かんたん Java3D

付録: javax.vecmath パッケージ詳説

- Java3D の数学

初版 1999,9/18
第 2 版 1999,10/25
第 3 版 1999,11/3
第 4 版 1999,11/30
第 5 版 1999,12/24
第 6 版 1999,12/27
第 7 版 2002,1/13
(株) 永和システムマネジメント

平鍋 健児
Kenji HIRANABE

はじめに

javax.vecmath は，Java3D API の一部として公開されているベクトル・マトリクス演算のためのパッケージです．Java3D の華々しさにくらべるとあまり注目されないパッケージですが，Java3D の数学の基礎を担うばかりでなく，Java3D 以外の API からも利用することができる代数・幾何演算がコンパクトにサポートされています．ここでは，このパッケージの API およびそれが扱うコンピュータグラフィックスの数学についてやや詳しく解説していきます．

vecmath パッケージの位置付けと用途

Java 3D は，表 1 のように大きく分けて以下の 3 つのパッケージで提供されています．

表 1　Java3D パッケージの構成

パッケージ名	内容
javax.media.j3d	Java3D 本体
javax.vecmath	ベクトル演算
com.sun.j3d.*	Sun が提供するツール群

このうち，javax.media.j3d と javax.vecmath は正式な API ですが， com.sun.j3d は Sun が提供するツール群です．javax.media.j3d と javax.vecmath は jar 形式のクラスファイル群として提供されていて，現在参照ソースコードは提供されていません．com.sunj3d については，参考コードという位置付けであり，ソースコードも見ることができます．

javax.vecmath は Java3D API の一部として公開されていますが，そのパッケージ名から Java3Dからは独立した存在であることが分かると思います．一般的な2,3,4,n次元のベクトルと点，およびその変換マトリクスを扱うことができ，Java3D以外の他の用途からも汎用的に利用可能です．もちろん主な用途はコンピュータグラフィックスであり，vecmath パッケージの最初の利用例は Java3Dです．また，vecmath はコンパクトでかつ JDK のコアパッケージ以外に依存していないため，Java2D や Swing, AWT などと共に使っても問題ありません．３次元的な表示を Java3D を用いずに AWT などとで簡便に得たい場合にも利用可能です．全く別の用途としては，vecmath が扱う代数の性質を利用して連立一次方程式を解くこともできます．特に n次元ベクトルである GVector と n×m次元一般マトリクスである GMatrix を組み合わせて多次元の連立一次方程式を解く，といった変わった用途にも使えます．

クラス構成

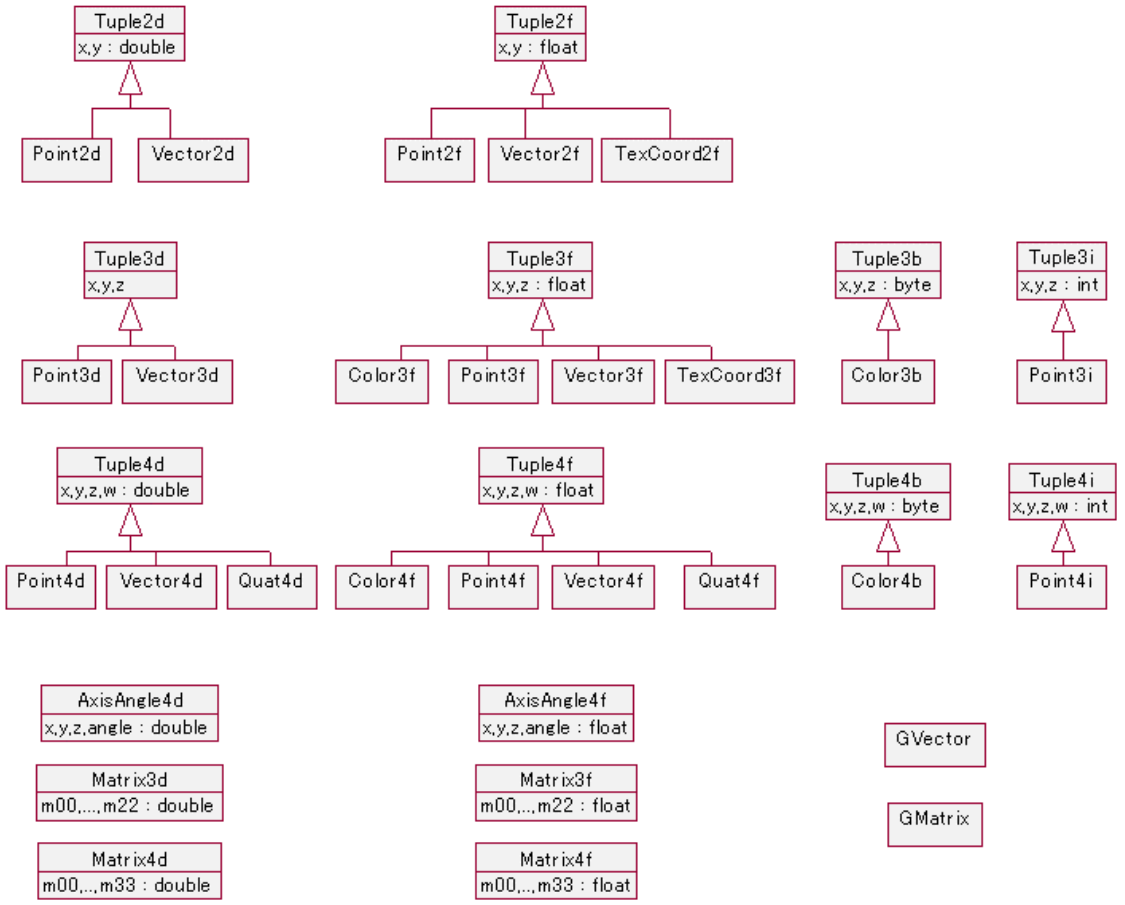

図 1　vecmath の全クラス

図 1 に vecmath パッケージの全クラスを示します．クラスは全部で 40 あまりと数は多いですが，非常にシンプルなクラスの集合で構成されています．一部に継承構造がありますが複雑な協調関係はありません．ほぼフラットなクラス群であり，名前が示す概念を知っていれば理解するのは容易でしょう．各クラスには豊富なメソッド群が用意されていますが，図では属性(インスタンス変数)のみが記述されています．クラス名の命名則を Vector3f の例で説明します．

Vector 3 f

概念　　次元 データ型

最初に概念の名前(Vector)を示し，次にその次元数(3)，その後に 内部表現に使用する基本データ型(f)が続きます．概念名およびデータ型を表す文字を，表2，3に示したので参照してください．ほとんどはなじみのある概念のはずです．

表 2　概念名

概念名	意味
Tuple	データ要素の列
Vector	ベクトル
Point	点
Matrix	マトリクス(変換行列)
Quat	クオータニオン(4 元数)
Color	色
TexCoord	テクスチャ座標
AxisAngle	軸周りの回転

表 3　データ型名

型名	基本データ型
f	float
d	double
b	byte
i	int

例えば Point3d なら，3次元の点を3つの doubleで表現したものであり，Matrix4f なら，4次元の変換マトリクスを 4x4=16 個の float で表現したものです．ほぼすべてのクラスがこの要領で命名されていますが，例外は GVector と GMatrix で，これらには double 型をデータ要素として使ったものしか提供されていません．’G’は Generic の意で，一般の次元(n 次元)を扱っていることを示しています．

設計思想

まず，vecmath パッケージのソフトウェア設計から見た側面についてその特徴を見てみましょう．vecmath パッケージは「速度最優先」という設計思想を持っているようです． これは，次のようなデザインに現れています．

1. インスタンス変数がすべて public

   通常のオブジェクト指向設計では，インスタンス変数，すなわちクラスのデータメンバを public と宣言するのは避けます． これはカプセル化の基本であり，内部表現の変更が利用者のコードに 伝搬するのを避ける意味があります．ところが，vecmath では あえて get/set メソッドを用意せず，直接データを public と宣言してメソッド呼び出しによる速度低下を減らしています．例えば Vector3f の3つのインスタンス変数である x, y, z は共に public であり，直接 float 値の代入と参照が可能になっています． ただし，このデザインはクラスの内部表現が非常に安定している (設計変更が予想されない)ことを前提としており，vecmath のように数学的にしっかりとした基礎を持ったクラス群でのみ妥当であると思われます．

```
Vector3f u = new Vector3f();

// x 座標の設定と取得
u.setX(1);              // 間違い(そのようなメソッドはない)
u.x = 1;                // 正しい
```

```
float x = u.getX();     // 間違い(そのようなメソッドはない)
x = u.x;                // 正しい
```

2. 内部に配列表現を持たない

例えば Matrix4d クラスなどでは，4x4 = 16 個の double 型のデータを保持する必要があります．普通に実装すれば double [4][4] の配列オブジェクトを内部に保持する形になるでしょう．ところが Matrix4d では，m00, m01, … ,m33 の独立した 16 個の double 型の変数を保持しています．このようにして，配列オブジェクトの new によるオーバーヘッドと，[]添え字演算による速度低下を押さえています．このような設計のため，逆に，添え字による要素メンバへのアクセスは速度の低下を招きます．特に Matrix4d の getElement(int i, int j) および setElement(int i, int j, double value)メソッド(m[i][j] にあたる)は非常に遅いことを覚えておくとよいでしょう．i,j をループで回すようなコーディングは簡単に速度を低下させてしまいます．このようなコーディングを使いたくなる場面では，実現したいことを行うメソッドが必ず用意されているはずなので，まず API を調べてみることをお勧めします．

```
Matrix4d m = new Matrix4d();

// m[1][2] の設定と取得
double e = m.getElement(1, 2);    // 正しいが遅い
e = m.m12;                        // 正しい
m.setElement(1, 2, 100);          // 正しいが遅い
m.m12 = 100;                      // 正しい
```

3. ほとんどのメソッドが final

Object クラスから継承した toString() や hashCode() などのメソッド以外のメソッドは，すべて final 宣言されています．すなわち，サブクラスでメソッドをオーバーライドすることはできません．これも一見オブジェクト指向設計に反するように思われます．しかし vecmath では１つ１つのメソッドに明確な仕事が定義されており，数学的にその意味が変更されることは考えられません．サブクラス化によって振舞の変更が妥当であるメソッドは見当たりません．このメソッドの final 化によって，コンパイラはメソッド呼び出しのコードを高速化することができます．

4. synchronized メソッドはない

すべてのメソッドは，synchronized 宣言されていません．これも，synchronized メソッドの呼び出しが速度面でのネックとなることが多いことを考慮した設計です．デメリットは，基本的に同期化を利用者側で考慮しなければならなくなることです．必要であれば，vecmath の利用者側で同期化をする手法(external synchronization)を 取ることができます．

5. 基本データ型以外を return しない

どのメソッドも float, double, boolean, int といった基本データ型以外を戻り値として使用していません．これは，インスタンス生成(new)が速度のネックになることが多いという事実の回避策です．その代わり，通常戻り値としてオブジェクトを期待するメソッドでは自分自身(this)の値を書き換える仕様が多用されています．例えば，ベクトルの和を求める add() メソッドでは戻り値として結果のベクトルを生成するのではなく，自分自身を上書きしてしまいます．

```
Vector3f u = new Vector3f(1, 0, 0);
Vector3f v = new Vector3f(0, 1, 0);
Vector3f w =  u.add(v);              // 間違い(コンパイルエラー)
Vector3f w =  new Vector3f();
w.add(u, v);                         // 正しい
```

この仕様のおかげで，インスタンスの生成タイミングを利用者側に任せることができ，より効率的なコードが利用者の工夫で書けます．例えば，「間違い」とコメントされたようなAPIの場合，Vector3fのadd()メソッドは中でVector3fをnewすることになり，ヘビーなループの中でこのメソッド呼び出しが行われるとおびただしい数のVector3fオブジェクトが生成されます．利用者側でnewが呼ばれるタイミングをコントロールできる ことは，速度面では非常に重要となります．

