001

Recreating Destiny's Weapon and Perk System in Unreal Engine 4/5
Project Support Document

Daniel George Mark Dore

December 2022 - January 2023

Abstract

Project still in progress

Contents

List of Figures

1 Introduction

Instead of designer's/games just giving a player a weapon and saying 'go and play', many new games are developing weapon customization systems that allow players a better decision on how they want to play. Some systems allow the player to choose different perks or buffs and some allow changes to their abilities entirely.

One of my favorite weapon systems is Destiny 2's (). In the game, a single weapon is defined through it's stats and visual appearance, but each weapon shares a similar weapon frame - who share the same damage and rate-of-fire stats. This allows the game to remain in a good state of balance, while also introducing a wide range of different weapons to the player.

The aim of this project is to re-create the weapon systems from Destiny 2 into Unreal Engine 4/5, including the stat and frame system, perk/plug system, three different weapon buckets, damage multipliers and interaction between weapons and the player.

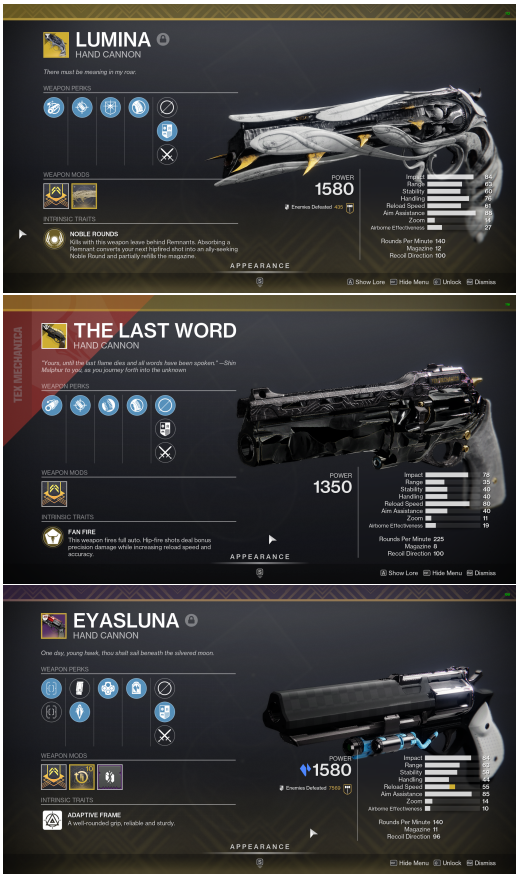

Figure 1. Destiny 2's Lumina, The Last Word and Eyasluna Hand Cannons

2 Plugs and Sockets

This section will go over how the plug and socket system works, the differences between plugs and how they control a weapons feel.

2.1 Introduction

If an item in Destiny 2 uses perks, selectables, unlockables or any other means of interaction, it uses a plug and socket system. Each perk is a different type of plug and - like plugs around the world - only fit into one type of socket.

2.2 Intrinsics

Intrinsic plugs are tied to the type of weapon and are used to define a weapon archetype / frame (see 2.2.1) and are *The base qualities of a weapon* (in-game). They cannot be changed or removed from a weapon and define how fast a weapon will fire, how much damage it will deal and how high its critical multiplier will be.

Exotic weapon traits are also classified as intrinsic perks and are created from other intrinsic perks themselves, unless that weapon has its own unique archetype. For example, the intrinsic *Noble Rounds* on *Lumina* will also feature the *Adaptive Frame* intrinsic, while the *Fan-Fire* on *The Last Word* wont feature any intrinsic perk as it has a unique fire rate profile.

2.2.1 What are Frames?

A weapon frame is the basic definition on how a weapon will perform. Players use this to know how a weapon will perform on a basic level without using the weapon themselves, as each weapon in a frame will perform similarly to each other. Hovering over a frame

will give a short description on how the weapon acts.

Figure 2. The Rapid-Fire Frame Intrinsic which states 'Deeper ammo reserves. Slightly faster reload when magazine is empty.'

This system also allows for weapons to be kept in line with one another reducing power creep and overall an easier time balancing a large pool of weapons.

