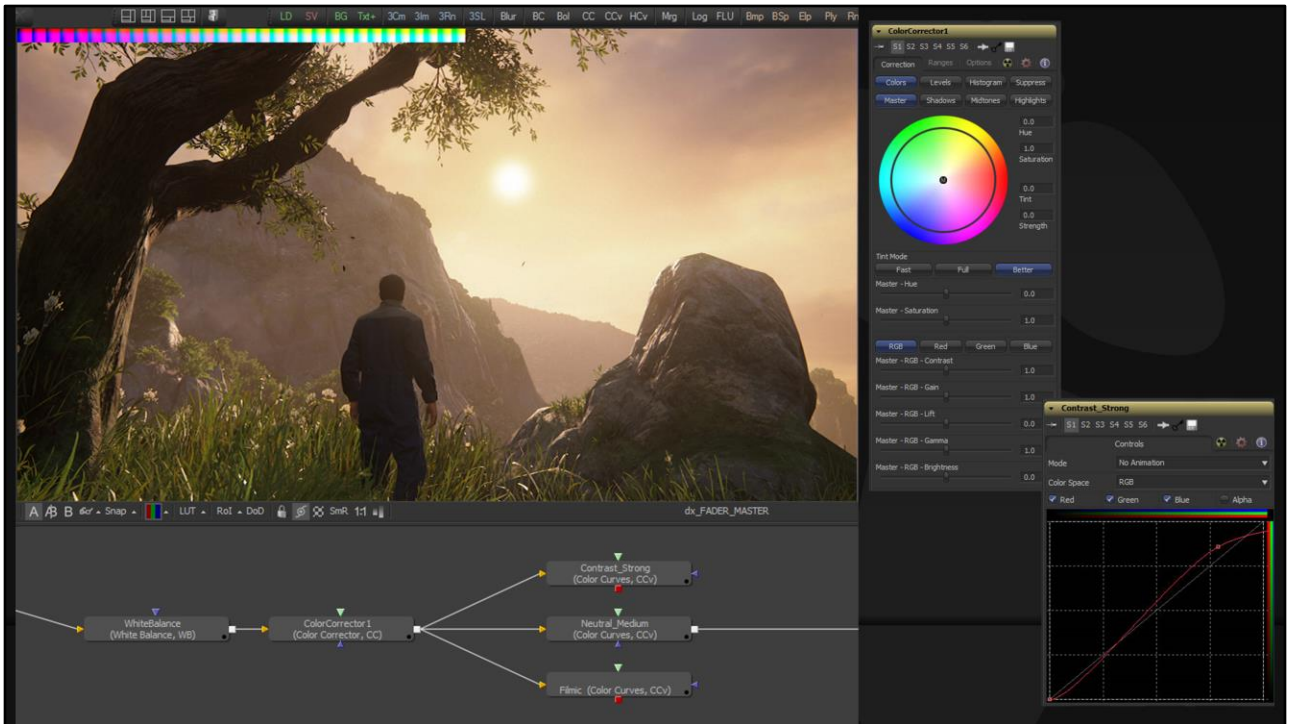
NAUGHTY DOG

Tonemap Controls

On previous projects we had the tonemapping function implemented by John Hable. But really, these input numbers were just controlling a parameteric curve that maps HDR values into an LDR space. We can get the same mapping with a curve editor in familiar graphics software. We COULD store this mapping in a grayscale LUT texture...

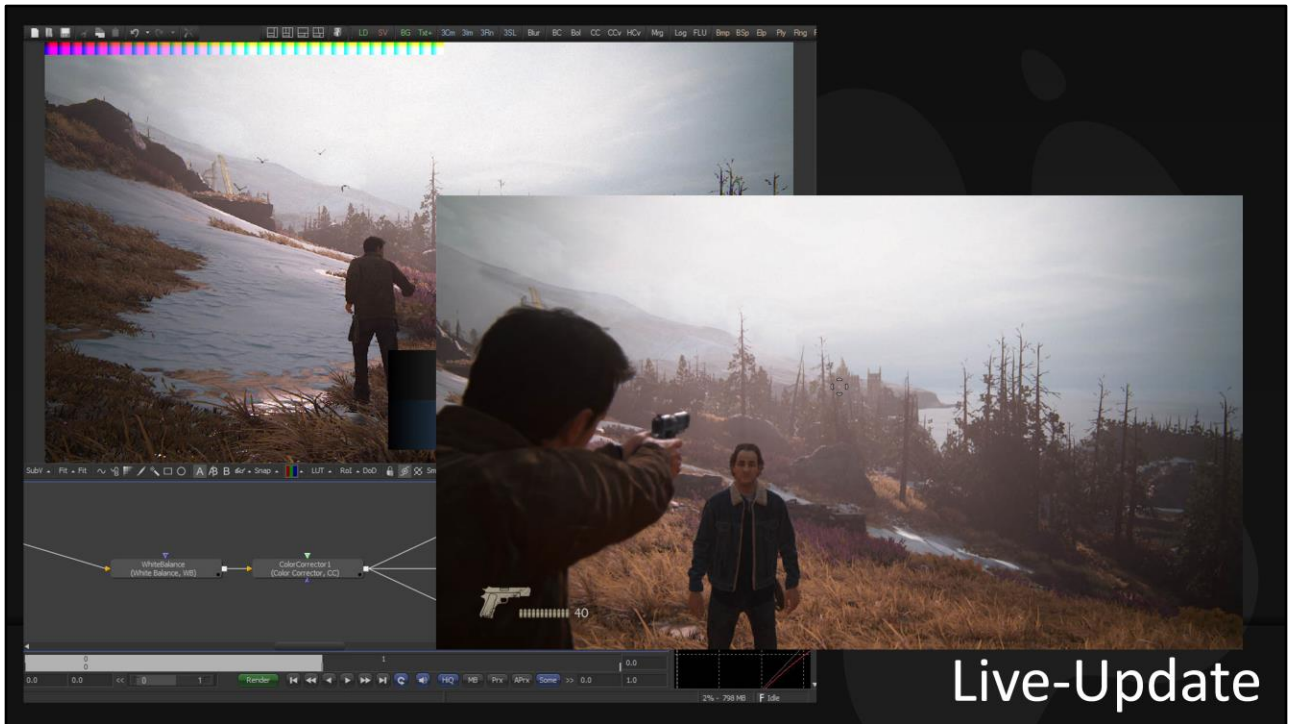


We decided to encode BOTH our artistic color grading AND our tonemapping curve into a single HDR LUT texture. Our LUT represents an input range of 0.0-2.0 sRGB instead of the traditional 0.0-1.0, with an output range of 0.0-1.0 sRGB. Note, we originally experimented with larger input ranges, up to 4.0 sRGB. There were no technical issues (no noticeable lack of precision), but for the painterly look of Uncharted 4, the extra range wasn't necessary.



We decided to use a compositing program called Fusion for our HDR color grading and tonemapping curve editing. (You could also use something like Nuke or DaVinci) Fusion works in HDR by default, and the node-based workflow is great for color grading – but the color correction tools themselves are extremely powerful and intuitive compared to what comes with Photoshop.

So what we have here is the flow of color operations. On the bottom-left are a few user-specified color correction nodes, which can get as complicated as the artist likes. To their right we provide a few presets for tonemapping curves, which the artist can choose from and modify as needed.



One other benefit of this workflow – as the artist tweaks the LUT in Fusion, it live-updates in-game. Fusion has some basic scripting ability. After every edit, we save the cropped LUT to disk, and send a message to the game to refresh. In practice this refresh happens at about 5fps, which is plenty fast for live editing.

One other major benefit of this workflow – the artist can freely look around the scene as they're tweaking colors to make sure it looks good everywhere – they're not locked into their initial screenshot.

Mip Fog





Next let's talk about fog. Our artists wanted very specific control of the fog color in different directions. Especially, they wanted the color of the fog to match the color of the skydome at the horizon. As a quick illustration, this is an image of our final result in-game....



Constant Fog Color

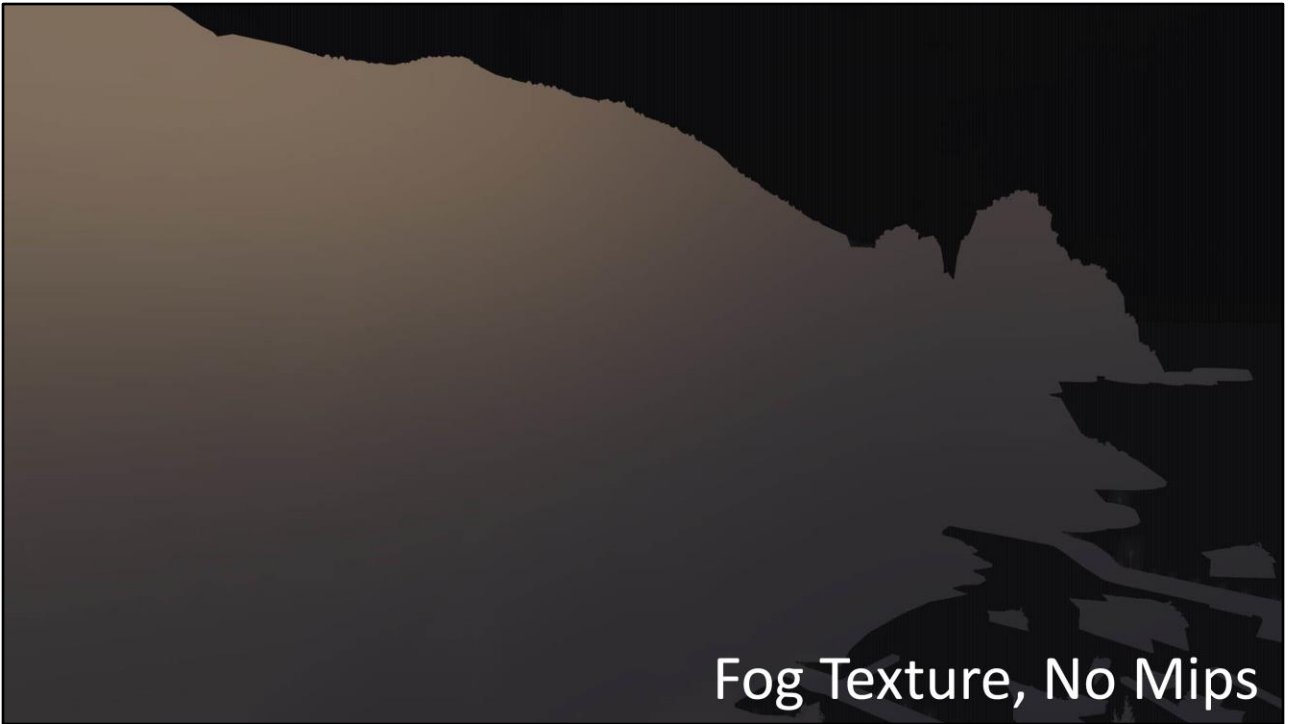
.. And here's just a standard monochromatic fog.



NAUGHTY DOG

Fog Texture

So we let our artists paint a texture that controls the color of the fog. Here's a sky texture and its associated fog texture. You can see the color and energy for the upper hemisphere is the same, and the artist manually extended the fog image below the horizon for the lower hemisphere.

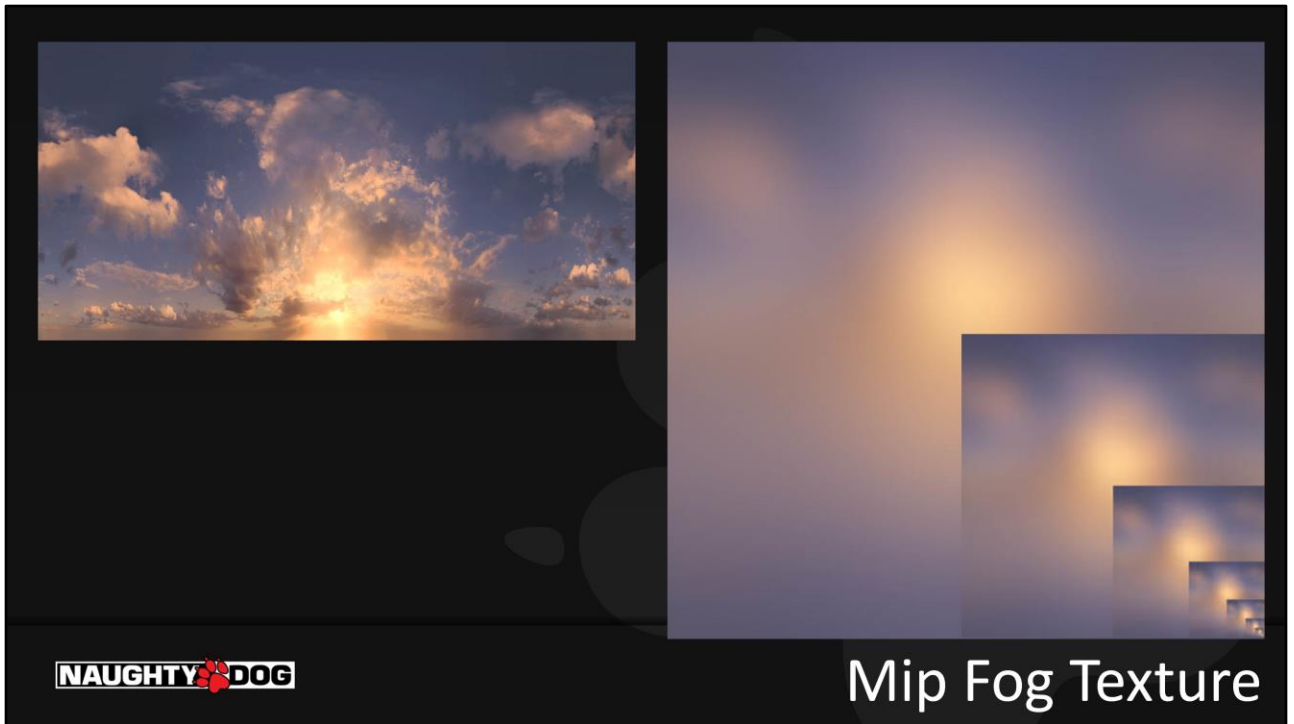


Here's that fog texture, in action. The problem though, is that objects in the foreground receive the exact same fog color as what's behind them. (The problem is even worse in an interior, or behind a wall!)