このように，vecmathのAPIは何よりもまず速度に重点を置いた設計となっています．これらのデザインは，純粋なオブジェクト指向に慣れた人には多少違和感を与えるかもしれません．しかし，3Dグラフィックスに携わったことのある人にはよく理解できると思います．「マシンから最後の１滴までパフォーマンスを絞り出す」，という3Dグラフィックスが 歩んで来た歴史と美学が感じられる API です．

点とベクトル

vecmathでは，点とベクトルを別のクラスとして区別しています．グラフィックスライブラリであるOpenGLなどのAPIでは，点もベクトルもfloat[3]などのように基本データ型の配列として扱われています．数学的に別の概念をしっかり区別してクラス化していることで，全APIに渡ってメソッドの引数の意味が明確になり，型チェックも有効に機能します．

余談になりますが，Java3D APIレビュー過程の初期では，点とベクトルはともにVec3fクラスでした．Sunの仕様策定者側は，APIが膨張するのを嫌って点とベクトルを単一のクラスとしました．その後Java3d-interest@java.sun.comメーリングリストでの活発な議論の結果，APIの膨張よりも仕様とドキュメントの明確さの方が優先されるべきだという結論に至りました．そしてJava 3D 1.0 APIでは，Tuple3fという３つの実数の組を表現する共通のスーパークラスを設け，それを継承する形でPoint3fとVector3fを別クラスとして提供することになったのです． その後，API 1.1 を経て現在の API 1.3 に至るまで，このクラス階層は変更されていません．

さて，ここですこし数学的な準備をしておきましょう．点もベクトルももともとは幾何的な実在であり，数とは直接関係していません．数と図形を関係付けるためには座標系を導入する必要があります．座標系を導入することで幾何を代数の言葉で表現できるようになります．すなわち，数式を使って図形を扱うことができるのです．これは数学者デカルトの功績であり，直交座標系はデカルト座標系とも呼ばれます．図２に，Java3Dおよびvecmathでもちいられる座標系を示します．

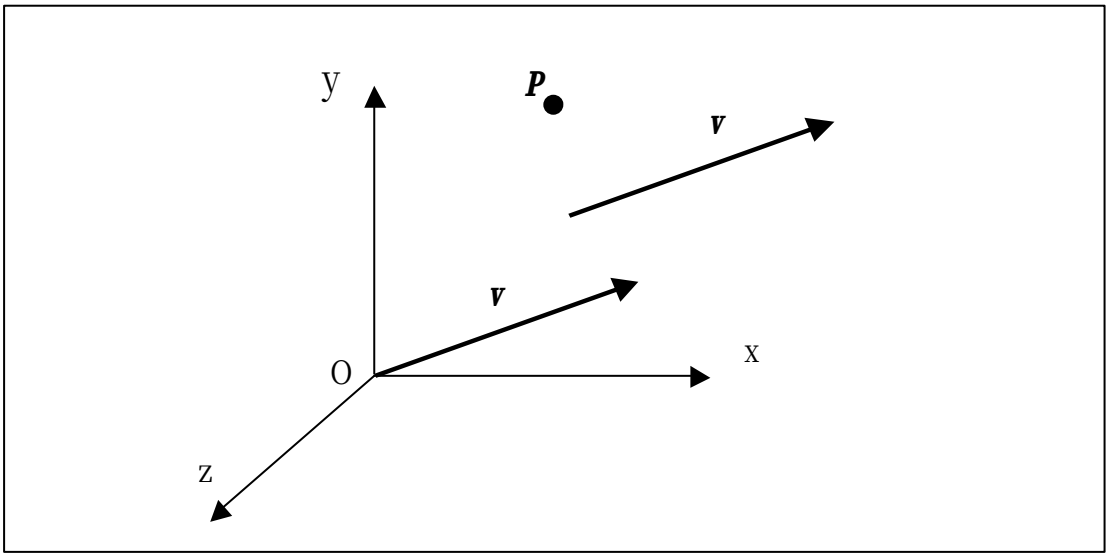

図 2　　座標系

O が座標系の原点です．座標系の配置は右手系，すなわち x 座標の＋方向から y 座標の＋方向へと右ねじを回したときにねじの進む方向が z座標の＋方向となるように規定されています．あるいは，右手の親指と人差し指と中指を互いに直交するように伸ばしたとき，x,y,z軸がそれぞれ親指，人差し指，中指になるような配置とも言えます．Java3D では一貫して右手系が使用されています．

図で $\boldsymbol{v}$ はベクトル，$\boldsymbol{P}$ は点を示しています．座標系を導入することで，点もベクトルも３つの実数の組で表現されます．ここではその数学的記述として，列ベクトルを用います．点 $\boldsymbol{P}$，およびベクトル $\boldsymbol{v}$ は以下のように表されます．

$$\boldsymbol{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \qquad \boldsymbol{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

点もベクトルも同じような表現になります．しかし，両者には違いがあり，区別して扱う必要があります．ベクトルは向きと大きさを持つ量であり，平行移動しても同じとみなされます．図２で $\boldsymbol{v}$ と記された2つのベクトルはともに同じ値です．これに対して，点は平行移動すると別の値になってしまいます．他にも違いがあります．２つの点の間には距離が定義できますが，２つのベクトルの間には距離が定義されません．その代わりベクトルには長さが定義されますが，点には長さが定義されません．その他の定義される演算にも違いがあります．ベクトル同士には足し算，引き算，およびスカラー(実数)倍の演算が規定されていますが，点同士は足し算ができませんし，スカラー倍も定義できません(これらの演算は座標系の取り方によって結果が変わってしまいます)．だだし，点同士の引き算はベクトルを生じ，点にベクトルを加えると新たな点が生成されます．

点は原点とその点を結ぶベクトル（位置ベクトル）として考えることができますが，座標系を換えてしまうとその位置ベクトルは変化してしまうため，点の位置ベクトルは束縛ベクトルとも呼ばれます．

vecmath では，Tuple クラスによって点とベクトルに共通する演算が定義され，点独自の演算は Point に，ベクトル独自の演算は Vector に定義されています．例えば２点間の距離は Point クラスの distance() に定義され，ベクトルの長さは Vector クラスの length() に定義されています．ただし，ベクトルの足し算を行う add() やスカラー倍する scale() などは，Tuple に定義されているため，前に定義できないとした点同士の足し算やスカラー倍を行うコードも，実際にはコンパイルできてしまうので注意が必要です．

点とベクトルの代表的な演算を，表４に挙げます．表中，u,v,wはベクトル，p,qは点，k,s,t,e,θ はスカラーであり，←は，代入を示します．

表 4　　代表的なベクトル・点の演算

操作		数学的表記	vecmath の表記
2 ベクトルの和		w ← u + v	w.add(u, v)
		u ← u + v	u.add(v)
2 ベクトルの差		w ← u　v	w.sub(u, v)
		u ← u　v	u.sub(v)
ベクトルのスカラー倍		u ← k v	u.scale(k, v)
		v ← k v	v.scale(k)
ベクトルの長さ		k ← $\|v\|$	k = v.length()
2 ベクトルの内積		k ← u･v = $\|u\|\|v\|\cos\theta$	k = u.dot(v)
2 ベクトルの外積		w ← u×v	w.cross(u, v)
		u ← u×v	u.cross(v)
ベクトルの単位化		u ← v /$\|v\|$	u.normalize(v)
		v ← v /$\|v\|$	v.normalize()
2 ベクトルの角度		θ ← $\cos^{-1}(u\cdot v / \|u\|\|v\|)$	θ = v.angle(u)
2点の距離	L_2	k ← $\|p-q\|$	k = p.distance(q)
	L_2 の２乗	k ← $\|p-q\|^2$	k = p.distanceSquared(q)
	L_1	k ← $\Sigma\|p_i-q_i\|$	k = p.distanceL1(q)

L_∞	$k \leftarrow \max(\lvert p_i - q_i \rvert)$	k = p.distanceLinf(q)
ベクトルの反転	u ← v	u.negate(v)
	v ← v	v.negate()
ベクトル・点の内分	w ← (1-t)u + tv	w.interpolate(u, v, t)
	u ← (1-t)u + tv	u.interpolate(v, t)
スカラー倍と和の混合	w ← su +v	w.scaleAdd(s, u, v)
	u ← su +v	u.scaleAdd(s, v)
ほぼ等しい	$\max(\lvert u_i - v_i \rvert) <= e$	u.epsilonEquals(v, e)
	$\max(\lvert p_i - q_i \rvert) <= e$	p.epsilonEquals(q, e)
凡例	u, v, w … ベクトル p, q … 点 k, s, t, e, θ … スカラー ← … 代入	

内積と外積

表４の演算の中で，内積と外積という２つのベクトル同士の演算が出てきます．高校数学で学習する内容ですが，ここでその図形的な意味と代数的な意味を見てみましょう．

表 5　内積と外積

	定義	vecmath の表記
内積	$\boldsymbol{a}\cdot\boldsymbol{b}=\lvert\boldsymbol{a}\rvert\lvert\boldsymbol{b}\rvert\cos\boldsymbol{q}=a_xb_x+a_yb_y+a_zb_z$	d = a.dot(b)
外積	$\boldsymbol{a}\times\boldsymbol{b}=\lvert\boldsymbol{a}\rvert\lvert\boldsymbol{b}\rvert\sin\boldsymbol{q}\,\boldsymbol{n}=(a_yb_z-a_zb_y,\ a_zb_x-a_xb_z,\ a_xb_y-a_yb_x)$	c.cross(a, b)

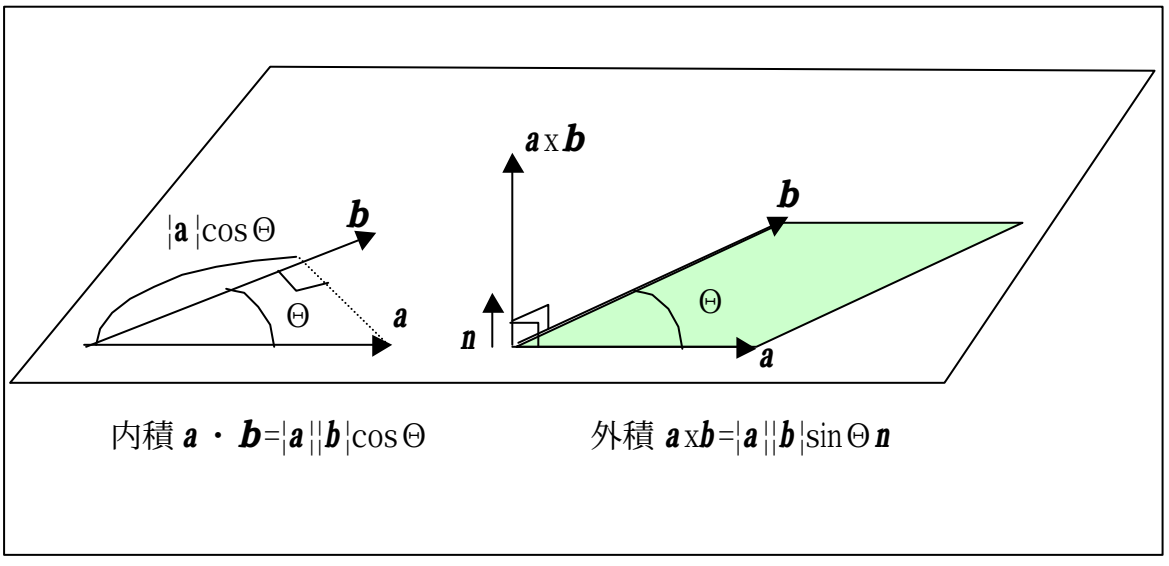

図 3　内積と外積

内積

内積は，別名 inner product, dot product, scalar product とも呼ばれ．図形的には２つのベクトルの間に図３のように定義されるスカラー(実数値)です．$\boldsymbol{a}$ を $\boldsymbol{b}$ に射影した時の長さ($|\boldsymbol{a}|\cos\Theta$)と，$\boldsymbol{b}$ の長さの積となります．定義から，最大値は $|\boldsymbol{a}||\boldsymbol{b}|$，最小値は $-|\boldsymbol{a}||\boldsymbol{b}|$ となり，２つのベクトルが平行に近いほど，大きな値が得られ，直交するときは0になり，逆平行の時に最小値を取ります．代数的には両ベクトルの各対応する要素を掛けて合計したものとなります．内積を直感的に捉えるには，２つのベクトルがどれだけ同じ方向を向いているかの指標と考えるとよいでしょう．

