3D アクションゲームプログラミング

○評価要件

☑アニメーション処理
☑ワンショット再生＆ループ再生
☑ブレンド補完

○概要

今回はキャラクターモデルにアニメーション処理を実装します。
一般的なキャラクター操作をするゲームでは走ったり、剣を振ったりなどの動きがあります。
このような動作を「モーション」または「アニメーション」と呼びます。

このアニメーションの処理を実装するには3Dモデルを構成するボーン構造について理解する必要があります。

ボーン構造について理解し、今回はあらかじめ用意されているモデルクラスを拡張してアニメーションの再生処理を実装します。
モデルが保持しているアニメーションを自由に切り替えれるようにしましょう。

○3D モデルについて

今まで触れてきませんでしたが、3D モデルの内部構造について学習しましょう。
課題ではあらかじめモデルクラスが提供されており、これを使ってゲーム画面に 3D モデルを表示しています。

3D モデルデータは FBX や OBJ など様々なフォーマットがありますが、課題では FBX からゲームに必要なデータを抽出した独自形式のデータ(.mdl)を作成し、使用しています。
今回はこの独自形式のモデルデータを作成するツールを起動して 3D モデルのデータ構造について学習しましょう。

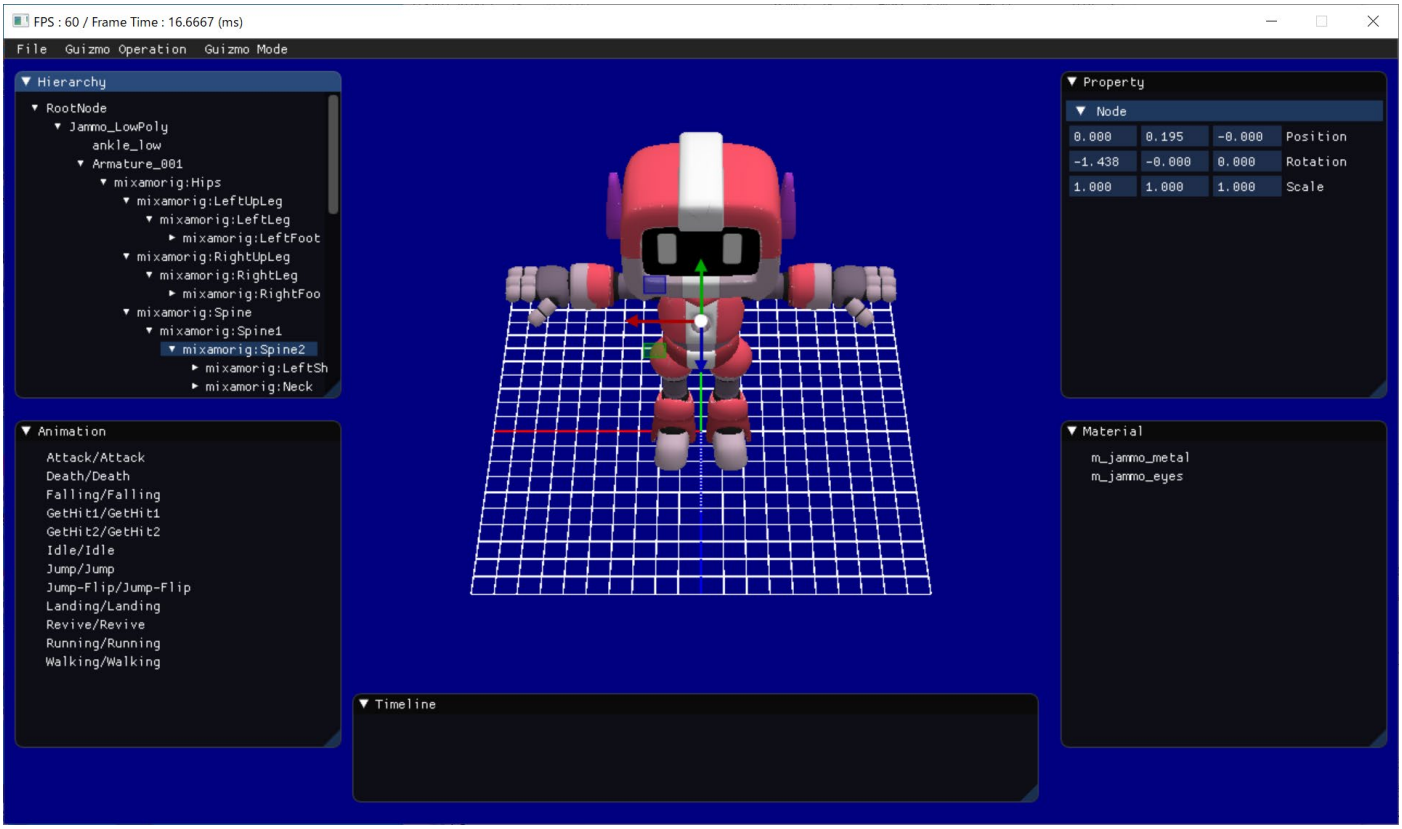
以下のディレクトリからモデルエディタを起動しましょう。

Tools/ModelEditor/ModelEditor.exe

モデルエディタを起動し、画面上部のメニューから「File」→「Open Model」を選択し、以下のモデルを読み込みましょう。

