

Introduction

Box2D is the world's most ubiquitous 2D physics engine. It's light, robust, efficient and highly portable. It has been battle-proven in many applications on many platforms, and it's open-source and free. Check out the Box2D website at <http://www.box2d.org>

A physics engine simulates the physics of objects to give them believable real-life movement. Although it can be used for other applications, the project was born primarily as a library for use in games, and games make up the majority of software using Box2D. The engine is written and maintained by one Erin Catto who it seems, not satisfied by simply making kick-ass physics as a day-job, set up Box2D to make kick-ass physics as a hobby as well.

Box2D is written in C++, but has been ported to many different languages by the user community. Here are just a few, you can also find more at <http://www.box2d.org/links.html>.

- (Flash) <http://www.box2dflash.org/>
- (Flash) <http://www.sideroller.com/wck/>
- (Java) <http://www.jbox2d.org/>
- (Python) <http://code.google.com/p/pybox2d/>
- (Javascript) <http://box2d-js.sourceforge.net/>
- (Javascript) <http://code.google.com/p/box2dweb/>
- (C#) <http://code.google.com/p/box2dx/>

A physics engine is not a game engine. Box2D is very good at simulating physics but it is not intended to draw nice graphics, use the network, build worlds or load a game level for you. If you came here hoping for a little more help with these needs, you might like to look into one of the links below.

- [cocos2d iPhone](#)
- [Citrus engine](#)
- [Corona SDK](#)
- [Game salad](#)
- [Pixelwave \(iOS\)](#)
- [Dragonfire SDK](#)
- [LibGDX](#)
- [Gamvas](#)
- todo: help me out here...

Box2D C++ tutorials

The [user manual](#) explains almost everything you need to know about using the library, at least for

C++. However I've been using Box2D for a while and reading the discussion forums I see the same questions coming up quite regularly. Since these are often things I have come across myself too, I decided I would write up some suggestions about how these issues could be solved. Then I figured if I was gonna do that, I might as well make a set of tutorials on using Box2D right from the beginning.

Looking around on the net, I found some people have already done a pretty good job of this:

[Allan Bishop](#) (Flash)
[Emanuele Feronato](#) (Flash)
[Flashy Todd](#) (Flash)
[Ray Wenderlich](#) (objc/c++/iPhone)
[Seth Ladd](#) (Javascript)
[Coding Owl](#) (Javascript)
... you? (let me know!)

As you can see these are predominantly in Flash, but I will be doing my tutorials in C++ since that's what I have used for most of my development. I will also be using a more recent version of Box2D than some of the tutorials above. Hopefully having C++ tutorials will be useful for someone, and my topics are not too similar to theirs! Eventually I am hoping to cover the following:

Basic usage

- Testbed setup (linux, windows, mac)
- Testbed structure
- Making a 'test' for the testbed
- Bodies
- Fixtures
- World settings
- Cancelling gravity
- Forces and impulses
- Moving at constant speed
- Keeping a body rotated at given angle
- Jumping
- Using debug draw
- Drawing your own objects
- User data
- Collision callbacks
- Collision filtering
- Sensors
- Raycasting
- World querying
- Removing bodies
- The 'touching the ground' question
- Joints
- Some gotchas

Advanced topics

- Vehicle suspension
- Sticky projectiles
- Projected trajectory

- Explosions
- Breakable bodies
- Top-down car physics
- Terrain
- One-way walls and platforms
- Conveyor belts
- Escalators?
- Elevators
- Arrow in flight
- Advanced character movement

Generally the focus will be on making a platform game, but I'll try to keep the content as broadly applicable as possible. Occasionally I will skip some features of the engine that are relevant to a particular topic, but are somewhat more advanced. Hopefully I can come back at a later date and add some 'advanced' sections here and there.

I will be using version 2.1.2 of the Box2D source code which seems to be the latest official release at this time. For most of the examples I will try to add the code for the tutorial as a 'test' in the 'testbed' which comes with the Box2D source code. This removes the need for setting up a project, window, etc and allows us to get straight into using the actual library itself.

Update: Version 2.1.2 is looking a little old now, but the main difference in the current (v2.3.0) version is the addition of chain shapes. Most of the material in these tutorials is still applicable, and I will try to add a little 'update' note like this in places where something should be mentioned.

Requirements

To follow these tutorials you will need to have an intermediate-level knowledge of C++. A beginner-level knowledge of OpenGL would be handy too, but not necessary. As for required software, you can download everything you'll need for free - no excuses now huh?

Dedication

I'm not sure if dedication is the right term to use here, but back in the day I learned OpenGL almost entirely from the NeHe site at nehe.gamedev.net, starting right from zero knowledge. The site was an incredibly useful resource and I was very grateful to have it available. Although I rarely visit the NeHe site these days, it was one of the major inspirations for me to set up this site. Thanks Jeff!!!

Feedback

If you spot any mistakes, have suggestions for improvement or any other feedback, write a comment on one of the topic pages or contact me at the gmail address: 'iforce2d'

Follow me on to be notified when topics are added:

Building the testbed (Mac OSX)

Let's look at the process of setting up the default 'testbed' which is part of the Box2d source code base. The testbed is a very useful tool which contains examples of various features which you can interact with using the mouse, adjust parameters, pause and step the simulation etc.

The simplest way to build the Box2D library is to use the cmake file which is included in the source code download. For this you will need the cmake tool, which you can download from [here](#), and XCode which you can download from Apple's developer site ([developer.apple.com](#)) after you have registered. When installing cmake be sure to choose the 'Install command line links' option so you can use the tool in a terminal. Since we'll be using cmake, all we really need for compiling is the gcc compiler itself, but apparently it is not distributed as a single tool so you'll need to get the whole XCode package (somebody let me know if I'm wrong about that). XCode is also available on the Snow Leopard DVD if you have one.

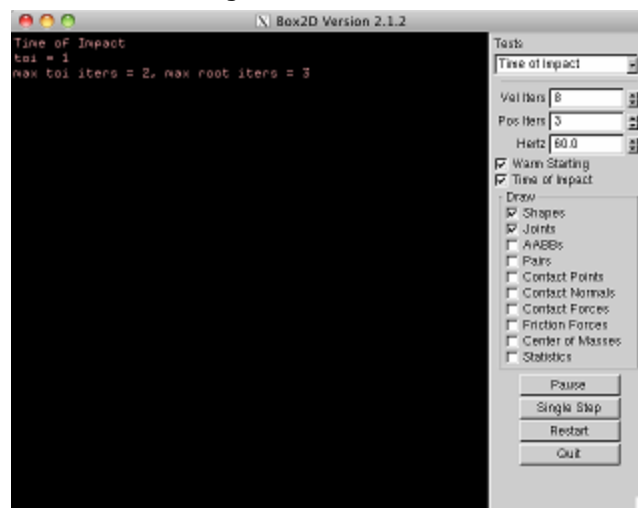
After installing XCode and cmake, download the Box2D source code archive from [here](#). The rest of the process is the same as for linux:

```
unzip Box2D_v2.1.2.zip
cd Box2D_v2.1.2/Box2D/Build
cmake ..
make
```

After this you should see that some new folders have been created, one of which is 'Testbed' and this in turn contains an executable file called 'Testbed', so the app can be started like this:

```
cd Testbed
./Testbed
```

You should see the testbed window showing like this:



Select from the drop-down list in the top right to try out the tests that are currently in the testbed. We will be adding our own items into this list later on.

Testbed features

Apart from the obvious features visible in the right hand control panel, the testbed also allows you

to:

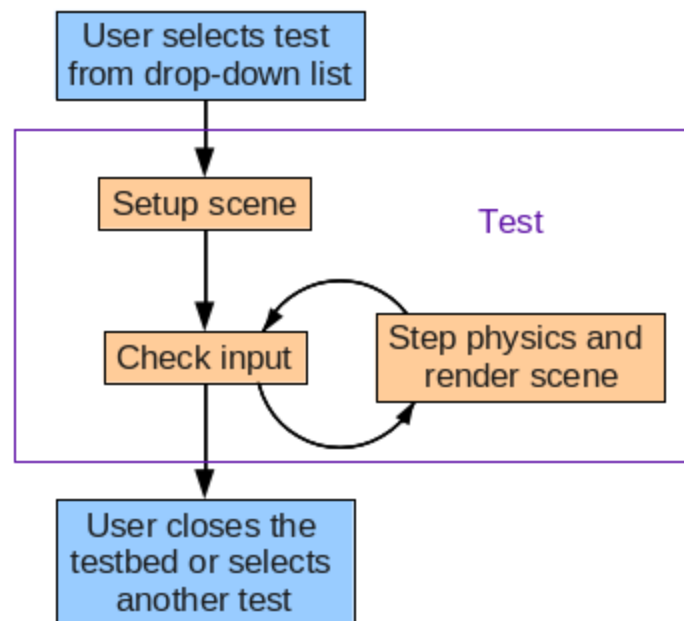
- Move the view around - arrow keys or drag with right mouse button
- Zoom the view - z,x keys
- Grab objects in the scene - left mouse button
- Launch a bomb into the scene from a random location - space bar
- Launch a bullet into the scene - drag left mouse button while holding shift, then let go

Depending on which test you view, you can sometimes use the keyboard to interact with the scene too. We will make use of the mouse and keyboard interactions in these tutorials.

Testbed structure

The testbed is set up in a way that allows a new test to be added efficiently. When a new test is defined, only the parts of the code which make it unique need to be written. Functions which are the same for all tests such as creating/destroying/resetting, and the control panel checkboxes and buttons, are handled by the main program code and never need to be changed.

Functions which are unique to each test - how the scene should be set up, what mouse/keyboard input should do, etc - can be specified as necessary. Before we add a test of our own, let's take a look at the life-cycle of a test.



The parts in orange are where we will be adding or changing code to make our own test. To be more specific, we will create a subclass of the Test class which handles all the common features of a test, and override some functions in our subclass. Let's take a look at some virtual functions in the Test class to see what we can use.

```
1 class Test ...{
2     virtual void Step(Settings* settings);
3     virtual void Keyboard(unsigned char key);
4     virtual void MouseDown(const b2Vec2& p);
5 }
```

```
virtual void MouseUp(const b2Vec2& p);
```

The first of these, Step() implements the stepping of the physics, and renders the scene. The other three, Keyboard(), MouseDown() and MouseUp() allow us to get some information about what the user is doing. There is no function to override to set up the scene, because this will be done in the constructor of the subclass.

Remember that the testbed has some [user input defined by default](#), so if this default input (mouse dragging of objects etc) is enough, then you may not need to override the input functions at all. Likewise if you are not doing anything special for rendering or physics control, you don't need to override Step() either. In fact, some of the example tests in the testbed are nothing more than a constructor to set up the scene. I told you it was simple didn't I...?

There are also some class member variables accessible from the subclass that will come in handy. These variables which keep track of the main Box2D world object which all the other thingies live in, and a body which represents the ground, a very common requirement in many simulations! We'll look at what a world and a body are a little later.

```
1 protected:  
2   b2World* m_world;  
3   b2Body* m_groundBody;
```

Making your own test

Let's add a test to the testbed, just a simple one that doesn't do much to start with but will serve as an example to look at all the places we need to edit.

First we'll need a subclass of the Test class to define the test itself. In the Testbed/Tests folder you should find a whole bunch of .h files with names similar to the tests you see in the testbed. In this folder, add another .h file of your own, with a clever name like uhh... FooTest. For now, we'll just make an empty test which shows some text at the top of the screen.

```
1 #ifndef FOOTEST_H  
2 #define FOOTEST_H  
3  
4 class FooTest : public Test  
5 {  
6   public:  
7     FooTest() { } //do nothing, no scene yet  
8  
9     void Step(Settings* settings)  
10    {  
11      //run the default physics and rendering  
12      Test::Step(settings);  
13  
14      //show some text in the main screen  
15      m_debugDraw.DrawString(5, m_textLine, "Now we have a foo test");  
16    }
```

```

1     m_textLine += 15;
3     }
1
4     static Test* Create()
1     {
5         return new FooTest;
1     }
6 };
1
7 #endif
1
8
1
9
2
0
2
1
2
2
2
3
2
4
2
5

```

The above class does not override any input functions, and doesn't set up a scene either. The Step() function is overridden, and uses the variable m_debugDraw of the parent class as a handy way to get some information on the screen. Another variable from the parent class, m_textLine is used here to increment the text drawing position, you would need to do this if you want to show more than one line of text without having them all on top of each other. The Create() function gives an instance of this class to the testbed framework when necessary.

Now to add this to the testbed, the file TestEntries.cpp in the same folder needs a couple of lines added:

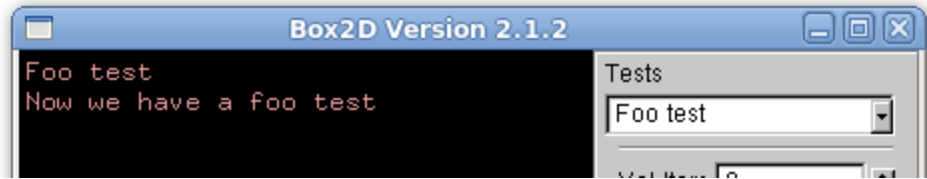
```

1 #include "FooTest.h"
2
3 {"Foo test", FooTest::Create},

```

These additions will go next to the existing lines of the same type which are already there. Take a look in the file and it will be pretty clear what I mean.

Now build the project again and run it. You should see your 'Foo test' option at the bottom of the drop-down list, and when selected the test should run and appear like this:



If you want your test to be selected by default when the testbed app starts up, just put the `FooTest::Create` line at the top of it's list.

Bodies

Bodies are the fundamental objects in the physics scene, but they are not what you actually see bouncing around and colliding with each other. Sound confusing? Hold on, I'll explain.

You can think of a body as the properties of an object that you cannot see (draw) or touch (collide with). These invisible properties are:

- mass - how heavy it is
- velocity - how fast and which direction it's moving
- rotational inertia - how much effort it takes to start or stop spinning
- angular velocity - how fast and which way it's rotating
- location - where it is
- angle - which way it is facing
-

Even if you know all of these characteristics of an object, you still don't know what it looks like or how it will react when it collides with another object. To define the size and shape of an object we need to use [fixtures](#), which are the subject of the next topic in this series, so for now we will just use a simple box and cover the details of fixtures later. Let's create a body and try setting some basic properties of it to see how they work.

There are three types of body available: static, dynamic and kinematic. The first two of these should be readily understandable, while the last one is probably not so intuitive. I'll cover the details a little further along. First we will make a dynamic body so we can see something moving around the scene, and we can try setting the velocity etc. for it.

Creating a body

Bodies are made by first setting up a definition, and then using this to create the body object itself. This can be handy if you want to make many bodies which are all the same, or very similar. In the constructor of the `FooTest` class, add the following code to set up a definition for a body:

```
1 b2BodyDef myBodyDef;  
2 myBodyDef.type = b2_dynamicBody; //this will be a dynamic body  
3 myBodyDef.position.Set(0, 20); //set the starting position  
4 myBodyDef.angle = 0; //set the starting angle
```

That's enough to define a basic body definition. Remember a body does not have any size, shape, so we don't define those here. You may be wondering why it has no mass yet - the usual way of

providing a mass for a body is by adding fixtures to it, which is coming up in the next step. Now, use this definition to create the actual body instance:

```
1 b2Body* dynamicBody = m_world->CreateBody(&myBodyDef);
```

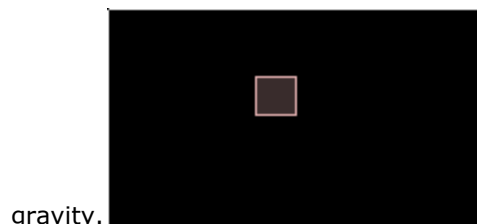
Here, we are using the `m_world` member variable of the parent class `Test` which is a `b2World` object. The world object is like the boss of everything in Box2D, it handles the creation and deletion of physics objects. We will take a look at the properties of the [world](#) in more detail a bit later. Ok, so now we have a body, but as we noted at the beginning of this page, a body is basically invisible, so if you build and run the program like this, there is nothing to see yet (if you turn on the 'Center of Masses' display you can see the position of the body falling though).

To give a body its size, shape, and other tangible characteristics, we add [fixtures](#) to it. Also, the default behaviour is that adding fixtures will affect the mass of the body too. A body can have many fixtures attached to it, each fixture added will affect the total mass of the body. For now, let's add one simple fixture to this body, a square, and look a bit further at fixtures in the next topic.

```
1 b2PolygonShape boxShape;  
2 boxShape.SetAsBox(1,1);  
3  
4 b2FixtureDef boxFixtureDef;  
5 boxFixtureDef.shape = &boxShape;  
6 boxFixtureDef.density = 1;  
7 dynamicBody->CreateFixture(&boxFixtureDef);
```

Although you can ignore most of the above section for now, notice the line referring to density. The area of the fixture is multiplied by the density of the fixture to calculate its mass, and this becomes the mass of the body.

Now when you run the program you should see a small box falling downwards - if you are quick enough you can catch it with the mouse cursor and throw it around, press restart (R key) if you lose it off the screen. Since this is a dynamic body it is able to move and rotate, and is affected by



Setting body properties

Now let's see what happens when we set some of the properties mentioned at the beginning of this topic. For example, change the starting position and angle:

```
1 dynamicBody->SetTransform( b2Vec2( 10, 20 ), 1 );
```

This will make the body start 10 units further to the right, 20 units higher, and rotated 1 radian

counter-clockwise. Box2D uses radians for angle measurements, so if you are like me and more used to thinking in angles, you could do something like this:

```
1 #define DEGTORAD 0.0174532925199432957f
2 #define RADTODEG 57.295779513082320876f
3
4 dynamicBody->SetTransform( b2Vec2( 10, 20 ), 45 * DEGTORAD ); //45 degrees
   counter-clockwise
```

We can also set the linear velocity and angular velocity of the body:

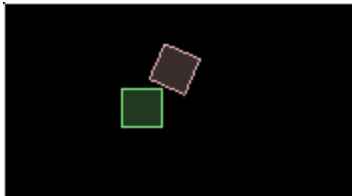
```
1 dynamicBody->SetLinearVelocity( b2Vec2( -5, 5 ) ); //moving up and left 5 units per second
2 dynamicBody->SetAngularVelocity( -90 * DEGTORAD ); //90 degrees per second clockwise
```

Static bodies

Now let's see what a static body does. Since we already have definitions for a body and a fixture, we can re-use them and just change the necessary features:

```
1 myBodyDef.type = b2_staticBody; //this will be a static body
2 myBodyDef.position.Set(0, 10); //slightly lower position
3 b2Body* staticBody = m_world->CreateBody(&myBodyDef); //add body to world
4 staticBody->CreateFixture(&boxFixtureDef); //add fixture to body
```

Notice here we didn't need to change the square fixture at all. Running the program now you should see another box in the scene, but this time it will not move. You will also find that while you can successfully use setTransform as above to change the location of this static body, setting the velocity properties for a static body will have no effect.



Kinematic bodies

Lastly, let's see what a 'kinematic' body is all about. As we have seen so far, dynamic bodies move and static bodies don't. When a static body and a dynamic body collide, the static body always 'wins' and holds its ground, and the dynamic body will retreat as necessary so that the two are not overlapping. A kinematic body is very similar to a static body in that when it collides with a dynamic body it always holds its ground and forces the dynamic body to retreat out of the way. The difference is that a kinematic body can move.

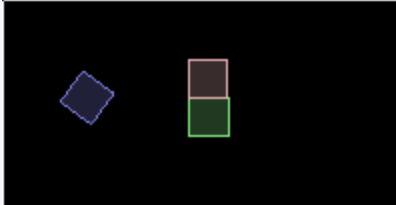
Try setting up a kinematic body like this:

```
1 myBodyDef.type = b2_kinematicBody; //this will be a kinematic body
2 myBodyDef.position.Set(-18, 11); // start from left side, slightly above the static body
```

```

3 b2Body* kinematicBody = m_world->CreateBody(&myBodyDef); //add body to world
4 kinematicBody->CreateFixture(&boxFixtureDef); //add fixture to body
5
6 kinematicBody->SetLinearVelocity( b2Vec2( 1, 0 ) ); //move right 1 unit per second
7 kinematicBody->SetAngularVelocity( 360 * DEGTORAD ); //1 turn per second counter-clockwise

```



The new body in the scene can move and rotate, but is not affected by gravity, and not affected when the dynamic body collides with it. Notice that when it touches the static body, there is no interaction between them.

For most games, dynamic bodies are used for the player and other actors in the scene, and static bodies are used for walls, floors and so on. Kinematic bodies came about to fill the need for a body which can move and rotate but doesn't get bumped around by the dynamic bodies. A perfect example is a moving platform in a platform game - it should always stay on its track no matter how it is jumped on or collided with.

Getting body properties

Often you will want to know where a body is or how fast it is moving, rotating etc. This is pretty easy so let's give it a try. For this we will need to access the body variables in the Step() function, so we'll need to make them a class member variable instead of just declaring them inside the constructor:

```

1 //in the class itself, not inside the constructor!
2 b2Body* dynamicBody;

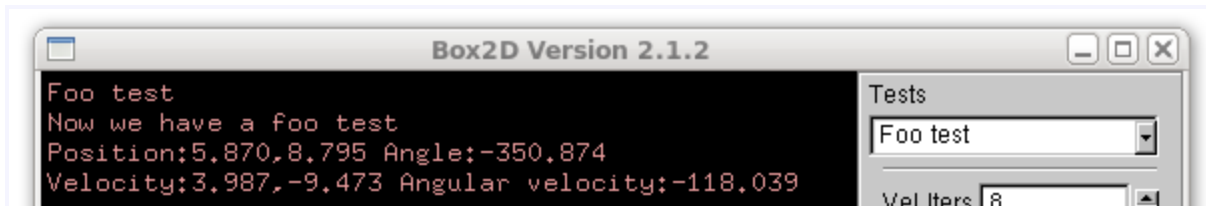
```

... and change the declarations where the bodies are created to use these class member variables. Now inside the Step() function, we can do this to print some information for a body:

```

1 b2Vec2 pos = dynamicBody->GetPosition();
2 float angle = dynamicBody->GetAngle();
3 b2Vec2 vel = dynamicBody->GetLinearVelocity();
4 float angularVel = dynamicBody->GetAngularVelocity();
5 m_debugDraw.DrawString(5, m_textLine,
6   "Position:%.3f,%.3f Angle:%.3f", pos.x, pos.y, angle * RADTODEG);
7 m_textLine += 15;
8 m_debugDraw.DrawString(5, m_textLine,
9   "Velocity:%.3f,%.3f Angular velocity:%.3f", vel.x, vel.y, angularVel * RADTODEG);
10 m_textLine += 15;
11

```



You can feed the `GetPosition()` and `GetAngle()` results back into `SetTransform` so that no change is made. For example, the following code will cause no change to a body's movement at all:

```
1 body->SetTransform( body->GetPosition(), body->GetAngle() );
```

Of course that's not very useful, but if you wanted to change only the position, or only the angle, you can do it in this way.

Iterating over the bodies in the world

If you want to look at all the bodies in the world, you can do it as below. The function `GetBodyList()` returns the first element in a linked list of bodies.

```
1 for ( b2Body* b = m_world->GetBodyList(); b; b = b->GetNext())
2 {
3     //do something with the body 'b'
4 }
```

Cleaning up

When you're done with a body, you can remove it from the scene by calling the world's `DestroyBody` function:

```
1 m_world->DestroyBody(dynamicBody);
```

When a body is destroyed like this, it takes care of deleting all the fixtures and joints attached to it. Remember not to use the deleted body pointer after this!

Fixtures

Fixtures are used to describe the size, shape, and material properties of an object in the physics scene. One body can have multiple fixtures attached to it, and the center of mass of the body will be affected by the arrangement of its fixtures. When two bodies collide, their fixtures are used to decide how they will react. The main properties of fixtures are:

- shape - a polygon or circle
- restitution - how bouncy the fixture is
- friction - how slippery it is

- density - how heavy it is in relation to its area

We'll go through each of these and experiment with them (there is also an `isSensor` property which will make the fixture a '[sensor](#)' which I will cover in a later topic) Let's start with a single dynamic body created as in the first part of the '[Bodies](#)' topic, so the constructor for the `FooTest` class will look like this:

```
1 FooTest() {
2     b2BodyDef myBodyDef;
3     myBodyDef.type = b2_dynamicBody; //this will be a dynamic body
4     myBodyDef.position.Set(-10, 20); //a little to the left
5
6     b2Body* dynamicBody1 = m_world->CreateBody(&myBodyDef);
7 }
```

Shapes

Every fixture has a shape which is used to check for collisions with other fixtures as it moves around the scene. A shape can be a circle or a polygon. Let's set up a circle shape...

```
1 b2CircleShape circleShape;
2 circleShape.m_p.Set(0, 0); //position, relative to body position
3 circleShape.m_radius = 1; //radius
```

... and a fixture which uses this shape:

```
1 b2FixtureDef myFixtureDef;
2 myFixtureDef.shape = &circleShape; //this is a pointer to the shape above
3 dynamicBody1->CreateFixture(&myFixtureDef); //add a fixture to the body
```

Running the program now you should see a circle in place of the square we had before:



When setting the position of the circle, the coordinates will be relative to the body's position. In this case we set the location of the circle as (0,0) but this will be attached to a body which is at (0,20), so the circle will be at (0,20) too.

Now let's try a polygon shape. With polygon shapes, you can set each individual vertex of the polygon to create a customized shape, or make use of a couple of convenience functions if you want boxes or lines. Here is a customized polygon with five vertices:

```

1 //set each vertex of polygon in an array
2 b2Vec2 vertices[5];
3 vertices[0].Set(-1, 2);
4 vertices[1].Set(-1, 0);
5 vertices[2].Set( 0, -3);
6 vertices[3].Set( 1, 0);
7 vertices[4].Set( 1, 1);
8
9 b2PolygonShape polygonShape;
10 polygonShape.Set(vertices, 5); //pass array to the shape
11
12 myFixtureDef.shape = &polygonShape; //change the shape of the fixture
13 myBodyDef.position.Set(0, 20); //in the middle
14 b2Body* dynamicBody2 = m_world->CreateBody(&myBodyDef);
15 dynamicBody2->CreateFixture(&myFixtureDef); //add a fixture to the body

```



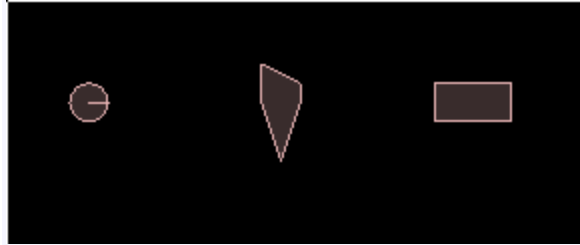
There are a few things to be careful of when creating polygon fixtures this way. Firstly there is a limit of 8 vertices per polygon by default. If you need more you can adjust the value `b2_maxPolygonVertices` in the file `b2Settings.h`. The vertices must also be specified in counter-clockwise order, and they must describe a convex polygon. A convex polygon is one which, if you were to walk around the edges of it, you always turn the same way at each corner (either always left, or always right).

