Box2D v2.3.0 User Manual

*Copyright © 2007-2013 Erin Catto*

Chapter 1 Introduction 2

Chapter 2 Hello Box2D 7

Chapter 3 Common 13

Chapter 4 Collision Module 15

Chapter 5 Dynamics Module 22

Chapter 6 Fixtures 23

Chapter 7 Bodies 26

Chapter 8 Joints 33

Chapter 9 Contacts 42

Chapter 10 World Class 49

Chapter 11 Loose Ends 55

Chapter 12 Debug Drawing 57

Chapter 13 Limitations 59

Chapter 14 References 60

* **Introduction**
* **About**

Box2Dは、ゲーム用の2D剛体シミュレーションライブラリです。プログラマーはゲームでそれを使用して、オブジェクトを現実的な方法で動かし、ゲームの世界をよりインタラクティブにします。ゲームエンジンの観点から見ると、物理エンジンは手続き型アニメーションの単なるシステムです。

Box2DはポータブルC ++で書かれています。エンジンで定義されるほとんどのタイプは、b2 prefixで始まります。これでゲームエンジンと名前が衝突するのを防ぐのに十分です。

* **Prerequisites**

このマニュアルでは、質量、力、トルク、衝撃などの基本的な物理概念に精通していることを前提としています。そうでない場合は、ggrks。

Box2Dは、Game Developer Conferenceの物理チュートリアルの一部として作成されました。これらのチュートリアルは、box2d.orgのダウンロードセクションから入手できます。

Box2DはC ++で記述されているため、C ++プログラミングの経験がある必要があります。 Box2Dは簡単ではありません！コンパイル、リンク、およびデバッグに慣れている必要があります。

* **About this Manual**

このマニュアルは、Box2D APIの大部分をカバーしています。ただし、すべての側面が網羅されているわけではありません。詳細については、Box2Dに含まれているテストベッドをご覧ください。また、Box2DコードベースにはDoxygen用にフォーマットされたコメントがあるため、ハイパーリンクされたAPIドキュメントを簡単に作成できます。このマニュアルは、新しいリリースでのみ更新されます。ソース管理のバージョンは古くなっている可能性があります。

* **Core Concepts**

Box2Dは、いくつかの基本的な概念とオブジェクトで動作します。ここでこれらのオブジェクトを簡単に定義します。詳細については、このドキュメントの後半で説明します。

* ***shape***

形状は、円や多角形などの2D幾何学的オブジェクトです。

* ***rigid body***

チャンク上の物質の2ビット間の距離が一定であるほど強い物質の塊。ダイヤモンドのように硬いです。次の説明では、剛体と交換可能にボディを使用します。

* ***fixture***

フィクスチャは、シェイプをボディにバインドし、密度、摩擦、反発などの材料特性を追加します。フィクスチャは、他のシェイプと衝突できるように、コリジョンシステム（ブロードフェーズ）にシェイプを配置します。

* ***constraint***

constraintは、身体から自由度を削除する物理的な接続です。 2Dボディには3つの自由度があります（2つの並進座標と1つの回転座標）。体を取り、それを（振り子のように）壁に固定すると、体は壁に拘束されます。この時点で、ボディはピンの周りでのみ回転できるため、拘束により2自由度が削除されました。

* ***contact constraint***

剛体の貫通を防ぎ、摩擦と反発をシミュレートするために設計された特別な制約。接触制約を作成しません。それらはBox2Dによって自動的に作成されます。

* ***joint***

これは、2つ以上のボディを一緒に保持するために使用される制約です。 Box2Dは、回転、角柱、距離など、いくつかのジョイントタイプをサポートしています。一部のジョイントには制限とモーターがあります。

* ***joint limit***

関節の制限は、関節の可動範囲を制限します。たとえば、人間の肘では特定の範囲の角度しか使用できません。

* ***joint motor***

ジョイントモーターは、ジョイントの自由度に従って接続されたボディの動きを駆動します。たとえば、モーターを使用して肘の回転を駆動できます。

* ***world***

物理世界は、相互作用するボディ、フィクスチャ、および制約のコレクションです。 Box2Dは複数の世界の作成をサポートしますが、これは通常、必要または望ましくありません。

* ***solver***

物理学の世界にはソルバーがあり、これを使用して時間を進め、接触とジョイントの制約を解決します。 Box2Dソルバーは、N回の順序で動作する高性能の反復ソルバーです。ここで、Nは制約の数です。

* ***continuous collision***

ソルバーは、離散時間ステップを使用してボディを時間的に進めます。介入なしでは、これはトンネリングにつながる可能性があります。



Box2Dには、トンネリングを処理する専用のアルゴリズムが含まれています。最初に、衝突アルゴリズムは2つの物体の動きを補間して、最初の衝撃（TOI）を見つけることができます。第二に、ボディを最初のインパクトに移動してから衝突を解決するサブステップソルバーがあります。

* **Modules**

Box2Dは、Common、Collision、Dynamicsの3つのモジュールで構成されています。 Commonモジュールには、割り当て、計算、および設定のためのコードがあります。衝突モジュールは、形状、ブロードフェーズ、および衝突機能/クエリを定義します。最後に、Dynamicsモジュールは、シミュレーションワールド、ボディ、フィクスチャ、およびジョイントを提供します。



* **Units**

Box2Dは浮動小数点数で動作し、Box2Dのパフォーマンスを向上させるには許容値を使用する必要があります。これらの許容値は、メートルキロ秒（MKS）単位で適切に機能するように調整されています。特に、Box2Dは、0.1から10メートルの間で移動する図形でうまく機能するように調整されています。そのため、スープ缶～バスのサイズのオブジェクトがうまく機能するはずです。静的形状の長さは問題なく最大50メートルです。

2D物理エンジンであるため、ピクセルを単位として使用するのは魅力的です。残念ながら、これは貧弱なシミュレーション及び恐らく奇妙な振る舞いにつながります。長さ200ピクセルのオブジェクトは、45階建ての建物のサイズとしてBox2Dに表示されます。

**Caution**

Box2D is tuned for MKS units. Keep the size of moving objects roughly between 0.1 and 10 meters. You'll need to use some scaling system when you render your environment and actors. The Box2D testbed does this by using an OpenGL viewport transform. DO NOT USE PIXELS.

Box2Dのボディは、アートワークを添付するビルボードの移動と考えるのが最適です。ビルボードはメートルの単位システムで移動する場合がありますが、単純なスケーリング係数でピクセル座標に変換できます。その後、それらのピクセル座標を使用してスプライトなどを配置できます。また、座標軸を反転させることもできます。

Box2Dは角度にラジアンを使用します。体の回転はラジアン単位で保存され、無制限に成長する場合があります。角度の大きさが大きくなりすぎる場合は、体の角度を正規化することを検討してください（b2Body :: SetAngleを使用してください）

**Caution**

Box2D uses radians, not degrees.

* **Factories and Definitions**

高速メモリ管理(まず覚えるべき?)は、Box2D APIの設計において中心的な役割を果たします。したがって、b2Bodyまたはb2Jointを作成するときは、b2Worldでファクトリー関数を呼び出す必要があります。これらのタイプを別の方法で割り当てようとしないでください。

まず、creation関数があります：

b2Body\* b2World::CreateBody(const b2BodyDef\* def)

b2Joint\* b2World::CreateJoint(const b2JointDef\* def)

そして、対応するdestruction関数があります：

void b2World::DestroyBody(b2Body\* body)

void b2World::DestroyJoint(b2Joint\* joint)

ボディまたはジョイントを作成するとき、定義を提供する必要があります。これらの定義には、身体または関節を構築するために必要なすべての情報が含まれています。このアプローチを使用することにより、構築エラーを防ぎ、関数パラメーターの数を小さく保ち、適切なデフォルトを提供し、アクセサーの数を減らすことができます。

fixtures (shapes)はボディの親である必要があるため、b2Bodyのファクトリメソッドを使用して作成および破棄されます。

b2Fixture\* b2Body::CreateFixture(const b2FixtureDef\* def)

void b2Body::DestroyFixture(b2Fixture\* fixture)

形状と密度から直接フィクスチャを作成するショートカットもあります。

b2Fixture\* b2Body::CreateFixture(const b2Shape\* shape, float32 density)

Factories は定義への参照を保持しません。したがって、スタック上に定義を作成し、一時リソースに保持することができます。

* **Hello Box2D**

Box2Dの配布フォルダには、Hello Worldプロジェクトがあります。このプログラムは、大きなグラウンドボックスと小さなダイナミックボックスを作成します。このコードにはグラフィックは含まれていません。表示されるのは、時間とともにボックスの位置をコンソールにテキスト出力することだけです。これは、Box2Dを起動して実行する方法の良い例です。

* **Creating a World**

すべてのBox2Dプログラムは、b2Worldオブジェクトの作成から始まります。 b2Worldは、メモリ、オブジェクト、およびシミュレーションを管理する物理ハブです。物理世界をスタック、ヒープ、またはデータセクションに割り当てることができます。

Box2Dワールドを簡単に作成できます。まず、重力ベクトルを定義します。

b2Vec2 gravity(0.0f, -10.0f);

次に、ワールドオブジェクトを作成します。スタック上にワールドを作成しているため、ワールドはスコープ内にある必要があります。

b2World world(gravity);

So now we have our physics world, let's start adding some stuff to it.

* **Creating a Ground Box**

ボディは、次の手順を使用して構築されます。

* position位置、減衰などを含むボディを定義します
* worldワールドオブジェクトを使用してボディを作成します。
* shape形状、摩擦、密度などを使用してフィクスチャを定義します。
* body身体に器具を作成します。

For step 1 we create the ground body. For this we need a body definition. With the body definition we specify the initial position of the ground body.

b2BodyDef groundBodyDef;

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

For step 2 the body definition is passed to the world object to create the ground body. The world object does not keep a reference to the body definition. Bodies are static by default. Static bodies don't collide with other static bodies and are immovable.

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

For step 3 we create a ground polygon. We use the SetAsBox shortcut to form the ground polygon into a box shape, with the box centered on the origin of the parent body.

b2PolygonShape groundBox;

groundBox.SetAsBox(50.0f, 10.0f);

The SetAsBox function takes the **half**-**width** and **half**-**height** (extents). So in this case the ground box is 100 units wide (x-axis) and 20 units tall (y-axis). Box2D is tuned for meters, kilograms, and seconds. So you can consider the extents to be in meters. Box2D generally works best when objects are the size of typical real world objects. For example, a barrel is about 1 meter tall. Due to the limitations of floating point arithmetic, using Box2D to model the movement of glaciers or dust particles is not a good idea.

