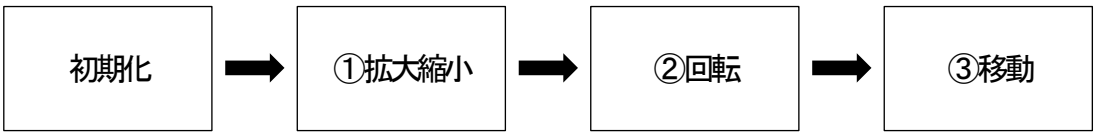
001.7

ゲームプログラミング応用／ゲームプログラミング実践

20 日で理解する3Dプログラミング 「その 5：移動と回転」

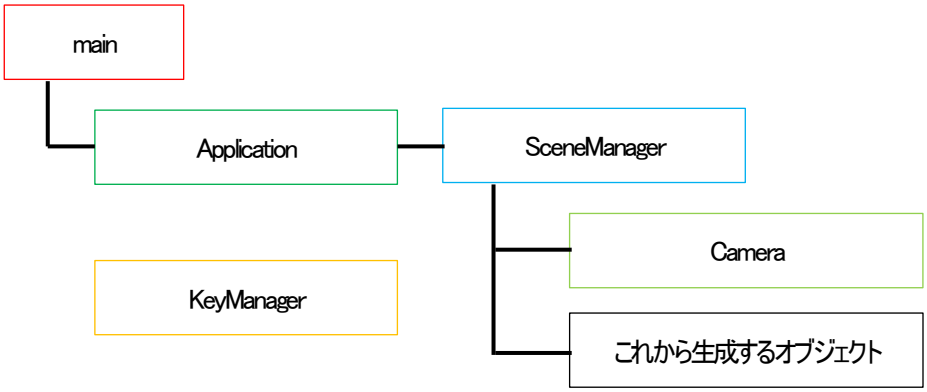
3D モデルなどの姿勢制御を行う方法は、大きく「拡大縮小」「回転」「移動」の 3 つの工程で行います。
最終的にはそれぞれの制御を「行列の演算」で行いますが、計算の都合で処理の順番は下記の様に統一していきます。

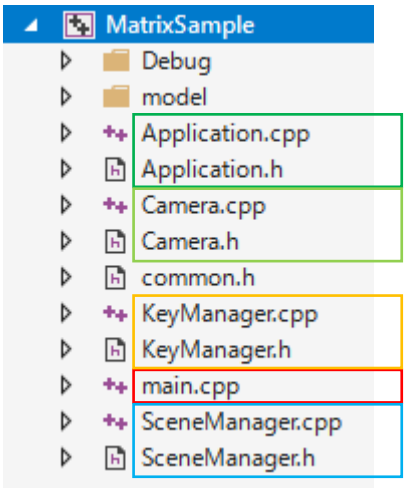


■プロジェクトの準備「Model00」

いつもの様に基本となるプロジェクトを作成します。main→Application→SceneManager の順番にクラスを構成し、カメラと今後作成するオブジェクトを管理していきます。キー入力はシングルトンで実装してみます。

※各ファイルのプログラムソースは省略。





モデルデータを描画する状態から始めます（アニメーション再生はなくても可能）。これまでのカメラとモデル表示のクラスを使用して実装していきましょう。

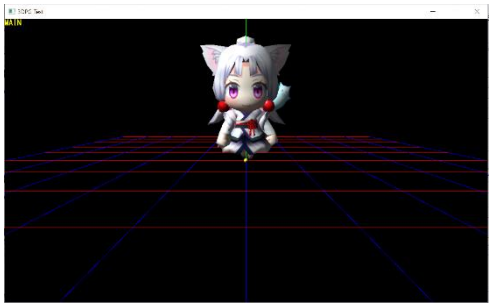
初期状態として、図の様にキャラクターが正面（-Z 方向）を向いた状態にしておきます。

Model00.h

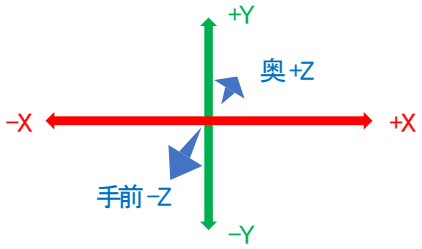
```
#pragma once

class Model00
{
public:
    Model00();
    ~Model00();
    void Init(void);
    void Update(void);
    void Render(void);

private:
    // モデルデータ用
    int mItakoModelID;

    VECTOR mPos;
    VECTOR mRol;
    VECTOR mScl;

    // アニメーション用
    int mAttachIndex;   // アニメーションデータ
    float mTotalTime;   // 再生総時間
    float mPlayTime;    // 再生箇所
};
```

Model00.cpp

```cpp
#include "DxLib.h"
#include "Model00.h"

Model00::Model00()
{
    mItakoModelID = MV1LoadModel("model/itako/itako.mv1");  // モデルデータ読み込み
    Init();
}

Model00::~Model00()
{
    MV1DeleteModel(mItakoModelID);  // モデルデータ削除
}

void Model00::Init(void)
{
    // 座標、回転、スケールの初期化
    mPos = VGet(0.0f, 0.0f, 0.0f);
    mRot = VGet(0.0f, 0.0f, 0.0f);
    mScl = VGet(1.0f, 1.0f, 1.0f);  // スケールは0.0fにしない様に注意！

    // アニメーションセット
    mAttachIndex = MV1AttachAnim(mItakoModelID, 7, -1, false);
    mTotalTime = MV1GetAttachAnimTotalTime(mItakoModelID, mAttachIndex);
    mPlayTime = 0.0f;
}

void Model00::Update(void)
{
    // アニメーションの時間を進める
    mPlayTime += 0.5f;
    if(mPlayTime >= mTotalTime)
    {
        mPlayTime = 0.0f;   // アニメーションの最後に達したら0に戻す
    }

    // モデルにアニメーションをセット
    MV1SetAttachAnimTime(mItakoModelID, mAttachIndex, mPlayTime);
}

void Model00::Render(void)
{
    // モデルの描画
    MV1DrawModel(mItakoModelID);
}
```



こんな感じになっていれば OK!



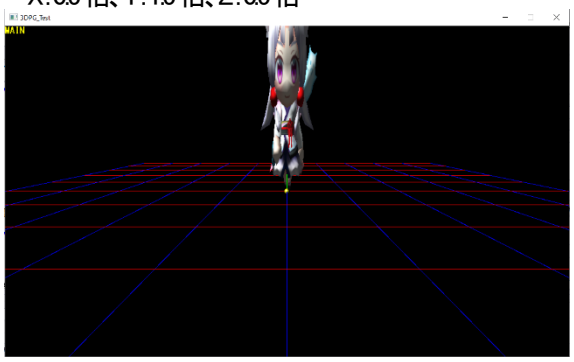
■拡大縮小

拡大縮小は、1.0f を基準とした倍率を XYZ 方向に設定して行います。MV1SetScale()命令で指定するだけです。

X:2 倍、Y:1 倍、Z:2 倍

```
void Model00::Update(void)
{
    // ①拡大縮小のセット
    mScl = VGet(2.0f, 1.0f, 2.0f);
    MV1SetScale(mItakoModelID, mScl);
```

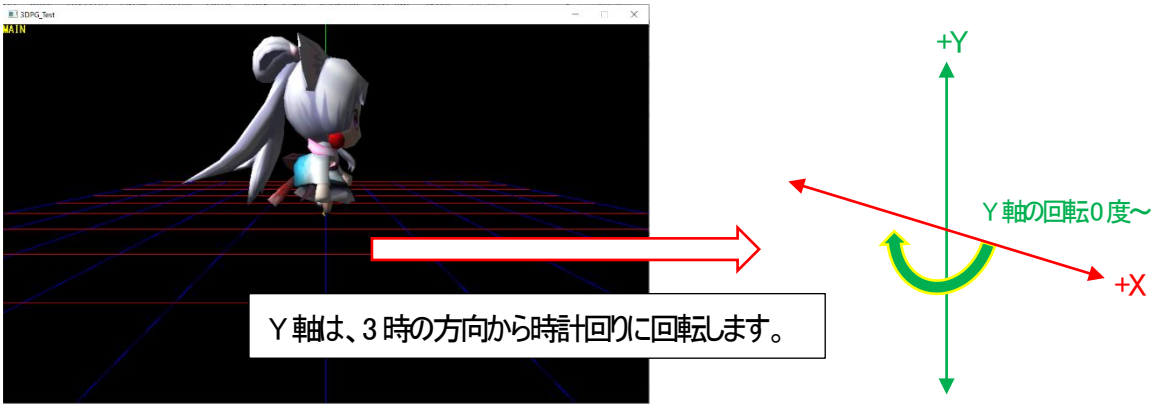
X:0.5 倍、Y:1.5 倍、Z:0.5 倍



```
void Model00::Update(void)
{
    // ①拡大縮小のセット
    mScl = VGet(0.5f, 1.5f, 0.5f);
    MV1SetScale(mItakoModelID, mScl);
```

