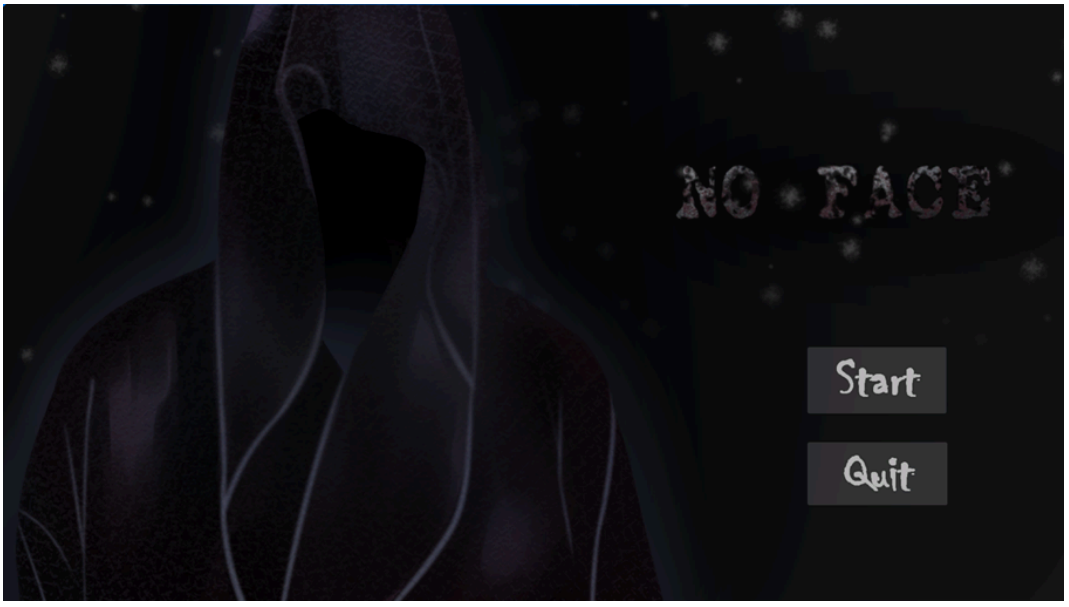
No-Face

develop period	@2024년 9월 6일 → 2024년 12월 4일
roll	기획 팀장 프로그래밍 프로젝트 매니저
category	언리얼 게임 팀프로젝트
size	Large
FrameWork	Unreal
언어	C++

개요



- 전남대학교의 3-2 캡스톤디자인 작품입니다.

전체 코드는 다음 깃에 올라가 있습니다.

https://github.com/SingABro/No-Face

게임 설명

절차적 콘텐츠 생성 알고리즘 (Procedural Content Generation, 이하 PCG) 알고리즘을 사용하여 자동 맵 생성이 되는 로그라이크 RPG 입니다. 던전을 탐색하며 몬스터를 처치한 뒤 마지막 보스 몬스터를 처치하면 게임이 끝나게 됩니다. 게임이 끝난 후 다시 실행하면 이전의 맵과 다른 새로운 맵이 생성되며, 플레이어는 새로운 환경에서 게임을 진행할 수 있게 됩니다.

PCG 알고리즘은 일련의 규칙을 반복적으로 수행하여 콘텐츠를 자동으로 생성하는 알고리즘입니다. 주로 게임 개발에서 사용되며, 게임의 다양성과 재미를 증가시키고, 특히 레벨 디자이너의 작업 부담을 줄이는 역할을 합니다. 최근에는 단순히 게임 맵의 지형지물을 자동, 랜덤하게 생성하는 것 외에도 적 생성, 스토리 생성, 음악 생성 등 다양한 분야로 확장되고 있으며 게임 개발의 주요 기술로 자리 잡고 있습니다.

해당 게임은 PC 윈도우 플랫폼에서 실행되며, 쿼터뷰로 마우스를 이용해 이동, 키보드를 이용해 스킬 및 기타 상호작용을 실행합니다.

이동 - 마우스 우클릭

기본 공격 - 마우스 좌클릭

대쉬 - Space Bar

스킬 - Q, W, E, R

무기 교체 - Z, X

미니맵 - Tab

플레이어는 3개의 무기(클래스)를 가지고 있으며 전투 중 자유롭게 변경하여 상황을 헤쳐나갈 수 있습니다. 또한 레벨업을 통해 각 무기의 스킬을 강화할 수 있습니다.

플레이 영상

https://prod-files-secure.s3.us-west-2.amazonaws.com/7ebf7000-f855-492a-973c-2e002d905ac7/41acda65-fe66-43ce-af3a-9081c7e2b444/%ED%99%94%EB%A9%B4_%EB%85%B9%ED%99%94_%EC%A4%91_2024-12-24_215406.mp4

개발

git lfs를 사용해 작업물을 공유

본인파트

맵 생성 알고리즘을 제외한 모든 C++ 코드 작성, 게임 기능 개발

- 전투 시스템
  - 기본 공격
  - 스킬 공격
  - 무기 교체
- 스텟 시스템
- UI
  - 스킬 쿨타임바
  - 미니맵 이동
  - 플레이어 체력바
- 몬스터
  - 6종 몬스터 기획 및 개발
  - Behavior Tree
- 스테이지
  - 다음 스테이지로 이동
  - 몬스터 스폰
  - 스테이지 상태
- 애니메이션
  - 캐릭터 애니메이션 블루프린트
  - 몬스터 애니메이션 블루프린트
  - 애니메이션 리타게팅

▼ 해당 화살표를 누르면 자세한 설명을 보실 수 있습니다.

코드 및 설정에 대한 설명입니다.

