Box2D v2.1.0 User Manual

Copyright © 2007-2010 Erin Catto

[Chapter 1 Introduction 2](#_Toc256461568)

[Chapter 2 Hello Box2D 5](#_Toc256461569)

[Chapter 3 API Design 11](#_Toc256461570)

[Chapter 4 Shapes 14](#_Toc256461571)

[Chapter 5 The World 17](#_Toc256461572)

[Chapter 6 Bodies 22](#_Toc256461573)

[Chapter 7 Fixtures 29](#_Toc256461574)

[Chapter 8 Joints 33](#_Toc256461575)

[Chapter 9 Contacts 41](#_Toc256461576)

[Chapter 10 Loose Ends 46](#_Toc256461577)

[Chapter 11 Settings 48](#_Toc256461578)

[Chapter 12 Debug Drawing 49](#_Toc256461579)

# Introduction

## About

Box2D is a 2D rigid body simulation library for games. Programmer's can use it in their games to make objects move in believable ways and make the game world more interactive. From the game's point of view a physics engine is just a system for procedural animation. Rather than paying (or begging) an animator to move your actors around, you can let Sir Isaac Newton do the directing.

Box2D is written in portable C++. Most of the types defined in the engine begin with the b2 prefix. Hopefully this is sufficient to avoid name clashing with your game engine.

## Prerequisites

In this manual I'll assume you are familiar with basic physics concepts, such as mass, force, torque, and impulses. If not, please first consult the many tutorials provided by Chris Hecker and David Baraff (Google these names). You do not need to understand their tutorials in great detail, but they do a good job of laying out the basic concepts that will help you use Box2D.

Wikipedia is also an excellent source of physics and mathematics knowledge. In some ways it is more useful than Google, because it has carefully crafted content.

This is not a prerequisite, but if you are curious about the about the inner workings of Box2D, you can look at these articles.

Since Box2D is written in C++, you are expected to be experienced in C++ programming. Box2D should not be your first C++ programming project. You should be comfortable with compiling, linking, and debugging.

**Caution**

Box2D should not be your first C++ project. Please learn C++ programming, compiling, linking, and debugging before working with Box2D. There are many resources for this on the net.

## About this Manual

This manual covers the majority of the Box2D API. However, not every aspect is covered. You are encouraged to look at the testbed included with Box2D to learn more. Also, the Box2D code base has comments formatted for Doxygen, so it is easy to create a hyper-linked API document.

## Feedback and Reporting Bugs

If you have feedback about anything related to Box2D, please leave a comment in the forum. This is also a great place for community discussion.

Box2D issues are tracked using a Google code project. This is a great way to track issues and ensures that your issue will not be lost in the depths of the forums.

File issues here: <http://code.google.com/p/box2d/>

You can help to ensure your issue gets fixed if you provide sufficient detail. A testbed example that reproduces the problem is ideal.

## Core Concepts

Box2D works with several fundamental objects. We briefly define these objects here and more details are given later in this document.

#### rigid body

A chunk of matter that is so strong that the distance between any two bits of matter on the chunk is completely constant. They are hard like a diamond. In the following discussion we use body interchangably with rigid body.

#### shape

A 2D geometrical object, such as a circle or polygon.

#### fixture

A fixture binds a shape to a body and adds material properties such as density, friction, and restitution.

#### constraint

A constraint is a physical connection that removes degrees of freedom from bodies. In 2D a body has 3 degrees of freedom (two translation coordinates and one rotation coordinate). If we take a body and pin it to the wall (like a pendulum) we have constrained the body to the wall. At this point the body can only rotate about the pin, so the constraint has removed 2 degrees of freedom.

#### contact constraint

A special constraint designed to prevent penetration of rigid bodies and to simulate friction and restitution. You do not create contact constraints, they are created automatically by Box2D.

#### joint

This is a contraint used to hold two or more bodies together. Box2D supports several joint types: revolute, prismatic, distance, and more. Some joints may have limits and motors.

#### joint limit

A joint limit restricts the range of motion of a joint. For example, the human elbow only allows a certain range of angles.

#### joint motor

A joint motor drives the motion of the connected bodies according to the joint's degrees of freedom. For example, you can use a motor to drive the rotation of an elbow.

#### world

A physics world is a collection of bodies, fixtures, and constraints that interact together. Box2D supports the creation of multiple worlds, but this is usually not necessary or desirable.

# Hello Box2D

In the distribution of Box2D is a Hello World project. The program creates a large ground box and a small dynamic box. This code does not contain any graphics, so prepare to be underwelmed. All you will see is text output in the console of the box's position over time.

This is a good example of how to get up and running with Box2D, more than the testbed.

## Creating a World

Every Box2D program begins with the creation of a b2World object. b2World is the physics hub that manages memory, objects, and simulation. You can allocate the physics world on the stack, heap, or data section. The decision is up to you.