2.2.2 Stat Ranges

Each frame also states the min and max bounds of each stat (see 3) which are linear interpolated between on the actual weapon. For example, when the range stat is at 38 on a Adaptive Hand Cannon it reaches 20.24m, while one at 78 reaches 23.7m.

Stat ranges can be changed through other means, such as traits, but frames control what can be achieved through stats.

2.2.3 Built-In Perks

Some frames also feature some traits that can be found on weapons built-in. These perks can also overlap and work together, amping the effect. Alternatively, they have stat changes to improve how the weapon fells to use.

Figure 3. Both Alloy Magazine (right) and Rapid-Fire Frame (left) share the same perk - 'Faster reloads when the magazine is empty'

2.3 Barrels, Magazines, Batteries and Sights

Barrels, Magazines, Batteries and Sights are all stat modifying perks. Most barrels and magazines are universal, being used by multiple weapon types (all in the case of barrels). Additionally, some magazine options give special bonuses - such as higher flinch or ricocheting rounds - instead of stat boosts. Batteries are only used by fusion-based weapons.

Figure 4. Arrowhead Break (left) increases handling and recoil direction, while Full Bore (right) increases range while decreasing handling and stability

Sights are unique (with one exception) where they change the model and zoom value of the weapon. They are rarely found on weapons anymore due to development costs - requiring a developer to make sure each sight works for a weapon.

2.4 Traits

Traits are what make different instances of a weapon stand-out from one another. Usually, a weapon can roll with two trait columns with a single choice in each column - but there are exceptions. Some weapons come with different selections in columns.

Figure 5. Some weapons - such as Cataclysmic (Adept) can spawn with multiple traits in each column

Not all traits are available on a single weapon and some traits are limited to a set of weapons. Some traits are only seen on a single weapon or frame.

Figure 6. The trait Headstone is only seen on weapons with the Stasis element, such as the Eyasluna seen in figure 1

3 Weapon Stats

This section will go over what each stat of a weapon controls and how it interacts with other stats where applicable.

3.1 Impact and Rate of Fire

A weapon's damage, critical multiplier and rate of fire are tied to its intrinsic frame, where all weapons in that frame share the same values. For example, a High-Impact Scout rifle will always have an impact of 67 and a rate of fire of 150 RPM. Rarely, traits can improve rate of fire without changing the damage profiles.

Figure 7. The trait Desperado increases weapon rate of fire after a precision kill for a short time

3.2 Element and Restrictions

Elements are restricted to certain weapon slots. Currently, only Kinetic and Stasis non-heavy weapons can be equipped in slot one, with any other element non-heavy weapon in slot two. All heavy weapons can only be equipped in slot three, however heavy weapons cannot be Kinetic.

3.3 Range and Zoom

Range is the most complex stat of the weapon. It defines not only the weapon range and when it starts experiencing damage dropoff, but how far its accuracy cone extends too and how tight that cone is.

Explained by Weisnewski, range 'can be explained as a cone that radiates out of the barrel of the gun... the better the range, the further that cone goes out and the narrower it is.' He also states the terms initial accuracy and final accuracy, or how accurate the weapon is on the first bullet and the last bullet before letting the weapon settle.

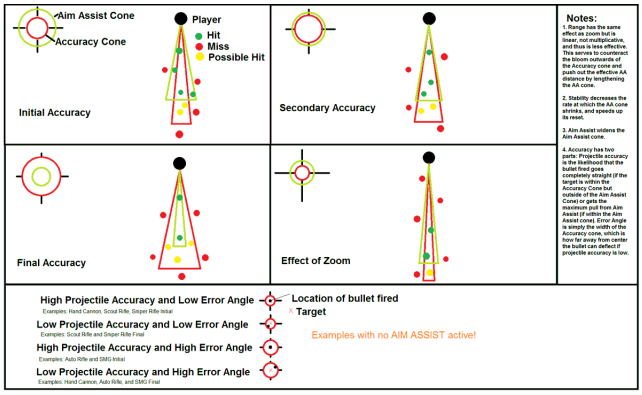

Figure 8. Mercules info-graphic on how accuracy cones work

Zoom is how far the player's camera zooms in when aiming, by changing the camera's field of view. It also acts as a multiplier for these range effects.

