

3D アクションゲームプログラミング

○評価要件

- ☒ 衝突形状の可視化
- ☒ プレイヤーと敵の衝突処理（球 vs 球）
- ☒ 敵同士の衝突処理（球 vs 球）

○概要

今回は衝突判定を学習します。

一般的な3Dゲームの場合、表示されているキャラクター同士が重なった場合にめり込むことはありません。衝突判定を行い、重なった場合に押し出し処理を行っているからです。

衝突処理は様々あり、単純に衝突範囲に侵入したことを検知するだけなのか、衝突範囲に侵入した後、重ならないように押し出し処理をするのか、状況によって使い分けます。

また、衝突判定の形状も様々です。

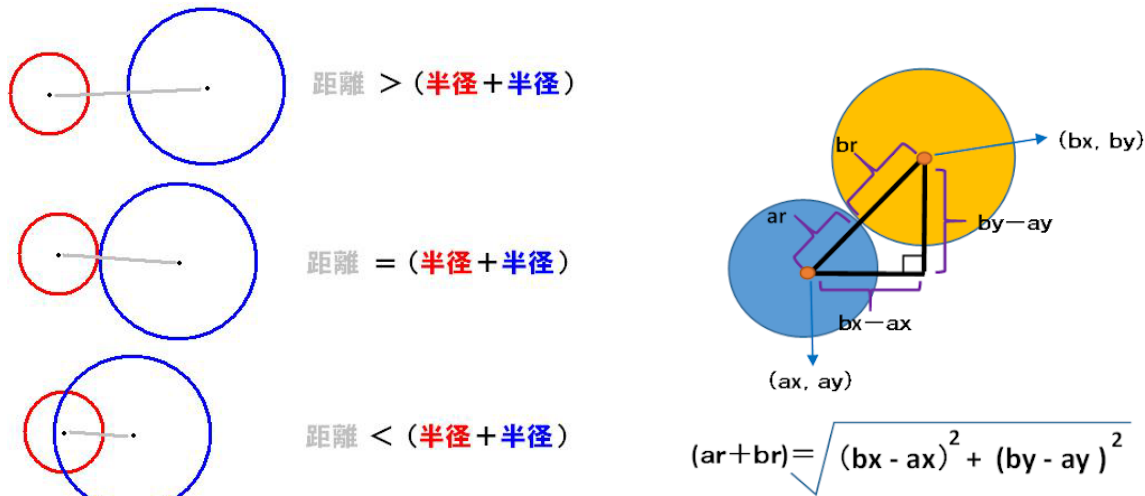
球、箱、カプセル、円柱、メッシュ、などの様々な形状の組み合わせの衝突判定があります。今回は一番簡単な球と球の衝突処理を実装します。

プレイヤーと敵を球の衝突処理で重ならないようにしましょう。

3D アクションゲームプログラミング

○球の衝突判定

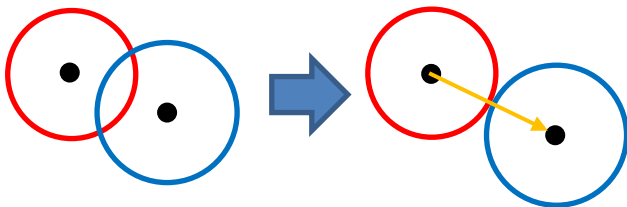
下図を見てわかるように2つの球には「位置」と「半径」があり、距離が2つの球の半径よりも小さくなった場合にめり込んでいます。



今回はキャラクター同士が重なった場合にめり込まないように押し出しましょう。

押し出し処理の方法はゲームの仕様によって考える必要がありますが、一番簡単な方法で実装しましょう。

2つの球の位置からベクトルを作成し、一方の位置(赤い球)から重ならないように押し出します。



○当たり判定の可視化

通常は衝突判定の形状が表示されることはありませんが、開発をしていく上で衝突判定の形状が可視化されていると色々と便利です。

まずは衝突判定に必要な「半径」の情報を実装しましょう。

今回はプレイヤーとエネミーの基底クラスのキャラクタークラスに情報を持たせます。

Character.h を開き、下記プログラムコードを追記しましょう。

Character.h

---省略---

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

3D アクションゲームプログラミング

```
---省略---

// 半径取得
float GetRadius() const { return radius; }

protected:
    ---省略---
    float radius = 0.5f;
};
```

