

Power Up Documentation

Dead Pedal - Working Title

Originally Created: 05/28/25

Last Updated: 07/27/25

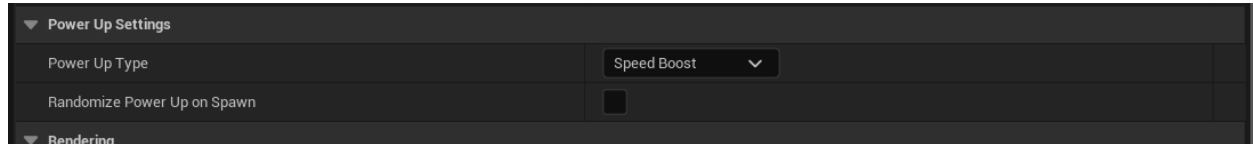
Documentation by: Matthew Wojciechowski

Copyright Faycrest Studios LLC 2025

Power Up System Basics

Spawning and Selecting Power Ups

Place as many BP_PowerupPickup_Base as you want in the level, upon clicking on one you should notice a tab on the right labeled 'Power Up Settings'.



Clicking on the drop down menu next to 'Power Up Type' will allow you to change the type of power up that will be available here, you'll also notice that as you change it in the editor, its VFX will also change in the editor too.

Below that is 'Randomize Power Up on Spawn', if this is ticked, then it will select a random power up from E_PowerUpTypes (which is an ENUM containing each power up) and assign the power up to it.

Keep in mind, if you have 'Randomize Power Up on Spawn' ticked, it will ignore whatever power up you selected for 'Power Up Type'.

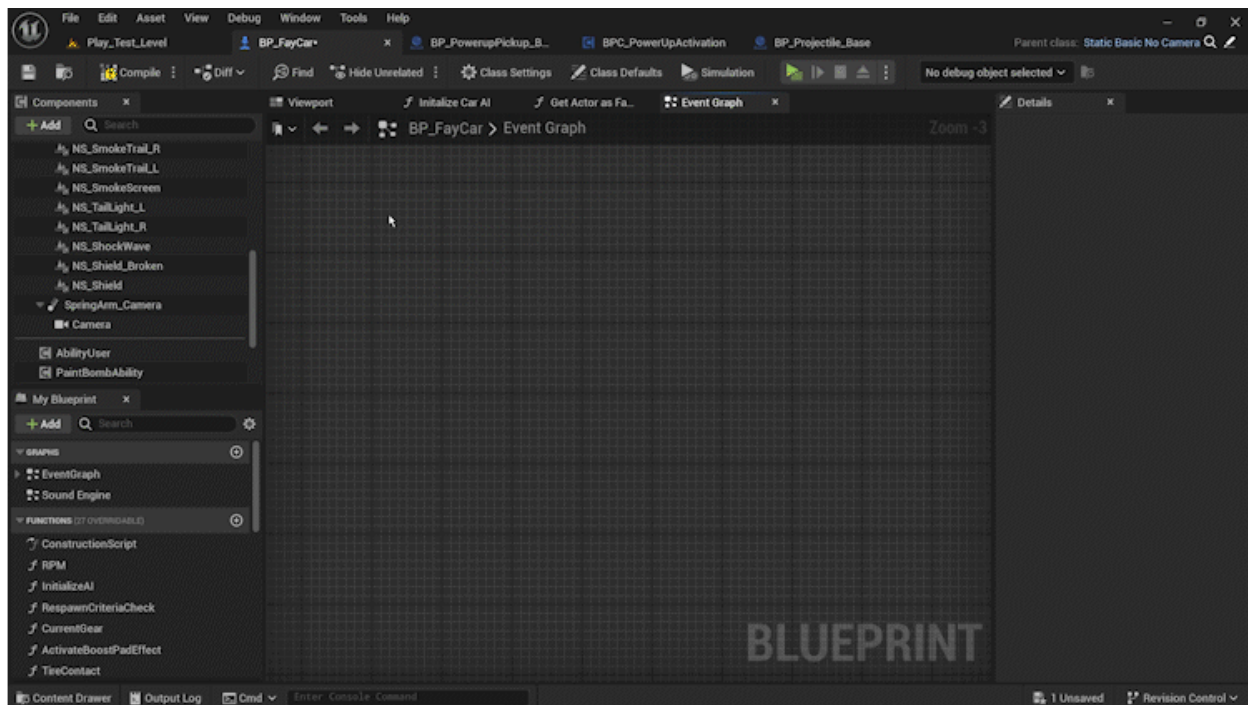
Activating Power Up Abilities

Power up abilities are comprised within the 'BPC_PowerUpActivation' actor component.

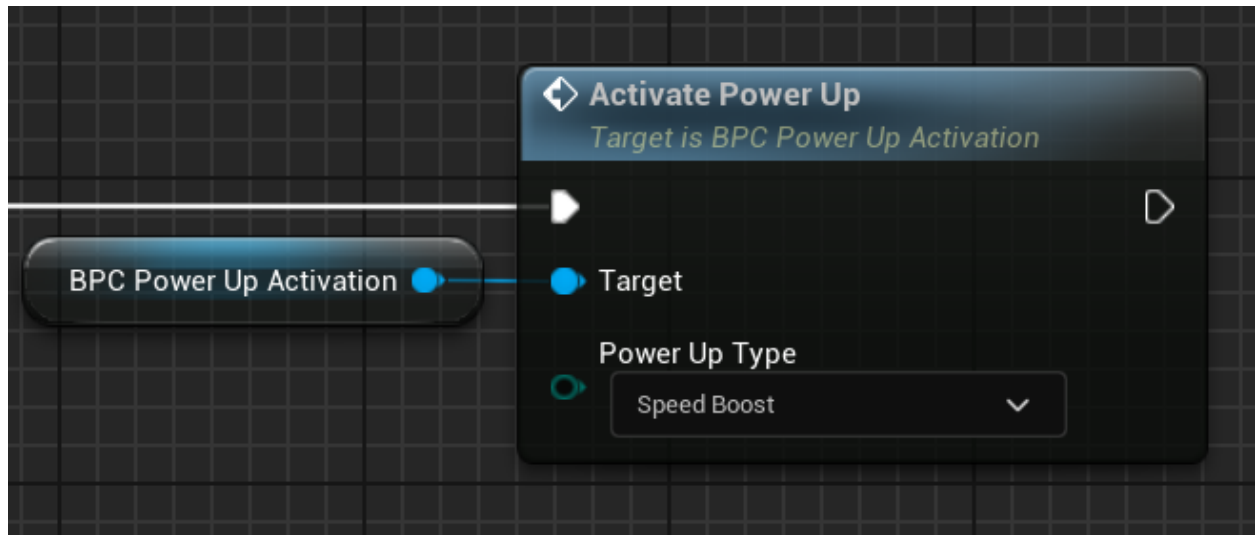
****THIS COMPONENT MUST BE PRESENT WITHIN AN ACTOR WHOM YOU WANT TO ACTIVATE ABILITIES****

To add 'BPC_PowerUpActivation' to an actor:

1. Open the actors blueprint
2. Navigate to the left side panel and find the 'Components' tab
3. Under the Components tab click the +ADD button, this will prompt the search bar
4. In the search bar, type in 'BPC' and you should find the component 'BPC_PowerUpActivation',
5. Click on 'BPC_PowerUpActivation'
6. 'BPC_PowerUpActivation' will now be added to the bottom of your components.
7. To get a reference to it, simply click and drag 'BPC_PowerUpActivation' from your components tab, on to your Event Graph.



To activate an ability, first **get** a reference to the 'BPC_PowerUpActivation', drag off of it and get its 'Activate Power Up' function.

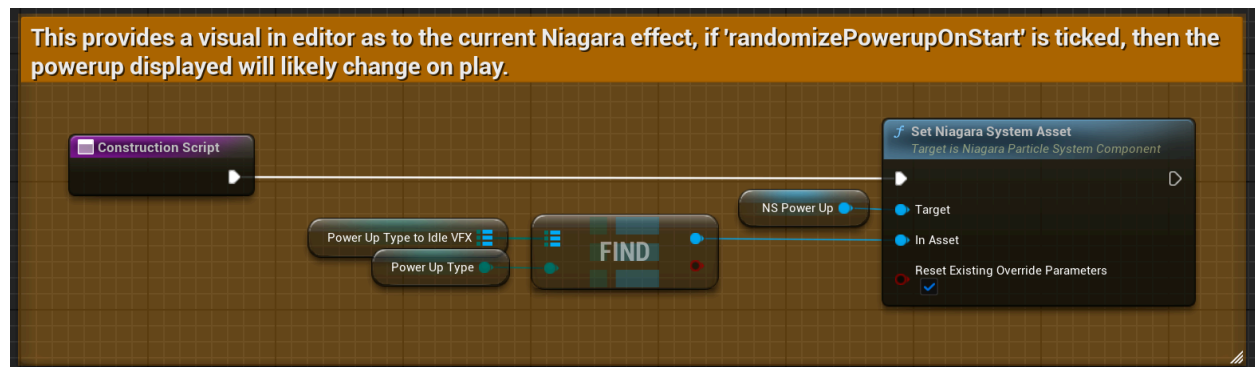


Clicking on 'Power Up Type' will allow you to change the power up you wish to activate, once the function is activated it will activate automatically.

BP_PowerupPickup_Base

The purpose of this blueprint is to spawn power ups for a FayCar to collide with, upon which, will assign the contained power up to the FayCar.

Construction Script



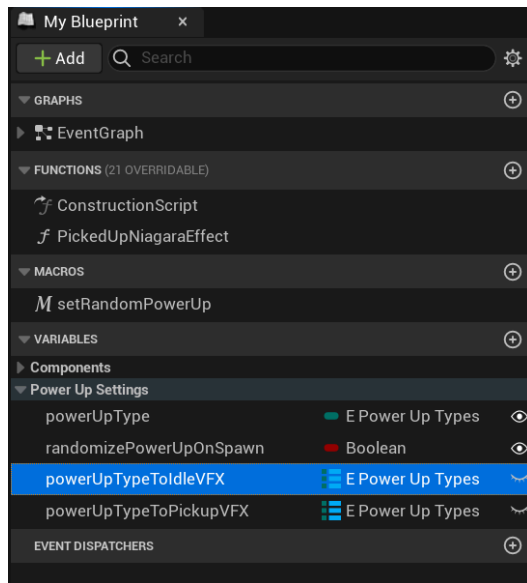
Within the construction script, the VFX for the power up pick up is assigned. Since it's within the construction script, this means that as you change the 'powerUpType' variable, it will change the VFX relative to the appropriate type.

The select node, inputting into the 'In Asset' of the 'Set Niagara System Asset', is where the niagara system is assigned per VFX type, simply click on one of the drop down menus below a power up name and you'll be able to edit the niagara system that appears for it.

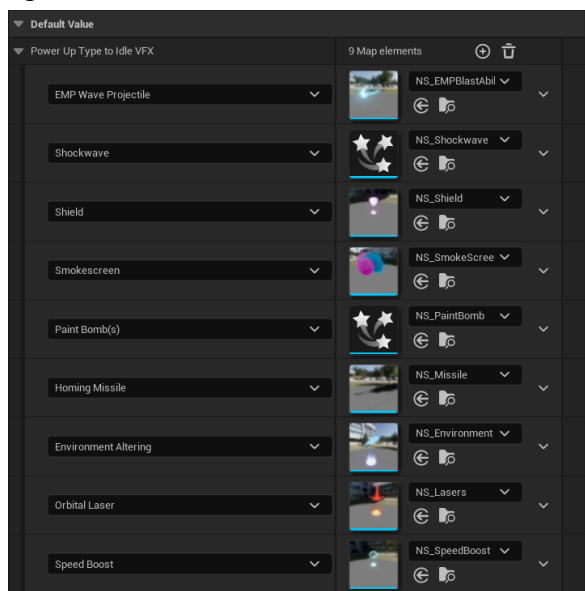
The system uses a map node that corresponds the 'powerUpType' to the desired VFX, then assigns that to the Niagara System.

Editing The VFX

To edit the desired VFX chosen for the selected 'powerUpType', first click on 'powerUpTypeToIdleVFX' located in the Variables section under the 'My Blueprint' tab on the left side of the screen.

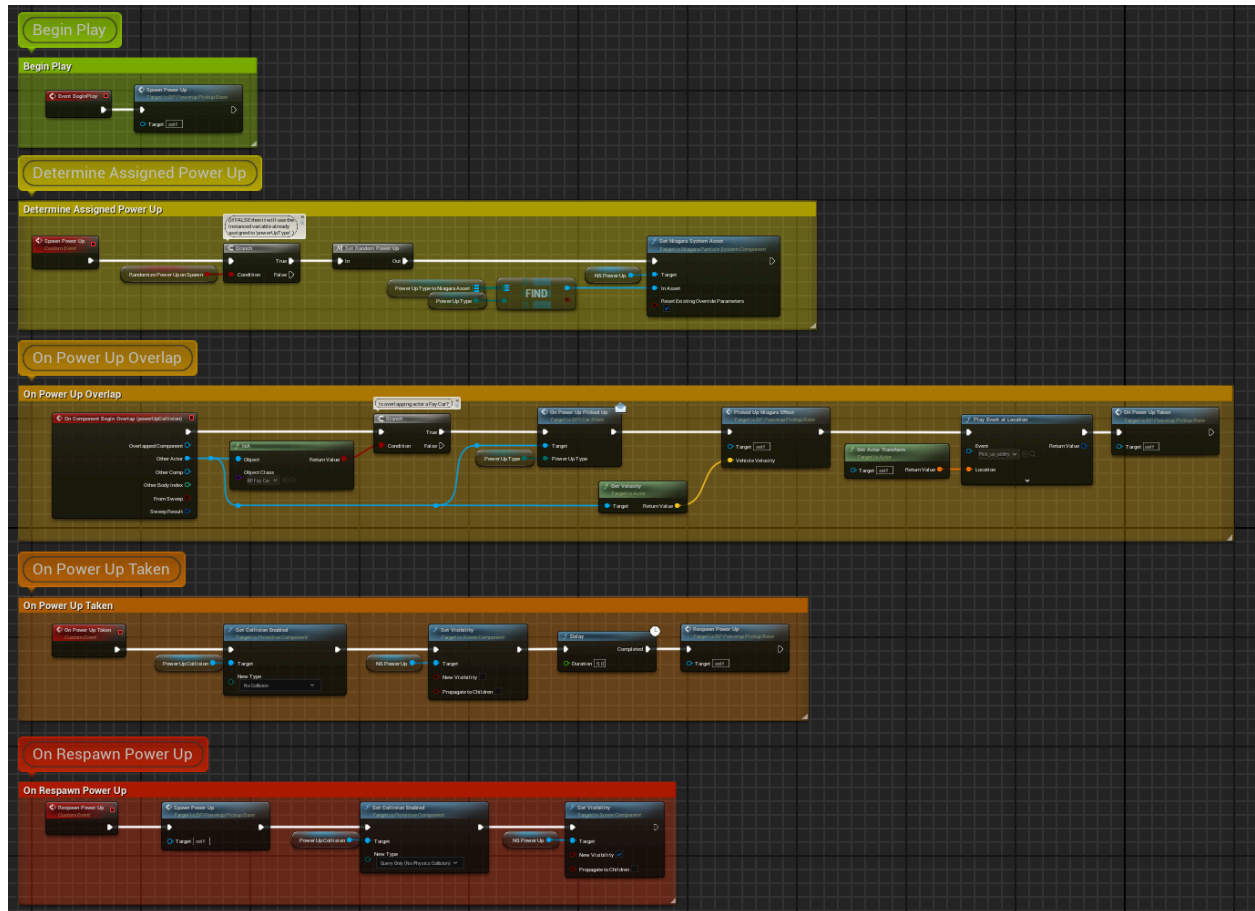


With this selected, you should notice a list of effects under the 'Details' panel on the right side of the screen under its 'Default Value'.



From here, you can use the right side of the table to swap out the VFX that corresponds to the left side power up listing.

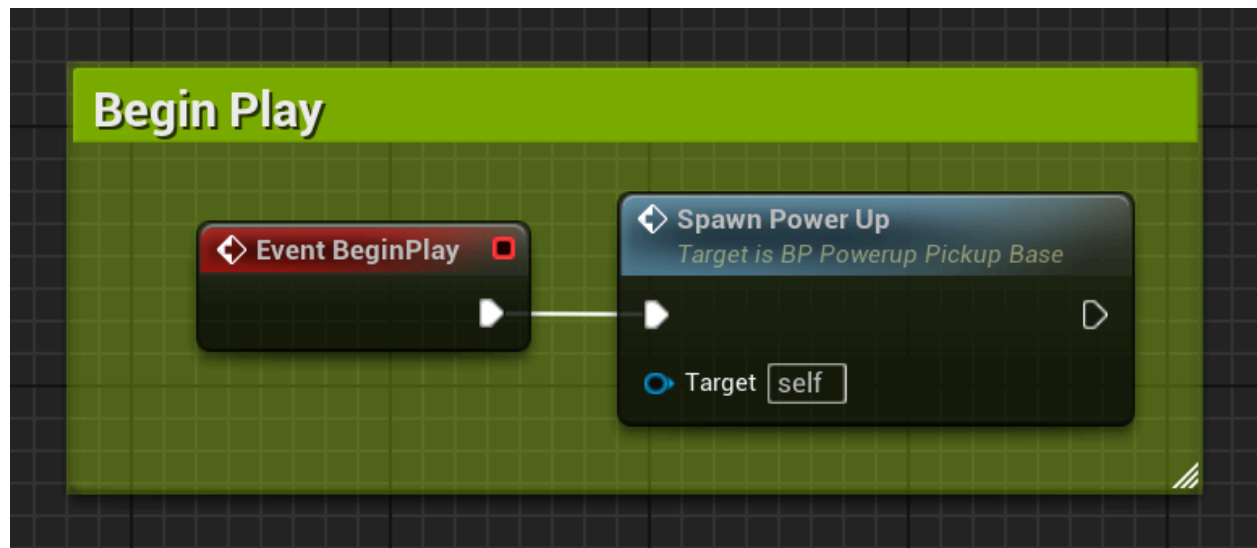
Event Graph



The event graph currently houses 5 distinct sections for the blueprints operation:

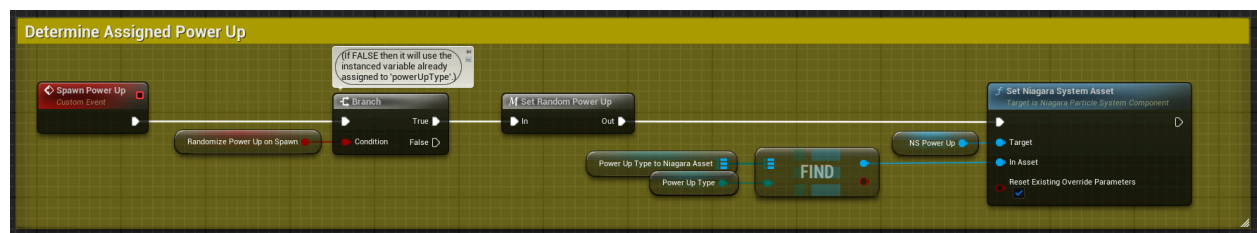
- Begin Play
- Determined Assigned Power Up
- On Power Up Overlap
- On Power Up Taken
- On Respawn Power Up

Begin Play



On 'Begin Play' the 'Spawn Power Up' function is run, which leads to the 'Determine Assigned Power Up' section.

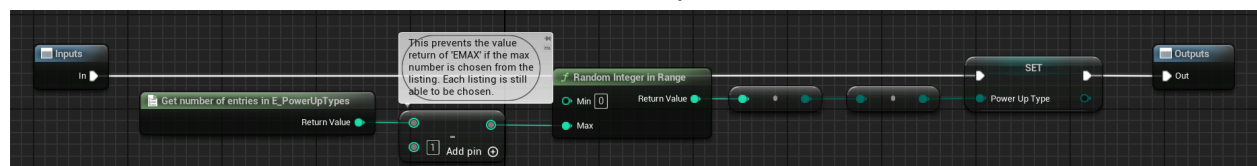
Determine Assigned Power Up



When the 'Spawn Power Up' function is run, it will immediately check if 'Randomize Power Up on Spawn' is ticked.

If FALSE then the event will end, and it will stick to the pre-assigned 'powerUpType' the instance has already been assigned.

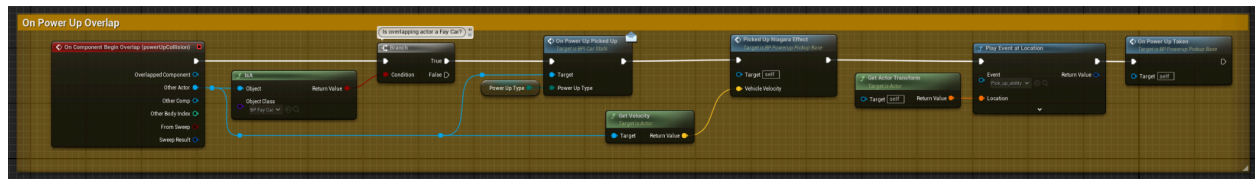
If TRUE, then the macro 'Set Random Power Up' will be run.



(The macro gets the total number of power ups, gets a random number between 1 and that total, then corresponds that number to the 'PowerUpTypes' list and sets 'powerUpType')

Once 'Set Random Power Up' is finished then, just like in the Construction Script, the niagara system VFX is assigned by the current 'powerUpType'.

On Power Up Overlap

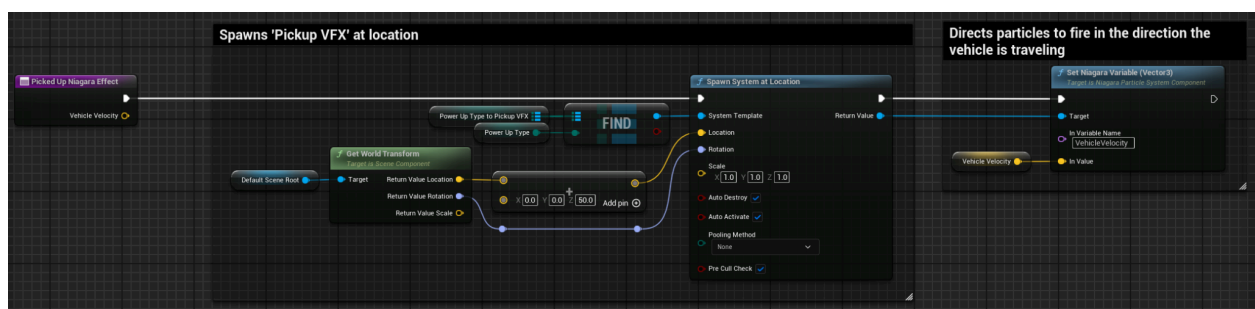


When the collision box 'powerUpCollision' is overlapped, it will first check if that overlapping actor is (or is a descendent of) BP_FayCar.

If FALSE then it will end the event.

If TRUE then will send a message to that actor's BPI_CarStats and run their 'On Power Up Picked Up' interface event. Following this, the BP_PowerupPickup_Base will activate its 'Picked Up Niagara Effect', play the PickupAbility event, and finally run the 'On Power Up Taken' event.

Picked up Niagara Effect (Function)



This function serves to set the spawn location of the VFX to be played when a power up is picked up, following this, the VFX system is supplied with the vehicle's velocity to fire its particles in the direction of travel.

Similar to section 'Editing The VFX' found in the 'Construction Script' section listed previously, to change the 'Picked Up' VFX, you repeat the previous steps but first select the 'powerUpTypeToPickupVFX' instead of 'powerUpTypeToldleVFX'.

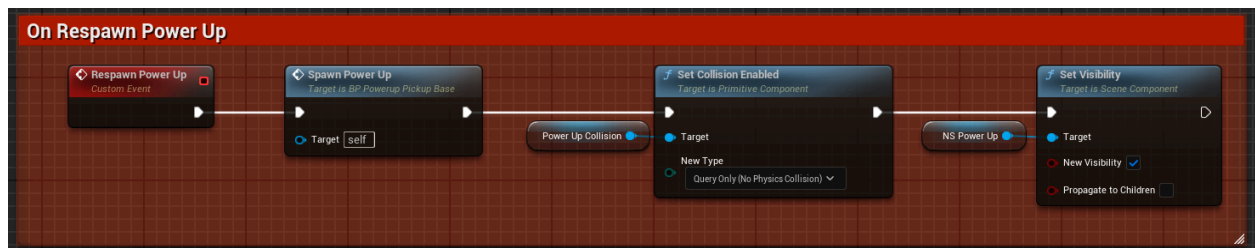
On Power Up Taken



Upon this event activating, the powerUpCollision will be immediately set to 'No Collision', and also set the NS_Power_Up VFX visibility to FALSE. This prevents the overlap collision from being activated again, and prevents the player from being able to see an already acquired power up.

Lastly, there is a delay of 5.0 seconds, this delay is the amount of time before a new power up will be respawned at that point. As after the delay, the 'Respawn Power Up' function will be run.

On Respawn Power Up



Upon activation, this event will immediately call the 'Spawn Power Up' function, followed by setting powerUpCollision back to 'Query Only', as well as its visibility to TRUE.

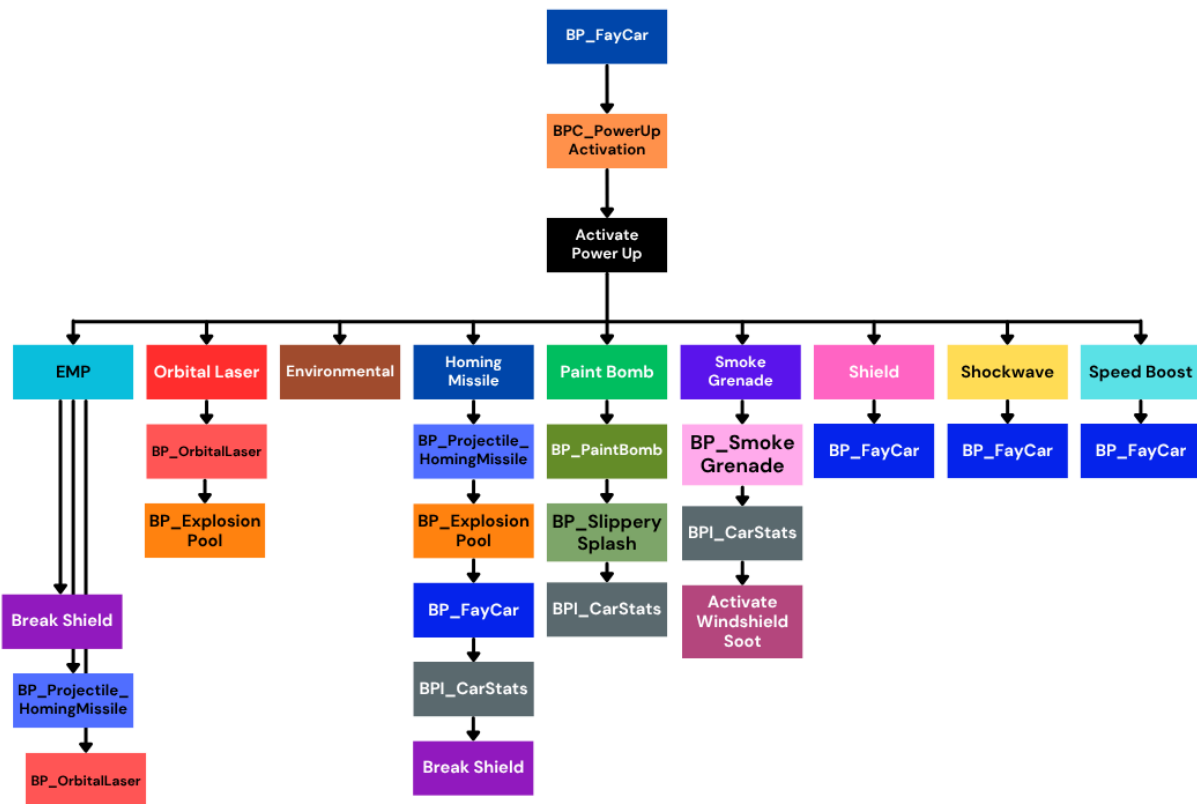
The purpose of this being its own separate event and not on the back end of 'On Power Up Taken', is to allow for potential re-use for re-rolling the power up (if randomizePowerUpOnSpawn is TRUE), or a future effect which alters the power ups within the level.

BPC_PowerUpActivation

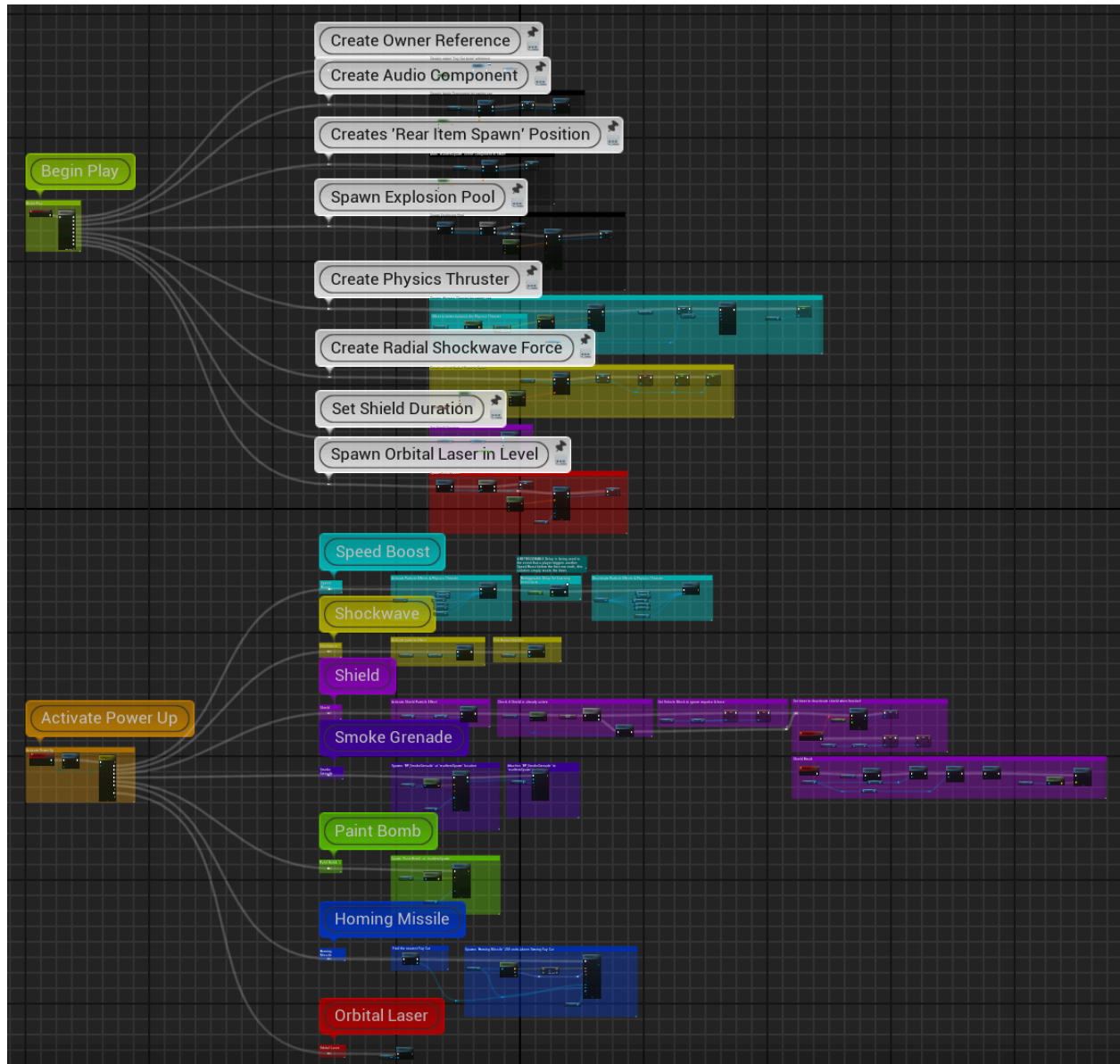
The purpose of this blueprint component is to allow its power ups to be spawned and utilized by its owning actor.

