

3 D表現について

1 . 概要

- ワールド座標系 絶対的な座標
- オブジェクト座標系 物体を中心にした座標
- カメラ座標系 視点を中心にした座標

[回転行列 / 移動行列]

- R_x X軸回転
- R_y Y軸回転
- R_z Z軸回転
- T 平行移動

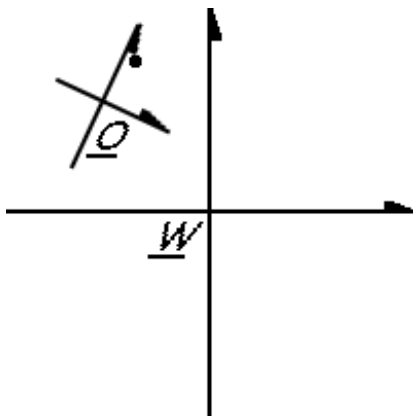
これら4つの行列を利用しやすいように1つの式に合成します。

$$C = R_x R_y R_z T$$

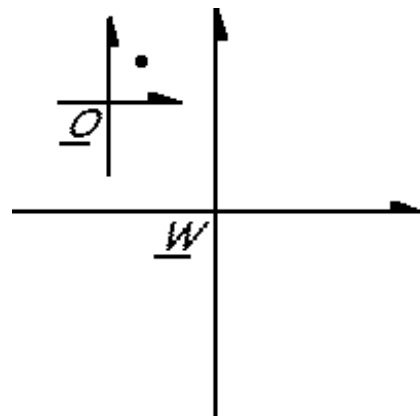
これを各座標間の変換行列とします。行列をかける順序については後述します。

2 . 座標変換

物体を表す座標値はオブジェクト座標系上で指定されます。この値に変換行列 O をかけてワールド座標系上の値にします。さらに変換行列 C をかけてカメラ座標系上の値に変換します。そしてその座標値をもとにスクリーンへ投影することによって画像を描画します。変換行列 O や C は、そのときのオブジェクト座標系とワールド座標系の傾きやワールド座標系とカメラ座標系の傾きなどによって逐次設定されます。



(図 1 _ 1)



(図 1 _ 2)

座標変換の過程を2次元図面上で説明します。まず前ページの図1__1をご覧ください。O座標系がW座標系にたいして傾いています。これをそれぞれのX軸、Y軸が平行となるようにするには、O座標系の軸を回転させる必要があります。図1__1では左側に 度 (< 90) 回転させれば良さそうです。ここで重要なのは「座標系の軸を回転する」ということは「座標系上の全ての点に原点回りの逆回転を施す」のに等しいことです。つまり図1__1から図1__2の状態へ移行したさい、図中の点の位置はO座標系上で右側に 度回転したことになります。

続いてO座標系の原点をW座標系の原点に重ねます。X軸方向へ a 、Y軸方向へ b 移動したとすると、図中の点のO座標系上の位置はX軸方向へ $-a$ 、Y軸方向へ b だけ移動します。このときO座標系はW座標系と完全に一致しているため、O座標系上の点の位置 = W座標系上の点の位置となり、座標変換の完了です。

3 . スクリーンへの投影

カメラはカメラ座標系上の原点でZ軸（奥行）方向を向いているものとします。

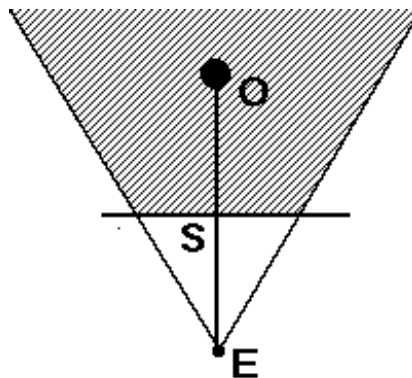


図2__1

まず図2__1を見ていただくと、E、S、Oの3点があります。これらはそれぞれE（視点）、S（スクリーン）、O（物体）を示しています。PCのディスプレイ上にはSに投影された画像を描画することになります。SとEの距離により視野（斜線部）が変わってきます。実際に描画する仕組みは、物体Oが属するSに平行な平面S'とEとの距離EOと視点からスクリーンまでの距離ESから投影用の関数 $t(a) = a * ES / EO$ を求め、各座標値にそれを適用します。冒頭に書いたようにカメラ方向はZ軸方向に平行なので $t(a)$ はZ座標値を用いて求められ、それらをX、Y座標値にかけたものをスクリーン上のX、Y座標値とします。

4 . 逆変換

2 . で述べた変換行列は、[オブジェクト座標系 ワールド座標系 カメラ座標系]と言ったような一方的な変換しか出来ませんでした。状況によってはワールド座標系の座標値をオブジェクト座標系で使いたいと言うようなこともあります。それを実現するにはそれぞれの変換行列の逆行列を使います。ここで注目すべき点は、回転行列が正規直交なベクトルの集まりだということです。正規直交なベクトル群によって構成される行列は転置行列と逆行列が一致します。

