

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つのファイルです。

ソースファイル

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_

```

1

2

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、確認ができたなら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

```

- 1 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
- 2 た。
- 3
- 4 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;
}

```

- 5
- 6 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
- 7 どは行っていないため、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。
- 8 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。
- 9

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; //XZ平面での移動速度。
    m_moveSpeed.z = 1.f;
    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

1
2
3
4
5
6
7
8
9
10

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が。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。

- 1 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
2 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。

3

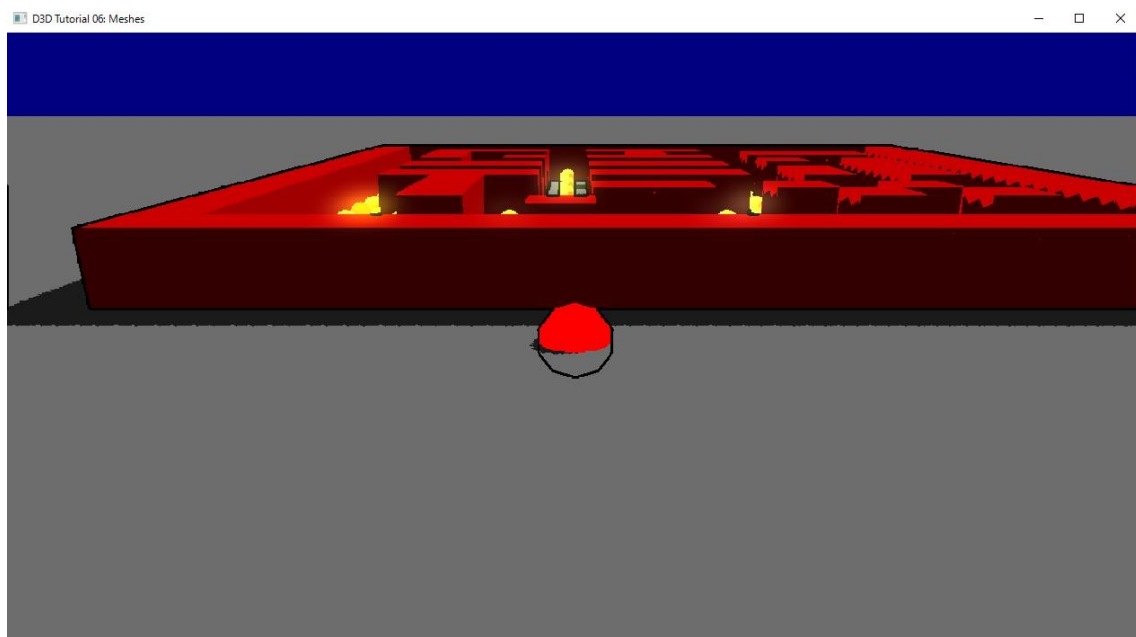
4 CCamera.cpp

```
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();
}
```

5

- 6 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。

7



8

9

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。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。

1
2
3

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** は下記のようになります。

$$\mathbf{V} = \mathbf{P} - \mathbf{E}$$

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

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 = Player0.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();
```

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 = Player0.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` 関数に追加すると食べ物を削除することができます。

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。

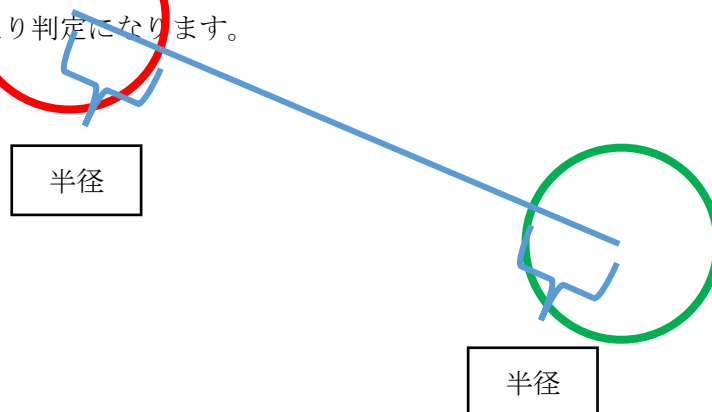
5.1 色々な当たり判定

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。

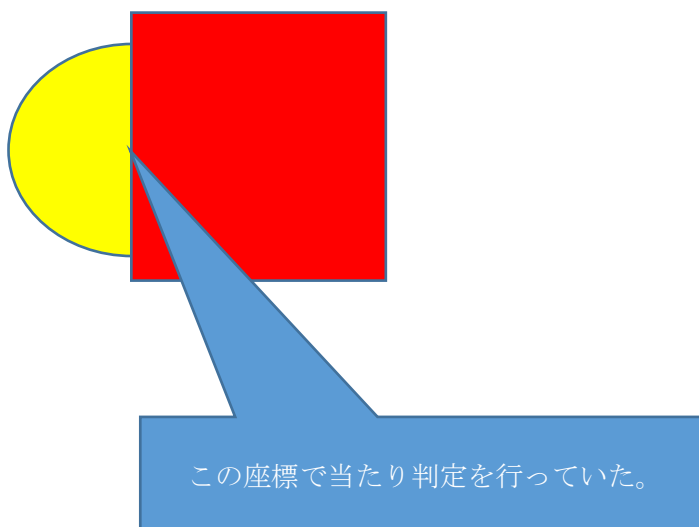
パックマンのマップは CMapBuilder の sMap 配列を使用して、構築されています。
 この SMap 配列の値が 1 ならば壁なので移動しないという処理を記述してやれば、パックマンは壁にめり込まなくなるはずです。ではこのトリックをお教えしましょう。
 パックマンの床と壁のサイズは GRID_SIZE になります。
 この GRID_SIZE でパックマンの座標を除算してやると、パックマンがマップ上のどの場所にいるのかを判定することができます。

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

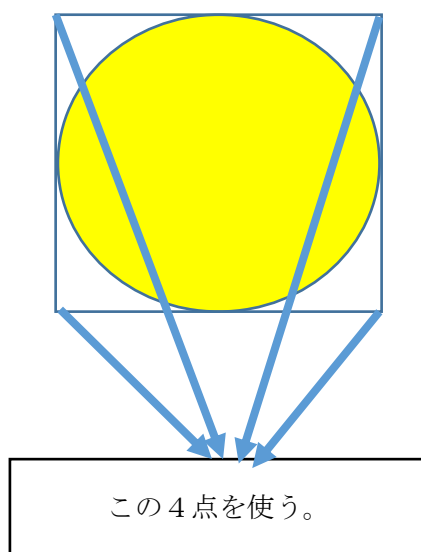
いかがでしょうか。非常に簡単なプログラムで壁の判定が行えました。では、ここまでの説明を参考にしてパックマンが壁をすり抜けないようにしてください。
 今回はパックマンが壁に半分めり込むと思いますが、それは考慮しなくて構いません。