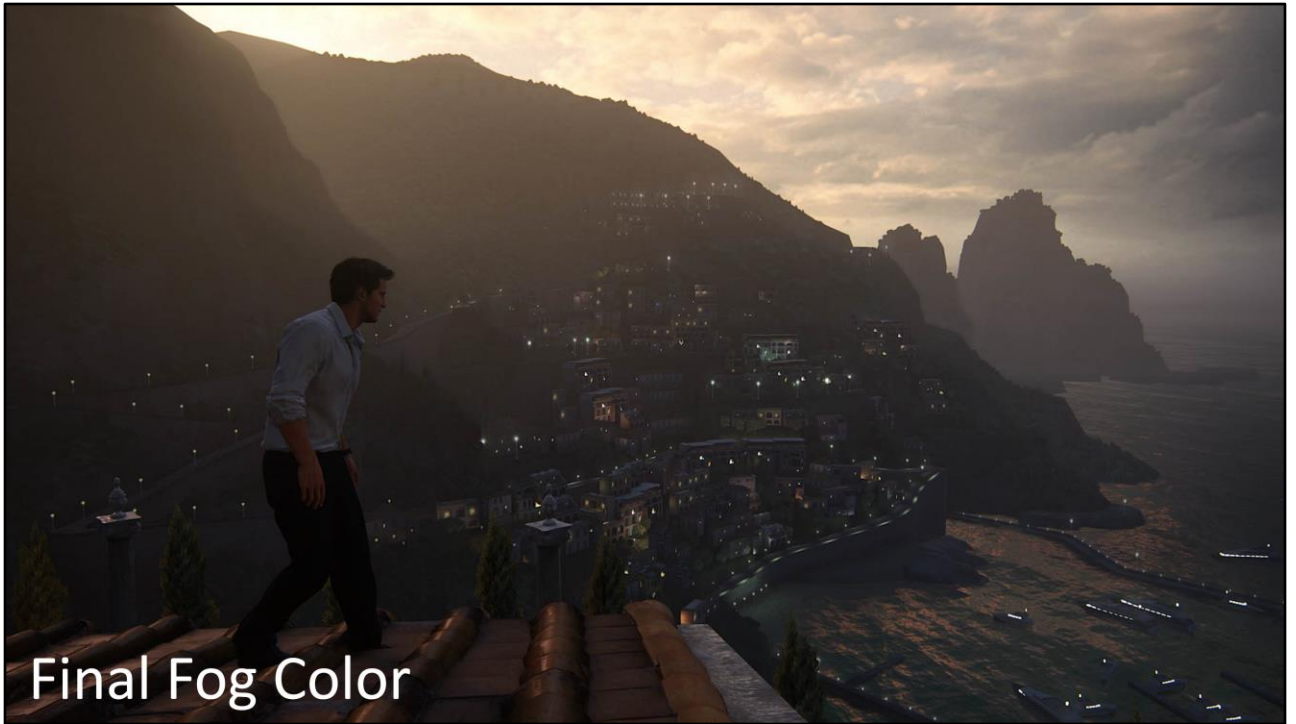


Fog Texture With Mips

So we use the low mips (full-res) at the horizon, and the high mips (low-res) close to the camera. In this image, Drake is just pulling from a 1x1 version of the fog texture... which means he's getting the average color of the entire image. And when you think about it, the color of the fog for a given pixel is the accumulation of all the light that strikes the particles in between the camera and the destination surface. (plus or minus a few scattering and absorption coefficients.)



Here's that fog texture again, with its mips, to illustrate the point. They get blurrier and blurrier, until it's just a single pixel with a solid color.



Ultimately, the mip fog rendering is fairly plausible, since it captures the color of the lighting affecting different parts of the scene.

```
mipLevel = (1.0 - saturate((depth - nearParam) / (farParam - nearParam))) * numMipLevels;  
fogColor = SampleFogTexture(cameraVec, mipLevel);
```



To sample the fog texture, we do an approximate translation of the camera vector to the sky's UV space, which we use to look up into the mip fog texture. The fog density is calculated separately, based on typical artist controls like height falloff, distance falloff, etc.



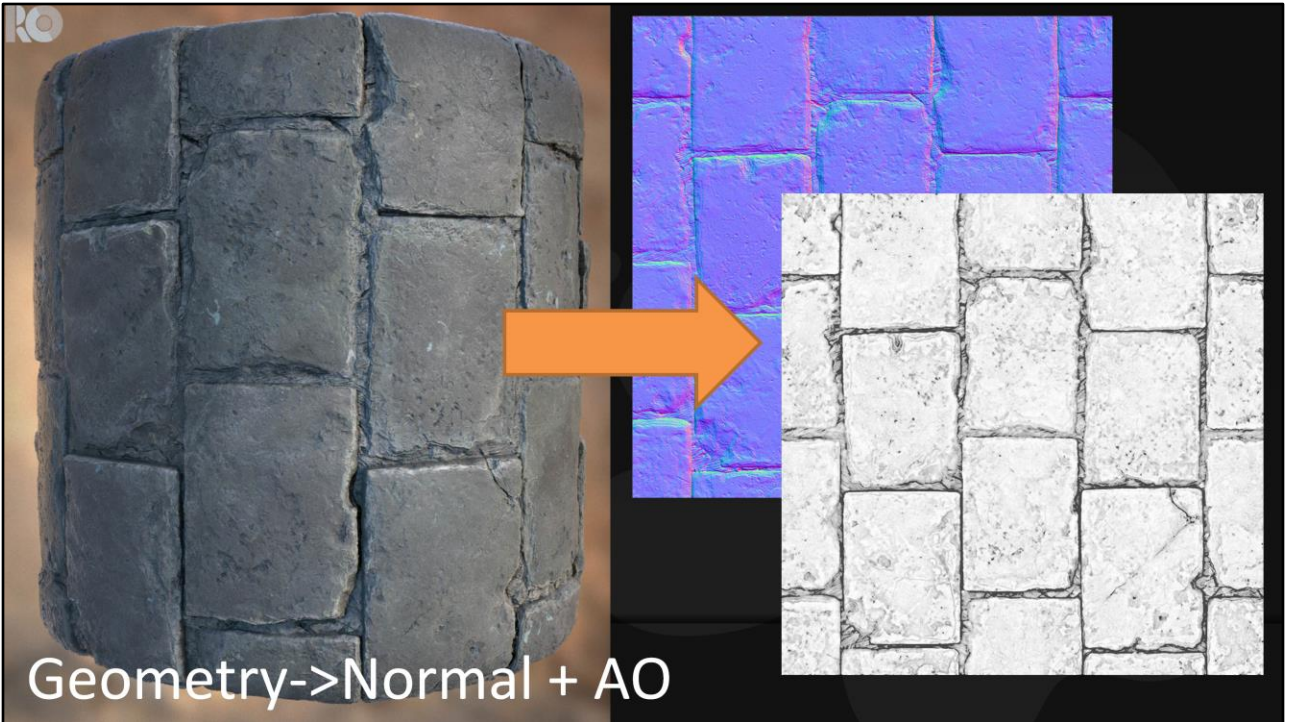
Also:

- Sun Volumetrics
- Runtime Volumetrics
- Horizon Blend Fog
- Distance Contrast
- Near Fog Color Override
- Particles

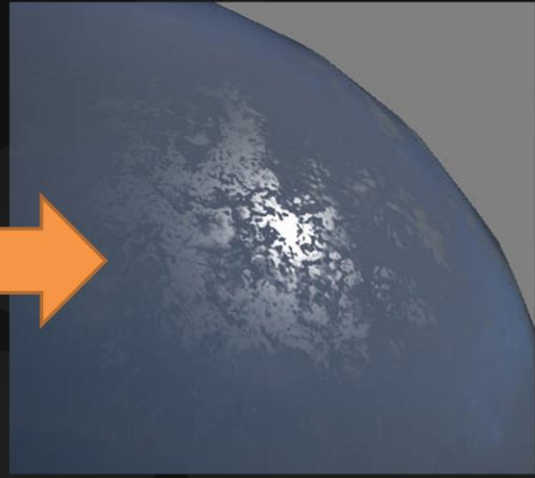
We have a lot of systems in place to build our atmosphere - sun volumetrics, runtime light volumetrics, a special fog term to blend the horizon, an artistic distance-based contrast control, a fog near-color override, and of course particles! But the mip fog is the place where our artists start.

Micro-Shadow BRDF Term





First, as a review let's talk about data: geometry, texture, and subpixel. Something like Albedo/Color or Specular IOR is pretty scale-agnostic. When we go from high-poly geo to low-poly, we typically capture the difference in a Normal map and Ambient Occlusion map.



NAUGHTY DOG

Geometry->Roughness

When we have geometry or texture data that becomes subpixel, that's essentially our BRDF, captured in parameters like Roughness or Anisotropic.

What's Lost?

Geometry->Texture

- Parallax Occlusion [Tatarchuk2006]
- Parallax Shadows [Tatarchuk2006]
- Bounce Light

Geometry->Texture->BRDF

- ... All of the above
- Roughness [Toksvig2005]
- Specular Anisotropy
- Lots more subtle stuff



At each reduction step we lose data, and the error in the amount of energy increases. By providing new types of input data and spending some ALU, we can improve on each of these phenomena. For example, by supplying a heightmap, we can perform parallax occlusion and parallax shadowing at the texture level. The point is, it's useful to examine these phenomena independently, because there may be ways to bring them back.



When we transitioned to Physically-Based Rendering, artists struggled with Ambient Occlusion, wanting to paint it into the base color, or wanting the AO to affect all light sources. Eventually we realized there's a physically-plausible way to do this. We call it "Micro Shadowing". Here's an example of a scene with the feature in place...



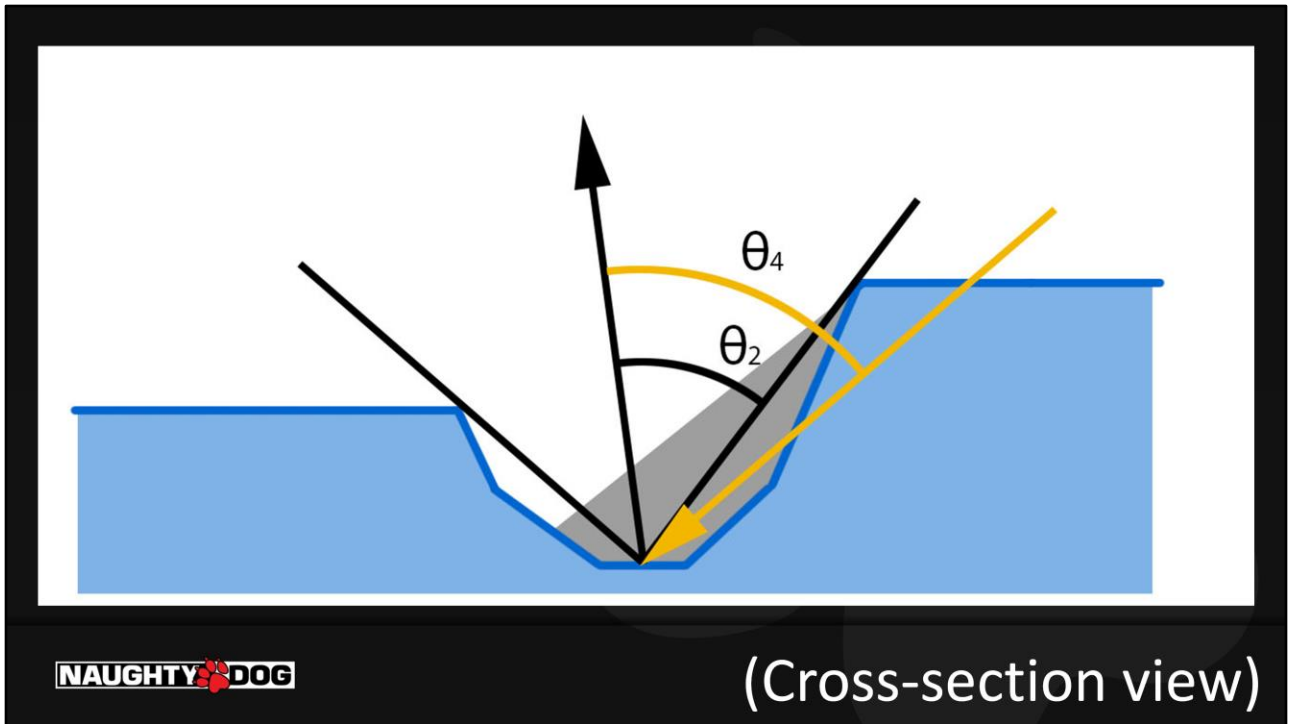
... And without. Note how it mostly appears at the light terminator, and fills in some of the cracks where the geometry represented by the normals doesn't cast a shadow, or where the sun shadow isn't precise enough.



And here's our sunlight isolated, with Micro Shadow on...



And micro shadow off



Here we have a cross-section of a surface.

Ambient Occlusion approximates the cone angle visible from a point.

We can define θ_2 as half of the cone angle.

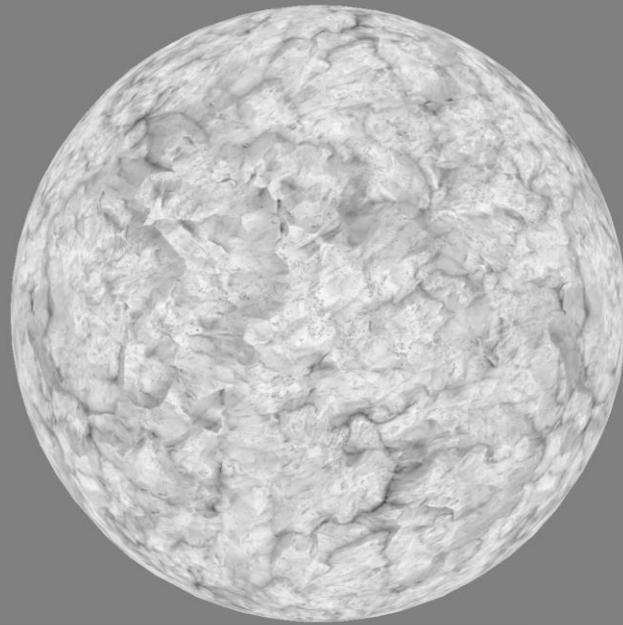
If we have an incoming light ray, we can define the angle between the light ray and surface normal θ . If $\theta < \theta_2$, the pixel receives light. If $\theta > \theta_2$, the pixel is in shadow.

```
float ApplyMicroShadow(float ao, float3 N, float3 L, float shadow)
{
    float aperture = 2.0 * ao * ao;
    float microShadow = saturate(abs(dot(L, N)) + aperture - 1.0);
    return shadow * microShadow;
}
// Ship it!
```



And this is where “tech” meets “art”. Our AO maps aren’t authored precisely or consistently. AO and a Normal aren’t enough data to represent the shadowing accurately anyway. So that’s already introducing quite a bit of error to the system. I’ll revisit this on the next project, try to get more correct without significantly introducing cost, but this was good enough to ship Uncharted 4.

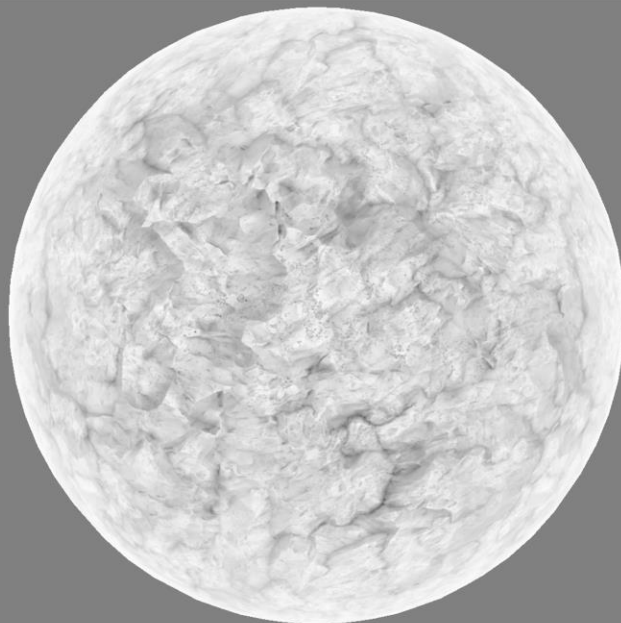
One other note - since we’re representing micro features here, we don’t want a hard shadow, we want a soft gradient, since we’re representing what percentage of each pixel is in shadow, not a hard cast shadow.



