ゲームプログラミング応用Ⅰ／ゲームプログラミング実践Ⅰ

20 日で理解する3Dプログラミング 「その 4:モデル」

■はじめに(ライセンスの問題)

3Dプログラミングで重要になるのが3DCGモデルの素材の準備です。動かないモデルはもちろん、アニメーション等を想定していくと、要求を果たすものを準備するのは結構大変です。最終的には自前で作成するか、CGアーティストにお願いする事になりますが、当面は要求を満たすフリー素材(有償でも可)を探して実装していきましょう。

という事で、比較的良さそうな素材データとして下記のフリー素材を使用させて頂く事にしました。

https://zunko.jp/

■利用の範囲

オオカミ姫　東北ずん子コラボ3Dモデルは非商用利用に限り利用が許諾されます。

＜利用NGの例＞

・同人などでも販売を伴う場合

・企業が利用する場合(東北企業も含む)

2017.10.12 ～ 2017/10/26 の期間「オオカミ姫と東北ずん子コラボキャンペーン」のゲーム内で利用された3Dモデルを、特別にヤマハミュージックエンタテイメントホールディングス社が非商用利用限定で3Dモデルの利用をさせていただけることになったものです。

Zunko　Itako　Chanko　Kiritan　Zundamon

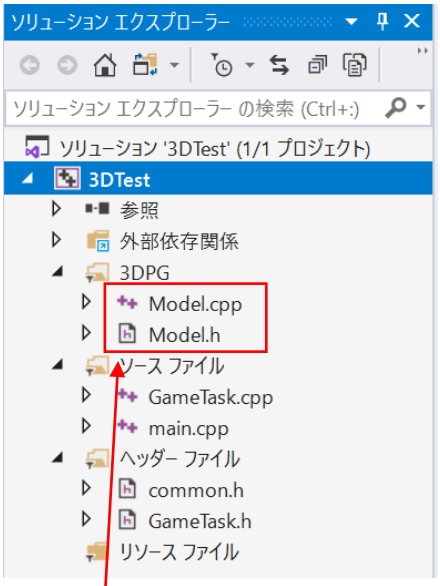
※自前で準備した素材を使用して課題を進める事は全く問題ありません。積極的に「探す」「作る」していきましょう。

■プロジェクトの準備

モデル制御用のクラスを準備して GameTask から呼び出しておきましょう。

Model.h

```
#pragma once

class Model
{
private:
    // モデルデータ用
    int model;
    VECTOR pos;
    VECTOR rol;
    VECTOR scl;

public:
    Model();
    ~Model();
    void Init();
    void Update();
    void Render();
};
```



「Model.cpp」「Model.h」
をプロジェクトに追加(新規作成)

Model.cpp

```
#include "DxLib.h"
#include "Model.h"

Model::Model()
{
}

Model::~Model()
{
}

void Model::Init()
{
}

void Model::Update()
{
}

void Model::Render()
{
}
```

GameTask.h

```
#pragma once

class Model;

class GameTask
{
public:
    Model* model;

public:
    GameTask();
    ~GameTask();
    int SystemInit();
    void GameMainLoop();
};
```

クラス名の宣言

Model クラスのポインタ変数を確保

GameTask.cpp

```
#include "DxLib.h"
#include "common.h"
#include "GameTask.h"
#include "Model.h"

// ----- コンストラクタ
GameTask::GameTask()
{
    SystemInit();    // 最初の初期化

    model = new Model();
}

// ----- デストラクタ
GameTask::~GameTask()
{
    delete model;
    model = nullptr;
}
```

Model.h をインクルード

Model をインスタンス

Model を解放

■モデルデータの描画

DirectXやUnityなどで3Dモデル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。
反対に、モデル表示はできてもアニメーションが上手く再生されないなどの不具合のあるデータが多く存在します。
①「Blender」でモデル・モーションデータを作成し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。

■モデルデータの描画（アニメーションなし）

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。
②モデルの「拡大縮小」を設定。
③モデルの「回転」を設定。
④モデルの「移動（座標）」を設定。

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);

