後期課題2

２日で理解する3Dプログラミング 「モデルデータ編」

■モデルデータの描画

３角形ポリゴンの集合体で3Dモデルとなっているデータを描画する為には、扱えるデータ形式に決まりがあります。
DxLib で使用できるデータ形式は下記の通りです。

・ｍｖ１形式	・・・	DxLib 専用のモデル。 ※アニメーションデータあり （FBX 形式を専用のツールでｍｖ１形式に変換する）
・ｘ形式	・・・	DirectX 形式。※アニメーションデータあり
・ｍｑｏ形式	・・・	メタセコイア形式。 ※アニメーションなし
・ｐｍｄ形式	・・・	MikuMikuDance 形式。 ※アニメーションあり
・ｐｍｘ形式	・・・	新MikuMikuDance 形式。 ※アニメーションあり

形式は色々ありますが、下記手順でデータを扱うと上手くいく確立が高く、アニメーションにも対応しています。
①「Blender」でモデルを作成しFBX形式で保存する。
②専用ツール「DxLibModelViewer」でFBX形式→MV1形式に変換して保存する。
③DxLibで使用する。

■モデルデータ（アニメーションなし）の描画

モデルデータを描画する為には、プリミティブやポリゴン描画等とほぼ同様の考え方となります。
①3Dモデルを読み込む。 ※MV1LoadModel()関数で int 型の変数に ID を保存する。
②モデルの「回転」を設定。 ※MV1SetRotationXYZ()関数で回転値をセットする。
③モデルの「移動」を設定。 ※MV1SetPotition()関数で座標をセットする。
④モデルの「拡大縮小」を設定。 ※MV1SetScale()関数で拡大値をセットする。

３Ｄモデルの読み込み。
```
int MV1LoadModel(ファイル名);
```

ファイル名で指定した３Ｄモデルファイルを読み込んで、モデルハンドルとして
int 型の変数にＩＤを保存する。
※読み込み失敗は-1 が返ってくる。

モデルの拡大値をセットする。
int MV1SetScale(モデルのハンドル, 拡大値);

セットする拡大値・・・VECTOR 型　// 1.0f を 100%として%で値をセットする。

モデルの回転値をセットする。
int MV1SetRotationXYZ(モデルのハンドル, 回転値);

セットする回転値・・・VECTOR 型

VECTOR 構造体の各メンバ変数(x, y, z)の値はそれぞれ x 軸回転値、y 軸回転値、z 軸回転値を代入しておきます。 (回転値の単位はラジアン値)

モデルの座標をセットする。
int MV1SetPosition(モデルのハンドル, セットする座標);

セットする座標・・・VECTOR 型　// (x, y, z)でワールド座標を設定。

モデルを描画する。 ※半透明の部分を考慮する場合は「MV1DrawFrame」「MV1DrawMesh」を使う。
int MV1DrawModel(モデルのハンドル);

