

NewLife

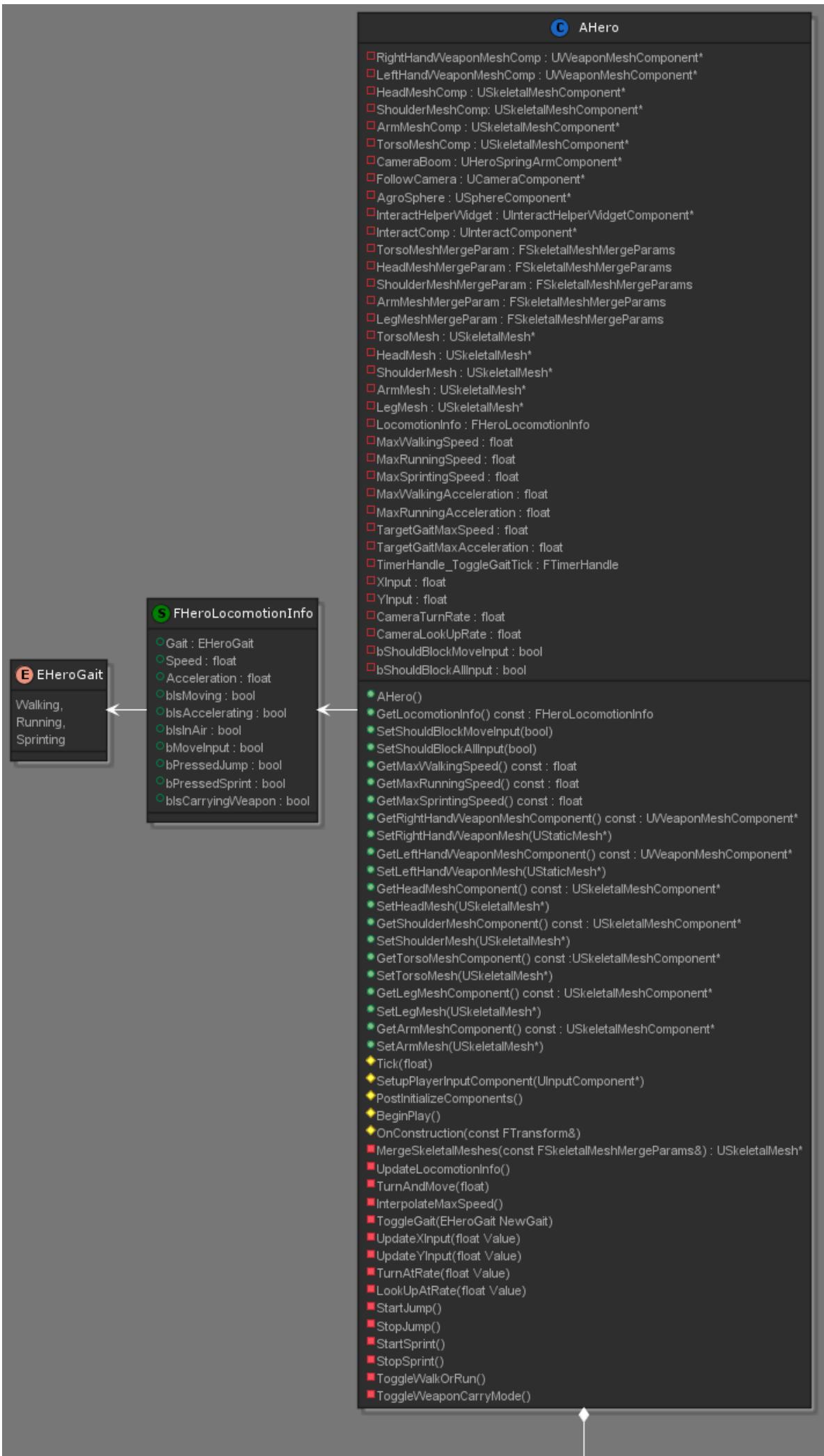
Unreal Engine 4.26 Portfolio

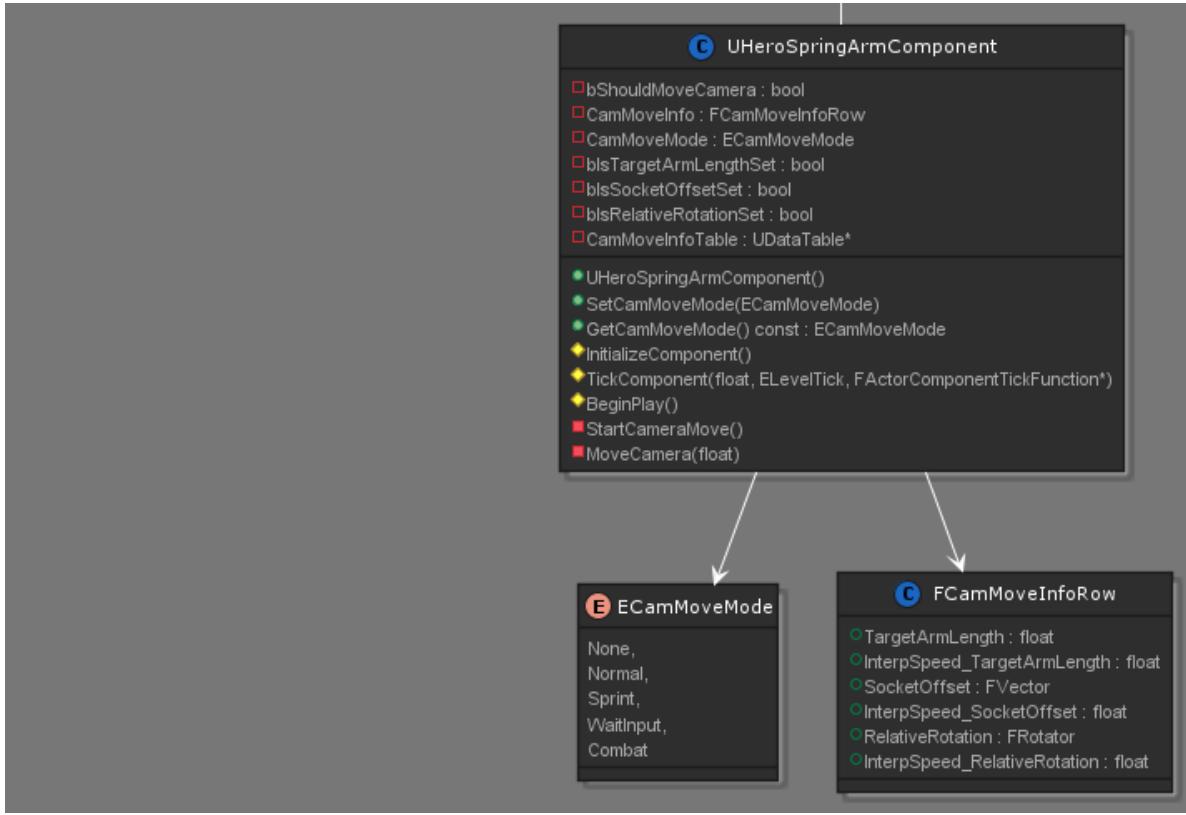
프로젝트 버전 : 0.6v



- 장르 : 3인칭 액션 게임
- 엔진버전 : Unreal Engine 4.26
- IDE : Rider for Unreal Engine 2021.1
- 작업 기간 : 두달 내외
- 영상 길이 : 4분 내외 (꼭 보시는 걸 추천합니다.)
- 본 프로젝트는 언리얼 엔진 코딩 스탠다드를 따릅니다.
- 언리얼 코딩 스탠다드 : (<https://docs.unrealengine.com/4.27/ko/ProductionPipelines/DevelopmentSetup/CodingStandard/>)
- 본 프로젝트의 애셋 이름은 이 링크의 명명 규칙을 따릅니다.
- 애셋 명명 규칙 : (https://github.com/ymkim50/ue4-style-guide/blob/master/README_Kor.md)
- 아래 서술된 내용들은 영상을 보고 궁금하실 수 있는 것들 위주로 설명하고 있습니다.
- 각 내용마다 참고한 페이지들이 나오는데 코드를 배끼지 않았으며 충분히 이해하고 사용했음을 알립니다.

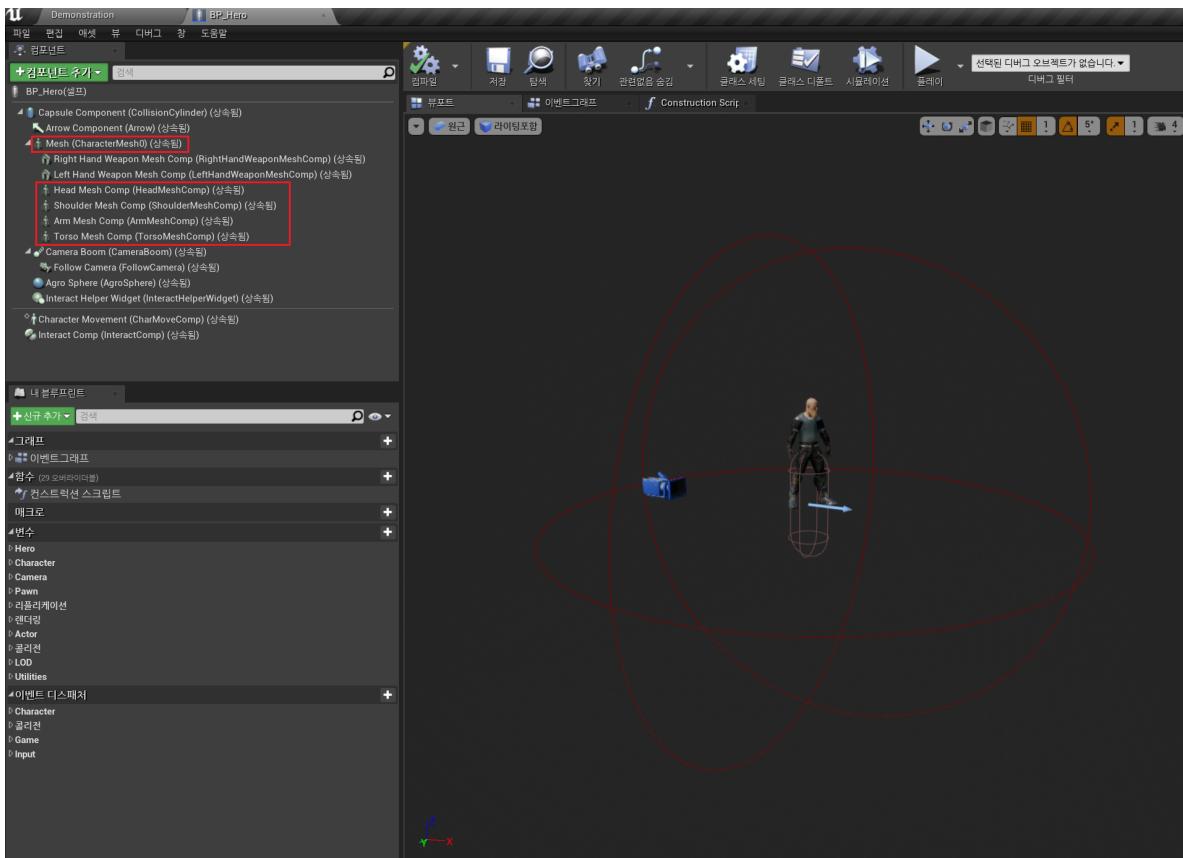
Character Setting(AHero)



