ゲームプログラミング改

○評価要件

☑指定方向へ弾丸を直進処理
☑指定方向へ向くように弾丸の姿勢制御
☑弾丸の寿命処理
☑弾丸の追尾処理

○概要

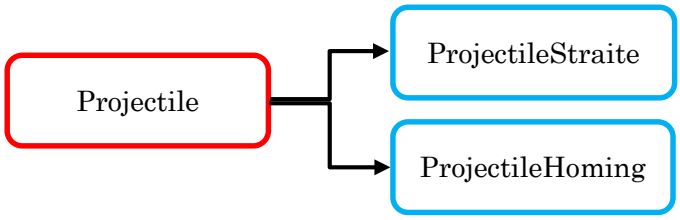
今回は弾丸処理を実装します。
弾丸処理とは銃の弾の挙動です。
一般的に弾丸を指定方向に発射して、オブジェクトに衝突する、一定時間経過するなどで弾丸は消滅します。

弾丸の種類も様々あり、今回は直進弾丸と追尾弾丸の２種類を実装します。

○弾丸処理

弾丸処理は、任意のタイミングで指定の方向に弾を飛ばします。
ボタンを押すとプレイヤーの腰の位置あたりからプレイヤーが向いている方向に弾を発射し、一定時間で弾が消滅するプログラムを実装していきましょう。

今回は以下のクラス設計でプログラムを実装していきます。



今まではキャラクタークラスが「位置」「回転」「スケール」をもとに「行列」を作成していました。
今回は「位置」「方向」「スケール」をもとに「行列」を作成する実装を学習します。

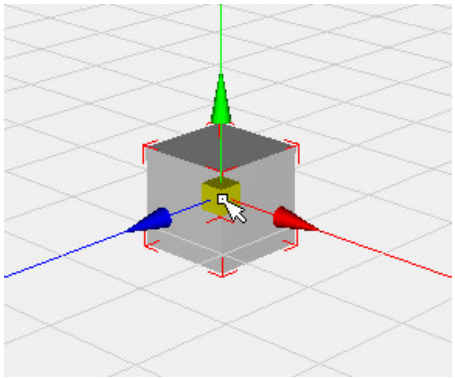
○方向指定による回転行列の作成

キャラクターではオイラー角による XYZ での回転値を元に行列を作成していました。
今回は指定方向が姿勢の前になる行列の作成方法を紹介します。

まず、行列のおさらいとして、行列は３つの軸と位置によってできています。
前と上と右方向のベクトルを算出し、位置がわかれば行列を作成できるわけです。

$$\begin{bmatrix} Xx & Xy & Xz & 0 \\ Yx & Yy & Yz & 0 \\ Zx & Zy & Zz & 0 \\ Px & Py & Pz & 1 \end{bmatrix}$$

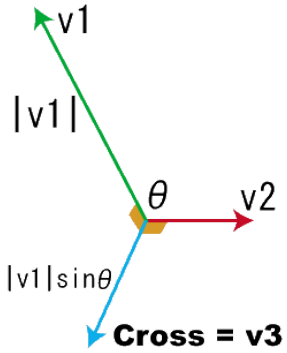
X 軸ベクトル
Y 軸ベクトル
Z 軸ベクトル
位置



今回は３D の「外積」を使用して３つの軸を算出する方法を学習します。

○３D の外積

２つのベクトルをもとに算出する２D の外積では「左右判定」ができるということを学習しました。
３D で２つのベクトルをもとに算出する外積の結果は「２つのベクトルに垂直な方向を向いたベクトル」になるという特性があります。
この特性を利用して３つの軸ベクトルを算出していきます。