플레이어 전투 시스템

- 기본공격
  - CharacterDefaultAttackComponent
    - 검 기본공격
      - 4 콤보까지 공격이 존재 → 애님 몽타주를 활용해 구현
      - Notify를 활용해 공격 판정
      - Notify 클래스와 Component를 Interface로 연결하여 컴파일 의존성을 줄임

- 부채꼴 공격 판정

- 활 기본공격
  - 콤보 공격이 없는 단일 공격
  - Notify를 활용해 활 애니메이션 실행
  - Notify를 활용해 화살 생성
  - Notify 클래스와 Component를 Interface로 연결하여 컴파일 의존성 줄임
  - 화살은 단일 공격으로 한번 몬스터와 부딪히면 없어짐
- 스태프 기본공격
  - 3 콤보까지 공격이 존재 → 애님 몽타주를 활용해 구현
  - Notify를 활용해 공격 판정
  - Notify 클래스와 Component를 Interface로 연결하여 컴파일 의존성 줄임
  - 다중 공격으로 몬스터들을 뚫고 뒤의 몬스터까지 피격

▼ **기본 공격 코드 흐름**

```
void ACharacterBase::OnAttackStart()
{
    if (Cast<AStoryBook>(UGameplayStatics::GetActorOfClass(GetWorld(), AStoryBook::Static

    if (TraceAttack() == false || SkillComponent->GetSkillState() == ESkillState::Progress)
    {
        return;
    }

    /* 스킬 캐스팅 중이면 해당 스킬 싱행 */
    if (SkillComponent->GetCastingFlag())
    {
        TFunction<void()> SkillAction;
        if (SkillComponent->SkillQueue.Dequeue(SkillAction))
        {
            RotateToTarget();
            SkillAction();
            return;
        }
    }

    OnClickStart();
    RotateToTarget();
    AttackComponent->BeginAttack();
}
```

1. 캐릭터 클래스에 `OnAttackStart()` 함수가 마우스 좌클릭을 클릭 시 호출되도록 바인딩 되어있습니다.
2. `OnClickStart()` 함수는 캐릭터의 움직임을 멈추고 `RotateToTarget()` 함수는 공격 방향으로 캐릭터를 회전시킵니다.

```
void UCharacterDefaultAttackComponent::BeginAttack()
{
    if (CurrentCombo == 0)
    {
        switch (CurrentWeaponType)
        {
        case 0:
            BeginSwordDefaultAttack();
            return;
        case 1:
            BeginBowDefaultAttack();
            return;
        case 2:
            BeginStaffDefaultAttack();
            return;
        default:
            return;
        }
    }

    if (CurrentWeaponType == 0)
    {
        if (!SwordComboTimer.IsValid())
        {
            SwordHasNextComboCommand = false;
        }
        else
        {
            SwordHasNextComboCommand = true;
        }
    }
    else if (CurrentWeaponType == 2)
    {
        if (!StaffComboTimer.IsValid())
        {
            StaffHasNextComboCommand = false;
        }
        else
        {
            StaffHasNextComboCommand = true;
        }
    }

}
```

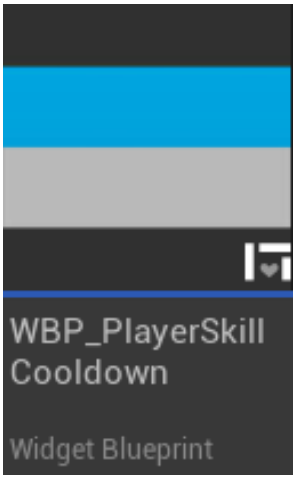
3. `AttackComponent` 는 `UCharacterDefaultAttackComponent`를 가리키는 포인터로 캐릭터의 현재 무기 상태에 해당하는 공격을 실행합니다.

- 스킬공격

- SkillComponent
  - 검 Q,W,E,R
    - Q - 4개의 검기를 날림
    - W - 플레이어 주변으로 파동을 날림
    - E - 몬스터의 공격을 막은 후 반격 (패링)
    - R - 일정 준비 시간 이후 발도
  - 활 Q,W,E,R
    - Q - 전방에 범위 공격
    - W - 폭탄 화살을 날려 떨어진 지점의 주위에 있는 몬스터들에게 대미지 적용 (캐스팅 스킬)
    - E - 백스텝
    - R - 현재 플레이어가 바라보는 방향의 뒤로 점프하여 전방에 대미지를 입히는 공격
  - 스태프 Q,W,E,R
    - Q - 메테오를 떨어트려 해당 지점 주위에 있는 몬스터들에게 대미지 적용 (캐스팅 스킬)
    - W - 몬스터들을 빨아들이는 블랙홀 생성 (캐스팅 스킬)
    - E - 보호막
    - R - 플레이어 주변에 낙뢰를 떨어트림
  - 대쉬
    - 플레이어가 현재 바라보는 방향으로 빠른 속도로 이동
- 스킬 쿨타임바
  - 총 12개의 스킬 각각 스킬을 시전하면 쿨타임바가 지나감
  - 무기 스킬마다 쿨타임이 존재
    - ex) 검 Q 실행 후 활로 무기를 바꿔서 바로 Q를 사용할 수 있음

