

3D 싱글 RPG 게임용 프레임워크 개발

작성자 : 서승호

1. 개요

언리얼엔진 5를 기반으로 제작된 3D 싱글 RPG 게임 개발 프레임워크를 제작했습니다. RPG 게임에 필수적인 세이브/로드, 인벤토리, 퀘스트 등의 기능을 구현했습니다.

2. Save, Load 기능 (Core/PlayerSaveObject.h, Core/CustomGameInstance.h)

언리얼 엔진의 **SaveGame** 모듈을 통해 세이브, 로드 기능을 구현했습니다.

- ✓ **UPlayerSaveObject** : 세이브 파일에 저장되는 플레이어 정보
- ✓ **PlayerInfo** : 파일 I/O 외에 플레이어 정보를 전달해야 할 때 사용하는 구조체 (EX : 맵 전환 시)

```
struct PlayerInfo
{
    FString PlayerName;           // 플레이어 이름
    int CurrentLevel;            // 현재 레벨
    int CurrentMap;              // 현재 위치한 맵 인덱스
    FVector CurrentLocation;     // 현재 맵 상의 위치
    int CurrentGold;             // 현재 골드 보유량
    FInventory Inventory;        // 현재 인벤토리
    TArray<FQuestStatus> QuestTable; // 현재 퀘스트 진행도

    int SlotIndex;               // 세이브 슬롯 번호
};
```

```
UCLASS()
class PORTFOLIO_API UPlayerSaveObject : public USaveGame
{
    GENERATED_BODY()

public:
    UPlayerSaveObject();

public:
    UPROPERTY(BlueprintReadWrite) FString mPlayerName;
    UPROPERTY(BlueprintReadWrite) int mCurrentLevel;
    UPROPERTY(BlueprintReadWrite) int mCurrentMap;
    UPROPERTY(BlueprintReadWrite)
```

해당 구조체를 통해 **UCustomGameInstance**에서 기능을 구현했습니다.

- ✓ **CreateSaveFile()** : 세이브 슬롯의 빈 자리를 찾아 세이브 파일을 생성한 후, 인덱스를 반환
- ✓ **SaveFile()** : 현재 게임 상태를 게임 시작 시 불러왔던 세이브 파일에 저장
- ✓ **LoadFile()** : 해당 인덱스의 세이브 파일을 불러와 게임을 시작

```
void UCustomGameInstance::SaveGame(TObjectPtr<APlayerCharacter> player)
{
    check(player);

    USaveSlot* saveSlot = Cast<USaveSlot>(UGameplayStatics::GetSaveSlot("SaveGame", 0));
    check(saveSlot);

    const int slotIndex = player->GetSlotIndex();
    check(slotIndex < saveSlot->mSlotNameList.Num());

    const FString playerName = saveSlot->mSlotNameList[slotIndex];
    UPlayerSaveObject* so = Cast<UPlayerSaveObject>(UGameplayStatics::CreateSaveGameObject(UPlayerSaveObject::StaticClass()));

    so->mPlayerName = player->mDisplayName;
    so->mCurrentLevel = player->GetCurrentLevel();
    so->mCurrentGold = player->GetCurrentGold();
    so->mCurrentMap = player->GetCurrentMapIndex();
    so->mCurrentLocation = player->GetActorLocation();
    so->mInventory = player->GetInventory_cpy();
    so->mQuestTable = player->GetQuestTable_cpy();
    so->mSlotIndex = player->GetSlotIndex();

    UGameplayStatics::SaveGameToSlot(so, player->mDisplayName, slotIndex);
}
```

```
void UCustomGameInstance::LoadGame(int slotIndex)
{
    USaveSlot* saveSlot = Cast<USaveSlot>(UGameplayStatics::GetSaveSlot("SaveGame", slotIndex));
    check(saveSlot);

    const FString playerName = saveSlot->mSlotNameList[slotIndex];
    UPlayerSaveObject* ls = Cast<UPlayerSaveObject>(UGameplayStatics::CreateSaveGameObject(UPlayerSaveObject::StaticClass()));
    ls = Cast<UPlayerSaveObject>(UGameplayStatics::LoadGameFromSlot(ls, playerName));

    check(ls);

    mTempPlayerInfo.PlayerName = ls->mPlayerName;
    mTempPlayerInfo.CurrentLevel = ls->mCurrentLevel;
    mTempPlayerInfo.CurrentMap = ls->mCurrentMap;
    mTempPlayerInfo.CurrentLocation = ls->mCurrentLocation;
    mTempPlayerInfo.CurrentGold = ls->mCurrentGold;
    mTempPlayerInfo.QuestTable = ls->mQuestTable;
    mTempPlayerInfo.Inventory = ls->mInventory;
    mTempPlayerInfo.SlotIndex = ls->mSlotIndex;

    TObjectPtr<ACustomController> controller = Cast<ACustomController>(GetWorld()->GetFirstPlayerController());
    check(controller);
    controller->CloseIntro();

    bIntro = false;

    UGameplayStatics::OpenLevel(this, FName(mLevelList[slotIndex]));
}
```

3. 게임 데이터 (Content/Data/)

아이템, 퀘스트, Npc 대화, 몹 정보 등의 게임 데이터를 Contents/Data 폴더 내의 각 json 파일에 작성해 게임 시작 시에 일괄적으로 불러옵니다. 목록은 다음과 같습니다.

- ✓ **ItemInfo.json** : 아이템 목록
- ✓ **Quest.json** : 퀘스트 목록
- ✓ **LevelInfo.json** : 맵 목록
- ✓ **EnemyInfo.json** : 적 캐릭터 목록
- ✓ **EnemyLabelInfo.json** : 적들이 가질 수 있는 특성 목록 (ex : 인간족, 몬스터, ...)
- ✓ **LevelData/맵 이름/** : 각 맵의 Npc, 적 정보를 담은 폴더
 - ✓ **Npc.json** : 맵의 Npc 정보
 - ✓ **Enemy.json** : 맵에 스폰할 적 정보

※ 단, LevelData 내의 맵 데이터는 게임 시작이 아닌 각 맵 로드시에
호출되는 OnMapLoaded() 함수에서 로드됩니다.

```
void UCustomGameInstance::Init()
{
    if (bLoaded == false)
    {
        // 맵 정보 로딩
        FCoreUObjectDelegates::PostLoadMapWithWorld();
        LoadGameData(); // 게임 데이터 로드
    }
}
```

```
void UCustomGameInstance::LoadGameData()
{
    // 각 게임 데이터 로딩
    ItemInfo::LoadInfoListFromJson(mItemInfoList);
    FQuest::LoadInfoListFromJson(mQuestInfoList);
    FEnemyLabelInfo::LoadInfoListFromJson(mEnemyLabelInfoList);
    FEnemyInfo::LoadInfoListFromJson(mEnemyInfoList);
}
```

```
void UCustomGameInstance::OnMapLoaded()
{
    const FString& currentLevelName =
    if (currentLevelName == "Test_Instance")
    {
        return;
    }

    LoadNpcList(world);
    LoadEnemyList(world);
}
```

1) 아이템 (Common/Item.h)

| | | |
|---|---|---|
| <code>struct ItemInfo</code> | <code>struct FGameItem</code> | <code>struct FInventory</code> |
| { int Index; // 게임 데이터 상의 인덱스 EItemType Type; // 아이템 타입 FString Name; // 아이템 이름 bool bSellable; // 상점에 팔 수 있는 아이템인가? int Price; // 상점 판매 가격 TArray<UTexture2D*> Thumbnail; // 썸네일 이미지 FString ItemMesh; // 게임에 표시되는 아이템 메시 | { GENERATED_USTRUCT_BODY() // ItemInfo의 인덱스 UPROPERTY(EditAnywhere, BlueprintReadWrite) int InfoIndex; // 보유 개수 UPROPERTY(EditAnywhere, BlueprintReadWrite) int Num; | { GENERATED_USTRUCT_BODY() FTypeInventory& GetTypeInventory(EItemType); const FTypeInventory& GetTypeInventory(EItemType); UPROPERTY(BlueprintReadWrite) TArray<FTypeInventory> TypeInventoryList; |

- ✓ **ItemInfo** : 각 아이템 정보 구조체. 타입, 이름, 가격 등의 정보를 담고 있습니다.
- ✓ **FGameItem** : 게임 내에서 주고 받을 수 있는 아이템 형태. Item Info 리스트 상의 인덱스, 개수를 가지고 있습니다.
- ✓ **FInventory** : 타입 별로 보관되는 FGameItem 배열. 플레이어의 인벤토리/Npc의 상점 리스트에 사용됩니다.

