Unreal Engine 5で実装した、キャラクターが壁を「登る」「乗り越える」動作を自動判定・実行するパルクールシステムです。プレイヤーが壁に近づいてアクションボタンを押すと、壁の高さや厚さを自動分析し、最適なアクション（Climb/Vault）を選択・実行します。

核心的な設計思想

1. コルーチンベースの非同期処理アーキテクチャ

従来のTimer/Tickベースの実装ではなく、**UE5Coro**ライブラリを活用した協調的マルチタスキングを採用しました。

従来の実装との比較:

【従来のTimer実装】

1. アニメーション再生開始

2. Timer設定(アニメーション時間後に呼ぶ)

3. フラグ管理(bIsAnimating等)

4. コールバック関数実装

→ コードが分散、状態管理が複雑化

【コルーチン実装】

1. co_await PlayMontageAsync(Climb);

2. co_await PlayMontageAsync(GettingUp);

3. co_await CleanupParkour();

→ 処理の流れが一連のコードとして記述可能

この設計を選んだ理由:

- アニメーション再生→待機→次の処理という流れを順次的に記述できる
- コールバック地獄を回避し、可読性が劇的に向上
- フレーム単位のTickが不要になり、**CPU**負荷を削減できる
- 処理の中断/再開が容易で、複雑なシーケンス制御が実装しやすい

---

2. 段階的壁分析アルゴリズム

壁を3段階のラントレースで分析し、適切なアクションを自動決定する仕組みを実装しました。

【分析フェーズ】

Phase 1: DetectWallImpact()

→ 前方にラントレースを実行

→ 壁の接触位置と法線ベクトルを取得

→ "NoParkour"タグチェック(特定オブジェクトの除外)

Phase 2: DetectWallTop()

→ 壁表面から上方向にトレース

→ 壁の上端位置を検出

→ 高さを計算(Height = TopZ - ImpactZ)

Phase 3: DetectWallThickness()

→ 壁を貫通して内側表面を検出

→ 内側の上端位置を取得

→ 厚さ判定(Thickness = TopZ - InnerTopZ)

この情報を組み合わせて**4**つのケースに自動分岐:

壁の種類	高さ	厚さ	実行アクション
低い薄い壁	< 80cm	< 30cm	Vault (飛び越える)
低い厚い壁	< 80cm	≥ 30cm	GettingUp (壁の上に乗る)
高い薄い壁	≥ 80cm	< 30cm	Climb → Vault (登って飛び越える)

高い厚い壁	≥ 80cm	≥ 30cm	Climb → GettingUp (登って壁の上に乗る)

システム全体構造

入力層

└→ UPlayerInputBinder

└→ IPlayerInputReceiver (インターフェイス)

└→ APlayerCharacter

└→ UParkourComponent

実行層

UParkourComponent

├── 壁検出サブシステム

│ ├─ DetectWallImpact() (接触検出)

│ ├─ DetectWallTop() (高さ検出)

│ └─ DetectWallThickness() (厚さ検出)

│

├── アクション判定エンジン

│ ├─ CalculateWallProperties() (プロパティ計算)

│ └─ CanPerformClimb() (実行可能性チェック)

│

├── コルーチン実行系

│ ├─ ParkourSequence() (メインシーケンス)

│ ├─ ExecuteClimbSequence() (登りアクション)

│ ├─ ExecuteVaultSequence() (飛び越えアクション)

```
│    └─ PlayMontageAsync() (アニメーション再生)
│
└── 物理・入力制御系
    ├─ DisablePhysicsAndCollision() (物理無効化)
    ├─ EnablePhysicsAndCollision() (物理有効化)
    ├─ DisableCharacterInput() (入力無効化)
    └─ EnableCharacterInput() (入力有効化)
```

データフロー

初期化フロー

```
1. UParkourComponent::BeginPlay()
   ↓
2. InitializeComponent()
   ├→ CachedCharacter = GetOwner() (ACharacter取得)
   ├→ CachedCapsule = GetCapsuleComponent() (コリジョン取得)
   ├→ CachedMovement = GetCharacterMovement() (移動コンポーネント取得)
   ├→ CachedAnimInstance = GetAnimInstance() (アニメーション取得)
   └→ TraceObjectTypes.Add(WorldStatic) (トレース設定)
```

設計意図: 頻繁にアクセスするコンポーネントを**TWeakObjectPtr**でキャッシュし、毎フレームの検索コストを削減しています。

パルクール実行フロー

```
【入力トリガー】
プレイヤーがパルクールボタン押下
   ↓
APlayerCharacter::OnParkourAction()
```