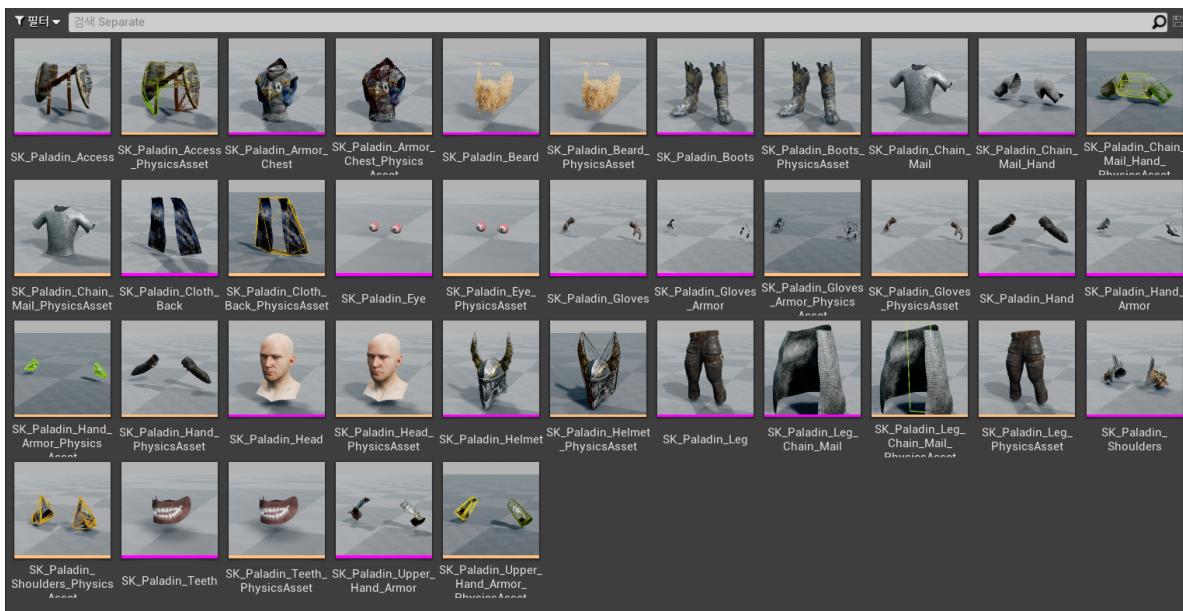


Modular Character

- 본 캐릭터는 5개의 파트로 구성된 모듈러 캐릭터입니다.
 - Mesh : 캐릭터의 발
 - HeadMeshComp : 캐릭터의 머리
 - ShoulderMeshComp : 캐릭터의 어깨
 - ArmMeshComp : 캐릭터의 팔
 - TorsoMeshComp : 캐릭터의 가슴
 - MasterPoseComponent를 사용하여 Mesh를 부모로, 나머지를 Mesh의 자손으로 구성됩니다.



- 문제는 사용해야 할 Skeletal Mesh의 부분이 저렇게 딱 5개의 파트로 나뉘지지 않았다는 점입니다.
- 제가 사용한 애셋은 굉장히 작은 부분들로 나누어져 있었습니다.



- 이 캐릭터의 머리부분을 만들기 위해 Eye, Head, Teeth, Beard를 합쳐야 하는데 이 부분은 스켈레탈 메시 병합을 이용해 해결했습니다.
- 이렇게 한 이유는 렌더링 비용 때문인데, 각 부분을 모두 USkeletalMeshComponent를 만들어서 사용하면 그 숫자만큼 드로우콜이 비례하기 때문입니다.
- 즉, 아주 작은 부분을 5개의 큰 부분으로 스켈레탈 메시 병합을 하고 5개의 큰 부분은 MasterPoseComponent를 통해 메시들이 같은 애니메이션을 따르게 되는 것입니다.

```
void AHero::OnConstruction(const FTransform& Transform)
{
    Super::OnConstruction(Transform);
```

```

// Merge seperated skeletal mesh parts of characters into 5 skeletal
meshes...\n
    // 디폴트 메시를 저장해놔야 디폴트 메시로 돌아와야 할 때 사용가능합니다.
    // 예: 아이템을 착용했다가 해제하는 경우
    TorsoMesh = MergeSkeletalMeshes(TorsoMeshMergeParam);
    HeadMesh = MergeSkeletalMeshes(HeadMeshMergeParam);
    ShoulderMesh = MergeSkeletalMeshes(ShoulderMeshMergeParam);
    LegMesh = MergeSkeletalMeshes(LegMeshMergeParam);
    ArmMesh = MergeSkeletalMeshes(ArmMeshMergeParam);

    // Set character's skeletal mesh components's mesh...
    GetMesh()->SetSkeletalMesh(LegMesh);
    HeadMeshComp->SetSkeletalMesh(HeadMesh);
    ShoulderMeshComp->SetSkeletalMesh(ShoulderMesh);
    ArmMeshComp->SetSkeletalMesh(ArmMesh);
    TorsoMeshComp->SetSkeletalMesh(TorsoMesh);

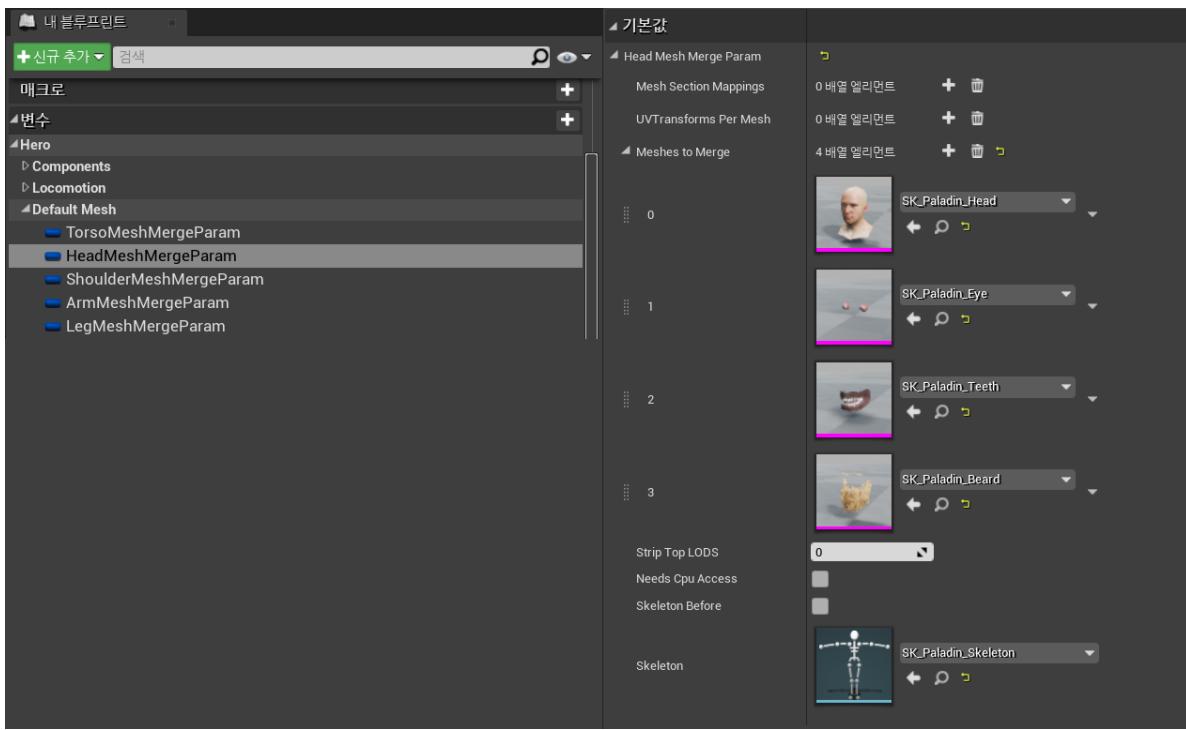
    // Set master pose component...
    HeadMeshComp->SetMasterPoseComponent(GetMesh());
    ShoulderMeshComp->SetMasterPoseComponent(GetMesh());
    ArmMeshComp->SetMasterPoseComponent(GetMesh());
    TorsoMeshComp->SetMasterPoseComponent(GetMesh());

    // ...
}

uskeletalMesh* AHero::MergeSkeletalMeshes(const FSkeletalMeshMergeParams&
MeshMergeParams)
{
    if(MeshMergeParams.MeshesToMerge.Num() == 0)
    {
        return nullptr;
    }
    else if(MeshMergeParams.MeshesToMerge.Num() == 1)
    {
        return MeshMergeParams.MeshesToMerge[0];
    }

    uskeletalMesh* Result =
UMeshMergeFunctionLibrary::MergeMeshes(MeshMergeParams);
    return Result;
}

```



참고 페이지

- 모듈식 캐릭터 작업
 - <https://docs.unrealengine.com/4.27/ko/AnimatingObjects/SkeletalMeshAnimation/WorkingwithModularCharacters/>
- Unreal Locomotion Blendspace with Rootmotion
 - https://www.youtube.com/watch?v=YtWbl50jwwc&t=1260s&ab_channel=CodeLikeMe
- Advanced Locomotion System V4
 - <https://www.unrealengine.com/marketplace/en-US/product/advanced-locomotion-system-v1?sessionInvalidated=true>
- Bringing a Hero from Paragon to Life with UE4
 - 움직이기 시작하는 모션과 멈추는 모션을 부드럽게 구현하기 위해 참고하였습니다.
 - https://www.youtube.com/watch?v=YIKA22Hzerk&t=991s&ab_channel=LaurentDelayen

AnimInstance and Foot IK Placement

- 캐릭터는 매 프레임마다 Locomotion 정보를 업데이트하고 이 정보에 따라 캐릭터가 움직이게 됩니다.

```
// Called every frame
void AHero::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // 캐릭터의 속도, 가속여부, 추락여부, 방향키 입력 여부 등의 Locomotion Info를 업데이트합니다.
    UpdateLocomotionInfo();

    // 캐릭터를 실제로 움직이는 함수
    TurnAndMove(DeltaTime);
}

void AHero::TurnAndMove(float DeltaTime)
```

```

{
    const FRotator CameraWorldRotation(0.f, GetControlRotation().Yaw, 0.f);
    const FVector
ForwardBasedXInput(FRotationMatrix(CameraWorldRotation).GetScaledAxis(EAxis::X)
* XInput);
    const FVector
RightBasedYInput(FRotationMatrix(CameraWorldRotation).GetScaledAxis(EAxis::Y) *
YInput);

    FVector DirectionToMove(ForwardBasedXInput + RightBasedYInput);
    DirectionToMove.Normalize();

    // Turn My Self
    if (LocomotionInfo.bIsMoving && LocomotionInfo.bMoveInput)
    {
        FRotator TurnTo;

        // 무기를 들고 있을때와 안들고 있을 때의 움직임의 방식이 다릅니다.
        if(LocomotionInfo.bIsCarryingWeapon)
        {
            const FVector
CameraForwardVector(FRotationMatrix(CameraWorldRotation).GetScaledAxis(EAxis::X)
);

            TurnTo = FRotator(FMath::RInterpTo(GetActorRotation(),
CameraForwardVector.Rotation(), DeltaTime,
                                         HERO_TURN_RATE));
        }
        else
        {
            TurnTo = FRotator(FMath::RInterpTo(GetActorRotation(),
DirectionToMove.Rotation(), DeltaTime,
                                         HERO_TURN_RATE));
        }

        TurnTo.Pitch = 0.f;
        SetActorRotation(TurnTo);
    }

    // Move My Self
    float MoveAmount;
    if (XInput == 0 || YInput == 0)
    {
        MoveAmount = FMath::Abs(XInput) + FMath::Abs(YInput);
    }
    else
    {
        MoveAmount = (FMath::Abs(XInput) + FMath::Abs(YInput)) / 2;
    }

    if(LocomotionInfo.bIsCarryingWeapon)
    {
        if(FVector::DotProduct(GetActorForwardVector(), DirectionToMove) <
-0.01f)
        {
            MoveAmount *= .75f;
        }
        else
    }
}

