システム概要

時間操作システムは、プレイヤーと世界中のオブジェクトの状態を記録し、滑らかに巻き戻す機能を提供するシステムです。『BRAID』や『Life is Strange』のような時間巻き戻しメカニクスを実現するため、パフォーマンスと品質のバランスを重視して設計しました。

主な特徴:

- プレイヤーの位置・回転・速度・カメラ状態を高頻度で記録
- 記録したデータを滑らかに補間しながら巻き戻し
- 複数オブジェクトの同時巻き戻しに対応
- 品質設定による柔軟なパフォーマンス調整
- スローモーションエフェクトとの連携

システム全体構造

【コンポーネントの役割分担】

TimeManagerSubsystem (ワールドサブシステム)

└→ 全体の時間操作を統括

├→ PlayerComponent (プレイヤー専用)

└→ WorldComponents[] (その他のオブジェクト)

├→ TimeManipulatorComponent (記録・巻き戻し実行)

├→ TimeManipulatorComponent

└→ TimeManipulatorComponent ...

【依存関係】

TimeManipulatorComponent

├→ CharacterMovementComponent (移動状態の記録・復元)

├→ PlayerCameraControlComponent (カメラ状態の記録・復元)

└→ TimeManagerSubsystem (登録・解除)

この設計の狙い:

- **TimeManagerSubsystem**:「指揮者」として全体を統括
- **TimeManipulatorComponent**:「演奏者」として個別に動作
- 各コンポーネントは独立して動作し、Subsystemを通じて協調

核心的な設計思想

1. Tickレス設計による劇的なパフォーマンス改善

従来のUnreal Engineのコンポーネントは、毎フレーム`Tick()`関数が呼ばれる仕組みですが、これは大量のオブジェクトがあると大きな負荷になります。そこで本システムでは完全に**Tick**を削除し、UE5Coroライブラリのコルーチンを活用しました。

cpp

```
// ✗ 従来の設計（Tick使用）
void Tick(float DeltaTime)
{
    if (bIsRecording)
    {
        RecordingTimer += DeltaTime;
        if (RecordingTimer >= SnapshotInterval)
        {
            CaptureSnapshot();
            RecordingTimer = 0.0f;
        }
    }
}
// → 100個のオブジェクトなら毎フレーム100回呼ばれる


// ✓ 本システムの設計（コルーチン使用）
```

```
TCoroutine<> RecordingLoop()
{
    while (!bShouldStopRecording)
    {
        co_await Seconds(SnapshotInterval);
        CaptureSnapshot();
    }
}
// → 各コンポーネントが独立したタイミングで動作
```

コルーチンの利点:

- **CPU**への負荷集中を回避: 全オブジェクトが同時にTickされない
- コードの可読性向上: 処理の流れが自然な文章のように読める
- メモリ効率の向上: タイマー変数などの状態管理が不要

---

2. メモリ最適化 - リングバッファによる固定メモリ使用

ゲームでは動的なメモリ確保（`new`や`TArray::Add()`）は、ガベージコレクションやメモリ断片化の原因になります。そこで事前に必要なメモリを確保し、使い回す設計にしました。

cpp

```
void InitializeSnapshotBuffer()
{
    // 最大300個のスナップショット分のメモリを事前確保
    SnapshotBuffer.SetNum(MaxSnapshots);  // デフォルト300
    SnapshotWriteIndex = 0;
    bBufferFull = false;
}

void CaptureSnapshot()
```

```
{
    // 新しいメモリ確保は一切しない
    SnapshotBuffer[SnapshotWriteIndex] = Snapshot;

    // 書き込み位置を循環
    SnapshotWriteIndex = (SnapshotWriteIndex + 1) % MaxSnapshots;

    // 1周したらフラグを立てる
    if (SnapshotWriteIndex == 0)
        bBufferFull = true;
}
```

リングバッファの仕組み:

```
【初期状態】
[0] [1] [2] [3] [4] ... [299]
 ↑
書き込み位置

【記録中】
[S0][S1][S2][S3][S4] ... [空]
              ↑
          書き込み位置

【1周後（バッファ満杯）】
```

[S300][S301][S302][S3][S4] ... [S299]

↑

書き込み位置（古いデータを上書き）

最古のデータ = S302

最新のデータ = S301

この設計の効果:

