.

○

# *Global Dynamic*
# *ue5*
# *Wind System*

∼ ○ ∼

an Unreal Engine project.

**Written by** Tamwyn,
**from** Galtouz Games.
**Last edit on** November 23, 2024.

The goal of this project is to provide a global wind system for your games inspiered by the laws of electrostatics. After placing some wind source and well you will obtain a three dimensional force field that can be used to move the object and the trees around. The wind direction can then make some spirals etc and is complexer than a global wind direction for the map. For the fine tuning you can add some boosters at certain locations and give them a direction to boost at. The trees and the particle system can follow the wind movement resulting in a great world dynamic.

The system can evolve with the time if you want or just be static an calculated at the beggining. The complexity of the calculation grows linearly with the number of source/well/boosters. A 2D texture carries the force field to the shader to allow the verticies of your trees, fabric etc to move in the correct direction. Moreover a phase factor can be taken into account to allow different response to the wind field depending on the material. For exemple a bigger tree will react with a small delay compared to a fir tree.

# Contents

# 1 How to use

This plugin has 4 main components to use. Two of them are core components and the other two are cosmetics.

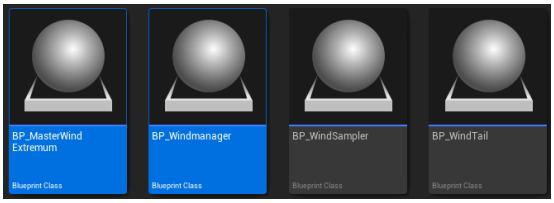



**Figure 1:** The four relevant components. The two outlined are the core components and the other one are cosmetics.

0 Download a ZIP of the repo and cut past the folder content at the same location than your content folder. This is important to do carefully to avoid any issue with the references.

1 Place the extrema where you want to and define them as boosters, wells or sources. You can play with their influence strength.


**Figure 2:** The three types of extremas. The green, orange and yellow debug circles gie you the radius at wich wr have 25, 50 and 75% of the influence.

2 Place the wind manager an store the wind extrema inside the array. Redraw and reshape updates the shaders.

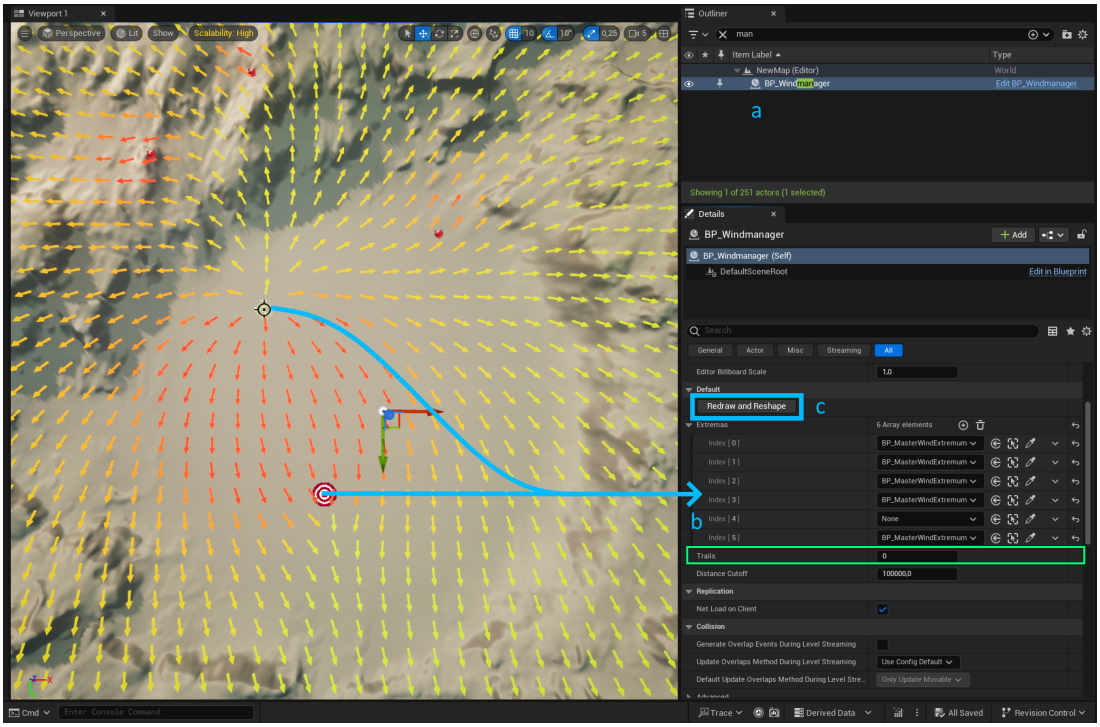



**Figure 3:** The wind manager. The green parameter stays how much wind tails will spawn arround the player.

3 Let the system know about the size of the world so it can correctly map the wind deformation to the trees. To do, this give the side length of the world to the material parameter collection `MP_Wind`.