↓

UParkourComponent::Parkour()

【前処理 - 同期処理】

1. 実行中チェック

if (bIsPerformingParkour) return false;

2. 壁の存在確認

DetectWallImpact(CurrentWallInfo)

→ 壁がない or NoParkourタグ付き → return false

3. 壁の上端検出

DetectWallTop(CurrentWallInfo)

→ 上端が検出できない → return false

【メインシーケンス - 非同期コルーチン】

ParkourSequence() 開始

↓

4. 初期化

├→ bIsPerformingParkour = true

├→ OnParkourStarted.Broadcast() (デリゲート通知)

└→ DisableCharacterInput() (入力無効化)

5. 詳細分析

├→ CalculateWallProperties() (高さ計算)

└→ DetectWallThickness() (厚さ判定)

6. アクション選択

```
if (bHasInnerSurface) {  // 内側表面あり
    if (CanPerformClimb()) {  // 前方に障害物なし
        co_await ExecuteClimbSequence();
    } else {  // 前方に障害物あり
        co_await ExecuteVaultSequence();
    }
} else {  // 内側表面なし(薄い壁)
    co_await ExecuteVaultSequence();
}
```

7. 終了処理

```
co_await CleanupParkour()
  ├→ EnablePhysicsAndCollision() (物理復帰)
  ├→ EnableCharacterInput() (入力復帰)
  ├→ bIsPerformingParkour = false
  └→ OnParkourEnded.Broadcast() (完了通知)
```

各アクションシーケンスの詳細

ExecuteClimbSequence() - 登りアクション

【実行条件】

- 壁に内側表面がある
- 前方に障害物がない
- 高さ ≥ 80cm

【処理フロー】

1. DisablePhysicsAndCollision()

   ├→ Capsule コリジョンを NoCollision に設定

   ├→ MovementComponent の速度をゼロにリセット

   ├→ 移動モードを MOVE_Falling に設定

   └→ 重力を 0.0 に設定

2. キャラクター位置を壁の上に移動

   TargetZ = CurrentWallInfo.TopLocation.Z + 20.0f

   SetActorLocation(TargetLocation)

3. Climb アニメーション再生

   co_await PlayMontageAsync(EParkourMontageType::Climb);

   → アニメーション終了まで自動待機

4. (厚い壁の場合のみ) GettingUp アニメーション再生

   if (bIsThickWall)

      co_await PlayMontageAsync(EParkourMontageType::GettingUp);

実装のポイント:

- 物理演算を完全停止することで、アニメーション中の意図しない落下を防止
- `co_await`により、アニメーション終了を自動待機
- 厚い壁の場合は2段階アニメーション（登る→立ち上がる）を実行

ExecuteVaultSequence() - 飛び越えアクション

【実行条件】

- 壁に内側表面がない(薄い壁) or

- 前方に障害物がある(Climbができない)

【処理フロー】

1. DisablePhysicsAndCollision()

2. ターゲット位置とアニメーションを決定

```
if (bIsThickWall) {
    // 厚い壁: 壁の表面方向に前進
    ForwardVector = 壁の法線ベクトルから計算
    TargetLocation = CurrentLocation + (ForwardVector * 50.0f)
    MontageType = GettingUp
} else {
    // 薄い壁: 壁の上に移動
    TargetLocation.Z = TopLocation.Z + 20.0f
    MontageType = Vault
}
```

3. キャラクター移動

```
SetActorLocation(TargetLocation)
```

4. アニメーション再生

```
co_await PlayMontageAsync(MontageType);
```

設計の工夫:

- 壁の厚さによって移動方向を動的に変更
- 薄い壁は垂直上昇、厚い壁は前方移動
- 法線ベクトルを使用することで、斜めの壁にも対応できるようにした

主要な実装上の工夫

工夫1: TWeakObjectPtr によるメモリ安全性

cpp

```
UPROPERTY()
TWeakObjectPtr<ACharacter> CachedCharacter;


UPROPERTY()
TWeakObjectPtr<UCapsuleComponent> CachedCapsule;
```

この設計を選んだ理由:

- オブジェクトが破棄された際のダングリングポインタを防止
- 使用前に`IsValid()`で安全性を確認できる

得られた効果:

- クラッシュリスクの大幅削減
- デバッグログによる問題の早期発見が可能に

工夫2: 物理演算の完全制御

cpp

```
void UParkourComponent::DisablePhysicsAndCollision()
{
    CachedCapsule->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    CachedMovement->StopMovementImmediately();
    CachedMovement->Velocity = FVector::ZeroVector;
    CachedMovement->SetMovementMode(EMovementMode::MOVE_Falling);
    CachedMovement->GravityScale = 0.0f;
}
```

