# Contents

# Box2DFlash v2.0.2 User manual



# About

Box2DFlash is a 2D rigid body simulation library for games. Programmers can use it in their games to make objects move in believable ways and make the world seem 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. Box2DFlash is written in AS3, and resides in the Box2d namespace. Most of the types defined in the engine begin with the b2 prefix, to match the C++ version.

## 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 inner workings of Box2D, you can look at these articles.

Since Box2DAS3 is written in Actionscript 3, you are expected to be experienced in this. If you come from an AS2 background, you may find it easier to start on an easier project before using Box2D, but it is not impossible.

## 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 piece of collision geometry that is rigidly attached to a body. Shapes have material properties of 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. 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 will never create a contact constraint, they are created automatically by Box2D.

joint
: This is a constraint used to hold two or more bodies together. Box2D supports these joint types: revolute, prismatic, distance, and more. Joints may support **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, shapes, and constraints that interact together. Box2D supports the creation of multiple worlds, but this is usually not necessary or desirable.

# Hello Box2D

This is a small example program that creates a large ground box and a small dynamic box. This code does not contain any graphics, so prepare to be underwelmed. :). The matching documents for this example have yet to be created.

## Creating a World

Every Box2D program begins with the creation of a world object. This is the physics hub that manages objects, and simulation.

To create a world object, first we need to define a bounding box for the world. Box2D uses the bounding box to accelerate collision detection. The size isn't critical, but a better fit will improve performance. It is better to make the box too big than to make it too small.

```
1 var worldAABB:b2AABB = new b2AABB();
2 worldAABB.lowerBound.Set(-100.0, -100.0);
3 worldAABB.upperBound.Set(100.0, 100.0);
```

**Caution**

The **world AABB** should always be bigger then the region where your bodies are located. It is better to make the world AABB too big than too small. If a body reaches the boundary of the world AABB it will be frozen and will stop simulating.

Next 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.

```
1 var gravity:b2Vec2 = new b2Vec2 (0.0, -10.0);
2 var doSleep:Boolean = true;
```

Now we create the world object.

```
1 var world:b2World = new b2World(worldAABB, 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 shapes with geometry, friction, density, etc.
4. Create shapes on the body.
5. Optionally adjust the body's mass to match the attached shapes.

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.

```
1 var groundBodyDef:b2BodyDef = new b2BodyDef();
2 groundBodyDef.position.Set(0.0, -10.0);
```

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.

```
1 var groundBody:b2Body = world.CreateBody(groundBodyDef);
```

For step 3 we create a ground polygon definition. 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.

```
1 var groundShapeDef:b2PolygonDef = new b2PolygonDef();
2 groundShapeDef.SetAsBox(50.0, 10.0);
```

The *SetAsBox* function takes the half-width and half-height. 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. However, it is possible to change unit systems, as discussed later in this document.

We finish the ground body in step 4 by creating the ground polygon shape on the ground body.

```
1 groundBody.CreateShape(groundShapeDef);
```

Again, Box2D does not keep a reference to the shape or body definitions. It copies the data into the *b2Body* structure.

Note that every shape must have a parent body, even shapes that are static. However, you can attach all static shapes 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*.

```
var bodyDef:b2BodyDef = new b2BodyDef();
bodyDef.position.Set(0.0, 4.0);

var body:b2Body = world.CreateBody(bodyDef);
```

Next we create and attach a polygon shape. Notice that we set density to 1. The default density is zero. Also, the friction on the shape is set to 0.3. Once the shape is attached, we instruct the body to compute it's mass properties from the attached shapes using the method *SetMassFromShapes*. This gives you a hint that you can attach more than one shape per body. If the computed mass is zero, then the body becomes truly static. Bodies have a mass of zero by default, that's why we didn't need to call *SetMassFromShapes* for the ground body.

```
var shapeDef:b2PolygonDef = new b2PolygonDef();
shapeDef.SetAsBox(1.0, 1.0);
shapeDef.density = 1.0;
shapeDef.friction = 0.3;
body.CreateShape(shapeDef);
body.SetMassFromShapes();
```

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 bit of numerical code 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. 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.

```
var timeStep:Number = 1.0 / 60.0;
```

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. The suggested iteration count for Box2D is 10. 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. Here is our chosen iteration count.

