Fine Kernel ToolKit System (C#版)
ユーザーズマニュアル

Version 4.2.13

(Manual Revision 2)

FineKernel Project

目次

はじめに

この文章で述べられている FK (Fine Kernel) System は、容易にインタラクティブな 3D 空間を表現するための Tool Kit である。

ここでいう Tool Kit とは、システムを構築する際に用いられる簡易インターフェースをプログラミング言語から呼び出す形で実現されたもののことをいう。平易な言葉で述べるなら、この FK System を用いれば簡単にインタラクティブな 3D の世界を創造することが可能であるということである。普通、なんらかのシステム構築の際には本質的でない部分に労力をさかねばならないことは周知のとおりである。それは、ときには学習であったり、ときには作業であったり、ときには試行錯誤であったりする。ツールキットは、それらをユーザに代わって肩代りをし、より本質的な部分にのみユーザが没頭することを助ける役割を持つ。

FK System が、3D 空間の作成をサポートすることは前述したが、大きな理念としての柱が幾つかある。それを列挙すると、

- オブジェクト指向概念の採用。
- モデルに対する制御の柔軟性。
- 形状の容易な定義や変形。
- 複雑な座標系処理の簡便化。
- ディスプレイリストの概念。
- インターフェースの柔軟な構築。
- 汎用性と高速描画の両立。
- 環境との非依存。

といった事柄を特に重要視して設計が行われている。

これらの概念を、我々はまず C++ 言語を用いたクラスライブラリとして実現した。さらに Ver.2 では、この C++ 版をベースとして CLI による実装も追加した。CLI 版を用いることで、C++ 版とほぼ同一の機能を C# や F# といった.NET 対応言語で開発することが可能となった。C++ と C# の両方に共通な 3D フレームワークはあまり多くはなく、両方の言語をシームレスに扱うことができるという点が、FK System のユニークな点である。また、F# のような関数型言語においては 3D プログラミングフレームワークはあまり普及しておらず、関数型プログラミングを用いた 3D プログラミングを行いたい場合で、FK は大変有用なライブラリとなるであろう。

本書は、13 章で構成されている。

第 1 章では、FK System の基本的な考え方を理解するため、簡単なサンプルを用いて機能を紹介していく。

第 2 章では、FK システムで準備された三次元座標値や三次元ベクトルに関しての扱い方を述べる。座標やベクトルは、特に第 4、8 章の内容と著しく関わる。形状は、もちろん三次元座標で表現されるし、モデルの挙動の制御にはベクトルや座標を多用するからである。

第 3 章では、マテリアルと呼ばれるカラー属性に関して述べる。これは、形状や光源に対して色を含む質感を設定する際に用いられるパラメータのことである。非常に細かな設定が可能であるが、簡易な使用法もあることをここでは述べている。

第 4 章では、形状の扱いに関して述べる。第 4 章では、基本形状の生成法を中心に述べる。さらに、第 5 章では任意形状の動的な生成や変形方法に関して解説する。

第 6 章では、FK の中で画像表示およびテクスチャマッピングを行うための方法について述べる。さらに、特別なテクスチャマッピングとして「文字列テクスチャ」を第 7 章で解説する。画面上に文字を表示したい場合には、この 2 つの章を参

考にしてほしい。

第 8 章では、FK のプログラミングで中心的な役割を果たす「モデル」についての解説を行う。モデルに対してマテリアルや形状の設定、移動や描画の有無などを行うことになる。

第 9 章では、モデル同士の干渉・衝突判定の方法を解説する。また、光線とモデルの干渉についても解説する。

第 10、11 章は、シーンとウィンドウについて解説する。描画対象とするモデルの登録や解除を行うものである。また、複数のシーンを使い分ける方法についても解説する。

第 12 章では、簡易な形状描画手段に関しての解説を述べる。

最終章は、全体を通しての様々なトピックやエッセンス、そして簡単な例題を掲載する。

第 1 章　さあはじめよう

一般的に、3 次元コンピュータグラフィックス (以下 3DCG) のために書かれたソースコードは、かなり長くなることが多い。ただ単に直方体が回転しているプログラムを書くために、500 行以上必要な場合もある。むしろ、そうでないプラットフォームの方が珍しい。何故か？

何故なら、3DCG アプリケーションには考えなければならない要素がとても多いからである。先ほど例に出した直方体の回転に関しても、次のような要素を考慮する必要がある。

- 「直方体」をどうやって生成するか？
- 回転をどうやって実現するか？
- 立体の色はどうするのか？
- 背景色はどうするのか？
- ウィンドウをどうやって作成するのか？
- 作成したウィンドウにどうやって表示するのか？
- アニメーションをどうやって実現するのか？
- アニメーションしている間、マウスやキーボードをどう扱うのか？

これらを全てプログラムソースとして書いていくと、すぐに 500 行にも 1000 行にも簡単に達してしまう。

FK ツールキット (以下 FK) は、このような状況を打破するために産み出された。複雑に絡んでいる各要素を整理し、オブジェクト指向の概念を利用して極力簡略化できるように設計されている。実際に、直方体を回転させるプログラムを記述してみる。

1.1　直方体回転のサンプルプログラム

ソースコード 1.1　直方体回転サンプル

```
1  using System;
2  using FK_CLI;
3
4  // 直方体形状を表す変数
5  var block = new fk_Block(10.0, 20.0, 15.0);
6
7  // モデルを表す変数
8  var model = new fk_Model();
9
10 // ウィンドウを表す変数
11 var window = new fk_AppWindow();
12
13 // モデルに直方体を設定
14 model.Shape = block;
15
```

```
16 // モデルの色を黄色に設定
17 model.Material = fk_Material.Yellow;
18
19 // カメラの位置と方向を設定
20 window.CameraPos = new fk_Vector(0.0, 0.0, 100.0);
21 window.CameraFocus = new fk_Vector(0.0, 0.0, 0.0);
22
23 // ウィンドウにモデルを設定
24 window.Entry(model);
25
26 // ウィンドウのサイズを設定
27 window.Size = new fk_Dimension(600, 600);
28
29 // ウィンドウを開く
30 window.Open();
31
32 // メインループ
33 while (window.Update() == true)
34 {
35     // 直方体を Y 軸を中心に回転させる。
36     model.GlRotateWithVec(0.0, 0.0, 0.0, fk_Axis.Y, Math.PI / 500.0);
37 }
```

各処理の詳細な解説は次章以降に譲るとして、ここでは大きな流れを見ていく。

1.2 4 つの “レイヤー”

プログラムの実際の中身を分析する前に、FK の根幹をなす 4 種類の “レイヤー” である「形状」、「モデル」、「シーン」、「ウィンドウ」を簡単に解説する。

「形状」は、文字通り立体形状を表す。FK では、形状として直方体、球、平面、円盤、線分、点など様々なものを、変数を 1 つ定義するだけで作成することができる。また、様々な 3 次元データファイル形式を入力することもできる。もちろん、その場合も「形状」を表すには変数を 1 つ準備するだけでよい。

「モデル」は、形状に対して位置や方向などを持たせた存在である。「形状」と「モデル」の概念が分離しているのには理由がある。例えば、同じ形状を持つ 100 台の車のカーチェイスゲームを想定してみよう。このとき、100 台分全てのデータをメモリ上に確保するのは大変無駄である。しかし、100 台の車は位置も方向も速度も、おそらく色も違うことであろう。したがって、これらは別々に存在していなければならない。こんなときに「モデル」の概念が役立つ。まず「形状」として 1 個の車体を準備し、100 個の「モデル」を準備する。各モデルは形状として先ほどの車体を設定し、それぞれ固有の位置や方向や色を持てば良い。これで、データ量の節約と 100 台の車の存在を両立することができる。また、モデルは瞬時に設定する形状を変更することができるので、形状を入れ替えることで変形アニメーションを簡単に実現することもできる。

「シーン」は、複数のモデルと 1 つのカメラから成り立っており、全体で 1 つの “空間” を表現する。ここには、実際に描画するモデルを全て登録しておく。最後に紹介する「ウィンドウ」はキャンバスのようなもので、ここに「シーン」を設定することで “空間” が実際に描写される。「シーン」と「ウィンドウ」は完全に独立した存在なので、ウィンドウに描画されるシーンを簡単に切り替えたり、逆に複数のウィンドウに同じシーンを描画することも簡単にできる。

1.3 プログラムの概要

では、実際にサンプルプログラムについて解説していこう。

- 1,2 行目は C# で名前空間の省略対象を設定している。1 行目は Visual Studio が自動的に記述してくれるので、2 行目の「`using FK_CLI;`」を忘れずに行っておこう。
- 10 〜 12 行目では各種変数の宣言を行っている。各変数の詳細は以下の通りである。

表 1.1　変数の意味

変数名	解説
block	直方体形状を表す変数。
model	直方体の「モデル」を表す変数。
window	ウィンドウを表す変数。

変数を準備することは、その時点でそのクラスが表現する「もの (オブジェクト)」を作成することだと考えてくれればよい。例えば、11 行目の直方体変数の宣言は、この記述によって直方体を生成したということになる。他の変数、例えばモデルやウィンドウも全て変数の定義の時点で生成される。あとは、これらオブジェクトに対して適切な設定を行っていけばよい。

- 15 行目の記述は、様々な色 (マテリアル) を初期化するための関数で、これは何か色設定を行う前に記述しておく必要がある。
- 18 行目はモデル「model」に対して形状を設定している部分である。ここでは形状として「block」を設定している。
- 21 行目では、モデルの色として「Yellow」を採用している。ちなみに、デフォルトでは灰色が設定されている。
- 24,25 行目は、カメラに対して位置と方向を設定している。24 行目の「setCameraPos」関数は、カメラの位置を指定する関数である。また、25 行目の「setCameraFocus」はカメラの被写体の位置 — これは CG 用語で「注視点」とか「注目点」などと呼ばれている — を指定する関数である。従って、ここではカメラ位置を $(0, 0, 100)$ に置き、原点の方向を向いていることになる。
- 28 行目は、準備したモデルをウィンドウに登録している部分である。FK では、形状やモデルは単に準備しただけでは表示対象とはならず、このようにウィンドウに登録して初めて実際に描画されるようになる。
- 34 行目は window を実際に描画する関数である。この関数を呼んだ時点ではじめてウィンドウが実際に画面に現れる。
- 36 〜 40 行目は while ループとなっており、これがプログラムの「メインループ」となる。36 行目の while 文の中にある「window.Update()」は現在の各モデル等に設定された情報に従い、3D シーンを再描画するものである。while 文中で各モデルの変化を記述していくことで、アニメーションプログラムが実現されている。なお、「Update()」メソッドは正常に表示されている場合は true を返し、表示されていない場合に false を返す仕様となっている。従って、この while ループはウィンドウが閉じられた場合に終了する仕組みとなる。
- 39 行目では、直方体を持つモデル「model」を y 軸を中心に回転させている。「GlRotateWithVec」メソッドは、モデルを回転させるメソッドである。ここでは、原点を中心に $\frac{\pi}{360}$ ラジアン $= 0.5°$ ずつ回転させている。

1.4 作成できる “形状” の種類

FK 中で作成できる基本的な「形状」には、現在次のようなものが用意されている。

表 1.2　形状を表すクラス群

形状	クラス名	必要な引数
点	fk_Point	位置ベクトル
線分	fk_Line	両端点の位置ベクトル
ポリライン	fk_Polyline	各頂点の位置ベクトル
閉じたポリライン	fk_Closedline	各頂点の位置ベクトル
多角形平面	fk_Polygon	各頂点の位置ベクトル
円	fk_Circle	分割数、半径
直方体	fk_Block	縦、横、高さ
球	fk_Sphere	分割数、半径
角柱 (円柱)	fk_Prism	角数、上面と底面の内接円半径、高さ
角錐 (円錐)	fk_Cone	角数、底面の内接円半径、高さ
インデックスフェースセット	fk_IndexFaceSet	ファイル名等
矩形テクスチャ	fk_RectTexture	画像ファイル名
三角形テクスチャ	fk_TriTexture	画像ファイル名
メッシュテクスチャ	fk_MeshTexture	画像ファイル名
IFS テクスチャ	fk_IFSTexture	画像ファイル名
文字列板	fk_TextImage	文字列またはテキストファイル
パーティクル	fk_ParticleSet	様々な設定
光源	fk_Light	タイプ

これらの変数を定義するときは、最初に初期値として様々な設定を行うことになる。例えば fk_Block 型、つまり空間上の「直方体」を表す変数を定義するとき、その直方体の各辺長を次のようにして設定することができる。

```
var block = new fk_Block(10.0, 5.0, 20.0);
```

このように、各形状クラスにはそれぞれ初期設定の方法が用意されている。具体的な設定項目については第 4 章で詳しく述べている。

例えば、サンプルプログラムで回転する形状を直方体ではなく円盤にしたいのであれば、10 行目の直方体の部分を

```
var circle = new fk_Circle(4, 20.0);
```

と変更し、18 行目の block を circle に変更するだけでよい。

1.5　モデルの制御

FK では、モデルに対して非常に豊富な機能を提供している。FK に限らず、一般的な 3DCG のプログラム中で最も多くの作業を必要とするのがこのモデルの制御である。大抵の 3D プログラミング環境では、座標軸回りの回転、平行移動、拡大縮小といった限られた命令セットしか準備されていないことが多い。プログラマはこれらを巧みに利用してモデルを制御することになるが、この部分の実現が思いの外難しい。というのも、実現には非常に難解な数式処理を必要とするからである。FK は、プログラマがそのような数学をあまり意識することなくモデルを扱う方法を何種類も提供し、サポートしている。詳細は 8 章に全て網羅してあるので、ここではダイジェストとして一部機能を紹介する。

```
var model = new fk_Model();

// (50, 10, -20) へ移動
```

```
model.GlMoveTo(50.0, 10.0, -20.0);

// (10, 20, 0) だけ平行移動
model.GlTranslate(10.0, 20.0, 0.0);

// (0, 0, 100) の方を向かせる
model.GlFocus(0.0, 0.0, 100.0);

// モデルの向きを (1, 1, 1) にする
model.GlVec(1.0, 1.0, 1.0);

// モデルの位置を (0, 10, 0) を中心に x 軸方向に
// 0.1 ラジアン回転した位置に移動する（向きはそのまま）
model.GlRotate(0.0, 10.0, 0.0, fk_Axis.X, 0.1);

// GlRotate の機能に加え、さらに向きも回転させる
model.GlRotateWithVec(0.0, 10.0, 0.0, fk_Axis.X, 0.1);
```

また、モデルには「継承関係」というモデル同士の関係を形成することができる。これは、複数のモデルをある 1 つのモデルに属した関係にするもので、FK の中では前者を「子モデル」、後者を「親モデル」と呼んでいる。親モデルを動かすと、それに従って子モデルも動いていく。従って、この機能は複数のモデルを 1 つのモデルのように扱いたい場合に効果を発揮する。具体的な応用としては第 13 章の各サンプルが例として挙げられる。

1.6 カメラと光源

fk_AppWindow クラスは、初期状態でカメラと光源が設定されている。カメラの制御方法としては、前述したプログラムのような方法の他に、fk_Model 型変数をカメラとして扱うこともできる。詳細は第 11 章で述べる。

光源については、デフォルトでは $(0, 0, -1)$ 方向の平行光源が設定されている。これに対し、別の方向からの平行光源を設定したい場合や、点光源などを設定したい場合は fk_Light というクラスを用いて光源を作成し、それを形状としてモデルに設定し、fk_AppWindow にモデルを登録するという手順を取る。詳細は 4 の光源に関する節で説明する。

1.7 シーン

「シーン」とは、一般的には描画すべき要素の集合のことを指す。FK における「シーン」とは、モデルのデータベースとなっており、意味的には空間全体を成すものである。従って、あるモデルを描画するかどうかはシーンに対して対象モデルを登録したり抹消すればよい。

fk_AppWindow では、最初から一つのシーンが内部に登録されており、そこへの登録は fk_AppWindow の「Entry()」メソッドで行うことができる。また、抹消は「Remove()」によって行う。

一方、アプリケーションによっては複数のシーンを使い分けたい場合がある。異なるシーンをそれぞれ保持しておき、状況によってウィンドウに表示するシーンを切り替えるような場合である。そのような機能を実現する手段として、FK では「fk_Scene」というクラスが用意されている。このクラスはシーン中に表示するモデルの登録や抹消、そしてカメラを管理するものであり、このクラスの変数を複数個用意することで、複数のシーンを容易に切り替えることができる。また、マルチウィンドウアプリケーションを作成する場合にも利用することになる。さらに、シーンには霧効果の設定など、fk_AppWindow よりも高度なシーン管理機能を利用することができる。これらについては、第 10 章で詳しく解説する。

1.8 デバイス状態取得

多くのリアルタイムアプリケーションでは、マウスやキーボードなどによるリアルタイムな操作を必要とすることが多い。FK でも、現時点でマウスの位置やボタン状態、キーボードの情報などをウィンドウオブジェクトから取得することができる。また、(やや高度なトピックになるが) どの形状、どの頂点、どの面がピックされたかを取得する機能も提供されている。これらの機能は、モデラーなどを作成する際には必須の機能である。これらに関する事項は、11 章を中心に記述されている。

1.9 次の段階は.....

以上が、FK の大体の概要説明である。FK は、もともとコンテンツ作成支援と CG 研究支援の両方を目的としているため、ここでは紹介できないかなり専門的な機能もある。例えば、FK では形状を変形する機能として最新の高度な CAD 技術が用いられている。

もし、読者が CG、数学、プログラムの全てに初心者意識があるのならば、次の順番に読み進めることをお勧めする。

$$13 \to 4 \to 3 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12 \to 2 \to 13$$

ある程度の CG プログラミングの経験があるのならば、次の順番で読み進めるのが効率がよいだろう。

$$2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12 \to 13$$

なんにしろ、読み方は自由である。各自で効果的な学習を試みてほしい。

第 2 章　ベクトル、行列、四元数 (クォータニオン)

この章では、ベクトル、行列、四元数、乱数といった数学的基本クラスの利用方法を紹介する。

2.1　3 次元ベクトル

この節では、fk_Vector と呼ばれるベクトルや座標を司るクラスに関して述べる。ベクトルは、単なる浮動小数点数が 3 つならんでいるという以上に多くの意味を持つ。例えば、ベクトルは加減法、実数との積や内積、外積といった多くの演算をほどこす必要が度々現われる。そのため、fk_Vector 型は大きく 2 つの役割を持っている。ひとつは、そのようなベクトルの演算をプログラムの中で比較的容易に実現することをサポートすることであり、もうひとつは形状やモデルの挙動を制御するための引数として扱うことである。この節では、特に前者について述べている。後者に関しては 4 章 〜 8 章で順次説明していく。この節での真価は、13 章のサンプルで様々な形で現われている。

2.1.1　ベクトルの生成・設定

ベクトルの生成はいたって単純に行われる。fk_Vector 型の変数を定義すればよい。ただし fk_Vector 型はクラス型であるので、new を用いる必要がある。

```
var vec = new fk_Vector();
```

このとき、vec 変数は初期成分として $(0, 0, 0)$ が代入される。また、初期値を最初から代入することも可能である。

```
var vec = new fk_Vector(1.0, 1.0, -3.0);
```

この記述は、vec に $(1, 1, -3)$ を代入する。
もちろん、生成後の代入も可能である。次のように記述すればよい。

```
var vec1 = new fk_Vector();
var vec2 = new fk_Vector();

vec1.Set(1.0, 1.0, -3.0);
vec2.Set(5.0, -2.5);
```

Set メソッドが、vec1 や vec2 に値を代入してくれる。vec2 のように引数が 2 個しか無い場合は、z 成分には自動的に 0 が代入されるため、結果的に vec2 に $(5, -2.5, 0)$ が代入されることになる。
また、単に

```
vec.Init();
```

と記述した場合、vec はゼロベクトル $(0, 0, 0)$ となる。

ベクトル中の x, y, z の各成分はすべて public メンバとなっているので、直接参照や代入を行うことが可能である。たとえば、z 要素が正か負かを調べたいときは、

```
if(vec.z < 0.0) {
    // 後処理
}
```

と書けばよい。代入に関しても、以下のような記述が可能である。

```
vec.x = tmpX;
vec.y = tmpY;
vec.z = tmpZ;
```

ベクトルを配列として定義した場合も、もちろん各成分を直接扱うことが可能である。以下に例を示す。

```
var vec = new fk_Vector[10];

for(int i = 0; i < 10; i++) {
    vec[i] = new fk_Vector();
    vec[i].x = (double)i;
    vec[i].y = (double)-i;
    vec[i].z = 1.0;
}
```

2.1.2 比較演算

fk_Vector 型はインスタンス同士が持つ値が等しいかどうかを判定する Equals() を利用することができる。通常の int 型の変数の場合は、(a == b) のようにして値を比較できるが、fk_Vector 型はクラスであるため、== を利用した場合はインスタンスが同一であるかの判定になってしまう。このため、vecA と vecB が持つ値が等しいか否かを判定するには、(vecA.Equals(vecB)) と記述する必要がある。判定結果が bool 型として返る点は、通常の比較演算と同様である。このとき注意しなければならないのは、この比較演算ではある程度の数値誤差を許していることである。具体的に述べると、もし vec1.y と vec2.y、vec1.z と vec2.z の値がともに等しいとして、vec1.x と vec2.x が 10^{-14} 程度の差が存在しても、(vec1.Equals(vec2)) は真 (true) を返すのである。現在、この許容誤差は 10^{-12} に設定されている。ちなみに、偽の場合は false を返す。

2.1.3 単項演算子

fk_Vector 型は、単項演算子 '-' を持っている。これは、ベクトルを反転したものを返すもので、例えば、

```
vec2 = -vec1;
```

という記述は vec2 に vec1 を反転したものを代入する。この際、vec1 の値そのものは変化しない。これは、通常の int 型や double 型の変数の場合と同じ挙動であるといえる。

2.1.4　二項演算子

fk_Vector の二項演算子は多様であり、どれも実践的なものである。表 2.1 にそれらを羅列する。

表 2.1　fk_Vector の二項演算子

演算子	形式	機能	返り値の型
+	Vector + Vector	ベクトルの和	fk_Vector
-	Vector - Vector	ベクトルの差	fk_Vector
*	double * Vector	ベクトルの実数倍	fk_Vector
*	Vector * double	ベクトルの実数倍	fk_Vector
/	Vector / double	ベクトルの実数商	fk_Vector
*	Vector * Vector	ベクトルの内積	double
^	Vector ^ Vector	ベクトルの外積	fk_Vector

2.1.5　代入と演算子

fk_Vector 型はクラスであるため、代入処理ではコピーが行われず、左辺値が右辺値と同一のインスタンスを参照するようになる。したがって、

```
var vec1 = new fk_Vector();
var vec2 = new fk_Vector(1.0, 1.0, 1.0);

vec1 = vec2;
vec2.x = 3.0;
```

という記述を行った場合、vec1 と vec2 ともに $3.0, 1.0, 1.0$ という値を指すことになる。fk_Vector 型の値をコピーしたい時は、コピーコンストラクタによって明示的に指示する必要がある。例えば、

```
var vec1 = new fk_Vector();
var vec2 = new fk_Vector(1.0, 1.0, 1.0);

vec1 = new fk_Vector(vec2);
vec2.x = 3.0;
```

このように記述した場合、vec1 は $1.0, 1.0, 1.0$ という値のコピーを保持し、vec2 の値は x 成分への代入によって $3.0, 1.0, 1.0$ に変更される。

その他の代入演算子として、次のようなものが使用できる。効果は、表 2.2 の右に掲載された式と同様の働きである。なお、効果で left は左辺、right は右辺を指す。

表 2.2　fk_Vector の代入演算子

演算子	形式	効果
+=	Vector += Vector	left = left + right
-=	Vector -= Vector	left = left - right
*=	Vector *= double	left = left * right
/=	Vector /= double	left = left / right

2.1.6　ノルム (長さ) の算出

ベクトルのノルム (長さ) を出力するメソッドとして Dist() が、ノルムの 2 乗を指す Dist2() があり、それぞれ double 型を返す。使用法は次のようなものである。

```
double l = vec1.Dist();
double l2 = vec1.Dist2();
Console.WriteLine($"length = {l}");
Console.WriteLine($"length^2 = {l2}");
```

処理速度は、Dist2() の方が Dist() よりも高速である。従って、ただ単にベクトルの長さを比較したいだけならば Dist2() を利用した方が効率的である。

2.1.7　ベクトルの正規化

正規化とは、零ベクトルでない任意のベクトル $\mathbf{V}$ に対し、

$$\mathbf{V}' = \frac{\mathbf{V}}{|\mathbf{V}|}$$

を求めることである。$\mathbf{V}'$ は、結果的には $\mathbf{V}$ と方向が同一で長さが 1 のベクトルとなる。3 次元中の様々な幾何計算や、コンピュータグラフィックスの理論では、しばしば正規化されたベクトルを用いることが多い。

fk_Vector 型の変数に対し、自身を正規化 (Normalize) するメソッドとして Normalize() がある。

```
var vec1 = new fk_Vector(5.0, 4.0, 3.0);
vec1.Normalize();
```

という記述は、vec1 を正規化する。ちなみに、Normalize() メソッドは bool 型を返し、true ならば正規化の成功、false ならば失敗を意味する。失敗するのは、ベクトルが零ベクトル (つまり長さが 0) の場合に限られる。

2.1.8　ベクトルの射影

ベクトルを用いた理論では、射影と呼ばれる概念がよく用いられる。射影とは、図 2.1 にあるようにあるベクトルに対して別のベクトルを投影することである。ここで、図 2.1 の $\mathbf{P}_{proj}$ を「$\mathbf{P}$ の $\mathbf{Q}$ に対する**射影ベクトル**」と呼ぶ。

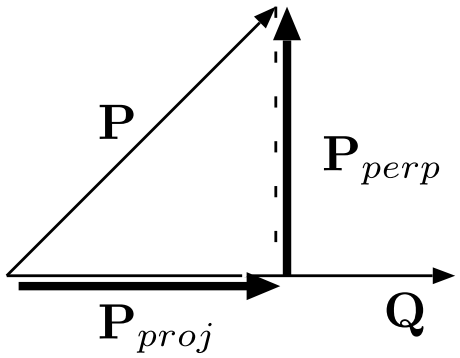

図 2.1　ベクトルの射影

この射影ベクトルを求める方法として、fk_Vector では Proj() メソッドを利用することができる。

```
var P = new fk_Vector();
var Q = new fk_Vector();
    :
var P_proj = P.Proj(Q);
```

また、投影の際の垂直成分 (図 2.1 中での $\mathbf{P}_{perp}$) を求めるときは、Perp() というメソッドを利用する。

```
P_perp = P.Perp(Q);
```

2.2 行列

この節は、FK System の持つ行列演算に関して紹介する。行列を用いることによってよりプログラムの記述が洗練されたり、あるいは高速な実行を可能にする。なお、この節の解説は線形代数、特に 1 次変換の知識があることを前提としている。初学者は、必要となるまで読み飛ばしてもらっても差し支えない。

2.2.1 FK における行列系

FK システムでは「MV 行列系」という行列系を採用している。これは、行列とベクトルの演算を以下のように規定するものである。

- ベクトルは、通常列ベクトルとして扱う。
- 行列とベクトルの積は、通常左側に行列、右側にベクトルを置くものとする。
- 行列の積が生成された場合に、ベクトルに先に作用するのは右項の行列である。

一般に、行列を扱う数学書においては、上記の MV 行列系を前提として記述されているものがほとんどであり、それらの理論を参考にするのに都合が良いと言える。一方で、3DCG の開発システムにおいて、MV 行列系とは逆の「VM 行列系」を採用しているものもあり[1)]、それらの理論や実装を参考にする際には注意が必要である。

2.2.2 行列の生成

一般的に、3DCG の世界で頻繁に用いられる行列は 4 行 4 列の正方行列である。これは、4 行 4 列であることによって、3 次元座標系の回転移動、平行移動、線形拡大縮小を全て表現できるからである。FK System において、4 行 4 列の正方行列は fk_Matrix という型で提供されている。

fk_Matrix 型は、定義時には単位行列 (零行列ではない) が初期値として設定される。fk_Matrix は、fk_Vector のように初期値設定の手段を持たない。そのかわり、多様な設定方法が存在する。表 2.3 はそれをまとめたものである。

1) 例えば、マイクロソフト社の「Direct3D」は行列系として VM 系を採用している。

表 2.3　fk_Matrix の初期値設定用メソッド

メソッド名と引数	引数の意味	効果
MakeRot(double, fk_Axis)	角度数と軸	回転行列の生成
MakeTrans(double, double, double)	x, y, z に対応する実数値	平行移動行列の生成
MakeTrans(fk_Vector)	ベクトル	上記に同じ
MakeScale(double, double, double)	x, y, z に対応する実数値	拡大縮小行列の生成
MakeScale(fk_Vector)	ベクトル	上記に同じ
MakeEuler(double, double, double)	オイラー角に相当する実数値	オイラー回転行列の生成
MakeEuler(fk_Angle)	オイラー角	上記に同じ

MakeRot メソッドは、2 つ目の引数に x 軸を表す fk_Axis.X、y 軸を表す fk_Axis.Y、z 軸を表す fk_Axis.Z のいずれかをとる。最初の引数である実数値は弧度法 (rad ラジアン) として扱われる。$180^\circ = \pi\, rad$ である。.NET 環境では π を表す定義値として Math.PI を提供している。したがって、たとえば 90 度は `Math.PI/2.0` と表せばよい。

MakeEuler メソッドは引数にそれぞれ順に heading 角、pitch 角、bank 角を与える。単位は MakeRot メソッドと同様に弧度法である。fk_Angle 型は、オイラー角を表わすクラスで、heading 角、pitch 角、bank 角にあたるメンバはそれぞれ h, p, b となっており、public アクセスが可能である。

2.2.3　比較演算

fk_Matrix 型も、fk_Vector 型のような Equals() メソッドを持ち合わせている。これらは、やはり許容誤差を持って判定される。

2.2.4　逆行列演算

fk_Matrix 型は、GetInverse() メソッドを持っている。これは逆行列を返すものであり、自身に変化は起こさない。以下のプログラムは、行列 A の逆行列を B に代入するものである。

```
var A = new fk_Matrix();
    :
var B = A.GetInverse();
```

2.2.5　二項演算子

fk_Matrix 型の二項演算子は表 2.4 の通りのものが用意されている。

表 2.4　fk_Matrix の二項演算子

演算子	形式	機能	返り値の型
+	Matrix + Matrix	行列の和	fk_Matrix
-	Matrix - Matrix	行列の差	fk_Matrix
*	Matrix * Matrix	行列の積	fk_Matrix
*	Matrix * Vector	ベクトルと行列の積	fk_Vector

2.2.6　代入演算子

fk_Matrix 型においても、代入処理における挙動は fk_Vector 型とまったく同様である。すなわち、変数同士の代入は参照先を同一にするのみで、明示的にコピーを行いたい場合はコピーコンストラクタを利用することとなる。その他の代入演

算子も用意されており、それは表 2.5 のようなものである。記述法は、表 2.2 と同様である。

表 2.5　fk_Matrix の代入演算子

演算子	形式	効果
+=	Matrix += Matrix	left = left + right
-=	Matrix -= Matrix	left = left - right
*=	Matrix *= Matrix	left = left * right
*=	Matrix *= Vector	left = right * left

2.2.7　各成分へのアクセス

行列成分へのアクセスは、配列演算子を用いる。これは 2 次元で定義されており、1 次元目が行を、2 次元目が列を表している。行と列はカンマで区切って指定する。

```
var mat = new fk_Matrix();
mat.MakeEuler(Math.PI/2.0, Math.PI/4.0, Math.PI/6.0);
Console.WriteLine($"mat[1][0] = {mat[1,0]}");
```

2.2.8　その他のメソッド

以下に、fk_Matrix で用いられる主要なメソッドを紹介する。

void Init()
　自身を単位行列にする。

void Set(int r, int c, double a)
　行番号が r、列番号が c に対応する成分の値を a に設定する。

void SetRow(int r, fk_Vector v)
　行番号が r である行ベクトルを v の各成分値に設定する。

void SetRow(int r, fk_HVector v)
　行番号が r である同次座標付き行ベクトルを v の各成分値に設定する。

void SetCol(int c, fk_Vector v)
　列番号が c である列ベクトルを v に設定する。

void SetCol(int c, fk_HVector v)
　列番号が c である同次座標付き列ベクトルを v に設定する。

fk_HVector GetRow(int r)
　行番号が r である行ベクトルを取得する。

fk_HVector GetCol(int c)
　列番号が c である列ベクトルを取得する。

bool IsSingular(void)
　自身が非正則行列であるかどうかを判定する。非正則行列とは、逆行列が存在しない行列のことである。非正則行列である場合は true を、そうでない場合は false を返す。

bool IsRegular(void)
　自身が正則行列であるかどうかを判定する。正則行列とは、逆行列が存在する行列のことである。正則行列である場合は true を、そうでない場合は false を返す。

bool Inverse(void)