We finish the ground body in step 4 by creating the shape fixture. For this step we have a shortcut. We do not have a need to alter the default fixture material properties, so we can pass the shape directly to the body without creating a fixture definition. Later we will see how to use a fixture definition for customized material properties. The second parameter is the shape density in kilograms per meter squared. A static body has zero mass by definition, so the density is not used in this case.

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

Box2D does not keep a reference to the shape. It clones the data into a new b2Shape object.

Note that every fixture must have a parent body, even fixtures that are static. However, you can attach all static fixtures to a single static body.

When you attach a shape to a body using a fixture, the shape’s coordinates become local to the body. So when the body moves, so does the shape. A fixture’s world transform is inherited from the parent body. A fixture does not have a transform independent of the body. So we don’t move a shape around on the body. Moving or modifying a shape that is on a body is not supported. The reason is simple: a body with morphing shapes is not a rigid body, but Box2D is a rigid body engine. Many of the assumptions made in Box2D are based on the rigid body model. If this is violated many things will break

* **Creating a Dynamic Body**

これで地上体ができました。同じ手法を使用して、動的なボディを作成できます。寸法以外の主な違いは、動的ボディの質量特性を確立する必要があることです。

まず、CreateBodyを使用して本文を作成します。デフォルトではボディは静的なので、構築時にb2BodyTypeを設定してボディを動的にします。

b2BodyDef bodyDef;

bodyDef.type = b2\_dynamicBody;

bodyDef.position.Set(0.0f, 4.0f);

b2Body\* body = world.CreateBody(&bodyDef);

Caution

力に応じてボディを移動させる場合は、ボディタイプをb2\_dynamicBodyに設定する必要があります。

次に、フィクスチャ定義を使用してポリゴンシェイプを作成してアタッチします。まず、ボックスシェイプを作成します。

b2PolygonShape dynamicBox;

dynamicBox.SetAsBox(1.0f, 1.0f);

次に、ボックスを使用してフィクスチャ定義を作成します。密度を1に設定していることに注意してください。デフォルトの密度はゼロです。また、形状の摩擦は0.3に設定されます。

b2FixtureDef fixtureDef;

fixtureDef.shape = &dynamicBox;

fixtureDef.density = 1.0f;

fixtureDef.friction = 0.3f;

Caution

ダイナミックボディには、密度がゼロ以外のフィクスチャーが少なくとも1つ必要です。そうしないと、奇妙な動作が発生します。

フィクスチャ定義を使用して、フィクスチャを作成できます。これにより、ボディの質量が自動的に更新されます。好きなだけフィクスチャーをボディに追加できます。それぞれが総質量に貢献します。

body->CreateFixture(&fixtureDef);

初期化は以上です。これで、シミュレーションを開始する準備が整いました。

* **Simulating the World (of Box2D)**

したがって、グラウンドボックスと動的ボックスを初期化しました。これで、Newtonを解いて彼のことをやる準備が整いました。考慮すべき問題がさらに2つあります。

Box2Dは、積分器と呼ばれる計算アルゴリズムを使用します。積分器は、離散的な時点で物理方程式をシミュレートします。これは、画面上の動きのフリップブックを本質的に持っている従来のゲームループと同じです。したがって、Box2Dのタイムステップを選択する必要があります。一般に、少なくとも60Hzまたは1/60秒というタイムステップのようなゲームの物理エンジン。より大きな時間ステップで逃げることができますが、あなたの世界の定義をセットアップすることについてより注意する必要があります。また、タイムステップが大きく変わるのも好きではありません。可変時間ステップは可変結果を生成するため、デバッグが困難になります。そのため、タイムステップをフレームレートに結び付けないでください（本当に、本当にそうしなければならない場合を除きます）。これ以上苦労することなく、ここにタイムステップがあります。

float32 timeStep = 1.0f / 60.0f;

積分器に加えて、Box2Dは制約ソルバーと呼ばれるより大きなビットのコードも使用します。制約ソルバーは、シミュレーションのすべての制約を一度に1つずつ解決します。単一の制約を完全に解決できます。ただし、1つの制約を解決すると、他の制約がわずかに混乱します。適切なソリューションを得るには、すべての制約を何度も繰り返す必要があります。

制約ソルバーには、速度フェーズと位置フェーズの2つのフェーズがあります。速度フェーズでは、ソルバーはボディが正しく動くために必要なインパルスを計算します。位置フェーズでは、ソルバーはボディの位置を調整して、オーバーラップとジョイントの剥離を減らします。各フェーズには、独自の反復カウントがあります。さらに、エラーが小さい場合、位置フェーズは反復を早期に終了する場合があります。

Box2Dの推奨される反復カウントは、速度が8、位置が3です。この数値はお好みに合わせて調整できますが、パフォーマンスと精度の間にはトレードオフがあることに注意してください。反復回数を減らすとパフォーマンスは向上しますが、精度が低下します。同様に、反復回数を増やすとパフォーマンスは低下しますが、シミュレーションの品質は向上します。この単純な例では、多くの反復は必要ありません。選択した反復回数は次のとおりです。

int32 velocityIterations = 6;

int32 positionIterations = 2;

タイムステップと反復カウントは完全に無関係であることに注意してください。反復はサブステップではありません。 1つのソルバー反復は、タイムステップ内のすべての制約に対する単一パスです。 1つのタイムステップ内で制約を複数回パスできます。

これで、シミュレーションループを開始する準備ができました。ゲームでは、シミュレーションループをゲームループにマージできます。ゲームループの各パスで、b2World :: Stepを呼び出します。フレームレートと物理タイムステップに応じて、通常は1回の呼び出しで十分です。

Hello Worldプログラムはシンプルに設計されているため、グラフィカルな出力はありません。このコードは、動的ボディの位置と回転を出力します。合計1秒のシミュレート時間で60タイムステップをシミュレートするシミュレーションループを次に示します。

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);

}

The output shows the box falling and landing on the ground box. Your output should look like this:

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

* **Cleanup**

When a world leaves scope or is deleted by calling delete on a pointer, all the memory reserved for bodies, fixtures, and joints is freed. This is done to improve performance and make your life easier. However, you will need to nullify any body, fixture, or joint pointers you have because they will become invalid.

* **The Testbed**

Once you have conquered the HelloWorld example, you should start looking at Box2D's testbed. The testbed is a unit-testing framework and demo environment. Here are some of the features:

* Camera with pan and zoom.
* Mouse picking of shapes attached to dynamic bodies.
* Extensible set of tests.
* GUI for selecting tests, parameter tuning, and debug drawing options.
* Pause and single step simulation.
* Text rendering.



The testbed has many examples of Box2D usage in the test cases and the framework itself. I encourage you to explore and tinker with the testbed as you learn Box2D.

Note: the testbed is written using freeglut and GLUI. The testbed is not part of the Box2D library. The Box2D library is agnostic about rendering. As shown by the HelloWorld example, you don't need a renderer to use Box2D.

* **Common**
* **About**

The Common module contains settings, memory management, and vector math.

* **Settings**

The header b2Settings.h contains:

* Types such as int32 and float32
* Constants
* Allocation wrappers
* The version number
* **Types**

Box2D defines various types such as float32, int8, etc. to make it easy to determine the size of structures.

* **Constants**

Box2D defines several constants. These are all documented in b2Settings.h. Normally you do not need to adjust these constants.

Box2D uses floating point math for collision and simulation. Due to round-off error some numerical tolerances are defined. Some tolerances are absolute and some are relative. Absolute tolerances use MKS units.

* **Allocation wrappers**

The settings file defines b2Alloc and b2Free for large allocations. You may forward these calls to your own memory management system.

* **Version**

The b2Version structure holds the current version so you can query this at run-time.

* **Memory Management**

A large number of the decisions about the design of Box2D were based on the need for quick and efficient use of memory. In this section I will discuss how and why Box2D allocates memory.

Box2D tends to allocate a large number of small objects (around 50-300 bytes). Using the system heap through malloc or new for small objects is inefficient and can cause fragmentation. Many of these small objects may have a short life span, such as contacts, but can persist for several time steps. So we need an allocator that can efficiently provide heap memory for these objects.

Box2D's solution is to use a small object allocator (SOA) called b2BlockAllocator. The SOA keeps a number of growable pools of varying sizes. When a request is made for memory, the SOA returns a block of memory that best fits the requested size. When a block is freed, it is returned to the pool. Both of these operations are fast and cause little heap traffic.

Since Box2D uses a SOA, you should never new or malloc a body, fixture, or joint. However, you do have to allocate a b2World on your own. The b2World class provides factories for you to create bodies, fixtures, and joints. This allows Box2D to use the SOA and hide the gory details from you. Never, call delete or free on a body, fixture, or joint.

While executing a time step, Box2D needs some temporary workspace memory. For this, it uses a stack allocator called b2StackAllocator to avoid per-step heap allocations. You don't need to interact with the stack allocator, but it's good to know it's there.

* **Math**

Box2D includes a simple small vector and matrix module. This has been designed to suit the internal needs of Box2D and the API. All the members are exposed, so you may use them freely in your application.

The math library is kept simple to make Box2D easy to port and maintain.

* **Collision Module**
* **About**

The Collision module contains shapes and functions that operate on them. The module also contains a dynamic tree and broad-phase to acceleration collision processing of large systems.

The collision module is designed to be usable outside of the dynamic system. For example, you can use the dynamic tree for other aspects of your game besides physics.

However, the main purpose of Box2D is to provide a rigid body physics engine, so the using the collision module by itself may feel limited for some applications. Likewise, I will not make a strong effort to document it or polish the APIs.

* **Shapes**

Shapes describe collision geometry and may be used independently of physics simulation. At a minimum, you should understand how to create shapes that can be later attached to rigid bodies.

Box2D shapes implement the b2Shape base class. The base class defines functions to:

* Test a point for overlap with the shape.
* Perform a ray cast against the shape.
* Compute the shape's AABB.
* Compute the mass properties of the shape.

In addition, each shape has a type member and a radius. The radius even applies to polygons, as discussed below.

