

# ゲームプログラミングⅢ

## ○評価要件

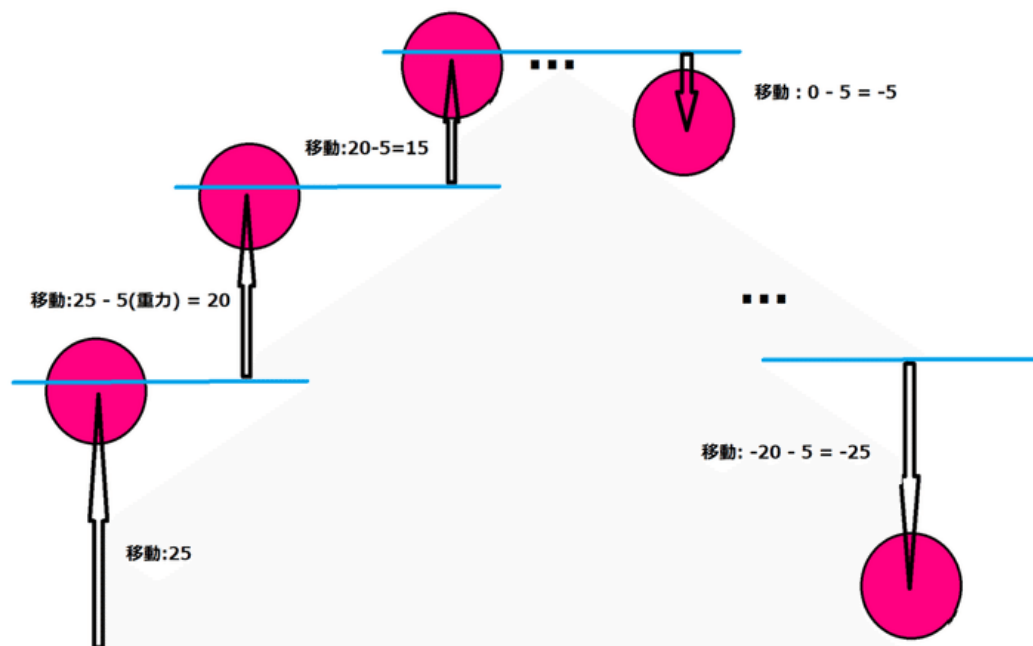
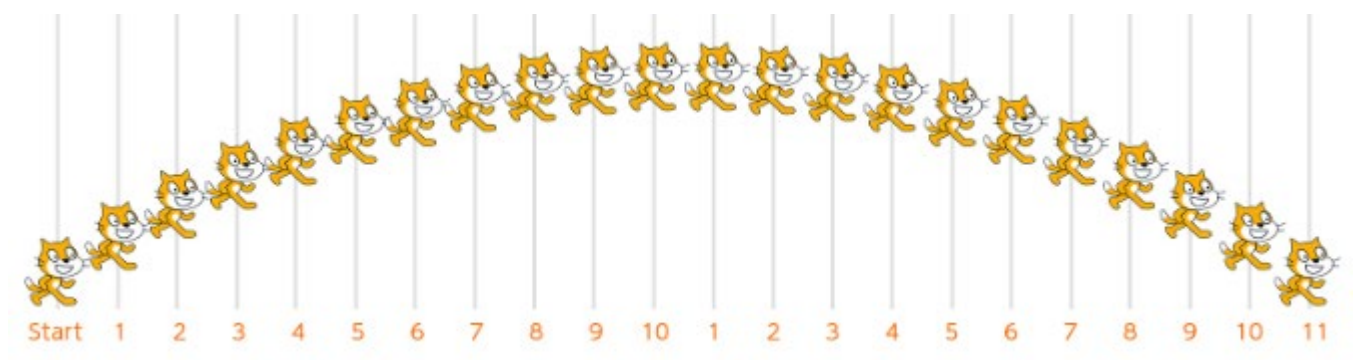
- ☑ジャンプ処理
- ☑ジャンプ回数制限
- ☑円柱形状の衝突処理
- ☑踏んづけ処理