特に，$\boldsymbol{b}$ が単位ベクトル $\boldsymbol{u}$ の時，$\boldsymbol{a}\cdot\boldsymbol{u}$ は「$\boldsymbol{a}$ の $\boldsymbol{u}$ 方向の成分」を表すことに注目してください．例えば，$\boldsymbol{a}$ を $\boldsymbol{u}$ 方向と，$\boldsymbol{u}$ に直交する方向の２つのベクトルに分解したいときは，内積を使って，

$$\boldsymbol{a}=\boldsymbol{a}_{//}+\boldsymbol{a}_{\perp},\quad \boldsymbol{a}_{//}=(\boldsymbol{a}\cdot\boldsymbol{u})\boldsymbol{u},\quad \boldsymbol{a}_{\perp}=\boldsymbol{a}-\boldsymbol{a}_{//}$$

と表現できます．また，x 軸方向，y軸方向，z軸方向のそれぞれの単位ベクトルを $\boldsymbol{i},\boldsymbol{j},\boldsymbol{k}$ とすると，$\boldsymbol{a}$ を内積を使って 3 方向に分解し，

$$\boldsymbol{a}=(\boldsymbol{a}\cdot\boldsymbol{i})\boldsymbol{i}+(\boldsymbol{a}\cdot\boldsymbol{j})\boldsymbol{j}+(\boldsymbol{a}\cdot\boldsymbol{k})\boldsymbol{k}=a_x\boldsymbol{i}+a_y\boldsymbol{j}+a_z\boldsymbol{k}=a_x\begin{pmatrix}1\\0\\0\end{pmatrix}+a_y\begin{pmatrix}0\\1\\0\end{pmatrix}+a_z\begin{pmatrix}0\\1\\1\end{pmatrix}=\begin{pmatrix}a_x\\a_y\\a_z\end{pmatrix}$$

と記述することで，$\boldsymbol{a}$ のベクトル成分を図形的な意味で認識することが可能です．また，同じベクトル同士を内積すると，Θ=０からそのベクトルの長さの２乗となります．すなわち，ベクトルの長さは内積を使って，

$$|\boldsymbol{a}|=\sqrt{\boldsymbol{a}\cdot\boldsymbol{a}}=\sqrt{a_x^{\,2}+a_y^{\,2}+a_z^{\,2}}$$

と書けます．

外積

外積は別名，outer product, cross product, vector product とも呼ばれ，図形的には２つのベクトルの間に図３のように定義されるベクトルです．内積と違って外積はベクトルを生じることに注意してください．外積ベクトルの長さは，２つのベクトルが作る平行四辺形の面積です．代数的には，両ベクトルの各要素をたすき掛けしたものとなります（表５参照）．外積ベクトルの長さの最大値は $|\boldsymbol{a}||\boldsymbol{b}|$，最小値は $-|\boldsymbol{a}||\boldsymbol{b}|$ となり，２つのベクトルが直交に近いほど大きな値が得られ，平行の時に０になります．また，外積ベクトルの向きは $\boldsymbol{a}$ と $\boldsymbol{b}$ が作る平面の法線方向で，$\boldsymbol{a}$ から $\boldsymbol{b}$ へ右ねじを回した時にねじの進む方向です．図３では，$\boldsymbol{n}$ で平面の単位法線を表現しています．外積を直感的に捉えるには，２つのベクトルがどれだけ直交しいているか，の指標と考えるとよいでしょう．また，同じベクトル同士の外積は０ベクトルを生じます．

変換とマトリクス

コンピュータグラフィックスでは，点およびベクトルの変換(移動)は最も基本的な操作です．ある座標系において，点やベクトルを移動したり視点を変更したりといった操作が頻繁に行われます．ここでは，変換とそれを表現するマトリクスについて説明します．

ベクトルと変換

まず，ベクトルの変換から考えてみます．３次元空間のベクトルは，３つの実数の組で表されます．ある変換 $\boldsymbol{T}$ を考え，ベクトル $\boldsymbol{v}$ が変換 $\boldsymbol{T}$ によって $\boldsymbol{v}'$ に変換されたとします．これを，

$$\boldsymbol{v} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \quad \xrightarrow{T} \quad \boldsymbol{v}' = \begin{pmatrix} v'_x \\ v'_y \\ v'_z \end{pmatrix}$$

と書き，簡単に，

$$\boldsymbol{v} \xrightarrow{T} \boldsymbol{v}'$$

とか，

$$\boldsymbol{v}' = T(\boldsymbol{v})$$

と書きます．この変換が，仮に９つのパラメータ m00～ m22 を使って，

$$\begin{aligned} v'_x &= m00\ v_x + m01\ v_y + m02\ v_z \\ v'_y &= m10\ v_x + m11\ v_y + m12\ v_z \\ v'_z &= m20\ v_x + m21\ v_y + m22\ v_z \end{aligned}$$

と記述できるとき，この変換を「線形変換」といいます．ここで 3x3 マトリクス $\boldsymbol{M}$ を，

$$\boldsymbol{M} = \begin{pmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{pmatrix}$$

と定義すると，この変換 $\boldsymbol{T}$ は，

$$\begin{pmatrix} v_x' \\ v_y' \\ v_z' \end{pmatrix} = \begin{pmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

と書け，さらに簡潔に，

$$\boldsymbol{v}' = \boldsymbol{M}\boldsymbol{v}$$

と書けます．$\boldsymbol{M}$ を $\boldsymbol{T}$ を表現する変換マトリクスと呼びます．すべての変換が線形変換ではありません．すなわち，すべての変換がマトリクス $\boldsymbol{M}$ で表現できるわけではありません．さまざまな変換の中でも，マトリクスの掛け算で表現できる変換を，線形変換と呼ぶのです．線形変換の性質については後述します．

マトリクスの行と列

マトリクスは日本語で「行列」と呼ばれますが，横の並びを行，縦の並びを列と呼びます．vecmath では行列の添え次は行，列，の順です．すなわち，M の i 行 j 列の要素が m_{ij} です．

線形変換において，マトリクスの各要素の位置は，第一列目(m00,m10,m20)が，x 軸方向の単位ベクトル(1,0,0)の変換先，第２列目(m01,m11,m21)が y 軸方向の単位ベクトル(0, 1, 0)の変換先，第３列目(m02,m12,m22)が z 軸方向の単位ベクトル(0, 0, 1) の変換先であることを覚えておくと便利です．

$$\boldsymbol{M} = \left(\begin{array}{c|c|c} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{array}\right) = \left(M\begin{pmatrix}1\\0\\0\end{pmatrix} \quad M\begin{pmatrix}0\\1\\0\end{pmatrix} \quad M\begin{pmatrix}0\\0\\1\end{pmatrix} \right)$$

数式上，変換マトリクスはベクトルの左から掛けるという点に注意をしてください．コンピュータグラフィックスの数学に関してはさまざまな文献がありますが，ベクトルを行ベクトルとして扱うか列ベクトルとして扱うか(マトリクスを右から掛けるか左からかけるか)，座標系を右手系にとるか左手系にとるか，図形の変換を扱うか座標系の変換を扱うか，さらに，行列の添え字(ij)のどちらが行を表しどちらが列を表すか，によって変換マトリクスは行と列が反転します．文献にあたる際は，この点に細心の注意が必要です．Java3D API では，一貫して

- ベクトル，点を表現する数ベクトルは列ベクトルで，マトリクスは左から掛ける
- 座標系は右手系
- マトリクスの添え字は，行が最初

というルールにしたがっています．最後のルールは，Matrix3f などのクラスではその public 変数である m_{ij} の i が行を表し，j が列を表すことを示しています．

回転変換

線形変換の例として，z 軸回りの θ ラジアン回転変換は，

$$\begin{pmatrix} v_x' \\ v_y' \\ v_z' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

と表現されます．この場合の変換マトリクスは，

$$\boldsymbol{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

です．前述のマトリクス各列の意味付けから，ベクトル(1, 0, 0) が (cos θ , sin θ ,0)に,ベクトル(0,1,0)が(-sin θ , cos θ , 0) に変換され，ベクトル(0, 0, 1) はそのまま動かないことが分かります．3 次元空間上の任意の軸に関する回転はやや複雑なので後述します．

例えば，ベクトル $\boldsymbol{v}$(1,0,0)を z 軸回りに 90 度回転したい場合，vecmath では次のようなコーディングになります．

```
Vector3d v = new Vector3d(1,0,0);   // ベクトルの初期値
Matrix3d Rz = new Matrix3d();       // マトリクスを作成
Rz.rotZ(Math.PI/2);                 // 回転マトリクスをセット
Rz.transform(v);                    // Rz を v に作用させる
System.out.println(v);              // (0,1,0) が出力される
```

ベクトルの回転変換の例

vecmath のコーディング上も，かならずマトリクスを表す変数はベクトルを表す変数の左に来ます．すなわち，マトリクス $\boldsymbol{M}$ とベクトル $\boldsymbol{v}$ の掛け算は数式上は，$\boldsymbol{Mv}$ であり，コーディング上は，M.transform(v) です．数式での変数の並び順が，常にコーディングテキスト上での並び順と一致するように API が配慮されています．

スケール変換

さて，回転と同様に，スケール変換（s 倍）に関する変換行列は，

$$\boldsymbol{S}(s)=\begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{pmatrix}=s\boldsymbol{I} \qquad \boldsymbol{I} \text{ は単位行列}$$

です．例えば，ベクトル $\boldsymbol{v}$(1,0,0)に対してスケール変換(2 倍)を行うコード例は，次のようになります．

```
Vector3d v = new Vector3d(1,0,0);   // ベクトルの初期値
Matrix3d  S = new Matrix3d();       // マトリクスを作成
S.set(2);                           // スケールマトリクスをセット
S.transform(v);                     // S を v に作用させる
System.out.println(v);              // (0,2,0) が出力される
```

ベクトルのスケール変換の例

線形変換の性質

これまで回転変換とスケール変換を例にして，線形変換を説明しました．ここでベクトルの線形変換の性質を改めて見てみましょう．ベクトルの線形変換には，名前からも明らかなように「線形性」があります．「線形性」とは，2つのベクトルの和を変換すると，個々のベクトルを変換した後に和をとったものと等しくなること，および，スカラー倍したベクトルを変換すると，もとのベクトルを変換した後にスカラー倍したものと等しくなることです．式で表すと，次のようになります．

$$T(\boldsymbol{v}_1+\boldsymbol{v}_2)=T(\boldsymbol{v}_1)+T(\boldsymbol{v}_2)$$
$$T(k\boldsymbol{v})=kT(\boldsymbol{v})$$

