

# Springhead Users Manual

Yuichi Tazaki and Springhead Development Team.



# 目次

第 1 章	はじめに	7
第 2 章	Getting Started	9
2.1	ダウンロード	9
2.2	SVN から入手する	9
2.3	開発環境	9
2.4	ライブラリのビルド	10
2.5	サンプルプログラムのビルド	11
2.6	アプリケーションの作成	11
第 3 章	Springhead の構成	17
3.1	ディレクトリ構成	17
3.2	ライブラリ構成	17
3.3	クラス・API の命名規則	17
3.4	インタフェースとディスクリプタ	18
第 4 章	Base	21
4.1	行列・ベクトル演算	21
4.2	スマートポインタ	27
4.3	その他の機能	27
第 5 章	Foundation	29
5.1	実行時型情報	29
5.2	オブジェクト	29
5.3	ネームマネージャとシーン	31
5.4	タイマ	32
第 6 章	Collision	35

6.1	概要 . . . . .	35
6.2	形状の作成 . . . . .	35
6.3	物性の指定 . . . . .	41
6.4	幾何情報の計算 . . . . .	42
第 7 章	Physics	45
7.1	概要 . . . . .	45
7.2	Physics SDK . . . . .	45
7.3	シーン . . . . .	46
7.4	剛体 . . . . .	48
7.5	関節 . . . . .	51
7.6	関節系の逆運動学 . . . . .	58
7.7	接触 . . . . .	65
7.8	関節座標系シミュレーション . . . . .	65
7.9	ギア . . . . .	65
第 8 章	Graphics	67
8.1	概要 . . . . .	67
8.2	Graphics SDK . . . . .	67
8.3	シーン . . . . .	68
8.4	描画アイテム . . . . .	71
8.5	フレーム . . . . .	72
8.6	カメラ . . . . .	74
8.7	ライト . . . . .	74
8.8	マテリアル . . . . .	75
8.9	メッシュ . . . . .	75
8.10	レンダラ . . . . .	75
第 9 章	FileIO	83
9.1	概要 . . . . .	83
9.2	FileIO SDK . . . . .	83
9.3	ファイルフォーマット . . . . .	84
9.4	ファイルのロード . . . . .	87
9.5	ファイルのセーブ . . . . .	87
9.6	インポート情報の管理 . . . . .	88

---

第 10 章	HumanInterface	89
10.1	概要 . . . . .	89
10.2	HumanInterface SDK . . . . .	89
10.3	クラス階層とデータ構造 . . . . .	89
10.4	実デバイス . . . . .	92
10.5	キーボード・マウス . . . . .	92
10.6	ジョイスティック . . . . .	95
10.7	トラックボール . . . . .	95
10.8	Spidar . . . . .	95
第 11 章	Creature	97
11.1	Creature モジュールの構成 . . . . .	97
第 12 章	Framework	99
12.1	概要 . . . . .	99
12.2	Framework SDK . . . . .	100
12.3	Framework シーン . . . . .	101
12.4	シーンのロードとセーブ . . . . .	104
12.5	Framework オブジェクト . . . . .	104
12.6	アプリケーションクラス . . . . .	105
12.7	ウィンドウ . . . . .	109
12.8	Framework を用いたシミュレーションと描画 . . . . .	109
12.9	デバッグ描画 . . . . .	110
12.10	力覚インタラクションのためのアプリケーション . . . . .	114
第 13 章	Python 言語との連携	117
13.1	利用法 . . . . .	117
第 14 章	トラブルシューティング	119
14.1	概要 . . . . .	119
14.2	プログラムのビルド . . . . .	119
14.3	物理シミュレーション関連 . . . . .	120
14.4	ファイル関連 . . . . .	120
14.5	その他のトラブル . . . . .	120
索引		121



## 第 1 章

# はじめに

このドキュメントはのマニュアルです。Springhead は物理シミュレーションやコンピュータグラフィクスなど使用するソフトウェアの作成を総合的に支援する C++ ライブラリです。

Springhead は多数のモジュールから成り立っており，中には仕様が流動的なものもあります。本ドキュメントでは，中でも使用頻度が高く，安定性の高い機能に焦点をしばって解説します。





## 第 2 章

# Getting Started

Springhead をダウンロードしてから使えるようにするまでの流れを説明します。

### 2.1 ダウンロード

Springhead のウェブサイト <http://springhead.info/wiki/> から zip アーカイブをダウンロードできます。ただし、アーカイブの更新は必ずしも頻繁ではありません（よくないことですが）ので、最新のコードが入手できない可能性があります。常に最新のコードを使用したい人は、次に説明する Subversion レポジトリからコードを入手してください。

### 2.2 SVN から入手する

Springhead は Subversion を用いてバージョン管理を行っています。この文書の執筆時点で Springhead の Subversion レポジトリは

`http://springhead.info/spr2/Springhead2/trunk`

です。レポジトリからのコードのダウンロードは任意のユーザが行えますが、コードをコミットするには開発者として登録されている必要があります。

### 2.3 開発環境

Springhead は処理系非依存の思想のもとで開発されています。このため、原理的には Windows, Mac, Unix などの多くの処理系で動作するはずです。しかしながら、ほとんどの開発メンバーが Windows 上の Visual Studio を用いて開発を行っているため、それ以外の環境で問題無く動作する保証は残念ながらありません（多分動かないでしょう）。

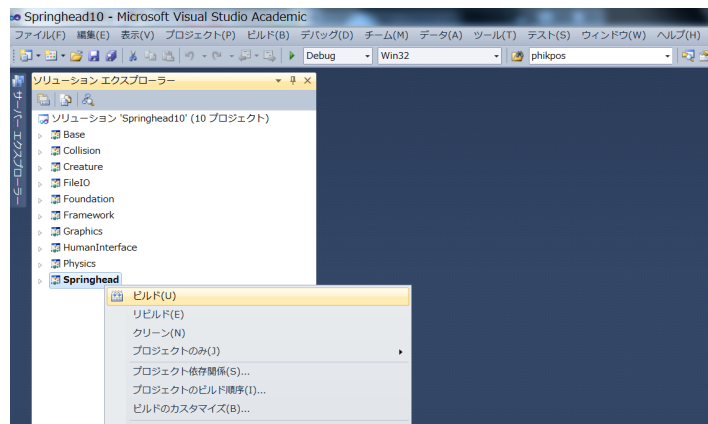


Fig. 2.1 Building the library

したがって、現状ではユーザーにも Windows + Visual Studio という環境での使用を推奨します。Windows や Visual Studio のバージョンについては、Windows XP/Vista/7, Visual Studio 2008/2010 では問題なく動作します。

## 2.4 ライブラリのビルド

Springhead を入手したら、まずライブラリをビルドします。以下では、Springhead を保存したディレクトリを C:\Springhead2 と仮定して話を進めます。まず、Visual Studio で以下のソリューションファイルを開いて下さい。

C:\Springhead2\src\Springhead\*\*.sln

ここで \*\* には Visual Studio のバージョン番号が入ります。Visual Studio 2010 なら Springhead10.sln, Visual Studio 2008 なら Springhead9.sln を開いて下さい。その他のソリューションファイル、プロジェクトファイルも同様の規則でナンバリングしてあります。

ソリューションを開いたら Fig. 2.1 に示すように Springhead プロジェクトをビルドしてください。ビルドに成功したら C:\Springhead2\lib\win32\ ディレクトリにライブラリファイル Springhead\*.lib が作成されるはずです。

Table 2.1 ビルドの設定ごとに異なるいくつかの構成が用意されています。ユーザアプリケーションの都合に合わせて使い分けてください。

Table 2.1 Build configurations

構成名	ビルド設定	作成されるライブラリファイル名
Release	multithread	Springhead**.lib
Debug	multithread, Debug	Springhead**D.lib
ReleaseDLL	multithread, DLL	Springhead**M.lib
DebugDLL	multithread, Debug, DLL	Springhead**MD.lib

## 2.5 サンプルプログラムのビルド

サンプルプログラムをビルドするには

```
C:\Springhead2\src\Samples\All**.sln
```

を開きます。ビルドしたいサンプルをスタートアッププロジェクトに設定し、ビルド、実行してください。各プロジェクトは C:\Springhead2\lib\win32\Springhead\*\*.lib をリンクするように設定されていますので、サンプルをビルドする前にライブラリのビルドが済んでいなければなりません。

残念なことです、すべてのサンプルプログラムが問題なく動作する状態には維持されていません。Physics/BoxStack や Physics/Joints が比較的良くメンテナンスされていますので試してみてください。

## 2.6 アプリケーションの作成

Springhead を使って簡単なアプリケーションプログラムを作成する道筋を説明します。以下では Visual Studio 2010 を想定します。

### プロジェクトの作成

「Visual C++ Win32 プロジェクト」を作成します。プロジェクト名は好きな名前を付けてください。アプリケーションの設定で「コンソールアプリケーション」を選び、空のプロジェクトをチェックします。

プロジェクトを作成したらソースファイルを作成します。ここでは仮に main.cpp とします。

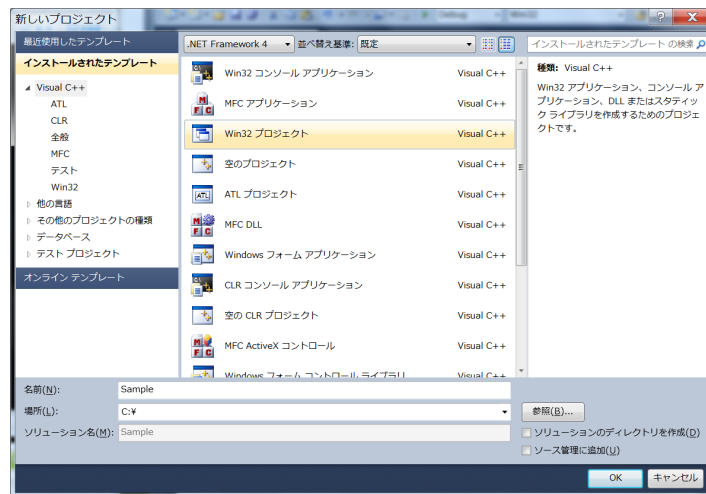


Fig. 2.2 Create new project



Fig. 2.3 Project configuration

## ソースコードの編集

作成した `main.cpp` に Table 2.2 に示すコードを書き込んでください。これが Springhead を使用した（ほぼ）最小のプログラムコードです。

## プロジェクト設定

ビルドするまえにいくつかのプロジェクト設定が必要です。まずプロジェクトのプロパティページを開き、構成を「すべての構成」としてください。次に「C/C++」と「全般」の追

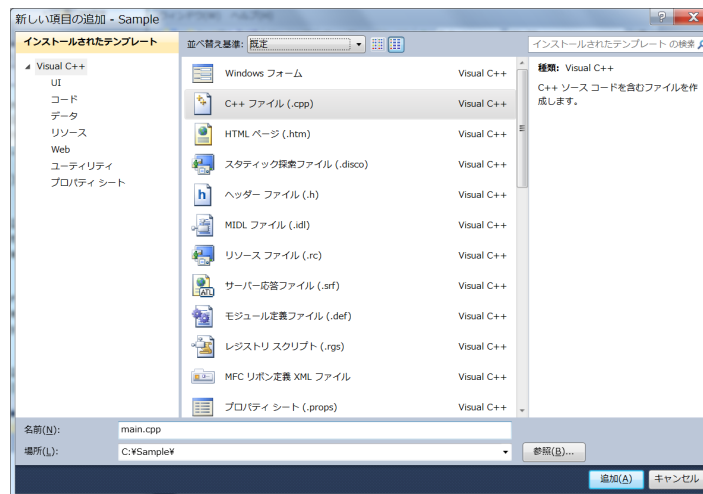


Fig. 2.4 Create source file

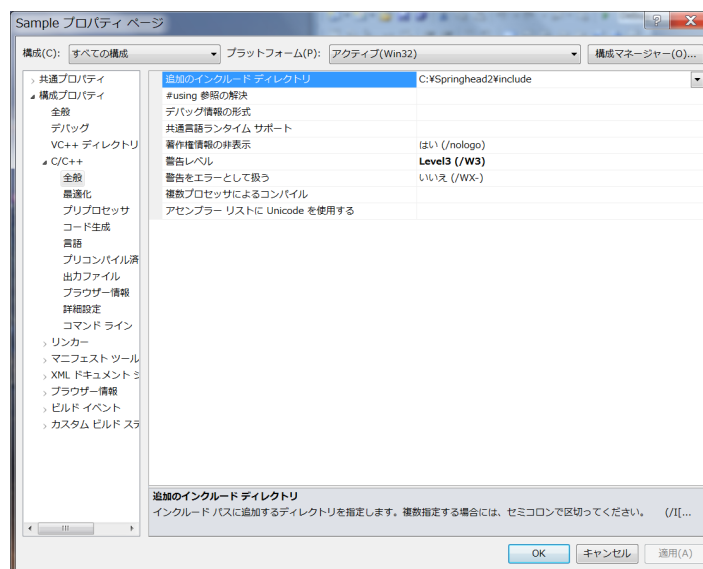


Fig. 2.5 Add include path

「追加のインクルードディレクトリ」に、Fig. 2.5 のように Springhead のインクルードファイルへのパスを指定してください。さらに、「リンカー」の「全般」の「追加のライブラリディレクトリ」に Fig. 2.6 のように Springhead のライブラリファイルへのパスを指定します。

今度は構成を「Debug」にします。「C/C++」の「コード生成」の「ランタイムライブラリ」を「マルチスレッド デバッグ (/MTd)」にします。次に「リンカー」の「入力」の「追加の依存ファイル」に Springhead10D.lib を追加してください。

最後に構成を「Release」に切り替えて同様の設定をします。ランタイムライブラリを「マルチスレッド (/MT)」として、追加の依存ファイルに Springhead10.lib を追加し

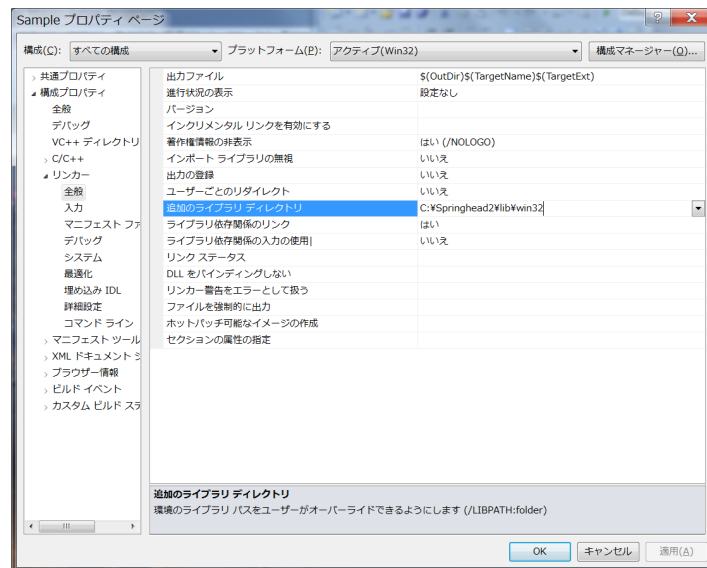


Fig. 2.6 Add library path

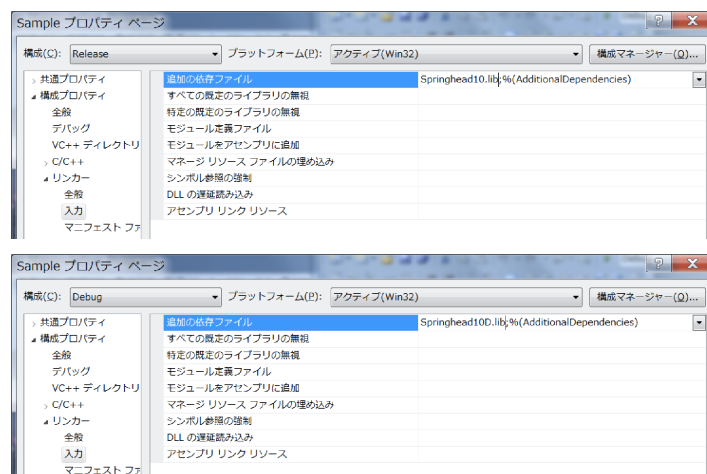


Fig. 2.7 Specify library file

ます。

## ビルド・実行

以上で準備完了です。ビルド (F7) して、実行 (F5) してみてください。Fig. 2.8 のような画面が出てくれば成功です。

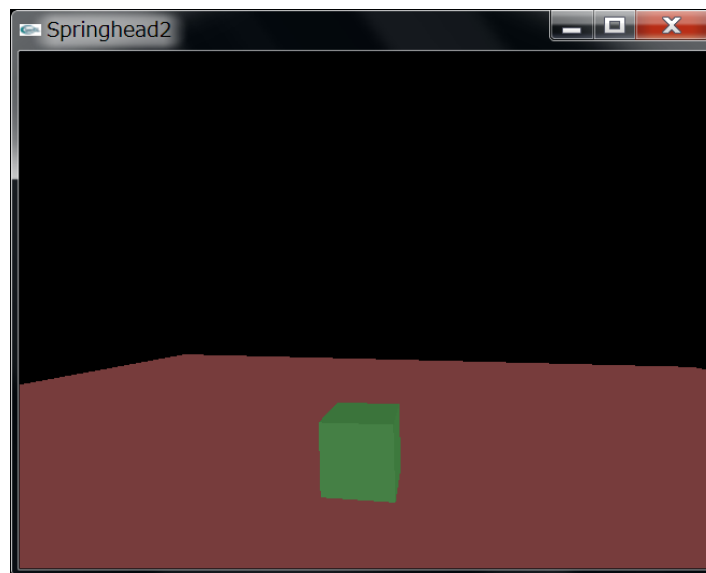


Fig. 2.8 Program running

Table 2.2 Simplest program code

```
#include <Springhead.h>
#include <Framework/SprFWApp.h>
using namespace Spr;

class MyApp : public FWApp{
public:
    virtual void Init(int argc = 0, char* argv[] = 0){
        FWApp::Init(argc, argv);

        PHSdkIf* phSdk = GetSdk()->GetPHSdk();
        PHSceneIf* phscene = GetSdk()->GetScene()->GetPHScene();
        CDBoxDesc bd;

        // 床を作成
        PHSolidIf* floor = phscene->CreateSolid();
        floor->SetDynamical(false);
        bd.bboxsize = Vec3f(5.0f, 1.0f, 5.0f);
        floor->AddShape(phSdk->CreateShape(bd));
        floor->SetFramePosition(Vec3d(0, -1.0, 0));

        // 箱を作成
        PHSolidIf* box = phscene->CreateSolid();
        bd.bboxsize = Vec3f(0.2f, 0.2f, 0.2f);
        box->AddShape(phSdk->CreateShape(bd));
        box->SetFramePosition(Vec3d(0.0, 1.0, 0.0));

        GetSdk()->SetDebugMode(true);
    }
} app;

int main(int argc, char* argv[]){
    app.Init(argc, argv);
    app.StartMainLoop();
    return 0;
}
```



## 第 3 章

# Springhead の構成

### 3.1 ディレクトリ構成

Springhead のディレクトリ構成を Fig. 3.1 に示します .

### 3.2 ライブラリ構成

Springhead は複数のモジュールから構成されています . Table 3.1 にモジュール一覧を示します . Table 3.2 にモジュール間の依存関係を示します . 通常 , ユーザは Springhead を使用するにあたってこれらの依存関係を陽に意識する必要はありません . また , 何らかの事情で Springhead の特定の機能 (たとえば物理シミュレーション) のみを用いたいという場合に対応できるように , モジュール間の依存関係はなるべく疎になるように設計されています . したがってこのような場合には用途に応じて必要なモジュールのみを使えるようになっています .

### 3.3 クラス・API の命名規則

各モジュールに含まれるクラスの名前には , Table 3.1 に示されるようなモジュール固有のプリフィックスがつきます (例: Physics モジュールの `PHSolid` , Collision モジュールの `CDShape`) . 一部にはこのルールにしたがわないクラスも存在します (例: Foundation モジュールの `Object`) .