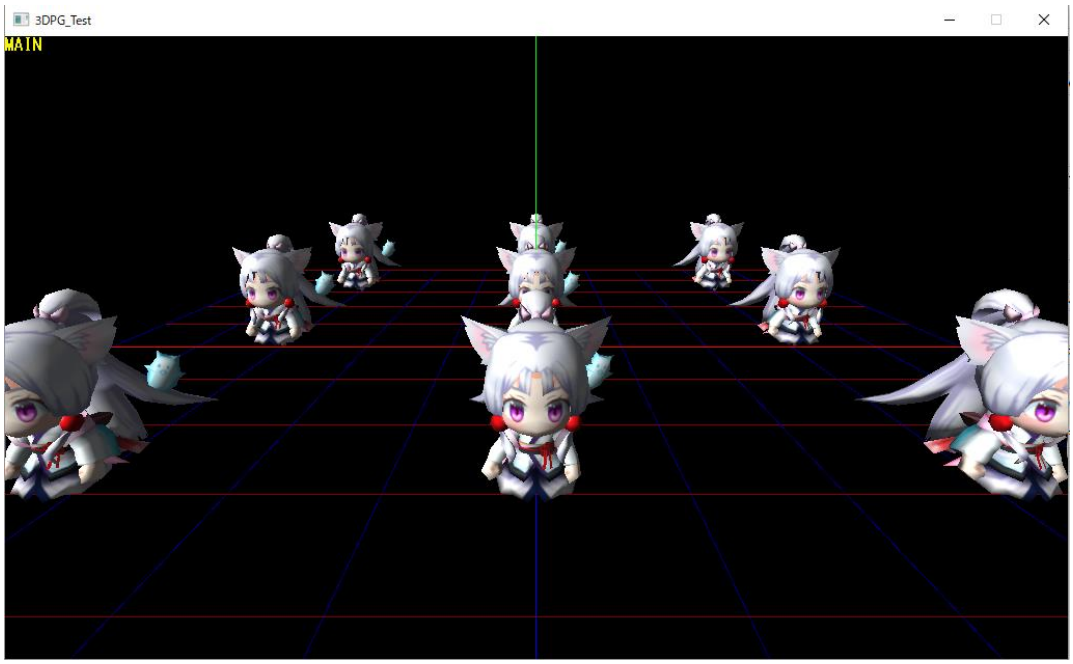
セットする座標・・・VECTOR 型　// (x, y, z)でワールド座標を設定。

モデル読み込み→設定→描画のサンプル

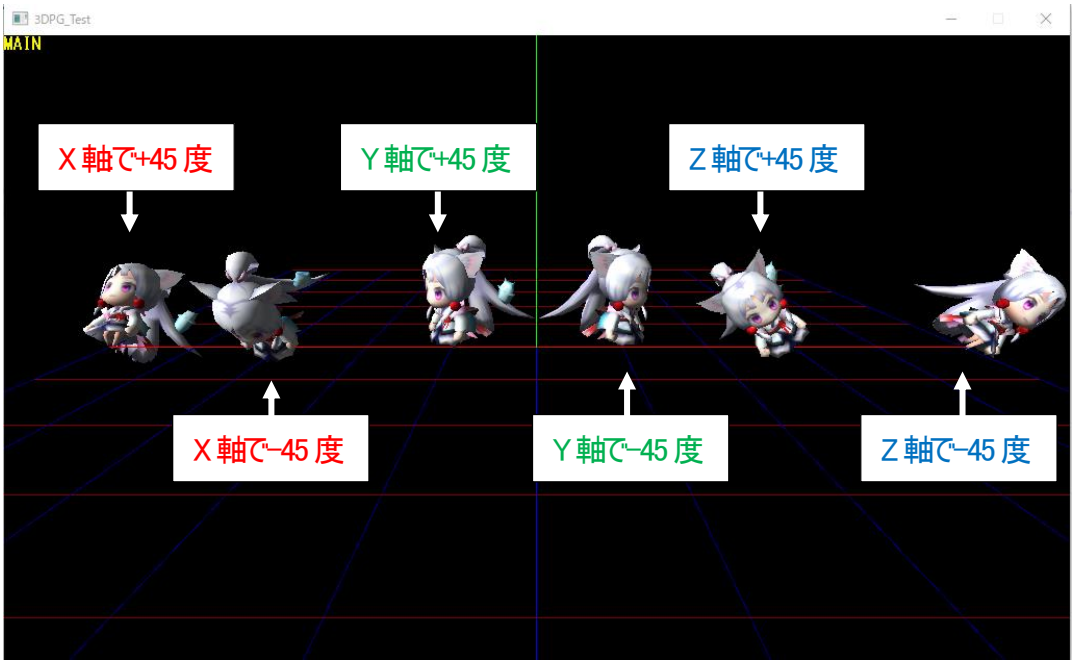
```
// ①モデルデータの読み込み
model = MV1LoadModel("model/itako/itako.mv1");

// 基本パラメータ
pos = VGet(0.0f, 0.0f, 0.0f);
rol = VGet(0.0f, 0.0f, 0.0f);
scl = VGet(1.0f, 1.0f, 1.0f);

// 「拡大縮小」「回転」「移動」の設定
MV1SetScale(model, scl);          // ②拡大縮小
MV1SetRotationXYZ(model, rol);    // ③回転
MV1SetPosition(model, pos);       // ④移動

// ⑤モデルの描画
MV1DrawModel(model);
```

