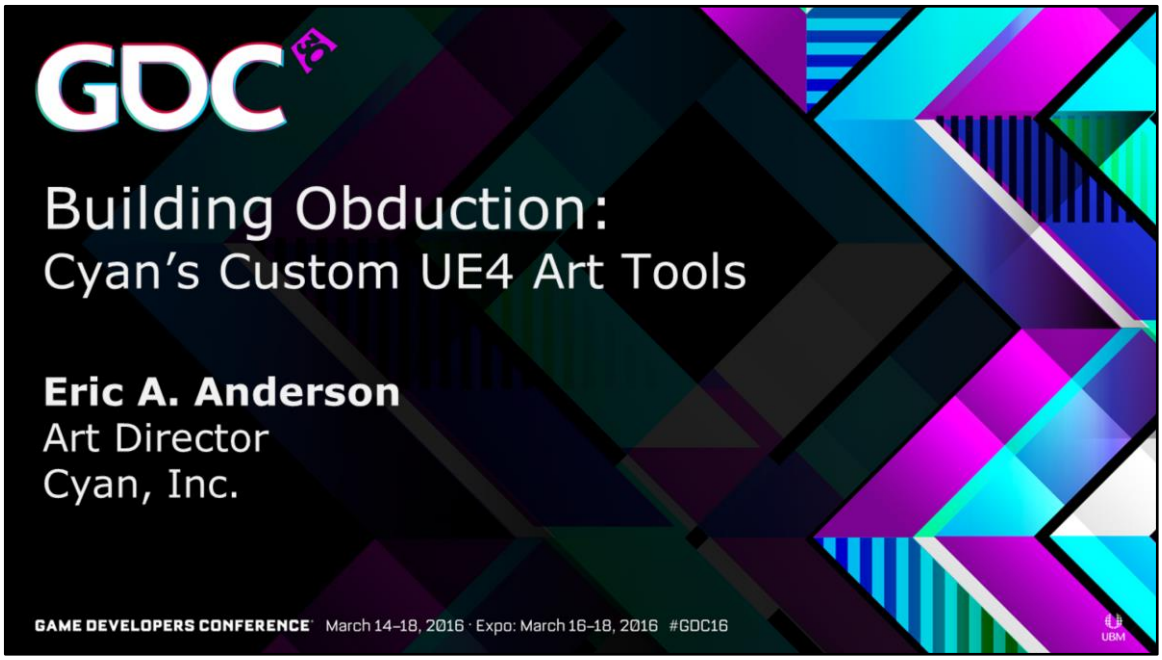

GDC 16
Building Obduction:
Cyan's Custom UE4 Art Tools
Eric A. Anderson
Art Director
Cyan, Inc.
GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 16–18, 2016 #GDC16
UBM

Welcome!

If you're here to learn about some art tools, you've come to the right place.

While this talk does assume at least a basic understanding of the Unreal Engine, it's really about the concepts behind these tools, not the exact implementation - So hopefully even non-Unreal users will be able to find some of this stuff helpful.

First I'm going to do some quick introductions...

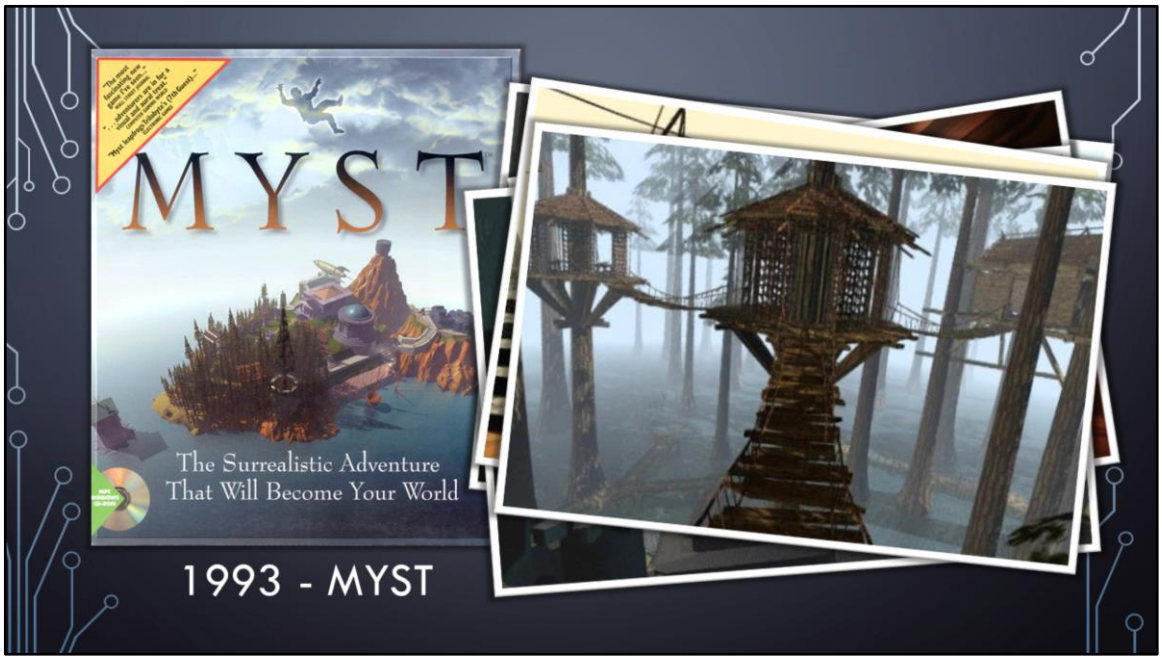
Hi.

I'm Eric Anderson.

I'm the Art Director at Cyan.

We are a small game development studio up in Spokane, Washington, and one of the oldest surviving independent studios in the industry.

I've been with the company on and off for over 15 years.



Cyan is best known for the game Myst, which was released in 1993.

In Myst, players find themselves trapped on an uninhabited - but visually lush island...

...surrounded by challenging puzzles - and are given little to no instruction on what to do - or where to go next.

It was a hit. Myst quickly became the best selling computer game of the 20th century.

It was followed up a few years later by Riven.

While Riven didn't quite sell as well as Myst, it was acclaimed for its visual artistry - its rich story - and its immersive world.

I joined the company in 2000 to work on Uru…

…Our ill-fated foray into Massively Multiplayer Online Games…

…which was released in 2003 as a single player game, due to publisher cutbacks.

Followed shortly by Myst 5…

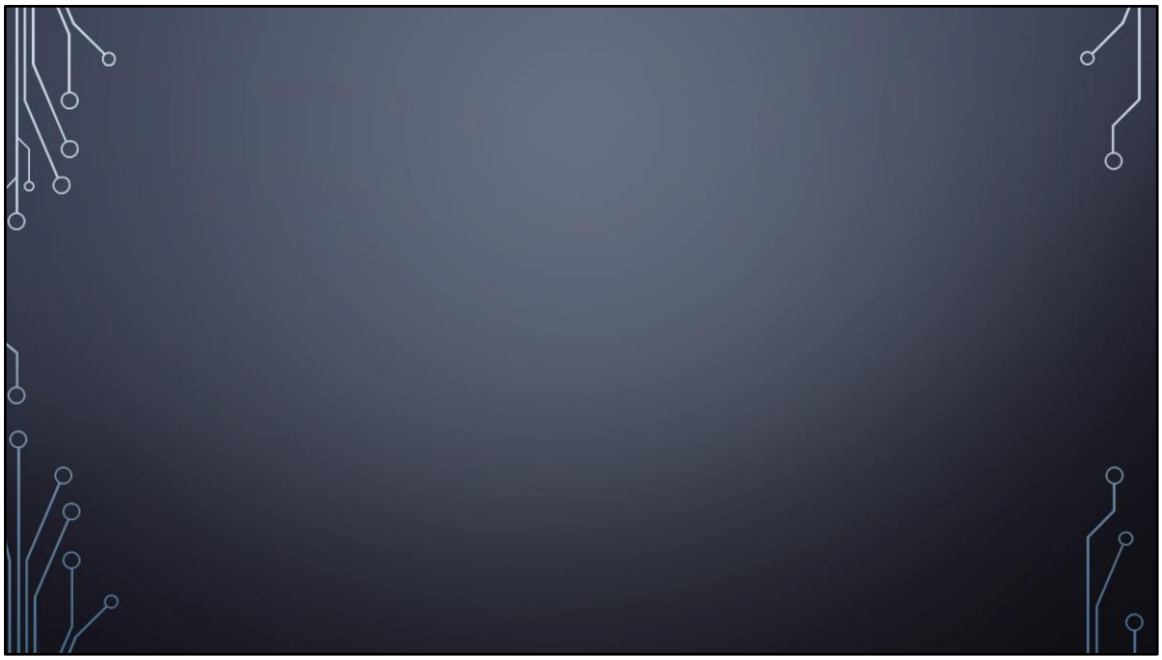
…which was mostly built from the bones of Uru's online content…

…and sold about as well as one might expect.

Then came the Dark Times.

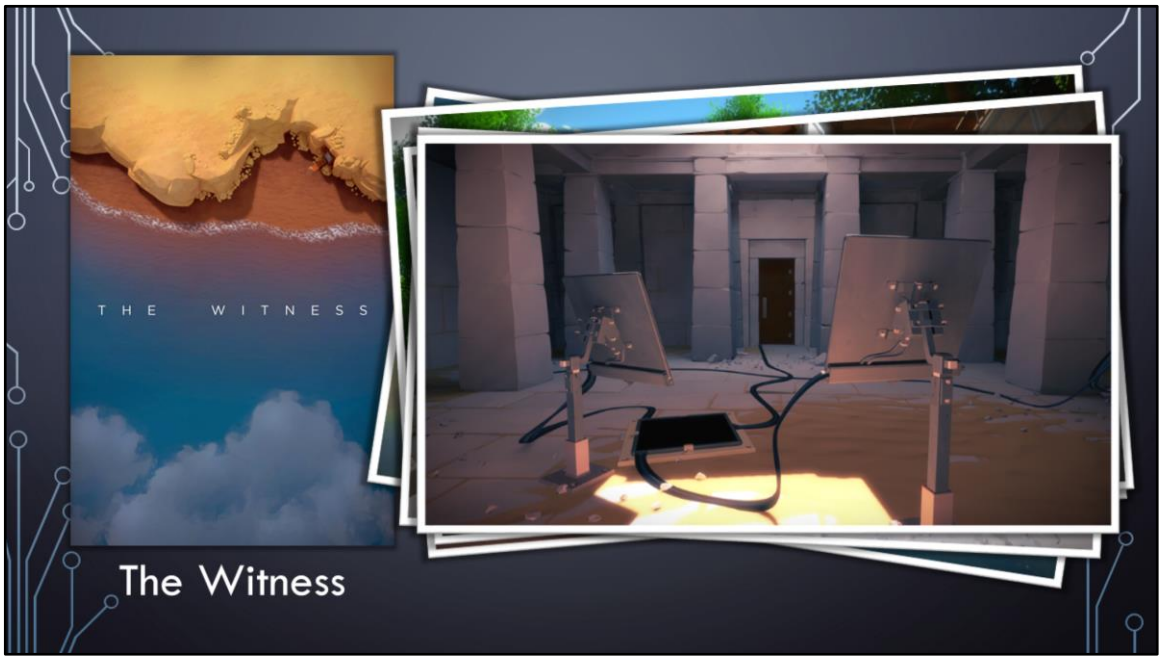
Aside from some failed attempts at landing publisher funding for speculative new projects…

…things at Cyan were quiet for a while.



I spent the next couple of years keeping busy with commercial animation jobs and work-for-hire.

Then, in 2011, I was invited to be a part of a different sort of adventure game project.



Jonathan Blow’s The Witness.

In The Witness, players find themselves trapped on an uninhabited - but visually lush island...

...surrounded by challenging puzzles - and are given little to no instruction on what to do - or where to go next.

I spent three years world-building and helping to develop the look of the game... and I’m honored to have been able to work with Jonathan and such a talented and dedicated team.

It’s a good game. It’s an important game. And if you haven't yet had a chance to check it out, please do.

Despite being scattered across different projects, many of the old Cyan crew were still in touch...

->

…and sometime in late 2012, we began seriously kicking around the idea of Cyan giving Kickstarter a try.

->

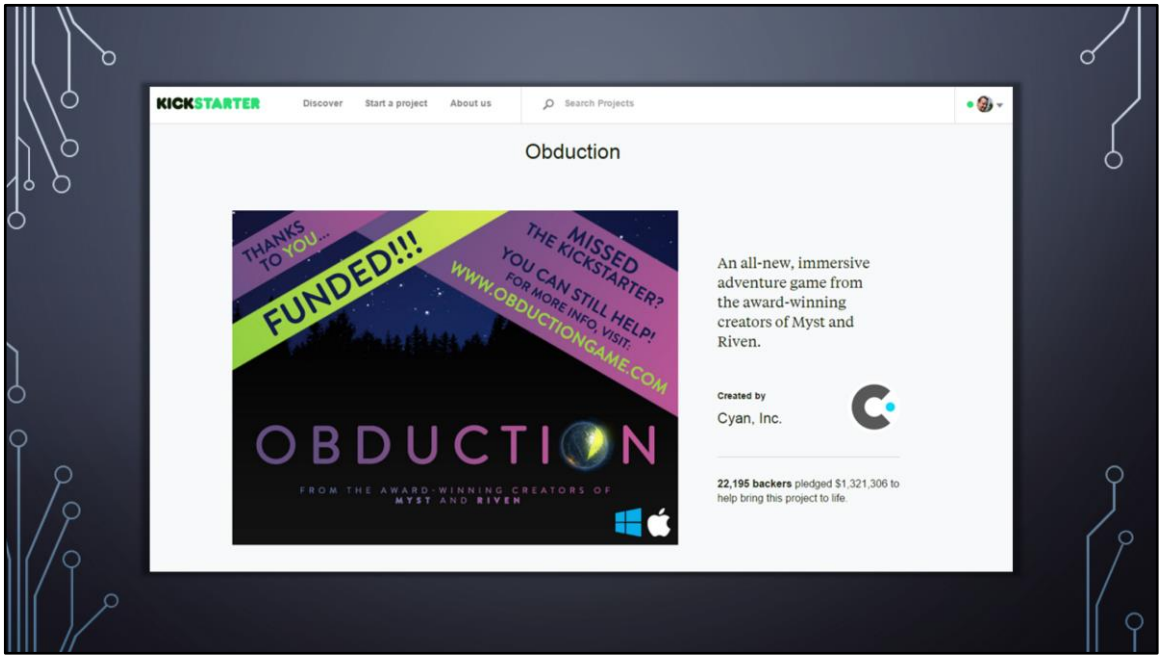
By this time, several notable adventure and RPG developers had found great success with Kickstarter, and we thought Cyan might still have enough of a fan following to give it a shot.

And that brings us to Obduction.

Obduction was conceived as a quote “spiritual successor to Myst and Riven”.

A familiar blend of world exploration and puzzle solving, but in an all new setting.

We sometimes referred to the concept as “Myst in Space”.



After almost a year of preparation, we launched the Obduction funding campaign in October 2013, and exceeded our goal.

In the summer of 2014 I wrapped up my work on The Witness and transitioned to Obduction full time.

Since then we've filled out or dev team.

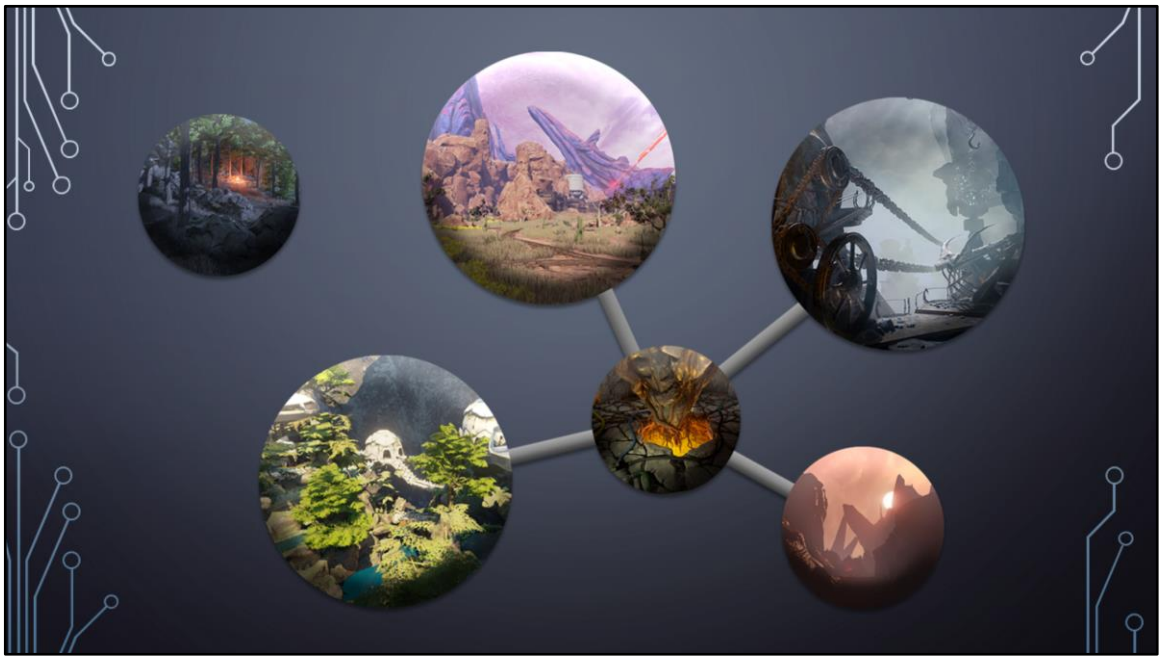
Our art staff is 7 people, including myself.

