3D の回転計算

3T 河内仁志

3D をはじめてからそろそろ 2 年になるが, われらが X680x0 同好会にはノウハウが少ないようである. そこで, 分かりづらい回転の計算に焦点を置いて, 分かりやすい資料 (となれば良いが) を残していこうと思う.

1 座標系

コンピュータ 3D における座標系はライブラリに依存するが, ある程度一般化がなされているようである. お手軽な DX ライブラリの 3D と日本ではあまり資料の少ない Irrlicht の両方で左手系が使われているようである. 一方拡張現実で有名な AR ツールキットでは右手系が使われている. これは DirectX は左手系が, OpenGL は右手系が使われているためのようだ.

座標系といっても右手系から左手系への変換は容易で, $z' = -z$ とすれば良いだけなので, ややこしく考える必要は無い. 分からなくなったのならば, 実際に計算してしまえば良いだけだからだ.

以降は主に左手系で話を進める.

2 カルダン角による回転

3D ライブラリでよく使われている回転は, カルダン角を使った方法である. カルダン角とはオイラー角の一種であり, X 軸中心の回転 →Y 軸中心の回転 →Z 軸中心の回転を行って回転する方法のことをいう.

回転角について左手系の場合は, 各軸に対して左ねじ方向に回転を行う方向を, 正とするのが慣例となっているようだ.

実際に X 軸中心の回転を行うと次のようになる. ただし, (x, y, z) から (x', y', z') の座標の変換を考えるとし, θ_x は X 軸中心の回転角を表すとする.

$$x' = x \tag{1}$$
$$y' = y\cos\theta_x - z\sin\theta_x \tag{2}$$
$$z' = y\sin\theta_x + z\cos\theta_x \tag{3}$$

この (1)〜(3) 式は行列でまとめた方が良いのでベクトルと行列で表すと,

$$(x'\ y'\ z') = (x\ y\ z)R_x = (x\ y\ z)\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x \\ 0 & -\sin\theta_x & \cos\theta_x \end{bmatrix} \tag{4}$$

となる.

同様に Y 軸中心の回転, Z 軸中心の回転を行列表記すると,

$$(x'\ y'\ z') = (x\ y\ z)R_y = (x\ y\ z)\begin{bmatrix} \cos\theta_y & 0 & -\sin\theta_y \\ 0 & 1 & 0 \\ \sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \tag{5}$$

$$(x'\ y'\ z') = (x\ y\ z)R_z = (x\ y\ z)\begin{bmatrix} \cos\theta_z & \sin\theta_z & 0 \\ -\sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

となる.

カルダン角は X 軸中心の回転 →Y 軸中心の回転 →Z 軸中心の回転であるため, 次の式で表現できる.

$$(x'\ y'\ z') = (x\ y\ z)R = (x\ y\ z)R_xR_yR_z \tag{7}$$

これらは通常の線形写像の計算方法と多少異なっていることに注意しなければならない. 通常の線形写像では $(x\ y\ z)^{\mathrm{T}} = \vec{v}$, $(x'\ y'\ z')^{\mathrm{T}} = \vec{v'}$ とし行列 A で変換すると,

$$\vec{v'} = A\vec{v} \tag{8}$$

である (ただし v の転置行列を v^{T} とする).

$(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$ であり, $A\vec{v'} = (\vec{v}^{\mathrm{T}}A^{\mathrm{T}})^{\mathrm{T}}$ である (この場合 $(x\ y\ z)^{\mathrm{T}} = \vec{v}$, $(x'\ y'\ z')^{\mathrm{T}} = \vec{v'}$) ため, (7) 式は次のように書き換えられる.

$$\vec{v'} = R^{\mathrm{T}}\vec{v} = R_z^{\mathrm{T}}R_y^{\mathrm{T}}R_x^{\mathrm{T}}\vec{v} \tag{9}$$

実際にこの計算式を用いて計算しているライブラリも存在するが, 今回は転置が面倒であるので式 (7) を用いることにする. とはいえ, 式 (7) も式 (9) も数値自体に変化は無い.

