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

Version 4.2.10

(Manual Revision 1)

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.3 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、12 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。

4 章では、基本形状の生成法を中心に述べる。さらに、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、9、10 章は、FK に関する大きな 3 つの概念 — すなわち、オブジェクト、シーン、ウィンドウ — に対しての詳細な説明を与えている。これらと形状を含めた 4 つの関係は単純明解な包有関係で、形状はオブジェクトに、オブジェクトはシーンに、ディスプレイリストはウィンドウに設定される。

第 11 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

```
 1: #include <FK/FK.h>
 2:
 3: using namespace FK;
 4:
 5: int main(int, char **)
 6: {
 7:     fk_Block        block(10.0, 20.0, 15.0);
 8:     fk_Model        model;
 9:     fk_AppWindow    window;
10:
11:     // 色パレットの初期化
12:     fk_Material::initDefault();
13:
14:     // モデルに直方体を設定
15:     model.setShape(&block);
16:
17:     // モデルの色を黄色に設定
18:     model.setMaterial(Material::Yellow);
19:
20:     // カメラの位置と方向を設定
21:     window.setCameraPos(0.0, 0.0, 100.0);
22:     window.setCameraFocus(0.0, 0.0, 0.0);
```

```
23:
24:     // ウィンドウにモデルを登録
25:     window.entry(&model);
26:
27:     // ウィンドウのサイズを設定
28:     window.setSize(800, 800);
29:
30:     // ウィンドウを開く
31:     window.open();
32:
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, 0.01);
37:     }
38:     return 0;
39: }
```

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

次に、実際にサンプルプログラムの解説を述べる。

FK でプログラムを記述する場合、まず 1 行目にあるように「`#include <FK/FK.h>`」を追加しておく。その後の 3 行目にある「`using namespace FK;`」は「FK 名前空間の就職を省略する」という意味である。FK のクラスは全て「FK 名前空間」で定義されており、本来であれば例えば fk_Model クラスであれば「FK::fk_Model」と記述する必要があるが、using 構文で「FK::」を省略することが可能となる。名前空間に関する詳細は C++ での解説書を参照してほしい。

以降、本書では全て「`using namespace FK;`」を冒頭で用いていると仮定して解説を進める。この宣言を行いたくない場合は、FK 関係のクラスや定数はすべて頭に「FK::」を付けて読み替えてほしい。

7 〜 9 行目で用意している変数は、それぞれ次のような意味を持っている。

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、5 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。

12 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。

15 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。

18 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。

21,22 行目は、カメラに対して位置と方向を設定している。21 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、22 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。

25 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。

28 行目はウィンドウのサイズをピクセル単位で指定するものである。今回のプログラムではモデルやカメラの設定後に行ったが、変数を作成した時点でいつでも設定が可能である。

31 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。

33 〜 37 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。33 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()」関数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。

36 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」関数は、モデルを回転させる関数である。ここでは、原点を中心に 0.01 ラジアン $\cong 0.57°$ 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_Point 型、つまり空間上の「点」を表す変数を定義するとき、その点の位置を次のようにして設定することができる。

```
fk_Point        pos(10.0, -5.0, 20.0);
```

この例の場合は、点の位置を $(10, -5, 20)$ として設定している。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、7 行目の直方体の部分を

```
fk_Circle       circle(4, 20.0);
```

と変更し、15 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。

```
fk_Model 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 12 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 10 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()」関数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 9 章で詳しく解説する。

1.8 ウィンドウと GUI

FK は、元々 OpenGL と FLTK というシステムを基盤に構築されている。このうち、OpenGL は 3D 描画のための機能であり、FLTK はグラフィカルユーザインターフェース (GUI) を作成するためのツールである。FK は、FLTK との強い親和性を意識して設計されており、FLTK の GUI 機能をそのまま利用することができる。その一例が 12 章の「FLTK を用いた GUI 構築」というサンプルに示されている。

1.9 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、10 章を中心に記述されている。

1.10 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。

$$12 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 2 \rightarrow 12$$

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。

$$2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12$$

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、12 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。

```
fk_Vector vec;
```

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。

```
fk_Vector vec(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。

```
fk_Vector vec1, vec2;

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

set メンバ関数が、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。

```
fk_Vector vec[10];

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

2.1.2 比較演算子

fk_Vector 型は比較演算子として等しいかどうかを判定する '==' と、その否定 '!=' を利用することができる。これらは、通常の int 型の変数の場合と同様に if 文中で利用でき、演算結果が bool 型として返る点でも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 == 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
^	$Vector \hat{} Vector$	ベクトルの外積	fk_Vector

2.1.5 代入演算子

fk_Vector 型は代入演算子 ‘ = ’ を使用することができる。このとき、
実際は中の値のコピーが行われるのであって、ポインタが代入されるわけではない。したがって、

```
fk_Vector vec1;
fk_Vector vec2(1.0, 1.0, 1.0);

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

という記述は vec1.x の値を変更しない。もちろん、

```
fk_Vector *pointer;
fk_Vector vec2(1.0, 1.0, 1.0);

pointer = &vec2;
pointer->x = 3.0;
```

といった記述は可能であり、この場合の vec2.x の値は 3.0 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) を出力する関数として 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();
printf("length = %lf\n", l);
printf("length^2 = %lf\n", 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) する関数として normalize() がある。

```
fk_Vector vec1(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() 関数は 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。

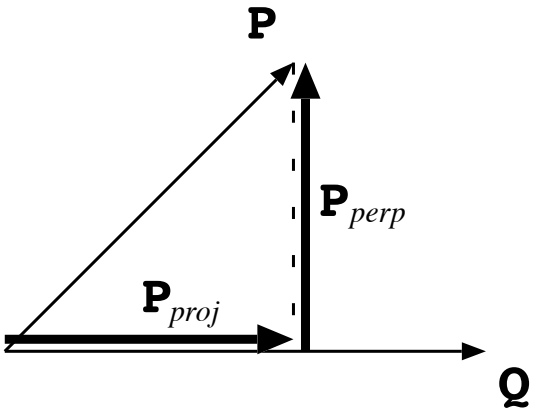

図 2.1　ベクトルの射影

この射影ベクトルを求める方法として、fk_Vector では proj() メンバ関数を利用することができる。

```
fk_Vector   P, Q, P_proj;
        :
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() というメンバ関数を利用する。

```
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。

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。

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。

表 2.3 fk_Matrix の初期値設定用メンバ関数

メンバ関数名と引数	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
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 関数は、2 つ目の引数に x 軸を表す fk_X、y 軸を表す fk_Y、z 軸を表す fk_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。$180° = \pi\, rad$ である。FK システムでは π を表す定義値として fk_Math::PI を提供している。したがって、たとえば 90 度は `fk_Math::PI/2.0` と表せばよい。

makeEuler 関数は引数にそれぞれ順に heading 角、pitch 角、bank 角を与える。単位は makeRot 関数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 比較演算子

fk_Matrix 型も、fk_Vector 型のような '`==`' 演算子と '`!=`' 演算子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 単行演算子

fk_Matrix 型は、単項演算子 '!' 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。

```
fk_Matrix   A, B;
        :
B = !A;
```

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 型の持つ代入演算子は、単純代入演算子として '=' を持っており、その挙動は fk_Vector 型とまったく同様である。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[2)]。

```
fk_Matrix mat;

mat.makeEuler(fk_Math::PI/2.0, fk_Math::PI/4.0, fk_Math::PI/6.0);
printf("mat[1][0] = %lf\n", mat[1][0]);
```

2.2.8　その他のメンバ関数

以下に、fk_Matrix で用いられる主要なメンバ関数を紹介する。

void init(void)
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)

2)　正確に述べると、配列の 1 次元目は行列の各行の先頭のアドレスを返す。

void setRow(int r, fk_HVector 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 setCol(int c, fk_Vector v)
void setCol(int c, fk_HVector v)
列番号が c である列ベクトルを 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)
自身が特異行列であるかどうかを判定する。特異行列とは、逆行列が存在しない行列のことである。特異行列である場合は 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。(つまり、「非正則行列」と「特異行列」はまったく同じ意味になる。) 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。特異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、一応専用のメンバ関数も準備されている。ispos() メンバ関数は同次座標を 1 に設定し、平行移動変換を有効とする。isvec() メンバ関数は同次座標を 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。

```
fk_Vector       v(0.2, 0.5, 1.0);

fk_Quaternion   q1(0.3, 0.4, 2.0, -2.0);
fk_Quaternion   q2(0.5, v);
```

設定用のメンバ関数としては、以下のようなものがある。

void init(void)
初期化のためのメンバ関数で、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)
設定のためのメンバ関数で、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)
設定のためのメンバ関数で、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。
~	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。

$$\begin{aligned} -\mathbf{q} &= -s - xi - yj - zk \\ \overline{\mathbf{q}} &= s - xi - yj - zk \\ \mathbf{q}^{-1} &= \frac{\overline{\mathbf{q}}}{|\mathbf{q}|^2} \end{aligned}$$

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
^	$Quaternion \hat{} Quaternion$	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メンバ関数が用意されている。以下にその仕様を述べる。

void makeEuler(const fk_Angle angle)

angle が示すオイラー角と同義の四元数を設定する。

void makeEuler(double h, double p, double b)
h をヘディング角、p をピッチ角、b をバンク角とするオイラー角と同義の四元数を設定する。

fk_Angle getEuler(void)
同義となるオイラー角を返す。

2.3.8 各種メンバ関数

fk_Quaternion には、これまで挙げた他にも以下のようなメンバ関数がある。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(void)
ノルム $|\mathbf{q}|^2 = s^2 + x^2 + y^2 + z^2$ を返す。

double abs(void)
絶対値 $|\mathbf{q}| = \sqrt{s^2 + x^2 + y^2 + z^2}$ を返す。

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。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。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 $\overline{\mathbf{q}}$ を求め、自身に上書きする。

fk_Matrix conv(void)
自身が表わす回転変換と同義の行列を返す。

2.3.9 補間関数

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 種類の関数が用意されている。

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。

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

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。

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

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。

```
fk_Color col;

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。

```
fk_Color col(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。

```
fk_Color col;

col.setR(0.5);
col.setG(0.6);
col.setB(0.7);
```

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 9 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 9 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 9 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。

```
fk_Material  mat;
fk_Color     amb(0.3, 0.5, 0.8);
fk_Color     dif(0.2, 0.4, 0.9);
fk_Color     emi(0.0, 0.5, 0.3);
```

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。

```
fk_Color     spe(1.0, 0.5, 1.0);

mat.setAlpha(0.5)               // 透明度の設定
mat.setAmbient(amb);            // 環境反射係数の設定
mat.setDiffuse(dif);            // 拡散反射係数の設定
mat.setEmission(emi);           // 放射光係数の設定
mat.setSpecular(spe);           // 鏡面反射係数の設定
mat.setShininess(64.0);         // 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。

```
fk_Material  mat;

mat.setAlpha(0.5);
mat.setAmbient(0.3, 0.5, 0.8);          // 環境反射係数の設定
mat.setDiffuse(0.2, 0.4, 0.9);          // 拡散反射係数の設定
mat.setEmission(0.0, 0.5, 0.3);         // 放射光係数の設定
mat.setSpecular(1.0, 0.5, 1.0);         // 鏡面反射係数の設定
mat.setShininess(64.0);                 // 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、setAmbDiff というメンバ関数を用いることである。これは、setAmbient メンバ関数や setDiffuse 関数と同様の使用法なのだが、関数に代入された値を同時に ambient と diffuse の両方の値に同じく設定する。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。

```
fk_Material     mat1, mat2;

mat1.setAmbDiff(1.0, 1.0, 0.0);         // 黄色いマテリアル
mat2.setAmbDiff(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。

これらの使い方は，以下のように FK::Material 名前空間を利用すればよい。

```
using namespace FK;

model.setMaterial(Material::Yellow);
```

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

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。

```
fk_Polyline poly;
```

pushVertex() 関数を使えば 1 個ずつ頂点を代入していくことができる。

```
        fk_Vector pos;
        fk_Polyline poly;
        int i;

        for(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 関数を用いるとよい。

```
        fk_Vector pos;
        fk_Polyline poly;
        int i;

        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() メンバ関数を用いれば実現できる。

```
        fk_Polyline     Poly;
                :
                :
        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。

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」関数で点を生成し、「setVertex」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。

```
fk_Line 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() 関数を用いて代入することである。

```
fk_Vector vec[2];
fk_Line line;

vec[0].set(1.0, 1.0, 1.0);
vec[1].set(-1.0, 1.0, 1.0);

line.setVertex(vec);
```

このような記述で、line 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 メンバ関数を使用すればよい。

```
fk_Line line;
fk_Vector a, b;

a.set(1.0, 1.0, 1.0);
b.set(-1.0, 1.0, 1.0);

ln.setVertex(0, a);
ln.setVertex(1, b);
```

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。

この場合での setVertex() 関数の最初の引数は、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() メンバ関数を使用する。次のようにすればよい。

```
fk_Line         line;
fk_Vector       vec1, vec2, vecArray[2];


vec1.set(0.0, 1.0, 0.0);
vec2.set(1.0, 0.0, 0.0);
line.pushLine(vec1, vec2);        // 2 つの fk_Vector を使う方法


vecArray[0].set(0.0, 0.0, 1.0);
vecArray[1].set(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_Polygon による多角形の生成は、以下のように std::vector を用いて行う。

```
fk_Polygon poly;
std::vector<fk_Vector> pos;


pos.push_back(fk_Vector(0.0, 10.0, 0.0));
pos.push_back(fk_Vector(-10.0, 0.0, 0.0));
pos.push_back(fk_Vector(10.0, 0.0, 0.0));


poly.setVertex(&pos);
```

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ンバ関数を持っている。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。

```
fk_Circle circ;
```

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。

```
fk_Circle circ(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 メンバ関数で半径を動的に制御することが可能である。

```
fk_Circle circ(4, 5.0);

circ.setRadius(10.0);
```

半径を変更する方法として、他にも setScale 関数がある。これは、半径を実数倍するものである。

```
fk_Circle circ(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 関数がある。引数として分割数を与えることができる。

```
fk_Circle circ;

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。

```
fk_Block 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。初期値を設定することも可能

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。

である。

```
fk_Block block(10.0, 1.0, 40.0);
```

メンバ関数 setSize は、大きさを動的に制御できる。

```
fk_Block 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。

```
fk_Block block

block.setSize(10.0, fk_X);
block.setSize(40.0, fk_Y);
block.setSize(50.0, fk_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、

$$
\begin{aligned}
(x/2, y/2, z/2), \qquad & (-x/2, y/2, z/2), \\
(x/2, -y/2, z/2), \qquad & (-x/2, -y/2, z/2), \\
(x/2, y/2, -z/2), \qquad & (-x/2, y/2, -z/2), \\
(x/2, -y/2, -z/2), \qquad & (-x/2, -y/2, -z/2).
\end{aligned}
$$

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 メンバ関数は、直方体を現状から実数倍するものである。この関数も 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。

```
fk_Block 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。

```
fk_Block block(10.0, 10.0, 10.0);

block.setScale(2.0, fk_X);
block.setScale(3.0, fk_Y);
block.setScale(4.0, fk_X);
```

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。

```
fk_Block 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。

```
fk_Sphere sphere(4, 10.0);
```

初期設定や setRadius()、setDivide()、setScale() といったメンバ関数の利用方法は全て 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。

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。

```
fk_Prism prism(5, 20.0, 30.0, 40.0, false);
```

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。

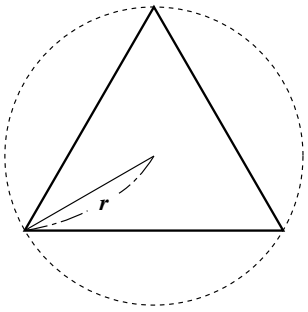

図 4.1 正多角形と外接円半径

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

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。

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。

なお、コンストラクタの最後の引数は「スムースシェーディング」の有無の設定で、true の場合は側面が円柱らしい雰囲気となる。逆に false の場合は各面の境界が明瞭になり、角柱らしい雰囲気となる。

次に述べる関数で、fk_Prism クラスの形状をいつでも動的に変形できる。

void setTopRadius(double r)
上面半径を r に変更する。

void setBottomRadius(double r)
底面半径を r に変更する。

void 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。

```
fk_Cone cone(5, 20.0, 40.0, false);
```

「半径」に関しては前節の fk_Prism 中の解説を参照してほしい。また、最後の引数は fk_Prism と同様にスムースシェーディングの有無の設定である。

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、以下のメンバ関数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。

void setRadius(double r)
底面半径を r に変更する。

void 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 VRML ファイルの取り込み

FK システムでは、形状モデラで作成した立体を取り込みたい場合の手段として VRML 2.0 形式で出力したファイルを読み込む機能が提供されている。

VRML 2.0 形式のファイルを読み込む方法は、以下のようにファイル名を引数にとればよい。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。

```
fk_IndexFaceSet   ifs;
if(ifs.readVRMLFile("sample.wrl", true) == true) {
    fl_alert("File Read Error!");
}
```

この場合、立体のマテリアルは VRML に記されているマテリアルを採用し、fk_Model での変更を受け付けなくなる。もし VRML 中に記述されているマテリアルを無視し、fk_Model でマテリアル制御を行いたい場合は次のように 2 番目の引数を false にすればよい。

```
fk_IndexFaceSet    ifs;
if(ifs.readVRMLFile("sample.wrl", false) == false) {
    fl_alert("File Read Error!");
}
```

4.11.2 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 関数を使用すればよい。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。

```
fk_IndexFaceSet     ifs;
if(ifs.readSTLFile("sample.stl") == false) {
    fl_alert("File Read Error!");
}
```

4.11.3 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。

```
fk_IndexFaceSet    ifs;
if(ifs.readSMFFile("sample.smf") == false) {
    fl_alert("File Read Error!");
}
```