制御の順番は、制御の都合で
「拡大縮小」→「回転」→「移動(座標」」
の順番にする事！

座標指定の例(MV1SetPosition()での指定)

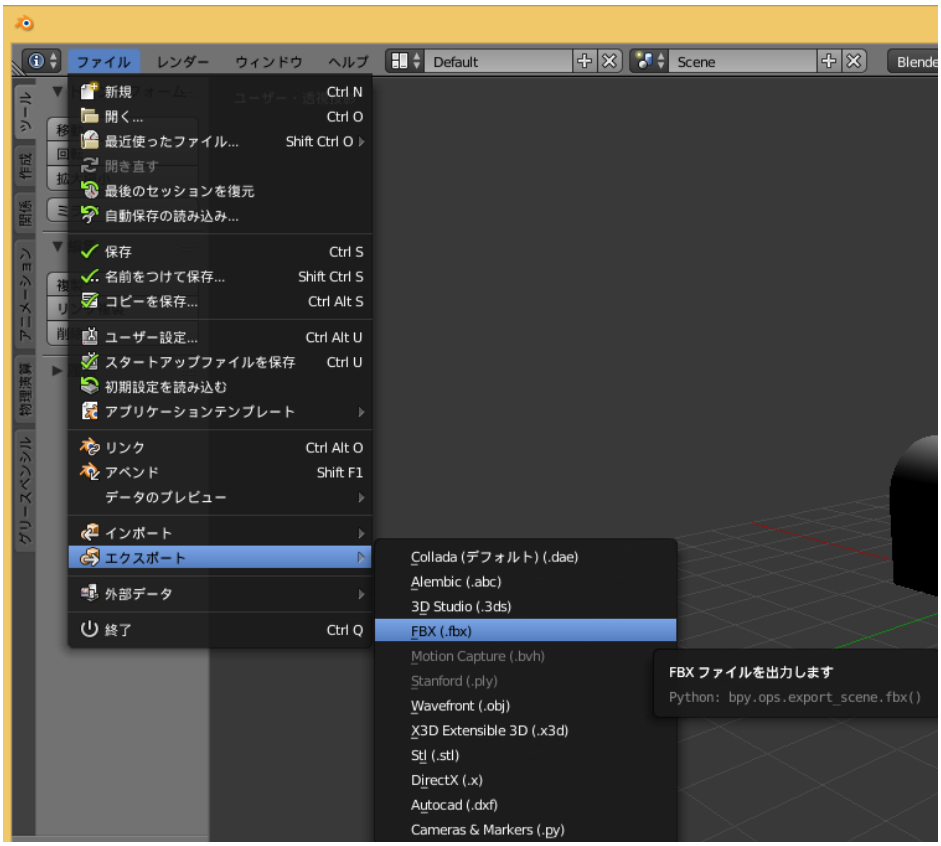


XYZ軸での回転の例(MV1SetRotationXYZ()での指定)

