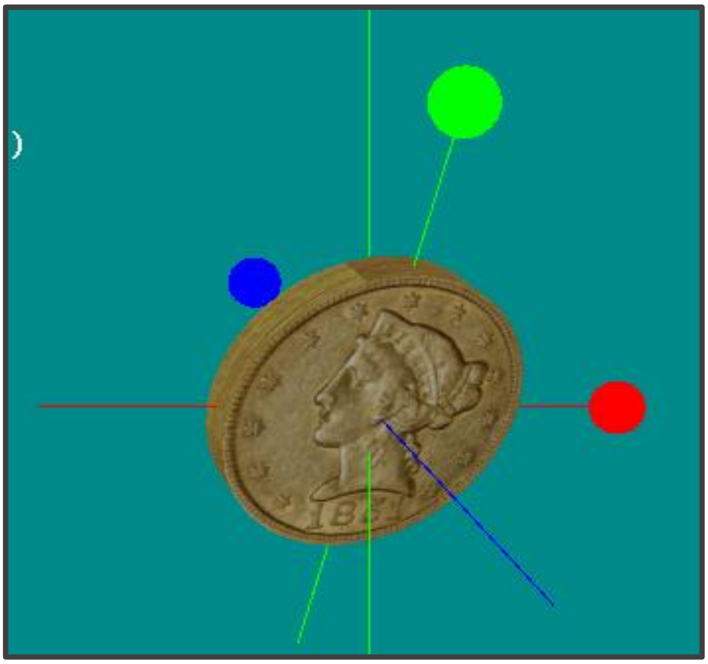
回転の制御方法

これまでは、２Ｄ的な回転制御を、３Ｄモデルに行ってきましたが、
いよいよ３Ｄならではの制御を行っていきます。



まずは、今まで通り、VECTOR型の角度(ラジアン)で、
コインが回転(自転)するようにしてみましょう。

X軸の回転、Y軸の回転、その両方の同時回転も、
きっと上手くいっていることでしょう。

それでは上図のように、コインから見た
XYZの方向に線を描画して、正の方向に球を表示させてみましょう。

Xが、、Yが、、SINが、、COSが、、、と非常に複雑になります。
そこで、行列を登場させます。
行列を使うと、計算がシンプルになりますし、
DxLibには、行列に関する便利な関数が充実しています。

一番の理由は、もっと複雑な平行移動が容易にできるというところに
なりますが、そのあたりの説明は省略させて頂きまして、
便利な点を抑えながら、これから使っていこうと思います。

DxLibのモデル制御の機能で、

MV1SetScale	大きさ
MV1SetRotationXYZ	回転
MV1SetPosition	位置

上記の３種類を使ってきましたが、

MV1SetMatrix

宣言	int MV1SetMatrix(int MHandle, MATRIX Matrix) ;	
概略	モデルの座標変換用行列をセットする	
引数	int	MHandle ： モデルのハンドル
		MATRIX Matrix ： 座標変換用行列
戻り値	０：成功	
	－１：エラー発生	

解説

MHandle のモデルハンドルが示す、
モデルの座標変換用行列をセットします。
この関数は MV1SetPosition 関数や MV1SetScale や
MV1SetRotationXYZ関数などの代わりに行列を使用して
ローカル → ワールド座標変換を行いたい場合に使用します。

この関数に単位行列以外の行列を渡すと、
以後 MV1SetPosition や MV1SetScale 等の
関数の設定は無視され、
MV1SetMatrix 関数で設定した行列のみを使用して
ローカル → ワールド座標変換が行われるようになります。
(解除する場合は MV1SetMatrix 関数に単位行列を渡します)

こちらを使っていきます。

```
// 行列 MATRIX(4次元)によるモデル制御

// 大きさ
mMatScl = MGetScale(SCALE);

// 回転
mMatRot = MGetIdent();

// ローカル調整
mMatRot = MMult(mMatRot, mMatRotLocal);

// 回転の合成
mMatRot = MMult(mMatRot, MGetRotX(mAngles.x));
mMatRot = MMult(mMatRot, MGetRotY(mAngles.y));
mMatRot = MMult(mMatRot, MGetRotZ(mAngles.z));

// 位置
mMatTrn = MGetTranslate(mPos);

// 行列の合成
MATRIX mat = MGetIdent();
mat = MMult(mat, mMatScl);
mat = MMult(mat, mMatRot);
mat = MMult(mat, mMatTrn);

// 行列をモデルに判定
MV1SetMatrix(mModel, mat);
```

慣れないかもしれませんが、
このような一連の作業を行うと、後々楽になります。

この行列方式でいるところの角度情報は、mMatRot行列になります。
この角度情報から、方向に変換する方法として、

```
// モデルの角度から、モデルの前方方向を取得する。
VECTOR forward = VNorm(VTransform({ 0.0f, 0.0f, 1.0f }, mMatRot));
```

