Fine Kernel ToolKit System
チュートリアル

Version 2.9.0
(Manual Revision 1)

FineKernel Project
1999 - 2014

第1章　初めてのC++、初めてのFK

1.1　プロジェクト作成とビルド

1.1.1　プロジェクトの作り方

1. Visual Studio 2013 (以下「VS」) を起動する。
2. [ファイル] → [新しいプロジェクト]
3. 表示されたダイアログ左側のツリーメニューで [インストール済み] → [テンプレート] → [Visual C++] → [FK] を選択。
4. 名前は「Hello」、「場所」は以下のようにしておく。

```
c:\users\User\documents\proj-semi
```

5. 「OK」を押す。
6. 「main 関数のみ」を選択し、「コンソール出力を使う」にチェックを入れる。
7. 「完了」を押す。

1.1.2　ソースプログラムの編集

VS 画面の右側にある「ソリューションエクスプローラー」の中にある、「main.cpp」をダブルクリックする。すると、main.cpp というファイルを編集する状態となる。この状態で、以下のようなプログラムを入力してみよう。

最初のプログラム

```
1: #include <iostream>
2:
3: using namespace std;
4:
5: int main(int, char **)
6: {
7:     cout << "Hello World" << endl;
8:     return 0;
9: }
```

1.1.3 ビルドと実行

書いたプログラムは、まずはPCが実行可能な状態に「翻訳」する必要がある。これを「コンパイル」または「ビルド」と呼ぶ。ビルドするには、[ビルド] → [ソリューションのビルド] を選択する。うまく行った場合は「正常終了」という文字の左側が1になっているが、失敗すると「失敗」という文字の左側が1になっている。正常終了となるまで修正を繰り返していく必要がある。

実行は、[デバッグ] → [デバッグなしで開始] を選択する。

1.2 最も簡単な3Dプログラム

改めて、新たなプロジェクトを作成する。プロジェクト名はなんでもよいが、日本語やスペースは避けること。作成したら以下のプログラムを入力してビルドする。

ウィンドウの表示

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow window;
 6:
 7:     window.setSize(800, 600);
 8:     window.setBGColor(0.3, 0.6, 0.8);
 9:     window.open();
10:
11:     while(window.update() == true) {
12:     }
13:
14:     return 0;
15: }
```

解説

- 1行目から4行目までは詳細は解説しませんが、当面この部分は「最初に書いておくべき呪文」と思ってください。
- C++のプログラムでは、「変数」と呼ばれるものを多く用います。変数は、主に
  - なんらかの情報を記録するためのもの
  - 特定の働きを命令するための道具

  として扱われます。各変数は必ず何らかの「型」を持ちます。5行目は、「fk_AppWindow」という型である「window」という変数を用意しているところです。このように、C++では「型名 変数名」という単位で変数を準備します。

- 基本的に、C++のプログラムは行毎に一つの指示を書いていきますが、命令の区切りは「;(セミコロン)」です。
- 7 行目は、window 変数を通して「setSize」という**関数**を用いて、ウィンドウのサイズを指定しています。このように、

  「変数名 ．関数名 (引数 1，引数 2，...);」

  という形式で様々な指示を記述していきます。**引数**というのは、関数に対して入力になる値のことです。今回の場合、1 番目の引数がウィンドウの横幅、2 番目が縦幅を表します。setSize 関数は fk_AppWindow 型に備わっている機能です。
- 8 行目は、ウィンドウの背景色を指定しています。光の三原色である赤、緑、青の強さを、最大値が 1, 最小値が 0 で指定します。
- 9 行目は、これまで指定した指示に従って実際にウィンドウを開く命令です。
- 11〜12 行目は、書いておかないといきなりウィンドウが消えてしまうのですが、現在はそれ以上の意味はありません。アニメーションをプログラムするときに、この中身を記述していきます。
- 14 行目の「`return 0;`」は、プログラムを終了する命令です。最後は数字の「ゼロ」であり、アルファベットの「オー」ではありませんので注意してください。

箱の追加

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:
 7:     window.setSize(800, 600);
 8:     window.setBGColor(0.3, 0.6, 0.8);
 9:     window.open();
10:
11:     fk_Block    block(10.0, 10.0, 10.0);
12:     fk_Model    model1;
13:     model1.setShape(&block);
14:     model1.setMaterial(Yellow);
15:     window.entry(model1);
16:
17:     while(window.update() == true) {
18:     }
19:
20:     return 0;
21: }
```

解説

- 11 行目は、新たに「fk_Block」という型の変数「block」を準備しています。これは直方体を生成するための型です。ここは window と違って変数名の後に括弧で 3 つの引数が羅列されています。このように、変数名の後に引数を入力することもあります。ここでの引数の意味はそれぞれ x, y, z 方向の大きさです。
- 12 行目の「fk_Model」は、FK の中で重要な概念となる「モデル」を作成するための型なのですが、詳細は次回以降に行います。ここでは、「物体を表示するために必要なもの」としておきます。13 行目で、このモデルの形状が 11 行目で準備した物体であるように設定を行っています。14 行目では、物体の色が黄色 (Yellow) となるように設定しています。15 行目は、このモデルを実際に表示するものとしてウィンドウにエントリー (entry) しています。

1.3 複数オブジェクトの表示

複数オブジェクトの表示

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:
 7:     window.setSize(800, 600);
 8:     window.setBGColor(0.3, 0.6, 0.8);
 9:     window.open();
10:
11:     fk_Block    block(10.0, 10.0, 10.0);
12:     fk_Model    model1;
13:     model1.setShape(&block);
14:     model1.setMaterial(Yellow);
15:     window.entry(model1);
16:
17:     fk_Sphere   sphere(8, 5.0);
18:     fk_Model    model2;
19:     model2.setShape(&sphere);
20:     model2.setMaterial(Red);
21:     model2.glMoveTo(20.0, 10.0, 0.0);
22:     window.entry(model2);
23:
24:     while(window.update() == true) {
25:     }
26:
27:     return 0;
28: }
```

解説

- 17 行目の「fk_Sphere」は球を生成するための型です。この変数での引数は、1 番目が分割数、2 番目が半径を表します。分割数は、大きければ大きいほど細かくリアルになっていきますが、表示は遅くなっていきます。
- 18 行目で新たな fk_Model 型の変数「model2」を準備しています。FK では、複数のモデルを表示する場合にはその個数のモデルを準備しておく必要があります。23 行目でエントリーを行っていますが、これと 15 行目により、model1 と model2 の両方が同時に表示されることになります。
- 21 行目の「glMoveTo」関数は、モデルを平行移動する関数です。従って、この球は $(20, 10, 0)$ の位置に移動することになります。

1.4 物体の向きの制御

箱の向きを変更する

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:
 7:     window.setSize(800, 600);
 8:     window.setBGColor(0.3, 0.6, 0.8);
 9:     window.open();
10:
11:     fk_Block    block(10.0, 10.0, 10.0);
12:     fk_Model    model1;
13:     model1.setShape(&block);
14:     model1.setMaterial(Yellow);
15:     model1.glVec(1.0, 1.0, 0.0);
16:     model1.glMoveTo(-20.0, -20.0, 0.0);
17:     window.entry(model1);
18:
19:     fk_Sphere   sphere(8, 5.0);
20:     fk_Model    model2;
21:     model2.setShape(&sphere);
22:     model2.setMaterial(Red);
23:     window.entry(model2);
24:
25:     while(window.update() == true) {
26:     }
27:
28:     return 0;
29: }
```

解説

- 15 行目の「glVec」関数は、モデルの向いている方向を変更する関数です。今回の場合は $(1, 1, 0)$ の方向に向かせることにより、結果的に直方体が斜めを向くかたちになります。ちなみに、モデルは初期状態として $-z$ 方向を向いています。長細い棒を作成した上で、その棒の向きをベクトルで指定したい場合は、z 方向に長い直方体を生成しておく必要があります。

1.5 練習課題

問題 1-1

箱を 5 個並んだ様子を描画せよ。

問題 1-2

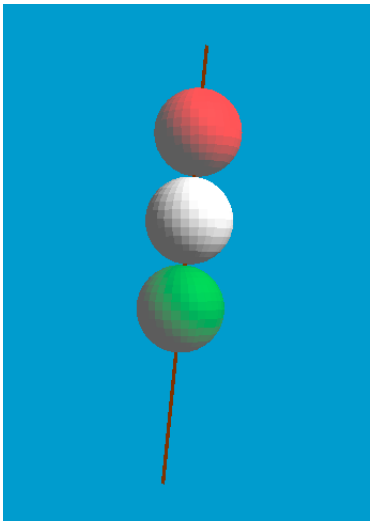
白と黒の球が交互に並ぶように配置せよ。(色を変更するには setMaterial 関数で白は「White」、黒は「GlossBlack」を指定する。)

