コンポーネント構成

- UPhysicsCalculatorComponent (ActorComponent)
- └── コルーチンベースの物理ループ
- ├── 非同期接地判定
- ├── 重力適用

└── 力の適用

主要な役割

UPhysicsCalculatorComponent

- 役割: カスタム物理演算エンジン（非同期最適化版）
- 主な機能:
    - 独自の重力シミュレーション
    - 力（Force）の適用と減衰
    - 接地判定（Ground Check）
    - 着地検出
    - 非同期トレース処理によるパフォーマンス最適化

データの流れ

初期化 → 物理ループ起動

1. `BeginPlay()` → `InitializeComponent()` → Ownerをキャッシュ
2. `StartPhysics()` → `PhysicsUpdateLoop()` コルーチン開始

毎フレームの物理計算（**PhysicsUpdateLoop**）

- while ループ（停止されるまで継続）
- ├── co_await NextTick() - 次のフレームまで待機
- ├── AsyncCheckGroundState() - 非同期接地判定
- │ └── 結果をbIsOnGroundに反映
- ├── ApplyGravity() - 重力適用（空中時のみ）
- │ └── 落下速度を加速 → 最大速度で制限
- └── ApplyForce() - 力を適用（ForceScaleが0より大きい時）

└── 力を減衰させながら移動

技術的特徴

1. **UE5Coro**による非同期処理

cpp

- TCoroutine<> PhysicsUpdateLoop()

TCoroutine<bool> AsyncCheckGroundState()

- 利点: Tick依存を排除、処理の細かい制御が可能
- 仕組み: `co_await NextTick()` で毎フレーム実行、重い処理もメインスレッドをブロックしない

2. キャッシュによる最適化

cpp

TWeakObjectPtr<AActor> CachedOwner;

- 毎フレーム `GetOwner()` を呼ばずキャッシュを使用
- パフォーマンス向上

3. 接地判定の仕組み

cpp

- AsyncCheckGroundState()
- ├── アクターの足元位置を計算
- ├── BoxSweepで下方向にトレース

└── 結果を返す(bool)

- **GroundCheckBoxExtent**: 判定ボックスのサイズ
- **GroundCheckDistance**: 判定する下方向の距離

4. 重力システム

cpp

- ApplyGravity(DeltaTime)
- ├── GravityTimer += DeltaTime(経過時間を蓄積)
- ├── FallSpeed = (GravityScale * GravityTimer) / GravityDivider
- ├── 最大速度で制限

└── AddActorLocalOffset(0, 0, -FallSpeed)

- 時間とともに加速する自然な落下
- `MaxFallingSpeed` で終端速度を設定可能

5. 力の適用と減衰

cpp

- ApplyForce(DeltaTime)
- ├── ForceScale -= DeltaTime * 10.0f（減衰）
- ├── MoveVector = ForceDirection * ForceScale

└── AddActorWorldOffset / AddActorLocalOffset

- 一度加えた力は自動的に減衰
- ローカル/ワールド座標の選択可能
- Sweep有効化でコリジョン対応

主要なパラメータ

パラメータ	デフォルト値	説明
GravityScale	9.8f	重力の強さ
MaxFallingSpeed	200.0f	落下の最大速度
GravityDivider	1.0f	重力の微調整係数
GroundCheckBoxExtent	(40, 20, 15)	接地判定のボックスサイズ
GroundCheckDistance	5.0f	接地判定の深さ

状態管理

cpp

- // 接地状態
- bIsOnGround　　// 現在接地しているか
- bWasOnGround　　// 前フレームで接地していたか
- bHasJustLanded　// 今フレームで着地したか（!bWasOnGround && bIsOnGround）
-
- // 物理状態

- bIsPhysicsActive　　// 物理ループが動いているか

bShouldStopPhysics　// 停止フラグ

使用例

cpp

- // 初期化（自動起動）
- // BeginPlayで自動的にStartPhysics()が呼ばれる
-
- // ジャンプ
- PhysicsCalc->AddForce(FVector::UpVector, 100.0f);
-
- // 横方向の力
- PhysicsCalc->AddForce(FVector::ForwardVector, 50.0f);
-
- // 重力設定変更
- PhysicsCalc->SetGravityScale(true, 19.6f, 1.0f); // 重力2倍
-
- // 接地確認
- if (PhysicsCalc->IsOnGround())
- {
- // 地面にいる時の処理
- }
-
- // 着地検出
- if (PhysicsCalc->HasJustLanded())
- {
- // 着地演出（カメラシェイクなど）
- CameraControl->PlayCameraShake(LandingShakeClass);
- }
-
- // 物理停止

PhysicsCalc->StopPhysics();

他システムとの連携

PlayerCameraControlComponent との連携

cpp

```
// 着地時にカメラシェイク
if (PhysicsCalc->HasJustLanded())
{
    CameraControl->PlayCameraShake(LandingCameraShake);
}
```

PostProcessEffectManager との連携

cpp

```
// 落下中はエフェクト
if (!PhysicsCalc->IsOnGround())
{
    EffectManager->ActivateEffect(EPostProcessEffectTag::Falling);
}
```

設計のポイント

なぜカスタム物理を使うのか？

- より細かい制御が欲しい（例: 壁走り、特殊なジャンプ挙動）
- 軽量な物理演算が必要

非同期処理の利点

- Tickに依存しないため、フレームレートの影響を受けにくい
- 重い接地判定を非同期化してパフォーマンス向上
- コルーチンで読みやすいコード
-