Data/Model/Jammo/Jammo.mdl

以下のような画面が表示されていれば OK です。

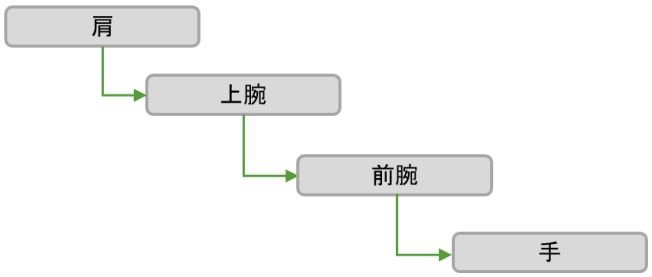


3D モデルは一般的にボーンという情報を持っています。
ボーンとは人体の骨格データを表したものであり、3D モデルも人間と同じように複数の骨が組み合わさって出来ています。

画面左上の「Hierarchy」ウインドウを見て下さい。
このウインドウには 3D モデルの骨格情報が木構造（ツリー）で表示されています。
この一つ一つをボーンと呼び、ボーンを動かすことでメッシュも動くように作られています。

○ボーンについて

ボーンには下図のように親子の関係があります。
親子関係の特性を自分の体を使って確かめてみましょう。
肩を動かすと、子供である上腕、前腕、手のすべてが動きます。
つまり、子は親の動きに追従するということです。

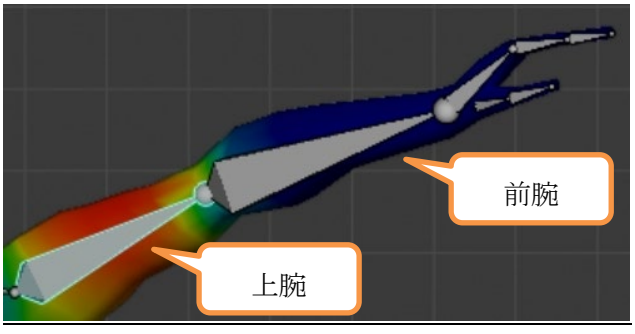


ボーンは今まで扱ってきたキャラクターの姿勢などと同じく、ボーン一本毎に「位置」「回転」「スケール」データを持っており、最終的に行列データを作成します。
今までと違うのは親子関係を表現するために子は親の動きを追従する必要があるということです。

○ワールド行列とローカル行列について

今までキャラクターが扱う行列のことをワールド行列と呼ぶことがありましたが、あれは「親がワールドの行列」と言い換えることができます。
「位置」「回転」「スケール」から作成された行列は「ローカル行列」と呼びます。
このローカル行列は親を基準とした位置関係のことを表しています。