問題 1-3

串 (Brown) に刺さった三色団子 (Red、White、Green) を作成せよ。

問題 1-4

串がわずかに斜めに傾いた三色団子を作成せよ。



第2章　配列と繰り返し

2.1　配列の作成と利用

配列

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        block(10.0, 10.0, 10.0);
 7:     fk_Model        model[2];
 8:
 9:     model[0].setShape(&block);
10:     model[0].setMaterial(Yellow);
11:     model[0].glMoveTo(-20.0, -20.0, 0.0);
12:     window.entry(model[0]);
13:
14:     model[1].setShape(&block);
15:     model[1].setMaterial(Red);
16:     model[1].glMoveTo(20.0, 20.0, 0.0);
17:     window.entry(model[1]);
18:
19:     window.setSize(800, 600);
20:     window.setBGColor(0.3, 0.6, 0.8);
21:     window.open();
22:
23:     while(window.update() == true) {
24:     }
25:
26:     return 0;
27: }
```

解説

- 7 行目で、これまでの変数の準備 (これを「変数の**宣言**」といいます) の際に、カギ括弧で 2 という数字を挟んでいます。このように、変数宣言の際にカギ括弧で数字を挟んでいる形式の変数を「**配列**」と言います。配列は、沢山の変数をひとまとまりで準備することができます。今回のように「`model[2]`」とした場

合は、「model[0]」と「model[1]」という変数を準備したことと同じ意味となります。このように、「配列名 [n]」と宣言した場合、配列名 [0] から配列名 [n-1] の n 個の変数が利用できるようになります。

- 9 行目と 14 行目では、同じ block を異なるモデルに設定しています。このように、FK では異なるモデルに同じ形状を設定することが可能です。例えば、カーレースゲームでは同じ形状のモデルに対して色を替えた車を複数同時に扱いたいものです。このようなとき、FK では各車のモデルを用意し、それぞれのモデルに同じ形状を設定するということができます。モデルはそれぞれ固有の色や位置を持つことができます。今回も、色や位置が二つのモデルで異なっていることがわかります。

2.2 for 文によるループ

for 文

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        block(1.0, 1.0, 1.0);
 7:     fk_Model        model[10];
 8:     int             i;
 9:
10:     for(i = 0; i < 10; i++) {
11:         model[i].setShape(&block);
12:         model[i].setMaterial(Yellow);
13:         model[i].glMoveTo(i*5, 0.0, 0.0);
14:         window.entry(model[i]);
15:     }
16:
17:     window.setSize(800, 600);
18:     window.setBGColor(0.3, 0.6, 0.8);
19:     window.open();
20:
21:     while(window.update() == true) {
22:     }
23:
24:     return 0;
25: }
```

解説

- 8 行目、int という型の「i」という変数が宣言されています。この変数については、これまでの変数と違って「情報を格納する」という目的で変数を利用していきます。int 型は、整数の値を 1 個だけ格納することができます。

- 10 行目の「for」というのは変数ではなく、C++の中で「ループ」を扱うための機能です。書式は以下の様になっています。

```
for(初期処理; 繰り返し条件; 繰り返し処理) {
        実行文 1;
        実行文 2;
           :
}
```

for 文では、以下の様な処理が行われます。

1. まず、「初期処理」を実行する。今回の場合は i に 0 が代入される。
2. 次に、「繰り返し条件」で条件判定が行われる。ここでは、i が 10 未満かどうかを判定する。もしこの判定に合格した場合、「{」と「}」で囲まれた「実行文」が逐次実行されていく。
3. 全ての実行文が実行された時点で、「繰り返し処理」が実行される。今回の場合は「i++;」と書いてあるが、これは「i に 1 を足す」という意味である。例えば、i に 0 が入っていたときは 1 に、3 が入っていたときは 4 になる。
4. 後は、ステップ 2 に戻って同様のことが行われていく。もし判定に合格しなかった場合は、「}」の外を実行していく。(MyLoop.java の場合、} の外には特に記述がないので、実行を終了する。)

結果として、この for 文の中身は 10 回実行され、また実行の度に i が 1 ずつ足されていくということになっています。

- 11〜13 行目では、model 配列のカギ括弧の中の数字 (これを**添字**と言います) が、直接の数値指定ではなく変数 i を用いています。このように、添字には変数や変数の計算結果 (例えば i+3 や i*2 など) を入れることが可能です。

- 13 行目の glMoveTo 関数では、第一引数として i に 5 倍した数値を入力しています。結果的に、最初の箱は $(0, 0, 0)$、二個目の箱は $(5, 0, 0)$、三個目の箱は $(10, 0, 0)$ というように、それぞれ別の場所に移動することになります。

2.3 練習課題

問題 2-1

2.2 節のサンプルプログラムに対し、10 個の箱がウィンドウの真ん中を中心に並ぶように修正せよ。

問題 2-2

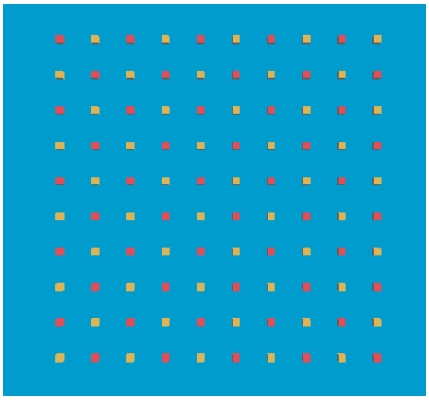
10 個の箱が斜めに並ぶように配置せよ。

問題 2-3

100 個の箱が 10×10 のマス目状に並ぶように配置せよ。

問題 2-4

マス目状に並んでいる 100 個の箱に対し、箱の色が赤と黄色で市松模様 (チェック模様) になるようにせよ。ただし、if 文はまだ用いないこと。(ヒント: 配列は複数宣言することが可能)



第3章　条件分岐とアニメーション

3.1　アニメーション

アニメーション

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        block(10.0, 10.0, 10.0);
 7:     fk_Model        model;
 8:
 9:     model.setShape(&block);
10:     model.setMaterial(Yellow);
11:     window.entry(model);
12:     window.setSize(800, 600);
13:     window.setBGColor(0.3, 0.6, 0.8);
14:     window.open();
15:
16:     while(window.update() == true) {
17:         model.glTranslate(0.1, 0.0, 0.0);
18:     }
19:
20:     return 0;
21: }
```

解説

- 16 行目の while についての解説はこれまで割愛していましたが、これは for 文と同様に繰り返しを実現するための文法です。

```
while(繰り返し条件) {
        実行文 1;
        実行文 2;
           :
}
```

つまり、while は for に対して「初期処理」と「繰り返し処理」を省いたものと同一です。実際、16 行目は

```
for(;window.update() == true;) {
```

と記述しても、まったく同様に動作します。

- while の中身の「`window.update() == true`」の意味ですが、ここでは 2 つの事が同時に行われています。

  1. 現在のモデルの状況に合わせて画面を更新する。
  2. ウィンドウが開かれているかどうかチェックする。

  while の中では、ウィンドウが開かれている場合に繰り返すようになっています。ウィンドウが閉じられた場合 (ウィンドウ右上の×印を押す、ESC キーが押されるなど) に、while の条件文が満たされなくなり、while 文の外に出ます。22 行目にある「`return 0;`」というのは、C++プログラムの実行を終了することを意味するもので、これによってアプリケーションが終了します。

- 17 行目の「glTranslate()」関数は、モデルを平行移動するものです。今回は直方体を $(0.1, 0, 0)$ だけ移動させています。これはわずかな移動量ですが、繰り返しで何度も移動することによって、パラパラ漫画の要領でスムースに移動するアニメーションとして表現されているわけです。

- 17 行目の内容を単に 10 行連続で記述すると、10 コマ分のアニメーションになると考えがちですが、実際には $(1, 0, 0)$ という移動量で一回だけ移動が行われるだけになります。なぜなら、画面更新は「`window.update()`」を通ったタイミングで行われるからです。アニメーションプログラムは、

  1. 次の瞬間のモデルの状態 (移動や変形など) を設定しておく。
  2. 画面を再描画する。

  の繰り返しになります。そのため、多くのモデルを同時に変更させていく必要があるため、プログラムはとても複雑になっていきます。

3.2 キー入力と if 文

キー入力と if 文

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        block(10.0, 10.0, 10.0);
 7:     fk_Model        model;
 8:
 9:     model.setShape(&block);
10:     model.setMaterial(Yellow);
11:     window.entry(model);
12:     window.setSize(800, 600);
13:     window.setBGColor(0.3, 0.6, 0.8);
14:     window.open();
15:
16:     while(window.update() == true) {
17:         if(window.getKeyStatus('R') == FK_SW_PRESS) {
18:             model.glTranslate(0.1, 0.0, 0.0);
19:         }
20:         if(window.getKeyStatus('L') == FK_SW_PRESS) {
21:             model.glTranslate(-0.1, 0.0, 0.0);
22:         }
23:     }
24:
25:     return 0;
26: }
```

解説

- ここでは、「**if 文**」という新しい文法が出てきます。if 文は、ある条件によって処理内容を変えていくときに使います。17 行目の記述は「現在 R キーが押されているかどうか」を判定しています。括弧の中のシングルクォーテーションで囲まれている文字が、調査対象キーを意味します。
- if 文の文法は、以下の様にできています。

```
if(条件 1) {
    // 条件 1 が正しい場合
    実行文 1;
} else if(条件 2) {
    // 条件 1 が正しくなくて、
    // 条件 2 が正しい場合
    実行文 2;
} else {
    // 条件 1, 条件 2 が共に正しくない場合
    実行文 3;
}
```

このように、ある条件が成り立たない場合は、「else」以下にさらに処理を記述することが可能です。else 以下の部分は (今回の例のように) 省略することもできます。

