

# プログラム説明

## 作品名 「ActionPlayerClass」

横浜デジタルアーツ専門学校 ゲーム科  
2年 新井 桜大

# ActionPlayerClass



ジャンル: アクションゲーム

動作環境: PC、Windows

開発環境: C++ / DxLib

使用ツール: Visual Studio

開発期間: 2025年11月06日～1月13日 約40時間

紹介動画:

GitURL :

## 作品概要:

2年次に DxLib と C++ を用いて開発した 2D アクションゲームです。

本作品では、どんな 2D アクションゲームにも組み込める 汎用的なプレイヤークラス を設計・実装した点が特徴です。

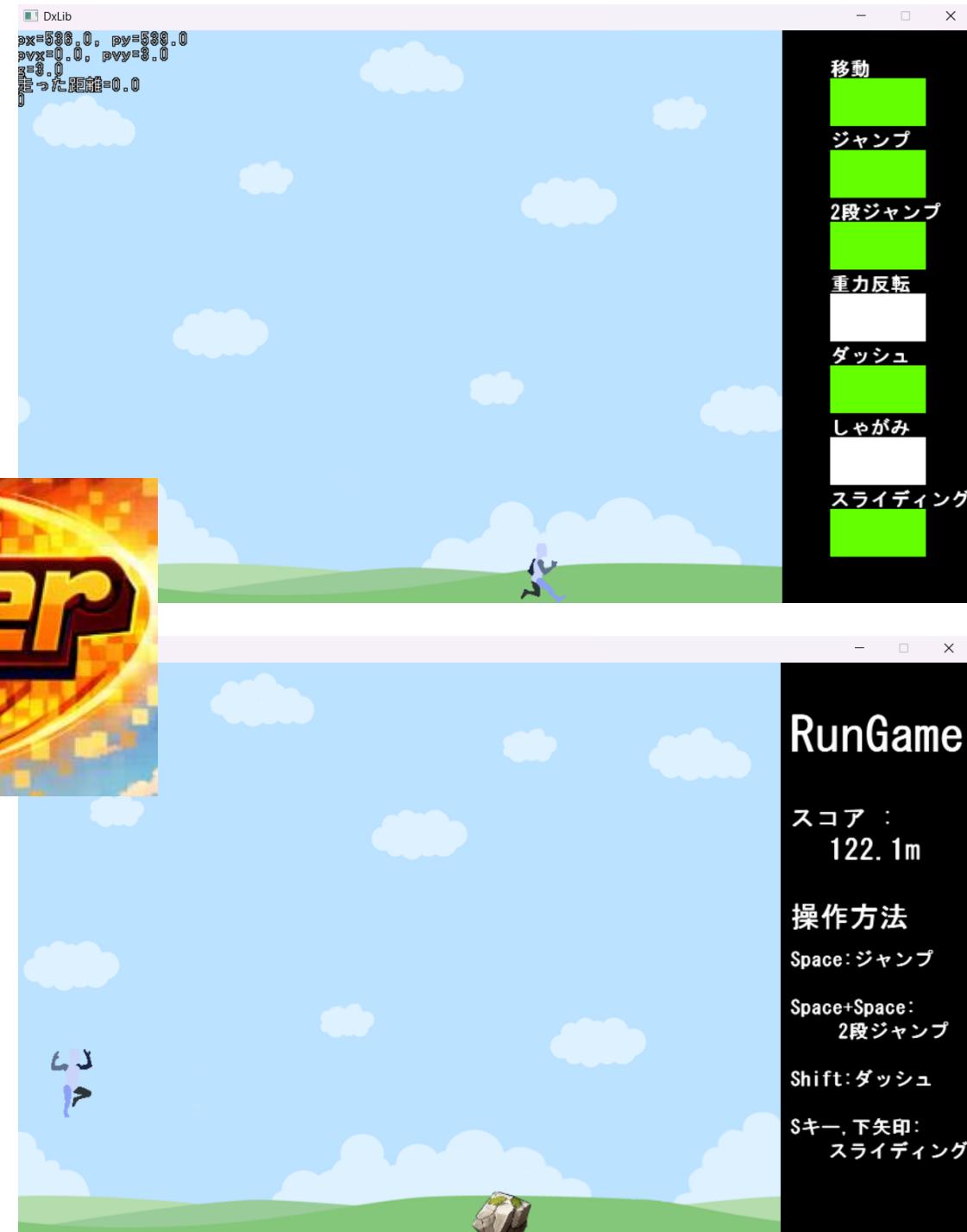
移動、ジャンプ、2 段ジャンプ、スライディングなどの各アクションをモジュール化し、ゲームロジックから独立した再利用性の高い構成を実現しました。さらに、これらのアクションは、設定で個別に有効・無効を切り替えられるようにし、必要な機能だけを選択して組み込める柔軟な設計としています。