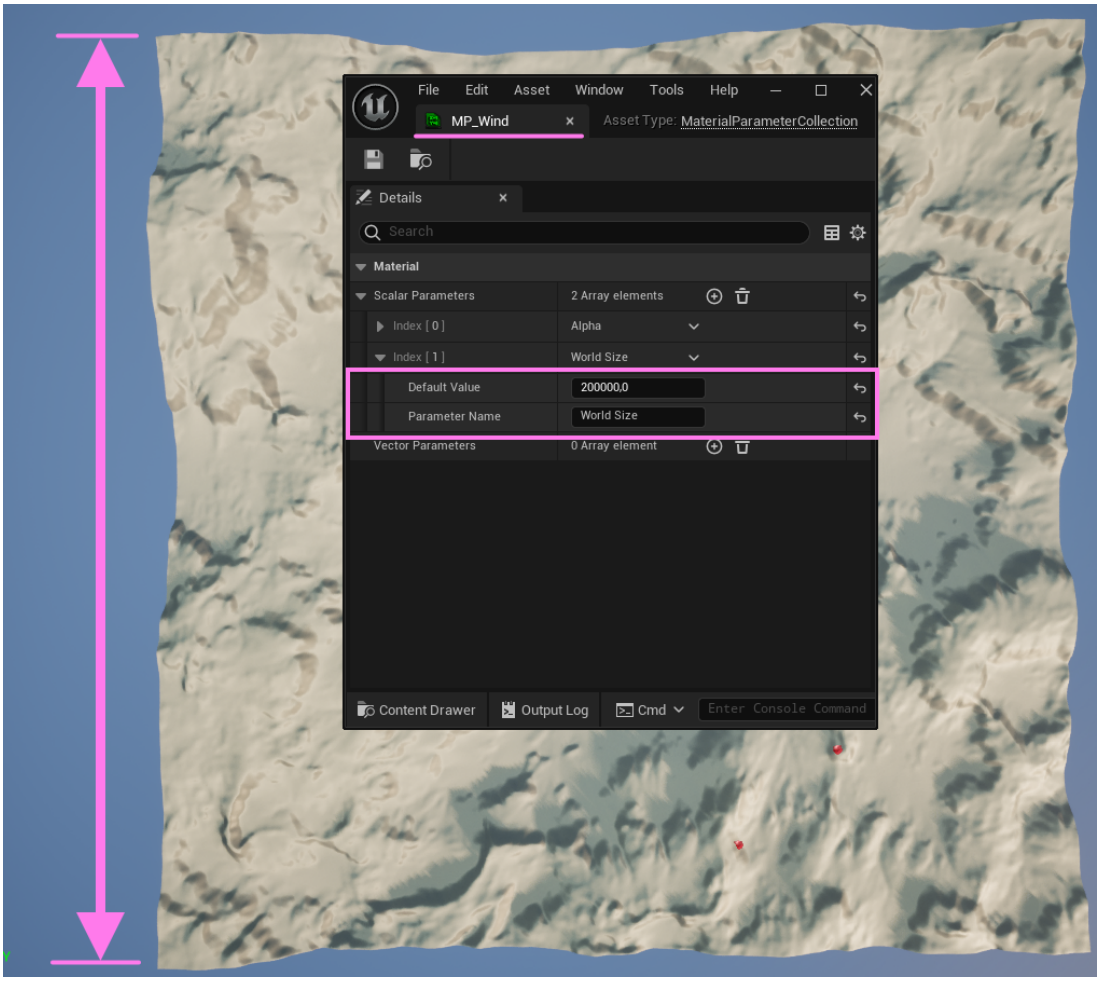



**Figure 4:** The parameter collection to let the system now about the world size.

4 Debug the wind using the wind sampler tool. Define a grid resolution and, and a spacement. Specify the wind manager to make to computations.

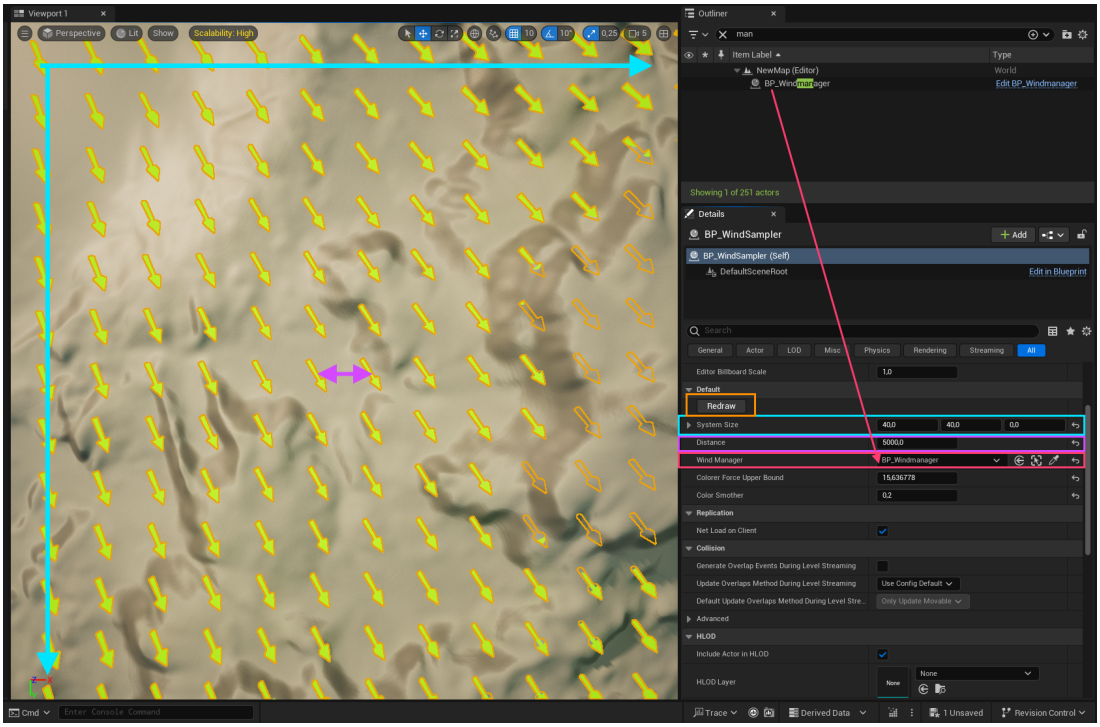



**Figure 5:** The wind sampler tool. Please reference the wind manager or this wont work properly.

5 Use the `Dynamic Wind` matieral function in your folliage.

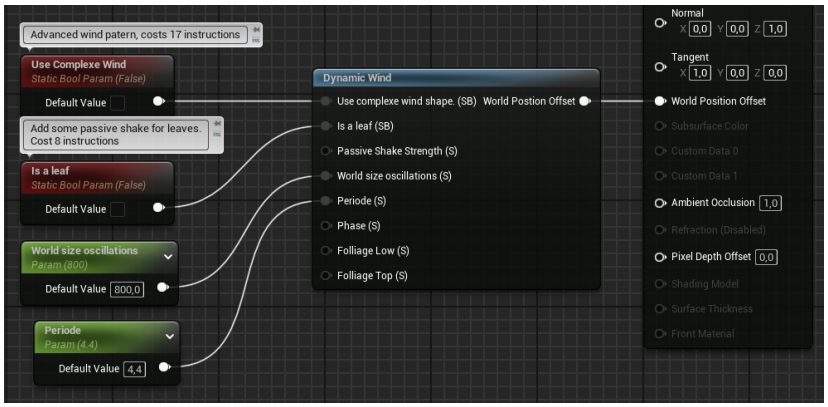



**Figure 6:** The use of the materal function.

An there we have it! The tail respawn after a while. To desactivate the trace on spawn, please search the event `RetraceAtSpawn` in `BP_WindTail` and naviagte to the node `Line Trace By Channel`. There, set the `Draw Debug mode` to none.

# 2 Physical principle

## 2.1 The force field

The laws of electrostatics or more presicly the Coulomb force is described as a force that attracts or repulses a particle based on its distance from a well/source. This is quite simple, if your are nearer to one of this so called extremas, you're going to expreience a stronger force than from an other extrema.

Now where does the force of an extrema point at? If we draw a line from the extremum to a particle, the force will act on this line and point towards the extremum if we are in an attractive

setup and outward if we are in the repsulive case.

The total picture is then represented as a sum over all forces as Newton describes it. Put in a mathematical way we have for the force of one extremm $i$:

$$F_{\text{extr},i}(\boldsymbol{r}) = I_i \cdot \frac{\boldsymbol{r} - \boldsymbol{r}_i}{||\boldsymbol{r} - \boldsymbol{r}_i||_2^2} \tag{1}$$

With $\boldsymbol{r}_i$ the location of the source and $I_i$ its influence power. This value is negative for a well such that the force points towards the well. For the booster we add the force in the direction that the booster points at and modualte with the distance.

$$F_{\text{boost},i}(\boldsymbol{r}) = I_i \cdot \frac{\boldsymbol{k}_i}{||\boldsymbol{r} - \boldsymbol{r}_i||_2^p} \tag{2}$$

where $\boldsymbol{k}_i$ describes the pointing vector of the boost. Tweeking the power $p$ we can make the force more sharp or more fallten. This is the `power` parameter in the Blueprints.
We now have $n \in [0, N_e]$ with $N_e$ the number of extrema and $m \in [0, N_b]$, $N_b$ the number of boosts. We then can write for the total force experienced at a location $\boldsymbol{r}$:

$$F_{\text{wind}}(\boldsymbol{r}) = \sum_n F_{\text{extr},n}(\boldsymbol{r}) + \sum_m F_{\text{boost},i}(\boldsymbol{r}) \tag{3}$$

This function is stored in the `BP_WindManager::GetWindAtLocation methode`. For those unknown with this notation it means `ObjectWhereTheFunctionIsStored::TheFunction`. In cpp the object is a class, but UE5 on the editor side has a class per file.