3 四元数による回転

四元数 (quaternion) は最初は良くわからないものだと思うし, 実際に数学的な意味は奥深いものが有る. だが今回はそういったことを省き, 回転のみで簡潔に要点を追って行くことにする.

まず四元数とは $q = s + iu + jv + kw$ という複素数のような数字である. このときの s, u, v, w は実数で, i, j, k には以下のような計算規則が存在する.

$$
\begin{aligned}
i^2 &= j^2 = k^2 = -1 & (10)\\
ij &= -(ji) = k & (11)\\
jk &= -(kj) = i & (12)\\
ki &= -(ik) = j & (13)\\
ijk &= -1 & (14)
\end{aligned}
$$

これら規則を用い, 四元数同士の掛け算を行おう. $q_1 = s_1 + iu_1 + jv_1 + kw_1$, $q_2 = s_2 + iu_2 + jv_2 + kw_2$, $q_3 = s_3 + iu_3 + jv_3 + kw_3$ とすると $q_3 = q_1 q_2$ のとき,

$$
\begin{aligned}
s_3 &= s_1 s_2 - u_1 u_2 - v_1 v_2 - w_1 w_2 & (15)\\
u_3 &= s_1 u_2 + u_1 s_2 + v_1 w_2 - w_1 v_2 & (16)\\
v_3 &= s_1 v_2 - u_1 w_2 + v_1 s_2 + w_1 u_2 & (17)\\
w_3 &= s_1 w_2 + u_1 v_2 - v_1 u_2 + w_1 s_2 & (18)
\end{aligned}
$$

四元数はライブラリ内ではよく $q = w + ix + jy + kz = w + \vec{v}$, $q = (w; x\ y\ z) = (w; \vec{v})$ と表記される. 以後はこの表記を用いることにする[1]. また, $q = w + \vec{v}$, $q' = w' + \vec{v'}$ とすると,

$$
qq' = (ww' - \vec{v}\vec{v'}) + (w\vec{v'} + w'\vec{v} + \vec{v} \times \vec{v'}) \quad (19)
$$

と書ける.

そして, 四元数には複素数と同じように共役四元数が存在する. 四元数 q の共役四元数を $\bar{q}$ とすると,

$$
\begin{aligned}
\bar{q} &= w - ix - jy - kz & (20)\\
q\bar{q} &= w^2 + x^2 + y^2 + z^2 & (21)\\
|q| &= \sqrt{q\bar{q}} = \sqrt{w^2 + x^2 + y^2 + z^2} & (22)
\end{aligned}
$$

となる. この共役四元数は後のベクトルを回転する際に使用するので覚えておくこと.

さて, ようやく四元数を用いて回転を行うことにする.

まず $(x_r\ y_r\ z_r)$ を回転軸にし, 左ねじ方向に θ 回転させる四元数 q_r を次のように定義する.

$$
\begin{aligned}
q_r &= \sin\frac{\theta}{2} + ix_r\cos\frac{\theta}{2} + jy_r\cos\frac{\theta}{2} + kz_r\cos\frac{\theta}{2} & (23)\\
&= \left(\sin\frac{\theta}{2}; x_r\cos\frac{\theta}{2}\ y_r\cos\frac{\theta}{2}\ z_r\cos\frac{\theta}{2}\right) & (24)
\end{aligned}
$$

[1] なぜこの表記が良いかは後述を参考.

この四元数を使ってベクトルを回転させるのは簡単で, 次のように表すことが出来る.

$$q_v{}' = q_r q_v \bar{q}_r \tag{25}$$

ただし, $q_v = (0; v) = (0; x\ y\ z)$, $q_v{}' = (0; v') = (0; x'\ y'\ z')$ と定義される.
これで v を回転したベクトル v' を求めることが出来る.
この様子を実際に行うプログラムを R 言語で記述すると次のようになる.