If you want a rectangular fixture, the easiest way to get one of these is with the `SetAsBox` function which we used in the last topic:

```

1 polygonShape.SetAsBox(2, 1); //a 4x2 rectangle
2 myBodyDef.position.Set(10,20); //a bit to the right
3
4 b2Body* dynamicBody3 = m_world->CreateBody(&myBodyDef);
5 dynamicBody3->CreateFixture(&myFixtureDef); //add a fixture to the body

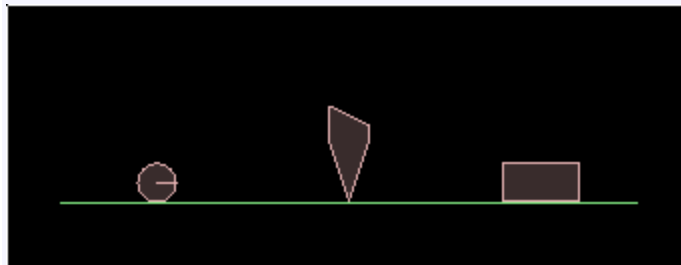
```



Notice that the parameters of `SetAsBox` are the 'half-width' and 'half-height' of the box, and it is centered at the location of the body it gets attached to. Notice also that because the fixture definition holds a reference pointer to the `polygonShape` we changed, and all we are changing is the shape, we don't need to do any of the other setup code this time.

Sometimes instead of a solid shape, you want to have a fixture represented by a simple line with zero thickness. This can be done with another convenience function, `SetAsEdge`. This function takes two points which will be the ends of the line. Let's make a static body with a long flat line fixture, near the bottom of the scene so the dynamic objects can fall onto it.

```
1 myBodyDef.type = b2_staticBody; //change body type
2 myBodyDef.position.Set(0,0); //middle, bottom
3
4 polygonShape.SetAsEdge( b2Vec2(-15,0), b2Vec2(15,0) ); //ends of the line
5 b2Body* staticBody = m_world->CreateBody(&myBodyDef);
6 staticBody->CreateFixture(&myFixtureDef); //add a fixture to the body
```



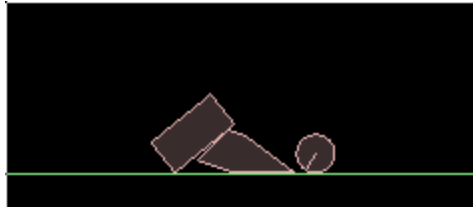
Note: The `SetAsEdge` function for polygons was removed in versions of Box2D after v2.1.2 because this kind of single-line shape has been given its own shape type, `b2EdgeShape`. Click [here](#) to see the equivalent usage of `b2EdgeShape`.

Density

By now you are probably wondering why these bodies don't spin around as expected when you grab them with the mouse cursor. This is because we haven't given any density setting for the fixtures. The density of a fixture multiplied by it's area becomes the mass of that fixture. Go back to where we first declared the fixture definition, and set a density for it:

```
1 b2FixtureDef myFixtureDef;
2 ...
3 myFixtureDef.density = 1; //new code
```

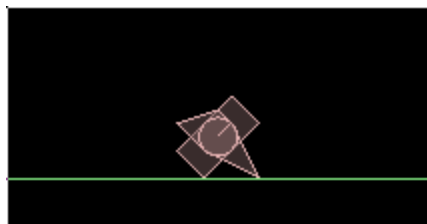

Since all the bodies were using the same fixture definition this will affect all of them.



In this example, all the fixtures having the same density means they weigh about as much as their visual size suggests they would, the circle being the lightest. Experiment by changing the density for each fixture and see how this setting gives a body its 'weight'. Remember that because all bodies are using the same fixture definition, you'll have to make sure the density you want is set in the fixture definition before calling `CreateFixture` each time.

Multiple fixtures

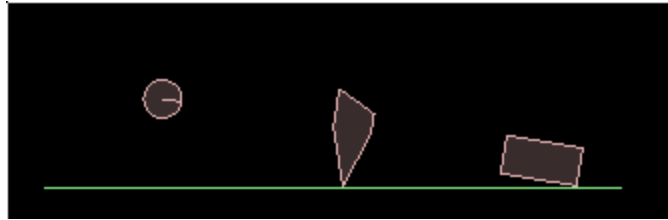
We can attach all three of these fixtures onto one body, and it's actually pretty easy - go back to each time you called `CreateFixture`, and instead of doing it for `dynamicBody1`, `dynamicBody2` and `dynamicBody3`, change those calls so that all three fixtures get added to `dynamicBody1`.



As you can see this causes all the fixture locations to be relative to `dynamicBody1` now. It's quite intuitive to see the fixtures globbed together on their body, but it's not strictly necessary for them to be like that. Fixtures can be added anywhere on a body, even leaving empty space between them. For example, to keep the two polygon fixtures in the same places they were before, we would need to go back and manually adjust their locations where we created them.

```
1 //for the custom polygon, add 10 to each x-coord
2 vertices[0].Set(-1 +10, 2);
3 vertices[1].Set(-1 +10, 0);
4 vertices[2].Set( 0 +10, -3);
5 vertices[3].Set( 1 +10, 0);
6 vertices[4].Set( 1 +10, 1);
7 ...
8 //for the box, use an extended version of the SetAsBox function which allows
9 //us to set a location and angle (location is offset from body position)
10 polygonShape.SetAsBox(2, 1, b2Vec2(20,0), 0 ); //moved 20 units right, same angle
```

You should get a see-saw kind of object as below (this is a good illustration of how the body itself is intangible):

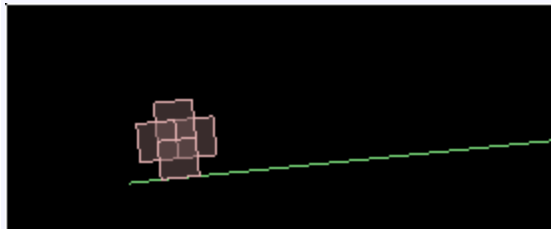


Friction

Now that we know how to set up a body with multiple fixtures, let's start with an empty scene again, and set up a body with four square fixtures on it to use in the next experiments.

```
1 FooTest() {
2     //set up a dynamic body
3     b2BodyDef myBodyDef;
4     myBodyDef.type = b2_dynamicBody;
5     myBodyDef.position.Set(0, 20); //middle
6     b2Body* dynamicBody = m_world->CreateBody(&myBodyDef);
7
8     //prepare a shape definition
9     b2PolygonShape polygonShape;
10    b2FixtureDef myFixtureDef;
11    myFixtureDef.shape = &polygonShape;
12    myFixtureDef.density = 1;
13
14    //add four square shaped fixtures around the body center
15    for ( int i = 0; i < 4; i++) {
16        b2Vec2 pos( sinf(i*90*DEGTORAD), cosf(i*90*DEGTORAD) ); //radial placement
17        polygonShape.SetAsBox(1, 1, pos, 0 ); //a 2x2 rectangle
18        dynamicBody->CreateFixture(&myFixtureDef); //add a fixture to the body
19    }
20
21    //make a static floor to drop things on
22    myBodyDef.type = b2_staticBody;
23    myBodyDef.position.Set(0, 0); //middle, bottom
24    b2Body* staticBody = m_world->CreateBody(&myBodyDef);
25    polygonShape.SetAsEdge( b2Vec2(-15,0), b2Vec2(15,3) ); //slightly sloped
26    staticBody->CreateFixture(&myFixtureDef); //add a fixture to the body
27 }
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

2
2
3
2
4
2
5
2
6
2
7



With this scene, you should see the body sliding a little as it rolls down the slope. As each fixture contacts the ground, its friction setting is used to calculate how much it should slide. Friction settings can be given a value between 0 and 1, zero being completely frictionless. When two fixtures collide with each other, the resulting friction tends toward the lower of their friction values.

Try giving the fixture definition zero friction:

```
1 myFixtureDef.density = 1;  
2 myFixtureDef.friction = 0; //new code
```

This time the body should rotate much less since its surface does not stick on the slope at all, and it will actually gain speed as it slides. Now try a friction value of 1 to compare the behaviour. Finally, to illustrate how **friction is a property of individual fixtures and not the body itself**, how about setting a different friction for each fixture - add a line inside the loop, just before the CreateFixture:

```
1 myFixtureDef.friction = i/4.0;
```

By playing around with the object you should be able to determine how much friction each fixture has.

* Note: a friction value of 1 does not always guarantee that there will be no sliding.

Restitution

Restitution measures how 'bouncy' a fixture is. Like friction, it is given a value between 0 and 1, where zero means that the fixture will not bounce at all, and one means that all the energy of the bounce will be conserved. When two fixtures collide with each other, the resulting restitution

tends toward the higher of their restitution values.

Restitution values can be experimented with in the same way as the friction values above. Try setting a different bounciness for each fixture.

```
1 myFixtureDef.friction = ...;
2 myFixtureDef.restitution = 0; //new code
```

* Note: a restitution value of 0 does not always guarantee that there will be no bouncing

* Note: in reality a tiny amount of energy can be lost in bouncing calculations

Changing fixture attributes at run-time

Sometimes you might want to alter the attributes of a fixture depending on events in the game. You can change the friction, density and restitution by setting these in the fixture using the setter functions. If you already have a reference to the fixture you want to change, this is pretty easy:

```
1 fixture->SetDensity( ... );
2 fixture->SetRestitution( ... );
3 fixture->SetFriction( ... );
```

If you only have a reference to the body, you'll need to iterate over the body's fixtures as shown [below](#) to find the one you want.

* Setting the density is a special case where one more step is necessary to have the changes take effect. After doing SetDensity() on the fixture, you need to call ResetMassData() on the body that the fixture belongs to.

Iterating over the fixtures in a body

If you have a body and you want to look at all the fixtures attached to it, you can do it as below. The function GetFixtureList() returns the first element in a linked list of fixtures.

```
1 for (b2Fixture* f = body->GetFixtureList(); f; f = f->GetNext())
2 {
3     //do something with the fixture 'f'
4 }
```

If you know there is only one fixture on the body, you could just use the first element of the list:

```
1 b2Fixture* f = body->GetFixtureList();
2 //do something with the fixture 'f'
```

Cleaning up

If you want to remove a fixture from a body, call the body's `DestroyFixture` function:

```
1 b2Fixture* myFixture = dynamicBody->CreateFixture(&myFixtureDef);  
2 ...  
3 dynamicBody->DestroyFixture(myFixture);
```

Remember not to use the deleted fixture pointer after this! When you destroy the body that a fixture is attached to, all the fixtures on it will be destroyed as well, so in this case too you must not use the fixtures after that.

Generally if you know the game logic, and how objects are likely to be destroyed, you can arrange your program to avoid using invalid fixture pointers. However if you are working on a complex project you might find it easier to use the 'destruction listener' feature provided by Box2D. This allows you to receive a notification when any fixture is destroyed so you know not to use it any more. Check out the section on 'Implicit destruction' in the 'Loose Ends' section of the [user manual](#).

Worlds

Worlds were briefly mentioned in one of the earlier topics as being the main entity in which all the Box2D bodies live. When you create or delete a body, you call a function of the world object to do this, so the world is managing all the allocations for the objects within it too. This means that the world is pretty important, so let's take a look at what we can do with one.

- define gravity
- tune the physics simulation
- find fixtures in a given region
- cast a ray and find intersected fixtures

The last two of these will be covered in a [later topic](#), so right now we will just look at the first of these, and the life cycle of a world. The testbed framework does this for us, and we've seen it as the `m_world` class member variable that we have been using, so instead of building a test scene we'll just take a quick look at the way it is done.

A world is set up like any normal class, taking a couple of fundamental settings in the constructor.

```
1 b2Vec2 gravity(0, -9.8); //normal earth gravity, 9.8 m/s/s straight down!  
2 bool doSleep = true;  
3  
4 b2World* myWorld = new b2World(gravity, doSleep);
```

The gravity setting affects every dynamic object in the world, and can be changed later by using `SetGravity`. For example, try adding this to one of the scenes we made earlier to make a zero-gravity world:

```
1 myWorld->SetGravity( b2Vec2(0,0) );
```

The sleep parameter says whether bodies should be allowed to 'sleep' if nothing is happening to them, for efficiency. If this is set to true, bodies will sleep when they come to rest, and are excluded from the simulation until something happens to 'wake' them again. This could be a collision from another body, a force applied to it etc.

Note: as of Box2D v2.2.1 the sleep parameter has been removed and defaults to true. To change this you can use `b2World::SetAllowSleeping(bool)`.

Once you have a world created as above, you can add bodies into it as we've been doing. To make anything interesting happen, we need to repeatedly call the Step function of the world to run the physics simulation. This is also being handled by the testbed framework, as part of the Step function of the Test class.

```
1 float32 timeStep = 1/20.0;    //the length of time passed to simulate (seconds)
2 int32 velocityIterations = 8; //how strongly to correct velocity
3 int32 positionIterations = 3; //how strongly to correct position
4
5 myWorld->Step( timeStep, velocityIterations, positionIterations);
```

In this example, each call to Step will advance the simulation by 1/20th of a second, so a body moving at 5m/s like in our first scene would move $5/20=0.25\text{m}$ in that time. The timeStep also affects how gravity gets to act on each body. You might find that things appear to fall at different accelerations depending on the time step. To make a realistic looking simulation, you will generally set the timeStep value to match the number of times per second you will be calling the world's Step() function in your game. For example in the testbed, the default framerate is 60 frames per second, so Step() is called 60 times a second with timeStep set to 1/60th of a second.

The velocity iterations and position iterations settings affect the way bodies will react when they collide. Typically in Box2D when a collision between two objects is detected, those objects are overlapping (stuck into other) and some calculation needs to be done to figure out how each body should move or rotate so that they are not overlapping any more. Making these values higher will give you a more correct simulation, at the cost of some performance.

(Update 2013/1/20): As requested by Nikolai in the comments, here is some more about what the iterations values actually do, to the best of my knowledge at least.

Firstly, these values are only relevant to collision resolution, so if nothing is colliding then they are not used at all. When two things collide, to resolve the collision (push both bodies so they don't overlap any more) they need to have their position and velocity changed. The need for changing the position should be obvious - this is to correct the overlap. The velocity also needs to be changed, for example to make sure that a ball bounces off a wall correctly, or to make something rotate if it is hit off-center.

The exact details of which body should move where, what their new velocities should be, whether their angular velocity should also be affected etc, is handled by an iterative solver. This means that the calculation does not give you a perfect result the first time, but each time you do it the result gets more accurate. Probably the simplest example of iterative solving is the Newton-Raphson method for finding the roots of a function. Here is a nice [animated gif](#) showing

this procedure (if you are not familiar with 'finding roots', it just means finding the point where the blue function line crosses the x-axis).

As you can see in the animated gif, the same calculation is done over and over, and each time the solution gets closer to the exact result. Typically you would stop doing this calculation when the result does not change much anymore, because that means you are so close to the solution that doing more iterations does not improve the answer. If your calculation time is limited, you could stop doing the calculation after a fixed number of iterations, even though the answer you get is not an exact solution.

Typically Box2D is used for fast-paced simulations where an exact solution is not necessary, and at 60 frames per second it's hard to see little imperfections anyway. So we usually want to set an upper limit on how many iterations to do, to keep the CPU time required reasonable.

One way to decide on a good balance of these values for your situation would be to start by setting them really low, even as low as 1 may sometimes be ok for sparse simulations. If you have fast collisions, or many things touching each other all at the same time (especially piles and stacks of bodies) you will need to raise these values to avoid having things penetrating each other and making things look sloppy. At some point, you will find that raising the values further doesn't really help anything.

One last point to note is that these values are only an upper limit. If Box2D decides that the solution it has is good enough, it will stop calculating without using all the iterations you have allowed it. So even if you set these to say, 50, that does not mean it will always take ten times longer than setting it to 5. It only means that in a demanding case (like a big stack of blocks) you have given it permission to take ten times as long. This means that your game could run nice and smooth, until you get a huge pile of things stacked up and then it mysteriously starts to get slow and choppy, so you should check that the values you set will perform ok for all cases that your game is likely to encounter.

Cleaning up

When you're done with a world object, delete it as normal:

```
1 delete myWorld;
```

When a world is destroyed like this, it takes care of deleting all the joints and bodies in it. Remember not to use the deleted body pointers after this!

Forces and impulses

To move things around, you'll need to apply forces or impulses to a body. Forces act gradually over time to change the velocity of a body while impulses can change a body's velocity immediately. As an example, imagine you have a broken down car and you want to move it. You could use a force by slowly driving another car up next to it until their bumpers touched and then push it, causing it

to accelerate over time. Or you could use an impulse by driving the other car into it at full speed. Both methods have their place depending on what you are trying to simulate.

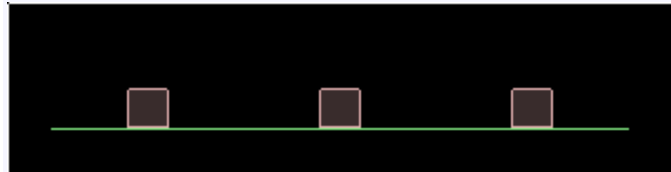
You can also 'warp' a body instantaneously by simply setting its location. This can be handy for games where you need a teleport feature, but bear in mind that this is not realistic physics! The whole point of a physics simulator like Box2D is to make things look real, and to this end I would recommend using forces and impulses to move bodies as much as you can. Sometimes it might seem tricky to think of a way to do this, but in the real world everything happens via forces or impulses, so unless you are creating a feature which is obviously not real-world (teleport etc), keep thinking. It may end up giving you less problems down the line.

Angular movement can also be controlled by forces and impulses, with the same gradual/immediate characteristics as their linear versions. Angular force is called torque. Think of this as a twisting strength, like when you twist the cap off a bottle - the bottle doesn't go anywhere, but you are still applying a force (torque) to it.

In this topic, we'll make three bodies and try using each one of the above-mentioned methods to move and twist them. Let's set up a scene similar to the one in the [fixtures](#) topic, but with all the shapes the same.

```
1 //class member variable to keep track of three bodies
2 b2Body* bodies[3];
3
4 FooTest() {
5     //body definition
6     b2BodyDef myBodyDef;
7     myBodyDef.type = b2_dynamicBody;
8
9     //shape definition
10    b2PolygonShape polygonShape;
11    polygonShape.SetAsBox(1, 1); //a 2x2 rectangle
12
13    //fixture definition
14    b2FixtureDef myFixtureDef;
15    myFixtureDef.shape = &polygonShape;
16    myFixtureDef.density = 1;
17
18    //create identical bodies in different positions
19    for (int i = 0; i < 3; i++) {
20        myBodyDef.position.Set(-10+i*10, 20);
21        bodies[i] = m_world->CreateBody(&myBodyDef);
22        bodies[i]->CreateFixture(&myFixtureDef);
23    }
24
25    //a static floor to drop things on
26    myBodyDef.type = b2_staticBody;
27    myBodyDef.position.Set(0, 0);
28    polygonShape.SetAsEdge( b2Vec2(-15,0), b2Vec2(15,0) );
29    m_world->CreateBody(&myBodyDef)->CreateFixture(&myFixtureDef);
30 }
```


2
1
2
2
2
2
3
2
4
2
5
2
6
2
7
2
8
2
9
3
0



Linear movement

We need some way of affecting these bodies without using the mouse. Now would be a good time to check out the keyboard input feature of the testbed. Override the Keyboard function of the Test class to use a different method for each body:

```
1 void Keyboard(unsigned char key)
2 {
3     switch (key)
4     {
5         case 'q':
6             //apply gradual force upwards
7             bodies[0]->ApplyForce( b2Vec2(0,50), bodies[0]->GetWorldCenter() );
8             break;
9         case 'w':
10            //apply immediate force upwards
11            bodies[1]->ApplyLinearImpulse( b2Vec2(0,50), bodies[1]->GetWorldCenter() );
12            break;
13        case 'e':
14            //teleport or 'warp' to new location
15            bodies[2]->SetTransform( b2Vec2(10,20), 0 );
```

```

1 break;
3 default:
1 //run default behaviour
4 Test::Keyboard(key);
1 }
5 }
1
6
1
7
1
8
1
9
2
0
2
1

```

The SetTransform function we covered in the [bodies](#) topic. The ApplyForce and ApplyLinearImpulse functions take two parameters, the first is what direction the force should be, in this case straight up. The second parameter is what point on the body should be pushed on - we'll get to this soon.

Run the test and try pressing the q/w/e keys. The impulsed body should react as if it was suddenly hit by something. The teleported body moves instantly to the new location, but notice that it retains its linear and angular velocity. What's happening with the body we applied the force to? Well, this Keyboard function is only called when we press the key down, not every timestep. In the example of car-pushing analogy, this would be like pushing the car for a fraction of a second, and then stopping. Since a force works over time, what we really need is some way to turn the force on and off instead of just blipping it on for a brief moment. We could use a class member variable to keep track of whether the force is on, and use the q key to toggle it. Make the following changes to the class:

```

1 //class member variable
2 bool forceOn;
3
4 //inside constructor
5 forceOn = false;
6
7 //modified case for q key in Keyboard() function
8 case 'q':
9     forceOn = !forceOn; //toggle bool value
1 break;
0
1 //inside Step() function
1 if (forceOn)
1     bodies[0]->ApplyForce( b2Vec2(0,50), bodies[0]->GetWorldCenter() );
2
1

```

3
1
4

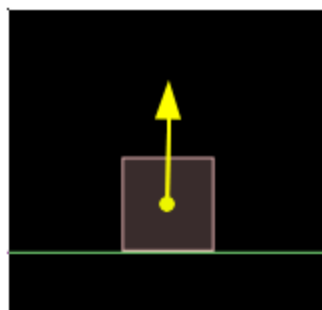
Now the q key should turn the force on and off, and you can see the gradual effect of a force.

Hmm.... we applied the same magnitude (50) for both impulse and force, so why does the force seem to be weaker than the impulse? Well, remember that gravity is a force too. Try turning gravity off and the answer might become clear. The reason is that the force acts a little bit each timestep to move the body up, and then gravity acts to push it back down again, in a continual up down up down struggle. The impulse on the other hand, does all its work before gravity gets a chance to interfere. With [gravity off](#), try turning the force on for about one second, then off. You will notice that after one second, the forced body has the same velocity as the impulsed body.

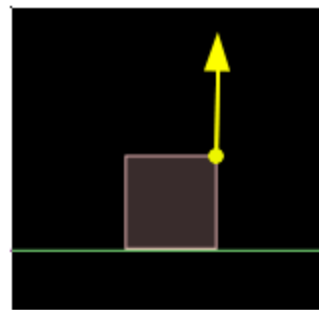
Now what about the last parameter of the apply force/impulse functions that we have ignored? So far we set this using the `GetWorldCenter()` of the body which will apply the force at the center of mass. As we can see, a force applied to the center of mass does not affect the rotation of the body. Imagine a CD on a friction-less flat surface, like an air-hockey table. If you put your finger in the hole in the middle of the CD and flick it across the table, it will not spin around as it moves. But if you do this with your finger anywhere else on the CD, it will spin around as it moves, and the further away from the middle your finger was, the more it will spin.

Let's offset the point at which we apply the force/impulse. Change the `ApplyForce` and `ApplyLinearImpulse` calls so that instead of using `GetWorldCenter()` to apply force at the center of mass, we'll apply it at the top right corner of the box:

```
1 //before
2 GetWorldCenter()
3
4 //after
5 GetWorldPoint( b2Vec2(1,1) )
```



`GetWorldCenter()`



`GetWorldPoint(b2Vec2(1,1))`

This time, you should see these boxes rotate when acted upon. The function `GetWorldPoint` is used to convert a location relative to the body (body coordinates) into world coordinates, so that we keep applying the force at the corner of the box even after it has rotated around. Note that while the picture above shows a scenario which might be interpreted as 'picking the box up by its

corner', the force being applied has nothing to do with the visible rectangle fixture - **forces and impulses are applied to bodies, not their fixtures**. The force could just as easily be applied at any old point, even in empty space where the body has no fixtures.

Angular movement

Angular movement is controllable by using angular forces (torque) and angular impulses. These behave similar to their linear counterparts in that force is gradual and impulse is immediate. Let's add a couple of cases to the Keyboard() function to try them out. (Although there is a third case where the rotation can be set instantaneously, we have already done that with SetTransform() above, so we'll skip that here.)