NAUGHTY DOG

Ambient Occlusion

One other closely-related feature is “Ambient Occlusion Occlusion”, or “AO Fresnel”. The concept is, ambient occlusion represents shadowing in the cracks – in this case, cracks that don’t exist in the low-poly model, but that have been baked down to normals. The thing is, when the object is viewed from a glancing angle, those cracks SHOULD be occluded by the geometry. Since the geometry doesn’t exist, we need to occlude the cracks in the shader.



NAUGHTY DOG

Ambient Occlusion Fresnel

... Which looks like this.

```
float aoFadeTerm = saturate(dot(vertexNormalWS, viewWS));  
ao = lerp(1.0, ao, aoFadeTerm);
```

- Only applied to baked objects



The code is dead simple. Probably we can find something that represents the curvature more correctly. Note we use the geometry normal, not the normal-map-normal, for the dot product.

Also note – this is only applied to surfaces with baked lighting. For objects with probe lighting (like characters), the AO map DOES include occlusion information from the in-game geometry, so we don't apply the viewing-angle fade there.



This is used throughout the whole game, by default, on every environmental surface. There are no artist controls to tune, it's just always applied.

Special-Case Materials

Moss



Let's go a little deeper into shading, and look at some specific material types.



Uncharted 4 features a lot of mossy surfaces (and more generally, a lot of “fuzzy” surfaces.)



Micro-fiber, not micro-facet!

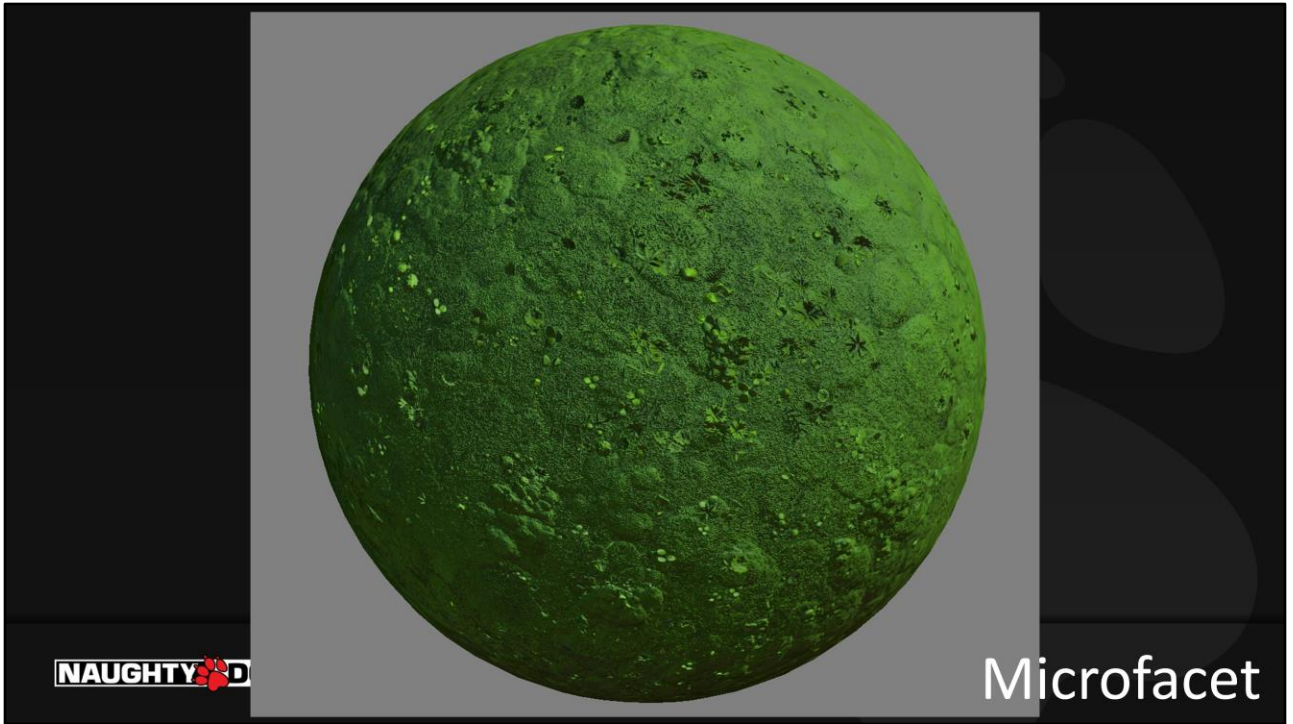
Moss doesn't follow a standard microfacet brdf model. It's a lot more porous, with a massive amount of self-shadowing, ambient occlusion, and transmitted light. The moss itself tends to have a lighter color at the tips than deeper towards the base. And there's a strong viewing-angle dependency – when viewed straight on, you can see down into the shadows between the fibers, and while viewed from an angle, only the tips are visible.



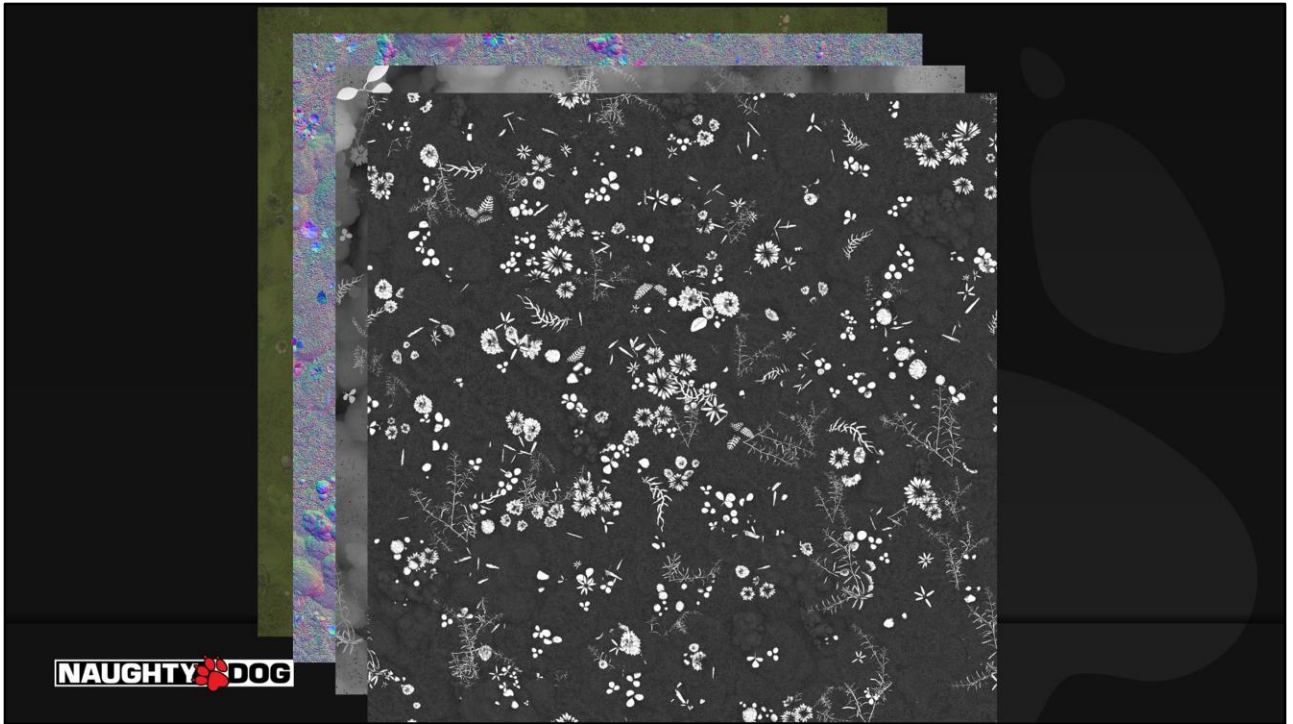
Here's a simplified example of the problem – imagine this is a close-up shot of a 1cm sphere. You can see how all of the subtle properties of the fuzzy surface start to come through. And when we zoom out so the details are subpixel, the resulting brdf looks very different than a microfacet model. We need to solve both of these cases for our moss shader in-game – moss details are large enough to be seen in the texture when the camera is close, but become subpixel when more than a few meters away.



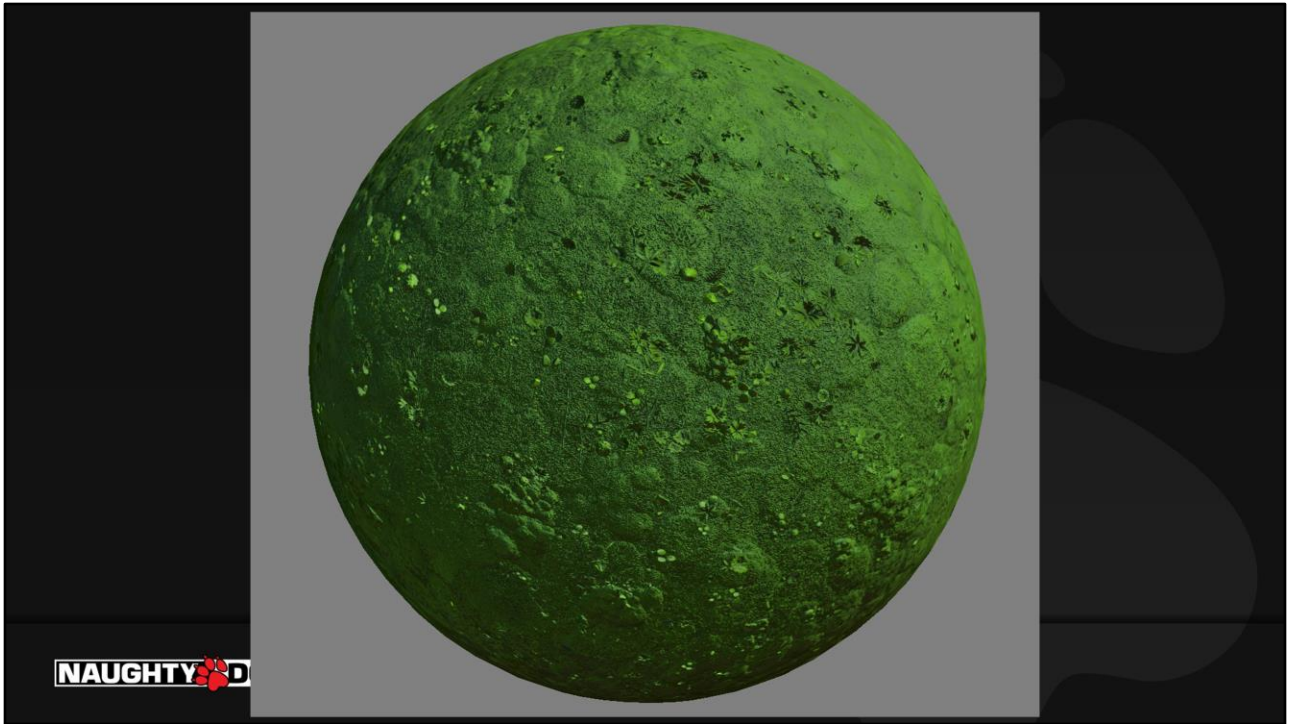
Here's a typical moss shader from Uncharted 4, with all the bells and whistles applied.



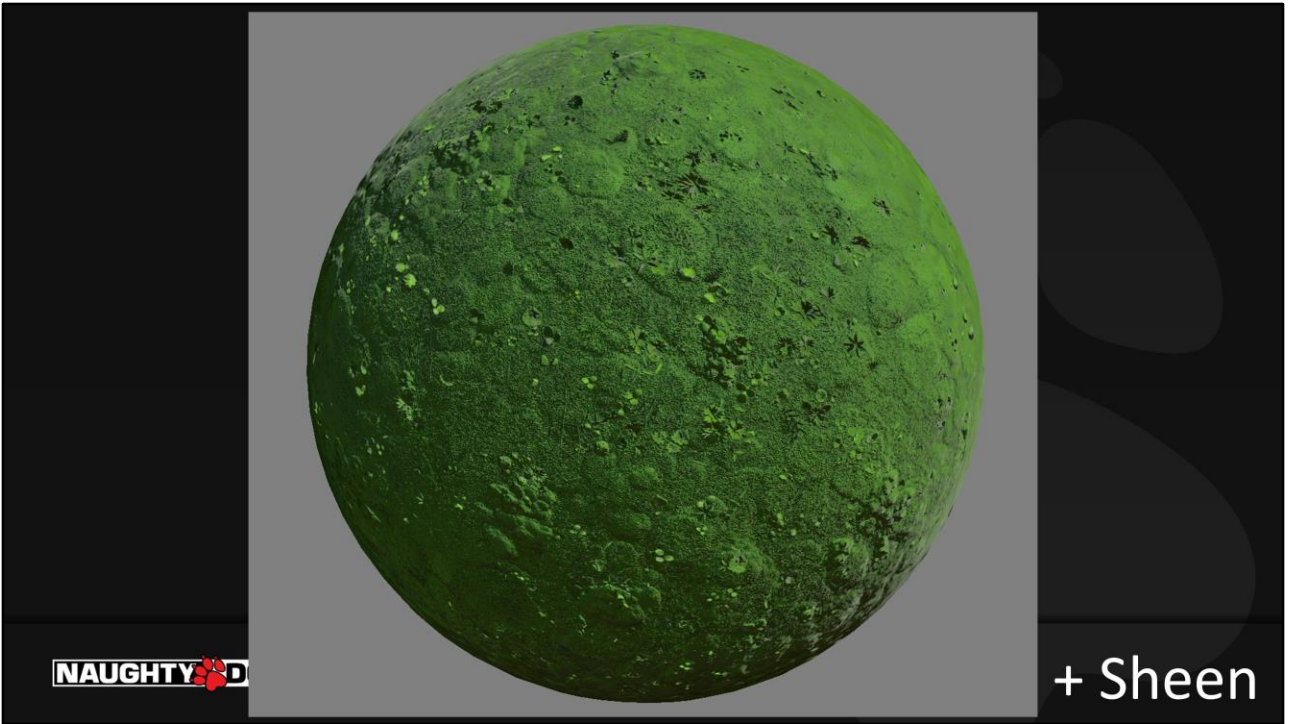
Compared to a standard microfacet model with the same texture inputs, which looks like hard plastic by comparison.