3.3 複雑な制御

if 文の入れ子

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        block(10.0, 10.0, 10.0);
 7:     fk_Model        model;
 8:     fk_Vector       pos;
 9:     int             flag;
10:
11:     model.setShape(&block);
12:     model.setMaterial(Yellow);
13:     window.entry(model);
14:     window.setSize(800, 600);
15:     window.setBGColor(0.3, 0.6, 0.8);
16:     window.open();
17:
18:     flag = 1;
19:     while(window.update() == true) {
20:         if(flag == 1) {
21:             model.glTranslate(0.1, 0.0, 0.0);
22:         } else if(flag == 2) {
23:             model.glTranslate(-0.1, 0.0, 0.0);
24:         }
25:
26:         pos = model.getPosition();
27:         if(flag == 1) {
28:             if(pos.x > 20.0) {
29:                 flag = 2;
30:             }
31:         }
32:     }
33:
34:     return 0;
35: }
```

解説

- プログラムの解説に入る前に、これまであまり明確にしてこなかった「条件の書き方」を述べておきます。for や while の終了条件や、if 文での条件部分は、基本的に「2 つの値の比較」によって行われます。これ

は、以下の様な種類があります。

記述法	意味	使い方例
`x == y`	x と y が等しい	`a == (b + 10)`
`x != y`	x と y が等しくない	`(a % 2) != 0`
`x > y`	x が y より大きい	`(a - b) > 10`
`x < y`	x が y 未満	`a < (2 * b)`
`x >= y`	x が y 以上	`a >= (b - c)`
`x <= y`	x が y 以下	`a <= ((b+c) * i)`

さらに、条件は 2 つ以上の条件を並べることも可能です。

記述法	意味	使い方例
`A && B`	A と B の両方が成り立つ場合	`x < y && y < z`
`A || B`	A と B がいずれかが成り立つ場合	`x > y || x == 0`

- 26 行目は、モデルの現在位置を求めるための機能で、8 行目で宣言した「fk_Vector 型」の変数「pos」にその位置が保存されます。この型は、「変数名.x」や「変数名.y」という形式でベクトルの x, y, z 各成分 (座標) を参照することができます。今回は、28 行目でモデルの x 座標を参照しています。
- 27〜31 行目は、if 文が 2 重に使われています。このように、if 文を何重にも囲って使っていくことを「if 文の入れ子」と言います。for 文や while 文も入れ子にすることが可能で、例えば

```
for(i = 1; i <= 9; i++) {
    for(j = 1; j <= 9; j++) {
        cout << i*j << endl;
    }
}
```

というプログラムは、九九の答えを順番に表示していくことになります。今回は、「flag が 1 のときで、モデル位置の x 座標が 20 を超えたとき、flag を 2 に変更する」という処理をしています。その結果、次の繰り返しからは 21 行目ではなく 23 行目が実行されるようになり、モデルが左に移動し始めるという仕組みになっています。

3.4 練習課題

問題 3-1

3.2 節のプログラムを改変し、「U」キーで上昇、「D」キーで下降、「F」キーで前進 ($-z$ 方向)、「B」キーで後退 ($+z$ 方向) に進む機能を追加せよ。

問題 3-2

3.3 節のプログラムの 29,30 行目では 2 つの if 文の入れ子になっているが、これを条件式を並べる記法を使って修正し、1 つの if 文だけで同じ機能を実現せよ。

問題 3-3

直方体がウィンドウの右側と左側を交互に行き来するプログラムを作成せよ。

問題 3-4

直方体が、長方形状の軌跡を時計回りに周回するようなプログラムを作成せよ。

問題 3-5

直方体が放物運動を描くアニメーションプログラムを作成せよ。(glMoveTo を利用してもよい。glTranslate だけでも実現できるが、やや難！)

第4章　物体の消去と干渉判定

4.1　カウンターとモデル消去

カウンターとモデル消去

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        block(10.0, 10.0, 10.0);
 7:     fk_Sphere       sphere(8, 5.0);
 8:     fk_Model        modelA, modelB;
 9:     int             count, div;
10:
11:
12:     modelA.setShape(&sphere);
13:     modelA.setMaterial(Green);
14:     modelA.glMoveTo(-20.0, 0.0, 0.0);
15:     window.entry(modelA);
16:
17:     modelB.setShape(&block);
18:     modelB.setMaterial(Yellow);
19:     modelB.glMoveTo(20.0, 0.0, 0.0);
20:     window.entry(modelB);
21:
22:     window.setSize(800, 600);
23:     window.setBGColor(0.6, 0.7, 0.8);
24:     window.open();
25:
26:     for(count = 0; window.update() == true; count++) {
27:
28:         div = count/100;
29:         if(div % 2 == 0) {
30:             window.entry(modelA);
31:             modelB.setMaterial(Yellow);
32:         } else {
33:             window.remove(modelA);
34:             modelB.setMaterial(Red);
35:         }
36:     }
37:
38:     return 0;
39: }
```

解説

- 26 行目では、アニメーションループで while ではなく for 文を用いています。これにより、count という変数が最初は 0 に初期化され、ループが 1 回まわる度に 1 ずつ増加していきます。このようにしておくことで、現在のループが何回目なのかが count に入ることになります。

  このように、なんらかの回数を数えておく変数を「**カウンター**」と呼ぶことがあり、前回紹介した「フラグ」と共によく利用されます。

- 28 行目は割り算の値を div に代入していますが、
  `count/100` は数学的には整数ではない数値もありえます。しかし、div は int 型ですので中に格納される値は整数値となります。このように、実際の計算結果が実数であるにも関わらず、無理矢理整数型に値を代入することを「**丸め演算**」と呼びます。int 型の丸め演算では、小数点以下は基本的に 切り捨て となります。ですので、

  ```
  int a = 59;
  int b = a/10;
  ```

  というプログラムは、b には (四捨五入した 6 ではなく)5 が入ります。

- 29 行目の、パーセント記号は「割った余り」を求めるための記号 (演算子) です。「`if(div % 2 == 0)`」というのは、div を 2 で割った余りが 0、つまり div が偶数のときということになります。結果として、count の百の位が偶数ならこの if の条件が真となり、奇数なら偽となります。

- 33 行目の remove 関数は、一度 window に登録したモデルを表示要素から外すための関数です。従って、div が奇数の場合は球が表示されなくなります。再び表示させるには、34 行目にあるように改めて表示したモデルを entry することで可能です。既に entry されているモデルに対して entry した場合は、内部的には二重に表示されるわけではなく、実際には特に動作に変化はありません。remove を何度も行う場合も同様です。

4.2 干渉判定

球同士の干渉判定

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Sphere       sphere(8, 5.0);
 7:     fk_Model        modelA, modelB;
 8:     fk_Vector       posA, posB, vec;
 9:     double          distance;
10:
11:     modelA.setShape(&sphere);
12:     modelA.setMaterial(Green);
13:     modelA.glMoveTo(-20.0, 2.0, 0.0);
14:     window.entry(modelA);
15:
16:     modelB.setShape(&sphere);
17:     modelB.setMaterial(Yellow);
18:     modelB.glMoveTo(0.0, -2.0, 0.0);
19:     window.entry(modelB);
20:
21:     window.setSize(800, 600);
22:     window.setBGColor(0.6, 0.7, 0.8);
23:     window.open();
24:
25:     while(window.update() == true) {
26:         modelA.glTranslate(0.05, 0.0, 0.0);
27:
28:         posA = modelA.getPosition();
29:         posB = modelB.getPosition();
30:         vec = posB - posA; // vec = ベクトル AB
31:         distance = vec.dist(); // vec の長さ
32:
33:         if(distance < 10.0) {
34:             modelA.setMaterial(Red);
35:         } else {
36:             modelA.setMaterial(Green);
37:         }
38:     }
39:
40:     return 0;
41: }
```

解説

- ここでは「**干渉判定**」を扱います。干渉判定とは、2 つの物体に重なっている部分 (干渉部分) があるかどうかを調べる処理のことです。一般的な形状同士の干渉判定はとても高度な理論ですが、今回は最も単純な「2 個の球体同士の干渉判定」を扱います。

- 2 個の球 A, B が干渉しているかどうかの判定は以下のようにして求めます。球の中心位置同士の距離を L, それぞれの半径を r_a, r_b としたとき、

$$L \leqq r_a + r_b \tag{4 - 1}$$

  となった場合、2 つの球は干渉しています。

- 球の中心同士の距離を求めるには、ベクトルを用います。2 個の球の中心位置ベクトルをそれぞれ $\mathbf{A}, \mathbf{B}$ としたとき、その距離は

$$|\overrightarrow{\mathbf{AB}}| = |\mathbf{B} - \mathbf{A}| \tag{4 - 2}$$

  となります。これは実際には

$$|\mathbf{B} - \mathbf{A}| = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2 + (B_z - A_z)^2} \tag{4 - 3}$$

  となるのですが、このような複雑な式をプログラムで記述しなくても、fk_Vector 型には「dist()」という便利な関数があり、そのベクトルの距離を得ることができます。31 行目が実際に dist 関数を利用している様子です。

- 9 行目にある「double」というのは、実数を保管しておくための型です。int 型変数は整数値しか保管できませんでしたが、double 型の変数は小数点以下についても保管が可能です。31 行目は、かなり正確な数値が保管されます。

- 30 行目で fk_Vector 型の変数で引き算を行っていますが、これは数値の引き算ではなくベクトルの引き算になります。fk_Vector 型は、ベクトル同士の加減算や実数との積を通常の数式と同様に記述することができます。

4.3 練習課題

問題 4-1

2 個の球を離して配置し、片方を適当なキーで移動操作できるようにする。2 つの球が干渉したら、止まっている方の球が消去されるようなプログラムを作成せよ。

問題 4-2

2 個の球のうち一つをキー操作で移動でき、もう一つを前回の問題 3-4 の動きをするようにする。干渉したら自動的に移動している側の球の色が変わるようなプログラムを作成せよ。

問題 4-3

小さな球を 横に 10 個配置し、やや大きめの球をキー操作で移動できるようにする。干渉した小球が消去されるようなプログラムを作成せよ。

問題 4-4

小さな球を格子状に $10 \times 10 = 100$ 個配置し、やや大きめの球をキー操作で移動できるようにする。干渉した小球が消去されるようなプログラムを作成せよ。

第5章　様々な形状とメッセージ出力機能

5.1　形状クラス紹介

この節では、これまで扱ってきた直方体 (fk_Block) や球 (fk_Sphere) 以外の形状の作成方法を記述します。基本的には、fk_Block や fk_Sphere と同様に変数宣言時に寸法を指定し、fk_Model に対して setShape() 関数で登録して利用します。

5.1.1　円

円の生成には、「fk_Circle」という型で変数を宣言します。

