BP_Spoonapult

Blueprint Author: Justin Cheng, Teddy Walsh, and Mustafa Elfayumi
Created: Summer 2023
Last Updated: April 20, 2024

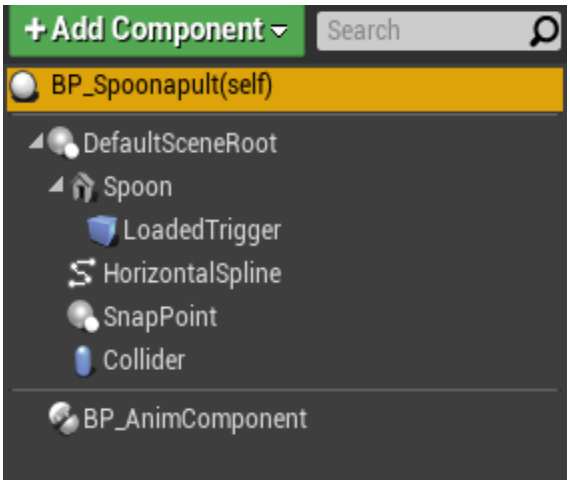
Blueprint Description:

A central level item that can load in a single item (object with the “Launchable” tag) which snaps into place. When animated, the spoon should play a brief animation and the object will use splines and math to travel in an arch, reaching the lego wall at the end, damaging it if it also has the ammo tag, otherwise it should bounce off.

Related Blueprints:

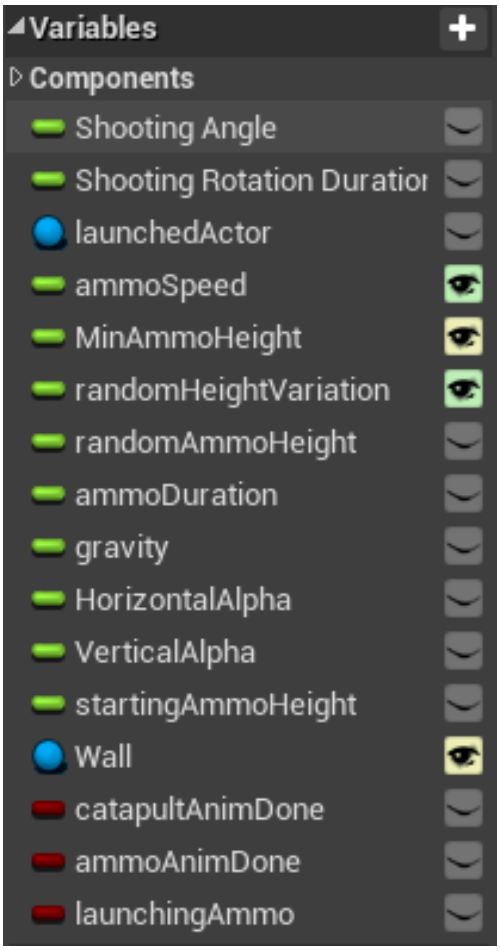
- BP_LegoWall (works with this blueprint/has references to this blueprint)
- No children or parents (a single orphan, so sad)

Components:



- **Spoon:** Static Mesh of the spoon catapult
    - **LoadedTrigger:** Box collision trigger for checking when objects are loaded into the catapult
- **HorizontalSpline:** Spline that launched objects will move along
- **SnapPoint:** Location objects will snap to when being placed in the spoon
- **BP_AnimComponent:** Component that makes the object animatable (See BP_AnimComponent documentation)
- **Collider:** A capsule collision used with BP_AnimComponent

Variables:

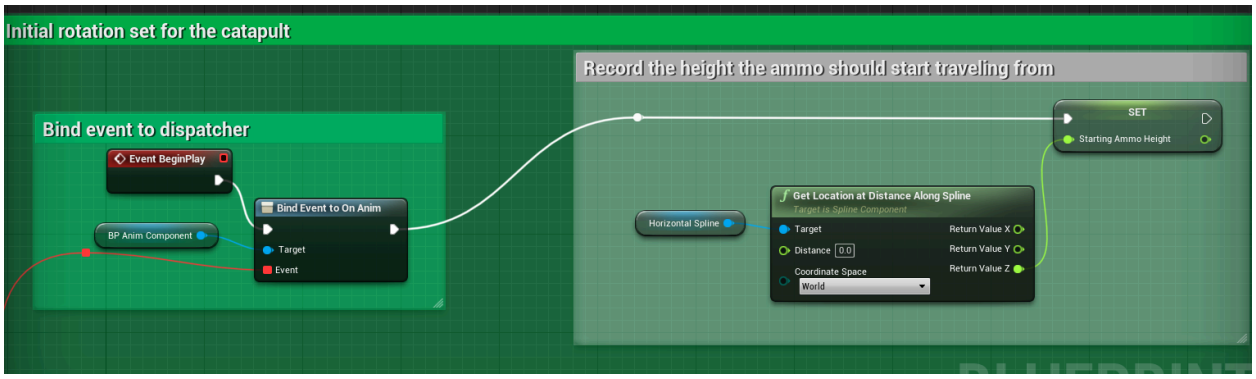


- **Shooting Angle (float):** Default to 30 degrees. How far the spoon model should rotate from its resting position in the peak of its animation
- **Shooting Rotation Duration (float):** Default to 1.5 seconds. How long the rotating animation should take
- **Launched Actor (object):** Object reference to the 1 object that was launched by the catapult and is now following the spline. Set when the catapult attempts to launch
- **ammoSpeed (public float):** 0 by default. If left at 0 then this will not be used and instead gravity will be used to calculate the speed. If it is set then this will overwrite the calculation using physics.
- **MinAmmoHeight (public float):** How high the ammo will reach in its arc. Minimum because it will add somewhere between 0 - randomHightVariation (so if min is set to 1000 and variation is set to 500, the actual hight will be between 1000 and 1500)
- **randomHeightVariation (public float):** The max increase to the height that the ammo may have when it launches
- **randomAmmoHeight (float):** Stores the randomly generated height somewhere between (MinAmmoHeight) - (MinAmmoHeight + randomHeightVariation)
- **Gravity (float):** Constant 981. Used to calculate how long it would take the ammo to fall
- **HorizontalAlpha (float):** Float between 0 and 1 that represents how far along the horizontal spline the ammo should be. 0 is at the catapult (beginning) and 1 is at the wall (the end). Moves in a linear rate from 0 to 1 as the Movement timeline plays
- **VerticalAlpha (float):** Float between 0 and 1 that represents how high up the ammo should be. 0 is at the level of the catapult (startingAmmoHeight) and is at its peak (randomAmmoHeight). Moves in a parabola from 0, then to 1 at the vertex, then back to 0

- **startingAmmoHeight (float):** Records the height of the horizontal spline start to know what is the base height for the arc
- **Wall (public BP_LegoWall):** Object reference to the wall this is firing at so it can send messages to the wall when its hit
- **catapultAnimDone & ammoAnimDone (both bools):** Records when the catapult is done rotating and when the ammo has hit the wall respectively. Both need to be true for the catapult to be able to be fired again.
- **Launching Ammo (bool):** True if there is a piece of ammo (or just a launchable item) traveling in an arc. Used for ammo snapping logic.