## 2.2 Derivation of a position update

We can simply use the newton lays of motion to derive the position update. We have for a mass of $m = 1$

$$\boldsymbol{F}(t) = m \cdot \boldsymbol{a}(t) \Leftrightarrow \boldsymbol{a}(t) = \frac{\boldsymbol{F}(t)}{m}.$$

Knowing $\frac{\mathrm{d}\boldsymbol{v}(t)}{\mathrm{d}t} = \boldsymbol{a}(t)$, we can get the velocity by integrating the acceleration over the time:

$$\boldsymbol{v}(t) = \int \boldsymbol{a}(t)\mathrm{d}t = t \cdot \boldsymbol{a}(t) + \boldsymbol{v}_0$$

and knowing that $\frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t} = \boldsymbol{v}(t)$ or the usaly "velocity equals distance over time" we obtain:

$$\boldsymbol{r}(t) = \int \boldsymbol{v}(t)\mathrm{d}t = \frac{1}{2}t^2 \cdot \boldsymbol{a}(t) + \boldsymbol{v}_0 \cdot t + \boldsymbol{r}_0 \tag{4}$$

$\boldsymbol{r}_0$ is the initial position of the particle and $\boldsymbol{v}_0$ the initial velocity. This are constant that adds up when we integrate.

# 3 Using the force field

If you now intend to use this force field information you can simply sample the force at a location. Then use the timestep to move the actor according to the force it should have at this location for the next frame. This is done for exemple in the wind tail actor `BP_WindTail`.

## 3.1 Optimising the use

So we have seen how the use of `BP_WindManager::GetWindAtLocation` methode scales linearly with the number of extrema and boost. For this reason we are not going for every tail to sample the force field on each tick. Instead we can compute it every second and store it in a variable that we can use.

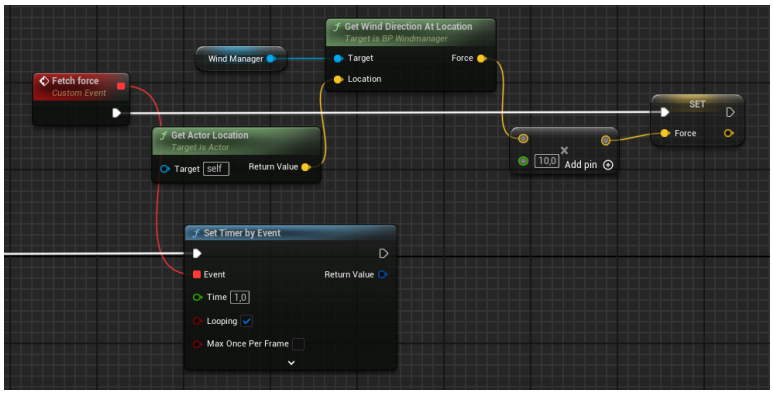



**Figure 7:** The timer updates the force at the time $t_s$, we compute the force at the location $\boldsymbol{r}(t_s)$. A factor of 10 is used. You can modify it as you want to.

## 3.2 Implementation of the motion

Using Eq.4 we can derive the following descritied formula, which means it works for finite time steps and not infinitesimal small ones like in the real life. The time step can be seen as the time between two frames $\Delta_t$ if we wish to update at each frame.

$$\boldsymbol{r}(t+\Delta_t) = \frac{1}{2}\Delta_t^2 \cdot \boldsymbol{F}_{\text{wind}}(\boldsymbol{r}(t_s)) + \Delta_t \cdot \boldsymbol{v}(t) + \boldsymbol{r}(t) \tag{5}$$

We can make a few observations. First the velocity is the position difference between the two updates divided by the time passed. we then have $\boldsymbol{v}(t) = (\boldsymbol{r}(t) - \boldsymbol{r}(t-\Delta_t))/\Delta_t$.

Second in the formal dervation we saw that the acceleration has to be multiplied with the square of the time the system lives $t^2$. This is because we add the acceleration to the *spawn* position of the system $\boldsymbol{r}_0$. Here however we add to the position *of the previous frame*. This means if we multiply by the simulation time $t$ to the square, the acceleration is going to play a way to inportant role which will be incorect. For this reason we use the time step $\Delta_t$.
Putt in a different way, the total displacement information in the first case is stored in the total time regarding the spawn position. In the second case this information is stored in the previous position so we just need a tiny shift by the acceleration. The same observation is or the velocity $\boldsymbol{v}_0$.

In Eq.4 the accceleration is describing the whole motion of the system involving an initial velocity but here the motion is mainly stored in the position. We only use the velocity to shift a bit the position and it's going to be implicitly stored in the position on the next frame.

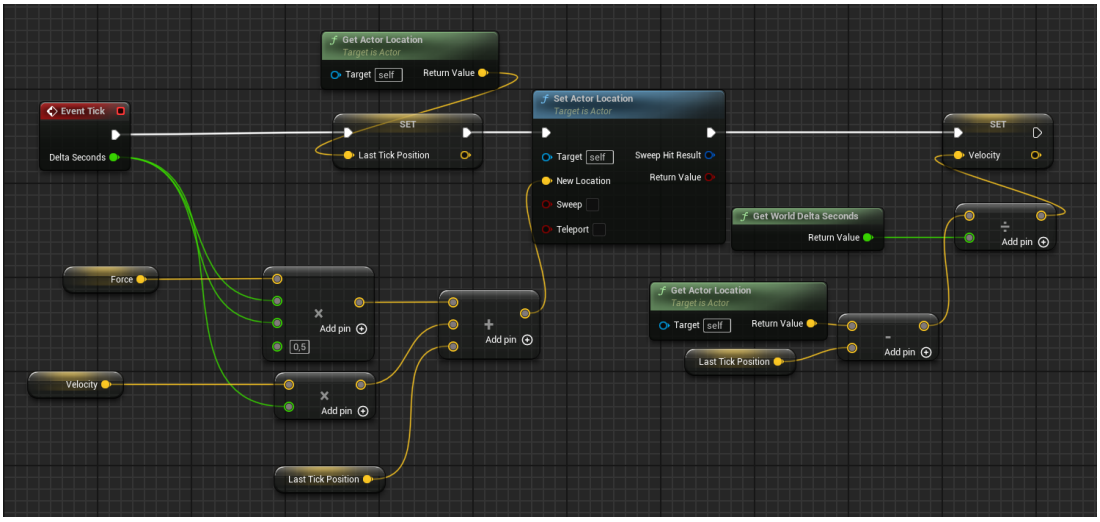



**Figure 8:** A the frame $t + \Delta_t$ we multiply the acceleration by $0.5\Delta_t^2$, the velocity with $\Delta_t$ and add it to the location from the previous update $\boldsymbol{r}(t)$. *After* the translation we update the velocity to be used on the next frame. If we update the velocity before the translation we are going to have $\boldsymbol{r}(t) - \boldsymbol{r}(t) = 0$. The last tick postion variable is used to compare before and after the update.

## 3.3 Moving the tail around its center

To achieve a more complexe motion we can decide to move the particle system around the tail center. This can be changed in `BP_WindTail` as the variable `Type`. For exemple the `SwingZ` mode makes the system having a vertical oscillation during the travel. One could also rotate the system around the propagation axis to make a spiral or simply let the line follow the current by doing nothing.

**SwingZ mode** This methode uses a sinusoidal function to make the tail oscillate along the $z$-axis of the tail.

$$\boldsymbol{r}_{\text{niag}}(t) = I_z \cdot \cos(\omega t + \varphi) \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + \boldsymbol{r}_{\text{tail}}(t)$$

The niagara system attached to the actor moves relatively to the actor iteself. $\boldsymbol{r}_{\text{tail}}(t)$ is calculated in the tick by the derivation we just made.