We've been building the game for roughly a year and a half. Here's how things look right now:

[Obduction Teaser Trailer here]

I should note that the game is playable here at GDC down at the Epic booth on the expo floor.

Come check it out if you get a chance.



The game consists of three large main worlds, two smaller worlds, and a Hub space that connects them.

Between the relatively small team size, the massive amount of real estate needing to be built, and the modest timetable... we really had our work cut out for us.

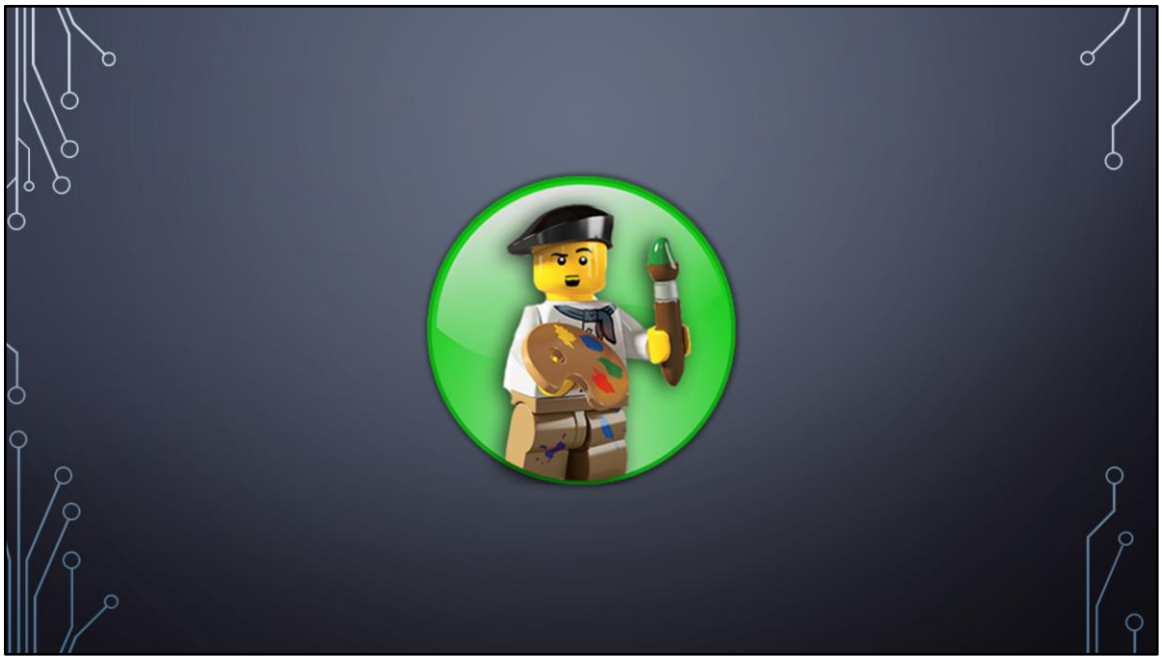
That’s where the tools come in.

We’re building Obduction in Unreal Engine 4.

Any game engine is defined by its weaknesses as much as its strengths.... and Unreal has plenty of both. Thankfully, on the whole, we have been really satisfied with it. If we had to do the whole thing all over again, I’m sure we’d make the same choice.

A big part of that is our programmers having easy access to the engine code - But it also had a lot to do with the material and blueprints systems... and how a technically-inclined artist (say, someone like me)... could implement robust tools without ever writing a single line of code.

Because of this, I’ve grouped all of these tips and tools into two categories.



Those that don’t require any low-level coding, and can be completely developed with visual scripting...

And those that will require the help of a true C++ programmer to fully implement.

We’re going to jump around a lot, and I’m not going to go over every element of these tools, as there simply isn’t enough time...

but hopefully this talk will give you a good conceptual overview of our approach, and if you like what you see from some of these tools, it shouldn’t be hard to implement them in your own projects.

So let’s get into it.

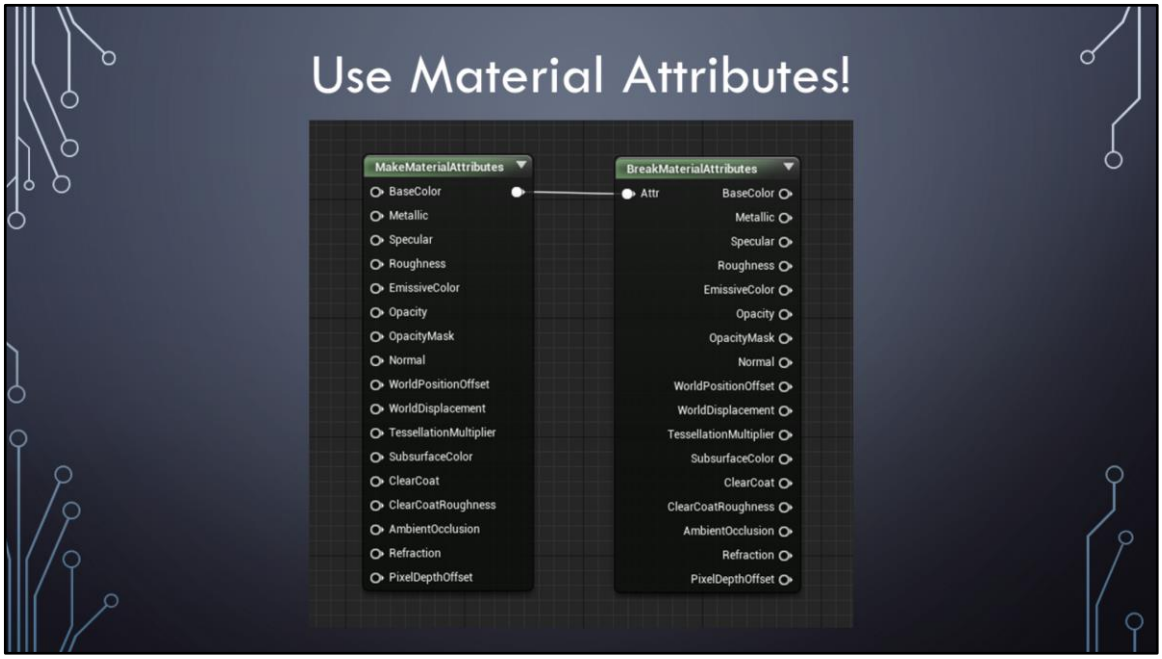
We’re gonna start with some shader techniques, as I think these are the least-daunting from an artist’s standpoint, and can be every bit as useful and time-saving as any other tool.

First up is a general-purpose multi-layer blend shader. This material is super simple in concept... take a small number of unique material layers, and combine them using blend maps and painted vertex color data.

Each vertex color channel (Red, Blue, and Green) represents a different material layer, with black, or no color, representing the base layer at the bottom. You just paint them in or out as needed.

As you can see here, it's pretty straightforward.

There's not just one approach to building a shader like this. And there a couple of tips I'd like to suggest.

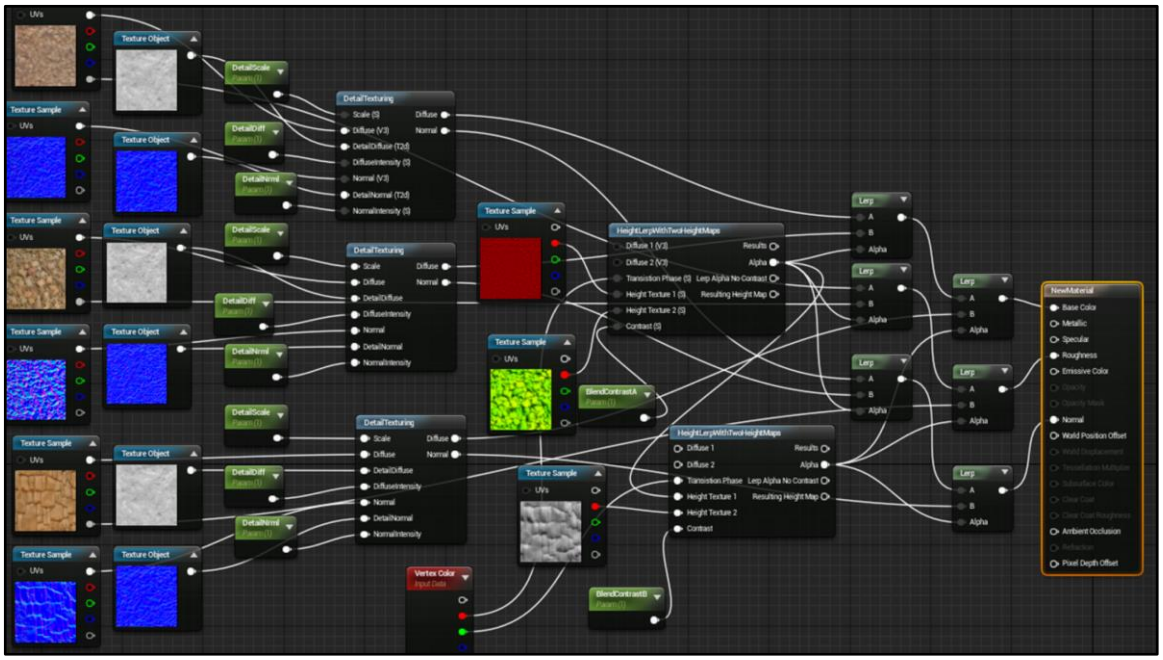


First... Use Material Attributes!

If you're not already familiar with the concept of Material Attributes... it's essentially just a data-struct that combines ALL possible material output pins into a single pin.

These Make and Break nodes can be used to build or deconstruct a Material Attribute struct whenever you need one.

It basically just packs all of the possible material outputs into a single wire.

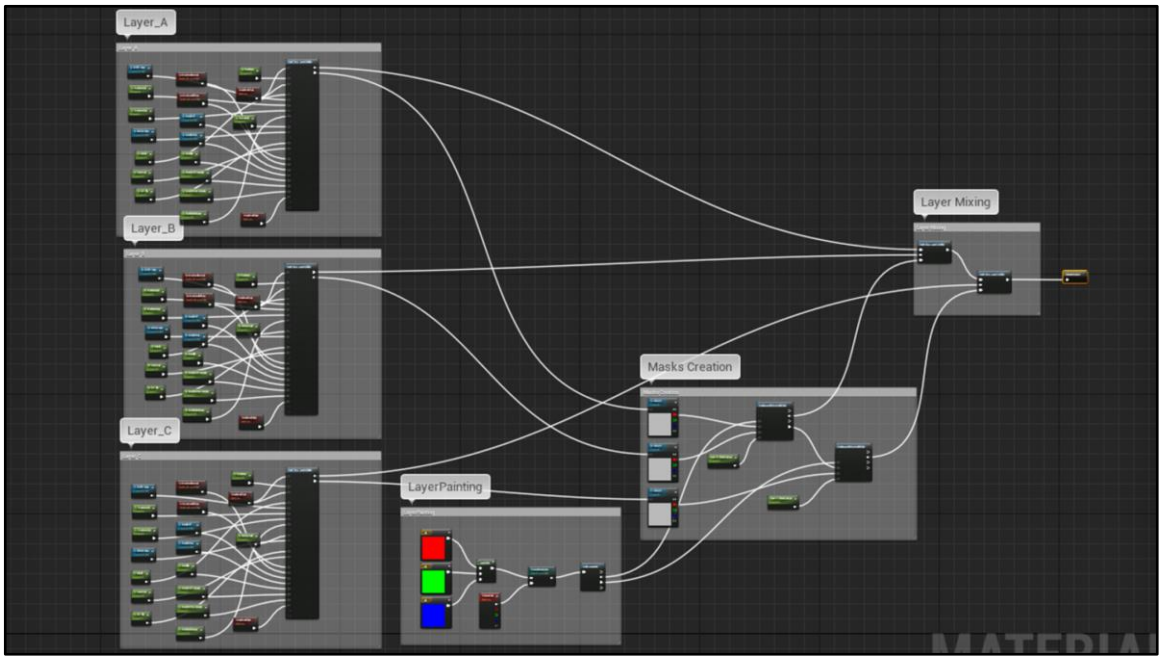


So for instance, here's a 3-layer blended shader that uses lerps to combine each material channel separately.

You need a lerp for each channel you want to output. And then multiply that by the number of layers you need. In this case we're just outputting BaseColor, Roughness, and Normal.

It works fine, but it becomes a real mess to work with.

And other than being multi-layer, it's not even doing anything very complicated.

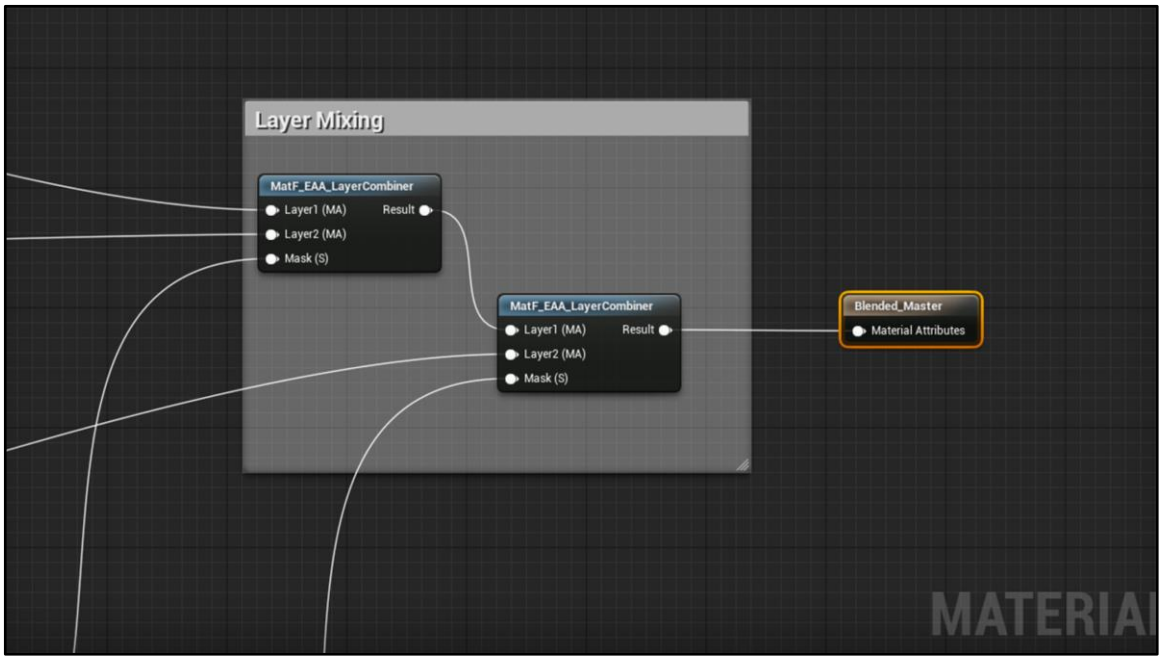


Instead, this is how I like to organize it...

This is a much more complex 3-layer blend shader.

If you look over on the left, you'll see I have each layer being built separately.