Flow Chart

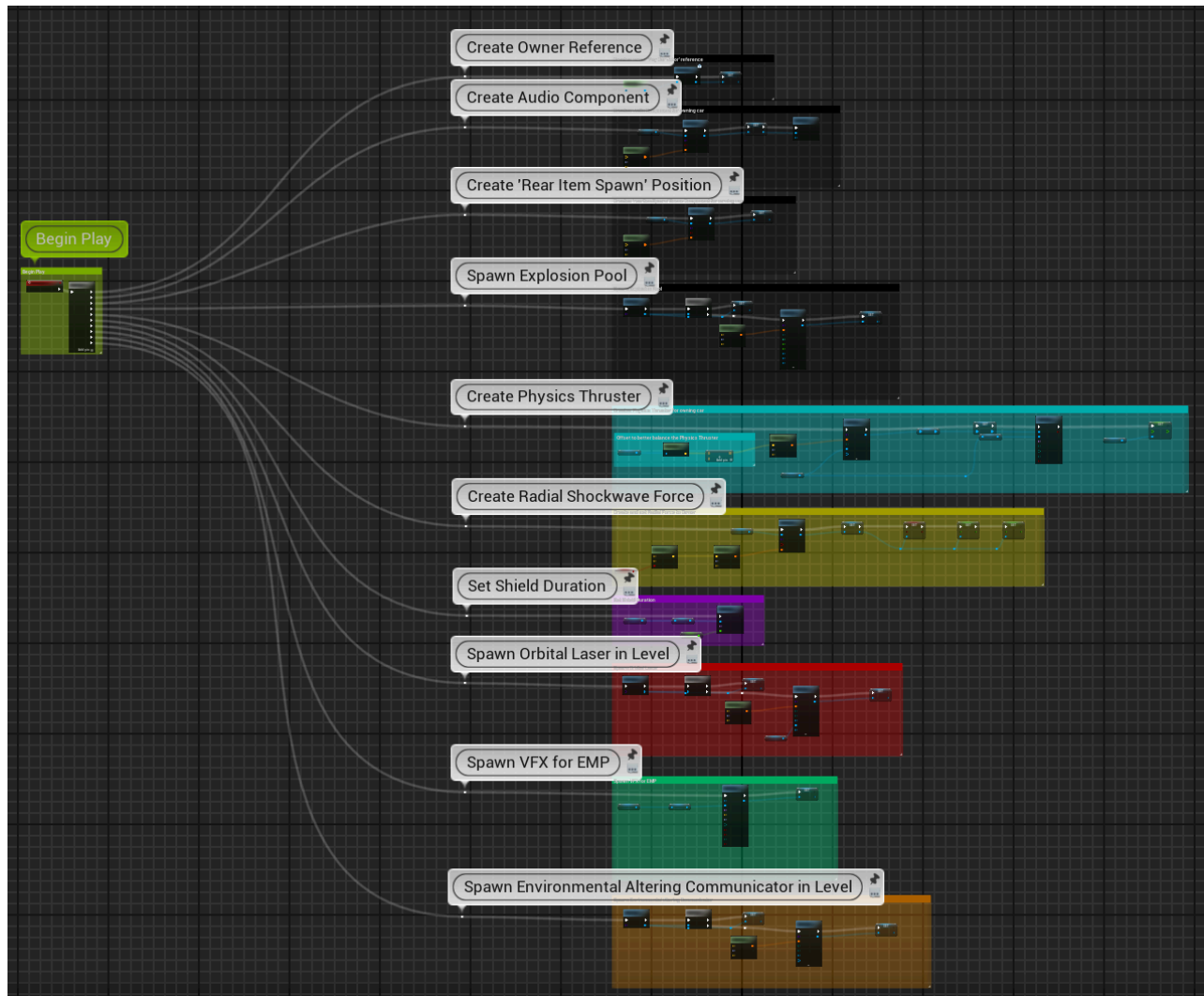
Power Up Hierarchy.



Event Graph



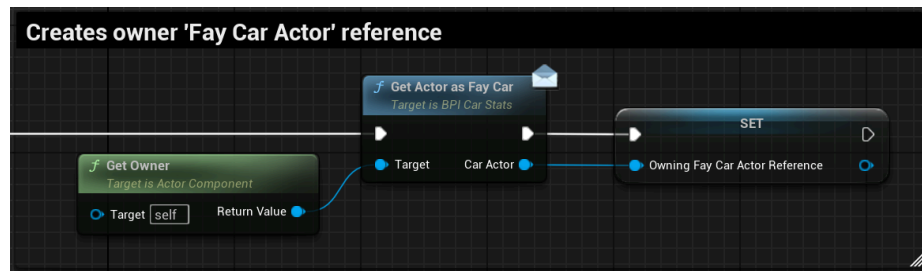
Begin Play



On 'Begin Play' there is currently a sequence outputting multiple paths.

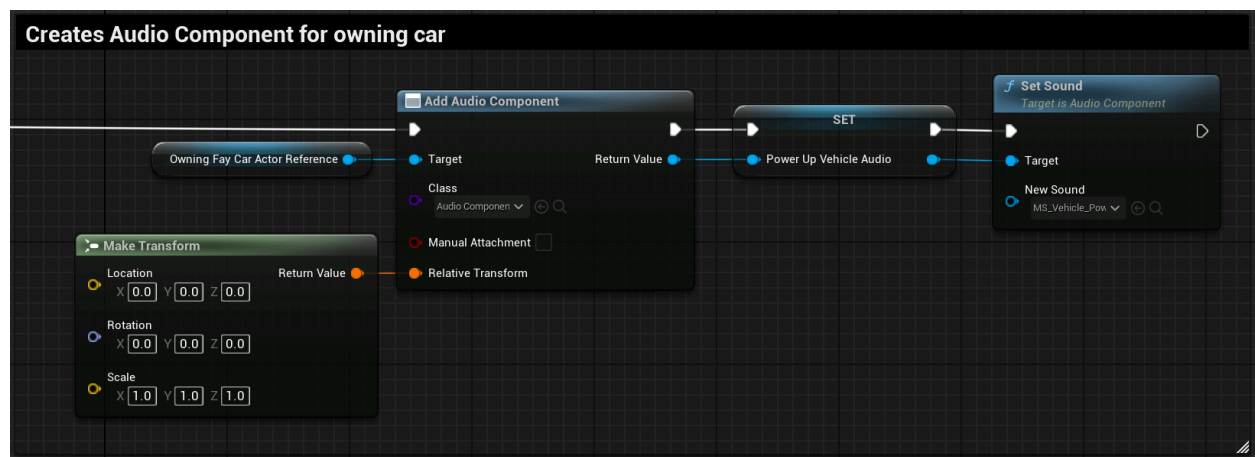
- Create Owner Reference
- Create Audio Component
- Create 'Rear Item Spawn' Position
- Spawn Explosion Pool
- Create Physics Thruster
- Create Radial Shockwave Force
- Set Shield Duration
- Spawn Orbital Laser in Level
- Spawn VFX for EMP
- Spawn Environmental Altering Communicator in Level

Create Owner Reference



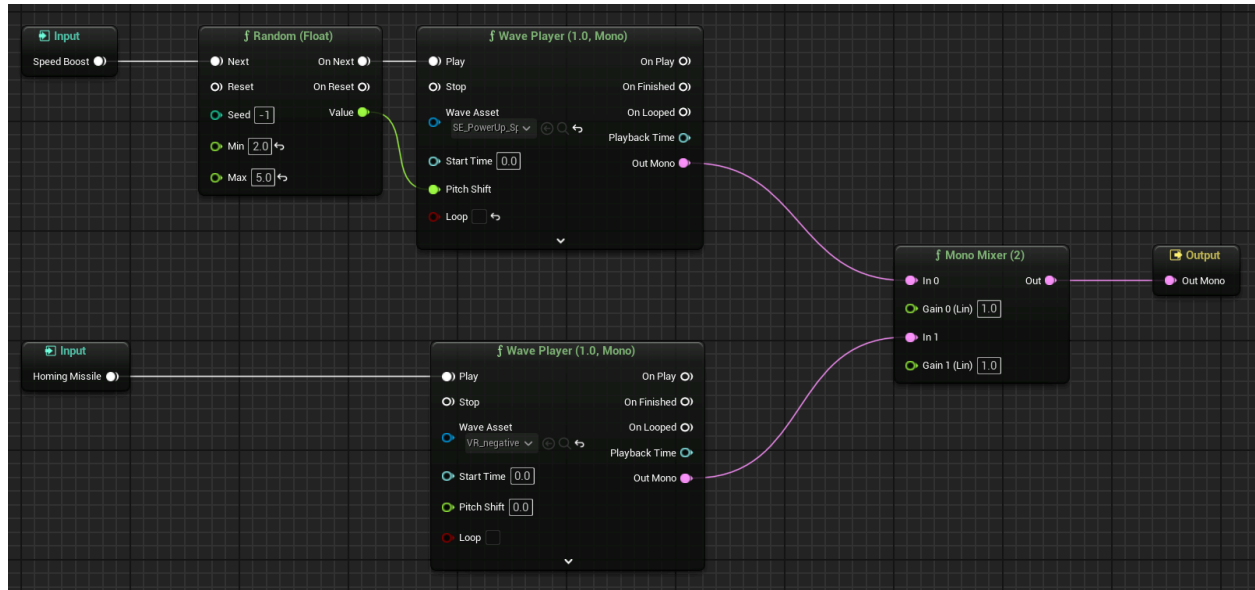
On 'Create Owner Reference' a message is sent to the 'BPC_PowerUpActivation' owner, and utilizes the 'BPI_CarStats' to get a reference to the owners 'BP_FayCar' instance. The variable 'owningFayCarActorReference' helps get information that will be needed in other elements of this blueprint component.

Create Audio Component



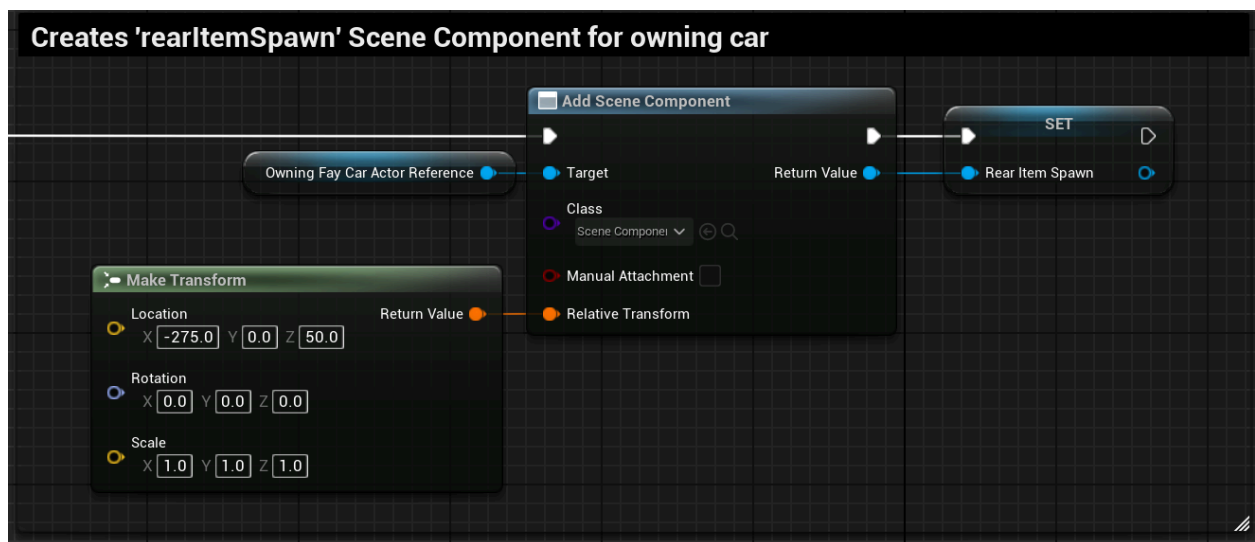
On 'Create Audio Component' the owner (owningFayCarActorReference) has a new audio component created for it, this audio component will be the basis for activating audio cues upon power up initiation. The sound system being assigned to this audio component is 'MS_Vehicle_PowerUp'.

MS_Vehicle_PowerUp



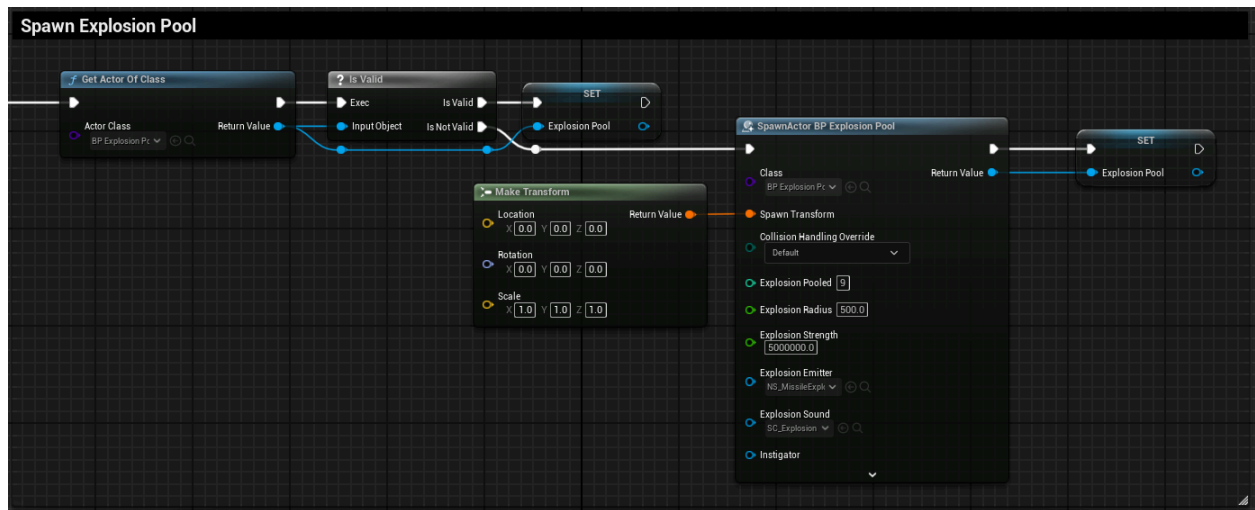
'MS_Vehicle_PowerUp' is a 'Meta Sound', utilizing a meta sound over a sound cue allows for much more control for blending the sounds each power up produces through the shared audio component.

Create 'Rear Item Spawn' Position



On 'Create Rear Item Spawn Position' a scene component is added to the owner, this scene component serves as the dedicated spawning location for power ups that are to be deployed behind the car.

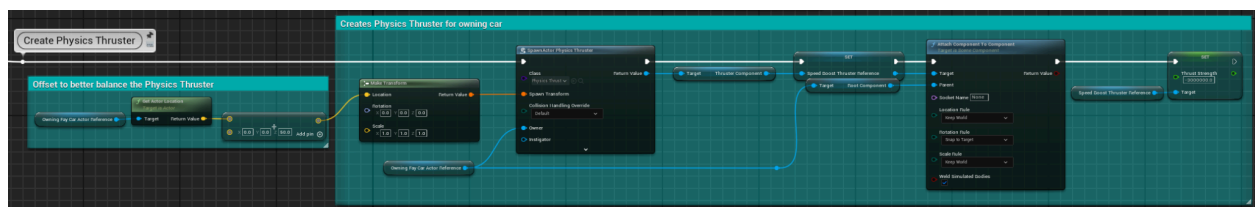
Spawn Explosion Pool



On 'Spawn Explosion Pool' the component will get the first instance of 'BP_ExplosionPool' found in the level, if one is found/valid, then the 'explosionPool' reference will be set to it. But if one isn't found, then a 'BP_ExplosionPool' will be spawned in the level at world 0.

Essentially how this works is, every instance of BPC_PowerUpActivation will be run at Begin Play for every actor that contains it, which means every actor will 'Get Actor of Class' for the 'BP_ExplosionPool', *however* only the first instance will spawn the actual explosion pool, while the rest just call the reference found in the level and assign themselves to it. This is to ensure that a 'BP_ExplosionPool' will always be spawned in the level without having to be pre-placed.

Create Physics Thruster

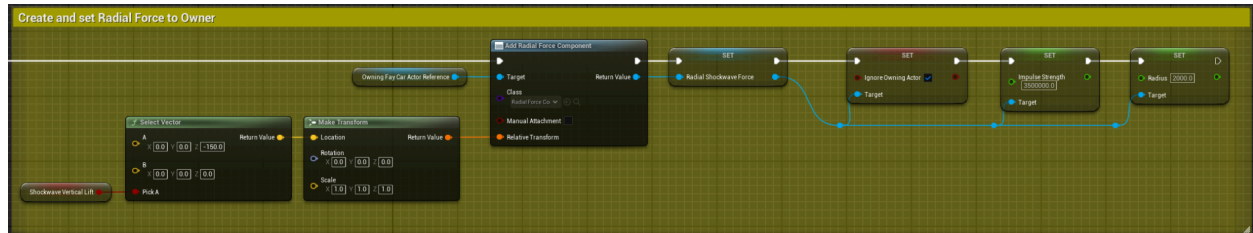


On 'Create PhysicsThruster' a physics thruster is spawned, and attached to the owner before finally having the 'Thrust Strength' set to -3,000,000. For the vehicle to be propelled *forward*, the 'Thrust Strength' **MUST** be a negative number.

Depending on the physics thrusters location on the vehicle, it can affect how much it tilts back and forth, thus is the reason for the offset +50 to the relative z axis. If there was no

offset, then the car would tilt up slightly during boost, which can cause the user to lose control.

Create Radial Shockwave Force

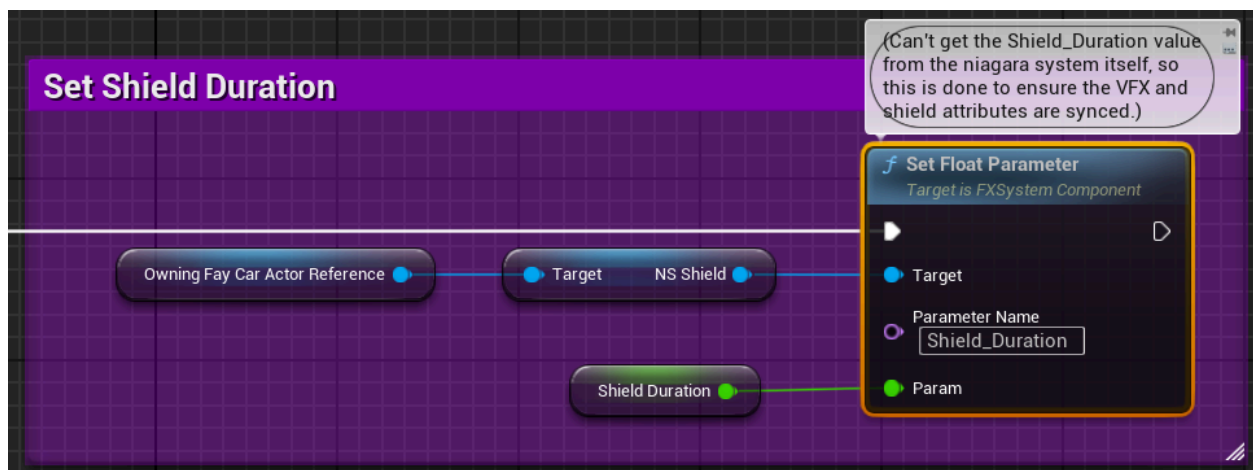


On 'Create Radial Shockwave Force' a radial force component is spawned on the owning FayCar. The relative transform contains a -150 offset to the Z axis, the purpose of this is to have the radial force originate from a lower point. By doing this, upon activation, surrounding affected objects will be not only propelled away from the origin, but they will also have force pushing them upward. If a vertical lift is not wanted, then the bool 'shockwaveVerticalLift' can be set to FALSE to *only* produce horizontal force.

Following this we set the 'Ignore Owing Actor' bool of the 'Radial Shockwave Force' to true. This prevents the force from having any impact on the owner, otherwise every time it were to be activated they would be jostled.

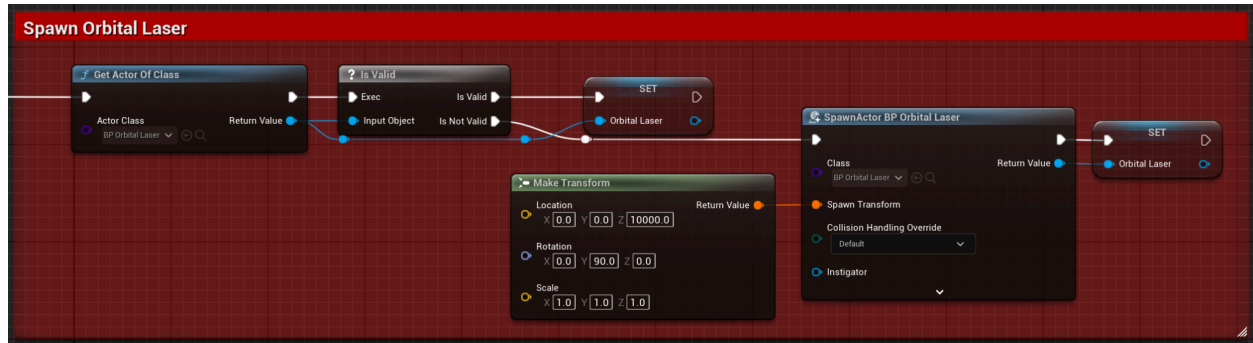
Finally the 'Impulse Strength' is set to 3,500,000 and the 'Radius' is set to 2,000. Pretty self explanatory, but the main thing to keep in mind here is that your numbers need to be *big* to have their impact felt.

Set Shield Duration



On 'Set Shield Duration' the particle effect for the owner's 'NS_Shield' has its duration set to the float value found in 'shieldDuration' (default is 10.0). Since we can't receive the value of the niagara effect, we must set it here, primarily so we can sync up its later interactions appropriately.

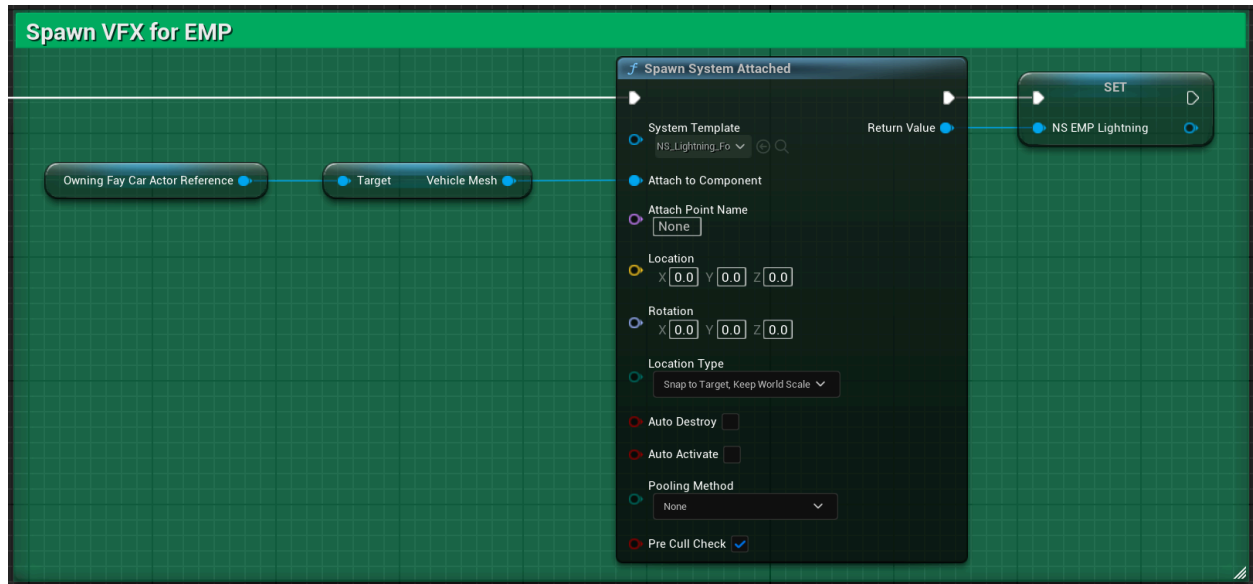
Spawn Orbital Laser in Level



On 'Spawn Orbital Laser in Level' the component will get the first instance of 'BP_OrbitalLaser' found in the level, if one is found/valid, then the 'orbitalLaser' reference will be set to it. But if one isn't found, then a 'BP_OrbitalLaser' will be spawned in the level at world 0 with a +10,000 offset to the Z axis and a +90 offset to the Y rotation to make it face down.

Essentially how this works is, every instance of BPC_PowerUpActivation will be run at Begin Play for every actor that contains it, which means every actor will 'Get Actor of Class' for the 'BP_OrbitalLaser', *however* only the first instance will spawn the actual laser, while the rest just call the reference found in the level and assign themselves to it. This is to ensure that a 'BP_OrbitalLaser' will always be spawned in the level without having to be pre-placed.

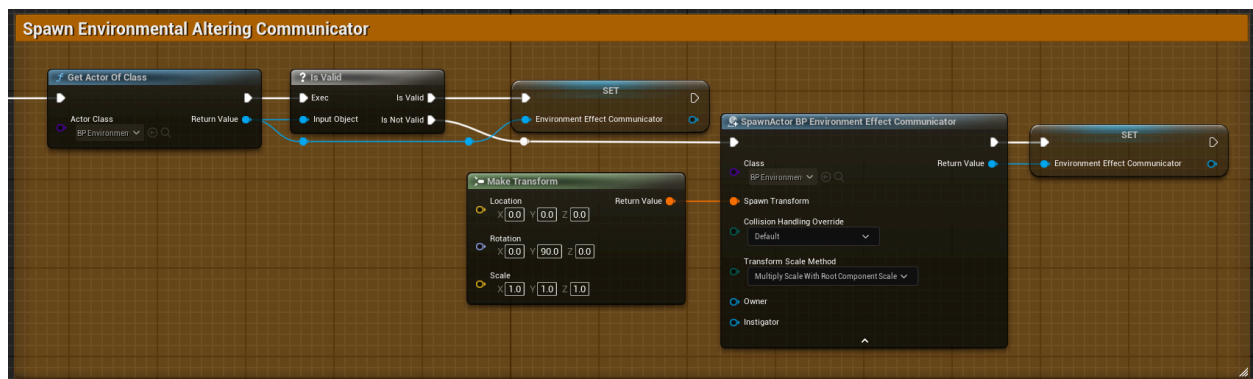
Spawn VFX for EMP



On 'Spawn VFX for EMP', the niagara system 'NS_Lightning_ForStaticMesh' is added and attached to the owner, with a reference to it also being set for later. This is specifically the VFX for a car that is under the effects of an EMP.

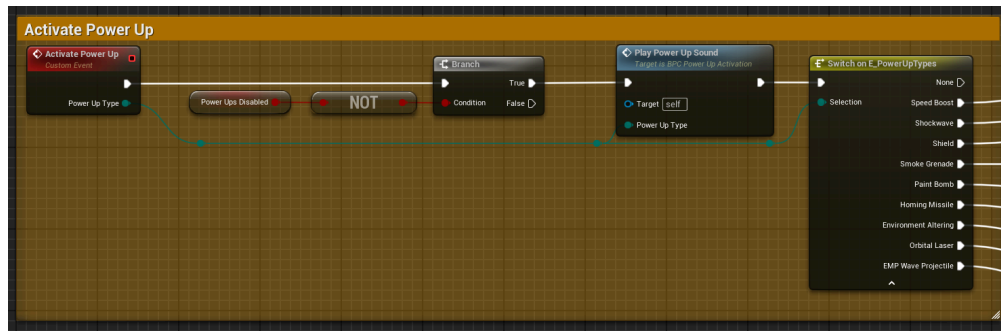
Also, within the 'EMP' folder there is a 'NS_Lightning_ForSkeletalMesh' as well, consider using this if the FayCars are ever changed to a skeletal.

Spawn Environmental Altering Communicator in Level



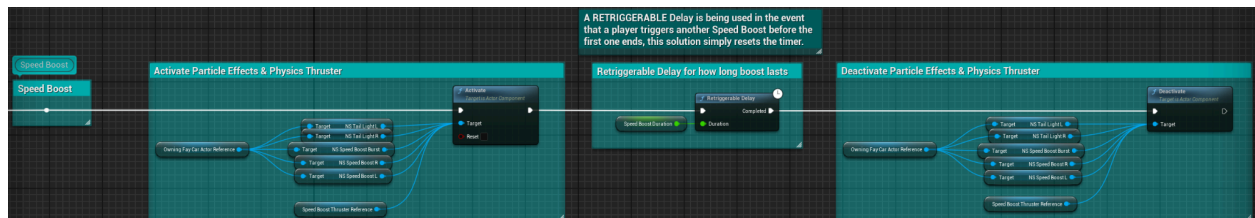
On 'Spawn Environmental Altering Communicator in Level' the component will get the first instance of 'BP_EnvironmentAlteringCommunicator' found in the level, if one is found/valid, then the 'environmentAlteringCommunicator' reference will be set to it. But if one isn't found, then a 'BP_EnvironmentAlteringCommunicator' will be spawned in the level at world 0. (This is the same process found in 'Spawn Orbital Laser')

Activate Power Up

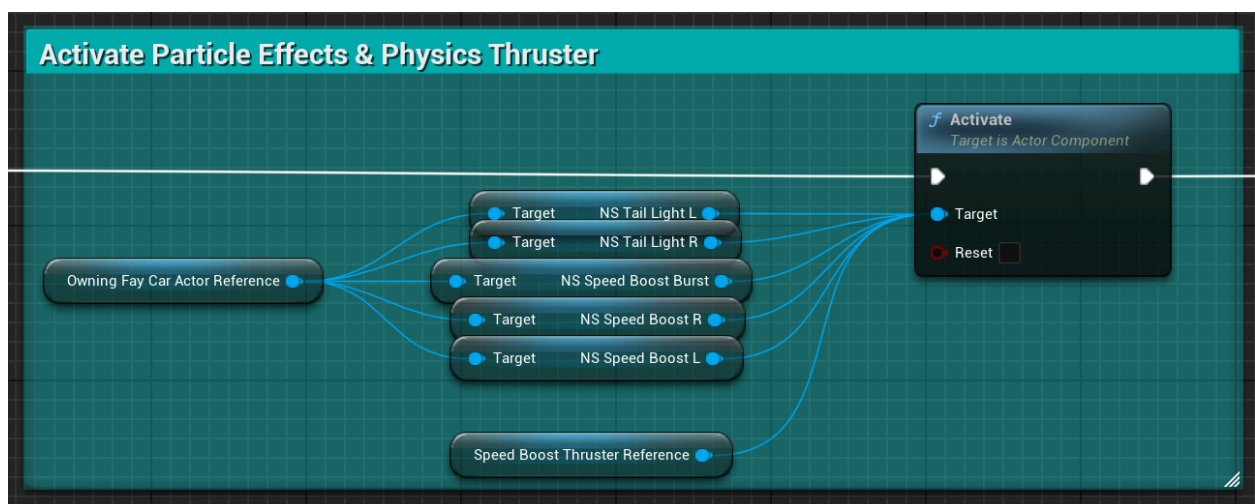


The 'Activate Power Up' event triggers the power up which has been supplied by its 'powerUpType' input on call. The event first triggers the function 'Play Power Up Sound', which triggers the corresponding power up 'Wave Asset' (sound) for the 'Meta Sound'. After this the desired power up is triggered through the 'Switch on E_PowerUpTypes' node.

Speed Boost

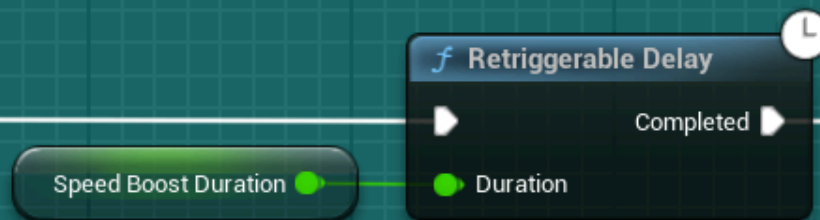


Upon 'Speed Boost' power up activation, the 'Speed Boost' VFX will be activated for the owner car, as well as the physics thruster (speedBoostThrusterReference).



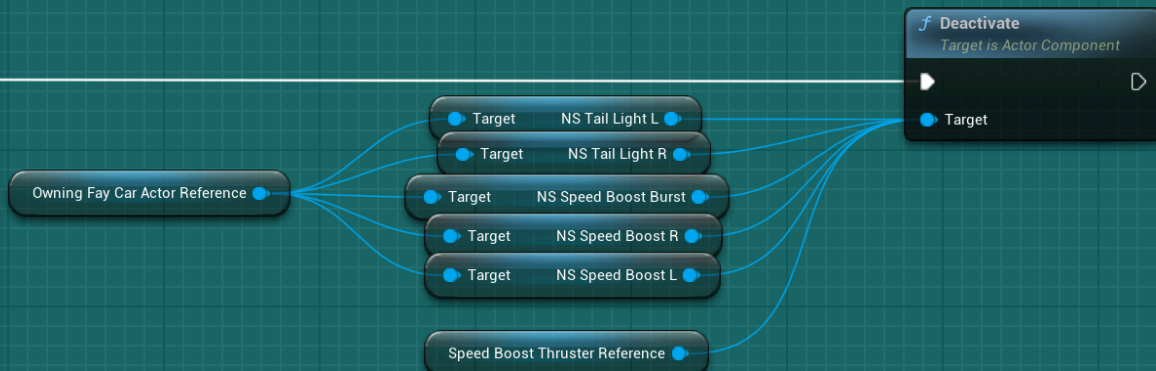
A RETRIGGERABLE Delay is being used in the event that a player triggers another Speed Boost before the first one ends, this solution simply resets the timer.