Figure 9. An example in changes of zoom on the Vacuna SR4 scout rifle

3.4 Stability and Recoil Direction

Both Stability and Recoil Direction are used to determine a weapon's recoil profile.

Weapon Recoil is split into two parts - actual and visual recoil. Actual recoil is how much the camera is affected when firing the weapon, while visual recoil is an animation with different properties used when firing.

Recoil Direction is also split into two parts - Bounce Intensity and Bounce Direction - and both are calculated from a graph. The higher a weapon's Recoil Direction the less intense its recoil bounce will be, while the second digit of the number states which direction the recoil will bounce towards.

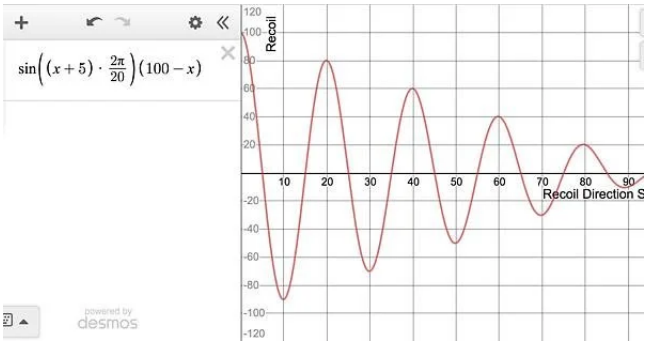

Figure 10. The Recoil Direction Graph, of formula

$$sin((x+5)\frac{2\pi}{20})(100-x) \quad (1)$$

. The closer a weapon is to 5, the more vertical its recoil will be.

3.5 Handling

Handling controls how fast the weapon manoeuvres. This includes stow, draw and aim-down-sight speeds but also drop recovery speed after falling. Some traits can bypass the need for a high handling stat.

3.6 Reload

Reload speed is simply how fast the weapon can be reloaded. Traits can control the reload duration - increasing the speed of the animation further.

3.7 Magazine Size and Ammo Types

Magazine Size is simply how much ammo can be held in a single magazine of a weapon.

Each weapon feeds off of a single ammo type based on its core weapon (Auto Rifle, Shotgun, etc), with some exceptions. All primary weapons have infinite ammo, while special and heavy weapons require ammo picked up from the corresponding brick to refill reserves.

Primary and Special weapons are only seen in the Kinetic and Energy slots, while Heavy weapons can only be equipped in the Heavy slot.

3.8 Aim Assist

Accuracy Cones

4 Project Goals

For this project, I plan to recreate this weapon system in detail utilizing Unreal Engine 4 and a mix of C++ and Blueprint. The main topics I plan to integrate are:

- Weapon Firing and basic actual recoil
- Accuracy Cones and some level of aim assist
- Stat controls, with max and min values
- More simplistic Plug and Socket system
- Working Traits
- Menu system

5 Methodology

5.1 Classes

We began project development by adding empty classes for the objects and characters that the project will use.

```
class WeaponParent
class TraitParent
class WeaponStatLibrary
class DamageNumber
class PlayerCharacter
class EnemyParent
class DamagableInterface
class WeaponInterface
class WeaponPickup
```

The WeaponParent class will have multiple children down the line - one for each fire type (full automatic, burst, etc), but most of the core functionality will be inherited from this class. We use two interfaces to streamline interaction between classes, as well as some objects could be damageable later on in development.

5.2 WeaponParent Setup

We then add two functions to the ParentWeapon class - SetupWeapon and FireBullet. Currently SetupWeapon does nothing, as this requires some data to be set up and present, while FireBullet simply completes a line trace from the player's first person camera.

```
bool AWeaponParent::FireBullet()
{
// Trace Properties
FHitResult hitResult(ForceInit);
FVector start =
↪  Player->FirstPersonCamera->GetComponentLocation();
FVector end = start +
↪  (Player->FirstPersonCamera->GetForwardVector() * 20000);

FCollisionQueryParams traceParams =
↪  FCollisionQueryParams(FName(TEXT("RV_Trace")), true, this);
traceParams.bTraceComplex = true;
traceParams.bReturnPhysicalMaterial = false;

// Complete Trace
GetWorld()->LineTraceSingleByChannel(
        hitResult,
        start,
        end,
        ECC_EngineTraceChannel2,
        traceParams
);
        return false;
}
```