API(クラスのメンバ関数) にも緩い命名規則があります . API 名は基本的に (動詞 + 目的語) という形式で処理内容を端的に表現します . また , 単語の先頭文字のみ大文字 , その他は小文字で表記します . 例としては `PHSolid::SetMass` , `GRSdk::CreateScene` などです .

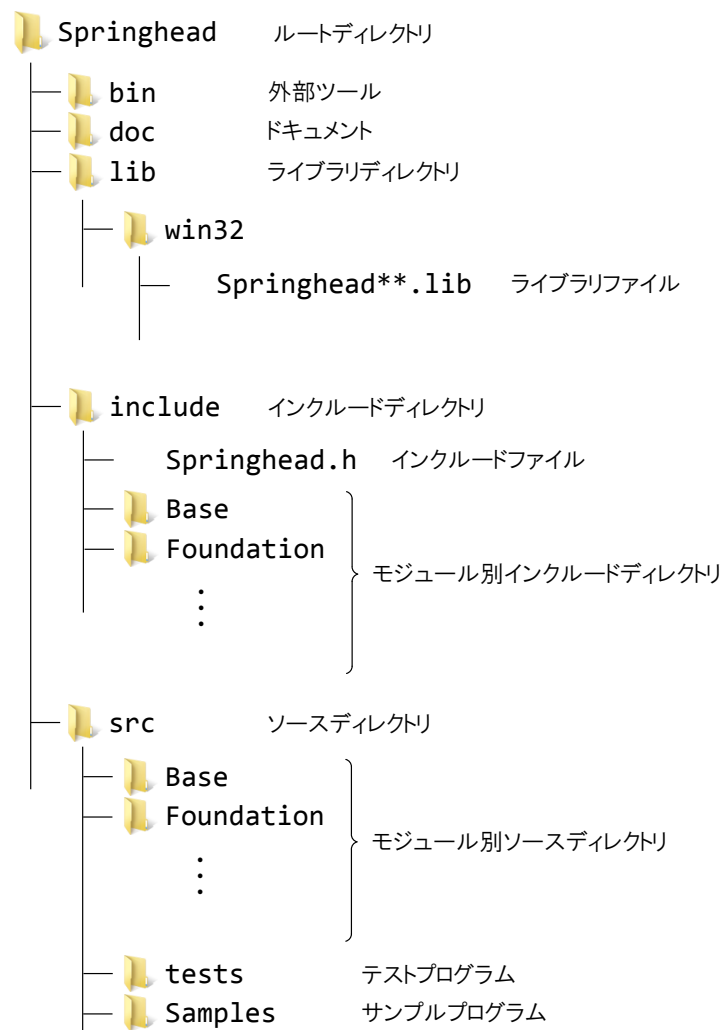


Fig. 3.1 Directory tree of Springhead

### 3.4 インタフェースとディスクリプタ

Springhead では仕様と実装を明確に分離するために、インタフェースクラスと実装クラスが分けられています。ユーザはインタフェースクラスのみを使用して Springhead の機能を利用します。ただし、Base と Foundation モジュールにあるごく基本的なクラス、および Framework のアプリケーションクラスは例外となっています。

また、Springhead のクラスにはそれぞれにディスクリプタが用意されています。ディスクリプタとは、そのクラスの読み書き可能な属性のみを集めた構造体です。ディスクリプタを利用することで、同じ設定のインスタンスを多数設定することが用意になります。また、ディスクリプタはファイルへのデータの保存や読み込みにおいても役立ちます。

Table 3.1 Springhead modules

モジュール名	プリフィックス	機能
Base	-	行列・ベクトル演算，スマートポインタ， その他基本機能
Foundation	UT	Springhead の基本クラス，実行時型情報
Collision	CD	衝突判定
Physics	PH	物理計算
Graphics	GR	シーングラフ，描画
FileIO	FI	ファイル入出力
HumanInterface	HI	ヒューマンインタフェースデバイスや インタラクション
Creature	CR	バーチャルクリーチャ
Framework	FW	モジュール間の連携と アプリケーション作成支援

Table 3.2 Module dependencies

モジュール名										
Base	-	-	-	-	-	-	-	-	-	-
Foundation	○	-	-	-	-	-	-	-	-	-
Collision	○	○	-	-	-	-	-	-	-	-
Physics	○	○	○	-	-	-	-	-	-	-
Graphics	○	○	-	-	-	-	-	-	-	-
FileIO	○	○	-	-	-	-	-	-	-	-
HumanInterface	○	○	-	-	-	-	-	-	-	-
Creature	○	○	-	○	-	-	-	-	-	-
Framework	○	○	-	○	○	○	○	-	-	-

以下に Physics モジュールの剛体を表す PHSolid クラスを例にとって説明します．

```
// given PHSolidIf* phScene,
```

```
PHSolidDesc desc;
```

```
desc.mass = 1.0;
```

```
PHSolidIf* solid = phScene->CreateSolid(desc);
```

上のコードで `PHSolidDesc` は `PHSolid` クラスのディスクリプタです。まずそのメンバ変数 `mass` に値をセットすることで剛体の質量を設定しています。次に、剛体を作成するために `CreateSolid` 関数が呼ばれます。ここで `CreateSolid` は物理シーンを表す `PHScene` クラスのメンバ関数です。実際には `PHScene` クラスのインタフェース `PHSceneIf` を取得する必要がありますが、ここでは既に得られているとしています。剛体が作成されると、`CreateSolid` からインタフェース `PHSolidIf` のポインタが返されます。これ以降の剛体の操作はこのインタフェースを介して行います。

```
solid->SetMass(5.0);
```

基本的に、ディスクリプタを介して設定可能な属性はインタフェースの `Get/Set` 系関数を使って取得、設定ができるようになっています。場合に応じて便利な方を使ってください。

Springhead オブジェクトはすべて内部でメモリ管理されていますので、ユーザが明示的に `delete` する必要はありません（また、してはいけません）。Create されたオブジェクトはプログラムの終了時に自動的に破棄されます。

## より詳しく知りたい人は

以降の章では各モジュールについてより詳しく説明します。Springhead を利用する上で、すべてのモジュールを詳しく理解する必要はありません。必要に応じて参照してください。

## 第 4 章

# Base

Base モジュールは基本機能の集合体です．以下では項目別にそれらについて解説します．

### 4.1 行列・ベクトル演算

Table 4.1 によく使われる行列・ベクトルクラスを示します．末尾の整数はベクトルや行列のサイズを表し，*f*，*d* はそれぞれ `float` 型，`double` 型に対応します．

#### ベクトル

ベクトル型は物体の位置や速度，力などの物理量を表現するために頻繁に使われます．例えば `double` 型の要素からなる 3 次元ベクトル *x* を定義するには

```
Vec3d x;
```

とします．要素アクセスは `[]` 演算子を用います．

```
x[0];    // 0-th element
```

この他，`Vec[2|3][f|d]` については `.x`，`.y`，`.z` でも要素アクセスできます．

任意の固定サイズのベクトルも使えます．`float` 型の 10 次元ベクトルは

```
TVector<10, float> x;    // 10-dimensional float vector
```

とします．可変長ベクトルは

```
VVector<float> x;
x.resize(10);           // can be resized at any time
```

Table 4.1 Matrix and vector classes

クラス名	機能
Vec[2 3 4][f d]	ベクトル
Matrix[2 3 4][f d]	行列
Quaternion[f d]	単位クォータニオン
Affine[f d]	アフィン変換
Pose[f d]	3次元ベクトルとクォータニオンの複合型

で使えます.

基本的な演算は一通りサポートされています .

```
Vec3d a, b, c;
double k;

c = a + b;           // addition
a += b;

c = a - b;           // subtraction
a -= b;

b = k * a;           // multiply vector by scalar
a *= k;

k = x * y;           // scalar product

x % y;               // vector product (3D vector only)
```

すべてのベクトル型について以下のメンバ関数が使えます .

```
a.size();           // number of elements
a.norm();           // norm
a.square();         // square of norm
a.unitize();        // normalize
b = a.unit();       // normalized vector
```

## 行列

行列は平行移動や回転などの変換や、剛体の慣性モーメントを表現するために使われます。例えば、`double` 型の要素からなる  $3 \times 3$  行列  $A$  は次のように定義します。

```
Matrix3d A;
```

要素アクセスは `[]` 演算子を用います。

```
x[0][1];    // element at 0-th row, 1-th column
```

任意の固定サイズの行列も使えます。メモリ上に列方向に要素が整列した行列は

```
TMatrixCol<2, 3, float> M;    // column-oriented 2x3 matrix
```

要素が行方向に整列した行列は

```
TMatrixRow<2, 3, float> M;    // row-oriented 2x3 matrix
```

となります。ちなみにさきほどの `Matrix3d` は `TMatrixCol<3,3,double>` と等価です。

可変サイズ行列は

```
VMatrixCol<float> M;
M.resize(10, 13);    // column-oriented variable matrix
```

で使えます。`VMatrixCol` では要素はメモリ上で列方向に並びます。一方 `VMatrixRow` では行方向に要素が並びます。

行列型についても、ベクトル型と同様の四則演算がサポートされています。行列とベクトル間の演算は次のようになります。

```
Matrix3d M;
Vec3d a, b;

b = M * a;    // multiplication
```

すべての行列型について以下のメンバ関数で行数および列数が取得できます。

```
M.height();    // number of rows
M.width();     // number of columns
```

$2 \times 2$ ,  $3 \times 3$  行列については以下の静的メンバ関数が用意されています。

```

Matrix2d N;
Matrix3d M;
double theta;
Vec3d axis;

// methods common to Matrix2[f|d] and Matrix3[f|d]
M = Matrix3d::Zero();           // zero matrix; same as M.clear()
M = Matrix3d::Unit();           // identity matrix
M = Matrix3d::Diag(x,y,z);      // diagonal matrix

N = Matrix2d::Rot(theta);       // rotation in 2D

M = Matrix3d::Rot(theta, 'x');   // rotation w.r.t. x-axis
                                // one can specify 'y' and 'z' too
M = Matrix3d::Rot(theta, axis);  // rotation along arbitrary vector

```

## アフィン変換

アフィン変換は主にグラフィクスにおける変換を指定するために使用します。アフィン変換型 `Affine[f|d]` は  $4 \times 4$  行列としての機能を備えています。加えて以下のメンバ関数が使えます。

```

Affinef A;
Matrix3f R;
Vec3f p;

R = A.Rot();           // rotation part
p = A.Trn();           // translation part

```

また、よく使用するアフィン変換を生成する静的メンバが用意されています。

```

A = Affinef::Unit();           // identity transformation
A = Affinef::Trn(x, y, z);      // translation
A = Affinef::Rot(theta, 'x');   // rotation w.r.t. x-axis
                                // one can specify 'y' and 'z' too
A = Affinef::Rot(theta, axis);  // rotation w.r.t. arbitrary axis

```



```
A = Affinef::Scale(x, y, z);    // scaling
```

## クォータニオン

クォータニオンは主に物理計算における剛体の向きや回転を表現するために使います。  
クォータニオンは4次元ベクトルの基本機能を備えています。  
要素アクセスは [] 演算子に加えて以下の方法が使えます。

```
Quaterniond q;
q.w;                // same as q[0]
q.x;                // same as q[1]
q.y;                // same as q[2]
q.z;                // same as q[3]
q.V();              // vector composed of x,y,z elements
```

演算は以下のように行います。まず、クォータニオン同士の積は回転の合成を表します。

```
Quaterniond q, q0, q1;
q0 = Quaterniond::Rot(Rad(30.0), 'x'); // 30deg rotation along x-axis
q1 = Quaterniond::Rot(Rad(-90.0), 'y'); // -90deg rotation along y-axis

q = q1 * q0;
```

つぎに、クォータニオンと3次元ベクトルとの積は、ベクトルの回転を表します。

```
Vec3d a(1, 0, 0);
Vec3d b = q0 * a;
```

このように、クォータニオンは基本的に回転行列を同じような感覚で使えます。  
Quaterniond[f|d] には以下のメンバ関数があります。まず回転軸と回転角度を取得するには

```
Vec3d axis = q.Axis();        // rotation axis
double angle = q.Theta();     // rotation angle
```

とします。また、逆回転を表す共役クォータニオンを得るには

```
q.Conjugate();                // conjugate (reverse rotation)
```

```

Quaterniond y;
y = q.Conjugated();    // return conjugated quaternion
y = q.Inv();           // return inverse (normalized conjugate)

```

とします。Conjugate はそのクォータニオン自体を共役クォータニオンに変換するのに対し、Conjugated は単位共役クォータニオンを返します。Inv は Conjugated とほぼ等価ですが、戻り値のノルムが 1 となるように正規化を行います。回転を表すクォータニオンは理論上は必ずノルムが 1 なので正規化は不要ですが、実際は数値計算における誤差で次第にノルムがずれてくることがあります。このような誤差を補正するために適宜正規化を行う必要があります。

回転行列と相互変換するには以下のようにします。

```

Matrix3d R = Matrix3d::Rot(Rad(60.0), 'z');
q.FromMatrix(R);    // conversion from rotation matrix
q.ToMatrix(R);      // conversion to rotation matrix

```

FromMatrix は渡された回転行列 R と等価なクォータニオンとして q を設定します。一方 ToMatrix は、参照渡しされた R を q と等価な回転行列として設定します。

同様に、以下はオイラー角との相互変換を行います。

```

Vec3d angle;
q.ToEuler(angle);    // to Euler angle
q.FromEuler(angle);  // from Euler angle

```

最後に、以下の関数は 2 つのベクトルに対し、片方をもう片方に一致されるような回転を表すクォータニオンを求めます。一般に 2 つのベクトルを一致させる回転は一意ではありませんが、RotationArc は両方のベクトルに直交する軸に関する回転、いわば最短距離の回転を求めます。

```

Vec3d r0(1, 0, 0), r1(0, 1, 0);
q.RotationArc(r0, r1);    // rotation that maps r0 to r1

```

## ポーズ

ポーズは位置と向きの変換型です。役割としてはアフィン変換に似ていますが、全部で 7 つの成分で表現できるためアフィン変換よりもコンパクトです。ポーズは物理計算での剛体の位置と向きを表現するためなどに用います。

Pose[f|d] 型のメンバ変数は Pos と Ori の 2 つのみで、それぞれポーズの並進成分 (Vec3[f|d]) と回転成分 (Quaternion[f|d]) への参照を返します。

```
Posed P;  
P.Pos() = Vec3d(1, 2, 3);  
P.Ori() = Quaterniond::Rot(Rad(45.0), 'x');  
Vec3d p = P.Pos();  
Quaterniond q = P.Ori();
```

## 4.2 スマートポインタ

参照カウントにもとづくスマートポインタです。参照カウントが 0 になったオブジェクトのメモリを自動的に解放するためにユーザが手動で delete を実行する手間が省け、メモリリークの危険が低減できます。

参照カウントクラスは URefCount です。Springhead のほとんどのクラスは URefCount を継承しています。スマートポインタはテンプレートクラス URef です。以下に例を示します。

```
class A : public URefCount{};  
URef<A> a = new A();  
// no need to delete a
```

## 4.3 その他の機能

### UTString

文字列型です。現状では std::string と等価です。

### UTTypeDesc

Springhead のクラスが持つ実行時型情報です。

### UTTreeNode

ツリー構造の基本クラスです。



## 第 5 章

# Foundation

Foundation モジュールはすべての Springhead クラスの基本クラスを定義します。普通に使っている限り、ユーザが Foundation の機能を直接利用することは少ないでしょう。

### 5.1 実行時型情報

(ほとんど)すべての Springhead オブジェクトは実行時型情報 (RTTI) を持っています。C++ にも `dynamic_cast` などの RTTI 機能がありますが、これよりも大幅にリッチな型情報が提供されます。

実行時型情報のクラスは `IfInfo` です。`IfInfo` は次節で紹介する `Object` クラスから取得できます。

### 5.2 オブジェクト

ほとんどすべての Springhead オブジェクトは `Object` クラスから派生します。オブジェクトは複数の子オブジェクトを持つことができます。Springhead のデータ構造はオブジェクトが成すツリー構造によって出来上がっています。Foundation モジュールにおける `Object` からのクラス階層を Fig. 5.1 に示します。

まず `Object` クラスの子オブジェクトの作成・管理に関係する関数を紹介します。

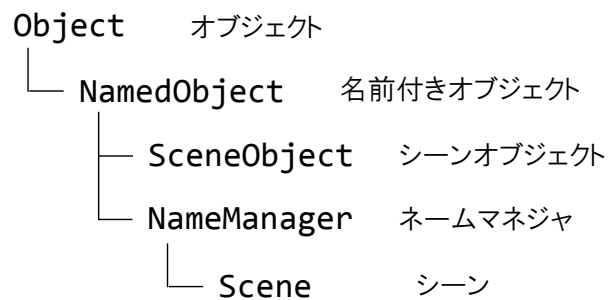


Fig. 5.1 Object class hierarchy

---

## ObjectIf

---

```
size_t NChildObject()
```

子オブジェクトの数を取得する。

```
ObjectIf* GetChildObject(size_t pos)
```

pos 番目の子オブジェクトを取得する。

```
bool AddChildObject(ObjectIf* o)
```

オブジェクト o を子オブジェクトとして追加する。正しく追加されたら true, それ以外は false を返す。

```
bool DelChildObject(ObjectIf* o)
```

オブジェクト o を子オブジェクトから削除する。正しく削除されたら true, それ以外は false を返す。

```
void Clear();
```

クリアする。

これらの関数は派生クラスによって実装されますので、追加できる子オブジェクトの種類や数などはクラスごとに異なります。また、Springhead を普通に使用する範囲内ではユーザがこれらの関数を直接呼び出す場面はないでしょう。

ストリーム出力のために以下の機能があります。

---

#### ObjectIf

---

```
void Print(std::ostream& os) const
```

オブジェクトの内容をストリーム os に出力する .

