



DISTANCE FIELDS AND SIMULATION TRICKS IN UE4

By Ryan Brucks

Introduction

About Me

- Started as **Intern** at Epic 15 years ago
- Previously worked as **LD** and **Environment Artist**
- Currently Principal **Technical Artist**
- Mixture of R&D for tech demos and game development



Some quick background about me so people realize I did not start out as a tech artist, but rather slowly eased into it.

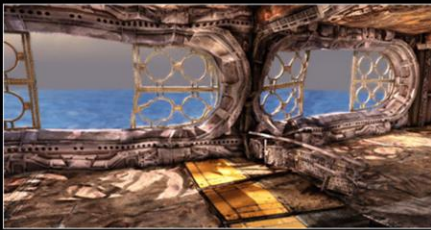
I hope to be able to demonstrate the link between how artists often intuitively approach solving problems and a more classical mathematical approach, and the union between them.

Overview

Analytical Simulation: Driving behavior with math

Going over making **Effects** using **Vertex Shaders**

Using **Distance Fields** and **UVs** to model behavior



This talk is meant to give people insight into different ways to think about creating effects where most things are done using vertex shaders and driven parametrically.

After going over some basic behavior modelling, we will show ways to use more advanced effects utilizing methods like distance fields or analytical gradients construction from easily accessible sources like mesh UVs.

While it will present some mathematical concepts, the goal is to not assume much if any prior knowledge and break things down from the simplest level up to more complex, interesting examples.

Analytical Simulation

Basic Description

- Modeling Behavior with Functions
- Parameters as inputs
- Replaces 'regular simulation'

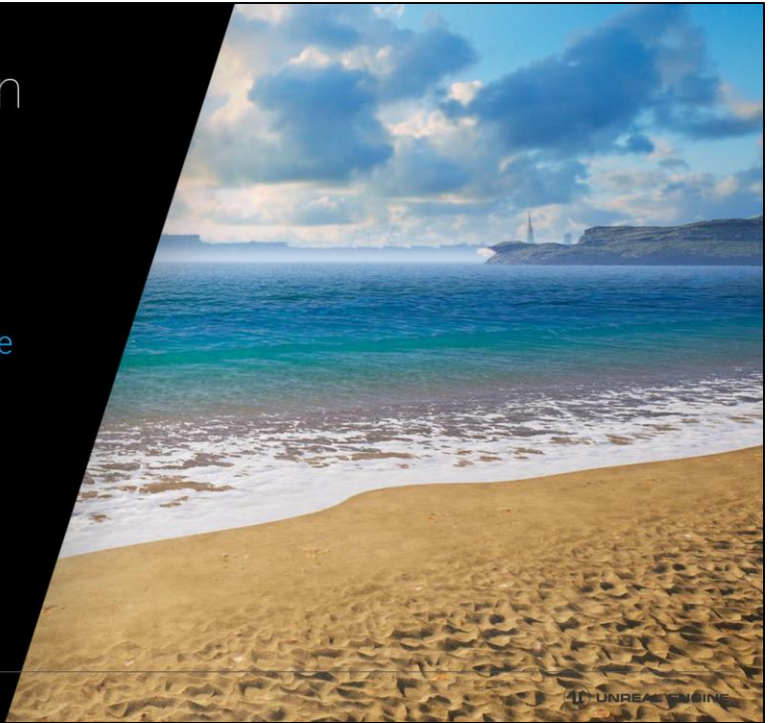
The basic idea for analytical simulation is that you pre-calculate the desired behavior into some function and then simply look up into that function to get the desired result later. This is in contrast to 'regular simulation' where you will actually be applying some behavior every frame and re-updating and writing to some buffer.

Most standard particle systems (ie, both Cascade and Niagara in ue4) will actually be writing to some buffer every frame to update positions, even if you are doing something simple like a just gravity force.

Analytical Simulation

Beneficial Properties

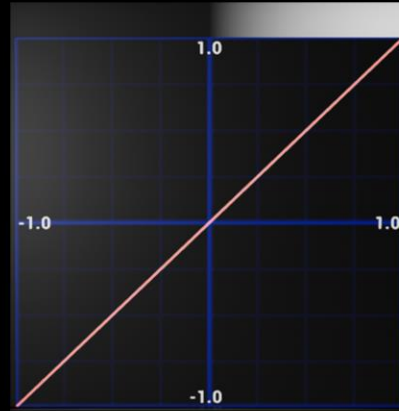
- Synchronized Effects
- Memory cost **easily trackable**
- Avoids Ticking cost
- Deterministic



Backing Up... What's a Function?????

A function **modifies** some **input**

- $y = f(x)$
- Simplest Function: **$y = x$**
- A fancy line!



This is just to first show the simplest possible function definition of $y=x$, which is a linear line with a slope of 1.

Breaking it down

The **f(x)** represents the function. Can be anything.

x represents your **Input**. Usually includes **Time** (plus other stuff).

The **y** represents output.

Naming it **y** indicates plotting results on the **Y axis**.

This slide may be a bit boring, and it is understandable to want to skim past it. It is not necessarily critical to applying the concepts presented, but I do feel it helps to establish exactly what is the connection between the math we write as functions and shaders and standard mathematical formulas. This is because you will often run into functions expressed this way when reading white papers or researching a concept that is expressed outside of the games industry.

Building practical examples

$y = \text{Sin}(x)$ is a simple function

Use **Time** for **x** to get an animated Sin Wave

This is the foundation for many effects

Let's build something **practical** from here!



We first look at a function less boring than a line. The most obvious one is the sine wave and it actually forms the basis for a whole plethora of effects including but not limited to: wind, water, earthquake shakes, tension on wires, flapping wings, bouncing balls etc.

A Bouncing Ball

By adding one operation to our function we can modify it to do something else.

Add the **Absolute Value** operator.

$$Y = \text{Abs} (\text{Sin} (x))$$



Just by adding the absolute value, we can force everything to be positive. This is useful for creating a series of pinched gradients that can be used for things like bouncing balls.

A Bouncing Ball

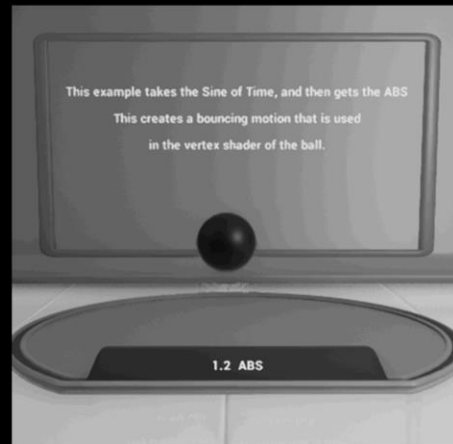
Now let's use the function in a shader

Convert: $y = \text{Abs}(\text{Sin}(x))$

To a Vertex Shader

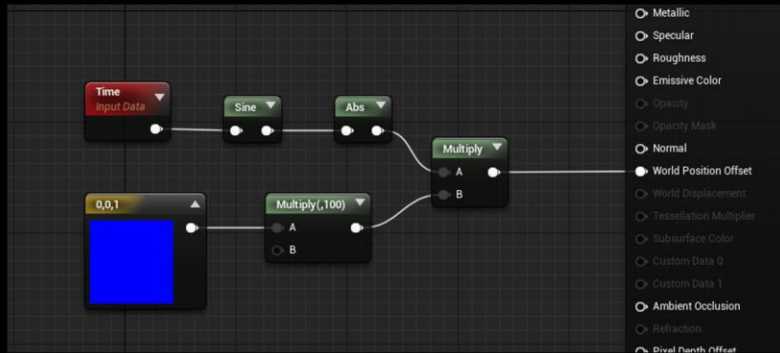
Using **Time** for x

Using the Output y as our **vertex offset**



Ok, enough “Math”, its material time!

What does that look like in UE4?



In UE4 creating vertex shaders is thankfully easy.
Just take a regular new material, and add instructions to the graph and hook them to World Position Offset.

What about user control?

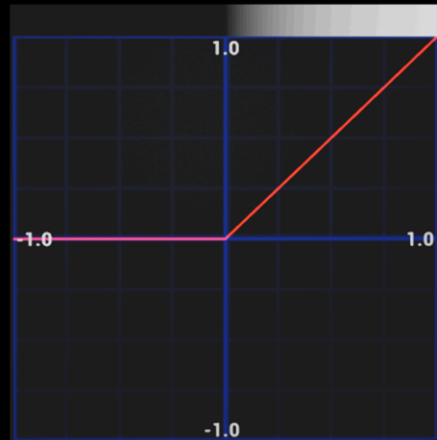
Control timing by **Subtracting Start Time**

Clamp Result to > 0

Only need to set one value once

Can modify the result from there

A Power adjustment is shown



Since time normally continues from 0 until as long as the game is running, we might need to prevent animation from happening until some interaction occurs.