```

```
        {
            MoveAmount *= .9f;
        }
    }

AddMovementInput(DirectionToMove, MoveAmount);
}
```

- 그리고 AnimInstance는 매 프레임마다 LocomotionInfo를 가져와서 애니메이션을 업데이트 합니다.
 - AnimInstance는 private 멤버로 FootIKManager를 가지고 있는데 이 FootIKManager가 Foot Placement를 계산합니다.

```

FHumanFootIKInfo CalculateFootIKInterpolation(float DeltaTime,
FHumanFootIKInfo CurrentFootIKInfo,
FHumanFootIKInfo
DesiredFootIKInfo);

static const float TraceDistanceFromFoot;
};

}

```

- UpdateFootIKInfo는 발 아래로 LineTrace를 해서 캐릭터의 양발 위치를 계산하고 발의 각도 또한 계산해줍니다.
- 실직적인 계산은 전부 여기서 이루어지고 발이 순간이동하는 걸 원하지 않기 때문에 값을 보간하여 조금이라도 부드럽게 이동하게 해줍니다.

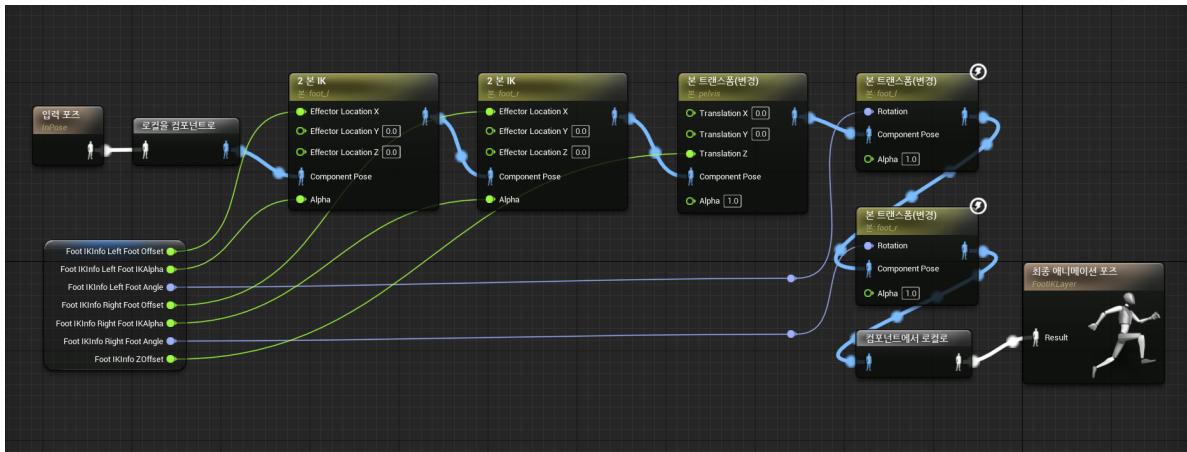
```

void UHeroAnimInstance::NativeUpdateAnimation(float DeltaTime)
{
    Super::NativeUpdateAnimation(DeltaTime);

    //...
    // Place Foot By Using FFootIKHelper
    if (!bIsInAir && !bIsMoving)
    {
        FootIKInfo = FootIKManager->UpdateFootIKInfo(GetWorld(), Owner,
FootIKInfo, DeltaTime);
    }
    else
    {
        const FHumanFootIKInfo EmptyFootIKInfo;
        FootIKInfo = EmptyFootIKInfo;
    }
}

```

- AnimGraph - Foot IK Layer



참고 페이지

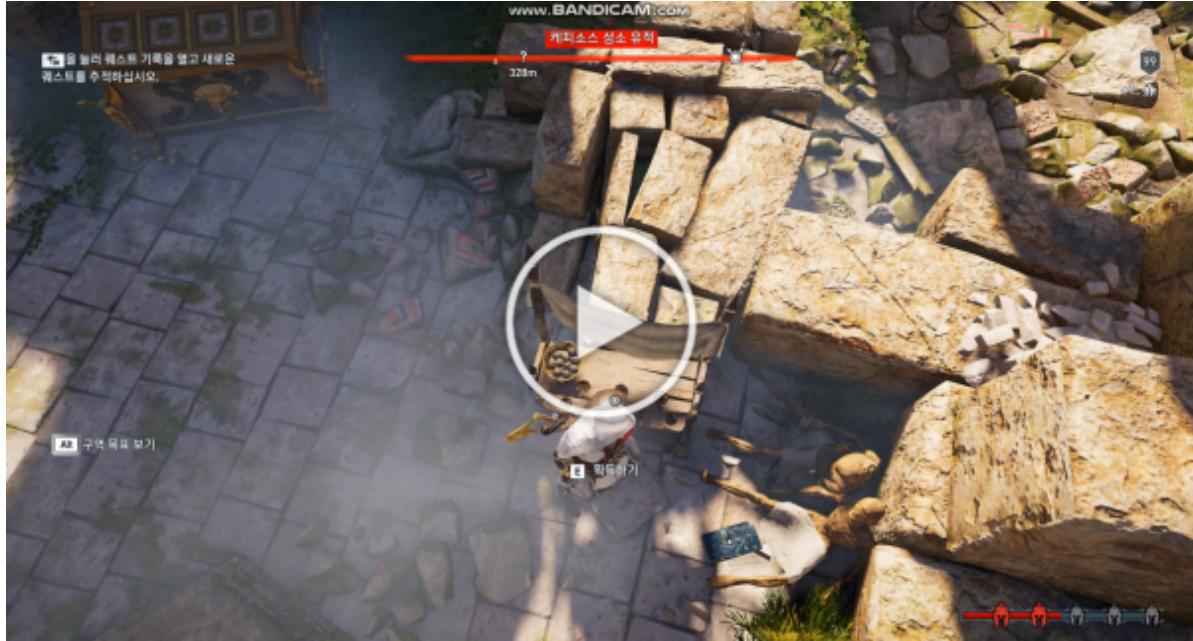
- CatDarkGame : UE4 Foot IK을 C++로 구현하기
 - <https://darkcatgame.tistory.com/23?category=813793>
- Unreal Leg IK 1 - Foot Placement
 - https://www.youtube.com/watch?v=kK9rQzbSzio&t=904s&ab_channel=CodeLikeMe
- Advanced Locomotion System V4
 - 이 프로젝트에서 Foot Placement 부분을 어떻게 사용했는지 참고했습니다.

- <https://www.unrealengine.com/marketplace/en-US/product/advanced-locomotion-system-v1?sessionInvalidated=true>

Interact

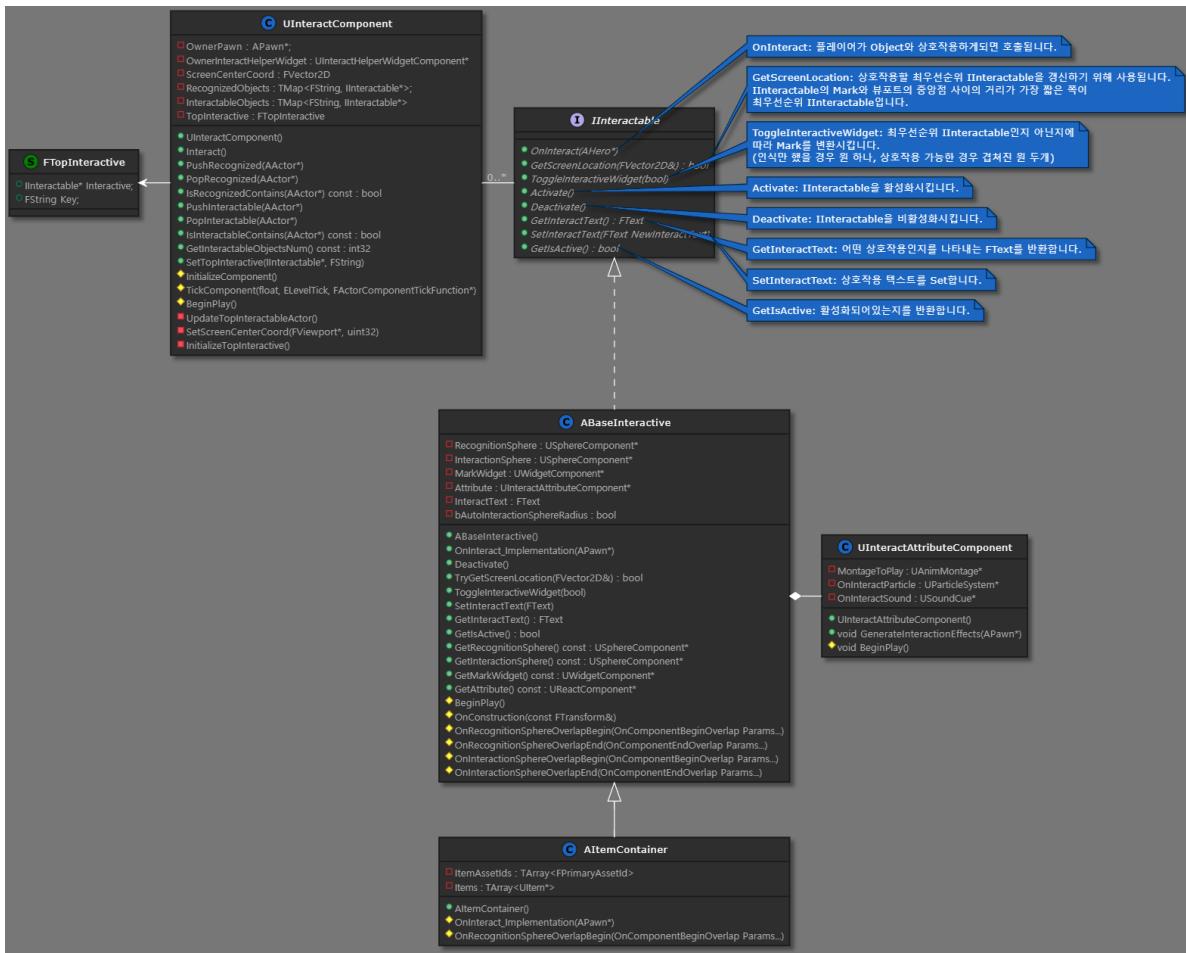
motivation

- 본 프로젝트의 상호작용은 어쌔신 크리드 오디세이에서 따왔습니다.(영상길이: 7초)



- AC Odyssey Interact features
 - 어느 정도 가까이가면 물체를 인식함(인식 여부는 물체에 원 이미지를 띠움)
 - 더 가까이 가면 물체와 상호작용할 수 있음(상호작용 가능 여부는 겹쳐있는 두개의 원 이미지를 띠움)
 - 유저가 바라보는 방향에 따라 상호작용 우선순위가 바뀜
 - 물체마다 상호작용 텍스트 설정 가능함

Interactable Actor UMG Diagram



- AHero(본 프로젝트의 플레이어블 캐릭터)는 UIInteractComponent를 들고 있고 이 컴포넌트가 IInteractable과의 상호작용을 담당합니다.