```
var iterations:Number = 10;
```

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 withing 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.

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 (var i:Number = 0; i < 60; ++i)

{
    world.Step(timeStep, iterations);
    var position:b2Vec2 = body.GetPosition();
    var angle:Number = body.GetAngle();
    trace(position.x +','+ position.y +','+ angle);

}
```

## The Testbed

Once you have conquered the HelloWorld example, you should start looking at Box2DAS3's testbed. The testbed is a unit-testing framework and demo environment. Here are some of the features:

- Mouse picking of shapes attached to dynamic bodies.
- Extensible set of tests.

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.

# API Design

## Memory Management

Box2dFlash is a straight port of Box2D, a C++ library. Hence in many places, memory is managed precisely, rather than relying on the garbage collector. This helps reduce stutter. Objects are recycled, hence the many Create/Destroy methods.

## 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:

```
b2World.CreateBody(def:b2BodyDef):b2Body
b2World.CreateJoint(def:b2JointDef):b2Joint
```

And there are corresponding destruction functions:

```
b2World.DestroyBody(body:b2Body):void

b2World.DestroyJoint(joint:b2Joint):void
```

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

```
b2Body.CreateShape(def:b2ShapeDef):b2Shape
b2Body.DestroyShape(shape:b2Shape):void
```

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 tolerance 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.

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 built in DebugDraw already features a scaling factor.

## User Data

The *b2Shape*, *b2Body*, and *b2Joint* classes allow you to attach any object as user data. 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** 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.

```
actor:GameActor = new GameCreateActor();
bodyDef:b2BodyDef = new b2BodyDef();
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 on one body, you should probably keep an actor on all bodies.

## Strawman

If you don't like this API design, that's ok! You have the source code! Seriously, if you have feedback about anything related to Box2D, please leave a comment in the forum.

# 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). Much of your interactions with Box2D will be with a *b2World* object.

## Creating and Destroying a World

Creating a world is fairly simple. You need to provide a bounding box and a gravity vector.

The axis-aligned bounding box should encapsulate the world. You can improve performance by making the bounding box a bit bigger than your world, say 2x just to be safe. If you have lots of bodies that fall into the abyss, your application should detect this and remove the bodies. This will improve performance and prevent floating point overflow.

```
1 var myWorld:b2World = new b2World(aabb, gravity, doSleep)
```

When myWorld goes out of use, it will automatically be deleted by the garbage collector.

**Caution**

Recall that the **world AABB** should always be bigger then the region where your bodies are located. If bodies leave the world AABB, then they will be frozen. This is not a bug.

## 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 an iteration count. For example:

```
1 var timeStep:Number = 1.0 / 60.;
2 var iterationCount:Number = 10;
3 myWorld.Step(timeStep, iterationCount);
```

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.

## Exploring the World

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

```
for (var b:b2Body = myWorld.GetBodyList(); b; b = b.GetNext())

{
    b.WakeUp();
}
```

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

```
for (var b:b2Body = myWorld.GetBodyList(); b; b = b.GetNext())

{
    var myActor:GameActor = b->GetUserData() as GameActor;
    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 body before destroying the body.

```
var node:b2Body = myWorld.GetBodyList();
while (node)

{
    var b:b2Body = node;
    node = node.GetNext();

    var myActor:GameActor = b->GetUserData() as GameActor;
    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.

```
var node:b2Body = myWorld.GetBodyList();
while (node)

{
    var b:b2Body = node;
    node = node.GetNext();

    var myActor:GameActor = b.GetUserData() as GameActor;
    if (myActor.IsDead())

    {
        var otherBodiesDestroyed:Boolean = 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 *b2World* returns an array of all the shapes that potentially intersect the AABB. This is not exact because what the function actually does is return all the shapes whose AABBs intersect the specified AABB. For example, the following code finds all the shapes that potentially intersect a specified AABB and wakes up all of the associated bodies.

```
var aabb:b2AABB = new b2AABB();
aabb.lowerBound.Set(-1.0, -1.0);
aabb.upperBound.Set(1.0, 1.0);

var k_bufferSize:Number = 10;
var buffer:Array = [];
var count:Number = myWorld.Query(aabb, buffer, k_bufferSize);

for (var i:Number = 0; i < count; ++i)
{
    buffer[i].GetBody().WakeUp();

}
```

**Note:** This calling syntax is rather obscure for typical AS3 use. This is due to Box2DAS3's C++ heritage, which does not handle variable sized arrays so easily.