自身が正則行列であった場合、自身を逆行列と入れ替えて true を返す。非正則行列であった場合は、false を返し自身は変化しない。

void Negate(void)

自身を転置する。転置とは、行列の行と列を入れ替える操作のことである。

2.2.9 4 次元ベクトル

fk_Matrix は 4 行 4 列の正方行列であるため、本来であれば、fk_Matrix 型に対応するベクトルは 4 次元でなければならない。しかし、fk_Vector 型は 3 次元であるため、そのままでは積演算ができないことになる。そのため、FK では 4 次元ベクトル用クラスとして fk_HVector クラスがあり、実際の行列演算は fk_HVector を用いて行うという仕組みになっている。

fk_HVector クラスは fk_Vector クラスの派生クラスであり、4 次元目の成分「w」を持つ。この成分は double 型の public メンバとして定義されており、自由にアクセスすることができる。座標変換においては、w 成分は同次座標を表わすことを想定している。同次座標は通常 1 で固定されるが、「射影幾何学」と呼ばれる数学分野においては、この同次座標を操作する理論もある。

fk_Matrix 型と fk_Vector 型の積演算は、実際には以下のような処理が FK 内部で行われる。

1. まず、fk_Vector 型変数に対し fk_HVector 型に暗黙の型変換が行われる。
2. 変換後の fk_HVector オブジェクトと fk_Matrix による積を算出し、その結果が fk_HVector 型として返される。
3. 返り値である fk_HVector オブジェクトから、暗黙の型変換によって fk_Vector 型変数に代入される。

この仕組みにより、FK の利用者が通常の行列演算において fk_HVector の存在を意識する必要はない。しかし、あえて fk_HVector を利用することによる利点もある。

4 行 4 列の行列は、回転等による姿勢変換と平行移動変換を、行列同士の積演算を行うことによって同時に内包することができる。物体の位置や形状頂点などの位置ベクトルに関しては、同次座標が 1 であることによってそのまま変換が可能であるが、モデルの姿勢等の方向ベクトルに関しては同次座標が 1 のまま変換を行うと間違った結果を生じてしまう。これは、方向ベクトルの変換に関しては姿勢変換のみが適用されるべきであるのに対し、平行移動も適用してしまうからである。

このようなとき、同次座標を 0 に設定したベクトルで処理を行うとよい。それにより、行列中の平行移動用成分 (4 列目) が結果に作用しなくなり、姿勢変換のみによる結果を得ることが可能となる。このような処理を実現するためには、単純に w メンバに 0 を代入するだけで可能であるが、一応専用のメソッドも準備されている。IsPos() メソッドは同次座標を 1 に設定し、平行移動変換を有効とする。IsVec() メソッドは同次座標を 0 に設定し、平行移動変換を無効とする。

上記のような処理をプログラム中で実現したい場合は、fk_HVector クラスを利用するとよいだろう。

2.3 四元数 (クォータニオン)

四元数 (クォータニオン) は、3 種類の虚数単位 i, j, k と 4 個の実数 s, x, y, z を用いて

$$\mathbf{q} = s + xi + yj + zk$$

という形式で表現される数のことであり、発見者のハミルトンにちなんで「ハミルトン数」と呼ばれることもある。この節では、FK 中で四元数を表わすクラス「fk_Quaternion」の利用方法を述べる。ただし、四元数の数学的定義や、オイラー角、行列と比較した長所と短所に関してはここでは扱わない。適宜参考書を参照されたい。

なお、FK における四元数と行列、オイラー角に関する関係は

- MV 行列系。
- 右手座標系。

を満たすことを前提に構築されている。

2.3.1 四元数クラスのメンバ構成

四元数の 3 種類の虚数単位をそれぞれ i, j, k とし、4 個の実数によって $\mathbf{q} = s + xi + yj + zk$ で表わされるとする。このとき、実数部である s を**スカラー部**、虚数部である $xi + yj + zk$ を**ベクトル部**と呼ぶ。これは、四元数ではしばしば虚数部係数をベクトル (x, y, z) として扱うと、都合の良いことが多々あるためである。

これにならい、四元数を表わすクラス fk_Quaternion では、四元数を 1 個の double 型実数 `s` と、fk_Vector 型のメンバ `v` によって表わしている。つまり、fk_Quaternion 型の変数を `q` とした場合、

q.s,　q.v.x,　q.v.y,　q.v.z

として各成分にアクセスできることになる。これらは全て public メンバとなっている。

2.3.2 四元数の生成と設定

fk_Quaternion クラスは、デフォルトコンストラクタとして何も引数をとらないコンストラクタがある。この場合、スカラー部である `s` が 1、ベクトル部に零ベクトルが設定される。

その他にも、fk_Quaternion には 2 種類のコンストラクタがある。まず、4 個の実数を引数にとるもので、これはそれぞれスカラー部、そしてベクトル部の x, y, z 成分に対応する。もう 1 つのコンストラクタは実数 1 個と fk_Vector 1 個をとるもので、やはり同様にスカラー部とベクトル部に対応する。

それぞれの書式例は、以下のようなものになる。

```
var v = new fk_Vector(0.2, 0.5, 1.0);
var q1 = new fk_Quaternion(0.3, 0.4, 2.0, -2.0);
var q2 = new fk_Quaternion(0.5, v);
```

設定用のメソッドとしては、以下のようなものがある。

void Init(void)
: 初期化のためのメソッドで、デフォルトコンストラクタと同様にスカラー部が 1、ベクトル部に零ベクトルが設定される。

void Set(double t, const fk_Vector &v)
: 設定のためのメソッドで、t がスカラー部、v がベクトル部に対応している。

void Set(double s, double x, double y, double z)
: 設定のためのメソッドで、四元数の各成分が順に対応している。

void SetRotate(double theta, const fk_Vector &v)
: 角度を theta、回転軸を v とするような回転変換を表わす四元数を生成する。実際に四元数に代入される値は、theta を θ、v を $\mathbf{V}$ としたとき、スカラー部が $\cos\frac{\theta}{2}$、ベクトル部が $\frac{\mathbf{V}}{|\mathbf{V}|}\sin\frac{\theta}{2}$ となる。

void SetRotate(double theta, double x, double y, double z)
: 角度を theta, 回転軸を (x, y, z) とするような回転変換を表わす四元数を生成する。ベクトル部の引数が実数となる以外は、前述の SetRotate と同様である。

2.3.3 比較演算子

fk_Quaternion 型は、比較演算子として '==' と '!=' を利用できる。fk_Vector と同様に、ある程度の数値誤差を許している。

2.3.4 単項演算子

fk_Quaternion 型では、以下の表 2.6 に挙げる 3 種類の単項演算子をサポートする。

表 2.6 fk_Quaternion の単項演算子

演算子	効果
-	符合転換を返す。
\mbox{}	共役を返す。
!	逆元を返す。

元となる四元数を $\mathbf{q} = s + xi + yj + zk$ とすると、符合転換 $-\mathbf{q}$、共役 $\overline{\mathbf{q}}$、逆元 $\mathbf{q}^{-1}$ はそれぞれ以下の式によって求められる。

$$-\mathbf{q} = -s - xi - yj - zk,$$
$$\overline{\mathbf{q}} = s - xi - yj - zk,$$
$$\mathbf{q}^{-1} = \frac{\overline{\mathbf{q}}}{|\mathbf{q}|^2}$$

なお、全ての単項演算子は自身には変化を及ぼさない。

2.3.5 二項演算子

fk_Quaternion 型がサポートする二項演算子を、表 2.7 に羅列する。

表 2.7 fk_Quaternion の二項演算子

演算子	形式	機能	返り値の型
+	Quaternion + Quaternion	四元数の和	fk_Quaternion
-	Quaternion - Quaternion	四元数の差	fk_Quaternion
*	Quaternion * Quaternion	四元数の積	fk_Quaternion
*	double * Quaternion	四元数の実数倍	fk_Quaternion
*	Quaternion * double	四元数の実数倍	fk_Quaternion
/	Quaternion / double	四元数の実数商	fk_Quaternion
^	Quaternion ^ Quaternion	四元数の内積	double
*	Quaternion * Vector	四元数によるベクトル変換	fk_Vector

この中で、内積値は $\mathbf{q}_1 = s_1 + x_1 i + y_1 j + z_1 k$, $\mathbf{q}_2 = s_2 + x_2 i + y_2 j + z_2 k$ としたとき、

$$\mathbf{q}_1 \cdot \mathbf{q}_2 = s_1 s_2 + x_1 x_2 + y_1 y_2 + z_1 z_2$$

という式によって求められる実数値のことである。また、ベクトル変換というのは四元数 $\mathbf{q}$ と 3 次元ベクトル $\mathbf{V}$ に対し、

$$\mathbf{V}' = \mathbf{q}\mathbf{V}\mathbf{q}^{-1}$$

という演算を施すことである。このとき、$\mathbf{q}$ が回転変換を表わす場合に、$\mathbf{V}'$ は $\mathbf{V}$ を回転したベクトルとなる。この演算は、行列による回転演算と比較して若干高速であることが知られている。

2.3.6 代入演算子

fk_Quaternion 型の代入演算子 ‘ = ’ は、fk_Vector の場合と同様に値のコピーが行われ、実体は別物となる。その他の代入演算子を表 2.8 に列挙する。

表 2.8 fk_Quaternion の代入演算子

演算子	形式	効果
+=	Quaternion += Quaternion	left = left + right
-=	Quaternion -= Quaternion	left = left - right
*=	Quaternion *= double	left = left * right
/=	Quaternion /= double	left = left / right
*=	Quaternion *= Quaternion	left = left * right

2.3.7 オイラー角との相互変換

四元数は任意軸による回転変換を表わすが、これはすなわちオイラー角と同義の情報を持つことを意味する。fk_Quaternion には、fk_Angle と相互変換を行うためのプロパティとして「Euler」が用意されており、fk_Angle 型での参照と代入が可能である。

2.3.8 各種メソッド・プロパティ

fk_Quaternion には、これまで挙げた他にも以下のようなメソッドがある。なお、文中では四元数自身を $\mathbf{q} = s + xi + yj + zk$ と想定する。

double Norm

ノルム $|\mathbf{q}|^2 = s^2 + x^2 + y^2 + z^2$ を返すプロパティ。

double Abs

絶対値 $|\mathbf{q}| = \sqrt{s^2 + x^2 + y^2 + z^2}$ を返すプロパティ。

bool Normalize(void)

自身の正規化四元数 $\frac{\mathbf{q}}{|\mathbf{q}|}$ を求め、自身に上書きし true を返すメソッド。ただし、成分が全て 0 であった場合は値を変更せずに false を返す。

bool Inverse(void)

自身の逆元 $\mathbf{q}^{-1}$ を求め、自身に上書きし true を返すメソッド。ただし、成分が全て 0 であった場合は値を変更せずに false を返す。

void Conj(void)

自身の共役 $\bar{\mathbf{q}}$ を求め、自身に上書きするメソッド。

fk_Matrix Conv

自身が表わす回転変換と同義の行列を返すプロパティ。

2.3.9 補間メソッド

四元数の最大の特徴は姿勢の補間である。あるオイラー角から別のオイラー角への変化をスムーズに実現することは、オイラー角や行列のみを用いる場合は難解であるが、四元数はこういった問題を解決するのに適している。

補間四元数を求めるために、FK では以下の 2 種類のメソッドが用意されている。

fk_Quaternion fk_Math.QuatInterLinear(fk_Quaternion q1, fk_Quaternion q2, double t)

q1 と q2 に対し、単純線形補間を行った結果を返す。t は 0 から 1 までのパラメータで、0 の場合 q1、1 の場合 q2 と完全に一致する。補間処理は以下の式に基づく。

$$\mathbf{q}(t) = (1-t)\mathbf{q}_1 + t\mathbf{q}_2 \tag{2.1}$$

fk_Quaternion fk_Math.QuatInterSphere(fk_Quaternion q1, fk_Quaternion q2, double t)

q1 と q2 に対し、球面線形補間を行った結果を返す。t は 0 から 1 までのパラメータで、0 の場合 q1、1 の場合 q2 と完全に一致する。補間処理は以下の式に基づく。

$$\mathbf{q}(t) = \frac{\sin((1-t)\theta)}{\sin\theta}\mathbf{q}_1 + \frac{\sin(t\theta)}{\sin\theta}\mathbf{q}_2 \qquad (\theta = \cos^{-1}(\mathbf{q}_1 \cdot \mathbf{q}_2)) \tag{2.2}$$

演算そのものは単純線形補間の方が高速である。しかし単純線形補間では、状況によっては不安定な結果を示す場合がある。このような現象が許容できない場合には、球面線形補間による処理が有効である。

2.4 乱数

乱数の取得は様々なプログラムで必要となるものであるが、FK では以下のような簡便な一様乱数取得用関数を提供している。

unsigned int fk_Math.Rand()

0 以上の一様乱数整数値を取得する。

int fk_Math.Rand(int m, int M)

$m \leqq r < M$ を満たす一様乱数整数値 r を取得する。

double fk_Math.DRand()

0 以上 1 未満の一様乱数実数を取得する。

double fk_Math.DRand(double m, double M)

$m \leqq r < M$ を満たす一様乱数実数値 r を取得する。

第 3 章　色とマテリアル

この章では fk_Material と呼ばれる立体の色属性を司るクラスの使用法を述べる。この章に書かれていることは、のちに立体の色属性を設定するために必要なものとなる。

3.1　色の基本 (fk_Color)

まず、色の構成に関しての記述から始めよう。光による色の 3 元色は、赤、緑、青である。これらの色の組合せによって、ディスプレイで映し出されるあらゆる色の表現が可能である。

これらの色の組合せを表現するのが fk_Color クラスの役目である。fk_Color クラスは次のように使用する。

```
var col = new fk_Color();
col.Init(0.5, 0.6, 0.7);
```

3 つの引数はそれぞれ Red, Green, Blue の値を表し、0 から 1 の値を代入することができる。つまり、すべてに 1 を代入したときに白色、すべてに黒を代入したときに黒色を表現することになる。これは、次のように初期設定によっての代入も可能である。

```
var col = new fk_Color(0.5, 0.6, 0.7);
```

また、次のように個別に代入することも可能である。コンストラクタやメソッドの場合は float 型と double 型の両方に対応しているが、個別に代入する場合は float 型の値でなければならない。

```
var col = new fk_Color();
col.r = 0.5f;
col.g = 0.6f;
col.b = 0.7f;
```

3.2　物質のリアルな表現 (fk_Material)

ここでは、マテリアルと呼ばれる物質色の表現法と、その設定法を記述する。前節で述べたような表現では、まだ物質を表現するには不十分なのである。たとえば蛍光色のような表現、光沢、透明度といった、非常に細かな設定がなされて初めて物質感を出すことができる。ここでは、それらの設定法を述べる。しかし、実際に自分の思い通りにマテリアルの設定が行えるようになるには、ある程度の試行錯誤が必要となるだろう。

fk_Material クラスは表 3.1 のようなステータスを持っている。

表 3.1　fk_Material の持つステータス

ステータス名	値の型	意味
Alpha	float	透明度
Ambient	fk_Color	環境反射係数
Diffuse	fk_Color	拡散反射係数
Emission	fk_Color	放射光係数
Specular	fk_Color	鏡面反射係数
Shininess	float	鏡面反射のハイライト

それぞれに対しての簡単な説明を付随する。

透明度は、文字通り物質の透明度を指し、0.0 のとき完全な透明、1.0 のときに完全な不透明を指す。注意しなければならないのは、例えばガラスのような透明な物質感を表現したいときは、他のステータスを黒に近い色に設定しないと、曇りガラスのような表現になってしまうことである。従って、立体が持つ透明感はこの値だけではなく、他の色属性も考慮に入れる必要がある。なお、立体のシーンへの登録の順序によって透過処理の有無が変わってしまうので、透過処理を行いたいモデルはできるだけ後に登録する必要がある。これに関する詳細は第 10 章で再び述べる。また、透過処理を行う場合は描画そのものが非常に低速になるため、シーンにおいて透過処理を実際に行うための設定を行う必要もある。これに関しても第 10 章で述べる。

環境反射係数は、環境光に対しての反射の度合を示すものである。環境光は、どのような状態にある面にも同様に照らされる (と仮定された) 光である。したがってこの値が高いと、光の当たってない面も光が当たっている面と同様な色合いを写し出すので、蛍光色に似たような雰囲気になる。逆に、この値が低いと露骨に光源の効果が出る。したがって、暗い部屋の中に光源があるような雰囲気が出る。

拡散反射係数は、普通一般にものの「色」と呼ばれているものを指す。具体的には、光源に当たることによって反映される色のことである。この色は、光源に対して垂直な角度になったときに最も明るく反映されるが、一旦面に照射されればすべての方向に均等に散乱するため、どの視点から見ても同じ明るさを示す。この値が高いと、物質の色が素直に現れる。この値が低く、Ambient や Diffuse や Emission の値も低い場合は、その物体は墨のように黒いものとなる。Diffuse の値が低く、その他の値のうちの幾つかが高い値を持つとき、変化に富んだ物質感が醸し出される。

放射光係数は、文字通り自身が放射する光の係数を示す。つまり、あたかも自身が発光しているかのような効果を出す。しかし、この物体自身は光源ではないので、他の物体の色に影響することはない。この値の働きは、あくまで自身が発光しているような効果を出すことだけである。光源の設定に関しては、8 章と 10 章で詳しく述べている。

鏡面反射係数は、文字通り反射の色合いを示すものである。この値は、ある特定の角度範囲からしか反映されない反射の強さを示すものである。この値が高いと、鏡のように反射が強くなる[1)]。この値が高いと、金属やプラスチックのように表面が滑らかな印象を受ける。逆に低い場合には、紙や石炭のように表面が粗い印象を受ける。

鏡面反射のハイライトは、鏡面反射の反射角度範囲を設定するものである。この値は、0 から 128 までの値をとり、この値が大きいほどハイライトは小さくなり、より金属の質感が増す。逆に値を小さくした場合、質感はプラスチックのようになる。

それぞれの設定の仕方は次のようになっている。

```
var mat = new fk_Material();
var amb = new fk_Color(0.3, 0.5, 0.8);
var dif = new fk_Color(0.2, 0.4, 0.9);
var emi = new fk_Color(0.0, 0.5, 0.3);
var spe = new fk_Color(1.0, 0.5, 1.0);
```

1)　この値を高くしても、実際の鏡のような効果 (他のオブジェクトが反射して映される) があるわけではない。

```
    mat.Alpha = 0.5f; // 透明度の設定
    mat.Ambient = amb; // 環境反射係数の設定
    mat.Diffuse = dif; // 拡散反射係数の設定
    mat.Emission = emi; // 放射光係数の設定
    mat.Specular = spe; // 鏡面反射係数の設定
    mat.Shininess = 64.0f; // 鏡面反射のハイライトの設定
```

また、fk_Color を引数にとる関数は次のように直接代入することもできる。

```
    var mat = new fk_Material();

    mat.Alpha = 0.5f;
    mat.Ambient = new fk_Color(0.3, 0.5, 0.8); // 環境反射係数の設定
    mat.Diffuse = new fk_Color(0.2, 0.4, 0.9); // 拡散反射係数の設定
    mat.Emission = new fk_Color(0.0, 0.5, 0.3); // 放射光係数の設定
    mat.Specular = new fk_Color(1.0, 0.5, 1.0); // 鏡面反射係数の設定
    mat.Shininess = 64.0f; // 鏡面反射のハイライトの設定
```

この作業は、ディティールを凝る分には非常にいいのであるが、ときには色つけは簡単に済ませたいという場面もあるだろう。そのようなとき、逐一値を設定するのは不便である。このとき、非常に簡易に済ませることのできる手段が 2 種類用意されている。

最初の手段は、Ambient と Diffuse プロパティに同じ値を設定することである。テストなど、あまり色の質感が関係ない状況ならば、通常はこれだけでも十分である。次のように記述することで、1 つの fk_Color の値を 2 つのプロパティに対して同時に設定できる。

```
    var mat1 = new fk_Material();
    var mat2 = new fk_Material();
    // Ambient と Diffuse どちらが先でも結果は一緒である。
    mat1.Ambient = mat1.Diffuse = new fk_Color(1.0, 1.0, 0.0); // 黄色いマテリアル
    mat2.Diffuse = mat2.Ambient = new fk_Color(1.0, 0.0, 1.0); // マゼンタのマテリアル
```

もうひとつの手段は、あらかじめ準備されているマテリアルを使用してしまうことである。全部で 40 種類あるこれらのマテリアル群は、どれもグローバルな変数として利用できる。大抵の場合、これで事が足りるだろう。なお、このマテリアルを羅列した表を付録 A に掲載しておく。参照して、適宜使用してほしい。

第 4 章　形状表現

この章では fk_Shape と言われる形状を司るクラスと、それから派生したクラスの使用法を述べる。これらのクラスは形状をなんらかの形で定義する手段を提供している。しかし、これらの形状はのちの fk_Model に代入が行われない限り描画されない。つまり、FK システムでは形状とオブジェクトの存在は別々に定義されている必要がある。たとえば、車を 3 台表示したければ、まず車の形状を定義し、次にモデルを 3 つ作成し、それらに車の形状を代入すればよい。このようなケースで、モデル 1 つずつに対して形状を改めて作成するのは非効率といえる。こういったことは、8 章で詳しく述べる。

FK システムにおいて、形状は表 4.1 のようなものを定義することができる。

表 4.1　形状の種類

形状	クラス名	必要な引数
点	fk_Point	位置ベクトル
線分	fk_Line	両端点の位置ベクトル
ポリライン	fk_Polyline	各頂点の位置ベクトル
閉じたポリライン	fk_Closedline	各頂点の位置ベクトル
多角形平面	fk_Polygon	各頂点の位置ベクトル
円	fk_Circle	分割数、半径
直方体	fk_Block	縦、横、高さ
球	fk_Sphere	分割数、半径
正多角柱・円柱	fk_Prism	上面半径、底面半径、高さ
正多角錐・円錐	fk_Cone	底面半径、高さ
インデックスフェースセット	fk_IndexFaceSet	ファイル名等
矩形テクスチャ	fk_RectTexture	画像ファイル名
三角形テクスチャ	fk_TriTexture	画像ファイル名
メッシュテクスチャ	fk_MeshTexture	画像ファイル名
IFS テクスチャ	fk_IFSTexture	画像ファイル名
文字列テクスチャ	fk_TextImage	文字列またはテキストファイル
パーティクル	fk_ParticleSet	様々な設定
光源	fk_Light	タイプ

次節から、これらの詳細な使用法をひとつずつ述べ、最後にそれらを統括的に扱う方法を述べる。

4.1　ポリライン (fk_Polyline)

ポリラインとは、いわば折れ線のことである。線分が複数つながったものと考えてもよい。構成される線の数は任意でよい。

定義方法は、普通に変数を準備すればよい。

```
var poly = new fk_Polyline();
```

PushVertex() メソッドを使えば 1 個ずつ頂点を代入していくことができる。

```
var pos = new fk_Vector();
var poly = new fk_Polyline();

for(int i = 0; i < 10; i++)
{
    pos.Set((double)(i*i), (double)i, 0.0);
    poly.PushVertex(pos);
}
```

このように、順番に位置ベクトルを代入していけばよい。この場合は、9 本の線分によって構成されたポリラインが生成される。もし途中で頂点位置を変更したい場合は、SetVertex メソッドを用いるとよい。

```
var pos = new fk_Vector();
var poly = new fk_Polyline();

for(int i = 0; i < 10; i++)
{
    pos.Set((double)(i*i), (double)i, 0.0);
    poly.PushVertex(pos);
}
pos.Set(5.0, 5.0, 5.0);
poly.SetVertex(5, pos);
```

上記のプログラムソースの最後の行で、ポリラインの 6 番目の頂点の位置を変えている。すべてを生成し直すよりも、この方が高速かつ手軽に処理できる。

なお、設定した全ての頂点情報を削除したい場合は、次のように AllClear() メソッドを用いれば実現できる。

```
var poly = new fk_Polyline();
    :
    :
poly.AllClear();
```

4.2 閉じたポリライン (fk_Closedline)

fk_Closedline クラスは、基本的には使用法は fk_Polyline クラスと変わりはない。唯一異なる点は、fk_Closedline は閉じたポリライン — つまり、始点と終点の間にも線分が存在することを意味する。したがって、多角形を線分で表現したい場合に適している。

4.3 点 (fk_Point)

「点」というのは、ここでは画面上に表示させる 1 ピクセル分の存在を指す。例えば、これらの集合を流動的に動かすことによって、空間中での流れを表現することができる[1)]。点は、それ自体が大きさを持たないことや、描画することが高速なことから、とても扱いやすい対象である。

1) 実際にこのような機能を実装する場合は、4.13 節にあるパーティクル用クラスの採用も検討するとよい。

fk_Point クラスの利用法は、fk_Polyline クラスとまったく同一である。つまり、PushVertex メソッドで点を生成し、SetVertex メソッドによって移動させることができる。fk_Polyline と異なる点は、ポリラインが表示されるか、複数の点が表示されるかということのみである。

なお、設定された頂点の全削除は fk_Polyline と同様に AllClear() を用いれば実現可能である。

4.4 線分 (fk_Line)

「線分」は、画面上に線分を表示させる。fk_Line は、もちろん 1 本の線分を表現することが可能だが、複数の線分を 1 つのオブジェクトで表現できる。

定義には特に特別な引数は必要としない。

```
var line = new fk_Line();
```

ただし、この場合には両端点の位置がともに原点になってしまうので、位置ベクトルをなんらかの形で代入する必要がある。1 つの手段として、両端点の位置ベクトルが並んだ fk_Vector 型の配列を用意しておき、それを SetVertex() メソッドを用いて代入することである。

```
var vec = new fk_Vector[2];
var line = new fk_Line();

vec[0] = new fk_Vector(1.0, 1.0, 1.0);
vec[1] = new fk_Vector(-1.0, 1.0, 1.0);

line.SetVertex(vec);
```

このような記述で、ln は $(1,1,1)$ と $(-1,1,1)$ を結ぶ線分を表現することになる。値の代入をしなおすことも可能である。それにはやはり SetVertex メソッドを使用すればよい。

```
var line = new fk_Line();
var a = new fk_Vector(1.0, 1.0, 1.0);
var b = new fk_Vector(-1.0, 1.0, 1.0);

line.SetVertex(0, a);
line.SetVertex(1, b);
```

この場合での SetVertex メソッドの最初の引数は、0 なら始点を、1 なら終点を代入することを意味する。2 つめの引数には fk_Vector 型の変数を代入すればよい。

また、fk_Line クラスのオブジェクトは複数の線分を持つことが可能である。新たに線分を追加したい場合は、PushLine() メソッドを使用する。次のようにすればよい。

```
var line = new fk_Line();
var vec1 = new fk_Vector(0.0, 1.0, 0.0);
var vec2 = new fk_Vector(1.0, 0.0, 0.0);

line.PushLine(vec1, vec2); // 2つの fk_Vector を使う方法
```

```
var vecArray = new fk_Vector[2];
vecArray[0] = new fk_Vector(0.0, 0.0, 1.0);
vecArray[1] = new fk_Vector(0.0, 0.0, -1.0);
line.PushLine(vecArray); // fk_Vector の配列を使う方法
```

これにより、fk_Line 中の線分が次々と追加されていく。

fk_Line における線分情報の全削除は、fk_Polyline と同様に AllClear() によって実現可能である。

4.5 多角形平面 (fk_Polygon)

この fk_Polygon クラスは、fk_Polyline クラスや fk_Closedline クラスと使用法は同様である。ただし、このクラスで定義されたオブジェクトは、平面として存在する。つまり、厚さのない 1 枚の板として存在することになる。

fk_Polyline や fk_Closedline と唯一異なる点は、fk_Polygon は面の向き、すなわち法線ベクトルを保持するという点である。これは、FK システムの中で自動的に計算が行われる。与えられている頂点が同一平面上にない場合、近似的な法線ベクトルが与えられる。

なお、頂点情報の全削除は fk_Polyline と同様に AllClear() を用いればよい。

4.6 円 (fk_Circle)

fk_Circle クラスは、ステータスとして半径と分割数を持つ。fk_Circle クラスのオブジェクトは、実際には多角形の集合によって構成されている。具体的に述べると、中心から放射状に伸びた三角形によって構成される。したがって、円は実際には正多角形の形をしていることになる。ここで問題になるのは、いくつの三角形によって円を疑似するかということである。当然、円により近くするには多くの三角形に分割した方がよい。しかし、多くの三角形が存在するということは、処理そのものも時間がかかるということである。特に多くのオブジェクトを操作するときや、あまりパフォーマンスのよくないマシンで扱う場合にはこの問題は切実となる。そこで、fk_Circle には分割数を指定するメソッドを持っている。ある条件によって、分割数を変更することができるのである。

実際の使用法を述べる。まず、定義はやはり通常どおり行えばよい。

```
var circ = new fk_Circle();
```

fk_Circle クラスでは、初期値として分割数と半径を指定することができる。

```
var circ = new fk_Circle(4, 100.0);
```

これによって、分割数 4、半径 100 の円が生成される[2)]。なお。この円は半径を r とすると $(r\cos\theta, r\sin\theta, 0)$ 上に境界線が存在し、面の法線ベクトルは必ず $(0, 0, -1)$ となっている。

また、SetRadius メソッドで半径を動的に制御することが可能である。

```
var circ = new fk_Circle(4, 5.0);
```

2) ここでいう分割数とは、円の $\frac{1}{4}$ を三角形に分割する数を指定するものである。したがって、分割数が 4 ならばその円は 16 個の三角形によって構成されることになる。

```
circ.SetRadius(10.0);
```

次の例は，半径を実数倍するものである。

```
var circ = new fk_Circle(4, 10.0);
double scale = 4.0;
circ.SetScale(scale);
```

他に、動的に分割数を変更する方法として SetDivide メソッドがある。引数として分割数を与えることができる。

```
var circ = new fk_Circle(4, 10.0);
circ.SetDivide(10);
```

4.7 直方体 (fk_Block)

直方体は、x, y, z 軸にそれぞれ垂直な 6 つの面で構成された立体である。この立体は横幅、高さ、奥行きのステータスを持ち、それぞれ x 方向、y 方向、z 方向の大きさと対応している。

定義は、通常通り行えばよい。

```
var block = new fk_Block();
```

このとき、初期値としてすべての辺の長さが 1 である立方体が与えられる。各辺長を初期値として設定することも可能である。

```
var block = new fk_Block(10.0, 1.0, 40.0);
```

SetSize メソッドは、大きさを動的に制御できる。

```
var block = new fk_Block();
block.SetSize(10.0, 40.0, 50.0);
```

SetSize は多重定義されており、次のようにひとつの要素だけを制御することもできる。

```
var block = new fk_Block();
block.SetSize(10.0, fk_Axis.X);
block.SetSize(40.0, fk_Axis.Y);
block.SetSize(50.0, fk_Axis.Z);
```

ここで注意しなければならないのは、この直方体の中心が原点の設定されていることである。つまり、直方体の 8 つの頂点の位置ベクトルは、

$$(x/2, y/2, z/2), \qquad (-x/2, y/2, z/2),$$
$$(x/2, -y/2, z/2), \qquad (-x/2, -y/2, z/2),$$
$$(x/2, y/2, -z/2), \qquad (-x/2, y/2, -z/2),$$
$$(x/2, -y/2, -z/2), \quad (-x/2, -y/2, -z/2).$$

ということになる。たとえば、yz 平面を地面にみたてて直方体を配置する場合、直方体を代入されたモデルの移動量は x 方向に $x/2$ 移動すればよい。

SetScale メソッドは、直方体を現状から実数倍するものである。このメソッドも 3 種類の多重定義がされている。大きさそのものを単純に実数倍する場合、次のように実数ひとつを引数に代入すればよい。

```
var block = new fk_Block(10.0, 10.0, 10.0);
block.SetScale(2.0);
```