▼ **스킬 쿨타임바 코드 흐름**

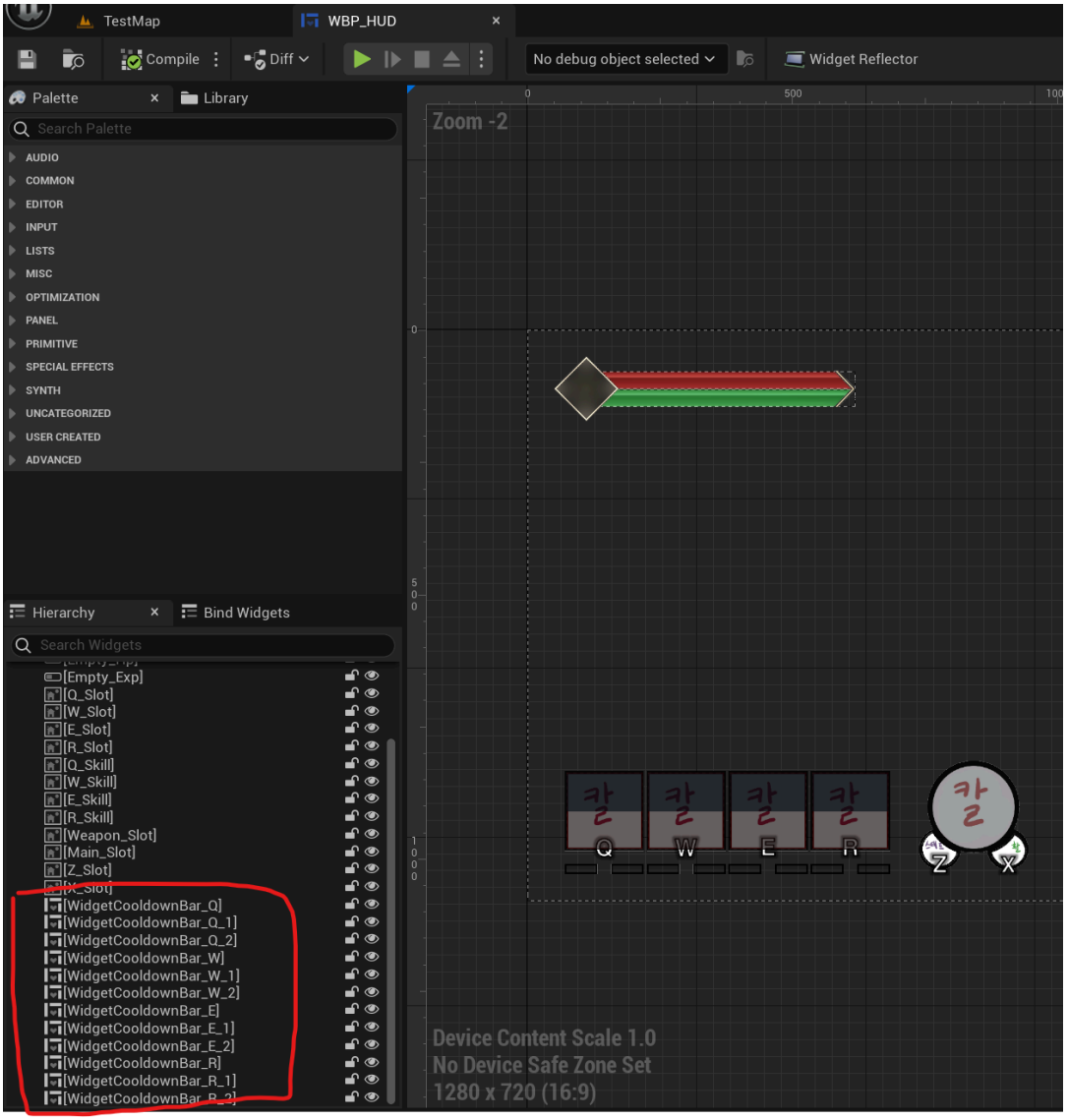
쿨타임바 위젯



- `WBP_PlayerSkillCooldown` 은 SkillCooldownUserWidget 이란 클래스를 상속받으며 C++로 기능이 구현되어 있습니다.

```
void USkillCooldownUserWidget::NativeConstruct()
{
   Super::NativeConstruct();

   CooldownBar = Cast<UProgressBar>(GetWidgetFromName(TEXT("SkillCooldownBar")));
   ensure(CooldownBar);

   //처음 UI 생성시 쿨타임바 초기화
   CooldownBar->SetPercent(0.f);
}

void USkillCooldownUserWidget::UpdateCooldownBar(float CurrentTime)
{
   //StartCooldown 함수가 명시적으로 호출되어야지 쿨타임바의 업데이트 시작
   if (bIsCooldownActive)
   {
      //외부에서 들어오는 시간을 그대로 적용한다.
      CooldownBar->SetPercent(CurrentTime / MaxCooldownTime);

      //최대 쿨타임보다 커지면 더 이상 업데이트를 진행하지 않으며 쿨타임바를 초기화한다.
      if (CurrentTime >= MaxCooldownTime - KINDA_SMALL_NUMBER)
      {
         bIsCooldownActive = false;
         CooldownBar->SetPercent(0.f);
      }
   }
}

//최대 쿨타임 시간을 정하는 함수
void USkillCooldownUserWidget::SetCooldownTime(float InMaxCooldownTime)
{
   MaxCooldownTime = InMaxCooldownTime;
}

//쿨타임바의 실행 트리거 함수
void USkillCooldownUserWidget::StartCooldown()
{
   bIsCooldownActive = true;
}
```

HUD 위젯



- `Q,W,E,R` / `Q_1, W_1, E_1, R_1` / `Q_2, W_2, E_2, R_2` 끼리 한 세트이며 각각 검, 활, 지팡이 스킬 쿨타임바를 의미합니다.
- HUD 클래스에서 인스턴스를 받아오고 있습니다.

```
UPROPERTY()
TObjectPtr<class USkillCooldownUserWidget> Sword_SkillCooldownBar_Q;

UPROPERTY()
TObjectPtr<class USkillCooldownUserWidget> Bow_SkillCooldownBar_Q;

UPROPERTY()
TObjectPtr<class USkillCooldownUserWidget> Staff_SkillCooldownBar_Q;

UPROPERTY()
TObjectPtr<class USkillCooldownUserWidget> Sword_SkillCooldownBar_W;

UPROPERTY()
```

```
    TObjectPtr<class USkillCooldownUserWidget> Bow_SkillCooldownBar_W;

    UPROPERTY()
    TObjectPtr<class USkillCooldownUserWidget> Staff_SkillCooldownBar_W;

    UPROPERTY()
    TObjectPtr<class USkillCooldownUserWidget> Sword_SkillCooldownBar_E;

    UPROPERTY()
    TObjectPtr<class USkillCooldownUserWidget> Bow_SkillCooldownBar_E;

    UPROPERTY()
    TObjectPtr<class USkillCooldownUserWidget> Staff_SkillCooldownBar_E;

    UPROPERTY()
    TObjectPtr<class USkillCooldownUserWidget> Sword_SkillCooldownBar_R;

    UPROPERTY()
    TObjectPtr<class USkillCooldownUserWidget> Bow_SkillCooldownBar_R;

    UPROPERTY()
    TObjectPtr<class USkillCooldownUserWidget> Staff_SkillCooldownBar_R;
```

```
ACharacterBase* Character = Cast<ACharacterBase>(GetOwningPlayerPawn());
if (Character)
{
    Character->SignedChangeWeapon.AddUObject(this, &UHUDWidget::SetSkillUI);
    Character->SetupHUDWidget(this);
}
```