2) 퀘스트 (Common/Quest.h)

- ✓ **FQuest** : 퀘스트 정보 구조체. 타입, 서브퀘스트 목록, 보상 등의 정보를 담음.
타입 : 순차 퀘스트/ 병렬 퀘스트)
- ✓ **FSubQuest** : 서브 퀘스트 정보. 타입, 설명, 타입별 부가정보 등을 담음.
 - **Arrival** : 특정 위치에 도달하는 퀘스트
 - **Hunt** : 특정 적 캐릭터를 처치하는 퀘스트
 - **Item** : 특정 아이템을 모아야 하는 퀘스트
 - **Action** : 그 외의 행동으로 완료되는 퀘스트
- ✓ **FQuestStatus** : 플레이어의 각 퀘스트 진행 상황을 저장하는 구조체

| | |
|--|---|
| <code>struct FSubQuest</code> | <code>USTRUCT(Atomic, BlueprintType)</code> |
| { GENERATED_USTRUCT_BODY() ESubQuestType Type; FString Explanation; int MainQuestIndex; union { struct ArrivalForm { int MapIndex; FVector Destination; UParticleSystem* FX; } ArrivalInfo; struct CollectForm { int Index; int Num; } CollectInfo; | <code>struct FQuest</code> |
| | { GENERATED_USTRUCT_BODY() FString Name; FString Explanation; EQuestType Type; TArray<FSubQuest> SubQuests; TArray<FQuestReward> Rewards; |
| | <code>struct FQuestStatus</code> |
| | { GENERATED_USTRUCT_BODY() UPROPERTY(BlueprintReadWrite) int Index; UPROPERTY(BlueprintReadWrite) EQuestType Type; UPROPERTY(BlueprintReadWrite) EQuestProgressType CurrProgress; UPROPERTY(BlueprintReadWrite) TArray<FSubQuestStatus> SubStatus; |

3) 대화 (Common/Dialogue.h)

- ✓ **FDialogueLine** : 주고 받는 대화의 한 단위입니다. 대사와 응답 선택지를 포함합니다.
- ✓ **FDialogueResponse** : 선택할 수 있는 응답입니다. 선택 시 발생하는 이벤트를 포함합니다.
- ✓ **FDialogueEvent** : 응답 선택시 발생하는 이벤트. 타입과 타입별 변수를 포함합니다.
 - **End** : 대화 종료
 - **Jump** : 특정 인덱스로 대화 이동
 - **CommitQuest** : 퀘스트를 시작
 - **CompleteQuest** : 퀘스트를 완료시킴
 - **OpenShop** : 상점을 엽니다
 - **GiveItem** : 아이템을 획득
 - **SetBookmark** : 다음에 대화할 때 특정 인덱스부터 대화 시작

```
struct FDialogueLine
{
    GENERATED_USTRUCT_BODY()

    FString SpeakerName; // 화자
    FString Text; // 대사 텍스트
    TArray<FDiscussionResponse> Responses;
}
```

```
struct FDiscussionResponse
{
    GENERATED_USTRUCT_BODY()

    FString Text; // 응답 텍스트
    TArray<FDiscussionEvent> Events;
}
```

```
struct FDiscussionEvent
{
    GENERATED_USTRUCT_BODY()

    EDiscussionEventType EventType;
    union // EventType에 따라 다른
    {
        int JumpIndex; // Jump
        int QuestIndex; // CommitQuest
        int ItemIndex; // GiveItem
        int BookmarkIndex; // SetBookmark
    };
}
```

4. 캐릭터 (Character/)

ACustomCharacter : 메인 캐릭터, NPC, 몹에 공통적으로 적용되는 속성/기능을 구현했습니다.

- ✓ 최대 체력, 현재 체력
- ✓ 데미지를 입었을 때 **OnHurt()** 함수 호출 (주로 경직을 거는 용도)
- ✓ 체력이 0이 되면 **OnDead()** 함수 호출

```
class PORTFOLIO_API ACustomCharacter : public ACharacter
{
    GENERATED_BODY()

public:
    ACustomCharacter();

protected:
    virtual void BeginPlay() override;

public:
    virtual void Tick(float DeltaTime) override;
    UFUNCTION()
    virtual float TakeDamage(float Damage, struct FDamag
        // 데미지를 입었을 때 호출되는 함수
    UFUNCTION()
    virtual void OnHurt() {};

    // HP가 0이 되었을 때 호출되는 함수
    UFUNCTION()
    virtual void OnDead() {};

    // 현재 체력 반환
    int GetCurrHp();
    // 최대 체력 반환
    int GetMaxHp();
}
```

5. 메인 캐릭터 (Character/PlayerCharacter.h)

- ✓ **APlayerCharacter** : 조작 가능한 캐릭터. 입력, 애니메이션, 상호작용 등의 기능이 구현되었습니다.

1) 입력

캐릭터 이동, 공격, UI창 열기 등 다양한 행동을 출력하도록 입력을 바인드했습니다.

```
void APlayerCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);

    PlayerInputComponent->BindAxis("MoveVertical", this, &APlayerCharacter::MoveVertical);
    PlayerInputComponent->BindAxis("MoveHorizontal", this, &APlayerCharacter::MoveHorizontal);
    PlayerInputComponent->BindAxis("TurnVertical", this, &APlayerCharacter::TurnVertical);
    PlayerInputComponent->BindAxis("TurnHorizontal", this, &APlayerCharacter::TurnHorizontal);

    PlayerInputComponent->BindAction("StartRun", IE_Pressed, this, &APlayerCharacter::StartRun);
    PlayerInputComponent->BindAction("StopRun", IE_Released, this, &APlayerCharacter::StopRun);
    PlayerInputComponent->BindAction("Jump", IE_Pressed, this, &APlayerCharacter::Jump);

    PlayerInputComponent->BindAction("TryInteract", IE_Pressed, this, &APlayerCharacter::TryInteract);

    PlayerInputComponent->BindAction("QuickSlot", IE_Pressed, this, &APlayerCharacter::TurnQuickSlot);
    PlayerInputComponent->BindAction("Equip", IE_Pressed, this, &APlayerCharacter::ChangeEquipment);

    PlayerInputComponent->BindAction("Attack", IE_Pressed, this, &APlayerCharacter::Attack);

    PlayerInputComponent->BindAction("OpenInventory", IE_Pressed, this, &APlayerCharacter::OpenInventory);
    PlayerInputComponent->BindAction("OpenMenu", IE_Pressed, this, &APlayerCharacter::OpenMenu);
    PlayerInputComponent->BindAction("OpenQuest", IE_Pressed, this, &APlayerCharacter::OpenQuestTable);
}
```

- ✓ **MoveVertical()**, **MoveHorizontal()** : 상하, 좌우로 캐릭터를 이동시킵니다. (WASD)
- ✓ **TurnVertical()**, **TurnHorizontal()** : 수직, 수평으로 카메라를 회전시킵니다. (마우스 수직/수평 이동)
- ✓ **StartRun()**, **StopRun()** : 캐릭터 이동 속도를 증가/감소시킵니다. (왼쪽 Shift 키 누름/떼)
- ✓ **Jump()** : 점프를 실행합니다. (스페이스 바 누름)
- ✓ **TryInteract()** : 상호작용을 시도합니다. (E키 누름)
- ✓ **QuickSlot()** : 인벤토리 상 다음 무기를 선택합니다. (T키 누름)
- ✓ **Equip()** : 선택된 무기의 장착 상태를 변경합니다. (R키 누름)
- ✓ **Attack()** : 선택, 장착된 무기로 공격합니다. (마우스 좌클릭)
- ✓ **OpenInventory()**, **OpenMenu()**, **OpenQuest()** : 인벤토리, 메뉴, 퀘스트 UI를 엽니다.

2) 애니메이션

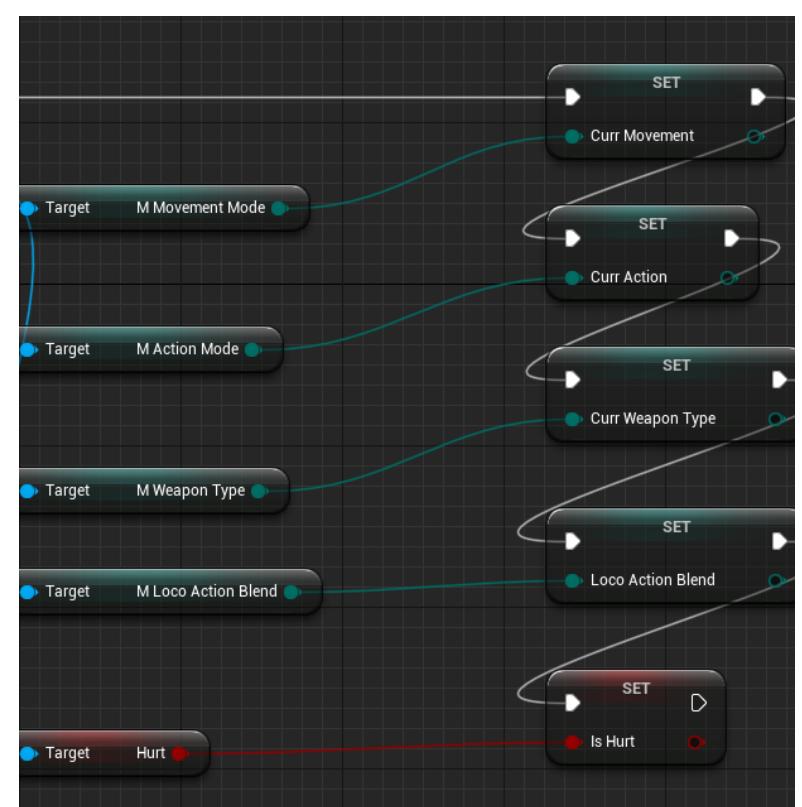
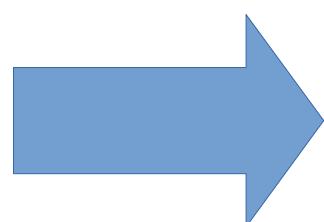
애니메이션 블루프린트를 통해 구현했습니다. 먼저, APlayerCharacter 상의 변수를 블루프린트로 전달합니다.