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のようになります。

```
//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す。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、下記のようなコードを記述してきたはずですが。

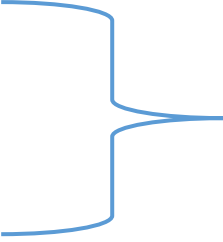
```
if(KeyInput().IsPressA() == true ){  
    position.x += 10;  
}
```

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。

```

1 while(true){
2     if(KeyInput().IsPressA() == true ){
3         position.x += 10;
4     }
5 }

```



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が、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。

```

17 while(true){
18     if(KeyInput().IsPressA() == true ){
19         position.x += 10;
20     }
21     //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。
22     WaitVSync();
23 }

```

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。

```
CGameObjectManager::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;
lGameObject* gameObjects[64];
int main()
{
    int numEntryGameObjct = 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 < numEntryGameObjct; 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。

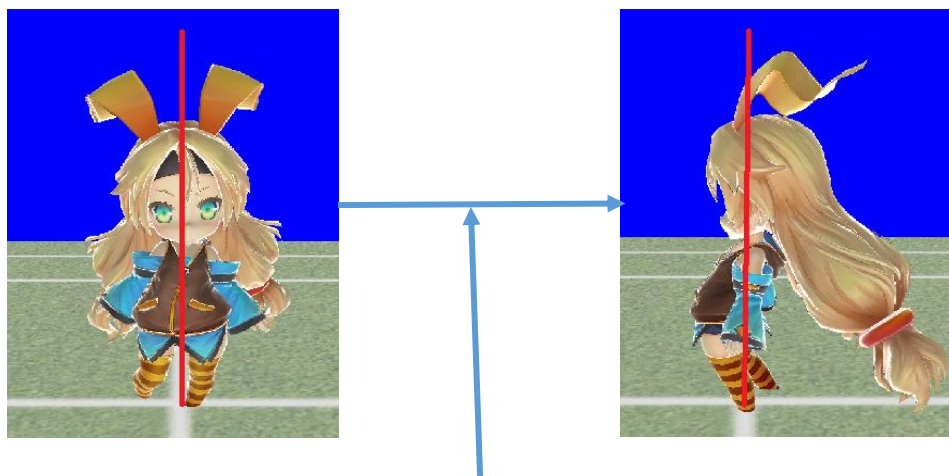
9.1 クォータニオン(四元数)

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の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

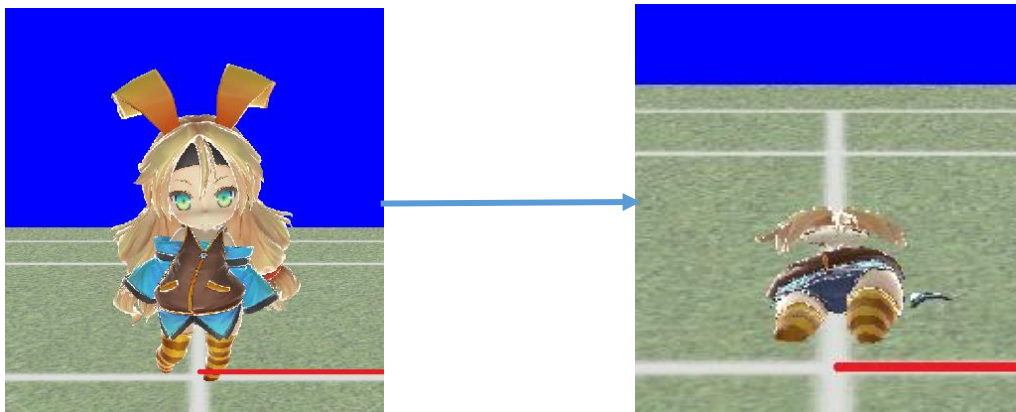
回転の表現にクォータニオンを使用する理由の大きな理由の一つに任意の軸周りの回転を簡単に扱うことができるというものがあります。では任意の軸周りの回転とはどのようなものなのか見ていきましょう。

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。

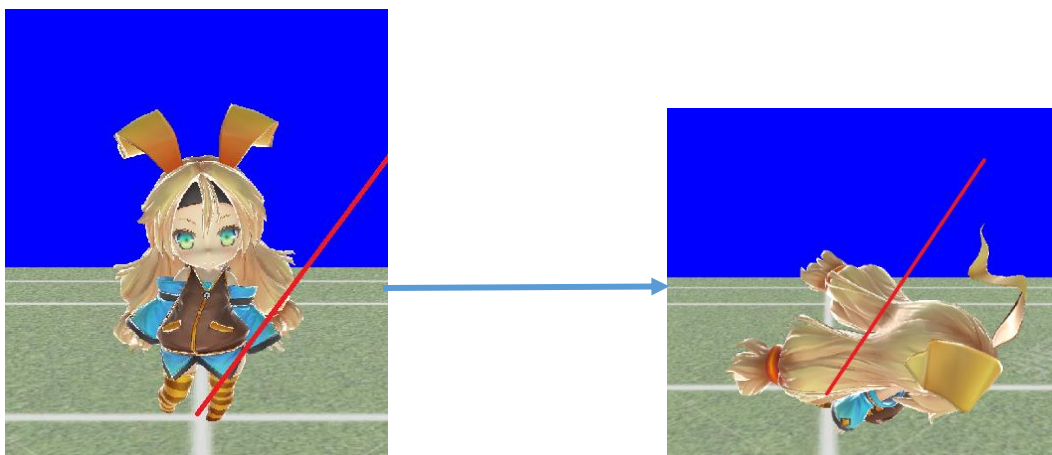


Y 軸周りに 90 度回転

では次は X 軸周りに回転させてみましょう。



では最後に下記のように斜めの軸で回してみましょう。



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。

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

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

```
1    qRot.y = axis.y * sin(0.5 θ)
```

```
2    qRot.z = axis.z * sin(0.5 θ)
```

```
3    qRot.w = cos(0.5 θ)
```