- HUD 클래스는 플레이어 클래스의 포인터를 받아와 `FOnSignedChangeWeapon` 델리게이트에 `SetSkillUI(int32 WeaponType)` 를 등록합니다.
- 해당 델리게이트는 캐릭터의 무기가 바뀔 때 브로드캐스트를 보내며 바인딩된 함수는 현재 무기 스킬의 쿨타임바만 보이도록 작동합니다.

- 그리고 쿨타임바 위젯 함수를 실행할 수 있는 함수들도 정의되어 있습니다.

```
void SetMaxCooldown(float InMaxCooldownTime, int32 WeaponType,
                                ESkillType SkillType);
void StartCooldown(int32 WeaponType, ESkillType SkillType);
void UpdateCooldownBar(float CooldownDuration, FTimerHandle& CooldownTimerHandle,
                                bool& bCanUseSkill, ESkillType SkillType,
                                int32 WeaponType, float& Timer);
```

- 해당 함수들은 현재 무기타입, 사용자가 누른 스킬 타입 (Q,W,E,R) 에 따라 쿨타임바를 움직이도록 작동합니다.

```
switch (WeaponType)
{
```

```
case 0:
   switch(SkillType)
   {
   case ESkillType::Q:
      로직 실행
      break;
   case ESkillType::W:
      로직 실행
      break;
   ...
   }
   break:
case 1:
...

}
```

이런식으로 이중 switch 문이다.

```
12개의 쿨타임바마다 아래 로직이 실행된다.
GetWorld()→GetTimerManager().SetTimer(CooldownTimerHandle,
   [&, SkillType, CooldownDuration]()
   {
      float ElapsedTime = GetWorld()→GetTimerManager().GetTimerElapsed(CooldownTimerH
      Timer += ElapsedTime;

      Sword_SkillCooldownBar_Q→UpdateCooldownBar(Timer);

      if (Timer >= CooldownDuration)
      {
         GetWorld()→GetTimerManager().ClearTimer(CooldownTimerHandle);
         bCanUseSkill = true;
         Timer = 0.f;
      }
   }, 0.01f, true);
```

쿨타임바를 업데이트 해주는 로직은 다음과 같습니다.
쿨타임바와 관련된 함수는 SkillComponent 클래스에서 호출됩니다.
SkillComponent에서 넘겨주는 인자를 바탕으로 쿨타임바를 업데이트 합니다.

```
void UpdateCooldownBar(float CooldownDuration, FTimerHandle& CooldownTimerHandle, bool& bCanUseSkill, ESkillType SkillType, int32 WeaponType, float& Timer);
```

float : 스킬의 MaxCooldown을 정하기 위해 인자를 받는다. 인자로 받기 때문에 람다 함수 안에서 사용하기 위해 캡처해줘야합니다.

FTimerHandle : 각 스킬 쿨타임을 관리해줍니다. 스킬 12개에 모두 필요합니다.

ESkillType : IPlayerSkillUIInterface.h 에 선언된 enum class 로 본래 캐릭터 클래스에 있는 HUD 포인터

를 SkillComponent에서 받아오기 위해 만들었습니다. 그런데 스킬을 구분할 플래그가 필요해서 다른 클래스를 추가로 만들지 않고 해당 인터페이스에 enum class를 만들고 HUDWidget 클래스에 포함시켰습니다.

bool : 스킬 플래그이며 SkillComponent에 정의되어 있습니다. 스킬들은 해당 플래그를 이용해 스킬을 온오프하며 원본이 바뀌어야 하기에 참조로 받습니다.

int32 : 현재 무기 상태를 구분하기 위한 인자입니다.

float& : 스킬이 사용된 후 몇 초가 지났는지 저장하는 변수입니다. SkillComponent에 각 스킬마다 정의되어 있으며 해당 변수의 +-에 따라 쿨타임바가 변합니다.

스킬 컴포넌트

- 스킬 컴포넌트에서는 쿨타임을 지정만 해줍니다.
- 해당 함수가 스킬 시전할 때마다 호출됩니다.

```
void USkillComponent::StartCooldown(float CooldownDuration,
FTimerHandle& CooldownTimerHandle, bool& bCanUseSkill,
ESkillType SkillType, int32 WeaponType, float& Timer)
{
   bCanUseSkill = false;

   Widget→SetMaxCooldown(CooldownDuration, CurrentWeaponType, SkillType);
   Widget→StartCooldown(CurrentWeaponType, SkillType);
   Widget→UpdateCooldownBar(CooldownDuration, CooldownTimerHandle,
                                          bCanUseSkill, SkillType, WeaponType,Timer);
}
```

- 무기 교체
  - 현재 어떤 무기를 지니고 있는지 enum class를 통해 표현
  - Z, X 키를 활용해 무기를 전환하며 각각 이전의 무기, 다음의 무기로 변경

  ▼ 무기 교체 코드 흐름

```
DECLARE_DELEGATE(FTakeItemDelegate);

USTRUCT()
struct FTakeItemDelegateWrapper
{
   GENERATED_BODY()

   FTakeItemDelegateWrapper() {}
   FTakeItemDelegateWrapper(const FTakeItemDelegate& InTakeItemDelegate)
   : TakeItemDelegate(InTakeItemDelegate) {}

   FTakeItemDelegate TakeItemDelegate;
};
```