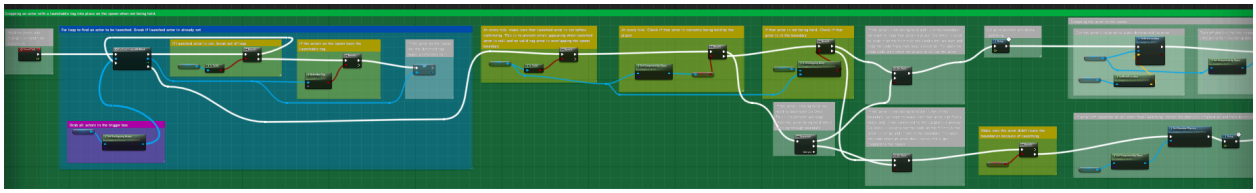
Event Graph:

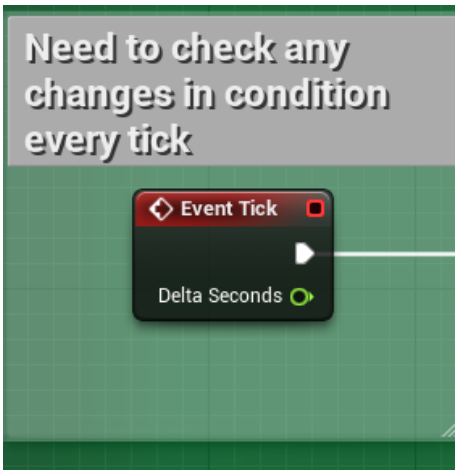
Event Begin Play



Upon loading the catapult, bind our Launch event (described later) to the Anim Component so that when players animate the catapult, the Launch event is called. Then set the ammo starting height based on the z value of the horizontal spline in the world.

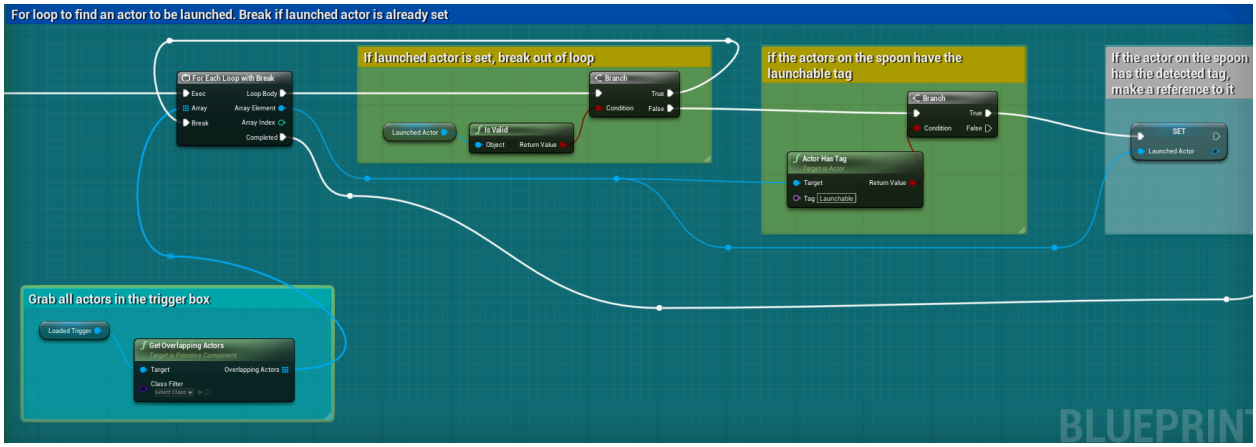
Event Tick (Snapping Ammo Into Place)



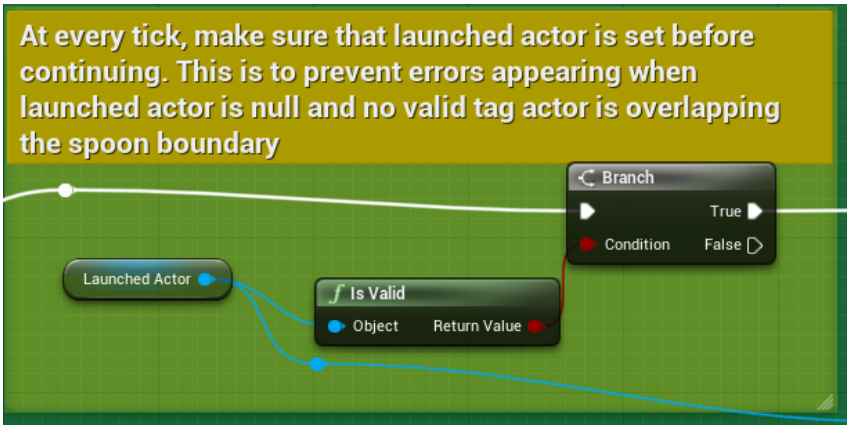


Using event tick to look for any changes in the catapult snapping area. Currently this is only possible way to constantly update and check for changes in the snapping location box and if an actor in that location is being held or not. There was an attempt to use OnBeginOverlap and OnExitOverlap to reduce processing power, however, this would cause problems since that only happens on the tick it enters/exits. One example would be if the object enters

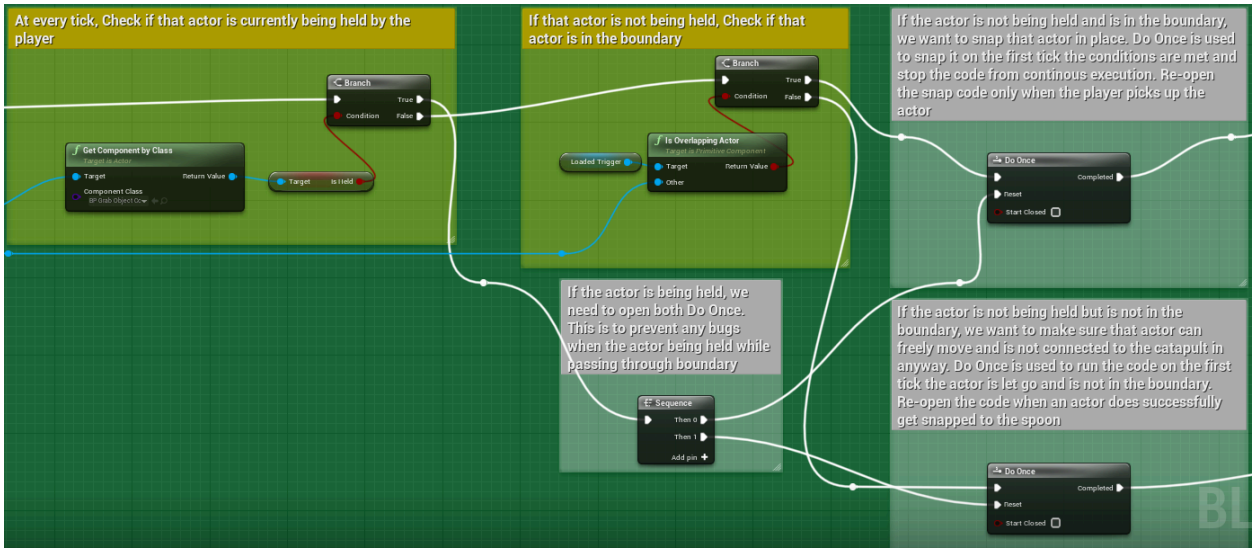
the bounding box while being held and then lets go. OnBeginOverlap would ignore the snap because the object was being held while entering the box or it would have to snap the object while the player is still holding onto it (which we don't want).



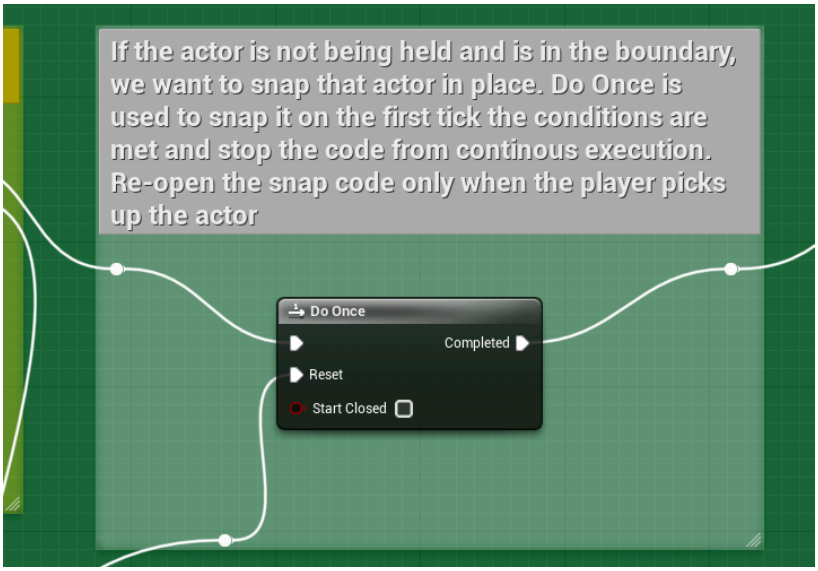
At every tick, the first thing we do is to make sure we have a reference to an object that is going to be launched. Using a for each loop, we grab every actor that is currently overlapping with the Loaded Trigger Collision Box. First we check if we already have a reference to an actor. If so, we break out of the loop because we want to keep that reference until that actor leaves the box. This also means that we will only snap the first viable actor in the array of overlapping actors. A viable actor would need to have the actor tag “Launchable”. If it does we, we set a reference to it and then break out of the loop. Otherwise, we iterate on the next overlapping actor.