UIInteractComponent

- 가장 주목해서 볼 것은 RecognizedObjects와 InteractableObjects입니다.
- 이 두 컨테이너는 키(액터 이름)와 값(IInteractable*)을 가지는 TMap입니다.
 - RecognizedObjects : 내 캐릭터가 인식했지만 아직 상호작용할 수 없는 액터들의 정보를 담습니다.
 - InteractableObjects : 상호작용할 수 있는 액터들의 정보를 담습니다.

UIInteractComponent의 기본 동작

- 캐릭터가 A라는 물체와 상호작용한다고 가정해보겠습니다.
 - 캐릭터는 A와의 거리가 어느정도 가까워지면 A를 인식하고 그 정보를 RecognizedObjects에 담습니다.
 - 캐릭터와 A와의 거리가 더 가까워지면 A를 RecognizedObjects에서 꺼내고 InteractableObjects에 담습니다.
 - InteractableObjects의 갯수가 0보다 크면 틱마다 상호작용 최우선순위 액터를 생성합니다.
 - 캐릭터와 A가 상호작용하면 A는 단순히 자신을 비활성화합니다.
 - 캐릭터와 A가 어느정도 멀어지면 A를 InteractableObjects에서 꺼내고 RecognizedObjects에 담습니다.
 - 캐릭터와 A가 더 멀어지면 RecognizedObjects에서 A를 꺼냅니다.
- 상호작용 우선순위를 정할 때는 상호작용할 액터들의 ScreenLocation을 구한 다음, 뷰포트의 중앙 점과의 거리를 비교하여 우선순위를 정하게 됩니다.

```

/* 틱마다 호출됩니다. */
void UIInteractComponent::UpdateTopInteractableActor()

```

```

{
    if(InteractableObjects.Num() == 0)
    {
        return;
    }

    float MinDistance = TNumericLimits<float>::Max();

    FTopInteractive Tmp;

    for (auto& Elem : InteractableObjects)
    {
        if(Elem.Value->GetIsActive() == false)
        {
            continue;
        }

        FVector2D ElemScreenLocation;
        if (Elem.Value->TryGetScreenLocation(ElemScreenLocation))
        {
            const float DistanceBetween = FVector2D::Distance(ScreenCenterCoord,
ElemScreenLocation);
            if (MinDistance > DistanceBetween)
            {
                MinDistance = DistanceBetween;
                Tmp.Interactive = Cast<IInteractable>(Elem.Value);
                Tmp.Key = Elem.Key;
            }
        }
    }

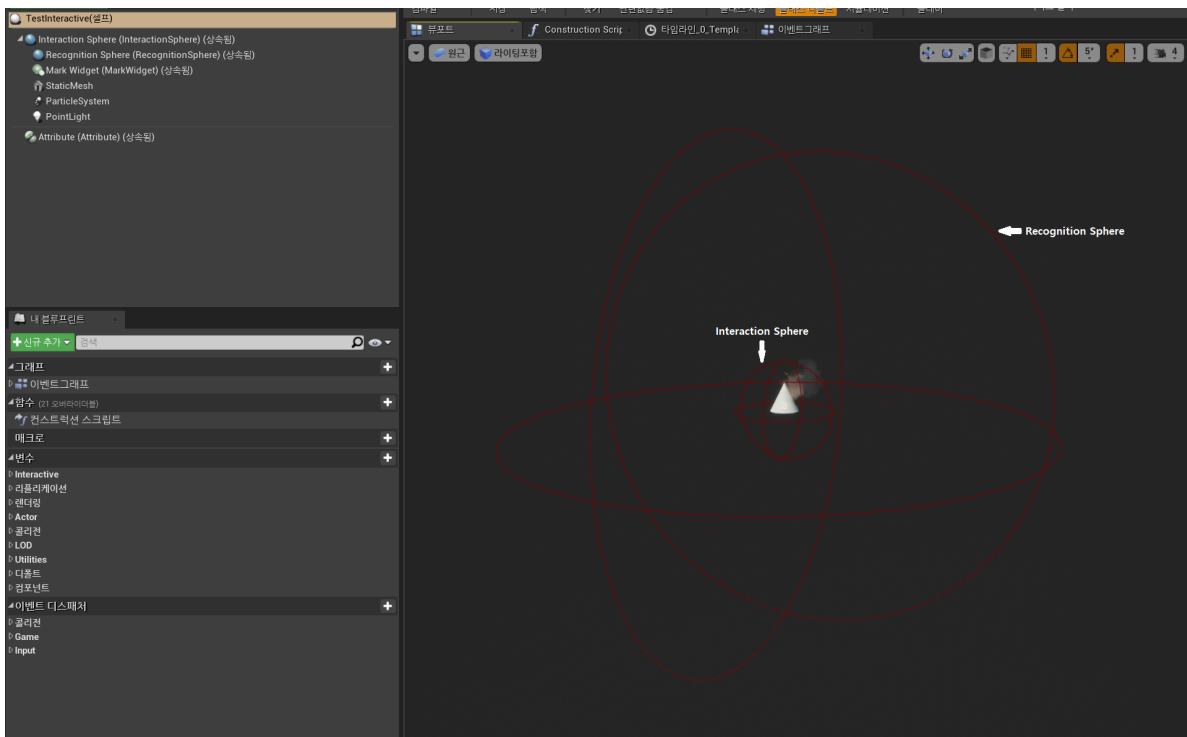
    if(TopInteractive.Interactive != nullptr)
    {
        TopInteractive.Interactive->ToggleInteractiveWidget(false);
    }

    if(Tmp.Interactive != nullptr)
    {
        Tmp.Interactive->ToggleInteractiveWidget(true);
        OwnerInteractHelperWidget->UpdateText(Tmp.Interactive-
>GetInteractText());
    }

    TopInteractive = Tmp;
}

```

ABaseInteractive



- 인식 범위와 상호작용 가능 범위는 ABaseInteractive.h에 정의해 두었습니다.

```
/* ABaseInteractive.h */

// 인식 범위
#define RECOGNITION_RANGE (800.f)

// ABaseInteractive 위에 띄워지는 원의 최대 높이
#define MAX.REACT_MARK_Z_LOCATION (270.f)

// 기본 Interaction Sphere의 반지름
#define DEFAULT_INTERACTION_SPHERE_RADIUS (32.f)

// 상호작용 가능 범위
#define INTERACTABLE_RANGE (100.f)
```

- Recognition Sphere와 Interaction Sphere의 반지름만 정해두면 되는거 아니냐고 물어보실 수 있는데 그렇게 한 이유는 물체의 크기는 가변적이기 때문입니다.
- 만약 Interaction Sphere의 반지름을 1미터로 정해두었는데 물체의 반지름이 1미터를 넘으면 어떻게 될까요?
- 이런 정의되지 않은 오류를 방지하기 위해 ABaseInteractive를 상속하는 클래스들은 Interaction Sphere의 반지름을 가변적으로 정하게 만들었습니다.

bAutoInteractionSphereRadius

- bAutoInteractionSphereRadius는 기본적으로 true이고, true면 OnConstruction()에서 Interaction Sphere의 반지름을 계산하여 정합니다.
- 반지름 계산 방법
 - 고려해야 할 것은 두가지 입니다.
 1. StaticMesh이든, SkeletalMesh이든 액터는 여러 Mesh들을 담고 있을 수 있다.
 2. 이 Mesh들은 반드시 액터 중앙에 위치한 것은 아니다.
 - 그렇기 때문에 Interaction Sphere의 반지름은 가장 먼 Mesh의 거리와 그 Mesh의 크기를 기준으로 상호작용 가능 범위를 더해서 정해집니다.

- Recognition Sphere의 반지름은 인식 가능 범위로 정해집니다.
- 예시
 - 가장 멀리 떨어져 있는 메시는 원이고 중앙에서 1미터 떨어져 있습니다. 이 원의 크기는 30cm입니다.
 - Interaction Sphere Radius = $100 + 30 + 100$ (상호작용 가능 범위)
 - Recognition Radius = 800(인식가능 범위)
 - Recognition Radius는 가변적이지 않은데 그 이유는 상호작용하는 물체가 너무 커지지 않길 원하기 때문입니다.
 - 하지만 이것은 정의되지 않은 오류가 생길 수 있기 때문에 개선해야할 사항이라고 보고 고쳐야 한다고 생각합니다.
- 단, Z축의 거리는 고려하지 않고 X, Y에서의 거리만 고려합니다.

```

/* ABaseInteractive.cpp */
void ABaseInteractive::OnConstruction(const FTransform& Transform)
{
    FVector MyOrigin = RootComponent->GetComponentLocation();

    const float OriginZLocation = MyOrigin.Z;
    float MarkWidgetZLocation = 0;

    FVector OriginXY = MyOrigin;
    OriginXY.Z = 0.f;
    float InteractionSphereRadius = DEFAULT_INTERACTION_SPHERE_RADIUS;

    // 컴포넌트를 모두 가져와서 그 컴포넌트가 UPrimitiveComponent이고 충돌이 켜져있으면
    // 그 컴포넌트의 정보는 Interaction Sphere Radius를 정하는데 사용됩니다.
    TArray<USceneComponent*> Components;
    RootComponent->GetChildrenComponents(true, Components);

    for (auto& Eelem : Components)
    {
        UPrimitiveComponent* CollisionComponent = Cast<UPrimitiveComponent>(Eelem);

        if (CollisionComponent && (CollisionComponent->GetCollisionEnabled() == ECollisionEnabled::Type::PhysicsOnly ||
            CollisionComponent->GetCollisionEnabled() == ECollisionEnabled::Type::QueryAndPhysics))
        {
            FVector ComponentLocation = CollisionComponent-
                >GetComponentLocation();
            FVector ComponentBoxExtent = CollisionComponent->Bounds.BoxExtent;

            const float TmpZ = ComponentLocation.Z - OriginZLocation +
ComponentBoxExtent.Z;
            if (TmpZ > MarkWidgetZLocation)
            {
                MarkWidgetZLocation = TmpZ;
            }

            if (!bAutoInteractionSphereRadius == false)
            {
                continue;
            }
        }
    }
}