1. 델리게이트를 구조체로 감싼 후 `TArray`로 관리합니다.

```
void ACharacterBase::NextWeapon()
{
   if (!AttackComponent->CanChangeWeapon()
   || !SkillComponent->CanChangeWeapon())
   {
      return;
   }

   WeaponIndex += 1;
   if (WeaponIndex > 2)
   {
      WeaponIndex = 0;
   }

   ChangeWeapon();
   AnimWeaponIndex();
}

void ACharacterBase::PrevWeapon()
{
   if (!AttackComponent->CanChangeWeapon()
   || !SkillComponent->CanChangeWeapon())
   {
      return;
   }

   WeaponIndex -= 1;
   if (WeaponIndex < 0)
   {
      WeaponIndex = 2;
   }

   ChangeWeapon();
   AnimWeaponIndex();
}
```

2. `NextWeapon()` 함수는 `Z` 키에 바인딩되어 있으며, 무기를 다음 무기로 전환합니다`PrevWeapon()` 함수는 `X` 키에 바인딩되어 있으며, 무기를 이전 무기로 전환합니다

   현재 기본 공격, 스킬 공격이 진행 중인지 검사 후 무기 전환 실행합니다.

   두 함수 모두 `WeaponIndex`를 조작하여 현재 선택된 무기의 인덱스를 변경합니다.

   인덱스는 0에서 2까지의 값을 가지며, 0은 검, 1은 활, 2는 지팡이를 나타냅니다.

   인덱스가 범위를 벗어나면 순환되도록 구현되어 있습니다.

```
void ACharacterBase::ChangeWeapon()
{
```

```
    SkillComponent->SetWeaponType(WeaponIndex);
    AttackComponent->SetWeaponType(WeaponIndex);
    TakeItemDelegateArray[WeaponIndex].TakeItemDelegate.ExecuteIfBound();
    CurrentWeaponType = static_cast<EWeaponType>(WeaponIndex);
    SignedChangeWeapon.Broadcast(WeaponIndex);
}
```

3. `ChangeWeapon()` 함수는 실질적으로 무기를 전환하는 함수입니다.

   `SkillComponent`와 `AttackComponent`에 새로운 무기 타입(인덱스)을 설정입니다.

   `TakeItemDelegateArray` 배열에 저장된 델리게이트를 호출하여 해당 무기를 장착합니다

   `CurrentWeaponType` 변수에 현재 무기의 타입을 저장하고, `SignedChangeWeapon` 이벤트를 통해 UI 또는 애니메이션과 연동됩니다.

```
void ACharacterBase::EquipSword()
{
    if (WeaponBase)
    {
        WeaponBase->Destroy();
    }

    UGameplayStatics::PlaySoundAtLocation(GetWorld(), ToSwordChangeSound, GetActorLocatio
    FVector SpawnLocation = GetMesh()->GetSocketLocation(TEXT("hand_rSocket"));
    FRotator SpawnRotation = GetMesh()->GetSocketRotation(TEXT("hand_rSocket"));
    GetCharacterMovement()->MaxWalkSpeed = 785.f;

    WeaponBase = GetWorld()->SpawnActor<ASword>(SwordClass, SpawnLocation, SpawnRotati
    WeaponBase->AttachToComponent(GetMesh(),
    FAttachmentTransformRules::SnapToTargetNotIncludingScale, TEXT("hand_rSocket"));
}
```

4. 검, 활, 스태프의 상위 클래스를 만들어 다형성을 이용해 무기를 관리합니다.

   이전의 무기를 `Destroy()` 를 통해 삭제 후 현재 무기 상태에 맞는 무기를 스폰됩니다.

- 스텟
  - CharacterStatComponent
    - 체력바
    - 경험치
    - 레벨

AI 몬스터 기획 및 구현

EnemyBase란 상위 클래스를 만들어 기본적인 기능을 구현한 후, 이를 상속받아 몬스터의 종류별로 기능을 더 자세히 구현하고자 했음

EnemyBase의 역할

- 공격받았을 때
  - 대미지 UI 띄우기
  - 특정 공격에 특정 모션 실행
  - 히트 파티클 실행
- 죽었을 때
  - 경험치
  - 죽음 플래그
  - 몬스터 삭제
- 스턴
- 체력바 띄우기
  - IHelixSkillInterface 상속
- IAIInterface 상속
  - BTDecorator, BTService, BTTask 클래스를 작성할 때 몬스터 클래스를 직접 참조하는 것이 아닌 IAIInterface를 통해 간접 참조