4.11.4 HRC ファイルの取り込み

HRC は、SoftImage 等で使用できるフォーマットである。SoftImage で作成したモデルは、この HRC ファイルに出力することによって FK で読み込むことが可能となる。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。

```
fk_IndexFaceSet   ifs;
if(ifs.readHRCFile("sample.hrc") == false) {
    fl_alert("File Read Error!");
}
```

4.11.5 RDS ファイルの取り込み

RDS は、Ray Dream Studio 形式の略で、多くの 3D モデリングソフトで出力が用意されている。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。

```
fk_IndexFaceSet   ifs;
if(ifs.readRDSFile("sample.rds") == false) {
    fl_alert("File Read Error!");
}
```

4.11.6 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。

```
fk_IndexFaceSet    ifs;
if(ifs.readDXFFile("sample.dxf") == false) {
    fl_alert("File Read Error!");
}
```

4.11.7 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() というメンバ関数を利用する。この関数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);
readMQOFile(string fileName, string objName, int matID, 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。下段の定義中の「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。

これ以降の引数に関してはデフォルト値が設定されており、省略可能である。「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。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。

```
fk_IndexFaceSet    ifs;
if(ifs.readMQOFile("sample.mqo", "obj1") == false) {
    fl_alert("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。

```
fk_IndexFaceSet    ifs;
if(ifs.readMQOFile("sample.mqo", "obj1", 1) == false) {
    fl_alert("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。

4.11.8 MQO データの取り込み

MQO ファイルデータは、4.11.7 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。関数は readMQOData() というもので、ファイル名を示す文字列のかわりに unsigned char 型配列の先頭アドレスを示すポインタを渡す以外は、readMQOFile() 関数と同じである。

```
fk_IndexFaceSet    ifs;
unsigned char      *buffer;

if(ifs.readMQOFile(buffer, "obj1", 1) == false) {
    fl_alert("File Read Error!");
}
```

4.11.9 DirectX (D3DX) ファイルの取り込み

DirectX 形式 (X 形式と呼ばれることもある) のフォーマット (以下「D3DX 形式」) を持つファイルを読み込むには、readD3DXFile() というメンバ関数を利用する。

MQO ファイルの場合と同様に、1 番目の引数にファイル名文字列、2 番目の引数にファイル中のオブジェクト名文字列を指定する。ただし、X 形式のファイルではオブジェクト名がファイル中に指定されていない場合もある。その場合は 2 番目の引数に空文字列「""」を入れる。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。以下のプログラムは、ファイル名「sample.x」、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。

```
fk_IndexFaceSet    ifs;
if(ifs.readD3DXFile("sample.x", "obj1") == false) {
    fl_alert("File Read Error!");
}
```

readD3DXFile() 関数は、readMQOFile() 関数と同様に 3 番目の引数としてマテリアル番号を指定することができる。また、マテリアル番号として -1 を指定した場合に全ての面を入力する点も同様である。

なお、D3DX ファイルに設定してあるテクスチャも読み込みたい場合は、6.1.3 節の fk_IFSTexture を利用することで可能である。

4.11.10 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 4 種類の情報取得メンバ関数が用意されている。

int getPosSize(void)
形状の頂点数を返す。

int getFaceSize(void)
形状の面数を返す。

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。

vector<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。また、返り値の vector<int> は STL と呼ばれる C++ の機能によるもので、詳細は第 5 章を参照してほしい。この中に、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、中身が空状態のオブジェクトが返される。

また、以下の 3 種類の方法で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。

bool moveVPosition(int vID, double *p)

インデックスが vID である頂点の位置ベクトルを、`(p[0], p[1], p[2])` に変更する。もし vID に対応する頂点が存在しなかった場合、false を返す。

4.11.11 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。それぞれの出力関数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。

光源の定義時に、初期値として光源の種類を指定する。

```
fk_Light parallel(fk_LightType::PARALLEL);
fk_Light point(fk_LightType::POINT);
fk_Light spot(fk_LightType::SPOT);
```

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。指定のなかった場合は、平行光源として定義される。

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() メンバ関数を使用する。

```
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。

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

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 setSpotCutOff() と setSpotExponent() というメンバ関数で設定する。

```
spot.setSpotCutOff(fk_Math::PI/6.0);
spot.setSpotExponent(0.00001);
```

setSpotCutOff() の引数は弧度法による角度を入力する。fk_Math::PI は円周率を表すので、例の場合は $\pi/6 = 30^\circ$ 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 12 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、次のようなメンバ関数が用意されている。

void init(void)
パーティクルを初期化する。

int getID(void)
パーティクルの ID を取得する。

unsigned int getCount(void)
パーティクルの年齢を取得する。

fk_Vector getPosition(void)
パーティクルの現在位置を取得する。

void setPosition(fk_Vector pos)
void setPosition(double x, double y, double z)
パーティクルの位置を pos または (x,y,z) に設定する。

fk_Vector getVelocity(void)
パーティクルの速度ベクトルを取得する。この値が有効なのは、setVelocity() 関数か setAccel() 関数を利用したときのみである。

void setVelocity(fk_Vector vel)
void setVelocity(double x, double y, double y)

パーティクルの速度ベクトルを vel または (x,y,z) に設定する。

fk_Vector getAccel(void)
パーティクルの加速度ベクトルを取得する。この値が有効なのは、setAccel() 関数を利用したときのみである。

void setAccel(fk_Vector acc)
void setAccel(double x, double y, double z)
パーティクルの加速度ベクトルを acc または (x,y,z) に設定する。

int getColorID(void)　パーティクルの色 ID を取得する。

void setColorID(int)
パーティクルの色 ID を設定する。

bool getDrawMode(void)
現在の描画状態を取得する。

void setDrawMode(bool flag)
描画状態を設定する。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仮想関数を上書きする形で利用する。まず、上書きすることになる仮想関数を紹介する。

4.13.2.1 fk_ParticleSet クラスの仮想関数

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オブジェクトが入っている。

4.13.2.2 fk_ParticleSet クラスの通常のメンバ関数

また、fk_ParticleSet クラスは他にも以下のようなメンバ関数を持っている。

void setAllMode(bool flag)
allMethod() 関数による処理の有効化/無効化を設定する。

void setIndivMode(bool flag)
indivMethod() 関数による処理の有効化/無効化を設定する。

void handle(void)
実際に処理を 1 ステップ実行する。

fk_Shape * getShape(void)
パーティクルを表す fk_Shape オブジェクトを返す。

fk_Particle * newParticle(void)
fk_Particle * newParticle(fk_Vector pos)
fk_Particle * newParticle(double x, double y, double z)
パーティクルを新たに生成する。初期位置を引数で設定することも可能である。新たに生成されたパーティクルオブジェクトを返す。

bool removeParticle(fk_Particle *p)
bool removeParticle(int)
パーティクルを消去する。引数として、パーティクルオブジェクトそのものと ID の 2 種類がある。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。

unsigned int getCount(void)
パーティクル集合の年齢を取得する。

unsigned int getParticleNum(void)
現状でのパーティクル個数を取得する。

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 nullptr を返す。

fk_Particle * getNextParticle(fk_Particle *)
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 nullptr 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、nullptr 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。

例えば、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 MyParticle::allMethod(void)
{
    fk_Particle   *p;
    fk_Vector     vec(0.0, 0.0, 0.0);
    p = getNextParticle(nullptr);
    while(p != nullptr) {
        vec += p->getPosition();
        p = getNextParticle(p);
    }
    vec /= double(getParticleNum());
}
```

void setMaxSize(unsigned int)
パーティクルの最大個数を設定する。パーティクル個数が設定した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。

4.14 D3DX 形式中のアニメーション動作

fk_IndexFaceSet クラスや、第 6.1.3 節で後述する fk_IFSTexture で D3DX 形式のファイルを入力したとき、ファイル中にアニメーションデータがある場合は、動的にアニメーション変形を行うことができる。アニメーションを実現するには、setAnimationTime() 関数を用いる。引数として、時間を表わす数値を double 型で入力する。以下のプログラムは、アニメーション動作を表示するサンプルである。

```
fk_IndexFaceSet     ifs;
double              timeCount;

timeCount = 0.0;
while(true) {
        :
        :  // 描画処理
        :
    ifs.setAnimationTime(timeCount);
    timeCount += 10.0;
}
```

上記では、1 回の描画につきアニメーション時間を 10 ずつ増加させている。fk_IndexFaceSet を例にしているが、fk_IFSTexture の場合でもまったく同様の方法でアニメーション動作ができる。

4.15 BVH 形式のモーション再生

前節で述べた D3DX 形式のアニメーションデータの代わりに、BVH 形式で記述されたモーションデータを利用することができる。D3DX 形式の形状データにはボーン情報が記述されており、そのボーン名と BVH 形式のデータ側のボーン名を合わせておくことによって、対応したボーンの動きが制御できるようになっている。
BVH 形式のデータを利用する際には fk_BVHMotion というクラスを用いる。まず、fk_BVHMotion クラスの変数を用意し、その変数に利用したい BVH 形式のデータを入力する。その変数を、D3DX 形式を入力した fk_IndexFaceSet か、fk_IFSTexture の変数に対して setBVHMotion() 関数を用いて割り当てる、という流れで利用する。以下にその利用例を示す。

```
fk_IndexFaceSet     ifs;
fk_BVHMotion        bvh;

// 先に D3DX 形式の形状データを読み込んでおく
if(ifs.readD3DXFile("sample.x", "obj1") == false) {
    fl_alert("D3DX File Read Error!");
}
// BVH 形式のモーションデータを読み込む
if(bvh.readBVHFile("sample.bvh") == false) {
    fl_alert("BVH File Read Error!");
}
// 形状データ側にモーションデータをセットする
ifs.setBVHMotion(&bvh);
```

モーションデータをセットした後は、D3DX 形式のアニメーションと同様に、setAnimationTime() 関数でアニメーション再生を制御できる。複数の BVH データをあらかじめ読み込んでおき、setBVHMotion() 関数で再生したいモーションを切り替える、といった利用方法が可能である。

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。

例として次のような三角錐を作成してみる。

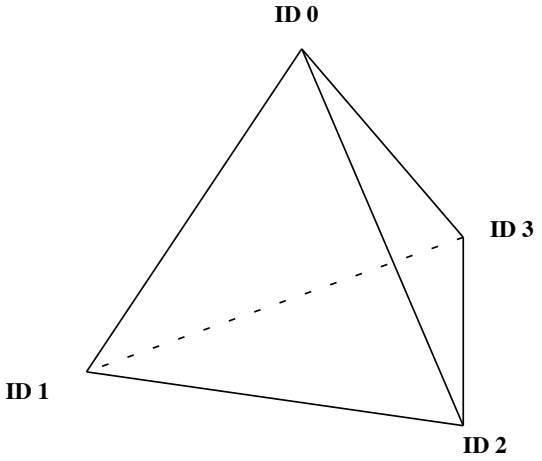

図 5.1　三角錐と各頂点 ID

ここで、それぞれの頂点の位置ベクトルは以下のようなものと想定する。

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 枚の面はそれぞれ次のような頂点を結ぶことで構成されていることが、図を参照することで確認できる。

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。

```
fk_IndexFaceSet     ifs;
fk_Vector           pos[4];
int                 IFSet[3*4];

pos[0].set(0.0, 10.0, 0.0);
pos[1].set(-10.0, -10.0, 10.0);
pos[2].set(10.0, -10.0, 10.0);
pos[3].set(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。

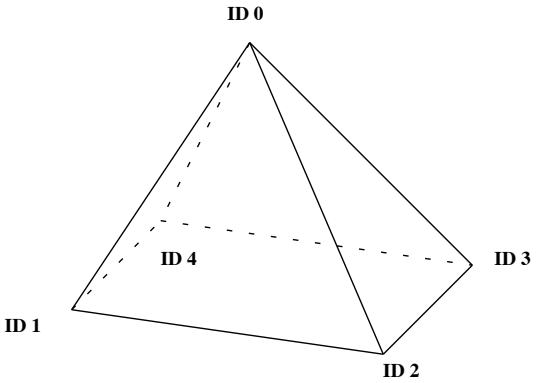

図 5.2 四角錐と各頂点 ID

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ようになる。

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。

```
fk_IndexFaceSet shape;
fk_Vector pos;
std::vector<fk_Vector> posArray;
std::vector<int> polygon;
std::vector< std::vector<int> > IFSet;

pos.set(0.0, 10.0, 0.0); posArray.push_back(pos);
pos.set(-10.0, -10.0, 10.0); posArray.push_back(pos);
pos.set(10.0, -10.0, 10.0); posArray.push_back(pos);
pos.set(10.0, -10.0, -10.0); posArray.push_back(pos);
pos.set(-10.0, -10.0, -10.0); posArray.push_back(pos);
// 1 枚目
polygon.clear();
polygon.push_back(0);
polygon.push_back(1);
polygon.push_back(2);
IFSet.push_back(polygon);
// 2 枚目
polygon.clear();
polygon.push_back(0);
polygon.push_back(2);
polygon.push_back(3);
IFSet.push_back(polygon);
// 3 枚目
polygon.clear();
polygon.push_back(0);
polygon.push_back(3);
polygon.push_back(4);
IFSet.push_back(polygon);
// 4 枚目
polygon.clear();
polygon.push_back(0);
polygon.push_back(4);
polygon.push_back(1);
IFSet.push_back(polygon);
```

```
        // 5 枚目
        polygon.clear();
        polygon.push_back(4);
        polygon.push_back(3);
        polygon.push_back(2);
        polygon.push_back(1);
        IFSet.push_back(polygon);

        shape.makeIFSet(&IFSet, &posArray);
```

上記のプログラム中では、「std::vector<***>」という形式で定義された変数が 3 個登場する。ここではこの形式 (STL と呼ばれる C++ の機能) の詳細は解説しないが、これを用いて次のようにして形状を定義していくことができる。

1. まず、std::vector<fk_Vector> 型の変数 (例では posArray) に対し、push_back 関数で頂点の位置ベクトルを次々に与えていく。
2. polygon を std::vector<int> 型の変数、IFSet を std::vector< vector<int> > 型の変数として、次のような形式で面を定義していく。

```
polygon.clear();
polygon.push_back(頂点 ID);
polygon.push_back(頂点 ID);
        :
polygon.push_back(頂点 ID);
IFSet.push_back(polygon);
```

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。

```
        fk_IndexFaceSet shape;
        fk_Vector       pos;
                :
                :
        pos.set(0.0, 1.0, 2.0);
        shape.moveVPosition(2, pos);
        shape.moveVPosition(3, 1.5, 2.5, 3.5);
```

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。

```
fk_RectTexture    texture;
```

画像ファイルの入力は、Windows Bitmap 形式の場合 readBMP() 関数を用いる。この関数は、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) {
    fl_alert("Image File Read Error!");
}
```

PNG 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() 関数に置き換える。

また、テクスチャの 3 次元空間上での大きさの指定には setTextureSize() 関数を用いる。

```
texture.setTextureSize(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() 関数を利用する。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.setTextureSize(100.0, 100.0);
texture.setRepeatMode(true);
texture.setRepeatParam(5.0, 10.0);
```

setRepeatMode 関数はリピートモードを用いるかどうかを設定するメンバ関数で、引数に true を代入するとリピートモードとなる。次の setRepeatParam メンバ関数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。

```
fk_TriTexture    texture;
texture.readBMP("samp.bmp");
```

fk_TriTexture の場合も、fk_RectTexture と同様に readBMP() を readPNG() に置き換えることで、PNG 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() 関数を利用する。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() 関数を利用する。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() 関数は、fk_Vector 型の変数を引数に持たせることも可能である。

```
fk_Vector    vec(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 ファイルと、D3DX ファイルの 2 種類のデータからの入力が可能となっている。MQO ファイルは readMQOFile() 関数、D3DX ファイルは readD3DXFile() 関数で形状データを入力することができる。なお、アニメーションに関しての機能が 4.14 節に記述してあるので、そちらも合わせて参照してほしい。

readMQOFile() 関数の引数構成は、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。

readD3DXFile() 関数の仕様に関しては、4.11.9 節での内容と同じであるので、そちらを参照してほしい。また、4.11.8 節と同様の用途として、readMQODataO 関数も利用できる。引数の仕様は最初の引数が unsigned char のポインタ型になる以外は上記 readMQOFile() 関数とと同様である。

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。

```
fk_IFSTexture  texture;

if(texture.readBMP("sample.bmp") == false) {
    fl_alert("Image File Read Error!");
}

if(texture.readMQOFile("sample.mqo", "obj1") == false) {
    fl_alert("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ンバ関数として、以下のようなものがある。

bool init(void)

テクスチャデータ及び形状データの初期化を行う。

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。

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 * getIFS(void)

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。この関数は、その変数へのポインタを返すものである。これを利用すると、頂点の移動なども 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 setTriNum() 関数で指定する。その後、fk_TriTexture と同様に setTextureCoord() 関数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、それぞれの関数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。

```
        fk_MeshTexture  texture;


        texture.setTriNum(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() 関数を利用する。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) {
            fl_alert("Image File Read Error!");
        }


        if(texture.readMQOFile("sample.mqo", "obj1") == false) {
            fl_alert("Shape 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 関数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。

```
fk_MeshTexture  texture;
fk_IndexFaceSet ifset;


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 setTexRendMode() 関数で設定を行えばよい。

```
fk_RectTexture  texture;
        :
        :
texture.setTexRendMode(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 setImage() というメンバ関数が用意されており、fk_Image クラスの変数をこのメンバ関数で入力することによって、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。

```
int             i, j;
fk_RectTexture  texture;
fk_Image        image;

