

## Chapter 1

キーボードのAが押されたらプレイヤーの移動速度を倍にしてみよう。

まず、キーボードのAが押されたということを記録できるようにする必要があります。

Packman のプロジェクトでキーボードの入力を記録しているプログラムは下記の二のファイルです。

ソースファイル

.¥Packman¥tkEngine¥Input¥tkInput¥tkKeyInput.cpp

ヘッダーファイル

.¥Packman¥tkEngine¥Input¥tkInput¥tkKeyInput.h

ではヘッダーファイルを下記のように編集してください。太字になっている部分が変更点です。

```

/*!
 * @brief      キー入力。
 */

#ifndef _TKKEYINPUT_H_
#define _TKKEYINPUT_H_

namespace tkEngine{
    class CKeyInput{
        enum EnKey{
            enKeyUp,
            enKeyDown,
            enKeyRight,
            enKeyLeft,
            enKeyA,
            enKeyNum,
        };
    public:
        /*!
         * @brief      コンストラクタ。
         */
        CKeyInput();
        /*!
         * @brief      デストラクタ。
         */
        ~CKeyInput();
        /*!
         * @brief      キー情報の更新。
         */
        void Update();
        /*!
         * @brief      上キーが押されている。
         */
        bool IsUpPress() const
        {
            return m_keyPressFlag[enKeyUp];
        }
        /*!
         * @brief      右キーが押されている。
         */
        bool IsRightPress() const
        {
            return m_keyPressFlag[enKeyRight];
        }
    };
}

```

```

        /*!
        * @brief      左キーが押されている。
        */
        bool IsLeftPress() const
        {
            return m_keyPressFlag[enKeyLeft];
        }
        /*!
        * @brief      下キーが押されている。
        */
        bool IsDownPress() const
        {
            return m_keyPressFlag[enKeyDown];
        }
        /*!
        * @brief      キーボードの A が押された。
        */
        bool IsAPress() const
        {
            return m_keyPressFlag[enKeyA];
        }
    private:
        bool    m_keyPressFlag[enKeyNum];
    };
}
#endif // _TKKEYINPUT_H_

```

続いてソースファイルを下記のように変更します。

```

/*!
 * @brief      キー入力
 */

#include "tkEngine/tkEnginePreCompile.h"
#include "tkEngine/Input/tkKeyInput.h"

namespace tkEngine{
    /*!
    * @brief      コンストラクタ。
    */
    CKeyInput::CKeyInput()
    {
        memset(m_keyPressFlag, 0, sizeof(m_keyPressFlag));
    }
    /*!
    * @brief      デストラクタ。
    */
    CKeyInput::~CKeyInput()
    {
    }
    /*!
    * @brief      キー情報の更新。
    */
    void CKeyInput::Update()
    {
        if (GetAsyncKeyState(VK_UP) & 0x8000) {
            m_keyPressFlag[enKeyUp] = true;
        }
        else {
            m_keyPressFlag[enKeyUp] = false;
        }
        if (GetAsyncKeyState(VK_DOWN) & 0x8000) {
            m_keyPressFlag[enKeyDown] = true;
        }
    }
}

```

```

        else {
            m_keyPressFlag[enKeyDown] = false;
        }
        if (GetAsyncKeyState(VK_RIGHT) & 0x8000) {
            m_keyPressFlag[enKeyRight] = true;
        }
        else {
            m_keyPressFlag[enKeyRight] = false;
        }
        if (GetAsyncKeyState(VK_LEFT) & 0x8000) {
            m_keyPressFlag[enKeyLeft] = true;
        }
        else {
            m_keyPressFlag[enKeyLeft] = false;
        }
        if ((GetAsyncKeyState('A') & 0x8000)
            | (GetAsyncKeyState('a') & 0x8000)) {
            m_keyPressFlag[enKeyA] = true;
        }
        else {
            m_keyPressFlag[enKeyA] = false;
        }
    }
}

```

これでキーボードの A が入力されると、m\_keyPressFlag[enKeyA]に true という値が記録されるようになりました。

では本当にキーボードの A が入力されたら true が設定されるか確認してみましょう。先ほどの書き換えた部分にコードを下記の黒字のコードを追加してみてください。

```

if ((GetAsyncKeyState('A') & 0x8000)
    | (GetAsyncKeyState('a') & 0x8000)) {
    m_keyPressFlag[enKeyA] = true;
    MessageBox(NULL, "A ボタンが押されたよ！", "成功", MB_OK);
}
else {
    m_keyPressFlag[enKeyA] = false;
}

```

A を押したらダイアログボックスがでましたか？

では、確認ができればメッセージボックスを表示するコードは削除してください。

では、正しくキーボードから入力を受け取ることができることが確認できたのでプレイヤーの移動速度を倍にしてみましょう。プレイヤーの移動処理は下記のファイルに記述されています。

ソースファイル

Packman¥Packman¥game¥Player¥ CPlayer.cpp

CPlayer.cpp を下記のように書き換えてみてください。

```

/*!
 * @brief      プレイヤー
 */

#include "stdafx.h"
#include "Packman/game/Player/CPlayer.h"

```

```

#include "Packman/game/CGameManager.h"

/*!
 * @brief      Update が初めて呼ばれる直前に一度だけ呼ばれる処理。
 */
void CPlayer::Start()
{
}

/*!
 * @brief      Update 関数が実行される前に呼ばれる更新関数。
 */
void CPlayer::PreUpdate()
{
    Move();
}

/*!
 * @brief      更新処理。60fps なら 16 ミリ秒に一度。30fps なら 32 ミリ秒に一度呼ばれる。
 */
void CPlayer::Update()
{
    m_sphere.SetPosition(m_position);
    m_sphere.UpdateWorldMatrix();
    CGameManager& gm = CGameManager::GetInstance();
    CMatrix mMVP = gm.GetGameCamera().GetViewProjectionMatrix();
    const CMatrix& mWorld = m_sphere.GetWorldMatrix();
    m_wvpMatrix.Mul(mWorld, mMVP);
    m_idMapModel.SetWVPMatrix(m_wvpMatrix);
    IDMap().Entry(&m_idMapModel);
    m_shadowModel.SetWorldMatrix(mWorld);
    ShadowMap().Entry(&m_shadowModel);
}

/*!
 * @brief      移動処理。
 */
void CPlayer::Move()
{
    float moveSpeed = 0.02f; //移動速度。
    if (KeyInput().IsAPress()) {
        //キーボードの A 押されていたら速度を倍にする。
        moveSpeed *= 2.0f;
    }
    if (KeyInput().IsUpPress()) {
        m_position.z += moveSpeed;
    }
    if (KeyInput().IsDownPress()) {
        m_position.z -= moveSpeed;
    }
    if (KeyInput().IsRightPress()) {
        m_position.x += moveSpeed;
    }
    if (KeyInput().IsLeftPress()) {
        m_position.x -= moveSpeed;
    }
}

/*!
 * @brief      描画処理。60fps なら 16 ミリ秒に一度。30fps なら 32 ミリ秒に一度呼ばれる。
 */
void CPlayer::Render(tkEngine::CRenderContext& renderContext)
{
    CGameManager& gm = CGameManager::GetInstance();
    m_sphere.RenderLightWVP(
        renderContext,
        m_wvpMatrix,
        gm.GetFoodLight(),
        false,
        true
    );
}

/*!
 * @brief      構築。
 */

```

```

    /*必ず先に CreateShape を一度コールしておく必要がある。
    */
    void CPlayer::Build( const CVector3& pos )
    {
        m_sphere.Create(0.08f, 10, 0xffff0000, true );
        m_idMapModel.Create(m_sphere.GetPrimitive());
        m_shadowModel.Create(m_sphere.GetPrimitive());
        m_position = pos;
    }

```

## Chapter 2

ジャンプできるようにしてみよう。

Chapter1 のプログラムを改造して、キーボードの A が入力されたらプレイヤーがジャンプするようにしてみましょう。今回編集するプログラムは下記のファイルになります。

ソースファイル

Packman¥Packman¥game¥Player¥ CPlayer.cpp

ヘッダーファイル

Packman¥Packman¥game¥Player¥ CPlayer.h

CPlayer.h を下記のように編集してください。

```

/*!
 * @brief プレイヤー
 */
#ifndef _CPLAYER_H_
#define _CPLAYER_H_

#include "tkEngine/shape/tkSphereShape.h"

class CPlayer : public tkEngine::IGameObject{
public:
    CPlayer() :
        m_position(CVector3::Zero)
    {
    }
    ~CPlayer(){}
    /*!
     * @brief          Update が初めて呼ばれる直前に一度だけ呼ばれる処理。
     */
    void Start() override final;
    /*!
     * @brief Update 関数が実行される前に呼ばれる更新関数。
     */
    void PreUpdate() override final;
    /*!
     * @brief          更新処理。60fps なら 16 ミリ秒に一度。30fps なら 32 ミリ秒に一度呼ばれる。
     */
    void Update() override final;
    /*!
     * @brief          描画処理。60fps なら 16 ミリ秒に一度。30fps なら 32 ミリ秒に一度呼ばれる。
     */
    void Render(tkEngine::CRenderContext& renderContext) override final;
    /*!
     * @brief          構築。
     *必ず先に CreateShape を一度コールしておく必要がある。
     */
    void Build( const CVector3& pos );
    /*!

