

ゲームプログラミング改

○評価要件

- ☒ 敵の頭上に HP ゲージスプライトを表示する
- ☒ マウスクリックした位置に敵を配置する

○概要

今回は 2D と 3D の座標の相互変換を学習します。

マウスやタッチ操作のゲームだとスクリーン座標をクリックすることでワールド空間に配置されている 3D オブジェクトを選択したりします。

これはスクリーン座標（2D）をワールド座標（3D）に変換することで実現しています。

また、同じようにワールド空間に配置されている 3D オブジェクトの頭の上あたりに UI を表示するために、ワールド座標（3D）をスクリーン座標（2D）に変換する場面もあります。

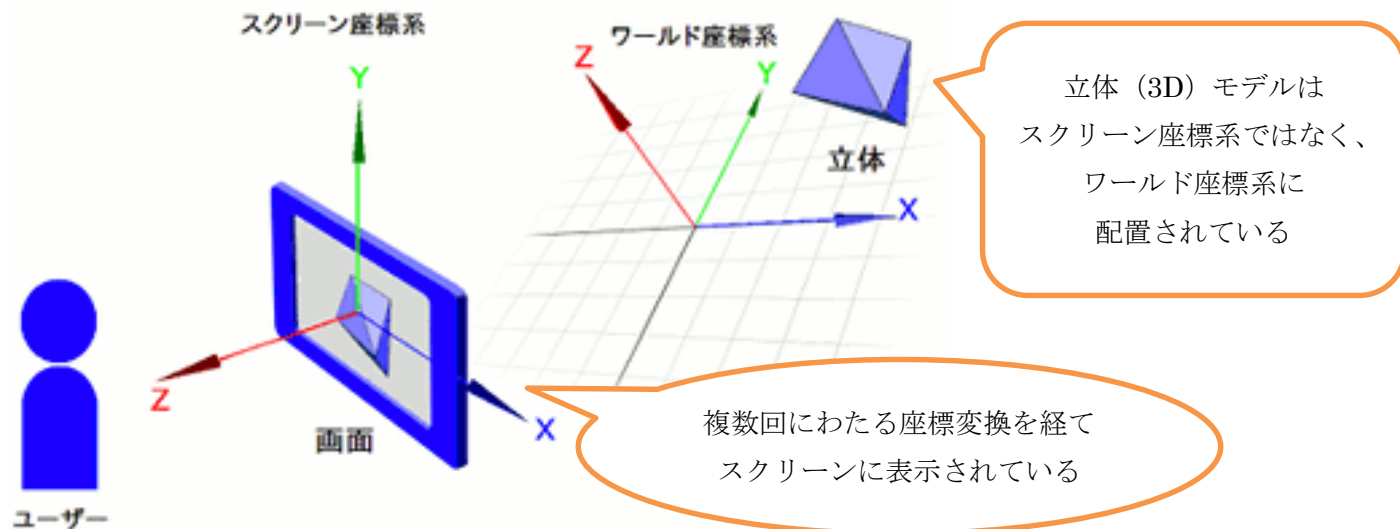
今回は 2D と 3D の相互座標変換がどのように行われているか学習し、座標変換を利用したプログラムを実装しましょう。