```
fk_Circle    circle(8, 5.0);
```

宣言時の引数は、最初の数字が分割数を表すもので、実際のはこの数値の 4 倍の角数を持った正多角形が表示されます。上記の場合は 32 角形が表示されることになります。この分割数が大きければ大きいほど滑らかな円が表示されますが、多数のモデルを表示する場合に速度が低下する場合があります。2 番目の引数は円の半径です。

なお、この円は裏側から見た場合は表示されません。表面の向きは、 1.4 節で紹介した glVec() 関数を使って制御します。

5.1.2　正多角柱・円柱

正多角柱や円柱の生成は、「fk_Prism」という型で変数を宣言します。

```
fk_Prism     prism(3, 4.0, 3.0, 5.0);
```

最初の引数は角数を表すもので、例の場合は三角柱を作成することになります。円柱は、角数の多い正多角柱として生成します。分割数が 32 を超える程度でほとんど円柱と見分けがつかなくなります。

2 番目,3 番目の引数はそれぞれ底面の外接円半径となります。この数値は普通は同一にしますが、台形型のように上面と底面を変えることも可能です。

4 番目の引数は、角柱 (円柱) の高さを表します。

初期配置は、底面の中心が原点となり、上面が $-z$ 方向、底面が $+z$ となります。

5.1.3 正多角錐・円錐

正多角錐や円錐の生成は、「fk_Cone」という型で変数を宣言します。

```
fk_Cone      cone(3, 4.0, 5.0);
```

最初の引数は角数を表すもので、例の場合は三角錐を作成することになります。円錐も円柱のときと同様に、角数の多い (32 程度以上の) 角錐として生成します。

2 番目の引数は底面の外接円半径となります。3 番目の引数は、角錐 (円錐) の高さを表します。

初期配置は、底面の中心が原点となり、頂点が $-z$ 方向、底面が $+z$ 方向となります。

5.2 キー操作捕捉

3.2 節で文字キーの状態取得を解説しましたが、矢印キーやエンターキーなどの特殊キーについての状態取得は述べていませんでした。これらの特殊キーの状態を検知するには、fk_Window クラスの「getSpecialKeyStatus()」という関数を利用します。

```
fk_AppWindow    win;
        :
        :
if(win.getSpecialKeyStatus(FK_UP) == FK_SW_PRESS) {
        :
        :
}
```

使い方自体は getKeyStatus() 関数と同様です。引数の「FK_UP」は、上矢印キーの状態を取得する場合になります。ここに、検知したいキーに合う値を引数にして下さい。以下に、値とキーの対応を記します。

値	キー
FK_SHIFT_R	右シフトキー
FK_SHIFT_L	左シフトキー
FK_CTRL_R	右コントロールキー
FK_CTRL_L	左コントロールキー
FK_ALT_R	右オルトキー
FK_ALT_L	左オルトキー
FK_ENTER	エンター (改行、リターン) キー
FK_BACKSPACE	バックスペース (後退) キー
FK_DELETE	デリート (削除) キー
FK_CAPS_LOCK	キャップスロックキー
FK_TAB	タブキー
FK_PAGE_UP	ページアップキー
FK_PAGE_DOWN	ページダウンキー
FK_HOME	ホームキー
FK_END	エンドキー
FK_INSERT	インサートキー
FK_LEFT	左矢印キー
FK_RIGHT	右矢印キー
FK_UP	上矢印キー
FK_DOWN	下矢印キー
FK_F1 〜 FK_F12	F1〜F12 ファンクションキー

5.3 メッセージ出力

プログラミングをしていると、現在の変数の値がどうなっているのかを参照したいことがよくあります。この方法は様々なものがあるのですが、FK のプログラムでは
「`fk_Window::printf()`」という関数を使うのが最も簡単です。この関数を用いると、メッセージを出力するためのウィンドウが別途現れ、そこに指定した文字列が表示されます。

```
fk_Window::printf("This is Sample.");
```

変数の値を表示したい場合は、整数 (int 型) の場合は「%d」、実数 (double 型) の場合は「%f」を用います。

```
int    a, b;

a = 10;
b = a + 5;
fk_Window::printf("a = %d, b = %d", a, b);
```

上記のプログラムは、メッセージウィンドウに「a = 10, b = 15」と表示されます。このように、printf は最初の引数に出力したい文字列の書式を入れ、変数の値を表示したい箇所に「%d」や「%f」を入れておきます。その後の引数に、それぞれ表示したい変数 (や計算式) を記述しておきます。

fk_Vector のようなベクトル値を表示したい場合は、以下の様にするとよいでしょう。

```
fk_Vector    vec;

fk_Window::printf("vec = (%f, %f, %f)", vec.x, vec.y, vec.z);
```

printf 関数は、小数点以下の桁数を指定するなど、細かな書式を指定することも可能です。この書式は、一般的な C++ 言語の「printf()」関数に準じていますので、詳しいことは参考書や Web による情報を閲覧して下さい。

5.4 練習課題

問題 5-1

円、多角柱、多角錐を実際に表示するプログラムを作成せよ。

問題 5-2

「問題 4-1」で作成したプログラムに対し、球の中心同士の距離が常に表示されるようにプログラムを変更せよ。

第6章　3Dプログラミングへの挑戦

6.1　モデルの回転

原点を通る y 軸回転

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        block(10.0, 10.0, 10.0);
 7:     fk_Model        modelA, modelB;
 8:     fk_Vector       origin(0.0, 0.0, 0.0);
 9:
10:     modelA.setShape(&block);
11:     modelA.setMaterial(Yellow);
12:     modelA.glMoveTo(20.0, 20.0, 0.0);
13:     window.entry(modelA);
14:
15:     modelB.setShape(&block);
16:     modelB.setMaterial(Red);
17:     modelB.glMoveTo(20.0, -20.0, 0.0);
18:     window.entry(modelB);
19:
20:     window.setSize(800, 600);
21:     window.setBGColor(0.6, 0.7, 0.8);
22:     window.open();
23:
24:     while(window.update() == true) {
25:         modelA.glRotate(origin, fk_Y, FK_PI/180.0);
26:         modelB.glRotateWithVec(origin, fk_Y, FK_PI/180.0);
27:     }
28:     return 0;
29: }
```

解説

- 8 行目では、fk_Vector 型 (ベクトル) の変数宣言の際に、引数に 3 個の実数を入力しています。fk_Vector 型では、このように記述しておくと変数宣言時にベクトルの初期値設定が可能です。「origin」というのは「(数学的な意味での) 原点」という意味の英単語で、原点を表すベクトル変数の名称としてよく用いられます。(もっと略して「org」とすると、「original」と混同してしまって若干紛らわしくなります。)

- 25,26 行目にある「glRotate()」と「glRotateWithVec()」という関数は、共にモデルに対して回転移動を行います。3 次元での回転は、「回転軸」と「角度」を指定することになります。(両者の違いは次の項目で述べます。)

  回転軸の指定の仕方は色々あるのですが、今回は「回転軸が通る点」と「回転軸に平行な座標軸」という指定の方法を用いています。

  最初の引数となっている「origin」が回転軸が通る点で、「fk_Y」は「y 軸に平行な回転軸」ということになります。結果的に、今回は共に y 軸そのものを中心に回転していることになります。

  3 番目の引数角度は「弧度法 (ラジアン)」という、2π を 1 回転とする角度単位で指定します。29 行目にある「FK_PI」というのは FK のプログラム中で用いることができる円周率 π を意味します。従って、今回のループ 1 回あたりの回転角は $\frac{\pi}{180}$(rad) で、これは度数法でいう 1° となります。

- glRotate() 関数と glRotateWithVec() 関数は共にモデルの回転移動を行うものですが、glRotate() の方はあくまで物体の位置のみを変化させるもので、物体の向きについては変更を行いません。それに対し、glRotateWithVec() は位置の移動と共に向きも回転します。用途によって使い分けて下さい。

6.2 ローカル座標系

グローバル回転とローカル回転

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        block(10.0, 10.0, 10.0);
 7:     fk_Model        modelA, modelB;
 8:     fk_Vector       origin(0.0, 0.0, 0.0);
 9:
10:     modelA.setShape(&block);
11:     modelA.setMaterial(Yellow);
12:     modelA.glMoveTo(20.0, 0.0, 0.0);
13:     window.entry(modelA);
14:
15:     modelB.setShape(&block);
16:     modelB.setMaterial(Red);
17:     modelB.glMoveTo(20.0, 0.0, 0.0);
18:     window.entry(modelB);
19:
20:     window.setSize(800, 600);
21:     window.setBGColor(0.6, 0.7, 0.8);
22:     window.open();
23:
24:     while(window.update() == true) {
25:         modelA.glRotateWithVec(origin, fk_Y, FK_PI/180.0);
26:         modelB.loRotateWithVec(origin, fk_Y, FK_PI/180.0);
27:     }
28:     return 0;
29: }
```

解説

- ここでは、「**グローバル座標系**」と「**ローカル座標系**」と呼ばれる概念を説明します。これまで、位置や方向を表す「座標軸」というのはプログラムの仮想世界の中では一つだけであり、不動のものでした。これに対し、各モデルの中心を原点とし、各モデルの向いている方向によって規定される座標系を「ローカル座標系」と言います。図 6.1 にその概念図を示します。xy がグローバル座標系、$x'y'$ がローカル座標系です。

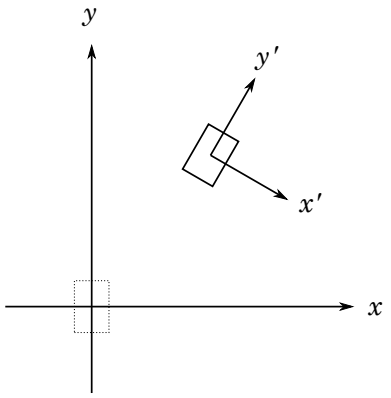


図 6.1: グローバル座標系とローカル座標系

fk_Model では多くの移動や方向指定の関数に、グローバル座標系とローカル座標系用の両方の関数が用意されています。グローバル座標系は、現在のモデルの位置や方向に従わないような指定、現実世界では「東西南北」を基本とした指定に向いています。一方のローカル座標系での指定は、モデル自身を中心とした指定、すなわち「前後左右」を基本とした指定に向いています。

- 26 行目にある「loRotateWithVec()」関数は、ローカル座標系による回転移動を行うものです。27 行目の glRotateWithVec() と同じく原点を通る y 軸回転を行っていますが、ローカル座標系の場合はモデルの中心がそのまま原点となりますから、modelB は同じ場所に留まって回転します。一方で modelA は (グローバル座標系の) 原点を中心に回転していきます。

- 平行移動にも glTranslate() と対となる「loTranslate()」関数があります。例えば fk_Model 型の変数「model」があったとして、