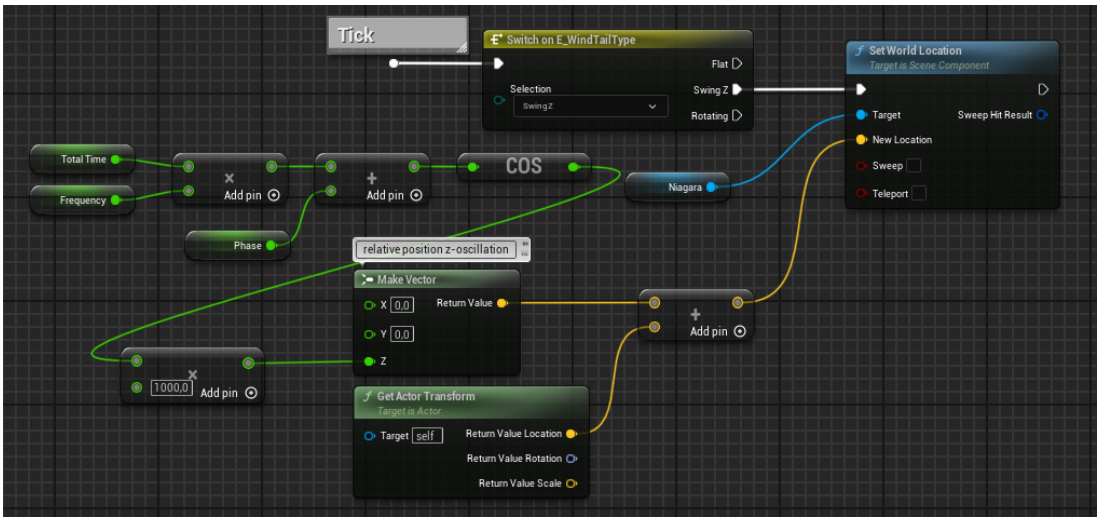



**Figure 9:** The tail is moving around the center of the actor. The oscillation is done by a sinusoidal function.

**Spiral rotation mode** Here we use the up and right vector of the facing direction of the force. The oscillation will then modulate the strength of each axis. Puting the same phase $\varphi$ results in a cricular rotation. Varying this parameter will interpolate the rotation between a diagonal to a ellipse and then a cricle. More exemple can be found at `https://www.idex-hs.com/resources/resources-detail/understanding-polarization` at section $a$.

$$\boldsymbol{r}_{\text{niag}}(t) = I \cdot \cos(\omega t + \varphi) \cdot \hat{\boldsymbol{e}}_{\text{up}} + I \cdot \sin(\omega t + \varphi) \cdot \hat{\boldsymbol{e}}_{\text{right}} + \boldsymbol{r}_{\text{tail}}(t)$$

with $I$ the amplitude of the oscillation and $\hat{\boldsymbol{e}}_{\text{up}}$ and $\hat{\boldsymbol{e}}_{\text{right}}$ the up and right vector of the facing direction of the force.

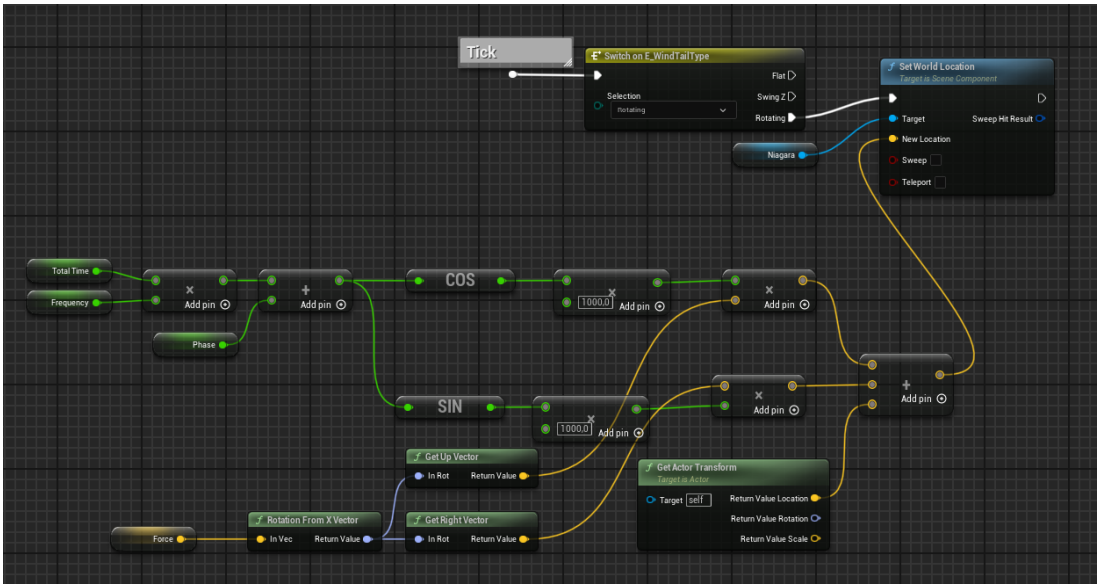



**Figure 10:** The particle is moving around the center of the tail. The rotation is done by a different sinusoidal function on each axis, perpendicular to the movement direction of the tail.

The function `BP_WindTail::PrintRoationArrows` can be used to visualise the rotation components of the tail.

### 3.4 The tail lifetime cycle

The tail spawns arround the player and move for a random time. After reaching the end of its life, the particle stops progressively and the actor is teleported back in the neighborhood of the player. The parameter are redefined in a random way to achieve more diversity. The event `BP_WindTail::RetraceAtSpawn` is used for this purpose. The event calls back itself after the lifecycle of the tail is over.

## 4 Shader computation and foliage

In this section we are going to see how storing the three dimensional force field in a 2D texture can be used to move the verticies of the foliage. For this purpose we are going to use render targets and more specifically blend between two render targest (called here buffers) to achieve an optimised dynamic wind evolution on the foliage.

The way to pass datas to a shader is to use textures. We are going to encode the 3D vector representing direction of the wind in each pixel. The coordinate of the pixel can then be mapped on the world position. By doing so we can play with the UV of the texture to sample the wind force at the location where the pixel (of the 3D scene on the viewport) is located. This system is very good scalable so if you want to lower of higher the resolution of the wind, you can improve the resolution of the render target. A resolution of $64 \times 64$ pixel was found to be a pretty good value for a $200k \times 200k$ unit sqaured map.

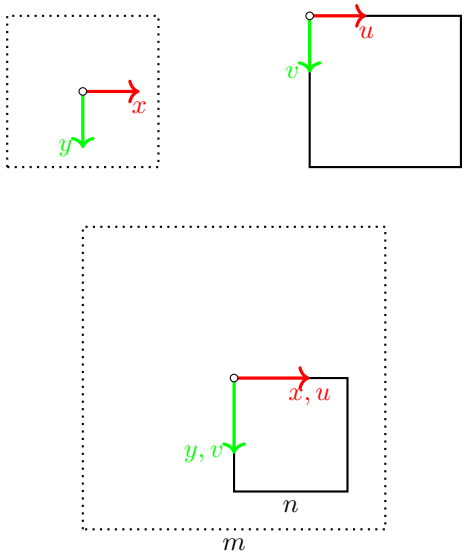
**Note:** (for mathematians or phisicians) The following uses the force field as a $\boldsymbol{k}$ "momentum" field to displace the shader.

