システム概要

Unreal Engine 5で実装した、時間操作・パルクール・壁走り・ブーストなど複数の特殊能力を持つプレイヤーキャラクターシステムです。状態パターンとコンポーネント指向設計により、複雑な機能群を疎結合で統合し、高い拡張性と保守性を実現しました。

核心的な設計思想

1. コンポーネント指向アーキテクチャによる機能分離

キャラクターの各機能を独立したコンポーネントとして実装し、APlayerCharacterはそれらを統合する「ハブ」として機能します。

```
APlayerCharacter (統合ハブ)
    ├── UPlayerStateManager (状態管理)
    ├── UPlayerInputBinder (入力管理)
    ├── UTimeManipulatorComponent (時間操作)
    ├── UParkourComponent (パルクール)
    ├── UWallRunComponent (壁走り)
    ├── UBoostComponent (ブースト)
    └── UPlayerCameraControlComponent (カメラ制御)
```

この設計を選んだ理由:

- 各機能が独立して開発・テスト可能
- 機能の追加・削除が容易（コンポーネントの付け替えのみ）
- 他のキャラクタータイプでもコンポーネントを再利用可能
- 責務が明確で保守性が高い

---

2. 状態パターンによる挙動制御

キャラクターの状態（通常・壁走り・着地硬直・巻き戻し等）によって入力への反応を動的に変更します。

```
【状態遷移図】
DefaultState (通常状態)
    ├→ WallRunState (壁走り中)
    ├→ LandingState (着地硬直中)
    ├→ RewindState (巻き戻し中)
    └→ ParkourState (パルクール中)
```

入力の状態別処理例:

入力	DefaultState	WallRunState	LandingState	RewindState
移動(WASD)	通常移動	壁に沿って移動	移動不可	移動不可
ジャンプ(Space)	通常ジャンプ	壁ジャンプ	ジャンプ不可	ジャンプ不可
ブースト(Shift)	ブースト発動	ブースト不可	ブースト不可	ブースト不可
パルクール(E)	パルクール実行	パルクール不可	パルクール不可	パルクール不可

実装の核心:

cpp
```
void APlayerCharacter::OnMove(const FInputActionValue& Value)
{
    if (StateManager && StateManager->GetCurrentState())
    {
        // 現在の状態に処理を委譲
        StateManager->GetCurrentState()->Movement(Value);
    }
}
```

この設計により、PlayerCharacter自体は状態を意識せず、常に同じ方法で入力を処理します。

---

3. インターフェイス駆動設計による疎結合

3つのインターフェイスを実装し、外部システムとの依存関係を最小化しました。

cpp
```
class APlayerCharacter :
    public IPlayerInfoProvider,      // 情報提供者
    public IPlayerInputReceiver,     // 入力受信者
    public ITimeControllable         // 時間制御可能
```

各インターフェイスの役割:

IPlayerInputReceiver - 入力の受け口
```
役割: 入力システムからの入力を受け取る契約
実装メソッド:
  - OnMove() - 移動入力
  - OnJump() - ジャンプ入力
```

- OnLook() - 視点入力
- OnBoost() - ブースト入力
- OnReplayAction() - リプレイ入力
... 全9種類
```

**IPlayerInfoProvider** - 情報の提供者
```
役割: 他のシステムがプレイヤー情報を取得する契約
実装メソッド:
- GetCamera() - カメラコンポーネント取得
- ChangeState() - 状態変更
- IsRewinding() - 巻き戻し中か判定
- PlayBoost() - ブースト実行
- PlayParkour() - パルクール実行
... etc
```

**ITimeControllable** - 時間制御の契約
```
役割: 時間操作機能の標準インターフェイス
実装メソッド:
- StartTimeRecording() - 記録開始
- StopTimeRecording() - 記録停止
- StartTimeRewind() - 巻き戻し開始
- IsRecording() - 記録中か判定
```

**この設計の効果:**
- InputBinderはIPlayerInputReceiverしか知らない（PlayerCharacterの詳細を知らない）
- UIシステムはIPlayerInfoProviderを通じてのみアクセス
- 時間操作システムは他のオブジェクトにも適用可能

---

## システム全体構造
```
【レイヤー構造】

