3D アクションゲームプログラミング

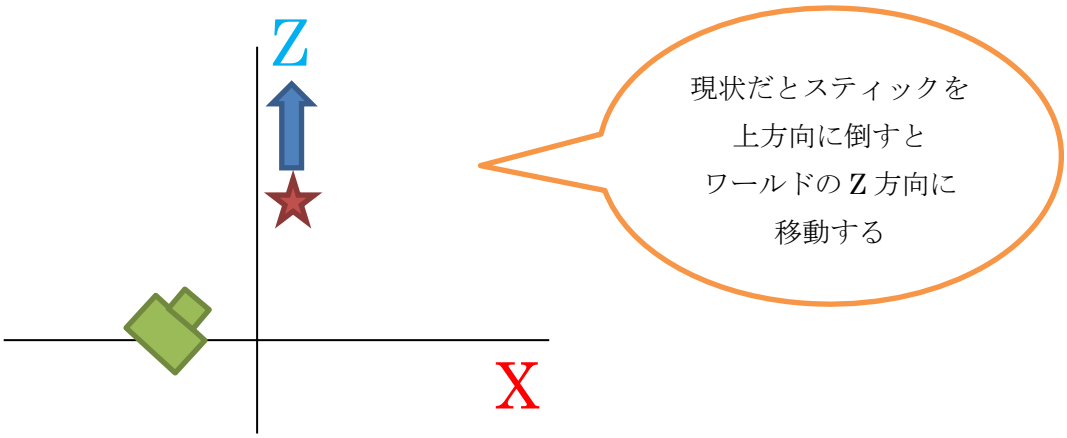
○評価要件

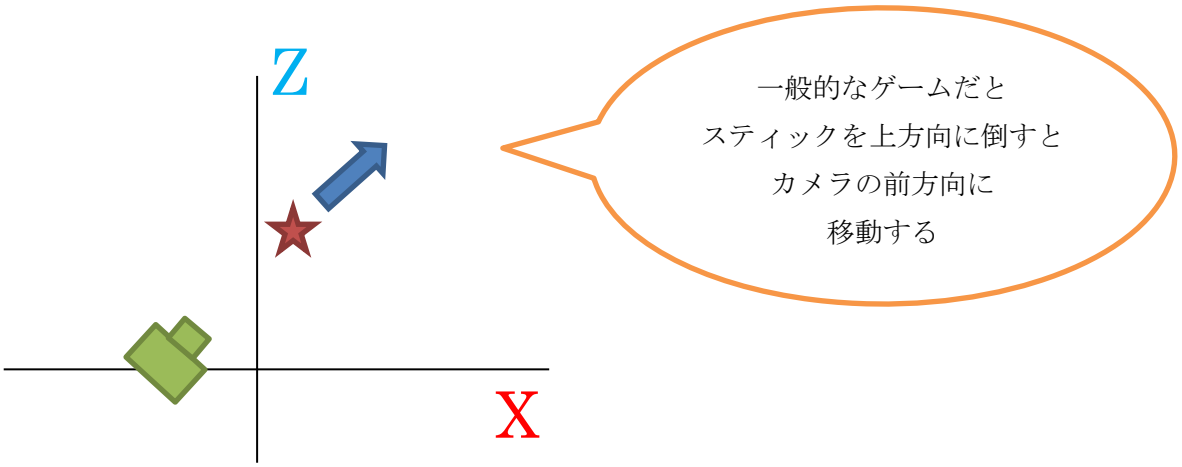
☑カメラ視点から入力値に対する進行ベクトルを取り出して移動
☑キャラクターが進行方向を向くように回転

