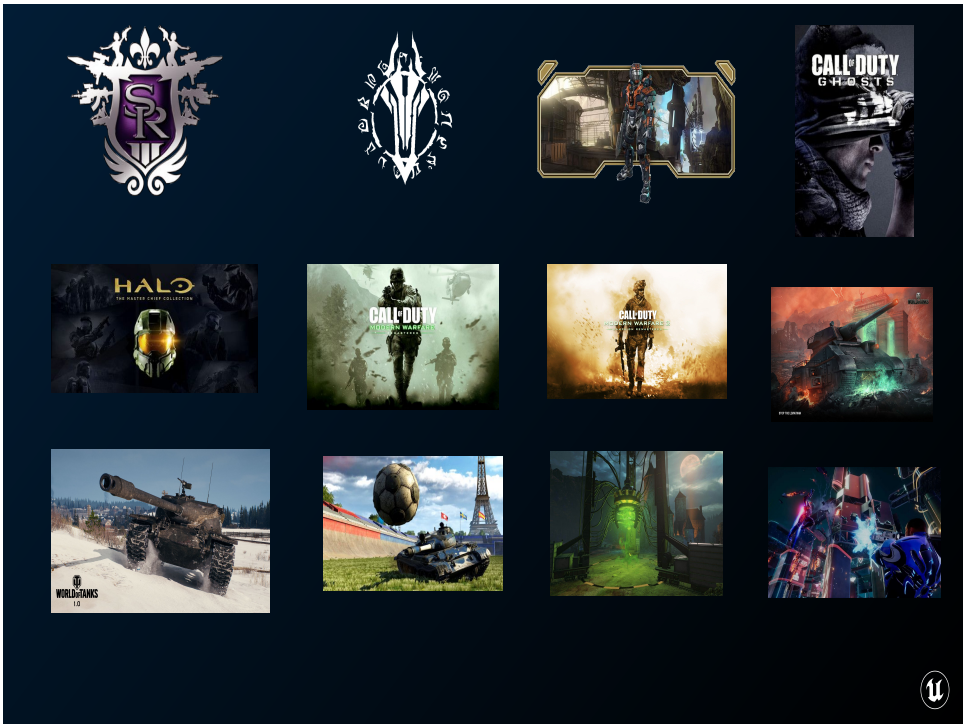
Hello everyone! Welcome back to the Moscone Center, it's so good to see you all again. My name is Matt Oztalay, I'm a Developer Relations Technical Artist at Epic Games, and today I want to talk to you about Bringing the World to Your Shaders. Your materials don't have to exist in a vacuum, and you can do some really awesome things you can do with your materials once you start passing in more information about the outside world. I'm gonna show you some examples of things you can do with your materials with that outside information, and a few different ways to get information into your materials as well as talking about their potential benefits.



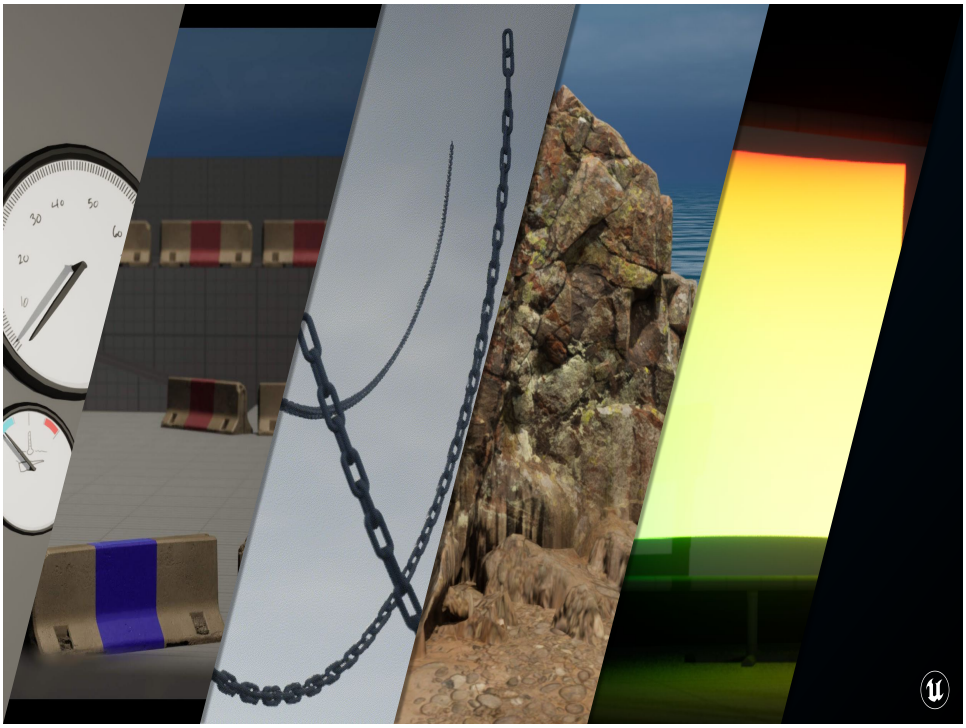
But first, a little bit about myself: I can't quite say "I've worked in the games industry for 11 years", since I'm not actively shipping titles anymore, and my job at Epic has me working with users of the Unreal Engine both in and outside of the games industry. It's probably accurate to say that I developed video games for nine years, shipped over a dozen projects across six different game engines (with another two or three either tangentially or in prototype), and through all of that I supported and enabled art and design teams to do their best work.

And a couple of years ago I joined Epic Games as a Developer Relations Technical Artist, and now I get to empower all users of the Unreal Engine push the boundaries of the technology through exploration, education, and advocacy.

I’m going to be presenting these techniques using the currently-in-preview Unreal Engine 5 using both our node-based scripting and node-based material systems. There’s gonna be two parts to each example, the first being how you get the data into the shader, and the second being what you do with it. The former will be using systems specific to the Unreal Engine, that exist in both 4 and 5. What you do with the data in the material should be more broadly applicable to any kind of shading language.

There is one thing I’ll show you in these examples that is easier in Unreal 5 than it is in Unreal 4, but it’s not impossible to do in Unreal 4. Otherwise all of these techniques are available in both versions of the engine.

(As an aside about the Unreal Engine, there’s some important distinctions between Shaders, Materials. As much as I will try to use the correct terminology (which in our case is Materials), I’ll sometimes slip and say Shader, and I’m sorry).



2:00

I’m gonna show some examples today that demonstrate five of the methods available pathways of communication between the world and materials.

I’m gonna show how to animate the dials on a car dashboard with Dynamic Material Instances, which allow us to change texture, vector, and scalar parameters on a material at runtime.

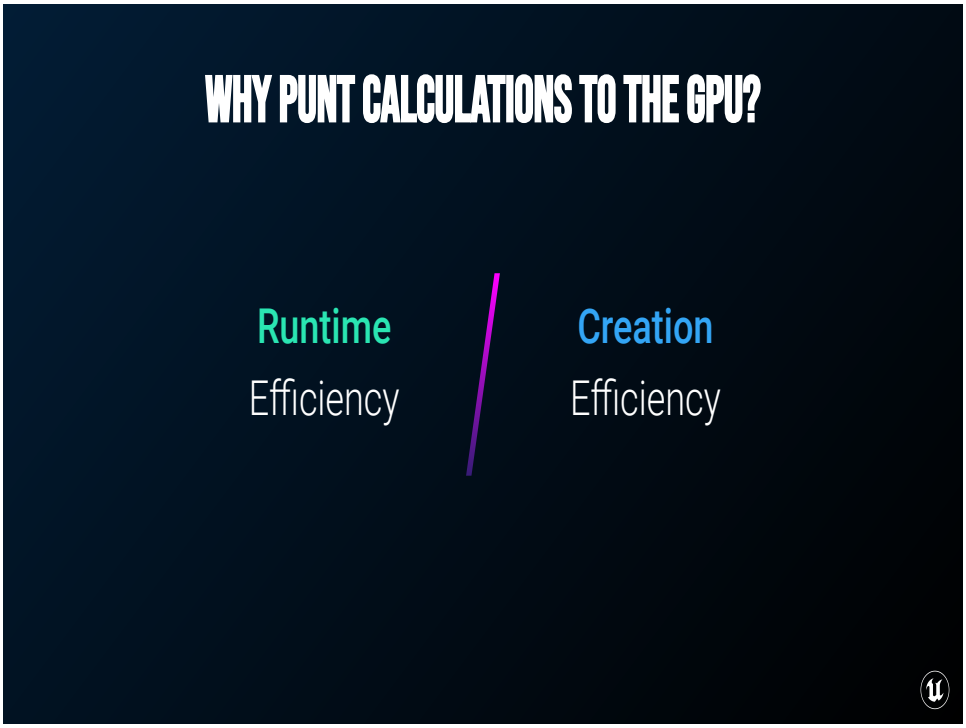
Then we’re going to vary team colors on an multiplayer shooter’s bases using Custom Primitive Data, which lets us store scalar and vector values on the geometry.

Following on that we’ll drive a swaying chain with information provided by Per Instance Custom Data

Then we’ll blend decoration into the landscape using Runtime

Virtual Textures.

And finally I'll show you how to project a texture into worldspace using Material Parameter Collections, which are global structs of scalar and vector parameters that can be read from a bunch of different materials.



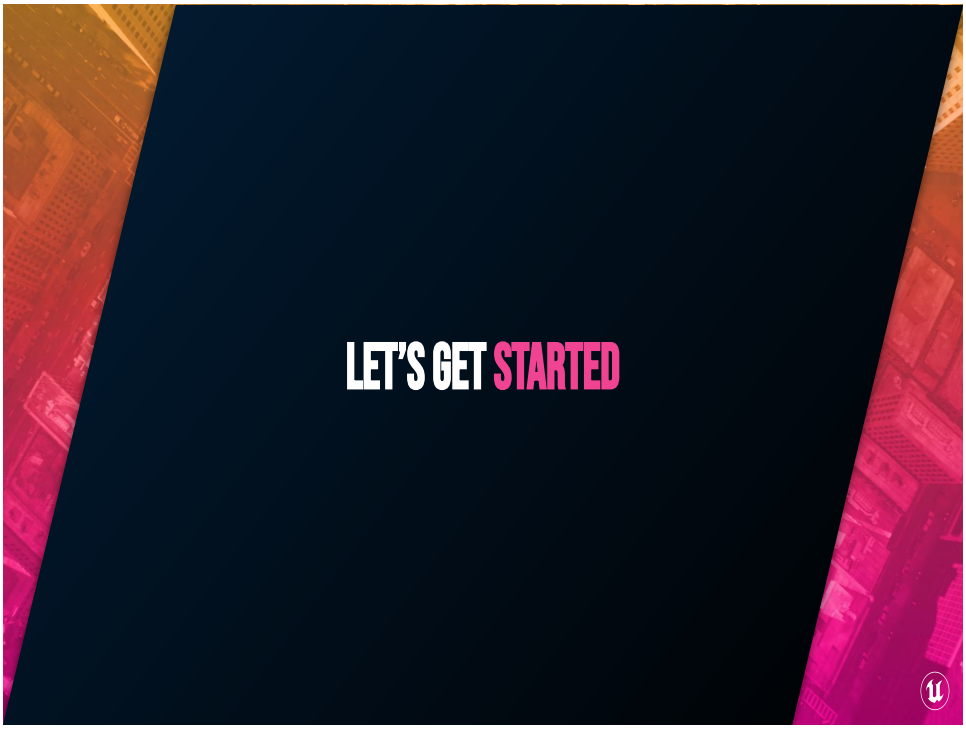
For some part of each of these examples, you'll notice that we're going to be shifting some calculations from the CPU To the GPU, and you might be wondering why.

Some of these examples are going to help us improve runtime efficiency. Depending on the situation, there can be a lot of performance benefits to moving certain kinds of calculations off to the GPU. And in one example, we'll end calculating a value on the GPU just once and reusing it over and over again.

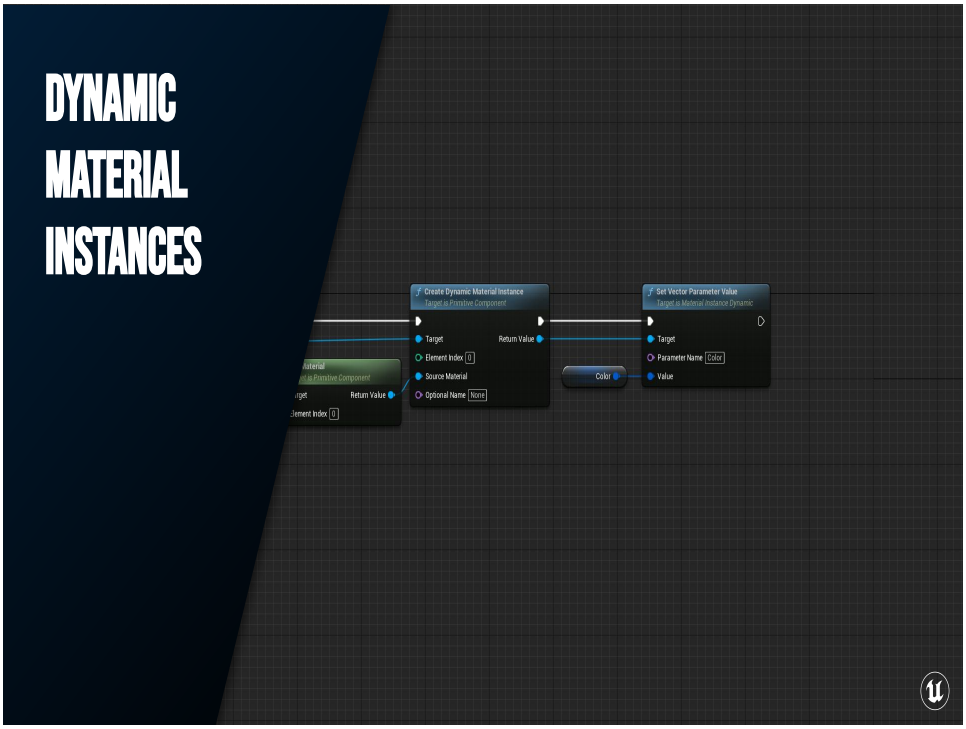
The other benefit to many of these techniques is the efficiency of creation. Many of these techniques will make it easier to create content by reducing iteration time, reduce the number of assets we need in the project, or make things Just Work.

I want to apologize in advance for glossing over all the implementation details. We just don’t have the time to do a full step-by-step tutorial for each of these techniques, so I’m going to focus on the specific parts of getting information from the world into a material and doing cool things with it.

But I do have some good news for you! We will soon be launching new community channels for UE5, and once we do you'll be able to download the project files and read more about the different techniques there!

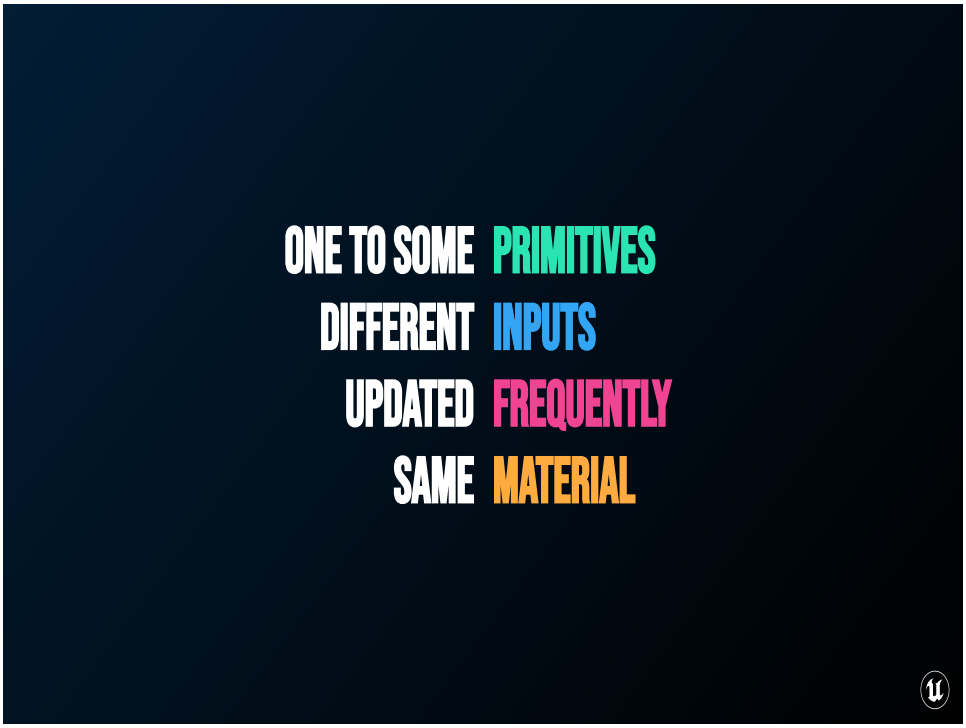


With that out of the way, let's get this show on the road!



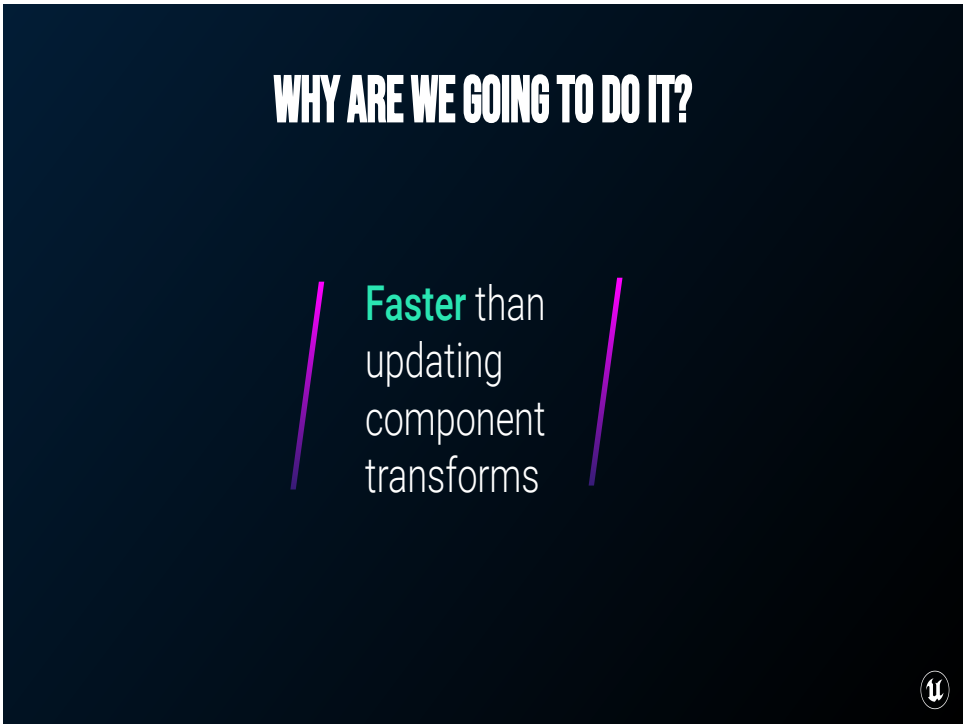
5:00

Let’s start with something that’s fairly common in the Unreal Engine. You can create dynamic instances of materials that support changing the parameters (or inputs) to those materials during runtime.

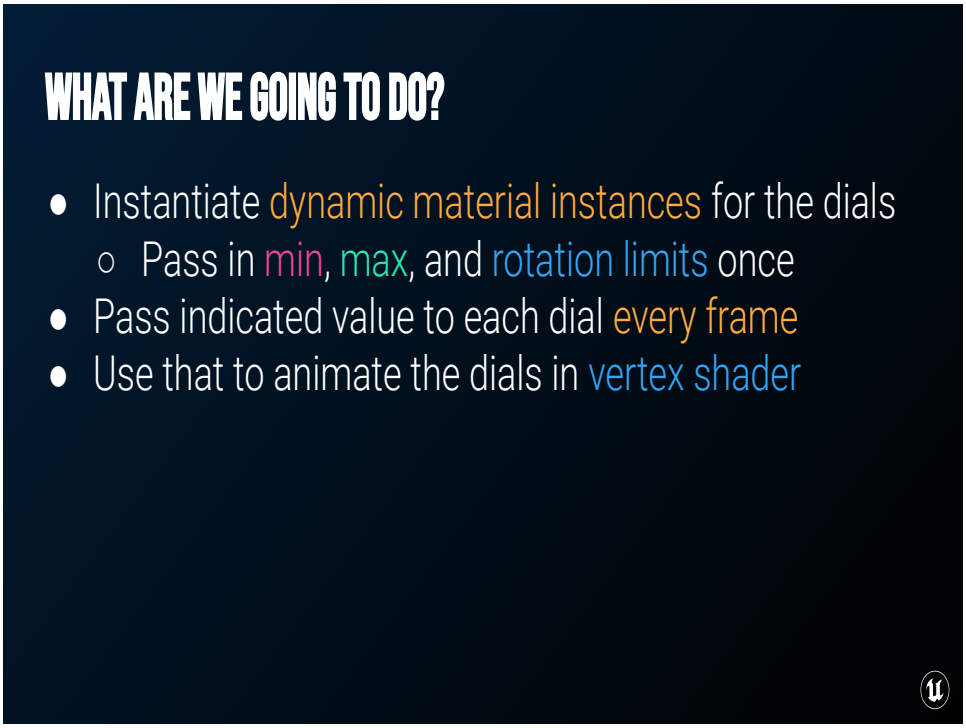


Dynamic material instances are great for situations where you have one to a few things that use the same material and need to frequently react to different inputs.

And to demonstrate that, we're going to animate the dials on the dashboard of a car just with their materials. I've set up a few different dials in the Advanced Vehicle Template that ships with Unreal, and we're going to animate each of these using world position offset (effectively an additional function run on top of the vertex shader). Each of these shows a different value, because we've got a Speedometer, Tachometer, Fuel gauge, and Temperature gauge.



So the traditional way of doing this, or the more straightforward, CPU-driven way would be to take all that information and update the position of the component every frame. That, however, can be rather expensive on the CPU, especially if we're trying to do something like this on a LOT of these types of components. Importantly, we don't care about things like the collision of the dial and there aren't any systems that need to look up the rotation of the dials. So instead, we can offload that rotation and those calculations to the GPU.

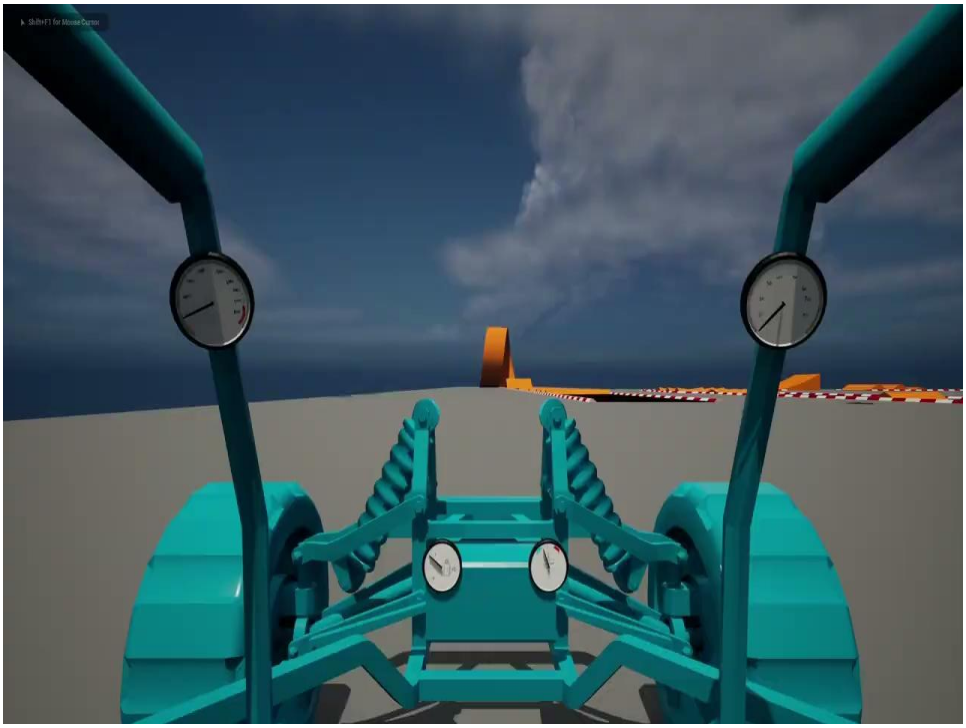


To do that, we'll first instantiate the material for each of the dials and set it up with a few one-time parameter updates for the min and max values of the dial, as well as its rotation limits.

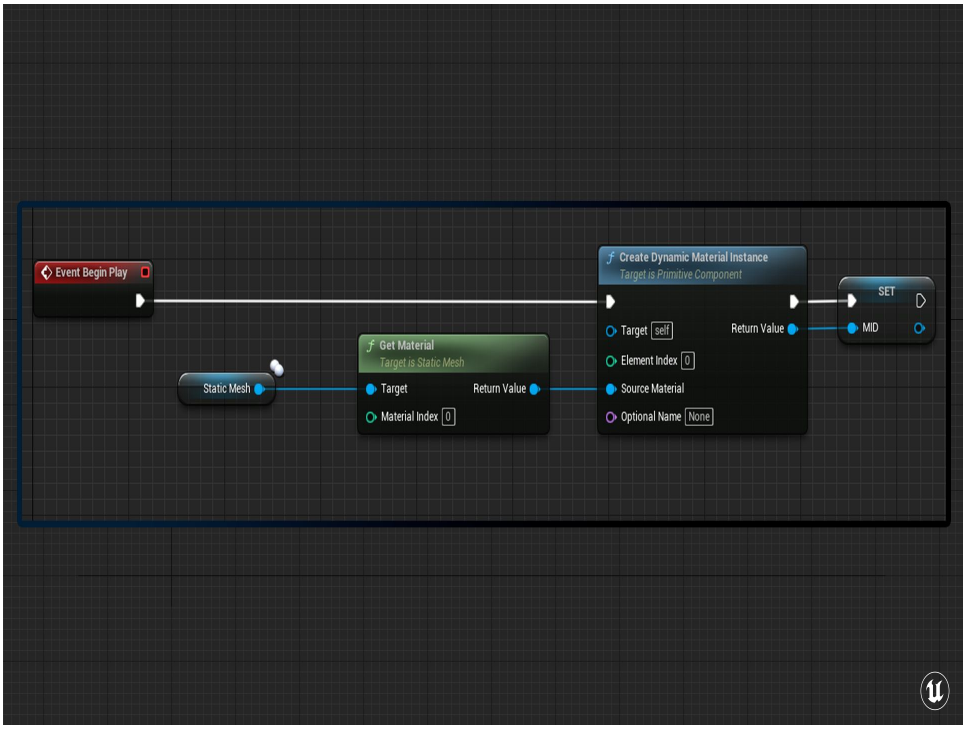
Then we'll pass the associated information into each of the materials. So for example an oil temp indicator might only use about ¼ of a full rotation to display values between 120 degrees and 320 degrees, but a tachometer can go from 0 to 10000 rpms over ¾ of a full rotation. Even though we could calculate this on the CPU and just pass a single "rotation amount" value to each of the dials, I'd much rather handle that calculation a few times on the GPU instead. My goal here is to take as much work as possible off the CPU.

Finally, in the material, we'll figure out the appropriate rotation for each of the dials based on all those input parameters, and rotate it using the vertex shader.

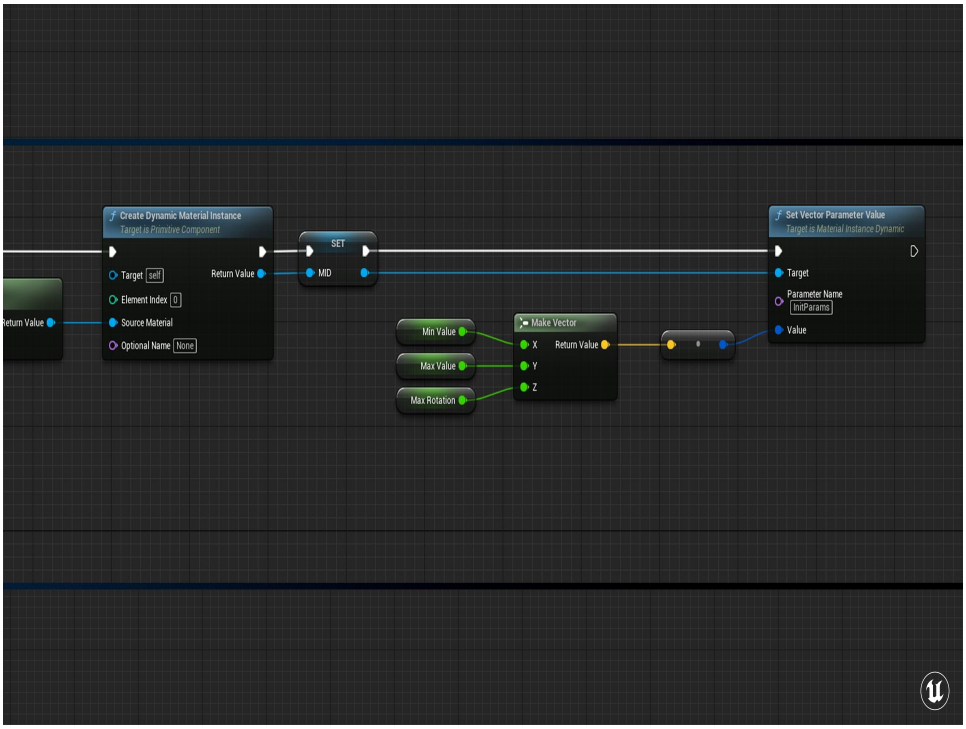
I’m going to show you what this all looks like put together, and I’m going to warn you that the video may get a bit motion sicky. I’ll let you know when it’s over.



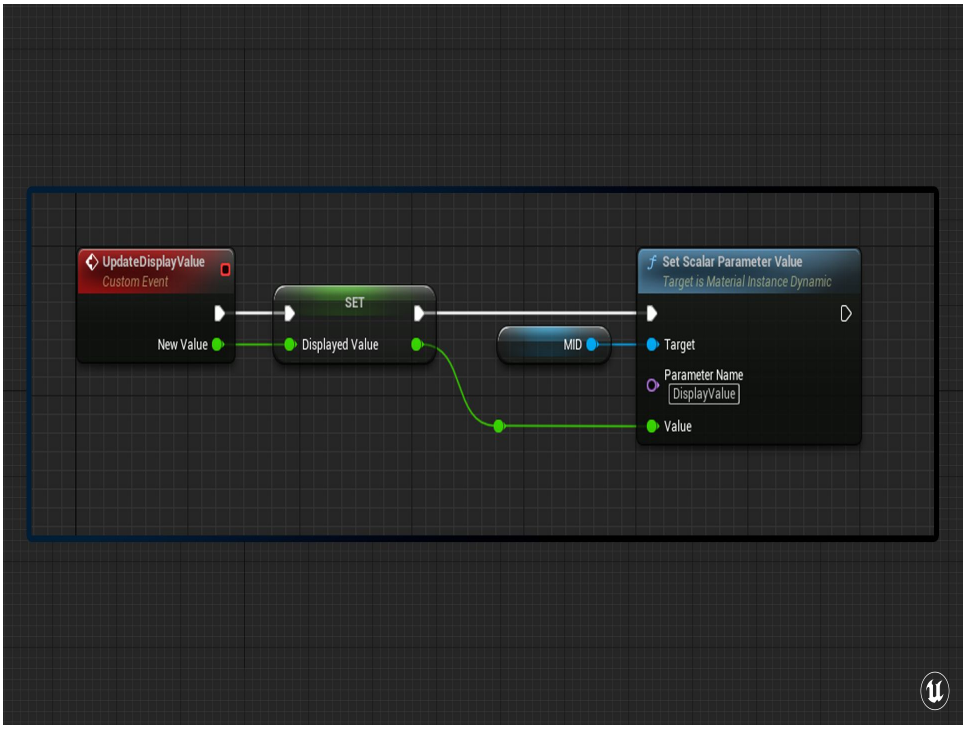
And to give y’all an idea of what this looks like when we’re all said and done, here’s this set up in the Advanced Vehicle Template that ships with the Unreal Engine



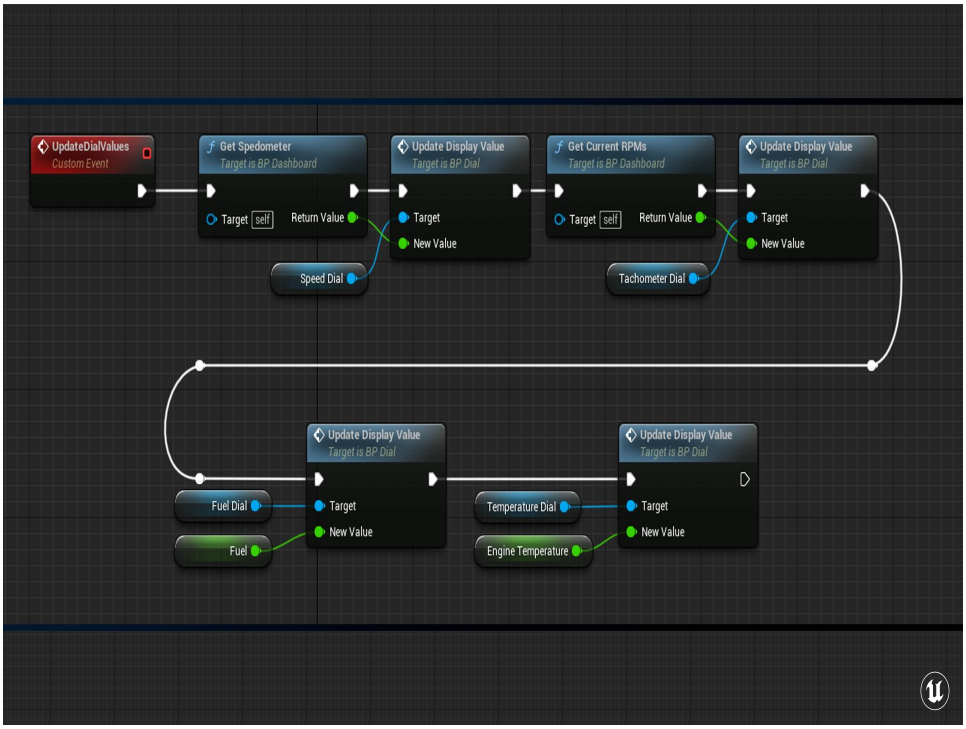
Firstly, each of these dials is its own separate component that handles passing information to its material. On begin play, I instantiate the material and store it for later retrieval.



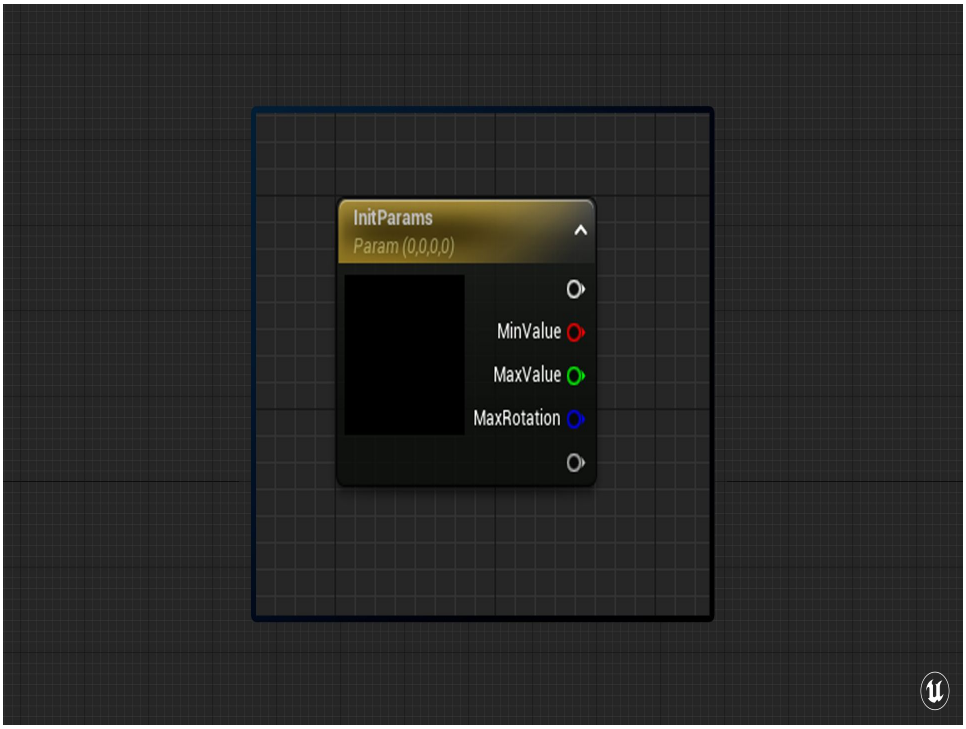
Then pass in some one-time parameter values. To make things a little more efficient, we pack the Min and Max values, and the rotation limit into a vector that we can pull apart inside the material. We're just calling "Set Vector Parameter" on the material instance and setting the name to one we'll set up in the material.