d

ゲームプログラミング改

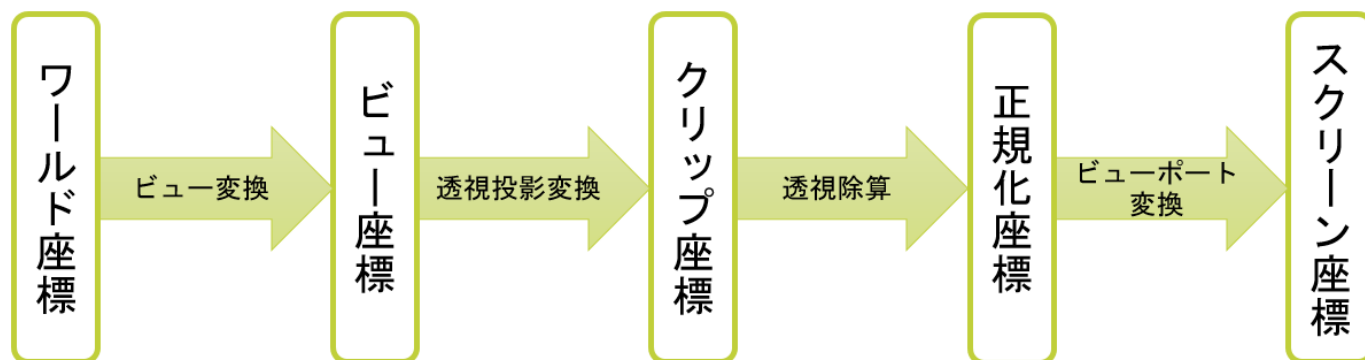
○3DCG の画面表示について

スクリーン座標（2D）とワールド座標（3D）の座標変換を実装するには 3DCG が画面に表示する方法について理解する必要があります。



3D コンピュータグラフィックスの映像は、仮想世界の全ての立体をスクリーン座標系の XY 平面に投射して表示しています。

○ワールド（3D）⇒ スクリーン（2D）座標変換の流れ



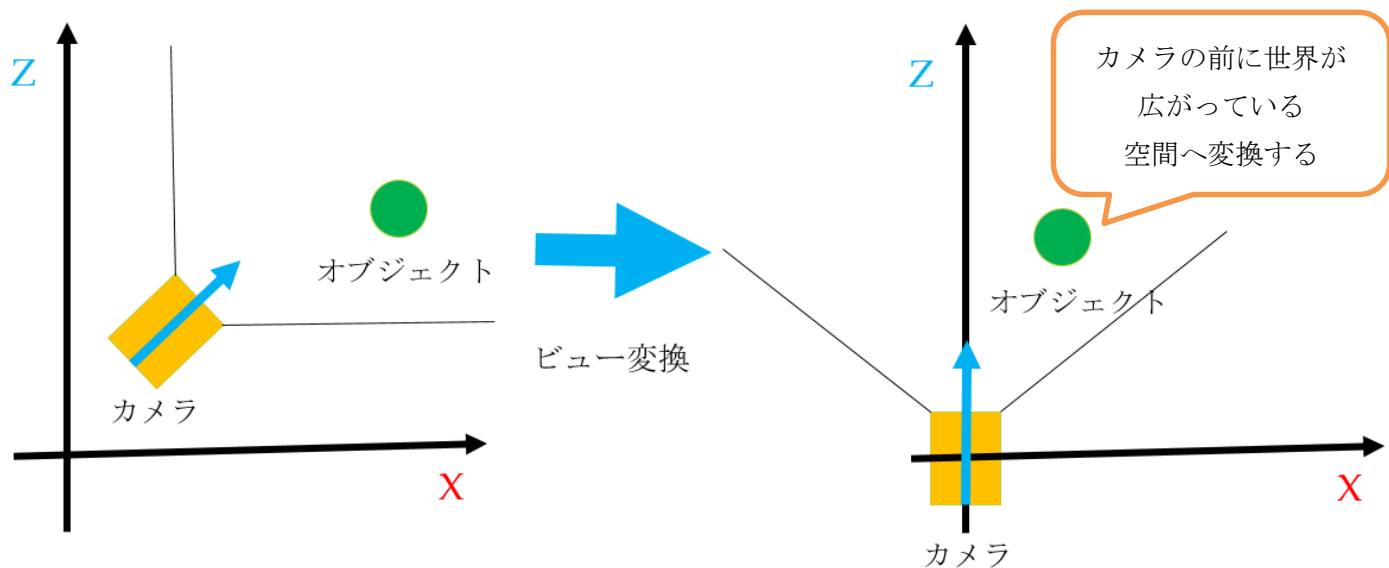
3D から 2D への座標変換は上図の様々な変換することで計算できます。