入力層
  └→ UPlayerInputBinder
       └→ IPlayerInputReceiver

キャラクター層
  └→ APlayerCharacter (統合ハブ)
       ├→ IPlayerInfoProvider (外部への情報提供)
       ├→ IPlayerInputReceiver (入力受信)
       └→ ITimeControllable (時間制御)
```
```

状態管理層
  └→ UPlayerStateManager
      ├→ DefaultState (通常)
      ├→ WallRunState (壁走り)
      ├→ LandingState (着地硬直)
      └→ RewindState (巻き戻し)

機能コンポーネント層
  ├→ UTimeManipulatorComponent (時間記録・巻き戻し)
  ├→ UParkourComponent (パルクール)
  ├→ UWallRunComponent (壁走り)
  ├→ UBoostComponent (ブースト)
  └→ UPlayerCameraControlComponent (カメラ制御)

エフェクト層
  ├→ PostProcessEffectHandle (ポストプロセス)
  ├→ SoundHandle (サウンド)
  └→ UIHandle (UI表示)

サブシステム層
  └→ UTimeManagerSubsystem (グローバル時間制御)
```

---

## データフロー

### 初期化フロー
```
1. APlayerCharacter::Constructor()
   ├→ 全コンポーネントを CreateDefaultSubobject で生成
   │   ├─ StateManager
   │   ├─ InputBinder
   │   ├─ TimeManipulator
   │   ├─ ParkourComponent
   │   ├─ WallRunComponent
   │   ├─ BoostComponent
   │   └─ CameraControl
   │
   └→ CameraControl を head ソケットにアタッチ

2. BeginPlay()
   ├→ StateManager->Init() (初期状態を設定)
   │
   ├→ TimeManipulator のデリゲートをバインド
   │   ├─ OnRewindStarted → OnRewindStarted()
   │   ├─ OnRewindStopped → OnRewindStopped()
```
```

```
   │   └─ OnRecordingStopped → OnRewindStopped()
   │
   ├─→ ParkourComponent のデリゲートをバインド
   │   ├─ OnParkourStarted → OnParkourStarted()
   │   └─ OnParkourEnded → OnParkourEnded()
   │
   └─→ TimeManagerSubsystem のデリゲートをバインド
       └─ OnSlowStopped → OnSlowStopped()

3. SetupPlayerInputComponent()
   └─→ InputBinder->BindInputs(EnhancedInput, this)
       └─→ 9種類の入力アクションを IPlayerInputReceiver にバインド
```

入力処理の完全なフロー（移動入力の例）

```
【フェーズ1: 入力検出】
プレイヤーが WASD キーを押下
  ↓
Enhanced Input System が MoveAction を検出
  ↓
UPlayerInputBinder::HandleMove(Value) が呼ばれる

【フェーズ2: インターフェイス経由の委譲】
InputBinder が保持する InputReceiver (= this) に委譲
  ↓
IPlayerInputReceiver::OnMove(Value)
  ↓
APlayerCharacter::OnMove(Value) が実行される

【フェーズ3: 状態への委譲】
StateManager->GetCurrentState()->Movement(Value)
  ↓
現在の状態に応じた処理が実行される

【フェーズ4: 状態別の実装】
DefaultState::Movement(Value)
  ├─→ CharacterMovementComponent に移動入力を伝達
  ├─→ アニメーションを更新
  └─→ 必要に応じて状態遷移を判定

WallRunState::Movement(Value)
  ├─→ 壁の法線ベクトルに基づいて移動方向を計算
  ├─→ 壁に沿った移動を実行
  └─→ 壁から離れたら DefaultState に遷移
```

```
LandingState::Movement(Value)
  └→ 何もしない（硬直中は移動不可）
```