ソースコード 1: Rquaternion.txt

```
radtoqu <- function(r, v)
{
  # ベクトルvを回転軸としてr(rad)回転
  cr <- cos(r/2)
  sr <- sin(r/2)
  Q <- c(cr, v[1]*sr, v[2]*sr, v[3]*sr)
  return(Q)
}

quprod <- function(q, r)
{
  # 四元数同士の掛け算
  S <- q[1]*r[1]-q[2]*r[2]-q[3]*r[3]-q[4]*r[4]
  S <- c(S, q[1]*r[2]+q[2]*r[1]+q[3]*r[4]-q[4]*r[3])
  S <- c(S, q[1]*r[3]+q[3]*r[1]+q[4]*r[2]-q[2]*r[4])
  S <- c(S, q[1]*r[4]+q[4]*r[1]+q[2]*r[3]-q[3]*r[2])
  return(S)
}
rollqu <- function(q, r)
{
  # ベクトルが元である四元数qを回転の四元数rを使って回転する
  Q <- quprod(r, q)
  R <- c(r[1], -r[2], -r[3], -r[4])
  return(quprod(Q, R))
}

# 実行
Q1 <- radtoqu(pi/10, c(0, 0, 1))
Q2 <- radtoqu(pi/10, c(0, 1, 0))
x <- 1
y <- 0
z <- 0
i <- 1
while(i<=10){
  Q.tmp <- rollqu(c(0, x[i], y[i], z[i]), Q1)
  x <- c(x, Q.tmp[2])
  y <- c(y, Q.tmp[3])
  z <- c(z, Q.tmp[4])
  i <- i+1
}
while(i<=20){
  Q.tmp <- rollqu(c(0, x[i], y[i], z[i]), Q2)
  x <- c(x, Q.tmp[2])
  y <- c(y, Q.tmp[3])
  z <- c(z, Q.tmp[4])
  i <- i+1
}
# 三次元用plot(パッケージインストールが必要)
library(scatterplot3d)
scatterplot3d(x, z, y, angle=20, col.grid="white", pch=16, color=grey(0:20/25))
```

この結果は次のようになる.

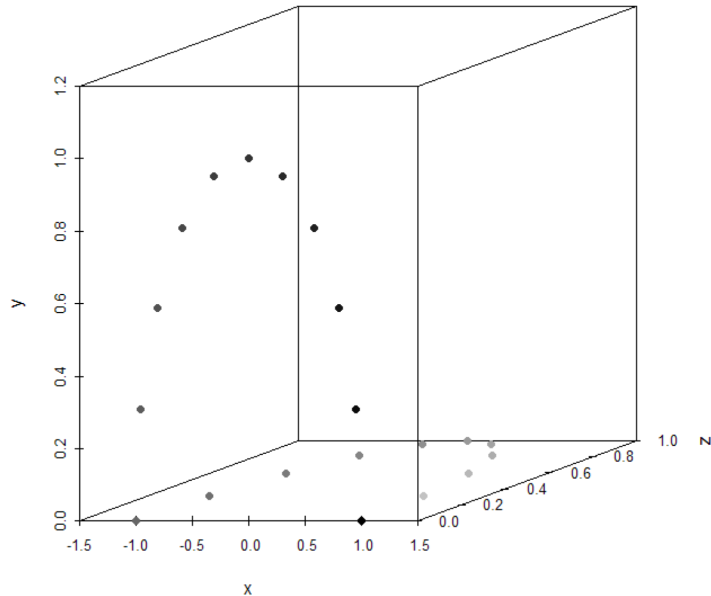

図 1: 四元数の座標移動

初めの点は黒く, だんだんと薄くなっていっているのが分かるだろうか. きちんと Z 軸中心での回転と, Y 軸中心での回転が行われているのが見えるだろう. 両方とも左ねじ方向に回転している. ちなみに, このコードでは左手系の座標系にするためにプロット時に y と z を入れ替えている.

4 任意軸回転行列による回転

カルダン角と同様な行列による計算を行い, 四元数と同様に任意のベクトルを回転軸に取ることができるという, 両方の良いとこ取りをしたようなものがある.

