順運動学

http://www.tmps.org/index.php?%BD%E7%B1%BF%C6%B0%B3%D8

一般的なキャラクタアニメーションプログラミングの基礎を扱ってみたいと思います．まずはキャラクタの動作記述の基礎となる順運動学（Forward Kinematics．単に運動学と呼ぶ場合も）について，数学的基礎から DirectX Graphics での実装方法をまとめてみます．MSDN では「トランスフォーム」「行列スタック」辺りが対応しています．

3D トランスフォーム

3 次元モデルの位置や方向は，モデルを構成する各頂点を別の座標に移動させることで操作されます．例えば，モデルの形状を保ったまま平行移動する操作は，全ての頂点を同じ方向に同じ量だけ平行移動させることで実現されます．モデルの移動操作には平行移動以外にも回転移動等もありますが，こうした座標の移動操作は総称して座標変換（coordinate transformation）と呼ばれます．なお，ここでは MSDN Library の記述に合わせて，座標変換のことをトランスフォームと記述することにします．3Dグラフィクスでは，以下に示す基本的なトランスフォームを組み合わせて使用することでモデルを操作します．

1. 回転（rotation）
   - 原点を中心にモデルを回転させます．
2. 平行移動（translation）
   - 原点を中心にモデルを平行移動させます．
3. 拡大縮小（scaling）
   - 原点を中心にモデルを拡大もしくは縮小させます．
4. 鏡像変換（mirroring）
   - ある軸に関してモデルを対称移動させます．
5. せん断（shear）
   - ある軸に沿ってモデルの各頂点を平行移動させます．この時，各頂点の移動量は軸からの距離に比例して増加します．

せん断について少し補足します．例えば，Fig.1a のような正方形を x 軸に沿ってせん断すると，Fig.1b のような平行四辺形が得られます．モデリング等ではよく使用される操作なのかもしれませんが，キャラクタアニメーションではほとんど登場しません．よって詳しい説明は冗長でしょうからここでは省略します．

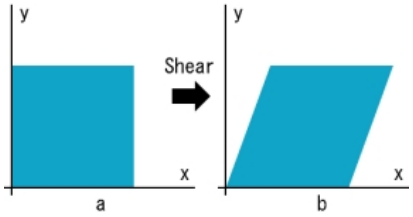

Fig.1 せん断

回転，平行移動，拡大縮小，鏡像変換の 4 種類のトランスフォームは総称して相似変換（similarity transformation）と呼ばれ，相似変換とせん断は総称してアフィン変換（affine transformation）と呼ばれます．ここで，アフィン変換は原点もしくは原点を通る任意の軸を基準としたトランスフォームである点に注意する必要があります．例えば，原点に位置する球を回転させると惑星が自転するような運動が得られますが，原点から離れた位置の球を回転させると，原点を中心として惑星が公転するような運動が得られることになります．

3D トランスフォーム行列

3 次元空間のアフィン変換は 4 次の正方行列であるアフィン変換行列（affine transformation matrix）で記述されます．3D ベクトル v をアフィン変換行列 M によって v' にトランスフォームするとき，DirectX Graphics におけるトランスフォーム計算（D3DXVec3Transform関数）は次のように表されます．

$$
\begin{aligned}
\mathrm{v}' &= \mathrm{vM} \\
(x', y', z') &= (x, y, z)\begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \\
x' &= xM_{11} + yM_{21} + zM_{31} + M_{41} \\
y' &= xM_{12} + yM_{22} + zM_{32} + M_{42} \\
z' &= xM_{13} + yM_{23} + zM_{33} + M_{43}
\end{aligned}
$$

ここで，3D ベクトルとアフィン変換行列では次数が一致しないので，ベクトル×行列の計算には w 成分（$w=1$）を加えた 4D ベクトル $v=\begin{pmatrix}x & y & z & 1\end{pmatrix}$ を用います．