外積のことを数学では「Cross」と呼びます。
ちなみに内積は「Dot」です。

○弾丸クラス

まずは基底となる弾丸クラスを実装しましょう。
Projectile.h を作成し、下記プログラムコードを記述しましょう。

Projectile.h

```
#pragma once

#include "Graphics/Shader.h"

// 弾丸
class Projectile
{
public:
    Projectile() {}
    virtual ~Projectile() {}

    // 更新処理
    virtual void Update(float elapsedTime) = 0;

    // 描画処理
    virtual void Render(ID3D11DeviceContext* dc, Shader* shader) = 0;

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

    // 位置取得
    const DirectX::XMFLOAT3& GetPosition() const { return position; }

    // 方向取得
    const DirectX::XMFLOAT3& GetDirection() const { return direction; }

    // スケール取得
    const DirectX::XMFLOAT3& GetScale() const { return scale; }

protected:
    // 行列更新処理
    void UpdateTransform();

protected:
    DirectX::XMFLOAT3      position = { 0, 0, 0 };
    DirectX::XMFLOAT3      direction = { 0, 0, 1 };
```

```
    DirectX::XMFLOAT3       scale = { 1, 1, 1 };
    DirectX::XMFLOAT4X4     transform = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 };
};
```

Projectile.cpp

```
#include "Projectile.h"

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

}

// 行列更新処理
void Projectile::UpdateTransform()
{
    // とりあえず、仮で回転は無視した行列を作成する。
    transform._11 = scale.x;
    transform._12 = 0.0f;
    transform._13 = 0.0f;
    transform._14 = 0.0f;
    transform._21 = 0.0f;
    transform._22 = scale.y;
    transform._23 = 0.0f;
    transform._24 = 0.0f;
    transform._31 = 0.0f;
    transform._32 = 0.0f;
    transform._33 = scale.z;
    transform._34 = 0.0f;
    transform._41 = position.x;
    transform._42 = position.y;
    transform._43 = position.z;
    transform._44 = 1.0f;
}
```

今は何も表示しない

この行列の計算は後で行う。
とりあえず、弾丸を確実に表示するために
回転行列の計算を今はしない。

次はエネミーの時と同じように弾丸を管理するマネージャークラスを作成しましょう。
ProjectileManager.cpp と ProjectileManager.h を作成しましょう。

ProjectileManager.h

```
#pragma once

#include <vector>
#include "Projectile.h"

// 弾丸マネージャー
class ProjectileManager
{
public:
    ProjectileManager();
    ~ProjectileManager();
```

エネミーマネージャーとは違い、
シングルトンにはしない。

```
    // 更新処理
    void Update(float elapsedTime);

    // 描画処理
    void Render(ID3D11DeviceContext* dc, Shader* shader);

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

    // 弾丸登録
    void Register(Projectile* projectile);

    // 弾丸全削除
    void Clear();

    // 弾丸数取得
    int GetProjectileCount() const { return static_cast<int>(projectiles.size()); }

    // 弾丸取得
    Projectile* GetProjectile(int index) { return projectiles.at(index); }

private:
    std::vector<Projectile*>      projectiles;
};
```

ProjectileManager.cpp

```
#include "ProjectileManager.h"

// コンストラクタ
ProjectileManager::ProjectileManager()
{
}

// デストラクタ
ProjectileManager::~ProjectileManager()
{
    Clear();
}

// 更新処理
void ProjectileManager::Update(float elapsedTime)
{
    // 更新処理
    
}

// 描画処理
void ProjectileManager::Render(ID3D11DeviceContext* context, Shader* shader)
{
    
```

```
}

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


}

// 弾丸登録
void ProjectileManager::Register(Projectile* projectile)
{

}

// 弾丸全削除
void ProjectileManager::Clear()
{


}
```

今回はプレイヤーが弾丸を発射するので、プレイヤーが発射する弾丸はプレイヤーが管理するようにします。
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---
#include "ProjectileManager.h"

// プレイヤー
class Player : public Character
{
    ---省略---
private:
    ---省略---
    ProjectileManager  projectileManager;
};
```

Player.cpp

```
---省略---

// 更新処理
void Player::Update(float elapsedTime)
```

```
{
    ---省略---

    // 速力更新処理
    ---省略---

    // 弾丸更新処理

    ---省略---
}

// 描画処理
void Player::Render(ID3D11DeviceContext* dc, Shader* shader)
{
    ---省略---

    // 弾丸描画処理

}

---省略---

// デバッグプリミティブ描画
void Player::DrawDebugPrimitive()
{
    ---省略---

    // 弾丸デバッグプリミティブ描画

}

---省略---
```

これでプレイヤーの中で弾丸を管理することができました。
次は本題の直進弾丸クラスを作成します。
まずはモデルの表示と指定方向に移動するプログラムを実装します。
ProjectileStraite.cpp と ProjectileStraite.h を作成し、下記プログラムコードを記述しましょう。

ProjectileStraite.h