Retriggerable Delay for how long boost lasts



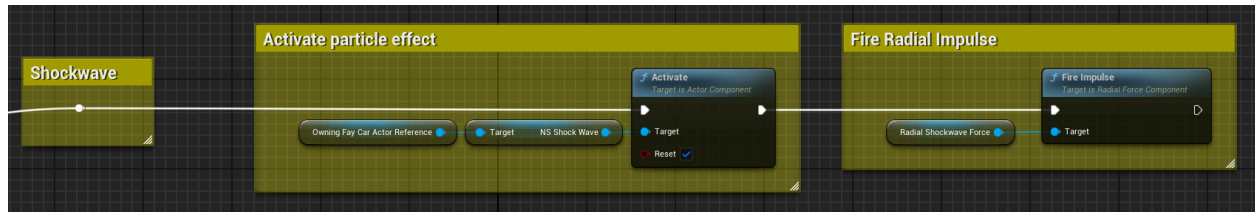
After the thruster is activated, a timer will begin with a duration corresponding to the value set for 'speedBoostDuration' (default is 3.0). The reasoning for a 'Retriggerable Delay' instead of a normal 'Delay', is in the event the player uses multiple Speed Boosts before the timer is up.

Deactivate Particle Effects & Physics Thruster



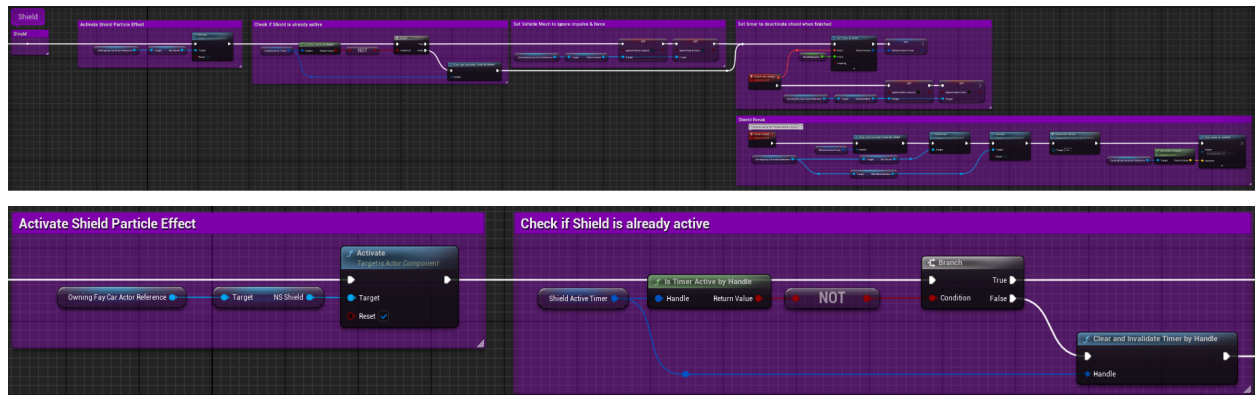
Finally, once the timer is complete, the 'Speed Boost' VFX and thruster are deactivated.

Shockwave

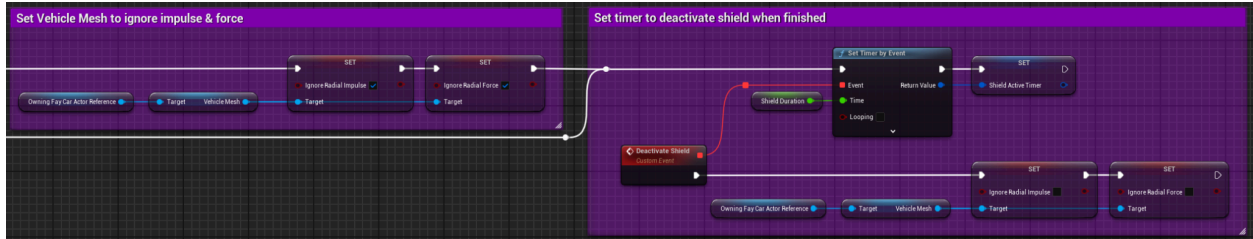


When 'Shockwave' is activated, the NS_ShockWave niagara effect will activate for the owner, followed by the previously added 'Radial Shockwave Force' firing. This will only send out a single pulse per initiation, so there's no issue with repeated calls like with the 'Speed Boost'.

Shield

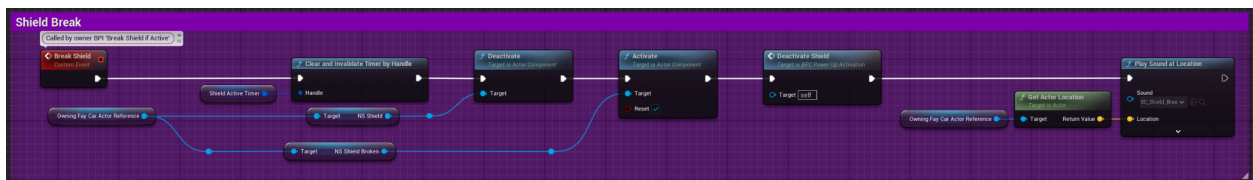


When the 'Shield' power up is activated, it first activates the 'NS_Shield' particle effect for the owner. It will then check if the 'shieldActiveTimer' is already running, clearing it if it is.



If 'shieldActiveTimer' is not already running, then the owner vehicle mesh will have both 'ignoreRadialImpulse' & 'ignoreRadialForce' ticked true, otherwise they will be skipped (because they will already be true). The theory behind this is that nearly every attack that could be initiated against a shield would involve some sort of physics force/impulse, this negates those effects so they have no impact on the owner (Note: It is theoretically possible that children meshes of the owner could be still affected, but in my testing I didn't ever notice this to be the case).

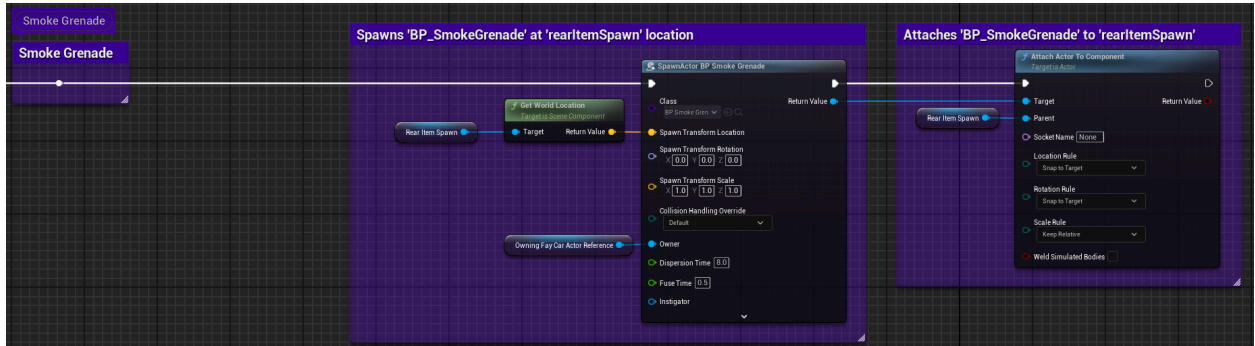
Following this a 'setTimerByEvent' node will be set with a timer that corresponds to the previously mentioned 'shieldDuration' float. Once the timer reaches the value of the float it will reset the owner mesh to no longer ignore 'Radial Impulse' & 'Radial Force'. It's important to note that this **DOES NOT** trigger the end of the NS_Shield, the particle effect scales it's effects based on the duration automatically, i.e. triggering the shield fades away when the timer reaches the end.



The 'Break Shield' event is called anytime another actor is runs the BPI event 'Break Shield if Active' which is found in 'BPI_CarStats', this messages the owner at checks if their 'shieldActiveTimer' is active, if not then the function just returns, if it is, then before it returns it runs the 'Break Shield' event for 'BPC_PowerUpActivation'.

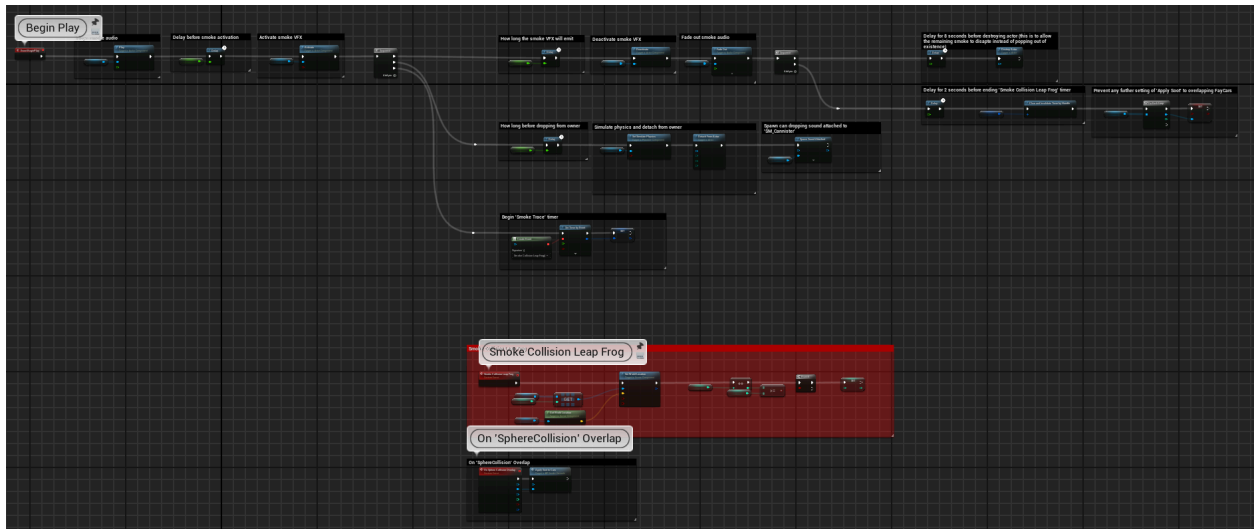
When 'Break Shield' is run, it will first clear the 'shieldActiveTimer' and deactivate the 'NS_Shield' of the owner. Following this it will trigger the 'NS_ShieldBroken' particle effect before running the 'Deactivate Shield' event, which will reset the impulse and force influence on the owner mesh. Finally it will play a sound at the location for the shield breaking.

Smoke Grenade

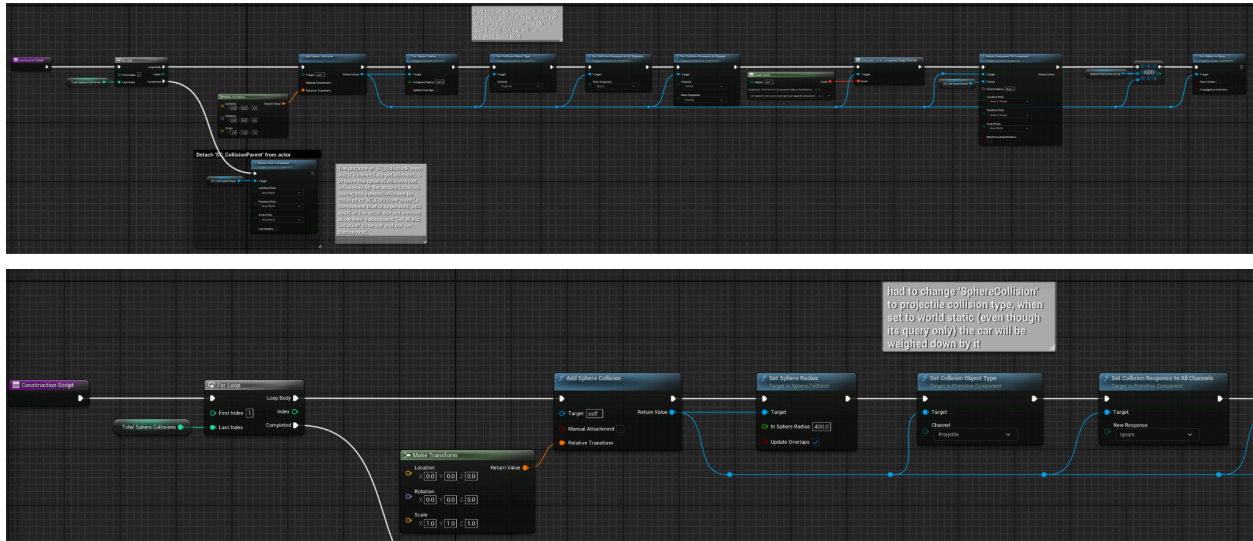


When the 'Smoke Grenade' power up is activated, a 'BP_SmokeGrenade' will be spawned and attached to the owner's 'rearItemSpawn' component.

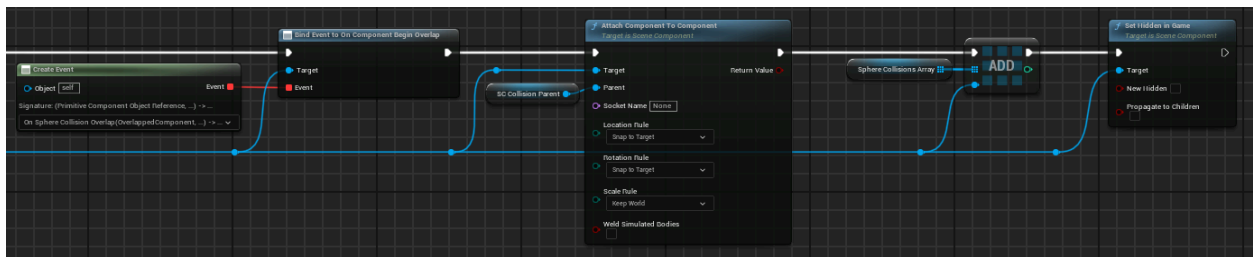
BP_SmokeGrenade



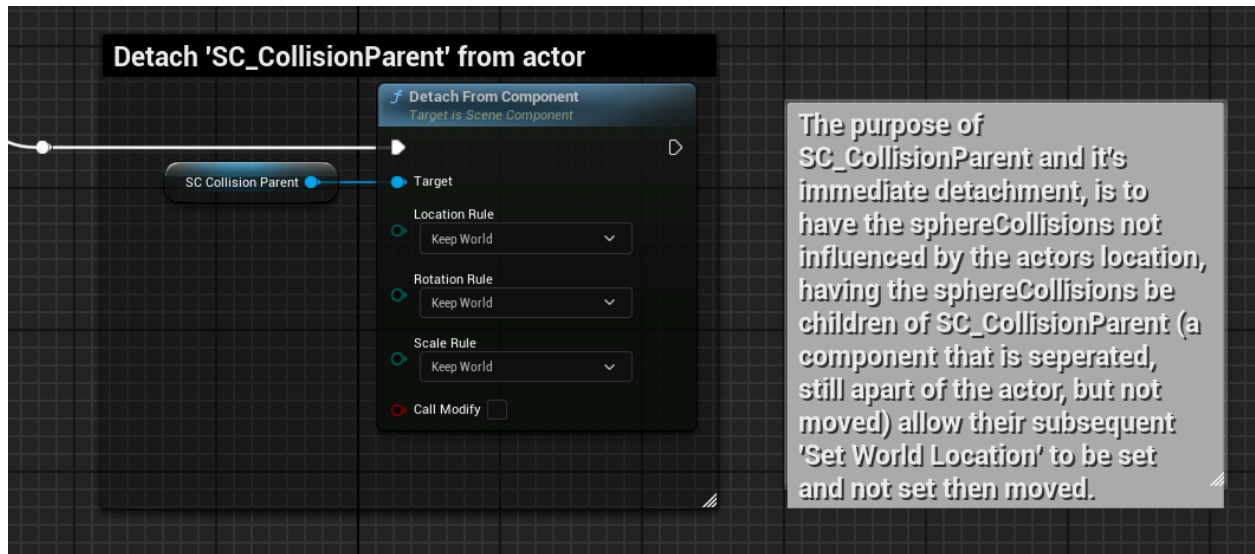
Construction Script



When the 'Construction Script' is run, a 'For Loop' begins for a number of times equal to the int value set for 'totalSphereCollisions'. For each loop an 'Add Sphere Collision' will be run, then pulling off of its 'Return Value' the sphere collision will have its sphere radius set to 400, have its collision object type set to 'Projectile' collision (this is to prevent an issue where the FayCar blocks 'WorldStatic' even if the sphere collision is set to ignore 'Vehicle'), next the sphere collision has its collision response for all channels set to 'Ignore', and then has the collision response channel of 'Vehicle' set to 'Overlap'.

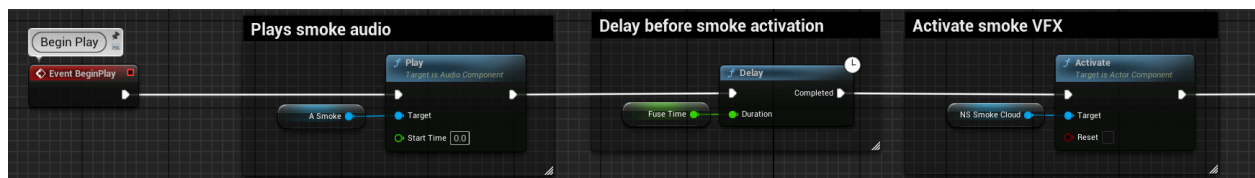


The sphere collision then has its 'On Component Begin Overlap' bound to a mutual event of 'On Sphere Collision Overlap' (located in the event graph). Next a 'Attach Component to Component' node is run to attach the sphere collision to the scene component 'SC Collision Parent' and is added to the 'sphereCollisionsArray'.

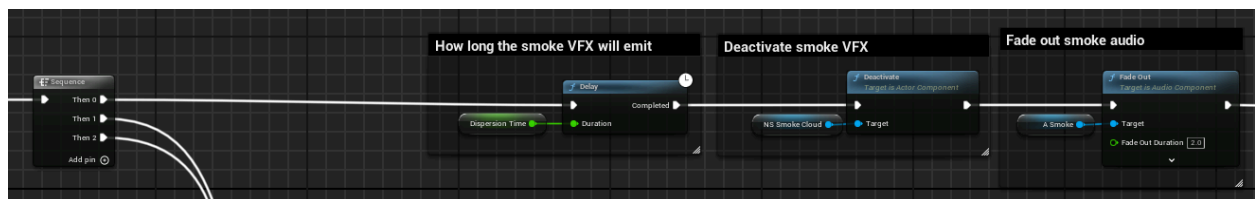


Once the 'For Loop' is completed, a 'Detach From Component' node is run for the 'SC Collision Parent'. This is primarily to prevent the children's sphere collisions from being influenced by the movement of the smoke grenade, and allows them to have their 'Set World Location' definitively placed.

Begin Play

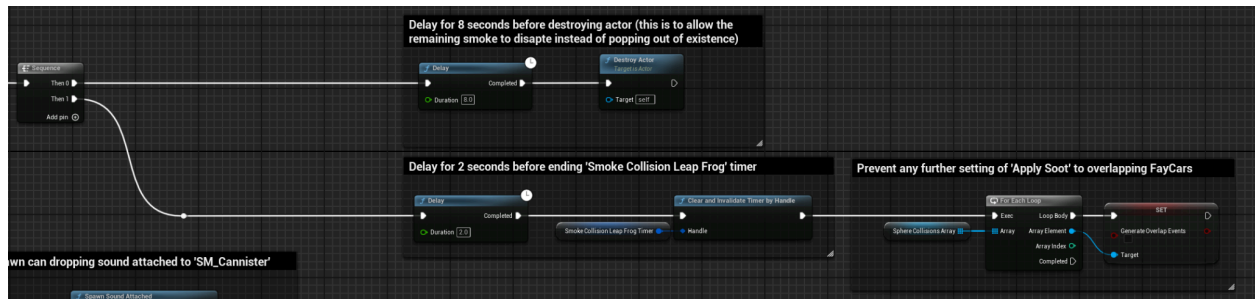


On 'Begin Play' the 'A_Smoke' audio is played, this sound effect is essentially “the pin being pulled”, signaling the impending smoke distribution. After the audio there is a brief delay with a 'fuseTime' float variable for its duration (default value is 0.5), before the 'NS_SmokeCloud' VFX are activated. Finally, the 'Smoke Trace Timer' is activated.



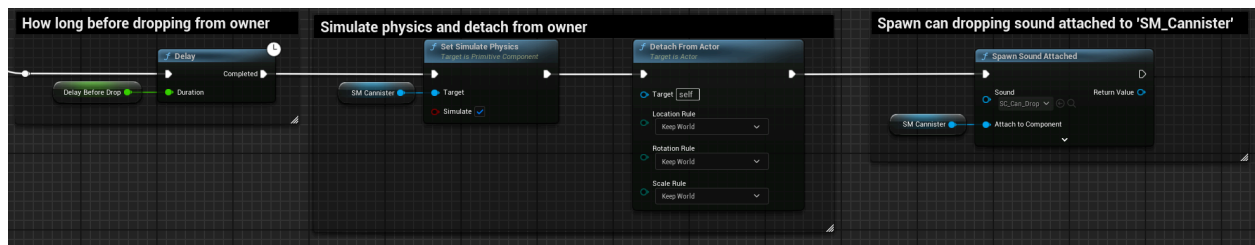
Following the first index of the sequence node, a delay is begun with the input being supplied by a 'dispersionTime' float variable (default value is 8.0), this delay dictates how long the smoke will emit for. Upon the delay nodes completion, the

'NS_SmokeCloud' VFX are deactivated, the 'A_Smoke' audio is faded out, and the 'Smoke Trace' timer is ended.

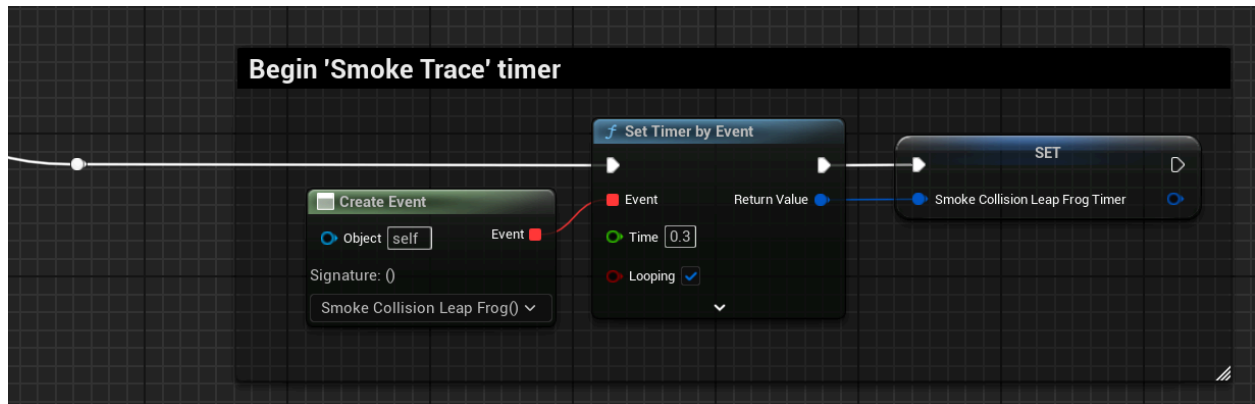


Next is the beginning of another sequence, the first index being a delay of 8.0 before the actor is destroyed. The purpose of this delay is to allow the lingering smoke created by the now deactivated 'NS_SmokeCloud' to dissipate. Without this, the smoke would prematurely pop out of existence.

Then for the next index of that second sequence, is a 2 second delay before the 'smokeCollisionLeapFrogTimer' is ended and each 'sphereCollision' in the 'sphereCollisionArray' has their 'Generate Overlap Events' bool set to false. With the timer being ended, this will end the application of 'soot' to overlapping fay cars, though the reason for setting the 'sphereCollisions' 'Generate Overlap Events' to false is to prevent any unnecessary 'overlap event' triggers.

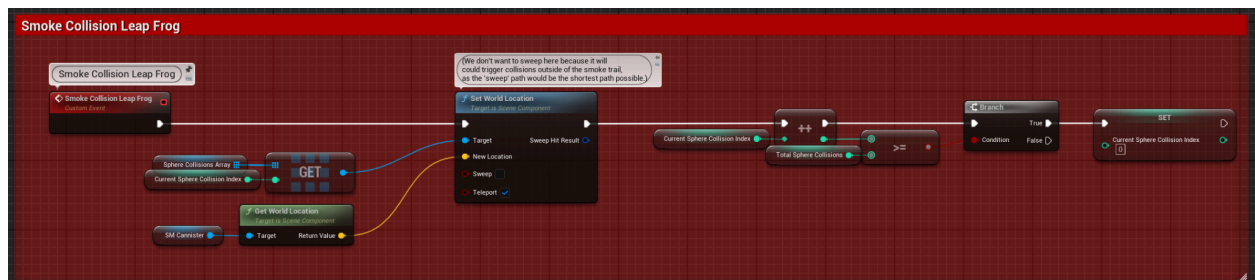


For the next index of the first sequence node there is an immediate delay with the supplied float variable of 'delayBeforeDrop' (default value is 2.0), this delay dictates how the smoke grenade stays attached to the owning FayCar. Upon the delay's completion, the 'SM_Cannister' mesh has its physics set to simulated before being detached from its actor, allowing it to roll freely in the game world. Also during this, a 'Spawn Sound Attached' node is used on 'SM_Cannister', this is a "can dropping" sound to signal its release.



The final index of the sequence begins the 'smokeCollisionLeapFrogTimer' with a looping duration of 0.3 seconds.

Smoke Collision Leap Frog

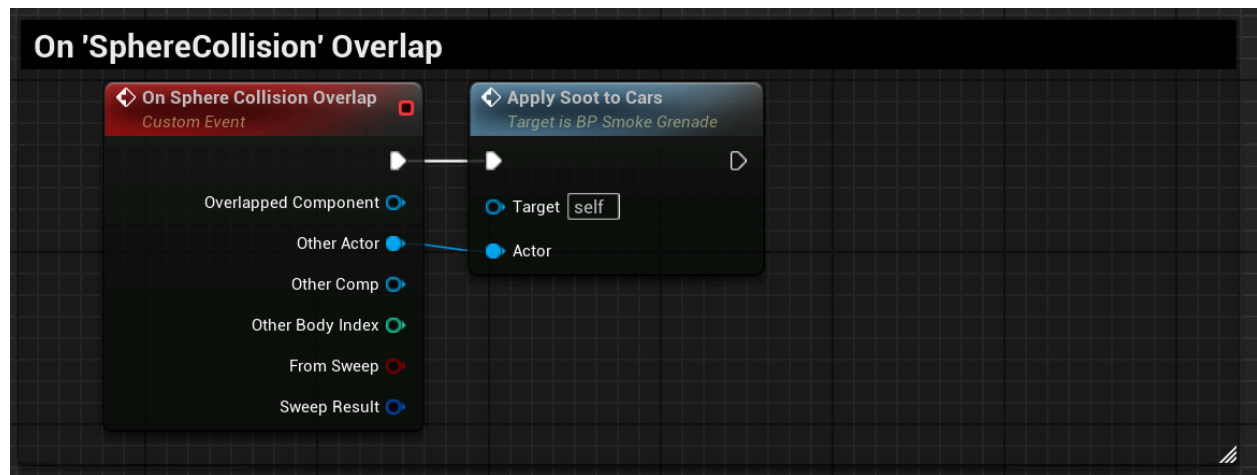


The 'Smoke Trace Leap Frog' event first begins by getting the 'currentSphereCollisionIndex' value equal to the 'sphereCollisionArray' and sets it's world location to 'SM Cannister'.

Following this the 'currentSphereCollisionIndex' is incremented and checked to see if it's value is greater than or equal to the 'totalSphereCollisions' int, if it is, then the 'currentSphereCollisionIndex' is set to 0.

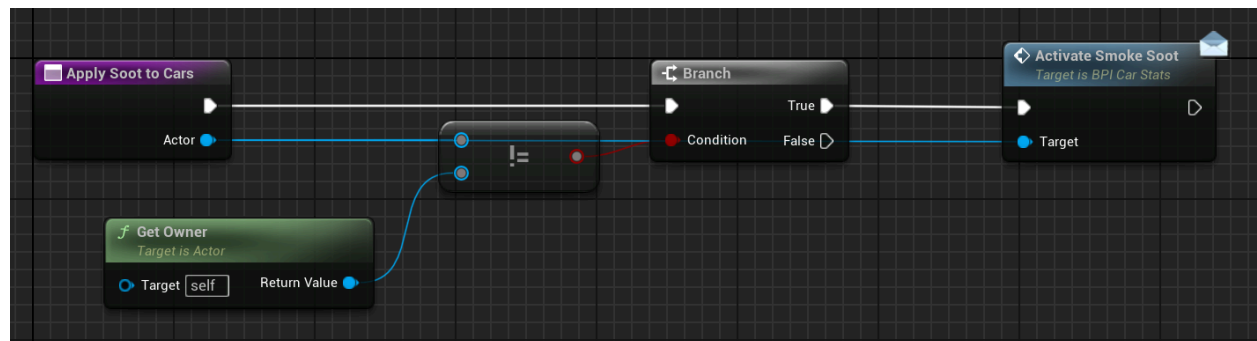
Essentially what this does is loops through each available sphere collision and moves the oldest one to the smoke grenade, hence the name 'Leap Frog'.

On 'SphereCollision' Overlap



When one of the spheres detects an overlap it runs the 'Apply Soot to Cars' function and supplies its 'Actor' input with the 'Other Actor' detected on the overlaps initiation.

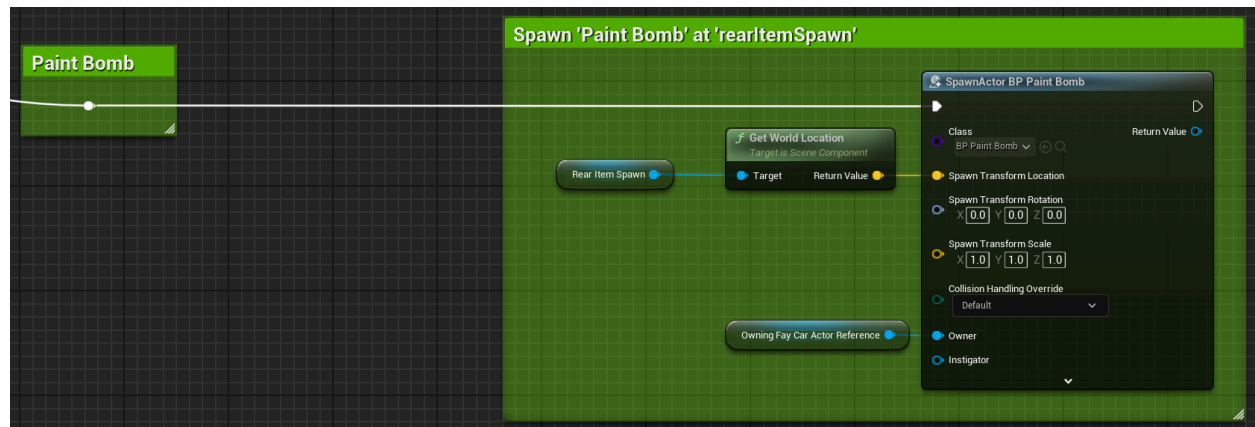
Apply Soot to Cars (Function)



On 'Apply Soot to Cars' it will first check that the actor overlapping is NOT the owner, if it's not the owner then the actor will have the 'Activate Smoke Soot' message sent to them via the 'BPI_CarStats'. This message will signal to the actor to have their screen thoroughly *sooted* (later done by UI).

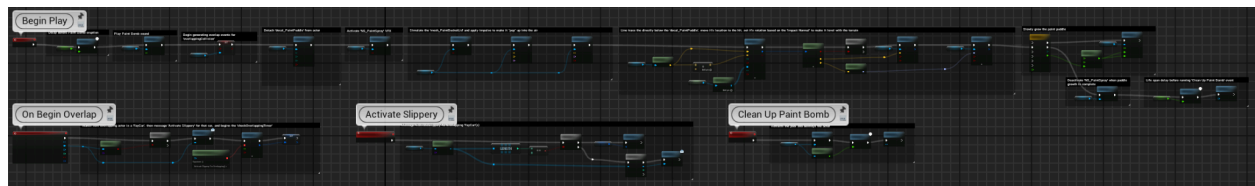
Again to reiterate, **THE FAYCAR OWNER OF THE SMOKE GRENADE DOES NOT HAVE SOOT APPLIED.**

Paint Bomb (v1 'Stationary')

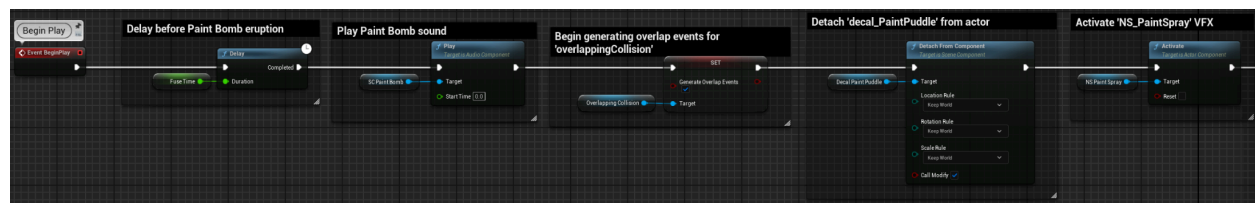


When the 'Paint Bomb' power up is activated, a 'BP_PaintBomb' will be spawned at the the owner's 'rearItemSpawn' component.