## ○概要

今回はジャンプ処理を実装します。

ジャンプは上方向に力を加えることによって上昇しますが、上方向の力が弱まってくると落下します。これは「重力」が働いているからです。

今まではスティックを入力することで移動していましたが、ジャンプを実装するにあたって外からの力が働くようなプログラムを実装する必要があります。



## ゲームプログラミングⅢ

### ○ジャンプに必要な情報

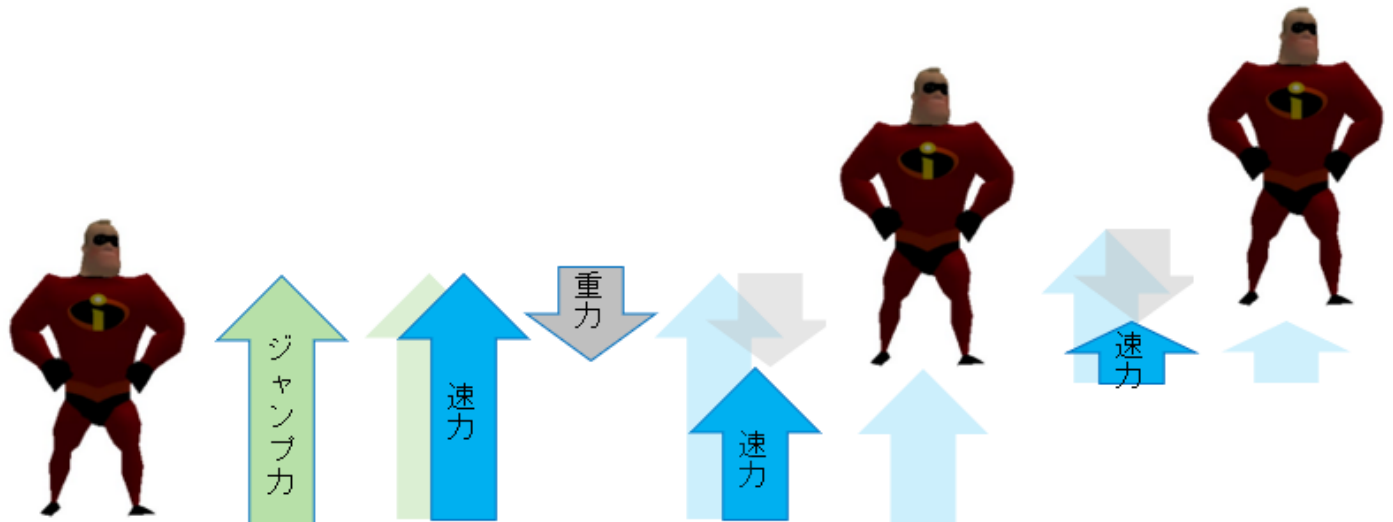
ジャンプ処理は単純でジャンプ力、重力、速力の3つで実現できます。

ジャンプ力はボタンを押した瞬間に上方向に与える力のことです。

重力は毎フレーム加算する下方向への力のことです。

速力は毎フレーム加算する移動量のことです。

この速力を毎フレーム動的に計算することでジャンプや慣性移動などの処理を実現できます。



### ○ジャンプ入力処理

プレイヤーにジャンプ入力処理を実装しましょう。

Player.h と Player.cpp を開き下記プログラムコードを追記しましょう。

Player.h

```
---省略---  
  
// プレイヤー  
class Player : public Character  
{  
public:  
    ---省略---  
  
    // ジャンプ処理  
    void Jump(float speed);  
  
    // 速力処理更新  
    void UpdateVelocity(float elapsedTime);  
  
    // ジャンプ入力処理  
    void InputJump();  
  
private:  
    ---省略---
```

## ゲームプログラミングⅢ

```
float      jumpSpeed = 12.0f;
float      gravity = -30.0f;
DirectX::XMFLOAT3 velocity = { 0, 0, 0 };
};
```