Keep in mind that a shape does not know about bodies and stand apart from the dynamics system. Shapes are stored in a compact form that is optimized for size and performance. As such, shapes are not easily moved around. You have to manually set the shape vertex positions to move a shape. However, when a shape is attached to a body using a fixture, the shapes move rigidly with the host body. In summary:

* When a shape is **not** attached to a body, you can view it’s vertices as being expressed in world-space.
* When a shape is attached to a body, you can view it’s vertices as being expressed in local coordinates.
* **Circle Shapes**

Circle shapes have a position and radius. Circles are solid. You cannot make a hollow circle using the circle shape.

b2CircleShape circle;

circle.m\_p.Set(2.0f, 3.0f);

circle.m\_radius = 0.5f;

* **Polygon Shapes**

Polygon shapes are solid convex polygons. A polygon is convex when all line segments connecting two points in the interior do not cross any edge of the polygon. Polygons are solid and never hollow. A polygon must have 3 or more vertices.



Polygons vertices are stored with a counter clockwise winding (CCW). We must be careful because the notion of CCW is with respect to a right-handed coordinate system with the z-axis pointing out of the plane. This might turn out to be clockwise on your screen, depending on your coordinate system conventions.



The polygon members are public, but you should use initialization functions to create a polygon. The initialization functions create normal vectors and perform validation.

You can create a polygon shape by passing in a vertex array. The maximal size of the array is controlled by b2\_maxPolygonVertices which has a default value of 8. This is sufficient to describe most convex polygons.

The b2PolygonShape::Set function automatically computes the convex hull and establishes the proper winding order. This function is fast when the number of vertices is low. If you increase b2\_maxPolygonVertices, then the convex hull computation might become slow. Also note that the convex hull function may eliminate and/or re-order the points you provide. Vertices that are closer than b2\_linearSlop may be merged.

// 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);

多角形には、ボックスを作成するための便利な機能がいくつかあります。

void SetAsBox(float32 hx, float32 hy);

void SetAsBox(float32 hx, float32 hy, const b2Vec2& center, float32 angle);

Polygons inherit a radius from b2Shape. The radius creates a skin around the polygon. The skin is used in stacking scenarios to keep polygons slightly separated. This allows continuous collision to work against the core polygon.



The polygon skin helps prevent tunneling by keeping the polygons separated. This results in small gaps between the shapes. Your visual representation can be larger than the polygon to hide any gaps.



* **Edge Shapes**

Edge shapes are line segments. These are provided to assist in making a free-form static environment for your game. A major limitation of edge shapes is that they can collide with circles and polygons but not with themselves. The collision algorithms used by Box2D require that at least one of two colliding shapes have volume. Edge shapes have no volume, so edge-edge collision is not possible.

// This an edge shape.

b2Vec2 v1(0.0f, 0.0f);

b2Vec2 v2(1.0f, 0.0f);

b2EdgeShape edge;

edge.Set(v1, v2);

In many cases a game environment is constructed by connecting several edge shapes end-to-end. This can give rise to an unexpected artifact when a polygon slides along the chain of edges. In the figure below we see a box colliding with an internal vertex. These *ghost* collisions are caused when the polygon collides with an internal vertex generating an internal collision normal.



If edge1 did not exist this collision would seem fine. With edge1 present, the internal collision seems like a bug. But normally when Box2D collides two shapes, it views them in isolation.

Fortunately, the edge shape provides a mechanism for eliminating ghost collisions by storing the adjacent *ghost* vertices. Box2D uses these ghost vertices to prevent internal collisions.



// This is 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;

In general stitching edges together this way is a bit wasteful and tedious. This brings us to chain shapes.

* **Chain Shapes**

The chain shape provides an efficient way to connect many edges together to construct your static game worlds. Chain shapes automatically eliminate ghost collisions and provide two-sided collision.



// 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].Set(-1.7f, 0.4f);

b2ChainShape chain;

chain.CreateChain(vs, 4);

You may have a scrolling game world and would like to connect several chains together. You can connect chains together using ghost vertices, like we did with b2EdgeShape.

// Install ghost vertices

chain.SetPrevVertex(b2Vec2(3.0f, 1.0f));

chain.SetNextVertex(b2Vec2(-2.0f, 0.0f));

You may also create loops automatically.

// Create a loop. The first and last vertices are connected.

b2ChainShape chain;

chain.CreateLoop(vs, 4);

Self-intersection of chain shapes is not supported. It might work, it might not. The code that prevents ghost collisions assumes there are no self-intersections of the chain. Also, very close vertices can cause problems. Make sure all your edges are longer than b2\_linearSlop (5mm).



Each edge in the chain is treated as a child shape and can be accessed by index. When a chain shape is connected to a body, each edge gets its own bounding box in the broad-phase collision tree.

// Visit each child edge.

for (int32 i = 0; i < chain.GetChildCount(); ++i)

{

b2EdgeShape edge;

chain.GetChildEdge(&edge, i);

…

}

* **Unary Geometric Queries**

You can perform a couple geometric queries on a single shape.

* **Shape Point Test**

You can test a point for overlap with a shape. You provide a transform for the shape and a world point.

b2Transfrom transform;

transform.SetIdentity();

b2Vec2 point(5.0f, 2.0f);

bool hit = shape->TestPoint(transform, point);

Edge and chain shapes always return false, even if the chain is a loop.

* **Shape Ray Cast**

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);

input.p2.Set(1.0f, 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);

…

}

* **Binary Functions**

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
* **Overlap**

You can test two shapes for overlap using this function:

b2Transform xfA = …, xfB = …;

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

Again you must provide child indices to for the case of chain shapes.

* **Contact Manifolds**

Box2Dには、重なり合う形状の接触点を計算する関数があります。サークルサークルまたはサークルポリゴンを考慮すると、1つの接触点と法線のみを取得できます。 polygon-polygonの場合、2つのポイントを取得できます。これらの点は同じ法線ベクトルを共有するため、Box2Dはそれらを多様体構造にグループ化します。接触ソルバーはこれを利用して、スタッキングの安定性を改善します。



通常、接触マニホールドを直接計算する必要はありませんが、シミュレーションで生成された結果を使用する可能性があります。

b2Manifold構造は、法線ベクトルと最大2つの接触点を保持します。法線と点はローカル座標で保持されます。コンタクトソルバーの利便性として、各ポイントには法線および接線（摩擦）インパルスが保存されます。

b2Manifoldに保存されているデータは、内部使用向けに最適化されています。このデータが必要な場合は、通常b2WorldManifold構造を使用して、接触法線と点の世界座標を生成するのが最善です。 b2Manifoldと形状変換と半径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];

…

}

ワールド多様体は元の多様体からのポイントカウントを使用することに注意してください。

シミュレーション中に形状が移動し、マニホールドが変化する場合があります。ポイントは追加または削除できます。これはb2GetPointStatesを使用して検出できます。

b2PointState state1[2], state2[2];

b2GetPointStates(state1, state2, &manifold1, &manifold2);

if (state1[0] == b2\_removeState)

{

// process event

}

* **Distance**

b2Distance関数を使用して、2つの形状間の距離を計算できます。距離関数では、両方の形状をb2DistanceProxyに変換する必要があります。繰り返される呼び出しの距離関数をウォームスタートするために使用されるキャッシュもあります。詳細はb2Distance.hで確認できます。



* **Time of Impact**

2つのシェイプが高速で移動している場合、1つのタイムステップで相互にトンネリングする可能性があります。



b2TimeOfImpact関数は、2つの移動する図形が衝突する時間を決定するために使用されます。これは影響時間（TOI）と呼ばれます。 b2TimeOfImpactの主な目的は、トンネルの防止です。特に、移動するオブジェクトが静的レベルジオメトリの外側をトンネリングしないように設計されています。

この関数は両方の形状の回転と平行移動を考慮しますが、回転が十分に大きい場合、関数は衝突を見逃す可能性があります。ただし、関数は重複していない時間を報告し、すべての並進衝突をキャプチャします。

衝突時間の関数は、初期の分離軸を識別し、形状がその軸上で交差しないようにします。これは、最終位置で明確な衝突を逃す可能性があります。このアプローチは衝突を逃す可能性がありますが、非常に高速であり、トンネルの防止に適しています。



回転の大きさに制限を加えることは困難です。小さな回転では衝突が見落とされる場合があります。通常、これらの見逃された回転衝突はゲームプレイに害を与えるべきではありません。それらはちらっと衝突する傾向があります。

この関数には、2つの形状（b2DistanceProxyに変換）と2つのb2Sweep構造が必要です。スイープ構造は、形状の初期および最終変換を定義します。

固定回転を使用して、シェイプキャストを実行できます。この場合、衝突時間関数は衝突を見逃しません。

* **Dynamic Tree**

b2DynamicTreeクラスは、多数の形状を効率的に整理するためにBox2Dによって使用されます。クラスは形状については知りません。代わりに、ユーザーデータポインターを使用して軸に合わせた境界ボックス（AABB）で動作します。

動的ツリーは階層的なAABBツリーです。ツリーの各内部ノードには2つの子があります。リーフノードはシングルユーザーAABBです。ツリーは、入力が縮退している場合でも、回転を使用してツリーのバランスを保ちます。

ツリー構造により、効率的なレイキャストと領域クエリが可能になります。たとえば、シーンに何百もの形状がある場合があります。各シェイプをレイキャストすることで、ブルートフォース方式でシーンに対してレイキャストを実行できます。これは、シェイプが広がっていることを利用していないため、非効率的です。代わりに、動的なツリーを維持し、ツリーに対してレイキャストを実行できます。これは、多数のシェイプをスキップしてツリーをレイを横断します。

領域クエリは、ツリーを使用して、クエリAABBと重複するすべてのリーフAABBを見つけます。これは、多くの形状をスキップできるため、ブルートフォースアプローチよりも高速です。





通常、動的ツリーを直接使用することはありません。むしろ、レイキャストとリージョンクエリのb2Worldクラスを使用します。独自の動的ツリーのインスタンス化を計画している場合は、Box2Dがどのように使用するかを調べることで、その使用方法を学ぶことができます。

* **Broad-phase**

物理ステップでの衝突処理は、狭相と広相に分けることができます。狭位相では、形状のペア間の接点を計算します。 N個の図形があるとします。ブルートフォースを使用して、N \* N / 2ペアの狭位相を実行する必要があります。

b2BroadPhaseクラスは、ペア管理に動的ツリーを使用することにより、この負荷を軽減します。これにより、狭位相呼び出しの数が大幅に削減されます。

通常、ブロードフェーズと直接対話することはありません。代わりに、Box2Dは内部で広範なフェーズを作成および管理します。また、b2BroadPhaseはBox2Dのシミュレーションループを念頭に置いて設計されているため、他のユースケースには適していません。

* **Dynamics Module**
* **Overview**

The Dynamics module is the most complex part of Box2D and is the part you likely interact with the most. The Dynamics module sits on top of the Common and Collision modules, so you should be somewhat familiar with those by now.