Print は、基本的にはそのオブジェクトの名前を出力し、子オブジェクトの Print を再帰的に呼び出します。ただし派生クラスによって Print で出力される内容がカスタマイズされている場合はその限りではありません。

NamedObject は名前付きオブジェクトです。NamedObject の派生クラスには名前を文字列で与えることができ、名前からオブジェクトを検索することができます。名前付きオブジェクトには、直接の親オブジェクト以外に、名前を管理するためのネームマネージャが対応します。

---

#### NamedObjectIf

---

```
const char* GetName()
```

名前を取得する。

```
void SetName(const char* n)
```

名前を設定する。

```
NameManagerIf* GetNameManager()
```

ネームマネージャを取得する。

名前付きオブジェクトからはさらにシーンオブジェクトが派生します。シーンオブジェクトからは周辺モジュールのオブジェクト (PHSolid, GRVisual など) が派生します。

---

#### SceneObjectIf

---

```
SceneIf* GetScene()
```

自身が所属するシーンを取得する。

## 5.3 ネームマネージャとシーン

ネームマネージャは名前付きオブジェクトのコンテナとして働き、それらの名前を管理します。また、ネームマネージャはそれ自身名前付きオブジェクトです。

NameManagerIf

---

NamedObjectIf\* FindObject(UTString name)

名前が name のオブジェクトを検索し，見つければそのオブジェクトを返す．見つからなければ NULL を返す．

シーンはシーンオブジェクトのコンテナです．シーンの基本クラスは Scene で，ここから各モジュールのシーン (PHScene, GRScene, FWScene など) が派生します．Scene クラスは特に機能を提供しません．

## 5.4 タイマ

タイマ機能も Foundation で提供されます．タイマクラスは UTTimer です．タイマを作成するには

```
UTTimerIf* timer = UTTimerIf::Create();
```

とします．UTTimer には以下の API があります．

[Get Set]Resolution	分解能の取得と設定
[Get Set]Interval	周期の取得と設定
[Get Set]Mode	モードの取得と設定
[Get Set]Callback	コールバック関数の取得と設定
IsStarted	動いているかどうか
IsRunning	コールバック呼び出し中
Start	始動
Stop	停止
Call	コールバック呼び出し

SetMode で指定できるモードには以下があります．

MULTIMEDIA	マルチメディアタイマ
THREAD	独立スレッド
FRAMEWORK	Framework が提供するタイマ
IDLE	Framework が提供するアイドルコールバック

マルチメディアタイマは Windows が提供する高機能タイマです．独立スレッドモードでは，タイマ用のスレッドが実行され Sleep 関数により周期が制御されます．FRAMEWORK



と IDLE モードを利用するには FWApp の `CreateTimer` 関数を用いる必要があります。基本的に FRAMEWORK モードでは GLUT のタイマコールバックが使われ、IDLE モードでは GLUT のアイドルコールバックが使われます。

Framework モジュールの FWApp を利用する場合は、FWApp の `CreateTimer` 関数を利用の方が便利でしょう。



## 第 6 章

# Collision

### 6.1 概要

Collision モジュールは物理計算の基礎となる衝突判定機能を提供します．事実上 Collision モジュールは Physics モジュールのサブモジュールとなっており，両者は密接に依存しています．ユーザは主として剛体に衝突判定用形状を割り当てる際に Collision モジュールの機能を利用することになります．

Collision モジュールのクラス階層を Fig. 6.1 に示します．衝突判定形状はすべて CDSShape から派生します．アルゴリズムの性質上，形状はすべて凸形状でなければなりません．

### 6.2 形状の作成

衝突判定形状は次の手順で作成・登録します．

1. 形状を作成する
2. 剛体へ形状を追加する
3. 形状の位置を設定する

以下に順を追って説明します．

まず形状を作成するには次のようにします．

```
// given PHSdkIf* phSdk  
  
CDBoxDesc desc;  
desc.boxsize = Vec3d(1.0, 1.0, 1.0);
```

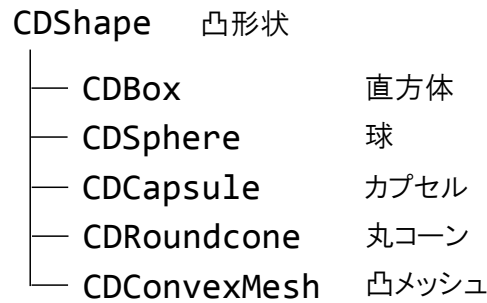


Fig. 6.1 Class hierarchy of Collision module

```
CDBoxIf* box = phSdk->CreateShape(desc)->Cast();
```

衝突判定形状のオブジェクトは Physics モジュールが管理します。このため、形状を作成するには PHSdk クラスの CreateShape 関数を使います。PHSdk については 7 章を参照してください。形状を作成するには、まず種類に応じたディスクリプタを作成し、寸法などのパラメータを設定します。この例では直方体クラス CDBox のディスクリプタを作成して一辺が 1.0 の立方体を作成します。ディスクリプタを指定して CreateShape を呼び出すと、対応する種類の形状が作成され、そのインタフェースが返されます。ただし戻り値は形状の基底クラスである CDSShape のインタフェースですので、派生クラス（ここでは CDBox）のインタフェースを得るには上のように Cast 関数で動的キャストする必要があります。

形状を作成したら、次にその形状を与えたい剛体に登録します。

```
// given PHSolidIf* solid

solid->AddShape(box);          // first box
```

剛体クラス PHSolid については 7 章を参照してください。ここで重要なことは、一度作成した形状は 1 つの剛体にいくつでも登録でき、また異なる複数の剛体にも登録できるということです。つまり、同じ形状を複数の剛体間で共有することで、形状の作成コストやメモリ消費を抑えることができます。

AddShape 関数で登録した直後の形状は、剛体のローカル座標系の原点に位置しています。これを変更したい場合は SetShapePose 関数を使います。

```
solid->AddShape(box);          // second box
solid->AddShape(box);          // third box

// move first shape 1.0 in x-direction
```

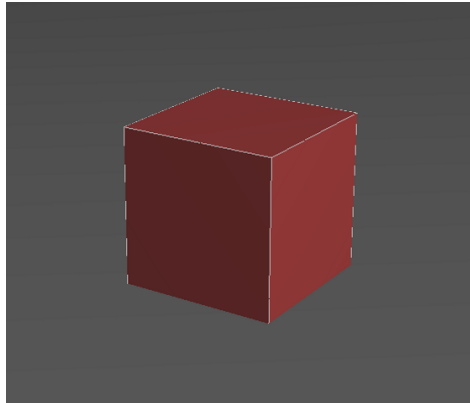


Fig. 6.2 Box geometry

```
solid->SetShapePose(0, Posed(Vec3d(1.0, 0.0, 0.0), Quaterniond()));

// rotate second shape 30 degrees along y-axis
solid->SetShapePose(1, Posed(Vec3d(),
                             Quaterniond::Rot(Rad(30.0), 'y')));
```

SetShapePose の第 1 引数は操作する形状の番号です．最初に AddShape した形状の番号が 0 で，AddShape するたびに 1 増加します．形状の位置や向きは剛体のローカル座標系で指定します．また，形状の位置・向きを取得するには GetShapePose 関数を使います．

以下では Springhead でサポートされている形状を種類別に解説します．

## 直方体

直方体 (Fig. 6.2) のクラスは CDBox です．

CDBoxDesc		
Vec3f	boxsize	各辺の長さ
CDBoxIf		
Vec3f	GetBoxSize()	
void	SetBoxSize(Vec3f)	

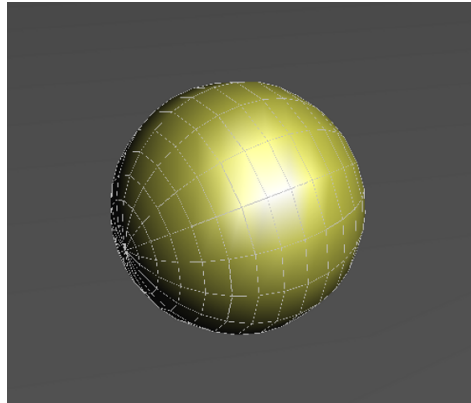


Fig. 6.3 Sphere geometry

## 球

球 (Fig. 6.3) のクラスは `CDSphere` です .

---

`CDSphereDesc`

<code>float</code>	<code>radius</code>	半径
--------------------	---------------------	----

---

`CDSphereIf`

`float` `GetRadius()`

`void` `SetRadius(float)`

## カプセル

カプセル (Fig. 6.4) のクラスは `CDCapsule` です . カプセルは円柱の両端に半球がついた形をしています .

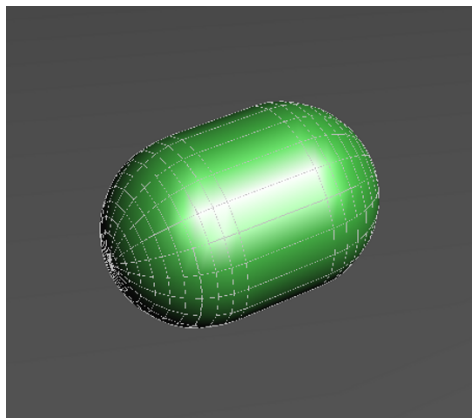


Fig. 6.4 Capsule geometry

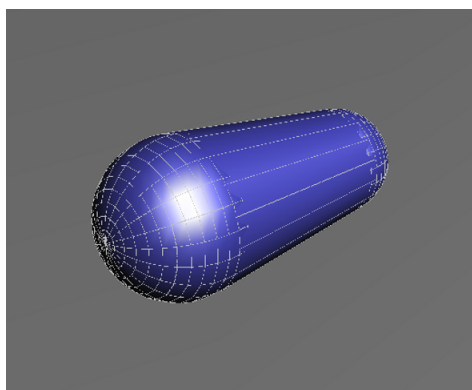


Fig. 6.5 Round cone geometry

---

#### CDCapsuleDesc

float	radius	半球の半径
float	length	円柱の長さ

---

#### CDCapsuleIf

```
float GetRadius()
void SetRadius(float)
float GetLength()
void SetLength(float)
```

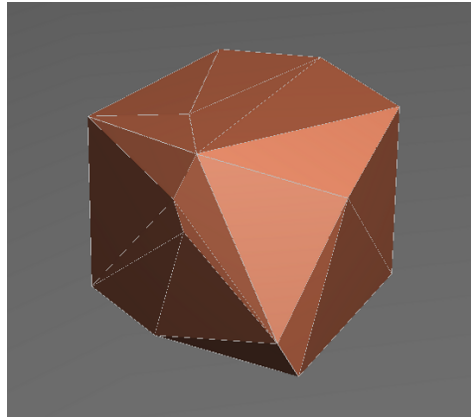


Fig. 6.6 Convex mesh geometry

## 丸コーン

丸コーン (Fig. 6.5) のクラスは `CDRoundCone` です。丸コーンはカプセルの両端の半径が非対称になったものです。

CDRoundConeDesc		
Vec2f	radius	各半球の半径
float	length	半球間の距離
CDRoundConeIf		
Vec2f	GetRadius()	
void	SetRadius(Vec2f)	
float	GetLength()	
void	SetLength(float)	
void	SetWidth(Vec2f)	

`SetWidth` 関数は、丸コーンの全長を保存したまま半径を変更します。

## 凸メッシュ

凸メッシュ (Fig. 6.6) のクラスは `CDConvexMesh` です。凸メッシュとは凹みや穴を持たない多面体です。頂点座標を指定することで自由な形を作成することができます。



---

CDConvexMeshDesc

---

vector&lt;Vec3f&gt; vertices 頂点座標の配列

---

CDConvexMeshIf

---

Vec3f\* GetVertices() 頂点配列の先頭アドレス

int NVertex() 頂点数

CDFaceIf\* GetFace(int i)  $i$  番目の面

int NFace() 面数

凸メッシュが作成される際, CDConvexMeshDesc::vertices に格納された頂点を内包する最小の凸多面体 (凸包) が作成されます. 多面体の面を表す CDFace のインタフェースを以下に示します.

---

CDFaceIf

---

int\* GetIndices() 頂点インデックス配列の先頭アドレス

int NIndex() 面の頂点数

NIndex は面を構成する頂点の数を返します (通常 3 か 4 です). 面は頂点配列を直接保有せず, インデックス配列として間接的に頂点を参照します. したがって, 面の頂点座標を得るには

```
// given CDConvexMeshIf* mesh
CDFaceIf* face = mesh->GetFace(0);           // get 0-th face
int* idx = face->GetIndices();
Vec3f v = mesh->GetVertices()[idx[0]];       // get 0-th vertex
```

とします.

## 6.3 物性の指定

形状には摩擦係数や跳ね返り係数などの物性を指定することができます. 形状の基本クラスである CDShape のディスクリプタ CDShapeDesc は PHMaterial 型の変数 material を持っています.

## PHMaterial

---

float	density	密度
float	mu0	静止摩擦係数
float	mu	動摩擦係数
float	e	跳ね返り係数
float	reflexSpring	跳ね返りバネ係数 (ペナルティ法)
float	reflexDamper	跳ね返りダンパ係数 (ペナルティ法)
float	frictionSpring	摩擦バネ係数 (ペナルティ法)
float	frictionDamper	摩擦ダンパ係数 (ペナルティ法)

形状作成後に物性を指定するには CDShapeIf の関数を使います。

## CDShapeIf

---

```

void SetDensity(float)
float GetDensity()
void SetStaticFriction(float)
float GetStaticFriction()
void SetDynamicFriction(float)
float GetDynamicFriction()
void SetElasticity(float)
float GetElasticity()
void SetReflexSpring(float)
float GetReflexSpring()
void SetReflexDamper(float)
float GetReflexDamper()
void SetFrictionSpring(float)
float GetFrictionSpring()
void SetFrictionDamper(float)
float GetFrictionDamper()

```

## 6.4 幾何情報の計算

形状に関する幾何情報を計算する関数を紹介します。

---

CDShapeIf

---

float CalcVolume()	体積を計算
Vec3f CalcCenterOfMass()	質量中心を計算
Matrix3f CalcMomentOfInertia()	慣性行列を計算

CalcVolume は形状の体積を計算します。体積に密度 (GetDensity で取得) を掛ければ質量が得られます。CalcCenterOfMass 関数は、形状のローカル座標系で表された質量中心の座標を計算します。CalcMomentOfInertia 関数は、形状のローカル座標系で表された質量中心に関する慣性行列を計算します。ただし、密度を 1 とした場合の値が返されますので、実際の慣性行列を得るには密度を掛ける必要があります。



## 第 7 章

# Physics

### 7.1 概要

Physics モジュールは物理シミュレーション機能を提供します。主にサポートされているのは、マルチボディダイナミクスと呼ばれる剛体と関節などの拘束からなる動力学シミュレーションです。今のところソフトボディや流体、パーティクルなどの機能はサポートされていません。

### 7.2 Physics SDK

Physics モジュールのすべてのオブジェクトは SDK クラス PHSdk によって管理されます。PHSdk クラスは、プログラムの実行を通してただ 1 つのオブジェクトが存在するシングルトンクラスです。PHSdk オブジェクトを作成するには以下のようにします。

```
PHSdkIf* phSdk = PHSdkIf::CreateSdk();
```

通常この操作はプログラムの初期化時に一度だけ実行します。また、Framework モジュールを使用する場合はユーザが直接 PHSdk を作成する必要はありません。

PHSdk の機能はシーンと形状の管理です。シーンに関する機能は次節で説明します。また、形状に関する機能は以下の通りです。

PHSdkIf

---

CDShapeIf* CreateShape(const CDShapeDesc&)	形状を作成
CDShapeIf* GetShape(int)	形状を取得
int NShape()	形状の数

異なるシーン間で形状を共有できるように、形状管理はシーンではなく PHSdk の機能になっています。詳しくは6章を参照してください。

## 7.3 シーン

シーンは物理シミュレーションを行う環境を表します。複数のシーンを作成できますが、シーン同士は互いに独立しており、ユーザが直接橋渡し処理をしない限りは影響を及ぼしあうことはありません。シーンクラスは PHScene で、PHScene オブジェクトは PHSdk により管理されます。

### PHSdkIf

---

PHSceneIf* CreateScene(const PHSceneDesc& desc)	シーンを作成
int NScene()	シーンの数
PHSceneIf* GetScene(int i)	シーンを取得
void MergeScene(PHSceneIf* scene0, PHSceneIf* scene1)	シーンを統合

シーンを作成するには以下のようにします。

```
PHSceneIf* phScene = phSdk->CreateScene();
```

引数にディスクリプタを指定することもできます。MergeScene は、scene1 が保有するオブジェクトをすべて scene0 に移動した後に scene1 を削除します。

シーンは剛体や関節などの様々な構成要素の管理を行うほか、物理シミュレーションに関する設定を行う機能を提供します。各構成要素の作成についてはそれぞれの節で説明しますので、以下ではシミュレーション設定機能について述べます。

### PHSceneDesc

---

double	timeStep	時間ステップ幅
unsigned	count	シミュレーションしたステップ数
Vec3d	gravity	重力加速度
double	airResistanceRate	空気抵抗係数
int	numIteration	LCP の反復回数

---

PHSceneIf

---

```
double GetTimeStep()
void SetTimeStep(double)
unsigned GetCount()
void SetCount(unsigned)
void SetGravity(const Vec3d&)
Vec3d GetGravity()
void SetAirResistanceRate(double)
double GetAirResistanceRate()
int GetNumIteration()
void SetNumIteration()
```

timeStep は一度のシミュレーションステップで進める時間幅です．小さいほどシミュレーションの精度は上がりますが，同じ時間シミュレーションを進めるのにかかる計算コストは増大します．

count はシーン作成後にシミュレーションした累積ステップ数です．count と timeStep の積が経過時間を表します．

gravity は重力加速度ベクトルです．

airResistanceRate は，シミュレーションの安定性を向上するために毎ステップに各剛体の速度に掛けられる係数です．例えば airResistanceRate が 0.95 であればステップごとに速度が 95% になります．このように強制的に減速をかけることで，精度を犠牲に安定性を得ることができます．

numIteration は，拘束力を計算するために内部で実行されるアルゴリズムの反復回数です．一般に，反復回数に関して指数関数的に拘束力の精度が向上し，計算コストは比例的に増大します．

## シミュレーションの実行

シミュレーションを 1 ステップ進めるには Step 関数を呼びます．

---

PHSceneIf

---

```
void Step() シミュレーションを 1 ステップ進める
```

Step を実行すると，おおまかに述べて内部で次の処理が行われます．

- 衝突判定と接触拘束の生成
- 拘束力の計算

- 剛体の速度および位置の更新

## 7.4 剛体

剛体は物理シミュレーションの基本要素です．剛体のクラスは `PHSolid` です．まず剛体を作成・管理するための `PHScene` の関数を示します．

`PHSceneIf`

---

<code>PHSolidIf* CreateSolid(const PHSolidDesc&amp;)</code>	剛体を作成する
<code>int NSolids()</code>	剛体の数
<code>PHSolidIf** GetSolids()</code>	剛体配列の先頭アドレス

剛体を作成するには

```
PHSolidIf* solid = phScene->CreateSolid();
```

とします．ディスクリプタを指定して作成することもできます．また，既存の剛体を取得するには

```
PHSolidIf* solid = phScene->GetSolids()[0];    // get 0-th solid
```

とします．

つぎに剛体自身の機能を説明します．

### 物性

`PHSolidDesc`

---

<code>double</code>	<code>mass</code>	質量
<code>Matrix3d</code>	<code>inertia</code>	慣性行列
<code>Vec3d</code>	<code>center</code>	質量中心
<code>bool</code>	<code>dynamical</code>	物理法則にしたがうか



## PHSolidIf

---

```

double GetMass()
double GetMassInv()
void SetMass(double)
Vec3d GetCenterOfMass()
void SetCenterOfMass(const Vec3d&)
Matrix3d GetInertia()
Matrix3d GetInertiaInv()
void SetInertia(const Matrix3d&)
void CompInertia()
void SetDynamical(bool)
bool IsDynamical()

```

GetMassInv と GetInertiaInv はそれぞれ質量の逆数と慣性行列の逆行列を返します。

CompInertia は、その剛体を持つ形状とそれらの密度をもとに剛体の質量、質量中心と慣性行列を計算し、設定します。

dynamical は、その剛体が物理法則に従うかどうかを指定するフラグです。もし dynamical が true の場合、その剛体に加わる力が計算され、ニュートンの運動法則にしたがって剛体の速度が変化します。一方、dynamical が false の場合は外力による影響を受けず、設定された速度で等速運動します。これはちょうど の質量をもつ場合と同じです。

## 状態

## PHSolidDesc

---

Vec3d	velocity	速度
Vec3d	angVelocity	角速度
Posed	pose	位置と向き

---

PHSolidIf

---

```
Vec3d GetVelocity()
void SetVelocity(const Vec3d&)
Vec3d GetAngularVelocity()
void SetAngularVelocity(const Vec3d&)
Posed GetPose()
void SetPose(const Posed&)
Vec3d GetFramePosition()
void SetFramePosition(const Vec3d&)
Vec3d GetCenterPosition()
void SetCenterPosition(const Vec3d&)
Quaterniond GetOrientation()
void SetOrientation(const Quaterniond&)
```

velocity, angVelocity, pose はそれぞれグローバル座標系に関する剛体の速度，角速度，位置および向きを表します．[Get|Set]FramePosition はグローバル座標系に関する剛体の位置を取得/設定します．これに対して [Get|Set]CenterPosition は剛体の質量中心の位置を取得/設定します．偏心している剛体はローカル座標原点と質量中心が一致しないことに注意してください．[Get|Set]Orientation はグローバル座標系に関する剛体の向きを取得/設定します．

## 力の印加と取得

剛体に加わる力には

- ユーザが設定する外力
- 重力
- 関節や接触から加わる拘束力

の 3 種類があり，それぞれについて並進力とトルクがあります．ここで，重力は重力加速度と剛体の質量より決まり，拘束力は拘束条件を満たすように内部で自動的に計算されます．以下ではユーザが剛体に加える外力を設定・取得する方法を示します．

```
PHSolidIf
void AddForce(Vec3d)
void AddTorque(Vec3d)
void AddForce(Vec3d, Vec3d)
Vec3d GetForce()
Vec3d GetTorque()
```

並進力を加えるには `AddForce` を使います .

```
solid->AddForce(Vec3d(0.0, -1.0, 0.0));
```

とすると剛体の質量中心に並進力  $(0, -1, 0)$  が加わります . ただし力はグローバル座標系で表現されます . 一方

```
solid->AddTorque(Vec3d(1.0, 0.0, 0.0));
```

とすると剛体の質量中心に関してモーメント  $(1, 0, 0)$  が加わります . 作用点を任意に指定するには

```
solid->AddForce(Vec3d(0.0, -1.0, 0.0), Vec3d(0.0, 0.0, 1.0));
```

とします . この場合は並進力  $(0, -1, 0)$  が作用点  $(0, 0, 1)$  に加わります . ここで作用点の位置は剛体のローカル座標ではなくグローバル座標で表現されることに注意してください . `AddForce` や `AddTorque` は複数回呼ぶと , それぞれで指定した外力の合力が最終的に剛体に加わる外力となります .