【証明】

お互いに直交するベクトル群を

$$Z = \{ Z_1, Z_2, \dots, Z_i, \dots, Z_n \} \quad (Z_i \text{ は列ベクトル})$$

とする。Z の転置行列は、

$$Z^T = \{ Z'_1, Z'_2, \dots, Z'_i, \dots, Z'_n \}^T \quad (Z'_i \text{ は行ベクトル})$$

となる。これらをかけ合わせると、

$$Z^T Z = \begin{matrix} Z'_1 Z_1 & , & \dots & , & Z'_1 Z_n \\ Z'_2 Z_1 & , & \dots & , & Z'_2 Z_n \\ \vdots & & & & \vdots \\ Z'_i Z_1 & , & \dots & , & Z'_i Z_n \\ \vdots & & & & \vdots \\ Z'_n Z_1 & , & \dots & , & Z'_n Z_n \end{matrix}$$

$$Z^T Z =$$

$$\begin{matrix} Z'_1 Z_1 & , & \dots & , & Z'_1 Z_n \\ Z'_2 Z_1 & , & \dots & , & Z'_2 Z_n \\ \vdots & & & & \vdots \\ Z'_i Z_1 & , & \dots & , & Z'_i Z_n \\ \vdots & & & & \vdots \\ Z'_n Z_1 & , & \dots & , & Z'_n Z_n \end{matrix}$$

各ベクトルは直交なので、

$$Z'_i Z_j = 0 \quad (i \neq j)$$

さらに正規なので各ベクトルのノルムは、

$$Z'_i Z_i = 1$$

となり、

$$Z'_i Z_i = 1$$

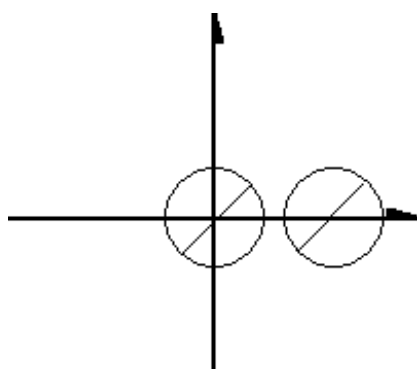
従って

$$Z^T Z = I_n$$

$$Z^T = Z^{-1}$$

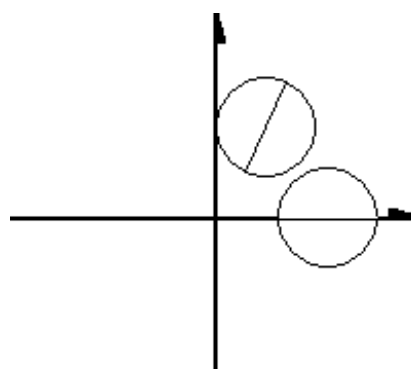
となって転置行列と逆行列が一致することが証明された。

これで逆回転は可能になりましたが、移動行列は正規直交の条件を満たしませんので別の方法を使います。そもそも移動行列の逆変換はx、y、zの移動分に負をかけた方向へ移動することによって行うことができます。ここで重要なのが回転と移動の順序です。



回転→移動

図 4 __ 1



移動→回転

図 4 __ 2

上の図は原点を起点として回転、移動の操作を順序を入れ替えて実行したものです。ご覧のとおり異なる結果が出ました。このようなことになるのは、回転の操作は原点中心に行われているからです。したがって逆変換のさいには、逆移動 逆回転の順序で行います。

5 . プログラム

実際にプログラムで動かすにはどうするかを如何に示します。以下のコードは3点の頂点情報を入力として、それらで構成される面を描画するメソッドです。

6 . 参考文献

<http://www.cc.rim.or.jp/~devilman/3dCoding/3dCoding.html>

```

public void Draw( Face f ) {
    //初期設定
    ver[0] = change(f.a);
    ver[1] = change(f.b);
    ver[2] = change(f.c);
    ver[0] = toei(ver[0]);
    ver[1] = toei(ver[1]);
    ver[2] = toei(ver[2]);
    snum = 0;
    maxy = ver[0].y; miny = ver[0].y;
    for ( int i = 0; i <= 2; i++ ) {
        if ( maxy < ver[i].y ) { maxy = ver[i].y; }
        if ( miny > ver[i].y ) { miny = ver[i].y; snum = i; }
    }
    anum = snum;
    bnum = snum;
    sy = miny;
    ey = maxy;

    //最初の頂点とその隣の頂点らから増分値を求める
    do {
        anum++;
        if ( anum > 2 ) { anum = 0; }
    } while( ver[anum].y < sy );
    snum = anum - 1;
    if ( snum < 0 ) { snum = 2; }
    dd = ( ver[anum].y - ver[snum].y ) >= 1 ? ver[anum].y - ver[snum].y :
1;

    dxL = ( ver[anum].x - ver[snum].x ) / dd;
    duL = ( ver[anum].u - ver[snum].u ) / dd;
    dvL = ( ver[anum].v - ver[snum].v ) / dd;
    x1 = ver[snum].x;
    z1 = ver[snum].z;
    u1 = ver[snum].u;
    v1 = ver[snum].v;