続いて Player.h と Player.cpp を開き、デバッグ球を表示するプログラムを実装しましょう。

Player.h

```
---省略---

// プレイヤー
class Player : public Character
{
public:
    ---省略---

    // デバッグプリミティブ描画
    void DrawDebugPrimitive();

    ---省略---
};
```

Player.cpp

```
---省略---

#include "Graphics/Graphics.h"

---省略---

// デバッグプリミティブ描画
void Player::DrawDebugPrimitive()
{
    DebugRenderer* debugRenderer = Graphics::Instance().GetDebugRenderer();

    // 衝突判定用のデバッグ球を描画
    debugRenderer->DrawSphere(position, radius, DirectX::XMFLLOAT4(0, 0, 0, 1));
}

---省略---
```

デバッグレンダラはこのプロジェクトであらかじめ用意しているデバッグ用の描画システムです。このシステムについての説明は特にしません。プログラムコードをよく読み、参考にしてください。

3D アクションゲームプログラミング

今後、開発する上で様々な形状のデバッグ表示をする状況が多々あります。
このクラスを拡張して様々な形状の表示をできるようにしましょう。

デバッグ球をシーンに表示しましょう。

SceneGame.cpp

```
// 描画処理
void SceneGame::Render ()
{
    ---省略---
    // 3Dデバッグ描画
    {
        // プレイヤーデバッグプリミティブ描画
        [ ]

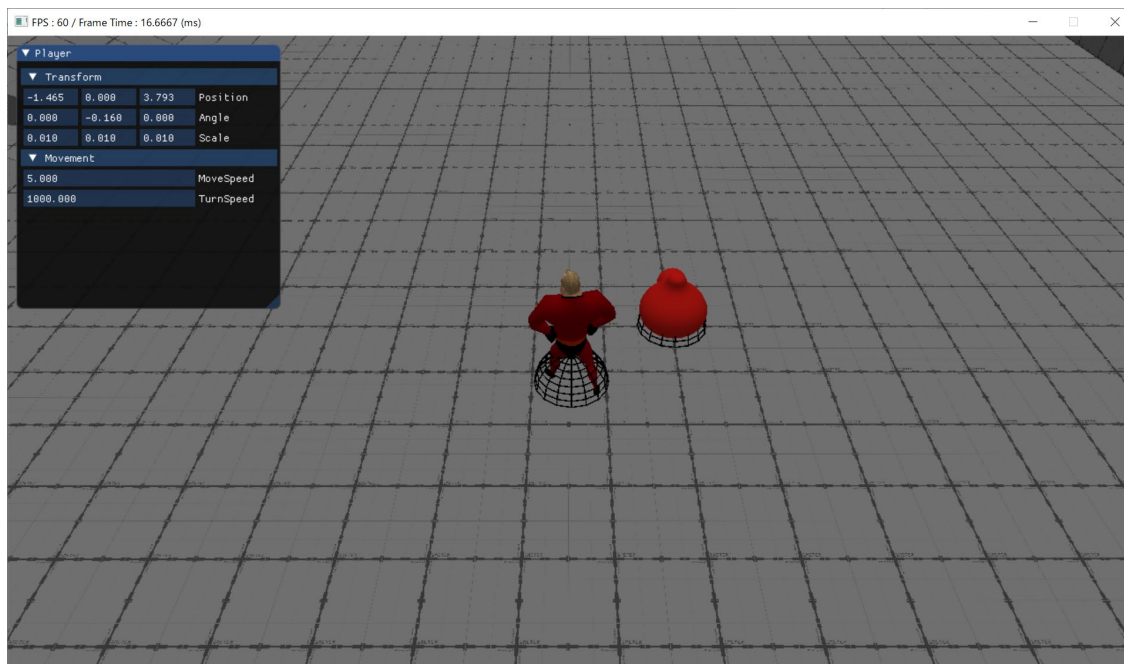
        // ラインレンダラ描画実行
        graphics.GetLineRenderer()->Render(dc, rc.view, rc.projection);

        // デバッグレンダラ描画実行
        graphics.GetDebugRenderer()->Render(dc, rc.view, rc.projection);
    }

    ---省略---
}
```