外力を取得するには `GetForce` , `GetTorque` を使います . ただし , これらの関数で取得できるのは直前のシミュレーションステップで剛体に作用した外力です . したがって直前のシミュレーションステップ後に `AddForce` した力は取得できません .

## 7.5 関節

拘束とは剛体と剛体の間に作用してその相対的運動に制約を加える要素です . 拘束のクラス階層を Fig. 7.1 に示します . まず拘束は関節と接触に分かれます . 関節はユーザが作成しますが , 接触は衝突判定結果にもとづいて自動的に生成・削除されます . 関節はさらにいくつかの種類に分けられます .

細かな説明は後回しにして , まずは関節の作成方法から見ていきます .

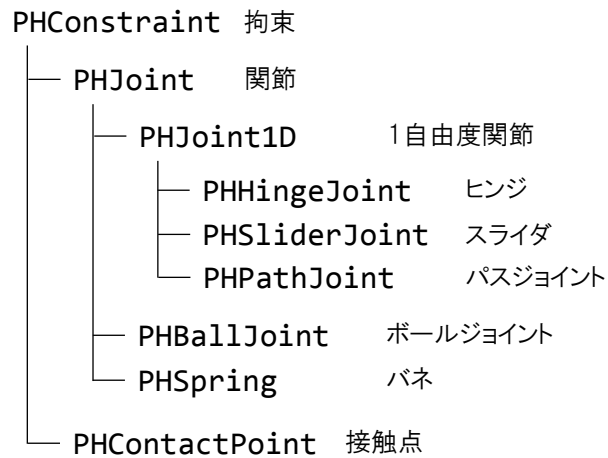


Fig. 7.1 Constraint class hierarchy

## 関節の作成

以下ではもっとも使用頻度の高いヒンジの作成を例にとって関節の作成方法を説明します．ヒンジを作成するには次のようにします．

```

PHSolidIf* solid0 = phScene->GetSolids()[0];
PHSolidIf* solid1 = phScene->GetSolids()[1];

PHHingeJointDesc desc;
desc.poseSocket.Pos() = Vec3d( 1.0, 0.0, 0.0);
desc.posePlug.Pos()   = Vec3d(-1.0, 0.0, 0.0);
PHHingeJointIf* joint
    = phScene->CreateJoint(solid0, solid1, desc)->Cast();
  
```

作成したい関節の種類に応じたディスクリプタを作成し，これを PHScene の CreateJoint 関数に渡して関節を作成します．このとき，ディスクリプタとともに連結したい剛体のインタフェースも渡します．CreateJoint は PHJointIf\* を返しますので，作成した関節のインタフェースを得るには Cast で動的キャストします．

関節に関する PHScene の関数を以下に示します．

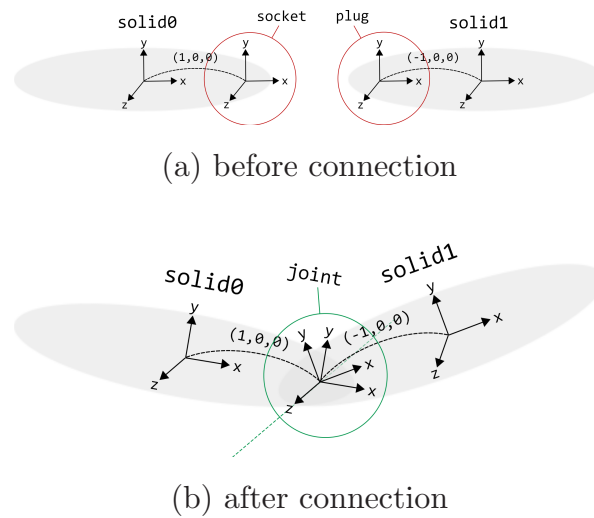


Fig. 7.2 Socket and plug

## PHSceneIf

```
PHJointIf* CreateJoint(PHSolidIf*, PHSolidIf*, const PHJointDesc&)
int NJoint()
PHJointIf* GetJoint(int i)
```

NJoint はシーン中の関節の個数を返します。GetJoint は  $i$  番目の関節を取得します。

## ソケットとプラグ

さて、上の例でディスクリプタに値を設定している箇所に注目してください。この部分で関節の取り付け位置を指定しています。Springhead では、ソケットとプラグと呼ばれるローカル座標系を用いて関節の取り付け位置を表現します。ソケットとプラグとは、その名前から連想するように、連結する剛体に取り付ける金具のようなものです。CreateJoint の第 1 引数の剛体にソケットがつき、第 2 引数の剛体にプラグがつきます。ソケットとプラグがそれぞれの剛体のどの位置に取り付けられるかを指定するのがディスクリプタの poseSocket と posePlug です。上の例ではソケットの位置が  $(1, 0, 0)$ 、プラグの位置が  $(-1, 0, 0)$  でした (Fig. 7.2(a))。この場合は Fig. 7.2(b) のように剛体が連結されます。後述するように、ヒンジはソケットとプラグの  $z$  軸を一致させる拘束です。したがって連結された剛体同士はソケットとプラグの  $z$  軸を回転軸として相対的に回転することができます。

ソケットとプラグに関するディスクリプタとインタフェースを紹介します。

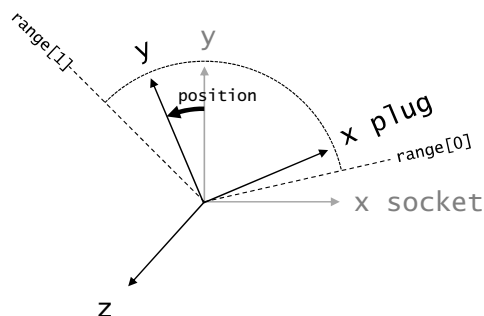


Fig. 7.3 Hinge joint

---

PHConstraintDesc

---

Posed	poseSocket	ソケットの位置と向き
Posed	posePlug	プラグの位置と向き

---

PHConstraintIf

---

PHSolidIf* GetSocketSolid()	ソケット側の剛体を取得
PHSolidIf* GetPlugSolid()	プラグ側の剛体を取得
void GetSocketPose(Posed&)	
void SetSocketPose(const Posed&)	
void GetPlugPose(Posed&)	
void SetPlugPose(const Posed&)	
void GetRelativePose(Posed&)	相対的な位置と向き
void GetRelativeVelocity(Vec3d&, Vec3d&)	相対速度
void GetConstraintForce(Vec3d&, Vec3d&)	拘束力

GetRelativePose はソケット座標系から見たプラグ座標系の相対的な位置と向きを取得します。同様に、GetRelativeVelocity はソケットから見たプラグの相対速度をソケット座標系で取得します。ここで第 1 引数が並進速度、第 2 引数が角速度です。GetConstraintForce はこの拘束が剛体に加えた拘束力を取得します (第 1 引数が並進力、第 2 引数がモーメント)。具体的には、ソケット側剛体に作用した拘束力をソケット座標系で表現したものが得られます。プラグ側剛体には作用反作用の法則によって逆向きの力が作用しますが、これを直接取得する関数は用意されていません。

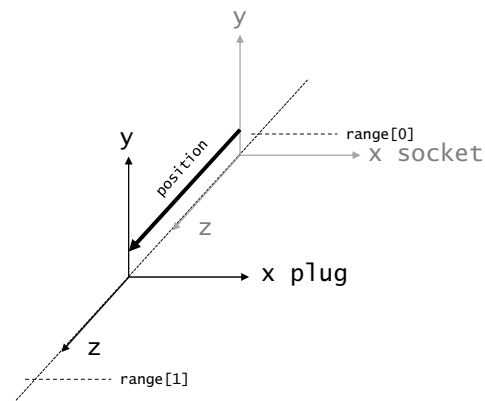


Fig. 7.4 Slider joint

## ヒンジ

ヒンジは 1 軸回転関節です。ヒンジは、Fig. 7.3 に示すようにソケットとプラグの  $z$  軸が一致するように拘束します。このときソケットの  $y$  軸とプラグの  $y$  軸の成す角 ( $x$  軸同士でも同じことですが) が関節変位となります。

## スライダ

スライダは 1 自由度の直動関節です。スライダは、Fig. 7.4 に示すようにソケットとプラグの  $z$  軸が同一直線上に乗り、かつ両者の  $x$  軸、 $y$  軸が同じ向きを向くように拘束します。このときソケットの原点からプラグの原点までが関節変位となります。

## パスジョイント

パスジョイントはソケットとプラグの相対位置関係を 1 パラメータの自由曲線で表現する関節です。

T.B.D.

## ボールジョイント

ボールジョイントは 3 自由度の回転関節です。ボールジョイントはソケットとプラグの原点が一致するように拘束します。

T.B.D.

## バネ

T.B.D.

## 有効化と無効化

---

PHConstraintDesc

bool   bEnabled     有効/無効フラグ

---

PHConstraintIf

void Enable(bool)

bool IsEnabled()

有効な拘束は拘束力を生じます．無効化された拘束は存在しないのと同じ状態になりますが，削除するのとは異なりいつでも再度有効化することができます．作成直後の拘束は有効化されています．

## 可動範囲

まず 1 自由度関節の可動範囲について説明します．

---

PHJoint1DDesc

double   lower	可動範囲下限
double   upper	可動範囲上限
double   rangeSpring	可動範囲制限用バネ係数
double   rangeDamper	可動範囲制限用ダンパ係数

---

PHJoint1DIf

---

```
void SetRange(double lower, double upper)
void GetRange(double& lower, double& upper)
void SetRangeSpring(double)
double GetRangeSpring()
void SetRangeDamper(double)
double GetRangeDamper()
bool IsLimit()
```



lower と upper はそれぞれ可動範囲の下限と上限を表します。lower < upper が満たされているときに限り可動範囲制約が有効となります。

関節の変位が可動範囲限界に到達したとき、範囲を超過しないように可動範囲制約の拘束力が作用します。このとき、関節変位を範囲内に押し戻す力はバネ・ダンパモデルで計算されます。このバネ係数とダンパ係数はそれぞれ rangeSpring, rangeDamper で指定します。

ボールジョイントの可動範囲：T.B.D.

## 関節制御

### 1 自由度関節の関節制御について説明します。

#### PHJoint1DDesc

double	spring	可動範囲下限
double	damper	可動範囲上限
double	targetPosition	可動範囲制限用バネ係数
double	targetVelocity	可動範囲制限用ダンパ係数
double	offsetForce	
double	fMax	

#### PHJoint1DIf

double	GetPosition()	関節変位を取得
double	GetVelocity()	関節速度を取得
void	SetSpring(double)	
double	GetSpring()	
void	SetDamper(double)	
double	GetDamper()	
void	SetTargetPosition(double)	
double	GetTargetPosition()	
void	SetTargetVelocity(double)	
double	GetTargetVelocity()	
void	SetOffsetForce(double)	
double	GetOffsetForce()	
void	SetTorqueMax(double)	最大関節トルクを設定
double	GetTorqueMax()	最大関節トルクを取得

関節を駆動する力  $f$  は次式で与えられます．

$$f = K(p_0 - p) + D(v_0 - v) + f_0$$

ここで  $p, v$  はそれぞれ関節変位と関節速度で `GetPosition`, `GetVelocity` で取得できます．その他の記号とディスクリプタ変数との対応は以下の通りです．

$K$	spring
$D$	damper
$p_0$	targetPosition
$v_0$	targetVelocity
$f_0$	offsetForce

上の式はバネ・ダンパモデルと PD 制御則の二通りの解釈ができます．前者としてとらえるなら  $K$  はバネ係数， $D$  はダンパ係数， $p_0$  はバネの自然長， $v_0$  は基準速度となります．後者としてとらえる場合は  $K$  は P ゲイン， $D$  は D ゲイン， $p_0$  は目標変位， $v_0$  は目標速度となります．また， $f_0$  は関節トルクのオフセット項です．上の式で得られた関節トルクは最後に  $\pm f_{\text{Max}}$  の範囲に収まるようにクランプされます．

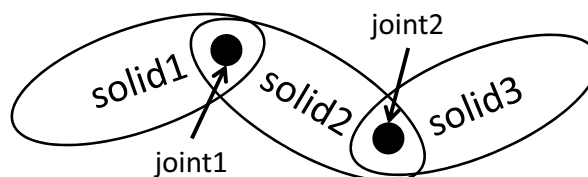
ボールジョイントの関節制御：T.B.D.

## 7.6 関節系の逆運動学

逆運動学 (IK) は，剛体関節系において剛体が目標位置に到達するよう関節を制御する機能です．

Springhead では，関節系のヤコビアンを用いた IK 機能が使用可能です．物理シミュレーションの 1 ステップごとに関節系のヤコビアンを計算し，それに基づいて剛体を目標位置・姿勢に近づけるような各関節の角速度を計算します．シミュレーションを続けることで，最終的に剛体が目標位置・姿勢となった状態が得られます．

Springhead 上の剛体関節系に対して IK を使用するには，少々下準備が必要です．次のように 3 つの剛体が直線状につながった関節系を例にとって解説します．



IK を使用するには，まず IK に用いるための関節を「アクチュエータ」として登録する必要があります．

```
// given PHSceneIf* phScene
// given PHSolidIf* solid1, solid2, solid3
// given PHHingeJointIf* joint1 (solid1 <-> solid2)
// given PHHingeJointIf* joint2 (solid2 <-> solid3)

PHIKHingeActuatorDesc descIKActuator;

PHIKHingeActuatorIf* ikActuator1
    = phScene->CreateIKActuator(descIKActuator);
ikActuator1.AddChildObject(joint1);

PHIKHingeActuatorIf* ikActuator2
    = phScene->CreateIKActuator(descIKActuator);
ikActuator1.AddChildObject(joint2);
```

PHIKHingeActuatorIf は PHHingeJointIf に対応するアクチュエータクラスです。

次に、関節系の親子関係を登録します。親アクチュエータに、子アクチュエータを登録します。

```
ikActuator1.AddChildObject(ikActuator2);
```

また、IK を用いて到達させる先端の剛体を「エンドエフェクタ」として登録する必要があります。

```
PHIKEndEffectorDesc descEndEffector;

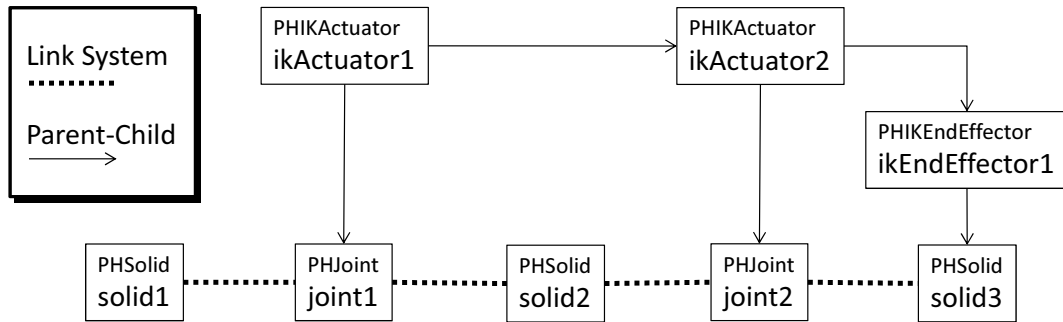
PHIKEndEffectorIf* ikEndEffector1
    = phScene->CreateIKEndEffector(descEndEffector);
ikEndEffector1.AddChildObject(solid3);
```

最後に、剛体関節系の親子関係において、エンドエフェクタの直接の親にあたるアクチュエータに対し、エンドエフェクタを登録します。

```
ikActuator2.AddChildObject(ikEndEffector1);
```

この例では solid1 -(joint1)-> solid2 -(joint2)-> solid3 のように関節が接続されていますから、関節系の末端である solid3 をエンドエフェクタにした場合、直接の親にあたるアクチュエータは joint2 に対応するアクチュエータ、すなわち ikActuator2 ということになります。

ここまでの作業で、生成されたオブジェクトの関係は以下のようになっているはずです。



これで下準備は終わりです。

目標位置をセットし，IK エンジンを有効にすると IK が動き始めます。

```

// solid3 goes to (2, 5, 0)
ikEndEffector1->SetTargetPosition(Vec3d(2, 5, 0));

phScene->GetIKEngine()->Enable(true);

...
phScene->Step(); // IK is calculated in physics step
...