例えば、前腕ボーンは上腕ボーンから見て、前方向に 30cm 先にあるという感じです。



○ローカル行列をワールド行列に変換

ローカル行列は親を基準として考えた時の姿勢データです。
先ほどの例でいくと前腕ボーンの位置が上腕ボーンからみて 30cm 前方向にあることがわかりましたが、前腕ボーンがワールド空間上のどこにあるのか知りたいことが多々あります。
ローカル行列からワールド行列を求めるには以下の計算で算出できます。

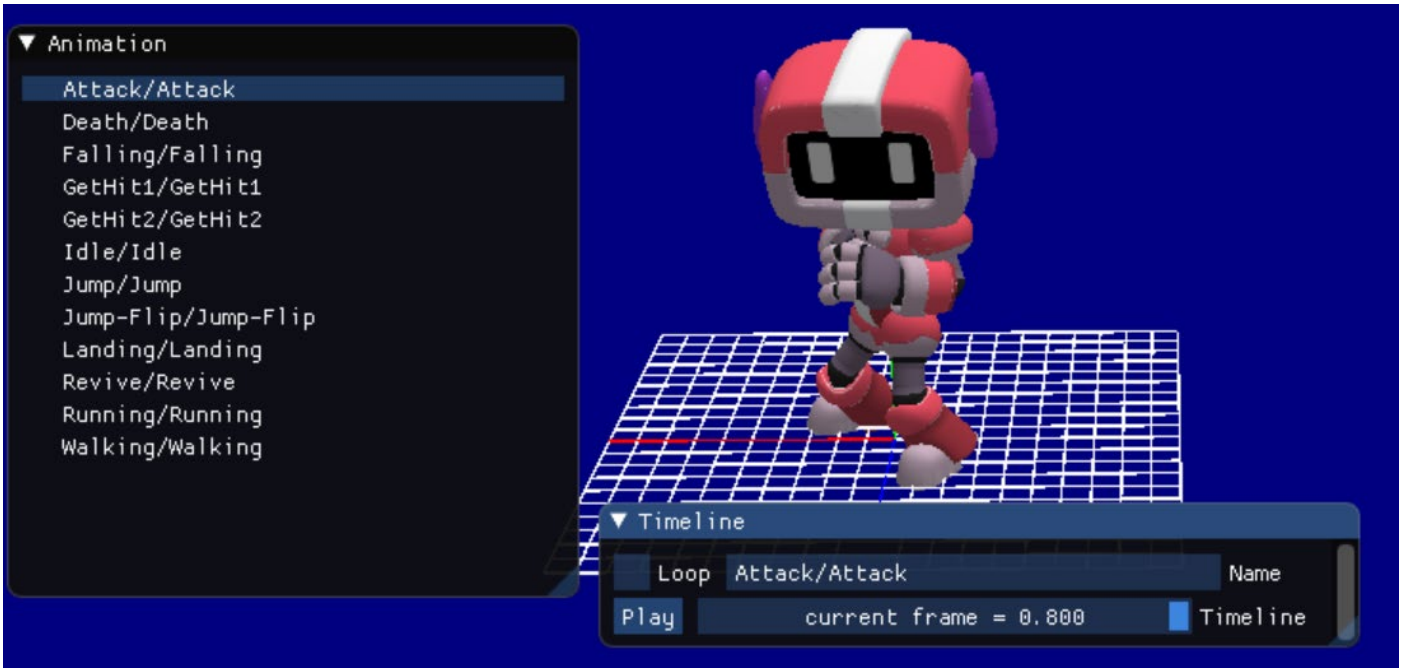
ワールド行列 ＝ ローカル行列　×　親のワールド行列

モデルクラスでは UpdateTransform()関数で実装されていますので確認しておいてください。

○アニメーションについて

モデルエディタの左下の「Animation」ウインドウを見て下さい。
このリストに表示されているものが、このモデルに内包されているアニメーションデータです。
適当なアニメーションを選択し、モデルエディタ下部の「Timeline」ウインドウで「Play」ボタンを押してアニメーションを再生してみましょう。