```
#pragma once

#include "Graphics/Model.h"
#include "Projectile.h"

// 直進弾丸
class ProjectileStraite : public Projectile
{
public:
    ProjectileStraite();
    ~ProjectileStraite() override;

    // 更新処理
```

```
    void Update(float elapsedTime) override;

    // 描画処理
    void Render(ID3D11DeviceContext* dc, Shader* shader) override;

    // 発射
    void Launch(const DirectX::XMFLOAT3& direction, const DirectX::XMFLOAT3& position);

private:
    Model*      model = nullptr;
    float       speed = 10.0f;
};
```

ProjectileStraite.cpp

```
#include "ProjectileStraite.h"

// コンストラクタ
ProjectileStraite::ProjectileStraite()
{
    model = new Model("Data/Model/SpikeBall/SpikeBall.mdl");

    // 表示サイズを調整
    scale.x = scale.y = scale.z = 0.5f;
}

// デストラクタ
ProjectileStraite::~ProjectileStraite()
{
    delete model;
}

// 更新処理
void ProjectileStraite::Update(float elapsedTime)
{
    // 移動
    float speed = this->speed * elapsedTime;
    
    // オブジェクト行列を更新
    UpdateTransform();

    // モデル行列更新
    model->UpdateTransform(transform);
}

// 描画処理
void ProjectileStraite::Render(ID3D11DeviceContext* dc, Shader* shader)
{
    shader->Draw(dc, model);
}

// 発射
```

```
void ProjectileStraite::Launch(const DirectX::XMFLOAT3& direction,
                               const DirectX::XMFLOAT3& position)
{
    this->direction = direction;
    this->position = position;
}
```

弾丸の実装ができたのでプレイヤーが弾丸を発射できるようにしましょう。

Player.h

```
---省略--

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

    // 弾丸入力処理
    void InputProjectile();

    ---省略--
};
```

Player.cpp

```
---省略---
#include "ProjectileStraite.h"

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

    // ジャンプ入力処理
    ---省略---

    // 弾丸入力処理
    InputProjectile();

    ---省略---
}

---省略---

// 弾丸入力処理
void Player::InputProjectile()
{
    GamePad& gamePad = Input::Instance().GetGamePad();

    // 直進弾丸発射
    if (gamePad.GetButtonDown() & GamePad::BTN_X)
```

```
    {
        // 前方向
        DirectX::XMFLOAT3 dir;
        dir.x =
        dir.y =
        dir.z =
        // 発射位置（プレイヤーの腰あたり）
        DirectX::XMFLOAT3 pos;
        pos.x =
        pos.y =
        pos.z =
        // 発射
        ProjectileStraite* projectile = new ProjectileStraite();
        projectile->Launch(dir, pos);
        projectileManager.Register(projectile);
    }
}
```

実装出来たら実行確認をしてみましょう。
プレイヤーの腰元から弾丸が発射されていれば OK です。

○弾丸に寿命をつける

現状では弾丸を射出してから永久に破棄されないので、弾丸に寿命をつけましょう。
また、寿命が尽きると弾丸マネージャーから削除されるようにする必要もあります。

まずは弾丸マネージャーから指定の弾丸を削除するプログラムを追加しましょう。
std::vector で管理されている要素を削除するにはいくつか注意点があります。

ProjectileManager.h

```
---省略---

// 弾丸マネージャー
class ProjectileManager
{
public:
    ---省略---

    // 弾丸削除
    void Remove(Projectile* projectile);

private:
    ---省略---
    std::vector<Projectile*>       removes;
};
```

ProjectileManager.cpp