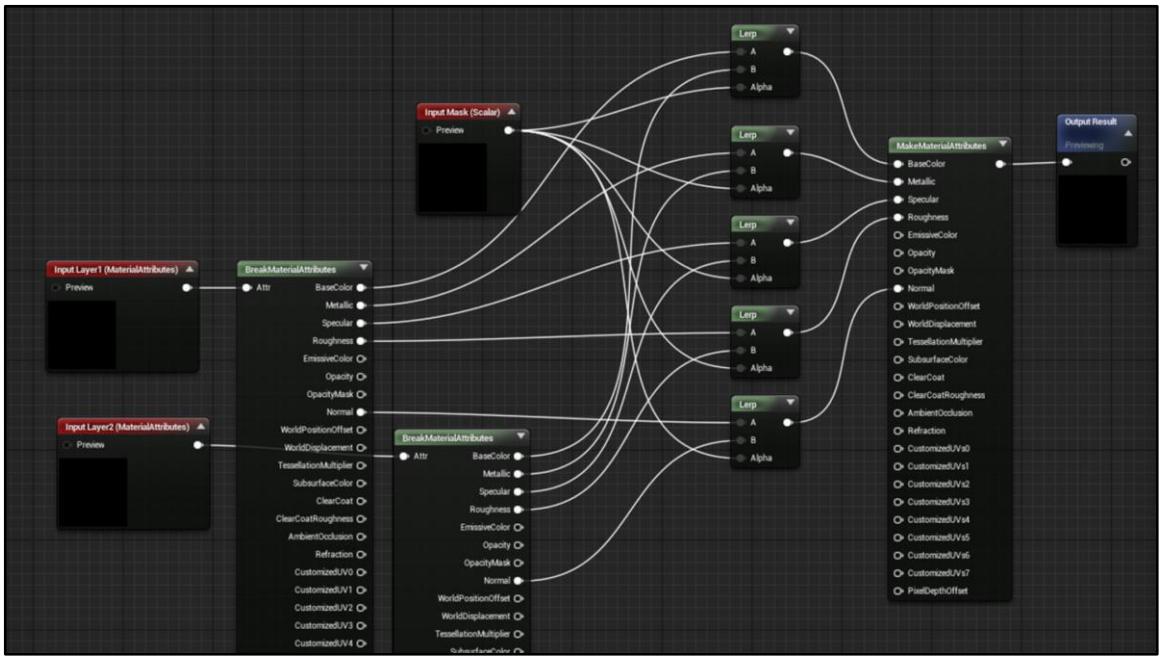
And then I'm simply bundling everything up inside a Material Attribute wire, and then mixing the layers all together.



Here's where the mixing happens.

Instead of a bunch of crazy lerps making a mess of things, I've just got a little function called a "Layer Combiner" that mixes these Material Attribute wires together.

Let's look at that real quick.



It’s literally just a stack of lerps... one for each attribute.

Two pins in, one pin out.

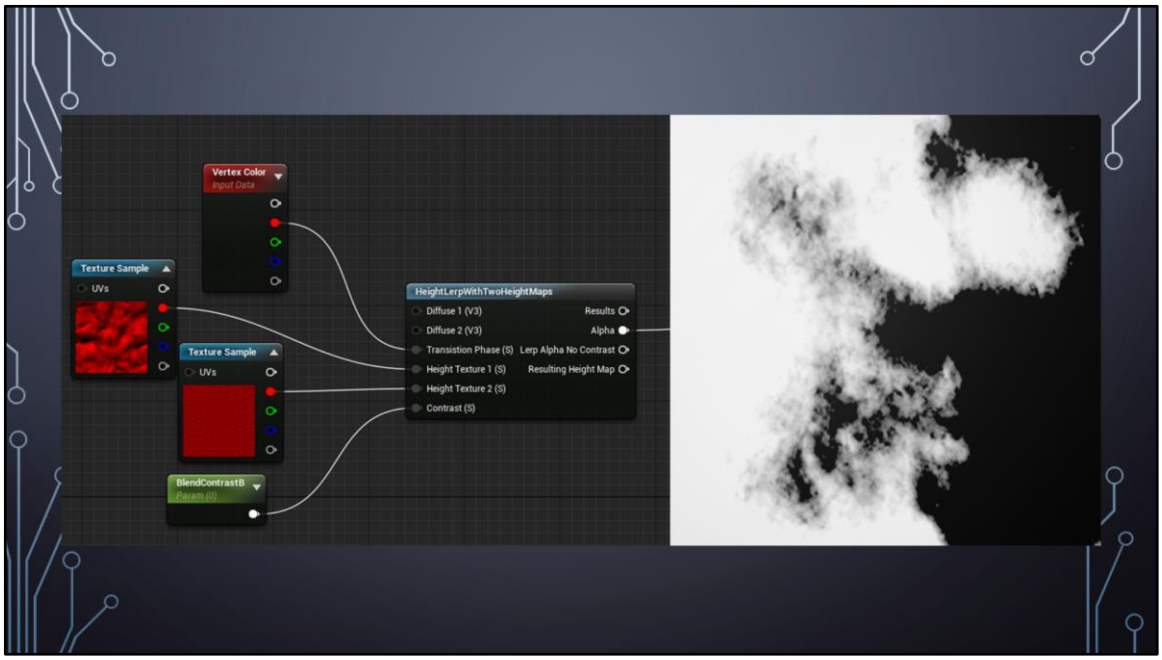
Keeping your parent material nice and organized.

Okay, so one crucial part of this system is how the layers get combined.

One common technique for this is using height-map data to drive the blend, and I think that's a great approach, with one clarification... Use a separate height-map for each layer.

I've seen a lot of tutorials showing layer blending using a single blend map... and that works and all... but it severely limits your flexibility.

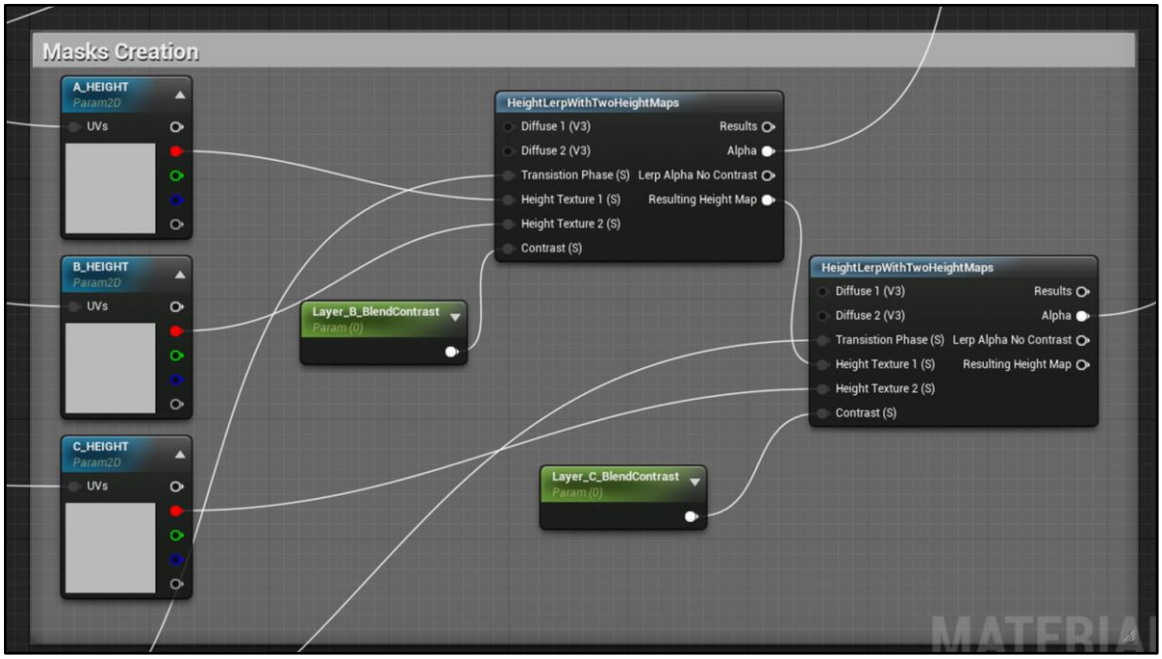
By using height maps unique to each layer, and combining them dynamically in the shader, you can do things like change the tiling of one layer independently of the others, and still get perfect blending.



The way I prefer to do this is with a built-in function called “Height Lerp With Two Height Maps”. You just plug in both height maps, a transition phase - which in this case it just the vertex color - and a contrast value.

That’s the other benefit of decoupling the layer maps... you can dynamically soften or sharpen the blends, independently for each layer.

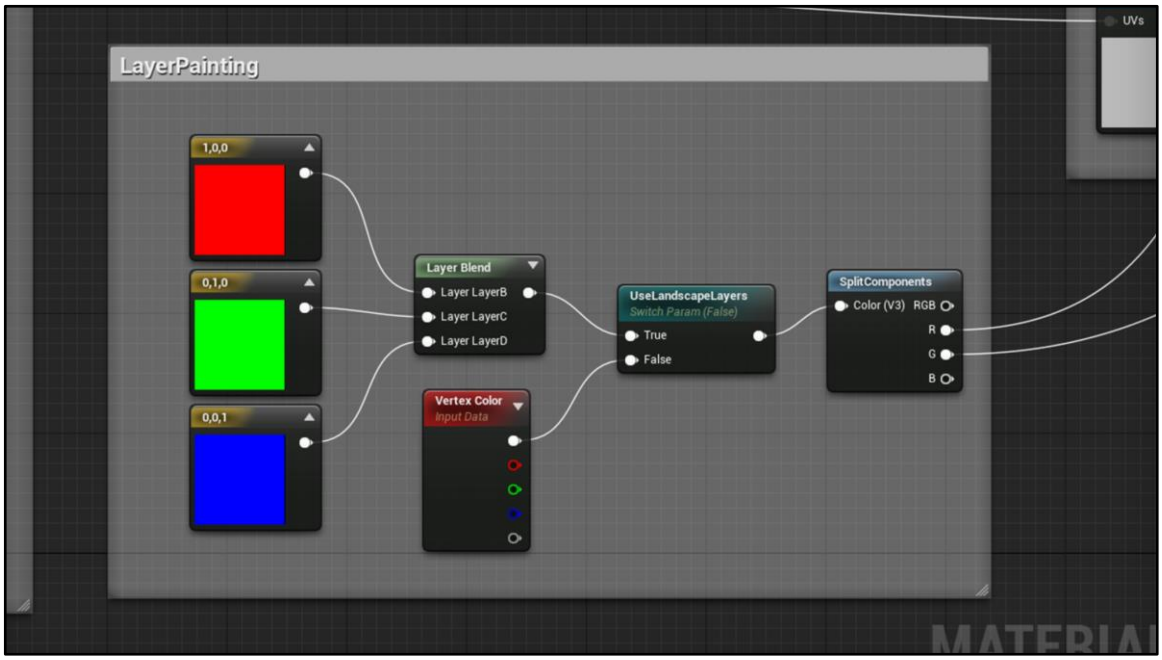
This node also has diffuse color inputs, since it’s normally meant to blend a single material channel... but all I need is the resulting alpha map, which I plug into my layer combiner function and it blends everything at once.



Note that for the three-layer blend, I'm first combining layers A and B.... and then taking the result of that, and combining it with layer C.

I'm mirroring the same behavior with my Layer Combiner as you saw earlier, and that's really all there is to it.

It's literally just a waterfall of blends, for as many layers as you have.

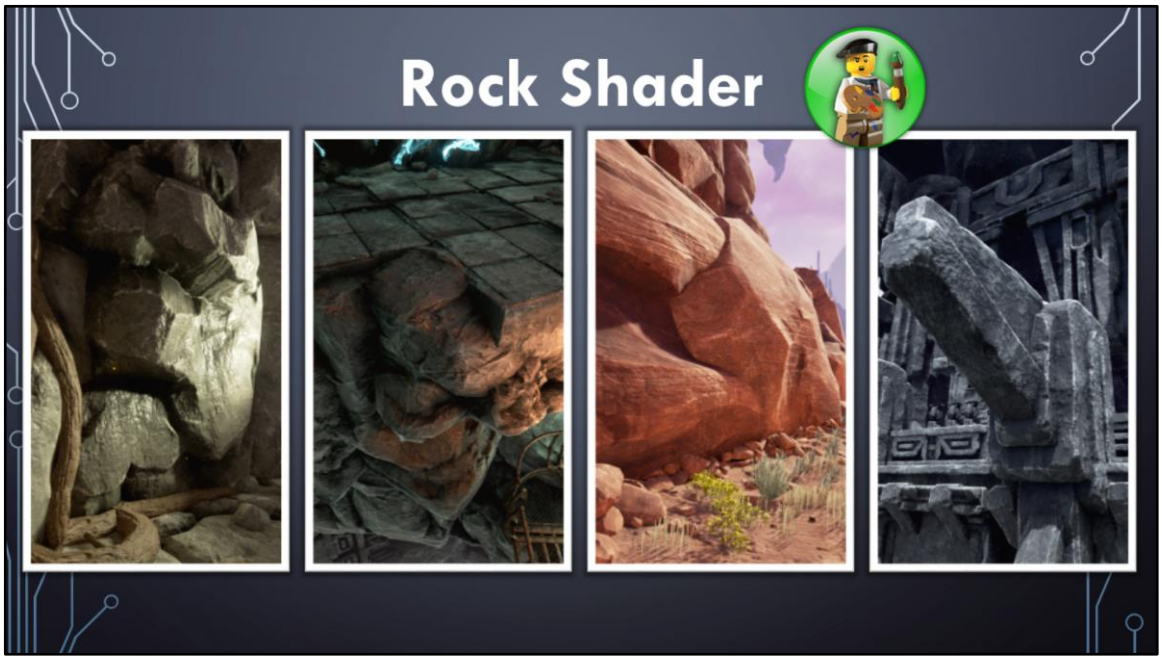


A quick side note... I'm using this same system on landscapes.

Since landscape meshes don't support vertex color and instead use the landscape paint system, we kind of need to hack this a bit.

I just added a "Layer Blend" node, which literally just mixes color inputs together based on the painted landscape layers. I'm just piping in pure Red, Blue, and Green flat colors, and then letting the landscape system mix that all up, and then I'm spitting it back out as if it were vertex color data.

The shader doesn't know the difference. I've got it gated with a static switch so you can just toggle between the two depending on what the material is applied to.

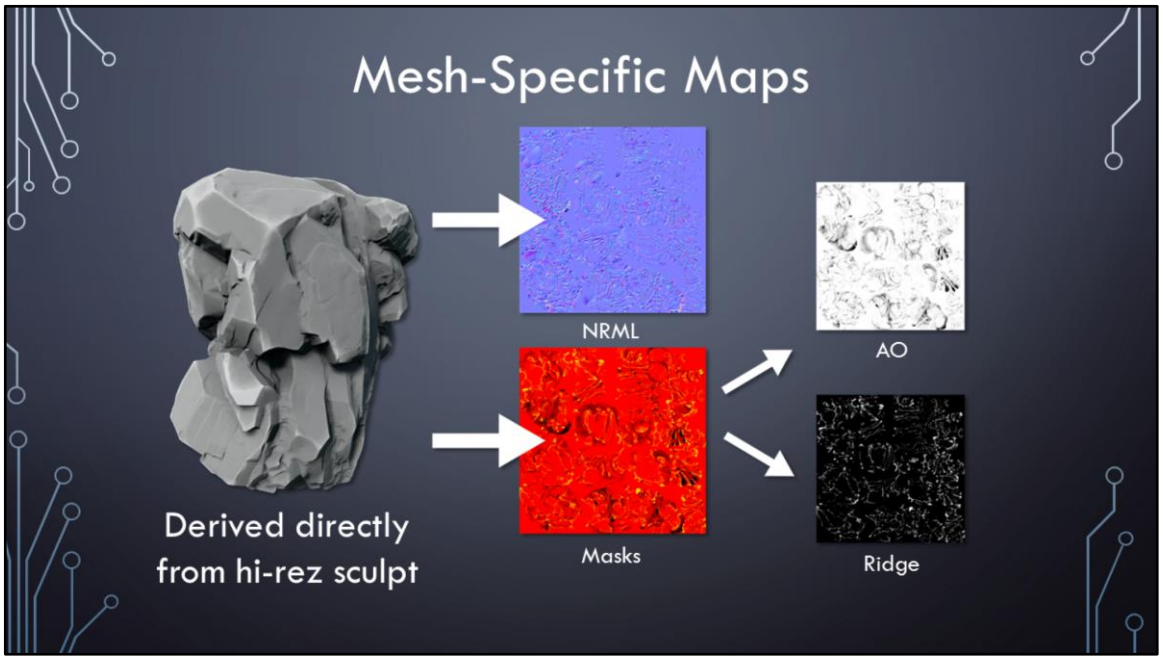


Next up is our Rock Shader. This one is fun.

Since we knew there would be rock structures all over the game... I wanted to write a shader that would maximize the versatility of those rock meshes.