BP_PaintBomb_Stationary



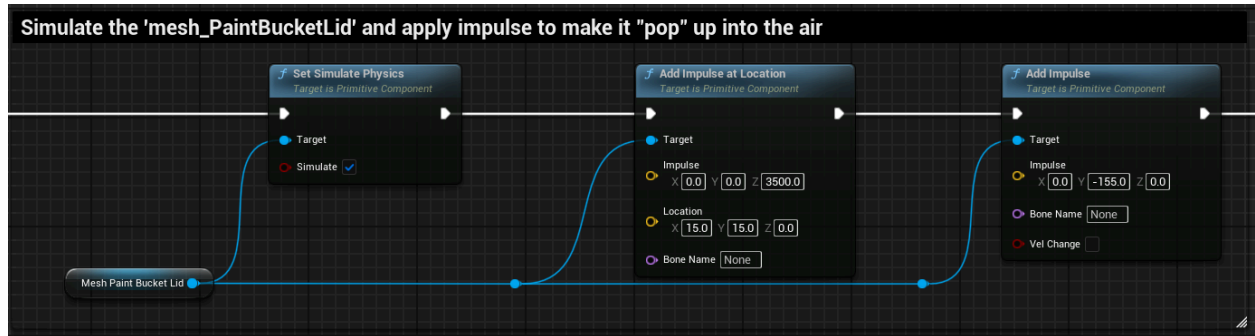
Begin Play



On 'Begin Play' there is an initial delay with the float variable 'fuseTime' imputed into its duration (default value is 1.0). Upon its completion the 'SC_PaintBomb' audio will begin playing. The 'overlappingCollision' will also have its 'generateOverlapEvents' set to TRUE (see section 'On Begin Overlap' for further context).

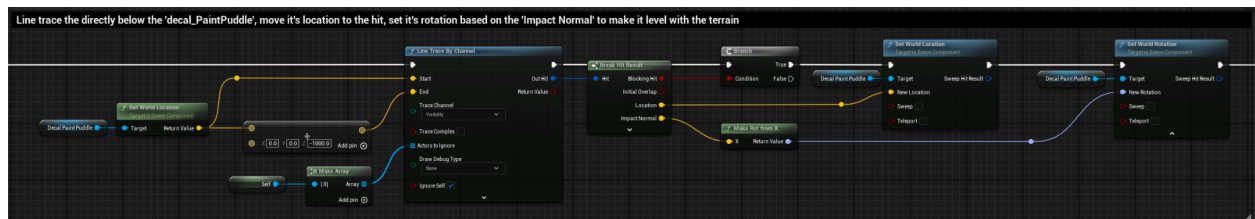
Next the 'decal_PaintPuddle' component will be detached from the actor, this is done to prevent the 'mesh_PaintBucket' movement from influencing its location going forward. Then the VFX for 'NS_PaintSpray' is activated.

Note: The current VFX of 'NS_PaintSpray' has particles which contain collisions. This works fine during one or two simultaneous instances of 'BP_PaintBomb', but can decrease performance if this amount is exceeded.



Following the VFX activation, the 'mesh_PaintBucketLid' is first simulated, and then has two separate impulses applied to it, the first propelling it into the air, while the second one causes it to flip/roll as it ascends. Essentially the lid pops into the air like a coin flip.

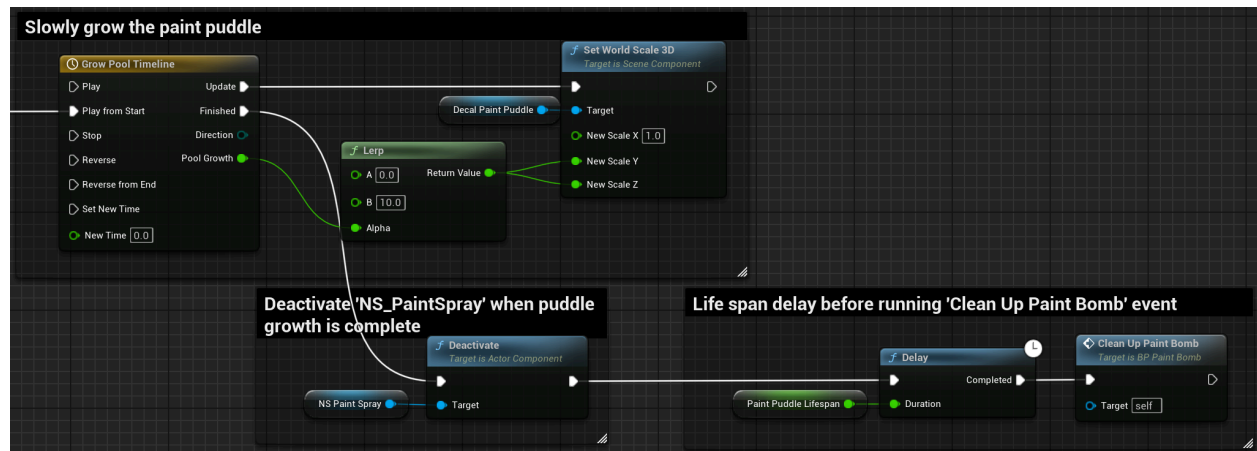
While this effect is interesting on its own, if the 'mesh_PaintBucket' manages to get flipped upside down before it activates, this "popping effect" will have enough influence to actually flip 'mesh_PaintBucket' over to land right side up.



After the "Lid Impulses" a 'Line Trace by Channel' is fired with the starting point of 'decal_PaintPuddle' current world location, and the end point being that same location but with a -1000 added to its Z axis (the purpose of adding the negative instead of subtracting is to prevent the potential scenario of subtracting a negative value, causing it to unintentionally become a greater negative rather than being reduced by the value of a positive). The hit result of this trace will give us the "surface location" to apply the decal at.

When a hit is detected, the 'decal_PaintPuddle' will first have its location moved to that location in world space, this will make the decal "flush" with the surface, negating any potential previous offsets.

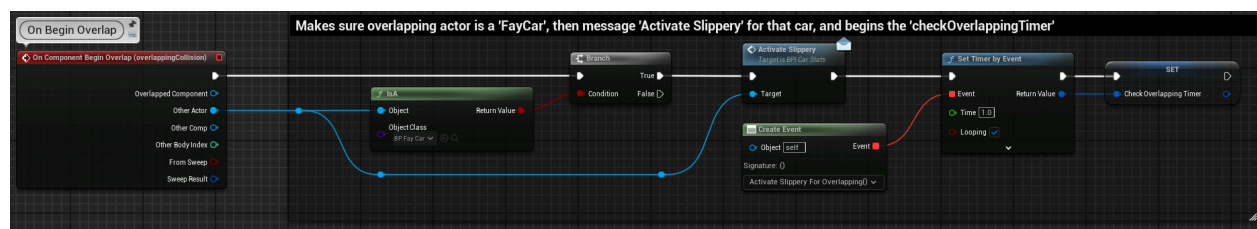
The 'decal_PaintPuddle' will further have its world rotation set based on the 'Impact Normal' of the trace. The purpose of the 'Impact Normal' into the 'Make Rot from X' is to set the 'decal_PaintPuddle' to have its rotation set relative to the surface it is being deployed on. A decal in itself can generally only be a flat cube, so when trying to apply it to a slanted surface, we're rotating it to meet the surface rotation supplied by the normals.



Finally, the timeline 'Grow Pool' is activated, the timeline sets the 'Pool Growth' float value from 0 to 1 over a duration of 5.0 seconds. Utilizing that 'Pool Growth' float, it's fed into the 'Alpha' input of a 'Lerp' node. This allows for the float value to transition relative to the curves of the 'Grow Pool Timeline' graph. As the timeline updates, it runs a 'Set World Scale 3D' node for the 'decal_PaintPuddle', changing the Y and Z values from a beginning of 0 to an ending of 10.0. The purpose of this is to make the decal grow wider as the timeline goes on, as well as widening the scale of its child 'overlappingCollision'. This is to try and keep the scale of 'decal_PaintPuddle' and 'overlappingCollision' in sync with one another.

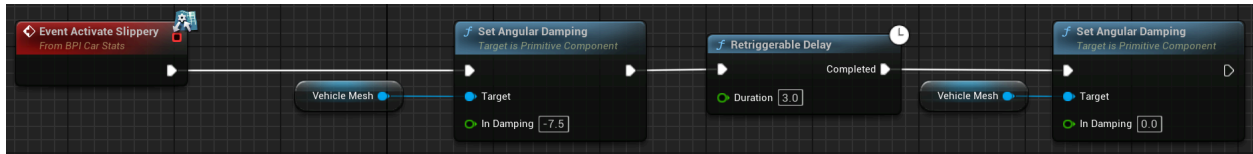
Once the timeline is finished, it first deactivates the 'NS_PaintSpray' VFX, then it enters another delay node that has its duration supplied by a 'paintPuddleLifespan' float variable (default value is 10.0), this delay dictates how long the puddle will remain in the world. Once the delay is complete, it runs the 'Clean Up Paint Bomb' event.

On Begin Overlap



'On Begin Overlap' is triggered when the 'overlappingCollision' component is *overlapped*. The component itself initially has its 'generateOverlapEvents' set to false to prevent premature slippery activations.

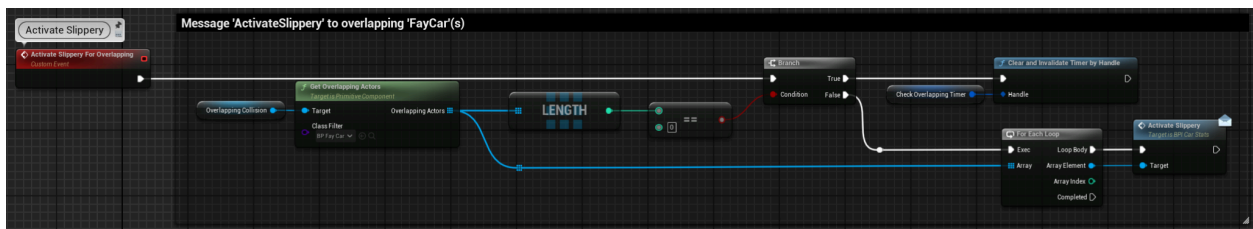
It will first check that the overlapping actor is a 'BP_FayCar', if TRUE then it will send a message of 'Activate Slippery' via the 'BPI_CarStats'. Then it will initiate a 'Set Timer by Event' node for the 'Activate Slippery For Overlapping' Event.



The interface event of 'Activate Slippery' found in 'BP_FayCar' works by setting the angular damping of the 'vehicleMesh' to a negative value. This results in turning with the vehicle to be way more sensitive than normal, emulating a loss of control. After it's set it goes into a 'Retriggerable Delay' node with a value of 3.0. The purpose of this is to allow the vehicles 'Slipperiness' to be prolonged in the event they receive the message of 'Activate Slippery' again. Once the delay is able to complete, the 'vehicleMesh' will have its angular damping set back to 0.0.

Note: If the 'vehicleMesh' ever has it's angular damping default value changed in the future, a float variable could be created that reads the 'vehicleMesh' current angular damping at begin play, then use a get of that float to set the angular damping here to the proper value.

Activate Slippery For Overlapping

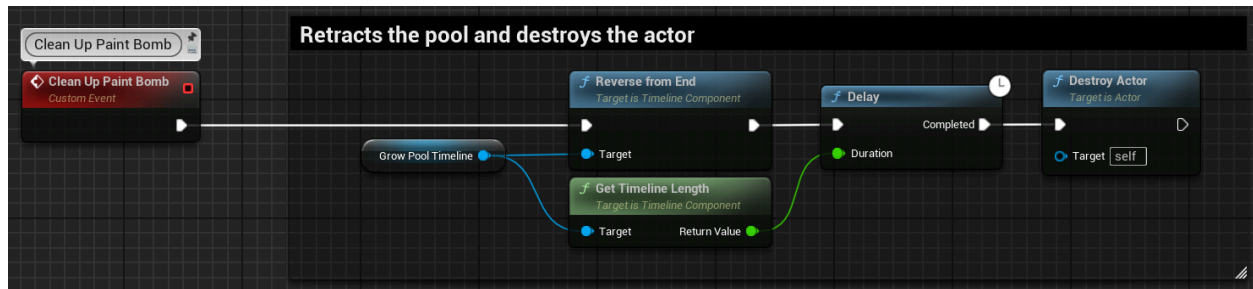


The 'Activate Slippery For Overlapping' event is initiated through the 'Set Timer by Event' node found in 'On Begin Overlap'. Per the inputs supplied to the timer, this event fires every 1.00 seconds on loop.

'Activate Slippery For Overlapping' will first get the total number of overlapping actors found within the 'overlappingCollision' component, if it finds that value is 0 then it will end the 'checkOverlappingTimer' which is responsible for looping this event. If however

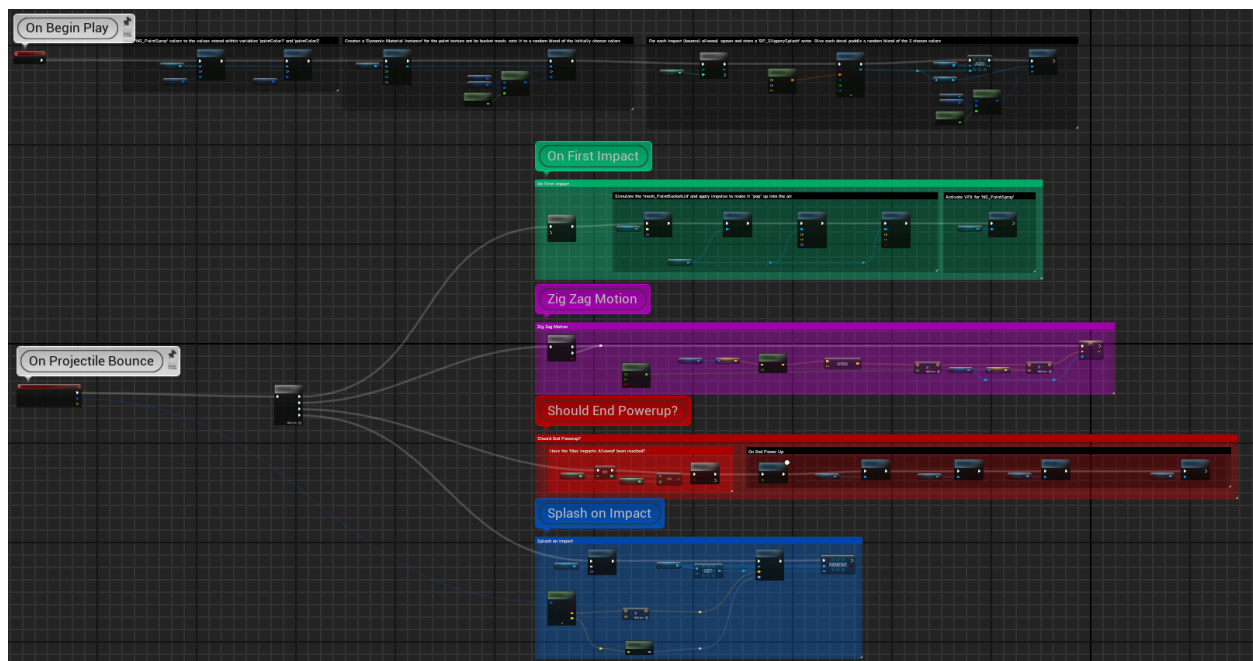
it finds the value is NOT 0, then it will run each of those overlapping actors through a 'For Each Loop', sending each of them a message of 'Activate Slippery' via 'BPI_CarStats'. In short, if there aren't any actors found within the overlap, then it's going to end the timer to save resources.

Clean Up Paint Bomb

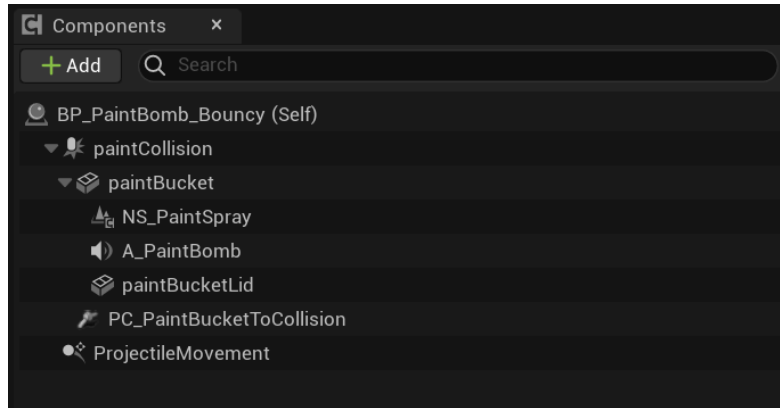


When the 'Clean Up Paint Bomb' event is run, it will play the 'Grow Pool Timeline' in reverse, shrinking it instead of growing it. Following this, a 'Get Timeline Length' is used to find the duration of the 'Grow Pool Timeline' and that float value is supplied to a delay node. Once the delay node is completed, the 'Grow Pool Timeline' will have scaled the 'decal_PaintPuddle' back down to 0, allowing for the actor to then be destroyed without being seen.

Paint Bomb (v2 'Bouncy')

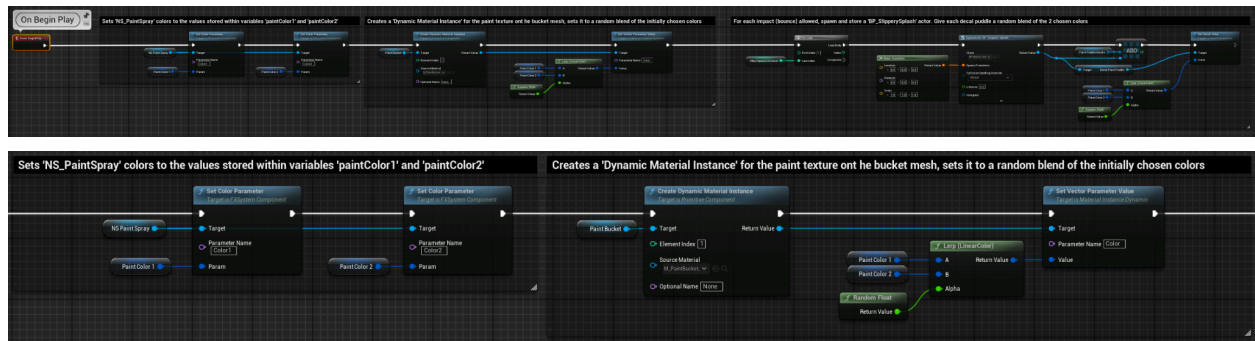


The 'Paint Bomb - Bouncy' variant requires a bit more explanation into exactly how it works, as it's a mash up between 'Physics Constraints' and the 'Projectile Movement' system.



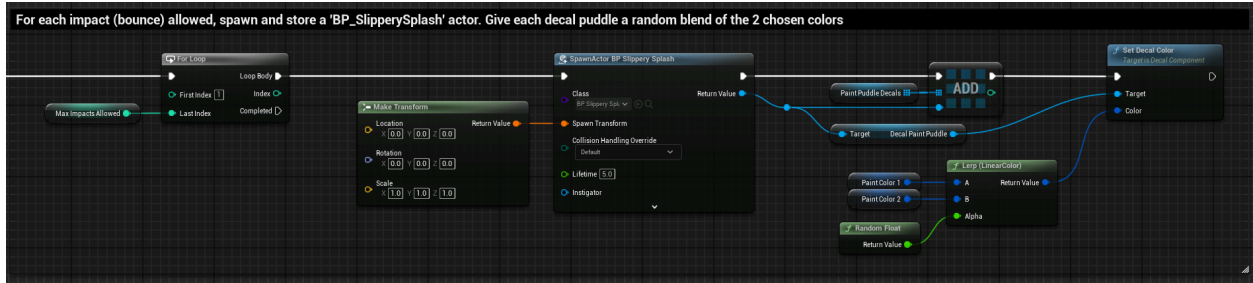
In short, the 'paintBucket' mesh is simulated, and a child in the 'PC_PaintBucketToCollision' physics constraint relationship. Linear movement is locked and Angular movement is free, this results in the buckets erratic spinning movements as it's dragged along by the influences of the 'Projectile Movement' component.

Begin Play



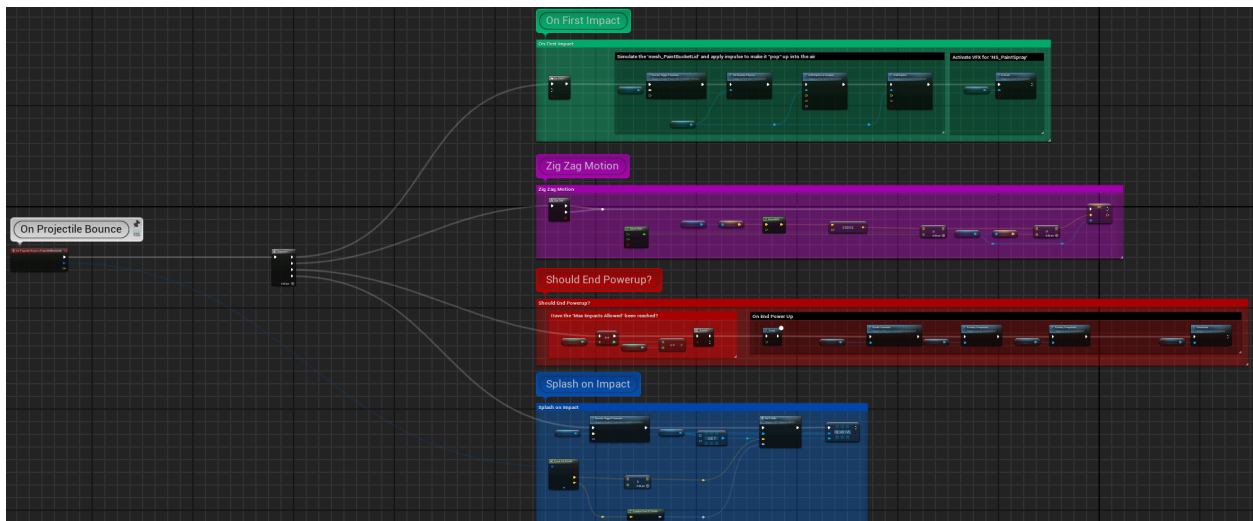
On 'Begin Play' the values of the linear color variables 'paintColor1' and 'paintColor2' are set to 'NS_PaintSpray' parameters of 'Color1' and 'Color2', setting the blend of colors for the power up. Following this, the paint material found on the 'paintBucket' mesh is created as a 'Dynamic Material Instance', then has it's color set to a random blend of 'paintColor1' and 'paintColor2'.

Note: 'paintColor1' and 'paintColor2' are the two blends of color which this blueprint will access, both are instance editable and exposed on spawn, so every instance can be customized.



Finally a 'For Loop' is run for each 'Max Impacts Allowed' set for the integer variable (this variable is essentially how many bounces the power up will have before deactivating). In preparation for this amount of impacts, a 'BP_SlippySplash' will be created and stored to a 'paintPuddleDecals' variable for application later on, their 'decalPaintPuddle' decal is also set to a blend of 'paintColor1' and 'paintColor2'.

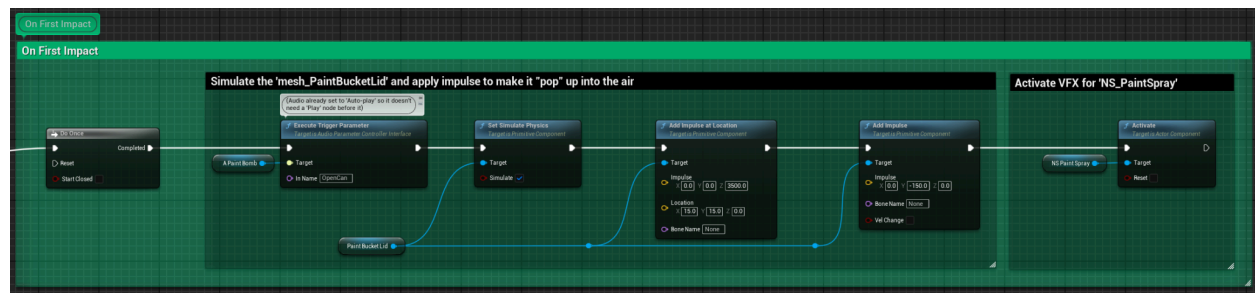
On Projectile Bounce



'On Projectile Bounce' has 4 distinct sequential parts

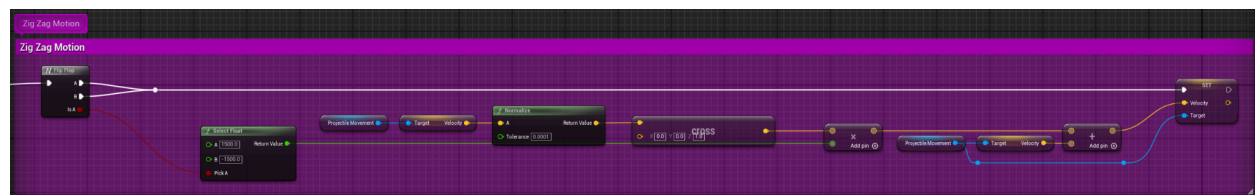
- On First Impact
- Zig Zag Motion
- Should End Powerup?
- Splash on Impact

On First Impact



This section is only fired a single time, as per the 'Do Once' node. It's responsible for activating the 'MetaSound' of 'A_PaintBomb', while also activating physics for the 'paintBucketLid' to cause a "popped off" kind of effect. Following this, the VFX for 'NS_PaintSpray' is activated.

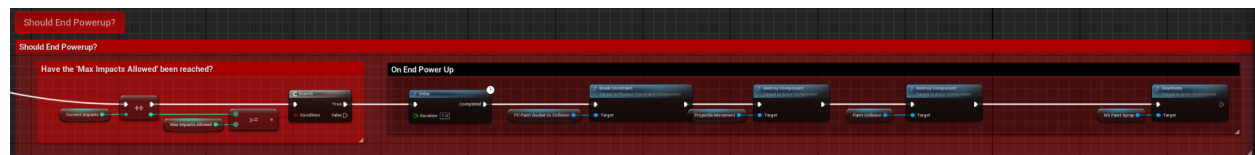
Zig Zag Motion



This section focuses on the 'Zig Zag' motion for every sequential bounce.

How it works is by getting the 'projectileMovement' current velocity and getting its cross plane value, it is then multiplied by a value of 1500 or -1500, before having that final result be set as the new 'projectileMovement' velocity.

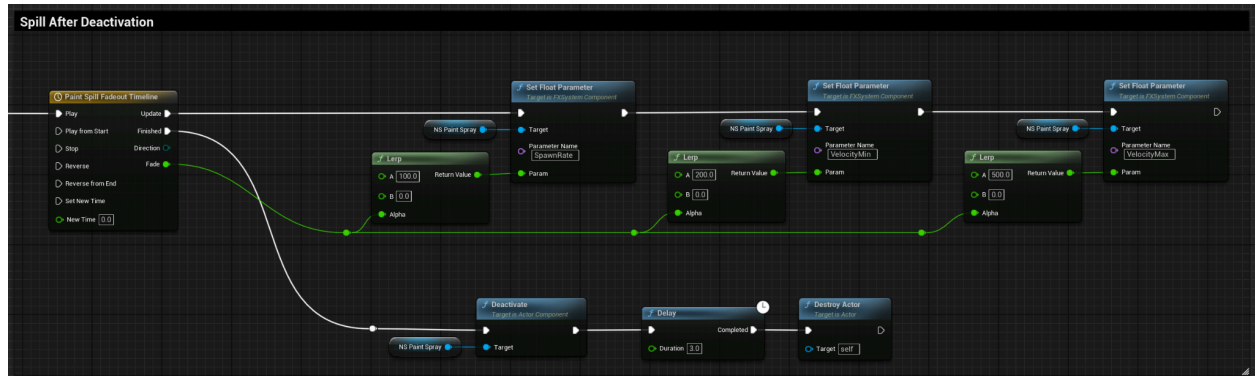
Should End Powerup?



This section determines if the power up has reached the end of it's expected life by incrementing every impact to a 'currentImpacts' integer, that value is then compared to 'maxImpactsAllowed' integer to see if it's value is greater or equal to it.

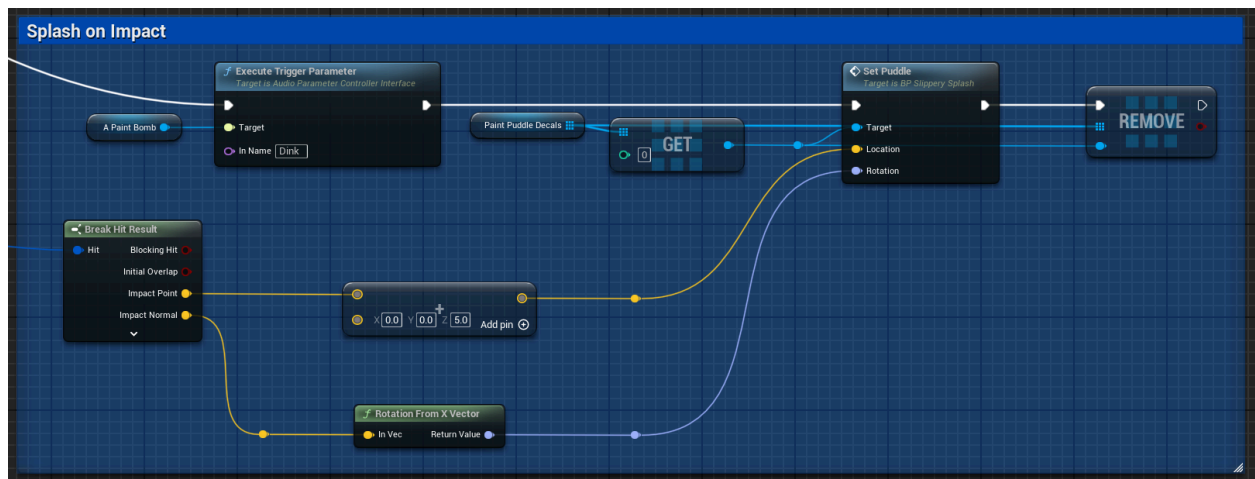
If TRUE then a delay will begin to allow for a bit more VFX to run before cutting it off, then the 'PC_PaintBucketToCollision' physics constraint will be broken which allows the

bucket to move beyond the restrictions that initially shackled its existence. Following this the 'projectileMovement' component is destroyed along with the 'paintCollision', this is really just a shortcut for ensuring that neither have any further impact on the game world going forward (I'm unsure if this is an 'optimal' strategy performance wise).



Finally, the 'Paint Spill Fadeout Timeline' will be run (Current duration length is 10.00) and begins setting the 'NS Paint Spray' Spawn Rate, Velocity Min, and Velocity Max reduced to 0 as the timeline progresses. Once the timeline finishes the 'NS Paint Spray' will be deactivated, following this is a brief delay of 3 seconds before the entire actor is destroyed.

Splash on Impact



This final section handles the SFX and setting of puddles per the bucket's impacts.

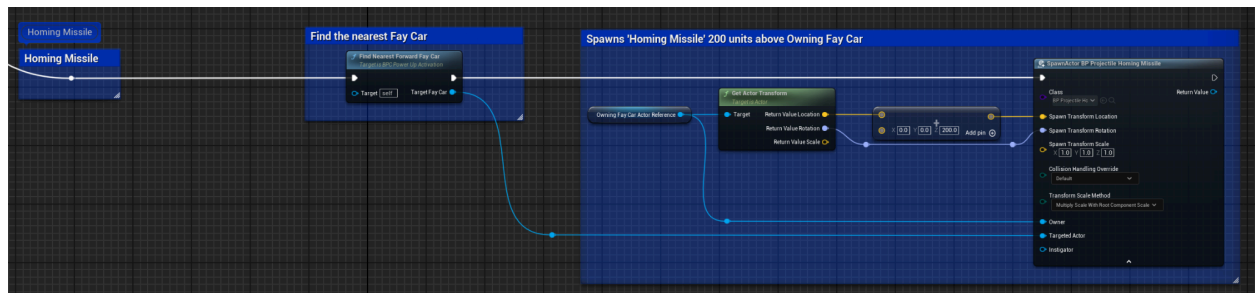
Firstly the 'MetaSound' for 'A_PaintBomb' initiates it's "Dink" trigger, a reference from the 'paintPuddleDecals' array is pulled and feeds the 'BP_SlippySplash' the information for moving it to the location and rotation requested. Finally that puddle is

removed from the 'paintPuddleDecals' array, this method allows for the next available puddle to always be at index 0.

Note: There is a +5 to the Z value of the 'Impact point', this is done to give the puddle a little bit of a gap between the contact surface; as the 'Impact Point' would make it flush with it.