○ビュー変換

ワールド座標をビュー座標へ変換する計算です。

ワールド座標はワールドを基準として考えた座標系で、ビュー座標とはカメラを基準として考えた座標のことです。

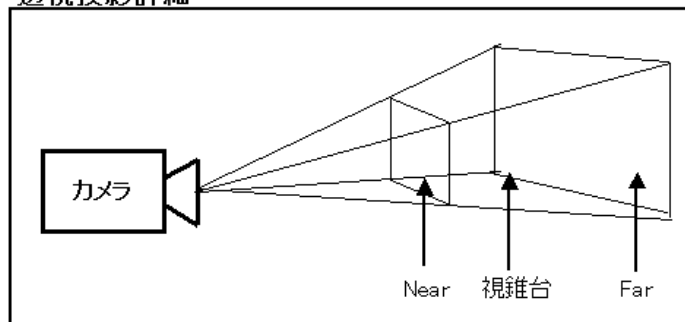
ワールド座標とビュー行列を乗算することで算出できます。



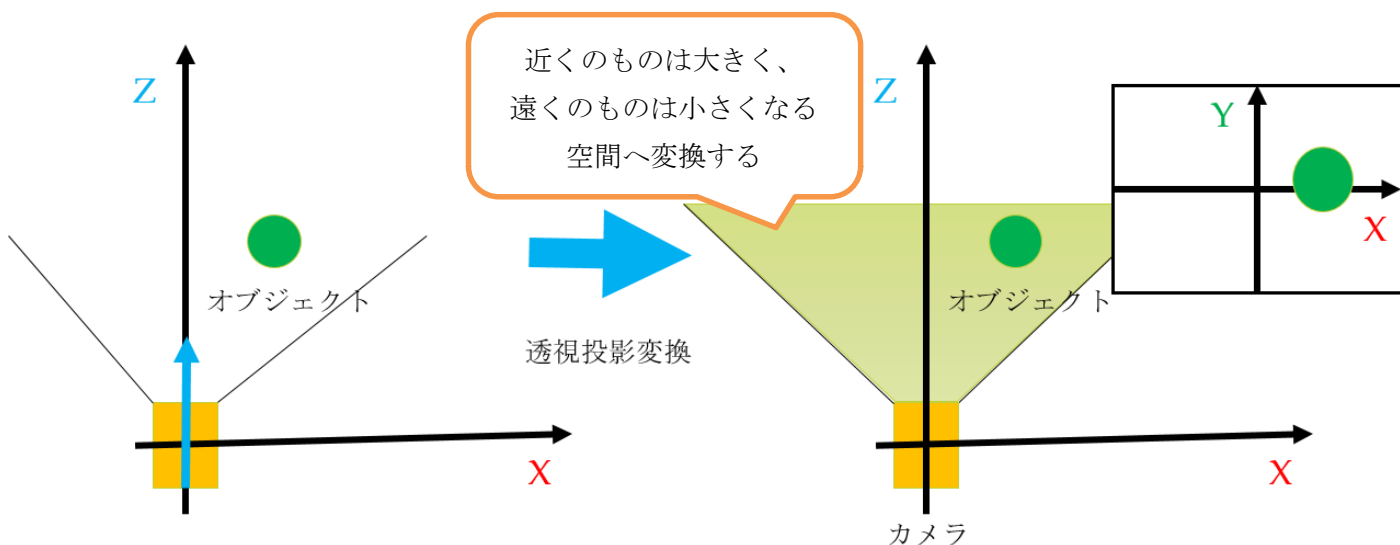
○透視投影変換

3D のカメラは視錐台と呼ぶ 3D 空間の内容を画面に表示する範囲が設定されています。この視錐台の範囲をスクリーン画面におさまるように変換する計算です。この計算を行った座標系をクリッピング座標と呼びます。

