

# Sicarius-Portfolio

Video Link : <https://youtu.be/z4qN8cH2Qo4> 프로젝트 링크 : <https://github.com/Naezan/Sicarius>

## • 프로젝트 정보

플랫폼 : 윈도우  
엔진 : 언리얼 엔진5  
개발 기간 : 4개월  
개발 인원 : 1명

## • 게임 설명

Sicarius는 저의 세번째 프로젝트이자 언리얼 엔진을 사용한 두번째 프로젝트입니다.  
이 게임은 1~3인의 멀티 플레이 게임으로 다양한 암살 스킬을 통해 적들을 처치해야하는 암살자로서의 사명을 가지고 있습니다.  
타겟이 있는 마을로 들어간 당신은 총 3명의 타겟을 처치하는 임무를 완수해야만합니다.

## 구현목록

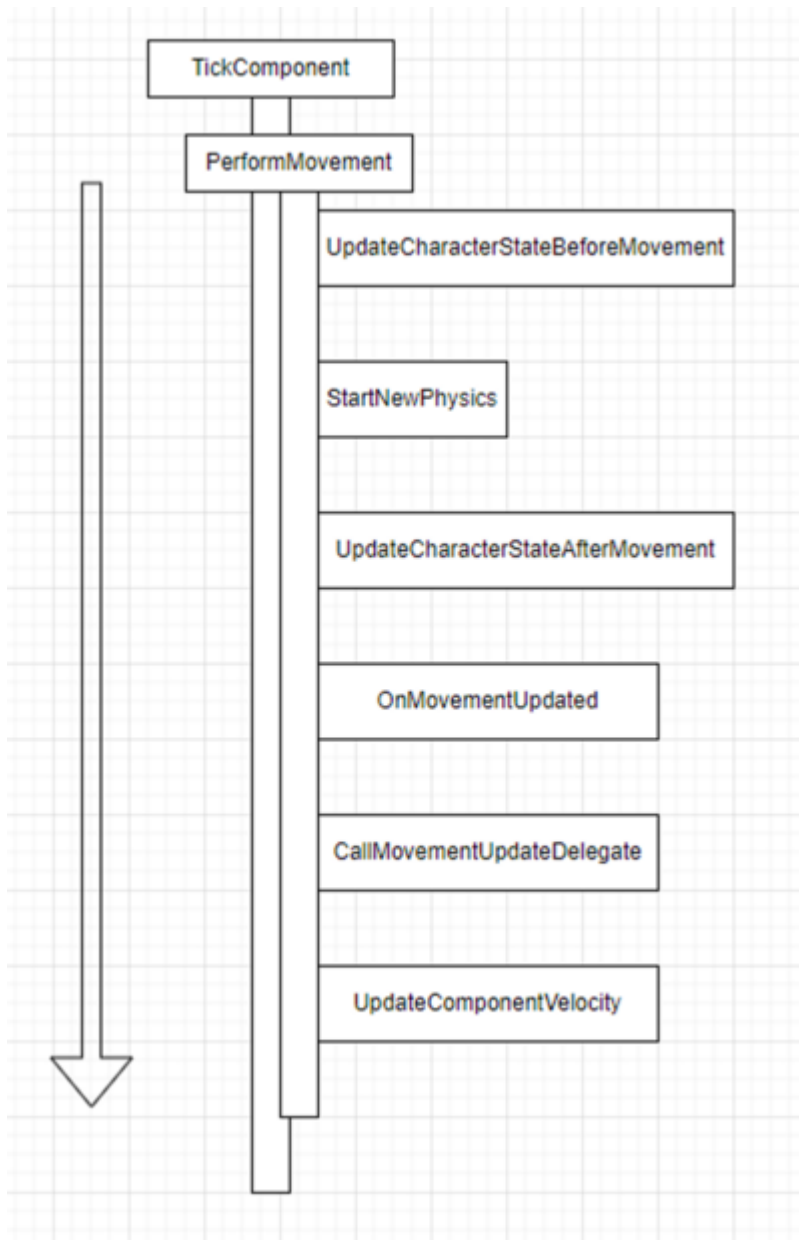
1. CharacterMovementComponent
2. AbilitySystemComponent
  - 2.3 던지기 어빌리티
  - 2.4 공격 어빌리티와 후방 암살(뒷잡)
  - 2.5 상호작용 어빌리티
  - 2.6 점프암살 어빌리티
  - 2.7 데미지 상호작용
3. InventorySystem
4. AI
  - 6.5 AIPerception Age
  - 6.6 SmartObject 탐색
5. OSS Steam
  - 7.1 멀티 세션의 구조 설계
  - 7.2 로비와 게임의 게임모드 분리
6. Spectator
  - 8.1 관전 입력
  - 8.2 관전자 전환
  - 8.3 리스폰
7. Extra : LineTracing Attack

## CharacterMovementComponent

제 게임에는 클라이밍(파쿠르) 요소를 구현하기 위해 `CharacterMovementComponent`의 `MovementMode`를 이용했습니다.