最初の式が，和に対する線形成，次の式がスカラー倍に対する線形成を表しています．$\boldsymbol{T}$ が線形変換の場合，$\boldsymbol{T}(\boldsymbol{v})=\boldsymbol{M}\boldsymbol{v}$ と書けることから上記の性質は自然に成り立つことがわかります．

$$T(\boldsymbol{v}_1+\boldsymbol{v}_2)=\boldsymbol{M}(\boldsymbol{v}_1+\boldsymbol{v}_2)=\boldsymbol{M}\boldsymbol{v}_1+\boldsymbol{M}\boldsymbol{v}_2=T(\boldsymbol{v}_1)+T(\boldsymbol{v}_2)$$
$$T(k\boldsymbol{v})=\boldsymbol{M}(k\boldsymbol{v})=k\boldsymbol{M}\boldsymbol{v}=kT(\boldsymbol{v})$$

最初，線形変換を「マトリクスで表現できること」と定義し，その性質として「線形性」を導きましたが，実は「線形性」そのものを線形変換の定義とし，そこから「マトリクスで表現できる」という性質を導くこともできます．

線形性に関する性質の中でも特にスカラー倍に関する性質から，「0 ベクトルは 0 ベクトルに変換される」，という大切な性質が導かれます．

$$T(\boldsymbol{0})=\boldsymbol{0}$$

点の平行移動

これまで回転変換とスケール変換を見てきました．これらはマトリクスで表現される線形変換です．次に，平行移動について考えてみます．ベクトルに関しては，平行移動は意味をなしません．平行移動してもベクトルの値は変化しないのです．点の平行移動は，点にベクトルを足すことで得られますが，これを変換マトリクスで表現することはできません．線形変換は 0 ベクトルを 0 ベクトルに変換するという性質がありますが，点を位置ベクトルとして捉えた場合，この性質は「原点は動かない」，と読み替えることができます．平行移動は明らかにこの性質を満たしま

せんから，線形変換ではありません．すなわち，点の平行移動はそのままマトリクスの掛け算として扱うことはできないのです．最初に述べたように，そもそも点という概念に対しては加算やスカラー倍が定義できないため，点の集合が代数でいう「線形空間」を形成してないことに問題があります．

そこで，3次元の点を4次元ベクトルで表現する，という工夫をします．これはコンピュータグラフィックスの数学で古くから行われている技法です．この4次元ベクトルを同次ベクトルと呼びます．4次元ベクトルを考え，4つ目の要素で残りの3つの要素を割り算したものを，3次元の点と同一視します．同じく，3次元ベクトルも4次元ベクトルで表現することができます．ベクトルの場合は，4つ目の要素を単純に無視して4次元ベクトルと同一視します．

$$\boldsymbol{p}=\begin{pmatrix}x\\y\\z\\w\end{pmatrix} \Leftrightarrow \boldsymbol{p}=\begin{pmatrix}x/w\\y/w\\z/w\end{pmatrix} \qquad \boldsymbol{v}=\begin{pmatrix}x\\y\\z\\w\end{pmatrix} \Leftrightarrow \boldsymbol{v}=\begin{pmatrix}x\\y\\z\end{pmatrix}$$

点の場合，通常は，w は常に 1 と考えておけば，4次元ベクトルの x,y,z 座標がそのまま3次元の点の x,y,z 座標と一致します．ベクトルの場合はwは常に0と考えておくと都合がいいです．同次ベクトルを導入することで，点の平行移動もマトリクスと列ベクトルとの積で表現できるようになります．すなわち，点 $\boldsymbol{P}$ およびベクトル $\boldsymbol{v}$ を，ベクトル $\boldsymbol{t}$ = (t_x, t_y, t_z) だけ平行移動して，それぞれ $\boldsymbol{P}'$, $\boldsymbol{v}'$ とすると，

$$\begin{aligned} p'_x &= p_x + t_x & v'_x &= v_x \\ p'_y &= p_y + t_y & v'_y &= v_y \\ p'_z &= p_z + t_z & v'_z &= v_z \end{aligned}$$

となりますから，同次ベクトルを用いて，

$$\begin{pmatrix}p_x'\\p_y'\\p_z'\\1\end{pmatrix}=\begin{pmatrix}1&0&0&t_x\\0&1&0&t_y\\0&0&1&t_z\\0&0&0&1\end{pmatrix}\begin{pmatrix}p_x\\p_y\\p_z\\1\end{pmatrix} \qquad \begin{pmatrix}v_x'\\v_y'\\v_z'\\0\end{pmatrix}=\begin{pmatrix}1&0&0&t_x\\0&1&0&t_y\\0&0&1&t_z\\0&0&0&1\end{pmatrix}\begin{pmatrix}v_x\\v_y\\v_z\\0\end{pmatrix}$$

あるいは，

$$\boldsymbol{p}'=\boldsymbol{T}(\boldsymbol{t})\boldsymbol{p} \qquad \boldsymbol{v}'=\boldsymbol{T}(t)\boldsymbol{v}, \quad \boldsymbol{T}(\boldsymbol{t})=\begin{pmatrix}1&0&0&\\0&1&0&\boldsymbol{t}\\0&0&1&\\0&0&0&1\end{pmatrix}$$

と表現できます．ベクトル場合は，平行移動では値が変わらないこと，すなわち $\boldsymbol{v}=\boldsymbol{v}'$ となり，点とは振る舞いが違うことに注意してください．

このように，4次元同次ベクトルと4次元マトリクスを用いることで，平行移動も線形変換と同様にマトリクスの掛け算で扱うことができます．これまで見た平行移動以外の回転，スケールの変換マトリクスは，そのまま4次元に拡張できるので，回転，スケール，平行移動の3種類の変換は，すべて 4x4 のマトリクスで表現できたことになります．

ちなみに 3 次元の点の4次元ベクトルへの拡張は，平行移動をも含めて点の変換をマトリクスの掛け算で扱おうという目的が主ですが，他にも，w=0 とすることである方向の無限遠点を表現できたり，特定の場面で数値計算による誤差を押さえたり，さらに後述するように回転変換を表す Quaternion との自然な演算が定義できるという数学的な利点もあります．

さて，vecmathでは，変換マトリクスをMatrix4f,Matrix4dクラスとして定義し，豊富なメソッド群を提供しています．４次元マトリクスは，vecmathの中で中心的な存在ですので，このクラスが提供するメソッドを理解すれば，Java3Dが扱う変換のほとんどの意味を理解したことになります．

表6に，代表的な変換の意味とマトリクスの中身，vecmathのMatrix4dクラスでの表記をまとめました．Matrix4fもほとんど同じです．表中，mはMatrix4dのオブジェクト，θは角度を表すdoubleのスカラー，sはスケールを表すdoubleのスカラー，tは平行移動を表すVector3d，m3は回転を表すMatrix3dをそれぞれ示します．

表 6　代表的なMatrix4dによる変換と対応するメソッド

意味	マトリクス	Matrix4dのメソッド
z軸回りの回転	$\boldsymbol{R}_z(\boldsymbol{q}) = \begin{pmatrix} \cos\boldsymbol{q} & -\sin\boldsymbol{q} & 0 & 0 \\ \sin\boldsymbol{q} & \cos\boldsymbol{q} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	m.rotZ(θ)
スケール	$\boldsymbol{S}(s) = \begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	m.set(s)
平行移動	$\boldsymbol{T}(\boldsymbol{t}) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$	m.set(t)
上記３つの混合（回転，スケール平行移動の順）	$\boldsymbol{T}(\boldsymbol{t})\boldsymbol{S}(s)\boldsymbol{R}_z(\boldsymbol{q}) = \begin{pmatrix} s\cos\boldsymbol{q} & -s\sin\boldsymbol{q} & 0 & t_x \\ s\sin\boldsymbol{q} & s\cos\boldsymbol{q} & 0 & t_y \\ 0 & 0 & s & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$	m.set(m3 t, s)
凡例	m … Matrix4d θ… 角度を表すdoubleのスカラー s … スケールを表すdoubleのスカラー t … 平行移動を表すVector3d m3 … 回転を表すMatrix3d	

各変換は連続して行うことができます．たとえば，z軸回りに回転した後，スケールして平行移動する変換マトリクス $\boldsymbol{M}$ は，

$$\boldsymbol{M} = \boldsymbol{T}(\boldsymbol{t})\boldsymbol{S}(s)\boldsymbol{R}_z(\boldsymbol{q})$$

という具合です．この場合，順に変換マトリクスを左から積算した結果できるマトリクスが合成変換のマトリクスとなります．

また，ここでは述べませんでしたが，他に，鏡像，せん断，と呼ばれる変換があります．鏡像は鏡に写した像を得る変換であり，せん断は斜めにつぶす変換です．せん断は，正方形を平行四辺形に変換します．回転，スケール，平行移動，鏡像，せん断から合成される変換をアフィン変換といいます．アフィン変換は，幾何的には平行線を平行線に写す，という特徴で定義されます．アフィン変換を図4に示します．図中，射影変換はアフィン変換ではあり

ませんが後述するので並べて掲載しています．中でも回転，スケール，平行移動の３つのアフィン変換は，図形の形を変えないためコンピュータグラフィックスで頻繁に用いられます．

アフィン変換を表現するの 4x4 のマトリクスは，

- 左上の 3x3 成分で，回転とスケールを表現
- 第４列（最も右の列）で平行移動を表現
- 第４行（最も下の行）はいつでも (0 0 0 1)

という特徴があります．これを知っていると，Matrix4f,Matrix4d の各メソッド名の意味をつかむのに便利です．4x4 マトリクスについて詳しくは後の章にまわすことにして，先に回転変換と密接な関係をもつ Quaternion について説明します．

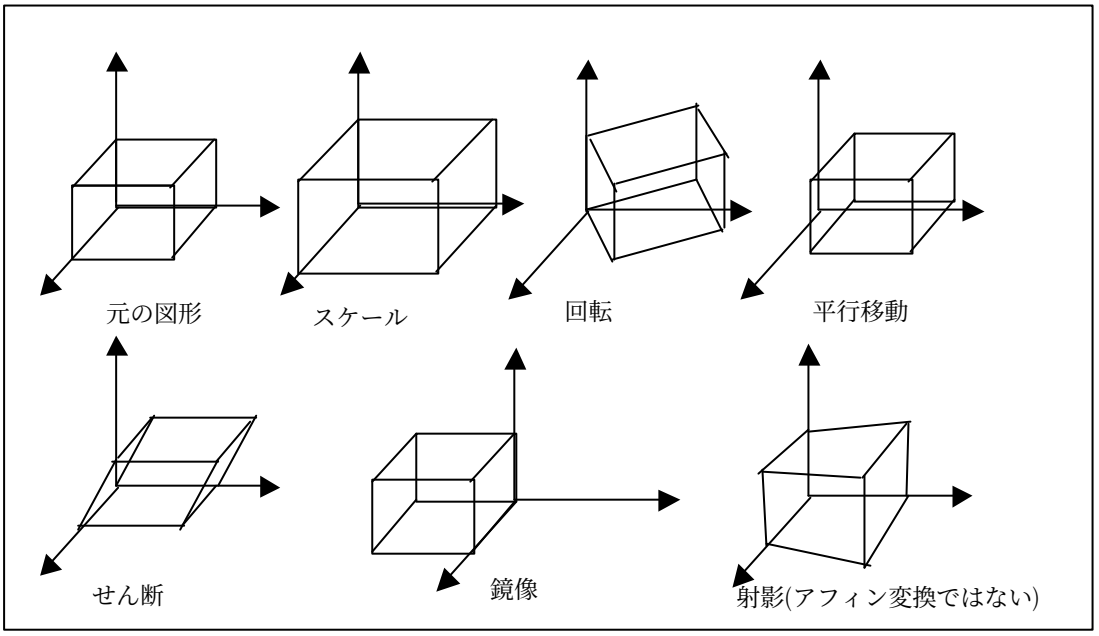

図 4 アフィン変換

回転と Quaternion

前節では，マトリクスによる回転の表現を見てきました．ｚ軸回りの回転のみを扱ったのは，任意の軸回りの回転は，マトリクス表現では非常に複雑になってしまうからです．また，マトリクスによる表現はすべてのアフィン変換を扱うことができるため，必要とされる実数要素(double や float)の数も多くなってしまいます．回転変換のみに焦点をあてたコンパクトな表現は他にないでしょうか？２次元ならば，回転角度だけで十分なのですが，３次元では任意の軸に関しての回転を表現する必要があるため，２次元ほど簡単ではないようです．

vecmath API では，３次元の回転を表現するために，４種類のクラスを用意しています．

表 7　回転を表現するクラス

クラス	構成要素
Matrix3f/Matrix3d	m00, m01, …, m22
Matrix4f/Matrix4d	m00, m01,…,m33
AxisAngle4f/AxisAngle4d	x, y, z, angle
Quat4f/Quat4d	x, y, z, w

