システム全体構造

UPlayerInputBinder (入力管理コンポーネント)

├── Enhanced Input System への接続

├── IPlayerInputReceiver (インターフェイス経由)

└── APlayerCharacter への委譲

主要な役割

UPlayerInputBinder

- 役割: 入力の交通整理役
- 主な機能:
    - 9種類の入力アクションを受付
    - 入力を IPlayerInputReceiver に委譲
    - MappingContext の自動登録
    - 入力先の動的切り替え

IPlayerInputReceiver（インターフェイス）

- 役割: 入力を受け取る契約書
- 仕組み:
    - 9つのメソッド定義（OnMove, OnJump等）
    - APlayerCharacter がこのインターフェイスを実装
    - InputBinder はインターフェイス経由でのみ通信

APlayerCharacter

- 役割: 入力の実際の受け手
- 主な機能:
    - IPlayerInputReceiver を実装
    - 9つの入力メソッドを実装
    - 各コンポーネントに処理を振り分け

データの流れ

初期化フロー

1. APlayerCharacter::SetupPlayerInputComponent()

  ↓

2. InputBinder->BindInputs(EnhancedInput, this)

  ↓

3. PlayerController から Enhanced Input Subsystem を取得

  ↓

4. DefaultMappingContext を登録

  ↓

5. InputReceiver = this (APlayerCharacter を保存)

  ↓

6. 9種類の InputAction をバインド

  - MoveAction → HandleMove

  - JumpAction → HandleJump

  - LookAction → HandleLook

  - BoostAction → HandleBoostSkill

  - SpecialAction → HandleSlowSkill

  - Action → HandleReplay

  - ReplayToWorldAction → HandleReplayToWorld

  - OnInteractAction → HandleOnInteractAction

  - OpenMenuAction → HandleOpenMenu

入力処理フロー（例: 移動入力）

1. プレイヤーが WASD キーを押す

  ↓

2. Enhanced Input System

「MoveAction が発動」

↓

3. UPlayerInputBinder::HandleMove(Value)

入力を受信

↓

4. if (InputReceiver)

InputReceiver->OnMove(Value);

↓

5. APlayerCharacter::OnMove(Value)

↓

6. StateManager->GetCurrentState()->Movement(Value)

現在の状態に処理を委譲

9種類の入力アクションとデータフロー

プレイヤー入力	InputAction	InputBinder	PlayerCharacter	最終処理先
WASD	MoveAction	HandleMove	OnMove	StateManager
Space	JumpAction	HandleJump	OnJump	StateManager

マウス移動	LookAction	HandleLook	OnLook	CameraControl
Shift	BoostAction	HandleBoostSkill	OnBoost	StateManager → BoostComponent
Ctrl	SpecialAction	HandleSlowSkill	OnSlowAction	TimeManagerSubsystem
E	Action	HandleReplay	OnReplayAction	StateManager
R	ReplayToWorldAction	HandleReplayToWorld	OnReplayToWorldAction	TimeManagerSubsystem
F	OnInteractAction	HandleOnInteractAction	OnInteractAction	Delegate経由
Esc	OpenMenuAction	HandleOpenMenu	OpenMenu	UIManager

設計上の特徴

1. 委譲パターン

cpp

```
// InputBinder側（仲介するだけ）
void UPlayerInputBinder::HandleMove(const FInputActionValue& Value)
{
    if (InputReceiver)
```

```
        InputReceiver->OnMove(Value);  // PlayerCharacterに丸投げ
}

// PlayerCharacter側（実際の処理）
void APlayerCharacter::OnMove(const FInputActionValue& Value)
{
    if (StateManager != nullptr)
        StateManager->GetCurrentState()->Movement(Value);
}
```

- 設計意図: InputBinder は入力の振り分けのみ担当
- メリット: 責任の明確な分離

2. インターフェイスベース通信

cpp

```
// InputBinder はインターフェイスのみを知る
TScriptInterface<IPlayerInputReceiver> InputReceiver;

// BindInputs で受け手を登録
void BindInputs(UEnhancedInputComponent* InputComponent,
                TScriptInterface<IPlayerInputReceiver> Receiver)
{
    InputReceiver = Receiver;  // インターフェイス経由で保存
    // バインド処理...
}
```

- 目的: InputBinder は APlayerCharacter の詳細を知らない
- 効果: プレイヤー以外（リプレイ、観戦モード）でも同じバインダーを使用可能