```

```

    *@brief 移動処理。
    */
    void Move();
    /*!
    *@brief 座標を取得。
    */
    const CVector3& GetPosition() const
    {
        return m_position;
    }
private:
    tkEngine::CSphereShape    m_sphere;
    CMatrix                   m_wvpMatrix;    //<ワールドビュープロジェクション行列。
    tkEngine::CIDMapModel     m_idMapModel;
    CVector3                  m_position;
    tkEngine::CShadowModel    m_shadowModel; //<シャドウマップへの書き込み用のモデル。
    CVector3                   m_moveSpeed;    //<移動速度。
};

#endif

```

プレイヤーに `m_moveSpeed` という移動速度を覚えるためのメンバ変数が追加されました。

では、続いて `tkPlayer.cpp` の `Move` 関数を下記のように編集してください。

```

void CPlayer::Move()
{
    //XZ平面での移動速度。
    m_moveSpeed.x = 0.02f;
    m_moveSpeed.z = 0.02f;
    if (KeyInput().IsAPress()) {
        //キーボードのAが押されていたら速度を倍にする。
        m_moveSpeed.x *= 2.0f;
        m_moveSpeed.z *= 2.0f;
        m_moveSpeed.y = 0.1f;
    }
    //Y方向への移動速度。
    if (KeyInput().IsUpPress()) {
        m_position.z += m_moveSpeed.z;
    }
    if (KeyInput().IsDownPress()) {
        m_position.z -= m_moveSpeed.z;
    }
    if (KeyInput().IsRightPress()) {
        m_position.x += m_moveSpeed.x;
    }
    if (KeyInput().IsLeftPress()) {
        m_position.x -= m_moveSpeed.x;
    }
    m_position.y += m_moveSpeed.y;
    //重力とかは考えない。
    m_moveSpeed.y -= 0.01f;
}

```

これでプレイヤーは A を押すとジャンプするようになりました。ただし地面と衝突判定などを行っていないため、このプレイヤーは A を押さないと奈落の底に落下していきます。次の Chapter ではプレイヤーが落下しないようにプログラムを変更してみます。

### Tips

ゲーム制作において、プレイヤーの挙動はそれっぽく見えれば OK なのでまじめに物理計算を行う必要があるわけではありません。ですが先ほどのジャンプ処理を重力を考慮してプログラムを書いてみましたので、せっかくですのでご紹介します。変更点は

*CPlayer* の *Move* 関数のみです。

```
void CPlayer::Move()
{
    //Moveが呼ばれる感覚は16ミリ秒で固定で考える。
    static const float deltaTime = 1.0f / 60.0f;
    //速度の単位をm/sに変更する。
    m_moveSpeed.x = 1.f;
    m_moveSpeed.z = 1.f; //XZ平面での移動速度。
    if (KeyInput().IsAPress()) {
        //ジャンプ。
        //初速度を2m/sで与える。
        m_moveSpeed.y = 2.0f;
    }
    CVector3 add(0.0f, 0.0f, 0.0f);
    add = m_moveSpeed;
    add.Scale(deltaTime);
    if (KeyInput().IsUpPress()) {
        m_position.z += add.z;
    }
    if (KeyInput().IsDownPress()) {
        m_position.z -= add.z;
    }
    if (KeyInput().IsRightPress()) {
        m_position.x += add.x;
    }
    if (KeyInput().IsLeftPress()) {
        m_position.x -= add.x;
    }
    m_position.y += add.y;
    //速度に重力加速度の影響を与える。
    //重力加速度 9.8m/s^2
    static const CVector3 gravity(0.0f, -9.8f, 0.0f);
    CVector3 addVelocity = gravity;
    addVelocity.Scale(deltaTime);
    m_moveSpeed.y += addVelocity.y;
}
```

変更を加えたプログラムを下記のパスに上げました。 *Debug* モードで実行すると処理が遅いのもっさりした挙動になるので、 *Release* モードで確認するといいいでしょう。

(影も壁に落ちるように改良してます・・・)

¥¥mmnas01¥student¥GC2016¥02\_授業¥ゲーム PG 1 ¥Lesson02¥Packman\_JumpGravity

## Chapter 3

地面に立てるようにしてみよう。

Chapter2 でパックマンがジャンプできるようになりましたが、A ボタンを押さないとパックマンは地面を突き抜けて自由落下していたはずですが。ではこれを地面に立てるように改造してみましょう。地面の位置は Y 座標で 0 の位置にあるので、パックマンの Y 座標が 0 より小さくなれば落下を行わないようにすれば地面に立てるはずです。ではプログラムを見ていきましょう。

今回編集するソースは下記のファイルになります。

ソースファイル

Packman¥Packman¥game¥Player¥ CPlayer.cpp

Packmap¥Packman¥game¥CCamera.cpp

```

/*
 * @brief 移動処理。
 */
void CPlayer::Move()
{
    //XZ 平面での移動速度。
    m_moveSpeed.x = 0.02f;
    m_moveSpeed.z = 0.02f;
    if (KeyInput().IsAPress()) {
        //キーボードの A が押されていたら速度を倍にする。
        m_moveSpeed.x *= 2.0f;
        m_moveSpeed.z *= 2.0f;
        m_moveSpeed.y = 0.1f;
    }
    //Y 方向への移動速度。
    if (KeyInput().IsUpPress()) {
        m_position.z += m_moveSpeed.z;
    }
    if (KeyInput().IsDownPress()) {
        m_position.z -= m_moveSpeed.z;
    }
    if (KeyInput().IsRightPress()) {
        m_position.x += m_moveSpeed.x;
    }
    if (KeyInput().IsLeftPress()) {
        m_position.x -= m_moveSpeed.x;
    }
    m_position.y += m_moveSpeed.y;
    //重力とかは考えない。
    m_moveSpeed.y -= 0.01f;
    if (m_position.y < 0.0f) {
        //座標が 0 以下になったので座標を補正。
        m_position.y = 0.0f;
    }
}

```

黒字の部分が今回変更した部分になります。

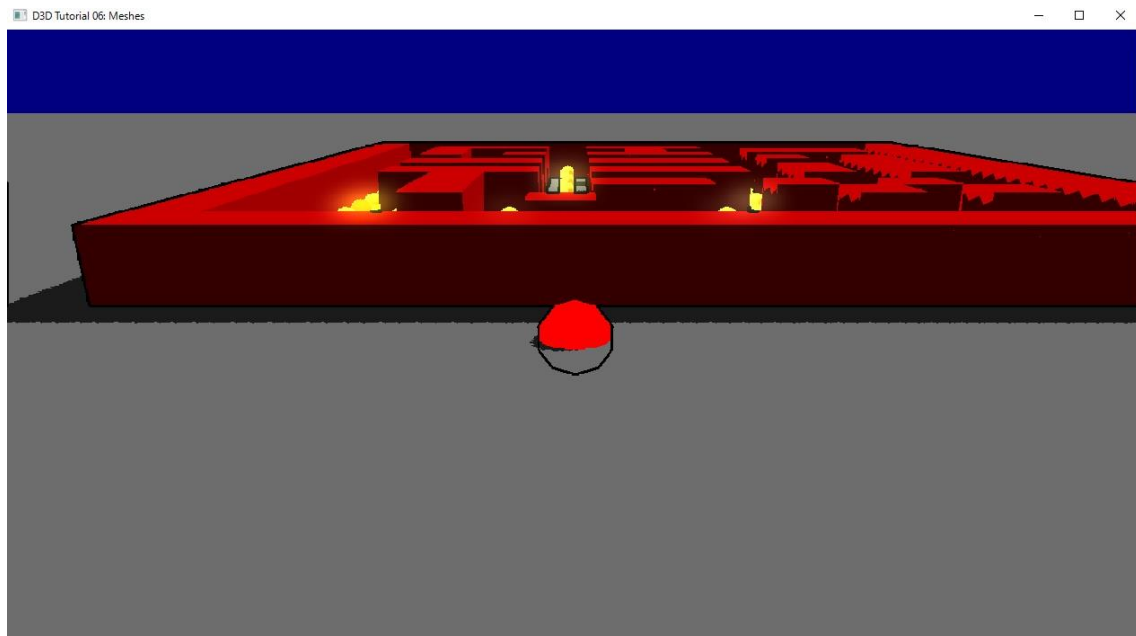
さて、この変更でパックマンは地面をすり抜けて落下しなくなりましたが、実はまだ問題が残っています。カメラのプログラムを下記のように書き換えてみてください。