アフィン変換行列の作成

鏡像変換とせん断を除いた基本的なアフィン変換行列（回転，平行移動，拡大縮小）の作成方法を示します．また，Fig.2 のような原点から伸びるベクトル $v=\begin{pmatrix}1 & 1 & 1\end{pmatrix}$ をトランスフォームした結果をそれぞれ図示します．

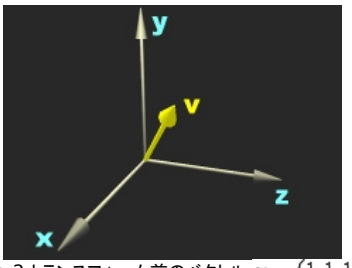

Fig.2 トランスフォーム前のベクトル $v=\begin{pmatrix}1 & 1 & 1\end{pmatrix}$

回転

各軸周りに θ だけ回転させるトランスフォーム行列は，それぞれ次のように表されます．例図では $\theta=90^{\circ}$，黄色の小矢印が回転中のベクトルの軌跡，緑の矢印が回転後のベクトルをを示します．なお，DirectX Graphics では各軸の先端方向から見て時計周り方向を正の回転方向として扱います．

$$R_x(\theta)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos(\theta) & \sin(\theta) & 0\\ 0 & -\sin(\theta) & \cos(\theta) & 0\\ 0 & 0 & 0 & 1\end{bmatrix},\ R_y(\theta)=\begin{bmatrix}\cos(\theta) & 0 & -\sin(\theta) & 0\\ 0 & 1 & 0 & 0\\ \sin(\theta) & 0 & \cos(\theta) & 0\\ 0 & 0 & 0 & 1\end{bmatrix},\ R_z(\theta)=\begin{bmatrix}\cos(\theta) & \sin(\theta) & 0 & 0\\ -\sin(\theta) & \cos(\theta) & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

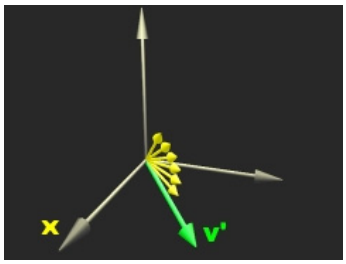

Fig.3a 回転トランスフォーム(x-axis)

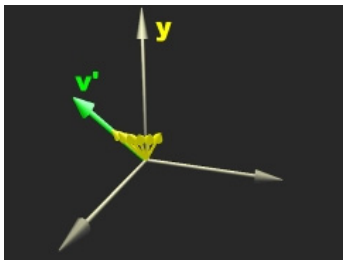

Fig.3b 回転トランスフォーム(y-axis)

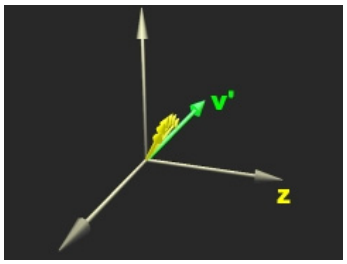

Fig.3c 回転トランスフォーム(z-axis)

平行移動

x 方向に t_x，y 方向に t_y，z 方向に t_z だけ平行移動させるトランスフォーム行列は，次のように表されます．例図では $t_x=t_y=0$，$t_z=1.0$ とし，黄色の小矢印が移動の軌跡を，緑の矢印が移動後のベクトルを示します．

$$\mathrm{T}(t_x,t_y,t_z)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ t_x & t_y & t_z & 1\end{bmatrix}$$

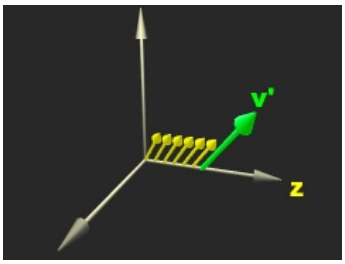

Fig.4 平行移動トランスフォーム

拡大縮小

x 方向に s_x，y方向に s_y，z方向に s_z だけ拡大もしくは縮小させるトランスフォーム行列は，次のように表されます．例図では $s_x=s_y=1.0$，$s_z=2.0$ とし，黄色の矢印がもとのベクトル，緑の矢印が拡大後のベクトルを示します．

$$\mathrm{S}(s_x,s_y,s_z)=\begin{bmatrix}s_x & 0 & 0 & 0\\ 0 & s_y & 0 & 0\\ 0 & 0 & s_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