It is easy to create a Box2D world. First, we define the gravity vector. Yes, you can make gravity go sideways (or you could just rotate your monitor). Also we tell the world to allow bodies to sleep when they come to rest. A sleeping body doesn't require any simulation.

b2Vec2 gravity(0.0f, -10.0f);

bool doSleep = true;

Now we create the world object. Note that we are creating the world on the stack, so the world must remain in scope.

b2World world(gravity, doSleep);

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

## Creating a Ground Box

Bodies are built using the following steps:

1. Define a body with a position, damping, etc.
2. Use the world object to create the body.
3. Define fixtures with a shape, friction, density, etc.
4. Create fixtures on the body.

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. The ground body is created as a static body. Static bodies don't collide with other static bodies and are immovable. Box2D determines that a body is static when it has zero mass. Bodies have zero mass by default, therefore they are static by default.

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 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.

groundBody->CreateFixture(&groundBox);

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. This need for static bodies is done to make the Box2D code more uniform internally, reducing the number of potential bugs.

You might notice a pattern here. Most Box2D types are prefixed with b2. This is done to reduce the chance for naming conflicts with your code.

## Creating a Dynamic Body

So now we have a ground body. We can use the same technique to create a dynamic body. The main difference, besides dimensions, is that we must establish the dynamic body's mass properties.

First we create the body using CreateBody. By default bodies are static, so we should set the b2BodyType at construction time to make the body dynamic.

b2BodyDef bodyDef;

bodyDef.type = b2\_dynamicBody;

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

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

Caution

You must set the body type to b2\_dynamicBody if you want the body to move in response to forces.

Next we create and attach a polygon shape using a fixture definition. First we create a box shape:

b2PolygonShape dynamicBox;

dynamicBox.SetAsBox(1.0f, 1.0f);

Next we create a fixture definition using the box. Notice that we set density to 1. The default density is zero. Also, the friction on the shape is set to 0.3.

b2FixtureDef fixtureDef;

fixtureDef.shape = &dynamicBox;

fixtureDef.density = 1.0f;

fixtureDef.friction = 0.3f;

Using the fixture definition we can now create the fixture. This automatically updates the mass of the body. You can add as many fixtures as you like to a body. Each one contributes to the total mass.

body->CreateFixture(&fixtureDef);

That's it for initialization. We are now ready to begin simulating.

## Simulating the World (of Box2D)

So we have initialized the ground box and a dynamic box. Now we are ready to set Newton loose to do his thing. We just have a couple more issues to consider.

Box2D uses a computational algorithm called an integrator. Integrators simulate the physics equations at discrete points of time. This goes along with the traditional game loop where we essentially have a flip book of movement on the screen. So we need to pick a time step for Box2D. Generally physics engines for games like a time step at least as fast as 60Hz or 1/60 seconds. You can get away with larger time steps, but you will have to be more careful about setting up the definitions for your world. We also don't like the time step to change much. A variable time step produces variable results, which makes it difficult to debug. So don't tie the time step to your frame rate (unless you really, really have to). Without further ado, here is the time step.

float32 timeStep = 1.0f / 60.0f;

In addition to the integrator, Box2D also uses a larger bit of code called a constraint solver. The constraint solver solves all the constraints in the simulation, one at a time. A single constraint can be solved perfectly. However, when we solve one constraint, we slightly disrupt other constraints. To get a good solution, we need to iterate over all constraints a number of times.

There are two phases in the constraint solver: a velocity phase and a position phase. In the velocity phase the solver computes the impulses necessary for the bodies to move correctly. In the position phase the solver adjusts the positions of the bodies to reduce overlap and joint detachment. Each phase has its own iteration count. In addition, the position phase may exit iteraions early if the errors are small.

The suggested iteration count for Box2D is 10 for both velocity and position. You can tune this number to your liking, just keep in mind that this has a trade-off between speed and accuracy. Using fewer iterations increases performance but accuracy suffers. Likewise, using more iterations decreases performance but improves the quality of your simulation. For this simple example, we don't need many iterations. Here are our chosen iteration counts.

int32 velocityIterations = 6;

int32 positionIterations = 2;

Note that the time step and the iteration count are completely unrelated. An iteration is not a sub-step. One iteration is a single pass over all the constraints within a time step. You can have multiple passes over the constraints within a single time step.

We are now ready to begin the simulation loop. In your game the simulation loop can be merged with your game loop. In each pass through your game loop you call b2World::Step. Just one call is usually enough, depending on your frame rate and your physics time step. After stepping, you should call b2World::ClearForces to clear any forces you applied to the bodies.

The Hello World program was designed to be dead simple, so it has no graphical output. Rather that being utterly boring by producing no output, the code prints out the position and rotation of the dynamic body. Yay! Here is the simulation loop that simulates 60 time steps for a total of 1 second of simulated time.

for (int32 i = 0; i < 60; ++i)

{

world.Step(timeStep, velocityIterations, positionIterations);

world.ClearForces();

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 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.

# API Design

## 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). 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. You do have to allocate a b2World. 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, ever, 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 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.

## Factories and Definitions

As mentioned above, memory management plays a central role in the design of the Box2D API. So when you create a b2Body or a b2Joint, you need to call the factory functions on b2World.

