Outland 프로젝트 내 전투 시스템 관련 코드 설명

주의 사항 : 해당 문서에서 프로젝트 내 전투 시스템에 대한 모든 내용을 설명하기엔 너무 규모가 크고 방대하므로 일부 기능만 가져와 설명을 진행하였습니다. 또한, 전투의 진행을 담당하는 클래스 Entity, BattleControl 등은 내용에서 생략되었는데 이는 해당 클래스들이 처리하는 작업이 스킬 효과 계산 및 전투 흐름 제어이기 때문에 관련된 클래스와 설명해야 할 데이터가 많고, 이를 모두 설명하기엔 문서의 양이 너무 커질 것으로 예상해서 생략되었습니다.

---

1. **ABattleController_Base** : 블루프린트로 만들어진 BattleControl의 부모 클래스로 전투에 필요한 단순 작업을 진행한다.

[코드]

```
AActor* ABattleController_Base::LoadEntity(FString entityName, FVector location, FRotator rotation)
{
    if (entityName == "")
        return nullptr;
    FActorSpawnParameters SpawnInfo;
    UClass* Entity = StaticLoadClass(AActor::StaticClass(), nullptr, *FString::Printf(TEXT("/Game/BPscripts_KHH/Data/Entity_Load/%s.%s_C"), *entityName, *entityName));
    return GetWorld()->SpawnActor(Entity, &location, &rotation, SpawnInfo);
}
```

- 대표적으로 내부에 구현된 LoadEntity 함수는 적 캐릭터를 정해진 위치에 소환하도록 할 때 이용한다.
- 전투가 이뤄지는 레벨은 따로 존재하며 해당 레벨에는 미리 캐릭터가 배치될 위치가 정해져 있다.

2. **BattleSystem** : 전투를 함수로 쉽게 호출 가능하도록 하기 위해 제작한 클래스로 UGameInstanceSubsystem을 상속하여 여러 블루프린트에서 쉽게 호출이 가능하다.

[코드]

```
UBattleSystem::UBattleSystem()
{
    static ConstructorHelpers::FObjectFinder<UDataTable> Combat(TEXT("/Game/BPscripts_KHH/Data/CombatData/Combat.Combat"));
    if(Combat.Succeeded())
    {
        BData = Combat.Object;
    }
}
```

- 해당 클래스는 생성자 부분에서 전투 데이터 테이블(미리 설계한 전투 상황(적, 배경 맵 등))을 불러온다.

  ⇨ 여기서 불러온 데이터는 다른 내부 함수에 자주 사용된다.

```cpp
void UBattleSystem::SimpleFight(FString enemy, int enemyOfNumber, FString stage) // 직접 적과 적의 수를 기입해 전투 시작
{
    Fight(enemy, enemyOfNumber, "", stage);
}

void UBattleSystem::ComplexFight(FString enemySet) // 데이터테이블에 저장해둔 전투 상황을 불러와 전투 시작
{
    Fight("", 0, enemySet, TEXT(""));
}

UClass* UBattleSystem::LoadEntityInformation(FString entityName)
{
    if (entityName == "")
        return nullptr;
    UClass* Entity = StaticLoadClass(AActor::StaticClass(), nullptr, *FString::Printf(TEXT("/Game/BPscripts_KHH/Data/Entity_Load/%s.%s_C"), *entityName, *entityName));
    return Entity;
}
```