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

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

```
    }

    // テクスチャに色を設定
    texture.setImage(&image);
```

fk_Image クラスのメンバ関数を以下に羅列する。

void init(void)
画像情報を初期化する。

bool readBMP(const 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。成功すれば true を、失敗すれば false を返す。

bool readPNG(const 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。成功すれば true を、失敗すれば false を返す。

bool readJPG(const 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。成功すれば true を、失敗すれば false を返す。

bool readBMPData(fk_ImType *buffer)
buffer に Windows Bitmap 形式のデータが格納されているという仮定のもと、readBMP() 関数と同様の挙動を行う。

bool readPNGData(fk_ImType *buffer)
buffer に PNG 形式のデータが格納されているという仮定のもと、readPNG() 関数と同様の挙動を行う。

bool writeBMP(const 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。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(const 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。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(const 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。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。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。

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。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。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。

int getWidth(void)
画像の横幅を int 型で返す。

int getHeight(void)
画像の縦幅を int 型で返す。

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。

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。

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。

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。

fk_Color getColor(int x, int y)
(x, y) の位置にある画素の色要素を fk_Color 型で返す。

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。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。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。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。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。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。成功すれば true を、失敗すれば false を返す。

bool setColor(int x, int y, const fk_Color &col)

(x, y) の位置にある画素に対し、色要素を col が表す色に設定する。成功すれば true を、失敗すれば false を返す。

void fillColor(const 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。

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。

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。

```
fk_SpriteModel  sprite;
```

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() メンバ関数を使って設定する。この関数は、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。(エラー表示には fl_alert 関数を用いている。)

---

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。

```
if(sprite.initFont("sample.ttf") == false) {
    fl_alert("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、setSpriteSize() 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 メンバである「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。

```
fk_SpriteModel     sprite;

sprite.text.setForeColor(1.0, 1.0, 1.0);
sprite.text.setBackColor(0.0, 0.0, 0.0);
```

この「text」メンバは 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() 関数によってスプライトを登録する。

```
fk_AppWindow       window;
fk_SpriteModel     sprite;

window.entry(&sprite);
```

fk_Scene に対して登録を行う場合は、fk_SpriteModel 型の entryFirst() 関数を用いる。引数には fk_Scene 型のインスタンスだけでなく、表示を想定する fk_Window 型のインスタンスも必要となる。

```
fk_Scene            scene;
fk_Window           window;
fk_SpriteModel      sprite;

sprite.entryFirst(&window, &scene);
```

7.1.4 文字列設定

表示する文字列の設定は、drawText() 関数を用いて行う。

```
fk_SpriteModel      sprite;

sprite.drawText("Sample");
```

引数は std::string 型なので、結果として std::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。

```
fk_SpriteModel      sprite;
int                 score = 100;

sprite.drawText("SCORE = " + std::to_string(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() 関数を用いる。

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 メンバは 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 setImage() メンバ関数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。

```
fk_TextImage    textImage;
fk_RectTexture  texture;


texture.setImage(&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 のメンバ関数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 クラスのメンバ関数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

void setDPI(int dpi)

void setPTSize(int ptsize)

setDPI は文字列の解像度を設定し、setPTSize は文字の大きさを設定する。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。

void setMonospaceMode(bool flg)
文字を等幅で表示するかどうかを設定する。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。

void setMonospaceSize(int size)
文字を等幅で表示する場合の、文字幅を設定する。デフォルトでは 0 に設定されているので、この関数で幅を設定しないと表示自体が行われない。

void setBoldStrength(int strength)
文字の太さを数値に応じて太くする。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。

void setSmoothMode(bool flg)
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。デフォルトでは true に設定されている。

void setShadowMode(bool flg)
影付き効果を行うかどうかを設定する。デフォルトでは false に設定されている。

void setForeColor(fk_Color col)
void setForeColor(float r, float g, float b, float a)
void setForeColor(double r, double g, double b, double a)
文字列テクスチャの文字色を指定する。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。

void setBackColor(fk_Color col)
void setBackColor(float r, float g, float b, float a)
void setBackColor(double r, double g, double b, double a)
文字列テクスチャの背景色を指定する。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。

void setShadowColor(fk_Color col)
void setShadowColor(float r, float g, float b, float a)
void setShadowColor(double r, double g, double b, double a)
影付き効果の影の色を指定する。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。

void setShadowOffset(int x, int y)
影付き効果の、影の相対位置を指定する。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。

void setCacheMode(bool mode)
文字画像のキャッシュを保持するかどうかを設定する。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(void)

setCacheMode 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。この関数は 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

void setAlign(fk_TextAlign align)

テキストのアライメントを設定する。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。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、setCharSkip や setLineSkip と同様に、精細度に依存するものである。

void setCharSkip(int skip)

文字同士の横方向の間にある空白の幅を設定する。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 skip が表す数値の 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。

void setLineSkip(int skip)

文字同士の縦方向の間にある空白の高さを設定する。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、setCharSkip() と同様のことが言える。

void setSpaceLineSkip(int)

空行が入っていた場合、その空行の高さを指定する。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、setCharSkip() と同様のことが言える。

void setMinLineWidth(int width)

通常、画像の横幅はもっとも横幅が長い行と同一となる。この関数は、生成される画像の横幅の最小値を width に設定する。生成される画像の幅が width 以内であった場合、強制的に width に補正される。

7.2.3.3 文字送りに関する設定

void setSendingMode(fk_TextSendingMode mode)

文字送り (7.2.8 節を参照のこと) のモードを設定する。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。

```
fk_UniStr       str;
        :
        :
str.convert("サンプルの文字列です", fk_StringCode::SJIS);
```

このように、convert メンバ関数を用いて設定する。1 番目の引数には文字列を、2 番目の引数にプログラムコードの文字コードに対応して以下のように設定する。

表 7.3　文字コード対応表

JIS コード	fk_StringCode::JIS
ShiftJIS コード	fk_StringCode::SJIS
EUC コード	fk_StringCode::EUC
ユニコード (UTF8)	fk_StringCode::UTF8
ユニコード (UTF16)	fk_StringCode::UTF16

文字列設定は、convert 関数の他に printf 関数がある。これは、いわゆる標準の printf() 関数とほぼ同一の機能を持つ物で、変数の値や計算結果などを書式付きで文字列に設定することができる。ただし、第 1 引数に前述した文字コードが入る点だけが異なっている。以下のコードは、str[0]〜str[9] に「0 です」〜「9 です」という文字列を格納する。

```
fk_UniStr     str[10];


for(int i = 0; i < 10; i++) {
    str[i].printf(fk_StringCode::SJIS, "%d です", i);
}
```

fk_UniStr 型変数に格納した文字列を fk_TextImage に設定するには、loadUniStr() メンバ関数を用いる。

```
fk_UniStr       str;
fk_TextImage    image;
        :
        :
str.convert("サンプルの文字列です", fk_StringCode::SJIS);
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() メンバ関数を用いて入力する。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_StringCode::SJIS);
```

loadStrFile() 関数の 2 番目の引数は、テキストの文字コードによって 7.2.4 節の表 7.3 に対応した値を入力する。

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、様々な情報を得るためメンバ関数として以下のようなものが提供されている。

int getLineNum(void)
読み込んだ文字列の行数を返す。

int getAllCharNum(void)
文字列全体の文字数を返す。

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。

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。

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。

int getMaxLineWidth(void)
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返す。

int getMaxLineHeight(void)
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返す。

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。

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。

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」。
- ソースコードの文字コードは ShiftJIS コード。
- 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。

```
        fk_TextImage            textImage;
        fk_RectTexture          texture;
        fk_Model                model;
        fk_UniStr               str;

        texture.setImage(&textImage);

        if(textImage.initFont("fontsample.ttf") == false) {
            fl_alert("Font Init Error!");
        }
        textImage.setDPI(72);
        textImage.setPTSize(72);
        textImage.setShadowMode(true);
        textImage.setLineSkip(20);
        textImage.setForeColor(0.5, 1.0, 0.8, 1.0);
        textImage.setBackColor(0.2, 0.7, 0.8, 0.3);
        textImage.setShadowColor(0.0, 0.0, 0.0, 1.0);
        textImage.setShadowOffset(5, 5);
        textImage.setAlign(fk_TextAlign::CENTER);

        str.convert("サンプルです。", fk_StringCode::SJIS);
```

```
        textImage.loadUniStr(&str);


        model.setShape(&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関数は、簡単にまとめると以下のとおりである。

setSendingMode()	文字送りモード設定
loadUniStr()	新規文字列設定
loadStrFile()	新規文字列をファイルから読み込み
send()	文字送り
finish()	全文字出力
clear()	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 setSendingMode() 関数を用いる。ここで fk_TextSendingMode::CHAR または fk_TextSendingMode::LINE が設定されていた場合、loadUniStr() 関数や loadStrFile() 関数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() 関数を呼んだ場合、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() 関数は、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() 関数は、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、setCacheMode() 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、10 章を参照のこと。)

```
        fk_AppWindow    window;
        fk_TextImage    textImage;
        fk_UniStr       str[10];
        int             loopCount, strCount;
                :
                :
        textImage.setSendingMode(fk_TextSendingMode::CHAR);
        textImage.setCacheMode(true);
        textImage.loadUniStr(&str[0]);
```

```
        loopCount = 1;
        strCount = 1;
        while(true) {
                        :
            if(window.getKeyStatus('c') == true) {
                // 「c」キーを押した場合
                textImage.clear();
            } else if(window.getKeyStatus('f') == 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。

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

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

        model.setShape(&line);
```

つまり前章で示した形状を持つオブジェクトのポインタを、setShape() メンバ関数の引数として与えてやればよい。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。

setShape() 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 setShape() によって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。

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

        model.setShape(&block);
                :
                :                   // 様々な処理が行われている。
                :
```

```
        block.setSize(100.0, fk_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。

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

        for(i = 0; i < 4; i++) {
                model[i].setShape(&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 setShape() 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。

ただし、この方式には落し穴がある。次のようなプログラムは、一見正常に動作するようかのように見えるが、実はそうではない。

```
        main()
        {
                fk_Model *model;
                model = retBlockModel(100.0, 100.0, 100.0)
                delete model;
        }

        fk_Model *retBlockModel(double x, double y, double z)
        {
                fk_Block block(x, y, z);
                fk_Model *model = new fk_Model;

                model->setShape(&block);

                return model;
        }
```

なぜならば、retBlockModel 関数が終了した時点で関数内の block オブジェクトも自動的に消滅してしまうからである。このとき、model のもつ形状ポインタは行き場を失ってしまう。しかも、コンパイラが事前にチェックすることは不可能なため、複雑なプログラムになってくると無意識にこのようなコーディングをしてしまい、泥沼にはまりかねない。実際、コンパイラがどのような挙動を起こすのかは予想できない。FK システムでは、多くの部分でこのような設計を行っているので、オブジェクトの存在の管理には常に気を配らなければならない。

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 setMaterial というメンバ関数を用いるが、引数には 3 章で述べた fk_Material 型のオブジェクトを用いる。

```
fk_Model model;
fk_Material material;
        :                       // この部分で material を
        :                       // 作成しておく。
model.setMaterial(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。

```
model.setMaterial(Material::Green);
```

このとき、(setShape 関数とは異なり) 値は実際に model の中に確保されたメモリー内に全てコピーされるので、マテリアルオブジェクトが途中で消滅してしまっても問題はない。

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、setDrawMode メンバ関数を用いることで実現することができる。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。

```
fk_Sphere       Sphere(4, 10.0);
fk_Model        SphereModel;


SphereModel.setShape(&Sphere);
SphereModel.setDrawMode(fk_Draw::LINE);
```

現在選択できる描画モードは、以下の通りである。

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	テクスチャを描画する。
fk_Draw::GEOM_POINT	曲線や曲面の点描画を行う。
fk_Draw::GEOM_LINE	曲線や曲面グリッド線を描画する。
fk_Draw::GEOM_FACE	曲面を描画する。

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。

```
fk_Sphere       Sphere(4, 10.0);
fk_Model        Solid;
fk_Scene        Scene;

Solid.setShape(&Sphere);
Solid.setDrawMode(fk_Draw::FACE | fk_Draw::LINE);
Scene.entryModel(&Solid);
```

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 setPointColor()、setLineColor() というメンバ関数を用いる。それぞれの関数は引数として fk_Color のポインタか、RGB を表す float 型の実数 3 個のいずれかを代入する。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。

```
fk_Sphere       Sphere(4, 10.0);
fk_Model        Model;

fk_Material::initDefault();
Model.setShape(&Shape);
Model.setDrawMode(fk_Draw::FACE | fk_Draw::LINE | fk_Draw::POINT);
Model.setMaterial(Yellow);
Model.setLineColor(1.0, 0.0, 0.0);
Model.setPointColor(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 pixel 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 setSize()、setWidth() メンバ関数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。

```
fk_Sphere       Sphere(4, 10.0);
fk_Model        Model;


Model.setShape(&Sphere);
Model.setDrawMode(fk_Draw::POINT | fk_Draw::LINE);
Model.setSize(3.0);
Model.setWidth(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 setSmoothMode() というメンバ関数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、関数の引数に true を与え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。

```
fk_Model        Model;


Model.setSmoothMode(true);      // スムースモード ON
Model.setSmoothMode(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 クラスではメンバ関数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ンバ関数を、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、12 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ンバ関数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ンバ関数は 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、getPosition メンバ関数を用いる。この関数は fk_Vector 型の返り値を持つ。

```
fk_Model model;
fk_Vector position;
        :
        :
position = model.getPosition();
```

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 getVec 関数と getUpvec 関数で参照できる。

```
fk_Model model;
fk_Vector vec, upvec;
        :
        :
vec = model.getVec();
upvec = model.getUpvec();
```

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、getAngle、getMatrix という名のメンバ関数を用いる。

```
fk_Model model;
fk_Angle angle;
fk_Matrix matrix;
        :
        :
angle = model.getAngle();
matrix = model.getMatrix();
```

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 種類のメンバ関数を用意している。

void glTranslate(fk_Vector);
void glTranslate(double, double, double);
void loTranslate(fk_Vector);
void loTranslate(double, double, double);
void glMoveTo(fk_Vector);
void glMoveTo(double, double, double);

8.10.1 glTranslate

glTranslate 関数は、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ンバ関数である。例えば、

```
fk_Vector vec(1.0, 0.0, 0.0);
fk_Model 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 関数は多重定義されており、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);
```

オブジェクト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 関数は、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。

```
fk_Model 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 関数が移動量を与えるのに対して、glMoveTo 関数は実際に移動する位置を直接指定する関数である。したがってこの関数においては、現在位置がどこであってもまったく関係がない。glMoveTo 関数を用いた移動表現は、Translate 関数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 関数は loTranslate で代用できるため、loVec 関数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ンバ関数を紹介する。3 種類のうち、この手法がもっとも直接的 (Primitive) である。

この節では次の 8 種類のメンバ関数を紹介する。

void glFocus(fk_Vector);
void glFocus(double, double, double);
void loFocus(fk_Vector);
void loFocus(double, double, double);
void glVec(fk_Vector);
void glVec(double, double, double);
void glUpvec(fk_Vector);
void glUpvec(double, double, double);
void loUpvec(fk_Vector);
void loUpvec(double, double, double);

このうち、多重定義されている関数は移動関数群と同じように 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 関数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関数である。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。

        fk_Model modelA, modelB;

        for(;;) {
                        :       // ここで、modelA と modelB の移動が
```

```
                    :       // 行われているとする。
                    :
            modelB.glFocus(modelA.getPosition());
                    :
    }
```

この関数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 関数を用いればよい。

8.11.2 loFocus

loFocus 関数は、glFocus のローカル座標系版である。lo 関数群に共通の、主観的な制御には好都合な関数である。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.loFocus(-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

この関数は、モデルの方向ベクトルを直接指定するものである。この関数を用いた場合、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。

この関数は、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 関数は提供されていない。なぜならば loVec 関数は意味的には loFocus 関数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

この関数は、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、この関数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関数ともいえる。実際この関数は、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

察しの通り、この関数は glUpvec のローカル座標系版である。この関数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。この関数は glFocus 関数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 種類の関数に関しての紹介が記述されている。4 つの関数は、次に示す通りである。

void glAngle(fk_Angle);
void glAngle(double, double, double);
void loAngle(fk_Angle);
void 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 つの関数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、次のように定義されている。

```
class fk_Angle {
  public:
      double h;              // ヘディング角
      double p;              // ピッチ角
      double b;              // バンク角
};
```

従って、ベクトルの場合と同様に直接メンバへのアクセスが可能である。

fk_Angle のメンバにしても、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 関数はオイラー角を直接設定する関数である。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 関数よりも glMoveTo 関数に近い。

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 と glAngle を getAngle 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 関数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。

```
        fk_Model model;
        fk_Vector init_pos;
```

```
fk_Angle init_angle;
        :
        :
init_pos = model.getPosition();          // 位置のスナップショット
init_angle = model.getAngle();           // 姿勢のスナップショット
        :
        :
model.glMoveTo(init_pos);                // スナップショットを行った
model.glAngle(init_angle);               // 状態に戻す。
```