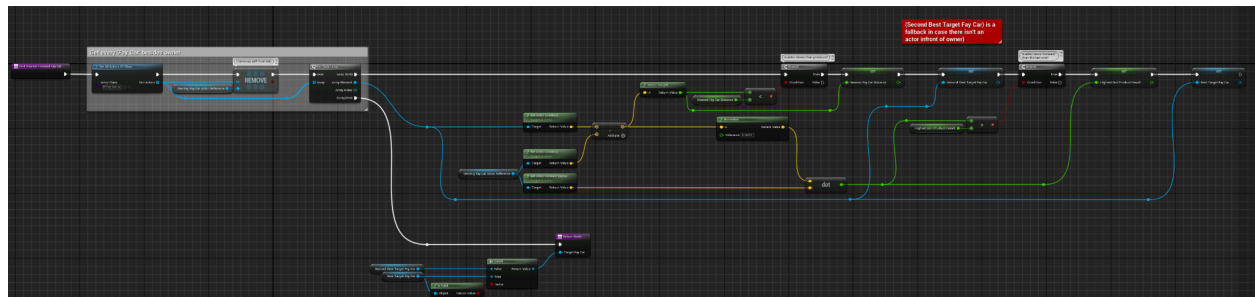
(See section 'BP_SlippySplash' for event 'Set Puddle')

Homing Missile

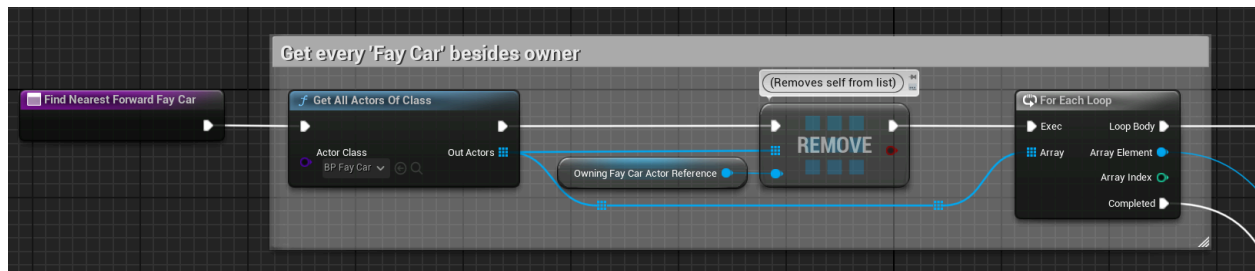


When the 'Homing Missile' power up is activated, it first activates the function 'Find Nearest Forward Fay Car'.

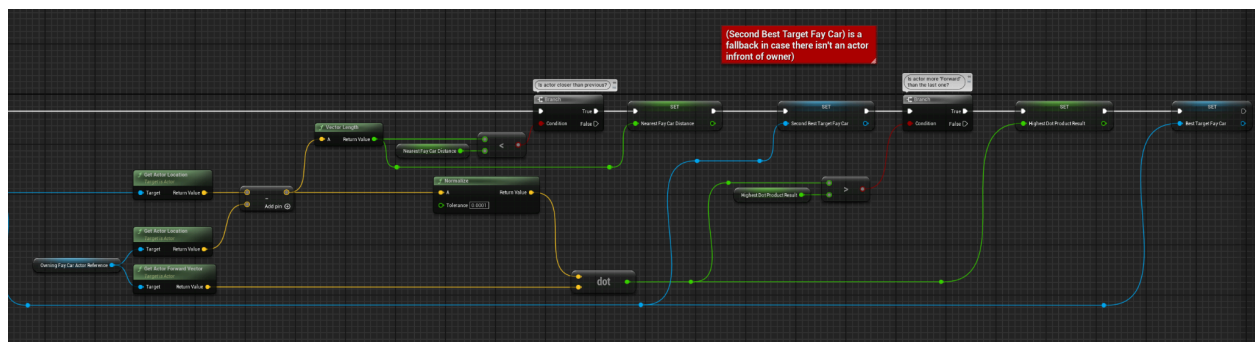
Find Nearest Fay Car (Function)



The purpose of this function is to find the nearest FayCar (omitting the owner), both in distance and relativity to the forward direction of the owner.



The function begins by first using a 'Get All Actors of Class' node to obtain each instance of 'BP_FayCar' within the level, immediately after this the owner is removed from the list to prevent them from being the target of their own search. The remaining actors in the array are then fed into a 'For Each Loop'.



For each instance of 'BP_FayCar' found within the level, their location is first retrieved and subtracted from the owners location, the result of this is fed into a 'Vector Length' node, the return value giving us the unit distance away that Fay Car is from the owner.

The distance is then compared to the 'nearestFayCarDistance' to see if it's less than the previous set value. (Note: The default value of 'nearestFayCarDistance' is 10000000000000000.0, the purpose of this is to make the first entry into the loop become the value to compare against the remaining Fay Car instances.)

If TRUE then that distance becomes the new set value of 'nearestFayCarDistance' and continues.

If FALSE then the loop ends for that actor.

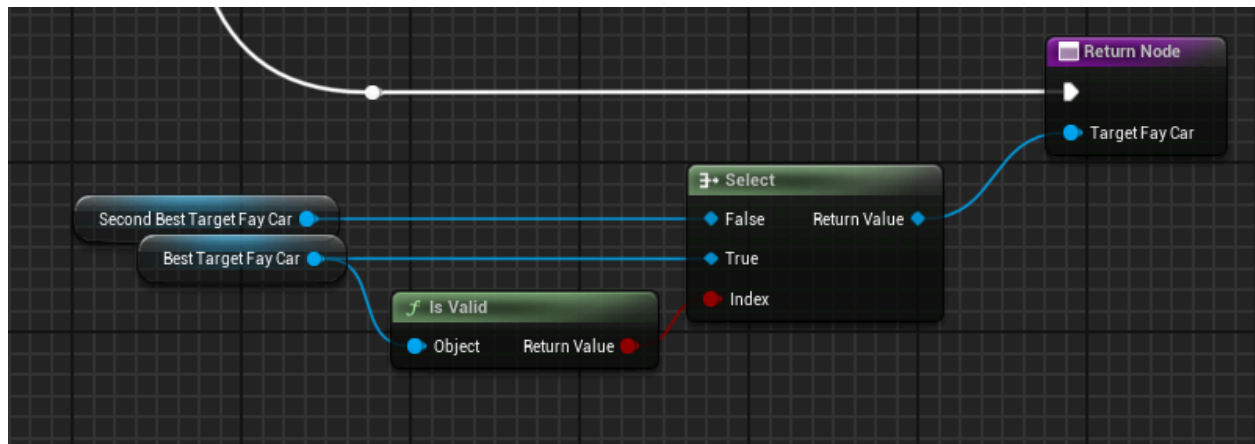
After the actor is deemed to have the 'nearestFayCarDistance', it is then set as the 'secondBestTargetFayCar'. This is done in the event that none of the Fay Car instances meet the directional requirement seen later.

After the instance is considered to be the 'nearest', it then has its location compared to the owner's forward direction via a 'Dot Product' node. The closer the result of the 'Dot Product' is to a value of **1**, the more 'in front' of the owner it is. Similarly, the closer the result is to **-1**, the more 'behind' of the owner it is.

To determine which Fay Car is more 'in front' of the owner, the value is compared to the 'highestDotProductResult' to see if it's greater (Note: The default value of 'highestDotProductResult' is 0).

If TRUE then it becomes the new set value of 'highestDotProductResult' and is set as 'bestTargetFayCar'.

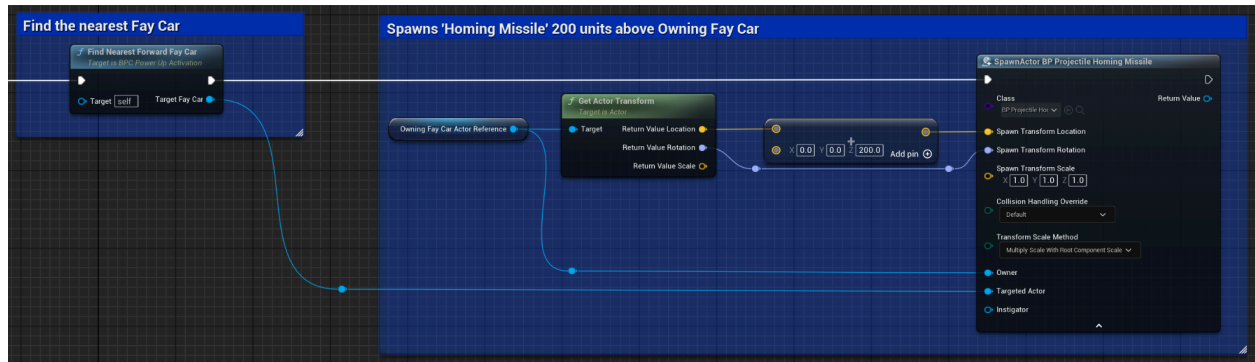
If FALSE then the loop ends for that actor.



Once the 'For Each Loop' has gone through each array item, then the function will output either the 'secondBestTargetFayCar' or the 'bestTargetFayCar', the best being selected if it's valid.

(Note: Ideally the simplest way to decide on the car to target would be with the owner reading what position they are in the race, subtracting by 1, finding the actor in that position, and setting them as the target. Unfortunately there isn't a simple way of obtaining that kind of information just yet.)

Homing Missile (cont.)

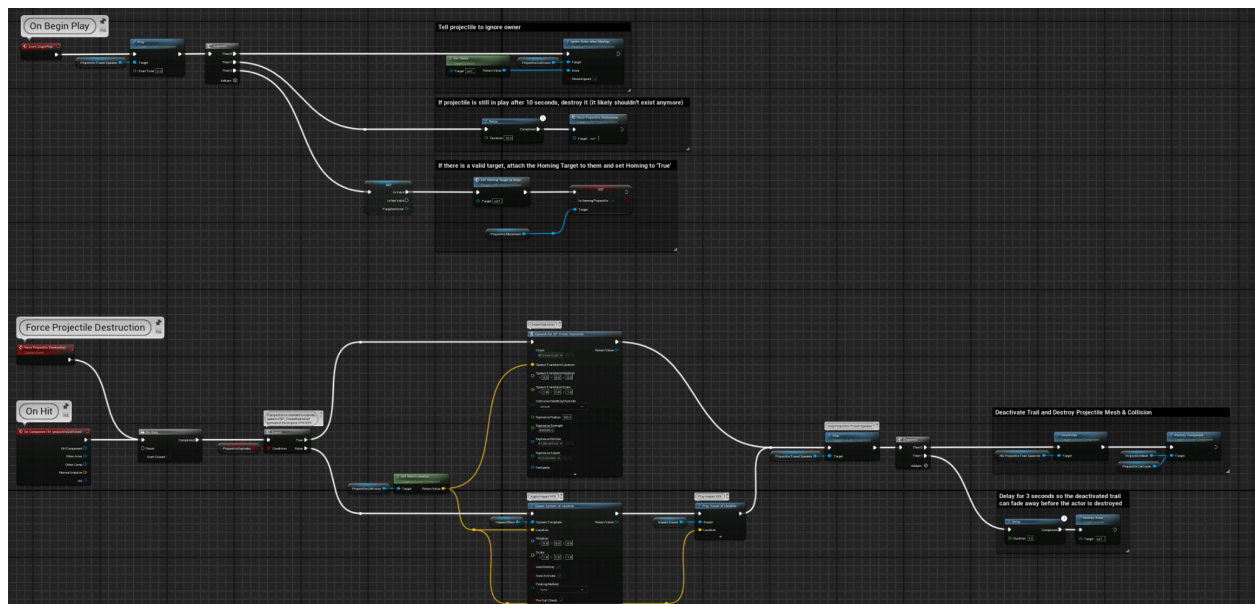


Once a 'targetFayCar' is supplied, a 'Spawn Actor From Class' node is used to spawn a 'BP_ProjectileHomingMissile'.

For the spawn location, we first use the 'Get Actor Transform' node and ADD a value of 200 to the Z axis of the location vector. This is done to spawn the missile above the owner, otherwise it would spawn inside/underneath and explode immediately. The rotation is also supplied, but doesn't require any further adjustments.

We use our owner reference for the 'Spawn Actor From Class' owner input, and also use the result of the 'findNearestForwardFayCar' for the 'targetedActor' input.

BP_Projectile_Base

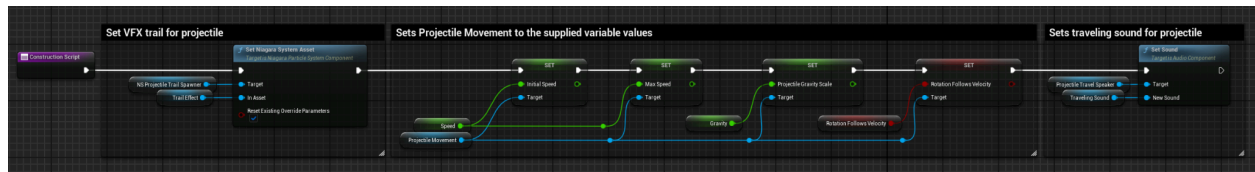


The 'BP_Projectile_HomingMissile' is a child of the blueprint 'BP_Projectile_Base', which houses all the code for the 'Homing Missile'.

'BP_Projectile_Base' consists of 3 major sections:

- Construction Script
- Begin Play
- On Hit

Construction Script



The Construction Script of 'BP_Projectile_Base' is responsible for setting the values supplied to an instance/child instance of the blueprint during spawn.

NS_ProjectileTrailSpawner - Setting the intended VFX supplied by 'trailEffect'.

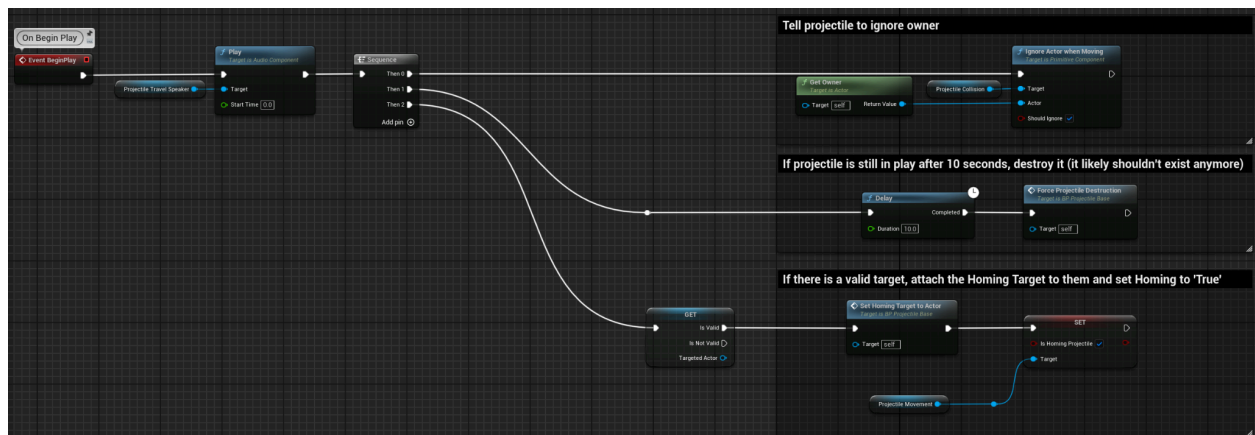
Speed - Setting the projectile's initial spawning, and max, speed.

Gravity - Setting how much downward pull a projectile has (default is 0.0).

RotationFollowsVelocity - If a projectile should rotate towards the direction it's moving.

ProjectileTravelSpeaker - Setting the intended SFX supplied by 'travelingSound'.

Begin Play



On 'Begin Play' the sound set to the projectile's speaker is told to begin playing, following this is a sequence that initiates the following:

Tell Projectile to Ignore Owner - This is an **ATTEMPT** to have the projectile collision to ignore the owner/weapon firing it. **HOWEVER**, it is notoriously disobedient and Unreal themselves have voiced the many inconsistencies they've experienced when utilizing the node. Nevertheless, it's only apparent if the projectile is spawned too close to its owner.

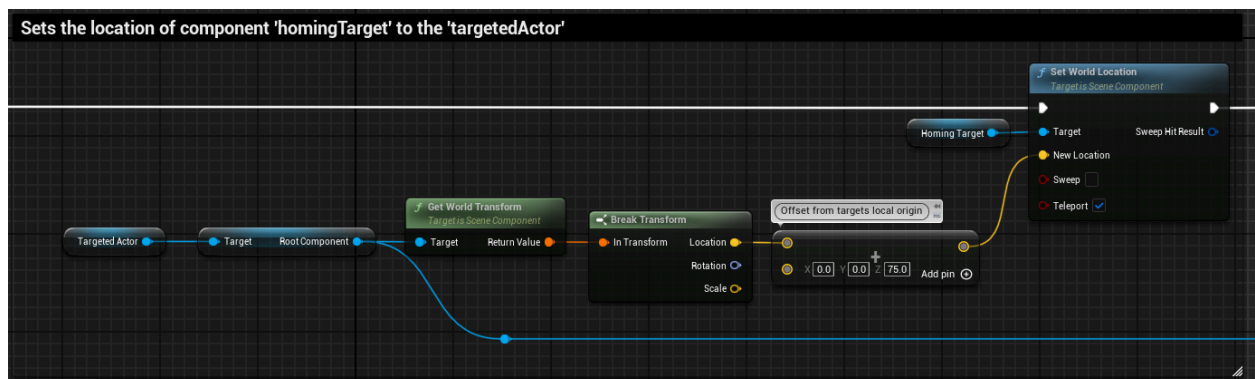
Destroy After 10 Seconds - An initial delay timer is started when the projectile comes into play, upon the timers completion the projectile will be forcibly destroyed. The purpose of this is to remove any projectiles that may have encountered a situation which *undesirably* prolongs their existence.

If Valid Homing Target - If the variable 'targetedActor' has been supplied with a valid value, then the function 'Set Homing Target to Actor' will be initiated, following this 'isHomingProjectile' will be set to TRUE.

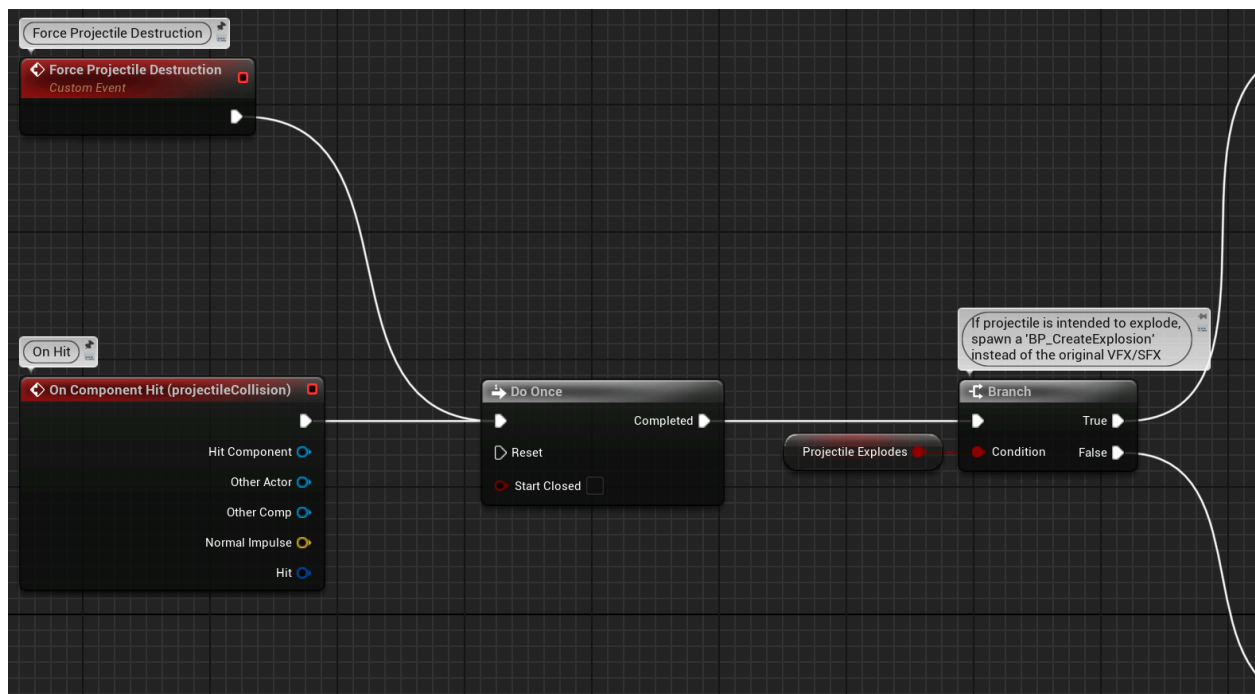
Set Homing Target to Actor (Function)



The purpose of this function is to move and attach the 'homingTarget' scene component to the 'targetedActor'. The reason this must be done is to direct the projectile to a more preferable location on the 'targetedActor', otherwise it would target their relative origin point.

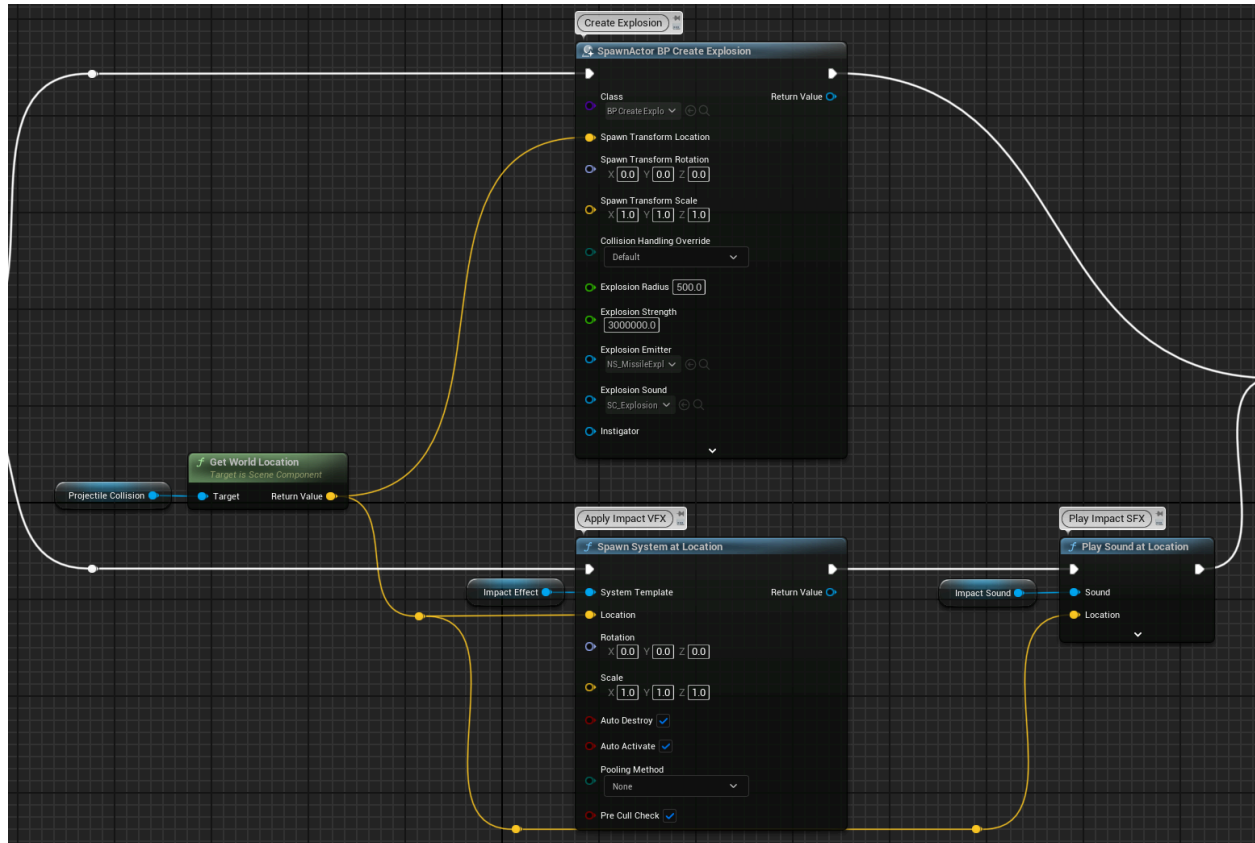


To start, we first get the 'targetedActor' root component world transforms, we then take its location and add a 75 unit offset to the Z axis of the vector, output of which gets fed into a 'Set World Location' node for the 'homingTarget' component.



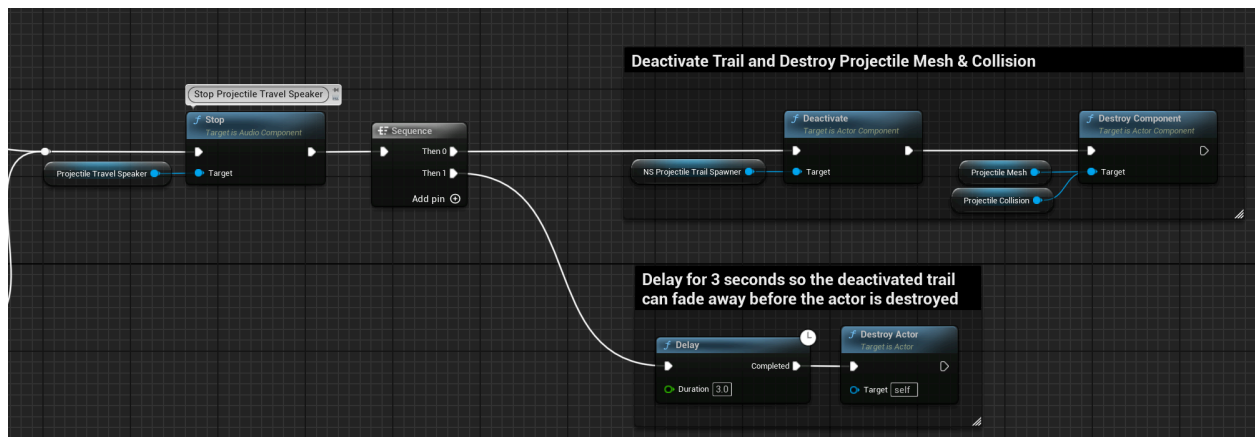
There is a 'Do Once' node placed at the beginning of the event to only allow the projectile to 'Hit' a single time.

The projectile will then first have its bool value of 'projectileExplodes' supplied to the condition of a branch node, this is done to differentiate projectiles that should spawn an explosion, or just produce effects on hit.



If the branch is TRUE then the projectile will use a 'Spawn Actor of Class' node to spawn a 'BP_CreateExplosion', which will produce its own VFX and force.

If the branch is FALSE then the projectile will utilize its supplied 'impactEffect' and 'impactSound' at the location of the projectile.

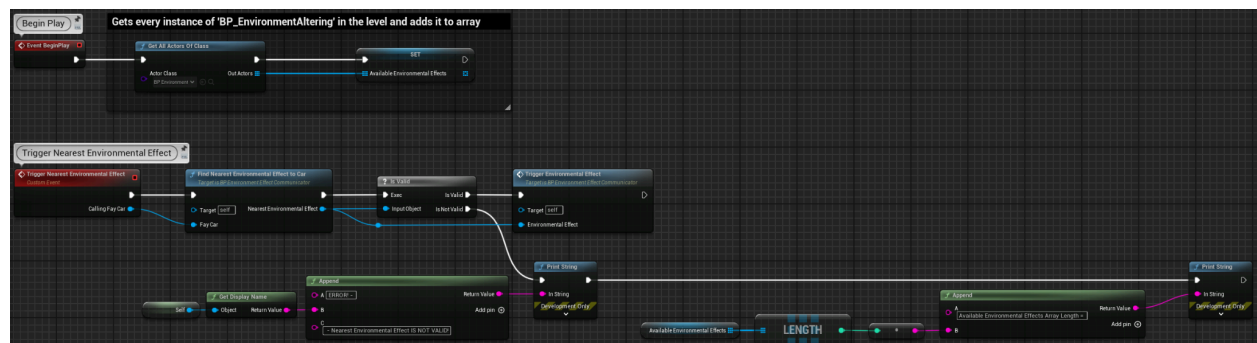


Both paths of the previous branch will join back for the final nodes before the actor's destruction.

Finally a delay will be initiated before the actor's destruction, the purpose of this delay is to allow the trail VFX of the projectile to fully fade away, otherwise the VFX will suddenly pop out of existence.

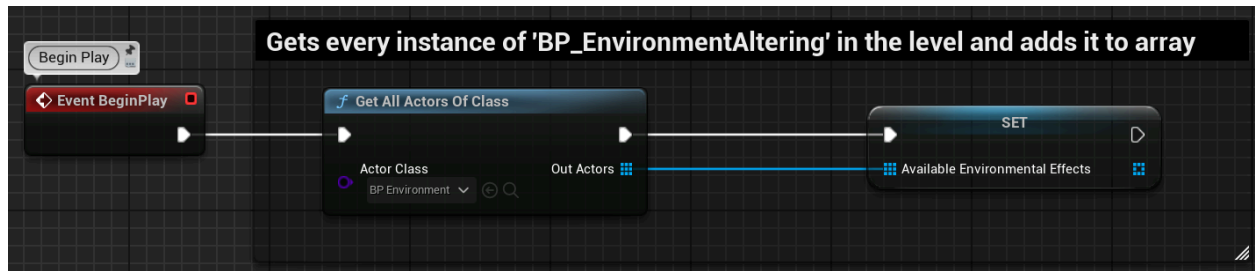
The screenshot displays a Unity HoloLens development environment. A scene is visible with a car and a building. A timeline at the bottom shows a sequence of actions: 'Environment Altering' (a blue box) and 'Activate Environment Effect' (a blue box). The 'Activate Environment Effect' box contains a 'Trigger Nearest Environmental Effect' node, which is connected to a 'Target' node. The 'Target' node is connected to a 'Calling Fay Car' node. The 'Calling Fay Car' node is connected to a 'Fay Car' object in the scene.

BP_EnvironmentalAlteringCommunicator



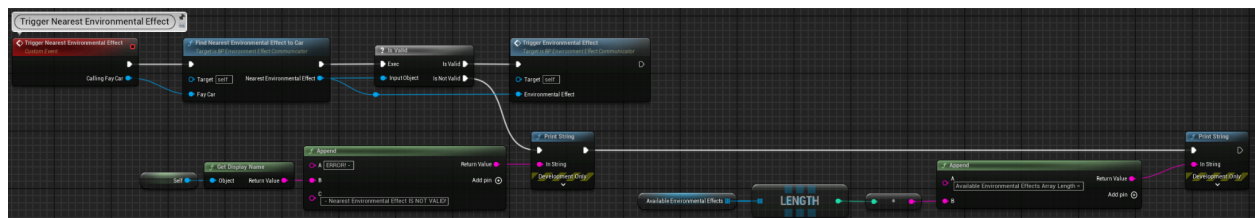
NOTE: 'BP_EnvironmentAlteringCommunicator' is automatically spawned in by the 'Begin Play' sequence found in 'BPC_PowerUpActivation' and should NEVER be manually placed within the level.

Begin Play



On 'Begin Play' every instance of 'BP_EnvironmentAltering_Base' within the current level will be gathered and assigned to the array 'availableEnvironmentalEffects'.

Trigger Nearest Environmental Effect

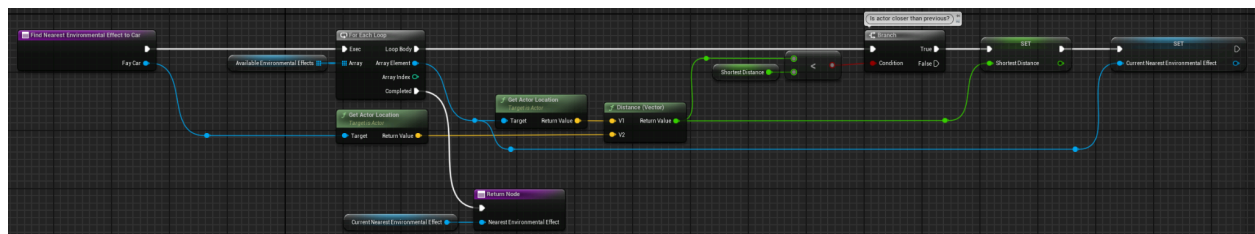


On 'Trigger Nearest Environmental Effect' the function 'Find Nearest Environmental Effect to Car' will be run, utilizing the reference of 'callingFayCar' from the event.

An 'Is Valid' will be run on the output of the function 'Nearest Environmental Effect', if it IS VALID then it will allow the function 'Trigger Environmental Effect' to run with the 'Nearest Environmental Effect' output as reference.

If it IS NOT VALID, then a print string will warn as such and another will state how many 'availableEnvironmentalEffects' are in the array. If this ever happens it's likely that every environmental effect within the level has already been triggered. In the future this will result in an alternative "universal" environmental effect to be triggered in its stead.

Find Nearest Environmental Effect to Car (Function)

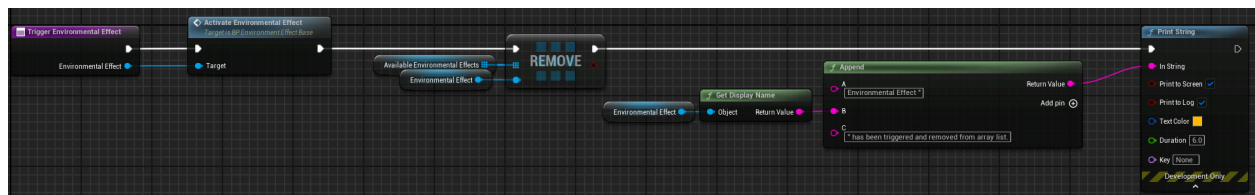


When 'Find Nearest Environmental Effect to Car' is run, it will run through the list of 'availableEnvironmentalEffects' and compare each one's world location to that of the reference supplied during the functions call of 'fayCar'.