```
1 case 'a':  
2     //apply gradual torque counter clockwise  
3     bodies[0]->ApplyTorque( 20 );  
4     break;  
5 case 's':  
6     //apply immediate spin counter clockwise  
7     bodies[1]->ApplyAngularImpulse( 20 );  
8     break;
```

Now use the a/s keys to see what happens. Once again you will see that to make the torque effective, you will need to add a class member variable to toggle the torque on/off so it can be applied constantly every time step:

```
1 //class member variable  
2 bool torqueOn;  
3  
4 //inside constructor  
5 torqueOn = false;  
6  
7 //modified case for a key in Keyboard() function  
8 case 'a':  
9     torqueOn = !torqueOn; //toggle bool value  
1    break;  
0  
1 //inside Step() function  
1    if (torqueOn)  
1        bodies[0]->ApplyTorque( 20 );  
2  
1  
3  
1  
4
```

With gravity on, you'll see that even though we are only trying to 'twist' these boxes they still move around a bit, but that's only because they collide with the ground and act like square wheels. For a better illustration of what torque and angular impulse are doing, [turn gravity off](#). In

a zero-gravity scene, now we can see why only one parameter is necessary for angular force/impulse functions - since no linear movement is caused, the only thing we need to specify is which way the rotation should be. Unlike linear forces above there is never any offset possible because the rotation is always about the body's center of mass.

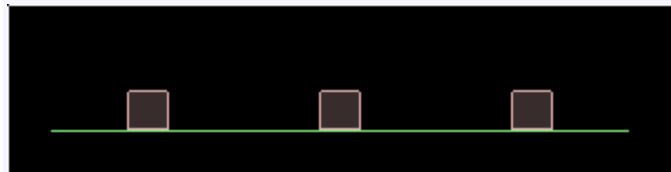
As with the linear forces, given the same magnitude parameter, `ApplyTorque` will take one second to gain as much rotational velocity as `ApplyAngularImpulse` does immediately.

Specifying a different gravity for each body

A question that comes up quite often is how to make certain bodies ignore gravity, while other bodies still obey gravity. This is really easy, so now that we know about forces let's try it. Start with the same scene as in the last topic, with three identical bodies.

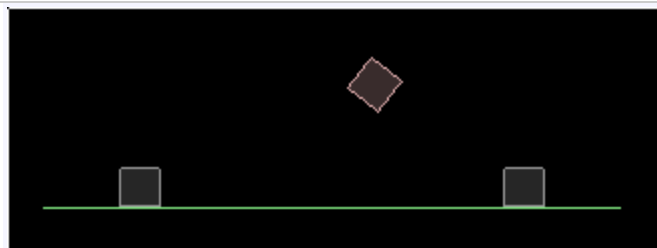
```
1 //class member variable to keep track of three bodies
2 b2Body* bodies[3];
3
4 FooTest() {
5     //body definition
6     b2BodyDef myBodyDef;
7     myBodyDef.type = b2_dynamicBody;
8
9     //shape definition
10    b2PolygonShape polygonShape;
11    polygonShape.SetAsBox(1, 1); //a 2x2 rectangle
12
13    //fixture definition
14    b2FixtureDef myFixtureDef;
15    myFixtureDef.shape = &polygonShape;
16    myFixtureDef.density = 1;
17
18    //create identical bodies in different positions
19    for (int i = 0; i < 3; i++) {
20        myBodyDef.position.Set(-10+i*10, 20);
21        bodies[i] = m_world->CreateBody(&myBodyDef);
22        bodies[i]->CreateFixture(&myFixtureDef);
23    }
24
25    //a static floor to drop things on
26    myBodyDef.type = b2_staticBody;
27    myBodyDef.position.Set(0, 0);
28    polygonShape.SetAsEdge(b2Vec2(-15,0), b2Vec2(15,0));
29    m_world->CreateBody(&myBodyDef)->CreateFixture(&myFixtureDef);
30 }
```

2
3
2
4
2
5
2
6
2
7
2
8
2
9
3
0



Since gravity is really just the same as applying a linear force downwards every time step, all we need to do to counteract it is apply the same force upwards. The required force is relative to the mass of the body:

```
1 //in the Step() function
2 //cancel gravity for body 1 only
3 bodies[1]->ApplyForce( bodies[1]->GetMass() * -m_world->GetGravity(),
  bodies[1]->GetWorldCenter() );
```



The same technique can be used to apply gravity in any direction you like, if you need to have something walking on walls or the ceiling for example.

Important: This gravity-cancelling force should be applied before the first time step. If you have the ApplyForce call after the world Step call in your main loop, the body will get one time step at normal gravity and it will have a chance to move down a tiny bit before the gravity is cancelled.

Note: the above is a typical solution for the v2.1.2 release of Box2D used for these tutorials. As of v2.2.1 each body has a 'gravity scale' to strengthen or weaken the effect of the world's gravity on it. This removes the need to manually apply a force every frame. You can set this in the body definition when you create it, or use:

```
1 //Box2D v2.2.1 onwards
2 body->SetGravityScale(0); //cancel gravity (use -1 to reverse gravity, etc)
```

Moving a body at a constant speed

A common requirement in games is to make a body move at a constant speed. This could be a player character in a platform game, a spaceship or car, etc. Depending on the game, sometimes a body should gain speed gradually, in other situations you might want it to start and stop instantaneously. It is very tempting to use the `SetLinearVelocity` function to explicitly set the velocity for a body to accomplish this, and indeed it does get the job done, but this approach has its drawbacks. While it often looks fine on the screen, setting the velocity directly means the body is not correctly participating in the physics simulation. Let's see how we can use the more realistic forces and impulses to move a body at a desired speed.

We'll look at two situations, one where the body should immediately start moving at the desired speed, and one where it should accelerate gradually until it reaches a specified top speed. To start with we'll need a scene with one dynamic body, and we'll make some static body walls to fence it in. This fence will come in handy in some of the upcoming topics too. To keep track of what the user wants to do, we'll have a class variable to store the last received input.

```
1 //enumeration of possible input states
2 enum _moveState {
3     MS_STOP,
4     MS_LEFT,
5     MS_RIGHT,
6 };
7
8 //class member variables
9 b2Body* body;
10 _moveState moveState;
11
12 FooTest() {
13     //body definition
14     b2BodyDef myBodyDef;
15     myBodyDef.type = b2_dynamicBody;
16
17     //shape definition
18     b2PolygonShape polygonShape;
19     polygonShape.SetAsBox(1, 1); //a 2x2 rectangle
20
21     //fixture definition
22     b2FixtureDef myFixtureDef;
23     myFixtureDef.shape = &polygonShape;
24     myFixtureDef.density = 1;
25
26     //create dynamic body
```

```

8  myBodyDef.position.Set(0, 10);
1  body = m_world->CreateBody(&myBodyDef);
9  body->CreateFixture(&myFixtureDef);
2
0  //a static body
2  myBodyDef.type = b2_staticBody;
1  myBodyDef.position.Set(0, 0);
2  b2Body* staticBody = m_world->CreateBody(&myBodyDef);
2
2  //add four walls to the static body
3  polygonShape.SetAsBox( 20, 1, b2Vec2(0, 0), 0);//ground
2  staticBody->CreateFixture(&myFixtureDef);
4  polygonShape.SetAsBox( 20, 1, b2Vec2(0, 40), 0);//ceiling
2  staticBody->CreateFixture(&myFixtureDef);
5  polygonShape.SetAsBox( 1, 20, b2Vec2(-20, 20), 0);//left wall
2  staticBody->CreateFixture(&myFixtureDef);
6  polygonShape.SetAsBox( 1, 20, b2Vec2(20, 20), 0);//right wall
2  staticBody->CreateFixture(&myFixtureDef);
7
2  moveState = MS_STOP;
8  }

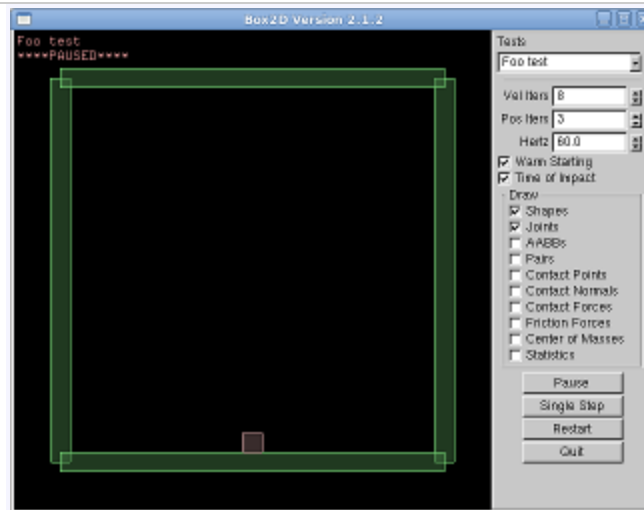
```

```

2
9
3
0
3
1
3
2
3
3
3
4
3
5
3
6
3
7
3
8
3
9
4
0
4
1
4
2
4

```


3
4
4
4
5
4
6
4
7



We'll need a Keyboard() function for input:

```

1 void Keyboard(unsigned char key)
2 {
3     switch (key)
4     {
5         case 'q': //move left
6             moveState = MS_LEFT;
7             break;
8         case 'w': //stop
9             moveState = MS_STOP;
10            break;
11        case 'e': //move right
12            moveState = MS_RIGHT;
13            break;
14        default:
15            //run default behaviour
16            Test::Keyboard(key);
17    }
18 }
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

6
1
7
1
8

From now, all further changes will be made in the Step() function to implement the movement behaviour depending on this input.

Setting velocity directly

Before we get started on the force/impulse methods, let's see how SetLinearVelocity works to directly specifying the velocity of the body. For many applications this may be good enough. Inside the Step() function, we will take whatever action is required each time step:

```
1 //inside Step()
2 b2Vec2 vel = body->GetLinearVelocity();
3 switch ( moveState )
4 {
5     case MS_LEFT: vel.x = -5; break;
6     case MS_STOP: vel.x = 0; break;
7     case MS_RIGHT: vel.x = 5; break;
8 }
9 body->SetLinearVelocity( vel );
```

Here we are getting the current velocity and leaving the vertical component unchanged, and feeding it back because we only want to affect the horizontal velocity of this body.

Trying this code in the testbed you'll see that this setup gives us the instantaneous speed scenario. To implement a gradual acceleration up to a maximum speed, you could do something like this instead.

```
1 switch ( moveState )
2 {
3     case MS_LEFT: vel.x = b2Max( vel.x - 0.1f, -5.0f ); break;
4     case MS_STOP: vel.x *= 0.98; break;
5     case MS_RIGHT: vel.x = b2Min( vel.x + 0.1f, 5.0f ); break;
6 }
```

This will increase the velocity linearly by 0.1 per time step to a maximum of 5 in the direction of travel - with the default testbed framerate of 60fps the body will take 50 frames or just under a second to reach top speed. When coming to a stop the speed is reduced to 98% of the previous frame's speed, which comes to about 0.98^{60} = a factor of about 0.3 per second. An advantage of this method is that these acceleration characteristics can be easily tuned.

Using forces

Forces are more suited to the gradual acceleration to top speed scenario, so let's try that first:

```
1  b2Vec2 vel = body->GetLinearVelocity();
2  float force = 0;
3  switch ( moveState )
4  {
5      case MS_LEFT: if ( vel.x > -5 ) force = -50; break;
6      case MS_STOP: force = vel.x * -10; break;
7      case MS_RIGHT: if ( vel.x < 5 ) force = 50; break;
8  }
9  body->ApplyForce( b2Vec2(force,0), body->GetWorldCenter() );
```

This is similar to the above in that the acceleration is linear and the braking is non-linear. For this example we have a pretty basic logic which simply applies the maximum force in every time step where the body is moving too slow. You will likely want to adjust this for the application you are making eg. a car might accelerate quickly at low speeds, but as it nears the maximum speed it's rate of acceleration decreases. For this you could just look at the difference between the current speed and the maximum speed and scale back the force as appropriate.

Remembering from the previous topic that forces act gradually, it might seem unlikely that we could use them to implement an instantaneous speed change. However, if we make the time span very short and the force very large, we can get the same effect as an impulse. First we need to do a little math...

The relationship between force and acceleration is $f = ma$ where m is the mass of the body we're moving, a is acceleration which is measured in "units per second per second", and f is the force we want to calculate. The acceleration could also be called "velocity per second", since velocity and "units per second" are the same thing. So we could write this as $f = mv/t$ where t is the length of time the force will be applied.

We can get m by using the body's `GetMass()` function. v will be the change in velocity we desire which is the difference between the maximum speed and the current speed. To get an instantaneous speed change effect, we would be applying the force for one time step or 1/60th of a second if using the default testbed framerate. Now we know everything except f , so we do something like this:

```
1  b2Vec2 vel = body->GetLinearVelocity();
2  float desiredVel = 0;
3  switch ( moveState )
4  {
5      case MS_LEFT: desiredVel = -5; break;
6      case MS_STOP: desiredVel = 0; break;
7      case MS_RIGHT: desiredVel = 5; break;
8  }
9  float velChange = desiredVel - vel.x;
10 float force = body->GetMass() * velChange / (1/60.0); //f = mv/t
```

```

1 body->ApplyForce( b2Vec2(force,0), body->GetWorldCenter() );
1

```

This should give you the same behaviour as the SetLinearVelocity did, while still remaining a realistic physics scenario.

Using impulses

Astute readers will notice that the code immediately above is basically simulating an impulse. However since impulses already take into account the length of the simulation timestep, we can just take the time part out and get the same effect with ApplyLinearImpulse:

```

1 b2Vec2 vel = body->GetLinearVelocity();
2 float desiredVel = 0;
3 switch ( moveState )
4 {
5     case MS_LEFT: desiredVel = -5; break;
6     case MS_STOP: desiredVel = 0; break;
7     case MS_RIGHT: desiredVel = 5; break;
8 }
9 float velChange = desiredVel - vel.x;
1 float impulse = body->GetMass() * velChange; //disregard time factor
0 body->ApplyLinearImpulse( b2Vec2(impulse,0), body->GetWorldCenter() );
1
1

```

For a gradual acceleration, just adjust the desired change in velocity as appropriate:

```

1 case MS_LEFT: desiredVel = b2Max( vel.x - 0.1f, -5.0f ); break;
2 case MS_STOP: desiredVel = vel.x * 0.98f; break;
3 case MS_RIGHT: desiredVel = b2Min( vel.x + 0.1f, 5.0f ); break;

```

Rotating a body to a given angle

This topic is similar to the previous topic but deals with rotating a body instead of linear movement. Rotating a body can be also done by setting the angle directly or by using torque/impulse methods, with the same point to note that setting the angle directly means the body is not participating correctly in the physics simulation.

To experiment with these all we need is one dynamic body, and it would be nice if there was no gravity so it stays on the screen for us. We will need to give the body a fixture which has a distinct direction, so we can check if it's facing the right way. Let's set it up as a polygon with one 'pointy' vertex:

```

1 //class member variable

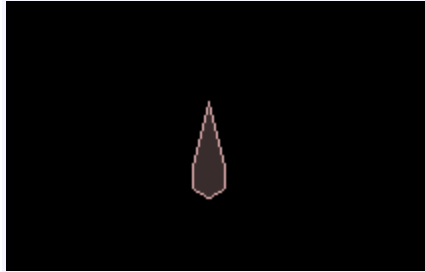
```

```

2  b2Body* body;
3
4  FooTest() {
5      //body definition
6      b2BodyDef myBodyDef;
7      myBodyDef.type = b2_dynamicBody;
8
9      //hexagonal shape definition
1     b2PolygonShape polygonShape;
0     b2Vec2 vertices[6];
1     for (int i = 0; i < 6; i++) {
1         float angle = -i/6.0 * 360 * DEGTORAD;
1         vertices[i].Set(sin(angle), cos(angle));
2     }
1     vertices[0].Set( 0, 4 ); //change one vertex to be pointy
3     polygonShape.Set(vertices, 6);
1
4     //fixture definition
1     b2FixtureDef myFixtureDef;
5     myFixtureDef.shape = &polygonShape;
1     myFixtureDef.density = 1;
6
1     //create dynamic body
7     myBodyDef.position.Set(0, 10);
1     body = m_world->CreateBody(&myBodyDef);
8     body->CreateFixture(&myFixtureDef);
1
9     //zero gravity
2     m_world->SetGravity( b2Vec2(0,0) );
0 }
2
1
2
2
2
3
2
4
2
5
2
6
2
7
2
8
2
9
3
0

```

3
1

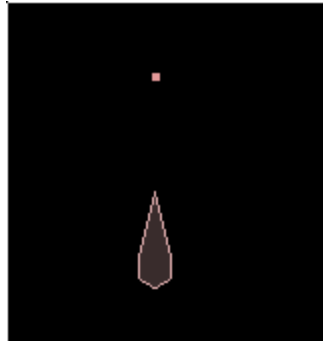


To set a point for this body to rotate towards, we'll use the mouse input function:

```
1 //class member variable
2 b2Vec2 clickedPoint;
3
4 //in class constructor
5 clickedPoint = b2Vec2(0,20); //initial starting point
6
7 //override parent class
8 void MouseDown(const b2Vec2& p)
9 {
10     //store last mouse-down position
11     clickedPoint = p;
12
13     //do normal behaviour
14     Test::MouseDown( p );
15 }
16
17 //inside Step()
18 glPointSize(4);
19 glBegin(GL_POINTS);
20 glVertex2f( clickedPoint.x, clickedPoint.y );
21 glEnd();
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
```

Don't worry if you don't recognize the code in the Step() function, this is just to draw the location

of clickedPoint on the screen.



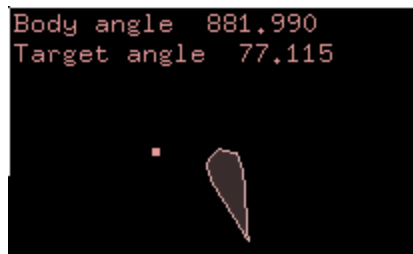
This point will move to wherever the mouse was last clicked. It still does the default behaviour of grabbing objects in the scene too, so it's a little strange but good enough for what we need.

Setting angle directly

This is as simple as using SetTransform to set the angle, but first we need to know what the angle should be, given the location of the body and the location of the target point. Add this to the Step() function to do it every time step:

```
1 //in Step() function
2 float bodyAngle = body->GetAngle();
3 b2Vec2 toTarget = clickedPoint - body->GetPosition();
4 float desiredAngle = atan2f( -toTarget.x, toTarget.y );
5
6 //view these in real time
7 m_debugDraw.DrawString(5, m_textLine, "Body angle %.3f", bodyAngle * RADTODEG);
8 m_textLine += 15;
9 m_debugDraw.DrawString(5, m_textLine, "Target angle %.3f", desiredAngle * RADTODEG);
10 m_textLine += 15;
```

Now click around the screen to see how the result changes depending on the angle between the body to the target point. Notice that if the body keeps spinning in one direction, its angle gets larger in that direction and can exceed 360 degrees, but the target angle we calculate is only ever between -180 and 180.



Now that we've seen how the angle calculation behaves, try the SetTransform method to align the body, and you should see the body align itself instantly to face at the target point.

```
1 body->SetTransform( body->GetPosition(), desiredAngle );
```



Try throwing the body so that it moves very slowly close by the target point, and you may see a side effect caused by this method not simulating physics realistically. I found that the body can either be pushed away from the target point, or go into an orbit around it. This is because for this body the center of mass is offset a little, and also I think because the angular velocity from the previous time step becomes invalid when we directly set the angle. In any case, it's worth noting that by setting the angular velocity to zero to eliminate the effects of the angular velocity from the previous time step, the problem goes away:

```
1 body->SetAngularVelocity(0);
```

To make the body turn gradually, just limit the change in angle for each time step:

```
1 float totalRotation = desiredAngle - bodyAngle;
2 float change = 1 * DEGTORAD; //allow 1 degree rotation per time step
3 float newAngle = bodyAngle + min( change, max(-change, totalRotation));
4 body->SetTransform( body->GetPosition(), newAngle );
```

Have you noticed something odd when the target position is below the body? Angles on the left are positive values, and angles on the right are negative, spanning from -180 to 180. This means that when the target crosses directly below the body, the desired angle can jump from say, 179 to -179 which causes the body to do almost a full rotation (358 degrees) even though the target was only 2 degrees away! We could fix this by noting that the body should never need to rotate more than 180 degrees to face the right direction. Whenever totalRotation exceeds 180 degrees we can adjust it like:

```
1 while ( totalRotation < -180 * DEGTORAD ) totalRotation += 360 * DEGTORAD;
2 while ( totalRotation > 180 * DEGTORAD ) totalRotation -= 360 * DEGTORAD;
```

This will also take care of the fact that the body's angle can be a very high value if it has rotated the same way many times.

Using torques

For a more physically realistic method, a torque can be applied to turn the body: We could try this to start with:

```
1 float totalRotation = desiredAngle - bodyAngle;
2 while ( totalRotation < -180 * DEGTORAD ) totalRotation += 360 * DEGTORAD;
3 while ( totalRotation > 180 * DEGTORAD ) totalRotation -= 360 * DEGTORAD;
4 body->ApplyTorque( totalRotation < 0 ? -10 : 10 );
```


Hmm... doesn't really work does it? The problem with this method is that the force is applied right up until the body is facing the right direction, which might sound ok but it means the body picks up angular momentum so that it swings right past the target point and then needs to be pushed back, and so on, forever. What we need to do is apply less force as the body gets closer to the correct angle... but simply scaling the force down relative to the remaining angle will not fix the problem either - try it.

Since the problem is caused by the current angular velocity affecting future time steps, we need to take that into account. We can calculate the angle of the body in the next time step without applying any torque - this is what would happen if we just left it alone - and use that in place of the current angle (with the default testbed framerate of 60Hz):

```
1 float nextAngle = bodyAngle + body->GetAngularVelocity() / 60.0;
2 float totalRotation = desiredAngle - nextAngle; //use angle in next time step
3 body->ApplyTorque( totalRotation < 0 ? -10 : 10 );
```

While that looks almost the same as before, if you wait a bit you'll see that at least it eventually stops in the right place instead of swinging about forever. While technically that completes this scenario, the slow convergence is probably not satisfactory for most applications. The solution is to look ahead more than one time step to adjust the rate at which the correct angle is reached. I found that looking ahead about 1/3 second gave a result like a magnetic compass needle:

```
1 float nextAngle = bodyAngle + body->GetAngularVelocity() / 3.0; // 1/3 second
```

How about an instantaneous spin? As in the previous topic, we could try a very large force for one time step, along with the 'looking ahead' idea above. The equation to find the torque to apply is the same as the linear version but uses angular velocity and angular mass. Angular mass is known as rotational inertia, commonly denoted as I .

Using the formula $T = I\omega/t$ where T is the torque we want to know, I is the rotational inertia of the body, ω is the rotational velocity and t is the time we will apply the torque, as before:

```
1 float nextAngle = bodyAngle + body->GetAngularVelocity() / 60.0;
2 float totalRotation = desiredAngle - nextAngle;
3 while ( totalRotation < -180 * DEGTORAD ) totalRotation += 360 * DEGTORAD;
4 while ( totalRotation > 180 * DEGTORAD ) totalRotation -= 360 * DEGTORAD;
5 float desiredAngularVelocity = totalRotation * 60;
6 float torque = body->GetInertia() * desiredAngularVelocity / (1/60.0);
7 body->ApplyTorque( torque );
```

Note that this is not quite instantaneous, but it usually gets the body in the right rotation within 2-3 time steps which is usually good enough for practical purposes.

Update: this will not work correctly for bodies with a center of mass that is not on their origin, like the one we have here :(Currently the Box2D API only allows access to the inertia about the body origin, whereas we are applying a torque about the center of mass. I am hoping future releases of the API will make the inertia about the center of mass available.

Using impulses

As in the last topic, instantaneous movement using impulses is the same as the above code, but without the time factor:

```
1 float nextAngle = bodyAngle + body->GetAngularVelocity() / 60.0;
2 float totalRotation = desiredAngle - nextAngle;
3 while ( totalRotation < -180 * DEGTORAD ) totalRotation += 360 * DEGTORAD;
4 while ( totalRotation > 180 * DEGTORAD ) totalRotation -= 360 * DEGTORAD;
5 float desiredAngularVelocity = totalRotation * 60;
6 float impulse = body->GetInertia() * desiredAngularVelocity; // disregard time factor
7 body->ApplyAngularImpulse( impulse );
```

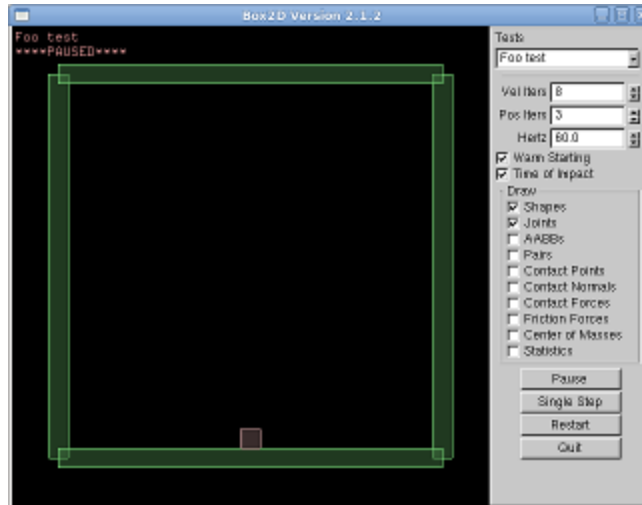
And for a gradual change, just limit the change in rotation allowed per time step:

```
1 float nextAngle = bodyAngle + body->GetAngularVelocity() / 60.0;
2 float totalRotation = desiredAngle - nextAngle;
3 while ( totalRotation < -180 * DEGTORAD ) totalRotation += 360 * DEGTORAD;
4 while ( totalRotation > 180 * DEGTORAD ) totalRotation -= 360 * DEGTORAD;
5 float desiredAngularVelocity = totalRotation * 60;
6 float change = 1 * DEGTORAD; //allow 1 degree rotation per time step
7 desiredAngularVelocity = min( change, max(-change, desiredAngularVelocity));
8 float impulse = body->GetInertia() * desiredAngularVelocity;
9 body->ApplyAngularImpulse( impulse );
```

Once again the swinging problem can be controlled by how far ahead the nextAngle variable looks.

Jumping

A platform game character's gotta jump, right? Let's take a look at some different ways of implementing a jump. We already kind of did this in the [forces and impulses](#) topic, but now we'll think about how each method would fit into a game. Start with the same scene as for the [moving at constant speed](#) topic, with the fenced in area and the left/right controls to move a dynamic body.



Setting the velocity directly

When the player character jumps their velocity changes, so let's try doing that to start with. Add a case to the Keyboard() function to get a jump input from the user:

```
1 case 'j': //jump
2 {
3     b2Vec2 vel = body->GetLinearVelocity();
4     vel.y = 10; //upwards - don't change x velocity
5     body->SetLinearVelocity( vel );
6 }
7 break;
```

Now using the j key will directly set the velocity, which as we know by now is not a physically realistic method. There are a few things to notice about this method that might make it unsuitable for a jump in a platform game. The momentum of the body is not taken into account, so the player can jump with equal strength even when they are falling quickly. The player can also jump in mid-air - however this might be what you want, say for a 'double-jump' feature. For now we'll settle for this jumping in mid-air, and in a later topic we'll look at how to determine when the player is standing on the ground or not.

Using a force

When you jump in real life, you are applying a force upwards on your body, so let's see if we can do that by using a strong force for a few timesteps. In the Keyboard() function we will simply make a note that the jump was started, and apply the force in the Step() function. Use a class member variable to keep track of how many more time steps the force should be applied:

```
1 //class member variable
2 int remainingJumpSteps;
3
4 //in constructor
```

```

5 remainingJumpSteps = 0;
6
7 //keyboard function
8 case 'k': //jump
9     remainingJumpSteps = 6; // 1/10th of a second at 60Hz
1    break;
0
1 //in Step() function
1 if ( remainingJumpSteps > 0 ) {
1     body->ApplyForce( b2Vec2(0,500), body->GetWorldCenter() );
2     remainingJumpSteps--;
1 }
3
1
4
1
5
1
6

```

Now use the k key to try this out. The behaviour is similar to before, but this time when the body is falling, jumping has less effect since the existing downward momentum is taken into account. The magnitude used for the force here is just a number that seemed about right. If you want to specify the take-off velocity though, you could use the same formula as for finding forces to move the body left/right at a desired velocity ($f=mv/t$), and apply it evenly across all time steps:

```

1 if ( remainingJumpSteps > 0 ) {
2     //to change velocity by 10 in one time step
3     float force = body->GetMass() * 10 / (1/60.0); //f = mv/t
4     //spread this over 6 time steps
5     force /= 6.0;
6     body->ApplyForce( b2Vec2(0,force), body->GetWorldCenter() );
7     remainingJumpSteps--;
8 }

```

Using an impulse

This is probably what you want in most cases, essentially the same force as above but applied in only one time step to take full effect instantly. As in the previous topics we can just leave out the time component:

```

1 case 'I':
2 {
3     //to change velocity by 10
4     float impulse = body->GetMass() * 10;
5     body->ApplyLinearImpulse( b2Vec2(0,impulse), body->GetWorldCenter() );
6 }

```

```
7 break;
```

This gives a behaviour closer to direct setting of the velocity than the force does, because each time step the force is applied, gravity gets a chance to push back. If you look closely you'll see that the forced jump does not quite go as high as the others. However if you have a situation where you want the jump to look a little softer, instead of like the character has been hit from below by a giant hammer, the force method might be useful.

Other methods

In real life, for every action there is an equal and opposite reaction. If you use your legs to push your body up into a jump, whatever you were standing on just got pushed down upon with an equal force. None of the methods above take this into account, and most likely in your game they don't need to. But if you wanted to have this modeled accurately, there are two ways you could do it, both of which we need some more study in order to implement so I'll just mention them in passing for now.