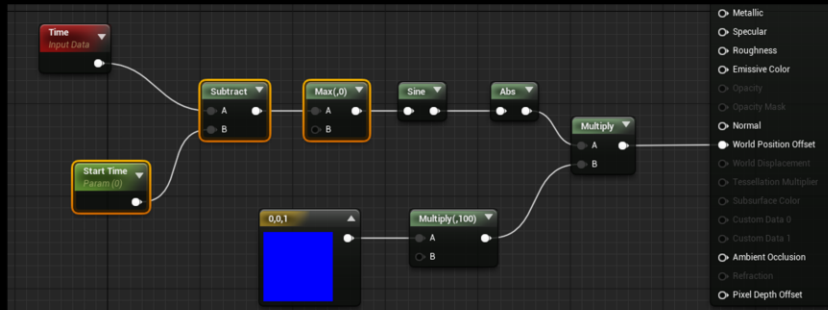
An easy way to do that is pass in a Scalar Parameter called “Start Time” and then clamping the result to positive values.

This is like having a linear, local time for an object and the value for the start time of the interaction only has to be set ONCE.

Then that result can be modified and turned into the desired modelled behavior.

The Material Changes

We add a Subtract with **Start Time** scalar



We add a **Max(x, 0)** to clamp above 0.

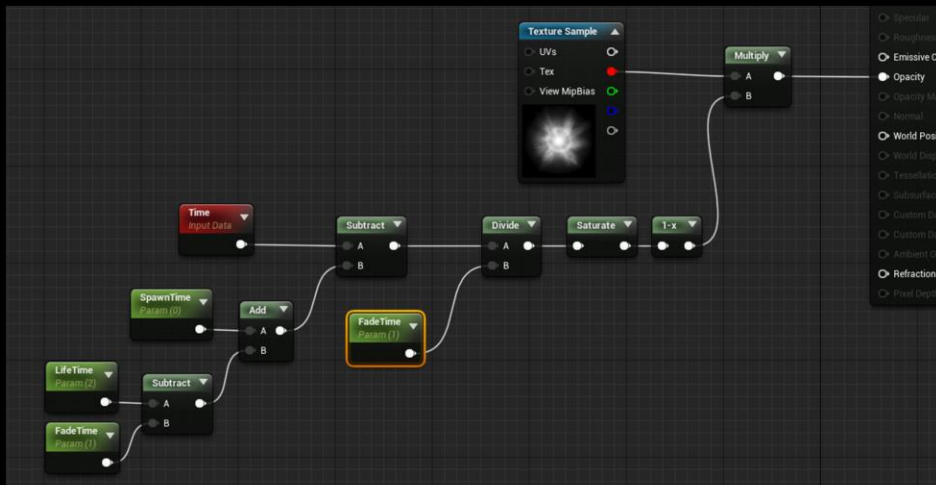
Fading Out Effects

Useful for Fading Decals and Mesh effects



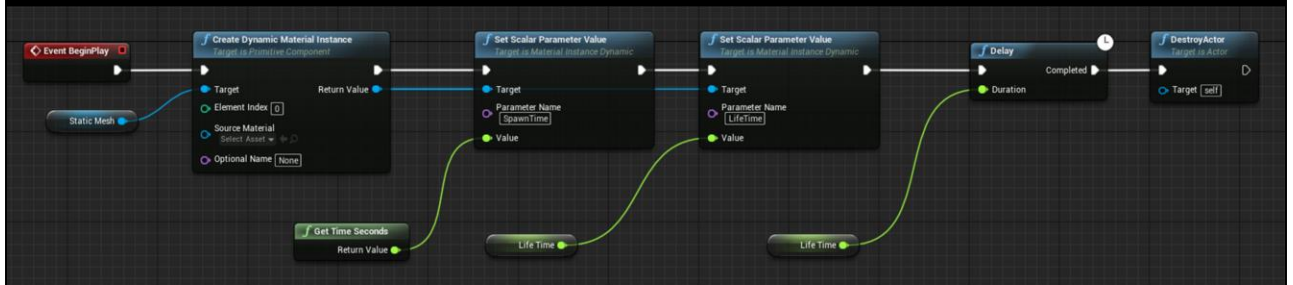
Using the “Time - StartTime” method, custom effects can be faded out automatically in the shader without ticking the material. It is true that UE4 decals provide this functionality, but knowing how to do this can be useful for many other reasons, such as when you need to fade things that are not standard decals.

Fade Out Logic: Material



This shows a method where the material will automatically fade out after the specified lifetime. Note that `FadeTime` is subtracted from `Lifetime` here. That means that the `Lifetime` can be changed by the blueprint and the effect will still be destroyed at the right time, accounting for the fade time.

Fade Out Logic: Blueprint



Create **MID** on spawn, set **SpawnTime** and **LifeTime**

Then **Destroy Actor** after LifeTime Delay

This shows the full blueprint logic for the decals in the fade out example. On spawn, a Material Instance Dynamic is created. The SpawnTime and LifeTime values are set one time and a delay timer is started. At the end of the delay, the actor deletes itself.

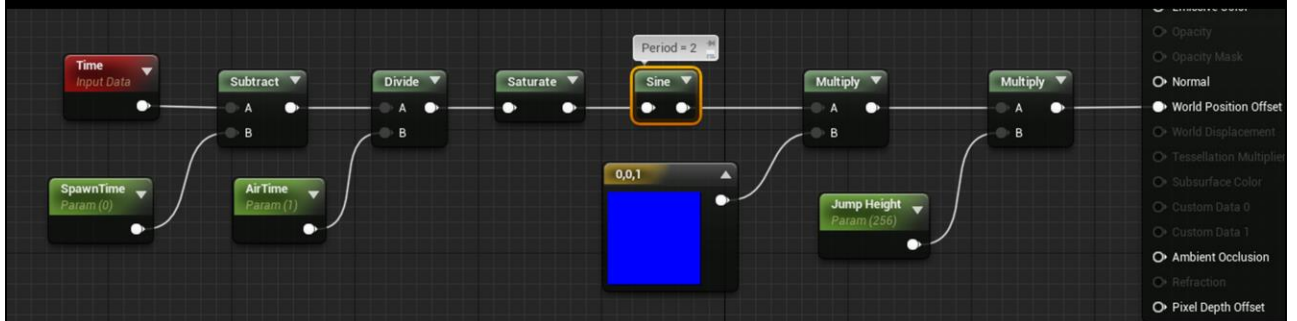
A Timed Jump Shader



Same Timed Fade can be used for motion

This is replacing the previous fade out timed example with vertex shader behavior. Instead of a linear fade, a sine wave is used and time is clamped so that the sine starts and stops at the desired moments.

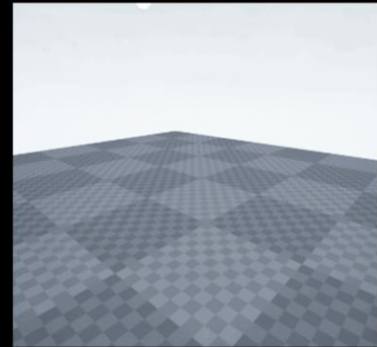
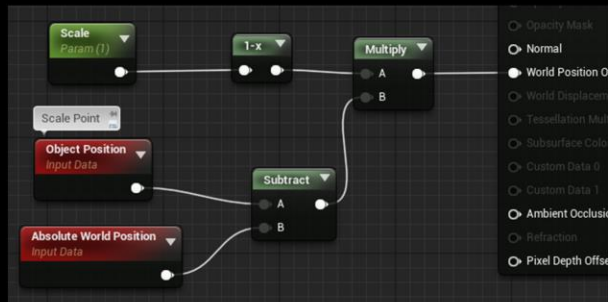
A Timed Jump Shader



Same Timed Fade can be used for motion

This is pretty much the same as previous example with the bouncing ball. The main difference is that we added the Start Time offset and also did a **Divide** to control the 'Air Time'. Notice that the **Period** of the **Sine** node was changed to **2**. This is because with a period of 1, sine would go both positive and negative. Since we use a saturate node, that clamps the input to 0-1. Setting the period to 2 means the time never gets past the 'halfway point' of the sine period, which effectively stops it when it hits the ground again. It is also worth pointing out that in UE4, the default period of Sine is 1, but in strict mathematical terms, the period is 2π . This is just something to be aware of, and when matching something from a whitepaper, you often want to set the period to 2π so your period will match the correct math default.