また、ある軸方向だけ実数倍したい場合は、2 つ目の引数に軸要素を代入すればよい。

```
var block = new fk_Block(10.0, 10.0, 10.0);
block.SetScale(2.0, fk_Axis.X);
block.SetScale(3.0, fk_Axis.Y);
block.SetScale(4.0, fk_Axis.Z);
```

x, y, z 軸の倍率を 1 度に指定することも可能である。次のプログラムは、上記のプログラムと同じ挙動をする。

```
var block = new fk_Block(10.0, 10.0, 10.0);
block.SetScale(2.0, 3.0, 4.0);
```

4.8 球 (fk_Sphere)

球は、円の場合と同様に半径と分割数を要素の持つ。基本的には、円と使用法はほとんど変わりがない。例えば、分割数 4、半径 10 の球を生成するには、以下のようにして fk_Sphere 型の変数を宣言すればよい。

```
var sphere = new fk_Sphere(4, 10.0);
```

初期設定や SetRadius()、SetDivide()、SetScale() といったメソッドの利用方法は全て fk_Circle クラスと同様なので、そちらのマニュアルを参照してほしい。

円と異なる点をあげていくと、分割数によって生成される三角形個数が異なる[3)]ことと、(当然ながら) 法線ベクトルの値が一定ではないことなどがあげられる。

3) 分割数を d とおくと、円では $4d$ 個であったが球では $8d(d-1)$ 個である。このことからもわかるように、球は大変多くの多角形から成り立つので扱いには注意が必要である。

4.9 正多角柱・円柱 (fk_Prism)

正多角柱、円柱を生成するには、fk_Prism クラスを用いることによって容易に生成できる。fk_Prism では、初期値として角数、上面半径、底面半径、高さをそれぞれ指定する。生成時は上面が $-z$ 方向を、底面が $+z$ 方向を向くように生成される。

```
var prism = new fk_Prism(5, 20.0, 30.0, 40.0);
```

ここで、上面と底面の「半径」とは、面を構成する多角形の外接円半径 (下図の r) のことを指す。

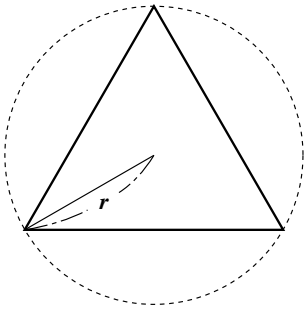

図 4.1 正多角形と外接円半径

円柱を生成するには、多角形の角数をある程度大きくすればよい。大体正 20 角形くらいでかなり円柱らしくなる。あとは、リアリティとパフォーマンスによって各自で調整してほしい。

次に述べるメソッドで、fk_Prism クラスの形状をいつでも動的に変形できる。

SetTopRadius(double r)
: 上面半径を r に変更する。

SetBottomRadius(double r)
: 底面半径を r に変更する。

SetHeight(double h)
: 高さを h に変更する。

4.10 正多角錐・円錐 (fk_Cone)

fk_Cone は正多角錐や円錐を生成するためのクラスである。このクラスでは、初期値として角数、底面半径、高さを指定する。なお、このクラスも fk_Prism と同様に底面は $+z$ 方向を向く。

```
var cone = new fk_Cone(5, 20.0, 40.0);
```

「半径」に関しては前節の fk_Prism 中の解説を参照してほしい。円錐を生成するには、やはり初期値の角数を大きくすればよいが、あまり大きな値を指定すると表示速度が遅くなるので注意が必要である。

なお、以下のメソッドによって形状をいつでも動的に変形することが可能である。

SetRadius(double r)

底面半径を r に変更する。

SetHeight(double h)

高さを h に変更する。

4.11 インデックスフェースセット (fk_IndexFaceSet)

インデックスフェースセットは、これまでに述べたような球や角錐のような典型的な形状ではない、一般的な形状を表現したいときに用いる。利用方法として、別のモデリングソフトウェアによって出力したファイルを取り込む方法と、形状情報をプログラム中で生成して与える方法がある。ここでは主にファイル入力による形状生成について解説する。プログラムによる形状生成方法は 5 章にまとめて解説してあるので、そちらを参照してほしい。

4.11.1 STL ファイルの取り込み

STL は、様々な CAD や 3 次元関連のソフトウェアで多く使われているフォーマットである。FK では、STL ファイルを読み込む機能も提供されている。STL ファイルを読み込むには、次に示すように fk_IndexFaceSet で ReadSTLFile メソッドを使用すればよい。ファイル読み込みに成功した場合 true を、失敗した場合 false を返す。もしファイル読み込みに失敗した場合、エラーメッセージが出力される。

```
var ifs = new fk_IndexFaceSet();
if(ifs.ReadSTLFile("sample.stl") == false)
{
    Console.WriteLine("File Read Error");
}
```

4.11.2 SMF ファイルの取り込み

SMF は、主に CG 関連で普及したフォーマットであり、プレーンなテキストファイルや簡単なデータ構造を特徴とするため、実際にエディタで記述するのが容易であるという利点も持っている。FK では、VRML や STL と同様に SMF を読み込む機能を持つ。使用法は次の通りである。ファイル読み込みに成功した場合 true を、失敗した場合 false を返す。もしファイル読み込みに失敗した場合、エラーメッセージが出力される。

```
var ifs = new fk_IndexFaceSet();
if(ifs.ReadSMFFile("sample.smf") == false)
{
    Console.WriteLine("File Read Error");
}
```

4.11.3 DXF ファイルの取り込み

DXF は、Autodesk 社が提唱している形状データ変換用フォーマットで、ほどんどの 3D モデリングソフトで入出力機能が用意されている。このフォーマットのファイルを読み込むには、次のように記述する。ファイル読み込みに成功した場合 true を、失敗した場合 false を返す。もしファイル読み込みに失敗した場合、エラーメッセージが出力される。

```
var ifs = new fk_IndexFaceSet();
if(ifs.ReadDXFFile("sample.dxf") == false)
{
    Console.WriteLine("File Read Error");
```

```
}
```

4.11.4 MQO ファイルの取り込み

MQO は、Metasequoia というフリーのモデラーの標準ファイルである。このフォーマットのファイルを読み込むには、ReadMQOFile() というメソッドを利用する。このメソッドは多重定義されており、二種類の引数構成がある。構成は以下の通りである。

ReadMQOFile(String fileName, String objName, bool solidFlg, bool contFlg, bool matFlg)

「fileName」はファイル名文字列、「objName」はファイル中のオブジェクト名文字列を指定する。「solidFlg」は、false の場合全てのポリゴンを独立ポリゴンとして読み込む。デフォルト引数では「true」となっている。「contFlg」は、テクスチャ断絶操作の有無を指定するためのもので、ここに関しては 6.1.3 節で詳しく説明する。デフォルトでは「true」となっている。最後の「matFlg」は、MQO ファイルからマテリアル情報を読み込むかどうかを設定するもので、デフォルトでは「false」になっている。ファイル読み込みに成功した場合 true を、失敗した場合 false を返す。

ReadMQOFile(String fileName, String objName, int matID, bool solidFlg, bool contFlg, bool matFlg)

「matID」は、特定のマテリアル ID 部分だけを抽出したい場合に、その ID を入力する。その他の引数は前の項目と同様である。

以下のプログラムは、ファイル名「sample.mqo」、オブジェクト名「obj1」という指定でデータを読み込む例である。

```
var ifs = new fk_IndexFaceSet();
if(ifs.ReadMQOFile("sample.mqo", "obj1") == false)
{
    Console.WriteLine("File Read Error");
}
```

また、MQO ファイル内で特定のマテリアル番号が指定されている面のみ入力したい場合には、以下のように記述すればよい。

```
var ifs = new fk_IndexFaceSet();
if(ifs.ReadMQOFile("sample.mqo", "obj1", 1) == false)
{
    Console.WriteLine("File Read Error");
}
```

3 番目の引数を「-1」にしたとき、3 番目の引数がない場合と同様に全ての面を入力する。

なお、Metasequoia 中でテクスチャを設定し、テクスチャも読み込みたい場合は、fk_IndexFaceSet ではなく 6.1.3 節の fk_IFSTexture を用いる必要がある。

4.11.5 MQO データの取り込み

MQO ファイルデータは、4.11.4 節ではファイルからの読み込み方法を述べたが、このファイル中のデータを全て展開した配列データからも読み込むことが可能である。メソッドは ReadMQOData() というもので、ファイル名を示す文字列のかわりに Byte 型配列の先頭アドレスを示すポインタを渡す以外は、ReadMQOFile() メソッドと同じである。

```
var ifs = new fk_IndexFaceSet();
var buffer = new Byte[1024];
```

```
if(ifs.ReadMQOFile(buffer, "obj1", 1) == false)
{
    Console.WriteLine("File Read Error");
}
```

4.11.6 形状情報の取得と、頂点座標の移動

fk_IndexFaceSet クラスは、他の形状を表すクラスと違ってファイルから情報を読み取ることも多いので、入力後に頂点や面の情報を取得する場面が考えられる。そこで、fk_IndexFaceSet クラスでは以下に示すようなプロパティやメソッドが用意されている。

int PosSize
: 形状の頂点数を取得する。

int FaceSize
: 形状の面数を取得する。

fk_Vector GetPosVec(int vID)
: インデックスが vID である頂点の位置ベクトルを返す。頂点のインデックスは 0 から順番に始まるもので、頂点数を vNum とすると 0 から (vNum-1) までの頂点が存在することになる。もし vID に対応する頂点が存在しなかった場合、零ベクトルが返される。

int [] GetFaceData(int fID)
: インデックスが fID である面の頂点インデックス情報を返す。面のインデックスは 0 から順番に始まるもので、面数を fNum とすると 0 から (fNum-1) までの面が存在することになる。返値となる配列に、参照した面を構成する頂点のインデックスが入力されて返される。もし fID に対応する面が存在しない場合、長さが 0 の配列が返される。

また、以下のようなメソッドによって各頂点を移動することも可能である。

bool MoveVPosition(int vID, fk_Vector pos)
: インデックスが vID である頂点の位置ベクトルを、pos に変更し、true を返す。もし vID に対応する頂点が存在しなかった場合、false を返す。

bool MoveVPosition(int vID, double x, double y, double z)
: インデックスが vID である頂点の位置ベクトルを、(x, y, z) に変更し、true を返す。もし vID に対応する頂点が存在しなかった場合、false を返す。

4.11.7 形状データの各種ファイルへの出力

fk_IndexFaceSet クラスでは、形状データをファイルに出力することが可能である。現在サポートされている形式は VRML、STL、DXF、MQO の 4 種類である。それぞれの出力メソッドの仕様は以下の通りである。

bool WriteVRMLFile(String fileName)
: 形状データを VRML 2.0 形式で出力する。「fileName」は出力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool WriteSTLFile(String fileName)
: 形状データを STL 形式で出力する。「fileName」は出力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool WriteDXFFile(String fileName)

形状データを DXF 形式で出力する。「fileName」は出力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool WriteMQOFile(String fileName)

形状データを MQO 形式で出力する。「fileName」は出力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。なお、オブジェクト名は「obj1」が自動的に付与される。

4.12 光源 (fk_Light)

この光源クラスのみ、他の fk_Shape の派生クラスとは性質が異なる。その他のクラスがなんらかの形状を表現するのに用いられるのに対し、このクラスは空間中の光源を設定するのに利用される。光源には、平行光源、点光源、スポットライトの 3 種類がある。これらの方向やその他のステータスは、基本的には fk_Model に代入を行ってから操作するものであり、fk_Light クラスのオブジェクトとして定義されるときは光源の種類を設定するのみである。

平行光源とは、空間中のあらゆる場所に同一方向から照らされる光の光源をいう。地球における太陽光のようなものと考えればよい。もっとも扱いやすいので、光を利用した特別な効果を用いないのであればこれで十分である。平行光源は属性として方向のみを持つ。

点光源は、空間中のある 1 点から光を放射する光源である。宇宙空間での恒星や、部屋の中での灯りなどは点光源を利用するとよい。点光源は、属性として位置と減衰係数を持つ。減衰係数とは、光源からの距離と照射される明るさとをどのような関係にするかを定義するもので、これはさらに一定減衰係数、線形減衰係数、2 次減衰係数の 3 つの係数がある。通常はデフォルトのままでよいだろう。詳細はリファレンスマニュアルを参照してほしい。

スポットライトは点光源の特殊な場合で、ある 1 定方向を特別に強く照射する働きを持ち、文字通りスポットライトとしての機能を持つ。そのため、スポットライトは属性として位置と方向の両方を持つが、さらに 3 つの属性も持っている。第 1 の属性は点光源と同じく減衰係数である。第 2 の属性はカットオフ係数で、これはスポットライトの照射角度のことである。この値が大きければ、スポットライトによって照らされる領域が広くなる。第 3 の属性は「スポットライト指数」と呼ばれるもので、この値が大きいと照射点の中心に近いほど明るくなる効果が強くなる。この値を 0 にすると、スポットライトの中心であろうが外側付近であろうが明るさは変わらない。このスポットライト指数も扱いが難しいパラメータなので、減衰係数と同じく通常は 0 でよい。

光源の作成は、まず fk_Light 型の変数を作成する。

```
var light = new fk_Light();
```

光源の種類を設定するには、Type プロパティに対して種類の設定を行う。

```
var parallel = new fk_Light();
var point = new fk_Light();
var spot = new fk_Light();
parallel.Type = fk_LightType.PARALLEL;
point.Type = fk_LightType.POINT;
spot.Type = fk_LightType.SPOT;
```

これにより、parallel は平行光源、point は点光源、spot はスポットライトとして定義される。デフォルトでは平行光源となる。

平行光源以外であれば、減衰係数を設定できる。減衰係数の設定には SetAttenuation() メソッドを使用する。

```
point.SetAttenuation(0.0, 0.01, 1.0);
spot.SetAttenuation(0.01, 0.0, 1.0);
```

引数はそれぞれ左から順番に線形減衰係数 k_l、2 次減衰係数 k_q、一定減衰係数 k_c を意味し、以下のような式で減衰関数 $f(d)$ は表される。

$$f(d) = \frac{1}{k_l d + k_q d^2 + k_c} \tag{4.1}$$

ただし、d は光源からの距離を表す。デフォルトでは線形減衰係数、2 次減衰係数が 0, 一定減衰係数が 1 に設定されており、これは距離による減衰がまったくないことを意味している。

スポットライトのカットオフ係数とスポットライト指数は、それぞれ SpotCutOff と SpotExponent というプロパティで設定する。

```
spot.SpotCutOff = Math.PI/6.0;
spot.SpotExponent = 0.00001;
```

SpotCutOff の値は弧度法による角度を入力する。Math.PI は円周率を表すので、例の場合は $\pi/6 = 30°$ となる。

なお、ここまで触れなかったが光源の大事な要素として色がある。色に関しては他の fk_Shape クラスと同じく要素として持つことはなく、fk_Model の属性として設定されていることに注意しなければならない。

本節で現れる用語は非常に難解で効果がわかりずらいものが多いと思われる。これらに関する詳しい解説や具体的な効果を (数学的に) 知りたい場合は、fk_Light のリファレンスマニュアルを参照してほしい。

4.13 パーティクル用クラス

FK では、パーティクルアニメーションをサポートするためのクラスとして fk_Particle 及び fk_ParticleSet が用意されている。厳密には、これらは fk_Shape クラスの派生クラスではなく、形状を直接表すものではないのだが、本質的に役割が似ていることから本章にて解説する。ここでは機能紹介にとどめるが、具体的な利用例に関しては 13 章にある「パーティクルアニメーション」サンプルを参照してほしい。

4.13.1 fk_Particle クラス

fk_Particle クラスは、パーティクル単体を表すクラスで、次のようなメソッドやプロパティが用意されている。

void Init(void)
: パーティクルを初期化するメソッド。

int ID
: パーティクルの ID を取得できるプロパティ。

uint Count
: パーティクルの年齢を取得できるプロパティ。

fk_Vector Position
: パーティクルの現在位置の設定や取得ができるプロパティ。

fk_Vector Velocity
: パーティクルの速度ベクトルの設定や取得ができるプロパティ。この値が有効なのは、Velocity プロパティか Accel プロパティに一回以上設定を行ったときのみである。

fk_Vector Accel

パーティクルの加速度ベクトルの設定や取得ができるプロパティ。この値が有効なのは、Accel プロパティに一回以上設定を行ったときのみである。

int ColorID

パーティクルの色 ID の設定や取得ができるプロパティ。

bool DrawMode

現在の描画状態の設定や取得ができるプロパティ。true ならば描画有効、false ならば無効となる。

4.13.2 fk_ParticleSet クラス

fk_ParticleSet クラスは、パーティクルの集合を表すクラスである。このクラスは、他のクラスの様に直接利用するものではなく、このクラスを継承させて抽象メソッドを上書きする形で利用する。まず、上書きすることになる抽象メソッドを紹介する。

4.13.2.1 fk_ParticleSet クラスの抽象メソッド

void GenMethod(fk_Particle p)

パーティクルの生成時に、パーティクルに対して行う処理を記述する。p には、新たに生成されたパーティクルオブジェクトが入っている。

void AllMethod(void)

毎ループ時に行う全体処理を記述する。

void IndivMethod(fk_Particle p)

毎ループ時の各パーティクルに個別に行う処理を記述する。p には、操作対象となるパーティクルインスタンスが入っている。

4.13.2.2 fk_ParticleSet クラスの通常のメソッド

また、fk_ParticleSet クラスは他にも以下のようなプロパティやメソッドを持っている。

bool AllMode

AllMethod() メソッドによる処理の有効化/無効化を設定するプロパティ。

bool IndivMode

IndivMethod() メソッドによる処理の有効化/無効化を設定するプロパティ。

void Handle()

実際に処理を 1 ステップ実行するメソッド。

fk_Shape Shape)

パーティクルを表す fk_Shape インスタンスを取得するプロパティ。

fk_Particle NewParticle(void)

パーティクルを新たに生成するメソッド。発生位置は原点となる。新たに生成されたパーティクルインスタンスを返す。

fk_Particle NewParticle(fk_Vector pos)

パーティクルを新たに生成するメソッド。初期位置は pos となる。新たに生成されたパーティクルインスタンスを返す。

fk_Particle NewParticle(double x, double y, double z)

パーティクルを新たに生成するメソッド。初期位置は (x, y, z) となる。新たに生成されたパーティクルインスタンスを返す。

bool RemoveParticle(fk_Particle p)

パーティクルを消去するメソッド。引数は消去したいパーティクルインスタンスである。通常は true を返すが、対象となるパーティクルが存在しなかった場合や、すでに消去されたパーティクルだった場合は false を返す。

bool RemoveParticle(int)

パーティクルを消去するメソッド。引数はパーティクル ID である。通常は true を返すが、対象となるパーティクルが存在しなかった場合や、すでに消去されたパーティクルだった場合は false を返す。

uint Count

パーティクル集合の年齢を取得するプロパティ。

uint ParticleNum

現状でのパーティクル個数を取得するプロパティ。

fk_Particle GetParticle(int)

ID を入力し、その ID を持つパーティクルを取得する。ID に相当するパーティクルが存在していない場合は null を返す。

fk_Particle GetNextParticle(fk_Particle p)

AllMethod 中で全パーティクルを取得する際に利用する。引数の種類によって、以下のようにパーティクルを返す。

1. 引数が null の場合は、ID が最も小さなパーティクルを返す。
2. 引数が最大の ID を持つパーティクルの場合は、null を返す。
3. 引数がそれ以外の場合は、入力パーティクル ID よりも大きな ID を持つものの中で最も小さな ID を持つパーティクルを返す。

uint MaxSize

パーティクルの最大個数の設定や参照を行うプロパティ。パーティクルの個数が最大値に達した場合、NewParticle() を呼んでも新たに生成されない。

void SetColorPalette(int ID, double r, double g, double b)

色パレットに色を設定する。

AllMethod 中で全てのパーティクルの平均座標ベクトルを求めるには、以下のように記述すればよい。(fk_ParticleSet クラスの派生クラス名を「MyParticle」とする。)

```
void AllMethod(void)
{
    fk_Particle   p;
    var vec = new fk_Vector();
    p = GetNextParticle(null);
    while(p != null)
    {
        vec += p.Position;
        p = GetNextParticle(p);
    }
    vec /= (double)ParticleNum;
}
```

第 5 章　動的な形状生成と形状変形

この章では、プログラム中で動的に形状を生成する手法を述べる。FK システムでは、形状に対する生成、参照、変形と言った操作の方法が数多く提供されているが、この章ではそれらの機能の中で比較的容易に扱える形状生成方法を解説する。

5.1　立体の作成方法 (1)

独立した頂点や線分ではなく、面を持つ立体を作成したい場合には、「**インデックスフェースセット (Index Face Set)**」(以下 IF セット) と呼ばれるデータを作成する必要がある。IF セットは、次の 2 つのデータから成り立っている。

- 各頂点の位置ベクトルデータ。
- 各面が、どの頂点を結んで構成されているかを示すデータ。

例として図 5.1 のような三角錐を作成してみる。

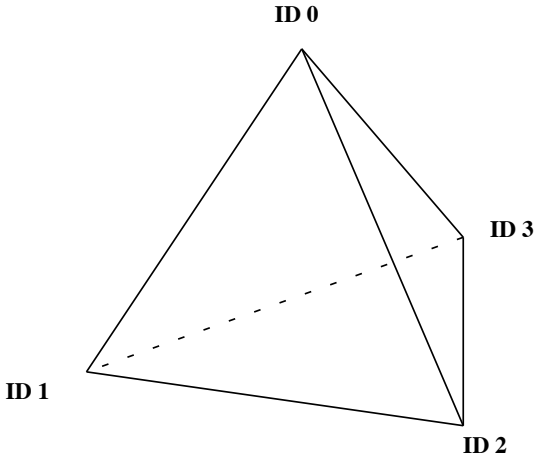

図 5.1　三角錐と各頂点 ID

ここで、それぞれの頂点の位置ベクトルは表 5.1 のとおりと想定する。

表 5.1　頂点位置ベクトル情報

頂点 ID	位置ベクトル
0	$(0, 10, 0)$
1	$(-10, -10, 10)$
2	$(10, -10, 10)$
3	$(0, -10, -10)$

このとき、4 枚の面はそれぞれ表 5.2 で示す頂点を結ぶことで構成されていることが、図 5.1 を参照することで確認できる。

表 5.2　面と頂点の ID 対応

平面番号	構成される頂点の ID
1 枚目	0, 1, 2
2 枚目	0, 2, 3
3 枚目	0, 3, 1
4 枚目	1, 3, 2

この 2 つのデータを、次のようにして入力する。「pos」が頂点の位置ベクトルを格納する配列、「IFSet」が各面の頂点 ID を格納する配列である。IFSet は、面数と角数を掛けた分を用意し、例にあるように続き番号で入力していく。

```
var ifs = new fk_IndexFaceSet();
var pos = new fk_Vector[4];
var IFSet = new int[3 * 4];

pos[0] = new fk_Vector(0.0, 10.0, 0.0);
pos[1] = new fk_Vector(-10.0, -10.0, 10.0);
pos[2] = new fk_Vector(10.0, -10.0, 10.0);
pos[3] = new fk_Vector(0.0, -10.0, -10.0);

IFSet[0] = 0; IFSet[1] = 1; IFSet[2] = 2;
IFSet[3] = 0; IFSet[4] = 2; IFSet[5] = 3;
IFSet[6] = 0; IFSet[7] = 3; IFSet[8] = 1;
IFSet[9] = 1; IFSet[10] = 3; IFSet[11] = 2;

ifs.MakeIFSet(4, 3, IFSet, 4, pos);
```

最終的には、MakeIFSet というメンバ関数を用いて fk_IndexFaceSet 型に情報を与えることになる。MakeIFSet は、次のような形式で用いることができる。

```
変数.MakeIFSet(面数, 角数, 各面頂点配列, 頂点数, 位置ベクトル配列);
```

例の場合、面数が 4、角数は三角形なので 3、頂点数は 4 になっている。今のところ、角数として用いることができるのは 3 か 4 (つまり三角形か四角形) のいずれかのみに制限されている。

5.2　立体の作成方法 (2)

前節では、全ての面が同じ角数であることが前提となっているが、ここでは各面で角数が同一でない立体の作成方法を解説する。今回は、次のような四角錐 (ピラミッド型) を想定する。

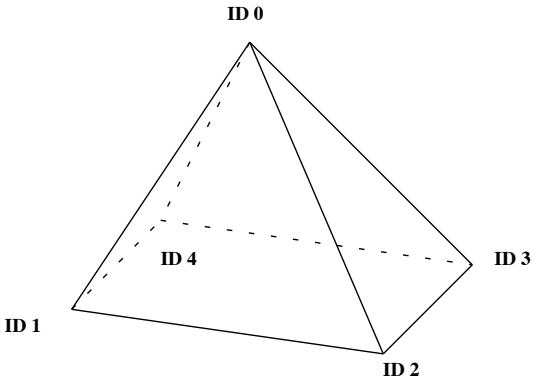

図 5.2　四角錐と各頂点 ID

前節と同様に、各頂点の位置ベクトルと面を構成する頂点 ID の表を記述すると次の表 5.3 のようになる。

表 5.3　位置ベクトルと頂点 ID 情報

頂点 ID	位置ベクトル
0	$(0, 10, 0)$
1	$(-10, -10, 10)$
2	$(10, -10, 10)$
3	$(10, -10, -10)$
4	$(-10, -10, -10)$

平面番号	構成される頂点の ID
1 枚目	0, 1, 2
2 枚目	0, 2, 3
3 枚目	0, 3, 4
4 枚目	0, 4, 1
5 枚目	4, 3, 2, 1

今回は、三角形と四角形が混在しているが、このような立体を作成するには次の例のようなプログラムを作成する。

```
var solid = new fk_IndexFaceSet();
List<int> polygon;

// 頂点リスト
var posArray = new List<fk_Vector>() {
    new fk_Vector(0.0, 10.0, 0.0),
    new fk_Vector(-10.0, -10.0, 10.0),
    new fk_Vector(10.0, -10.0, 10.0),
    new fk_Vector(10.0, -10.0, -10.0),
    new fk_Vector(-10.0, -10.0, -10.0)
};

// 面リスト
var IFSet = new List< List<int> >() {
    new List<int>() { 0, 1, 2 },
    new List<int>() { 0, 2, 3 },
    new List<int>() { 0, 3, 4 },
    new List<int>() { 0, 4, 1 },
    new List<int>() { 4, 3, 2, 1 }
};

solid.MakeIFSet(IFSet, posArray);
```

fk_IndexFaceSet クラスでの形状生成において、面の角数は 3 か 4 に限られるが、混在させることは可能である。

5.3 頂点の移動

第 5.1 節 〜 第 5.2 節で述べた方法で作成した様々な形状に対し、頂点を移動することで変形操作を行うことができる。頂点移動をするには、moveVPosition() を用いる。以下のプログラムは、ID が 2 である頂点を fk_Vector を用いて $(0, 1, 2)$ へ移動し、ID が 3 である頂点を数値だけで $(1.5, 2.5, 3, 5)$ へ移動させるものである。

```
var shape = new fk_IndexFaceSet();
var pos = new fk_Vector();
    :
    :
pos.Set(0.0, 1.0, 2.0);
shape.MoveVPosition(2, pos);
shape.MoveVPosition(3, 1.5, 2.5, 3.5);
```

5.3.1 fk_IndexFaceSet クラスでの汎用フォーマットファイル入出力

fk_IndexFaceSet では、各種形状データフォーマットでの入出力が可能となっている。現在サポートされているフォーマットを表 5.4 に示す。

表 5.4 fk_IndexFaceSet で利用できるファイルフォーマット

入力	SMF, VRML2.0, STL, HRC, RDS, DXF, MQO, Direct3D X
出力	VRML2.0, STL, DXF, MQO

以下に、各入出力用メソッドを個別に紹介する。

bool ReadSMFFile(string fileName)

SMF 形式のファイルから、形状データを入力する。「fileName」は入力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool ReadVRMLFile(string fileName)

VRML2.0 形式のファイルから、形状データを入力する。「fileName」は入力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool ReadSTLFile(string fileName)

STL 形式のファイルから、形状データを入力する。「fileName」は入力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool ReadHRCFile(string fileName)

HRC 形式のファイルから、形状データを入力する。「fileName」は入力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool ReadRDSFile(string fileName)

RDS(Ray Dream Studio) 形式のファイルから、形状データを入力する。「fileName」は入力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool ReadDXFFile(string fileName)

DXF 形式のファイルから、形状データを入力する。「fileName」は入力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool ReadMQOFile(string fileName, string objName, bool solidFlg, bool contFlg, bool matFlg)

MQO 形式のファイルから、形状データを入力する。各引数については、fk_IndexFaceSet の同名関数と同様なので、4.11.4 節を参照のこと。

bool ReadMQOFile(string fileName, string objName, int matID, bool solidFlg, bool contFlg, bool matFlg)

MQO 形式のファイルから、形状データを入力する。各引数については、fk_IndexFaceSet の同名関数と同様なので、4.11.4 節を参照のこと。

bool WriteVRMLFile(string fileName)

形状データを VRML 2.0 形式で出力する。「fileName」は出力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool WriteSTLFile(string fileName)

形状データを STL 形式で出力する。「fileName」は出力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool WriteDXFFile(string fileName)

形状データを DXF 形式で出力する。「fileName」は出力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。

bool WriteMQOFile(string fileName)

形状データを MQO 形式で出力する。「fileName」は出力ファイル名を指定する。成功すれば true、失敗した場合は false を返す。なお、オブジェクト名は「obj1」が自動的に付与される。

第 6 章　テクスチャマッピングと画像処理

この章では、画像を 3D 空間上に表示する「テクスチャマッピング」と呼ばれる技術の利用方法と、画像処理機能の使用方法を述べる。基本的に、テクスチャマッピングは 4 章で述べてきた形状の一種であり、利用方法は他の形状クラスとあまりかわらない。しかし、画像情報を扱うため独特の機能を多く保持するため、独立した章で解説を行う。

6.1　テクスチャマッピング

テクスチャマッピングとは、2 次元画像の全部及び一部を 3 次元空間上に配置して表示する技術である。テクスチャマッピングは、細かな質感を簡単に表現できることや、高速な表示機能が搭載されているハードウェアが普及してきていることから、非常に有用な技術である。FK システムでは、現在「矩形テクスチャ」、「三角形テクスチャ」、「IFS テクスチャ」の 3 種類のテクスチャマッピング方法をサポートしている。現在、入力可能な画像フォーマットは Windows Bitmap 形式、PNG 形式、JPEG 形式の 3 種類である。

6.1.1　矩形テクスチャ

最初に紹介するのは「矩形テクスチャ」と呼ばれるものである。これは、2 次元画像全体をそのまま (つまり長方形の状態で) 表示するための機能で、非常に簡単に利用できる。クラスとしては、「fk_RectTexture」というものを利用することになる。

6.1.1.1　基本的な利用方法

生成は、他の形状オブジェクトと同様に普通に変数を定義するだけでよい。

```
var texture = new fk_RectTexture();
```

画像ファイルの入力は、Windows Bitmap 形式の場合 ReadBMP() メソッドを用いる。このメソッドは、入力に成功した場合は true を、入力に失敗した場合は false を返す。

```
if(texture.ReadBMP("samp.bmp") == false)
{
    Console.WriteLine("File Read Error");
}
```

PNG 形式や JPEG 形式の画像ファイルを読み込みたい場合は、上記の ReadBMP() を ReadPNG(), ReadJPG() メソッドに置き換える。

また、テクスチャの 3 次元空間上での大きさの指定には TextureSize プロパティを用いる。

```
texture.TextureSize = new fk_TexCoord(50.0, 30.0);
```

この状態で、中心を原点、向きを $+z$ 方向としたテクスチャが生成される。

6.1.1.2 画像中の一部分の切り出し

fk_RectTexture クラスでは、画像の一部分を切り出して表示することも可能である。切り出し部分の指定方法として、「テクスチャ座標系」と呼ばれる座標系を用いる。テクスチャ座標系というのは、画像ファイルのうち一番左下の部分を $(0, 0)$、右上の部分を $(1, 1)$ として、画像の任意の位置をパラメータとして表す座標系のことである。例えば、画像の中心を表わすテクスチャ座標は $(0.5, 0.5)$ となる。また、100×100 の画像の左から 70 ピクセル、下から 40 ピクセルの位置のテクスチャ座標は $(0.7, 0.4)$ ということになる。