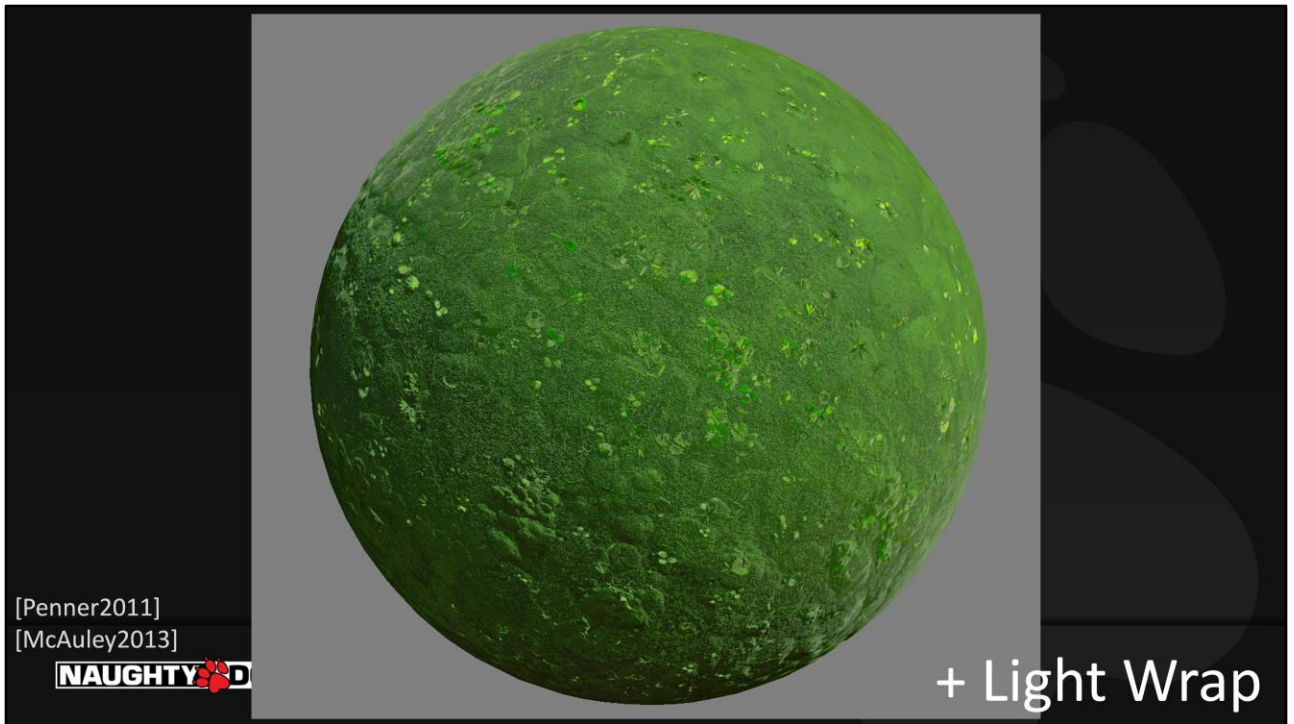
Here are the four main contributing textures – Color, Normal, Heightmap, and Ambient Occlusion. Note how dark the AO map appears – we’ll get back to that in a minute.



Let's build up the shader one feature at a time. Here's our base microfacet BRDF.



We add in Sheen... it helps a little, but not nearly enough.



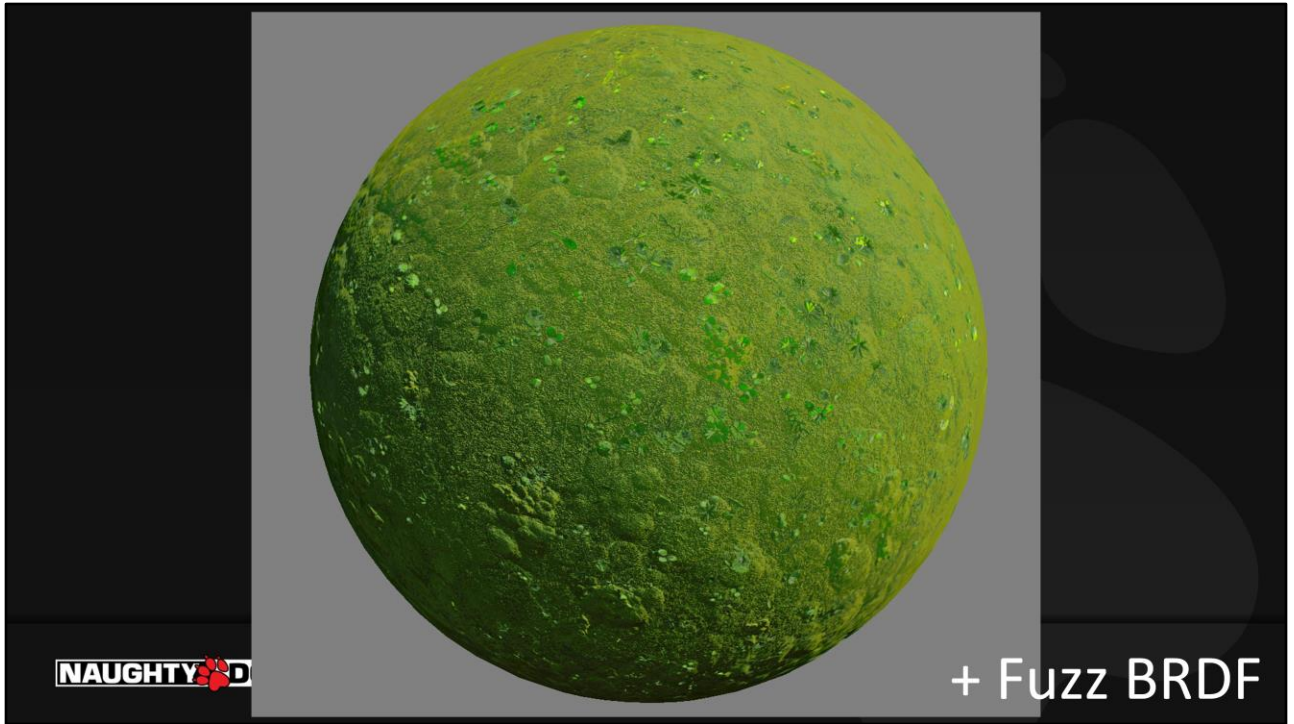
[Penner2011]

[McAuley2013]

NAUGHTY D

+ Light Wrap

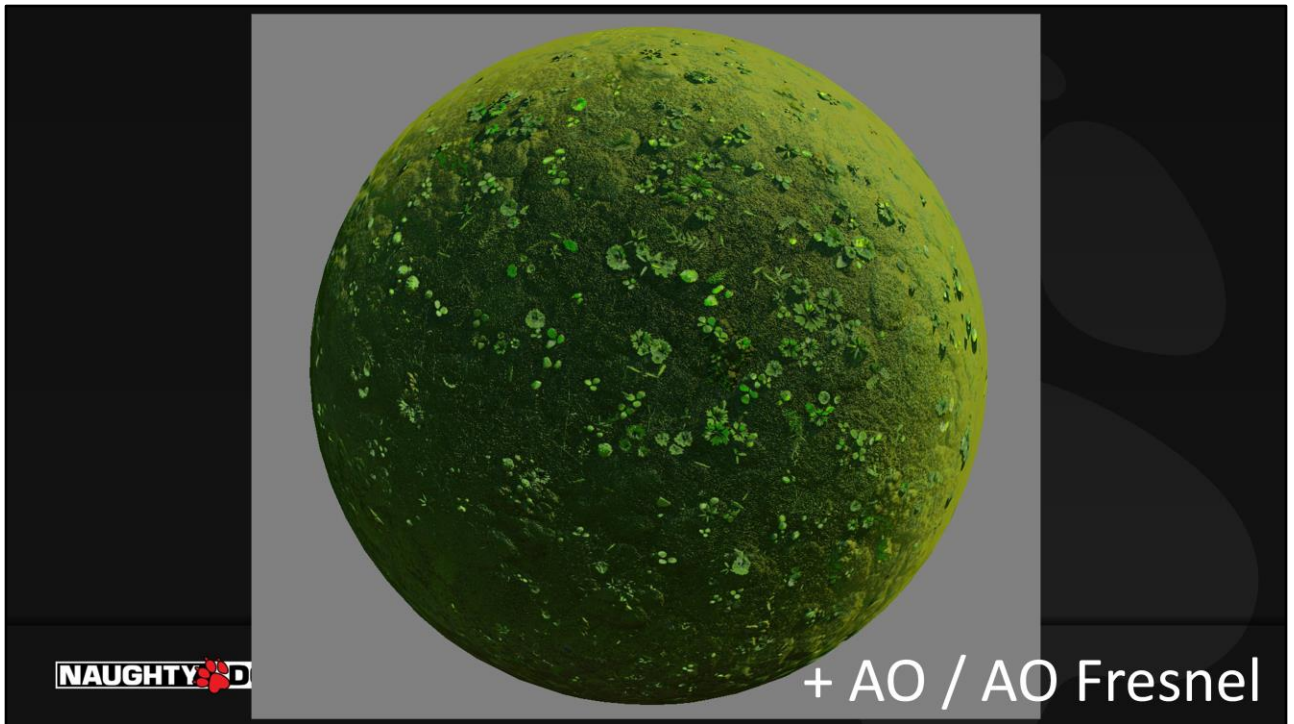
We add a Light Wrap feature. This modifies the basic NdotL of the light, letting it travel further around the mesh. This simulates both Subsurface Scattering, and also the phenomenon where fuzzy fibers beyond the usual 90-degree light terminator may not be in shadow. Our light wrap ignores the normal map, so it fills in the cracks more nicely. And the artist can also supply a tint for the wrapped light (in this case, green!) This is inspired by Eric Penner's Pre-Integrated Subsurface Scattering. For a good monochromatic solution, refer to Steve McAuley's Extension to Energy-Conserving Wrapped Diffuse



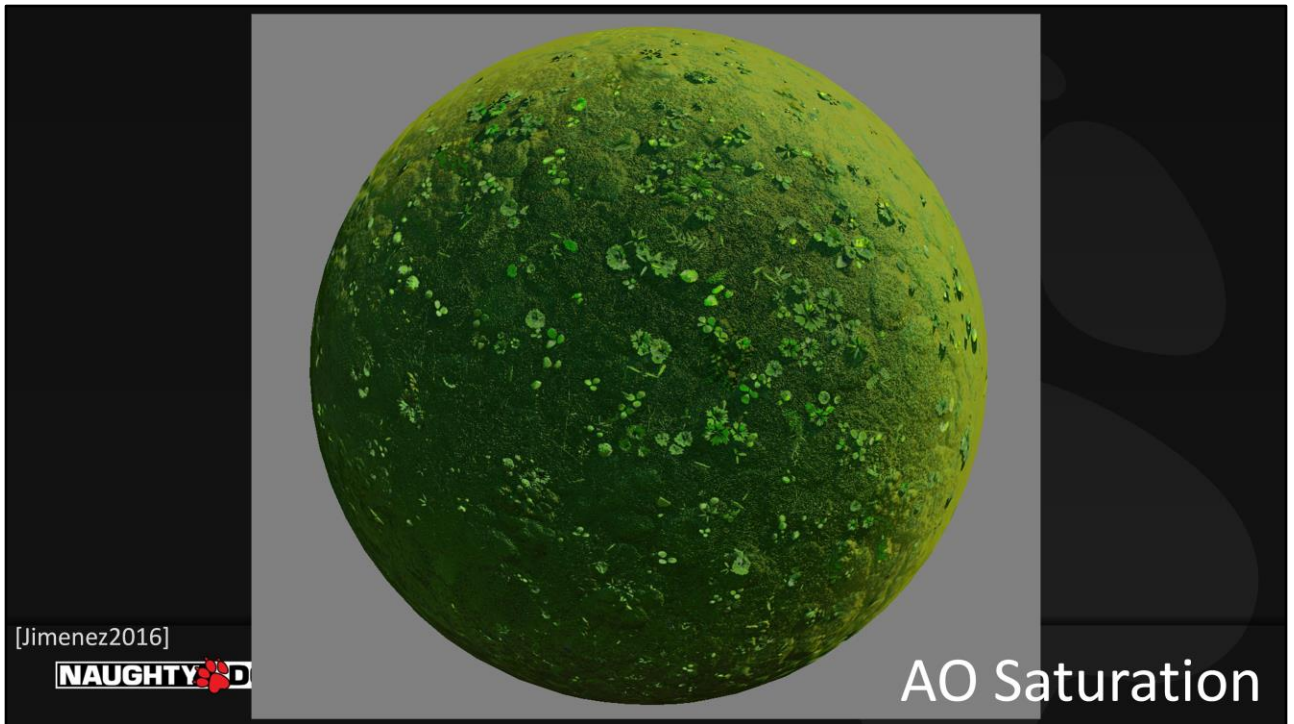
We have a Fuzz BRDF feature, which is the subpixel, “microfiber” version of this whole phenomenon. The artist has the ability to specify a separate color for the fuzz and we render it as a microfiber layer on top of a microfacet layer.



For some cases where we can afford it, we apply parallax shadows. (Thanks Natalya!)



We apply AO. For moss, the AO goes very dark. Two features we discussed earlier are very critical here – the AO fades to white at a glancing angle, and our Sun Micro Shadowing is applied.

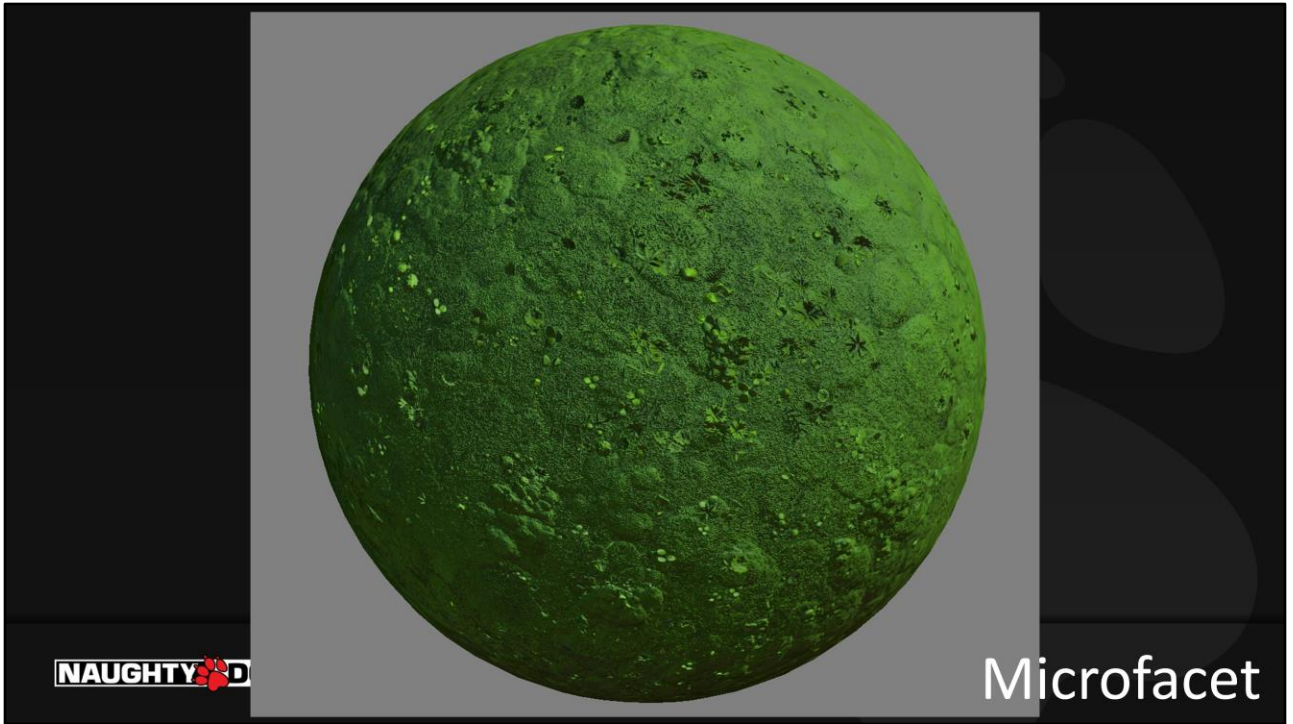


[Jimenez2016]

NAUGHTY D

AO Saturation

Finally, we add back a bit of color where the ambient occlusion gets dark. This simulates the effects of bounce light and subsurface scattering. Our code was a simple hack, but refer to the talk by Jorge Jimenez for a really good physically-based solution to this problem.