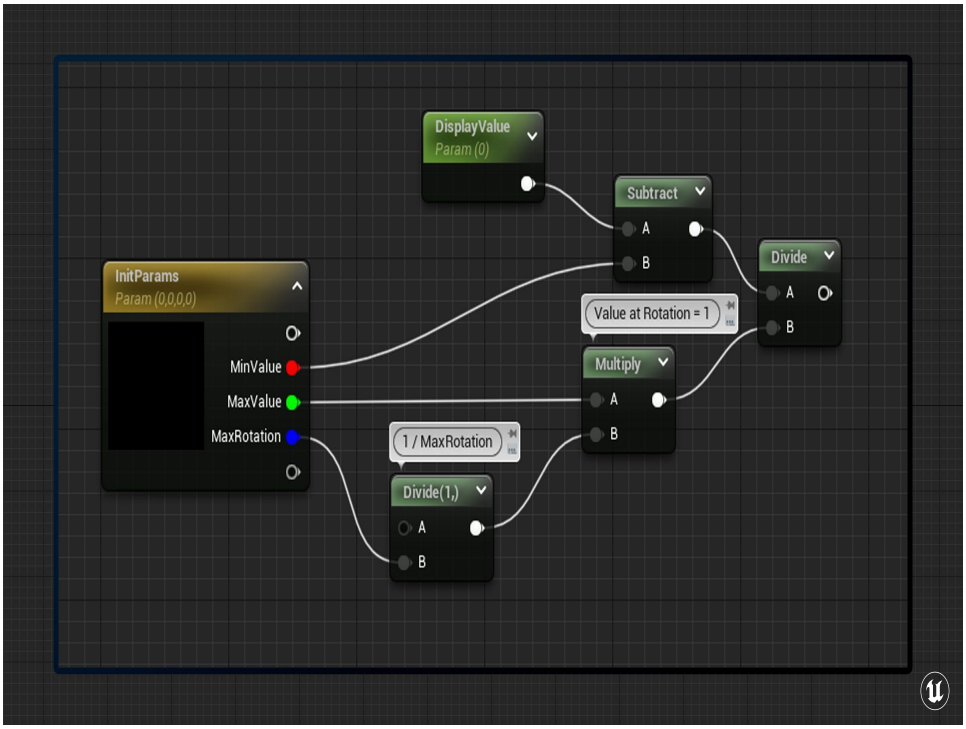
Then this blueprint has a function that passes an input “Display Value” into the material instance as well. All these dials are going to use the same parameter name, DisplayValue, because they don’t care what they’re displaying.



Then, in the dashboard Blueprint that handles all of the dials, we pass in the appropriate values for each of the different dials. RPMs to Tachometer, Speed to Speedometer, etc. This gets called every frame to keep those values up to date.



In the material for the dial, I've got a vector parameter set up, and to help myself I've named each of the channels to match what I passed in from Blueprint. You don't have to do this, I just find it helpful. This parameter has the same name as the one we're using on begin play!

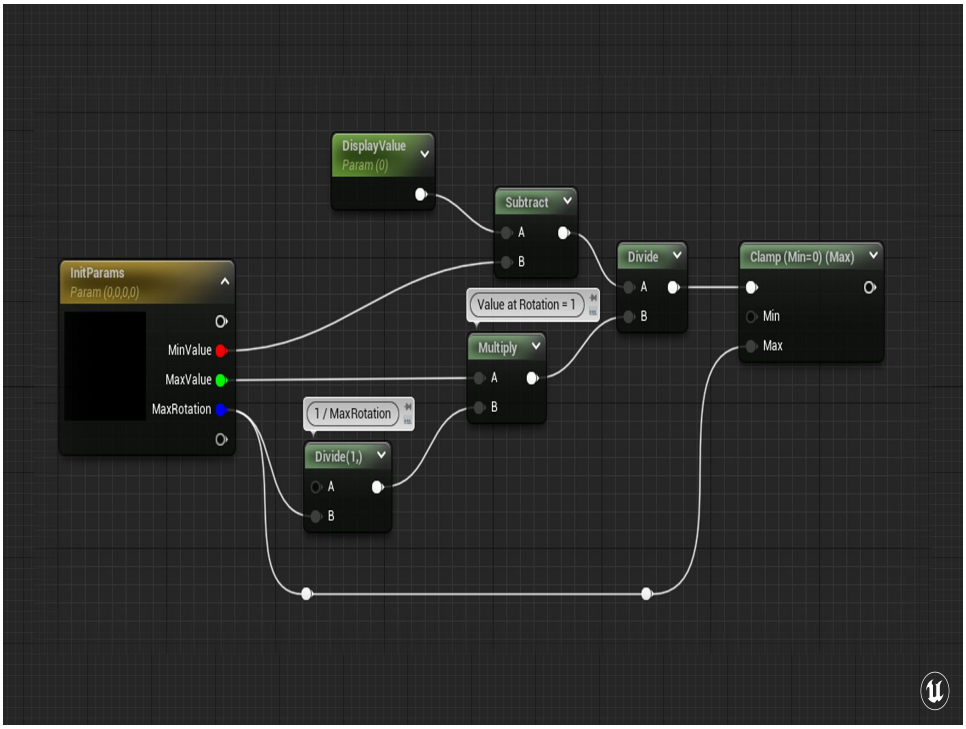


We're going to rotate the indicator with World Position Offset, or the Vertex Shader. And to do that we need a rotation value between 0 and 1. Sort of!

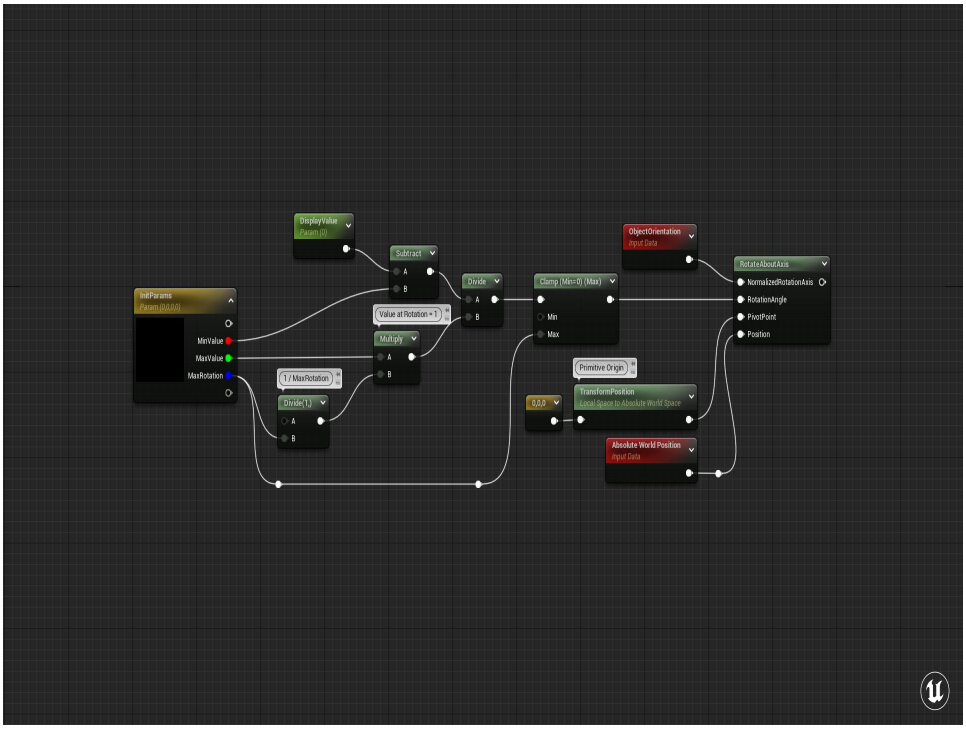
First we need to normalize the input display, but the Min and Max values correspond to, respectively, 0 and "RotationLimit" (which is a value less than 1) instead of 0 to 1.

(For example, the oil temperature displays values from 120 to 320, but it only rotates about ¼ of the way all around.)

We want to normalize the input DisplayValue between 0 and Max * 1/RotationLimit, because Max * 1/RotationLimit gives us the value that we'd display at a rotation of 1.



Then of course, just to make sure that we don’t rip the dial out of its socket, we’ll also clamp that value between 0 and RotationLimit.



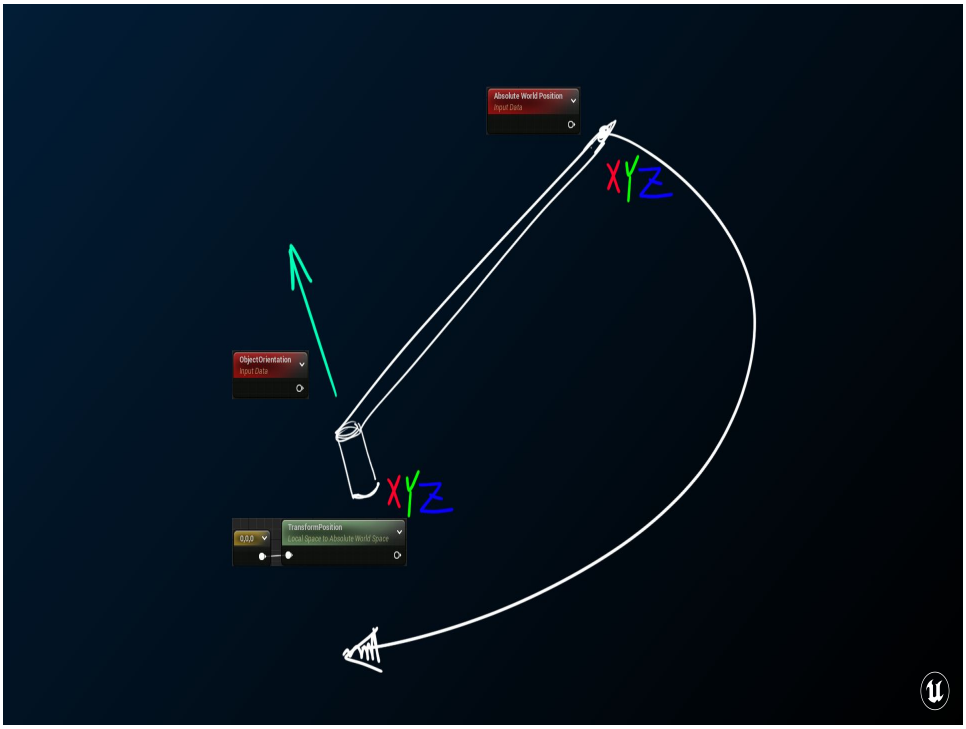
Now that we have a rotation value in the 0 to 1 range, we can use the RotateAboutAxis node to figure out how much we need to offset the position of each vertex.

RotateAboutAxis is great for giving you the delta values you would need to rotate any given vector a given % of a full rotation around a given axis from a given pivot point. I'll have a slide that covers all the math that's happening in this node in the appendix for y'all to check out later.

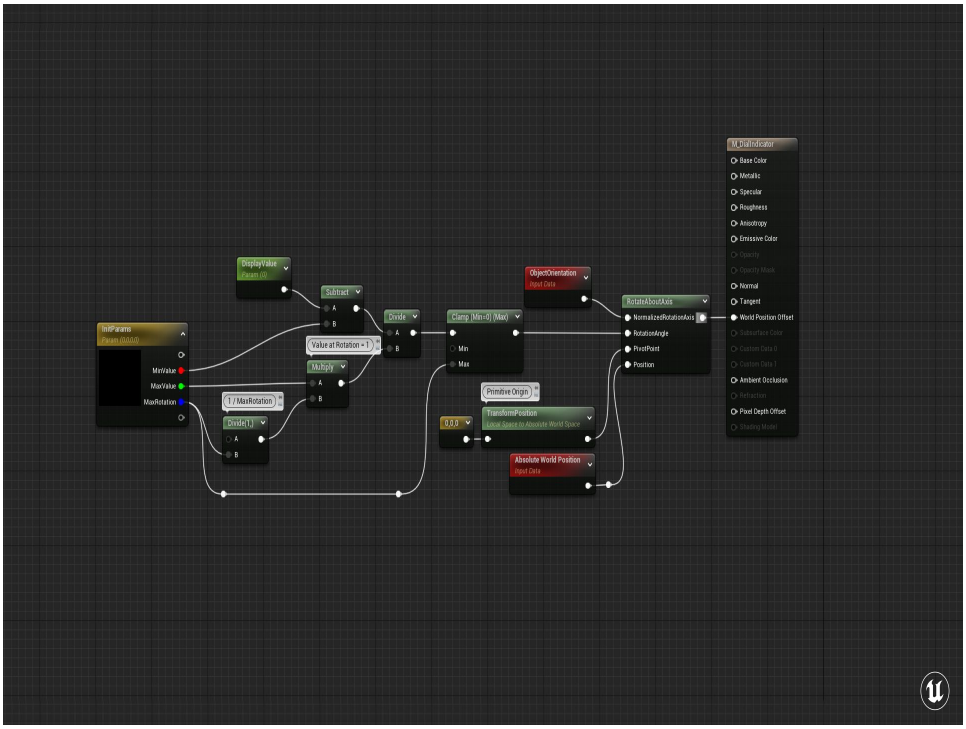
For this, we'll use the ObjectOrientation as our rotation axis, that's the vector the current primitive is pointing in worldspace.

For the pivot point, because this dial is embedded inside an actor what we really need is the origin of the primitive itself, not the actor, nor the ObjectPosition since that's the center of the bounds. So to get what we really need here, we'll transform 0,0,0 from LocalSpace to WorldSpace.

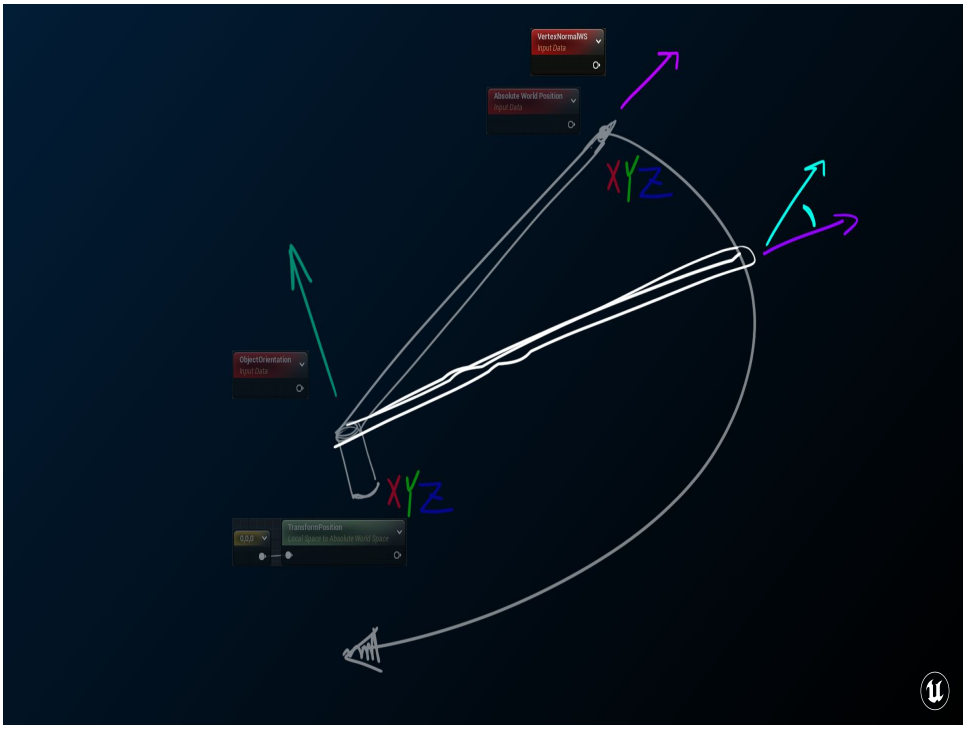
Then the position we're going to rotate is the Absolute World Position input. This is the location of the vertex.



To draw that out, we need to rotate this dial (at each vertex), from the worldspace origin of the primitive, around the axis ObjectOrientation, by the amount we calculated earlier.

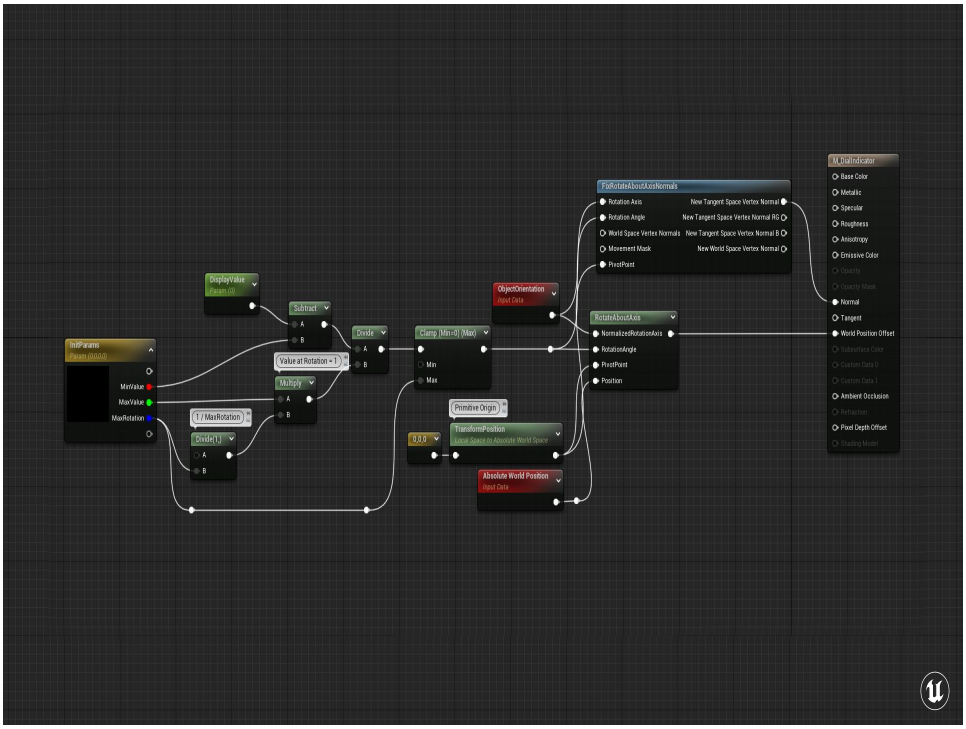


Then we just pass that output of the RotateAboutAxis node to the WorldPositionOffset input of the main material node. For Unreal, we run a standard vertex shader for each vertex that puts it at the position encoded in the geometry, and we have the option to offset that position in worldspace. That’s all we’re doing here!



Of course if we update the position of a vertex, especially if we rotate it, then its normal is going to be pointing in the wrong direction. That means it's going to light and shade all wrong.

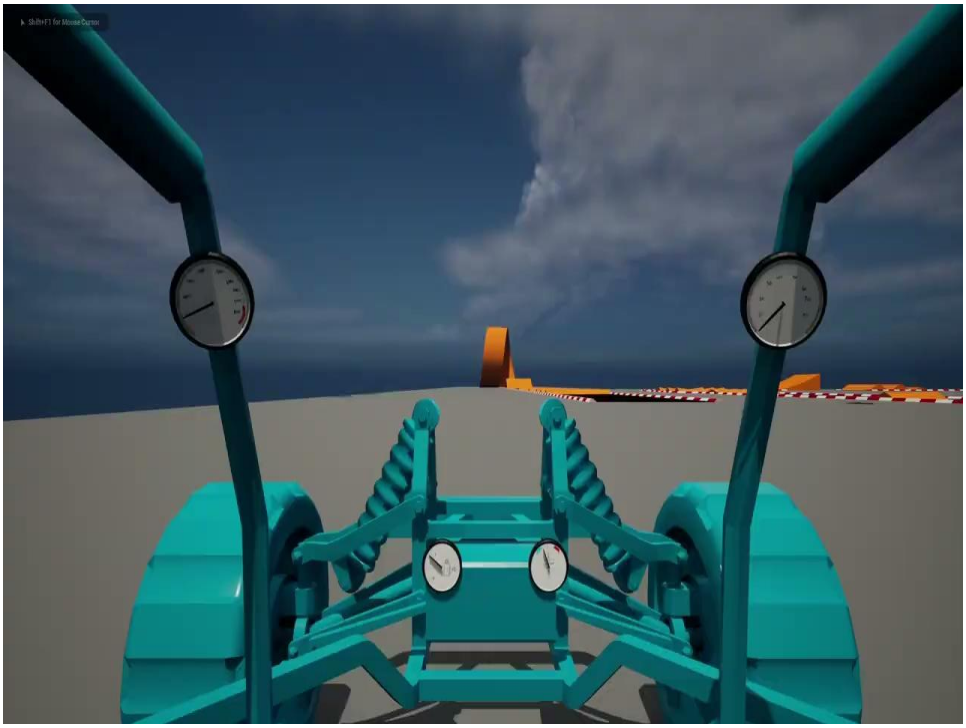
For that, all we need to do is *also* rotate the vertex normal the same way we rotated the vertex itself.



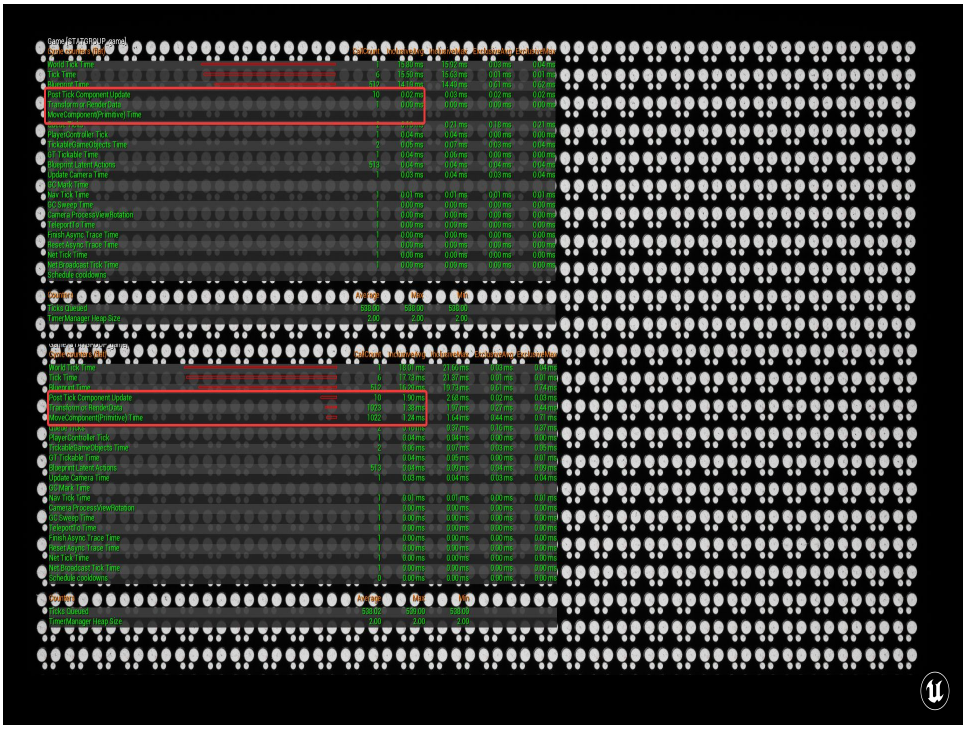
Luckily we've got a node that does that for you in Unreal already. We'll just hook up the required inputs, and pass the new output to the Normal input of the material node. Now our dial normals will be pointing in the correct direction based on how much their position has changed.

Then you can use this as the worldspace base Normal value and blend any additional normal maps you'd need on top of this. I've got some references for how to do that at the end of the show.

Okay, once again a quick motion sickness warning as I show you again what that all looks like put together. I'll let you know when it's over.



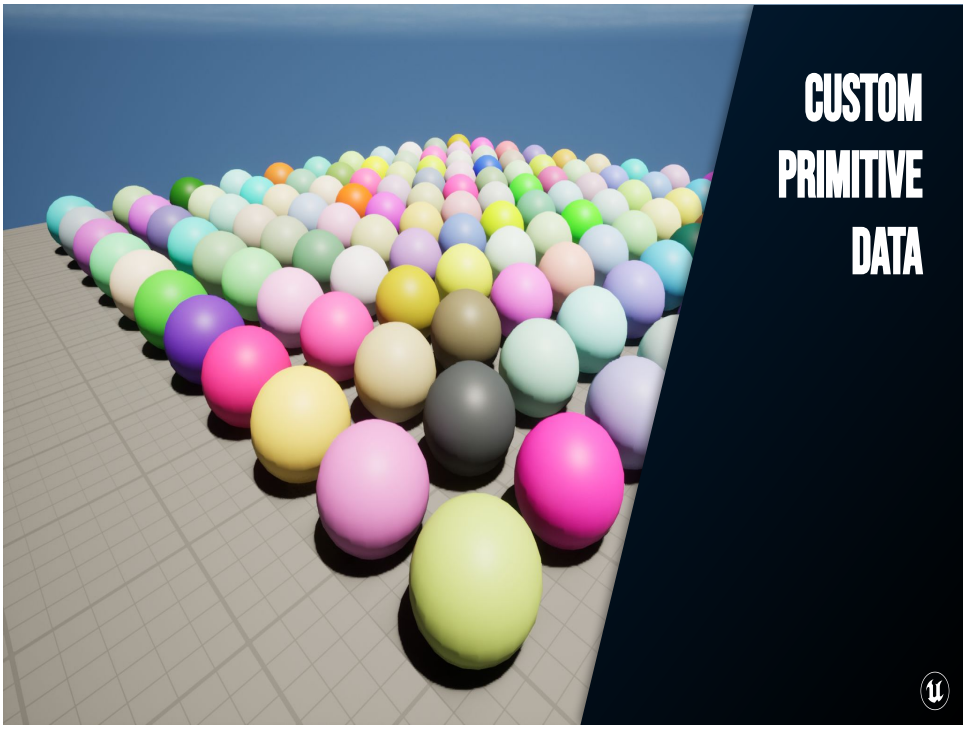
So to loop back again, here’s the final result as we’re driving the car around the map. RPMs go up and down as we shift gears, speed goes up and down as we brake, and I’ve exaggerated the fuel drain so that goes down slowly but surely.



Now, I'd be remiss if I didn't prove out the performance benefits, so I set up my dashboard blueprint to switch between updating the material values and updating the component transforms of the dials. Using the Blueprint method added almost 5ms to the game thread between PostTickComponentUpdate, Transform or RenderData, and MoveComponent(PrimitiveTime). So it takes at least 1.25ms *just* to move the components, even if we optimize away any of its collision or overlaps or anything like that. If I switched back to the Material Instance method those costs disappear!

I spent a little bit of time brainstorming some of the things you can do when you can pass certain simple state information into a material, and came up with quite a bit! This is, I think, the most adaptable technique of the everything I'm going to show y'all today, and I wanted to get your heads in the right place before we start getting deeper.

Take a breathe!



14:00

Sweet, so, we can make the game modify information on a dynamic instance of a material! This is great!

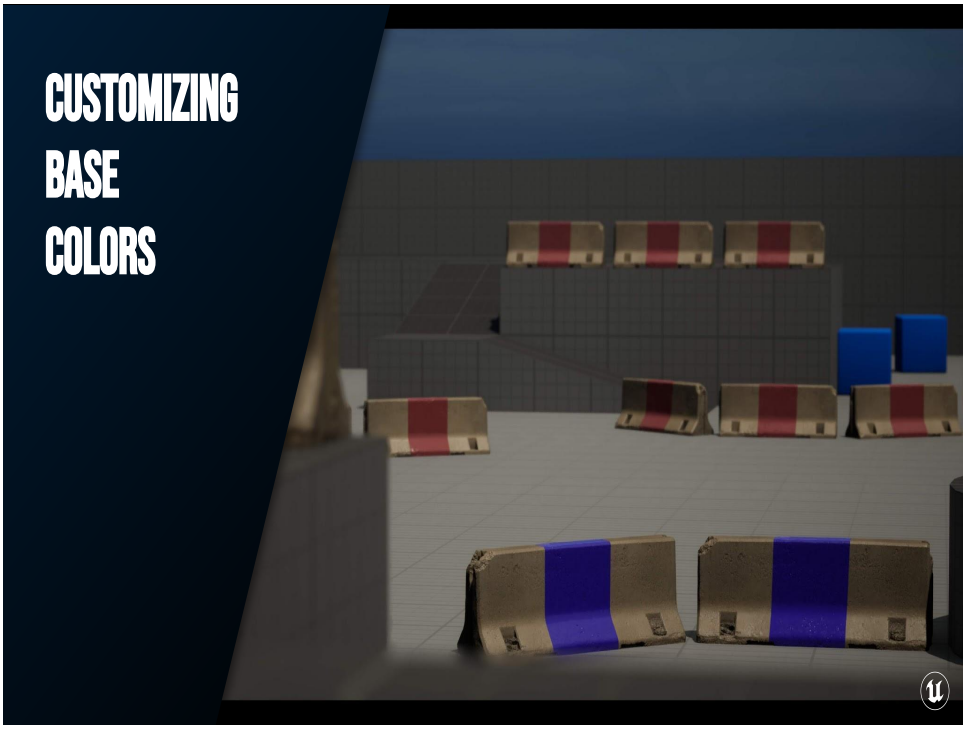
But maybe I don't want to worry about spinning up new material instances and managing them at runtime, especially because each new dynamic instance is a separate draw call even if you account for Unreal's built-in dynamic mesh batching that tries to group same geometry/same material into a single call.

I can use the same material, but modify a value on the *geometry* instead. If I could look that value up in a material instead of doing it on a per-instance basis then I can batch all the same material into one draw call, and I don't have to keep track of a bunch of different instance assets.

In Unreal that's called Custom Primitive Data, and I'll show

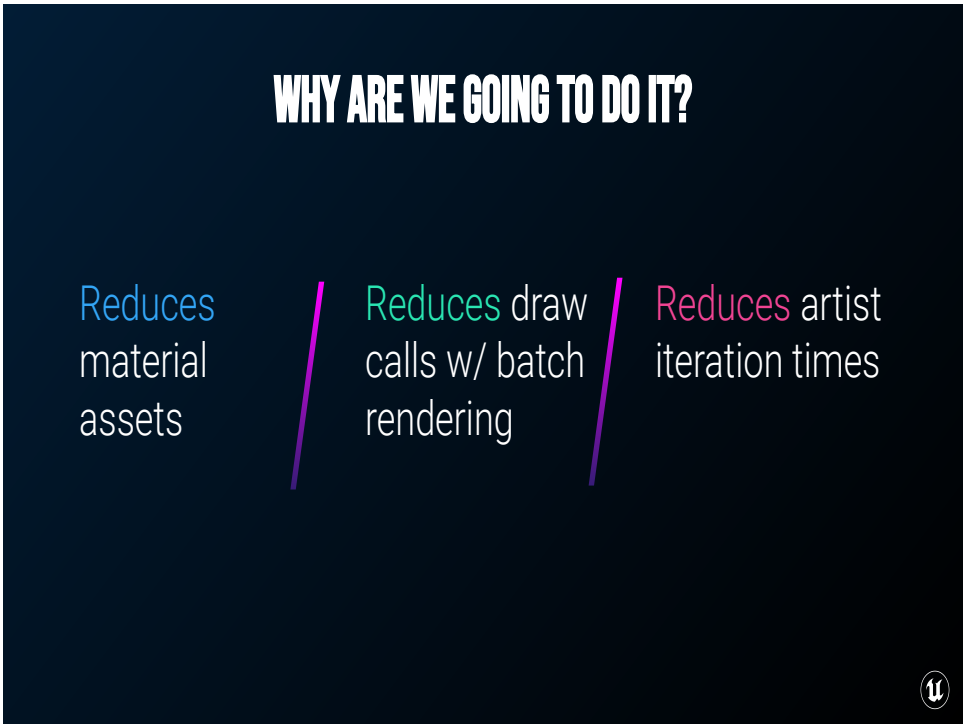
you how that works, too!

Custom primitive data is great for situations where you have many things that need to react to different inputs on the same material, and can be updated at a variable frequency.



To show this off, I've got a simple little FPS greybox map here. (We've all heard of programmer art, but has anyone ever seen Tech Artist Level Design?)

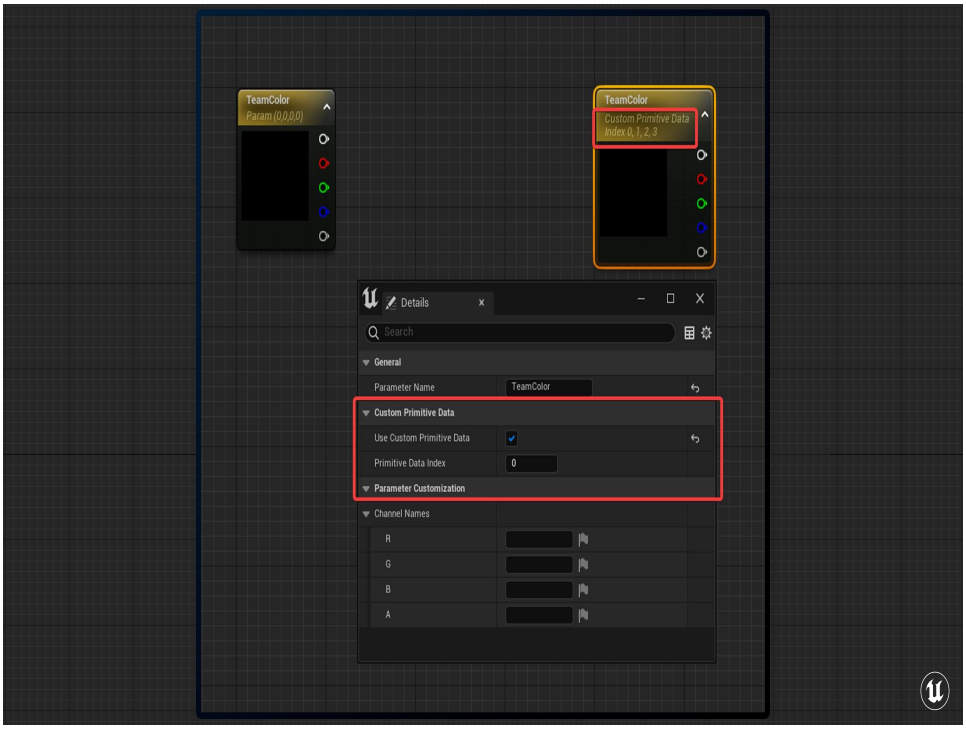
On one side of the map I've got the Blue team, and on the other the Red team and we're going to use a value exposed on the primitive (or geometry) to set the color of the base.



I don’t want to have to set up Red and Blue versions of each of the base construction materials, and I want to keep everything as part of the same batched draw call. I also want my artists to have a bit of freedom in creating their maps. This isn’t just for Red and Blue bases, letting artists set these parameters without having to spin up new material instance assets gives them a lot of freedom, and it reduces their iteration times.

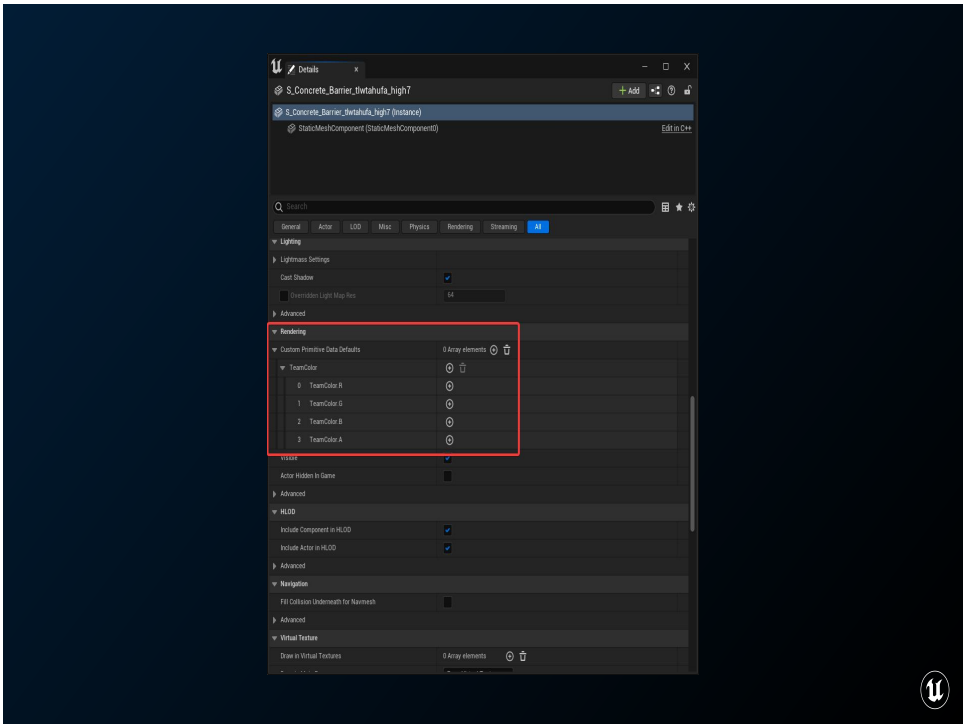
WHAT ARE WE GOING TO DO?

