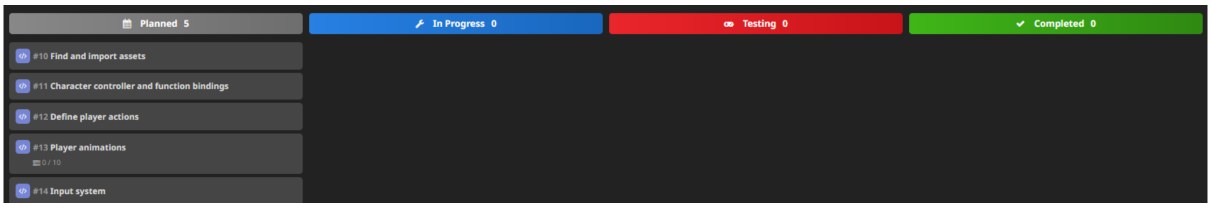
Week 1 (19/02/2025-26/02/2025)

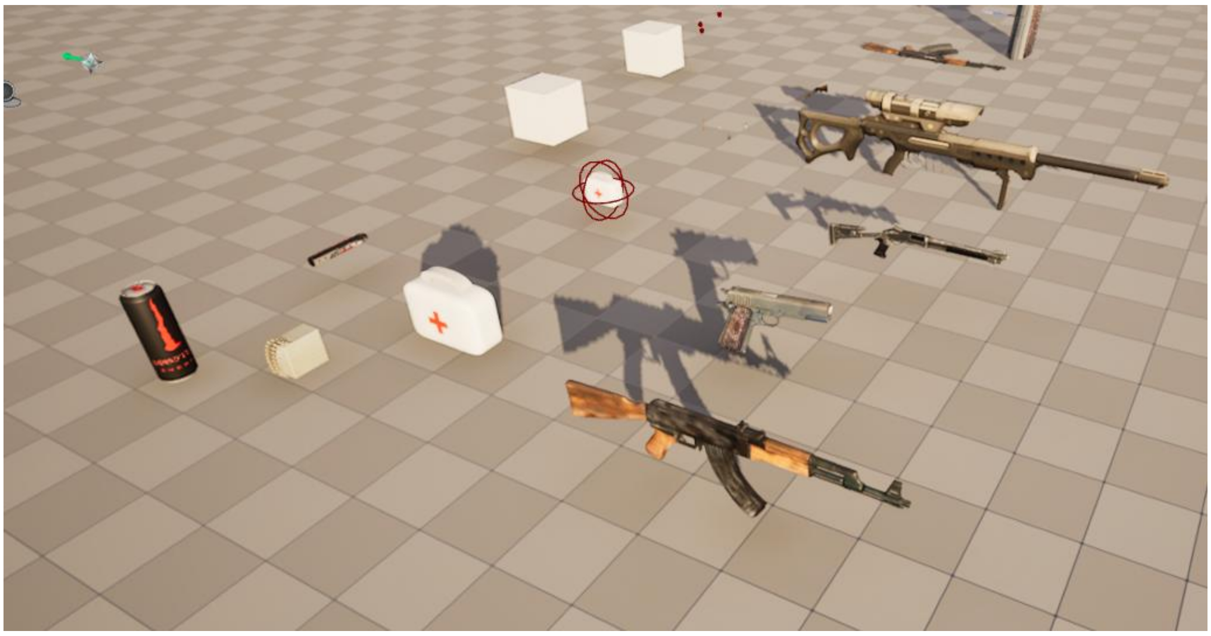
Kanban – start of the week:

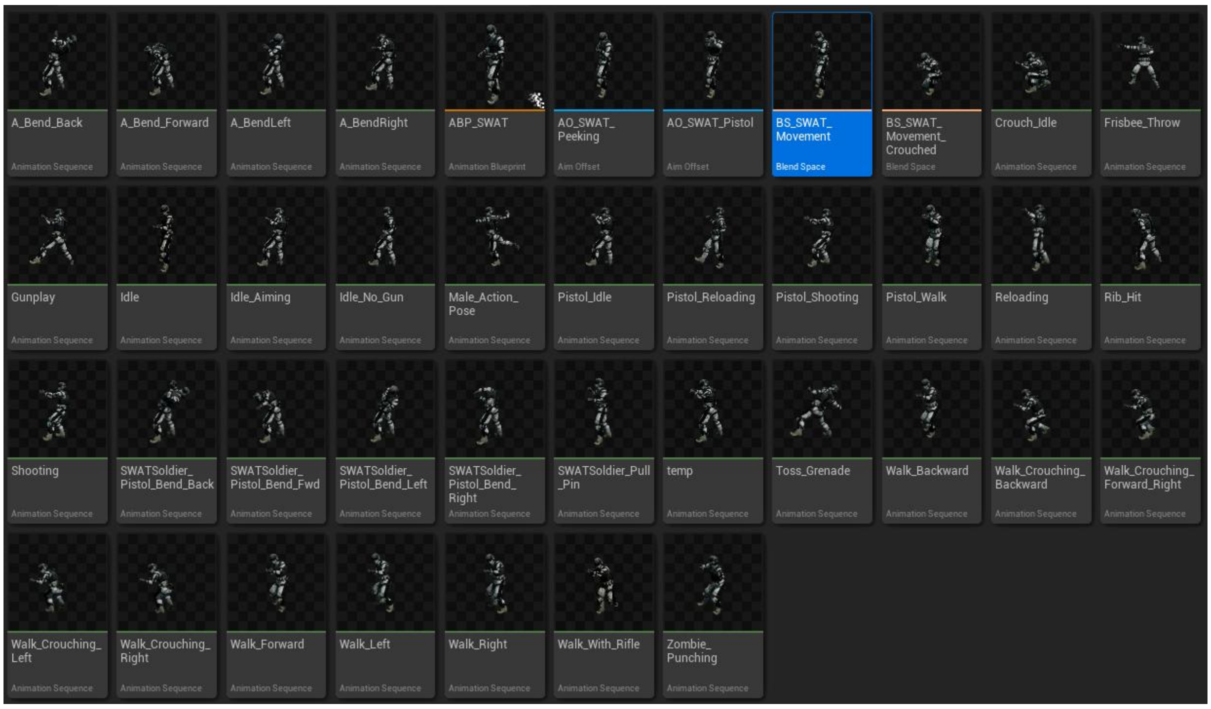


Planned 5	In Progress 0	Testing 0	Completed 0
#10 Find and import assets			
#11 Character controller and function bindings			
#12 Define player actions			
#13 Player animations 0 / 10			
#14 Input system			

Asset finding:

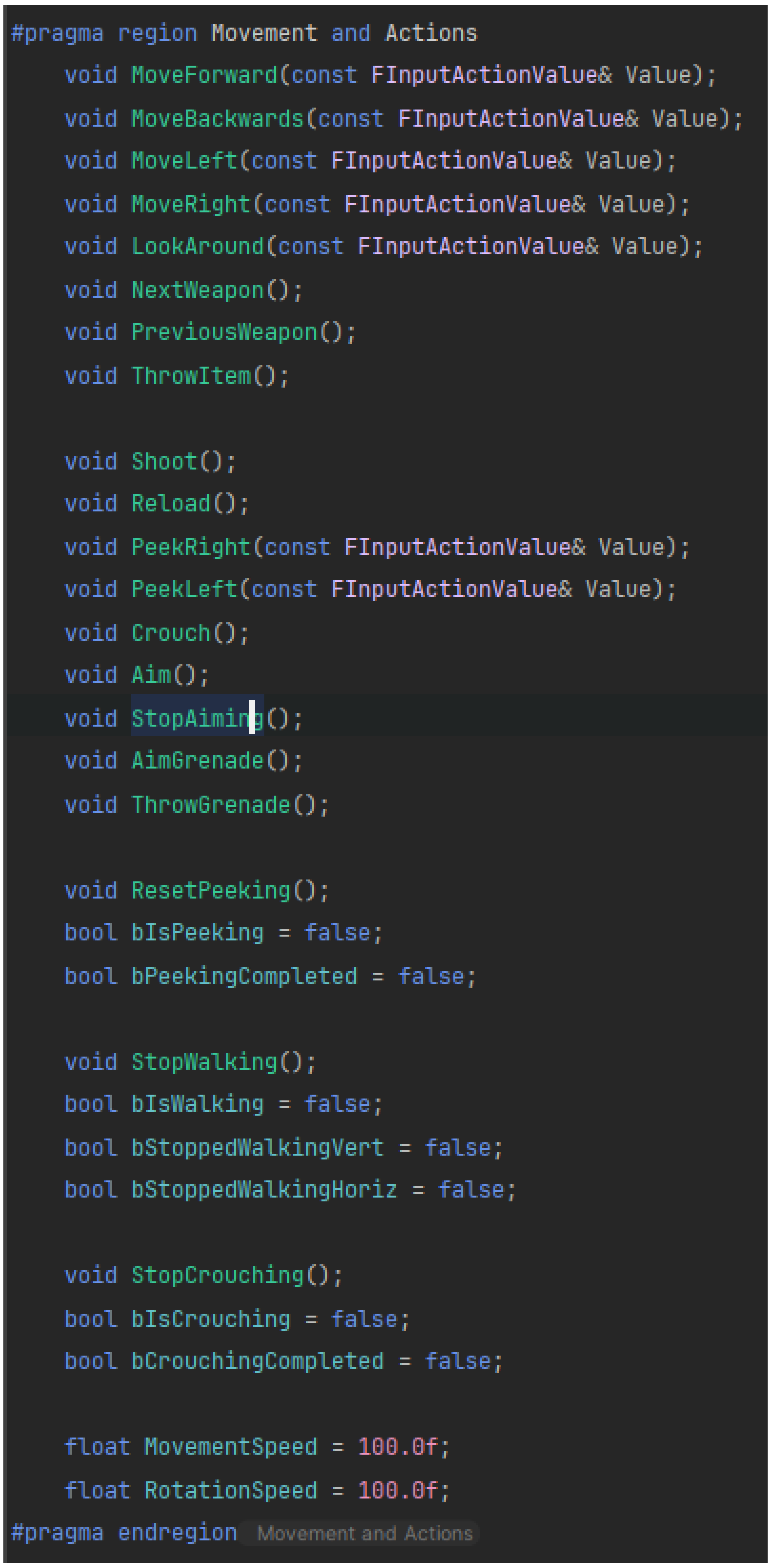
I managed to find assets for the character, weapons, pickups and animations from various sources, such as Mixamo and Turbosquid and import them into the project.





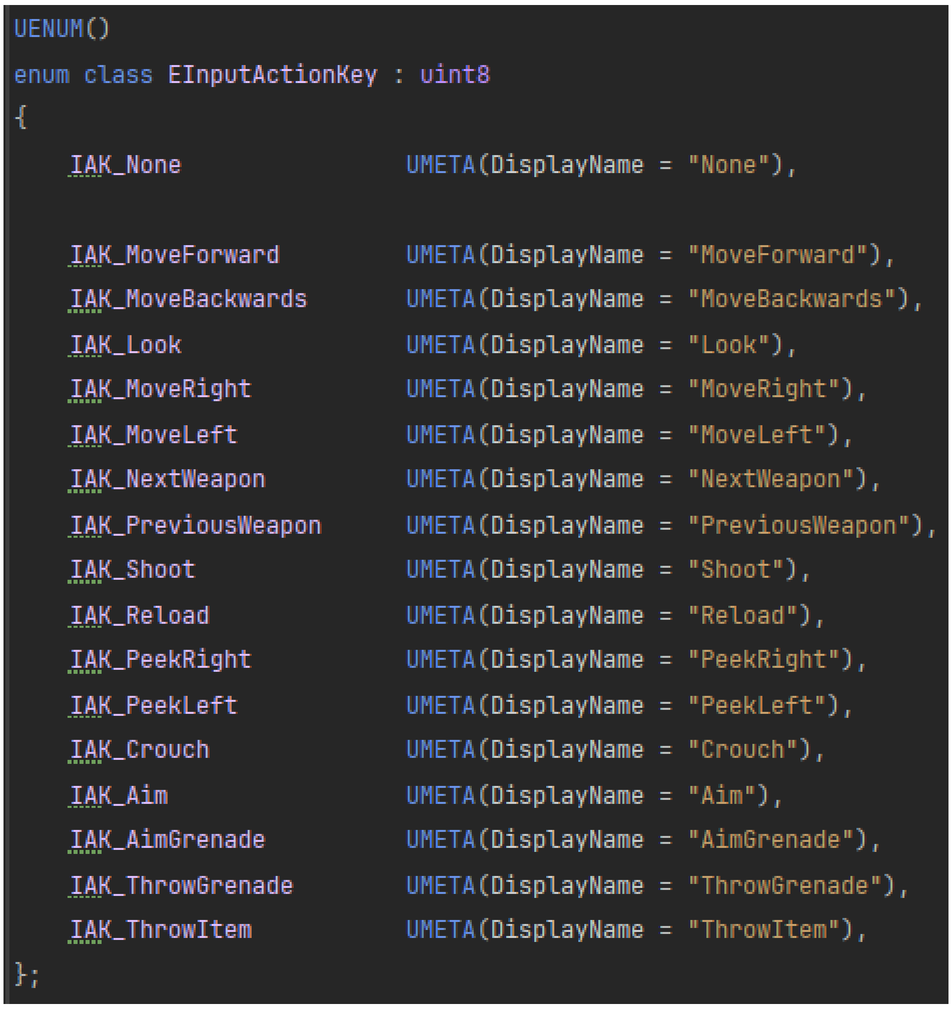
Character controller and input system:

I created a custom character controller class in C++ and defined the main functions the player would require.

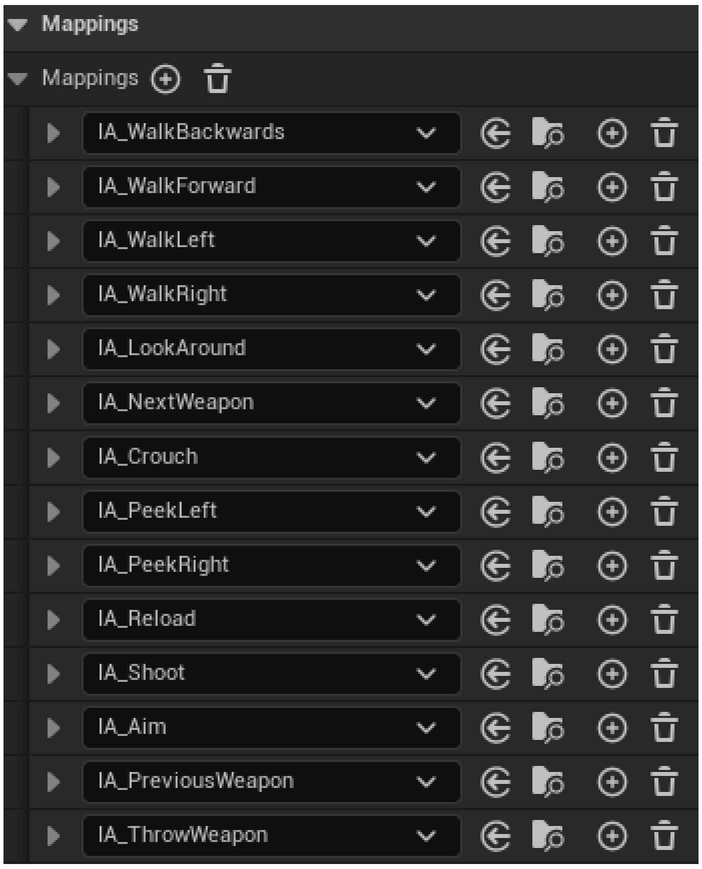


```cpp
#pragma region Movement and Actions
    void MoveForward(const FInputActionValue& Value);
    void MoveBackwards(const FInputActionValue& Value);
    void MoveLeft(const FInputActionValue& Value);
    void MoveRight(const FInputActionValue& Value);
    void LookAround(const FInputActionValue& Value);
    void NextWeapon();
    void PreviousWeapon();
    void ThrowItem();

    void Shoot();
    void Reload();
    void PeekRight(const FInputActionValue& Value);
    void PeekLeft(const FInputActionValue& Value);
    void Crouch();
    void Aim();
    void StopAiming();
    void AimGrenade();
    void ThrowGrenade();

    void ResetPeeking();
    bool bIsPeeking = false;
    bool bPeekingCompleted = false;

    void StopWalking();
    bool bIsWalking = false;
    bool bStoppedWalkingVert = false;
    bool bStoppedWalkingHoriz = false;

    void StopCrouching();
    bool bIsCrouching = false;
    bool bCrouchingCompleted = false;

    float MovementSpeed = 100.0f;
    float RotationSpeed = 100.0f;
#pragma endregion Movement and Actions
```

I also defined the input system actions inside the controller, as well as creating them in engine and added them to the input system component.



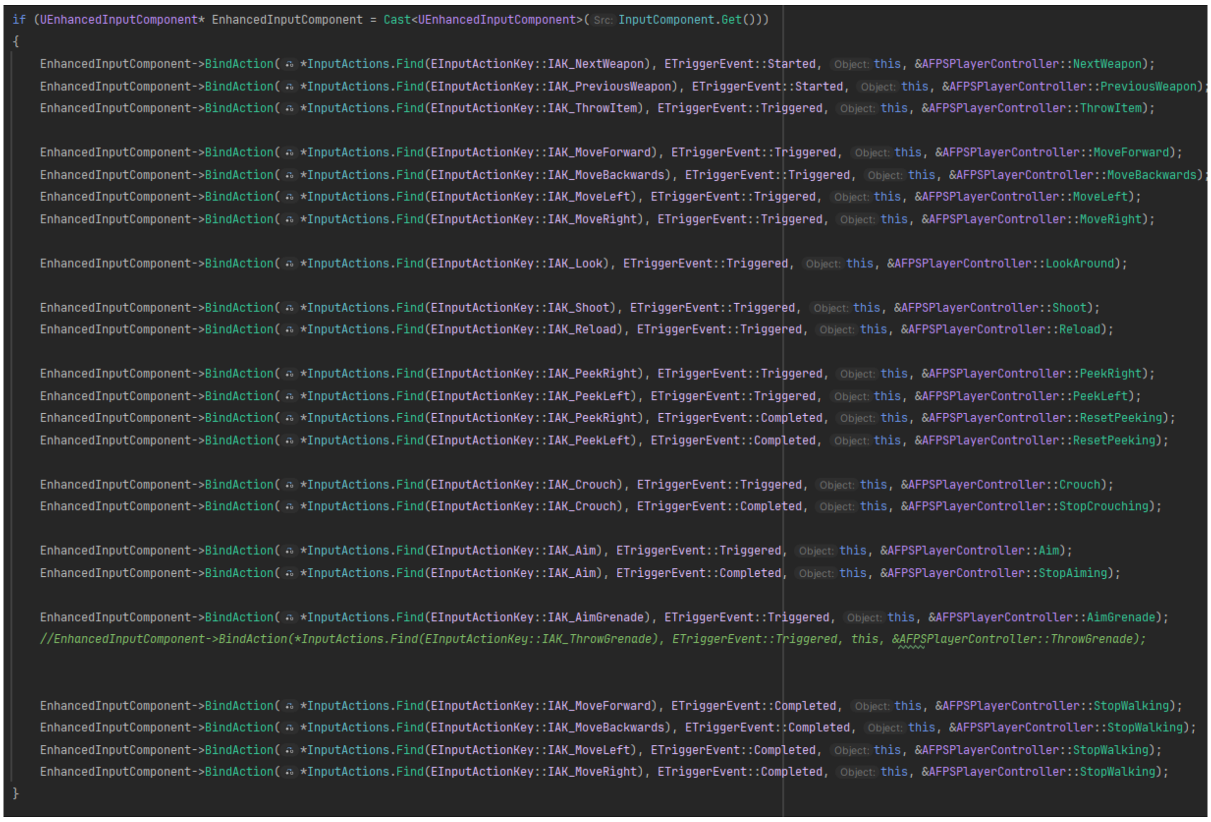
```
UENUM()
enum class EInputActionKey : uint8
{
    IAK_None                UMETA(DisplayName = "None"),

    IAK_MoveForward         UMETA(DisplayName = "MoveForward"),
    IAK_MoveBackwards       UMETA(DisplayName = "MoveBackwards"),
    IAK_Look                UMETA(DisplayName = "Look"),
    IAK_MoveRight           UMETA(DisplayName = "MoveRight"),
    IAK_MoveLeft            UMETA(DisplayName = "MoveLeft"),
    IAK_NextWeapon          UMETA(DisplayName = "NextWeapon"),
    IAK_PreviousWeapon      UMETA(DisplayName = "PreviousWeapon"),
    IAK_Shoot               UMETA(DisplayName = "Shoot"),
    IAK_Reload              UMETA(DisplayName = "Reload"),
    IAK_PeekRight           UMETA(DisplayName = "PeekRight"),
    IAK_PeekLeft            UMETA(DisplayName = "PeekLeft"),
    IAK_Crouch              UMETA(DisplayName = "Crouch"),
    IAK_Aim                 UMETA(DisplayName = "Aim"),
    IAK_AimGrenade          UMETA(DisplayName = "AimGrenade"),
    IAK_ThrowGrenade        UMETA(DisplayName = "ThrowGrenade"),
    IAK_ThrowItem           UMETA(DisplayName = "ThrowItem"),
};
```