```
---省略---
```

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

    // 破棄処理
    // ※projectilesの範囲for文中でerase()すると不具合が発生してしまうため、
    //   更新処理が終わった後に破棄リストに積まれたオブジェクトを削除する。
    for (Projectile* projectile : removes)
    {
        // std::vectorから要素を削除する場合はイテレーターで削除しなければならない
        std::vector<Projectile*>::iterator it = std::find(projectiles.begin(), projectiles.end(),
                                                          projectile);

        if (it != projectiles.end())
        {
            projectiles.erase(it);
        }

        // 弾丸の破棄処理
        delete projectile;
    }
    // 破棄リストをクリア
    removes.clear();
}

---省略---

// 弾丸削除
void ProjectileManager::Remove(Projectile* projectile)
{
    // 破棄リストに追加
    removes.emplace_back(projectile);
}
```

std::vector で管理されている要素を削除するには erase()関数を使用する。

破棄リストのポインタからイテレーターを検索し、erase()関数に渡す。

直接、projectiles の要素を削除してしまうと範囲 for 文で不具合を起こすため、破棄リストに追加する

弾丸マネージャーから指定の弾丸を削除するプログラムを実装しました。
次は弾丸に寿命をつけ、自分自身を破棄するプログラムを実装します。

Projectile.h

```
---省略---

// 前方宣言
class ProjectileManager;

// 弾丸
class Projectile
{
public:
    ~~Projectile() {}~~
    Projectile(ProjectileManager* manager);

    ---省略---
```

相互インクルードしないように前方宣言する

登録されるマネージャーを保持するようにする

```
    // 破棄
    void Destroy();

    ---省略---

protected:
    ---省略---
    ProjectileManager*      manager = nullptr;
};
```

Projectile.cpp

```
---省略---
#include "ProjectileManager.h"

// コンストラクタ
Projectile::Projectile(ProjectileManager* manager)
    : manager(manager)
{
    manager->Register(this);
}

---省略---

// 破棄
void Projectile::Destroy()
{
    manager->Remove(this);
}
```

生成時にマネージャーに登録する

マネージャーから自分を削除する

ProjectileStraite.h

```
---省略---

// 直進弾丸
class ProjectileStraite : public Projectile
{
public:
    ~ProjectileStraite();
    ProjectileStraite(ProjectileManager* manager);

    ---省略---

private:
    ---省略---
    float       lifeTimer = 3.0f;
};
```

ProjectileStraite.cpp

```
---省略---

// コンストラクタ
```

```
ProjectileStraite::ProjectileStraite(ProjectileManager* manager)
    : Projectile(manager)
{
    ---省略---
}

---省略---

// 更新処理
void ProjectileStraite::Update(float elapsedTime)
{
    // 寿命処理
    [                              ]
    if ([                          ])
    {
        // 自分を削除
        Destroy();
    }

    ---省略---
}

---省略---
```

基底クラスのコンストラクタを呼び出す。

寿命が尽きたら
自分は破棄する。

Player.cpp

```
---省略---

// 弾丸入力処理
void Player::InputProjectile()
{
    GamePad& gamePad = Input::Instance().GetGamePad();

    // 直進弾丸発射
    if (gamePad.GetButtonDown() & GamePad::BTN_X)
    {
        // 前方向
        ---省略---
        // 発射位置（プレイヤーの腰あたり）
        ---省略---
        // 発射
        ProjectileStraite* projectile = new ProjectileStraite(&projectileManager);
        projectile->Launch(dir, pos);
        ~~projectileManager.Register(projectile);~~
    }
}
```

弾丸クラスのコンストラクタで
呼び出すようになったので削除。

ここまで実装出来たら実行確認をしてみましょう。
弾丸を発射し、一定時間後に弾丸が削除されていれば OK です。
これで寿命処理は完了しました。

○弾丸を発射方向に向かせる

次は弾丸を発射方向へ向かせるようにしましょう。
今は弾丸に対して回転制御を行っていません。
表示する弾の形状が球などのように向きを考慮する必要がない場合は回転制御をする必要がありませんが、ナイフを投げるなどの場合はナイフの先端が先行方向になった方が自然です。
今回は確認しやすいように弾丸のモデルを剣にして実装してみましょう。

ProjectileStraite.cpp

```
---省略---

// コンストラクタ
ProjectileStraite::ProjectileStraite(ProjectileManager* manager)
    : Projectile(manager)
{
    model = new Model("Data/Model/SpikeBall/SpikeBall.mdl");
    model = new Model("Data/Model/Sword/Sword.mdl");

    // 表示サイズを調整
    scale.x = scale.y = scale.z = 0.5f;
    scale.x = scale.y = scale.z = 3.0f;
}

---省略---
```