1. Expose material parameters to the primitive
2. Pass values from primitive to material
3. Use exposed parameters to customize asset colors



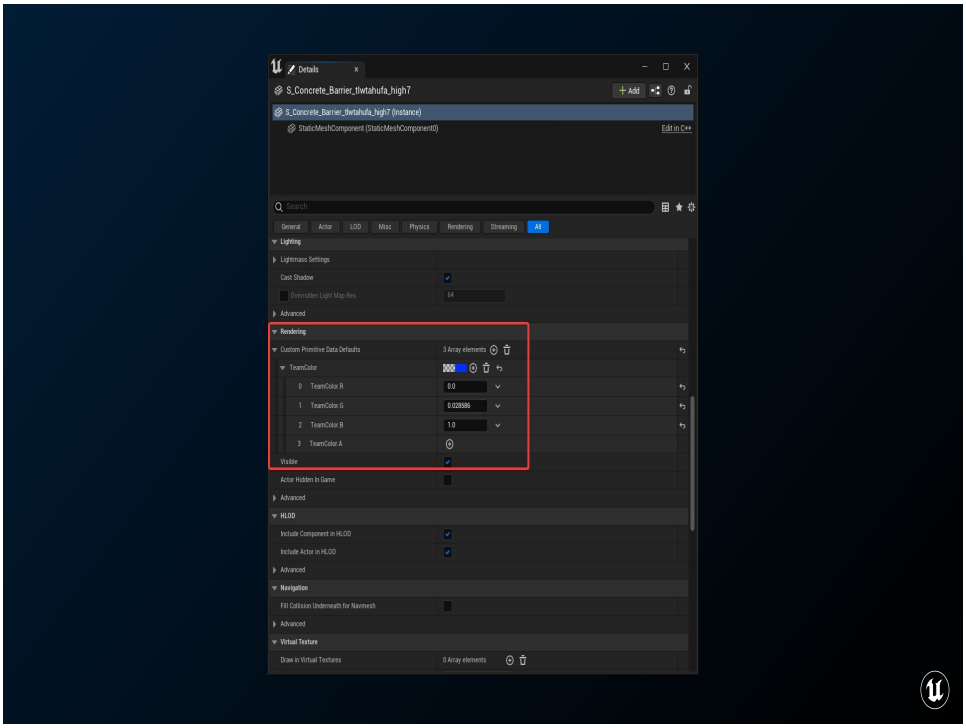
The first step is rather trivial! On the material, I have a vector parameter called TeamColor. I just check this checkbox that says “Use Custom Primitive Data”. This pipes into a mask that overlays the jersey barrier’s base material.

You’ll notice that then on the Vector Parameter it changes to this “Custom Primitive Data Index 0, 1, 2, 3”. That’s because under the hood each Custom Primitive Data is a single float packed into the geometry. Then, under the hood, we can interpret four consecutive floats as a vector parameter.

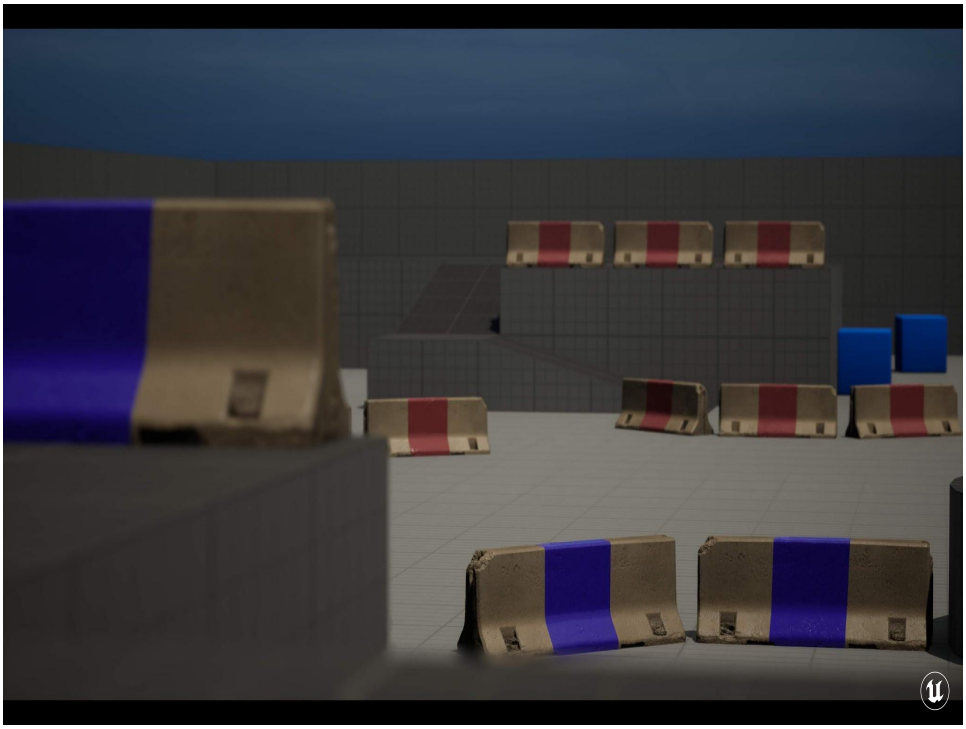


Then, on each of these primitives where we want to use the parameter, we'll add an element to the array of Custom Primitive Data Defaults.

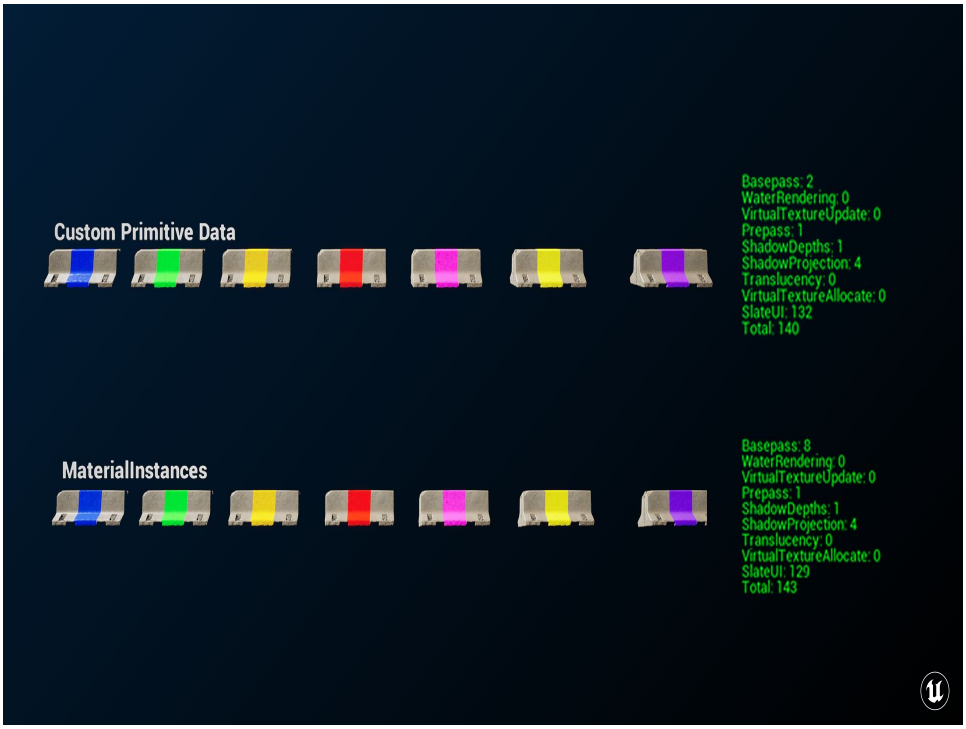
You'll notice that here in Unreal Engine 5, the vector is pre-populated with the name of our parameter from the material. My colleague Alex Stevens added this new functionality for 5.0, and will be available to you all just as soon as we release the engine. In Unreal 4, you'd just add three or four floats here and set each one individually for Red, Green, Blue, and Alpha. In Unreal 5, we can now display these as named parameters and edit them as Vectors with the color picker!



So all I need to do is add three custom primitive data floats, and the engine knows that the first one is Red, then green, then Blue. If I want to save myself one float worth of memory, I can skip adding the four one for the alpha channel because we don’t use that in our material. If we added additional custom primitive data parameters we’d need to add the alpha channel, though, because these indexes are order dependent!

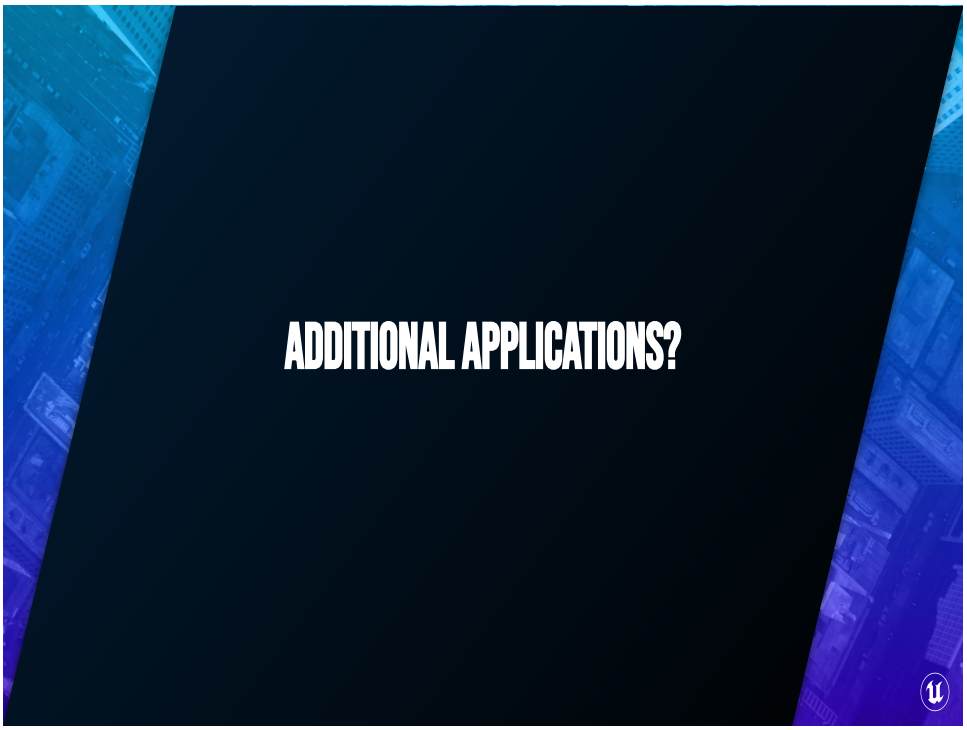


And now I can dupe these meshes throughout the level, and change the team color on them ad nauseum!



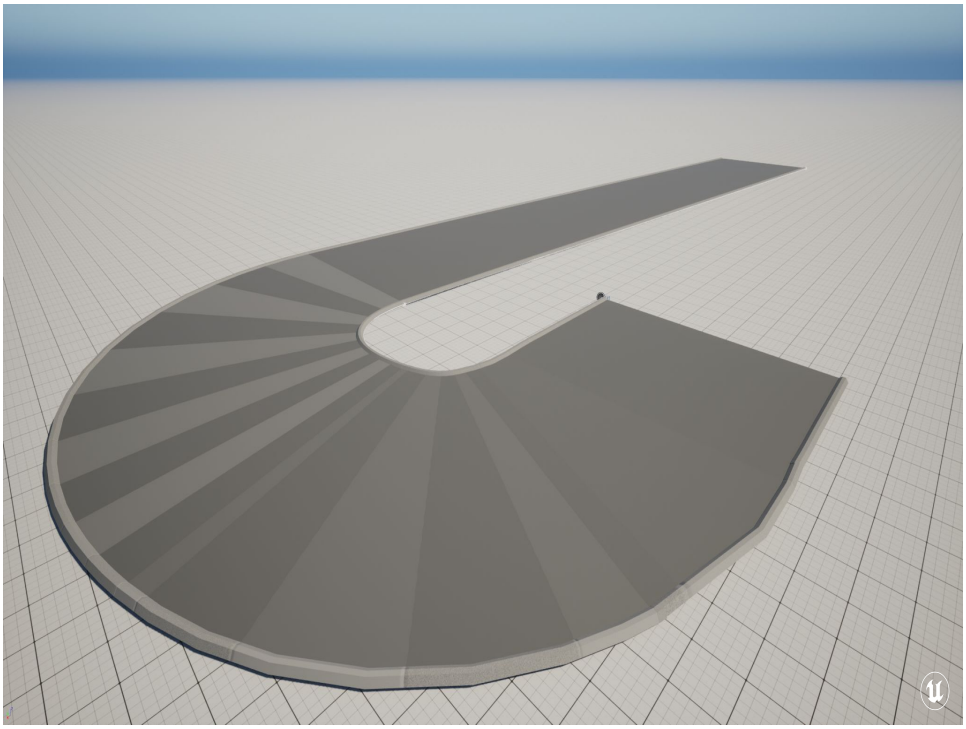
For testing this one, I just created a set of 7 material instances of the barrier material that used a parameter for its color instead of custom primitive data, then I set up 7 barriers below those using CPD, and looked at the difference in the BasePass draw counts which, hey, I think that's a good savings! Part of the reason for that is because of the Unreal Engine's built-in Dynamic Mesh Instancing that looks for same geometry-same material in a given part of the screen and groups them all into the same draw call at runtime.

I'll freely admit this technique isn't all that fancy on the material side of things, but the performance benefits are substantial!



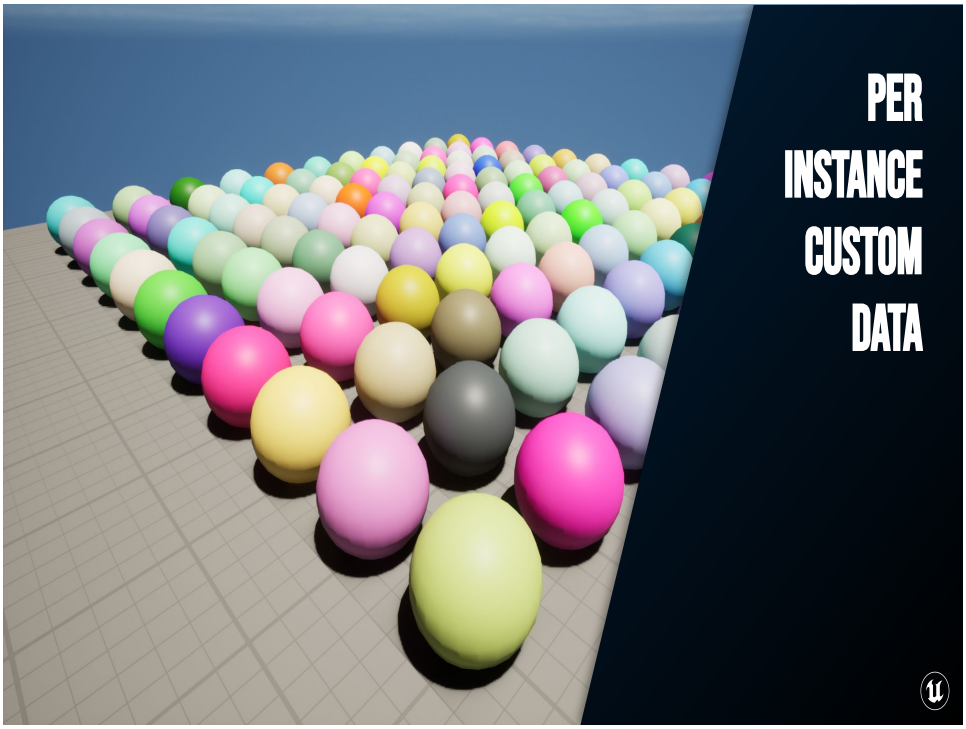
But it’s not just about making team colors on bases, CustomPrimtiveData has a variety of uses and I wanted to show you a couple that have come up over the years.

You can also use it to drive things like wear and tear on an object, Or use it to SubUV select some decals that should be applied to a surface!



Most recently, I used it to pass in a “Bendiness” parameter to a spline mesh race track maker. As the start and end tangents of the section diverge, I lerp in that little stepped pattern in order to let players know that the track is curving.

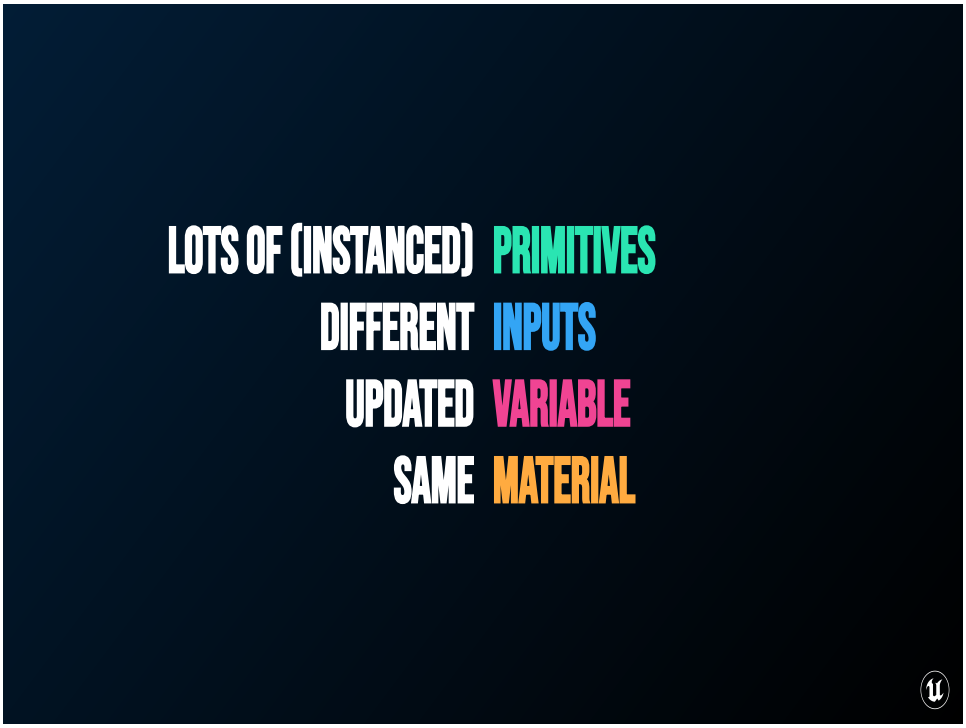
Take a breathe!



19:00

There’s another way you can pass information to a material through geometry that’s specific to instanced geometry. It’s effectively the same concept, it’s just set Per Instance instead of per-primitive like in our last example.

You may notice that this lead image looks the same as the one for Custom Primitive Data, but the keen-eyed among you will notice that these are instanced static meshes and not static mesh actors!

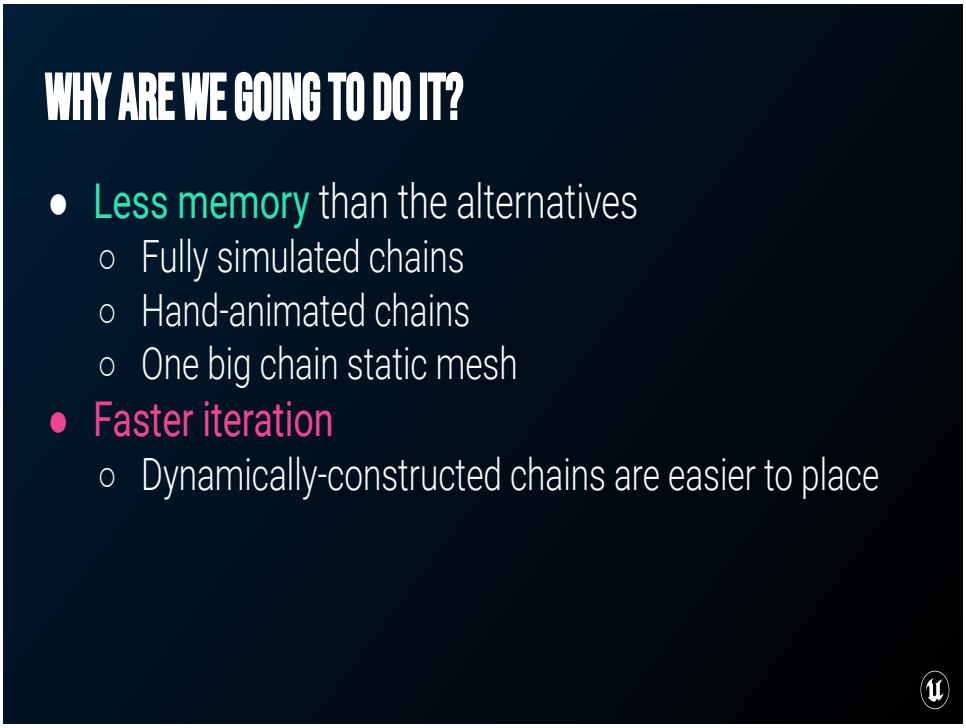


Then by extension, Per-instance custom data is great for situations where you have a LOT of primitives that need to react to different inputs that you define programmatically, but aren’t updated all that often. Of course, because we’re dealing with instanced primitives they’re all going to be using the same material.

To demonstrate this, I'm going to make a chain sway back and forth in the breeze without animating anything. In fact, for this example, I don't even need to update anything on tick. Just like the dials example, we're going to do this all in the vertex shader. But unlike the dials example, we're not going to create a dynamic material instance for each link the in chain. Instead we're going to leverage hierarchical instanced static meshes (IE instanced geometry that can LOD) and the Per-Instance Custom Data to pass the necessary data for the swaying action to each link in the chain.

For this example, take for granted that I have a blueprint which programmatically places instances of a chain link along a spline, and that there's controls for all that already.

In the Unreal Engine we have Hierarchical Instanced Static Mesh Components which handle a lot of the work of managing a bunch of instances of the same static mesh, and that's what I use here.

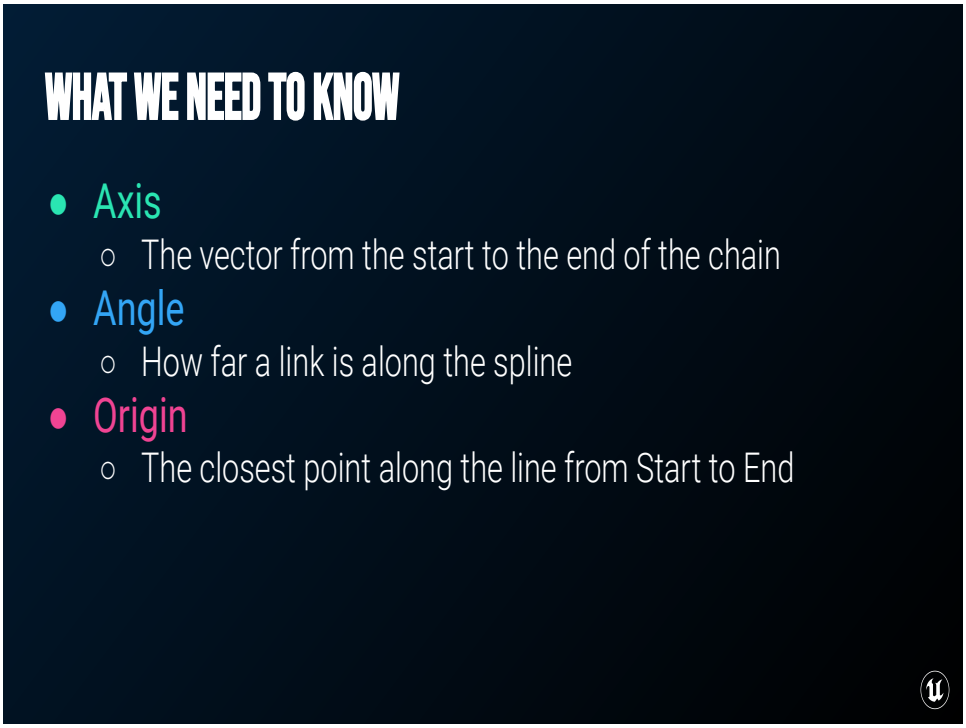


The benefit for this technique, especially with the Unreal Engine, is that it's going to be far more efficient than a lot of the alternatives.

Because we're doing some simple swaying on the GPU it's going to be faster to calculate than using fully simulated chains, and because it's going to be a static mesh it's going to be faster than using a skinned mesh that's been hand- or sim-animated. Because it uses Hierarchical Instanced Static Meshes it's also going to be more efficient on the GPU than if we had a single large chain static mesh because the engine will cull out the links we don't see. It'll also be cheaper because we're just stamping one chainlink along a spline than needed unique geometry for each part of the chain.

And, because this is dynamically constructed based on a few user parameters, they're easier to place and change around. You don't need 'ChainA, B, C, D, E' for each situation in which you might want a chain. Now artists are free to chain

up the entire map like Bob and Jacob Marley.



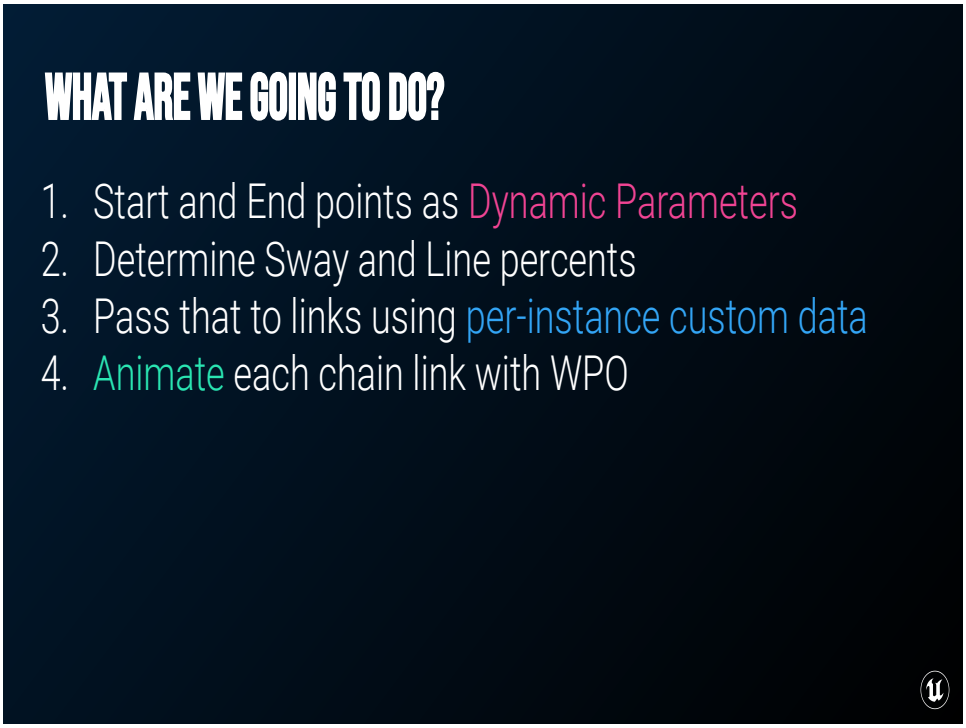
There's a lot we need to know to pull this off:

- The axis of rotation is just going to be the vector from the start to the end of the chain.
- To get the angle, I need to know how much the chain should sway based on how far it is to the middle of the chain
- And to get the origin for the rotation, I need to find the closest point along the Start->End vector to each link in the chain.

Importantly, how far a link is from the end of the chain is a DIFFERENT number from the closest point along the vector.

The good news is that it's fairly trivial to get all this information in blueprint.

And because of Unreal's Per-Instance Custom Data, we can set these values on each of the instances of the chain link through script.



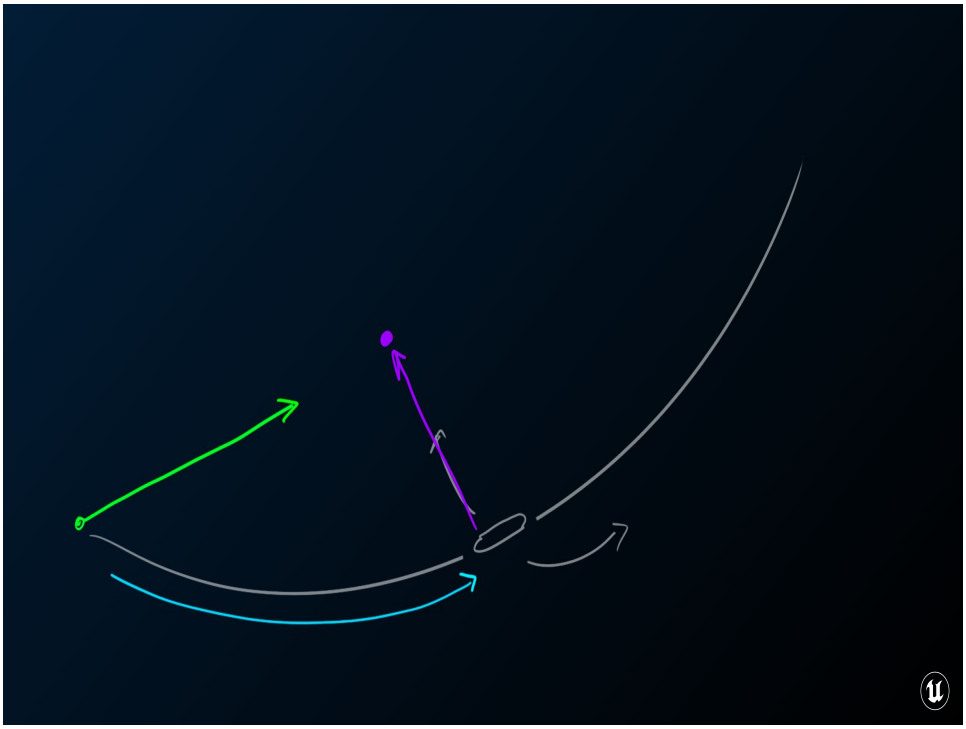
So, just like the dials, we're going to instantiate a dynamic material instance for the chain, and pass in the Start and End points as vector parameters.

Then, for each link in the chain, we're going to figure out two important things:

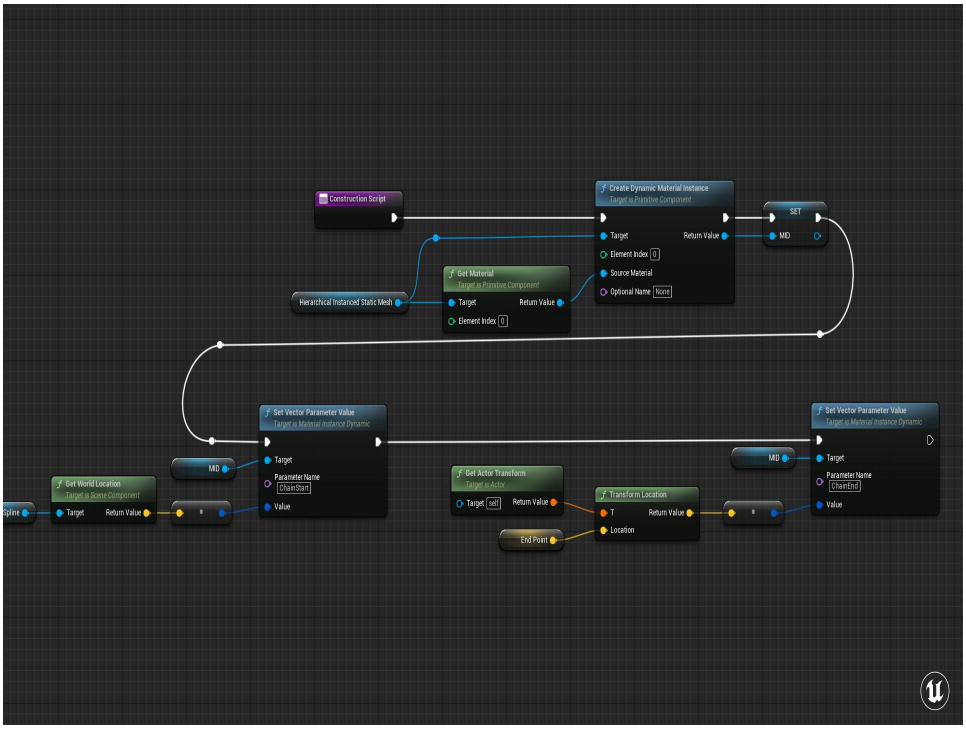
1. Based on where the link relative to the other links, how much show the link sway. That's our sway percent.
2. Then, so we can construct the axis of rotation, we're just going to use one scalar value that's the percent distance between the start and end points where we can find the closest point on the line. That's the LinePercent

We'll pass those two float values to each instance using Per Instance Custom Data

Then, just like the dials, we'll animate those links using World Position Offset

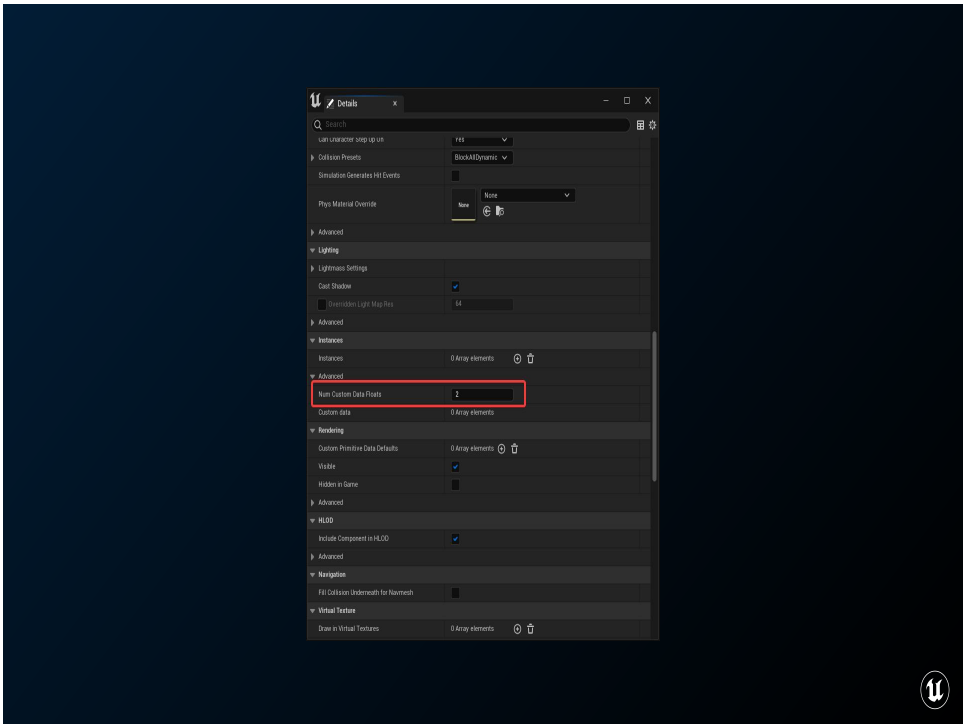


And for those of you who’re visually inclined



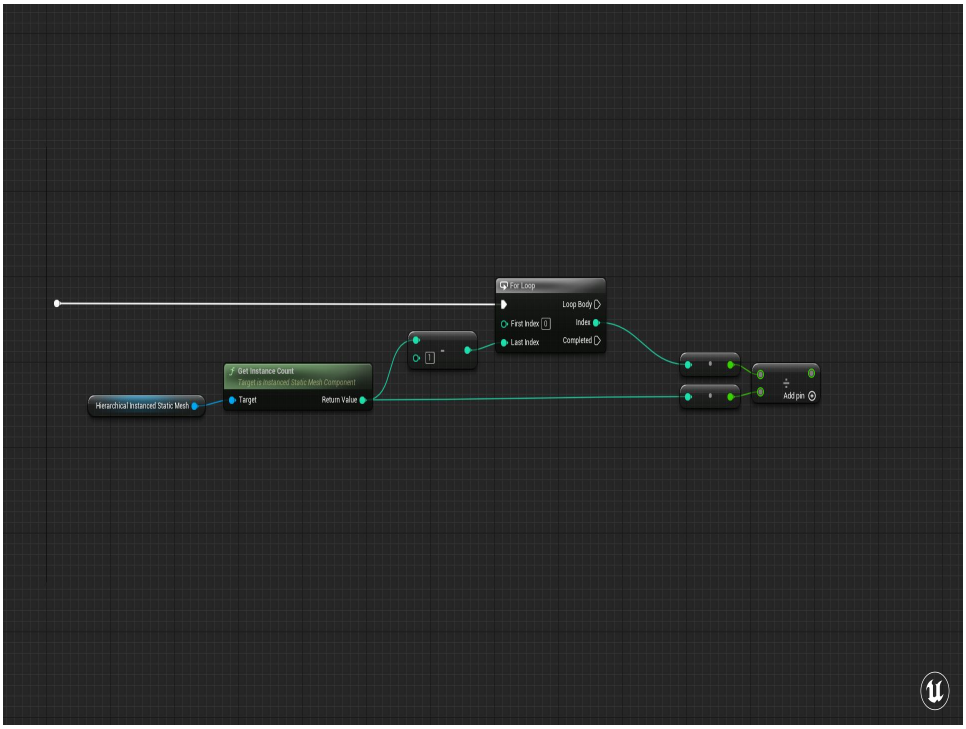
But not everything needs to be per-instance, because some of these values are global to the whole chain, like the start and end points of the chain. For that, like before, create a dynamic material instance and set the StartPosition and EndPosition of the chain as vectors on the material.