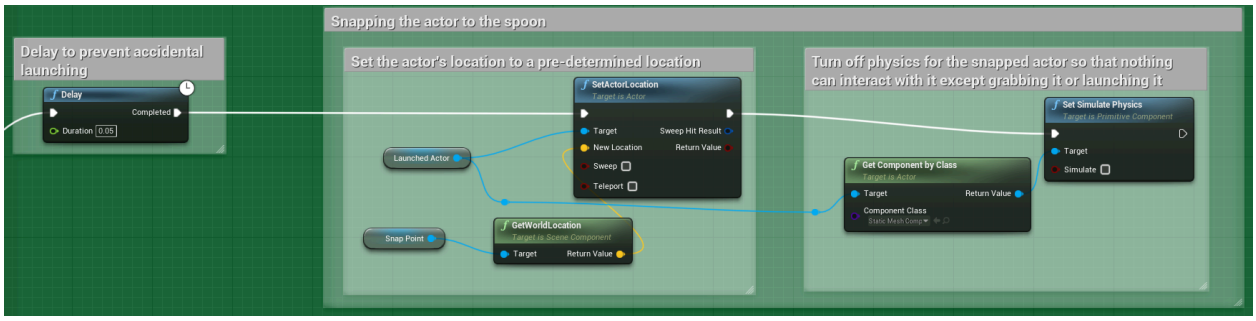
Once the loop is completed/broken, we start our checks to see the condition of our reference actor. But first, we check to see we actually have a reference to a valid actor. If not, there is no point of executing following code, allowing us to prevent bugs but also reducing runtime processing



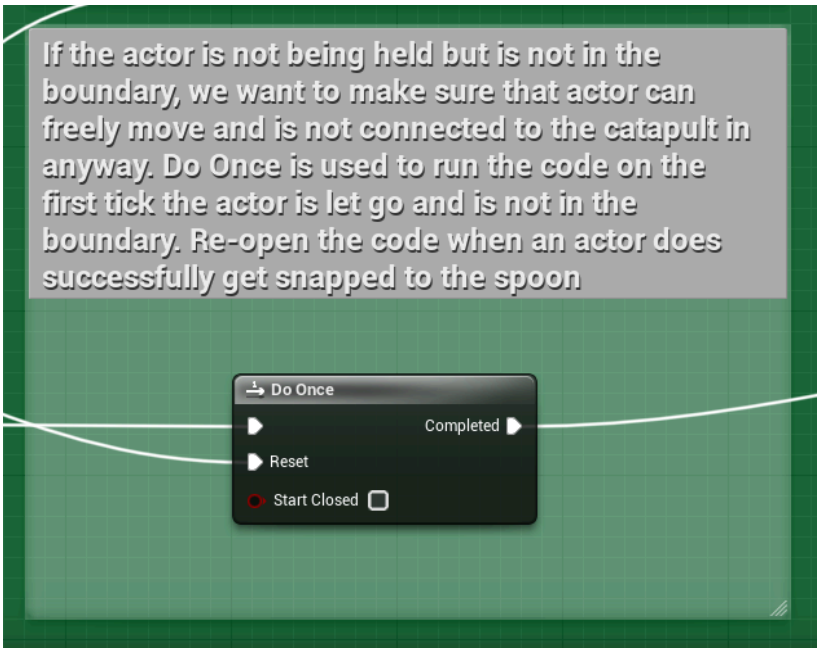
If we have a valid reference, then we need to check if that referenced actor is currently being held by the player. We ONLY want to snap the object when it's in the boundary and not being held by the player. When the player is holding the object, we can use this time to reset the following Do Once codes to prevent bugs. If the player is not holding the referenced actor, we need to check if the actor is not in the boundary.



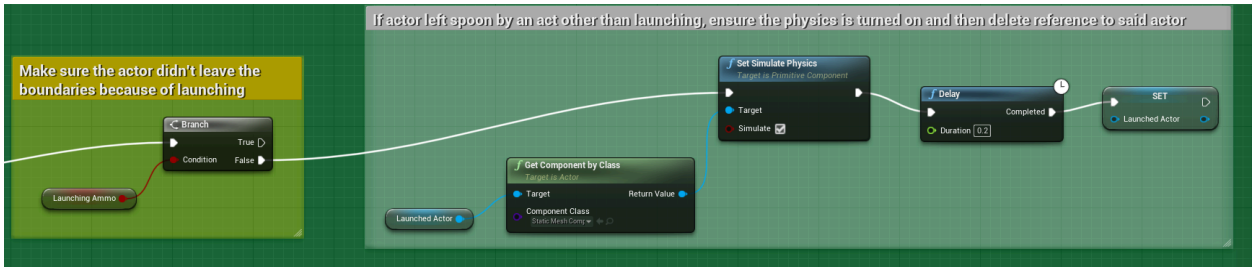
If the actor is in the boundary, we run through code to snap the object. We use a Do Once here because we want the snapping execution only to happen the first time to reduce processing power and prevent bugs.



Before we snap the actor, we put a small delay just to make sure there are no accidental launching from the player letting go of the trigger buttons (Has to do with the interaction between our control scheme and how we animate things. See BP_AnimComponent for more information). To snap the actor, we set the actor's world location to a predetermined point that is on the spoon. Then, to prevent interactions other than picking up and launching, we turn off the physics on the actor.

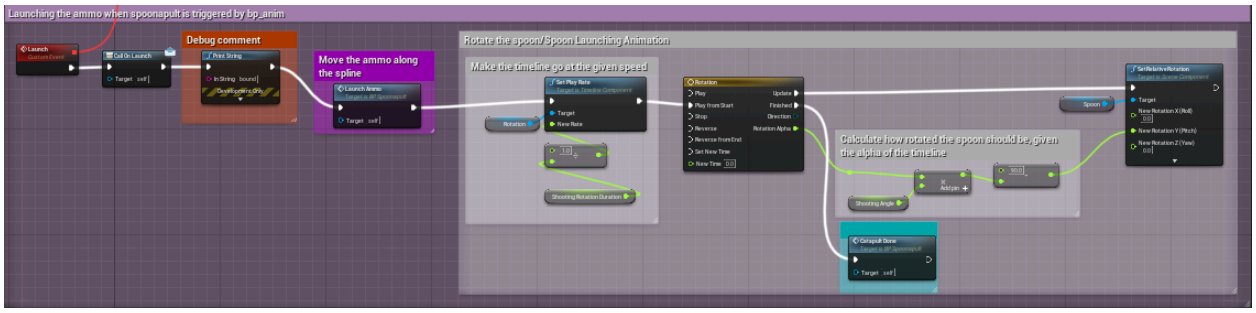


If the actor is not in the boundary, we run through code to ensure that the object is back to normal and not referenced anymore. We use a Do Once here because we want the execution only to happen the first time to reduce processing power and prevent bugs.