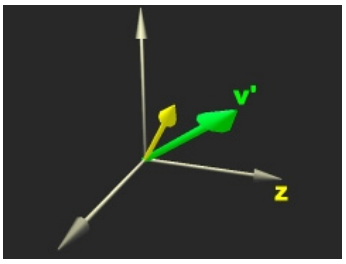

Fig.5 拡大縮小トランスフォーム

トランスフォーム合成

高度な 3D アニメーションを実現するためには複雑なトランスフォームを扱う必要がありますが，それらは全て，回転，平行移動，拡大縮小の基本トランスフォームを組み合わせることで実現されます．このように複数のトランスフォームを組み合わせることでより複雑なトランスフォームを生成する操作をトランスフォーム合成といい，トランスフォーム行列の乗算によって実装されます．

ここで，行列 A と B の乗算結果は掛け合わせる順序によって異なること，つまり「AB≠BA」である点が重要になります． つまり，トランスフォームの組み合わせの順序によって合成結果が異なることを意味します． 例えば，Fig.3a の回転と Fig.4 の平行移動を合成し，Fig.2 のベクトルをトランスフォームする場合， 回転→平行移動の順序で合成すると Fig.6a，平行移動→回転の順序で合成すると Fig.6b の結果が得られます． 図中の黄色の小矢印が 1 番目のトランスフォームによる軌跡，赤の小矢印が 2 番目のトランスフォームによる軌跡を示し，そして緑の矢印が合成トランスフォームによるトランスフォーム結果を示します．

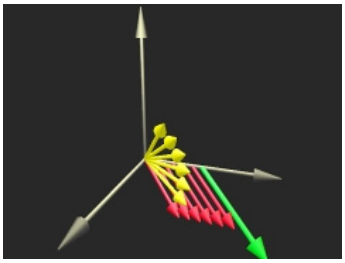

Fig.6a 回転→平行移動

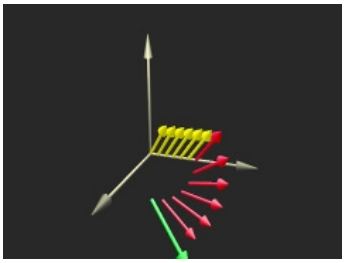

Fig.6b 平行移動→回転

また，Fig.6a と 6b の計算式はそれぞれ次のように表されます．

$$\mathrm{v}' = \mathrm{v}\mathbf{R}_x\mathbf{T}$$

$$\mathrm{v}' = \mathrm{v}\mathbf{T}\mathbf{R}_x$$

このように，[トランスフォーム対象ベクトル]×[1番目のトランスフォーム]×[2番目のトランスフォーム]…の順番で 乗算することによって求められます．

順運動学

多関節リンク構造

ロボット工学の分野では，複数のリンクを関節で連結した構造を多関節リンク構造と呼びます．また，リンク構造の起点はルート（root），終端はエフェクタ（end-effector）と呼ばれます．例えば，ロボットアームにおけるルートは地面に接する台座，エフェクタはアームの先端を指します．3D アニメーションで扱うキャラクタ， 例えば人体も前腕，上腕，大腿，下腿，etc…のリンクが種々の関節で連結された多関節リンク構造の一種とみなすことができます（Fig.7）．キャラクタアニメーションにおいては，一般的に頭頂や手先，足先をエフェクタとして扱いますが，指も曲がるような構造の場合は各指の先端がエフェクタとなりますし，あご先をエフェクタとすることもできます．一方，ルートは腰に設定するのが一般的です．もちろん，ルートは絶対に腰に設定すべきだというわけではありませんが，特別な理由がない限りは慣習に従っておいたほうがよいでしょう．