# 画面イメージ



**ENDLESS RUN**

**SIMULATION MODE**



# ActionPlayerClass

## アピールポイント・こだわった点

- ゲームマネージャー設計
- プレイヤー操作
  - フラグ機能のON／OFF管理
  - 状態駆動アニメーション
  - アクションのモジュール化
- 障害物マネージャー
  - 障害物管理の一元化
  - メモリ安全性と外部依存の抑制
  - 更新処理の整理
- UI設計
  - 描画処理の最適化
  - ゲームモード別UI制御
  - 拡張性を意識したUI構造

# ゲームマネージャー設計

## ActionPlayerClass

- グローバルな状態管理 : ゲーム全体を見渡す中枢として機能
- インスタンス生成の統制 : 意図しない多重生成や参照ミスを防ぐ
- 拡張前提のアーキテクチャ : 多様なシーン構成にも柔軟に対応

```
class GameManager {  
private:  
    GameManager(const GameManager&) = delete; //コピー禁止  
    GameManager& operator=(const GameManager&) = delete; //代入禁止  
public:  
    //シングルトンインスタンスを取得  
    static GameManager& GetInstance() {  
        static GameManager instance;  
        return instance;  
    }  
};
```

```
//シーン遷移処理  
void GameManager::SceneTransition(int cx, int cy, TitleUI& t)  
{  
    switch (sceneState)  
    {  
        case TITLE_SCENE:  
            UpdateTitle(cx, cy, t);  
            break;  
        case GAME_SCENE:  
            UpdateGame();  
            break;  
        case RESULT_SCENE:  
            UpdateResult();  
            break;  
    }  
}
```

- ・「状態の一元管理によるバグ発生の抑制」  
シングルトン化により、ゲーム状態を一箇所で管理し、不整合や重複処理を防止。
- ・「複数シーンを跨ぐ共通データの安定運用」  
タイトル～ゲーム～リザルトまで、シーンを跨いでも一貫したデータを保持。

関数	処理
SceneTransition()	ゲーム全体のシーン遷移を集約し、フローを一元管理
UpdateTitle()	タイトル遷移処理
UpdateGame()	ゲーム遷移処理
UpdateResult()	リザルト遷移処理

- ・拡張しやすいシーンアーキテクチャを構築  
新しいシーンも switch に追加するだけで容易に拡張できる柔軟な構造。
- ・シーンごとのロジックを独立化し保守性を向上  
各シーンの仕様変更が他へ影響しない高い保守性を実現。