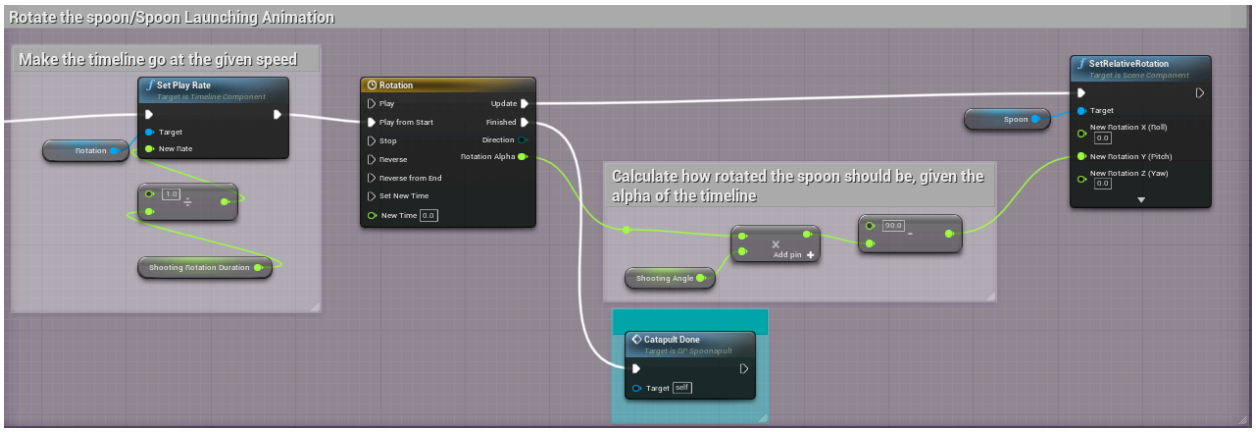


Before losing the reference to the actor, we check to see if the actor is not in the boundary because of the Launch event (below). We want to keep the actor as is if the launch event caused the exit of the boundary. If the actor left the boundary for other reasons, we turn the physics back on and then set our reference back to null. This way the actor is now back to being a normal object and no longer connected with the catapult.

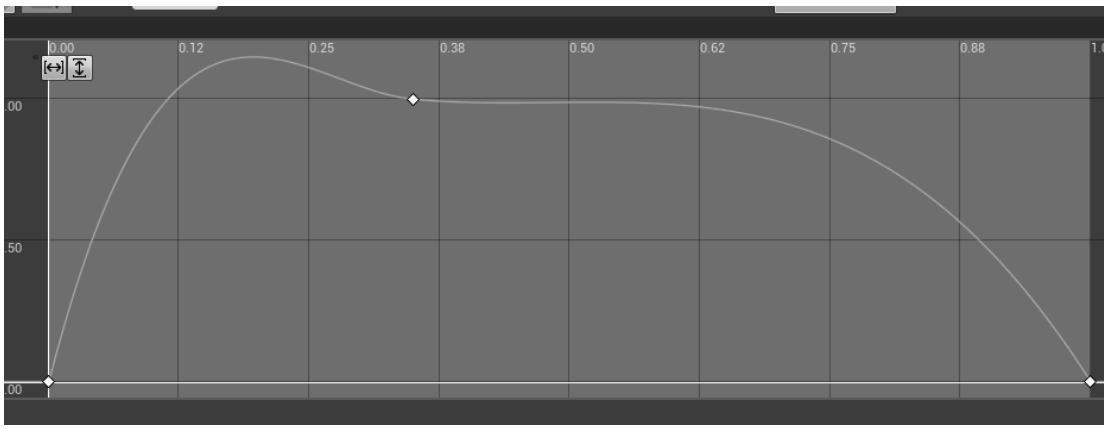
Event Launch



This event is binded to the BP_AnimComponent so that whenever the player animates the Spoonapult, it would call this launch event. First , this event calls the Launch Ammo event to move the referenced actor along a spline (described later).



Then, we play an animation to rotate the spoon. We set the animation duration and the play the rotation animation (described below). After the animation is done, we call the Catapult Done event.

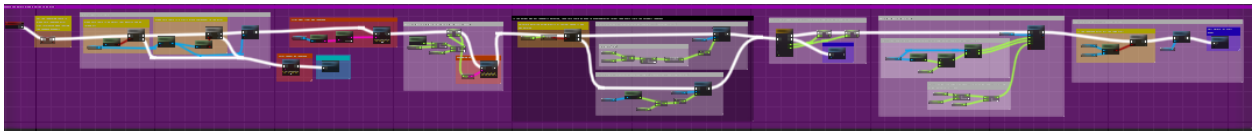


Here is the Rotation Timeline. This centers around the Rotation Timeline, which (by default) over 1 second, outputs alpha starting at 0, going a bit over 1, then resting at 1 for a moment, then going back to 0. The alpha represents how rotated the spoon should be with 0 being fully flat and 1 being the shooting angle (30 degrees by default)

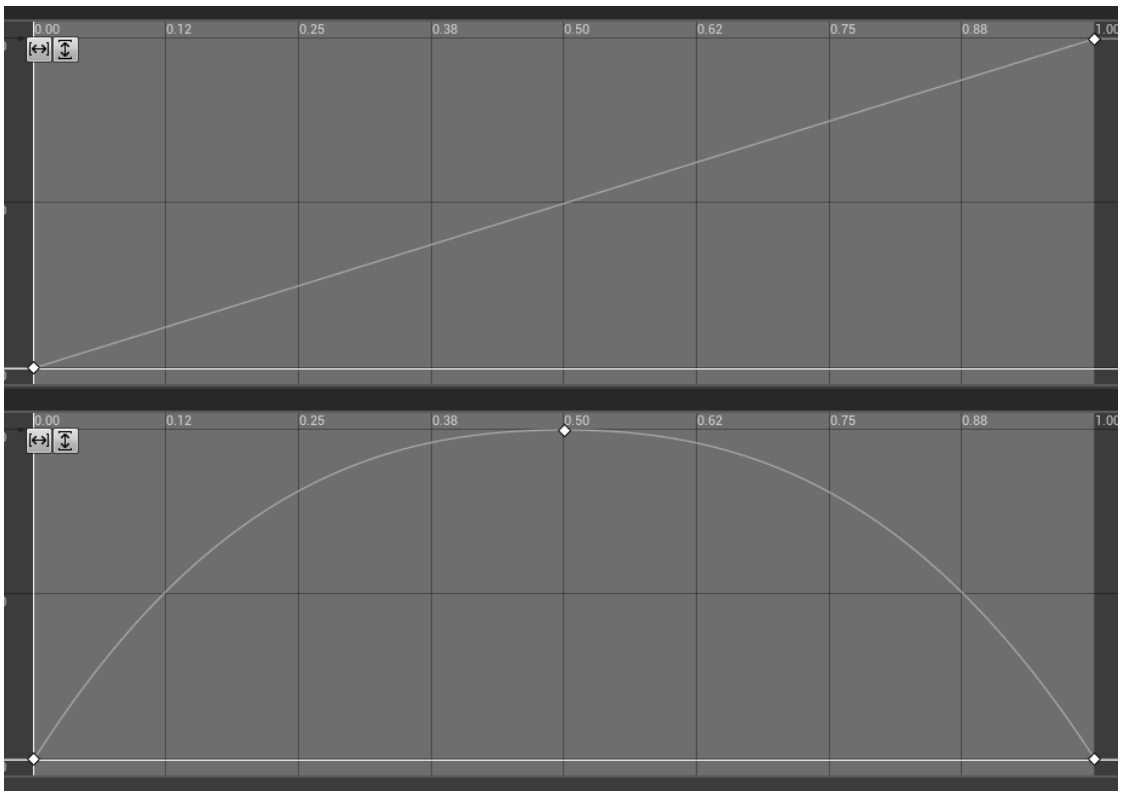
1) The playrate of the timeline is adjusted so that instead of lasting 1 second, it lasts for the shooting rotation duration (1.5 seconds by default).