There are creation functions:

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

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

And there are corresponding destruction functions:

void b2World::DestroyBody(b2Body\* body)

void b2World::DestroyJoint(b2Joint\* joint)

When you create a body or joint, you need to provide a definition or def for short. These definitions contain all the information needed to build the body or joint. By using this approach we can prevent construction errors, keep the number of function parameters small, provide sensible defaults, and reduce the number of accessors.

Since fixtures must be parented to a body, they are created and destroyed using a factory method on b2Body:

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

void b2Body::DestroyFixture(b2Fixture\* fixture)

There is also shortcut to create a fixture directly from the shape and density.

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

Factories do not retain references to the definitions. So you can create definitions on the stack and keep them in temporary resources.

## Units

Box2D works with floating point numbers, so some tolerances have to be used to make Box2D perform well. These tolerances have been tuned to work well with meters-kilogram-second (MKS) units. In particular, Box2D has been tuned to work well with moving objects between 0.1 and 10 meters. So this means objects between soup cans and buses in size should work well. Static objects may be up to 50 meters without too much trouble.

Being a 2D physics engine, it is tempting to use pixels as your units. Unfortunately this will lead to a poor simulation and possibly weird behavior. An object of length 200 pixels would be seen by Box2D as the size of a 45 story building. Imagine trying to simulate the movement of a high-rise building with an engine that is tuned to simulate ragdolls and barrels. It isn't pretty.

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 examples do this by using an OpenGL viewport transform.

It is best to think of Box2D bodies as moving billboards upon which you attach your artwork. The billboard may move in a unit system of meters, but you can convert that to pixel coordinates with a simple scaling factor. You can then use those pixel coordinates to place your sprite, etc.

Box2D uses radians for angles. The body rotation is stored in radians and may grow unbounded. Consider normalizing the angle of your bodies if the magnitude of the angle becomes too large (use b2Body::SetAngle).

## User Data

The b2Fixture, b2Body, and b2Joint classes allow you to attach user data as a void pointer. This is handy when you are examining Box2D data structures and you want to determine how they relate to the data structures in your game engine.

For example, it is typical to attach an actor pointer to the rigid body on that actor. This sets up a circular reference. If you have the actor, you can get the body. If you have the body, you can get the actor.

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.

## Settings

The file b2Settings.h gathers all the constants used by Box2D. It also allows you to override the memory allocation.

You may modify the constants to suit your application. However, normally the default values work well. Once you become a serious Box2D user, you should read over the comments in b2Settings.h.

## 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.

# Shapes

## About

Shapes describe collision geometry and may be used independently of physics simulation. You may perform several operations with shapes.

## Abstract Shapes

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.

## Circle Shapes

Circle shapes have a position and radius.

Circles are solid discs. You cannot make a hollow circle. However, you can create chains of line segments using polygon shapes.

b2CircleShape circle;

circle.m\_p.Set(1.0f, 2.0f, 3.0f);

circle.m\_radius = 0.5f;

## Polygon Shapes

Polygon shapes are 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. However, you can create line segments using 2 point polygons.



You must create polygons 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.



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.

// 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);

The polygon shape has some custom initialization functions to create boxes and edges (line segments).

void SetAsBox(float32 hx, float32 hy);

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

void SetAsEdge(const b2Vec2& v1, const b2Vec2& v2);

Edges may be chained together create free-form vector geometry. For example, this code creates an approximate sinusoidal curve as a sequence of line segments:

float32 x1 = -20.0f;

float32 y1 = 2.0f \* cosf(x1 / 10.0f \* b2\_pi);

for (int32 i = 0; i < 80; ++i)

{

float32 x2 = x1 + 0.5f;

float32 y2 = 2.0f \* cosf(x2 / 10.0f \* b2\_pi);

b2PolygonShape shape;

shape.SetAsEdge(b2Vec2(x1, y1), b2Vec2(x2, y2));

ground->CreateFixture(&shape, 0.0f);

x1 = x2;

y1 = y2;

}

## Shape Functions

You can retrieve the type of a shape. This is used to cast a generic shape into a specific class.

# The World

## 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 destory 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);

After the time step you can examine your bodies and joints for information. Most likely you will grab the position off the bodies so that you can update your actors and render them. You can perform the time step anywhere in your game loop, but you should be aware of the order of things. For example, you must create bodies before the time step if you want to get collision results for the new bodies in that frame.

As I discussed above in the HelloWorld tutorial, you should use a fixed time step. By using a larger time step you can improve performance in low frame rate scenarios. But generally you should use a time step no larger than 1/30 seconds. A time step of 1/60 seconds will usually deliver a high quality simulation.

The iteration count controls how many times the constraint solver sweeps over all the contacts and joints in the world. More iterations always yields a better simulation. But don't trade a small time step for a large iteration count. 60Hz and 10 iterations is far better than 30Hz and 20 iterations.

