モデル座標系と行列による変換

竹内 亮太

1　はじめに

3DCG プログラミングにおいて避けて通れない要素のひとつに「座標系」の問題がある。座標系とは空間全体の x, y, z 各軸によって構成される「グローバル座標系」と、個々の形状を描画する際に用いられる「ローカル座標系」に大別され、多くの 3DCG システムにおいてはこれらを行列によって管理し、演算に使用する。

FK の場合は fk_Model クラスが座標系の管理を担っており、ある程度のアプリケーションならば行列を意識せずに構築することが可能だが、ゲームキャラクター同士の接触判定や、モデラーにおける操作地点と形状要素の選択判定などにおいては行列を利用した演算が必要になる場合が多い。またモデルの親子関係を用いた場合には、特定階層の座標系における変換座標を得たいケースもあるだろう。

本稿では 3DCG における座標変換の基礎を今一度整理し、FK を用いたプログラミングにおける効率的な座標変換処理のノウハウをまとめたい。

2　座標変換の基礎

3DCG で形状を描画する場合、形状ごとにまとめて平行移動・回転・拡大縮小の操作を行うことが多い。これらの操作は、各頂点の座標と座標変換を表す行列の乗算によって実現できる。座標変換行列は順序を守って乗算していくことで合成することが可能である。詳しくは「コンピュータグラフィクスの基礎理論」第 9,10 回を参照して欲しい。

FK システムの場合は、fk_Model のオブジェクトに対して平行移動や回転を指示するだけで、セットした形状を自由な位置に、自由な姿勢で、自由な大きさによって空間内に配置することができる。これは、fk_Model のオブジェクトに対する操作を全て行列に変換して保持することにより実現されている。すなわち、実際にグローバル座標上に配置される形状の座標は以下の式によって求められる。

$$\mathbf{P_G} = \mathbf{M}\mathbf{P_L} \tag{1}$$

$\mathbf{P_L}$ は形状データが持つ頂点座標、$\mathbf{P_G}$ が実際に配置されるグローバル座標を表す。$\mathbf{M}$ は fk_Model が保持している位置・姿勢・拡大率を表している行列である。この行列は fk_Model に対する操作に応じて更新され、平行移動 × 回転 × 拡大縮小 の順に乗算することで合成されている。この行列上で管理されている座標系に従うと、そのモデルがどの方向を向いているかに関わらず、そのモデルにとっての前方向は $-z$ 方向、右方向は $+x$ 方向、上方向は $+y$ 方向を指すようになる。モデルのローカル座標と呼んでいたものは、実はこの行列によって構築されていたものである。

描画する形状のデータ自体は、fk_Model の移動・回転操作で変更されることはない。つまり、形状データから得られる頂点座標などは「ローカル座標系」のものであると言える。これに対して、fk_Model::getPosition() で得られる座標値は「グローバル座標系上における」ローカル座標の中心位置 $(0, 0, 0)$ である。

このような状態であるため、グローバル座標とローカル座標を比較したり、異なるモデル同士のローカル座標を比較することには「全くもって意味がない」ことに注意して欲しい。もしそのような処理を行ってたまたまうまく動いていたとしても、それはモデルを動かしたりしていなかったため偶然うまくいったにすぎない。正しい座標間の距離を得るには「同一の座標系に変換した上で比較する」必要がある。

3 fk_Model による行列を用いた座標変換

手っ取り早いのはどちらもグローバル座標系に変換してしまうことだ。ローカル座標からグローバル座標を得るには、ローカル座標にモデルの行列を乗算すればよい。あるモデルが保持している行列は fk_Model::getMatrix() 関数で取得することができる。これを利用し、前に示した数式をソースコードで表現するなら以下のようになる。

```
fk_Matrix mat = model.getMatrix();
fk_Vector posL(0.0, 20.0, 0.0);
fk_Vector posG = mat*posL;
```

ある形状の高さ 20 の位置 $(0, 20, 0)$ が、fk_Model による移動や回転の後でどの位置になるかを知りたい場合は、以上の処理で求めることができる。「行列*ベクトル」の順番を守らなくてはならないことに注意しよう。

4 fk_Model による行列を用いた方向ベクトルの回転変換

ある方向ベクトルが、fk_Model による回転でどちらを向くのか知りたい場合は、いくつか方法がある。スマートさ、クールさ、複雑さでそれぞれ一長一短があるが、列挙してみよう。

