1. イントロダクション

1.1 概要
Box2D はゲーム向けの 2 次元剛体シミュレーションライブラリです.プログラマーは Box2D を使用して,ゲーム中の物体をリアルかつインタラクティブに動かすことができます.ゲームエンジンという視点から見れば,物理エンジンとはアニメーション処理のシステムに過ぎません.

Box2D は移植可能な C++で記述されています.ライブラリで定義されているほとんどの関数･クラスには“b2”という接頭辞がついています.これによって作り手のゲーム内で関数名などが衝突するのをできるかぎり防いでいます.

1.2 必須事項
このマニュアルでは読者が質量,力,トルク,力積などの基礎的な物理の概念を学んでいることを想定しています.もしそうでなければ,Google か Wikipedia にまず相談してください.

Box2D はゲーム開発者会議(Game Developer Conference)の物理チュートリアルの一つとして開発されました.これらのチュートリアルは box2d.org のダウンロードページからダウンロードできます.

Box2D は C++で書かれているので,C++のプログラミング経験が必要です.C++を学ぶ上で,Box2D を初めてのプログラムにするのはお勧めできません.コンパイル,リンク,デバッグについてあらかじめ学んでおく必要があります.

1.3 このマニュアルについて
このマニュアルは Box2D API の多くを網羅していますが,すべて状況において言及されているわけではありません.Box2D 内にある testbed を読んでより多く勉強されることをお勧めします.また,Box2D のコードは Doxygen の書式でコメントされているので,hyper-linked API ドキュメントを容易に作成できます.

1.4 フィードバックとバグ報告
Box2D について質問やフィードバックすることがあれば,フォーラム(http://code.google.com/p/box2d/)にコメントしてください.このフォーラムはまたディスカッションに最適な場所です.
Box2D の話題は Google Code Project で管理されているので過去の話題をさかのぼるのに最適ですし,議題がフォーラムの奥深くに落ち込んでしまうこともありません.ぶつかった問題の詳細をフォーラムに提供できれば問題解決の手助けとなるでしょう.

1.5 コンセプト
Box2D はいくつかの基本的なオブジェクトで動作します.ここではそれらのオブジェクトについて簡潔に述べ,詳細は後ほどのチャプターで説明します.

シェイプ(shape)
円や多角形といった物体の幾何学的な性質です.

剛体(rigid body)
どのような力がかかっても形や大きさが変化しないダイヤモンドのように固い物体です.以後の説明において物体はすべて剛体であるとします.

フィクスチャ(fixture)
物体の形を保持し,密度,摩擦係数,反発係数などといった物理条件を設定できます.

制限(constraint)
制限とは,物体から自由度を除いた物理的な連結です.2 次元の物体は自由度を 3 つ持っています（X 軸と Y 軸への移動と回転）.もし物体を振り子のように壁にピンで留めれば,その物体を壁に制限したことになりま

す.この場合,物体はピンの回りでのみ回転できるので,自由度を 2 つ制限したことになります.

接触制限(contact constraint)
物体同士の貫通を防ぎ,反発と摩擦を計算するために,特別な制限が用意されています.これらの接触制限は Box2D が自動で生成し,プログラマーが生成することはできません.

ジョイント(joint)
2 つかそれ以上の物体を相互に接続するのに使用します.Box2D は回転(revolute),プリズム(prismatic),間隔(distance)等のジョイントを提供しています.動く範囲を制限し,モーターのように回転させることもできます.

ジョイントモーター(joint motor)
ジョイントモーターは,ジョイントの自由度に合わせて接続している物体を動かします.例えば,モーターで肘を回転させることができます.

ワールド(world)
ワールドにはボディやフィクスチャ,制限が相互に関連付けられています.Box2D は複数のワールドを宣言できますが,通常はその必要はありません.

ソルバ(solver)
ワールドには時間を進め,ジョイントや接触を計算するのに使われるソルバが備えられています.Box2D のソルバは N 次の制限で動作する高性能な反復ソルバです.

連続衝突
ソルバは離散タイムステップで物体を進めます.これは干渉なしに物体を突き抜けてしまいます.

-- 図(トンネル効果) --

Box2D はこの貫通をうまく処理するアルゴリズムを含んでいます.まず,衝突アルゴリズムが始めの衝突時間を算出するために 2 物体の位置を補間します.次に,サブステップソルバが 2 物体を衝突の位置まで移動させ,衝突を計算します.

1.6 モジュール

Box2D は Common,Collision,Dynamics の 3 つのモジュールから成ります.Common モジュールはメモリ割り当て,計算,設定のコードです.Collision モジュールはシェイプ,a broad-phase,衝突関数・クエリを定義します.Dynamics モジュールはシミュレーションのワールド,ボディ,フィクスチャ,ジョイントを提供します.

1.7 単位

Box2D は浮動小数点数で動作し,許容値は Box2D を最適化するために使用されるべきです.これら許容値は MKS 単位系(メートル,キログラム,秒)で最適化するように調整されています.特に,動く物体の大きさは,0.1～10 メートルの範囲で Box2D は最も適切な答えを返します.言い換えれば,スープ缶からバスまでの大きさです.静的な物体なら最大でも 50 メートルまでならたいしたトラブルもないでしょう.

単位としてピクセルを使用する傾向にありますが,それは不幸にもシミュレーション結果としては精度に欠け,不可思議な動きを招くことにつながります.200 ピクセルの物体は Box2D では 45 階建てビルの高さに相当するからです.

注意:
Box2D は MKS 単位で使うよう調整されています.動く物体の大きさをおおよそ 0.1～10 メートルに留めてください.単位を変換するスケールシステムを使う必要があります.Box2D の testbed では OpenGL のビューポート変換を使用しています。絶対にビクセルを使わないように.

Box2D での描画をビルの上にある電光掲示板として考えるとよいでしょう.電光掲示板の表示はメートル単位で動きますが,シンプルなスケール変換でピクセル同等の単位に変換できます.そして,そのピクセル座標を使って,ゲームのグラフィックス上に図形を描画します.

Box2D は角度の単位としてラジアンを使用します.物体の回転角度はラジアンで保存され,無制限に増大し

ます.物体の角度が大きくなりすぎたときは b2Body::SetAngle を使って同等の小さな値に修正するよう注意してください.

1.8 ファクトリと定義

メモリ管理は Box2D API の構造において中心的な機能です.b2Body や b2Joint インスタンスを生成するときは b2World のファクトリを使用する必要があります.決して別の方法を使ってこれらのインスタンスを生成するべきではありません.

生成関数:
b2Body* b2World::CreateBody(const b2BodyDef* def)
b2Joint* b2World::CreateJoint(const b2JointDef* def)
破棄する関数:
void b2World::DestroyBody(b2Body* body)
void b2World::DestroyJoint(b2Joint* joint)
body や joint を生成したら,グローバル変数として扱うべきです.これらインスタンスはすべての body や joint のビルドに必要な情報を保持しています.この方法で構造のエラーを防ぎ,関数のパラメータを最小に抑え,適切なデフォルト値を提供し,setter や getter の数を減らせます.

フィクスチャはボディに属するため,フィクスチャは b2body のファクトリ・メソッドで生成・破棄されます.
b2Fixture* b2Body::CreateFixture(const b2FixtureDef* def)
void b2Body::DestroyFixture(b2Fixture* fixture)
シェイプと密度から直接フィクスチャを生成するためのショートカットです.
b2Fixture* b2Body::CreateFixture(const b2Shape* shape, float32 density)
ファクトリはこれら定義への参照を保持しませんので,スタックやテンポラリリソースに保持する必要があります.

1.9 ユーザーデータ

B2 fixture,b2Joint,b2Body クラスにユーザー定義の void 型ポインタをアタッチできます.Box2D データ構造を検査するときや,それらがゲームエンジン内の実体にどう関係するか知りたいときに便利です.
例えば,キャラクタのポインタを関連付けられている剛体に結びつけるのが典型的な方法です.これで循環参照を設定できます.このキャラクタがあればボディを取得できますし,ボディがあればキャラクタを取得できます.

```
GameActor* actor = GameCreateActor();
b2BodyDef bodyDef;
bodyDef.userData = actor;
actor->body = box2Dworld->CreateBody(&bodyDef);
```

ユーザーデータを必要とする状況の例です.

- 衝突の結果を使ってキャラクタにダメージを与えたいとき.
- もしキャラクタが axis-aligned box 内にいてイベントを実行するとき.
- Box2D からジョイントが破棄されると通知されてゲーム構造にアクセスするとき.

ユーザーデータを使うかは任意であり,なんでも結びつけることができますが,同一の型を配置する必要があります.例えば,あるボディにキャラクタのポインタを保存したのであれば,ほかの全てのボディでも同じ型のポインタを代入するべきです.決して,あるボディにキャラクタのポインタを保持し,ほかのボディでは違う型のポインタをセットしないでください.キャラクタのポインタのつもりで違う型をキャストすればクラッシュしてしまうからです.
ユーザーデータはデフォルトで null です.

2. Hello Box2D

2.1 ワールドの生成

すべての Box2D プログラムは b2World オブジェクトの生成から始まります.b2World はメモリ管理,オブジェクト,計算で物理の核です.スタック,ヒープ,もしくはデータセクションにワールドを割り当てることができます.

ワールドの生成は簡単です.まず,重力加速度を定義します.また,物体が静止すればスリープして良いとワールドに知らせます.ボディをスリープさせるとシミュレーションされません.

```
b2Vec2 gravity(0.0f, -10.0f);
bool doSleep = true;
```

そしてワールドのオブジェクトを生成します.ワールドはスタックに生成されるので,変数をスコープ内に保持させる必要があります.

```
b2World world(gravity, doSleep);
```

これで独自の物理世界を持てました.それではこれから必要な物を追加していきましょう.

2.2 グラウンドボックスの作成

ボディは次のステップで生成します.

1. 位置と減衰でボディを定義します.
2. World オブジェクトでボディを生成します.
3. 形状,摩擦係数,密度などをフィクスチャで定義します.
4. ボディにフィクスチャを生成します.

ステップ 1 のためにグラウンドボディを生成します.そのためにボディの定義が必要です.ボディ定義でグラウンドボディの初期位置を明確にします.

```
b2BodyDef groundBodyDef;
groundBodyDef.position.Set(0.0f, -10.0f);
```

ステップ 2 のためにボディの定義はグラウンドボディを生成するために World オブジェクトを通します.World オブジェクトはボディ定義への参照を保持しません.デフォルトではボディはスタティック(静的)です.スタティックボディは他のスタティックボディをぶつかりませんし,動くこともありません.

```
b2Body* groundBody = world.CreateBody(&groundBodyDef);
```

ステップ 3 のためにグラウンドの形状を生成します.SetAsBox 関数でグラウンドの形状を箱形,位置を親ボディの中心として作り出します.

```
b2PolygonShape groundBox;
groundBox.SetAsBox(50.0f, 10.0f);
```

SetAsBox 関数には縦横長さの半分の値を渡します.この場合,グラウンドボックスは幅 100,高さ 20 です.Box2D はこの値をメートル,キログラム,もしくは秒に変換します.したがって,100 メートルと 20 メートルとして考えることができます.Box2D は通例,物体が実在世界での物体の大きさであると最良の結果を返します.例えば,ドラム缶であれば高さ 1 メートルです.浮動小数点計算の限界から,氷河や微粒子の動きのシミュレーションには向きません.

ステップ 4 でフィクスチャを生成して,このグラウンドボディは完成です.このステップでは手順をショートカットします.フィクスチャの物理特性をデフォルトから変更する必要がないため,フィクスチャを定義することなく直接形状をボディに適用します.フィクスチャを使って物理特性をカスタマイズする方法は後ほど説明します.2 つめの引数は密度(Kg/m^2)です.スタティックボディは質量を持たないため,ここでは密度を使用しません.

```
groundBody->CreateFixture(&groundBox, 0.0f);
```

Box2D はシェイプへの参照を保持しません.そのデータを新しい b2Shape オブジェクトにコピーします.