```

## IK エンジン

IK の計算は，PHScene が持つ IK エンジン (PHIKEngine) によって実現されています。IK エンジンデフォルトでは無効となっています。

```
phScene->GetIKEngine()->Enable(true);
```

を実行することで有効となります。GetIKEngine() は，PHScene が持つ IK エンジンを取得する API です。

Springhead における IK の計算原理は，関節系のヤコビ行列 (ヤコビアン) に基づきます。全アクチュエータの関節角度に微小変化量  $\Delta\theta$  を与えた時の，全エンドエフェクタの位置の微小変化量  $\Delta r$  は，関節系のヤコビアン  $J$  を用いて

$$\Delta r = J \Delta \theta$$

と表されます。毎ステップごとに関節系ヤコビアン  $J$  および目標位置に向かう微小変位  $\Delta r$  を計算し，上記の線形連立方程式を解くことで各関節に与える角速度を求めます。

線形連立方程式の求解にはガウス＝ザイデル法による繰り返し解法を用いています。そのため 1 ステップあたりの繰り返し計算の回数によって計算速度と計算精度のトレードオフがあります。繰り返しの回数は，

```
// 20 iteration per 1 physics step  
phScene->GetIKEngine()->SetNumIter(20);
```

のようにして設定することができます。

## Reference

### PHSceneIf クラス

PHIKEngineIf\* GetIKEngine()

IK エンジンを取得します。

### PHIKEngineIf クラス

Enable(bool b)

IK エンジンの有効・無効を切り替えます。引数が true ならば有効化し, false ならば無効化します。

SetNumIter(int n)

IK の繰り返し計算回数を 1 ステップあたり n 回にセットします。

## アクチュエータ

Springhead では, IK に使用する各関節をアクチュエータと呼びます。IK は, アクチュエータを駆動させて剛体を目標位置に到達させます。

<オブジェクト関係図>

IK エンジンはアクチュエータを複数保持し, 各アクチュエータが各関節を保持します。アクチュエータオブジェクト一つにつき, 関節が一つ対応します。アクチュエータオブジェクトの具体的な役割は, 関節の状態を IK エンジンに伝え, IK の計算のうち関節ヤコビアンなどの計算など関節ごとに行う部分を実行し, IK の計算結果に従って関節を動かす事です。

### アクチュエータクラスの種類と作成

本稿執筆時点では, IK 用アクチュエータとして使用できるのはヒンジとボールジョイントのみです。それぞれに対応したアクチュエータクラスがあります。

- PHIKHingeActuator は PHHingeJoint に対するアクチュエータです。ヒンジジョイントの 1 自由度を駆動に用います。
- PHIKBallActuator は PHBallJoint に対するアクチュエータです。ボールジョイントは 3 自由度の関節ですが, 後述するエンドエフェクタの姿勢制御を行わない(エンドエフェクタの位置のみを制御する)場合は, エンドエフェクタの位置を変化

させることのできる 2 自由度のみを駆動に用います (使用する 2 自由度の軸は 1 ステップごとに更新されます) .

```
// given PHSceneIf* phScene

PHIKHingeActuatorDesc descIKActuator;
PHIKHingeActuatorIf* ikActuator
    = phScene->CreateIKActuator(descActuator);
```

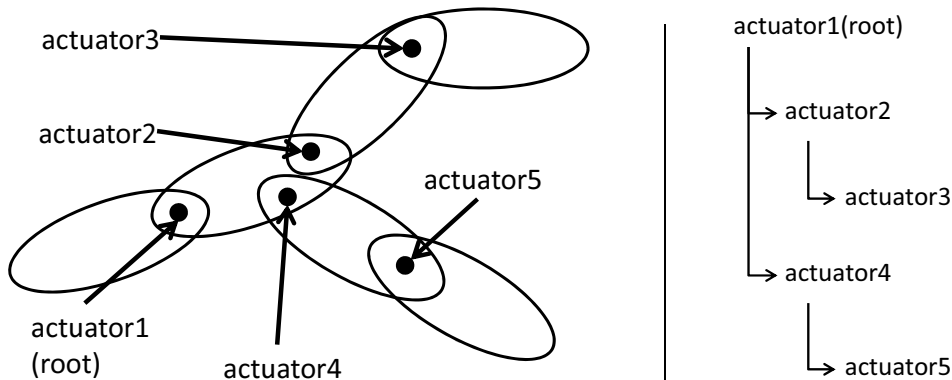
アクチュエータを作成するには, PHSceneIf の CreateIKActuator 関数を用います. 引数はアクチュエータのディスクリプタです. PHIKHingeActuatorDesc 型のディスクリプタを渡すとヒンジ用のアクチュエータが作成され, PHIKBallActuatorDesc 型のディスクリプタを渡すとボールジョイント用のアクチュエータが作成されます.

作成された時点では, アクチュエータは関節と対応付けがされていません. アクチュエータの子要素に関節を登録することで対応付けが行われます.

```
// given PHHingeJointIf* joint
ikActuator->AddChildObject(joint);
```

#### アクチュエータの親子関係の登録

次のように二股に分岐したリンクを例にとります:



計算上, IK で駆動する関節系は木構造でなければなりません. Springhead では, アクチュエータの親子関係を作ることによって関節の木構造を設定します.

```
// given PHIKActuator ikActuator1, ikActuator2
ikActuator1->AddChildObject(ikActuator2);
```

AddChildObject を呼び出すと, アクチュエータに対し「子要素」となるアクチュエータを登録することができます. これを全てのアクチュエータに対して行うことでアクチュエータの木構造が設定されます. このときアクチュエータの親子関係は, 前出の図の右側のようにになります.

### 関節のダンパ係数と IK

関節の運動は IK 機能によって計算された目標関節角速度を関節に `SetTargetVelocity` することで実現します。目標速度に関する関節の振る舞いは、関節の `damper` パラメータによって変化します。一般に `damper` が大きいほど関節は固くなり、外乱の影響を受けづらくなります。この性質はそのまま IK の振る舞いにも受け継がれます。

### 重み付き IK

通常、IK は全ての関節を可能な限り均等に使用して目標を達成するように計算されます。一方、キャラクタの動作に用いる場合などで、手先を優先的に動かし胴体はあまり動かせない、といった重み付けが要求される場面があります。

Springhead の IK には、このような重み付けを設定することができます。

```
// given PHIKActuator ikActuator1, ikActuator2
ikActuator1->SetBias(2.0);
ikActuator2->SetBias(1.0);
```

`SetBias` は、指定した関節をあまり動かさないように設定する関数です。Bias には 1.0 以上の値を設定します。大きな値を設定した関節ほど、IK による動作は小さくなります。デフォルトではどのアクチュエータも 1.0 となっており、全関節が均等に使用されます。

T.B.D.

### エンドエフェクタ

剛体・関節系を構成する剛体の一部を、「エンドエフェクタ」に指定することができます。エンドエフェクタには目標位置・姿勢を指示することができます。IK エンジンには、エンドエフェクタ剛体が指定された目標位置・姿勢を達成するようアクチュエータを制御します。

### エンドエフェクタの作成

エンドエフェクタは `PHSceneIf` の `CreateIKEndEffector` を用いて作成します。引数には `PHIKEndEffectorDesc` を渡します。

```
// given PHSceneIf* phScene

PHIKEndEffectorDesc descEndEffector;

PHIKEndEffectorIf* ikEndEffector
    = phScene->CreateIKEndEffector(descEndEffector);
```

アクチュエータ同様，エンドエフェクタも作成時点では剛体との対応を持ちません．  
AddChildObject により剛体の子要素として登録する必要があります．

```
// given PHSolidIf* solid
ikEndEffector.AddChildObject(solid);
```

次に，エンドエフェクタ剛体を親剛体に連結しているアクチュエータに対し，エンドエフェクタを子要素として登録します．

```
// given PHIKActuatorIf* ikActuatorParent
ikActuatorParent.AddChildObject(ikEndEffector);
```

こうすることでエンドエフェクタはアクチュエータ木構造の葉ノードとなり，IK の計算に使用できるようになります．

なお，エンドエフェクタは一つの関節系に対して複数作成することができます．この場合，IK は複数のエンドエフェクタが可能な限り同時に目標位置・姿勢を達成できるようアクチュエータを制御します．また，エンドエフェクタは関節系の先端剛体に限りません．

### 目標位置の設定

エンドエフェクタの目標位置は SetTargetPosition によって指定します．

```
// solid3 goes to (2, 5, 0)
ikEndEffector->SetTargetPosition(Vec3d(2, 5, 0));
```

エンドエフェクタに目標姿勢を指示し，エンドエフェクタが特定の姿勢をとるように関節系を動作させることもできます．

```
ikEndEffector->SetTargetOrientation( Quaterniond::Rot('x', rad(30)) );
ikEndEffector->EnableOrientationControl(true);
```

目標姿勢は Quaterniond で設定します．姿勢制御はデフォルトでは無効になっており，使用するには EnableOrientationControl を呼んで有効化する必要があります．

EnablePositionControl および EnableOrientationControl を用いると，位置制御・姿勢制御の両方を個別に有効・無効化することができます．

```
// 位置制御あり，姿勢制御なし（デフォルト）
ikEndEffector->EnablePositionControl(true);
ikEndEffector->EnableOrientationControl(false);
```

```
// 位置制御なし，姿勢制御あり
ikEndEffector->EnablePositionControl(false);
ikEndEffector->EnableOrientationControl(true);
```



```
// 位置制御あり, 姿勢制御あり  
ikEndEffector->EnablePositionControl(true);  
ikEndEffector->EnableOrientationControl(true);
```

## 7.7 接触

T.B.D.

## 7.8 関節座標系シミュレーション

T.B.D.

## 7.9 ギア

T.B.D.



## 第 8 章

# Graphics

### 8.1 概要

Graphics は 3D シーンの描画機能を提供するモジュールです。

### 8.2 Graphics SDK

Graphics モジュールのすべてのオブジェクトは SDK クラス `GRSdk` によって管理されます。`GRSdk` クラスは、プログラムの実行を通してただ 1 つのオブジェクトが存在するシングルトンクラスです。`GRSdk` オブジェクトを作成するには以下のようにします。

```
GRSdkIf* grSdk = GRSdkIf::CreateSdk();
```

通常この操作はプログラムの初期化時に一度だけ実行します。また、Framework モジュールを使用する場合はユーザが直接 `GRSdk` を作成する必要はありません。

`GRSdk` には以下の機能があります。

- レンダラの作成
- デバイスの作成
- シーンの管理

レンダラとは処理系に依存しない抽象化された描画機能を提供するクラスです。レンダラのクラスは `GRRender` です。

一方、デバイスは処理系ごとの描画処理の実装を行うクラスです。現在の Springhead では OpenGL による描画のみがサポートされています。OpenGL 用デバイスクラスは `GRDeviceGL` です。

レンダラをデバイスに関する `GRSdk` の関数を以下に示します。

## GRSdkIf

---

GRRenderIf* CreateRender()	レンダラを作成
GRDeviceGLIf* CreateDeviceGL()	OpenGL デバイスを作成

## GRRenderIf

---

void SetDevice(GRDeviceIf*)	デバイスの設定
GRDeviceIf* GetDevice()	デバイスの取得

## 初期化

Graphics モジュールを使用するには以下の初期化処理を必ず実行する必要があります。

```
GRRenderIf* render = grSdk->CreateRender();
GRDeviceIf* device = grSdk->CreateDeviceGL();
device->Init();
render->SetDevice(device);
```

GRRender の SetDevice 関数でデバイスを登録すると、レンダラは実際の描画処理をそのデバイスを用いて行います。将来的に処理系ごとにデバイスを使い分けることを想定し、上の処理はユーザが行うことになっています。Framework モジュールを使用する場合はユーザ自身で上の手続きを行う必要はありません。

## 8.3 シーン

### シーンの作成

Graphics モジュールのシーンは、コンピュータグラフィクスにおけるいわゆるシーングラフと同等のものです。シーンクラスは GRScene です。シーンを作成するには次のようにします。

```
GRSceneIf* grScene = grSdk->CreateScene();
```

GRScene はディスクリプタによる設定項目を持ちません。また、Fig. 8.1 に示すように GRSdk オブジェクトは任意の数のシーンを保持できます。

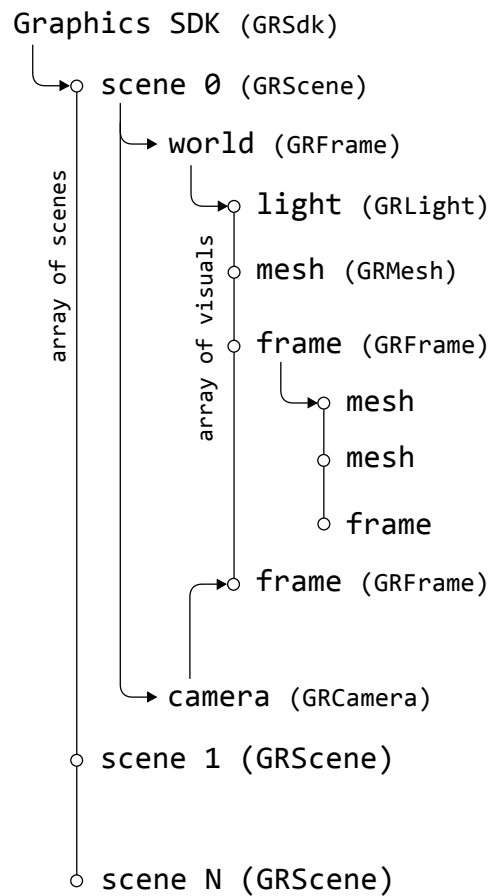


Fig. 8.1 Graphics data structure

シーン作成に関する GRSdk の関数は以下の通りです。

#### GRSdkIf

GRSceneIf* CreateScene()	シーンを作成
GRSceneIf* GetScene(size_t)	シーンを取得
size_t NScene()	シーンの数
void MergeScene(GRSceneIf*, GRSceneIf*)	シーンの統合

#### シーンの機能

シーンを作成したら、次はそのコンテンツであるフレームやメッシュ、カメラやライトなどを作成してシーンに加えていきます。この方法としては完全に手動でシーンを構築する他にも FileIO モジュールを使用してファイルからシーンをロードする方法もあります。以下に GRScene の関数を示します。

## GRSceneIf

---

GRFrameIf* GetWorld()	ワールドフレームの取得
GRCameraIf* GetCamera()	カメラの取得
void SetCamera(const GRCameraDesc&)	カメラの設定
GRVisualIf* CreateVisual(const GRVisualDesc&, GRFrameIf*)	描画アイテムの作成
void Render(GRRenderIf*)	描画

---

Fig. 8.1 に示すように, 1 つのシーンはただ 1 つのワールドフレームを持ち, それを基点として任意の数の描画アイテムがツリー状に連なります。ワールドフレームは `GetWorld` で取得します。

特殊な描画アイテムにカメラがあります。カメラはワールドフレーム以下のツリーとは別に, `GRScene` が保持します (Fig. 8.1)。カメラの設定は `SetCamera` で行います。カメラを取得するには `GetCamera` を使います。また, カメラはシーングラフ中の 1 つのフレームを参照し, これを視点の設定に用います。イメージとしてはカメラが参照先のフレームに取り付けられていると考える方が自然でしょう。参照先のフレームの移動に応じてカメラもシーン中を移動することになります。

## シーンの描画

描画処理はプログラムの描画ハンドラで行います。GLUT を使う場合は `glutDisplayFunc` で登録したコールバック関数がこれにあたり, また Framework モジュールの `FWApp` を使う場合は `Display` 仮想関数がこれにあたります。以下が典型的な描画処理です。

```
render->ClearBuffer();           // clear back buffer
render->BeginScene();             // begin rendering

grScene->Render(render);          // render scene

render->EndScene();               // end rendering
render->SwapBuffers();            // swap buffers
```

`ClearBuffer` は描画バッファを所定の色で塗りつぶします。塗りつぶし色の取得/設定は `GRRender` の `GetClearColor`, `SetClearColor` を使います。

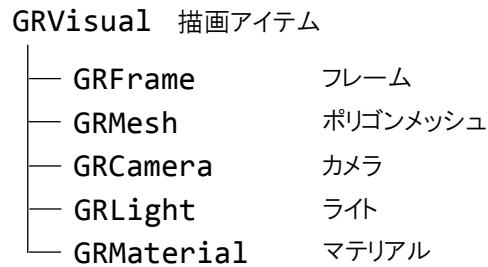


Fig. 8.2 Class hierarchy of visual items

```

render->SetClearColor(Vec4f(1.0f, 0.0f, 0.0f, 1.0f));
render->ClearBuffer();          // clear back buffer in red

```

BeginScene と EndScene はシーンの描画の前後で必ず呼び出します。SwapBuffers はフロントバッファとバックバッファを切り換えることで描画内容を画面上に表示します。

GRScene の Render 関数は、カメラ (GRCamera) の Render とワールドフレーム (GRFrame) の Render を順次呼び出します。まずカメラの描画によって視点と投影変換が設定され、次にワールドフレームの描画によってシーングラフが再帰的に描画されます。

## 8.4 描画アイテム

シーングラフを構成する描画アイテムの基本クラスは GRVisual です。GRVisual から派生するクラスを Fig. 8.2 に示します。描画アイテムには以下の共通の機能があります。

```

GRVisualIf
-----
void Render(GRRenderIf*)
void Rendered(GRRenderIf*)
void Enable(bool)
bool IsEnabled()

```

Render はアイテムの描画を行い、Rendered は描画の後処理を行います。描画処理は描画アイテムの種類ごとに異なります。これについては次節以降で説明します。

Enable 関数は描画処理の有効化/無効化を行います。無効化されたアイテムは描画されません。IsEnabled 関数は有効/無効状態を返します。

描画アイテムを作成するには GRScene の CreateVisual 関数に種類ごとのディスクリプタを指定して呼び出します。

## 8.5 フレーム

フレームは座標変換を定義すると同時に他の描画アイテムのコンテナとしての役割を持ちます。フレームのクラスは `GRFrame` です。次のコードは、フレームを作成してワールドフレームの子として登録します。

```
GRFrameDesc desc;
GRFrameIf* frame =
    grScene->CreateVisual(desc, grScene->GetWorldFrame())->Cast();
```

`CreateVisual` 関数は指定されたディスクリプタに対応する描画アイテムを作成し、指定された親フレームの子として登録します。親フレームを省くとデフォルトでワールドフレームに登録されます。したがって上のコードは `CreateVisual(desc)` としてもかまいません。

`GRFrame` の `Render` 関数は、子描画アイテムの `Render` を順次呼び出します。

### 親子関係

フレーム間の親子関係を管理する関数には次のものがあります。

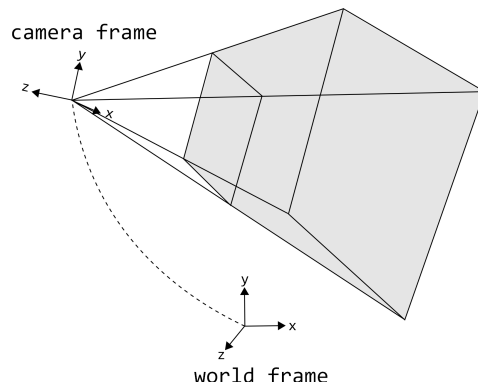
```
GRFrameIf
-----
GRFrameIf* GetParent()
void SetParent(GRFrameIf*)
int NChildren()
GRVisualIf** GetChildren()
```

`GetParent` は親フレームを取得します。`SetParent` はそのフレームの親フレームを変更するために使います。`NChildren` はそのフレームの子である描画アイテムの数を返します。これらにはフレーム以外の描画アイテムも含まれることに注意してください。`GetChildren` は子描画アイテムの配列を取得します。

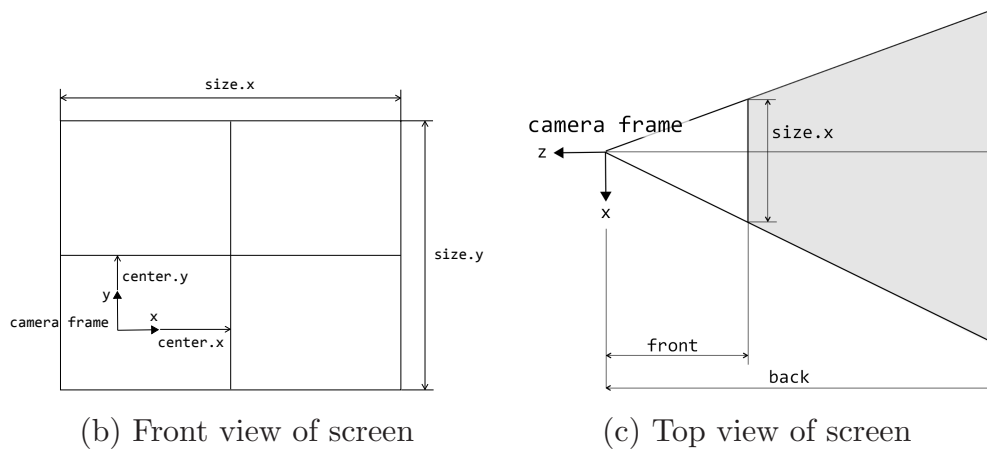
### 座標変換

フレームの座標変換を操作する関数は以下の通りです。





(a) Perspective frustum



(b) Front view of screen

(c) Top view of screen

Fig. 8.3 Camera parameters

### GRFrameIf

---

```

Affinef GetTransform()
Affinef GetWorldTransform()
void SetTransform(const Affinef&)

```

GetTransform, SetTransform はそれぞれフレームとその親フレームとの間の相対的な座標変換を取得/設定します。例えば

```
frame->SetTransform(Affinef::Trn(1.0, 0.0, 0.0));
```

とすると親フレームに対して相対的に x 方向に 1.0 移動します。

## 8.6 カメラ

カメラは描画における視点の設定と投影変換を管理します．はじめにカメラのディスクリプタを見ていきます．

### GRCameraDesc

---

Vec2f	size	スクリーンサイズ
Vec2f	center	スクリーン中心座標
float	front	前方クリップ面
float	back	後方クリップ面