몬스터 6종 제작

모두 EnemyBase 클래스를 상속받아 제작

1. AI Perception

   각 몬스터의 `AI Controller`에 `AI Perception`을 등록

   근접 몬스터는 Sight, Damage 감각을 인식, 원거리 몬스터는 Sight 감각만을 인식

   몬스터 종류마다 인식 범위가 다르며 플레이어를 인식하면 Blackboard의 데이터를 새로 고침

   ▼ AI Perception 설정값

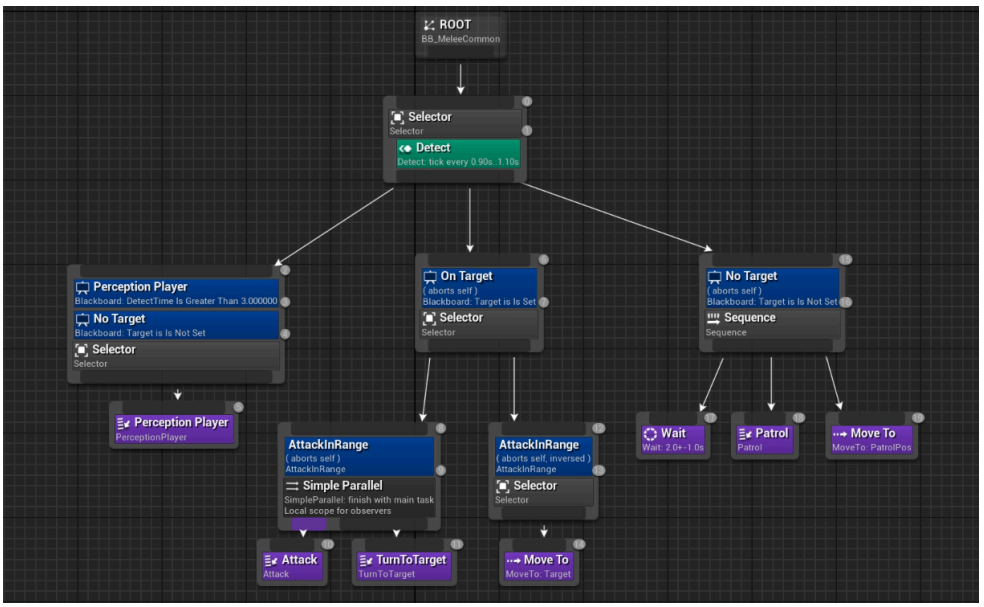
   1. 근접 탱커
      a. 기본 이동 속도 600
      b. 2콤보 기본 공격, 공격 사거리 200
      c. 최초 플레이어 탐지 거리 1500, 플레이어를 놓치는 거리 2000, 시야각 270
      d. 탐지 감각 시각, 대미지
      e. 공격을 받을 때마다 SkillEnergy가 10씩 쌓이며 50이 되면 스킬 사용
   2. 근접 일반
      a. 기본 이동 속도 600
      b. 단일 콤보 기본 공격, 공격 사거리 300
      c. 최초 플레이어 탐지 거리 1500, 플레이어를 놓치는 거리 2000, 시야각 270
      d. 탐지 감각 시각, 대미지
   3. 근접 어쌔신
      a. 기본 이동 속도 600
      b. 2콤보 기본 공격, 공격 사거리 400
      c. 최초 플레이어 탐지 거리 1500, 플레이어를 놓치는 거리 2000, 시야각 270

d. 탐지 감각 시각, 대미지

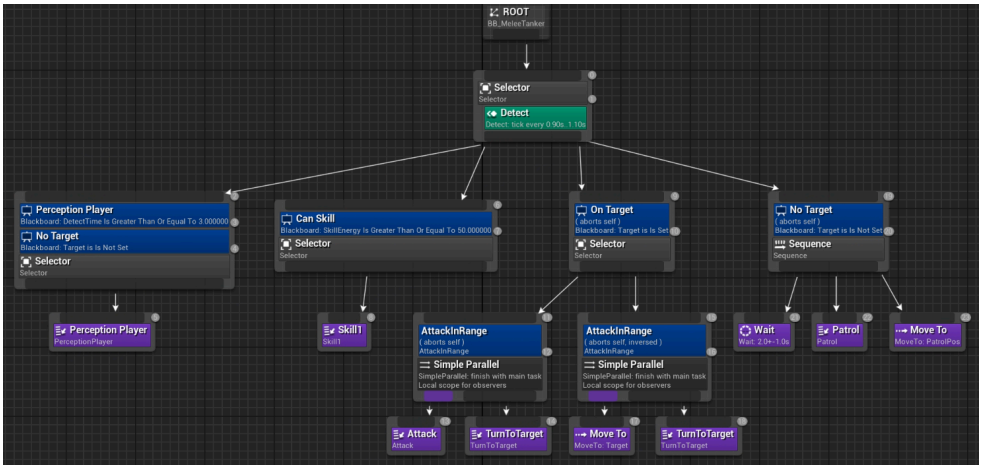
4. 원거리 일반
    a. 플레이어와 거리가 400 이하라면 근접 공격
    b. 플레이어와 거리가 2000 이하라면 원거리 기본 공격
5. 원거리 시즈
    a. 플레이어와 거리가 400 이하라면 근접 공격
    b. 플레이어와 거리가 2500 이하라면 원거리 기본 공격
    c. 플레이어와 거리가 2500 이상이라면 시즈 공격