すべてのフィクスチャは親となるボディと,静的なフィクスチャを持つ必要があります.しかし,すべての静的フィクスチャを一つの静的ボディに結びつけることができます.

2.3 動的ボディの生成

さきほどグラウンドボディを作りました.それと同じ方法を使って動的なボディを作れます.主な違いは,大きさ

以外に動的ボディの質量を定義することです.

まず,CreateBody でボディを生成します.デフォルトでボディは静的なので,b2BodyType で動的なボディを指定する必要があります.
b2BodyDef bodyDef;
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(0.0f, 4.0f);
b2Body* body = world.CreateBody(&bodyDef);

注意:
力を加えたときにボディを動かしたいのなら,ボディタイプに b2_dynamicBody を指定します.

次にフィクスチャの定義を使って多角形のシェイプを関連付け,生成します.まずボックスのシェイプを定義します.
b2PolygonShape dynamicBox;
dynamicBox.SetAsBox(1.0f, 1.0f);

次に,このボックスを使ってフィクスチャ定義を生成します.密度は 1 としました.デフォルトでは密度は 0 です.摩擦係数は 0.3 とします.
b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;
fixtureDef.density = 1.0f;
fixtureDef.friction = 0.3f;

フィクスチャ定義からフィクスチャを生成します.ここでボディの質量は自動的に計算されます.ボディには好きなだけたくさんのフィクスチャを追加できます.それらのどれもが全質量に影響します.
body->CreateFixture(&fixtureDef);

以上でおわりです.それではシミュレーションの準備をしましょう.

2.4 Box2D ワールドのシミュレーション
地面と動的な物体を初期化できました.考慮するべき事はあと一つだけです.

Box2D は integrator という数値計算アルゴリムを使っています.Integrator は離散的な時間で運動方程式を解析します.これはスクリーン上のパラパラ漫画のように昔からよくあるゲームループと同じです.そこで Box2D に時間ステップを指定する必要があります.一般的にゲーム用物理エンジンには 60Hz か 1/60 秒という時間ステップが適切です.もっと大きなステップにすることもできますが,ワールドへの定義をより慎重に設定すべきです.また,私たちはこれ以外に時間ステップを変更すべきではないと考えます.時間ステップを途中で変えると結果も異なり,デバッグを困難にします.なので,よほどな事情がない限りはここにある時間ステップを使用してください.
float32 timeStep = 1.0f / 60.0f;

また,integrator では,Box2D は制約ソルバと呼ばれる上位ビットコードを使用します.このソルバはシミュレーション中の全ての制約を解析します.一つの制約は問題なく解けます.しかし,一つの制約を解くとき,他の制約をわずかに中断させます.正確なシミュレーションのために,多数の制約を全て反復する必要があります.

制約ソルバには 2 つの状態,速度フェーズと位置フェーズがあります.速度フェーズでは,ソルバは物体の正確な移動に欠かせない力積を計算します.位置フェーズでは,ソルバは物体の重複と分離を減らすために位置を調整します.どちらのフェーズでもそれぞれの反復回数を持っています.加えて,位置フェーズはエラーが小さければ早めに反復からブレイクします.

Box2D のために提案された反復回数は,速度で 8,位置で 3 です.計算のスピードと正確さを引き替えに,この数値は自由に調節できます.少ない回数であればパフォーマンスは向上し,精度に欠けます.より大きい回数であればパフォーマンスは低下し,シミュレーション精度は向上します.この例では,たいした反復を必要としません.以下にその回数を示します.
int32 velocityIterations = 6;
int32 positionIterations = 2;

時間ステップと反復回数は完全に無関係です.反復はサブステップではありません.あるソルバ反復が時間ステップ内で全ての制約を a single pass です.

シミュレーション開始の準備が整いました.ゲーム内ではシミュレーションループはゲームループとマージされることができます.ゲームループを通してどのパスでもb2World::Stepを呼べます.ゲームのフレームレートとタイムステップに依存しているなら,ただ一回の呼び出しで普通は十分です.

このHello Worldプログラムは単純にできているので,グラフィカルな出力がありません.コードは動的ボディの位置と回転角を表示するだけです.1秒のシミュレーション時間でトータル60回の計算を行うシミュレーションを以下に示します.

```
for (int32 i = 0; i < 60; ++i) {
        world.Step(timeStep, velocityIterations, positionIterations);
        b2Vec2 position = body->GetPosition();
        float32 angle = body->GetAngle();
        printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);
}
```

出力はボックスの落下と地面への着地を示しています.出力結果は以下のようになるはずです.

```
0.00 4.00 0.00
0.00 3.99 0.00
0.00 3.98 0.00
...
0.00 1.25 0.00
0.00 1.13 0.00
0.00 1.01 0.00
```

2.5 消去

Worldがスコープの外になるか削除されれば,ボディやフィクスチャ,ジョイントに関わる全てのメモリが解放されます.これはパフォーマンス向上とプログラマーの手間を省くためです.しかし,ボディやフィクスチャ,ジョイントへのポインタを無効にするために,nullにする必要があります.

2.6 Testbed

一度このHello Worldを理解できれば,Box2Dのtestbedを見ることをお勧めします.Testbedはユニットテストのフレームワークとデモ環境です.特長は以下の通りです.

- カメラの移動とズーム
- 動的な物体をマウスでつまむ
- 拡張可能なテストセット
- テスト選択,パラメータ調整,デバッグドローへのGUI
- 一時停止とコマ送りのシミュレーション
- テキストレンダリング

--- 図 ---

Testbedはテストケースとフレームワーク自身に多くのサンプルを含んでいます.Box2Dを学ぶためにもtestbedを探検し,いじくりまわしてみましょう.

注意:
Testbedはfreeglutと GLUTを使って書かれています.TestbedはBox2Dのライブラリではありません.Box2Dはレンダリングを知り得ません.Hello worldの例で示したように,Box2Dではレンダラーを必要としません.

3. Common

3.1 概要
Common モジュールは設定,メモリ管理,ベクトル計算を含みます.

3.2 設定
ヘッダー b2Settinge.h の中身:
- **Types such as int32 and float32**
- **Constants**
- **Allocation wrappers**
- **The version number**

タイプ
構造体のサイズを簡潔に決定するために,float32,int8 などたくさんのタイプを定義しています.

定数
いくつかの定数があり,b2Settings.h に記述されています.通常はこの定数を変更する必要はありません.

浮動小数点計算を衝突とシミュレーションに使用します.ラウンドオフ・エラーのために許容値が定義されています.いくつかの許容値は絶対的であり,いくつかは相対的です.絶対許容値は MKS 単位系を使用しています.

アロケーション・ラッパー
設定ファイルは大きめのメモリ・アロケーションに b2Alloc と b2Free を定義しています.将来的にオリジナルのメモリ管理システムを構築する際これらを使用できます.

バージョン
b2Version 構造体はカレントバージョンを保持しているので,実行時に呼び出せます.

3.3 メモリ管理
Box2D の構造についての多くの決定は,メモリの高速化と効率化をベースとしています.このセクションでは,Box2D がどのようにメモリを配置しているか説明します.

Box2D では容量の小さいオブジェクト(50～300 バイト近辺)を配置する傾向にあります.小さいオブジェクトに malloc 関数や new 演算子でヒープシステムを使うことは,非効率でありフラグメンテーションの原因です.これらのオブジェクトの多くは,衝突など,ライフサイクルが短いのですが,いくつかの時間ステップで残存してしまいます.そのため,効率的にヒープメモリを提供できるアロケーターを必要とします.

Box2D での解決策は,b2BlockAllocator と呼ばれるスモール・オブジェクト・アロケーター(SOA)です.The SOA keeps a number of growable pools of varying sizes.要求がメモリを生み出したとき,SOA は要求されたサイズぴったりのメモリブロックを返します.ブロックが解放されたとき,プールに返却されます.どちらの操作も高速なため,ヒープ・トラフィックを生じません.

SOA を使うことで,ボディやフィクスチャ,ジョイントを malloc,new する必要がありません.しかし,b2World は自分で割り当てなければなりません.B2World クラスはボディやフィクスチャ,ジョイントを生成するファクトリを提供します.これで Box2D が SOA を使用し,プログラマーから恐ろしい細部を隠します.絶対に,ボディやフィクスチャ,ジョイントで delete や free を呼ばないでください.

時間ステップの実行中,Box2D はいくらかの一時メモリを必要とします.このために,ステップを通してヒープへの割り当てを避けるために b2StackAllocator を使います.スタック割り当てと相互に作用し合う必要はありませんが,知っておくことはよいことです.

3.4 計算
Box2D は単純で小さいベクトルと行列のモジュールを含んでいます.それらは Box2D と API の内部的な要求に沿うように設計されています.すべて公開されているので,アプリケーション内で自由に使用することができます.

Box2D を簡単に移植,メンテナンスできるように計算のライブラリはシンプルに保たれています.

4. Collision モジュール

4.1 概要
衝突モジュールは衝突操作のためのシェイプと機能を含んでいます.また,大規模システムの加速衝突処理のための動的ツリーと broad-phase も含んでいます.

衝突モジュールは,動的システムの外側を使用に適したものにするために設計されています.例えば,動的ツリーを物理のほか,ゲームの他の外観ために使えます.

4.2 シェイプ
シェイプが衝突の幾何学と物理シミュレーションとは独立して使用されるだろうことを物語っています.シェイプでいくつかの操作を行えます.
- シェイプと重なるポイントを調べる.
- シェイプに対してレイキャストを行う.
- シェイプの AABB を計算する.
- シェイプの質量特性を計算する.

加えて,どのシェイプもタイプメンバーと範囲を持っています.以下に説明するように,範囲はポリゴンにも適応されます.

4.3 円形
円は位置と半径を持ちます.

円は剛体で,中が空洞の円を作れません.ポリゴンシェイプを使ってラインを分割したチェーンを作れます.
```
b2CircleShape circle;
circle.m_p.Set(2.0f, 3.0f);
circle.m_radius = 0.5f;
```

4.4 ポリゴンシェイプ
ポリゴンシェイプは剛体の凸型多角形です.すべてのラインの内側の 2 点での接続がポリゴンのどの辺の接線とも交差しないとき,ポリゴンは凸型です.ポリゴンは剛体で,中空ではありません.

--- 図 ---

ポリゴンは CCW(反時計回り)で生成されます.CCW の概念は,平面内側から外側方向の Z 軸で右手座標系であるので注意する必要があります.使用している軸システムの依存から,画面上では時計回りであると分かるでしょう.

--- 図 ---

ポリゴンのメンバ関数は Public ですが,ポリゴン生成のためには初期化関数を使用してください.初期化関数は法線ベクトルを定義し,演算を有効化します.

頂点座標の配列を渡すことでポリゴンシェイプを生成できます.配列の最大サイズはデフォルトで 8 であり,b2_maxPolygonVertics によって管理されています.これはたいていのポリゴンを扱うのに十分な数です.
```
// This defines a triangle in CCW order.
b2Vec2 vertices[3];
vertices[0].Set(0.0f, 0.0f);
vertices[1].Set(1.0f, 0.0f);
vertices[2].Set(0.0f, 1.0f);
int32 count = 3;
b2PolygonShape polygon;
polygon.Set(vertices, count);
```

ポリゴンシェイプにはボックスの生成のためのカスタム初期化関数があります.
```
void SetAsBox(float32 hx, float32 hy);
void SetAsBox(float32 hx, float32 hy, const b2Vec2& center, float32 angle);
```

ポリゴンは b2Shape から範囲を継承します.範囲はポリゴンの周りにスキンを生成します.スキンは,ポリゴンをわずかに離したまま積み重ねるのに使われます.これは中心となるポリゴンをもとに連続衝突を可能にします.