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 関数よりもむしろローカル座標系関数である loAngle で真骨頂を発揮する。loAngle 関数では、先に述べた loFocus 関数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。(fk_Math::PI は円周率である。)

```
model.loAngle(fk_Math::PI, 0.0, 0.0);                    // 後ろを向く
model.loAngle(fk_Math::PI/2.0, 0.0, 0.0);                // 右を向く
model.loAngle(-fk_Math::PI/2.0, 0.0, 0.0);               // 左を向く
model.loAngle(0.0, fk_Math::PI/2.0, 0.0);                // 上を向く
model.loAngle(0.0, -fk_Math::PI/2.0, 0.0);               // 下を向く
model.loAngle(fk_Math::PI/2.0, fk_Math::PI/4.0, 0.0);    // 右上を向く
model.loAngle(-fk_Math::PI/4.0, fk_Math::PI/4.0, 0.0);   // 左前上方を向く
model.loAngle(fk_Math::PI/100.0, 0.0, 0.0);              // わずかに右を向く
model.loAngle(0.0, fk_Math::PI/100.0, 0.0);              // わずかに上を向く
model.loAngle(0.0, 0.0, fk_Math::PI);                    // 視線を軸に半回転。
model.loAngle(0.0, 0.0, fk_Math::PI/2.0);                // モデルを右に傾ける。
model.loAngle(0.0, 0.0, fk_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 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、ここで述べる関数群は位置を回転によって制御するためのものといえる。ここで述べられる関数は全部で 16 種類ある。

void glRotate(fk_Vector, fk_Axis, double);
void glRotate(double, double, double, fk_Axis, double);
void glRotate(fk_Vector, fk_Vector, double);

void glRotate(double, double, double, double, double, double, double);

void loRotate(fk_Vector, fk_Axis, double);

void loRotate(double, double, double, fk_Axis, double);

void loRotate(fk_Vector, fk_Vector, double);

void loRotate(double, double, double, double, double, double, double);

void glRotateWithVec(fk_Vector, fk_Axis, double);

void glRotateWithVec(double, double, double, fk_Axis, double);

void glRotateWithVec(fk_Vector, fk_Vector, double);

void glRotateWithVec(double, double, double, double, double, double, double);

void loRotateWithVec(fk_Vector, fk_Axis, double);

void loRotateWithVec(double, double, double, fk_Axis, double);

void loRotateWithVec(fk_Vector, fk_Vector, double);

void loRotateWithVec(double, double, double, double, double, double, double);

この関数群も、実質 8 種類の関数が引数として 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 関数は、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。

```
fk_Model model;
fk_Vector pos(0.0, 0.0, 0.0);
        :
        :
model.glRotate(pos, fk_Axis::X, fk_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 関数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, fk_Math::PI/100.0);
                :
                :
}
```

glRotate 関数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 関数を用いるとよい。この関数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

れらの関数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 メンバ関数と loRotateWithVec メンバ関数は、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 関数と多くの機能が重複している。その例を表 8.2 に示す。ただし、origin は原点を、angle は回転角度を指す。

表 8.2 RotateWithVec と loAngle の比較

RotateWithVec 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、次のようなメンバ関数が提供されている。

```
void setScale(double);
void setScale(double, fk_Axis);
void setScale(double, double, double);
void prdScale(double);
void prdScale(double, fk_Axis);
void prdScale(double, double, double);
```

setScale() メンバ関数はモデルの絶対倍率を設定するための関数である。引数として double 1 個のみをとる関数は、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。

```
fk_Model        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。

```
fk_Model        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関数である。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。

また、現在の拡大率を調べたい場合は次のような拡大率が用意されている。

double getScale(void);

double getScale(fk_Axis);

引数がない場合、全体の拡大率が返る。fk_Axis 型の引数を入れた場合、その引数が示す軸方向の拡大率が返る。

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 setParent メンバ関数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。

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

        CarBody.setShape(&block);
        CarBody.glTranslate(0.0, 100.0, 500.0);
        for(i = 0; i < 4; i++) CarTire[i].setShape(&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].setParent(&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、親モデルが CarBody であることを示すために setParent 関数を呼んでいる。

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ンバ関数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、例のプログラムにおいて `CarTire[i].setShape()` 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 getPosition 関数やオイラー角を得る getAngle 関数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が返り値となっている。

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 getInhPosition (Inh は Inheritance – 継承の略) というメンバ関数を持っている。この関数は getPosition 関数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が返ってくる。これと同様にして getAngle に対応した getInhAngle、getVec に対応した getInhVec、getUpvec に対応した getInhUpvec、getMatrix に対応した getInhMatrix といったメンバ関数を 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 8.3 に示す関数群は親に対する相対的な値を返す。

表 8.3 相対的な値を返す関数群

返り値の型	関数名
fk_Vector	getPosition
fk_Vector	getVec
fk_Vector	getUpvec
fk_Angle	getAngle
fk_Matrix	getMatrix

それに対し、表 8.4 の関数群は絶対的なグローバル座標を返す。

表 8.4　絶対的な値を返す関数群

返り値の型	関数名
fk_Vector	getInhPosition
fk_Vector	getInhVec
fk_Vector	getInhUpvec
fk_Angle	getInhAngle
fk_Matrix	getInhMatrix

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 関数は 2 番目に bool 型の引数と取ることで制御することが可能である。

```
        fk_Model    modelA, modelB;


        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 関数の 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　親子関係に関する関数

以下に、親子関係に関する fk_Model のメンバ関数を羅列する。

void setParent(fk_Model *p, bool flag)

p 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 p 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。

void 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() 関数と同様。

void entryChild(fk_Model *c, bool flag)

c を子モデルの 1 つとして設定する。flag に関しては、setParent() 関数と同様。

void deleteChild(fk_Model *c, bool flag)

c が子モデルであった場合、親子関係を解除する。c 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() 関数と同様。

void 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() 関数と同様。

fk_Model * getParent(void)
自身に設定されている親モデルを返す。親モデルが設定されていない場合、nullptr を返す。

fk_Model * foreachChild(fk_Model *c)
自身に設定されている子モデルを順番に返す。まず c に nullptr が入力されたとき、最初に設定された子モデルが返る。ただし、子モデルが登録されていなかった場合は nullptr が返ってくる。
次に、1 番目に返ってきたモデルを引数として入力したとき、2 番目に設定された子モデルが返ってくる。このようにして、設定されている子モデルを順番に参照することができる。最後の子モデルが引数に入力されたとき、nullptr が返る。

8.17 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。

8.17.1 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 8.1 に境界ボリュームのイメージ図を示す。

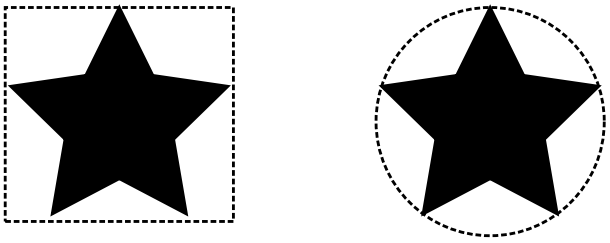

図 8.1 境界ボリューム

以下、FK で利用可能な境界ボリュームを紹介する。

8.17.2 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。

8.17.3 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 8.2 に示す。

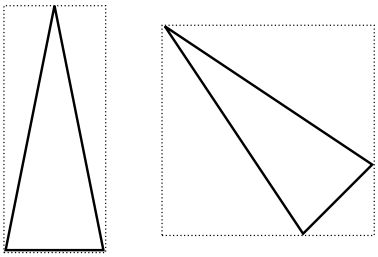

図 8.2　回転による AABB の変化

8.17.4 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 8.3 にモデルの回転と追従するボックスの様子を示す。

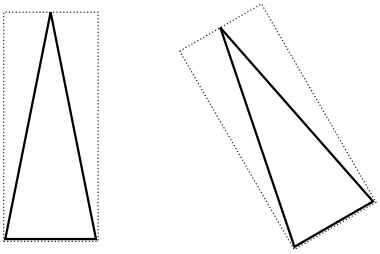

図 8.3　モデル回転に追従する OBB

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。

8.17.5 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 8.4 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。

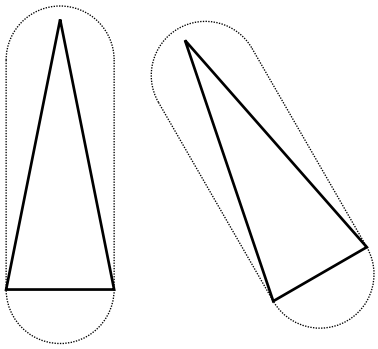

図 8.4　カプセル型の境界ボリューム

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。

8.17.6　干渉判定の方法

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、setBMode() 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.setBMode(fk_BoundaryMode::OBB);
```

setBMode() の引数の種類は、以下の通りである。

表 8.5　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 クラスのメンバ関数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 setShape() 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.setShape(&shape);
    modelA.adjustOBB();
```

あとは、他のモデルを引数として isInter() 関数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。
}
```

8.17.7 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、setInterMode() 関数で true を引数に入力する。

```
modelA.setInterMode(true);
```

その後、干渉が生じたことがある場合は getInterStatus() で true が返ってくる。

```
if(modelA.getInterStatus() == 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。

8.17.8 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、setInterStopMode() で true を引数に与えることで、干渉自動停止モードが ON となる。

```
modelA.setInterStopMode(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 fk_Model::setInterStopMode() のリファレンスマニュアルに掲載されている。

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。

8.17.9 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() 関数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。

第 9 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。

9.1　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。

```
fk_Model model1, model2, model3;
fk_Scene scene;

scene.entryModel(&model1);
scene.entryModel(&model2);
scene.entryModel(&model3);
```

モデルを登録する際、引数はポインタとして渡す。scene オブジェクトの中でこのポインタが参照される。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。

```
fk_Model model1;
fk_Scene disp;
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 setBlendStatus() メンバ関数を用いて透過処理の設定を行う必要がある。これは第 9.4 節に詳しく述べる。

9.2 カメラ (視点) の設定

カメラ (視点) の設定は、モデル (のポインタ) を entryCamera() メンバ関数によって代入することによって行われる。

```
fk_Model camera;
fk_Scene scene;


scene.entryCamera(&camera);
```

entryCamera() 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、別のモデルのポインタを代入すればよい。

9.3 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 setBGColor() メンバ関数を用いることで背景色を変更することができる。引数としては、RGB 要素を示す 0 〜 1 までの数値 3 つ、あるいは fk_Color 型の変数 1 つの 2 種類がある。以下の例は、背景色を青色に設定している例である。

```
fk_Scene scene;
       :
       :
scene.setBGColor(0.0, 0.0, 1.0);
```

9.4 透過処理の設定

モデルのマテリアルにおいて、setAlpha() メンバ関数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 setBlendStatus() メンバ関数を用いて設定を行わなければならない。具体的には、次のように引数に true を入力することによって透過処理の設定が行える。

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。

```
fk_Scene scene;
        :
        :
scene.setBlendStatus(true);
```

透過処理を無効にしたい場合は、false を入力すればよい。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。

9.5 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 9.5.1 節で解説し、霧効果の詳細な利用方法を 9.5.2 節で述べる。

9.5.1 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。

```
fk_Scene        scene;


scene.setFogMode(fk_FogMode::LINEAR);
scene.setFogOption(fk_FogOption::FASTEST);
scene.setFogLinearMap(0.0, 400.0);
scene.setFogColor(0.3, 0.4, 1.0, 0.0);
```

まず、setFogMode() 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 を指定すれば良い。

次に、setFogOption() でオプションを指定する。これは 9.5.2 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、setFogLinearMap() によって霧効果の現れる領域を設定する。最初の数値が霧が出始める距離、後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、setFogColor() 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。

9.5.2 霧効果の詳細な利用方法

霧効果に関連する機能として、各関数の解説を述べる。

void setFogMode(fk_FogMode mode)

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	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。

void setFogOption(fk_FogOption opt)

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 が選択されている。

void setFogDensity(double d)

減衰関数として fk_FogMode::EXP (e^{-dz}) 及び及び 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。

void setFogLinearMap(double S, double E)

減衰関数として FK_LINEAR_FOG ($\frac{E-z}{E-S}$) 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。

void setFogColor(fk_Color col)

void setFogColor(float R, float G, float B, float A)

void setFogColor(double R, double G, double B, double A)

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。

9.6 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 10 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。

10.1　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。

```
fk_AppWindow       window;
```

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、setSize() 関数を使ってピクセル単位で大きさを指定する。

```
window.setSize(600, 600);
```

また、背景色を設定するには setBGColor() を用いる。

```
window.setBGColor(0.3, 0.5, 0.2);
```

setBGColor() は fk_Color 型変数を引数に取ることもできる。

これらの設定を行った後、open() 関数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();
```

10.2　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() 関数を呼び出すことで行われる。この関数が呼ばれた時点で、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() 関数は、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() == true) {
            :
}
```

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。

```
    fk_AppWindow        window;

    window.setSize(600, 600);
    window.setBGColor(0.1, 0.2, 0.3);
    window.open();
    while(window.update() == true) {
            :
            :       // 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。

10.3 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() 関数を用いて行われる。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通りである。

表 10.1 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、数などは以下の関数によって制御可能である。

hideGuide()
座標軸・グリッドを消去する。

setGuideAxisWidth(double w)
座標軸の線幅を w に設定する。

setGuideGridWidth(double w)
グリッドの線幅を w に設定する。

setGuideScale(double s)
グリッドの幅を s に設定する。

setGuideNum(int n)
グリッドの分割数を n に設定する。

10.4 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 クラスは様々なメンバ関数を提供している。ここでは、それらの使用法を説明する。

10.4.1 getKeyStatus() メンバ関数

この関数は、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関数である。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', fk_Switch::PRESS) == true) {
        :       // キーが押された時の処理を行う。
}
```

という記述を行う。この関数は、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 10.2 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') == true)
        :       // キーが押された時の処理を行う。
}
```

としてもよい。

10.4.2 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() 関数の第 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) == true) {
        :       // 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。特殊キーと関数の引数の対応は以下の表 10.3 のとおりである。

表 10.3 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

10.4.3 getMousePosition() 関数

この関数は、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;
fk_AppWindow    window;
        :
        :
pos = window.getMousePosition();
```

というように、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。

getMouseStatus() 関数

この関数は、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_MouseButton::M1、中ボタンが fk_MouseButton::M2、右ボタンが fk_MouseButton::M3 に対応しており、

```
fk_AppWindow       window;

if(window.getMouseStatus(fk_MouseButton::M1, fk_Switch::PRESS) == true) {
        :       // 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。この関数も他と同様に、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。

10.5 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() というメンバ関数が準備されている。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。

10.5.1 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() というメンバ関数を用いることで、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。

```
fk_Vector       in, out;
fk_AppWindow    window;
        :
        :
win.getWindowPosition(in, &out);
```

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。出力となる 2 番目の引数は getMousePosition() の場合と同様にアドレス渡しをする必要がある。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。

10.5.2 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() メンバ関数を利用する。引数は以下の通りである。

```
getProjectPosition(ウィンドウ x 座標，ウィンドウ y 座標，距離、出力変数);
```

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。

```
fk_Vector       pos, out;
fk_AppWindow    window;
fk_Vector
        :
        :
pos = win.getMousePosition();
win.getProjectPosition(pos.x, pos.y, 500.0, &out);
```

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。

```
fk_Plane    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。

```
getProjectPosition(ウィンドウ x 座標成分，ウィンドウ y 座標成分，平面、出力変数);
```

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。

```
fk_Vector       outPos;                 // 出力用変数
fk_AppWindow    win;                    // ウィンドウ変数
fk_Vector       pos;                    // マウス座標用変数
fk_Plane        plane;                  // 平面を表す変数
fk_Vector       planePos, planeNorm;    // 平面生成用変数
        :
        :

// 平面の任意点と法線ベクトルを設定する。
planePos.set(0.0, 0.0, 0.0);
planeNorm.set(0.0, 0.0, 1.0);

// 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.getMousePosition();

// ウィンドウ座標と平面から、3 次元空間中の座標を得る。
win.getProjectPosition(pos.x, pos.y, &plane, &outPos);
```

10.6 高度なウィンドウ制御

FK では、ウィンドウ用のクラスとして fk_AppWindow の他に fk_Window というクラスがある。fk_AppWindow が簡易的な制御を優先した設計であるのに対し、fk_Window はより高度な機能を用いる際に用いられるものである。この節では、fk_Window クラスの主要な機能について説明する。ただし、このマニュアル中だけでは全てを掲載することはかなりの文量を要してしまうため、概略だけに留める。詳細はリファレンスマニュアルを参照してほしい。

10.6.1 ウィンドウの生成

fk_Window クラスは内部的に FLTK と呼ばれる GUI ツールキットを用いており、まず FLTK のメインウィンドウを開き、その中に FK システム用の描画ウィンドウを生成する。

手順としては、まず FLTK 用のウィンドウオブジェクトである Fl_Window クラスのオブジェクトを生成し、その後に fk_Window クラスのオブジェクトを定義する。

```
Fl_Window       mainWin(320, 320, "FK Test");
fk_Window       window(10, 10, 300, 300);


MainWin.end();
```

ここで、Fl_Window オブジェクトの引数はそれぞれ幅、高さ、タイトルバーに表示する文字列を表す。fk_Window オブジェクトの引数は (x, y, w, h) とすると (x, y) はメインウィンドウとの相対位置を、(w, h) は fk_Window 自体の大きさを表す。mainWin.end() は、全ての fk_Window オブジェクトを定義し終わった後に呼ぶ関数である。したがって、fk_Window オブジェクトを 2 つ利用してマルチウィンドウなプログラムを作りたい場合には、

```
Fl_Window       mainWin(640, 320, "Multi Win Test");
fk_Window       window1(10, 10, 300, 300);
fk_Window       window2(330, 10, 300, 300);


MainWin.end();
```

というようにすればよい。また、ウィンドウを実際に開く関数は show() 関数である。具体的には、

```
mainWindow.show();
window.show();
```

というように、ウィンドウを実際に開きたい部分で show() 関数を呼び出せばよい。

10.6.2 シーンの設定

fk_Window クラスは fk_AppWindow クラスとは異なり、シーン制御機能を自身では保持していない。そのため、fk_Window クラスを用いる際には fk_Scene クラスのインスタンスを作成し、そのインスタンスでシーン制御を行う必要がある。

作成した fk_Window クラスのウィンドウに対し、次のように setScene() 関数を用いることによって fk_Scene クラス内のシーンをウィンドウ中に表示することができる。

```
Fl_Window       mainWin
fk_Window       window;
fk_Scene        scene;


mainWin.end();
        :
        :
window.setScene(&scene);
```