4 となる。

5
6 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、
7 下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。

8
9 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));
```

10
11 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
12 大きさ 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);  
//大きさ 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));
```

15 9.2 クォータニオンの乗算

16 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
17 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
18 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
19 ニオン同士の乗算は下記のように行います。

20
21 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
22 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。

```
23    qRot.w = qRotX.w * qRotY.w - qRotX.x * qRotY.x - qRotX.y * qRotY.y - qRotX.z * qRotY.z  
24    qRot.x = qRotX.w * qRotY.x + qRotX.x * qRotY.w + qRotX.y * qRotY.z - qRotX.z * qRotY.y  
25    qRot.y = qRotX.w * qRotY.y - qRotX.x * qRotY.z + qRotX.y * qRotY.w + qRotX.z * qRotY.x  
26    qRot.z = qRotX.w * qRotY.z + qRotX.x * qRotY.y - qRotX.y * qRotY.x + qRotX.z * qRotY.w
```

27
28 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
29 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
30 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くに注意してください。例えば 4×5 は 5×4 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ってしまうので。  
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。

Chapter 10 行列

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(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ることが考えられるはずです。

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();
};
```

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 になる場合は平行移動行列は下記のようになります。

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

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記録されることになります。

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;
```

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 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。
```

10.3 回転行列

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 θ 回転する回転行列の場合は下記ようになります。

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

続いて X 軸周りに θ 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 θ 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 θ 、Y 軸周りに θ 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。

```
//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);
```

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 × rotationMatrix と rotationMatrix × scaleMatrix の結果は異なります。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ようになります。

拡大行列 × 回転行列 × 平行移動行列

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。

10.6 モデルの頂点座標のワールド変換

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、変換後の頂点座標を V' とします。

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

行列とベクトルの乗算は下記のように計算されます。

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

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常に重宝されることになります。

Chapter 11 ゲームエンジン

ゲーム制作を強烈にサポートしてくれるものに、ゲームエンジンと呼ばれるものがあります。ゲームエンジンは Unity、UnrealEngine4 といった商用のものと、各ゲーム会社が独自に作成を行っている内製ゲームエンジンがあります。下記に有名どころの内製ゲームエンジンを上げてみます。

- **MTフレームワーク**

カプコンの内製エンジン。XBox360、PS3 の世代で使用された。

- **クリスタルツールズ**

スクウェアエニックスのツールの集まりのような開発環境。ゲームエンジンと呼ぶほどのものでもなかった？FF13、FF14 で使用されている。

- **Fox エンジン**

メタルギアソリッド 5 などで使用されたコナミ製のエンジン。

これらは社外に名前がでてきたものですが、名前が出てこないだけで各社独自のエンジンを保持しています。

11.1 ゲームエンジン時代

日本でスマートフォンアプリを中心に Unity が広く使われるようになり、PS4 などのコンソール機で UnrealEngine4 採用タイトルが多数発売されだしました。そのため、現在は各社内製エンジンの開発などを行わなくてもゲームを開発できるようになっています。しかし、今でも内製エンジンの開発を行っている会社があります。何故でしょうか？理由としては下記の点が考えられます。

- **UnrealEngine4、Unity のライセンス料**

商用のゲームエンジンは当然ですがお金を支払わないと企業は使用することはできません。Unity などはライセンス料は比較的安価なのですが、UnrealEngine4 だと高価になるため使用しない場合があります。

- ・昔から開発していたタイトルだと、エンジンもあり、ノウハウもたまっているため外部エンジンを使うメリットが弱い。

続編物のタイトルなどでこの傾向が強い。

- ・ UnrealEngine4 や Unity よりある一点に特化すれば強力なエンジンが作れる。

アンチャーテッド、CoD、FF15、ナルティメットストームなどがそれに該当します。

このような理由で今も各会社が作成した内製ゲームエンジンが使用されています。

11.1 河原内製ゲームエンジン

ゲーム P G の授業では **tkEngine** という河原内製の **DirectX** を使用したゲームエンジンを使用して授業を進めてきました。ここから先は、このエンジンをさらに活用して多数のミニゲームを作成していきます。

11.2 なぜ Unity や UnrealEngine4 を使わないの？

この授業で内製ゲームエンジンを使用する理由は下記があげられます。

- ・ C++を学ぶため

- ・コンソール系の会社への就職作品として Unity、UnrealEngine4 を使った作品では 厳しい。まず無理。

なぜか？

C++を駆使して DirectX、OpenGL などゲームが作れる人は、Unity、

UnrealEngine4 は勉強すればほぼ使えるが逆は怪しい。

Unreal 専門会社のヒストリアでさえ、学生には C++ でゲームが作れるスキルを求めている。

- ・スマホ系の会社への就職作品としても C++からやっておけば Unity、

UnrealEngine4 は割と簡単。

内製エンジンを使うというのは、難易度的には下のようになります。

エンジンを作る>>>>>>>>内製エンジンを使う>>>>Unity や UE4 などを使う

2年生になるとエンジンを自分で作り、ゲーム大賞作品や就職活動作品を作ることになる

ですが、1年生はtkEngineを使用してゲーム制作を進めていきます。

Chapter 12 ゲームオブジェクトマネージャー

このチャプターでは tkEngine のゲームオブジェクトマネージャーについて学んでみましょう。これは tkEngine の中で呼称している俗称になるのですが、処理の自体は Unity、UnrealEngine4、各ゲーム会社の内製エンジンどこにでも似たようなものが存在しています。

12.1 Update Method パターン

復習になりますが、7.2 の UpdateMethod パターンを思い出してください。このパターンは「インスタンスを登録すると、自動的に更新関数が呼ばれる」というものでした。ゲームオブジェクトマネージャーはこのデザインパターンを活用しています。

12.2 インスタンスの登録

では、実際にインスタンスを登録してみましょう。GamePG_1/GameTmplate を適当な場所にコピーして、GameTemplate.sln を立ち上げてソリューションエクスプローラーから main.cpp を開いてください。

まず、IGameObject を継承して Player クラスを作成します。

```
class Player : public IGameObject{
public:
    void Update(); //IGameObject を継承したクラスは必ず Update を実装する必要がある。
};
```

では Player::Update を実装してみましょう。

```
void Player::Update()
{
    if(GetKeyAsyncState('A') != 0){ //A ボタンが押されたら
        MessageBox(NULL, "Press A Button", "メッセージ", MB_OK);
    }
}
```

