プロジェクト概要

- ジャンル：アクションゲーム
- 使用エンジン：Unreal Engine 5.6
- 制作期間：3か月
- 開発人数：1名
- 担当範囲：ゲームプログラム全般

  （戦闘、AI、マップ生成、データ管理、最適化）

プレイ動画・ポートフォリオサイト

- プレイ動画：https://youtu.be/awJbMQSwKiA
- ポートフォリオサイト：http://famicom1983.dothome.co.kr/portfolios/index.jsp

制作目標

コンボ攻撃とパリィを軸に、スピーディーで爽快感のあるアクションゲームを目標として制作しました。

プレイするたびにマップ構造と敵の配置が変化し、異なる戦闘状況を楽しめるようにしています。

プレイヤーアクション

プレイヤーには、以下のアクションを実装しました。

- 弱攻撃・強攻撃
- 弱攻撃と強攻撃を組み合わせたコンボ
- SPを消費する属性攻撃
- 敵の攻撃を受け流すパリィ
- ターゲットを追跡するロックオン

属性攻撃を使用するとSPを消費します。SPは攻撃後、一定時間が経過すると徐々に回復します。

コンボ入力システム


```
FVector InputDir =
    GetCurrentInputDirection();;

if (InputDir.IsNearlyZero())
{
    InputDir = GetActorForwardVector();
}

InputDir.Normalize();
SetActorRotation(InputDir.Rotation());

if (AtkName.ToString()[0] == 'A')
{
    LaunchCharacter(InputDir * 1000, true, true);
    AnimInst->Montage_Play(AttackMontage);
    AnimInst->Montage_JumpToSection(AtkName);
}
else if (AtkName.ToString()[0] == 'H')
{
    LaunchCharacter(InputDir * 1500, true, true);
    AnimInst->Montage_Play(HeavyAttackMontage);
    AnimInst->Montage_JumpToSection(AtkName);
}
else if (AtkName.ToString()[0] == 'C')
{
    LaunchCharacter(InputDir * 1500, true, true);
    AnimInst->Montage_Play(ComboAttackMontage);
    AnimInst->Montage_JumpToSection(AtkName);
}

AnimInst->SetIgnoreRootMotion(true);
AtkQueue.Dequeue(AtkName);
```

弱攻撃と強攻撃を組み合わせたテンポの速いコンボにおいて、入力の取りこぼしを防ぎながら、指定したタイミングでのみ次の攻撃へ派生できる仕組みが必要でした。

Animation Notifyを利用してコンボ入力の受付期間を制御し、受付期間中に入力された攻撃情報を入力バッファへ保存しました。

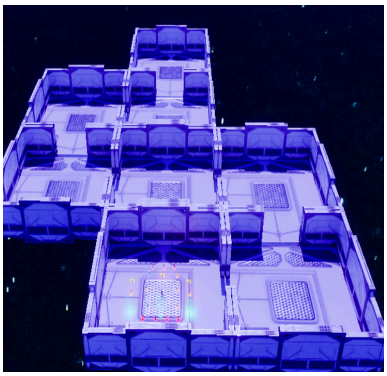
現在の攻撃モーションが終了すると、保存された入力を取り出し、その入力に対応する次の攻撃へ移行します。

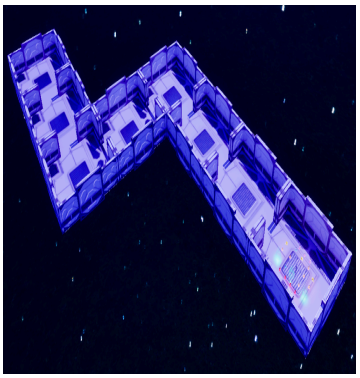
設計上の振り返り

現在の実装では、複数の入力を受け取った順序で処理するためにキューを使用しています。一方、次の攻撃を一つだけ予約する仕様であれば、単一の入力変数でより簡潔に実装できます。