## 4.1 Capturing the 2D texture

The first step is to capture the 3D texture. Trained eyes will have noticed that a texture is 2D and not 3D. This means we can only store, per texture, a *slice* of the world to make the tree shake. A realistic aproach would be to have mutliple textures that we can choose according on the $z$ value. However this is going to cost us a lot of memory so we are going to use a trick.

If we think about it, the terrain is just a texture elevated along the $z$ axis, in other words if we look at it from the top, it looks flat. This means, instead of storing multiple slices, we could make some traces from the sky to the ground to store the $z$ location of the terrain, where we expect the trees to be.

**Getting the plane $xy$-coordinates** To find the right transformation between the world coordinate (XY) and the texture coordinate (UV) we can start with a scheme. Here we supperpose the UV and the XY origin.




An important information has to be highlighted, UE works in a left handed coordinate system. This means that the $v$ axis is pointing down. This differs from the conventions in mathematics/physics where the $y$ axis is pointing up.
So the first step is to scale the texture to fill the world size and then replace the center of the UV to the top left corner of the XY terrain. We can take a coordinate $\boldsymbol{r}(x, y)$ on the landscape and write a transformation to get the UV coordinate $\boldsymbol{k}(u, v)$.

$$\boldsymbol{r}(x, y) = \boldsymbol{k}(u, v)\frac{m}{n} - \frac{m}{2}\begin{pmatrix}1\\1\end{pmatrix}$$

In the same way the transformation from the world coordinate to the UV are:

$$\boldsymbol{k}(u, v) = \left(\boldsymbol{r}(x, y) + \frac{m}{2}\begin{pmatrix}1\\1\end{pmatrix}\right)\frac{n}{m}$$

which is the inverse process. We first move the world coordinate so that the upper left corner lays on the origin of the UV and then we scale it down.

This transformation can be found in the shader material function `MF_WindDeform`. The convention is to have $\boldsymbol{k}(u, v)$ using $u, v$ betweem 0 and 1. In such circumstances we have $n = 1$.

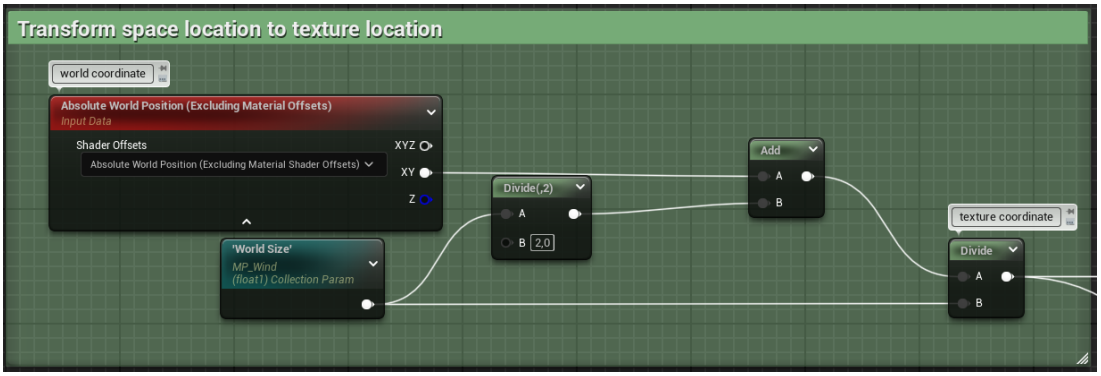



**Figure 11:** The transformation from the world coordinate to the UV coordinate. The `World Size` scalar parameter is the world side length $m$. In the default UE5 project this is $200k$ units. You should set it to the size of your world. If too small you may see the wind patern repeat itself and having some discontinuities on the edges. On the other hand, if too big, you will project only a part of the world on the texture.

We can then find to a map location the corresponding pixel in the texture to write the force at.

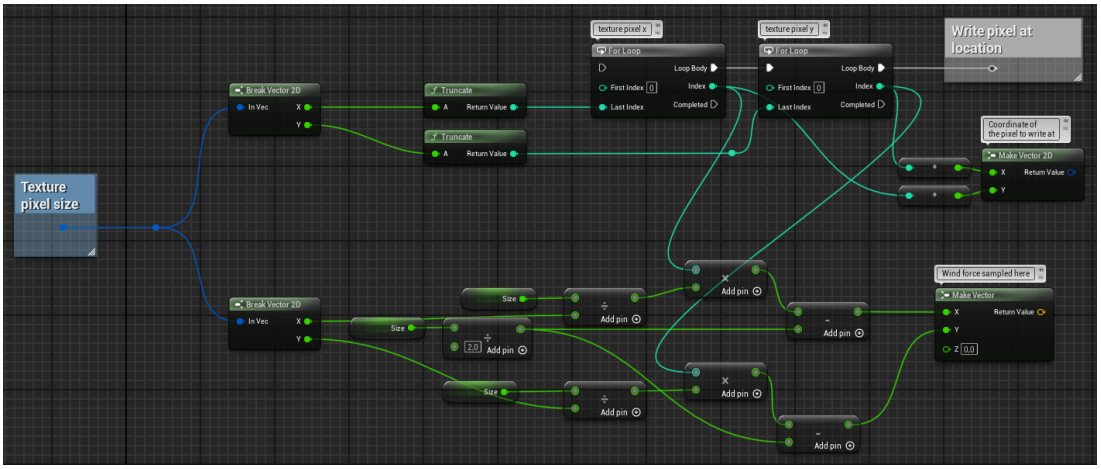



**Figure 12:** How we determine the pixel location from the world position. The texture beeing $64 \times 64$ pixels, a U of 1 corresponds to 64 pixels.

**Getting the $z$-coordinate** The next step is to get the elevaion $z$-coordinate. This is done by casting a ray from the sky to the ground. Since the terrain is flat and stays roughly the same during the runtime, this only need to be made one at the beggining og the game. For the $x$ and $y$ coordinate we can base ourselves on the texture size as just discussed. The trace should start a bit above the heighest point of the map and end a bit below the lowest point. Then we can store the $z$ value in an array to find the force later at this heigh.

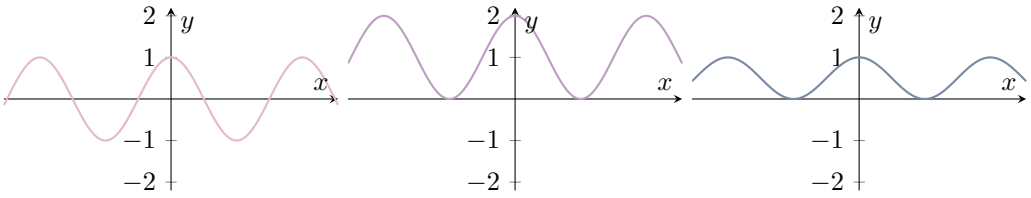
However there is no possbility to write matrcies in UE5 where the enrty index would be the x and y coordinate and the value the $z$ coordinate. However we can imagine puting this matrix row by row in a array. This means if the world in $10 \times 10$, the first 10 values of the array describe the z coordinate from (1,1) to (1,10), the next 10 values from (2,1) to (2,10) and so on. The general formula is then:

$$\text{index} = (y - 1) \cdot x$$

this is achieved in `BP_WindManager::ComputeHeight`. Knowing in advanced that the array should be $N_x \cdot N_y$ items long, we can already preallocate the entries to save some time. We can use the same forula to get the $z$ value out of the array while writing the wind.