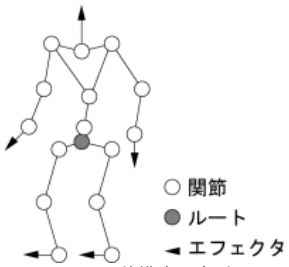

Fig.7 人体構造モデル例

階層構造

Fig.7 の人体モデルの右腕の姿勢を操作する場合を考えます．例えば，右手首を 90°曲げると，右手首を中心に右手が回転移動します．このとき，身体のほかの部位の姿勢は全く変化しません． 次に，右ひじを 90°曲げると，右ひじを中心に右前腕が回転移動すると同時に，手首の曲げ角を保った状態で右手も回転移動します．このように，あるリンクの姿勢が複数の関節の階層的な作用によって変化する構造を階層構造と呼びます．言い換えれば，ある関節の回転が下位階層にあたる全てのリンクの姿勢を変化させるような構造が階層構造であるともいえます．Fig.7 の人体モデルを例に挙げると，モデルの関節は Fig.8 のような階層構造を持っていることになります．

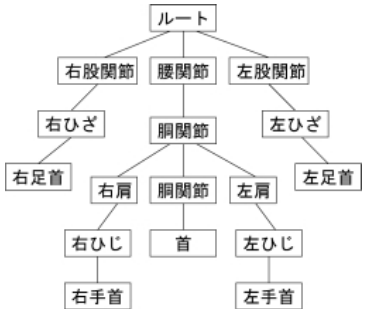

Fig.8 Fig.7の人体モデルにおける関節の階層構造

階層構造はリンク（もしくは関節）の親子関係を示すものです．例えば Fig.8 に示すように右肩関節は右手→右前腕→右上腕の3つのリンクの親関節となっており， これらの 3 つのリンクの 3 次元空間中における姿勢に影響する関節であることを示しています．また，右肩は胴関節の子関節にあたるため，胸部リンクからルートに至る親リンクの姿勢には全く影響しないことを示します．ルートは全てのリンクの親になっており，3 次元空間中における位置と身体全体の回転を扱います．

キャラクタの階層構造においては，1 つの親に対して複数の子が存在しますが，1 つの子に対しては 1 つの親しか存在しないような，一対多の親子関係になる場合がほとんどです．したがってそうした階層構造は単純な木構造を用いて実装することができます．木構造では構成要素をノード（node）と呼ぶことから，キャラクタのリンクや関節のことをノードと呼ぶ場合もあるようです． 余談ですが，多対多の親子関係を持つような構造をロボット工学では並列機構（parallel mechanism）と呼ぶようです．並列機構は Fig.9b のようにリンクが閉ループを構成するような構造を指します．構造中に閉ループが存在すると，逆運動学（inverse kinematics）や逆動力学（inverse dynamics）の計算が途端に複雑になります．キャラクタアニメーションで並列機構を扱わなければならないケースは少ないでしょうから，閉ループを含まないような設計に留意したほうがよいでしょう．

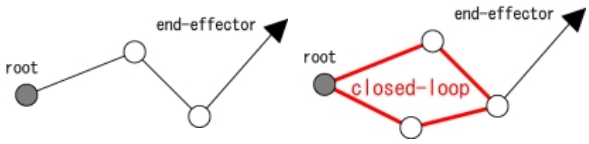

Fig.9a 開ループ構造　　Fig.9b 閉ループ構造

順運動学

キャラクタを構成する全てのリンクが固定長の剛体リンクのとき，そのキャラクタの姿勢は各関節の回転角度によってのみ決まります．この時，キャラクタの姿勢は全関節の回転角度をパラメータとする関数によって定式化することが可能です．このように，リンク構造の姿勢を決定するパラメータから，各リンクの位置や方向を計算する問題を順運動学(forward kinematics)と呼びます．関節 i の位置を $p_i=(p_x\ p_y\ p_z)$，キャラクタを構成する全ての関節の角度からなるベクトルを $\theta=(\theta_0\ \theta_1\cdots\theta_N)$ とすると，順運動学は次のように定式化されます．

$$\mathrm{p}_i = f\ (\theta)$$

関数 f は関節角度ベクトル θ から関節位置 p への写像を計算する写像関数です．写像関数は全ての関節の角度を引数としていますが，キャラクタは階層構造を構成していますので，実際には関節 i の親関節の角度のみを用いて計算することになります．