And here's the microfacet BRDF again for comparison.

```

float3 ApplyLightWrap(float3 lightWrapColor, float3 normalWS, float3 vertexNormalWS, float3 lightDirWS)
{
    float lightWrapDistance = 0.1;
    float3 wrapLight = lightWrapDistance * lightWrapColor;
    float NdotL      = dot(normalWS, lightDirWS);
    NdotL            = lerp(max(wrapLight.r, max(wrapLight.g, wrapLight.b)), 1.0, NdotL);

    float wrapForwardNdotL = max(NdotL, dot(vertexNormalWS, lightDirWS))
    float3 wrapForward     = lerp( wrapLight, float3(1.0, 1.0, 1.0), wrapForwardNdotL);
    float3 wrapRecede      = lerp(-wrapLight, float3(1.0, 1.0, 1.0), NdotL);
    float3 wrapLighting    = saturate(lerp(wrapRecede, wrapForward, lightWrapColor));

    return wrapLighting;
}

```



Here's our light wrap code. It's somewhat energy conserving, in that it broadens the terminator without just adding energy. Also note, the only artist control is a color – as the color approaches 0, the whole system converges to NdotL.

I won't show the fuzz code, because it got kind of hacky. Pick your favorite microfiber/fabric brdf and give it some user-friendly controls. Easy!



Next Steps...

Many surfaces in the world exhibit this “fuzzy” mossy behavior. From fabric to distant foliage, we see similar behavior. Even subtle phenomena like dust or peach fuzz have these same fuzzy characteristics. I want to try encapsulating all of the above techniques into a standardized interface of our artists, make them into a single brdf feature.

Special-Case Materials

Wetness





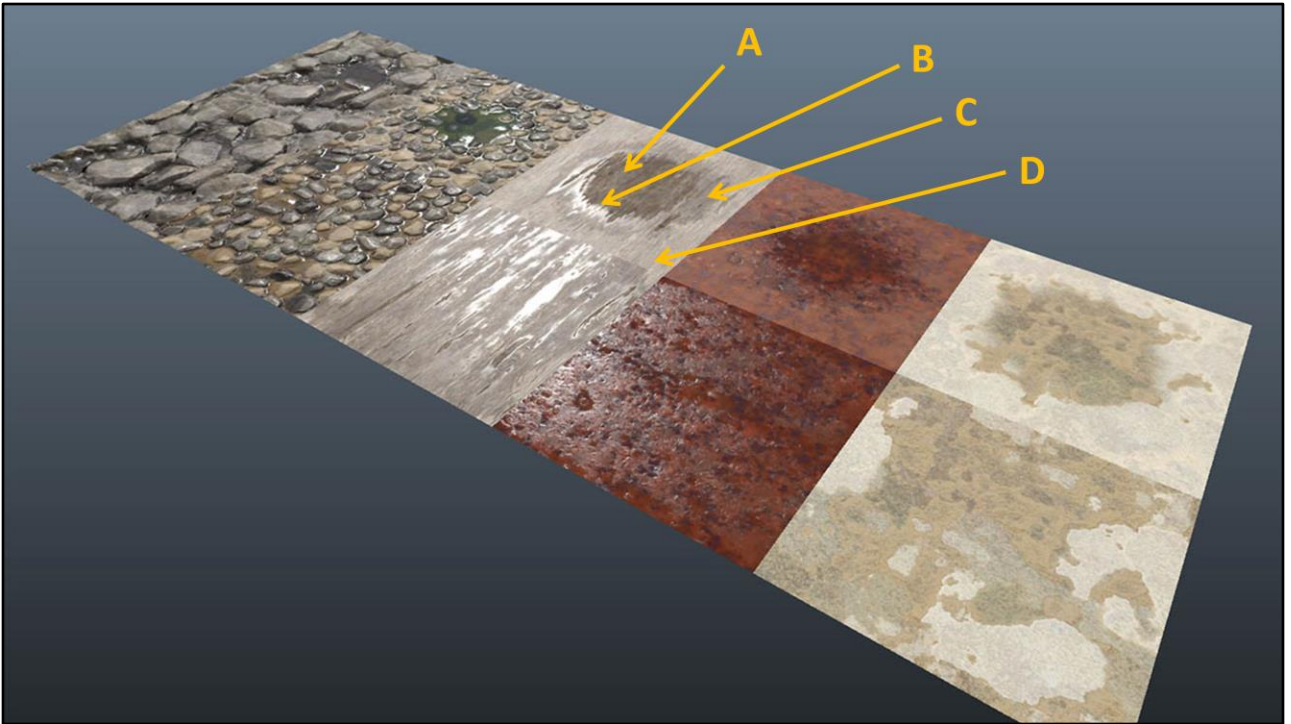
For Uncharted 4, we wanted a unified wetness function. Here's an example of a shipped scene. We're not the first ones to tackle this problem, but I'm happy with our solution, so I thought I'd share.



... And the same scene with wetness turned off.



And here's a visualization of the wetness mask the artists painted. Note – for some scenes we wanted the ability to drive wetness dynamically via gameplay. (like when a rain storm starts.) But for most of the game we just wanted visual consistency, physical correctness, and an intuitive interface for the artists.



Here are a few more examples. Note how the puddles show four characteristic regions. A) Core of the puddle, the water surface is totally flat. B) Region where surface tension causes water to cling to the surface underneath, causing a shrink-wrapped look. C) Region where water has saturated the surface, causing a darkening of the albedo, but not significantly affecting normals or specular response. D) Dry, unmodified surface.

```

float ClampRange(float input, float minimum, float maximum)
{
    return saturate((input - minimum) / (maximum - minimum));
}
...
float3 baseColorSqr = baseColor*baseColor;
baseColor          = lerp(baseColor, baseColorSqr, ClampRange(wetness, 0.0, 0.35) * porosity);
roughness          = lerp(roughness, 0.1,          ClampRange(wetness, 0.2, 1.0));
specular           = lerp(specular, 0.25,          ClampRange(wetness, 0.25, 0.5));
ao                 = lerp(ao, 1.0,                  ClampRange(wetness, 0.45, 0.95));
normalTS           = lerp(normalTS, float2(0.5),    ClampRange(wetness, 0.45, 0.95));
vertexNormalWS     = normalize(lerp(vertexNormalWS, float3(0,1,0), ClampRange(wetness, 0.98, 1.0)));

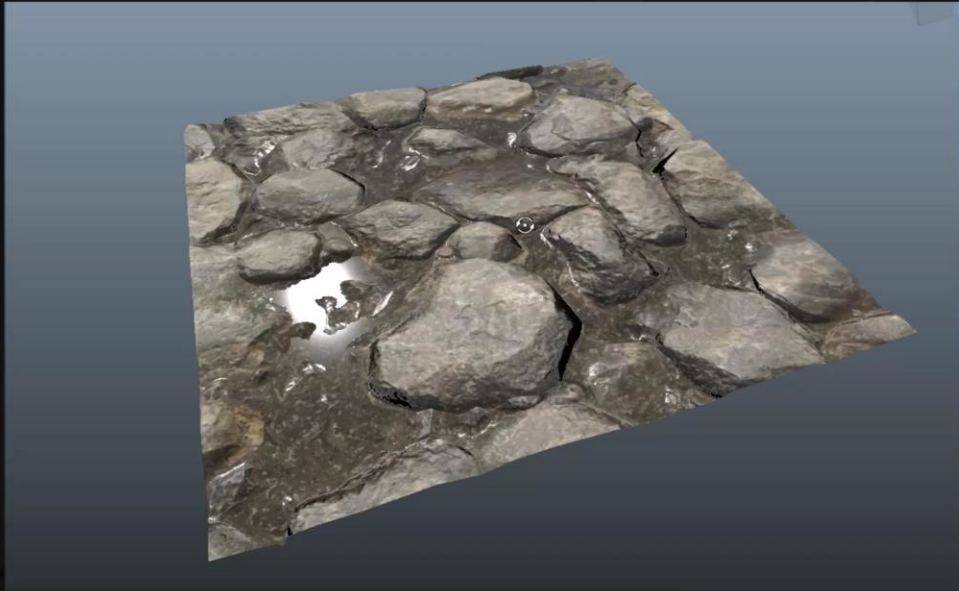
```

Artist Inputs: Wetness, Porosity

See also: [Lagarde2013]



And here's the code. The artists have control of two parameters, "Wetness" and "Porosity", which can be blended independently. Wetness drives the transition between the different phases explained on the previous slide. Porosity is an optional control (if it's omitted, Roughness is used instead). Porosity affects darkening of the base color – surfaces that are more porous darken more, surfaces that are less porous, more "hard", darken less. Note that we just lerp to a "squared" version of the base color – compared to real-world reference, this is a reasonable approximation of the darkening that happens when a surface gets wet.



NAUGHTY DOG

Here it is in action – you can see, for any input values of the mask, it gives meaningful wetness results, so it's very intuitive to use. Note, when wetness strength gets above a value of 98%, we force the vertex normal to point straight up, which does a very convincing job of making the surface of the pool feel flat even if the underlying geometry isn't flat.



Our artists used this feature in nearly every scene. When reinforced by lighting and fx, it really helps sell the look.

Special-Case Materials

Glass





We had a variety of glass surfaces on Uncharted 4. We were able to achieve all of them with a fairly minimal set of tools.



Our simplest case of glass is windows. Technically there should be a refractive component, but it's cheaper (and less artifact-prone) to just do alpha blend.

```
finalLight = diffuseLight + (specularLight / max(visibility, 0.01));
```



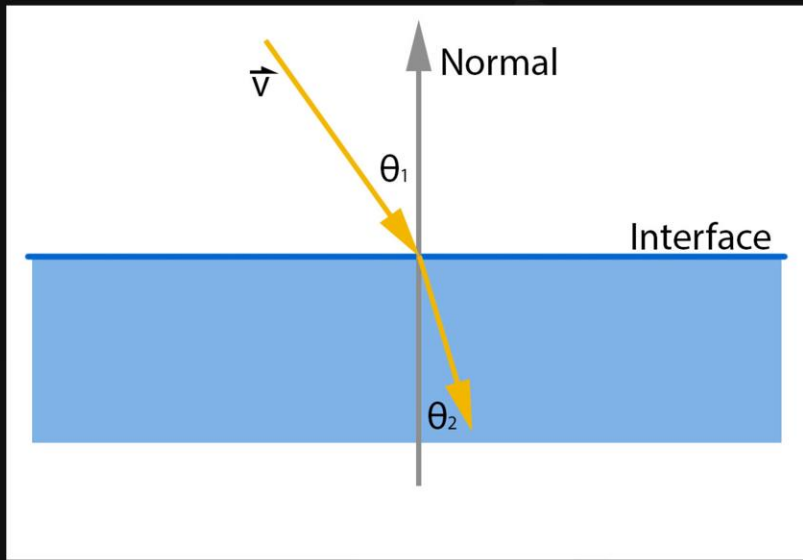
When you think of glass as “transparent”, it’s really the diffuse component that’s transparent – the specular component is identical to any other hard surface. So if you’re doing standard alpha blending you can pre-divide out the alpha channel, and end up with a result where only the diffuse is affected by the transparency. The surface can have varying transparency (say, if the window is grimy), and the specular result should appear constant across the surface. The only issue is floating point inaccuracy as transparency approaches 0.

Note, if you perform premultiplied alpha, this problem becomes much simpler – only apply the transparency to the diffuse, not to the specular!



For solid glass (or ice) surfaces, we incorporate refraction. This follows the same rule as before, where only the diffuse component becomes transparent and the specular is preserved. But this time we blend against the screen buffer in the shader – we use the transparency map to lerp our diffuse component against the pixels refracted from behind the object.

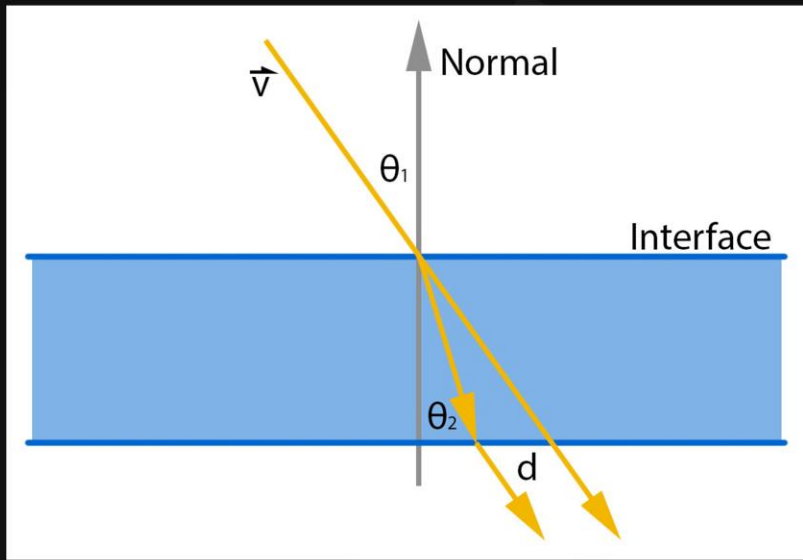
For more information on this subject, refer to Morgan McGuire's talk.



```
float3 refractWS = refract(viewWS, normalWS, ior);
```

NAUGHTY DOG

We can use Snell's Law to calculate how the view vector is bent when it transitions from air into a refractive surface. (using the intrinsic function "refract()").



NAUGHTY DOG

But the adjusted ray isn't enough. At some point the ray will exit the material. In the example above, the ray has been offset by a distance d . We can use that as an offset into the Screen UV coordinates. Note that d increases as v becomes perpendicular to the surface, and as the thickness of the surface increases.