--- 図 ---

ポリゴンのスキンは,ポリゴン同士を離しておくことでお互いが貫通することを防いでいます.このため,ポリゴンとの間にわずかなすき間を生じてしまいます.このすき間を埋めるために画像などを少し大きめにする必要があります.

--- 図 ---

4.5 エッジシェイプ

エッジシェイプとはラインです.静的な囲いをゲーム内に作ることができます.エッジは円や多角形といったポリゴンと衝突しますが,エッジ同士では干渉し合いません.Box2D の衝突アルゴリズムでは,2 つのシェイプのうち少なくとも一方は体積を持っている必要があります.エッジシェイプは体積を持ちません.したがって,エッジ同士の衝突は不可能です.

```
// This an edge shape.
b2Vec2 v1(0.0f, 0.0f);
b2Vec2 v2(1.0f, 0.0f);
b2EdgeShape edge;
edge.Set(v1, v2);
```

多くの場合,ゲーム画面内はいくつかのエッジシェイプで囲まれています.これは,複数のエッジが繋がった壁に沿ってポリゴンが移動するとき,予期しない人工物を生じさせます.下の図は,ボックスがエッジとエッジの接続点と衝突しています.このような衝突は,ポリゴンが衝突法線を持つ内部頂点と衝突することで生じます.

--- 図 ---

もしエッジ 1 がなければ衝突は起きません.エッジ 1 があると,内部接触はバグであるかのように見えますが,
But normally when Box2D collides two shapes, it views them in isolation.

嬉しいことにエッジシェイプは,隣接する透明な頂点を保持して望まない衝突を排除する機構を提供します.Box2D はこれらの透明頂点を使って内部衝突を防ぎます.

--- 図 ---

```
// This an edge shape with ghost vertices.
b2Vec2 v0(1.7f, 0.0f);
b2Vec2 v1(1.0f, 0.25f);
b2Vec2 v2(0.0f, 0.0f);
b2Vec2 v3(-1.7f, 0.4f);

b2EdgeShape edge;
edge.Set(v1, v2);
edge.m_hasVertex0 = true;
edge.m_hasVertex3 = true;
edge.m_vertex0 = v0;
edge.m_vertex3 = v3;
```

この方法による一般的なエッジの繋ぎ合わせは,やや無駄であり冗長です.この問題は次節のシェイプタイプに持ち越しましょう.

4.6 チェーンシェイプ

チェーン(鎖状)シェイプは複数のエッジを繋ぎ合わせるのに有効な方法です.

--- 図 ---

```
// This a chain shape with isolated vertices
b2Vec2 vs[4];
vs[0].Set(1.7f, 0.0f);
vs[1].Set(1.0f, 0.25f);
vs[2].Set(0.0f, 0.0f);
vs[3].(-1.7f, 0.4f);
b2ChainShape chain;
chain.CreateChain(vs, 4);
```

スクロールゲームでは,しばしばいくつかのチェーンをお互い繋げたいことがあります.チェーンは b2EdgeShape で透明頂点を使って繋げることができます.

```
// Install ghost vertices
chain.SetPrevVertex(b2Vec2(3.0f, 1.0f));
chain.SetNextVertex(b2Vec2(-2.0f, 0.0f));
```

ループもできます.

```
// Create a loop. The first and last vertices are connected.
b2ChainShape chain;
chain.CreateLoop(vs, 4);
```

チェーンを交差させることはサポートしていませんので,予期せぬ動作をすることがあります.システムは交差したチェーンがないことを前提としています.

--- 図 ---

チェーン上のどのエッジも子シェイプとして扱われ,インデックスを使ってアクセスできます.

```
// Visit each child edge.
for (int32 i = 0; i < chain.GetChildCount(); ++i) {
        b2EdgeShape edge;
        chain.GetChildEdge(&edge, i);
        …
}
```

4.7 シェイプ座標のテスト

シェイプで重複部分のための座標をテストできます.シェイプとワールド座標に b2Transfrom を供給します.

```
b2Transfrom transform;
transform.SetIdentity();
b2Vec2 point(5.0f, 2.0f);
bool hit = shape->TestPoint(transform, point);
```

エッジとチェーンのシェイプはループしていたとしても常に false を返します.

4.8 レイキャスト

You can cast a ray at a shape to get the point of first intersection and normal vector. No hit will register
if the ray starts inside the shape. A child index is included for chain shapes because the ray cast will only
check a single edge at a time.

```
b2Transfrom transform;
transform.SetIdentity();
b2RayCastInput input;
input.p1.Set(0.0f, 0.0f, 0.0f);
input.p2.Set(1.0f, 0.0f, 0.0f);
input.maxFraction = 1.0f;
int32 childIndex = 0;
b2RayCastOutput output;
bool hit = shape->RayCast(&output, input, transform, childIndex);
if (hit) {
        b2Vec2 hitPoint = input.p1 + output.fraction * (input.p2 – input.p1);
```

```
        …
}
```

4.9 両側関数

The Collision module contains bilateral functions that take a pair of shapes and compute some results.
These include:

- Overlap
- Contact manifolds
- Distance
- Time of impact

4.10 オーバーラップ

以下の関数を使ってシェイプのオーバーラップをテストできます.

```
b2Transform xfA = …, xfB = …;
bool overlap = b2TestOverlap(shapeA, indexA, shapeB, indexB, xfA, xfB);
```

ここで,チェーンシェイプの場合のために子インデックスを提供する必要があります

4.11 接触の多様性

Box2D はシェイプの重複のために,接触点を計算する関数を持っています.もし円と円,円と多角形の接触の場合,接触点と法線は 1 点のみ検出されます.多角形と多角形の場合,2 点検出されます.これらの座標は同じ法線ベクトルを共有するので,Box2D はそれらをさまざまな構造体に分類します.接触のソルバはスタックをより安定させるためにこの方法をとります.

--- 図 ---

通常,contact manifolds をユーザーが直接計算する必要はありませんが,この計算結果を使用することができます.

b2Manifold 構造体は法線ベクトルと 2 点接触座標に至るまで保持します.法線と座標はローカル座標で保持されています.接触ソルバの利便性のために,どの座標も法線と接線(摩擦)の力積を持っています.

b2Manifold にストアされているデータは,内部的に最大限利用されます.もしこのデータが必要なら,通常は接触法線と座標をのワールド座標を生成するために,b2WorldManifold 構造体を使うことが最良です.
b2Manifold とシェイプ変換,範囲を提供する必要があります.

```
b2WorldManifold worldManifold;
worldManifold.Initialize(&manifold, transformA, shapeA.m_radius,
                                    transformB, shapeB.m_radius);
for (int32 i = 0; i < manifold.pointCount; ++i) {
        b2Vec2 point = worldManifold.points[i];
        …
}
```

シミュレーション中,シェイプは移動して多様性が変わります.座標は加算され,削除されます.これは b2GetPointStates で知ることができます.

```
b2PointState state1[2], state2[2];
b2GetPointStates(state1, state2, &manifold1, &manifold2);
if (state1[0] == b2_removeState) {
        // process event
}
```

4.12 距離

b2Distance 関数は 2 つのシェイプ間の距離を計算します.この距離関数を b2DistanceProxy に変換するために両方のシェイプを必要とします.距離関数を繰り返し呼び出すのにホットスタートするためのメモリを必要とします.詳細は b2Distance.h にあります.

--- 図 ---

4.13 Time of impact

もし 2 つの物体が高速で移動していたら,シングル時間ステップではトンネルのように通り抜けてしまいます.

--- 図 ---

2 つの移動物体が衝突するとき,b2TimeOfImpact はその時間を計算するために使用されます.これは撃力時間(TOI)と呼ばれます.b2TimeOfImpact の主な目的はトンネル効果の回避です.特に,静的で水平な幾何学の外側からトンネル効果を防ぐために設計されています.

この関数が物体の回転と移動の情報を提供しますが,もし回転角度が十分大きいと,関数は失敗する可能性があります.しかし,関数は依然として非オーバーラップされた時間を返し,すべての衝突移動を取り込みます.

The time of impact function identities an initial separating axis and ensures the shapes do not cross on
that axis.これは最終位置でクリアされて衝突し損なうでしょう.この方法ではいくつかの衝突で失敗しますが,非常に高速でトンネル回避には十分です.

--- 図 ---
--- 図 ---

回転の大きさを制限することは困難です.小さめの回転では衝突の失敗を引き起こす可能性があります.通常,ゲームでそのような失敗をするべきではありません.

この関数は(b2DistanceProxy に変換するために)2 つのシェイプと b2Sweep 構造体を必要とします.sweep 構造体はシェイプの最初と最後の変換を定義します.

シェイプキャストに変換するために固定角を使用できます.この場合,TOI 関数はどのような衝突でも失敗しません.

4.14 動的ツリー

b2DynamicTree クラスは,多数のシェイプを効率的にまとめるために Box2D が使用します.このクラスはシェイプについて知り得ません.その代わりにユーザーデータと共に AABB を操作します.

動的ツリーは階層的な AABB ツリー構造です.ツリーのどのノードも 2 つの子を持ちます.リーフ(葉)ノードはシングルユーザー AABB です.退化した入力の場合,ツリーのバランスを取るためローテーションを使用します.

ツリー構造は効率的なレイキャストとリージョンクエリが可能です.例えば,画面内に数千のシェイプを持っているとします.どのシェイプにもレイキャストによる強引な方法で,画面に対してレイキャストすることができます.これは,シェイプが散らばって進めなくなるので非効率です.代わりに,動的ツリーを保ち,ツリーに対してレイキャストを実行できます.大量のシェイプをスキップすることを通して,レイが横断します.

リージョンクエリは,すべてのリーフノードのクエリ AABB と重複する AABB を見つけるためにツリーを使います.これは多くのシェイプをスキップするので,強引な方法よりも高速です.

--- 図 ---
--- 図 ---

通常は,動的ツリーを直接使用するべきではありません.それよりレイキャストとリージョンクエリを活用してください.

4.15 ブロードフェーズ

物理ステップ内の衝突処理はナロー(狭い)フェーズとブロード(広い)フェーズという 2 つの状態に分けられます.ナローフェーズでは,2 物体の衝突点を計算します.N 個の物体があると考えてください.撃力を使って,

N*N/2 個のペアについてナローフェーズを実行する必要があります.

b2BroadPhase クラスは動的ツリーを使用することで,この負荷を削減します.これはナローフェーズを呼び出す回数を軽減させる非常に有効な方法です.

通常,ブロードフェーズと直接情報を交換する必要はありません.その代わり,Box2D は内部的にブロードフェーズを生成,管理します.また,b2BroadPhase は Box2D のシミュレーションループを考慮して設計されているため,他の用途に使用するには向いていません.

5. Dynamics(力学)モジュール

5.1 概要

Dynamics モジュールは Box2D の中でも最も複雑な部分であり,最も興味深い部分でもあるでしょう.
Dynamics モジュールは Common,Collision モジュールの先頭にあり,今すぐにでも理解する必要があります.

Dynamics モジュールの内容:

- **shape fixture class**
- **rigid body class**
- **contact class**
- **joint classes**
- **world class**
- **listener classes**

これらクラスの多くには相互依存が存在し,あるクラスを他のクラスを参照することなしに理解することは困難です.以下のチャプターでは,まだ解説や理解のできていないクラスをいくつか参照することになるでしょう.
そのために,このチャプターをざっと読んでから後のチャプターを熟読するといいでしょう.

Dynamics モジュールは以下のチャプターの内容を含んでいます.

6. フィクスチャ

6.1 概要

シェイプはボディについて知らないということ,物理シミュレーションの独立で使用されることを思い出してください.なので,Box2D はシェイプとボディを関連付けるために b2Fixture を提供します.

- シングルシェイプ
- ブロードフェーズ・プロキシ
- 密度,摩擦,反発
- 衝突フィルタリング・フラグ
- 親のボディに戻るポインタ
- ユーザーデータ
- センサー・フラグ

これらは以下のセクションで説明します.