各変数の定義は Fig. 8.3(b),(c) を参照してください．設定を変更するには以下のようにします．

```
GRCameraDesc desc;
grScene->GetCamera()->GetDesc(&desc);
desc.front = 3.0f;
grScene->SetCamera(desc);
```

上では GetDesc 関数で既存の設定をディスクリプタにコピーし，front を変更してから SetCamera 関数で再設定しています．

一方，GRCamera の関数は以下の通りです．

### GRCameraIf

---

```
GRFrameIf* GetFrame()
void SetFrame(GRFrameIf*)
```

GetFrame，SetFrame 関数はカメラフレームを取得/設定します．Fig. 8.3(a) のように，カメラフレームはカメラの視点を定義します．

## 8.7 ライト

ライトはシーンの照明を設定するための描画アイテムです．ライトのクラス GRLight のディスクリプタの代表的な変数を以下に示します．

GRLightDesc		
Vec4f	ambient	環境光
Vec4f	diffuse	拡散光
Vec4f	specular	鏡面光
Vec4f	position	ライト位置

減衰係数やスポットライトなどのより詳細な設定項目についてはソースファイルを参照してください。OpenGL の仕様と同様，position の第 4 成分 position.w が 0 の場合は平行光源となり，(x,y,z) 方向の無限遠にライトがあることになり，position.w が 1 の場合は (x,y,z) の位置に点光源がおかれます。

## 8.8 マテリアル

マテリアルは材質を指定するためのアイテムです。マテリアルのクラスは GRMaterial です。通常，マテリアルは次節で説明するメッシュの子描画アイテムとなります。ファイルからメッシュをロードする場合は，メッシュの作成と同時にマテリアルも自動的に作成され，メッシュの子として追加されます。

## 8.9 メッシュ

メッシュは形状を表現するための描画アイテムです。メッシュのクラスは GRMesh です。メッシュはディスクリプタを用いて手動で生成することも可能ですが，通常は FileIO モジュールを利用して外部のモデリングソフトで出力したファイルからメッシュをロードします。

## 8.10 レンダラ

レンダラの機能を項目別に説明します。レンダラは提供するプリミティブな描画機能は非常に多岐に渡りますが，これらのほとんどの関数は特別な描画処理を必要としない限りユーザが直接呼び出すことはありません。個々の関数を詳しく説明していくと膨大な量になりますので，ここでは一覧程度にとどめます。詳細な仕様はソースコードのコメントを参照してください。

### 基本機能

描画時のお決まりの処理です。8.3 節を参照してください。

## GRRenderIf

---

<code>void GetClearColor(Vec4f&amp;)</code>	背景色の取得
<code>void SetClearColor(const Vec4f&amp;)</code>	背景色の設定
<code>void ClearBuffer()</code>	描画バッファをクリア
<code>void BeginScene()</code>	描画の開始
<code>void EndScene()</code>	描画の完了
<code>void SwapBuffers()</code>	描画バッファのスワップ

## ディスプレイリスト

ディスプレイリストに関係する機能です。GRMesh が内部で使います。

## GRRenderIf

---

<code>int StartList()</code>	ディスプレイリスト作成開始
<code>void EndList()</code>	ディスプレイリスト作成完了
<code>void DrawList(int)</code>	ディスプレイリストの描画
<code>void ReleaseList(int)</code>	ディスプレイリストの解放

## デプステスト, アルファブレンディング, ライティング

描画機能を切り替えるための関数です。

## GRRenderIf

---

<code>void SetDepthWrite(bool)</code>	デプスバッファへの書き込み On/Off
<code>void SetDepthTest(bool)</code>	デプステストの On/Off
<code>void SetDepthFunc(TDepthFunc)</code>	デプスバッファの判定条件
<code>void SetAlphaTest(bool)</code>	アルファブレンディングの On/Off
<code>void SetAlphaMode(TBlendFunc, TBlendFunc)</code>	アルファブレンディングのモード
<code>void SetLighting(bool)</code>	ライティングの On/Off

## テクスチャ

GRRenderIf

---

<code>int LoadTexture(UTString)</code>	テクスチャのロード
<code>void SetTextureImage(UTString, int, int, int, int, char*)</code>	テクスチャの設定

## シェーダ

GRRenderIf

---

<code>void InitShader()</code>	シェーダの初期化
<code>void SetShaderFormat(ShaderType)</code>	シェーダフォーマットの設定
<code>bool CreateShader(UTString, UTString, GRHandler&amp;)</code>	シェーダオブジェクトの作成
<code>GRHandler CreateShader()</code>	シェーダオブジェクトの作成
<code>bool ReadShaderSource(GRHandler, UTString)</code>	シェーダプログラムをロード
<code>void GetShaderLocation(GRHandler, void*)</code>	ロケーション情報の取得

## 直接描画

### GRRenderIf

---

<code>void SetVertexFormat(const GRVertexElement*)</code>	頂点フォーマットの指定
<code>void SetVertexShader(void*)</code>	頂点シェーダーの指定
<code>void DrawDirect(TPrimitiveType, void*, size_t, size_t)</code>	頂点を指定してプリミティブを描画
<code>void DrawIndexed(TPrimitiveType, size_t*, void*, size_t, size_t)</code>	頂点とインデックスを指定してプリ ミティブを描画
<code>void DrawArrays(TPrimitiveType, GRVertexArray*, size_t)</code>	頂点の成分ごとの配列を指定して, プリミティブを描画
<code>void DrawArrays(TPrimitiveType, size_t*, GRVertexArray*, size_t)</code>	インデックスと頂点の成分ごとの配 列を指定して, プリミティブを描画

## 基本形状描画

GRRenderIf

---

<code>void DrawLine(Vec3f, Vec3f)</code>	線分を描画
<code>void DrawArrow(Vec3f, Vec3f, float, float, float, int, bool)</code>	矢印を描画
<code>void DrawBox(float, float, float, bool)</code>	直方体を描画
<code>void DrawSphere(float, int, int, bool)</code>	球体を描画
<code>void DrawCone(float, float, int, bool)</code>	円錐の描画
<code>void DrawCylinder(float, float, int, bool)</code>	円筒の描画
<code>void DrawCapsule(float, float, int, bool)</code>	カプセルの描画
<code>void DrawRoundCone(float, float, float, int, bool)</code>	球円錐の描画
<code>void DrawGrid(float, int, float)</code>	グリッドを描画
<code>void SetFont(const GRFont&amp;)</code>	フォントの設定
<code>void DrawFont(Vec2f, UTString)</code>	2次元テキストの描画
<code>void DrawFont(Vec3f, UTString)</code>	3次元テキストの描画

## マテリアル

GRRenderIf

---

<code>void SetMaterial(const GRMaterialDesc&amp;)</code>	描画マテリアルの設定
<code>void SetMaterial(const GRMaterialIf*)</code>	描画マテリアルの設定
<code>void SetMaterial(int)</code>	描画マテリアルの設定
<code>void SetLineWidth(float)</code>	線の太さの設定

## カメラ

GRRenderIf

---

<code>void SetCamera(const GRCameraDesc&amp;)</code>	カメラの設定
<code>const GRCameraDesc&amp; GetCamera()</code>	カメラの取得

## ライト

GRRenderIf

---

<code>void PushLight(const GRLightDesc&amp;)</code>	ライトをプッシュ
<code>void PushLight(const GRLightIf*)</code>	ライトをプッシュ
<code>void PopLight()</code>	ライトをポップ
<code>int NLights()</code>	ライトの数

## 座標変換

GRRenderIf

---

<code>void Reshape(Vec2f, Vec2f)</code>	ウィンドウサイズの変更
<code>void SetViewport(Vec2f, Vec2f)</code>	ビューポートの設定
<code>Vec2f GetViewportPos()</code>	ビューポート原点の取得
<code>Vec2f GetViewportSize()</code>	ビューポートサイズの取得
<code>Vec2f GetPixelSize()</code>	1 ピクセルの物理サイズを取得
<code>Vec3f ScreenToCamera(int, int, float, bool)</code>	スクリーン座標からカメラ座標への変換
<code>void EnterScreenCoordinate()</code>	スクリーン座標系へ切り替える
<code>void LeaveScreenCoordinate()</code>	スクリーン座標系から戻る



---

GRenderIf

---

<code>void SetViewMatrix(const Affinef&amp;)</code>	視点行列の設定
<code>void GetViewMatrix(Affinef&amp;)</code>	視点行列の取得
<code>void SetProjectionMatrix(const Affinef&amp;)</code>	投影行列の設定
<code>void GetProjectionMatrix(Affinef&amp;)</code>	投影行列の取得
<code>void SetModelMatrix(const Affinef&amp;)</code>	モデル行列の設定
<code>void GetModelMatrix(Affinef&amp;)</code>	モデル行列の取得
<code>void MultModelMatrix(const Affinef&amp;)</code>	モデル行列に変換をかける
<code>void PushModelMatrix()</code>	モデル行列をプッシュ
<code>void PopModelMatrix()</code>	モデル行列をポップ
<code>void ClearBlendMatrix()</code>	ブレンド変換行列のクリア
<code>bool SetBlendMatrix(const Affinef&amp;, int)</code>	ブレンド変換行列の設定



## 第 9 章

# FileIO

### 9.1 概要

FileIO はファイル入出力機能を提供するモジュールです。

### 9.2 FileIO SDK

FileIO モジュールのすべてのオブジェクトは SDK クラス FISdk によって管理されます。FISdk クラスは、プログラムの実行を通してただ 1 つのオブジェクトが存在するシングルトンクラスです。FISdk オブジェクトを作成するには以下のようにします。

```
FISdkIf* fiSdk = FISdkIf::CreateSdk();
```

通常この操作はプログラムの初期化時に一度だけ実行します。また、Framework モジュールを使用する場合はユーザが直接 FISdk を作成する必要はありません。

FISdk には以下の 2 つの機能があります。

- ファイルオブジェクトの作成
- インポートオブジェクトの作成

ファイルオブジェクトは、ファイルからのシーンのロードおよびセーブを担います。ファイルの基底クラスは FIFile で、ファイルフォーマットの種類ごとに専用のファイルクラスが派生します (Fig. 9.1)。

ファイル作成に関する FISdk の関数を以下に示します。



Fig. 9.1 Class hierarchy of file objects

## FISdkIf

---

```

FIFileSprIf* CreateFileSpr()
FIFileBinaryIf* CreateFileBinary()
FIFileXIf* CreateFileX()
FIFileVRMLIf* CreateFileVRML()
FIFileCOLLADAIf* CreateFileCOLLADA()
FIFileIf* CreateFileFromExt(UTString filename)

```

CreateFileFromExt は filename の拡張子からファイルフォーマットを判別して対応するファイルオブジェクトを作成します。

## 9.3 ファイルフォーマット

この節では Springhead でロード・セーブできるファイルのファイルフォーマットを紹介します。

### 9.3.1 spr ファイル

拡張子 .spr のファイルは、Springhead 独自のファイル形式です。人が読み書きしやすく、Springhead の仕様が変化しても余り影響を受けないような形式になっています。ファイルを手書きする場合はこの形式を使ってください。

spr ファイルはノード定義の繰り返しです。spr ファイルの例を示します。

```

PHSdk{                                #PHSdk ノード
    CDSphere sphere{                  # の子ノードに CDSphere ノードを追加
        material = {                  # CDSphere の material(PHMaterial 型) の

```

```

        mu = 0.2      # 摩擦係数 mu に 0.2 を代入
    }
    radius = 0.5      # radius に 0.5 を代入
}
CDBox bigBox{
    boxsize = 2.0 1.1 0.9
}

```

Spr ファイルのノードはディスクリプタ (3.4) 節を参照) に 1 対 1 で対応します。ディスクリプタさえ用意すれば自動的に使えるノードの型が増えます。ファイルで値を代入しないと、ディスクリプタの初期値になります。上の例では、PHSdk に追加される sphere(CDSphere 型) は、

```

CDSphereDesc desc;
desc.material.mu = 0.2;
desc.radius = 0.5;

```

としたディスクリプタ desc で作るのと同じことになります。

Spr ファイルの文法を BNF + 正規表現で書くと

```

spr    = node*
node   = node type, (node id)?, block
block  = '{' (node|refer|data)*  '}'
refer  = '*' node id
data   = field id, '=', (block | right)
right  = '[' value*, ']' | value
value  = bool | int | real | str | right

```

となります。right 以降の解釈は field の型に依存します。

### 9.3.2 X ファイル

「X ファイル」は、Direct3D のファイルフォーマットで、拡張子は .x です。モデリングソフト XSI で使われており、多くのモデリングツールで出力できます。3D の形状データ、マテリアル、テクスチャ、ボーンなどを含めることができます。Springhead2 では、標準的な X ファイルのロードと、Springhead2 独自のノードのロードとセーブができます。ただし独自ノードを手書きする場合は ??ecSprFile の方が書きやすく便利ですので

そちらの使用をおすすめします。

X ファイルの例を示します。

```
xof 0302txt 0064          #最初の行はこれから始まる

#   ノードは ,
#       型名 , ノード名 { フィールドの繰り返し   子ノード }
#   かなる .
PHScene scene1{
    0.01;0;;                #フィールド は 値; の繰り返し
    1;0;-9.8;0;;            #値は 数値, 文字列またはフィールド
    PHSolid soFloor{        #子ノードは, ノードと同じ
        (省略)
    }
}
# コメントは #以外に // も使える
```

### 独自ノードの定義

Springhead2 の通常のノードは、オブジェクトのディスクリプタ(??節)に1対1で対応します。ロード時には、ディスクリプタに対応するオブジェクトが生成され、シーングラフに追加されます。セーブ時には、オブジェクトからディスクリプタを読み出し、ノードの形式でファイルに保存されます。

オブジェクトのディスクリプタには、必ず対応するノードがあります。例えば、SprPHScene.h には、

```
struct PHSceneState{
    double timeStep;        ///< 積分ステップ
    unsigned count;         ///< 積分した回数
};

struct PHSceneDesc:PHSceneState{
    /// 接触・拘束解決エンジンの種類
    enum ContactMode{ MODE_NONE, MODE_PENALTY, MODE_LCP};
    Vec3f gravity;          ///< 重力加速度ベクトル。デフォルト値は(0.0f, -9.8f, 0.0f) .
};
```

のように、ステートとディスクリプタが宣言されています。この PHSceneDesc に対応する X ファイルのノードは、

```
PHScene scene1{
    0;;          #PHSceneState::count      最後の; は PHSceneState 部の終わりを
    示す。
    1;           #PHSceneDesc::ContactMode
    0;-9.8;0;;   #PHSceneDesc::gravity     最後の; は PHSceneDesc 部の終わりを
    示す。
}
```

のようになります。クラスのメンバ変数がそのままフィールドになります。また、基本クラスは、先頭にフィールドが追加された形になります。

通常ノードの一覧は TBU: デスクリプター一覧のページ を参照下さい。

#### X ファイルのノード

Springhead2 の独自ノードだけでなく、普通の X ファイルのノードもロードできます。X ファイルには、

```
Frame{
    FrameTransfromMatrix{ 1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1; }
}
```

のようなフレームのノード型がありますが、Springhead2 には対応するディスクリプタやオブジェクトがありません。そこで、これらは、GRFrame や PHFrame に変換されてロードされます。TBU: ノード一覧のページ (pageNodeDefList) を参照下さい。

## 9.4 ファイルのロード

T.B.D.

## 9.5 ファイルのセーブ

T.B.D.

## 9.6 インポート情報の管理

T.B.D.



## 第 10 章

# HumanInterface

### 10.1 概要

HumanInterface モジュールは、ハードウェアや入力デバイスを利用するための処理系に依存しないインタフェースを提供します。

ほとんどの場合、HumanInterface の機能は Framework モジュールを介してアクセスすることになります。この場合は、後述するヒューマンインタフェースオブジェクトやデバイスの作成をユーザ自身で行う必要はありません。

### 10.2 HumanInterface SDK

HumanInterface モジュールのすべてのオブジェクトは SDK クラス HISdk によって管理されます。HISdk クラスは、プログラムの実行を通してただ 1 つのオブジェクトが存在するシングルトンクラスです。HISdk オブジェクトを作成するには以下のようにします。

```
HISdkIf* hiSdk = HISdkIf::CreateSdk();
```

通常この操作はプログラムの初期化時に一度だけ実行します。また、Framework モジュールを使用する場合はユーザが直接 HISdk を作成する必要はありません。

### 10.3 クラス階層とデータ構造

HumanInterface モジュールのクラス階層を Fig. 10.1 に示します。

デバイスには実デバイスと仮想デバイスがあります。実デバイスは現実のハードウェアに対応し、例えば Win32 マウスやあるメーカーの A/D 変換ボードを表す実デバイスがあります。一方、仮想デバイスは実デバイスが提供する機能単位を表し、処理系に依存しま

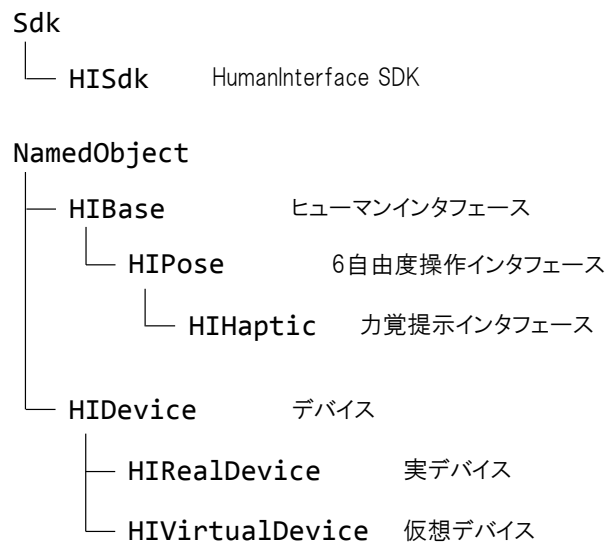


Fig. 10.1 HumanInterface class hierarchy



Fig. 10.2 HumanInterface module data structure

せん．例えば，1つの A/D 変換ポートや抽象化されたマウスインタフェースがこれにあたります．基本的に，初期化時を除いてはユーザは実デバイスに触れることはなく，仮想デバイスを通じてそれらの機能を利用することになります．

ヒューマンインタフェースはデバイスよりも高度で抽象化された操作インタフェースを提供します．

次に HumanInterface モジュールのデータ構造を Fig. 10.2 に示します．HISdk オブジェクトはヒューマンインタフェースプールとデバイスプールを持っています．デバイス

プールとは実デバイスの集まりで、それぞれの実デバイスはその機能をいくつかの仮想デバイスとして外部に提供します。

デバイスの機能を使うには、

1. 実デバイスを作成する
2. 実デバイスが提供する仮想デバイスにアクセスする

という 2 段階の手順を踏みます。以下にそれに関する HISdk の関数を紹介します。

HISdkIf

---

```
HIRealDeviceIf* AddRealDevice(const IfInfo* ii, const void* desc = NULL)
```

型情報 ii とディスクリプタ desc を指定して実デバイスを作成する。

たとえば、内部で GLUT を用いるキーボード・マウス実デバイスを作成するには

```
hiSdk->FindRealDevice(DRKeyMouseGLUTIf::GetIfInfoStatic());
```

とします。

仮想デバイスを取得および返却する方法には HISdk を介する方法と HIRealDevice を直接呼び出す方法の 2 通りがあります。

HISdkIf

---

```
HIVirtualDeviceIf* RentVirtualDevice(const IfInfo* ii, const char* name, int portNo)
```

型情報 ii で指定した仮想デバイスを取得する。オプションで実デバイスの名前 name とポート番号 portNo を指定できる。

```
bool ReturnVirtualDevice(HIVirtualDeviceIf* dev)
```

不要になった仮想デバイスを返却する。

RentVirtualDevice はデバイスプールをスキャンして型情報に合致した最初の仮想デバイスを返します。実デバイスを限定したい場合は name で実デバイス名を指定します。また、複数の仮想デバイスを提供する実デバイスもあります。この場合はポート番号 portNo で取得したい仮想デバイスを指定できます。

デバイスの競合を防ぐために、一度取得された仮想デバイスは利用中状態になります。利用中のデバイスは新たに取得することはできません。使い終わったデバイスは

ReturnVirtualDevice で返却することによって再び取得可能になります。

HIRealDeviceIf

---

```
HIVirtualDeviceIf* Rent(const IfInfo* ii, const char* name, int  
portNo)
```

```
bool Return(HIVirtualDeviceIf* dev)
```

こちらは実デバイスから直接取得，返却するための関数です．機能は同様です．

## 10.4 実デバイス

Springhead ではいくつかのメーカ製のハードウェアが実デバイスとしてサポートされていますが，処理系に強く依存する部分であるため本ドキュメントの対象外とします．興味のある方はソースコードを見てください．

## 10.5 キーボード・マウス

キーボードおよびマウスの機能は包括して 1 つのクラスとして提供されています．キーボード・マウスの仮想デバイスは DVKeyMouse です．実デバイスとしては Win32 API を用いる DRKeyMouseWin32 と GLUT を用いる DRKeyMouseGLUT があります．

### 仮想キーコード

Ascii 外の特殊キーには処理系依存のキーコードが割り当てられています．この差を吸収するために以下のシンボルが DVKeyCode 列挙型で定義されています．

## DVKeyCode

---

ESC	エスケープ
F1 - F12	ファンクションキー
LEFT	
UP	
RIGHT	
DOWN	
PAGE_UP	Page Up
PAGE_DOWN	Page Down
HOME	Home
END	End
INSERT	Insert

必要に応じてシンボルが追加される可能性がありますので、完全なリストはヘッダファイルで確認してください。

## コールバック

DVKeyMouse からのイベントを処理するには DVKeyMouseHandler クラスを継承し、イベントハンドラをオーバーライドします。DVKeyMouseHandler はいくつかのヒューマンインタフェースクラスが継承しているほか、後述するアプリケーションクラス FWApp も継承しています。

---

DVKeyMouseHandler

---

```
virtual bool OnMouse(int button, int state, int x, int y)
```

マウスクリック時の処理

button DVButtonMask 値の組み合わせ

state DVButtonSt 値

x, y カーソル座標

```
virtual bool OnDoubleClick(int button, int x, int y)
```

ダブルクリック時の処理

button DVButtonMask 値の組み合わせ

x, y カーソル座標

```
virtual bool OnMouseMove(int state, int x, int y, int zdelta)
```

マウスカーソル移動時の処理

state DVButtonSt 値

x, y カーソル座標

zdelta マウスホイール回転量

```
virtual bool OnKey(int state, int key, int x, int y)
```

キー入力処理

state DVButtonSt 値

key ascii コードか DVKeyCode 値

x, y カーソル座標

上のイベントハンドラに渡される列挙子は以下の通りです .