2. Behavior Tree

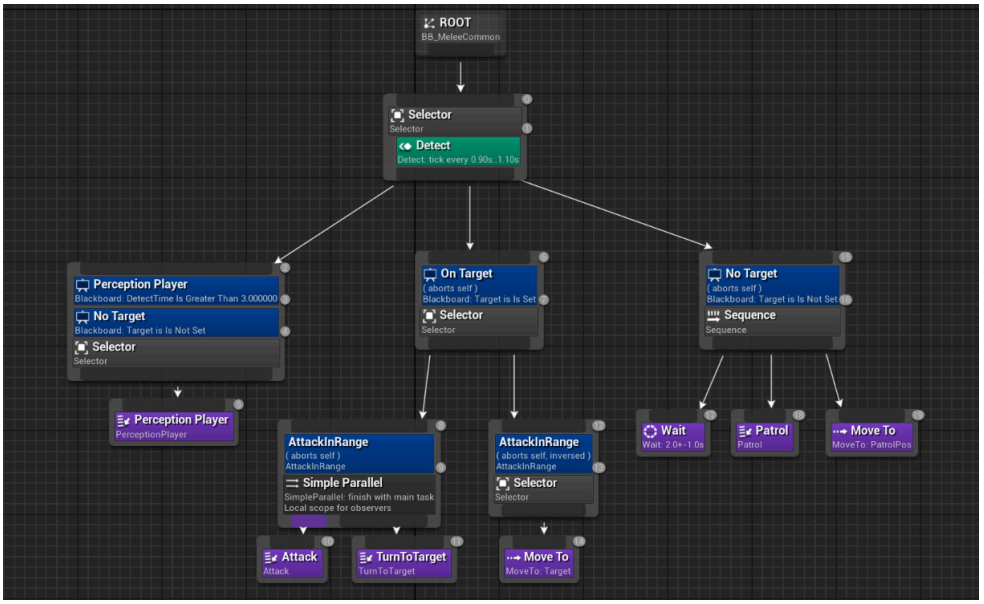
▼ 근접 일반 몬스터



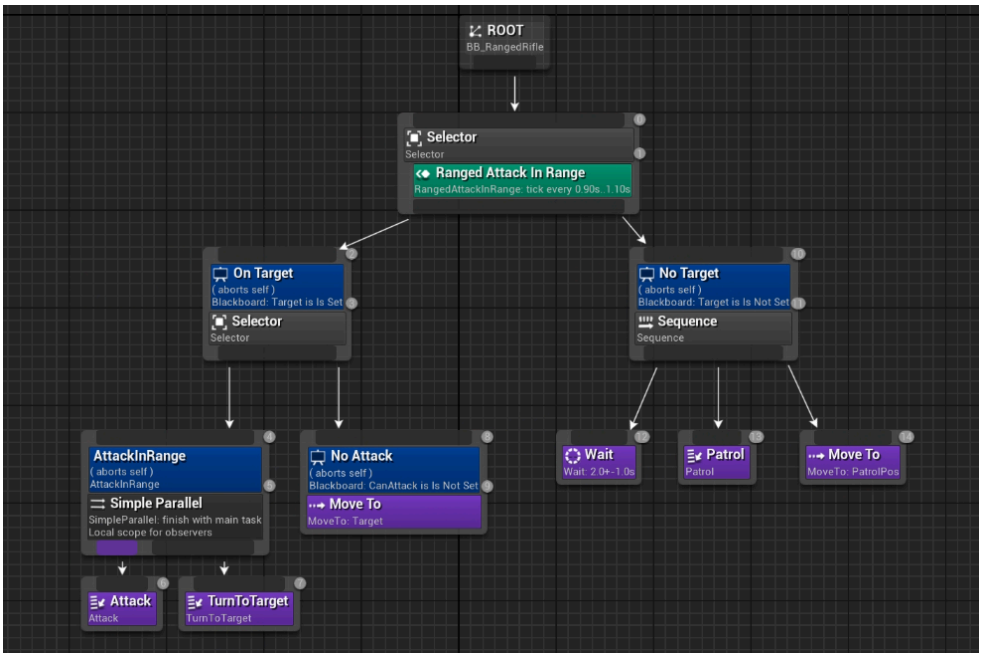
▼ 근접 탱커 몬스터



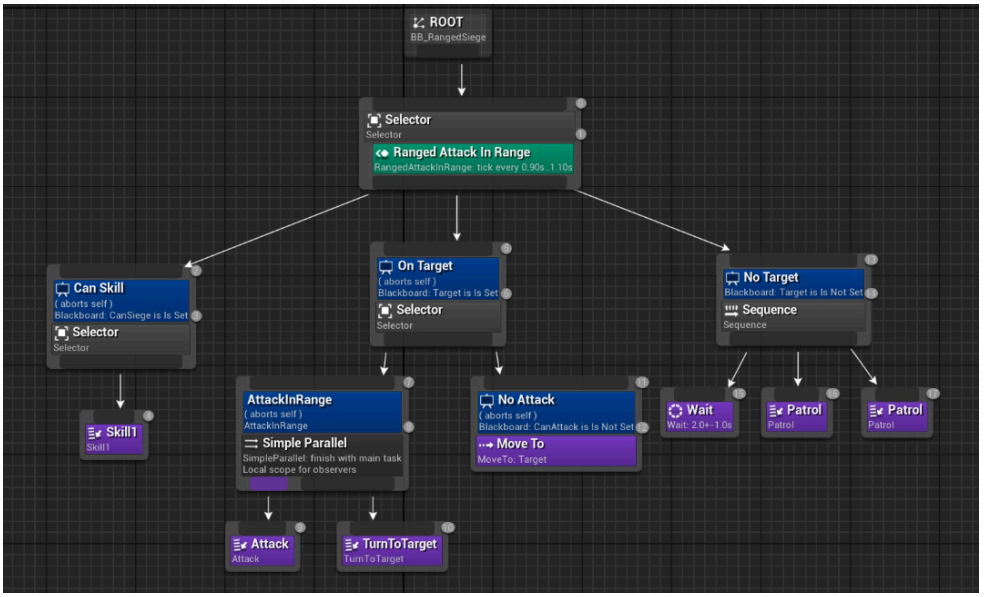
▼ 근접 어쌔신 몬스터



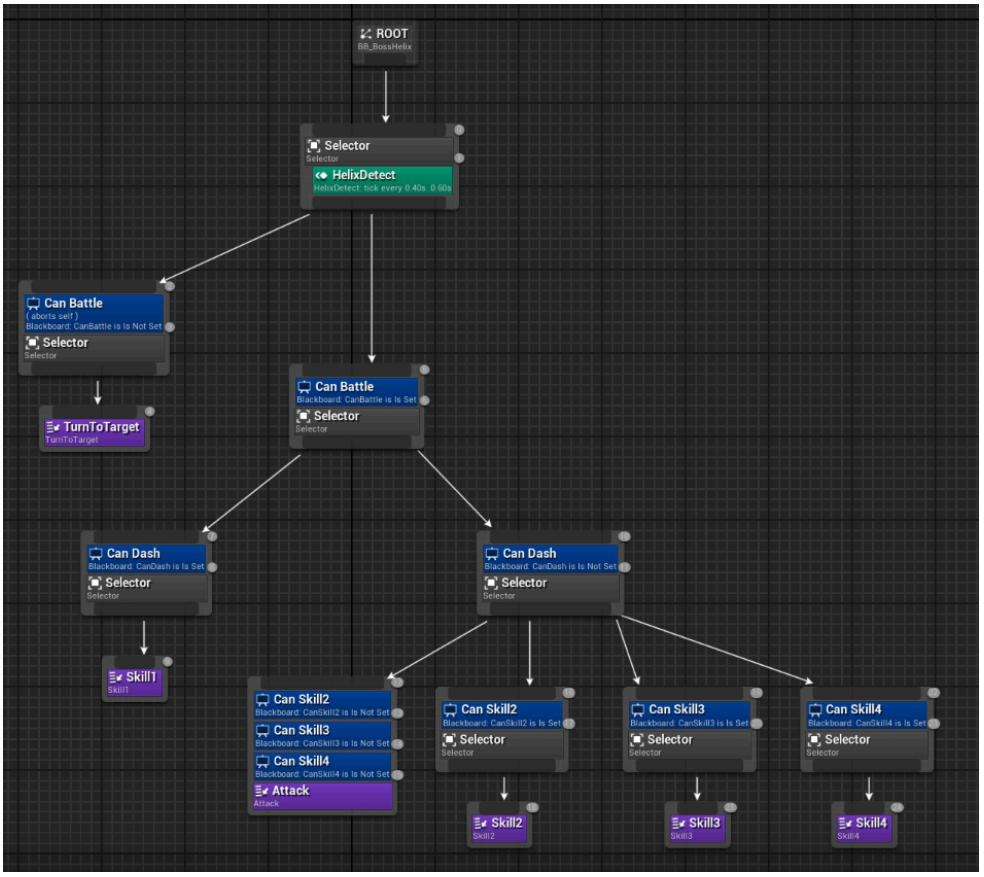
▼ 원거리 일반 몬스터



▼ 원거리 시즈 몬스터


ROOT
BB_RangedSiege
Selector
Selector
Ranged Attack In Range
RangedAttackInRange: tick every 0.90s..1.10s
Can Skill
(aborts self)
Blackboard: CanSiege is Is Set
Selector
Selector
On Target
(aborts self)
Blackboard: Target is Is Set
Selector
Selector
No Target
Blackboard: Target is Is Not Set
Sequence
Sequence
Skill1
Skill1
AttackInRange
(aborts self)
AttackInRange
Simple Parallel
SimpleParallel: finish with main task
Local scope for observers
No Attack
(aborts self)
Blackboard: CanAttack is Is Not Set
Move To
MoveTo: Target
Wait
Wait: 2.0+-1.0s
Patrol
Patrol
Patrol
Patrol
Attack
Attack
TurnToTarget
TurnToTarget

▼ 보스 몬스터


ROOT
BB_BossHelix
Selector
Selector
HelixDetect
HelixDetect: tick every 0.40s..0.60s
Can Battle
(aborts self)
Blackboard: CanBattle is Is Not Set
Selector
Selector
Can Battle
Blackboard: CanBattle is Is Set
Selector
Selector
TurnToTarget
TurnToTarget
Can Dash
Blackboard: CanDash is Is Set
Selector
Selector
Can Dash
Blackboard: CanDash is Is Not Set
Selector
Selector
Skill1
Skill1
Can Skill2
Blackboard: CanSkill2 is Is Not Set
Can Skill3
Blackboard: CanSkill3 is Is Not Set
Can Skill4
Blackboard: CanSkill4 is Is Not Set
Attack
Attack
Can Skill2
Blackboard: CanSkill2 is Is Set
Selector
Selector