Matrix3f/Matrix3d はすでに述べたように３次元回転変換マトリクスであり，Matrix4f/Matrix4d は同次ベクトルに対する３次元回転変換マトリクスです．AxisAngle4f/AxisAngle4d は，もっとも人間の直感に近い回転表現でしょう．AxisAngle の要素は，(x, y, z, angle) であり，(x, y, z) で回転軸を定義し，その軸回りの回転角を angle で定義します．

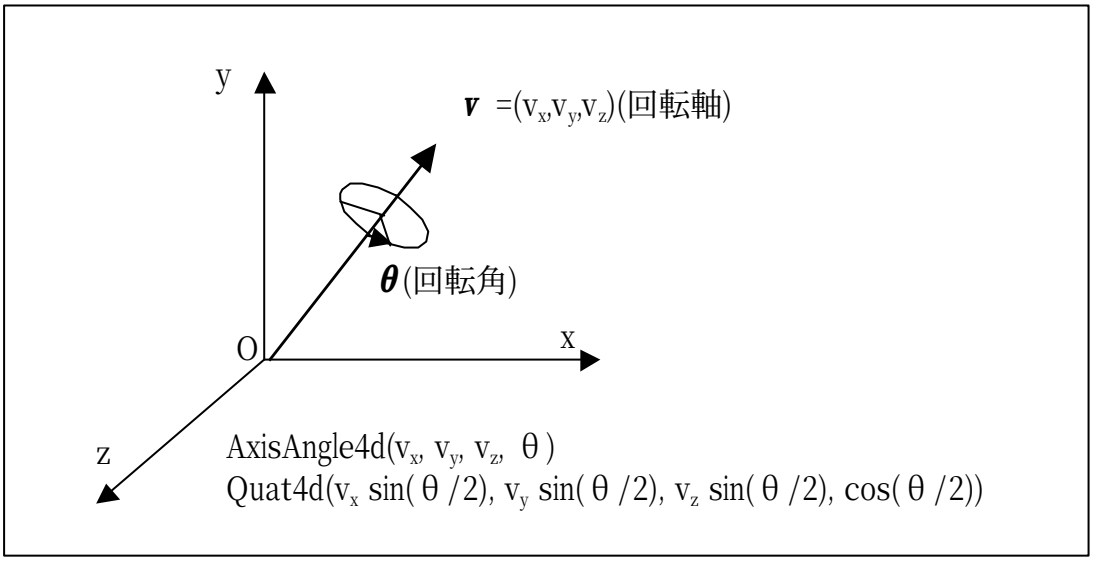

図 5　3 次元軸回りの回転

Quaternion は数学者ハミルトンによって導入された数です．クオータニオンと読み，日本語では４元数と訳されます(Quarter は ’４’ の意)．文字通り４つの要素を持つ数で，vecmath では Tuple4f,Tuple4d をそれぞれ継承した Quat4f, Quat4d というクラスでこの数を表現しています．複素数の自然拡張になっており，３次元での回転変換を表現するのに便利であるため特にコンピュータグラフィックスで最近よく利用されます．

Quaternion は代数的に非常に美しい回転変換の表現です．美しいというのは，例えば AxisAngle では AxisAngle 同士の足し算や掛け算は意味をもって定義できませんが，Quaternion ではそれらが意味をもって定義できるということです．また，マトリクスのようにアフィン変換すべてを表現しているわけではありません．３次元の回転変換の表現によくマッチした数であり，「スピナー」という回転を意味する別名を持っています．結論だけ先に簡潔に述べると，単位ベクトル $\boldsymbol{v}=(v_x, v_y, v_z)$ 回りの θ ラジアン回転を表す Quaternion $\boldsymbol{q}$ は，

$$\boldsymbol{q} = \begin{pmatrix} v_x \sin\dfrac{\boldsymbol{q}}{2} \\ v_y \sin\dfrac{\boldsymbol{q}}{2} \\ v_z \sin\dfrac{\boldsymbol{q}}{2} \\ \cos\dfrac{\boldsymbol{q}}{2} \end{pmatrix}$$

と表現されます．以下では，簡単に Quaternion を導入し回転変換との関係を解説しましょう．

定義

２つの実数で複素数を表現した時と同様に，４つの実数で Quaternion を表現します．まず，３つの単位虚数 $\boldsymbol{i},\boldsymbol{j},\boldsymbol{k}$ を導入することからはじめましょう．

$\boldsymbol{i},\boldsymbol{j},\boldsymbol{k}$ は次の演算規則に従うものとします．

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = -1$$
$$\boldsymbol{ij} = \boldsymbol{k},\ \boldsymbol{jk} = \boldsymbol{i},\ \boldsymbol{ki} = \boldsymbol{j}$$
$$\boldsymbol{ijk} = -1$$

そして，４つの実数要素 x, y, z, w を持つ Quaternion $\boldsymbol{q}$ を以下のように定義します．

$$\boldsymbol{q} = (x, y, z, w) = x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} + w = ((x, y, z), w) = (\boldsymbol{v}, w) \qquad x, y, z \in \boldsymbol{R}, \boldsymbol{v} \in \boldsymbol{R}^3$$

w が実部を示し，(x,y,z)が虚部を示します．ここでは，紙面節約のために列ベクトルをカンマで区切って行ベクトルとして表現しています．

性質

Quaternion は３つの虚数単位の演算規則を適用することで，和，スカラー倍，積，等が計算できます．また，以下の性質が定義，導出されます．

表 8　Quaternion の性質

意味	数学的表記	vecmath の表記
定義	$\boldsymbol{q} = (x, y, z, w) = x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} + w = (\boldsymbol{v}, w) \quad x, y, z \in \boldsymbol{R}, \boldsymbol{v} \in \boldsymbol{R}^3$	q.set(x,y,z,w)
和	$\boldsymbol{q}_1 + \boldsymbol{q}_2 = (x_1 + x_2, y_1 + y_2, z_1 + z_2, w_1 + w_2) = (\boldsymbol{v}_1 + \boldsymbol{v}_2, w_1 + w_2)$	q1.add(q2) q.add(q1,q2)
スカラー倍	$k\boldsymbol{q} = (kx, ky, kz, kw) = (k\boldsymbol{v},\ kw)$	q.scale(k) q.scale(q1, k)
積	$\boldsymbol{q}_1\boldsymbol{q}_2 = (\boldsymbol{v}_1 \times \boldsymbol{v}_2 + w_1\boldsymbol{v}_2 + w_2\boldsymbol{v}_1,\ -\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 + w_1 w_2)$	q1.mul(q2) q.mul(q1, q2)
ノルム(長さ)	$\lvert \boldsymbol{q} \rvert = \sqrt{x^2 + y^2 + z^2 + w^2} = \sqrt{\lvert \boldsymbol{v} \rvert^2 + w^2}$	
単位化	$\boldsymbol{q} / \lvert \boldsymbol{q} \rvert$	q.normalize() q.normalize(q1)
共役	$\boldsymbol{q}^* = (-x, -y, -z, w) = -x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k} + w = (-\boldsymbol{v}, w)$	q.conjugate() q.conjugate(q1)
共役との積	$\boldsymbol{q}\boldsymbol{q}^* = \boldsymbol{q}^*\boldsymbol{q} = x^2 + y^2 + z^2 + w = \lvert \boldsymbol{q} \rvert^2$	
逆元	$\boldsymbol{q}^{-1} = \boldsymbol{q}^* / \lvert \boldsymbol{q} \rvert^2$	q.inverse()
逆との積	$\boldsymbol{q}\boldsymbol{q}^{-1} = \boldsymbol{q}^{-1}\boldsymbol{q} = 1$	

ある Quaternion に対して共役 Quaternion とは，実部のみもとの Quaternion と同じで虚部がもとの値のマイナスになっているものです．Quaternion とその共役 Quaternion との積はスカラーになり，Quaternion のノルム(長さ)の２乗になります．

特に，ノルム(長さ)が１の Quaternion を単位 Quaternion と呼びます，単位 Quaternion は３次元の回転変換と結びつく興味深い性質があり，vecmath での脇役としての重要な役割を果たしますので，次に説明します．

単位 Quaternion と回転変換

ここでは，単位 Quaternion が３次元でのある軸回りの回転を表すのに非常に適した数学的な表現であることを説明したいと思います．

表９　　単位 Quaternion の性質

意味	数学的表記
単位 Quaternion の定義	$\|\boldsymbol{q}\|=1$
逆 Quaternion	$\boldsymbol{q}^{-1}=\boldsymbol{q}^{*}$
三角関数表現	$\boldsymbol{q}=(\boldsymbol{v}\sin\Theta,\cos\Theta)$　　$^{\exists}\boldsymbol{v}\in\boldsymbol{R}^3,\|\boldsymbol{v}\|=1,^{\exists}\Theta\in\boldsymbol{R}$

単位 Quaternion では，逆 Quaternion は共役 Quaternion そのものです．また，単位 Quaternion の三角関数表現を見ると，絶対値１の複素数の表現と非常によく似ていることがわかります．単位 Quaternion も，$\boldsymbol{v}$ とΘを適当に選べば絶対値１の複素数と同じ形式で記述できます．数学に詳しい人なら，複素平面において絶対値１の複素数の掛け算が回転変換を表現することを思い出すと思います．驚くべきことに，Quaternion においてもこの表記は，３次元空間での $\boldsymbol{v}$ を回転軸としたベクトルの回転を表現しているのです．

では，点の回転を Quaternion で表現することを考えてみましょう．点の回転を Quaternion で回転を表現するためには，３次元の点と Quaternion の演算を定義しなければなりません．３次元の点 $\boldsymbol{P}$ を４次元同次ベクトルで表し，それを Quaternion で表現しましょう．そして Quaternion の積の規則を使って，$\boldsymbol{P}\rightarrow\boldsymbol{P}'$ の変換 $\boldsymbol{T}$ を次のように定義します．

$$\boldsymbol{p}'=T(\boldsymbol{p})=\boldsymbol{q}\boldsymbol{p}\boldsymbol{q}^{-1}$$

これで，点と Quaternion の演算が可能になります．実際に単位 Quaternion $\boldsymbol{q}$ と点 $\boldsymbol{P}$ について，前の変換を計算してみましょう．

$$\boldsymbol{q}=(\boldsymbol{v}\sin\Theta,\cos\Theta),\ \boldsymbol{p}=(\boldsymbol{u},w)\qquad|\boldsymbol{v}|=1$$

として，表９に記述した積の性質を利用して計算を行うと，

$$\boldsymbol{p}'=\boldsymbol{q}\boldsymbol{p}\boldsymbol{q}^{-1}=(-\sin^2\Theta\ \boldsymbol{u}\times\boldsymbol{v}\times\boldsymbol{u}+\cos^2\Theta\ \boldsymbol{u}-2\sin\Theta\cos\Theta\ \boldsymbol{u}\times\boldsymbol{v}+\sin^2\Theta(\boldsymbol{u}\cdot\boldsymbol{v})\ \boldsymbol{v},\quad w)$$