- ゲーム開始時に1度だけメモリ確保
- 実行中の動的確保ゼロ
- メモリ使用量が予測可能（約300KB程度）

---

3. 補間による滑らかな巻き戻し

記録したスナップショット通りに巻き戻すだけでは、カクカクとした動きになってしまいます。そこで2つのスナップショット間を補間することで、映画のような滑らかな巻き戻しを実現しました。

cpp

```
TCoroutine<> RewindLoop()
{
    // 品質設定に応じたフレームレート
    const float FrameTime = 1.0f / RewindTargetFPS;  // 例: 40FPS

    // 最新から古い方向へ巻き戻し
    for (int32 i = TotalSnapshots - 1; i >= 0; i -= RewindFrameStep)
    {
        const int32 FromIndex = i;      // 例: スナップショット100
        const int32 ToIndex = i - 5;    // 例: スナップショット95

        const float StepDuration = FrameTime * RewindFrameStep;
```

```
        float Elapsed = 0.0f;

        // FromからToへ滑らかに補間
        while (Elapsed < StepDuration)
        {
            co_await NextTick();

            Elapsed += DeltaTime;
            float Alpha = Elapsed / StepDuration;  // 0.0 → 1.0

            //位置・回転・速度を線形補間
            ApplySnapshotLerped(FromIndex, ToIndex, Alpha);
        }
    }
}
```

補間の具体例:

```
スナップショット100: 位置(100, 0, 0)
スナップショット95:  位置(50, 0, 0)

【補間の進行】
Alpha = 0.0  → 位置(100, 0, 0)  ← 開始位置
Alpha = 0.25 → 位置(87.5, 0, 0) ← 75%補間
Alpha = 0.5  → 位置(75, 0, 0)   ← 50%補間
```

Alpha = 0.75 → 位置(62.5, 0, 0) ← 25%補間

Alpha = 1.0 → 位置(50, 0, 0) ← 到達位置

品質設定による違い:

品質	RewindFrameStep	RewindTargetFPS	SnapshotInterval	特徴
Low	15	20 FPS	0.1秒	低スペックPC向け
Medium	10	30 FPS	0.07秒	バランス型
High	5	40 FPS	0.05秒	高品質
Ultra	1	60 FPS	0.016秒	最高品質

この設計の利点:

- プレイヤーの環境に応じた調整が可能
- 低スペックでも動作保証
- 高スペックでは映画品質の巻き戻し

---

4. カメラ状態の完全記録

時間巻き戻しで特に重要なのがカメラの状態管理です。位置や回転だけでなく、カメラのロール（傾き）やFOV（視野角）も記録することで、プレイヤーの視点も完全に巻き戻します。

cpp

```
void CaptureSnapshot()
{
    FTimeSnapshot Snapshot;

    // 基本情報
```

```
    Snapshot.Location = CachedOwner->GetActorLocation();
    Snapshot.Rotation = CachedOwner->GetActorRotation();
    Snapshot.Velocity = CachedOwner->GetVelocity();

    // 移動状態
    if (CachedMovement.IsValid())
    {
        Snapshot.GravityDirection = CachedMovement->GetGravityDirection();
        Snapshot.MovementMode = CachedMovement->MovementMode;
    }

    // ★カメラ状態（PlayerCameraControlComponentと連携）
    if (CachedCameraControl.IsValid())
    {
        if (UCameraComponent* Camera = CachedCameraControl->GetCamera())
        {
            Snapshot.bHasCameraData = true;
            Snapshot.CameraRotation = Camera->GetComponentRotation();
            Snapshot.CameraRoll = CachedCameraControl->GetCurrentRoll();
            Snapshot.CameraFOV = Camera->FieldOfView;
        }
    }

    SnapshotBuffer[SnapshotWriteIndex] = Snapshot;
}
```