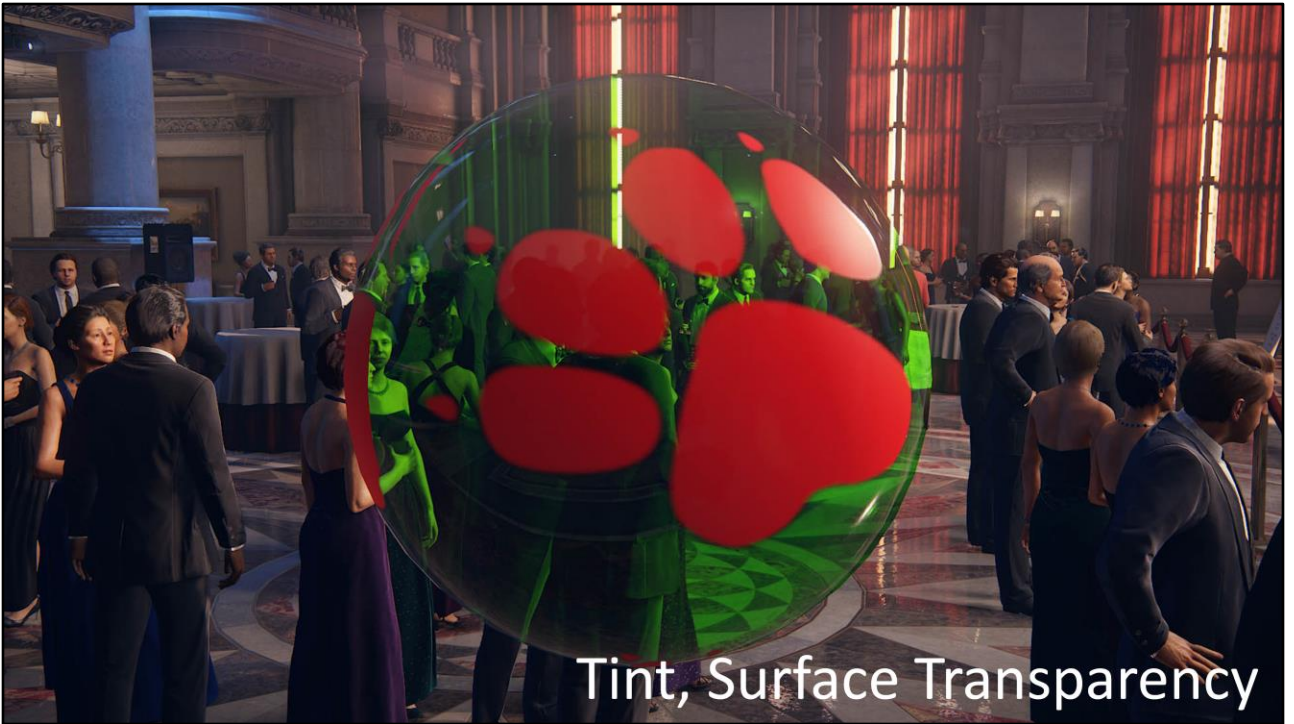
NAUGHTY DOG

Thin Shell vs Solid

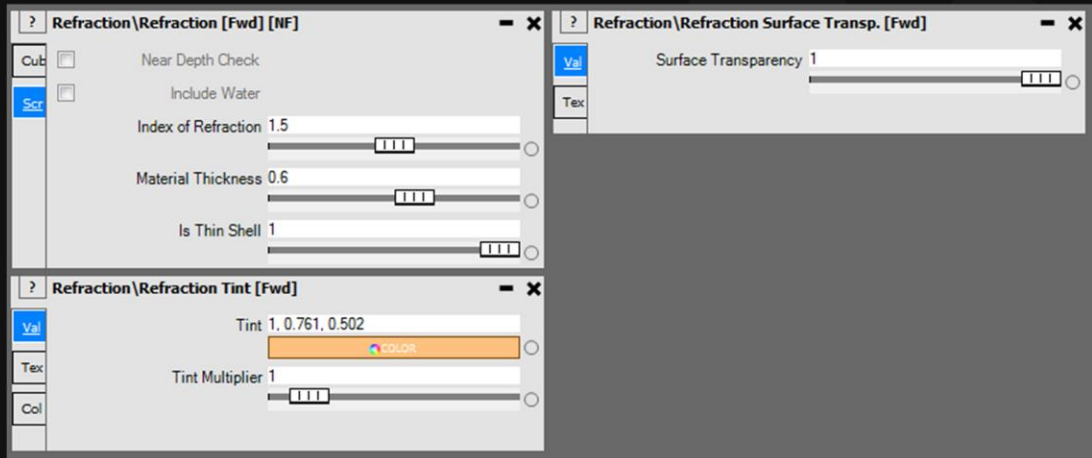
Like any good physics problem, we can simplify things by treating the surface as a perfect sphere. We have two major classes of refractive surfaces – “thin shell”, which represents a hollow sphere like a Christmas ornament, or “solid”, like a crystal ball.



Here's an example - observe that we used "thin shell" refraction for the empty part of the tumbler, and "solid" refraction for the base.



We also have parameters for the medium to have a tint, like for colored glass, and for the surface diffuse to be opaque, which is useful for labels or grime.



Artist Interface

The user interface is comparatively simple. Here are all the controls we've discussed...

```

float NdotV = dot(normalWS, viewWS);
float3 refractWS = refract(viewWS, normalWS, ior);

// Calculate distance ray travels through the medium
float rayLengthSolid = thickness;
float rayLengthShell = thickness / max(pixelDepth * NdotV, 0.5);
float rayLength = lerp(rayLengthSolid, rayLengthShell, isShell);

float3 refractVectorWS = refractWS * rayLength;
// Convert world-space to view space, add to current screen UV, sample screen buffer
float3 refractionLight = FetchRefractedPixel(refractVectorWS);

// Final color based on Fresnel darkening and remaining user inputs
refractionLight *= mediumColor * SchlickFresnel(NdotV, ior);
diffuseLight = lerp(diffuseLight, refractionLight, surfaceTransparency);

```



And here's the code that's driven by those parameters. Note our approximation: for solid objects the ray travels a uniform distance, for thin-shell objects the distance increases based on glancing angle.



This ice sculpture is a good example of all these parameters in action. It's treated as a "solid" material type. It has a slight blue tint, and varying thickness (the body is much thicker than the arm, for example.) And the artist used the Surface Transparency control to blend in some frost, applying it with a Fresnel falloff, only to the upward-facing surfaces.



We only support one layer of refraction that reads from the screen buffer. For the wine inside of these glasses, instead of fetching a pixel from the screen buffer, we keep the refraction vector in world space, and fetch the refraction color from the nearest cubemap (the same one that supplies the specular reflections.) The artistic controls for this type of refraction are the same.



In the end, the results were good enough, but can always be better. The main issues we encountered were sort-order artifacts, and transparent objects receiving the correct cubemap from the environment. (Particularly an issue when a glass window can be seen from inside and out.) For the refraction tech we want to try simulating frosted glass by multi-sampling the screen buffer, possibly with temporal jitter. But having a physically-plausible basis is a good base to build on.

And now...

NAUGHTY DOG

Video Link in Notes

GENERAL-PURPOSE VERTEX PROCESSING

IN

UNCHARTED **4** *A Thief's End*



Please follow this link to find the video:

<https://www.youtube.com/watch?v=LjCjXFmkX-4>

General Purpose Vertex Processing

Shaders are not just for shading:

- Globally Integrated Wind System
- Foliage, Hair, Cloth Simulation
- Baked-pivot Billboards/Imposters
- Morph Targets
- Procedural animation
- And lots-lots more....

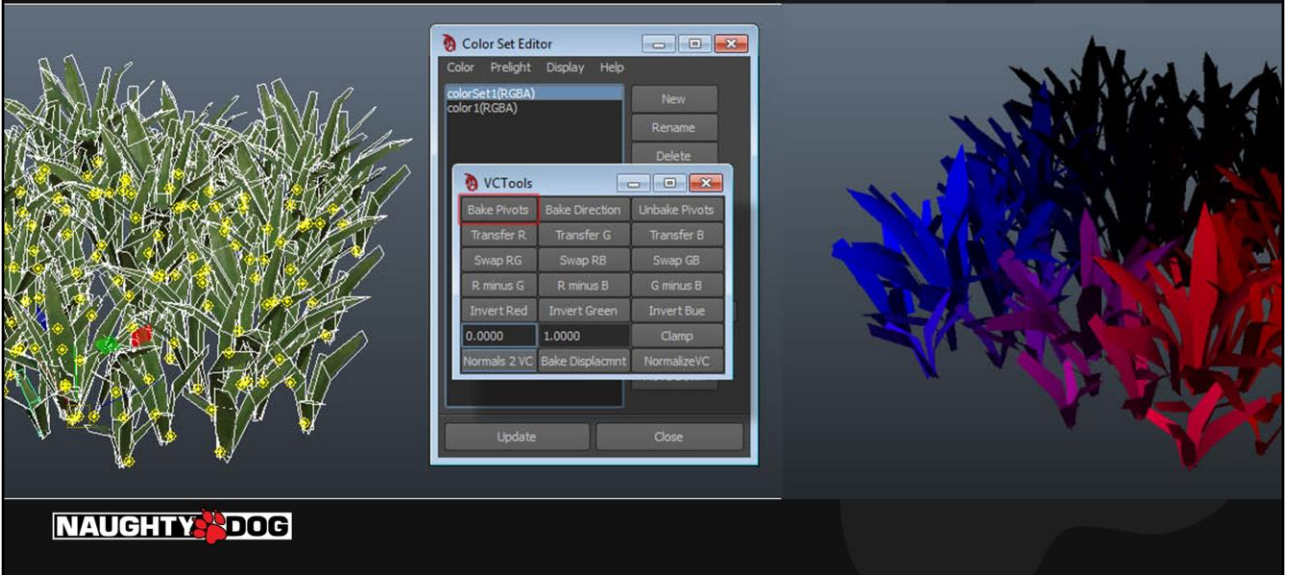


There are a lot of vertex processing problems to be solved in a game production. On Uncharted 4 we were able to take this approach much further than we initially anticipated and did our best to keep it very generalized and accessible.



But first things first - the original problem we were trying to solve was foliage animation. We had a ton of it and needed a simple and cheap way to get everything on screen moving.

Pivot Data



We wanted a more physically accurate implementation so we went for pivot-based rotation. Pivot positions were stored in vertex colors.

Generalize Vertex Data

Vertex Colors => General Purpose Float3 Data

UV Coordinates => General Purpose Float2 Data

- Unclamped, Signed

Eventually just generic data streams



This solidified the mental shift we had for treating vertex attributes as just general purpose data.

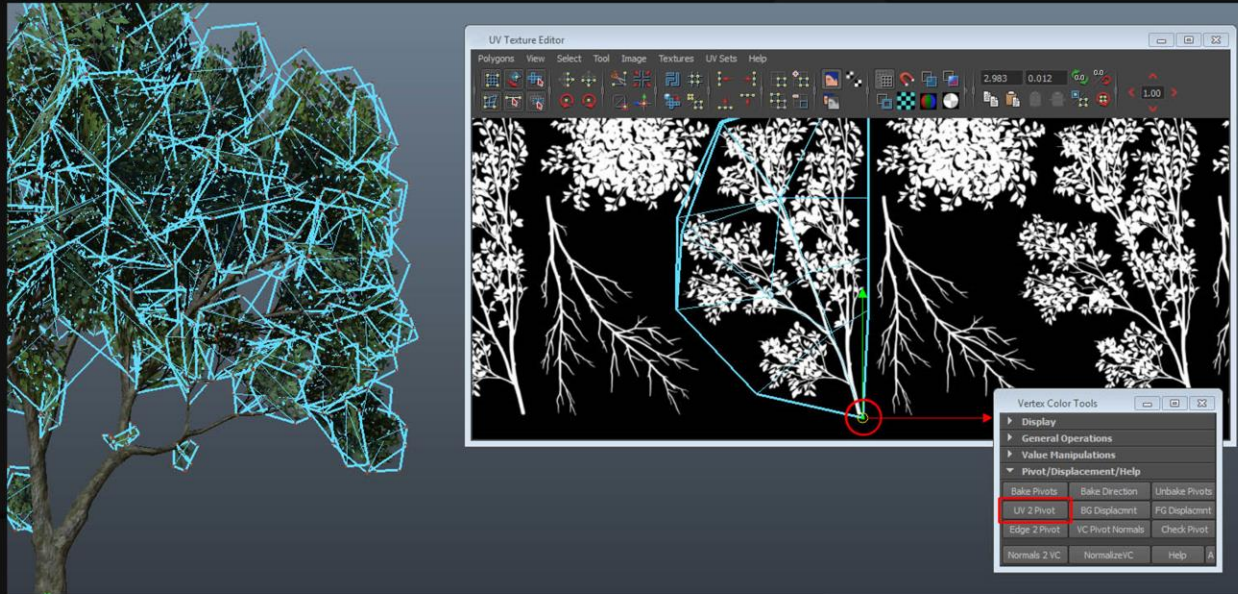
Try this in Maya:

```
def bakePivots():  
    selection = cmds.ls(sl=1)  
    for object in selection:  
        pivot = cmds.xform( object, query=True, pivots=True, worldSpace=True )  
        cmds.polyColorPerVertex( object, rgb=( pivot[0], pivot[1], pivot[2] ) )
```



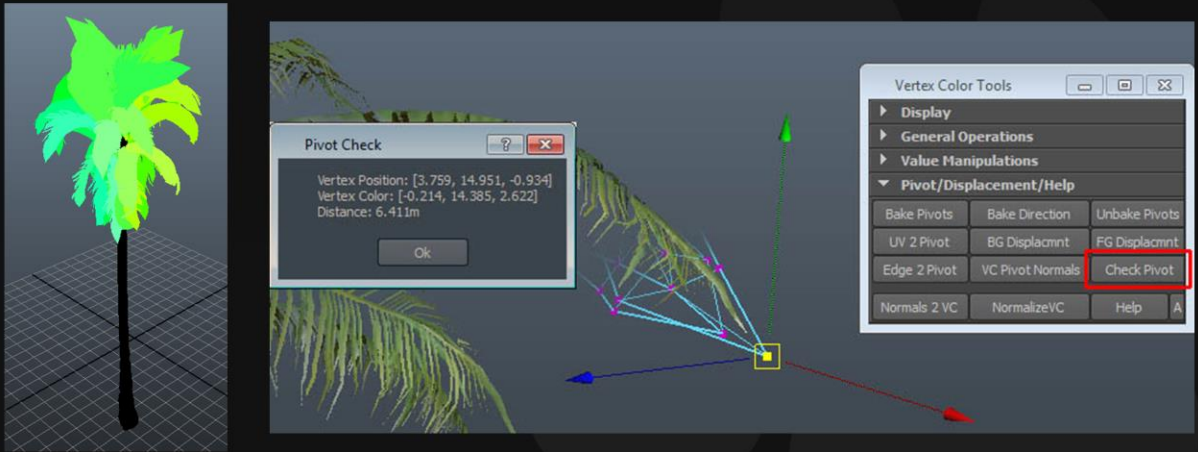
Maya does not have a very robust vertex color scripting pipeline so for most operations we had to write our own functionality using Maya API. Nevertheless you can use this snippet to get going with baking pivots in maya.

Improved Vertex Color Pivot Authoring



Eventually we came up with simplified ways to author pivot data. Like using a UV-shell based approach.

Invest in debug tools



NAUGHTY DOG

The data is not very visual so it can get confusing for artists. Consider debug tools that allow them to check their pivot data in their authoring tool. We had full animation preview in Maya but had to sacrifice that in Viewport 2.0.

Wind System Hierarchy

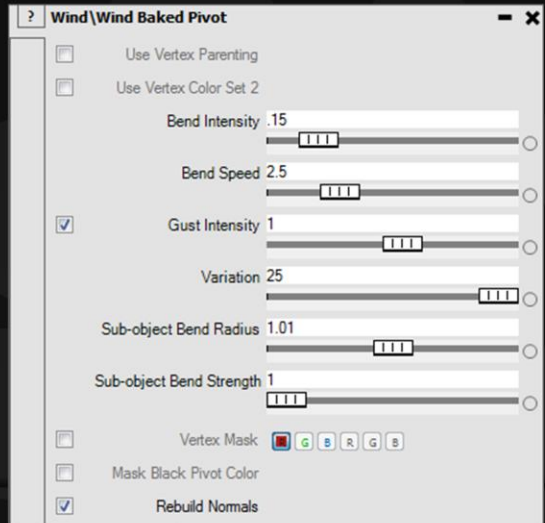
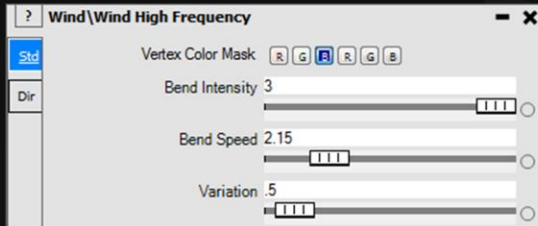
- Object Pivot
- Baked Pivot 1
- Baked Pivot 2
- Normal-based High Frequency Movement



NAUGHTY DOG

The order of operations for vertex shader rotations effectively allowed us to do hierarchical animation without having to store any hierarchical data. Just pivots. We also found it necessary for very high-frequency wind motion to have a classic vertex normal based animation on tips of leaves and fronds.