After stepping, you should clear any forces you have applied to your bodies. This is done with the command b2World::ClearForces. This lets you take multiple sub-steps with the same force field.

myWorld->ClearForces();

### Exploring the World

The world is a container for bodies, contacts, and joints. You can grab the body, contact, and joint lists off the world and iterate over them. For example, this code wakes up all the bodies in the world:

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

{

b->WakeUp();

}

Unfortunately real programs can be more complicated. For example, the following code is broken:

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.

}

}

Everything goes ok until a body is destroyed. Once a body is destroyed, its next pointer becomes invalid. So the call to b2Body::GetNext() will return garbage. The solution to this is to copy the next pointer before destroying the body.

b2Body\* node = myWorld->GetBodyList();

while (node)

{

b2Body\* b = node;

node = node->GetNext();

GameActor\* myActor = (GameActor\*)b->GetUserData();

if (myActor->IsDead())

{

myWorld->DestroyBody(b);

}

}

This safely destroys the current body. However, you may want to call a game function that may destroy multiple bodies. In this case you need to be very careful. The solution is application specific, but for convenience I'll show one method of solving the problem.

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();

}

}

}

Obviously to make this work, GameCrazyBodyDestroyer must be honest about what it has destroyed.

### AABB Queries

Sometimes you want to determine all the shapes in a region. The b2World class has a fast log(N) method for this using the broad-phase data structure. You provide an AABB in world coordinates and an implementation of b2QueryCallback. The world calls your class with each fixture whose AABB overlaps the query AABB. Return true to continue the query, otherwise return false. For example, the following code finds all the fixtures that potentially intersect a specified AABB and wakes up all of the associated bodies.

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);

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

### Ray Casts

You can use ray casts to do line-of-site checks, fire guns, etc. You perform a ray cast by implementing a callback class and providing the start and end points. The world class calls your class with each fixture hit by the ray. Your callback is provided with the fixture, the point of intersection, the unit normal vector, and the fractional distance along the ray. You cannot make any assumptions about the order of the callbacks.

You control the continuation of the ray cast by returning a fraction. Returning a fraction of zero indicates the ray cast should be terminated. A fraction of one indicates the ray cast should continue as if no hit occurred. If you return the fraction from the argument list, the ray will be clipped to the current intersection point. So you can ray cast any shape, ray cast all shapes, or ray cast the closest shape by returning the appropriate fraction.

You may also return of fraction of -1 to filter the fixture. Then the ray cast will proceed as if the fixture does not exist.

Here is an example:

// 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);

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.

# 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 defintions:

#### b2\_staticBody

A static body has does not move under simulation and behaves as if it has infinite mass. Internally, Box2D stores zero for the mass and the inverse mass. Static bodies can be moved manually by the user. A static body has zero velocity. Static bodies do not collide with other static or kinematic bodies.

#### b2\_kinematicBody

A kinematic body moves under simulation according to its velocity. Kinematic bodies do not respond to forces. They can be moved manually by the user, but normally a kinematic body is moved by setting its velocity. A kinematic body behaves as if it has infinite mass, however, Box2D stores zero for the mass and the inverse mass. Kinematic bodies do not collide with other static or kinematic bodies.

#### b2\_dynamicBody

A dynamic body is fully simulated. They can be moved manually by the user, but normally they move according to forces. A dynamic body can collide with all body types. A dynamic body always has finite, non-zero mass. If you try to set the mass of a dynamic body to zero, it will automatically acquire a mass of one kilogram.

Bodies are the backbone for fixtures. Bodies carry fixtures and move them around in the world. Bodies are always rigid bodies in Box2D. That means that two fixtures attached to the same rigid body never move relative to each other.

Fixtures have collision geometry and density. Normally, bodies acquire their mass properties from the fixtures. However, you can override the mass properties after a body is constructed. This is discussed below.

You usually keep pointers to all the bodies you create. This way you can query the body positions to update the positions of your graphical entities. You should also keep body pointers so you can destroy them when you are done with them.

## Body Definition

Before a body is created you must create a body definition (b2BodyDef). You can create a body definition on the stack or build it into your game's data structures. The choice is up to you.

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;

### 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.

A body has two main points of interest. The first point is the body's origin. Fixtures and joints are attached relative to the body's origin. The second point of interest is the center of mass. The center of mass is determined from mass distribution of the attached shapes or is explicitly set with b2MassData. Much of Box2D's internal computations use the center of mass position. For example b2Body stores the linear velocity for the center of mass.

When you are building the body definition, you may not know where the center of mass is located. Therefore you specify the position of the body's origin. You may also specify the body's angle in radians, which is not affected by the position of the center of mass. If you later change the mass properties of the body, then the center of mass may move on the body, but the origin position does not change and the attached shapes and joints do not move.

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

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

### Damping

Damping is used to reduce the world velocity of bodies. Damping is different than friction because friction only occurs with contact. Damping is not a replacement for friction and the two effects should be used together.