※描画する前に「拡大」「回転」「座標」をセットしておく。

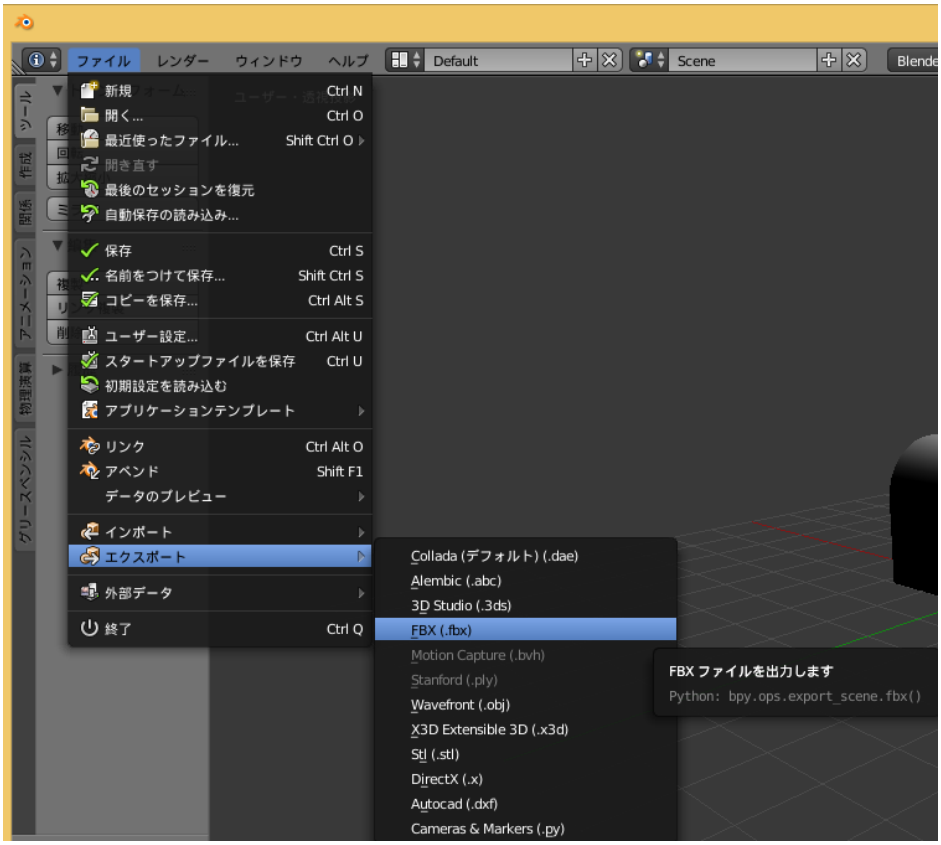
モデル読み込み→設定→描画のサンプル

```
// ----- 初期設定
int model = MV1LoadModel("model/Player01.mv1"); // 3Dﾓﾃﾞﾙ準備
VECTOR pos = VGet(0.0f, 0.0f, 0.0f);    // 座標
VECTOR rol = VGet(0.0f, 0.0f, 0.0f);    // 回転値
VECTOR scl = VGet(0.0f, 0.0f, 0.0f);    // 拡大値

// ----- 座標ｾｯﾄ(拡大→回転→移動の順に行う)
MV1SetScale(model, scl);          // 拡大値ｾｯﾄ
MV1SetRotationXYZ(model, rol);    // 回転値ｾｯﾄ
MV1SetPosition(model, pos);       // 座標ｾｯﾄ

// ----- 描画
MV1DrawModel(model);
```