切り出す部分は、切り出し部分の左下と右上のテクスチャ座標を設定することになる。指定には SetTextureCoord() メソッドを利用する。以下は、左下のテクスチャ座標として $(0.2, 0.3)$、右上のテクスチャ座標として $(0.5, 0.6)$ を指定するサンプルである。

```
    texture.SetTextureCoord(0.2, 0.3, 0.5, 0.6);
```

6.1.1.3 リピートモード

ビルの外壁や地面を表すテクスチャを生成するときに、1 枚の画像をタイルのように行列状に並べて配置したい場合がある。これを全て別々のテクスチャとして生成するのはかなり処理時間に負担がかかってしまう。このようなとき、fk_RectTexture の「リピートモード」を用いると便利である。リピートモードとは、テクスチャを 1 枚だけ張るのではなく、タイル状に並べて配置するモードである。これを用いるには、次のようにすればよい。

```
    texture.TextureSize = new fk_TexCoord(100.0, 100.0);
    texture.RepeatMode = true;
    texture.RepeatParam = new fk_TexCoord(5.0, 10.0);
```

RepeatMode プロパティはリピートモードを用いるかどうかを設定するもので、true を代入するとリピートモードとなる。次の RepeatParam プロパティは並べる個数を設定するもので、例の場合は横方向に 5 枚、縦方向に 10 枚の合計 50 枚を並べることになる。それら全体のサイズが 100x100 なので、1 枚のサイズは 20x10 ということになる。ただし、リピートモードを用いる場合には画像サイズに制限があり、縦幅と横幅はいずれも 2^n (n は整数) である必要があり、現在のサポートは $2^6 = 64$ から $2^{16} = 65536$ までの間のいずれかのピクセル幅でなければならない。(ただし、縦幅と横幅は一致する必要はない。) 従って、リピートモードを用いるときはあらかじめ画像ファイルを補正しておく必要がある。

また、リピートモードを用いた場合は一部の切り出しに関する設定は無効となる。

6.1.2 三角形テクスチャ

次に紹介するのは「三角形テクスチャ」である。これは、入力した画像の一部分を三角形に切り出して表示する機能を持つ。これは、「fk_TriTexture」というクラスを利用する。テクスチャ用変数の定義や画像ファイル読み込みに関しては fk_RectTexture と同様である。

```
    var texture = new fk_TriTexture();
    texture.ReadBMP("samp.bmp");
```

fk_TriTexture の場合も、fk_RectTexture と同様に ReadBMP() を ReadPNG() や ReadJPG() に置き換えることで、PNG 形式や JPEG 形式の画像ファイルを入力できる。

次に、画像のどの部分を切り出すかを指定する。指定の方法は、前節で述べた「テクスチャ座標系」を利用する。切り出す部分は、このテクスチャ座標系を利用して 3 点それぞれを指定することになる。指定には SetTextureCoord() メソッドを利用する。最初の引数は、各頂点の ID を表わし、0, 1, 2 の順番で反時計回りとなるように設定する。2 番目、3 番目の引数はテクスチャ座標の x, y 座標を入力する。

```
texture.SetTextureCoord(0, 0.0, 0.0);
texture.SetTextureCoord(1, 1.0, 0.0);
texture.SetTextureCoord(2, 0.5, 0.5);
```

次に、3 点の 3 次元空間上での座標を設定する。設定には setVertexPos() メソッドを利用する。最初の引数が頂点 ID、2,3,4 番目の引数で 3 次元座標を指定する。

```
texture.SetVertexPos(0, 0.0, 0.0, 0.0);
texture.SetVertexPos(1, 50.0, 0.0, 0.0);
texture.SetVertexPos(2, 20.0, 30.0, 0.0);
```

SetVertexPos() メソッドは、fk_Vector 型の変数を引数に持たせることも可能である。

```
var vec = new fk_Vector(100.0, 0.0, 0.0);
texture.SetVertexPos(0, vec);
```

6.1.3 IFS テクスチャ

次に紹介する「IFS テクスチャ」は、多数の三角形テクスチャをひとまとめに扱うための機能を持つクラスで、クラス名は「fk_IFSTexure」である。このクラスでは、Metasequoia によって作成したテクスチャ付きの MQO ファイルを入力することができる入力用メソッド ReadMQOFile() の引数構成は、以下のようになっている。

```
ReadMQOFile(String fileName, String objName, int matID, bool contFlg);
```

「fileName」には MQO のファイル名、「objName」はファイル中のオブジェクト名を入力する。「matID」は、特定のマテリアルを持つ面のみを抽出する場合はその ID を指定する。全ての要素を読み込みたい場合は matID に -1 を入力する。

最後の「contFlg」はテクスチャ断絶のための設定である。これは、テクスチャ座標が不連続な箇所に対し、形状の位相を断絶する操作を行うためのものである。これを true にした場合断絶操作が行われ、テクスチャ座標が不連続な箇所が幾何的にも不連続に表示されるようになる。ほとんどの場合、この操作を行った場合の方がより適した描画となる。注意しなければならないのは、この断絶操作によって MQO データ中の位相構造とは異なる位相状態が内部で形成されることである。そのため、頂点、稜線、面といった位相要素は MQO データよりも若干増加する。

なお、「matID」と「contFlg」はそれぞれ「-1」と「true」というデフォルト引数が設定されており、このままで良いのであれば省略可能である。また、4.11.5 節と同様の用途として、ReadMQOData() メソッドも利用できる。引数の仕様は最初の引数が Byte 型配列になる以外は上記 ReadMQOFile() メソッドとと同様である。

以下の例は MQO ファイルからの読み込みのサンプルで、テクスチャ用画像ファイル名 (Windows Bitmap 形式) が「sample.bmp」、MQO ファイル名が「sample.mqo」、ファイル中のオブジェクト名が「obj1」であることを想定している。

```
var texture = new fk_IFSTexture();

if(texture.ReadBMP("sample.bmp") == false)
{
    Console.WriteLine("Image File Read Error");
}

if(texture.ReadMQOFile("sample.mqo", "obj1") == false)
{
    Console.WriteLine("Shape File Read Error");
}
```

ちなみに、ここで読み込んだ形状データは 8.6 節で述べているスムースシェーディングの制御に対応している。

fk_IFSTexture クラスが持つその他のメソッドとして、以下のようなものがある。

void Init()

テクスチャデータ及び形状データの初期化を行うメソッド。

fk_TexCoord GetTextureCoord(int triID, int vID)

三角形 ID が triID、頂点 ID が vID である頂点に設定されているテクスチャ座標を返すメソッド。

void SetTextureCoord(int triID, int vID, fk_TexCoord coord)

三角形 ID が triID、頂点 ID が vID である頂点に coord をテクスチャ座標として設定する。

fk_IndexFaceSet IFS)

fk_IFSTexture クラスは、形状データとして内部では fk_IndexFaceSet クラスによる変数を保持しており、その中に形状データを格納している。このプロパティは、そのインスタンスを取得するものである。頂点の移動などは、このインスタンスを介して fk_IndexFaceSet の機能を用いて可能となる。

6.1.4 メッシュテクスチャ

最後に、「メッシュテクスチャ」を紹介する。メッシュテクスチャは、前述した三角形テクスチャを複数枚同時に定義できる機能を持っている。これは、「fk_MeshTexture」というクラスを用いて実現できる。このクラスは、前述の 6.1.3 節で述べた IFS テクスチャとよく似ているが、以下のような点が異なっている。これらの性質を踏まえて、両方を使い分けてほしい。

表 6.1 IFS テクスチャとメッシュテクスチャの比較

項目	IFS テクスチャ	メッシュテクスチャ
形状生成	ファイル入力のみ	ファイル入力とプログラムによる動的生成
描画速度	高速	IFS テクスチャより若干低速
テクスチャ断絶	対応	非対応
D3DX アニメーション	対応	非対応

使い方は、まず生成する三角形テクスチャの枚数を TriNum プロパティに対して設定する。その後、fk_TriTexture と同様に SetTextureCoord() メソッドで各頂点のテクスチャ座標を、SetVertexPos() で空間上の位置座標を入力していくが、それぞれのメソッドの引数の最初に三角形の ID を入力するところだけが異なっている。

```
var texture = new fk_MeshTexture();

texture.TriNum = 2;
```

```
texture.SetTextureCoord(0, 0, 0.0, 0.0);
texture.SetTextureCoord(0, 1, 1.0, 0.0);
texture.SetTextureCoord(0, 2, 0.5, 0.5);
texture.SetTextureCoord(1, 0, 0.0, 0.0);
texture.SetTextureCoord(1, 1, 0.5, 0.5);
texture.SetTextureCoord(1, 2, 0.0, 1.0);
texture.SetVertexPos(0, 0, 0.0, 0.0, 0.0);
texture.SetVertexPos(0, 1, 50.0, 0.0, 0.0);
texture.SetVertexPos(0, 2, 20.0, 30.0, 0.0);
texture.SetVertexPos(1, 0, 0.0, 0.0, 0.0);
texture.SetVertexPos(1, 1, 20.0, 30.0, 0.0);
texture.SetVertexPos(1, 2, 0.0, 50.0, 0.0);
```

別の生成方法として、Metasequoia によって生成したテクスチャ付きの MQO ファイルを読み込むことも可能である。以下のように、ReadMQOFile() メソッドを利用する。例では、テクスチャ用画像ファイル名が「sample.bmp」、MQO ファイル名が「sample.mqo」、ファイル中のオブジェクト名が「obj1」と想定している。

```
fk_MeshTexture  texture;

if(texture.ReadBMP("sample.bmp") == false)
{
    Console.WriteLine("File Read Error");
}

if(texture.ReadMQOFile("sample.mqo", "obj1") == false)
{
    Console.WriteLine("File Read Error");
}
```

また、fk_MeshTexture クラスは複数のテクスチャ三角形平面によって構成されることになるが、PutIndexFaceSet メソッドを用いることによりその形状を fk_IndexFaceSet 型の形状として出力することが可能である。

```
var texture = new fk_MeshTexture();
var ifset = new fk_IndexFaceSet();

texture.PutIndexFaceSet(ifset);
```

6.1.5 テクスチャのレンダリング品質設定

矩形テクスチャ、三角形テクスチャ、IFS テクスチャ、メッシュテクスチャの全てにおいて、レンダリングの品質を設定することができる。やりかたは、以下のように RendMode プロパティで設定を行えばよい。

```
var texture = new fk_RectTexture();
    :
    :
texture.RendMode = fk_TexRendMode.SMOOTH;
```

モードは、通常モードである「fk_TexRendMode.NORMAL」と、アンチエイリアシング処理で高品質なレンダリングを行う「fk_TexRendMode.SMOOTH」が指定できる。デフォルトでは通常モード (fk_TexRendMode.NORMAL) となっている。

上記の例は矩形テクスチャで行っているが、fk_RectTexture, fk_TriTexture, fk_IFSTexture のいずれの型でも同様に利用できる。

6.2 画像処理用クラス

第 6.1 節で述べたテクスチャは、画像をファイルから読み込むことを前提としていたが、用途によってはプログラム中で画像を生成し、それをテクスチャマッピングするという場合もある。そのような場合、fk_Image というクラスを用いて画像を生成することが可能である。fk_RectTexture、fk_TriTexture、fk_IFSTexture、fk_MeshTexture にはそれぞれ Image というプロパティが用意されており、fk_Image クラスの変数をこのプロパティに設定することによって、それぞれのテクスチャに画像情報が反映されるようになっている。

以下のプログラムは、赤から青へのグラデーションを表す画像を生成し、fk_RectTexture に画像情報を反映させるプログラムである。

```
var texture = new fk_RectTexture();
var image = new fk_Image();

// 画像サイズを 256x256 に設定
image.NewImage(256, 256);

// 各画素に色を設定
for(int i = 0; i < 256; i++) {
    for(int j = 0; j < 256; j++) {
        Image.SetRGB(256-j, 0, j);
    }
}

// テクスチャに色を設定
texture.Image = image;
```

fk_Image クラスの主要なメソッドやプロパティを以下に羅列する。

void Init()
画像情報を初期化するメソッド。

bool ReadBMP(String fileName)
ファイル名が fileName である Windows Bitmap 形式の画像ファイルを読み込むメソッド。成功すれば true を、失敗すれば false を返す。

bool ReadPNG(String fileName)
ファイル名が fileName である PNG 形式の画像ファイルを読み込むメソッド。成功すれば true を、失敗すれば false を返す。

bool ReadJPG(String fileName)
ファイル名が fileName である JPEG 形式の画像ファイルを読み込むメソッド。成功すれば true を、失敗すれば false を返す。

bool WriteBMP(String fileName, bool transFlag)
現在格納されている画像情報を、Windows Bitmap 形式でファイル名が fileName であるファイルに書き出すメソッド。transFlag を true にすると、透過情報を付加した 32bit データとして出力し、false の場合通常のフルカラー 24bit 形式で出力する。書き出しに成功すれば true を、失敗すれば false を返す。

bool WritePNG(String fileName, bool transFlag)

現在格納されている画像情報を、PNG 形式でファイル名が fileName であるファイルに書き出すメソッド。transFlag を true にすると、透過情報も合わせて出力する。false の場合は透過情報を削除したファイルを生成する。書き出しに成功すれば true を、失敗すれば false を返す。

bool WriteJPG(String fileName, int quality)

現在格納されている画像情報を、JPEG 形式でファイル名が fileName であるファイルに書き出すメソッド。quality は 0 から 100 までの整数値を入力し、画像品質を設定する。数値が低いほど圧縮率は高いが画像品質は低くなる。数値が高いほど圧縮率は悪くなるが画像品質は良くなる。書き出しに成功すれば true を、失敗すれば false を返す。

void NewImage(int w, int h)

画像の大きさを横幅 w, 縦幅 h に設定するメソッド。これまでに保存されていた画像情報は失われる。

void CopyImage(const fk_Image image)

image の画像情報をコピーするメソッド。

void CopyImage(const fk_Image image, int x, int y)

現在の画像に対し、image の画像情報を左上が (x, y) となる位置に上書きを行うメソッド。image はコピー先の中に完全に包含されている必要があり、はみ出てしまう場合には上書きは行われない。

void SubImage(const fk_Image image, int x, int y, int w, int h)

元画像 image に対し、左上が (x, y)、横幅 w, 縦幅 h となるような部分画像をコピーするメソッド。x, y, w, h に不適切な値が与えられた場合は、コピーを行わない。

fk_Dimension Size

画像の縦横幅を得るプロパティ。

int GetR(int x, int y)

(x, y) の位置にある画素の赤要素の値を int 型で返すメソッド。

int GetG(int x, int y)

(x, y) の位置にある画素の緑要素の値を int 型で返すメソッド。

int GetB(int x, int y)

(x, y) の位置にある画素の青要素の値を int 型で返すメソッド。

int GetA(int x, int y)

(x, y) の位置にある画素の透明度要素の値を int 型で返すメソッド。

bool SetRGBA(int x, int y, int r, int g, int b, int a)

(x, y) の位置にある画素に対し、赤、緑、青、透明度をそれぞれ r, g, b, a に設定するメソッド。成功すれば true を、失敗すれば false を返す。

bool SetRGB(int x, int y, int r, int g, int b)

(x, y) の位置にある画素に対し、赤、緑、青をそれぞれ r, g, b に設定するメソッド。成功すれば true を、失敗すれば false を返す。

bool SetR(int x, int y, int r)

(x, y) の位置にある画素に対し、赤要素を r に設定するメソッド。成功すれば true を、失敗すれば false を返す。

bool SetG(int x, int y, int g)

(x, y) の位置にある画素に対し、緑要素を g に設定するメソッド。成功すれば true を、失敗すれば false を返す。

bool SetB(int x, int y, int b)

(x, y) の位置にある画素に対し、青要素を b に設定するメソッド。成功すれば true を、失敗すれば false を返す。

bool SetA(int x, int y, int a)

(x, y) の位置にある画素に対し、透明度要素を a に設定するメソッド。成功すれば true を、失敗すれば false を返す。

void FillColor(fk_Color col)

画像中の全てのピクセルの色要素を col が表わす色に設定するメソッド。

void FillColor(int r, int g, int b, int a)

画像中の全てのピクセルの色要素を、r を赤、b を青、b を緑、a を透明度として設定するメソッド。

また、以下のようにすることで、fk_Colol 型によって画素色値の参照や設定を行うことができる。

```
var image = new fk_Image();

// (x, y) = (100, 100) の画素色値を取得
var color = image[100, 100];

// (x, y) = (50, 70) の画素色値を設定
image[50, 70] = new fk_Color(0.4, 0.5, 0.3);
```

第 7 章　文字列表示

第 6 章でテクスチャマッピングについての解説を述べたが、FK システムでは特別なテクスチャマッピングとして、文字列を画面上に表示するクラスが用意されている。

文字列用テクスチャは 2 種類あり、容易に表示を実現する「fk_SpriteModel」クラスと、高度な機能を持つ「fk_TextImage」クラスである。この節では、まず fk_SpriteModel について述べる。

7.1　スプライトモデル

まず、簡易に文字列表示を実現する fk_SpriteModel クラスの利用方法を解説する。このクラスによって表示する文字列 (等) を「スプライトモデル」と呼ぶ[1)]。

スプライトモデルを使用した文字列表示は、以下のような手順を踏む。

1. fk_SpriteModel 型の変数を用意する。
2. フォント情報を読み込む。
3. サイズや表示位置などの各種設定を行っておく。
4. fk_Scene または fk_AppWindow に登録する。
5. DrawText() によって文字列を設定する。

以下、各項目を個別に説明する。

7.1.1　変数の準備とフォントの読み込み

まず、fk_SpriteModel 型の変数を準備する。

```
    var sprite = new fk_SpriteModel();
```

バージョン 4.2.10 以降の FK ではデフォルトで「Rounded M+」と呼ばれるフォントが設定してある。そのままデフォルトフォントを利用する場合はフォント情報を設定する必要はないが、もしフォントを変更したい場合はフォント情報の入力を行う。

fk_SpriteModel オブジェクトは、TrueType 日本語フォントを読み込むことができるので、まずは TrueType 日本語フォントを準備する。大抵の場合、拡張子が「ttf」または「ttc」となっているファイルである。TrueType フォントが格納されている場所は OS によって異なるが、容易に取得できるはずである[2)]。

TrueType フォントファイルが準備できたら、あとはそのファイル名を InitFont() メソッドを使って設定する。このメソッドは、フォントファイルの読み込みに成功したときは true を、失敗したときは false を返す。プログラムは、以下のように記述しておくとフォント読み込みの成功失敗を判定することができる。

```
    if(sprite.InitFont("sample.ttf") == false)
```

1)　本来の「スプライト」という単語は技術用語で、1980 年代頃の PC やゲーム機に搭載されていた機能であり、現在の PC やゲーム機ではこの技術は用いられていない。しかし、画面上の文字列やアイコンの表示に当時スプライト技術が用いられていた慣例から、現在でも画面上に表示される文字やアイコンを「スプライト」と呼称することがある。

2)　各 OS に搭載されているフォントデータは、他の PC にコピーすることがライセンス上禁じられていることも多い。別の PC にコピーすることを前提とする場合は、フリーライセンスを持つフォントを用いる必要がある。

```
    {
        Console.WriteLine("Font Init Error");
    }
```

7.1.2 各種設定

実際に表示を行う前に、必要な各種設定を行っておく。最低限必要な設定は表示位置の設定で、これは SetPositionLT() を利用する。

```
    sprite.SetPositionLT(-280.0, 230.0);
```

指定する数値はウィンドウ (描画領域) を原点とし、x の正方向が右、y の正方向が上となる。また、単位は (本来の 3D 座標系とは違い) ピクセル単位となる。スプライトモデルは、空間中の 3D オブジェクトとして配置するものではなく、画面の特定位置に固定して表示されることを前提としているためである。

その他、Size プロパティでスプライトを表す画像のサイズを設定しなおすなど、様々な設定項目があるが、これらについてはリファレンスマニュアルを参照してほしい。

文字色など、文字表示に関する細かな設定は fk_SpriteModel クラスには用意されていない。これらの設定は、fk_SpriteModel 型の public フィールドである「Text」に対して行う。例えば、以下のコードは文字色を白、背景色を黒に設定している。

```
    var sprite = new fk_SpriteModel();

    sprite.Text.ForeColor = new fk_Color(1.0, 1.0, 1.0);
    sprite.Text.BackColor = new fk_Color(0.0, 0.0, 0.0);
```

この「Text」フィールドは fk_TextImage 型である。詳細は 7.2 節を参照してほしい。

7.1.3 シーンやウィンドウへの登録

ウィンドウに fk_AppWindow を用いている場合は、通常のモデルと同様に Entry() メソッドによってスプライトを登録する。

```
    var window = new fk_AppWindow();
    var sprite = new fk_SpriteModel();
    window.Entry(sprite);
```

7.1.4 文字列設定

表示する文字列の設定は、DrawText() メソッドを用いて行う。

```
    var sprite = new fk_SpriteModel();
    sprite.DrawText("Sample");
```

引数は String 型なので、結果として String 型となるものであればよい。例えば、int 型の score という変数の値を用いて「SCORE = 100」のような表示を行いたい場合は、以下のようにすればよい。

```
var sprite = new fk_SpriteModel();
int score = 100;
sprite.drawText($"SCORE = {score}");
```

なお、DrawText() を二回以上呼び出した場合、通常は以前の文字列に追加した形で表示される。例えば、

```
sprite.DrawText("ABCD");
sprite.DrawText("EFGH");
```

とした場合、画面には「ABCDEFGH」と表示される。以前の「ABCD」を消去し「EFGH」と表示したい場合は、

```
    sprite.DrawText("ABCD");
    sprite.DrawText("EFGH", true);
```

というように、第 2 引数に「true」を入れるとよい。
また、表示した文字列を完全に消去したい場合は ClearText() メソッドを用いる。

7.2 高度な文字列表示

fk_SpriteModel にて簡単な文字列表示を実現できるが、より高度な文字列表示機能を提供するクラスとして「fk_TextImage」がある。fk_TextImage では、以下のような機能が実現できる。

- 画面上でのスプライト表示ではなく、3D 空間中の任意の位置に文字列テクスチャを表示する。
- 文字列に対し、色、大きさ、文字間や行間の幅、影効果など様々な設定を行う。
- 各文字を一文字ずつ順番に表示していくなどの文字送り機能を使う。

また、fk_SpriteModel の Text フィールドは fk_TextImage 型であり、この節で述べられている設定を施すことにより、fk_SpriteModel による文字表示においても同様の細かな設定を行うことができる。
fk_TextImage による文字列テクスチャの表示を行うには、以下のようなステップを踏むことになる。

1. fk_TextImage, fk_RectTexture 型のオブジェクトを用意する。
2. フォント情報を読み込む。
3. 文字列テクスチャに対する各種設定を行う。
4. 文字列情報を読み込む。
5. fk_RectTexture 型のオブジェクトに fk_TextImage 型のオブジェクトを設定する。

あとは、普通の fk_RectTexture 型と同様にして表示が可能となる。これらの項目は、次節以降でそれぞれを解説する。

7.2.1 文字列テクスチャの生成

fk_TextImage による文字列テクスチャを作成するには、最低でも fk_TextImage 型のインスタンスと fk_RectTexture 型のインスタンスが必要となる。従って、まずはそれぞれの変数を準備する。そして、fk_RectTexture の Image プロパティを用いて文字列テクスチャ (の画像イメージ) を fk_RextTexture インスタンスに設定しておく。

```
var textImage = new fk_TextImage();
var texture = new fk_RectTexture();
texture.Image = textImage;
```

7.2.2 フォント情報の読み込み

次に、フォント情報の読み込みを行う。フォント情報は fk_SpriteModel と同様に InitFont() を用いて行う。詳細は 7.1.1 節を参照してほしい。

7.2.3 文字列テクスチャの各種設定

次に、文字列テクスチャの各種設定を行う。設定できる項目として、以下のようなものが fk_TextImage のプロパティやメソッドとして提供されている。なお、fk_TextImage クラスは fk_Image クラスの派生クラスであり、以下のものに加えて第 6.2 節で述べた fk_Image クラスのメソッドやプロパティも全て利用することができる。

7.2.3.1 フォントに関する設定

int DPI, PTSize

DPI は文字列の解像度を設定するプロパティで、PTSize は文字の大きさを設定するプリパティである。デフォルトは両方とも 48 である。現状の FK システムではこの 2 つには機能的な差異がなく、結果的に 2 つの数値の積が文字の精細さを表すことになっている。以後、この 2 つの数値の積 (解像度 × 文字の大きさ) を「精細度」と呼ぶ。

bool MonospaceMode

文字を等幅で表示するかどうかを設定するプロパティ。true で等幅、false で非等幅となる。true の場合、元々のフォントが等幅でない場合でも等幅に補正して表示する。デフォルトでは true に設定されている。

int MonospaceSize

文字を等幅で表示する場合の、文字幅を設定するプロパティ。デフォルトでは 0 に設定されているので、このプロパティに幅を設定しないと表示自体が行われない。

int BoldStrength

文字の太さを数値に応じて太くするプロパティ。初期状態を 1 とし、高い値を与えるほど太くなる。どの程度太くなるのかは精細度による。

bool SmoothMode

出力される画像に対しアンチエイリアシング処理を行うかどうかを設定するプロパティ。デフォルトでは true に設定されている。

bool ShadowMode

影付き効果を行うかどうかを設定するプロパティ。デフォルトでは false に設定されている。

fk_Color ForeColor

文字列テクスチャの文字色を指定するプロパティ。デフォルトでは $(1, 1, 1, 1)$ つまり無透明な白に設定されている。

fk_Color BackColor

文字列テクスチャの背景色を指定するプロパティ。デフォルトでは $(0, 0, 0, 1)$ つまり無透明な黒に設定されている。

fk_Color ShadowColor

影付き効果の影の色を指定するプロパティ。デフォルトでは $(0, 0, 0, 1)$ つまり無透明な黒に設定されている。

fk_Dimension ShadowOffset

影付き効果の、影の相対位置を指定するプロパティ。x が正の場合右、負の場合左にずれる。y が正の場合下、負の場合上にずれる。デフォルトの値は、両方とも 0 に設定されている。

bool CacheMode

文字画像のキャッシュを保持するかどうかを設定するプロパティ。true の場合、一度読み込んだ文字のビットマップをキャッシュとして保持するようになるため、再度その文字を利用する際に処理が高速になる。ただし、キャッシュを行う分システムが利用するメモリ量は増加することになる。なお、キャッシュはシステム全体で共有

するため、異なるインスタンスで読み込んだ文字に関してもキャッシュが効くことになる。デフォルトでは false に設定されている。

void ClearCache()

CacheMode でキャッシュモードが有効であった場合に、保存されているキャッシュを全て解放するメソッド。このメソッドは static 宣言されているため、インスタンスがなくても「fk_TextImage::ClearCache();」とすることで利用可能である。

7.2.3.2 文字列配置に関する設定

fk_TextAlign Align

テキストのアライメントを設定するプロパティ。設定できるアライメントには以下のようなものがある。

表 7.1 文字列坂テキストのアライメント

fk_TextAlign.LEFT	文字列を左寄せに配置する。
fk_TextAlign.CENTER	文字列をセンタリング (真ん中寄せ) に配置する。
fk_TextAlign.RIGHT	文字列を右寄に配置する。

デフォルトでは fk_TextAlign.LEFT、つまり左寄せに設定されている。

void SetOffset(int up, int down, int left, int right)

文字列テクスチャの、縁と文字のオフセット (幅) を指定するメソッド。引数は順番に上幅、下幅、左幅、右幅となる。デフォルトでは全て 0 に設定されている。この値は、CharSkip や LineSkip と同様に、精細度に依存するものである。

int CharSkip

文字同士の横方向の間にある空白の幅を設定するプロパティ。デフォルトでは 0、つまり横方向の空間は「なし」に設定されている。この値は、前述した精細度に依存するもので、精細度が高い場合には表す数値の 1 あたりの幅は狭くなる。従って、精細度が高い場合にはこの数値を高めに設定する必要がある。

int LineSkip

文字同士の縦方向の間にある空白の高さを設定するプロパティ。デフォルトでは 0、つまり縦方向の空間は「なし」に設定されている。この値も精細度に依存するので、CharSkip と同様のことが言える。

int SpaceLineSkip

空行が入っていた場合、その空行の高さを指定するプロパティ。デフォルトでは 0、つまり空行があった場合は結果的に省略される状態に設定されている。この値も精細度に依存するので、CharSkip と同様のことが言える。

int MinLineWidth

通常、画像の横幅はもっとも横幅が長い行と同一となる。このプロパティは、生成される画像の横幅の最小値を設定する。生成される画像の幅が MinLineWidth 以内であった場合、強制的に MinLineWidth に補正される。

7.2.3.3 文字送りに関する設定

fk_TextSendingMode SendingMode

文字送り (7.2.8 節を参照のこと) のモードを設定するプロパティ。設定できるモードには以下のようなものがある。

表 7.2　文字送りのモード

fk_TextSendingMode.ALL	文字送りを行わず、全ての文字を一度に表示する。
fk_TextSendingMode.CHAR	一文字ずつ文字送りを行う。
fk_TextSendingMode.LINE	一行ずつ文字送りを行う。

デフォルトでは fk_TextSendingMode.ALL に設定されている。

7.2.4　文字列の設定

次に、表示する文字列を設定する。文字列を設定するには、fk_UniStr という型の変数を用いる。具体的には、次のようなコードとなる。

```
var str = new fk_UniStr();
    :
    :
str.Convert("サンプルの文字列です");
```

このように、Convert メソッドを用いて設定する。

fk_UniStr 型変数に格納した文字列を fk_TextImage に設定するには、LoadUniStr() メソッドを用いる。

```
var str = new fk_UniStr();
var image = new fk_TextImage();
    :
    :
str.Convert("サンプルの文字列です");
image.LoadUniStr(str);
```

7.2.5　文字列情報の読み込み

文字列を設定する方法は、前述した fk_UniStr 型を用いる方法の他に、テキストファイルを読み込むという方法もある。まず、文字列テクスチャに貼りたい文字列を事前にテキストファイルをどこか別のファイルに保存しておく。文字列を保存する際には、文字列テクスチャ内で改行したい箇所とテキストファイル内の改行は必ず合わせておく。あとは、そのファイル名を fk_TextImage オブジェクトに LoadStrFile() メソッドを用いて入力する。以下は、テキストファイル「str.txt」を入力する例である。

```
textImage.LoadStrFile("str.txt", FK_STR_SJIS);
```

第二引数には、以下の表 7.3 中にある対応した値を入力する。

表 7.3　文字コード対応表

fk_StringCode.UTF16	Unicode (UTF-16)
fk_StringCode.UTF8	Unicode (UTF-8)
fk_StringCode.JIS	ISO-2024-JIS (JIS コード)
fk_StringCode.SJIS	Shift-JIS (SJIS コード)
fk_StringCode.EUC	EUC

7.2.6　文字列読み込み後の情報取得

実際に文字列を読み込んだ後、fk_TextImage クラスには様々な情報を得るため以下のようなプロパティやメソッドが提供されている。

int LineNum
　読み込んだ文字列の行数を取得するプロパティ。

int AllCharNum
　文字列全体の文字数を取得するプロパティ。

int GetLineCharNum(int lineID)
　最初の行を 0 行目としたときの、lineID 行目の文字数を返すメソッド。

int GetLineWidth(int lineID)
　最初の行を 0 行目としたときの、lineID 行目の行幅 (単位ピクセル) を返すメソッド。

int GetLineHeight(int lineID)
　最初の行を 0 行目としたときの、lineID 行目の行の高さ (単位ピクセル) を返すメソッド。

int MaxLineWidth
　生成された行のうち、もっとも行幅 (単位ピクセル) が大きかったものの行幅を取得するプロパティ。

int MaxLineHeight
　生成された各行のうち、もっとも行の高さ (単位ピクセル) が大きかった行の高さを取得するプロパティ。

int GetLineStartXPos(int lineID)
　最初の行を 0 行目としたときの、lineID 行目の行の左上を表わす画素の x 方向の位置を返すメソッド。

int GetLineStartYPos(int lineID)
　最初の行を 0 行目としたときの、lineID 行目の行の左上を表わす画素の y 方向の位置を返すメソッド。