■XZ 平面を基準とした「回転」 その1：準備

次に XY 軸での回転を行います。 ※3 軸の回転は次の段階で行います。
が、角度の扱いが 0 度の時の場所を把握しておく必要があります。例えば、Y 軸であれば 0 度の向きは時計の 3 時の方向となる為、XYZ 軸でいう+X 軸の方を向いている時が 0 度となり、その方向が正面という事になります。



そのまま描画した場合の状況を確認します

```
void Model00::Update(void)
{
    // ①拡大縮小のセット
    mScl = VGet(1.0f, 1.0f, 1.0f);
    MV1SetScale(mItakoModelID, mScl);

    // ②モデルの回転値をXYZ軸を基準にセットする
    mRol = VGet(0.0f, 0.0f, 0.0f);
    MV1SetRotationXYZ(mItakoModelID, mRol);
```

Ｙ軸が０度の時にモデルが右方向を向く様に補正します。 ※角度はラジアン指定です！

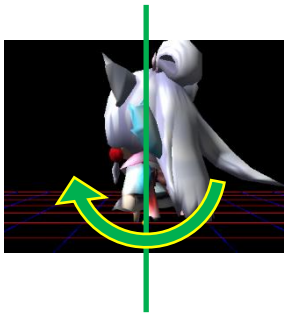
```
void Model00::Update(void)
{
    // ①拡大縮小のセット
    mScl = VGet(1.0f, 1.0f, 1.0f);
    MV1SetScale(mItakoModelID, mScl);

    // ②モデルの回転値をXYZ軸を基準にセットする
    mRol = VGet(0.0f, 0.0f, 0.0f);
    MV1SetRotationXYZ(mItakoModelID, mRol);
    MV1SetRotationXYZ(mItakoModelID, VGet(mRol.x, mRol.y+(DX_PI_F/180)*-90, mRol.z));
```

□XZ 平面を基準とした「回転」 その2：実践

キー入力によりそれぞれの軸に対して回転するようにします。

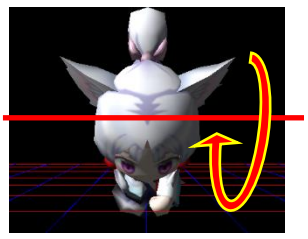
Ｙ軸回転

```
// ②モデルの回転値をXYZ軸を基準にセットする
//mRol = VGet(0.0f, 0.0f, 0.0f);
// Y軸右
if(KeyManager::GetInstance().mNewKey[KEY_RIGHT])
{
    mRol.y += (DX_PI_F / 180) * 2.0f;
}
// Y軸左
if (KeyManager::GetInstance().mNewKey[KEY_LEFT])
{
    mRol.y += (DX_PI_F / 180) * -2.0f;
}
```

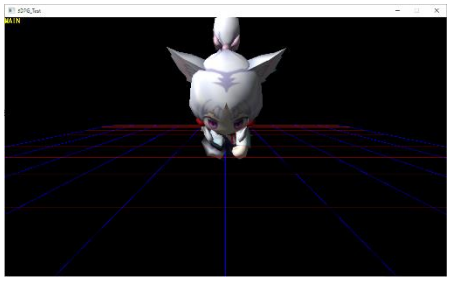


Ｘ軸回転

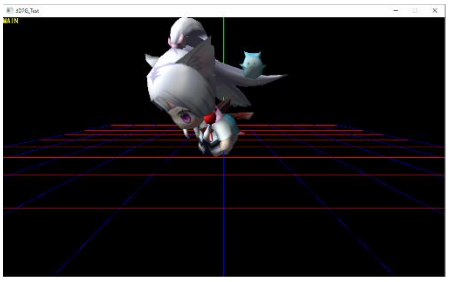
```
// X軸上
if (KeyManager::GetInstance().mNewKey[KEY_UP])
{
    mRol.x += (DX_PI_F / 180) * 2.0f;
}
// X軸下
if (KeyManager::GetInstance().mNewKey[KEY_DOWN])
{
    mRol.x += (DX_PI_F / 180) * -2.0f;
}
```



Ｙ軸回転、Ｘ軸回転の組み合わせもできます。



Ｘ軸で回転



Ｙ軸で回転

※体の軸で回転していない事に注意！

■XZ 平面を基準とした「移動」

次に移動を行います。

普通にXY軸方向に移動しても良いのですが、折角ですのでＹ軸回転をしたモデルの向きに移動をさせてみましょう。

「W]「S]キーで向いている方向に前後移動

```
// ③位置
// 前
if(KeyManager::GetInstance().mNewKey[KEY_W])
{
    mPos.z -= sinf(mRol.y) * 5.0f;
    mPos.x += cosf(mRol.y) * 5.0f;
}
// 後
if(KeyManager::GetInstance().mNewKey[KEY_S])
{
    mPos.z -= sinf(mRol.y) * -5.0f;
    mPos.x += cosf(mRol.y) * -5.0f;
}
```

#include<math.h>

でmath.hをインクルードしておきましょう

「A]「D]キーで向いている方向に対して左右移動

```
// 左
if (KeyManager::GetInstance().mNewKey[KEY_A])
{
    mPos.z -= sinf(mRol.y + (DX_PI_F/180) * -90) * 5.0f;
    mPos.x += cosf(mRol.y + (DX_PI_F/180) * -90) * 5.0f;
}
// 右
if (KeyManager::GetInstance().mNewKey[KEY_D])
{
    mPos.z -= sinf(mRol.y + (DX_PI_F / 180) * 90) * 5.0f;
    mPos.x += cosf(mRol.y + (DX_PI_F / 180) * 90) * 5.0f;
}

MV1SetPosition(mItakoModelID, mPos);
```

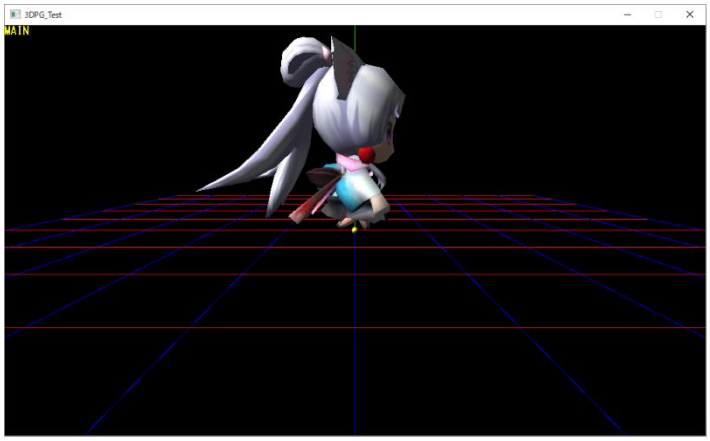
キャラクターを左右回転させて向いた方向に動くし、左右移動もできるし、上下移動(ジャンプなど)もできるのでこれはこれで3Dゲームを作るには十分な制御ができています。

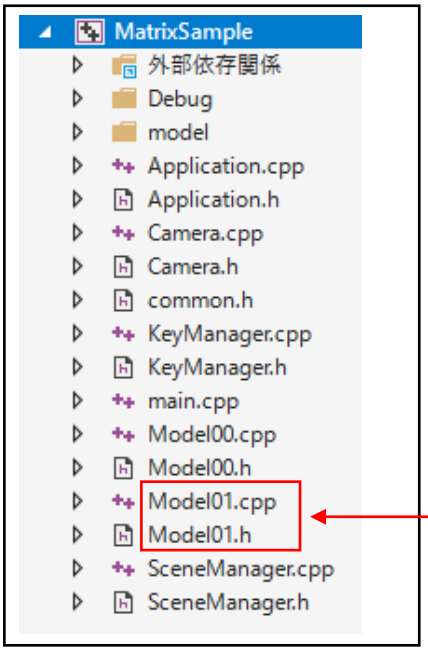
ただ、ゲームによっては3D空間を自由に動きまわる様なものもあるため、このままでは不十分です。