1. 座標と同じように「行列*方向ベクトル」を計算し、モデルの座標値 (fk_Model::getPosition() で得たベクトル) を引く方法。ちょっとダサイけど、まぁシンプルな方法。
2. fk_HVector クラスを使い、同次座標成分を 0 にして行列と乗算する方法。fk_HVector クラスは x, y, z 成分に加えて w 成分を持ったクラスで、使い方はほぼ同じである。w 成分は通常 1 になっているが、**0 をセットすることで行列との乗算時に平行移動成分を無視する**ことができる。何故そうなるかは行列とベクトルの乗算式を確認してみよう。
3. モデルのオイラー角 (fk_Model::getAngle()) を取得し、それを用いて回転行列を生成して方向ベクトルと乗算する方法。手順は複雑だが、これは今後色々と応用が利く手法なので解説する。

座標変換に使う行列には平行移動量や角度をセットする必要があるが、fk_Matrix クラスには任意の回転や平行移動を表す行列を手軽に作成できるメンバ関数が用意されている。これを利用して、モデルと同じ姿勢を取るための回転行列を作成し、それと方向ベクトルを乗算すれば良い。コードは以下の通り。

```
fk_Matrix mat;
mat.makeRot(model.getAngle());
fk_Vector vecL(1.0, 0.0, 0.0);
fk_Vector vecG = mat*vecL;
```

あるモデルのローカル座標系における x 軸方向の向きを知りたい場合は、以上の処理で求めることができる。fk_Matrix::makeRot() 関数にオイラー角を渡してやると、その姿勢の分だけベクトルを回転させる行列が生成される。これ以外にも、平行移動行列を生成する makeTrans() や、任意軸の回転行列を生成する makeRot()、拡大縮小行列を生成する makeScale() が存在する。これらを用いて生成した行列を乗算することで合成変換も表現できるため、覚えておくと計算処理が非常にスマートになるだろう。

5　グローバル座標からローカル座標への変換

グローバル座標系で計算を行った結果、座標値なり移動ベクトルなりを元のローカル座標系に戻したい場合もあるだろう。グローバル座標を任意のローカル座標系上に持って行くには、グローバル座標にモデルの逆行列を乗算する。これは最初に述べた数式 (1) を逆方向に解くことで導くことが出来る。

$$\mathbf{M^{-1}P_G} = \mathbf{M^{-1}MP_L} \tag{2}$$

行列と逆行列の積は単位行列になるため、以下のようにまとめられる。

$$\mathbf{M^{-1}P_G} = \mathbf{P_L} \tag{3}$$

ソースコードでは以下の通り。fk_Model で扱う行列は必ず逆行列を持つので、本来は条件分岐せずとも良いのだが、念のためである。また fk_Matrix::inverse() を使わずに posL = !mat*posG; という手もある。逆行列の利用がその場限りの場合には便利だが、逆行列の算出にはそこそこ時間がかかるため、計算発生が最小限に抑えられるようにするべきだ。

```
if(mat.inverse()) {
    posL = mat*posG;
}
```

また、あるモデルの逆行列は自前で求めずとも、メンバ関数を通じて取得することもできる。fk_Model::getInvMatrix() がそれである。これも必要な際には利用すると良い。方向ベクトルを扱う場合の注意点は、順行列による変換と同様である。座標値を足し引きする方法だとまどろっこしいので、同次座標を 0 にした変換か、回転行列を作ってその逆行列を求める方法が、手順的には簡潔になるだろう。

6　親子関係と行列

親子関係とは、親モデルの移動や回転に対して子モデルが追従する様子を表すのに用いられる概念である。今更そのなんたるかは語るまでもないだろうが、ここまでで述べた行列の知識を持って復習すると、根本的な理論レベルでの理解が出来るはずである。

ユーザーズマニュアルにおける親子関係のセクションを読み返してみよう。「子モデルの持っていたグローバル座標系での位置と姿勢は、親モデルのローカル座標系でのそれとして扱われるようになる」とある。これは言い換えれば**親モデルのローカル座標系が子モデルにとってのグローバル座標系となる**ということである。