Mappings

Mappings

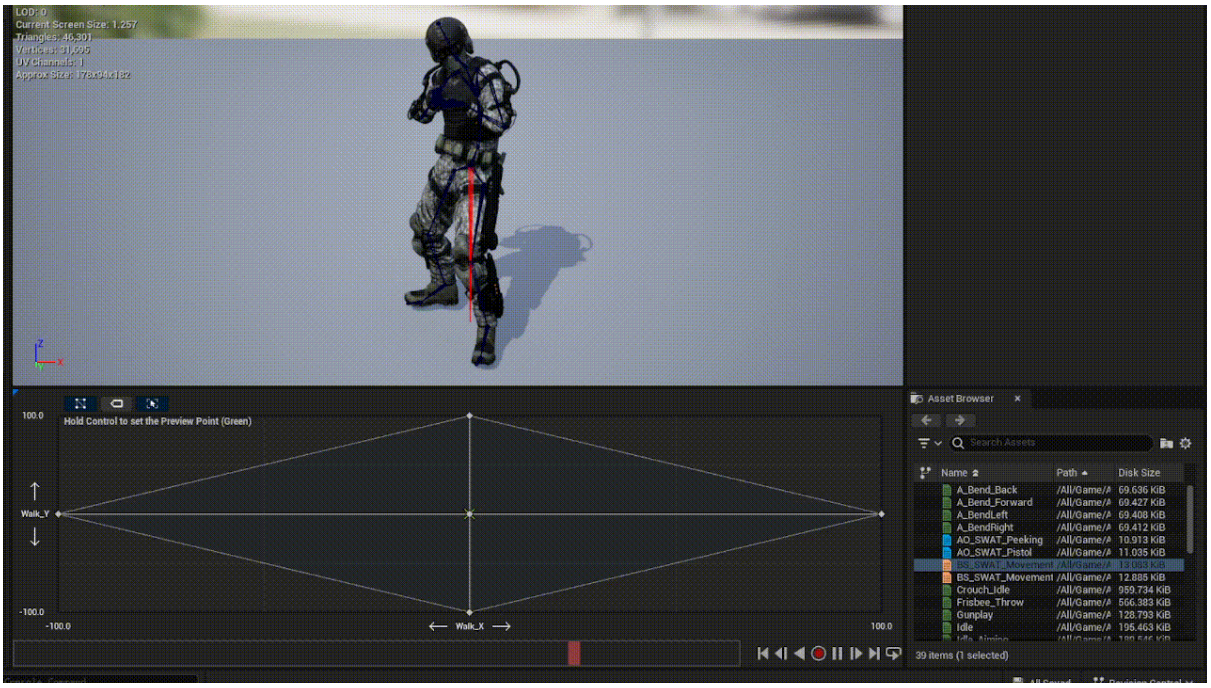
- IA_WalkBackwards
- IA_WalkForward
- IA_WalkLeft
- IA_WalkRight
- IA_LookAround
- IA_NextWeapon
- IA_Crouch
- IA_PeekLeft
- IA_PeekRight
- IA_Reload
- IA_Shoot
- IA_Aim
- IA_PreviousWeapon
- IA_ThrowWeapon

I then binded all the functions to the appropriate actions.


```cpp
if (UEnhancedInputComponent* EnhancedInputComponent = Cast<UEnhancedInputComponent>( Src: InputComponent.Get()))
{
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_NextWeapon), ETriggerEvent::Started,  Object: this, &AFPSPlayerController::NextWeapon);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_PreviousWeapon), ETriggerEvent::Started,  Object: this, &AFPSPlayerController::PreviousWeapon);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_ThrowItem), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::ThrowItem);

    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_MoveForward), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::MoveForward);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_MoveBackwards), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::MoveBackwards);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_MoveLeft), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::MoveLeft);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_MoveRight), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::MoveRight);

    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_Look), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::LookAround);

    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_Shoot), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::Shoot);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_Reload), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::Reload);

    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_PeekRight), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::PeekRight);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_PeekLeft), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::PeekLeft);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_PeekRight), ETriggerEvent::Completed,  Object: this, &AFPSPlayerController::ResetPeeking);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_PeekLeft), ETriggerEvent::Completed,  Object: this, &AFPSPlayerController::ResetPeeking);

    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_Crouch), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::Crouch);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_Crouch), ETriggerEvent::Completed,  Object: this, &AFPSPlayerController::StopCrouching);

    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_Aim), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::Aim);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_Aim), ETriggerEvent::Completed,  Object: this, &AFPSPlayerController::StopAiming);

    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_AimGrenade), ETriggerEvent::Triggered,  Object: this, &AFPSPlayerController::AimGrenade);
    //EnhancedInputComponent->BindAction(*InputActions.Find(EInputActionKey::IAK_ThrowGrenade), ETriggerEvent::Triggered, this, &AFPSPlayerController::ThrowGrenade);


    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_MoveForward), ETriggerEvent::Completed,  Object: this, &AFPSPlayerController::StopWalking);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_MoveBackwards), ETriggerEvent::Completed,  Object: this, &AFPSPlayerController::StopWalking);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_MoveLeft), ETriggerEvent::Completed,  Object: this, &AFPSPlayerController::StopWalking);
    EnhancedInputComponent->BindAction( *InputActions.Find(EInputActionKey::IAK_MoveRight), ETriggerEvent::Completed,  Object: this, &AFPSPlayerController::StopWalking);
}
```

Player animations:

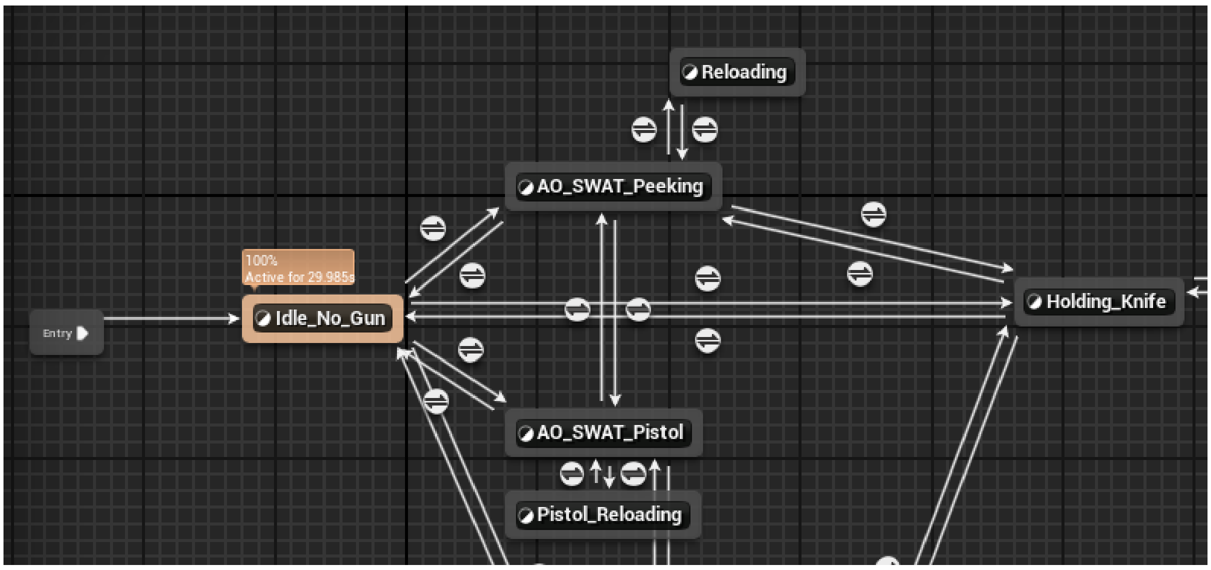
I managed to create a blend space for player locomotion that is responsible for 4 directional movement with individual animations for each direction.

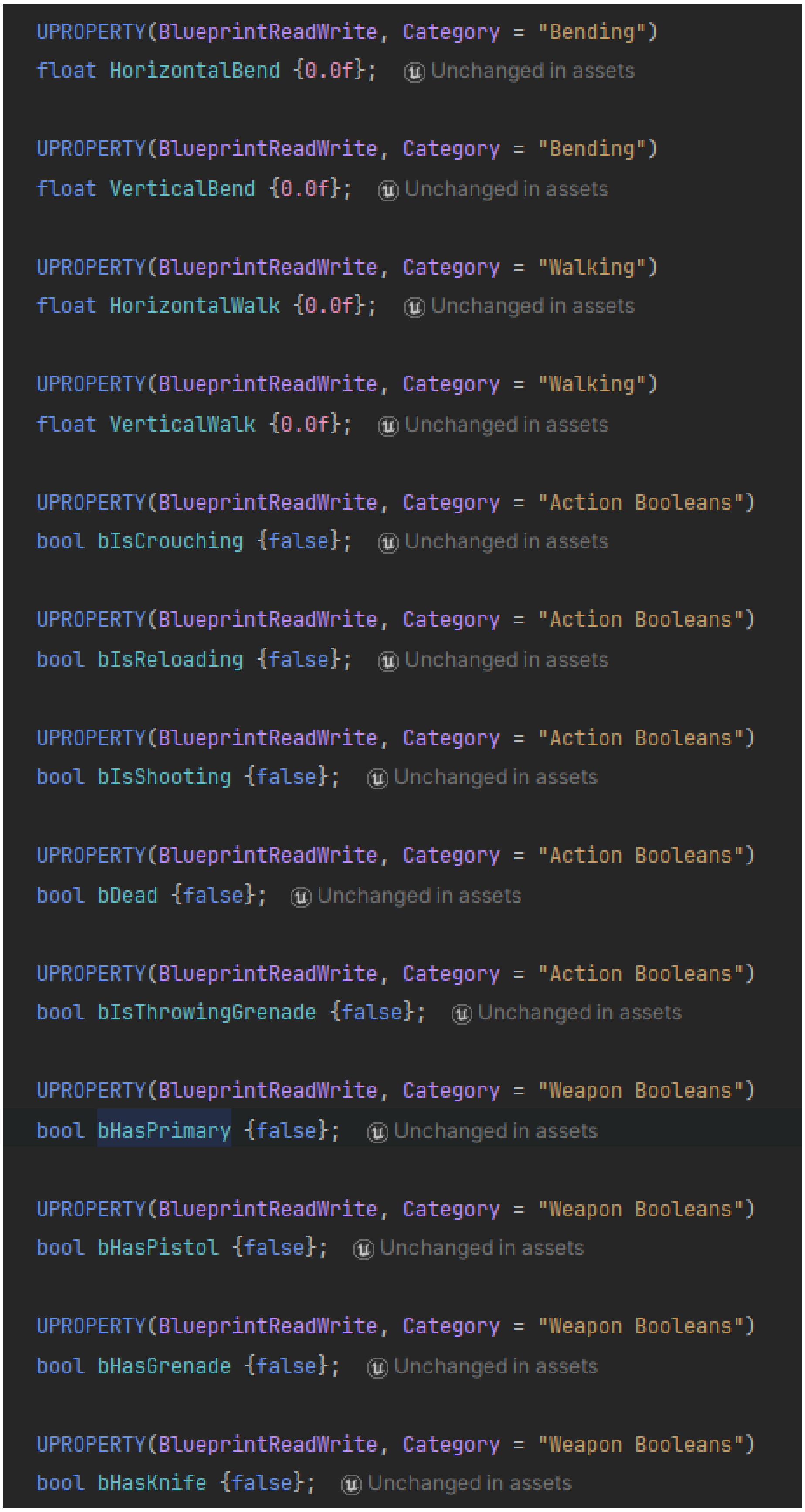


Name	Path	Disk Size
A_Bend_Back	/All/Game/A	69.636 KiB
A_Bend_Forward	/All/Game/A	69.427 KiB
A_BendLeft	/All/Game/A	69.408 KiB
A_BendRight	/All/Game/A	69.412 KiB
AO_SWAT_Peeking	/All/Game/A	10.913 KiB
AO_SWAT_Pistol	/All/Game/A	11.035 KiB
BS_SWAT_Movement	/All/Game/A	13.083 KiB
BS_SWAT_Movement	/All/Game/A	12.885 KiB
Crouch_Idle	/All/Game/A	959.734 KiB
Frisbee_Throw	/All/Game/A	566.383 KiB
Gunplay	/All/Game/A	128.793 KiB
Idle	/All/Game/A	195.463 KiB
Idle_Aiming	/All/Game/A	189.546 KiB

39 items (1 selected)

I also created a state machine for player actions, with Booleans determining which animation to play depending on the current player action. I included these variables into the player controller code and added logic to toggle the truth values of said variables.

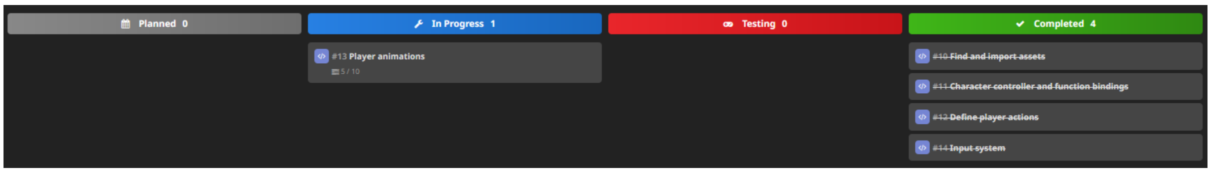




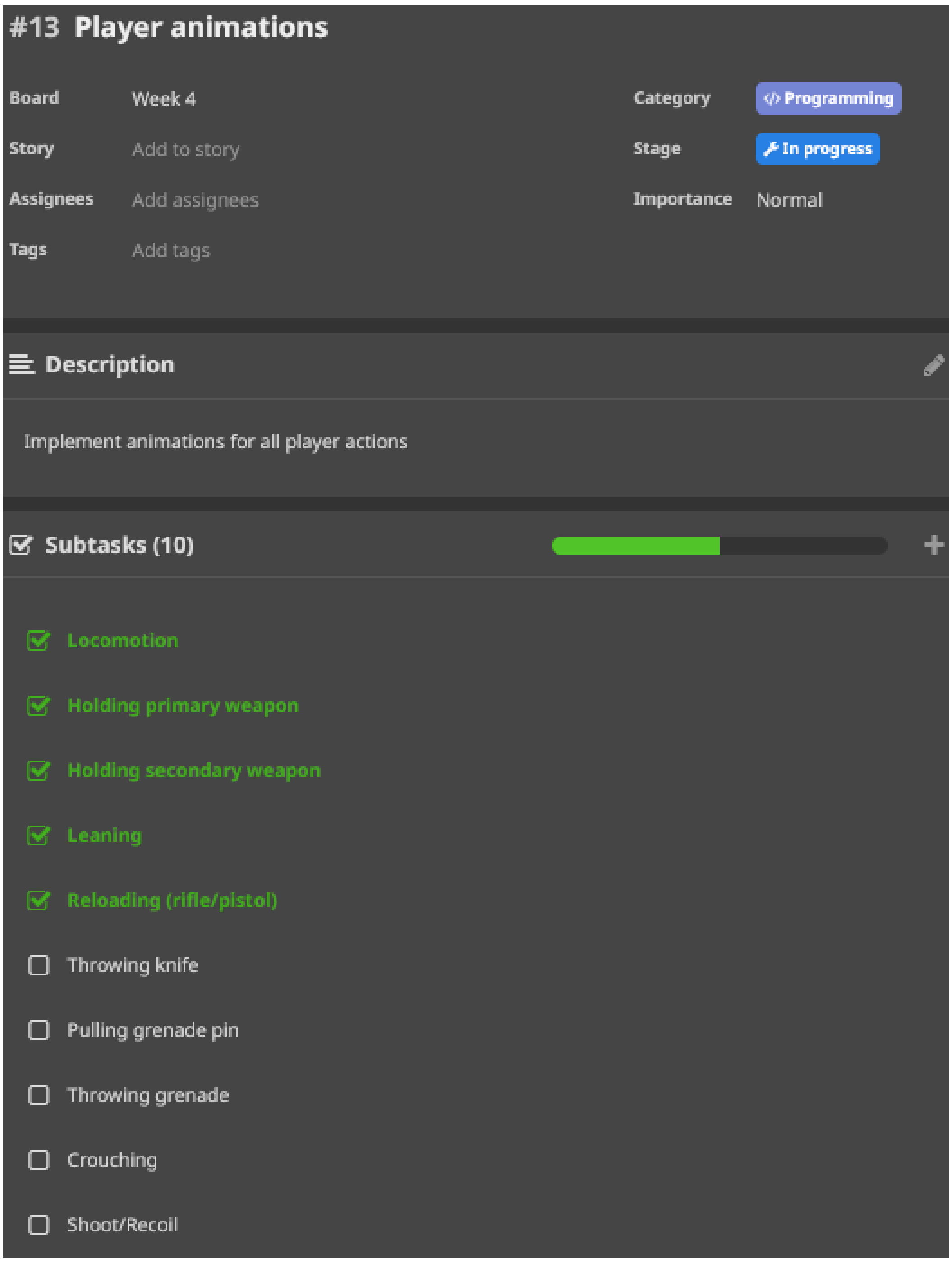
```
UPROPERTY(BlueprintReadWrite, Category = "Bending")
float HorizontalBend {0.0f};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Bending")
float VerticalBend {0.0f};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Walking")
float HorizontalWalk {0.0f};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Walking")
float VerticalWalk {0.0f};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Action Booleans")
bool bIsCrouching {false};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Action Booleans")
bool bIsReloading {false};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Action Booleans")
bool bIsShooting {false};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Action Booleans")
bool bDead {false};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Action Booleans")
bool bIsThrowingGrenade {false};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Weapon Booleans")
bool bHasPrimary {false};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Weapon Booleans")
bool bHasPistol {false};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Weapon Booleans")
bool bHasGrenade {false};  Unchanged in assets

UPROPERTY(BlueprintReadWrite, Category = "Weapon Booleans")
bool bHasKnife {false};  Unchanged in assets
```

Kanban board – end of the week:

I managed to complete all the set tasks, with animations remaining only half-completed since the scope of the animation system overlaps with the second week.