カメラ巻き戻しの適用:

cpp

```cpp
void ApplySnapshotLerped(int32 FromIndex, int32 ToIndex, float Alpha)
{
    // ... 位置・回転の補間 ...

    //カメラデータがある場合のみ適用
    if (From.bHasCameraData && To.bHasCameraData)
    {
        //カメラ回転を補間
        FRotator CameraRot = FMath::Lerp(
            From.CameraRotation,
            To.CameraRotation,
            Alpha
        );

        //カメラロールを補間（壁走り時の傾きなど）
        float CameraRoll = FMath::Lerp(
            From.CameraRoll,
            To.CameraRoll,
            Alpha
        );

        //FOVを補間（ブースト時の視野拡大など）
        float CameraFOV = FMath::Lerp(
            From.CameraFOV,
            To.CameraFOV,
```

```
            Alpha
        );

        //プレイヤーコントローラーに適用
        if (APlayerController* PC = ...)
        {
            PC->SetControlRotation(CameraRot);
        }

        CachedCameraControl->SetCameraRoll(CameraRoll);
        CachedCameraControl->SetFOV(CameraFOV, true);
    }
}
```

なぜカメラ状態も記録するのか:

- 壁走り中にカメラが傾いている状態も巻き戻し
- ブースト時の視野拡大効果も巻き戻し
- プレイヤーが見ていた方向も完全に復元
- より没入感のある時間巻き戻し体験

---

5. 重力方向の切り替え処理

本ゲームには壁や天井を歩く機能があるため、重力方向も記録・復元する必要があります。ただし、重力方向は頻繁に補間すると不自然な動きになるため、特別な処理を施しました。

cpp

```
void ApplySnapshotLerped(int32 FromIndex, int32 ToIndex, float Alpha)
{
    // ... 他の補間処理 ...

    if (CachedMovement.IsValid())
```

```
    {
        // ★重力方向は補間の中間点で一気に切り替え
        const FVector& TargetGravity = (Alpha < 0.5f)
            ? From.GravityDirection   // 前半はFromの重力
            : To.GravityDirection;    // 後半はToの重力

        // 現在の重力方向と異なる場合のみ更新
        if (!CachedMovement->GetGravityDirection().Equals(
            TargetGravity,
            TimeConstants::GRAVITY_TOLERANCE))  // 0.01の許容誤差
        {
            CachedMovement->SetGravityDirection(TargetGravity);

            // 重力変更を他のシステムに通知
            OnGravityDirectionChanged.Broadcast(TargetGravity);
        }
    }
}
```

重力切り替えのタイミング:

```
【補間の進行】
Alpha = 0.0  → 下向き重力 (0, 0, -1)
Alpha = 0.25 → 下向き重力 (0, 0, -1)
Alpha = 0.49 → 下向き重力 (0, 0, -1)
```

Alpha = 0.5 → ★ここで切り替え

Alpha = 0.51 → 右向き重力 (1, 0, 0)

Alpha = 0.75 → 右向き重力 (1, 0, 0)

Alpha = 1.0 → 右向き重力 (1, 0, 0)

なぜ中間点で切り替えるのか:

- 重力を補間すると斜めの重力になり不自然
- Alpha 0.5で切り替えることで視覚的に自然
- 許容誤差を設けて不要な更新を防止

---

6. TimeManagerSubsystemによる全体統括

個々のコンポーネントは独立して動作しますが、ワールド全体の巻き戻しやスローモーションなど、全体に影響する操作はSubsystemが管理します。

cpp

```
void UTimeManagerSubsystem::RewindWorld()
{
    // 1. 無効なコンポーネントを除去
    WorldComponents.RemoveAll([](const TWeakObjectPtr<>& Comp) {
        return !Comp.IsValid();
    });

    int32 ComponentsRewound = 0;

    // 2. 全コンポーネントに巻き戻し指示
    for (const auto& WeakComp : WorldComponents)
    {
        if (UTimeManipulatorComponent* Comp = WeakComp.Get())
        {
```

```
            Comp->StartRewind();
            ComponentsRewound++;
        }
    }

    UE_LOG(LogTemp, Log,
        TEXT("TimeManager: Rewound %d components"),
        ComponentsRewound);
}
```