透視投影詳細



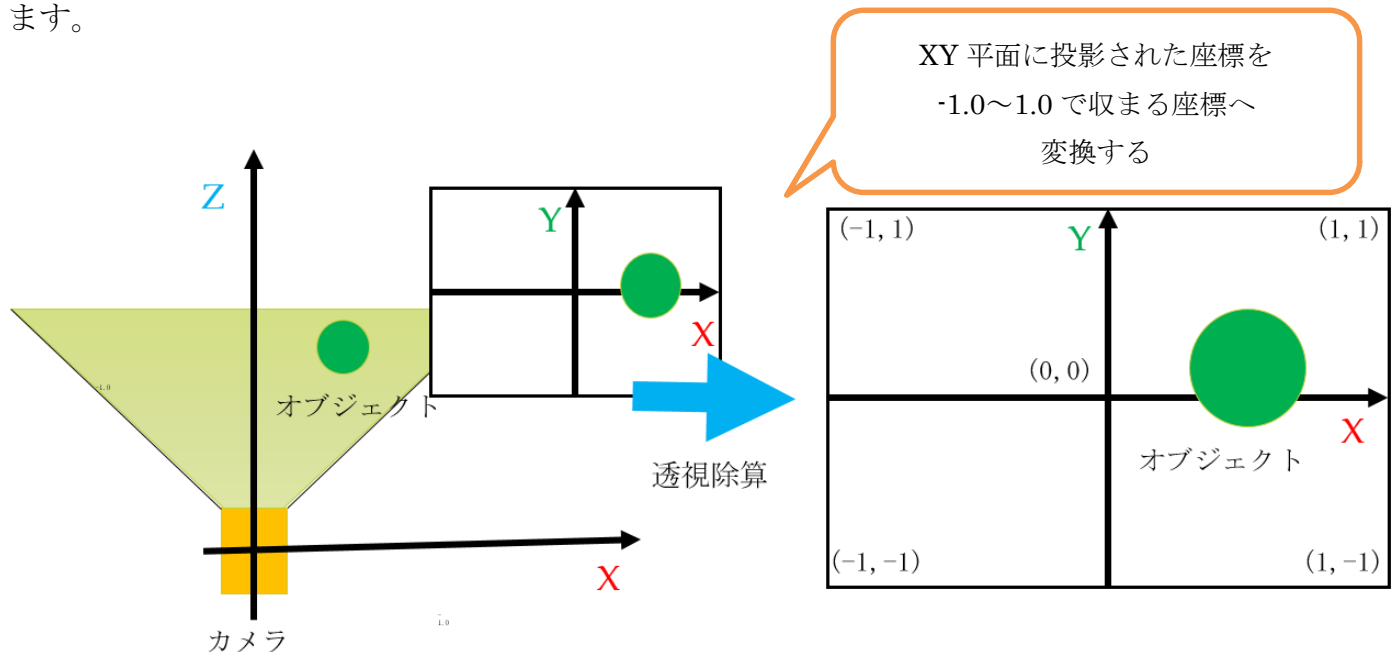
この計算により、
3D 空間から 2D 空間へ
変換されている



ゲームプログラミング改

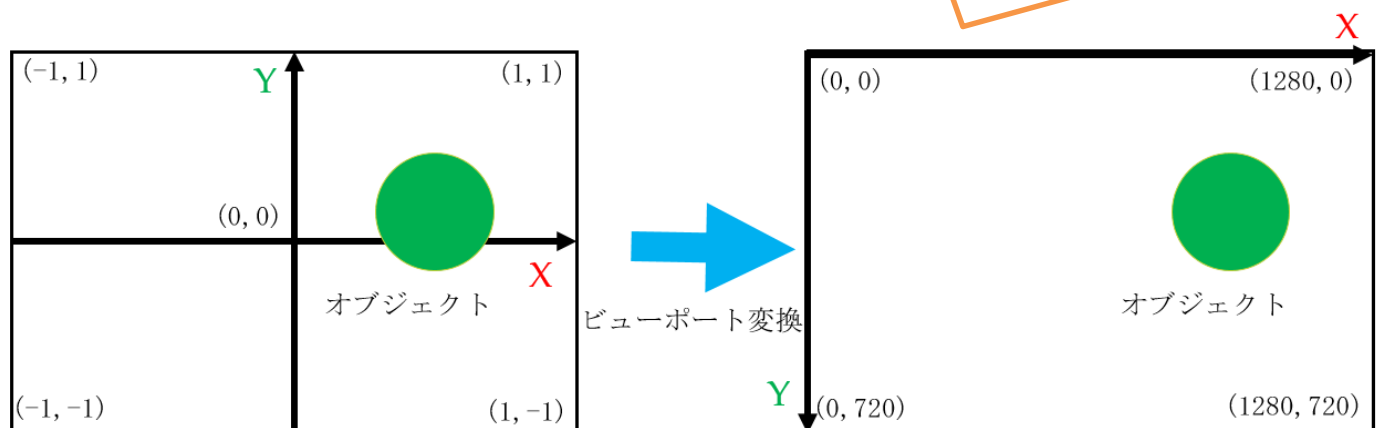
○透視除算

先ほどの透視投影変換の続きでクリッピング座標を正規化デバイス座標へ変換する計算です。正規化デバイス座標（NDC 座標）とは画面の座標を-1.0~1.0 で収まる座標のことです。左端の X 座標が-1.0 で右端の X 座標が 1.0 という感じです。透視投影変換と透視除算の計算はビュー座標とプロジェクション行列を乗算することで計算できます。



○ビューポート変換

正規化デバイス座標からスクリーン座標へ変換します。



○計算関数

上記までの流れを計算していくことでワールド座標からスクリーン座標への計算を算出できます。そして逆順に計算をすることでスクリーン座標からワールド座標への変換もできます。

ゲームプログラミング改

本来、一つずつ計算して実装していくのですが、DirectXMath では一発でこの計算結果を算出してくれる関数が用意されているので、今回はこちらを使いましょう。

ワールド座標からスクリーン座標へ変換する関数

```
ScreenPosition = DirectX::XMVector3Project(  
    WorldPosition,          // ワールド座標  
    ViewportX,              // ビューポート左上X位置  
    ViewportY,              // ビューポート左上Y位置  
    ViewportWidth,          // ビューポート幅  
    ViewportHeight,         // ビューポート高さ  
    ViewportMinZ,           // 深度値の範囲を表す最小値 (0.0でよい)  
    ViewportMaxZ,           // 深度値の範囲を表す最大値 (1.0でよい)  
    ProjectionTransform,    // プロジェクション行列  
    ViewTransform,          // ビュー行列  