10.6.3 ウィンドウの描画

ウィンドウの描画は、drawWindow() 関数を用いて行う。

```
window.drawWindow();
```

この関数が呼ばれた時点で、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。ただし、この関数はウィンドウが実際にまだ開いていないときにも描画されてしまうため、そのまま使用した場合は誤動作を起こす場合がある。そこで、ウィンドウが実際に開いているかどうかを判定する関数として winOpenStatus() 関数が用意されている。これは true か false かを返し、ウィンドウが開いている場合には true を返す。また、FLTK が提供しているチェック関数もここで呼んでおかなければならない。これは、Fl_Window クラスの visible() 関数と、Fl::check() を用いる。

これを踏まえ、実際の描画ループは次のようになる。

```
Fl_Window       mainWindow(320, 320, "FK TEST");
fk_Window       window(10, 10, 300, 300);

mainWindow.end();
mainWindow.show();
window.show();
while(true) {
    // メインウィンドウが開いているかどうかのチェック
    if(mainWindow.visible() == 0) {
        // 開いていなかった（最小化の状態の）場合
        if(Fl::wait() == 0) {
            // 終了命令がでた場合
            break;
        } else {
            // そのままループの最初に戻る
            continue;
        }
    }
    // 描画処理
    if(window.drawWindow() == 0) break;

    // FLTK から重大なエラーの報告がないかをチェック
    if(Fl::check() == 0) break;

    // fk_Window が描画できる状態になっているかをチェック
    if(window.winOpenStatus() == false) continue;
        :
        :   // モデルの制御
        :
```

```
            }
        }
```

drawWindow() 関数は、もし window 上で ESC キーが押された場合には 0 は返ってくるようになっている。従って、例のプログラムの場合は ESC が押されると終了するようになっている。

10.6.4 デバイス情報取得と座標系変換

fk_Window におけるデバイス情報取得機能は、fk_AppWindow とほぼ互換のメンバ関数が準備されている。具体的な利用方法は 10.4 節を参照されたい。

また、10.5 節で説明した座標系変換の機能も fk_AppWindow と共通である。

10.6.5 ウィンドウ表示状態の画像情報取り込み

fk_Window に表示されている状態を、画像ファイルや画像データに取り込みたい場合は、snapImage() というメンバ関数を利用する。仕様は以下の通りである。

bool snapImage(string fileName, fk_ImageType type, fk_SnapProcMode mode)

現在ウィンドウに表示されている画像を、指定した画像形式で fileName にあるファイル名で保存する。type は以下のいずれかを指定する。

- fk_ImageType::BMP (Windows Bitmap 形式)
- fk_ImageType::PNG (PNG 形式)、
- fk_ImageType::JPG (JPEG 形式)

また、mode はどの画像バッファから情報を取得するかを指定する物で、以下のいずれかを設定する。

- fk_SnapProcMode::FRONT (OpenGL フロントバッファ)
- fk_SnapProcMode::BACK (OpenGL バックバッファ)
- fk_SnapProcMode::WIN32_GDI (Windows のみ)

type と mode はデフォルト引数が設定されており、省略可能である。省略した場合、type は fk_ImageType::BMP、mode は fk_SnapProcMode::FRONT となる。

出力に成功すれば true を、失敗すれば false を返す。

bool snapImage(fk_Image *image, fk_SnapProcMode mode)

現在ウィンドウに表示されている画像を、fk_Image 型の変数に格納する。mode に関しては上記のファイル出力の場合と同様である。また、mode はデフォルト引数が設定されており、省略可能である。省略した場合 mode は fk_SnapProcMode::FRONT となる。

出力に成功すれば true を、失敗すれば false を返す。

引数としてファイル名を取る物と fk_Image 型 (のポインタ) 変数を取るものがある。前者は、指定されたファイル名で指定した形式で保存するものである。後者は、fk_Image 型の変数に画像情報を格納する。以下のプログラムは、「image.bmp」というファイルと「image」変数にそれぞれ表示画像を格納するものである。

```
fk_Window       win(0, 0, 300, 300);
fk_Image        image;

win.snapImage("image.bmp", fk_ImageType::BMP, fk_SnapProcMode::BACK);
win.snapImage(&image, fk_SnapProcMode::BACK);
```

10.6.6 ウィンドウのサイズ変更

fk_Window で生成したウィンドウ領域を途中で変更したい場合には、resizeWindow() というメンバ関数を利用する。この関数は、引数として生成時と同様に 4 つの数値を取る。

```
fk_Window       win(0, 0, 300, 300);

win.resizeWindow(0, 0, 200, 200);
```

この関数は、元の Fl_Window の大きさ変更に追随する機能を持たせる時に利用することができる。次のプログラムは、ウィンドウの大きさ変更を可能とする典型的なサンプルである。

```
Fl_Window   mainWin(300, 300, "Resize Test");
fk_Window   fkWin(0, 0, 300, 300);

mainWin.end();

// 全体の大きさの最小値を（100, 100）にセット。
mainWin.size_range(100, 100);

mainWin.show();
fkWin.show();

while(true) {
    // ウィンドウの大きさを mainWin と同一に合わせる。
    fkWin.resizeWindow(0, 0, mainWin.w(), mainWin.h());

    // 後は通常と共通
    if(mainWin.visible() == 0) {
        if(Fl::wait() == 0) {
            break;
        } else {
            continue;
        }
    }
```

```
        if(fkWin.drawWindow() == 0) break;
        if(Fl::check() == 0) break;
        if(fkWin.winOpenStatus() == false) continue;

            :
            :
    }
```

10.6.7 テクスチャメモリの解放

テクスチャマッピングを行う際、何度も画像の切り替えを行っているとアプリケーションが利用するメモリ量が非常に増加することがある。これは、テクスチャの画像情報をシステムが保持し続けることが原因である。そこで、これらのメモリをクリアする **clearTextureMemory(void)** 関数を利用することで、一旦システムが保持したメモリを解放することができる。

ただし、この関数によってメモリを解放した後は、テクスチャの画像情報を再度読み込み直すことになるので、処理時間が増える可能性もある。適宜タイミングを調整してほしい。

10.6.8 メッセージ出力

fk_Window には、メッセージを出力する機能として以下のようなメンバ関数が提供されている。

void setPutStrMode(fk_PutStrMode mode)

メッセージを出力する際の、出力方法を指定する。引数として選択できるものは、以下のとおり。

値	意味
fk_PutStrMode::BROWSER	メッセージ出力用ブラウザに出力する。
fk_PutStrMode::CONSOLE	標準出力に出力する。
fk_PutStrMode::ERR_CONSOLE	エラー出力に出力する。
fk_PutStrMode::FILE	setPutFile() 関数で指定したファイルに出力する。
fk_PutStrMode::NONE	出力を行わない。

デフォルトは fk_PutStrMode::BROWSER となっている。

fk_PutStrMode getPutStrMode(void)

現在のメッセージ出力モードを返す。

bool setPutFile(string fileName)

メッセージ出力のモードとして「fk_PutStrMode::FILE」を指定した場合の出力先となるファイル名を指定する。この関数が呼ばれた時点で、指定したファイルが存在していなかった場合は新規に空ファイルが作成される。既に存在していた場合は、その中身を消去せずに最後部分から後にメッセージを出力する。ファイル書き込みの準備に成功した場合 true を、失敗した場合に false を返す。

void putString(string message)

message の内容を出力する。

void printf(char *format, ...)

標準の printf 関数に準拠した書式指定に従って文字列を出力する。

void clearBrowser(void)

メッセージ出力用ブラウザの内容を消去する。

デフォルトの出力ブラウザを利用してメッセージを出力する場合は、単に putString() 関数あるいは printf() 関数を利用すればよい。以下のプログラムは、printf() を用いたサンプルである。

```
fk_Window       window(10, 10, 300, 300);
int             counter;
        :
        :
counter = 0;
while(true) {
        :
        :
        counter++;
        window.printf("counter = %d", counter);
}
```

また、ここで紹介した関数群は fk_Window のメンバ関数ではあるが、fk_Window 型の変数がなくても関数名の前に「`fk_Window::`」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_Window::putString(outString);
fk_Window::printf("value = %d", value);
```

1) これは、これらの関数が static 宣言をしているためである。

第 11 章　簡易形状表示システム

10 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。

11.1　形状表示

利用するには、まず fk_ShapeViewer 型の変数を定義する。

```
fk_ShapeViewer  viewer;
```

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_IndexFaceSet 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() メンバ関数を用いる。setShape() メンバ関数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。

```
fk_ShapeViewer  viewer;
fk_IndexFaceSet solid;
fk_Sphere       sphere;

viewer.setShape(0, &solid);
viewer.setShape(1, &sphere);
```

あとは、draw() 関数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() 関数は 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 節及び 12.5 節に解説が記述されている。

11.2 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。

11.3 fk_ShapeViewer のメンバ関数

fk_ShapeViewer クラスの具体的な利用法は、第 12.5 節に記述するが、ここでは fk_ShapeViewer クラスのメンバ関数一覧を掲載する。

bool draw(void)

実際に描画を行う。

void setWindowSize(int w, int h)

描画領域の大きさを (w, h) にする。形状やその他の状態は保持される。

void setShape(int ID, fk_Shape *shape)

形状を描画対象として設定する。ID には何か任意の整数値を入れる。複数の形状を同時に描画したい場合、ID を変えることで実現できる。逆に、前に設定した形状と描画対象を入れ替えたい場合は、前に設定した際の ID をそのまま入力することで実現できる。

void setDrawMode(fk_DrawMode mode)

形状の描画モードを選択する。mode に入力できる選択肢は、8.3 節で述べているものと同一のものが選択できる。

void setBGColor(fk_Color color)

void setBGColor(float r, float g, float b)

形状描画領域の背景色を設定する。

void setHead(double r)

カメラのヘッド角を r ラジアンに設定する。

void setPitch(double r)

カメラのピッチ角を r ラジアンに設定する。

void setBank(double r)

カメラのバンク角を r ラジアンに設定する。

void setScale(double s)

表示倍率を s に設定する。

void setAxisMode(bool mode)

座標軸について、mode が true なら描画有り、false なら無しに設定する。

void setAxisScale(double l)

座標軸の長さを l に設定する。

void setCenter(fk_Vector pos)

void setCenter(double x, double y, double z)

カメラの注視点を pos または (x, y, z) に設定する。

void setMaterial(int ID, fk_Material mat)

ID が表す形状のマテリアルを mat が表すマテリアルに変更する。

void setEdgeColor(int ID, fk_Color col)

ID が表す形状の稜線色を、col が表す色に変更する。

void setVertexColor(int ID, fk_Color col)

ID が表す形状の頂点色を、col が表す色に変更する。

void setPosition(int ID, fk_Vector pos)

void setPosition(int ID, double x, double y, double z)

ID が表す形状の位置を変更する。

void setAngle(int ID, fk_Angle angle)

void setAngle(int ID, double h, double p, double b)

ID が表す形状の姿勢を変更する。h にはヘディング角を、p にはピッチ角を、b にはバンク角を入力する。

void setVec(int ID, fk_Vector vec)

void setVec(int ID, double x, double y, double z)

ID が表す形状の方向ベクトルを変更する。

void setUpvec(int ID, fk_Vector vec)

void setUpvec(int ID, double x, double y, double z)

ID が表す形状のアップベクトルを変更する。

setBlendStatus(bool mode)

透過処理について、mode が true であれば有効、false であれば無効に設定する。

void clearModel(void)
現在の描画登録を全てクリアする。

fk_Shape * getShape(int ID)
ID に設定されている形状データ (のアドレス) を返す。

fk_DrawMode getDrawMode(void)
現在の描画モードを返す。

double getHead(void)
現在設定されているカメラのヘッド角をラジアンで返す。

double getPitch(void)
現在設定されているカメラのピッチ角をラジアンで返す。

double getBank(void)
現在設定されているカメラのバンク角をラジアンで返す。

double getScale(void)
現在の表示倍率を返す。

bool getAxisMode(void)
現在の座標軸描画の有無を返す。

double getAxisScale(void)
現在の座標軸の長さを返す。

fk_Vector getCenter(void)
現在のカメラの注視点を返す。

fk_Color getBGColor(void)
背景色を返す。

bool getBlendStatus(void)
現在の透過処理状態を返す。

bool snapImage(const string fileName, fk_ImageType type, fk_SnapProcMode mode)
現在ウィンドウに表示されている画像を、指定した画像形式で fileName にあるファイル名で保存する。type は以下の 3 種類のうちいずれかを指定する。

- fk_ImageType::BMP (Windows Bitmap 形式)
- fk_ImageType::PNG (PNG 形式)
- fk_ImageType::JPG (JPEG 形式)

また、mode はどの画像バッファから情報を取得するかを指定する物で、以下の 3 種類のいずれかを指定する。

- fk_SnapProcMode::FRONT (OpenGL フロントバッファ)
- fk_SnapProcMode::BACK (OpenGL バックバッファ)
- fk_SnapProcMode::WIN32_GDI (Windows のみ)

bool snapImage(fk_Image *image, fk_SnapProcMode mode)

現在ウィンドウに表示されている画像を、fk_Image 型の変数に格納する。mode に関しては上記のファイル出力の場合と同様である。また、mode はデフォルト引数が設定されており、省略可能である。省略した場合 mode は FK_SNAP_GL_FRONT となる。

出力に成功すれば true を、失敗すれば false を返す。

void setPutStrMode(fk_PutStrMode mode)

fk_PutStrMode getPutStrMode(void)

bool setPutFile(string fileName)

void putString(string message)

void printf(char *format, ...)

void clearBrowser(void)

これらの関数は、fk_Window クラスと同様にメッセージ出力機能を利用するためのものである。詳細は 10.6.8 節を参照のこと。

第 12 章　サンプルプログラム

12.1　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。

- 1 行目の include 文は、FK システムを使用する場合に必ず記述する。
- 3 行目の using 文を書いておくと、名前空間「FK」を省略することができる。using を用いない場合は、FK のクラス名の前には「FK::」を追加する必要がある。
- 7 行目において、メインウィンドウを生成している。
- 8 行目において、視点 (camera)、直方体 (blockModel)、線分 2 本 (lineModel[2]) 用の変数を定義している。
- 10 行目は線分形状、11 行目は直方体形状を作成するための変数を作成している。
- 14 行目の記述は、最初から準備されている「標準マテリアル」を用いるための準備である。標準マテリアルを使用したい場合は、プログラムの最初にこの fk_Material::initDefault() 関数を呼ぶ。
- 17 行目にて、ウィンドウのサイズをピクセル単位で指定している。
- 20 行目で、blockModel に block の形状情報をリンクしている。
- 21 行目で、blockModel へのマテリアルとして Material::Yellow を設定している。
- 24 〜 31 行目で、線分形状を生成し lineModel[] にリンクしている。28 行目では &pos[0] を代入することによって pos[0] と pos[1] が、29 行目では &pos[2] を代入することによって pos[2] と pos[3] がそれぞれ線分の端点となる。
- 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 setLineColor() を用いる。
- 38,39 行目で、線分を直方体の子モデルに設定している。これにより、直方体を移動すると線分も追従して移動していくようになる。
- 42 〜 44 行目で、視点の位置を設定している。この場合は、位置を $(0, 0, 2000)$、視線方向を原点に向け、カメラの上方向 (アップベクトル) が $(0, 1, 0)$ になるように設定している。
- 47 行目でカメラの登録を、48 〜 50 行目で、各モデルをウィンドウへ登録している。
- 52 行目でウィンドウを開いている。この処理を行わないと、fk_AppWindow 型の変数を用意しても表示はされないので注意が必要である。
- 56 行目の for 文中にある window.update() が実際に表示を行う処理である。ウィンドウが正常に描画された場合は true が返るため、ループが継続する。ウィンドウが閉じられていた場合、false が返ってくるので for 文を抜けることになる。
- 60 行目で、視点位置を $-z$ 方向に speed の数値分移動させている。
- 63 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、speed に $\pi/300 = 0.6^\circ$ を掛け合わせた角度である。
- 66 〜 68 行目で、カメラの z 座標が 1000 未満になる場合に、視点にひねりを加えている。
- 71 〜 73 行目で、もし視点が原点を越えてしまった場合に注視点を原点に向かせるようにしている。

```
1: #include <FK/FK.h>
```

```
 2:
 3: using namespace FK;
 4:
 5: int main(int, char *[])
 6: {
 7:     fk_AppWindow    window;
 8:     fk_Model        camera, blockModel, lineModel[2];
 9:     fk_Vector       pos[4];
10:     fk_Line         line[2];
11:     fk_Block        block(50.0, 70.0, 40.0);
12:
13:     // マテリアルの初期化
14:     fk_Material::initDefault();
15:
16:     // ウィンドウ設定
17:     window.setSize(800, 800);
18:
19:     // 直方体の設定
20:     blockModel.setShape(&block);
21:     blockModel.setMaterial(Material::Yellow);
22:
23:     // 線分の設定
24:     pos[0].set(0.0, 100.0, 0.0);
25:     pos[1].set(100.0, 0.0, 0.0);
26:     pos[2] = -pos[0];
27:     pos[3] = -pos[1];
28:     line[0].setVertex(&pos[0]);
29:     line[1].setVertex(&pos[2]);
30:     lineModel[0].setShape(&line[0]);
31:     lineModel[1].setShape(&line[1]);
32:
33:     // 線分の色設定
34:     lineModel[0].setLineColor(1.0f, 0.0f, 0.0f);
35:     lineModel[1].setLineColor(0.0f, 1.0f, 0.0f);
36:
37:     // 直方体を線分の親モデルに設定
38:     lineModel[0].setParent(&blockModel);
39:     lineModel[1].setParent(&blockModel);
40:
41:     // 視点の位置と姿勢を設定
42:     camera.glMoveTo(0.0, 0.0, 2000.0);
43:     camera.glFocus(0.0, 0.0, 0.0);
44:     camera.glUpvec(0.0, 1.0, 0.0);
```