The distance of each environmental effect will be compared to the stored local variable of 'shortestDistance', if the distance is less than that found in 'shortestDistance', then that distance will become the new value of 'shortestDistance' and the environmental effect will be set to 'currentNearestEnvironmentalEffect'.

Once the array is finished, the function will return with the value of 'currentNearestEnvironmentalEffect' supplied to 'nearestEnvironmentalEffect'.

Trigger Environmental Effect (Function)



When 'Trigger Environmental Effect' is run, it will utilize the reference supplied on the functions call of 'environmentalEffect' and run its 'Activate Environmental Effect' event.

The same environmental effect will then be removed from the 'availableEnvironmentalEffect' array.

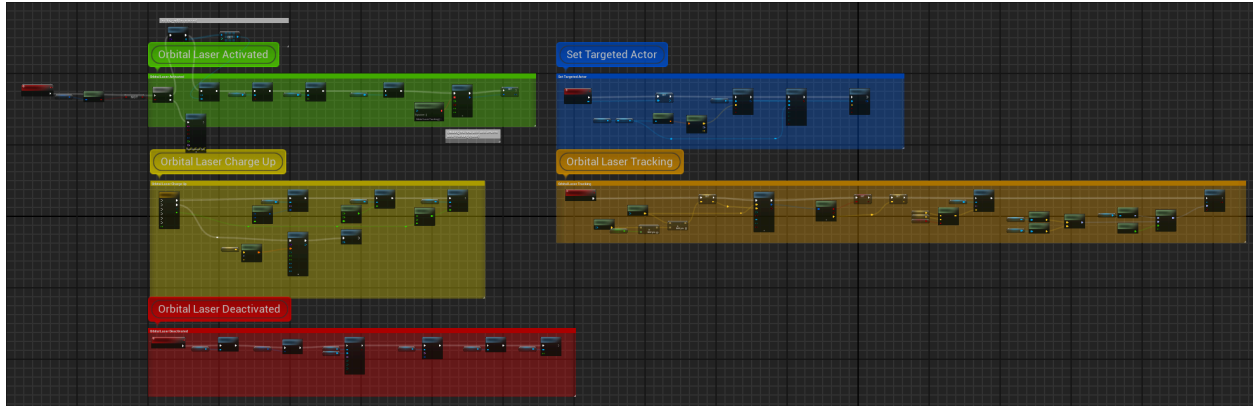
Finally, a print string is run to notify of which environmental effect has been run, and that it has been removed from the array.

Orbital Laser



When the 'Orbital Laser' powerup is activated, it uses the preset reference of the levels 'orbitalLaser' and activates it.

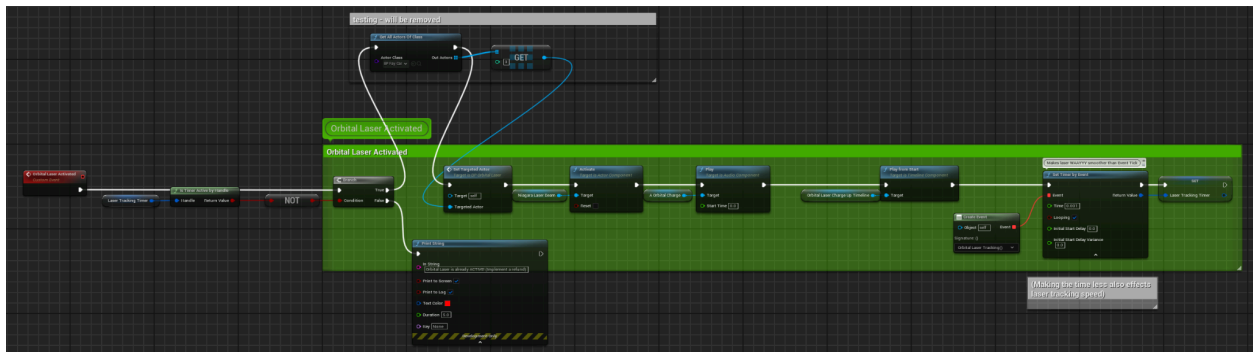
BP_OrbitalLaser

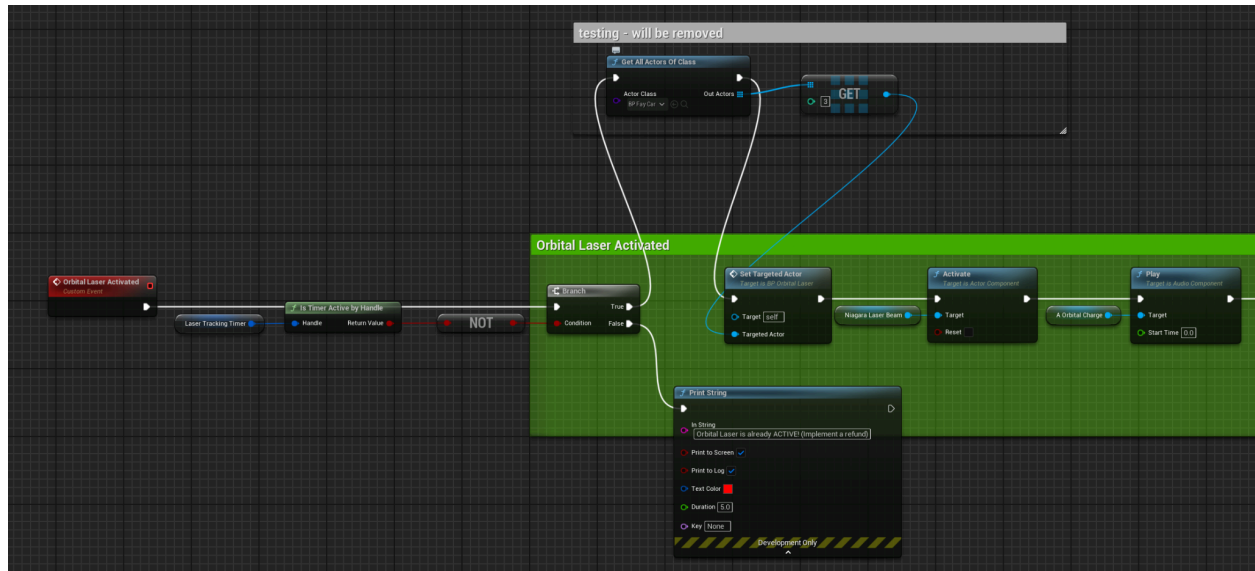


The 'BP_OrbitalLaser' blueprint contains 5 sections to its functionality:

- Orbital Laser Activated
- Set Targeted Actor
- Orbital Laser Charge Up
- Orbital Laser Tracking
- Orbital Laser Deactivated

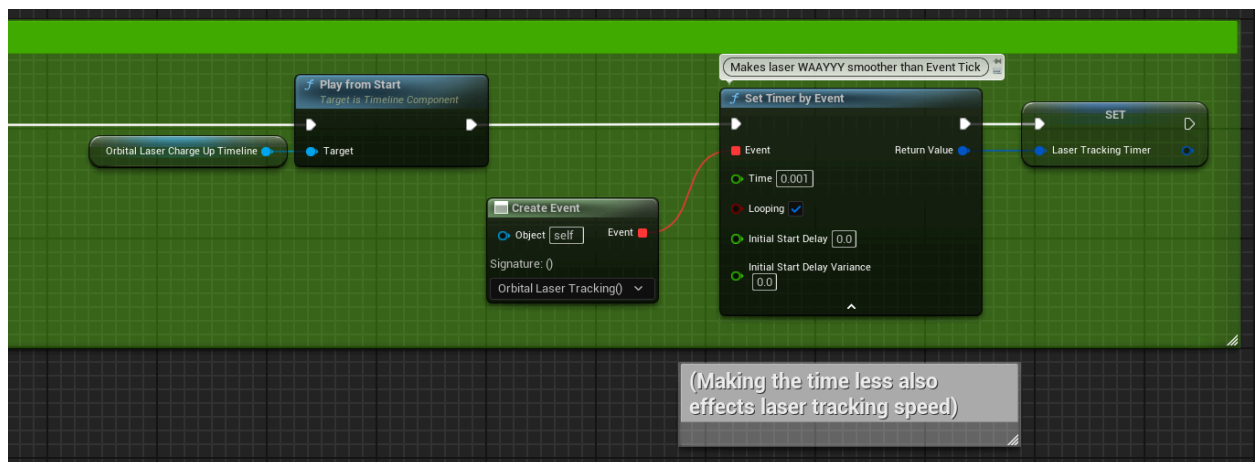
Orbital Laser Activated





On 'Orbital Laser Activated' it will first check to see if the 'laserTrackingTimer' is active, if it is, then the abilities use should be refunded back to the player that activated it (as only one should ever be active at a time), if not then it sets the targeted actor. In the future this will get whichever car is in first place and target them automatically, but for testing, it's currently relying on a 'Get All Actors of Class' node.

Once the target is set, 'NiagaraLaserBeam' particle effect is activated, followed by the audio component for the 'A_OrbitalCharge' being told to play.

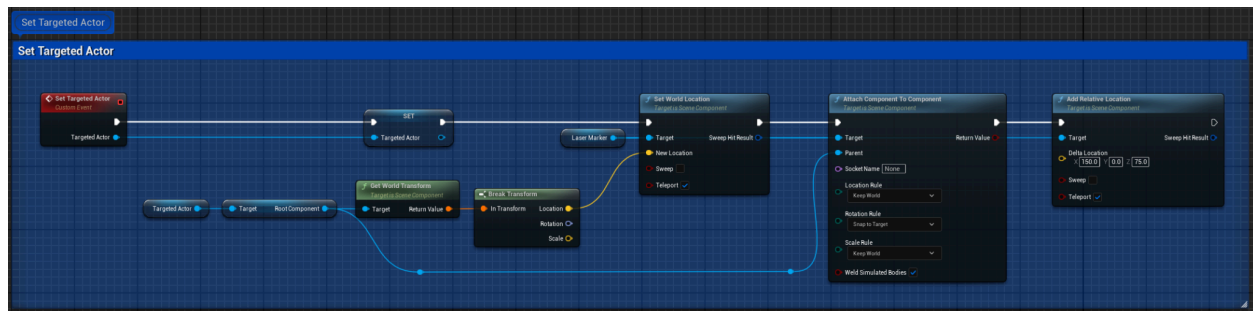


After which, the 'orbitalLaserChargeUpTimeline' will begin for manipulating the niagara VFX. Finally, the 'Orbital Laser Tracking' event will be set by a looping 'Set Timer by Event' node. The laser tracking system was initially used with an event tick to update the laser's position to the target car, however I found in testing that if it's called more often than tick then the laser tracking is incredibly more smooth. The main potential

issue with this current approach is just how many times that event is fired, which is essentially 1,000 times per second, I did notice the game slowed down when it was set to a lower amount but in my testing it has been stable at a time 0.001. It is entirely likely that this could slow the game down when there are more actors and systems happening (like in a real match), once I'm able to test that (and if it does in fact cause performance issues), then the timer can simply be reduced.

A final factor of this section, the 'Time' that is set for this timer does have a direct effect on just how fast the laser is at tracking it's target, with higher values causing it to be slower. This, and the later mentioned 'RInterp To Interp Speed', will be the direct way of adjusting the laser's tracking speed.

Set Targeted Actor



When the 'Set Targeted Actor' event is called, it will take the supplied reference of 'targetedActor' and set it to the variable 'targetedActor'.

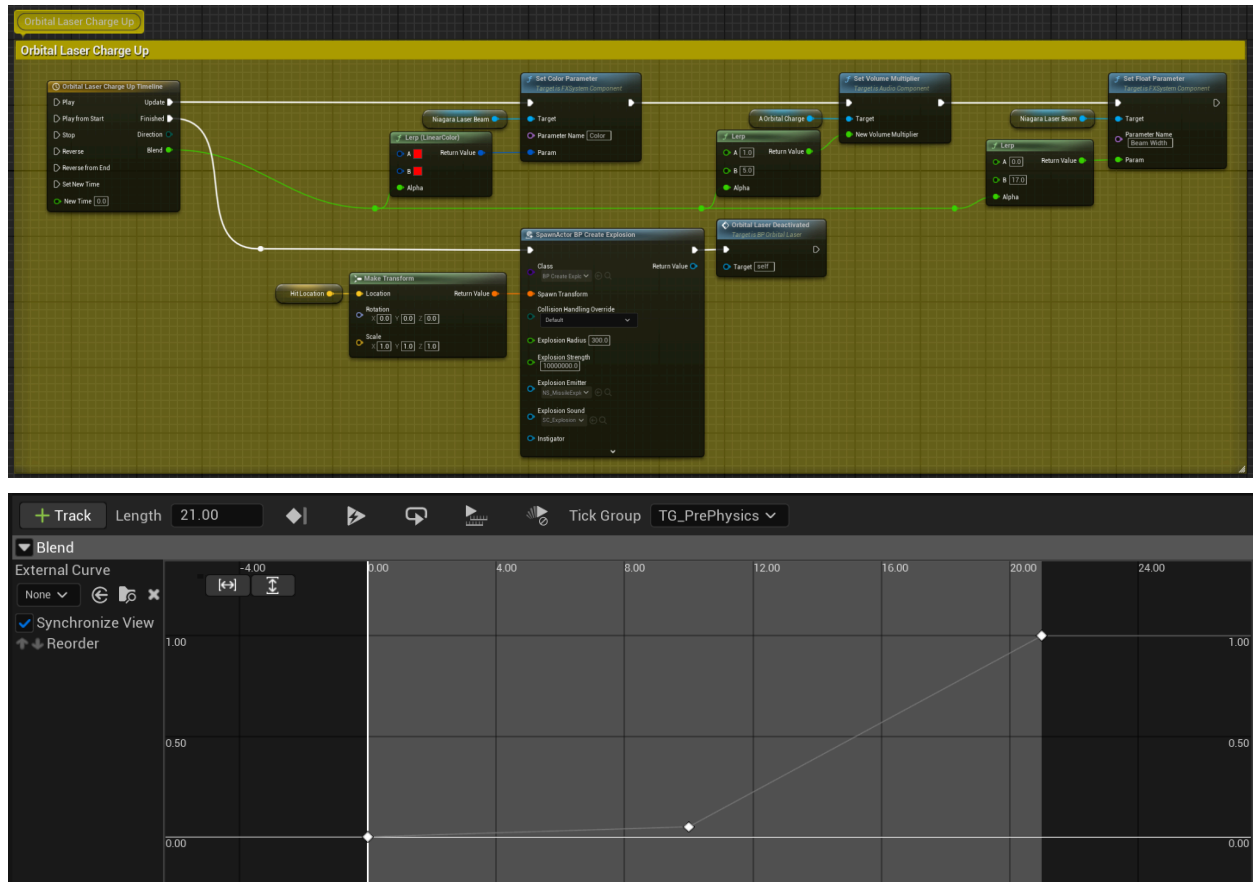
It will then take the 'laserMarker' component (which is invisible sphere mesh) and set its world location to that of the 'targetedActor' variable before using an 'Attach Component to Component' node to attach the 'laserMarker' to the 'targetedActor' root component. Keeping its Location and Scale rule set to 'Keep World' while having its Rotation rule 'Snap to Target' (this is done to line up its facing direction with the facing direction of the actor. Technically the rotation of the 'laserMarker' doesn't matter, but in the event that we need to know the forward direction of the vehicle, we can get it from the 'laserMarker' instead of trying to retrieve it from the 'laserMarker' attach parent i.e. the vehicle.)

After the 'laserMarker' is attached, it will have its *relative* location adjusted so that it lines up with the hood of the FayCar.

Note: When 'Attach Component to Component' node is used, it automatically detaches from the 'BP_OrbitalLaser' actor, yet the reference of 'laserMarker' still remains valid and accessible.

An additional purpose of having 'Set Targeted Actor' as its own separate event, is to allow the laser to target a new actor while it's charging up.

Orbital Laser Charge Up

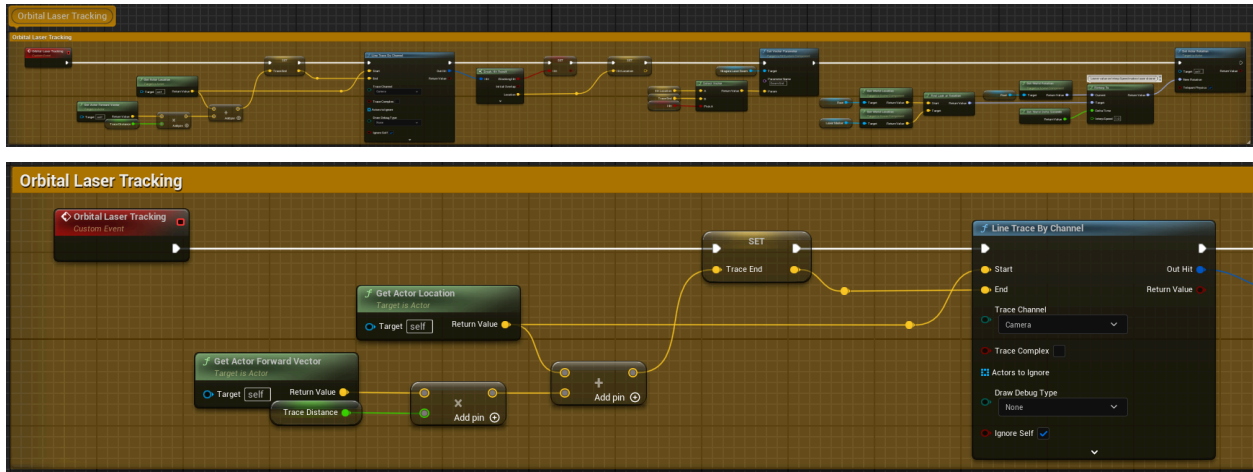


'Orbital Laser Charge Up' is currently a 21 second timer that is responsible for setting the 'blend' float value in accordance with the above key layout. Staying a miniscule value for the first half of the timeline before increasing exponentially to the end of it.

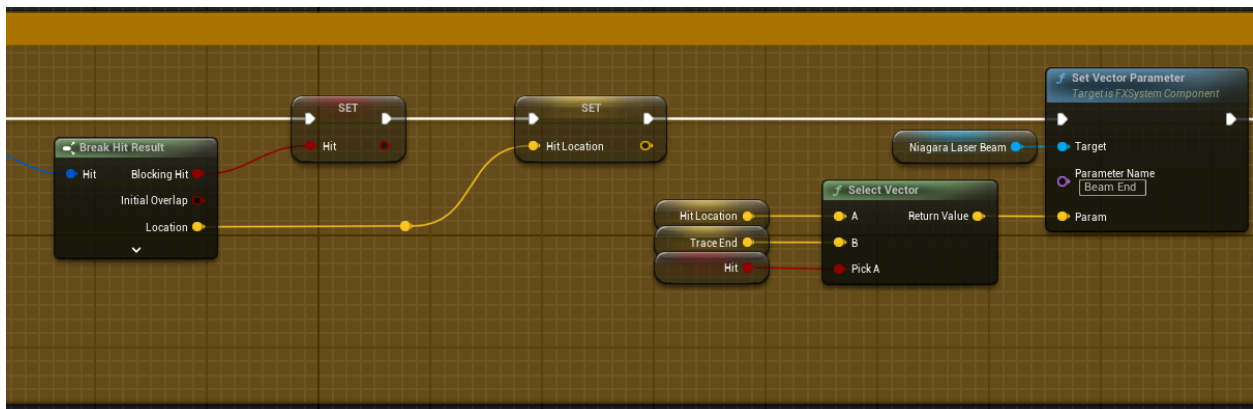
Once the timeline is activated by 'Orbital Laser Activated', it begins to update color (primarily emission) of the 'niagaraLaserBeam', increases the volume of the 'A_OrbitalCharge', and *thickens* the 'Beam Width' of the 'niagaraLaserBeam', each through their own Lerp. In short, the values will start at 'A' when the timeline begins and turn into 'B' as it ends (it's more related to how close the 'blend' float value is from 0 - 1, but that's the general idea).

When the timer finishes, it will spawn a 'BP_CreateExplosion' at the end of the laser line trace before calling the event 'Orbital Laser Deactivated'.

Orbital Laser Tracking



'Orbital Laser Tracking' initially starts with a 'Line Trace by Channel' that begins from 'BP_OrbitalLaser' actor location, and ends based on its forward vector multiplied by the requested 'traceDistance' (default is 50,000).

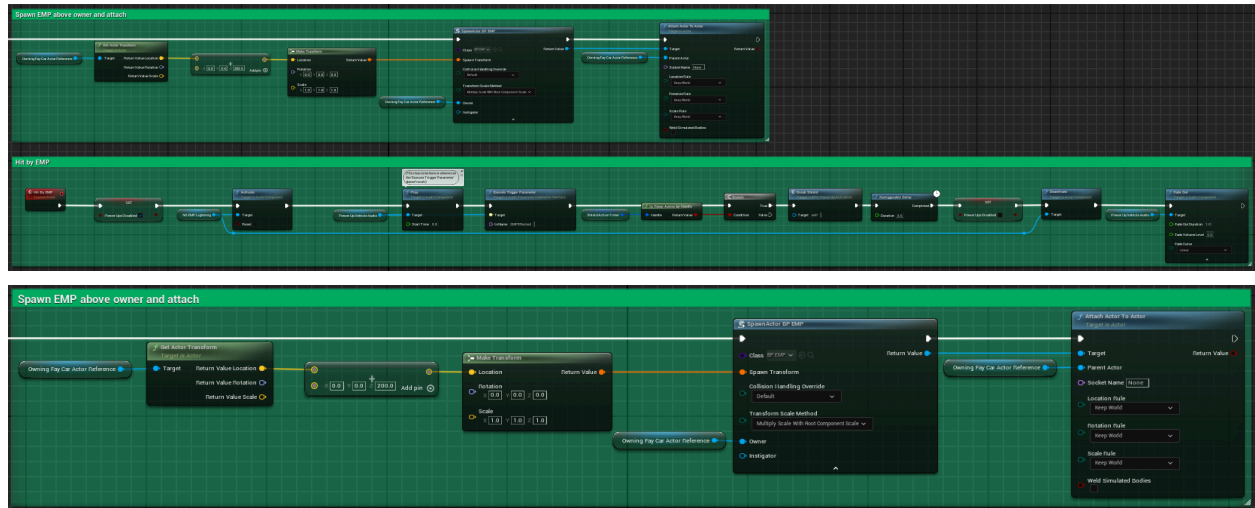


On the break hit result of the 'Line Trace by Channel', we first set the bool 'bHit' if the line trace hit something, and we also set the 'hitLocation' of where the hit took place. Using both of those variables, we indicate where the 'niagaraLaserBeam' VFX should end (so it looks like a laser being blocked).

Finally, the 'niagaraLaserBeam' is set back to its original color, while the 'A_OrbitalCharge' audio component is stopped and has it's volume multiplier set back to 1.0.

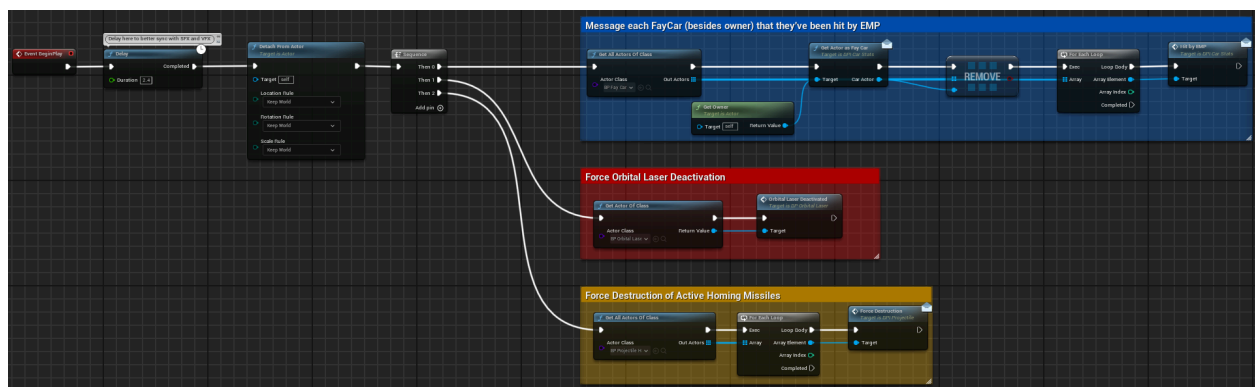
Note: The 'Orbital Laser Deactivated' event can be used to prematurely end laser buildup at any time.

EMP



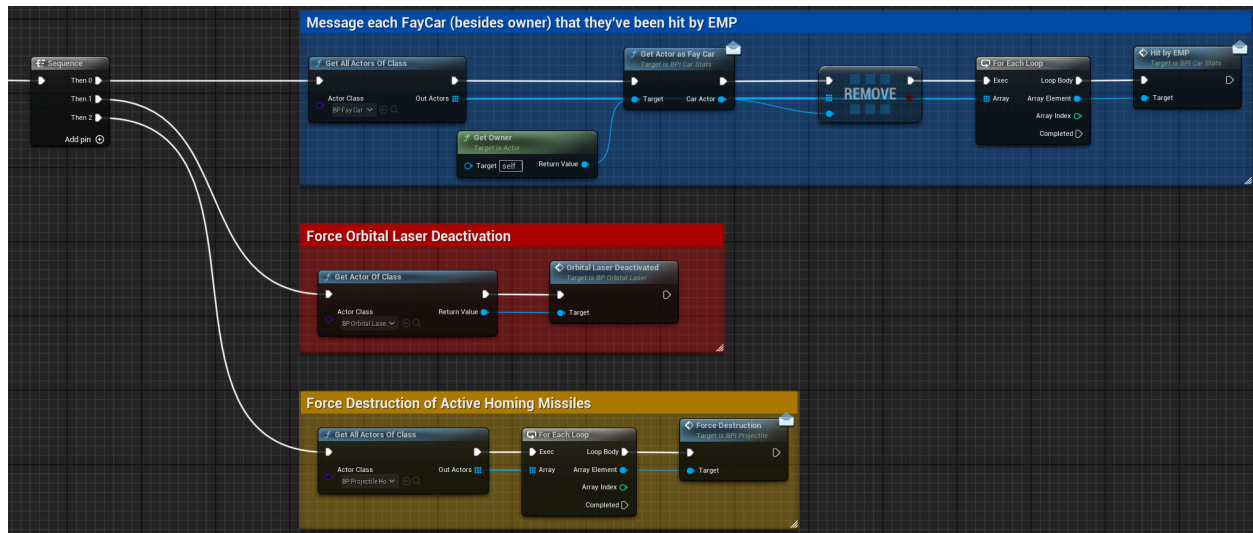
When the 'EMP' power up is activated, a 'BP_EMP' will be spawned at the the owner's +200 units of the owners Z location before being subsequently attached to the owner.

BP_EMP



When 'BP_EMP' is spawned, its VFX (NS_EMP) and SFX (A_EMP) both auto-activate. Following the 'Begin Play' is a delay of 2.4 seconds, this delay is specifically for timing the 'effect code' to the VFX and SFX, ideally these would all be packaged together with a trigger executing when the following code is needed, but this is where it's at for now.

After the delay the 'BP_EMP' has the 'Detach From Actor' node run. The reason for this is to detach the EMP right before it starts to expand over the entire level; were it not detached, then as the car moved and turned the EMP would be seen moving and rotating in the world as well.



Following this is a sequence containing 3 paths:

- Message FayCar about EMP
- Force Orbital Laser Deactivation
- Force Destruction of Active Homing Missiles

Message FayCar about EMP

Every world instance of 'BP_FayCar' is retrieved utilizing the 'Get All Actors of Class' node, we then get a reference to the owner (FayCar that activated the EMP), retrieve it as a 'Fay Car' via BPI_CarStats, then remove them from the world instances found. The array from 'Get All Actors of Class' is then thrown into a 'For Each Loop' and a message to each reference is sent that they've been 'Hit by EMP'.

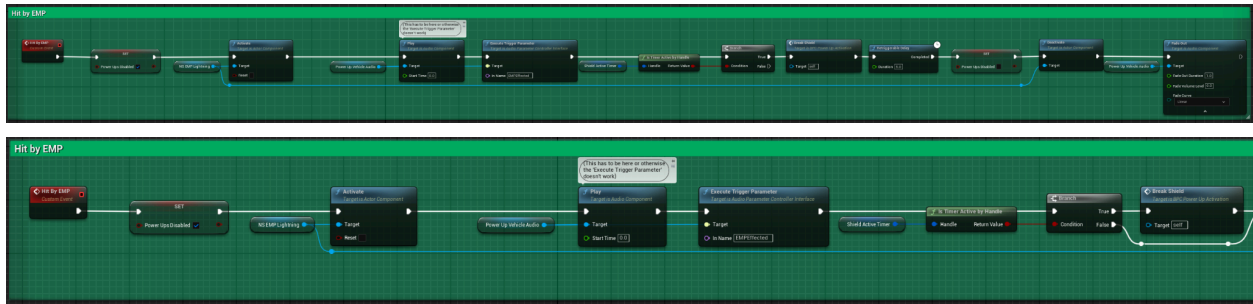
Force Orbital Laser Deactivation

There should only be **ONE** instance of 'BP_OrbitalLaser' within the game at any given time, thus, we use a 'Get Actor Of Class' to retrieve the first index of 'BP_OrbitalLaser', out of the return value we run it's personal event 'Orbital Laser Deactivated'. This will deactivate the laser from further tracking it's target, if the 'Orbital Laser' isn't currently tracking anything, it will still be deactivated but won't incur any issues with doing so.

Force Destruction of Active Homing Missiles

A 'Get All Actors of Class' node is used to obtain any 'BP_Projectile_HomingMissile' found within the level, it then takes the array list of these actors and puts it through a 'For Each Loop', then messages each one of those 'BP_Projectile_HomingMissile' via 'BPI_Projectile' to 'Force Destruction', causing the missile to blow up immediately.

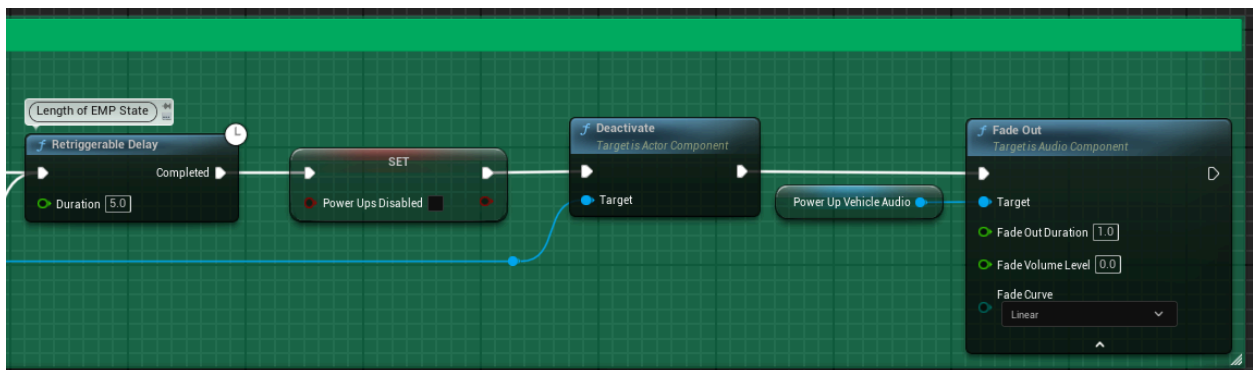
Hit by EMP



On 'Hit by EMP' the bool 'powerUpsDisabled' is set to true (this prevents any abilities from being activated), next the 'NS_EMP_Lightning' VFX reference is told to activate along with the 'powerUpVehicleAudio' being told to execute trigger 'EMPEffected'.

The reason 'powerUpVehicleAudio' is being used in this situation is for 2 reasons, the first being that 'powerUpVehicleAudio' will never be triggered so long as 'powerUpsDisabled' is TRUE, which means nothing will accidentally interrupt it. The second reason is to try and avoid hitting Unreal's limitation on the max number of audio components that can be active in the world at a given time.

The 'shieldActiveTimer' is then checked to see if the 'Shield' power up is active, and ordering it to run 'Break Shield' if it is.