### Player.cpp

```
---省略---

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

    // ジャンプ入力処理
    InputJump();

    // 速力処理更新
    UpdateVelocity(elapsedTime);

    // プレイヤーと敵との衝突処理
    ---省略---
}

---省略---

// ジャンプ処理
void Player::Jump(float speed)
{
    // 上方向の力を設定
    
}

// 速力処理更新
void Player::UpdateVelocity(float elapsedTime)
{
    // 重力処理
    

    // 移動処理
    

    // 地面判定
    if (position.y < 0.0f)
    {
        position.y = 0.0f;
        velocity.y = 0.0f;
    }
}

// ジャンプ入力処理
void Player::InputJump()
{
    GamePad& gamePad = Input::Instance().GetGamePad();
    if (gamePad.GetButtonDown() & GamePad::BTN_A)
```

とりあえず、地面の高さは  
0.0 として実装しましょ  
う。

## ゲームプログラミングⅢ

```
{  
    Jump(jumpSpeed);  
}
```

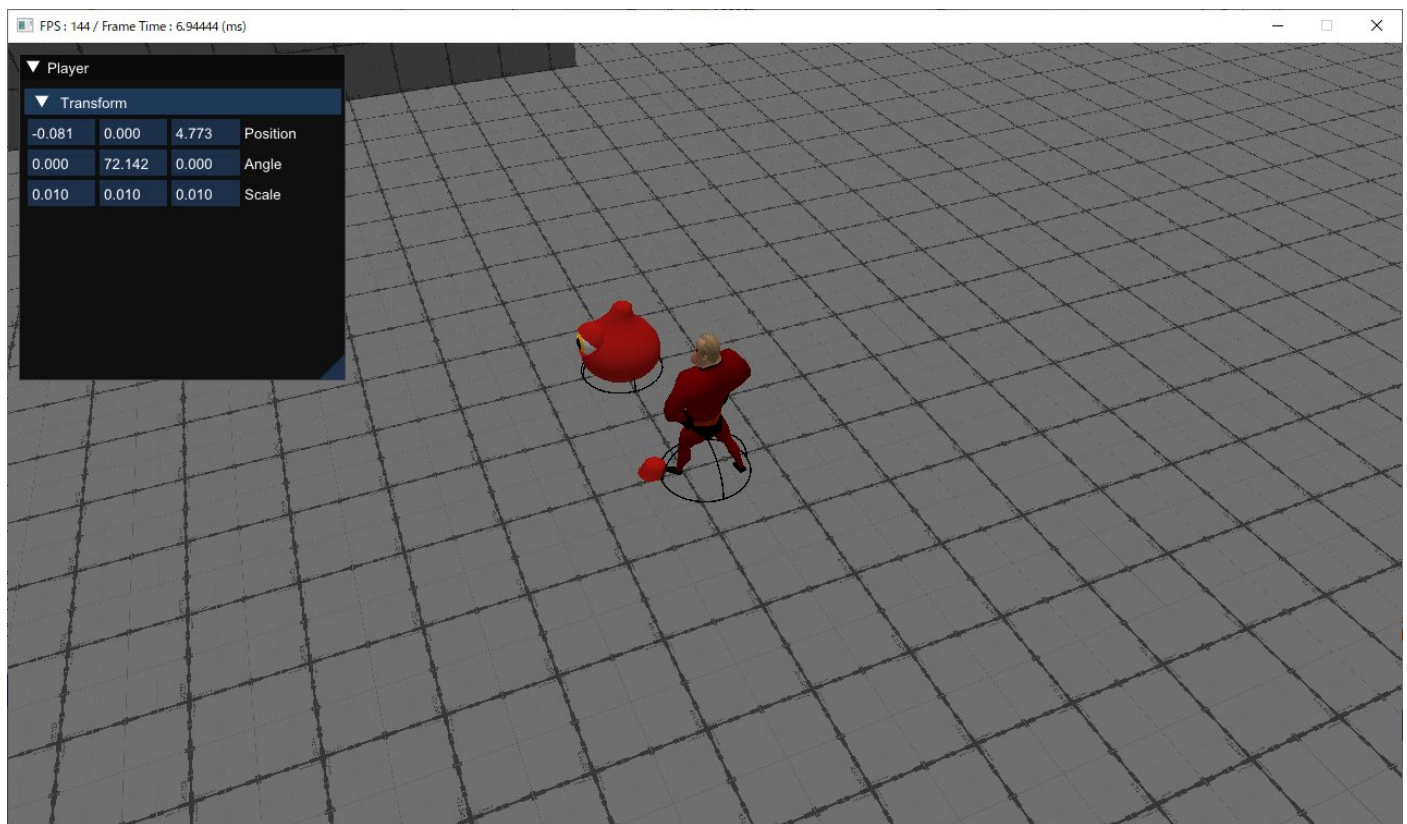
実装できたら実行確認を行きましょう。

A ボタン(キーボード Z)を押してジャンプできていれば OK です。

さて、ジャンプ処理ができたので、ここで敵との当たり判定をもう一度確認してみましょう。

今度はジャンプで敵を踏みつけてみて下さい。

今までの処理が完璧に実装出来ていれば下図のように敵が地面にめり込んでいるはずです。



これは、プレイヤーは地面判定をしましたが、敵は地面判定をしていないためです。

単純にプレイヤーで実装した内容をそのまま敵に対しても実装すれば良いだけなのですが、同じ処理を何度も実装するのはナンセンスです。

### ○リファクタリング

プレイヤーで実装した処理の一部は敵にも流用できそうなものがいくつかあります。

プレイヤーと敵はキャラクタークラスという共通の基底クラスがあるので、プレイヤークラスの関数をキャラクタークラスへ引っ越ししましょう。

## ゲームプログラミングⅢ

### Player.h

```
---省略---

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

private:
    ---省略---

    // 移動処理