5.3 Weapon Data

Next, we add a large amount of data structs and enumerators to the *WeaponStatLibrary.h*. Multiple structs are used to keep data as easy to read as possible.

```
enum EAmmoType
enum EAspectType
enum EStatsType
enum ETraitType
enum ETriggerType
enum EWeaponType


struct FTriggers
struct FImpact
struct FWeaponFrame
struct FWeaponStats
struct FTraitStats
struct FTraitColumn
struct FWeaponVisual
struct FWeaponInfo
```

We also create the data curves for some tests stats, including the Recoil Direction graph.

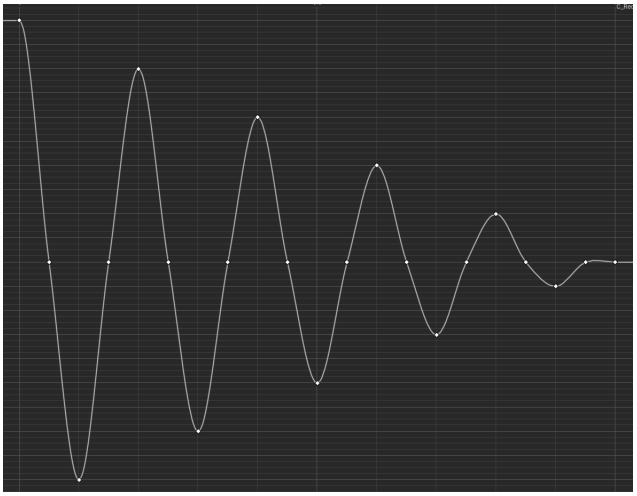

Figure 10. The Recoil Direction Graph in Unreal Engine

Currently, CurveFloats limit to only one data curve per UObject, so the data sturcts may need to be modified later during development if a single stat contains multiple data curves (Handling, for example, holds Stow, Draw and ADS)

5.4 Setting Weapons

We then added the functionality to set up a weapon from an ID supplied. When the player character object wants to set up a new weapon, it clears the class currently occupying the corresponding weapon slot, sets the class to the new weapon's class and then calls SetupWeapon inherited from the class parent.

```
bool AWeaponParent::SetupWeapon()
{
// Find Row
UDataTable* dT = LoadObject<UDataTable>(this,
↪  TEXT("/Game/Weapons/Data/WeaponTable.WeaponTable"));
FWeaponInfo* row = dT->FindRow<FWeaponInfo>(ID, "", false);

if (row == nullptr)
{
        return false;
}

// Set the stats and magazine
Stats = row->BaseStats;
CurrentMagazine = Stats->Magazine;

// Set the visuals of the weapon
Body->SetSkeletalMesh(row->BaseVisual.BodyMesh);
Magazine->SetSkeletalMesh(row->BaseVisual.MagazineMesh);

return true;
}
```

We then removed the need to find the data table on SetupWeapon call and instead have a pointer stored in the player instead and pass through the FWeaponInfo struct - as the player will be the only class that can create a new weapon.

```
bool APlayerCharacter::SetNeweapon(FName ID)
{
// Find the weapons info
FWeaponInfo* foundWeapon =
↪  WeaponDataTable->FindRow<FWeaponInfo>(ID, "", false);
if (foundWeapon == nullptr) { return false; }

// Set the child actor class and cast to the new class
KineticWeaponActor->SetChildActorClass
    (foundWeapon->BaseStats.Frame.Class);
AWeaponParent* nw =
↪  Cast<AWeaponParent>(KineticWeaponActor->GetChildActor());
nw->Player = this;

// Store the weapon at the correct index in the weapon array
CurrentWeapons.Insert(nw, 0);

// Finally, setup the weapon
nw->SetupWeapon(ID, foundWeapon->BaseStats,
↪  foundWeapon->BaseVisual);

return false;
}
```

5.5 Player Movement

We implemented player movement next, which allows the player to move around with WASD and aim with the mouse.