Scaling by Life

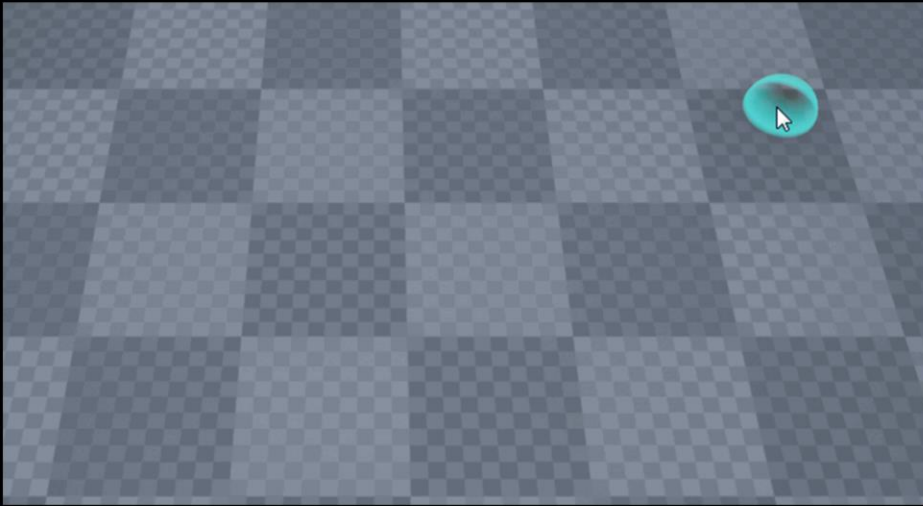


(Center - WorldPosition) **collapses** verts to **center** when used as Vertex Offset

Vertex shaders can scale objects from any point. One way to think about this is by thinking about what WorldPosition is. It is the position of any vert in the world. So if you pass in negative worldposition as a vertex offset, it effectively collapses all verts to the origin (0,0,0). Then if you add in a 'center point', it will instead collapse to that center point instead of the origin. Then this offset can be scaled to smoothly blend the collapse effect on and off.

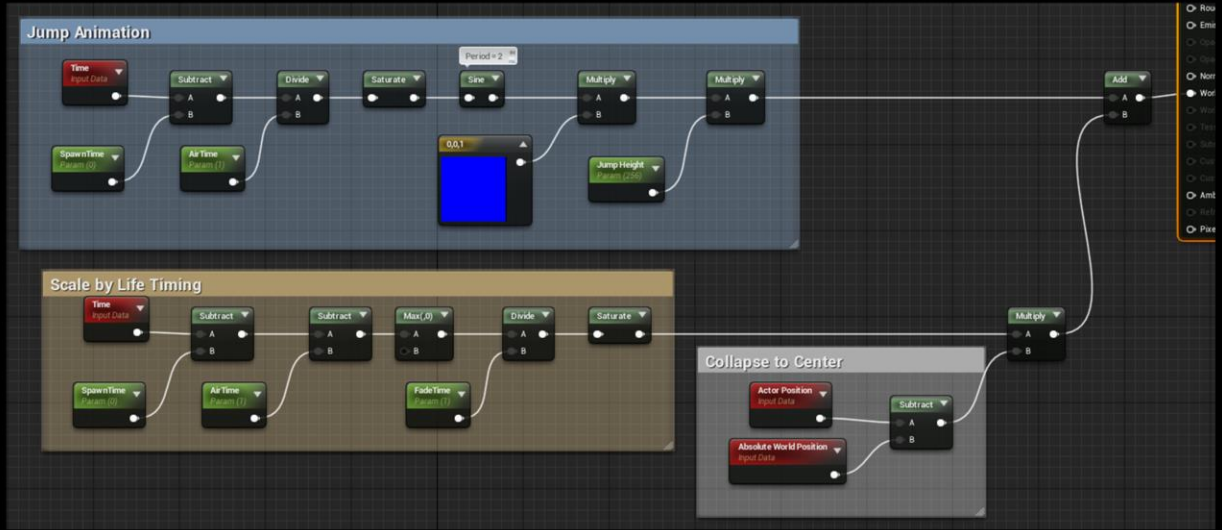
Note that the collapse is 'full' when passed in without a multiplier (ie, a multiplier of 1.0). To rename our multiplier as "Scale" rather than "Collapse", we simply perform a 1-x on the value and can now think of it as scale.

Jumping with Scaling by Life



It is often useful to combine more than one function to get the desired effect. Here we show the previous timed jump with a timed Scale by Life effect.

Combining Two Pieces



We combine the two pieces from before: Jumping and Scaling. Each operation has its own Time Offset and then they are both added together. Note that the bottom 'Scale' area subtracts 'AirTime' instead of Dividing, since it is designed to wait until the jump animation is complete. Then the saturate nodes are just a clamp between 0 and 1.

A Looping Effect

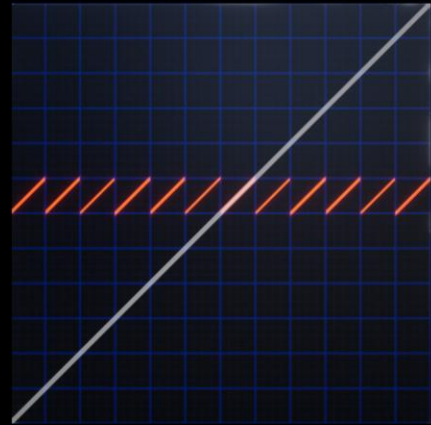
Frac: The basis of a loop

Returns the **fractional part** of number

Repeats **0-1**

Think of each 0-1 repeat as one **Lifetime**

Offset time w/ random values encoded in mesh



Using the frac operation usually forms the basis for custom looping effects in vertex animation. First you divide Time by the desired 'Lifetime' and then get the frac. Then you have a repeating 0-1 lifetime and can perform all of the custom animation effects previously listed and they will repeat. When multiple particles (or static mesh quads) are involved, the time can be randomized by providing a time offset, usually in the form of a pre-stored random vertex color. The Pivot Painter script by Jon Lindquist provides one great way to seed the data.

Manual Debug Time

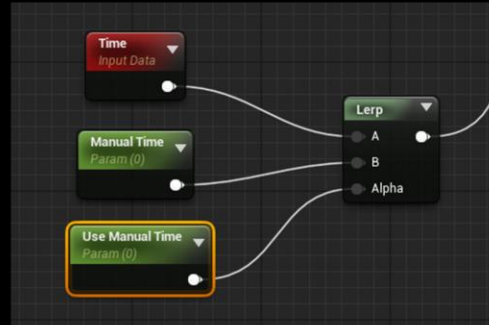
Often useful to be able to **debug time**

Lerp between Time and **Manual Time**

Then set the Alpha to either 0 and 1

Values can be **MPC** for to sync' materials

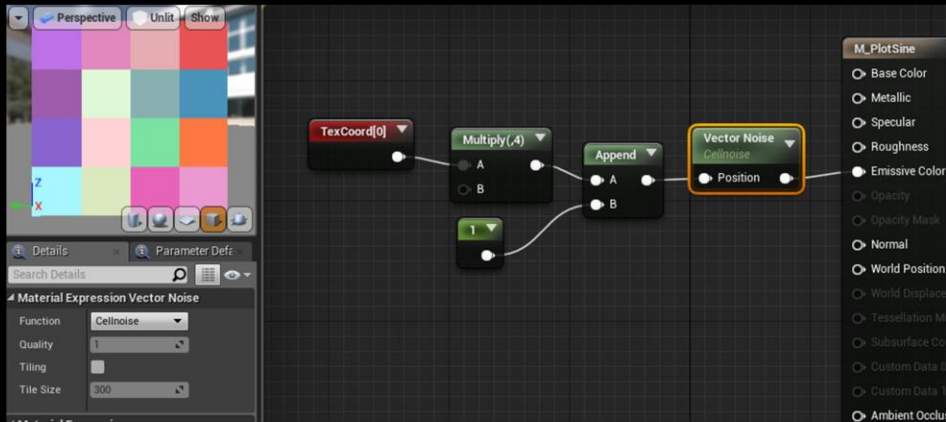
Allows quickly toggling looping vs manual scrubbing of value



This is a quick operation that can be done to let you dynamically toggle between using the actual Time node and a user specified value. This can be really useful when you need to inspect your function at a specific time more carefully. It is often helpful to use Material Parameter Collections for these values, then you can have control over full time dilation across many shaders.

Randomization

Vector Noise: The most useful UE4 material randomizer



UE4 has a material node called Vector Noise which has a few modes. The one I want to draw attention to right now is the 'Cellnoise' mode. This node takes a V3 input. For each unit cell (ie, each integer sized input), it returns a random V3 output between 0-1. Note in the input, I used the UV's multiplied times 4 as the input (and appended a 1 to make it 3d). Thus, it created a 4x4 grid of random colors to use. This is the perfect thing for taking things like a linear index and making them random for a variety of effects.