そこで、思い通りの制御ができる様に、更に座標の回転と移動について掘り下げていきましょう。

■3軸での回転と移動（行列とベクトルを使用します）「Model01」

XZ 平面の制御と分ける為に、3軸用に新たにクラスを作って処理の比較ができるようにします。





Model00 クラスの最初の状態と同様のものを複製して
Model01 クラスを作成します。
SceneManager からは、Model01 クラスのみインスタンスして
Update()と Render()を呼び出す様にしておきます。

■行列とベクトルを使う

それぞれの回転の状態をまとめる場合は「行列」を使用します。行列を使用すると「回転」「移動」「拡大縮小」をひとまとめにできて、なおかつ位置情報などを管理するベクトルに対して一度に計算できるメリットがあります。

3D 座標系に使用するベクトルと行列は下記の通りになります。

ベクトル　※VECTOR 型

(x , y , z)

行列（4×4行列）　※MATRIX 型

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

4×4行列1つで「拡大縮小・回転・移動」すべてを制御します。

では、実際にどの様に活用していくかを見ていきます。

■行列を使おう

行列を使うにはMATRIX型の構造体(４×４行列)を使用します。
姿勢制御の計算を自前で行うにはあまりにも大変なので、ここは潔く「行列」と「行列の計算式」を使用しましょう。

MATRIX 型

※下記の様に[4][4]の二次元配列で float 値を管理するイメージです。

m[0][0]	m[0][1]	m[0][2]	m[0][3]
m[1][0]	m[1][1]	m[1][2]	m[1][3]
m[2][0]	m[2][1]	m[2][2]	m[2][3]
m[3][0]	m[3][1]	m[3][2]	m[3][3]

【拡大縮小率を得る】
スケール値は各行の長さと等しくなります。

この3次元ベクトル「ABC」の長さ ＝X 軸のスケール値
この3次元ベクトル「DEF」の長さ ＝Y 軸のスケール値
この3次元ベクトル「GHI」の長さ ＝Z 軸のスケール値

A	B	C	0
D	E	F	0
G	H	I	0
J	K	L	1

【移動量を得る】
変換行列から移動値を得るには、4 行目の 1,2,3 列目がそれぞれ X,Y,Z 軸方向への移動量となります。

A	B	C	0
D	E	F	0
G	H	I	0
J	K	L	1

X 軸方向への移動量　Y 軸方向への移動　Z 軸方向への移動量

【XYZ 軸の向きを表すベクトルを得る(回転状況)】
1～3 行目を 3 次元のベクトルにするだけで、変換後の各軸の正方向を表すベクトルを得ることができます。

この3次元ベクトル「ABC」が「変換後の X 軸の向き」
この3次元ベクトル「DEF」が「変換後の Y 軸の向き」
この3次元ベクトル「GHI」が「変換後の Z 軸の向き」

A	B	C	0
D	E	F	0
G	H	I	0
J	K	L	1

【転置行列を取得する】
転置行列とは行列要素の行と列を逆転したものです。
転置する行列が回転行列の場合は逆回転する行列となります。

■単位行列を取得する
MATRIX MGetIdent(void);

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

単位行列とは、
正方行列において、対角成分が 1 でその他が 0 の行列で、簡単にいうと倍率が 1 で回転や移動が 0 の基本形となる行列を表します。

例) MATRIX matrix = MGetIdent( ); // 単位行列を matrix に代入する

□引数 VECTOR で指定された拡大値で拡大する行列を戻り値として返す
MATRIX MGetScale (VECTOR XYZ 軸の拡大値);

例) MATRIX matrix;
matrix = MGetScale(VGet(1.0f, 1.0f, 1.0f)); // 拡大行列を matrix に代入する

※実際の拡大処理は、ベクトル(元の拡大率)に対して VTransform 関数で変換を行う必要があります。

□平行移動行列を取得する
MATRIX MGetTranslate(VECTOR 平行移動ベクトル);

例) VECTOR trans;
MATRIX matrix;
trans = VGet(X 移動量, Y 移動量, Z 移動量);
matrix = MGetTranslate(trans); // trans で指定された平行移動を行う行列を返す。

※実際の移動処理は、ベクトル(元の座標)に対して VTransform 関数で変換を行う必要があります。

□X 軸の回転後の回転行列を取得する
MATRIX MGetRotX(float 回転値※ラジアン);

例) MATRIX matrix;
matrix = MGetRotX(rotX); // X 軸に rotX 回転した回転行列を返す。

※実際の回転後の座標は、ベクトル(元の座標)に対して VTransform 関数で変換を行う必要があります。

□Y 軸の回転後の回転行列を取得する
MATRIX MGetRotY(float 回転値※ラジアン);

例) MATRIX matrix;
matrix = MGetRotY(rotY); // Y 軸に rotY 回転した回転行列を返す。

※実際の回転後の座標は、ベクトル(元の座標)に対して VTransform 関数で変換を行う必要があります。

□Z 軸の回転後の回転行列を取得する
MATRIX MGetRotZ(float 回転値※ラジアン);

例) MATRIX matrix;
matrix = MGetRotZ(rotZ); // Z 軸に rotZ 回転した回転行列を返す。

※実際の回転後の座標はベクトル(元の座標)に対して VTransform 関数で変換を行う必要があります。

■引数のベクトルを引数の行列を使って変換したベクトルを返す
VECTOR VTransform(VECTOR 変換するベクトル , MATRIX 変換に使用する行列);

例) VECTOR newVec, oldVec;
MATRIX matrix;
newVec = VTransform(oldVec, matrix); // oldVec を matrix で変換したものを newVec に入れる。

■引数のベクトルを引数の行列を使って変換したベクトルを返す(スケーリング、回転のみ)
VECTOR VTransformSR(VECTOR 変換するベクトル , MATRIX 変換に使用する行列);

例) VECTOR newVec, oldVec;
MATRIX matrix;
newVec = VTransform(oldVec, matrix); // oldVec を matrix で変換したものを newVec に入れる。

※合成した行列の中には、平行移動 m[3][0]～m[3][2]が含まれており、回転だけをしたい時にこの部分の値が邪魔な場合があります(挙動に影響します)。この関数の場合は、平行移動 m[3][0]～m[3][2]を1として乗算する仕組みになっているので、スケーリングと回転情報のみを取り出して計算する事ができます。

■指定軸で指定角度回転する行列を取得する

MATRIX MGetRotAxis(VECTOR 回転軸 , float 回転値※ラジアン);

例) VECTOR rolVector; // 回転させる軸のベクトル
float rolAngle; // 回転値※ラジアン
MATRIX matrix; // 回転行列
matrix = MGetRotAxis(rolVector, rolAngle); // rolVector を rolAngle で回転した行列を返す。

■行列の合成(行列の掛け算)

MATRIX MMult(MATRIX 行列A , MATRIX 行列B);

例) MATRIX matC = MMult(matA, matB); // matA と matB を合成した結果を matC に入れる。

□拡大縮小行列(のメカニズム)

点(x, y, z)を原点に関してX軸方向にX倍、Y軸方向にY倍、Z軸方向にZ倍する行列

$$\begin{bmatrix} X' & Y' & Z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} X倍 & 0 & 0 & 0 \\ 0 & Y倍 & 0 & 0 \\ 0 & 0 & Z倍 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X' = (x＊X倍)+(y＊0)+(z＊0)+(1＊0) = x＊X倍
Y' = (x＊0)+(y＊Y倍)+(z＊0)+(1＊0) = y＊Y倍
Z' = (x＊0)+(y＊0)+(z＊Z倍)+(1＊0) = z＊Z倍
1 = (x＊0)+(y＊0)+(z＊0)+(1＊1) = 1

点(x, y, z) を3軸に対してそれぞれの拡大縮小率した値を(X', Y', Z')に保持する
X' = x＊X倍;
Y' = y＊Y倍;
Z' = z＊Z倍;

例題) 元のサイズ(1, 0.5, 1)をX軸方向に0.5、Y軸方向に2、Z軸方向に1倍した計算後の拡大率(X', Y', Z')

$$\begin{bmatrix} 1 & 0.5 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

⇒

X' = (1＊0.5) = 0.5
Y' = (0.5＊2) = 1
Z' = (1＊1) = 1

計算後の拡大率(0.5, 1, 1)