2) The timeline is started.
    a) The timeline is handmade by Teddy Walsh and is likely temporary until an actual animator can make something nicer.
3) Every frame the timeline is updated, it outputs the alpha value, which is used calculate how rotated the spoon should be
    a) Note that the spoon laying down flat is rotated at 90 degrees. The spoon fully upright is rotated at 0 degrees. The shooting angle of 30 degrees means that it rotates 30 away from the resting angle of 90 towards the upright angle of 0 (so the actual rotation on the mesh at the height of the timeline will be 60).
4) The spoon mesh's rotation is updated with the current rotation value
5) When the timeline finishes, mark that the catapult is done with its animation (used in logic later to make sure that both the spoon is done animating and the ammo is done flying (if there was any ammo) before the spoon can be animated again.

Event Launch Ammo



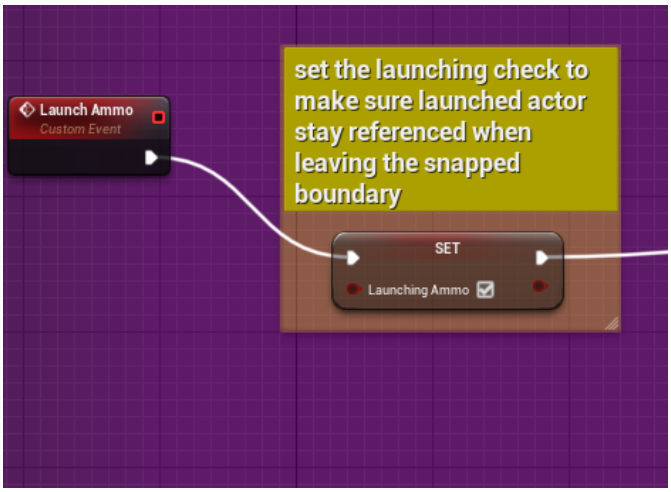
Here is where we launched the ammo along the spline path.The spline path timeline is described below.



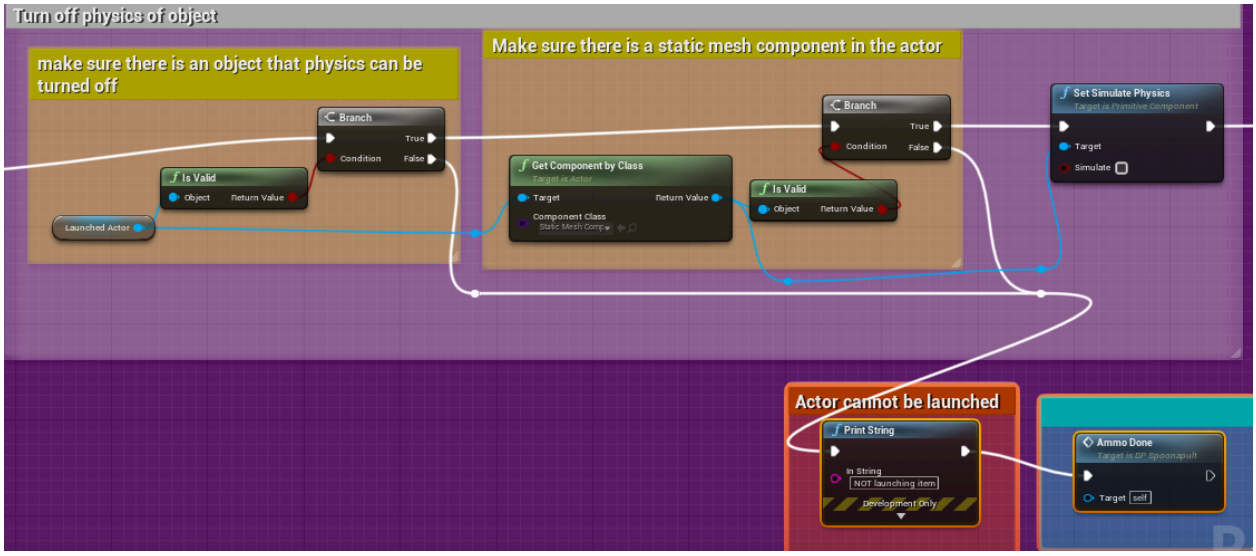
Movement Timeline. (Top: Horizontal Alpha. Bottom: Vertical Alpha).

This uses much of the same logic as rotating the spoon (detailed in the Launch event) by updating a transform using the output of a timeline, but in this case it also uses a spline to get a location. It also uses math and physics algorithms and concepts (not an in-game physics simulation) to figure out how fast it should be traveling (more on that later).

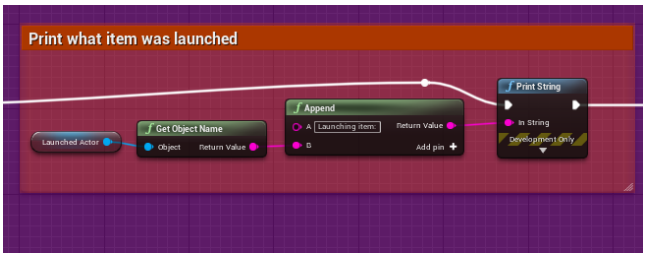
The logic behind this is based on physics. An object traveling in an arch starts with an upwards and forwards velocity. The only acceleration acting on the object is gravity pulling it down (ignoring air resistance, which we are). So the horizontal velocity remains constant since there is no acceleration acting on it, and the vertical velocity is a parabola, starting at 0 going upwards and slowly accelerating downwards until it reaches the base height (0) again. This is why the graph has 2 outputs, a linear line (horizontal) and a parabola (vertical).



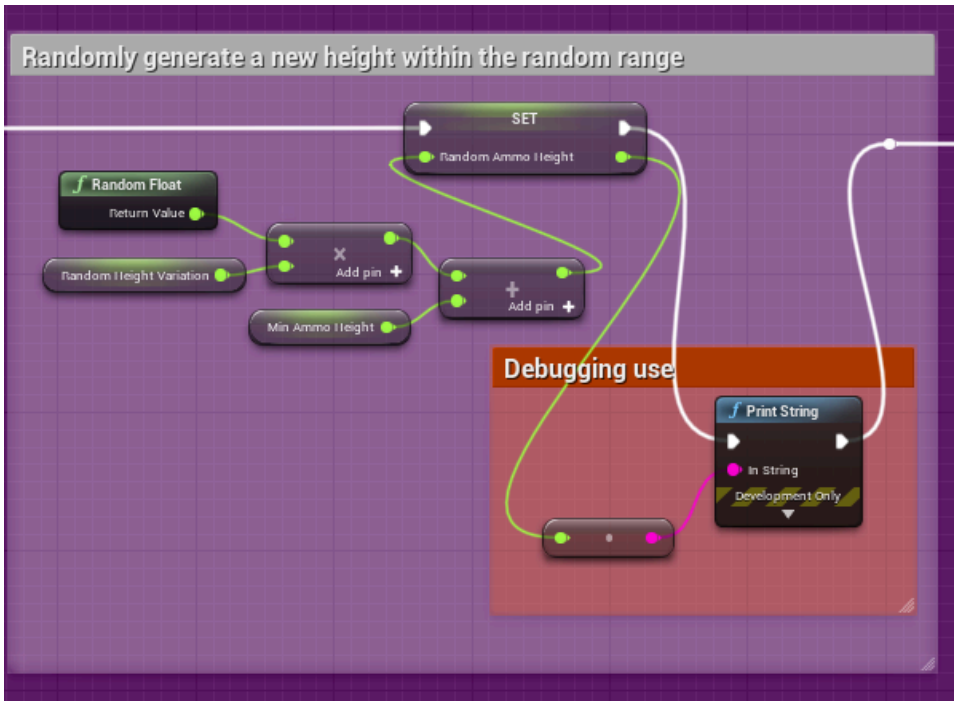
1) Mark that there is ammo launching (used for object snapping logic)