```
45:
46:     // 各モデルをウィンドウに登録
47:     window.setCameraModel(&camera);
48:     window.entry(&blockModel);
49:     window.entry(&lineModel[0]);
50:     window.entry(&lineModel[1]);
51:
52:     window.open();
53:
54:     double speed = 5.0;
55:
56:     for(int i = 0; window.update() == true; i++) {
57:         double z = camera.getPosition().z;
58:
59:         // 視点を原点に近づける
60:         camera.glTranslate(0.0, 0.0, -speed);
61:
62:         // 直方体(と子モデルの線分)を Y 軸中心に回転
63:         blockModel.glRotateWithVec(0.0, 0.0, 0.0, fk_Axis::Y, speed * fk_Math::PI/300.0);
64:
65:         // カメラが 1000 より近くなったら Z 軸回転
66:         if(z < 1000.0) {
67:             camera.loRotateWithVec(0.0, 0.0, 0.0, fk_Axis::Z, speed * fk_Math::PI/500.0);
68:         }
69:
70:         // 視点が原点を越えたら、向きをもう一度原点に向かせる。
71:         if(z < -fk_Math::EPS) {
72:             camera.glFocus(0.0, 0.0, 0.0);
73:         }
74:
75:     }
76:
77:     return 0;
78: }
```

12.2 LOD 処理とカメラ切り替え

次のサンプルは、ボールが弾む様子を描いたプログラムである。このプログラムを実行すると、ボールが 2 回弾む間は鳥瞰的なカメラ視点だが、その後に青色のブロックからボールを見る視点に切り替わる。プログラムのおおまかな流れは自分で解析して頂きたいが、ここでは最初のサンプルにはなかった概念に関して説明する。

まず、17 行目にて Ball クラスを定義し、この中で fk_Model や fk_Sphere のオブジェクトを持つようにしている。このようなプログラミングスタイルは、ややオブジェクト指向を意識したものと言えよう。

25 〜 27 行目にて球を分割数を変えて 3 種類定義しているが、これは 78 行目からの lod() メンバ関数内で Ball の実際の

形状を動的に選択できるようにするためである。78 行目から実現している LOD 処理とは、視点とオブジェクトの距離によって形状の精密さを動的に変化させるテクニックである。たとえば、多くのポリゴンでできている形状は精密で迫力があるが、視点からとても遠くて非常に小さく表示されているような場合は無駄に処理されていることになる。そこで、遠くにあって小さく表示されている場合は粗い形状を表示して処理の高速化を計るのが LOD 処理の目的である。このサンプルの LOD 処理はわかりやすくするために露骨に変化が見られるが、実際にプログラムを作成するときはわかりにくくなるように視点距離との関係を調整する。

```
 1: #include <FK/FK.h>
 2:
 3: const double    DOWN_ACCEL      = 1.0500;   // 降下時の加速度
 4: const double    RISE_ACCEL      = 1.0530;   // 上昇時の減速度
 5: const int       DOWN_MODE       = 0;        // 降下モード
 6: const int       RISE_MODE       = 1;        // 上昇モード
 7: const int       LOW_MODE        = 0;        // ブロック視点モード
 8: const int       HIGH_MODE       = 1;        // 鳥瞰モード
 9: const int       LOD4_HIGH       = 200;      // 4 分割距離（鳥瞰）
10: const int       LOD3_HIGH       = 300;      // 3 分割距離（鳥瞰）
11: const int       LOD4_LOW        = 90;       // 4 分割距離（ブロック）
12: const int       LOD3_LOW        = 120;      // 3 分割距離（ブロック）
13: const double    TOP_BALL_POS    = 300.0;    // ボール始点高さ
14: const double    BTM_BALL_POS    = 18.0;     // ボール跳ね返り高さ
15: const double    BALL_SIZE       = 12.0;     // ボール半径
16:
17: class Ball {
18:
19: private:
20:     int         direction;      // ボールの状態 (DOWN_MODE or RISE_MODE)
21:     int         view_mode;      // 視点モード
22:     int         bound_count;    // バウンド回数を数える変数
23:     double      y_trs;          // ボールの y 座標移動量
24:     fk_Model    ball_model;     // ボールのモデル
25:     fk_Sphere   BALL2;          // 2 分割形状
26:     fk_Sphere   BALL3;          // 3 分割形状
27:     fk_Sphere   BALL4;          // 4 分割形状
28:
29: public:
30:
31:     Ball(void);
32:     void        init(void);
33:     fk_Model *  getModel(void);
34:     fk_Vector   getPosition(void);
35:     void        lod(fk_Vector);
36:     void        accel(void);
```

```
37:     void        bound(void);
38:     int         draw(fk_Vector);
39: };
40:
41: // コンストラクタ
42: Ball::Ball(void)
43: {
44:     init();
45: }
46:
47: // 初期化
48: void Ball::init(void)
49: {
50:     direction    = DOWN_MODE;
51:     y_trs        = 0.1;
52:     view_mode    = HIGH_MODE;
53:     bound_count  = 1;
54:     BALL2.setRadius(BALL_SIZE);
55:     BALL2.setDivide(2);
56:     BALL3.setRadius(BALL_SIZE);
57:     BALL3.setDivide(3);
58:     BALL4.setRadius(BALL_SIZE);
59:     BALL4.setDivide(4);
60:
61:     ball_model.glMoveTo(0.0, TOP_BALL_POS, 0.0);
62:     ball_model.setShape(&BALL2);
63: }
64:
65: // fk_Model を返す関数
66: fk_Model * Ball::getModel(void)
67: {
68:     return &ball_model;
69: }
70:
71: // ボールの現在位置を返す関数
72: fk_Vector Ball::getPosition(void)
73: {
74:     return ball_model.getPosition();
75: }
76:
77: // 視点からの距離によってボールの分割数を変える関数 (Level Of Detail)
78: void Ball::lod(fk_Vector pos){
79:     double    distance;
```

```
 80:
 81:     distance = (ball_model.getPosition() - pos).dist();
 82:
 83:     switch(view_mode) {
 84:       case HIGH_MODE:
 85:
 86:         if(distance < LOD4_HIGH) {
 87:             ball_model.setShape(&BALL4);
 88:         } else if(distance < LOD3_HIGH) {
 89:             ball_model.setShape(&BALL3);
 90:         } else {
 91:             ball_model.setShape(&BALL2);
 92:         }
 93:         break;
 94:
 95:       case LOW_MODE:
 96:
 97:         if(distance < LOD4_LOW) {
 98:             ball_model.setShape(&BALL4);
 99:         } else if(distance < LOD3_LOW) {
100:             ball_model.setShape(&BALL3);
101:         } else {
102:             ball_model.setShape(&BALL2);
103:         }
104:         break;
105:
106:       default:
107:         fl_alert("Err!! View Mode is wrong.");
108:         break;
109:     }
110:
111:     return;
112: }
113:
114: // ボールを加速させる関数
115: void Ball::accel(void)
116: {
117:     switch(direction) {
118:       case DOWN_MODE:
119:         y_trs *= DOWN_ACCEL;
120:         ball_model.glTranslate(0.0, -y_trs, 0.0);
121:         break;
122:
```

```
123:       case RISE_MODE:
124:         y_trs /= RISE_ACCEL;
125:         ball_model.glTranslate(0.0, y_trs,0.0);
126:         break;
127:
128:       default:
129:         fl_alert("Err!! Direction Mode is wrong.");
130:         break;
131:     }
132: }
133:
134: // ボールの跳ね返り判定をする関数
135: void Ball::bound(void)
136: {
137:     if(ball_model.getPosition().y < BTM_BALL_POS) {
138:         direction = RISE_MODE;
139:     } else if(y_trs < 0.01) {
140:         if(direction == RISE_MODE) {
141:             if(bound_count % 4 < 2) {
142:                 view_mode = HIGH_MODE;
143:             } else {
144:                 view_mode = LOW_MODE;
145:             }
146:             bound_count++;
147:         }
148:         direction = DOWN_MODE;
149:     }
150:
151:     return;
152: }
153:
154: // ボールの運動に関する関数．返り値は視点モード
155: int Ball::draw(fk_Vector pos)
156: {
157:     lod(pos);
158:     bound();
159:     accel();
160:     //４回跳ね返ると初期化
161:     if(bound_count > 4) init();
162:     return view_mode;
163: }
164:
165: int main(int argc, char *argv[])
```

```
166: {
167:     fk_Scene        scene;
168:     int             view_mode = HIGH_MODE;
169:     Ball            ball;
170:     fk_Model        viewModel, lightModel, groundModel, blockModel;
171:     fk_Light        light;
172:     fk_Circle       ground(4, 100.0);
173:     fk_Block        block(10.0, 10.0, 10.0);
174:
175:
176:     // ### WINDOW ###
177:     Fl_Window        mainWin(620, 620, "BALL TEST");
178:     fk_Window        win(10, 10, 600, 600);
179:     mainWin.end();
180:     fk_Material::initDefault();
181:
182:     // ### VIEW POINT ###
183:     // 上の方から見た視点
184:     viewModel.glMoveTo(0.0, 400.0, 80.0);
185:     viewModel.glFocus(0.0, 30.0, 0.0);
186:     viewModel.glUpvec(0.0, 1.0, 0.0);
187:
188:     // ### LIGHT ###
189:     light.setLightType(FK_POINT_LIGHT);
190:     light.setAttenuation(0.0, 0.0);
191:     lightModel.setShape(&light);
192:     lightModel.setMaterial(White);
193:     lightModel.glTranslate(-60.0, 60.0, 0.0);
194:
195:     // ### GROUND ###
196:     groundModel.setShape(&ground);
197:     groundModel.setMaterial(LightGreen);
198:     groundModel.setSmoothMode(true);
199:     groundModel.loRotateWithVec(0.0, 0.0, 0.0, fk_X, -FK_PI/2.0);
200:
201:     // ### VIEW BLOCK ###
202:     blockModel.setShape(&block);
203:     blockModel.setMaterial(Blue);
204:     blockModel.glMoveTo(60.0, 30.0, 0.0);
205:     blockModel.setParent(&groundModel);
206:
207:     // ### BALL ###
208:     ball.getModel()->setMaterial(Red);
```

```
209:     ball.getModel()->setSmoothMode(true);
210:
211:     // ### Model Entry ###
212:     scene.entryCamera(&viewModel);
213:     scene.entryModel(&lightModel);
214:     scene.entryModel(ball.getModel());
215:     scene.entryModel(&groundModel);
216:     scene.entryModel(&blockModel);
217:     win.setScene(&scene);
218:
219:     mainWin.show();
220:     win.show();
221:
222:     // ### MAIN LOOP ###
223:     while(true) {
224:
225:         if(mainWin.visible() == 0) {
226:             if(Fl::wait() == 0) {
227:                 break;
228:             } else {
229:                 continue;
230:             }
231:         }
232:         if(win.drawWindow() == 0) break;
233:         if(Fl::check() == 0) break;
234:         if(win.winOpenStatus() == false) continue;
235:
236:         // ボールを弾ませて，カメラの状態を取得。
237:         view_mode = ball.draw(viewModel.getPosition());
238:
239:         if(view_mode == HIGH_MODE) {
240:             // カメラを上からの視点にする。
241:             viewModel.glMoveTo(0.0, 400.0, 80.0);
242:             viewModel.glFocus(0.0, 30.0, 0.0);
243:             viewModel.glUpvec(0.0, 1.0, 0.0);
244:             scene.entryModel(&blockModel);
245:         } else {
246:             // カメラをブロックからの視点にする。
247:             viewModel.glMoveTo(blockModel.getInhPosition());
248:             viewModel.glTranslate(0.0, 10.0, 0.0);
249:             viewModel.glFocus(ball.getPosition());
250:             viewModel.glUpvec(0.0, 1.0, 0.0);
251:             scene.removeModel(&blockModel);
```

```
252:         }
253:
254:         // 地面をくるくる回転させましょう。
255:         groundModel.glRotateWithVec(0.0, 0.0, 0.0, fk_Y, 0.02);
256:     }
257:
258:     return 0;
259: }
```

12.3 FLTK を用いた GUI 構築

本節で説明するサンプルは、GUI (Graphical User Interface) を用いた立体ビューワーである。ウィンドウの章でも少し触れたが、FK システムは内部で FLTK と呼ばれる GUI ツールキットを用いている。ここでは、その FLTK を用いて GUI を作成し、FK システムと併用するサンプルを紹介する。なお、本書では FLTK の詳細な解説は掲載しないが、FLTK の解説は `http://www.fltk.org/` という URL に詳しく記載されている。

ボールのサンプルと同じく、プログラム行のそれぞれの意味はコメントで記述しているので、全体の流れを通して幾つかわかりにくい部分を記述する。

- 最初に include するファイルが FK/FK だけでなく、FLTK 用の様々なファイルが多くある。FLTK を用いる場合は、このように適宜必要なファイルを include しなければならない。
- 10 行目から始まる GUISet クラスは、GUI の部品を管理するためのものである。このクラスのコンストラクタ (41 〜 115 行目) 内で様々な部品を作成するようにしてあるので、GUISet オブジェクトの宣言 (264 行目) をするだけでコンストラクタが呼び出され、GUI が作成されるようになっている。
- 122 行目からある getFileName メンバ関数は、Fl_File_Chooser クラスを用いてファイル名を取得するための関数である。まず 128、129 行目でファイル取得用ダイアログを生成し、131 行目の while 文でダイアログが終了されるまで処理を行う。もしダイアログ中でキャンセルされていた場合は、135 行目の fCharP に nullptr が入るため、136 行目の if 文に該当する。そうでない場合は、143 行目まで進んで文字列 (ファイル名) を返す。
- 161 行目では数学関数の pow を用いている。pow(a, b) は a の b 乗を返す関数なので、scaleRoller の値が実際には 2 であれば $100(=10^2)$ を、3 であれば $1000(=10^3)$ を、-1 であれば $0.1(=10^{-1})$ を返す。
- 314、315 行目では照明を Y 軸中心に少しだけ回転させている。形状を回転させずに立体感を出す方法として、このように照明を回転させることも有効な手段である。
- 318、319 行目で、GUI のスライダーやローラーで設定された値を実際に反映させている。
- 325 行目にて、fl_color_chooser によってカラー設定ウィンドウを呼び出している。この関数の 2 〜 4 番目の引数は実は参照呼び出しになっている。従って、colorR, colorG, colorB の 3 変数は色設定の後に値がちゃんと反映されている。

```
1: #include <FK/FK.h>
2: #include <FL/Fl_Value_Slider.h>
3: #include <FL/Fl_Roller.h>
4: #include <FL/Fl_Group.h>
5: #include <FL/Fl_Check_Button.h>
6: #include <FL/Fl_Color_Chooser.h>
7: #include <FL/Fl_File_Chooser.h>
8: #include <FL/fl_ask.h>
9:
```

```
10: class GUISet {
11: private:
12:
13:     Fl_Value_Slider *headSlider;            // ヘディング角用スライダー
14:     Fl_Value_Slider *pitchSlider;           // ピッチ各用スライダー
15:     Fl_Roller       *scaleRoller;           // スケーリング用ローラー
16:
17:     Fl_Group        *materialGroup;         // ラジオボタンのグループ
18:     Fl_Check_Button *ambientButton;         // Ambient 用ボタン
19:     Fl_Check_Button *diffuseButton;         // Diffuse 用ボタン
20:     Fl_Check_Button *specularButton;        // Specular 用ボタン
21:     Fl_Check_Button *emissionButton;        // Emission 用ボタン
22:     Fl_Button       *colorChooser;          // カラー設定呼び出しボタン
23:
24:     Fl_Button       *fileOpenButton;        // ファイル選択呼び出しボタン
25:     Fl_Button       *exitButton;            // 終了ボタン
26:
27: public:
28:     GUISet(void);                           // コンストラクタ
29:     ~GUISet();                              // デストラクタ
30:
31:     string          getFileName(void);      // ファイル名取得
32:     double          getHead(void);          // ヘディング角取得
33:     double          getPitch(void);         // ピッチ角取得
34:     double          getScale(void);         // スケール取得
35:     int             materialSelect(void);   // マテリアル種類取得
36:     bool            toggleColorChooser(void);// カラー設定呼出判定
37:     bool            toggleFileOpen(void);   // ファイル取得呼出判定
38:     bool            toggleExit(void);       // 終了判定
39: };
40:
41: GUISet::GUISet(void)
42: {
43:     // ヘディング角用スライダー各種設定
44:     headSlider = new Fl_Value_Slider(130, 330, 180, 20, "Head Angle");
45:     headSlider->type(FL_HOR_NICE_SLIDER);  // ナイスなスライダー
46:     headSlider->minimum(-FK_PI);           // 最小値は -3.14
47:     headSlider->maximum(FK_PI);            // 最高値は 3.14
48:     headSlider->value(0.0);                // 初期値は 0
49:     headSlider->labelsize(12);             // ラベル文字のサイズを 12pt に
50:     headSlider->textsize(12);              // カウンタのサイズも 12 に
51:
52:     // ピッチ角用スライダー各種設定
```