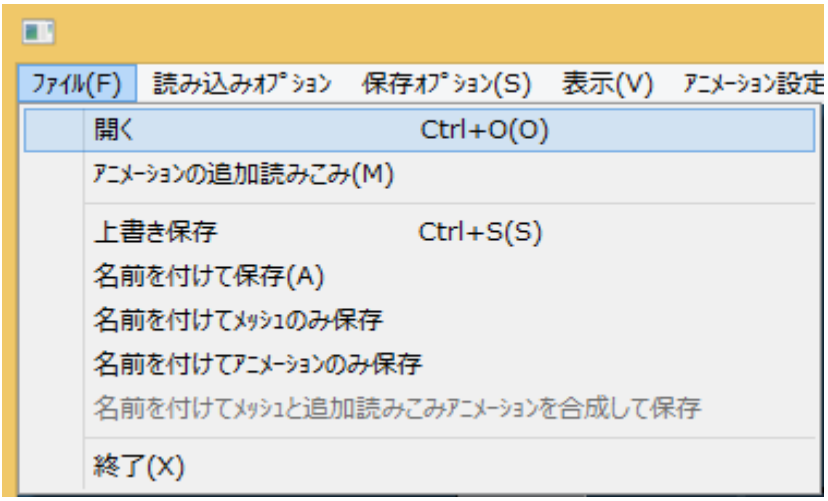
■MV1形式のモデルデータの準備

1)「Blender」からFBXファイルをエクスポートする。

ファイル→エクスポート→FBXを選んでFBX形式で 3D データを書き出す。

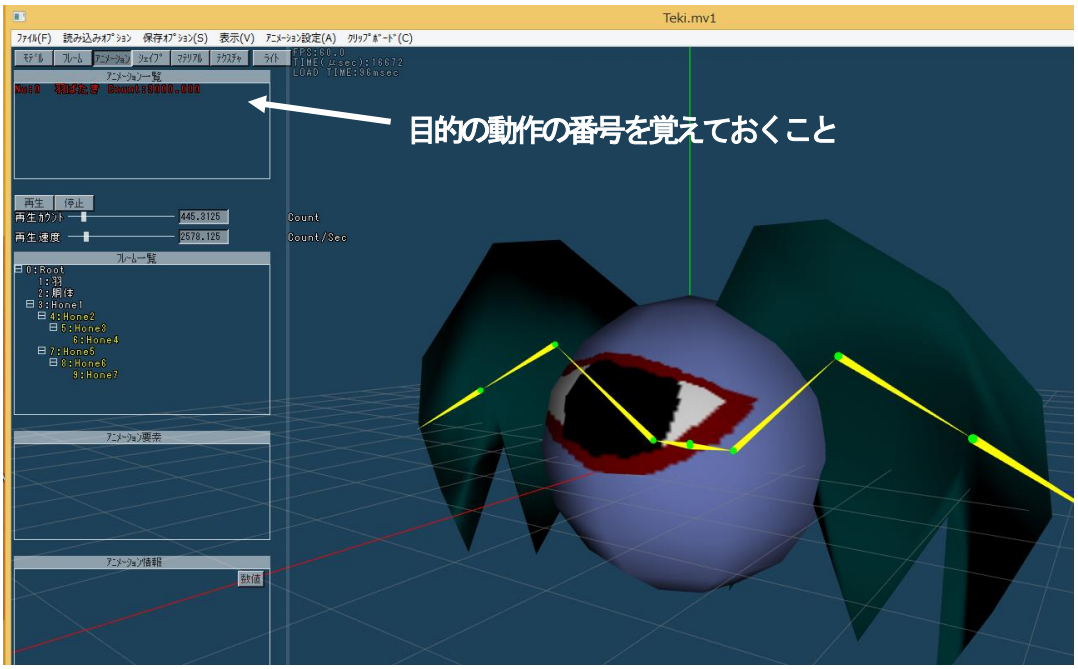


2)「DXLibModelViewer」でFBXファイルを開く。



3)アニメーションデータが入っている場合は動きが確認できる。

左上の「アニメーション」のタブをクリックして確認する。 ※ここでの番号が再生する時に必要！

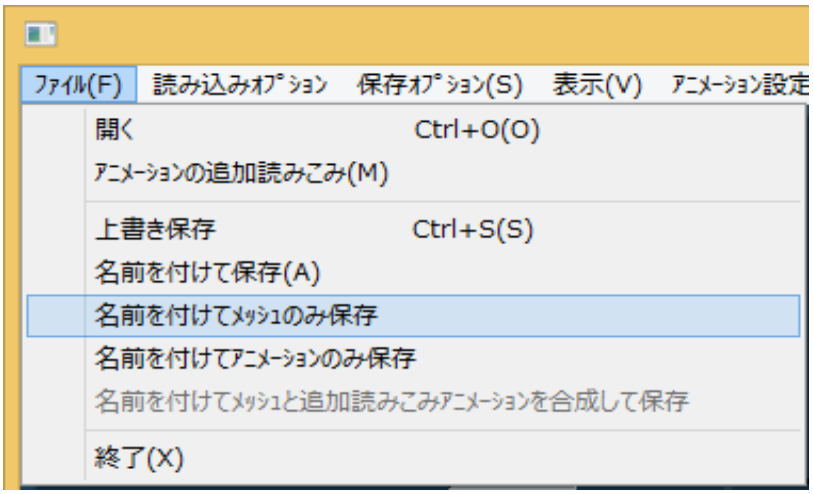


4)「DXLibModelViewer」からMV1形式で保存を行う。

保存は2つの方法があります。

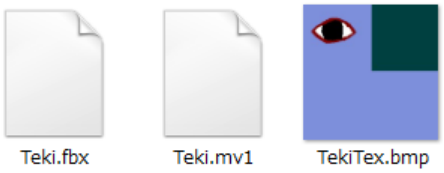
①「モデルデーター」と「モーションデータ」を同時に保存。

②「モーションデータ」のみ保存。



5)MV1形式のデータの完成！

MV1形式のモデルデータとテクスチャのデータがあるので全て同じフォルダで管理する事。



■モーション(アニメーション)付きモデルの描画

アニメーションの手順

モデル、モーションデータを読み込む ①モデルデータのみ ②モデル+モーションデータ ③モーションデータのみ ※モーションデータが複数ある場合は 複数用意する。	MV1LoadModel(モデルデータ)
モデルデータにアニメデータをセットする	MV1AttachAnim(モデル、アニメ)
アニメーションの総時間を計算する	MV1GetAttachAnimTotalTime()
アニメーションの時間を進める (どの時間帯のアニメーションを表示するか)	MV1SetAttachAnimTime()
モデルの表示	MV1DrawModel(モデルデータ)

モデルデータにアニメデータをセットする