그다음 CMC에서 `virtual void PhysCustom(float deltaTime, int32 Iterations);`를 재정의하여 파쿠르 전용 함수를 통해 움직임을 처리하였습니다.

`CharacterMovementComponent`은 `PerformMovement`에서 움직임과 관련된 메인 로직이 수행되는데, 아래와 같은 호출 순서를 가지게 됩니다.



#### CMC 호출흐름

이때 `StartNewPhysics`에서 **MovementMode**에 따른 로직을 수행하는데, 호출 시점을 통일하기 위해 모든 **MovementMode**로직을 `StartNewPhysics`이전에 호출되는 `UpdateCharacterStateBeforeMovement`에서 수행하도록 변경하였습니다. 그 결과 서버와 동기화 과정에서 문제가 발생하지 않도록 구조로 설계할 수 있었습니다.

[↑ Back to Top](#)

## AbilitySystemComponent

## 던지기 어빌리티

플레이어가 바라보는 방향으로 타겟을 탐지하고 타겟이 있으면 던지기를, 타겟이 없으면 흘리기를 수행했습니다.

**1. 흘리기(눈먼 돌 던지기)** 흘리기는 키를 입력받고 `UGameplayStatics::PredictProjectilePath` 함수를 통해 경로를 그림니다. 그리고 아이템 메쉬에서 `AddImpulse`에 `Velocity` 값을 매개변수로 하여 호출하였습니다.

**2. 던져 맞추기** 던져 맞추기도 동일하게 `UGameplayStatics::PredictProjectilePath`를 통해서 타겟을 통해 발사하지만 이때 도착할 지점으로 타겟의 소켓정보를 받아 소켓위치로 던지게 만들었습니다.

### ↑ Back to Top

## 공격 어빌리티와 후방 암살(뒷잡)

공격 어빌리티는 2가지로 분기하여 설계했습니다.

**1. 일반공격** 일반 공격은 기본적으로 적이 없거나 적의 정면일때만 수행합니다.

**2. 후방암살** 후방암살은 적이 있을때 후방조건을 탐색하는 방식으로 설계하였습니다.

이때 플레이어의 방향과 적의 방향을 내적하여 `내적한 값이 양수` 일때만 후방으로 처리하였습니다.

그리고 `모션위핑`을 이용하여 플레이어의 위치를 보간했습니다.

### ↑ Back to Top

## 상호작용 어빌리티

상호작용을 하기 위해 `오브젝트를 찾는 역할을 해주는 기능`, 어빌리티의 `활성화와 제거`를 담당해야 했기때문에 `GASShooter`와 `Lyra` 샘플 프로젝트와 유사하게 어빌리티화하여 설계하였습니다.

```
//탐색 어빌리티 클래스
class SICARIUS_API USrAbility_DetectInteraction : public USrGameplayAbility

//상호작용 어빌리티 클래스
class SICARIUS_API USrAbility_Interact : public USrGameplayAbility
```

상호작용 오브젝트가 탐지되면 `UAbilityTask_WaitInputPress`태스크를 통해 입력을 기다리고 입력이 들어오면 `Interact`어빌리티를 활성화하였습니다.

수집이나 장착을 수행할 때는 `IPickupable`와 `IEquipable`의 인터페이스를 사용하여 의존성을 최소화한 형태로 구현하였습니다.

### ↑ Back to Top

## 점프암살 어빌리티

점프 암살은 플레이어가 `Falling`상태이면 `SingleSweepSphere`을 이용해 지속적으로 체크해주었습니다.

지속적인 체크가 필요했기 때문에 어빌리티 시스템에서 제공해주는 `AbilityTask_Repeat`를 사용하였습니다.

```
RepeatTask = UAbilityTask_Repeat::RepeatAction(this, 0.1f, 50);
RepeatTask->OnPerformAction.AddDynamic(this, &ThisClass::OnDetectTarget);
RepeatTask->ReadyForActivation();
```

그리고 수행 도중 타겟을 찾으면 `UAbilityTask_WaitInputPress`를 이용하여 입력이 들어올때까지 기다렸습니다.

## ↑ Back to Top

## 데미지 상호작용

언리얼에서는 기본적으로 **TakeDamage**라는 데미지 상호작용 기능이 있습니다.