I’m doing this all in worldspace to simplify the calculations on the GPU.



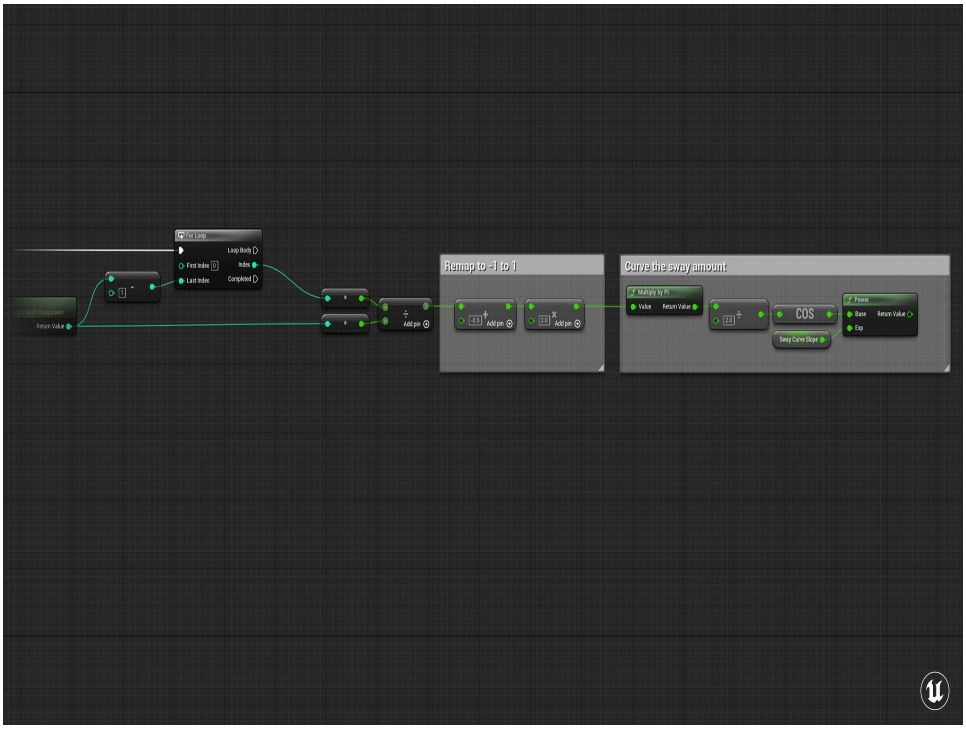
On the InstancedStaticMeshComponent I need to tell the engine that I’m going to pass in some per-instance data with the . This is going to be the “SwayAmount” and the “LinePercent”. NumCustomDataFloats. You can ALSO set CustomPrimitiveData on InstancedStaticMeshes, but they’ll be applied to all the instanced in the component.

LinePercent is how far along the vector from Start to End is the chain placed, and the SwayAmount is a value we’re going to calculate based on how far along the chain a given link is



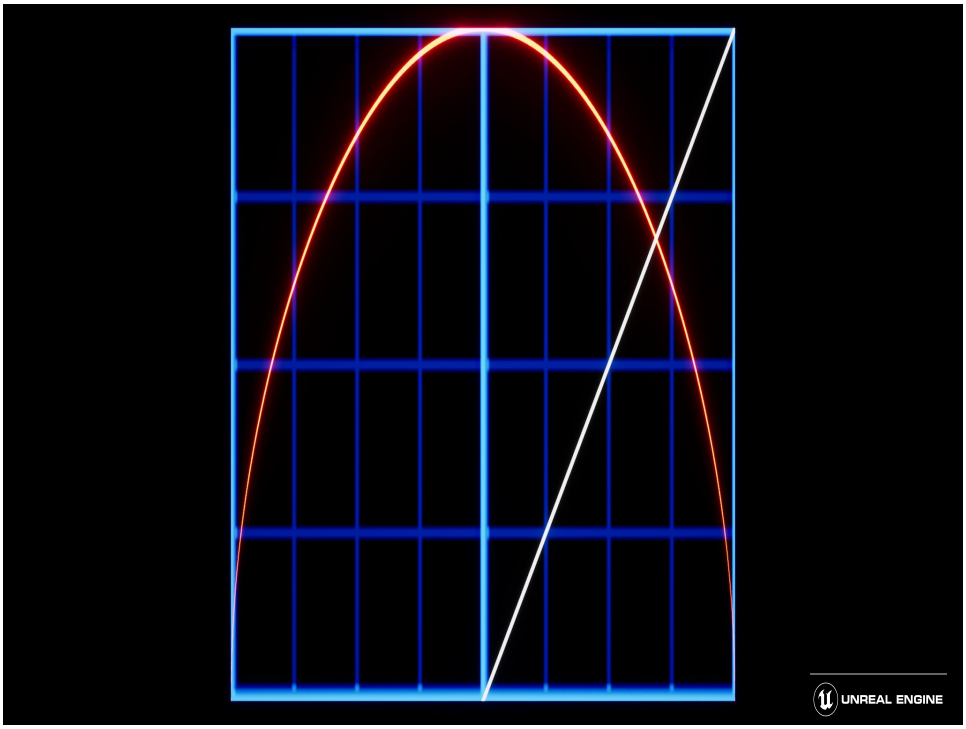
We’re going to use the ChainPercent to figure out the overall SwayAmount. Since we’re only figuring this out once, I’m okay handling this calculation on the CPU instead of sorting it out on the GPU.

In the construction script we’re going to loop over all the instances to set these values. So the ChainPercent is the current chain number divided by the total number of links.

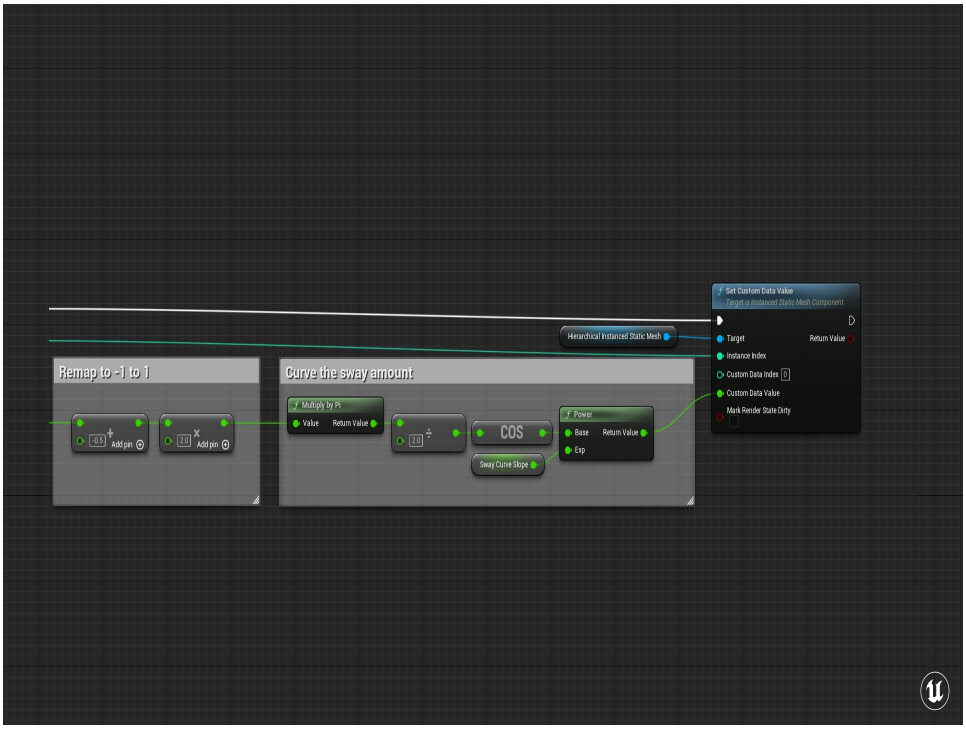


Now to figure out how much the link should sway, I’m going to remap the 0 to 1 ChainPercent between -1 and 1

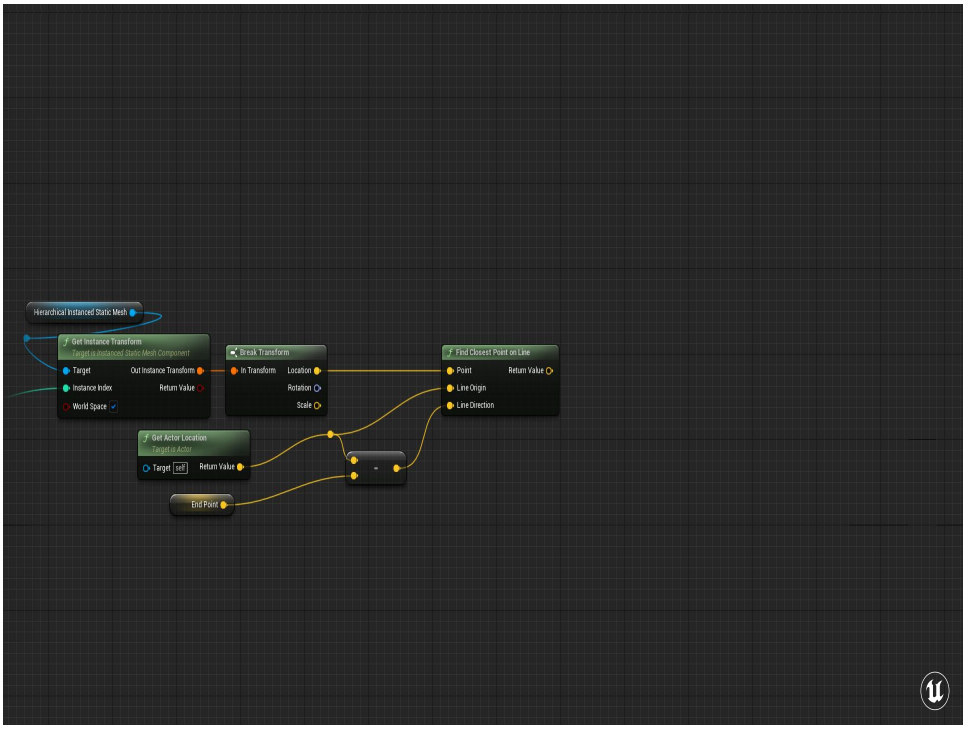
Then feed that into this little function here which is the cosine of pi * RemappedChainPercent, divided by two, raised to a user-controlled power.



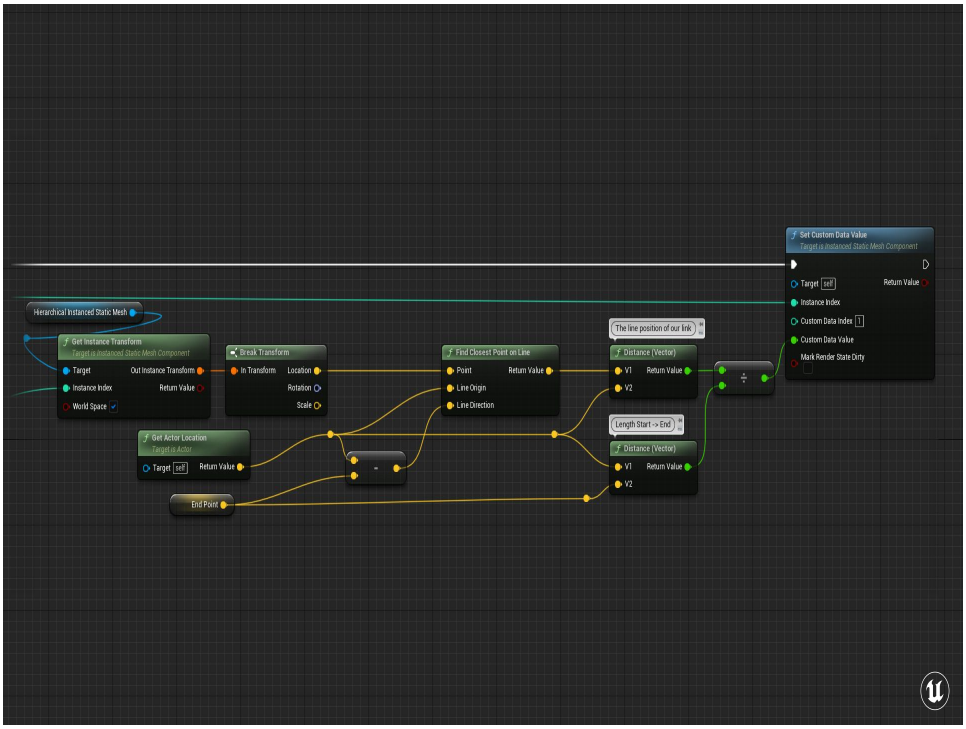
And just to illustrate that, here’s what that curve ends up looking like. I end up with a much nicer curve and if I’d just done a simple lerp or used the raw ChainPercent result. I’ll include a link at the end of the slide for some other handy equations for getting graphs of different shapes.



Then I can set that 0 to 1 value as one of my Per-Instance Custom Data. I know going into this that my “SwayAmount” is going to be the first Custom Data Float, so that’s the Custom Data Index of 0, and the instance index here is the chain that I’m operating on right now.



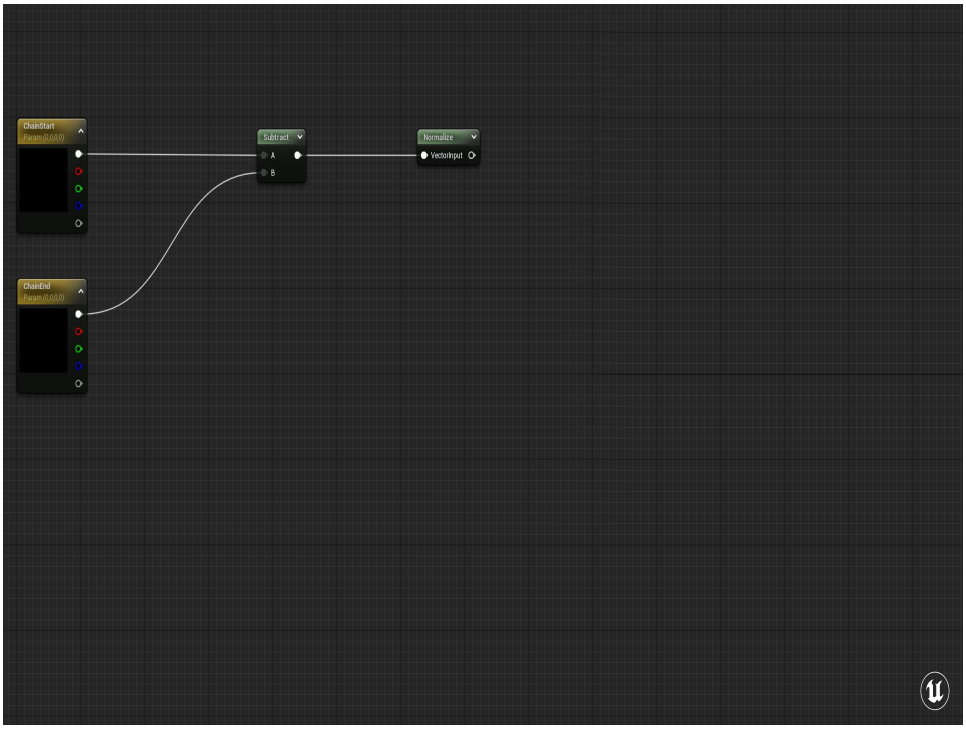
The line percent is a bit trickier, but luckily Unreal has a built in “get closest point along a line” node. For the point, I just ask the instanced component for the worldspace location of the instance we’re operating upon. The origin of the line is the origin of the actor, and the direction is the Start -> End Vector.



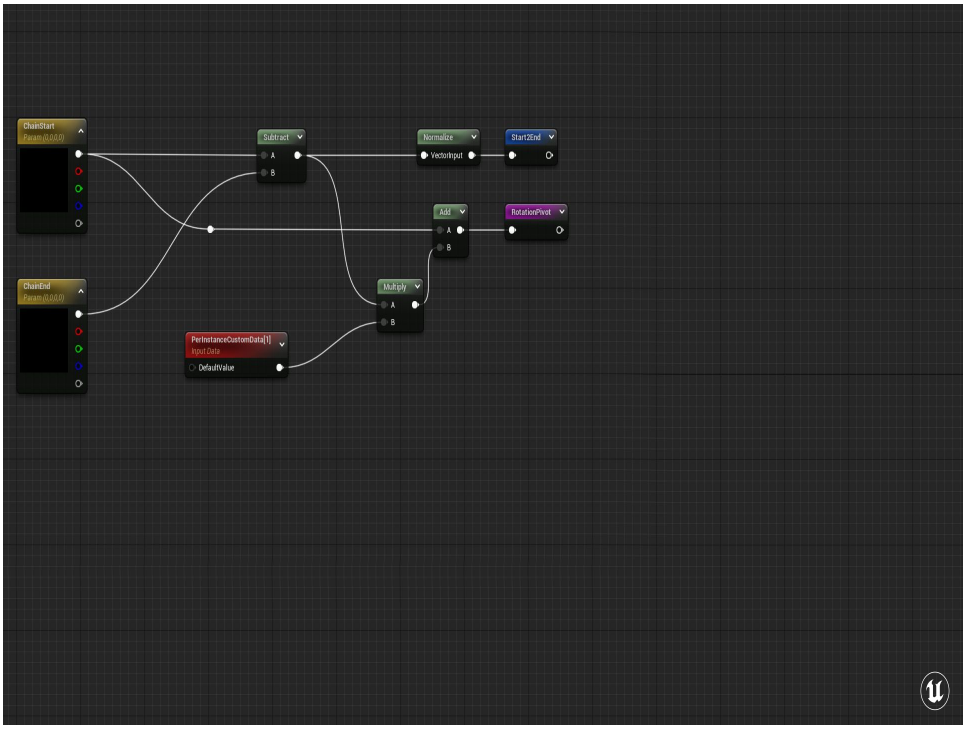
and I can use that to figure out how far along that vector this closest point by getting the distance between that point and the origin of the chain, then dividing that by the total length from the Start to the End of the chian.

And then set that as the second Custom Data value on the component.

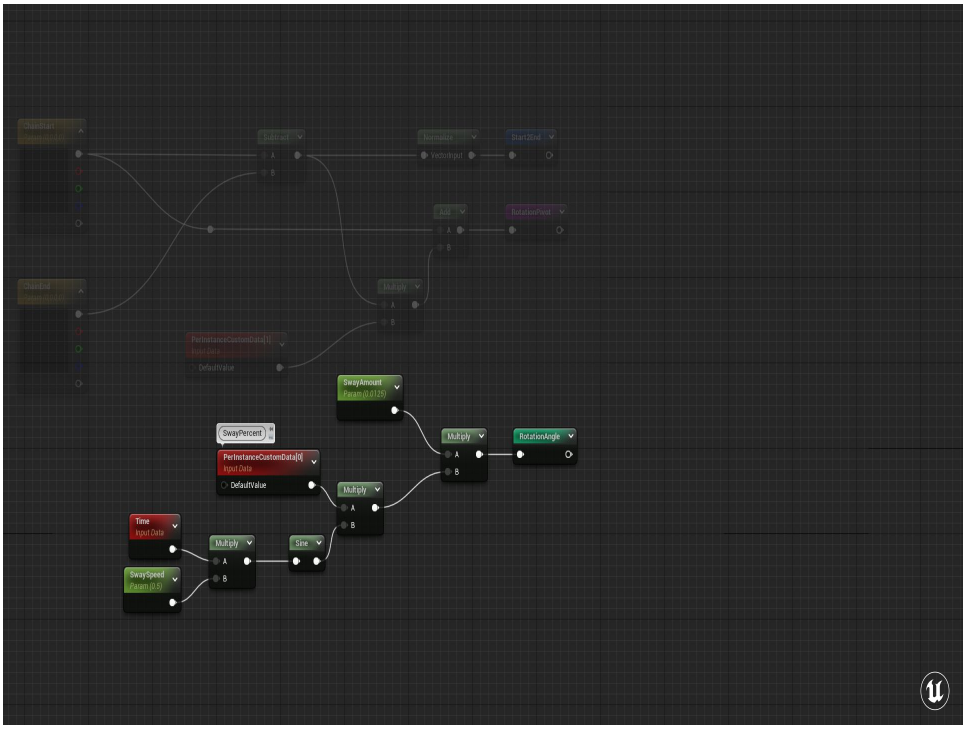
I'll use this in the material to recreate the full vector, since it'll be cheaper to do that then to take up three per-instance custom data scalars to do so.



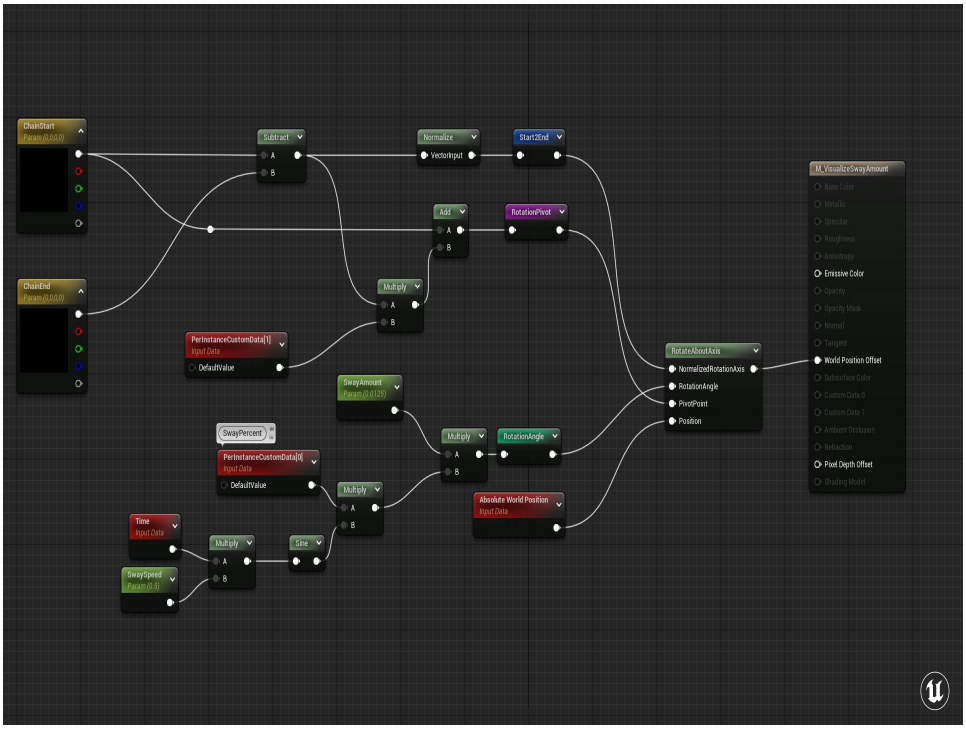
In the material, I'll first create the vector Start -> End and normalize it for the rotation axis. Because we're once again using CustomPrimitiveData to pass in the SwaySpeed and SwayAmounts, I'll



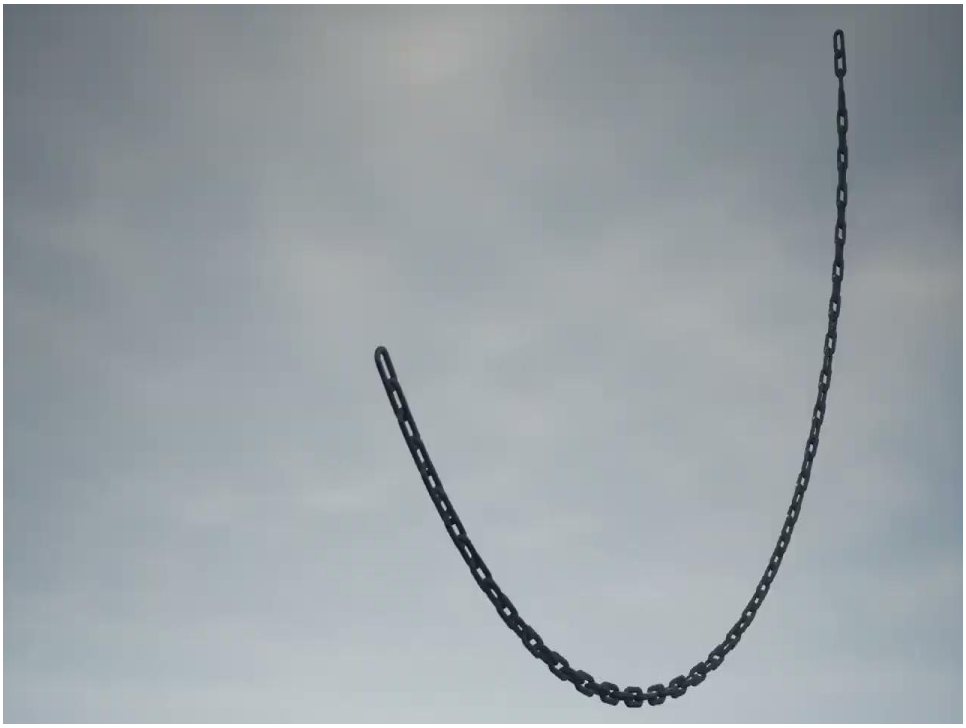
Then I'll multiply the un-normalized version of that vector by the LinePercent we calculated in the construction script, and add it to the start position to create my origin point.



And for the rotation angle, I'll multiply the SwayPercent by the Sine of Time, then multiply that by a total SwayAmount that limits the extent of the rotation

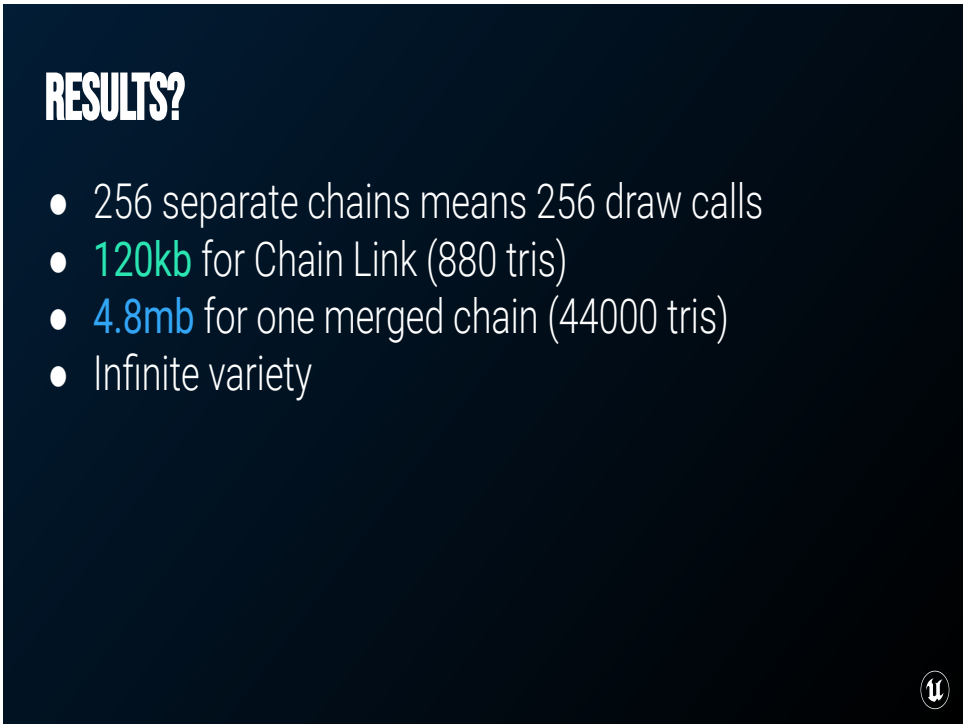


And finally plug all those into the RotateAboutAxis node, and pipe that out to WorldPositionOffset



And POOF, chain swaying in the breeze!

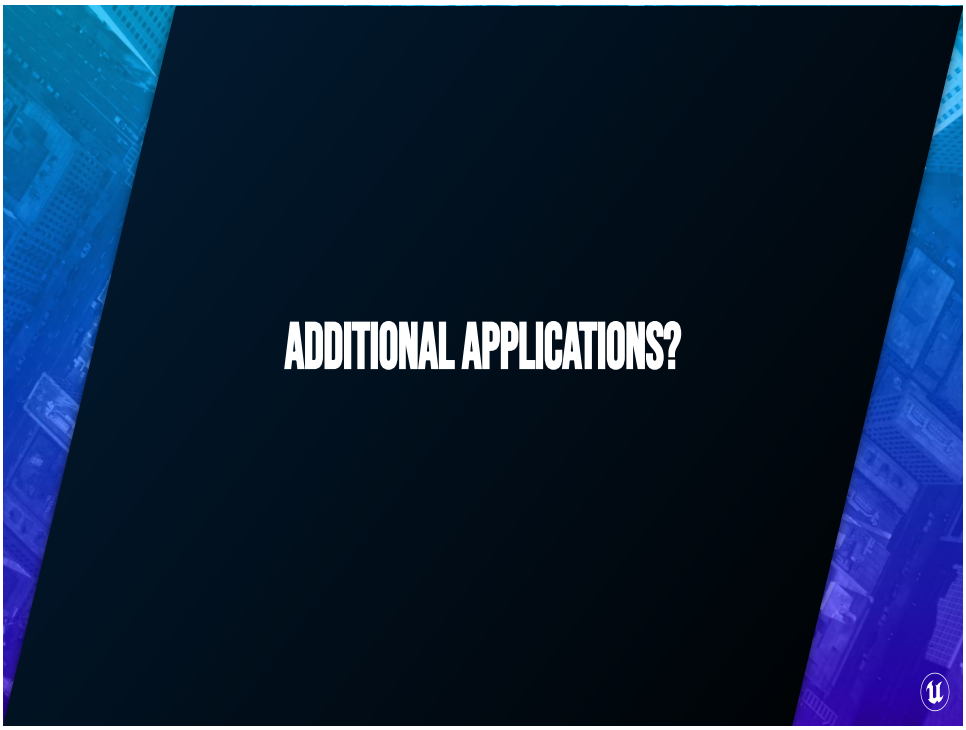
The great thing about this chain is that, 1) because I set it up as a blueprint, I can make a bunch of these and morph the chains around all I want. and 2) Each chain is only one draw call because each chain is set up as an InstancedStaticMeshComponent.



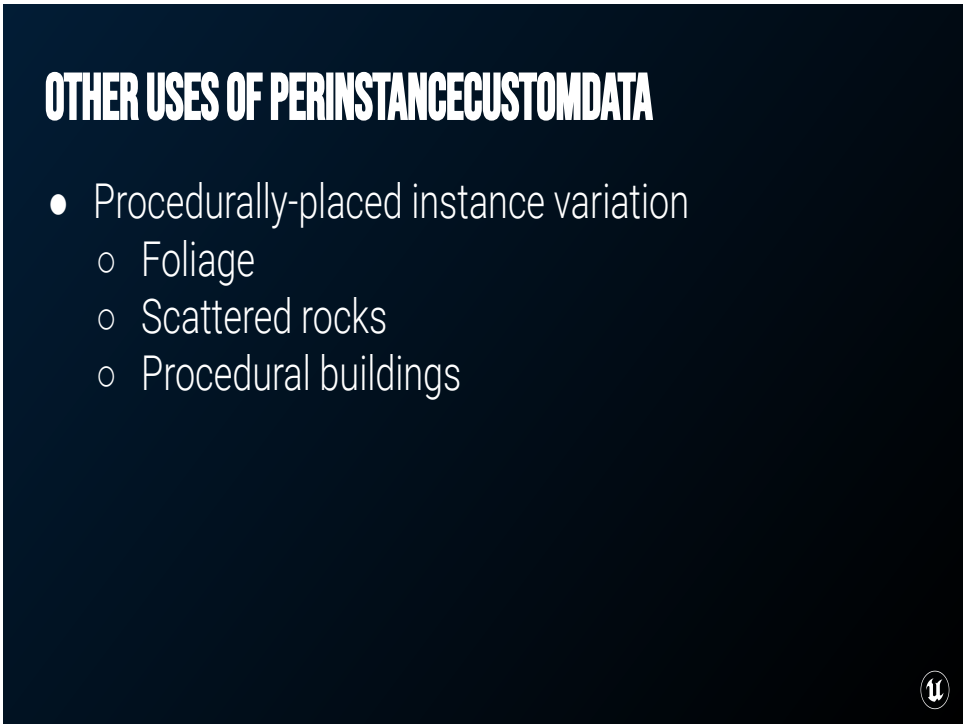
One limitation of this specific technique is that each HierarchicalInstancedStaticMeshComponent will be its own separate draw call, those aren't dynamically batched together.

The benefits here are more about the memory footprint (1 chainlink vs. X chainlinks times all the different chain variations you'd need). For example, I used I think a 20-sided stretched torus for my chain link which you probably wouldn't want to do if you were to have a set of unique chains, but for comparison's sake that single link was only about 120kb in memory, whereas one merged chain of those links was 4.8mb.

The other great benefit is that now artists are more free to chain up whatever they want, wherever it is, without spinning up new assets every time, or being chained down by a limited selection of pre-built chains.



But it’s not just about making team colors on bases, CustomPrimtiveData has a variety of uses and I wanted to show you a couple that have come up over the years.



PerInstanceCustom Data really shines in situations where you're programmatically constructing a lot of objects all at once. You can use this for foliage or scattered rock variation, or use it when you're procedurally constructing buildings to drive information about the different windows and such.

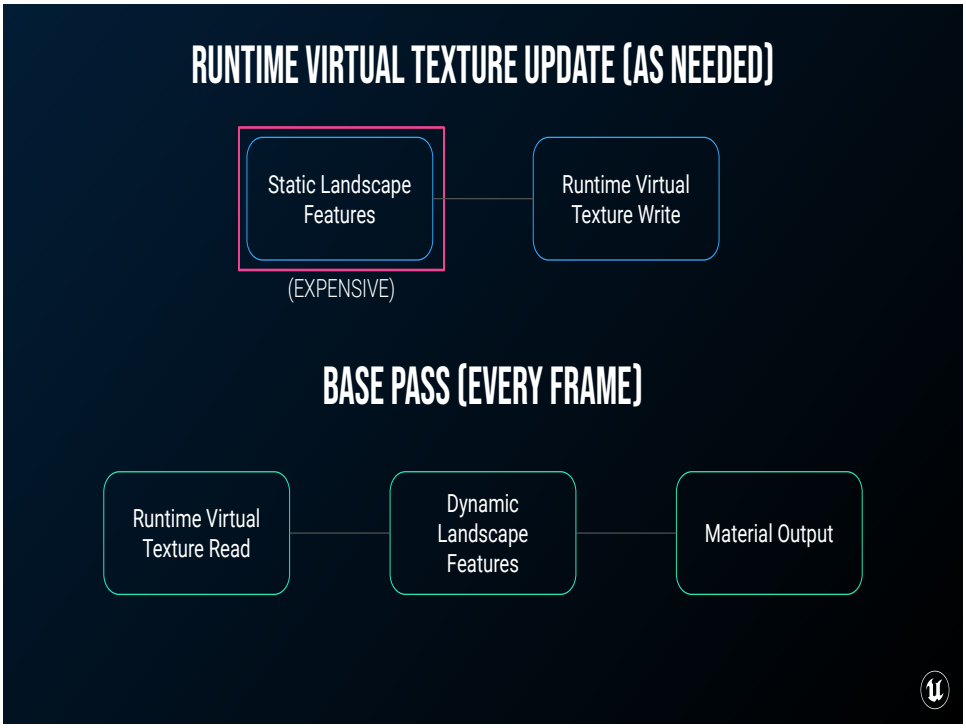
I’m also playing around with the idea of using PICD to drive a perf visualization in the game

Take a breathe!

29:00

Fourth technique, we’re almost at the end!

This one uses Runtime Virtual Textures, which create their texel data on demand using the GPU at runtime. To oversimplify it they’re large, highly efficient render targets that get rendered into from top down.