After first making sure that the player is standing on something, you could then apply the same force/impulse to that object if it's a dynamic body. This would look better when you are jumping on a swing bridge, for example, the bridge would get a realistic kick downwards as the player jumps instead of merely having the player's weight removed. For this we'll need to know how to tell what the player is touching.

Using a prismatic (sliding) joint to join two bodies together is probably the most accurate way to model a character jumping. In this method the player would have a main body like we are using here, and another smaller body joined to it which can slide vertically. The joint motor could be used to shove this smaller body downwards, much like you shove your legs downwards to jump. This has the nice advantage that we don't need to bother checking if the player is standing on anything because if it's not, pushing the 'legs' downward will just do nothing. We also get an opposite force applied downwards to whatever was under the player eg. to kick the swing bridge downwards. Furthermore, if the player is standing on something 'soft' like a swing bridge, the jump will be less effective, just like real-life. Of course apart from being more work to implement, this approach has it's own issues to consider. I hope to discuss this method in another topic after we've covered joints and joint motors.

Stopping the body from rotating

You've probably noticed the body is still freely rotating which is starting to feel a bit out of place now that we are starting to look at the body as a player character. The recommended way to stop this is simply to set the body as a fixed rotation body:

```
1 body->SetFixedRotation(true);
```

Using debug draw

The testbed makes use of a feature known as "debug draw" to draw the shapes you see. Obviously if you are making a game you will want to have all kinds of eye-popping fancy graphics instead of these boring polygons, but the debug draw feature can be very useful when you are having trouble getting the physics scene to work right. Sometimes the problem can be in the rendering part of your game, for example a sprite may be drawn at the wrong position or rotation, giving the appearance of incorrect physics. I recommend keeping the debug draw ready to hand for times when you want to check *exactly* what is going on in the Box2D world.

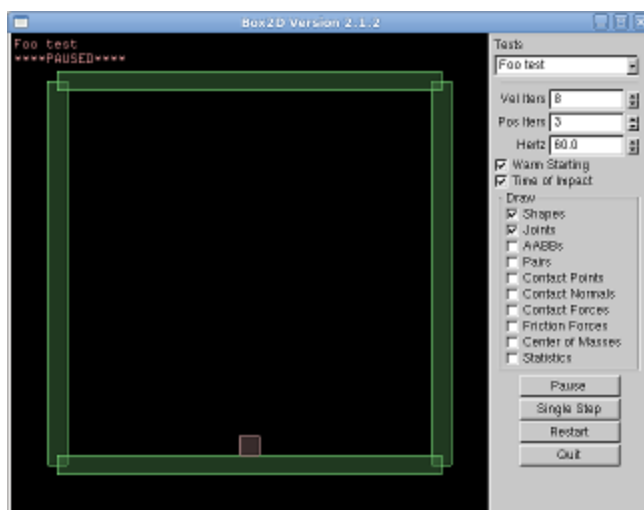
The way it works is quite simple. Box2D tells you all the shapes it sees and where they are eg. "a circle of radius r at x,y", or "an edge from a to b", etc and you draw what you are told. You don't have to do any transforming or worry about where the actual body positions are or which fixtures belong to which bodies, you simply draw the geometry. The idea is that if all you have to do is draw a few lines, you can't mess it up :-)

The default debug draw used by the testbed is done by subclassing the `b2DebugDraw` class, which has a bunch of virtual functions that can be overridden. Here are the main players:

```
1 virtual void DrawPolygon(b2Vec2* vertices, int32 vertexCount, b2Color& color) = 0;
2 virtual void DrawSolidPolygon(b2Vec2* vertices, int32 vertexCount, b2Color& color) = 0;
3 virtual void DrawCircle(b2Vec2& center, float32 radius, b2Color& color) = 0;
4 virtual void DrawSolidCircle(b2Vec2& center, float32 radius, b2Vec2& axis, b2Color& color) = 0;
5
6 virtual void DrawSegment(b2Vec2& p1, b2Vec2& p2, b2Color& color) = 0;
virtual void DrawTransform(const b2Transform& xf) = 0;
```

Separating the rendering code into one location like this makes it easy to implement debug drawing for different APIs like DirectX or OpenGL ES, or switch between different methods easily.

Although the testbed's default debug draw works just fine and we don't really need to customize it, since we're on the topic let's try using our own subclass to alter the appearance a bit. Any scene from the previous topics will be ok to use - I'll start with the same scene as for the [moving at constant speed](#) topic.



To make our own debug draw class, we will need to implement all of the pure virtual functions. For now let's just make them all empty:

```
1 class FooDraw : public b2DebugDraw
2 {
3 public:
4 void DrawPolygon(const b2Vec2* vertices, int32 vertexCount, const b2Color& color) {}
5 void DrawSolidPolygon(const b2Vec2* vertices, int32 vertexCount, const b2Color& color) {}
6 void DrawCircle(const b2Vec2& center, float32 radius, const b2Color& color) {}
7 void DrawSolidCircle(const b2Vec2& center, float32 radius, const b2Vec2& axis, const
8 b2Color& color) {}
9 void DrawSegment(const b2Vec2& p1, const b2Vec2& p2, const b2Color& color) {}
1 void DrawTransform(const b2Transform& xf) {}
0 };
```

To tell the Box2D world to use this class instead of the default one, we use the SetDebugDraw function. This function takes a pointer to a b2DebugDraw object, so we'll need to have an instance of the class to point to. This is easily done by just declaring a variable of your new class at global scope.

```
1 //at global scope
2 FooDraw fooDrawInstance;
3
4 //in constructor, usually
5 m_world->SetDebugDraw( &fooDrawInstance );
6
7 //somewhere appropriate
8 fooDrawInstance.SetFlags( b2DebugDraw::e_shapeBit );
```

This will tell the world what class instance it should direct all the drawing instructions to. Notice the last part which selects a certain category of debug information to display. Right now we are interested in seeing the shapes (fixtures) in the world, but you can also set this flag to include the following:

- e_shapeBit (draw shapes)
- e_jointBit (draw joint connections)
- e_aabbBit (draw axis aligned bounding boxes)
- e_pairBit (draw broad-phase pairs)
- e_centerOfMassBit (draw a marker at body CoM)

I say to put this setting 'somewhere appropriate' because it's something you might want to alter at run time, like in the situation I mentioned above when you might occasionally want to confirm that your rendered game entities and Box2D are doing the same thing. In the testbed, you can see these settings as the checkboxes on the right hand panel.

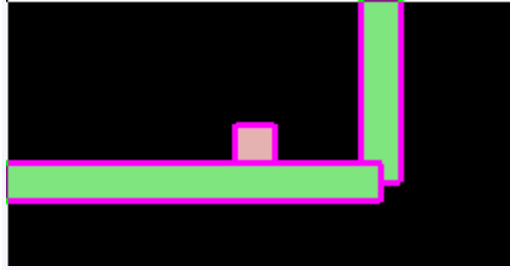
If you use a debug draw class in your own projects, **there is one other important point to be aware of:** you need to call the DrawDebugData() function of your world, which will in turn cause Box2D to call back to your debug draw class functions for each shape that needs drawing.

Running the testbed now you should see nothing showing in the scene. This is because we still have a completely empty debug draw implementation. From this point, what you fill in the drawing

functions depends on what platform and rendering API you are using. As an example, let's implement the DrawSolidPolygon function in OpenGL ES, as used on embedded platforms such as the iPhone. This is a handy example because OpenGL ES is a subset of OpenGL so we can still run it on our PC in the testbed as normal, and also because it's one of those questions that comes up often.

OpenGL ES does not have the glBegin/glEnd/glVertex functions, so rendering is done with vertex arrays instead:

```
1 void DrawSolidPolygon(const b2Vec2* vertices, int32 vertexCount, const b2Color& color)
2 {
3     //set up vertex array
4     GLfloat glverts[16]; //allow for polygons up to 8 vertices
5     glVertexPointer(2, GL_FLOAT, 0, glverts); //tell OpenGL where to find vertices
6     glEnableClientState(GL_VERTEX_ARRAY); //use vertices in subsequent calls to glDrawArrays
7
8     //fill in vertex positions as directed by Box2D
9     for (int i = 0; i < vertexCount; i++) {
10         glverts[i*2] = vertices[i].x;
11         glverts[i*2+1] = vertices[i].y;
12     }
13
14     //draw solid area
15     glColor4f( color.r, color.g, color.b, 1);
16     glDrawArrays(GL_TRIANGLE_FAN, 0, vertexCount);
17
18     //draw lines
19     glLineWidth(3); //fat lines
20     glColor4f( 1, 0, 1, 1 ); //purple
21     glDrawArrays(GL_LINE_LOOP, 0, vertexCount);
22 }
```

The other drawing functions can be done in much the same way. Depending on what rendering API you are using, circles might have to be drawn as a polygon with many sides.

Update: I noticed that there is a good implementation for an OpenGL ES debug draw in the Box2D source code iPhone contribution. This is written for Obj-C but it appears to be regular C++ so you can use it without modification:

[GLES-Render.mm](#)

Drawing your own objects

From the previous topic, it's obvious that using debug draw will not make for a very visually appealing game. Usually we would like to use our own method for drawing objects in the scene, and access the physics information from Box2D so that we know where to draw them. In this topic, we'll set up a class to use as a game entity, and then look at how to keep it in the right place. We will do this by storing a pointer to a Box2D body in the game entity.

For now all the entity class will do is render itself, but later we will expand it to demonstrate other topics. The focus of these tutorials is not on rendering, so we'll just draw a basic smiley face on a circle shape to confirm that it moves and rotates correctly.

For this topic let's set up an empty fenced area to start with.

```

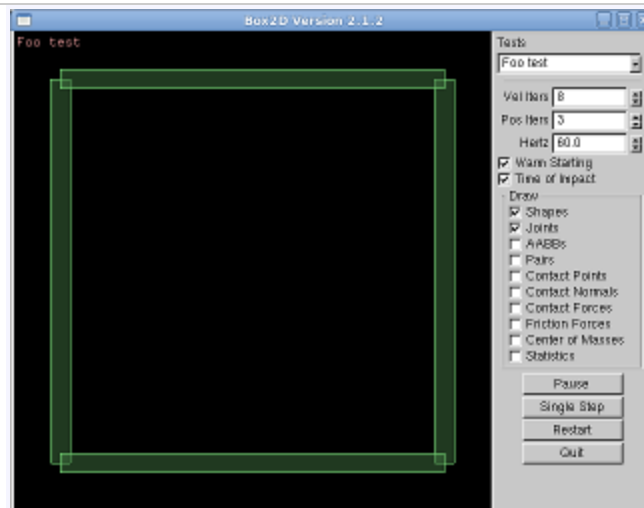
1  FooTest() {
2      //a static body
3      b2BodyDef myBodyDef;
4      myBodyDef.type = b2_staticBody;
5      myBodyDef.position.Set(0, 0);
6      b2Body* staticBody = m_world->CreateBody(&myBodyDef);
7
8      //shape definition
9      b2PolygonShape polygonShape;
10
11     //fixture definition
12     b2FixtureDef myFixtureDef;
13     myFixtureDef.shape = &polygonShape;
14
15     //add four walls to the static body
16     polygonShape.SetAsBox( 20, 1, b2Vec2(0, 0), 0); //ground
17     staticBody->CreateFixture(&myFixtureDef);
18     polygonShape.SetAsBox( 20, 1, b2Vec2(0, 40), 0); //ceiling
19     staticBody->CreateFixture(&myFixtureDef);

```

```

1 polygonShape.SetAsBox( 1, 20, b2Vec2(-20, 20), 0);//left wall
5 staticBody->CreateFixture(&myFixtureDef);
1 polygonShape.SetAsBox( 1, 20, b2Vec2(20, 20), 0);//right wall
6 staticBody->CreateFixture(&myFixtureDef);
1 }
7
1
8
1
9
2
0
2
1
2
2
2
3
2
4

```



Let's call our game entity class Ball since it will be round and bouncing:

```

1 //outside and before the FooTest class
2 class Ball {
3 public:
4 //class member variables
5 b2Body* m_body;
6 float m_radius;
7
8 public:
9 Ball(b2World* world, float radius) {
10     m_body = NULL;
11     m_radius = radius;

```

```

1  }
1  ~Ball() {}
1
2  };
1
3  //FooTest class member variable
1  std::vector<Ball*> balls;
4
1
5
1
6
1
7
1
8

```

(To get the last part to compile, you may have to #include <vector> at the top of the file.) Notice that now instead of storing references to Box2D bodies directly, we let each game entity look after that and we store a reference to the game entities instead. Add a render function to the Ball class to draw a nice smiley face. The important thing to note here is that we are drawing the face so that it is centered on the point (0,0) and it will not be rotated. The radius is taken to be 1 for the default rendering.

```

1  //Ball::render
2  void render() {
3      glColor3f(1,1,1);//white
4
5      //nose and eyes
6      glPointSize(4);
7      glBegin(GL_POINTS);
8      glVertex2f( 0, 0 );
9      glVertex2f(-0.5, 0.5 );
1     glVertex2f( 0.5, 0.5 );
0     glEnd();
1
1     //mouth
1     glBegin(GL_LINES);
2     glVertex2f(-0.5, -0.5 );
1     glVertex2f(-0.16, -0.6 );
3     glVertex2f( 0.16, -0.6 );
1     glVertex2f( 0.5, -0.5 );
4     glEnd();
1
5     //circle outline
1     glBegin(GL_LINE_LOOP);
6     for (float a = 0; a < 360 * DEGTORAD; a += 30 * DEGTORAD)
1         glVertex2f( sinf(a), cosf(a) );
7     glEnd();

```

```

1  }
8
1
9
2
0
2
1
2
2
2
3
2
4
2
5
2
6

```

Just a couple more things to do. In the FooTest constructor, after the fenced area is set up, add a Ball entity into the scene (this should really be deleted in the class destructor if you wish to add that too).

```

1  //add ball entity to scene in constructor
2  Ball* ball = new Ball(m_world, 1);
3  balls.push_back( ball );

```

And finally, to actually draw the ball entities we'll need to add to the Step() function. If you put this after the call to Test::Step(), the ball will be drawn on top of the existing debug draw data.

```

1  //in Step() function, after Test::Step()
2  for (int i = 0; i < balls.size(); i++)
3      balls[i]->render();

```



Now we have one ball entity, but it's being drawn at the default location of (0,0) and we didn't need a physics engine to do that huh? Let's add to the constructor of the Ball class to set up a circle shape body for the ball. Now it becomes apparent why the Ball constructor requires the b2World pointer:

```

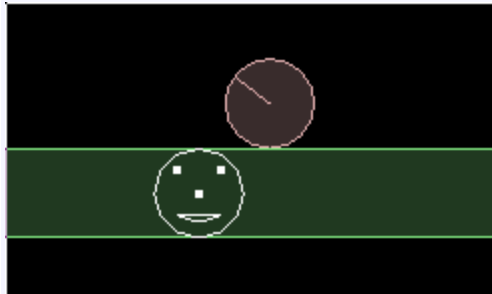
1  // Ball class constructor
2  Ball(b2World* world, float radius) {

```

```

3  m_body = NULL;
4  m_radius = radius;
5
6  //set up dynamic body, store in class variable
7  b2BodyDef myBodyDef;
8  myBodyDef.type = b2_dynamicBody;
9  myBodyDef.position.Set(0, 20);
10 m_body = world->CreateBody(&myBodyDef);
11
12 //add circle fixture
13 b2CircleShape circleShape;
14 circleShape.m_p.Set(0, 0);
15 circleShape.m_radius = m_radius; //use class variable
16 b2FixtureDef myFixtureDef;
17 myFixtureDef.shape = &circleShape;
18 myFixtureDef.density = 1;
19 m_body->CreateFixture(&myFixtureDef);
20 }

```



Ok, there is a physics body in the scene but our face is not being drawn at the body's position. To do this, let's add another function to the Ball class to set up the OpenGL transform before rendering. Depending on what rendering API you use, there may be better methods for doing this.

```

1  //in Ball class
2  void renderAtBodyPosition() {
3
4  //get current position from Box2D
5  b2Vec2 pos = m_body->GetPosition();
6  float angle = m_body->GetAngle();

```

```

7
8 //call normal render at different position/rotation
9 glPushMatrix();
10 glTranslatef( pos.x, pos.y, 0 );
11 glRotatef( angle * RADTODEG, 0, 0, 1 );//OpenGL uses degrees here
12 render();//normal render at (0,0)
13 glPopMatrix();
14 }
15
16
17
18
19
20

```

Don't forget to change the rendering call in the Step() function to use the new `renderAtBodyPosition`. Now even if you turn the debug draw display off (uncheck the 'Shapes' checkbox on in the control panel) your own rendering code still draws the ball at the correct position and rotation.



Just for fun we could use some more feedback from the physics engine to change the color of the ball at different speeds. For example, set the color like this:

```

1 //in Ball::render
2 b2Vec2 vel = m_body->GetLinearVelocity();
3 float red = vel.Length() / 20.0;
4 red = min( 1, red );
5 glColor3f(red,0.5,0.5);

```

How about adding a whole bunch of balls to the scene - just change the `FooTest` constructor to loop a few times:

```

1 //in FooTest constructor
2 for (int i = 0; i < 20; i++) {
3     Ball* ball = new Ball(m_world, 1);
4     balls.push_back( ball );
5 }

```



We could make the balls random sizes too, since variable sizes were considered when we set up the Ball class:

```
1 for (int i = 0; i < 20; i++) {
2     float radius = 1 + 2 * (rand()/(float)RAND_MAX); //random between 1 - 3
3     Ball* ball = new Ball(m_world, radius);
4     balls.push_back( ball );
5 }
```



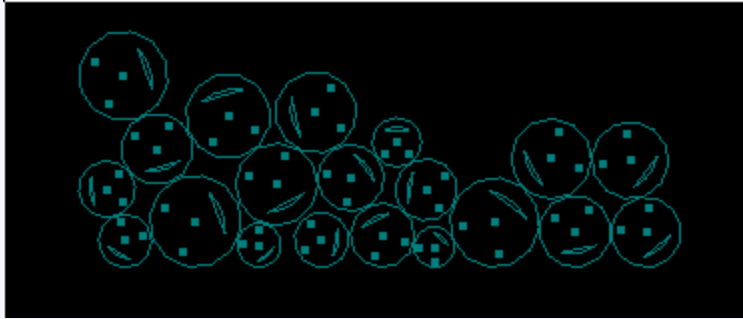
Uh-oh... something's not right here, the balls don't collide properly anymore. Or at least that's what you could be thinking if this was a more complex scene, and you had confidently thrown away the debug draw code in favour of your eye-popping fancy graphics. Since we have the debug draw ready to hand, it only takes a second to see what's really going on in the physics



engine:

Okay so that was a bit contrived, but I just wanted to illustrate the usefulness of having a debug draw implementation available throughout your development progress. Finally, to fix the size problem we just need to scale the rendering code:

```
1 //inside Ball::renderAtBodyPosition
2 glPushMatrix();
3 glTranslatef( pos.x, pos.y, 0 );
4 glRotatef( angle * RADTODEG, 0, 0, 1 );
5 glScalef( m_radius, m_radius, 1 ); //add this to correct size
6 render();
7 glPopMatrix();
```



User data

In the previous topic, we saw how useful it was to have a reference to a physics object from a game entity class. Sometimes it is also useful to have the opposite - a pointer from a physics object to an entity in your game. In Box2D this is called user data, and it is just a pointer which you can set to hold some information that may be useful for you. The following classes have this functionality:

- `b2Body`
- `b2Fixture`
- `b2Joint`

Box2D doesn't care what this information is, and it doesn't do anything with it. It just holds it and tells you what it is when you ask. The above classes all have the following functions to do this:

```
1 //in b2Body, b2Fixture, b2Joint
2 void SetUserData(void* data);
3 void* GetUserData();
```

Setting something in the user data for bodies and fixtures will be extremely useful in upcoming topics, so let's try a simple example to get the hang of it. In this example we will implement exactly what we just did in the previous topic (getting the render position and velocity of the game entity) but we will do it without storing a pointer to the physics body in the Ball class. Instead, we will store a pointer to a Ball object in each physics body, and after each time step we will update class member variables of the ball object with the new information. Note that this is not really a practical way to approach the task, it's just for demonstration.

In order to access the user data for each physics body, we'll also need to try out a method for looping across all bodies in the scene. Instead of a loop in the proper sense, this is done by a linked list of bodies. We can use `b2World::GetBodyList()` to get the first element in the list to start iterating on.

So, start with the source code from the previous topic and modify it a bit. Firstly, take out all references to the `m_body` member variable of the Ball class, and alter its constructor to set the Ball class itself in the user data of the created physics body:

```
1 Ball(b2World* world, float radius) {
2     m_radius = radius;
3 }
```



```

4 //set up dynamic body
5 b2BodyDef myBodyDef;
6 myBodyDef.type = b2_dynamicBody;
7 myBodyDef.position.Set(0, 20);
8 b2Body* body = world->CreateBody(&myBodyDef);
9
1 //set this Ball object in the body's user data
0 body->SetUserData( this );
1
1 //add circle fixture
1 b2CircleShape circleShape;
2 circleShape.m_p.Set(0, 0);
1 circleShape.m_radius = m_radius;
3 b2FixtureDef myFixtureDef;
1 myFixtureDef.shape = &circleShape;
4 myFixtureDef.density = 1;
1 body->CreateFixture(&myFixtureDef);
5 }
1
6
1
7
1
8
1
9
2
0
2
1

```

Next, add some member variables to the Ball class to hold the physics information we'll need:

```

1 b2Vec2 m_position;
2 float m_angle;
3 b2Vec2 m_linearVelocity;

```

... and replace the locations where calls to `m_body->GetPosition()`, `m_body->GetAngle()` and `m_body->GetLinearVelocity()` were previously used, to simply use these new member variables. For example, the section in `renderAtBodyPosition` to position the smiley face will now be:

```

1 glTranslatef( m_position.x, m_position.y, 0 );
2 glRotatef( m_angle * RADTODEG, 0, 0, 1 );

```

Running this as is, you'll see we are back at the initial stage of the last topic where the all the balls are rendered at (0,0) with no rotation:



So to fix this in the Step() function, after the Test::Step() call and before we do our own rendering, we need to update the ball object positions from their physics bodies. To do this, iterate over all bodies in the world, get the necessary position/angle/velocity values and set them in the Ball object contained in the user data:

```
1  b2Body* b = m_world->GetBodyList();//get start of list
2  while ( b != NULL ) {
3
4      //obtain Ball pointer from user data
5      Ball* ball = static_cast<Ball*>( b->GetUserData() );
6      if ( ball != NULL ) {
7          ball->m_position = b->GetPosition();
8          ball->m_angle = b->GetAngle();
9          ball->m_linearVelocity = b->GetLinearVelocity();
10     }
11
12     //continue to next body
13     b = b->GetNext();
14 }
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
```

This should get the behaviour back to how it was before. Once again, this is not a recommended way of linking your game entities to a physics body for the purposes of rendering, it's just to demonstrate the user data functionality. For rendering, the previous method was much easier to implement and manage. Driving the information updates from the physics body doesn't help us much, because we are continuously rendering all the time regardless of what's happening in the physics simulation.

So what kind of practical uses does this user data feature have then? The usefulness comes when we want to be informed about something happening in the physics engine that we *can't* easily predict, such as when fixtures collide, what they collided with and what the reaction forces will be, etc. Other useful things include ray-casting or AABB queries to find intersected fixtures (and get the related game entity from the user data in the fixture).

Setting complex user data

Since the user data accepts a void pointer, anything that can be cast to a void pointer can be set in the user data. This could be a single number, an existing object pointer (as above) or a pointer that you make specifically to hold some complex information relating to the physics object. Here are some examples:

```
1 //setting and retrieving an integer
2 int myInt = 123;
3 body->SetUserData( (void*)myInt );
4 ...later...
5 int udInt = (int)body->GetUserData();
6
7
8 //setting and retrieving a complex structure
9 struct bodyUserData {
10     int entityType;
11     float health;
12     float stunnedTimeout;
13 };
14 bodyUserData* myStruct = new bodyUserData;
15 myStruct->health = 4;//set structure contents as necessary
16 body->SetUserData(myStruct);
17 ...later...
18 bodyUserData* udStruct = (bodyUserData*)body->GetUserData();
19
20
21
22
23
24
25
26
27
28
```

In each case, you should set the same type of value for the user data. For example, if you give a fixture an integer user data as in the first example here, then all fixtures should be given an integer for their user data. If you were to give some fixtures an integer and other fixtures a structure, when retrieving the user data from a fixture with `GetUserData()` it would be difficult to tell whether you are dealing with an integer or a structure (the [collision callbacks](#) topic will explain why you cannot always know which fixture or body you are dealing with when a collision is detected).

For most applications it is very handy to set a structure with multiple members in the user data. Box2D does not delete any of your user data objects when you destroy a body/fixture/joint, so you must remember to clean these up yourself when they are no longer needed.

In the structure example above the members of the structure are fixed, which gives a limited set of attributes to use, and not much flexibility if you have different entity types in your game. See

the section of the collision callbacks topic titled '[real scenarios](#)' for an example of using class inheritance for a more practical approach for when you have many types of entities.

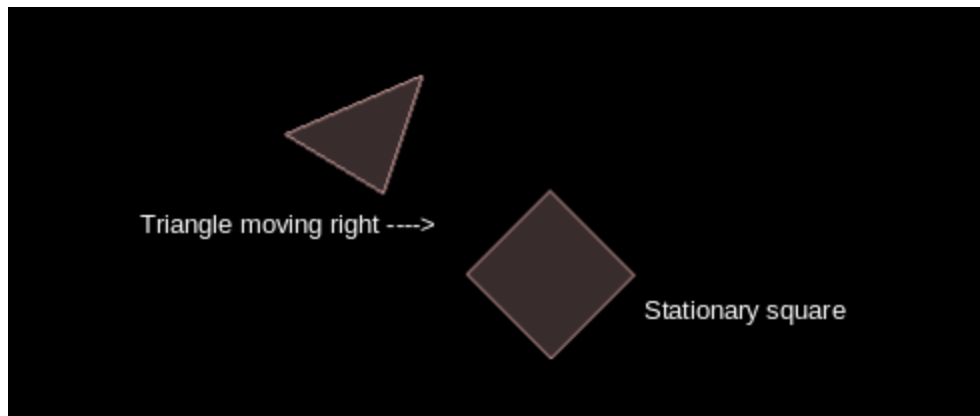
What's in a collision?

In Box2D it's common to think of bodies colliding with each other, but it's really the fixtures which are used to detect when a collision occurs. Collisions can happen in all kinds of ways so they have a lot of information that can be used in the game logic. For example, you might want to know:

- when a collision starts and ends
- what point on the fixtures is touching
- the normal vector of the contact between the fixtures
- how much energy was involved and how the collision was responded to

Usually collisions happen very quickly, but in this topic we are going to take one collision and slow it riiiiight down so we can take a look at the details of what is going on, and all the possible information we can get from it.

The scenario will be two polygon fixtures colliding, in a zero-gravity world so we can control it easier. One fixture is a stationary box, the other is a triangle moving horizontally towards the box.