이와 비슷하게 `GAS`에서는 **GameplayEffect**를 이용하여 자신이나 다른 플레이어의 **Attributes**를 변경하는 데미지 상호작용 기능들을 제공합니다.

언리얼의 `GAS`에서는 이러한 **데미지상호작용**을 보조해주는 2가지 기능, `ModifierMagnitudeCalculations(MMC)`과 `GameplayEffectExecutionCalculations(ExecCalc)`이 존재하는데, 이 중 `GameplayEffectExecutionCalculations`을 사용하여 구현했습니다.

그리고 공격 어빌리티에서 타겟이 탐지되었을때 `ApplyGameplayEffectToTarget`함수를 호출하여 구현했습니다.

## ↑ Back to Top

## InventorySystem

### TArray Or TMap

인벤토리의 아이템은 삭제와 추가가 빈번하게 일어날 수 있고 중간에서 삭제, 추가될수도있습니다.

시카리우스에서 인벤토리는 인덱스를 통해 UI에서 접근할수도 있어야 했습니다. 그래서 인덱스 접근이 빠른 `TArray`로 구현하였습니다.

장착 함수들은 서버를 통해서 진행되며 기존의 장비체크 후 장착해제, 해제한 슬롯에 장착할 장비를 착용합니다.

- `USrEquipmentSlotComponent.h`

```
void USrEquipmentSlotComponent::SetCurrentSlotItem_Implementation(int32 SlotIndex)
{
    if (EquipmentSlots.IsValidIndex(SlotIndex) && CurrentSlotIndex != SlotIndex)
    {
        UnEquipItemInSlot();

        CurrentSlotIndex = SlotIndex;

        EquipItemInSlot();
    }
}
```

## ↑ Back to Top

# AI

제 게임의 AI는 **순찰**, **경계**, **탐색**, **공격**의 4가지의 상태를 가지도록 하였습니다.

또한 이 과정에서 **EQS**와 **SmartObject**를 통해 **공격**과 **탐색**을 다채롭게 구현하였습니다.



## 상태별 흐름

**AI Perception**에 의해 반응을 하게되면 최상위 상태인 공격, 경계상태로 시작하여 최하위 상태(순찰)까지 내려가는 구조를 가집니다. 상태 변경의 경우 **AI Perception Component**에서 제공하는 **OnTargetPerceptionUpdated** 델리게이트를 통해 구현하였습니다.

**순찰상태**에서는 **SplineComponent**를 이용해서 월드에 위치를 지정한 뒤 **Spline**의 경로에 따라 이동하도록 만들었습니다.

**경계상태**는 **AI Perception**의 **Hearing**이 감지되었을때 감지된 위치로 단순하게 이동합니다. 그리고 **경계상태**가 종료되면 **순찰상태**로 상태수준을 한 단계 낮춥니다.

**공격상태**의 경우 **AI Perception**의 **Sight**나 **Damage**를 통해 활성화되고 **EQS**를 이용하여 공격하기 위한 적절한 위치를 지정하였습니다.

**탐색상태**의 경우 자체적으로 트리거되지 않으며 무조건 **공격상태**를 거쳐서 진행됩니다. **탐색상태**는 **SmartObject**를 이용하여 주변 오브젝트를 탐색하도록 구현하였습니다.

## ↑ Back to Top

## AI Perception Age

적이 시야에서 멀어지거나 **AI Perception**의 **Age**가 만료된 경우 탐색상태로 변경됩니다.

**Age**의 만료된 시점의 경우엔 **AI Perception**의 **virtual void HandleExpiredStimulus(FAIStimulus& StimulusStore) override;** 함수를 재정의 하여 구현했습니다. 제가 구현한 코드에서는 델리게이트를 만들고 **AI Controller**의 함수를 바인딩하여 처리했습니다.

- **USrAI Perception Component.cpp**

```

DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FPerceptionStimulusExired, const
FAIStimulus&, Stimulus);
  
```

```
void USrAIPerceptionComponent::HandleExpiredStimulus(FAIStimulus& StimulusStore)
{
    OnStimuliExpired.Broadcast(StimulusStore);
}
```

## [↑ Back to Top](#)

## SmartObject

언리얼 엔진에서는 AI와 플레이어의 상호작용에 **SmartObject**라는 최신 기술이 존재합니다.

이 기술은 **GameplayTag**로 동작하는데, 이를 통해 AI의 상호작용을 의존성을 최소화하여 구현하였습니다.