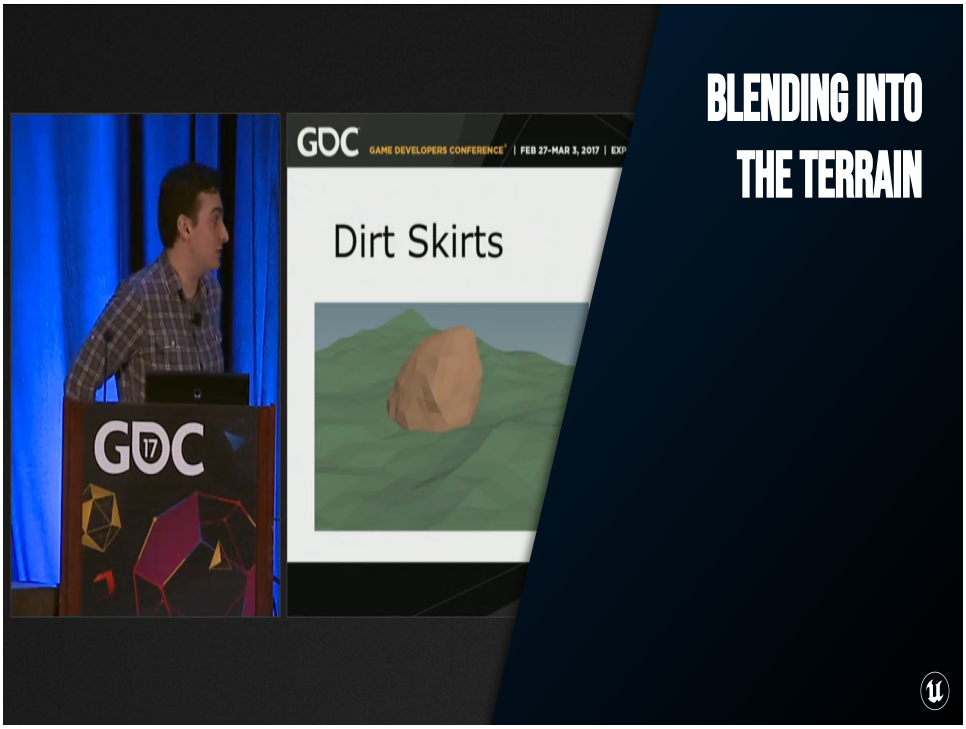
This them great for things like Landscapes. Many landscape materials can get quite expensive as layers stack up, but after an artist is done painting the landscape, these are largely static. You can offload most of the calculations for these materials to the infrequently-updated runtime virtual texture, and sample back from that to draw onto the landscape itself.

This way, the more expensive and non-dynamic parts of your landscape material are no longer calculated every frame.

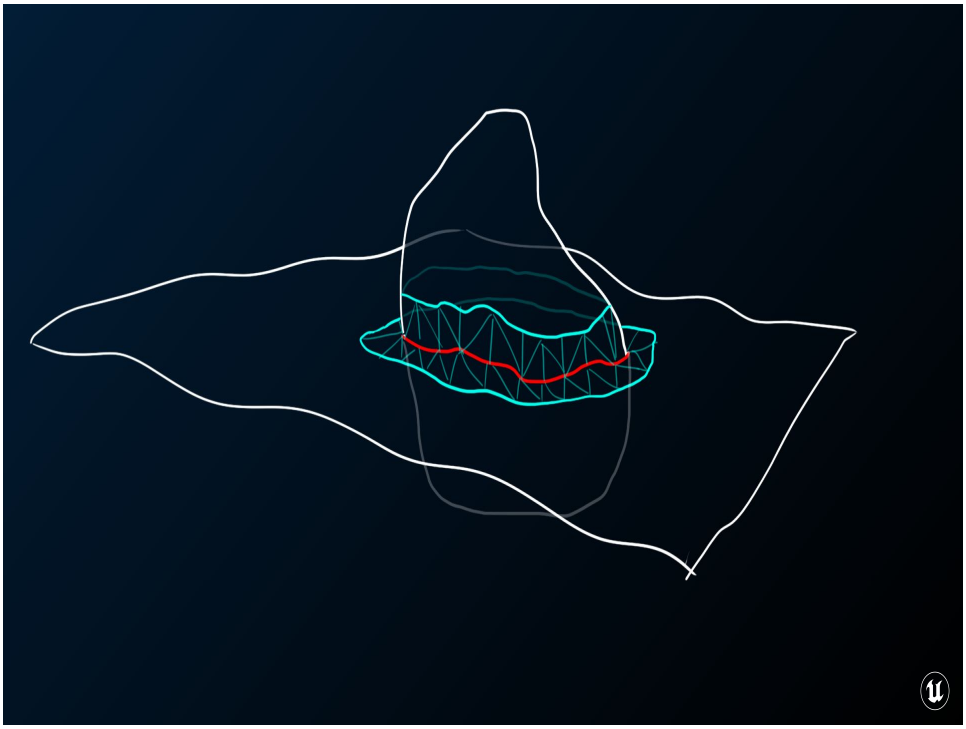
When it comes to finally drawing a pixel on screen, all you're doing is sampling one RVT at the pixel's world position.

There's two sides to how we're going to use RVTs, but for the purposes of consistency on the *read* side of things, these are great for when you have many primitives that need to react to the same set of input data that varies per pixel, but that set of input data doesn't have to update every frame.

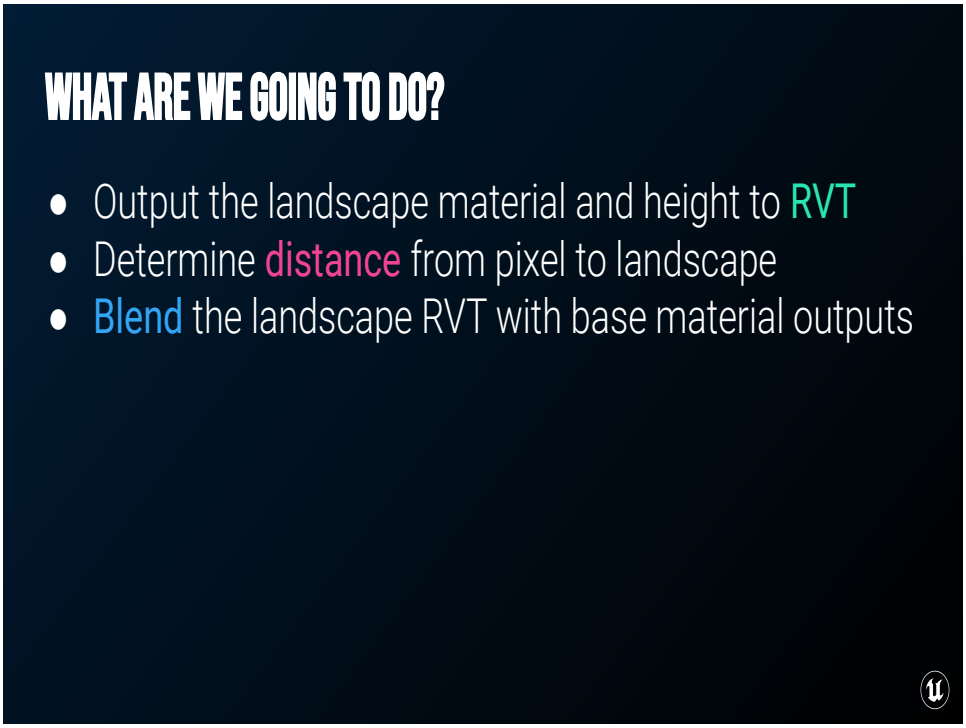
To demonstrate this technique, I want to show you how you can blend geometry into the terrain without any additional geometry, or having to recalculate material values on the fly.



One of the many challenges we've faced over the years is making sure the static geometry we use to accentuate our landscape looks grounded and connected to the landscape. I remember chatting with a different former colleague about this as far back as I think 2013 or 14. My former colleague, Luiz Kruel, showed how to do this with geometry called Dirt Skirts at the Technical Artist Bootcamp in 2017. We used this trick to great effect in production on at least one title.



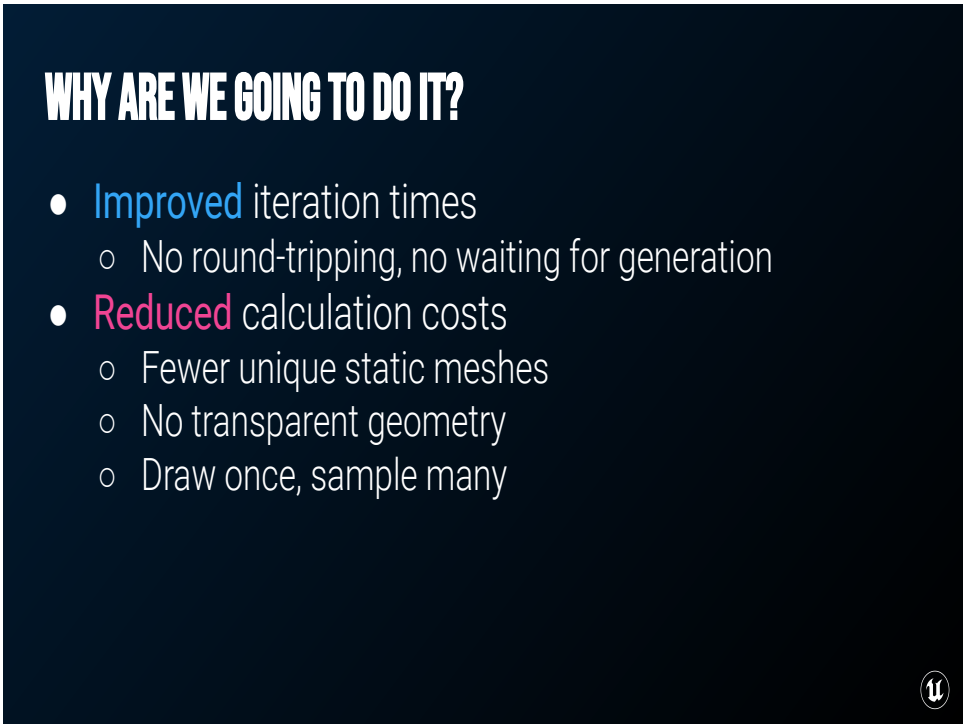
For that technique, you export part of your landscape and your whole rock to Houdini (or now in the Unreal Engine you can do this with the Houdini Engine directly in your application), determine the intersection between the two, create a little skirt of geometry with some alpha around the edges, and apply the landscape material (or a similarly-shaded purpose-built material) to the skirt.



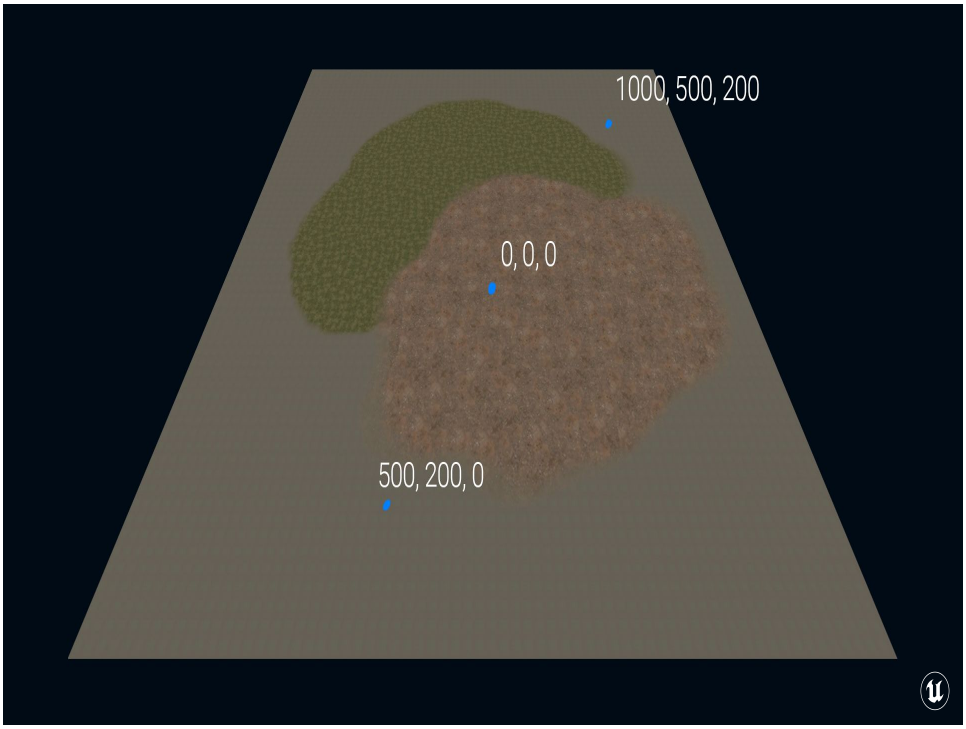
So what we need to do is output the landscape material, and its height into a texture set

Then we need to figure out how far a pixel is to the terrain

And use that to blend between the landscape texture set and the base material

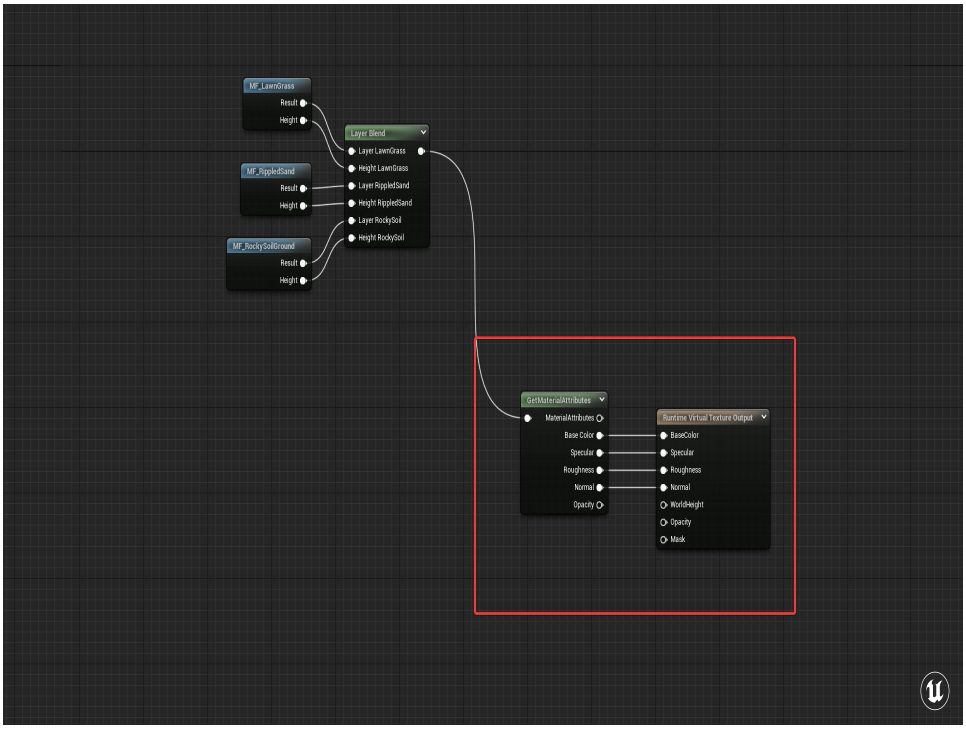


So now instead of doing something like… recalculating the entire landscape material for each pixel of the landscape *and* each pixel of geo that's meant to blend with it, I can draw the landscape once, and sample it in a bunch of different places as a simple texture read. Much, MUCH faster.



The reason this works is that RVTs aren’t limited to just the material writing into them. They can be sampled from other materials, you can just use an XY world position and the RVT sampler will return the values from the texture set at that location.

So the great thing about this technique is that as I move meshes around the level, they're automatically grabbing the nearby parts of the landscape. I don't have to worry about regenerating additional geometry each time I move it.

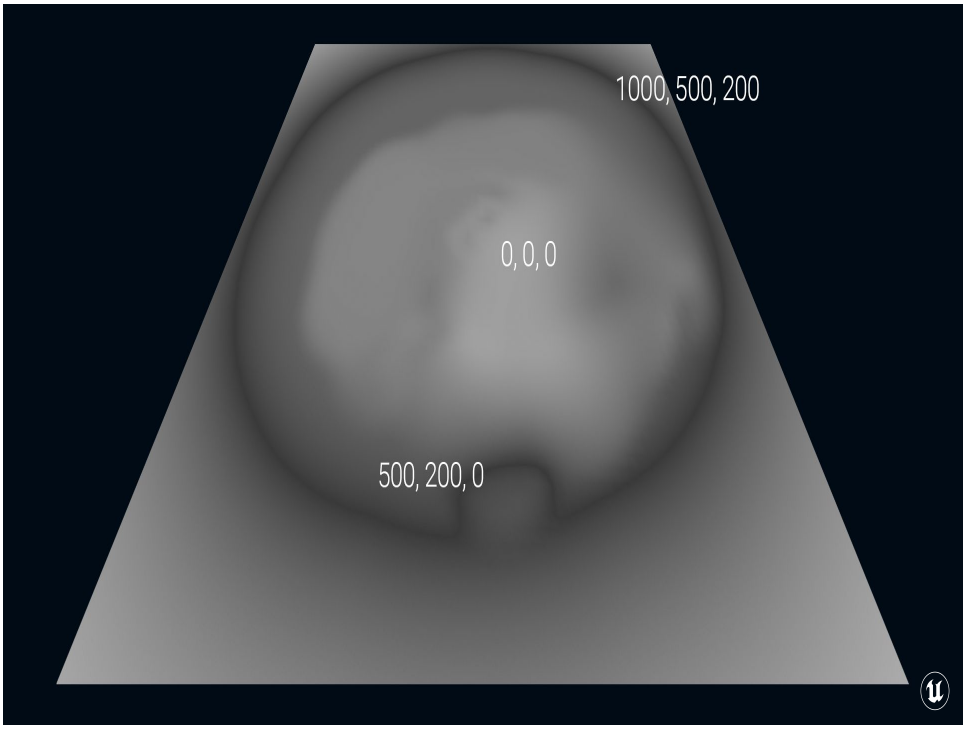


The landscape material is set up to pipe out its base color, specular, roughness, and normal to the RVT Output in the material. I've got all the particulars for that output handled behind the curtain, so to speak. This is the key part here, this just means that whenever this material is drawn in the RVT context, output these values to this texture.

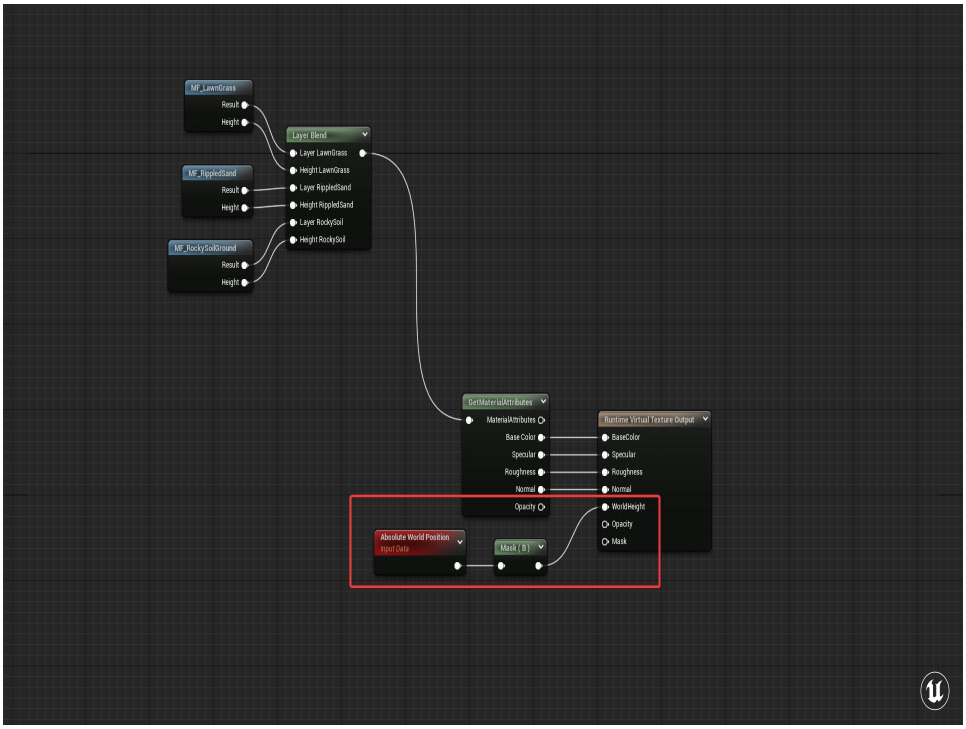
And that's all we need to do for that. Now I've got the landscape material and instead of having to do something like run the whole landscape material in blending geometry.

But what good is sampling the RVT from a given world position if I don't know where the landscape is relative to my geometry? I have to know where to stop the blend, and I want to do that relative to the landscape itself.

As a quick aside, in Unreal there's the a concept of "Material Attributes", which is a giant struct of all possible material inputs that get passed around the graph, and it's really useful for organizing large material graphs or breaking up complex operations. So that's what that GetMaterialAttributes block was in the previous graph.

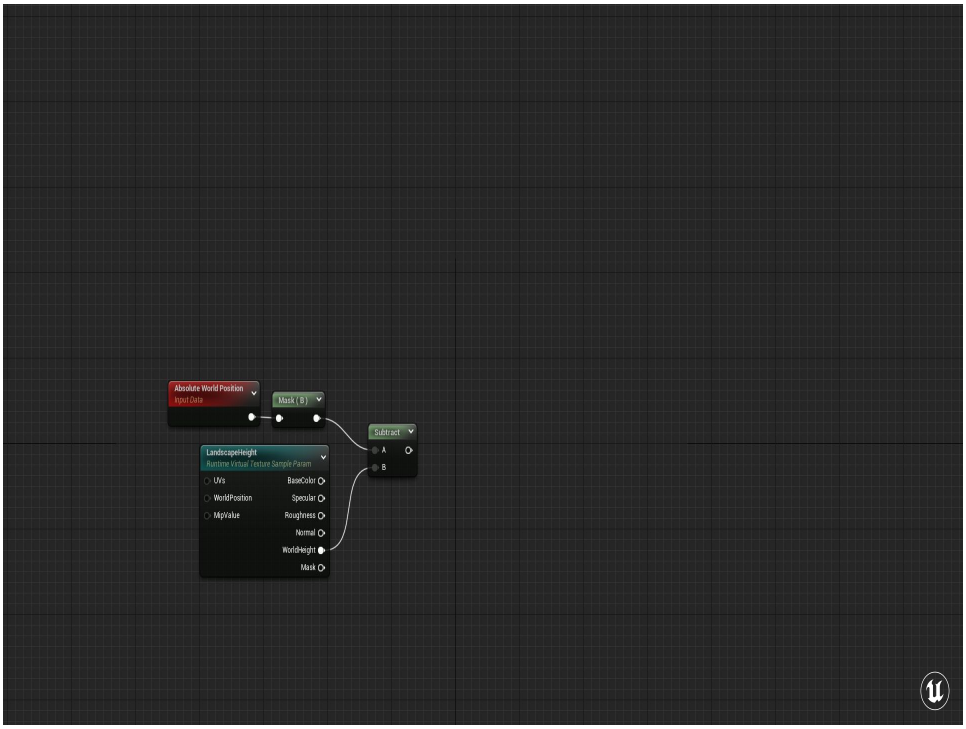


The great news is that one of the other channels to which you can write is *height*. This means there's a whole texture that's meant to associate a Z height value with a given X/Y Position. And if I write the height of the landscape out to that runtime virtual texture, then I can look up the height of the landscape at any point in the world in any material.



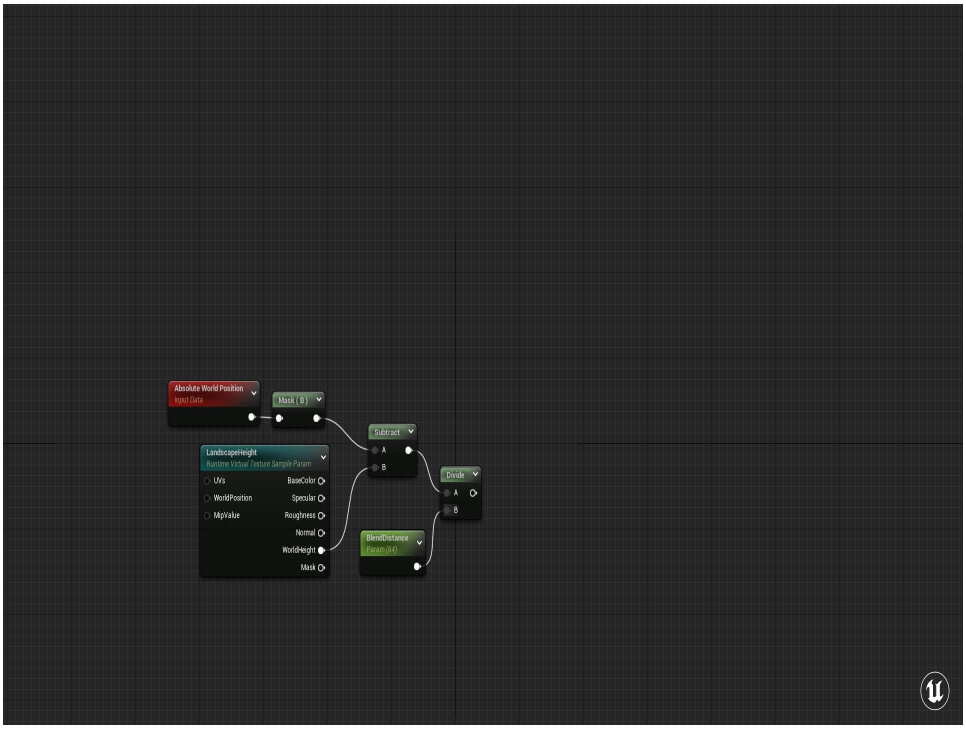
Jumping back to the Landscape material, all I'm doing here is taking the Absolute World Position of the landscape, masking it to the B component (or Z) and passing that into the World Height input of the Runtime Virtual Texture Output Node.

This means that any time this material is on a primitive drawn into a Runtime Virtual Texture set to Height, this value will be written into that RVT. Easy!

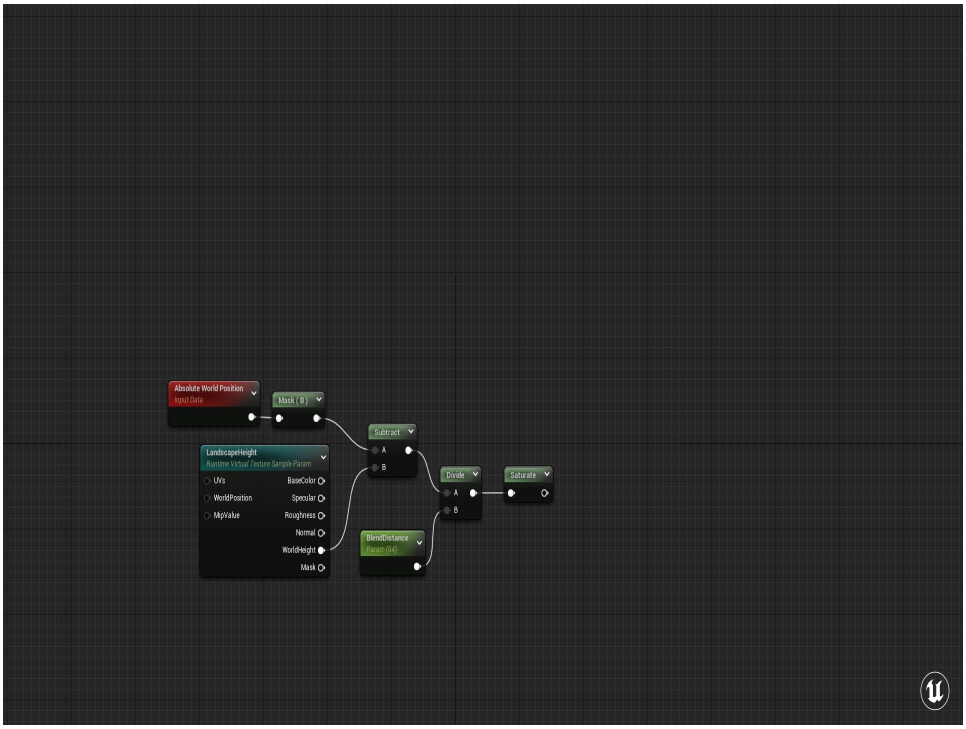


What we'll do is sample the Height RVT from the current pixel position (which is the default). And I want a single scalar value that we can tell us when the pixel being drawn is closer or farther from the landscape, and caps out beyond a certain distance.

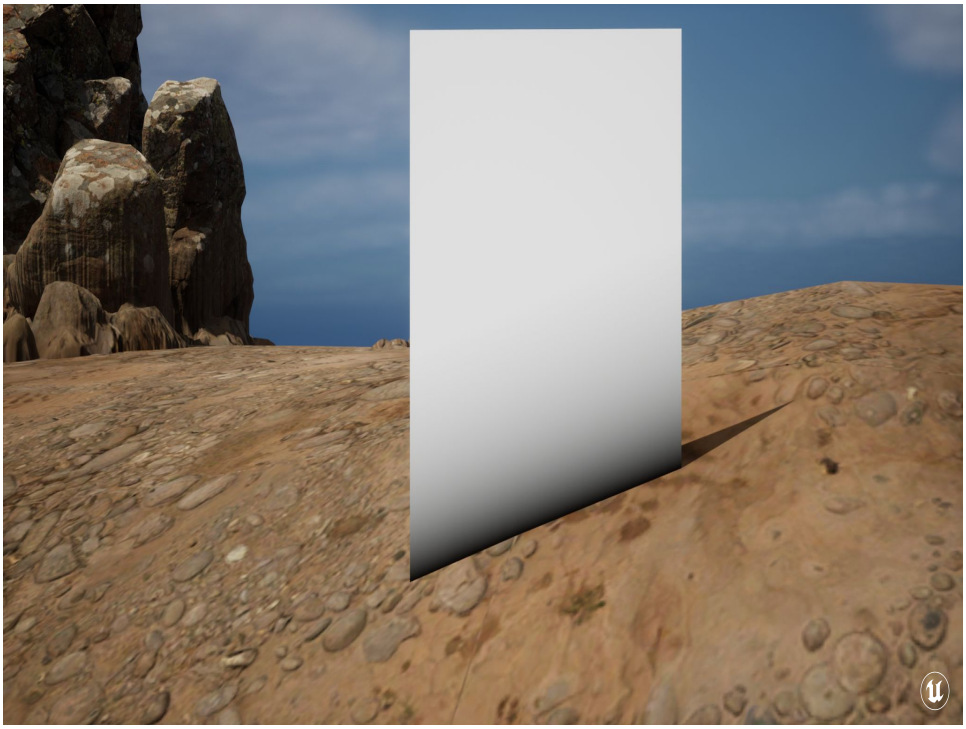
So we'll do a simple normalize here and subtract the World Height from the Z position of the current pixel.



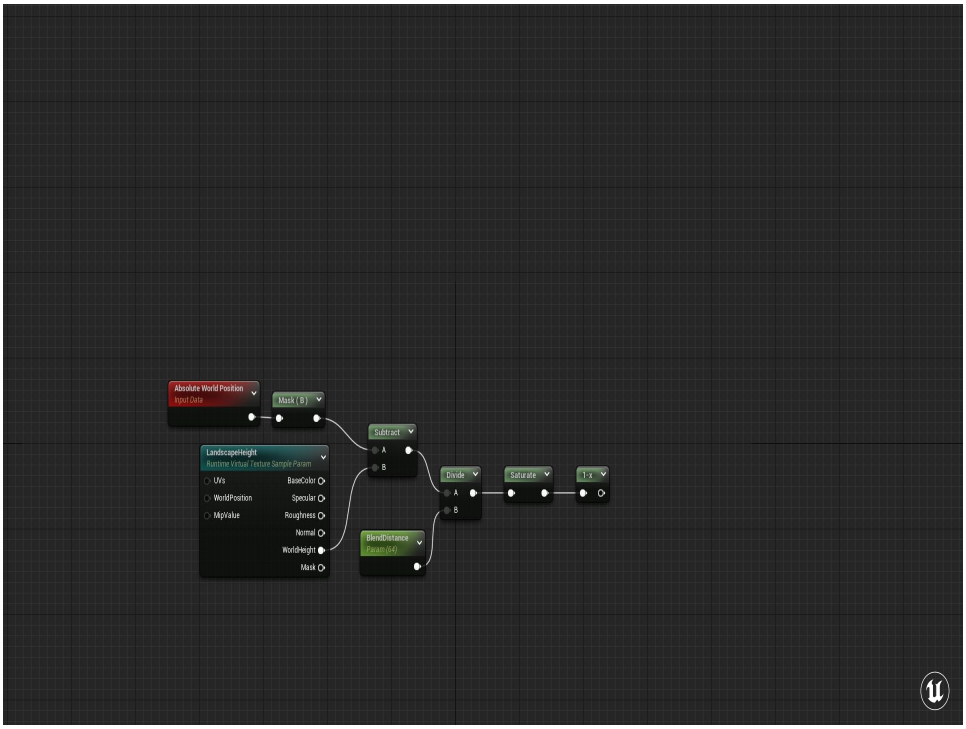
Then divide that value by a Scalar Parameter (so I can modify this on a per-material instance basis) called BlendDistance, and set that to 128.



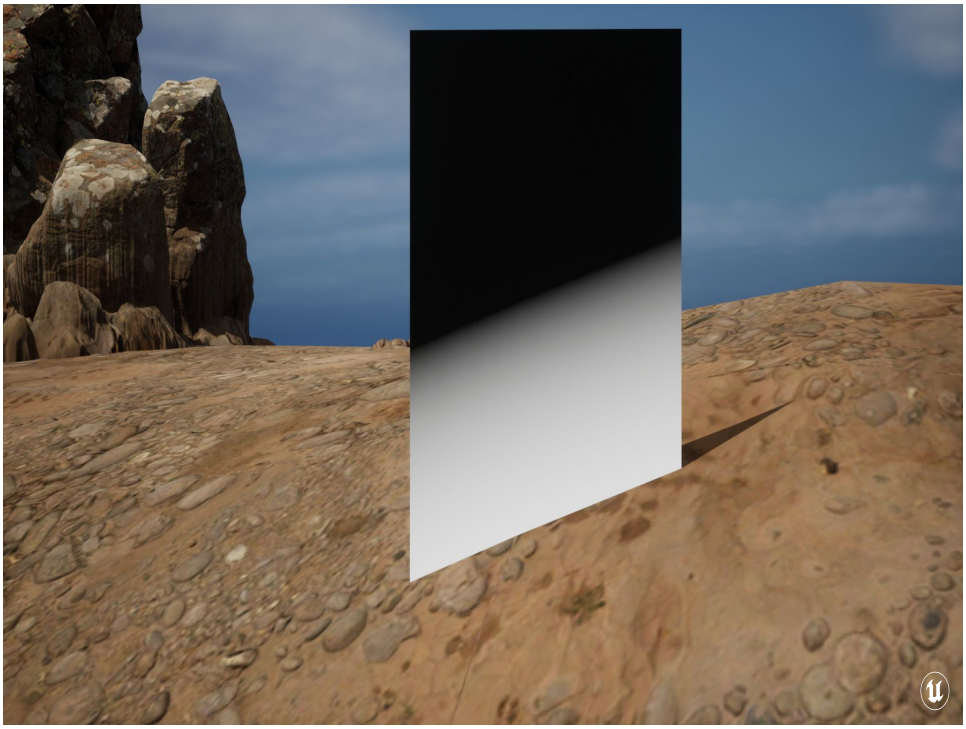
Then saturate the result, to keep it between 0 and 1. A saturate is just like a Clamp(0, 1) but is less computationally expensive or even free depending on the graphics hardware.