---

DVButtonMask

---

LBUTTON	左ボタン押し下げ
RBUTTON	右ボタン押し下げ
MBUTTON	中ボタン押し下げ
SHIFT	Shift キー押し下げ
CONTROL	Ctrl キー押し下げ
ALT	Alt キー押し下げ

---

DVButtonSt

---

DOWN	ボタンプッシュ
UP	ボタンリリース

---

DVKeySt

---

PRESSED	押されている
TOGGLE_ON	トグルされている

## 10.6 ジョイスティック

ジョイスティックの仮想デバイスは DVJoyStick です。実デバイスとしては GLUT を用いる DRJoyStickGLUT のみがあります。

T.B.D.

## 10.7 トラックボール

トラックボールはキーボード・マウスにより並進・回転の 6 自由度を入力するヒューマンインタフェースです。

T.B.D.

## 10.8 Spidar

Spidar はワイヤ駆動型の 3 軸・6 軸力覚提示ヒューマンインタフェースです。

T.B.D.



## 第 11 章

# Creature

Creature モジュールは、物理シミュレータを用いてバーチャルクリーチャ（自律動作するキャラクタ）を作成する機能を提供します。

Springhead の物理シミュレーション機能は、人間・動物・キャラクタ・ロボット等の身体動作をシミュレーションすることにおいても利用価値があります。剛体・関節系で身体モデルを作成し、関節に組み込まれた制御機能や関節系の IK 機能を用いて身体動作を生成することができます。物理シミュレータ内の情報（物体の運動・形状・接触力等）を利用してバーチャルな感覚（センサ）情報の生成もできます。感覚・制御のループを回すことで自律動作するキャラクタやロボットが実現できます。

こうしたバーチャルなキャラクタ・ロボット等を総称して、バーチャルクリーチャ（Creature：生き物）と呼びます。

### 11.1 Creature モジュールの構成

下図に Creature モジュールの構成を示します。

```
CRCreature
+-- CBody
|+-- CBodyPart (CObject)
+-- CEngine
+-- CController
```

CRCreature は、バーチャルクリーチャ 1 体分の機能を統括するオブジェクトです。身体と制御エンジンを有しています。

CBody は、バーチャルクリーチャの身体モデルを統括します。身体モデルは身体構成部品の集合体です。CBodyPart は、身体構成部品ひとつひとつに対応するオブジェクトです。剛体あるいは関節の 2 種類があり、さらに IK での動作に対応するかどうかによ

て計 4 種類に細分されます。

CREngine は、バーチャルクリーチャのステップ処理の実行主体です。CRCreature の Step 関数が 1 回呼ばれるたびに、CRCreature が保持する全ての CREngine の Step 関数が順に実行されます。CRController は CREngine の一種で、バーチャルクリーチャの身体制御を担当します。実際の制御機能は CRController を継承した各クラスが担当します。眼球運動制御、首運動制御等があります。

## 第 12 章

# Framework

### 12.1 概要

Framework はモジュール間の連携を促進してアプリケーションの作成を支援するためのモジュールです。

Framework モジュールのデータ構造を Fig. 12.1 に示します。最上位にはアプリケーションクラス FWApp があります。ユーザは FWApp を継承することで独自のアプリケーションを作成します。FWApp の中にトップレベルウィンドウ (FWWin) の配列、Framework SDK (FWSdk)、およびウィンドウマネージャ (FWGraphicsAdaptee) を持ちます。

FWWin はトップレベルウィンドウで、そのウィンドウに対応する入力デバイスやビューポート情報を保持するレンダラ、そのウィンドウと関連付けられたシーンへの参照などを持ちます。また、図中では省略されていますがサブウィンドウや GUI コントロールを持つこともできます。

FWSdk の役割は周辺モジュールの機能統合です。その中に周辺モジュールの SDK クラスや Framework シーン (FWScene) の配列を持ちます。

ウィンドウマネージャは処理系に依存するデバイスの初期化やイベントハンドリングを行います。ウィンドウマネージャはインタフェースを公開していませんのでユーザはその存在を陽に意識する必要はありません。図ではデータ構造の説明のためにあえて記載しています。

以下では個々の構成要素について説明していきます。

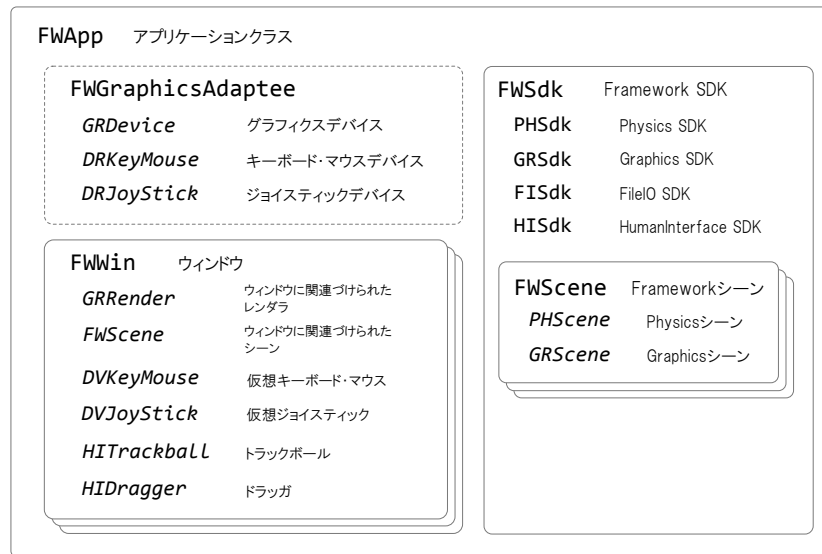


Fig. 12.1 Framework data structure

## 12.2 Framework SDK

Framework モジュールのすべてのオブジェクトは SDK クラス `FWSdk` によって管理されます。 `FWSdk` クラスは、プログラムの実行を通してただ 1 つのオブジェクトが存在するシングルトンクラスです。 `FWSdk` オブジェクトを作成するには以下のようにします。

```
FWSdkIf* fwSdk = FWSdkIf::CreateSdk();
```

通常この操作はプログラムの初期化時に一度だけ実行します。 `FWSdk` を作成すると、同時に `PHSdk`、`GRSdk`、`FISdk`、`HISdk` も作成されます。したがってこれらをユーザが手動で作成する必要はありません。各モジュールの機能にアクセスするには以下の関数により SDK を取得します。

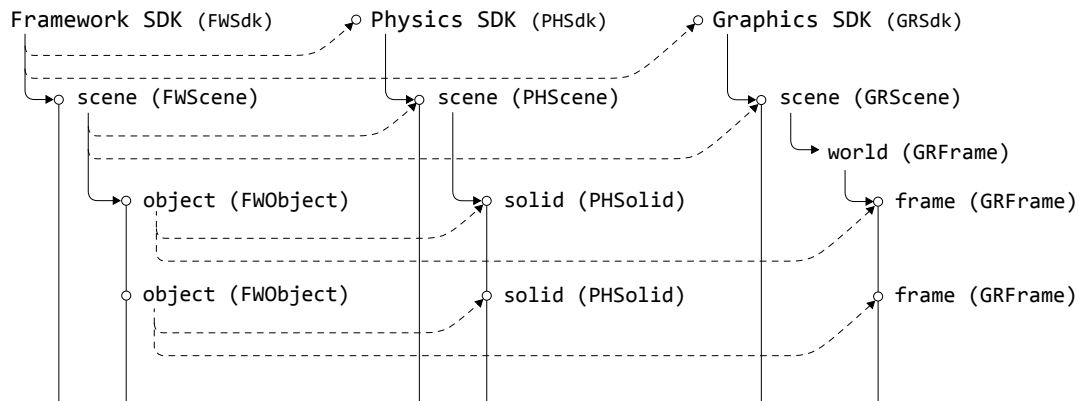


Fig. 12.2 Data structure of Framework, Physics and Graphics modules

### FWSdkIf

PHSdkIf\* GetPHSdk()

Physics SDK を取得する .

GRSdkIf\* GetGRSdk()

Graphics SDK を取得する .

FISdkIf\* GetFISdk()

FileIO SDK を取得する .

HISdkIf\* GetHISdk()

HumanInterface SDK を取得する .

## 12.3 Framework シーン

Framework モジュールの主な機能の 1 つに Physics シーンと Graphics シーンの同期があります . Fig. 12.2 に 3 つのモジュールの SDK とシーンの関係を示します . FWSdk は任意の数のシーン (FWScene クラス) を保持します . また , シーンは任意の数のオブジェクト (FWObject クラス) を保持します . Fig. 12.2 に示すように , オブジェクトは Physics モジュールの剛体と Graphics モジュールのトップフレームを一対一に対応づけます . ここでトップフレームとはワールドフレームの直下にあるフレームのことです . 物理シミュレーションにより計算される剛体の運動をフレームの座標変換に反映させること

で、シミュレーションの様子を Graphics モジュールの機能を利用して可視化することができます。

シーン作成に関する FWSdk の関数を以下に示します。

FWSdkIf

---

```
FWSceneIf* CreateScene(const PHSceneDesc&, const GRSceneDesc&)
```

シーンを作成する。

```
int NScene()
```

シーンの数を取得する

```
FWSceneIf* GetScene(int i)
```

i 番目のシーンを取得する。

```
void MergeScene(FWSceneIf* scene0, FWSceneIf* scene1)
```

scene1 の子オブジェクトを scene0 に移す。

シーンを作成するには以下のようにします。

```
FWSceneIf* fwScene = fwSdk->CreateScene();
```

FWScene を作成すると、同時に PHScene と GRScene も作成され、FWScene とリンクされます。CreateScene にディスクリプタを指定することもできます。NScene は作成したシーンの数を返します。

シーンを取得するには GetScene を使います。GetScene に指定する整数は作成された順番にシーンに与えられる通し番号です。

```
fwSdk->CreateScene();           // create two scenes
fwSdk->CreateScene();
FWSceneIf *fwScene0, *fwScene1;
fwScene0 = fwSdk->GetScene(0);  // get 1st scene
fwScene1 = fwSdk->GetScene(1);  // get 2nd scene
```

MergeScene を使うと 2 つのシーンを統合して 1 つのシーンにできます。

```
fwSdk->MergeScene(fwScene0, fwScene1);
```

上のコードでは scene1 が持つ FWObject が scene0 に移され、同時にシーンが参照する

PHScene と GRScene に関してもそれぞれの MergeScene 関数により統合が行われます。

次に、FWScene の基本機能を以下に示します。

#### FWSceneIf

---

<code>void SetPHScene(PHSceneIf*)</code>	Physics シーンの設定
<code>PHSceneIf* GetPHScene()</code>	Physics シーンの取得
<code>void SetGRScene(GRSceneIf*)</code>	Graphics シーンの設定
<code>GRSceneIf* GetGRScene()</code>	Graphics シーンの取得
<code>FWObjectIf* CreateFWObject()</code>	オブジェクトの作成
<code>int NObject()const</code>	オブジェクトの数
<code>FWObjectIf** GetObjects()</code>	オブジェクト配列の取得
<code>void Sync(bool)</code>	同期

[Set|Get] [PH|GR]Scene 関数はシーンに割り当てられた PHScene や GRScene を取得したり，別のシーンを割り当てたりするのに使用します。

CreateFWObject 関数は FWObject オブジェクトを作成します。このとき，新たに作成された FWObject には PHSolid および GRFrame は割り当てられていない状態になっているので注意してください。これらも同時に作成するには，以下のコードを 1 セットで実行します。

```
FWObjectIf* fwObj = fwScene->CreateFWObject();
fwObj->SetPHSolid(fwScene->GetPHScene()->CreateSolid());
fwObj->SetGRFrame(
    fwScene->GetGRScene()->CreateVisual(GRFrameDesc())->Cast);
```

Sync 関数は PHScene と GRScene の同期に用います。

```
fwScene->Sync(true);
```

とすると，このシーンが参照する PHScene 中の剛体の位置と向きが，同じくこのシーンが参照する GRScene 中のトップフレームの位置と向きに反映されます。このときの剛体とトップフレームとの対応関係は FWObject により定義されます。

逆に

```
fwScene->Sync(false);
```

とすると，同様のメカニズムで各トップフレームの位置と向きが対応する剛体に反映されます。

## 12.4 シーンのロードとセーブ

FileIO モジュールを利用してシーンをロード，セーブするための関数が用意されています．まずロードには以下の関数を用います．

FWSdkIf

---

```
bool LoadScene(UTString path, ImportIf* imp, const IfInfo* ii,  
ObjectIfs* objs)  
シーンをロードする．
```

path はロードするファイルへのパスを格納した文字列です．imp にはインポート情報を格納するための Import オブジェクトを与えます．インポート情報を記憶する必要のない場合は NULL で構いません．ii はロードするファイルの種類を明示するための型情報です．NULL を指定するとパスの拡張子から自動判別されます．objs はロードによって作成されるオブジェクトツリーの親オブジェクトを格納した配列です．

ロードに成功すると true，失敗すると false が返されます．ロードされたシーンは FWSdk のシーン配列の末尾に加えられます．

次に，シーンをセーブするには以下の関数を使います．

FWSdkIf

---

```
bool SaveScene(UTString path, ImportIf* imp, const IfInfo* ii,  
ObjectIfs* objs)  
シーンをセーブする．
```

引数の意味は LoadScene と同様です．imp にはロード時に記憶したインポート情報を与えます．省略するとシーン全体が単一のファイルにセーブされます．

セーブに成功すると true，失敗すると false が返されます．

## 12.5 Framework オブジェクト

FWObject は PHSolid と GRFrame の橋渡しが主な役割ですので，それ自体はそれほど多くの機能を持っていません．



## 12.6 アプリケーションクラス

Springhead を利用するアプリケーションの作成を容易にするために、アプリケーションクラス FWApp が用意されています。2.6 に FWApp を使って簡単なアプリケーションを作成する方法について説明しましたのでそちらも合わせて参考にしてください。

冒頭で説明した通り、Springhead のほとんどのオブジェクトは、親オブジェクトの Create 系関数を使って作成しますが、FWApp は例外的に、C++ のクラス継承を用いてユーザのアプリケーションクラスを定義する方法をとります。この方が仮想関数によって動作のカスタマイズがフレキシブルに行えるからです。

以下では FWApp の機能やユーザが実装すべき仮想関数について順に見ていきます。

### 初期化

FWApp の初期化処理は仮想関数 Init で行います。

#### FWApp

---

```
virtual void Init(int argc, char* argv[])
```

以下に Init 関数のデフォルトの実装を示します。

```
void FWApp::Init(int argc, char* argv[]){
    // create SDK
    CreateSdk();
    // create a single scene
    GetSdk()->CreateScene();
    // initialize window manager
    GRInit(argc, argv);
    // create main window
    CreateWin();
    // create timer
    CreateTimer();
}
```

はじめに

```
CreateSdk();
```

で SDK を作成します．つぎに

```
GRInit(argc, argv);
```

でウィンドウマネジャが作成されます．デフォルトでは GLUT を用いるウィンドウマネジャが作成されます．さらに

```
GetSdk()->CreateScene();
```

で FWScene を 1 つ作成します．つづいて

```
CreateWin();
```

でメインウィンドウを作成します．最後に

```
CreateTimer();
```

でタイマを作成します．

この基本処理に追加してなんらかの処理を行う場合は

```
virtual void Init(int argc = 0, char* argv[] = 0){  
    // select GLUI window manager  
    SetGRAdaptee(TypeGLUI);  
  
    // call base Init  
    FWApp::Init(argc, argv);  
  
    // do extra initialization here  
  
}
```

のように，FWApp::Init を実行してから追加の処理を行うのが良いでしょう．一方，以下に挙げるようなカスタマイズが必要な場合は Init 関数の処理全体を派生クラスに記述する必要があります．

- シーン生成をカスタマイズしたい
- ウィンドウの初期サイズやタイトルを変更したい
- 異なる種類のタイマが使いたい

この場合は，上に載せた Init のデフォルト処理をもとに必要な部分に修正を加えるのが

良いでしょう。

プログラムの全体の構造は通常以下ようになります。

```
MyApp app;

int main(int argc, char* argv[]){
    app.Init(argc, argv);
    app.StartMainLoop();
    return 0;
}
```

ここで MyApp はユーザが定義した FWApp の派生クラスです（もちろん他の名前でも構いません）。MyApp のインスタンスをグローバル変数として定義し、main 関数で Init、StartMainLoop を順次実行します。StartMainLoop 関数はアプリケーションのメインループを開始します。

## タイマ

タイマの作成には CreateTimer 関数を使います。通常、CreateTimer は Init の中で呼びます。

FWApp

---

```
UTTimerIf* CreateTimer(UTTimerIf::Mode mode)
```

引数 mode に指定できる値は UTimer の SetMode と同じです。5.4 節を参照してください。戻り値として UTimer のインタフェースが返されます。周期などの設定はこのインタフェースを介して行います。

シミュレーション用と描画用に 2 つのタイマを作成する例を以下に示します。

```
UTTimerIf *timerSim, *timerDraw;
timerSim = CreateTimer(MULTIMEDIA);
timerSim->SetInterval(10);
timerDraw = CreateTimer(FRAMEWORK);
timerDraw->SetInterval(50);
```