```
model.glTranslate(0.0, 0.0, -1.0);
```

はモデルがどちらを向いていようが $(0, 0, -1)$ だけ平行移動しますが、

```
model.loTranslate(0.0, 0.0, -1.0);
```

の場合は物体の向いている方向に前進します。(FK では $(0, 0, -1)$ が物体の前方向を意味します。)

6.3 カメラの移動

カメラモデルの制御

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Model        camera;
 7:     fk_Vector       origin(0.0, 0.0, 0.0);
 8:
 9:     camera.glMoveTo(0.0, 1.0, 0.0);
10:     window.setCameraModel(&camera);
11:     window.showGuide(FK_GRID_XZ);
12:     window.setSize(800, 600);
13:     window.setBGColor(0.6, 0.7, 0.8);
14:     window.open();
15:
16:     while(window.update() == true) {
17:         if(window.getSpecialKeyStatus(FK_UP) == FK_SW_PRESS) {
18:             camera.loTranslate(0.0, 0.0, -0.1);
19:         }
20:
21:         if(window.getSpecialKeyStatus(FK_LEFT) == FK_SW_PRESS) {
22:             camera.loRotateWithVec(origin, fk_Y, FK_PI/180.0);
23:         }
24:     }
25:     return 0;
26: }
```

解説

- このサンプルプログラムでは、これまでのように形状を作成する型が何も用いられていません。そのかわり、11 行目にある fk_AppWindow 型変数での
「showGuide()」という関数が記述されています。この関数は、仮想世界上に「グリッド」という格子状の線を表示するためのもので、今回は xz 平面に沿ってグリッドが作成されます。

- 10 行目の「setCameraModel()」という関数は、ある fk_Model 型の変数を&記号を付けて引数として入力することで、そのモデルを仮想世界上の「カメラ」となるように設定する関数です。これまで FK プログラムのカメラは不動のものでしたが、このモデルを中心に一人称視点で仮想世界上を動き回るプログラムを作成することができます。

- 18 行目では、前節の解説で説明した「loTranslate()」関数が用いられています。27 行目でカメラが回転した場合でも、ローカル座標系による指定なので上矢印キーが押された場合は常に前進 $((0, 0, -0.1)$ の平行移

動) するようになっています。

6.4 練習課題

問題 6-1

6.3 節のプログラムに対し、下矢印キーで後退、右矢印キーで右回転する機能を追加せよ。

問題 6-2

上記のプログラムに対し、建物に見立てた形状を空間内に配置し、街中を動き回るようなプログラムを作成せよ。

問題 6-3

さらに上記のプログラムに対し、作成した建物をすり抜けないような処理を追加せよ。

第7章　親子関係と運動

7.1　モデルの親子関係

親子関係

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Block        bodyShape(1.0, 0.2, 2.0);
 7:     fk_Prism        tireShape(20, 0.2, 0.2, 0.2);
 8:     fk_Model        body, tire[4];
 9:     fk_Vector       origin(0.0, 0.0, 0.0);
10:     int             i;
11:
12:     body.setShape(&bodyShape);
13:     body.setMaterial(Red);
14:     window.entry(&body);
15:
16:     tire[0].glMoveTo(0.5, 0.0, 0.6);
17:     tire[0].glVec(1.0, 0.0, 0.0);
18:     tire[1].glMoveTo(0.5, 0.0, -0.6);
19:     tire[1].glVec(1.0, 0.0, 0.0);
20:     tire[2].glMoveTo(-0.5, 0.0, 0.6);
21:     tire[2].glVec(-1.0, 0.0, 0.0);
22:     tire[3].glMoveTo(-0.5, 0.0, -0.6);
23:     tire[3].glVec(-1.0, 0.0, 0.0);
24:
25:     for(i = 0; i < 4; i++) {
26:         tire[i].setMaterial(MatBlack);
27:         tire[i].setShape(&tireShape);
28:         tire[i].setParent(&body);
29:         window.entry(&tire[i]);
30:     }
31:
32:     window.setCameraPos(0.0, 10.0, 30.0);
33:     window.setCameraFocus(0.0, 0.0, 0.0);
34:     window.setSize(800, 600);
35:     window.setBGColor(0.6, 0.7, 0.8);
36:     window.open();
37:     window.showGuide(FK_GRID_XZ);
38:
39:     body.glMoveTo(10.0, 0.1, 0.0);
40:
41:     while(window.update() == true) {
42:         body.glRotateWithVec(origin, fk_Y, FK_PI/180.0);
43:     }
44:     return 0;
45: }
```

解説

- Word や PowerPoint などのアプリケーションでは、複数の図形を一つのグループとしてまとめて扱う「グループ化」という機能があります。FK の fk_Model にも同様の機能があり、これを「親子関係」と呼んでいます。モデル A をモデル B の親モデルとして設定した場合、A を動かすと同じように B も追従して動きます。今回の例は、直方体と円柱を使って簡易的な車モデルを作成しています。直方体を「body」という親モデルとし、円柱を「tire」という子モデルにすることで、車体とタイヤを一括して操作しています。

- 親子関係を設定しているのは 28 行目の「setParent()」関数です。これにより、各 `tire[i]` モデルは body モデルの子モデルとして設定されることになります。なお、29 行目でタイヤモデルも window に登録していますが、親子関係は (原則として) 位置や方向を制御するものであり、色や描画属性には関与しないものなので、マテリアル設定やウィンドウへの登録は別途行う必要があります。

- 前回、カメラモデルの制御を行いましたが、カメラを固定して設定する場合はわざわざ fk_Model 型のカメラ用変数を用意しなくても、32,33 行目にあるように直接位置と方向を指定することが可能です。33 行目の setCameraFocus() 関数は方向ベクトルを指定するものではなく、カメラの注視点を指定するものです。

- 42 行目では body モデルしか制御していませんが、実際にプログラムを実行してみるとタイヤも追従していることがわかります。もちろん、回転だけではなく平行移動や方向指定でも同様に追従します。

7.2 運動と速度

落下運動

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Prism        bodyShape(20, 0.5, 0.5, 2.0);
 7:     fk_Sphere       headShape(10, 0.5);
 8:     fk_Model        body, head;
 9:     fk_Vector       pos, speed;
10:
11:     body.setShape(&bodyShape);
12:     body.setMaterial(Yellow);
13:     window.entry(&body);
14:
15:     head.setShape(&headShape);
16:     head.setMaterial(Yellow);
17:     head.glMoveTo(0.0, 0.0, -2.5);
18:     head.setParent(&body);
19:     window.entry(&head);
20:
21:     body.glVec(0.0, 1.0, 0.0);
22:     body.glMoveTo(-15.0, 0.0, 0.0);
23:
24:     window.setCameraPos(0.0, 10.0, 50.0);
25:     window.setCameraFocus(0.0, 0.0, 0.0);
26:     window.setSize(800, 600);
27:     window.setBGColor(0.6, 0.7, 0.8);
28:     window.showGuide(FK_GRID_XZ);
29:     window.open();
30:
31:     speed.set(0.05, 0.1, 0.0);
32:
33:     while(window.update() == true) {
34:         pos = body.getPosition();
35:         if(pos.y >= 0.0) {
36:             speed.y -= 0.001;
37:         } else {
38:             speed.y = 0.0;
39:         }
40:         body.glTranslate(speed);
41:     }
42:     return 0;
43: }
```

解説

- ゲームの中で頻繁に用いられる「ジャンプ」ですが、これをプログラムで実装するには少し数学や物理を知る必要があります。一般的に、落下運動は「放物線」という 2 次関数を描くことが知られていますが、このことだけでプログラムを記述するのは結構大変です。そこで、「速度」に着目します。
- 物体が動いているのであれば、それはその物体の**速度**が 0 ではない ことを意味します。もっと数学的に言えば

  「速度ベクトルがゼロベクトルではない」

  ということです。この速度ベクトルを、今回のプログラム中では「speed」という名前の変数として用意しています。そして、「速度」とは「単位時間あたりの移動量」を指します。今回は単位時間を「1 回描画が行われるまでの時間」としておきます。すると、FK の場合はそれを glTranslate() 関数で指定することで平行移動を実現できますから、40 行目にあるように glTranslate() 関数に速度ベクトルを指定するだけで物理運動アニメーションが実現できることになります。
- 次に速度の制御です。重力が働いている空間では、下方向に「加速度」が働きます。これの意味するところは、「単位時間あたりに下向き ($-y$ 方向) に速度ベクトルが追加される」という意味です。つまり、今の速度に対して一定量 y 成分が減少することになります。36 行目で y 成分を減らしています。
- そのままだと永久に落下してしまうので、現在位置の高さが正の場合は落下し、負の場合は y 方向の速度は 0 としておきます。これにより、某配管工主人公ゲームのような動きが実現できるというわけです。
- このように、ゲームでは「位置」ではなく「速度」や「加速度」の方を調整する方が簡単にプログラムが組めることが多くあります。