## 4.2 Writing the wind

Now that we have the actual position, we can use our force field to sample the strength and direction of the wind at that location. The main problem is now how we are going to encode this in a texture. Well, the strength is represented by three components, the $x$, $y$ and $z$ component of the force. We can then use the red, green and blue channel of the texture to store this information. However if the wind points in $(-1, -1, -1)$ or has any negative values, we are going to have a problem. In fact the way texture are encoded restricts us in the use of negative values. All the values has to be maped between 0 and 1. There is fortunately a methode to store negative values, which is also used in normal maps. An exemple involving a sinusoidal wave can be helpful.




$\cos(x)$ $\cos(x) + 1$ $(\cos(x) + 1)/2$

By doing so, we have stored the same information (the curve shape) in the zero to one range. Whene decoding we value we should do the opposite operation otherwise our wind will only show in the poisitve direction with a maximal strength of 1. Asuming that the maximal value of a function $f_{\text{real}}$ should be $f_{max}$, we have

$$f_{\text{encoded}} = \left( \frac{f_{\text{real}}}{f_{\text{max}}} + 1 \right) \frac{1}{2}.$$

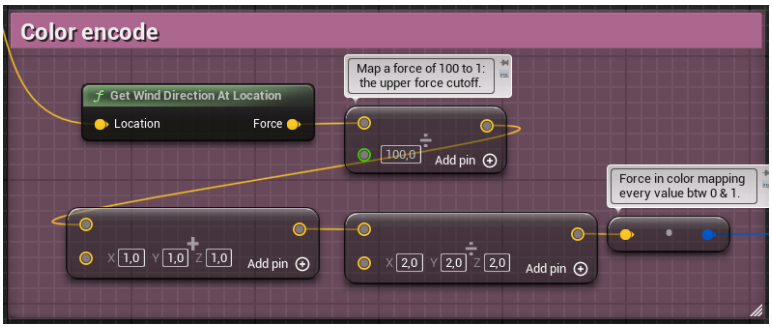
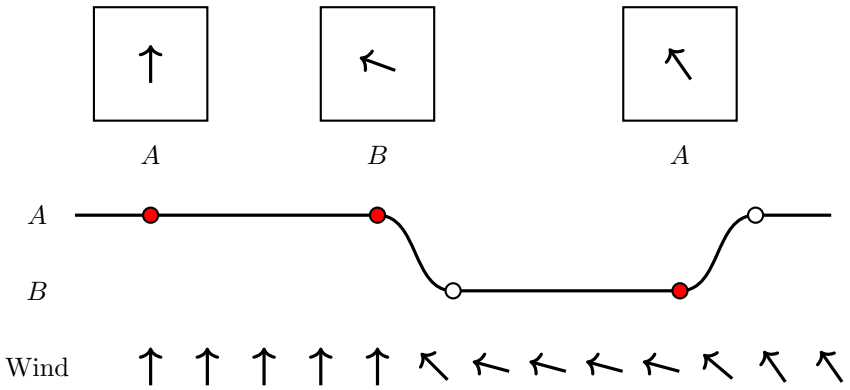



**Figure 13:** We treat our signal as the force and map every component between 0 and 1 according to the upper scheme.

The deocde when sampling the texture is then simply $f_{\text{real}} = 2f_{\text{encoded}} \cdot f_{\text{max}} - 1$. This is done in `MF_WindDeform`.

**Achieving dynamic shader wind** Until now, when we recompute the wind for the shader and therefore modify the texture, we observe a sudden change in the wind patern if the force field has changed. This is due to the fact that we are directly writing the new wind in the texture. A first way to solve this would be to recompute the texture every tick. This is however a demanding operation. The BP script is executed on the CPU. Writing the texture takes however place on the GPU. Now a second important information. The code can run on a thread. It's basically an instruction line. Like you would be in a queue. Whene an instruction does something, this blocs the whole thread (queue). Writing the texture takes some time because the information travel from the CPU to the GPU and then back to tell the GPU is finished. In this sense writing a texture a blocking operation.

Now the sad part about it is that if this queue happens to be the "game thread", the game will freeze until the texture is written. This causes periodical anoying freezes. The first solution would be to write the texture every 10s for exemple, store it and interpolating between the two textures stored in $A$ and $B$ to acheive a dynamic transition between the two states.




In this timeline, the red dots correspond to the storing of the texture in the cooresponding buffer. When a new buffer is written, the wind is interpolated between the current buffer and the newly written buffer. Resulting in a continuous wind evolution, even if the wind is only captured three times. This is achieved in `BP_WindManager::CacheNewWind` and the actual update is done in the tick.
The function we use there to interpolate is an hyperbolic tangent. This looks very similar to a besier curve. We have to shift it have the slopes bewteen 0 and 1 because it's nicer to work with. Additionaly the whole slope should be mapped using an $x$ between 0 and 1, so we have to increase the exponent to make this slope stronger. In the limit of $x$ going to infinity, the function is a step at 0.5. Going back means simply starting from zero and inverting the curve, what is done with the $(1 - x)$ in the select node.

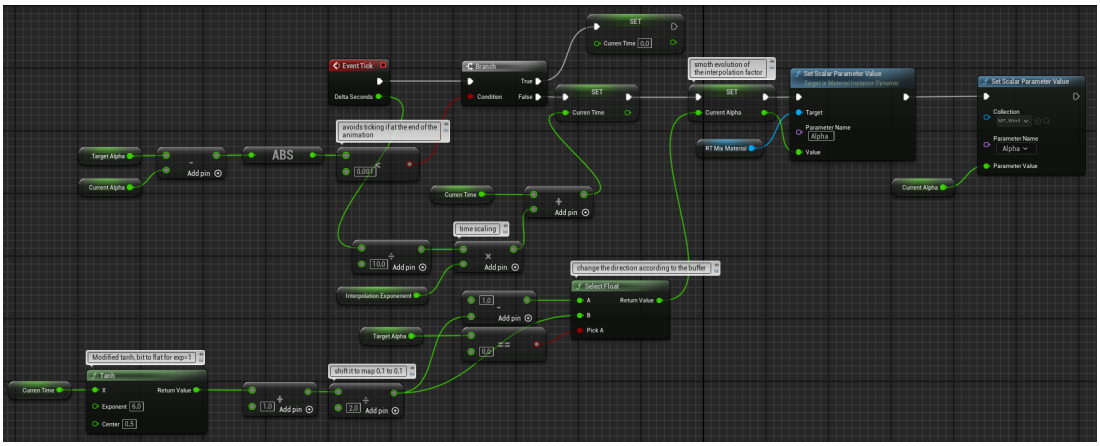



**Figure 14:** The interpolation function used to blend between the two textures. The actual shader interpolation is just a lerp between the two textures and is found in `MF_WindDeform`.

## 4.3 Applying the wind on the foliage

The breakdown of the shader itself quit simple. First we need to displace the verticies in the direction stored at by the texture. Second we can progagate a wave trough the foliage. For the later one, you might familiar with the use of a normal map combined to a WorldAlignedTextures used in grass shaders to achieve a wavy effect. Well, we are going to use something more simple but wich work very good as well and follows the lines.