7.2.7　文字列テクスチャ表示のサンプル

前節までで各項目の解説を述べたが、ここではこれまでの記述を踏まえて典型的なコード例を示す。以下のコードは次のような条件を満たすようなコードである。

- TrueType フォント名は「fontsample.ttf」。
- 解像度、文字の大きさはそれぞれ 72, 72。
- 影付き効果を有効にする。
- 文字列の行間を「20」に設定。
- 文字色は「$(0.5, 1, 0.8)$」で無透明にする。
- 背景色は「$(0.2, 0.7, 0.8)$」で半透明にする。
- 影色は「$(0, 0, 0)$」で無透明にする。
- 影の相対配置は「$(5, 5)$」に設定。
- アラインはセンタリングにする。

```
var textImage = new fk_TextImage();
var texture = new fk_RectTexture();
var model = new fk_Model();
var str = new fk_UniStr();

texture.Image = textImage;

if(textImage.InitFont("fontsample.ttf") == false)
{
```

```
        Console.WriteLine("Font Init Error");
    }

    textImage.DPI = 72;
    textImage.PTSize = 72;
    textImage.ShadowMode = true;
    textImage.LineSkip = 20;
    textImage.ForeColor = new fk_Color(0.5, 1.0, 0.8, 1.0);
    textImage.BackColor = new fk_Color(0.2, 0.7, 0.8, 0.3);
    textImage.ShadowColor = new fk_Color(0.0, 0.0, 0.0, 1.0);
    textImage.ShadowOffset = new fk_Dimension(5, 5);
    textImage.Align = fk_TextAlign.CENTER;

    str.Convert("サンプルです。");
    textImage.LoadUniStr(str);

    model.Shape = texture;
```

7.2.8 文字送り

「文字送り」とは、読み込んだ文字列を最初は表示せず、一文字ずつ、あるいは一行ずつ徐々に表示していく機能のことである。この制御のために利用するメソッドは、簡単にまとめると表 7.4 のとおりである。

表 7.4 文字送りモード

SendingMode プロパティ	文字送りモード設定
LoadUniStr() メソッド	新規文字列設定
LoadStrFile() メソッド	新規文字列をファイルから読み込み
Send() メソッド	文字送り
Finish() メソッド	全文字出力
Clear() メソッド	全文字消去

以下に、詳細を述べる。

文字送りのモード設定に関しては前述した SendingMode プロパティを用いる。ここで fk_TextSendingMode.CHAR または fk_TextSendingMode.LINE が設定されていた場合、LoadUniStr() メソッドや LoadStrFile() メソッドで文字列が入力された時点では文字は表示されない。

Send() は、文字送りモードに応じて一文字 (fk_TextSendingMode.CHAR)、一列 (fk_TextSendingMode.LINE)、あるいは文字列全体 (fk_TextSendingMode.ALL) をテクスチャ画像に出力する。既に読み込んだ文字を全て出力した状態で Send() メソッドを呼んだ場合、特に何も起らずに false が返る。そうでない場合は一文字、一列、あるいは文字列全体をモードに従って出力を行い、true を返す。(つまり、最後の文字を Send() で出力した時点では true が返り、その後にさらに Send() を呼び出した場合は false が返る。)

Finish() メソッドは、文字送りモードに関わらずまだ表示されていない文字を全て一気に出力する。返り値は bool 型で、意味は Send() と同様である。

Clear() メソッドは、これまで表示していた文字を全て一旦消去し、読み込んだ時点と同じ状態に戻す。いわゆる「巻き戻し」である。1 文字以上表示されていた状態で Clear() を呼んだ場合 true が返り、まだ 1 文字も表示されていない状態で Clear() を呼んだ場合 false が返る。

具体的なプログラムは、以下のようになる。このプログラムは、描画ループが 10 回まわる度に一文字を表示し、現在表示中の文字列で、文字が全て表示されていたら str[] 配列中の次の文字列を読み込むというものである。処理の高速化をはかるため、CacheMode でキャッシュを有効としている。また、「c」キーを押した場合は表示されていた文字列を一旦消去し、「f」キーを押した場合は現在表示途中の文字列を全て出力する。(ウィンドウやキー操作に関しては、11 章を参照のこと。)

```
        var window = new fk_AppWindow();
        var textImage = new fk_TextImage();
        var str = new fk_UniStr[10];
        int loopCount, strCount;
                :
                :
        textImage.SendingMode = fk_TextSendingMode.CHAR;
        textImage.CacheMode = true;
        textImage.LoadUniStr(str[0]);


        loopCount = 1;
        strCount = 1;
        while(true)
        {
                        :
            if(window.GetKeyStatus('C', fk_Switch.UP) == true)
            {
                // 「c」キーを押した場合
                textImage.Clear();
            }
            else if(window.GetKeyStatus('F', fk_Switch.UP) == true)
            {
                // 「f」キーを押した場合
                textImage.Finish();
            }
            else if(loopCount % 10 == 0)
            {
                if(textImage.Send() == false && strCount != 9)
                {
                    textImage.LoadUniStr(str[strCount]);
                    strCount++;
                }
            }
            loopCount++;
        }
```

第 8 章　モデルの制御

この章では、fk_Model というモデルを司るクラスの使用法を述べる。「モデル」という単語は非常に曖昧な意味を持っている。FK でいうモデルとは、位置や方向を持った 1 個のオブジェクトとしての存在のことを指す。例えば、FK システムの中で建物や車や地形といったものを創造したいならば、それらをひとつのモデルとして定義して扱うことになる。fk_Shape クラスの派生クラス群による形状は、このモデルに代入されて初めて意味を持つことになる。形状は、形状でしかない。逆に、モデルにとって形状は 1 つのステータスである。

形状が、モデルのステータスであることは重要な意味を持つ。もし、モデルに不変の形状が存在してしまうなら、そのモデルの形状を変化させる手段は直接形状を変化させる以外にない。しかし、ある条件によってモデルの持つ形状を「入れ換えたい」と思うことはよくあることである。

例えば、視点から遠くにあるオブジェクトが大変細かなディティールで表現されていたとしても、処理速度の面から考えれば明らかに無駄である。それよりも、普段は非常に簡素な形状で表現し、視点から近くなったときに初めてリアルな形状が表現できればよい。このとき、モデルに対して形状を簡単に代入できる機能は大変重宝することになる。あるいは、アニメーション機能などの実現も容易に行なうことが可能であろう。

8.1　形状の代入

形状の代入法は大変単純である。次のように行なえばよい。

```
var sPos = new fk_Vector(100.0, 0.0, 0.0);
var ePos = new fk_Vector(0.0, 0.0, 0.0);
var line = new fk_Line();
var model = new fk_Model();

line.SetVertex(0, sPos);
line.SetVertex(1, ePos);

model.Shape = line;
```

つまり、形状インスタンスを Shape プロパティに代入すればよい。この例では fk_Line 型のオブジェクトを用いたが、fk_Shape クラスから派生したクラスのオブジェクトならばなんでもよい。

Shape プロパティによって形状の設定を行った後に、形状そのものに対し編集を行った場合、その編集結果は Shape プロパティに設定したモデルに直ちに反映する。次のプログラムを見てほしい。

```
var block = new fk_Block(100.0, 50.0, 200.0);
var model = new fk_Model();

model.Shape = block;
    :
    :   // 様々な処理が行われている。
    :
block.SetSize(100.0, fk_Axis.Y);
```

このプログラムの最後の行で、block の持つ形状が変化するわけだが、同時に model の持つ形状も変化することを意味する。

この考えを発展させれば、1 つの形状に対して複数のモデルに設定することが可能であることがわかる。次のプログラムはそれを示している。

```
var sphere = new fk_Sphere(4, 100.0);
var model = new fk_Model[4];
int i;

for(i = 0; i < 4; i++) {
    model[i] = new fk_Model();
    model[i].Shape = sphere;
}
```

このような手法は、プログラムの効率を上げるためにも効果的なものである。従って同じ形状を持つモデルは、同じ変数に対して Shape プロパティによる設定を行うべきである。球などは、その最も好例であると言える。

8.2 色の設定

モデルは、色を表すマテリアルステータスを持っている。色の指定には Material というプロパティに設定することによって行うが、その際には 3 章で述べた fk_Material 型のインスタンスを用いる。

```
var model = new fk_Model();
var material = new fk_Material();
    :   // この部分で material を
    :   // 作成しておく。
model.Material = material;
```

あるいは、付録 A に記述されているマテリアルオブジェクトをそのまま代入してもよい。

```
fk_Material.InitDefault();
model.Material = fk_Material.Green;
```

8.3 描画モードと描画状態の制御

モデルに与えられた形状が例えば球 (fk_Sphere) であった場合通常は面表示がなされるが、これをワイヤーフレーム表示や点表示に切り替えたい場合、DrawMode プロパティへの設定で実現できる。例えば球をワイヤーフレーム表示したい場合は、以下のようにすればよい。

```
var sphere = new fk_Sphere(4, 10.0);
var model = new fk_Model();

model.Shape = sphere;
model.DrawMode = fk_Draw.LINE;
```

現在選択できる描画モードは、以下の表 8.3 の通りである。

表 8.1　選択できる描画モード一覧

fk_Draw.POINT	形状の頂点を描画する。
fk_Draw.LINE	形状の稜線を描画する。
fk_Draw.FACE	形状の面のうち、表面のみを描画する。
fk_Draw.BACK_FACE	形状の面のうち、裏面のみを描画する。
fk_Draw.FRONTBACK_FACE	形状の面のうち、表裏両面を描画する。
fk_Draw.TEXTURE	テクスチャ画像を描画する。

点表示したい場合は fk_Draw.POINT、ワイヤーフレーム表示したい場合は fk_Draw.LINE、面表示したい場合は fk_Draw.FACE を引数として与えることで、モデルの表示を切り替えることができる。ただし、形状が fk_Point である場合に fk_Draw.LINE を与えたり、fk_Line である場合に fk_Draw.FACE を与えるといったような、表示状態が解釈できないような場合は何も表示されなくなるので注意が必要である。

また、この描画モードは 1 つのモデルに対して複数のモードを同時に設定することができる。例えば面表示とワイヤーフレーム表示を同時に行いたい場合は、次のように各モードを「|」で続けて記述することで実現できる。

```
    var model = new fk_Model();
    model.DrawMode = fk_Draw.FACE | fk_Draw.LINE;
```

なお、fk_Draw.POINT と fk_Draw.LINE では光源設定は意味がない。線や点に対する色設定に関しては、8.4 を参照してほしい。

8.4　線や点の色付け (マテリアル)

線や点に対して色を設定するには、それぞれ PointColor、LineColor というプロパティを用いる。それぞれのプロパティは fk_Color 型のインスタンスを設定する。次の例は、1 つの球を面の色が黄色、線の色が赤、点の色が緑に表示されるように設定したものである。

```
    var sphere = new fk_sphere(4, 10.0);
    var model = new fk_Model();

    model.Shape = sphere;
    fk_Material.InitDefault();
    model.DrawMode = fk_Draw.FACE | fk_Draw.LINE | fk_Draw.POINT;
    model.Material = fk_Material.Yellow;
    model.LineColor = new fk_Color(1.0, 0.0, 0.0);
    model.PointColor = new fk_Color(0.0, 1.0, 0.0);
```

8.5　線の太さや点の大きさの制御

描画モードで fk_Draw.POINT か fk_Draw.LINE を選択した場合、デフォルトでは描画される点のピクセルにおける大きさ、線の幅はともに 1 に設定されている。これをもっと大きく (太く) したい場合は、それぞれ PointSize、LineWidth プロパティを用いることで実現できる。次の例は、球に対して点描画と線描画を同時に行うモデルを作成し、点の大きさや線幅

を制御しているものである。

```
    var sphere = new fk_Sphere(4, 10.0);
    var model = new fk_Model();

    model.Shape = sphere;
    model.DrawMode = fk_Draw.POINT | fk_Draw.LINE;
    model.PointSize = 3.0;
    model.LineWidth = 5.0;
```

線の太さや点の大きさに関しては、環境による制限が生じる場合がある。例えば、ある太さ・大きさで固定されてしまう場合や、一定以上の太さ・大きさでは描画されないといった現象が起きることがある。この原因は主にグラフィックスハードウェア側の機能によるもので、プログラムで直接制御することは難しい場合が多い。

8.6 スムースシェーディング

FK システムでは、隣り合う面同士をスムースに描画する機能を保持している。これは、fk_Model 中の SmoothMode というプロパティを用いることで制御が可能である。これを用いると、例えば球などの本来は曲面で表現されている形状をよりリアルに表示することが可能となる。次のように、プロパティに true を代入することによってそのモデルはスムースシェーディングを用いて描画される。

```
    var model = new fk_Model();

    model.SmoothMode = true;    // スムースモード ON
    model.SmoothMode = false;   // スムースモード OFF
```

なお、この設定は後述するモデル間の継承関係の影響を受けない。

8.7 モデルの位置と姿勢

通常 3 次元のアプリケーションを作成する場合には、とても厄介な座標変換に悩まされる。これは、平行移動や回転を行列によって表現し、それらの合成によって状況を構築しなければならないからである。どのような 3 次元アプリケーションも結果的には行列によって視点やオブジェクトの位置や姿勢を表現するのだが、直接的に扱う場合は多くの困難な壁がある。

もう少し直感的な手段として、位置と姿勢を表現する方法が 2 つ存在する。1 つはベクトルを用いた方法であり、もう 1 つはオイラー角を用いた方法である。ベクトルを用いる場合、次の 3 つのステータスをオブジェクトは保持する。

- 位置ベクトル
- 前方向を表すベクトル (方向ベクトル)
- 上方向を表すベクトル (アップベクトル)

この表現は多くの開発者にとって直感的であろう。特に LookAt — オブジェクトがある位置からある位置を向く — の実装と大変相性が良い。この手段を用いた場合には最終的にはすべてを回転変換で表現できるような変換式を用いる。このとき問題となるのが、オブジェクトが真上と真下を見た場合に、変換式が不定になってしまうことである。

もう 1 つの手段としてのオイラー角は、普段から聞きなれた言葉ではない。オイラー角とは、実際には 3 つの角度から構

成されている。それぞれヘディング角、ピッチ角、バンク角 (しばしばロール角とも表現される) と呼ばれる。簡単に述べると、ヘディング角は東西南北のような緯度方向を、ピッチ角は高度を示す経度方向を、バンク角は視線そのものを回転軸とした回転角を表す。

この表現はすべての状況に矛盾を起こさないとても便利な手段である。しかしアプリケーションを作成する側から見ると、LookAt のような機能の実装には球面逆三角関数方程式と呼ばれる式を解かねばならず、骨が折れることだろう。また、この場合でも変換の際には解が不定となる場合が存在するので、根本的な解決とはならない。

FK システムでは、独自の方法でこれを回避している。FK システムは、モデルの各々が次のようなステータスを保持している。

- 位置
- 方向ベクトルとアップベクトル
- オイラー角
- 行列

実はこの方法では同じ意味を違う方法で 3 通りにも渡って表現していることになる。ここでは詳しく述べないが、この 3 通りもの表現は、互いに弱点を補間しあっており、あるステータスが不定になるような場合には他のステータスが適用されるようにできている。

一方で、fk_Model クラスではメソッドによってこれらの制御を行うのであるが、FK システムを用いた開発ではさきほど述べたようなわずらわしさからは一切解放される。これらはすべて内部的に行われ、ベクトル表現、オイラー角表現、行列表現のいずれもが常に整合性を保ち続けることを保証している。

次節からは、モデルに対しての具体的な位置と姿勢の操作を行うためのメソッドを、具体的な説明を交えながら述べていく。数は多いが、体系的なものなので理解はさほど難しくないだろう。

8.8 グローバル座標系とローカル座標系

3 次元アプリケーションの持つ座標系の重要な概念として、グローバル座標系とローカル座標系が挙げられる。グローバル座標系は、しばしばワールド座標系とも呼ばれる。平易な言葉で述べるなら、グローバル座標系は客観的な視点であり、ローカル座標系は主観的な視点である。

理解しやすくするために、車の運転を例にとって説明する。車が走っている場面をある場所から傍観しているとしよう。車は、背景の中を運転者の気の向くままに挙動している。つまり、3 次元座標内を前方向に前進しているということになる。一定時間が経てば、車の位置は進行方向にある程度進んでいることだろう。グローバル座標系は、このような運動を扱うときに用いられる。グローバル座標系はすべてのモデルが共通して持つ座標系であり、静的なモデル — この例では背景 — の位置座標は変化しない。

今度は車の運転者の立場を考えよう。運転者にとっては、前進することによって背景が後ろに過ぎ去っていくように見える。ハンドルを切れば、背景が回転しているように見える。もし北に向かって走っていれば右方向は東になるが、西に向かっていれば右は北になる。このとき、運転者にとっての前後左右がローカル座標系である。それに対し、東西南北にあたるものがグローバル座標系となる。

FK システムにおけるモデルの制御では、一部の例外を除いて常にグローバル座標系とローカル座標系のどちらを使用することもできる。グローバル座標系は、次のような制御に適している。

- 任意の位置への移動。
- グローバル座標系で指定された軸による回転。
- グローバル座標系による方向指定。

それに対し、ローカル座標系は次のような制御に適している。

- 前進、方向転換。
- オブジェクトを中心とした回転。
- ローカル座標系による方向指定。

かなり直観的な表現を使うと、グローバル座標系は東西南北を指定するときに用いられ、ローカル座標系は前後左右を指定するときに用いられると考えられる。どちらも、それぞれに適した場面が存在する。具体的な使用例は、13 章のプログラム例に委ねることにする。この章での目的は、実際の機能の紹介にある。

FK システムでは、グローバル座標系を扱うメソッドではプレフィックスとして Gl を冠し、ローカル座標系を扱うメソッドは Lo を冠するよう統一されている。以上のことを念頭において、ここからの記述を参照されたい。ちなみに、FK システムでは次のような左手座標系を採用しており、ローカル座標系もこれにならう。

1. モデルにとって、前は $-z$ 方向を指す。
2. モデルにとって、上は $+y$ 方向を指す。
3. モデルにとって、右は $+x$ 方向を指す。

8.9 モデルの位置と姿勢の参照

モデルの位置を参照したいときには、Position プロパティを用いる。このプロパティは fk_Vector 型のインスタンスとなる。

```
var model = new fk_Model();
fk_Vector pos;

pos = model.Position;
```

同様にして、モデルの方向ベクトルとアップベクトルもそれぞれ Vec プロパティと Upvec プロパティで参照できる。

```
var model = new fk_Model();
fk_Vector vec, upvec;
    :
    :
vec = model.Vec;
upvec = model.Upvec;
```

したがって、あるモデルの位置と姿勢を別のモデルにそっくりコピーしたいときは、この 3 つのステータスを代入すればよい。(代入法に関しては後述する。)

その他、モデルの持つオイラー角や行列も参照できる。それぞれ、fk_Angle 型の「Angle」、fk_Matrix 型の「Matrix」という名のプロパティを用いる。

```
var model = new fk_Model();
fk_Angle angle;
fk_Matrix matrix;
    :
    :
angle = model.Angle;
matrix = model.Matrix;
```

fk_Angle 型は、オイラー角を表現するクラスである。位置と方向ベクトルとアップベクトルを用いてモデルの状態をコピーすることを前述したが、これは位置とオイラー角を用いても可能である。単にコピーするだけならば、オイラー角を用いた方が便利であろう。ある法則を持ってずらして移動させる (たとえば元モデルの後部に位置させるなど) ような高度な制御を行うような場合には、ベクトル表現を用いて処理する方が良いときも多い。適宜選択するとよい。

8.10 平行移動による制御

モデルの方向を変化させず、モデルを移動させる手段として、fk_Model クラスでは 6 種類のメソッドを用意している。

GlTranslate(fk_Vector)
GlTranslate(double, double, double)
LoTranslate(fk_Vector)
LoTranslate(double, double, double)
GlMoveTo(fk_Vector)
GlMoveTo(double, double, double)

8.10.1 GlTranslate

GlTranslate メソッドは、モデルの移動ベクトルをグローバル座標系で与えるためのメソッドである。例えば、

```
var vec = new fk_Vector(1.0, 0.0, 0.0);
var model = new fk_Model();
int i;

for(i = 0; i < 10; i++) {
    model.GlTranslate(vec);
        :
        :
}
```

というプログラムは、ループの 1 周毎に model を x 方向に 1 ずつ移動させる。GlTranslate メソッドはベクトルの各要素を直接代入してもよい。

```
model.GlTranslate(1.0, 0.0, 0.0);
```

モデルに対して非常に静的な制御を行う場合には、むしろこの方が便利であろう。

8.10.2 LoTranslate

LoTranslate メソッドは、ローカル座標系で移動を制御する。最も多用される表現は、前進を表す次の記述である。

```
var model = new fk_Model();
double length;
int i;
```

```
    for(i = 0; i < 10; i++) {
        length = (double)i * 10.0;
        model.LoTranslate(0.0, 0.0, length);
            :
            :
    }
```

これにより、等加速度運動が表現されている (length は 10 ずつ増加しているから)。また (向いている方向によらずに) モデルを自身の右へ平行移動させることも、次の記述で可能である。

```
    for(int i = 0; i < 10; i++) {
        model.LoTranslate(0.0, 10.0, 0.0);
            :
            :
    }
```

例では述べられていないが、引数として fk_Vector 型のオブジェクトをとることも許されている。

8.10.3 GlMoveTo

GlTranslate メソッドが移動量を与えるのに対して、GlMoveTo メソッドは実際に移動する位置を直接指定するメソッドである。したがってこのメソッドにおいては、現在位置がどこであってもまったく関係がない。GlMoveTo メソッドを用いた移動表現は、Translate メソッド群を用いるよりも直接的なものとなる。

```
    for(int i = 0; i < 10; i++) {
        model.GlMoveTo(0.0, 0.0, (double)i * 10.0);
            :
            :
    }
```

このプログラムは、次のプログラムと同じ挙動をする。

```
    model.GlMoveTo(0.0, 0.0, 0.0);
    for(int i = 0; i < 10; i++) {
        model.GlTranslate(0.0, 0.0, 10.0);
            :
            :
    }
```

大抵の場合は工夫次第で同じ動作を多種に渡る表現によって実現可能であることは多い。できるだけ素直な表現を選択するよう努めるとよいだろう。よほど多くのモデルを相手にするのでなければ、選択によるパフォーマンスの差は問題にならない程度である。

なお、LoMoveTo メソッドは LoTranslate で代用できるため、LoVec メソッドと同一の理由で提供されていない。

8.11　方向ベクトルとアップベクトルの制御

FK システムにおいて、モデルの姿勢を制御する手法は大別すると方向ベクトルとアップベクトルを用いるもの、オイラー角を用いるもの、回転変換を用いるものの 3 種類がある。この節では、このうち方向ベクトルとアップベクトルを用いて制御するために提供されているメソッドを紹介する。3 種類のうち、この手法がもっとも直接的である。

この節では次の 8 種類のメソッドを紹介する。

GlFocus(fk_Vector)
GlFocus(double, double, double)
LoFocus(fk_Vector)
LoFocus(double, double, double)
GlVec(fk_Vector)
GlVec(double, double, double)
GlUpvec(fk_Vector)
GlUpvec(double, double, double)
LoUpvec(fk_Vector)
LoUpvec(double, double, double)

このうち、多重定義されているメソッドは移動メソッド群と同じように fk_Vector によるか、3 次元ベクトルを表す 3 つの実数を代入するかの違いでしかないので、実質的には 4 種類となる。

8.11.1　GlFocus

GlFocus メソッドは簡単に述べてしまうと、任意の位置を代入することによってその位置の方にモデルを向けさせるメソッドである。これは、あるモデルが別のモデルの方向を常に向いているというような制御を行いたいときに、特に真価を発揮する。次のプログラムは、それを容易に実現していることを示すものである。

```
// modelA は、常に modelB に向いている。

var modelA = new fk_Model();
var modelB = new fk_Model();

for(;;) {
        :   // ここで、modelA と modelB の移動が
        :   // 行われているとする。
        :
    modelB.GlFocus(modelA.Position);
        :
}
```

このメソッドで注意しなければならないのは、直接方向ベクトルを指定するものではないということである。直接指定するような処理を行いたい場合には、GlVec メソッドを用いればよい。

8.11.2　loFocus

LoFocus メソッドは、GlFocus のローカル座標系版である。Lo メソッド群に共通の、主観的な制御には好都合なメソッドである。例えば、

```
model.LoFocus(0.0, 0.0, 1.0);   // 後ろを向く。
model.LoFocus(1.0, 0.0, 0.0);   // 右を向く。
model.LFocus(-1.0, 0.0, 0.0);   // 左を向く。
model.LoFocus(0.0, 1.0, 0.0);   // 上を向く。
model.LoFocus(0.0, -1.0, 0.0);  // 下を向く。
model.LoFocus(1.0, 1.0, 0.0);   // 右上を向く。
model.LoFocus(-1.0, 1.0, -1.0); // 左前上方を向く。
model.LoFocus(0.01, 0.0, -1.0); // わずかに右を向く。
model.LoFocus(0.0, 0.01, -1.0); // わずかに上を向く。
```

といったような扱い方が代表的なものである。

8.11.3 GlVec

このメソッドは、モデルの方向ベクトルを直接指定するものである。このメソッドを用いた場合、アップベクトルの方向が前の状態とは関係なく自動的に算出されるため、モデルの姿勢を GlUpvec 等を用いて制御しない場合、思わぬ姿勢になることがある。

このメソッドは、もちろん GlUpvec 等と併用してモデルの姿勢を定義するのに有効だが、特に光源 (fk_Light) や円盤 (fk_Circle) のようにアップベクトルの方向に意味がないモデルを簡単に制御するのに向いているといえる。

なお、LoVec メソッドは提供されていない。なぜならば LoVec メソッドは意味的には LoFocus メソッドとまったく同じ機能を持つので、そのまま代用が可能となるからである。

8.11.4 GlUpvec

このメソッドは、アップベクトルを直接代入する。アップベクトルは本来方向ベクトルと直交している必要があるが、与えられたベクトルが方向ベクトルと平行であったり零ベクトルであったりしない限り、適当な演算が施されるので心配はいらない。逆に、このメソッドは方向ベクトルに依存して与えたアップベクトルを書き換えてしまうので、非常に融通の利かないメソッドともいえる。

実際このメソッドは、モデルのアップベクトルを常に固定しておく以外にはあまり使用することはない。アップベクトルを直接扱うことはある程度難解である。大抵の場合は、後述の回転変換を用いれば解決してしまう。ブランコのような表現や、コマのような表現も、回転変換を用いた方が明らかに簡単である。

8.11.5 LoUpvec

このメソッドは GlUpvec のローカル座標系版である。このメソッドはアップベクトルが方向ベクトルと直交していなければならないという理由から、z 方向の値は意味を持たない。このメソッドは GlFocus メソッドと比べてもさらに特殊な状況でしか扱われないであろう。ここでは紹介程度にとどめておく。

8.12 オイラー角による姿勢の制御

この節では、オイラー角による制御を提供する 4 種類のメソッドに関しての紹介が記述されている。4 つのメソッドは、次に示す通りである。

GlAngle(fk_Angle)

GlAngle(double, double, double)

LoAngle(fk_Angle)

LoAngle(double, double, double)

それぞれ多重定義がなされているが、3 つの実数を引数にとる 2 つのメソッドはこれまでのようにベクトルを意味しているのではなくオイラー角の 3 要素を示しており、3 つの引数はそれぞれヘディング角、ピッチ角、バンク角を表している。fk_Angle クラスはオイラー角を表現するためのクラスであり、プロパティとしてヘディング角を表す「h」、ピッチ角を表す「p」、バンク角を表す「b」に対して代入や参照を行うことができる。

fk_Angle のプロパティにしても、GlAngle(double, double, double) や LoAngle(double, double, double) にしても、値はすべて弧度法 (ラジアン) による。つまり、直角の値は $\frac{\pi}{2} \doteqdot 1.570796$ となる。

8.12.1 GlAngle

GlAngle メソッドはオイラー角を直接設定するメソッドである。相対的な変化量ではなく絶対的なオイラー角の値をここでは代入する。そういった点では、これは GlTranslate メソッドよりも GlMoveTo メソッドに近い。

オイラー角による表現は非常に手軽である反面、慣れないと把握が難しい。また、制御をベクトルによって行うかオイラー角によって行うかはアプリケーションそのものの設計にも深く関わってくる。あまり明示的な動作の指定や位置座標の指定を多用しないアプリケーションなら、オイラー角を用いた方が効果的な場合もある。しかし、GlFocus メソッドと GlAngle メソッドを Angle プロパティを用いずに併用することは、明らかに混乱を巻き起こすだろう。

GlAngle メソッドの効果的な使用法の 1 つとして、姿勢の初期化が上げられる。初期状態の姿勢を fk_Angle 型のオブジェクトに保管しておくことによって、いつでも姿勢を初期化できる。

```
var model = new fk_Model();
fk_Vector init_pos;
fk_Angle init_angle;
    :
    :
init_pos = model.Position;  // 位置のスナップショット
init_angle = model.Angle;   // 姿勢のスナップショット
    :
    :
// スナップショットを行った状態に戻す。
model.GlMoveTo(init_pos);
model.GlAngle(init_angle);
```

また、オイラー角の変化による立体の回転はアプリケーションのユーザにとって直観的であるため、ユーザインターフェースを介して立体を意のままに動かすようなアプリケーションにも威力を発揮するであろう。

8.12.2 LoAngle

オイラー角による制御は、GlAngle メソッドよりもむしろローカル座標系メソッドである LoAngle で真骨頂を発揮する。LoAngle メソッドでは、先に述べた LoFocus メソッドと非常によく似た機能を持つが、バンク角の要素を持つために LoFocus よりも応用性は高い。ここにその機能を羅列してみる。(Math.PI は円周率である。)

```
model.LoAngle(Math.PI, 0.0, 0.0);   // 後ろを向く。
model.LoAngle(Math.PI/2.0, 0.0, 0.0);   // 右を向く。
model.LoAngle(-Math.PI/2.0, 0.0, 0.0);  // 左を向く。
model.LoAngle(0.0, Math.PI/2.0, 0.0);   // 上を向く。
model.LoAngle(0.0, -Math.PI/2.0, 0.0);  // 下を向く。
model.LoAngle(Math.PI/2.0, Math.PI/4.0, 0.0);   // 右上を向く。
model.LoAngle(-Math.PI/4.0, Math.PI/4.0, 0.0);  // 左前上方を向く。
model.LoAngle(Math.PI/100.0, 0.0, 0.0); // わずかに右を向く。
model.LoAngle(0.0, Math.PI/100.0, 0.0); // わずかに上を向く。
model.LoAngle(0.0, 0.0, Math.PI);   // 視線を軸に半回転。
model.LoAngle(0.0, 0.0, Math.PI/2.0);   // モデルを右に傾ける。
```

```
model.LoAngle(0.0, 0.0, Math.PI/100.0); // わずかに右に傾ける。
```

LoFocus と比較してみてほしい。この LoAngle メソッドの特徴は、回転を角度代入によって行うことにある。こちらのほうが、開発者は直観的に定量的な変化を行うことが可能であろう。また、LoFocus と違ってアップベクトルの挙動の予想もできる。LoFocus の乱用は、時としてアップベクトルに対して予想と食い違った処理を施す可能性もある。LoAngle ではその心配はない。

8.13 回転による制御

前節のオイラー角による制御が姿勢を定義するためのものであるならば、ここで述べるメソッド群は位置を回転によって制御するためのものといえる。ここで述べられるメソッドは全部で 16 種類ある。

GlRotate(fk_Vector, fk_Axis, double)
GlRotate(double, double, double, fk_Axis, double)

GlRotate(fk_Vector, fk_Vector, double)
GlRotate(double, double, double, double, double, double, double)

LoRotate(fk_Vector, fk_Axis, double)
LoRotate(double, double, double, fk_Axis, double)

LoRotate(fk_Vector, fk_Vector, double)
LoRotate(double, double, double, double, double, double, double)

GlRotateWithVec(fk_Vector, fk_Axis, double)
GlRotateWithVec(double, double, double, fk_Axis, double)

GlRotateWithVec(fk_Vector, fk_Vector, double)
GlRotateWithVec(double, double, double, double, double, double, double)

LoRotateWithVec(fk_Vector, fk_Axis, double)
LoRotateWithVec(double, double, double, fk_Axis, double)

LoRotateWithVec(fk_Vector, fk_Vector, double)
LoRotateWithVec(double, double, double, double, double, double, double)

このメソッド群も、実質 8 種類のメソッドが引数として fk_Vector 型をとるものと 3 つの実数をとるもので多重定義がなされている。行間なく記されているもの同士が対応している。