## CCamera.cpp

```
void CGameCamera::Start()
{
    CVector3 cameraTarget;
    m_playerDist.Set(0.0f, 0.5f, -1.5f);
    cameraTarget = m_playerDist;
    cameraTarget.z = 0.0f;
    m_camera.SetPosition(m_playerDist);
    m_camera.SetTarget(CVector3::Zero);
    m_camera.SetUp(CVector3::Up);
    m_camera.SetFar(100000.0f);
    m_camera.SetNear(0.1f);
    m_camera.SetViewAngle(CMath::DegToRad(45.0f));
    m_camera.Update();
}
```

カメラが下記の図のようにプレイヤーの後方に移動したはずですが。



実はパックマンの座標は球体の中心を指しています。そのため Y 座標 0 で判定を行うとパックマンが地面にめり込んでしまいます。つまり判定する Y 座標をパックマンの半径分押し上げてやればめり込まなくなるはずですが。ではめり込まないようにプログラムを改良し

てみましょう。パックマンの半径は 0.08 とします。

# CPlayer.cpp

```

/*
 *@brief 移動処理。
 */
void CPlayer::Move()
{
    //XZ 平面での移動速度。
    m_moveSpeed.x = 0.02f;
    m_moveSpeed.z = 0.02f;
    if (KeyInput().IsAPress()) {
        //キーボードの A が押されていたら速度を倍にする。
        m_moveSpeed.x *= 2.0f;
        m_moveSpeed.z *= 2.0f;
        m_moveSpeed.y = 0.1f;
    }
    //Y 方向への移動速度。
    if (KeyInput().IsUpPress()) {
        m_position.z += m_moveSpeed.z;
    }
    if (KeyInput().IsDownPress()) {
        m_position.z -= m_moveSpeed.z;
    }
    if (KeyInput().IsRightPress()) {
        m_position.x += m_moveSpeed.x;
    }
    if (KeyInput().IsLeftPress()) {
        m_position.x -= m_moveSpeed.x;
    }
    m_position.y += m_moveSpeed.y;
    //重力とかは考えない。
    m_moveSpeed.y -= 0.01f;
    if (m_position.y < 0.08f) {
        //座標が半径以下になったので座標を補正。
        m_position.y = 0.08f;
    }
}

```

いかがでしょうか？パックマンが地面にめり込まなくなったはずです。

## Chapter 4

### 食べ物を食べられるようにしてみよう。

このチャプターではパックマンと食べ物の距離が一定値以下になったら、食べ物を食べたと判定して、食べ物を削除する処理を実装してみましょう。

#### 4.1 ゲームループ

まず、少し話がズレますが、どんなゲームのプログラムでもゲームが起動している間はループし続けるゲームループと言われるものがあります。このループは 60fps のゲームなら 16 ミリ秒に一度、30fps のゲームなら 33 ミリ秒に一度の周期でループしています。そしてどのゲームでも必ず、このループの中にゲームの状態の更新処理や描画処理が記述されています。

```
int main()
{
    //これがゲームループ。
    while(true){
        //ゲームの状態を更新する。
        //キャラの座標とか、HPとか色々。
        Update();
        //画面に絵を描く。
        Render();
        //画面のリフレッシュレートに同期させる。
        WaitVSync();
    }
}
```

非常に簡素なプログラムですが、どのゲームでも同様のコードが必ずあります。皆さんが今まで見てきた、パックマンのプログラムであれば、CFood::Update や CPlayer::Update が状態の更新処理。CFood::Render や CPlayer::Render が描画

処理ということになります。

#### 4.2 2点間の距離の計算

今回のお題の食べ物を削除する処理はゲームの状態を更新する処理になります。ですので、どこかの Update 関数にそのコードを記述すればいいことになります。

その手のコードをどこに記述するのかは、プログラマがやりやすいように決めていいのですが、今回は CFood::Update 関数の中にそのコードを記述していくことにします。

では話を戻して、まずプレイヤーと食べ物の距離を計算することができないと削除を行うことはできません。プレイヤーと食べ物の距離は下記の計算で求められます。

プレイヤーの座標を P、食べ物の座標を E としたとき、この 2 点間の距離 L は下記のように求められます。

$$V = P - E$$

$$L = \sqrt{V.x^2 + V.y^2 + V.z^2}$$

どこかで見たことがある計算式ではないでしょうか？これは中学校で習う三平方の定理を使用した計算式になります。数式がでてきたので嫌になる子もいるかもしれませんが、安心してください。今回の実習で使うプログラムには、簡単に距離を求めることができる関数を用意しています。

```
//プレイヤーの座標を取得。
CVector3 p = Player().GetPosition();
Cvector3 e = m_position;
CVector3 v;

v.x = p.x - e.x;
v.y = p.y - e.y;
v.z = p.z - e.z;
//これで L に長さが入る。
//関数の中で 3 平方の定理の計算をしている。
float L = v.Length();
```

ると食べ物を削除することができます。

いかがでしょうか？思っていたより簡単なコードではないでしょうか。あなたが記述した数式は引き算だけです。ゲーム会社ではベクトル計算の多くは関数として簡単に使用できるように用意されています。慣れてくるまでは難しく感じるかもしれませんが、使い始めるとそこまで難しくありません。

### 4.3 実際に消してみよう

距離の計算の仕方も分かったので、実際に食べ物を消してみましよう。私の作ったエンジンでは食べ物を消すためには下記のような少々特殊なコードを記述する必要があります。

```
CGameManager::Instance().DeleteGameObj  
ect(this);
```

では、食べ物を削除するコードを記述します。

```
CVector3 p = Player().GetPosition();
CVector3 e = m_position;
CVector3 v;
v.x = p.x - e.x;
v.y = p.y - e.y;
v.z = p.z - e.z;
float L = v.Length();
if (L < 0.08f) {
    CGameManager::Instance().DeleteGameO  
bject(this);
}
```

このコードを CFood::Update 関数に追加す

## Chapter 5

### パックマンと壁の当たり判定

ゲームをプレイしていて、キャラクターが壁を簡単にすり抜けてしまうゲームは基本的にゲームとしては欠陥品になってしまいます。そのため、世に出回っているゲームのほとんどは何かしらの当たり判定を実装しています。このチャプターでは 2D ゲームの頃から使われていた、簡単な当たり判定を実装してパックマンが壁にめり込まないようにしてみましょう。

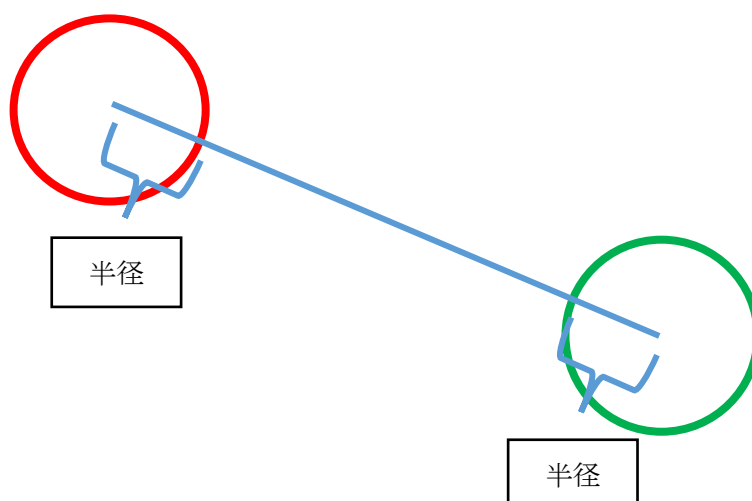
#### 5.1 色々な当たり判定

パックマンの当たり判定の説明を行う前に、3D ゲームで使われている、比較的簡単な当たり判定をいくつか紹介します。(簡単ではないかもしれませんが)