7.3 練習課題

問題 7-1

7.1 節のサンプルプログラムの車に対し、さらに椅子や人などのオブジェクトを追加せよ。

問題 7-2

問題 7-1 のプログラムに対してさらに、上下矢印キーで前後に進み、左右矢印キーでその場で回転するようにプログラムを修正せよ。

問題 7-3

7.2 節のプログラムに対し、「地上にいる間にスペースキーを押すと再びジャンプする」という機能を追加せよ。

問題 7-4

問題 7-2 のプログラムと 7-3 のプログラムを統合し、車がジャンプ機能を持ち自在に移動できるプログラムを作成せよ。

第8章　テクスチャマッピングと画面上への文字表示

8.1　テクスチャマッピング

テクスチャ画像表示

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_RectTexture  texture;
 7:     fk_Block        block(1.0, 1.0, 1.0);
 8:     fk_Model        texModel, blockModel;
 9:     fk_Vector       origin(0.0, 0.0, 0.0);
10:
11:     texture.readPNG("Chinmi.png");
12:     texture.setTextureSize(7.0, 5.0);
13:     texModel.setShape(&texture);
14:     texModel.setMaterial(White);
15:     texModel.glMoveTo(0.0, 5.0, 0.0);
16:     window.entry(texModel);
17:
18:     blockModel.setShape(&block);
19:     blockModel.glMoveTo(2.0, 6.0, 0.0);
20:     blockModel.setMaterial(Yellow);
21:     window.entry(blockModel);
22:
23:     window.setCameraPos(0.0, 5.0, 20.0);
24:     window.setCameraFocus(0.0, 5.0, 0.0);
25:     window.setSize(800, 600);
26:     window.setBGColor(0.6, 0.7, 0.8);
27:     window.open();
28:     window.showGuide(FK_GRID_XZ);
29:
30:     while(window.update() == true) {
31:         texModel.glRotateWithVec(origin, fk_Y, FK_PI/360.0);
32:     }
33:     return 0;
34: }
```

解説

- 3D グラフィックスの重要な技術に「**テクスチャマッピング**」というものがあります。これは画像の全体または一部分を 3 次元空間内に表示するもので、リアルタイムグラフィックス技術の中でも最重要と言えるものです。このサンプルプログラムは、その最も基本的な利用方法として 1 枚の画像を空間上に配置しています。表示用の画像をあらかじめ授業用ページからダウンロードしておいて下さい。

- FK にも、テクスチャマッピングを行う方法は様々なものが用途に合わせて準備されていますが、今回使う「fk_RectTexture」という型は、画像をそのまま一枚の長方形状に扱うためのものです。6 行目にあるように、まずは変数を宣言しておきます。
- 11 行目で画像を読み込んでおきます。この画像ファイルは、exe ファイルがあるフォルダと同じ場所に置いておく必要があります。今回は PNG 形式の画像を入力していますが、他に「readJPEG()」という関数で JPEG 形式を入力することや、「readBMP()」という関数で BMP 形式の画像を入力することもできます。
- 12 行目の setTextureSize 関数は、FK 内での画像テクスチャの横幅と縦幅をそれぞれ設定します。縦横比 (アスペクト比) は、必ずしも元画像の縦横比と一致している必要はありません。
- 14 行目でマテリアルを設定しています。テクスチャ画像の元の色をほぼそのまま再現したい場合は、ここで白に近いマテリアルを設定しておく必要があります。
- 31 行目でテクスチャ画像を回転させていますが、裏側から見ている場合は見ることができません。

8.2 文字列の画面内表示

スプライト表示

```
 1: #include <FK/FK.h>
 2: #include <sstream>
 3:
 4: string IntToString(int argI)
 5: {
 6:     stringstream ss;
 7:     ss << argI;
 8:     return ss.str();
 9: }
10:
11: int main(int, char **)
12: {
13:     fk_AppWindow    window;
14:     fk_SpriteModel  sprite;
15:     fk_Block        block(1.0, 1.0, 1.0);
16:     fk_Model        model;
17:     fk_Vector       origin(0.0, 0.0, 0.0);
18:     int             count;
19:     std::string     str;
20:
21:     if(sprite.initFont("rounded-mplus-1m-bold.ttf") == false) {
22:         fl_alert("Font Read Error");
23:     }
24:     sprite.setPositionLT(-280.0, 230.0);
25:     window.entry(sprite);
26:
27:     model.setShape(&block);
28:     model.glMoveTo(0.0, 6.0, 0.0);
29:     model.setMaterial(Yellow);
30:     window.entry(model);
31:
32:     window.setCameraPos(0.0, 5.0, 20.0);
33:     window.setCameraFocus(0.0, 5.0, 0.0);
34:     window.setSize(800, 600);
35:     window.setBGColor(0.6, 0.7, 0.8);
36:     window.open();
37:     window.showGuide(FK_GRID_XZ);
38:
39:     count = 0;
40:     while(window.update() == true) {
41:         str = "count = " + IntToString(count);
42:         sprite.drawText(str, true);
43:         model.glRotateWithVec(origin, fk_Y, FK_PI/360.0);
44:         count++;
45:     }
46:     return 0;
47: }
```

解説

- ここでは「スプライト表示」と呼ばれる手法について紹介します。これは簡単に言ってしまうと、3 次元空間上ではなくスクリーンに直接画像を投影することです。先ほど、テクスチャマッピングを使って 3D 上に画像を配置する技術を紹介しました。しかし、これを使って字幕などを表示した場合、ある物体がその字

幕の手前に配置された場合、字幕を遮ってしまいます。そこで、他の物体がどんなに近くにあっても画面上に表示される画像というのが必要となります。それを「スプライト表示」と言います。

「スプライト」は本来は旧世代のゲーム機やハードウェアなどでビデオ情報に直接画素値を書き込む技術のことを言いましたが、現在はこのような技術はもう用いられていません。しかし、今でもそのときの名残で画面に常に表示される画像を「スプライト」と呼ぶ慣習があります。

スプライト表示の中でもとりわけ需要が高いものに「メッセージ表示」があります。たとえばゲームでの現在の得点などがそれに当たります。今回は、それを行う仕組みを紹介します。

- 4〜9 行目の解説はまだ現時点でのこの授業の範疇を超えますが、これを書いておくことによって「IntToString()」という自作の関数を作っており、これによって数値を文字列に変換することができるものです。
- 14 行目にスプライト専用の型である「fk_SpriteModel」型の変数が宣言されています。
- 19 行目の「string」型は、C++で標準の文字列を格納するための型です。
- 21 行目で、スプライトで表示する文字のフォント用ファイルを読み込んでいます。今回は「Rounded M+」というフリーのフォントを使用します。このフォント用ファイルも授業用ページからダウンロードしておき、exe と同じフォルダにおいて下さい。Windows にもフォントファイルは多く搭載されており、それを表示に使用することも可能です [1]。
- 24 行目は画面上のどこにスプライトを表示するかを指定します。これは FK の標準的な座標系ではなく、画面内のピクセル単位で指定します。原点は画面の真ん中になります。
- 41 行目ですが、これは「count =」という文字列と、count の値を文字列に変換したものを合わせています。たとえば count の値が 500 だった場合は、str の中は「count = 500」という文字列が入ります。
- 42 行目で実際にスプライトの内容を書き換えます。第二引数を false にした場合、既に表示されていた文字列を消去せず、その後ろに続けて表示していくようになります。

[1] ただし、そのファイルを他の PC にコピーすることはライセンス違反となることがあります。

8.3 練習課題

問題 8-1

自分で画像を用意し、それをテクスチャマッピング表示せよ。

問題 8-2

6 枚のテクスチャを使って立方体 (サイコロ形状) を作成し、X,Y,Z キーでそれぞれ x, y, z 軸を中心に回転するプログラムを作成せよ。

問題 8-3

前回の問題 7-4 のプログラム (7-4 が完成できなかった場合は 7-2 でもよい) に対し、現在の車の 3 次元空間中の位置をスプライト表示するような機能を作成せよ。double 型の実数を string に変換するには、以下のような関数を main 関数の前に追加しておくとよい。

```
string DoubleToString(double argD)
{
    stringstream ss;
    ss << argD;
    return ss.str();
}
```

第9章　3Dデータの表示

9.1　形状データの読み込み

メタセコイアデータの読み込み

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_IFSTexture   ifsTex[2];
 7:     fk_Model        model[2], parent;
 8:     fk_Vector       origin(0.0, 0.0, 0.0);
 9:     int             i;
10:
11:     for(i = 0; i < 2; i++) {
12:         if(ifsTex[i].readJPG("violin.jpg") == false) {
13:             fl_alert("Image read error");
14:         }
15:         model[i].setShape(&ifsTex[i]);
16:         model[i].setMaterial(White);
17:         model[i].setParent(&parent);
18:         window.entry(model[i]);
19:     }
20:
21:     if(ifsTex[0].readMQOFile("violin.mqo", "nomal") == false) {
22:         fl_alert("Shape read error");
23:     }
24:     if(ifsTex[1].readMQOFile("violin.mqo", "mirror_z") == false) {
25:         fl_alert("Shape read error");
26:     }
27:
28:     parent.setScale(0.1);
29:
30:     window.setCameraPos(0.0, 50.0, 50.0);
31:     window.setCameraFocus(0.0, 0.0, 0.0);
32:     window.setSize(800, 600);
33:     window.setBGColor(0.6, 0.7, 0.8);
34:     window.open();
35:     window.showGuide(FK_GRID_XZ);
36:
37:     while(window.update() == true) {
38:         parent.glRotateWithVec(origin, fk_Y, FK_PI/360.0);
39:     }
40:     return 0;
41: }
```