```

}

int WINAPI wWinMain(
    HINSTANCE hInst,
    HINSTANCE hPrevInstance,
    LPWSTR lpCmdLine,
    int nCmdShow
)
{
    //tkEngine の初期化。
    InitTkEngine( hInst );
    NewGO<Player>(0);           //プレイヤーのインスタンスの生成。
    Engine0.RunGameLoop();      //ゲームループを実行。
    return 0;
}

```

キーボードの A が押されたら **Press A Button** というメッセージボックスが表示される **Update** 関数が実装できました。では **Player** クラス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みましょう。**wWinMain** 関数に下記のようなコードを追加してください。

NewGO は **IGameObject** を継承したクラスのインスタンスを生成して、ゲームオブジェクトマネージャーに登録するための関数です。ちょっと難しいかもしれませんが、今はそういうものだと思っていてください。

では、これがどのようなケースで使われるのか **FPS** ゲームを例に考えてみましょう。例えば **FPS** ゲームでは銃のトリガーを引いたら弾を発射することができますよね？そのため下記のような実装が考えられます。

まず **IGameObject** を継承した **Bullet** クラスを作成します。

```

class Bullet : public IGameObject{
public:
    void Update();
};

```

そして、**BulletUpdate** を実装します。

```

void Bullet::Update()
{
    MessageBox(NULL, “発砲されたよ”, “メッセージ”, MB_OK);
}

```


- 1 これで弾丸のクラスは作成できました。では先ほどの **Player::Update** を改造して、キー
 2 ボードの A ボタンが押されたら弾丸を発射できるようにしてみましょう。

```
void Player::Update()
{
    if(GetKeyAsyncState('A') != 0){ //A ボタンが押されたら
        NewGO<Bullet>(0); //弾丸のインスタンスを生成してゲームオブジェクトマネージャーに登録。
    }
}
```

3
4
5

6 12.3 インスタンスの登録解除

- 7 インスタンスの登録ができるのであれば、登録解除も行える必要があります。先ほどの
 8 FPS を例にすると、銃から発砲された弾丸は、キャラクターにヒットしたり、壁にヒット
 9 したりしたらその寿命を終えます。ゲームの世界から消えたはずなのにいつまでも
 10 Update 関数が呼ばれているのはおかしいですね。そのため、役割を終えたインスタン
 11 スはゲームオブジェクトマネージャーから登録解除してやる必要があります。今回は 2 パ
 12 ターンの解除の方法を見てみましょう。

13

14 12.3.1 NewGO の戻り値を使う

- 15 NewGO は生成したインスタンスのアドレスを返してきます。それを使用して登録解除を
 16 行う方法を見てみましょう。

```
class Player : public IGameObject{
public:
    int numBullet = 0; //NewGO を使用して生成したインスタンスの数。
    Bullet* bulletArray[100]; //Bullet のポインタ型の配列。
    void Update();
};
```

- 17 Player クラスのメンバ変数に生成したインスタンスの数を記録するメンバ変数と、Bullet
 18 のポインタ型の配列が追加されました。では Update 関数を見てみましょう。

```
void Player::Update()
{
    if(GetAsyncKeyState('A') != 0 && numBullet < 100){
        //Bullet のインスタンスを生成。
        bulletArray[numBullet] = NewGO<Bullet>(0);
        //生成したので数をインクリメント。
        numBullet++;
    }
}
```

- 19 キーボードの A が押されると最大 100 個まで弾丸を作れるようになりました。では、キー
 20 ボードの B が押されると弾丸を一つ削除することができるようプログラムを書き換えま
 21 しょう。

```
void Player::Update()
{
    if(GetAsyncKeyState('A') != 0 && numBullet < 100){
        //Bullet のインスタンスを生成。
        bulletArray[numBullet] = NewGO<Bullet>(0);
        //生成したので数をインクリメント。
        numBullet++;
    }
    if(GetAsyncKeyState('B') != 0 && numBullet > 0){
        numBullet--;
        DeleteGO(bulletArray[numBullet]);
    }
}
```

このように、tkEngine のゲームオブジェクトマネージャーは **DeleteGO** という関数に削除したいインスタンスのアドレスを渡してやることによって、削除を行うことができます。

12.3.2 this ポインタを使う

this ポインタを使用して、インスタンスが自分自身を削除することが可能です。例えば弾丸はインスタンスが生成されてから 60 フレーム経過すると削除されるという仕様だった場合、下記のようなコードを書く事ができます。

```
class Bullet : public IGameObject{
public:
    int frameCount = 0; //フレーム数をカウントするためのメンバ変数。
    void Update();
};
```

Bullet クラスにフレーム数をカウントするための **frameCount** という変数が追加されました。では Update 関数の実装を見てみましょう。

```
void Bullet::Update()
{
    frameCount++; //インスタンスが生成されてから経過したフレーム数をカウントアップ。
    if(frameCount == 60){
        //インスタンスが生成されてから 60 フレーム経過した
        DeleteGO(this); //自分自身を削除。
    }
}
```

これでインスタンスが生成されてから 60 フレーム経過したら自動的に削除が行われます。

Chapter 13 マップを作ろう

このチャプターでは Unity のエディタ拡張を活用してマップの作成を行い、作成したマップを tkEngine を使用してゲーム中に表示する方法を勉強しましょう。

13.1 tkTools でモデルを配置

下記のパスに UnityEditor を拡張した tkTools という Unity のプロジェクトがあります。このツールを使用してオブジェクトを配置して、マップを作成していきましょう。下記のパスに tkTools の使い方の動画がありますので、そちらを参照してください。

GamePG_1/動画/tkTools を使ったオブジェクト配置方法.mp4

13.2 配置情報を元にマップを表示する。

tkTools から出力した配置情報を使用して、マップを表示しているサンプルは下記のパスにあります。

GamePG_1/MapChip

ではサンプルプログラムの解説をしていきます。今回重要になるソースファイルは下記の4つになります。

Map.cpp

Map.h

MapChip.cpp

MapChip.h

13.3 Map クラス

Map クラスは tkTools から出力された配置情報を使って、マップを構築しています。配置情報は下記のように #include のトリックを使ってプログラムに組み込んでいます。

1