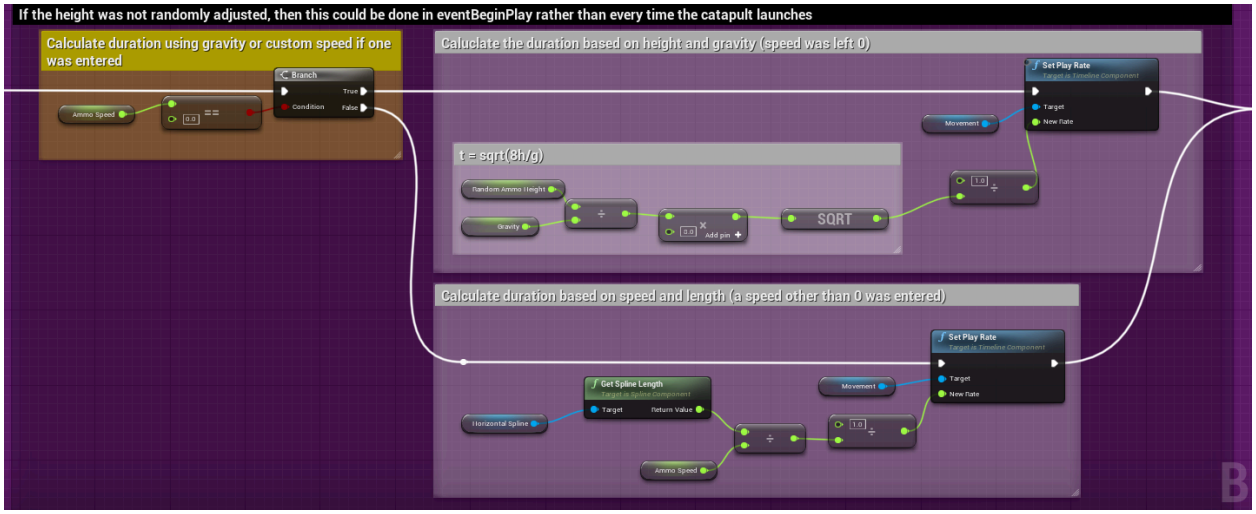
2) Check if there is a launchable object in the catapult (saved as Launched Actor)
    a) If there is: turn of its physics and continue
    b) If there isn't: End this process



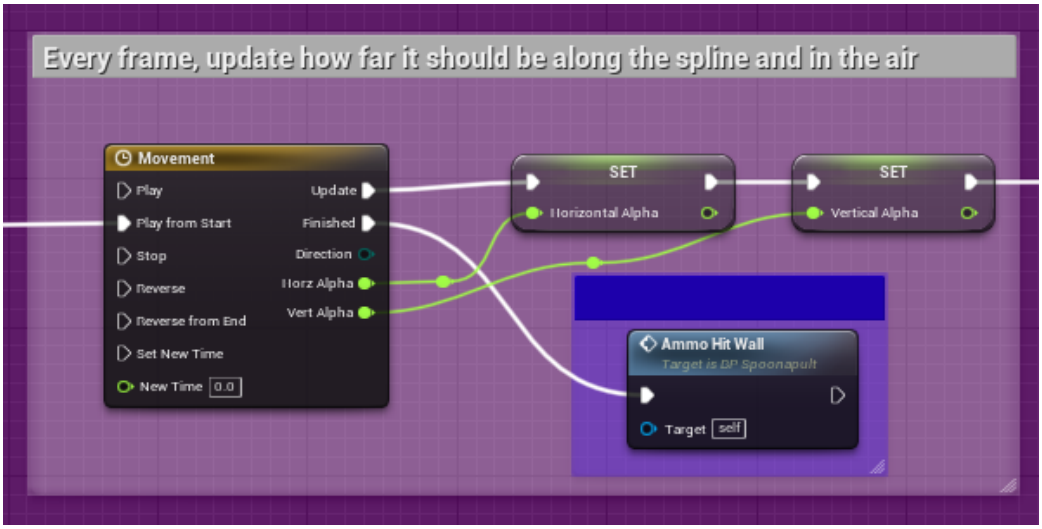
3) Print what item is launched (just used for debugging)



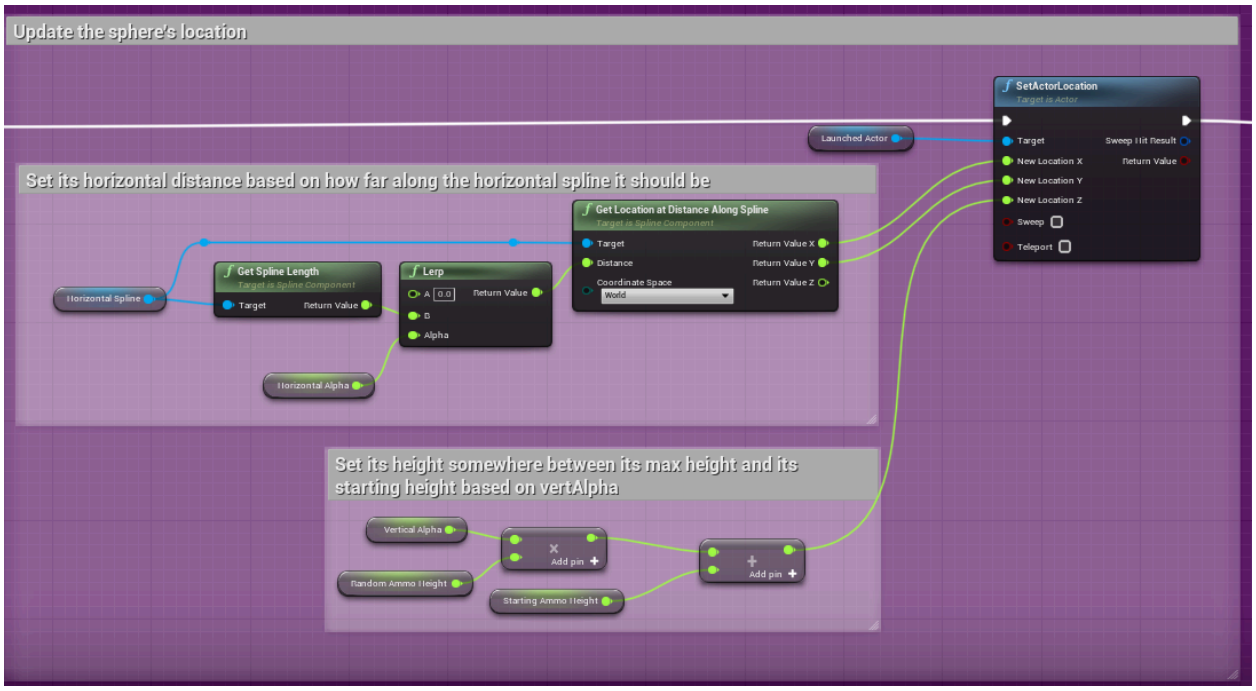
4) Randomly generate a height for the object to be launched at its peak. Somewhere between Min Ammo Height (public float variable) and Min Ammo Height + Random Height Variation (public float variable) and save it as Random Ammo Height.+



5) Calculate the speed of the timeline by calculating the duration the ammo should be in the air.
    a) If a custom value for ammoSpeed was entered, use that and the length of the horizontal spline to see how long it would take to reach the end going at that speed
    b) If no custom value for ammoSpeed was entered (it was left as 0), use physics to see how long it would last in the air. This was derived from using an object at the given height having 0 vertical velocity (as it would at the parabola) with only gravity accelerating it, how long would it take to reach the ground. We then multiply that by 2 because it would take exactly the same amount of time to reach the peak from the ground (same total change in velocity with the same constant acceleration).
    c) These calculations could be done on EventBeginPlay rather than every time the catapult is launched if a new random height wasn't generated every time.