Encoding for Randomization

Typically use **Vertex Colors** which are **8-bit**

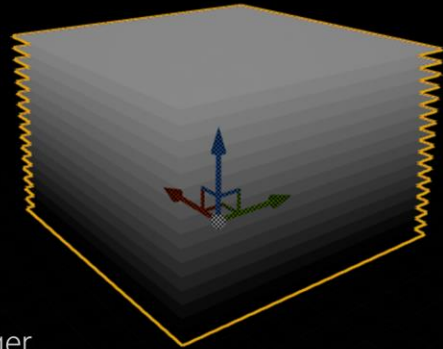
This means values **0-255** available

Values will appear normalized 0-1

Write values using 1/255 increments

On material, multiply color by 255 to restore integer

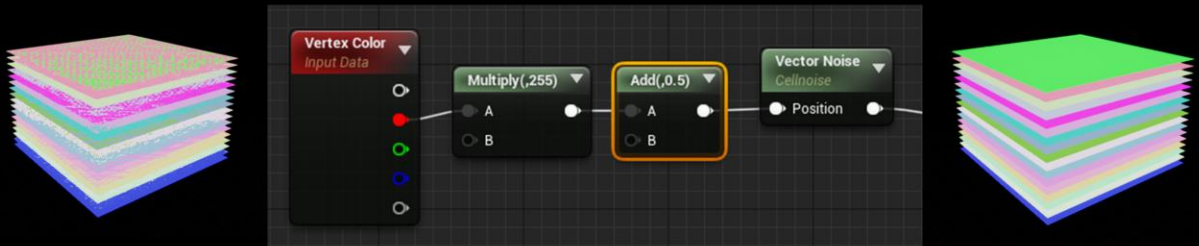
Shown: one mesh with 16 quads indexed 0-16 in vertex color



Vertex colors are encoded as 8bit so you have values 0-255 available. Most dcc apps will normalize the input though, so painting a value of 1.0 is actually 'really' painting the 8bit value of 255. It is actually more useful mathematically for randomization to think in the 8bit values for a variety of reasons (ie, a more advanced concept but this allows quantization, ie using 2 vertex color channels to store 65,536 precise indices). Using this method, using a single channel vertex color can get you full RGB randomization which saves the other channels for other uses.

Decoding for Randomization

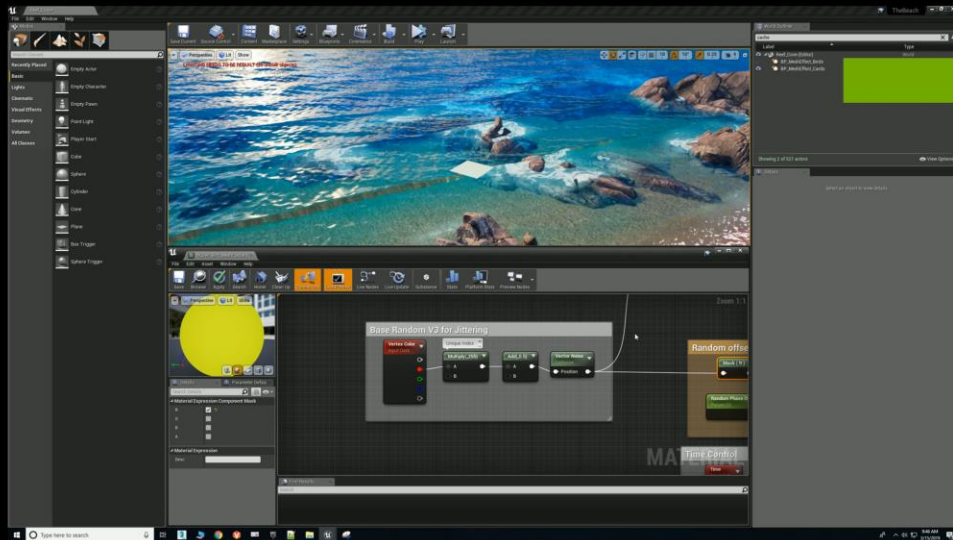
With Cellnoise, we can get **precision artifacts** sampling right at integers



Fixed by doing $\text{Color} * 255 + 0.5$

If you just take the $\text{VertexColor} * 255$, the output will be the exact integer index of each quad. Since cellnoise is defined as a different random value per Unit input, it transitions right on the integer values. So an exact integer value is expected to cause a precision problem. This is fixed by just adding 0.5 after getting the integer value, in effect sampling each cell in the center instead of the edge.

Building the random loop



If you just take the $\text{VertexColor} * 255$, the output will be the exact integer index of each quad. Since cellnoise is defined as a different random value per Unit input, it transitions right on the integer values. So an exact integer value is expected to cause a precision problem. This is fixed by just adding 0.5 after getting the integer value, in effect sampling each cell in the center instead of the edge.

A Fortnite Example

The heart FX in emote **True Heart** make use of these methods



Effect by Scott Kennedy

This emote and effect were originally authored for Paragon but were later brought to Fortnite. The heart at the end of the effect demonstrates very similar methods that this talk describes.

True Heart Effect

Heart is based on a **static mesh**

This is the **wireframe** for the mesh

Lots of tiny quads, **random vertex colors**

The **vertex shader** expands into sprites

Cascade controlled timing for scale for shader



The core heart in the effect was a single static mesh that has many small quads premade. This was done in this case because there was no built in way to have particles spawn in a specific artist controlled shape using cascade.

Each quad got a random vertex color value used to offset size and motion.

The vertex normals were encoded to be the direction to the heart shape spline.

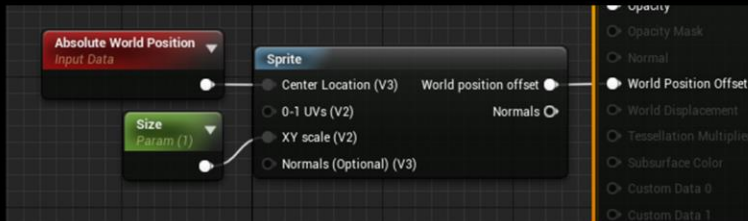
These normals were handled in a clever way by Scott Kennedy.

By spawning the particles from the heart shape in Maya, the quads were oriented along their spawning normal.

Because the quads are made very tiny and then later expanded in the vertex shader, the fact that they are oriented strangely in the original mesh does not affect anything visible.

Sprite Expanding Shader

WorldPosition works as a **center** for small quads

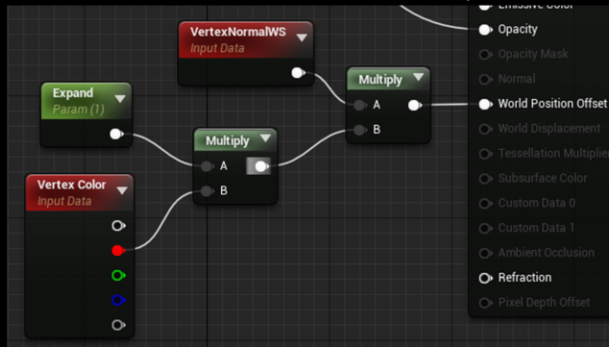


The material function 'Sprite' can be used to turn any polygon into a camera facing sprite.

Typically something like 'object position' or 'particle position' is used to specify the 'Center Location', but it also works in a pinch to use WorldPosition, assuming the quads are very small to start with. It is not shown here, but the Size was also modulated in the end by multiplying with the random vertex color value. Shown in a further slide.

Positional Jittering

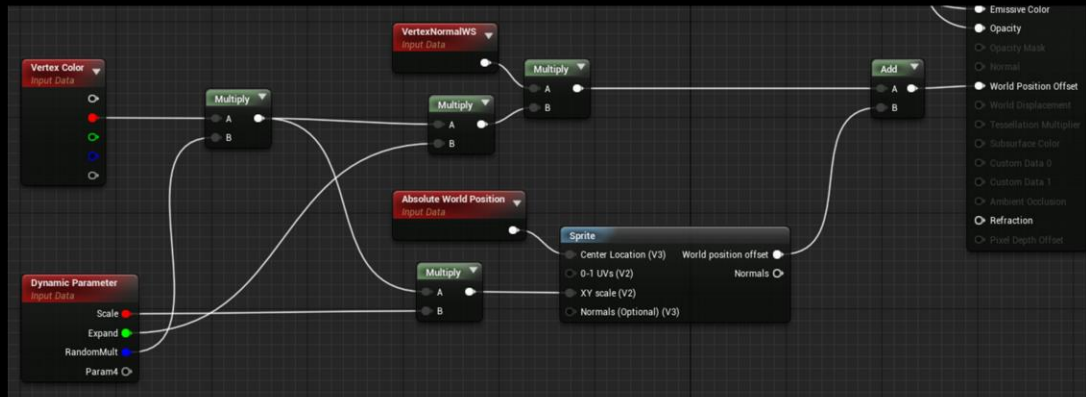
Normals Encode Direction from Shape



Since the vertex normals stored the direction to the original heart shape, the 'VertexNormalWS' node could be used to push the sprites along this direction. This push is multiplied by the 'Vertex Color' node containing the encoded random values. This causes some sprites to move more than other, effectively jittering the amount of offset. This creates interesting noise in the motion.