- ✓ **EPlayerMovementMode** : 이동, 점프, 낙하 등의 이동 상태를 나타냅니다. **UpdateMovement()** 함수로 틱마다 결정합니다.
- ✓ **EPlayerActionMode** : 공격, 상호작용 등의 행동 상태를 나타냅니다. **UpdateAction()** 함수로 틱마다 결정합니다.
- ✓ **bHurt** : 데미지를 입어 경직 중인 상태인가?
- ✓ **Weapon Type** : 현재 장착중인 무기의 타입 (현재는 Rifle 타입만 구현되었습니다)
- ✓ **Loco_Action_Bind** : Action Mode에 따라 어떤 스테이트 머신을 사용할지를 결정하는 변수
(애니메이션 에셋이 모든 상황에 맞게 구비되어있지 않아 사용하는 임시 방편입니다.)

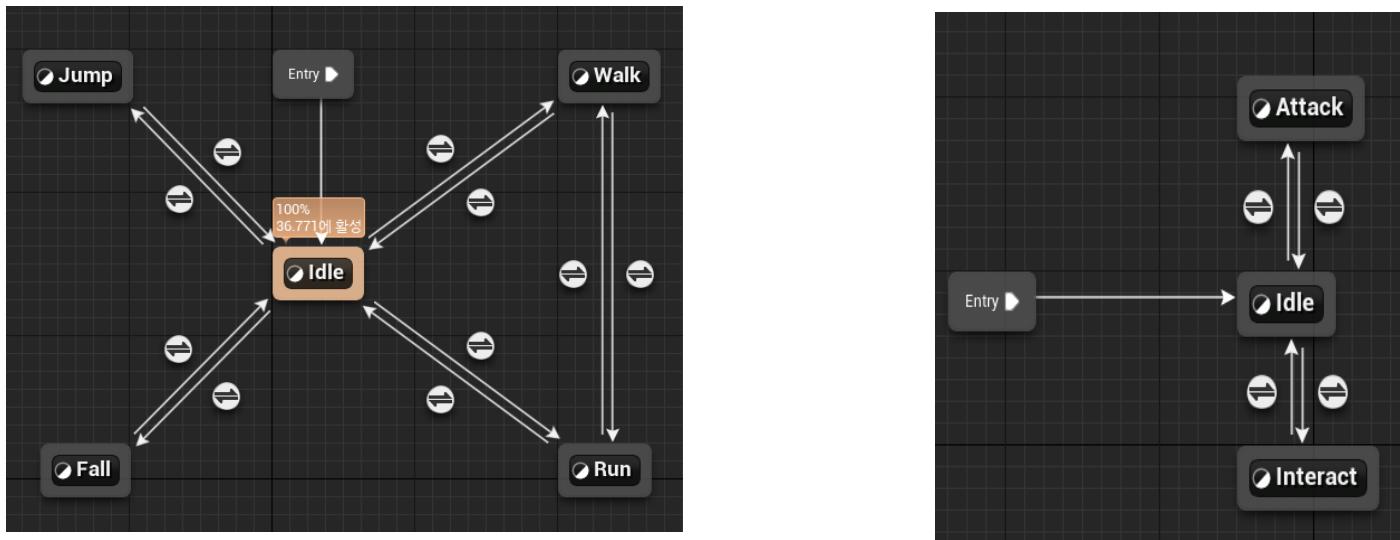
```
UENUM(BlueprintType)
enum class EPlayerActionMode : uint8
{
    Idle,
    Attack,
    Interact,
    Count,
};

UENUM(BlueprintType)
enum class EPlayerMovementMode : uint8
{
    Idle,
    Walk,
    Run,
    Jump,
    Fall,
    Count,
};

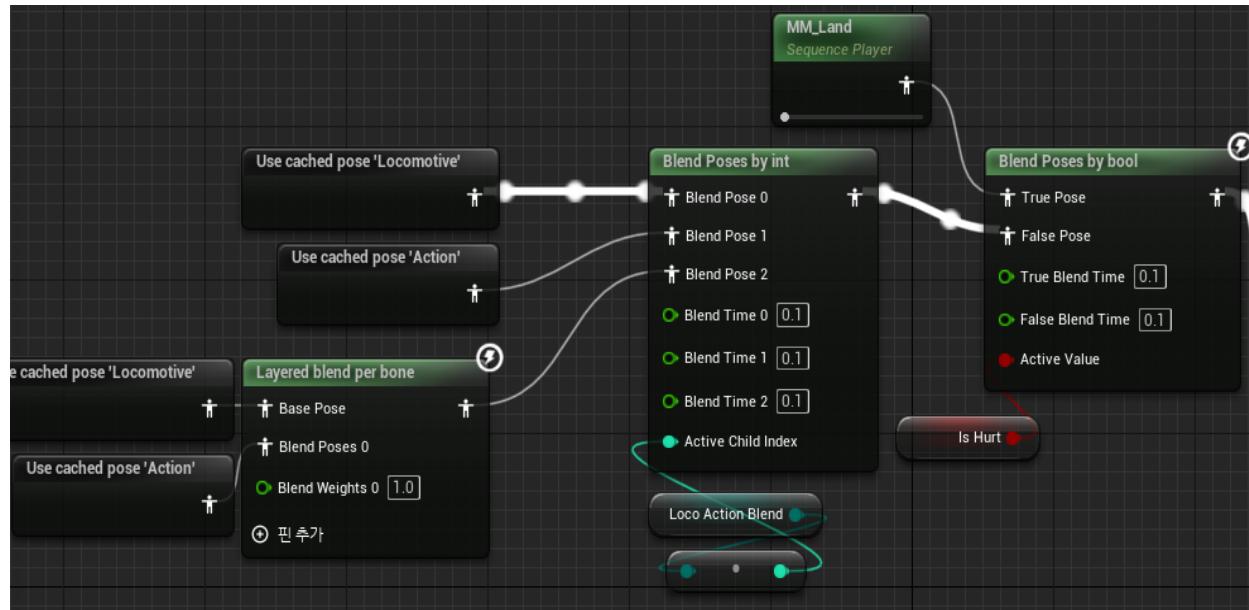
UPROPERTY(BlueprintReadWrite)
bool bHurt;
// Locomotive/Action/Blend (Animation)
UPROPERTY(BlueprintReadWrite)
uint8 mLoco_Action_Bind;
```



애니메이션 블루프린트에는 이동 모션을 담당할 **Locomotive**, 액션을 담당할 **Action** 스테이트 머신을 구현했습니다.



앞에서 변수로 받은 **Loco_Action_Blend**를 통해 어떤 스테이트 머신을 사용할지 결정하고, 최종적으로 경직 상태인지 아닌지를 판단해 경직 모션을 재생할지 말지를 결정합니다.



3) 상호작용

IInteraction : 플레이어와 상호작용 가능한 액터들이 상속받는 인터페이스.(Common/Interface)