**機能ON/OFF管理** : フラグでプレイヤー仕様を柔軟に切り替え

**アクションのモジュール化** : 各動作を関数単位で独立管理

**状態駆動アニメーション** : 挙動に応じて自動的に描画を切り替え

### ・操作ON/OFFを切り替えられる設計

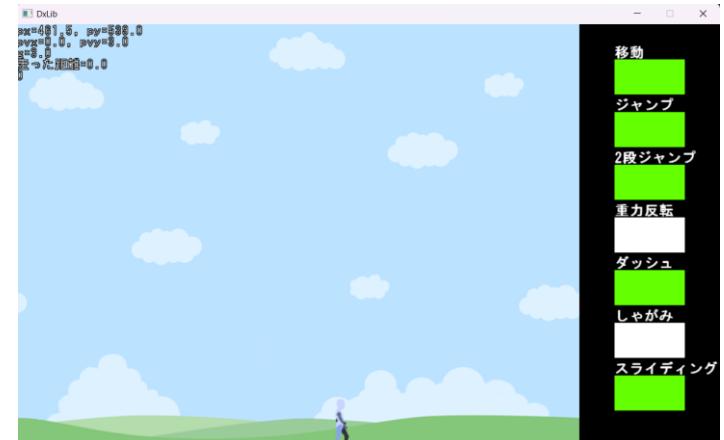
```
//Player操作 (自分で移動操作、ジャンプ、2段ジャンプ、重力反転、ダッシュ、しゃがみ、スライディング)  
Player player(op.GetMoveOperation(), op.GetJump(), op.GetJumpTwo(), op.GetGravity(), op.GetDash(), op.GetCrouch(), op.GetSlide());
```

### ・操作ON/OFFの初期値を与えるクラス

```
class Operation {  
private:  
    //有り: true / 無し: false  
    bool moveOperation; //横移動操作  
    bool jump;          //ジャンプ  
    bool jumpTwo;       //2段ジャンプ  
    bool gravity;       //重力反転  
    bool dash;          //ダッシュ  
    bool crouch;        //しゃがみ  
    bool slide;         //スライディング  
  
public:  
    Operation()  
        :moveOperation(false)  
        , jump(true)  
        , slide(true)  
    {}  
    //ゲット関数  
    bool GetMoveOperation() { return moveOperation; }  
    bool GetJump() { return jump; }  
    bool GetSlide() { return slide; }  
};
```

プレイヤーの各アクション（移動・ジャンプ・ダッシュ・重力反転など）を個別のフラグで管理することで、**ゲーム進行やギミックに応じて操作制限を動的に変更できる設計**にしています。これにより、チュートリアル、難易度調整、別ゲームなどを**コード改修なしで実現可能**です。

### ・シミュレーションモード



**機能ON/OFF管理** : フラグでプレイヤー仕様を柔軟に切り替え

**アクションのモジュール化** : 各動作を関数単位で独立管理

**状態駆動アニメーション** : 挙動に応じて自動的に描画を切り替え

### ・アクションのモジュール化、状態駆動アニメーション

#### 移動操作なし

```
//ジャンプ中且空中にいる時
if (jump && air) {
    DrawJump();
}
//移動操作なしの時は、下矢印かSキーでスライディング
else if (sliding) {
    DrawSlide();
}
//Shiftキーを押している間ダッシュ
else if (dash) {
    DrawDash();
}
//ジャンプや、スライディングじゃなければ全て移動
else {
    DrawRun();
}
```

#### 移動操作あり

```
//ジャンプ中且空中にいる時
if (jump && air) {
    DrawJump();
}
//しゃがんでいてダッシュした時スライディング
else if (sliding) {
    DrawSlide();
}
//しゃがみ状態
else if (crouch) {
    DrawCrouch();
}
//移動状態
else if (move) {
    if (opDash) {
        if (dash) {
            DrawDash();
        }
        else {
            DrawRun();
        }
    }
    else {
        DrawRun();
    }
}
//静止状態
else {
    DrawIdle();
}
```