Combining Offsets

Add both offsets. Replace scalars with **Dynamic Parameter**.



The random scaling and position offset effects are chained together by Adding them. Also, in order to synchronize the timing of the heart with the rest of the effect which was made in cascade, the Scale and Expand parameters were replaced with a 'Dynamic Parameter' node, allowing curves from cascade (or Niagara if this was remade today), to drive the values.

This demonstrates that sometimes the reasons for building an effect like this can have to do with technical limitations or just giving a much greater level of artist control.

Llama Fireworks

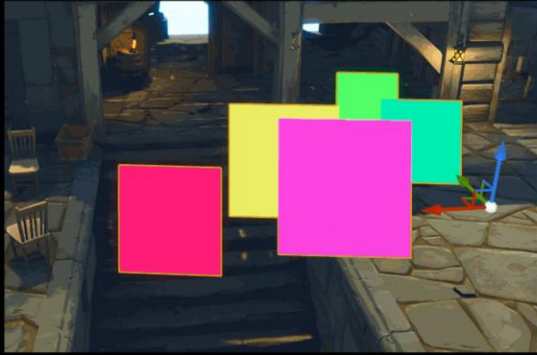
The **Llama Fireworks** effect by [Fredrik Seehuusen](#) uses similar methods



A very similar method is used here to create the expanding Llama firework effect. The shader is very similar to the previous example and uses cascade to control a Dynamic Parameter to provide timing control.

Environmental Effects

Often used for simple environment effects



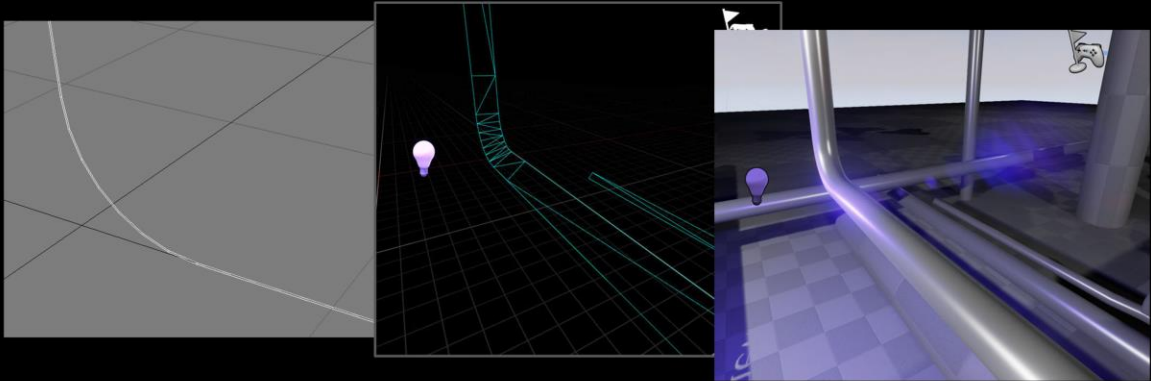
Used when there's **0 budget** for game thread cost

We have often used simple static meshes for persistent environmental effects like fog haze cards or dust modes.

The shader setup is very similar to the previous heart example. Random vertex colors will be encoded and then used to apply some sort of jittered offset to size and/or velocity.

Spline Thicken

Like **Sprites**, but projected along **Splines** instead of **Points**



Spline Thicken was originally conceived of by Jordan Walker for the UE3 Samaritan Demo in R&D. I later expanded on it and added accurate world normal support. This method is frequently used by Epic FX and Environment artists for effects where some kind of cheap spline or cylinder is needed.

Spline Thicken Example

Used for Environment assets and FX



Used for FN **Grappler** and **Tracers**



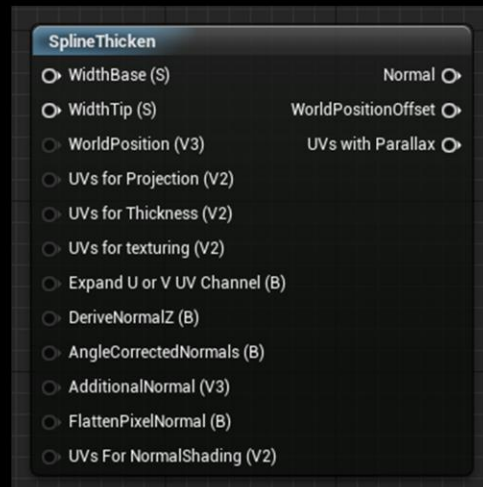
Spline Thicken was originally conceived of by Jordan Walker for the UE3 Samaritan Demo in R&D. I later expanded on it and added accurate world normal support. This method is frequently used by Epic FX and Environment artists for effects where some kind of cheap spline or cylinder is needed.

Spline Thicken Material Function

Supply parameters for **Thickness**

Options for which **UV axis** to project along

Options for applying texture **normal maps**



Spline Thicken was originally conceived of by Jordan Walker for the UE3 Samaritan Demo in R&D. I later expanded on it and added accurate world normal support. This method is frequently used by Epic FX and Environment artists for effects where some kind of cheap spline or cylinder is needed.

The Coalition: Swift Particle System



Referred to as **Swift Particle System**, written by **James Sharpe**

From discussions with Colin Penty at the Coalition, many of the in game effects for Gears5 are being authored using vertex shaders, primarily the environmental effects but also some weapons and gameplay effects.

They were having problems with the CPU cost of particle simulation. For GPU particles, even spawning was a problem since there was CPU logic involved. Switching over to this method freed them up to do more.

What about Spatial info?

I.E., reacting based on proximity, or looping period



Water FX

Gerstner Waves are a common method for modelling oceans

Based on the Sum of different length **Sine Waves**

$$z = \text{Amplitude} * \cos(\text{WaveDir} \cdot \text{Pos} - \text{Frequency} * \text{Time})$$



Amplitude = Wave Height

WaveDir = Direction of Wave Travel. It has magnitude $k = 2\pi / \text{Wavelength}$

Pos = Starting position of vertex to evaluate

Frequency = Frequency of wave

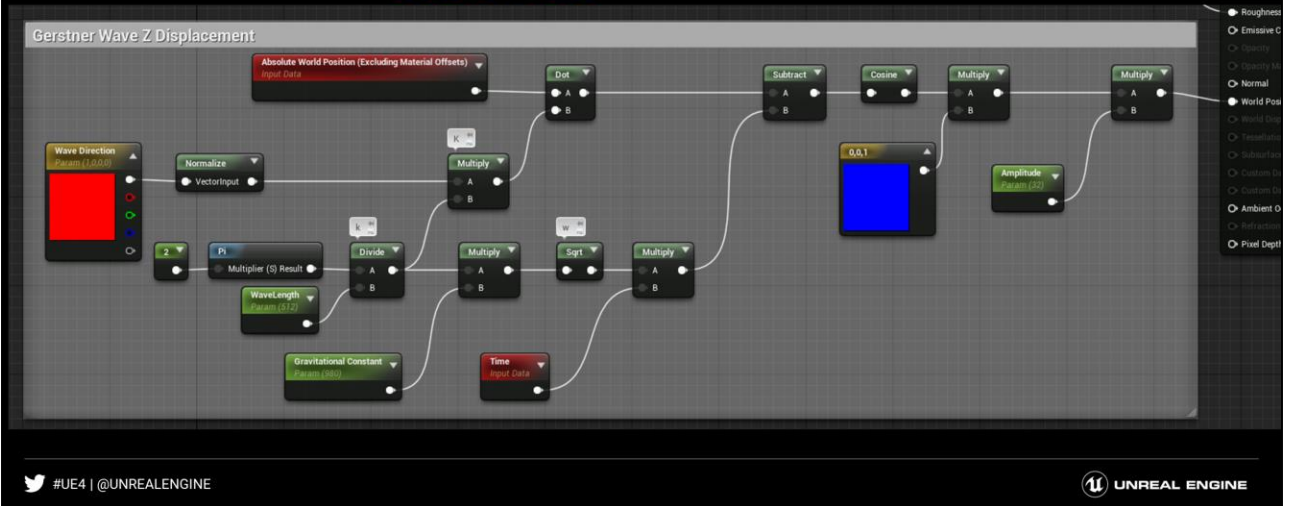
Time = Our friend Time again

Note, not shown is the formula to create the steep crests. This is a separate function that uses Sine instead of Cosine and displaces vertices sideways.