The Dynamics module contains:

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

There are many dependencies between these classes so it is difficult to describe one class without referring to another. In the following, you may see some references to classes that have not been described yet. Therefore, you may want to quickly skim this chapter before reading it closely.

The dynamics module is covered in the following chapters.

* **Bodies**
* **About**

Bodies have position and velocity. You can apply forces, torques, and impulses to bodies. Bodies can be static, kinematic, or dynamic. Here are the body type definitions:

* ***b2\_staticBody***

静的ボディは、シミュレーションでは移動せず、無限の質量を持っているかのように動作します。内部的に、Box2Dは質量と逆質量に対してゼロを保存します。静的ボディは、ユーザーが手動で移動できます。静的ボディの速度はゼロです。静的ボディは、他の静的または運動学的ボディと衝突しません。

* ***b2\_kinematicBody***

運動体は、その速度に応じてシミュレーションの下で動きます。運動体は力に反応しません。これらはユーザーが手動で移動できますが、通常、運動体はその速度を設定することで移動します。キネマティックボディは無限の質量を持っているかのように動作しますが、Box2Dは質量と逆質量にゼロを格納します。キネマティックボディは、他のキネマティックボディまたは静的ボディと衝突しません。

* ***b2\_dynamicBody***

動的ボディは完全にシミュレートされます。これらはユーザーが手動で移動できますが、通常は力に従って移動します。動的なボディは、すべてのボディタイプと衝突する可能性があります。動的な物体は常に有限のゼロでない質量を持っています。動的なボディの質量をゼロに設定しようとすると、自動的に1キログラムの質量が取得され、回転しません。

ボディは、フィクスチャ（形状）のバックボーンです。ボディはフィクスチャーを運び、世界中で動かします。 Box2Dでは、ボディは常に剛体です。つまり、同じリジッドボディに取り付けられた2つのフィクスチャが互いに相対的に移動することはなく、同じボディに取り付けられたフィクスチャが衝突することはありません。

フィクスチャには衝突ジオメトリと密度があります。通常、ボディはフィクスチャから質量特性を取得します。ただし、ボディの構築後にマスプロパティをオーバーライドできます。

通常、作成するすべてのボディへのポインターを保持します。これにより、グラフィックエンティティの位置を更新するために、ボディの位置を照会できます。また、ボディポインタを保持して、使用後にそれらを破棄できるようにする必要があります。

* **Body Definition**

Before a body is created you must create a body definition (b2BodyDef). The body definition holds the data needed to create and initialize a body.

Box2D copies the data out of the body definition; it does not keep a pointer to the body definition. This means you can recycle a body definition to create multiple bodies.

Let’s go over some of the key members of the body definition.

* **Body Type**

As discussed at the beginning of this chapter, there are three different body types: static, kinematic, and dynamic. You should establish the body type at creation because changing the body type later is expensive.

bodyDef.type = b2\_dynamicBody;

Setting the body type is mandatory.

* **Position and Angle**

The body definition gives you the chance to initialize the position of the body on creation. This has far better performance than creating the body at the world origin and then moving the body.

**Caution**

Do not create a body at the origin and then move it. If you create several bodies at the origin, then performance will suffer.

ボディには2つの主要な関心点があります。最初のポイントは体の原点です。フィクスチャとジョイントは、ボディの原点を基準にしてアタッチされます。 2番目の関心点は重心です。重心は、アタッチされた形状の質量分布から決定されるか、b2MassDataで明示的に設定されます。 Box2Dの内部計算の多くは、重心位置を使用します。たとえば、b2Bodyは重心の線速度を格納します。

身体の定義を構築するとき、重心の位置がわからない場合があります。したがって、ボディの原点の位置を指定します。ボディの角度をラジアンで指定することもできますが、これは重心の位置の影響を受けません。後でボディの質量特性を変更すると、質量中心がボディ上で移動する場合がありますが、原点位置は変更されず、アタッチされたシェイプとジョイントは移動しません。

bodyDef.position.Set(0.0f, 2.0f); // the body's origin position.

bodyDef.angle = 0.25f \* b2\_pi; // the body's angle in radians.

剛体も参照のフレームです。そのフレームでフィクスチャとジョイントを定義できます。これらのフィクスチャとジョイントアンカーは、ボディのローカルフレーム内で移動することはありません。

* **Damping**

減衰は、ボディのワールド速度を下げるために使用されます。摩擦は接触によってのみ発生するため、減衰は摩擦とは異なります。減衰は摩擦の代替ではなく、2つの効果を一緒に使用する必要があります。

減衰パラメータは0から無限大の間で、0は減衰なし、無限大は完全減衰を意味します。通常、0〜0.1の減衰値を使用します。体が浮いているように見えるので、私は一般的に線形減衰を使用しません。

bodyDef.linearDamping = 0.0f;

bodyDef.angularDamping = 0.01f;

ダンピングは、安定性とパフォーマンスのために近似されています。減衰値が小さい場合、減衰効果はほとんど時間ステップに依存しません。減衰値が大きい場合、減衰効果は時間ステップとともに変化します。固定タイムステップを使用する場合、これは問題ではありません（推奨）。

* **Gravity Scale**

重力スケールを使用して、単一のボディの重力を調整できます。ただし、重力を上げると安定性が低下する可能性があることに注意してください。

// Set the gravity scale to zero so this body will float

bodyDef.gravityScale = 0.0f;

* **Sleep Parameters**

睡眠とはどういう意味ですか？ボディをシミュレートするのは費用がかかるため、シミュレートする必要が少ないほど良いです。体が休息するようになると、体のシミュレーションを停止します。

ボディ（またはボディのグループ）が静止したとBox2Dが判断すると、ボディはスリープ状態に入り、CPUオーバーヘッドはほとんどありません。体が目覚めていて、眠っている体と衝突すると、眠っている体が起きます。ボディは、それらに接続されたジョイントまたはコンタクトが破壊された場合にも起動します。手動で体を起こすこともできます。

ボディ定義では、ボディがスリープできるかどうか、およびボディがスリープ状態で作成されるかどうかを指定できます。

bodyDef.allowSleep = true;

bodyDef.awake = true;

* **Fixed Rotation**

You may want a rigid body, such as a character, to have a fixed rotation. Such a body should not rotate, even under load. You can use the fixed rotation setting to achieve this:

bodyDef.fixedRotation = true;

The fixed rotation flag causes the rotational inertia and its inverse to be set to zero.

* **Bullets**

通常、ゲームシミュレーションは、特定のフレームレートで再生される一連の画像を生成します。これは、離散シミュレーションと呼ばれます。離散シミュレーションでは、剛体は1つのタイムステップで大量に移動できます。物理エンジンが大きな動きを考慮していない場合、一部のオブジェクトが互いに誤って通過することがあります。この効果はトンネリングと呼ばれます。

デフォルトでは、Box2Dは連続衝突検出（CCD）を使用して、動的ボディが静的ボディをトンネリングしないようにします。これは、古い位置から新しい位置に形状をスイープすることによって行われます。エンジンはスイープ中に新しい衝突を探し、これらの衝突の影響時間（TOI）を計算します。ボディは最初のTOIに移動し、ソルバーはサブステップを実行してフルタイムステップを完了します。サブステップ内に追加のTOIイベントがある場合があります。

通常、ダイナミックボディ間ではCCDは使用されません。これは、パフォーマンスを適切に保つために行われます。一部のゲームシナリオでは、CCDを使用するために動的なボディが必要です。たとえば、動的なブリックのスタックで高速の弾丸を撃つことができます。 CCDがないと、弾丸がレンガを通り抜ける可能性があります。

Box2Dの高速移動オブジェクトには、箇条書きのラベルを付けることができます。弾丸は、静的なボディと動的なボディの両方でCCDを実行します。ゲームデザインに基づいて、弾丸にする体を決定する必要があります。ボディを箇条書きとして扱う必要があると判断した場合は、次の設定を使用します。

bodyDef.bullet = true;

Bulletフラグは、ダイナミックボディにのみ影響します。

* **Activation**

ボディを作成したいが、コリジョンやダイナミクスに参加したくない場合があります。この状態は、ボディが他のボディによって起こされず、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ことを除いて、Sleepingに似ています。これは、ボディが衝突、レイキャストなどに参加しないことを意味します。

ボディを非アクティブ状態で作成し、後で再度アクティブにすることができます。

bodyDef.active = true;

Joints may be connected to inactive bodies. These joints will not be simulated. You should be careful when you activate a body that its joints are not distorted.

Note that activating a body is almost as expensive as creating the body from scratch. So you should not use activation for streaming worlds. Use creation/destruction for streaming worlds to save memory.

* **User Data**

User data is a void pointer. This gives you a hook to link your application objects to bodies. You should be consistent to use the same object type for all body user data.

b2BodyDef bodyDef;

bodyDef.userData = &myActor;

* **Body Factory**

Bodies are created and destroyed using a body factory provided by the world class. This lets the world create the body with an efficient allocator and add the body to the world data structure.

b2Body\* dynamicBody = myWorld->CreateBody(&bodyDef);

... do stuff ...

myWorld->DestroyBody(dynamicBody);

dynamicBody = NULL;

**Caution**

You should never use new or malloc to create a body. The world won't know about the body and the body won't be properly initialized.