- ✓ **Notify()** : 플레이어와 상호작용이 가능한 범위에 들어왔을 때 플레이어에 알려야 함
(ex : 위젯 등으로 게임 상에 표시)
- ✓ **Interact()** : 플레이어가 상호작용 키를 눌렀을 때 호출되는 이벤트 구현

```
void APlayerCharacter::UpdateNotifyInteraction()

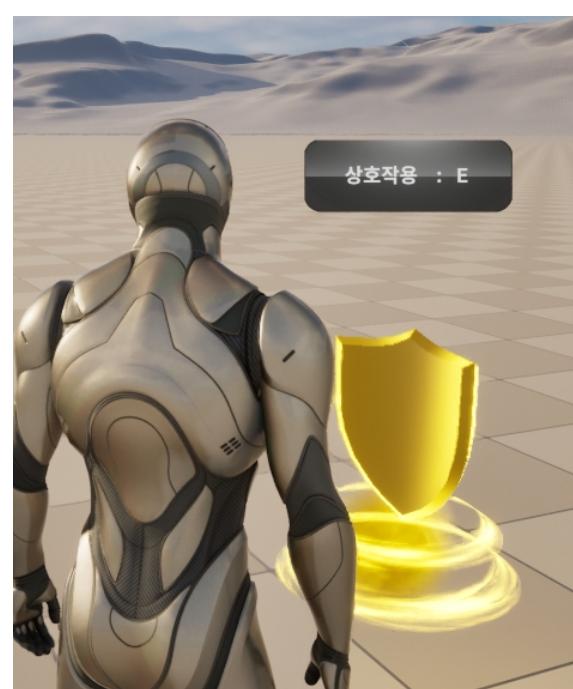
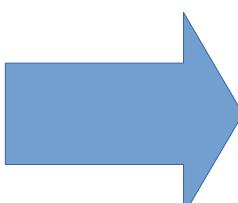
float nearestDist = MAX_FLT;
IInteraction* oldNearest = mNearestInteractable;
for (auto interactable : mInteractableList)
{
    if (!interactable)
    {
        mInteractableList.Remove(interactable);
        continue;
    }

    // 감지 범위 내의 상호작용 액터 중 가장 가까운 액터 찾기
    const float dist = FVector::Dist(interactable->GetActorLocation(), GetActorLocation());
    if (nearestDist > dist)
    {
        nearestDist = dist;
        mNearestInteractableActor = Cast<IIInteraction>(interactable);
        if (!mNearestInteractableActor)
        {
            UE_LOG(LogTemp, Fatal, TEXT("Failed to cast to IIInteraction"));
        }
    }
}
```

```
class PORTFOLIO_API IInteraction : public IGenericTeamAgentInterface
{
    GENERATED_BODY()

public:
    virtual FGenericTeamId GetGenericTeamId() const override { ... }

    // 상호 작용 가능한 상태에 진입했을 때 호출
    virtual void Notify(TObjectPtr<AAActor> player) {};
    // 상호 작용 가능한 상태에서 벗어났을 때 호출
    virtual void UnNotify(TObjectPtr<AAActor> player) {};
    // 상호 작용 시작
    virtual void Interact(TObjectPtr<AAActor> player) {};
    // 상호 작용 끝남
    virtual void UnInteract(TObjectPtr<AAActor> player) {};
};
```



구현된 상호작용 액터들은 ACustomController의 Perception Component에 감지됩니다.

- ✓ **ChangeNotification()** : 새로운 상호작용 액터가 범위 내에 감지되면, 리스트에 추가한다.
- ✓ **UpdateNotifyInteraction()** : 틱마다 호출. 리스트 내의 액터들과 플레이어와의 거리를 계산해, 가장 가까운 액터를 찾아 Notify()를 호출

4) 무기 컴포넌트 (Component/UWeapon)

무기 메쉬, 타입 및 데미지를 포함하는 컴포넌트입니다.

- ✓ **mAudioComponent** : 공격 시에 재생되는 효과음 (ex : 총 발사음)
- ✓ **mBulletClass** : 총 타입 무기에 한해, 공격 시 스폰되는 총알 액터의 클래스
- ✓ **mDamage** : 공격 시 적중 대상에게 적용할 데미지
- ✓ **Attack()** : 공격 함수 (타입에 따라 다른 함수를 호출)
- ✓ **LoadFromItemInfo()** : 무기 아이템 정보를 통해 컴포넌트를 재설정

```
UCLASS()
class PORTFOLIO_API UWeapon : public UStaticMesh
{
    GENERATED_BODY()

    UWeapon(const FObjectInitializer& objInitial
public:
    void Attack();
    void Punch();
    void Fire();

    FString GetWeaponName() const;
    EWeaponType GetType() const;
    float GetDamage() const;

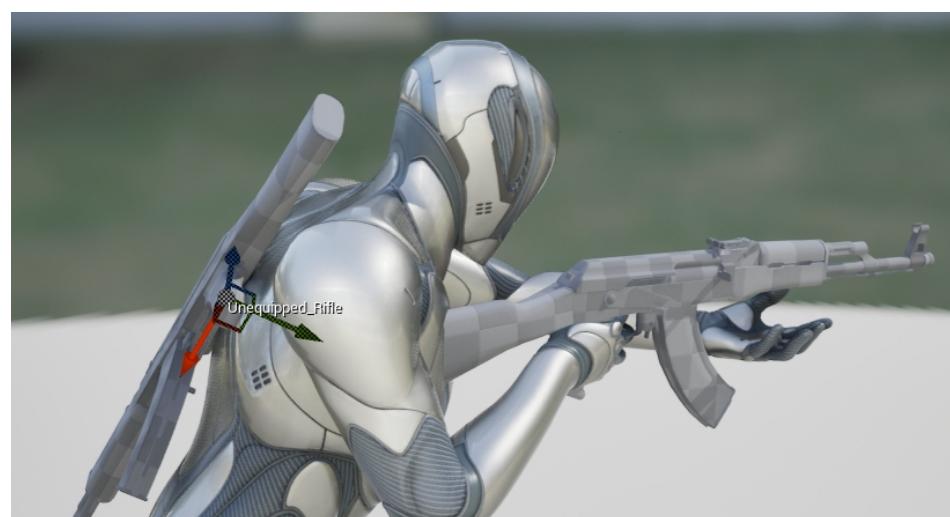
    void LoadFromItemInfo(int itemIndex);
protected:
    UPROPERTY(BlueprintReadWrite)
    TObjectPtr<UAudioComponent> mAudioComponent;

    TSubclassOf<ABullet> mBulletClass;

    FString mWeaponName;
    EWeaponType mType;
    float mDamage;
};
```

```
void APlayerCharacter::SetEquipment(bool isEquip)
{
    // 착용/해제 상태에 따라 소켓에 어태치한다.
    FString socketStr;
    if (isEquip == true)
    {
        socketStr = "Equipped_" + mWeapon->GetW
        mWeaponType = mWeapon->GetType();
    }
    else
    {
        socketStr = "Unequipped_" + mWeapon->Ge
        mWeaponType = EWeaponType::Fist;
    }

    bEquipped = isEquip;
    mWeapon->AttachToComponent(GetMesh(), FAtta
}
```

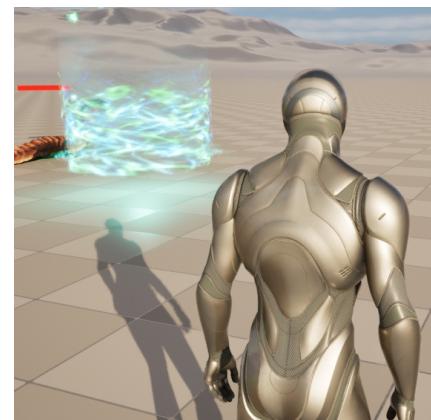


- ✓ UWeapon 컴포넌트는 플레이어가 인벤토리에서 무기 아이템을 선택하거나 퀘슬롯을 돌렸을 때 재설정됩니다.
- ✓ 장착/비장착 상태에 따라 플레이어 메쉬의 특정 소켓에 붙습니다.

5) 퀘스트 처리

- ✓ **CommitQuest** : 해당 퀘스트를 진행 중 상태로 변경합니다.
- ✓ **RegisterSubQuest** : 서브 퀘스트를 시작합니다.
(순차 퀘스트의 경우 이전 서브퀘스트가 완료되어야 다음 서브퀘스트를 시작할 수 있습니다)
- ✓ **CompleteQuest** : 해당 퀘스트를 완료 상태로 변경합니다.
- ✓ **CompleteSubQuest** : 해당 서브퀘스트를 완료시킵니다.
- ✓ **RevertSubQuest** : 해당 서브퀘스트를 미완료 상태로 돌려놓습니다.
- ✓ **ReportArrival** : Arrival 서브퀘스트를 완료시킵니다.
(Arrival 서브퀘스트는 등록 시에 해당 위치에 파티클 액터(AIndicator)를 생성합니다.)
- ✓ **ReportKill** : Enemy 캐릭터가 죽을 때, 플레이어에게 본인의 라벨 목록을 해당 함수에 전달합니다. Hunt 서브퀘스트의 상태를 업데이트합니다.
- ✓ **ReportItem** : 아이템을 얻거나, 잃을 때, Collect 서브퀘스트의 상태를 변경합니다.

```
// 퀘스트 등록
void CommitQuest(int index);
// 서브 퀘스트 등록
void RegisterSubQuest(int questIndex, int subIndex);
// 퀘스트 완료
void CompleteQuest(int index);
// 서브 퀘스트를 완료
void CompleteSubQuest(int questIndex, int subIndex);
// 서브 퀘스트를 다시 미완료 상태로 돌려놓음
void RevertSubQuest(int questIndex, int subIndex);
// Arrival 퀘스트를 완료시킴
void ReportArrival(int questIndex, int subIndex);
// 해당 라벨의 목을 처리하는 퀘스트 상태를 업데이트함
void ReportKill(TArray<int> label);
// 해당 아이템을 수집하는 퀘스트 상태를 업데이트함
void ReportItem(int infoIndex, int Num);
```



6. Enemy (Character/EnemyCharacter.h)

- ✓ **AEnemyCharacter** : 플레이어를 감지하고, 공격할 수 있습니다.
- ✓ 행동 방식을 정의하기 위해 비헤이비어 트리와 블랙보드를 사용했습니다.