    void Move(float elapsedTime, float vx, float vz, float speed);

    // 旋回処理
    void Turn(float elapsedTime, float vx, float vz, float speed);

    // ジャンプ処理
    void Jump(float speed);

    // 速力処理更新
    void UpdateVelocity(float elapsedTime);

    ---省略---

private:
    ---省略---
    float gravity = -30.0f;
    DirectX::XMVECTOR velocity = { 0, 0, 0 };
};
```

これらの関数を  
キャラクタークラスへ  
引っ越し

変数も引っ越し

### Character.h

```
---省略---

// キャラクター
class Character
{
    ---省略---
protected:
    // 移動処理
    void Move(float elapsedTime, float vx, float vz, float speed);

    // 旋回処理
    void Turn(float elapsedTime, float vx, float vz, float speed);

    // ジャンプ処理
    void Jump(float speed);

    // 速力処理更新
    void UpdateVelocity(float elapsedTime);

protected:
    ---省略---
```

継承先で使えるように  
protected にすること

## ゲームプログラミングⅢ

```
float gravity = -30.0f;
DirectX::XMFLOAT3 velocity = { 0, 0, 0 };
};
```

Player.cpp と Character.cpp も引っ越し作業を行ってください。

引っ越しが終わったら、それらの関数を使ってエネミーが地面にめり込まないように実装しましょう。

### EnemySlime.cpp

```
---省略---

// 更新処理
void EnemySlime::Update(float elapsedTime)
{
    // 速力処理更新
    UpdateVelocity(elapsedTime);

    ---省略---
}
```

実装出来たら実行確認してみてください。

敵を踏みつけてキャラクターにも地面にもめり込まなかったら OK です。

### ○着地判定

キャラクター制御を実装する上で地面に接地しているか、着地した瞬間などを判定することが多々あります。

Character.cpp と 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:
    ---省略---