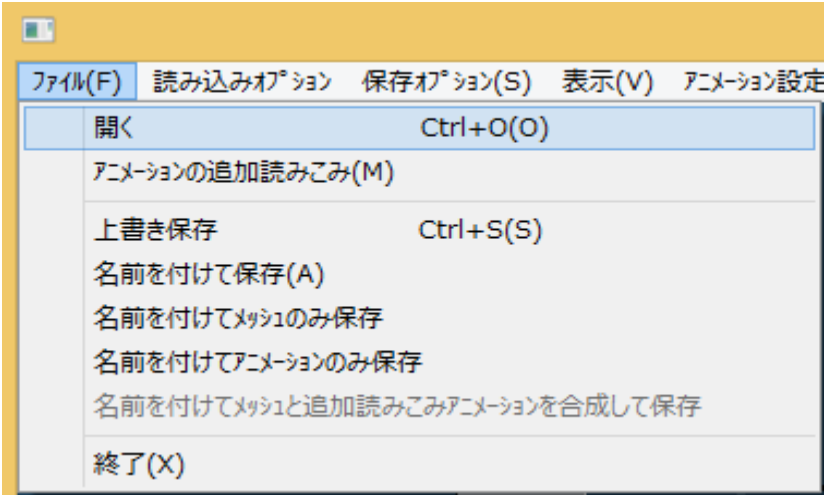


■MV1形式のモデルデータの準備方法

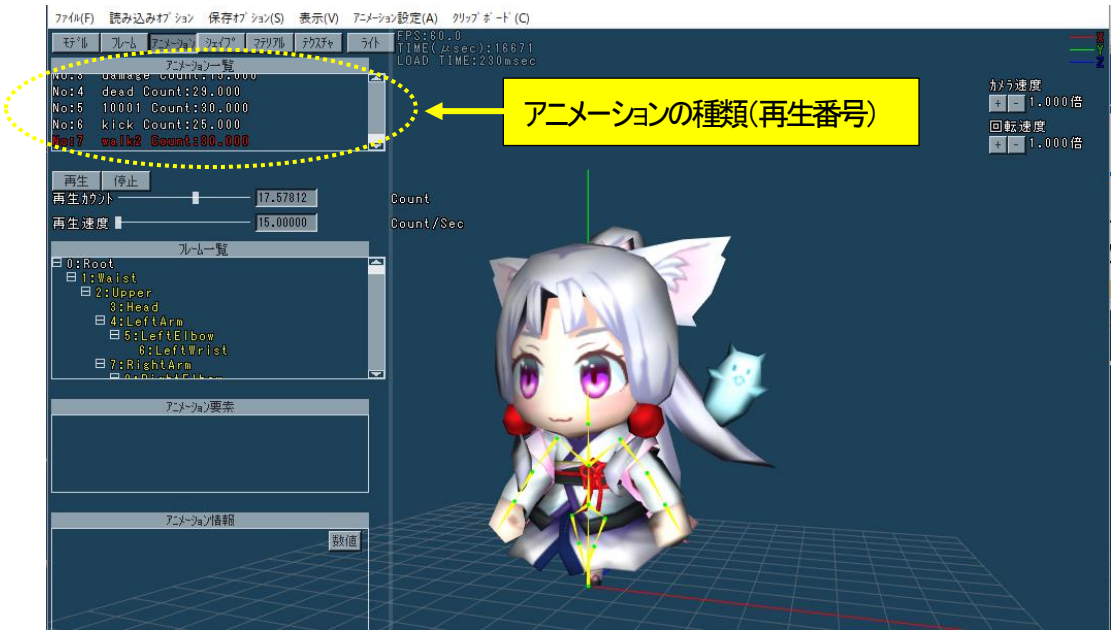
1)「Blender」からFBXファイルをエクスポートする。
ファイル→エクスポート→FBXを選んでFBX形式で 3D データを書き出す。