6.2 フィクスチャの生成

フィクスチャはその定義を初期化し,親となるボディに渡すことで生成されます.

```
b2FixtureDef fixtureDef;
fixtureDef.shape = &myShape;
fixtureDef.density = 1.0f;
b2Fixture* myFixture = myBody->CreateFixture(&fixtureDef);
```

これでフィクスチャを生成し,ボディに結び付けます.親のボディが破棄されるとフィクスチャも自動的に破棄されるので,フィクスチャへのポインタを保持する必要はありません.1 つのボディに対して複数のフィクスチャを関連付けられます.

破壊できるモデルを作るなら,親ボディにあるフィクスチャを削除することができます. そうでなければ,フィクスチャのみを残すことができ,Otherwise you can just leave the fixture alone and let the body destruction take care of destroying the attached fixtures.

```
myBody->DestroyFixture(myFixture);
```

密度

フィクスチャの密度は親ボディの質量特性の計算に使われます.その値は 0 以上です.スタックの安定性が向上するので,通常は同じ密度を全てのフィクスチャに使用すべきです.

密度をセットしてもボディの質量に反映はされません.ResetMassData を呼ぶ必要があります.

```
fixture->SetDensity(5.0f);
body->ResetMassData();
```

摩擦

摩擦は物体をお互い現実的に滑らせるのに使用されます.Box2D は静摩擦,動摩擦をサポートしますが,どちらも同じ値を使用します.摩擦は Box2D 内で正確に計算され,摩擦力は垂直抗力に比例します(これをクーロンの摩擦法則と呼びます).摩擦のパラメータは通常,0 から 1 の間の正の数です.パラメータが 0 のとき摩擦力は無く,1 のとき強力な摩擦力を生み出します.摩擦力が 2 物体で計算されるとき,Box2D は親となる 2 つのフィクスチャの摩擦パラメータを結合させなければなりません.これは幾何学的には次のような意味です.

```
float32 friction;
friction = sqrtf(shape1->friction * shape2->friction);
```

したがって,一方のフィクスチャの摩擦パラメータが 0 なら,接触時どちらにも摩擦力は発生しません.

反発

反発は物体が跳ね上がるのに使われます.反発のパラメータ(反発係数)は通常,0 から 1 の間です.床にボールが落ちてくるところを考えてみてください.0 の反発係数は弾まず,非弾性衝突と呼ばれます.反発係数が 1 の場合は,反発前後のボールの速さは変化しないことを意味します.これを完全弾性衝突と呼びます.反発係数は以下の式で結合されます.

```
float32 restitution;
restitution = b2Max(shape1->restitution, shape2->restitution);
```

フィクスチャは,ゲーム中の物体間で衝突を避けるように衝突フィルタの情報を運びます.

シェイプが複数接触を展開するとき,反発はおおよそ計算されます.これは Box2D が反復ソルバを使用するためです.Box2D はまた,衝突速度が小さいとき,非弾性衝突を使用します.これは揺らぎを防ぐためです.

フィルタリング
衝突フィルタは物体同士が衝突するのを防ぐことができます.例えば,自転車に乗るキャラクタを作りたいとします.自転車やキャラクタと地面は接触させたいでしょうが,自転車とキャラクタは接触させたくないでしょう(なぜなら,自転車とキャラクタは重なり合っているからです).Box2D ではこのような衝突回避をカテゴリとグループを使うことで実現できます.

Box2D は 16 の衝突カテゴリを提供します.全てのフィクスチャに対して,どのカテゴリに所属させるか指定できます.また,他のどのカテゴリと接触できるかを指定することもできます.例えば,マルチプレイヤーのゲームで,プレーヤー同士は接触させたくないし,モンスター同士も接触させたくない,という場合でも,プレーヤーとモンスターは接触させることができます.これは次のようなマスクで実現できます.

```
playerFixtureDef.filter.categoryBits = 0x0002;
monsterFixtureDef.filter.categoryBits = 0x0004;
playerFixtureDef.filter.maskBits = 0x0004;
monsterFixtureDef.filter.maskBits = 0x0002;
```

次に,接触が起こる条件を示します.

```
uint16 catA = fixtureA.filter.categoryBits;
uint16 maskA = fixtureA.filter.maskBits;
uint16 catB = fixtureB.filter.categoryBits;
uint16 maskB = fixtureB.filter.maskBits;
if ((catA & maskB) != 0 && (catB & maskA) != 0) {
        // fixtures can collide
}
```

接触グループには整数のグループ・インデックスが必要です.常に接触する(正のインデックス),常に接触しない(負のインデックス) インデックスを持つすべての同じグループのフィクスチャを持てます.グループ・インデックスは通常,自転車の一部分のような,なんらかの方法で関係する物に使われます.以下の例では,フィクスチャ 1 とフィクスチャ 2 は常に接触し,フィクスチャ 3 とフィクスチャ 4 は決して接触しません.

```
fixture1Def.filter.groupIndex = 2;
fixture2Def.filter.groupIndex = 2;
fixture3Def.filter.groupIndex = -8;
fixture4Def.filter.groupIndex = -8;
```

異なるグループ・インデックスにあるフィクスチャの接触は,カテゴリとマスクによってフィルタリングされます.

以下のリストは,Box2D にあるその他のフィルタです.

- スタティックボディにあるフィクスチャは,ダイナミックボディにしか接触できない.
- kinematic ボディにあるフィクスチャは,ダイナミックボディにしか接触できない.
- 同じボディにあるフィクスチャはお互いに接触しない.
- ボディ上のジョイントで結びつけられたフィクスチャとの接触の可否を任意にセットできます.

時々,フィクスチャを生成したあとになって,接触フィルタを変更する必要があるかもしれません.
b2Fixture::GetFilterData や b2Fixture::SetFilterData を使うことで,そのフィクスチャに対して b2Filiter 構造体の取得・セットができます.ただし,フィルタリングの変更は次の時間ステップまで反映されないことに注意してください(World クラスを参照).

6.3 センサー

時々ゲームで,2 物体が衝突することなく重なり合っているか知りたいということがあるでしょう.それはセンサーを使えば可能です.センサーは物体に力を及ぼすことなく衝突判定ができます.

センサーはどんなフィクスチャでも使用できます.センサーは恐らく静的か動的.一つの Body に複数のフィクスチャが存在しているかも知れないこと,どんなセンサーと固体フィクスチャの組み合わせもできることを忘れないでください.

センサーは接触点を生成できません.センサーの状態にアクセスするには次の 2 つの方法があります.

1. b2Contact::IsTouching
2. b2ContactListener::BeginContact と EndContact

7. ボディ

7.1 概要

ボディは位置と速度を持っていて,力,トルク,力積を与えることができます.ボディには,スタティック(static),キネマティック(kinematic),ダイナミック(dynamic)の 3 つの状態があり,以下に定義を示します.

b2_staticBody
スタティックボディはシミュレーション下で動かず,無限大の質量を持っているかのように振る舞います.内部的には,Box2D は質量とその逆数に 0 を保持します.スタティックボディはプログラマーが手動で動かせます.速度は 0 です.他のどのスタティックボディやキネマティックボディとも接触できません.

b2_kinematicBody
キネマティックボディはシミュレーション下で速度に従って動き,力に反応しません.プログラマーによって手動で動かすこともできますが,通常は速度の設定によって動きます.無限大の質量を持っているかのように振る舞いますが,Box2D は質量とその逆数に 0 を保持します.また,その他のスタティックボディやキネマティックボディと接触しません.

b2_dynamicBody
ダイナミックボディは完全にシミュレーションされます.プログラマーによって手動で動かすことができますが,通常は力によって動きます.ダイナミックボディは他の全てのボディタイプと接触します.また,0 でない有限の質量を持ちます.もし質量を 0 にしようとすれば,自動的に質量 1 キログラムに設定されます.

ボディはフィクスチャの骨格であり,フィクスチャをワールド中に動かします.ボディは Box2D では常に剛体です.これは同じボディに関連付けられた 2 つのフィクスチャは,互いに相互移動しないことを意味します.

フィクスチャは衝突幾何学と密度を持ちます.通常,ボディはフィクスチャから自身の質量特性を取得します.しかし,ボディが生成された後,その質量特性をオーバーライドできます.これを以下で説明します.

通常,生成したすべてのボディへのポインタを保持するかと思います.これは対応するキャラクタなどの画像の位置をセットするために,ボディの位置を取得する方法です.また,ボディが必要なくなって削除するときのためにも,ボディへのポインタを保持するべきです.

7.2 ボディの定義

ボディの生成をする前に,b2BodyDef でボディの定義をしないといけません.ボディ定義は,ボディの生成と初期化に必要なデータを保持しています.

Box2D はそのボディ定義のデータをコピーします.ボディ定義へのポインタは保持されません.これは複数のボディを生成するときに,同じボディ定義を再利用できるようにするためです.

それでは,ボディ定義の主要なメンバを見てみましょう.

ボディタイプ
このチャプターの最初で説明したように,3 つのボディタイプ(スタティック,キネマティック,ダイナミック)があります.後でボディタイプを変更することはおすすめできないので,生成時にボディタイプをセットするべきです.
bodyDef.type = b2_dynamicBody;

必ずボディタイプをセットしてください.

位置と角度
ボディ定義でボディが生成される位置を初期化できます.これはワールドの原点に生成してから指定の位置まで動かすよりはるかに優れています.

注意:
原点にボディを生成してから任意の位置に動かす方法で初期化しないでください.原点に複数のボディを生成したときにパフォーマンスが劣ります.

ボディには 2 つの重要なポイントがあります.1 つ目はボディの原点です.フィクスチャとジョイントはボディの原点を基準に結びつけられています.2 つ目は質量の中心です.質量中心は関連するシェイプの質量分布,もしくは b2MassData で明示的に決められます.Box2D の内部計算ではこの質量中心を多用します.例えば,

b2Body では質量中心の直線速度を保持します.

ボディ定義時,どこに質量中心があるのか知らないでしょう.したがって,プログラマーが明示的にボディの原点位置を指定する必要があります.また,質量中心に影響しないボディの角度をラジアンで指定できます.後にボディの質量特性を変更すると,質量中心はボディ上を移動しますが,関連するシェイプとジョイントは動かず,座標原点も変化しません.
bodyDef.position.Set(0.0f, 2.0f); // the body's origin position.
bodyDef.angle = 0.25f * b2_pi; // the body's angle in radians.

減衰
減衰はボディの速度を落とすために使います.摩擦は接触時のみ作用するので,減衰と摩擦は違います.減衰を摩擦で置き換えることはできず,両者が同時に作用することもありません.

減衰のパラメータは 0 から無限大で,0 は減衰無し,無限大は完全減衰を意味します.通常は 0 から 0.1 の間で使用すると良いでしょう.linearDamping は浮いたように見えるので,通常は使用しないでください.
bodyDef.linearDamping = 0.0f;
bodyDef.angularDamping = 0.01f;

減衰は安定とパフォーマンスのために近似されます.小さい減衰値では,減衰効果は時間ステップにほとんど依存します.大きい値では,時間ステップによって変動します.固定時間ステップ(推奨)を使用しているなら,これらは関係しません.

重力のスケール(縮小比)
1 つのボディにスケールされた重力を使えます.重力の増加は安定性の低下につながるので注意してください.
// Set the gravity scale to zero so this body will float
bodyDef.gravityScale = 0.0f;

スリープ・パラメータ
スリープってどういう意味でしょうか.ボディのシミュレーションは負荷がかかるので,より効率的に行う必要があります.ボディが静止したら,その計算を中断することができるのです.

Box2D はボディ(もしくはボディグループ)が静止すると,ボディを CPU にほとんど負荷をかけないスリープ状態にします.スリープしたボディを awake させるか他のボディがぶつかれば,シミュレーションを再開します.ボディは関連付けられたジョイントかコンタクトが削除されても計算を再開します.また,手動で再開させることもできます.