○概要

今回はキャラクターの移動について学習しましょう。
現状では左スティックを入力すると世界の X 方向、Z 方向に移動します。

一般的な３D ゲームだと左スティックを上に倒すとカメラから見て前方向に移動します。
カメラの向きに対応した移動を実装しましょう。



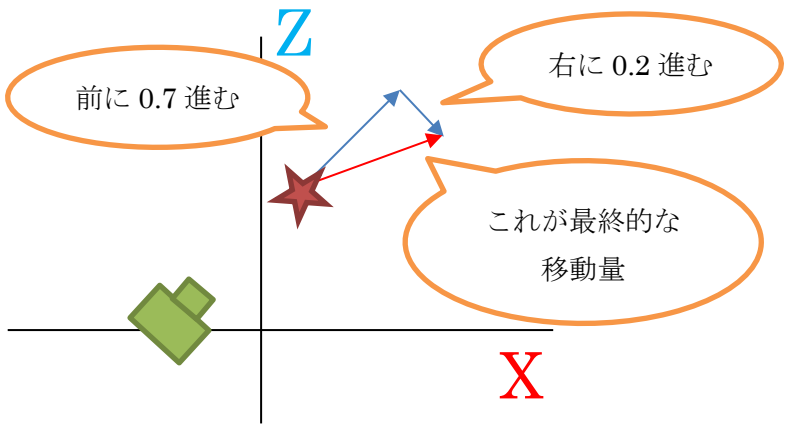


この実装をするために必要な情報を考えていきます。

○実装方法

この処理をするにはカメラの方向とスティックの入力値が必要になります。
スティックの上下入力値はカメラの前方向に対応し、
スティックの左右入力値はカメラの右方向に対応するようにできれば実装できます。

仮にスティックの入力値を（0.2, 0.7）と仮定して考えてみましょう。



図の赤い矢印（ベクトル）を求めることができれば良いわけです。
この赤い矢印は「ベクトルの合成」をすることで算出できます。
「ベクトルの合成」は単純に２つのベクトルを足し算するだけです。

算出した移動ベクトルをキャラクターの位置に加算するとうまくいきそうです。

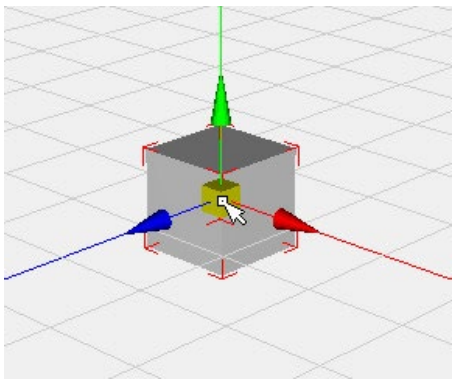
○カメラ方向

このキャラクター移動にはまずカメラの方向が必要ということがわかりました。
カメラクラスを拡張してカメラの方向を取り出せるようにしましょう。

前回、行列は以下のように３つの軸と位置で出来ていると学習しました。
この場合だと今回必要な方向を取り出すのは簡単です。

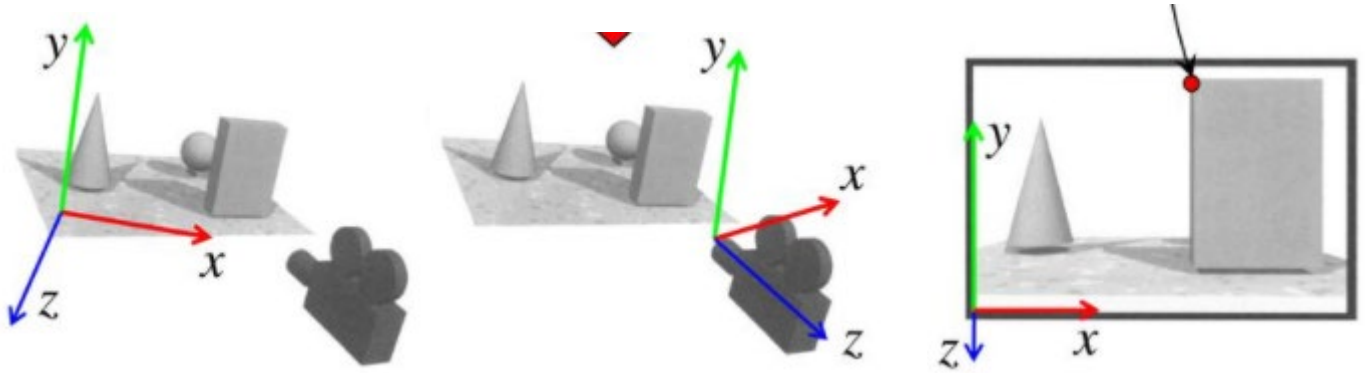
$$\begin{bmatrix} Xx & Xy & Xz & 0 \\ Yx & Yy & Yz & 0 \\ Zx & Zy & Zz & 0 \\ Px & Py & Pz & 1 \end{bmatrix}$$