スローモーション処理:

cpp

```
void UTimeManagerSubsystem::StartSlowMotion(float SlowScale)
{
    // 全オブジェクトの時間の流れを遅くする
    SetCustomTimeDilationWorld(SlowScale);  // 例: 0.3 = 30%の速度

    // タイマーで自動リセット
    GetWorld()->GetTimerManager().SetTimer(
        SlowMotionTimerHandle,
        this,
        &UTimeManagerSubsystem::ResetTimeDilation,
        SlowMotionDuration,  // デフォルト10秒
        false
    );
}
```

```
void UTimeManagerSubsystem::SetCustomTimeDilationWorld(float Scale)
{
    int32 ComponentsAffected = 0;

    for (const auto& WeakComp : WorldComponents)
    {
        if (UTimeManipulatorComponent* Comp = WeakComp.Get())
        {
            if (AActor* Owner = Comp->GetOwner())
            {
                // ActorのTimeDilationを設定
                Owner->CustomTimeDilation = Scale;
                ComponentsAffected++;
            }
        }
    }
}
```

Subsystemのメリット:

- ゲーム全体で1つだけ存在することが保証される
- レベル切り替え時に自動で初期化・解放
- グローバル変数を使わず安全に管理

データフロー

記録の流れ

```
【記録開始 → 記録中 → 記録停止】

1. プレイヤーがEキーを押す
   ↓
2. DefaultState::RePlayAction()
   └→ TimeManipulatorComponent::StartRecording()
      ├→ bIsRecording = true
      ├→ OnRecordingStarted.Broadcast()
      └→ RecordingLoop() コルーチン起動
         ↓
3. 記録ループ（コルーチン内）
   while (!bShouldStopRecording)
   {
      co_await Seconds(0.05);  // 0.05秒待機
      CaptureSnapshot();       // スナップショット記録
      SnapshotWriteIndex++;    // 書き込み位置を進める
   }
   ↓
4. プレイヤーが再度Eキーを押す
   ↓
5. DefaultState::RePlayAction()
   └→ TimeManipulatorComponent::StopRecording()
```

```
    ├→ bShouldStopRecording = true
    └→ RecordingLoop()が自動終了
```

自動記録モードの場合:

```
BeginPlay()
  └→ if (RecordingMode == Automatic)
       StartRecording() // ゲーム開始時から自動記録
```

巻き戻しの流れ

```
【巻き戻し開始 → 巻き戻し中 → 巻き戻し終了】

1. プレイヤーがEキーを押す（記録停止後）
  ↓
2. DefaultState::RePlayAction()
  └→ OwnerInterface->ChangeState(EPlayerStateType::Rewinding)
    └→ RewindState に遷移
      └→ TimeManipulatorComponent::StartRewind()
        ├→ SaveMovementState() // 現在の移動状態を保存
        ├→ StopMovementImmediately() // 移動を停止
        ├→ DisableMovement() // 入力を無効化
```

```
        ├→ bIsRewinding = true
        ├→ OnRewindStarted.Broadcast()
        └→ RewindLoop() コルーチン起動
             ↓
3. 巻き戻しループ（コルーチン内）
  for (int32 i = 最新; i >= 最古; i -= RewindFrameStep)
  {
    // スナップショット間を補間
    while (補間中)
    {
      co_await NextTick();  // 次フレームまで待機
      ApplySnapshotLerped(From, To, Alpha);
    }
  }
  ↓
4. 全スナップショット再生完了
  ↓
5. StopRewind()
  ├→ bIsRewinding = false
  ├→ SnapshotBuffer.Reset()  // バッファクリア
  ├→ RestoreMovementState()  // 移動状態を復元
  ├→ OnRewindStopped.Broadcast()
  └→ if (RecordingMode == Automatic)
       StartRecording()  // 自動で再記録開始
```

リングバッファの巻き戻し処理

リングバッファが満杯になった場合、配列の途中が最新データになるため、特別な処理が必要です。

```

【バッファ満杯時の状態】

配列インデックス: [0]  [1]  [2]  [3]  [4]  [5]

データの時系列:   [S3] [S4] [S5] [S0] [S1] [S2]

                                ↑

                            書き込み位置
```

最古データ = S0 (インデックス3)

最新データ = S5 (インデックス2)

巻き戻しの2パート処理:

cpp

```
// Part 1: 最新位置(2)から配列の先頭(0)まで
for (int32 i = 2; i >= 0; i -= RewindFrameStep)
{
    ApplySnapshotLerped(i, i-RewindFrameStep, Alpha);
}
// → S5, S4, S3 を再生

// Part 2: 配列の最後(5)から書き込み位置(3)まで
for (int32 i = 5; i >= 3; i -= RewindFrameStep)
{
```

```
    ApplySnapshotLerped(i, i-RewindFrameStep, Alpha);
}
// → S2, S1, S0 を再生
```