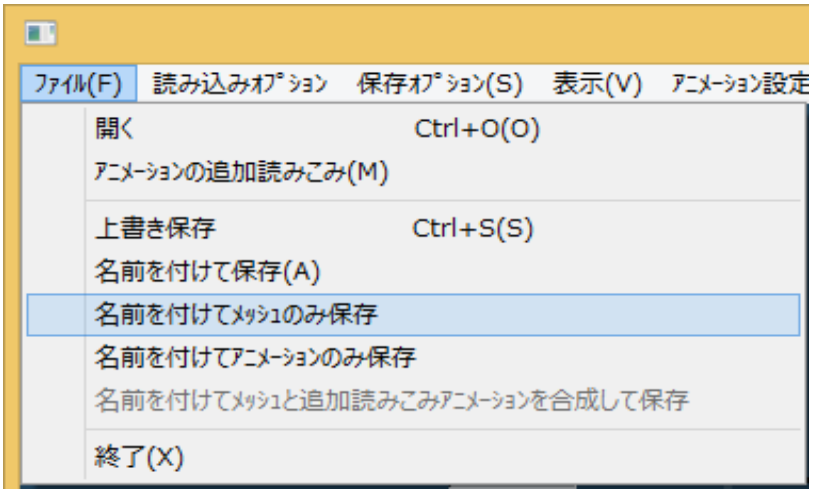
2)「DXLibModelViewer」でFBXファイルを開く。



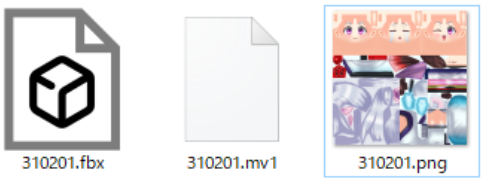
3)アニメーションデータが入っている場合は動きが確認できる。
左上の「アニメーション」のタブをクリックして確認する。 ※ここでの番号が再生する時に必要！



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。



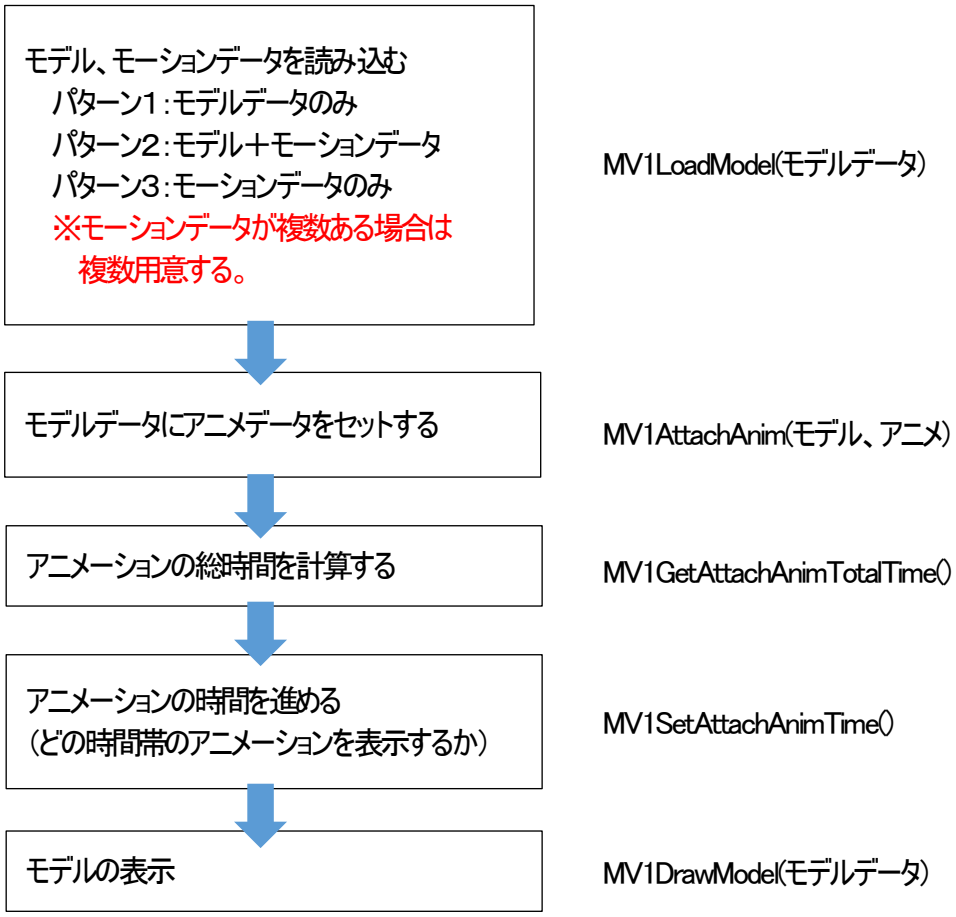
5)MV1形式のデータの完成！
MV1形式のモデルデータとテクスチャのデータがあるので全て同じフォルダで管理する事。