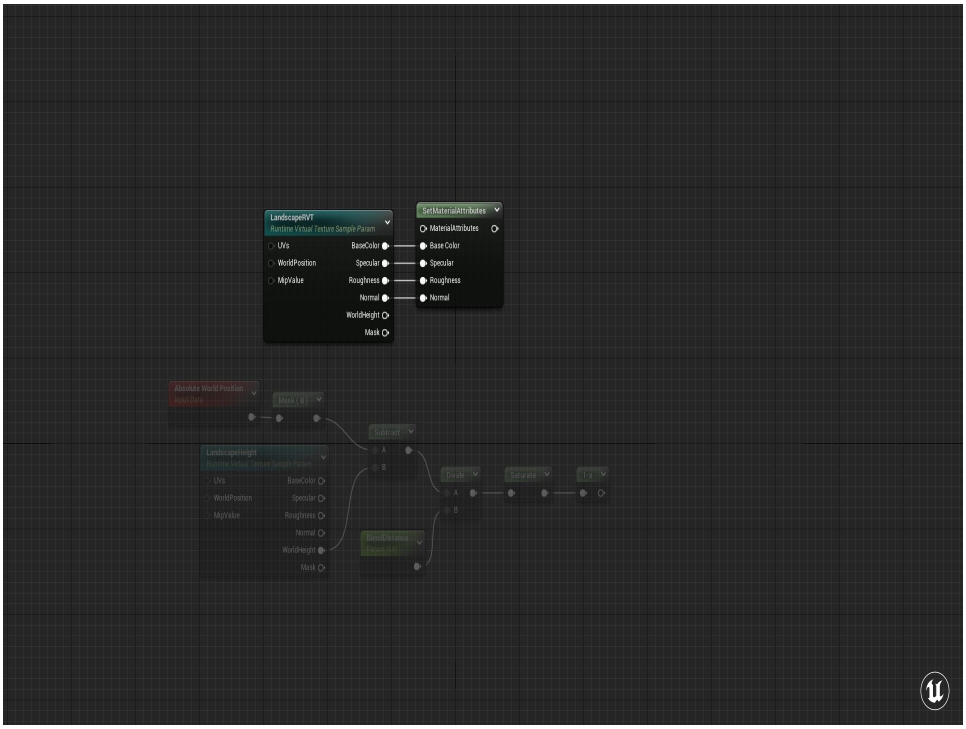
Here’s what that looks like if I just apply this value directly to a material.



And to make it all make sense to my brain, I pass that through a OneMinus node.



So that at the landscape the value is 1, and it fades off to zero the further away it gets.

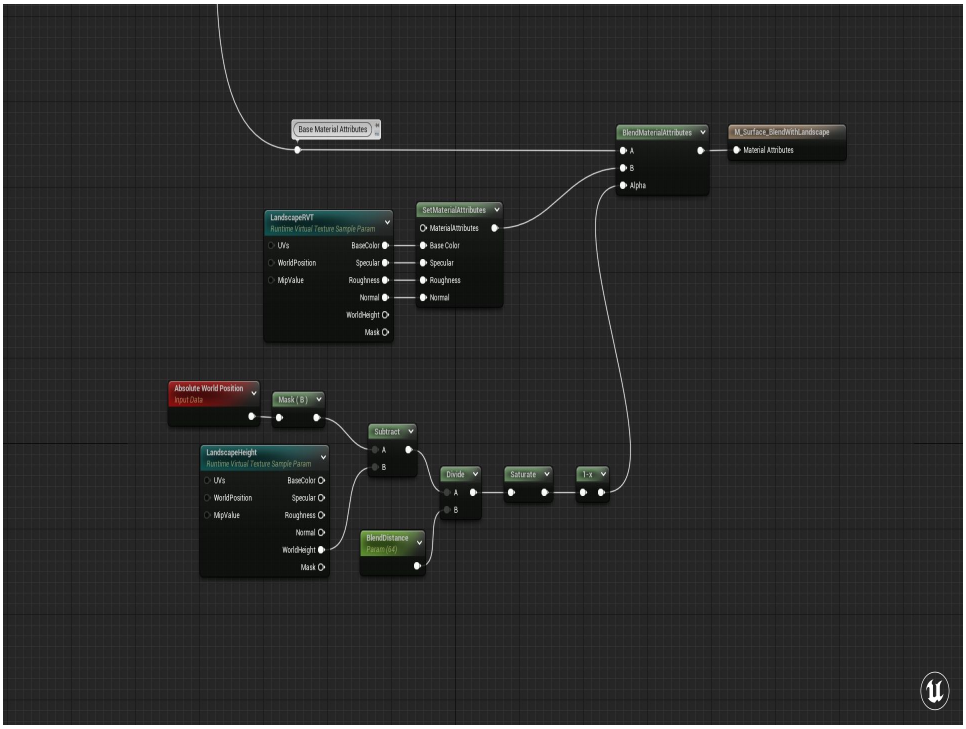


Now that we know how close the rock is to the landscape, we need to recreate the landscape in the rock material.

Same thing as before, we sample that RVT, and we're going to do something with the other pins here now.

But we're not going to Lerp each of these values in my base material, instead let me show you something cool you can do with MaterialAttributes

First off, if I pipe that material attributes block straight to the material output, it’s just showing what the landscape is showing.



So all we need to do is is use the BlendMaterialAttributes node to blend our landscape material with the base material of the rock.

And use the distance-blending value we created earlier to drive that blend.

Here’s what that looks like with the blend.

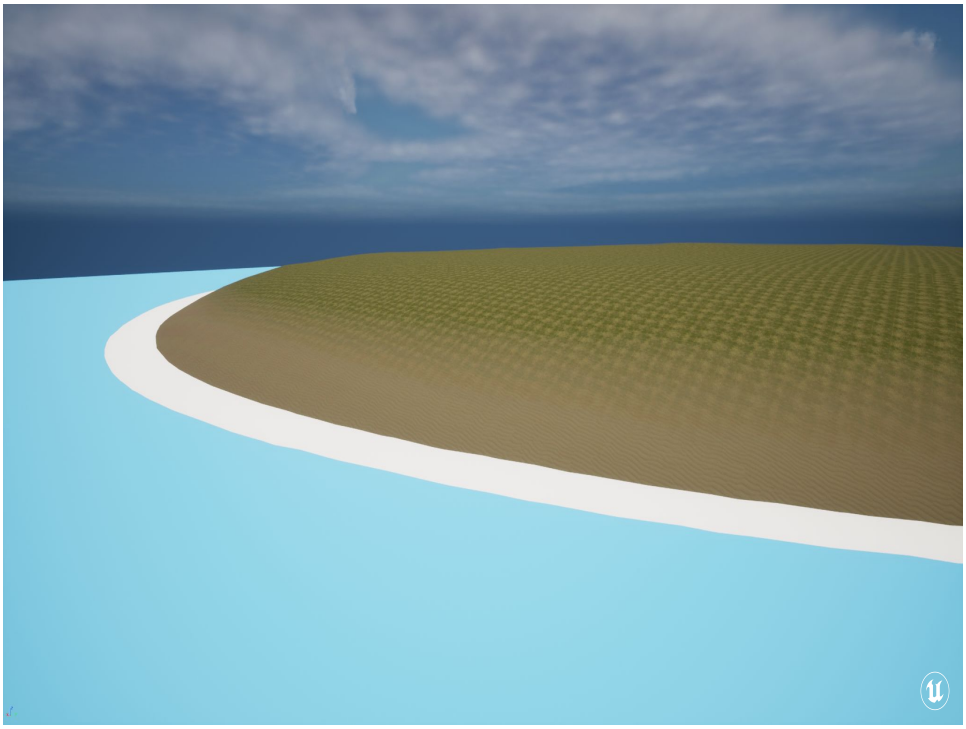
For comparison, here’s what that looks like without the blend

An additional benefit here is that this lets me blend more shallow meshes like these rock clusters.

To test the performance differences between this technique and some alternatives, I created a full-screen quad and swapped between two materials: the base RVT blending material and one that has a material function that blends all the material layers together. Swapping back and forth between those two showed a roughly .1ms difference in the base pass cost on the GPU.

This has come up a few times over the last couple of years, and there’s a few other tricks you can if you know the height or composition of a landscape.

You can use it to mask falling rain or snow, since you know the height of the uppermost Thing. With that you can also do a sort of faked caustics shadow casting underwater without having to worry about using light functions or anything expensive like that.

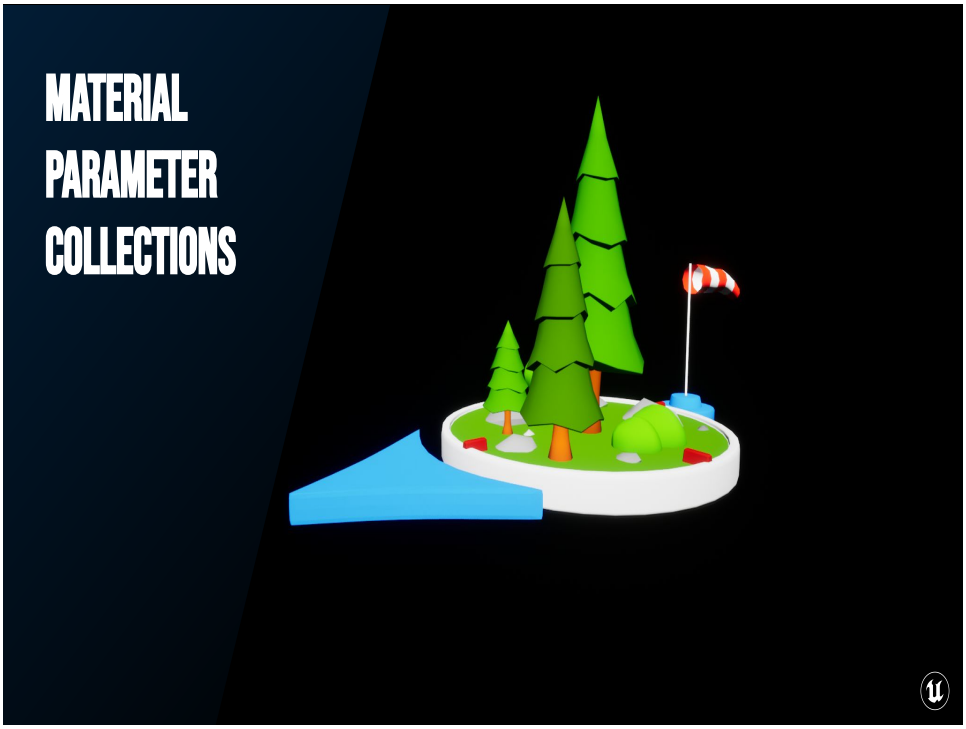


And you can use it for shoreline detection, instead of relying on something like a DepthFade

You can also the landscape’s color RVT to vary the colors of foliage, and ground the color of your foliage to the landscape itself.

You can also, with the height of the landscape, snap the vertexes of something like tree roots down to the ground, instead of having pre-generated assets that you need to try and fit on to a landscape perfectly.

Take a breathe!



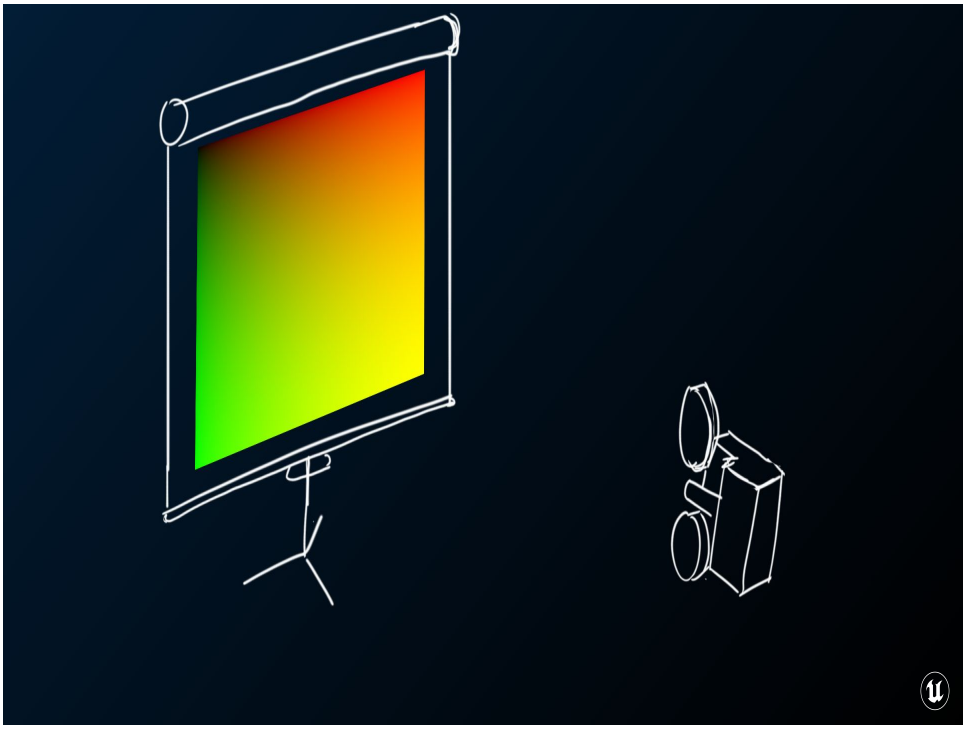
40:00

Lastly, I want to show you what you can do with Material Parameter Collections. These are global structs that any material can pick up on and react to. It also means that whenever you update these values, you only have to do it once instead of trying to manage it in multiple places.

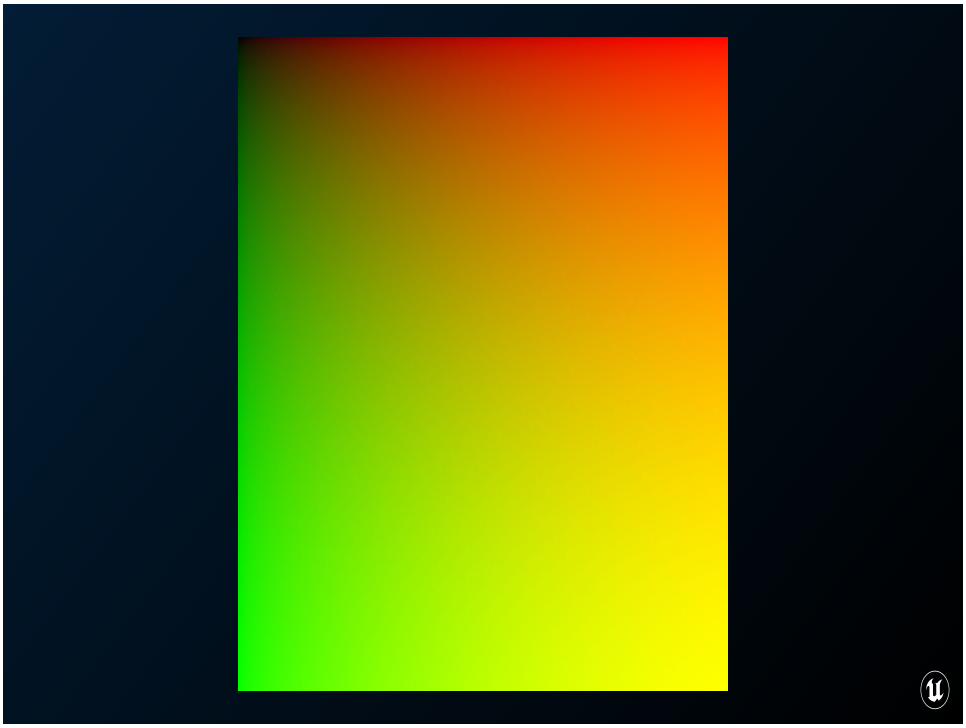
So Material Parameter Collections are great for times when you have many primitives that all need to react to either once or frequently react to the same input values

I want to show you how you can project a texture into worldspace using the transforms of a “projector” object.

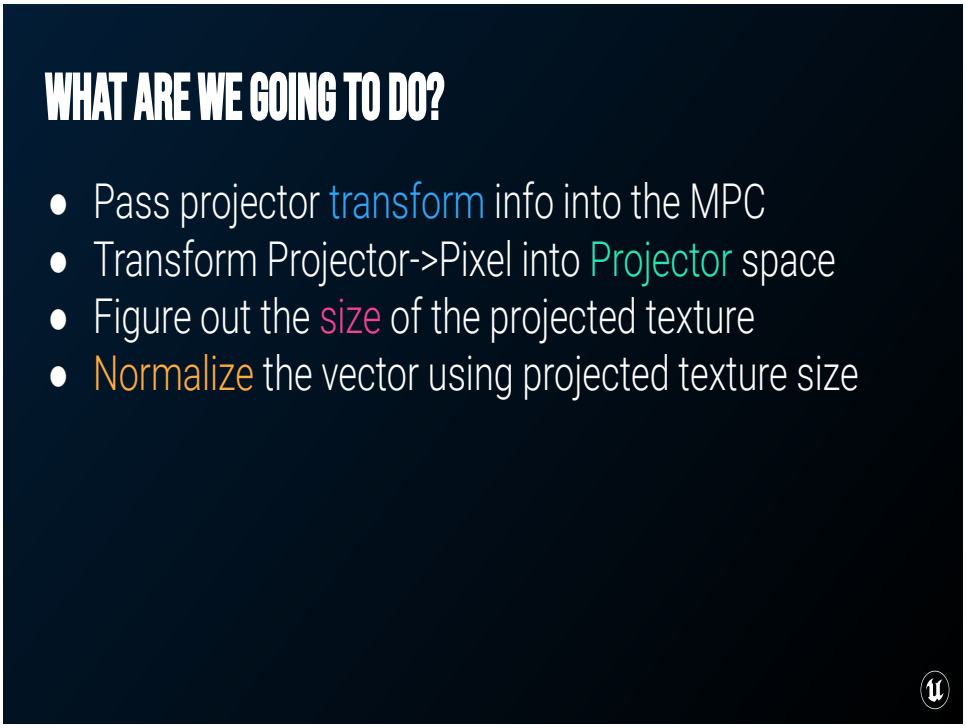
One of the things we can do with this, for example, is project a movie into a bunch of particle effects to really impress and intimidate all those who seek an audience with our great and powerful magnificence.



So, let’s say I have a screen (like that one over there) and a projector (like uhh.. that one over there!). Let’s say I projected a 0 to 1 UV coordinates image from the projector onto the screen…



Yeah! Like that! Okay, good night everybody! This is all well and good, but what if I wanted to look up those UV coordinates *in the screen's material*?



So, how can we look that up? Well, for that we need to know where the projector is, and which way it's pointing. Once we have that, we can transform the vector from Projector -> Pixel into "projector space", which will tell us how much that vector diverges from the forward vector of the projector.

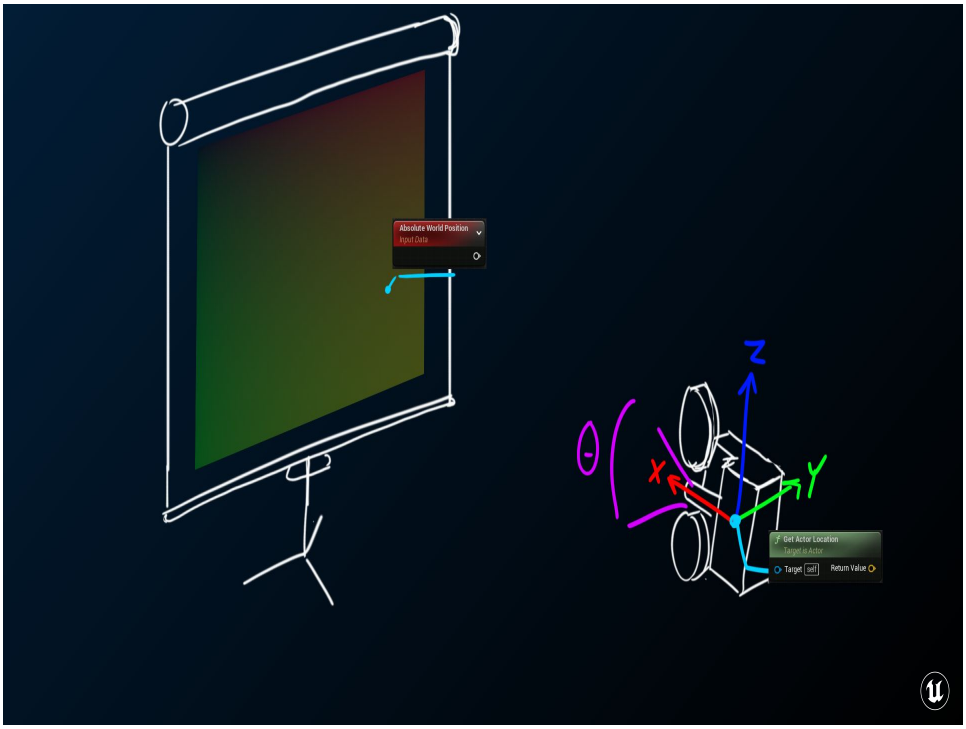
and figure out the size the projected texture would be at that point in the world (because we can't just walk up to the screen with a measuring tape)

Then we can make our final UVs by normalizing the transformed vector using the size of the texture at that point in the world

WHY ARE WE GOING TO DO IT?

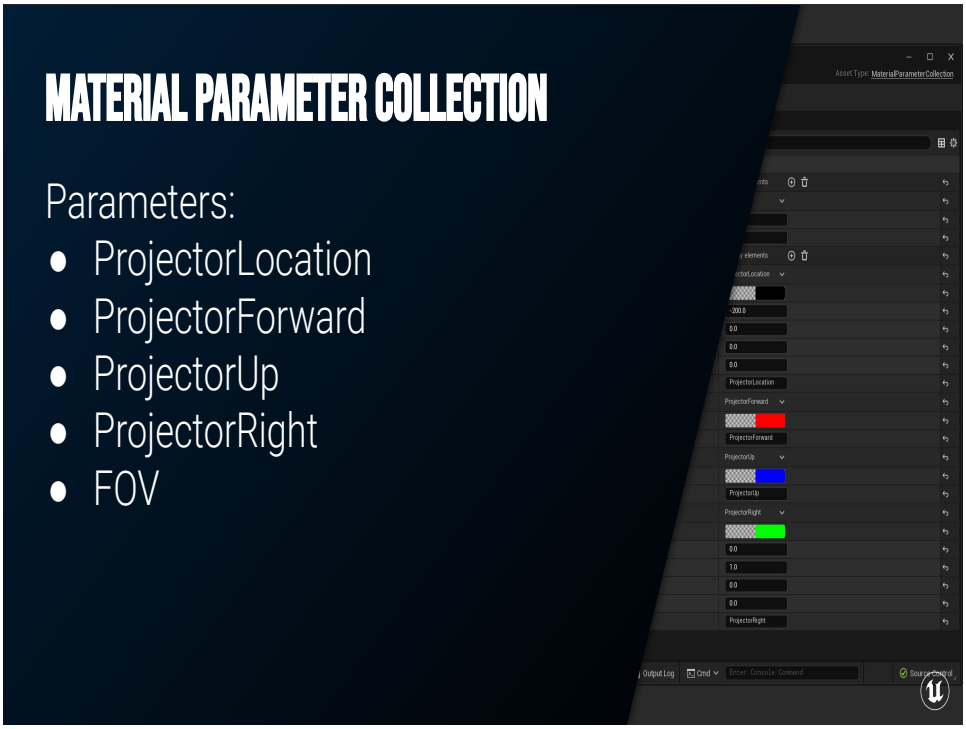
Any material can use global params

Set the value once
Sample many places

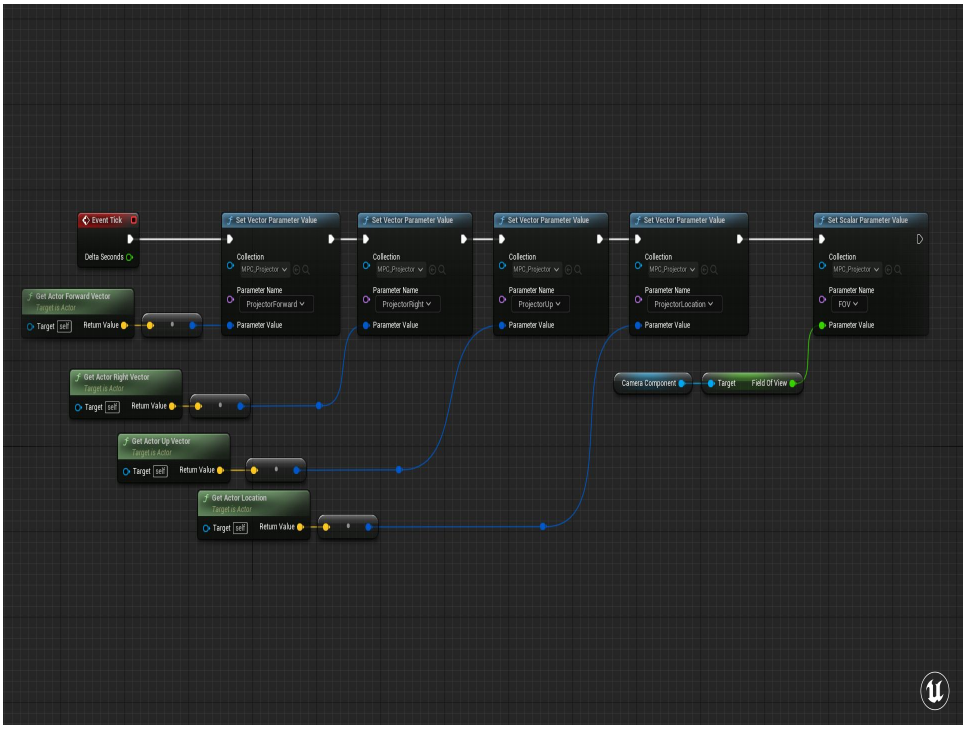


Coming back to the problem, what we know is:

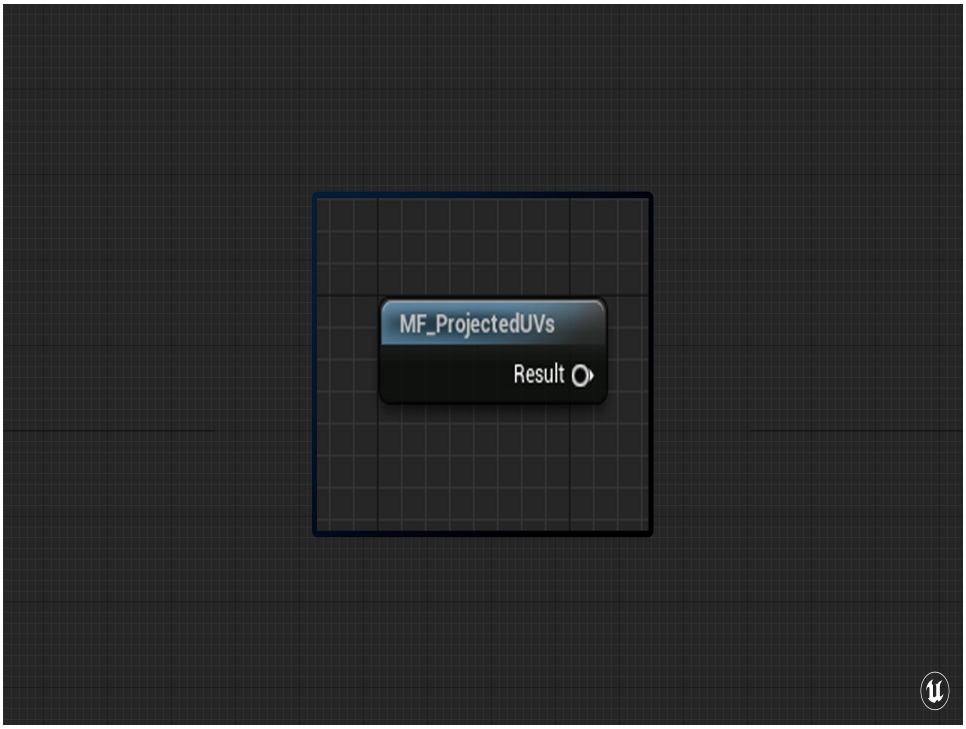
1. The FOV of the projector
2. The projector's location, forward, up, and right vectors
3. Where the pixel is in worldspace



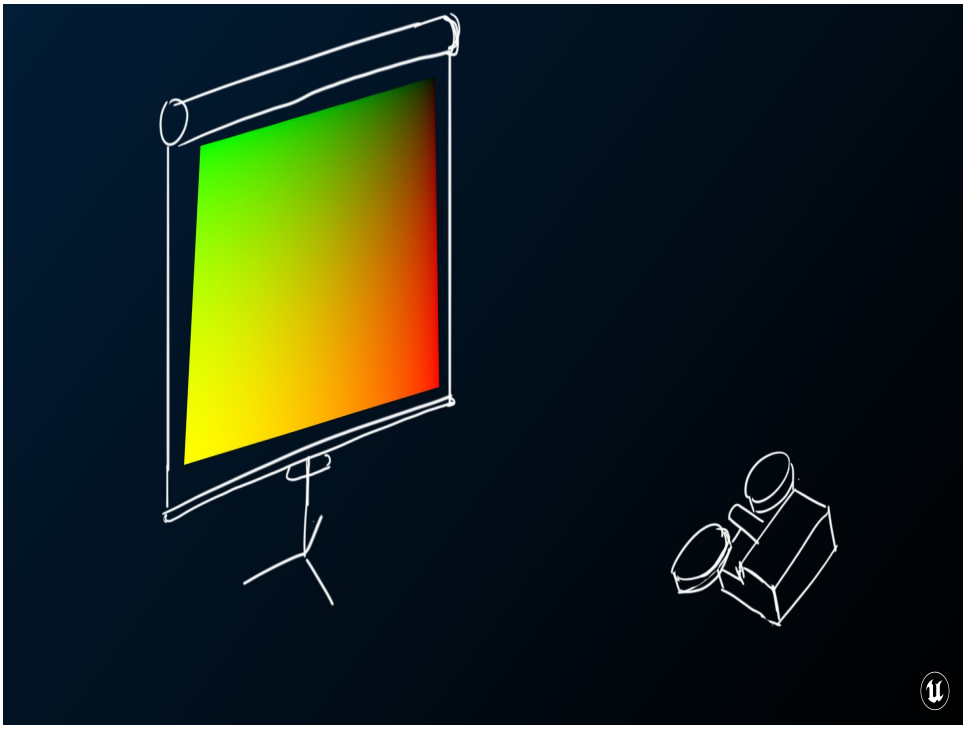
We can set up a Material Parameter Collection called MPC_Projector, and set up four vector parameters for location, forward, up, and right vectors. There's also a scalar parameter called FOV.



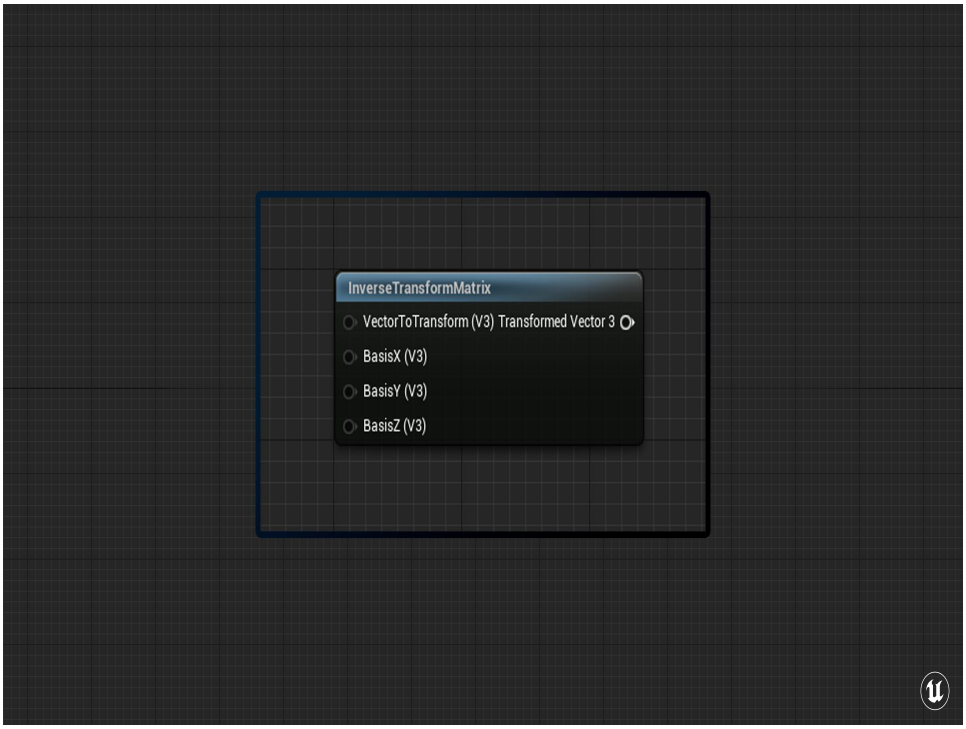
I create a Blueprint based off the Camera actor called BP_Projector. Every frame it uses Set Collection Parameter pointing to that MPC we created to pass in its location, forward, up, and right vectors, and its FOV. We're taking all our known information and sending it along to the material.



On the material side of things, I'm going to be putting all this logic into a Material Function to make it ever more portable. This is just going to output my UV values and I can use this any way I want in other materials.



So the first thing I need to do is make sure that I’m always operating in “Projector Space”, so that as the camera moves, the projection follows with it. Because again, we’re trying to figure out how much the Projector to Pixel vector diverges from the forward vector of the projector *in the projector’s space*



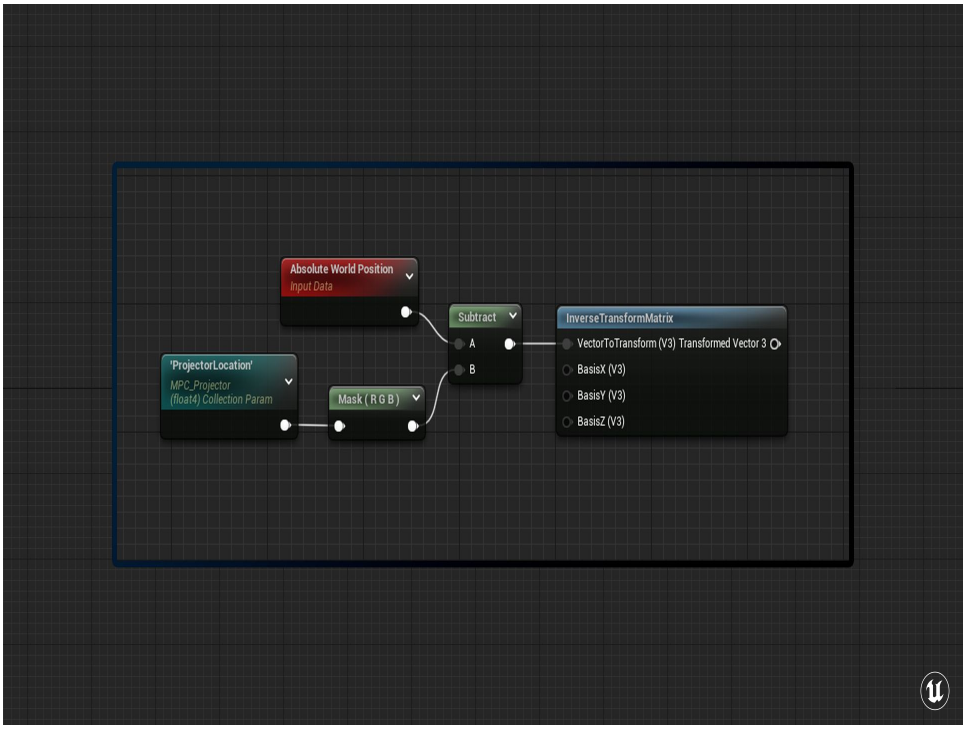
Unreal doesn't have the ability to construct and transform vectors into arbitrary spaces though, but it does have the InverseTransformMatrix material function, which does the same thing when you pass in vectors that define the space.

A quick background on matrix multiplication! Our input vector is a 1x3 matrix, and our three basis vectors make up a 3x3 matrix. The resulting matrix is a 1x3 matrix that, BASICALLY, is the dot product of each column of the basis matrix with the vector.

So this function is basically just dotting the input vector with each of the Basis, and using that as the components of the output. So it dots InVec with BasisX to get the X Component and so on. We skip trying to make a matrix with the built-in nodes and just treat handle the components of the matrix multiplication.