int MV1AttachAnim(モデルハンドル, アニメ番号, アニメハンドル, アタッチするかのフラグ);

モデルハンドル・・・int model; ※読み込んだモデルハンドル

アニメ番号・・・・・int no; ※1 種類のアニメデータの場合は 0 を指定

アニメハンドル・・・int anim; ※読み込んだアニメハンドル
モデルハンドルと同じ場合は-1 を入れる。

フラグ・・・・・・・int flag; ※モデルとアニメのフレーム名が違う場合、
アタッチするかどうか。
(true:しない、false:アタッチする)

モデルデータの総時間を取得する
int MV1GetAttachAnimTotalTime(モデルハンドル, セットされたアニメ番号);

モデルハンドル・・・int model; ※読み込んだモデル
アニメハンドル・・・int no; ※MV1AttachAnim でセットしたアニメハンドル

アニメションの再生箇所をセットする
int MV1SetAttachAnimTime(モデルハンドル, セットされたアニメ番号, 再生箇所);

モデルハンドル・・・int model; ※読み込んだモデル
アニメハンドル・・・int no; ※MV1AttachAnim でセットしたアニメハンドル
再生箇所・・・・・・int time; ※再生する時間軸（再生箇所）

1)プログラムの例(初期化)

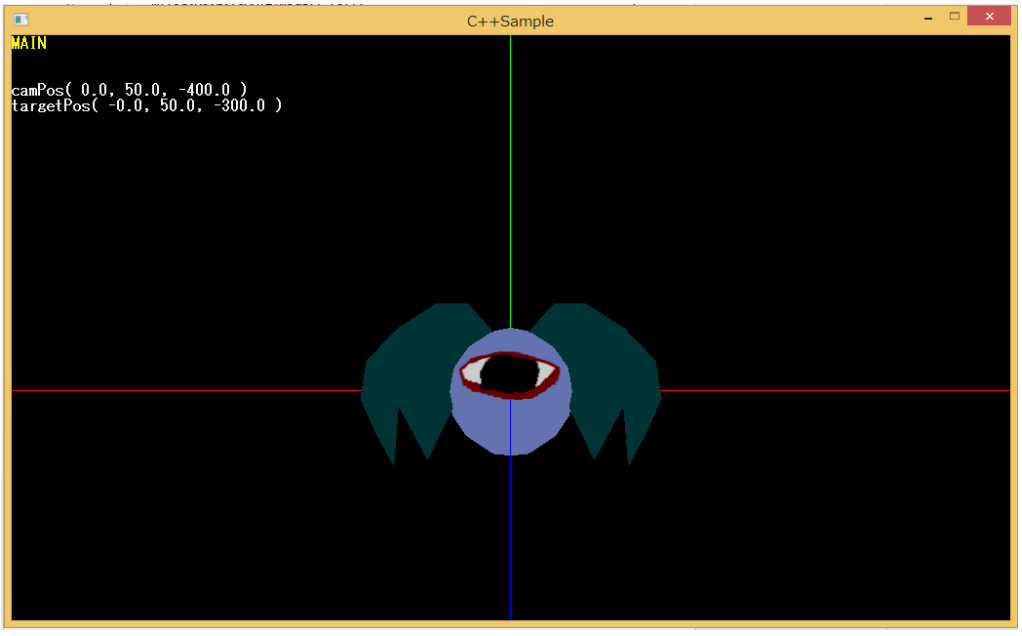
```
int model;       // ﾓﾃﾞﾙのID
VECTOR pos;      // ﾓﾃﾞﾙの座標
VECTOR rol;      // ﾓﾃﾞﾙの回転
VECTOR scl;      // ﾓﾃﾞﾙの拡大

int attachIndex;     // ﾓｰｼｮﾝを読み込んだｱﾆﾒﾊﾝﾄﾞﾙ
float totalTime;     // ｱﾆﾒﾃﾞｰﾀの総時間
float playTime;      // ｱﾆﾒﾃﾞｰﾀの再生箇所
```