## ポイント

**入力状態 × キャラ状態 × 優先順位**を整理して描画。  
ジャンプ・スライディング・しゃがみの競合を回避。

# プレイヤー操作

ActionPlayerClass

**機能ON/OFF管理** : フラグでプレイヤー仕様を柔軟に切り替え

**アクションのモジュール化** : 各動作を関数単位で独立管理

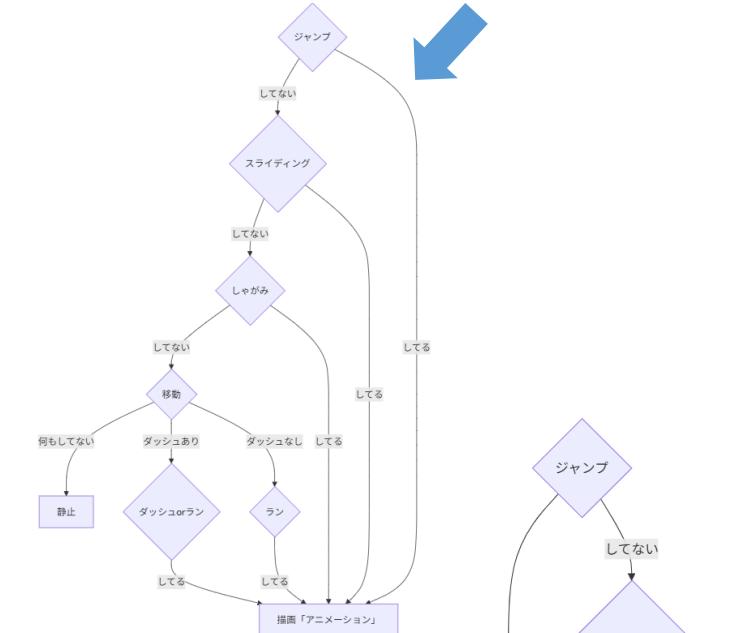
**状態駆動アニメーション** : 挙動に応じて自動的に描画を切り替え

## ・アクションのモジュール化、状態駆動アニメーション

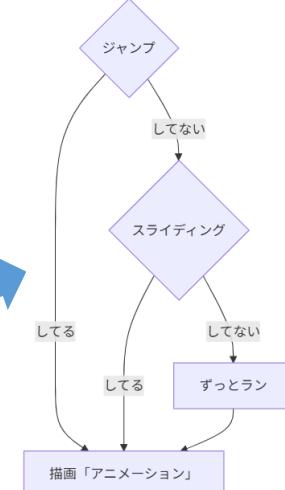
関数	処理
Draw()	プレイヤー全体の描画処理を集約し、一元管理
DrawIdle()	プレイヤー静止状態
DrawRun()	プレイヤーラン状態
DrawDash()	プレイヤーダッシュ状態
DrawJump()	プレイヤージャンプ状態
DrawCrouch()	プレイヤーシャがみ状態
DrawSlide()	プレイヤースライディング状態

描画処理では「入力状態」と「キャラクター状態」を分離し、ジャンプ・スライディング・ダッシュなどの**競合しやすい状態に優先順位を持たせて制御**しています。これにより、意図しないアニメーションの切り替わりを防ぎ、安定した見た目を維持しています。

移動操作ありのアニメーション描画フロー図



移動操作なしの  
アニメーション描画  
フロー図



障害物管理の一元化	: マネージャークラスによる責務分離
メモリ安全性	: unique_ptrによる自動解放設計
更新処理の整理	: 生成・移動・削除の明確化
外部依存の抑制	: ForEachによる安全なアクセス

### ・障害物管理の一元化

```
class ObstaclesManager {  
private:  
    //障害物をまとめて管理する  
    std::vector<std::unique_ptr<Obstacles>> obstacles;  
    //スpawn時間計測用  
    int spawnTimer = 0;  
    //スpawn時間  
    const int spawnTime = 120;  
public:  
    const std::vector<std::unique_ptr<Obstacles>>& GetObstacles() const  
    {  
        return obstacles;  
    }  
  
    //障害物全体に対して処理を行う関数  
    //func : 障害物1つを引数に取る処理  
    void ForEach(const std::function<void(Obstacles&)>& func);  
    void Update();           //障害物更新処理  
    void Draw();             //描画処理  
private:  
    void Spawn();            //障害物生成処理  
    void Move();              //障害物移動処理  
    void RemoveOutOfScreen(); //障害物画面外削除処理  
};
```