This scenario is set up so that the bottom of the triangle will just collide with the top corner of the box. For this topic the finer details of this arrangement are not important, the focus is on what kind of information we can get at each step of the process so I will direct you to the source code if you'd like to replicate this.

Getting info about collisions

Information about a collision is contained in a `b2Contact` object. From this you can check which two fixtures are colliding, and find out about the location and direction of the collision reaction. There are two main ways you can get these `b2Contact` objects from Box2D. One is to look at the current list of contacts for each body, and the other is to use a contact listener. Let's take a quick look at each of these so we know what we're talking about in the rest of this topic.

- **Checking the list of contacts**

- You can look at all the contacts in the world anytime by checking the world's contact list:

```
• for (b2Contact* contact = world->GetContactList(); contact; contact =
  contact->GetNext())
  contact->... //do something with the contact
```

- Or you can look at the contacts for a single body:

```
• for (b2ContactEdge* edge = body->GetContactList(); edge; edge = edge->next)
  edge->contact->... //do something with the contact
```

- A very important point to note if you do this, is that **the existence of a contact in these lists does not mean that the two fixtures of the contact are actually touching** - it only means their AABBs are touching. If you want to know if the fixtures themselves are really touching you can use `IsTouching()` to check. Moron that later.

-
-
-

- **Contact listeners**

- Checking the list of contacts becomes inefficient for large scenarios where many collisions are occurring frequently. Setting a contact listener allows you to have Box2D tell you when something interesting happens, rather than you doing all the work of keeping track of when things start and stop touching. A contact listener is a class (`b2ContactListener`) with four functions which you override as necessary.

```
• void BeginContact(b2Contact* contact);
  void EndContact(b2Contact* contact);
  void PreSolve(b2Contact* contact, const b2Manifold* oldManifold);
  void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse);
```

- Note that depending on what is happening, some events give us more than just the `b2Contact` object. During the world's Step function, when Box2D detects that one of these events has occurred, it will 'call back' to these functions to let you know about it. Practical applications of these '[collision callbacks](#)' will be looked at in other topics, for now we are focusing on when they occur.

-
-

Generally I would recommend the contact listeners method. It may seem a little unwieldy at first but is more efficient and more useful in the long run. I have yet to come across a situation where checking the contact lists would have any great advantage.

Either way you get these contacts, they contain the same information. The most fundamental piece of info is which two fixtures are colliding, which is obtained by:

```
1 b2Fixture* a = contact->GetFixtureA();
2 b2Fixture* b = contact->GetFixtureB();
```

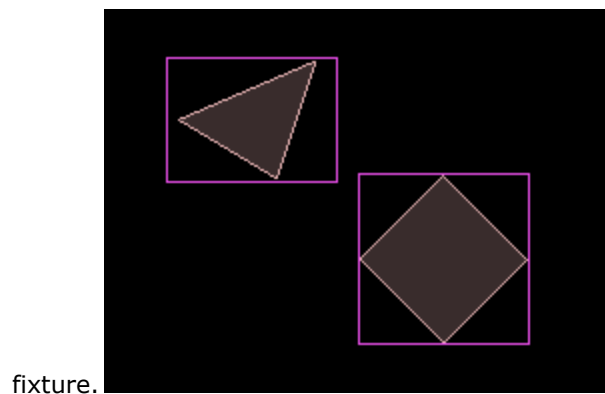
If you are checking the contact lists of a body, you may already know what one of the fixtures of the contact will be, but if you are using a contact listener you will rely on these two functions to find out what is colliding with what. There is no particular ordering of the A and B fixtures, so you will often need to have [user data](#) set in the fixtures or their bodies so you can tell what object the

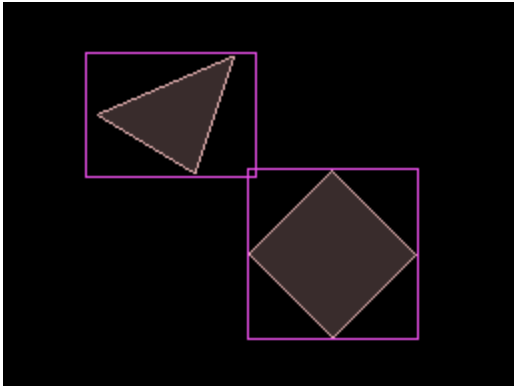
fixtures belong to. From these fixtures, you can `GetBody()` to find the bodies that collided.

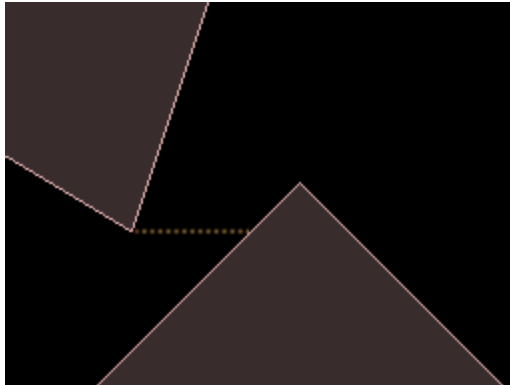
Breakdown of a collision

Now let's take an in-depth look at the sequence of events in the collision above. Hopefully this will be easy to follow in table form, with a visual reference on the left to show what the scene looks like at each step. You might like to download the tutorials testbed from the [source code](#) page and run it while reading. In the testbed you can pause the simulation and then restart, then press 'Single Step' to see things happening in detail.

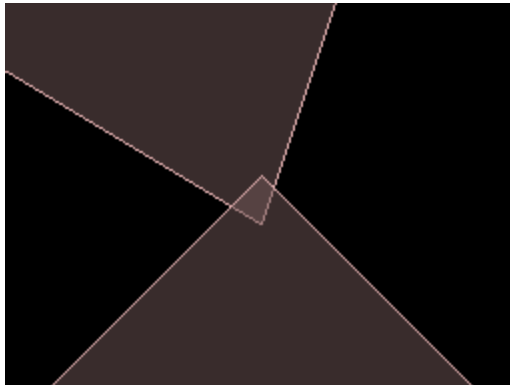
We should start from a point where the AABBs of the fixtures are still not overlapping, so we can follow the whole story. Click the 'AABBs' checkbox to see these as purple rectangles around each



	<p>Fixture AABs begin to overlap <small>(b2ContactManager::AddPair)</small></p> <p>Although the fixtures themselves are not yet overlapping, at this point a <code>b2Contact</code> is made and added to the list of contacts for the world and the list of contacts for each body. If you are checking these contact lists as shown above you will be able to tell that a collision could potentially occur here, but in most cases you don't really care until the fixtures themselves really overlap.</p> <p><u>Outcome:</u></p> <ul style="list-style-type: none"> • Contact exists but <code>IsTouching()</code> returns false
<p>Continues until fixtures themselves overlap...</p>	
<p>Step n</p>	<p>Fixtures begin to overlap <small>(b2Contact::Update)</small></p>



Step n+1 (non bullet bodies)



Step n+1 (triangle as bullet body)



Zooming in on the upper corner of the box, you will see the transition as shown in the upper two images on the left. This occurs *in one time step*, which means that the real collision point (as shown by the dotted line in the top image) has been skipped over. This is because Box2D moves all the bodies and then checks for overlaps between them, at least with the default settings. If you want to get the real impact position you can set the body to be a 'bullet' body which will give you the result shown in the bottom image. You can do this by:

```
bodyDef.bullet = true; //before
                        creating body, or
body->SetBullet(true); //after
                        creating body
```

Bullet bodies take more CPU time to calculate this accurate collision point, and for many applications they are not necessary. Just be aware that with the default settings, sometimes collisions can be missed completely - for example if the triangle in this example had been moving faster it may have skipped right over the corner of the box! If you have very fast moving bodies that must NOT skip over things like this, for example uh... bullets :) then you will need to set them as bullet bodies. For the rest of this discussion we will be continuing with the non-bullet body setting.

Outcome:

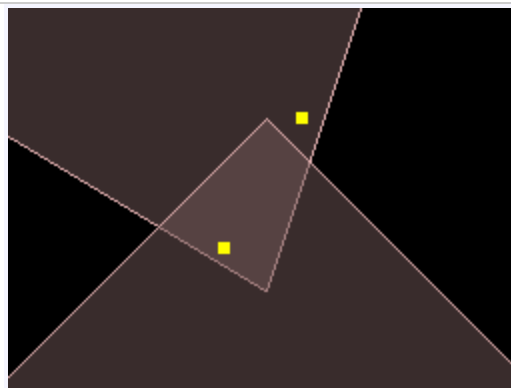
- IsTouching() now returns true
- BeginContact callback function will be called

Collision points and the normal

At this point we have a contact which is actually touching, so that means we should be able to answer some of the questions at the beginning of this topic. First let's get the location and normal for the contact. For the code sections below, we'll assume that these are either inside the `BeginContact` of the contact listener, or that you have obtained a contact to use by manually checking the contact lists of the body or world.

Internally, the contact stores the location of the collision point in local coordinates for the bodies involved, and this is usually not so great for us. But we can ask the contact to give us a more useful 'world manifold' which will hold the collision location in world coordinates. 'Manifold' is just a fancy name for the line which best separates the two fixtures.

```
1 //normal manifold contains all info...
2 int numPoints = contact->GetManifold()->pointCount;
3
4 //...world manifold is helpful for getting locations
5 b2WorldManifold worldManifold;
6 contact->GetWorldManifold( &worldManifold );
7
8 //draw collision points
9 glBegin(GL_POINTS);
10 for (int i = 0; i < numPoints; i++)
11     glVertex2f(worldManifold.points[i].x, worldManifold.points[i].y);
12 glEnd();
```



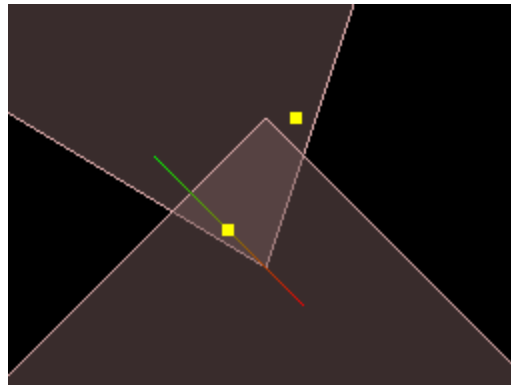
These are the points that will be used in the collision reaction when applying an impulse to push the fixtures apart.

Although they are not at the exact location where the fixtures would first have touched (unless you're using bullet-type bodies), in practise these points are often adequate for use as collision points in game logic.

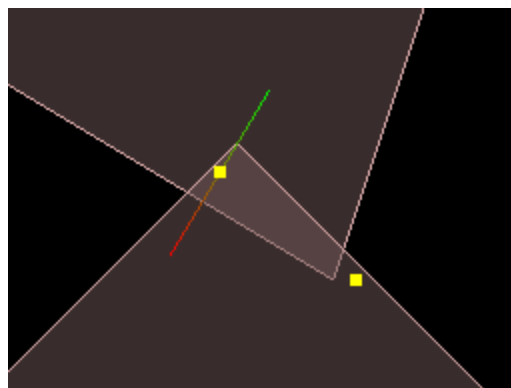
Next let's show the collision normal, which points from fixtureA to fixtureB:

```
1         float normalLength = 0.1f;
2         b2Vec2 normalStart = worldManifold.points[0] - normalLength * worldManifold.normal;
3         b2Vec2 normalEnd = worldManifold.points[0] + normalLength * worldManifold.normal;
4
5         glBegin(GL_LINES);
6         glColor3f(1,0,0); //red
7         glVertex2f( normalStart.x, normalStart.y );
8         glColor3f(0,1,0); //green
9         glVertex2f( normalEnd.x, normalEnd.y );
1        glEnd();
10
```

It seems that for this collision, the quickest way to resolve the overlap is to apply an impulse that will push the corner of the triangle up and left, and the corner of the square down and right. Please note that the normal is only a direction, it does not have a location and is not connected to either one of the points - I'm just drawing it at the location of points[0] for convenience.



It's important to be aware that **the collision normal does not give you the angle that these fixtures collided at** - remember the triangle was moving horizontally here, right? - it only gives the shortest direction to move the fixtures so that they don't overlap. For example, imagine if the triangle had been moving a little faster and the overlap was like this:



... then the shortest way to separate the two fixtures would be to push the triangle up and right. So it should be clear that using this normal as an indication of the angle that the fixtures collided at is not a good idea. If you want to know the actual direction that these two corners impacted at, you can use:

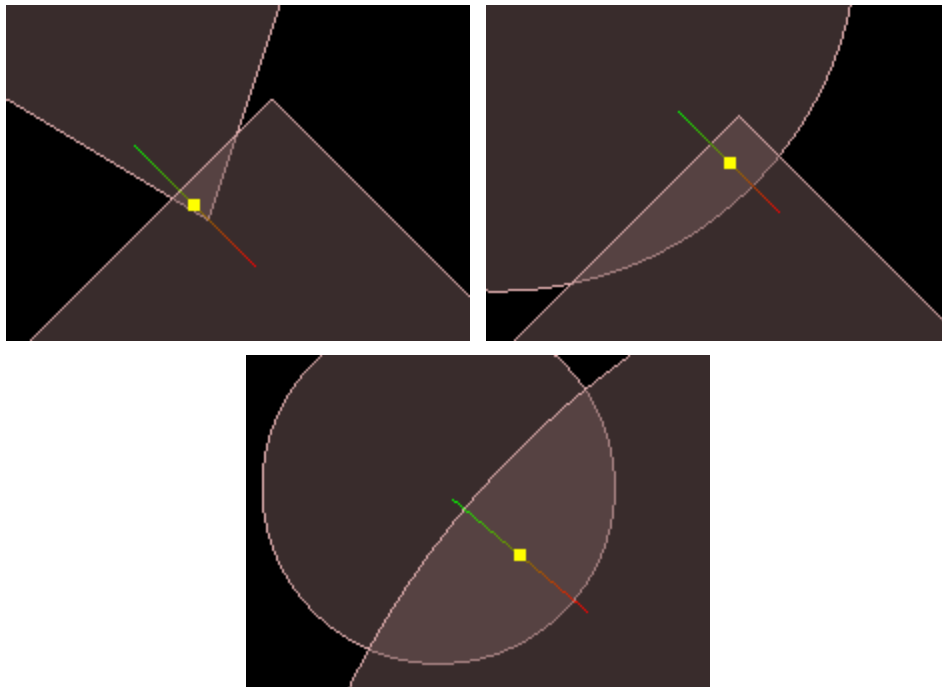
```

1 b2Vec2 vel1 = triangleBody->GetLinearVelocityFromWorldPoint( worldManifold.points[0] );
2 b2Vec2 vel2 = squareBody->GetLinearVelocityFromWorldPoint( worldManifold.points[0] );
3 b2Vec2 impactVelocity = vel1 - vel2;

```

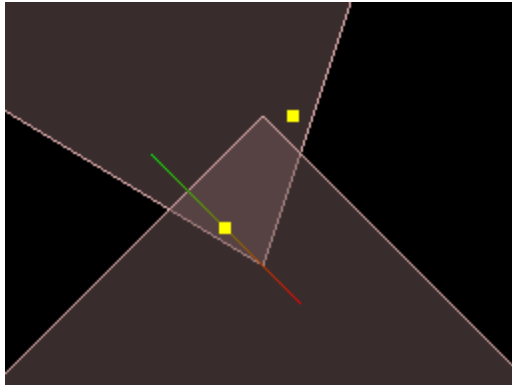
... to get the actual relative velocity of the points on each body that collided. For the simple example we are looking at we could also simply take the linear velocity of the triangle because we know the square is stationary and the triangle is not rotating, but the above code will take care of cases when both bodies could be moving or rotating.

Another thing to note is that not every collision will have two of these collision points. I have deliberately chosen a relatively complex example where two corners of a polygon are overlapping, but more common collisions in real situations have only one such point. Here are some other collision examples where only one point is necessary:

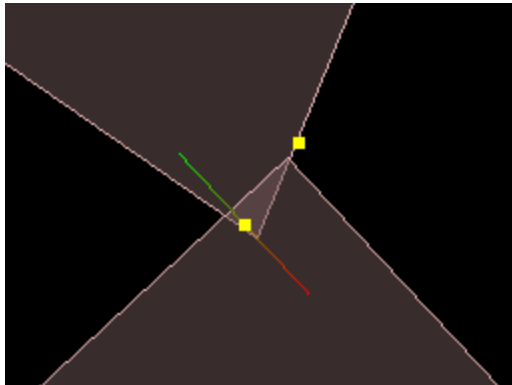


Okay, so we've seen how to find the collision points and normal, and we are aware that these points and the normal will be used by Box2D to apply a collision response to correct the overlap. Let's get back to the sequence of events...

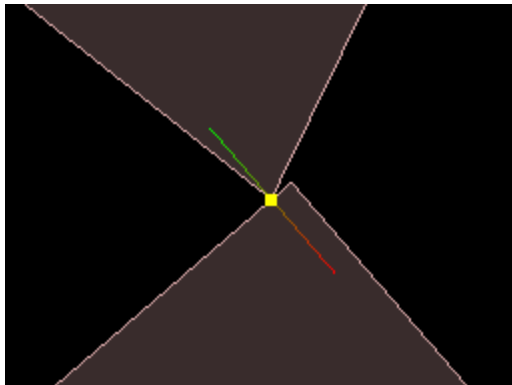
While fixtures continue to overlap...	
Impact	<p>Collision response is applied <small>(b2Contact::Update, b2Island::Report)</small></p> <p>When fixtures are overlapping, Box2D's default behavior is to apply an impulse to each of them to push them apart, but this does not always succeed in a single time step. As shown here, for this</p>



Impact + 1



Impact + 2



particular example the two fixtures will be overlapping for three time steps before the 'bounce' is complete and they separate again.

During this time we can step in and customize this behavior if we want to. If you are using the contact listener method, the PreSolve and PostSolve functions of your listener will be repeatedly called **inevery time step while the fixtures are overlapping**, giving you a chance to alter the contact before it is processed by the collision response (PreSolve) and to find out what impulses were caused by the collision response after it has been applied (PostSolve).

To make this clearer, here is the output obtained for this example collision by putting a simple printf statement in the main Step function and each of the contact listener functions:

```
...
Step
Step
BeginContact
PreSolve
PostSolve
Step
PreSolve
PostSolve
Step
PreSolve
PostSolve
Step
EndContact
Step
Step
...
```

Outcome:

- PreSolve and PostSolve are called repeatedly

PreSolve and PostSolve

Both PreSolve and PostSolve give you a `b2Contact` pointer, so we have access to the same points and normal information we just looked at for `BeginContact`. PreSolve gives us a chance to change the characteristics of the contact before the collision response is calculated, or even to cancel the response altogether, and from PostSolve we can find out what the collision response was.

Here are the alterations you can make to the contact in PreSolve:

```
1 void SetEnabled(bool flag); //non-persistent - need to set every time step
2
3 //these available from v2.2.1
4 void SetFriction(float32 friction); //persists for duration of contact
5 void SetRestitution(float32 restitution); //persists for duration of contact
```


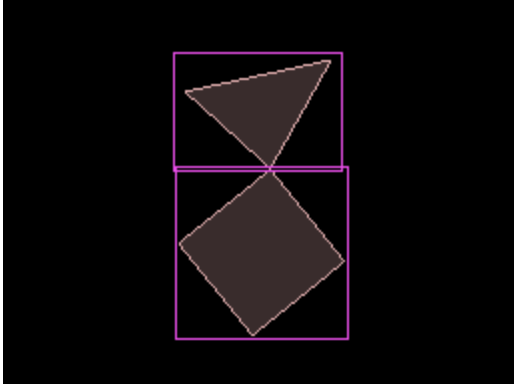
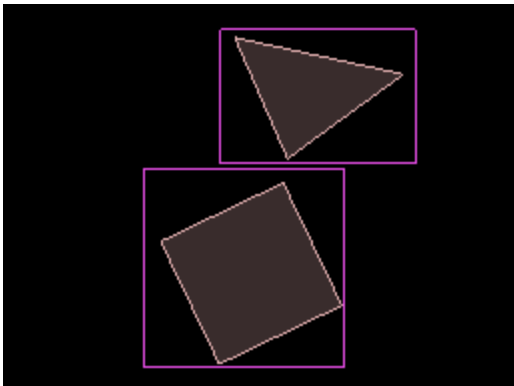
Calling `SetEnabled(false)` will disable the contact, meaning that the collision response that normally would have been applied will be skipped. You can use this to temporarily allow objects to pass through each other. A classic example of this is the one-way wall or platform, where the player is able to pass through an otherwise solid object, depending on various criteria that can only be checked at runtime, like the position of the player and which direction they are heading, etc.

It's important to note that the **contact will revert back to being enabled in the next time step**, so if you want to disable contacts like this you'll need to call `SetEnable(false)` every time step.

As well as the contact pointer, PreSolve has a second parameter from which we can find out info about the collision manifold of the previous time step. If anybody has an idea about what this could be used for, let me know :D

PostSolve is called after the collision response has been calculated and applied. This also has a second parameter, in which we can find information about the impulse that was applied. A common use for this information is to check if the size of the collision response was over a given threshold, perhaps to check if an object should break, etc. See the ['sticky projectiles'](#) topic for an example of using PostSolve to determine whether an arrow should stick into a target when it hits.

Okay, on with the timeline...

 <p>(zoomed out)</p> 	<p>Fixtures finish overlapping <small>(b2Contact::Update)</small></p> <p>The AABBs are still overlapping, so the contact remains in the contact list for the body/world.</p> <p><u>Outcome:</u></p> <ul style="list-style-type: none"> • EndContact callback function will be called • IsTouching() now returns false
<p>Continues until fixture AABBs no longer overlap...</p>	
	<p>Fixture AABBs finish overlapping <small>(b2ContactManager::Collide)</small></p> <p><u>Outcome:</u></p> <ul style="list-style-type: none"> • Contact is removed from body/world contact list

While the EndContact call to the contact listener passes a b2Contact pointer, at this point the fixtures are no longer touching, so there will be no valid manifold information to get. However the EndContact event is still an indispensable part of the contact listener because it allows you to check which fixtures/bodies/game objects have ended contact. See the next topic for a basic example usage.

Summary

I hope this topic gives a clear overview of the events going on millisecond-by-millisecond under the hood of a Box2D collision. It may not have been the most interesting read (it sure was the least exciting topic to write so far!) but I get the feeling from reading questions on the forums that some of the details discussed here are often missed, and a lot of time is spent on various workarounds and wondering what's going on. I've also noticed a tendency to shy away from implementing a contact listener, when the listener usually becomes less work in the long run. Knowing these details should allow for a better understanding of what is actually possible, better design, and time saved in implementation.

Collision callbacks

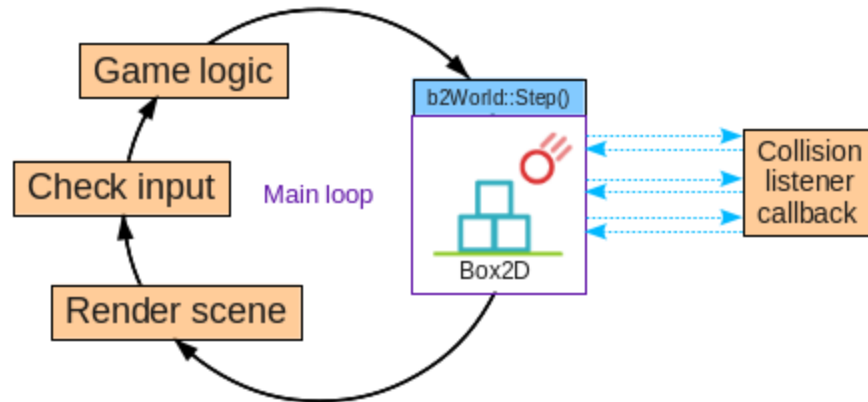
When bodies move around in the physics scene and bounce off each other, Box2D will handle all the necessary collision detection and response so we don't need to worry about that. But the whole point of making a physics game is that occasionally something should happen in the game as a result of some of the bodies hitting each other eg. when the player touches a monster he should die, or a ball bouncing on the ground should make a sound etc. We need a way to get this information back from the physics engine.

In the [drawing your own objects](#) topic, we held a reference to the body in our game entity, and queried it every frame to get the current location of the body to draw it. This was a reasonable thing to do because we will be rendering the body every frame, and the location/rotation are likely to be changing every frame too. But collisions are a bit different - they don't occur every frame, they usually happen much less frequently. So asking the physics engine every time step if a body had collided with something would be like the child in the back seat on a car trip continually saying "are we there yet...?". Almost all the time the answer is no, nothing new is accomplished and if you consider that we would have to do this for every entity in the world, it's inefficient too. It would be far better for the driver to say "just keep quiet and I'll let you know when something happens".

In Box2D we do this with a callback function. This is a function that we write to take some action when two entities collide. We don't know yet what the two entities will be, so they must be passed to the function as a parameter when the collision actually happens. Then we give this function to Box2D as if to say "when a collision occurs, call back to this function".

Callback timing

Collisions between entities in the world will be detected during the `b2World::Step()` function that we call every time step. As we saw in the topic on [worlds](#), the `b2World::Step` function advances the simulation, moving all the bodies around and so on. As soon as a collision is detected, the program flow will be given to your callback function to do something, and then goes back to `b2World::Step` to continue with more processing. It's important to note that since your callback is being called right in the middle of the stepping process, you shouldn't do anything to change the scene right away - there may be more collisions occurring in the same time step.



Setting up a collision callback function is done in the same way as customizing the debug draw. We make a subclass of the `b2ContactListener` class, which has a bunch of virtual functions that can be overridden. These are the functions we'll use:

```

1 //b2ContactListener
2 // Called when two fixtures begin to touch
3 virtual void BeginContact(b2Contact* contact);
4
5 // Called when two fixtures cease to touch
6 virtual void EndContact(b2Contact* contact);

```

Notice that we can also detect when a collision finishes, so we'll try using both of these a bit later.