解説

- FK では、「メタセコイア」というモデラー (3D 形状を作成するためのソフトウェア) の入力をサポートしています。メタセコイア自体は有料のソフトウェアですが、機能を制限した無料版もあります。FK で入力して用いる分には、無料版でも十分ですので、興味がある人は使ってみて下さい。現在 Ver.4 がリリースされていますが、まだ Ver.4 によるデータが FK で利用できるかどうか十分に検証がなされていませんので、Ver.3 系列の「メタセコイア LE R3.0」の利用を推奨します。URL は `http://metaseq.net/`です。

- 6 行目にある「fk_IFSTexture」という型が、形状の入力や表示をおこなうためのものです。今回は 2 つの異なるオブジェクトを入力するために配列にしています。メタセコイア内では、一つのファイルに対して複数の「オブジェクト」を生成することが可能です。データの中ではそれぞれ分離した形状として扱われていますので、FK で入力する場合にはそれぞれ別個に読み込む必要があります。

- この例ではテクスチャマッピングを行った形状の表示を行っていますが、テクスチャを使っていない形状の表示は「fk_IndexFaceSet」という型を用います。テクスチャ画像の入力を行わない点以外は、利用方法は fk_IFSTexture と同じです。

- 12 行目でまずはテクスチャ画像となるファイルを読み込みます。ここでは JPEG 形式の画像となっているので「readJPG」という関数を用いていますが、PNG 形式の場合は関数名を「readPNG」、BMP 形式の場合は関数名を「readBMP」として下さい。なお、fk_IFSTexture では画像を入力する前に形状データを入力するとプログラムが不正終了してしまうことがありますので、注意して下さい。

- 13 行目では「fl_alert」という関数が出てきます。これは、エラーメッセージを出力するダイアログを出すための関数で、今回のようにファイル読み込みに失敗したときなど、プログラムでなんらかのエラー状態となったときに用います。この関数が呼ばれると、ダイアログを閉じられるまでプログラムは一旦停止します。

- 17 行目で「parent」というモデルを親モデルとして登録しています。これにより、異なるオブジェクトに対し「parent」を操作するだけで両方を同時に動かすことが可能となります。

- 21,25 行目にある「readMQOFile」という関数がメタセコイアの形状データである MQO ファイルを入力する関数です。この関数は 2 つの文字列による引数を取ります。最初の引数は MQO ファイル名です。次の引数はオブジェクト名となります。この結果が false で合った場合、入力に失敗したことになります。今回のサンプルである violin.mqo ファイルには、「nomal」、「mirror_z」という 2 つのオブジェクトがあり、それぞれを別の fk_IFSTexture 型変数に読み込んでいます。

- 30 行目の「setScale」関数は fk_Model 型の変数にある機能の一つで、全体の拡大縮小を行います。今回のように、親モデルとなっているときに setScale を行うと、子モデル全体に対しても適用されます。

9.2 FK Performer からの読み込み

Performer データの読み込み

```
 1: #include <FK/FK.h>
 2:
 3: int main(int, char **)
 4: {
 5:     fk_AppWindow    window;
 6:     fk_Performer    chara;
 7:     fk_Vector       origin(0.0, 0.0, 0.0);
 8:     int             motionID;
 9:
10:     // 形状読み込み
11:     if(chara.loadObjectData("girl.mqo") == false) {
12:         fl_alert("Object read error");
13:     }
14:
15:     // 設定読み込み
16:     if(chara.loadJointData("girl_setup.fkc") == false) {
17:         fl_alert("Joint read error");
18:     }
19:
20:     // 歩きモーション読み込み
21:     if(chara.loadMotionData("walk.fkm") == false) {
22:         fl_alert("Walk motion read error");
23:     }
24:
25:     // パンチ 1 読み込み
26:     if(chara.loadMotionData("punch1.fkm") == false) {
27:         fl_alert("Punch motion read error");
28:     }
29:
30:     window.entry(chara);
31:     window.setCameraPos(0.0, 25.0, 100.0);
32:     window.setCameraFocus(0.0, 25.0, 0.0);
33:     window.setSize(800, 600);
34:     window.setBGColor(0.6, 0.7, 0.8);
35:     window.open();
36:
37:     motionID = -1;
38:     while(window.update() == true) {
39:
40:         if(window.getSpecialKeyStatus(FK_RIGHT) == FK_SW_PRESS) {
41:             chara.getBaseModel()->glRotateWithVec(origin, fk_Y, FK_PI/180.0);
42:         }
43:
44:         if(window.getSpecialKeyStatus(FK_LEFT) == FK_SW_PRESS) {
45:             chara.getBaseModel()->glRotateWithVec(origin, fk_Y, -FK_PI/180.0);
46:         }
47:
48:         // モーション選択
49:         if(window.getKeyStatus('1') == true) {
50:             motionID = 0;
51:         } else if(window.getKeyStatus('2') == true) {
52:             motionID = 1;
53:         } else if(window.getKeyStatus(' ') == true) {
54:             motionID = -1;
55:         }
56:
57:         if(motionID == 0) {
58:             // 歩きモーションはリピート再生
59:             chara.playMotion(0);
60:         } else if(motionID == 1) {
61:             // パンチは、終了しているかどうかを判定
62:             if(chara.isMotionFinished(motionID) == true) {
63:                 // 終了していたら巻き戻し
64:                 chara.setNowFrame(motionID, 0);
65:                 motionID = -1;
66:             } else {
67:                 // 終了していなかったら再生
68:                 chara.playMotion(motionID);
69:             }
70:         }
71:
72:
73:     }
74:     return 0;
75: }
```

解説

- 前の例では形状の動作 (モーション) がありませんでしたが、キャラクターモデル等はモーションを付加したい場合が多々あります。そのような場合、FK では「FK Performer」というアプリケーションを使ってモーションを作成します。Performer は
  `http://gamescience.jp/~rita/FKP/` からダウンロードできます。このページに使い方も載っていますので、利用を検討している人はこのページを参照して下さい。

- Performer を使う場合、プログラム中では
  「fk_Performer」という型を用います。6 行目がそれにあたります。今回は変数を 1 個しか用意していませんが、もちろん複数の変数や配列を用いることも可能です。

- fk_Performer では、3 種類のファイルを入力します。まず形状データそのものである MQO ファイルを入力するには 11 行目にあるように「loadObjectData」関数を用います。ただし、fk_IFSTexture と違ってテクスチャ画像を事前に入力する必要はなく、オブジェクト名も指定する必要はありません。

- 次に、16 行目にあるように「FKC ファイル」を読み込みます。これは、Performer による全体設定のためのファイルです。

- 次に、モーションを表すデータを入力します。モーションは一つのキャラクタに対して複数のものを使うことが想定されるため、Performer 側では個別のモーションを別々のファイルに保存しておきます。21,26 行目の「loadMotionData」関数がモーションを FK 側で入力する関数となりますが、今回は歩きとパンチの 2 種類のモーションを入力してみます。なお、入力したモーションはそれぞれに ID が付与され、最初に読み込んだものが 0 番、次が 1 番 · · · となります。

- 30 行目では、fk_Model と同様に fk_AppWindow に登録しています。

- 入力した形状データに対する位置や向きの制御ですが、これは 41,45 行目にあるように

  ```
  変数.getBaseModel()->関数 ()
  ```

  という記述で行います。関数は fk_Model 型で用いることのできた関数、他に「glTranslate」や「setScale」等を用いることが可能です。詳しい説明、特に「->」という部分の解説はやや高度になるのでここでは触れません。

- モーションの再生を行うには 59 行目や 68 行目にあるように「playMotion」関数を用います。引数にはモーションの ID を入れます。「モーションの再生」と書きましたが、厳密には形状のアニメーションを 1 コマ進めるという意味になります。

- モーションは、一回の再生で終わらせたい場合と、繰り返し再生したい場合があります。今回の場合、歩きのモーションは繰り返したいものであり、パンチのモーションは一回の再生で終わらせたいものと想定しています。繰り返したい場合は、59 行目にあるようにひたすら playMotion 関数を使います。最後まで再生された場合に最初に巻き戻し、改めて再生を行っていきます。

- 一方、パンチのように一回で終わらせたい場合は、あるモーションが既に再生が最後まで来ているかどうかを判断する必要があります。それを行っているのが 62 行目の「isMotionFinished」関数です。モーション ID を引数として関数を呼ぶと、もし再生が終了していた場合に true が、そうでない場合 false が返されます。今回のサンプルの場合、終了と判定された場合に 64 行目に進むことになります。

- 64 行目の「setNowFrame」関数は、モーションの経過コマを指定するものです。1 番目の引数がモーション ID、2 番目がコマの指定です。コマを 0 にするというのは、つまり最初に巻き戻すことになります。また、65 行目にあるように motionID に -1 を代入しておくことによって、あらためてこのモーションが再生されるのを防いでいます。

9.3 練習課題

問題 9-1

様々なモーションデータをプログラム中で再生せよ。

問題 9-2

FK Performer を使って自分でモーションを作って、それを再生せよ。

第10章　サウンドの再生

10.1　音関係のセットアップ

10.1.1　事前のセットアップ

C:\FK_VC13\redist にある「oalinst.exe」という名前のインストーラを実行してください。これは 1 回やっておけば Windows を再インストールしない限りは今後する必要はありません。