# Bodies

## About

Bodies have position and velocity. You can apply forces, torques, and impulses to bodies. Bodies can be static or dynamic. Static bodies never move and don't collide with other static bodies.

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

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.

Lets go over some of the key members of the body definition.

## Mass Properties

There are a few ways to establish the mass properties for a body.

1. Set the mass properties explicitly in the body definition.
2. Set the mass properties explicitly on the body (after it has been created).
3. Set the mass properties based on the density of the attaced shapes.

In many game scenarios it makes sense to compute mass based on shape densities. This helps to ensure that bodies have reasonable and consistent mass values.

However, other game scenarios may require specific mass values. For example, you may have a mechanism, like a scale that needs precise mass values.

You can explicitly set the mass properties in the body definition as follows:

```
bodyDef.massData.mass = 2.0;              // the body's mass in kg

```

```
bodyDef.massData.center.SetZero();       // the center of mass in
local coordinates
bodyDef.massData.I = 3.0;                // the rotational
inertia in kg*m^2.
```

The other methods of setting the mass properties are covered elsewhere in this document.

## Position and Angle

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

A body has two main points of interest. The first point is the body's **origin**. Shapes 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(0.0, 2.0);     // the body's origin
position.
bodyDef.angle = 0.25 * b2Settings.b2_pi;       // the body's
angle in radians.
```

## Damping

Damping is used to reduce the world velocity of bodies. Damping differs from friction in that friction only occurs when two surfaces are in contact. Damping is also much cheaper to simulate than friction. Note, however, that damping is not a replacement for friction; 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 1.0. I generally do not use linear damping because it makes bodies look **floaty**.

```
bodyDef.linearDamping = 0.0;
bodyDef.angularDamping = 0.1;
```

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 an awake body 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.isSleeping = false;
```

## Bullets

Game simulation usually generates a sequence of images that are played at some frame rate. In this setting 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 simulated to the end of the original time step. This process is repeated as necessary.

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 thin wall. Without CCD, the bullet my tunnel through the wall.

Fast moving objects in Box2D are called bullets. 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.

```
1 bodyDef.isBullet = true;
```

The bullet flag only affects dynamic bodies.

CCD is expensive so you probably don't want all moving bodies to be bullets. So by default Box2D only uses CCD between moving bodies and static bodies. This is an effective approach to prevent bodies from escaping your game world. However, you may have some fast moving bodies that that require CCD all the time.

## 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.

```
1 var dynamicBody:b2Body = myWorld.CreateBody(bodyDef);
2
3 //... do stuff ...
4 myWorld.DestroyBody(dynamicBody);
5 dynamicBody = null;
```

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 simulate on their own and they never collide with other static 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. See [[Python_Manual#Implicit_Destruction|Implicit Destruction] for details.

Suppose you want to connect a dynamic body to ground with a joint. You'll need to connect the joint to a static body. If you don't have a static body, you can get a shared ground body from your world object. You can also attach static shapes to the ground body.

```
var ground:b2Body = myWorld.GetGroundBody();
//... build a joint using the ground body ...
```

## 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

You can adjust the mass of a body at run-time. This is usually done when you create or destroy shapes on a body. You may want to adjust the mass of the body based on the current shapes.

```
b2Body.SetMassFromShapes():void;
```

You may also want to set the mass properties directly. For example, you may change the shapes, but want to use your own mass formulas.

```
b2Body.SetMass(massData:b2MassData):void;
```

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

```
b2Body.GetMass():Number;
b2Body.GetInertia():Number;
b2Body.GetLocalCenter():b2Vec;
```

## State Information

Box2D keeps track of many aspects of a body's state. You can efficiently access this state data through the following functions:

```
b2Body.IsBullet():Boolean;
b2Body.SetBullet(flag:Boolean):void;


b2Body.IsStatic():Boolean
b2Body.IsDynamic():Boolean;

b2Body.IsFrozen():Boolean;


b2Body.IsSleeping():Boolean;
b2Body.AllowSleeping(flag:Boolean):void;
b2Body.WakeUp():void;
```

The bullet state is described in Bullets. The frozen state is described in World Boundary.