Planned 0	In Progress 1	Testing 0	Completed 4
	#13 Player animations 5 / 10		~~#10 Find and import assets~~
			~~#11 Character controller and function bindings~~
			~~#12 Define player actions~~
			~~#14 Input system~~



#13 Player animations

Board	Week 4	Category	Programming
Story	Add to story	Stage	In progress
Assignees	Add assignees	Importance	Normal
Tags	Add tags		

Description

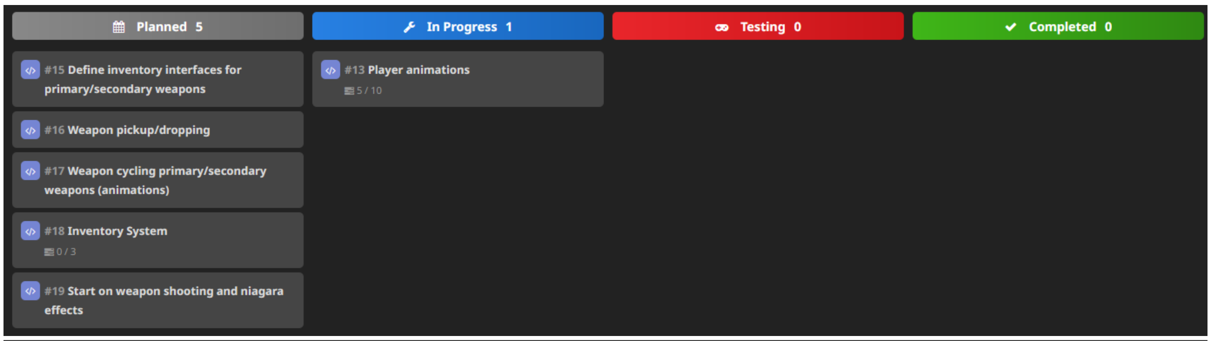
Implement animations for all player actions

Subtasks (10)

- [x] Locomotion
- [x] Holding primary weapon
- [x] Holding secondary weapon
- [x] Leaning
- [x] Reloading (rifle/pistol)
- [ ] Throwing knife
- [ ] Pulling grenade pin
- [ ] Throwing grenade
- [ ] Crouching
- [ ] Shoot/Recoil

Week 2 (26/02/2025-05/03/2025)

Kanban – start of the week:



Planned 5

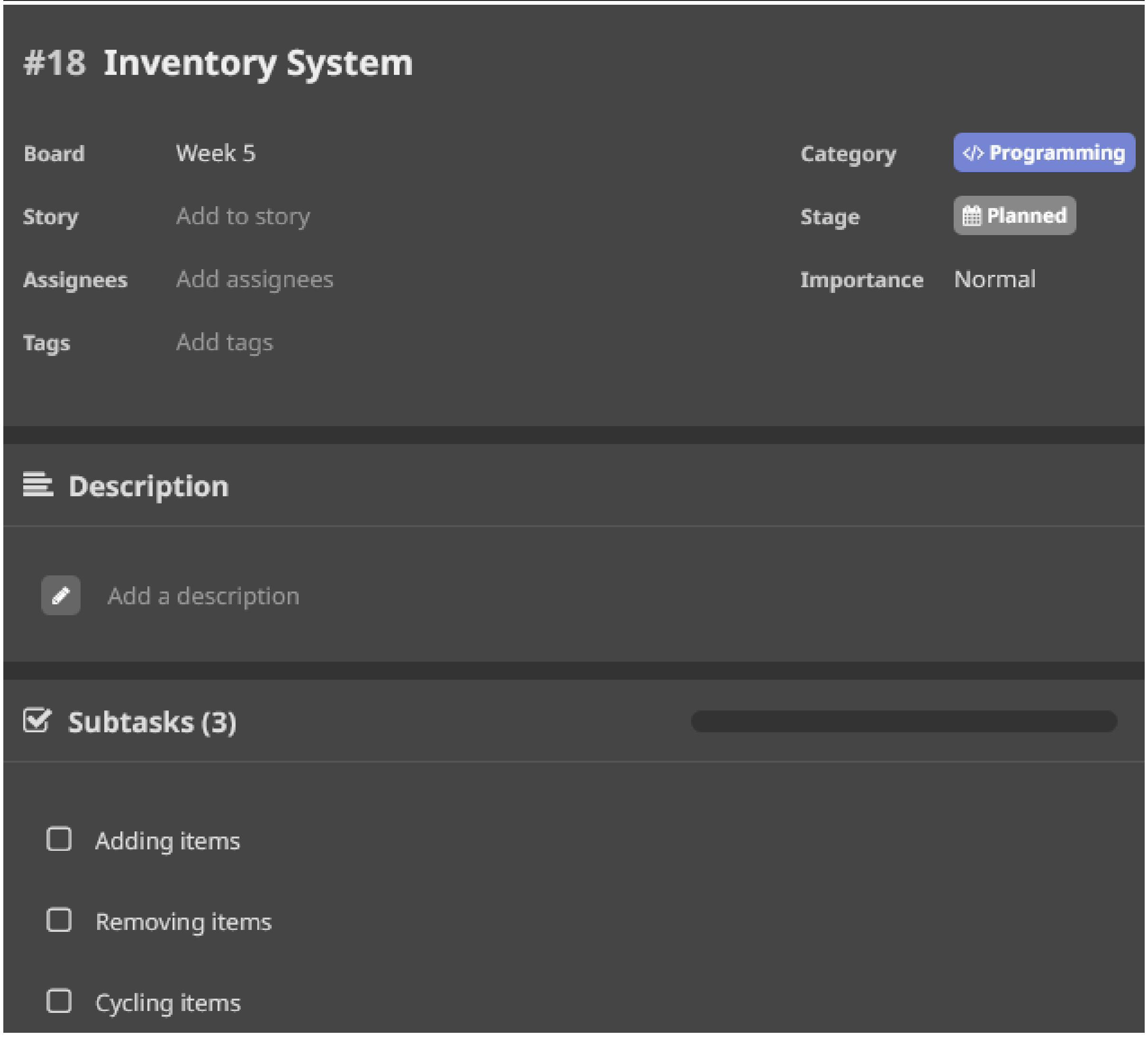
- #15 Define inventory interfaces for primary/secondary weapons
- #16 Weapon pickup/dropping
- #17 Weapon cycling primary/secondary weapons (animations)
- #18 Inventory System 0 / 3
- #19 Start on weapon shooting and niagara effects

In Progress 1

- #13 Player animations 5 / 10

Testing 0

Completed 0



#18 Inventory System

Board	Week 5	Category	Programming
Story	Add to story	Stage	Planned
Assignees	Add assignees	Importance	Normal
Tags	Add tags		

Description

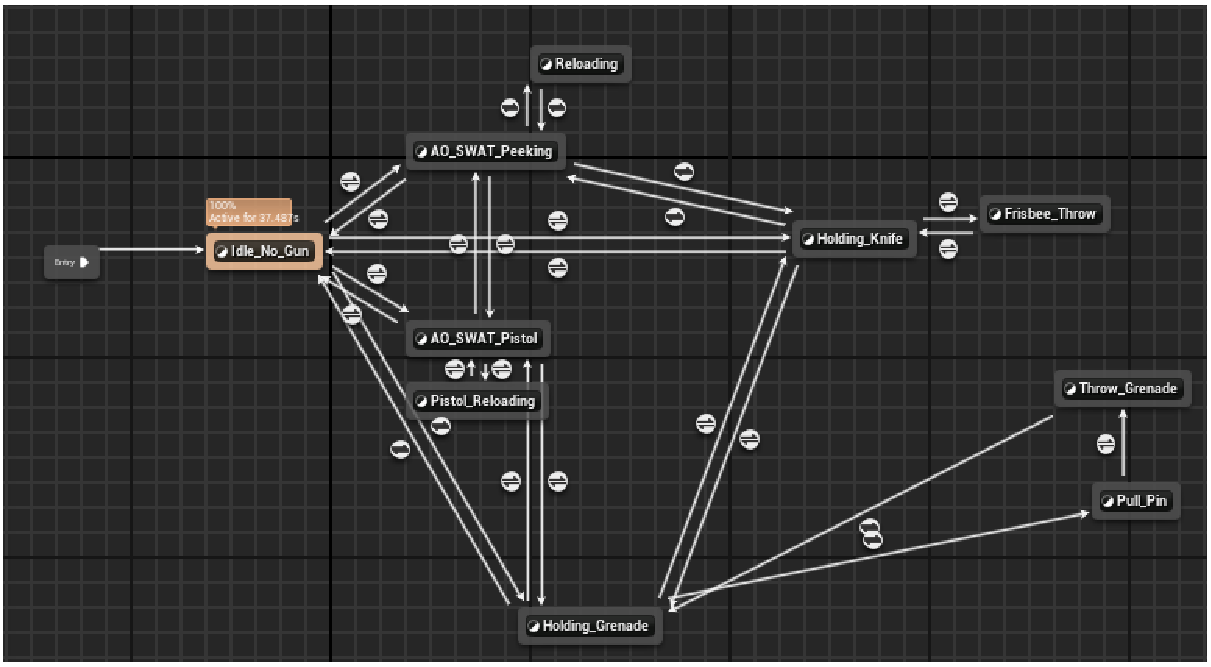
Add a description

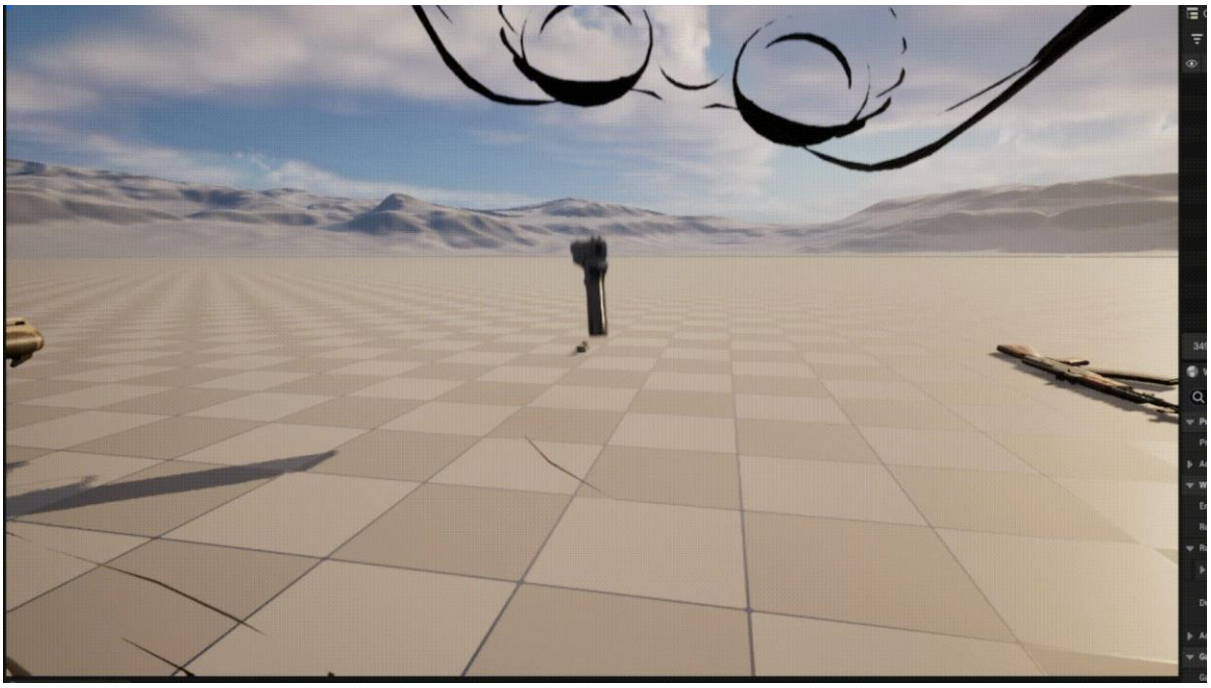
Subtasks (3)

- [ ] Adding items
- [ ] Removing items
- [ ] Cycling items

Player animations:

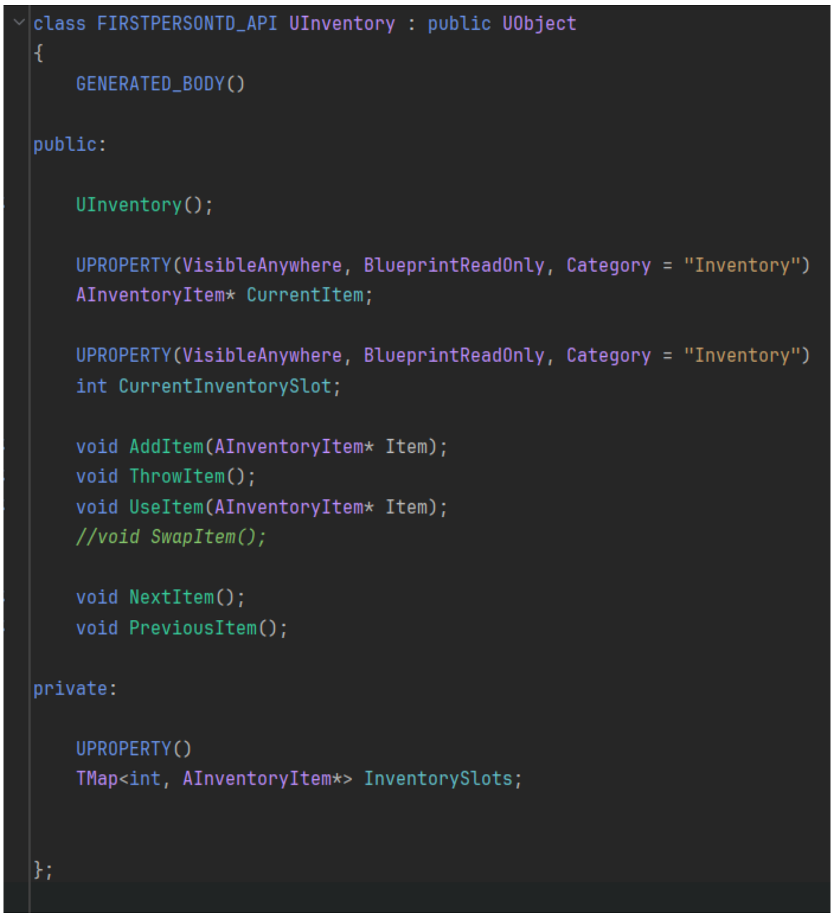
I finished the player animation system started last week. The most notable additions were the blending of the crouching with the rest of the animations, as well as pulling the grenade pin and throwing it. I must mention that at this stage, some of the animations, particularly the weapon-related ones are tied with the inventory system, therefore are still pending implementation once the inventory system is complete. Please find more on this below in the weapon cycling section.





Inventory system (WIP):

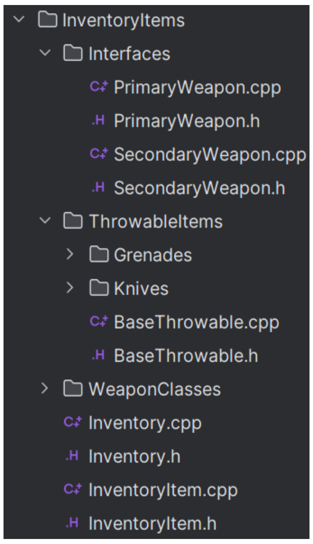
The inventory system task was a challenging one and is still pending the additions of grenades and the knife. However, it is made modularly, so that any amount of weapons can be added and cycled through. This task is broken down further in the next three sections. Here I will talk about defining the basic actions of the inventory system, such as adding and/or removing items, using and cycling through them. As you can see below, I created the basic function definitions and properties that allow for the handling of any inventory item.



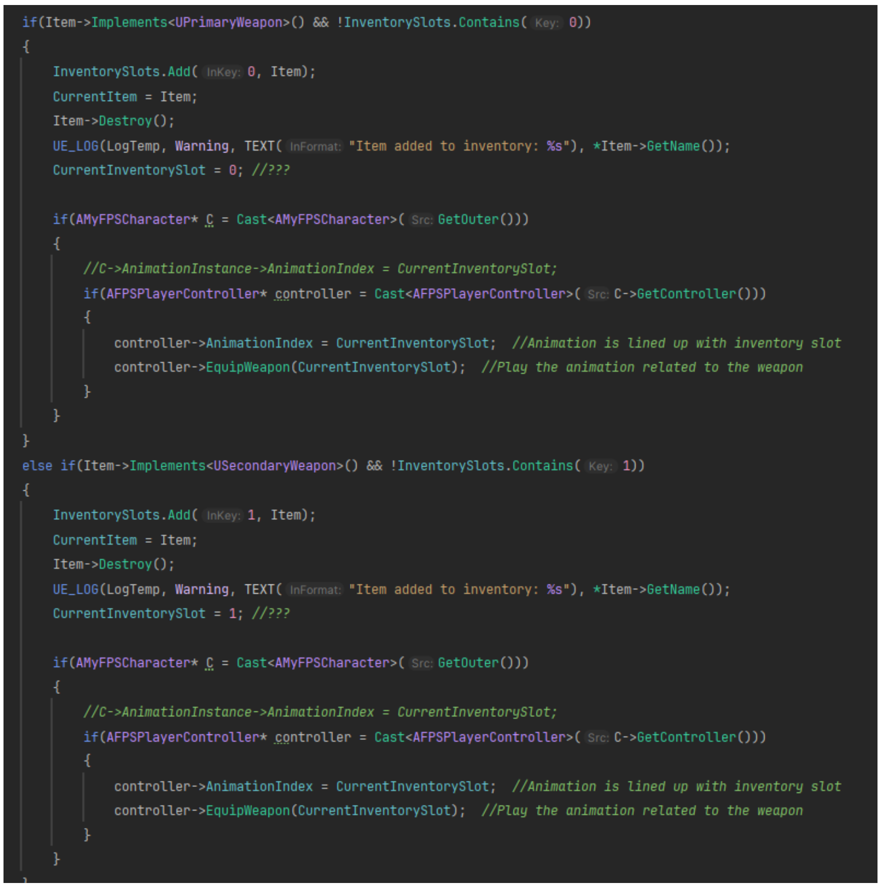
```
class FIRSTPERSONTD_API UInventory : public UObject
{
    GENERATED_BODY()

public:

    UInventory();

    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Inventory")
    AInventoryItem* CurrentItem;

    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Inventory")
    int CurrentInventorySlot;

    void AddItem(AInventoryItem* Item);
    void ThrowItem();
    void UseItem(AInventoryItem* Item);
    //void SwapItem();

    void NextItem();
    void PreviousItem();

private:

    UPROPERTY()
    TMap<int, AInventoryItem*> InventorySlots;


};
```

Interface definition:

The key part of the inventory system was defining the interfaces that would allow for inventory sorting and organizing. Each weapon has its own type and implements an interface based on said type. The inventory function responsible for adding items then makes a decision based on the interface on what index to give the item and what animation to play. So far, interfaces for primary and secondary weapons have been implemented. Please find the code below.