モデルを変更したら実行確認してみましょう。
弾丸の向きが常に進行方向ではなく、Z プラス方向を向いているはずです。
これから進行方向に向くように実装していきます。

冒頭で「外積」を使用して３つの軸を算出すると説明しました。
外積は２つのベクトルに対して垂直な方向のベクトルを算出できるという特性を持っています。
これは前方向ベクトルと上方向ベクトルが分かっていれば右方向ベクトルを算出できるということです。
他にも前方向ベクトルと右方向ベクトルが分かっていれば上方向ベクトルを算出できるということでもあります。
では、このヒントを参考にして発射方向ベクトルから行列を作成するプログラムを実装しましょう。

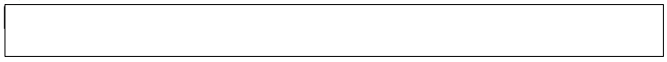
Projectile.cpp

```
---省略---

// 行列更新処理
void Projectile::UpdateTransform()
{
    // とりあえず、仮で回転は無視した行列を作成する。
    transform._11 = scale.x;
    transform._12 = 0.0f;
```

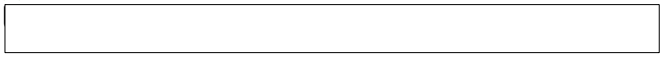
```
transform._13 = 0.0f;
transform._14 = 0.0f;
transform._21 = 0.0f;
transform._22 = scale.y;
transform._23 = 0.0f;
transform._24 = 0.0f;
transform._31 = 0.0f;
transform._32 = 0.0f;
transform._33 = scale.z;
transform._34 = 0.0f;
transform._41 = position.x;
transform._42 = position.y;
transform._43 = position.z;
transform._44 = 1.0f;

DirectX::XMVECTOR Front, Up, Right;

// 前ベクトルを算出
```

```
// 仮の上ベクトルを算出
```

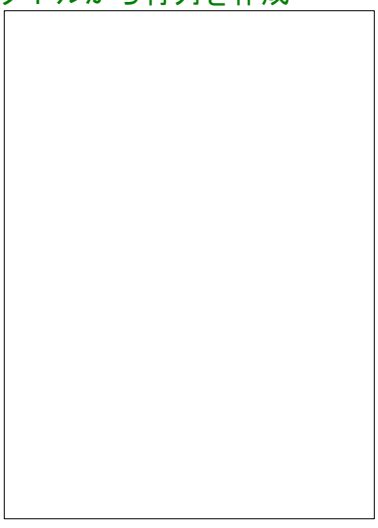
```
// 右ベクトルを算出
```



```
// 上ベクトルを算出
```



```
// 計算結果を取り出し
DirectX::XMFLOAT3 right, up, front;
DirectX::XMStoreFloat3(&right, Right);
DirectX::XMStoreFloat3(&up, Up);
DirectX::XMStoreFloat3(&front, Front);

// 算出した軸ベクトルから行列を作成
transform._11 =
transform._12 =
transform._13 =
transform._14 =
transform._21 =
transform._22 =
transform._23 =
transform._24 =
transform._31 =
transform._32 =
transform._33 =
transform._34 =
transform._41 =
transform._42 =
transform._43 =
transform._44 =
```



```
// 発射方向
this->direction = front;
```

```
}
---省略---
```

実装ができたら実行確認をしてみましょう。
弾丸が発射方向を向いて射出されていれば OK です。

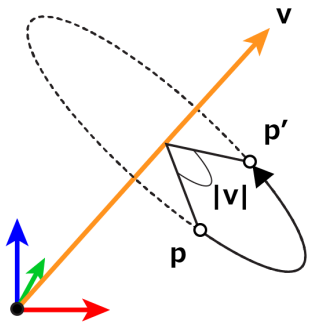
○追尾処理

追尾弾丸の基本的な考え方はキャラクターの移動と同じです。
キャラクターの移動の時は「内積」と「外積」を利用して進行方向ベクトルの方向へ Y 軸角度を調整するというやり方でした。
この時は２D（XZ 平面）での計算でしたが、今回は３D（XYZ 空間）で計算を行うことになります。