今回の課題では、このアニメーションを再生できるようにします。

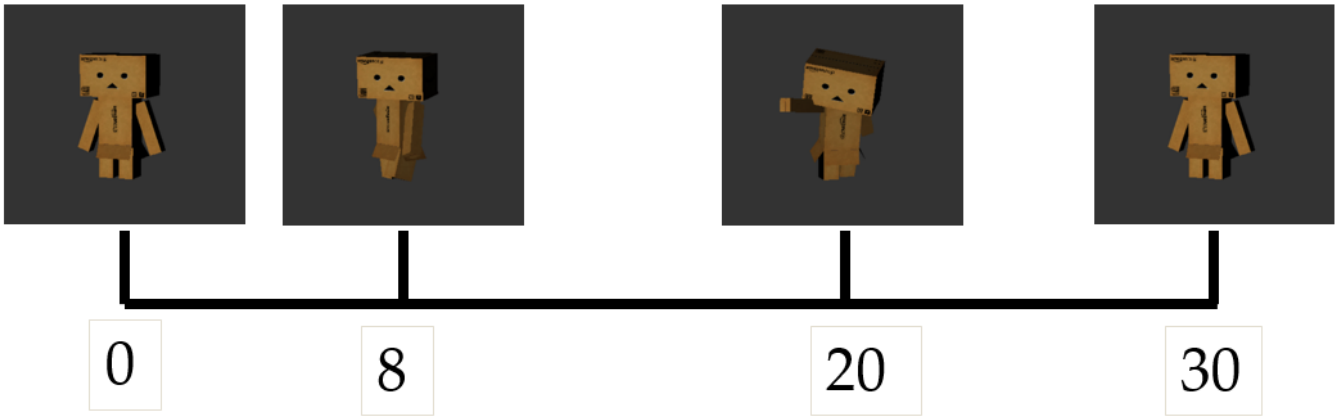


アニメーションシステムの設計については様々な考えがあり、実装者によってことなりますが、基本的な考えは同じだと思います。

テレビアニメやパラパラ漫画と同じく、「複数のポーズ」を連続で表示することによって動いているように見せています。

テレビアニメでは 1 秒間に 24 枚の絵を連続で表示して動かしているようです。

CG の世界でも同じように DCC ツール（Maya、Blender など）で表現したいポーズを作成しています。



テレビアニメやパラパラ漫画と違うのはポーズとポーズの間をプログラムで計算し、補完することができるという点です。

一般的なゲームは 60FPS、または 30FPS で動いています。
上図の例で考えると 14 フレーム目はどういうポーズになっているでしょうか。
このポーズを計算で算出するのが今回の課題です。