← クラス変数で定義する

```
// ----- ﾓﾃﾞﾙ読み込み
model = MV1LoadModel("model/enemy/Teki.mv1");

// ----- 座標系初期化
pos = VGet((float)GetRand(600)-300, (float)GetRand(600)-300, 0.0f);
rol = VGet(0.0f, (DX_PI_F/180)*90, 0.0f);
scl = VGet(0.5f, 0.5f, 0.5f);

// ----- ﾓｰｼｮﾝ設定
// ①ﾓﾃﾞﾙﾃﾞｰﾀに入っているﾓｰｼｮﾝﾃﾞｰﾀをｾｯﾄしattachIndexに保存する
attachIndex = MV1AttachAnim(model, 0, -1, false);

// ②attachIndexに保存されたﾓｰｼｮﾝの総時間を取得
totalTime = MV1GetAttachAnimTotalTime(model, attachIndex);

// ③再生する時間をｾｯﾄ(通常は0から再生させる)
playTime = 0;
```

2)プログラムの例(更新・描画)

```
void Enemy::Update()
{
    // ----- 座標更新
    MV1SetPosition(model, pos);
    MV1SetRotationXYZ(model, rol);
    MV1SetScale(model, scl);

    // ----- 再生箇所を設定
    playTime += 100.0f;

    // 再生時間がｱﾆﾒｰｼｮﾝの総再生時間に達したら再生時間を０に戻す
    if (playTime >= totalTime){
        playTime = 0.0f;
    }
}

void Enemy::Render()
{
    // 再生時間をセットする
    MV1SetAttachAnimTime(model, attachIndex, playTime);

    // 描画
    MV1DrawModel(model);
}
```



■同じモデルを複数使用する場合の注意点 ※重要！

ザコ敵など、見た目が同じものが大量に登場するキャラクターがいる場合など、その数だけ MV1LoadModel をしてしまうと同じ 3D モデルデータをメモリ上に読み込む事になり効率が悪くなります（読み込み時間とメモリを多く使用）。その様な場合には、キャラクターモデル一つにつき一回だけ MV1LoadModel で読み込んで、MV1DuplicateModel を使用して基礎データを元にした分身データを利用して処理の簡略化をします。

注意点としては、2D 画像の読み込みの場合に一度読み込んだ画像を使いまわす（シングルトンで処理した方法）やり方とはイメージが違って、あくまでも分身を作って処理を行うという事です。2D 画像の時の様なやり方でも良いのですが、3D モデルの場合は保持している状態情報が多い為、分身を作る方が速度的に有利になります。

尚、作成（分身）されるモデルハンドルには、座標系やアタッチしたモーションデータなどは継承されません。

```
指定したモデルと同じ分身モデルを作成する
int MV1DuplicateModel( 一度読み込んだモデルハンドル );

一度読み込んだモデルハンドル・・・int model; ※読み込んだモデル
```