ボディ定義でスリープできるかどうか,ボディをスリープ状態で生成するかどうか設定できます.
bodyDef.allowSleep = true;
bodyDef.awake = true;

固定ローテーション
キャラクタのような剛体を固定ローテーションとして必要なときがあります.そのようなボディは力がかかっても回転することはありません.次の設定で固定ローテーションとしてセットできます.
bodyDef.fixedRotation = true;

このフラグが回転を無効にし,それ自身を 0 にするために逆にします.

ブレット(弾丸)
ゲームシミュレーションでは普通,フレームレートに応じて一連のイメージを形成します.これを離散シミュレーションと呼びます.離散シミュレーションでは,物体は一回のステップで大きく動きます.もし物理エンジンがこの動きを考慮しなければ,物体は意図せずに互いに通り抜けてしまうでしょう.これをトンネル効果と呼びます.

デフォルトでは,Box2D は連続衝突検知(CCD)でダイナミックボディがスタティックボディを通り抜けることを防いでいます.これはボディを現在の位置から次の位置へ一括で移動させています.エンジンは移動と衝突時間(TOI)を計算する間,次の衝突点を求めます.ボディは自身の最初の衝突時間へと移動し,時間ステップの残りの間は一時停止します.

CCD は通常,ダイナミックボディ間では使用されません.これは合理的にパフォーマンスを保つためです.いくつかのゲームの場面によっては,ダイナミックボディに CCD を使用する必要があります.例えば,積み重ねられたブロックに弾丸を撃ち込みたいときです.CCD 無しでは,弾丸はブロックを通り抜けてしまうかも知れませ

ん.

Box2D において高速で動く物体は,ブレットとしてラベルすることができます.ブレットはスタティックボディとダイナミックボディのどちらも CCD で処理されます.ゲームの設計を基にして,どの物体をブレットとすべきか決めてください.どれをブレットにするか決まれば,次の設定を使用してください.
bodyDef.bullet = true;

ブレットのフラグはダイナミックボディのみに影響します.

Box2D は逐次,連続接触を処理するので,ブレットは高速の物体を見落とすかも知れません.

アクティベーション
ボディが生成されても,運動や接触に感知して欲しくないことがあるかも知れません.この状態は,ボディが他のボディによってシミュレーションを再開しない点,ボディのフィクスチャがブロードフェーズに配置されない点を除けば,スリープと同じです.これはボディが衝突やレイキャスト等に関与しないことを意味しています.

非アクティブの状態でボディを生成し,後でアクティブにできます.
bodyDef.active = true;

ジョイントが非アクティブなボディに接続しているかも知れません.これらのジョイントもシミュレーションされません.ボディをアクティブにするときは,それらのジョイントがねじれていないか注意してください.

ユーザーデータ
ユーザーデータは void 型のポインタです.アプリケーションのオブジェクトとボディを結びつけるホックの役目をします.すべてのボディのユーザーデータは,一貫性を持った同じオブジェクトの型を使用すべきです.
b2BodyDef bodyDef;
bodyDef.userData = &myActor;

7.3 ボディの生成

ボディは World クラスのボディ・ファクトリで生成,破棄されます.ワールドが効率的なメモリ配置でボディを生成させ,ボディにワールド・データ構造体を追加します.

ボディは質量特性に応じてダイナミックかスタティックになります.どちらのボディタイプも同じ生成関数と破棄関数を使用します.
b2Body* dynamicBody = myWorld->CreateBody(&bodyDef);
... do stuff ...
myWorld->DestroyBody(dynamicBody);
dynamicBody = NULL;

注意:
ボディの生成に new や malloc を絶対に使わないでください.ワールドがボディの存在を知り得ませんし,適切に初期化されません.

スタティックボディは他のボディの影響下では動きません.スタティックボディを手動で移動させることはできますが,2 つ以上のスタティックボディでダイナミックボディを挟んでつぶさないように気を付ける必要があります.スタティックボディを動かすと摩擦が正しく働きません.スタティックボディはスタティックボディもしくはキネマティックボディと絶対に接触しません.いくつかのシェイプとスタティックボディを関連付けることは,互いにシェイプが 1 つのいくつかのスタティックボディを生成することより早いです.内部的には,Box2D はスタティックボディの質量とその逆数を 0 にセットします.それで計算はうまくいくので,ほとんどのアルゴリズムはスタティックボディを特別なケースとして処理する必要が無くなります.

Box2D はボディ定義へのポインタや定義が保持するどのデータも保持しません(ユーザーデータへのポインタを除く).なのでボディ定義の一時変数を作って,同じボディ定義で何回も再利用できます.

b2World オブジェクトを削除しても,後ですべて破棄できるのなら,ボディの破棄を避けることができます.保持したそれらのポインタに null を参照することを忘れないように留意する必要があります.

ボディを破棄したとき,結びつけられているフィクスチャやジョイントも自動的に破棄されます.これはシェイプやジョイントのポインタを管理方法について重要な結果です.

7.4 ボディの使用

ボディ生成後,ボディに対してできる操作はたくさんあります.それらは質量特性のセット,位置と速度の取得,力を加える,座標とベクトルの変換を含みます.

質量データ
どのボディも質量(スカラー),質量中心(2 次ベクトル),回転慣性(スカラー)を持っています.スタティックボディでは,質量と回転慣性が 0 です.ボディが固定ローテーションであれば,回転慣性は 0 です.

通常,ボディの質量特性はフィクスチャがボディにセットされたときに自動的に計算されます.また,実行時にボディの質量を適応することもできます.これはゲームで質量が変化する特別な設定のある場面で使います.
void SetMassData(const b2MassData* data);

ボディに直接質量をセットしたら,フィクスチャの影響によって自然な質量に復帰することを願うでしょう.以下を実行してください.
void ResetMassData();

ボディの質量データは以下の関数で利用可能です.
float32 GetMass() const;
float32 GetInertia() const;
const b2Vec2& GetLocalCenter() const;
void GetMassData(b2MassData* data) const;

状態の情報
ボディの状態には多くの側面があります.以下の関数でこれらの状態に効率的にアクセスできます.
void SetType(b2BodyType type);
b2BodyType GetType();
void SetBullet(bool flag);
bool IsBullet() const;
void SetSleepingAllowed(bool flag);
bool IsSleepingAllowed() const;
void SetAwake(bool flag);
bool IsAwake() const;
void SetActive(bool flag);
bool IsActive() const;
void SetFixedRotation(bool flag);
bool IsFixedRotation() const;

位置と速度
ボディの位置と速度を取得できます.これはゲームのキャラクタを描画するときは一般的です.また,あまり一般的ではありませんが,動きを計算するために Box2D を通常に使用するのであれば,位置をセットすることも可能です.
bool SetTransform(const b2Vec2& position, float32 angle);
const b2Transform& GetTransform() const;
const b2Vec2& GetPosition() const;
float32 GetAngle() const;

ローカル座標系,ワールド座標系での質量中心の座標を取得できます.Box2D の内部的なシミュレーションの多くは質量中心を使用します.しかし,通常はこれらを取得するべきではありません.その代わりにボディ変換が使えます.例えば,正方形のボディを持っているとします.ボディの原点は正方形の角にあり,一方,質量中心は正方形の中心にあります.
const b2Vec2& GetWorldCenter() const;
const b2Vec2& GetLocalCenter() const;

速度と角速度を取得できます.速度は質量中心に作用します.したがって,もし質量特性が変化すれば速度も変化するでしょう.

8. ジョイント

8.1 概要

ジョイントはボディを互いに,もしくはワールドに束縛するのに使われます.ゲームでの典型的な例として,ラグドール(人形),シーソー,滑車が挙げられます.ジョイントは様々な方法で組み合わせ,おもしろい動きをさせることができます.

ジョイントによっては,動ける範囲を限定することができます.また,所定の速さで与えられた力・トルクを超えるまでジョイントを駆動させるモーターを提供します.

ジョイントモーターは様々な方法で使用できます.モーターで望んだ位置と実際の位置の距離差に比例したジョイントの速度を与えることで,位置をコントロールできます.また,以下の方法を使って,モーターでジョイントの摩擦を計算できます.ジョイントの速度を 0 にし,小さな,しかし重要な最大の力・トルクを与えます.するとモーターは,荷重が超過するまでジョイントを動かせません.

8.2 ジョイントの定義

どのジョイントタイプも b2JointDef に由来する定義を持ちます.すべてのジョイントは 2 つの異なるボディを結合させます.そのうち 1 つはスタティックかもしれません.スタティックとキネマティック,スタティックかキネマティックのそれぞれボディ間のジョイントも可能ですが,効果はありませんし,多くの処理時間を必要とします.

どのジョイントタイプにもユーザーデータを明確にし,取り付けられたボディ同士が互いに接触しないようにフラグを使用してください.これは実際には標準の動作で,つながったボディへの衝突を可能にするために collideConnected Boolean を設定する必要があります.

多くのジョイント定義はボディに関連付けられた幾何学のデータを必要とします.しばしばジョイントはアンカーポイントによって定義されます.それらは取り付けられたボディの固定ポイントです.Box2D は,ローカル座標系での座標を明確にするためにこれらのポイントを必要とします.この方法でジョイントは,ボディがジョイントの制約に違反して変換されても位置を把握できます.加えて,いくつかのジョイント定義はボディのデフォルトの相対的な角度を知る必要があります.これは正しく回転を制限するために必要です.

幾何学データの初期化は割とうんざりするので,その手間を省くために多くのジョイントはカレントボディ変換に使用する初期化関数を持っています.しかし,これらの初期化関数は通常プロトタイピングで使用されます.プロダクションコードは直接,幾何学を定義するべきです.これはジョイントの動きをより強健にします.

ジョイント定義の残りの説明はジョイントタイプに依存します.これからそれらを見ていきましょう.

8.3 ジョイント・ファクトリ

ジョイントはワールドのファクトリ・メソッドで生成・破棄されます.これに関しては何度も言及したように,以下の注意が必要です.

注意:
ジョイントをスタック・ヒープ上のどちらにも new や malloc で生成しないでください.ジョイントやボディの生成・破棄は対応する b2World クラスの生成関数・破棄関数を使用してください.

以下に回転ジョイントの生成から破棄までのライフサイクルの例を示します.

```
b2RevoluteJointDef jointDef;
jointDef.bodyA = myBodyA;
jointDef.bodyB = myBodyB;
jointDef.anchorPoint = myBodyA->GetCenterPosition();
b2RevoluteJoint* joint = (b2RevoluteJoint*)myWorld->CreateJoint(&jointDef);
... do stuff ...
myWorld->DestroyJoint(joint);
joint = NULL;
```

破棄された後,ポインタに null を参照しておけば完璧です.もし破棄されたポインタを再利用しようとすれば,この規則ではプログラムがクラッシュします.

ジョイントのライフサイクルは単純ではありません.以下を心に留めておきましょう.

注意:
取り付けられたボディが破棄されるとジョイントも破棄されます.

この注意は常に必要ではありません.プログラマーは,ジョイントが常にボディより先に破棄されるようにするでしょう.このケースでは,リスナークラスを実装する必要はありません.詳細は,暗黙的デストラクタのセクションを見てください.

8.4 ジョイントを使う

多くのシミュレーションではジョイントを生成し,破棄されるませ再びアクセスしません.しかし,ジョイントにはシミュレーションをより豊かにするための有益なデータがたくさん含まれています.