２D の時は２つのベクトルの外積を計算することで左右判定を行い、内積を計算することで角度の算出と回転速度の調整をすることで旋回処理を実現しました。

３D では同じく２つのベクトルの外積を計算することで「回転軸」を算出し、内積を計算することで角度の算出と回転速度の調整することで旋回処理を実現します。

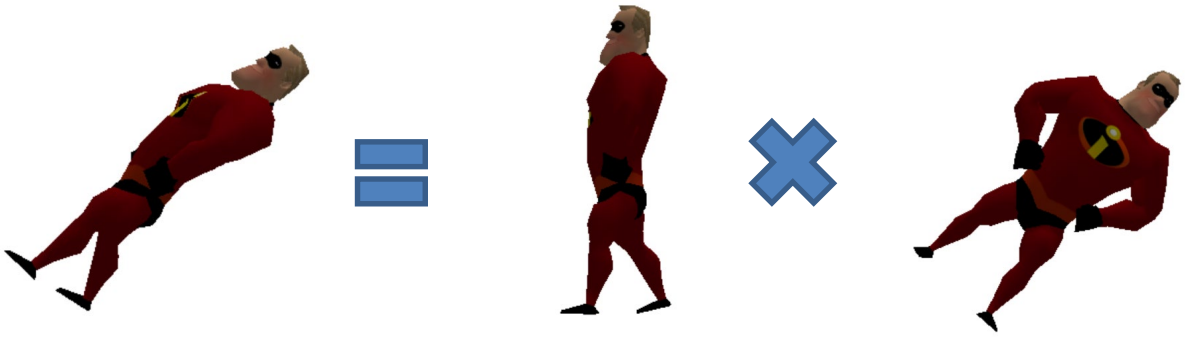
「回転軸」という新しい単語が出てきました。今までは回転軸といえば、X 軸、Y 軸、Z 軸といった固定の軸に対して回転させていましたが、任意の軸に回転させるということができます。



この図でいうと、２つのベクトルから外積で黄色の軸を算出し、内積で求めた回転角度で行列を回転させる。

この回転方法を「任意軸回転」と呼びます。
今まではキャラクターの回転角度（XYZ のオイラー角）を設定するだけだったので、あまり意識はしていませんでしたが、行列の回転方法について学習しましょう。

行列の回転は行列同士の掛け算によって回転後の行列を算出することができます。



回転後の姿勢　　回転姿勢　　回転前の姿勢

このように行列の掛け算をすることによって上図のような姿勢になることをイメージしましょう。
ちなみに掛け算をする順番が変わると結果が変わってしまうので注意してください。

○追尾弾丸クラス

「任意軸回転」と「行列の掛け算」を使うことによって３D の旋回処理を実装していきます。
ProjectileHoming.cpp と ProjectileHoming.h を作成し、下記プログラムコードを記述しましょう。

ProjectileHoming.h

```
#pragma once

#include "Graphics/Model.h"
#include "Projectile.h"

// 追尾弾丸
class ProjectileHoming : public Projectile
{
public:
    ProjectileHoming(ProjectileManager* manager);
    ~ProjectileHoming() override;

    // 更新処理
    void Update(float elapsedTime) override;

    // 描画処理
    void Render(ID3D11DeviceContext* dc, Shader* shader) override;

    // 発射
```

```
    void Launch(const DirectX::XMFLOAT3& direction,
                const DirectX::XMFLOAT3& position,
                const DirectX::XMFLOAT3& target);

private:
    Model*              model = nullptr;
    DirectX::XMFLOAT3   target = { 0, 0, 0 };
    float               moveSpeed = 10.0f;
    float               turnSpeed = DirectX::XMConvertToRadians(180);
    float               lifeTimer = 3.0f;
};
```