Damping parameters should be between 0 and infinity, with 0 meaning no damping, and infinity meaning full damping. Normally you will use a damping value between 0 and 0.1. I generally do not use linear damping because it makes bodies look floaty.

bodyDef.linearDamping = 0.0f;

bodyDef.angularDamping = 0.01f;

Damping is approximated for stability and performance. At small damping values the damping effect is mostly independent of the time step. At larger damping values, the damping effect will vary with the time step. This is not an issue if you use a fixed time step (recommended).

### Sleep Parameters

What does sleep mean? Well it is expensive to simulate bodies, so the less we have to simulate the better. When a body comes to rest we would like to stop simulating it.

When Box2D determines that a body (or group of bodies) has come to rest, the body enters a sleep state which has very little CPU overhead. If a body is awake and collides with a sleeping body, then the sleeping body wakes up. Bodies will also wake up if a joint or contact attached to them is destroyed. You can also wake a body manually.

The body definition lets you specify whether a body can sleep and whether a body is created sleeping.

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 roation flag causes the rotational inertia and its inverse to be set to zero.

### Bullets

Game simulation usually generates a sequence of images that are played at some frame rate. This is called discrete simulation. In discrete simulation, rigid bodies can move by a large amount in one time step. If a physics engine doesn't account for the large motion, you may see some objects incorrectly pass through each other. This effect is called tunneling.

By default, Box2D uses continuous collision detection (CCD) to prevent dynamic bodies from tunneling through static bodies. This is done by sweeping shapes from their old position to their new positions. The engine looks for new collisions during the sweep and computes the time of impact (TOI) for these collisions. Bodies are moved to their first TOI and then halted for the remainder of the time step.

Normally CCD is not used between dynamic bodies. This is done to keep performance reasonable. In some game scenarios you need dynamic bodies to use CCD. For example, you may want to shoot a high speed bullet at a stack of dynamic bricks. Without CCD, the bullet might tunnel through the bricks.

Fast moving objects in Box2D can be labeled as bullets. Bullets will perform CCD with both static and dynamic bodies. You should decide what bodies should be bullets based on your game design. If you decide a body should be treated as a bullet, use the following setting.

bodyDef.bullet = true;

The bullet flag only affects dynamic bodies.

Box2D performs continuous collision sequentially, so bullets may miss fast moving bodies.

### Activation

You may wish a body to be created but not participate in collision or dynamics. This state is similar to sleeping except the body will not be woken by other bodies and the body's fixtures will not be placed in the broad-phase. This means the body will not participate in collisions, ray casts, etc.

You can create a body in an inactive state and later re-activate it.

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.

### 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.

Bodies can be dynamic or static depending on the mass properties. Both body types use the same creation and destruction methods.

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

... do stuff ...

myWorld->DestroyBody(dynamicBody);

dynamicBody = NULL;

Caution

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.

Static bodies do not move under the influence of other bodies. You may manually move static bodies, but you should be careful so that you don't squash dynamic bodies between two or more static bodies. Friction will not work correctly if you move a static body. Static bodies never collide with static or kinematic bodies. It is faster to attach several shapes to a static body than to create several static bodies with a single shape on each one. Internally, Box2D sets the mass and inverse mass of static bodies to zero. This makes the math work out so that most algorithms don't need to treat static bodies as a special case.

Box2D does not keep a reference to the body definition or any of the data it holds (except user data pointers). So you can create temporary body definitions and reuse the same body definitions.

Box2D allows you to avoid destroying bodies by deleting your b2World object, which does all the cleanup work for you. However, you should be mindful to nullify any body pointers that you keep in your game engine.

When you destroy a body, the attached shapes and joints are automatically destroyed. This has important implications for how you manage shape and joint pointers.

## Using a Body

After creating a body, there are many operations you can perform on the body. These include setting mass properties, accessing position and velocity, applying forces, and transforming points and vectors.

### Mass Data

Every body has a mass (scalar), center of mass (2-vector), and rotational inertia (scalar). For static bodies, the mass and rotational inertia are set to zero. When a body has fixed rotation, its rotational inertia is zero.

Normally the mass properties of a body are established automatically when fixtures are added to the body. You can also adjust the mass of a body at run-time. This is usually done when you have special game scenerios that require altering the mass.

void SetMassData(const b2MassData\* data);

After setting a body's mass directly, you may wish to revert to the natural mass dictated by the fixtures. You can do this with:

void ResetMassData();

The body's mass data is available through the following functions:

float32 GetMass() const;

float32 GetInertia() const;

const b2Vec2& GetLocalCenter() const;

void GetMassData(b2MassData\* data) const;

### State Information

There are many aspects to the body's state. You can access this state data efficiently through the following functions:

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

You can access the position and rotation of a body. This is common when rendering your associated game actor. You can also set the position, although this is less common since you will normally use Box2D to simulate movement.

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

const b2Transform& GetTransform() const;

const b2Vec2& GetPosition() const;

float32 GetAngle() const;

You can access the center of mass position in local and world coordinates. Much of the internal simulation in Box2D uses the center of mass. However, you should normally not need to access it. Instead you will usually work with the body transform. For example, you may have a body that is square. The body origin might be a corner of the square, while the center of mass is located at the center of the square.

const b2Vec2& GetWorldCenter() const;

const b2Vec2& GetLocalCenter() const;

You can access the linear and angular velocity. The linear velocity is for the center of mass. Therefore, the linear velocity may change if the mass properties change.