    WorldTransform          // ワールド行列 (単位行列でよい)  
);
```

スクリーン座標からワールド座標へ変換する関数

```
WorldPosition = DirectX::XMVector3Unproject(  
    ScreenPosition,         // スクリーン座標 (Zの値は0.0~1.0)  
    ViewportX,              // ビューポート左上X位置  
    ViewportY,              // ビューポート左上Y位置  
    ViewportWidth,          // ビューポート幅  
    ViewportHeight,         // ビューポート高さ  
    ViewportMinZ,           // 深度値の範囲を表す最小値 (0.0でよい)  
    ViewportMaxZ,           // 深度値の範囲を表す最大値 (1.0でよい)  
    ProjectionTransform,    // プロジェクション行列  
    ViewTransform,          // ビュー行列  
    WorldTransform          // ワールド行列 (単位行列でよい)  
);
```

○敵キャラクターの頭の上に HP ゲージ(UI)を表示する

ワールド座標 (3D) からスクリーン座標 (2D) への座標変換を行い、敵キャラクターの頭上の HP ゲージ UI を表示するように実装しましょう。

Character.h

---省略---

ゲームプログラミング改

```
// キャラクター
class Character
{
public:
    ---省略---

    // 健康状態を取得
    int GetHealth() const { return health; }

    // 最大健康状態を取得
    int GetMaxHealth() const { return maxHealth; }

    ---省略---

protected:
    ---省略---
    int health = 5;
    int maxHealth = 5;
    ---省略---
};
```