0) EEnemyState (Character/Enemy/EnemyController.h)

Enemy Character의 현재 상태를 나타내기 위해 정의하였습니다.

- ✓ **Patrol** : 플레이어를 찾지 못한 상태. 특정 범위 내를 돌아다닙니다.
- ✓ **Caution** : 플레이어가 퍼셉션에 처음 감지된 상태. 플레이어를 가만히 응시합니다.
- ✓ **Detected** : 플레이어를 완벽하게 인지한 상태. 플레이어를 따라다니고, 공격합니다.
- ✓ **Hurt** : 데미지를 입은 상태. 아무런 행동을 하지 못하고 잠시 경직됩니다.

```
UENUM(BlueprintType)
enum class EEnemyState : uint8
{
    Patrol,
    Caution,
    Detected,
    Hurt,
    Count,
};
```

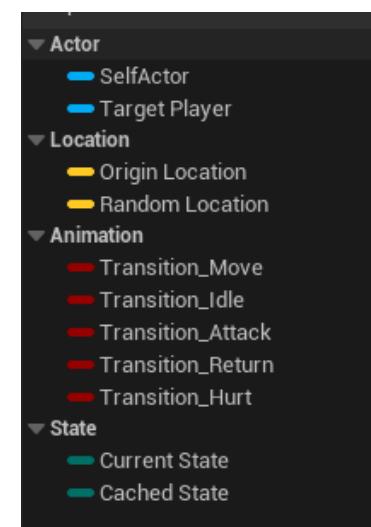
※ 스테이트는 **AEnemyController** 내의 퍼셉션 컴포넌트에 플레이어가 감지됨에 따라 변화합니다.

```
if (Stimulus.WasSuccessfullySensed() == true)
{
    if (mCurrState == EEnemyState::Patrol) // 플레이어를 처음 발견한 상황, 2초간 대기 후 Detected 모드로 전환
    {
        SetState(EEnemyState::Caution);
        GetWorld()->GetTimerManager().SetTimer(mTimer_CautionToDetected, this, &AEnemyController::CautionToDetected, 0.5f, false, 2.0f);
        GetBlackboardComponent()->SetValueAsObject(TEXT("Target Player"), SourceActor);
    }
    else if(mCurrState == EEnemyState::Detected) // Detected 상태에서 플레이어가 시야에서 벗어났다가 다시 발견됨
    {
        GetWorld()->GetTimerManager().ClearTimer(mTimer_ReleaseTarget);
    }
}
else
{
    if (mCurrState == EEnemyState::Caution) // Caution 상태에서 플레이어가 시야에서 벗어남
    {
        SetState(EEnemyState::Patrol);
        GetWorld()->GetTimerManager().ClearTimer(mTimer_CautionToDetected);
    }
    else if (mCurrState == EEnemyState::Detected) // Detected 상태에서 플레이어가 시야에서 벗어난 상황. 5초간 대기 후 Patrol 상태로 전환
    {
        GetWorld()->GetTimerManager().SetTimer(mTimer_ReleaseTarget, this, &AEnemyController::ReleaseTarget, 0.2f, false, 5.0f);
    }
}
```

1) 블랙보드

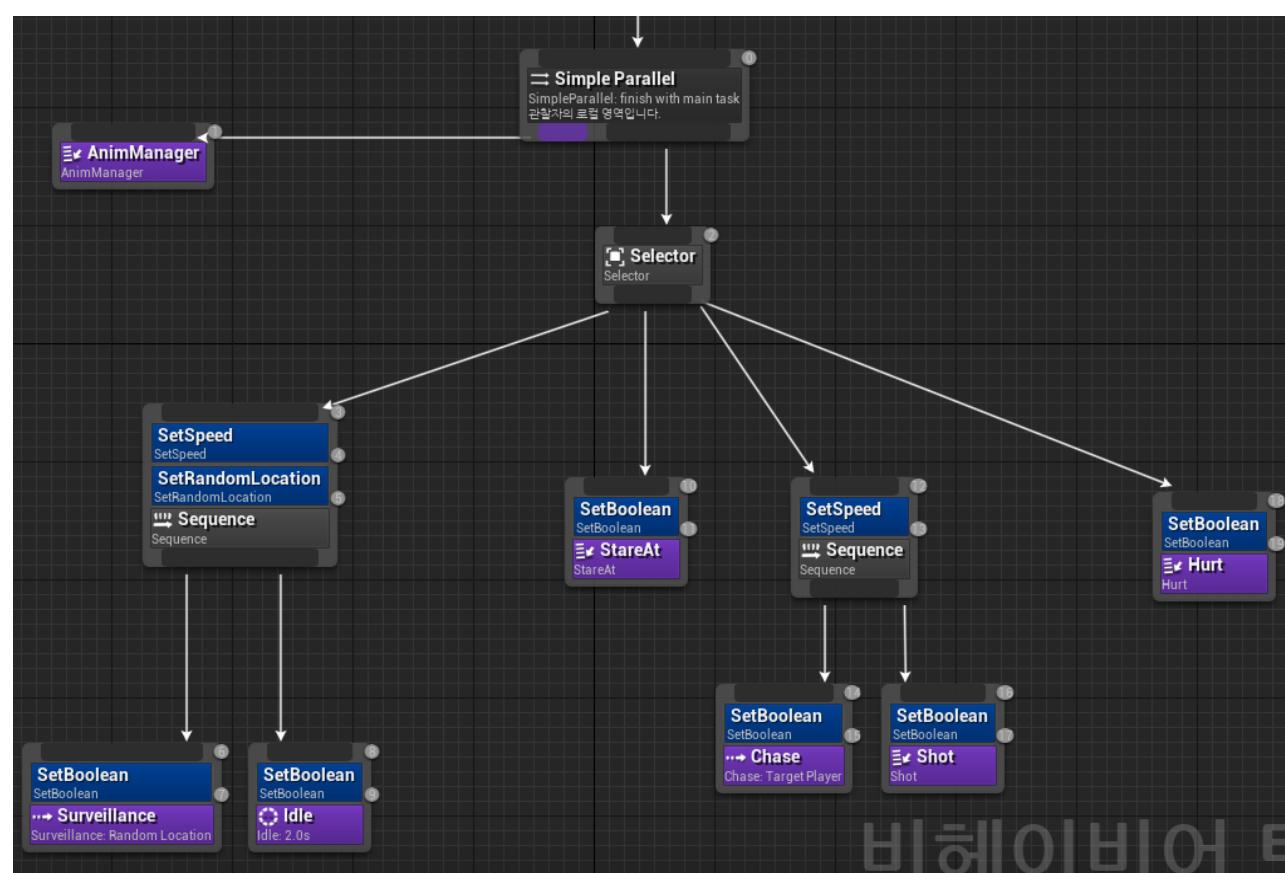
AEnemyCharacter의 하위 클래스인 ASshooter 캐릭터에 사용되는 **ShooterBlackboard**를 예시로 설명하겠습니다.

- ✓ **TargetPlayer** : 컨트롤러의 퍼셉션에 플레이어가 감지되면 해당 플레이어를 담는 포인터
- ✓ **Origin Location** : 캐릭터의 시작 위치 (Patrol의 기준 위치)
- ✓ **Random Location** : 캐릭터가 Patrol 상태일 때, 범위 내에서 이동할 랜덤 위치
- ✓ **Transition_...** : 해당 부울값이 활성화되면 이에 상응하는 애니메이션을 재생하도록 하는 역할
- ✓ **Current State** : 캐릭터의 현재 스테이트
- ✓ **Cached State** : 직전 State (Hurt 상태라면, 경직이 끝난 후 직전 상태로 돌려놔야 합니다).



2) 비헤이비어 트리

AEnemyCharacter의 하위 클래스인 ASshooter 캐릭터에 사용되는 **ShooterBehaviorTree**를 예시로 설명하겠습니다.



- ✓ **UDecorator_SetRandomVector** : 기준 위치를 중심으로 일정한 거리만큼의 랜덤 위치를 설정합니다.

```
// Pivot에서부터 Radius만큼 떨어진 랜덤 지점(Destination)
navSys->GetRandomReachablePointInRadius(pivotLocation, Radius, destination, navData);
if (FVector::Dist(currentLocation, destination.Location) > Radius * 0.5f) // 단, 현재 위치
{
    blackboardComp->SetValueAsVector(Output.SelectedKeyName, destination.Location);
    return;
}
```

- ✓ **UShooterTask_Surveillance** : 지정된 랜덤 위치로 이동합니다. MoveTo 태스크를 상속받아, Enemy State가 **Patrol**을 벗어나면 태스크를 Fail 시킵니다.

```
// Patrol 상태에서 벗어나면 태스크 실패 처리
EEEnemyState Mode = static_cast<EEEnemyState>(OwnerComp.GetAIOwner()->GetEnemyState());
if (Mode != EEnemyState::Patrol)
{
    FinishLatentTask(OwnerComp, EBTNodeResult::Failed);
}
```

- ✓ **UEnemyTask_Idle** : 해당 위치에서 n초간 대기합니다. Enemy State가 **Patrol**을 벗어나면 태스크를 Fail 시킵니다.