計算方法はさておき, この回転行列を示そう. $(x_r\ y_r\ z_r)$ を回転軸にし, 左ねじ方向に θ 回転させる行列は次のようになる.

$$R = \begin{bmatrix} {x_r}^2(1-\cos\theta)+\cos\theta & x_r y_r(1-\cos\theta)+z_r\sin\theta & x_r z_r(1-\cos\theta)+y_r\sin\theta \\ x_r y_r(1-\cos\theta)-z_r\sin\theta & {y_r}^2(1-\cos\theta)+\cos\theta & y_r z_r(1-\cos\theta)+x_r\sin\theta \\ x_r z_r(1-\cos\theta)+y_r\sin\theta & y_r z_r(1-\cos\theta)-x_r\sin\theta & {z_r}^2(1-\cos\theta)+\cos\theta \end{bmatrix} \quad (26)$$

このときの行列 R は式 (7) に代入する.

この様子を実際に行うプログラムを R 言語で記述すると次のようになる.

ソースコード 2: ARM.txt

```
radtomat <- function(r, v)
{
  # ベクトルvを回転軸としてr(rad)回転
  sr <- sin(r)
  cr <- cos(r)

  ARM <- matrix(0,3,3)
  ARM[1,1] <- v[1]^2 * (1-cr) + cr
  ARM[2,1] <- v[1]*v[2] * (1-cr) - v[3]*sr
  ARM[3,1] <- v[1]*v[3] * (1-cr) + v[2]*sr
  ARM[1,2] <- v[1]*v[2] * (1-cr) + v[3]*sr
  ARM[2,2] <- v[2]^2 * (1-cr) + cr
  ARM[3,2] <- v[2]*v[3] * (1-cr) - v[1]*sr
  ARM[1,3] <- v[1]*v[3] * (1-cr) - v[2]*sr
  ARM[2,3] <- v[2]*v[3] * (1-cr) + v[1]*sr
  ARM[3,3] <- v[3]^2 * (1-cr) +cr

  return(ARM)
}

# テストコード
M1 <- radtomat(pi/10, c(0, 0, 1))
M2 <- radtomat(pi/10, c(0, 1, 0))
x <- 1
y <- 0
z <- 0
i <- 1
while(i<=10){
  v <- matrix(0,1,3)
  v <- c(x[i], y[i], z[i])
  M.tmp <- v%*%M1
  x <- c(x, M.tmp[1,1])
  y <- c(y, M.tmp[1,2])
  z <- c(z, M.tmp[1,3])
  i <- i+1
}
while(i<=20){
  v <- matrix(0,1,3)
  v <- c(x[i], y[i], z[i])
  M.tmp <- v%*%M2
  x <- c(x, M.tmp[1,1])
  y <- c(y, M.tmp[1,2])
  z <- c(z, M.tmp[1,3])
  i <- i+1
}
# 三次元用plot(パッケージインストールが必要)
library(scatterplot3d)
scatterplot3d(x, z, y, angle=20, col.grid="white", pch=16, color=grey(0:20/25))
```

この結果は図 1 と一致した.

5 実際に C++で実装する

単純な回転を行うときにはカルダン角を用いることが便利である. これは, 各種ライブラリに付随するサンプルの中身から見ても, そういう書き方をしているものが多いことからも確かである.

あくまで一例だが, X-Z 平面上から離れない場合は次のように簡単に書くことができる (DX ライブラリを用いた場合).

ソースコード 3: Sample1.cpp

```
VECTOR pos, cardan;
float v, r;

pos = VGet(0.0f, 0.0f, 0.0f);
cardan = VGet(0.0f, 0.0f, 0.0f);
v = 1.0f;
r = 0.05f

while((dxlib_clearscreen() == 0)&&(CheckHitKey(KEY_INPUT_ESCAPE) == 0)){
  // 進行方向を計算し，速度から座標を加算
  pos = VAdd(VGet(-v*sin(cardan.y), 0, -v*cos(cardan.y)));
  cardan = VGet(0, cardan.y+r, 0);
  // 座標と角度を設定
  MV1SetPosition(MHandle, pos);
  MV1SetRotationXYZ(MHandle, cardan);
  // 描画
  MV1DrawModel(MHandle);
}
```

このようにプログラムの仕様上楽に作成できる場合は良いのだが, 完全に自由な回転をさせないとならない場合は任意軸回転行列で一括に処理するほうが良い.

次は任意軸回転行列を用いて回転を行う場合を提示する.