    // 地面に接地しているか
    bool IsGround() const { return isGround; }

protected:
    ---省略---

    // 着地した時に呼ばれる
    virtual void OnLanding() {}

protected:
```

## ゲームプログラミングⅢ



```
---省略---
bool isGround = false;
};
```

### Character.cpp

```
---省略---

// 速力処理更新
void Character::UpdateVelocity(float elapsedTime)
{
    ---省略---

    // 地面判定
    if (position.y < 0.0f)
    {
        position.y = 0.0f;
        velocity.y = 0.0f;

        // 着地した
        
    }
    else
    {
        
    }
}
```

isGround 変数を制御し、  
正しく OnLanding()を  
呼び出そう

### ○ジャンプ回数制限

現時点では無限にジャンプができるようになっています。

着地判定ができるようになったのでプレイヤーのジャンプは2段ジャンプまでに制限しましょう。

### Player.h

```
---省略---

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

protected:
    // 着地した時に呼ばれる
    void OnLanding() override;

    ---省略---


private:
```

## ゲームプログラミングⅢ


```
---省略---
int          jumpCount = 0;
int          jumpLimit = 2;
};
```

Player.cpp

```
---省略---

// 着地した時に呼ばれる
void Player::OnLanding()
{
    
}

---省略---

// ジャンプ入力処理
void Player::InputJump()
{
    // ボタン入力でジャンプ（ジャンプ回数制限つき）
    GamePad& gamePad = Input::Instance().GetGamePad();
    if (gamePad.GetButtonDown() & GamePad::BTN_A)
    {
        
    }
}
}
```

実行確認してみましょう。

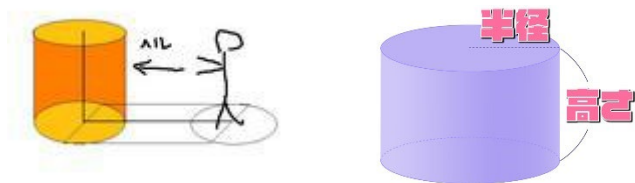
ジャンプが2回まで制限できていれば OK です。

### ○円柱の当たり判定

現状、プレイヤーも敵も球形状の衝突判定を行っていますが、3D モデルの形状と合っていません。

見た目と衝突の形状を合わせるために円柱形状の衝突判定にしましょう。

今回実装する円柱は回転には対応しない簡単な処理にします。

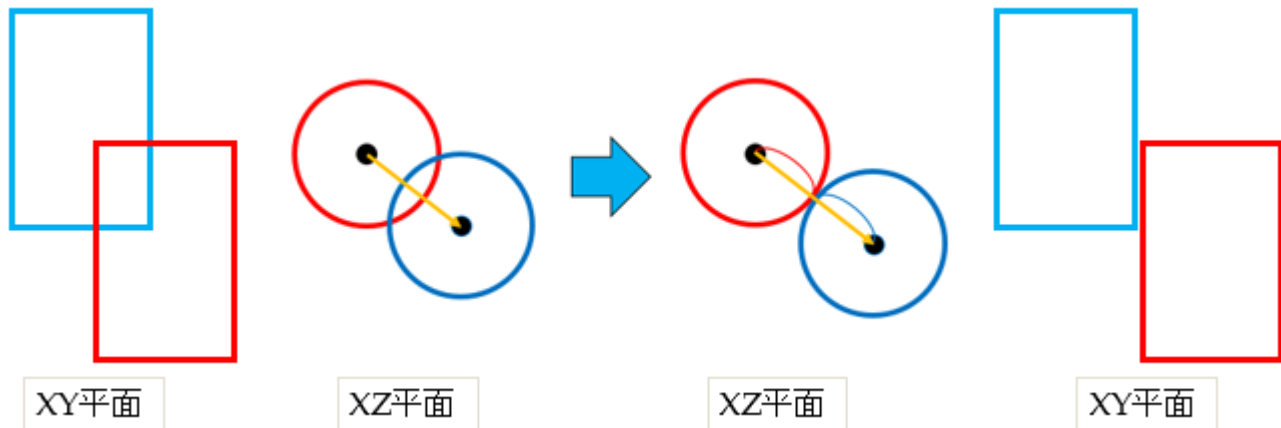


円柱の当たり判定に必要な情報は「位置」「半径」「高さ」の3つです。



## ゲームプログラミングⅢ

実装方法は単純で XZ 平面は球と同じく距離判定を行い、Y 平面に位置と高さのチェックをすることで実装できます。



Character.h には「位置」と「半径」が既にあるので「高さ」を追加しましょう。

Character.h