Can Skill3
Blackboard: CanSkill3 is Is Set
Selector
Selector
Can Skill4
Blackboard: CanSkill4 is Is Set
Selector
Selector
Skill2
Skill2
Skill3
Skill3
Skill4
Skill4

맵

- 스테이지 이동

```
UENUM(BlueprintType)
enum class EStageState : uint8
{
   READY = 0,
   FIGHT,
   NEXT
};
```

  - 각 스테이지는 3가지의 상태를 가지고 있음
    - READY - 플레이어 캐릭터가 탐지되면 몬스터 스폰 후 맵의 상태를 **FIGHT**로 전환
    - FIGHT - 스테이지의 문이 닫히며 몬스터와 전투 시작
    - NEXT - **FIGHT** 상태에서 스폰된 몬스터가 모두 없어지면 전환되며 다음 스테이지로 가는 문이 열림
  - 문이 열리는 방향은 비트 플래그 값을 바탕으로 함

비트 플래그	위	아래	왼쪽	오른쪽	결과
방향	+x	-x	-y	+y	
1			-y		-y
2	+x				+x
4				+y	+y
8		-x			-x
3	+x		-y		+x -y
5			-y	+y	-y +y
6	+x			+y	+x +y
7	+x		-y	+y	+x -y +y
9		-x	-y		-x -y
10	+x	-x			+x -x
11	+x	-x	-y		+x -x -y
12		-x		+y	-x +y
13		-x	-y	+y	-x -y +y
14	+x	-x		+y	+x -x +y
15	+x	-x	-y	+y	+x -x -y +y

회고

새로 알게된 점

1. 언리얼의 서브시스템에 대해 알게 되었고, UGameInstanceSubsystem 를 활용해 싱글톤 패턴을 구현했다.
2. Actor 클래스에 있는 TakeDamage 가상함수를 사용하지 않고, 커스텀 TakeDamage 함수를 정의해서 들어온 공격의 유형에 따라 피격 모션과 이펙트, 사운드를 변경할 수 있게 구현 가능함을 알게되었다.
3. git lfs를 이용해 언리얼 프로젝트를 공유할 때, 블루프린트 애셋은 조심해서 다뤄야 한다는 것을 알게 되었다. 협업을 하며 블루프린트 애셋이 지워진적이 많아서 힘들었다.

4. 캐스팅 스킬을 구현할 때 TQueue에 스킬 함수를 객체로 넣어 꺼내서 사용했던 것이 이벤트 큐 패턴과 유사한 것 같다.
5. 상태를 통해 다른 로직을 실행하는 것은 FSM 패턴과 유사한 것 같다.

아쉬운점

1. 리소스 관리를 하지 못하여 용량이 큰점

   → 리소스를 받아오는 과정에서 사용하지 않은 리소스는 제거하고 필요한 것만 사용한 뒤 삭제하는 과정이 필요했을 것 같다.
2. 무기 클래스를 따로 만들어 무기를 전환할 때마다 생성과 제거를 반복하고 있는데, 무기 클래스에 따로 기능을 부여한 것이 없음

   → 차라리 캐릭터에 MeshComponent를 붙여 메쉬를 바꾸는 방법같이 아예 다른 구현 방법을 이용하거나, 오브젝트 풀링 패턴을 사용하여 최적화하는 것이 더 나은 방향같다.