Wind Shader Features



Minimal amount of parameters is imperative for any efficient system. 2 main parameters we used were “Bend Intensity” – Max Rotation Angle and “Bend Speed” – which was the Speed of Animation. Sub-object Bend parameters are inputs for our time offset calculation explained on page 93-94. Variation was a per sub-object randomization multiplier.

Wind Animation: Ambient + Gust

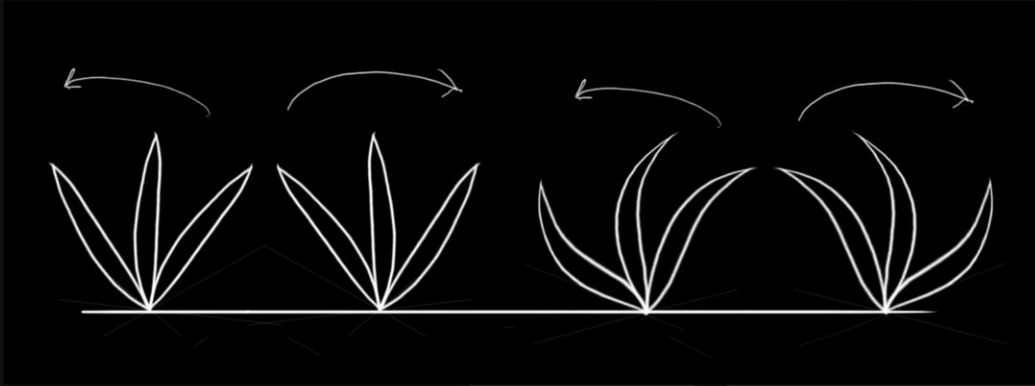
```
float noise = Get2dNoise(dot(windDirection, pivotWS), time);  
float animation = sin(windSpeed * time + timeOffset);  
float angle = windIntensity * animation * noise + gust;
```



While physically-based, getting the angle function required some iteration and magic art numbers to get a satisfactory animation with minimal amount of instructions.

Vertex Time Offset?

Fake Inertia – Very Important for organic look:



NAUGHTY DOG

“timeOffset” variable from the previous slide allows us to have some vertices animating further in time. If we modulate that by distance to pivot we get a nice swaying motion that is Crucial for achieving an organic look.

Update Normals!

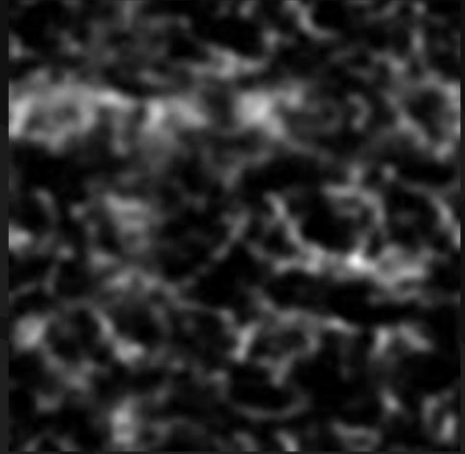


Pivot based rotation allows for exceptionally easy vertex normal updating (use the same axis and angle as for vertex rotation, but origin([0,0,0]) for the pivot). It was optional in our first implementation but we are definitely going to hard code it "On" going forward.

Global Wind Parameters

name	value	category	type
Wind_Direction	1.0, 1.0, 1.0	Wind	Direction
Wind_Gust_Intensity	1.0	Wind	Float
Wind_Gust_Speed	0.05	Wind	Float
Wind_Intensity	1.0	Wind	Float
Wind_Speed	1.0	Wind	Float

Sampled in every wind type:
Baked Pivot, Object Pivot, Cloth, Hair,
UV Wind, etc...



NAUGHTY DOG

All of the parameters up to this point were on a per-asset level. We had a set of similar level-wide parameters that would control the wind globally. All of our foliage was setup in a neutral wind environment with the entire foliage library side by side for reference. That way we made sure we are getting consistent wind motion across the board.

Apart from our wind parameters we had our global 2d Wind Gust function which was in essence a wave-pattern texture projected top-down into the world. It was sampled in every wind type creating consistent motion across varied surfaces and objects.



First pass player interaction was created by rotating a sub-object away from a character based on proximity.

Player Push



But we didn't have to do just rotation. Pushing objects away based on pivot distance was a viable option and helped us get interactivity fast and cheap where we wouldn't have been able to otherwise.



Notable, most of these interactions are also possible in the Pixel Shader. This is an example of UV based wind and player interaction.

Improving Interactivity

Particle World Influence Render Target:



NAUGHTY DOG

To extend the functionality and allow for persistence of data we implemented a Particle Influence render target that stored direction and intensity of rotation and could be written into with particles. That allowed us to have an unlimited amount of character and vehicle interactions for the constant cost of sampling this 2d buffer for every vertex. It was a good tradeoff.



Vehicle Foliage Interaction

Having vehicle press down foliage was an important problem to solve since we have a lot of vehicular gameplay.

Particle Interaction With Shaders:

- Foliage History
- Terrain Deformation
- Footstep Material Blend
- StepMorph persistence, etc...



NAUGHTY DOG

Adding a generic communication pipeline between FX and Shaders allowed us other fun uses, like characters masking certain material layers from the objects underneath.

Pivots on Skinned Meshes?

Skinned Vertices – animate in local-space invalidating baked pivot data.



NAUGHTY DOG

Porting the same logic to skinned meshes wasn't quite trivial though 😊

Pivots on Skinned Meshes

Solution:

- expose LS position
of a vertex to
another vertex.

“Parent Vertices”



NAUGHTY DOG

As a solution, for every hair card we chose one pivot vertex and made it's post-skinning position available in the shader for every vertex in the common shell.

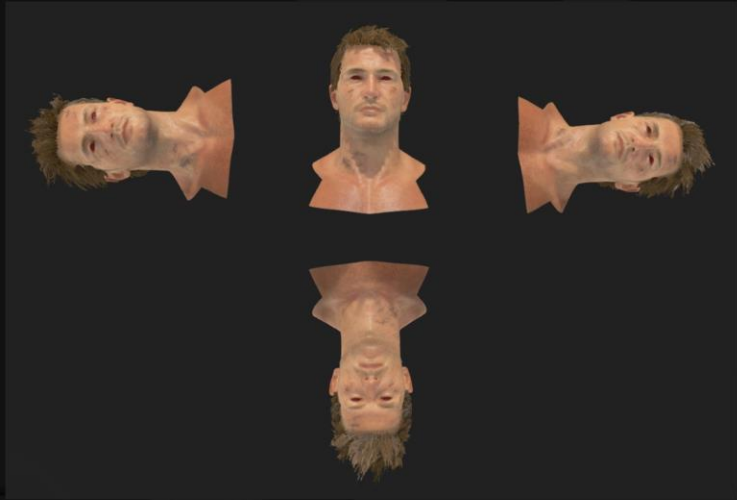
Hair Wind



Here's an example of the per hair card movement.

Direction on Skinned Meshes

Needed generic WS direction for skinned meshes for all kinds of directional effects.

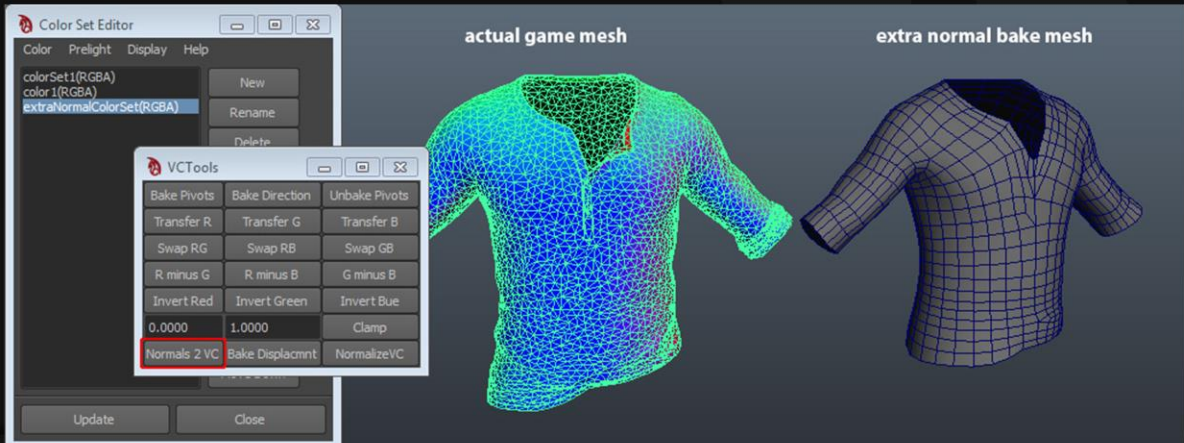


NAUGHTY DOG

But we also wanted directional data. For example to limit hair wind to just the side of the head that is directly facing wind direction. But there was no proper generalized direction data, as vertex normals in this situation are all over the place.

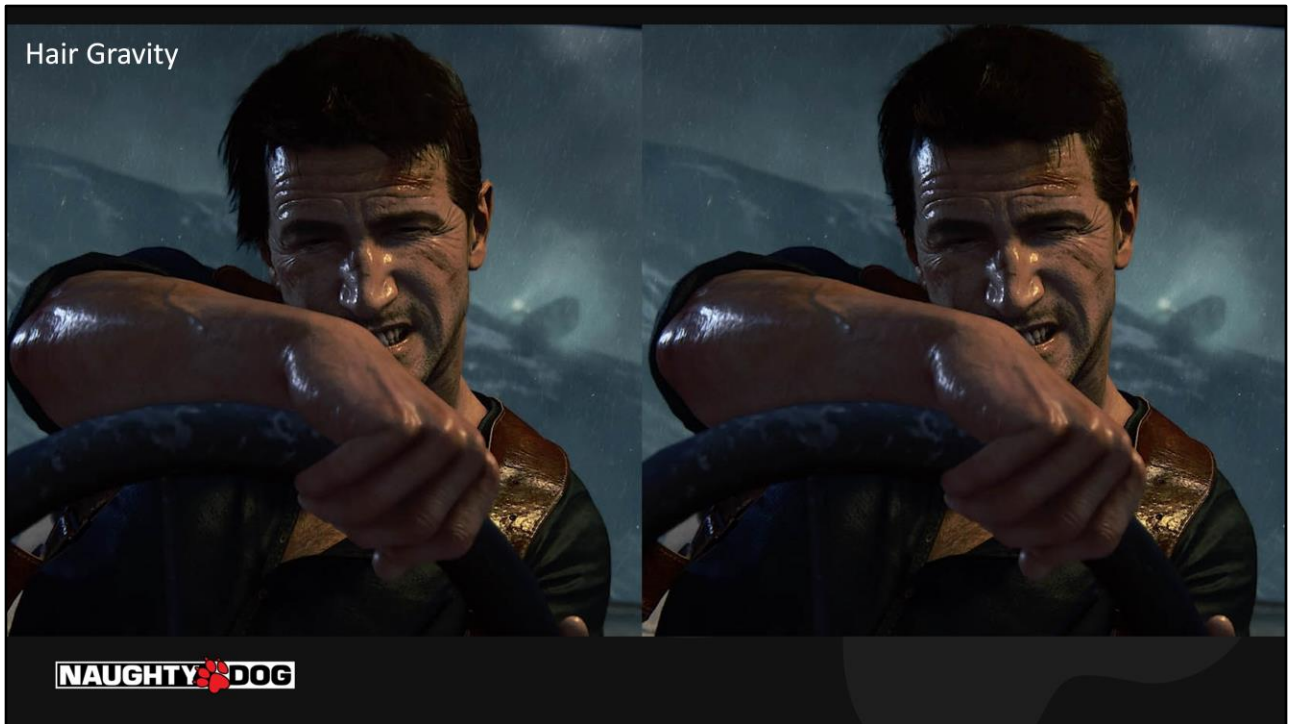
Direction on Skinned Meshes

- Extra Normal Color Set



NAUGHTY DOG

Our solution was an additional vertex color set that would store generalized directional information and would be pushed through the normal skinning compute. Basically a generic extra normal set.



Another use of that directional data was the procedural hair gravity feature we used to simulate Drake's wet hair. It relied on the same inputs as the wind which made it quite painless to implement.

Hair Gravity code

```
float3 directionWS = LocalToWorld(vertexColors3);

//check if vertex is "facing" down
float directionMask = 1.0 - saturate(-directionWS.y) + directionMaskFill);

//check how far the vertex is from the root
float rootMask = (1.0 - SphereMask(pivotWS, positionWS, hairGravityRadius));
float offset = saturate(Pow2(rootMask * directionMask * gravityIntensity));

positionWS.y -= offset;
```



As you can see we didn't actually rotate the hair cards but rather pushed the verts that are furthest from the pivot down in world-space. It was a simpler solution and the stretching wasn't noticeable.



Another use-case of the directional info was the shader cloth wind.



Using the skinned extra normal set and Wind direction we could derive a mask for the part of the shirt that would have cloth wind ripples. Then we projected our 2d wave function from both X and Z axes and animated it scrolling down in the vertex shader. That gave us the silhouette change but normal update was a very big part of the effect. So we just added a normal map of the same wave pattern and blended that in the pixel shader.

Morph Target Wind



An additional feature of our wind system is blending in a vertex color morph target based on global wind gust.



That way we combine procedural pivot and normal based wind simulation with 2 discrete “keyframes” for extreme intensities.

Step Morph



Another feature to come out of this was StepMorph™. We would blend to a morph target based on player distance to sub-object pivot to create more response to the player in the world.

Step Morph

```
float3 displacement = LocalToWorld(vertexColors1);  
float3 pivotWS     = LocalToWorld(vertexColors2);  
float  distanceMask = SphereMask(playerPosition, pivotWS, footstepMorphRadius);  
float  blend       = saturate(distanceMask * footstepMorphIntensity);  
positionWS        = lerp(positionWS, displacement, blend);
```



As you can see the code is extremely simple. Particle World Influence render target was a way to provide persistence to this technology as well.

Skinned Morph Target



As you've noticed by now all the features described in this presentation take already existing technology and add just enough on top to ungate new and exciting opportunities. For this particular case we needed a good vehicle damage model that was fast and quick to implement. So we decided to take our skinned vertex color channel and use it as a morph target. Since our Skinning Compute would normalize normal data we had to store the magnitude of the transformation as a float attribute in a separate vertex color channel.