```
//マップの配置情報。
SMapInfo mapLocInfo[] = {
#include "locationInfo.h"
};
```

2

3 続いて、Map::Start 関数でマップの配置情報を元に MapChip クラスのインスタンスを生
4 成しています。MapChip クラスがマップを構成する各オブジェクトの描画を行うクラス
5 になります。例えば、マップに 100 個のオブジェクトが配置されている場合は、100 個の
6 MapChip クラスのインスタンスが生成されます。

7

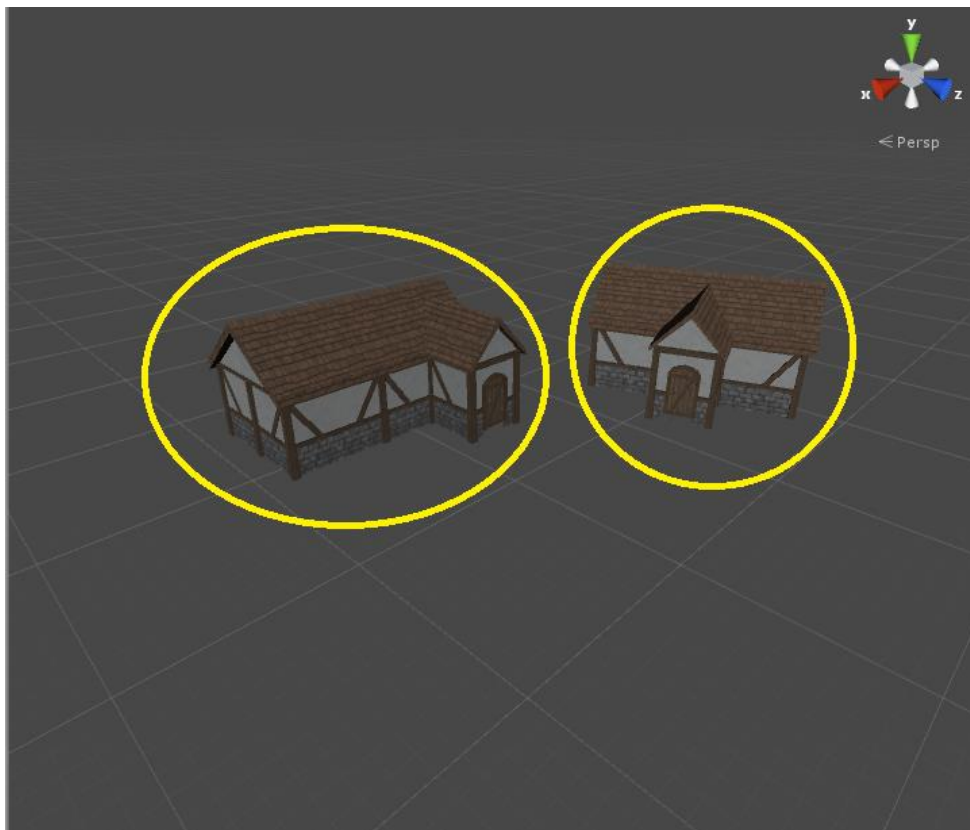
```
//マップにいくつのオブジェクトが配置されているか調べる。
int numObject = sizeof(mapLocInfo) / sizeof(mapLocInfo[0]);
//置かれているオブジェクトの数だけマップチップを生成する。
for (int i = 0; i < numObject; i++) {
    MapChip* mapChip = NewGO<MapChip>(0);
    //モデル名、座標、回転を与えてマップチップを初期化する。
    mapChip->Init(
        mapLocInfo[i].modelName,
        mapLocInfo[i].position,
        mapLocInfo[i].rotation
    );
```

8

9 13.4 MapChip クラス

10 MapChip クラスはオブジェクトの一つ一つを表すクラスになります。下記のようなマ
11 ップであれば MapChip のインスタンスは二つ生成されることになります。

12



- 1
- 2
- 3
- 4

では MapChip クラスの Init 関数を見てみましょう。

```
void MapChip::Init(const char* modelName, CVector3 position, CQuaternion rotation)
{
    //ファイルパスを作成する。
    char filePath[256];
    sprintf(filePath, "Assets/modelData/%s.x", modelName );
    //モデルデータをロード。
    skinModelData.LoadModelData(filePath, NULL);
    //CSkinModelを初期化。
    skinModel.Init(&skinModelData);
    //デフォルトライトを設定して。
    skinModel.SetLight(&g_defaultLight);
    //ワールド行列を更新する。
    //このオブジェクトは動かないので、初期化で一回だけワールド行列を作成すればおk。
    skinModel.Update(position, rotation, CVector3::One);
}
```

1 Init 関数は引数にモデルの名前、配置座標、回転を受け取っています。この引数を元に、
2 モデルをロードして、初期化を行っています。

3
4 あとは、描画を行うだけです。Draw 関数を見てみましょう。