##### 5.1.1 球と球の当たり判定

ゲームで最も良く使われる当たり判定かもしれません。非常に高速に動作して、実装も容易なためいろいろなゲームで使われています。

実は、この当たり判定は皆さんすでに実装を行っています。なんのことか分かるでしょうか？皆さんに Chapter4 で、パックマンと食べ物の距離を調べてある一定距離以下なら食べ物を食べるというプログラムを組んでもらったと思います。実はあれが球と球の当たり判定になります。



球 A と球 B の中心座標を減算すると、球 A と球 B を結ぶベクトルが見つかります。そして、このベクトルの長さは 3 平方の定理を使用すれば求められます。そして、このベクトルの長さが球 A の半径と球 B の半径を足した大きさよりも小さければ衝突していると判定できます。Chapter4 で行ったことと全く同じです。

### 5.1.2 AABB と AABB の当たり判定

AABB というのは軸平行のバウンディングボックスと言われるものです。オブジェクトを内容する箱形状の当たり判定だと思ってください。AABB は一般的に下記のような構造体で表現されます。

```
struct Aabb{
    Vector3  vMax;    //箱の最大値。
    Vector3  vMin;    //箱の最小値。
};
```

そして、箱 A(aabbA)と箱 B(aabbB)の衝突判定は下記のような条件文で実装できます。

```
if(( aabbA.vMin.x <= aabbB.vMax.x )
    && ( aabbA.vMax.x >= aabbB.vMin.x )
    && ( aabbA.vMin.y <= aabbB.vMax.y )
    && ( aabbA.vMax.y >= aabbB.vMin.y )
    && ( aabbA.vMin.z <= aabbB.vMax.z )
    && ( aabbA.vMax.z >= aabbB.vMin.z )
){
    //衝突している。
}
```

### 5.1.3 配列を使用した当たり判定

ここまで見てきた当たり判定は、みなさんが個人制作で 3D ゲームの作成を行いだすと、恐らく一番お世話になる当たり判定になると思います。しかし、まだ少し難しいのではないかと思います。そこで今回は配列を使用した、古典的な、しかしゲームによっては今でも使える当たり判定を実装してもらいます。

パックマンのマップは CMapBuilder の sMap 配列を使用して、構築されています。この SMap 配列の値が 1 ならば壁なので移動しないという処理を記述してやれば、パックマンは壁にめり込まなくなるはずですが。ではこのトリックをお教えしましょう。

パックマンの床と壁のサイズは GRID\_SIZE になります。

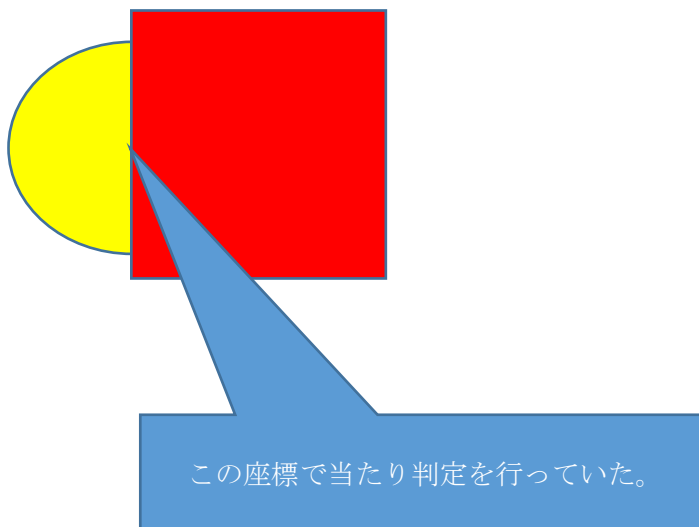
この GRID\_SIZE でパックマンの座標を除算してやると、パックマンがマップ上のどの場所にいるのかを判定することができます。

```
int x = (int)(m_position.x / GRID_SIZE);
int z = (int)(m_position.z / -GRID_SIZE);
if(sMap[z][x] == 1){
    //壁だよ!!!
}
```

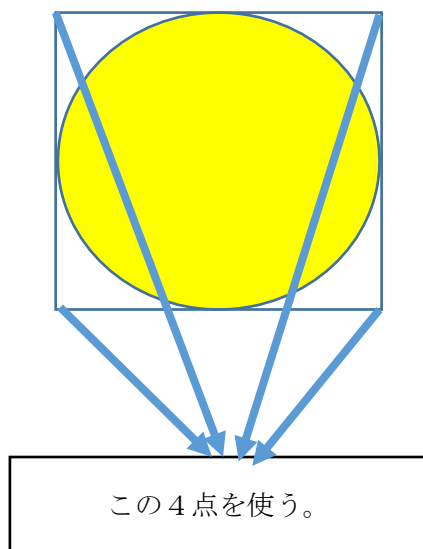
いかがでしょうか。非常に簡単なプログラムで壁の判定が行えました。では、ここまでの説明を参考にしてパックマンが壁をすり抜けないようにしてください。  
今回はパックマンが壁に半分めり込むと思いますが、それは考慮しなくて構いません。

## 5.2 壁にめり込まないようにしてみよう。

前節で行った当たり判定ですと、パックマンの体が半分めり込んでしまっていました。これはパックマンの中心座標を使って、壁との当たり判定を行っていたことが原因になります。



では、バウンディングボックスを使ってもう少しだけマシにしてみましょう。中心座標で判定を行っているのがめり込みの原因ですので、めり込まないようにパックマンを内包するバウンディングボックスの4隅の座標を使って当たり判定を行ってみようと思います。



プログラムで記述すると下記ようになります。

//プレイヤーを内包する四角形の4隅の当たり判定を行う。

bool isHitWall = false;

float radius = 0.075f;

//左上

int x = (int)((pos.x - radius) / GRID\_SIZE);

int z = (int)((pos.z - radius) / -GRID\_SIZE);

if (sMap[z][x] == 1) {

//壁

isHitWall = true;

}

//右上

x = (int)((pos.x + radius) / GRID\_SIZE);

z = (int)((pos.z - radius) / -GRID\_SIZE);

if (sMap[z][x] == 1) {

//壁

isHitWall = true;

}

//左下

x = (int)((pos.x - radius) / GRID\_SIZE);

z = (int)((pos.z + radius) / -GRID\_SIZE);

if (sMap[z][x] == 1) {

//壁

isHitWall = true;

}

//右下

x = (int)((pos.x + radius) / GRID\_SIZE);

z = (int)((pos.z + radius) / -GRID\_SIZE);

if (sMap[z][x] == 1) {

//壁

isHitWall = true;

}

}



## Chapter 6

### 経路探索

ゲーム AI を語る上で外せない要素の一つに経路探索と言われるものがあります。数年前まで、ゲーム AI の技術といえば経路探索というくらいホットな話題でした。フォトリアルな FPS ゲームで敵兵が壁をすり抜けてきたら興ざめすると思います。チャプター 6 では Lesson10 のプログラムを使いながら経路探索について見ていきます。

#### 6.1 プログラムの説明

経路探索のプログラムは下記のパスにあります。

- tkEngine/AI/tkPathFinding.h
- tkEngine/AI/tkPathFinding.cpp

採用されているアルゴリズムはダイクストラ法。

ネットワーク構造の経路探索データを受け取ることで経路探索が行えます。

実際に経路探索をゲームで使用しているプログラムは下記のパスになります。

- PackMan/game/Enemy/CEntity.h
- PackMan/game/Enemy/CEntity.cpp

#### 6.2 アルゴリズム

経路探索のアルゴリズムで代表的なものといえば下記の二つがあげられます。

- ダイクストラ法
- A\*アルゴリズム

A\*アルゴリズムはダイクストラ法の改良版となっており、多くのゲームにおいては A\*アルゴリズムの方が高速なアルゴリズムになります。ただし、ダイクストラ法の方が高速な場合もあって、ゴール地点が決まっている場合はダイクストラ法の方が高速になります。FPS などのような広大なマップでダイクストラ法を採用すると計算量が膨大になり、パフォーマンスが大きく低下するため、まずありえません。今回のサンプルプログラムではダイクストラ法を使用していますが、ダイクストラ法と A\*アルゴリズムとで実装難易度に差はありませんので、是非自分たちでチャレンジしてみてください。