Callback information

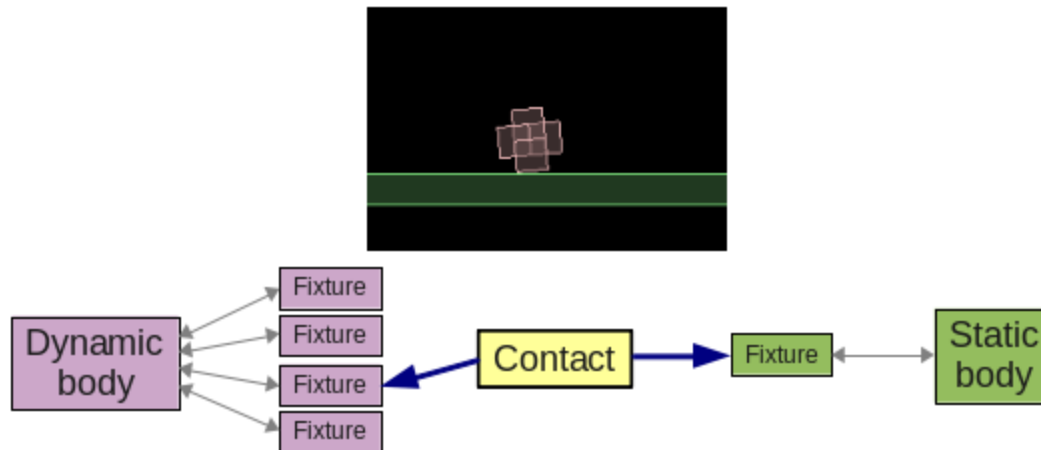
Once we've set up a subclass for `b2ContactListener`, we will know when a pair of entities has collided, and do something with them. But hang on... there are no bodies supplied to `BeginContact/EndContact`, so how can we tell what entities collided? The contact parameter contains all the information we need. The main thing we want to know is what two fixtures were involved in the collision, and we can get these from the contact with these functions:

```

1 //b2Contact
2 // Get the first fixture in this contact
3 b2Fixture* GetFixtureA();
4
5 // Get the second fixture in this contact
6 b2Fixture* GetFixtureB();

```

But still no bodies...? The fixture class has a `GetBody()` function which we can use to get the body of each fixture. It might be nice if the contact also had a function to get the body directly, but it's not really necessary and I actually find it a good reminder that **collisions occur between fixtures, not bodies**. To reinforce that point, and to clarify how things are related, this may help:



In this example the dynamic body has four fixtures, one of which has just begun to contact the single fixture of the static body under it. The contact parameter given to `BeginContact` will return one of these fixtures with `GetFixtureA()`, and the other with `GetFixtureB()`. We don't know which is which, so we'll need to check.

Example

For this topic's demonstration, let's use the `BeginContact/EndContact` collision callbacks to change the color of a ball when it hits something. Start with the scene we had in the [drawing your own objects](#) topic, just after we set the smiley face to draw in the correct position. At this point there was only one ball in the scene.

Add a boolean member variable to the `Ball` class to track whether it is currently hitting something, and functions to call when the physics engine tells us about contacts changing. In the render function, we will set the color depending on the current contact state.

```

1  //Ball class member variable
2  bool m_contacting;
3
4  //in Ball class constructor
5  m_contacting = false;
6
7  //Ball class functions
8  void startContact() { m_contacting = true; }
9  void endContact() { m_contacting = false; }
1
0  //in Ball::render
1  if ( m_contacting )
1  glColor3f(1,0,0);//red
1  else
2  glColor3f(1,1,1);//white
1
3
1

```


4
1
5

We will also need to set the user data of the ball body to be our Ball object in the constructor, as in the [user data](#) topic:

```
1 //in Ball constructor
2 body->SetUserData( this ); //set this Ball object in the body's user data
```

Now, we want Box2D to tell us when the contact state changes. Make a subclass of b2ContactListener and implement the BeginContact/EndContact functions as follows:

```
1 class MyContactListener : public b2ContactListener
2 {
3     void BeginContact(b2Contact* contact) {
4
5         //check if fixture A was a ball
6         void* bodyUserData = contact->GetFixtureA()->GetBody()->GetUserData();
7         if ( bodyUserData )
8             static_cast<Ball*>( bodyUserData )->startContact();
9
10        //check if fixture B was a ball
11        bodyUserData = contact->GetFixtureB()->GetBody()->GetUserData();
12        if ( bodyUserData )
13            static_cast<Ball*>( bodyUserData )->startContact();
14    }
15
16    void EndContact(b2Contact* contact) {
17
18        //check if fixture A was a ball
19        void* bodyUserData = contact->GetFixtureA()->GetBody()->GetUserData();
20        if ( bodyUserData )
21            static_cast<Ball*>( bodyUserData )->endContact();
22
23        //check if fixture B was a ball
24        bodyUserData = contact->GetFixtureB()->GetBody()->GetUserData();
25        if ( bodyUserData )
26            static_cast<Ball*>( bodyUserData )->endContact();
27    }
28 };
29
30
31
32
32
32
32
```

```
3
2
4
2
5
2
6
2
7
2
8
2
9
3
0
```

That seems like a lot of code but if you take a look it's mostly just repeated. First we check if the body of the fixture has any user data. If it does it must be a ball, if it doesn't it must be one of the wall fixtures of the static body. For a simple scene like this it's easy to say make this kind of deduction, but for a real game you would need to have something more sophisticated. Anyway, once we've decided that the fixture belongs to a Ball entity, we cast the user data pointer to a Ball object and call the appropriate state change function. Note that we need to do this for both fixtures A and B because either of them could be a ball.

Now to tell the Box2D world to use these functions for collisions, we use the SetContactListener function. This function takes a pointer to a b2ContactListener object, so we'll need to have an instance of the class to point to. This is easily done by just declaring a variable of your new class at global scope.

```
1 //at global scope
2 MyContactListener myContactListenerInstance;
3
4 //in FooTest constructor
5 m_world->SetContactListener(&myContactListenerInstance);
```

That's it. Now when the ball touches a wall it should appear red, and when it moves away again it should return to white.



This seems to work ok, especially for a simple scene like this one, but **there is a problem with this method**. Grab the ball with the mouse and pull it into a corner so that it is touching two walls at the same time. Now slide it along one wall, out of the corner. You will find that the ball returns to white, even though it is still touching a wall! What's happening here is that when the ball moves

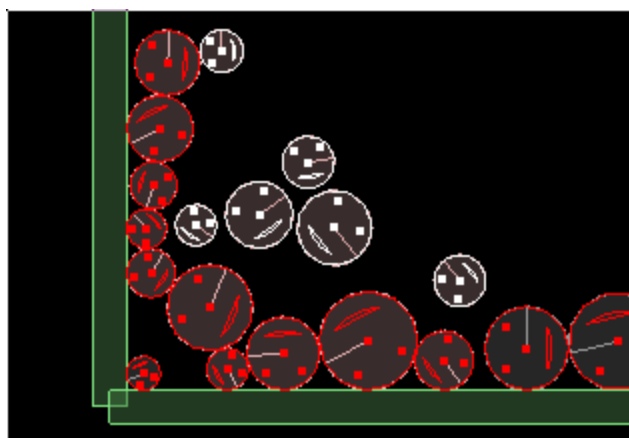
out of the corner, an EndContact occurs for the wall that is no longer touching, so the m_contacting boolean variable of the ball gets set to false. Fortunately that's easy to fix - instead of using a boolean to store a simple on/off state, we just need to store an integer and increment/decrement it to count the number of other fixtures currently being touched.

```

1 //replace old boolean variable m_contacting in Ball class
2 int m_numContacts;
3
4 //in Ball constructor
5 m_numContacts = 0;
6
7 //new implementation for contact state change
8 void startContact() { m_numContacts++; }
9 void endContact() { m_numContacts--; }
10
11 //altered rendering code
12 if ( m_numContacts > 0 )
13     glColor3f(1,0,0); //red
14 else
15     glColor3f(1,1,1); //white
16
17
18
19
20
21
22
23
24
25

```

Now try the corner contact test again, and the problem should be fixed. Adding a whole bunch of balls and bashing them around can be a good way to check if you've got this right. Notice that when things are in a pile bouncing around, the contact state can flicker on and off at a very quick rate. There's nothing wrong with that, but if you are running game logic from these state changes, keep this in mind.



How about one more example to demonstrate how we can use the relationship between the contacting objects. So far all we've done is look at whether a ball touched something, now we will look at what it touched. Let's set up a scene where only one ball is red, and when it hits another ball the red will pass onto the other ball, kind of like playing tag where one ball is 'it'.

Add a boolean variable to the Ball class to track whether it is currently 'it', and set up a function at global scope that takes two balls and switches their 'it' status.

```
1 //in Ball class
2 bool m_imIt;
3
4 //in Ball constructor
5 m_imIt = false;
6
7 //in Ball::render
8 if ( m_imIt )
9     glColor3f(1,0,0);//red
10 else
11     glColor3f(1,1,1);//white
12
13 //in global scope
14 void handleContact( Ball* b1, Ball* b2 ) {
15     bool temp = b1->m_imIt;
16     b1->m_imIt = b2->m_imIt;
17     b2->m_imIt = temp;
18 }
```

This time in the contact listener callback, we only need to implement a BeginContact function:

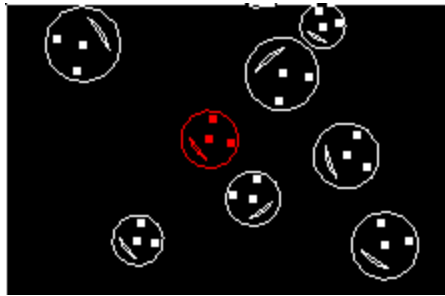
```
1 void BeginContact(b2Contact* contact) {
2     //check if both fixtures were balls
3     void* bodyAUserData = contact->GetFixtureA()->GetBody()->GetUserData();
4     void* bodyBUserData = contact->GetFixtureB()->GetBody()->GetUserData();
5     if ( bodyAUserData && bodyBUserData )
6         handleContact( static_cast<Ball*>( bodyAUserData ),
7                         static_cast<Ball*>( bodyBUserData ) );
8 }
```

Finally, we need to make one of the balls 'it' to begin with:

```
1 //at end of FooTest constructor
2 balls[0]->m_imIt = true;
```

In order to more clearly see whats happening you might want to [turn off gravity](#) so the balls can spread out a bit more, and give the balls a higher [restitution](#) so they keep bouncing for longer

without needing to push them around manually.



Real scenarios

This was a pretty basic demonstration, but the idea can be expanded on to handle more interesting logic in your game, for example when a player touches a monster, etc. In a more complex scene with different kinds of entities colliding, you would want to set something in the fixture user data from which you can tell what kind of entity it represents. One possibility is instead of directly putting the Ball pointer in the user data, you could set an object of a generic parent class, of which Ball and every other entity are subclasses. This parent class would have a virtual function to return what type of entity it is, something like this:

```
1 class Entity
2 {
3     virtual int getEntityType() = 0;
4 };
5
6 class Ball : public Entity
7 {
8     int getEntityType() { return ET_BALL; }
9 }
```

In the collision callback you could then use this to check what kind of entities are colliding and cast them appropriately.

Another way might be to use dynamic casting. To be honest I haven't really come up with a method that I like much, because somewhere along the way they all seem to end up in bloated routines with a lot of if/else checks to get the right case. If anyone has a nice clean way to handle this, let us know!

Update: As Marcos points out in the comments, C++ also has the 'typeid' method for determining what class a pointer is. See also <http://en.wikipedia.org/wiki/Typeid>.

Collision filtering

So far in every scene we have made, all the fixtures were able to collide with all the other fixtures. That is the default behaviour, but it's also possible to set up 'collision filters' to provide finer control over which fixtures can collide with each other. Collision filtering is implemented by setting some flags in the fixture definition when we create the fixture. These flags are:

- categoryBits
- maskBits
- groupIndex

Each of the 'bits' values are a 16 bit integer so you can have up to 16 different categories for collision. There is a little more to it than that though, because it is the *combination* of these values that determines whether two fixtures will collide. The group index can be used to override the category/mask settings for a given set of fixtures.

Category and mask bits

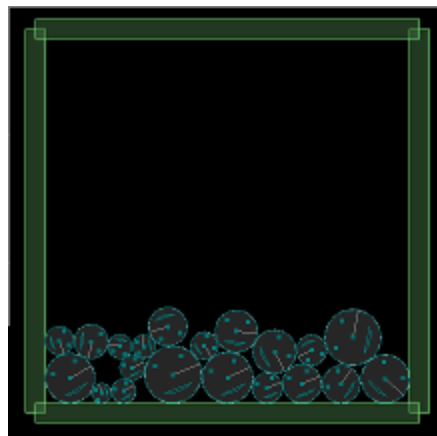
The categoryBits flag can be thought of as the fixture saying 'I am a ...', and the maskBits is like saying 'I will collide with a ...'. The important point is that **these conditions must be satisfied for both fixtures in order for collision to be allowed**.

For example let say you have two categories, cat and mouse. The cats might say 'I am a cat and I will collide with cats and mice', but mice generally not being so interested in colliding with cats, could say 'I am a mouse and I will collide with mice'. With this set of rules, a cat/cat pair will collide, and a mouse/mouse pair will collide, but a cat/mouse pair will not collide (even though the cats were ok with it). Specifically, the check is done by a bitwise and of these two flags, so it might help to see the code:

```
1 bool collide =
2     (filterA.maskBits & filterB.categoryBits) != 0 &&
3     (filterA.categoryBits & filterB.maskBits) != 0;
```

The default values are 0x0001 for categoryBits and 0xFFFF for maskBits, or in other words every fixture says 'I am a thing and I will collide with every other thing', and since all the fixtures have the same rules we found that everything was indeed colliding with everything else.

Let's experiment with changing these flags to see how they can be used. We need a scenario with many entities and we want to see them all bumping around together without flying off the screen, so I'll use the scene from the end of the [drawing your own objects](#) topic where we had a bunch of circular objects inside a 'fence':



By now you should have a pretty good knowledge of how to set up a scene like this so I won't be

covering it here. If you want to use your own scene the important point is to have many bodies that you can set a different size and color for.

In this example we want to set different size and color for each entity, and leave them as that color for the duration of the test. We already have a parameter in the constructor for a radius, so add the necessary code to set a color as well, and use it when rendering. We will also add parameters to set the categoryBits and maskBits in each entity:

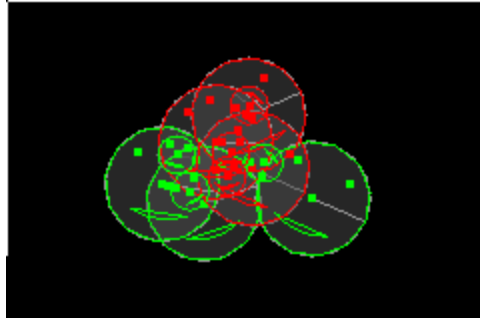
```
1 //Ball class member variable
2 b2Color m_color;
3
4 //edit Ball constructor
5 Ball(b2World* world, float radius, b2Color color, uint16 categoryBits, uint16 maskBits) {
6     m_color = color;
7     ...
8     myFixtureDef.filter.categoryBits = categoryBits;
9     myFixtureDef.filter.maskBits = maskBits;
10
11 //in Ball::render
12 glColor3f(m_color.r, m_color.g, m_color.b);
13
14
15
16
17
18
19
20
21
22
```

Ok now what will we use the size and color for? A good example for this feature is a top-down battlefield scene where you have vehicles of different categories, only some of which should collide with each other, eg. surface vehicles should not collide with aircraft. Let's set up a scenario where we have ships and aircraft, and each entity will also have a friendly or enemy status. We will use size to visually symbolize whether an entity is a ship or a plane, and color to show friendly/enemy status. Let's throw a bunch of entities into the scene, leaving the flags blank for now.

```
1 //in FooTest constructor
2 b2Color red(1,0,0);
3 b2Color green(0,1,0);
4
5 //large and green are friendly ships
6 for (int i = 0; i < 3; i++)
7     balls.push_back( new Ball(m_world, 3, green, 0, 0) );
8 //large and red are enemy ships
9 for (int i = 0; i < 3; i++)
10    balls.push_back( new Ball(m_world, 3, red, 0, 0) );
11 //small and green are friendly aircraft
12 for (int i = 0; i < 3; i++)
13    balls.push_back( new Ball(m_world, 1, green, 0, 0) );
14 //small and red are enemy aircraft
15 for (int i = 0; i < 3; i++)
16    balls.push_back( new Ball(m_world, 1, red, 0, 0) );
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

1
5
1
6

I would also [turn off gravity](#) to make things easier to observe. You should have something like this:



It's immediately obvious that nothing is colliding with anything anymore. This is what you get if the maskBits are zero in a fixture, it will *never* collide. However that's not what we wanted, so let's look at how to get more than just simple 'all or nothing' control. We'll use the following rules to set the allowable collisions between entities:

1. All vehicles collide with the boundary
2. Ships and aircraft do not collide
3. All ships collide with all other ships
4. Aircraft will collide with opposition aircraft, but not with their teammates

That seems like a pretty complex situation, but if we go through it from the point of view of each category, it's not so bad. Firstly let's define some bit flags to use for each category:

```
1 enum _entityCategory {  
2     BOUNDARY = 0x0001,  
3     FRIENDLY_SHIP = 0x0002,  
4     ENEMY_SHIP = 0x0004,  
5     FRIENDLY_AIRCRAFT = 0x0008,  
6     ENEMY_AIRCRAFT = 0x0010,  
7 };
```

Since the default category for a fixture is 1, this arrangement means we don't need to do anything special for the boundary fixture. For the other ones, consider the 'I am a ... and I collide with ...' for each type of vehicle:

Entity	I am a ... (categoryBits)	I collide with ... (maskBits)
Friendly ship	FRIENDLY_SHIP	BOUNDARY FRIENDLY_SHIP ENEMY_SHIP
Enemy ship	ENEMY_SHIP	BOUNDARY FRIENDLY_SHIP ENEMY_SHIP
Friendly	FRIENDLY_AIRCRAFT	BOUNDARY ENEMY_AIRCRAFT

aircraft		
Enemy aircraft	ENEMY_AIRCRAFT	BOUNDARY FRIENDLY_AIRCRAFT

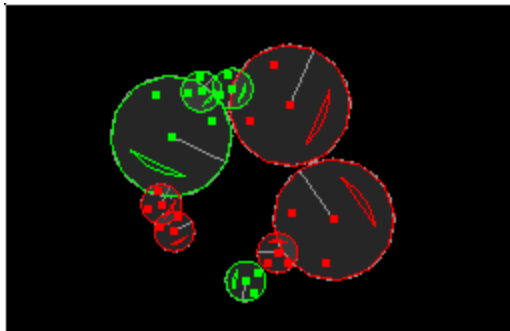
So using the above as a guide we can go back and set the appropriate parameters when creating each entity.

```

1 //large and green are friendly ships
2 for (int i = 0; i < 3; i++)
3     balls.push_back( new Ball(m_world, 3, green, FRIENDLY_SHIP, BOUNDARY | FRIENDLY_SHIP |
4 ENEMY_SHIP ) );
5 //large and red are enemy ships
6 for (int i = 0; i < 3; i++)
7     balls.push_back( new Ball(m_world, 3, red, ENEMY_SHIP, BOUNDARY | FRIENDLY_SHIP |
8 ENEMY_SHIP ) );
9 //small and green are friendly aircraft
10 for (int i = 0; i < 3; i++)
11     balls.push_back( new Ball(m_world, 1, green, FRIENDLY_AIRCRAFT, BOUNDARY |
12 ENEMY_AIRCRAFT ) );
13 //small and red are enemy aircraft
14 for (int i = 0; i < 3; i++)
15     balls.push_back( new Ball(m_world, 1, red, ENEMY_AIRCRAFT, BOUNDARY |
16 FRIENDLY_AIRCRAFT ) );

```

Now running this you should find that the rules specified above are fulfilled.



Using group indexes

The `groupIndex` flag of a fixture can be used to override the category and mask settings above. As the name implies it can be useful to group together fixtures that should either always collide, or never collide. The `groupIndex` is used as a signed integer instead of a bitflag. Here's how it works - read it slowly because it can be a bit confusing at first. When checking two fixtures to see if they should collide:

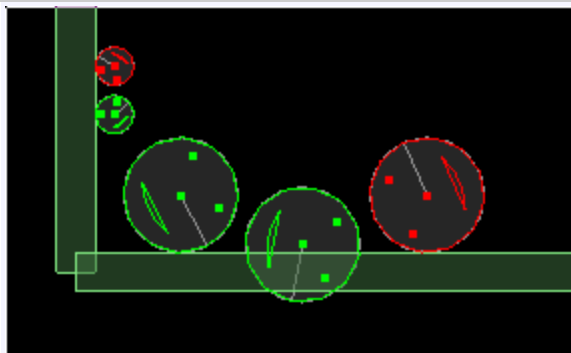
- if either fixture has a `groupIndex` of zero, use the category/mask rules as above
- if both `groupIndex` values are non-zero but different, use the category/mask rules as above
- if both `groupIndex` values are the same and positive, collide
- if both `groupIndex` values are the same and negative, don't collide

The default value for the `groupIndex` is zero, so it has not been playing a part in anything so far. Let's try a simple example where we override the category/mask settings above. We will put the

boundary walls and one vehicle into the same group, and make the group value negative. If you paid attention to the explanation above, you'll see that this will specify that one sneaky vehicle can escape from the boundary fence.

You could change the Ball constructor to add a parameter for a groupIndex, or just do it a quick and hacky way:

```
1 //in FooTest constructor, before creating walls
2 myFixtureDef.filter.groupIndex = -1;//negative, will cause no collision
3
4 //(hacky!) in global scope
5 bool addedGroupIndex = false;
6
7 //(hacky!) in Ball constructor, before creating fixture
8 if ( !addedGroupIndex )
9     myFixtureDef.filter.groupIndex = -1;//negative, same as boundary wall groupIndex
10 addedGroupIndex = true;//only add one vehicle to the special group
11
12
```



If we had set the groupIndex to a positive value, this vehicle would always collide with the wall. In this case all the vehicles already collide with the walls anyway so it wouldn't have made any difference, that's why we set it negative so we could see something happen. In other situations it might be the opposite, for example the ships and aircraft never collide, but you might like to say that a subset of the aircraft can actually collide with ships. Low-flying seaplanes perhaps... :)

Need more control?

If you need even finer control over what should collide with what, you can set a contact filter callback in the world so that when Box2D needs to check if two fixtures should collide, instead of using the above rules it will give you the two fixtures and let you decide. The callback is used in the same way as the debug draw and collision callbacks, by subclassing `b2ContactFilter`, implementing the function below and letting the engine know about this by calling the world's `SetContactFilter` function.

```
1 bool b2ContactFilter::ShouldCollide(b2Fixture* fixtureA, b2Fixture* fixtureB);
```

Changing the collision filter at run-time

Sometimes you might want to alter the collision filter of a fixture depending on events in the game. You can change each of the categoryBits, maskBits, groupIndex by setting a new b2Filter in the fixture. Quite often you only want to change one of these, so it's handy to be able to get the existing filter first, change the field you want, and put it back. If you already have a reference to the fixture you want to change, this is pretty easy:

```
1 //get the existing filter
2 b2Filter filter = fixture->GetFilterData();
3
4 //change whatever you need to, eg.
5 filter.categoryBits = ...;
6 filter.maskBits = ...;
7 filter.groupIndex = ...;
8
9 //and set it back
1 fixture->SetFilterData(filter);
0
```

If you only have a reference to the body, you'll need to [iterate over the body's fixtures](#) to find the one you want.

Sensors

In the previous topic we saw some settings that can be used to prevent two fixtures from colliding. They just pass through each other as if they couldn't see each other at all, even though we can see on the screen that they are overlapping. While this might be the physical behaviour we want, it comes with a drawback: since the fixtures don't collide, they never give us any BeginContact/EndContact information!

What if we want to have two entities not collide, like the ships and the aircraft in the previous example, but we still wanted to get some feedback about when they are overlapping. For example, you might want an enemy plane to drop a bomb when it flies over a friendly ship, or some other kind of game logic. Simply turning off collision between ships and aircraft with collision filtering will not let us know about these events, but there is another method.

A fixture can be made into a 'sensor' by setting the isSensor member of the fixture definition to true when you create it, or by calling SetSensor(bool) on the fixture after it has been created if you need to change it during the simulation. Sensors behave as if their maskBits is set to zero - they never collide with anything. But they do generate BeginContact/EndContact callbacks to let us know when they start or stop overlapping another fixture.

All other features of sensor fixtures remain the same as for a normal fixture. They can be added to any type of body. They still have a mass and will affect the overall mass of the body they are attached to. Remember you can have more than one fixture on a body so you can have a mix of solid shapes and sensors, allowing for all kinds of neat things. Here are a few ideas:

- detect entities entering or leaving a certain area
- switches to trigger an event
- detect ground under player character
- a field of vision for an entity

As far as coding goes, all you really need to do to use this feature is add the `isSensor = true` to your fixtures, but stopping here would make this topic way too short :) Let's make a demonstration in the testbed for the last example above, once again with the top-down battlefield scenario where we use a sensor to represent what an entity can see. Along with sensors, we'll throw in a bit of each of the last few topics - user data, collision filtering and callbacks.

In this demonstration, our scene will have one friendly ship and some enemy aircraft. The ship will be equipped with a circular radar to detect the aircraft when they come within range. We will also have a friendly radar tower in a fixed position, with a similar but rotating radar to detect the enemy aircraft. The collision bits will be set as you might expect: all entities at surface level collide with each other, but the aircraft only collide with the boundary fence.

When one of the friendly entities detects an enemy aircraft, we will change it's color so we can see that the code is working correctly. We will also go one step further and keep track of which enemies each friendly can see, so that we can have efficient access to an up-to-date list of enemies within range for each friendly - this is obviously a useful thing to have for things like AI and other game logic.

My dog has fleas. Ha... just checking if you're still awake after all that text with no code or screenshots. Anyway enough waffle, let's get started.

Example usage

Since this scenario is very much like the last one, we'll use that as a beginning point. Remember we had large circles as ships, and small circles as aircraft. This time we will create just one ship as a friendly, and three aircraft as enemies. Since we have a couple of new entity types, we'll need to add those categories to the flags we use for collision filtering:

```
1 enum _entityCategory {
2     BOUNDARY = 0x0001,
3     FRIENDLY_SHIP = 0x0002,
4     ENEMY_SHIP = 0x0004,
5     FRIENDLY_AIRCRAFT = 0x0008,
6     ENEMY_AIRCRAFT = 0x0010,
7     FRIENDLY_TOWER = 0x0020,
8     RADAR_SENSOR = 0x0040,
9 };
```

Set up the entities in `FooTest` constructor like this...

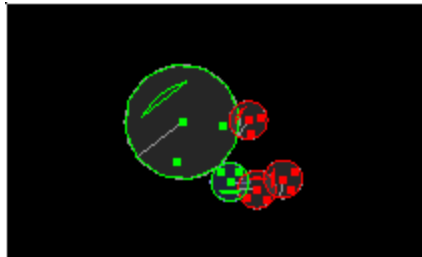
```
1 //one friendly ship
2 Ball* ship = new Ball(m_world, 3, green, FRIENDLY_SHIP, BOUNDARY | FRIENDLY_TOWER );
3 balls.push_back( ship );
4
```

```

5 //three enemy aircraft
6 for (int i = 0; i < 3; i++)
7     balls.push_back( new Ball(m_world, 1, red, ENEMY_AIRCRAFT, BOUNDARY | RADAR_SENSOR )
8 );
9
1 //a tower entity
0 Ball* tower = new Ball(m_world, 1, green, FRIENDLY_TOWER, FRIENDLY_SHIP );
1 tower->m_body->SetType(b2_kinematicBody);
1 balls.push_back( tower );
1
2

```

The first parts you already understand, the last part is almost the same except we don't want the tower to be a dynamic body. Rather than add a parameter to the Ball class constructor it's simpler just to alter it like this for a one-off case. Why couldn't we make the tower a static body? Well we could, but then it wouldn't be able to rotate - remember we want this to be a rotating radar. You should be seeing something like this:

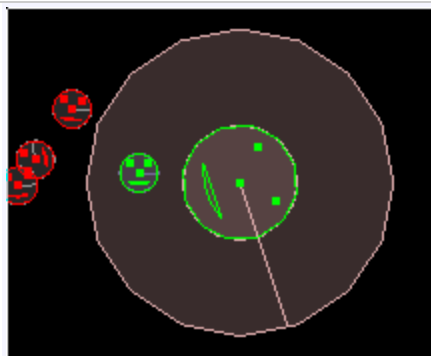


Now let's make some sensors. After the code above, we can add a sensor to the ship like this:

```

1 //add radar sensor to ship
2 b2CircleShape circleShape;
3 circleShape.m_radius = 8;
4 myFixtureDef.shape = &circleShape;
5 myFixtureDef.isSensor = true;
6 myFixtureDef.filter.categoryBits = RADAR_SENSOR;
7 myFixtureDef.filter.maskBits = ENEMY_AIRCRAFT; //radar only collides with aircraft
8 ship->m_body->CreateFixture(&myFixtureDef);

```

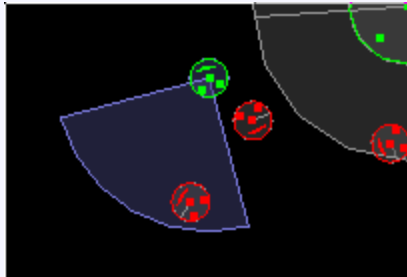


The tower will get a semi-circle sensor, and the angular velocity will be non-zero:

```

1 //add semicircle radar sensor to tower
2 float radius = 8;
3 b2Vec2 vertices[8];
4 vertices[0].Set(0,0);
5 for (int i = 0; i < 7; i++) {
6     float angle = i / 6.0 * 90 * DEGTORAD;
7     vertices[i+1].Set( radius * cosf(angle), radius * sinf(angle) );
8 }
9 polygonShape.Set(vertices, 8);
10 myFixtureDef.shape = &polygonShape;
11 tower->m_body->CreateFixture(&myFixtureDef);
12
13 //make the tower rotate at 45 degrees per second
14 tower->m_body->SetAngularVelocity(45 * DEGTORAD);
15
16
17
18
19
20

```



Now all that's left to do is the collision callback implementation. Remember we want to store a list of the enemy aircraft that each friendly entity can currently see. For that obviously we will need to have a list to store these in, and it would also be good to have functions to make changes to the list when Box2D lets us know something has changed.