まず始めに,ボディ,アンカーポイント,ユーザーデータの取得です.

```
b2Body* GetBodyA();
b2Body* GetBodyB();
b2Vec2 GetAnchorA();
b2Vec2 GetAnchorB();
void* GetUserData();
```

すべてのジョイントは力とトルクからの反作用を持ちます.この反作用はアンカーポイントでボディ 2 に与えられます.ジョイントの破壊もしくは他のゲームイベントのトリガとして反作用を使えます.これらの関数は多くの計算で使用されるので,結果が必要なとき以外は不用意に呼ばないでください.

```
b2Vec2 GetReactionForce();
float32 GetReactionTorque();
```

8.5 ディスタンス・ジョイント

最もシンプルなジョイントの 1 つが,2 つのボディの 2 点間の距離が一定を保つディスタンス・ジョイントです.ディスタンス・ジョイントを定義するとき,2 つのボディは適切な位置にいるべきです.そして,2 つのアンカーポイントをワールド座標上で明確にします.最初のアンカーポイントはボディ 1 につなげ,次のアンカーポイントはボディ 2 につなげます.これらのポイントは制限する距離の長さを示唆します.

--- 図 ---

以下はディスタンス・ジョイント定義の例です.この例では,ボディが衝突できるようにしました.

```
b2DistanceJointDef jointDef;
jointDef.Initialize(myBodyA, myBodyB, worldAnchorOnBodyA, worldAnchorOnBodyB);
jointDef.collideConnected = true;
```

ディスタンス・ジョイントは,バネとダンパが接続されたように柔らかくもできます.この動きは,testBed 内の例「Web」を見てください.

柔らかさは定義にある 2 つの定数,振動数と減衰率で実現しています.振動数はギターの弦のような音響器の振動数として考えてください.この振動数の単位は Hz です.一般的には振動数は,時間ステップ振動数の半分より少なくするべきです.もし時間ステップに 60Hz を使っていれば,ディスタンス・ジョイントの振動数は 30Hz より少なくするべきです.これはナイキスト振動数に関係しています

減衰率は無次元の単位で,通常は 0 から 1 の間です.1 より大きくすることもできます.1 では,減衰は臨界します(振動しません).

```
jointDef.frequencyHz = 4.0f;
jointDef.dampingRatio = 0.5f;
```

8.6 回転ジョイント

回転ジョイントは 2 つのボディに,しばしばヒンジポイントと呼ばれる,同じアンカーポイントを使わせます.2 つのボディの相対的な回転なので,回転ジョイントは 1 自由度です.これをジョイントアングルと呼びます.

--- 図 ---

回転を明確にするために,2 つのボディと 1 つのアンカーポイントが必要です.初期化関数はそれらのボディ

がすでに適切な位置にあると想定しています.

この例では,2 つのボディが最初の質量中心の回転ジョイントでつながっています.
b2RevoluteJointDef jointDef;
jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter());

ボディ B が回転中心に対して反時計回り(CCW)に回転すれば,回転ジョイントの角度は正です.Box2D のすべての角度のように,回転角度はラジアンです.ジョイントが Initialize()で生成されると,2 つのボディの現在の角度に関わらず回転角度は 0 です.

ジョイントの角度をコントロールしたいこともあるかも知れません.回転ジョイントは任意でジョイントとモーター,ジョイントもしくはモーターをシミュレートできます.

ジョイント制限はジョイント角度を最小値と最大値の間で制限します.この制限は角度をそこに留めておくだけのトルクをジョイントに与えます.制限の範囲は 0 を含みます.そうでなければ,シミュレーション開始時にジョイントは傾いてしまいます.

ジョイントモーターではジョイントの速さ(角度の時間微分)を指定できます.速さは正の数か負の数です.
モーターは無限大の力を持ちますが,普通は望ましくありません.ここで永遠の疑問を思い出しましょう.

“とてつもない力が不動の物体にかかったら,何が起きるか?”

この疑問は面白くも何ともありません.では,ジョイントモーターに回転とは逆の最大トルクを与えてみてください.ジョイントモーターは,かかるトルクが最大値を超えるまでは与えられた速さを保ちます.最大値を超えると,ジョイントの速さは低下し逆に回転し始めます.

ジョイントモーターを使ってジョイントの摩擦を計算できます.ジョイントの速さを 0 にして,トルクを少しずつ有効な最大値までかけます.モーターは回転しないように頑張りますが,あるトルクに達すると屈してしまいます.

以下に回転ジョイント定義の修正を載せます.ここではジョイントは制限を持ち,モーターは動きません.モーターはジョイント摩擦を計算する設定です.
b2RevoluteJointDef jointDef;
jointDef.Initialize(bodyA, bodyB, myBodyA->GetWorldCenter());
jointDef.lowerAngle = -0.5f * b2_pi; // -90 degrees
jointDef.upperAngle = 0.25f * b2_pi; // 45 degrees
jointDef.enableLimit = true;
jointDef.maxMotorTorque = 10.0f;
jointDef.motorSpeed = 0.0f;
jointDef.enableMotor = true;

回転ジョイントの角度,速さ,モーターのトルクを取得できます.
float32 GetJointAngle() const;
float32 GetJointSpeed() const;
float32 GetMotorTorque() const;

ジョイントモーターはいくつかの興味深い能力を持っています.時間ステップごとにジョイントの速さを更新することで,サインカーブのように,もしくは実装したいあらゆる動きで好きなように前後にジョイント動かせます.
... Game Loop Begin ...
myJoint->SetMotorSpeed(cosf(0.5f * time));
... Game Loop End ...

次のように,ジョイントモーターで指定した角度を追跡することもできます.
... Game Loop Begin ...
float32 angleError = myJoint->GetJointAngle() - angleTarget;
float32 gain = 0.1f;
myJoint->SetMotorSpeed(-gain * angleError);
... Game Loop End ...

一般にジョイントが不安定になるので,ゲインは大きすぎるべきではありません.

8.7 平行ジョイント

平行ジョイントは,ある軸に沿った 2 ボディの相対的な平行移動です.平行ジョイントは相対的に回転させないので,1 自由度です.

--- 図 ---

平行ジョイントの定義は,Translation の代わりに Angle を使えば,回転ジョイントでの説明と似ています.
b2PrismaticJointDef jointDef;
b2Vec2 worldAxis(1.0f, 0.0f);
jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter(), worldAxis);
jointDef.lowerTranslation = -5.0f;
jointDef.upperTranslation = 2.5f;
jointDef.enableLimit = true;
jointDef.maxMotorForce = 1.0f;
jointDef.motorSpeed = 0.0f;
jointDef.enableMotor = true;

回転ジョイントは画面に垂直な軸を持っていますが,平行ジョイントは画面に平行な軸を持ちます.この軸は 2 ボディに固定され,ボディの動きに従います.

回転ジョイントのように Initialize()で生成すると,平行ジョイントの平行移動は 0 です.なので,0 は制限の上限と下限の間にあるか,確認してください.

平行ジョイントの使用も回転ジョイントと似ています.以下は関連のあるメンバ関数です.
float32 GetJointTranslation() const;
float32 GetJointSpeed() const;
float32 GetMotorForce() const;
void SetMotorSpeed(float32 speed);
void SetMotorForce(float32 force);

8.8 滑車ジョイント

滑車は理想的な滑車として使用されます.滑車は 2 ボディと地面とお互いをつなぎます.一方のボディが上がれば,他方は下がります.滑車のロープの全長は初期設定に従って一定です.
length1 + length2 == constant

滑車をシミュレートするのに,比率を与えることもできます.これで滑車の伸びる方の側が他方より早くなります.けれども,一方を制約する力は他方より小さいです.機械的なテコを作るために,これを使えます.
length1 + ratio * length2 == constant

例えば,もしその比が 2 なら,length1 は length2 の 2 倍に変わります.また,body1 に取り付けられたロープにかかる力は body2 に取り付けられたロープの制約力の半分です.

--- 図 ---

滑車は一方をめいっぱいまで伸ばすと,他方のロープ長が 0 になるのでやっかいです.長さが 0 になると,制限方程式が特異解となります(良くないことです).これを避けるためにシェイプのの衝突を設定するべきです.

以下は滑車の定義例です.
b2Vec2 anchor1 = myBody1->GetWorldCenter();
b2Vec2 anchor2 = myBody2->GetWorldCenter();
b2Vec2 groundAnchor1(p1.x, p1.y + 10.0f);
b2Vec2 groundAnchor2(p2.x, p2.y + 12.0f);
float32 ratio = 1.0f;
b2PulleyJointDef jointDef;
jointDef.Initialize(myBody1, myBody2, groundAnchor1, groundAnchor2, anchor1,
anchor2, ratio);

滑車ジョイントの長さを取得します.
float32 GetLengthA() const;
float32 GetLengthB() const;

8.9 ギアジョイント
もし精巧な機械機構を作りたいのなら,ギアを使いたいでしょう.原理的には,Box2D でギア歯をシェイプの組み合わせで作れば,ギアを実現可能です.しかしこれは非常に非効率的で,さすがにうんざりします.スムーズにギアを噛み合わせるために,ギアの並びに気を付けなくてはなりません.Box2D にはギアジョイントという,ギアを生成するシンプルな方法があります.

--- 図 ---

ギアジョイントは回転ジョイントと平行ジョイント同士でしか接続できません.

滑車比のように,ギア比を設定できます.しかし,この場合ギア比は負の数です.また,一方が回転ジョイントで他方が平行ジョイントのとき,ギア比は長さの単位もしくは長さ以上の単位となることに注意してください.
coordinate1 + ratio * coordinate2 == constant

ギアジョイントの例です.ボディ myBodyA と myBodyB は,同じボディでない限り,2 つのジョイントからのボディです.
```
b2GearJointDef jointDef;
jointDef.bodyA = myBodyA;
jointDef.bodyB = myBodyB;
jointDef.joint1 = myRevoluteJoint;
jointDef.joint2 = myPrismaticJoint;
jointDef.ratio = 2.0f * b2_pi / myLength;
```

ギアジョイントは他の 2 ジョイントに依存します.これは脆い状況を生み出します.もしこれらのジョイントが削除されたら何が起こるだろう?

注意:
常に,回転・平行ジョイントより先にギアジョイントを削除してください.そうしなければ,ギアジョイント上の参照元のないジョイントのポインタのせいでクラッシュします.また,含まれるどのボディを削除する前に,ギアジョイントを削除するべきです.

8.10 マウスジョイント
マウスジョイントは,testbed でボディをマウスで操作するときに使われています.ボディ上の座標をマウスポインタの方へ向かって動かす傾向にあります.回転に制限はありません.

マウスジョイントの定義はターゲットとなる座標,最大力,振動数,減衰率を持っています.ターゲットとなる座標は,最初はボディのアンカーポイントと一致します.最大力は,重なったダイナミックボディが相互に作用するとき,激しい反応を避けるのに使われます.

多くのプログラマーが,ゲームでマウスジョイントを使えるようにしてきました.しばしば正確な位置判定と高速なレスポンスを望みますが,この場合マウスジョイントはうまく動きません.代わりにキネマティックボディの使用を検討してください.

8.11 ホイールジョイント
ホイールジョイントはボディ B 上の点をボディ A 上の直線に制限します.ホイールジョイントはサスペンション・スプリングでもあります.詳細は b2WheelJoint.h と Car.h を参照してください.

--- 図 ---

8.12 結合ジョイント
結合ジョイントは 2 ボディ間の相対的な動きの全てを制限します.どのような動きかは,testbed の Cantilever.h を見てください.

結合ジョイントは破壊可能な構造物の定義に使われます.しかし,Box2D は反復ソルバなので,ジョイントは少し柔らかいです.なので,結合ジョイントで繋げられたボディの束縛は柔軟です.

代わりに,破壊可能な構造物は,複数のフィクスチャを持った 1 つのボディで作るほうが良いでしょう.ボディが壊れたときフィクスチャを消し,それらを新しいボディに再生成できます.詳しくは testbed の例を参照してく

ださい.

8.13 ロープジョイント
ロープジョイントは,2 ボディ間の最大距離を制限します.これは高負荷下であっても,ボディ同士のつながりが伸びすぎてしまわないようにするのに便利です.詳細は b2RopeJoint.h と RopeJoint.h を見てください.