#### 6.3 データ構造

経路探索を行うためには、通れる道を表すデータが必要になります。現在のゲームで主流となって使用されているデータはナビゲーションメッシュと言われるものです。ナビメッシュを作るまでもないようなゲームになると、ウェイポイントと言われるものを使っているものもあります。

ナビゲーションメッシュ、ウェイポイントどちらを採用しても基本的なデータ構造に大きな違いはありません。両者とも隣接するノードとのリンクをもったネットワークのようなデータ構造になっています。

プログラムでは下記のような構造体が使われます。

```
struct SNode{
    SNode*   linkNode[3]; //隣接ノード
    CVector3  position;    //ノードの座標。
};
```

#### 実習課題

Lesson10 のプログラムは敵が最初から最後までプレイヤーを追いかけてくるだけの挙動になっています。この敵の挙動をもう少しマシにして面白いゲームにしてください。仕様は自由です。あなたの好きなように面白いゲームを作ってみてください。

## Chapter 7

### ゲームで使われるデザインパターン

デザインパターンとは、先人たちが考えた優れた設計をたくさんの人が使いやすいようにカタログ化したものです。デザインパターンで最も有名なものは **GOF** デザインパターンと言われる **23** 種類のデザインパターンです。

デザインパターンとは、この **23** 種類だけではなく、各種分野によってさまざまなデザインパターンが存在します。今回はゲームで古くから使われていて、今現在 **Unity** や **UnrealEngine** などでも使われているデザインパターンを紹介します。今まで皆さんに見てもらってきたパックマンのプログラムでも使われています。

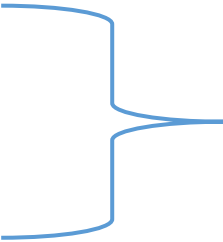
#### 7.1 ゲームループ

まずは 4.1 でも紹介した、ゲームループについて見ていきましょう。ゲームループとはゲームプログラミングを行う上で欠かすことができないパターンになります。ほぼすべてのゲームで採用されていて、ゲーム以外のプログラムではあまり使われません。ここまで皆さん学んできたように、画面上に表示されるゲームキャラクタというのは、3Dゲームであれば3D空間上でどこにいるのかというベクトル型の座標のデータを保持しています。そして、3D空間上でキャラクターを動かすためには、この座標を移動させる必要がありました。例えば、コントローラーで右方向が入力されたらX方向に10ずつ移動する場合、下記ののようなコードを記述してきたはずです。

```
if(KeyInput().IsPressA() == true){
    position.x += 10;
}
```

では、上のプログラムをどこで、何回実行すればいいのでしょうか？もしこのコードが一度しか実行されないのであれば、キャラクターはX方向に10移動した後はピクリともしなくなるでしょう。つまりこのコードはゲーム起動中ずっと実行される必要があるのです。そのため、ゲームプログラムにはゲーム起動中は終了することのないゲームループと言われるものが実装されています。

```
while(true){
    if(KeyInput().IsPressA() == true){
        position.x += 10;
    }
}
```



ゲームループ

さて、これでもまだ問題が残ります。このゲームのループする速度が実行される環境に依存するということです。例えば、**core i7** を積んでいるコンピューターであれば、このループは1秒間に100万回実行されるかもしれません。しかし **core i3** を積んでいるコンピューターであれば、このループは1秒間に10万回しか実行されないかもしれません。このように、コンピューターのスペックによってゲームの実行速度が大きく変わってしまうと、まともにゲームを遊ぶことはできません。そのため、多くのゲームでは1秒間にループできる回数の上限を設定しています。60fps とか 30fps とか聞いたことあると思います。60fps で動作するゲームの場合は1秒間に60回ループします。30fps の場合は1秒間に30回ループします。

```
while(true){
    if(KeyInput().IsPressA() == true){
        position.x += 10;
    }
    //垂直同期待ち。ここでゲームループの処理が 1/60 秒経過するまで待機する。
    WaitVSync();
}
```

## 7.1.2 画面のリフレッシュレート

先ほどの1秒間にループする回数ですが、この回数は画面のリフレッシュレートに深くかかわっています。では画面のリフレッシュレートとは一体なんなのか見てみましょう。皆さんがいつも見ているパソコンのディスプレイや、テレビに表示されている画像は、リフレッシュレートが60のモニタであれば、1秒間に60回ほど画面が切り替わっているパラパラ漫画のようなものになっています。この画面を切り替える回数がリフレッシュレートと言われるもので、1秒間に30回画面を切り替えるもののリフレッシュレートは30になります。1秒間に120回切り替えるものであれば、リフレッシュレートは120になります。基本的なゲームループは一回の画面切り替えにかかる時間の倍数を垂直同期待ちの時間に設定していることが多いです。(このリフレッシュレートをわざと考慮してないゲームも存在します。その場合、ティアリングという現象が発生してしまうことになります。)

## 7.2 UpdateMethod

このデザインパターンは古くはナムコが開発したギャラクシアン、現在であれば Unity、UnrealEngine4 まで脈々と受け継がれてきているパターンです。そして、皆さんが実習で使っていたパックマンのプログラムでも使用しています。

このパターンはタスクシステムやゲームオブジェクトマネージャーなど色々な名前と呼ばれていますが、このパターンを短い文章で説明すると次のようになります。

「インスタンスを登録すると、以降自動で毎フレーム更新関数が呼び出される。」

パックマンのプログラムであれば、下記のようなコードが該当します。

```
CGameManager::Instance().NewGameObject<CEntity>(0);
```

このコードは CEntity のインスタンスを作成して、そのインスタンスをゲームオブジェクトマネージャーに登録しています。登録されたインスタンスは以降、毎フレーム Update 関数と Render 関数が自動で呼び出されます。では、このパターンのサンプルコードを見てみましょう。非常にシンプルなサンプルですが、C++を学んでいる最中の皆さんにとっては、UpdateMethod パターンよりもポリモーフィズム(多態性)を活用しているコードが興味深いかもしれません。

```
////////////////////////////////////
//ゲームオブジェクトのインターフェースクラス。
////////////////////////////////////
class IGameObject{
public:
    virtual void Update() = 0; //純粋仮想関数。
};
////////////////////////////////////
//IGameObject のインターフェースを継承した Enemy クラス。
////////////////////////////////////
class Enemy : public IGameObject{
public:
    void Update();
};
//Enemy の Update 関数の実装。
void Enemy::Update()
{
    std::cout << "Enemy だよ！";
}
////////////////////////////////////
//IGameObject のインターフェースを継承した Player クラス。
////////////////////////////////////
class Player : public IGameObject{
public:
    void Update();
}
//プレイヤーの Update 関数の実装。;
void Player::Update()
{
    std::cout << "プレイヤーだよ！";
}

////////////////////////////////////
// メイン処理
```

```
////////////////////////////////////  
const int MAX_GAME_OBJECT = 64;  
IGameObject* gameObjects[64];  
int main()  
{  
    int numEntryGameObject = 2;  
    gameObject[0] = new Enemy; //Enemy のインスタンスを生成。  
    gameObject[1] = new Player; //Player のインスタンスを生成。  
    while(true){ //ゲームループ。  
        for(int i = 0; i < numEntryGameObject; i++){  
            gameObject[i]->Update(); //これで Enemy と Player の Update が実行される。  
                                     //これがポリモーフィズム！！  
        }  
        WaitVSync();  
    }  
}
```