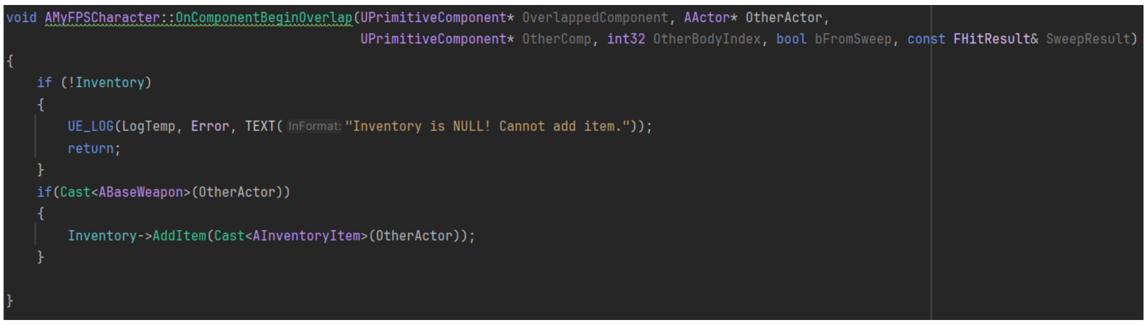
```
InventoryItems
    Interfaces
        PrimaryWeapon.cpp
        PrimaryWeapon.h
        SecondaryWeapon.cpp
        SecondaryWeapon.h
    ThrowableItems
        Grenades
        Knives
        BaseThrowable.cpp
        BaseThrowable.h
    WeaponClasses
    Inventory.cpp
    Inventory.h
    InventoryItem.cpp
    InventoryItem.h
```



```cpp
if(Item->Implements<UPrimaryWeapon>() && !InventorySlots.Contains( Key: 0))
{
    InventorySlots.Add( InKey: 0, Item);
    CurrentItem = Item;
    Item->Destroy();
    UE_LOG(LogTemp, Warning, TEXT( InFormat: "Item added to inventory: %s"), *Item->GetName());
    CurrentInventorySlot = 0; //???

    if(AMyFPSCharacter* C = Cast<AMyFPSCharacter>( Src: GetOuter()))
    {
        //C->AnimationInstance->AnimationIndex = CurrentInventorySlot;
        if(AFPSPlayerController* controller = Cast<AFPSPlayerController>( Src: C->GetController()))
        {
            controller->AnimationIndex = CurrentInventorySlot;  //Animation is lined up with inventory slot
            controller->EquipWeapon(CurrentInventorySlot);  //Play the animation related to the weapon
        }
    }
}
else if(Item->Implements<USecondaryWeapon>() && !InventorySlots.Contains( Key: 1))
{
    InventorySlots.Add( InKey: 1, Item);
    CurrentItem = Item;
    Item->Destroy();
    UE_LOG(LogTemp, Warning, TEXT( InFormat: "Item added to inventory: %s"), *Item->GetName());
    CurrentInventorySlot = 1; //???

    if(AMyFPSCharacter* C = Cast<AMyFPSCharacter>( Src: GetOuter()))
    {
        //C->AnimationInstance->AnimationIndex = CurrentInventorySlot;
        if(AFPSPlayerController* controller = Cast<AFPSPlayerController>( Src: C->GetController()))
        {
            controller->AnimationIndex = CurrentInventorySlot;  //Animation is lined up with inventory slot
            controller->EquipWeapon(CurrentInventorySlot);  //Play the animation related to the weapon
        }
    }
```

Weapon pickup/drop off:

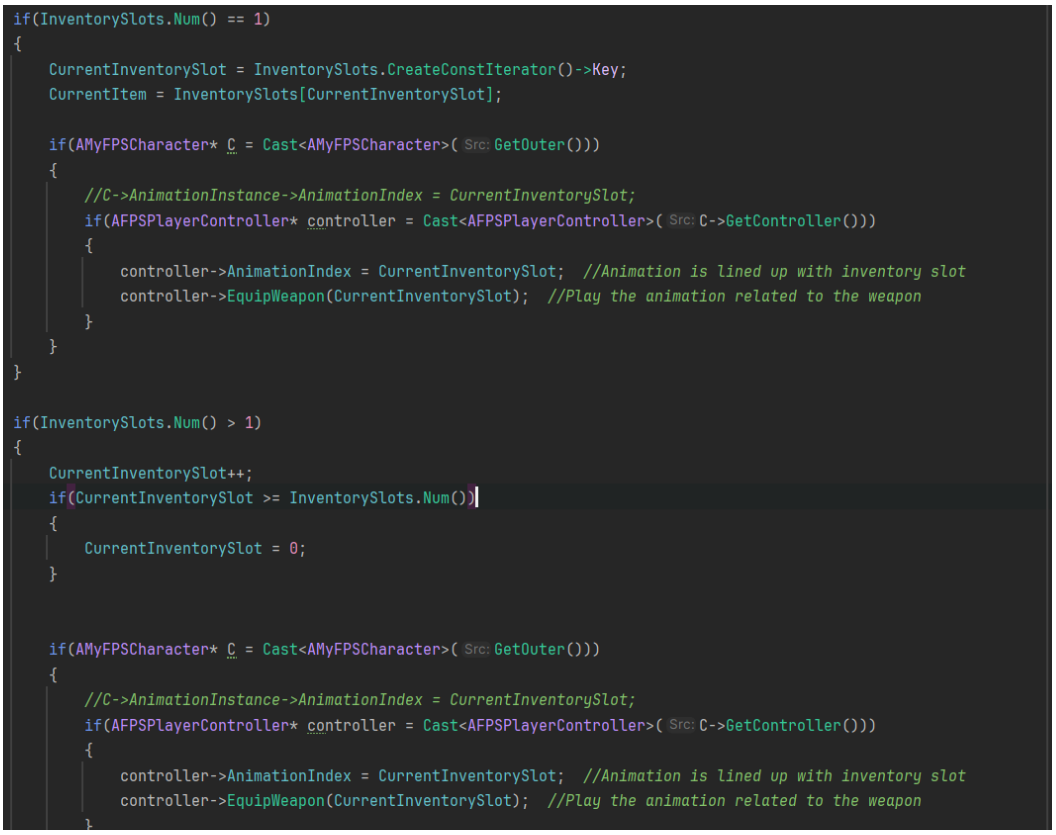
This task was fairly straight-forward and involved setting mesh collisions for each weapon then defining the pick up logic on overlap that would allow the weapon to be picked up. Note in the available videos that the weapons are still not attached to the body, however the logic still persists.



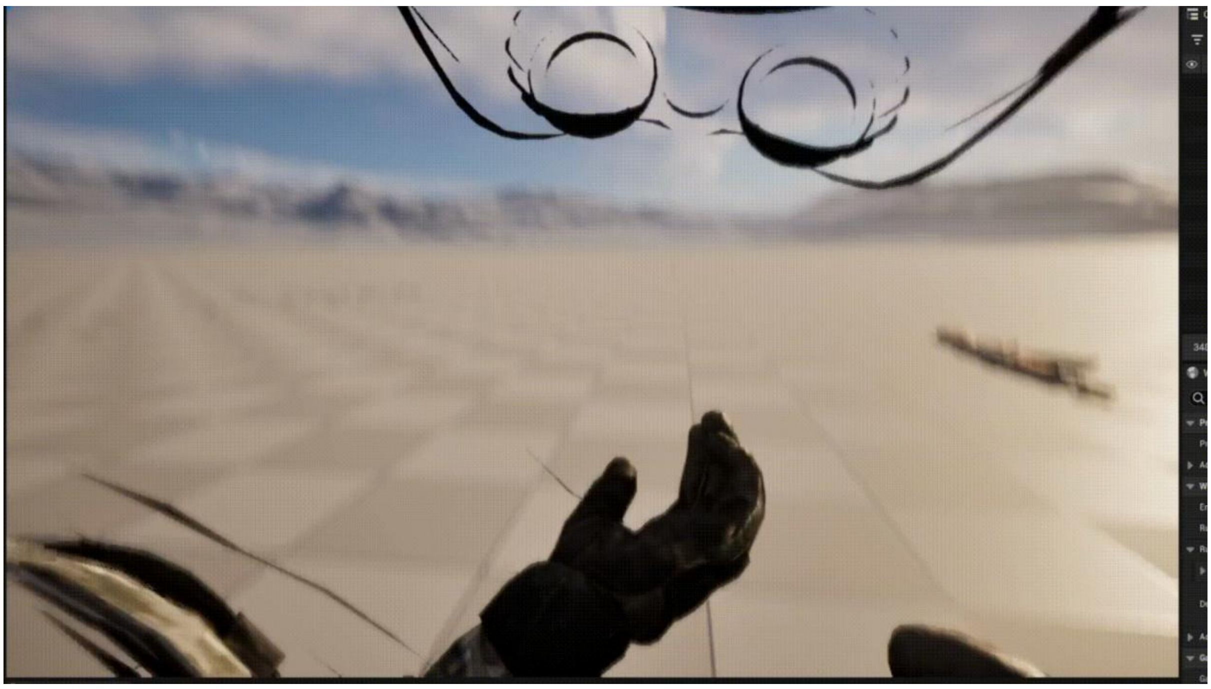
```cpp
void AMyFPSCharacter::OnComponentBeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
                                              UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    if (!Inventory)
    {
        UE_LOG(LogTemp, Error, TEXT( InFormat: "Inventory is NULL! Cannot add item."));
        return;
    }
    if(Cast<ABaseWeapon>(OtherActor))
    {
        Inventory->AddItem(Cast<AInventoryItem>(OtherActor));
    }

}
```

Weapon cycling animations:

This task is tightly coupled with the others, mainly that the inventory and the animations always need to align. I have done this by introducing some Boolean values to the player animation instance, which I change the value of when swapping weapons. As you can see below, I have an animation index that keeps track of the current animation needed.

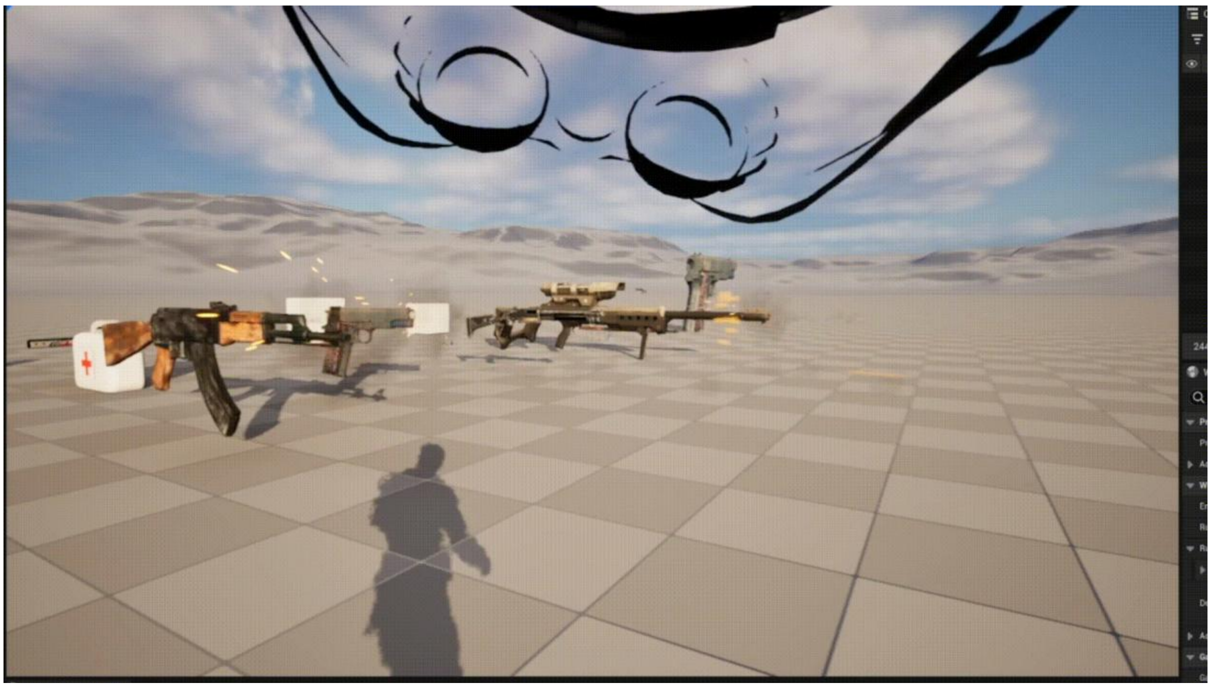


```
if(InventorySlots.Num() == 1)
{
    CurrentInventorySlot = InventorySlots.CreateConstIterator()->Key;
    CurrentItem = InventorySlots[CurrentInventorySlot];

    if(AMyFPSCharacter* C = Cast<AMyFPSCharacter>( Src: GetOuter()))
    {
        //C->AnimationInstance->AnimationIndex = CurrentInventorySlot;
        if(AFPSPlayerController* controller = Cast<AFPSPlayerController>( Src: C->GetController()))
        {
            controller->AnimationIndex = CurrentInventorySlot;  //Animation is lined up with inventory slot
            controller->EquipWeapon(CurrentInventorySlot);  //Play the animation related to the weapon
        }
    }
}

if(InventorySlots.Num() > 1)
{
    CurrentInventorySlot++;
    if(CurrentInventorySlot >= InventorySlots.Num())
    {
        CurrentInventorySlot = 0;
    }


    if(AMyFPSCharacter* C = Cast<AMyFPSCharacter>( Src: GetOuter()))
    {
        //C->AnimationInstance->AnimationIndex = CurrentInventorySlot;
        if(AFPSPlayerController* controller = Cast<AFPSPlayerController>( Src: C->GetController()))
        {
            controller->AnimationIndex = CurrentInventorySlot;  //Animation is lined up with inventory slot
            controller->EquipWeapon(CurrentInventorySlot);  //Play the animation related to the weapon
        }
```



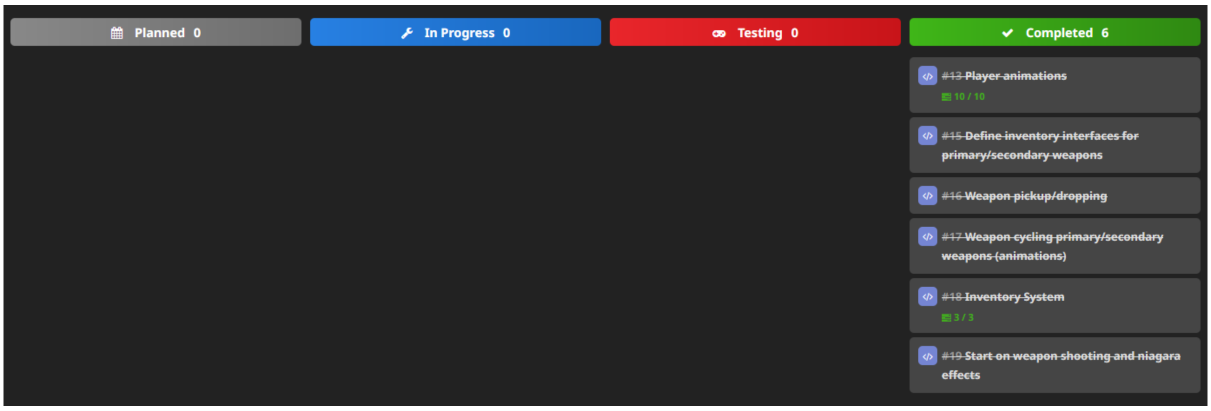
Weapon shooting and Niagara effects (WIP):

I must start this section by mentioning this is still work in progress at the time of making this devlog, as it was not on the top of my priority list for this week. However, I believe the weapons system and the respective effects go hand in hand, particularly with my implementation, so I decided to lay some ground for it for later weeks. Each weapon has its own projectile class and instantiates one (or multiple in the case of the shotgun) projectiles upon shooting. Each projectile has its own mesh and a Niagara effect attached to it to simulate bullet traces.



Kanban – end of the week:

I completed all the set tasks and their respective subtasks within the week with the shooting and Niagara effects on the weapons system being still work in progress. The inventory system also remains work in progress at this stage, but has been completed for the scope of this week, that is for primary and secondary weapons. Next week I plan to attach the weapons to the body on pickup and add grenades and the knife to the inventory/animation system.



Planned 0

In Progress 0

Testing 0

Completed 6

~~#13 Player animations~~
10 / 10

~~#15 Define inventory interfaces for primary/secondary weapons~~

~~#16 Weapon pickup/dropping~~

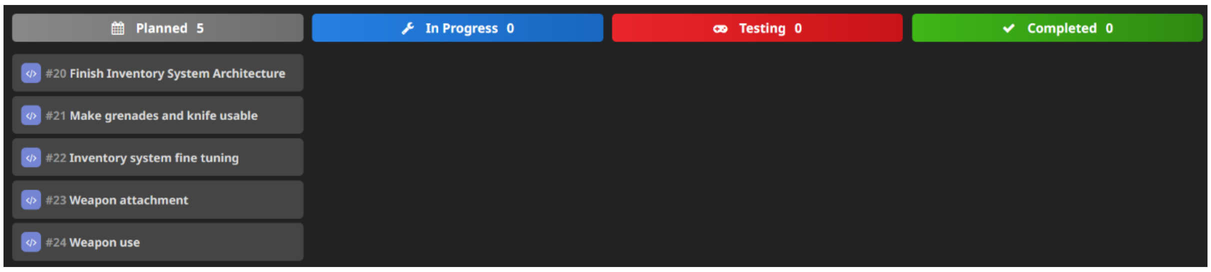
~~#17 Weapon cycling primary/secondary weapons (animations)~~

~~#18 Inventory System~~
3 / 3

~~#19 Start on weapon shooting and niagara effects~~

Week 3 (05/03/2025-12/03/2025)

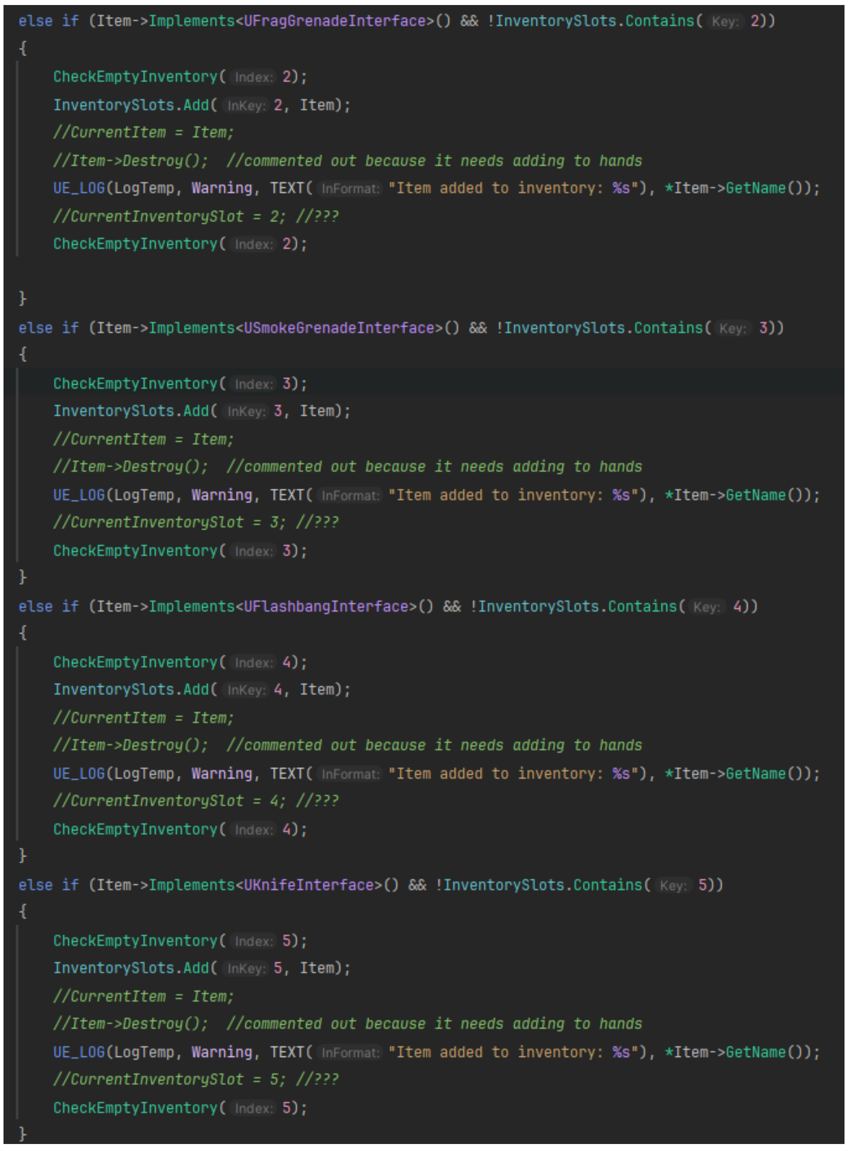
Kanban – start of the week:


Planned 5 | In Progress 0 | Testing 0 | Completed 0

- #20 Finish Inventory System Architecture
- #21 Make grenades and knife usable
- #22 Inventory system fine tuning
- #23 Weapon attachment
- #24 Weapon use

Inventory System Architecture:

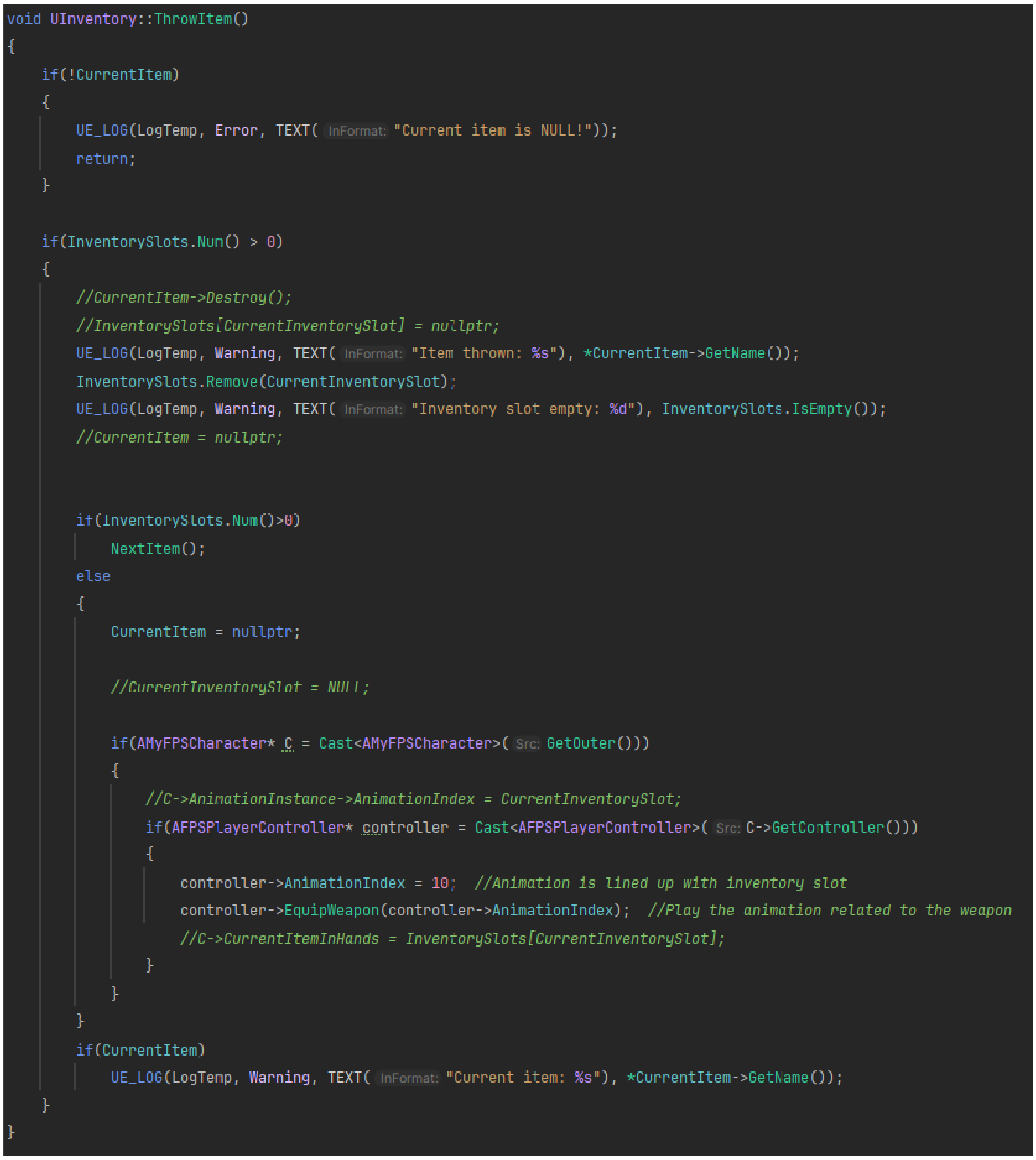
I managed to complete the inventory system architecture by introducing interfaces and use cases for all inventory items. Each weapon and class has an individual interface that the inventory checks for when adding the item. It now gives each item a value and sorts the array, aligning the animation as well.



```cpp
else if (Item->Implements<UFragGrenadeInterface>() && !InventorySlots.Contains( Key: 2))
{
    CheckEmptyInventory( Index: 2);
    InventorySlots.Add( InKey: 2, Item);
    //CurrentItem = Item;
    //Item->Destroy();  //commented out because it needs adding to hands
    UE_LOG(LogTemp, Warning, TEXT( InFormat: "Item added to inventory: %s"), *Item->GetName());
    //CurrentInventorySlot = 2; //???
    CheckEmptyInventory( Index: 2);

}
else if (Item->Implements<USmokeGrenadeInterface>() && !InventorySlots.Contains( Key: 3))
{
    CheckEmptyInventory( Index: 3);
    InventorySlots.Add( InKey: 3, Item);
    //CurrentItem = Item;
    //Item->Destroy();  //commented out because it needs adding to hands
    UE_LOG(LogTemp, Warning, TEXT( InFormat: "Item added to inventory: %s"), *Item->GetName());
    //CurrentInventorySlot = 3; //???
    CheckEmptyInventory( Index: 3);
}
else if (Item->Implements<UFlashbangInterface>() && !InventorySlots.Contains( Key: 4))
{
    CheckEmptyInventory( Index: 4);
    InventorySlots.Add( InKey: 4, Item);
    //CurrentItem = Item;
    //Item->Destroy();  //commented out because it needs adding to hands
    UE_LOG(LogTemp, Warning, TEXT( InFormat: "Item added to inventory: %s"), *Item->GetName());
    //CurrentInventorySlot = 4; //???
    CheckEmptyInventory( Index: 4);
}
else if (Item->Implements<UKnifeInterface>() && !InventorySlots.Contains( Key: 5))
{
    CheckEmptyInventory( Index: 5);
    InventorySlots.Add( InKey: 5, Item);
    //CurrentItem = Item;
    //Item->Destroy();  //commented out because it needs adding to hands
    UE_LOG(LogTemp, Warning, TEXT( InFormat: "Item added to inventory: %s"), *Item->GetName());
    //CurrentInventorySlot = 5; //???
    CheckEmptyInventory( Index: 5);
}
```

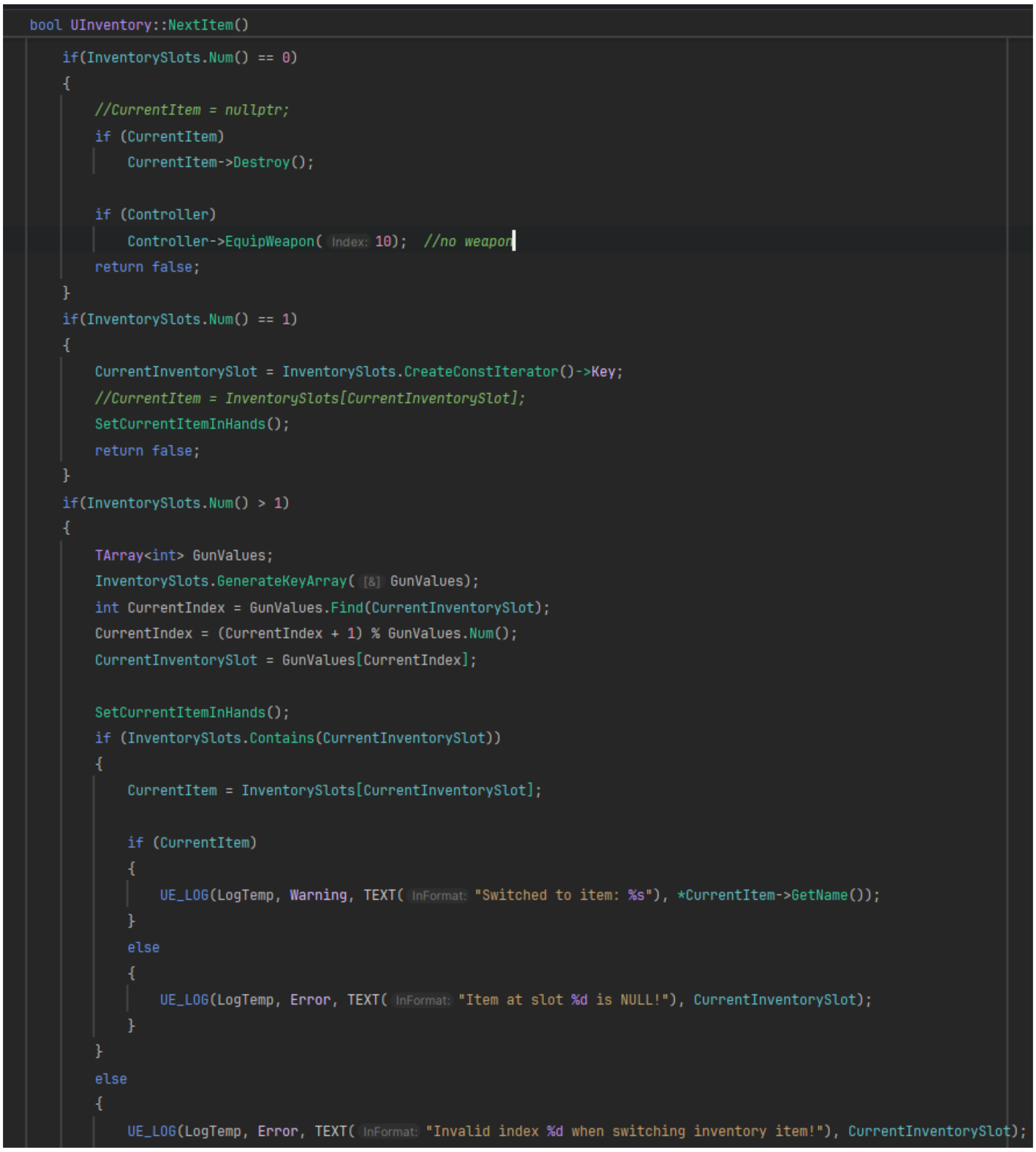
Inventory System Fine-Tuning:

I improved the weapon throwing and made all the new inventory items throwable. The inventory now circles to the next item if one is thrown, spawning it and aligning the animation.

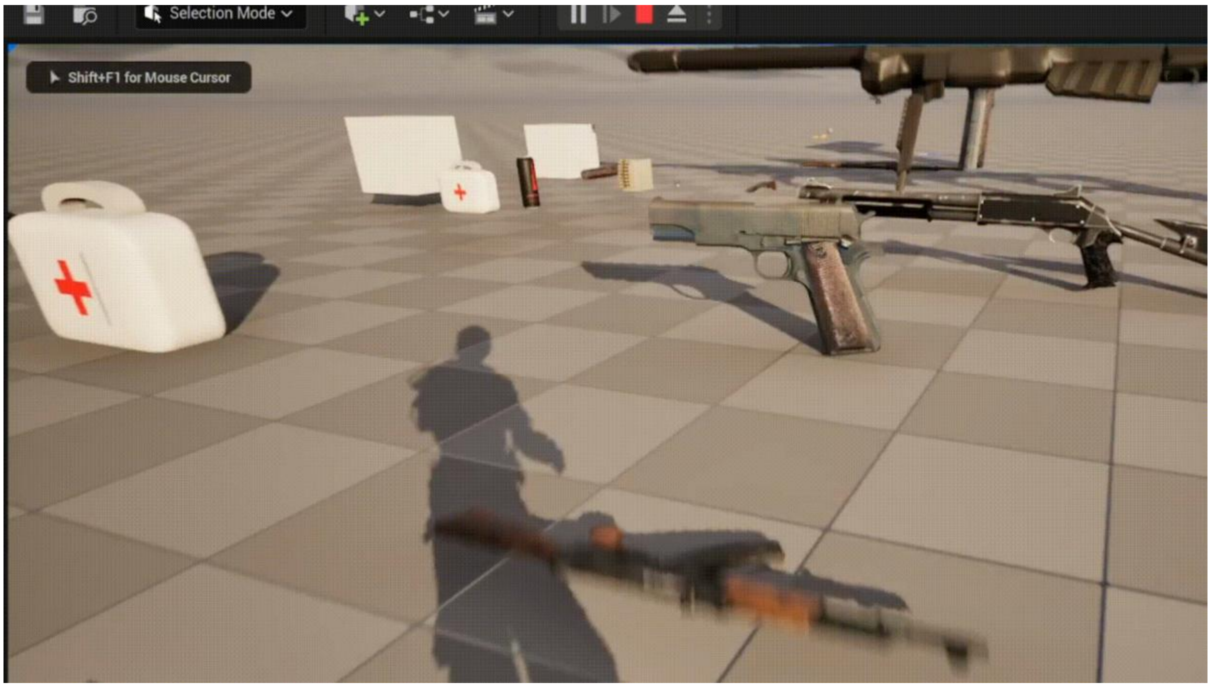


```cpp
void UInventory::ThrowItem()
{
    if(!CurrentItem)
    {
        UE_LOG(LogTemp, Error, TEXT( InFormat: "Current item is NULL!"));
        return;
    }

    if(InventorySlots.Num() > 0)
    {
        //CurrentItem->Destroy();
        //InventorySlots[CurrentInventorySlot] = nullptr;
        UE_LOG(LogTemp, Warning, TEXT( InFormat: "Item thrown: %s"), *CurrentItem->GetName());
        InventorySlots.Remove(CurrentInventorySlot);
        UE_LOG(LogTemp, Warning, TEXT( InFormat: "Inventory slot empty: %d"), InventorySlots.IsEmpty());
        //CurrentItem = nullptr;


        if(InventorySlots.Num()>0)
            NextItem();
        else
        {
            CurrentItem = nullptr;

            //CurrentInventorySlot = NULL;

            if(AMyFPSCharacter* C = Cast<AMyFPSCharacter>( Src: GetOuter()))
            {
                //C->AnimationInstance->AnimationIndex = CurrentInventorySlot;
                if(AFPSPlayerController* controller = Cast<AFPSPlayerController>( Src: C->GetController()))
                {
                    controller->AnimationIndex = 10;  //Animation is lined up with inventory slot
                    controller->EquipWeapon(controller->AnimationIndex);  //Play the animation related to the weapon
                    //C->CurrentItemInHands = InventorySlots[CurrentInventorySlot];
                }
            }
        }
        if(CurrentItem)
            UE_LOG(LogTemp, Warning, TEXT( InFormat: "Current item: %s"), *CurrentItem->GetName());
    }
}
```

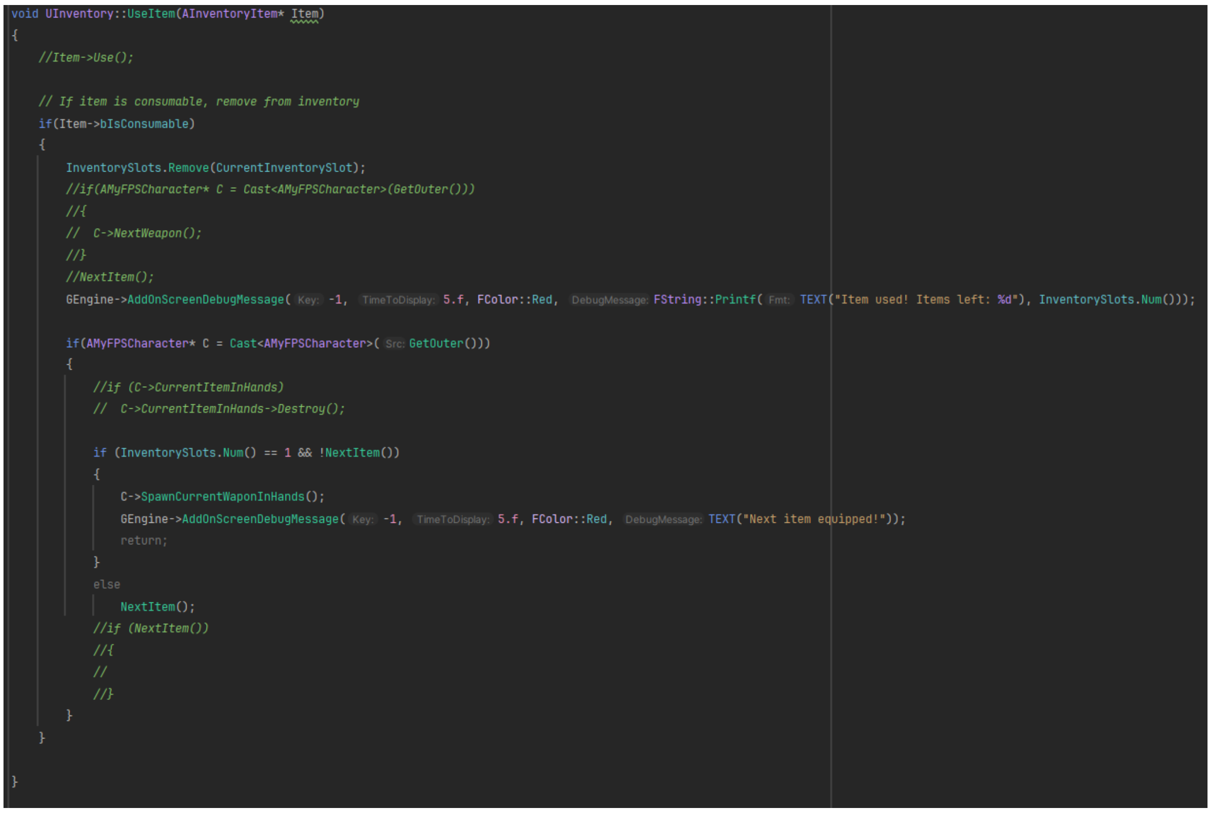
I also improved the next item function by adding null pointer checks and changing the current animation index to ‘No Weapon’ in case all weapons are thrown. There are still some bugs that are normal at this stage with ensuring the new weapon spawning is consistent in location and size. The gif you will see below showcases said bug that is actively being worked on.



```
bool UInventory::NextItem()
    if(InventorySlots.Num() == 0)
    {
        //CurrentItem = nullptr;
        if (CurrentItem)
            CurrentItem->Destroy();

        if (Controller)
            Controller->EquipWeapon( Index: 10);  //no weapon
        return false;
    }
    if(InventorySlots.Num() == 1)
    {
        CurrentInventorySlot = InventorySlots.CreateConstIterator()->Key;
        //CurrentItem = InventorySlots[CurrentInventorySlot];
        SetCurrentItemInHands();
        return false;
    }
    if(InventorySlots.Num() > 1)
    {
        TArray<int> GunValues;
        InventorySlots.GenerateKeyArray( [&] GunValues);
        int CurrentIndex = GunValues.Find(CurrentInventorySlot);
        CurrentIndex = (CurrentIndex + 1) % GunValues.Num();
        CurrentInventorySlot = GunValues[CurrentIndex];

        SetCurrentItemInHands();
        if (InventorySlots.Contains(CurrentInventorySlot))
        {
            CurrentItem = InventorySlots[CurrentInventorySlot];

            if (CurrentItem)
            {
                UE_LOG(LogTemp, Warning, TEXT( InFormat: "Switched to item: %s"), *CurrentItem->GetName());
            }
            else
            {
                UE_LOG(LogTemp, Error, TEXT( InFormat: "Item at slot %d is NULL!"), CurrentInventorySlot);
            }
        }
        else
        {
            UE_LOG(LogTemp, Error, TEXT( InFormat: "Invalid index %d when switching inventory item!"), CurrentInventorySlot);
```



I then added a function for using the inventory item that creates different behaviour based on what the item is.