This is directly inspiered by the theorie of the plane wave propagation in physics. Asuming that the wave propagates itself along the $\boldsymbol{k}$ vector, we have for the amplitude of the wave at a given point $\boldsymbol{r}$ and time $t$ `https://en.wikipedia.org/wiki/Sinusoidal_plane_wave`:

$$A(\boldsymbol{r}, t) = A_0 \sin(\boldsymbol{k} \cdot \boldsymbol{r} - \omega t + \varphi)$$

$\boldsymbol{k} \cdot \boldsymbol{r}$ describes a scar product and is a scalar gien as $k_x r_x + k_y r_y + k_z r_z$. Another description is the following:

$$\boldsymbol{k} \cdot \boldsymbol{r} = |\boldsymbol{k}||\boldsymbol{r}| \cos(\theta)$$

where $\theta$ is the angle between the two vectors. If $\theta$ is 90 or $-90$ degrees, the cosine is null and this term dosn't contribute to the phase of the wave. This means, if this angle is observed bewteen the velocity and the location we get 0. On the other hand if the velocity is aligned with the vector position, the phase contribution is maximal and the tree is going to be a bit in advance than the other locations.

The $\omega t$ term is the time evolution of the wave, which allows the propagation of the wave. The $\varphi$ term is an additional phase for the wave. This can be used to add some delay in a tree if it's bigger for exemple.

Now we can multiply this oscillation with the wind displacment to acheive a "real" wave propagation. However the sine gives values between -1 and 1. This means that the tree is going to move back an forth. This is not exactly what we want. The tree can become straigth again but it doesn't tilt in the opposite direction of the wind. This means we want to exclude the negative values but still want to keep the complete oscillation. This can be done with the similar methode we saw earlier to encode the texture with $(\sin(x) + 1)/2$.

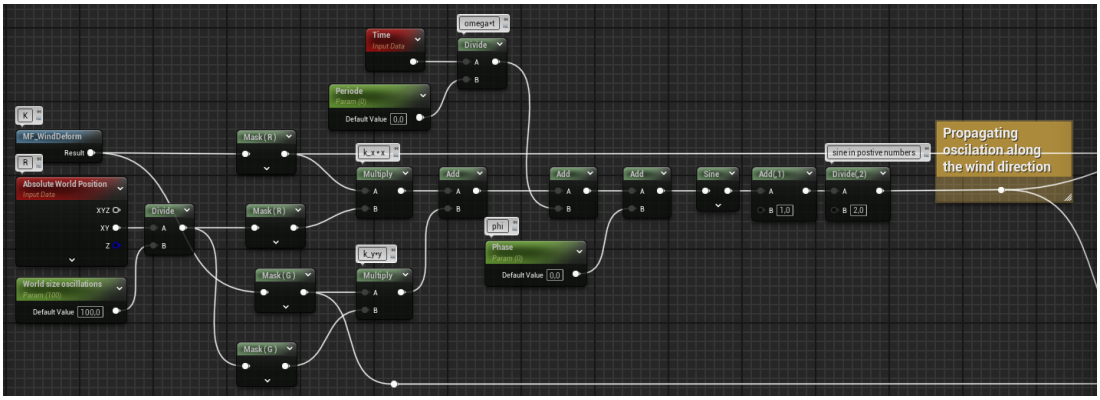



**Figure 15:** The propagating oscillation as an amplitude dependent of the location. The sine wave is then shifted to have only positive values.

However now we get an oscillation between 0 and 1 but we don't have any vector to displace along. Now we can simply multiply the oscillation with the wind vector on each component $x$ and $y$. The $z$ component, we want the tip of the tree to move a bit down. We combine bothe the $x$ and $y$ oscillation, scale it down and use this for the $z$ component.

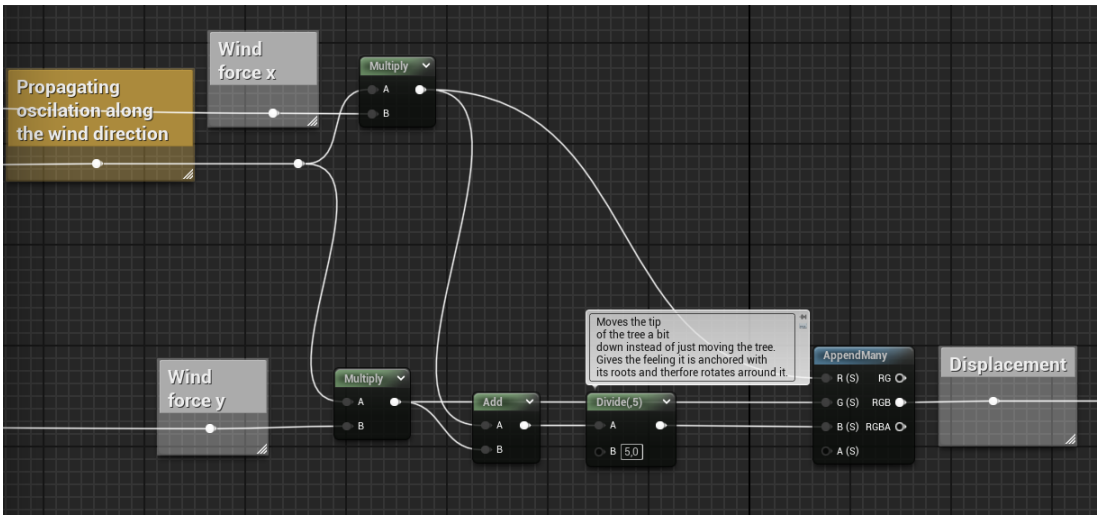



**Figure 16:** The final displacement of the tree. The oscillation is multiplied with the wind vector to get the final displacement.

However now we get a uniform displacement for the complete tree. The tree just slides and come back. We want its top to move and its bottom to be fixed. In other words there is a gradient in the displacement from fixed verticies on the roots to a fully moving top of the tree. To do this we can use the UV of the trees.

We can then simply use the $z$ position of the verticies in the tree space and map it between 0 and 1. 1 beeing the point at which the tree receive the full displacement of the wind. You can adjust this using the parameter `Folliage Top` in the material. These should be unreal units.

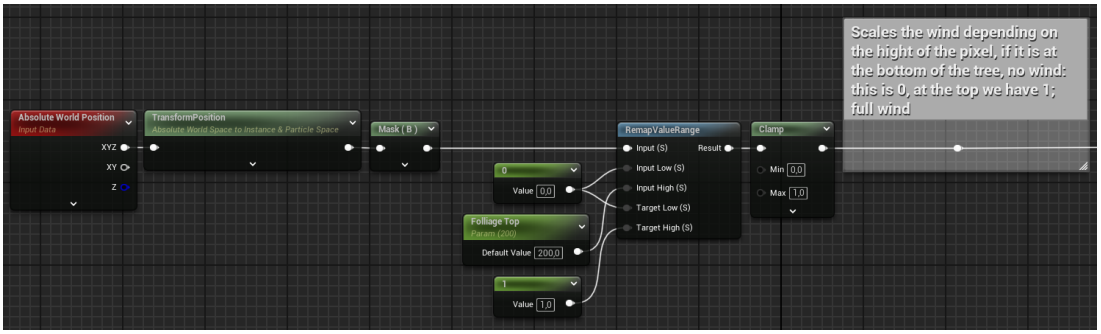
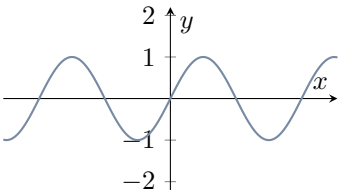



**Figure 17:** The gradient displacement of the tree. The top of the tree is fully displaced by the wind while the bottom is fixed. This scalar just multiplies the displacement we descriped.