Box2Dは、本体定義または保持するデータ（ユーザーデータポインターを除く）への参照を保持しません。したがって、一時的なボディ定義を作成し、同じボディ定義を再利用できます。

Box2Dでは、b2Worldオブジェクトを削除することにより、ボディの破壊を回避できます。これにより、すべてのクリーンアップ作業が実行されます。ただし、ゲームエンジンに保持しているボディポインターを無効にすることに注意する必要があります。

ボディを破壊すると、アタッチされたフィクスチャーとジョイントは自動的に破壊されます。これは、シェイプポインターとジョイントポインターの管理方法に重要な意味を持ちます。

* **Using a Body**

ボディを作成した後、ボディに対して実行できる多くの操作があります。これらには、質量特性の設定、位置と速度へのアクセス、力の適用、点とベクトルの変換が含まれます。

* **Mass Data**

ボディには、質量（スカラー）、重心（2ベクトル）、および回転慣性（スカラー）があります。静的ボディの場合、質量と回転慣性はゼロに設定されます。ボディの回転が固定されている場合、その回転慣性はゼロです。

通常、フィクスチャーがボディに追加されると、ボディの質量特性が自動的に確立されます。実行時にボディの質量を調整することもできます。これは通常、質量を変更する必要がある特別なゲームシナリオがある場合に行われます。

void SetMassData(const b2MassData\* data);

ボディの質量を直接設定した後、器具によって指示された自然の質量に戻したい場合があります。これを行うには：

void ResetMassData();

体の質量データは、次の機能を介して利用できます。

float32 GetMass() const;

float32 GetInertia() const;

const b2Vec2& GetLocalCenter() const;

void GetMassData(b2MassData\* data) const;

* **State Information**

体の状態には多くのaspectsがあります。次の機能を使用して、この状態データ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;

* **Position and Velocity**

体の位置と回転にアクセスできます。これは、関連付けられたゲームアクターをレンダリングする場合に一般的です。位置を設定することもできますが、通常はBox2Dを使用して動きをシミュレートするため、これはあまり一般的ではありません。

bool SetTransform(const b2Vec2& position, float32 angle);

const b2Transform& GetTransform() const;

const b2Vec2& GetPosition() const;

float32 GetAngle() const;

ローカル座標とワールド座標で重心位置にアクセスできます。 Box2Dの内部シミュレーションの多くは、重心を使用しています。ただし、通常はアクセスする必要はありません。代わりに、通常はボディトランスフォームを使用します。たとえば、正方形のボディがある場合があります。体の原点は正方形の角かもしれませんが、重心は正方形の中心にあります。

const b2Vec2& GetWorldCenter() const;

const b2Vec2& GetLocalCenter() const;

線速度と角速度にアクセスできます。線速度は重心用です。したがって、質量特性が変化すると、線速度が変化する場合があります。

* **Fixtures**
* **About**

シェイプはボディを認識しないため、物理シミュレーションとは独立して使用できることを思い出してください。そのため、Box2Dはb2Fixtureクラスを提供して、シェイプをボディにアタッチします。ボディには0個以上のフィクスチャが含まれる場合があります。複数のフィクスチャを持つボディは、複合ボディと呼ばれることもあります。

Fixtures hold the following:

* a single shape
* broad-phase proxies
* density, friction, and restitution
* collision filtering flags
* back pointer to the parent body
* user data
* sensor flag

These are described in the following sections.

* **Fixture Creation**

Fixtures are created by initializing a fixture definition and then passing the definition to the parent body.

b2FixtureDef fixtureDef;

fixtureDef.shape = &myShape;

fixtureDef.density = 1.0f;

b2Fixture\* myFixture = myBody->CreateFixture(&fixtureDef);

This creates the fixture and attaches it to the body. You do not need to store the fixture pointer since the fixture will automatically be destroyed when the parent body is destroyed. You can create multiple fixtures on a single body.

You can destroy a fixture on the parent body. You may do this to model a breakable object. Otherwise you can just leave the fixture alone and let the body destruction take care of destroying the attached fixtures.

myBody->DestroyFixture(myFixture);

* **Density**

The fixture density is used to compute the mass properties of the parent body. The density can be zero or positive. You should generally use similar densities for all your fixtures. This will improve stacking stability.

The mass of a body is not adjusted when you set the density. You must call ResetMassData for this to occur.

fixture->SetDensity(5.0f);

body->ResetMassData();

* **Friction**

Friction is used to make objects slide along each other realistically. Box2D supports static and dynamic friction, but uses the same parameter for both. Friction is simulated accurately in Box2D and the friction strength is proportional to the normal force (this is called Coulomb friction). The friction parameter is usually set between 0 and 1, but can be any non-negative value. A friction value of 0 turns off friction and a value of 1 makes the friction strong. When the friction force is computed between two shapes, Box2D must combine the friction parameters of the two parent fixtures. This is done with the geometric mean:

float32 friction;

friction = sqrtf(fixtureA->friction \* fixtureB->friction);

So if one fixture has zero friction then the contact will have zero friction.

You can override the default mixed friction using b2Contact::SetFriction. This is usually done in the b2ContactListener callback.

* **Restitution**

Restitution is used to make objects bounce. The restitution value is usually set to be between 0 and 1. Consider dropping a ball on a table. A value of zero means the ball won't bounce. This is called an inelastic collision. A value of one means the ball's velocity will be exactly reflected. This is called a perfectly elastic collision. Restitution is combined using the following formula.

float32 restitution;

restitution = b2Max(fixtureA->restitution, fixtureB->restitution);

Restitution is combined this way so that you can have a bouncy super ball without having a bouncy floor.

You can override the default mixed restitution using b2Contact::SetRestitution. This is usually done in the b2ContactListener callback.

When a shape develops multiple contacts, restitution is simulated approximately. This is because Box2D uses an iterative solver. Box2D also uses inelastic collisions when the collision velocity is small. This is done to prevent jitter. See b2\_velocityThreshold in b2Settings.h.

* **Filtering**

Collision filtering allows you to prevent collision between fixtures. For example, say you make a character that rides a bicycle. You want the bicycle to collide with the terrain and the character to collide with the terrain, but you don't want the character to collide with the bicycle (because they must overlap). Box2D supports such collision filtering using categories and groups.

Box2D supports 16 collision categories. For each fixture you can specify which category it belongs to. You also specify what other categories this fixture can collide with. For example, you could specify in a multiplayer game that all players don't collide with each other and monsters don't collide with each other, but players and monsters should collide. This is done with masking bits. For example:

playerFixtureDef.filter.categoryBits = 0x0002;

monsterFixtureDef.filter.categoryBits = 0x0004;

playerFixtureDef.filter.maskBits = 0x0004;

monsterFixtureDef.filter.maskBits = 0x0002;

Here is the rule for a collision to occur:

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

}

Collision groups let you specify an integral group index. You can have all fixtures with the same group index always collide (positive index) or never collide (negative index). Group indices are usually used for things that are somehow related, like the parts of a bicycle. In the following example, fixture1 and fixture2 always collide, but fixture3 and fixture4 never collide.

fixture1Def.filter.groupIndex = 2;

fixture2Def.filter.groupIndex = 2;

fixture3Def.filter.groupIndex = -8;

fixture4Def.filter.groupIndex = -8;

Collisions between fixtures of different group indices are filtered according the category and mask bits. In other words, group filtering has higher precedence than category filtering.

Note that additional collision filtering occurs in Box2D. Here is a list:

* A fixture on a static body can only collide with a dynamic body.
* A fixture on a kinematic body can only collide with a dynamic body.
* Fixtures on the same body never collide with each other.
* You can optionally enable/disable collision between fixtures on bodies connected by a joint.

Sometimes you might need to change collision filtering after a fixture has already been created. You can get and set the b2Filter structure on an existing fixture using b2Fixture::GetFilterData and b2Fixture::SetFilterData. Note that changing the filter data will not add or remove contacts until the next time step (see the World class).

* **Sensors**

Sometimes game logic needs to know when two fixtures overlap yet there should be no collision response. This is done by using sensors. A sensor is a fixture that detects collision but does not produce a response.

You can flag any fixture as being a sensor. Sensors may be static, kinematic, or dynamic. Remember that you may have multiple fixtures per body and you can have any mix of sensors and solid fixtures. Also, sensors only form contacts when at least one body is dynamic, so you will not get a contact for kinematic versus kinematic, kinematic versus static, or static versus static.

Sensors do not generate contact points. There are two ways to get the state of a sensor:

* b2Contact::IsTouching
* b2ContactListener::BeginContact and EndContact
* **Joints**
* **About**

Joints are used to constrain bodies to the world or to each other. Typical examples in games include ragdolls, teeters, and pulleys. Joints can be combined in many different ways to create interesting motions.

Some joints provide limits so you can control the range of motion. Some joint provide motors which can be used to drive the joint at a prescribed speed until a prescribed force/torque is exceeded.

Joint motors can be used in many ways. You can use motors to control position by specifying a joint velocity that is proportional to the difference between the actual and desired position. You can also use motors to simulate joint friction: set the joint velocity to zero and provide a small, but significant maximum motor force/torque. Then the motor will attempt to keep the joint from moving until the load becomes too strong.

* **The Joint Definition**

Each joint type has a definition that derives from b2JointDef. All joints are connected between two different bodies. One body may static. Joints between static and/or kinematic bodies are allowed, but have no effect and use some processing time.

You can specify user data for any joint type and you can provide a flag to prevent the attached bodies from colliding with each other. This is actually the default behavior and you must set the collideConnected Boolean to allow collision between to connected bodies.

Many joint definitions require that you provide some geometric data. Often a joint will be defined by anchor points. These are points fixed in the attached bodies. Box2D requires these points to be specified in local coordinates. This way the joint can be specified even when the current body transforms violate the joint constraint --- a common occurrence when a game is saved and reloaded. Additionally, some joint definitions need to know the default relative angle between the bodies. This is necessary to constrain rotation correctly.

Initializing the geometric data can be tedious, so many joints have initialization functions that use the current body transforms to remove much of the work. However, these initialization functions should usually only be used for prototyping. Production code should define the geometry directly. This will make joint behavior more robust.

The rest of the joint definition data depends on the joint type. We cover these now.

* **Joint Factory**

Joints are created and destroyed using the world factory methods. This brings up an old issue:

**Caution**

Don't try to create a joint on the stack or on the heap using new or malloc. You must create and destroy bodies and joints using the create and destroy methods of the b2World class.

Here's an example of the lifetime of a revolute joint:

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;

It is always good to nullify your pointer after they are destroyed. This will make the program crash in a controlled manner if you try to reuse the pointer.

The lifetime of a joint is not simple. Heed this warning well:

**Caution**

Joints are destroyed when an attached body is destroyed.

This precaution is not always necessary. You may organize your game engine so that joints are always destroyed before the attached bodies. In this case you don't need to implement the listener class. See the section on Implicit Destruction for details.

* **Using Joints**

Many simulations create the joints and don't access them again until they are destroyed. However, there is a lot of useful data contained in joints that you can use to create a rich simulation.

First of all, you can get the bodies, anchor points, and user data from a joint.

b2Body\* GetBodyA();

b2Body\* GetBodyB();

b2Vec2 GetAnchorA();

b2Vec2 GetAnchorB();

void\* GetUserData();

All joints have a reaction force and torque. This the reaction force applied to body 2 at the anchor point. You can use reaction forces to break joints or trigger other game events. These functions may do some computations, so don't call them if you don't need the result.

b2Vec2 GetReactionForce();

float32 GetReactionTorque();

* **Distance Joint**

One of the simplest joint is a distance joint which says that the distance between two points on two bodies must be constant. When you specify a distance joint the two bodies should already be in place. Then you specify the two anchor points in world coordinates. The first anchor point is connected to body 1, and the second anchor point is connected to body 2. These points imply the length of the distance constraint.