仕様に応じて適切な入力バッファの構造を選択する必要があると学びました。

グリッドベースのランダムルーム生成





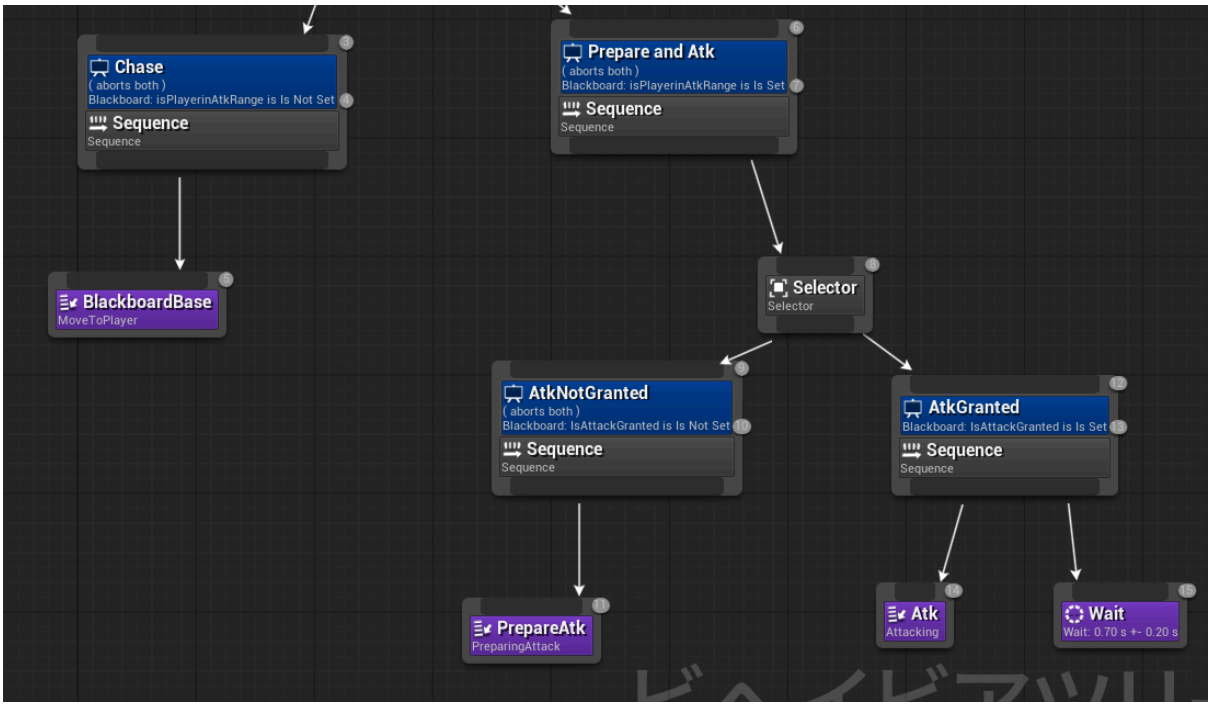
```cpp
void URoomSubsystem::GenerateGridLocation(FIntPoint Location, int32 count)
{
    if (count >= MaxRoomCnt)
    {
        CreateRooms();
        return;
    }

    TArray<FIntPoint> tempArr;
    tempArr.Add(FIntPoint(Location.X - 1, Location.Y));
    tempArr.Add(FIntPoint(Location.X + 1, Location.Y));
    tempArr.Add(FIntPoint(Location.X, Location.Y - 1));
    tempArr.Add(FIntPoint(Location.X, Location.Y + 1));

    for (int i = 0; i < SpawnedRoomsLocations.Num(); i++)
    {
        if (FIntPoint(Location.X - 1, Location.Y) == SpawnedRoomsLocations[i])
        {
            tempArr.Remove(FIntPoint(Location.X - 1, Location.Y));
        }
        if (FIntPoint(Location.X + 1, Location.Y) == SpawnedRoomsLocations[i])
        {
            tempArr.Remove(FIntPoint(Location.X + 1, Location.Y));
        }
        if (FIntPoint(Location.X, Location.Y - 1) == SpawnedRoomsLocations[i])
        {
            tempArr.Remove(FIntPoint(Location.X, Location.Y - 1));
        }
        if (FIntPoint(Location.X, Location.Y + 1) == SpawnedRoomsLocations[i])
        {
            tempArr.Remove(FIntPoint(Location.X, Location.Y + 1));
        }
    }

    FIntPoint temp = FIntPoint(0, 0);

    int32 RandomIndex = FMath::RandRange(0, tempArr.Num() - 1);

    temp = tempArr[RandomIndex];

    SpawnedNeighborRooms.Add(FRoomData{ Location, temp });

    SpawnedRoomsLocations.Add(temp);

    GenerateGridLocation(temp, ++count);
}
```