□平行移動行列(のメカニズム)

点(x, y, z)をX軸方向にTx、Y軸方向にTy、Z軸方向にTz、移動する行列

$$\begin{bmatrix} X' & Y' & Z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}$$

$$X' = (x*1)+(y*0)+(z*0)+(1*Tx) = x+Tx$$
$$Y' = (x*0)+(y*1)+(z*0)+(1*Ty) = y+Ty$$
$$Z' = (x*0)+(y*0)+(z*1)+(1*Tz) = z+Tz$$
$$1 = (x*0)+(y*0)+(z*0)+(1*1) = 1$$

点(x, y, z) を 3 軸に対してそれぞれの平行移動した値を(X', Y', Z')に保持する

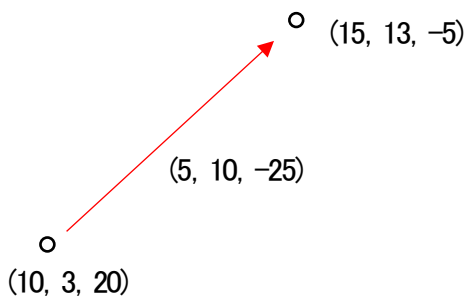
X' = x + Tx;
Y' = y + Ty;
Z' = z + Tz;

例題) 点(10, 3, 20)をX軸方向に5、Y軸方向に10、Z軸方向に−25、移動した後の座標(X', Y', Z')

$$\begin{bmatrix} 10 & 3 & 20 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 10 & -25 & 1 \end{bmatrix} \Rightarrow$$

X' = (10 * 1) + (1 * 5) = 10 + 5 = 15;
Y' = (3 * 1) + (1 * 10) = 3 + 10 = 13;
Z' = (20 * 1) + (1 * −25) = 20 − 25 = −5;

移動後の座標(15, 13, −5)



□X 軸回転（のメカニズム）

点(x, y, z)をX軸に θ 分回転した時の(X', Y', Z')を求める行列

$$\begin{bmatrix} X' & Y' & Z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} X' &= (x*1)+(y*0)+(z*0)+(1*0) \\ Y' &= (x*0)+(y*\cos\theta)+(z*\sin\theta)+(1*0) \\ Z' &= (x*0)+(y*-\sin\theta)+(z*\cos\theta)+(1*0) \\ 1 &= (x*0)+(y*0)+(z*0)+(1*1) \end{aligned}$$

点(x, y, z)の座標をX軸に θ 分回転した時の(X', Y', Z')

X' =(x＊1);

Y' =(y＊cos θ)+(z＊−sin θ);

Z' =(y＊sin θ)+(z＊cos θ);

例題) 点(15, 10, 20)をX軸で+55 度方向に回転した後の座標(X', Y', Z')

$$\begin{bmatrix} 15 & 10 & 20 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos & \sin & 0 \\ 0 & -\sin & \cos & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow$$

X' =(15＊1)

Y' =(10＊cos(55 度)+20＊−sin(55 度))

Z' =(10＊sin(55 度)+20＊cos(55 度))

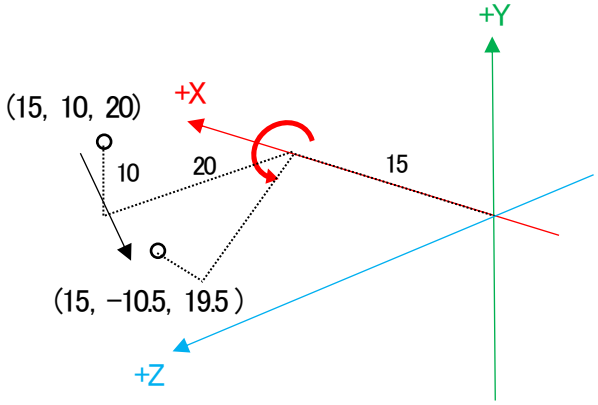
X' = 15

Y' =(10＊0.57+20＊−0.81)=5.7−16.2=−10.5

Z' =(10＊0.81+20＊0.57)=8.1+11.4=19.5

計算後の座標(15, 10.5, 19.5)

Y 軸の＋ーが sin、cos の値と逆なので、回転方向が逆になっている事に注意！



□Y 軸回転(のメカニズム)

点(x, y, z)をＹ軸に θ 分回転した時の(X', Y', Z')を求める行列

$$\begin{bmatrix} X' & Y' & Z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$X' = (x*\cos\theta)+(y*0)+(z*\sin\theta)+(1*0)$$
$$Y' = (x*0)+(y*1)+(z*0)+(1*0)$$
$$Z' = (x*-\sin\theta)+(y*0)+(z*\cos\theta)+(1*0)$$
$$1 = (x*0)+(y*0)+(z*0)+(1*1)$$

点(x, y, z)の座標をＸ軸に θ 分回転した時の(X', Y', Z')

X' =(x＊cosθ)+(z＊sinθ);

Y' =(y＊1);

Z' =(x＊−sinθ)+(z＊cosθ);

例題) 点(15, 10, 20)をＹ軸で+55 度方向に回転した後の座標(X', Y', Z')

$$\begin{bmatrix} 15 & 10 & 20 & 1 \end{bmatrix} \begin{bmatrix} \cos & 0 & -\sin & 0 \\ 0 & 1 & 0 & 0 \\ \sin & 0 & \cos & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow$$

X' =(15＊cos(55 度)＋20＊sin(55 度))

Y' =(10＊1)

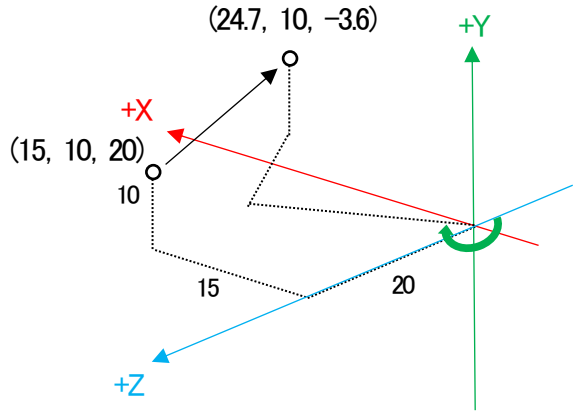
Z' =(15＊−sin(55 度)＋20＊cos(55 度))

X' =(15＊0.57＋20＊0.81)= 8.55＋16.2 = 24.7

Y' = 10

Z' =(15＊−0.81＋20＊0.57)= −12.15＋11.4 = −3.6

計算後の座標(24.7, 10, −3.6)



□Z 軸回転(のメカニズム)

点(x, y, z)をＺ軸に θ 分回転した時の(X', Y', Z')を求める行列

$$\begin{bmatrix} X' & Y' & Z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$X' = (\ x*\cos\theta\ )+(\ y*-\sin\theta\ )+(\ z*0\ )+(\ 1*0\ )$$
$$Y' = (\ x*\sin\theta\ )+(\ y*\cos\theta\ )+(\ z*0\ )+(\ 1*0\ )$$
$$Z' = (\ x*0\ )+(\ y*0\ )+(\ z*1\ )+(\ 1*0\ )$$
$$1 = (\ x*0\ )+(\ y*0\ )+(\ z*0\ )+(\ 1*1\ )$$

点(x, y, z)の座標をＺ軸に θ 分回転した時の(X', Y', Z')

X' =(x＊cos θ)+(y＊−sin θ);

Y' =(x＊sin θ)+(y＊cos θ);

Z' =(z＊1);

例題) 点(15, 10, 20)をＺ軸で+55 度方向に回転した後の座標(X', Y', Z')

$$\begin{bmatrix} 15 & 10 & 20 & 1 \end{bmatrix} \begin{bmatrix} \cos & \sin & 0 & 0 \\ -\sin & \cos & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow$$

X' =(15＊cos(55 度)+10＊−sin(55 度))

Y' =(15＊sin(55 度)+10＊cos(55 度))