✓ **Patrol 상태에서는 위의 두 태스크를 반복합니다. (랜덤 위치에 도착 → n초간 대기)**

- ✓ **UEnemyTask_StareAt** : Idle 태스크와 같지만, **Caution** 상태일 때만 지속됩니다.
- ✓ **UEnemyTask_Chase** : 공격 사거리에 도달할 때까지 타겟을 쫓습니다. **Detected** 상태일 때만 지속됩니다.
- ✓ **UShooterTask_Shot** : 공격 사거리에 도달하면, 타겟을 향해 총알을 발사합니다. 애니메이션이 끝날 때까지 지속됩니다.

| | |
|---|---|
| <pre>// 타겟을 향해 몸을 돌림 const FVector dir = (target->GetActorLocation() - GetActorLocation()).GetSafeNormal(); const FRotator rot = FRotationMatrix::MakeFromX(dir).Rotator(); mOwnerShooter->SetActorRotation(rot); // 해당 방향으로 발사 mOwnerShooter->Shoot();</pre> | <pre>// 발사 애니메이션이 끝났거나, Detected 상태에서 벗어나게 되면 태스크를 종료 const bool isPlaying = mOwnerShooter->GetMesh()->GetSingleNodeInstance()->IsPlaying(); const EEnemyState state = static_cast<EEEnemyState>(OwnerComp.GetAIOwner()->GetEnemyState()); if ((isPlaying == false) (state != EEnemyState::Detected)) { FinishLatentTask(OwnerComp, EBTNodeResult::Succeeded); }</pre> |
|---|---|

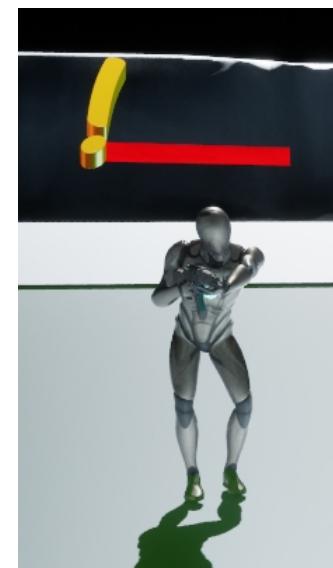
- ✓ **UShooterTask_AnimManager** : Simple Parallel 노드를 통해 계속 실행되는 태스크입니다. Transition키의 값이 true가 되면 상응하면 애니메이션을 재생합니다.

```
// 각 부울값이 true로 변경되면 해당 애니메이션으로 전환한다.
if (blackboardComp->GetValueAsBool(Transition_Idle_Key.SelectedKeyName) == true)
{
    blackboardComp->SetValueAsBool(Transition_Idle_Key.SelectedKeyName, false);
    ownerCharacter->GetMesh()->PlayAnimation(LoadHelper::LoadObjectFromPath<UAnimSequence>(TEXT("/Game/Character/Idle")));
}
else if (blackboardComp->GetValueAsBool(Transition_Move_Key.SelectedKeyName) == true)
{
    blackboardComp->SetValueAsBool(Transition_Move_Key.SelectedKeyName, false);
    ownerCharacter->GetMesh()->PlayAnimation(LoadHelper::LoadObjectFromPath<UAnimSequence>(TEXT("/Game/Character/Move")));
}
```

3) 그 외

```
// 캐릭터 위에 표시되는 체력 바 UI의 클래스
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "UI")
TSubclassOf<UHpBar> HpBarClass;
// 물음표 마크 (Caution 상태일 때 활성화)
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "UI")
TObjectPtr<UBillboardComponent> mQuestionMark;
// 느낌표 마크 (Detected 상태일 때 활성화)
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "UI")
TObjectPtr<UBillboardComponent> mExclamationMark;
// Hp 바 (플레이어의 감지 범위에 들어오면 활성화)
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "UI")
TObjectPtr<UWidgetComponent> mHpBarWidget;
public:
```

- ✓ **QuestionMark** : Caution State일 때 물음표 마크를 띠워 무언가 감지했음을 알립니다.
- ✓ **ExclamationMark** : Detected State일 때 느낌표 마크를 띠워 플레이어를 공격함을 알립니다.
- ✓ **HpBarWidget** : 현재 Hp를 나타내는 프로그레스 바 위젯입니다.

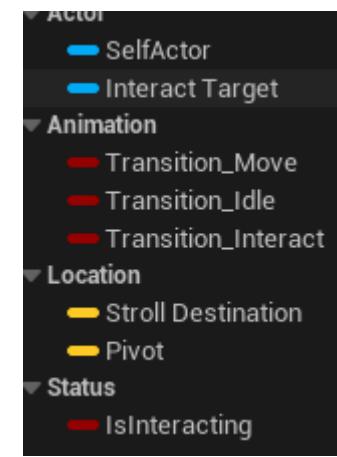


7. Npc (Character/NpcCharacter.h)

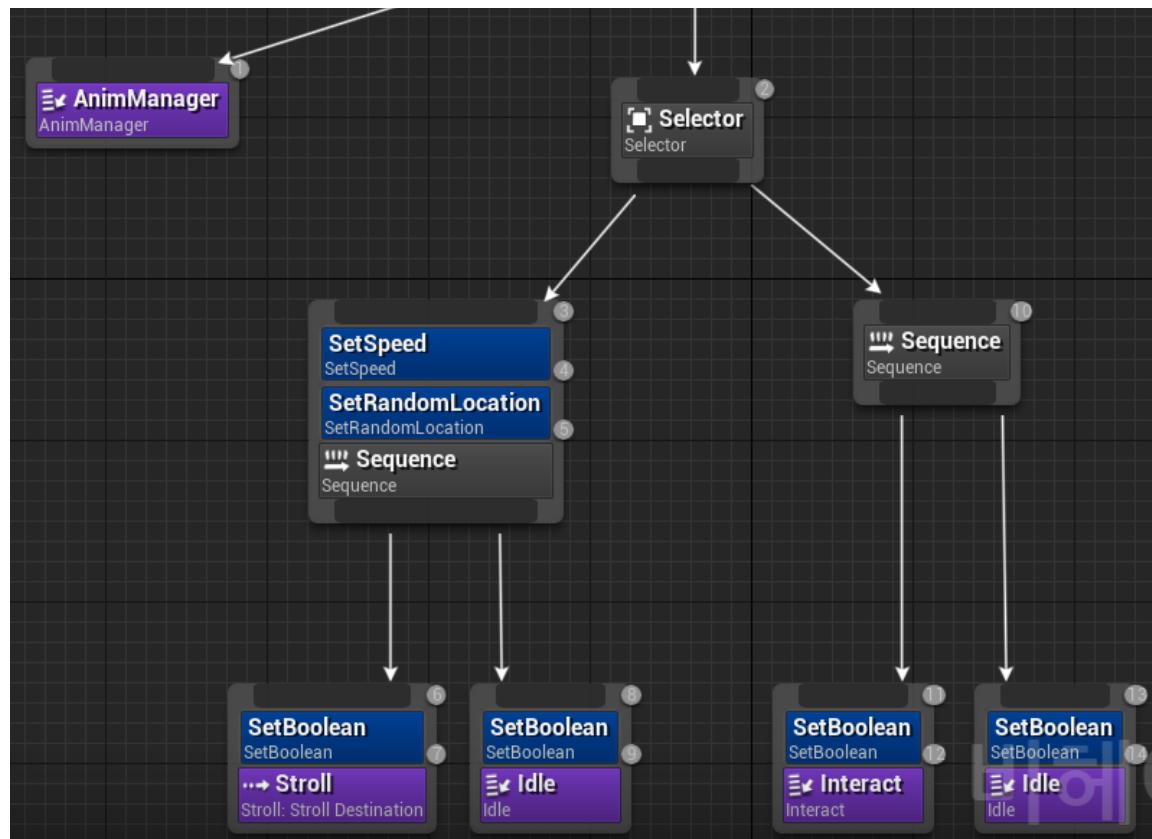
- ✓ **ANpcCharacter** : 플레이어와 상호 작용할 수 있는 캐릭터입니다. 상호 작용 시 대화가 가능하고, 아이템을 사고 팔 수 있습니다.
- ✓ EnemyCharacter와 마찬가지로 비헤이비어 트리와 블랙보드를 사용해 행동 방식을 정의했습니다.

1) 블랙보드

- ✓ **Interact Target** : 상호작용 중인 타겟 플레이어
- ✓ **Pivot** : 캐릭터의 시작 위치 (Patrol의 기준 위치)
- ✓ **Stroll Destination** : 캐릭터가 범위 내에서 이동할 랜덤 위치
- ✓ **Transition_...** : 해당 부울값이 활성화되면 이에 상응하는 애니메이션을 재생하도록 하는 역할
- ✓ **IsInteracting** : 현재 캐릭터가 상호작용 중인가?