Early on, I created some rock assets using more of a “traditional” method, creating regular diffuse maps, normal maps, with baked-in surface details, etc... but it was far too limiting... Rotating, and scaling the meshes caused obvious inconsistencies, and the assets were difficult to repurpose elsewhere.

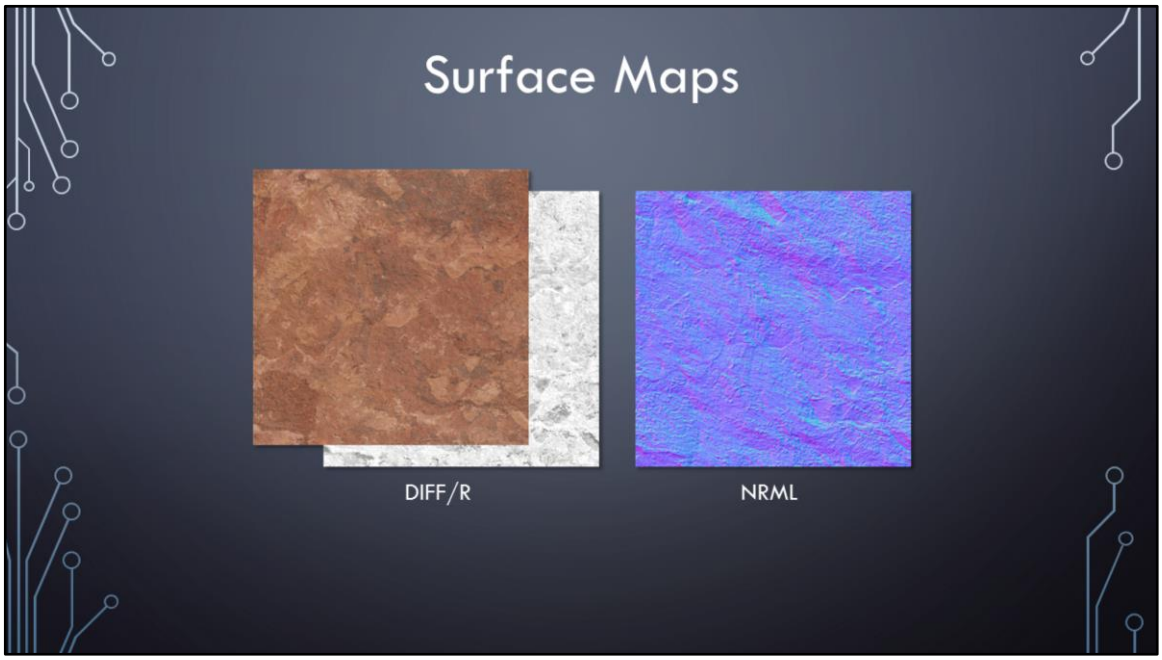
What I ended up doing - and this would become a recurring theme throughout many of our shaders - is de-coupling mesh-specific-maps from surface-maps.



OK - what do I mean by that? Mesh-maps are anything that is specific to the mesh itself.

Things like sculpted normals.... ambient occlusion, curvature maps... Things that MUST be mapped correctly onto the low-poly mesh in a very specific way.

This is almost always just two maps... a Normal Map and a MASKS Map.



Surface Maps on the other hand are just tileable generic textures meant to be used on any mesh.

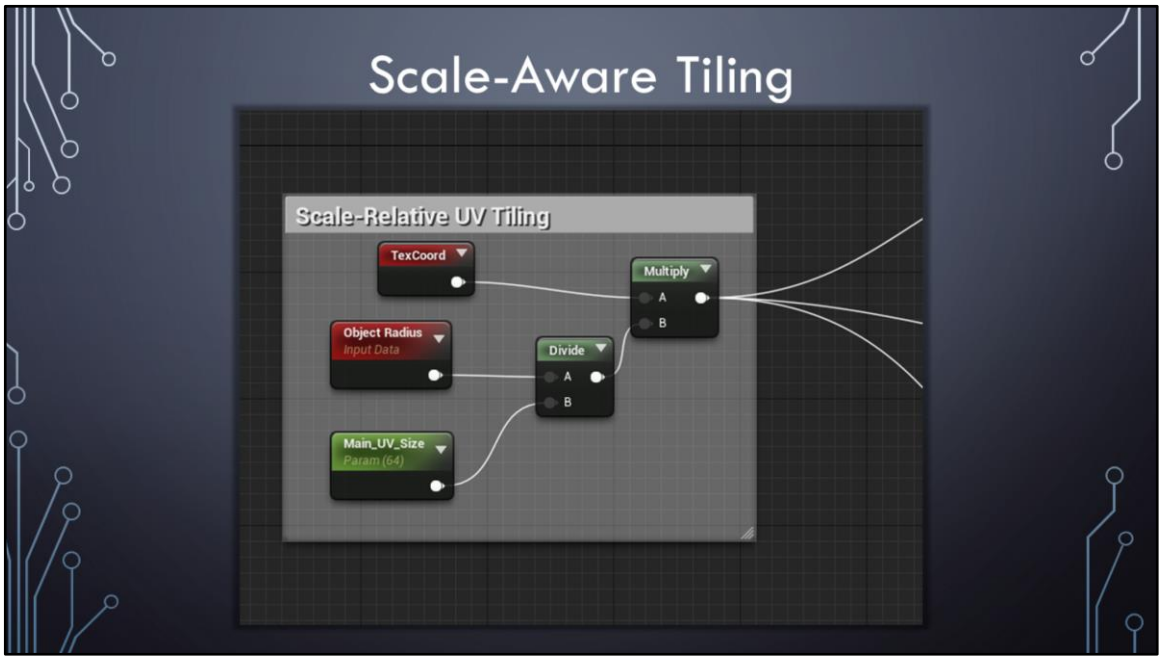
This is almost always just a Diffuse/Roughness Map and a Normal Map.

By de-coupling the textures, you get all sorts of benefits.

First, and maybe most obvious, it becomes trivial to repurpose a mesh for different areas, simply by switching out the surface maps.

It also becomes really easy to re-scale meshes, and make sure they still retain the same relative texel density of nearby rocks.

This is called scale-aware tiling, and it’s not hard to implement.



You just use the Object Radius node, and divide that by a scalar parameter.

You then multiply the result against a Texture Coordinate node and that's it.

Because the Mesh-maps are just using the UV Coordinates directly, they always stay right where they are supposed to.

I’m simply combining the overall Mesh Normals... so from far away you still see the correct sculpted form of the rock, and from up close you get surface detail that matches the diffuse coloration.

By separating Mesh maps from Surface maps, you can retain mesh-specific AO and Ridge Highlighting even while the underlying surface map is moving all over the place.

Both of those can be modulated further with vertex color painting, for extra control.

There’s really nothing magical going on there, just multiplying or adding the mask channels onto the surface color, and using a vertex color channel to lerp each effect up or down.

So then comes some really cool stuff. World-aligned textures.

For rocks in particular, there are a few phenomena that it's important to not have them move around with the mesh.

In this case, it's things like Water Staining.... Which always flows in a downward direction. And Stratification, which in this example always appears and horizontal ridges.

In both of these cases, the effects are painted-in using more vertex color channels, giving artists full control over where they appear.

The world-aligned textures only help to strengthen the overall form, and help hide otherwise obvious seams between meshes. Plus, you can always re-adjust the position, rotation, or scale of the meshes without breaking the flow from one mesh to the next.

This is all are achieved using the built in nodes “World Aligned Texture” and “World Aligned Normal”.

They do add more rendering cost, but you gain SO much flexibility

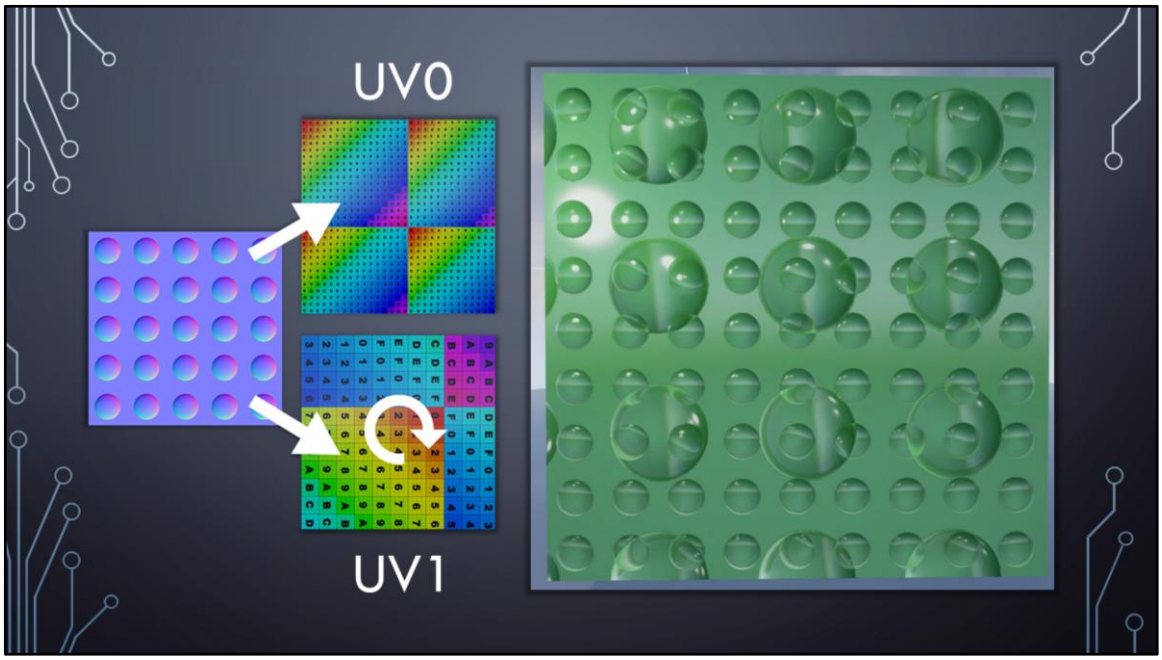
Okay, enough with the rocks. Time to get super geeky.

Let’s talk about our architectural shader.

I’m going to re-iterate our theme of de-coupling Mesh maps from Surface maps...

On the rock shader, everything was on UV0. But for more complex surfaces, what I really wanted to do is put my Mesh-specific maps on a separate UV channel, for reasons we’ll get into shortly.

But before we get there, let’s address a big problem.



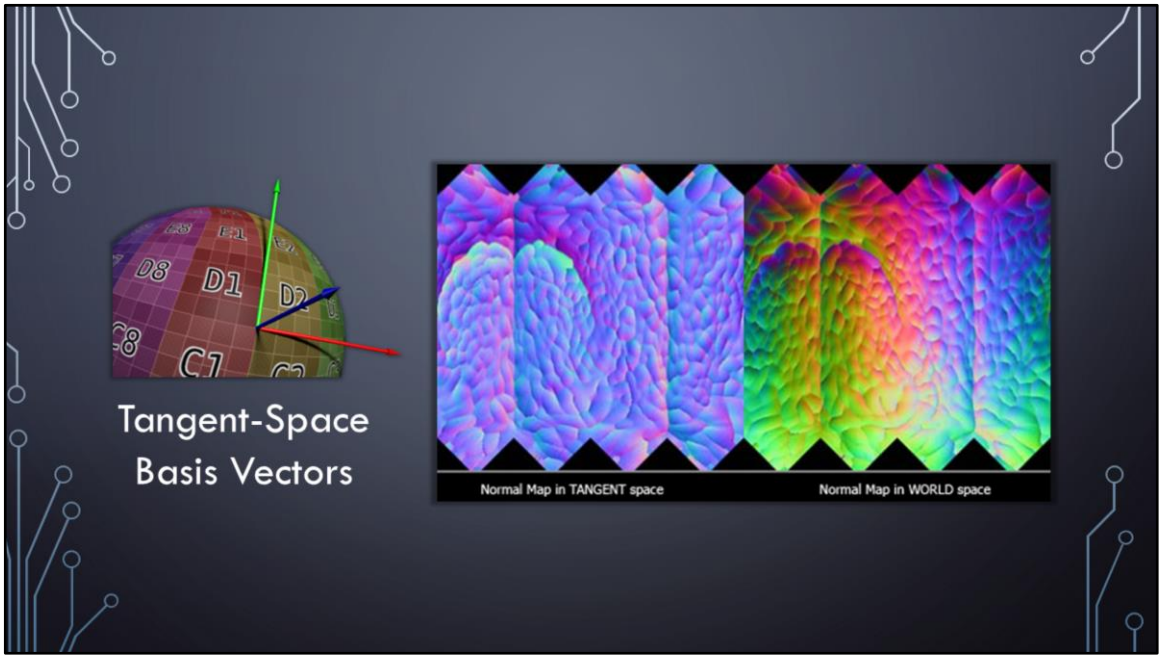
I don’t know if any of you have ever tried to combine normal maps on two different UV channels in Unreal - but this is what happens...

Here we have a simple plane with two different sets of UV coordinates… and the same normal map applied to both channels.

Please note that the second UV Channel has been rotated 90 degrees.

Now look at what’s happening with the lighting.

Do you see that some of the highlights are responding all wrong… Even the horizon reflection is completely sideways on the bumps which are mapped on that second UV. Totally not cool.

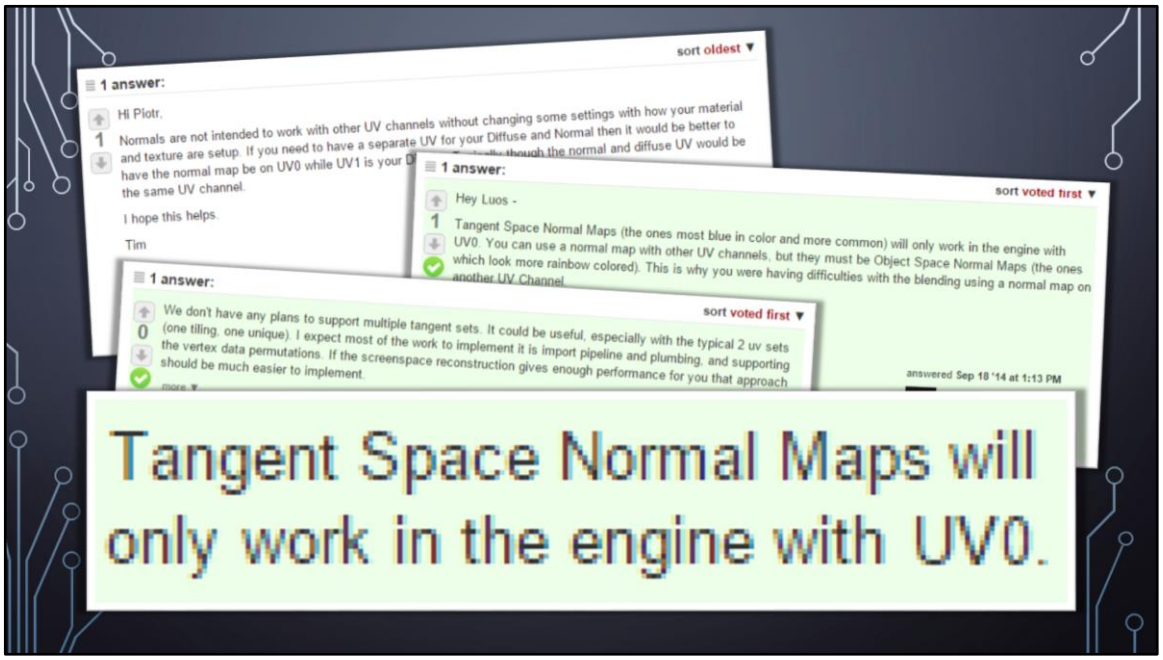


The reason for this is what's called Tangent Basis.

A Tangent-Space Normal map has to be properly transformed from Tangent (or UV) Space into World Space, and Unreal does this calculation for you automatically.