○アニメーションの計算

１つのアニメーションには特定のフレーム（時間）にポーズ（姿勢）データが存在します。このデータのことをキーフレームと呼びます。

ゲームでは 1 フレームずつ時間が進んでいきますが、アニメーションデータにはキーフレームが 1 フレーム毎に存在するとは限りません。

このとき、キーフレームが存在しないフレームのポーズ（姿勢）を計算で算出します。

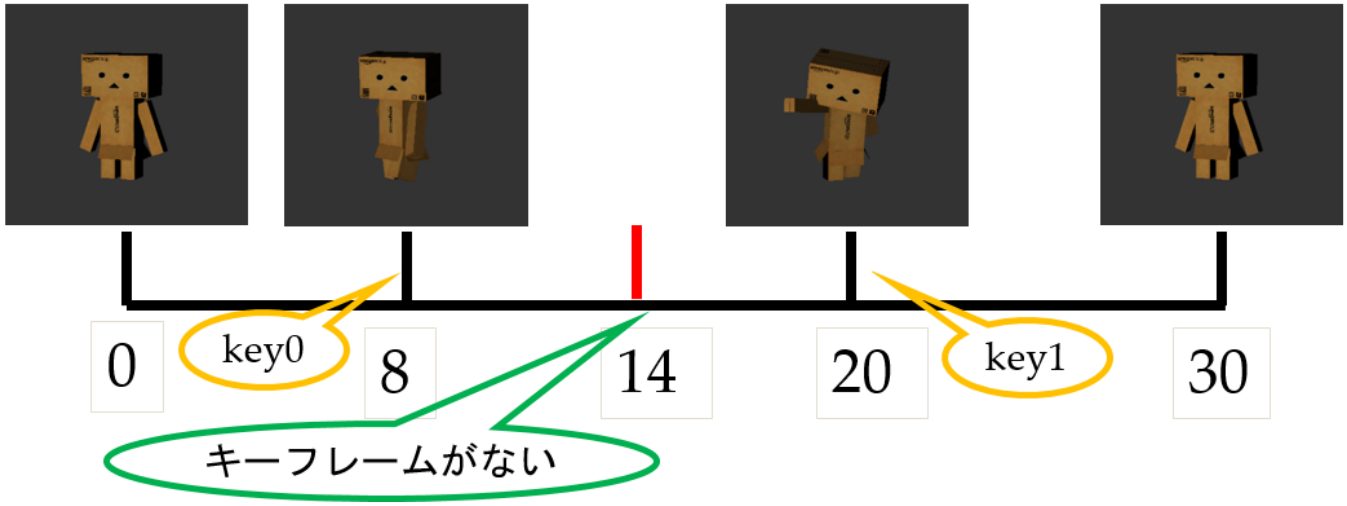
まず、ポーズ（姿勢）を計算するために必要なデータをおさらいしましょう。
姿勢は「行列」で表現されます。その「行列」は「位置」「回転」「スケール」の３つのデータから作成されます。

そして「回転」については今まで「オイラー」を使用してきましたが、今回は「クォータニオン」で行います。
「クォータニオン」はジンバルロックなどの弱点がなく、全ての回転を表現できるからです。

ポーズ（姿勢）の計算には「線形補完」と「球面線形補完」という計算法を使います。
どちらも考え方は同じで、２点間の値を係数（0.0～1.0)で求めるというものです。

例えば 10 と 20 という２つの値があった場合、係数が 0.3 だった場合の答えは 13 になります。

では下図の場合、14 フレーム目の姿勢を算出する計算を考えてみましょう。




```
float t = キーフレームの時間から割合を求める
DirectX::VECTOR Position = DirectX::XMVectorLerp(Key0_Position, Key1_Position, t);
DirectX::XMVECTOR Rotation = DirectX::XMQuaternionSlerp(Key0_Rotation, Key1_Rotation, t);
```

○アニメーション処理実装

アニメーションを計算する更新処理関数、アニメーションの再生命令関数、アニメーションが再生中かを確認する関数の３つを実装します。
Graphics/Model.h と Model.cpp を開き、下記プログラムコードを記述しましょう。

Model.h

```
---省略---

// モデル
class Model
{
public:
    ---省略---

    // アニメーション更新処理
    void UpdateAnimation(float elapsedTime);

    // アニメーション再生
    void PlayAnimation(int index);
```

```
    // アニメーション再生中か
    bool IsPlayAnimation() const;

private:
    ---省略---
    int     currentAnimationIndex = -1;
    float   currentAnimationSeconds = 0.0f;
};
```