となります．少し複雑に見えますが，$\boldsymbol{P}'$ はベクトル $\boldsymbol{v}$ を軸として $\boldsymbol{P}$ を２Θラジアン回転したものになっていることを示しましょう．$\boldsymbol{u}$ を $\boldsymbol{v}$ と平行な成分 $\boldsymbol{u}_{//}$ と $\boldsymbol{v}$ と直交する成分 $\boldsymbol{u}_{\perp}$との２つのベクトルの和であると考えます．

$$\boldsymbol{p}=(\boldsymbol{u}_{//}+\boldsymbol{u}_{\perp},\quad w)\qquad\boldsymbol{u}_{//}=(\boldsymbol{u}\cdot\boldsymbol{v})\boldsymbol{v},\ \boldsymbol{u}_{\perp}=\boldsymbol{u}-\boldsymbol{u}_{//}$$

$\boldsymbol{u}_{//}$ と $\boldsymbol{u}_{\perp}$はその定義から以下の性質があります．

$$\boldsymbol{v}\times\boldsymbol{u}_{//}=\boldsymbol{u}_{//}\times\boldsymbol{v}=\boldsymbol{0},\qquad\boldsymbol{u}_{\perp}\cdot\boldsymbol{v}=0,\qquad\boldsymbol{u}_{\perp}\times\boldsymbol{v}\times\boldsymbol{u}_{\perp}=\boldsymbol{u}_{\perp}$$

これらを使って，点の Quaternion による変換の計算をさらに進めると，

$$\boldsymbol{p}'=\boldsymbol{q}\boldsymbol{p}\boldsymbol{q}^{-1}=(\boldsymbol{u}_{//}+\cos2\Theta\ \boldsymbol{u}_{\perp}+\sin2\Theta\ \boldsymbol{v}\times\boldsymbol{u}_{\perp},\quad w)$$

というすっきりした式に置きかえることができます．計算途中で三角関数の２倍角の公式を使っています．$\boldsymbol{v}\times\boldsymbol{u}_{\perp}$は，$\boldsymbol{v}$ と $\boldsymbol{u}_{\perp}$ の両方に直交し，$\boldsymbol{u}_{\perp}$と同じ長さのベクトルになります．この式を見ると，$\boldsymbol{u}$ の $\boldsymbol{v}$ 軸に平行成分はそのままにして，垂直成分のみを軸の回りに２Θラジアン回転させていることが分かるでしょう．

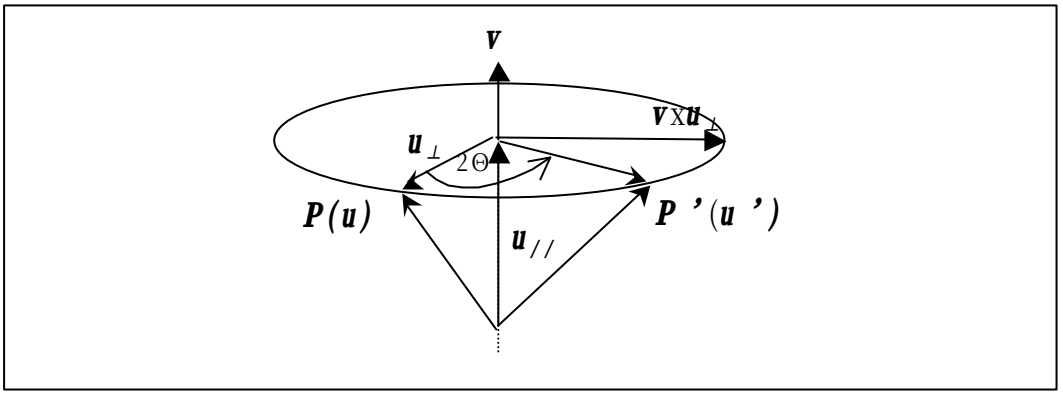

図６ Ｑｕａｔｅｒｎｉｏｎ による回転

すなわち，単位ベクトル $\boldsymbol{v}=(v_x, v_y, v_z)$ 回りのθラジアン回転を表す Quaternion $\boldsymbol{q}$ は，$\theta=2\Theta$ と置けば，

$$\boldsymbol{q}=\left(\boldsymbol{v}\sin\frac{\theta}{2},\cos\frac{\theta}{2}\right)=\left(v_x\sin\frac{\theta}{2},v_y\sin\frac{\theta}{2},v_z\sin\frac{\theta}{2},\cos\frac{\theta}{2}\right)$$

である，という非常に美しい結論が得られました．

大円補間

Quaternion のもう１つの大きな特長は，２つの Quaternion の補間ができるということです．ある回転と別の回転との中間にあたる回転を求めたり，２つの回転の間を滑らかにつなぐ回転の計算を得意とします．一般の３次元の回転は，回転軸が固定されているならば角度のみの補間でかまいませんが，回転軸が変化する場合 Matrix や AxisAngle ではうまい補間法がありません．回転をオイラー角のように x,y,z 軸の回転に分解する方法でも，分解方法が一意でないことから，補間は難しいとされています．

単位 Quaternion は，４次元空間で原点を中心とする半径１の球の表面にマップして考えることができます．すなわち，どんな単位 Quaternion でもその球面上の１点と考えることができるのです．これは，長さ１の Quaternion が $x^2 + y^2 + z^2 + w^2 = 1$ を満たすことから容易に想像できるでしょう．Quaternion の補間では，球面上の２つの Quaternion を結ぶ大円(球面上の半径最大の円)を考え，その大円上で補間を行います．この大円補間は Quat4d.interpolate(q1,q2,t) メソッドで提供され，以下のように定式化されます．

$$\boldsymbol{q}=\frac{\sin(1-t)\theta}{\sin\theta}\boldsymbol{q}_1+\frac{\sin t\theta}{\sin\theta}\boldsymbol{q}_2 \qquad \theta=\angle\boldsymbol{q}_1\boldsymbol{q}_2=\cos^{-1}(\boldsymbol{q}_1\cdot\boldsymbol{q}_2)$$

この式の意味を理解するために，まず簡単な幾何の定理を図７左に示しました．ここでは，長さ１の３つのベクトル $\boldsymbol{a}$，$\boldsymbol{b}$，$\boldsymbol{c}$ が，角度α，βをなして並んでいます．この３つのベクトルには，

$$\boldsymbol{c}\sin(\alpha+\beta)=\boldsymbol{a}\sin\beta+\boldsymbol{b}\sin\alpha$$

の関係が成り立つことが知られています．この定理をふまえて，大円での断面図上に３つの Quaternion を描いたものを図７右に示しました．前述の定理から，Quaternion の大円補間の意味が理解できると思います．

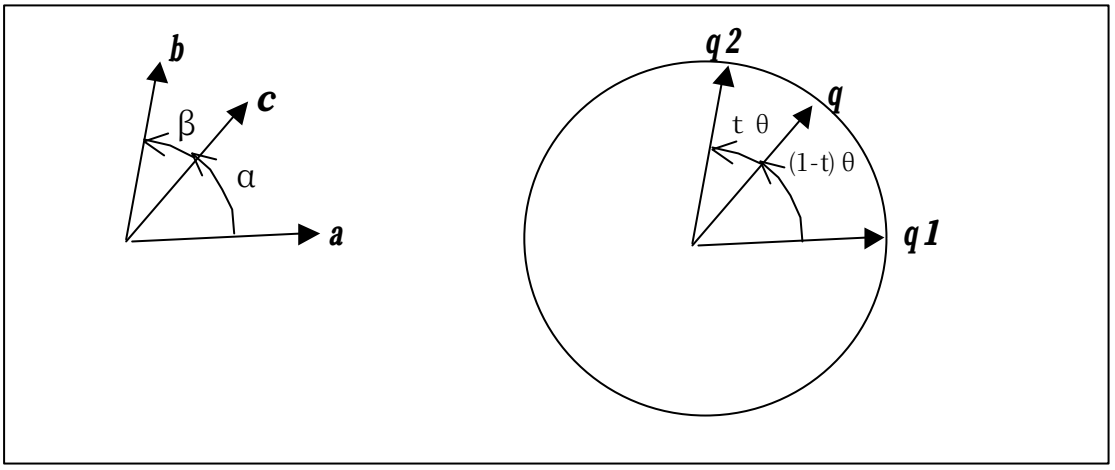

図 7　Quaternion の大円補間

回転表現の変換

回転を表現するには，AxisAngle, Quaternion, Matrix という別々のクラスがありますが，vecmath ではそれらを互いに変換することができるように，それぞれのクラスにメソッドが容易されています．名前はすべて set()です．引数の型によってオーバーロードされていて，渡す引数によって正しいメソッドが選ばれます．以下の記述では，m は Matrix4d，a は AxisAngle4d, q は Quat4d 型の変数であるとしています．

表 10　各種回転クラスの変換

変換	呼び出し方	変換メソッド
Matrix4d → AxisAngle4d	a.set(m)	AxisAngle4d.set(Matrix4d)
AxisAngle4d→Matrix4d	m.set(a)	Matrix4d.set(AxisAngle4d)
Matrix4d→Quat4d	q.set(m)	Quat4d.set(Matrix4d)
Quat4d→Matrix4d	m.set(q)	Matrix4d.set(Quat4d)
Quat4d→AxisAngle4d	a.set(q)	AxisAngle4d.set(Quat4d)
AxisAngle4d→Quat4d	q.set(a)	Quat4d.set(AxisAngle4d)
凡例	m … Matrix4d a … AxisAngle4d q … Quat4d	

4x4 マトリクス

4x4 マトリクスである Matrix4d, Matrix4f は vecmath の中で非常に重要な働きをする中心的なクラスです．メソッド数も非常に多くなります．ここでは，このマトリクスを詳しく説明したいと思います．4x4 マトリクスは前に述べたようにアフィン変換を表現することがほとんどです．アフィン変換を考える場合，マトリクスを次のように４つの区画に区切って考えます．

$$\boldsymbol{M} = \begin{pmatrix} \begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{bmatrix} & \begin{bmatrix} m03 \\ m13 \\ m23 \end{bmatrix} \\ \begin{bmatrix} m30 & m31 & m32 \end{bmatrix} & \begin{bmatrix} m33 \end{bmatrix} \end{pmatrix} = \begin{pmatrix} \begin{bmatrix} RotationScale \end{bmatrix} & \begin{bmatrix} Translation \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 1 \end{bmatrix} \end{pmatrix}$$

- 左上 3x3

左上の 3x3 部分で，スケールと回転を表現します．この区画は API 上で RotationScale と呼ばれます．例えば Matrix4d.setRotationScale(Matrix3d) というメソッドは，この区画のみ与えられた Matrix3d で変更します．

- 右上 3x1

右上の列ベクトルは，平行移動を表現します．この区画は API 上で Translation と呼ばれます．例えば Matrix4d.setTranslation(Vector3d)というメソッドは，この区画のみ与えられた Vector3d で変更します．

- 左下 1x3

左下の行ベクトルは，アフィン変換においては常に０が３つ並びます．この区画に０以外が入ると，その変換は射影変換と呼ばれ，遠近感を持ったパース変換などに使用されます．射影変換では，平行線が変換後にも平行線であるというアフィン変換の特徴は失われてしまいます．

- 右下 1x1

右下の m33 要素は，アフィン変換では常に１です．この値を変えることで全体をスケールすることができますが，通常はこの要素は１以外の値で使用されません．

さて，右上の３ｘ３区画である RotationScale ですが，この部分は回転とスケールの変換が混ざってしまうため，どの要素が回転を示しどの要素がスケールを示すということが言えません．しかし，数学的に回転とスケールを掛け算の形で分離することが可能です．すなわち，RotationScale 区画を $\boldsymbol{M}_{3x3}$ とすると，

$$\boldsymbol{M}_{3\times3} = \boldsymbol{S}(s)\boldsymbol{R} \qquad |\boldsymbol{R}| = 1,\ \boldsymbol{S}(s) = s\boldsymbol{I}$$