障害物の**生成・更新・描画・削除処理**を ObstaclesManager に集約し、ゲーム本体側からは「**障害物全体**」として扱える設計としています。これにより、個々の障害物クラスが管理処理を持たず、**役割を明確に分離した構成**を実現しています。

障害物管理の一元化	: マネージャークラスによる責務分離
メモリ安全性	: unique_ptrによる自動解放設計
外部依存の抑制	: ForEachによる安全なアクセス
更新処理の整理	: 生成・移動・削除の明確化

### ・メモリ安全性

```
constexpr float OUT_OF_SCREEN_X = -200.0f; //削除条件用
```

```
//障害物画面外削除処理
void ObstaclesManager::RemoveOutOfScreen() {
    obstacles.erase(
        std::remove_if(obstacles.begin(), obstacles.end(),
                      [] (const std::unique_ptr<Obstacles>& o) {
                          //画面左に完全に出たら削除
                          return o->GetX() < OUT_OF_SCREEN_X;
                      }),
        obstacles.end()
    );
}
```

障害物は `std::unique_ptr` を用いて管理し、生成から削除までのライフサイクルを `ObstaclesManager` が **一元管理**しています。画面外削除時には `erase-remove` イディオムを用い、**削除と同時に自動でメモリ解放**される安全な構成としました。

### ・外部依存の抑制

```
//障害物全体に処理を適用する
//外部から vector を直接触らせないための関数
void ObstaclesManager::ForEach(const std::function<void(Obstacles&)>& func) {
    for (auto& o : obstacles) {
        func(*o); //実体参照にして渡す
    }
}
```

障害物リストを外部に**直接公開せず**、**ForEach** 関数を通じて障害物全体に処理を適用できる構成とします。これにより、障害物管理の**責務を一元化**し、将来的な実装変更や拡張にも**対応しやすい設計**としました。プレイヤーの当たり判定やデバッグ表示など、障害物全体に対する処理を安全かつ柔軟に行うための**インターフェースとして ForEach** を用意しています。

- 障害物管理の一元化 : マネージャークラスによる責務分離
- メモリ安全性 : unique\_ptrによる自動解放設計
- 外部依存の抑制 : ForEachによる安全なアクセス
- 更新処理の整理 : 生成・移動・削除の明確化

### ・更新処理の整理

```
//障害物更新処理
void ObstaclesManager::Update() {
    Spawn(); //生成処理
    Move(); //移動処理
    RemoveOutOfScreen(); //画面外削除処理
}

//描画処理
void ObstaclesManager::Draw() {
    for (auto& o : obstacles) {
        o->Draw();
    }
}

//障害物生成処理
void ObstaclesManager::Spawn() {
    if (spawnTimer <= 0) {
        SpawnObstacle(ObstacleType::ROCK);
        spawnTimer = spawnTime;
    } else {
        spawnTimer--;
    }
}
```

### Update (更新) 処理フロー図



処理順を固定することで挙動を把握しやすくし、障害物ロジックのデバッグ性と可読性を高めています。

### ・指定障害物生成関数

```
//列挙対
enum class ObstacleType {
    ROCK,
    // LONG_ROCK,
};

//指定障害物生成処理
void ObstaclesManager::SpawnObstacle(ObstacleType type) {
    switch (type) {
        case ObstacleType::ROCK: //岩生成
            obstacles.push_back(std::make_unique<Rock>());
            break;
    }
}
```

- 描画処理の最適化
  - ユーザー目線のUI改善
  - ゲームモード別UI制御
  - 拡張性を意識したUI構造
- : フォント管理と文字描画の抽象化  
: 操作理解と試行錯誤を妨げないUI設計  
: 状況に応じた情報表示  
: 操作項目追加に強い設計