X 軸ベクトル
Y 軸ベクトル
Z 軸ベクトル
位置



しかし、カメラのビュー行列は特殊な行列で、そのまま直接は取り出せないのです。
ビュー行列はプロジェクション行列と組み合わせることでスクリーン画面に世界を表示するという役割をもっています。

カメラが見た世界をスクリーンに表示するということは、カメラが世界の中心という考え方に置き換えることができます。
通常の行列は世界を中心に考えたもので、これをワールド行列と呼びます。
ビュー行列はカメラを中心に考えたものなので、ビュー行列のままでは世界を中心と考えた前方向を取り出すことはできないのです。



ワールド空間　　ビュー空間　　スクリーン空間

しかし安心してください。ビュー行列からワールド行列へ変換する方法があります。
ビュー行列を「逆行列化」することでワールド行列へ変換できます。

○逆行列

逆行列についてのゲームプログラミングでのイメージを説明します。
行列とは姿勢を表します。
今、皆さんが知っている知識では世界の中心から見て、どこの位置にいて、どっちを向いているかを表しています。これがワールド行列です。

このワールド行列を逆行列化することで自分自身を世界の中心として考えることができるのです。
実はビュー行列はカメラのワールド行列を逆行列化したものだったのです。

そして逆行列化した行列に対して、さらに逆行列化することで元の行列に戻すことができます。
つまり、ビュー行列を逆行列化することでワールド行列に戻すことができるのです。

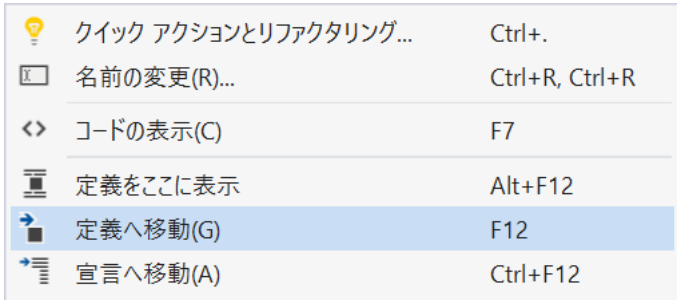
逆行列についての考え方は慣れや感覚が必要なので、今完璧に理解する必要はありません。
難しいことなので、ひとまずビュー行列を逆行列化するとワールド行列になると覚えておきましょう。

○DirectXMath での行列の確認

XMFLOAT4X4 構造体を確認してみよう。

右クリックして
「定義に移動」してみよう

```
DirectX::XMFLOAT4X4 transform =
{
    1, 0, 0, 0, // X軸ベクトル
    0, 1, 0, 0, // Y軸ベクトル
    0, 0, 1, 0, // Z軸ベクトル
    0, 0, 0, 1  // 位置
};
```



DirectXMath.h

union（共用体）なので
transform._11 または
transform.m[0][0] のようにして
メンバ変数にアクセスする。

```
struct XMFLOAT4X4
{
    union
    {
        struct
        {
            float _11, _12, _13, _14; // X軸ベクトル
            float _21, _22, _23, _24; // Y軸ベクトル
            float _31, _32, _33, _34; // Z軸ベクトル
            float _41, _42, _43, _44; // 位置
        };
        float m[4][4];
            :
          省略
    }
};
```