## [↑ Back to Top](#)

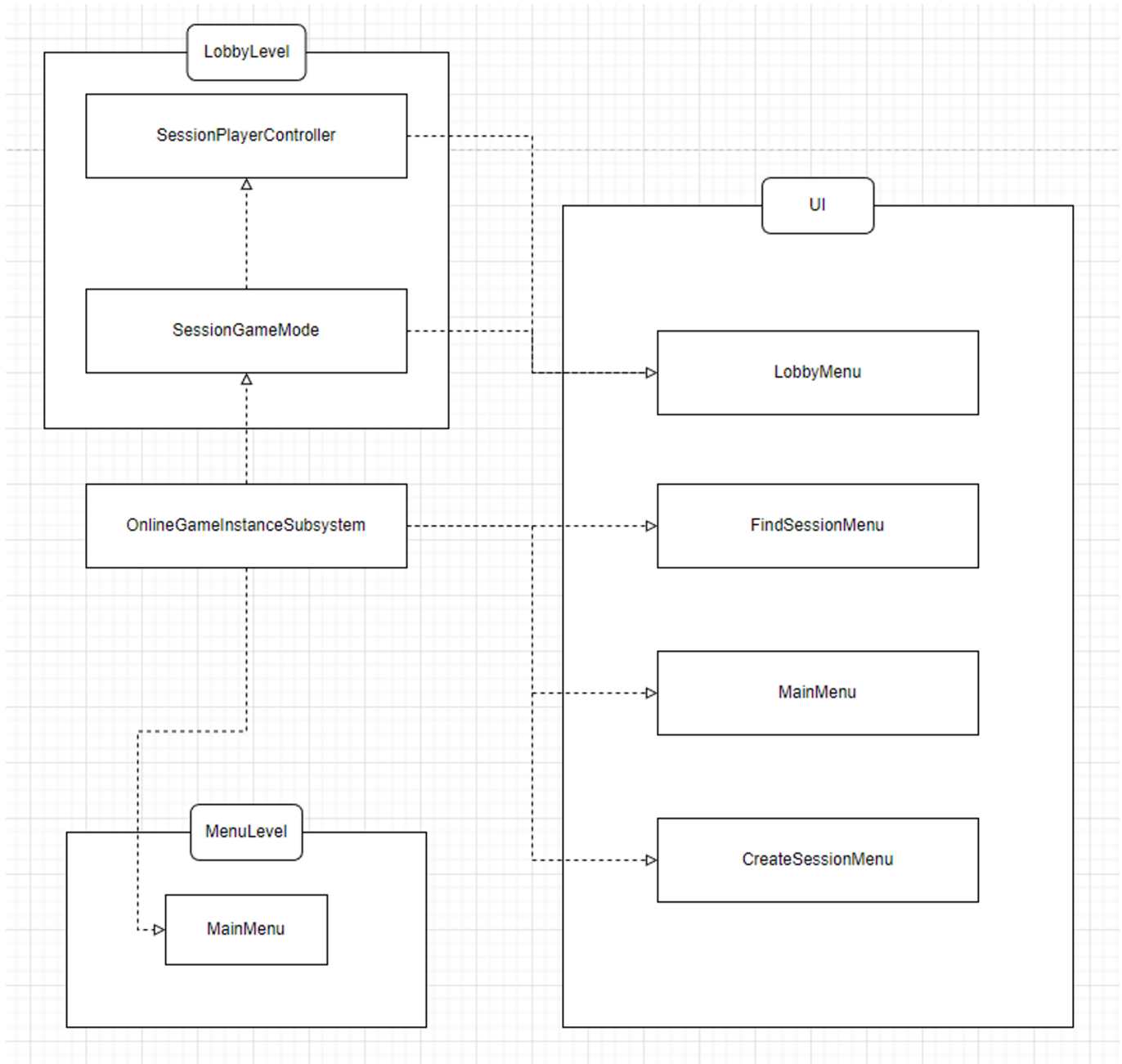
## OSS Steam

### 멀티 세션의 구조 설계

**멀티 세션 시스템**은 단순히 현재 프로젝트에만 사용하는데 그치지 않고 **범용적으로 사용할 수 있도록 플러그인**의 형태로 구현하였습니다.

### 로비와 게임의 게임모드 분리

실제 게임에서 사용하는 게임모드와는 별개로 **Lobby**에 들어왔을때는 **게임의 로직과는 다른 구조로 로직이 구성되기 때문에** 세션게임모드와 컨트롤러를 별개로 생성하여 관리했습니다.



### OnlineGameInstanceSubsystem에 의존하고 있는 객체들

그리고 세션을 통해 월드 이동시 컨트롤러의 Possess문제가 발생하여 연결을 끊고 재연결하는 **Non-SeamlessTravel**을 사용했습니다.