5.6 Swapping Weapons

We implemented the ability to swap between any of the three weapons through different means. The player can directly choose which slot to swap too, swap to the next one in the inventory or swap to the previous weapon. All of the swap functions lead to the same function, allowing outside stimuli to swap weapons where needed.

```
// Swapping Functions
void APlayerCharacter::SwapNext()
```

```
{
if ((CurrentWeaponIndex + 1) >= 3) {
        NewWeaponIndex = 0;
        StowCurrentWeapon();
}
else {
        NewWeaponIndex = CurrentWeaponIndex + 1;
        StowCurrentWeapon();
}
}

void APlayerCharacter::SwapPrev()
{
if ((CurrentWeaponIndex - 1) <= -1) {
        NewWeaponIndex = 2;
        StowCurrentWeapon();
}
else {
        NewWeaponIndex = CurrentWeaponIndex - 1;
        StowCurrentWeapon();
}
}

void APlayerCharacter::SwapTo(int Index)
{
if (Index != CurrentWeaponIndex) { NewWeaponIndex = Index;
↪  };
}
```

5.7 Reloading and Reserves

We add the ability to reload the weapon. First, we check if we have enough ammo to reload before starting the reload animation. If successful, we then play the animation and call another function to refill the magazine based on the reserves.

```
bool AWeaponParent::GetCanReload()
{
if (ReserveAmmo == -1 || ReserveAmmo >= 1) { return true; };

return false;
}

void AWeaponParent::RefilMagazine()
{
if (ReserveAmmo == -1) {
        CurrentMagazine = Stats.Magazine;
}
else if (Stats.Magazine < CurrentMagazine + ReserveAmmo) {
        ReserveAmmo = (ReserveAmmo + CurrentMagazine) +
↪  Stats.Magazine;
        CurrentMagazine = CurrentMagazine + ReserveAmmo;
}
else if (Stats.Magazine > CurrentMagazine + ReserveAmmo) {
        CurrentMagazine = CurrentMagazine + ReserveAmmo;
        ReserveAmmo = 0;
}
}
```

We also add Reserves to FWeaponFrame. The function to refill the reserves is also added. To simplify this function, only two inputs are used - 0 for a small ammo brick and 1 for a large one. Any other number goes through *default*.

```
void AWeaponParent::AddToReserves(int Size)
{
        switch (Size) {
        case 0: // A finder brick
                break;

        case 1: // A normal brick
                break;

        default:
                break;
        }
}
```

5.8 Damaging and Interface

We implement a function to determin how much damage a bullet should do over it's trace range. We also calculate the damage it should deal if the trace hits a target's head.

```
float AWeaponParent::GetDamageFromRange(float TraceDistance)
{
float dist =
↪  Stats.Frame.Range.Dropoff->GetFloatValue(Stats.Range);
if (TraceDistance <= dist) {
        return Stats.Frame.Imapct.Damage;
}
else if (TraceDistance <= dist + Stats.Frame.Range.EndDist) {
        return (FMath::Lerp(Stats.Frame.Imapct.Damage,
↪  Stats.Frame.Imapct.DropoffDamage, TraceDistance - dist /
↪  Stats.Frame.Range.EndDist));
}
else {
        return Stats.Frame.Imapct.DropoffDamage;
}
}
```

Following, we implement an interface function *TakeDamage*, which will only damage targets with the interface. C++ interfaces are somewhat complex compared to Blueprint interfaces, utilizing some specialized macros and identifiers to work properly. We also implement the ability to deal damage to enemies with a function in the parent enemy.

5.9 Full Auto

We implement our first trigger class - Full Auto. When the Fire button is held down in this mode, the weapon fires a bullet after a set delay until the button is released or the magazine is emptied. We extend this class from WeaponParent and override the functions *StartFire* and *ContinueFire*, removing the need for another interface.

```
void AFullAutoTrigger::StartFire(bool bCanFire)
{
if (bCanFire) {
        if (CurrentMagazine > 0) {
        FireBullet();
        // VisRecoil
        // ActualRecoil
        GetWorldTimerManager().SetTimer(AutoTimerHandle, this,
↪  &AFullAutoTrigger::ContinueFire,
↪  Stats.Frame.Imapct.RateOfFire / 60, true);
        }
    else {
        GetCanReload();
        }
}
else {
        GetWorldTimerManager().ClearTimer(AutoTimerHandle);
}
}

void AFullAutoTrigger::ContinueFire()
{
if (CurrentMagazine > 0) {
        FireBullet();
        // VisRecoil
        // ActualRecoil
}
else {
        GetCanReload();
}
}
```