とても短くてシンプルなプログラムですが、ポリモーフィズムの典型的な活用例になっています。

## Chapter 8 テニスゲームを作ろう

このチャプターでは未完成のテニスゲームのサンプルプログラムを使用して、ゲームを完成させてみましょう。

### 8.1 プログラムの構成

#### Game.cpp,Game.h

Game クラス。ゲームのメイン関数のような処理になる。

Game::Update 関数。ゲームループから毎フレーム呼ばれる更新関数。

Game::Render 関数。ゲームループから毎フレーム呼ばれる描画処理。

Game::Start 関数。ゲーム開始して一度だけ呼ばれる開始関数。

#### Player.cpp,Player.h

Player クラス。ユーザーが操作するプレイヤークラス。

Player::Update 関数。Game::Update からコールされている更新関数。

Player::Render 関数。Game::Render からコールされている描画関数。

Player::Init 関数。Game::Start 関数からコールされている初期化関数。

#### ball.cpp,ball.h

Ball クラス。テニスボールクラス。テニスコートの外に出ようとすると反射する。プレイヤーに衝突しても反射します。

Ball::Update 関数。Game::Update からコールされている更新関数。

Ball::Render 関数。Game::Render からコールされている描画関数。

#### Court.cpp,Court.h

Court クラス。テニスコートクラス。

#### GameCamera.cpp,GameCamera.h

GameCamera クラス。

### 8.2 アニメーション付き 3D モデル表示。

今回のサンプルからアニメーション付き 3D モデルを表示する機能が追加されています。この節ではその機能について説明していきます。

#### 8.2.1 X ファイル

X ファイルとはモデルフォーマットと言われるもので、DirectX9 までサポートされていた形式になります。DirectX10 以降はサポートされていないため、X ファイル自体は過去の遺物となっています。しかし、モデルフォーマットというのはどこに行っても通用する標準化されたものというものは存在しません。そして、3D モデルを表示する基本的な理論というのは 10 年以上前から変化がなく、枯れた技術となっています。そのため X ファイルを使ったモデル表示を学ぶことは無駄にはなりません。

## 8.2.2 CSkinModel、CSkinModelData

この二つのクラスは X ファイルを使用した、3D モデルを表示するための機能を提供するクラスです。典型的な使用方法を下記に記述します。

### X ファイルのロード

```
CSkinModelData modelData;
CSkinModel model;
void Init()
{
    modelData.LoadModelData("Assets/modelData/player.x", NULL);
    model.Init(&modelData);
}
```

### ワールド行列の更新

```
void Update()
{
    model.UpdateWorldMatrix(position, CQuaternion::Identity, CVector3::One);
}
```

### モデルの表示

```
void Render(CRenderContext& renderContext)
{
    model.Draw(renderContext, gameCamera->GetViewMatrix(), gameCamera->GetProjectionMatrix());
}
```

### 実習課題

- ① 対戦相手を表示できるようにする。
  - ② 対戦相手もボールを返せるようにする。
  - ③ 対戦相手は AI で勝手に動作するようにする。
- (AI というほど大したものである必要はありません。自動で動いていれば OK です。)



## Chapter 9 回転

このチャプターでは Lesson\_12 のプログラムを使用して、3D オブジェクトを回転させる方法を学んでいこうと思います。

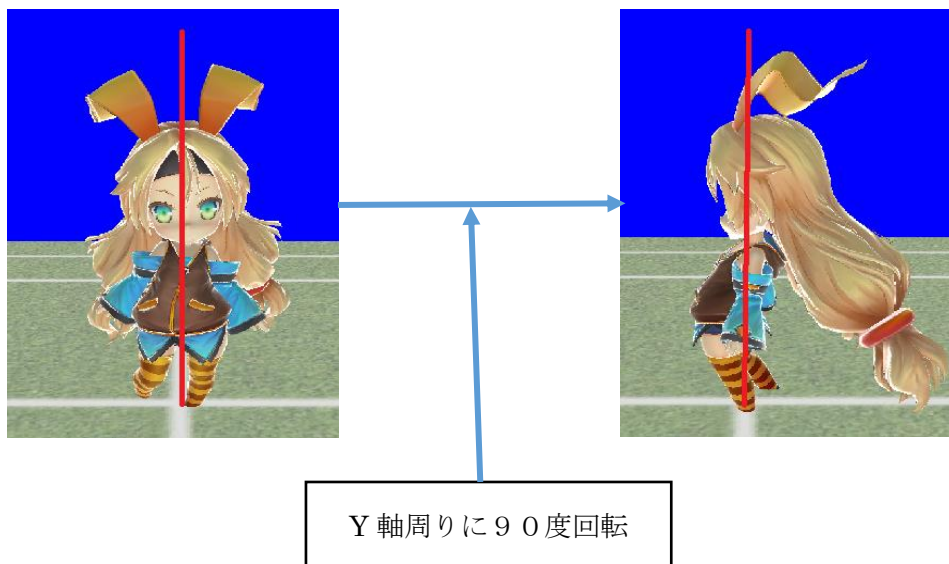
### 9.1 クォータニオン(四元数)

3D モデルの回転を表現するにはいくつか手法があるのですが、今回は 3D ゲームで主流となっているクォータニオンを使用した回転の表現について見ていきましょう。この授業は数学の授業ではないのでクォータニオンの数学的な定義や証明は行いません。クォータニオンをゲームでどのように使用するかという点に注視して説明を行います。

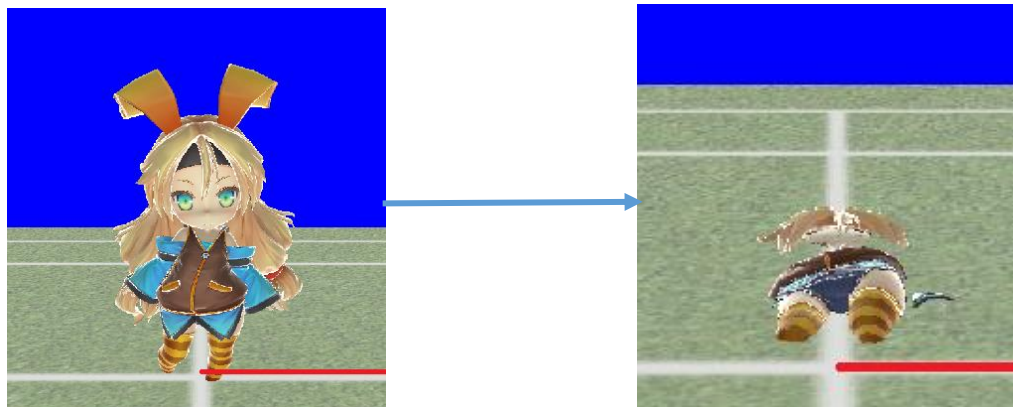
#### 9.1.1 任意の軸周りの回転

回転の表現にクォータニオンを使用する理由の大きな理由の一つに任意の軸周りの回転を簡単に扱うことができるというものがあります。では任意の軸周りの回転とはどのようなものなのか見ていきましょう。

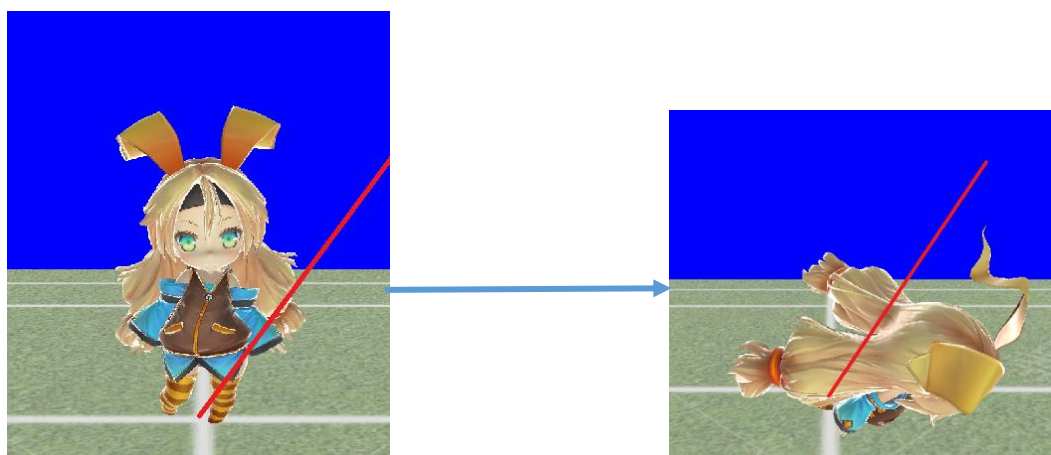
例えば、下のようにユニティちゃんを回転させた場合は Y 軸周りに 90 度回転させることになります。