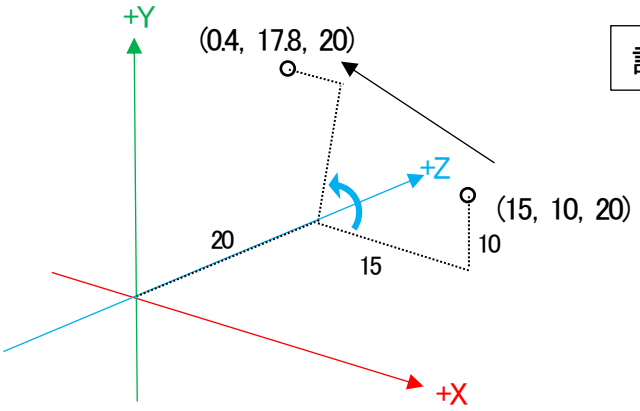
Z' =(20＊1)

X' =(15＊0.57+10＊−0.81)=8.5−8.1=0.4

Y' =(15＊0.81+10＊0.57)=12.1+5.7=17.8

Z' = 20

計算後の座標(0.4, 17.8, 20)



■行列を使った実際の処理（準備、初期化）

まず準備

```
class Model01
{
public:
    Model01();
    ~Model01();
    void Init(void);
    void Update(void);
    void Render(void);

private:
    // モデルデータ用
    int mItakoModelID;

    // 基本座標系
    VECTOR mPos;
    VECTOR mRol;
    VECTOR mScl;

    // 管理用行列
    MATRIX mMatrix; // 全体の行列

    MATRIX mMatScl; // 拡大縮小用
    MATRIX mMatRot; // 回転用
    MATRIX mMatTrans;   // 移動用

    // アニメーション用
    int mAttachIndex;   // アニメーションデータ
    float mTotalTime;   // 再生総時間
    float mPlayTime;    // 再生箇所
};
```

計算用を含めて MATRIX 型の変数を準備する。

初期化

```
mMatrix = MGetIdent();  // 全体の行列

mMatScl = MGetIdent();  // 拡大縮小用
mMatRot = MGetIdent();  // 回転用
mMatTrans = MGetIdent();    // 移動用
```

MGetIdent()で
単位行列を代入して初期化する

■行列を使った実際の処理（①拡大縮小：スケーリング）

スケール用の行列(mMatScl)に、MGetScale()で VECTOR で指定した拡大縮小率を変換して代入します。
※行列を使用する為、MV1SetScale()が使用できなくなります。

```
// ①拡大縮小のセット
mScl = VGet(1.0f, 1.0f, 1.0f);
//MV1SetScale(mItakoModelID, mScl); // MV1SetMatrix()を使用すると無効になる
mMatScl = MGetScale(mScl);  // スケール用の行列を作成
```

■行列を使った実際の処理（②回転）

回転の場合は、それぞれの軸毎に行列を作成して合成していきます。サンプルでは、分かりやすいようにローカル変数で対応しています。最後に、回転用の行列(mMatRot)に合成していますが、毎回単位行列を代入して初期化しておく必要があるので注意してください。

```
// X軸回転の行列を作成
MATRIX mx = MGetRotX(mRot.x);

// Y軸回転の行列を作成
MATRIX my = MGetRotY(mRot.y + (DX_PI_F / 180) * -90);

// 行列に合成
mMatRot = MGetIdent();  // 一旦初期化
mMatRot = MMult(mMatRot, mx);
mMatRot = MMult(mMatRot, my);
```

■行列を使った実際の処理（③移動）

移動情報は、MGetTranlate()で指定したベクトル分移動した行列を取得していきます。
※スケール同様、行列を使用する為、MV1SetPosition()が使用できなくなります。

```
// 移動行列を作成する
//MV1SetPosition(mItakoModelID, mPos);  // MV1SetMatrix()を使用すると無効になる
mMatTrans = MGetTranslate(mPos);
```

■行列を使った実際の処理(④合成)

最後に全ての行列を合成します。合成する順番も「スケール」→「回転」→「移動」の順番に気を付けてください。
興味のある方は、試しに順番を変えてみてください。最終的に出来上がった行列をモデルに反映させるのは
MV1SetMatrix()を使用します。mMatrix には、スケール、回転、移動の情報が全て1つに集約されています。

```
// モデルデータに情報を反映
mMatrix = MGetIdent();  // 全体の行列
mMatrix = MMult(mMatrix, mMatScl);  // スケール情報を合成
mMatrix = MMult(mMatrix, mMatRot);  // 回転情報を合成
mMatrix = MMult(mMatrix, mMatTrans);    // 位置情報を合成

MV1SetMatrix(mItakoModelID, mMatrix);  ← モデルデータに行列を反映させる
```

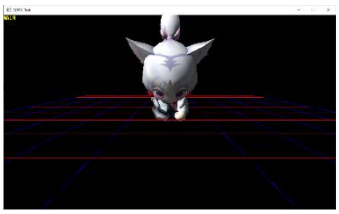
■行列を使った処理の問題点

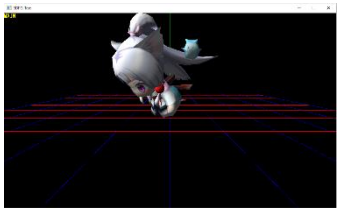
回転を加えていくと思い通りに回転しない時があります。実は、X、Y、Z 軸の3軸の回転を組み合わせるだけでは、

思い通りの方向に向ける事が出来ないのです。

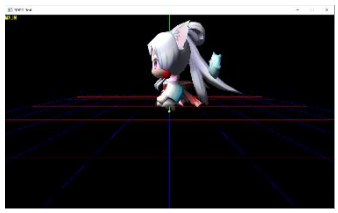
2軸の回転までは問題ないのですが、3軸の回転になるとそうはいきません。特にZ 軸回転が厄介です。
又、常に XYZ 軸での回転になっているのも問題です。できる事ならモデルを軸とした回転にしたい所です。

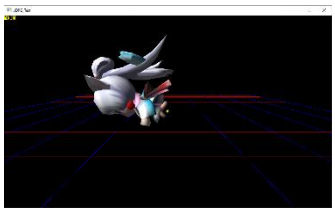
・(例) X 軸回転 ＋ Y 軸回転 ＋Z 軸回転


X 軸で手前に回転


Y 軸で横に回転


Z 軸で右に回転・・??


Y 軸で右方向に回転

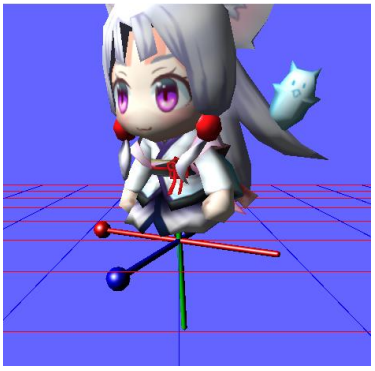

Z 軸で左右に回転・・??