このようなやり方があります。

VTransform　　　・・・　　行列を使った座標変換

宣言	VECTOR VTransform(VECTOR InV, MATRIX InM) ;
概略	行列を使ったベクトルの変換
引数	VECTOR InV ： 変換処理を行いたいベクトル MATRIX InM ： 変換処理に使用するベクトル
戻り値	変換後のベクトル
解説	引数 InV のベクトルを引数 InM の行列を使用して 変換処理を行います。 計算的には InV を 1 x 4 行列として扱い (4つめの要素は 1.0f とします) InM の行列の左から 乗算した結果を返します。

```
戻り値.x = InV.x * InM.m[0][0] + InV.y * InM.m[1][0] + InV.z * InM.m[2][0] + InM.m[3][0] ;
戻り値.y = InV.x * InM.m[0][1] + InV.y * InM.m[1][1] + InV.z * InM.m[2][1] + InM.m[3][1] ;
戻り値.z = InV.x * InM.m[0][2] + InV.y * InM.m[1][2] + InV.z * InM.m[2][2] + InM.m[3][2] ;
```

InVに{ 0.0f, 0.0f, 1.0f }を指定することによって、

```
戻り値.x = InV.x * InM.m[0][0] + InV.y * InM.m[1][0] + InV.z * InM.m[2][0] + InM.m[3][0] ;
戻り値.y = InV.x * InM.m[0][1] + InV.y * InM.m[1][1] + InV.z * InM.m[2][1] + InM.m[3][1] ;
戻り値.z = InV.x * InM.m[0][2] + InV.y * InM.m[1][2] + InV.z * InM.m[2][2] + InM.m[3][2] ;
```

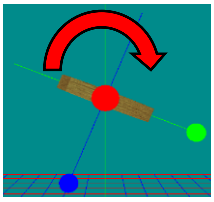
XYの成分がごっそり消えて、Z(手前、奥行)が残ります。
それを正規化することで、単位ベクトル、方向を取り出すことができます。
同じ要領で、

```
VECTOR up = VNorm(VTransform({ 0.0f, 1.0f, 0.0f }, mMatRot));
```

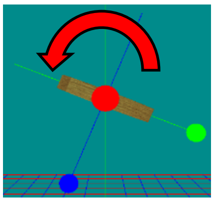
とすると、モデルの上方向を取り出すことができます。

回転、拡大縮小、平行移動がサポートされ、
とても便利になりますが、回転行列には弱点があります。

ジンバルロックといって、
オイラー角表現において、2つの回転軸が同じ向きになってしまったとき
回転の自由度が1つ落ちてしまうことです。
再現してみましょう。



```
// ジンバルロックの再現
//mAngles.x += AsoUtility::Deg2RadF(1.0f);
//mAngles.y = AsoUtility::Deg2RadF(90.0f);
//mAngles.z = AsoUtility::Deg2RadF(0.0f);
```



```
// 同じ軸の回転になってしまう
//mAngles.x = AsoUtility::Deg2RadF(0.0f);
//mAngles.y = AsoUtility::Deg2RadF(90.0f);
//mAngles.z += AsoUtility::Deg2RadF(1.0f);
```

これを解決するのが、クォータニオンです。
私も良くわかっていませんが、複素数や虚数などを用いて、
３Ｄの回転や姿勢制御を良い感じしてしてくれる計算です。

クォータニオンの中身は省略させて頂いて。。。
使って、慣れていきたいと思います。

```
// 行列＆クォータニオンによるモデル制御

// 大きさ
mMatScl = MGetScale(SCALE);

// 回転(ジンバルロック解消)
Quaternion tmpQ = Quaternion::Euler(mAngles.x, mAngles.y, mAngles.z);

// ローカル調整
tmpQ = tmpQ.Mult(mQRotLocal);

// 行列に変換
mMatRot = Quaternion::ToMatrix(tmpQ);

// 位置
mMatTrn = MGetTranslate(mPos);

// 行列の合成
MATRIX mat = MGetIdent();
mat = MMult(mat, mMatScl);
mat = MMult(mat, mMatRot);
mat = MMult(mat, mMatTrn);

// 行列をモデルに判定
MV1SetMatrix(mModel, mat);
```

オイラー角を、いっぺんに計算して、回転情報を作ってくれます。
しかし、DxLibは回転行列でモデルを制御しますので、
クォータニオンを行列に変換して、DxLibに渡してあげます。

これで、ジンバルロックは解消されるはずです。

とはいえ、元となっているのが、オイラー角ですので、
フルクォータニオンで実装する方法もやってみましょう。