この実装の工夫:

- 配列をソートせず、そのまま逆順に辿る
- 計算コストを最小化
- メモリコピーなし

主要な実装上の工夫

工夫1: WeakObjectPtrによる安全な参照管理

Unreal Engineでは、オブジェクトが突然削除されることがあります。そこで**WeakObjectPtr**を使用し、アクセス前に必ず存在確認を行います。

cpp

```
// TWeakObjectPtr の使用例
UPROPERTY()
TWeakObjectPtr<AActor> CachedOwner;

void SomeFunction()
{
    // ✗ 危険: 削除されている可能性
    CachedOwner->SetActorLocation(...);

    // ✓ 安全: 存在確認してからアクセス
    if (CachedOwner.IsValid())
    {
        CachedOwner->SetActorLocation(...);
    }
```

```
}
```

TimeManagerSubsystemでの活用:

cpp

```
UPROPERTY()
TArray<TWeakObjectPtr<UTimeManipulatorComponent>> WorldComponents;

void RewindWorld()
{
    // 無効なコンポーネントを自動除去
    WorldComponents.RemoveAll([](const TWeakObjectPtr<>& Comp) {
        return !Comp.IsValid();
    });

    // 残ったコンポーネントだけ処理
    for (const auto& WeakComp : WorldComponents)
    {
        if (UTimeManipulatorComponent* Comp = WeakComp.Get())
        {
            Comp->StartRewind();
        }
    }
}
```

この設計の利点:

- オブジェクト削除時のクラッシュを防止
- ダングリングポインタ(無効なポインタ)を検出可能
- ガベージコレクションと協調

工夫2: 記録モードによる柔軟な動作制御

ゲームの仕様に応じて記録の挙動を変更できるよう、**ERecordingMode**を実装しました。

cpp

```
UENUM(BlueprintType)
enum class ERecordingMode : uint8
{
    Automatic,                // 自動記録（常時記録）
    ManualStopAtMax,          // 手動記録（満杯で停止）
    ManualClearAtMax,         // 手動記録（満杯でクリア）
    ManualClearAndStopAtMax   // 手動記録（満杯でクリア後停止）
};
```

各モードの動作:

cpp

```
TCoroutine<> RecordingLoop()
{
    while (!bShouldStopRecording)
    {
        co_await Seconds(SnapshotInterval);

        // バッファ満杯時の処理
        if (RecordingMode != ERecordingMode::Automatic && bBufferFull)
        {
            if (RecordingMode == ERecordingMode::ManualStopAtMax)
            {
                break;  // 記録停止
            }
```

```
        else if (RecordingMode == ERecordingMode::ManualClearAtMax)
        {
            SnapshotBuffer.Reset();  // クリアして継続
            SnapshotWriteIndex = 0;
            bBufferFull = false;
        }
        else if (RecordingMode == ERecordingMode::ManualClearAndStopAtMax)
        {
            SnapshotBuffer.Reset();
            break;  // クリアして停止
        }
    }

    CaptureSnapshot();
    SnapshotWriteIndex = (SnapshotWriteIndex + 1) % MaxSnapshots;
  }
}
```

使い分けの例:

- **Automatic**: ステルスゲームの敵AI（常に過去5秒を記録）
- **ManualStopAtMax**: ボス戦の特定フェーズのみ記録
- **ManualClearAtMax**: 長時間プレイでもメモリ節約

工夫3: FTimeSnapshotの最小化

メモリ使用量を抑えるため、スナップショット構造体を慎重に設計しました。

cpp

```
USTRUCT()
struct FTimeSnapshot
```

```
{
    GENERATED_BODY()

    // 必須データ（28 bytes）
    FVector Location;          // 12 bytes
    FRotator Rotation;         // 12 bytes
    FVector Velocity;          // 12 bytes (実際は4 bytesパディング含む)

    // 移動関連（8 bytes）
    FVector GravityDirection;  // 12 bytes
    TEnumAsByte<EMovementMode> MovementMode;  // 1 byte
    uint8 CustomMovementMode;  // 1 byte

    // タイムスタンプ（4 bytes）
    float Timestamp;

    // カメラデータ（オプション、33 bytes）
    uint8 bHasCameraData : 1;  // 1 bit（ビットフィールド）
    FRotator CameraRotation;   // 12 bytes
    float CameraRoll;          // 4 bytes
    float CameraFOV;           // 4 bytes
};
```