それらを解決する為には、

回転する XYZ 軸自体を回転軸に合わせる**「指定軸の回転」**を使用する必要があります。

■基準軸を表示しよう

物体の回転状態が分かるように、その物体の基準となる XYZ 軸を描画しましょう。
他でも流用できるように汎用性のある関数化しておくと良いです。

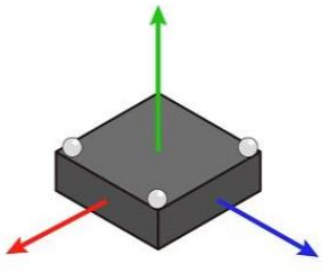


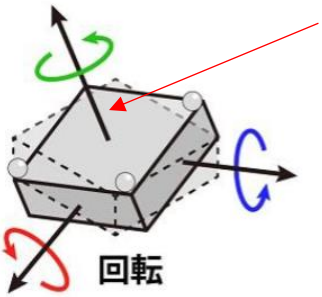
```
void Model01::Render(void)
{
    // モデルの描画
    MV1DrawModel(mItakoModelID);

    // 基準のXYZ軸
    DrawLocalXYZ(mPos, mMatrix, 100.0f, 3.0f);
}
```

```
void Model01::DrawLocalXYZ(VECTOR pos, MATRIX matrix, float length, float weight)
{
    VECTOR sPos = VGet(0.0f, 0.0f, 0.0f);
    VECTOR ePos = VGet(0.0f, 0.0f, 0.0f);
    VECTOR axis = VGet(0.0f, 0.0f, 0.0f);

    // Y(緑)
    axis = VTransformSR(VGet(0.0f, 1.0f, 0.0f), matrix);
    axis = VNorm(axis);     // ベクトルの正規化
    sPos.x = pos.x + axis.x * length;
    sPos.y = pos.y + axis.y * length;
    sPos.z = pos.z + axis.z * length;
    ePos.x = pos.x + axis.x * -length;
    ePos.y = pos.y + axis.y * -length;
    ePos.z = pos.z + axis.z * -length;
    DrawSphere3D(sPos, 10, 4, 0x00ff00, 0xffffff, true);    // +方向の目印の球
    DrawCapsule3D(sPos, ePos, weight, 4, 0x00ff00, 0xffffff, true);

    // X(赤)
    axis = VTransformSR(VGet(1.0f, 0.0f, 0.0f), matrix);
    axis = VNorm(axis); // ベクトルの正規化
    sPos.x = pos.x + axis.x * length;
    sPos.y = pos.y + axis.y * length;
    sPos.z = pos.z + axis.z * length;
    ePos.x = pos.x + axis.x * -length;
    ePos.y = pos.y + axis.y * -length;
    ePos.z = pos.z + axis.z * -length;
    DrawSphere3D(sPos, 10, 4, 0xff0000, 0xffffff, true);    // +方向の目印の球
    DrawCapsule3D(sPos, ePos, weight, 4, 0xff0000, 0xffffff, true);

    // Z(青)
    axis = VTransformSR(VGet(0.0f, 0.0f, 1.0f), matrix);
    axis = VNorm(axis); // ベクトルの正規化
    sPos.x = pos.x + axis.x * length;
    sPos.y = pos.y + axis.y * length;
    sPos.z = pos.z + axis.z * length;
    ePos.x = pos.x + axis.x * -length;
    ePos.y = pos.y + axis.y * -length;
    ePos.z = pos.z + axis.z * -length;
    DrawSphere3D(sPos, 10, 4, 0x0000ff, 0xffffff, true);    // +方向の目印の球
    DrawCapsule3D(sPos, ePos, weight, 4, 0x0000ff, 0xffffff, true);
}
```

■指定軸で回転とは？

基本のXYZ軸での回転だけでは物体の姿勢制御が正しくできません。傾いた物体を基準としたXYZ軸で回転させる為には、回転状態に合わせたXYZ軸を指定しそれぞれを回転する様にしていきます。





回転させる軸を、回転状態に合わせて変換し指定軸とする。

■指定軸に対しての「回転」(のメカニズム)

指定軸(nx, ny, nz)で θ 回転

$$
\begin{bmatrix} x & y & z & 1 \end{bmatrix}
\begin{bmatrix}
(nx*nx)*(1-\cos\theta)+\cos\theta & (ny*nx)*(1-\cos\theta)+nz*\sin\theta & (nz*nx)*(1-\cos\theta)-ny*\sin\theta & 0 \\
(nx*ny)*(1-\cos\theta)-nz*\sin\theta & (ny*ny)*(1-\cos\theta)+\cos\theta & (nz*ny)*(1-\cos\theta)+nx*\sin\theta & 0 \\
(nx*nz)*(1-\cos\theta)+ny*\sin\theta & (ny*nz)*(1-\cos\theta)-nx*\sin\theta & (nz*nz)*(1-\cos\theta)+\cos\theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

点(x, y, z)の座標を(nx, ny, nz)軸に θ 分回転した時の(X', Y', Z')

$$
\begin{aligned}
X' = {} & x*((nx*nx)*(1-\cos\theta)+\cos\theta) \\
& + y*((nx*ny)*(1-\cos\theta)-nz*\sin\theta) \\
& + z*((nx*nz)*(1-\cos\theta)+ny*\sin\theta);
\end{aligned}
$$

$$
\begin{aligned}
Y' = {} & x*((ny*nx)*(1-\cos\theta)+nz*\sin\theta) \\
& + y*((ny*ny)*(1-\cos\theta)+\cos\theta) \\
& + z*((ny*nz)*(1-\cos\theta)-nx*\sin\theta);
\end{aligned}
$$

$$
\begin{aligned}
Z' = {} & x*((nz*nx)*(1-\cos\theta)-ny*\sin\theta) \\
& + y*((nz*ny)*(1-\cos\theta)+nx*\sin\theta) \\
& + z*((nz*nz)*(1-\cos\theta)+\cos\theta);
\end{aligned}
$$

※ロドリゲスの回転公式とも言います

■指定軸での回転を使用した回転処理「Model02」

回転処理用に新たにクラスを作って処理の比較ができるようにします。

□手順

1. スケール

   ※スケール用の行列(mMatScl)を使用する

   MGetScale(拡大縮小ベクトル)で mMatScl を生成する。

2. 回転

   ※回転用の行列(mMatRot)を使用する。

   X 軸を回転状態に合わせて変換し、その回転した軸を基準に回転させた行列を mMatRot に合成する。

   Y軸を回転状態に合わせて変換し、その回転した軸を基準に回転させた行列を mMatRot に合成する。

   Z軸を回転状態に合わせて変換し、その回転した軸を基準に回転させた行列を mMatRot に合成する。

3. 移動

   ※移動用の行列(mMatTrans)を使用する

   前後移動・・・回転状態の Z 軸を計算し、その方向に前後移動量の移動をさせて mPos を更新する。

   左右移動・・・回転状態の X 軸を計算し、その方向に左右移動量の移動をさせて mPos を更新する。

   上下移動・・・回転状態の Y 軸を計算し、その方向に上下移動量の移動をさせて mPos を更新する。

   mMatTrans に、MGetTranslate(mPos)で移動行列を生成する。

4. 行列の合成

   マスターの行列を初期化

   マスター行列とスケール行列を合成し、マスター行列に入れる。

   マスター行列と回転行列を合成し、マスター行列に入れる。

   マスター行列と移動行列を合成し、マスター行列に入れる。

5. モデルデータに合成した行列を反映する

■指定軸の回転の処理（①拡大縮小：スケーリング）

スケールに関しては、前回の行列を使用した方法と同様です。
XYZ 方向での拡大縮小率を mScl で指定します。

```
//  ==================== ①拡大縮小のセット
mScl = VGet(1.0f, 1.0f, 1.0f);
mMatScl = MGetScale(mScl);  // スケール用の行列を作成
```

■指定軸の回転の処理（②回転）

回転処理が今回大きく異なる所です。X→Y→Z の順番を決めてそれぞれ軸を変更しながら回転させます。
又、ゲームループのフレーム毎に回転量を加算していきますので、これまでの様にそれぞれの軸で何度回転している等の、累積した角度では管理しないので注意が必要です。

```
//  ==================== ②モデルの回転値をXYZ軸を基準にセットする
mRol = VGet(0.0f, 0.0f, 0.0f);

// Y軸右
if (KeyManager::GetInstance().mNewKey[KEY_RIGHT])
{
    mRol.y = (DX_PI_F / 180) * 2.0f;
}
// Y軸左
if (KeyManager::GetInstance().mNewKey[KEY_LEFT])
{
    mRol.y = (DX_PI_F / 180) * -2.0f;
}

// X軸上
if (KeyManager::GetInstance().mNewKey[KEY_UP])
{
    mRol.x = (DX_PI_F / 180) * 2.0f;
}
// X軸下
if (KeyManager::GetInstance().mNewKey[KEY_DOWN])
{
    mRol.x = (DX_PI_F / 180) * -2.0f;
}

// Z軸上
if (KeyManager::GetInstance().mNewKey[KEY_RA])
{
    mRol.z = (DX_PI_F / 180) * 2.0f;
}
// Z軸下
if (KeyManager::GetInstance().mNewKey[KEY_RB])
{
    mRol.z = (DX_PI_F / 180) * -2.0f;
}
```

毎回初期化します

毎回のループで
回転する量だけ設定します

【まずX軸回転をやってみます】

mMatRotに全体の回転行列が入っているので、基準となるX軸を回転した状態に変換します。

その後、その軸を基準に今回回転する分を回転させた行列を計算しローカル変数に一時保存します(mat)。

その後、mMatRotにmatを合成しておきます。

```
VECTOR axis = VGet(0.0f, 0.0f, 0.f);
MATRIX mat = MGetIdent();
```