1)プログラム例（初期化） 基礎モデルハンドルをコンストラクタで渡してオブジェクト生成を行う。

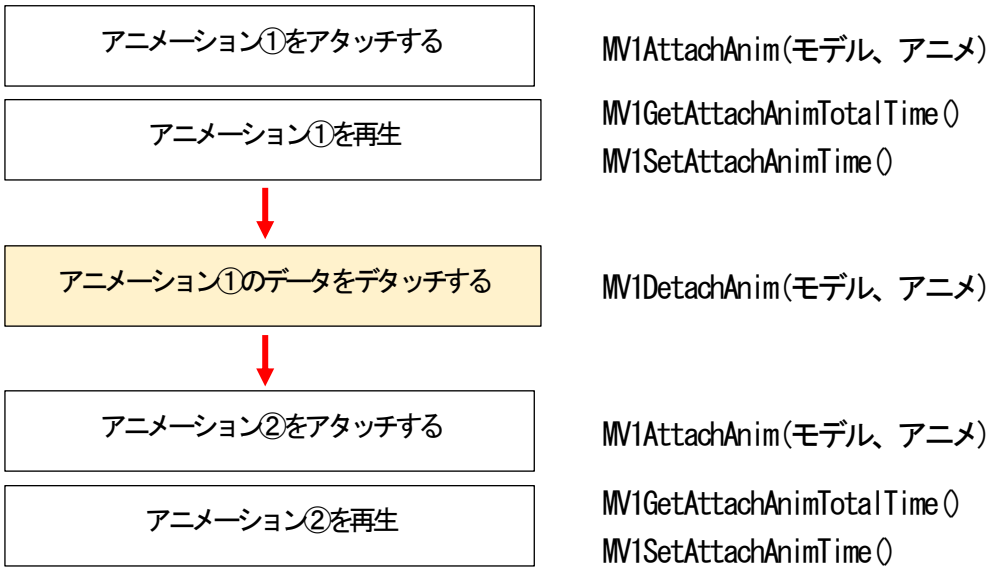
```
// ----- 基礎ﾓﾃﾞﾙを読み込む
int model = MV1LoadModel("model/enemy/Teki.mv1");

// ----- 基礎ﾓﾃﾞﾙﾃﾞｰﾀを利用して複数のｵﾌﾞｼﾞｪｸﾄを生成する
for(int i = 0; i < 20; i++){
    enemy[i] = new Enemy( model );  // ﾓﾃﾞﾙﾊﾝﾄﾞﾙを代入
}
```

2)プログラム例（生成時） 受け取った基礎モデルのハンドルを元に分身としてオブジェクトを生成する。

```
Enemy::Enemy(int index)
{
    Init(index);
}

Enemy::~Enemy()
{
}

void Enemy::Init(int index)
{
    // ----- ﾓﾃﾞﾙ読み込み
    //model = MV1LoadModel("model/enemy/Teki.mv1");
    model = MV1DuplicateModel(index);   // 既存のﾓﾃﾞﾙﾊﾝﾄﾞﾙより分身を作成
```