```
---省略---  
  
// キャラクター  
class Character  
{  
public:  
    ---省略---  
  
    // 高さ取得  
    float GetHeight() const { return height; }  
  
    ---省略---  
  
protected:  
    ---省略---  
    float          height = 2.0f;  
};
```

次は円柱の衝突形状を可視化しましょう。

プレイヤーと敵のデバッグプリミティブの表示を球から円柱に変更しましょう。

同じようにエネミーの表示も円柱に変更しましょう。

Character.cpp

```
---省略---  
  
// デバッグプリミティブ描画  
void Character::RenderDebugPrimitive(const RenderContext& rc, ShapeRenderer* renderer)
```

## ゲームプログラミングⅢ

```
{
// 衝突判定用のデバッグ球を描画
renderer->RenderSphere(rc, position, radius, DirectX::XMFLLOAT4(0, 0, 0, 1));
// 衝突判定用のデバッグ円柱を描画
renderer->RenderCylinder(rc, position, radius, height, DirectX::XMFLLOAT4(0, 0, 0, 1));
}
```

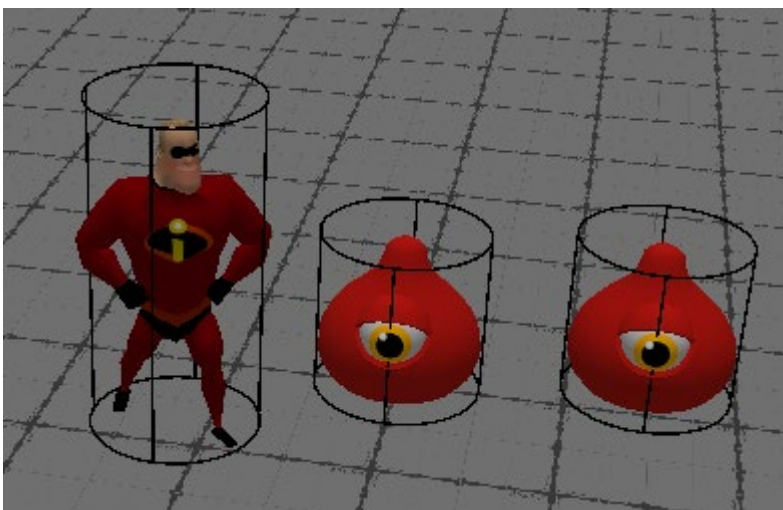
衝突形状が 3D モデルの表示に合うように調整しましょう。

### EnemySlime.cpp

```
---省略---

// コンストラクタ
EnemySlime::EnemySlime()
{
    ---省略---

    // 幅、高さ設定
    radius = 0.5f;
    height = 1.0f;
}
```



衝突判定に必要な準備ができたので衝突処理を実装しましょう。

### Collision.h

```
---省略---

// コリジョン
class Collision
{
public:
    ---省略---
```

```
// 円柱と円柱の交差判定
static bool IntersectCylinderVsCylinder(
    const DirectX::XMFLOAT3& positionA,
    float radiusA,
    float heightA,
    const DirectX::XMFLOAT3& positionB,
    float radiusB,
    float heightB,
    DirectX::XMFLOAT3& outPositionB
);
};
```

## Collision.cpp

---省略---