- 함수 SimpleFight와 ComplexFight는 전투 호출 시 필요한 데이터를 미리 데이터테이블에 저장한 경우와 그렇지 않은 경우 각각 구분해서 전투 호출을 할 수 있도록 하였다.
    - ■ 여기서 더 세부적인 내용은 함수 Fight에서 이어진다.

```cpp
void UBattleSystem::Fight(FString enemy, int enemyOfNumber, FString enemySet, FString stage)
{
    FTimerDelegate TimerDelegate;
    TimerDelegate.BindUFunction(this, FName("LoadFight"), enemy, enemyOfNumber, enemySet, stage);
    FTimerHandle TimerHandle;
    FTimerManager& TimerManager = GetWorld()->GetTimerManager();
    float DelaySeconds = 0.5f;
    if (enemySet == "B00_07")
    {
        PlaySound(LoadObject<USoundWave>(nullptr, TEXT("/Game/Resource_2_KHH/Sound/SE/horn-a.horn-a")));
        DelaySeconds = 1.5f;
        UClass* bossIntro = LoadClass<UUserWidget>(nullptr, *FString::Printf(TEXT("/Game/BPscripts_KHH/Systems/Battle/UI/Boss_Appearace_%s.Boss_Appearace_%s_C"), *enemySet, *enemySet));
        CreateWidget<UUserWidget>(UGameplayStatics::GetPlayerController(GetWorld(), 0), bossIntro)->AddToViewport();
    }
    TimerManager.SetTimer(TimerHandle, TimerDelegate, DelaySeconds, false);
}

void UBattleSystem::PlaySound(USoundWave* Sound)
{
    UGameplayStatics::SpawnSound2D(this, Sound, 1.0f,1.0f, 0.0f, nullptr, true);
}
```

- Fight 함수에서는 전투를 돌입하기 전 효과를 담당하는 부분으로 보스전과 같은 경우 보스에 따라 다른 화면 효과가 나오도록 만들었다.
- 여기선 TimerDelegate를 이용해 일정 시간(DelaySeconds) 후에 함수 LoadFight(전투맵 이동 및 캐릭터 생성 등 처리)를 호출하게 했다.
    - ■ 함수 PlaySound를 이용한 이유는 해당 화면 효과에서 나오는 효과음이 레벨이 전환되어도 끊기지 않게 하기 위해서이다.

```cpp
void UBattleSystem::LoadFight(FString enemy, int enemyOfNumber, FString enemySet, FString stage)
{
    UClass* battleStart = LoadClass<UUserWidget>(nullptr, TEXT("/Game/BPscripts_KHH/Systems/Battle/UI/Enter_Battle.Enter_Battle_C"));
    CreateWidget<UUserWidget>(UGameplayStatics::GetPlayerController(GetWorld(), 0), battleStart)->AddToViewport();

    FTimerHandle TimerHandle;
    FTimerManager& TimerManager = GetWorld()->GetTimerManager();

    USaveDataManager* newData = NewObject<USaveDataManager>();
    USaveDataManager* save = Cast<USaveDataManager>(UGameplayStatics::LoadGameFromSlot(TEXT("PlayerSave"), 0));

    if (save != nullptr)
        newData = save;

    TSharedPtr<struct FBattleCase> battleCase = MakeShared<struct FBattleCase>();

    if (enemySet == "") // 상황 1) 같은 몬스터를 여러 개 등장시킬 때
    {
        for (int j = 0; j < enemyOfNumber; j++)
        {
            battleCase->Enemys.Add(enemy);
        }
        for (int j = 0; j < 5 - enemyOfNumber; j++)
        {
            battleCase->Enemys.Add("");
        }
    }
    else // 상황 2) 전투 상황을 미리 설계한 경우
    {
        FBattleCase* Item = BData->FindRow<FBattleCase>(FName(enemySet), FString(""));
        if (Item != nullptr)
        {
            for (FString obj : Item->Enemys)
            {
                battleCase->Enemys.Add(obj);
            }
            int randomNum = FMath::RandRange(0, Item->StageName.Num()-1);
            stage = Item->StageName[randomNum];
        }
    }

    // 불러올 전투 데이터 등록
    newData->CurrentBattleState = (*battleCase);

    // 현재 레벨 등록 <= 전투 후 복귀하기 위함.
    newData->CurrentLevel = GetWorld()->GetMapName();
    newData->CurrentLevel.RemoveFromStart(GetWorld()->StreamingLevelsPrefix);

    // 현재 상태 저장
    UGameplayStatics::SaveGameToSlot(newData, TEXT("PlayerSave"), 0);

    map = FName(stage);
    // 전투 레벨 호출
    TimerManager.SetTimer(TimerHandle, FTimerDelegate::CreateLambda([this]()
        {
            UGameplayStatics::OpenLevel(this, map);
        }), 2.0f, false);
}
```