主要機能の実装詳細

1. 時間操作システム

時間記録の仕組み:
```
【記録開始】
1. プレイヤーがCtrlキーを押下
   ↓
2. OnSlowAction() が呼ばれる
   ↓
3. TimeManagerSubsystem->StartSlowMotion(0.1)
   └→ グローバル時間を0.1倍速に設定

4. ApplySlowMotionPostProcess()
   └→ 画面に青い色調エフェクトを適用

5. USoundHandle::PlaySE("SlowTime", true)
   └→ ループするスロー効果音を再生

【記録停止】
1. プレイヤーが再度Ctrlキーを押下
   ↓
2. TimeManager->ResetTimeDilation()
   └→ 時間を通常速度に戻す

3. RemoveSlowMotionPostProcess()
   └→ エフェクトを削除

4. USoundHandle::StopSE("SlowTime")
   └→ 効果音を停止
```

巻き戻しの仕組み:
```
【個人巻き戻し】
1. プレイヤーがEキーを押下（DefaultState時のみ）
   ↓
2. DefaultState::RePlayAction()
   └→ PlayerCharacter->StartTimeRewind(3.0f)

3. TimeManipulator->StartRewind(3.0f)
```

```
   ├→ 記録された位置・回転を逆再生
   ├→ OnRewindStarted デリゲートを発火
   └→ 3秒間巻き戻し

4. OnRewindStarted() コールバック
   └→ ApplyRewindPostProcess()
      └→ 画面に紫の渦巻きエフェクトを適用

5. 巻き戻し完了
   ↓
6. OnRewindStopped() コールバック
   ├→ RemoveRewindPostProcess()
   └→ StateManager->GetCurrentState()->SkillActionStop()
      └→ 状態をリセット

【ワールド巻き戻し】
1. プレイヤーがRキーを押下
   ↓
2. OnReplayToWorldAction()
   └→ TimeManagerSubsystem->RewindToWorld(10)
      └→ ワールド全体を10秒巻き戻し
```

設計の工夫:

- 個人巻き戻しとワールド巻き戻しを分離
- デリゲートによりエフェクトとコア機能を疎結合化
- サブシステムでグローバルな時間制御を一元管理

---

2. パルクールと壁走りの連携

パルクール開始時の壁走り無効化:

cpp

```cpp
void APlayerCharacter::OnParkourStarted()
{
    if (WallRunComponent)
    {
        // 壁走り検出を無効化
        WallRunComponent->SetDetectionEnabled(false);

        // すでに壁走り中なら強制終了
        if (WallRunComponent->IsWallRunning())
        {
            WallRunComponent->ExitWallRun();
        }
    }
```

```
}

void APlayerCharacter::OnParkourEnded()
{
    if (WallRunComponent)
    {
        // パルクール終了後、壁走り検出を再開
        WallRunComponent->SetDetectionEnabled(true);
    }
}
```

なぜこの実装が必要だったか:
- パルクール中に壁走りが発動すると、アニメーションが競合する
- 壁走り中にパルクールを実行した場合、壁走りを優先的に終了させる
- パルクール終了後、自動的に壁走り検出を再開

デリゲートを使った理由:
- ParkourComponentはWallRunComponentの存在を知らない
- PlayerCharacterが仲介役として両者を連携
- 疎結合を維持しつつ、適切なタイミングで制御

---

3. 着地硬直システム

落下距離に応じた動的な硬直時間:
```
【着地検出フロー】
1. Landed() イベントが発火
   ↓
2. 現在の状態が DefaultState かチェック
   if (!StateManager->IsStateMatch(EPlayerStateType::Default))
       return;  // 壁走り中等は硬直なし

3. DefaultState から落下距離を計算
   FallDistance = LastGroundHeight - CurrentHeight

4. 硬直時間を計算（DefaultStateの計算式）
   例: 500cm落下 → 0.5秒硬直
       1000cm落下 → 1.0秒硬直

5. LandingState に遷移
   StateManager->ChangeState(EPlayerStateType::Landing)

6. 硬直時間を設定
   LandingState->SetLagDuration(LagDuration)
```

7. カメラ振動エフェクト
   CameraControl->PlayHeavyShake()

8. 硬直時間経過後、自動的に DefaultState に戻る

この設計の特徴:

- 落下距離を動的に計算し、リアルな硬直を実現
- DefaultStateが前回の地面の高さを記録
- 壁走り中や巻き戻し中は硬直が発生しない
- カメラ振動により視覚的なフィードバック