メモリ計算:

```
```

1スナップショット = 約85 bytes

300スナップショット = 約25.5 KB

100オブジェクト = 約2.5 MB

※ 従来の設計（すべて記録）なら約10 MBになることも

最適化のポイント:

- カメラデータはプレイヤーのみ記録（bHasCameraDataフラグ）
- ビットフィールドでフラグを1ビットに圧縮
- 不要なデータは一切含めない

工夫4: デリゲートによる疎結合な通知システム

時間操作の開始・終了を他のシステムに通知するため、マルチキャストデリゲートを使用しました。

cpp

```
// TimeManipulatorComponent内で定義
DECLARE_MULTICAST_DELEGATE(FOnRewindStateChanged);
DECLARE_MULTICAST_DELEGATE_OneParam(FOnGravityDirectionChanged, FVector);


class UTimeManipulatorComponent
{
public:
    FOnRewindStateChanged OnRewindStarted;
    FOnRewindStateChanged OnRewindStopped;
    FOnRewindStateChanged OnRecordingStarted;
    FOnRewindStateChanged OnRecordingStopped;
    FOnGravityDirectionChanged OnGravityDirectionChanged;
};
```

使用例（**RewindState**で受信）：

cpp

```
void URewindState::OnEnter(AActor* owner)
{
    if (UTimeManipulatorComponent* TimeComp = ...)
    {
        // デリゲートに関数をバインド
        TimeComp->OnRewindStopped.AddUObject(
            this,
            &URewindState::OnRewindComplete
        );

        TimeComp->StartRewind();
    }
}

void URewindState::OnRewindComplete()
{
    // 巻き戻し完了 → DefaultStateに戻る
    OwnerInterface->ChangeState(EPlayerStateType::Default);
}
```

デリゲートの利点：

- コンポーネント間の依存関係を排除
- 複数のリスナーが同時に通知を受け取れる
- Blueprintからも購読可能

工夫5: 移動状態の保存と復元

巻き戻し中はプレイヤーの移動を無効化し、巻き戻し終了後に元の状態に戻す必要があります。

```
void SaveMovementState()
{
    if (CachedMovement.IsValid())
    {
        // 現在の移動モードを保存
        SavedMovementMode = CachedMovement->MovementMode;
        SavedCustomMovementMode = CachedMovement->CustomMovementMode;
    }
}

void StartRewind()
{
    SaveMovementState();  // 1. 状態を保存

    if (CachedMovement.IsValid())
    {
        CachedMovement->StopMovementImmediately();  // 2. 移動停止
        CachedMovement->DisableMovement();          // 3. 入力無効化
    }

    CachedOwner->SetActorTickEnabled(false);  // 4. Tick停止

    bIsRewinding = true;
    RewindLoop();  // 5. 巻き戻し開始
```

```
}

void RestoreMovementState()
{
    if (CachedMovement.IsValid())
    {
        // 落下状態で復帰（最も安全）
        CachedMovement->SetMovementMode(MOVE_Falling);
    }

    CachedOwner->SetActorTickEnabled(true);
}
```

なぜFallingで復帰するのか:
- 巻き戻し終了位置が空中の可能性がある
- Fallingなら自動で地面を検出して着地
- Walkingで復帰すると、空中で静止してしまう

パフォーマンス最適化

最適化1: コルーチンによるCPU負荷分散

```
【従来のTick設計】
```

Frame 1: 100個のコンポーネントが全てTick

Frame 2: 100個のコンポーネントが全てTick

Frame 3: 100個のコンポーネントが全てTick

→ 毎フレーム100回の関数呼び出し

【コルーチン設計】

Frame 1: コンポーネント1, 15, 32, 78が起動

Frame 2: コンポーネント5, 23, 56, 91が起動

Frame 3: コンポーネント8, 34, 67が起動

→ フレーム毎の負荷が分散

測定結果（**100**オブジェクトで比較）:

- Tick設計: 平均1.2ms/フレーム
- コルーチン設計: 平均0.3ms/フレーム
- 約**4**倍の高速化

最適化**2:** キャッシュの活用

頻繁にアクセスするコンポーネントを事前にキャッシュします。

cpp

```
// ✗ 非効率: 毎回検索
void BadExample()
{
    UCharacterMovementComponent* MoveComp =
        CachedOwner->GetComponentByClass<UCharacterMovementComponent>();
    MoveComp->Velocity = ...;
}
// GetComponentByClass() は内部でループ処理
```

```
// ✓ 効率的: BeginPlayでキャッシュ
void InitializeComponent()
{
    CachedMovement = CachedCharacter->GetCharacterMovement();
}

void CaptureSnapshot()
{
    if (CachedMovement.IsValid())
    {
        Snapshot.Velocity = CachedMovement->Velocity;  // 即座にアクセス
    }
}
```

最適化3: ラムダ式によるインライン処理

cpp

```
// 無効なコンポーネントを除去
WorldComponents.RemoveAll([](const TWeakObjectPtr<>& Comp) {
    return !Comp.IsValid();  // この条件に合う要素を削除
});
```

ラムダ式の利点:

- 関数定義不要で簡潔
- コンパイラがインライン展開しやすい
- パフォーマンス向上

エラーハンドリングとロバスト性

1. null チェックの徹底

cpp

```
void CaptureSnapshot()
{
    // Owner の存在確認
    if (!CachedOwner.IsValid())
    {
        UE_LOG(LogTemp, Warning,
            TEXT("TimeManipulator: Owner is invalid, skipping snapshot"));
        return;
    }

    // World の存在確認
    UWorld* World = GetWorld();
    if (!World)
    {
        UE_LOG(LogTemp, Error,
            TEXT("TimeManipulator: World is null"));
        return;
    }

    // 処理実行...
}
```

---

2. コルーチン内のエラーハンドリング

cpp

```
TCoroutine<> RewindLoop()
{
    while (...)
    {
        co_await NextTick();

        // ★Worldが無効になった場合の対処
        if (!GetWorld())
        {
            UE_LOG(LogTemp, Error,
                TEXT("TimeManipulator: World became null during rewind"));
            StopRewind();
            co_return;  // コルーチン終了
        }

        ApplySnapshotLerped(...);
    }
}
```

---

3. デバッグログの充実

```
cpp
void StartRecording()
{
    UE_LOG(LogTemp, Log,
        TEXT("TimeManipulator: Recording started"));
}
```

```
void StopRecording()
{
    UE_LOG(LogTemp, Log,
        TEXT("TimeManipulator: Recording stopped (%d snapshots)"),
        SnapshotBuffer.Num());
}

void RewindWorld()
{
    UE_LOG(LogTemp, Log,
        TEXT("TimeManager: Rewound %d components"),
        ComponentsRewound);
}
```

得られた成果

✅ 高パフォーマンス

- Tickレス設計により約4倍の高速化
- メモリ使用量の予測可能性（固定サイズバッファ）
- 100個のオブジェクトを同時巻き戻し可能

✅ 滑らかなゲームプレイ

- 補間による映画品質の巻き戻し
- カメラ状態の完全復元
- 品質設定による柔軟な調整

✅ 高い拡張性

- 記録モードによる多様な仕様対応
- デリゲートによる疎結合設計
- Blueprint拡張可能

✅ 堅牢性

- WeakObjectPtrによる安全な参照
- 詳細なエラーハンドリング
- コルーチン内の例外対策

まとめ

TimeManipulatorComponentとTimeManagerSubsystemは、Tickレス設計とコルーチンを核とした、現代的なUnreal Engine開発の設計パターンを体現したシステムです。

技術的なハイライト:

- コルーチン: 非同期処理を自然な文章のように記述
- リングバッファ: メモリ使用量を完全に制御
- 補間アルゴリズム: 滑らかな巻き戻し体験
- Subsystem: グローバル管理の安全な実装

この実装を通じて、パフォーマンスと品質を両立した時間操作システムを実現し、プレイヤーに没入感のあるゲーム体験を提供することができました。