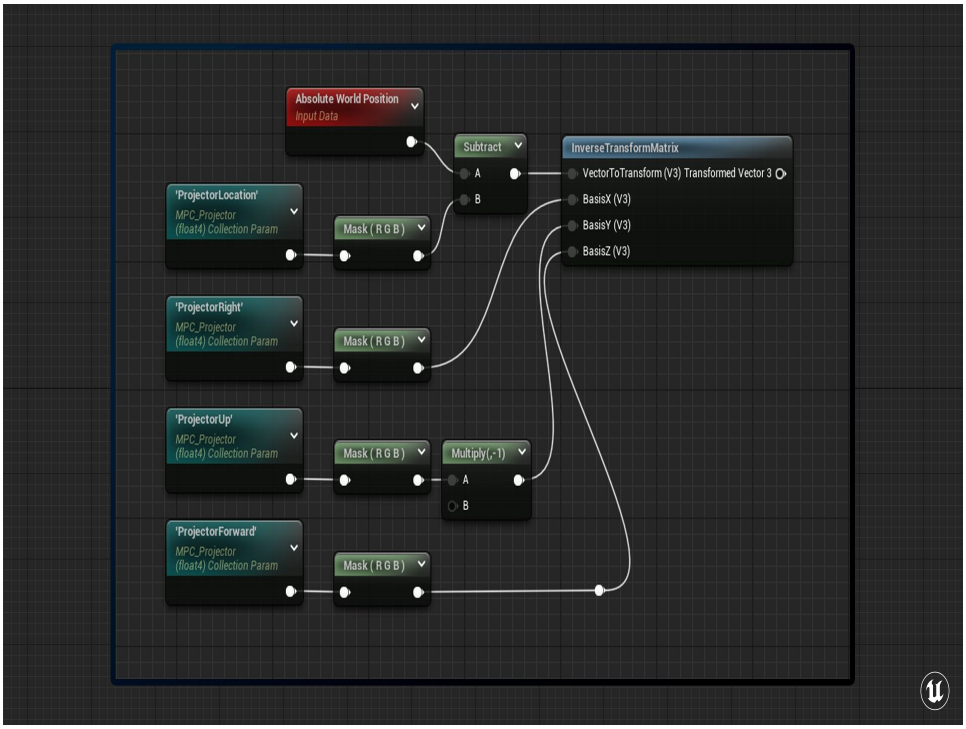
Quick refresher, dot products return a scalar value that tells

you how aligned two input vectors are. If the vectors are perpendicular to each other, then the values are 0, if they’re totally opposite then the result is negative, and if they’re totally aligned the result is positive.



We just need to pass in the Vector to Transform, which is the WorldPosition-ProjectorLocation vector.

Quick note, when you’re sampling values from a Material Parameter Collection, they automatically come in as Float4s, so I mask that out to just xyz.

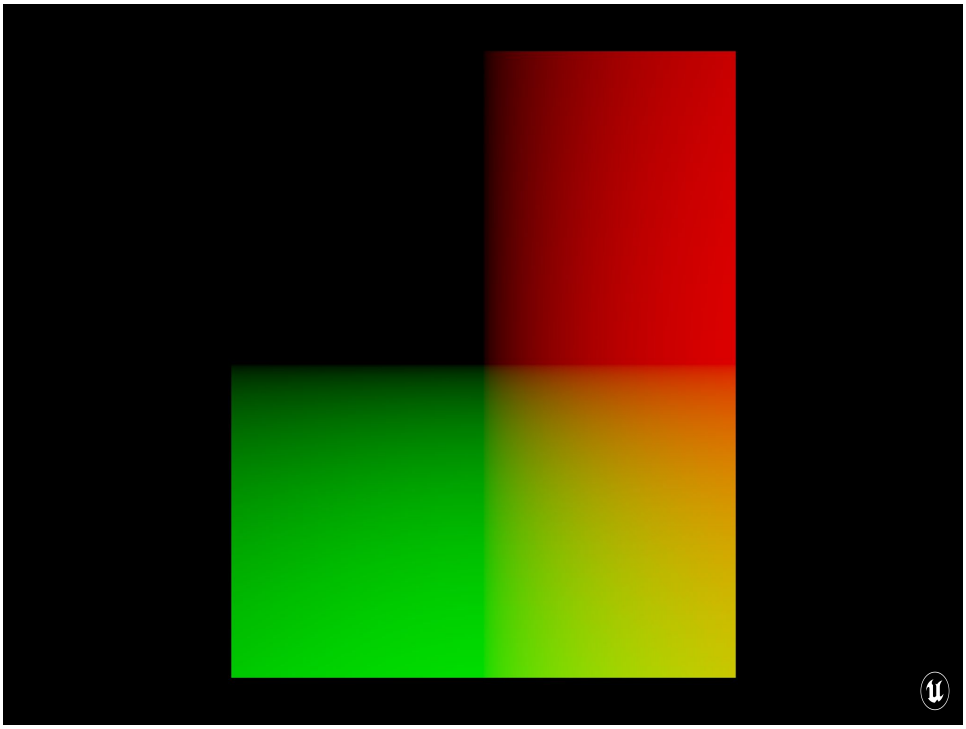


Now, this space we're constructing might look a bit strange… why would I use the projector's Right axis as the BasisX?

Well, remember how dot products give us the alignment of two vectors? And the resulting vector of this node is Pixel/Projector Dot Basis? I want the X value of the resulting vector to tell me how aligned the PixelProject vector is with the right vector. So as PixelProjector moves to the right of the projector's forward vector the value increases.

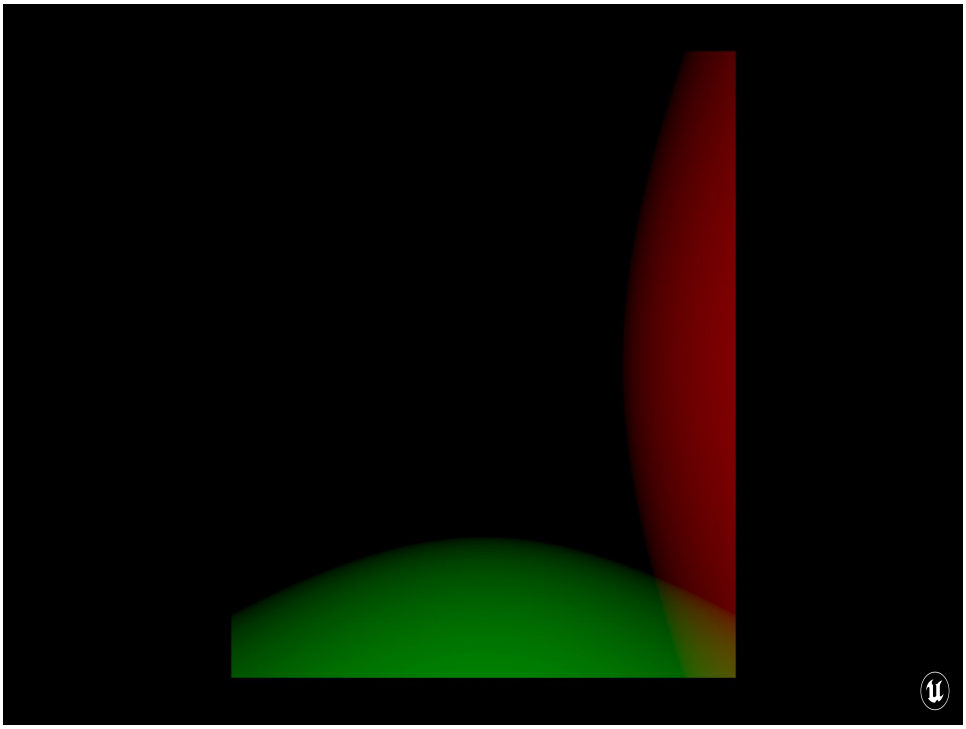
Similarly, I want the Y value of the resultant vector to give me the alignment of PixelProjector with the Up vector of the projector BUT I want that value to increase as it goes down, and decrease as it goes up, so I invert it.

Finally, to round out the space, we need the forward alignment to be the Z alignment.

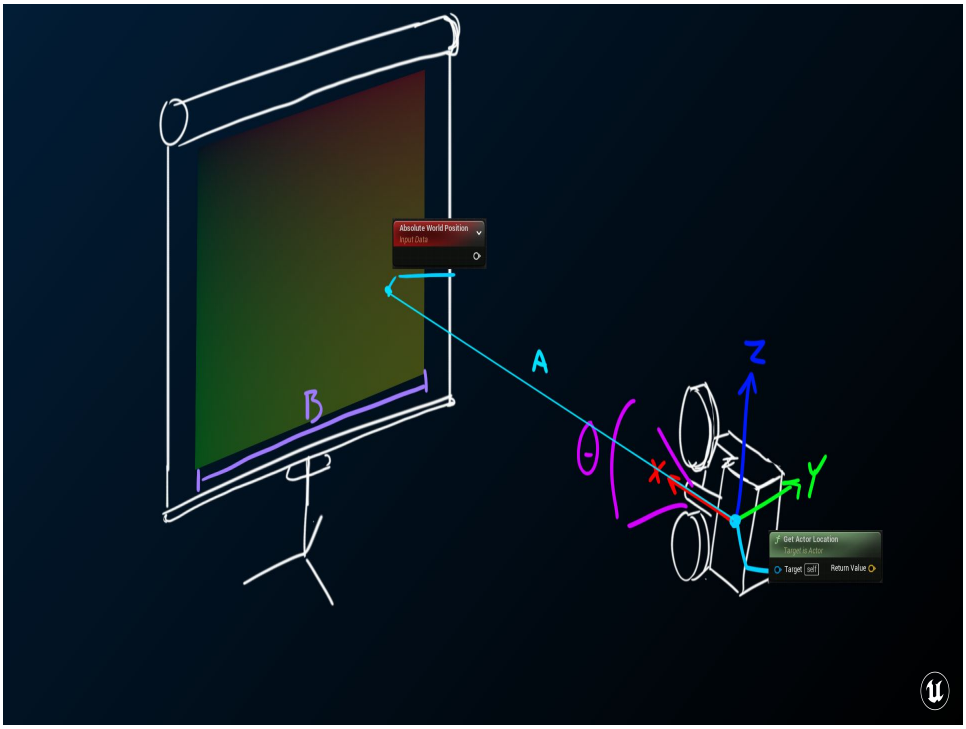


Alright, so if I mask out the Z component and normalize the transformed vector what I get is an X component that increase from 0 to 1 as the PixelProjector vector goes to the right of the forward vector. I also get a Y component that increase from 0 to 1 as the PixelProjector vector goes below the forward vector. And these look an awful lot like UVs!

But because I want my projection to be sampled with .5 .5 over the forward vector, I should just be able to subtract .5 .5 from the transformed vector to get what I need.

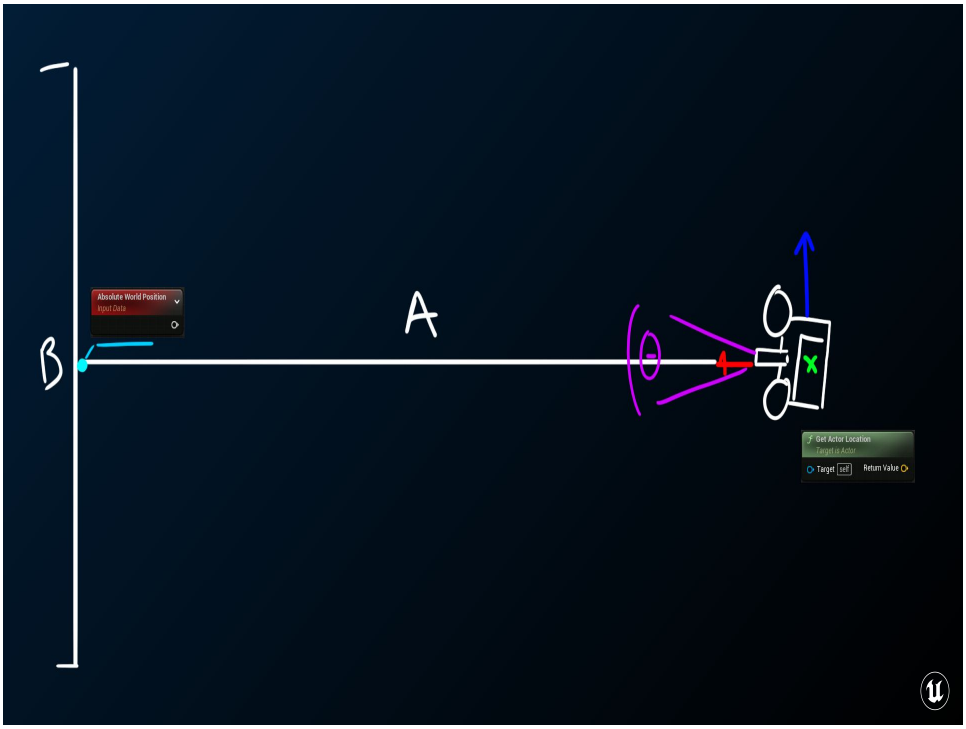


Except not, because we can’t just normalize the transformed vector against itself. The PixelProjector vector isn’t a unit vector, it’s got magnitude equal equal to the distance between the projector and the pixel. And at that, this doesn’t account for the FOV of the projector. This vector wouldn’t give us a 0 to 1 space until it reaches full down or fully right, which isn’t helpful for us at all.

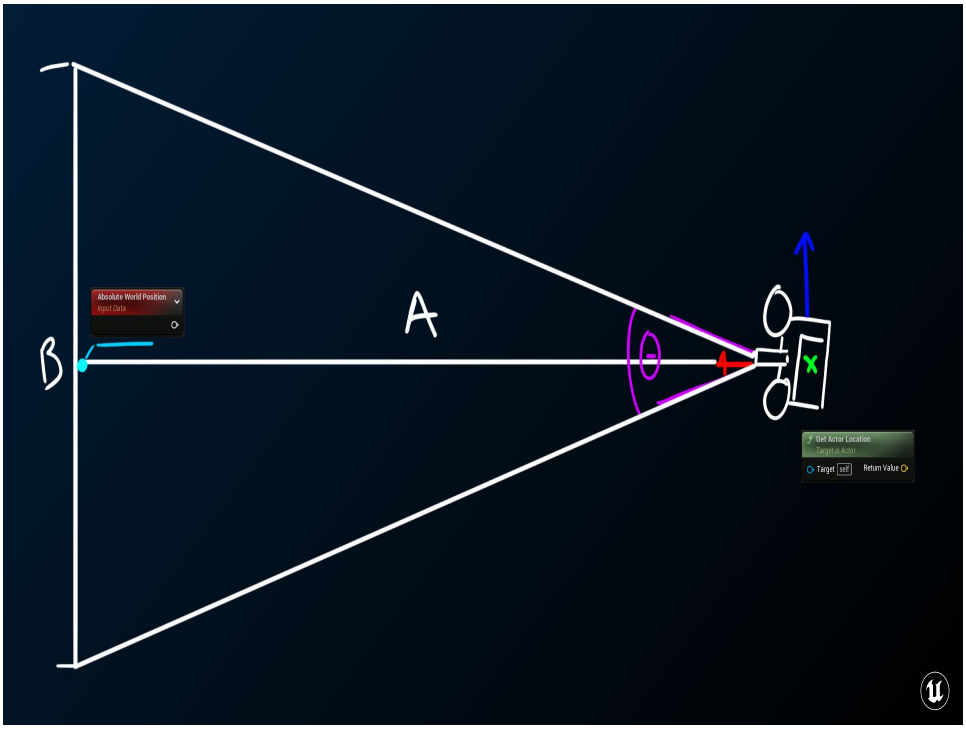


In order to properly offset and normalize that vector to the 0 to 1 space, we have to know how big the texture would be as if it was projected from our projector. We need something to divide it by!

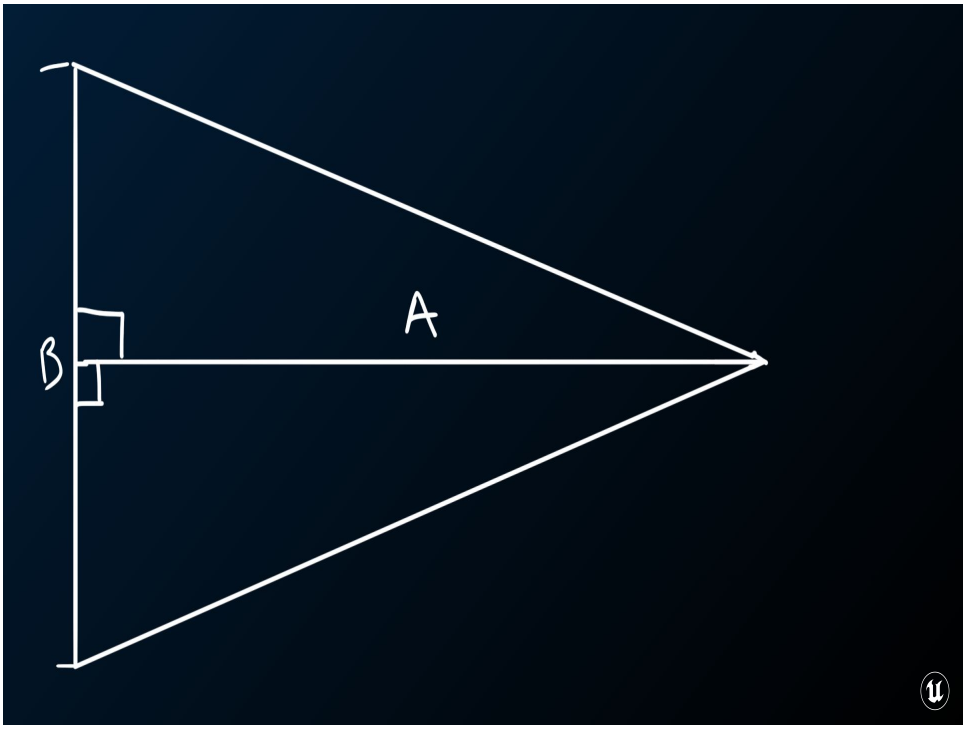
So we've got the PixelProjector vector, we know how long that is. We know the FOV of the projector…



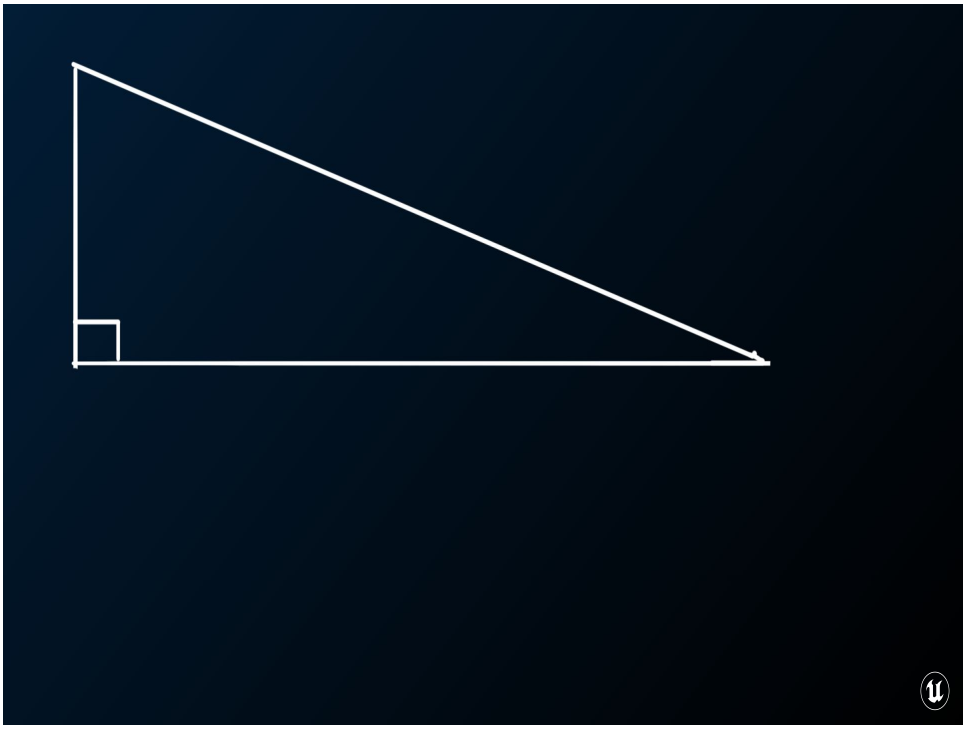
I dunno about y'all, but my brain's starting to hurt. Lemme flatten that out. Sometimes I think it's easier to solve a problem if you can just drop a dimension. Right, so I've got A, and that Theta over there. I need to solve for B.



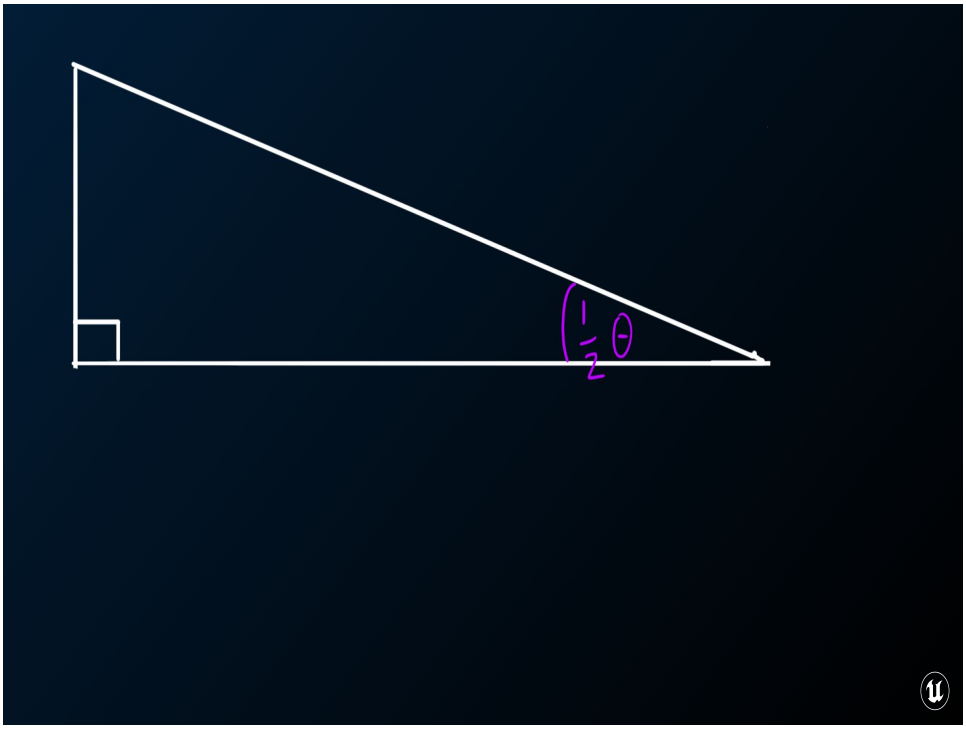
Sometimes I like to doodle while I’m problem solving, so If I extend lines from the projector out to the screen where the screen should end, that starts to look an awful lot like an isosceles triangle…



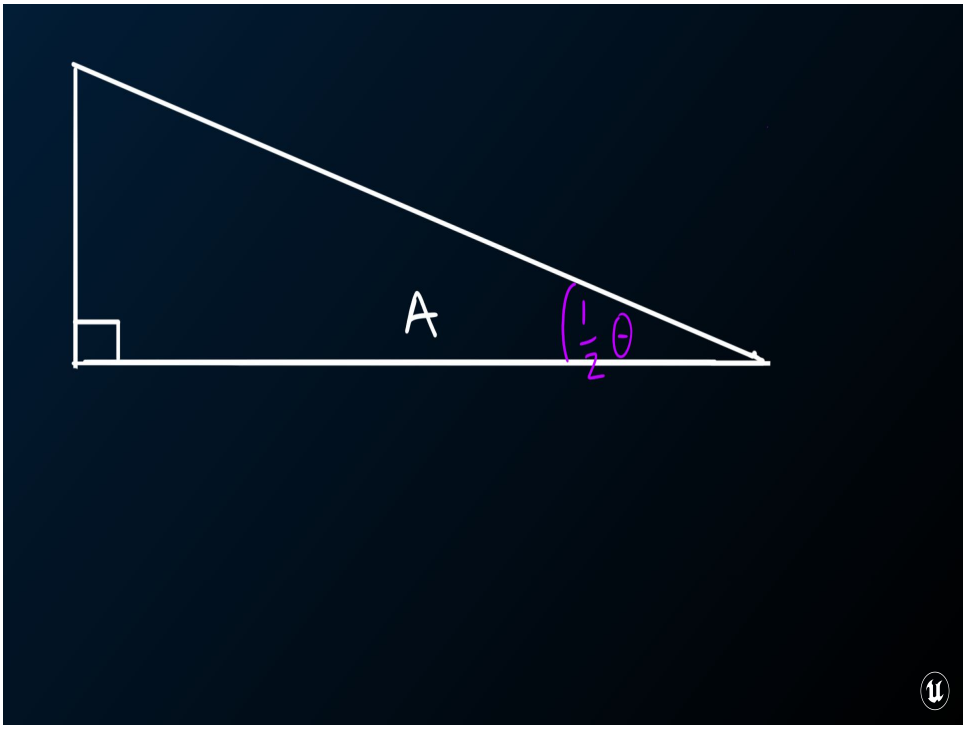
And I remember from high school that an isosceles triangle is just two right triangles wearing a trench coat…



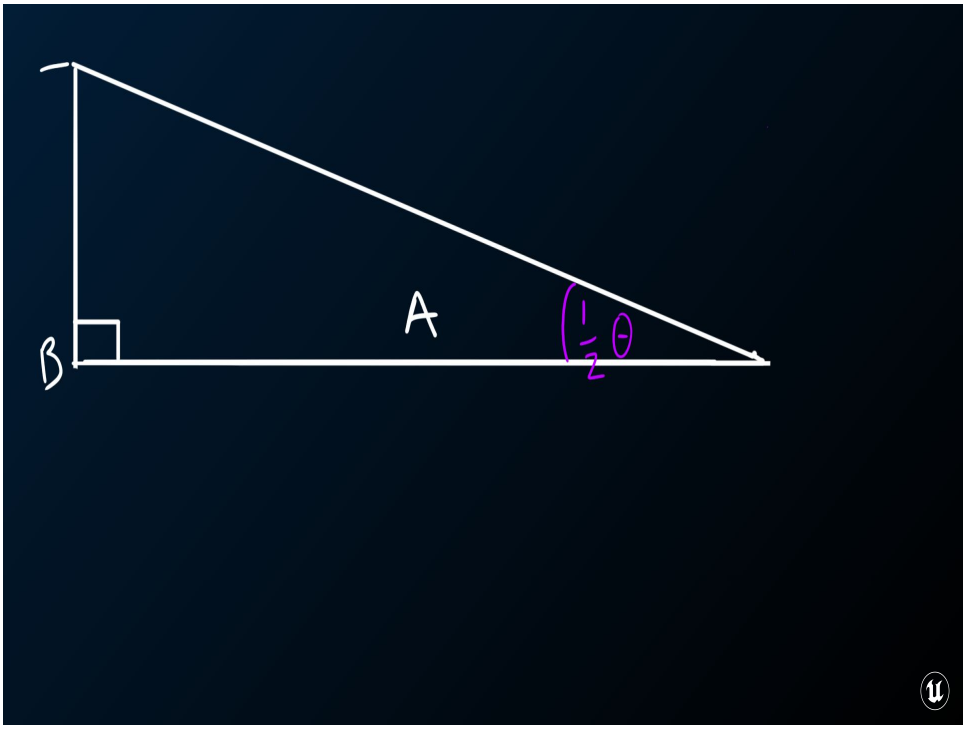
And if I remember my high school trigonometry at all, I know that you can learn a lot about a triangle given only the length of one side, and one angle.



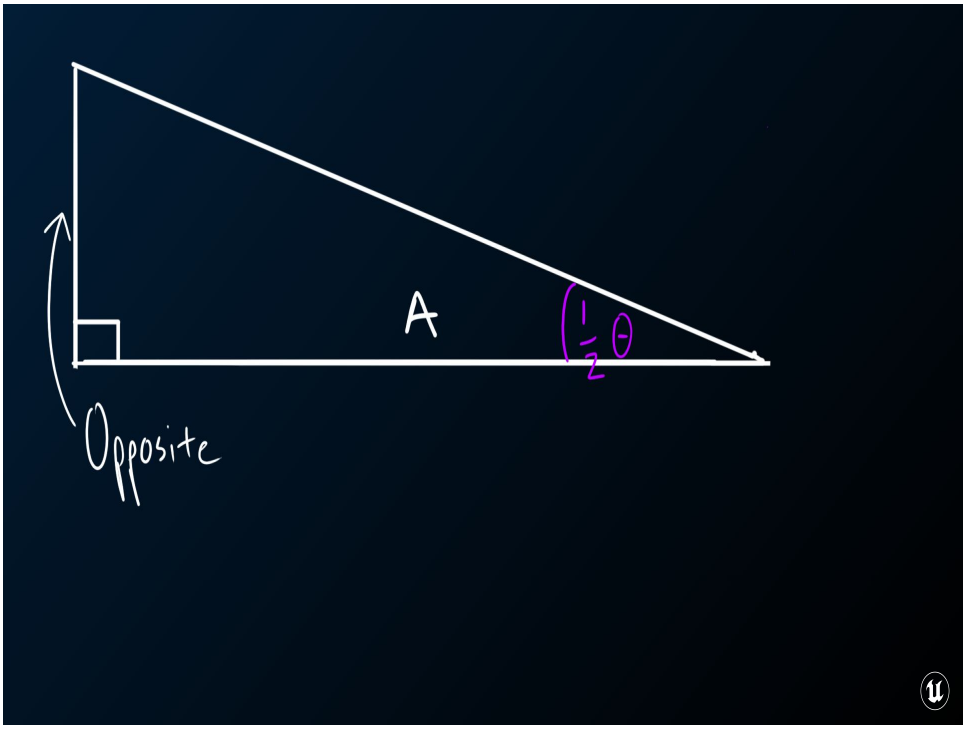
I know the FOV, and if a right triangle is half an isosceles triangle then I guess that means this angle is half the FOV



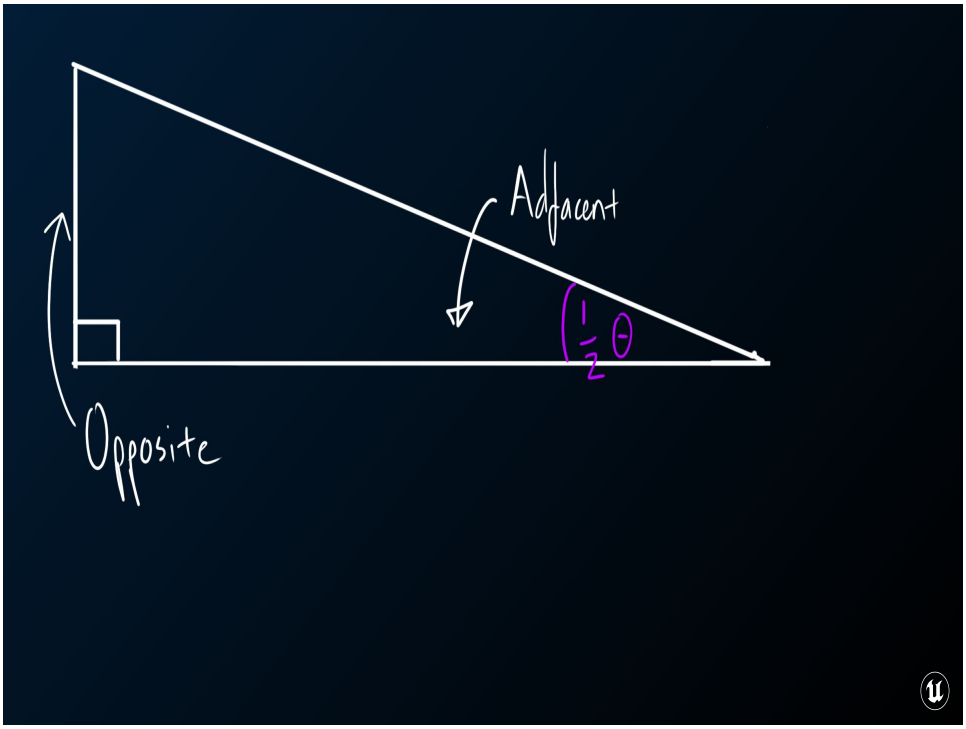
And I know the length of this side of the triangle, that's the distance from the projector to the screen



So if I can figure out how big this other side of this triangle is, I should be in good shape.



I guess this side is.. opposite of the angle?

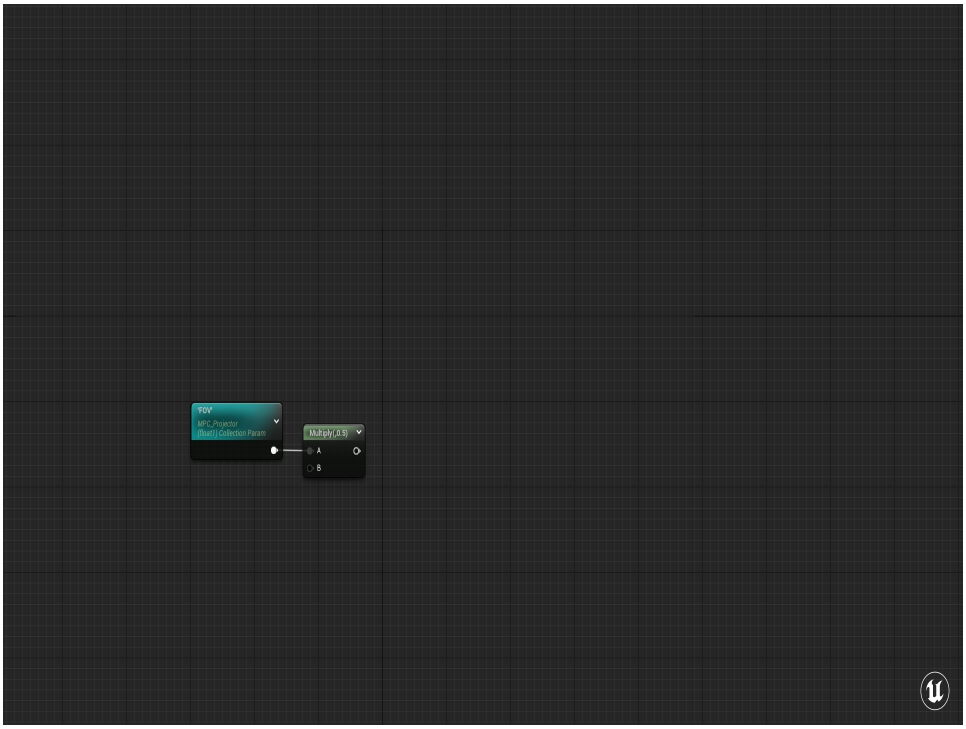


Which means this side would be adjacent the angle. Gee, I really wish I remembered high school trigonometry… Wasn't there some kind of acronym that was supposed to help me remember this stuff?

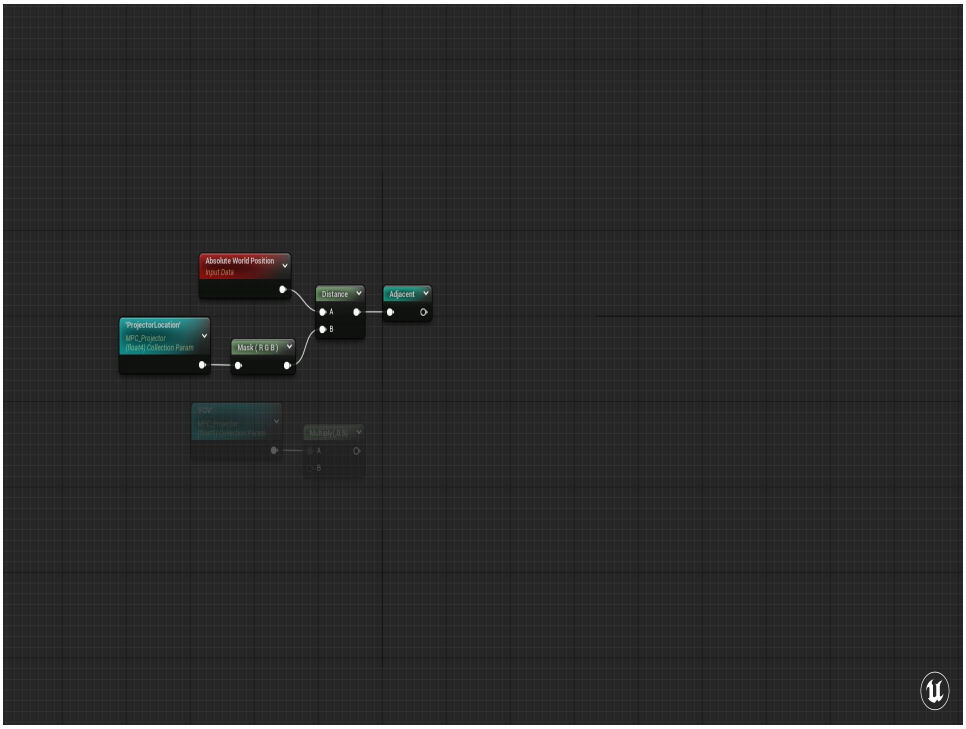
OH RIGHT, Soak a Toe-a!

Boy, really had to dust off some stuff there. This means we know that the tangent of the angle is equal to the Opposite side divided by the Adjacent side.