ここで設定するターゲットに
向かって追尾する

ProjectileHoming.cpp

```
#include "ProjectileHoming.h"

// コンストラクタ
ProjectileHoming::ProjectileHoming(ProjectileManager* manager)
    : Projectile(manager)
{
    model = new Model("Data/Model/Sword/Sword.mdl");

    // モデルが小さいのでスケーリング
    scale.x = scale.y = scale.z = 3.0f;
}

// デストラクタ
ProjectileHoming::~ProjectileHoming()
{
    delete model;
}

// 更新処理
void ProjectileHoming::Update(float elapsedTime)
{
    // 寿命処理
    {
    }

    // 移動
    {
        float moveSpeed = this->moveSpeed * elapsedTime;
    }

    // 旋回
    {
        float turnSpeed = this->turnSpeed * elapsedTime;
```

```
        // ターゲットまでのベクトルを算出
        DirectX::XMVECTOR Position =
        DirectX::XMVECTOR Target   =
        DirectX::XMVECTOR Vec      =

        // ゼロベクトルでないなら回転処理
        DirectX::XMVECTOR LengthSq = DirectX::XMVector3LengthSq(Vec);
        float lengthSq;
        DirectX::XMStoreFloat(&lengthSq, LengthSq);
        if (lengthSq > 0.00001f)
        {
            // ターゲットまでのベクトルを単位ベクトル化
            Vec =

            // 向いている方向ベクトルを算出
            DirectX::XMVECTOR Direction =

            // 前方方向ベクトルとターゲットまでのベクトルの内積(角度)を算出
            DirectX::XMVECTOR Dot =

            float dot;
            DirectX::XMStoreFloat(&dot, Dot);

            // ２つの単位ベクトルの角度が小さいほど
            // 1.0に近づくという性質を利用して回転速度を調整する


            // 回転角度があるなら回転処理をする
            if (                    )
            {
                // 回転軸を算出
                DirectX::XMVECTOR Axis =

                // 回転軸と回転量から回転行列を算出
                DirectX::XMMATRIX Rotation =

                // 現在の行列を回転させる
                DirectX::XMMATRIX Transform = DirectX::XMLoadFloat4x4(&transform);
                Transform =

                // 回転後の前方方向を取り出し、単位ベクトル化する

                DirectX::XMStoreFloat3(&direction, Direction);
            }
        }
    }

    // オブジェクト行列を更新
    UpdateTransform();

    // モデル行列更新
    model->UpdateTransform(transform);
}
```

```
// 描画処理
void ProjectileHoming::Render(ID3D11DeviceContext* dc, Shader* shader)
{
    shader->Draw(dc, model);
}

// 発射
void ProjectileHoming::Launch(const DirectX::XMFLOAT3& direction,
                              const DirectX::XMFLOAT3& position,
                              const DirectX::XMFLOAT3& target)
{
    this->direction = direction;
    this->position = position;
    this->target = target;

    UpdateTransform();
}
```

追尾弾丸クラスを実装出来たらプレイヤーが発射するプログラムを実装しましょう。

Player.cpp

```
---省略---
#include "ProjectileHoming.h"

---省略---

// 弾丸入力処理
void Player::InputProjectile()
{
    GamePad& gamePad = Input::Instance().GetGamePad();

    // 直進弾丸発射
    if (gamePad.GetButtonDown() & GamePad::BTN_X)
    {
        ---省略---
    }
    // 追尾弾丸発射
    if (gamePad.GetButtonDown() & GamePad::BTN_Y)
    {
        // 前方向
        DirectX::XMFLOAT3 dir;
        dir.x =
        dir.y =
        dir.z =

        // 発射位置（プレイヤーの腰あたり）
        DirectX::XMFLOAT3 pos;
        pos.x =
        pos.y =
        pos.z =

        // ターゲット（デフォルトではプレイヤーの前方）
```

```
        DirectX::XMFLOAT3 target;
        target.x =
        target.y =
        target.z =

        // 一番近くの敵をターゲットにする


        // 発射

    }
}
```

実装ができたら実行確認をしてみましょう。
敵に向かって弾丸が追尾していれば OK です。

○追尾解除

現状ではターゲットに向かって追尾しながら移動しますが、ターゲットを通り過ぎても旋回し続け、ターゲットのまわりをグルグル回ってしまう現象がおきてしまいます。
ターゲットまでの距離が一定以下になった場合に追尾を止める、または一定時間で追尾をやめるなどして自然な弾丸に自分で改造しましょう。

お疲れさまでした。