この例ではシミュレーション用には周期を 10[ms] のマルチメディアタイマを使い、描画用には周期 50[ms] のフレームワークタイマ（GLUT タイマ）を使っています。

タイマを始動すると、周期ごとに以下の仮想関数が呼ばれます。

FWApp

---

```
virtual void TimerFunc(int id)
```

タ

イマの判別は引数 *id* で行います。

TimerFunc のデフォルトの振る舞いでは、カレントウィンドウのシーンの Step を呼び、つぎに PostRedisplay で再描画要求を発行します（その結果、直後に Display 関数が呼び出されます）。この振る舞いをカスタマイズしたい場合は TimerFunc 関数をオーバーライドします。

```
void TimerFunc(int id){
    // proceed simulation of scene attached to current window
    if(id == timerSim->GetID()){
        GetCurrentWin()->GetScene()->Step();
    }
    // generate redisplay request
    else if(id == timerDraw->GetID()){
        PostRedisplay();
    }
}
```

この例ではシミュレーションと描画に異なる 2 つのタイマを使用しています。

## 描画

描画処理は次の仮想関数で行います。

FWApp

---

```
virtual void Display()
```

Display は描画要求が発行されたときに呼び出されます。描画要求は PostRedisplay 関数で行います。

FWApp

---

```
virtual void PostRedisplay()
```

Display 関数のデフォルトの振る舞いではカレントウィンドウの Display 関数が呼ばれます。

## キーボード・マウスイベント

FWApp は各ウィンドウに関連付けられた仮想キーボード・マウスデバイス DVKeyMouse にコールバック登録されています。したがって以下の仮想関数をオーバーライドすることでキーボード・マウスイベントを処理できます。

FWApp

---

```
virtual bool OnMouse(int button, int state, int x, int y)
virtual bool OnDoubleClick(int button, int x, int y)
virtual bool OnMouseMove(int state, int x, int y, int zdelta)
virtual bool OnKey(int state, int key, int x, int y)
```

各イベントハンドラの詳細については 10.5 節を参照して下さい。

## 12.7 ウィンドウ

ウィンドウやその他の GUI コントロールの作成も Framework によってサポートされています。すでに述べてきたとおり、FWApp はトップレベルウィンドウの配列を持ちます。

## 12.8 Framework を用いたシミュレーションと描画

Framework モジュールを介して物理シミュレーションを行うには以下の関数を使います。

FWSdkIf

---

```
void Step()
FWSdk の Step はアクティブシーンの Step を呼びます。したがって GetScene()->Step()
```

と等価です。一方 FWScene の Step は、保持している PHScene の Step を呼びます。したがって GetPHScene()->Step() と等価です。両方とも薄いラッパー関数ですが、ユーザのタイプ回数節約のために用意されています。

Framework を用いた描画には 2 通りの方法があります。1 つは Graphics のシーングラフを用いる方法、もう 1 つは Physics シーンを直接描画する方法です。後者はデバッグ描画とも呼ばれています。

---

#### FWSdkIf

---

```
void Draw()
void SetDebugMode(bool)
bool GetDebugMode()
```

Draw 関数は描画モードに応じた描画処理を行います。Draw は通常アプリケーションの描画ハンドラから呼び出します。[Set|Get]DebugMode は通常描画モード (false) とデバッグ描画モード (true) を切り替えます。

通常描画モードにおいて Draw 関数を呼ぶと、はじめにアクティブシーンについて Sync(true) が呼ばれ、剛体の状態がシーングラフに反映されます。次にアクティブシーンが参照する GRScene の Render 関数が呼ばれ、シーングラフが描画されます。この方法ではシーングラフが持つライトやテクスチャなどの情報を最大限利用してフォトリアルスティックな描画が可能です。その反面、物理シミュレーションが主目的である場合にはシーングラフの構築という付加的なコストを支払わなければならないというデメリットもあります。

デバッグ描画については次節で説明します。

## 12.9 デバッグ描画

デバッグ描画モードでは PHScene の情報だけを用いて描画が行われるので、シーングラフ構築の手間が省けます。また、剛体に加わる力などの物理シミュレーションに関する情報を可視化することができます。一方で、予約色しか使えないなど、描画の自由度には一定の制約が生じます。

デバッグ描画モードでは FWScene の DrawPHScene 関数により描画処理が行われます。

---

#### FWSceneIf

---

```
void DrawPHScene(GRRenderIf* render)
```

DrawPHScene は、各剛体に割り当てられている衝突判定形状、座標軸、作用している力、接触断面などを描画します。項目別に描画を行ったり、描画色を設定するには後述する描画制御関数を用います。

## デバッグ描画時のカメラとライト

デバッグ描画においてもカメラの情報は GRScene が参照されます。もし GRScene がカメラを保有している場合はそのカメラの Render が呼ばれ、視点と投影変換が設定されます。GRScene がカメラを持たない場合は手動で設定する必要があります。

ライトについては、もし外部でレンダラに対してライト設定がされている場合はその設定が優先され、レンダラが 1 つもライトを持たない場合は内部でデフォルトライトが設定されます。

## 個別の描画

以下の関数は DrawPHScene から呼び出されますが、ユーザが個別に呼び出すこともできます。

FWSceneIf

---

void DrawSolid(GRRenderIf*, PHSolidIf*, bool)	剛体を描画
void DrawShape(GRRenderIf*, CDShapeIf*, bool)	形状を描画
void DrawConstraint(GRRenderIf*, PHConstraintIf*)	拘束を描画
void DrawContact(GRRenderIf*, PHContactPointIf*)	接触を描画
void DrawIK(GRRenderIf*, PHIKEngineIf*)	IK 情報を描画

## 描画制御

以下の関数は描画の On/Off を切り替えます。

FWSceneIf

---

```
void SetRenderMode(bool solid, bool wire)
void EnableRender(ObjectIf* obj, bool enable)
void EnableRenderAxis(bool world, bool solid, bool con)
void EnableRenderForce(bool solid, bool con)
void EnableRenderContact(bool enable)
void EnableRenderGrid(bool x, bool y, bool z)
void EnableRenderIK(bool enable)
```

SetRenderMode はソリッド描画（面を塗りつぶす）とワイヤフレーム描画（面の輪郭）の On/Off を切り替えます。

EnableRender は指定したオブジェクトの描画の On/Off を切り替えます。項目ではなくオブジェクトレベルで描画制御したい場合に便利です。obj に指定できるのは剛体 (PHSolidIf\*) か拘束 (PHConstraintIf\*) です。

EnableRenderAxis は項目別に座標軸の描画を設定します。world はワールド座標軸，solid は剛体，con は拘束の座標軸です。

EnableRenderForce は力とモーメントの描画を設定します。solid は剛体に加わる力（ただし外力のみで拘束力は除く），con は拘束力です。

























EnableRenderGrid は各軸に関してグリッドの描画を設定します。

EnableRenderIK は IK 情報の描画を設定します。

以下の関数は描画属性を指定するのに使います。



Table 12.1 Reserved colors

RED	 (1.00, 0.00, 0.00)	GREEN	 (0.00, 1.00, 0.00)
BLUE	 (0.00, 0.00, 1.00)	YELLOW	 (1.00, 1.00, 0.00)
CYAN	 (0.00, 1.00, 1.00)	MAGENTA	 (1.00, 0.00, 1.00)
WHITE	 (1.00, 1.00, 1.00)	GRAY	 (0.50, 0.50, 0.50)
ORANGE	 (1.00, 0.27, 0.00)	BROWN	 (0.19, 0.00, 0.00)
LIGHT_BLUE	 (0.67, 0.84, 0.89)	MEDIUM_PURPLE	 (0.57, 0.43, 0.85)
DARK_GREEN	 (0.00, 0.39, 0.00)	DARK_VIOLET	 (0.57, 0.00, 0.82)
DARK_CYAN	 (0.00, 0.54, 0.54)	GREEN_YELLOW	 (0.67, 1.00, 0.18)
LIME_GREEN	 (0.19, 0.80, 0.19)	INDIAN_RED	 (0.80, 0.35, 0.35)
INDIGO	 (0.29, 0.00, 0.50)	GREEN_INDIGO	 (0.00, 0.19, 0.19)
OLIVE_GREEN	 (0.19, 0.39, 0.00)	NAVY_BLUE	 (0.19, 0.39, 0.79)
TURQUOISE_BLUE	 (0.39, 1.00, 0.79)	EMERALD_GREEN	 (0.59, 1.00, 0.39)

## FWSceneIf

---

```

void SetSolidMaterial(int mat, PHSolidIf* solid)
void SetWireMaterial (int mat, PHSolidIf* solid)
void SetAxisMaterial(int matX, int matY, int matZ)
void SetAxisScale(float world, float solid, float con)
void SetAxisStyle(int style)
void SetForceMaterial(int matForce, int matMoment)
void SetForceScale(float scaleForce, float scaleMoment)
void SetContactMaterial(int mat)
void SetGridOption(char axis, float offset, float size,
int slice)
void SetGridMaterial(int matX, int matY, int matZ)
void SetIKMaterial(int mat)
void SetIKScale(float scale)

```

SetSolidMaterial は指定した剛体のソリッド描画色を指定します。mat に指定できる値は GRRenderBaseIf が持つ予約色 (全 24 色, Table 12.1 参照) です。solid に NULL を指定するとすべての剛体の色が指定された値になります。SetWireMaterial は同様に剛体のワイヤフレーム描画色を指定します。

SetAxisMaterial は座標軸の色を x, y, z 個別に指定します。SetAxisScale は座標軸の縮尺を指定します。SetAxisStyle は座標軸のスタイルを指定します。

SetForceMaterial, SetForceScale はそれぞれ力（並進力とモーメント）の描画色と縮尺を指定します。

SetContactMaterial は接触断面の描画色を指定します。

SetGridOption はグリッドのオプションを指定します。SetGridMaterial はグリッドの描画色を指定します。

SetIKMaterial, SetIKScale は IK 情報の描画色と縮尺を指定します。

## 12.10 力覚インタラクションのためのアプリケーション

Springhead2 にはシーンとの力覚インタラクションのためのエンジン PHHapticEngine が含まれています。ここでは力覚インタラクションのためのアプリケーションの作成方法について説明します。まずは、通常の Framework アプリケーションの作成と同様に、ひな形クラスである FWApp を継承してアプリケーションを作成します。そして、Init 関数内で力覚インタラクションを有効化と、力覚インタラクションシステムのモードを設定します。

```
// given PHSceneIf* phScene,
    phScene->GetHapticEngine()->EnableHapticEngine(true);
    phScene->GetHapticEngine()->
        SetHapticEngineMode(PHHapticEngineDesc::MULTI_THREAD);
```

力覚インタラクションシステムのモードはシングルスレッドアプリケーションのための SINGLE\_THREAD, マルチメディアアプリケーションのための MULTI\_THREAD, 局所シミュレーションを利用した LOCAL\_DYNAMICS の 3 種類があります。標準では MULTI\_THREAD が設定されています。MULTI\_THREAD, LOCAL\_DYNAMICS のモードはマルチスレッドを利用したアプリケーションとなり、物理シミュレーションを実行する物理スレッド、力覚レンダリングを実行する力覚スレッドが並列に動きます。そのため、それぞれのスレッドをコールバックするためにタイマを設定し直す必要があります。

```
// given PHSceneIf* phScene,
int physicsTimerID, hapticTimerID // 各タイマの ID
// FWApp::TimerFunc をオーバーライドしたコールバック関数
void MyApp::TimerFunc(int id){
    if(hapticTimerID == id){
        // 力覚スレッドのコールバック
        phScene->StepHapticLoop();
    }else{
        // 物理スレッドのコールバック
        phScene->GetHapticEngine()->StepPhysicsSimulation();
        PostRedisplay(); // 描画
    }
}
```

次にユーザがオブジェクトとインタラクションするためのポインタ，力覚ポインタ PHHapticPointer を作ります．そして，どのインタフェースと結合するのかを設定します．PHHapticPointer は PHScene から作ることができます．PHHapticPointer は PHSolid を継承したクラスで PHSolid の関数を利用して，質量，慣性テンソル，形状などを合わせて設定します．例えば Spidar-G6 と接続する場合には，

```
// given PHSceneIf* phScene,
// given HISpidarIf* spg,
PHHapticPointerIf* pointer = phScene->CreateHapticPointer();
/*
    質量，慣性テンソル，形状などを設定する
*/
pointer->SetHumanInterface(spg);
```

とします．さらに PHHapticPointer について以下の関数を用いて，力覚提示のためのパラメータを設定します．

---

PHHapticPointerIf

---

```
void SetHumanInterface(HIBaseIf* interface)
void SetDefaultPose(Posed pose)
void SetPosScale(double scale)
void SetReflexSpring(float s)
void SetReflexDamper(float s)
void EnableFriction(bool b)
void EnableVibration(bool b)
void SetLocalRange(float s)
void SetHapticRenderMode(PHHapticPointerDesc::HapticRenderMode
m )
```

SetHumanInterface は力覚ポインタにヒューマンインタフェースを割り当てます。SetDefaultPose はシーン内での力覚ポインタの初期位置を指定します。SetPosScale はシーン内での力覚ポインタの可動スケールを指定します。SetReflexSpring は力覚レンダリング（反力計算）のためのバネ係数値を設定します。SetReflexDamper は力覚レンダリングのためのダンパ係数値を設定します。EnableFriction は力覚ポインタの摩擦提示を有効化します。EnableVibration は力覚ポインタの振動提示を有効化します。SetLocalRange は局所シミュレーションシステムを使用時の局所シミュレーション範囲を指定します。SetHapticRenderMode は力覚レンダリングのモードを指定します。

最後の SetHapticRenderMode には PENALTY, CONSTRAINT のモードがあります。PENALTY は力覚ポインタが剛体に接触した時の各接触点の侵入量とバネダンパ係数を乗じたものを足しあわせたものが反力として計算され、インタフェースから出力されます。CONSTRAINT は力覚ポインタが剛体に侵入していない状態（プロキシ）を求め、力覚ポインタとプロキシの距離の差分にバネダンパ係数を乗じたものを反力として計算します。

## 第 13 章

# Python 言語との連携

EmbPython モジュールは、スクリプト言語 Python との連携機能を提供します。Python インタプリタから Springhead の機能呼び出したり、Springhead アプリケーションに Python インタプリタを組み込んでスクリプティングエンジンとして使用するといった事ができます。

EmbPython モジュールの使用により、Python インタプリタ上に Springhead API クラスへのインタフェースクラスが提供されます。ユーザは Python インタフェースクラスを使用して Springhead の各機能にアクセスします。Python インタフェースクラスは内部的に Springhead の機能呼び出し、結果を Python インタフェースクラスに変換して返します。

### 13.1 利用法

大きく分けて二通りの利用法を想定しています。

一つは、C++ で実装された Springhead アプリケーションに対し、Python インタプリタを組み込むことです。Springhead アプリケーションの機能の一部を Python スクリプト記述し、拡張性を高めます。

もう一つは、Python インタプリタに対する外部拡張モジュール (Python DLL, pyd) として提供された Springhead を利用することで、Python アプリケーションに Springhead の機能を組み込む利用法です。

#### Springhead への Python 組込み

Springhead アプリケーションに Python インタプリタを組み込んで利用する方法を解説します。本節ではまず Springhead に同梱された Python インタプリタ組み込みサンプ

ルを紹介し、簡単な使い方を説明します。その後、サンプルにおける Python インタプリタ組み込みのためのソースコードについて解説します。

#### PythonSpr サンプルの紹介

Python インタプリタ組み込みサンプルは `src\Samples\EmbPython\PythonSpr` にあります。ビルドすると `PythonSpr.exe` ができます。

PythonSpr サンプルは標準的な Springhead サンプルアプリケーションフレームワークに Python インタプリタを組み込んだもので、物理シーンを構築・シミュレーション・描画する事ができます。Python インタプリタからは `phSdk` や `fwSdk` にアクセスすることができ、表示機能を切り替えたりシーンにオブジェクトを作成したりといったことが Python から行えます。

## 第 14 章

# トラブルシューティング

### 14.1 概要

本章ではよくあるトラブルとその対処方法を紹介します。

### 14.2 プログラムのビルド

主に Visual Studio 上で Springhead をビルド際によく遭遇するトラブルと対処法について述べます。

- |     |                                                               |
|-----|---------------------------------------------------------------|
| 現象  | 「include ファイルを開けません」というコンパイルエラーが出る                            |
| 解決策 | プロジェクト設定で Springhead のインクルードパスが設定されているか確認してください。              |
| 現象  | 「未解決の外部シンボル~が~で参照されました」というリンカエラーが出る                           |
| 解決策 | プロジェクト設定で Springhead のライブラリパスが設定されているか確認してください。               |
| 現象  | 「~は既に~で定義されています」というリンカエラーが出る                                  |
| 解決策 | プロジェクト設定で、ランタイムライブラリがユーザプログラムと Springhead ライブラリとで同一か確認してください。 |

## 14.3 物理シミュレーション関連

## 14.4 ファイル関連

- 現象            ファイルがロードできない
- 解決策 1       プログラムの実行ディレクトリを確認し、ファイルのパスが適切か確認してください。
- 解決策 2       バイナリ形式の X ファイルは読めません。テキスト形式で保存してください。
- 解決策 3       特殊な方言が使われていて読めないこともあります。開発者に問い合わせてください。

## 14.5 その他のトラブル

- 現象            なぜか動かない
- 解決策        バージョンが古い可能性があります。まず `svn update` を実行してコードを最新にし、次に Springhead ライブラリのリビルドを行ってください。
- 現象            それでも動かない
- 解決策        たまに (?)Springhead のバグであることがあります。開発者に問い合わせてください。



# 索引

Affined, 24  
Affinef, 24

Base, 21

CDBox, 37  
CDCapsule, 38  
CDConvexMesh, 40  
CDRoundCone, 40  
CDShape, 35  
CDSphere, 38  
Collision, 35

FIFile, 83  
FileIO, 83  
FISdk, 83  
Foundation, 29  
Framework, 99  
FWApp, 105  
FWObject, 101  
FWScene, 101  
FWSdk, 100

Graphics, 67  
GRCamera, 74  
GRDeviceGL, 67  
GRFrame, 72  
GRLight, 74  
GRMaterial, 75  
GRMesh, 75  
GRRender, 67, 75  
GRScene, 68  
GRSdk, 67  
GRVisual, 71

HISdk, 89  
HumanInterface, 89

IfInfo, 29

Matrix2d, 23  
Matrix2f, 23  
Matrix3d, 23  
Matrix3f, 23

NameManager, 31

Object, 29

PHBallJoint, 55  
PHConstraint, 51  
PHHingeJoint, 55  
PHJoint, 51  
PHMaterial, 41  
PHPathJoint, 55  
PHScene, 46  
PHSdk, 45  
PHSliderJoint, 55  
PHSolid, 48  
PHSpring, 56  
Physics, 45  
Posed, 26  
Posef, 26

Quaterionf, 25  
Quaterniond, 25

Scene, 32  
Springhead, 7

UTRef, 27  
UTRefCount, 27  
UTString, 27  
UTTimer, 32  
UTTreeNode, 27  
UTTypeDesc, 27

Vec2d, 21  
Vec2f, 21  
Vec3d, 21  
Vec3f, 21  
Vec4d, 21  
Vec4f, 21

アフィン変換, 24

カプセル, 38  
カメラ, 74  
関節座標系, 65

ギア, 65  
球, 38  
行列, 23

クォータニオン, 25

スマートポインタ, 27  
スライダ, 55

接触, 65

ソケット, 53

タイマ, 32

直方体, 37

デバイス, 67

凸メッシュ, 40

バスジョイント, 55  
バネ, 56

ヒンジ, 55

ファイル, 83  
物性, 41  
プラグ, 53  
フレーム, 72

ベクトル, 21

ポーズ, 26  
ボールジョイント, 55

マテリアル, 75  
丸コーン, 40

メッシュ, 75

ライト, 74

レンダラ, 67, 75