8.14 摩擦ジョイント
摩擦ジョイントは包括的な摩擦で使用されます.このジョイントは 2 次元上の移動摩擦と,角摩擦を提供します.詳しくは b2FrictionJoint.h と ApplyForce.h を見てください.

9. コンタクト

9.1 概略
コンタクトは 2 つのフィクスチャ間の接触を管理するオブジェクトです.もしそのフィクスチャがチェーンシェイプのような子を持つのなら,関連するどの子にもコンタクトが存在します.Contact クラスから派生するコンタクトには,異なる種類のフィクスチャを管理するために,異なる種類のコンタクトが存在します.例えば,多角形と多角形の接触を管理するコンタクトクラス,円と円の接触を管理するクラスです.

以下にコンタクトに関係する用語があります.

コンタクト・ポイント
コンタクト・ポイントは 2 つの形状の接触点です.Box2D は接触を a small number of points で近似します.

垂直接触
垂直接触は,あるシェイプの座標から他の座標への単位ベクトルです.規則で,法線の座標はフィクスチャ A からフィクスチャ B の方向です.

複数接触
2 つの凸型物体が接触した場合,接触点は 2 点までとなり,それより多い接触点は存在しません.どちらの点も同じ法線を持ち,連続した接触の領域の近似として複数接触に分類されます.

9.2 Contact クラス
すでに述べたように,Contact クラスのオブジェクトは Box2D によって生成・破棄されます.ユーザーがクラスのオブジェクトを生成することはできませんが,オブジェクトにアクセスすることはできます.

9.3 接触の取得
接触の取得は複数の方法で実現できます.ワールドのコンタクトに直接アクセスする方法と,ボディ構造体を使う方法です.コンタクト・リスナーを実装する方法もあります.

ワールドの全ての接触を走査できます.
```
for( b2Contact* c = myWorld->GetContactList(); c; c = c->GetNext() ) {
	// process c
}
```

ボディから走査もできます.コンタクトエッジ構造体を使うことで,グラフに保持されます.
```
for ( b2ContactEdge* ce = myBody->GetContactList(); ce; ce = ce->next ) {
	b2Contact* c = ce->contact;
	// process c
}
```

以下の章で説明するように,コンタクト・リスナーを使って接触を取得する方法もあります.

注意:
b2World や b2Body に頼って接触を取得する方法は,時間ステップの中盤で起こる瞬間的な接触で失敗する可能性があります.b2ContactListener を使って正確な結果を取得してください.

9.4 コンタクト・リスナー
ContactListener の実装で接触が通知されます.コンタクト・リスナーは begin,end,pre-solve,post-solve の各イベントをサポートしています.

```
class MyContactListener : public b2ContactListener {
public:
	void BeginContact(b2Contact* contact) {
		/* handle begin event */
	}
	void EndContact(b2Contact* contact) {
		/* handle end event */
```

```
	}
	void PreSolve(b2Contact* contact, const b2Manifold* oldManifold) {
		/* handle pre-solve event */
	}
	void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse) {
		/* handle post-solve event */
	}
};
```

注意:
b2ContactListener に通知するポインタへの参照を保持しないでください.その代わりにバッファに接触点のデータをコピーさせます.その方法の例を下方に示します.

実行時にリスナーのインスタンスを生成し,b2World::SetContactListener で登録します.World インスタンスが存在する限りリスナーも有効です.

Begin Contact イベント
このイベントは 2 物体が重なり合ったときにセンサーの有り無しに関わらず呼ばれます.time step 内でのみ発生します.

End Contact イベント
このイベント 2 つの物体が重なり続けるとセンサーの有り無しに関わらず呼ばれます.これは body が破棄されたときに呼ばれるので,time step の外で発生します.

Pre-Solve イベント
このイベントは衝突が検出された後,接触計算が行われる前に呼ばれます.これはあなたに現在の設定を基礎とした接触を無効にする機会を与えます.例えば,このコールバックと b2Contact::SetEnabled(false)で a one-sided platform を実装できます.この接触は衝突が進行している間は毎回有効となるので,各 time step ごとに接触を無効にする必要があります.連続的な衝突を検出するために,pre-solve イベントは一接触あたりの time step ごとの数回呼ばれます.

```
void PreSolve(b2Contact* contact, const b2Manifold* oldManifold) {
	b2WorldManifold worldManifold;
	contact->GetWorldManifold(&worldManifold);
	if (worldManifold.normal.y < -0.5f) {
		contact->SetEnabled(false);
	}
}
```

また,このイベントはポイントの状態,接触の接近速度を決定するのに最適です.

```
void PreSolve(b2Contact* contact, const b2Manifold* oldManifold) {
	b2WorldManifold worldManifold;
	contact->GetWorldManifold(&worldManifold);
	b2PointState state1[2], state2[2];
	b2GetPointStates(state1, state2, oldManifold, contact->GetManifold());
	if (state2[0] == b2_addState) {
		const b2Body* bodyA = contact->GetFixtureA()->GetBody();
		const b2Body* bodyB = contact->GetFixtureB()->GetBody();
		b2Vec2 point = worldManifold.points[0];
		b2Vec2 vA = bodyA->GetLinearVelocityFromWorldPoint(point);
		b2Vec2 vB = bodyB->GetLinearVelocityFromWorldPoint(point);
		float32 approachVelocity = b2Dot(vB - vA, worldManifold.normal);
		if (approachVelocity > 1.0f) {
			MyPlayCollisionSound();
		}
	}
}
```

Post-Solve イベント
このイベントでは接触による撃力を取得できます.もし撃力を必要としないなら,pre-solve を実装するだけでい

いでしょう.

このイベントはコールバック内で物理条件を変えるためにゲームで実装される傾向にあります.例えば,ダメージを与える接触や関係するキャラクタの rigid body を破壊しようするときです.しかし Box2D は,現在処理しているオブジェクトをユーザーが削除したがために null 参照を引き起こすので,コールバック内で物理世界を変更することを認めません.

接触が現在も進行中の接触点を処理する推奨方法は,time step の後でそれらの処理を行うためにすべての接触データをバッファに入れてしまうことです.time step の後ただちにその接触点を処理する必要があります.一方,他のクライアントコードが might alter the physics world, invalidating the contact buffer.接触情報をバッファに待避させてから物理条件を変更できますが,そのバッファ内に null 参照を残さないように注意する必要があります.testbed には接触点処理の安全な方法があります.

CollisionProcessing テストから引用した以下のコードは,コンタクトバッファを処理するときに参照元のない body をどう扱うか示しています.コメントよく読め.このコードはすべての接触点が b2ContactPoint array m_points の中のバッファに入っていると想定しています.

```
// We are going to destroy some bodies according to contact
// points. We must buffer the bodies that should be destroyed
// because they may belong to multiple contact points.
const int32 k_maxNuke = 6;
b2Body* nuke[k_maxNuke];
int32 nukeCount = 0;
// Traverse the contact buffer. Destroy bodies that
// are touching heavier bodies.
for (int32 i = 0; i < m_pointCount; ++i) {
	ContactPoint* point = m_points + i;
	b2Body* body1 = point->shape1->GetBody();
	b2Body* body2 = point->shape2->GetBody();
	float32 mass1 = body1->GetMass();
	float32 mass2 = body2->GetMass();
	if (mass1 > 0.0f && mass2 > 0.0f) {
		if (mass2 > mass1) {
			nuke[nukeCount++] = body1;
		}
		else {
			nuke[nukeCount++] = body2;
		}
		if (nukeCount == k_maxNuke) {
			break;
		}
	}
}
// Sort the nuke array to group duplicates.
std::sort(nuke, nuke + nukeCount);
// Destroy the bodies, skipping duplicates.
int32 i = 0;
while (i < nukeCount) {
	b2Body* b = nuke[i++];
	while (i < nukeCount && nuke[i] == b)	{
		++i;
	}
	m_world->DestroyBody(b);
}
```

9.5 コンタクトフィルタ

しばしばゲームで全ての物体を衝突させたくないときがあります.例えば,あるキャラクタのみが通り抜けられるドアを作ろうとしたときです.これはいくつかの相互作用を取り除けるので,コンタクトフィルタと呼ばれます.

b2ContactFilter クラスを使って好きなコンタクトフィルタを作成できます.このクラスは 2 つの b2Shape ポイン

タを受け取るために,ShouldCollide 関数を実装するよう要求します.

ShouldCollide のデフォルト実装は,チャプター 6 で定義された b2FilterData を使用します.

```
bool b2ContactFilter::ShouldCollide(b2Fixture* fixtureA, b2Fixture* fixtureB) {
	const b2Filter& filterA = fixtureA->GetFilterData();
	const b2Filter& filterB = fixtureB->GetFilterData();
	if (filterA.groupIndex == filterB.groupIndex && filterA.groupIndex != 0) {
		return filterA.groupIndex > 0;
	}
	bool collide = (filterA.maskBits & filterB.categoryBits) != 0 && (filterA.categoryBits &
filterB.maskBits) != 0;
	return collide;
}
```

実行時にコンタクトフィルタのインスタンスを生成し,b2World::SetContactFilter でセットできます.フィルタは World が存在している間は有効です.

```
MyContactFilter filter;
world->SetContactFilter(&filter);
// filter remains in scope …
```

10. World クラス

10.1 概要
b2World クラスはボディとジョイントを含んでいます.このクラスはシミュレーションのすべての側面を管理し,(AABB クエリやレイキャストのような)非同期クエリを提供します.Box2D におけるプログラマーの興味の大部分は b2World オブジェクトでしょう.

10.2 ワールドの生成と破棄
ワールドの生成はかなり単純です.重力加速度ベクトルとボディがスリープして良いかどうかを与えればいいだけです.ワールドの生成と破棄には,通常 new か delete を使用します.
```
b2World* myWorld = new b2World(gravity, doSleep);
... do stuff ...
delete myWorld;
```

10.3 ワールドを使う
World クラスはボディとジョイントの生成と破棄のためのファクトリを持っています.これらのファクトリについては,後半にあるボディとジョイントのセクションで説明します.ここでは b2World での他の相互作用について,いくつか説明していきます.

10.4 シミュレーション
ワールドクラスはシミュレーションの運用に使用されます.時間ステップと速度,位置の反復回数を与えます.例えば,
```
float32 timeStep = 1.0f / 60.f;
int32 velocityIterations = 10;
int32 positionIterations = 8;
myWorld->Step(timeStep, velocityIterations, positionIterations);
```

時間ステップ後,ボディとジョイントに関連する情報を取得できます.最も欲しい情報は,キャラクタを描画するための位置情報だと思います.時間ステップはゲームループ内のどこででも実行できますが,物の順番に気を付けるべきです.例えば,もしそのフレーム内で新しいボディの衝突結果を取得したいなら,時間ステップの前にボディを生成しなければなりません.

HelloWorld チュートリアルで前述したように,固定時間ステップを使用するべきです.大きなステップを使うことでパフォーマンスは向上し,フレームレートは低下します.特別な理由がなければ,1/30 秒以下のステップを使用すべきです.1/60 秒で高品質なシミュレーションが実現します.

反復回数は何回,制限ソルバがワールド内の接触とジョイントをスウィープするかを管理します.多い反復回数はより良いシミュレーション結果を生みます.かといって,大きな反復回数と引き替えに時間ステップを小さくしないでください.30Hz で 20 反復回数よりも,60Hz で 10 反復回数の方がはるかに良いです.