```

1 //Ball class member
2 std::vector<Ball*> visibleEnemies;
3
4 //in Ball class
5 void radarAcquiredEnemy(Ball* enemy) {
6     visibleEnemies.push_back( enemy );
7 }
8 void radarLostEnemy(Ball* enemy) {
9     visibleEnemies.erase( std::find(visibleEnemies.begin(), visibleEnemies.end(), enemy ) );
10 }
11
12 //in Ball::render
13 if ( visibleEnemies.size() > 0 )
14     glColor3f(1,1,0); //yellow
15 else
16
17

```

```

3   glColor3f(m_color.r, m_color.g, m_color.b);
1
4
1
5
1
6

```

I'm really starting to think the name 'Ball' for this class is not appropriate anymore... maybe we should have called it Entity to begin with :)

Now set up a [collision callback](#) to tell the friendly entities when their radar sensor begins or ends contact with an enemy aircraft. Since much of the code here tends to get repeated, I have put the repetitive part in a sub function so that the main logic in the callback itself is easier to see.

```

1  //helper function to figure out if the collision was between
2  //a radar and an aircraft, and sort out which is which
3  bool getRadarAndAircraft(b2Contact* contact, Ball*& radarEntity, Ball*& aircraftEntity)
4  {
5      b2Fixture* fixtureA = contact->GetFixtureA();
6      b2Fixture* fixtureB = contact->GetFixtureB();
7
8      //make sure only one of the fixtures was a sensor
9      bool sensorA = fixtureA->IsSensor();
1     bool sensorB = fixtureB->IsSensor();
0     if ( ! (sensorA ^ sensorB) )
1         return false;
1
1     Ball* entityA = static_cast<Ball*>( fixtureA->GetBody()->GetUserData() );
2     Ball* entityB = static_cast<Ball*>( fixtureB->GetBody()->GetUserData() );
1
3     if ( sensorA ) { //fixtureB must be an enemy aircraft
1         radarEntity = entityA;
4         aircraftEntity = entityB;
1     }
5     else { //fixtureA must be an enemy aircraft
1         radarEntity = entityB;
6         aircraftEntity = entityA;
1     }
7     return true;
1 }
8
1 //main collision call back function
9 class MyContactListener : public b2ContactListener
2 {
0     void BeginContact(b2Contact* contact) {
2         Ball* radarEntity;
1         Ball* aircraftEntity;
2         if ( getRadarAndAircraft(contact, radarEntity, aircraftEntity) )
2             radarEntity->radarAcquiredEnemy( aircraftEntity );

```

```

2   }
3
2   void EndContact(b2Contact* contact) {
4       Ball* radarEntity;
2       Ball* aircraftEntity;
5       if ( getRadarAndAircraft(contact, radarEntity, aircraftEntity) )
2           radarEntity->radarLostEnemy( aircraftEntity );
6   }
2   };

```

Almost there... to complete the system we need to set the user data in the physics body to point to our entity object, and let the physics engine know about our contact listener:

```

1 //in Ball constructor

```

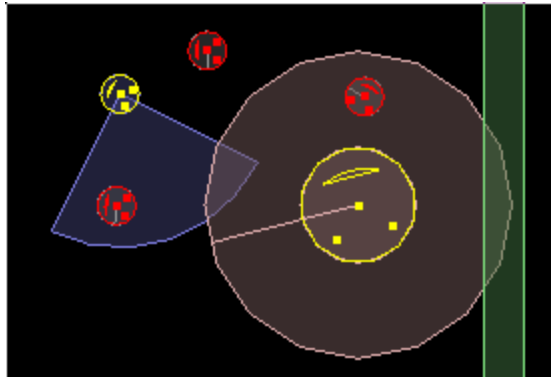


```

2 m_body->SetUserData( this );
3
4 //at global scope
5 MyContactListener myContactListenerInstance;
6
7 //in FooTest constructor
8 m_world->SetContactListener(&myContactListenerInstance);

```

Now if all goes well, you should see the ship and the ship and the tower turn yellow whenever any aircraft are touching their radar sensor. Neat!

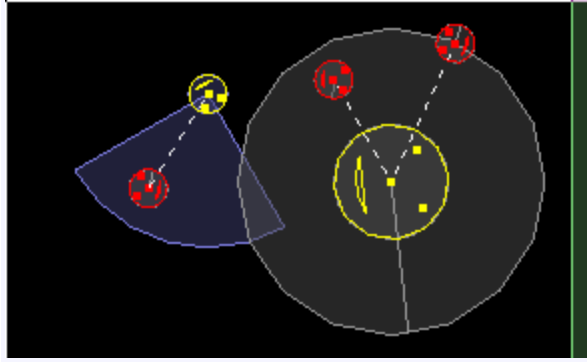


As one last little exercise, remember how we wanted to have an up-to-date list of all visible enemies for each friendly entity that we could call efficiently? Well now we have one, so let's take it for a spin. We can draw a line between each radar and the entities in view.

```

1 //in Ball::renderAtBodyPosition
2 b2Vec2 pos = m_body->GetPosition(); //(existing code)
3 glColor3f(1,1,1); //white
4 glLineStipple( 1, 0xF0F0 ); //evenly dashed line
5 glEnable(GL_LINE_STIPPLE);
6 glBegin(GL_LINES);
7 for (int i = 0; i < visibleEnemies.size(); i++) {
8     b2Vec2 enemyPosition = visibleEnemies[i]->m_body->GetPosition();
9     glVertex2f(pos.x, pos.y);
10    glVertex2f(enemyPosition.x, enemyPosition.y);
11 }
12 glEnd();
13 glDisable(GL_LINE_STIPPLE);
14
15
16
17
18
19
20
21
22
23

```



Ray casting

Ray casting is often used to find out what objects are in a certain part of the world. A ray is just a straight line, and we can use a function provided by Box2D to check if the line crosses a **fixture**. We can also find out what the normal is at the point the line hits the fixture.

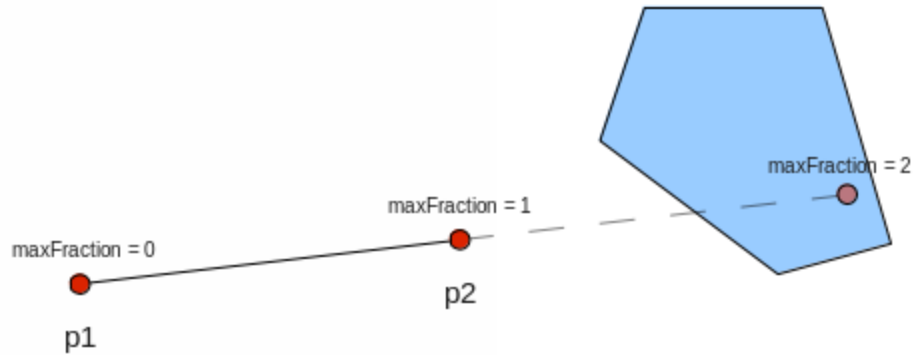
Here is the function I'm talking about, which returns true if the ray hits the fixture. Notice it's a member of the `b2Fixture` class, which means we'll first need to have one of those to cast a ray against.

```
1 bool b2Fixture::RayCast(b2RayCastOutput* output, const b2RayCastInput& input);
```

Now what are these input and output parameters. Well, straight from the source code here is what a `b2RayCastInput` contains:

```
1 // Ray-cast input data. The ray extends from p1 to p1 + maxFraction * (p2 - p1).
2 struct b2RayCastInput
3 {
4     b2Vec2 p1, p2;
5     float32 maxFraction;
6 };
```

The points `p1` and `p2` are used to define a direction for the ray, and the `maxFraction` specifies how far along the ray should be checked for an intersection. The following image may make this clearer. A `maxFraction` of 1 simply means the segment from `p1` to `p2`, which in this case would not intersect the shape, but a `maxFraction` of 2 would.



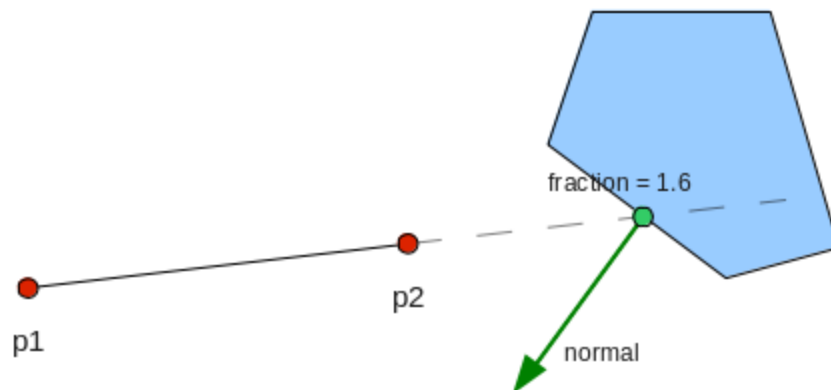
And here is what a `b2RayCastOutput` contains:

```

1 // Ray-cast output data. The ray hits at p1 + fraction * (p2 - p1), where p1 and p2
2 // come from b2RayCastInput.
3 struct b2RayCastOutput
4 {
5     b2Vec2 normal;
6     float32 fraction;
7 };

```

If the ray does intersect the shape, `b2Fixture::RayCast` will return true and we can look in the output struct to find the actual fraction of the intersect point, and the normal of the fixture 'surface' at that point:



Example

To try out this very handy function, let's set up a scene with a fenced area and some shapes floating inside in a zero-gravity environment. By now you should be getting used to this one, so we'll make the walls as edges instead of boxes just to spice things up.

```

1 FooTest() {
2
3     //a static body
4     b2BodyDef myBodyDef;

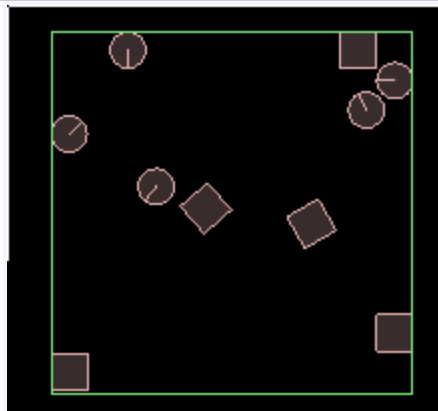
```

```

5  myBodyDef.type = b2_staticBody;
6  myBodyDef.position.Set(0, 0);
7  b2Body* staticBody = m_world->CreateBody(&myBodyDef);
8
9  //shape definition
1 b2PolygonShape polygonShape;
0
1  //fixture definition
1  b2FixtureDef myFixtureDef;
1  myFixtureDef.shape = &polygonShape;
2
1  //add four walls to the static body
3  b2Vec2 bl(-20, 0);
1  b2Vec2 br( 20, 0);
4  b2Vec2 tl(-20,40);
1  b2Vec2 tr( 20,40);
5  polygonShape.SetAsEdge( bl, br ); //ground
1  staticBody->CreateFixture(&myFixtureDef);
6  polygonShape.SetAsEdge( tl, tr );//ceiling
1  staticBody->CreateFixture(&myFixtureDef);
7  polygonShape.SetAsEdge( bl, tl );//left wall
1  staticBody->CreateFixture(&myFixtureDef);
8  polygonShape.SetAsEdge( br, tr );//right wall
1  staticBody->CreateFixture(&myFixtureDef);
9
2  myBodyDef.type = b2_dynamicBody;
0  myBodyDef.position.Set(0,20);
2  polygonShape.SetAsBox(2,2);
1  myFixtureDef.density = 1;
2  for (int i = 0; i < 5; i++)
2      m_world->CreateBody(&myBodyDef)->CreateFixture(&myFixtureDef);
2
3  //circles
2  b2CircleShape circleShape;
4  circleShape.m_radius = 2;
2  myFixtureDef.shape = &circleShape;
5  for (int i = 0; i < 5; i++)
2      m_world->CreateBody(&myBodyDef)->CreateFixture(&myFixtureDef);
6
2  //turn gravity off
7  m_world->SetGravity( b2Vec2(0,0) );
2
8  }
2
9
3
0
3
1
3

```

2
3
3
3
3
4
3
5
3
6
3
7
3
8
3
9
4
0
4
1
4
2
4
3
4
4
4
5
4
6
4
7

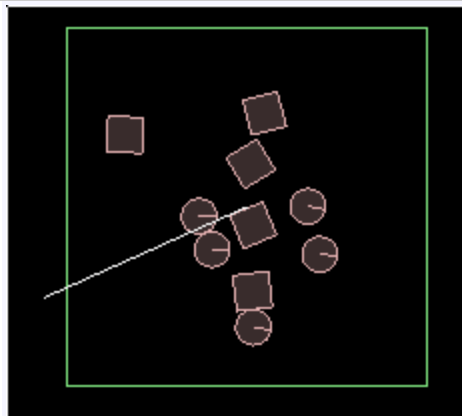


Now we need a ray to cast against these shapes. Let's make a ray starting from the center of the screen and going outward, and rotating slowly around. The only state we need to keep for this is the current angle, so instead of making a special class for it, we'll just keep a variable at global scope.

```

1 //at global scope
2 float currentRayAngle = 0;
3
4 //in Step() function
5 currentRayAngle += 360 / 20.0 / 60.0 * DEGTORAD; //one revolution every 20 seconds
6
7 //calculate points of ray
8 float rayLength = 25; //long enough to hit the walls
9 b2Vec2 p1( 0, 20 ); //center of scene
10 b2Vec2 p2 = p1 + rayLength * b2Vec2( sinf(currentRayAngle), cosf(currentRayAngle) );
11
12 //draw a line
13 glColor3f(1,1,1); //white
14 glBegin(GL_LINES);
15 glVertex2f( p1.x, p1.y );
16 glVertex2f( p2.x, p2.y );
17 glEnd();
18
19
20
21
22
23
24
25
26
27

```



You should now see a white line circling the scene. Now we just need to use the RayCast function to get the distance to the closest intersected shape, and draw the line at that length. We will check every fixture of every shape, which is not the best way to do this, but will do as an example (see the [world querying](#) topic for a more efficient method). It also means we can take a look at how you can iterate over the contents of the world:

```

1 //in Step() function, continuing on from section above
2
3 //set up input
4 b2RayCastInput input;
5 input.p1 = p1;

```

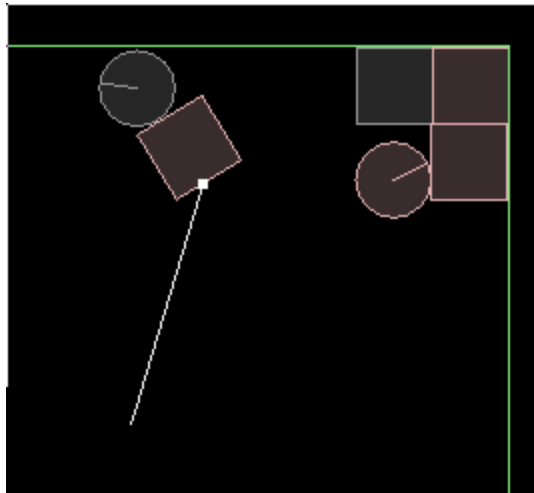
```

6   input.p2 = p2;
7   input.maxFraction = 1;
8
9   //check every fixture of every body to find closest
1  float closestFraction = 1; //start with end of line as p2
0  b2Vec2 intersectionNormal(0,0);
1  for (b2Body* b = m_world->GetBodyList(); b; b = b->GetNext()) {
1      for (b2Fixture* f = b->GetFixtureList(); f; f = f->GetNext()) {
1
2          b2RayCastOutput output;
1          if ( ! f->RayCast( &output, input ) )
3              continue;
1          if ( output.fraction < closestFraction ) {
4              closestFraction = output.fraction;
1              intersectionNormal = output.normal;
5          }
1      }
6  }
1
7  b2Vec2 intersectionPoint = p1 + closestFraction * (p2 - p1);
1
8  //draw a line
1  glColor3f(1,1,1); //white
9  glBegin(GL_LINES);
2  glVertex2f( p1.x, p1.y );
0  glVertex2f( intersectionPoint.x, intersectionPoint.y );
2  glEnd();
1
2  //draw a point at the intersection point
2  glPointSize(5);
2  glBegin(GL_POINTS);
3  glVertex2f( intersectionPoint.x, intersectionPoint.y );
2  glEnd();
4
2
5
2
6
2
7
2
8
2
9
3
0
3
1
3
2

```

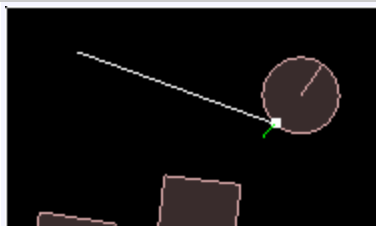
3
3
3
4
3
5
3
6
3
7
3
8

You might notice that now we are drawing two lines on top of each other... for clarity, delete the first one and you should get this result:

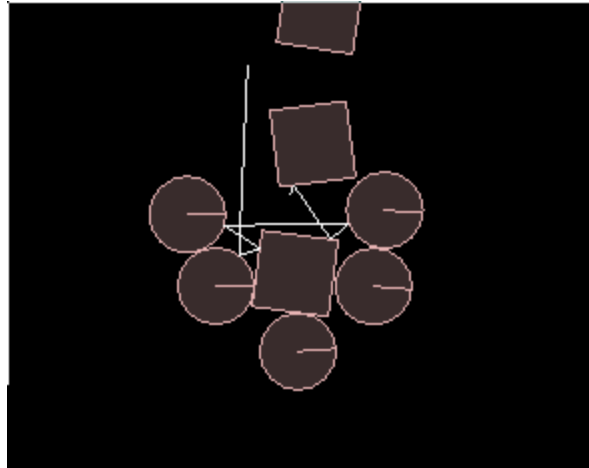


Well that's about all there is to finding the intersection point. We can use the normal in the output struct for some interesting stuff though, so let's try it out. First, we'll simply render the normal to see what it looks like:

```
1 //draw intersection normal
2 b2Vec2 normalEnd = intersectionPoint + intersectionNormal;
3 glColor3f(0,1,0); //green
4 glBegin(GL_LINES);
5 glVertex2f( intersectionPoint.x, intersectionPoint.y );
6 glVertex2f( normalEnd.x, normalEnd.y );
7 glEnd();
```



And for a grand finale, we can put the raycasting code into its own function and call it recursively to reflect the ray off the intersected fixtures until it runs out. This is not really anything to do with Box2D, I just think it's neat :)



Here is the code:

```

1 //new function for FooTest class
2 void drawReflectedRay( b2Vec2 p1, b2Vec2 p2 )
3 {
4     //set up input
5     b2RayCastInput input;
6     input.p1 = p1;
7     input.p2 = p2;
8     input.maxFraction = 1;
9
10    //check every fixture of every body to find closest
11    float closestFraction = 1; //start with end of line as p2
12    b2Vec2 intersectionNormal(0,0);
13    for (b2Body* b = m_world->GetBodyList(); b; b = b->GetNext()) {
14        for (b2Fixture* f = b->GetFixtureList(); f; f = f->GetNext()) {
15
16            b2RayCastOutput output;
17            if ( ! f->RayCast( &output, input ) )
18                continue;
19            if ( output.fraction < closestFraction ) {
20                closestFraction = output.fraction;
21                intersectionNormal = output.normal;
22            }
23        }
24    }
25
26    b2Vec2 intersectionPoint = p1 + closestFraction * (p2 - p1);
27
28    //draw this part of the ray
29    glBegin(GL_LINES);
30    glVertex2f( p1.x, p1.y );
31    glVertex2f( intersectionPoint.x, intersectionPoint.y );
32    glEnd();

```

```

2
1  if ( closestFraction == 1 )
2      return; //ray hit nothing so we can finish here
2  if ( closestFraction == 0 )
2      return;
3
2  //still some ray left to reflect
4  b2Vec2 remainingRay = (p2 - intersectionPoint);
2  b2Vec2 projectedOntoNormal = b2Dot(remainingRay, intersectionNormal) *
5intersectionNormal;
2  b2Vec2 nextp2 = p2 - 2 * projectedOntoNormal;
6
2  //recurse
7  drawReflectedRay(intersectionPoint, nextp2);
2  }
8
2
9
3
0
3
1
3
2
3
3
3
4
3
5
3
6
3
7
3
8
3
9
4
0
4
1
4
2
4
3
4
4
4
5

```

... and then inside Step() you would just have:

```
1 //calculate points of ray
2 float rayLength = 25;
3 b2Vec2 p1( 0, 20 ); //center of scene
4 b2Vec2 p2 = p1 + rayLength * b2Vec2( sinf(currentRayAngle), cosf(currentRayAngle) );
5
6 glColor3f(1,1,1); //white
7 drawReflectedRay(p1, p2);
```

World querying

Often you will want to know what entities are in a given part of the scene. For example if a bomb goes off, everything in the vicinity should take some damage, or in an RTS type game you might want to let the user select units by dragging a rectangle around them. The method of quickly checking what objects are in that part of the world to use for further detailed processing is known as 'world querying'.

Box2D provides two tools for this - ray casting and AABB testing. Ray casting... didn't we just do that? Yes, we did it the manual way, by looping through every fixture in the world and checking the ray against them all to find out which one was the closest. This can be very inefficient when you have a large number of fixtures in the scene. A better way is to use the RayCast function of the world itself. This allows the engine to focus on fixtures which it knows are near the ray's path.

Ray casting, the efficient way

Since we have already looked at ray casting in some detail, we'll just take a quick look at the world's RayCast function without making a demonstration. The function looks like this:

```
1 void RayCast(b2RayCastCallback* callback, const b2Vec2& point1, const b2Vec2& point2);
```

The last two parameters are pretty simple, they are the beginning and end point of the ray, and we use them as we would for a ray cast against a single fixture. The 'callback' parameter is new, but by now we are familiar with how callbacks in Box2D usually work, so it's no surprise that we would implement this by making a subclass of b2RayCastCallback, and passing an instance of it to this function as the callback. The b2RayCastCallback class has only one function to override, and it is:

```
1 //in b2RayCastCallback class
2 float32 ReportFixture(b2Fixture* fixture, const b2Vec2& point, const b2Vec2& normal,
float32 fraction);
```

During the ray cast calculation, each time a fixture is found that the ray hits, this callback function will be called. For each intersection we can get the fixture which was hit, the point at which it was

hit, the normal to the fixture 'surface' and the fraction of the distance from point1 to point2 that the intersection point is at. The diagrams in the [ray casting](#) topic may help to explain the fraction parameter.

The final point to cover is the return value that you should give for this callback function. Remember that if your ray is long enough there could be many fixtures that it intersects with, and your callback could be called many times during one RayCast. Very importantly, **this raycast does not detect fixtures in order of nearest to furthest**, it just gives them to you in any old order - this helps it to be efficient for very large scenes. The engine lets you decide how you want to deal with each fixture as it is encountered. This is where the return value of the callback comes in. You will return a floating point value, which you can think of as adjusting the length of the ray while the raycast is in progress. This can be a little confusing so take it slow...

- return -1 to completely ignore the current intersection
- return a value from 0 - 1 to adjust the length of the ray, for example:
 - returning 0 says there is now no ray at all
 - returning 1 says that the ray length does not change
 - returning the fraction value makes the ray just long enough to hit the intersected fixture

Here the fraction value refers to the 'fraction' parameter that is passed to the callback. If I can be bothered I might make a diagram for this later, but if the above description is not clear just remember the common cases:

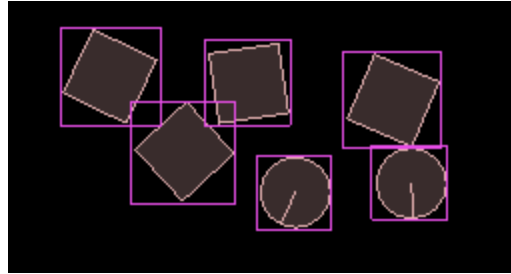
- To find only the closest intersection:
 - - return the fraction value from the callback
 - - use the most recent intersection as the result
- To find all intersections along the ray:
 - - return 1 from the callback
 - - store the intersections in a list
- To simply find if the ray hits anything:
 - - if you get a callback, something was hit (but it may not be the closest)
 - - return 0 from the callback for efficiency
-

Area querying (aka AABB querying)

The Box2D world has another function for finding fixtures overlapping a given area, the AABB query. This one allows us to define a rectangular region, and the engine will then find all fixtures in that region and call a callback function for each of them. Typically the callback is used to fill a list of fixtures in preparation for some other processing to follow, but this method allows you to implement other kinds of processing efficiently too. For example, if there are a large number of fixtures in the given area but all you want to do is find out if their total mass is more than a certain amount, you could add their masses as the query progressed, and as soon as you find the necessary mass, the query can finish.

The function for this is called QueryAABB and it gets the name because the area must be specified as an 'axis-aligned bounding box'. This just means that the area is rectangular and can't be rotated at an odd angle. It also means that instead of testing whether fixtures are within the

area, **it tests if the AABBs of fixtures are within the area**. You can turn on rendering of the AABBs for fixtures in the testbed with one of the checkboxes on the right. Here is an example of the AABBs (purple) of some fixtures.



The function looks like this:

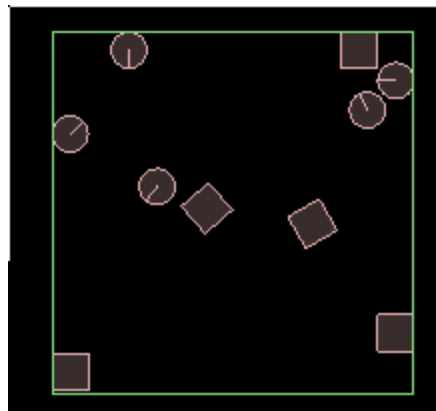
```
1 void QueryAABB(b2QueryCallback* callback, const b2AABB& aabb);
```

Do you like these callback functions? I hope so because here is another one. As usual, you need to make a subclass of the b2QueryCallback class and implement a function. The single function in this class is:

```
1 bool ReportFixture(b2Fixture* fixture);
```

Awesome, only one parameter. Since the AABB query just finds out whether a fixture is inside the area or not, there is no need for any trickiness like the ray cast query above. When you call QueryAABB, your ReportFixture callback will just be called for each fixture in the area, passing that fixture as the parameter. Let's try an example of using this feature.

We will set up a test where we define a rectangular area, and then draw marker on each fixture which is currently overlapping the area. We'll use the same scene as at the beginning of the last topic, so go copy the code from there.



First let's get the area displaying on the screen. We can override the MouseUp and MouseDown functions of the Test class to store the corners of the rectangle as member variables of our FooTest class. Then in the Step() function, draw a rectangle using those points:

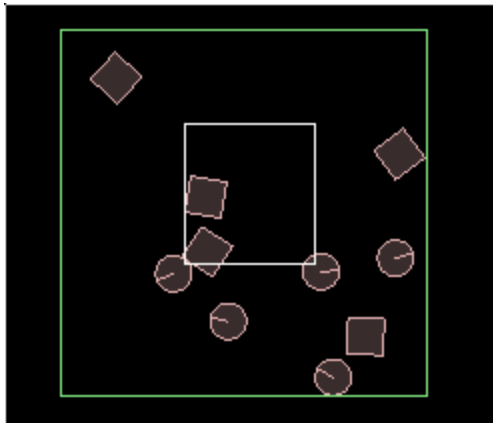
```
1 //FooTest class member variable
2 b2Vec2 mouseDownPos, mouseUpPos;
3
4 //FooTest class functions
```

```

5 void MouseDown(const b2Vec2& p) { mouseDownPos = mouseUpPos = p;
6 Test::MouseDown(p); }
7 void MouseUp(const b2Vec2& p) { mouseUpPos = p; Test::MouseUp(p); }
8
9 //in Step() function, draw the rectangle
1 b2Vec2 lower( min(mouseDownPos.x,mouseUpPos.x), min(mouseDownPos.y,mouseUpPos.y)
0);
1 b2Vec2 upper( max(mouseDownPos.x,mouseUpPos.x), max(mouseDownPos.y,mouseUpPos.y)
1);
1
2 glColor3f(1,1,1);//white
1 glBegin(GL_LINE_LOOP);
3 glVertex2f(lower.x, lower.y);
1 glVertex2f(upper.x, lower.y);
4 glVertex2f(upper.x, upper.y);
1 glVertex2f(lower.x, upper.y);
5 glEnd();
1
6
1
7
1
8

```

Unfortunately this version (v2.1.2) of Box2D's testbed doesn't allow us to override the MouseMove function, so we can't actually see the box as we drag the mouse to draw it.



With a query area in place, we now get on with the actual AABB query itself. First we'll need to make a subclass of the b2QueryCallback class to hold the callback function, and a list to hold the fixtures that were found inside the query region. You could put the list in global or class scope, but as world query results tend to be very temporal in nature, it can be more intuitive to keep the results list in the callback class itself and use a new instance of it each time.