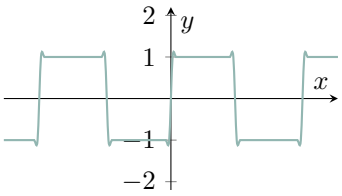
Finely a strengh can be ajusted if a folliage is bigger it will obviouly not move very far.

## 4.4 Complexer movement

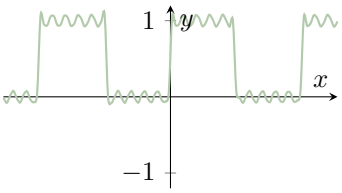
You may have noticed that the wind is not only a simple oscillation and it looks very peacfull like the bottom of the sea. The real wind has some bursts and more chill movments sometimes. We need a complexer signal with tiny oscillations and strong ones overlapping.



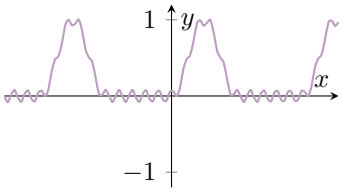

$\sin(x)$




$\text{sign}(\sin(x))$




$\text{Max}(\text{sign}(\sin(x)), 0) + 0.05 \cdot \cos(10 \cdot x)$




$\text{Max}(\text{sign}(\sin(x)), 0) \cdot \sin(x)^2 + 0.05 \cdot \cos(10 \cdot x)$

After doing this construction, we see that we obtain some moments with strong wind and some where just the tiny, chill oscillations are present.
To activate this more complexe wave paterns you can check the `Complex Wind` in the material. This will be done in the cost of 18 additional instructions.

The next level will be to shake the leaves of the tree, beacause for now the tree moves as a mass, treeting the leaves in the same way as the trunc. To do so we can inspire us from the tiny oscillations just presented. Morover we could use the relative heigth of the leaf to make the small shake propagate along the height of the tree. This means we would just have some delay along the heigth. Recalling that the phase can add some delay, we can directly use the heigth information as a phase. Depeneding on the wind strength we can make this oscillations go faster by multipling the time with the speed making for a twice as strong wind, some twice as rapid oscillations. To apply this leaf movements, check the `Is a leaf` parameter

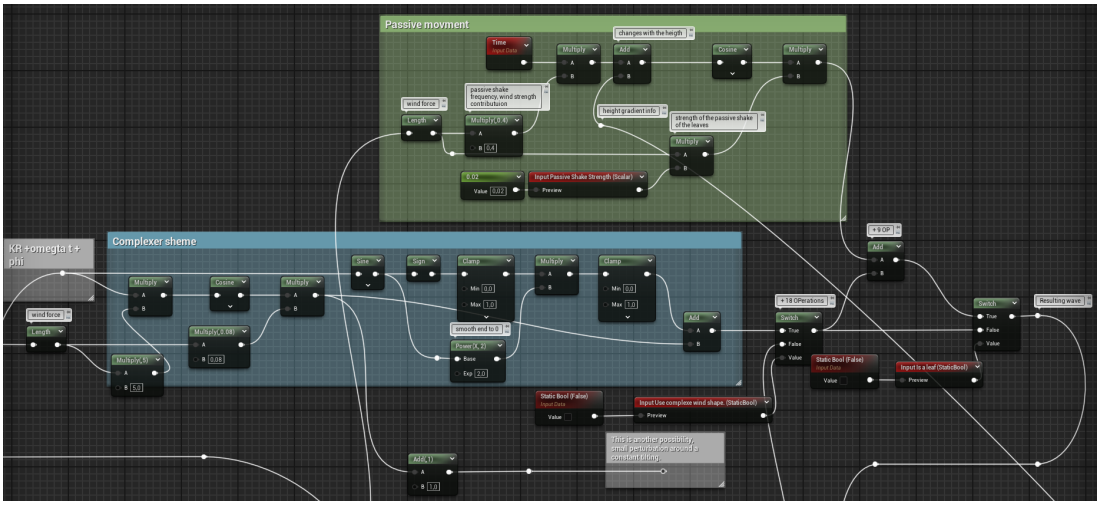



**Figure 18:** The possbility to activate the passive leaf shake and the global complexer wind patern involving bursts and chill oscillations.

### 4.5 Shader cost

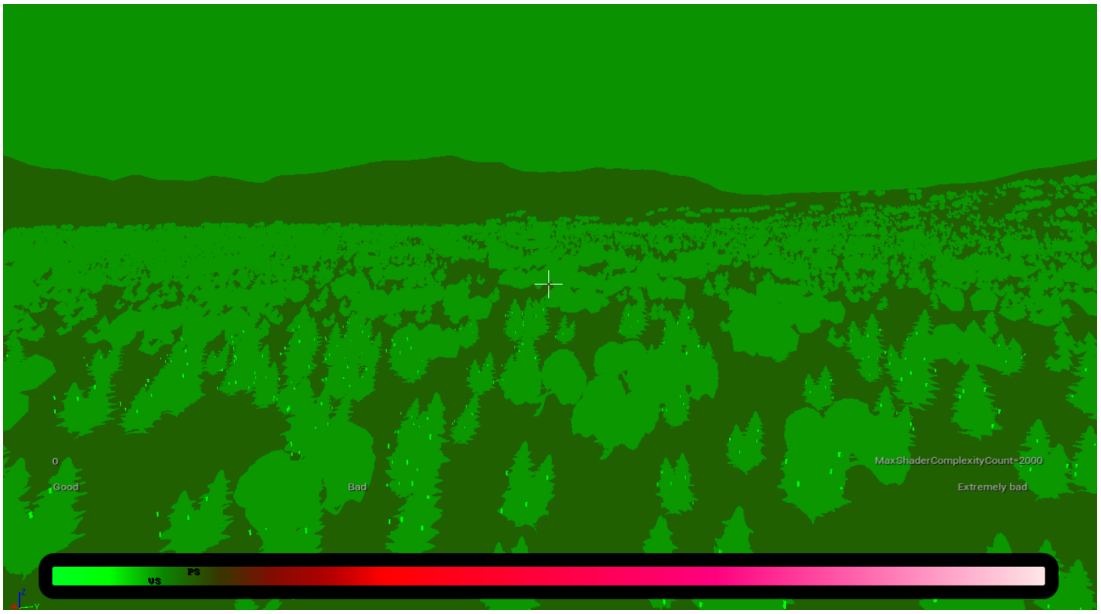



**Figure 19:** The cost of the shader. The cost is arround 220 instructions.

If you have some fps drop you may desactivate the TSR `r.AntiAliasingMethod 3` or maybe lumen `r.DynamicGlobalIlluminationMethod 0`. You should check the LOD of the trees as well. The shader is not very demanding but the more trees you have the more it will cost to render.

## 5 Beyond the scope

Now to get a changing wind you then dislace the sources, the well and the boosters. Remeber to recache the wind system after you have moved the sources. If the motion is continuous please consider updating only every 10s or so. This is to save on performance and avoid freezes. You still want some computational power left for your own systems :p

**Temerature, perturbations etc.** If you have a temperature system, one could imagine it affects the way the sources moves or how strong the wind is. All you have to do is using your own systems to change dynamicaly all the parameters we talked about. The wind system will then take care of the rest, so let place to your imagination.

### 5.1 Todos

One could now write the texture in an asynchronous way to save some perf. This can be done at some point, please open an issue on the github if you want this feature.

### 5.2 Questions?

You can join my discord server at `https://discord.gg/u3K6fjd`[1] or open an issue on the github page at `https://github.com/Tamwyn001/WindSystem`.

[1] You're welcome for a chat as well :)