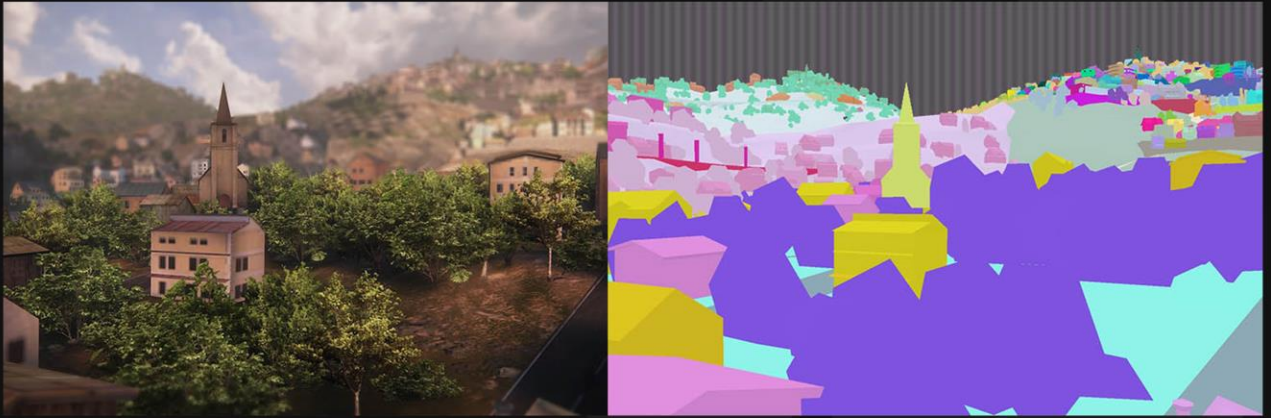


Skinned Morph Target

Then all we had to do was to have our vehicle damage particles write into our Particle Influence render target and read that in the shader to reveal the damaged morph target.

Pivot Billboards/Imposters

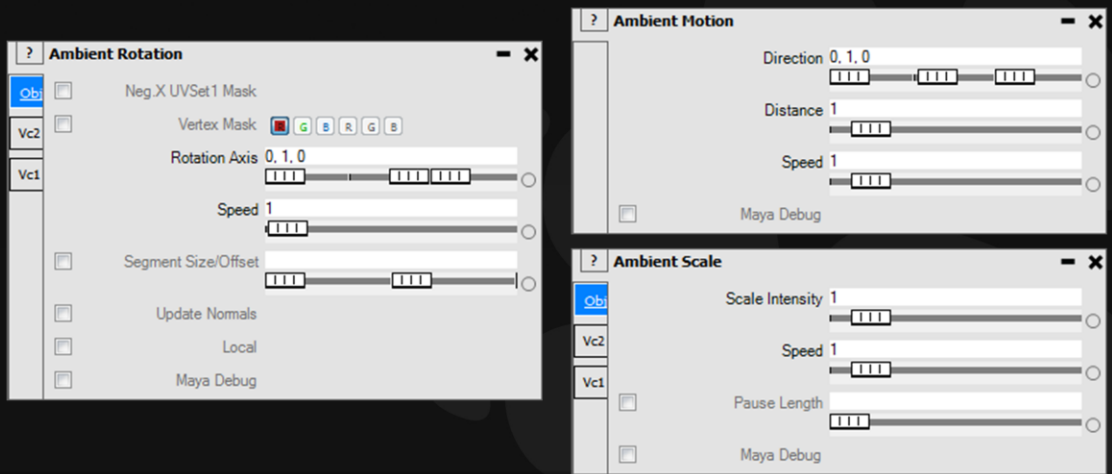
One Drawcall - a lot of objects



NAUGHTY DOG

Another big optimization for us was Pivot Billboarding/Imposters. Since we could get cards to rotate around a baked pivot instead of object pivot, we could attach them all together saving hundreds of drawcall submissions for all of our mid and wide camera-facing cards.

Ambient Animation



Another thing we did was a generalized ambient animation pipeline. Our artists had access to looping motion, rotation and scale animations that could be combined in all kinds of ways.



For example this washing machine animation is a small segment rotation around object pivot + subtle motion back and forth across the floor.



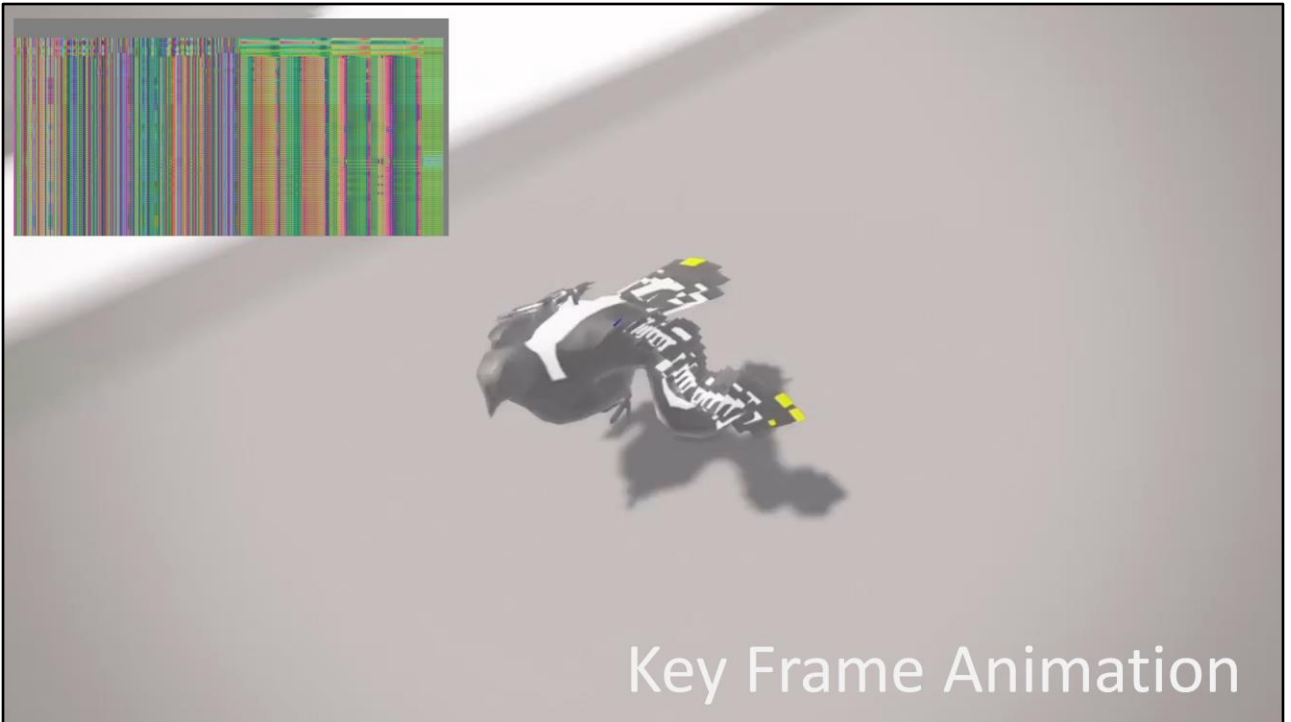
The bobbing boats effect was created using up-down movement with some left-right rotation.



Geometry rain ripples were using our ambient scale to appear and disappear. We had to write a raycast solution to auto populate them. We stored their positions as a collection matrixes in a binary so that we could load and control their density at runtime.



We even had a prototype of vertex shader swarm simulation. Each of the flies is a camera facing card but all of them together are just one object. They orient themselves according to their velocity and as you can see in the video are pretty controllable.



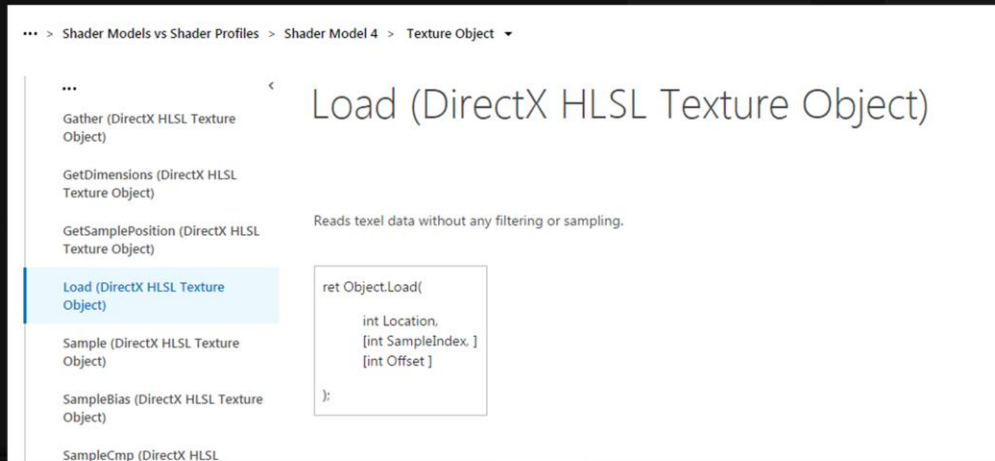
Next fun thing we did in vertex shader was keyframe animation. This tech was spearheaded by our great FX artists Raymond Popka and Lauren Simpson.



We used it for particle crowds and distant particle animals. This tech also allows us to update normals while storing them in a separate keyframe pixel row.

Key Frame Animation

Use “Load”



The screenshot shows a documentation page for the 'Load' function in DirectX HLSL. The breadcrumb trail at the top reads: '... > Shader Models vs Shader Profiles > Shader Model 4 > Texture Object'. The main heading is 'Load (DirectX HLSL Texture Object)'. Below the heading, there is a list of functions on the left: 'Gather (DirectX HLSL Texture Object)', 'GetDimensions (DirectX HLSL Texture Object)', 'GetSamplePosition (DirectX HLSL Texture Object)', 'Load (DirectX HLSL Texture Object)' (highlighted in blue), 'Sample (DirectX HLSL Texture Object)', 'SampleBias (DirectX HLSL Texture Object)', and 'SampleCmp (DirectX HLSL...'. To the right of the 'Load' function, there is a description: 'Reads texel data without any filtering or sampling.' Below the description is a code block showing the function signature:

```
ret Object.Load(  
    int Location,  
    [int SampleIndex, ]  
    [int Offset ]  
);
```



Definitely use the Load function to sample your texture to avoid filtering or sampling messing up your animation.



This tech paid off bigtime in this particular implementation. For each particle flipbook we create a keyframe texture that has local space bounds of the visible pixels in each frame. That allows us to modify FX quads in vertex shader to occupy only the space currently needed by the flipbook frame. Thus dramatically reducing fx overdraw and pixel shader pressure. As an additional pass we added UV positions to the keyframe texture and were able to repack our flipbooks to half the size with minimal to no resolution loss. The entire thing took just a couple of days to implement.

Vertex Shader Population



We also experimented with vertex shader based population of assets. Which wasn't really population. We had a single mesh that we would wrap around the player position on a sub-object level based on his distance from WS sub object pivot.

Vertex Shader Population



Great effect for a magic druid game!

Performance

- ~9000 Animated Submeshes
- 1ms on GPU



Not as expensive as we thought. Still could be made cheaper by moving to compute.

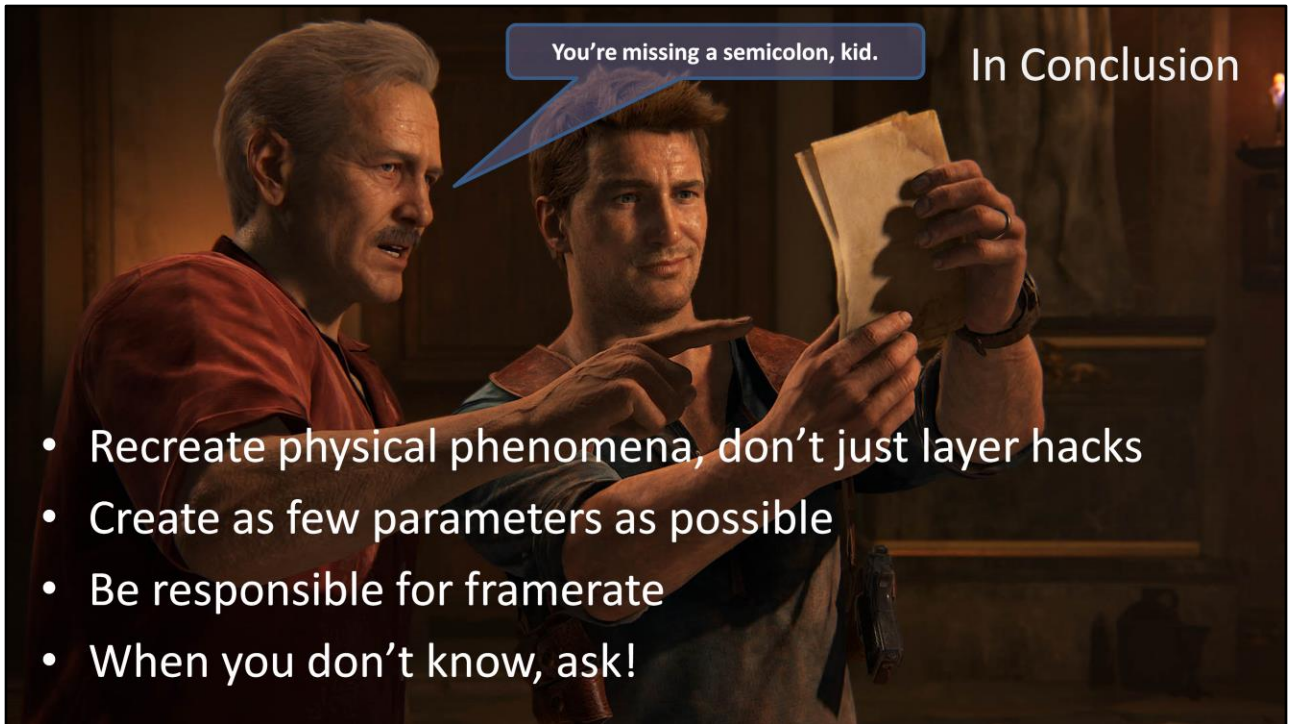
Future Research

- Compute Shader
- Scriptable Triggers
- Curves as Artistic Inputs

Future-Future

Per-instance data available today already pushes outside current concept of Instance.

A more generic mesh processing step as a first-class hardware citizen seems likely and useful.



- Recreate physical phenomena, don't just layer hacks
- Create as few parameters as possible
- Be responsible for framerate
- When you don't know, ask!

Hopefully you found it useful seeing these examples of our work. Looking back, there were a few common philosophies that made our work successful. By following these in all of our work, the technical artists at Naughty Dog were able to contribute a significant number of features that were key to the look of Uncharted 4.

: At the end of the day, the game has to be in frame. If you write inefficient code, then somewhere down the line, art will have to be cut.

Naughty Dog is Hiring!

<http://www.naughtydog.com/work>
jobs@naughtydog.com



Acknowledgements

- Vincent Marxen, Ramy El-Garawany, Ke Xu, Marshall Robin, Wang Fengquan, Carlos Gonzalez-Ochoa, Pal Engstad, Jerome Durand, Teagan Morrison, Raymond Popka, Lauren Simpson
- For help with the talk: Mark Shoaf, Joe Pettinati, Grant Voegtle, and to Mikki Rose and Gracie Arenas Strittmatter for coordinating this talk!



References

- [Toksvig2005] – Frequency Domain Normal Map Filtering
- [Tatarchuk2006] – Practical Parallax Occlusion Mapping for Highly Detailed Surface Rendering
- [Hable2010] – Filmic Tonemapping Operators
- [Penner2011] – Pre-integrated Skin Shading
- [Burley2012] – Physically Based Shading at Disney
- [Lagarde2013] – The Art and Rendering of Remember Me
- [McAuley2013] – Extension to Energy-Conserving Wrapped Diffuse
- [Jimenez2016] – Practical Real-Time Strategies for Accurate Indirect Occlusion
- [McGuire2016] – Peering Through a Glass, Darkly at the Future of Real-Time Transparency



Contact Us!

waylon_brinck@naughtydog.com

andrew_maximov@naughtydog.com