Next, a 'Retriggerable Delay' is activated for a duration of 5.0 seconds, **the duration value of 'Retriggerable Delay' is how long the EMP effect will last for**. The reason for the 'Retriggerable Delay' over a normal 'Delay' is in preparation for a similar scenario found with the

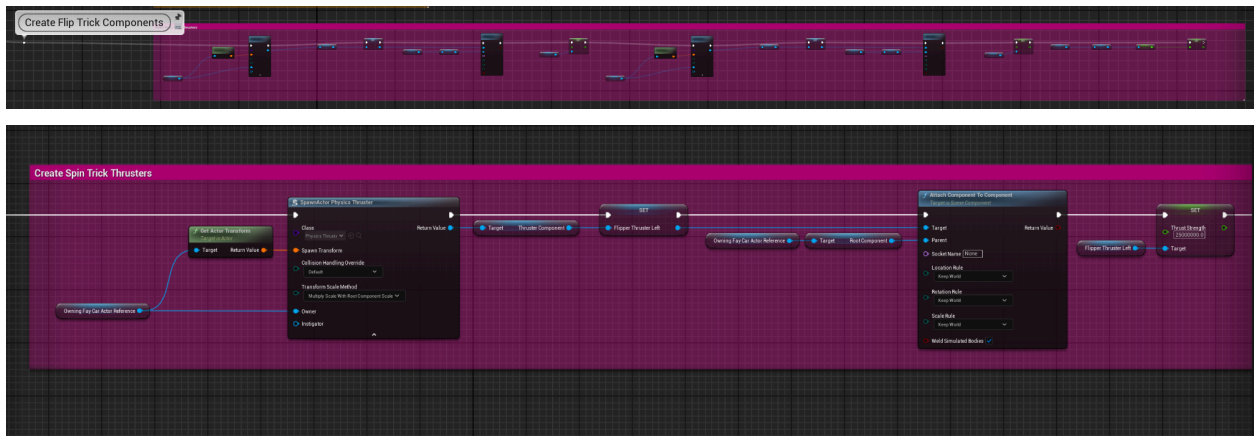
'Speed Boost' power up, essentially accounting for any additional calls of 'Hit by EMP' to simply "extend the duration" (so to speak) of how long the EMP effects are active for.

Once the 'Retriggerable Delay' is complete, the bool 'powerUpsDisabled' will be set to FALSE (allowing FayCars to once again use power ups), the VFX for 'NS_EMP_Lightning' is deactivated, and the 'powerUpVehicleAudio' is set to 'Fade Out' of a duration of 1.0 seconds.

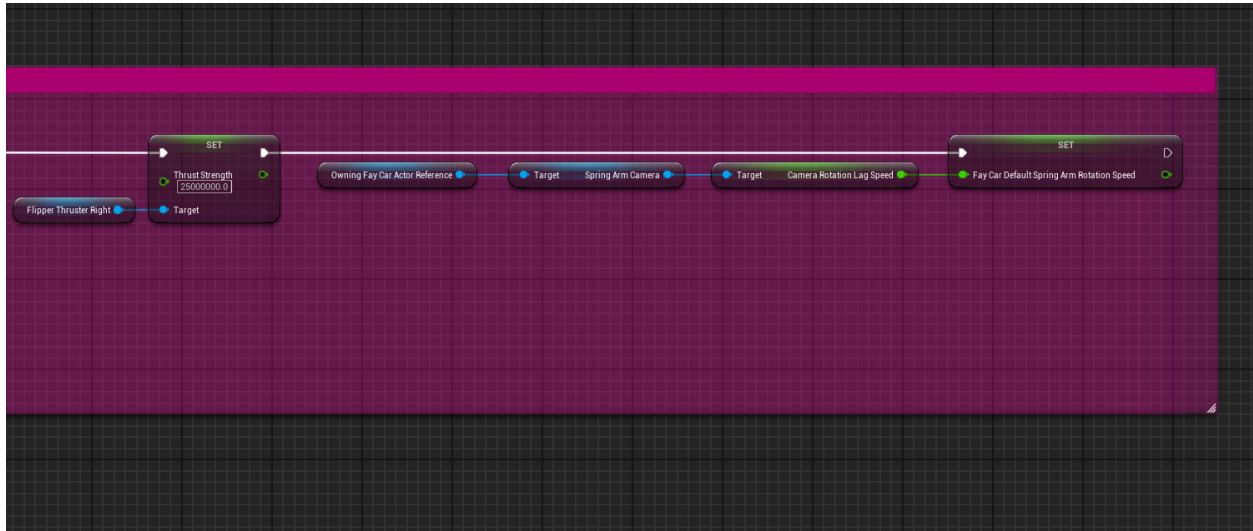
Note: If the a player under the effects of an EMP finally has it end, but the 'powerUpVehicleAudio' is still in the process of 'Fade Out', I'm unsure if any audio they trigger during this period will inject itself into the place of the previously found electrical sound of 'powerUpVehicleAudio', and cause itself to be immediately faded out by accident. It's equally possible that a 'Fade Out' is negated the moment the audio component is told to 'Play', but again I'm not sure.

Flip Tricks

Begin Play

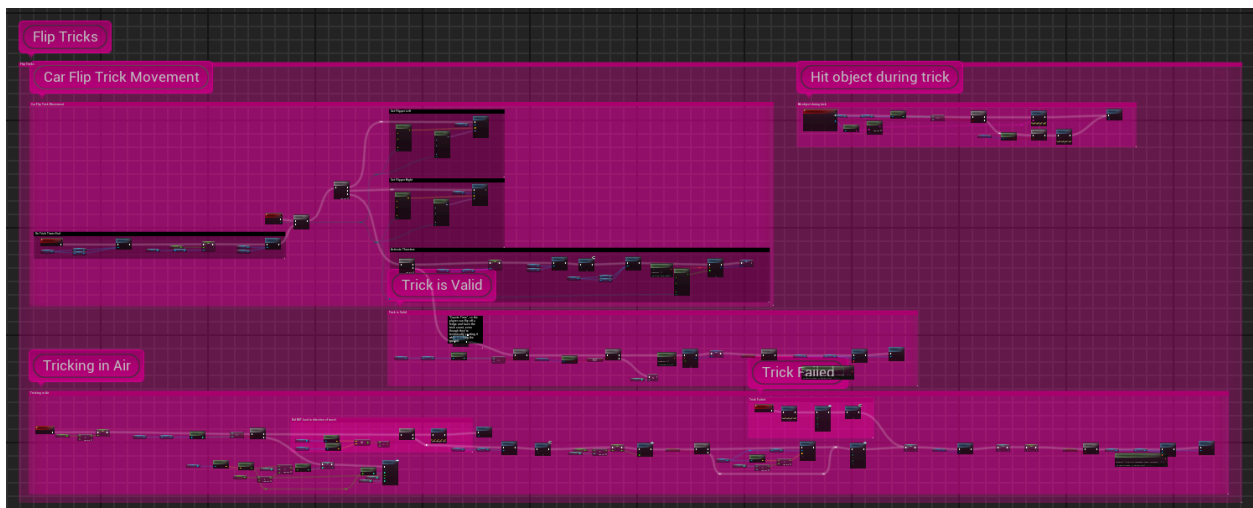


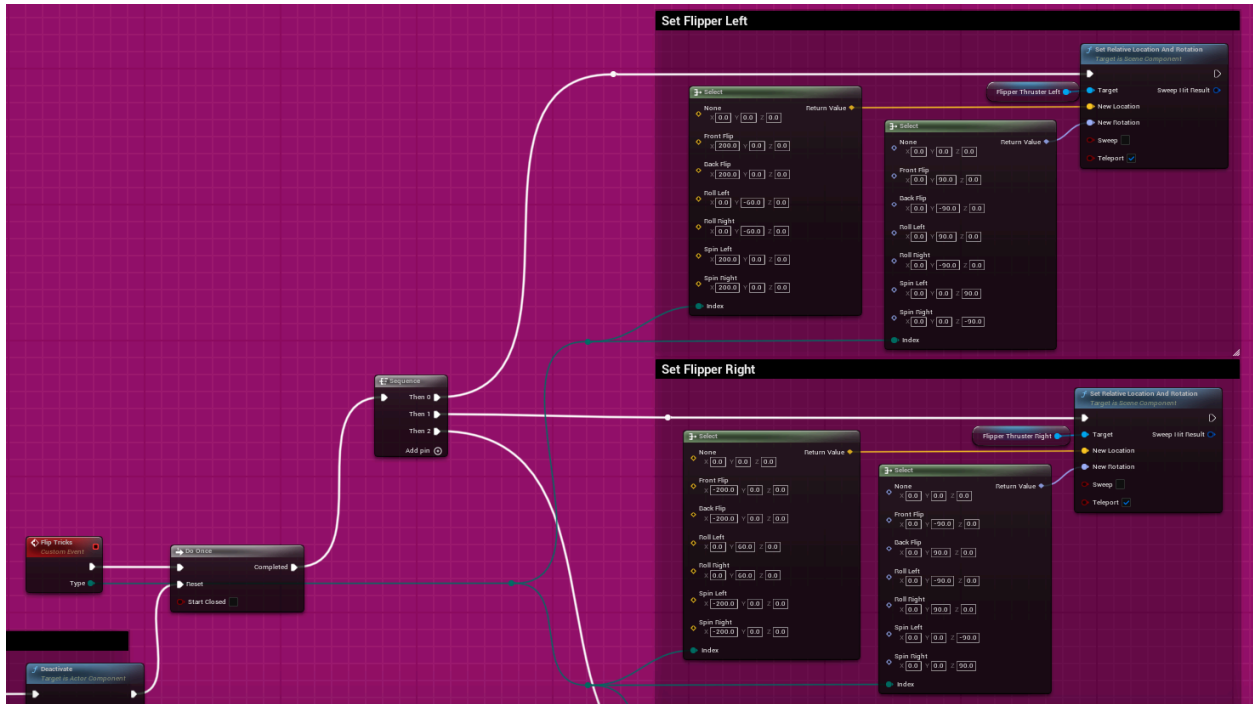
On 'Begin Play' 2 'Physic Thruster' Actors will be spawned and attached to the owning FayCar. One designated as 'flipperThrusterLeft', and the other as 'flipperThrusterRight'. They will then further have their thrust strength set to 25000000.



Next, a reference to the owning FayCar ‘Spring Arm Camera’ will be obtained, with the value of it’s ‘Camera Rotations Lag Speed’ be saved to the variable ‘fayCarDefaultSpringArmRotationSpeed’.

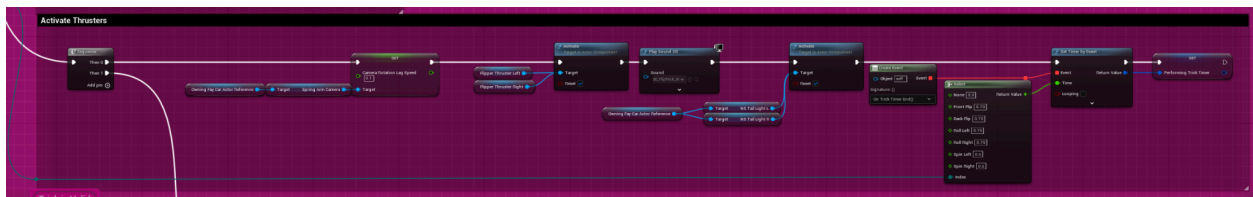
Flip Tricks





When the event 'Flip Tricks' begins, it will first enter a 'Do Once' node (this is to ensure a trick fully completes before another can take place), which will then plug into a 'Sequence' node. The first two outputs of the 'Sequence' sets the relative location and rotation of both 'flipperThrusterLeft' and 'flipperThrusterRight'. These locations and rotations are set based on the trick type that was supplied by the ENUM output of the 'Flip Tricks' event. That ENUM feeds into select nodes that have been curated for moving the thrusters to specific locations to allow a particular impact on the FayCar.

To adjust these, or add new ones, you will have to set the location and rotation values within the select nodes, then run the game and test it. It might be a pain in the ass not having a visual cue within the viewport, but after you do it once it's really not that bad; plus this method saves on additional resources.



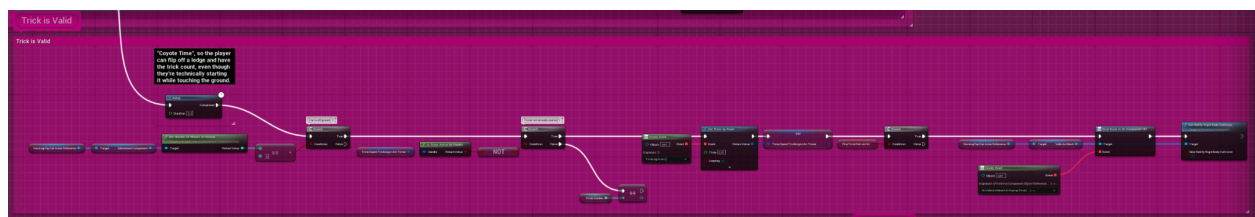
Out of that final output of that previously mentioned sequence node, it will feed into an additional sequence dedicated to activating the thrusters and registering a trick being performed.

First, the owning FayCar 'Spring Arm Camera' will have its 'Camera Rotation Lag Speed' set to 0.1, the purpose of this is to make the camera not follow the rotation of the FayCar while it's performing tricks. I tried many different methods to get this to work, and this was by far the best one that produced no snapping.

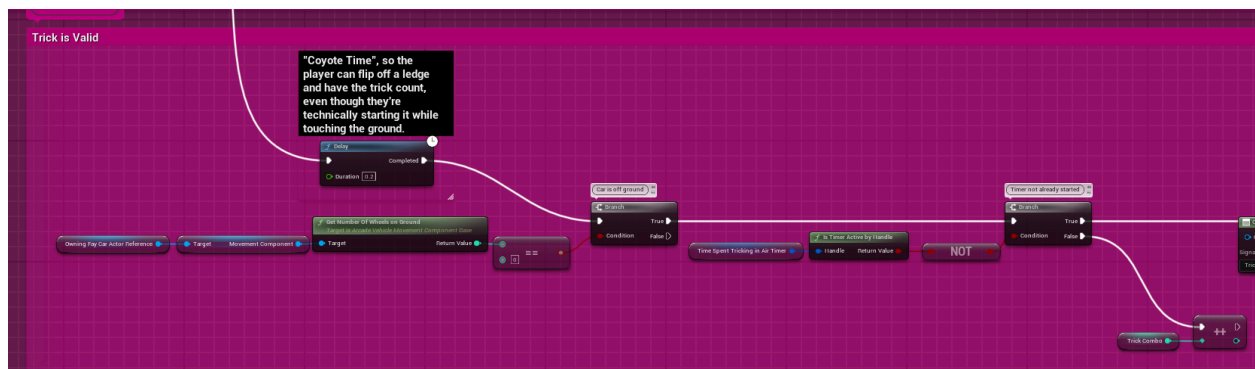
Next 'flipperThrusterLeft' and 'flipperThrusterRight' will be 'Activated', and the sound cue 'SC_FlipTrick_Start' will be played in '2D'. (My intention here was to avoid additional audio components, or interfering with pre-existing ones. Also, in multiplayer this should hopefully allow for only the owning player to hear that sound being triggered. ***Every player should not hear every other player's 'SC_FlipTrick_Start' being triggered, that would be incredibly annoying even if attenuated, trust me.***)

After this the owning `FayCar` 'NS_TailLights_L' and 'NS_TailLights_R' will be activated, this is to give a nice trail effect during tricks while also reusing pre-existing VFX.

Finally, a 'Set Timer by Event' will be started for the 'On Trick Timer End' event, the time is set via a 'Select' node that has been supplied by the 'Flip Trick' event ENUM. The times set here are specifically how long the thrusters will remain active to produce the desired trick. Now this obviously isn't foolproof, but it works decently and consistently enough.



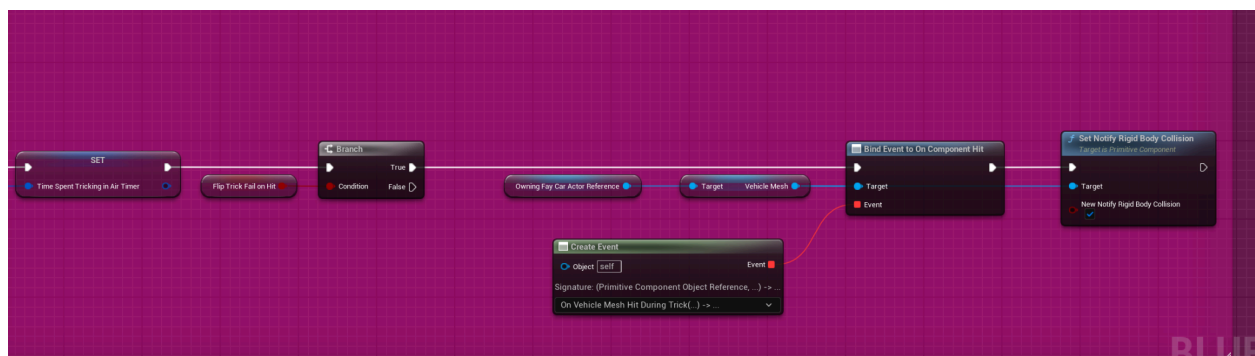
The next output of the sequence is in charge of ensuring the trick being performed is valid, as well as tracking each trick performed.



This section immediately starts with a 'Delay' node of 0.2 seconds, as per the note, the purpose of this delay is an example of "Coyote Time" for the players. Essentially, if this delay wasn't here, and the player attempted to do a front flip right as they were leaving the lip of a ramp, then the trick wouldn't count because they started it on the ground. So this delay allows a moment to pass before ensuring the FayCar is indeed off the ground.

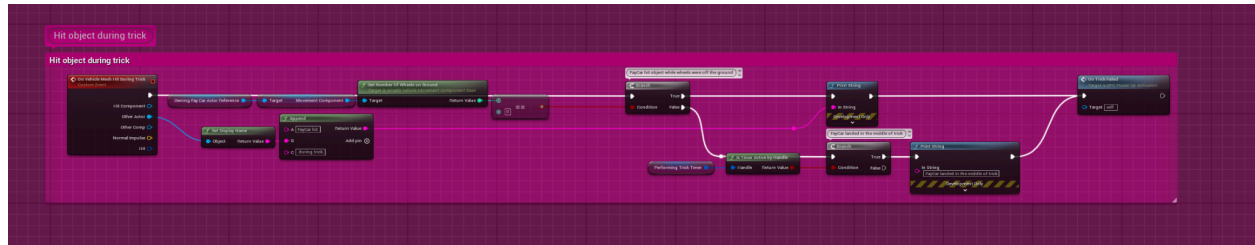
After the delay is the method for determining if a FayCar is off the ground or not. Utilizing the 'Movement Component' of the owning FayCar, there is a function supplied by the 'Arcade Vehicle Movement' plugin that is capable of informing how many wheels of the vehicle are touching the ground. Using this, we plug the result of it into an ==, checking to make sure it is in fact == to 0, so long as that's true the event continues.

Next the 'timeSpentTrickingInAirTimer' will be checked to see if it's NOT already active and running, if it is running however, then it will simply +1 to the 'trickCombo' integer. But so long as it isn't running, then that timer will be started for the 'Tricking in Air' event with a time of 0.01



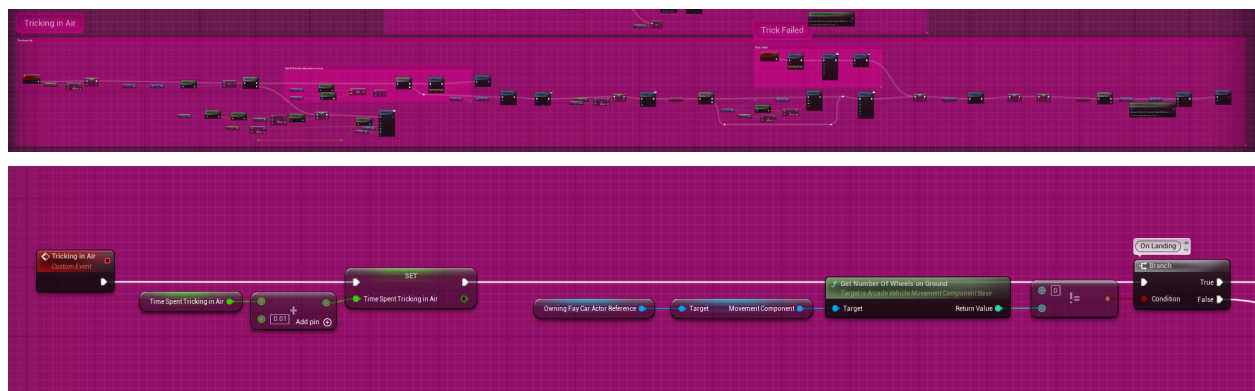
Finally there will be a check to see if 'flipTrickFailOnHit' is set to true, if it is, then it will bind the 'Component Hit' event of the owning FayCar 'vehicleMesh' to the 'On Vehicle Mesh Hit During Trick' event, as well as additionally setting that vehicle mesh to 'Set Notify Rigid Body Collision'. The purpose of this is to have the trick automatically fail if it hits anything during the trick. Now the reason it's set to an optional bool (THAT IS DEFAULTED TO OFF), is because it is abhorredly irritating. Primarily, if the vehicle lands perfectly on its wheels, the mesh might still "Belly Slam" into the ground, registering a hit. Now you may think there's other checks you can do to prevent this, but those checks (such as making a dedicated collision, checking where the hit location came from, etc.) ALL will produce additional issues with other unique hit interactions that could take place during a race. So while option is there, I strongly advise leaving it off, it's much more enjoyable that way.

Hit Object During Trick

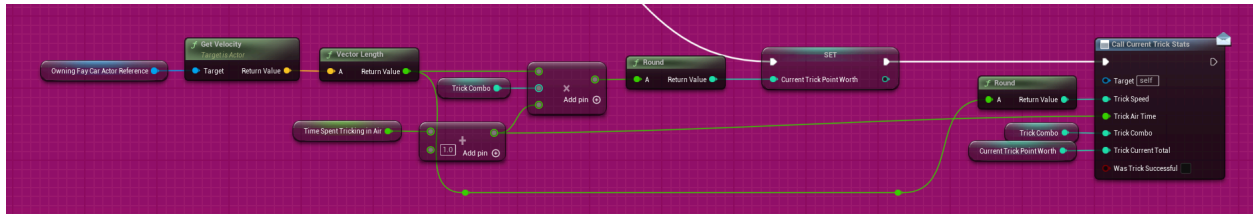


During the 'On Vehicle Mesh Hit During Trick' event, there will first be a check to make sure the vehicle is off the ground during the hit, if they are, then the 'On Trick Failed' event will be run. If they aren't (as in they're landing), then there will a check to see if 'performingTrickTimer' is active, and if it is then the 'On Trick Failed' event will be run (This is because they would be landing in the middle of a trick, you can't land your skateboard in the middle of a kickflip.) Though if the timer isn't running, and the FayCar has at least one wheel touching the ground, then this should mean that they're landing from a trick and the hit will be ignored. **AGAIN**, this shit doesn't work, it should, but it doesn't. My best guess is because something to do with how physics work in Unreal, the tick rate, and how all that plays in to make it seem that the FayCar is hitting the ground before the wheels can be considered touching it.

Tricking in Air



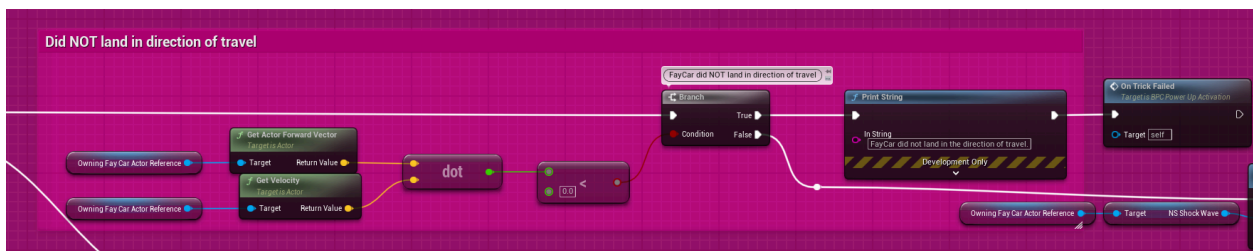
During the 'Tricking in Air' event, the float value of 'timeSpentTrickInAir' will have 0.01 added to its already existing value before setting it once more. Following this will be a continuous check to see if the owning FayCar has landed.



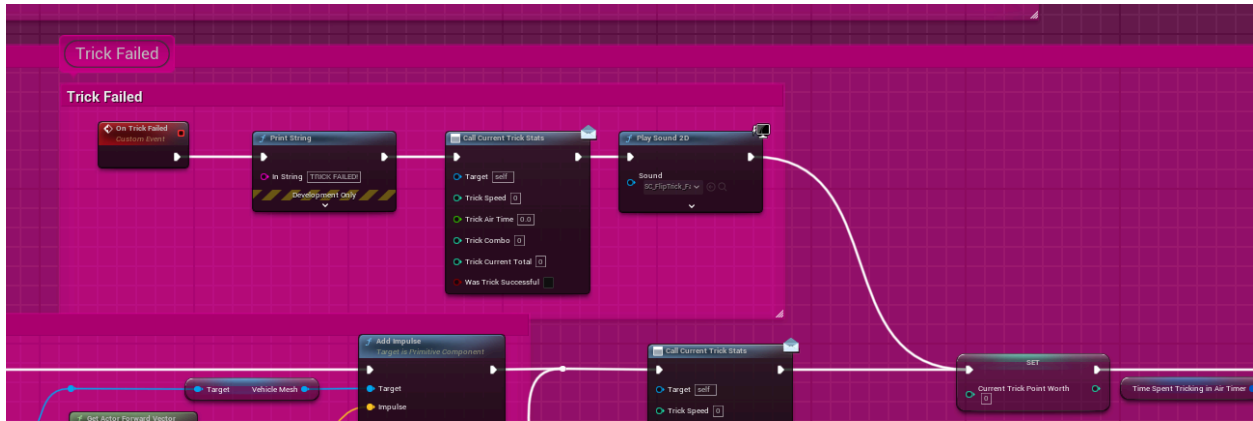
If the owning FayCar is NOT landing, then the trick score will be continuously tallied up. The trick score is accumulated based on the **speed at which they are currently at, the amount of time they spent in air, as well as the amount of tricks performed**. All this information is fed into an event dispatcher called 'currentTrickStats', this event dispatcher is relayed to the 'ControllerDisplay_V2' and serves to inform the player at the bottom of the screen as to how much points their trick is currently worth.

(Note: Keep in mind that a large portion of the point total is based on '*speed at which they are currently at*', what this means if the player slows down during the trick then they will see the score go down, and likely not fully understand why. This was initially a bug as I intended that their max speed achieved during the trick being what is considered. However, after further consideration, I felt that the speed at which the player lands at would be considerably more challenging and worth rewarding.)

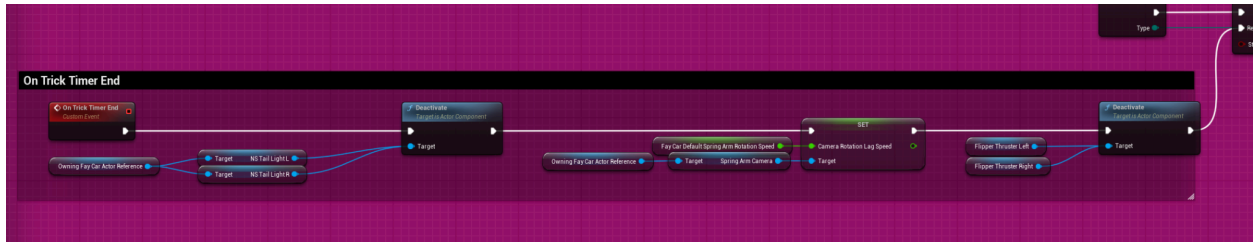
Now if the player IS instead considered landing, then there will be a check to determine if the trick was successful or not.



Upon landing, there will be a check to determine if the player landed facing the general direction they're traveling in. This is done by ensuring that the owning FayCar's forward vector is consistent with their velocity via a dot product, if that value is found to be below 0.0, then that means they landed askewed (sideways, backwards, crooked, essentially any way that would make them lose velocity rather than maintain it on landing).

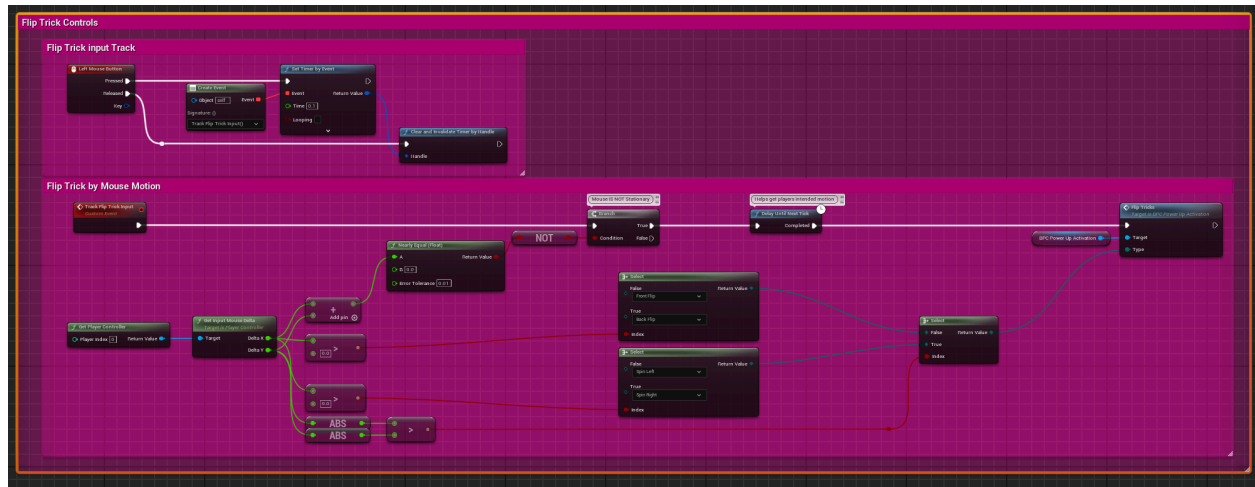


If at any point the 'On Trick Failed' event is run, then all benefits acquired from having completed a successful trick will be bypassed, with the 'currentTricksStats' returning without 'wasTrickSuccessful' being ticked. Finally, the sound cue 'SC_FlipTrick_Failed' will be played in 2D.



When the 'On Trick Timer End' event is run, it will first deactivate both the 'NS_TailLight_L' and 'NS_TailLight_R' VFX (Note if the player is boosting during this, then those VFX will be prematurely deactivated). Next the owning FayCar will have its 'Spring Arm Camera' 'Camera Rotation Lag Speed' set back to its original default value. Finally, the 'flipperThrusterLeft' and 'FlipperThrusterRight' will be activated before the 'Do Once' for the 'Flip Trick' event is reset.

Flip Trick Controls



At the moment, the controls for doing a 'Flip Trick' are being housed within the 'BP_FayCar' blueprint. Essentially what's happening here is that as long as the player is holding down 'Left Mouse Button' the 'Track Flip Trick Input' event will be run every 0.1 seconds.

During this event there will first be a check to make sure the mouse isn't stationary, next is a 'Delay Until Next Tick' node (this helps with getting the player's intended mouse motion). Finally, the trick type is decided based on the mouse motion.

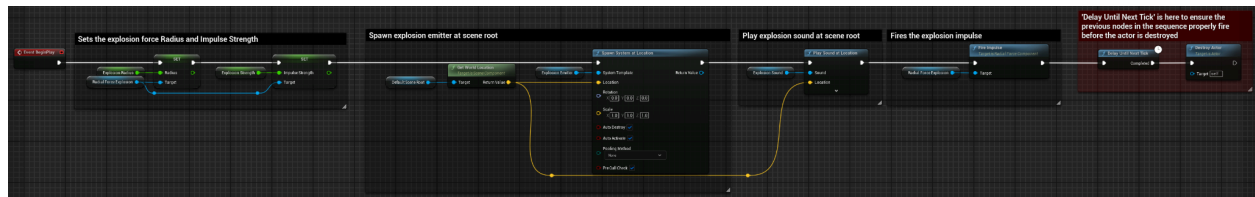
Utilizing the player controller's mouse delta, there will be a check to see which axis movement is greater than the other (up down or left right). Next it will see if that axis value is positive or negative with positive being up/right and negative being down/left.

So in short, how does this work?

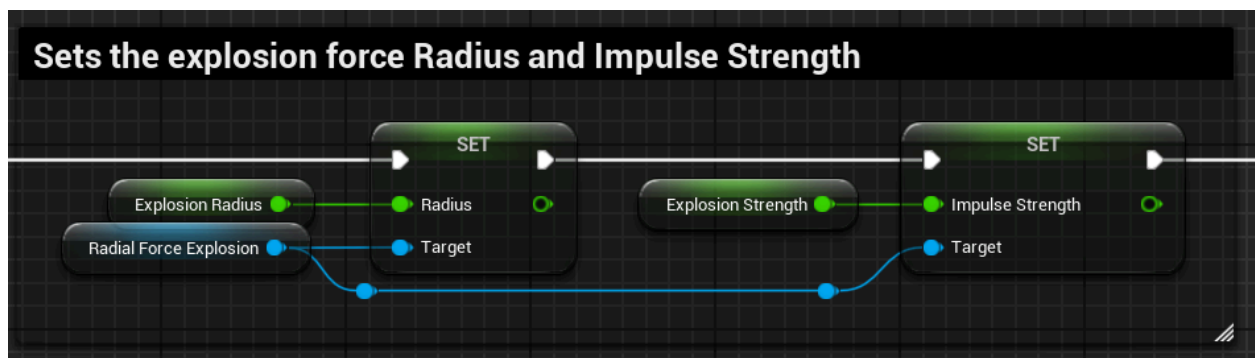
1. ***Hold down left click.***
2. ***Drag mouse in one of the four potential axis.***
3. ***~~*Magic*~~ The car will flip or spin in the direction relative to your mouse's movement.***

BP_CreateExplosion (Deprecated)

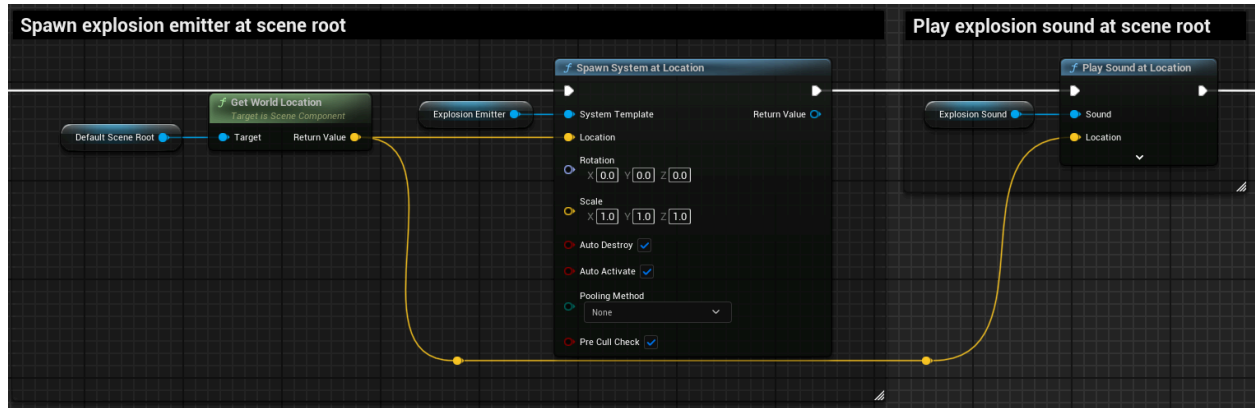
The purpose of this blueprint is to create an explosion at the point in space in which it's spawned.