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 carefule, because the contact list may not contain all the contacts that existed during the previous time step.

# Fixtures

TODO\_ERIN update chapter for fixtures

## About

Fixtures attach collision geometry (shapes) to bodies. Fixtures are also used to define the mass of a body. This lets you specify the density and let Box2D do the work of computing the mass properties.

Fixtures have properties of friction and restitution. Fixtures carry collision filtering information to let you prevent collisions between certain game objects.

Fixtures are always owned by a body. You can attach multiple fixtures to a single body. Shapes are abstract classes so that many types of shapes can be implemented in Box2D. If you are brave, you can implement your own shape type (and collision algorithms).

## The Shape Definition

Shape definitions are used to create shapes. There is common shape data held by b2ShapeDef and specific shape data held by derived classes.

### Friction and Restitution

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. A value of zero turns off friction and a value of one makes the friction strong. When the friction is computed between two shapes, Box2D must combine the friction parameters of the two shapes. This is done with the following formula:

float32 friction;

friction = sqrtf(shape1->friction \* shape2->friction);

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(shape1->restitution, shape2->restitution);

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.

### Density

Box2D optionally computes the mass and rotational inertia of bodies using the mass distribution implied by the attached shapes. Specifying mass directly can often lead to poorly tuned simulations. Therefore, the recommended way of specifying body mass is by setting the shape densities and calling b2Body::SetMassFromShape once all the shapes are attached to the body.

### Filtering

Collision filtering is a system for preventing collision between shapes. 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 shape you can specify which category it belongs to. You also specify what other categories this shape 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:

playerShapeDef.filter.categoryBits = 0x0002;

monsterShapeDef.filter.categoryBits = 0x0004;

playerShape.filter.maskBits = 0x0004;

monsterShapeDef.filter.maskBits = 0x0002;

Collision groups let you specify an integral group index. You can have all shapes 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, shape1 and shape2 always collide, but shape3 and shape4 never collide.

shape1Def.filter.groupIndex = 2;

shape2Def.filter.groupIndex = 2;

shape3Def.filter.groupIndex = -8;

shape4Def.filter.groupIndex = -8;

Collisions between shapes 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 shape on a static body never collides with a shape on another static body.
* Shapes on the same body never collide with each other.
* You can optionally enable/disable collision between shapes on bodies connected by a joint.

Sometimes you might need to change collision filtering after a shape has already been created. You can get and set the b2FilterData structure on an existing shape using b2Shape::GetFilterData and b2Shape::SetFilterData. Box2D caches filtering results, so you must manually re-filter a shape using b2World::Refilter.

### Sensors

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

You can flag any shape as being a sensor. Sensors may be static or dynamic. Remember that you may have multiple shapes per body and you can have any mix of sensors and solid shapes.

myShapeDef.isSensor = true;

### Circle Definitions

b2CircleDef extends b2ShapeDef and adds a radius and a local position.

b2CircleDef def;

def.radius = 1.5f;

def.localPosition.Set(1.0f, 0.0f);

### Polygon Definitions

b2PolyDef is used to implement convex polygons. They are a bit tricky to use correctly, so please read closely. The maximum vertex count is defined by b2\_maxPolyVertices which is currently 8. If you need to use more vertices, you must modify b2\_maxPolyVertices in b2Settings.h.

When you build a polygon definition you must specify the number of vertices you will use. The vertices must be specified in counter-clockwise (CCW) order about the z-axis of a right-handed coordinate system. This might turn out to be clockwise on your screen, depending on your coordinate system conventions.

Polygons must be convex. In other words, each vertex must point outwards to some degree. Finally, you must not overlap any vertices. Box2D will automatically close the loop.

Here is an example of a polygon definition of a triangle:

b2PolygonDef triangleDef;

triangleDef.vertexCount = 3;

triangleDef.vertices[0].Set(-1.0f, 0.0f);

triangleDef.vertices[1].Set(1.0f, 0.0f);

triangleDef.vertices[2].Set(0.0f, 2.0f);

The vertices are defined in the coordinate system of the parent body. If you need to offset a polygon within the parent body, then just offset all the vertices.

For convenience, there are functions to initialize polygons as boxes. You can have either an axis-aligned box centered at the body origin or an oriented box offset from the body origin.

b2PolygonDef alignedBoxDef;

float32 hx = 1.0f; // half-width

float32 hy = 2.0f; // half-height

alignedBoxDef.SetAsBox(hx, hy);

b2PolygonDef orientedBoxDef;

b2Vec2 center(-1.5f, 0.0f);

float32 angle = 0.5f \* b2\_pi;

orientedBoxDef.SetAsBox(hx, hy, center, angle);

## Shape Factory

Shapes are created by initializing a shape definition and then passing the definition to the parent body.

b2CircleDef circleDef;

circleDef.radius = 3.0f;

circleDef.density = 2.5f;

b2Shape\* myShape = myBody->CreateShape(&circleDef);

[optionally store shape pointer somewhere]

This creates the shape and attaches it to the body. You do not need to store the shape pointer since the shape will automatically be destroyed when the parent body is destroyed (see Section 10.2, “Implicit Destruction”).