8.13.1 GlRotate と GlRotateWithVec

GlRotate メソッドは、大きく 2 つの機能を持っている。次の引数を持つ場合、モデルはグローバル座標軸を中心に回転する。

```
var model = new fk_Model();
```

```
var pos = new fk_Vector(0.0, 0.0, 0.0);
    :
    :
model.GlRotate(pos, fk_Axis.X, Math.PI/4.0);
```

このうち、最初の引数には回転の中心となる軸上の点を指定する。例の場合は原点を指定している。次の引数は回転軸をどの軸に平行な直線にするかを指定するもので、fk_Axis.X、fk_Axis.Y、fk_Axis.Z から選択する。最後の引数は回転角を弧度法で入力する。中心は原点でなくてもよい。

一方ベクトル 2 つと実数 1 つを引数に取る場合には、GlRotate メソッドは任意軸回転演算として働く。回転軸直線上の 2 点を代入すればよい。最後の引数は回転角である。実数 7 つをとる場合も、1 番目から 3 番目、4 番目から 6 番目がそれぞれ 2 点の位置ベクトルを表す。次のプログラムは、モデルを $(100, 50, 0), (50, 100, 0)$ を通る回転軸を中心に 1 回転させるものである。

```
for(int i = 0; i < 200; i++) {
    model.GlRotate(100.0, 50.0, 0.0, 50.0, 100.0, 0.0, Math.PI/100.0);
        :
        :
}
```

GlRotate メソッドはあくまでモデルの位置を回転移動するためのものであり、姿勢や方向ベクトルはまったく変化しない。したがって、回転軸がモデルの位置を通る場合には、位置の移動がないために変化がない。回転移動の際に、方向ベクトルも同じように回転してほしい場合には GlRotateWithVec メソッドを用いるとよい。このメソッドは位置の回転とともに姿勢の回転も行われる。また回転軸がモデルの位置を通るように設定すれば、モデルは移動せずに方向だけ回転させることができる。これらのメソッドはモデルの挙動を予想しやすいので、安心して使用することができるであろう。

8.13.2 LoRotate と LoRotateWithVec

LoRotate メソッドと LoRotateWithVec メソッドは、ローカル座標系版であることを除けば GlRotate や GlRotateWithVec となんら変わりはない。特に LoRotateWithVec は、LoAngle メソッドと多くの機能が重複している。その例を表 8.13.2 に示す。ただし、origin は原点を、angle は回転角度を指す。

表 8.2 LoRotateWithVec と LoAngle の比較

LoRotateWithVec による表現	LoAngle による表現
LoRotateWithVec(origin, fk_Axis.X, angle)	LoAngle(0.0, angle, 0.0)
LoRotateWithVec(origin, fk_Axis.Y, angle)	LoAngle(angle, 0.0, 0.0)
LoRotateWithVec(origin, fk_Axis.Z, angle)	LoAngle(0.0, 0.0, angle)

回転の中心や軸の方向を任意にできることから、LoRotate の方が LoAngle よりも柔軟であるといえるだろう。

8.14 モデルの拡大縮小

FK システムでは、モデルに対して拡大や縮小を行うことが可能であり、次のようなメソッドが提供されている。

SetScale(double)

SetScale(double, fk_Axis)

SetScale(double, double, double)

PrdScale(double)
PrdScale(double, fk_Axis)
PrdScale(double, double, double)

SetScale() メソッドはモデルの絶対倍率を設定するためのメソッドである。引数として double 1 個のみをとるメソッドは、モデルの拡大や縮小を単純に行う。fk_Axis を引数にとる場合、指定された軸方向に対して拡大や縮小を行う。具体的には、次のように記述を行う。

```
var model = new fk_Model();

model.SetScale(2.0, fk_Axis.X); // X 方向に 2 倍に拡大
model.SetScale(0.4, fk_Axis.Y); // Y 方向に 0.4 倍に縮小
model.SetScale(2.5, fk_Axis.Z); // Z 方向に 2.5 倍に拡大
```

また、引数が double 3 個のものはそれぞれ x 方向、y 方向、z 方向への拡大率を示す。上の例は、次のように書き換えられる。

```
var model = new fk_Model();
model.setScale(2.0, 0.4, 2.5);
```

SetScale() は、現在のモデルの拡大率に対して相対的な拡大率を設定するものではなく、リンクされた形状に対する絶対的な拡大率を設定するためのメソッドである。もし相対的な指定を行いたい場合は、SetScale() ではなく PrdScale() を用いる。引数の意味は SetScale() と同様である。

8.15 モデルの親子関係と継承

8.15.1 モデル親子関係の概要

モデルに関する最後のトピックは継承に関してのものである。

FK システムを利用した開発者が車をデザインしたいと思ったとしよう。車は多くの部品から成り立っている。それらを最初から形状モデラで作成し、1 つの fk_Solid として読み込むのも 1 つの手である。しかしその他のプリミティブな形状、例えば fk_Block や fk_Sphere 等を利用して簡単な疑似自動車をデザインするような場合を考える。当然プリミティブな立体から車をデザインすることも容易な作業ではないが、問題はその後である。タイヤにあたる部分は車の中心から 4 つの隅方に地面に接して並んでいる。問題は、車が回転するような運動を行なった時にタイヤ自体は非常に複雑な動作をすることにある。これはベクトルの合成を用いて解決することは可能だが、プログラムが複雑になることに変わりはない。

そこで、FK システムでは複数のモデルをまとめて 1 つのモデルとして扱えるような機能を用意している。もし車体とタイヤの全てをグルーピングし、1 つのモデルとして制御できるのならば何の問題もない。これは、**継承**と呼ばれる手法を用いて実現することができるのである。

車の場合を考えよう。まず車体を準備する。次に 4 つのタイヤを車体に対して適当な位置に設定する。この相対的な位置関係が固定されれば、自動車は 1 つのモデルとして扱えるわけである。そこで、4 つのタイヤに対して自分の**親モデル**が車体であることを教えてやるのである。こうすれば、親の動きに合わせて**子モデル**は相対的な位置を保つような挙動を起こす。もう少し厳密に言うならば、子モデルは親モデルのローカル座標を固定されているわけである。

この機能は、fk_Model の持つ Parent プロパティによって実現されている。引数として親モデルのポインタを与える。このとき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。具体的なプログラムをここに示す。

```
var sphere = new fk_Sphere(4, 50.0);
var block = new fk_Block(300.0, 100.0, 500.0);
var CarBody = new fk_Model();
var CarTire = new fk_Model[4];
int i;

CarBody.Shape = block;
CarBody.GlTranslate(0.0, 100.0, 500.0);
for(i = 0; i < 4; i++)
{
    CarTire[i] = new fk_Model();
    CarTire[i].Shape = sphere;
}

CarTire[0].GlMoveTo(150.0, -50.0, 150.0);
CarTire[1].GlMoveTo(-150.0, -50.0, 150.0);
CarTire[2].GlMoveTo(150.0, -50.0, -150.0);
CarTire[3].GlMoveTo(-150.0, -50.0, -150.0);

for(i = 0; i < 4; i++)
{
    CarTire[i].Parent = CarBody;
}
```

プログラムを簡易なものにするため車体を fk_Block、タイヤを fk_Sphere で表現している。まず、CarBody モデルを GlTranslate によってある程度移動させる。次にタイヤの位置を、親モデルとの相対位置になる地点に CarTire を持ってくる。上記例の場合は球なので方向は関係ないが、必要ならばこのときに姿勢を定義しておく。そして、Parent プロパティに CarBody に設定することで、CarBody を親モデルとしている。

このプログラムは、以後に CarBody を動作させるとそれに付随して CarTire も動作するようになる。もし CarTire に対して移動を行うメソッドを呼ぶとどうなるか? このとき、CarTire は CarBody に対しての相対位置が変更される。これは、上記のプログラムにおいて `CarTire[i].Shape` と `CarTire[i].GlMoveTo()` の順序を反転させても支障がないことを示す。

また、既にあるモデルの子モデルとなっているモデルに対し、さらにその子モデル (元の親モデルからすれば、いわゆる「孫モデル」) を指定できる。例えば、タイヤをさらにリアルにするためにボルトを付加させることもできる。このときには、やはりボルトを (タイヤに対して) 相対的な位置に設定しておけばよい。

また、継承は座標系だけではなく、モデルのマテリアル属性にも反映される。もし子モデルのマテリアルが未定義であった場合、親モデルの色が設定される。子モデルにすでに色が設定されている場合には子モデルのマテリアルが優先される。

8.15.2 親子関係とモデル情報取得

モデルが親子関係を持った場合、モデルの情報取得は単純な話ではなくなる。例えば、モデルの位置を取得する Position プロパティやオイラー角を得る Angle プロパティの場合を考えてみる。

通常、これらの値はグローバル座標系における自身の座標ベクトルやオイラー角が返ってくるのだが、子モデルとなっている場合には親モデルに対する相対座標ベクトルや相対オイラー角となる。

これは多くの場合は都合が悪い。これらのみを用いる場合、子モデルの中心が実際にグローバル座標系でどこに位置するのかを知ることができない。そこで、子モデルのグローバル座標系における位置や姿勢等を得たいときのために、fk_Model は InhPosition (Inh は Inheritance – 継承の略) というプロパティを持っている。このプロパティは Position プロパティとまったく同様の使用法ではあるが、例え親モデルを持っていても正確なグローバル座標系による位置となる。これと同様に

して Angle に対応した InhAngle、Vec に対応した InhVec、Upvec に対応した InhUpvec、Matrix に対応した InhMatrix といったプロパティを fk_Model クラスは持っている。

もしモデルが親を持っていたとき、表 8.15.2 に示すプロパティ群は親に対する相対的な値を返す。

表 8.3　相対的な値を返すプロパティ群

プロパティの型	プロパティ名
fk_Vector	Position
fk_Vector	Vec
fk_Vector	Upvec
fk_Angle	Angle
fk_Matrix	Matrix

それに対し、表 8.15.2 のプロパティ群は絶対的なグローバル座標を返す。

表 8.4　絶対的な値を返すプロパティ群

プロパティの型	プロパティ名
fk_Vector	InhPosition
fk_Vector	InhVec
fk_Vector	InhUpvec
fk_Angle	InhAngle
fk_Matrix	InhMatrix

8.16　親子関係とグローバル座標系

通常、モデル同士に親子関係を設定したとき、子モデルは位置・姿勢共に変化する。これは前述したように、元々子モデルが持っていた位置や姿勢が、親モデルからの相対的なものとして扱われるようになるためである。

しかし、モデル間の親子関係は結びたいが、子モデルの位置や姿勢は変化させたくないというケースもある。同様に、親子関係を解消しても、子モデルの位置や姿勢を変化させたくないということもありえる。このような要求に応えるため、SetParent というメソッドでの第 2 引数で制御することが可能である。

```
    var modelA = new fk_Model();
    var modelB = new fk_Model();

    modelA.GlMoveTo(10.0, 0.0, 0.0);
    modelB.SetParent(modelA, true);
```

上記のプログラムで、通常では SetParent によって modelB もグローバル座標系では移動するのであるが、SetParent メソッドの 2 番目の引数に「true」を指定すると、SetParent による設定後も modelB は元の位置・姿勢を保つ。2 番目の引数に「false」を指定した場合、または 2 番目の引数を省略した場合、元の位置や姿勢や modelA からの相対的なものとして扱われるため、結果的に modelB は移動することになる。

8.16.1 親子関係に関するメソッド

以下に、親子関係に関する fk_Model のメソッドを羅列する。

SetParent(fk_Model m, bool flag)

m を親モデルとして設定する。flag が true の場合、設定後も元モデルのグローバル座標系での位置・姿勢が変化しない。false の場合は、元の位置・姿勢が m の相対的な位置・姿勢として扱われる。2 番目の引数を省略した場合、false と同じとなる。

DeleteParent(bool flag)

設定してあった親モデルとの関係を解除する。元々親モデルが設定されていなかった場合は何も起こらない。flag に関しては、SetParent() メソッドと同様。

EntryChild(fk_Model m, bool flag)

m を子モデルの 1 つとして設定する。flag に関しては、SetParent() メソッドと同様。

DeleteChild(fk_Model m, bool flag)

m が子モデルであった場合、親子関係を解除する。m が子モデルではなかった場合は何も起こらない。flag に関しては、SetParent() メソッドと同様。

DeleteChildren(bool flag)

自身に設定されている全ての子モデルに対し、親子関係を解除する。flag に関しては、SetParent() メソッドと同様。

第 9 章　干渉・衝突判定

ゲームのようなアプリケーションの場合、物体同士の衝突を検出することは重要な処理である。FK では、モデル同士の干渉や衝突を検出する機能を利用することができる。ここではまず、本書における用語の定義を行う。

干渉判定:
ある瞬間において、物体同士の干渉部分が存在するかどうかを判定する処理のこと。

衝突判定:
ある瞬間から一定時間の間に、物体同士が衝突するかどうかを判定する処理のこと。

干渉判定は、あくまで「ある瞬間」の時点での干渉状態を調べるものである。それに対し衝突判定は、「ある瞬間」において干渉状態になかったとしても、それから一定時間の間に衝突するようなケースも考慮することになる。文献によっては、両者を区別することがなかったり、干渉判定のことを「衝突判定」と呼称しているものも多いが、本書においては両者は厳密に区別する。

9.1　境界ボリューム

3 次元形状は多くの三角形 (ポリゴン) によって構成されており、これら全てに対し干渉判定や衝突判定に必要な幾何計算を行うことは、莫大な処理時間を要することがある。そのため、通常は判定する物体に対し簡略化された形状によって近似的に干渉・衝突判定を行うことが多い。このときの簡略化形状を**境界ボリューム**と呼ぶ。図 9.1 に境界ボリュームのイメージ図を示す。

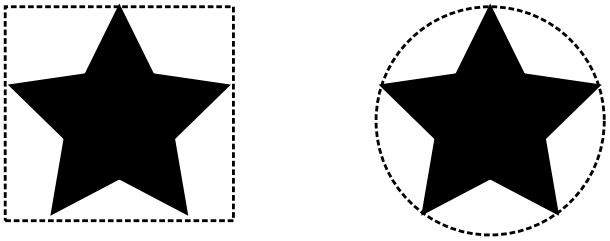

図 9.1　境界ボリューム

以下、FK で利用可能な境界ボリュームを紹介する。

9.1.1　境界球

最も簡易な境界ボリュームとして**境界球**がある。球同士の干渉判定は他の境界ボリュームと比べて最も高速であり、また FK 上で衝突判定を行える唯一の境界ボリュームである。その反面、長細い形状などでは判定の誤差が大きくなるという難点がある。

9.1.2　軸平行境界ボックス (AABB)

「**軸平行境界ボックス**」(Axis Aligned Boundary Box, 以下「**AABB**」) は、境界球と並んで処理の高速な干渉判定手法である。AABB では図形を直方体で囲み、その直方体同士で判定を行う。ただし、境界となる直方体の各辺は必ず x, y, z 軸のいずれかと平行となるように配置する。

直方体は、一般的に球よりも物体の近似性が高いため、物体の姿勢が常に変化せずに移動のみを行う場合はしばしば最適な手法となる。しかしながら、物体が回転する場合は注意が必要である。それは、物体が回転した場合に AABB の大きさも変化するためである。その様子を図 9.2 に示す。

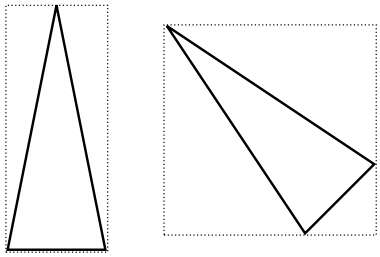

図 9.2　回転による AABB の変化

9.1.3　有向境界ボックス (OBB)

「**有向境界ボックス**」(Oriented Bounding Box, 以下「**OBB**」) は、AABB と同様に直方体による境界ボリュームであるが、モデルの回転に伴ってボックスも追従して回転する。モデルが回転してもボックスの大きさは変わらないので、非常に扱いやすい境界ボリュームである。図 9.3 にモデルの回転と追従するボックスの様子を示す。

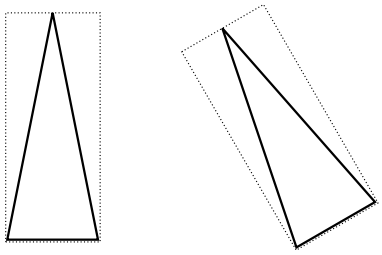

図 9.3　モデル回転に追従する OBB

しかしながら、OBB の干渉判定処理は実際にはかなり複雑であり、全境界ボリューム中最も処理時間を要する。モデルが少ない場合は支障はないが、多くのモデル同士の干渉判定を行う場合は処理時間について注意が必要である。

9.1.4　カプセル型

「**カプセル型**」とは、円柱に対し上面と下面に同半径の半球が合わさった形状である。モデルが回転した場合は、OBB と同様に方向は追従する。図 9.4 は、カプセル型境界ボリュームがモデルの回転に追従する様子である。

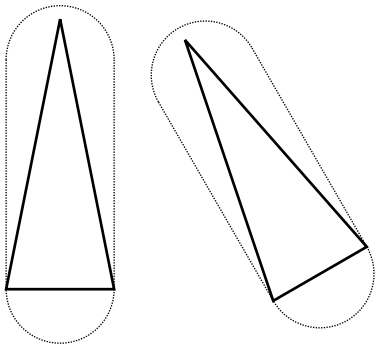

図 9.4　カプセル型の境界ボリューム

カプセル型は、形状は複雑に感じられるが、干渉判定の処理速度は OBB よりもかなり高速である。境界球では誤差が大きく、境界ボリューム自体もモデルの回転に追従してほしいが、多くのモデル同士の干渉判定が必要となる場面では、カプセル型が最も適していると言える。

9.2 モデル同士の干渉判定

9.2.1 干渉判定の基本

干渉判定を行うには、大きく以下の手順を行う。

1. どの境界ボリュームを利用するかを決める。
2. 境界ボリュームの大きさを設定する。
3. (モデルに対し干渉判定関係の設定を行う。)
4. 実際に他のモデルとの干渉判定を行う。

まずはどの境界ボリュームを利用するかを検討しよう。境界ボリューム今のところ 4 種類あり、それぞれ長所と短所があるので、モデルの形状や利用用途を考えて決定しよう。ここでは借りに「OBB」を利用すると想定する。境界ボリュームの種類設定は、BMode プロパティで行う。以降、サンプルプログラム中の「modelA」、「modelB」という変数は共に fk_Model 型であるとする。

```
    modelA.BMode = fk_BoundaryMode.OBB;
```

BMode に設定できる種類は、以下の表 9.2.1 の通りである。

表 9.1 境界ボリュームの設定値

fk_BoundaryMode.SPHERE	境界球
fk_BoundaryMode.AABB	軸平行境界ボックス (AABB)
fk_BoundaryMode.OBB	有向境界ボックス (OBB)
fk_BoundaryMode.CAPSULE	カプセル型

次に、境界ボリュームの大きさを設定する。これは自分自身で具体的な数値を設定する方法と、既にモデルに設定してある形状から自動的に大きさを算出する方法がある。自身で設定する場合は、fk_Model クラスの基底クラスとなっている fk_Boundary クラスのメソッドを利用することになるので、そちらのリファレンスマニュアルを参照してほしい。自動的に設定する場合は、あらかじめ Shape プロパティで形状を設定しておき、以下のように AdjustOBB() を呼び出すだけでよい。

```
    modelA.Shape = shape;
    modelA.adjustOBB();
```

あとは、他のモデルを引数として IsInter() メソッドを用いれば干渉判定ができる。(もちろん、modelB の方も事前に上記の各種設定をしておく必要がある。)

```
    if(modelA.IsInter(modelB) == true) {
        // modelA と modelB は干渉している。
    }
```

9.2.2 干渉継続モード

通常、干渉判定はその瞬間において干渉しているかどうかを判定するものであるが、「過去に他のモデルと干渉したことがあるか」を知りたいという場面は多い。そのようなときは、ここで紹介する「**干渉継続モード**」を利用するとよい。この機能を用いると、(明示的にリセットするまでは) 過去の干渉判定の際に一度でも干渉があったかどうかを取得することができる。

干渉継続モードを有効とするには、InterMode プロパティに true を設定する。

```
modelA.InterMode = true;
```

その後、干渉が生じたことがある場合は InterStatus が true となる。

```
if(modelA.InterStatus == true) {
    // 過去に干渉があった場合
}
```

この情報をリセットしたい場合は、ResetInter() を用いる。

```
modelA.ResetInter();
```

なお、このモードを利用するときに注意することとして、「過去に干渉があった」というのはあくまで「IsInter() を用いて干渉と判断された」ということを意味するということである。実際には他モデルと干渉していたとしても、そのときに IsInter() を用いて判定を行っていなかった場合は、このモードで「過去に干渉があった」とはみなされないので、注意が必要である。

9.2.3 干渉自動停止モード

他の物体と接触したときに、自動的に動きを停止するという制御がしばしば用いられる。例えば、動く物体が障害物に当たった場合に、その障害物にめり込まないようにするという場合である。そのような処理を自動的に実現する方法として「**干渉自動停止モード**」がある。このモードでは、あらかじめ干渉判定を行う他のモデルを事前に登録しておき、内部で常に干渉判定処理を行うので、自身で干渉判定を行う必要はない。

まずは事前に境界ボリュームの設定を行っておく。その上で、干渉判定を行いたいモデルを EntryInterModel() で登録しておく。

```
modelA.EntryInterModel(modelB);
```

その後、InterStopMode プロパティに true を設定することで、干渉自動停止モードが ON となる。

```
modelA.InterStopMode = true;
```

このモデルは、今後移動や回転を行った際に登録した他モデルと干渉してしまう場合、その移動や回転が無効となり、物体は停止する。この判定に関しての詳細はリファレンスマニュアルの fk_Model::InterStopMode の項に掲載されている。

このモードは便利ではあるが、状況次第ではモデルがまったく動かせなくなってしまう恐れもあるので、状況に応じて OFF にするなどの工夫が必要となることもあるので、注意してほしい。

9.3 モデル同士の衝突判定

衝突判定は、干渉判定と比べると複雑な処理を行うことになるが、物体同士の衝突を厳密に判定できるという利点がある。衝突判定を利用するには、以下のような手順を行う。

1. 境界球の大きさを設定しておく。
2. 衝突判定を行うモデルで事前に SnapShot() を呼んでおく。
3. モデルの移動や回転を行う。
4. IsCollision() を用いて衝突判定を行う。
5. 衝突していた場合、Restore() を衝突した瞬間の位置にモデルを移動させる。

まず、モデルに対し境界球の大きさを設定する。これも半径を直接指定する方法と、AdjustSphere() を用いて自動的に設定する方法がある。

次に、衝突判定を行うモデルに対し、毎回のメインループの中で SnapShot() を呼ぶ。

```
    while(true) {
            :
        modelA.SnapShot();
        modelB.SnapShot();
            :
    }
```

その後、モデルの移動や回転などを行った後に、IsCollision() で他モデルとの衝突判定を行う。IsCollision() メソッドは返値として衝突判定結果を返すが、衝突があった場合は第二引数にその時間を返すようになっている。もし衝突していた場合に Restore() を呼ぶようにしておく。

```
    double   t;

    while(true) {
            :
        if(modelA.IsCollision(modelB, t) == true) {
            modelA.Restore(t);
        }
            :
    }
```

Restore() では、引数を省略すると 0 を入力したときと同じ意味となり、その場合は SnapShot() を用いた時点での位置と姿勢に戻る。自身のプログラム中での挙動として、適している方を用いること。

9.4　光線とモデルの干渉判定

干渉判定はモデル同士だけではなく、光線とモデルの間でも行うことができる。この機能を使えば、例えばキャラクターの視線から見える物体の判定や、マウスが指しているモデルを検出することなど、様々な用途に利用することができる。

9.4.1　fk_Ray による光線設定

光線の生成は、fk_Ray クラスを用いて行う。fk_Ray 型変数では、始点と終点を設定することで空間上の線として配置される。始点と終点は変数生成時のコンストラクタで設定できる。(コンストラクタで引数を省略した場合、両端点が原点に設定される。)

```
var S = new fk_Vector(0.0, 0.0, 0.0);
var E = new fk_Vector(0.0, 0.0, 1.0);
var ray = new fk_Ray(S, E);
```

また、Set() メソッドを用いて動的に変更することも可能である。

```
var ray = new fk_Ray();
var S = new fk_Vector(0.0, 0.0, 0.0);
var E = new fk_Vector(0.0, 0.0, 1.0);
ray.Set(S, E);
```

fk_Ray が持つ形状の種類は「直線」「半直線」「有向線分」の 3 種類が選択できる。本書では、この 3 種の総称を「直線群」と呼称する。直線群の種類設定は Type プロパティを用いる。

```
var ray = new fk_Ray();
ray.Type = fk_LineType.HALF;
```

設定値は以下の表 9.2 の通りである。

表 9.2　直線群の種類

種類	設定値
直線	fk_LineType.OPEN
半直線	fk_LineType.HALF
有向線分	fk_LineType.CLOSE

光線の位置を直接指定する他に、fk_Model の変数を親モデルとして設定することもできる。設定には SetParent() メソッドを用いる。

```
var ray = new fk_Ray();
var model = new fk_Model();

ray.SetParent(model);
```

親子関係を用いることで、モデルの移動に追随して光線も移動することができる。例えば、カメラモデルの子モデルとして光線を設定しておけば、カメラの真正面にあるモデルを抽出することができる。

9.4.2 光線とモデルの干渉判定

光線とモデルの干渉判定を行うには、IsInter() メソッドを用いる。

```
var ray = new fk_Ray();
var model = new fk_Model();

model.BMode = fk_BoundaryMode.OBB;
model.AdjustOBB();

if(ray.IsInter(model) == true) {
    // 光線とモデルが干渉している。
}
```

干渉判定を行う際、モデル側は事前に境界ボリュームの種類と大きさを設定しておく必要がある。境界ボリュームの種類は、サポートされている全ての種類が使用できる。また境界ボリュームの種類を動的に変更しても問題ない。

複数のモデルに対して、一括して干渉判定を行いたい場合、EntryModel() と GetInterList() を用いると便利である。以下のプログラムは、複数のモデルを干渉判定対象として事前に登録するプログラムである。

```
var ray = new fk_Ray();
var model = new fk_Model [100];

for(int i = 0; i < 100; ++i) {
    model[i] = new fk_Model();
    model[i].BMode = fk_BoundaryMode.OBB;
    model[i].AdjustOBB();
    ray.EntryModel(model[i]);
}
```

登録したモデルに対して、一括干渉判定を行うには GetInterList() を用いる。

```
var ray = new fk_Ray();
var model = new fk_Model [100];

// (model[0]～model[99]が事前に登録されているものとする。
var list = ray.GetInterList(true);
if(list.Count > 0) {
    var (interModel, _) = list.First();
}
```

GetInterList() を用いると、list に干渉している全てのモデルが入る。一つも干渉していない場合、list は空となるので、まずはそのチェックを行う。GetInterList() の引数は、干渉モデルを始点に近い順にソートするかどうかの設定である。もし始点に一番近いモデルを取得したいのであれば、true にしておくとよい。(false の場合は EntryModel() での登録順となる。) GetInterList() の詳細は、リファレンスマニュアルを参照してほしい。

第 10 章　シーン

この章ではシーンと呼ばれる概念と、それを FK システム上で実現した fk_Scene というクラスに関しての使用法を述べる。

シーンは、複数のモデルとカメラからなる「場面」を意味する。シーンには複数の描画するためのモデル及びカメラを示すモデルを登録する。このシーンをウィンドウに設定することによって、そのシーンに登録されたモデル群が描画される仕組みになっている。

このシーンは、(fk_Scene クラスのオブジェクトというかたちで) 複数存在することができる。これは、あらかじめ全く異なった世界を複数構築しておくことを意味する。状況によって様々な世界を切替えて表示したい場合には、どのシーンをウィンドウに設定するかをうまく選択していけばよい。

10.1　モデルの登録

モデルの登録は、EntryModel() というメンバ関数を用いる。これは次のように使用される。

```
var model1 = new fk_Model();
var model2 = new fk_Model();
var model3 = new fk_Model();
var scene = new fk_Scene();

scene.EntryModel(model1);
scene.EntryModel(model2);
scene.EntryModel(model3);
```

登録したモデルをリストから削除したい場合は、RemoveModel() 関数を用いる。もし、シーン中に登録されていないモデルに対して RemoveModel() を用いた場合には、特に何も起こらない。以下の例は、DrawModeFlag が true の場合はモデルをシーンに登録し、そうでない場合はモデルをシーンから削除する。

```
var model1 = new fk_Model();
var scene = new fk_Scene();
bool DrawModelFlag;

if(DrawModelFlag == true) {
    scene.EntryModel(model1);
} else {
    scene.RemoveModel(model1);
}
```

また、一旦シーン中のモデルを全てクリアしたい場合には、ClearModel() という関数を呼ぶことで実現できる。

透明度が設定されているモデルがある場合、シーンへの登録の順序によって結果が異なることがある。具体的に述べると、透明なモデルの後に登録されたモデルは、透明なモデルの裏側にあっても表示されなくなる。これを防ぐには、透明な立体を常にディスプレイリストのできるだけ後ろに登録しておく必要がある。具体的には、別のモデルを登録した後に透明

なモデルを EntryModel() メンバ関数によって再び登録しなおせばよい[1)]。

ちなみに、実際に描画の際に透過処理を行うには fk_Scene オブジェクトにおいて BlendStatus プロパティを用いて透過処理の設定を行う必要がある。これは第 10.4 節に詳しく述べる。

10.2 カメラ (視点) の設定

カメラ (視点) の設定は、モデルを Camera プロパティに代入することによって行われる。

```
var camera = new fk_Model();
var scene = new fk_Scene();
scene.Camera = camera;
```

Camera プロパティによってカメラが登録された場合、カメラを意味するオブジェクトの位置や方向が変更されれば、再代入する必要なくカメラは変更される。これは多くの場合都合がよい。もし、別のモデルをカメラとして利用したいのならば、別のモデルを代入すればよい。

10.3 背景色の設定

FK システムでは、背景色はデフォルトで黒に設定されているが、次のように fk_Scene の BGColor プロパティを用いることで背景色を変更することができる。このプロパティは fk_Color 型のオブジェクトを設定または取得することができる。以下の例は、背景色を青色に設定している例である。

```
var scene = new fk_Scene();
scene.BGColor = new fk_Color(0.0, 0.0, 1.0);
```

10.4 透過処理の設定

モデルのマテリアルにおいて、Alpha プロパティを用いて立体に透明度を設定することが可能であるが、実際に透過処理を行うには fk_Scene クラスのオブジェクトにおいて BlendStatus プロパティを用いて設定を行わなければならない。具体的には、次のようにプロパティに true を設定することによって透過処理の設定が行える。

```
var scene = new fk_Scene();
scene.BlendStatus = true;
```

透過処理を無効にしたい場合は、false を設定すればよい。もし透過処理を ON にした場合、(実際に透明なモデルが存在するか否かに関わらず) 描画処理がある程度低速になる。

10.5 霧の効果

FK システムでは、シーン全体に霧効果を出す機能がサポートされている。まず、霧効果の典型的な利用法を 10.5.1 節で解説し、霧効果の詳細な利用方法を 10.5.2 節で述べる。

1) わざわざ RemoveModel() を呼ばなくても、自動的に重複したモデルはシーンから削除されるようになっている。

10.5.1　霧効果の典型的な利用方法

霧効果は、次の 4 項目を設定することで利用できる。

- 減衰関数の設定。
- オプションの設定。
- 係数の設定。
- 霧の色設定。

これを全て行うコード例は、以下のようなものである。

```
    var scene = new fk_Scene();

    scene.FogMode = fk_FogMode.LINEAR;
    scene.FogOption = fk_FogOption.FASTEST;
    scene.FogLinearStart = 0.0;
    scene.FogLinearEnd = 400.0;
    scene.FogColor = new fk_Color(0.3, 0.4, 1.0, 0.0);
```

まず、FogMode プロパティによって減衰関数を指定する。通常は、例にあるように fk_FogMode.LINEAR_FOG を指定すれば良い。

次に、FogOption プロパティでオプションを指定する。これは 10.5.2 節で述べるような 3 種類があるので、目的に応じて適切に設定する。

次に、霧効果の現れる領域を設定する。FogLineaerStart プロパティの数値が霧が出始める距離、FogLinearEnd プロパティの数値が霧によって何も見えなくなる距離である。