○カメラ方向の取り出し

カメラクラスを拡張してカメラ方向を取り出せるようにしましょう。
ついでにカメラの視点位置と、注視点位置も取得できるようにしておきます。
Camera.cpp と Camera.h を開き、下記プログラムコードを追記しましょう。

Camera.h

```
#pragma once

#include <DirectXMath.h>

// カメラ
class Camera
{
public:
    ---省略---
    // 視点取得
```

```
    const DirectX::XMFLOAT3& GetEye() const { return eye; }

    // 注視点取得
    const DirectX::XMFLOAT3& GetFocus() const { return focus; }

    // 上方向取得
    const DirectX::XMFLOAT3& GetUp() const { return up; }

    // 前方向取得
    const DirectX::XMFLOAT3& GetFront() const { return front; }

    // 右方向取得
    const DirectX::XMFLOAT3& GetRight() const { return right; }

private:
    ---省略---
    DirectX::XMFLOAT3      eye;
    DirectX::XMFLOAT3      focus;

    DirectX::XMFLOAT3      up;
    DirectX::XMFLOAT3      front;
    DirectX::XMFLOAT3      right;
};
```

Camera.cpp

```
#include "Camera.h"

// 指定方向を向く
void Camera::SetLookAt(const DirectX::XMFLOAT3& eye, const DirectX::XMFLOAT3& focus,
                                                     const DirectX::XMFLOAT3& up)
{
    ---省略---
    // ビューを逆行列化し、ワールド行列に戻す
    DirectX::XMMATRIX World = DirectX::XMMatrixInverse(nullptr, View);
    DirectX::XMFLOAT4X4 world;
    DirectX::XMStoreFloat4x4(&world, World);

    // カメラの方向を取り出す
    this->right.x = world._11;
    this->right.y = world._12;
    this->right.z = world._13;

    this->up.x = world._21;
    this->up.y = world._22;
    this->up.z = world._23;

    this->front.x = world._31;
    this->front.y = world._32;
    this->front.z = world._33;

    // 視点、注視点を保存
    this->eye   = eye;
    this->focus = focus;
}
```

```
---省略---
```

カメラの方向を取り出すことができました。
次はカメラ方向とスティックの入力値から進行方向ベクトルを算出しましょう。

○進行方向ベクトルの算出

続いてプレイヤークラスでスティックの入力によって進行方向ベクトルを算出しましょう。
Player.h と Player.cpp を開き、下記プログラムコードを追記、修正しましょう。

Player.h

```
---省略---

// プレイヤー
class Player : public Character
{
    ---省略---
private:
    // スティック入力値から移動ベクトルを取得
    DirectX::XMFLOAT3 GetMoveVec() const;

private:
    ---省略---
    float       moveSpeed = 5.0f;
};
```

Player.cpp

```
---省略---
#include "Camera.h"

---省略---

// 更新処理
void Player::Update(float elapsedTime)
{
    // 進行ベクトル取得
    DirectX::XMFLOAT3 moveVec = GetMoveVec();

    // 移動処理
    float moveSpeed = this->moveSpeed * elapsedTime;
    position.x += moveVec.x * moveSpeed;
    position.z += moveVec.z * moveSpeed;

    // 入力情報を取得
    GamePad& gamePad = Input::Instance().GetGamePad();
    float ax = gamePad.GetAxisLX();
    float ay = gamePad.GetAxisLY();

    // 移動操作
    float moveSpeed = 5.0f * elapsedTime;
    {
```