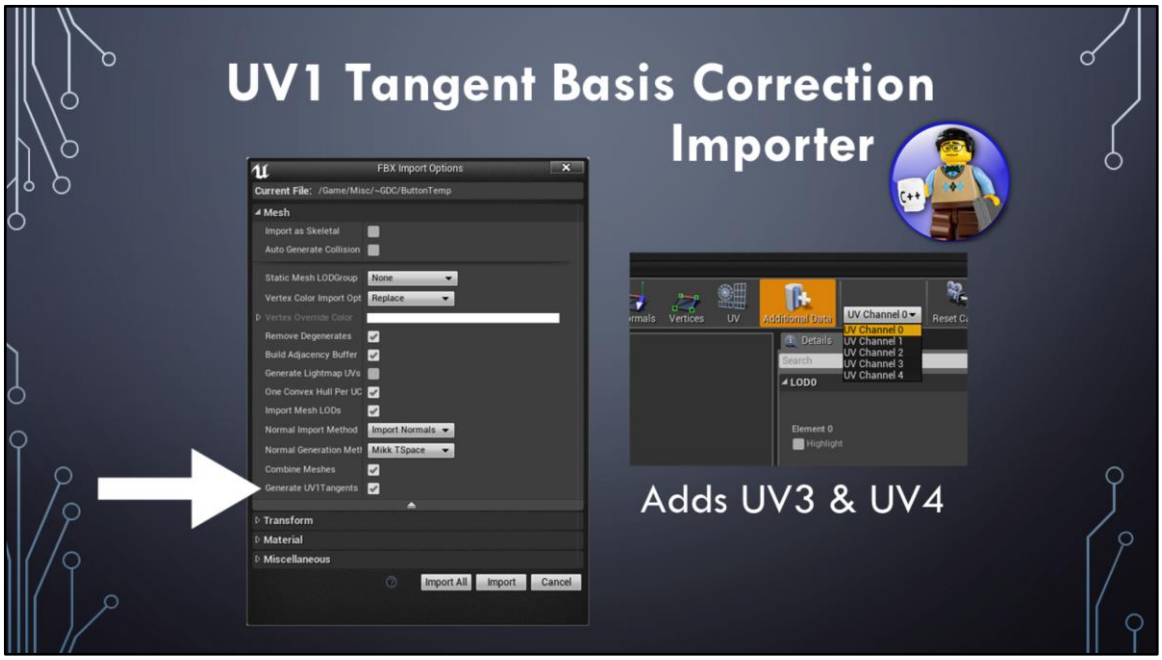
In the simplest of terms, the engine needs to know how the normal map relates to the orientation of U and V in the texture coordinates. The only problem is that the Unreal Engine only calculates a single Tangent Basis per mesh, on UV0.

If your two map-channels happen to be perfectly oriented, so U and V are aligned in each channel, you'll have no problems... but if they are rotated differently at all, you get all kinds of problems. Lighting direction on that secondary channel will never be reliable.



According to Epic, this is just the way the engine works, and there's no way to do that.

So... here's how we did that.

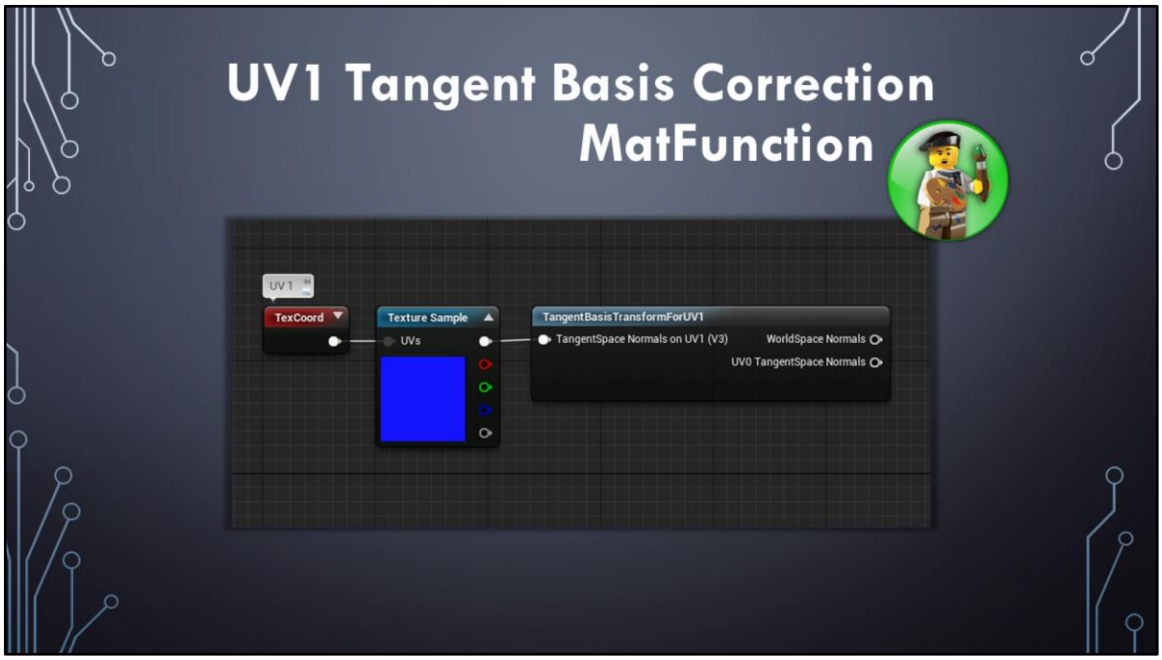


This one you're definitely going to need a programmer buddy for.

What we did was to modify the engine's importer, and add an optional checkbox called "Generate UV1 Tangents". When a new mesh is imported - and that box is checked - we calculate the Tangent Vector for UV 1 (essentially just re-running the same code the engine is already doing for UV0), and we are store that data in extra UV channels on the imported mesh...

Specifically, UV3 and UV4. We end up having to store three floats for the Tangent Vector, and an extra one for whether or not the UV's are flipped (to account for UV mirroring).

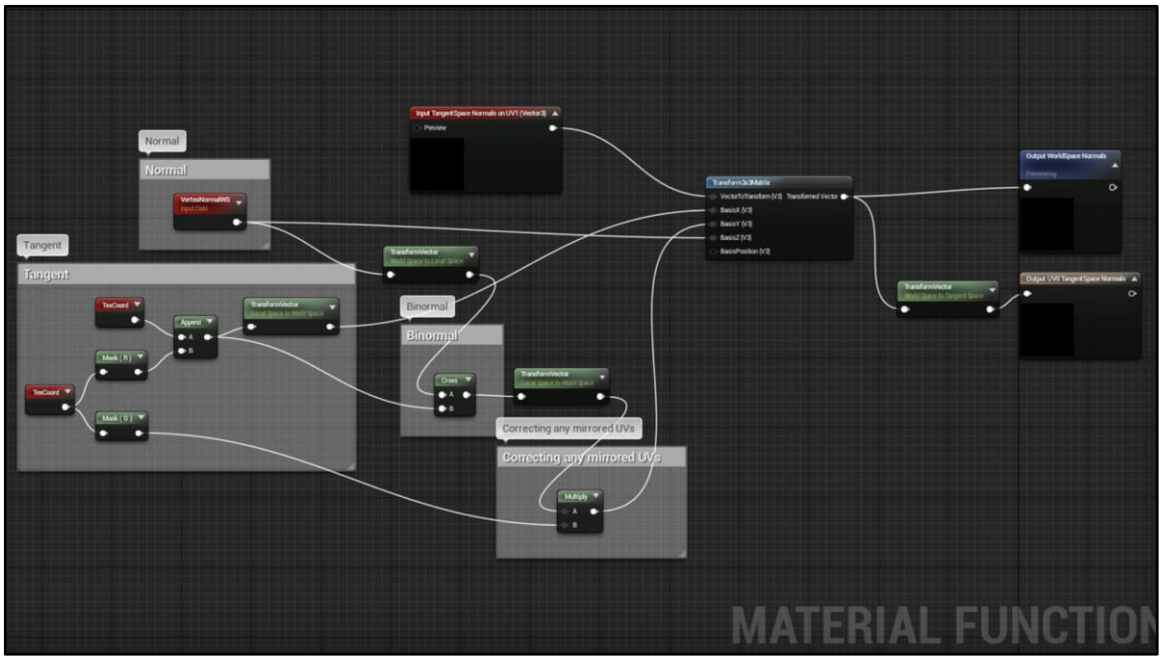
That means we essentially have a 4 component vector to store... and Unreal's UV channels are only 2-channel vectors, so we just pack the 4 values into two different UV sets.



After that, we have a material function to handle the rest.

This function, called “Tangent Basis Transform For UV1” just has a single input.

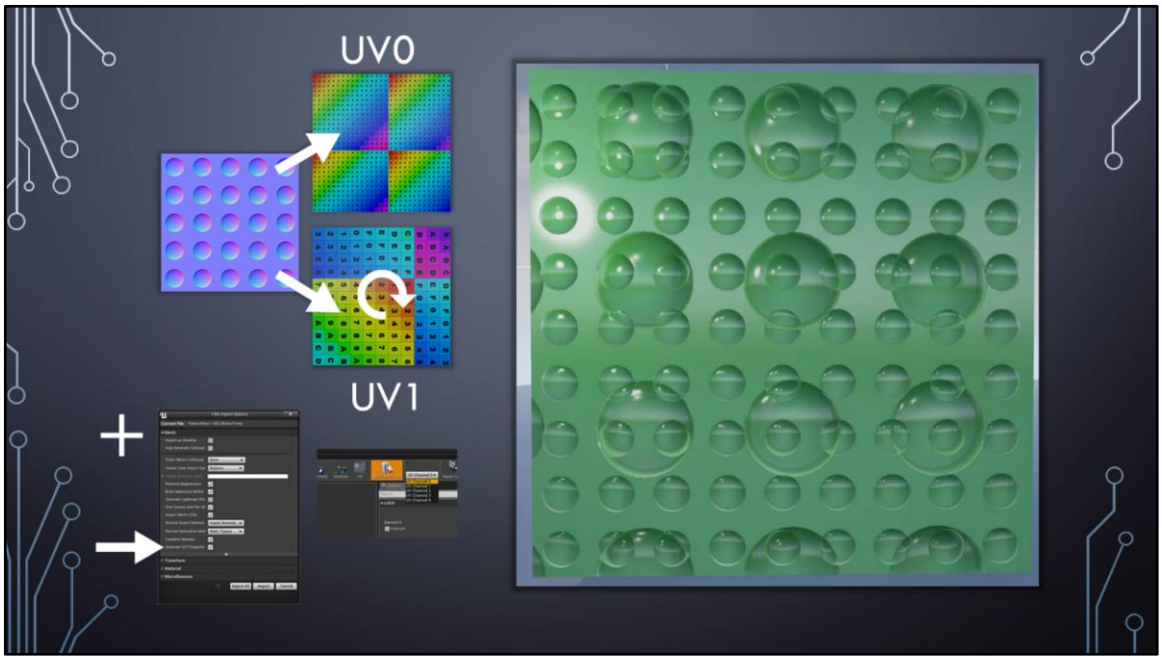
You just pipe in the normal map from UV1.



Inside the function, we fetch our vertex normal, and also the tangent we packed into UV3 and UV4, and use those to rebuild a BiNormal.

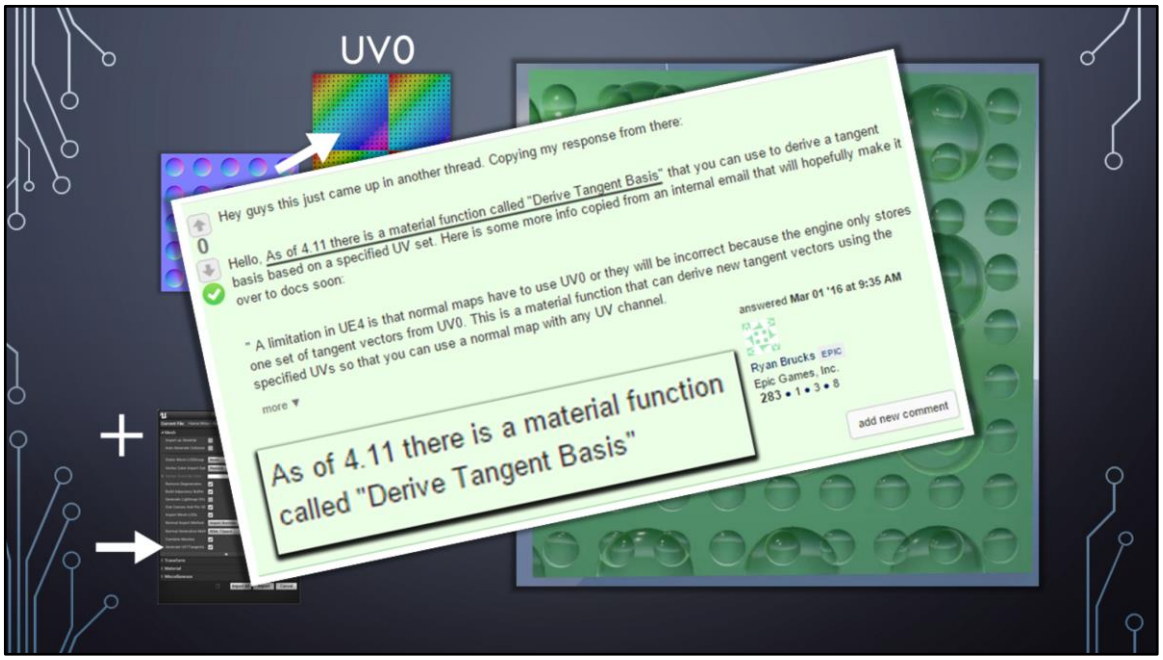
Those three vectors are our new Tangent Basis.

We then use a Transform 3x3 Matrix which outputs our UV1 normals in correct world-space.



The point is, now we can treat them just like we do any other normal map.

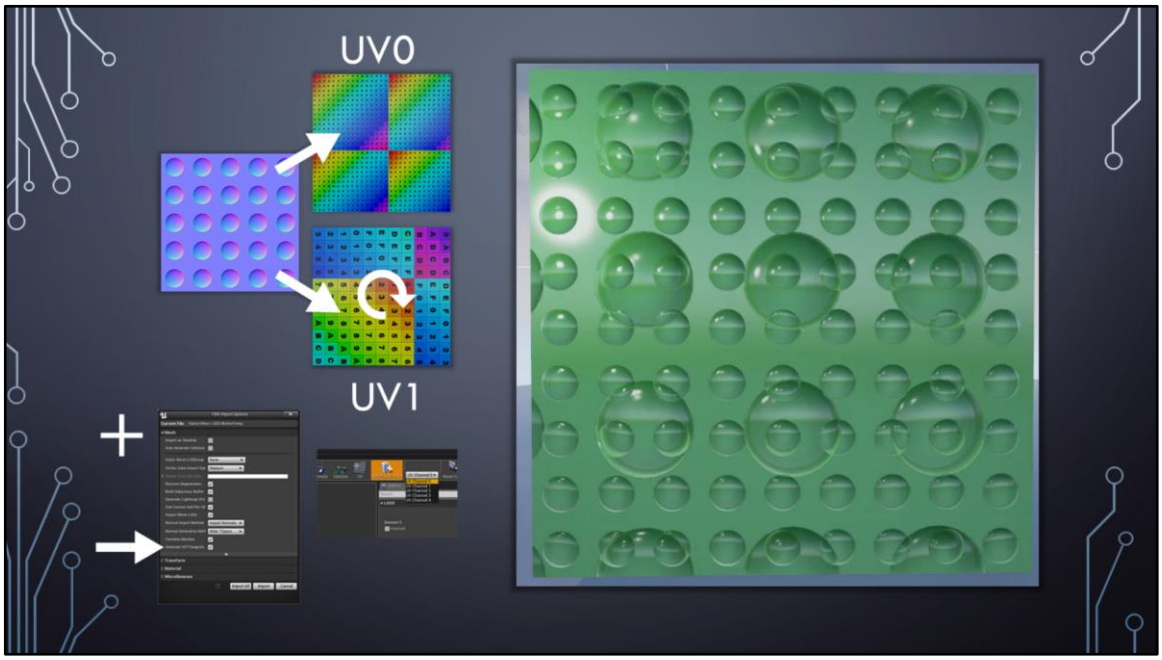
Using this method, we get correct lighting on both UV channels, regardless of orientation.



Quick Addendum:

Very recently, it was announced that similar functionality would be introduced in the upcoming 4.11 version of Unreal.