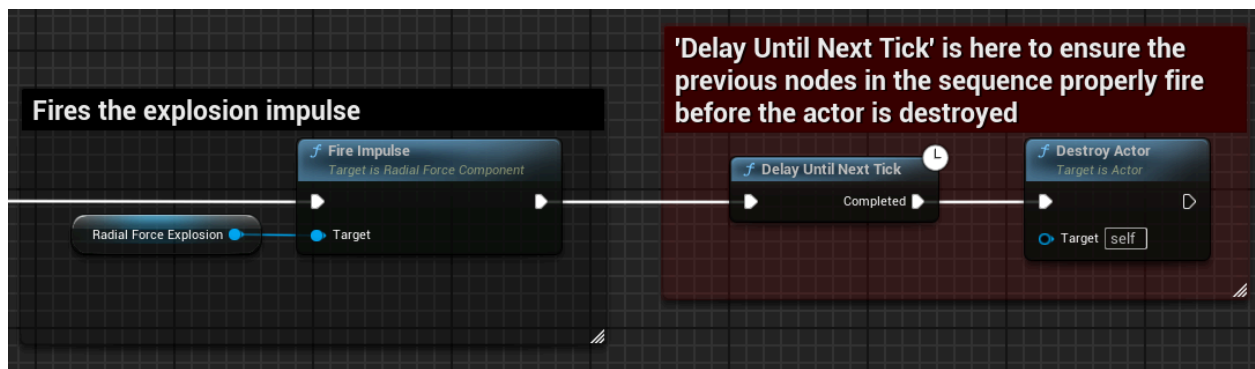
The ideal utilization of this blueprint is for it to be spawned whenever an explosion is required, it has variables such as the radius, strength, VFX, and SFX exposed to allow for ease of customizability at the user's leisure.



On Begin Play the blueprint first takes the supplied 'explosionRadius' and 'explosionStrength', and sets them to the radius and impulse strength of the 'radialForceExplosion' component.



Following this, supplied VFX and SFX are spawned at the 'DefaultSceneRoot' of the 'BP_CreateExplosion', both of which activate automatically.

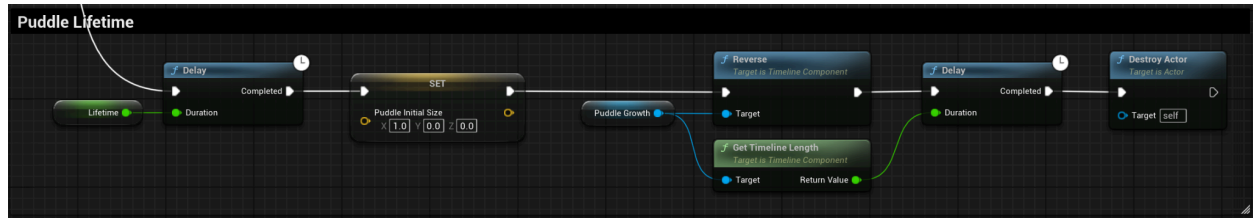


Finally the 'radialForceExplosion' impulse is fired before it's destroyed. The 'Delay Until Next Tick' node is there to ensure the blueprint is fully executed before being destroyed.

BP_SlipperySplash

'BP_SlipperySplash' are puddles responsible for causing the 'FayCar(s)' to experience the slippery effects, primarily called by the 'Paint Bomb' power up.

The 'X' value within this operation is to be treated as the “height” of the actor (the same way the 'Z' value would normally be), if the 'X' value is ever set to a value lower than one then this can result in the decal not being properly seen on surfaces.



Once the 'Puddle Growth' timeline has finished, this will trigger a delay with its input supplied by the float variable 'lifetime' (default value of 5.0), **the 'lifetime' float variable is how long you want the puddle to exist in the world.**

The previously used 'puddleInitialSize' variable is then set to (1.0, 0.0, 0.0) and the 'Puddle Growth' timeline is told to play in *Reverse*. This is a cheap way of getting the puddle to shrink down at the same speed it grew at before it's destroyed, eliminating the possibility of the player seeing it pop out of existence.

On Begin Overlap



'On Begin Overlap' is triggered when the 'overlappingCollision' component is *overlapped*. The component itself initially has its 'generateOverlapEvents' set to false to prevent premature slippiness activations.

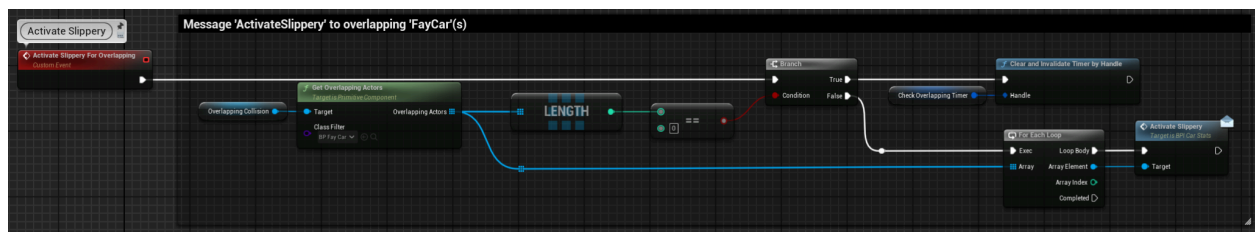
It will first check that the overlapping actor is a 'BP_FayCar', if TRUE then it will send a message of 'Activate Slippy' via the 'BPI_CarStats'. Then it will initiate a 'Set Timer by Event' node for the 'Activate Slippy For Overlapping' Event.



The interface event of 'Activate Slippery' found in 'BP_FayCar' works by setting the angular damping of the 'vehicleMesh' to a negative value. This results in turning with the vehicle to be way more sensitive than normal, emulating a loss of control. After it's set it goes into a 'Retriggerable Delay' node with a value of 3.0. The purpose of this is to allow the vehicles 'Slipperiness' to be prolonged in the event they receive the message of 'Activate Slippery' again. Once the delay is able to complete, the 'vehicleMesh' will have its angular damping set back to 0.0.

Note: If the 'vehicleMesh' ever has it's angular damping default value changed in the future, a float variable could be created that reads the 'vehicleMesh' current angular damping at begin play, then use a get of that float to set the angular damping here to the proper value.

Activate Slippery For Overlapping



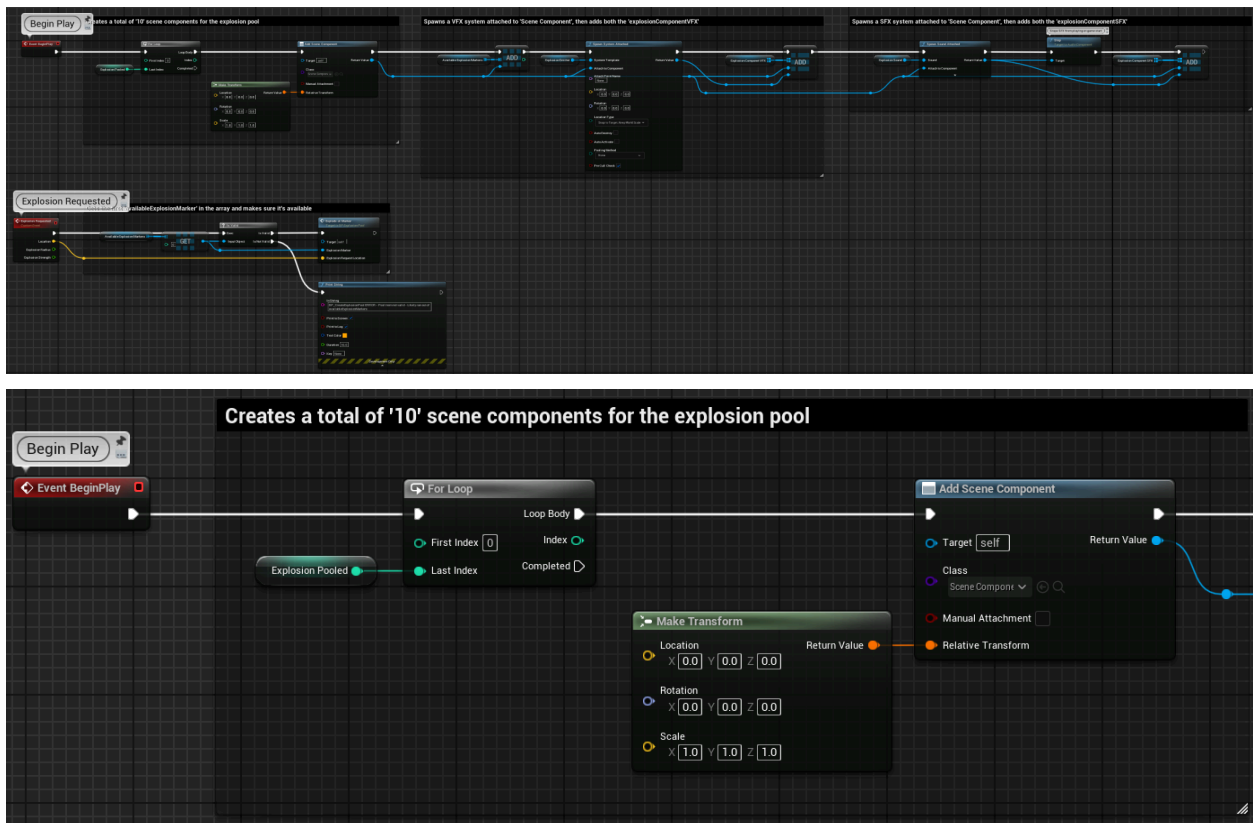
The 'Activate Slippery For Overlapping' event is initiated through the 'Set Timer by Event' node found in 'On Begin Overlap'. Per the inputs supplied to the timer, this event fires every 1.00 seconds on loop.

'Activate Slippery For Overlapping' will first get the total number of overlapping actors found within the 'overlappingCollision' component, if it finds that value is 0 then it will end the 'checkOverlappingTimer' which is responsible for looping this event. If however it finds the value is NOT 0, then it will run each of those overlapping actors through a 'For Each Loop', sending each of them a message of 'Activate Slippery' via 'BPI_CarStats'. In short, if there aren't any actors found within the overlap, then it's going to end the timer to save resources.

BP_ExplosionPool

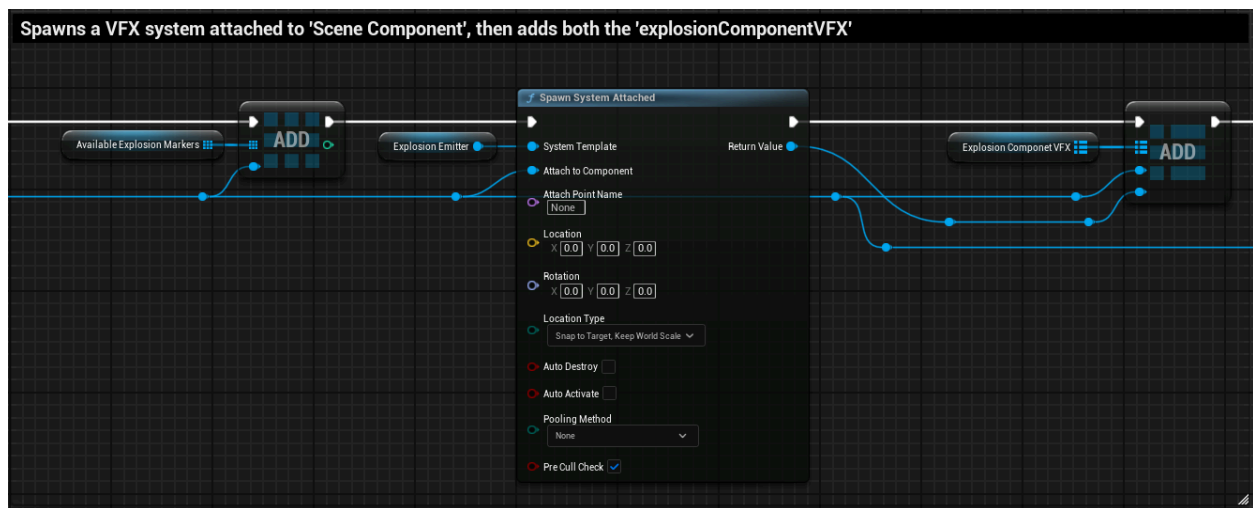
'BP_ExplosionPool' is responsible for storing a set number of explosions within the game world to be called at any given time, reducing the overhead that comes with spawning a 'fresh' explosion.

NOTE: 'BP_ExplosionPool' is automatically spawned in by the 'Begin Play' sequence found in 'BPC_PowerUpActivation' and should NEVER be manually placed within the level.

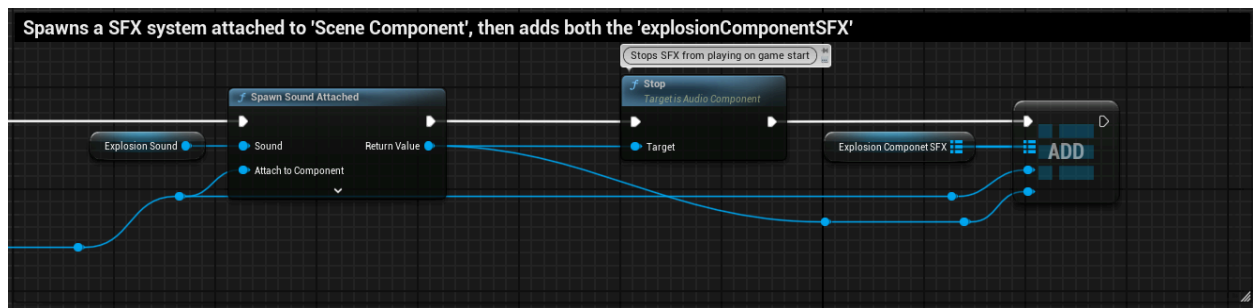


On 'Begin Play' the process of creating the explosives to store begins. A 'For Loop' is run with the first index at 0 and the last index being set by a 'explosionsPooled' integer (default value of 9), **the 'explosionsPooled' integer is how many explosions you wanted pooled within the level.** Also, since numbers start at 0, setting the 'explosionsPooled' to a value of 10 will actually give you a total of 11 explosions, so setting 'explosionsPooled' to 9 will get you a total of 10.

During the 'For Loop' a 'Scene Component' will be added to the actor, these will serve as the moveable markers for which explosions will appear upon.



After creating a 'Scene Component', it will be added to an array of 'availableExplosionMarkers' for later reference. Following this a 'Spawn System Attached' node is used to create the explosion VFX, from which the reference of 'explosionEmitter' will be used to dictate exactly which niagara system shall be implemented. The system will also be attached to the previously made 'Scene Component' and both will then be added to a map variable which will correlate that specific 'Scene Component', with that specific VFX system.

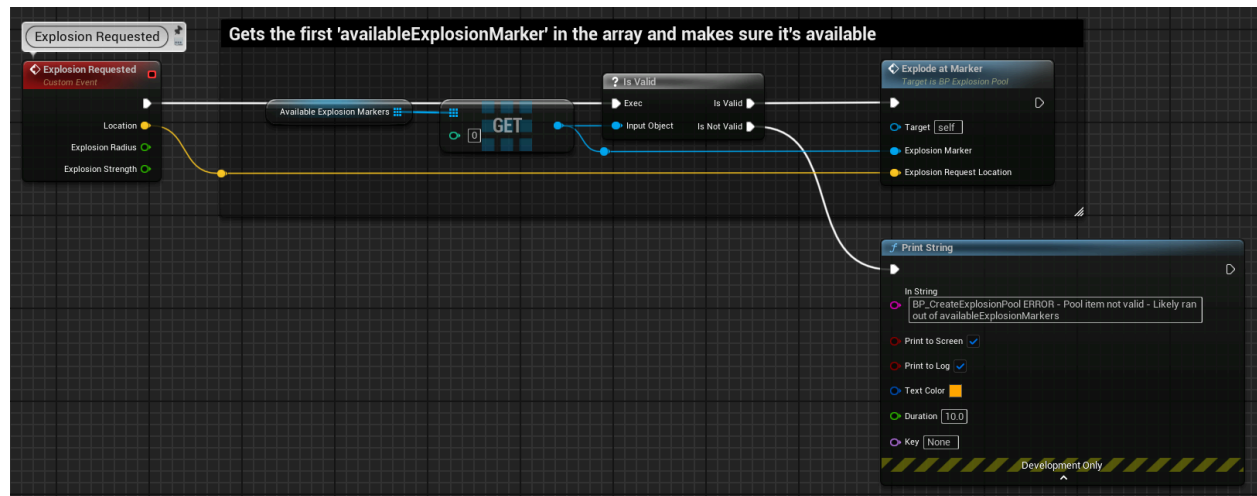


Following this, a similar process will be done for creating the explosion SFX, using 'Spawn Sound Attached' and the 'Scene Component' as the 'Attach to Component'. A 'explosionSound' variable is also used to dictate the sound that will be associated with this explosion.

Immediately following the sound creation however, a 'Stop' node must be used, otherwise those sounds will all simultaneously play when the game starts (No. disabling 'auto-activate' is not an option in the drop down menu of the 'Spawn Sound Attached' node).

Finally, the audio component will be attached to the previously made 'Scene Component' and both will then be added to a map variable which will correlate that specific 'Scene Component', with that specific SFX system.

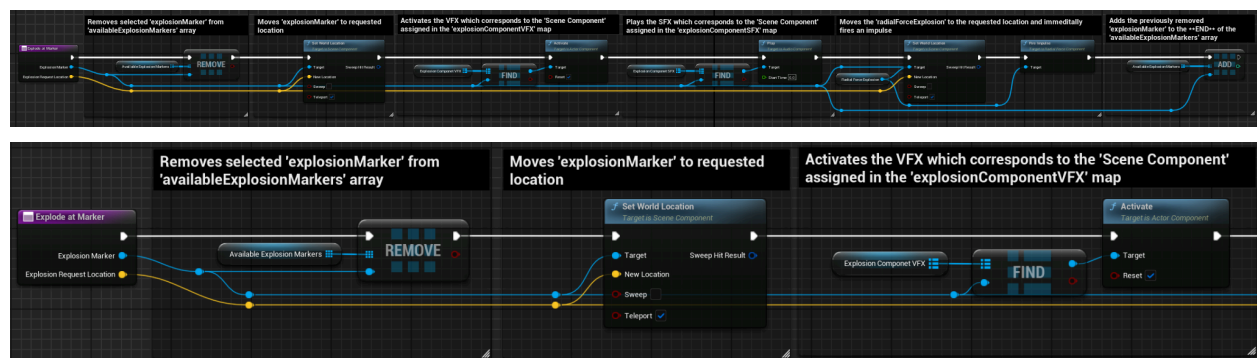
Explosion Requested



On 'Explosion Requested' there will first be a check to make sure the first index of 'availableExplosionMarkers' is valid, if TRUE then it will supply the 'Explode at Marker' event with the 'location' vector supplied by the initial 'Explosion Requested' caller.

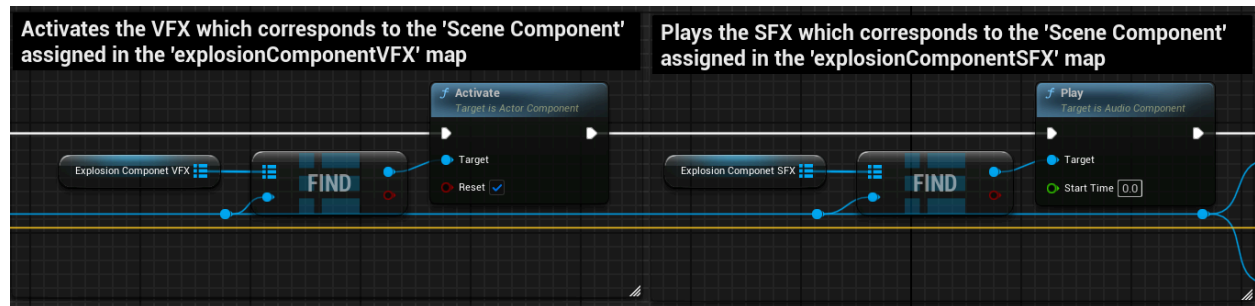
Note: 'Explosion Radius' and 'Explosion Strength' on the 'Explosion Requested' event are leftovers from testing / a feature to be implemented in the future. In theory you would be able to alter the size of any explosion + the amount of force they impulse.

Explode at Marker (Function)



When the 'Explode at Marker' function is run, it will get the previously supplied index 0 of the 'availableExplosionMarkers' and immediately remove it from the array. This causes the next item in the array, which would be index 1, to shift down and become the new index 0 for the next call of the function.

The 'explosionMarker' then has its world location teleported to the point supplied by 'explosionRequestLocation'.



Following this, both the VFX and SFX map variables use the 'explosionMarker' supplied to activate the explosion that correlates with the requested 'Scene Component'.



Finally, the 'radialForceExplosion' is also moved to the requested location and immediately fires an impulse.

Note: There is only one 'radialForceExplosion' inside the actor, additional ones are not spawned per 'Scene Component'. My thinking is that the 'radialForceExplosion' impulse only really needs to be in that area for a frame or so, after that, the objects affected by the impulse already have the information needed to have their physics blown away.

Once this is all complete, the 'Scene Component' that was originally removed from the 'availableExplosionMarkers' array at the beginning of the function is added back to it. When items are added to an array, they are added on top / at the end of it. This is important because once an explosion is requested, it will take 9 additional explosions to

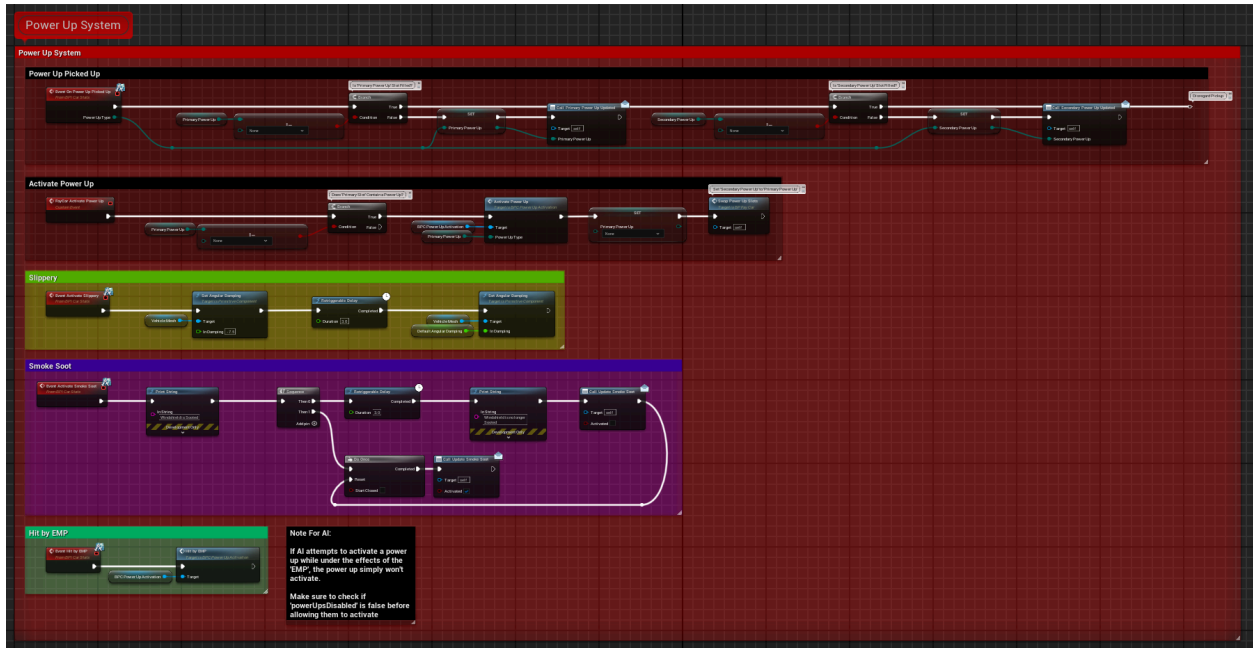
be requested before that first instance will be called again. Given how short the VFX and SFX are in a single explosion, this generally means that the first one will be ready to go relatively soon after it's already been called. Why is this important? Because otherwise there would have to be a delay for every single explosion, a delay that waits till it's complete before adding the 'Scene Component' back to the 'availableExplosionMarkers' array.

To put it bluntly, this system is built on the hypothesis that no more than 10 explosions will ever happen within the game world at a given time, and that in the time an explosion goes off and is requested again, even if it's ripped from its previous existence, so much other shit will be going on that a player likely wouldn't notice the audio fade off of the explosion suddenly being cut off.

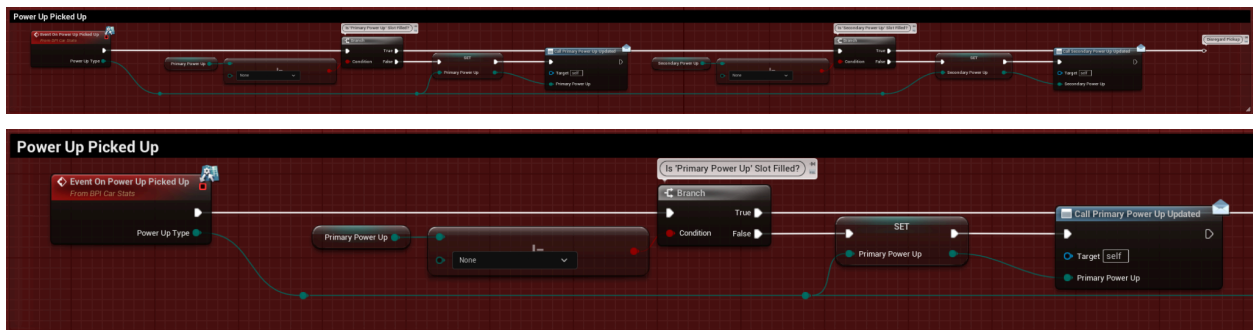
Note: There is one current bug I haven't tracked down entirely yet, if you spam a bunch of explosions, you'll notice explosions happening in the level where the missiles were nowhere near, it's only noticeable if you *really* spam it. I think this is a 'Race Condition' between when the 'explosionMarker' location is set, and when the VFX are triggered. It may be solvable with something as simple as a 'Delay Until Next Tick' be placed after the 'Set World Location' node for the 'explosionMarker', but since this sequence of events takes place within a function, a 'Delay Until Next Tick' can't be used. Perhaps keeping it as an event on the 'Event Graph', or even making it into a macro instead would solve this problem. But again, it happens so rarely and turning the 'explosionPooled' amount up also helps negate the problem.

BP_FayCar (Power Up System)

This section of 'BP_FayCar' holds events which either directly correlate to 'BPC_PowerUpSystem' itself, or aid in applying the effects of certain power ups to the 'FayCar'.



Power Up Picked Up



When the message of 'On Power Up Picked Up' is received to target 'FayCar' via 'BPI_FayCarStats', the variable 'primaryPowerUp' is first checked to see if it contains a value other than 'None'.

If FALSE then 'primaryPowerUp' will be set to the value supplied by the initial call of 'On Power Up Picked Up' before additionally calling the event dispatcher 'primaryPowerUpUpdated'.

If TRUE then the variable 'secondaryPowerUp' is then checked to see if it contains a value other than 'None'.

If FALSE then 'secondaryPowerUp' will be set to the value supplied by the initial call of

'On Power Up Picked Up' before additionally calling the event dispatcher 'secondaryPowerUpUpdated'.

If TRUE then the event will simply end with no variables set or calls made.

Note: The 'BP_PowerupPickup_Base' will still disappear and act as if it has been picked up, though the power up stored within that variable will not be transferred to the colliding 'FayCar'. What this means is that a 'FayCar' can collide with a string of 'BP_PowerupPickup_Base' and while they will all act as if they were collected, that 'FayCar' will only have those first 2 collided with added to their primary and secondary power up slots. It won't change the power ups already held by that 'FayCar'.

FayCar Activate Power Up



(This event is triggered by the 'BP_Racer_Controller' 'InputAction Primary_Ability' event)

On 'FayCar Activate Power Up' is activated it will first check if the variable 'primaryPowerUp' is first checked to see if it contains a value other than 'None'.

If FALSE then the event will end.

If TRUE then the reference of 'BPC_PowerUpActivation' will have its event 'Activate Power Up' run with the variable found in 'primaryPowerUp' set to its 'Power Up Type' input. Following this the variable 'primaryPowerUp' will be set to 'None' before running the function 'Swap Power Up Slots'.

Note: The purpose of this is to automatically set the power up found in the secondary slot (if valid) to the new value of the primary slot, otherwise the 'FayCar' would have to manually swap power ups after activating.

Swap Power Up Slots (Function)

(This event can be additionally triggered by the 'BP_Racer_Controller' 'InputAction Secondary_Ability' event)

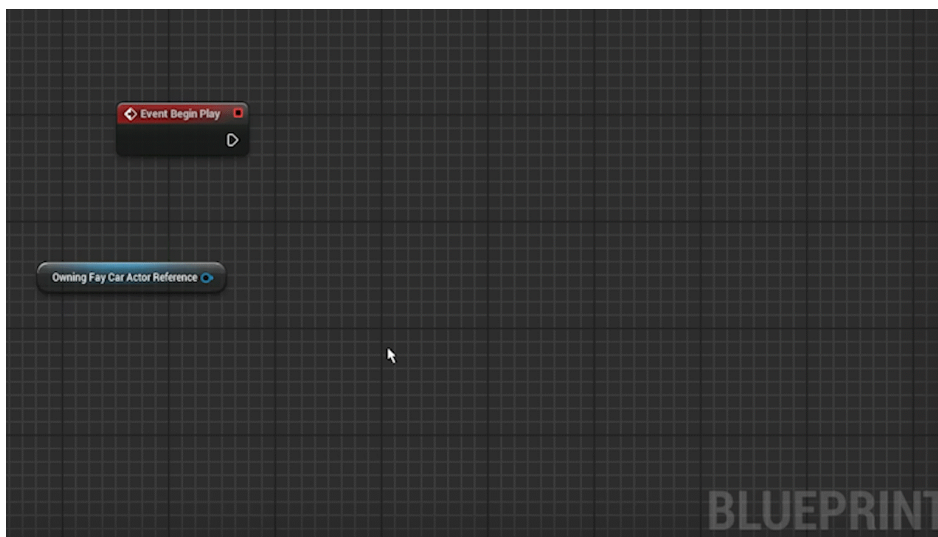


On 'Swap Power Up Slots', a local variable of 'powerUpInSecondarySlot' will be set with the variable of 'secondaryPowerUp'.

Next, the variable of 'secondaryPowerUp' will be set with the variable of 'primaryPowerUp' before calling the 'primaryPowerUpUpdated' event dispatcher.

Finally, the variable of 'primaryPowerUp' will be set with the local variable of 'powerUpInSecondarySlot' before calling the 'secondaryPowerUpUpdated' event dispatcher.

How to Use with Outside Systems



To update systems outside of 'BP_FayCar' as to the change in values to both 'primaryPowerUp' and 'secondaryPowerUp':

First get a reference to the 'BP_FayCar'

Next pull off of that reference and get 'Assign Primary Power Up Updated', this will create a 'Bind Event to Primary Power Up Updated' with a custom event attached to its 'Event' input.

Plug the 'Bind Event to Primary Power Up Updated' into your begin play (or equivalent), you want this assigned early on.

Now anytime that the event dispatcher of 'primaryPowerUpUpdated' is run by 'BP_FayCar' the custom event attached to your 'Bind Event to Primary Power Up Updated' will be automatically run.

Repeat the same steps for 'secondaryPowerUpUpdated'.