この部分は
もう必要ないので
削除する

```
        // 左スティックの入力情報をもとにXZ平面への移動処理
        position.x += ax * moveSpeed;
        position.z += ay * moveSpeed;
    }

    // 回転操作
    float rotateSpeed = DirectX::XMConvertToRadians(360) * elapsedTime;
    if (gamePad.GetButton() & GamePad::BTN_A)
    {
        // X軸回転操作
        angle.x += rotateSpeed;
    }
    if (gamePad.GetButton() & GamePad::BTN_B)
    {
        // Y軸回転操作
        angle.y += rotateSpeed;
    }
    if (gamePad.GetButton() & GamePad::BTN_X)
    {
        // Z軸回転操作
        angle.z += rotateSpeed;
    }

    ---省略---
}

---省略---

// スティック入力値から移動ベクトルを取得
DirectX::XMFLOAT3 Player::GetMoveVec() const
{
    // 入力情報を取得
    GamePad& gamePad = Input::Instance().GetGamePad();
    float ax = gamePad.GetAxisLX();
    float ay = gamePad.GetAxisLY();

    // カメラ方向とスティックの入力値によって進行方向を計算する
    Camera& camera = Camera::Instance();
    const DirectX::XMFLOAT3& cameraRight = camera.GetRight();
    const DirectX::XMFLOAT3& cameraFront = camera.GetFront();

    // 移動ベクトルはXZ平面に水平なベクトルになるようにする

    // カメラ右方向ベクトルをXZ単位ベクトルに変換
    float cameraRightX      = cameraRight.x;
    float cameraRightZ      = cameraRight.z;
    float cameraRightLength = sqrtf(cameraRightX * cameraRightX + cameraRightZ * cameraRightZ);
    if (cameraRightLength > 0.0f)
    {
        // 単位ベクトル化
        cameraRightX /= cameraRightLength;
        cameraRightZ /= cameraRightLength;
    }

    // カメラ前方向ベクトルをXZ単位ベクトルに変換
    float cameraFrontX      = cameraFront.x;
    float cameraFrontZ      = cameraFront.z;
```

```
    float cameraFrontLength = sqrtf(cameraFrontX * cameraFrontX + cameraFrontZ * cameraFrontZ);
    if (cameraFrontLength > 0.0f)
    {
        // 単位ベクトル化
        cameraFrontX /= cameraFrontLength;
        cameraFrontZ /= cameraFrontLength;
    }

    // スティックの水平入力値をカメラ右方向に反映し、
    // スティックの垂直入力値をカメラ前方向に反映し、
    // 進行ベクトルを計算する
    DirectX::XMFLOAT3 vec;
    vec.x = (cameraRightX * ax) + (cameraFrontX * ay);
    vec.z = (cameraRightZ * ax) + (cameraFrontZ * ay);
    // Y軸方向には移動しない
    vec.y = 0.0f;

    return vec;
}
```

実行確認をしてみましょう。
カメラを回転させながら左スティックを操作してみましょう。
カメラの向きに対応した移動になっていれば OK です。

○リファクタリング

移動処理を実装したばかりですが、処理を関数化しましょう。
Update()関数は１フレーム内で行うすべての処理が書かれるため膨大になりがちです。
大した量ではないですが、移動処理を関数化しておきましょう。

Player.h と Player.cpp に下記プログラムコードを追記、修正しましょう。

Player.h

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

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

    // 移動入力処理
    void InputMove(float elapsedTime);

    ---省略---
};
```

Player.cpp

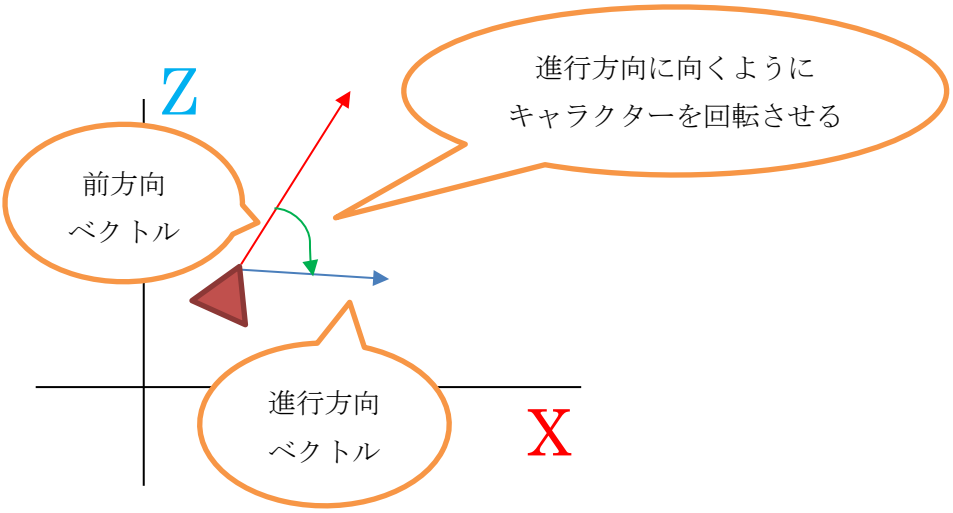
```
---省略---