```

```

        if (ComponentLocation.X - OriginXY.X < 0.f)
        {
            ComponentBoxExtent.X *= -1.f;
        }

        if (ComponentLocation.Y - OriginXY.Y < 0.f)
        {
            ComponentBoxExtent.Y *= -1.f;
        }

        FVector Point1 = FVector(ComponentLocation.X + ComponentBoxExtent.X,
        ComponentLocation.Y, 0.f);
        FVector Point2 = FVector(ComponentLocation.X, ComponentLocation.Y +
        ComponentBoxExtent.Y, 0.f);
        FVector Point3 = FVector(ComponentLocation.X + ComponentBoxExtent.X,
                                ComponentLocation.Y + ComponentBoxExtent.Y,
                                0.f);

        const float Distance1 = FVector::Distance(OriginXY, Point1);
        const float Distance2 = FVector::Distance(OriginXY, Point2);
        const float Distance3 = FVector::Distance(OriginXY, Point3);

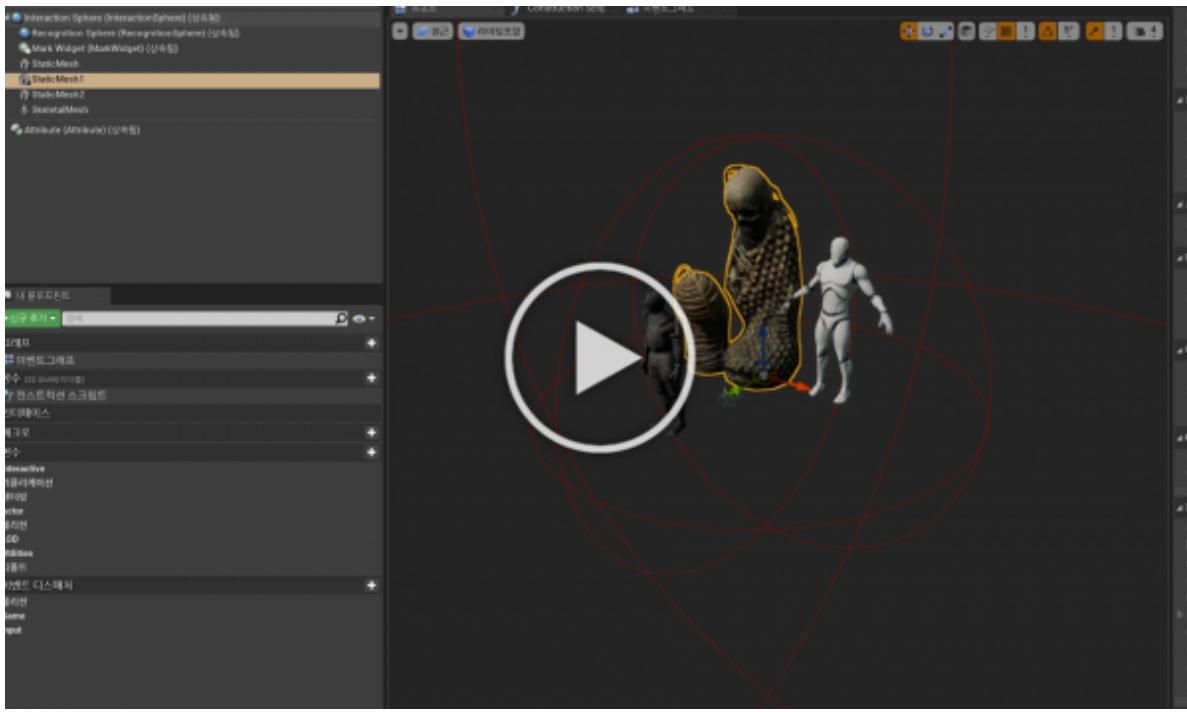
        const float TmpRadius = FMath::Max3(Distance1, Distance2,
        Distance3);
        if (TmpRadius > InteractionSphereRadius)
        {
            InteractionSphereRadius = TmpRadius;
        }
    }

    InteractionSphere->SetSphereRadius(InteractionSphereRadius +
INTERACTABLE_RANGE);
    RecognitionSphere->SetSphereRadius(RECOGNITION_RANGE);

    MarkWidgetZLocation = FMath::Clamp(MarkWidgetZLocation, 0.f,
MAX.REACT_MARK_Z_LOCATION);
    MarkWidget->SetRelativeLocation(FVector(0.f, 0.f, MarkWidgetZLocation));
}

```

- 결과물 영상(1분 내외)



- 물론 경우에 따라 bAutoInteractionSphereRadius를 끌 수 있습니다.
- 본 포트폴리오 메인 영상에서 본 자동문은 bAutoInteractionSphereRadius가 꺼져있습니다. 켜면 InteractionSphere의 크기가 너무 커지기 때문입니다.

UInteractAttributeComponent

- 이 컴포넌트는 상호작용할 때 캐릭터가 재생해야 할 몽타주, 소리, Effect를 가지고 있고 ABaseInteractive::OnInteract_Implementation(APawn*)에서 해당 애셋들을 사용합니다.

```
/* ABaseInteractive.cpp */
void ABaseInteractive::OnInteract_Implementation(APawn* PlayerPawn)
{
    // GenerateInteractionEffects에서 몽타주, 소리, ParticleSystem을 재생합니다.
    Attribute->GenerateInteractionEffects(PlayerPawn);
}
```

쉬운 사용성

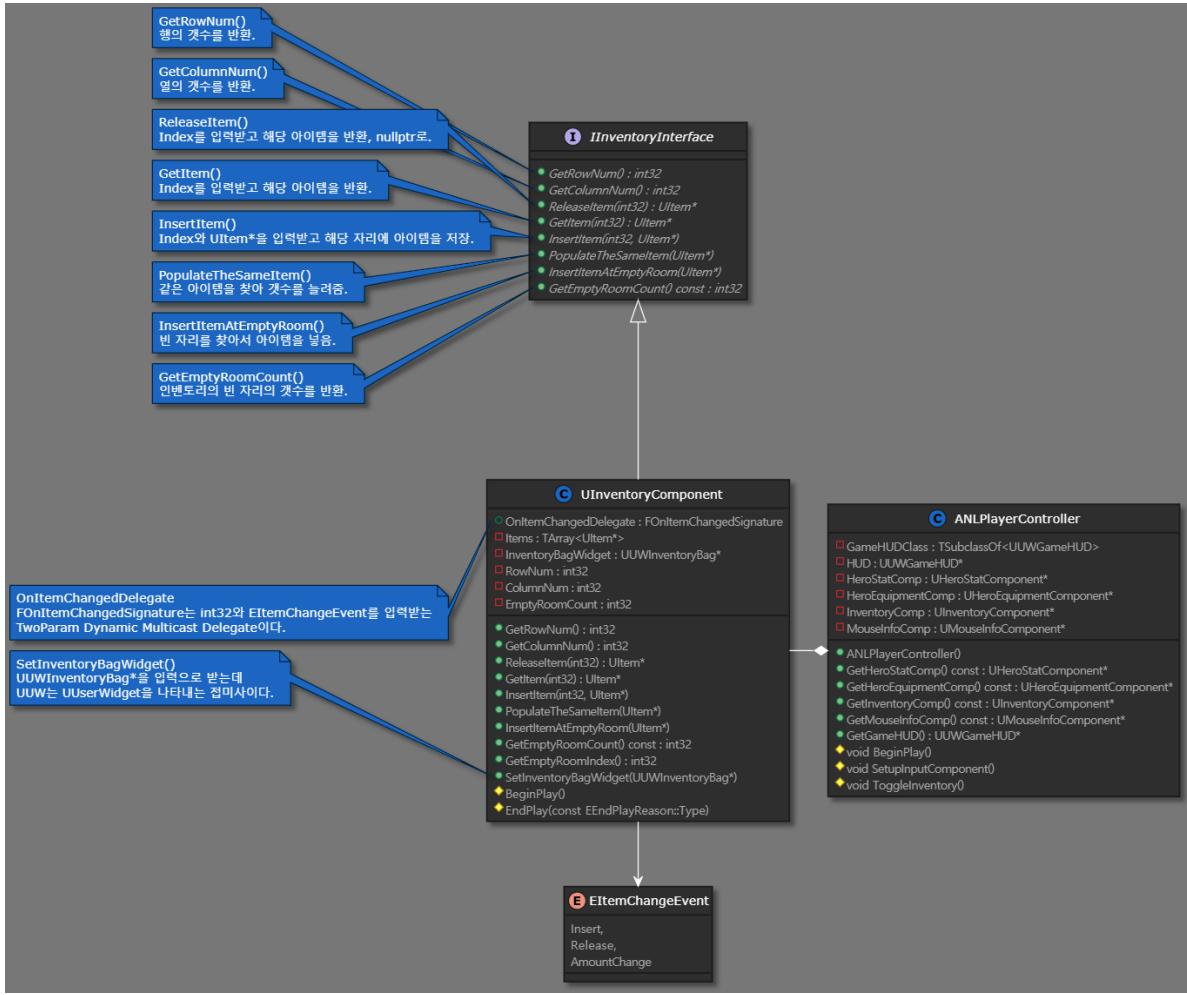
- BaseInteractive를 만들어 두었으니 상호작용 로직은 ABaseInteractive::OnInteract_Implementation(APawn*)을 override하여 정의합니다.
- 굳이 cpp로 작성할 필요가 없고 빠른 로직 작성은 블루프린트가 유용하게 사용될 수 있습니다.
- 본 포트폴리오 영상 데모의 자동문과 꼬깔콘의 로직은 블루프린트로 작성되었습니다.

참고 페이지

InventorySystem

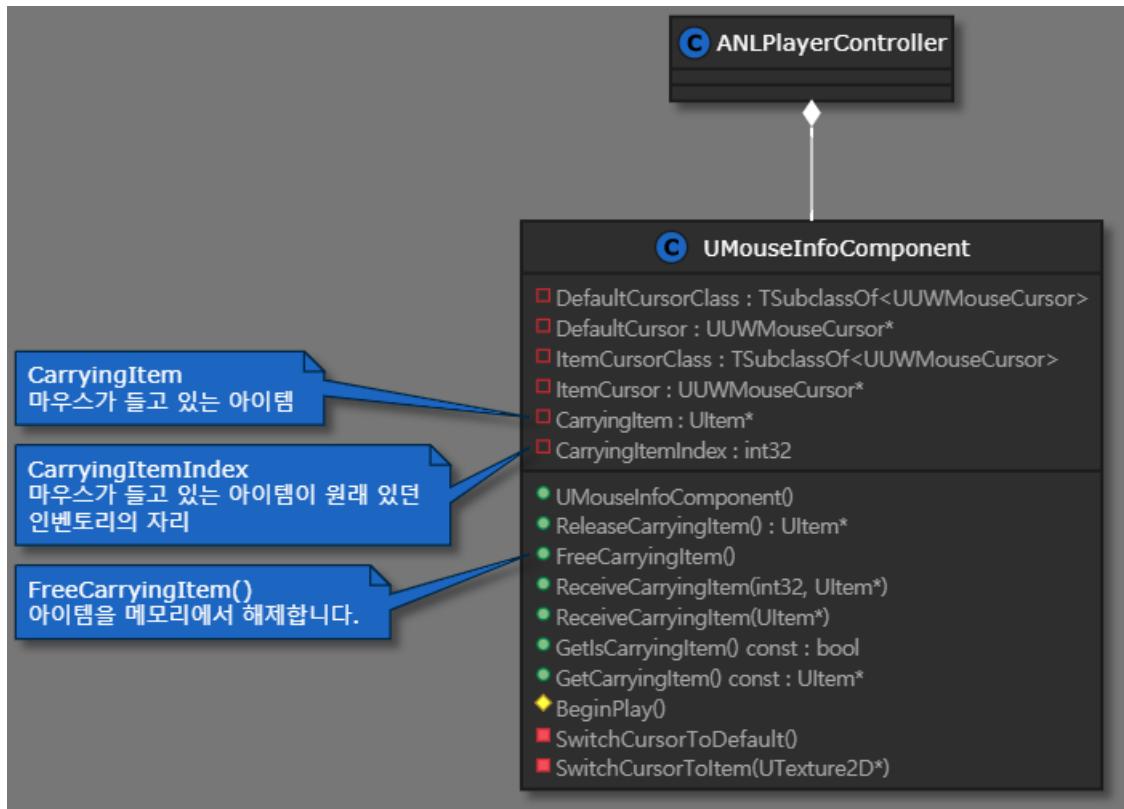
ANLPlayerController과 UInventoryComponent

- 인벤토리는 ANLPlayerController를 구성하는 컴포넌트입니다.
- 인벤토리를 살펴보기 이전에 ANLPlayerController를 살펴볼 필요가 있습니다.
- ANLPlayerController는 APlayerController를 상속합니다.