実行確認してみましょう。

プレイヤーの足元に黒い球が表示されていれば OK です。



敵も同じように実装しましょう。

3D アクションゲームプログラミング

Enemy.h

```
---省略---

// エネミー
class Enemy : public Character
{
public:
    ---省略---

    // デバッグプリミティブ描画
    virtual void DrawDebugPrimitive();
};
```

Enemy.cpp

```
#include "Enemy.h"
#include "Graphics/Graphics.h"

// デバッグプリミティブ描画
void Enemy::DrawDebugPrimitive()
{
    
}
```

EnemyManager.h

```
---省略---

// エネミーマネージャー
class EnemyManager
{
private:
    ---省略---

public:
    ---省略---

    // デバッグプリミティブ描画
    void DrawDebugPrimitive();

    ---省略---
};
```

EnemyManager.cpp

```
---省略---

// デバッグプリミティブ描画
void EnemyManager::DrawDebugPrimitive()
{
    
}
```

3D アクションゲームプログラミング

```
}  
---省略---
```

SceneGame.cpp

```
---省略---  
  
// 描画処理  
void SceneGame::Render()  
{  
    ---省略---  
  
    // 3Dデバッグ描画  
    {  
        ---省略---  
  
        // エネミーデバッグプリミティブ描画  
          
  
        // ラインレンダラ描画実行  
        ---省略---  
    }  
  
    ---省略---  
}
```

プレイヤーとエネミーの衝突範囲が可視化されました。

○プレイヤーと敵の衝突処理

次は本題の衝突処理を実装しましょう。衝突処理は様々な場面で使用します。

衝突関係の処理をするコリジョンクラスを作成しましょう。

Collision.cpp と Collision.h を作成し、下記プログラムコードを記述しましょう。

Collision.h

```
#pragma once  
  
#include <DirectXMath.h>  
  
// コリジョン  
class Collision  
{  
public:  
    // 球と球の交差判定  
    static bool IntersectSphereVsSphere(  
        const DirectX::XMFLAT3& positionA,
```

3D アクションゲームプログラミング

```
float radiusA,  
const DirectX::XMFLOAT3& positionB,  
float radiusB,  
DirectX::XMFLOAT3& outPositionB  
);  
};
```

Collision.cpp

```
#include "Collision.h"  
  
// 球と球の交差判定  
bool Collision::IntersectSphereVsSphere(  
const DirectX::XMFLOAT3& positionA,  
float radiusA,  
const DirectX::XMFLOAT3& positionB,  
float radiusB,  
DirectX::XMFLOAT3& outPositionB)  
{  
    // A→Bの単位ベクトルを算出  
    DirectX::XMVECTOR PositionA =   
    DirectX::XMVECTOR PositionB =   
    DirectX::XMVECTOR Vec =   
    DirectX::XMVECTOR LengthSq =   
    float lengthSq;  
    DirectX::XMStoreFloat(&lengthSq, LengthSq);  
  
    // 距離判定  
    float range =   
    if (  )  
    {  
        return false;  
    }  
  
    // AがBを押し出す  
      
    DirectX::XMStoreFloat3(&outPositionB, );  
  
    return true;  
}
```

実装が終わったら、プレイヤークラスで全てのエネミーと総当たりの衝突処理を実装します。
まずはエネミーマネージャーからエネミーを取得する関数を実装します。

EnemyManager.hを開き、下記プログラムコードを追記しましょう。

EnemyManager.h

```
---省略---  
  
// エネミーマネージャー  
class EnemyManager
```

3D アクションゲームプログラミング

```
{
public:
    ---省略---

    // エネミー数取得
    int GetEnemyCount() const { return static_cast<int>(enemies.size()); }

    // エネミー取得
    Enemy* GetEnemy(int index) { return enemies.at(index); }

    ---省略---
};
```

エネミーの情報が取得できるようになったのでプレイヤークラスに衝突処理を実装しましょう。

Player.h

```
---省略---

// プレイヤー
class Player : public Character
{
    ---省略---
private:
    ---省略---

    // プレイヤーとエネミーとの衝突処理
    void CollisionPlayerVsEnemies();

    ---省略---
};
```

Player.cpp

```
---省略---
#include "EnemyManager.h"
#include "Collision.h"

---省略---

// 更新処理
void Player::Update(float elapsedTime)
{
    ---省略---

    // プレイヤーと敵との衝突処理
    CollisionPlayerVsEnemies();



    // オブジェクト行列を更新
    ---省略---
}

---省略---
```