```
// 円柱と円柱の交差判定
bool Collision::IntersectCylinderVsCylinder(
    const DirectX::XMFLOAT3& positionA,
    float radiusA,
    float heightA,
    const DirectX::XMFLOAT3& positionB,
    float radiusB,
    float heightB,
    DirectX::XMFLOAT3& outPositionB)
{
    // Aの足元がBの頭より上なら当たっていない
    if ( )
    {
        return false;
    }
    // Aの頭がBの足元より下なら当たっていない
    if ( )
    {
        return false;
    }
    // XZ平面での範囲チェック
    float vx = 
    float vz = 
    float range = 
    float distXZ = 
    if ( )
    {
        return false;
    }
    // 単位ベクトル化
      

    // AがBを押し出す
    outPositionB.x = 
    outPositionB.y = 
    outPositionB.z = 

    return true;
}
```

## ゲームプログラミングⅢ

```
}
```

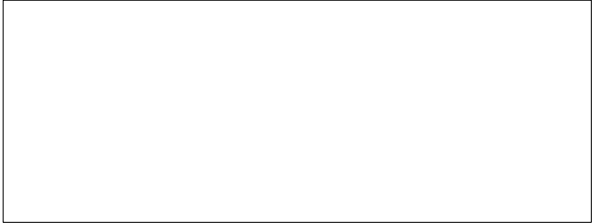
衝突処理が実装できたらプレイヤーと敵の衝突処理を球×球から円柱×円柱に変更しましょう。

### Player.cpp

```
---省略---

// 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(
            position, radius,
            enemy->GetPosition(),
            enemy->GetRadius(),
            outPosition))
        {
            // 押し出し後の位置設定
            enemy->SetPosition(outPosition);
        }
        if (Collision::IntersectCylinderVsCylinder(
            
        ))
        {
            // 押し出し後の位置設定
            enemy->SetPosition(outPosition);
        }
    }
}
```


### EnemyManager.cpp

```
---省略---

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

```

```
size_t enemyCount = enemies.size();
for (int i = 0; i < enemyCount; ++i)
{
    Enemy* enemyA = enemies.at(i);
    for (int j = i + 1; j < enemyCount; ++j)
    {
        Enemy* enemyB = enemies.at(j);

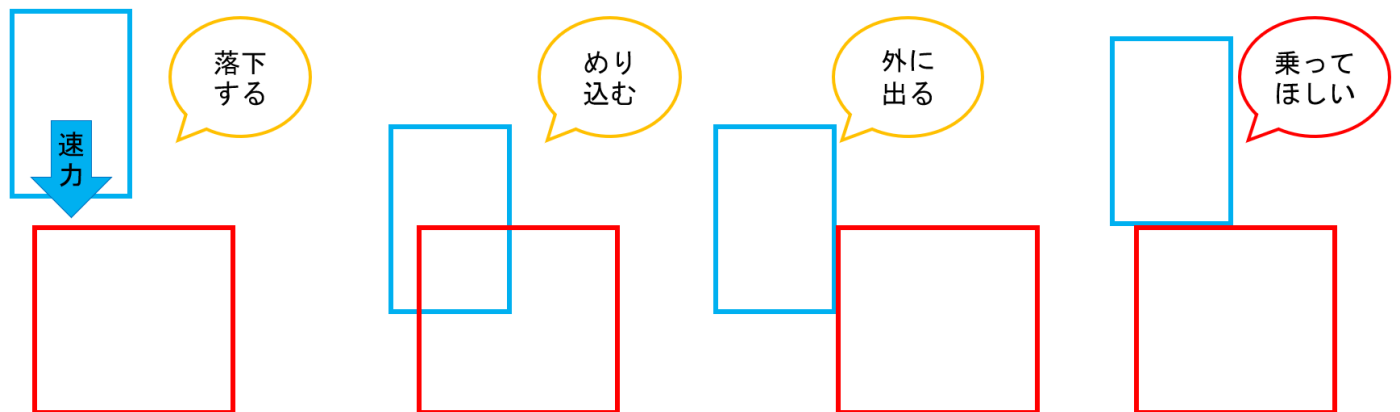
        DirectX::XMFLOAT3 outPosition;
        if (Collision::IntersectSphereVsSphere(
            enemyA->GetPosition(),
            enemyA->GetRadius(),
            enemyB->GetPosition(),
            enemyB->GetRadius(),
            outPosition))
        {
            enemyB->SetPosition(outPosition);
        }
        if (Collision::IntersectCylinderVsCylinder(
            