と分解することができます．$\boldsymbol{R}$ の行列式を１とすることで，スケール成分を $\boldsymbol{S}$ に追い出しています．API 上には，Matrix4d.setScale(double) とか Matrix4d.setRotaion(Matrix3d) というメソッドがありますが，これらのメソッドではマトリクスの RotationScale 区画を一旦スケール成分と回転成分に分解し，おのおのを指定された成分で置き換えてから再度合成して $\boldsymbol{M}_{3x3}$ をセットしています．

表 11 に Matrix4d の主要なメソッドをあげました．表中，m00 等は成分１つ，m[][]は２次元配列，m[]は１次元配列，m,m’,m1,m2 は 4x4 マトリクス，m3x3 は 3x3 マトリクス，v はベクトル，q は Quaternion，a は AxisAngle，p は点，s はスカラー，i, j は要素を示す添え字を示します．

表 10　Matrix4d の主要メソッド

意味	詳細	vecmath の表記
全要素をリセット	全要素個別指定　M ← [m_{ij}]	Matrix4d(m00,…,m33), m.set(m00,…,m33)
	4x4 配列指定	Matrix4d(m[4][4]), m.set(m[4][4])
	16 配列指定	Matrix4d(m[16]) m.set(m[16])
	4x4 マトリクス指定　M’←M	Matrix4d(m), m’.set(m)
	0 マトリクスにセット　M ← O	Matrix4d(), m.setZero()
	単位マトリクスにセット　M← I	m.setIdentity()
アフィン変換で全要素をリセット	回転・スケール・平行移動	Matrix4d(q, v, s), m.set(q, v, s)

		Matrix4d(m3x3, v, s), m,set(m3x3, v, s)
	回転	m.set(q)
		m.set(m3x3)
		m.set(a)
	x 軸回り回転	m.rotX(θ)
	y 軸回り回転	m.rotY(θ)
	z 軸回り回転	m.rotZ(θ)
	スケール M ← S(s)	m.set(s)
	平行移動 M ← T(v)	m.set(v)
	平行移動・スケール　M←S(s)・T(v)	m.set(v,s)
	スケール・平行移動　M←T(v)・S(s)	m.set(s,v)
成分のみセット	回転(マトリクス)	m.setRotation(m3x3)
	回転(Quaternion)	m.setRotation(q)
	回転(AxisAngle)	m.setRotation(a)
	スケール	m.setScale(s)
	平行移動	m.setTranslation(v)
	回転・スケール混合	m.setRotationScale(m3x3)
成分取得	回転(マトリクス)	m.get(m3x3)
	回転(Quaternion)	m.get(q)
	平行移動	m.get(v)
	スケール	s = m.getScale()
	回転・スケール混合	m.getRotationScale(m3x3)
	回転・スケール・平行移動	s = m.get(m3x3, v)
要素セット	m_{ij}←value	m.setElement（i,j,value）
要素取得	value←m_{ij}	value= m.getElement(i,j)
列セット	ベクトル	m.setColumn(j, v)
	配列	m.setColumn(j, m[4])
	要素	m.setColumn(j, x, y, z, w)
列取得	ベクトル	m.getColumn(j, v)
	配列	m.getColumn(j, m[4])
行セット	ベクトル	m.setRow(i, v)
	配列	m.setRow(i, m[4])
	要素	m.setRow(i, x, y, z, w)
行取得	ベクトル	m.getRow(i, v)
	配列	m.getRow(i, m[4])
逆行列	M ← M^{-1}	m.invert()
	M ’ ← M^{-1}	m ’.invert(m)
行列式	s←$\|M\|$	s = m.determinant()
点の変換	p ← Mp	m.transform(p)
	p ’←Mp	m.transform(p, p ’)
ベクトルの変換	v ← Mv	m.transform(v)
	v ’←Mv	m.transform(v, v ’)
転置	M ← ^tM	m.transpose()
	M ’ ← ^tM	m ’.transpose(m)
積	M ← M1 M2	m.mul(m1,m2)
	M ← M M1	m.mul(m1)
転置積 1	M ← M1 ^{t}M2	m.mulTransposeRight(m1,m2)
	M ← M ^{t}M1	m.mulTransposeRight(m1)
転置積 2	M ← $^tM1$ M2	m.mulTransposeLeft(m1,m2)
	M ← ^tM M1	m.mulTransposeLeft(m1)

転置積 3	M ← ᵗM1 ᵗM2	m.mulTransposeBoth(m1,m2)
	M ← ᵗM ᵗM1	m.mulTransposeBoth(m1)
凡例	m,m1,m2 … Matrix4d m3x3 … Matrix3d v … Vector3d q … Quat4d p … Point3d a … AxisAngle4d s … スカラー i,j … 添え字 mij … マトリクス要素 m[] … 1 次元配列 m[][] … 2 次元配列	

3D の点およびベクトルの変換は，transform() メソッドで行われます．与える引数が Point3d か Vector3d かで得られる結果が異なることに注意してください．transform()メソッドは型でオーバーロードされています．引数が Point3d の場合は，４番目の要素 w = 1 と拡張した 4 次元ベクトルしてマトリクスとの掛け算が行われますが，ベクトルの場合は４番目の要素 w = 0 として拡張した 4 次元ベクトルで計算されます．すなわち，点の変換はマトリクス第４列の平行移動成分の影響を受けますが，ベクトルはこの成分の影響を受けません．

プログラミング Tips

ここでは，vecmath を利用する上での，プログラミング上の Tips をいくつか挙げてみます．特に速度に関してより有利なコーディング上のイディオムを紹介していきます．

高速な距離計算

距離という非常に直感的な概念が，ベクトルと点のメソッドをリストした表４では４種類示されています．これらの違いを指摘しておきます．

L_2 距離は最も一般的に使用される距離で，ある点の近くを示す時に半径 r 以内という使い方ができます．すなわち，３次元の２点 PQ 間の L_2 距離は $\sqrt{(p_x-q_x)^2+(p_y-q_y)^2+(p_z-q_z)^2}$ と定義されます．L_2 の２乗は，ルートの計算を省いたものです．L_2 距離の特徴は，もっとも人間の直感に合っていることですが，短所として計算に時間がかかることがあげられます．もし，「近傍」という曖昧な近さの判定だけでよいなら，より計算時間がかからない L_2 以外の距離概念を用いることができます．

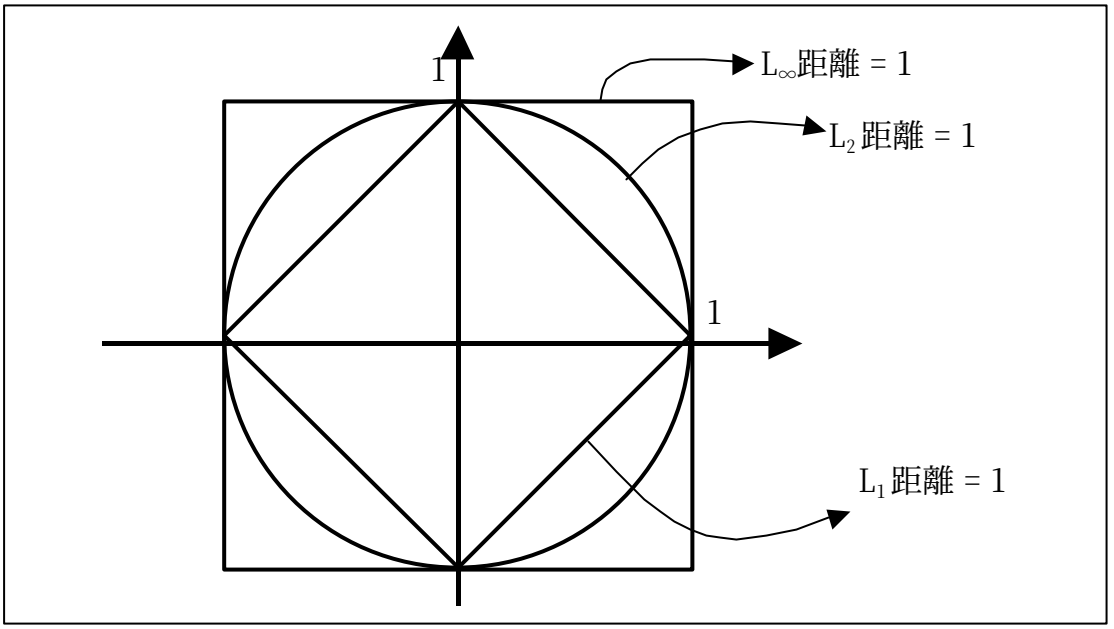

図 8　等距離線

図 8 に，３つの距離を概念的に示すために，それぞれの距離の定義に基づいて原点から１だけ離れた等距離線を描いてみました．この図から分かるとおり，３つの距離はどれも「近傍」を表現しています．計算機上は L_∞ がもっとも速く計算できるので，近くにあるというラフなチェックでよければそれを使うべきです．このチェックに特化したメソッドとして epsilonEquals() が用意されています．これは，与えられた距離以内にあるかどうかを，L_∞ 距離を用いてすばやく判定するものです．このメソッドを用いるのが最も効率的ですが，どうしても L_2 距離による近傍判定が必要な場合でも，distance() をそのまま用いずに，distanceSquared() を用いて２乗した距離値と比べることによって時間が掛かる平方根の計算を省くことが可能です．

ちなみに，L_α 距離は，次のように定義されています．

$$\left(| p_x - q_x |^a + | p_y - q_y |^a + | p_z - q_z |^a \right)^{1/a}$$

ここで，αが１ならば差の絶対値の足し算だけになり，L_1 の式となります．また，αが∞の場合，差の絶対値が最大のものが支配的になり，L_∞ の式となります．αが２の場合が通常の距離です．

逆行列より直交行列の転置

マトリクスの逆マトリクス計算には時間が掛かります．特に 3x3 行列が単位直交であるとき，すなわちスケール成分を持たず回転成分のみで構成される場合には，逆マトリクスは転置マトリクスで代用できます．メソッドとしては，逆マトリクスを求める invert() でなく転置マトリクスを求める transpose() を使います．transpose() は行列の要素を入れ替えるだけなので高速に計算できます．

$$\boldsymbol{M}_{3\times3}^{-1} = {}^t\boldsymbol{M}_{3\times3}$$

またスケール成分を持つ場合でも，

$$\boldsymbol{M}_{3\times3}^{-1} = \frac{1}{s}\ {}^t\boldsymbol{M}_{3\times3}$$

の計算で高速化可能です．4x4 行列においても，マトリクスがアフィン変換であり平行移動成分をもたない場合は，転置マトリクスがそのまま使えます．平行移動成分を持つ場合でも，

$$\boldsymbol{M}^{-1} = \begin{pmatrix} \boldsymbol{M}_{3\times3}{}^{-1} & -\boldsymbol{M}_{3\times3}{}^{-1}\boldsymbol{t} \\ 0 & 1 \end{pmatrix}$$

という計算手法でより高速に逆行列を求めることができます．このように，もし利用する変換の種類が分かっている場合は，その知識を使って逆行列の計算を高速化できる可能性があるのです．

実は Java3D には Transform3D という変換クラスがもう１つ用意されています．これは vecmath には含まれませんが，ほぼ Matrix4d と同じメソッドとデータで構成されています．このクラスは内部で変換のタイプによるカテゴリ分けを行っていますので，より高速に逆行列を求めることが可能です．Transform3D から Matrix4d 等を取りだしてからその逆行列を求めるよりも，Transform3D の invert() メソッドをそのまま利用した方が高速なのです．