```
53:     pitchSlider = new Fl_Value_Slider(130, 370, 180, 20, "Pitch Angle");
54:     pitchSlider->type(FL_HOR_NICE_SLIDER);
55:     pitchSlider->minimum(-FK_PI);
56:     pitchSlider->maximum(FK_PI);
57:     pitchSlider->value(0.0);
58:     pitchSlider->labelsize(12);
59:     pitchSlider->textsize(12);
60:
61:     // スケーリング用ローラー各種設定
62:     scaleRoller = new Fl_Roller(160, 410, 120, 20, "Scale");
63:     scaleRoller->type(FL_HORIZONTAL);       // ローラーは横向き
64:     scaleRoller->minimum(-100.0);
65:     scaleRoller->maximum(100.0);
66:     scaleRoller->value(0.0);
67:     scaleRoller->labelsize(12);
68:
69:     // 左側のマテリアル関係のボタンをグループ化しておく。
70:     materialGroup = new Fl_Group(10, 320, 100, 170);
71:     materialGroup->box(FL_THIN_UP_FRAME);   // ちょっと盛り上げる
72:
73:     // Ambient 用ラジオボタン
74:     ambientButton = new Fl_Check_Button(10, 320, 100, 30, "Ambient");
75:     ambientButton->type(FL_RADIO_BUTTON);
76:     ambientButton->down_box(FL_DIAMOND_DOWN_BOX);
77:
78:     // Diffuse 用ラジオボタン
79:     diffuseButton = new Fl_Check_Button(10, 350, 100, 30, "Diffuse");
80:     diffuseButton->type(FL_RADIO_BUTTON);
81:     diffuseButton->down_box(FL_DIAMOND_DOWN_BOX);
82:
83:     // Specular 用ラジオボタン
84:     specularButton = new Fl_Check_Button(10, 380, 100, 30, "Specular");
85:     specularButton->type(FL_RADIO_BUTTON);
86:     specularButton->down_box(FL_DIAMOND_DOWN_BOX);
87:
88:     // Emission 用ラジオボタン
89:     emissionButton = new Fl_Check_Button(10, 410, 100, 30, "Emission");
90:     emissionButton->type(FL_RADIO_BUTTON);
91:     emissionButton->down_box(FL_DIAMOND_DOWN_BOX);
92:
93:     // カラー設定呼出ボタン
94:     colorChooser = new Fl_Button(20, 450, 80, 30, "ColorChange");
95:     colorChooser->type(0);          // 普通のボタンとして扱う
```

```
 96:     colorChooser->labelsize(12);
 97:
 98:     // MaterialGroup のグループの終了
 99:     materialGroup->end();
100:
101:     // とりあえず Ambient にチェックを入れる。
102:     ambientButton->value(1);
103:
104:     // ファイル取得呼出ボタン
105:     fileOpenButton = new Fl_Button(120, 460, 90, 30, "File Open");
106:     fileOpenButton->type(0);
107:     fileOpenButton->labelsize(12);
108:
109:     // 終了ボタン
110:     exitButton = new Fl_Button(220, 460, 90, 30, "Exit");
111:     exitButton->type(0);
112:     exitButton->labelsize(12);
113:
114:     return;
115: }
116:
117: GUISet::~GUISet()
118: {
119:     return;
120: }
121:
122: string GUISet::getFileName(void)
123: {
124:     Fl_File_Chooser       *fc;
125:     const char            *fCharP;
126:     string                fileName;
127:
128:     fc = new Fl_File_Chooser(".", "*.wrl", Fl_File_Chooser::SINGLE,
129:                              "VRML File Select");
130:     fc->show();
131:     while(fc->visible()) {
132:         Fl::wait();
133:     }
134:
135:     fCharP = fc->value(1);
136:     if(fCharP == nullptr) {
137:         delete fc;
138:         return fileName;
```

```
139:     }
140:
141:     fileName = fCharP;
142:     delete fc;
143:     return fileName;
144: }
145:
146:
147: double GUISet::getHead(void)
148: {
149:     return headSlider->value();
150: }
151:
152: double GUISet::getPitch(void)
153: {
154:     return pitchSlider->value();
155: }
156:
157: double GUISet::getScale(void)
158: {
159:     // 10 の (scaleRoller->value()) 乗 を返す。
160:
161:     return pow(10.0, scaleRoller->value());
162: }
163:
164: int GUISet::materialSelect(void)
165: {
166:     // どのラジオボタンが選択されているかを取得
167:
168:     if(ambientButton->value() == 1) return 1;
169:     if(diffuseButton->value() == 1) return 2;
170:     if(specularButton->value() == 1) return 3;
171:     if(emissionButton->value() == 1) return 4;
172:
173:     return -1;
174: }
175:
176: bool GUISet::toggleColorChooser(void)
177: {
178:     if(colorChooser->value() == 1) return true;
179:     return false;
180: }
181:
```

```
182: bool GUISet::toggleFileOpen(void)
183: {
184:     if(fileOpenButton->value() == 1) {
185:         fileOpenButton->value(0);
186:         return true;
187:     }
188:     return false;
189: }
190:
191: bool GUISet::toggleExit(void)
192: {
193:     if(exitButton->value() == 1) return true;
194:     return false;
195: }
196:
197: void getMaterial(int index, fk_Material *mat, double *r, double *g, double *b)
198: {
199:     // Index によってマテリアルの各種属性を (R, G, B) に代入
200:
201:     switch(index) {
202:       case 1:
203:         *r = mat->getAmbient()->getR();
204:         *g = mat->getAmbient()->getG();
205:         *b = mat->getAmbient()->getB();
206:         break;
207:       case 2:
208:         *r = mat->getDiffuse()->getR();
209:         *g = mat->getDiffuse()->getG();
210:         *b = mat->getDiffuse()->getB();
211:         break;
212:       case 3:
213:         *r = mat->getSpecular()->getR();
214:         *g = mat->getSpecular()->getG();
215:         *b = mat->getSpecular()->getB();
216:         break;
217:       case 4:
218:         *r = mat->getEmission()->getR();
219:         *g = mat->getEmission()->getG();
220:         *b = mat->getEmission()->getB();
221:         break;
222:       default:
223:         break;
224:     }
```

```
225:     return;
226: }
227:
228: void setMaterial(int index, fk_Material *mat, double r, double g, double b)
229: {
230:     // Index によって (R, G, B) を各種マテリアルに設定
231:
232:     switch(index) {
233:       case 1:
234:         mat->setAmbient(r, g, b);
235:         break;
236:       case 2:
237:         mat->setDiffuse(r, g, b);
238:         break;
239:       case 3:
240:         mat->setSpecular(r, g, b);
241:         break;
242:       case 4:
243:         mat->setEmission(r, g, b);
244:         break;
245:       default:
246:         break;
247:     }
248:
249:     return;
250: }
251:
252: int main(int argc, char *argv[])
253: {
254:     double          colorR, colorG, colorB;
255:     fk_Material     material;
256:     fk_IndexFaceSet shape;
257:     fk_Light        light;
258:     fk_Model        shapeModel, lightModel[2], camera;
259:     fk_Scene        scene;
260:     string          fileName;
261:
262:     Fl_Window       mainWindow(320, 500, "VRML Viewer");
263:     fk_Window       viewWin(10, 10, 300, 300);
264:     GUISet          gui;
265:
266:     mainWindow.end();
267:
```

```
268:     fk_Material::initDefault();
269:
270:     // 光源の設定。今回は 2 つの平行光源
271:     lightModel[0].setShape(&light);
272:     lightModel[1].setShape(&light);
273:     lightModel[0].setMaterial(White);
274:     lightModel[1].setMaterial(White);
275:     lightModel[0].glFocus(-1.0, -1.0, 0.0);
276:     lightModel[1].glFocus(1.0, -1.0, 0.0);
277:
278:     // ソリッドモデルの初期マテリアルに Yellow を用いる。
279:     material = Yellow;
280:     shapeModel.setShape(&shape);
281:     shapeModel.setMaterial(material);
282:
283:     // 視点の設定
284:     camera.glTranslate(0.0, 100.0, 1000.0);
285:     camera.glFocus(0.0, 0.0, 0.0);
286:     camera.glUpvec(0.0, 1.0, 0.0);
287:
288:     // ディスプレイリストへソリッド、光源、視点を登録
289:     scene.entryModel(&shapeModel);
290:     scene.entryModel(&lightModel[0]);
291:     scene.entryModel(&lightModel[1]);
292:     scene.entryCamera(&camera);
293:
294:     // ViewWin にディスプレイリストを登録
295:     viewWin.setScene(&scene);
296:
297:     mainWindow.show();
298:     viewWin.show();
299:
300:     while(true) {
301:
302:         if(mainWindow.visible() == 0) {
303:             if(Fl::wait() == 0) {
304:                 break;
305:             } else {
306:                 continue;
307:             }
308:         }
309:         if(viewWin.drawWindow() == 0) break;
310:         if(Fl::check() == 0) break;
```

```
311:         if(viewWin.winOpenStatus() == false) continue;
312:
313:         // ライトを Y 軸中心に回転
314:         lightModel[0].glRotateWithVec(0.0, 0.0, 0.0, fk_Y, FK_PI/100.0);
315:         lightModel[1].glRotateWithVec(0.0, 0.0, 0.0, fk_Y, FK_PI/100.0);
316:
317:         // スライダーやローラーに従ってソリッドモデルの姿勢と大きさを決定
318:         shapeModel.glAngle(gui.getHead(), gui.getPitch(), 0.0);
319:         shapeModel.setScale(gui.getScale());
320:
321:         if(gui.toggleColorChooser() == true) {
322:             // カラー設定モード
323:             getMaterial(gui.materialSelect(), &material,
324:                         &colorR, &colorG, &colorB);
325:             fl_color_chooser("COLOR SET", colorR, colorG, colorB);
326:             setMaterial(gui.materialSelect(), &material,
327:                         colorR, colorG, colorB);
328:             shapeModel.setMaterial(material);
329:         }
330:
331:         if(gui.toggleFileOpen() == true) {
332:             // ファイル取得モード
333:             fileName = gui.getFileName();
334:             if(fileName != "") {
335:                 // キャンセルを押されたのでないなら
336:                 if(shape.readVRMLFile(fileName, true, false) == false) {
337:                     // VRML ファイルではなかったら
338:                     fl_alert(" is not VRML2.0 file.", fileName.c_str());
339:                 } else {
340:                     shapeModel.setMaterialMode(FK_PARENT_MODE);
341:                 }
342:             }
343:         }
344:
345:         if(gui.toggleExit() == true) {
346:             // 終了ボタンが押されたら while ループを抜ける。
347:             break;
348:         }
349:     }
350:
351:     return 0;
352: }
```

12.4 マルチウィンドウ

次のサンプルは、マルチウィンドウを用いて複数の視点を同時に表示するプログラムである。このプログラムは、同一シーンで異なる 3 つのウィンドウが表示される。それぞれの視点は左から順に、車の運転者視点、建物から見た視点、車を斜め後ろから見下ろした鳥瞰視点を表している。

このプログラムでは、116 〜 126 行目で 3 つのウィンドウを同時に処理している以外は、特に目新しい部分はない。このように、特別な処理を施す必要なくマルチウィンドウプログラムを作成することが可能である。

```
 1: #include <FK/FK.h>
 2:
 3: const double    BUILDWIDTH  = 25.0;     // 建物幅の基本単位
 4: const double    SPEED       = 2.0;      // 車のスピード
 5: const double    CIRCUITX    = 150.0;    // コースの X 方向幅
 6: const double    CIRCUITY    = 250.0;    // コースの Y 方向幅
 7: const double    EPS         = 0.001;    // 誤差判定用数値
 8:
 9: class Car {
10: private:
11:     fk_Model    carModel;       // 車全体モデル
12:     fk_Model    bodyModel;      // 車体モデル
13:     fk_Model    tireModel[4];   // 各タイヤモデル
14:     fk_Model    driverModel[2]; // 運転者モデル
15:     fk_Model    birdModel;      // 鳥瞰視点モデル
16:
17:     fk_Block    body;           // 車体形状
18:     fk_Circle   tire;           // タイヤ形状
19:     fk_Sphere   driver;         // 運転者形状
20:
21: public:
22:     Car(void) {     // コンストラクタ
23:         init();
24:     }
25:
26:     void        init(void);
27:     void        entryScene(fk_Scene *, bool);
28:     fk_Vector   getCarPosition(void);
29:     fk_Model *  getBirdModel(void);
30:     void        forward(void);
31:     void        rotate(fk_Model *, fk_Vector, fk_Vector);
32: };
33:
34: class World {
```

```
35: private:
36:     fk_Model    buildModel[6], groundModel, lightModel[2];
37:     fk_Block    buildShape, groundShape;
38:     fk_Light    lightShape;
39:
40:     void        defLight(int, double, double, double);
41:     void        makeBuild(int, double, double, double, fk_Material *);
42:
43: public:
44:     World(void) {
45:         init();
46:     }
47:
48:     void        init(void);
49:     void        entryScene(fk_Scene *, bool);
50: };
51:
52: int main(int argc, char *argv[])
53: {
54:     Car             carObj;
55:     World           worldObj;
56:     Fl_Window       mainWindow(940, 320, "FK TEST");
57:     fk_Scene        carViewScene, buildViewScene, birdViewScene;
58:     fk_Window       carViewWindow(10, 10, 300, 300);
59:     fk_Window       buildViewWindow(320, 10, 300, 300);
60:     fk_Window       birdViewWindow(630, 10, 300, 300);
61:     fk_Color        bgColor(0.2, 0.7, 1.0);
62:
63:     fk_Model        buildViewModel, birdViewModel;
64:
65:     mainWindow.end();
66:     fk_Material::initDefault();
67:
68:     carObj.init();
69:     worldObj.init();
70:
71:
72:     // 各ウィンドウにバックグラウンドカラー設定
73:     carViewScene.setBGColor(bgColor);
74:     buildViewScene.setBGColor(bgColor);
75:     birdViewScene.setBGColor(bgColor);
76:
77:
```

```
78:     // 各モデルをディスプレイリストに登録
79:     worldObj.entryScene(&carViewScene, false);
80:     worldObj.entryScene(&buildViewScene, true);
81:     worldObj.entryScene(&birdViewScene, false);
82:
83:     carObj.entryScene(&carViewScene, false);
84:     carObj.entryScene(&buildViewScene, true);
85:     carObj.entryScene(&birdViewScene, true);
86:
87:     // 建物ウィンドウの視点設定
88:     buildViewModel.glMoveTo(-250.0, 100.0, 100.0);
89:     buildViewModel.glFocus(carObj.getCarPosition());
90:     buildViewModel.glUpvec(0.0, 0.0, 1.0);
91:     buildViewScene.entryCamera(&buildViewModel);
92:
93:     // 鳥瞰ウィンドウの視点設定
94:     birdViewScene.entryCamera(carObj.getBirdModel());
95:
96:     // ウィンドウへディスプレイリストを登録
97:     carViewWindow.setScene(&carViewScene);
98:     buildViewWindow.setScene(&buildViewScene);
99:     birdViewWindow.setScene(&birdViewScene);
100:
101:    mainWindow.show();
102:    carViewWindow.show();
103:    buildViewWindow.show();
104:    birdViewWindow.show();
105:
106:    while(true) {
107:
108:        if(mainWindow.visible() == 0) {
109:            if(Fl::wait() == 0) {
110:                break;
111:            } else {
112:                continue;
113:            }
114:        }
115:
116:        if(carViewWindow.drawWindow() == 0 ||
117:           buildViewWindow.drawWindow() == 0 ||
118:           birdViewWindow.drawWindow() == 0) break;
119:
120:        if(Fl::check() == 0) break;
```

```
121:
122:         if(carViewWindow.winOpenStatus() == false ||
123:            buildViewWindow.winOpenStatus() == false ||
124:            birdViewWindow.winOpenStatus() == false) {
125:             continue;
126:         }
127:
128:         carObj.forward();
129:         buildViewModel.glFocus(carObj.getCarPosition());
130:         buildViewModel.glUpvec(0.0, 0.0, 1.0);
131:     }
132:
133:     return 0;
134: }
135:
136: void Car::init(void)
137: {
138:     int     i;
139:
140:     body.setSize(7.0, 6.0, 20.0);
141:     tire.setRadius(2.0);
142:     tire.setDivide(2);
143:     driver.setRadius(2.0);
144:     driver.setDivide(2);
145:
146:     bodyModel.setShape(&body);
147:     bodyModel.glMoveTo(0.0, 5.0, 0.0);
148:     bodyModel.setMaterial(Yellow);
149:     bodyModel.setParent(&carModel);
150:
151:
152:     tireModel[0].glMoveTo(-4.0, 1.0, -8.0);
153:     tireModel[0].glVec(1.0, 0.0, 0.0);
154:     tireModel[1].glMoveTo(4.0, 1.0, -8.0);
155:     tireModel[1].glVec(-1.0, 0.0, 0.0);
156:     tireModel[2].glMoveTo(-4.0, 1.0, 8.0);
157:     tireModel[2].glVec(1.0, 0.0, 0.0);
158:     tireModel[3].glMoveTo(4.0, 1.0, 8.0);
159:     tireModel[3].glVec(-1.0, 0.0, 0.0);
160:
161:     for(i = 0; i < 4; i++) {
162:         tireModel[i].setShape(&tire);
163:         tireModel[i].setMaterial(Gray2);
```

```
164:         tireModel[i].setParent(&carModel);
165:     }
166:
167:     driverModel[0].setShape(&driver);
168:     driverModel[1].setShape(&driver);
169:     driverModel[0].glMoveTo(-2.0, 10.0, 0.0);
170:     driverModel[1].glMoveTo(2.0, 10.0, 0.0);
171:     driverModel[0].setMaterial(Cream);
172:     driverModel[1].setMaterial(Cream);
173:     driverModel[0].setSmoothMode(true);
174:     driverModel[1].setSmoothMode(true);
175:     driverModel[0].setParent(&carModel);
176:     driverModel[1].setParent(&carModel);
177:
178:     birdModel.glMoveTo(0.0, 100.0, 200.0);
179:     birdModel.glFocus(0.0, 5.0, 0.0);
180:     birdModel.glUpvec(0.0, 1.0, 0.0);
181:     birdModel.setParent(&carModel);
182:
183:     carModel.glMoveTo(CIRCUITX, CIRCUITY, 0.0);
184:     carModel.glVec(0.0, -1.0, 0.0);
185:     carModel.glUpvec(0.0, 0.0, 1.0);
186:
187:     return;
188: }
189:
190: void Car::entryScene(fk_Scene *scene, bool viewFlag)
191: {
192:     int     i;
193:
194:     scene->entryModel(&bodyModel);
195:
196:     for(i = 0; i < 4; i++) {
197:         scene->entryModel(&tireModel[i]);
198:     }
199:
200:     scene->entryModel(&driverModel[0]);
201:     if(viewFlag == true) {
202:         scene->entryModel(&driverModel[1]);
203:     } else {
204:         scene->entryCamera(&driverModel[1]);
205:     }
206:
```