10.1.2　プロジェクト毎のセットアップ

音を利用するプロジェクトを作成した際には、ビルド後の _exe フォルダに対し、C:\FK_VC13\bin にある以下のファイルをコピーしてください。

- libogg.dll
- libvorbis.dll
- libvorbisfile.dll

10.2 BGM の再生

Ogg ファイルの再生

```
 1: #include <FK/FK.h>
 2: #include <FK/Audio.h>
 3:
 4: int main(int, char **)
 5: {
 6:     fk_AppWindow    window;
 7:     fk_Block        block(1.0, 1.0, 1.0);
 8:     fk_Model        blockModel;
 9:     fk_Vector       origin(0.0, 0.0, 0.0);
10:     fk_AudioStream  bgm;
11:     double          volume;
12:
13:     if(bgm.open("epoq.ogg") == false) {
14:         fl_alert("BGM Open Error");
15:     }
16:     bgm.setLoopMode(true);
17:
18:     blockModel.setShape(&block);
19:     blockModel.glMoveTo(3.0, 3.0, 0.0);
20:     blockModel.setMaterial(Yellow);
21:     window.entry(blockModel);
22:
23:     window.setCameraPos(0.0, 1.0, 20.0);
24:     window.setCameraFocus(0.0, 1.0, 0.0);
25:
26:     window.setSize(800, 600);
27:     window.setBGColor(0.6, 0.7, 0.8);
28:     window.open();
29:     window.showGuide(FK_GRID_XZ);
30:
31:     volume = 0.5;
32:     while(window.update() == true) {
33:         blockModel.glRotateWithVec(origin, fk_Y, FK_PI/360.0);
34:
35:         if(window.getKeyStatus('Z') == FK_SW_DOWN && volume < 1.0) {
36:             volume += 0.1;
37:         }
38:         if(window.getKeyStatus('X') == FK_SW_DOWN && volume > 0.0) {
39:             volume -= 0.1;
40:         }
41:
42:         bgm.setGain(volume);
43:         bgm.play();
44:     }
45:     return 0;
46: }
```

解説

- FK では音を出力するための入力として「wav 形式」と「ogg 形式」をサポートしています。wav 形式はほとんどの音を扱うソフトウェアで利用可能な形式ですが、サイズが大きいという欠点があります。ogg 形式は mp3 や aac 等と同じくファイルサイズが小さな形式です。サポートされているソフトウェアが多いとは言え

ませんが、ライセンスや特許に縛られずに利用できるという利点があり、フリーのプログラムでは広く利用されています。ogg 形式で音声を作成できるソフトウェアとしては、多機能なものとして「XRECODE[1]」、国産として「SoundEngine[2]」などがあります。

- 音を使うプログラムでは、先に記述したセットアップの他に、2 行目にあるように
  `#include <FK/Audio.h>` という記述を追加しておいて下さい。

- 10 行目にあるのが、BGM の読み込みや再生を行うための変数です。基本的に BGM は wav ファイルですと巨大になりますので、ogg ファイルのみを用います。

- 13 行目に ogg ファイルを指定することで再生の準備を行います。ファイルがない場合や、正常に入力できなかった場合はエラー用のダイアログを 14 行目で開くようにしています。

- 16 行目は、リピート再生を行うための設定です。このように「setLoopMode()」関数で true を引数として入力しておくと、曲の終端に来てから最初に戻って再生しなおします。false にすると、終端で再生を停止します。デフォルトでは false になっています。

- 31 行目で音量を制御するための変数「volume」の値を 0.5 に設定していますが、この値は Z,X キーを押すことで上下できるようなプログラムになっています。35 行目で「getKeyStatus()」関数を用いてキーの状態を取得していますが、これまでの「FK_SW_PRESS」ではなく「FK_SW_DOWN」となっています。これは、「押された瞬間」を意味します。「FK_SW_PRESS」の場合、押されている間はずっと true となります。そのため、volume の値はわずかな時間で大きく変動してしまいます。「FK_SW_DOWN」の場合、押された瞬間しか検知しないため、いわゆる「押さえっぱなし」の状況のときは false となります。

- 42 行目の「setGain()」関数は音量を設定するための関数です。1 で最大、0 で無音となります。

- 43 行目の「play()」関数ですが、これによって再生が行われます。これがゲーム中でサウンドを扱う特殊な面なのですが、この「play()」関数は再生している間、何度も呼ぶ必要があります。以下、少し詳しく説明します。(理解しなくても利用には差し支えありません。)

  音データは全てを読み込んでしまうと膨大なメモリを消費します。そのため、音データを「チャンク」と呼ばれる細かなデータに分け、チャンクを少しずつファイルから読み込み、システムに流すという方法で内部では再生を行っています。この「play()」という関数は、内部ではチャンクをファイルから取り出し、システムの状況を検査してチャンクを新たに受け付けられる状況であれば流すという処理を行っているのです。そのため、かなり頻繁にこの関数を呼ばないと途中で再生が止まってしまうのです。

[1] http://www.xrecode.com/
[2] http://soundengine.jp/software/soundengine/

10.3 サウンドエフェクト (SE) の制御

WAV ファイルの入力

```
 1: #include <FK/FK.h>
 2: #include <FK/Audio.h>
 3:
 4: int main(int, char **)
 5: {
 6:     fk_AppWindow        window;
 7:     fk_Block            block(1.0, 1.0, 1.0);
 8:     fk_Model            blockModel;
 9:     fk_Vector           origin(0.0, 0.0, 0.0);
10:     fk_AudioWavBuffer   se[2];
11:     bool                seFlag[2];
12:     int                 i;
13:
14:     if(se[0].open("MIDTOM2.wav") == false) {
15:         fl_alert("Sound(BD) Open Error");
16:     }
17:     if(se[1].open("SDCRKRM.wav") == false) {
18:         fl_alert("Sound(SD) Open Error");
19:     }
20:
21:     blockModel.setShape(&block);
22:     blockModel.glMoveTo(3.0, 3.0, 0.0);
23:     blockModel.setMaterial(Yellow);
24:     window.entry(blockModel);
25:
26:     window.setCameraPos(0.0, 1.0, 20.0);
27:     window.setCameraFocus(0.0, 1.0, 0.0);
28:
29:     window.setSize(800, 600);
30:     window.setBGColor(0.6, 0.7, 0.8);
31:     window.open();
32:     window.showGuide(FK_GRID_XZ);
33:
34:     for(i = 0; i < 2; i++) {
35:         seFlag[i] = false;
36:         se[i].setGain(1.0);
37:     }
38:
39:     while(window.update() == true) {
40:         blockModel.glRotateWithVec(origin, fk_Y, FK_PI/360.0);
41:
42:         if(window.getKeyStatus('Z') == FK_SW_DOWN) {
43:             seFlag[0] = true;
44:             se[0].seek(0.0);
45:         }
46:
47:         if(window.getKeyStatus('X') == FK_SW_DOWN) {
48:             seFlag[1] = true;
49:             se[1].seek(0.0);
50:         }
51:
52:         for(i = 0; i < 2; i++) {
53:             if(seFlag[i] == true) {
54:                 seFlag[i] = se[i].play();
55:             }
56:         }
57:     }
58:     return 0;
59: }
```

解説

- このプログラムは、先ほどの BGM 再生よりもやや高度な処理としてサウンドエフェクト (SE) を制御してみます。SE は、打撃音やジャンプの音など、瞬間的〜数秒程度で再生が終わる音素材のことです。SE は BGM と違って特定条件下で何度も再生を行うことが多いので、やや扱いが複雑となります。
- 10 行目の「fk_AudioWavBuffer」は SE 用の音素材の読み込みや再生を行うための型で、wav 形式に対応しています。もし ogg 形式のファイルを読み込みたい場合は「fk_AudioOggBuffer」を使用して下さい。型名 (クラス名) が異なる以外は、使い方は同じです。

  なお、前節の「fk_AudioStream」との違いですが、
  fk_AudioStream ではチャンクに分けて読み込みを行うのに対し、こちらの「fk_AudioWavBuffer」はファイルのデータを全てメモリ上に読み込んでしまいます。メモリを多く利用しますが、高速な動作に対応できます。
- 14〜19 行目でそれぞれ個別に SE 素材となる音ファイルを入力しておきます。
- 35 行目は、bool 型という「true」か「false」しか格納できない変数 (の配列) です。36 行目は前節のプログラムと同様、音量設定です。
- 先に 52〜56 行目を説明します。54 行目にあるように、「play()」関数で音を再生します。今回はリピート再生を無効としていますので、終端で再生が終わります。「play()」関数は音が終端にきた場合には「false」を返すようになっています。結果として、while でのループのたびに play() が実行され、再生が終わってない場合は `seFlag[i]` に true が入っているため次回のループでも再生され、再生が終わったら `seFlag[i]` に false が入るので、54 行目は実行されずに再生もなされないという仕組みです。
- 44,49 行目にある「seek()」という関数は、再生箇所を変更する関数で、単位は「秒」になっています。今回は「0」を指定することで、最初まで巻き戻しを行っていることになります。このとき、もし `seFlag[0]` や `seFlag[1]` が false になっていた場合、true に変更して再び再生が行われるようにしています。

  この「seek()」関数による指定ですが、経験則上やや不正確です。実際に指定した箇所からピンポイントに再生を行いたい場合は、数値をプログラム上で微調整して試行錯誤を行う必要があります。

10.4 音制御クラスの色々な機能

これまでにサンプルプログラム中で紹介した機能の他に、fk_AudioStream, fk_AudioWavBuffer, fk_AudioOggBuffer には制御を行う関数があります。ここではそれを紹介しておきます。

pause()

BGM や SE の再生を停止します。その後 play() を実行すると、止めた箇所から再び再生が始まります。

tell()

現時点での再生位置を取得しますが、やや不正確です。以下の様な記述をします。

```
fk_AudioStream    bgm;
double            playTime;
              :
              :
playTime = bgm.tell();
```