計算用の変数を準備
毎回フレーム初期化して使用します

VTransform命令で、X軸(1.0f, 0.0f, 0.0f)をmMatRotの回転行列で変換し、ベクトルaxisに代入します。

※VNorm命令でベクトルを正規化する事もできます(今回は正規化しなくても大丈夫です)。

```
// X軸回転の行列を作成
axis = VTransform(VGet(1.0f, 0.0f, 0.0f), mMatRot); // X軸を回転状態に合わせる

//axis = VNorm(axis);   // ベクトルを正規化する
```

MGetRotAxis命令で、指定した軸(ベクトル)axisで、mRot.x分回転した、回転行列を一時保存します。

その後、mMatRot行列に合成して手順終了。

```
mat = MGetRotAxis(axis, mRot.x);    // 変換したX軸でX回転する行列を算出する

mMatRot = MMult(mMatRot, mat);  // 回転行列に合成
```

■指定軸の回転の処理(③移動)

移動処理も回転と同様に、毎フレーム毎にXYZ方向にどれだけ移動するかで処理していきます。

XYZ方向の移動量が決まった所で、それぞれの基本軸を全体の回転行列で回転させておき、その方向に移動させるという2段階の手順を行います。

毎フレーム毎に移動量を設定します。

```
//  ==================== ③位置
VECTOR mov = VGet(0.0f, 0.0f, 0.0f);

// 前
if (KeyManager::GetInstance().mNewKey[KEY_W])
{
    mov.z = 5.0f;   // 移動量のみローカル変数に設定
    //mPos.z -= sinf(mRot.y) * 5.0f;
    //mPos.x += cosf(mRot.y) * 5.0f;
}
// 後
if (KeyManager::GetInstance().mNewKey[KEY_S])
{
    mov.z = -5.0f;  // 移動量のみローカル変数に設定
    //mPos.z -= sinf(mRot.y) * -5.0f;
    //mPos.x += cosf(mRot.y) * -5.0f;
}
```

毎回初期化を行います
※ローカル変数で処理を行います

直接mPosに加算するのではなく
移動量のみ設定します。

※左右、上下移動は省略

VTransform 命令で、移動させたい方向(Z 軸)を、mMatRot で変換し、ベクトル frontBack に代入します。
その後、その正規化したベクトルに対して移動量を掛けていきます。

```
// 前後
VECTOR frontBack = VTransform(VGet(0.0f, 0.0f, 1.0f), mMatRot);
//frontBack = VNorm(frontBack); // ベクトルを正規化する

mPos.x += frontBack.x * mov.z;
mPos.y += frontBack.y * mov.z;
mPos.z += frontBack.z * mov.z;
```

移動方向のベクトルに移動量を掛けて
mPos の座標を更新します。

最後に MGetTranslate 命令で、原点からの移動行列を完成させます。

```
// 移動行列を作成する
mMatTrans = MGetTranslate(mPos);
```

■指定軸の回転の処理(④合成)

最後に全ての行列を合成し、モデルデータに mMatrix を反映させれば完了です。

```
// ==================== 全体の行列に情報を反映
mMatrix = MGetIdent();  // 全体の行列
mMatrix = MMult(mMatrix, mMatScl);  // スケール情報を合成
mMatrix = MMult(mMatrix, mMatRot);  // 回転情報を合成
mMatrix = MMult(mMatrix, mMatTrans);    // 位置情報を合成

// ==================== モデルデータに反映
MV1SetMatrix(mItakoModelID, mMatrix);
```

■指定軸の回転の処理(完成)

下記制御を実装して処理を完成させましょう。

【Y 軸回転】

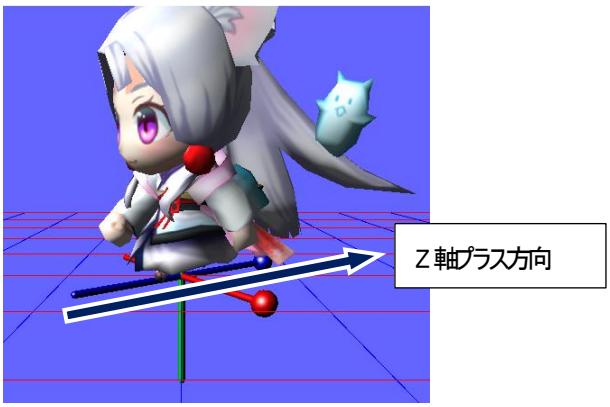
【Z 軸回転】

【左右移動】

【上下移動】

■超細かい処理（回転の補正）

モデルデータによっては、0 度の状態に配置しても正面の向きが違っている場合があります。
下記のモデルの場合は、正面を向かせる為にＹ軸に 180 度回転をさせる必要があります。



今回は、最終的に回転オフセット量を反映させる方法をやってみます。

まず、管理用の行列 drawMatrix を準備します。

```
// 描画用に基準に向く様にY軸回転で補正する
MATRIX drawMatrix = MGetIdent();    // 描画用の行列を準備
```

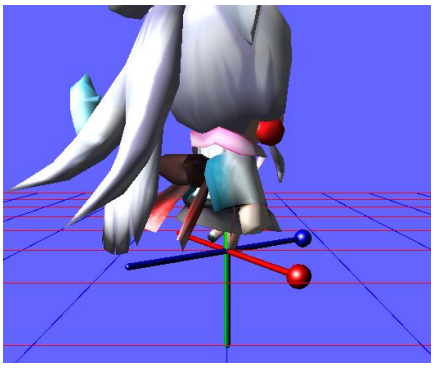
補正用のＹ軸を回転行列で変換し、そのベクトルを指定軸として必要分回転させて、その行列を保存しておきます。

```
VECTOR v = VTransform(VGet(0.0f, 1.0f, 0.0f), mMatRot); // Y軸を回転状況に合わせて回転
MATRIX m = MGetRotAxis(v, (DX_PI_F / 180) * 180);   // Y軸で180度回転させる
```

最後に、描画用の行列を合成し、モデルに反映させます。

```
drawMatrix = MMult(drawMatrix, mMatScl);    // スケール情報を合成
drawMatrix = MMult(drawMatrix, mMatRot);    // 回転情報を合成
drawMatrix = MMult(drawMatrix, m);          // 補正用の回転情報を合成
drawMatrix = MMult(drawMatrix, mMatTrans);  // 位置情報を合成

// ==================== モデルデータに反映
//MV1SetMatrix(mItakoModelID, mMatrix);
MV1SetMatrix(mItakoModelID, drawMatrix);
```

完成です。Ｚ軸のプラス方法にオブジェクトが向く様になりました。

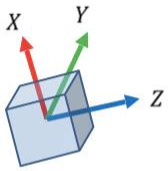


■クォータニオン(四元数)を使用した回転処理

クォータニオン（四元数）は、「航空機」「宇宙機」「ロボティクス」「3D ゲーム」「3DCG」などで使われるものです。
Unity での 3D ゲーム開発でも頻繁に出て来るキーワードなので、少しだけ触れてみましょう。

□クォータニオンでやりたい事

宇宙開発などでも使われるぐらいのものですが、大ざっぱに言うと
3D 空間における「回転」と「どういう回転でその向きになったかという姿勢」を制御すると思ってください。



姿勢 = 物体がどういう向きになっているか
= 物体がどういう**回転**でその向きになったか

□そもそもクォータニオンとは

実体は四次元ベクトルです。ただし以下の点で特別な四次元ベクトルになっています。
通常は、ベクトル同士での足し算、引き算はできても、掛け算は定義できません。※内積はが結果がベクトルでない。
しかしながら、クォータニオンの場合は、クォータニオン同士の掛け算ができて、結果もクォータニオンになるのです。
これにより何が良いのかというと、①の回転、②の回転・・という風に立て続けに回転を重ねていっても、最終的に
1つの回転に集約されるという便利な性質があります。
クォータニオンは任意軸(指定軸)回転がキーワードとして言及される場面も多いです。
いかなる回転（三次元空間内）も表現できることが強みです。具体的には以下のように定義します。