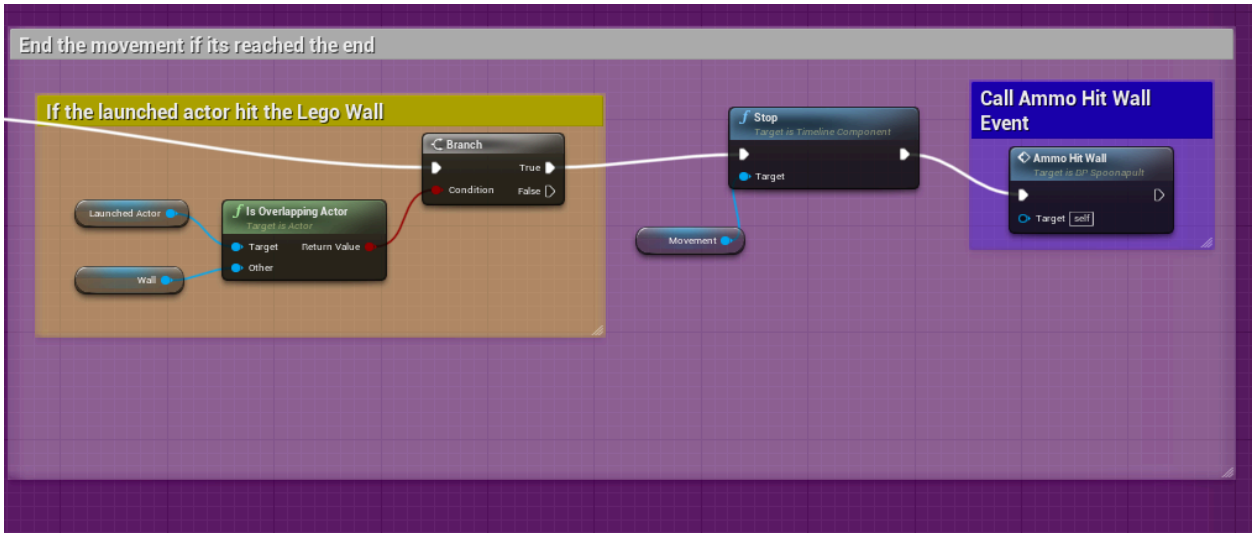


6) Start the timeline
7) Every frame the timeline is updated, record the horizontal alpha and vertical alpha and use it to calculate the new position of the ammo



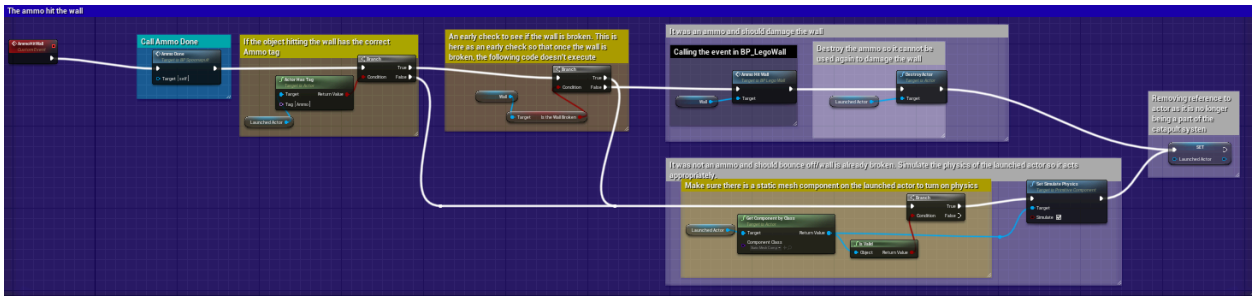
a) Vertical alpha is used to lerp between the origin point of the horizontal spline and the end point, which is used for the X and Y of the position
b) Horizontal alpha is used to lerp between the base height and the max randomly generated height of the arch, which is used for the Z of the position.

8) Update the ammo's position.

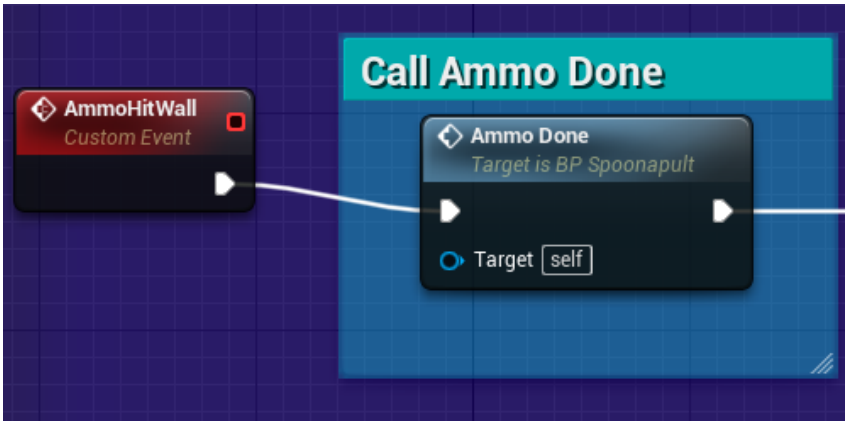


9) Check if the ammo is colliding with the wall this frame. If it is, run logic for colliding with the wall. This is also called if the ammo reaches the end of the spline without colliding (safeguard in case the random height somehow caused the ammo to miss the wall).

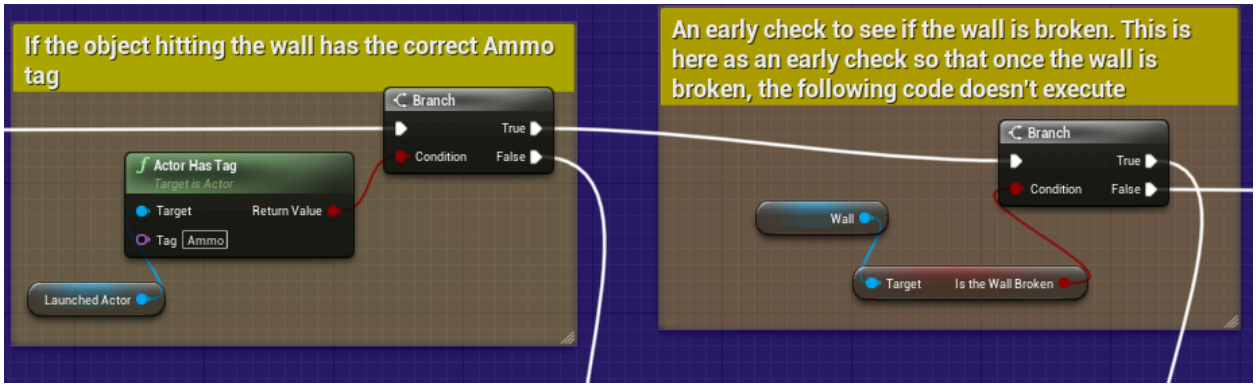
Event Ammo Hit Wall



This Event handles when the launched actor has finished traversing through the spline and now either does damage to the Wall or becomes a regular actor in the scene



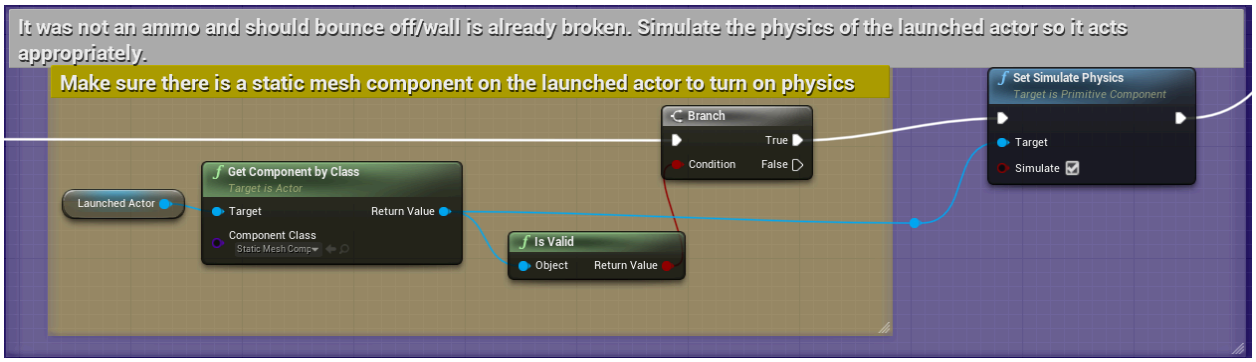
1) Record that the ammo has reached the end (this is used in logic to make sure both the catapult and ammo are no longer animated before letting the catapult be animated again).