### ・描画処理の最適化

```
//指定文字描画処理
void UI::DrawChar(int r, int g, int b, float x,
float y, int size, const TCHAR* str) {
    //色指定
    color = GetColor(r, g, b);
    //フォントはサイズごとに再利用、新しいサイズなら作られる
    font = GetFont(size);
    //文字描画
    DrawStringFToHandle(x, y, str, color, font);
}
```

本作では、文字描画処理を **UI クラス**に集約し、**色・座標・サイズ・文字列**を引数として渡す共通関数 **DrawChar** を用意しています。これにより、タイトル画面・ゲーム画面・操作説明画面で **同一の描画インターフェース** を利用できる設計としました。

### 宣言

```
std::map<int, int> fontTable; //フォント作成格納テーブル
```

### コード

```
//フォント指定用処理
int UI::GetFont(int size) {
    //count(size) はサイズが存在すれば 1 なければ 0 を返す
    //すでにそのサイズのフォントがあるかどうか
    if (fontTable.count(size) == 0) {
        //まだ作っていない場合は新規作成(作ったフォントをmapに保存)
        fontTable[size] =
            CreateFontToHandle(NULL, size, 6, DX_FONTTYPE_ANTIALIASING);
    }

    return fontTable[size]; //既に作られていたらそのままor今作ったものを返す
}
```

フォントは **サイズごとに map** で管理し、**未作成時のみ生成**、作成されていれば**再利用**することで、無駄なリソース生成を防ぎ、安定した描画処理を実現しています。

**描画処理の最適化**

: フォント管理と文字描画の抽象化

**ユーザー目線のUI改善**

: 操作理解と試行錯誤を妨げないUI設計

**ゲームモード別UI制御**

: 状況に応じた情報表示

**拡張性を意識したUI構造**

: 操作項目追加に強い設計

### ・ゲームモード別UI制御、ユーザー目線のUI改善

ランモード シュミレーションモード



**GameMode (RUN / SIMULATION) を基準**にUI表示を切り替え、  
ランゲーム中はスコアや操作一覧を表示し、シミュレーションモードでは  
設定用UIを表示しています。これにより、プレイヤーの**目的に応じた情報提示**を行っています。

### ・デバッグ表示

```
//デバッグ表示処理
void Player::DrawDebug() {
    printfDx(L"px=% .1f, py=% .1f\n", px, py);
    printfDx(L"px=% .1f, py=% .1f\n", ppx, ppy);
    printfDx(L"g=% .1f\n", g);
}
```

px=268.2, py=589.0  
px=0.0, py=3.0  
g=3.0  
走った距離=0.0  
0

文字サイズや配置を統一することで、  
プレイヤーが直感的に操作内容を理解できるUIを意識

シミュレーションモードでは、**プレイヤー情報を**  
見ることができるデバッグを表示

**描画処理の最適化**

: フォント管理と文字描画の抽象化

**ユーザー目線のUI改善**

: 操作理解と試行錯誤を妨げないUI設計

**ゲームモード別UI制御**

: 状況に応じた情報表示

**拡張性を意識したUI構造**

: 操作項目追加に強い設計

## ・拡張性を意識したUI構造

### 構造体

```
//構造体 ボタン座標用
struct ButtonCoordinate{
    float sx; float sy;
    float ex; float ey;
};
```

### 宣言

```
ButtonCoordinate posButton[OP_COUNT]; //プレイヤー操作設定用ボタン座標
```

### 初期化

```
//ボタン座標初期化
for (int i = 0; i < OP_COUNT; i++)
{ posButton[i] = { SCREEN_WIDTH - 150.0f, 50.0f + (75 * i), SCREEN_WIDTH - 50.0f, 100.0f + (75 * i) };
```

OP\_COUNT(MAX)分、  
等間隔で並ぶ



プレイヤー操作は `enum PlayerOp` と `OP_COUNT` によって一元管理し、  
UI側では操作数に応じて自動的にループ描画を行う設計とっています。  
操作項目を追加した場合でも、UI描画処理の大きな修正が不要となり、  
将来的な機能拡張に対応しやすい構造を意識しました。