Here is an example of a distance joint definition. In this case we decide to allow the bodies to collide.

b2DistanceJointDef jointDef;

jointDef.Initialize(myBodyA, myBodyB, worldAnchorOnBodyA, worldAnchorOnBodyB);

jointDef.collideConnected = true;

The distance joint can also be made soft, like a spring-damper connection. See the Web example in the testbed to see how this behaves.

Softness is achieved by tuning two constants in the definition: frequency and damping ratio. Think of the frequency as the frequency of a harmonic oscillator (like a guitar string). The frequency is specified in Hertz. Typically the frequency should be less than a half the frequency of the time step. So if you are using a 60Hz time step, the frequency of the distance joint should be less than 30Hz. The reason is related to the Nyquist frequency.

The damping ratio is non-dimensional and is typically between 0 and 1, but can be larger. At 1, the damping is critical (all oscillations should vanish).

jointDef.frequencyHz = 4.0f;

jointDef.dampingRatio = 0.5f;

* **Revolute Joint**

A revolute joint forces two bodies to share a common anchor point, often called a hinge point. The revolute joint has a single degree of freedom: the relative rotation of the two bodies. This is called the joint angle.



To specify a revolute you need to provide two bodies and a single anchor point in world space. The initialization function assumes that the bodies are already in the correct position.

In this example, two bodies are connected by a revolute joint at the first body's center of mass.

b2RevoluteJointDef jointDef;

jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter());

The revolute joint angle is positive when bodyB rotates CCW about the angle point. Like all angles in Box2D, the revolute angle is measured in radians. By convention the revolute joint angle is zero when the joint is created using Initialize(), regardless of the current rotation of the two bodies.

In some cases you might wish to control the joint angle. For this, the revolute joint can optionally simulate a joint limit and/or a motor.

A joint limit forces the joint angle to remain between a lower and upper bound. The limit will apply as much torque as needed to make this happen. The limit range should include zero, otherwise the joint will lurch when the simulation begins.

A joint motor allows you to specify the joint speed (the time derivative of the angle). The speed can be negative or positive. A motor can have infinite force, but this is usually not desirable. Recall the eternal question:

*"What happens when an irresistible force meets an immovable object?"*

I can tell you it's not pretty. So you can provide a maximum torque for the joint motor. The joint motor will maintain the specified speed unless the required torque exceeds the specified maximum. When the maximum torque is exceeded, the joint will slow down and can even reverse.

You can use a joint motor to simulate joint friction. Just set the joint speed to zero, and set the maximum torque to some small, but significant value. The motor will try to prevent the joint from rotating, but will yield to a significant load.

Here's a revision of the revolute joint definition above; this time the joint has a limit and a motor enabled. The motor is setup to simulate joint friction.

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;

You can access a revolute joint's angle, speed, and motor torque.

float32 GetJointAngle() const;

float32 GetJointSpeed() const;

float32 GetMotorTorque() const;

You also update the motor parameters each step.

void SetMotorSpeed(float32 speed);

void SetMaxMotorTorque(float32 torque);

Joint motors have some interesting abilities. You can update the joint speed every time step so you can make the joint move back-and-forth like a sine-wave or according to whatever function you want.

... Game Loop Begin ...

myJoint->SetMotorSpeed(cosf(0.5f \* time));

... Game Loop End ...

You can also use joint motors to track a desired joint angle. For example:

... Game Loop Begin ...

float32 angleError = myJoint->GetJointAngle() - angleTarget;

float32 gain = 0.1f;

myJoint->SetMotorSpeed(-gain \* angleError);

... Game Loop End ...

Generally your gain parameter should not be too large. Otherwise your joint may become unstable.

* **Prismatic Joint**

A prismatic joint allows for relative translation of two bodies along a specified axis. A prismatic joint prevents relative rotation. Therefore, a prismatic joint has a single degree of freedom.



The prismatic joint definition is similar to the revolute joint description; just substitute translation for angle and force for torque. Using this analogy provides an example prismatic joint definition with a joint limit and a friction motor:

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;

The revolute joint has an implicit axis coming out of the screen. The prismatic joint needs an explicit axis parallel to the screen. This axis is fixed in the two bodies and follows their motion.

Like the revolute joint, the prismatic joint translation is zero when the joint is created using Initialize(). So be sure zero is between your lower and upper translation limits.

Using a prismatic joint is similar to using a revolute joint. Here are the relevant member functions:

float32 GetJointTranslation() const;

float32 GetJointSpeed() const;

float32 GetMotorForce() const;

void SetMotorSpeed(float32 speed);

void SetMotorForce(float32 force);

* **Pulley Joint**

A pulley is used to create an idealized pulley. The pulley connects two bodies to ground and to each other. As one body goes up, the other goes down. The total length of the pulley rope is conserved according to the initial configuration.

length1 + length2 == constant

You can supply a ratio that simulates a block and tackle. This causes one side of the pulley to extend faster than the other. At the same time the constraint force is smaller on one side than the other. You can use this to create mechanical leverage.

length1 + ratio \* length2 == constant

For example, if the ratio is 2, then length1 will vary at twice the rate of length2. Also the force in the rope attached to body1 will have half the constraint force as the rope attached to body2.



Pulleys can be troublesome when one side is fully extended. The rope on the other side will have zero length. At this point the constraint equations become singular (bad). You should configure collision shapes to prevent this.

Here is an example pulley definition:

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);

Pulley joints provide the current lengths.

float32 GetLengthA() const;

float32 GetLengthB() const;

* **Gear Joint**

If you want to create a sophisticated mechanical contraption you might want to use gears. In principle you can create gears in Box2D by using compound shapes to model gear teeth. This is not very efficient and might be tedious to author. You also have to be careful to line up the gears so the teeth mesh smoothly. Box2D has a simpler method of creating gears: the gear joint.



The gear joint can only connect revolute and/or prismatic joints.

Like the pulley ratio, you can specify a gear ratio. However, in this case the gear ratio can be negative. Also keep in mind that when one joint is a revolute joint (angular) and the other joint is prismatic (translation), and then the gear ratio will have units of length or one over length.

coordinate1 + ratio \* coordinate2 == constant

Here is an example gear joint. The bodies myBodyA and myBodyB are any bodies from the two joints, as long as they are not the same bodies.

b2GearJointDef jointDef;

jointDef.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.joint1 = myRevoluteJoint;

jointDef.joint2 = myPrismaticJoint;

jointDef.ratio = 2.0f \* b2\_pi / myLength;

Note that the gear joint depends on two other joints. This creates a fragile situation. What happens if those joints are deleted?

**Caution**

Always delete gear joints before the revolute/prismatic joints on the gears. Otherwise your code will crash in a bad way due to the orphaned joint pointers in the gear joint. You should also delete the gear joint before you delete any of the bodies involved.

* **Mouse Joint**

The mouse joint is used in the testbed to manipulate bodies with the mouse. It attempts to drive a point on a body towards the current position of the cursor. There is no restriction on rotation.

The mouse joint definition has a target point, maximum force, frequency, and damping ratio. The target point initially coincides with the body’s anchor point. The maximum force is used to prevent violent reactions when multiple dynamic bodies interact. You can make this as large as you like. The frequency and damping ratio are used to create a spring/damper effect similar to the distance joint.

Many users have tried to adapt the mouse joint for game play. Users often want to achieve precise positioning and instantaneous response. The mouse joint doesn’t work very well in that context. You may wish to consider using kinematic bodies instead.

* **Wheel Joint**

The wheel joint restricts a point on bodyB to a line on bodyA. The wheel joint also provides a suspension spring. See b2WheelJoint.h and Car.h for details.



* **Weld Joint**

The weld joint attempts to constrain all relative motion between two bodies. See the Cantilever.h in the testbed to see how the weld joint behaves.

It is tempting to use the weld joint to define breakable structures. However, the Box2D solver is iterative so the joints are a bit soft. So chains of bodies connected by weld joints will flex.

Instead it is better to create breakable bodies starting with a single body with multiple fixtures. When the body breaks, you can destroy a fixture and recreate it on a new body. See the Breakable example in the testbed.

* **Rope Joint**

The rope joint restricts the maximum distance between two points. This can be useful to prevent chains of bodies from stretching, even under high load. See b2RopeJoint.h and RopeJoint.h for details.

* **Friction Joint**

The friction joint is used for top-down friction. The joint provides 2D translational friction and angular friction. See b2FrictionJoint.h and ApplyForce.h for details.

* **Motor Joint**

A motor joint lets you control the motion of a body by specifying target position and rotation offsets. You can set the maximum motor force and torque that will be applied to reach the target position and rotation. If the body is blocked, it will stop and the contact forces will be proportional the maximum motor force and torque. See b2MotorJoint and MotorJoint.h for details.

* **Contacts**
* **About**

Contacts are objects created by Box2D to manage collision between two fixtures. If the fixture has children, such as a chain shape, then a contact exists for each relevant child. There are different kinds of contacts, derived from b2Contact, for managing contact between different kinds of fixtures. For example there is a contact class for managing polygon-polygon collision and another contact class for managing circle-circle collision.

Here is some terminology associated with contacts.

* **contact point**

A contact point is a point where two shapes touch. Box2D approximates contact with a small number of points.

* **contact normal**

A contact normal is a unit vector that points from one shape to another. By convention, the normal points from fixtureA to fixtureB.

* **contact separation**

Separation is the opposite of penetration. Separation is negative when shapes overlap. It is possible that future versions of Box2D will create contact points with positive separation, so you may want to check the sign when contact points are reported.

* **contact manifold**

Contact between two convex polygons may generate up to 2 contact points. Both of these points use the same normal, so they are grouped into a contact manifold, which is an approximation of a continuous region of contact.

* **normal impulse**

The normal force is the force applied at a contact point to prevent the shapes from penetrating. For convenience, Box2D works with impulses. The normal impulse is just the normal force multiplied by the time step.

* **tangent impulse**

The tangent force is generated at a contact point to simulate friction. For convenience, this is stored as an impulse.

* **contact ids**

Box2D tries to re-use the contact force results from a time step as the initial guess for the next time step. Box2D uses contact ids to match contact points across time steps. The ids contain geometric features indices that help to distinguish one contact point from another.

Contacts are created when two fixture’s AABBs overlap. Sometimes collision filtering will prevent the creation of contacts. Contacts are destroyed with the AABBs cease to overlap.