After you finish adding shapes to a body, you may want to recompute the mass properties of the body based on the child shapes.

myBody->SetMassFromShapes();

This function is expensive, so you should only call it when necessary.

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

myBody->DestroyShape(myShape);

After removing shapes from a body, you may want to call SetMassFromShapes again.

## Using a Shape

There's not much to say here. You can get a shape's type and its parent body. You can also test a point to see if it is contained within the shape. Look at b2Shape.h for details.

# 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. If you want to waste memory, then create a joint between two static bodies. :)

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 constraint rotation correctly via joint limits or a fixed relative angle.

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.

### 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(myBody1, myBody2, worldAnchorOnBody1, worldAnchorOnBody2);

jointDef.collideConnected = true;

### 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(myBody1, myBody2, myBody1->GetWorldCenter());

The revolute joint angle is positive when body2 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 an 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. Have you ever heard the expression:

"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(body1, body2, myBody1->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;

### 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(myBody1, myBody2, myBody1->GetWorldCenter(), worldAxis);

jointDef.lowerTranslation = -5.0f;

jointDef.upperTranslation = 2.5f;

jointDef.enableLimit = true;

jointDef.motorForce = 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.

### 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). Therefore the pulley joint constrains the maximum length that either side can attain. Also, you may want to control the maximum lengths for gameplay reasons. So the maximum lengths improve stability and give you more control.

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);

jointDef.maxLength1 = 18.0f;

jointDef.maxLength2 = 20.0f;

### Gear Joint

If you want to create a sophisticated mechanical contraption you might want to use a 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 requires the you have two bodies connected to ground by a revolute or prismatic joint. You can use any combination of those joint types. Also, Box2D requires that the revolute and prismatic joints were created with the ground as body1.

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), then the gear ratio with have units of length or one over length.

coordinate1 + ratio \* coordinate2 == constant

Here is an example gear joint:

b2GearJointDef jointDef;

jointDef.body1 = myBody1;

jointDef.body2 = myBody2;

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. Please see the testbed and b2MouseJoint.h for details.

## 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 body or 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.body1 = myBody1;

jointDef.body2 = myBody2;

jointDef.anchorPoint = myBody1->GetCenterPosition();

b2RevoluteJoint\* joint = 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 Section 10.2, “Implicit Destruction” for details.

## Using Joints

Many simulations create the joints and don't access them again until they are detroyed. 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\* GetBody1();

b2Body\* GetBody2();

b2Vec2 GetAnchor1();

b2Vec2 GetAnchor2();

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();

### Using Distance Joints

Distance joints don't have motors or limits, so there are no extra runtime methods for distance joints.

### Using Revolute Joints

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.

### Using Prismatic Joints

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);

### Using Pulley Joints

Pully joints provide the current lengths.

float32 GetLength1() const;

float32 GetLength2() const;

### Using Gear Joints

Gear joints don't provide any information beyond the functions defined in b2Joint.

### Using Mouse Joints

The mouse joint is able to manipulate the attached body by updating the target point each time step.

# Contacts

## About

Contacts are objects created by Box2D to manage collision between shapes. There are different kinds of contacts, derived from b2Contact, for managing contact between different kinds of shapes. For example there is a contact class for managing polygon-polygon collision and another contact class for managing circle-circle collision. This is normally not important to you, I just thought you might like to know.

Here is some terminlogy associated with contacts. This terminology is particular to Box2D, but you might find similar terminology in other physics engines.

#### contact point

A contact point is a point where two shapes touch. In reality objects may touch over regions when surfaces touch. Box2D approximates contact with a small number of points.

#### contact normal

A contact normal is a unit vector that points from shape1 to shape2.

#### 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.

#### normal force

Box2D use an iterative contact solver and stores the results with the contact points. You can safely use the normal force to guage the collision intensity. For example, you can use the force to trigger breakables or to play collision sounds.

#### tangent force

The tangent force is the contact solver's estimate of the friction force.

#### 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 shape's AABBs overlap. Sometimes collision filtering will prevent the creation of contacts. Box2D sometimes needs to create a contact even though the collision is filtered. In this case it uses a b2NullContact that prevents collision from occuring. Contacts are destroyed with the AABBs cease to overlap.

So you might gather that there may be contacts created for shapes 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.

## Contact Listener

You can receive contact data by implementing b2ContactListener. This listener reports a contact point when it is created, when it persists for more than one time step, and when it is destroyed. Keep in mind that two shapes may have multiple contact points.

class MyContactListener : public b2ContactListener

{

public:

void Add(const b2ContactPoint\* point)

{

// handle add point

}

void Persist(const b2ContactPoint\* point)

{

// handle persist point

}

void Remove(const b2ContactPoint\* point)

{

// handle remove point

}

void Result(const b2ContactResult\* point)

{

// handle results

}

};

**Caution**

Do not keep a reference to the contact points returned 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.