SceneGame.h

```
---省略---
#include "Graphics/Sprite.h"

// ゲームシーン
class SceneGame : public Scene
{
public:
    ---省略---

private:
    // エネミーHPゲージ描画
    void RenderEnemyGauge(
        ID3D11DeviceContext* dc,
        const DirectX::XMFLOAT4X4& view,
        const DirectX::XMFLOAT4X4& projection
    );

private:
    ---省略---
    Sprite* gauge = nullptr;
};
```

SceneGame.cpp

```
---省略---

// 初期化
void SceneGame::Initialize()
{
    ---省略---
```

```
// ゲージスプライト
gauge = new Sprite();
}

// 終了化
void SceneGame::Finalize()
{
    // ゲージスプライト終了化
    if (gauge != nullptr)
    {
        delete gauge;
        gauge = nullptr;
    }

    ---省略---
}

---省略---

// 描画処理
void SceneGame::Render()
{
    ---省略---

    // 2Dスプライト描画
    {
        RenderEnemyGauge(dc, rc.view, rc.projection);
    }

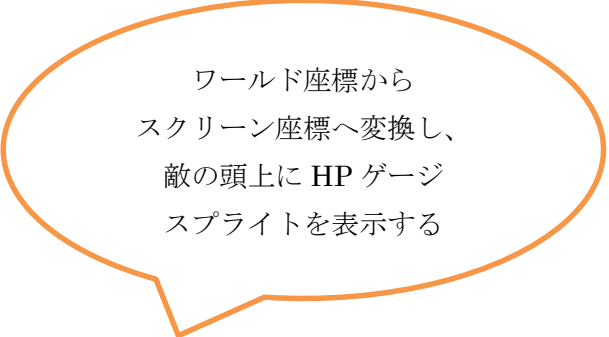
    ---省略---
}

// エネミーHPゲージ描画
void SceneGame::RenderEnemyGauge(
    ID3D11DeviceContext* dc,
    const DirectX::XMFLOAT4X4& view,
    const DirectX::XMFLOAT4X4& projection)
{
    // ビューポート
    D3D11_VIEWPORT viewport;
    UINT numViewports = 1;
    dc->RSGetViewports(&numViewports, &viewport);

    // 変換行列
    DirectX::XMMATRIX View = DirectX::XMLoadFloat4x4(&view);
    DirectX::XMMATRIX Projection = DirectX::XMLoadFloat4x4(&projection);
    DirectX::XMMATRIX World = DirectX::XMMatrixIdentity();

    // 全ての敵の頭上にHPゲージを表示
    EnemyManager& enemyManager = EnemyManager::Instance();
    int enemyCount = enemyManager.GetEnemyCount();

    for (int i = 0; i < enemyCount; ++i)
    {
        Enemy* enemy = enemyManager.GetEnemy(i);
```