Gerstner waves were largely popularized by Jerry Tessendorf's paper Simulating Ocean Water

Stop with the Math Already

UE4 Vertex Shader of a Gerstner Wave



Amplitude = Wave Height

WaveDir = Direction of Wave Travel

Pos = Starting position of vertex to evaluate

Wavelength = Frequency of wave

Time = Our friend Time again

Phase = Offset, basically same as "Start Time" in previous examples

Note, not shown is the formula to create the steep crests. This is a separate function that uses Sine instead of Cosine and displaces vertices sideways.

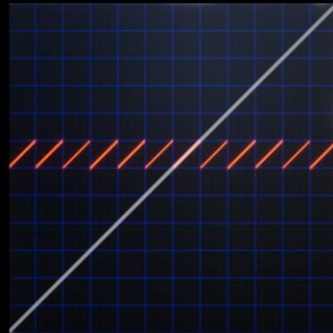
Synchronized, Tickless Splashes

Sample the **Largest Wave(s)** using a Material Function

Roll a function that **resets** every time the **wave passes** some point

Our old buddy, frac

Just divide time by **wave frequency**



Since we know the frequency of the wave, we can simply use that to drive the Frac that repeats with each passing wave.

Applying At Scale

Match the frac function with **wave period**,

Modify it like we did with our initial **Power tweak**

We can apply a **gravity arc** using **sine**

Make particles **jump up** with waves



Once we extract the period of the wave using the frac function, we can then use that like linear repeating time that begins with each passing wave. It is also necessary to offset the phase at this point by determining how far between wave crests the current point is, using a time value of 0.

Synchronized Ocean



#UE4 | @UNREALENGINE

UNREAL ENGINE

This video demonstrates many of the techniques built up in this talk. Material Parameter Collections (MPC) are used to easily share the same global parameters between the various materials involved so the parameters only need to be set once. This way, all the timing and things like wavelength match between the ocean, the splashes and the rock foam.

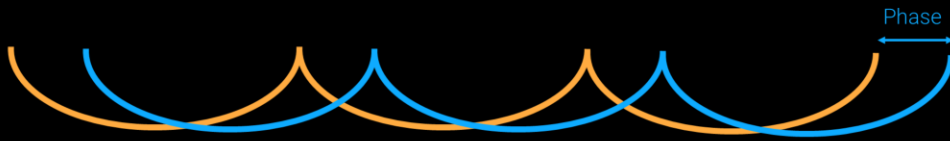
Distance Fields in the mix

Distance Fields tell you how far you are from the **closest** surface.

We can use them in a **variety** of ways to aid Analytical Simulation.

Let's expand on the Gerstner Wave formula:

$$z = \text{Amplitude} * \cos(\text{WaveDir} \cdot \text{Pos} - \text{Frequency} * \text{Time} + \text{Phase})$$



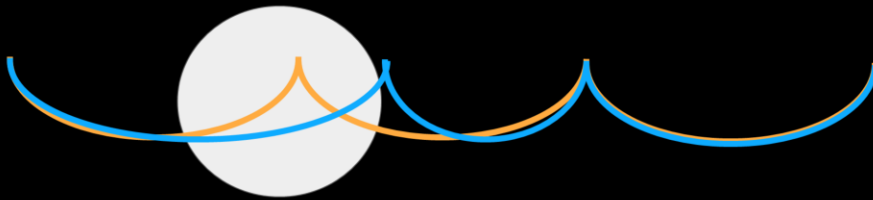
Notice that we now added a Phase parameter as a time offset. In the typical implementation, the phase is just a random scalar constant for each wave, and only serves to offset the starting values. But this value does not HAVE to be a constant. It can have variation, as long as it is minor.

DF Phase Offset

We can use Distance Fields as a **PhaseOffset**

Can be used to **model** the effect of rocks in the ocean

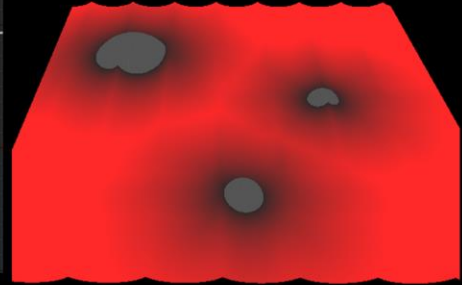
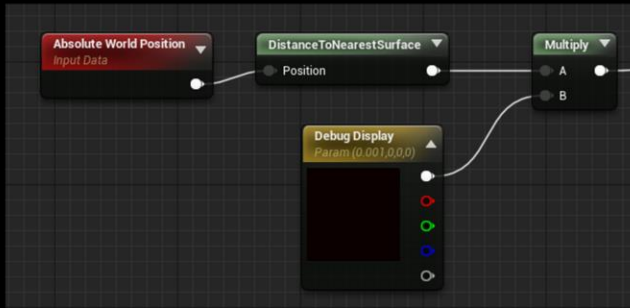
Adds some **resistance** to the water, delaying waves slightly.



Since the DF varies spatially, it can be used to alter the phase spatially, based on the environment.

Distance Fields Debug

UE4 has **Distance to Nearest Surface** node

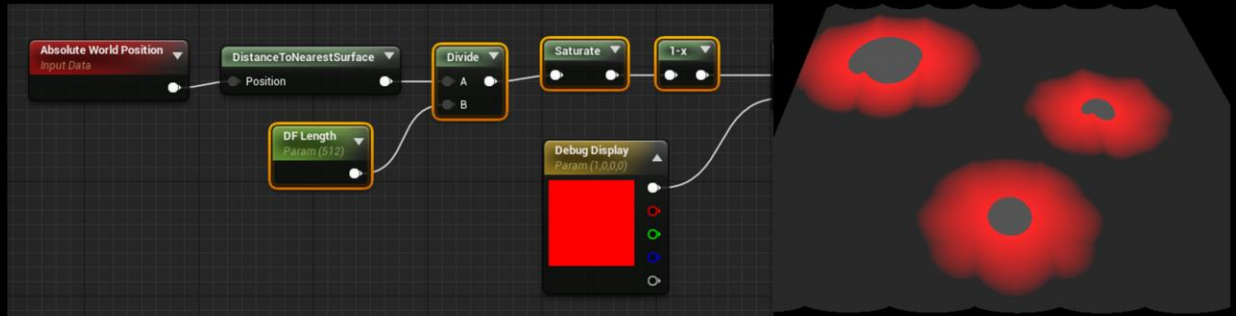


In **World Units**. Shown here by multiplying by small red value.

For UE4 projects with distance fields enabled, the material node DistanceToNearestSurface returns the Global Distance Field value, which is a combination of all mesh distance fields in one volume texture surrounding the camera.

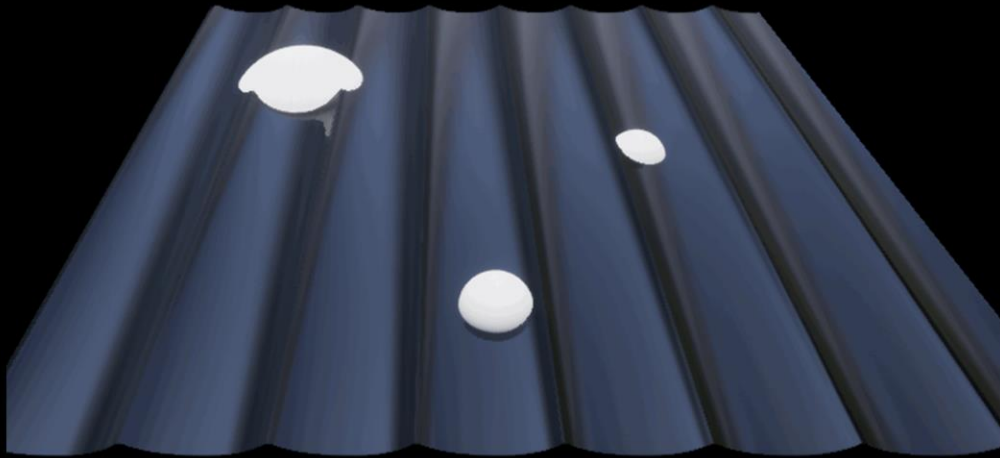
Extracting a Linear Range

First we must **remap the range** of the distance field



This slide shows how to extract an linear mask from 0 to 1 of the specified world unit size. Saturate is just a clamp between 0 and 1 that costs no instructions. '1-x' inverts the result so that it is 1 at the nearest surface and 0 at the defined width.