- 인벤토리에서의 주 기능은 Insert와 Release입니다.
- ReleaseItem(int32)는 아이템의 메모리를 해제하진 않고 해당 자리의 아이템을 nullptr로 만들면서 되돌려 줍니다.
- 아이템을 넣고 뺀 일은 위젯에서 담당합니다.
- OnItemChangedDelegate는 Insert하거나 Release할 때 Broadcast합니다.
- 몇번 Index의 어떤 EItemChangeEvent인지 확인 후, UI를 업데이트합니다.

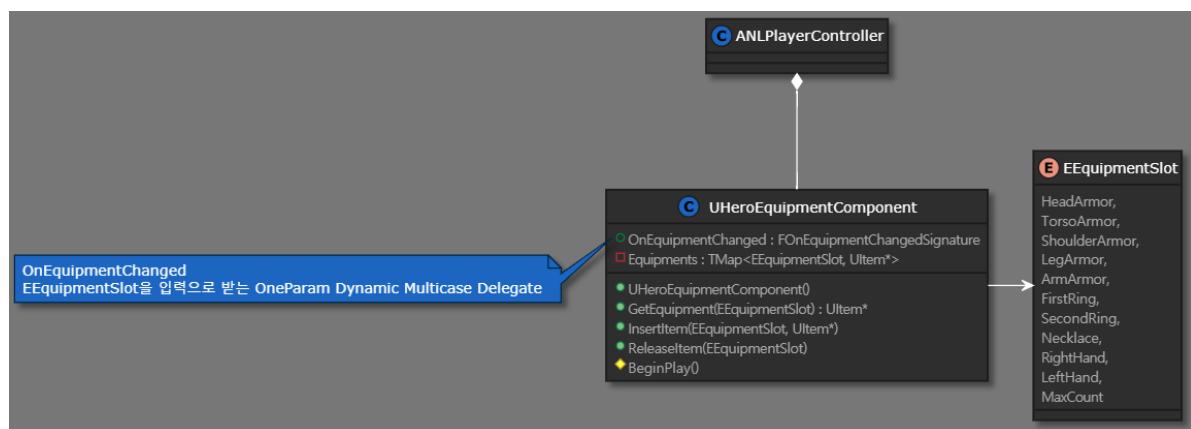
UMouseInfoComponent



- 마우스로 아이템을 클릭하면 마우스 아이콘이 아이템 아이콘으로 바뀌면서 아이템을 UIInventoryComponent에서 Release합니다.
- Release한 아이템은 ANLPlayerController::MouseInfoComp가 가지게 됩니다.
- 마우스 아이콘이 아이템 아이콘일 때, 즉 MouseInfoComp가 어떤 아이템을 들고 있을 때 인벤토리에 넣으면 빈공간이 채워지거나 해당 자리의 아이템과 스왑됩니다.
- 아이템을 메모리에서 해제하는 것은 오직 아이템을 마우스가 들고 있을 때만 가능합니다.

UHeroEquipmentComponent

- ANLPlayerController를 구성합니다.
- 캐릭터가 착용한 장비들을 저장합니다.



- Equipments는 EquipmentSlot에 따른 아이템을 갖는 컨테이너입니다.
- 마우스가 아이템을 들고 있을 때 캐릭터 장비창의 알맞는 슬롯에 클릭하면 Equipments에 아이템이 들어갑니다.

```

/* UHeroEquipmentComponent.cpp */
UHeroEquipmentComponent::UHeroEquipmentComponent()
{
}

```

```

PrimaryComponentTick.bCanEverTick = false;

// Initialize Equipments...
for (int i = 0; i < static_cast<int32>(EEquipmentslot::MaxCount); ++i)
{
    Equipments.Emplace(static_cast<EEquipmentslot>(i), nullptr);
}

void UHeroEquipmentComponent::InsertItem(EEquipmentslot EquipmentslotType,
UItem* Item)
{
    // Equipments에 아이템을 실질적으로 넣는 코드입니다.
    // 넣으려는 아이템과 장비 슬롯이 일치하는지는 InsertItem이 호출되기 전 검사합니다.
    Equipments.Add(EquipmentslotType, Item);

    // ... 생략된 코드는 UHeroStatComponent에 스탯들을 업데이트하는 작업입니다.

    if(OnEquipmentChanged.IsBound())
    {
        OnEquipmentChanged.Broadcast(EquipmentslotType);
    }
}

```

UHeroStatComponent

- 캐릭터의 스탯을 담당합니다.
- float 스탯
 - 경험치, 최대 경험치, 방어력, 최소 데미지, 최대 데미지, 최대 체력, 체력, 분당 체력 회복률, 최대 마나, 마나, 분당 마나 회복률
- int32 스탯
 - 레벨, 사용 가능한 스탯 포인트, 힘, 민첩, 활력, 지능
- 스탯이 워낙 많고 그에 따라 델리게이트도 많기 때문에 그림이 복잡할 거 같아 Diagram을 생략합니다.
- 하지만 이해하기 쉽게 간단한 스탯적용 예시 코드를 첨부합니다.

```

/* UHeroEquipmentComponent.cpp */

// 스탯을 적용하려면 당연히 아이템을 장비에 넣는 것 부터 하게 됩니다.
void UHeroEquipmentComponent::InsertItem(EEquipmentslot EquipmentslotType,
UItem* Item)
{
    // ...

    ANLPlayerController* PlayerController = Cast<ANLPlayerController>
(GetOwner());
    check(PlayerController);

    UHeroStatComponent* HeroStatComp = PlayerController->GetHeroStatComp();
    check(HeroStatComp);

    // UItem은 UItemData를 가지고 있고 UEquipmentData는 UItemData를 상속합니다.
    // UEquipmentData는 아이템의 스탯 보너스 정보를 담고 있습니다.

```

```

// 아래 ItemData를 설명할 때 다시 나옵니다.
UEquipmentData* EquipmentData = Cast<UEquipmentData>
(*Equipment.Find(EquipmentSlotType))->GetItemData();
HeroStatComp->ApplyEquipEffect(EEquipType::Equip, EquipmentData);

// ...
}

/* UHeroStatComponent.cpp */
void UHeroStatComponent::ApplyEquipEffect(EEquipType EquipType, UEquipmentData* EquipmentData)
{
    const int32 Coefficient = EquipType == EEquipmentType::Equip ? 1 : -1;
    // Coefficient는 일종의 계수입니다.
    // Coefficient는 스텟 정보들과 곱하여 더하거나 빼게 해줍니다.
    // EquipType은 Equip과 UnEquip이 있습니다.
    // EquipType이 Equip이면 더해야 하기 때문에 1이고
    // EquipType이 UnEquip이면 도로 빼야 -1인 것입니다.

    for (const auto Effect : EquipmentData->GetEquipmentEffects())
    {
        // EquipmentEffects는 Equipment의 스텟 보너스 정보들을 담고 있습니다.
        // 아래의 ItemData를 설명할 때 나오는데 간략하게 설명하면
        // EquipmentData는 ItemData를 상속하고 PrimaryStat(힘, 민첩, 활력, 지능)에 따른 값을 저장하는
        // TMap 컨테이너인 EquipmentEffects를 들고 있습니다.
        // 즉 각 EquipmentEffect는 Key로 PrimaryStat, Value로 int32를 가집니다.
        AddOrSubStatByType(Effect.Key, Coefficient * Effect.Value);
    }
}

void UHeroStatComponent::AddOrSubStatByType(EPrimaryStat PrimaryStatType, int32 Value)
{
    switch (PrimaryStatType)
    {
    case EPrimaryStat::Strength:
        AddOrSubStrength(Value);
        break;
    case EPrimaryStat::Dexterity:
        AddOrSubDexterity(Value);
        break;
    case EPrimaryStat::Vitality:
        AddOrSubVitality(Value);
        break;
    case EPrimaryStat::Intelligence:
        AddOrSubIntelligence(Value);
        break;
    default:
        checkNoEntry();
    }
}

void UHeroStatComponent::AddOrSubStrength(int32 value)
{
    Strength += value;

    AddOrSubMinDamage(value * 2.f);
}

```

```
AddOrSubMaxDamage(Value * 2.f);

if(OnStrengthChanged.IsBound())
{
    OnStrengthChanged.Broadcast(Strength);
}

}
```

참고 페이지

Item

- 아이템의 구조는 오랜기간 고민한 부분입니다.
- 아이템은 객체의 실체적 데이터(갯수, 내구도)와 고유 정보(스탯 보너스, 무기 공격력, 방어구 방어도 등)이 있습니다.
- 고유정보같은 경우에는 굳이 똑같은 데이터가 여러개 있을 필요가 없으므로 UIItemData라는 클래스를 구체화하기로 하고, 실체적 데이터는 실존하는 아이템마다 있어야 하니 UIItem이라는 클래스를 만들기로 결정하였습니다.
- 그러면 결국엔 UIItem은 UIItemData를 가지고 스탯정보를 수정할 때는 UIItemData를 참조하여 스탯을 수정하게 됩니다.
- 또, 아이템에는 어떤 능력이 있을 수 있는데 이런 능력을 UNLAbility라는 클래스를 만들어서 구체화하는 쪽으로 정하고 UIItemData가 UNLAbility를 참조하여 아이템을 사용할 때 UNLAbility를 참조하여 사용하기로 결정했습니다.
- 그래서 대략적인 구조는 다음 그림과 같습니다.