ワールド座標から
スクリーン座標へ変換し、
敵の頭上に HP ゲージ
スプライトを表示する

```
}  
}
```

実装できたら実行確認をしてみましょう。

敵の頭上に HP ゲージが表示されていれば OK です。

敵にダメージを与えて HP ゲージが減っていくことも確認しておきましょう。



○マウスクリックした位置に敵を配置する

スクリーン座標 (2D) からワールド座標 (3D) への座標変換を行い、マウスクリックした位置に敵を配置するプログラムを実装しましょう。

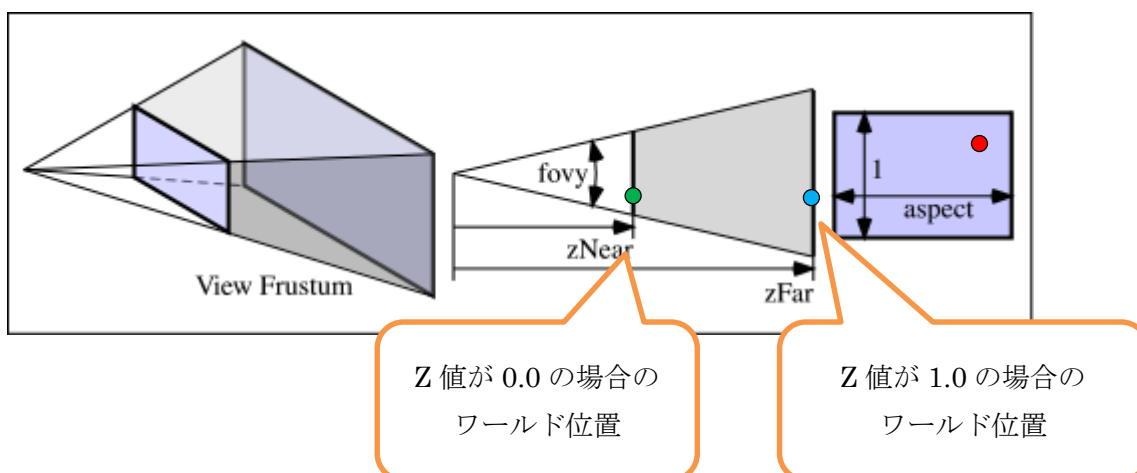
スクリーン座標 (2D) からワールド座標 (3D) への座標変換を行うには 2D の Z 座標に気を付けましょう。

スクリーン座標の XY 座標はマウスカーソルの位置をそのまま入れれば良いですが、Z 座標はビューポートの最小深度値から最大深度値の間の値を設定する必要があります。