ソースコード 4: Sample2.cpp

```
VECTOR Axis;      // 回転軸
float rot;        // 回転角(rad表記)
float FrontV;     // 前方向のスピード
VECTOR Pos;       // 座標
MATRIX Matr;      // 回転するための行列

Pos = VGet(0.0f, 0.0f, 0.0f);
FrontV = 10.0f;
Axis = VGet(0.0f, 1.0f, 0.0f);
rot = 0.05f;

while((dxlib_clearscreen() == 0)&&(CheckHitKey(KEY_INPUT_ESCAPE) == 0)){
  // 前方向のベクトルを計算
  MATRIX tempM = MV1GetMatrix(MHandle);
  VECTOR tempv = VNorm(VTransformSR(VGet(0.0f, 0.0f, -1.0f), tempM));
  Pos = VAdd(Pos, VScale(tempv, FrontV));
  // 任意軸回転行列の生成
  Matr = MGetRotAxis(Axis, rot);
  // 座標・角度反映
  mmultiple(tempM, Matr);
  tempM.m[3][0] = Pos.x;
  tempM.m[3][1] = Pos.y;
  tempM.m[3][2] = Pos.z;
  MV1SetMatrix(MHandle, tempM);
  // 描画
  MV1DrawModel(MHandle);
}
```

DX ライブラリにおける行列用 (MATRIX 構造体) の配列の対応は次のようになっている.

$$M = \begin{bmatrix} m[0][0] & m[0][1] & m[0][2] & m[0][3] \\ m[1][0] & m[1][1] & m[1][2] & m[1][3] \\ m[2][0] & m[2][1] & m[2][2] & m[2][3] \\ m[3][0] & m[3][1] & m[3][2] & m[3][3] \end{bmatrix} \tag{27}$$

上記のように 4×4 行列なのだが, これまでの回転行列の説明では 3×3 行列で行っていた. そこで, 補足として 3D で使われる変換行列の扱いを説明する.

回転行列と同様に変換は次のように行われる.

$$(x' \; y' \; z' \; 1) = (x \; y \; z \; 1)M = (x \; y \; z \; 1)M_S M_R M_P \tag{28}$$

ここで注意しなければならないのは, ベクトルの第 4 成分に必ず 1 が入り, 4×4 行列が用いられるということである. このとき, M_S はスケーリング (拡大縮小) 用行列を M_R は回転用行列を, M_P は平行移動用行列を表す. 各行列の内容は次のようになっている.

$$M_S = \begin{bmatrix} x_S & 0 & 0 & 0 \\ 0 & y_S & 0 & 0 \\ 0 & 0 & z_S & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

$$M_R = \begin{bmatrix} R[0][0] & R[0][1] & R[0][2] & 0 \\ R[1][0] & R[1][1] & R[1][2] & 0 \\ R[2][0] & R[2][1] & R[2][2] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{30}$$

$$M_P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_P & y_P & z_P & 1 \end{bmatrix} \tag{31}$$

先のコード 4 の 21〜23 行目での行列成分への直接代入はこの計算を利用している.

6　最後に

3D としては簡単な回転 (回転軸が変化しないなど) を扱う場合は, ライブラリに存在するカルダン角を使った回転で十分である. だが, 複雑な回転 (角運動量保存則を用いるなど) を導入せざるを得ない場合は, 任意軸回転行列を用いる必要性があるだろう. 特に, 角速度ベクトルによって回転を行わせる場合 (角運動量は角速度ベクトルと慣性テンソルの行列計算によって求まり, 外力が無い場合は保存される) は任意軸回転行列がそのまま使えるため, 他の回転方式を採用するより比較的楽になるだろう.

では, 四元数は使わないのかというと, そうでもない. 二つの回転状態の差を四元数で表すことが出来たならば, 回転角を一定間隔で区切ることで滑らかに間を補完することができる. この「線形補完」を行うことが出来るのが四元数の最大の魅力である. 他の表現方法では, 差の回転状態を等間隔で区切っても回転状態が等間隔とならず, 非線形なものになってしまう. 3D アニメーションの補完には, 無くてはならないものなのだ.

表現方法のそれぞれの違いと強みを理解したうえで, 選択していって欲しい.