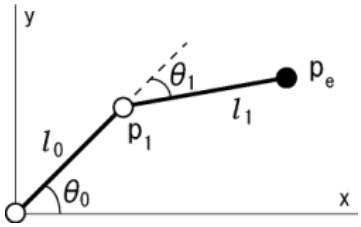

Fig.10 2 次元-2 リンク構造

Fig.10 のような2次元上の 2 リンク構造を例に順運動学の計算例を示します．なお，ここでは DirectX Graphics にならって時計回り方向を正の回転方向とします．Fig.10 は，リンク0(リンク長: l_0)とリンク1(リンク長: l_1)を x 軸上で連結し，リンク0の根元を θ_0，リンク0とリンク1の連結点を θ_1 回転させた状態を示します．このとき，連結点の位置 $p_1=(p_{1x}\ p_{1y})$ と，エフェクタの位置 $p_e=(p_{ex}\ p_{ey})$ は次のように計算できます．

$$\begin{aligned}
p_{1,x} &= l_0\cos(-\theta_1)\\
p_{1,y} &= -l_0\sin(-\theta_1)\\
p_{e,x} &= l_1\cos(\theta_1-\theta_0)+l_0\cos(-\theta_0)\\
p_{e,y} &= -l_1\sin(\theta_1-\theta_0)-l_0\sin(-\theta_0)
\end{aligned}$$

ここで，リンク0の初期状態を $l_0=(l_0\ 0)$，リンク1の初期状態を $l_1=(l_1\ 0)$ とベクトル表記すると，上式は次のように行列表記できます．

$$\begin{aligned}
p_1 &= l_0R(-\theta_0)\\
&= \emptyset T(l_0)R(-\theta_0)\\
p_e &= l_1R(\theta_1-\theta_0)+l_0R(-\theta_0)\\
&= l_1R(\theta_1)R(-\theta_0)+l_0R(-\theta_0)\\
&= \emptyset\{T(l_1)R(\theta_1)+T(l_0)\}R(-\theta_0)\\
&= \emptyset T(l_1)R(\theta_1)T(l_0)R(-\theta_0)\\
R(\theta) &= \begin{bmatrix}\cos(\theta) & -\sin(\theta) & 0\\ \sin(\theta) & \cos(\theta) & 0\\ 0 & 0 & 1\end{bmatrix}, T(v(v_x,v_y))=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ v_x & v_y & 1\end{bmatrix}, \emptyset=(0,0,1)
\end{aligned}$$

このように，任意の関節の位置はその関節からルートに至る全ての親リンクと親関節をたどりながら，リンクの初期状態を表す平行移動行列 T と関節の回転行列 R によって ゼロベクトル $\emptyset$ を順次トランスフォームすることで求められます．つまり，関節 i の位置 p_i を求める写像関数は次のように表されます．

$$p_i = f(\theta) = \emptyset T_iR(\theta_{i-1})T_{i-1}\cdots R(\theta_0)T_0R(\theta_{root})T_{root}$$

以上は全リンクが固定長剛体リンクであると仮定した定式化ですが，直動関節(可変長リンク)を伴う構造を扱う場合にも，上式の T を $T(l\cdot v_0)$ (l はリンク長，v_0 はリンクの初期方向を示す単位ベクトル)のように 拡張することで容易に対応できます．

DirectX Graphicsによる実装

キャラクタのように多くの分岐構造を持つモデルにおいて順運動学計算を行う場合，毎回ルートから計算対象リンクに至るトランスフォーム合成計算を行っていると，計算のオーバーヘッドが大きくなります．階層構造では，ある親関節からルートに至る合成トランスフォームは複数の子関節で共有することができます．例えば，Fig.8 のモデルにおいて右肩と左肩の位置を同時に計算する場合，ルートから胴関節までの合成トランスフォームを計算し，これに胴関節→右肩関節，胴関節→左肩関節へのトランスフォームをそれぞれ乗算することで，無駄な計算を省くことが可能です．