Model.cpp

```
---省略---

// アニメーション更新処理
void Model::UpdateAnimation(float elapsedTime)
{
    // 再生中でないなら処理しない
    if (!IsPlayAnimation()) return;

    // 指定のアニメーションデータを取得
    const std::vector<ModelResource::Animation>& animations = resource->GetAnimations();
    const ModelResource::Animation& animation = animations.at(currentAnimationIndex);

    // アニメーションデータからキーフレームデータリストを取得
    const std::vector<ModelResource::Keyframe>& keyframes = animation.keyframes;
    int keyCount = static_cast<int>(keyframes.size());
    for (int keyIndex = 0; keyIndex < keyCount - 1; ++keyIndex)
    {
        // 現在の時間がどのキーフレームの間にいるか判定する
        const ModelResource::Keyframe& keyframe0 = keyframes.at(keyIndex);
        const ModelResource::Keyframe& keyframe1 = keyframes.at(keyIndex + 1);
        if (currentAnimationSeconds >= keyframe0.seconds &&
            currentAnimationSeconds < keyframe1.seconds)
        {
            // 再生時間とキーフレームの時間から補完率を算出する
            float rate = (currentAnimationSeconds - keyframe0.seconds) / (keyframe1.seconds
                                                                          - keyframe0.seconds);
            int nodeCount = static_cast<int>(nodes.size());
            for (int nodeIndex = 0; nodeIndex < nodeCount; ++nodeIndex)
            {
                // ２つのキーフレーム間の補完計算
                const ModelResource::NodeKeyData& key0 = keyframe0.nodeKeys.at(nodeIndex);
                const ModelResource::NodeKeyData& key1 = keyframe1.nodeKeys.at(nodeIndex);

                Node& node = nodes[nodeIndex];

                DirectX::XMVECTOR S0 = DirectX::XMLoadFloat3(&key0.scale);
                DirectX::XMVECTOR S1 = DirectX::XMLoadFloat3(&key1.scale);
                DirectX::XMVECTOR R0 = DirectX::XMLoadFloat4(&key0.rotate);
                DirectX::XMVECTOR R1 = DirectX::XMLoadFloat4(&key1.rotate);
                DirectX::XMVECTOR T0 = DirectX::XMLoadFloat3(&key0.translate);
                DirectX::XMVECTOR T1 = DirectX::XMLoadFloat3(&key1.translate);

                DirectX::XMVECTOR S = DirectX::XMVectorLerp(S0, S1, rate);
                DirectX::XMVECTOR R = DirectX::XMQuaternionSlerp(R0, R1, rate);
```

```
                DirectX::XMVECTOR T = DirectX::XMVectorLerp(T0, T1, rate);
                // 計算結果をボーンに格納
                DirectX:: XMStoreFloat3(&node.scale, S);
                DirectX::XMStoreFloat4(&node.rotate, R);
                DirectX::XMStoreFloat3(&node.translate, T);
            }
            break;
        }
    }

    // 時間経過
    currentAnimationSeconds += elapsedTime;

    // 再生時間が終端時間を超えたら
    if (currentAnimationSeconds >= animation.secondsLength)
    {
        // 再生時間を巻き戻す
        currentAnimationSeconds -= animation.secondsLength;
    }
}

// アニメーション再生
void Model::PlayAnimation(int index)
{
    currentAnimationIndex = index;
    currentAnimationSeconds = 0.0f;
}

// アニメーション再生中か
bool Model::IsPlayAnimation() const
{
    if (currentAnimationIndex < 0) return false;
    if (currentAnimationIndex >= resource->GetAnimations().size()) return false;
    return true;
}
```

Player.cpp

```
---省略---

// コンストラクタ
Player::Player()
{
    model = new Model("Data/Model/Mr.Incredible/Mr.Incredible.mdl");
    model = new Model("Data/Model/Jammo/Jammo.mdl");
    model->PlayAnimation(0);

    ---省略---
}

---省略---

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

```
    ---省略---

    // モデルアニメーション更新処理
    model->UpdateAnimation(elapsedTime);

    // モデル行列更新
    ---省略---
}

---省略---
```

実装出来たら実行確認をしてみましょう。
攻撃アニメーションを再生できていれば OK です。

○ワンショットアニメーション対応

現在の実装はアニメーションを再生するとアニメーション終了フレームを超えると巻き戻されてループ再生されます。
アニメーションを再生する際、ループ再生するかワンショット再生するかを選択できるようにしましょう。

Model.h