```

```

do {
    bnum--;
    if ( bnum < 0 ) { bnum = 2; }
} while( ver[bnum].y < sy );
snum = bnum + 1;
if ( snum > 2 ) { snum = 0; }
dd = ( ver[bnum].y - ver[snum].y ) >= 1 ? ver[bnum].y - ver[snum].y :
1;

dxR = ( ver[bnum].x - ver[snum].x ) / dd;
duR = ( ver[bnum].u - ver[snum].u ) / dd;
dvR = ( ver[bnum].v - ver[snum].v ) / dd;
x2 = ver[snum].x;
z2 = ver[snum].z;
u2 = ver[snum].u;
v2 = ver[snum].v;

//y を 1 つずつ増やししながら増分値に基づいて x 間を補間していく
for ( y = (int)sy; y < (int)ey; y++ ) {
    if ( ver[anum].y <= y ) {
        do {
            anum++;
            if ( anum > 2 ) { anum = 0; }
        } while( ver[anum].y <= y );
        snum = anum - 1;
        if ( snum < 0 ) { snum = 2; }
        dd = ( ver[anum].y - ver[snum].y ) >= 1 ? ver[anum].y
- ver[snum].y : 1;

        dxL = ( ver[anum].x - ver[snum].x ) / dd;
        dzL = ( ver[anum].z - ver[snum].z ) / dd;
        duL = ( ver[anum].u - ver[snum].u ) / dd;
        dvL = ( ver[anum].v - ver[snum].v ) / dd;
        x1 = ver[snum].x;
        z1 = ver[snum].z;
        u1 = ver[snum].u;
        v1 = ver[snum].v;
    }
}

```

```

if ( ver[bnum].y <= y ) {
    do {
        bnum--;
        if ( bnum < 0 ) { bnum = 2; }
    } while( ver[bnum].y <= y );
    snum = bnum + 1;
    if ( snum > 2 ) { snum = 0; }
    dd = ( ver[bnum].y - ver[snum].y ) >= 1 ? ver[bnum].y
- ver[snum].y : 1;

    dxR = ( ver[bnum].x - ver[snum].x ) / dd;
    dzR = ( ver[bnum].z - ver[snum].z ) / dd;
    duR = ( ver[bnum].u - ver[snum].u ) / dd;
    dvR = ( ver[bnum].v - ver[snum].v ) / dd;
    x2 = ver[snum].x;
    z2 = ver[snum].z;
    u2 = ver[snum].u;
    v2 = ver[snum].v;
}

double du=0.1, dv=0.1;

double xx1 = x1, xx2 = x2; //x1 が変化しないように xx を使う
double xu1 = u1, xu2 = u2;
double xv1 = v1, xv2 = v2;
double tmp;
if ( x1 > x2 ) {
    tmp = xx1; xx1 = xx2; xx2 = tmp;
    tmp = xu1; xu1 = xu2; xu2 = tmp;
    tmp = xv1; xv1 = xv2; xv2 = tmp;
}
double wid = x2 - x1;
if ( wid > 0 ) {
    du = ( xu2 - xu1 )/wid;
    dv = ( xv2 - xv1 )/wid;
}
ix1 = (int)xx1; ix2 = (int)xx2;

```

```

int tx, ty;
int u, v;
for ( int x = ix1; x < ix2; x++ ) {
    u = (int)(xu1*Tex_W);
    v = (int)(xv1*Tex_H);
    tx = u%Tex_W;
    ty = v%Tex_H;
    ty = Tex_H - ty -1;

    int ppp = 50*ty+tx-1;
    if ( ppp > Tex_W*Tex_H-1 ) { ppp = Tex_W*Tex_H-1; } else
if ( ppp < 0 ) { ppp = 0; }

    int pixel = tbuf[ppp];
    int red   = (pixel >> 16) & 0xff;
    int green = (pixel >> 8) & 0xff;
    int blue  = (pixel) & 0xff;
    if ( -250 < x && x < 250 && -250 < y && y < 250 ) {
        pbuf[500*(y+CenterY)+x+CenterX] = (red<<16)
| (green<<8) | blue | 0xff000000;
    }

    xu1 += du; xv1 += dv;
}
x1 += dxL; x2 += dxR;
u1 += duL; u2 += duR;
v1 += dvL; v2 += dvR;
}
}

```