最後に、FogColor プロパティによって霧の色を設定する。通常は背景色と同一の色を設定すればよい。また、最後の透過度数値は 0 を指定すれば良い。

以上の項目を設定するだけで、シーンに霧効果を出すことが可能となる。より詳細な設定が必要な場合は、次節を参照すること。

10.5.2　霧効果の詳細な利用方法

霧効果に関連する機能として、次のような fk_Scene のプロパティ群が提供されている。

fk_FogMode FogMode
fk_FogOption FogOption
double FogDensity
double FogLinearStart
double FogLinearEnd
fk_Color FogColor

以下に、各プロパティの解説を述べる。

fk_FogMode FogMode

霧による減衰の関数を設定する。以下のような項目が入力できる。各数値の設定はその他の設定関数を参照すること。

fk_FogMode.LINEAR_FOG	減衰関数として $\frac{E-z}{E-S}$ が選択される。
fk_FogMode.EXP	減衰関数として e^{-dz} が選択される。
fk_FogMode.EXP2	減衰関数として $e^{-(dz)^2}$ が選択される。
fk_FogMode.NONE	霧効果を無効とする。

なお、デフォルトでは fk_FogMode.NONE が選択されている。

fk_FogOption FogOption

霧効果における描画オプションを設定する。以下のような項目が入力できる。

fk_FogOption.FASTEST	描画の際に、速度を優先する。
fk_FogOption.NICEST	描画の際に、精細さを優先する。
fk_FogOption.NOOPTION	特にオプションを設定しない。

なお、デフォルトでは fk_FogOption.NOOPTION_FOG が選択されている。

double FogDensity

減衰関数として fk_FogMode.EXP (e^{-dz}) 及び fk_FogMode.EXP2 ($e^{-(dz)^2}$) が選択された際の、 減衰指数係数 d を設定する。ここで、z はカメラから対象地点への距離を指す。

double FogLinearStart

double FogLinearEnd

減衰関数として fk_FogMode.LINEAR ($\frac{E-z}{E-S}$) が選択された際の、減衰線形係数 S, E を設定する。もっと平易に述べると、霧効果が現れる最初の距離 S と、霧で何も見えなくなる距離 E を設定する。

fk_Color FogColor

霧の色を設定する。大抵の場合、背景色と同一の色を指定し、透過度は 0 にしておく。

10.6 オーバーレイモデルの登録

表示したい要素の中には、物体の前後状態に関係なく常に表示されてほしい場合がある。例えば、画面内に文字列を表示する場合などが考えられる。このような処理を実現するため、通常のモデル登録とは別に「オーバーレイモデル」として登録する方法がある。モデルをオーバーレイモデルとしてシーンに登録した場合、表示される大きさや色などは通常の場合と変わらないが、描画される際に他の物体よりも後ろに存在していたとしても、常に全体が表示されることになる。

オーバーレイモデルは一つのシーンに複数登録することが可能である。その場合、表示は物体の前後状態は関係なく、後に登録したモデルほど前面に表示されることになる。

基本的には、通常のモデル登録と同様の手順でオーバーレイモデルを登録することができる。オーバーレイモデルを扱うメンバ関数は、以下の通りである。

void EntryOverlayModel(fk_Model model)

モデルをオーバーレイモデルとしてシーンに登録する。「model」が既にオーバーレイモデルとして登録されていた場合は、そのモデルが最前面に移動する。

void RemoveOverlayModel(fk_Model model)

「model」がオーバーレイモデルとして登録されていた場合、リストから削除する。

void ClearOverlayModel(void)

全ての登録されているオーバーレイモデルを解除する。

第 11 章　ウインドウとデバイス

FK システムでは、ウィンドウを制御するクラスとして fk_AppWindow クラスと fk_Window クラスを提供している。fk_AppWindow は簡易に様々な機能を実現できるものであり、実装を容易に行うことを優先したものとなっている。fk_Window は fk_AppWindow よりも利用方法はやや複雑であるが、多くの高度な機能を持っており、マルチウィンドウや GUI と組み合わせたプログラムを作成することができる。

本章では、まず fk_AppWindow による機能を紹介し、その後に fk_Window 固有の機能について解説を行う。

11.1　ウィンドウの生成

ウィンドウは、一般的なウィンドウシステムにおいて描画をするための画面単位である。FK システムでは、fk_AppWindow クラスのオブジェクトを作成することによって 1 つのウィンドウを生成できる。

```
var window = new fk_AppWindow();
```

fk_AppWindow に対して最低限必要な設定は大きさの設定である。以下のように、Size プロパティに fk_Dimension 型のインスタンスを代入することで、ピクセル単位で大きさを指定する。

```
window.Size = new fk_Dimension(600, 600);
```

また、背景色を設定するには BGColor プロパティに fk_Color 型のインスタンスを代入することで行う。

```
window.BGColor = new fk_Color(0.3, 0.5, 0.2);
```

これらの設定を行った後、Open() メソッドを呼ぶことで実際にウィンドウが画面に表示される。

```
window.Open();
```

11.2　ウィンドウの描画

ウィンドウの描画は、ウィンドウ用の変数 (インスタンス) で Update() メソッドを呼び出すことで行われる。このメソッドが呼ばれた時点で、リンクされているシーンに登録されている物体が描画される。

Update() メソッドは、ウィンドウが正常に描画された場合に true を、そうでない場合は false を返す。false を返すケースは、ウィンドウが閉じられた場合となる。そのため

```
while(window.Update()) {
    :
}
```

というコードでは、ウィンドウが閉じられると while ループを脱出するようになる。これを踏まえ、実際の描画ループは次のようになる。

```
var window = new fk_AppWindow();

window.Size = new fk_Dimension(600, 600);
window.BGColor = new fk_Color(0.1, 0.2, 0.3);
window.Open();
while(window.Update()) {
    :
    :   // モデルの制御
    :
}
```

なお、生成されたウィンドウは通常の OS によるウィンドウ消去の方法以外に、ESC キーを押すことで消去する機能がある。

11.3 表示モデルの登録と解除

fk_AppWindow クラスでは、第 10 章で説明した「シーン」が最初からデフォルトで登録されており、fk_Scene 型の変数を用いなくても表示モデルを登録することが可能である。

表示モデルの登録は、Entry() メソッドを用いる。

```
var window = new fk_AppWindow();
var model = new fk_Model();

window.Entry(model);
```

Entry() メソッドは、fk_Model の他に fk_SpriteModel 型のインスタンスにも用いることができる。

一度登録したモデルを描画対象から外したい場合は、Remove() メソッドを用いる。

```
var window = new fk_AppWindow();
var model = new fk_Model();


window.Remove(model);
```

また、登録したモデルを全て解除したい場合は、ClearModel() というメソッドを用いる。

```
window.ClearModel(); // 全ての登録を解除
```

11.4　シーンの切り替え

描画対象をモデル単位ではなくシーン単位で切り替えたい場合は、まず第 10 章で説明した fk_Scene 型を用いてシーンを構築しておく。その後、「Scene」というプロパティに設定することで表示対象となるシーンが切り替わる。

```
    var window = new fk_AppWindow();
    var scene = new fk_Scene;

    window.Scene = scene;
```

また、表示シーンを fk_AppWindow がデフォルトで保持しているシーンに切り替えたい場合は、SetSceneDefault() というメソッドを用いる。

11.5　カメラ制御

fk_AppWindow でカメラを制御する方法は、大きく 2 種類ある。

手っ取り早く制御する方法は、「CameraPos」というプロパティと「CameraFocus」というプロパティを用いるというものである。CameraPos プロパティは fk_Vector 型で、カメラ位置の設定や取得ができる。また、CameraFocus プロパティもやはり fk_Vector 型で、カメラが注目する位置 (注視点) を指定する。

```
    window.CameraPos = new fk_Vector(0.0, 0.0, 100.0);
    window.CameraFocus = new fk_Vector(100.0, 0.0, 100.0);
```

ただし、これらの方法だけでは柔軟な制御は困難である。

第二の方法として、fk_Model 型インスタンスを利用するというものである。まずは以下のような操作により、カメラ用の fk_Model 型インスタンスを準備する。

```
    var win = new fk_AppWindow();
    var camera = new fk_Model();

    win.CameraModel = camera;
```

この後、camera 変数に対して 8 章で説明した fk_Model クラスの各種メソッドを用いることで、多様な操作を実現することができる。例として、あるモデルを注視しながら周囲を回転する「オービットカメラ」という効果は、以下のようなコードによって実現できる。

```
    var window = new fk_AppWindow();
    var model = new fk_Model();
    var camera = new fk_Model();
    window.CameraModel = camera;

    while(window.Update())
    {
        camera.GlFocus(model.Position);
```

```
        camera.GlRotateWithVec(model.Position, fk_Axis.Y, 0.01);
    }
```

11.6 座標軸やグリッドの表示

fk_AppWindow には座標軸やグリッドを表示する機能が備わっている。座標軸とは、原点から座標軸方向に描画される線分のことである。また、グリッドとは座標平面上に表示されるメッシュのことである。

座標軸とグリッドの表示は共に ShowGuide() メソッドを用いて行われる。表示したい対象を以下のように並べて指定する。

```
window.ShowGuide(fk_Guide.AXIS_X | fk_Guide.AXIS_Y | fk_Guide.AXIS_Z | fk_Guide.GRID_XZ);
```

ShowGuide() で指定できる項目は以下の表 11.6 の通りである。

表 11.1 座標軸・グリッドの指定項目

fk_Guide.AXIS_X	x 軸
fk_Guide.AXIS_Y	y 軸
fk_Guide.AXIS_Z	z 軸
fk_Guide.GRID_XY	xy 平面上のグリッド
fk_Guide.GRID_YZ	yz 平面上のグリッド
fk_Guide.GRID_XZ	xz 平面上のグリッド

なお、引数を省略した場合は x, y, z 各座標軸と xz 平面グリッドが表示される。

座標軸の長さやグリッドの幅、数などは以下のメソッドやプロパティによって制御可能である。

HideGuide() メソッド

座標軸・グリッドを消去する。

double AxisWidth プロパティ

座標軸の線幅を w に設定する。

double GridWidth プロパティ

グリッドの線幅を w に設定する。

double GuideScale プロパティ

グリッドの幅を s に設定する。

int GuideNum プロパティ

グリッドの分割数を n に設定する。

11.7 デバイス情報の取得

fk_AppWindow 上でのマウスやキーボードの状態を調べるため、fk_AppWindow クラスは様々なメソッドを提供している。ここでは、それらの使用法を説明する。

11.7.1 GetKeyStatus() メソッド

このメソッドは、キーボード上の文字キーが現在押されているかどうかを調べるためのメソッドである。たとえば、'g' というキーが押されているかどうかを調べたければ、

```
if(window.GetKeyStatus('g'))
    :   // キーが押された時の処理を行う。
}
```

という記述を行う。このメソッドは、1 番目の引数にキーを表す文字を代入する。2 番目の引数はどのようなキーの状態を検知するかを設定するもので、以下のような種類がある。

表 11.2 キーの状態設定

fk_Switch.RELEASE	離しっぱなしの状態
fk_Switch.UP	離した瞬間
fk_Switch.DOWN	押した瞬間
fk_Switch.PRESS	押しっぱなしの状態

上記のサンプルプログラムは「押した状態にあるかどうか」を検知するものであるが、「押した瞬間」であるかどうかを検知する場合は「fk_Switch.PRESS」のかわりに「fk_Switch.DOWN」を第 2 引数に入力する。

なお、第 2 引数が fk_Switch.PRESS の場合は省略が可能である。先述のプログラムは、

```
if(window.GetKeyStatus('g'))
    :   // キーが押された時の処理を行う。
}
```

としてもよい。

11.7.2 特殊キーの状態取得

エンターキーやシフトキーなど、文字として表現できないキーについては、GetKeyStatus() メソッドの第 1 引数に fk_Key 型の特定の値を入力すればよい。たとえば、左シフトキーが押されているかどうかを調べたければ、

```
if(window.GetKeyStatus(fk_Key.SHIFT_L)) {
    :   // キーが押されている時の処理を行う。
}
```

といったような記述を行う。特殊キーとメソッドの引数の対応は以下の表 11.7.2 のとおりである。

表 11.3　特殊キーと引数値の対応

引数名	対応特殊キー	引数名	対応特殊キー
fk_Key.SHIFT_R	右シフトキー	fk_Key.CAPS_LOCK	Caps Lock キー
fk_Key.SHIFT_L	左シフトキー	fk_Key.PAGE_UP	Page Up キー
fk_Key.CTRL_R	右コントロールキー	fk_Key.PAGE_DOWN	Page Down キー
fk_Key.CTRL_L	左コントロールキー	fk_Key.HOME	Home キー
fk_Key.ALT_R	右 ALT キー	fk_Key.END	End キー
fk_Key.ALT_L	左 ALT キー	fk_Key.INSERT	Insert キー
fk_Key.ENTER	改行キー	fk_Key.RIGHT	右矢印キー
fk_Key.BACKSPACE	Back Space キー	fk_Key.LEFT	左矢印キー
fk_Key.DELETE	Del キー	fk_Key.UP	上矢印キー
fk_Key.TAB	Tab キー	fk_Key.DOWN	下矢印キー
fk_Key.F1 〜 fk_Key.F12	F1 〜 F12 キー	fk_Key.SPACE	スペースキー

11.7.3　MousePosition プロパティ

このプロパティは、現在のマウスポインタの位置を調べる時に使用する。使い方は、

```
fk_Vector pos;
var window = new fk_AppWindow();
    :
pos = window.MousePosition;
```

とすることで、fk_Vector 型の変数にマウスポインタのウィンドウ座標系による現在位置を得られる。ウィンドウ座標系では、描画領域の左上部分が原点となり、x 成分は右方向、y 成分は下方向に正となり、数値単位はピクセルとなる。

11.7.4　GetMouseStatus() メソッド

このメソッドは、現在マウスボタンが押されているかどうかを調べる時に使用する。引数値として左ボタンが FK_MOUSE1、中ボタンが FK_MOUSE2、右ボタンが FK_MOUSE3 に対応しており、

```
var window = new fk_AppWindow();

if(window.GetMouseStatus(FK_MOUSE1, fk_Switch.PRESS)) {
    :   // 左ボタンが押されている。
    :
}
```

といった様にして現在のボタン状態を調べることができる。このメソッドも他と同様に、fk_AppWindow 上にマウスポインタがない場合は常に false が返ってくる。

11.8　ウィンドウ座標と 3 次元座標の相互変換

3D のアプリケーションを構築する際、ウィンドウ中のある場所が、3 次元空間ではどのような座標になるのかを知りたいということがしばしば見受けられる。あるいは逆に、3 次元空間中の点が実際にウィンドウのどの位置に表示されるのかをプログラム中で参照したいということもよくある。FK でこれを実現する方法として h、fk_AppWindow クラスに GetWindowPosition(), GetProjectPosition() というメソッドが準備されている。以下に、3 次元からウィンドウへの変換、ウィンドウから 3 次元への変換を述べる。

11.8.1　3 次元座標からウィンドウ座標への変換

3 次元空間中のある座標は、fk_AppWindow クラスの GetWindowPosition() というメソッドを用いることで、ウィンドウ中で実際に表示される位置を知ることができる。引数として入力、出力を表す fk_Vector 型の変数を取る。以下に例を示す。

```
var in = new fk_Vector(0.0, 0.0, 0.0);
var out = new fk_Vector(0.0, 0.0, 0.0);
var window = new fk_AppWindow();
    :
    :
var (status, out) = win.GetWindowPosition(in);
```

ここで、in には元となる 3 次元空間の座標を設定しておく。これにより、out の x 成分、y 成分にそれぞれウィンドウ座標が設定される。なお、この場合の out の z 成分には 0 から 1 までのある値が入るようになっており、カメラから遠いほど高い値が設定される。

11.8.2　ウィンドウ座標から 3 次元座標への変換

3 次元→ウィンドウの場合と比べて、ウィンドウ座標から 3 次元座標への変換はやや複雑である。というのも、3 次元座標からウィンドウ座標へ変換する場合は、結果が一意に定まるのであるが、その逆の場合は単にウィンドウ座標だけでは 3 次元空間中の位置が決定しないからである。もう少し具体的に述べると、本来得たい空間中の位置とカメラ位置を結ぶ直線 (以下これを「指定直線」と呼ぶ) が求まるが、その直線上のどこなのかを特定するにはもう 1 つの基準を与えておく必要がある。FK ではこの基準として

- カメラからの距離
- 任意平面

の 2 種類を用意している。

まずカメラからの距離によって指定する方法を紹介する。3 次元空間上の座標を取得するには GetProjectPosition() メソッドを利用する。引数は以下の通りである。

```
(取得成否, 出力) = GetProjectPosition(ウィンドウx座標, ウィンドウy座標, 距離);
```

例えば、以下の例は現在のマウスが指す 3 次元空間の座標を得るプログラムである。このプログラム中ではカメラからの距離を 500 としている。

```
var pos = new fk_Vector(0.0, 0.0, 0.0);
var out = new fk_Vector(0.0, 0.0, 0.0);
var window = new fk_AppWindow();
    :
    :
pos = win.MousePosition;
var (status, out) = win.GetProjectPosition(pos.x, pos.y, 500.0);
```

もう 1 つの方法として、平面を指定する方法がある。前述の指定直線と与えた平面が平行でないならば、その交点を出力することになる。これは、例えば xy 平面上の点や部屋の壁のようなものを想定するような場合に便利である。

まずは平面を作成する必要があるが、これは fk_Plane というクラスの変数を利用する。平面指定の方法として、

- 平面上の任意の 1 点と平面の法線ベクトルを指定する。
- 平面上の (同一直線上にない) 任意の 3 点を指定する。
- 平面上の任意の 1 点と、平面上の互いに平行ではない 2 つのベクトルを指定する。

の 3 種類があり、以下のように指定する。

```
var plane = new fk_Plane(); // 平面を表す変数
    :
    :
//  1点 + 法線ベクトルのパターン
//  pos ... 平面上の任意の1点で、fk_Vector 型
//  norm .. 平面の法線ベクトルで、fk_Vector 型
plane.SetPosNormal(pos, norm);

//  3点のパターン (3点は同一直線上にあってはならない)
//  pos1 ～ pos3 ... 平面上の任意の点で、全て fk_Vector 型
plane.Set3Pos(pos1, pos2, pos3);

//  1点 + 2つのベクトルのパターン
//  pos ... 平面上の任意の1点で、fk_Vector 型
//  uVec .. 平面に平行なベクトル (fk_Vector型)
//  vVec .. 平面に平行なベクトルで、uVec に平行でないもの (fk_Vector 型)
plane.SetPosUVVec(pos, uVec, vVec);
```

これにより、平面が生成できたら、以下の形式で 3 次元空間中の座標を取得することができる。

```
(取得成否, 出力) = GetProjectPosition(ウィンドウx座標成分, ウィンドウy座標成分, 平面);
```

ちなみに、平面と出力変数はアドレス渡しにしておく必要がある。以下の例は、マウス位置が指している場所の xy 平面上の座標を得るサンプルである。

```
var outPos = new fk_Vector(0.0, 0.0, 0.0); // 出力用変数
var win = new fk_AppWindow(); // ウィンドウ変数
var pos = new fk_Vector(0.0, 0.0, 0.0); // マウス座標用変数
var plane = new fk_Plane(); // 平面を表す変数
var planePos = new fk_Vector(0.0, 0.0, 0.0); // 平面生成用変数
var planeNorm = new fk_Vector(0.0, 0.0, 1.0); // 平面生成用変数
    :
    :
// 情報を平面に設定
plane.SetPosNormal(planePos, planeNorm);

// ウィンドウからマウス座標を得る。
pos = win.MousePosition;

// ウィンドウ座標と平面から、3次元空間中の座標を得る。
var (status, outPos) = win.GetProjectPosition(pos.x, pos.y, plane);
```

11.8.3　シーンの設定

もし fk_Scene を使ってシーン管理を行いたい場合は、fk_AppWindow クラスの Scene プロパティによって対象シーンの描画を行うことができる。

```
var window = new fk_AppWindow();
var scene = new fk_Scene();

window.Scene = scene;
```

11.8.4　メッセージ出力

fk_AppWindow には、メッセージを出力する機能として以下のようなメソッドが提供されている。

void PutString(string message)

message の内容をメッセージ出力用ブラウザに出力する。

void ClearBrowser(void)

メッセージ出力用ブラウザの内容を消去する。

以下のプログラムは、PutString() を用いて変数値を出力するサンプルである。

```
int counter;
    :
    :
while(true) {
    window.PutString($"counter value is {counter}");
}
```

また、ここで紹介したメソッド群は fk_AppWindow のメソッドではあるが、fk_AppWindow 型の変数がなくてもメソッド名の前に「`fk_AppWindow.`」を付加することによって、どこでも利用ができる[1)]。以下にサンプルを示す。

```
    string  outString;
    int     value;
            :
            :
    fk_AppWindow.PutString(outString);
    fk_AppWindow.PutString($"value = {value}");
```

1)　これは、これらのメソッドが static 宣言をしているためである。

第 12 章　簡易形状表示システム

11 章では、FK システムでの多彩なウィンドウやデバイス制御機能を紹介したが、中には FK システムで作成した形状を簡単に表示し、様々な角度から閲覧したいという用途に用いたい利用者もいるであろう。そのようなユーザは、fk_AppWindow や fk_Window によって閲覧する様々な機能を構築するのはやや手間である。そこで、FK システムではより簡単に形状を表示する手段として fk_ShapeViewer というクラスを提供している。

12.1　形状表示

利用するには、まず fk_ShapeViewer 型の変数を定義する。

```
var viewer = new fk_ShapeViewer(600, 600);
```

この時点で、多くの GUI が付加したウィンドウが生成される。形状は、4 章で紹介したいずれの種類でも利用できるが、ここでは例として fk_Solid 型及び fk_Sphere 型の変数を準備する。この変数が表す形状を表示するには、SetShape() メソッドを用いる。SetShape() メソッドは二つの引数を取り、最初の引数は立体 ID を表す整数、後ろの引数には形状を表す変数 (のアドレス) を入力する。複数の形状を一度に表示する場合は、ID を変えて入力していくことで実現できる。

```
var viewer = new fk_ShapeViewer(600, 600);
var solid = new fk_IndexFaceSet();
var sphere = new fk_Sphere();

viewer.SetShape(0, solid);
viewer.SetShape(1, sphere);
```

あとは、Draw() メソッドを呼ぶことで形状が描画される。通常は、次のように繰り返し描画を行うことになる。

```
while(viewer.Draw() == true) {
    // もし形状を変形するならば、ここに処理内容を記述する。
}
```

もし終了処理 (ウィンドウが閉じられる、「Quit」がメニューで選択されるなど) が行われた場合、Draw() メソッドは false を返すので、その時点で while ループを抜けることになる。形状変形の様子をアニメーション処理したい場合には、while 文の中に変形処理を記述すればよい。具体的な変形処理のやり方は、5.3 節に解説が記述されている。

12.2　標準機能

この fk_ShapeViewer クラスで生成した形状ブラウザは、次のような機能を GUI によって制御できる。これらの機能は、何もプログラムを記述することなく利用することができる。

- VRML、STL、DXF 各フォーマットファイル入力機能と、VRML ファイル出力機能。

- 表示されている画像をファイルに保存。
- 面画、線画、点画の各 ON/OFF 及び座標軸描画の ON/OFF。
- 光源回転有無の制御。
- 面画のマテリアル及び線画、点画での表示色設定。
- GUI によるヘディング、ピッチ、ロール角制御及び表示倍率、座標軸サイズの制御。
- 右左矢印キーによるヘディング角制御。
- スペースキーを押すことで表示倍率拡大。また、シフトキーを押しながらスペースキーを押すことで表示倍率縮小。
- マウスのドラッギングによる形状の平行移動。

第 13 章　サンプルプログラム

13.1　基本的形状の生成と親子関係

次に掲載するプログラムは、原点付近に 1 個の直方体と 2 本の線分を作成し表示するプログラムである。ただ表示するだけでは面白くないので、線分を直方体の子モデルにし、直方体を回転させると線分も一緒に回転することを試してみる。また、視点も最初は遠方に置いて段々近づけていき、ある程度まで接近したらひねりを加えてみる。

ソースコード 13.1　親子関係

```
using FK_CLI;
using System;

// ウィンドウ生成
var win = new fk_AppWindow();

// ウィンドウサイズ設定
win.Size = new fk_Dimension(600, 600);

// 直方体モデル生成
var blockModel = new fk_Model();
var block = new fk_Block(50.0, 70.0, 40.0);
blockModel.Shape = block;
blockModel.Material = fk_Material.Yellow;
win.Entry(blockModel);

// 線分モデル生成
fk_Vector[] pos = new fk_Vector[4];
pos[0] = new fk_Vector(0.0, 100.0, 0.0);
pos[1] = new fk_Vector(100.0, 0.0, 0.0);
pos[2] = -pos[0];
pos[3] = -pos[1];
fk_Line[] line = new fk_Line[2];
fk_Model[] lineModel = new fk_Model[2];
for (int i = 0; i < 2; i++)
{
    line[i] = new fk_Line();
    line[i].PushLine(pos[2 * i], pos[2 * i + 1]);
    lineModel[i] = new fk_Model();
    lineModel[i].Shape = line[i];
    lineModel[i].Parent = blockModel;
    win.Entry(lineModel[i]);
}
lineModel[0].LineColor = new fk_Color(1.0, 0.0, 0.0);
lineModel[1].LineColor = new fk_Color(0.0, 1.0, 0.0);

// カメラモデル設定
var camera = new fk_Model();
camera.GlMoveTo(0.0, 0.0, 2000.0);
camera.GlFocus(0.0, 0.0, 0.0);
camera.GlUpvec(0.0, 1.0, 0.0);
```

```
42 win.CameraModel = camera;
43 win.Open();
44
45 var origin = new fk_Vector(0.0, 0.0, 0.0);
46
47 var mat = new fk_Material();
48 var color = new fk_Color();
49 for (int count = 0; win.Update() == true; count++)
50 {
51     // カメラ前進
52     camera.GlTranslate(0.0, 0.0, -1.0);
53
54     // ブロックを y 軸中心に回転
55     blockModel.GlRotateWithVec(origin, fk_Axis.Y, Math.PI / 300.0);
56
57     // カメラの注視点を原点に向ける
58     camera.GlFocus(origin);
59
60     // カウンターが1000を上回ったらカメラをz軸中心に回転
61     if (count >= 1000) camera.LoAngle(new fk_Angle(0.0, 0.0, Math.PI / 500.0));
62 }
```

- 1 行目の using 文は、FK システムを使用する場合に必ず記述する必要がある。
- 11 行目以降で直方体の生成を行っている。基本的には、モデルと形状を用意し、モデルの Shape プロパティに形状を設定しておき、モデルに対して各種の設定を行うというスタイルとなる。作成したモデルは 15 行目にあるようにウィンドウに登録を行わないと表示されないので注意が必要である。
- 18 行目では、線分形状の端点となる各点の位置ベクトルを配列として生成している。この時点では pos 内の各変数のインスタンスはまだ確保されていないため、19,20 行目のように new によってインスタンスを生成する必要があることに注意する。
- 29,30 行目ではそれぞれ線分形状と線分モデルを配列として生成している。これも前述した pos と同様、32 行目や 34 行目にあるようにインスタンスを new で用意する必要がある。
- 32 行目では、lineModel[i] の親モデルを blockModel に設定している。親子関係は、このように Parent プロパティに親モデルとなる fk_Model 型インスタンスを設定する方法がもっとも標準的なものである。これにより、各線分モデルは直方体モデルの動きに追従するようになる。しかし、親子関係は表示属性にはなんら影響はしないため、各線分を表示するためには 37 行目にあるようにウィンドウへの登録が必要となる。
- 34, 35 行目で、線分に対して色設定を行っている。どのような形状であっても、線に対して色を設定する場合は LineColor プロパティを用いる。
- 38 〜 43 行目で、カメラ (視点) モデルの設定を行っている。カメラモデルは通常の fk_Model インスタンスを準備すればよく、このサンプルの場合は位置を $(0, 0, 2000)$、方向を原点に向け、ウィンドウの上方向が $(0, 1, 0)$ になるように設定している。ある fk_Model 型インスタンスをカメラモデルとして設定するには、42 行目にあるように fk_AppWindow クラスの CameraModel プロパティに設定を行えばよい。設定後にカメラモデルを動作させれば視点も動的に変化し、任意のタイミングで別のインスタンスに変更することも可能である。
- 49 行目の for 文中にある Update() メソッドは 2 つの働きがある。まず、現在のモデル設定に従って画面の描画を更新する。また、そのタイミングでウィンドウが消去されていないかをチェックし、もし消去されていた場合は false を返し、正常な場合は true を返す。

  ウィンドウは、画面上で「ESC」キーが押されていたり、ウィンドウが OS 内の機能で強制的に閉じられた場合に消去される。
- 55 行目で、直方体 (と子モデルである線分) を Y 軸中心に回転させている。回転角度は、$\pi/300 = 0.6°$ である。ここ

にある「Math.PI」は C# で利用できる円周率を表す。

- 58 行目で、常にカメラの注視点が原点に向くように補正を行っている。
- 61 行目で、count が 1000 を超えた場合に視点にひねりを加えている。

13.2 Boid アルゴリズムによる群集シミュレーション

Boid アルゴリズムとは、群集シミュレーションを実現するための最も基本的なアルゴリズムである。このアルゴリズムでは、群を構成する各エージェントは以下の 3 つの動作規則を持つ。

分離 (Separation): エージェントが、他のエージェントとぶつからないように距離を取る。
整列 (Alignment): エージェントが周囲のエージェントと方向ベクトルと速度ベクトルを合わせる。
結合 (Cohesion): エージェントは、群れの中心方向に集まるようにする。

図 13.1 は各概念を表す概念図であり、左から順に分離、整列、結合を意味している。

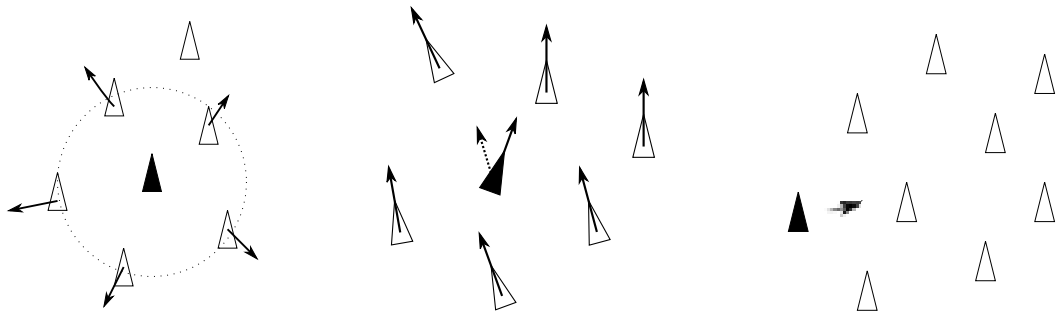

図 13.1 Boid の概念図

これを実現するための計算式は、以下の通りである。なお、n 個のエージェントの識別番号を i $(0, 1, \cdots, n-1)$ とし、それぞれの位置ベクトル、速度ベクトルを $\mathbf{P}_i, \mathbf{V}_i$ とする。

分離:
エージェント i が j から離れたければ、元の速度ベクトルに対し $\overrightarrow{\mathbf{P}_j\mathbf{P}_i} = \mathbf{P}_i - \mathbf{P}_j$ を合わせれば良いので、α をやや小さめの適当な数値として

$$\mathbf{V}_i{}' = \mathbf{V}_i + \alpha \frac{\mathbf{P}_i - \mathbf{P}_j}{|\mathbf{P}_i - \mathbf{P}_j|} \tag{13.1}$$

とすればよい。これを近隣にあるエージェント全てにおいて計算すればよい。近隣エージェントの番号集合を N とすれば、

$$\mathbf{V}_i{}' = \mathbf{V}_i + \alpha \sum_{j \in N} \frac{\mathbf{P}_i - \mathbf{P}_j}{|\mathbf{P}_i - \mathbf{P}_j|} \tag{13.2}$$

となる。

整列:
周囲のエージェントの速度ベクトルの平均を出し、その分を自身の速度ベクトルに足せばよい。数式としては以下の通り。

$$\mathbf{V}_i{}' = \frac{\mathbf{V}_i + \beta \sum_{j \in N} \mathbf{V}_j}{|\mathbf{V}_i + \beta \sum_{j \in N} \mathbf{V}_j|}. \tag{13.3}$$