- LoadFight 함수는 전투 레벨로 이동함과 동시에 전투 상황에 맞게 캐릭터들을 배치하는 역할을 하는 함수이다.
    - ■ 처음엔 전투의 돌입 효과를 재생하고, 이후 TimerManager를 이용해 일정 시간 대기 후 전투 맵으로 이동하게 되어있다.
    - ■ 전투 맵 이동 전에는 전투 데이터를 불러오고, 이동할 전투 맵을 설정한다.
    - ■ 이 과정에서 battleCase 라는 구조체를 이용하는데 해당 구조체는 전투 맵에서 전투 데이터(적)와 스테이지 정보를 저장하는 일종의 데이터 덩어리이다.
    - ■ 불러온 전투 데이터는 현재의 전투 상황으로 저장하여 전투 맵에서 이용하게 한다.
    - ■ 전투 레벨 이동 전에는 현재 레벨 또한 기록하는데 이는 전투 이후 원래의 레벨로 돌아가야 하기 때문이다.
    - ■ 전투 레벨에 이동하면 BattleControl 클래스가 존재하며 해당 클래스가 전투 진행을 처리한다.

3. **EntityPattern_Auto :** 게임 내 존재하는 적 패턴에 따라 적의 행동을 결정해주는 클래스이다.

```
/*
    < emPattern >
    (1) 평범한: 모든종류의 기술을 비슷한 빈도로 사용한다.
    (2) 성급한: 감정소모가 적은 기술을 높은 빈도로 사용한다.
    (3) 신중한: 감정소모가 큰 키술을 높은 빈도로 사용한다.

    < skPattern >
    (1) 저돌적: 공격기술을 더 많이 사용한다.
    (2) 전략적: 보조기술을 더 많이 사용한다.
    (3) 일반적: 모든종류의 기술을 비슷한 빈도로 사용한다.
    (4) 소극적: 방어를 더 많이 사용한다.
    (5) 지정 : 정한 순서대로 반복. 이때는 emPattern 값을 추가할 값으로 정함. ex) 12  => 2번째 스킬 다음 1번째 스킬
*/
int UEntityPattern_Auto::FindNextSkill(int emPattern, int skPattern, bool noDefense, int randomAddValue)
{
    int i = 0, randomValue;
    int best = 0;


    if (tempSkillIndex.Num() != 0)
        tempSkillIndex.Reset();

    if (skPattern != 0)
    {
        if (skPattern != 5)
        {
            for (FSkill currentSkill : skills)
            {
                srand(time(NULL) + randomAddValue);
                randomValue = rand() % 100;
                switch (skPattern)
                {
                case 1:

                    if (currentSkill.value == 1)
                        tempSkillIndex.Add(i);
                    else if (randomValue < 20 || noDefense)
                        tempSkillIndex.Add(i);
                    break;
                case 2:
                    if (currentSkill.value == 2)
                        tempSkillIndex.Add(i);
                    else if(randomValue < 20 || noDefense)
                        tempSkillIndex.Add(i);
                    break;
                case 3:
                    tempSkillIndex.Add(i);
                    break;
                case 4:
                    if (randomValue > 20 && !noDefense) return -1;
                    else if (randomValue <= 20 || noDefense)
                    {
                        tempSkillIndex.Add(i);
                    }
                    break;
                }
                i++;
            }
        }
```

- FindNextSkill 함수는 매개변수로 어떤 패턴인지 값을 받게 되는데 이때 emPattern 은 스킬 사용에 필요한 감정(마나와 비슷한 개념)에 따라 기술을 선택하는 패턴을 말하고, skPattern 은 스킬 형태에 따라 기술을 선택하는 패턴을 말한다.
    - FindNextSkill 함수는 여기서 받은 매개변수를 이용해 적이 다음으로 어떤 스킬을 사용할지 적절히 결정하게 된다.