■モーションパターンが複数あって切り替えをしたい場合

モーションを切り替える場合は、アタッチしているモーションをデタッチ(外す)し、その後別のモーションをアタッチします。

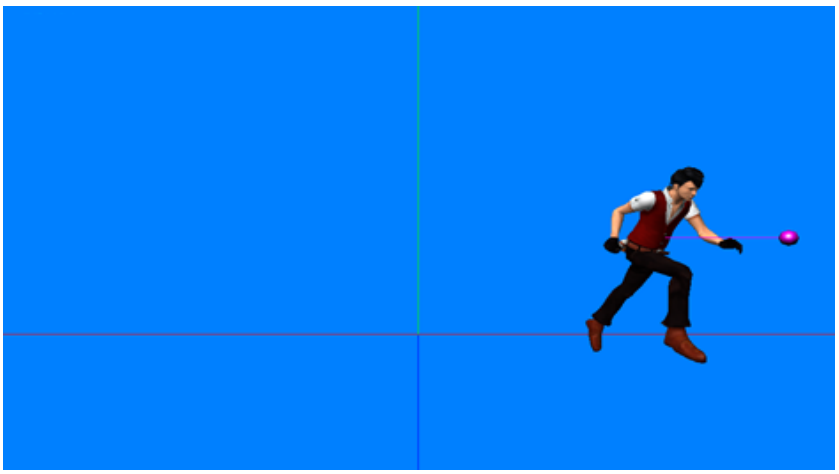


1)プログラム例(初期化)　モーションデータのハンドルを複数取得する。

```
void Enemy::Render(int index)
{
    model = MV1DuplicateModel(index);    // 既存のﾓﾃﾞﾙﾊﾝﾄﾞﾙより分身を作成

    // ----- 座標系初期化
    pos = VGet(0.0f, 0.0f, 0.0f);
    rol = VGet(0.0f, (DX_PI_F / 180) * 90, 0.0f);
    scl = VGet(0.5f, 0.5f, 0.5f);

    // ----- ﾓｰｼｮﾝ設定
    // ①ﾓﾃﾞﾙﾃﾞｰﾀに入っているﾓｰｼｮﾝﾃﾞｰﾀをｾｯﾄしplayerAttachに保存する
    animNeutral = MV1LoadModel("model/anim_Neutral.mv1");
    animRun = MV1LoadModel("model/anim_Run.mv1");
    animAttack = MV1LoadModel("model/anim_Attack.mv1");

    playerAttach = MV1AttachAnim(model, 0, animNeutral);

    // ②attachIndexに保存されたﾓｰｼｮﾝの総時間を取得
    totalTime = MV1GetAttachAnimTotalTime(model, playerAttach);

    // ③再生する時間をｾｯﾄ(通常は0から再生させる)
    playTime = 0;
}
```

2)プログラム例(実行時)　切り替えたいタイミングでデタッチしてアタッチし直す。

```
void Enemy::Update()
{
    // ----- 座標更新
    MV1SetPosition(model, pos);
    MV1SetRotationXYZ(model, rol);
    MV1SetScale(model, scl);

    // ----- 切り替えたい時

    // ※「走り」にしたい時
    MV1DetachAnim(model, playerAttach);
    playerAttach = MV1AttachAnim(model, 0, animRun);
    totalTime = MV1GetAttachAnimTotalTime(model, playerAttach);
    playTime = 0;

    // ※「Attack」にしたい時
    MV1DetachAnim(model, playerAttach);
    playerAttach = MV1AttachAnim(model, 0, animAttack);
    totalTime = MV1GetAttachAnimTotalTime(model, playerAttach);
    playTime = 0;

    // ----- 再生箇所を設定
    playTime += 100.0f;

    // 再生時間がｱﾆﾒｰｼｮﾝの総再生時間に達したら再生時間を０に戻す
    if (playTime >= totalTime){
        playTime = 0.0f;
    }
}
```



■アニメーションの移動を打ち消す！

モデルに登録されているモーションデータ自体にモデルの移動が含まれている場合があります。
例えば、歩くアニメーションの場合に実際に移動しながら歩く様な設定がされていた場合、その場足踏みをしたい場合でも前に進んだ様になってしまいます。
これば、実際のモデルの移動処理とアニメーション内での移動が重複してしまう事で発生する問題です。
そういう場合にモーションデータ内の移動を打ち消す事で、動作だけを再生する事を達成します。

指定したフレームをモデルの中から検索する（移動情報が入っているフレームを探す）
int MV1SearchFrame(モデル, フレーム名);

モデル・・・・・int model; ※モデルハンドル
フレーム名・・・char* name; ※フレーム名 普通は”root”

指定したモデルに指定した変換行列を設定する
int MV1SetFrameUserLocalMatrix(モデル, フレーム名, 行列);

モデル・・・・・int model; ※モデルハンドル
フレーム名・・・char* name; ※フレーム名 普通は”root”
行列・・・・・・MATRIX matrix; ※単位行列【MGetIdent()】

1)プログラム例(実行時) モーションをアタッチした後に移動量を打ち消す設定を行う。

```
// 移動量を打ち消す
playerRootflame = MV1SearchFrame(playerModel1, "root");
MV1SetFrameUserLocalMatrix(playerModel1, playerRootflame, MGetIdent());
```

□バックグラウンド色の変え方

バックグラウンド色の設定
SetBackgroundColor(赤, 緑, 青);