左より「FBX」「MV1」「テクスチャ」データの順。
テクスチャ等のファイル名は、登録時の情報になっているので勝手に書き換えない事！

■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);

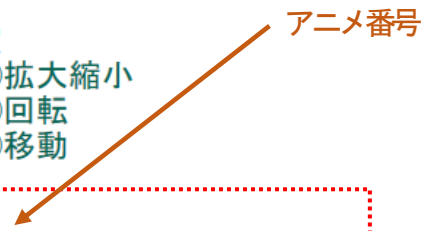
モデルハンドル・・・int model; ※読み込んだモデル
アニメハンドル・・・int no; ※MV1AttachAnim でセットしたアニメハンドル

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）

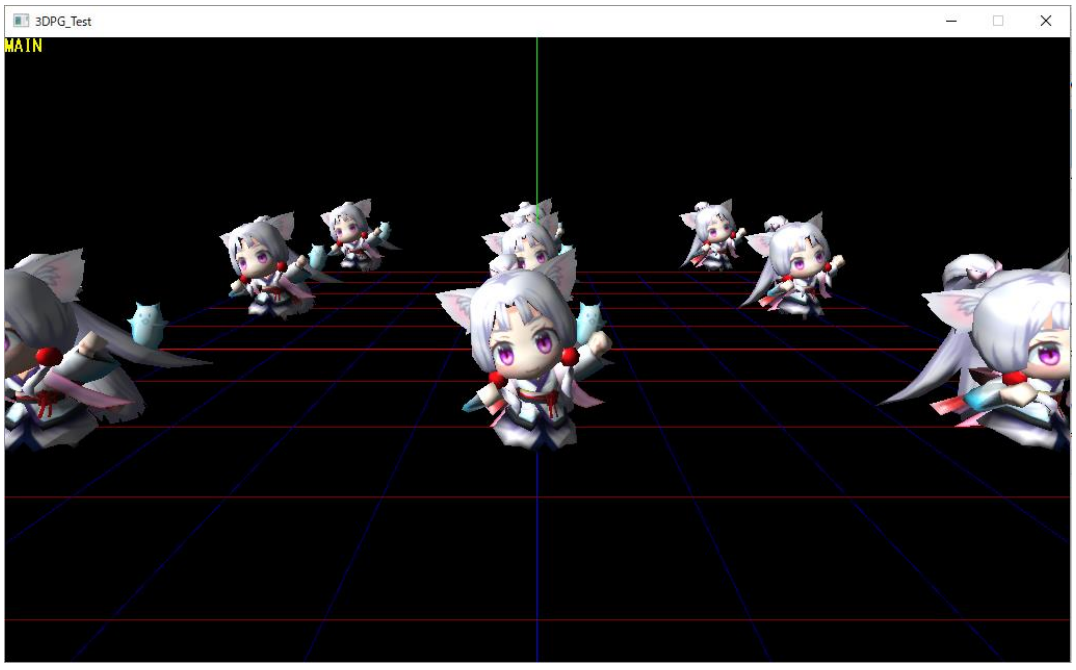
1)プログラムの例(読み込み→アニメーション設定)

```
// ①モデルデータの読み込み
model = MV1LoadModel("model/itako/itako.mv1");

// 基本パラメータ
pos = VGet(0.0f, 0.0f, 0.0f);
rol = VGet(0.0f, 0.0f, 0.0f);
scl = VGet(1.0f, 1.0f, 1.0f);

// 「拡大縮小」「回転」「移動」の設定
MV1SetScale(model, scl);          // ②拡大縮小
MV1SetRotationXYZ(model, rol);    // ③回転
MV1SetPosition(model, pos);       // ④移動

// ---------- アニメーション設定
// 7番アニメを設定。-1はmodelよりという意味
attachIndex = MV1AttachAnim(model, 7, -1, FALSE);

// アニメーションのトータル時間を計測
totalTime = MV1GetAttachAnimTotalTime(model, attachIndex);
playTime = 0;    // 再生開始場所を最初からにする

// modelにアニメーションをセット
MV1SetAttachAnimTime(model, attachIndex, playTime);
```