Phase Offset Example



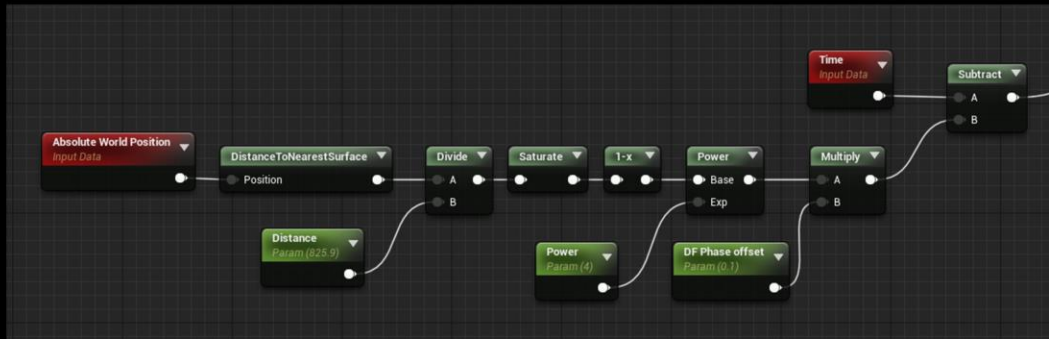
#UE4 | @UNREALENGINE

UNREAL ENGINE

This shows the effect of the phase offset using a defined width from the obstacles. The phase offset is modified during the animation, showing how the result can do some crazy things if it is not kept sensible. But maybe somebody out there will find a way to use such an exaggerated effect to artistic purpose.

Phase Offset Graph

Use result to **subtract a small value** from **Time**



Notice that emphasis is now placed on the Phase parameter. Previously this was set to 0.

In the typical implementation, the phase is just a random scalar constant for each wave, and only serves to offset the starting values.

But this value does not HAVE to be a constant. It can have variation, as long as it is minor.

Distance Field Gradient Uses

By combining **Gradient** with **Wave Direction** we can enhance modelling



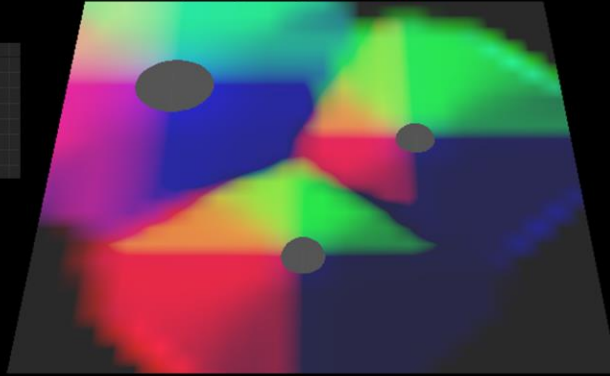
Notice that emphasis is now placed on the Phase parameter.
Previously this was set to 0.

In the typical implementation, the phase is just a random scalar constant for each wave, and only serves to offset the starting values.

But this value does not HAVE to be a constant. It can have variation, as long as it is minor.

Distance Field Gradient Debug

DF Gradients are like Normal Maps, giving Direction to nearest surface



They tell how fast values change in each direction

Distance Field gradients give the slope of the change in distance in each direction. It is common to normalize them to make the value unit vector length, similar to normal maps.

Gradient Dot Product

Dot Product between **Gradient** and another vector can be computed



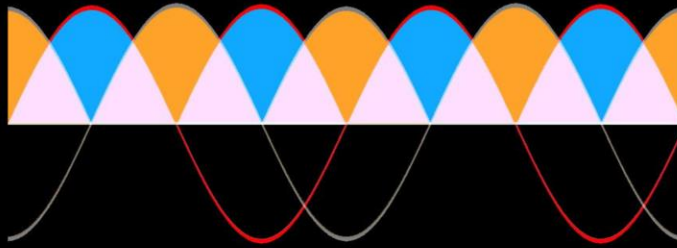
This shows where DF Gradient **aligns** with the second vector

The dot product between the DF gradient and another vector will show where the DF gradient aligns with that vector. If both vectors are normalized (ie, unit vector length), then the output will be 1 where the vectors align, -1 where they are opposing, and 0 where they are perpendicular. Using this knowledge, you can determine which side of an obstacle needs forward buildup foam vs trailing turbulent foam.

How about Flowmaps?

Flowmaps: a popular method to model fluid flow

An artistic effect. Seeks to **distill the essence** of fluid flow.



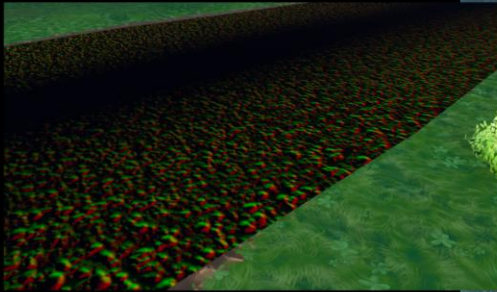
Two textures **advect** and **fade** with **offset phase**

This is just an illustration for how flowmaps work. There are two textures that each get advected by some texture containing velocity or direction. Since if you advect or distort the UVs of a texture too much things look bad, flowmaps reset every period, and fade to another texture with an offset phase. When one texture has an alpha of 1, the other has an alpha of 0. These are actually using sine and cosine waves, with the absolute value. **Note: UE4's flowmap function actually uses a triangle wave**, but the difference is minimal.

How about Flowmaps?

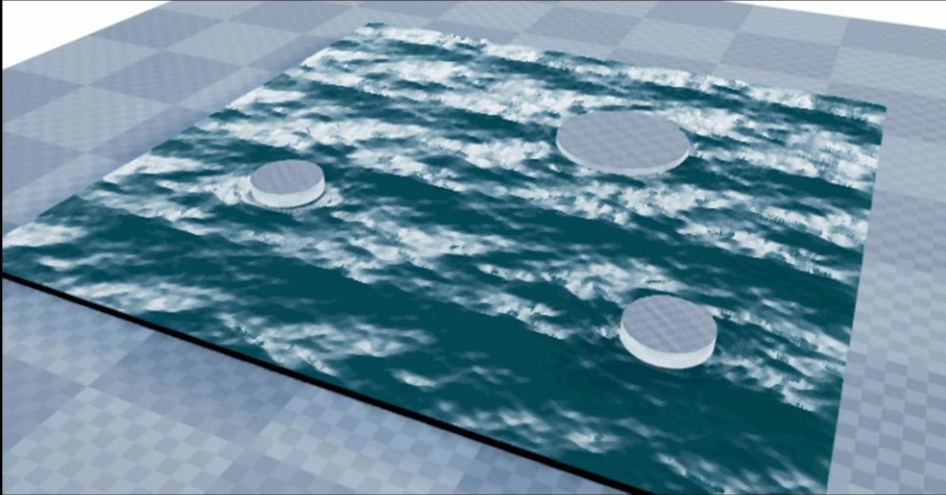
Distance Fields and normal maps can enhance flowmaps

Use the Gradient to alter flow



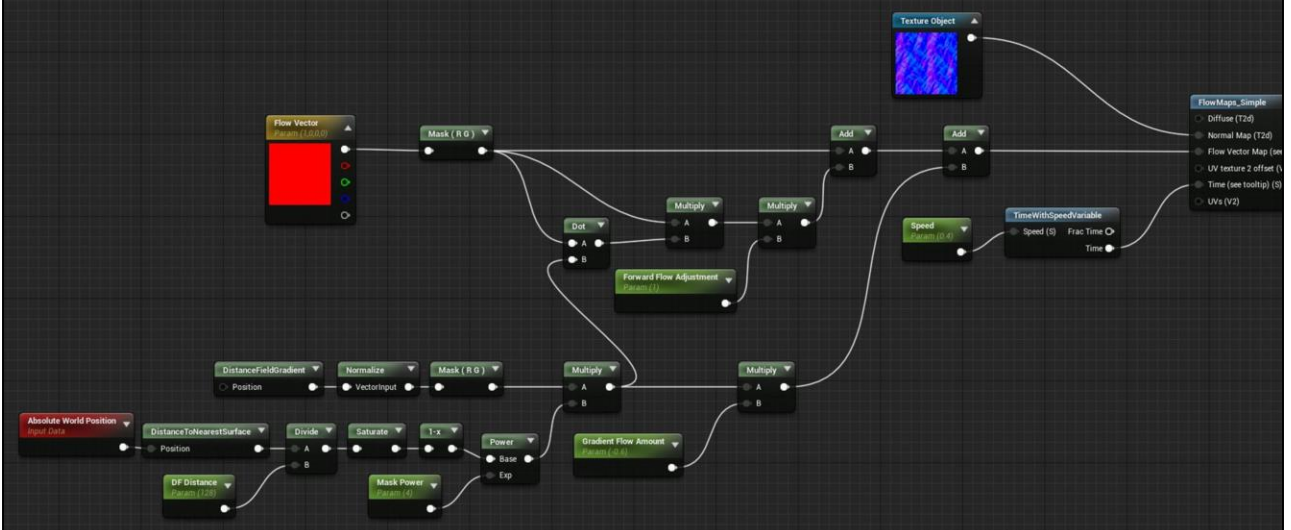
This example shows a fortnite river effect. The water flowmaps near the edge of the shore use the rock normalmaps to make it look like the water parts around the pebbles. Since this pulls in reflection details, it also creates the illusion of small whitewater, which is not actually present. Note that the normal map was pre-blurred to help dilate the normals away from the rocks themselves. For other content it might not be necessary.