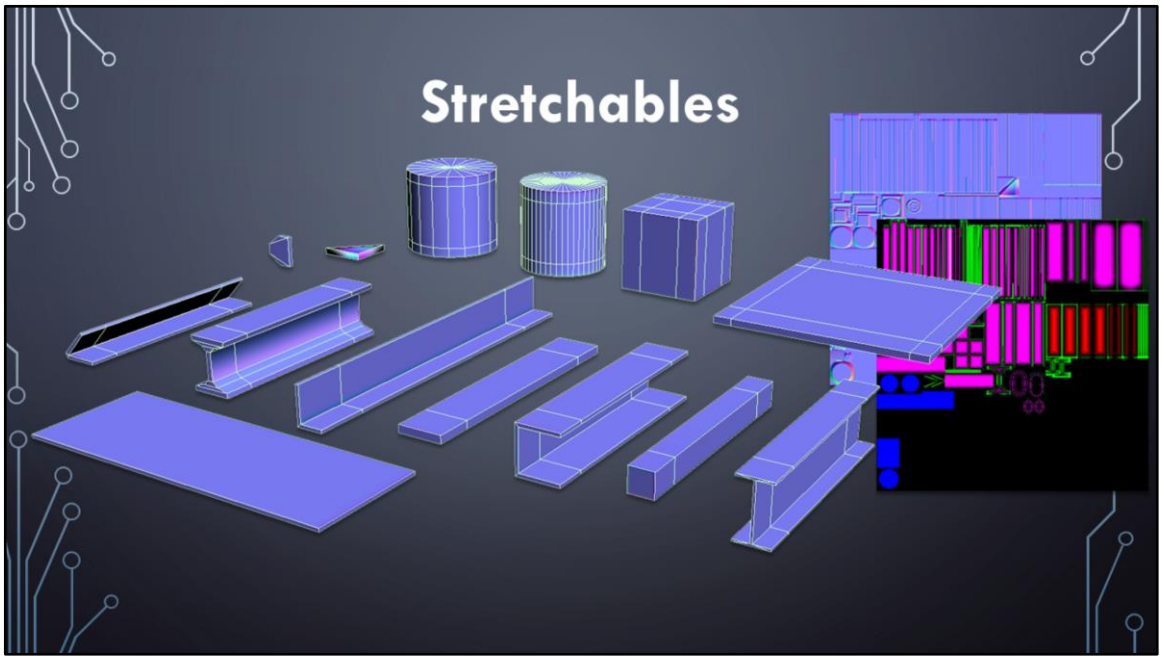
Not having tried it yet, it's unclear how performant this solution will be, as it appears to be calculating the tangent basis on the fly, presumably in the vertex shader… But since our solution pre-calculates all that data on import and bakes it into the mesh, we're likely to keep doing that.



So… Who cares? Why does any of this matter?

The idea here - just like with the Rock Shader - is to maximize re-use of both mesh assets and generic textures, which speeds up production and reduces memory overhead.

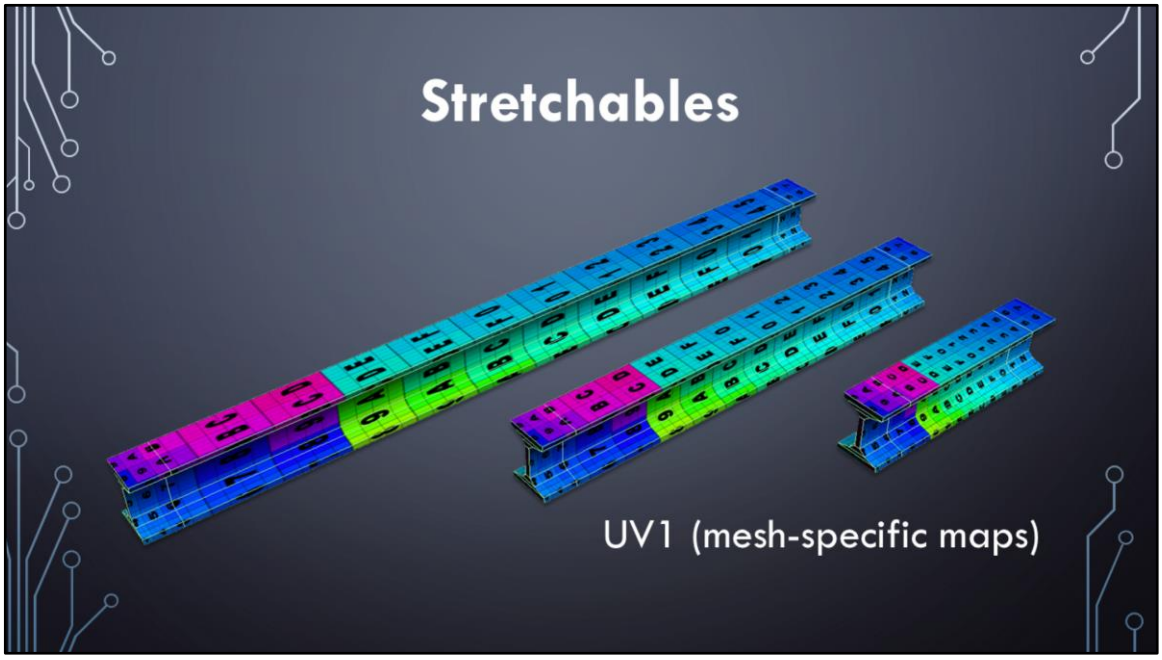
But since architectural components are often just differing dimensions of the same basic shape - and we've now got the ability to put mesh-specific coordinates on their own UV channel - we can get even fancier.



Enter “Stretchables”. Stretchables are our term for generic kit parts that can easily be modified in a 3D editor to be any length or size we need, without screwing up the associated Mesh-specific maps.

We do this with careful placement of control loops to make sure things like mask gradients don’t get distorted, except along purely linear stretches, where the distortion isn’t apparent. The same concept applies to planks and sheets of material.

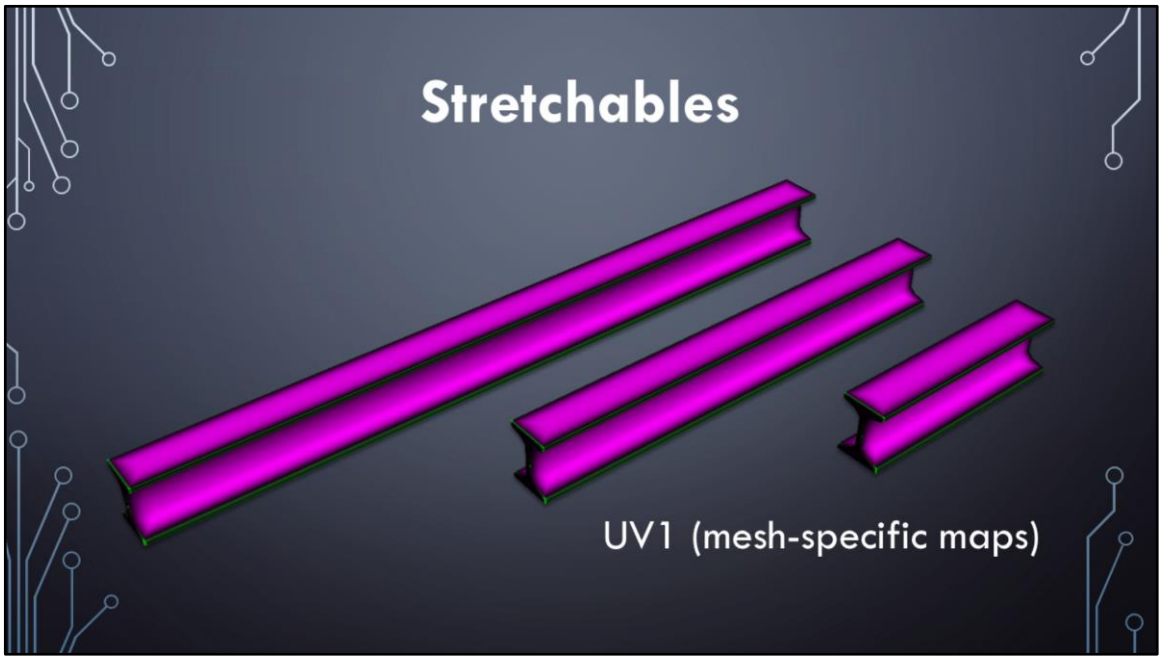
Here’s one of our construction kits... This kit has two mesh-specific maps.. A normal map, and a Masks map. The Masks map contains things like large-form edge gradients and crisp edge highlighting, each on different channels.



For this example, we're going to use just one I-Beam component.

We can make variations of any length, that all use the same texture set and still retain full fidelity.

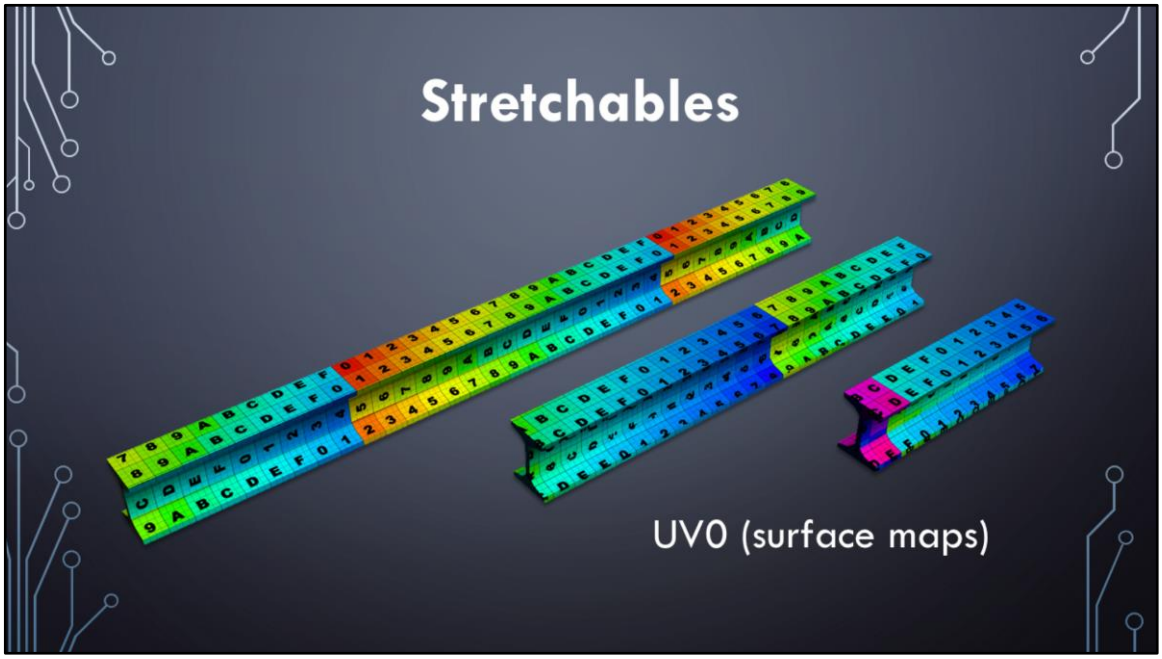
Now, if we look at the UV coordinates here, we can see that there is actually some extreme stretching happening...



...but it's not apparent when looking at the mesh-specific maps, because this asset was built with stretchability in mind.

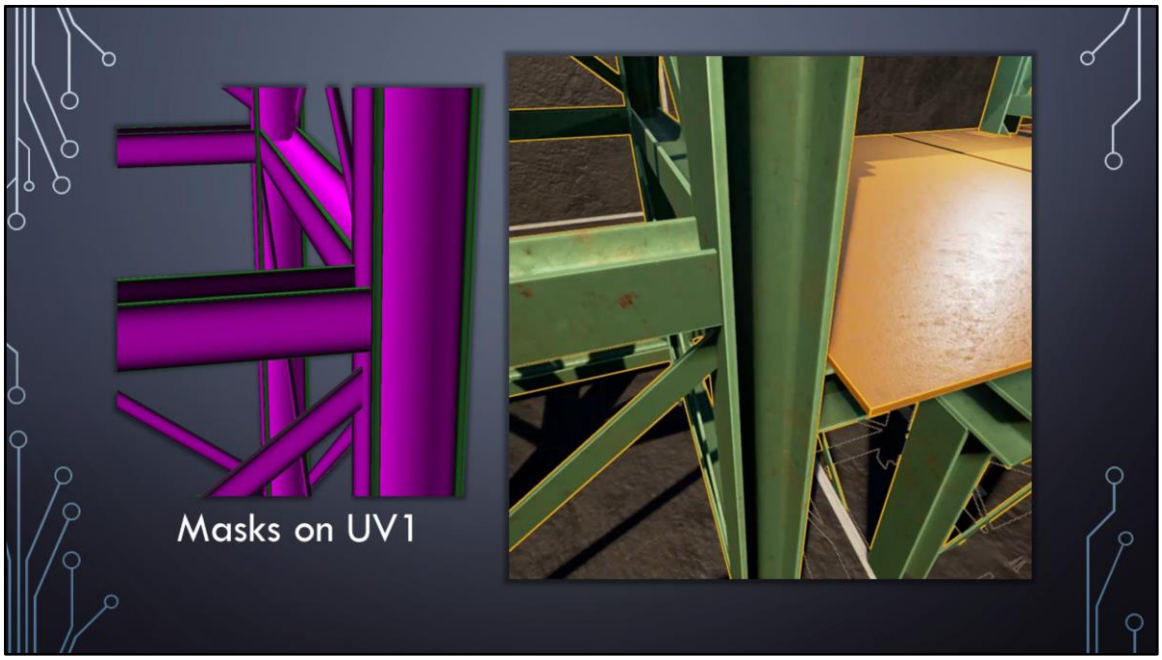
And this is where the multiple maps channel thing comes in.

Because we have the ability to put normal maps on more than one UV channel, we can keep our mesh-specific maps on UV1...



...but put our generic surface maps on UV0...

And they don't need to be related in the slightest. For simple setups like this, I often just apply my UV0 textures with a straight-up box map.

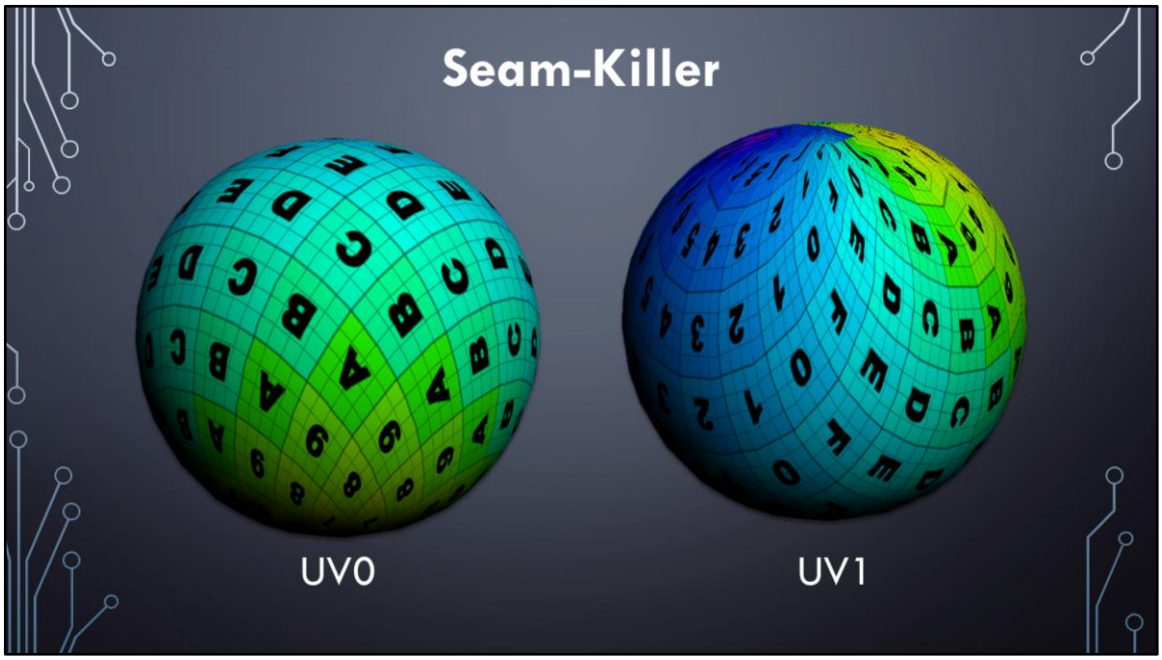


Now, back to that masks-map for a second...

Because we know we always have a correctly-applied edge gradient mask applied to our mesh, we get the added benefit Edge-Aware Painting.