```
void MapChip::Render(CRenderContext& renderContext)
{
    skinModel.Draw(
        renderContext,
        g_gameCamera->GetViewMatrix(),
        g_gameCamera->GetProjectionMatrix()
    );
}
```

6
7 特別特殊な処理はありませんね？MapChip クラスは単にモデルをロードして、描画して
8 いるだけのクラスです。

13 Chapter 14 コリジョン処理

14 このチャプターではゲームを作る上で避けて通ることができない、コリジョン処理につ
15 いて見ていきましょう。このチャプターではサンプルプログラムとして
16 GamePG_1/CollisionDemo を使用します。

18 13.1 衝突処理

19 ゲームにおいて物体同士の衝突処理というのはとても難易度の高いプログラムです。特
20 に 3D ゲームにおける衝突処理の難易度は 2D ゲームの比ではありません。衝突処理には
21 大きく分けて二つのフェーズがあります。一つ目は衝突検出、そして二つ目は衝突解決
22 です。衝突検出は物体がぶつかっているかどうかを調べる処理、衝突解決は物体が衝突し
23 ている場合にそのめり込みを解決する処理です。

25 13.2 物理エンジン

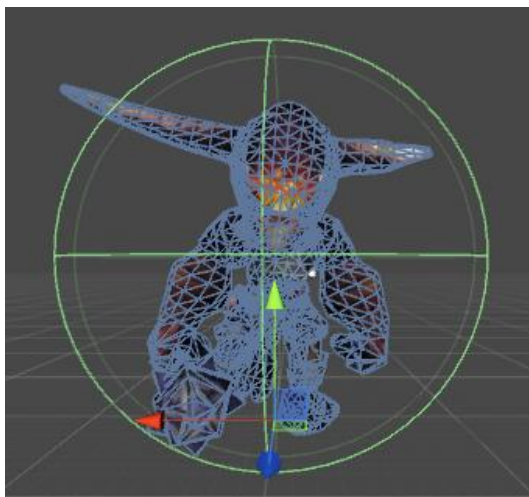
26 衝突処理はコリジョン処理とも呼ばれ、以降はコリジョンというキーワードを使用しま
27 す。前節でコリジョン処理は非常に難易度が高いという話をしましたが、物理エンジンの

登場によりその難易度は大幅に下がりました。河原内製エンジンの **tkEngine** は **bulletPhysics** というオープンソースの物理エンジンを使用しています。そのため、今回皆さんがコリジョン処理の難易度の高い部分をプログラミングするということはありません。ありがたく物理エンジンを活用しましょう。

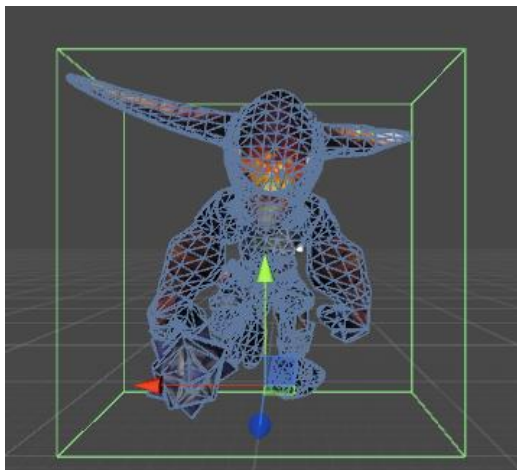
13.3 Collider

コリジョン処理を行うためには物体の形状を表すデータを作成する必要があります。このデータのことを **Unity** では **Collider** と呼称しています。この教材では **Unity** に合わせて当たりデータのことを **Collider** と呼称します。**Collider** にはいくつか種類があります。代表的な **Collider** を下記に示します。

- **SphereCollider** (球体形状の Collider)



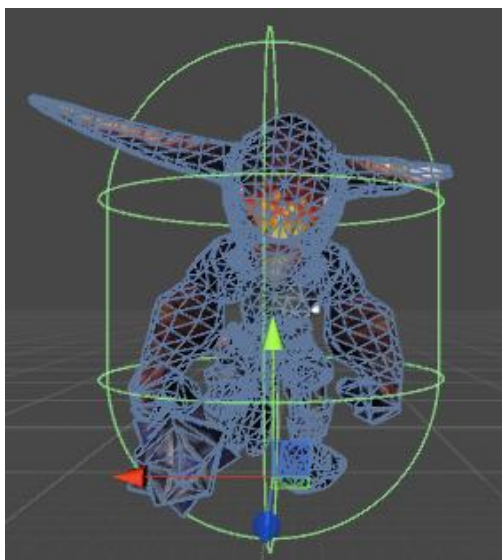
- **BoxCollider** (箱形状の Collider)



1

2

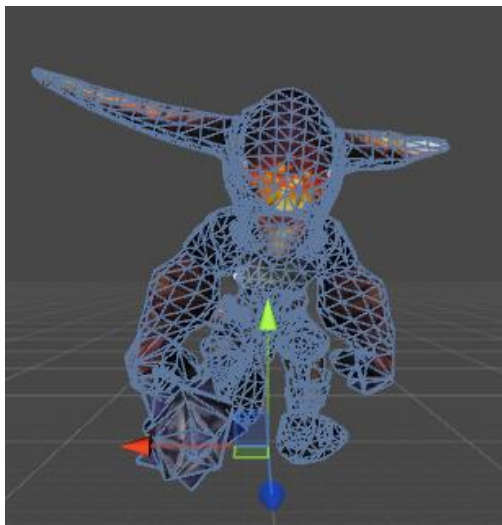
• **CapsuleCollider** (カプセル形状の Collider)



3

4

• **MeshCollider**(メッシュ形状の Collider)



5

6

多くのゲームではモデルの形状をそのまま Collider にする MeshCollider を使うことはありません。コリジョン処理は複雑な形状で処理を行うほど、処理が重くなり、コリジョン抜けが発生する可能性が上がります。そのため、BoxCollider、SphereCollider、CapsuleCollider を組み合わせて Collider を作成することが一般的です。

13.4 MeshCollider

前節で多くのゲームでは MeshCollider を使うことはないといいましたが、MeshCollider であれば、モデルデータをそのまま流用して Collider を作成することが出来ます。そのため、ここでは MeshCollider を使用してオブジェクトに当たりデータを作成してみようと思います。GamePG_1/CollisionDemo/MapChip.h を開いて下さい。

```
class MapChip : public IGameObject
{
    .
    .
    .
    CMeshCollider meshCollider;//メッシュコライダー。
    .
    .
    .
};
```

MapChip クラスに meshCollider というメンバ変数が追加されています。
続いて、MapChip.cpp を開いてください。

```
void MapChip::Init(const char* modelName, CVector3 position, CQuaternion rotation)
{
    .
    .
    .
    //メッシュコライダーの作成。
    meshCollider.CreateFromSkinModel (&skinModel, &skinModel.GetWorldMatrix());
    .
    .
    .
}
```

MapChip::Init 関数に MeshCollider を作成するコードが追加されています。
第一引数に MeshCollider の元になる skinModel のインスタンスを渡しています。第二引数には skinModel のワールド行列を渡しています。これで MeshCollider の作成は完了です。

13.4 RigidBody

前節のプログラムで **MeshCollider** を作成することが出来ました。しかしまだコリジョン処理を行うことが出来ません。物理エンジンでコリジョン処理を行うためには剛体を作成して、その剛体を物理ワールドに登録する必要があります。剛体とは変形しない物体ということです。では剛体を作成するプログラムを見ていきましょう。**MapChip.h** を開いてください。

```
class MapChip : public IGameObject
{
    .
    .
    .
    CRigidBody rigidBody; //剛体。
};
```

MapChip クラスに **CRigidBody** 型の **rigidBody** というメンバ変数が追加されました。では剛体の作成のコードを見てみましょう。**MapChip.cpp** を開いてください。

```
void MapChip::Init(const char* modelName, CVector3 position, CQuaternion rotation)
{
    .
    .
    .