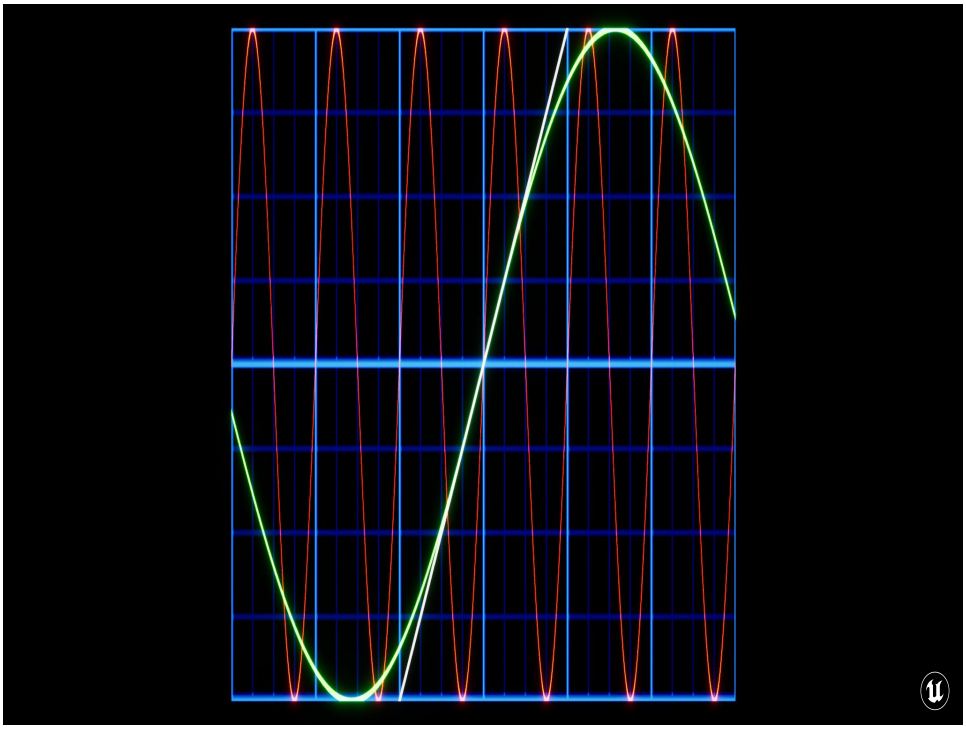
and doing a bit of algebra that means tan(theta) * Adjacent = Opposite. Phew, okay, let’s get back into Unreal before I get scared.



We’re passing in the FOV of the camera, so all we have to do is halve that.

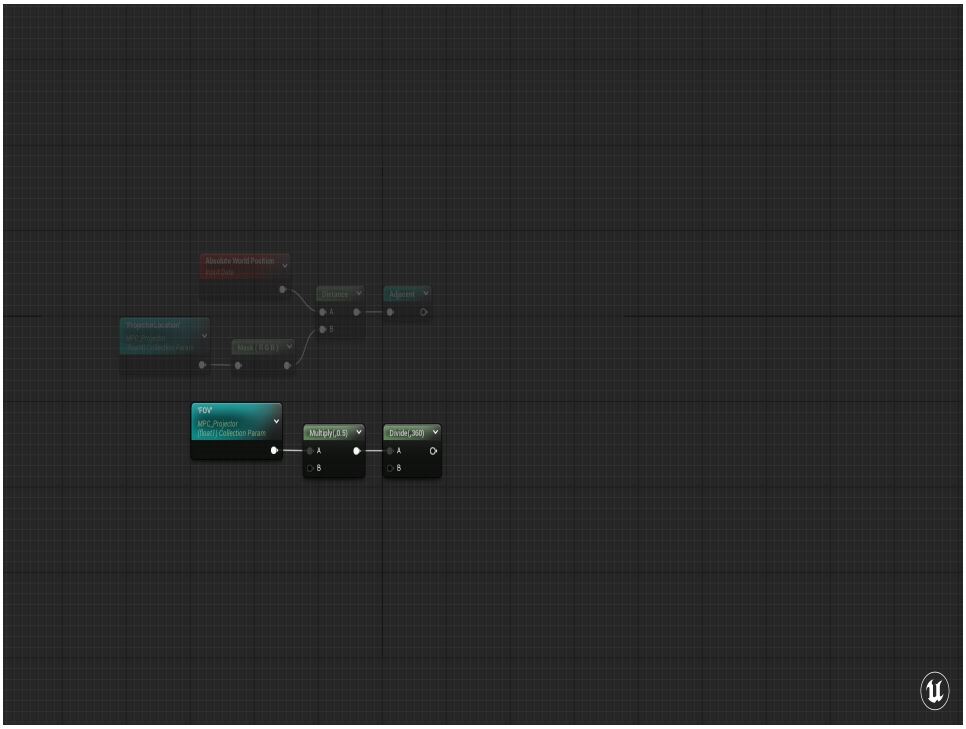


Then we can get the adjacent side as the distance between the pixel and the projector. To make things a little more clear I’m using the new Named Reroute nodes. There’s gonna be a lot to keep track of in this material.

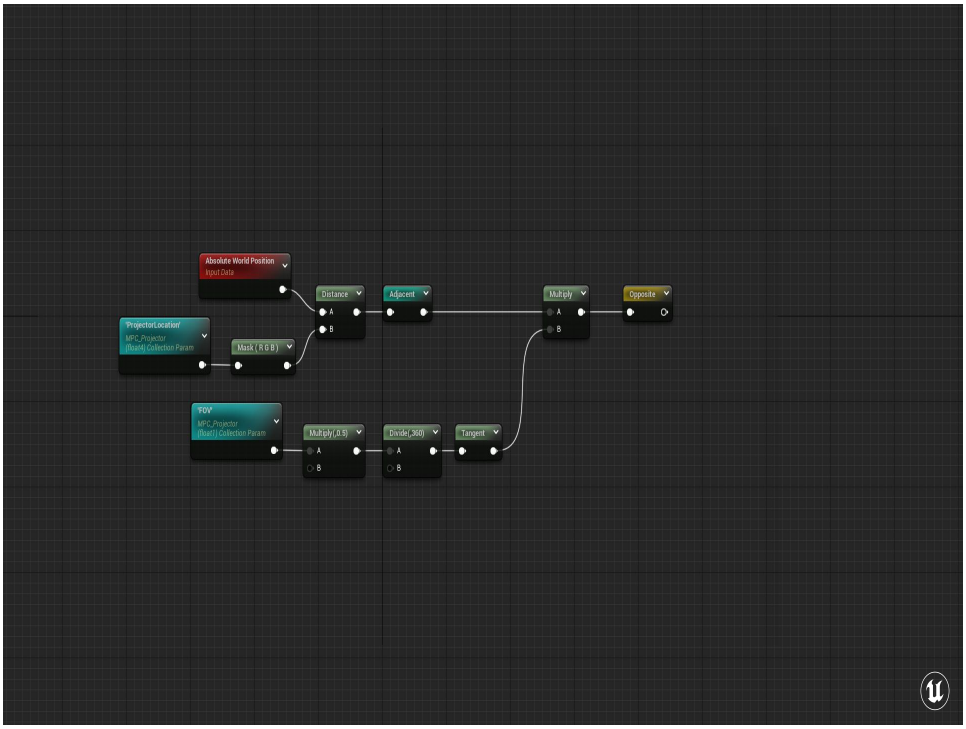


Then we need the tangent of our angle. The FOV is input as degrees, and in the Unreal Engine our material trig functions are in a period 1, or unitless. This is a big difference from your graphing calculator, which uses a period of 2pi. I graphed those out here so you can see the difference.

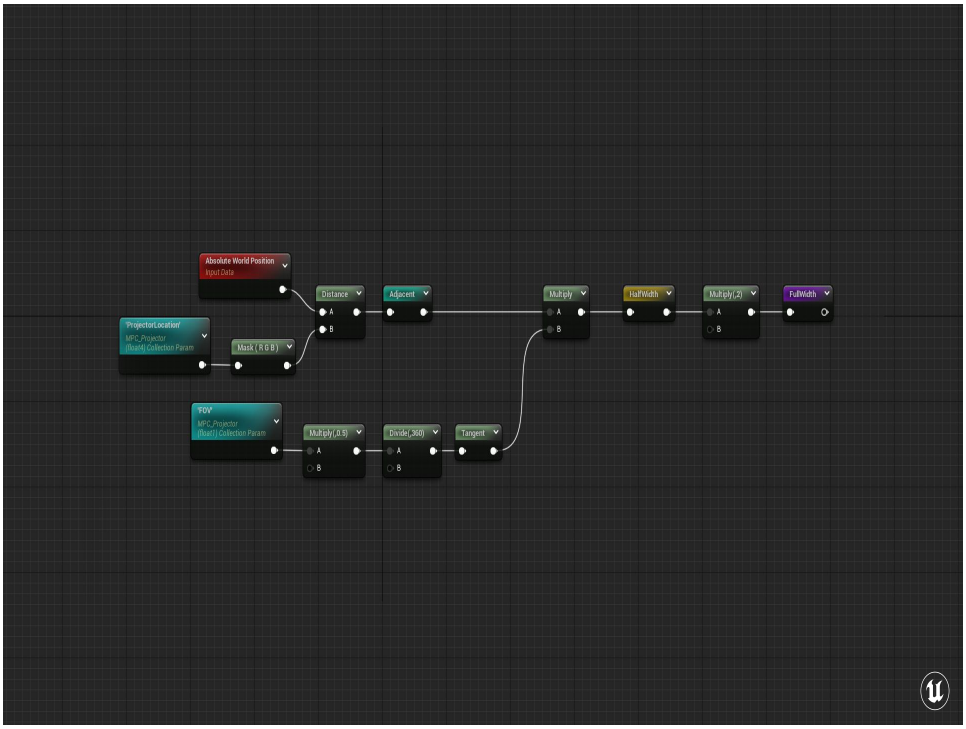
So one 2pi radians is 360 / 2pi degrees or about 57. For period 1 radians, a radian is just 1 degree



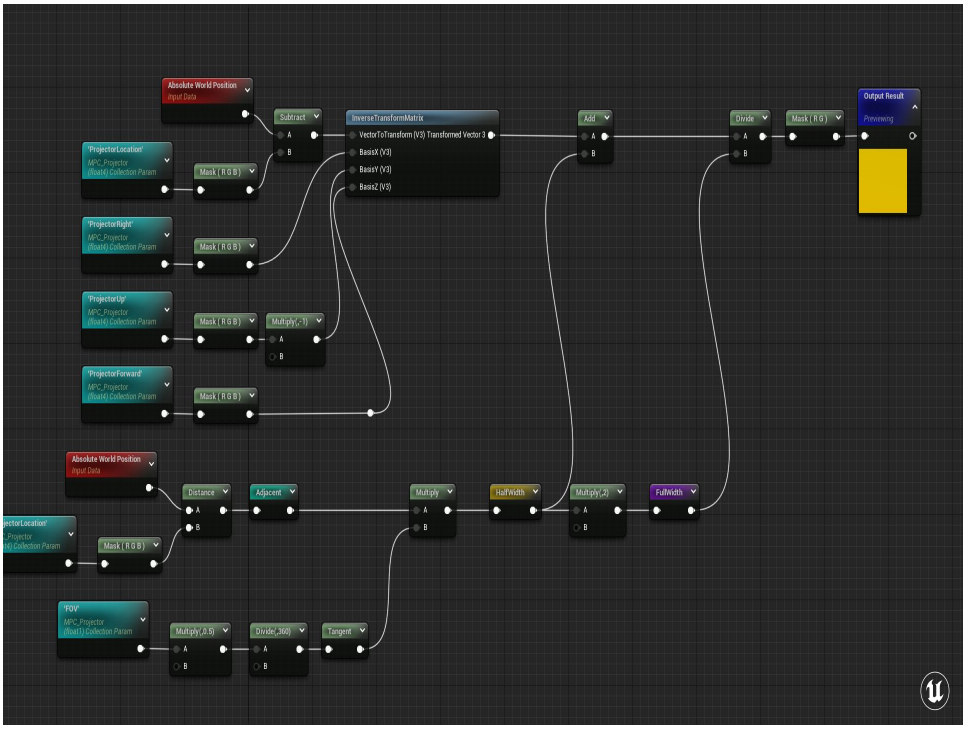
Which means that in order to convert our FOV’s degrees to “Radians” we just need to divide it by 360.



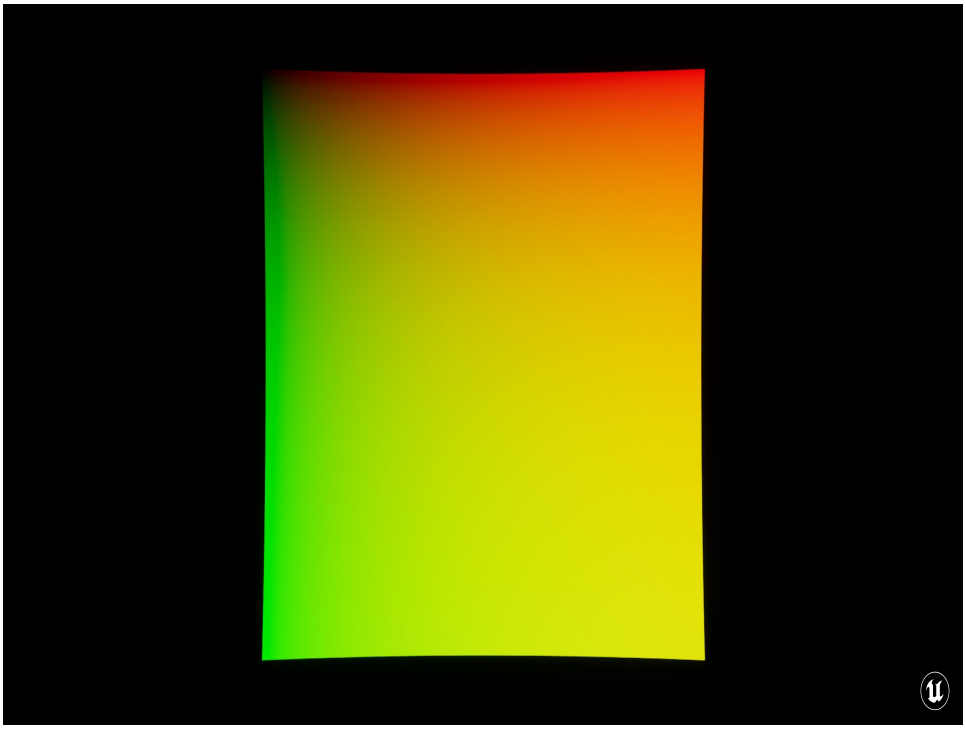
Then multiply tan(theta) by adjacent to get half the size of the texture at the Adjacent distance.



Now that we have the size of the opposite side of the triangle with half the FOV, that means that the full size of the projected texture is double that.

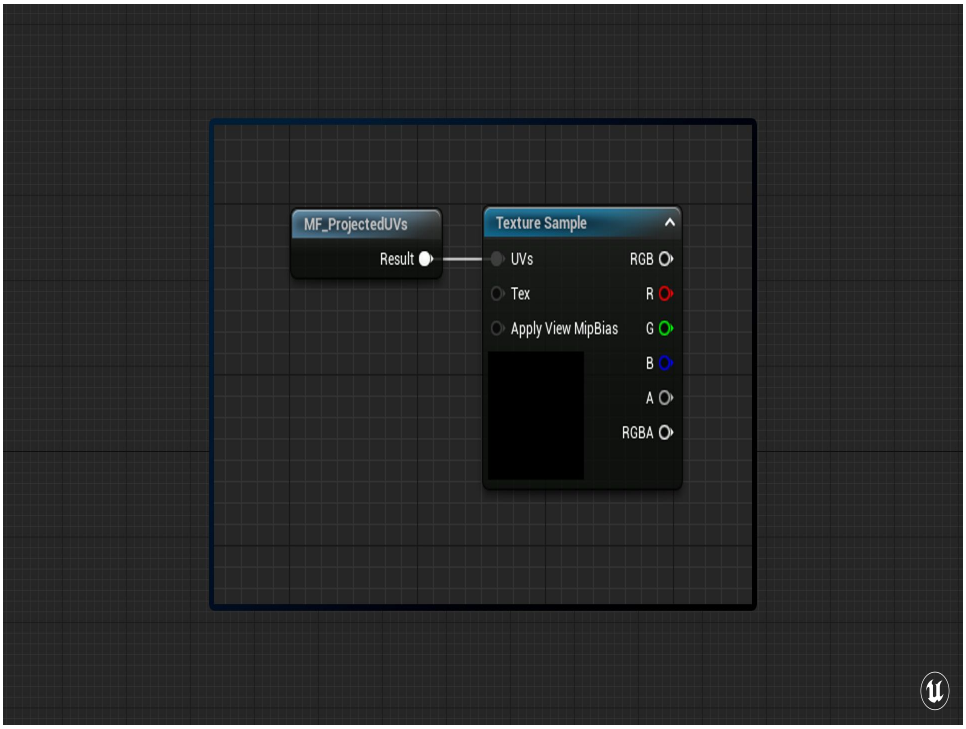


And because right now our “UVs” aren’t centered at .5, .5 we need to bias it over by half the width of the texture and dividing that by the full width of the texture to get the values from 0 to 1.



And taking a look at that in the preview, it looks like our UVs have shifted so that .5/.5 is right over the forward axis.

(For the sake of clarity, I've masked it out ot just the 0 to 1 space)



Then all you have to do is use that material function outputting its UVs into a texture sampler. For this example, this is all I need to look sample the looping video for the man behind the curtain. From here I can plug this into the rest of my emissive stack to blend it in however I need!

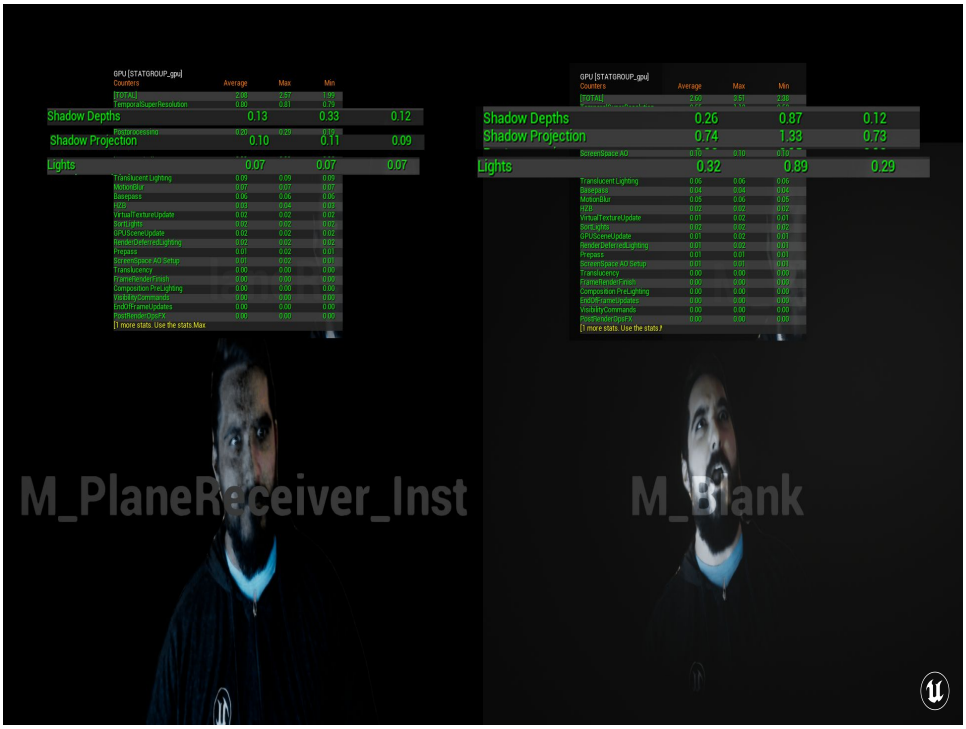
And behold! All of these particle effects are using that same material function created earlier to sample the little video of our wizard's face. All of this is happening on the particle, and none of it is happening with more expensive alternatives like light functions.

In the Unreal Engine, you can apply a material to a spotlight, so why not use that? That seemed like an awful lot of work!

Light functions have a few limitations that didn't make them perfectly suited for this. For example, light functions are greyscale, they only change the intensity of the cast light, not its color, so for a full-color projection you'd need to place a Red, Green, and Blue light with the same material.

Which can be complicated, since adding a light function to a light means that the light will be treated as a shadow-caster, and dynamic. This may be prohibitively expensive for your performance budgets.

Instead, with a bit of information from the outside world, you can look up your texture in your receiving materials with a bit of math, and without the overhead of shadowcasting lights.



To test the performance differences here, I again set up a full-screen quad with two different materials: one that displays the projected texture like we were doing with the particles, and one that's just a default lit, grey material. When I swap to the grey material I switch to a three spot-light set that are each throwing Red, Green, and Blue light respectively, along with a light function material that's just playing the video.

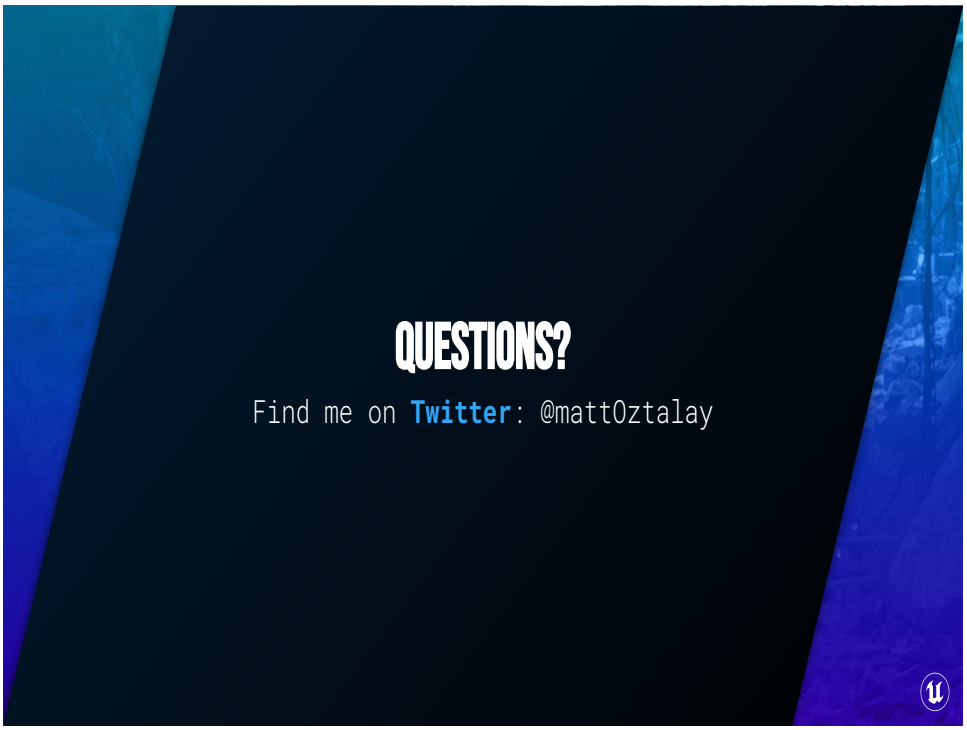
The difference in base pass costs was negligible (at least on my machine), but the lightfunction version added about 1ms between Lights, ShadowDepths, and ShadowProjection

This has actually come up a few times over the last couple of years, and it's been a surprisingly useful tool in my belt, so I thought I'd share it with y'all today. Here's a few of the ways I've used it, and if this talk inspires you at all I'd love to find out how you've used it too!

Take a breathe!

Before I wrap up, I gotta say a few thank-yous and acknowledgements:

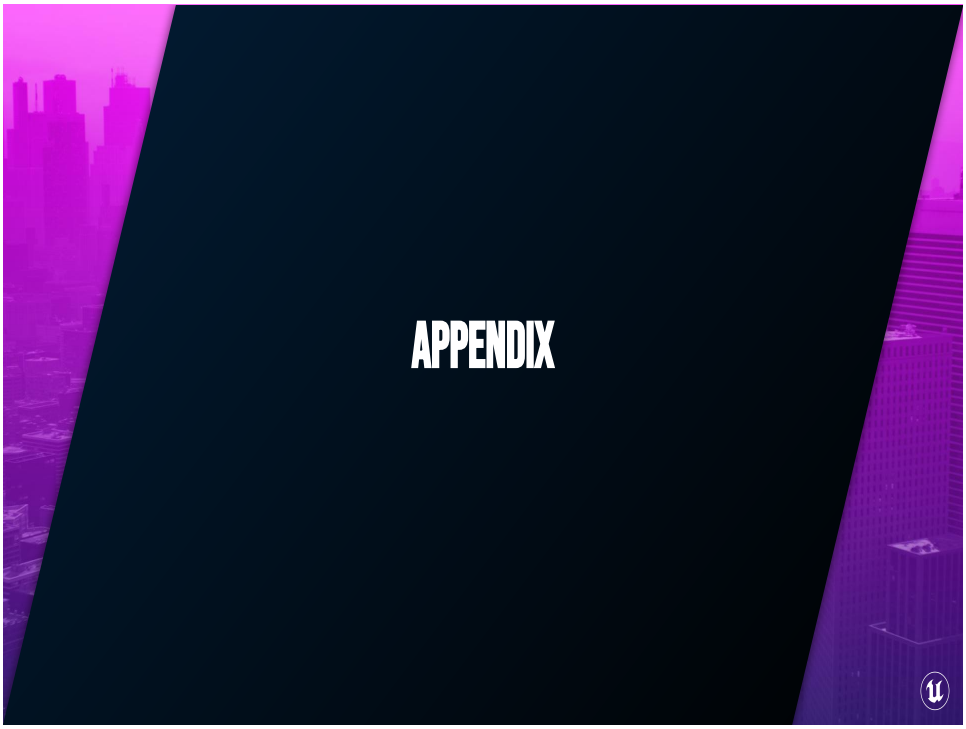
- I absolutely would not be on the stage today without the incredible support of the summit organizers, Marys Cassin and Denman. Congratulations on your third tech art summit, I'm super excited for the rest of today's speakers.
- I don't think I would be standing here today were it not for the groundwork laid by Ben Cloward, upon whose mighty shoulders I one day hope to stand.
- I'd also like to give a big shout out to the team at Quixel, whose assets I used in the landscape demo.
- And finally, a big thank you to my colleagues at Epic Games for their invaluable feedback on the presentation.



55:00
Thank you all so much for joining me today. I hope this has been somewhat informative, and I look forward to seeing what you're able to make with these tricks. For more Unreal Engine tips and tricks, you can always follow me on Twitter, @mattOztalay. I'll also post there when I release the project files for the talk!

Thank you, and good night!


EPIC
GAMES

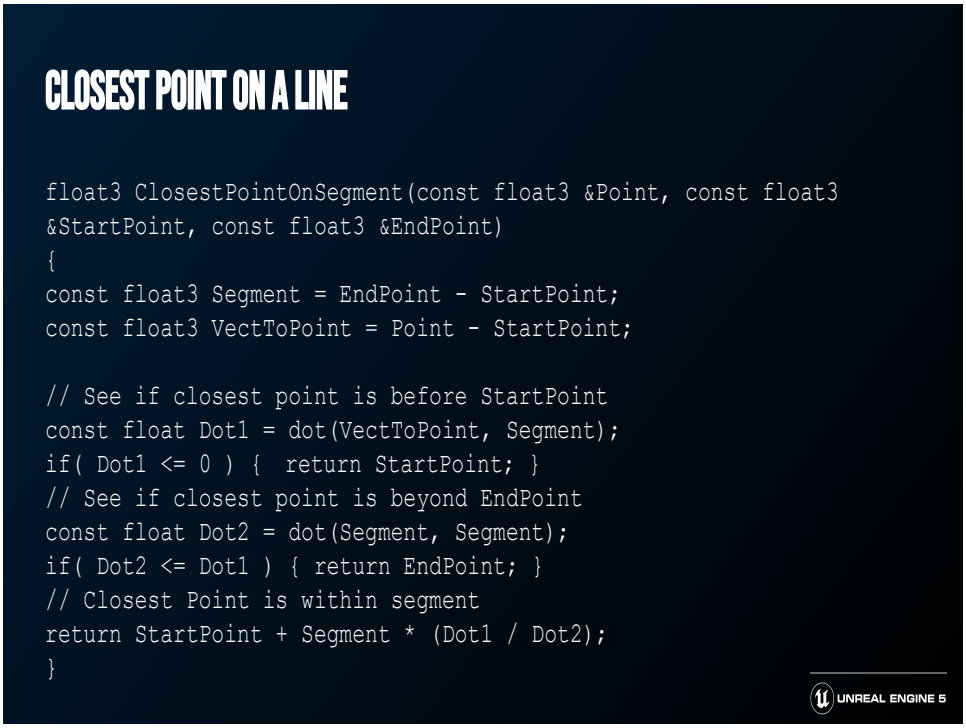

APPENDIX

ROTATE ABOUT AXIS

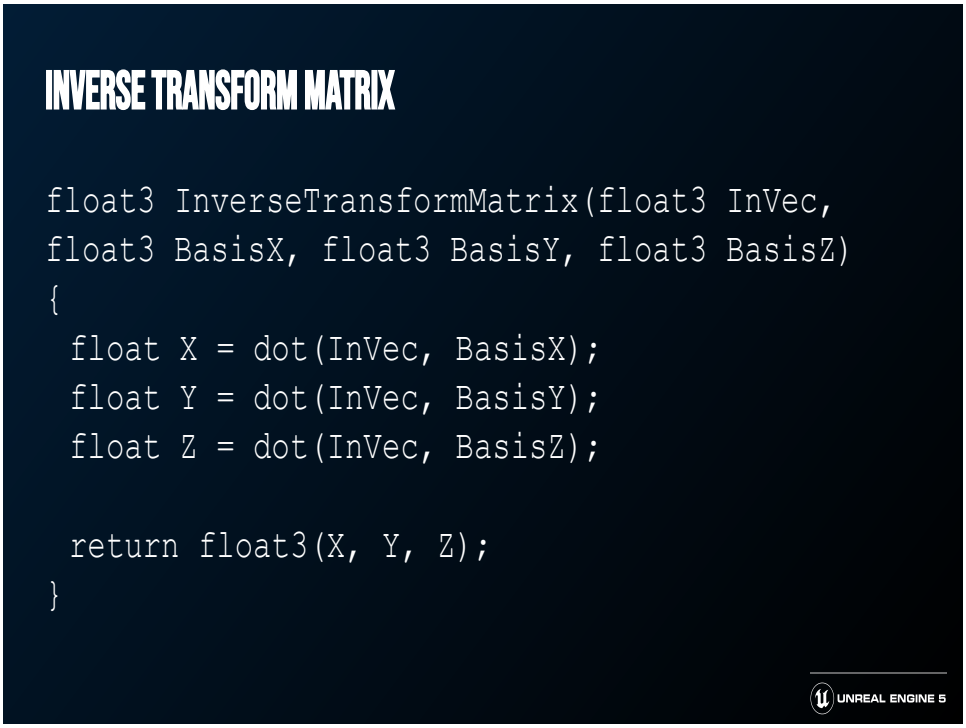
```
float3 RotateAboutAxis(float3 Axis, float3 Pivot, float3 Position, float Angle);
{
// Project Position onto axis and find the closest point on the axis to Position
float3 ClosestPointOnAxis = Pivot + Axis * dot(Axis, Position - Pivot);
// Construct orthogonal axes in the plane of the rotation
float3 UAxis = Position - ClosestPointOnAxis;
float3 VAxis = cross(Axis, UAxis);
float CosAngle = cos(Angle);
float SinAngle = sin(Angle);
// Rotate using the orthogonal axes
float3 R = UAxis * CosAngle + VAxis * SinAngle;
// Reconstruct the rotated world space position
float3 RotatedPosition = ClosestPointOnAxis + R;

// Convert from position to a position offset
return RotatedPosition - Position;
}
```

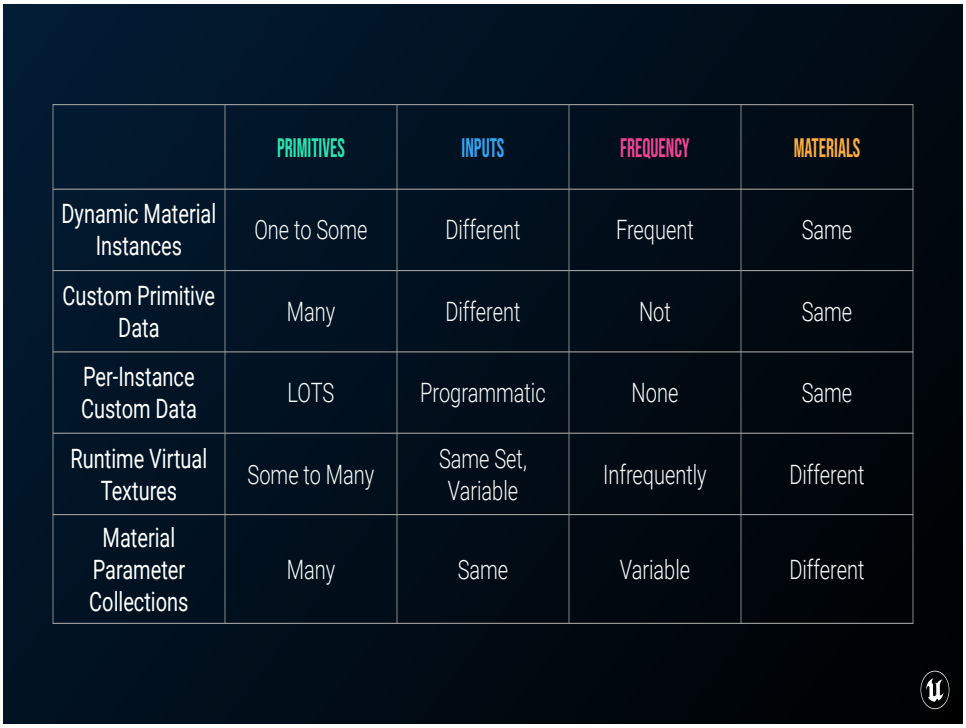
Method for rotating an input position around an axis from a pivot point by Angle% of a full rotation. Written in HLSL.



Method for finding the closest point on a line, spelled out using HLSL-like pseudocode

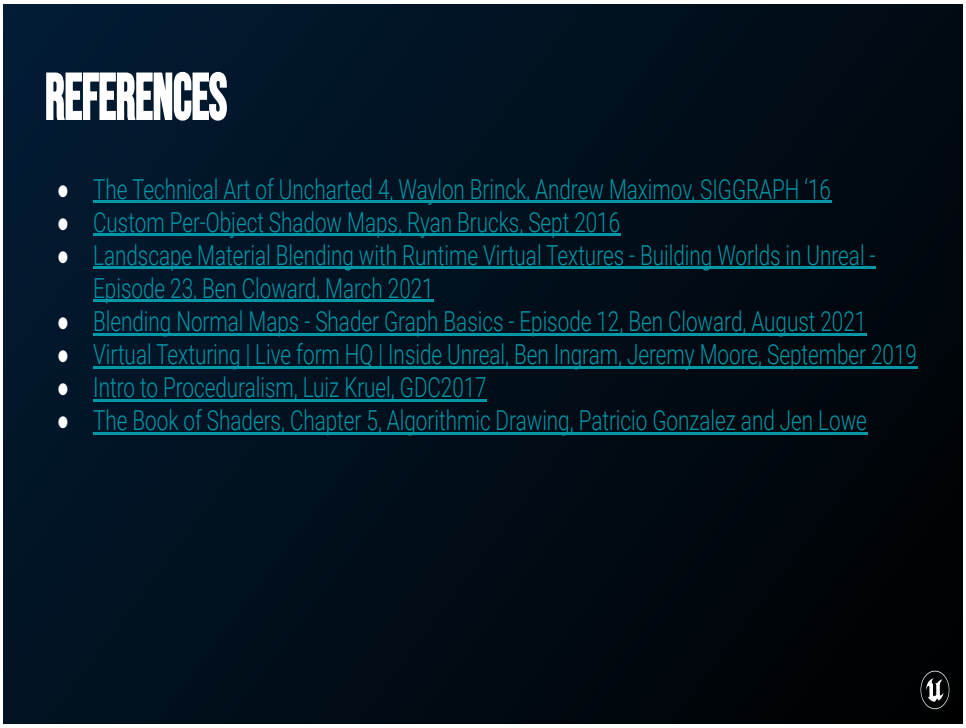


The InverseTransformMatrix material function, spelled out using functions available as graph nodes in the material editor.



	PRIMITIVES	INPUTS	FREQUENCY	MATERIALS
Dynamic Material Instances	One to Some	Different	Frequent	Same
Custom Primitive Data	Many	Different	Not	Same
Per-Instance Custom Data	LOTS	Programmatic	None	Same
Runtime Virtual Textures	Some to Many	Same Set, Variable	Infrequently	Different
Material Parameter Collections	Many	Same	Variable	Different

Wrapping all that up, here's each of the different techniques and their optimal uses.



If you're interested in more of the math behind some of these techniques, or want to explore some other ways to pass information into materials I've included some references