なぜこの実装が必要だったか:

1. コリジョン無効化: アニメーション中に壁にめり込まないようにする
2. 速度リセット: ジャンプ中の慣性を完全に打ち消す
3. 落下モード設定: 空中アニメーションを自然に再生するため
4. 重力ゼロ: アニメーション中の落下を完全に防止

復帰処理も確実に実装:

cpp

```cpp
void UParkourComponent::EnablePhysicsAndCollision()
{
    CachedCapsule->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    CachedMovement->SetMovementMode(EMovementMode::MOVE_Walking);
    CachedMovement->GravityScale = NormalGravityScale;  // 5.0に復帰
}
```

工夫3: NoParkourタグによる除外システム

cpp

```cpp
if (const AActor* HitActor = HitResult.GetActor())
{
    if (HitActor->ActorHasTag(FName("NoParkour")))
    {
        return false;  // パルクール不可
    }
}
```

実装の意図:

- ガラス窓、脆い壁、装飾オブジェクトなど、登るべきでない物体を除外
- レベルデザイナーがタグ1つで制御可能
- コード変更不要で柔軟なマップ設計が可能

工夫4: デリゲートによる外部連携

cpp

UPROPERTY(BlueprintAssignable, Category = "Parkour|Events")

FOnParkourStateChanged OnParkourStarted;

UPROPERTY(BlueprintAssignable, Category = "Parkour|Events")

FOnParkourStateChanged OnParkourEnded;

```

**想定している活用例:**

- **UIシステム**: パルクール開始時にクールダウンUIを表示
- **サウンドシステム**: 効果音やBGMの再生
- **スタミナシステム**: パルクール実行時にスタミナ消費
- **アチーブメントシステム**: 連続パルクール回数のカウント

**設計意図:** UParkourComponentは他システムの存在を知らず、**疎結合を維持**

---

## 入力システムとの統合

### 全体の入力フロー

```

【入力層】

プレイヤーがEキー押下

  ↓

UPlayerInputBinder::HandleAction()

```
  ↓
IPlayerInputReceiver::OnAction() (インターフェイス)
  ↓
APlayerCharacter::OnAction()
  ↓
StateManager->GetCurrentState()->ParkourAction()

【状態による分岐】
DefaultState::ParkourAction()
  └→ ParkourComponent->Parkour() ← ここで実行

WallRunState::ParkourAction()
  └→ 何もしない (壁走り中はパルクール不可)

LandingState::ParkourAction()
  └→ 何もしない (着地硬直中は不可)
```

この設計の特徴:

- 状態パターンにより、キャラクターの状態に応じてパルクール実行を制御
- UParkourComponentは状態を意識せず、単純に「実行できるか」だけを判定

パフォーマンス最適化

1. Tick を使用しない設計

cpp

```
UParkourComponent::UParkourComponent()
{
    PrimaryComponentTick.bCanEverTick = false;  // Tick完全無効
```

```
}
```

この設計による効果:

- 毎フレームの処理がゼロ
- パルクール実行時のみCPUリソースを使用
- 大量のキャラクターがいる場合でもスケーラブル

---

2. ライントレース設定のキャッシュ

cpp

```
// 初期化時に一度だけ設定
void UParkourComponent::InitializeComponent()
{
    TraceObjectTypes.Add(UEngineTypes::ConvertToObjectType(ECC_WorldStatic));
}

// 実行時は再利用
bool UParkourComponent::PerformLineTrace(...)
{
    return UKismetSystemLibrary::LineTraceSingleForObjects(
        GetWorld(),
        Start,
        End,
        TraceObjectTypes,  // キャッシュされた設定を使用
        ...
    );
}
```

最適化の効果:

- トレース実行ごとの配列生成を回避

- メモリアロケーションの削減

---

3. constexpr による定数の最適化

cpp

```
bool UParkourComponent::PerformLineTrace(...) const
{
    constexpr bool bTraceComplex = false;
    constexpr bool bIgnoreSelf = true;
    constexpr EDrawDebugTrace::Type DrawDebugType = EDrawDebugTrace::None;
    ...
}
```

この実装の効果:

- コンパイル時に定数が展開される
- 実行時のメモリアクセスがゼロ

エラーハンドリングとロバスト性

null チェックの徹底

cpp

```
TCoroutine<> UParkourComponent::ExecuteClimbSequence()
{
    if (!CachedCharacter.IsValid() || CurrentWallInfo.bRequiresClimbing)
    {
        co_return;  // 早期リターンで安全に終了
    }
    ...
}
```