So I can do things like have rust or chipping appear first on the exposed edges of a mesh, and then transition inward as I paint more and more. And of course the influence from that mask is adjustable as a parameter in the material

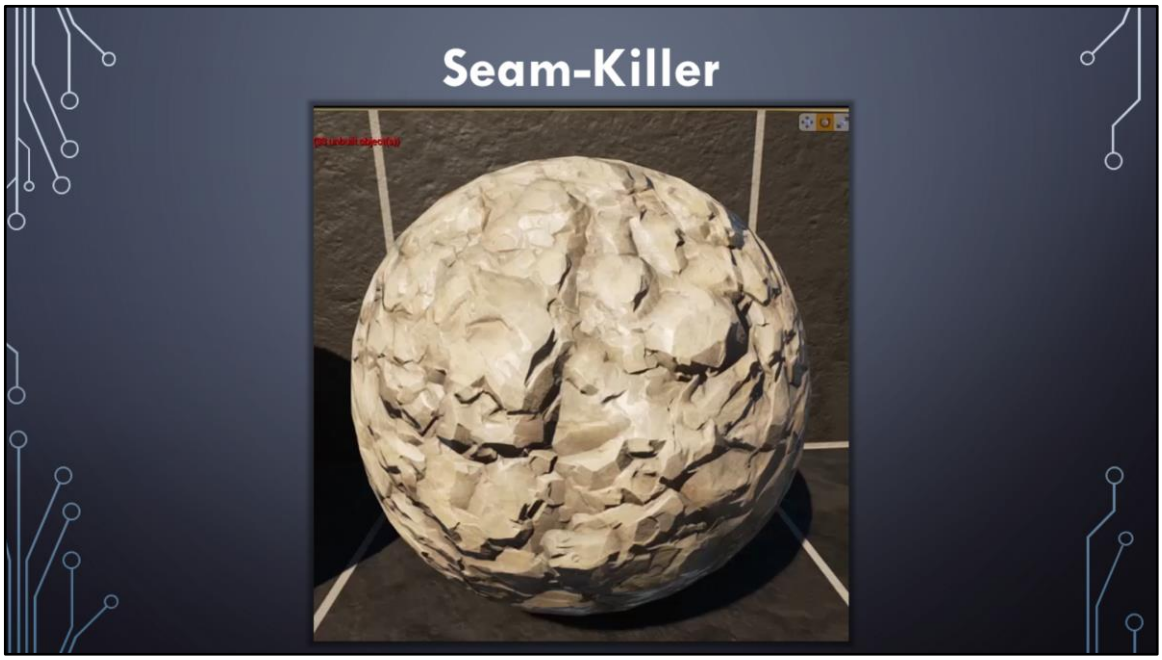
This technique has allowed us to rapidly construct really complex structures while retaining utmost flexibility - often times not ever having to bake out a single new map, since so much of it is kit-based.



Quick side-note... The ability to use normal maps on multiple UV channels also lets us do crazy things like this shader I call Seam-Killer.

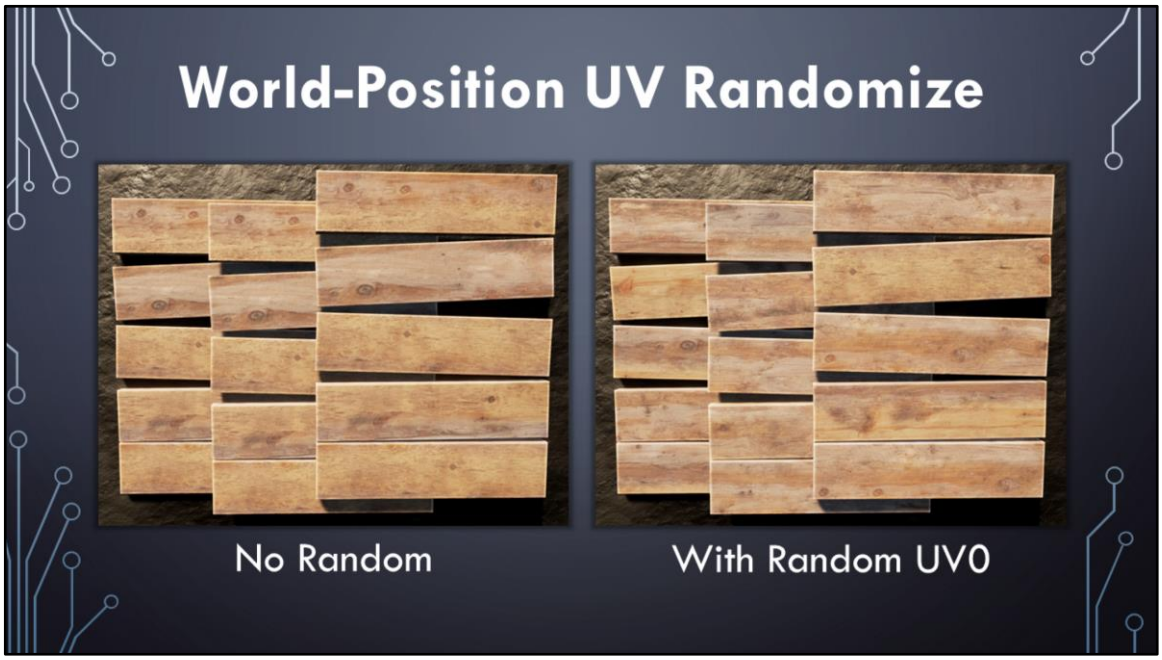
So you can take a mesh that would otherwise be impossible to unwrap using tiling textures.... like say, a sphere.... and apply two different UV mappings to it.

In this case, it's two opposing shrink-wraps... with their pinched poles on opposite sides of the sphere.



Seam-Killer is nothing more than a two-layer blend shader that uses the exact same inputs for both layers... the only difference is the second layer uses UV1.

You then just vertex paint the mesh, and it blends both layers together - using the same dual height map technique we covered earlier... and voila... no visible seams.

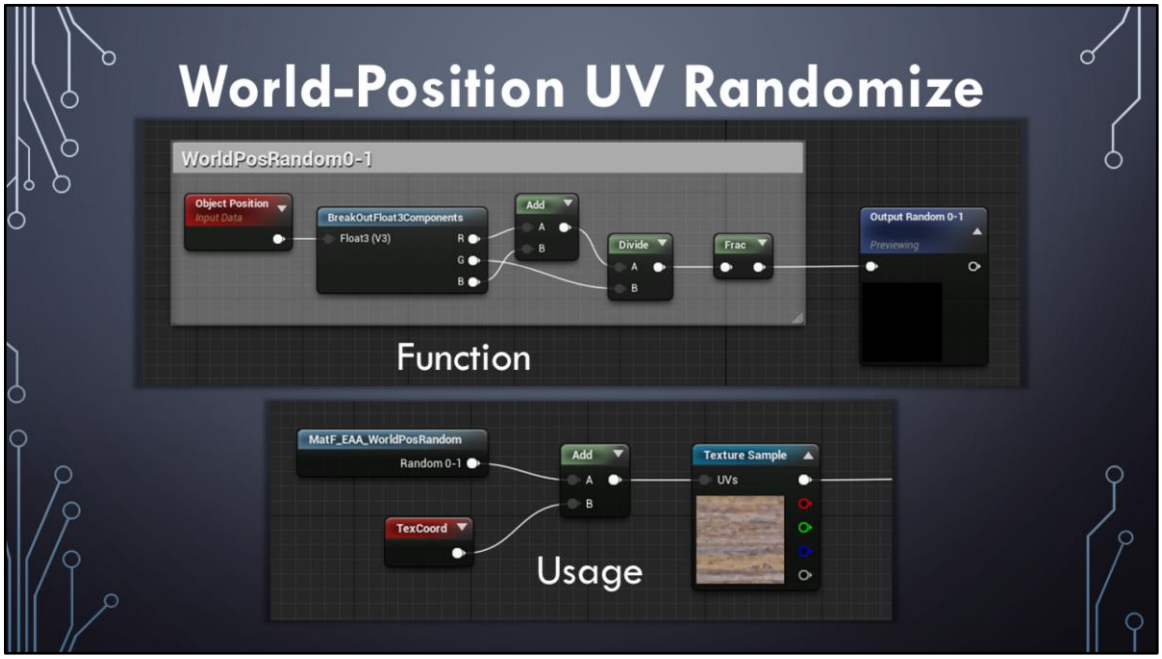


This one is super easy, but crazy useful. We all know that one drawback to modular construction in games is the potential for visual repetition. Usually, the most dead-giveaway that something is being copied is texture repetition.

Copy, Copy, Copy. Pretty obvious, due to the texture.

But since we are all about de-coupling mesh maps from normal maps... there's nothing stopping us from messing with the surface texture of each copy individually.

Now, obviously, we don't want to make different material instances for each one. So let's be smart about it. Let's use the object's world-position to make them each unique.



We've got a function for that!

You're just going to grab the object's position, break out the three components.... and then just do some simple math on those values. Here I'm just adding and diving. Then take the Fraction of that.

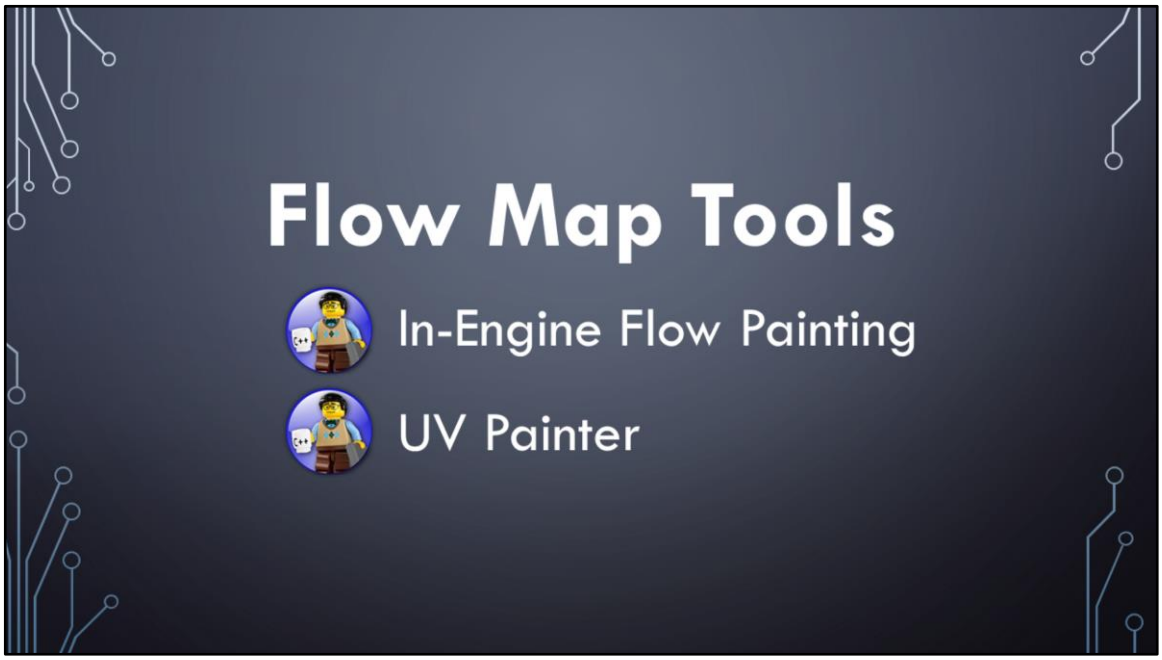
What you'll end up with is a semi-random 0-1 float value, unique to each object.

If you take that value and add it to your UV0 coordinates, you've got random textures.

Here's a look at this in action. I'm also combining it with some UV-driven plank tinting, but the randomization concept is the same.

Add in multi-layer vertex painting, and you've got a ton of extra mileage out of the same assets.

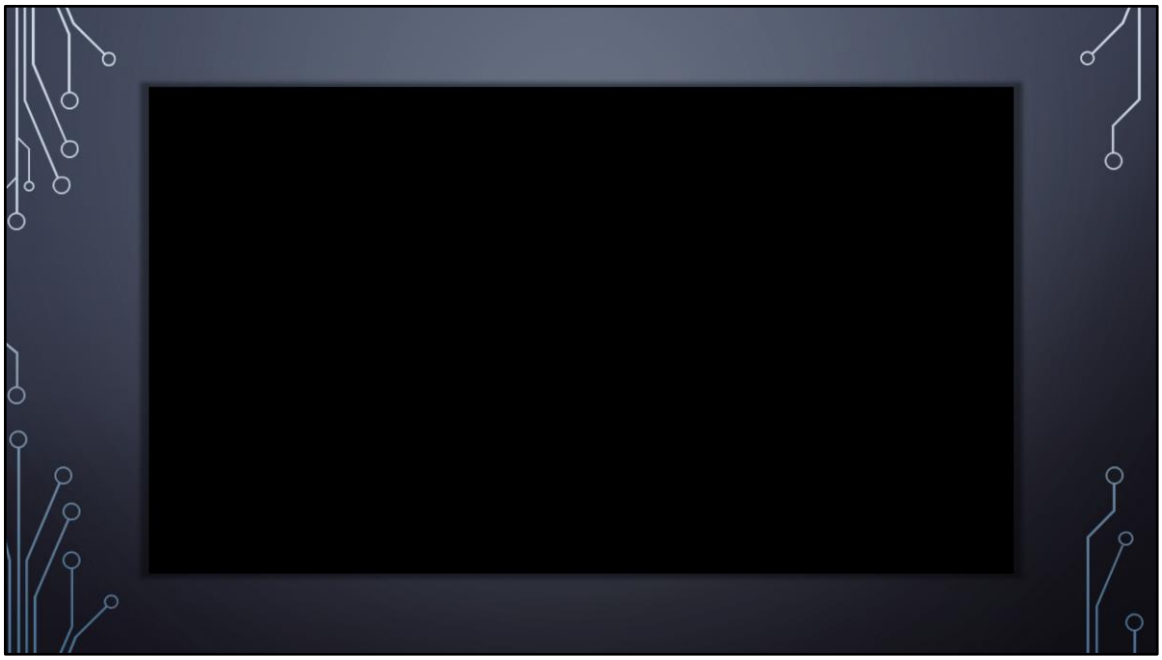
Keep in mind this won't work well on animated objects, so it's best to keep this optional!



This little section is all programmer-mandatory.

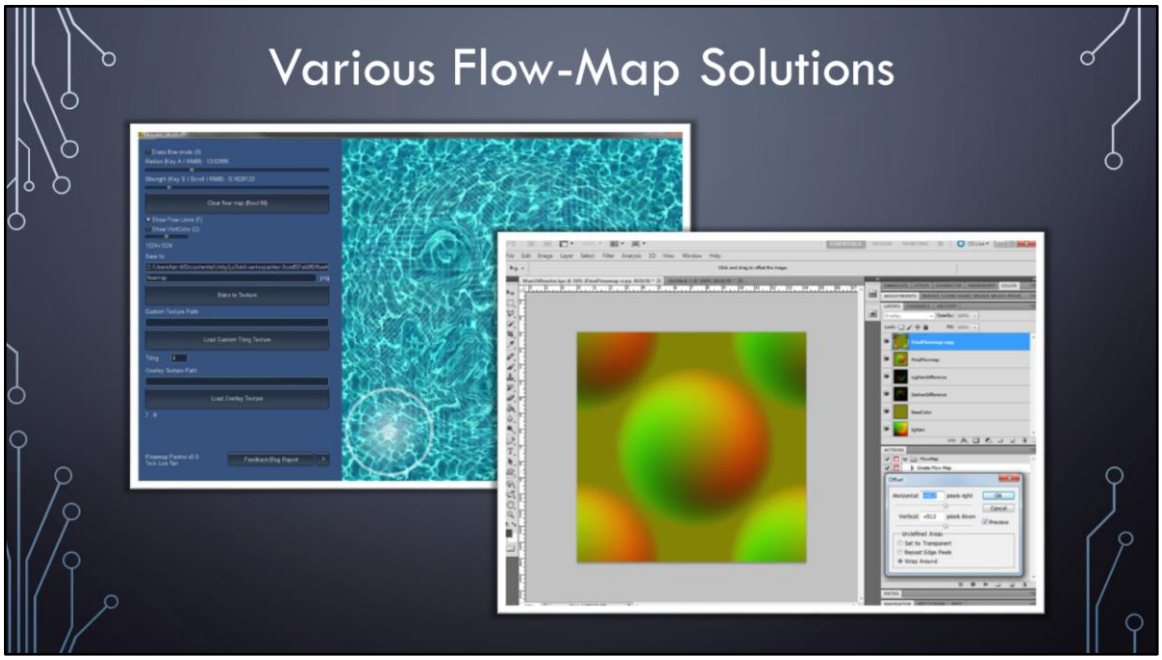
Let's talk about flow mapping tools.

There are several areas in Obduction where we make heavy use of Flow Maps.



Often for fluid motion, but sometimes for environmental or other effects.