We also add two more child actors to the player character - one for the heavy weapon and one for the secondary weapon. As a test to make sure the weapons equip properly, we create a data row in the Data Table and add it on the player's BeginPlay.

```
// Sets default values
APlayerCharacter::APlayerCharacter()...
KineticWeaponActor =
↪  CreateDefaultSubobject<UChildActorComponent>(TEXT("Kinetic
↪  Weapon Component"));
EnergyWeaponActor =
↪  CreateDefaultSubobject<UChildActorComponent>(TEXT("Energy
↪  Weapon Component"));
HeavyWeaponActor =
↪  CreateDefaultSubobject<UChildActorComponent>(TEXT("Heavy
↪  Weapon Component"));
```

```
void APlayerCharacter::BeginPlay()...
SetNeweapon(FName("AR_AD_001"));
```

5.10 Tracing and Firing

We then implement our functions to fire, fire in full auto and return if we have hit something. We will need to change from returning a bool to a specified struct later on in development. Additionally, we add a *DrawDebugLine* to test that our traces are being complete.

5.11 Attaching Swapped Weapons

We then finish swapping weapons by attaching any non-equipped weapons to slots on the players back.

```
SwapToWeapon...
// Attach the old weapon to a slot on the back
FAttachmentTransformRules
↪  attachRules(EAttachmentRule::SnapToTarget, false);
CurrentWeapons[CurrentWeaponIndex]->AttachToComponent(ActiveWeaponLoc,
↪  attachRules, "");

// Attach the new weapon to the ActiveWeapon component
bool bfirstslot = false;
for (int i = 0; i >= 2; i++) {
        if (i != Index) {
                if (bfirstslot == false) {
                        CurrentWeapons[i]->AttachToComponent(WeaponBackL,
↪  attachRules, "");
                }
                else {
                        CurrentWeapons[i]->AttachToComponent(WeaponBackR,
↪  attachRules, "");
                }
        }
}
CurrentWeapons[Index]->AttachToComponent(ActiveWeaponLoc,
↪  attachRules, "");
CurrentWeaponIndex = Index;
```

5.12 Semi Auto and Burst Fire

We implement semi automatic and burst fire triggers next. Semi auto doesn't use any timers and simply fires a bullet each time the fire button is pressed, but not when the button is released. Burst Fire uses a timer to fire *n-1* rounds in the weapons burst. The amount of bullets in a burst and the delay between each bullet can be set in the class.

```
//ABurstTrigger::StartFiring...
if (bCantFire == true && bInBurst == false) {
        if (CurrentMagazine > 0) {
                BurstShotsRemain = BurstShots - 1;
                ...
                GetWorldTimerManager().SetTimer(BurstTimerHandle,
↪  this, &ABurstTrigger::ContinueFire, InBurstDelay, true);
        }

//ABurstTrigger::ContinueFire...
UE_LOG(LogTemp, Warning, TEXT("Continue Fire"));
if (--BurstShotsRemain <= 0) {
        GetWorldTimerManager().ClearTimer(BurstTimerHandle);
}
```

5.13 Fixing attaching weapons

We implement a change that fixed attaching weapons to the back of the player - we no longer attach the classes and instead attach the child actors.

```
case 0: // Kinetic to set held
                KineticWeaponActor->AttachToComponent(ActiveWeaponLoc,
↪  attachRules, "");
                EnergyWeaponActor->AttachToComponent(WeaponBackL,
↪  attachRules, "");
                HeavyWeaponActor->AttachToComponent(WeaponBackR,
↪  attachRules, "");
                break;

        case 1: // Energy to set held
                EnergyWeaponActor->AttachToComponent(ActiveWeaponLoc,
↪  attachRules, "");
                KineticWeaponActor->AttachToComponent(WeaponBackL,
↪  attachRules, "");
                HeavyWeaponActor->AttachToComponent(WeaponBackR,
↪  attachRules, "");
                break;

        case 2: // Heavy to set held
                HeavyWeaponActor->AttachToComponent(ActiveWeaponLoc,
↪  attachRules, "");
                KineticWeaponActor->AttachToComponent(WeaponBackL,
↪  attachRules, "");
                EnergyWeaponActor->AttachToComponent(WeaponBackR,
↪  attachRules, "");
                break;
```

5.14 Damage Number and Controller

We implemented the damage number's functionality, where it starts not in use under the map in an object pool. Once needed, the controller finds the first available damage number in the pool, moves it to the needed location (where a damageable was hit) and sets it active.