## Position and Velocity

You 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.

```
SetXForm(position:b2Vec2, angle:Number):Boolean;
```

```
GetXForm():b2XForm;
GetPosition():b2Vec2;
GetAngle():Number;
```

You can access the center of mass position in 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.

```
GetWorldCenter():b2Vec2;
```

You can access the linear and angular velocity. The linear velocity is for the center of mass.

```
function SetLinearVelocity(v:b2Vec2):void;

function GetLinearVelocity():b2Vec2;
function SetAngularVelocity(omega:Number):void;
function GetAngularVelocity():Number;
```

## 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.

```
ApplyForce(force:b2Vec2, point:b2Vec2):void;
ApplyTorque(float32 torque):void;
ApplyImpulse(impulse:b2Vec2, point:b2Vec2):void;
```

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.IsSleeping() == false)
{
    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, so use them with impunity.

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

## Lists

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

```
for (var s:b2Shape = body.GetShapeList(); s; s = s.GetNext())
{
    var data:MyShapeData = s.GetUserData() as MyShapeData;
    ... do something with data ...

}
```

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

# Shapes

## About

Shapes are the collision geometry attached to bodies. Shapes 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.

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

Shapes are always owned by a body. You can attach multiple shapes 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:

```
1 var friction : Number = Math.sqrt(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.

```
1 var restitution: Number = Math.Max(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:

```
1 playerShapeDef.filter.categoryBits   = 0x0002;
2 monsterShapeDef.filter.categoryBits  = 0x0004;
```

```
playerShapeDef.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 precendence 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

Some times 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.

```
var def:b2CircleDef;
def.radius = 1.5;
def.localPosition.Set(1.0, 0.0);
```

## 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 file *b2Settings*.

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.


convex_concave.gif

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

```
var triangleDef:b2PolygonDef = b2PolygonDef()
```

```
triangleDef.vertexCount = 3

triangleDef.vertices[0].Set(-1.0, 0.0)
triangleDef.vertices[1].Set(1.0, 0.0)

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

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.

```
var alignedBoxDef:b2PolygonDef = new b2PolygonDef();

var hx:Number = 1.0; // half-width
var hy:Number = 2.0; // half-height

alignedBoxDef.SetAsBox(hx, hy);

var orientedBoxDef:b2PolygonDef = new b2PolygonDef();
var center:b2Vec2 = new b2Vec2(-1.5, 0.0);

var angle:Number = 0.5 * b2Settings.b2_pi;
orientedBoxDef.SetAsOrientedBox(hx, hy, center, angle);
```

## Shape Factory

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

```
var circleDef:b2CircleDef = new b2CircleDef();
circleDef.radius = 3.0;
circleDef.density = 2.5;

var myShape:b2Shape = myBody.CreateShape(circleDef);
//[optionally store shape object 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 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.as* 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 joints 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 be 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 two 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 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, which we will cover next.

## 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.

 distanceJoint.gif

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

```
1 var jointDef:b2DistanceJointDef = new b2DistanceJointDef();
2 jointDef.Initialize(myBody1, myBody2, worldAnchorOnBody1,
  worldAnchorOnBody2);
3 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**.


revoluteJoint.gif

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.

```
var jointDef:b2RevoluteJointDef = new b2RevoluteJointDef();
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:

**Caution**

"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.

```
var jointDef:b2RevoluteJointDef = new b2RevoluteJointDef();
jointDef.Initialize(body1, body2, myBody1.GetWorldCenter());
jointDef.lowerAngle     = -0.5 * b2Settings.b2_pi; // -90 degrees

jointDef.upperAngle     = 0.25 * b2Settings.b2_pi; // 45 degrees
jointDef.enableLimit    = true;
jointDef.maxMotorTorque = 10.0;
jointDef.motorSpeed     = 0.0;
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.