### [↑ Back to Top](#)

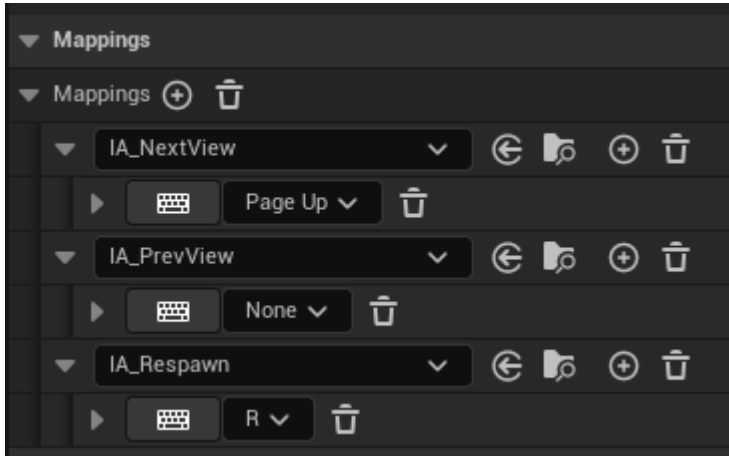
## Spectator

Styx : Shard of Darkness 게임에서 캐릭터가 사망 이후 특정 플레이어를 관전하고 리스폰하는 시스템에 영감을 얻어서 제 프로젝트에도 구현해보면 어떨까라는 생각으로 관전 시스템을 구현해봤습니다.

언리얼에서 제공해주는 기본 SpectatorPawn에 관전과 관련된 기능을 추가하여 구현했습니다.

### 관전 입력

관전 시점에서는 실제 게임의 모든 입력이 막혀야 하고 관전시점에서의 새로운 입력이 존재해야합니다. 이를 근거로하여 관전용 새로운 입력맵핑을 만들었습니다.



## 관전용 입력값들

### 관전자 전환

관전과 관련된 기능은 기본적으로 **APlayerController**를 통해서 동작하기때문에 기존의 **SrPlayerController**에서 관전상태로 변환해주는 로직을 추가해줬습니다.

```
void ASrSpectatorPawn::ViewNextPlayer()
{
    if (APlayerController* PC = GetController<APlayerController>())
    {
        PC->ServerViewNextPlayer();
    }
}
```

또한 **SpectatorPawn**은 로컬에서만 생성되어야 하기 때문에 **ChangeState**(서버호출)함수에서 관전자용 폰을 삭제하고 **ClientGotoState**를 통해 **SpectatorPawn**을 생성해줬습니다.

### ↑ Back to Top

### 리스폰

플레이어가 다시 스폰되는 과정은 입력은 컨트롤러에서 발생하지만 실제 스폰되는 과정은 서버를 통해서 이루어져야합니다.

그러기 때문에 모든 리스폰 로직은 게임모드에서 동작하도록 구현하였습니다.

게임모드에서는 먼저 기존의 관전셋팅을 게임셋팅으로 변환하고 **DefaultPawnClass**를 이용하여 플레이어를 적절한 위치에 스폰시켜줬습니다.

```
APawn* RespawnPawn = GetWorld()->SpawnActorDeferred<APawn>(DefaultPawnClass,
SpawnTransform, InPlayerController);

InPlayerController->Possess(RespawnPawn);

RespawnPawn->FinishSpawning(SpawnTransform);
```




기존의 `SpawnActor`를 사용하면 `Possess`와 `BeginPlay`중 `BeginPlay`가 먼저 실행되어서 로직이 꼬이는 문제가 발생하였는데, 이를 `SpawnDeffered`를 통해 `Possess`와 `BeginPlay`의 호출 순서를 의도적으로 변경해주었습니다.

[↑ Back to Top](#)

## Extra : LineTracing Attack

간혹 공격 애니메이션이 너무 빠르거나 프레임저하로 인해 모션이 스킵되면 공격이 무시되는 문제가 발생했습니다. 문제의 원인은 무기의 충돌체가 빠르게 움직이면서 판정을 무시하게 되는 것이었습니다.

이 문제를 이전 소켓위치와 다음 소켓위치를 통해 라인 트레이싱으로 공격 판정을 변경하여 적용하였습니다.

라인트레이싱공격

### 라인 트레이싱 공격 설계

이 방법을 사용하면 애니메이션의 프레임과 프레임 사이가 멀어도 어느정도 보간해서 충돌위치를 찾아낼 수 있습니다.

[↑ Back to Top](#)