2) 비헤이비어 트리



- ✓ **UNpcTask_Stroll** : 앞의 **Surveillance** 태스크와 비슷합니다. 상호작용이 시작되면 태스크를 실패 처리합니다.

```

Super::TickTask(OwnerComp, NodeMemory, DeltaSeconds);

const bool isInteracting = OwnerComp.GetAIOwner()->GetBool("IsInteracting");
if (isInteracting == true)
{
    FinishLatentTask(OwnerComp, EBTNodeResult::Failed);
}
  
```

- ✓ **UNpcTask_Idle** : 앞의 **Idle** 태스크와 비슷합니다. 상호작용이 시작되면 태스크를 실패 처리합니다.
- ✓ **UNpcTask_Interact** : 플레이어와 상호작용 하는 동안 실행되는 태스크입니다. 상호작용이 끝나면 태스크를 성공시킵니다.

```

d UNpcTask_Interact::TickTask(UBehaviorTreeComponent& OwnerC
const bool isInteracting = OwnerComp.GetAIOwner()->GetBool("IsInteracting");
if (isInteracting == false)
{
    FinishLatentTask(OwnerComp, EBTNodeResult::Succeeded);
}
  
```

- ✓ **종합** : 일정 범위를 돌아다니다가, 플레이어와 상호 작용하면 그 동안은 특정 위치에 고정됩니다.

8. UI (UI/...)

- ✓ 공통적으로, 모든 UI는 **UUserWidget**을 상속받아 C++로 로직을 작성한 후, 이를 블루프린트로 상속받아 디자인하는 방식으로 구현했습니다.

```
void ACustomController::OpenMenu(TObjectPtr<APlayerCharacter> player)
{
    SetInputMode(FInputModeUIOnly());
    SetShowMouseCursor(true);
    mMenuWidget->AddToViewport();
    mMenuWidget->Open(player);
}
```

- ✓ **ACustomController** 클래스가 UI 오브젝트를 가지고 있고, 플레이어 입력 또는 로직에 따라 UI를 뷰포트에 표시/삭제합니다.

1) 인트로 (UI/IntroWidget.h)

게임의 첫 화면을 담당하는 UI입니다. 위젯 스위처를 사용해 어떤 버튼을 누르느냐에 따라 특정 위젯들을 보이게 하는 방식으로 구현했습니다.



- ✓ **New Game** : 새로운 게임을 시작합니다.
- ✓ **Load Game** : 기존 게임을 불러옵니다.



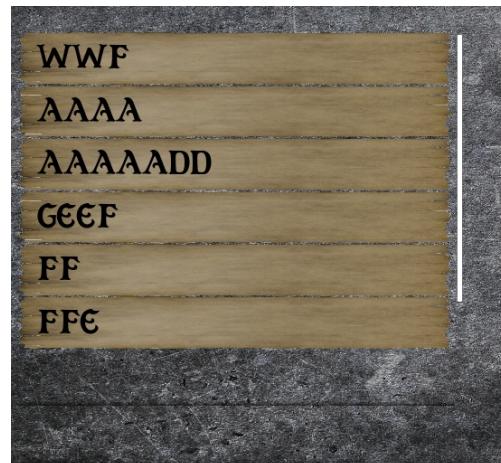
```
const FText newName = mNewNameText->GetText();
if (newName.IsEmpty() == true)
{
    // 이름이 입력되지 않음
    ShowPopup(TEXT("이름을 입력하세요"));

    return;
}

const int savedSlot = gi->CreateSaveFile(mNewNameText);
if (savedSlot < 0)
{
    ShowPopup(TEXT("저장 슬롯이 가득 창습니다"));
    return;
}

gi->LoadGame(savedSlot);
```

- ✓ **New Game 선택 시** : 이름을 입력하는 칸과 게임을 시작하는 버튼이 나타납니다. 버튼을 누르면 세이브 파일이 생성되고, 게임이 시작됩니다.

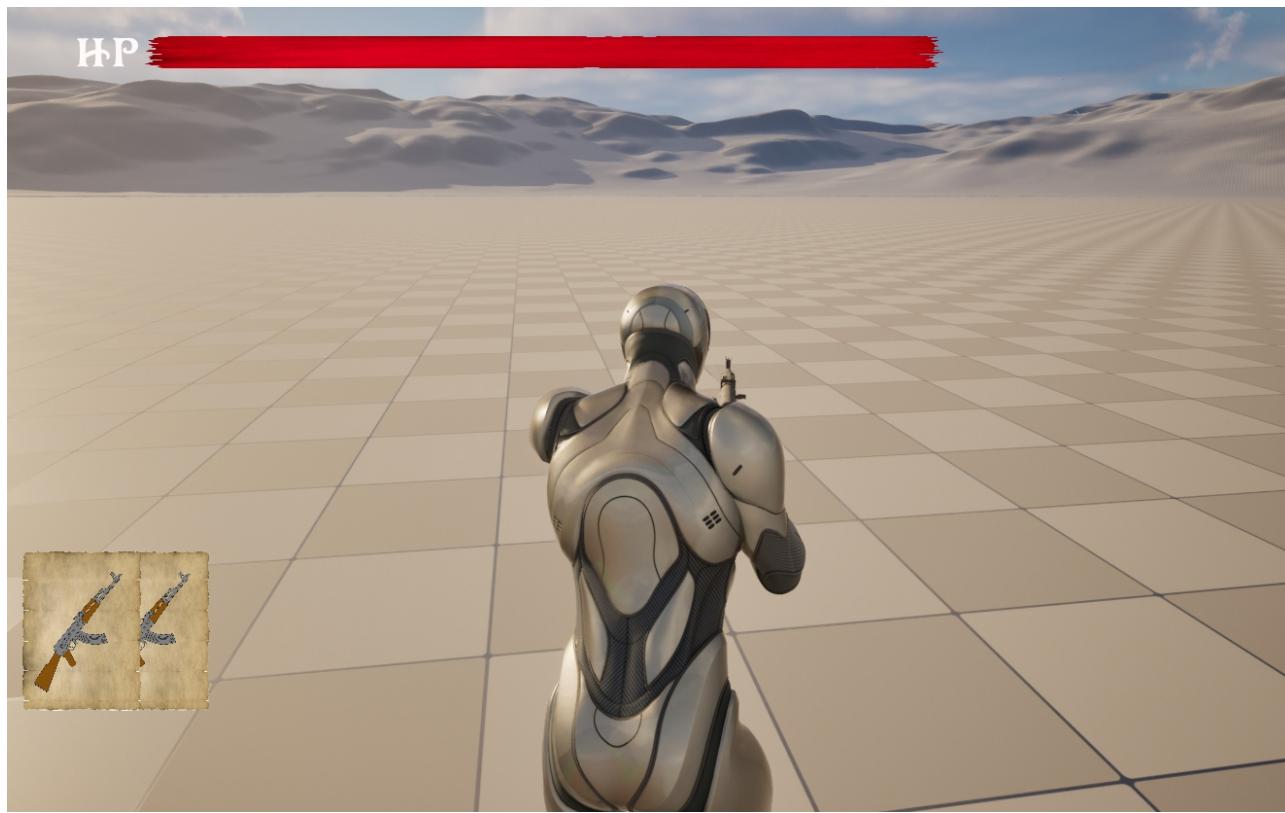


```
void UIintroWidget::OnClicked_LoadGame()
{
    TObjectPtr<UCustomGameInstance> gi =
        check(gi);

    gi->LoadGame(mSelectedSlot);
}
```

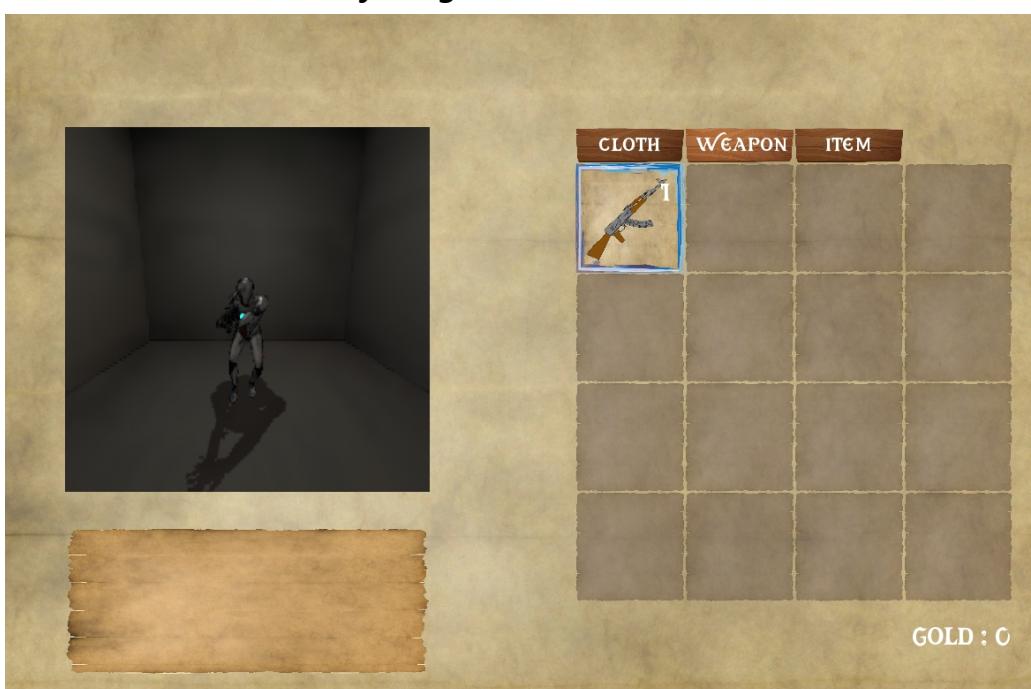
- ✓ **Load Game 선택 시** : 세이브 슬롯 목록과, 슬롯을 눌렀을 때 간단한 정보를 출력하는 출력창, 게임을 시작하는 버튼이 있습니다. 시작 버튼을 누르게 되면, 세이브 파일로부터 데이터를 받아와 게임을 시작합니다.