    //剛体の作成。
    RigidBodyInfo rbInfo;
    //剛体のコライダーを渡す。
    rbInfo.collider = &meshCollider;
    //剛体の質量。0.0だと動かないオブジェクト。背景などは0.0にしよう。
    rbInfo.mass = 0.0f;
    rbInfo.pos = position;
    rbInfo.rot = rotation;
    rigidBody.Create(rbInfo);
    //作成した剛体を物理ワールドに追加する。
    PhysicsWorld().AddRigidBody(&rigidBody);
}
```

MapChip::Init 関数に剛体を作成して、作成した剛体を物理ワールドに登録しています。まず、剛体を作成するための情報となる **RigidBodyInfo** を作成しています。剛体の形状を表す **collider** の設定や質量の設定などを行っています。質量が **0.0** になっている点に注意してください。質量を **0.0** にすると動かないオブジェクトになります。建物や地面などの動くと困るオブジェクトは質量を **0.0** にしましょう。

13.5 CharacterController

これで背景のコライダーの作成もできて、剛体も物理ワールドに登録できました。あとはキャラクターを動かしてコリジョン処理を行うだけです。**tkEngine** にはキャラクターの制御を行ってくれる **CharacterController** というクラスがあります。では、**GamePG_1/CollisionDemo/Player.h** を開いてください。

```
#include "tkEngine/character/tkCharacterController.h"
```

```
class Player : public GameObject
{
    .
    .
    .
    CCharacterController characterController //キャラクターコントローラー。
};
```

Player クラスに CCharacterController 型の characterController というメンバ変数が追加されています。では GamePG_1/CollisionDemo/Player.cpp を開いて下さい。

```
void Player::Start()
{
    .
    .
    .
    characterController.Init(0.5f, 1.0f, position);
}
```

Player の Start 関数で characterController の初期化を行っています。第一引数はキャラクターの半径。第二引数はキャラクターの高さです。第三引数はキャラクターの初期位置です。

```
void Player::Update()
{
    //キャラクターの移動速度を決定。
    CVector3 move = characterController.GetMoveSpeed();
    move.x = -Pad(0).GetLStickXF() * 5.0f;
    move.z = -Pad(0).GetLStickYF() * 5.0f;

    //決定した移動速度をキャラクタコントローラーに設定。
    characterController.SetMoveSpeed(move);
    //キャラクターコントローラーを実行。
    characterController.Execute();
    //実行結果を受け取る。
    position = characterController.GetPosition();
    //ワールド行列の更新。
    skinModel.Update(position, CQuaternion::Identity, CVector3::One);
}
```

CharacterController に移動速度を設定して、Execute 関数を実行すると CharacterController の座標が移動速度に応じて移動します。CharacterController::Execute 関数の中で

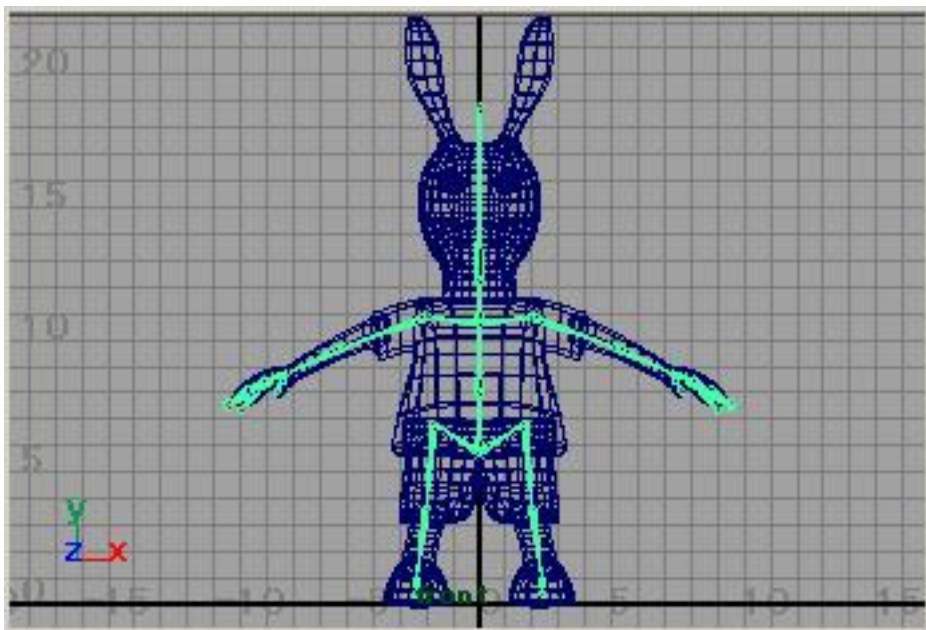
リジョン検出と解決を行っています。そのため、本関数を読んだ後で `CharacterController` の `GetPosition` を使用すると移動した結果の座標を返してきます。これでキャラクターのコリジョン処理が実装できました。

Chapter 15 アニメーション

このチャプターではサンプルプログラムの `GamePG_1/AnimationDemo` を使ってキャラクターをアニメーションさせるプログラムを勉強しましょう。

15.1 スケルトンアニメーション

スケルトンアニメーションとは多くの 3D モデルのアニメーションで採用されている手法です。3D モデルにボーン(骨)と言われるデータを設定して、その骨をアニメーションさせて、その骨に関連づいている 3D モデルの頂点を動かすことでアニメーションを実現しています。



15.2 アニメーション付き X ファイル

X ファイルはモデルデータだけではなくアニメーションデータを含めることもできます。

GamePG_1/AnimationDemo/Game/Game/Assets/modelData/Player.X をテキストエディタで開いてください。開いたらエディタの検索機能を使って **AnimationSet** という単語を検索してください。すると下記のような記述が見つかります。

```
AnimationSet death {  
  
    Animation Anim {  
  
        { thief_thief }  
  
        AnimationKey rot {  
            0;  
            1;  
            0;4:-0.707107,0.707107,0.000000,0.000000;;  
        }  
  
        AnimationKey scale {  
            1;  
            1;  
            0;3:0.393701,0.393701,0.393701;;  
        }  
        以下略
```

これが X ファイルに付加されているアニメーションデータです。

15.3 CAnimation

tkEngine では CAnimation を使用することで、3D モデルにアニメーションを流すことができます。では使い方を見ていきましょう。AnimationDemo/Game/Game/ Player.cpp と Player.hを開いてください。

まず、Player クラスに CAnimation のメンバ変数を保持させます。

Player.h