prismaticJoint.gif

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:

```
var worldAxis:b2Vec2 = new b2Vec2(1.0, 0.0);

var jointDef:b2PrismaticJointDef = new b2PrismaticJointDef();
jointDef.Initialize(myBody1, myBody2, myBody1.GetWorldCenter(),
worldAxis);
jointDef.lowerTranslation= -5.0;
jointDef.upperTranslation= 2.5;
jointDef.enableLimit     = true;
jointDef.maxMotorForce   = 1.0;
jointDef.motorSpeed      = 0.0;
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
```


pulleyJoint.gif

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:

```
var anchor1:b2Vec2 = myBody1.GetWorldCenter();
var anchor2:b2Vec2 = myBody2.GetWorldCenter();
var groundAnchor1:b2Vec2 = new b2Vec2(p1.x, p1.y + 10.0);

var groundAnchor2:b2Vec2 = new b2Vec2(p2.x, p2.y + 12.0);
var ratio:Number = 1.0;

var jointDef:b2PulleyJointDef = new b2PulleyJointDef();
jointDef.Initialize(myBody1, myBody2, groundAnchor1,
groundAnchor2, anchor1, anchor2, ratio);
jointDef.maxLength1 = 18.0;
jointDef.maxLength2 = 20.0;
```

## 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**.

 gearJoint.gif

The gear joint requires that you have two bodies connected to a 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 will have units of length or one over length.

```
coordinate1 + ratio * coordinate2 == constant
```

Here is an example gear joint:

```
var jointDef:b2GearJointDef = new b2GearJointDef();
jointDef.body1 = myBody1;
jointDef.body2 = myBody2;
jointDef.joint1 = myRevoluteJoint;
jointDef.joint2 = myPrismaticJoint;
jointDef.ratio = 2.0 * b2Settings.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.as 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 yourself using *new*. 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:

```
var jointDef:b2RevoluteJointDef = new b2RevoluteJointDef();
jointDef.body1 = myBody1;
jointDef.body2 = myBody2;
jointDef.anchorPoint = myBody1.GetCenterPosition();

var 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 Implicit Destruction for details.

## Using Joints

Many simulations create 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.

```
GetBody1():b2Body;
GetBody2():b2Body;
GetAnchor1():b2Vec2;
GetAnchor2():b2Vec2;
GetUserData():*;
```

All joints have both a reaction force and a reaction torque. These are 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.

```
function GetReactionForce():b2Vec2;
function GetReactionTorque():Number;
```

## 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.

```
function GetJointAngle():Number;
function GetJointSpeed():Number;

function GetMotorTorque():Number;
```

You also update the motor parameters each step.

```
function SetMotorSpeed(speed:Number):void;

function SetMaxMotorTorque(torque:Number):void;
```

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(Math.cos(0.5 * time));

// ... Game Loop End ...
```

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

```
// ... Game Loop Begin ...
var angleError:Number = myJoint.GetJointAngle() - angleTarget;
var gain:Number = 0.1;
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:

```
function GetJointTranslation() :Number
function GetJointSpeed():Number

function GetMotorForce():Number
function SetMotorSpeed(speed:Number)
function SetMotorForce(force:Number)
```

## Using Pulley Joints

Pully joints provide the current lengths.

```
function GetLength1():Number
function GetLength2():Number
```

## 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.

Below is a small glossary of contact terminology. 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 uses an iterative contact solver and stores the results with the contact points. You can safely use the normal force to gauge the collision intensity. For example, you can use the force to trigger breakage or to play appropriate 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 feature indices that help to distinguish one contact point from another.

Contacts are created when two shapes' 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 when 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.

```
01 public class MyContactListener extends b2ContactListener
02
03 {
04         public override function Add(point:b2ContactPoint) : void
05         {
06                 // handle add point
07
08         }
09
10         public override function Persist(point:b2ContactPoint) :
   void
11         {
12                 // handle persist point
13
14         }
15
16         public override function Remove(point:b2ContactPoint) :
   void
17         {
18                 // handle remove point
19
20         }
21
22         public override function Result(point:b2ContactResult) :
   void
23         {
24                 // handle results
25
26         }
27 };
```