2) 메인 UI (UI/HUDWidget.h)



- ✓ 메인 UI는 현재 체력을 나타내는 **HP바**, 현재 선택 중인 무기를 나타내는 **퀵슬롯**으로 이루어져 있습니다.

3) 인벤토리 (UI/InventoryWidget.h)

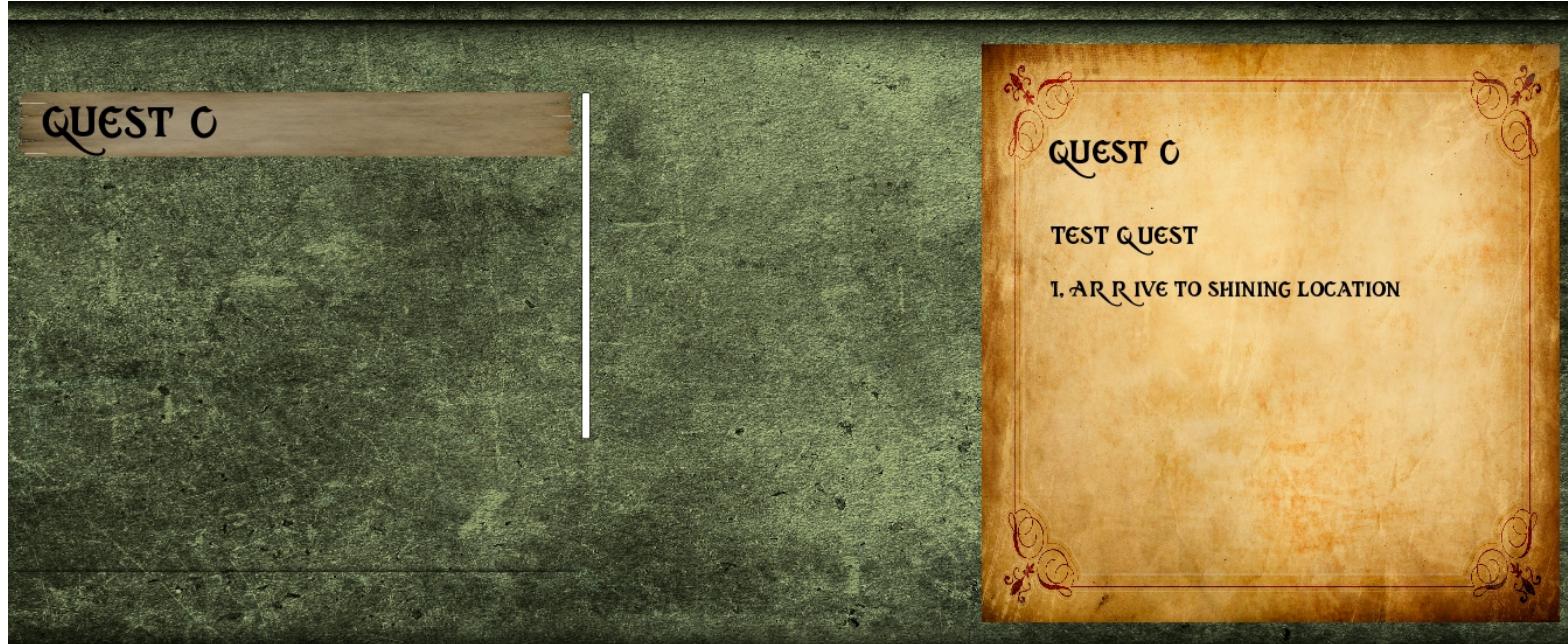


- ✓ 인벤토리는 **아이템 타입 템**으로 나뉘어져 있고, 각 템을 누르게 되면 템에 맞는 아이템만 나타납니다. 구현한 사항은 아래와 같습니다.



- ✓ 슬롯에 포인터를 옮겨놓을 때(**Hovered**) : 아이템의 이름을 설명란에 출력합니다. (스크린샷에는 마우스 포인터가 보이지 않습니다)
- ✓ 슬롯을 클릭했을 때(**Clicked**) : 아이템을 선택합니다.
- ✓ 프리뷰 캐릭터 : 레벨 액터를 통해 지면 아래에 프리뷰 캐릭터를 스폰합니다. 인벤토리 내에서 캐릭터의 선택을 그대로 보여줍니다.
(APreviewCharacter 클래스를 사용합니다.)

4) 퀘스트 상태창 (UI/QuestWidget.h)



현재 맵은 퀘스트 목록과, 각 목록을 클릭했을 때 정보를 출력할 상태창으로 이루어져 있습니다. 출력할 정보는 다음과 같습니다.

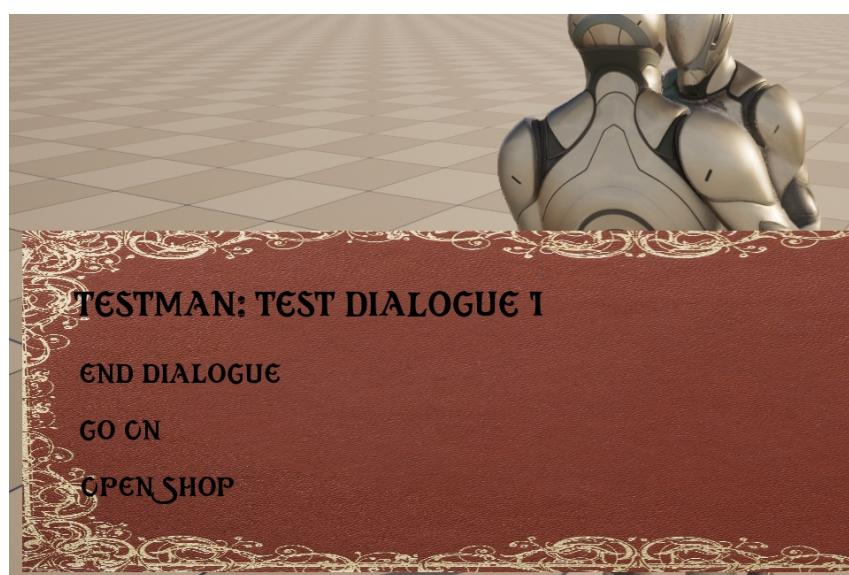
- ✓ 퀘스트 이름
- ✓ 퀘스트 설명
- ✓ 서브 퀘스트 설명 및 진행도

5) 상점 (UI/ShopWidget.h)



- ✓ 인벤토리와 비슷하게 구현하였으며, NPC쪽 슬롯을 선택하면 구매, 플레이어 쪽 슬롯을 선택하면 판매를 진행할 수 있습니다.
- ✓ 슬롯을 누르면 판매/구매 개수를 정할 수 있고, 확인 버튼을 누르면 완료됩니다.

6) 다이얼로그 (UI/DialogueWidget.h)



- ✓ NPC와 상호작용하면 나타나는 대화 UI입니다. 대사와 선택 가능한 응답이 존재합니다.
- ✓ 해당 응답을 선택하면 FDialogueEvent로 정의된 이벤트를 발생시킵니다.

```
void UDialogueWidget::OnPressed_Response(int i)
{
    check(i < MAX_DIALOGUE_RESPONSE);

    bool jumped = false;
    bool end = false;
    bool openShop = false;

    for (auto& event : mOwnerNpc->GetDialogue())
    {
        switch (event.EventType)
        {
        case EDialougeEventType::End:
            end = true;
            break;
        case EDialougeEventType::Jump:
            jumped = true;
            mCurrIndex = event.JumpIndex;
            break;
        case EDialougeEventType::CommitQuest:
        {
            mOwnerPlayer->CommitQuest(event.Quest);
            break;
        }
    }
}
```