全ての外部参照に対して実装:

- `IsValid()`による存在確認
- 早期リターンによる安全な処理中断
- UE_LOGによるデバッグ情報の出力

状態フラグによる二重実行防止

cpp

```
bool UParkourComponent::Parkour()

{

    if (bIsPerformingParkour)  // 実行中チェック

    {

        return false;  // 二重実行を防止

    }

    ...

}
```

この実装の効果:

- 連打による意図しない挙動を防止
- アニメーションの中断を回避

拡張性の設計

1. TMap によるアニメーション管理

cpp

```
UPROPERTY(EditAnywhere, Category = "Parkour|Animation")

TMap<EParkourMontageType, UAnimMontage*> AnimMontageMap;
```

この設計の利点:

- 新しいアクションタイプを簡単に追加可能
- Blueprintエディタで視覚的に設定できる

- コード変更なしで異なるキャラクターに適用可能

---

2. パラメータの外部公開

cpp

```
UPROPERTY(EditAnywhere, Category = "Parkour|Detection")
float WallDetectionDistance = 70.0f;  // 壁検出距離

UPROPERTY(EditAnywhere, Category = "Parkour|Detection")
float ClimbHeightThreshold = 80.0f;  // 登り判定の高さ閾値

UPROPERTY(EditAnywhere, Category = "Parkour|Detection")
float ThicknessThreshold = 30.0f;  // 厚さ判定の閾値
```

想定している活用シーン:

- レベルデザイナーがゲーム体験を調整
- キャラクターごとに異なる能力を設定
- プロトタイプ段階での迅速な調整

技術的チャレンジと解決策

チャレンジ1: 壁の厚さをどう判定するか

直面した問題: 壁の表面しか見えない状態で、内側があるかを判定する必要があった

採用した解決策:

```
1. 壁表面の法線ベクトルを取得
2. 法線の逆方向に50cm進んだ位置から上方向にトレース
3. ヒットすれば内側表面あり、しなければ薄い壁
```

結果: 様々な形状の壁に対応可能になった

チャレンジ2: アニメーション中の物理制御

直面した問題: アニメーション再生中に重力で落下してしまう

採用した解決策:

```
1. コリジョンを完全無効化
2. 重力をゼロに設定
3. 速度をリセット
4. 落下モードに設定(空中アニメーション用)
```

結果: 滑らかで自然なパルクールアニメーションを実現

チャレンジ3: コルーチンの学習コスト

直面した課題: UE5Coroは比較的新しいライブラリで、チーム全体での理解が必要だった

採用した対策:

```

1. 各コルーチン関数に詳細なコメントを記述

2. 処理の流れを視覚的に記述

3. co_awaitの使用箇所を最小限に保つ
```

結果: 新メンバーでも理解しやすいコードに

得られた成果

✅ 保守性の向上

- コルーチンによる処理フローの明確化
- 各メソッドの責務が単一かつ明確
- コメントとログによる追跡が容易

✅ パフォーマンスの改善

- Tick不使用による常時CPU負荷ゼロ
- キャッシュ戦略による検索コスト削減
- 必要な時のみ実行される効率的な設計

✅ 高い拡張性

- 新しいパルクールタイプの追加が容易
- パラメータ調整による多様なゲーム体験
- デリゲートによる他システムとの柔軟な連携

✅ ロバスト性の確保

- TWeakObjectPtrによるメモリ安全性

- 徹底したnullチェック
- 状態フラグによる二重実行防止

✅ チーム開発への適合

- レベルデザイナーがタグで制御可能
- アニメーターがBlueprintで設定可能
- プログラマー以外でも調整・拡張が可能

まとめ

このパルクールシステムは、コルーチンベースの非同期処理、段階的な壁分析アルゴリズム、徹底した物理制御を核として設計しました。

従来のTimer/Tickベースの実装と比較して、コードの可読性が大幅に向上し、パフォーマンスも改善されています。また、デリゲートやタグシステムを活用することで、他のシステムとの疎結合を維持しながら、柔軟な機能拡張を可能にしました。

使用した主要技術:

- UE5Coro (非同期処理)
- Enhanced Input System (入力管理)
- State Pattern (状態管理)
- Delegate Pattern (イベント通知)
- TWeakObjectPtr (メモリ安全性)

実装を通じて、UE5Coroの強力さと、適切な設計パターンの組み合わせによって、保守性と拡張性を両立できることを実感しました。