        ))
        {
            enemyB->SetPosition(outPosition);
        }
    }
}
```

実装が出来たら実行確認をしましょう。  
見た目通り衝突処理ができていれば OK です。

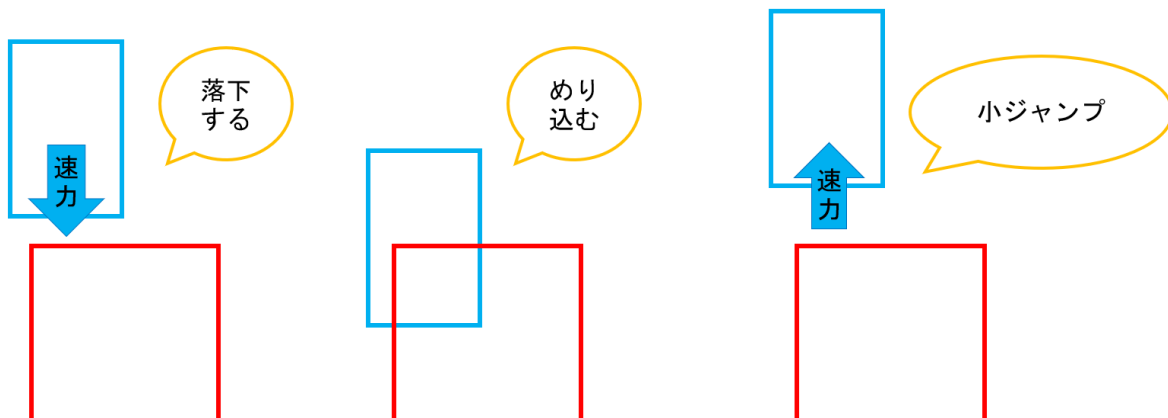
### ○踏みつけ処理

今回の簡易的な円柱の衝突判定では大きいオブジェクトにめり込んだ場合に一瞬で外側に押し出されてワープして見える問題があります。

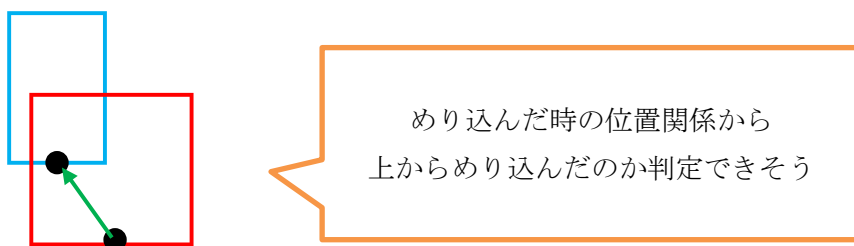
## ゲームプログラミングⅢ



本来は円柱の上に円柱が乗ってほしいのですが、計算がかなり難しいので実装できません。今回はこの問題をスーパーマリオでよくある踏んづけ処理で誤魔化します。



めり込んだ際に上から踏んづけたかを判定し、小ジャンプさせます。



Player.cpp

```
---省略---  
  
// 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();
```

## ゲームプログラミングⅢ

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

    // 衝突処理
    DirectX::XMFLOAT3 outPosition;
    if (Collision::IntersectCylinderVsCylinder(---省略---))
    {
        // 押し出し後の位置設定
        enemy->SetPosition(outPosition);
    }
}
```

プレイヤーと敵の衝突で  
敵とプレイヤーのベクトルで  
上から踏みつけたかを  
判定してみよう

実装したら実行確認をしてみましょう。  
敵を踏みつけると小ジャンプできていれば OK です。

お疲れさまでした。