---

4. カメラ制御の動的切り替え

頭追従モードと安定モードの切り替え:

cpp
```
void APlayerCharacter::SetCameraAttachToHead_Implementation(bool bAttachToHead)
{
    // 既に同じ状態なら処理をスキップ
    if (bIsCameraAttachedToHead == bAttachToHead)
        return;

    bIsCameraAttachedToHead = bAttachToHead;

    if (bAttachToHead)
    {
        // 頭のソケットにアタッチ（揺れあり）
        mesh->SetOwnerNoSee(false);  // メッシュ表示
        CameraControl->AttachToComponent(
            mesh,
            FAttachmentTransformRules::SnapToTargetNotIncludingScale,
            HeadSocketName  // "head" ソケット
        );
        CameraControl->SetRelativeLocation(SavedLocalLocation);
    }
    else
    {
        // ルートコンポーネントにアタッチ（揺れなし）
        mesh->SetOwnerNoSee(true);  // メッシュ非表示
        CameraControl->AttachToComponent(
            GetRootComponent(),
            FAttachmentTransformRules::SnapToTargetNotIncludingScale
        );
        CameraControl->SetRelativeLocation(FVector(0, 0, 60));
    }
```

```
    // CameraControl にも通知
    CameraControl->SetCameraAttachedToHead(bAttachToHead);
}
```

使用シーン:

- 頭追従モード: カットシーン、リアルなFPS体験
- 安定モード: 通常プレイ、激しいアクション中

設計の意図:

- 状況に応じてカメラの挙動を動的に変更
- インターフェイス経由で外部から制御可能
- カットシーンシステム等が自由に切り替えられる

主要な実装上の工夫

工夫1: デリゲートによる機能間連携

実装例: インタラクションシステム

cpp

```
// デリゲートの宣言（PlayerCharacter.h）
UPROPERTY(BlueprintAssignable, Category = "Interaction")
FOnInteractPressed OnInteractPressed;

// 入力時にデリゲートを発火
void APlayerCharacter::OnInteractAction(const FInputActionValue& Value)
{
    OnInteractPressed.Broadcast(this);
}

// 外部オブジェクトが登録
void APlayerCharacter::SubscribeToInteract(UObject* Object, FName FunctionName)
{
    if (Object)
    {
        FScriptDelegate Delegate;
        Delegate.BindUFunction(Object, FunctionName);
        OnInteractPressed.Add(Delegate);
    }
}
```

活用例:

```cpp
//ドアクラス
void ADoor::BeginPlay()
{
    //プレイヤーのインタラクションに登録
    PlayerCharacter->SubscribeToInteract(this, "OnPlayerInteract");
}

void ADoor::OnPlayerInteract(APlayerCharacter* Player)
{
    //プレイヤーが近くにいるかチェック
    if (IsPlayerNearby(Player))
    {
        OpenDoor();
    }
}
```

この設計の利点:

- PlayerCharacterはドアやスイッチの存在を知らない
- マップ上の全てのインタラクト可能オブジェクトが自動的に反応
- 新しいオブジェクトタイプを追加してもPlayerCharacterは変更不要

工夫2: ポストプロセスエフェクトの一元管理

効果ごとに独立したエフェクト管理:

```cpp
// 記録中エフェクト
void ApplyRecordingPostProcess()
{
    UPostProcessEffectHandle::ActivateEffect(
        this,
        EPostProcessEffectTag::Recording,
        true
    );
}

// 巻き戻し中エフェクト
void ApplyRewindPostProcess()
{
    UPostProcessEffectHandle::ActivateEffect(
        this,
        EPostProcessEffectTag::Rewinding,
        true
    );
```

```
}

// スロー中エフェクト（Blueprint実装）
void ApplySlowMotionPostProcess_Implementation()
{
    // Blueprintでカスタムエフェクトを実装可能
}
```

設計の特徴:

- 各エフェクトがタグで管理され、重複適用を防止
- PostProcessEffectHandleが一元管理
- Blueprint側で視覚効果をカスタマイズ可能