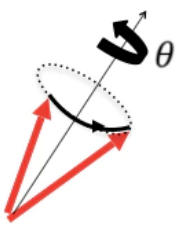
三次元空間において

- 方向ベクトル $(\lambda_x, \lambda_y, \lambda_z)$ を回転軸として
- 角度 θ だけ回転する

x, y, z, t の4つのパラメータ
だけで管理を行います！

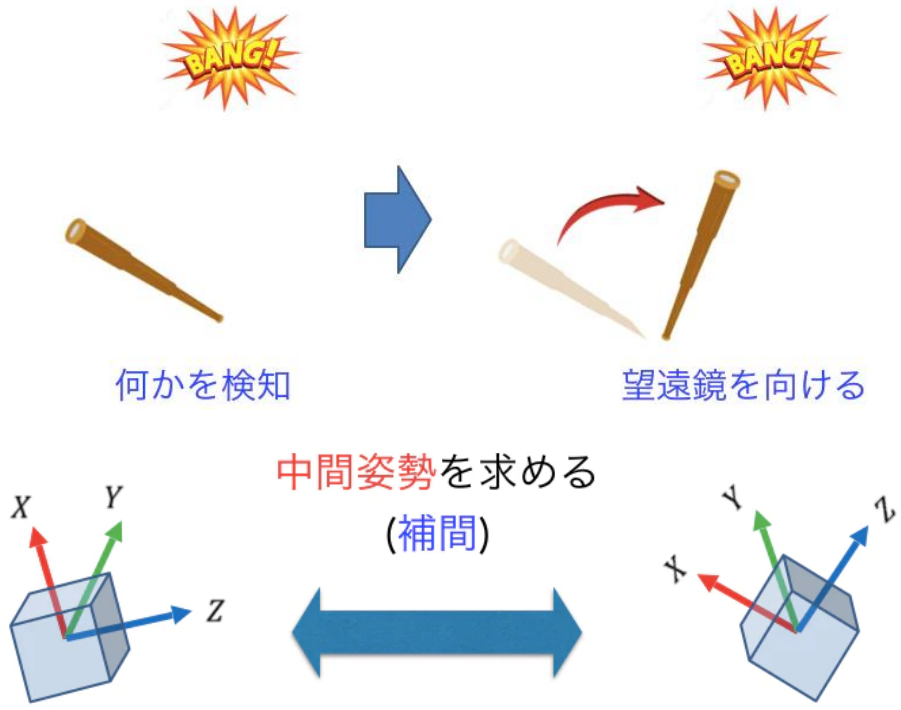
という「回転」を表すクォータニオンは、**四次元ベクトル** $(\lambda_x \sin\frac{\theta}{2}, \lambda_y \sin\frac{\theta}{2}, \lambda_z \sin\frac{\theta}{2}, \cos\frac{\theta}{2})$ で表します。

回転軸 (λx, λy, λz)



□要するに・・

・キャラクターの視線をある方向へと徐々に向けて行く（3D ゲーム）
・宇宙機に搭載した望遠鏡を所望の方向へ向ける（航空宇宙）
といった 3 次元空間での制御の場面で必要になるものです。Unity でも知らず知らずのうちに使われてます。
球面回転補間といって、回転状態 A と回転状態 B の中間の状態をいとも簡単に求めることができます。



Unity で、オブジェクトの方向を指定する場合の例

・LookAt 関数・・・すぐに指定方向を向かせる時に使う
・Quaternion.Slerp 関数・・・方向を補完しゆっくり回転させる時に使う

```
float t = Time.time;
Quaternion from_qua = Quaternion.Euler(0f, 0f, 0f);
Quaternion to_qua = Quaternion.Euler(0f, 0f, 90f);
this.transform.rotation = Quaternion.Slerp(from_qua, to_qua, t);
```

という感じに Quaternion.Slerp の第三引数 t は 0.0～0.1 の範囲で指定します。t=0 なら from_qua に一致し、t=1 なら to_qua に一致します。上記のコードは 1 秒間で 90 度回転するまでの回転の制御ができるようになっています。

■まずは回転で使ってみる「Model03」

回転処理用に新たにクラスを作って処理の比較ができるようにします。

□手順

1. 準備

   Model02 の h、cpp ファイルを複製して、新たに Model03 を準備する。

   QUATERNION.h ファイルをプロジェクトに追加する。

2. 回転

   ※回転用の Quaternion の変数(mRotQ)を作成する。→初期化も必要。

   X 軸回転を mRotQ に合成する。

   Y軸回転を mRotQ に合成する。

   Z軸回転を mRotQ に合成する。

   回転用の Quaternion を、回転行列(mMatRot)に変換する。

■指定軸で指定角度回転するクォータニオンを作成する

```
CreateRotationQuaternion( VECTOR 回転軸 , double 回転値※ラジアン );
   例) VECTOR   rolVector;   // 回転させる軸のベクトル
       double   rolAngle;          // 回転値※ラジアン
       QUARTERNION   q;        // 回転クォータニオン
       q = CreateRotationQuaternion( rolVector,  rolAngle );   // rolVector を rolAngle で回転。
```

■クォータニオンの合成(掛け算)

```
QUATERNION QuaternionMulti( QUATERNION q1, QUATERNION q2 );
   例)QUATERNION   q1;    // クォータニオン 1
      QUATERNION   q2;    // クォータニオン 1
      QUATERNION   q;      // 合成後のクォータニオン
      q = QuaternionMulti( q1,  q2 );    // q1 と q2 を合成した結果を q に入れる。
```

■クォータニオンから回転行列へ変換する

```
MATRIX QuaternionToMatrix( QUATERNION q );
   例) QUATERNION   q;       // 回転クォータニオン
       MATRIX   mRotMat;     // 回転行列
       mRotMat = QuaternionToMatrix( q );   // q を回転行列(mRotMat)に変換する
```

■クォータニオンによる回転(初期化)

ヘッダーに回転管理用のクォータニオン変数を作成します。

```
QUATERNION mRotQ;  // 回転制御用(クォータニオン計算)
```

最初に4つのパラメータを初期化します。

```
// クォータニオンの初期化
mRotQ.x = mRotQ.y = mRotQ.z = 0.0f;
mRotQ.t = 1.0f;
```

※4つ目のtは1.0fにする事

■クォータニオンによる回転(実際の回転)

それぞれの回転軸ての回転状態のクォータニオンを作成し(CreateRotrationQuaternion 命令)、全体管理用のクォータニオン(mRotQ)に QuaternionMulti 命令で合成していきます。

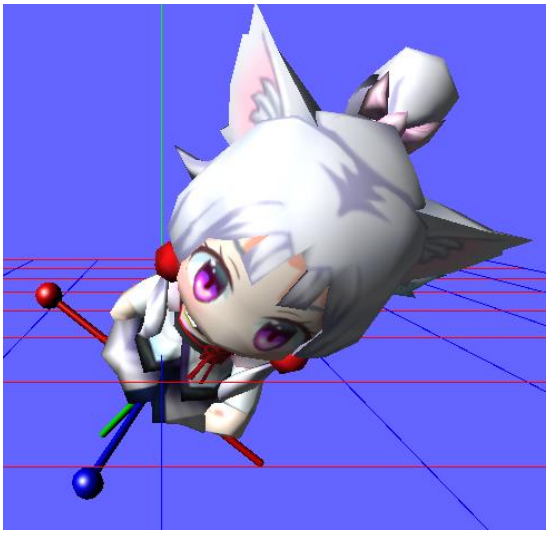
※クォータニオンの良い所は XYZ 軸のどの軸から合成しても大丈夫な所です。

```
// ========== ②回転
// ----- 回転行列をクォータニオンで計算
// X軸回転(pitch)
VECTOR xAxis = VGet(1.0f, 0.0f, 0.0f);  // 回転状態のX軸を取得
mRotQ = QUATERNION::QuaternionMulti(mRotQ, QUATERNION::CreateRotationQuaternion(xAxis, mRol.x));

// Y軸回転(yow)
VECTOR yAxis = VGet(0.0f, 1.0f, 0.0f);  // 回転状態のY軸を取得
mRotQ = QUATERNION::QuaternionMulti(mRotQ, QUATERNION::CreateRotationQuaternion(yAxis, mRol.y));

// Z軸回転(roll)
VECTOR zAxis = VGet(0.0f, 0.0f, 1.0f);  // 回転状態のZ軸を取得
mRotQ = QUATERNION::QuaternionMulti(mRotQ, QUATERNION::CreateRotationQuaternion(zAxis, mRol.z));
```

最後に合成して完成です。

これまでの行列の時と同様に、スケールと移動を加算して最終的な制御行列を作成し、モデルに反映していきます。

```
// ----- 回転MATRIX完成
mMatRot = QUATERNION::QuaternionToMatrix(mRotQ);
```



自由自在！
どんな方向でもグリグリ
動かせます！