このような計算の枠組みは行列スタックとして，DirectX Graphics はもちろん，多くの 3D グラフィクス API において提供されています．行列スタックは多関節リンク構造など階層構造を持つモデルの描画では必ず利用する機能です．サンプルプログラムでは，行列スタックを用いて Fig.11 のような単純なリンク構造を操作し，各関節の座標を計算します．

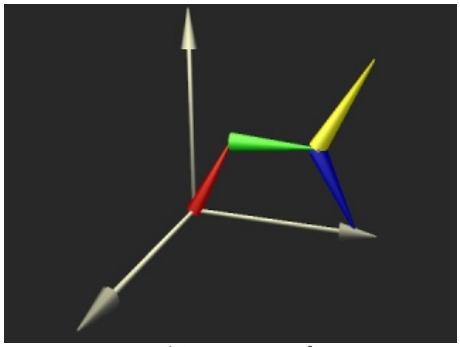

Fig.11 実行画面のスナップショット

サンプルプロジェクトは以下になります(プロジェクト名 : Hierarchy). プログラムを実行すると,「0」キー～「3」キーで各リンクを回転できます. SHIFT キーを同時に押すと逆回転になります. また,「FK」メニューを選択すると, 各リンク先端の座標がメッセージボックスに表示されます. なお, かなり汚い実装ですが, Fig.2～Fig.6 を制作するためのプログラムも含んでいます.

VC++.NET2003プロジェクトファイル(33Kb)

行列スタックは, 行列の乗算結果を格納するためのスタック構造です. DirectX Graphics では ID3DXMatrixStack インターフェイスを通じて利用できます. Fig.12 は ID3DXMatrixStack インターフェイスの提供する命令と, 命令に対応したスタック内容の変化を図示しています. まず, LoadIdentity 関数が呼ばれると, スタック先頭に単位行列がセットされます. 続いて, 平行移動行列を乗算する Translate 関数が呼ばれると,「スタック先頭行列 × $T(v_0)$」の計算結果でスタック先頭内容が上書きされます. 同様に, 回転行列を乗算する RotateYawPitchRoll 関数(Yaw:Y 軸周りの回転, Pitch:X 軸周りの回転, Roll:Z 軸周りの回転)が呼ばれると, スタック先頭要素が「スタック先頭行列 × $R_x(\theta_x)$」で上書きされます.

次に Push 関数が呼ばれると, 呼び出し時のスタック先頭要素がコピー&プッシュされます. つまり, スタック先頭のトランスフォーム行列をバックアップする命令だとみなせます. なお, 直前にバックアップされた行列は Pop 関数によって取り出せますが, そのときの先頭要素は破棄されます.

また Fig.12 に示すように, スタック先頭行列に回転行列を乗算する関数には, RotateYawPitchRoll 関数と RotateYawPitchRollLocal 関数の 2 種類があります. RotateYawPitchRoll 関数は「スタック先頭行列×回転行列」, RotateYawPitchRollLocal 関数は「回転行列×スタック先頭行列」というように, これらは行列の乗算順序が異なります. 平行移動行列の乗算(Translate 関数, TranslateLocal 関数)や拡大縮小関数(Scale 関数, ScaleLocal 関数)においても,「～Local」を伴う場合は左からの乗算, 伴わない場合は右側からの乗算になります.