3. 動的な入力先切り替え

cpp

// 通常プレイモード

InputBinder->BindInputs(EnhancedInput, PlayerCharacter);

// リプレイ再生モードへ切り替え

InputBinder->UnbindInputs();

InputBinder->BindInputs(EnhancedInput, ReplayController);

// 元に戻す

InputBinder->UnbindInputs();

InputBinder->BindInputs(EnhancedInput, PlayerCharacter);

```

- **仕組み**: InputReceiver を差し替えるだけ

- **用途**: ゲームモード切り替え、カットシーン中の入力無効化

---

## PlayerCharacter 内部での処理振り分け

```

APlayerCharacter が入力を受け取った後の流れ

OnMove(Value)

  └→ StateManager->GetCurrentState()->Movement(Value)

     ├→ DefaultState: 通常移動

├→ WallRunState: 壁に沿って移動

├→ LandingState: 移動不可

└→ RewindState: 移動不可

OnJump(Value)

└→ StateManager->GetCurrentState()->Jump(Value)

├→ DefaultState: 通常ジャンプ

├→ WallRunState: 壁ジャンプ

└→ その他: ジャンプ不可

OnLook(Value)

└→ CameraControl->ProcessLookInput(Value)

└→ カメラ回転処理（状態に関係なく常に動作）

OnBoost(Value)

└→ StateManager->GetCurrentState()->BoostAction(Value)

└→ DefaultState: PlayerCharacter->PlayBoost()

└→ BoostComponent->Boost()

OnSlowAction(Value)

└→ TimeManagerSubsystem->StartSlowMotion(0.1)

└→ PostProcessEffect 適用

└→ 効果音再生

OnReplayAction(Value)

└→ StateManager->GetCurrentState()->RePlayAction(Value)

OnReplayToWorldAction(Value)

└→ TimeManagerSubsystem->RewindToWorld(10)

OnInteractAction(Value)

└→ OnInteractPressed.Broadcast(this)

└→ 周囲のオブジェクトに通知

OpenMenu(Value)

└→ UIHandle::ShowWidget(this, EWidgetCategory::Menu, "Menu")

実装の工夫ポイント

工夫1: null チェックの徹底

cpp

```
void APlayerCharacter::OnMove(const FInputActionValue& Value)
{
    if (StateManager != nullptr && StateManager->GetCurrentState() != nullptr)
    {
        StateManager->GetCurrentState()->Movement(Value);
    }
}
```

- 目的: 安全性確保、クラッシュ防止
- 実装箇所: 全ての入力メソッド

工夫2: 状態による処理の分岐

cpp

```
// PlayerCharacter は状態を意識せず、StateManager に委譲

StateManager->GetCurrentState()->BoostAction(Value);


// 各状態が独自の処理を実装

DefaultState::BoostAction() → ブースト発動

WallRunState::BoostAction() → 何もしない（壁走り中はブースト不可）

LandingState::BoostAction() → 何もしない（硬直中はブースト不可）
```

- 設計: 状態パターンの活用
- メリット: PlayerCharacter のコードがシンプルに

工夫3: デリゲート経由の疎結合

cpp

```
// PlayerCharacter は誰が反応するか知らない

void APlayerCharacter::OnInteractAction(const FInputActionValue& Value)

{

    OnInteractPressed.Broadcast(this);

}


// 周囲のオブジェクトが勝手に反応

void ADoor::BeginPlay()

{

    PlayerCharacter->SubscribeToInteract(this, "OnPlayerInteract");

}


void ADoor::OnPlayerInteract(APlayerCharacter* Player)
```

```
{
    //ドアを開ける処理
}
```

- 効果: PlayerCharacter がドアやスイッチの存在を知らなくて良い

この設計がもたらす価値

✅ 拡張性

- 新しい入力アクションの追加が容易
- 既存コードに影響なし

✅ 再利用性

- InputBinder は他のプロジェクトでも使用可能
- インターフェイスを実装するだけで適用可能

✅ テスタビリティ

- モック Receiver を作成してテスト可能
- 実際のゲームを起動せずに入力処理をテスト

✅ 保守性

- 入力処理の流れが明確
- 各クラスの責任が明確

設計意図

入力の受付と処理を完全分離することで、

- **InputBinder**:「どの入力が来たか」のみを管理
- **PlayerCharacter**:「その入力で何をするか」のみを実装

この分離により、柔軟で保守性の高いシステムを実現しました。