一般的にビューポートの最小深度値は 0.0、最大深度値は 1.0 に設定されることが多いです。

下図、右側の赤●のスクリーン座標に対して Z 値を 0.0 で座標変換を行った場合、中央の図の緑●の位置が算出されます。

また、Z 値を 1.0 で座標変換を行った場合は中央の図の青●の位置が算出されます。



このことを理解した上で敵の配置を行います。

ゲームプログラミング改

敵の配置にはレイキャストを使って配置します。

上図の緑●と青●の位置を始点と終点とすることでレイになります。

このレイを使ってレイキャストすることでレイとステージの交差した位置に敵を配置するようにします。

デバッグ的な内容の実装なので、仮の実装として先ほどの HP ゲージ描画関数内の続きで実装します。ちゃんと実装したい人は専用の関数をつくるようにしましょう。

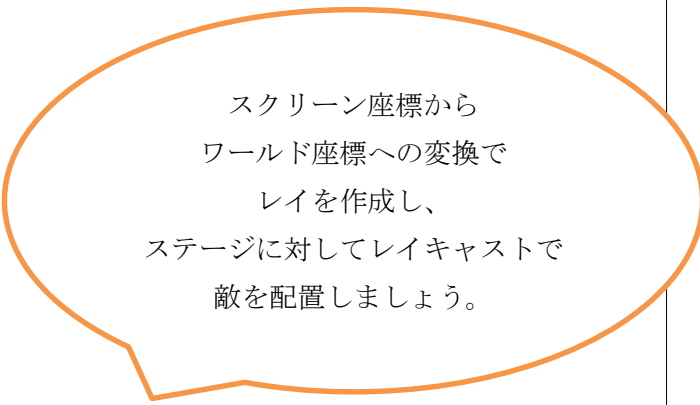
SceneGame.cpp

```
---省略---
#include "Input/Input.h"

---省略---

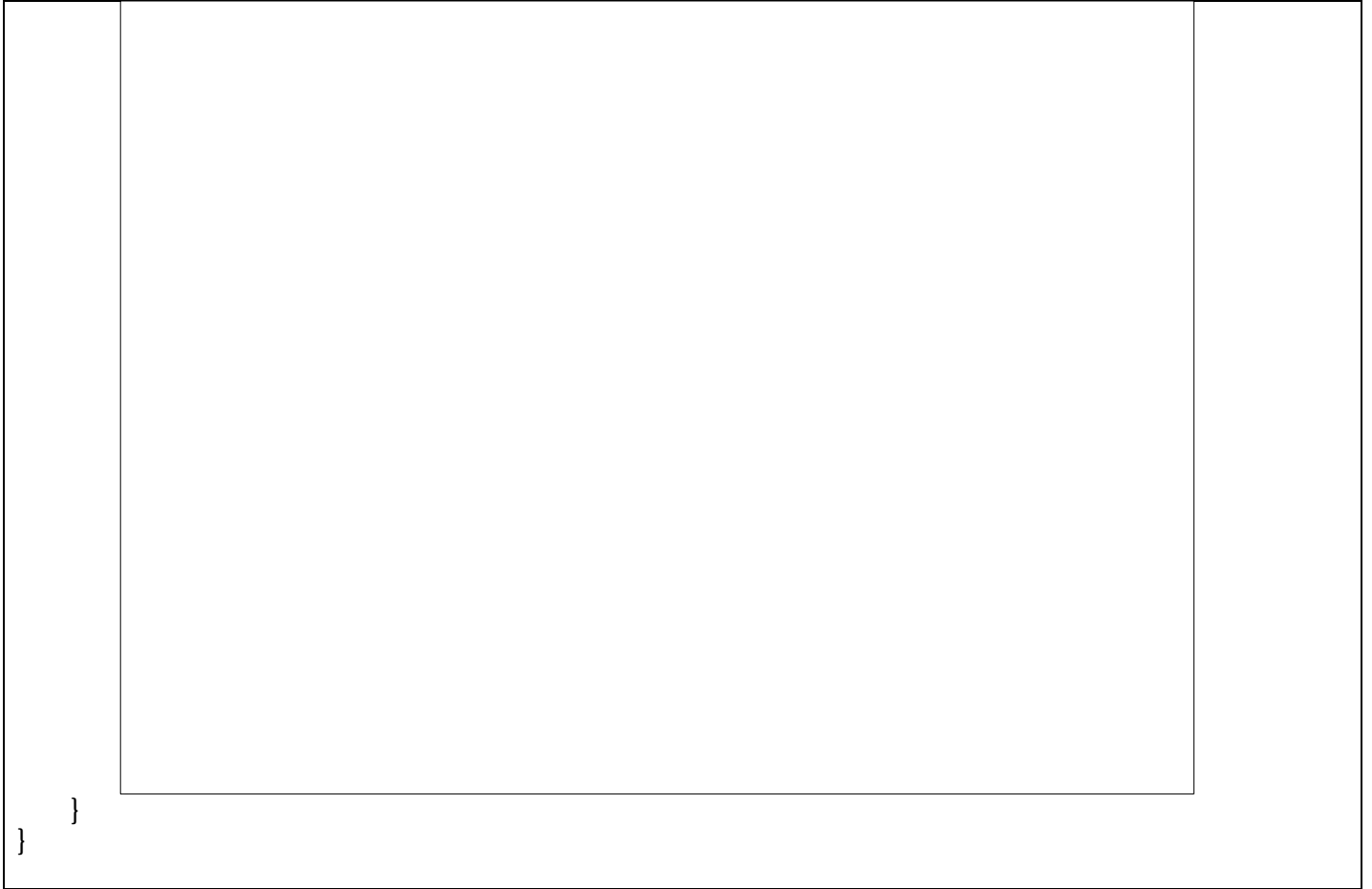
// エネミーHPゲージ描画
void SceneGame::RenderEnemyGauge(
    ID3D11DeviceContext* dc,
    const DirectX::XMFLOAT4X4& view,
    const DirectX::XMFLOAT4X4& projection)
{
    ---省略---

    // エネミー配置処理
    Mouse& mouse = Input::Instance().GetMouse();
    if (mouse.GetButtonDown() & Mouse::BTN_LEFT)
    {
        // マウスカーソル座標を取得
        DirectX::XMFLOAT3 screenPosition;
        screenPosition.x = static_cast<float>(mouse.GetPositionX());
        screenPosition.y = static_cast<float>(mouse.GetPositionY());
```



スクリーン座標から
ワールド座標への変換で
レイを作成し、
ステージに対してレイキャストで
敵を配置しましょう。

ゲームプログラミング改



実装出来たら実行確認をしてみましょう。

ゲームシーンでマウスクリックをした場所に敵が配置できていれば **OK** です。

お疲れさまでした。