工夫3: コンポーネントの遅延初期化と安全なアクセス

全てのコンポーネントアクセスで **null** チェック:

```
cpp
void APlayerCharacter::OnMove(const FInputActionValue& Value)
{
    // StateManager の存在確認
    if (StateManager && StateManager->GetCurrentState())
    {
        StateManager->GetCurrentState()->Movement(Value);
    }
}

void APlayerCharacter::OnLook(const FInputActionValue& Value)
{
    // CameraControl の存在確認
    if (CameraControl)
    {
        CameraControl->ProcessLookInput(Value);
    }
}

bool APlayerCharacter::PlayParkour()
{
    // ParkourComponent の存在確認
    return ParkourComponent && ParkourComponent->Parkour();
}
```

この実装の効果:
- コンポーネントが未設定でもクラッシュしない

- 開発中の段階的な実装が可能
- エディタでの動作確認が安全

工夫4: 状態とコンポーネントの明確な責務分離

責務の分担:
```
PlayerCharacter の責務:
  - コンポーネント間の連携
  - デリゲートの管理
  - 入力の振り分け

StateManager の責務:
  - 状態の管理と遷移
  - 状態に応じた挙動制御

各コンポーネントの責務:
  - 特定機能の実装
  - 自身の状態管理
  - 外部への通知（デリゲート）
```

この設計により:

- PlayerCharacter のコードがシンプルに保たれる
- 各機能が独立してテスト可能
- 新機能の追加が容易

パフォーマンス最適化

1. Tick の効率的な使用

```cpp
void APlayerCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    // StateManager のみ毎フレーム更新
    if (StateManager)
    {
        StateManager->GetCurrentState()->OnUpdate(DeltaTime);
    }
}
```

最適化のポイント:

- PlayerCharacter 自体は最小限の処理のみ
- 各コンポーネントは必要な時のみ動作
- 状態の更新のみを毎フレーム実行

---

2. コンポーネントのキャッシュ

cpp
```
APlayerCharacter::APlayerCharacter()
{
    //コンストラクタで一度だけ生成
    StateManager =
CreateDefaultSubobject<UPlayerStateManager>(TEXT("StateManager"));
    InputBinder = CreateDefaultSubobject<UPlayerInputBinder>(TEXT("InputBinder"));
    // ... 他のコンポーネント
}
```

効果:

- 実行時の動的生成を回避
- メモリアロケーションが最小限
- アクセスが高速（ポインタ参照のみ）

エラーハンドリングとロバスト性

デリゲートバインドの検証

cpp
```
void APlayerCharacter::BeginPlay()
{
    if (ParkourComponent && WallRunComponent)
    {
        ParkourComponent->OnParkourStarted.AddDynamic(this,
&APlayerCharacter::OnParkourStarted);
        ParkourComponent->OnParkourEnded.AddDynamic(this,
&APlayerCharacter::OnParkourEnded);

        UE_LOG(LogTemp, Log, TEXT("Parkour delegates bound successfully"));
    }
    else
    {
        //コンポーネントが見つからない場合、詳細をログ出力
        if (!ParkourComponent)
```

```
            UE_LOG(LogTemp, Error, TEXT("ParkourComponent not found"));
        if (!WallRunComponent)
            UE_LOG(LogTemp, Error, TEXT("WallRunComponent not found"));
    }
}
```

この実装の意図:

- 初期化時に問題を早期発見
- どのコンポーネントが欠けているか明確化
- 開発中のデバッグを容易に

拡張性の設計

1. Blueprint Native Event による拡張ポイント

```
cpp
// C++で基本実装、Blueprintで拡張可能
UFUNCTION(BlueprintNativeEvent, Category = "Effects")
void ApplySlowMotionPostProcess();

UFUNCTION(BlueprintNativeEvent, Category = "Effects")
void RemoveSlowMotionPostProcess();
```

活用シーン:

- アーティストが視覚効果をカスタマイズ
- プロトタイプ段階での迅速な調整
- プロジェクト固有のエフェクト実装

---

2. インターフェイスによる機能追加

```
cpp
// 新しいインターフェイスを追加するだけで機能拡張
class APlayerCharacter :
    public IPlayerInfoProvider,
    public IPlayerInputReceiver,
    public ITimeControllable,
    public INewFeatureInterface  // 新機能を追加
```