While it is active, the widget portion of the class will face the player, a random direction to move in will be chosen and a timeline will begin. This timeline follows a supplied FCurveFloat object, outputting a float that will be used in a lerp to move the object in the given direction. Once finished, it will be moved back into the pool of other DamageNumbers under the map.

```
bool ADamageNumber::NewDamageNumber(FVector Loc, bool bNewCrit,
↪  int NewDamage)
{
// Set this damage numbers location
SetActorLocation(Loc);
StartLoc = Loc;

bInUse = true;

// Update the info for the widget
Damage = NewDamage;
bCrit = bNewCrit;

// Choose a random diretction to move in
float randomDir = FMath::FRandRange(-180, 179);
Core->SetWorldRotation(FRotator(0.0f, randomDir, 0.0f));
TravelLine->PlayFromStart();

return true;
}

void ADamageNumber::TravelLineCallback(float val)
{
FMath::Lerp(StartLoc, FVector(StartLoc.X + Distance,
↪  StartLoc.Y, StartLoc.Z), val);
}

void ADamageNumber::TravelLineFinishedCallback()
{
bInUse = false;
SetActorLocation(FVector(0.0F, 0.0F, 0.0F));
}
```

5.15 Damage Number Widget

We implement the C++ and Blueprint classes for the DamageNumberWidget. The C++ class only features two functions - one to set the properties to be shown in the widget and the *SynchronizeProperties* function that is required for UserWidget classes.

Figure 11. Example of damage number in action

5.16 Aim Down Sight

We implement the ability to aim down sight. Simply, it moves the *ActiveWeaponLoc* component between the resting hip-fire location to in-front of the player. We implemented this in Blueprint, as this timeline is more complex than the *DamageNumber* timeline. We also take the location of the weapon's sight in effect, moving the final ADS location based on it's location.

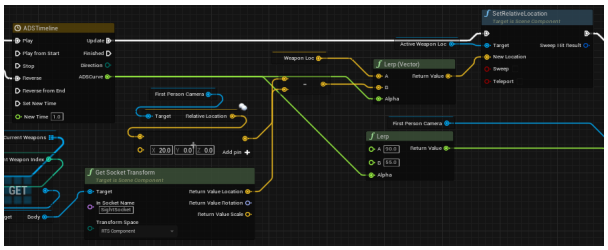

Figure 12. The ADS timeline in Blueprint

We also add the ability to change the aim down sight speed. As the timeline to aim down sight is on the player, we pass through the new speed to the player through function *SetADSSpeed*. As the timeline is a second total, we divide the new value by 6, as Set Playback Speed acts as a multiplier.

5.17 Reload

We implemented the ability to reload weapons next. Similarly to aiming down sights, we use a timeline to move the magazine mesh between its position on the weapon to out of view, then back again. We also stop the player from firing their weapon while the animation is playing, while also allowing the player to cancel the animation.

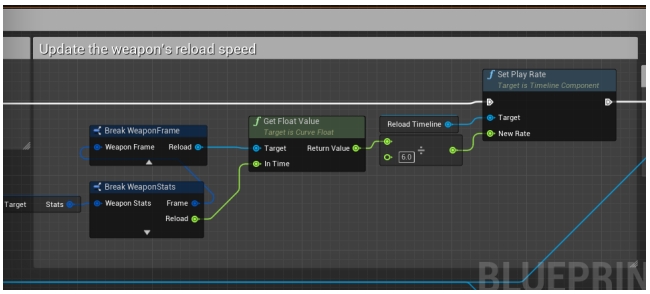

Figure 14. The reload speed is updated when the reload starts, allowing for traits to modify the speed later in development