2)プログラムの例(更新→描画)

```
void Model::Update()
{
    // アニメーションの時間を進める
    playTime += 1.0f;
    if (playTime >= totalTime)
    {
        playTime = 0.0f;    // アニメーションの最後に達したら0に戻す
    }
    // 再生時間をセットする
    MV1SetAttachAnimTime(model, attachIndex, playTime);
}

void Model::Render()
{
    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、おなじキャラが大量に登場する場合など、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モデルを読み込み、それ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画像の場合に一度読み込んだ画像を使いまわす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。3Dモデルの場合は保持している状態情報が多い為、1つのモデルを使いまわす為にその都度全パラメーターを初期化する必要があるなど分身を作る方が速度的にも有利です。

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);

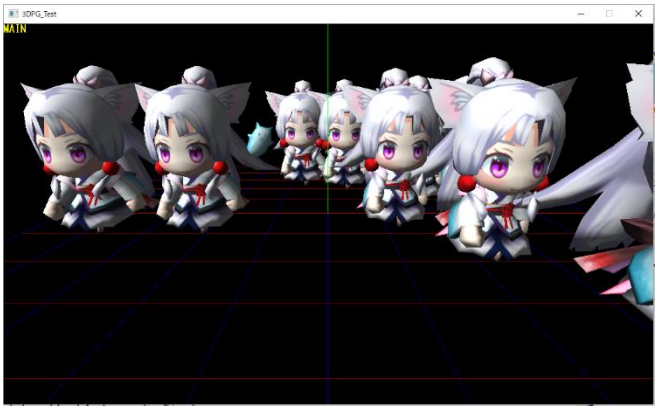
一度読み込んだモデルハンドル・・・int model; ※読み込んだモデル

1)プログラム例（生成時）　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。

```
int modelBase = MV1LoadModel("model/itako/itako.mv1");
for(int i = 0; i < 10; i++)
{
    model[i] = new Model(modelBase);
}
```

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。

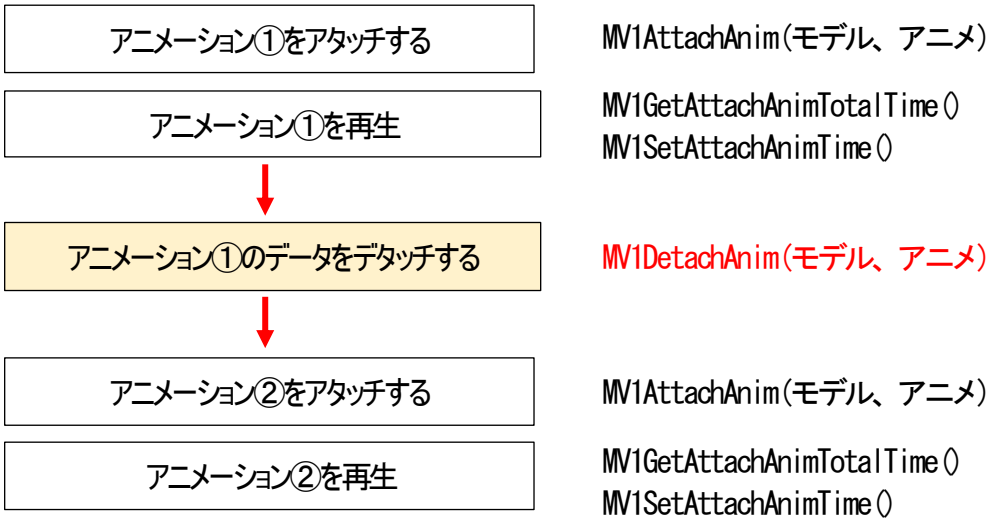
```
Model::Model(int modelBase)
{
    //model = MV1LoadModel("model/itako/itako.mv1");
    model = MV1DuplicateModel(modelBase);