```
---省略---

// モデル
class Model
{
public:
    ---省略---

    // アニメーション再生
    void PlayAnimation(int index);
    void PlayAnimation(int index, bool loop);

private:
    ---省略---
    bool    animationLoopFlag = false;
    bool    animationEndFlag = false;
};
```

Model.cpp

```
---省略---

// アニメーション更新処理
void Model::UpdateAnimation(float elapsedTime)
{
    ---省略---

    // 最終フレーム処理
    if (animationEndFlag)
    {
```

```
        animationEndFlag = false;
        currentAnimationIndex = -1;
        return;
    }

    // 時間経過
    currentAnimationSeconds += elapsedTime;

    // 再生時間が終端時間を超えたら
    if (currentAnimationSeconds >= animation.secondsLength)
    {
        // 再生時間を巻き戻す
        currentAnimationSeconds -= animation.secondsLength;
        if (animationLoopFlag)
        {
            currentAnimationSeconds -= animation.secondsLength;
        }
        // 再生終了時間にする
        else
        {
            currentAnimationSeconds = animation.secondsLength;
            animationEndFlag = true;
        }
    }
}

// アニメーション再生
void Model::PlayAnimation(int index)
void Model::PlayAnimation(int index, bool loop)
{
    currentAnimationIndex = index;
    currentAnimationSeconds = 0.0f;
    animationLoopFlag = loop;
    animationEndFlag = false;
}
```

Player.cpp

```
---省略---

// コンストラクタ
Player::Player()
{
    ---省略---
    model->PlayAnimation(0);

    ---省略---
}

---省略---

// 更新処理
void Player::Update(float elapsedTime)
{
    // Bボタン押下でワンショットアニメーション再生
```

```
    GamePad& gamePad = Input::Instance().GetGamePad();
    if (gamePad.GetButtonDown() & GamePad::BTN_B)
    {
        model->PlayAnimation(0, false);
    }
    // ワンショットアニメーション再生が終わったらループアニメーション再生
    if (!model->IsPlayAnimation())
    {
        model->PlayAnimation(1, true);
    }

    ---省略---
}
---省略---
```

実装ができたら実行確認をしてみましょう。
B ボタンを押すと攻撃アニメーションが再生し、アニメーション終了後に死亡アニメーションに切り替わり、ループ再生されていれば OK です。。

○ブレンド補完

基本的なアニメーション再生までの実装が出来た人はアニメーション切り替え時の補完にも挑戦してみましょう。
現状ではアニメーション切り替え時に、次のアニメーションに一瞬で切り替わってしまうため、違和感があるので、アニメーション切り替え時に一定時間は現在の姿勢から次のアニメーションの姿勢に滑らかに補完するように考えてみましょう。

Model.h

```
---省略---

// モデル
class Model
{
public:
    ---省略---

    // アニメーション再生
    void PlayAnimation(int index, bool loop);
    void PlayAnimation(int index, bool loop, float blendSeconds = 0.2f);

private:
    ---省略---
    float   animationBlendTime = 0.0f;
    float   animationBlendSeconds = 0.0f;
};
```

Model.cpp

```
---省略---
```

```
// アニメーション更新処理
void Model::UpdateAnimation(float elapsedTime)
{
    // 再生中でないなら処理しない
    ---省略---

    // ブレンド率の計算
    float blendRate = 1.0f;
    if (animationBlendTime < animationBlendSeconds)
    {
        animationBlendTime += elapsedTime;
        if (animationBlendTime >= animationBlendSeconds)
        {
            animationBlendTime = animationBlendSeconds;
        }
        blendRate = animationBlendTime / animationBlendSeconds;
        blendRate *= blendRate;
    }