```
class Player : public IGameObject
{
public:
    //ここからメンバ関数。
    Player();
    ~Player();
    void Start();
    void Update();
    void Render(CRenderContext& renderContext);
    //ここからメンバ変数。
    CSkinModel      skinModel;           //スキンモデル。
    CSkinModelData   skinModelData;       //スキンモデルデータ。
    CAnimation       animation;           //アニメーション。
    CVector3         position = CVector3::Zero; //座標。
    int              currentAnimationNo = 0; //現在のアニメーション番号。
};
```

そして、CSkinModelData::LoadModelData の第二引数に CAnimation のインスタンスのアドレスを渡して、アニメーションの初期化を行います。

Player.cpp

```
void Player::Start()
{
    skinModelData.LoadModelData("Assets/modelData/Player.X", &animation);
    skinModel.Init(&skinModelData);
    skinModel.SetLight(&_defaultLight); //デフォルトライトを設定。
    animation.PlayAnimation(currentAnimationNo); //アニメーションを再生。
}
```

これでアニメーションの初期化が完了です。アニメーションを再生する場合は再生したいアニメーションの番号を CAnimation::PlayAnimation に渡してください。

PlayAnimation を行っただけでは、アニメーションは再生されません。毎フレーム

CAnimation::Update を呼び出してください。

Player.cpp

```
void Player::Update()
{
    .
    .
}
```

```

    .
    //アニメーションを更新。
    animation.Update(1.0f/60.0f); //1/60 秒アニメーションを進める。
    //ワールド行列の更新。
    skinModel.Update(CVector3::Zero, CQuaternion::Identity, CVector3::One);
}

```

1 CAnimation::Update の第一引数はアニメーションを進める時間です。

3 15.4 アニメーション補間

4 アニメーション補間を行うとスムーズにアニメーションの切り替えを行うことができます。例えば、待機アニメーションから走りアニメーションに切り替えを行う場合、普通に
5 切り替えを行うとアニメーションがすっ飛ばすように切り替えが発生します。このアニメーションがすっ飛ばす現象を解決するものがアニメーション補間と言われるものです。待機ア
6 ニメーションと走りアニメーションをブレンドして、本来存在しない姿勢を動的に
7 作成する手法です。

11 待機アニメーション 歩きアニメーション ブレンドされたアニメーション



13 CAnimation::PlayAnimation の第二引数に補間時間が指定できます。例えば、第二引数に
14 0.3 を指定すると、0.3 秒かけてモーションが滑らかに切り替わります。

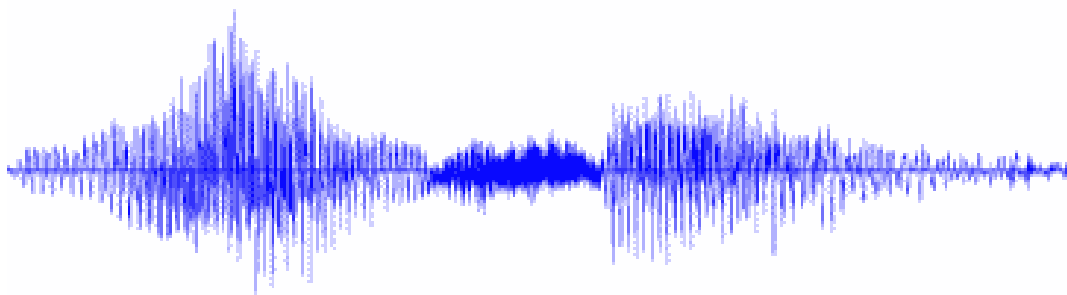
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

Chapter 16 サウンド

このチャプターでは GamePG_1/SoundDemo を使用して、サウンドを再生させるプログラムを勉強しましょう。

16.1 PCM データ

音は空気を振動して伝わってきます。その振動が鼓膜に伝わり鼓膜を揺らすことで私たちは音を認識することができます。下記のような図を見たことがあるのではないのでしょうか。



この波が音の振動です。コンピュータで音を再生するためには上記の波の振動のデジタル化を行う必要があります。このデジタル化したデータを PCM データ(波形データ)と呼ばれます。コンピュータはこのデータを流し込まれることによって、スピーカーを振動させて音を発生させているのです。

16.2 様々なサウンドファイル

サウンドファイルには色々なフォーマットがあります。有名どころでは wave、mp3、ogg などでしょうか。wave フォーマットは PCM データをそのまま保存しています。そのため音質の劣化がないため、品質が最も高いフォーマットです。続いて mp3、ogg といったフォーマットは PCM データに非可逆圧縮を行ったフォーマットです。そのため音質は wave フォーマットに比べると若干劣化していることになります。

16.3 CSoundSource

tkEngine では CSoundSource というクラスを使うことで、サウンドを再生することができます。サウンドソース(音源)という意味です。ではサンプルコードを使って CSoundSource の使い方を見ていきましょう。GamePG_1/SoundDemo/SoundDemo.cpp を開いてください。

CSoundSource は IGameObject を継承したクラスです。そのため NewGO でインスタンスを生成することができます。そしてインスタンスを生成したら CSoundSource::Init 関数に再生したい音楽データのファイルパスを指定して初期化を行ってください。tkEngi

ne のサウンドエンジンがサポートしているファイルフォーマットは **wave** だけですので注意してください。初期化が終わったら **CSoundSource::Play** 関数を使用して再生を開始してください。引数はループするかどうかのフラグです。true の場合音楽はループします。

SoundDemo.cpp

```
void SoundDemo::Start()
{
    //サウンドソースのインスタンスを生成して、ゲームオブジェクトマネージャーに登録する。
    bgmSource = NewGO<CSoundSource>(0);
    //BGM をロードして初期化。
    bgmSource->Init("Assets/sound/BGM. wav");
    //ループフラグを true にして再生。
    bgmSource->Play(true);
}
```

CSoundSource のインスタンスはループ再生しない場合は、再生が完了すると自動的に削除されます。ループ再生の場合は DeleteGO を使用して明示的に削除を行う必要があります。

SoundDemo.cpp

```
SoundDemo::~SoundDemo()
{
    if (bgmSource != NULL) {
        //NULL でなければサウンドソースのインスタンスが生成されているので削除する。
        DeleteGO(bgmSource);
    }
}
```