```
207:     return;
208: }
209:
210: fk_Vector Car::getCarPosition(void)
211: {
212:     return carModel.getPosition();
213: }
214:
215: fk_Model * Car::getBirdModel(void)
216: {
217:     return &birdModel;
218: }
219:
220: void Car::forward(void)
221: {
222:     fk_Vector   carPosition, carVelocity;
223:     fk_Vector   Xplus(1.0, 0.0, 0.0), Xminus(-1.0, 0.0, 0.0);
224:     fk_Vector   Yplus(0.0, 1.0, 0.0), Yminus(0.0, -1.0, 0.0);
225:     double      X = CIRCUITX;
226:     double      Y = CIRCUITY;
227:
228:     carPosition = carModel.getPosition();
229:     carVelocity = carModel.getVec();
230:     carModel.loTranslate(0.0, 0.0, -SPEED); // 前進
231:
232:     // サーキットの外にでた場合、回転する。
233:     if(carPosition.x > X) {
234:         rotate(&carModel, carVelocity, Xplus);
235:     }
236:     if(carPosition.x < -X) {
237:         rotate(&carModel, carVelocity, Xminus);
238:     }
239:     if(carPosition.y > Y) {
240:         rotate(&carModel, carVelocity, Yplus);
241:     }
242:     if(carPosition.y < -Y) {
243:         rotate(&carModel, carVelocity, Yminus);
244:     }
245:
246:     return;
247: }
248:
249: void Car::rotate(fk_Model *model, fk_Vector velocity, fk_Vector orgVec)
```

```
250: {
251:     // velocity と orgVec の内積値が正、つまり角度が 90 度以内の場合回転
252:     if(velocity * orgVec > EPS) {
253:         model->loAngle(FK_PI/100.0, 0.0, 0.0);
254:     }
255:     return;
256: }
257:
258: void World::defLight(int lightID, double x, double y, double z)
259: {
260:     lightModel[lightID].setShape(&lightShape);
261:     lightModel[lightID].setMaterial(White);
262:     lightModel[lightID].glTranslate(0.0, 0.0, 0.0);
263:     lightModel[lightID].glFocus(x, y, z);
264:
265:     return;
266: }
267:
268: void World::makeBuild(int buildID, double x, double y,
269:                       double heightScale, fk_Material *buildMat)
270: {
271:     buildModel[buildID].setShape(&buildShape);
272:     buildModel[buildID].setScale(heightScale, fk_Z);
273:     buildModel[buildID].glMoveTo(x, y, (BUILDWIDTH * heightScale)/2.0);
274:     buildModel[buildID].setMaterial(*buildMat);
275:
276:     return;
277: }
278:
279: void World::init(void)
280: {
281:     // 照明の設定
282:     defLight(0, 1.0, 1.0, -1.0);
283:     defLight(1, -1.0, -1.0, -1.0);
284:
285:     // 建物の設定
286:     buildShape.setSize(BUILDWIDTH, BUILDWIDTH, BUILDWIDTH);
287:     makeBuild(0, -250.0, 100.0, 5.0, &Red);
288:     makeBuild(1, -150.0, 400.0, 2.0, &DimYellow);
289:     makeBuild(2, 50.0, 250.0, 4.0, &Blue);
290:     makeBuild(3, 300.0, 200.0, 3.0, &Gray1);
291:     makeBuild(4, 250.0, -250.0, 0.5, &Green);
292:     makeBuild(5, -50.0, -350.0, 6.0, &Orange);
```

```
293:
294:     // 地面の設定
295:     groundShape.setSize(1000, 1000, 2.0);
296:     groundModel.setShape(&groundShape);
297:     groundModel.glTranslate(0.0, 0.0, -1.0);
298:     groundModel.setMaterial(Brown);
299:
300:     return;
301: }
302:
303: void World::entryScene(fk_Scene *scene, bool buildFlag)
304: {
305:     int     i;
306:
307:     scene->entryModel(&groundModel);
308:
309:     for(i = 0; i < 2; i++) {
310:         scene->entryModel(&lightModel[i]);
311:     }
312:
313:     for(i = 0; i < 6; i++) {
314:         scene->entryModel(&buildModel[i]);
315:     }
316:
317:     if(buildFlag == true) scene->removeModel(&buildModel[0]);
318:     return;
319: }
```

12.5 形状の簡易表示とアニメーション

次のサンプルは、fk_ShapeViewer クラスの典型的な利用法を示したものである。

- まず、13 〜 19 行目で 11 行 11 列の行列として並んでいる状態の座標を計算している。その際、$z = \frac{x^2 - y^2}{10}$ として z 成分は計算されている。
- 次に、22 〜 29 行目でインデックスフェースセットを表す配列を作成している。インデックスフェースセットに関しては、第 5.1 節を参照すること。
- 32 行目で実際に形状を生成する。この部分の解説も、第 5.1 節に記述がある。
- 35 行目では、作成した形状を描画形状として登録している。ID には 0 が選択されているが、特にどのような整数値でも構わない。
- 38,39 行目では、表裏の両面及び稜線や頂点を描画するように設定している。
- 42 行目では while ループ中で描画が行われるよう記述されている。これにより、43 〜 56 行目が実行される度に描画処理が行われるようになる。
- 46,47 行目では、アニメーションの際の頂点移動量が計算されている。移動は z 方向のみ行われ、移動量は $\sin \frac{counter+10j}{5\pi}$ である。counter はループの度に 56 行目で 10 ずつ追加されているので、描画の度に移動量が異なるこ

とになる。

- 50 行目で初期位置に移動量が足され、53 行目で実際に各頂点を移動している。

```
 1: #include <FK/FK.h>
 2:
 3: int main()
 4: {
 5:     fk_ShapeViewer   viewer(300, 360);
 6:     fk_IndexFaceSet  shape;
 7:     fk_Vector        pos[121], moveVec, movePos;
 8:     int              IFSet[4*100];
 9:     int              i, j, counter;
10:     double           x, y;
11:
12:     // 各頂点位置の設定
13:     for(i = 0; i <= 10; i++) {
14:         for(j = 0; j <= 10; j++) {
15:             x = double(i - 5);
16:             y = double(j - 5);
17:             pos[i*11 + j].set(x, y, (x*x - y*y)/10.0);
18:         }
19:     }
20:
21:     // インデックスフェースセットの生成
22:     for(i = 0; i < 10; i++) {
23:         for(j = 0; j < 10; j++) {
24:             IFSet[(i*10 + j)*4 + 0] = i*11 + j;
25:             IFSet[(i*10 + j)*4 + 1] = (i+1)*11 + j;
26:             IFSet[(i*10 + j)*4 + 2] = (i+1)*11 + j+1;
27:             IFSet[(i*10 + j)*4 + 3] = i*11 + j+1;
28:         }
29:     }
30:
31:     // 形状の生成
32:     shape.makeIFSet(100, 4, IFSet, 121, pos);
33:
34:     // fk_ShapeViewer へ形状を設定
35:     viewer.setShape(0, &shape);
36:
37:     // 各種設定
38:     viewer.setDrawMode(FK_FRONTBACK_POLYMODE | FK_LINEMODE);
39:     viewer.setScale(10.0);
40:
41:     counter = 0;
```

```
42:     while(viewer.draw() == true) {
43:         for(i = 0; i <= 10; i++) {
44:             for(j = 0; j <= 10; j++) {
45:                 // 各頂点の移動量計算
46:                 moveVec.set(0.0, 0.0,
47:                             sin(double(counter+j*40)*0.05/FK_PI));
48:
49:                 // 各頂点を移動場所を計算
50:                 movePos = moveVec + pos[i*11 + j];
51:
52:                 // 各頂点を実際に移動
53:                 shape.moveVPosition(i*11 + j, movePos);
54:             }
55:         }
56:         counter += 10;
57:     }
58:
59:     return 0;
60: }
```

12.6 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、getShape() 関数を利用して 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。

- 8 〜 15 行目は、fk_ParticleSet クラスを継承した「MyParticle」というクラスを定義している。定義の際、仮想関数である genMethod()、allMethod()、indivMethod() の各関数の宣言を必ず行う。
- 19 〜 29 行目は MyParticle クラスのコンストラクタである。ここで、パーティクルの初期設定を行う。
- 21 行目の setMaxSize() 関数は fk_ParticleSet クラスのメンバ関数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、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。
- 23,24 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、allMethod() や indivMethod() の記述は無視される。
- 27 行目は、パーティクル全体の色パレットを設定しているものである。ここでは緑色を ID 1 として登録している。
- 33 〜 40 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オブジェクトが入っており、これに対して様々な設定を行う。36 行目では初期位置を、38 行目では色 ID を設定している。
- 43 〜 54 行目では、allMethod() メンバ関数を再定義している。allMethod() 関数には、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。

- 57 〜 82 行目では、indivMethod() メンバ関数を再定義している。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、65 〜 74 行目で速度ベクトルの入力を行っている。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) での水流を表す偏微分方程式は以下のようなものである。

$$\frac{\partial}{\partial t}\mathbf{P} = \mathbf{V} + \frac{R^3}{2}\left(\frac{\mathbf{V}}{r^3} - \frac{3\mathbf{V}\cdot\mathbf{P}}{r^5}\mathbf{P}\right)$$

ただし、

$$\mathbf{V} = (-V_x, 0, 0), \qquad \mathbf{P} = (x, y, 0), \qquad r = |\mathbf{P}|$$

今回は、$V_x = 0.2$(60 行目の「water」変数)、$R = 15$(61 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、74 行目で設定している。

- 77 〜 79 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 78 行目で削除を行っている。
- 93 行目では、パーティクル集合を fk_ShapeViewer で表示するために getShape() 関数を用いている。
- 98 行目にあるように、handle() メンバ関数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。

```
 1: #include <FK/FK.h>
 2:
 3: double myRandom(void)
 4: {
 5:     return double(rand())/double(RAND_MAX);
 6: }
 7:
 8: class MyParticle: public fk_ParticleSet {
 9: protected:
10:     void        genMethod(fk_Particle *);
11:     void        allMethod(void);
12:     void        indivMethod(fk_Particle *);
13: public:
14:     MyParticle(void);
15: };
16:
17: // コンストラクタ。
18: // ここに、様々な設定を記述しておく。
19: MyParticle::MyParticle(void)
20: {
21:     setMaxSize(1000);   // パーティクルの最大数設定。
22:     srand(time(0));     // 乱数の初期化。
23:     setIndivMode(true); // 個別処理 (indivMethod) を ON にしておく。
24:     setAllMode(true);   // 全体処理 (allMethod) を ON にしておく。
25:
26:     // パレットに色を設定しておく。
```

```
27:     setColorPalette(1, 0.0, 1.0, 0.6);
28:     return;
29: }
30:
31: // ここにパーティクル生成時の処理を記述する。
32: // p に新たなパーティクル要素が入っている。
33: void MyParticle::genMethod(fk_Particle *p)
34: {
35:     // 生成時の位置を設定
36:     p->setPosition(50.0, myRandom()*50.0 - 25.0, myRandom()*50.0 - 25.0);
37:     // パーティクルの色 ID を設定
38:     p->setColorID(1);
39:     return;
40: }
41:
42: // ここに毎ループ時の全体処理処理を記述する
43: void MyParticle::allMethod(void)
44: {
45:     for(int i = 0; i < 5; i++) {
46:         if(myRandom() < 0.3) {
47:             // 新たなパーティクルを生成。
48:             // 生成時に genMethod() が呼ばれる。
49:             newParticle();
50:         }
51:     }
52:
53:     return;
54: }
55:
56: // ここに毎ループ時の各パーティクルへの処理を記述する。
57: void MyParticle::indivMethod(fk_Particle *p)
58: {
59:     fk_Vector       pos, vec, tmp1, tmp2;
60:     fk_Vector       water(-0.2, 0.0, 0.0);
61:     double          R = 15.0;
62:     double          r;
63:
64:     // パーティクルの位置を取得。
65:     pos = p->getPosition();
66:     pos.z = 0.0;
67:     r = pos.dist(); // |p| を r に代入。
68:
69:     // パーティクルの速度ベクトルを計算
```

```
70:     tmp1 = water/(r*r*r);
71:     tmp2 = ((3.0 * (water * pos))/(r*r*r*r*r)) * pos;
72:     vec = water + ((R*R*R)/2.0) * (tmp1 - tmp2);
73:     // パーティクルの速度ベクトルを代入
74:     p->setVelocity(vec);
75:
76:     // パーティクルの x 成分が -50 以下になったら消去。
77:     if(pos.x < -50.0) {
78:         removeParticle(p);
79:     }
80:
81:     return;
82: }
83:
84: int main()
85: {
86:     fk_ShapeViewer      viewer(600, 600);
87:     MyParticle          particle;
88:     fk_Prism            prism(40, 15.0, 15.0, 50.0);
89:
90:     viewer.setShape(3, &prism);
91:     viewer.setPosition(3, 0.0, 0.0, 25.0);
92:     viewer.setDrawMode(3, FK_POLYMODE);
93:     viewer.setShape(2, particle.getShape());
94:     viewer.setDrawMode(2, FK_POINTMODE);
95:     viewer.setScale(10.0);
96:
97:     while(viewer.draw() == true) {
98:         particle.handle(); // パーティクルを 1 ステップ実行する。
99:     }
100:    return 0;
101:}
```

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	(0.3, 0.3, 0.3)	(0.0, 0.0, 0.7)	(1.0, 1.0, 1.0)	(30.0)
Brown	(0.2, 0.1, 0.0)	(0.35, 0.15, 0.0)	(1.0, 1.0, 1.0)	(30.0)
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)	(1.0, 1.0, 1.0)	(30.0)
Cream	(0.3, 0.3, 0.3)	(0.8, 0.7, 0.6)	(1.0, 1.0, 1.0)	(30.0)
Cyan	(0.3, 0.3, 0.3)	(0.0, 0.6, 0.6)	(1.0, 1.0, 1.0)	(30.0)
DarkBlue	(0.1, 0.1, 0.4)	(0.0, 0.0, 0.25)	(1.0, 1.0, 1.0)	(30.0)
DarkGreen	(0.1, 0.4, 0.1)	(0.0, 0.2, 0.0)	(1.0, 1.0, 1.0)	(30.0)
DarkPurple	(0.3, 0.1, 0.3)	(0.3, 0.0, 0.3)	(1.0, 1.0, 1.0)	(30.0)
DarkRed	(0.2, 0.0, 0.0)	(0.4, 0.0, 0.0)	(1.0, 1.0, 1.0)	(30.0)
DarkYellow	(0.3, 0.3, 0.3)	(0.4, 0.3, 0.0)	(1.0, 1.0, 1.0)	(30.0)
DimYellow	(0.18, 0.14, 0.0)	(0.84, 0.86, 0.07)	(0.92, 0.82, 0.49)	(30.0)
Flesh	(0.3, 0.3, 0.3)	(0.8, 0.6, 0.4)	(1.0, 1.0, 1.0)	(30.0)
GlossBlack	(0.0, 0.0, 0.0)	(0.04, 0.04, 0.04)	(1.0, 1.0, 1.0)	(30.0)
GrassGreen	(0.0, 0.1, 0.0)	(0.0, 0.7, 0.0)	(0.47, 0.98, 0.49)	(30.0)
Gray1	(0.3, 0.3, 0.3)	(0.6, 0.6, 0.6)	(0.1, 0.1, 0.1)	(30.0)
Gray2	(0.3, 0.3, 0.3)	(0.2, 0.2, 0.2)	(0.1, 0.1, 0.1)	(30.0)
Green	(0.3, 0.3, 0.3)	(0.0, 0.5, 0.0)	(1.0, 1.0, 1.0)	(30.0)
HolidaySkyBlue	(0.01, 0.22, 0.4)	(0.2, 0.66, 0.92)	(0.47, 0.74, 0.74)	(30.0)
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	(0.3, 0.3, 0.3)	(0.4, 0.4, 0.9)	(1.0, 1.0, 1.0)	(30.0)
LightCyan	(0.1, 0.2, 0.2)	(0.0, 0.5, 0.5)	(0.2, 0.2, 0.2)	(60.0)
LightGreen	(0.3, 0.3, 0.3)	(0.5, 0.7, 0.3)	(1.0, 1.0, 1.0)	(30.0)
LightViolet	(0.3, 0.3, 0.3)	(0.5, 0.4, 0.9)	(1.0, 1.0, 1.0)	(30.0)
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)	(30.0)
Orange	(0.3, 0.3, 0.3)	(0.8, 0.3, 0.0)	(0.2, 0.2, 0.2)	(30.0)
PaleBlue	(0.3, 0.3, 0.3)	(0.5, 0.7, 0.7)	(1.0, 1.0, 1.0)	(30.0)
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)	(1.0, 1.0, 1.0)	(30.0)
Purple	(0.3, 0.3, 0.3)	(0.7, 0.0, 0.7)	(1.0, 1.0, 1.0)	(30.0)
Raw	(1.0, 1.0, 1.0)	(1.0, 1.0, 1.0)	(0.0, 0.0, 0.0)	(0.0)
Red	(0.3, 0.3, 0.3)	(0.7, 0.0, 0.0)	(1.0, 1.0, 1.0)	(30.0)
TrueWhite	(1.0, 1.0, 1.0)	(1.0, 1.0, 1.0)	(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	(0.3, 0.3, 0.3)	(0.4, 0.0, 0.8)	(1.0, 1.0, 1.0)	(30.0)
White	(0.3, 0.3, 0.3)	(1.0, 1.0, 1.0)	(0.1, 0.1, 0.1)	(30.0)
WhiteLight	(0.0, 0.0, 0.0)	(1.0, 1.0, 1.0)	(1.0, 1.0, 1.0)	(30.0)
Yellow	(0.3, 0.3, 0.3)	(0.8, 0.6, 0.0)	(1.0, 1.0, 1.0)	(30.0)