結合:
群れ全体の重心 $\mathbf{G}$ を算出し、そこに向かうようなベクトルを速度ベクトルに追加する。

$$\mathbf{V}_i{}' = \mathbf{V}_i + \gamma(\frac{\sum \mathbf{P}_i}{n} - \mathbf{P}_i). \tag{13.4}$$

以下、C# によるサンプルプログラムを掲載する。これらついての詳細な解説は割愛するが、各エージェントを表す「Agent」クラスと、群を表す「Boid」クラスを構築することにより、メインループがかなりシンプルになっていることがわかる。また、「S」「A」「C」キーを押すと、それぞれ「分離」「整列」「結合」規則を無効にするようになっているので、各規則がどのようにエージェントの動作に影響しているかがわかるだろう。

ソースコード 13.2　Boid アルゴリズムによる群集シミュレーション

```
using System;
using FK_CLI;

// エージェント用クラス
class Agent
{
    fk_Model model;
    fk_Vector newVec;
    readonly double SPEED = 0.05;   // 速度設定値
    public static readonly double AREA = 15.0; // 移動領域の広さ設定値

    // コンストラクタ
    public Agent()
    {
        model = new fk_Model();
        model.Material = fk_Material.Red;
        model.GlVec(fk_Math.DRand(-1.0, 1.0), fk_Math.DRand(-1.0, 1.0), 0.0);
        model.GlMoveTo(fk_Math.DRand(-AREA, AREA), fk_Math.DRand(-AREA, AREA), 0.0);
    }

    // 位置ベクトル用プロパティ
    public fk_Vector Pos
    {
        get
        {
            return model.Position;
        }
    }

    // 方向ベクトル用プロパティ
    public fk_Vector Vec
    {
        set
        {
            newVec = value;
        }
        get
        {
            return model.Vec;
        }
    }

    // 形状参照プロパティ
    public fk_Shape Shape
    {
        set
        {
            model.Shape = value;
        }
    }

    // ウィンドウへのモデル登録
    public void Entry(fk_AppWindow argWin)
    {
        argWin.Entry(model);
    }

```

```
58      // 前進処理
59      public void Forward()
60      {
61          model.GlVec(newVec);
62          model.LoTranslate(0.0, 0.0, -SPEED);
63      }
64  }
65
66  // 群衆用クラス
67  class Boid
68  {
69      private Agent[] agent; // エージェント用配列
70      private fk_Cone cone;   // 形状（円錐）
71
72      private double paramA, paramB, paramC, paramLA, paramLB;
73
74      public Boid(int argNum)
75      {
76          cone = new fk_Cone(16, 0.4, 1.0);
77          if (argNum < 0) return;
78          agent = new Agent[argNum];
79
80          for (int i = 0; i < argNum; ++i)
81          {
82              agent[i] = new Agent();
83              agent[i].Shape = cone;
84          }
85
86          paramA = 0.2;
87          paramB = 0.02;
88          paramC = 0.01;
89          paramLA = 3.0;
90          paramLB = 5.0;
91      }
92
93      public void SetParam(double argA, double argB, double argC, double argLA, double argLB)
94      {
95          paramA = argA;
96          paramB = argB;
97          paramC = argC;
98          paramLA = argLA;
99          paramLB = argLB;
100     }
101
102     public void SetWindow(fk_AppWindow argWin)
103     {
104         foreach (Agent M in agent)
105         {
106             M.Entry(argWin);
107         }
108     }
109
110     // 群集の更新処理
111     public void Update(bool argSMode, bool argAMode, bool argCMode)
112     {
113         var gVec = new fk_Vector();
114         fk_Vector[] pArray = new fk_Vector[agent.Length];
115         fk_Vector[] vArray = new fk_Vector[agent.Length];
116
117         // 位置ベクトルと速度ベクトルのコピー
118         for (int i = 0; i < agent.Length; i++)
119         {
120             pArray[i] = agent[i].Pos;
121             vArray[i] = agent[i].Vec;
```

```
            gVec += pArray[i];
        }

        // 群集重心の算出
        gVec /= (double)(agent.Length);

        for (int i = 0; i < agent.Length; i++)
        {
            fk_Vector p = new fk_Vector(pArray[i]);
            fk_Vector v = new fk_Vector(vArray[i]);
            for (int j = 0; j < agent.Length; j++)
            {
                if (i == j) continue;
                var diff = p - pArray[j];
                double dist = diff.Dist();

                // 分離 (Separation) 処理
                if (dist < paramLA && argSMode == true)
                {
                    v += paramA * diff / (dist * dist);
                }

                // 整列 (Alignment) 処理
                if (dist < paramLB && argAMode == true)
                {
                    v += paramB * vArray[j];
                }
            }

            // 結合 (Cohesion) 処理
            if (argCMode == true)
            {
                v += paramC * (gVec - pArray[i]);
            }

            // 領域の外側に近づいたら方向修正
            if (Math.Abs(p.x) > Agent.AREA && p.x * v.x > 0.0 && Math.Abs(v.x) > 0.01)
            {
                v.x -= v.x * (Math.Abs(pArray[i].x) - Agent.AREA) * 0.2;
            }

            if (Math.Abs(p.y) > Agent.AREA && p.y * v.y > 0.0 && Math.Abs(v.y) > 0.01)
            {
                v.y -= v.y * (Math.Abs(pArray[i].y) - Agent.AREA) * 0.2;
            }

            v.z = 0.0;
            agent[i].Vec = v;
        }

        foreach (Agent M in agent)
        {
            M.Forward();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        var win = new fk_AppWindow();
        var boid = new Boid(200);

```

```
186         boid.SetWindow(win);
187
188         // ウィンドウ各種設定
189         win.Size = new fk_Dimension(800, 800);
190         win.BGColor = new fk_Color(0.6, 0.7, 0.8);
191         win.ShowGuide(fk_Guide.GRID_XY);
192         win.CameraPos = new fk_Vector(0.0, 0.0, 80.0);
193         win.CameraFocus = new fk_Vector(0.0, 0.0, 0.0);
194         win.TrackBallMode = true;
195
196         win.Open();
197
198         while (win.Update() == true)
199         {
200             // Sキーで「Separate(分離)」を無効に
201             bool sMode = win.GetKeyStatus('S', fk_Switch.RELEASE);
202
203             // Aキーで「Alignment(整列)」を無効に
204             bool aMode = win.GetKeyStatus('A', fk_Switch.RELEASE);
205
206             // Cキーで「Cohesion(結合)」を無効に
207             bool cMode = win.GetKeyStatus('C', fk_Switch.RELEASE);
208
209             // 群集の更新処理
210             boid.Update(sMode, aMode, cMode);
211         }
212     }
213 }
```

13.3 パーティクルアニメーション

パーティクルアニメーションとは、粒子の移動によって気流や水流などを表現する手法である。FK システムでは、パーティクルアニメーションを作成するためのクラスとして fk_Particle 及び fk_ParticleSet クラスを用意している。これらの細かな仕様に関しては 4.13 節に記述してあるが、ここではサンプルプログラムを用いておおまかな利用法を説明する。

fk_ParticleSet クラスは、これまで紹介したクラスとはやや利用手法が異なっている。まず、fk_ParticleSet クラスを継承したクラスを作成し、いくつかの仮想関数に対して再定義を行う。あとは、Shape プロパティを利用して fk_Model に形状として設定したり、fk_ShapeViewer を利用して描画することができる。

ここでは、サンプルとして円柱の周囲を流れる水流のシミュレーションの様子を描画するプログラムを紹介する。

ソースコード 13.3　パーティクルを用いた流体アニメーション

```
1  using System;
2  using FK_CLI;
3
4  static fk_ParticleSet ParticleSetup()
5  {
6      var particle = new fk_ParticleSet();
7      particle.MaxSize = 1000; // パーティクル最大数
8      particle.AllMode = true; // AllMethod() の有効化
9      particle.IndivMode = true; // IndivMethod() の有効化
10
11     // パーティクル生成時処理をラムダ式で設定
12     particle.GenMethod = (P) =>
13     {
14         // 生成時の位置を(ランダムに)設定
15         double y = fk_Math.DRand(-25.0, 25.0);
16         double z = fk_Math.DRand(-25.0, 25.0);
17         P.Position = new fk_Vector(50.0, y, z);
18     };
19
```

```
20      // パーティクル全体処理をラムダ式で設定
21      particle.AllMethod = () =>
22      {
23          for (int i = 0; i < 5; i++)
24          {
25              if (fk_Math.DRand() < 0.3) // 発生確率は 30% (を5回)
26              {
27                  particle.NewParticle(); // パーティクル生成処理
28              }
29          }
30      };
31
32      // パーティクル個別処理をラムダ式で設定
33      particle.IndivMethod = (P) =>
34      {
35          fk_Vector pos, vec, tmp1, tmp2;
36          var water = new fk_Vector(-0.5, 0.0, 0.0);
37          double R = 15.0;
38          double minSpeed = 0.3;
39          double maxSpeed = 0.6;
40          double r;
41
42          //Console.WriteLine("count A {0}", P.ID);
43          pos = P.Position;         // パーティクル位置取得。
44          pos.z = 0.0;
45          r = pos.Dist();           // |p| を r に代入。
46
47          // パーティクルの速度ベクトルを計算
48          tmp1 = water / (r * r * r);
49          tmp2 = ((3.0 * (water * pos)) / (r * r * r * r * r)) * pos;
50          vec = water + ((R * R * R) / 2.0) * (tmp1 - tmp2);
51          P.Velocity = vec;
52
53          // パーティクルの色を計算
54          double speed = vec.Dist();
55          double s = (speed - minSpeed) / (maxSpeed - minSpeed);
56          double t = Math.Clamp(s, 0.0, 1.0);
57          P.Color = fk_Color.GetPseudoColor(t);
58
59          // パーティクルの x 成分が -50 以下になったら消去
60          if (pos.x < -50.0)
61          {
62              particle.RemoveParticle(P);
63          }
64      };
65
66      // Main() にインスタンスを返す。
67      return particle;
68  }
69
70  static fk_AppWindow WindowSetup()
71  {
72      var window = new fk_AppWindow();
73      window.Size = new fk_Dimension(800, 800);
74      window.BGColor = new fk_Color(0.0, 0.0, 0.0);
75      window.TrackBallMode = true;
76      window.FPS = 60;
77      return window;
78  }
79
80  static void ParticleModelSetup(fk_ParticleSet argParticle, fk_AppWindow argWindow)
81  {
82      var model = new fk_Model();
83      model.Shape = argParticle.Shape;
```

```
84      model.DrawMode = fk_Draw.POINT;
85
86      // パーティクルごとに色を変えたい場合の設定
87      model.ElementMode = fk_ElementMode.ELEMENT;
88
89      // パーティクル描画の際の大きさ（ピクセル）
90      model.PointSize = 5.0;
91
92      argWindow.Entry(model);
93  }
94
95  static void PrismSetup(fk_AppWindow argWindow)
96  {
97      var prism = new fk_Prism(40, 15.0, 15.0, 50.0);
98      var prismModel = new fk_Model();
99      prismModel.Shape = prism;
100     prismModel.GlMoveTo(0.0, 0.0, 25.0);
101     prismModel.DrawMode = fk_Draw.FACE | fk_Draw.LINE | fk_Draw.POINT;
102     prismModel.Material = fk_Material.Yellow;
103     prismModel.LineColor = new fk_Color(0.0, 0.0, 1.0);
104     prismModel.PointColor = new fk_Color(0.0, 1.0, 0.0);
105     argWindow.Entry(prismModel);
106 }
107
108 // パーティクルセットの生成
109 var particle = ParticleSetup();
110
111 // ウィンドウ設定
112 var window = WindowSetup();
113
114 // パーティクルモデル設定
115 ParticleModelSetup(particle, window);
116
117 // 内部円柱設定
118 PrismSetup(window);
119
120 // ウィンドウ生成
121 window.Open();
122
123 // メインループ
124 while (window.Update())
125 {
126     particle.Handle(); // パーティクルを 1 ステップ実行する。
127 }
```

- 4 〜 71 行目でパーティクルに対する各種設定を行っている。6 行目でパーティクル用インスタンスを生成し、7 〜 9 行目で各種設定を行っている。
- 7 行目の MaxSize プロパティは fk_ParticleSet クラスのメンバで、パーティクル個数の最大値を設定する。もしパーティクルの個数がこの値と等しくなったときは、NewParticle() メソッドを呼んでもパーティクルは新たに生成されなくなる。
- 8, 9 行目はそれぞれ個別処理、全体処理に対するモード設定である。ここで true に設定しない場合、IndivMethod() や AllMethod() の記述は無視される。
- 12 〜 18 行目では、新たにパーティクルが生成された際の処理を記述する。引数の P に新パーティクルのインスタンスが入っており、これに対して様々な設定を行う。今回のサンプルでは初期位置設定を行っている。

  ここにある GenMethod() や AllMethod(), IndivMethod() は fk_Particle 型を引数に持つ void 型のメソッドか、またはラムダ式で設定を行うことができる。今回のサンプルではラムダ式を用いている。ラムダ式についての詳細は C# に関する一般的な書籍を参照されたい。

- 21 〜 30 行目では、AllMethod() メソッドを再定義している。AllMethod() メソッドには、パーティクル集合全体に対しての処理を記述する。ここではランダムにパーティクルの生成を行っているだけであるが、パーティクル全体に対して一括の処理を記述することもできる。
- 33 〜 64 行目では、IndivMethod() メソッドを再定義している。IndivMethod 関数には、個別のパーティクルに対する処理を記述する。
- IndivMethod() 中では、48 〜 51 行目で速度ベクトルの設定を行っている。中心が原点で、z 軸に平行な半径 R の円柱の周囲を速度 $(-V_x, 0, 0)$ の水流が流れているとする。このとき、各地点 (x, y, z) での水流を表す方程式は以下のようなものである。

$$\frac{\partial}{\partial t}\mathbf{P} = \mathbf{V} + \frac{R^3}{2}\left(\frac{\mathbf{V}}{r^3} - \frac{3\mathbf{V}\cdot\mathbf{P}}{r^5}\mathbf{P}\right). \tag{13.5}$$

  ただし、

$$\mathbf{V} = (-V_x, 0, 0), \qquad \mathbf{P} = (x, y, 0), \qquad r = |\mathbf{P}| \tag{13.6}$$

  である。今回は、$\mathbf{V} = (0.2, 0, 0)$(36 行目の「water」変数)、$R = 15$(37 行目の「R」変数) として算出している。この式から、各パーティクルの速度ベクトルを算出し、51 行目で設定している。
- パーティクルの色は、速度が minSpeed 未満の場合は青、maxSpeed 以上の場合は赤とし、その中間の場合は赤色と青色をブレンドした色となるように設定している。その色値の算出と設定を 54 〜 57 行目で行っている。パーティクルの速度 s が $m < s < M$ を満たすとき、

$$t = \frac{s - m}{M - m} \tag{13.7}$$

  とし、t に対し擬似カラーを求めて色値を決定している。
- 60 〜 63 行目でパーティクル削除判定を行っている。パーティクルが $x = -50$ よりも左へ流れてしまった場合には 62 行目で削除を行っている。
- パーティクル集合を表示するには、83 行目にあるように fk_ParticleSet クラスの Shape プロパティを fk_Model の Shape プロパティに代入する必要がある。
- 126 行目にあるように、Handle() メソッドを用いることでパーティクル全体に 1 ステップ処理が行われる。その際には、設定した速度や加速度にしたがって各パーティクルが移動する。特に再設定しない限り、加速度は処理終了後も保存される。今回のプログラムではパーティクルの加速度は一切設定していないため、IndivMethod() メソッド内の 51 行目の速度設定のみがパーティクルの動作を決定する要因となる。

13.4 音再生

FK では音再生用のクラスとして幾つかのクラスが用意されているが、そのうちユーザーが簡易に用いることを想定したクラスは fk_Sound, fk_BGM の 2 つである。

fk_BGM クラスは音楽 (Back Ground Music) を再生するためのクラスで、基本的な利用はまずコンストラクタで BGM ファイルを指定し、その後 Start() で再生開始、Gain プロパティで音量制御というシンプルなものとなっている。

fk_Sound は効果音 (Sound Effect, 以下「SE」) を再生するためのものである。SE は BGM とは異なり任意のタイミングで再生が開始となること、複数の SE が同時に発生することがあること、リピート再生が行われないことなどの差異があるため、BGM とは別クラスとなっている。このクラスの機能としては、コンストラクタで入力する SE のファイル数、LoadData() で SE ファイルの指定、StartSE() で各 SE の再生開始というものとなっている。

両クラスとも、使用終了時やプログラムの終了時には「StopStatus」プロパティに true を代入する必要がある。これを行わないと、プログラムが正常に終了しない可能性があるので注意すること。

ソースコード 13.4　音の再生

```
1 using System;
2 using FK_CLI;
```

```
3
4  // ウィンドウの各種設定
5  var win = new fk_AppWindow();
6  WindowSetup(win);
7
8  // 立方体の各種設定
9  var blockModel = new fk_Model();
10 BlockSetup(blockModel, win);
11
12 // BGM用変数の作成
13 var bgm = new fk_BGM("epoq.ogg");
14 double volume = 0.5;
15
16 // SE用変数の作成
17 var se = new fk_Sound(2);
18 // 音源読み込み（IDは0番）
19 se.LoadData(0, "MIDTOM2.wav");
20 // 音源読み込み（IDは1番）
21 se.LoadData(1, "SDCRKRM.wav");
22
23 // ウィンドウ表示
24 win.Open();
25
26 // BGM再生スタート
27 bgm.Start();
28
29 // SE出力スタンバイ
30 se.Start();
31
32 var origin = new fk_Vector(0.0, 0.0, 0.0);
33 while (win.Update())
34 {
35     blockModel.GlRotateWithVec(origin, fk_Axis.Y, Math.PI / 360.0);
36
37     // volume値の変更
38     volume = ChangeVolume(volume, win);
39     bgm.Gain = volume;
40
41     // SE再生
42     PlaySE(se, win);
43 }
44
45 // BGM変数とSE変数に終了を指示
46 //（最後にこれをやらないといつまでも再生が止まらずプログラムが終了しません）
47 bgm.StopStatus = true;
48 se.StopStatus = true;
49 bgm.Dispose();
50 se.Dispose();
51
52 /////////// メソッド群
53
54 // ウィンドウ設定メソッド
55 static void WindowSetup(fk_AppWindow argWin)
56 {
57     argWin.CameraPos = new fk_Vector(0.0, 1.0, 20.0);
58     argWin.CameraFocus = new fk_Vector(0.0, 1.0, 0.0);
59     argWin.Size = new fk_Dimension(600, 600);
60     argWin.BGColor = new fk_Color(0.6, 0.7, 0.8);
61     argWin.ShowGuide(fk_Guide.GRID_XZ);
62     argWin.TrackBallMode = true;
63     argWin.FPS = 60;
64 }
65
66 // 立方体モデル設定メソッド
```

```
67  static void BlockSetup(fk_Model argModel, fk_AppWindow argWin)
68  {
69      // 立方体の各種設定
70      var block = new fk_Block(1.0, 1.0, 1.0);
71      argModel.Shape = block;
72      argModel.GlMoveTo(3.0, 3.0, 0.0);
73      argModel.Material = fk_Material.Yellow;
74      argWin.Entry(argModel);
75  }
76
77  // 音量調整メソッド
78  static double ChangeVolume(double argVol, fk_AppWindow argWin)
79  {
80      double tmpV = argVol;
81
82      // 上矢印キーで BGM 音量アップ
83      if (argWin.GetSpecialKeyStatus(fk_Key.UP, fk_Switch.DOWN) == true)
84      {
85          tmpV = Math.Min(tmpV + 0.1, 1.0);
86      }
87
88      // 下矢印キーで BGM 音量ダウン
89      if (argWin.GetSpecialKeyStatus(fk_Key.DOWN, fk_Switch.DOWN) == true)
90      {
91          tmpV = Math.Max(tmpV - 0.1, 0.0);
92      }
93
94      return tmpV;
95  }
96
97  // SE再生(トリガー)メソッド
98  static void PlaySE(fk_Sound argSE, fk_AppWindow argWin)
99  {
100     // Z キーで 0 番の SE を再生開始
101     if (argWin.GetKeyStatus('Z', fk_Switch.DOWN) == true) argSE.StartSound(0);
102
103     // X キーで 1 番の SE を再生開始
104     if (argWin.GetKeyStatus('X', fk_Switch.DOWN) == true) argSE.StartSound(1);
105 }
```

13.5 文字列表示

この節では、第 7.1 節で紹介したスプライトモデルを用いた文字列表示のサンプルを示す。

ソースコード 13.5 文字列表示

```
1   using System;
2   using FK_CLI;
3
4   var window = new fk_AppWindow();
5
6   // 文字盤用変数の生成
7   var sprite = new fk_SpriteModel();
8
9   var block = new fk_Block(1.0, 1.0, 1.0);
10  var model = new fk_Model();
11  var origin = new fk_Vector(0.0, 0.0, 0.0);
12
13  // 文字盤に対するフォントの読み込み
14  if (sprite.InitFont("rm1b.ttf") == false)
15  {
16      Console.WriteLine("Font Error");
```

```
}

// 文字盤の表示位置設定
sprite.SetPositionLT(-330.0, 240.0);

// 文字盤の(等幅)サイズを設定
sprite.Text.MonospaceSize = 12;
window.Entry(sprite);

model.Shape = block;
model.GlMoveTo(0.0, 6.0, 0.0);
model.Material = fk_Material.Yellow;
window.Entry(model);
window.CameraPos = new fk_Vector(0.0, 5.0, 20.0);
window.CameraFocus = new fk_Vector(0.0, 5.0, 0.0);
window.Size = new fk_Dimension(800, 600);
window.BGColor = new fk_Color(0.6, 0.7, 0.8);
window.TrackBallMode = true;
window.Open();
window.ShowGuide(fk_Guide.GRID_XZ);

for (int count = 0;  window.Update(); count++)
{
    // 文字列を文字盤に入力
    sprite.DrawText($"count = {count,4}", true);

    // 文字盤表示位置の再設定
    sprite.SetPositionLT(-330.0, 240.0);
    model.GlRotateWithVec(origin, fk_Axis.Y, Math.PI / 360.0);
}
```

- ここではフォントファイルとして「rm1b.ttf」というファイル名のフォントデータを利用している。フォントファイルは作成したプロジェクト中の Resources フォルダの中に入れておく。
- フォントを設定するには 14 行目にあるように InitFont() メソッドを用いる。
- 23 行目で Text プロパティに設定を行っているが、この Text プロパティは fk_TextImage 型であり、fk_TextImage が持つ様々な設定を変更することで色やフォントサイズなど、様々な設定を変更することができる。詳細はリファレンスマニュアルのクラスの説明を参照のこと。
- 20 行目の SetPositionLT() メソッドは表示位置を指定するものである。この SetPositionLT() は 44 行目でもメインループの度に位置の再設定を行っているが、その理由は文字列を表すテクスチャ画像の幅が変更されたときに再設定を行わないと、表示位置がずれてしまうためである。表示文字列が変更されていない場合や、文字列テクスチャの幅が変わっていないことが保証されている場合は、メインループ中にこのメソッドを呼ぶ必要はない。実際、今回のサンプルでは (等幅設定を行い、描画文字数が変化しないことから) 文字盤の画像幅は終始変動しないため、44 行目をコメントアウトしても正常に動作するが、等幅設定を用いない場合などは文字列の変更の際に幅変動も伴うので、このメソッドを呼ばないと表示が適切になされない場合がある。

13.6 四元数

この節では、第 2.3 節で紹介した四元数 (クォータニオン) を用いて、姿勢補間を行うサンプルプログラムを示す。3DCG のプログラミングでは、ベクトル、行列、オイラー角、四元数といった多くの代数要素を扱う必要があるが、このサンプルプログラムはそれらの利用方法をコンパクトにまとめたものとなっている。

四元数は姿勢を表現する手段として強力な数学手法であるが、四元数の成分を直接扱うことは現実的ではなく、通常はオイラー角を介して制御を行う。このサンプルプログラムでも、まず 2 種類の姿勢を angle1, angle2 という変数で設定 (12,13

行目) してから、それを四元数に変換 (37,38 行目) している。そして、45 行目で球面線形補間を行った四元数を算出し、それを 48 行目でモデルの姿勢として設定している。

ソースコード 13.6　四元数

```
1  using System;
2  using FK_CLI;
3
4  var win = new fk_AppWindow();
5  win.Size = new fk_Dimension(800, 800);
6  var model = new fk_Model();
7  var pointM = new fk_Model();
8  var cone = new fk_Cone(3, 4.0, 15.0);
9  var pos = new fk_Vector(0.0, 0.0, -15.0);
10
11 // オイラー角用変数の生成
12 var angle1 = new fk_Angle(0.0, 0.0, 0.0);
13 var angle2 = new fk_Angle(Math.PI / 2.0, Math.PI / 2.0 - 0.01, 0.0);
14
15 // 四元数用変数の生成
16 var q1 = new fk_Quaternion();
17 var q2 = new fk_Quaternion();
18 fk_Quaternion q;
19
20 // 角錐頂点の軌跡用
21 var point = new fk_Polyline();
22
23 model.Shape = cone;
24 model.Material = fk_Material.Yellow;
25 model.GlAngle(angle1); // モデルのオイラー角を (0, 0, 0) に
26
27 pointM.Shape = point;
28 pointM.LineColor = new fk_Color(1.0, 0.0, 0.0);
29
30 win.BGColor = new fk_Color(0.7, 0.8, 0.9);
31 win.Entry(model);
32 win.Entry(pointM);
33 win.TrackBallMode = true;
34 win.ShowGuide();
35 win.Open();
36
37 q1.Euler = angle1; // q1 にオイラー角 (0, 0, 0) を意味する四元数を設定
38 q2.Euler = angle2; // q2 にオイラー角 (π/2, π/2-0.01, 0) を意味する四元数を設定
39
40 for (double t = 0.0;  win.Update(); t += 0.005)
41 {
42     if (t < 1.0)
43     {
44         // q に q1 と q2 を球面補間した値を設定
45         q = fk_Math.QuatInterSphere(q1, q2, t);
46
47         // モデルの姿勢を q に設定
48         model.GlAngle(q.Euler);
49
50         // 頂点軌跡の追加
51         point.PushVertex(model.Matrix * pos);
52     }
53 }
```

付録 A マテリアル一覧

表 A.1　FK システム中のデフォルトマテリアル一覧

色名	環境反射係数	拡散反射係数	鏡面反射係数	ハイライト
AshGray	(0.2, 0.2, 0.2)	(0.4, 0.4, 0.4)	(0.01, 0.01, 0.01)	(10.0)
BambooGreen	(0.15, 0.28, 0.23)	(0.23, 0.47, 0.19)	(0.37, 0.68, 0.28)	(20.0)
Blue	(0.0, 0.0, 0.0)	(0.0, 0.0, 0.7)	(0.0, 0.0, 0.0)	(0.0)
Brown	(0.2, 0.1, 0.0)	(0.35, 0.15, 0.0)	(0.0, 0.0, 0.0)	(0.0)
BurntTitan	(0.1, 0.07, 0.07)	(0.44, 0.17, 0.1)	(0.6, 0.39, 0.1)	(16.0)
Coral	(0.5, 0.3, 0.4)	(0.9, 0.5, 0.7)	(0.0, 0.0, 0.0)	(0.0)
Cream	(0.0, 0.0, 0.0)	(0.8, 0.7, 0.6)	(0.0, 0.0, 0.0)	(0.0)
Cyan	(0.0, 0.0, 0.0)	(0.0, 0.6, 0.6)	(0.0, 0.0, 0.0)	(0.0)
DarkBlue	(0.1, 0.1, 0.4)	(0.0, 0.0, 0.25)	(0.0, 0.0, 0.0)	(0.0)
DarkGreen	(0.1, 0.4, 0.1)	(0.0, 0.2, 0.0)	(0.0, 0.0, 0.0)	(0.0)
DarkPurple	(0.3, 0.1, 0.3)	(0.3, 0.0, 0.3)	(0.0, 0.0, 0.0)	(0.0)
DarkRed	(0.2, 0.0, 0.0)	(0.4, 0.0, 0.0)	(0.0, 0.0, 0.0)	(0.0)
DarkYellow	(0.0, 0.0, 0.0)	(0.4, 0.3, 0.0)	(0.0, 0.0, 0.0)	(0.0)
DimYellow	(0.18, 0.14, 0.0)	(0.84, 0.86, 0.07)	(0.92, 0.82, 0.49)	(0.0)
Flesh	(0.0, 0.0, 0.0)	(0.8, 0.6, 0.4)	(0.0, 0.0, 0.0)	(0.0)
GlossBlack	(0.0, 0.0, 0.0)	(0.04, 0.04, 0.04)	(0.0, 0.0, 0.0)	(0.0)
GrassGreen	(0.0, 0.1, 0.0)	(0.0, 0.7, 0.0)	(0.47, 0.98, 0.49)	(0.0)
Gray1	(0.0, 0.0, 0.0)	(0.6, 0.6, 0.6)	(0.1, 0.1, 0.1)	(0.0)
Gray2	(0.0, 0.0, 0.0)	(0.2, 0.2, 0.2)	(0.1, 0.1, 0.1)	(0.0)
Green	(0.0, 0.0, 0.0)	(0.0, 0.5, 0.0)	(0.0, 0.0, 0.0)	(0.0)
HolidaySkyBlue	(0.01, 0.22, 0.4)	(0.2, 0.66, 0.92)	(0.47, 0.74, 0.74)	(0.0)
IridescentGreen	(0.04, 0.11, 0.07)	(0.09, 0.39, 0.18)	(0.08, 0.67, 0.1)	(14.0)
Ivory	(0.36, 0.28, 0.18)	(0.56, 0.52, 0.29)	(0.72, 0.45, 0.4)	(33.0)
LavaRed	(0.14, 0.0, 0.0)	(0.62, 0.0, 0.0)	(1.0, 0.46, 0.46)	(18.0)
LightBlue	(0.0, 0.0, 0.0)	(0.4, 0.4, 0.9)	(0.0, 0.0, 0.0)	(0.0)
LightCyan	(0.1, 0.2, 0.2)	(0.0, 0.5, 0.5)	(0.2, 0.2, 0.2)	(60.0)
LightGreen	(0.0, 0.0, 0.0)	(0.5, 0.7, 0.3)	(0.0, 0.0, 0.0)	(0.0)
LightViolet	(0.0, 0.0, 0.0)	(0.5, 0.4, 0.9)	(0.0, 0.0, 0.0)	(0.0)
Lilac	(0.21, 0.09, 0.23)	(0.64, 0.54, 0.6)	(0.4, 0.26, 0.37)	(15.0)
MatBlack	(0.0, 0.0, 0.0)	(0.0, 0.0, 0.0)	(0.0, 0.0, 0.0)	(0.0)
Orange	(0.0, 0.0, 0.0)	(0.8, 0.3, 0.0)	(0.2, 0.2, 0.2)	(0.0)
PaleBlue	(0.0, 0.0, 0.0)	(0.5, 0.7, 0.7)	(0.0, 0.0, 0.0)	(0.0)
PearWhite	(0.32, 0.29, 0.18)	(0.64, 0.61, 0.5)	(0.4, 0.29, 0.17)	(15.0)
Pink	(0.6, 0.2, 0.3)	(0.9, 0.55, 0.55)	(0.0, 0.0, 0.0)	(0.0)
Purple	(0.0, 0.0, 0.0)	(0.7, 0.0, 0.7)	(0.0, 0.0, 0.0)	(0.0)
Red	(0.0, 0.0, 0.0)	(0.7, 0.0, 0.0)	(0.0, 0.0, 0.0)	(0.0)
UltraMarine	(0.01, 0.03, 0.21)	(0.07, 0.12, 0.49)	(0.53, 0.52, 0.91)	(11.0)
Violet	(0.0, 0.0, 0.0)	(0.4, 0.0, 0.8)	(0.0, 0.0, 0.0)	(0.0)
White	(0.0, 0.0, 0.0)	(0.8, 0.8, 0.8)	(0.1, 0.1, 0.1)	(0.0)
Yellow	(0.0, 0.0, 0.0)	(0.8, 0.6, 0.0)	(0.0, 0.0, 0.0)	(0.0)
TrueWhite	(1.0, 1.0, 1.0)	(1.0, 1.0, 1.0)	(0.0, 0.0, 0.0)	(0.0)