親子関係を用いない場合は「グローバル座標=モデル行列*ローカル座標」という式が成り立つが、このモデルが親モデルを持つ場合は、この式で求められるのは「親モデルのローカル座標」となる。つまり、グローバル座標を得るためには更に親モデルの行列による変換が必要であることを意味する。このような多段階の座標変換は、行列の乗算によって合成することが可能である。従って以下の式が成り立つ。

$$\mathbf{P_{Global}} = \mathbf{M_{Parent}M_{Child}P_{ChildL}} \tag{4}$$

このように、親モデルの行列を前方に乗算していくことで親子関係にあるモデルのローカル座標を求める変換行列が作成できる。親子の階層が深い場合も同様に乗算すれば良いが、より上位の親行列が先に来る点に注意しよう。この計算式を fk_Matrix を用いた計算処理コードにそのまま当てはめれば、子モデル座標系における座標値から絶対的なグローバル座標を得ることができる。

子モデルの中心がグローバル座標系のどこに位置するのかは、頻繁に取得したくなる情報である。その都度行列を用いた計算をせずとも良いように、fk_Model には一発でグローバル座標を取得できる関数が用意されている。それが getInh〜で始まる関数群である。Inh 版の関数は内部で上記のような行列演算を行ってくれるものである。

モデルの情報を取得する関数として getPosition(), getVec(), getUpvec(), getAngle(), getMatrix() などがあるが、それぞれに対して getInhPosition() のように Inh 版が存在する。通常の関数が「そのモデルにとってのグローバル座標系」の値を返すのに対して、Inh 版は「絶対的なグローバル座標系」の値を返してくれる。親モデルを持たないモデルの場合は、通常版と Inh 版で得られる値は同一となる。

単純に子モデルの状態を知りたい場合は、Inh 版の関数を用いるだけで事足りるが、任意の親モデル座標系での値を取得したい場合は、行列を用いて変換行列を作る必要があるだろう。

7　親子関係を結ぶ時、切り離す時

親子関係を設定するには setParent() 関数を用いるが、セットした途端にモデルの配置がおかしくなった、と感じたことはないだろうか？バグのように思えるかもしれないが、これは通常の動作である。マニュアルにもあったように「子モデルの持っていたグローバル座標系での位置と姿勢は、親モデルのローカル座標系でのそれとして扱われるようになる」ため、**親子関係の構築前にグローバル座標系で適切な位置関係に調整していても、その設定値をそのまま親モデルのローカル座標として反映すると、意図しない位置や姿勢になる**事が多い。

とはいえ、人体やロボットのようなモデルを構築したい場合は前述したように「今の位置関係を保持したまま親子関係を結びたい」と思うケースは非常に多い。その場合は、setParent() 関数の呼び出し方を変更し、**setParent(親モデルのポインタ, true)** というように、bool 値の引数を追加する。これは C++特有の関数定義方法で、省略されている場合は特定の値が指定されているものとし、引数が指定されている場合はその値を受け取る、という関数として作られているためである。これを「デフォルト引数」と呼ぶ。

setParent() の際に true を指定することにより、現状を維持したまま親子関係を結ぶことができる。FKUT においては true がデフォルトになるようにしていたため、ユーザはこの問題を意識したことは無かったはずである。これによって塊魂のように「モノにくっつく」という表現も可能になる。同様に、親子関係を解除する際にも同じフラグが選択でき、親子関係を結んで出来た位置関係を維持したまま、関係を切り離すことも可能である。

ではこの原理はどうなっているのだろうか？答えはこの資料内から得られる知識で説明できるはずである。考えてみよう。

8　確認問題

それでは以下の問題を解いて、今日得た知識や概念を確認しておこう。

1. setParent() で「位置関係を維持したまま親子関係を結ぶ」際に、どのような計算が行われているのかを説明しなさい。数式、プログラム、文章、いずれでも良い。

2. 高さ 20 程度の Cone を表示し、その先端に赤い球をぷすっと刺すように配置してみよう。親子関係を結ぶやり方と、グローバル座標を直接計算して配置するやり方の両方を試してみること。

3. Performer サンプルの女の子を読み込み、手に赤いボールを持たせてみよう。もちろん、モーション再生して手が動いてもちゃんと追従するようにすること。

4. ボールを持った女の子を 2 体表示して、片方を動かせる (単純な平行移動で良い) ようにし、お互いの持っているボールが触れあった時にボールの色が変わるようにしてみよう。