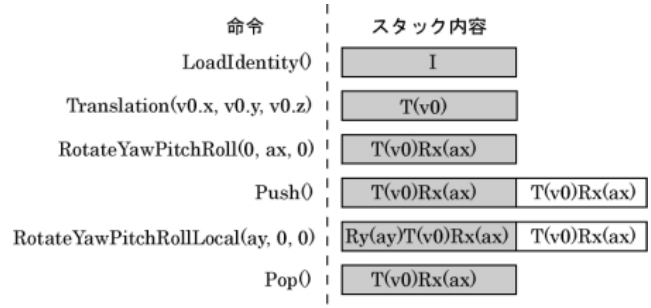

Fig.12 DirectX Graphicsの行列スタック

ここで, Fig.11 の青リンクと黄リンクの姿勢を計算する場合を考えます. 行列スタックを用いないと, ルート→青リンク, ルート→黄リンクへの合成トランスフォームをそれぞれ独立して計算する必要があります. しかし, これら 2 つのトランスフォーム合成の過程において, ルートから緑リンク先端に至る計算が共通することがわかります. そこで行列スタックを導入し, まずルート→緑リンク先端への合成トランスフォームをスタック先頭要素にセットし, スタックをプッシュして先頭要素をバックアップします. 次に, スタック先頭要素に緑リンクから青リンクへのトランスフォームを合成し, 青リンクの姿勢を計算します. 続いて黄リンクの姿勢を計算しますが, ルート→緑リンク先端への合成トランスフォームはスタックをポップして得られますので, あとは緑リンク→黄リンクへのトランスフォームを合成するだけで計算できます. このように, 階層構造をもつモデルの順運動学計算では, 行列スタックを用いることで効率的に計算量を削減できます.

以下に, 行列スタックを用いた多関節リンク構造の描画関数の実装例(一部抜粋)を示します. 詳細な説明は省略しますので, Fig.12 などと併せて読み進めてみてください.

```
  "CSceneView.cpp"

  HRESULT CSceneView::DrawLinks(void)
  {
    LPD3DXMESH pMesh;
    LPD3DXMATRIXSTACK pMatStack;

    // 初期状態 円錐ポリゴンメッシュの生成
    D3DXCreateCylinder(m_pDeviceD3D, 0.1f, 0.01f, 1.0f, 10, 10, &pMesh, NULL);

    // 行列スタックの生成
    D3DXCreateMatrixStack(0, &pMatStack);

    // 手順① 行列スタックにリンク構造全体の回転トランスフォームをセット
    pMatStack->RotateYawPitchRoll(0.0f, D3DXToRadian(m_pfRotation[0]), 0.0f);

    // 手順② 原点→第1リンク（赤）の中心位置へのトランスフォーム
    pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
    // 第1リンクの描画
    DrawMeshSub(pMesh, *pMatStack->GetTop());

    // 手順③ 第1リンク中心→第1リンク先端への平行移動トランスフォーム
    pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
    // 手順④ 第2リンク以降の回転トランスフォーム
    pMatStack->RotateYawPitchRollLocal(0.0f, D3DXToRadian(m_pfRotation[1]), 0.0f);

    // 手順⑤ 第1リンク先端→第2リンク（緑）の中心への平行移動トランスフォーム
    pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
    // 第2リンクの描画
    DrawMeshSub(pMesh, *pMatStack->GetTop());

```

```
32 |   // 手順⑥ 第2リンク中心→第2リンク先端への平行移動トランスフォーム
33 !   pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
34 |
35 |   pMatStack->Push();
36 |     // 手順⑦ 第3リンク（青）以降の回転トランスフォーム
37 !     pMatStack->RotateYawPitchRollLocal(0.0f, D3DXToRadian(m_pfRotation[2]), 0.0f);
38 |     // 手順⑧ 第2リンク先端→第3リンク中心への平行移動トランスフォーム
39 !     pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
40 |     // 第3リンクの描画
41 !     DrawMeshSub(pMesh, *pMatStack->GetTop());
42 |   pMatStack->Pop();
43 |
44 |   pMatStack->Push();
45 |     // 手順⑨ 第4リンク（黄）以降の回転トランスフォーム
46 !     pMatStack->RotateYawPitchRollLocal(0.0f, D3DXToRadian(m_pfRotation[3]), 0.0f);
47 |     // 手順⑩ 第2リンク先端→第4リンク中心への平行移動トランスフォーム
48 !     pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
49 |     // 第4リンクの描画
50 !     DrawMeshSub(pMesh, *pMatStack->GetTop());
51 |   pMatStack->Pop();
52 |
53 |   pMatStack->Release();
54 |   pMesh->Release();
55 |
56 |   return S_OK;
57 ! }
```