UIItem

갯수, 내구도 등

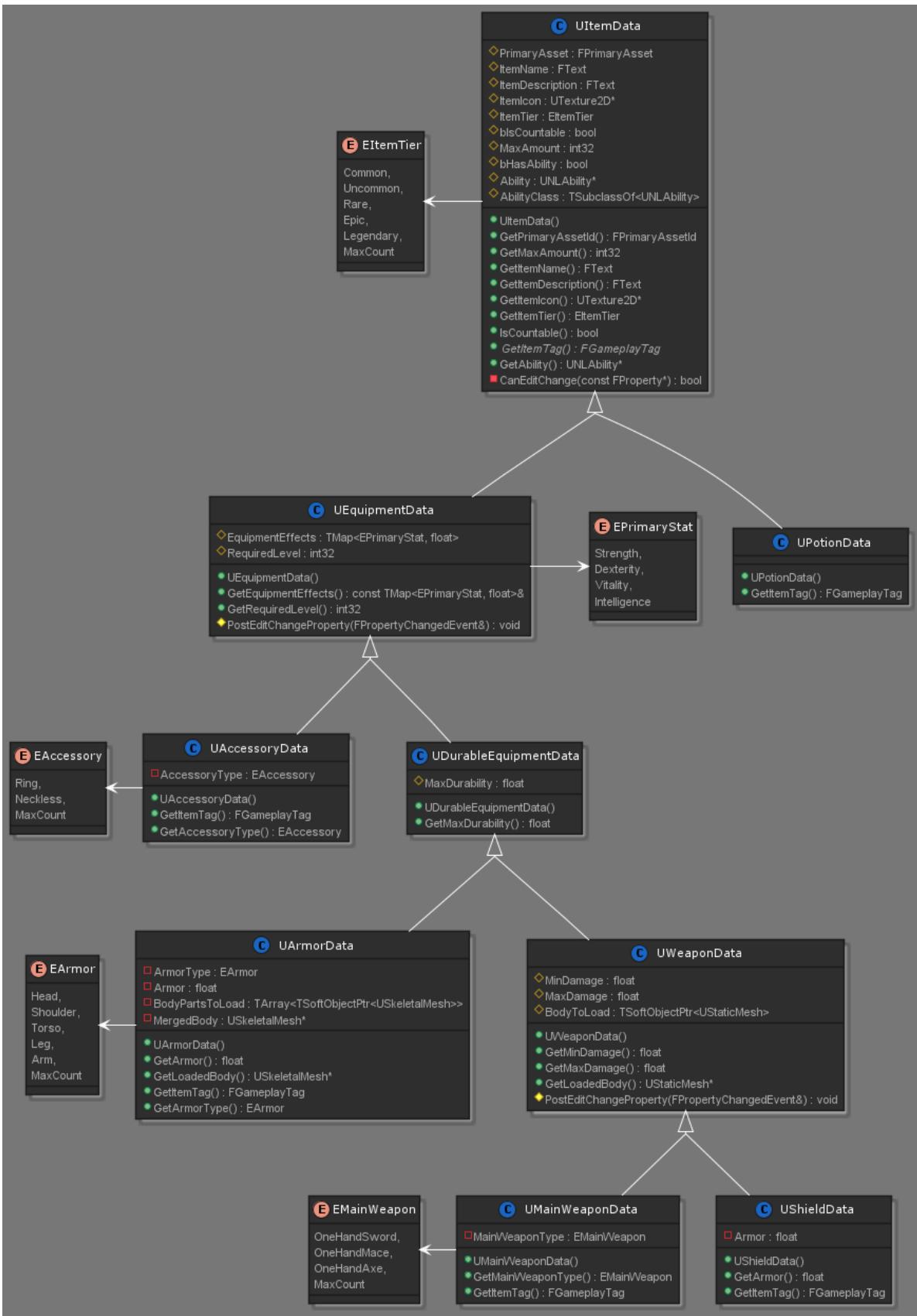
UIItemData

아이템 이름, 아이템 설명, 아이콘, Mesh, GameplayTag 등

UNLAbility

TemporaryBuff, PermanentBuff, Recovery 등

UIItemData



UItemData의 특징

- **UItemData**는 **UPrimaryDataAsset**를 상속합니다.
- 아이템 데이터를 만들 때 저에겐 두가지 선택지가 있었습니다.
 - 첫째는 테이블에 모든 데이터를 넣어서 처리할 것인가
 - 두번째는 하나하나 데이터를 만들어서 그 데이터를 사용하는 식으로 하느냐
 - 첫번째 방법은 테이블에서 문자열 키로 검색해서 맞는 아이템데이터를 찾아야 하기 때문에 아이템 데이터를 가지고 있어야할 어떤 액터(AltentContainer)가 문자열 키를 들고 있어야 한다

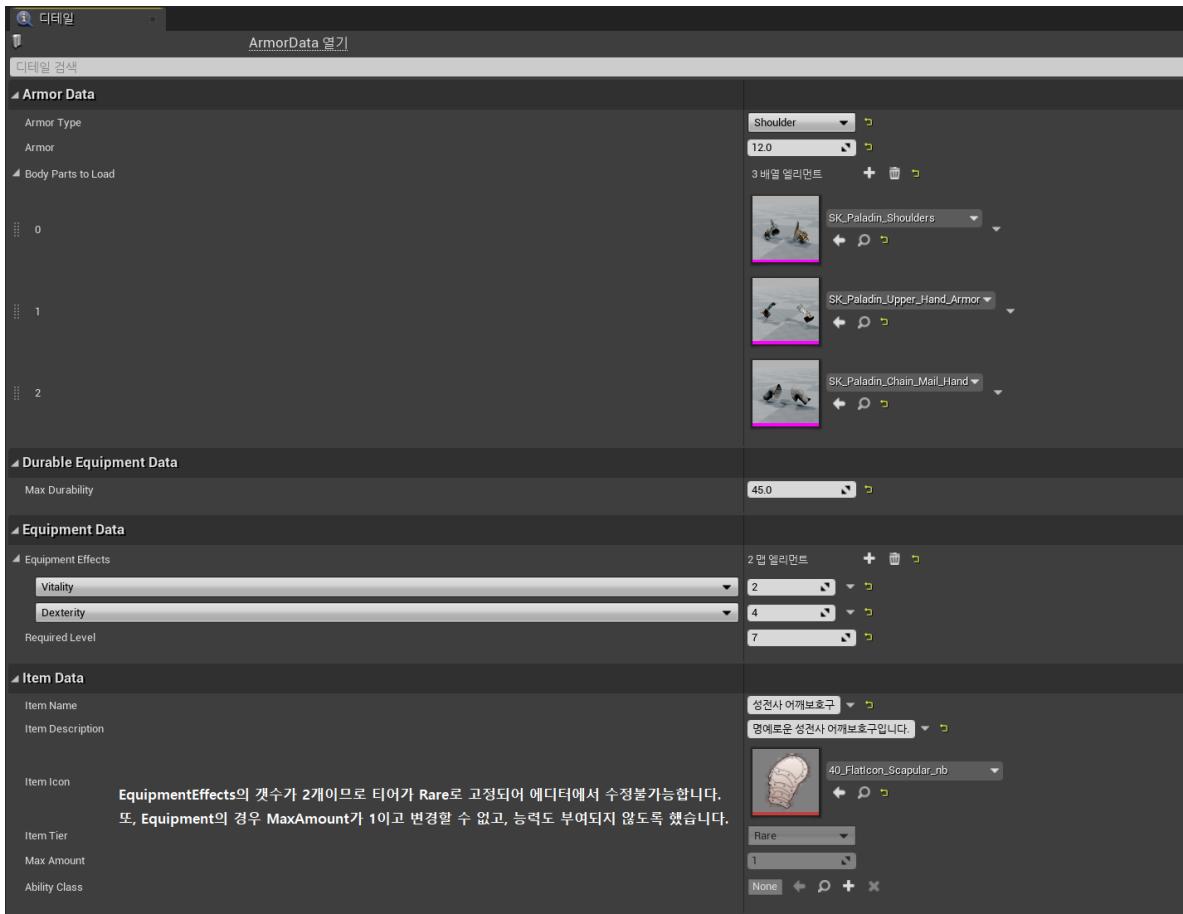
는 점이 문제가 되었습니다.

- 왜 문제가 되냐, 이걸 사용하는 디자이너 입장에서 생각해봤을 때 드롭박스에서 검색해서 선택하는 것이 아닌 그냥 문자열을 입력하게될텐데 문자열을 틀리게 입력했을 때 어떻게 처리해야 하느냐가 문제였습니다.
 - 두번째 방법도 문제가 없는 것은 아니었습니다. 일단 하나하나 아이템을 생성해서 입력해야 할텐데 csv 테이블을 사용하는 것보다 사용 편의성과 효율성이 떨어진다는 느낌이었습니다.
 - 저는 아이템 종류가 많지 않기 때문에 두번째를 선택했는데, 결국엔 첫번째와 두번째 방법을 섞는게 가장 좋은 방법인 것 같습니다.
 - 그 방법은 일단 아이템 데이터를 테이블로 만들어둔 다음, 이 정보를 읽어서 엔진 런타임에서 데이터를 만들거나 수정하는 방법이 가장 나은 것 같습니다.
 - 아무튼 두번째 방법을 선택했으니 저는 UAssetManager를 이용해서 비동기로드를 할 것입니다.
- 지금 당장 존재하는 아이템은 포션과 장비뿐이므로 UItemData를 상속하는건 UEquipmentData 와 UPotionData 뿐입니다.
 - 그리고 UEquipmentData를 상속하는 여러 장비아이템 데이터가 있음을 확인하실 수 있습니다.
 - UEquipmentData의 멤버 변수들 중에서는 디자이너가 직접 변경할 수 없는 값도 있는데 아이템의 티어가 그것입니다.
 - 이런식으로 데이터를 수정하는 데 있어서 사용하는 사람이 실수하지 않게 하기위한 장치를 필요한 ItemData에 해두었습니다.
 - 아래는 UEquipmentData의 예입니다.

```
/* UEquipmentData.cpp */

void UEquipmentData::PostEditChangeProperty(FPropertyChangedEvent& PropertyChangedEvent)
{
    Super::PostEditChangeProperty(PropertyChangedEvent);

    // EquipmentEffects의 숫자에 따라 아이템티어가 정해집니다.
    if(PropertyChangedEvent.GetPropertyName() == FName("EquipmentEffects"))
    {
        if (EquipmentEffects.Num() < 1)
        {
            ItemTier = EItemTier::Common;
        }
        else if (EquipmentEffects.Num() < 2)
        {
            ItemTier = EItemTier::Uncommon;
        }
        else if (EquipmentEffects.Num() < 3)
        {
            ItemTier = EItemTier::Rare;
        }
        else if (EquipmentEffects.Num() < 4)
        {
            ItemTier = EItemTier::Epic;
        }
        else
        {
            ItemTier = EItemTier::Legendary;
        }
    }
}
```



- UDurableEquipmentData는 게임에서 보이는 StaticMesh나 SkeletalMesh를 가지고 있습니다.
 - UArmorData는 BodyPartsToLoad가 TSoftObjectPtr<USkeletalMesh>의 배열인데, 이런 Secondary Asset들은 Secondary Asset들의 나머지(보통은 UI 관련 Asset)를 먼저 비동기로 드하고 비동기로드가 완료되면 다시 Secondary Asset들을 비동기로드 하는식으로 했습니다.
 - 이러한 비동기 로드는 UGameInstanceSubsystem을 상속하는 UItemManagerSubsystem에서하게 됩니다.