1 では次は X 軸周りに回転させてみましょう。



2  
3  
4 では最後に下記のように斜めの軸で回してみましよう。



5  
6  
7  
8 では、クォータニオンではこれらの回転をどのように表現するのかを見ていきましょう。

9  
10 任意の軸を  $axis$ 、回転角度を  $\theta$  として、回転を表すクォータニオンを  $qRot$  とすると

11  $qRot.x = axis.x * \sin(0.5 \theta)$

12  $qRot.y = axis.y * \sin(0.5 \theta)$

13  $qRot.z = axis.z * \sin(0.5 \theta)$

14  $qRot.w = \cos(0.5 \theta)$

15 となる。

16  
17 今回皆さんに提供しているサンプルプログラムには、この計算は関数化されているため、下  
18 記のようなプログラムを記述するだけで、回転クォータニオンを作成できます。

Y 軸周りに 90 度回転させるクォータニオンを作成するサンプルコード。

```
CQuaternion qRot;
qRot.SetRotation(CVector3(0.0f, 1.0f, 0.0f), CMath::DegToRad(90.0f));
```

続いて、斜めの軸で回転させる場合のサンプルコードを紹介します。任意の軸というのは大きき 1 のベクトルにする必要があるということに注意してください。

```
//斜めの軸を作成する。
CVector3 rotAxis(1.0f, 1.0f, 0.0f);
//大きき 1 にするためにベクトルを正規化。
rotAxis.Normalize();
//Unityちゃんを回す。
CQuaternion qRot;
qRot.SetRotation(rotAxis, CMath::DegToRad(-90.0f));
```

## 9.2 クォータニオンの乗算

ゲームで回転を扱いだすと、「Y 軸に 90 度回した後で、X 軸に 45 度回したいとか」、「毎フレーム Y 軸周りに 5 度ずつ回したい」などと言ったことを実装したくなってきます。これらの仕様はクォータニオン同士の乗算を実装すると実現することができます。クォータニオン同士の乗算は下記のように行います。

X 軸周りの回転を表しているクォータニオンを qRotX、Y 軸周りの回転を表しているクォータニオンを qRotY としたとき、乗算されたクォータニオン qRot は下記の計算で求まる。

```
qRot.w = qRotX.w * qRotY.w - qRotX.x * qRotY.x - qRotX.y * qRotY.y - qRotX.z * qRotY.z
qRot.x = qRotX.w * qRotY.x + qRotX.x * qRotY.w + qRotX.y * qRotY.z - qRotX.z * qRotY.y
qRot.y = qRotX.w * qRotY.y - qRotX.x * qRotY.z + qRotX.y * qRotY.w + qRotX.z * qRotY.x
qRot.z = qRotX.w * qRotY.z + qRotX.x * qRotY.y - qRotX.y * qRotY.x + qRotX.z * qRotY.w
```

皆さんに提供している tkEngine にはクォータニオンの乗算を行う処理を用意していますので、上記の計算を行う必要はありません。tkEngine の CQuaternion を使用した場合のクォータニオンの乗算のサンプルコードを下記に示します。

```
CQuaternion qRotX, qRotY, qRot;
//X 軸周りに 45 度回転するクォータニオンを作成する。
qRotX.SetRotation(CVector3(1.0f, 0.0f, 0.0f), CMath::DegToRad(45.0f));
//Y 軸周りに 90 度回転するクォータニオンを作成する。
qRotY.SetRotation(CVector3(0.0f, 1.0f, 0.0f), CMath::DegToRad(90.0f));
//X 軸周りの回転と Y 軸周りの回転を乗算する。
qRot.Multiply(qRotX, qRotY);
```

クォータニオンの乗算は交換法則が成り立っていないところに注意してください。例えば  $4 \times 5$  は  $5 \times 4$  にしても結果は 20 になります。これは交換法則が成り立っています。しかしクォータニオンは  $qRotX \times qRotY$  と  $qRotY \times qRotX$  で結果が異なります。

#### 実習課題

Lesson\_13 のプログラムを使用して下記の仕様を実装しなさい。解答例となる実行ファイルを下記のパスにアップしているので、挙動はそれを参考にしなさい。

Lesson\_13/解答のデモプログラム/ Answer.exe

① キーボードの左右キーが押されるとユニティちゃんを Y 軸周りに回転させる。

② キーボードの上下キーが押されるとユニティちゃんを X 軸周りに回転させる。

## Chapter 10 拡大

ここまで 3D ゲームのキャラクターの位置を表すデータとして 3 要素のベクトル型、回転を表すデータとしてクォータニオンを紹介してきました。このチャプターではキャラクターの拡大を見ていこうと思います。

3D ゲームで拡大下記のように 3 要素のベクトル型として扱われます。

```
//プレイヤークラス。
class Player{
public:
    CVector3    position; //座標
    CQuaternion rotation; //回転
    CVector3    scale;    //拡大
};
```

そして、プレイヤーを等倍で表示したい場合は拡大率を下記のように設定します。

```
Player player;
player.scale.x = 1.0f; //X 軸方向の拡大率。
player.scale.y = 1.0f; //Y 軸方向の拡大率。
player.scale.z = 1.0f; //Z 軸方向の拡大率。
```

もう勘のいい人は気づいているかと思いますが、プレイヤーを X 軸方向に拡大したい場合は player.scale.x の値を変更してやることになります。Y 軸であれば player.scale.y、Z 軸であれば player.scale.z の値を変更してやることになります。

### 10.1 ミラー

拡大の考え方はそこまで難しいものではないと思いますので、この節では拡大のちょっとしたトリックのようなテクニックを使った、ミラーモデルと言われる、まるで鏡に映っているかのように左右反転しているモデルの表示の仕方を教えます。実はミラーモデルは X 軸方向の拡大率を-1.0 倍するだけで実現できます。

```
Player player;
player.scale.x = -1.0f;
```

たったこれだけです。どうでしょうか非常に簡単でしょう？

モデルを X 軸方向に-1.0 倍するだけで、ミラーモデルの完成です。ただし、実はまだこれだけでは絵は正しく表示されません。実は 3D モデルを構成するポリゴンと言われるものには表面と裏面というものが存在します。そして、描画負荷を上げないために大抵の場合は裏面の描画は行われなくなっています。モデルを X 軸方向に-1.0 倍したということはモデルをひっくり返したことになりますので、ポリゴンの裏面が表面に来てしまうことに

なるので、このままではモデルはまともに表示されません。そのため、下記のようなコードを Draw 関数の前で実行して、これから各モデルは裏面を描画しますよ～という風に GPU に教えてやる必要があります。

```
//これから描画するモデルが裏面描画であることを GPU に教える。  
renderContext.SetRenderState(RS_CULLMODE, CULL_CW);  
skinModel.Draw(renderContext,camera.GetViewMatrix(), camera.GetProjectionMatrix());  
//モデルを書いたら表面描画に戻しておく。じゃないと、これ以降の描画がおかしくなりますので。  
renderContext.SetRenderState(RS_CULLMODE, CULL_CCW);
```

#### 実習課題

Lesson\_14 を使用して下記の仕様を実装しなさい。解答となるデモプログラムは下記のパスにアップしているので、それを参考にしなさい。

Lesson\_14/解答のデモプログラム/ Answer. exe

- ① 上下のキーが押されるとユニティちゃんが Y 軸方向に拡大縮小する。
- ② 左右のキーが押されるとユニティちゃんが X 軸方向に拡大縮小する。
- ③ X 軸方向のミラーモデルの処理を実装する。

## 1 Chapter 10 行列

2 ここまでで、3D モデルをワールド空間で移動、回転、拡大する方法を見てきました。移  
3 動は3要素のベクトル(x, y, z)、回転はクォータニオン(x, y, z, w)、拡大は3要素のベクトル  
4 (x, y, z)を使用して表現していました。このチャプターまでの内容をしっかりと理解できて  
5 いれば、3D ゲームのプレイヤーを作成しようと考えた場合、下記のようなクラスを作成す  
6 ることが考えられるはずです。

### 7 Player.h

```
//プレイヤークラスの定義。
class Player{
public:
    //////////////////////////////////////
    //メンバ変数
    //////////////////////////////////////
    CVector3    position;    //座標。
    CQuaternion rotation;    //回転。
    CVector3    scale;       //拡大率。

    //////////////////////////////////////
    //メンバ関数。
    //////////////////////////////////////
    //コンストラクタ。
    Player();
    //デストラクタ。
    ~Player();
    //更新処理。
    void Update();
};
```