次は，行列スタックを用いた順運動学計算により，各リンク先端（関節）の座標を計算する関数の実装例です． ルート→各リンク先端までの合成トランスフォームを計算し，それによってゼロベクトルをトランスフォームした結果を返します． 描画関数に非常に似ていますが，リンクの描画処理が不要になっている分シンプルになっています．

```
 1   "CSceneView.cpp"
 2
 3   D3DXVECTOR3 CSceneView::GetLinkTipPosition(int nLink)
 4 | {
 5 |   LPD3DXMATRIXSTACK pMatStack;
 6 |   D3DXVECTOR3 vZero(0.0f, 0.0f, 0.0f);
 7 |
 8 |   // 行列スタックの生成
 9 !   D3DXCreateMatrixStack(0, &pMatStack);
10 |
11 |   // 行列スタックにリンク構造全体の回転トランスフォームをセット
12 !   pMatStack->RotateYawPitchRoll(0.0f, D3DXToRadian(m_pfRotation[0]), 0.0f);
13 |
14 |   // 原点→第1リンク（赤）先端への平行移動トランスフォーム
15 !   pMatStack->TranslateLocal(0.0f, 0.0f, 1.0f);
16 |   if (nLink == 0)
17 |     D3DXVec3TransformCoord(&vZero, &vZero, pMatStack->GetTop());
18 |
19 |   // 第2リンク（緑）以降の回転トランスフォーム
20 !   pMatStack->RotateYawPitchRollLocal(0.0f, D3DXToRadian(m_pfRotation[1]), 0.0f);
21 |   // 第1リンク先端→第2リンク先端への平行移動トランスフォーム
22 !   pMatStack->TranslateLocal(0.0f, 0.0f, 1.0f);
23 |   if (nLink == 1)
24 |     D3DXVec3TransformCoord(&vZero, &vZero, pMatStack->GetTop());
25 |
26 |   pMatStack->Push();
27 |     // 第3リンク（青）以降の回転トランスフォーム
28 !     pMatStack->RotateYawPitchRollLocal(0.0f, D3DXToRadian(m_pfRotation[2]), 0.0f);
29 |     // 第2リンク先端→第3リンク先端への平行移動トランスフォーム
30 !     pMatStack->TranslateLocal(0.0f, 0.0f, 1.0f);
31 |     if (nLink == 2)
32 |       D3DXVec3TransformCoord(&vZero, &vZero, pMatStack->GetTop());
33 |   pMatStack->Pop();
34 |
35 |   pMatStack->Push();
36 |     // 第4リンク（黄）以降の回転トランスフォーム
37 !     pMatStack->RotateYawPitchRollLocal(0.0f, D3DXToRadian(m_pfRotation[3]), 0.0f);
38 |     // 第2リンク先端→第4リンク先端への平行移動トランスフォーム
39 !     pMatStack->TranslateLocal(0.0f, 0.0f, 1.0f);
40 |     if (nLink == 3)
41 |       D3DXVec3TransformCoord(&vZero, &vZero, pMatStack->GetTop());
42 |   pMatStack->Pop();
43 |
44 |   pMatStack->Release();
45 |   return vZero;
46 ! }
```

まとめ

本稿では 3 次元座標変換の基礎と順運動学の基礎について概説し，DirectX Graphics の行列スタックを用いた実装例を示しました． MSDN Library や CG 標準テキストブック(CG-ARTS協会刊)，ロボット工学の文献等を参考にしつつ， 詳細をかなり省略しましたが，それでもなかなか骨の折れる内容でした．特に用語等には気を配りましたが，不正確な記述等ありましたらぜひお知らせください．こちらでもたまに見直しつつ加筆，修正していきたいと思います．