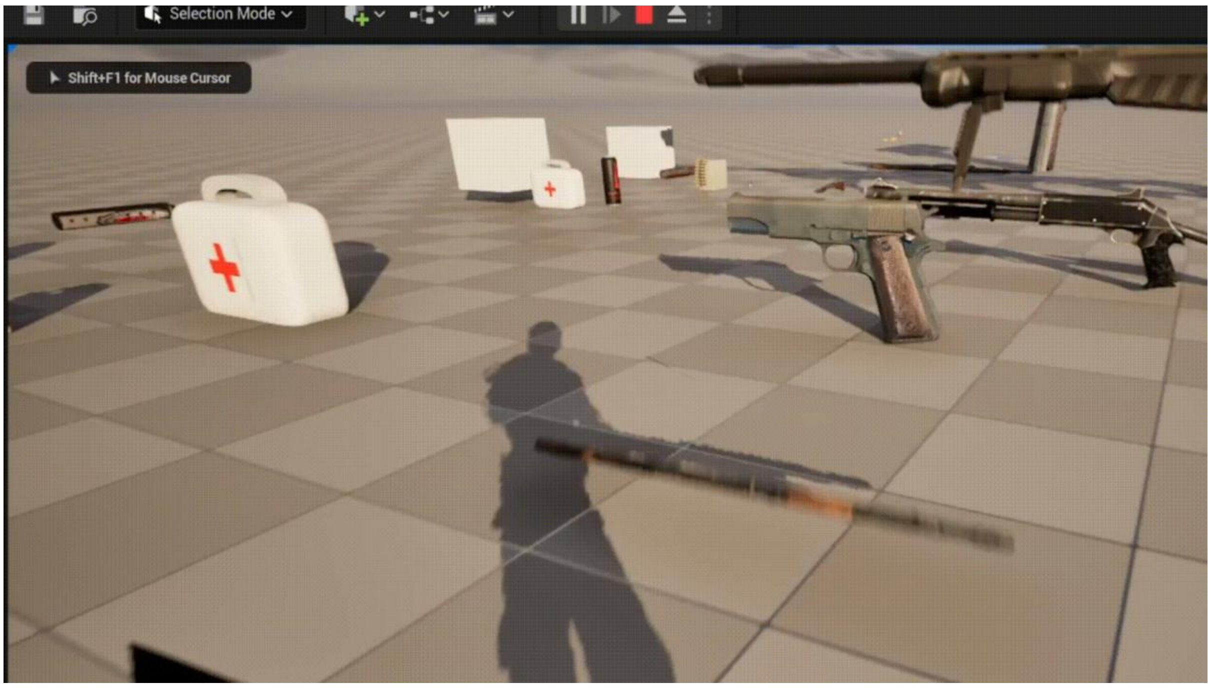
```
void UInventory::UseItem(AInventoryItem* Item)
{
    //Item->Use();

    // If item is consumable, remove from inventory
    if(Item->bIsConsumable)
    {
        InventorySlots.Remove(CurrentInventorySlot);
        //if(AMyFPSCharacter* C = Cast<AMyFPSCharacter>(GetOuter()))
        //{
        //  C->NextWeapon();
        //}
        //NextItem();
        GEngine->AddOnScreenDebugMessage( Key: -1,  TimeToDisplay: 5.f, FColor::Red,  DebugMessage: FString::Printf( Fmt: TEXT("Item used! Items left: %d"), InventorySlots.Num()));

        if(AMyFPSCharacter* C = Cast<AMyFPSCharacter>( Src: GetOuter()))
        {
            //if (C->CurrentItemInHands)
            //  C->CurrentItemInHands->Destroy();

            if (InventorySlots.Num() == 1 && !NextItem())
            {
                C->SpawnCurrentWaponInHands();
                GEngine->AddOnScreenDebugMessage( Key: -1,  TimeToDisplay: 5.f, FColor::Red,  DebugMessage: TEXT("Next item equipped!"));
                return;
            }
            else
                NextItem();
            //if (NextItem())
            //{
            //
            //}
        }
    }

}
```

This closely relates to the next two sections, but I shall explain the principle in the next few sentences. As per design, all weapons derive from a parent class – 'Inventory Item'.

This parent class has one virtual function called 'Use' that creates the necessary behaviour for any particular item. Each child class then overrides said function with its own behaviour, whether it is a weapon spawning a projectile or a usable item like a grenade exploding. When an item is used, its function mandates what happens with it. Please find the two main examples in the sections below.

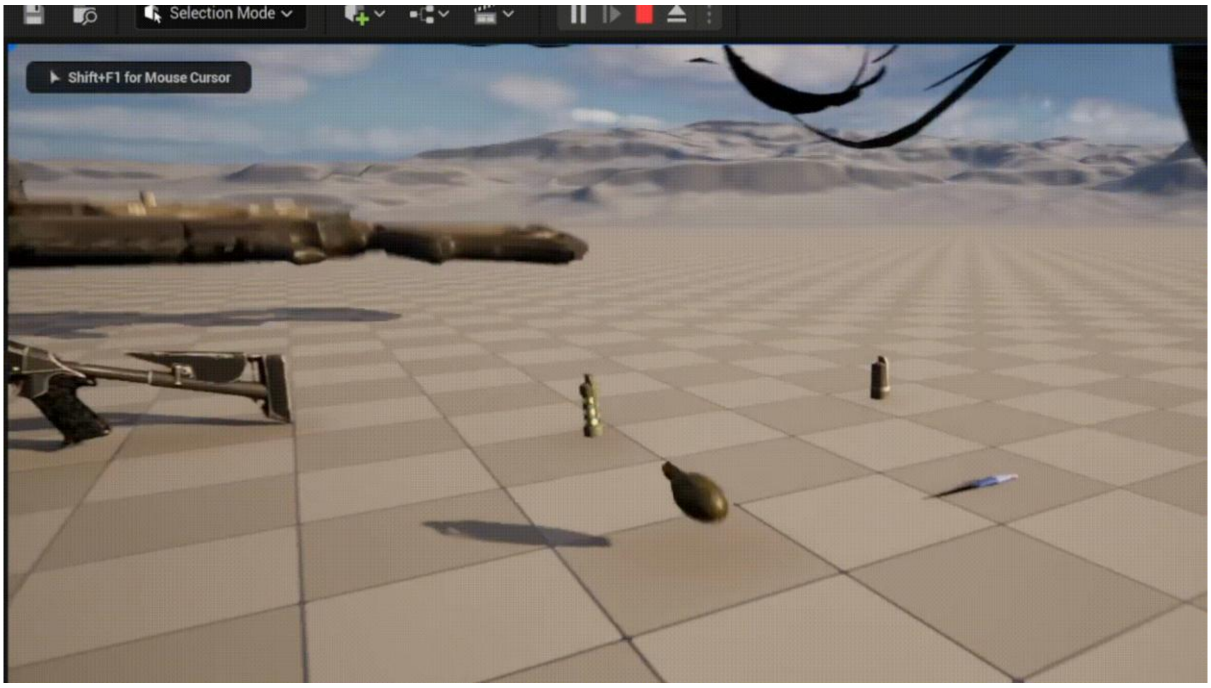
Weapon Attachment and Use (WIP):

The weapon attachment is still work in progress due to me not being able to find a reliable and scalable way to create modular attachments that look good and feel natural to the player. However, the weapons do attach to the weapons socket and can be used once taken. Unsurprisingly, the weapons shoots when it is used. In the gif below you will also be able to see some juice added through bullet landing holes.

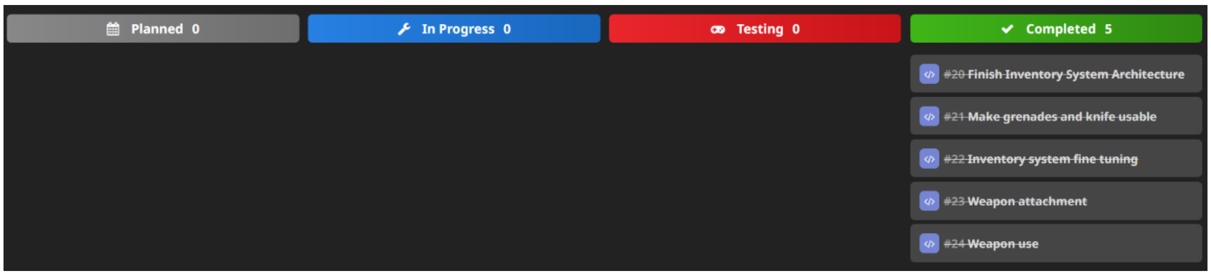


Grenade and Knife Use:

As mentioned above, the throwables (grenades and knife) also have their own use functions. The challenge here was in ensuring all systems are updated when a grenade is used. Thus, the animation BP and inventory update based on grenade use. If no items are left in the inventory, the animation goes to the default one, if an item is present, it swaps to it.



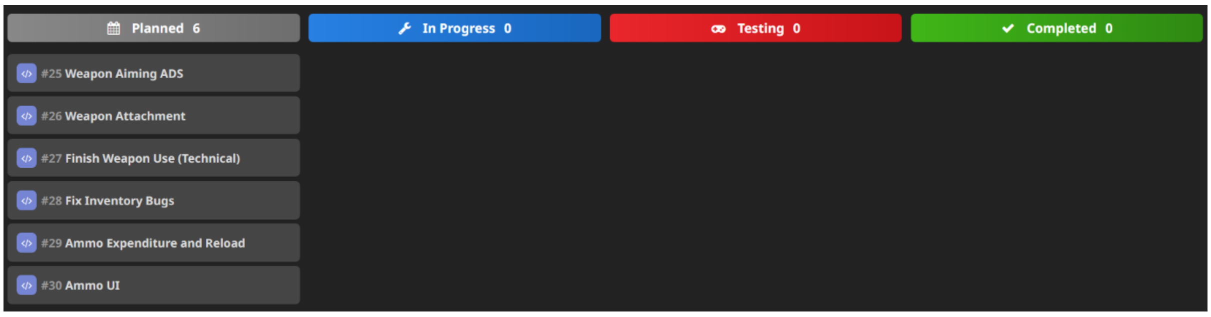
Kanban – end of the week:



Planned 0	In Progress 0	Testing 0	Completed 5
			~~#20 Finish Inventory System Architecture~~
			~~#21 Make grenades and knife usable~~
			~~#22 Inventory system fine tuning~~
			~~#23 Weapon attachment~~
			~~#24 Weapon use~~

Week 4 (12/03/2025-19/03/2025)

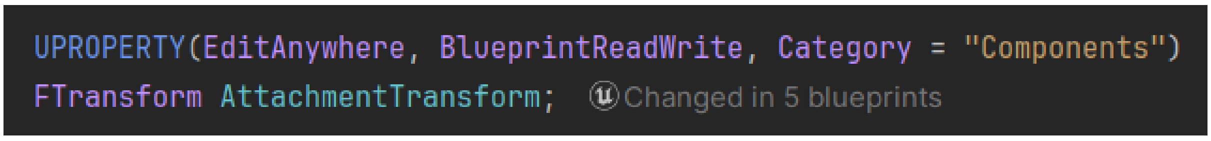
Kanban – start of the week:



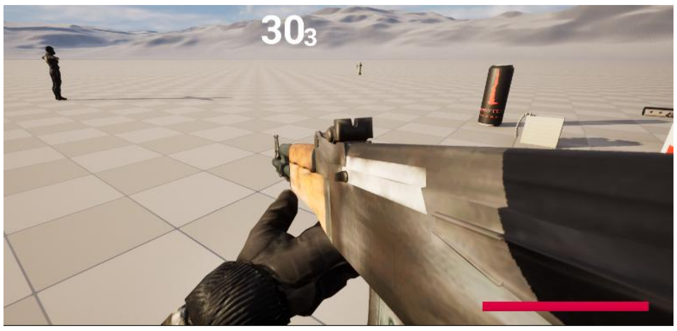
Planned 6	In Progress 0	Testing 0	Completed 0
#25 Weapon Aiming ADS			
#26 Weapon Attachment			
#27 Finish Weapon Use (Technical)			
#28 Fix Inventory Bugs			
#29 Ammo Expenditure and Reload			
#30 Ammo UI			

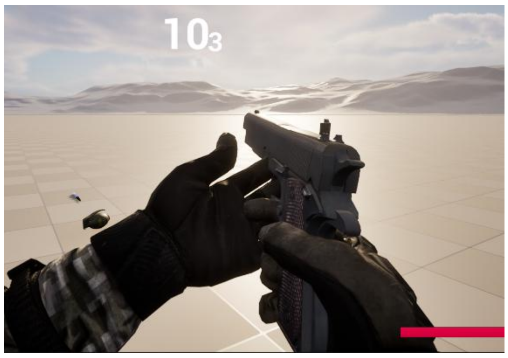
Weapon Attachment and Use Finish:

I managed to finally find a scalable and good solution to the problem of weapon attachment. The way I've gone about it was creating an attachment transform for each inventory item, then tweak it individually in engine for each one in order to make it look natural when attached to the hand. It may still require some fine tuning in the polishing stage, but I am content with this for the time being.



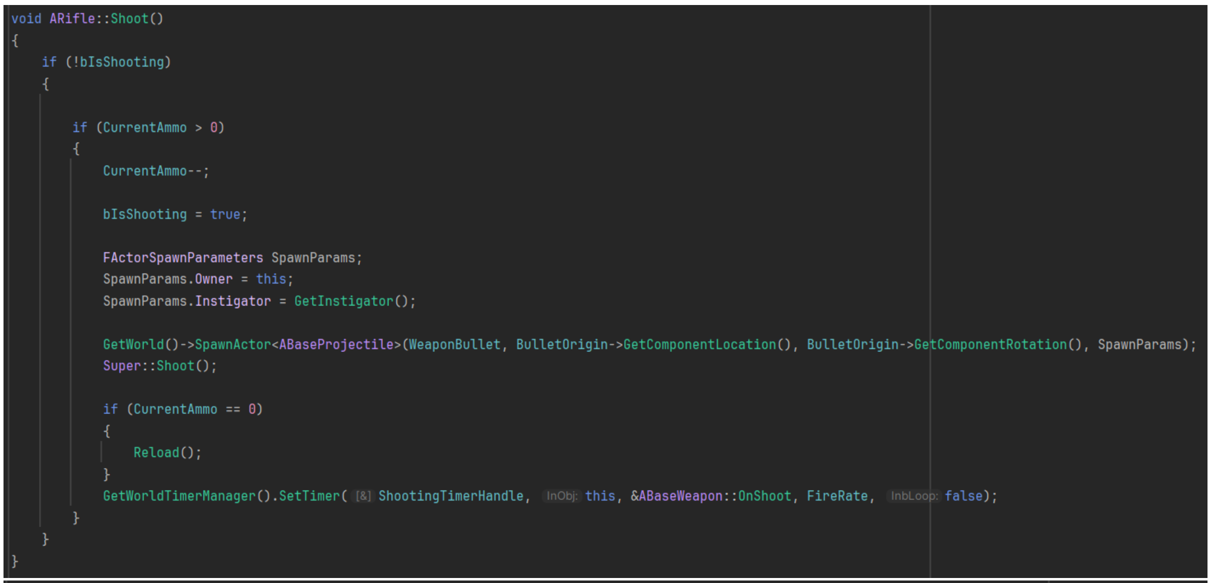
```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Components")
FTransform AttachmentTransform;   Changed in 5 blueprints
```



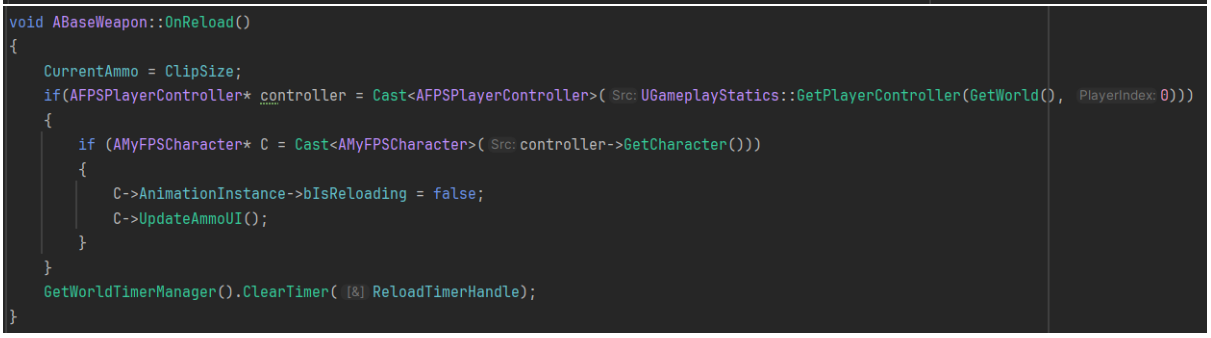


Weapon Functions (Ammo Expenditure, Reload, Aiming):

The next challenge was incorporating the common weapon functions into the weapon classes, as prior to this, the weapons had no way to track bullet counts or reload. This proved to be relatively easy, with almost no added difficulties. The only real challenge was tying this up to the weapon UI, which I will elaborate on in the next section.



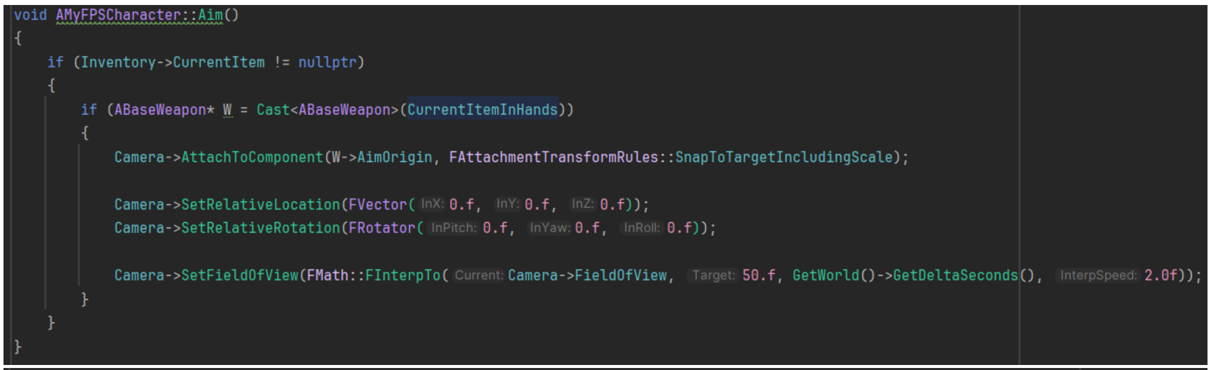
```
void ARifle::Shoot()
{
    if (!bIsShooting)
    {

        if (CurrentAmmo > 0)
        {
            CurrentAmmo--;

            bIsShooting = true;

            FActorSpawnParameters SpawnParams;
            SpawnParams.Owner = this;
            SpawnParams.Instigator = GetInstigator();

            GetWorld()->SpawnActor<ABaseProjectile>(WeaponBullet, BulletOrigin->GetComponentLocation(), BulletOrigin->GetComponentRotation(), SpawnParams);
            Super::Shoot();

            if (CurrentAmmo == 0)
            {
                Reload();
            }
            GetWorldTimerManager().SetTimer( [&] ShootingTimerHandle,  InObj: this, &ABaseWeapon::OnShoot, FireRate,  InbLoop: false);
        }
    }
}
```



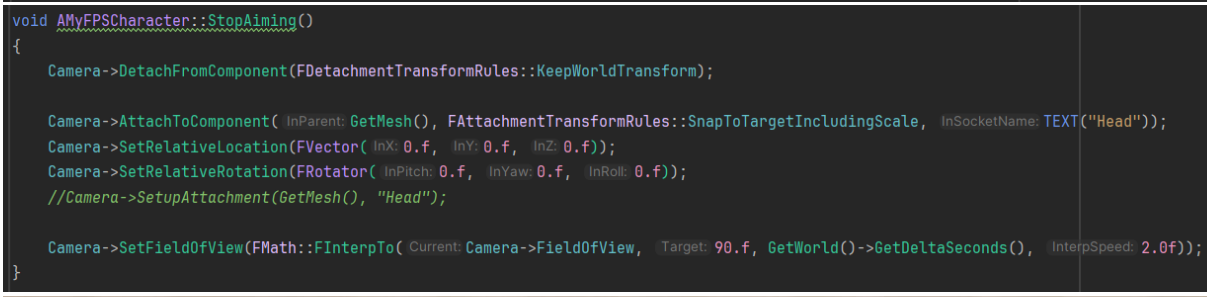
```
void ABaseWeapon::OnReload()
{
    CurrentAmmo = ClipSize;
    if(AFPSPlayerController* controller = Cast<AFPSPlayerController>( Src: UGameplayStatics::GetPlayerController(GetWorld(),  PlayerIndex: 0)))
    {
        if (AMyFPSCharacter* C = Cast<AMyFPSCharacter>( Src: controller->GetCharacter()))
        {
            C->AnimationInstance->bIsReloading = false;
            C->UpdateAmmoUI();
        }
    }
    GetWorldTimerManager().ClearTimer( [&] ReloadTimerHandle);
}
```

As you can see, the base weapon function handles general weapon functionalities such as reloading, and each weapon class overrides or adds functionality to specific functions such as shooting. Each weapon type overrides the shooting function because it requires its own implementation. Now each weapon will keep track of its ammo count and act accordingly, for instance when the ammo goes to zero, it will reload. When no ammo and no reserve are left, it will play an error sound. You can see a video of this in the next section.