So you might gather that there may be contacts created for fixtures that are not touching (just their AABBs). Well, this is correct. It's a "chicken or egg" problem. We don't know if we need a contact object until one is created to analyze the collision. We could delete the contact right away if the shapes are not touching, or we can just wait until the AABBs stop overlapping. Box2D takes the latter approach because it lets the system cache information to improve performance.

* **Contact Class**

As mentioned before, the contact class is created and destroyed by Box2D. Contact objects are not created by the user. However, you are able to access the contact class and interact with it.

You can access the raw contact manifold:

b2Manifold\* GetManifold();

const b2Manifold\* GetManifold() const;

You can potentially modify the manifold, but this is generally not supported and is for advanced usage.

There is a helper function to get the b2WorldManifold:

void GetWorldManifold(b2WorldManifold\* worldManifold) const;

This uses the current positions of the bodies to compute world positions of the contact points.

Sensors do not create manifolds, so for them use:

bool touching = sensorContact->IsTouching();

This function also works for non-sensors.

You can get the fixtures from a contact. From those you can get the bodies.

b2Fixture\* fixtureA = myContact->GetFixtureA();

b2Body\* bodyA = fixtureA->GetBody();

MyActor\* actorA = (MyActor\*)bodyA->GetUserData();

You can disable a contact. This only works inside the b2ContactListener::PreSolve event, discussed below.

* **Accessing Contacts**

You can get access to contacts in several ways. You can access the contacts directly on the world and body structures. You can also implement a contact listener.

You can iterate over all contacts in the world:

for (b2Contact\* c = myWorld->GetContactList(); c; c = c->GetNext())

{

// process c

}

You can also iterate over all the contacts on a body. These are stored in a graph using a contact edge structure.

for (b2ContactEdge\* ce = myBody->GetContactList(); ce; ce = ce->next)

{

b2Contact\* c = ce->contact;

// process c

}

You can also access contacts using the contact listener that is described below.

**Caution**

Accessing contacts off b2World and b2Body may miss some transient contacts that occur in the middle of the time step. Use b2ContactListener to get the most accurate results.

* **Contact Listener**

You can receive contact data by implementing b2ContactListener. The contact listener supports several events: begin, end, pre-solve, and post-solve.

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 \*/ }

};

**Caution**

Do not keep a reference to the pointers sent to b2ContactListener. Instead make a deep copy of the contact point data into your own buffer. The example below shows one way of doing this.

At run-time you can create an instance of the listener and register it with b2World::SetContactListener. Be sure your listener remains in scope while the world object exists.

* **Begin Contact Event**

これは、2つのフィクスチャーが重なり始めるときに呼び出されます。これは、センサーおよび非センサーに対して呼び出されます。このイベントはタイムステップ内でのみ発生します。

* **End Contact Event**

これは、2つのフィクスチャがオーバーラップしなくなったときに呼び出されます。これは、センサーおよび非センサーに対して呼び出されます。これは、ボディが破棄されるときに呼び出される可能性があるため、このイベントはタイムステップ外で発生する可能性があります。

* **Pre-Solve Event**

これは、衝突検出の後、衝突解決の前に呼び出されます。これにより、現在の構成に基づいて連絡先を無効にすることができます。たとえば、このコールバックを使用してb2Contact :: SetEnabled（false）を呼び出して、片側プラットフォームを実装できます。連絡先はコリジョン処理を通じて毎回再度有効になります。そのため、タイムステップごとに連絡先を無効にする必要があります。 pre-solveイベントは、継続的な衝突検出により、コンタクトごとのタイムステップごとに複数回起動される場合があります。

void PreSolve(b2Contact\* contact, const b2Manifold\* oldManifold)

{

b2WorldManifold worldManifold;

contact->GetWorldManifold(&worldManifold);

if (worldManifold.normal.y < -0.5f)

{

contact->SetEnabled(false);

}

}

The pre-solve event is also a good place to determine the point state and the approach velocity of collisions.

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 Event**

解析後イベントは、衝突インパルスの結果を収集できる場所です。衝動を気にしない場合は、おそらく事前解決イベントを実装する必要があります。

コンタクトコールバック内の物理世界を変更するゲームロジックを実装するのは魅力的です。たとえば、ダメージを与える衝突が発生し、関連するアクタとその剛体を破壊しようとする場合があります。ただし、Box2Dでは、Box2Dが現在処理しているオブジェクトを破棄し、孤立したポインターを引き起こす可能性があるため、コールバック内の物理世界を変更することはできません。

連絡先を処理するための推奨される方法は、気になるすべての連絡先データをバッファし、タイムステップ後に処理することです。タイムステップの直後に、常に連絡先を処理する必要があります。そうしないと、他のクライアントコードによって物理世界が変更され、コンタクトバッファが無効になる可能性があります。コンタクトバッファを処理するとき、物理世界を変更できますが、コンタクトポイントバッファに格納されたポインタを孤立させないように注意する必要があります。テストベッドには、孤立したポインターから安全なサンプルの接点処理があります。

CollisionProcessingテストのこのコードは、コンタクトバッファーを処理するときに、孤立したボディを処理する方法を示しています。以下は抜粋です。リストのコメントを必ず読んでください。このコードは、すべての接点がb2ContactPoint配列m\_pointsにバッファリングされていることを前提としています。

// 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\* bodyA = point->fixtureA->GetBody();

b2Body\* bodyB = point->FixtureB->GetBody();

float32 massA = bodyA->GetMass();

float32 massB = bodyB->GetMass();

if (massA > 0.0f && massB > 0.0f)

{

if (massB > massA)

{

nuke[nukeCount++] = bodyA;

}

else

{

nuke[nukeCount++] = bodyB;

}

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);

}

* **Contact Filtering**

Often in a game you don't want all objects to collide. For example, you may want to create a door that only certain characters can pass through. This is called contact filtering, because some interactions are filtered out.

Box2D allows you to achieve custom contact filtering by implementing a b2ContactFilter class. This class requires you to implement a ShouldCollide function that receives two b2Shape pointers. Your function returns true if the shapes should collide.

The default implementation of ShouldCollide uses the b2FilterData defined in Chapter 6, Fixtures.

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;

}

At run-time you can create an instance of your contact filter and register it with b2World::SetContactFilter. Make sure your filter stays in scope while the world exists.

MyContactFilter filter;

world->SetContactFilter(&filter);

// filter remains in scope …

* **World Class**
* **About**

The b2World class contains the bodies and joints. It manages all aspects of the simulation and allows for asynchronous queries (like AABB queries and ray-casts). Much of your interactions with Box2D will be with a b2World object.

* **Creating and Destroying a World**

Creating a world is fairly simple. You just need to provide a gravity vector and a Boolean indicating if bodies can sleep. Usually you will create and destroy a world using new and delete.

b2World\* myWorld = new b2World(gravity, doSleep);

... do stuff ...

delete myWorld;

* **Using a World**

The world class contains factories for creating and destroying bodies and joints. These factories are discussed later in the sections on bodies and joints. There are some other interactions with b2World that I will cover now.

* **Simulation**

The world class is used to drive the simulation. You specify a time step and a velocity and position iteration count. For example:

float32 timeStep = 1.0f / 60.f;

int32 velocityIterations = 10;

int32 positionIterations = 8;

myWorld->Step(timeStep, velocityIterations, positionIterations);

タイムステップの後、身体と関節の情報を調べることができます。ほとんどの場合、アクターを更新してレンダリングできるように、ボディから位置を取得します。ゲームループ内のどこでもタイムステップを実行できますが、物事の順序に注意する必要があります。たとえば、そのフレーム内の新しいボディの衝突結果を取得するには、タイムステップの前にボディを作成する必要があります。

上記のHelloWorldチュートリアルで説明したように、固定の時間ステップを使用する必要があります。より大きな時間ステップを使用することにより、低フレームレートのシナリオでパフォーマンスを改善できます。ただし、通常は、1/30秒以下のタイムステップを使用する必要があります。通常、1/60秒のタイムステップは、高品質のシミュレーションを提供します。

反復回数は、制約ソルバーが世界中のすべての接触とジョイントをスイープする回数を制御します。反復回数が多いほど、より良いシミュレーションが得られます。ただし、小さなタイムステップを大量の反復回数と引き換えにしないでください。 60Hzおよび10回の反復は、30Hzおよび20回の反復よりもはるかに優れています。

踏み込んだ後、体に加えた力をすべて取り除く必要があります。これは、コマンドb2World :: ClearForcesで実行されます。これにより、同じ力場で複数のサブステップを実行できます。

myWorld->ClearForces();

* **Exploring the World**

世界は、身体、接触、関節の容器です。ボディ、コンタクト、ジョイントリストを世界中から取得して、それらを反復処理できます。たとえば、次のコードは世界のすべての身体を目覚めさせます。

for (b2Body\* b = myWorld->GetBodyList(); b; b = b->GetNext())

{

b->SetAwake(true);

}

残念ながら、実際のプログラムはより複雑になる可能性があります。たとえば、次のコードは壊れています。

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.

}

}

体が破壊されるまで、すべてがうまくいきます。ボディが破棄されると、その次のポインターは無効になります。そのため、b2Body :: GetNext（）の呼び出しはガベージを返します。これに対する解決策は、ボディを破壊する前に次のポインター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);

}

}

これにより、現在のボディが安全に破壊されます。ただし、複数のボディを破壊する可能性のあるゲーム関数を呼び出したい場合があります。この場合、非常に注意する必要があります。ソリューションはアプリケーション固有ですが、便宜上、問題を解決する1つの方法を示します。

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();

}

}

}

明らかにこの作業を行うには、GameCrazyBodyDestroyerが何を破壊したかについて正直でなければなりません。

* **AABB Queries**

場合によっては、領域内のすべての形状を決定したいことがあります。 b2Worldクラスには、ブロードフェーズデータ構造を使用するこのための高速log（N）メソッドがあります。世界座標でのAABBとb2QueryCallbackの実装を提供します。世界は、AABBがクエリAABBと重複する各フィクスチャでクラスを呼び出します。クエリを続行するにはtrueを返し、そうでない場合はfalseを返します。たとえば、次のコードは、指定されたAABBと交差する可能性があるすべてのフィクスチャを検索し、関連するすべてのボディを起動します。

class MyQueryCallback : public b2QueryCallback

{

public:

bool ReportFixture(b2Fixture\* fixture)

{

b2Body\* body = fixture->GetBody();

body->SetAwake(true);

// 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);

You cannot make any assumptions about the order of the callbacks.