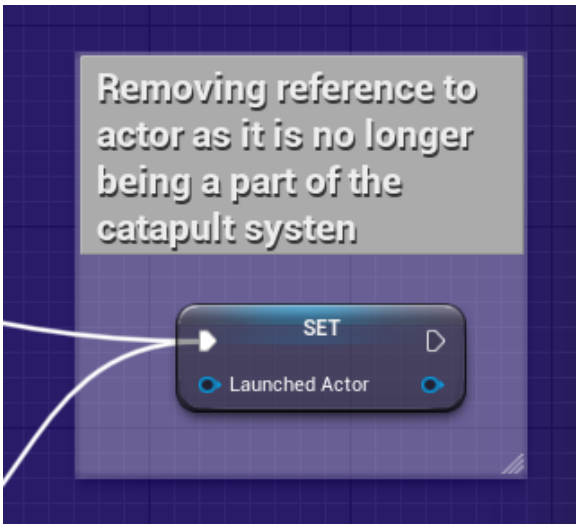
2) Test if the ammo has the “Ammo” tag and the wall isn’t already broken

a) If both are true, tell the wall that it's been hit by calling Ammo Hit Wall within the wall blueprint (this is covered here),

destroy the ammo.

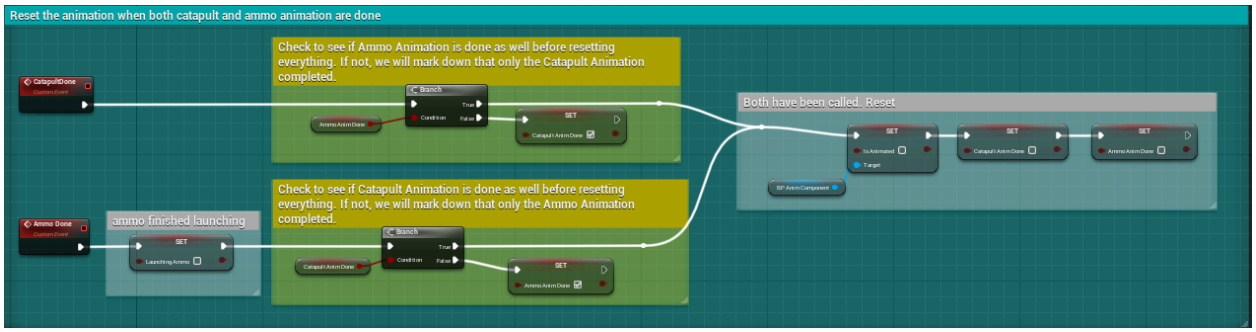


b) If either the thing being launched didn't have the ammo tag (its possible for some objects to be launchable but not do damage to the wall (have the "Launchable" tag but not the "Ammo" tag) such as random physics objects or the original design for the apples in the apple task) or the wall is already broken (likely impossible since there should only be enough ammo around the map to break the wall exactly and not have any more), then instead turn the physics of the launched object back on.

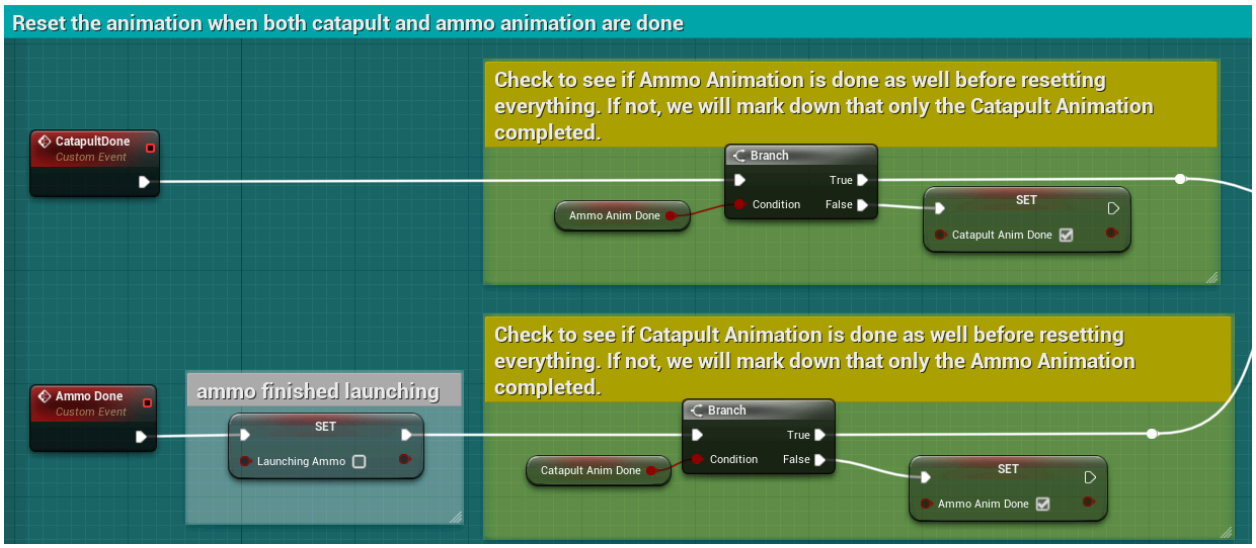


c) Finally, clear the launched variable for both options

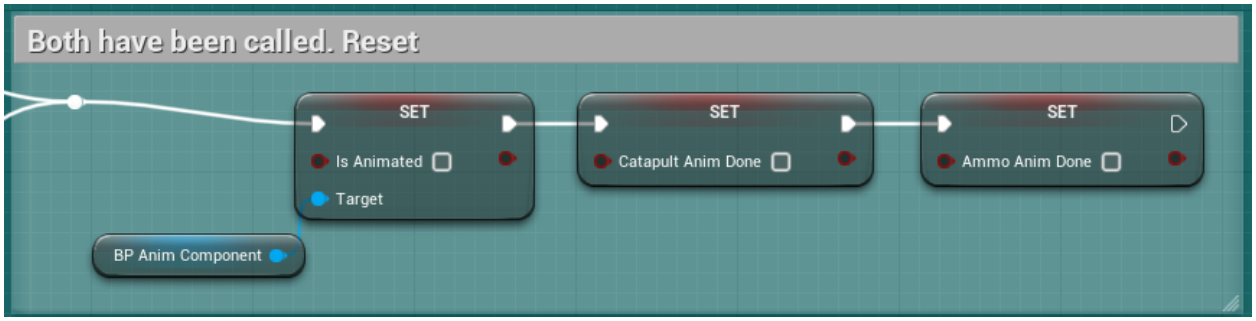
Event CatapultDone & Event AmmoDone



This Event handles resetting the Spoonapult after all animations have completed and another actor is ready to launch. This is so that the catapult doesn't try to launch a new piece of ammo when one is already launching or try to animate while the catapult is still doing its rotation animation.



1) When either event is called, check if the other one has been called yet
    a) If no, record that this event has been called so when the next is called, it remembers
    b) If yes, reset the anim component and both of these variables so that the catapult can be animated again



Changelog:

- ***August 2, 2023:*** *Deleting deprecated code. All code that is only used for physics based catapult and trial code are deleted. This is because the catapult now uses splines and animations to launch objects. (Justin Cheng)*
- ***August 7, 2023:*** *Deleted K2Node "Call on Ammo Hit Wall" In the Ammo Hit Wall Event Definition. I believe this wasn't doing anything in that code. (Justin Cheng)*
- ***August 9, 2023:*** *ReColor-Coded the events for more contrast and clarity (Justin Cheng)*