UItemManagerSubsystem

C UItemManagerSubsystem

- ItemColorTable : UDataTable*
- Initialize(FSubsystemCollectionBase&)
- Deinitialize()
- GetItemColor(UItemData*) : FLinearColor
- CreateItemWithAsyncLoad(FPrimaryAssetId, UItem**, UObject*)
- ForceLoadAndCreateItem(FPrimaryAssetId, UItem**, UObject*)
- AsyncLoadSecondaryAssets(UItemData*)
- OnPrimaryItemAssetLoadCompleted(FPrimaryAssetId, UItem**, UObject*)
- CreateItem(UItemData*, UObject*) : UItem*

- ItemManagerSubsystem은 UGameInstanceSubsystem을 상속하여 GameInstance와 수명을 함께 합니다.
- 저는 싱글턴을 만들어야 했을 때는 이런 프로그래밍 서브시스템을 사용했습니다.
- 프로그래밍 서브시스템은 Engine, Editor, GameInstance, LocalPlayer, World의 서브시스템이 있는데 이런 서브시스템들은 수명관리가 쉽고 뭔가 언리얼 다워서(...) 주로 사용했습니다.

- ItemManagerSubsystem은 UItemData를 사용해서 UItem을 생성하는 핵심적인 역할을 담당합니다.
- 그리고 UItem을 생성하면서 비동기로드를 하게 해줍니다.
- 아이템 획득 루틴
 1. ABaseInteractive를 상속하는 AltemContainer는 UItemData를 가리키는 PrimaryAssetID를 들고 있습니다.
 2. 캐릭터가 AltemContainer의 Recognition Sphere안에 들어오게 되면 아이템을 얻으려는 의도가 있다고 간주, PrimaryAssetID들을 이용해서 UItemData를 비동기로드하도록 UItemManagerSubsystem에 요청하게 됩니다.
 3. UItemManagerSubsystem은 일차적으로 필요한 애셋들(UI 애셋)을 먼저 비동기로드한 후, 로드된 UItemData들을 이용해서 UItem을 생성, AltemContainer에 돌려줍니다.
 4. 또, UItemManagerSubsystem은 UI 애셋들의 비동기로드가 끝나면 세컨더리 애셋들 (SkeletalMesh, StaticMesh)들을 비동기로드합니다.
 5. 캐릭터가 AltemContainer와 상호작용하면 생성된 UItem*들을 UInventoryComponent에 삽입합니다.
 6. 이때 아직까지도 UItem이 생성되지 않았다면 동기로드하여 생성하도록 UItemManagerSubsystem에 요청합니다.

```

/* ItemContainer.cpp */

void AItemContainer::OnInteract_Implementation(APawn* PlayerPawn)
{
    Super::OnInteract_Implementation(PlayerPawn);

    // ...

    for (int32 i = ItemAssetIds.Num() - 1; i >= 0; --i)
    {
        if (ItemAssetIds[i].IsValid())
        {
            if (Items[i] == nullptr)
            {
                // 어떤 Items가 nullptr이면 아직 비동기로드 및 생성이 안되어 있다는 뜻이
                //므로
                // ItemManager에게 동기로드하도록 요청합니다.
                ItemManager->ForceLoadAndCreateItem(ItemAssetIds[i], &Items[i],
                this);
                check(Items[i]);
            }

            Inventory->PopulateTheSameItem(Items[i]);
        }
    }

    // PopulateTheSameItem은 같은 아이템이 인벤토리에 있는 경우 갯수를 늘려주는
    함수인데
    // 이 함수에서는 입력으로 들어가는 Item의 갯수가 성공적으로 더해지면 더해진 만
    큼 갯수를 줄입니다.
    // 만약 성공적으로 모든 아이템의 갯수가 더해지면 입력으로 들어간 Item의 갯수는
    0개가 되므로
    // 그 아이템은 메모리에서 해제합니다.
    if (Items[i]->GetAmount() == 0)
    {
        Items[i]->ConditionalBeginDestroy();
        Items[i] = nullptr;
    }
}

```

```

        else
        {
            if(Inventory->GetEmptyRoomCount() > 0)
            {
                Inventory->InsertItemAtEmptyRoom(Items[i]);
                Items[i] = nullptr;
            }
        }

        // ...
    }

// ...
}

```

참고 페이지

- Interaction Inventory System : Unreal Engine Market Place
 - <https://www.unrealengine.com/marketplace/en-US/product/interaction-inventory-system>
 - 인벤토리 UI를 어떤식으로 디자인, 구성해야 하는지 도저히 참고할 레퍼런스가 없어서 마켓플레이서에서 구입해서 UI 디자인만 참고하였습니다.
- Adventurer's Inventory Kit : Unreal Engine Market Place
 - <https://www.unrealengine.com/marketplace/ko/product/adventurer-s-inventory-kit>
 - 작업 초기에 원래 UI를 어쌔신 크리드 비슷하게 해보려고 구입해서 살펴보았으나 둘러봐도 뭐가 뭔지 하나도 몰랐기 때문에 위의 프로젝트를 먼저 참고하는 쪽으로 방법을 선회했습니다.
 - 지금까지 해보니 어쌔신 크리드 UI를 구성하는데 어느정도 감은 잡하고 지금은 불가능은 아니라는 생각이 듭니다.
- Learning to Make Games with UE4 and Action RPG | GDC 2019 | Unreal Engine
 - <https://www.youtube.com/watch?v=x5KqpYX4H6g>
 - 액션 RPG 프로젝트는 본 프로젝트를 하면서 굉장히 많은 도움이 되었습니다.
 - GameplayTag의 이해, AssetManager를 이용한 비동기 로드의 이해 등 정말 많은 것을 배웠습니다.
- 프로그래밍 서브시스템
 - <https://docs.unrealengine.com/4.27/ko/ProgrammingAndScripting/Subsystems/>
- Unreal-style Singletons with Subsystems
 - <https://benui.ca/unreal/subsystem-singleton/>
- 비동기 애셋 로딩
 - [https://docs.unrealengine.com/4.27/ko/ProgrammingAndScripting/ProgrammingWithC++/Assets/AsyncLoading/](https://docs.unrealengine.com/4.27/ko/ProgrammingAndScripting/ProgrammingWithCPP/Assets/AsyncLoading/)
- C++ - Runtime Async Load Modular Character (Intermediate)
 - <https://forums.unrealengine.com/t/tutorial-c-runtime-async-load-modular-character-intermediate/4160/18>
 - 14년에 포스팅된 튜토리얼이어서 지금 사용하는 코드와는 많이 달랐지만 Modular Character 를 구성함에 있어서 비동기로드를 어떻게 해야하는지 많은 도움이 되었습니다.
- 프로퍼티 지정자
 - <https://docs.unrealengine.com/4.26/ko/ProgrammingAndScripting/GameplayArchitecture/Properties/Specifiers/>
 - 아이템 데이터를 구성할 때 여러 강제해야할 점이 많았는데 이 문서를 참고했습니다.

본 프로젝트를 진행하면서 느낀점

1. 기획자의 필요성

- 기획자가 없으니까 고민하는 시간이 너무 많아지고 삽질하는 횟수가 너무 많아져서 기획자가 정말 절실히 필요하다는 걸 느꼈습니다. 특히 본 프로젝트는 플레이어는 어크 비슷하게 하고, 인벤토리, 아이템은 WOW를 따라하고 스탯 시스템은 디아블로를 따라했는데 UI를 끄고 키다보니 아 이거 큰일 났구나 싶었습니다만... 일단 이 정도로 마무리 지었고 왜 게임개발에 기획자가 꼭 필요한지 절실히 느끼게 되었습니다.

2. 게임 개발의 경험

- 게임 개발의 경험이 아예 없다보니 어떤 시스템을 구성할 때 뭘 참고해야 하고, 구성은 또 어떻게 할지, 이게 진짜 맞는 길인지 수없이 스스로에게 묻고 또 묻는 경험을 했습니다. 경험이 너무 적다보니 경험의 중요성이 정말 중요하다는 걸 느꼈습니다.
- 또 어떤 시스템이 필요할 때마다 그때가서 또 만들고 또 만들고 하다보니 뭔가 짬뽕이 되고 시스템이 결국엔 정갈하지 못하다는 느낌을 많이 받았습니다.
- 좀더 게임 프로그래밍 시스템에 대해 직관이 있었다면 더 수월하게 할 수 있지 않을까 느꼈습니다.

3. 기존의 습관이 매우 잘못되었다는 것을 깨달음

- 저는 원래 완벽주의 성격이라고 해야 할까, 머리속에서 대강의 구조가 잡혀야 그제서 일을 시작하는 버릇이 있었습니다. 그런데 아이템 데이터 구조를 만들면서 너무 고민하는 시간이 길어지니까 이것이 너무 큰 단점으로 생각되어서 일단 만들고 수정하고 또 수정하는 식으로 바꿨습니다. 프로그래밍에서의 가장 안 좋은 습관이 완벽주의 습관이라고 생각되고 이번 프로젝트를 하면서 좀 그 습관을 좀 깨나갔다 스스로 자평하고 싶습니다.

4. 써드파티 프로그램 혹은 플러그인의 필요성

- 아무래도 엔진을 좀 더 효율적으로 활용하기 위해서는 써드파티 프로그램의 개발 필요성을 느끼게 되었습니다.
 - 데이터 테이블의 관리같은 경우에 엑셀을 이용하는게 엔진을 이용하는 것보다 더 편리한데
 - 테이블을 관리(테이블의 생성 및 삭제 등)을 할 때는 엔진을 통하는 것보다 C# 비주얼 프로그래밍을 이용한 써드파티 프로그램이 더 적격이라고 생각됩니다.
- 또 다른 프로젝트이지만서도 같은 동작이 필요할 때 플러그인을 사용하는게 편리할 것 같은 느낌을 받았습니다.
 - 예시로는 Foot IK Placement를 예로 들 수 있을 것 같습니다.
 - 그리고 보통 블루프린트 클래스나 오브젝트를 네이티브 코드에서 참조하려면 하드코딩해왔습니다. 아무래도 참고할 책에서는 직접 레퍼런스 복사하고 코드에 갖다 붙이고 이런 과정을 거쳤고 그걸 따라하다보니 그랬는데, 블루프린트 클래스나 오브젝트를 관리할 매니저 플러그인을 사용하면 어떨까, 결국엔 위치가 바뀔 수 있기 때문에 하드코딩은 너무 불편하다 이런 느낌을 받았습니다.

5. 변수 및 함수 이름은 코드 가독성에 매우 중요

- 영어에 익숙치 않기 때문에 변수나 함수이름을 지었다가 고친적이 100번은 넘은 듯 합니다. 이게 맞나 저게 맞나, 어떻게 해야 더 읽기 편할까 고민을 많이 했는데 헛되지 않은 고민이었던 것 같고 영어 실력이 미래다라는 걸 깨달았습니다.

마무리

- 이 프로젝트는 계속해서 업데이트될 예정입니다.
- 다음 시스템은 퀘스트 시스템입니다.
- 지금까지 읽어주셔서 감사하고 오늘도 좋은 하루 보내셨으면 좋겠습니다.