8

### 9 Player.cpp

```
//コンストラクタ
Player::Player()
{
    //コンストラクタでメンバ変数を初期化。
    position.x = 0.0f;
    position.y = 0.0f;
    position.z = 0.0f;

    rotation.x = 0.0f;
    rotation.y = 0.0f;
    rotation.z = 0.0f;
    rotation.w = 1.0f;

    scale.x = 1.0f;
    scale.y = 1.0f;
    scale.z = 1.0f;
}

void Player::Update()
```

```

{
    if(KeyInput().IsLeftPress()){
        //左のキーが押された。
        position.x -= 0.5f;
    }
    if(KeyInput().IsRightPress()){
        //左のキーが押された。
        position.x += 0.5f;
    }
    if(KeyInput().IsUpPress()){
        //左のキーが押された。
        position.z += 0.5f;
    }
    if(KeyInput().IsDownPress()){
        //左のキーが押された。
        position.x -= 0.5f;
    }
}

```

しかし、実は position、rotation、scale の値をいくら変更しても 3D モデルはワールド空間上を移動、回転、拡大することはありません。3D モデルをワールド空間で動かすためには最終的には行列というものを作成する必要があるからです。

## 10.1 行列とは？

行列とはその名のとおり、行と列で成り立つデータです。3D ゲームでは主として、4×4 行列か 3×4 行列が使用されます。このチャプターでは 4×4 行列を扱います。4x4 行列は下記のように表現されます。

$$\begin{pmatrix} 1 & 2 & 2 & 3 \\ 4 & 4 & 3 & 4 \\ 4 & 12 & 2 & 1 \\ 6 & 5 & 4 & 4 \end{pmatrix}$$

プログラムで記述すると下記のようなコードになります。

```
float matrix[4][4];
```

3D モデルをワールド空間で移動、回転、拡大するためには、それらの情報を使って**ワールド行列**と言われるものを作成する必要があります。

## 10.2 平行移動行列

まず、ワールド行列の前に平行移動行列について見ていきましょう。名前から推測できる人もいるかもしれませんが、これがワールド空間上の座標を表すことになります。平行移動行列は下記のような形になります。例えば、プレイヤーの座標が 10、20、30 になる場合は



1 平行移動行列は下記ようになります。

$$2 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \mathbf{10} & \mathbf{20} & \mathbf{30} & \mathbf{1} \end{pmatrix}$$

3 赤字になっている部分が平行移動成分となります。つまり、行列の 4 行目に平行移動成分が  
4 記録されることになります。

5 では、ベクトルから平行移動行列を作成するコードを見てみましょう。

```
CMatrix transMatrix;
//一行目を設定していく。
transMatrix.m[0][0] = 1.0f;
transMatrix.m[0][1] = 0.0f;
transMatrix.m[0][2] = 0.0f;
transMatrix.m[0][3] = 0.0f;
//2行目を設定していく。
transMatrix.m[1][0] = 0.0f;
transMatrix.m[1][1] = 1.0f;
transMatrix.m[1][2] = 0.0f;
transMatrix.m[1][3] = 0.0f;
//3行目を設定していく。
transMatrix.m[2][0] = 0.0f;
transMatrix.m[2][1] = 0.0f;
transMatrix.m[2][2] = 1.0f;
transMatrix.m[2][3] = 0.0f;
//4行目を設定していく。
transMatrix.m[3][0] = position.x;
transMatrix.m[3][1] = position.y;
transMatrix.m[3][2] = position.z;
transMatrix.m[3][3] = 1.0f;
```

6  
7 非常に長いコードになってしまいました。私が皆さんに提供している tkEngine の CMatrix  
8 クラスには平行移動行列を生成するメンバ関数が用意されていて、それを使用すると下記  
9 のようなコードになります。

```
CMatrix transMatrix;
transMatrix.MakeTranslation(position); //平行移動行列を作成する。
```

10

### 11 10.3 回転行列

12 続いて回転を表す回転行列について見てみましょう。例えば Y 軸周りに  $\theta$  回転する回転  
13 行列の場合は下記ようになります。

$$14 \begin{pmatrix} \mathbf{\cos\theta} & 0 & \mathbf{-\sin\theta} & 0 \\ 0 & 1 & 0 & 0 \\ \mathbf{\sin\theta} & 0 & \mathbf{\cos\theta} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

15

16 続いて X 軸周りに  $\theta$  回転する回転行列は下記ようになります。

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

最後に Z 軸周りに  $\theta$  回転する回転行列は下記ようになります。

$$\begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

X 軸周りに  $\theta$ 、Y 軸周りに  $\theta$  回転する行列は行列の乗算を行うことで求めることができます。

では、クォータニオンから回転行列を作成するコードを見てみましょう。クォータニオンから回転行列を生成するプログラムは非常に複雑になるので、最初から **tkEngine** の **CMatrix** クラスのメンバ関数を使用します。

```
CMatrix rotationMatrix;
rotationMatrix.MakeRotationFromQuaternion(rotation); //クォータニオンから回転行列を生成。
```

#### 10.4 拡大行列

最後に拡大を表す拡大行列について見ていきましょう。例えば、プレイヤーの拡大率が X 軸に 2.0 倍、Y 軸に 1.5 倍、Z 軸に 0.5 倍の場合は下記ようになります。

$$\begin{pmatrix} 2.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.5 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

このようになります。

では、拡大行列を生成するコードを見ていきましょう。このコードも **CMatrix** クラスのメンバ関数を使用します。

```
CMatrix scaleMatrix;
scaleMatrix.MakeScaling(scale);
```

#### 10.5 ワールド行列

では、いよいよ 3D モデルをワールド空間で動かすためのワールド行列を見ていきましょう。ワールド行列は平行移動行列、回転行列、拡大行列を混ぜ合わせたものとなります。このまぜ合わせは行列の乗算によって実現できます(クォータニオンと似ています)。

**CMatrix** クラスには行列の乗算を行うメンバ関数が用意されています。その関数を利用して、ワールド行列を作成するコードを見てみましょう。

1

```

//平行移動行列を作成する。
CMatrix transMatrix;
transMatrix.MakeTranslation(position);
//回転行列を作成する。
CMatrix rotationMatrix;
rotationMatrix.MakeRotationFromQuaternion(rotation);
CMatrix scaleMatrix;
scaleMatrix.MakeScaling(scale);
//ワールド行列を作成する。
CMatrix worldMatrix;
worldMatrix.Mul(scaleMatrix, rotationMatrix);
worldMatrix.Mul(worldMatrix, transMatrix);

```

2

3 行列の乗算はクォータニオンと同様に交換法則が成り立っていません。scaleMatrix ×  
4 rotationMatrix と rotationMatrix × scaleMatrix の結果は異なります。

5 ワールド行列を作成するときの乗算の順番は基本的に下記のようになります。

6 **拡大行列 × 回転行列 × 平行移動行列**

7 この乗算順番を間違えると、意図しない結果になることがあります。

8

## 9 10.6 モデルの頂点座標のワールド変換

10 3D モデルがワールド空間で移動、回転、拡大するということはモデルの頂点が移動、回  
11 転、拡大していることになります。モデルの頂点座標は 3 次元のベクトルで表現されてい  
12 ます。このモデルの頂点座標を先ほど作成したワールド行列で変換していくことでモデル  
13 はワールド空間を歩く、旋回、拡大することができるようになります。では 3 次元のベクト  
14 ルを行列を使って変換する式を見てみましょう。変換前の頂点座標を  $V$ 、変換後の頂点座標  
15 を  $V'$  とします。

$$16 \begin{pmatrix} V'.x \\ V'.y \\ V'.z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 10 & 30 & 40 & 1 \end{pmatrix} \begin{pmatrix} V.x & V.y & V.z & 1 \end{pmatrix}$$

17

18 行列とベクトルの乗算は下記のように計算されます。

19

$$20 \begin{pmatrix} V.x * \cos\theta + V.z * \sin\theta + 10 \\ V.y + 30 \\ V.x * -\sin\theta + V.z * \cos\theta + 40 \\ 1 \end{pmatrix}$$

21

22 この頂点座標の変換は GPU で実行されるシェーダープログラムで行われます。このシェー  
23 ダープログラムもプログラマが記述する必要があります。今回はシェーダーの説明は行い

- 1    ませんが、シェーダーは非常に重要な技術でシェーダーがかけるプログラマというのは非
- 2    常に重宝されることになります。