この設計の利点:
- 既存コードへの影響が最小限
- 新機能が明示的に分離される

- 他のクラスでも同じインターフェイスを実装可能

技術的チャレンジと解決策

チャレンジ1: 複数の特殊能力の競合管理

直面した問題: パルクール・壁走り・ブースト等が同時に発動すると挙動が不安定

採用した解決策:
```
1. 状態パターンで排他制御
   - パルクール中は他のアクションを無効化
   - 壁走り中はパルクールを無効化

2. デリゲートで相互通知
   - パルクール開始時に壁走りを強制終了
   - 着地時に適切な状態に遷移

3. フラグによる二重実行防止
   - bIsPerformingParkour でパルクール実行中を管理
   - IsWallRunning() で壁走り中を判定
```

結果: 複雑な能力の組み合わせでも安定動作

チャレンジ2: カメラとメッシュのアタッチメント管理

直面した問題: FPSカメラで頭に追従すると揺れが激しい、固定すると没入感が低い

採用した解決策:
```
1. 動的なアタッチメント切り替え機能を実装
   - 通常プレイ: ルートにアタッチ（安定）
   - カットシーン: 頭ソケットにアタッチ（リアル）

2. メッシュの表示・非表示も連動
   - 頭追従時: メッシュ表示（TPSライク）
   - 固定時: メッシュ非表示（FPSライク）

3. インターフェイス経由で外部制御可能
   - IPlayerInfoProvider::SetCameraAttachToHead()
```

結果: 状況に応じた最適なカメラ体験を実現

チャレンジ3: 時間操作とアニメーションの同期

直面した問題: スロー中もアニメーションは通常速度で再生されてしまう

採用した解決策:
```
1. グローバルタイムディレーション使用
   TimeManagerSubsystem->StartSlowMotion(0.1)
   → ワールド全体の時間を0.1倍速に

2. サウンドもスローに連動
   USoundHandle::PlaySE("SlowTime", true)
   → ループ再生でスロー感を演出

3. ポストプロセスで視覚的フィードバック
   ApplySlowMotionPostProcess()
   → 青い色調で時間操作を明示
```

結果: アニメーション・物理・サウンドが完全に同期

得られた成果

✅ 高度な機能統合

- 7つの独立したコンポーネントをシームレスに統合
- 状態パターンによる挙動の動的切り替え
- デリゲートによる疎結合な連携

✅ 優れた拡張性

- 新機能はコンポーネントとして追加可能
- インターフェイスによる契約ベース設計
- Blueprint拡張ポイントの提供

✅ 高い保守性

- 責務が明確に分離
- null チェックによる安全性確保
- 詳細なログ出力によるデバッグ容易性

✅

続ける

21:53

パフォーマンス

- 必要最小限のTick処理
- コンポーネントのキャッシュ
- 効率的なイベント駆動設計

✅ チーム開発への適合

- 各機能が独立して開発可能
- アーティストがBlueprintで拡張可能
- 明確なインターフェイスによる連携

まとめ

このプレイヤーキャラクターシステムは、コンポーネント指向設計、状態パターン、インターフェイス駆動設計を核として実装しました。

複数の特殊能力を持つ複雑なキャラクターでありながら、各機能が疎結合で実装されているため、保守性と拡張性を両立しています。デリゲートを活用した機能間連携により、PlayerCharacter自体はシンプルな「ハブ」としての役割に徹し、実際の処理は各コンポーネントと状態クラスに委譲されています。

使用した主要技術:

- Component-Based Architecture (コンポーネント指向)
- State Pattern (状態パターン)
- Interface-Driven Design (インターフェイス駆動)
- Delegate Pattern (デリゲートパターン)
- Blueprint Native Event (Blueprint拡張)

実装を通じて、適切なデザインパターンの組み合わせにより、複雑なゲームプレイ要素を整理された形で実装できることを実感しました。各コンポーネントが独立しているため、チーム開発でも並行作業が容易で、機能の追加・削除・変更が柔軟に行えるアーキテクチャとなっています。