再ノーマライズ

マトリクスを積算する場合，徐々に誤差が蓄積されることがあります．最初は綺麗な正規直交マトリクスだったものでも，誤差が積もり積もって徐々にその正規性，直交性が劣化することがよくあります．このような場合，適当なタイミングでマトリクスをノーマライズして「洗濯」するとよいことが知られています．Matrix を表現する各クラスには，normalize() というメソッドがあります．このメソッドは，マトリクスをスケール，回転，平行移動の各成分に分解して，スケールを１に補正した後再度組み立てます．これを行うことで，頻繁に行われる変換の誤差の蓄積を回避することが可能です．これは，Quaternion にも当てはまります．また，Transform3D には autoNormalize と言う属性があり，これをセットすると Transform3D の変更時に自動的にノーマライズすることが可能です．

内・外積と角度

２つのベクトル $\boldsymbol{a}, \boldsymbol{b}$ のなす角度を内積から逆算するには，

$$\boldsymbol{q} = \cos^{-1}\left(\frac{\boldsymbol{a}\cdot\boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|}\right) \qquad (0 \le \boldsymbol{q} \le \boldsymbol{p})$$

とします．この場合，角度Θには向きがありません．２ベクトル間の角が０ラジアンからπラジアン（180°）の間で求まります．$\cos^{-1}$ の計算には，java.lang.Math クラスの acos() メソッドが使用できます．また，vecmath の Vector クラスには，angle() メソッドが用意されており，０ラジアンからπラジアンの間の角度が計算されます．この方法は，ベクトルが２次元であっても３次元であっても，またｎ次元であっても適用できる汎用的な方法です．

外積からも角度を逆算することができます．まず２次元ベクトルの角度について考えます．２次元ベクトルでは，外積もスカラーであることに注目してください．

$$\boldsymbol{a}\times\boldsymbol{b} = a_x b_y - a_y b_x$$

$$\boldsymbol{q} = \sin^{-1}\left(\frac{\boldsymbol{a}\times\boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|}\right) \qquad \left(-\frac{\boldsymbol{p}}{2} \le \boldsymbol{q} \le \frac{\boldsymbol{p}}{2}\right)$$

この方法ではπ/2 以上の角度が測れません．角度Θがπ/2 を超えてしまうと，(π-Θ)の値が求まってしまうという短所があります．その代わり，２次元平面上で反時計回りを正として $\boldsymbol{a}$ から $\boldsymbol{b}$ に向かう方向が符号付きの角度で計算できます．

３次元でも外積からも角度を逆算することはできますが，外積はベクトルとなるため，その長さを使う必要があります．

$$\theta = \sin^{-1}\left(\frac{|\boldsymbol{a}\times\boldsymbol{b}|}{|\boldsymbol{a}||\boldsymbol{b}|}\right) \qquad \left(0 \le \theta \le \frac{\pi}{2}\right)$$

この場合もπ/2 以上の角度が測れません.

通常，２つのベクトルの角度を求めるには，内積から角度を求めますが，その場合，数値計算上気をつけなくてはいけないことがあります．Θが０に近い場合，cos の逆関数($\cos^{-1}$)は誤差を増幅します．$\Theta = \cos^{-1} x$ のグラフは，Θ=0 の付近でほぼ垂直になるため，ほんの少し内積の計算値がずれるだけでΘも大きくずれてしまいます．Θが０に近いことが分かったら，$\cos^{-1}$ すなわち java.lang.Math クラスの acos() メソッド使って角度を求めるのは精度の低下を簡単に招きます．sin の逆関数　asin() メソッド - を使うか，tan の逆関数 atan() メソッド - を使うことをお勧めします.

実はとっておきの方法として，どんな場合でも数値計算的に安定して角度を求めることができる手法があります．それは atan2() と呼ばれる２変数の関数を用いる方法です．atan2() は２つの引数 y, x をこの順で取り，ベクトル(x,y)と x 軸のなす角を求める関数です．java では java.lang.Math クラスにこのメソッドが提供されています．２つのベクトルのなす角度を求めるには，atan2 の２つの引数に，外積と内積を与えます．2 次元の場合は，

$$\theta = \tan^{-1}\left(\frac{\sin\theta}{\cos\theta}\right) = \operatorname{atan2}(\boldsymbol{a}\times\boldsymbol{b}, \boldsymbol{a}\cdot\boldsymbol{b}) \qquad (2\text{次元}, \ -\pi \le \theta \le \pi)$$

として角度が求まります．外積 **a**x**b** がスカラーであることから，その値（符号付き）を第一引数にそのまま与えます．角度は **a** から **b** に反時計回りに向かう符号つきであり，-πラジアン(-180°)～πラジアン(180°)の値となります．3 次元の場合，外積 **a**x**b** がベクトルになるためその長さを使って atan2 の第一引数とします.

$$\theta = \tan^{-1}\left(\frac{|\sin\theta|}{\cos\theta}\right) = \operatorname{atan2}(|\boldsymbol{a}\times\boldsymbol{b}|, \boldsymbol{a}\cdot\boldsymbol{b}) \qquad (3\text{次元}, \ 0 \le \theta \le \pi)$$

３次元の場合は，向きなしの 0 ラジアン～πラジアン(180°)の値が求まります．acos() を使った方法と数学的には同じですが，０ラジアン付近の誤差はずっと少なくなっています.

このように，内積と外積を両方計算し atan2() メソッドを使用すれば２つのベクトルのどちらかが 0 に非常に近い場合を除いては角度の計算は非常に安定しています．ただし，vecmath の各 Vector クラスが持っている angle() メソッドは，安全な実装なら atan2 を，速度を優先する実装なら，acos を使った実装になっているでしょう．Sun の仕様はその実装方法までは規定していません．Sun の vecmath 実装を使ってみると，acos をそのまま使った後者の実装になっているようです．もしΘ=0 付近での角度計算が不安定になるようなら，自分で atan2() を使った手法に変えてみることをお勧めします.

```
// 2D の場合

Vector2d a = new Vector2d(1,2);
Vector2d b = new Vector2d(2,3);

double theta = a.angle(b);              // a,b のなす角(通常の方法)
double theta2 = Math.atan2(a.x*b.y   a.y*bx, a.dot(b));    // a,b のなす角(正確，符号付き)

// 3D の場合
Vector3d a = new Vector3d(1,2,3);
Vector3d b = new Vector3d(2,3,4);
Vector3d c = new Vector3d()

double theta = a.angle(b);              // a,b のなす角(通常の方法)
c.cross(a, b);                           // c = axb
double theta2 = Math.atan2(c.length(), a.dot(b));    // a,b のなす角(正確)
```

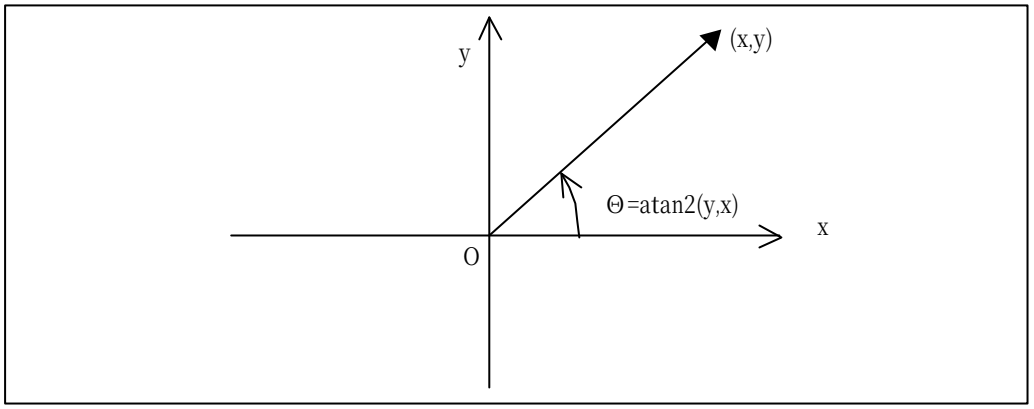

図 9 atan2 関数

他の利用法(連立方程式を解く)

最後に，vecmath パッケージの興味深い別の用途について述べてみたいと思います．vecmath の API を利用して，連立一次方程式を解くことができます．これは，3x3 および 4x4 マトリクスが逆行列を求めるメソッドを用意していることを利用しています．例えば，以下の方程式を解いてみましょう．

$$\begin{aligned} x + y + z &= 6 \\ x - z &= 2 \\ y - z &= 1 \end{aligned}$$

簡単な３元１次連立方程式です．これを，マトリクスとベクトルを用いて，$\mathbf{A}\boldsymbol{x}=\boldsymbol{b}$ の形に書くと次のようになります．

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 6 \\ 2 \\ 1 \end{pmatrix}$$

これを解くには，$\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{b}$ とすればいいでしょう．

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix}^{-1} \begin{pmatrix} 6 \\ 2 \\ 1 \end{pmatrix}$$

この計算を Matrix3d が用意する逆マトリクスメソッド invert() によって行うことができます．具体的なコーディングは次のようになります．

```
Matrix3d A = new Matrix3d(1,1,1,  1,0,-1  ,0,1,-1);
Vector3d b = new Vector3d(6,2,1);
Vector3d x = new Vector3d();

// Ax = b を解く．まず A の逆行列を求める
A.invert();
```

```
// x を求める x = A^-1 * b
A.transform(b, x);

// x をプリント
System.out.println(x);
```

連立一次方程式を解く

2,3,4 次元の連立一次方程式はこのように求めることができますが，一般の n 元連立一次方程式では，逆行列を求めることは速度上も精度上も効率的ではありません．n 元連立一次方程式では LU 分解という手法によって A を分解しておき，必要に応じて b を与えて x を求めます．同じ方程式を GMatrix と GVector を用いて LU 分解法によって解いて見ましょう．これは一般的な n 次元の連立方程式で用いることができます．

```
double [] array = {1,1,1,  1,0,-1,  0,1,-1};
double [] vector = {6, 2, 1};

// A の定義
GMatrix A = new GMatrix(3, 3, array);

// LU 分解用の領域の確保
GMatrix LU = new GMatrix(3, 3);
GVector permutation = new GVector(3);

// b ベクトルの定義
GVector b = new GVector(vector);

// x ベクトル(解)の領域
GVector x = new GVector(3);

// Ax = b を解く．まず A の LU 分解を求める
A.LUD(LU, permutation);

// x を求める
x.LUDBackSolve(LU, b, permutation);

// x をプリント
System.out.println(x);
```

LU 分解による n 元一次連立方程式の解

一般行列 GMatrix には，LU 分解法による求解の他に，SV 分解による求解メソッドが準備されています．これは特異値分解(Singular Value Decomposition)と呼ばれ，行と列の数が異なる一般連立一次方程式を解くことができます．この手法を使えば，変数が式より多いケースや逆に式が変数より多いケースにおいても「最小２乗解」と呼ばれる平均解を求めることができます．式が変数より多い場合の利用例としては，ある現象を数式でモデル化し，誤差を含む多くのサンプルからそのモデル式の最適なパラメータを求めるケースが挙げられます．逆に変数が式より多い場合の利用例としては，多くの自由度の中から与えられた初期値にできるだけ近い解を求めたいケースが挙げられます．利用例は省略しますが，GMatrix.SVD() と Gvector.SVDBackSolve() を利用し LU 分解とほぼ同じコーディングによって，SV 分解による求解ができます．

おわりに

javax.vecmath パッケージの API とその背後の数学について，やや詳しく解説しました．詳細な API 仕様については Sun の仕様書，特に Java3D ユーザーズガイドを参照することをお勧めします．3D グラフィックスとその数学は奥が深く，それだけで１冊の本が書ける内容を含んでいます．この記事では，その一部を実践的な形で紹介しまし

た．また，ここで述べたすべての API の筆者による実装ソースコードが Java3D vecmath 非公式実装という形で公開されています．

http://ObjectClub.esm.co.jp/vecmath/

上記 URL では java バージョンの他に，高速な C++ の平行移植バージョンも提供しています．この記事で紹介した数学とアルゴリズムの詳細に興味のある方は是非参照してください．

Kenji Hiranabe <hiranabe@esm.co.jp>