FIntPoint(0, 0)を開始地点とする2次元グリッドを構成し、隣接するセルの中から次のルーム座標をランダムに決定します。

生成済みの座標を管理することでルームの重複を防ぎ、隣接するルームの方向情報を基に出入口を接続しています。

また、ルーム内に出現する敵もランダムに選択することで、プレイするたびに異なるマップ構造と戦闘状況が発生するようにしました。

敵AI

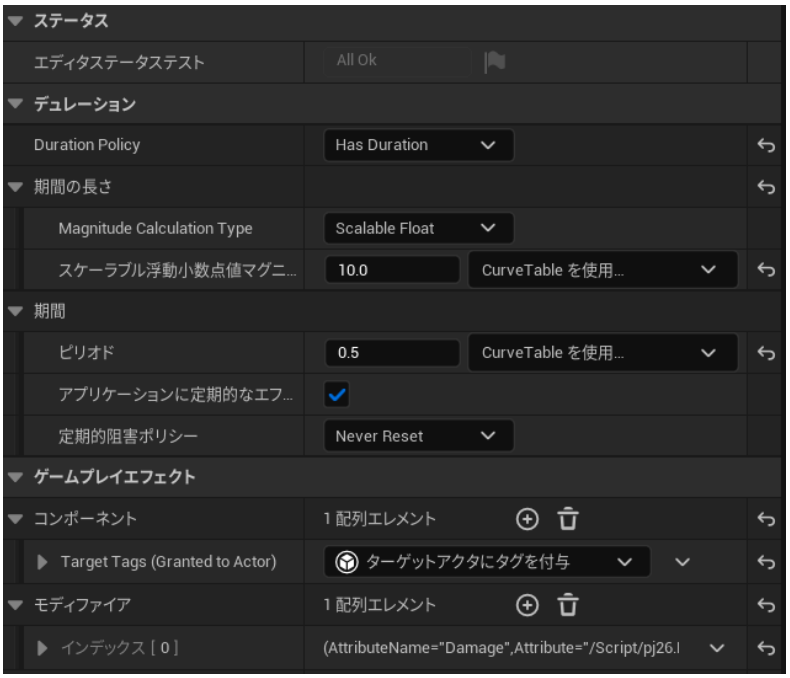


AI Perceptionを使用してプレイヤーを感知し、Behavior Treeによって追跡、移動、待機、攻撃の行動を制御しています。

敵が攻撃可能な距離まで接近すると、EQSを使用してプレイヤー周辺の候補地点を評価し、適切な位置へ移動して待機します。

攻撃範囲内にいる敵の中から攻撃を行う敵を選択し、攻撃権を与える仕組みを実装しました。これにより、複数の敵が同時に攻撃する状況を抑えながら、敵が順番に攻撃を仕掛ける戦闘を実現しました。

GASを利用した状態異常



属性攻撃が命中すると、攻撃属性に応じた状態異常を敵へ付与します。

- 火属性：一定時間、継続的にダメージを与える
- 雷属性：一定時間、移動速度を低下させる
- 風属性：一定時間、防御力を低下させる

状態異常の効果量と持続時間はGameplay Effectで管理し、戦闘処理と状態異常処理を分離しました。

データドリブンなゲームデータ管理

	行の名前	Enemy Name	Atk Power	Def	HP	SP	Drop Item	Drop Item 2	Drop Item 3	Item Drop Percenta
1	WeakEnemy	WeakEnemy	5.000000	5.000000	15.000000	10.000000	HPItem	AtkItem	DefItem	60.000000
2	NormalEnemy	NormalEnemy	5.000000	10.000000	60.000000	50.000000	HPItem	AtkItem	DefItem	50.000000
3	DashEnemy	DashEnemy	20.000000	20.000000	50.000000	10.000000	HPItem	AtkItem	DefItem	50.000000
4	LongDistAtkEnen	LongDistAtkEnemy	5.000000	1.000000	1.000000	1.000000	HPItem	AtkItem	DefItem	80.000000
5	Boss	Boss	10.000000	20.000000	550.000000	150.000000	HPItem	AtkItem	DefItem	0.000000

```
USTRUCT(BlueprintType)
파생된 블루프린트 클래스 0개
struct FEnemyInfo : public FTableRowBase
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    FName EnemyName;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float AtkPower;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float Def;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float HP;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float SP;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    FName DropItem1;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    FName DropItem2;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    FName DropItem3;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float ItemDropPercentage;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    TSubclassOf<AEnemyOrigin> EnemyClass;
};
```