3D アクションゲームプログラミング

```
// プレイヤーとエネミーとの衝突処理
void Player::CollisionPlayerVsEnemies()
{
    EnemyManager& enemyManager = EnemyManager::Instance();

    // 全ての敵と総当たりで衝突処理
    int enemyCount = enemyManager.GetEnemyCount();
    for (int i = 0; i < enemyCount; ++i)
    {
        Enemy* enemy = enemyManager.GetEnemy(i);

        // 衝突処理
        DirectX::XMFLOAT3 outPosition;
        if (Collision::IntersectSphereVsSphere(
            
        ))
        {
            // 押し出し後の位置設定
            
        }
    }
}
```

実装ができたなら実行確認してみましょう。

プレイヤーをスライムの方へ移動させてめり込まず押し出しができていれば OK です。

○敵同士の衝突処理

プレイヤーと敵の衝突処理ができたので、敵同士の衝突処理も挑戦しましょう。

EnemyManager.h

```
---省略---

// エネミーマネージャー
class EnemyManager
{
    ---省略---

private:
    // エネミー同士の衝突処理
    void CollisionEnemyVsEnemies();

    ---省略---
};
```

3D アクションゲームプログラミング

EnemyManager.cpp

```
---省略---
#include "Collision.h"

// 更新処理
void EnemyManager::Update(float elapsedTime)
{
    ---省略---

    // 敵同士の衝突処理
    CollisionEnemyVsEnemies();
}

---省略---

// エネミー同士の衝突処理
void EnemyManager::CollisionEnemyVsEnemies()
{
    
}
}
```

敵同士の衝突判定を確認するためにもう一体敵を配置しましょう。

SceneGame.cpp

```
---省略---

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

    // エネミー初期化
    EnemyManager& enemyManager = EnemyManager::Instance();
    EnemySlime* slime = new EnemySlime();
}
```

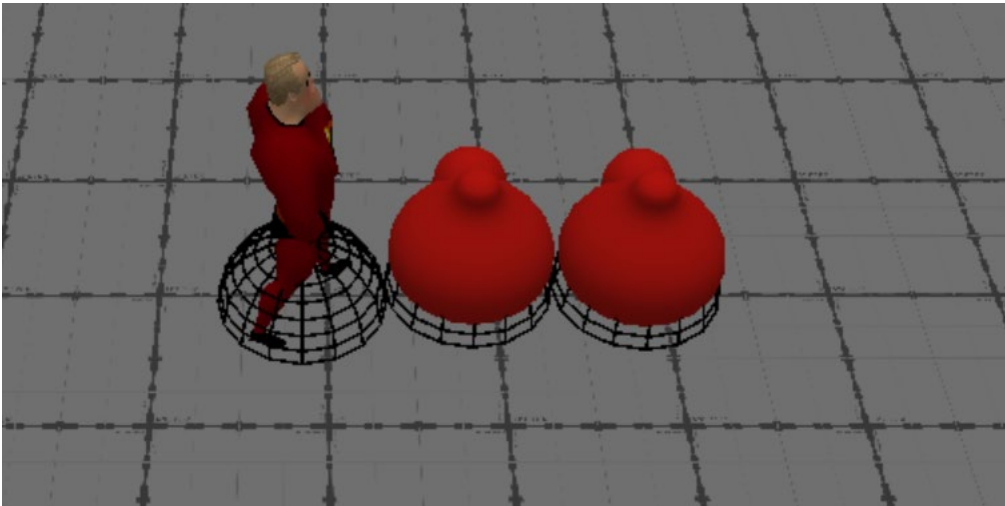
3D アクションゲームプログラミング

```
slime->SetPosition(DirectX::XMVECTOR(0, 0, 5));  
enemyManager.Register(slime);  
for (int i = 0; i < 2; ++i)  
{  
    EnemySlime* slime = new EnemySlime();  
    slime->SetPosition(DirectX::XMVECTOR(i * 2.0f, 0, 5));  
    enemyManager.Register(slime);  
}
```

実行確認をしてみましょう。

スライム自身は自分で動くことはできないので、プレイヤーが片方のスライムを押して行ってもう片方のスライムに衝突させてみましょう。

スライム同士がめり込むことなく、衝突処理ができていると OK です。



お疲れさまでした。