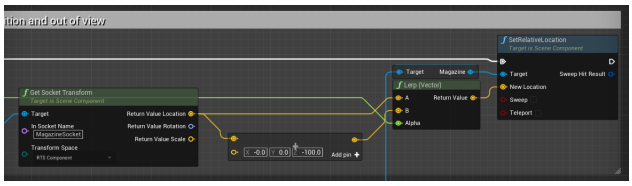

Figure 15. The magazine lerps between two different locations

5.18 Zoom

We finished aiming down sight by implementing a modifiable zoom. When zooming, we divide whatever the base player's field of view is by the zoom value and lerp between the two during the timeline.

5.19 Recoil

We implemented weapon recoil next. Our weapon recoil system is made of three parts - weapon kickback, visual recoil and actual recoil. Weapon kickback moves the weapon back in the player's hand, visual recoil moves the barrel of the weapon upwards and actual recoil moves the player's camera.

However, to implement it successfully, we needed to backtrack slightly and change how our weapon is animated. We removed the timeline for reloading, removed the magazine mesh and in their place implemented a single skeletal mesh with animations. We also changed the WeaponVisual struct to reflect these changes.

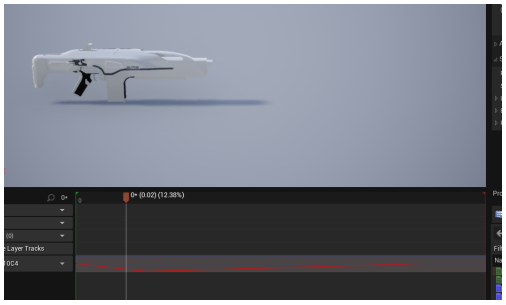

Figure 16. The kickback animation is only a few frames long and moves the weapon back and forward a short distance

Following this, we created a short kickback animation for the weapon that will be played when the weapon traces for a bullet.

```
SetupWeapon...
// Set the anim blueprint for the weapon mesh
Body->SetAnimInstanceClass(Visual.AnimBP->GeneratedClass);

FireBullet...
Body->GetAnimInstance()->Montage_Play(Visual.FireAnim, 1.0f);
```

Visual and Actual recoil are completed in a similar way. Both take the weapon's Recoil Direction and Stability stats into effect - where, in our case, recoil direction determines the angle and stability determines the random angle severity and horizontal severity. We use a timeline for Visual Recoil to return the weapon to the correct rotation after firing.

```
void AWeaponParent::Recoil()
{
// Get the float value from the recoil curve and calculate the
↪  angle from the curve
float recoilAngleFromCurve =  MaxRecoilAngle *
↪  (Player->RecoilCurve->GetFloatValue(Stats.Recoil));

// Calculate a random recoil angle from stability
float sHD =
↪  Stats.Frame.Stability.HorizontalKick->GetFloatValue(Stats.Stability);
float calcRecoilAngle = FMath::RandRange(recoilAngleFromCurve -
↪  sHD, recoilAngleFromCurve + sHD);

// Do Visual and Actual Recoil
Player->VisualRecoil(calcRecoilAngle,
↪  Stats.Frame.Stability.VerticalKick->GetFloatValue(Stats.Stability));
}
```

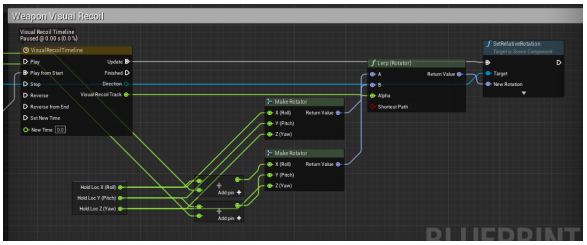

Figure 17. Blueprint of Visual Recoil

Finally, we gathered a new test model through Charm ().

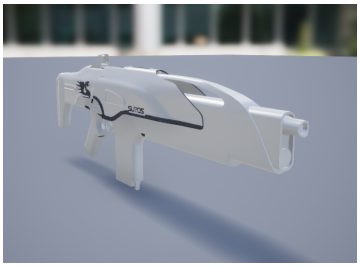

Figure 18. The Galliard-42 auto rifle, gathered from Charm

5.