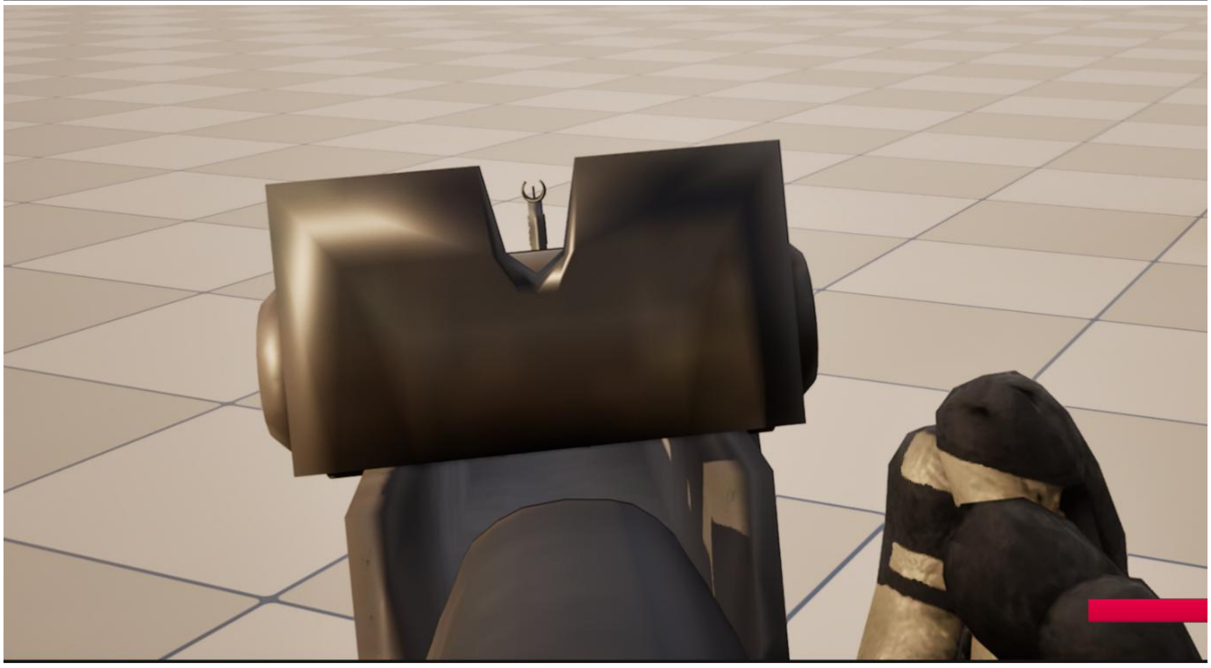
Another big challenge was the aiming down sights for the rifle, pistol and shotgun. My approach was similar to the hand attachment, where I defined a transform for every gun, then attached the player camera to it. The effects are reversed when the player stops aiming, and the camera is re-attached to the head.



```
void AMyFPSCharacter::Aim()
{
    if (Inventory->CurrentItem != nullptr)
    {
        if (ABaseWeapon* W = Cast<ABaseWeapon>(CurrentItemInHands))
        {
            Camera->AttachToComponent(W->AimOrigin, FAttachmentTransformRules::SnapToTargetIncludingScale);

            Camera->SetRelativeLocation(FVector( InX: 0.f,  InY: 0.f,  InZ: 0.f));
            Camera->SetRelativeRotation(FRotator( InPitch: 0.f,  InYaw: 0.f,  InRoll: 0.f));

            Camera->SetFieldOfView(FMath::FInterpTo( Current: Camera->FieldOfView,  Target: 50.f, GetWorld()->GetDeltaSeconds(),  InterpSpeed: 2.0f));
        }
    }
}
```

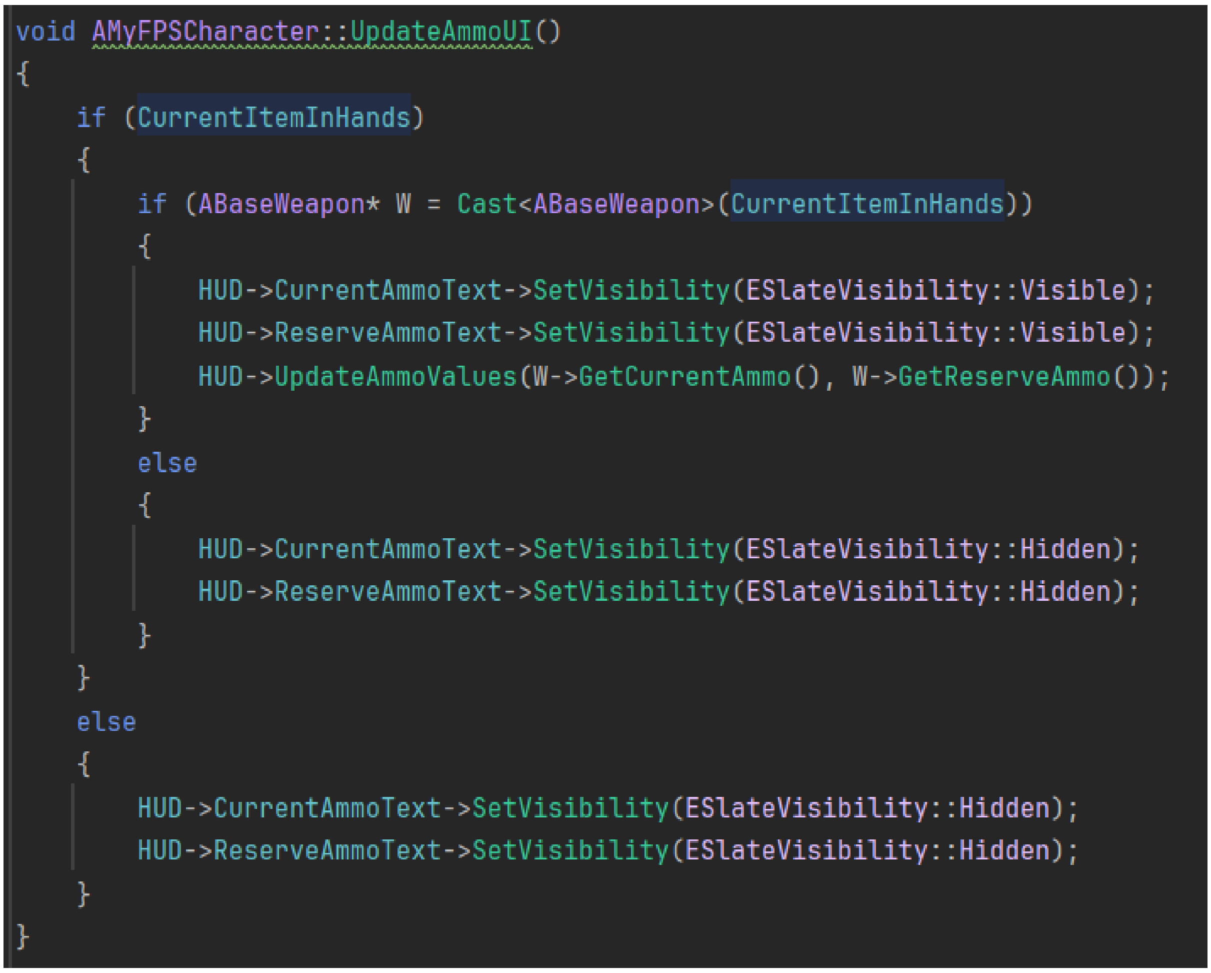


```
void AMyFPSCharacter::StopAiming()
{
    Camera->DetachFromComponent(FDetachmentTransformRules::KeepWorldTransform);

    Camera->AttachToComponent( InParent: GetMesh(), FAttachmentTransformRules::SnapToTargetIncludingScale,  InSocketName: TEXT("Head"));
    Camera->SetRelativeLocation(FVector( InX: 0.f,  InY: 0.f,  InZ: 0.f));
    Camera->SetRelativeRotation(FRotator( InPitch: 0.f,  InYaw: 0.f,  InRoll: 0.f));
    //Camera->SetupAttachment(GetMesh(), "Head");

    Camera->SetFieldOfView(FMath::FInterpTo( Current: Camera->FieldOfView,  Target: 90.f, GetWorld()->GetDeltaSeconds(),  InterpSpeed: 2.0f));
}
```

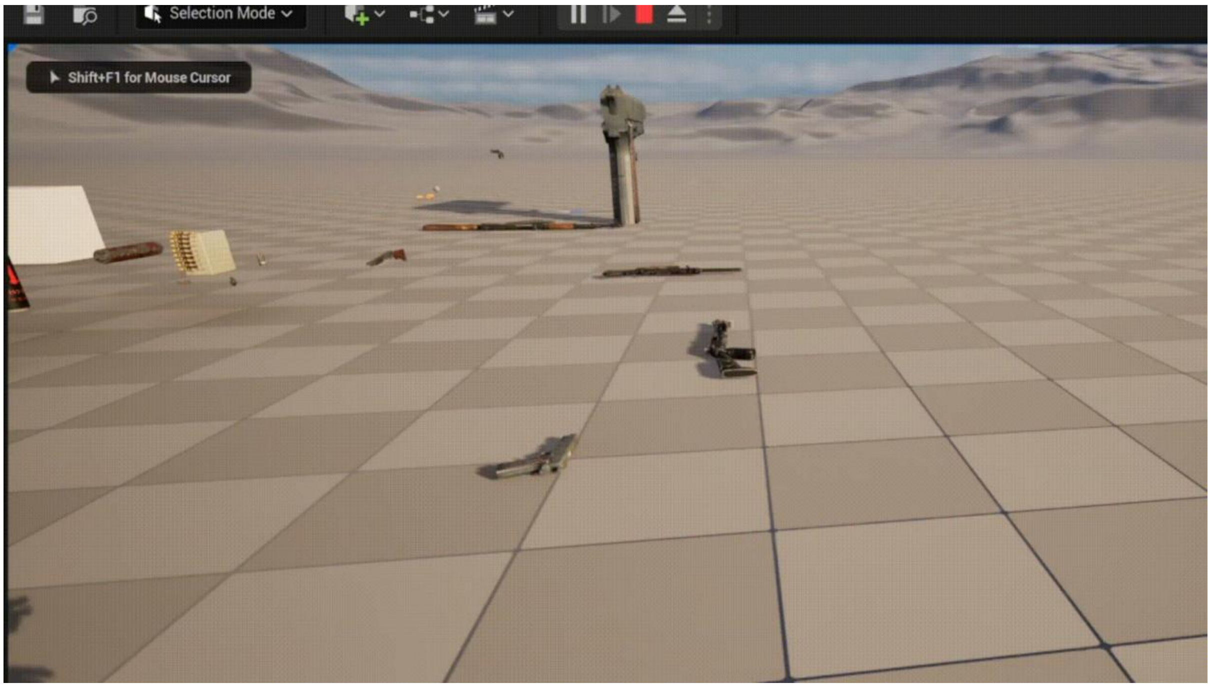


Weapon UI:

This was not a particularly challenging task, with the main difficulty being doing it all through code and keeping track of which weapon was in the hands. I tied the current and reserve ammo values to the UI and switched them based on the current weapon. A function to update the UI was created that is called on shoot or reload. When no weapons are held, the ammo count disappears. There is more functionalities to be worked on in the upcoming weeks, such as the inventory, for instance, which is why this task will remain in progress for the upcoming weeks.



```cpp
void AMyFPSCharacter::UpdateAmmoUI()
{
    if (CurrentItemInHands)
    {
        if (ABaseWeapon* W = Cast<ABaseWeapon>(CurrentItemInHands))
        {
            HUD->CurrentAmmoText->SetVisibility(ESlateVisibility::Visible);
            HUD->ReserveAmmoText->SetVisibility(ESlateVisibility::Visible);
            HUD->UpdateAmmoValues(W->GetCurrentAmmo(), W->GetReserveAmmo());
        }
        else
        {
            HUD->CurrentAmmoText->SetVisibility(ESlateVisibility::Hidden);
            HUD->ReserveAmmoText->SetVisibility(ESlateVisibility::Hidden);
        }
    }
    else
    {
        HUD->CurrentAmmoText->SetVisibility(ESlateVisibility::Hidden);
        HUD->ReserveAmmoText->SetVisibility(ESlateVisibility::Hidden);
    }
}
```

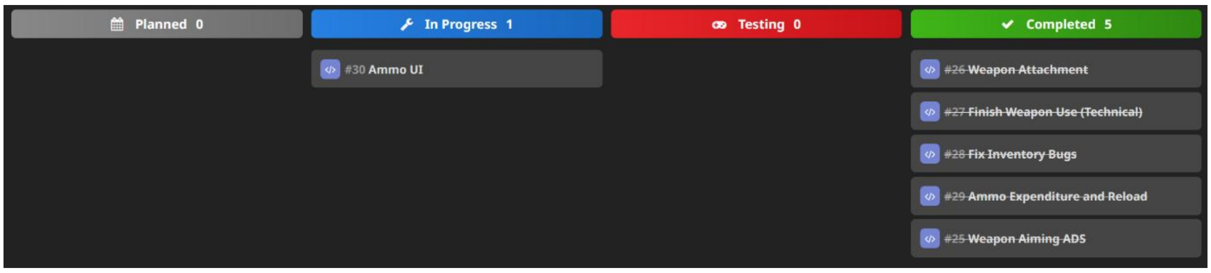


Inventory Bug Fixing:

Slight modifications to the inventory system and its functionality came with certain new bugs. This week I worked on eradicating the old bugs that had been there from the beginning, as well as the new ones that came as a consequence of the rework. Unfortunately, it is not something as tangible that can be showcased, but it did pose a great challenge this week. These changes will be clear to the user when playing through the smooth use and functionality of the system.

Kanban – end of the week:

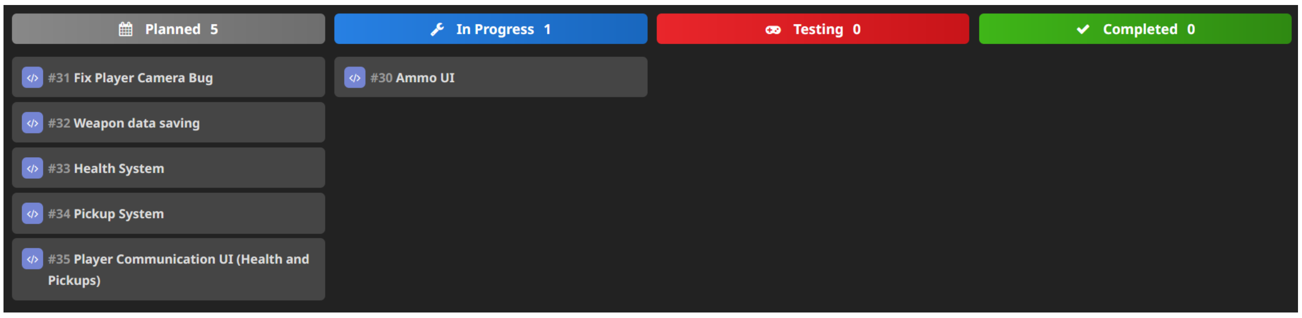
I managed to complete all the set tasks, with the ammo UI being left as work in progress, as it overlaps with other systems and requires more work to be done in the upcoming weeks.



Planned 0	In Progress 1	Testing 0	Completed 5
	#30 Ammo UI		~~#26 Weapon Attachment~~
			~~#27 Finish Weapon Use (Technical)~~
			~~#28 Fix Inventory Bugs~~
			~~#29 Ammo Expenditure and Reload~~
			~~#25 Weapon Aiming ADS~~

Week 5 (19/03/2025-26/03/2025)

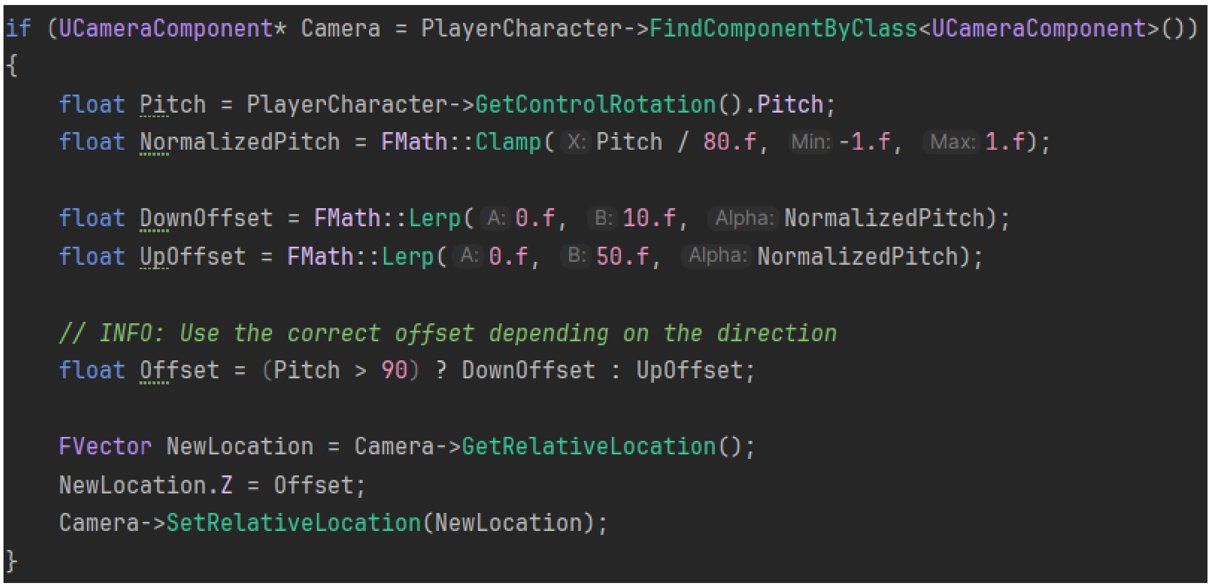
Kanban – start of the week:



Planned 5	In Progress 1	Testing 0	Completed 0
#31 Fix Player Camera Bug	#30 Ammo UI		
#32 Weapon data saving			
#33 Health System			
#34 Pickup System			
#35 Player Communication UI (Health and Pickups)			

Bug Fixing (Player Camera):

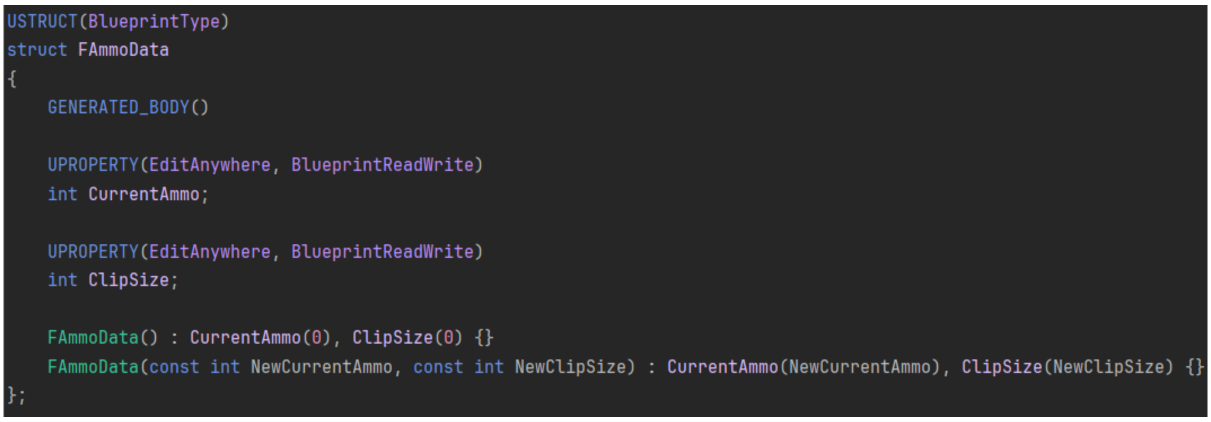
I started off by doing some bug fixing for the overall player experience. One annoying bug was the player being able to look inside its own torso/head when looking up/down. I fixed this issue by putting the camera on a 'swivel' through code, where looking up or down would modify the camera position in the socket by a given offset, which would make it look seamless.