// 更新処理
void Player::Update(float elapsedTime)
{
    // 進行ベクトル取得
    DirectX::XMFLOAT3 moveVec = GetMoveVec();

    // 移動処理
    float moveSpeed = this->moveSpeed * elapsedTime;
    position.x += moveVec.x * moveSpeed;
    position.z += moveVec.z * moveSpeed;

    // 移動入力処理
    InputMove(elapsedTime);

    ---省略---
}

// 移動処理
void Player::Move(float elapsedTime, float vx, float vz, float speed)
{
    speed *= elapsedTime;
    position.x += vx * speed;
    position.z += vz * speed;
}

// 移動入力処理
void Player::InputMove(float elapsedTime)
{
    // 進行ベクトル取得
    DirectX::XMFLOAT3 moveVec = GetMoveVec();

    // 移動処理
    Move(elapsedTime, moveVec.x, moveVec.z, moveSpeed);
}
```

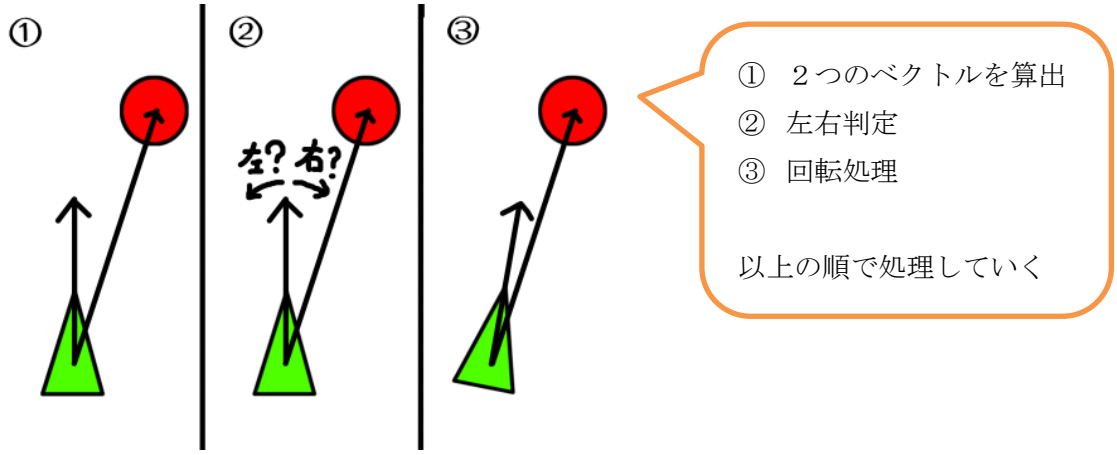
この部分は
もう必要ないので
削除する

○進行方向に合わせた旋回

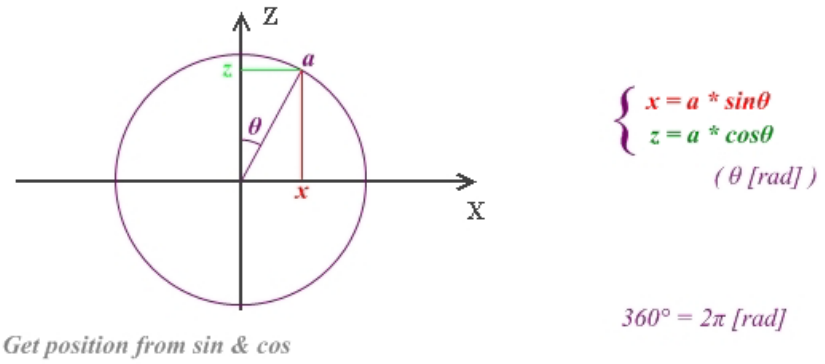
スティックの入力にあわせて自由に移動できるようになりました。
しかし、キャラクターの向きは常に固定されています。
一般的な３D ゲームでは進行方向に向くようになっています。

キャラクターの前方向ベクトルを進行方向ベクトルへ向くように回転させます。