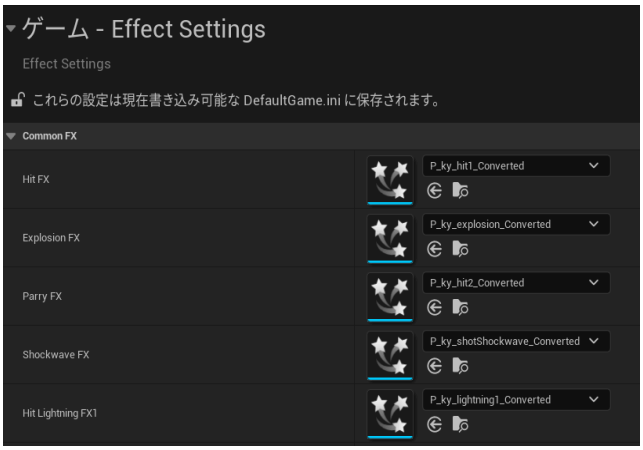
```
struct FPlayerInfo : public FTableRowBase
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    FName PlayerName;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float AtkPower;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float SPAtkPower;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float Def;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float HP;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float SP;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float Speed;
    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float ThunderDmg;

    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float WindDmg;

    UPROPERTY(EditAnywhere, BlueprintReadOnly)
    0 Blueprints에서 변경됨
    float FireDmg;

};
```

敵、プレイヤー、アイテムの設定データをゲームロジックから分離するため、CSVとDataTableを利用しました。

各データに対応するFTableRowBase構造体を定義し、最大HP、攻撃力、移動速度、アイテムの効果量などの静的データを行単位で管理しています。

これにより、C++コードを変更・再コンパイルすることなく、ゲームバランスを調整できる構造にしました。

Niagaraアセットのプリロードとコンポーネントプール




```
LoadedHitFX = FXSettings->HitFX.Get();
LoadedExplosionFX = FXSettings->ExplosionFX.Get();
LoadedParryFX = FXSettings->ParryFX.Get();
LoadedShockwaveFX = FXSettings->ShockwaveFX.Get();
LoadedLightningFX = FXSettings->HitLightningFX3.Get();
LoadedWindFX = FXSettings->HitWindFX3.Get();
LoadedDistortionShockFX = FXSettings->DistortionFX.Get();
```

```
TArray<FSoftObjectPath> AssetsToLoad;
if (!FXSettings->HitFX.IsNull()) AssetsToLoad.Add(FXSettings->HitFX.ToSoftObjectPath());
if (!FXSettings->ExplosionFX.IsNull()) AssetsToLoad.Add(FXSettings->ExplosionFX.ToSoftObjectPath());
if (!FXSettings->ParryFX.IsNull()) AssetsToLoad.Add(FXSettings->ParryFX.ToSoftObjectPath());
if (!FXSettings->ShockwaveFX.IsNull()) AssetsToLoad.Add(FXSettings->ShockwaveFX.ToSoftObjectPath());
if (!FXSettings->HitLightningFX3.IsNull()) AssetsToLoad.Add(FXSettings->HitLightningFX3.ToSoftObjectPath());
if (!FXSettings->HitWindFX3.IsNull()) AssetsToLoad.Add(FXSettings->HitWindFX3.ToSoftObjectPath());
if (!FXSettings->DistortionFX.IsNull()) AssetsToLoad.Add(FXSettings->DistortionFX.ToSoftObjectPath());
```

戦闘中に頻繁に使用するNiagara Systemアセットを非同期でプリロードし、ロード完了後の参照をキャッシュしています。

これにより、エフェクトを初めて再生する際に発生する可能性のあるロード遅延を抑えました。

アクターオブジェクトプール

```cpp
void UEffectActorSubsystem::PrewarmPool(TSubclassOf<APooledActor> ActorClass, int32 Count)
{
    if (!GetWorld() || !ActorClass)
    {
        return;
    }

    FActorPool& Pool = Pools.FindOrAdd(ActorClass);

    for (int32 i = 0; i < Count; ++i)
    {
        APooledActor* NewActor = GetWorld()->SpawnActor<APooledActor>(ActorClass, FTransform::Identity);
        if (NewActor)
        {
            NewActor->DeactivateToPool();
            Pool.Actors.Add(NewActor);
        }
    }
}
```

```cpp
void UEffectActorSubsystem::ClearPool()
{
    for (auto& PoolPair : Pools)
    {
        FActorPool& Pool = PoolPair.Value;

        for (APooledActor* Actor : Pool.Actors)
        {
            if (IsValid(Actor))
            {
                Actor->Destroy();
            }
        }

        Pool.Actors.Empty();
    }

    Pools.Empty();
}
```

頻繁に生成・破棄されるアクターには、独自のオブジェクトプールを適用しました。

ステージの初期化時に一定数のアクターをあらかじめ生成し、必要になった際にプールから取得します。使用後は状態を初期化してプールへ返却し、次回の生成時に再利用できるようにしました。

これにより、ゲームプレイ中に繰り返されるアクターのSpawn・Destroy処理を抑える構造にしました。