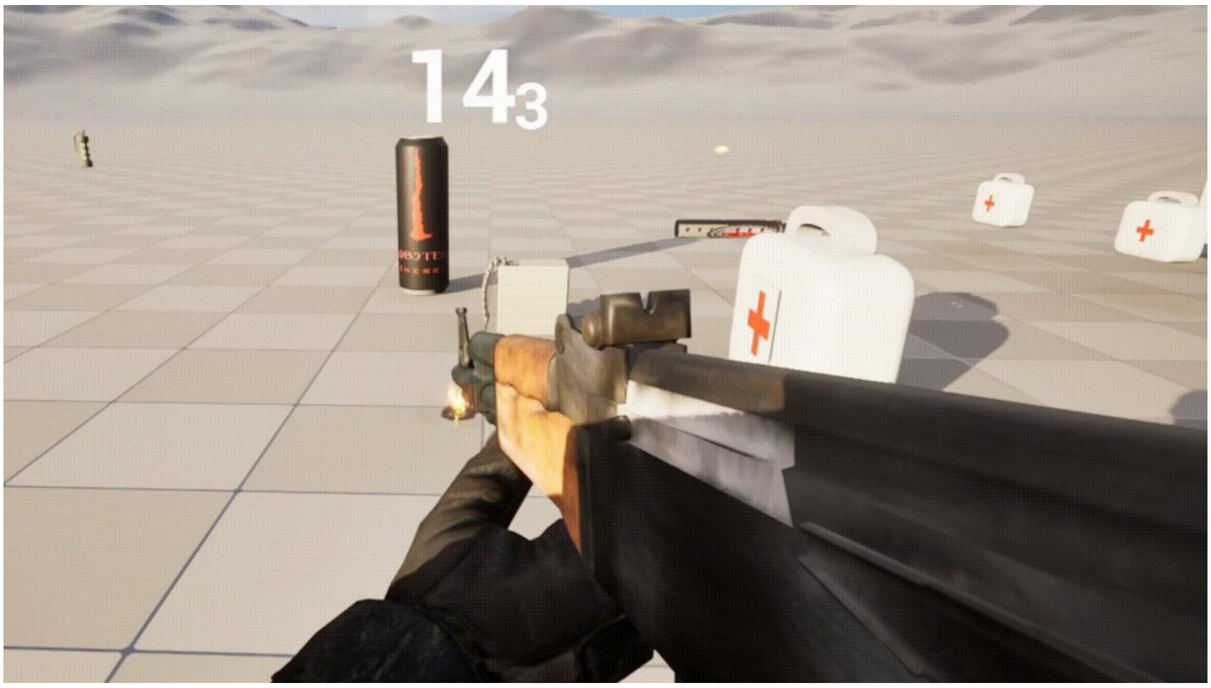
```
if (UCameraComponent* Camera = PlayerCharacter->FindComponentByClass<UCameraComponent>())
{
    float Pitch = PlayerCharacter->GetControlRotation().Pitch;
    float NormalizedPitch = FMath::Clamp( X: Pitch / 80.f,  Min: -1.f,  Max: 1.f);

    float DownOffset = FMath::Lerp( A: 0.f,  B: 10.f,  Alpha: NormalizedPitch);
    float UpOffset = FMath::Lerp( A: 0.f,  B: 50.f,  Alpha: NormalizedPitch);

    // INFO: Use the correct offset depending on the direction
    float Offset = (Pitch > 90) ? DownOffset : UpOffset;

    FVector NewLocation = Camera->GetRelativeLocation();
    NewLocation.Z = Offset;
    Camera->SetRelativeLocation(NewLocation);
}
```

Weapon Data Saving:

A relatively simple task was the data saving for the weapons. It posed a problem because the way the inventory handles throwing or adding items to inventory is by destroying one instance and creating another. I circumvented that issue by creating a struct in the player character class that would keep track of the weapons and their ammo. If the weapon is thrown, the current ammo data is saved, and applied on the next pickup of the weapon.

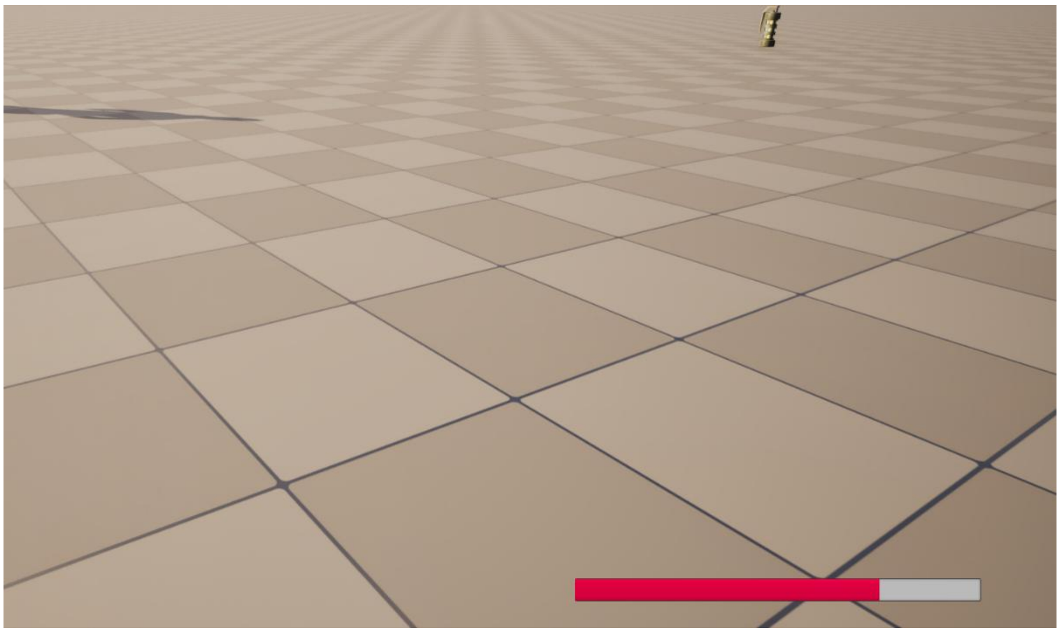


```
USTRUCT(BlueprintType)
struct FAmmoData
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int CurrentAmmo;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int ClipSize;

    FAmmoData() : CurrentAmmo(0), ClipSize(0) {}
    FAmmoData(const int NewCurrentAmmo, const int NewClipSize) : CurrentAmmo(NewCurrentAmmo), ClipSize(NewClipSize) {}
};
```



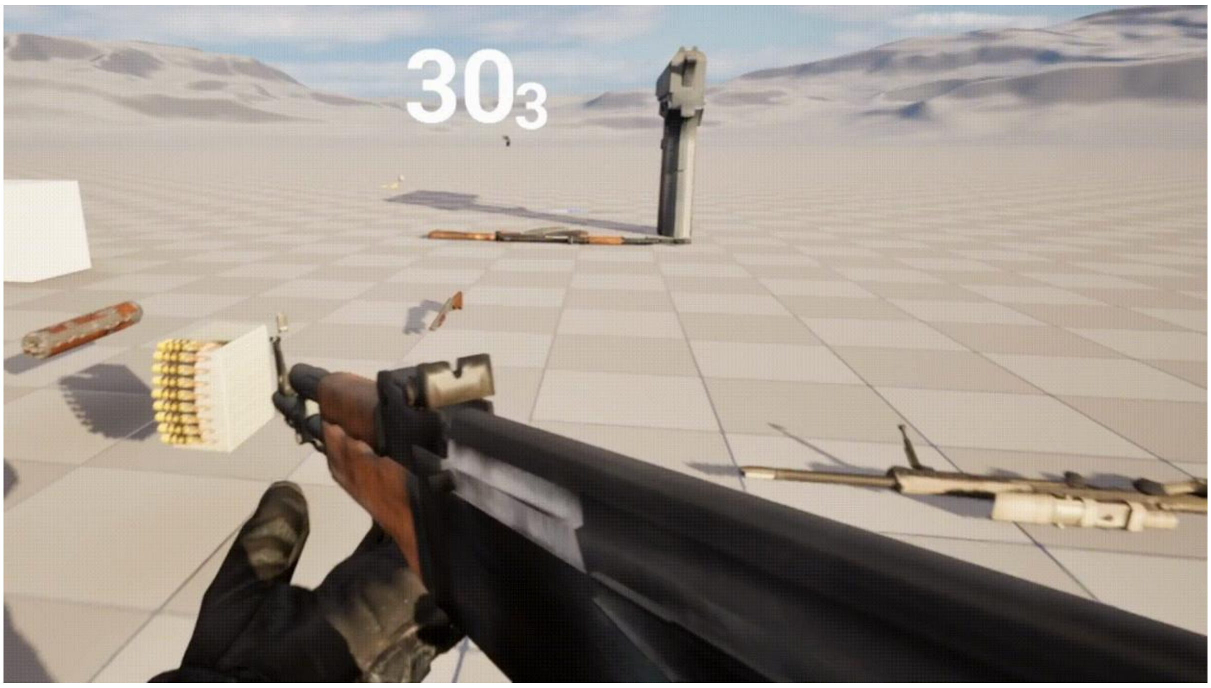
Health System:

Again, a relatively straight-forward and easy task. I defined current and max health variables and added code to subtract from current health if the player is hit with any type of projectile, each of them carrying a different amount of damage. I also added a healing function that I will elaborate on in the next section where I talk about the pickup system. I then created some simple UI for a health bar along with an UI update function and tied the health variables to it. One challenge was communicating the damage to the player through UI. I added an overlay of blood splatter on the edges of the screen that plays a quick animation on hit. You will be able to see the video of it below, in the Player Communication UI section.



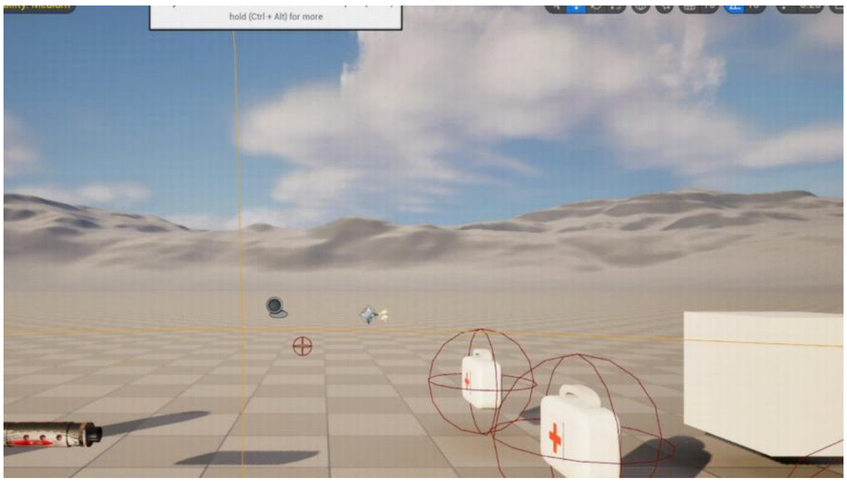
Pickup System:

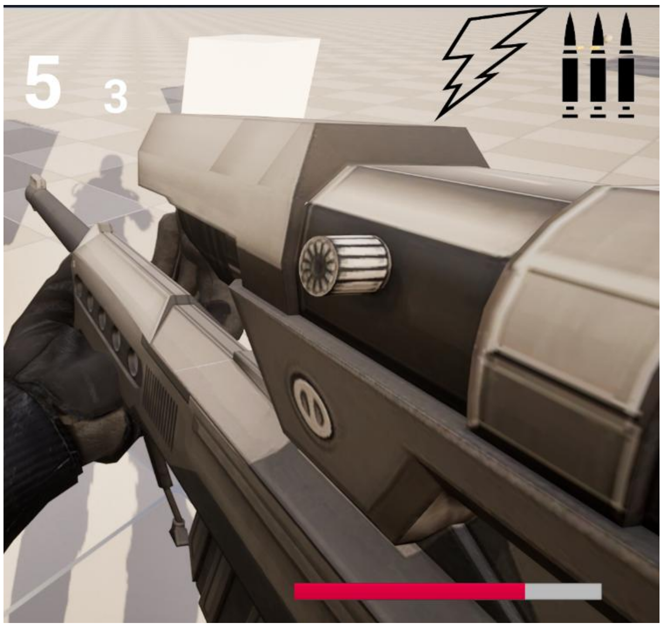
I completed this task following the GDD, but ended up tweaking some of the values, such as the player speed buff. It turns out that my initial value had a barely noticeable effect, so I changed it to double the original amount to make it more useful for the player. The other pickups are straightforward, one being a healing kit, an ammo damage buff and an ammo reserve mag. Again, the challenge stood in communicating the effects to the player in a clear and diegetic manner, please find more in the next section.



Player Communication UI (Damage and Pickups):

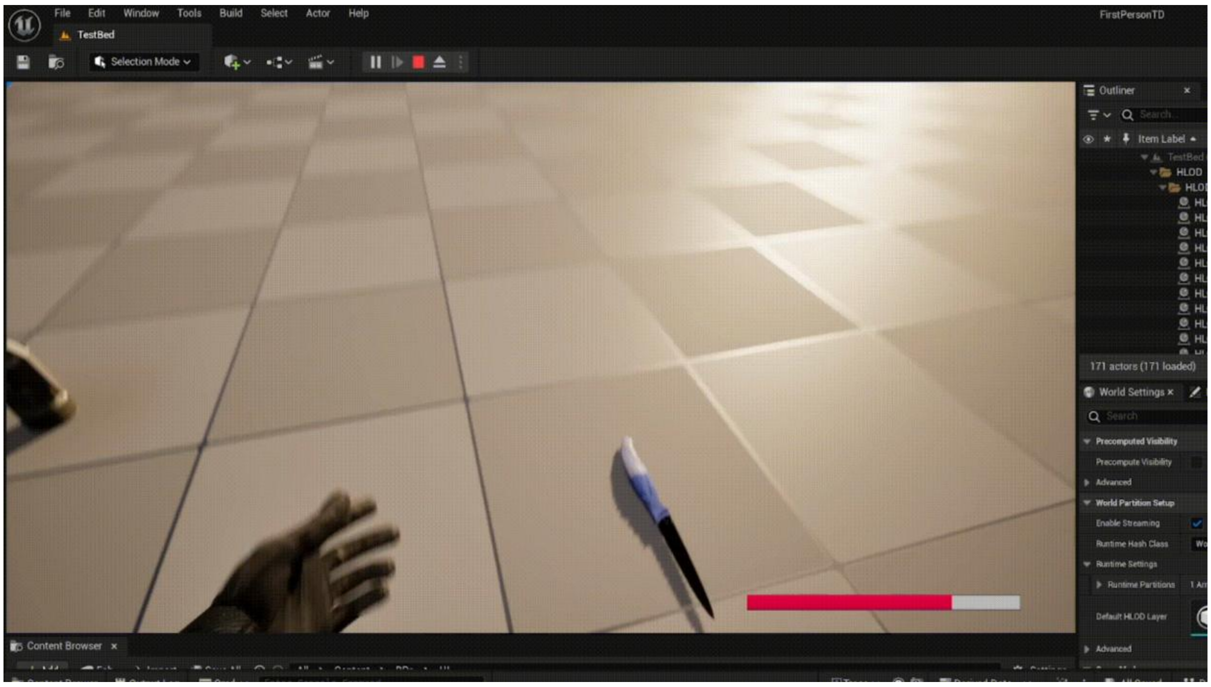
I believe this was one of the most important additions of this week in terms of juice and overall game feel, since it greatly impacts the player experience. As discussed in the two previous sections, I needed to find a way to communicate the effects taking place on the player through UI. For healing, I came up with a simple green overlay to signify gaining life, and for the speed and damage buffs I resorted to icons in the top right corner of the screen. Please find these below.





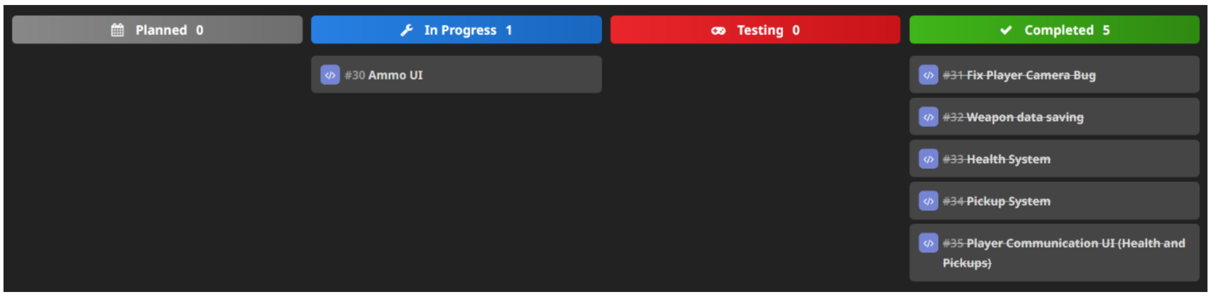
Ammo UI (Continued):

The continuation of this section this week is an UI addition of a progress bar when throwing the knife. It takes a while for the player to throw it, and it requires holding down the mouse button. In order to communicate this to the player, I have created a progress bar that charges up as you hold down the LMB. Please find this implementation below.



Kanban – end of the week:

As you can clearly see, I completed all the set tasks, with the Ammo UI staying in progress yet again, since it requires more work, like getting the inventory up on the screen too.



Planned 0	In Progress 1	Testing 0	Completed 5
	#30 Ammo UI		~~#31 Fix Player Camera Bug~~
			~~#32 Weapon data saving~~
			~~#33 Health System~~
			~~#34 Pickup System~~
			~~#35 Player Communication UI (Health and Pickups)~~

Week 5 (26/03/2025-02/04/2025)