進行方向ベクトルは既に算出しましたが、前方向ベクトルの算出も学習しましょう。
角度（今回では Y 軸角度）から sin と cos によってベクトルを算出できます。
プログラムを実装する際は下図を参考にベクトルを算出してみましょう。



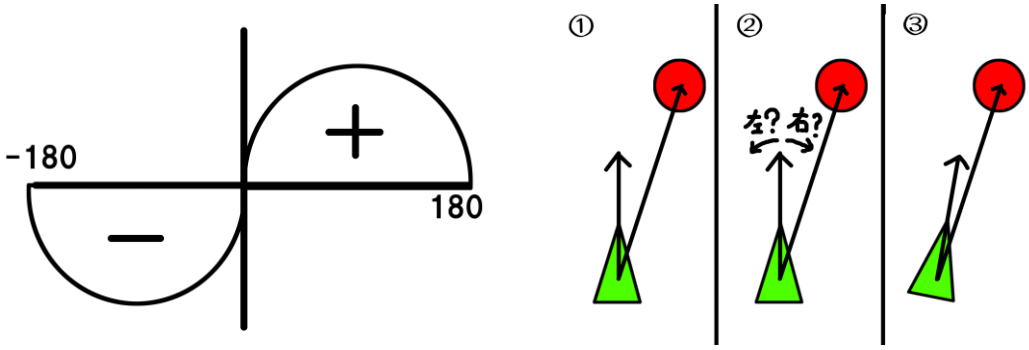
２つのベクトルが用意できたところで回転に必要なテクニックを紹介していきます。
今回の処理では２つのベクトルで算出できる「内積」と「外積」を使います。
３D ゲームですが、真上から見た XZ 平面の２D ベクトルで考えていきます。

○左右判定

前方向ベクトルを進行方向ベクトルに重なるように回転させるわけですが、まずは左右どちらに回転するべきかを「外積」をつかって判定するプログラムを作ります。

２つの２D ベクトルの外積の結果は正の場合は右、負の場合は左、という判定ができる特性をもっています。

理屈的には２つの単位ベクトルの外積を求めることで sin θ （-1.0～1.0）を求めることができ、
下図のような sin θ の波の結果になるため、左右判定ができます。
ピンとこない人はとりあえず、「2D の外積の結果で左右判定ができる」と覚えておきましょう。



外積の計算式は以下のようになります。

```
float cross = (vec1.z * vec2.x) - (vec1.x * vec2.z);
```

左右判定方法が分かったのでプログラムを実装しましょう。
Player.cpp と Player.h 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
{
private:
    ---省略---

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

private:
    ---省略---
    float       turnSpeed = DirectX::XMConvertToRadians(720);
};
```