**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 estimate 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 is an excerpt ported from the CollisionProcessing test (test_CollisionProcessing.py) and shows how to handle orphaned bodies when processing the contact buffer.

```
// 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.
var k_maxNuke:int = 6

var nuke:Array = new Array();
var nukeCount:int = 0


// Traverse the contact results. Destroy bodies that
// are touching heavier bodies.
for each(var point:ContactPoint in contactPoints){
    var body1:b2Body = point.body1.GetBody();
    var body2:b2Body = point.body2.GetBody();
    var mass1:Number = body1.GetMass();
    var mass2:Number = body2.GetMass();


    if(mass1 > 0.0 and mass2 > 0.0){
        if(mass2 > mass1){

            nuke_body = body1
        } else {
            nuke_body = body2
        }
        if (nuke.indexOf(nuke_body) == -1){

            nuke.push(nuke_body);
            if(nuke.length == k_maxNuke)
                break
        }
    }

}
// Destroy the bodies, skipping duplicates.
for each(var b:b2Body in nuke){
    trace("Nuking:", b);
    self.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 the Filtering section of the Shapes chapter Filtering.

```
public class b2ContactFilter
{

```

```
04
05          /// Return true if contact calculations should be
performed between these two shapes.
06          /// @warning for performance reasons this is only called
when the AABBs begin to overlap.
07          public virtual function ShouldCollide(shape1:b2Shape,
shape2:b2Shape) : Boolean{
08
09                  var filter1:b2FilterData =
shape1.GetFilterData();
10                  var filter2:b2FilterData =
shape2.GetFilterData();
11
12                  if (filter1.groupIndex == filter2.groupIndex &&
filter1.groupIndex != 0)
13
14                  {
15                          return filter1.groupIndex > 0;
16                  }
17
18                  var collide:Boolean = (filter1.maskBits &
filter2.categoryBits) != 0 && (filter1.categoryBits &
filter2.maskBits) != 0;
19                  return collide;
20          }
21
22
23          static public var b2_defaultFilter:b2ContactFilter = new
b2ContactFilter();
24
25  };
```

# Loose Ends

## World Boundary

You can implement a *b2BoundaryListener* that allows *b2World* to inform you when a body has gone outside the world AABB. When you get the callback, you shouldn't try to delete the body, instead you should mark the parent actor for deletion or error handling. After the physics time step, you should handle the event.

```
01  public class b2BoundaryListener
02  {
03
04
05          /// This is called for each body that leaves the world
boundary.
06          /// @warning you can't modify the world inside this
callback.
07          public virtual function Violation(body:b2Body) : void{};
08
09
10  };
```

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

```
1  myWorld.SetListener(myBoundaryListener);
2  // Use SetBoundaryListener() in newer versions of Box2D
```

## 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.

```
01 public class b2DestructionListener
02
03 {
04
05         /// Called when any joint is about to be destroyed due
06         /// to the destruction of one of its attached bodies.
07         public virtual function SayGoodbyeJoint(joint:b2Joint) :
   void{};
08
09
10         /// Called when any shape is about to be destroyed due
11         /// to the destruction of its parent body.
12         public virtual function SayGoodbyeShape(shape:b2Shape) :
   void{};
13
14
15 };
```

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

```
1 myWorld.SetListener(myDestructionListener);
2 // Use SetDestructionListener() in newer versions of Box2D
```

# Settings

## About

There is a source file included in Box2D that are supplied specifically for user customization, called *b2Settings.as*.

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 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.as*.

# Rendering

## Debug Drawing

**Note:** Debug Drawing should probably not be used for your final game. Its purpose is as the name implies, debugging.

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 the preferred method of drawing these physics entities for debugging, 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.

```
01 //Assuming a world is set up, under variable m_world
02
03
04 //debugSprite is some sprite that we want to draw our debug
   shapes into.
05 var debugSprite = new Sprite();
06 addChild(debugSprite);
07
08 // set debug draw
09 var dbgDraw:b2DebugDraw = new b2DebugDraw();
10
11
12 dbgDraw.m_sprite = debugSprite;
13 dbgDraw.m_drawScale = 30.0;
14 dbgDraw.m_fillAlpha = 0.3;
15 dbgDraw.m_lineThickness = 1.0;
16 dbgDraw.m_drawFlags = b2DebugDraw.e_shapeBit |
   b2DebugDraw.e_jointBit;
17 m_world.SetDebugDraw(dbgDraw);
18
19 }}
```

## Drawing Sprites

Iterate through each actor, find each shape that belongs to the appropriate body, with the local position of the shape in mind, draw a sprite at the specified position and orientation. This article goes into further detail.

## What to do now

Todd's Box2D Tutorials!

Original Site Design by Andreas Viklund