* **Ray Casts**

レイキャストを使用して、視線の確認、射撃などを行うことができます。コールバッククラスを実装し、開始点と終了点を指定することで、レイキャストを実行します。ワールドクラスは、各フィクスチャがレイにヒットしてクラスを呼び出します。コールバックには、フィクスチャ、交差点、単位法線ベクトル、および光線に沿った小数距離が提供されます。コールバックの順序について仮定することはできません。

分数を返すことにより、レイキャストの継続を制御します。ゼロの端数を返すことは、レイキャストを終了する必要があることを示します。 1の端数は、ヒットが発生していないかのようにレイキャストを続行する必要があることを示します。引数リストから分数を返すと、レイは現在の交点にクリップされます。したがって、適切な割合を返すことにより、任意の形状をレイキャスト、すべての形状をレイキャスト、または最も近い形状をレイキャストできます。

フィクスチャをフィルタリングするために、-1の端数を返すこともできます。その後、フィクスチャが存在しないかのようにレイキャストが進行します。

以下に例を示します。

// 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);

**Caution**

Due to round-off errors, ray casts can sneak through small cracks between polygons in your static environment. If this is not acceptable in your application, please enlarge your polygons slightly.

void SetLinearVelocity(const b2Vec2& v);

b2Vec2 GetLinearVelocity() const;

void SetAngularVelocity(float32 omega);

float32 GetAngularVelocity() const;

* **Forces and Impulses**

You can apply forces, torques, and impulses to a body. When you apply a force or an impulse, you provide a world point where the load is applied. This often results in a torque about the center of mass.

void ApplyForce(const b2Vec2& force, const b2Vec2& point);

void ApplyTorque(float32 torque);

void ApplyLinearImpulse(const b2Vec2& impulse, const b2Vec2& point);

void ApplyAngularImpulse(float32 impulse);

Applying a force, torque, or impulse wakes the body. Sometimes this is undesirable. For example, you may be applying a steady force and want to allow the body to sleep to improve performance. In this case you can use the following code.

if (myBody->IsAwake() == true)

{

myBody->ApplyForce(myForce, myPoint);

}

* **Coordinate Transformations**

The body class has some utility functions to help you transform points and vectors between local and world space. If you don't understand these concepts, please read "Essential Mathematics for Games and Interactive Applications" by Jim Van Verth and Lars Bishop. These functions are efficient (when inlined).

b2Vec2 GetWorldPoint(const b2Vec2& localPoint);

b2Vec2 GetWorldVector(const b2Vec2& localVector);

b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);

b2Vec2 GetLocalVector(const b2Vec2& worldVector);

* **Lists**

You can iterate over a body's fixtures. This is mainly useful if you need to access the fixture's user data.

for (b2Fixture\* f = body->GetFixtureList(); f; f = f->GetNext())

{

MyFixtureData\* data = (MyFixtureData\*)f->GetUserData();

... do something with data ...

}

You can similarly iterate over the body's joint list.

The body also provides a list of associated contacts. You can use this to get information about the current contacts. Be careful, because the contact list may not contain all the contacts that existed during the previous time step.

* **Loose Ends**
* **User Data**

b2Fixture、b2Body、およびb2Jointクラスを使用すると、ユーザーデータをvoidポインターとして添付できます。これは、Box2Dデータ構造を調べており、それらがゲームエンジンのオブジェクトにどのように関連しているかを判断する場合に便利です。

たとえば、通常、actorの剛体にactorポインターをアタッチします。これにより、循環参照が設定されます。actorがいる場合は、身体を取得できます。体があれば、actorを得ることができます。

GameActor\* actor = GameCreateActor();

b2BodyDef bodyDef;

bodyDef.userData = actor;

actor->body = box2Dworld->CreateBody(&bodyDef);

Here are some examples of cases where you would need the user data:

* Applying damage to an actor using a collision result.
* Playing a scripted event if the player is inside an axis-aligned box.
* Accessing a game structure when Box2D notifies you that a joint is going to be destroyed.

Keep in mind that user data is optional and you can put anything in it. However, you should be consistent. For example, if you want to store an actor pointer on one body, you should keep an actor pointer on all bodies. Don't store an actor pointer on one body, and a foo pointer on another body. Casting an actor pointer to a foo pointer may lead to a crash.

User data pointers are NULL by default.

For fixtures you might consider defining a user data structure that lets you store game specific information, such as material type, effects hooks, sound hooks, etc.

struct FixtureUserData

{

int materialIndex;

…

};

FixtureUserData myData = new FixtureUserData;

myData->materialIndex = 2;

b2FixtureDef fixtureDef;

fixtureDef.shape = &someShape;

fixtureDef.userData = myData;

b2Fixture\* fixture = body->CreateFixture(&fixtureDef);

…

delete fixture->GetUserData();

fixture->SetUserData(NULL);

body->DestroyFixture(fixture);

* **Implicit Destruction**

Box2D doesn't use reference counting. So if you destroy a body it is really gone. Accessing a pointer to a destroyed body has undefined behavior. In other words, your program will likely crash and burn. To help fix these problems, the debug build memory manager fills destroyed entities with FDFDFDFD. This can help find problems more easily in some cases.

If you destroy a Box2D entity, it is up to you to make sure you remove all references to the destroyed object. This is easy if you only have a single reference to the entity. If you have multiple references, you might consider implementing a handle class to wrap the raw pointer.

Often when using Box2D you will create and destroy many bodies, shapes, and joints. Managing these entities is somewhat automated by Box2D. If you destroy a body then all associated shapes and joints are automatically destroyed. This is called implicit destruction.

When you destroy a body, all its attached shapes, joints, and contacts are destroyed. This is called implicit destruction. Any body connected to one of those joints and/or contacts is woken. This process is usually convenient. However, you must be aware of one crucial issue:

**Caution**

When a body is destroyed, all fixtures and joints attached to the body are automatically destroyed. You must nullify any pointers you have to those shapes and joints. Otherwise, your program will die horribly if you try to access or destroy those shapes or joints later.

To help you nullify your joint pointers, Box2D provides a listener class named b2DestructionListener that you can implement and provide to your world object. Then the world object will notify you when a joint is going to be implicitly destroyed

Note that there no notification when a joint or fixture is explicitly destroyed. In this case ownership is clear and you can perform the necessary cleanup on the spot. If you like, you can call your own implementation of b2DestructionListener to keep cleanup code centralized.

Implicit destruction is a great convenience in many cases. It can also make your program fall apart. You may store pointers to shapes and joints somewhere in your code. These pointers become orphaned when an associated body is destroyed. The situation becomes worse when you consider that joints are often created by a part of the code unrelated to management of the associated body. For example, the testbed creates a b2MouseJoint for interactive manipulation of bodies on the screen.

Box2D provides a callback mechanism to inform your application when implicit destruction occurs. This gives your application a chance to nullify the orphaned pointers. This callback mechanism is described later in this manual.

You can implement a b2DestructionListener that allows b2World to inform you when a shape or joint is implicitly destroyed because an associated body was destroyed. This will help prevent your code from accessing orphaned pointers.

class MyDestructionListener : public b2DestructionListener

{

void SayGoodbye(b2Joint\* joint)

{

// remove all references to joint.

}

};

You can then register an instance of your destruction listener with your world object. You should do this during world initialization.

myWorld->SetListener(myDestructionListener);

* **Pixels and Coordinate Systems**

Recall that Box2D uses MKS (meters, kilograms, and seconds) units and radians for angles. You may have trouble working with meters because your game is expressed in terms of pixels. To deal with this in the testbed I have the whole *game* work in meters and just use an OpenGL viewport transformation to scale the world into screen space.

float lowerX = -25.0f, upperX = 25.0f, lowerY = -5.0f, upperY = 25.0f;

gluOrtho2D(lowerX, upperX, lowerY, upperY);

If your game must work in pixel units then you should convert your length units from pixels to meters when passing values from Box2D. Likewise you should convert the values received from Box2D from meters to pixels. This will improve the stability of the physics simulation.

You have to come up with a reasonable conversion factor. I suggest making this choice based on the size of your characters. Suppose you have determined to use 50 pixels per meter (because your character is 75 pixels tall). Then you can convert from pixels to meters using these formulas:

xMeters = 0.02f \* xPixels;

yMeters = 0.02f \* yPixels;

In reverse:

xPixels = 50.0f \* xMeters;

yPixels = 50.0f \* yMeters;

You should consider using MKS units in your game code and just convert to pixels when you render. This will simplify your game logic and reduce the chance for errors since the rendering conversion can be isolated to a small amount of code.

If you use a conversion factor, you should try tweaking it globally to make sure nothing breaks. You can also try adjusting it to improve stability.

* **Debug Drawing**

b2DebugDrawクラスを実装して、物理世界の詳細な描画を取得できます。利用可能なエンティティは次のとおりです。

* 形状の輪郭
* 共同接続
* 広相の軸に沿った境界ボックス（AABB）
* mass重心



これは、データに直接アクセスするよりも、これらの物理エンティティを描画する好ましい方法です。その理由は、必要なデータの多くが内部のものであり、変更される可能性があるためです。

テストベッドは、デバッグ描画機能と接触リスナーを使用して物理エンティティを描画するため、デバッグ描画の実装方法と接触点の描画方法の主要な例として機能します。

* **Limitations**

Box2Dは、いくつかの近似を使用して、剛体の物理を効率的にシミュレートします。これにはいくつかの制限があります。

現在の制限は次のとおりです。

* heavy重い物体をはるかに軽い物体の上に積み重ねることは安定していません。質量比が10：1を超えると、安定性が低下します。
* joint軽いボディが重いボディをサポートしている場合、ジョイントで接続されたボディのチェーンが伸びることがあります。たとえば、軽量ボディのチェーンに接続するレッキングボールは安定しない場合があります。質量比が10：1を超えると、安定性が低下します。
* typically形状と形状の衝突では、通常約0.5cmの傾斜があります。
* ⦁連続衝突はジョイントを処理しません。そのため、高速で移動するオブジェクトでジョイントが伸びる場合があります。
* ⦁Box2Dはシンプレクティックオイラー積分スキームを使用します。発射体の放物線運動を再現せず、1次精度しかありません。ただし、高速で安定性に優れています。
* ⦁Box2Dは、反復ソルバーを使用してリアルタイムのパフォーマンスを提供します。厳密な衝突やピクセル完璧な精度は得られません。反復回数を増やすと、精度が向上します。
* **References**

Erin CattoのGDCチュートリアル：<http://code.google.com/p/box2d/downloads/list>

インタラクティブ3D環境での衝突検出、Gino van den Bergen、2004

リアルタイム衝突検出、Christer Ericson、2005