Continuous physics uses sub-stepping, so a contact point may be added and removed within the same time step. This is normally not a problem, but your code should handle this gracefully.

Contact points are reported immediately when they are added, persisted, or removed. This occurs before the solver is called, so the b2ContactPoint object does not contain the computed impulse. However, the relative velocity at the contact point is provided so that you can estimated the contact impulse. If you implement the Result listener function, you will receive b2ContactResult objects for solid contact points after the solver has been called. These result structures contain the sub-step impulses. Again, due to continuous physics you may receive multiple results per contact point per b2World::Step.

It is tempting to implement game logic that alters the physics world inside a contact callback. For example, you may have a collision that applies damage and try to destroy the associated actor and its rigid body. However, Box2D does not allow you to alter the physics world inside a callback because you might destroy objects that Box2D is currently processing, leading to orphaned pointers.

The recommended practice for processing contact points is to buffer all contact points that you care about and process them after the time step. You should always process the contact points immediately after the time step, otherwise some other client code might alter the physics world, invalidating the contact buffer. When you process the contact point buffer you can alter the physics world, but you still need to be careful that you don't orphan pointers stored in the contact point buffer. The testbed has example contact point processing that is safe from orphaned pointers.

This code from the CollisionProcessing test shows how to handle orphaned bodies when processing the contact buffer. Here is an excerpt. Be sure to read the comments in the listing. This code assumes that all contact points have been buffered in the b2ContactPoint array m\_points.

// 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);

}

## 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(b2Shape\* shape1, b2Shape\* shape2)

{

const b2FilterData& filter1 = shape1->GetFilterData();

const b2FilterData& filter2 = shape2->GetFilterData();

if (filter1.groupIndex == filter2.groupIndex &&

filter1.groupIndex != 0)

{

return filter1.groupIndex > 0;

}

bool collide = (filter1.maskBits & filter2.categoryBits) != 0 &&

(filter1.categoryBits & filter2.maskBits) != 0;

return collide;

}

# Loose Ends

## 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 shapes 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 b2WorldListener that you can implement and provide to your world object. Then the world object will notify you when a joint is going to be implicity destroyed.

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);

# Settings

## About

There are two source files included in Box2D that are supplied specifically for user customization. These are b2Settings.h and b2Settings.cpp.

Box2D works with floating point numbers, so some tolerances have to be used to make Box2D perform well.

## Tolerances

There are many tolerances settings that depend on using MKS units. See Section 3.3, “Units” for a deeper explanation of the unit system. See the doxygen docs for explanation of the individual tolerances.

## Memory Allocation

All of Box2D's memory allocations are funnelled through b2Alloc and b2Free except in the following cases.

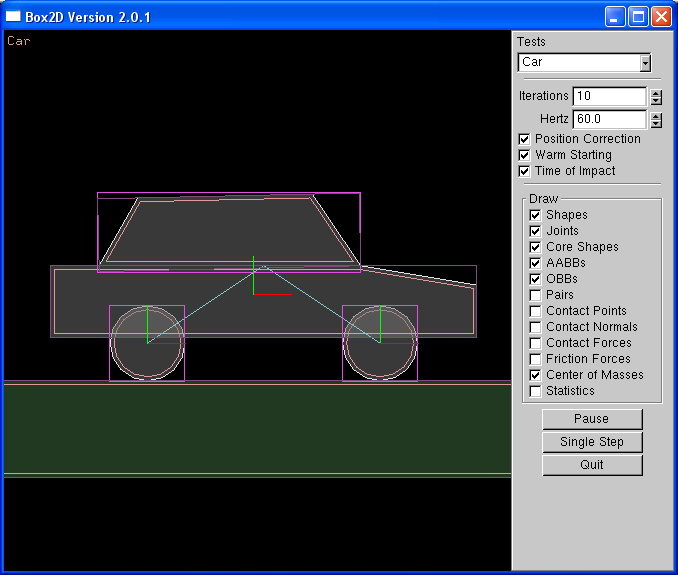
* b2World can be built on the stack or with whatever allocator you like.
* Any other Box2D class that you create without using a factory method. This includes things like callbacks classes and contact point buffers.

Feel free to redirect the allocation by modifying b2Settings.cpp.

# Debug Drawing

You can implement the b2DebugDraw class to get detailed drawing of the physics world. Here are the available entities:

* shape outlines
* joint connectivity
* core shapes (for continuous collision)
* broad-phase axis-aligned bounding boxes (AABBs), including the world AABB
* polygon oriented bounding boxes (OBBs)
* broad-phase pairs (potential contacts)
* center of mass



This is the preferred method of drawing these physics entities, rather than accessing the data directly. The reason is that much of the necessary data is internal and subject to change.

The testbed draws physics entities using the debug draw facility and the contact listener, so it serves as the primary example of how to implement debug drawing as well as how to draw contact points.