DF Flowmap modifications



This example shows the effect of altering flowmap vectors using a distance field graph. The parameter for the distance of the effect is adjusted during the animation. Note that when the effect is too big and the magnitude is not adjusted properly, stretching can show up. This is the kind of thing that requires tweaking to taste and can be very content dependent.

DF Flowmap Graph



This is the graph to modify a flowmap vector using a distance field. The approach when using a normal map (like the fortnite example) is actually pretty similar, you just have to transform the normal map from tangent to world space, and do the same with the flow vector and use world coordinates (or the opposite if using tangent vectors, transform the normal map into the 'flow space').

Wrapping up

Analytical Simulation is a great method to consider

Great for environmental FX and synchronization

Powerful cost savings


A useful way to begin thinking about effects

This example shows a fortnite river effect. The water flowmaps near the edge of the shore use the rock normalmaps to make it look like the water parts around the pebbles. Since this pulls in reflection details, it also creates the illusion of small whitewater, which is not actually present. Note that the normal map was pre-blurred to help dilate the normals away from the rocks themselves. For other content it might not be necessary.



THANK YOU FOR LISTENING

Screenshot from VR Editor Demo GDC 2017. Environment and FX by Ryan Brooks. Sky material by Peter Sumnasant.

 #UE4 | @UNREALENGINE

 UNREAL ENGINE

References

Simulating Ocean Water - Jerry Tessendorf - 1999

<https://www.semanticscholar.org/paper/Simulating-Ocean-Water-Tessendorf/70dad4fed48bfbb5b1d417e141ef68d1568e4911>

Flow Visualization Using moving Textures - Nelson Max & Barry Becker - 1995

<http://www.heathershrewsbury.com/dreu2010/wp-content/uploads/2010/07/FlowVisualizationUsingMovingTextures.pdf>

GPU Gems Chapter 1 - Mark Finch - 2004

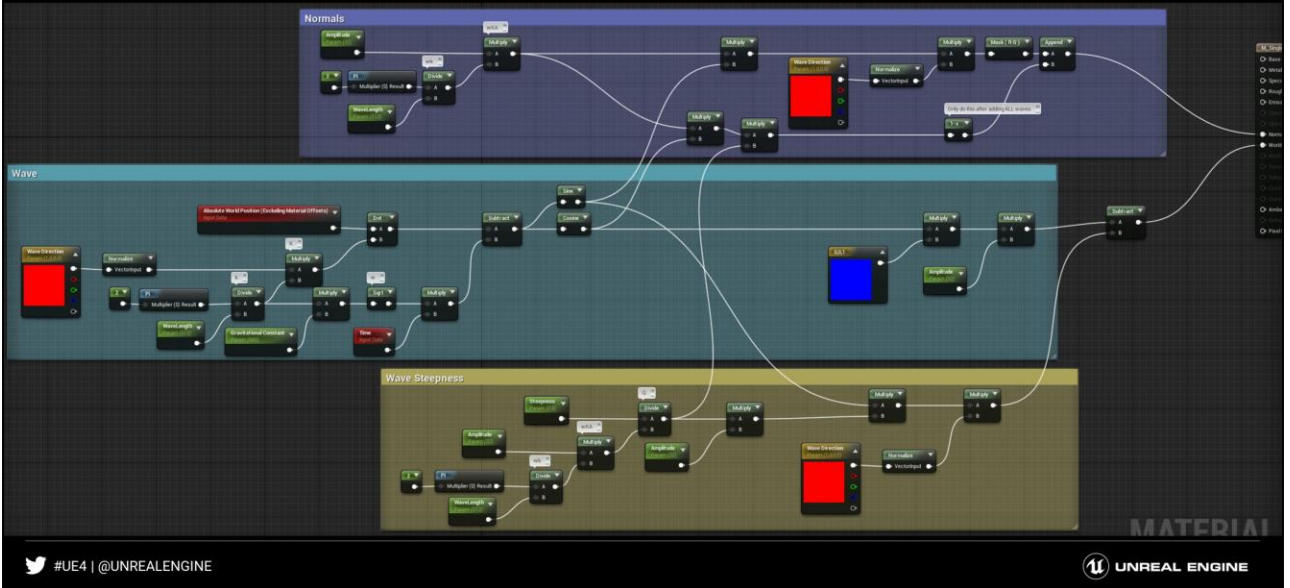
https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch01.html

Notice that emphasis is now placed on the Phase parameter.
Previously this was set to 0.

In the typical implementation, the phase is just a random scalar constant for each wave, and only serves to offset the starting values.

But this value does not HAVE to be a constant. It can have variation, as long as it is minor.

Bonus Slide 1: Full Gerstner Wave



This is the full graph for a single gerstner wave. I know its probably a bit hard to read, I tried a few times to cram it into a single graph. For a better understanding please refer to the referenced GPU gems chapter.

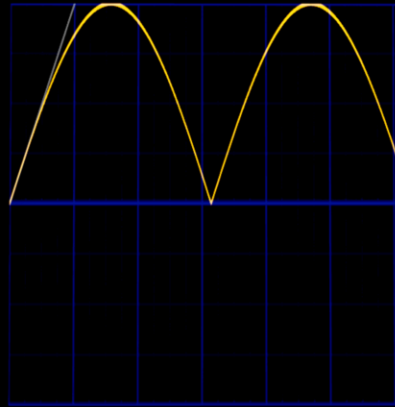
Bonus Slide 2: A Lagged Wave

Lagged waves are **incredibly** useful for many things

```
# Code for a Lagged Wave
BaseWave = abs(sin(Time));

# Build the Tail
WaveTail = frac(2* ((Time / (2*PI))) + 0.5);
WaveTail = (WaveTail / (LagValue * 2.01));
WaveTail = 2 * PI * (WaveTail + 0.25);

# Combine the Tail and the Base Wave
LaggedWave = max(BaseWave, sin(WaveTail));
```



Notice that emphasis is now placed on the Phase parameter. Previously this was set to 0.

In the typical implementation, the phase is just a random scalar constant for each wave, and only serves to offset the starting values.

But this value does not HAVE to be a constant. It can have variation, as long as it is minor.

Bonus Slide 3: Curl Noise

Curl noise is a **Divergence Free** vector field

Great for adding **noise or turbulence** to any simulation

Suggested reading:

Curl-Noise for Procedural Fluid Flow

<https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph2007-curlnoise.pdf>



I ran out of time and had too many examples already, so here is a bonus slide on curl noise. The suggested reading is super mathy, so don't feel bad if it feels too advanced. I can describe the take away pretty simply though: Use curl noise in low pressure zones. The low pressure zone is always the backside of the object against the flow. From previous examples in this talk, if you did a dot product with the distance field gradient and flow vector, the negative area would be the low pressure zone. You can start by simply using that as a mask to control curl noise intensity in flowmap vectors. For additional modelling, you can take additional offset samples of the DF against the flow gradient to refine the shape of the wake. The listed reference will go into how to make this more correct and divergence free while preventing flows from intersecting analytical boundaries, among other things.

Bonus Slide 4: Scale Flowmaps

Modification on flowmaps: Scale by point instead of advection



Scale flowmaps were used in the ocean video to give the foam falling off rocks effect. The timing was controlled using the repeating frac function with its period matched to the largest wave. There is a UE4 material function called 'ScaleUVsbyCenter' that can be used to replace the advection inside of the Flowmap function. To scale by center from scratch all that you need to do is do $\text{NewUV} = (\text{StartUV} - \text{CenterPoint}) * \text{Scale} + \text{CenterPoint}$.