Player.cpp

```
---省略---

// 旋回処理
void Player::Turn(float elapsedTime, float vx, float vz, float speed)
{
    speed *= elapsedTime;

    // 進行ベクトルがゼロベクトルの場合は処理する必要なし
    float length = sqrtf(vx * vx + vz * vz);
    if (length < 0.001f) return;

    // 進行ベクトルを単位ベクトル化
    vx /= length;
    vz /= length;

    // 自身の回転値から前方向を求める
    float frontX = sinf(angle.y);
    float frontZ = cosf(angle.y);

    // 左右判定を行うために２つの単位ベクトルの外積を計算する
    float cross = (frontZ * vx) - (frontX * vz);

    // 2Dの外積値が正の場合か負の場合によって左右判定が行える
    // 左右判定を行うことによって左右回転を選択する
    if (cross < 0.0f)
    {
        angle.y -= speed;
    }
    else
    {
        angle.y += speed;
    }
}

// 移動入力処理
```

```
void Player::InputMove(float elapsedTime)
{
    ---省略---

    // 旋回処理
    Turn(elapsedTime, moveVec.x, moveVec.z, turnSpeed);
}
```

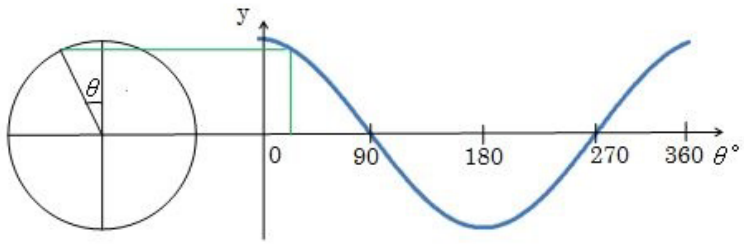
実装ができたら実行確認してみましょう。
キャラクターが移動した方向に向いてくれれば OK です。
ただし、この時点ではガタガタしています。
次はこの問題を解決していきます。

○回転量の調整

ガタガタしている問題がどういうことかというと、例えば毎フレーム５度ずつ回転するとします。
残りの回転角度が３度だった場合に５度回転すると２度余計に回ってしまいます。
次のフレームには余分に回りすぎてしまったので逆方向にまた５度回転します。
すると、また余計に３度回ってしまい、これが繰り返してガタガタしているわけです。

というわけで、残り角度が少なくなった場合に回転量を調整する工夫が必要になります。
２つの単位ベクトルの角度は「内積」を使って求めることできます。
正確には角度ではなく、cos θ （-1.0～1.0）を求めることができ、cos の性質を利用して回転量を調整するテクニックを実装します。

下図は cos のグラフです。



角度	値
0	1.0
90	0.0
180	-1.0
270 (-90)	0.0

角度が 0 に近いほど値が 1.0 に近づき、角度が大きいほど-1.0 に近づくという特性をもっています。
この特性を活かして角度が狭くなればなるほど 0.0 に近づくプログラムを作ります。
以下のプログラムを組むことによって内積値（-1.0～1.0）を（0.0～2.0）に補正します。

```
float dot = (vec1.x * vec2.x) + (vec1.z * vec2.z); // 内積

float rot = 1.0 - dot; // 補正値
```

ここで求めた「rot」という値は角度が小さくなればなるほど 0.0 に近づくようになりました。
このプログラムを利用して滑らかに旋回するようにしましょう。

Player.cpp

```
---省略---

// 旋回処理
void Player::Turn(float vx, float vz, float speed)
{
    ---省略---

    // 回転角を求めるため、２つの単位ベクトルの内積を計算する
    float dot = (frontX * vx) + (frontZ * vz);

    // 内積値は-1.0～1.0で表現されており、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
    float rot = 1.0f - dot;
    if (rot > speed) rot = speed;

    // 左右判定を行うために２つの単位ベクトルの外積を計算する
    float cross = (frontZ * vx) - (frontX * vz);

    // 2Dの外積値が正の場合か負の場合によって左右判定が行える
    // 左右判定を行うことによって左右回転を選択する
    if (cross < 0.0f)
    {
        angle.y -= speed;
        angle.y -= rot;
    }
    else
    {
        angle.y += speed;
        angle.y += rot;
    }
}
```

実装できたら実行確認をしてみましょう。
ガタガタせず、滑らかに旋回できていれば OK です。

内積と外積はゲームプログラミングをしていく上で、頻繁に利用するので必ずマスターしましょう。
お疲れさまでした。