    Init();
}
```



一度読み込んだデータの分身により
複数のオブジェクトを生成した例

■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。



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 Model::Update()
{
    if(CheckHitKey(KEY_INPUT_0))
    {
        MV1DetachAnim(model, attachIndex);
        attachIndex = MV1AttachAnim(model, 0, -1, FALSE);    // 0番読み込み
        totalTime = MV1GetAttachAnimTotalTime(model, attachIndex);
        playTime = 0;
        MV1SetAttachAnimTime(model, attachIndex, playTime); // アニメセット
    }
    if(CheckHitKey(KEY_INPUT_7))
    {
        MV1DetachAnim(model, attachIndex);
        attachIndex = MV1AttachAnim(model, 7, -1, FALSE);    // 7番読み込み
        totalTime = MV1GetAttachAnimTotalTime(model, attachIndex);
        playTime = 0;
        MV1SetAttachAnimTime(model, attachIndex, playTime); // アニメセット
    }
    // アニメーションの時間を進める
    playTime += 0.5f;
    if (playTime >= totalTime)
    {
        playTime = 0.0f;     // アニメーションの最後に達したら0に戻す
    }
    // 再生時間をセットする
    MV1SetAttachAnimTime(model, attachIndex, playTime);
}
```

■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。

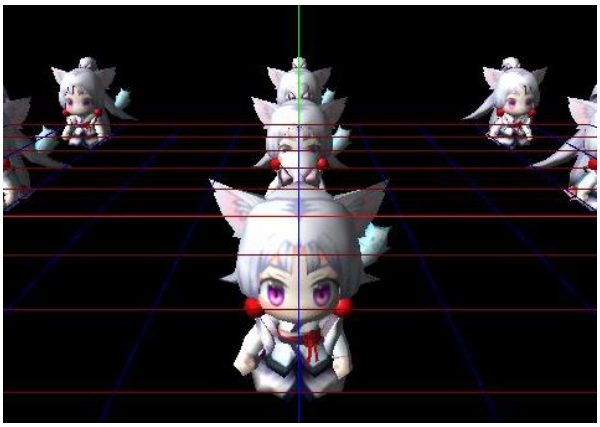
```
// 移動量を打ち消す
playerRootflame = MV1SearchFrame(playerModel1, "root");
MV1SetFrameUserLocalMatrix(playerModel1, playerRootflame, MGetIdent());
```

■Z バッファの使用

どんな順序で立体物を描画しても必ずカメラに近い物が最終描画結果として画面の前面に表示される様にする方法。
これをしておかないと、後から描いたものが今まで描いてあったものを上描きしてしまう場合があります。

```
int GameTask::SystemInit()
{
    ChangeWindowMode(true); // ウインドウモード
    SetWindowText(WINDOW_NAME);
    SetGraphMode(SCREEN_SIZE_X, SCREEN_SIZE_Y, 16);
    if (DxLib_Init() == -1) return -1;  // 初期化と裏画面化
    SetDrawScreen(DX_SCREEN_BACK);

    SetUseZBuffer3D(true);// Ｚバッファを有効にする
    SetWriteZBuffer3D(true);// Ｚバッファへの書き込みを有効にする

    return 0;
}
```

Ｚバッファ有効:ON
Ｚバッファ書き込み有効:ON



Ｚバッファ有効:OFF
Ｚバッファ書き込み有効:OFF
※後に描画したグリッド線が上に描かれている

実際の画面への反映は、
・Ｚバッファを使用する設定になっている状態である。
・Ｚバッファへの書き込みを行うかどうかの設定をする。
の2つの設定をする必要があるため、「SetUseZBuffer3D()」「SetWriteZBuffer3D()」の両方とも有効(true)にしておきます。