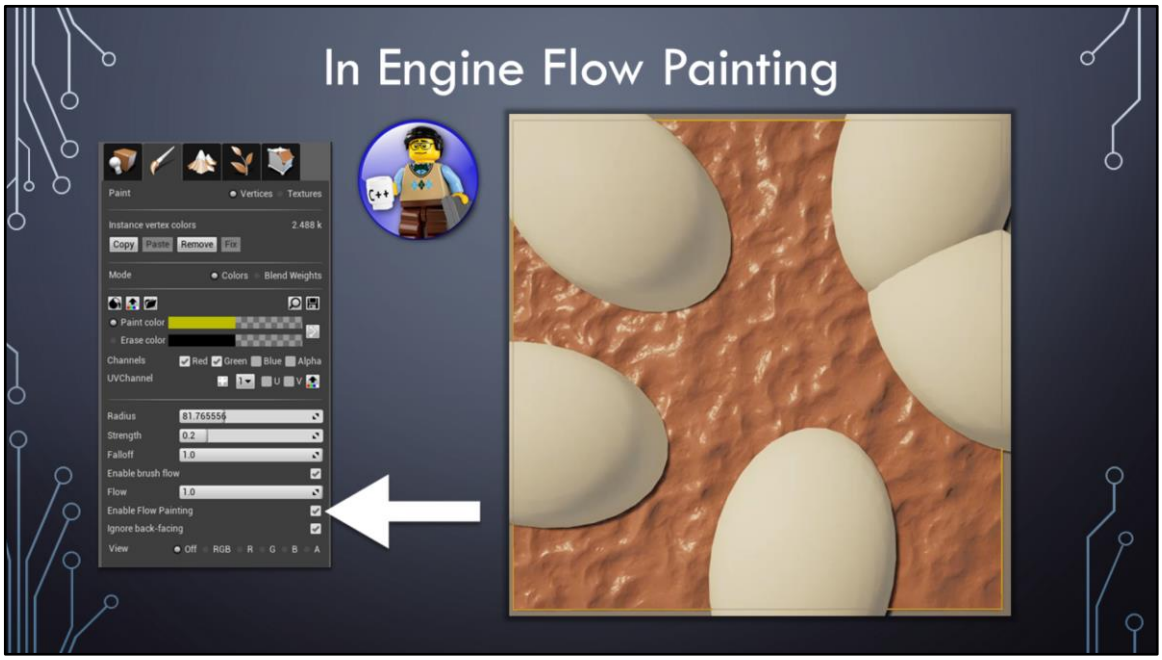
I’m not going to get too detailed about Flow Mapping, as there’s plenty of existing resources and tutorials, but I do want to cover a couple really awesome features we added.



If you've ever worked with Flow Maps inside Unreal, you probably know what a pain they are to author.

There's no standard process, and most of the suggested tools are a big interruption in workflow.

So one day, I was working on a river shader, and one of our programmers - saw the process I was using.



He looked at it it said - I can make this better.

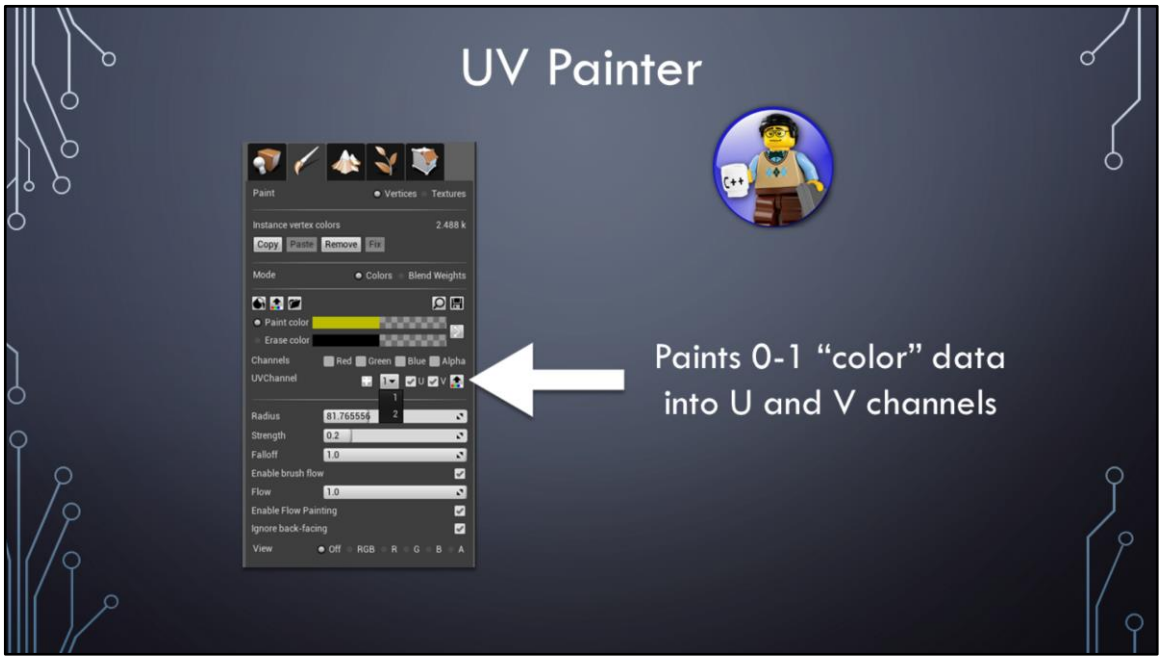
And about 10 minutes later, he had.

He added a checkbox to the Vertex Painter called “Enable Flow Painting”.

Essentially, it just converts the paint brush movement into tangent space, and then figures out what color to paint based on direction.

The result is fully interactive in-engine flow map editing. It’s still fun, every time I use it.

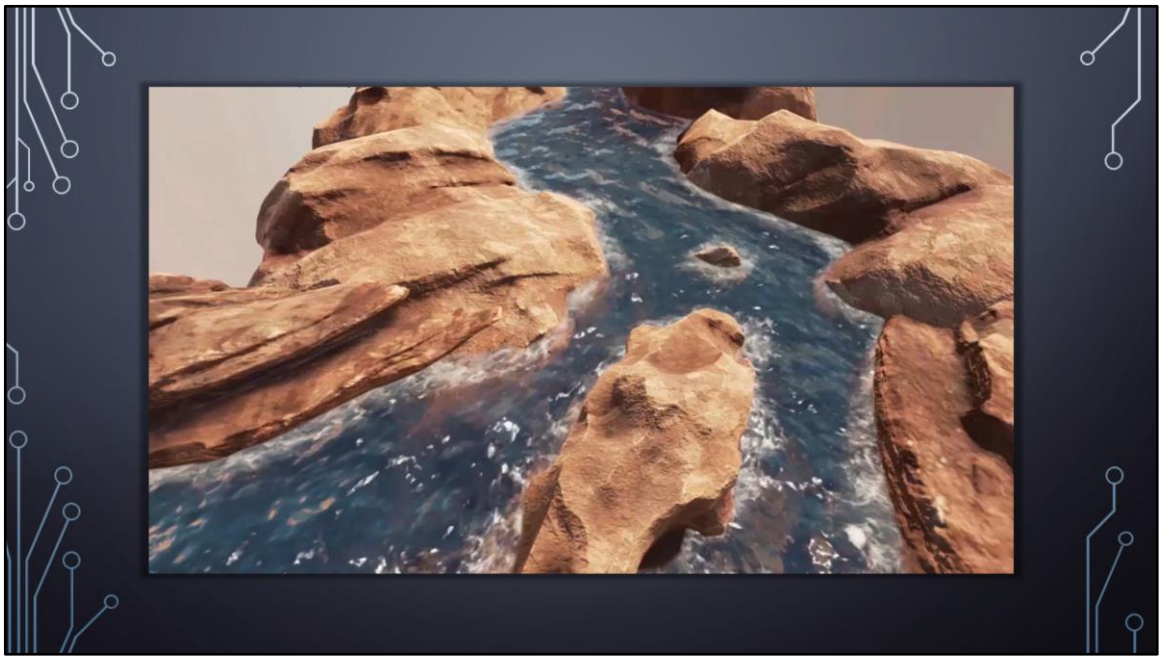
Here’s a look at what it’s doing to the vertex colors.



So then we took it a step further.

What if you have a system of water and you want to switch between multiple flow maps, but still store everything in the vertex data?

Jason expanded the vertex paint tool even further, and added the ability to paint arbitrary values into extra UV channels.



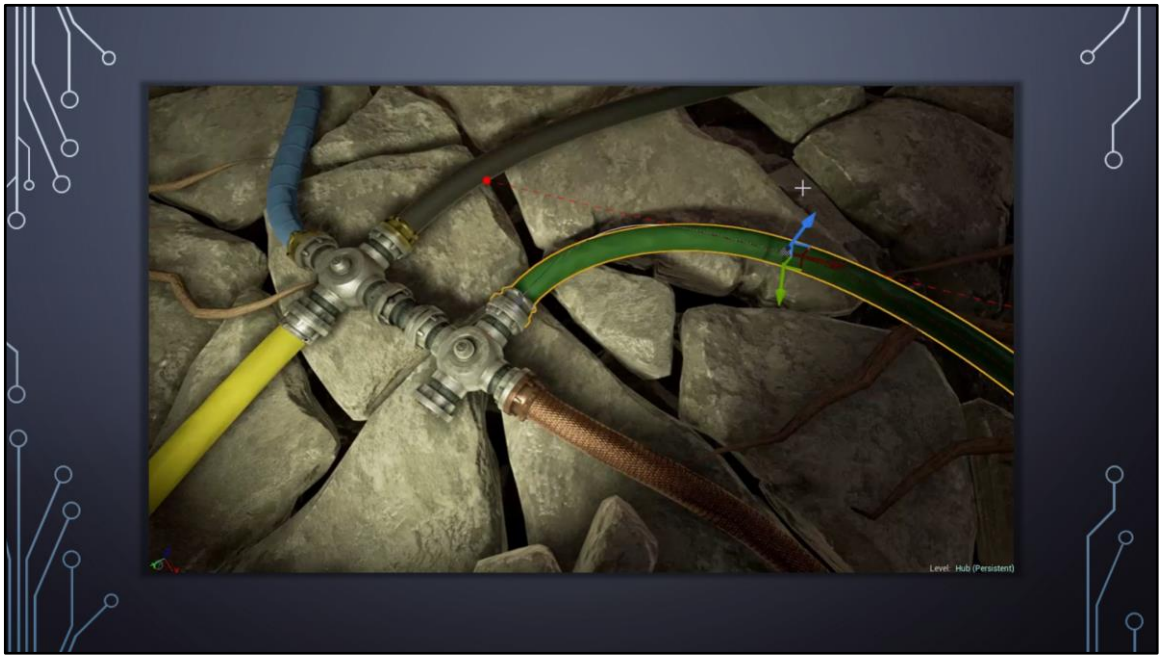
This - coupled with the Flow Paint tool - allow us to pack multiple flow states into a single mesh, and switch them on the fly in the shader, which you can see here.

This is literally just lerping between different vertex painted data.

We even use the UV-Paint feature in other places to get access to more vertex paint channels than we can with just RGBA by itself.

Okay, last up... A couple of quick notes on Spline-based Tools.

We use custom spline tools a LOT in this game.



For obvious things like hoses...

power lines...

Railroad tracks.

And everything you see here is pretty run of the mill...

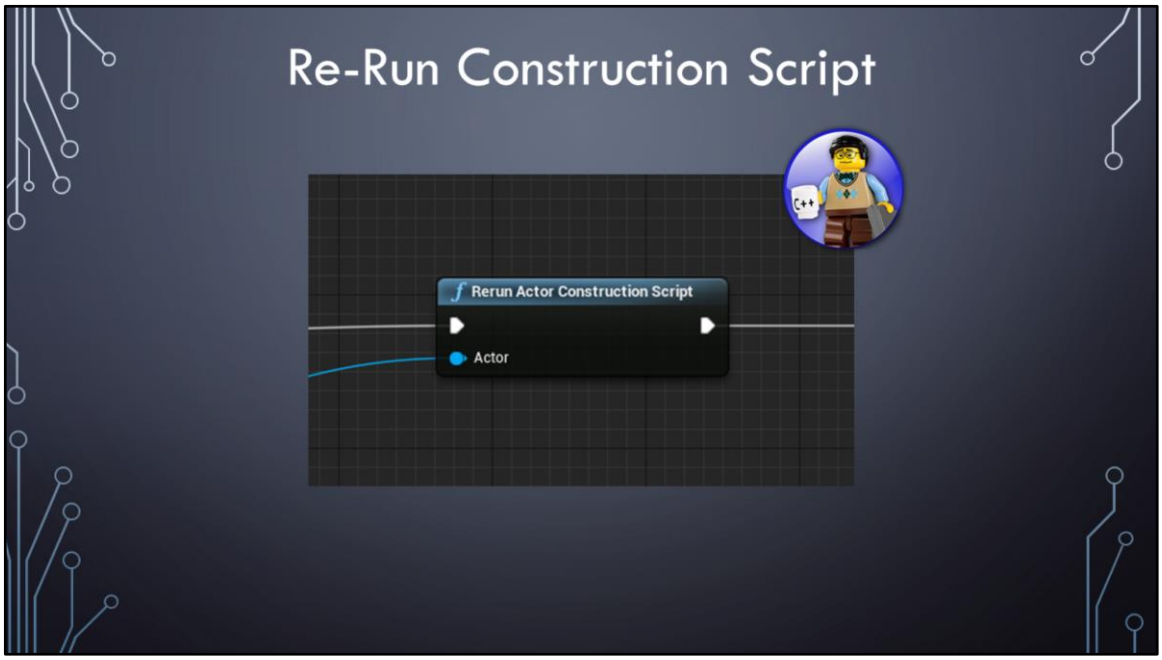
Spline-deformed Meshes and such.

But two little things we added that have been absolute life-savers.

First is Spline Point Snapping.

Essentially, this is just some added functionality in the blueprint that searches for nearby spline points, and then snaps on to them when you get within range.

The guarantees perfectly aligned splines and tangents when building complex networks with multiple splines

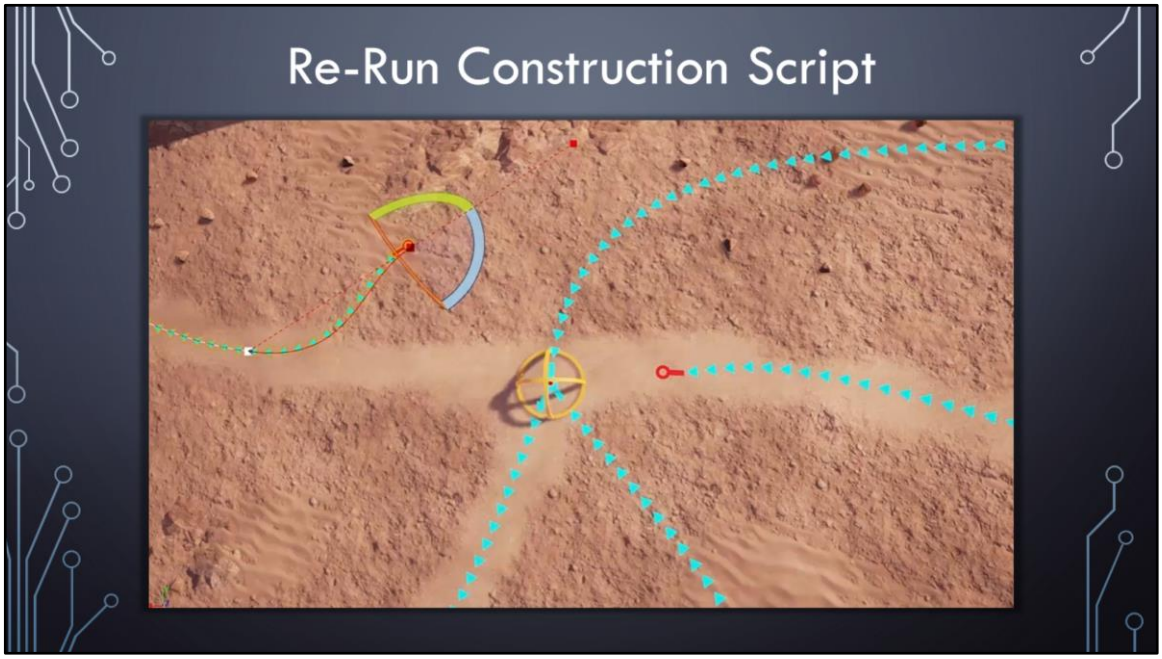


Second, is a blueprint node we added called “Re-Run Object Construction Script”.

And that’s literally what it does.

You specify a target object, and it just kicks its construction script.

That’s useless by itself, but if you first pass along new variable settings to those target objects, you can do things like this...



This is one of our spline-based player-navigation networks....

It’s comprised of spline objects and nodes, but each element is a separate object.

In this system, the splines snap to the nodes... and if you move a node, it automatically updates the construction scripts of all splines that are attached to it - in real time.

Okay - That's all the time I have!

Thank you so much for coming to this talk!

If anyone has any questions or feedback.... Please feel free to contact me online! Here's my info.

Have a great GDC!