    // 指定のアニメーションデータを取得
    ---省略---

    // アニメーションデータからキーフレームデータリストを取得
    ---省略---
    for (int keyIndex = 0; keyIndex < keyCount - 1; ++keyIndex)
    {
        // 現在の時間がどのキーフレームの間にいるか判定する
        ---省略---
        if (currentAnimationSeconds >= keyframe0.seconds &&
            currentAnimationSeconds <= keyframe1.seconds)
        {
            // 再生時間とキーフレームの時間から補完率を算出する
            ---省略---
            for (int nodeIndex = 0; nodeIndex < nodeCount; ++nodeIndex)
            {
                ---省略---

                // ブレンド補完処理
                if (blendRate < 1.0f)
                {
                    // 現在の姿勢と次のキーフレームとの姿勢の補完
                    DirectX::XMVECTOR S0 = DirectX::XMLoadFloat3(&node.scale);
                    DirectX::XMVECTOR S1 = DirectX::XMLoadFloat3(&key1.scale);
                    DirectX::XMVECTOR R0 = DirectX::XMLoadFloat4(&node.rotate);
                    DirectX::XMVECTOR R1 = DirectX::XMLoadFloat4(&key1.rotate);
                    DirectX::XMVECTOR T0 = DirectX::XMLoadFloat3(&node.translate);
                    DirectX::XMVECTOR T1 = DirectX::XMLoadFloat3(&key1.translate);

                    DirectX::XMVECTOR S = DirectX::XMVectorLerp(S0, S1, blendRate);
                    DirectX::XMVECTOR R = DirectX::XMQuaternionSlerp(R0, R1, blendRate);
                    DirectX::XMVECTOR T = DirectX::XMVectorLerp(T0, T1, blendRate);
                    // 計算結果をボーンに格納
                    DirectX::XMStoreFloat3(&node.scale, S);
                    DirectX::XMStoreFloat4(&node.rotate, R);
                    DirectX::XMStoreFloat3(&node.translate, T);
                }
                // 通常の計算
```

```
                else
                {
                    // 前のキーフレームと次のキーフレームの姿勢を補完
                    DirectX::XMVECTOR S0 = DirectX::XMLoadFloat3(&key0.scale);
                    DirectX::XMVECTOR S1 = DirectX::XMLoadFloat3(&key1.scale);
                    DirectX::XMVECTOR R0 = DirectX::XMLoadFloat4(&key0.rotate);
                    DirectX::XMVECTOR R1 = DirectX::XMLoadFloat4(&key1.rotate);
                    DirectX::XMVECTOR T0 = DirectX::XMLoadFloat3(&key0.translate);
                    DirectX::XMVECTOR T1 = DirectX::XMLoadFloat3(&key1.translate);

                    DirectX::XMVECTOR S = DirectX::XMVectorLerp(S0, S1, rate);
                    DirectX::XMVECTOR R = DirectX::XMQuaternionSlerp(R0, R1, rate);
                    DirectX::XMVECTOR T = DirectX::XMVectorLerp(T0, T1, rate);

                    // 計算結果をボーンに格納
                    DirectX::XMStoreFloat3(&node.scale, S);
                    DirectX::XMStoreFloat4(&node.rotate, R);
                    DirectX::XMStoreFloat3(&node.translate, T);
                }
            }
            break;
        }
    }
    ---省略---
}

// アニメーション再生
~~void Model::PlayAnimation(int index, bool loop)~~
void Model::PlayAnimation(int index, bool loop, float blendSeconds)
{
    ---省略---
    animationBlendTime = 0.0f;
    animationBlendSeconds = blendSeconds;
}
```

```
---省略---

// 更新処理
void Player::Update(float elapsedTime)
{
    // Bボタン押下でワンショットアニメーション再生
    GamePad& gamePad = Input::Instance().GetGamePad();
    if (gamePad.GetButtonDown() & GamePad::BTN_B)
    {
        ~~model->PlayAnimation(0, false);~~
        model->PlayAnimation(0, false, 0.1f);
    }
    ---省略---
}
---省略---
```

実行確認をしてみましょう。
滑らかに補完できていたら OK です。
お疲れ様でした。