ステップ実行後,b2World::ClearForces で,ボディに与えてきたあらゆる力をクリアするべきです.これで同じ力の場で複数のサブステップを取ることができます.
```
myWorld->ClearForces();
```

10.5 ワールドを探検
ワールドはボディと接触,ジョイントの入れ物です.それらのリストをイテレートできます.例えば,以下のコードだとクラッシュします.
```
for( b2Body* b = myWorld->GetBodyList(); b; b = b->GetNext() ) {
        GameActor* myActor = (GameActor*)b->GetUserData();
        if( myActor->IsDead() ) {
                myWorld->DestroyBody(b); // ERROR: now GetNext returns garbage.
        }
}
```

ボディが削除されるまでは全てがうまくいきます.いったんボディが削除されると,それが保持する次のポインタが無効となります.ですので,b2Body::GetNext()は不要な値を返します.この解決策は,ボディを削除する前

に,次のポインタをコピーすることです.

```
b2Body* node = myWorld->GetBodyList();
while ( node ) {
	b2Body* b = node;
	node = node->GetNext();
	GameActor* myActor = (GameActor*)b->GetUserData();
	if( myActor->IsDead() ) {
		myWorld->DestroyBody(b);
	}
}
```

これで安全に現在のボディを削除できます.しかし,複数のボディを削除したいこともあるでしょう.この場合には非常に注意を要します.この解決方法はアプリケーション固有で,便利なことを除けば,以下に 1 つの方法を提示します.

```
b2Body* node = myWorld->GetBodyList();
while( node ) {
	b2Body* b = node;
	node = node->GetNext();
	GameActor* myActor = (GameActor*)b->GetUserData();
	if( myActor->IsDead() ) {
		bool otherBodiesDestroyed = GameCrazyBodyDestroyer(b);
		if (otherBodiesDestroyed) {
			node = myWorld->GetBodyList();
		}
	}
}
```

このコードを実現するには,言うまでもなく GameCrazyBodyDestroyer が削除したことを正確に返してもらう必要がある.

10.6 AABB クエリ

ある範囲内にあるすべてのシェイプを知りたいということがあるでしょう.b2World クラスはそのために,ブロードフェーズデータ構造を使用した高速 Log(N)法を持っています.ワールド座標系の AABB と,b2QueryCallback の実装を与えます.ワールドは,AABB が要求された AABB と重なる全てのフィクスチャとクラスを呼びます.クエリを続けるには true を返し,そうでなければ false を返します.例えば,以下のコードは指定された AABB と潜在的に交差する全てのフィクスチャを見つけ,関連付けられた全てのボディの計算を再開します.

```
class MyQueryCallback : public b2QueryCallback {
public:
	bool ReportFixture( b2Fixture* fixture ) {
		b2Body* body = fixture->GetBody();
		body->WakeUp();
		// Return true to continue the query.
		return true;
	}
};
...
MyQueryCallback callback;
b2AABB aabb;
aabb.lowerBound.Set(-1.0f, -1.0f);
aabb.upperBound.Set(1.0f, 1.0f);
myWorld->Query(&callback, aabb);
```

このコールバックがいつ呼ばれるか,予測することはできません.

10.7 レイキャスト

視線チェック,銃を撃つときなどにレイキャストを使えます.コールバック･クラスの実装と,始点と終点を与えることでレイキャストを実行できます.ワールドクラスは,光線に当った全てのフィクスチャのクラスを呼びます.コールバックにフィクスチャと光線の交差点,単位法線ベクトル,光線に沿った断片的な距離を提供します.このコールバックがいつ呼ばれるか,予測することはできません.

fraction を返すことで,レイキャストを継続できます.fraction を 0 で返すことは,レイキャストがそこで終端であることを示します.fraction が 1 とは,レイキャストがどこにも当らないかのうように続行することを意味します.引数の fraction をそのまま返すと,光線は現在の交点からクリップされます.それで,どんなシェイプにも,全てのシェイプにレイキャストできます.適切な fraction を返すことで一番近いシェイプにレイキャストもできます.

フィクスチャをフィルタするために,fraction に-1 の値を返せます.このとき,レイキャストはそのフィクスチャが存在しないかのように進みます.

以下はその例です.

```
// This class captures the closest hit shape.
class MyRayCastCallback : public b2RayCastCallback {
public:
	MyRayCastCallback() {
		m_fixture = NULL;
	}
	float32 ReportFixture(b2Fixture* fixture, const b2Vec2& point,
				const b2Vec2& normal, float32 fraction) {
		m_fixture = fixture;
		m_point = point;
		m_normal = normal;
		m_fraction = fraction;
		return fraction;
	}
	b2Fixture* m_fixture;
	b2Vec2 m_point;
	b2Vec2 m_normal;
	float32 m_fraction;
};
MyRayCastCallback callback;
b2Vec2 point1(-1.0f, 0.0f);
b2Vec2 point2(3.0f, 1.0f);
myWorld->RayCast(&callback, point1, point2);
```

注意:
丸め誤差によるエラーのため,スタティックボディの狭いすき間では,レイキャストは不安定になります.これが受け入れがたい場合は,それらのすき間を大きくしてください.

```
void SetLinearVelocity(const b2Vec2& v);
b2Vec2 GetLinearVelocity() const;
void SetAngularVelocity(float32 omega);
float32 GetAngularVelocity() const;
```

10.8 力と力積

ボディに力,トルク,力積を加えられます.力や力積を加えるとき,作用点のワールド座標が必要です.トルクの場合,しばしば質量中心です.

```
void ApplyForce(const b2Vec2& force, const b2Vec2& point);
void ApplyTorque(float32 torque);
void ApplyLinearImpulse(const b2Vec2& impulse, const b2Vec2& point);
void ApplyAngularImpulse(float32 impulse);
```

力,トルク,力積を加えるとボディがシミュレーションを再開します.時々これは望ましくありません.
例えば,一定の力を加え,パフォーマンス向上のためにボディをスリープさせたいとします.この場合,以下のコードで実現できます.

```
if (myBody->IsAwake() == true) {
	myBody->ApplyForce( myForce, myPoint );
}
```

10.9 座標変換

ボディ・クラスは,ローカルとワールド間の座標とベクトルを変換するためのユーティリティ関数をいくつか持っ

ています.もしこれがどういう意味か分からなければ,Jim Van Verth,Lars Bishop 共著「Essential Mathematics for Games and Interactive Applications」(ゲームと双方向アプリケーションのための基礎数学)を読んでください.これらの関数は効率的です.

```
b2Vec2 GetWorldPoint(const b2Vec2& localPoint);
b2Vec2 GetWorldVector(const b2Vec2& localVector);
b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);
b2Vec2 GetLocalVector(const b2Vec2& worldVector);
```

10.10 リスト

ボディのフィクスチャをイテレートできます.もしフィクスチャのユーザーデータを取得したいときに,これは便利です.

```
for( b2Fixture* f = body->GetFixtureList(); f; f = f->GetNext() ) {
        MyFixtureData* data = (MyFixtureData*)f->GetUserData();
        ... do something with data ...
}
```

ジョイントのリストについても,同じくイテレートできます.

ボディはまた,関係する接触のリストも提供します.それを使って現在の接触の情報を取得できます.この接触リストは,過去の時間ステップに存在した全ての接触を含んでいるわけではないので注意してください.

11. 未解決の問題

11.1 暗黙的な破棄

Box2D は参照カウントを使用しません.もしボディを破棄すると,それは本当に破棄されます.破棄されたボディのポインタへアクセスしたときの動作は定義されていません.言い換えれば,プログラムはクラッシュしやすくなるということです.これらの問題の対処を助けるために,デバッグ・ビルド・メモリ・マネージャーは破棄されたエントリを FDFDFDFD で埋めます.場合によっては,これで問題を見つけるのがより簡単になります.

Box2D のエントリを破棄するとき,破棄されるオブジェクトへの参照をすべて取り除くのはプログラマーの仕事です.これは 1 つのエントリに対して,参照が 1 つしかなければ容易です.複数の参照があれば,raw ポインタをラップするハンドル・クラスの実装を検討してください.

Box2D の使用でしばしば,大量のボディやシェイプ,ジョイントを生成・破棄するでしょう.それらのエントリの管理は,多少 Box2D によって自動化されています.ボディを破棄すれば,関連する全てのシェイプとジョイントは自動的に破棄されます.これを暗黙的な破棄と呼びます.

ボディを破棄するとき,それに関連する全てのシェイプ,ジョイント,コンタクトも破棄されます.これを暗黙的な破棄と呼びます.これらのジョイントと,もしくはコンタクトの 1 つにつながれたどのボディもアウェイクします.この処理は通常は便利ですが,以下の決定的な問題を認知するべきです.

注意:
ディを破棄すれば,関連する全てのシェイプとジョイントは自動的に破棄されます.これらのシェイプとジョイントへのどのポインタも無効にするべきです.さもなくば,後でそのシェイプやジョイントにアクセス,破棄しようとすると,プログラムはクラッシュします.

ポインタの無効化に一役買うために,Box2D はワールド・オブジェクトに実装,提供できる b2DestructionListener と呼ばれるリスナー・クラスを提供します.もしジョイントが暗黙的に破棄されようとすると,それをワールド・オブジェクトが知らせます.

ジョイントやフィクスチャが明示的に破棄されたときは,リスナーからの通知はありません.この場合,所有権はクリアされ,その場所に必要なクリーンアップを実行できます.望めば,集権化されたクリーンアップ・コードを維持するためにオリジナルの b2DestructionListener を実装できます.

暗黙的な破棄は多くの場合で非常に便利ですが,プログラムをバラバラにもできます.コードのどこかに,シェイプやジョイントへのポインタを保持するでしょう.関係するボディが破棄されれば,それらのポインタは参照元がなくなります.結びつけられたボディ管理のために,無関係のコードの一部によってジョイントがたびたび生成されると,この状況は最悪です.例えば,testbed は b2MouseJoint をボディの相互的な操作のために生成します.暗黙的に破棄されたとき,それを知らせようと Box2D はコールバック機構を提供します.このときが参照元のないポインタを無効にする機会です.このコールバック機構はこのマニュアルの後半に説明してあります.

関係するボディは破棄されたので,シェイプやジョイントが暗黙的に破棄されたと b2World が知らせるように,b2DestructionListener を実装できます.これは参照元のないポインタにアクセスするのを防ぎます.

```
class MyDestructionListener : public b2DestructionListener {
	void SayGoodbye( b2Joint* joint ) {
		// remove all references to joint.
	}
};
```

そのとき,破棄リスナーのインスタンスをワールド・オブジェクトに登録できます.それはワールドの初期化時にするべきです.

```
myWorld->SetListener(myDestructionListener);
```

12. デバッグ・ドロー

物理世界の詳細を描くために,b2DebugDraw クラスを実装できます.以下は使用できる機能です.

- シェイプの外形
- ジョイントの接続能力
- ブロードフェーズ AABB
- 質量中心

--- 図 ---

これは,直接データにアクセスするよりは好まれるエントリの描画方法です.その理由は,必要なデータ変化の多くが内部的だからです.

testbed は,デバッグドローとコンタクト・リスナーで物理のエントリを描くので,接触点の描画だけでなく,デバッグドローの実装方法の根本的な例としての働きをします.

13. 制限

Box2D は効率的に剛体をシミュレーションするために,いくつか近似を使います.それがいくつかの制限を発生させます.

現段階での制限:

- とても軽いボディの上に非常に重いボディを重ねると動作は安定しません.重量比を 10:1 までにすると安定します.
- もし軽いボディで重いボディを支えているのなら,ジョイントでつなげられた鎖状のボディは伸びます.例えば,破壊するボールが軽重量の鎖状ボディとつながると不安定です.重量比を 10:1 までにすると安定します.
- There is typically around 0.5cm of slop in shape versus shape collision.
- 連続接触はジョイントを扱いません.オブジェクトが初めて動くとき,ジョイントが伸びるかもしれません.

14. 参考文献

Erin Catto’s GDC Tutorials: http://code.google.com/p/box2d/downloads/list
Collision Detection in Interactive 3D Environments, Gino van den Bergen, 2004
Real-Time Collision Detection, Christer Ericson, 2005