```

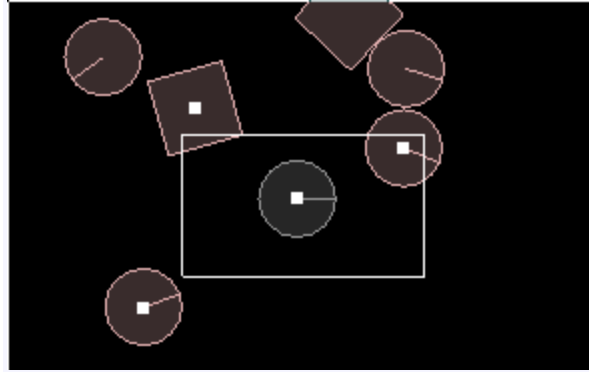
1 //subclass b2QueryCallback
2 class MyQueryCallback : public b2QueryCallback {
3 public:
4     std::vector<b2Body*> foundBodies;
5

```

```

6  bool ReportFixture(b2Fixture* fixture) {
7      foundBodies.push_back( fixture->GetBody() );
8      return true; //keep going to find all fixtures in the query area
9  }
1  };
0
1  //in Step() function
1  MyQueryCallback queryCallback;
1  b2AABB aabb;
2  aabb.lowerBound = lower;
1  aabb.upperBound = upper;
3  m_world->QueryAABB( &queryCallback, aabb );
1
4
1  //draw a point on each body in the query area
5  glPointSize(6);
1  glBegin(GL_POINTS);
6  for (int i = 0; i < queryCallback.foundBodies.size(); i++) {
1      b2Vec2 pos = queryCallback.foundBodies[i]->GetPosition();
7      glVertex2f( pos.x, pos.y );
1  }
8  glEnd();
1
9
2
0
2
1
2
2
2
2
3
2
4
2
5
2
6
2
7

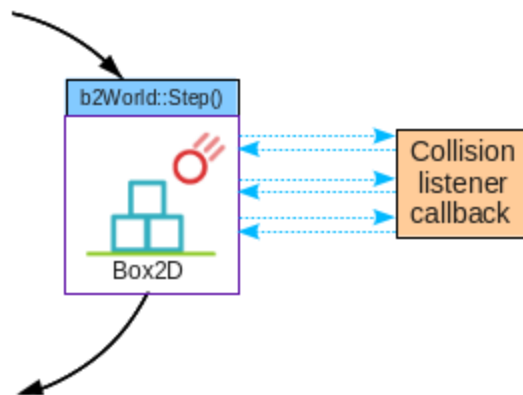
```



Note that we need to return true from the callback function to make sure the query keeps going until all of the overlapping fixtures have been put into the list. You should see a white spot on each fixture which has an AABB overlapping the region. In the screenshot above, you can see in the bottom left an example of how the AABB can overlap the region while the fixture itself does not.

Removing bodies safely

Unless you are making a very basic game, you will probably want to remove some bodies from the scene at some point. This could be for example if the player kills an enemy, the bullets they fire need to be cleaned up, or two objects collide and one breaks, etc. The actual code to remove a body is dead simple - you just say `world->DestroyBody(b2Body*)`, and if you are removing the body outside the timestep that's all there is to it. The problem comes when you want to remove a body as a result of something that happened within a timestep, usually a collision callback. If we remind ourselves of the timing of collision callbacks, we see that we cannot remove a body inside a collision callback, because the world is right in the middle of performing a physics step, and removing the data it is working with is a bad idea.

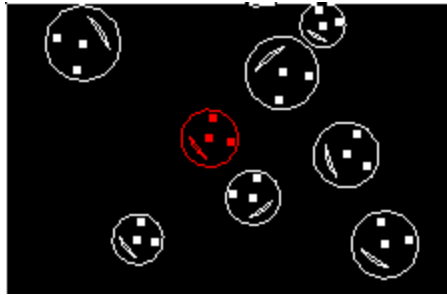


A very similar situation is encountered if you use threads or timers to remove bodies, for example if you want a body to be removed a few seconds after some event in the game. If the timer fires while the world is stepping, this can also cause problems.

The solution to this is quite simple. When you get a collision callback which means a body should be removed, just add the body to a list of bodies scheduled for removal, and remove them all together after the time step has finished.

Example

As an example, let's set up a demonstration where bodies will be removed from the scene at runtime depending on the collisions between them. We will bring back the friendly smiley faces from the [collision callbacks](#) topic to help us with this. In fact, we can pick up right where we left off at the end of that topic. To recap, we had a situation where one ball was red, and when it bounced against another ball the red state would swap between them as if they were playing tag and the red ball was 'it':



This time, let's imagine that the red ball has caught a deadly virus and is about to die, but not until it has infected one other ball first. As soon as the virus has been passed to another ball by a collision, the first ball with the virus will be removed from the world.

There are only a few small changes we need to make to accomplish this. We'll need a list to store the bodies to remove from the world after the timestep, and the collision callback should be changed to put the already infected ball into that list. To actually delete a ball properly we should add a destructor to the Ball class and inside it, we will remove the physics body from the Box2D world.

```
1 //at global scope
2 std::set<Ball*> ballsScheduledForRemoval;
3
4 //revised handleContact function
5 void handleContact( Ball* b1, Ball* b2 ) {
6     if ( ! outbreak )
7         return;
8     if ( b1->m_imIt ) {
9         ballsScheduledForRemoval.push_back(b1);
10        b2->m_imIt = true;
11    }
12    else if ( b2->m_imIt ) {
13        ballsScheduledForRemoval.push_back(b2);
14        b1->m_imIt = true;
15    }
16 }
17
18 //implement Ball class destructor
19 Ball::~~Ball()
20 {
21     m_body->GetWorld()->DestroyBody( m_body );
22 }
```

6
1
7
1
8
1
9
2
0
2
1
2
2

The 'outbreak' variable is optional, I just added this as a global variable initially set to false so that I can use the Keyboard function to start the infection spreading when I choose - otherwise the balls are too bunched up in the beginning and most of them get infected and die in the blink of an eye which is no fun. You could also just space them out more to start with I guess, it's up to you.

Now in the Step() function, after calling Test::Step() and before rendering the scene, we'll need to process the entities that got scheduled for removal.

```
1 //process list for deletion
2 std::set<Ball*>::iterator it = ballsScheduledForRemoval.begin();
3 std::set<Ball*>::iterator end = ballsScheduledForRemoval.end();
4 for (; it!=end; ++it) {
5     Ball* dyingBall = *it;
6
7     //delete ball... physics body is destroyed here
8     delete dyingBall;
9
10    //... and remove it from main list of balls
11    std::vector<Ball*>::iterator it = std::find(balls.begin(), balls.end(), dyingBall);
12    if ( it != balls.end() )
13        balls.erase( it );
14 }
15
16 //clear this list for next time
17 ballsScheduledForRemoval.clear();
```

Now you should see just one red ball in the scene, infecting another ball and dying when it collides

with it.

Dealing with the removal bodies as a result of a thread or timing routine is very similar, just gather all the bodies to be deleted in a list, and deal with them when you can be sure that the world is not stepping. One simple method for implementing a 'timed removal', that is for example if you want to wait a certain time before deleting something, is just to calculate how many time steps that would be for your game loop (eg. 2 seconds is 120 time steps at 60fps), set that value in the entity and then decrement it every frame until it gets to zero, then remove it.

By the way, although this topic has focused on removing bodies since that is the most common case to cause problems, the creation of new bodies, and the creation and removal of fixtures must be dealt with in the same way.

Is my character on the ground?

If ever there was a question about Box2D, "how can I tell if my player character is on the ground?" is probably it. I think the reason this comes up so often is firstly because of the callback-style method of getting contact information - the callback method itself can throw off those who are not familiar with it - and also because keeping track of what is touching what is left to the programmer to implement instead of being built-in to the library. In either case, we have covered both of these topics already. The [collision callbacks](#) topic dealt with getting the necessary begin and end contact events, and the [sensors](#) topic showed an example where we kept track of a 'currently touching' list.

So what's left to talk about? Well, I'm guessing many people will find this page as their first stop when trying to tackle this problem, so I wanted to point out those previous two topics as better places to start for a proper understanding. And I also thought it would be good to make a demonstration of these techniques in the setting that they are most commonly wanted, the platformer game, rather than the battlefield scenario of the previous topics.

We'll look at an example which is quite effective at solving this question, yet not too complex. The player body will have a main fixture to represent the player itself, and another smaller sensor fixture attached underneath it to detect when something is under-foot. This foot sensor will keep track of what other fixtures in the world it is touching, and because we know it is directly beneath the player and not too big, we will just consider any situation where the foot sensor is touching something to be a jumpable situation. There are a couple of small tweaks we can make but that's the general idea.

Because these are concepts we've already covered just put in a different setting, we'll go one step further and instead of simply keeping track of whether the player is standing on something, we'll also keep track of what kind of something(s) are being stood on. The scene will have two types of boxes of different size, and depending on which the player is standing on we can alter his jumpability.

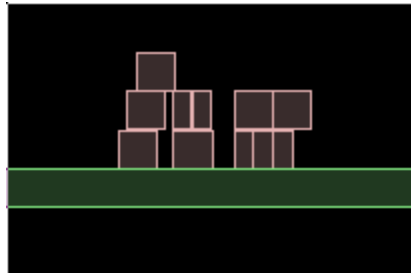
Preventing jumping in the air

Yes you guessed it, we're gonna start with the same scenario as for the [Jumping](#) topic. Dig that one up, and in the constructor add a section like this to create some bodies to jump around on in the scene:

```
1 //add some bodies to jump around on
2 {
3     //body definition
4     b2BodyDef myBodyDef;
5     myBodyDef.position.Set(-5,5);
6     myBodyDef.type = b2_dynamicBody;
7
8     //shape definition
9     b2PolygonShape polygonShape;
10    polygonShape.SetAsBox(1, 1); //a 2x2 rectangle
11
12    //fixture definition
13    b2FixtureDef myFixtureDef;
14    myFixtureDef.shape = &polygonShape;
15    myFixtureDef.density = 1;
16
17    for (int i = 0; i < 5; i++)
18    {
19        b2Fixture* fixture =
20        m_world->CreateBody(&myBodyDef)->CreateFixture(&myFixtureDef);
21        fixture->SetUserData( (void*)1 );//tag square boxes as 1
22    }
23
24    //change size
25    polygonShape.SetAsBox(0.5, 1); //a 1x2 rectangle
26
27    for (int i = 0; i < 5; i++)
28    {
29        b2Fixture* fixture =
30        m_world->CreateBody(&myBodyDef)->CreateFixture(&myFixtureDef);
31        fixture->SetUserData( (void*)2 );//tag smaller rectangular boxes as 2
32    }
33 }
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
```

2
9
3
0
3
1

This is getting pretty routine by now, the important point is that we have two different types of body there, with different shapes and user data tags. The user data does not refer to any other objects in the program, it's just a simple number tag so we can tell in the collision callback what type of things are colliding.



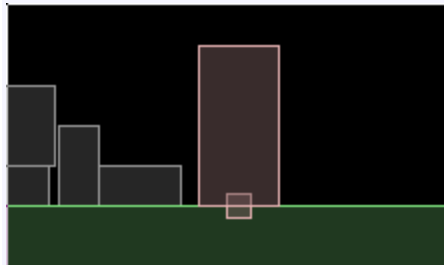
Now at the beginning of the constructor, we'll need to make a couple of changes to the 'player' box. Firstly we'll make it a bit taller so we can tell which one it is easier, and we will also add the foot-sensor fixture to it, with a user data tag of 3. You can edit the existing section to do this, but here is the full player body creation for clarity:

```
1 //player body
2 {
3     //body definition
4     b2BodyDef myBodyDef;
5     myBodyDef.type = b2_dynamicBody;
6     myBodyDef.fixedRotation = true;
7
8     //shape definition for main fixture
9     b2PolygonShape polygonShape;
10    polygonShape.SetAsBox(1, 2); //a 2x4 rectangle
11
12    //fixture definition
13    b2FixtureDef myFixtureDef;
14    myFixtureDef.shape = &polygonShape;
15    myFixtureDef.density = 1;
16
17    //create dynamic body
18    myBodyDef.position.Set(0, 10);
19    m_body = m_world->CreateBody(&myBodyDef);
20
21    //add main fixture
22    m_body->CreateFixture(&myFixtureDef);
23
24    //add foot sensor fixture
25    polygonShape.SetAsBox(0.3, 0.3, b2Vec2(0,-2), 0);
```

```

1  myFixtureDef.isSensor = true;
8  b2Fixture* footSensorFixture = m_body->CreateFixture(&myFixtureDef);
1  footSensorFixture->SetUserData( (void*)3 );
9  }
2
0
2
1
2
2
2
3
2
4
2
5
2
6
2
7
2
8
2
9

```



NOTE: it's not necessary for this foot sensor to actually be a sensor, and it doesn't need to exist at all in order to get collision events, but it provides some advantages.

- Firstly, since we know it's under the player, we know a collision with it means the player is standing
- on something. If we just used the main body to get collision events, those events could be from collisions
- with the walls or ceiling too, so we would have to check the direction before we could say it was a
- ground collision.
- Secondly, as the player moves around, especially on slopes the main body tends to bounce slightly on
- the ground which causes a large number of begin/end events to occur in quick succession. If we were
- using the main body to determine jumpability, it's possible that the user could try to jump just when

- the main body is off the ground for a few milliseconds, even though it appears to be on the ground, and
- that would just be annoying huh? Using a sensor like this will cause a smoother and more reliable contact
- detection because the sensor will stay in contact with the ground during the tiny bounces.
- Thirdly, having the foot sensor and the main body as separate fixtures means they can be given individual
- shapes as the situation requires. For example, in this case the foot sensor is much narrower than the main
- body meaning you won't be able to jump when you are teetering on the edge of something, but this could
- easily be adjusted by changing the size, shape or location of the sensor.

Now to know when we can jump, we just need to keep track of how many other fixtures are currently touching the foot sensor. Add a global variable to hold this number:

```
1 //global scope
2 int numFootContacts;
3
4 //in constructor
5 numFootContacts = 0;
```

Then we can use a collision callback similar to the standard one in the [collision callbacks](#) topic, except this time we are checking the fixture's user data instead of the body's user data:

```
1 class MyContactListener : public b2ContactListener
2 {
3     void BeginContact(b2Contact* contact) {
4         //check if fixture A was the foot sensor
5         void* fixtureUserData = contact->GetFixtureA()->GetUserData();
6         if ( (int)fixtureUserData == 3 )
7             numFootContacts++;
8         //check if fixture B was the foot sensor
9         fixtureUserData = contact->GetFixtureB()->GetUserData();
1        if ( (int)fixtureUserData == 3 )
10            numFootContacts++;
11    }
12
13    void EndContact(b2Contact* contact) {
14        //check if fixture A was the foot sensor
15        void* fixtureUserData = contact->GetFixtureA()->GetUserData();
16        if ( (int)fixtureUserData == 3 )
17            numFootContacts--;
18        //check if fixture B was the foot sensor
19        fixtureUserData = contact->GetFixtureB()->GetUserData();
20        if ( (int)fixtureUserData == 3 )
21            numFootContacts--;
22    }
23 };
24
25
```

```
8  
1  
9  
2  
0  
2  
1  
2  
2  
2  
3  
2  
4
```

Don't forget to make an instance of this class and **tell the Box2D world to refer to it for callbacks**, with `SetContactListener` !

Finally, to actually use this information, we need to ignore any attempts to jump when the foot sensor is not touching anything. You could add something like this to the relevant parts of the switch statement in the Keyboard function:

```
1 if ( numFootContacts < 1 ) break;
```

And since we're demonstrating the jumpable state let's make it really clear by showing it on-screen too:

```
1 //in Step function  
2 m_debugDraw.DrawString(5, m_textLine, "Can I jump here? %s",  
3 numFootContacts>0?"yes":"no");  
   m_textLine += 15;
```

Now run the test, and you should see the jumpability state changing correctly as the foot sensor's contact state changes, and jumping will be prevented when the player body is not on the ground. Great! Well, *almost*. As mentioned earlier a little tweaking is still required.

Preventing jumping again immediately after just jumping

When jumping using `ApplyForce` or `ApplyLinearImpulse`, if you hold down the jump key you will find that although it's not as bad as before, the player can still jump really high. This is simply because even though the jump has started and the body is moving upwards, the keypress repeats many times while the body is still close enough to the ground for the sensor to touch, so many forces/impulses get added in that short time span. To counter this, we can just limit the frequency in which jumps are possible, eg. if the player has just jumped, they shouldn't be jumping again for at least another 100ms.

To code this you would typically use a fast timer to compare the times between when you jumped, and the current time. You can find implementations of such timers for various platforms around on

the net, and the latest version of Box2D includes one too. Since the version of Box2D we are using, v2.1.2, does not have any timer we'll just make use of the fact that one timestep is 1/60 seconds to suffice as a basic timer. Let's stop the player from making jumps within 1/4 second of each other, which is 15 time steps at 60Hz. This should be enough time for the foot sensor to have moved above the ground.

```
1 //class member variable
2 int m_jumpTimeout;
3
4 //in class constructor
5 m_jumpTimeout = 0;
6
7 //in Step()
8 m_jumpTimeout--;
9
10 //in Keyboard routine (only showing the impulse version here as an example)
11 case 'I':
12     if ( numFootContacts < 1 ) break;
13     if ( m_jumpTimeout > 0 ) break;
14     //ok to jump
15     m_body->ApplyLinearImpulse( b2Vec2(0, m_body->GetMass() * 10),
16     m_body->GetWorldCenter() );
17     m_jumpTimeout = 15;
18     break;
```

Preventing jumping depending on ground type

The above method is a basic way to stop the player from jumping when he's already in the air. How about if we wanted to allow jumping only when standing on certain objects? Or maybe the jump should be higher from certain objects, etc. Let's try making jumps possible only when standing on the larger boxes in the scene. To do this we'd like to have a list of what's currently being stood on, to check in the Keyboard function when the user presses the jump key. A list like this can be kept up to date by adding fixtures to it in BeginContact, and removing them in EndContact, just like the radar example in the [sensors](#) topic.

Alternatively, you could split the numFootContacts into two variables to keep track of the number of large boxes and the number of small boxes currently stood on separately - but this is quite limited information, and if you introduce a third type of box you'll then need a third integer variable to keep track of the new type, and so on. For this demonstration we'll go with the more useful method of keeping a list of the boxes currently stood on, because it's a better long-term solution,

and also because I want to use the stood on boxes a little later.

Add a class member variable to hold the current set of boxes being stood on. You could make this a list of bodies but I will make it hold the original fixtures because remember we can also stand on the static 'ground' body which in a real game would likely have many fixtures and we would probably like to be able to tell them apart.

```
1 //global variable (include <set>)
2 set<b2Fixture*> fixturesUnderfoot;
3
4 //revised implementation of contact listener
5 class MyContactListener : public b2ContactListener
6 {
7     void BeginContact(b2Contact* contact) {
8         //check if fixture A was the foot sensor
9         void* fixtureUserData = contact->GetFixtureA()->GetUserData();
1        if ( (int)fixtureUserData == 3 )
0            fixturesUnderfoot.insert(contact->GetFixtureB()); //A is foot so B is ground
1        //check if fixture B was the foot sensor
1        fixtureUserData = contact->GetFixtureB()->GetUserData();
1        if ( (int)fixtureUserData == 3 )
2            fixturesUnderfoot.insert(contact->GetFixtureA()); //B is foot so A is ground
1    }
3
1    void EndContact(b2Contact* contact) {
4        //check if fixture A was the foot sensor
1        void* fixtureUserData = contact->GetFixtureA()->GetUserData();
5        if ( (int)fixtureUserData == 3 )
1            fixturesUnderfoot.erase(contact->GetFixtureB()); //A is foot so B is ground
6        //check if fixture B was the foot sensor
1        fixtureUserData = contact->GetFixtureB()->GetUserData();
7        if ( (int)fixtureUserData == 3 )
1            fixturesUnderfoot.erase(contact->GetFixtureA()); //B is foot so A is ground
8    }
1 };
9
2
0
2
1
2
2
2
3
2
4
2
5
2
6
2
```

7
2
8

Note that the only part of the contact listener we changed is to insert/erase from the set instead of incrementing/decrementing the counter. A set is a useful container that prevents the same object being added twice and is a little more convenient to erase from.

Now in the Keyboard function, instead of merely checking the number of stood on things, we'll check what type they are too. Specifically, we want to prevent jumping when the only thing being stood on is a small box. This means we check the current fixturesUnderfoot list, and if there is a fixture with user data tag of either 1 (a large box) or NULL (a static 'ground' fixture), then the jump should be permitted. Because this jumpability check is getting a bit long now and we want to call it from various places, I'll put this logic in a function of it's own to keep it tidy:

```
1 //class function
2 bool CanJumpNow()
3 {
4     if ( m_jumpTimeout > 0 )
5         return false;
6     set<b2Fixture*>::iterator it = fixturesUnderfoot.begin();
7     set<b2Fixture*>::iterator end = fixturesUnderfoot.end();
8     while (it != end)
9     {
10         b2Fixture* fixture = *it;
11         int userDataTag = (int)fixture->GetUserData();
12         if ( userDataTag == 0 || userDataTag == 1 ) //large box or static ground
13             return true;
14         ++it;
15     }
16     return false;
17 }
```

One drawback with the set container is the slightly more inconvenient way you need to iterate over the contents...

Replace the previous checks using numFootContacts with this function, and your ready to go. Try the program and check that you can only jump from the large boxes or the normal static ground.

Adding some reaction to the environment

Well that about covers the basics of preventing jumping in certain situations. As one last little demonstration, let's give the currently stood on boxes a kick downwards when the player jumps. This is one advantage of keeping a list instead of just an integer to keep track of them.

In the Keyboard function when a jump is performed, you could do something like this:

```
1 //in Keyboard
2 //kick player body upwards
3 b2Vec2 jumpImpulse(0, m_body->GetMass() * 10);
4 m_body->ApplyLinearImpulse( jumpImpulse, m_body->GetWorldCenter() );
5 m_jumpTimeout = 15;
6
7 //kick the underfoot boxes downwards with an equal and opposite impulse
8 b2Vec2 locationOfImpulseInPlayerBodyCoords(0, -2); //middle of the foot sensor
9 b2Vec2 locationOfImpulseInWorldCoords =
10     m_body->GetWorldPoint(locationOfImpulseInPlayerBodyCoords);
11 set<b2Fixture*>::iterator it = fixturesUnderfoot.begin();
12 set<b2Fixture*>::iterator end = fixturesUnderfoot.end();
13 while (it != end)
14 {
15     b2Fixture* fixture = *it;
16     fixture->GetBody()->ApplyLinearImpulse( -jumpImpulse, locationOfImpulseInWorldCoords );
17     ++it;
18 }
19
20
21
22
23
24
25
26
27
28
```

Now when jumping from a large box, you should see the box react to being stomped on during the jump. Admittedly this scene is not a very good one for illustrating this because if the large box is directly on the static ground no reaction will be visible - try jumping from a large box which is sitting on top of other boxes, and then you should see it kicked downwards a little.

(Note: This topic doesn't really fit in too well with the rest of the tutorial topics because they are based on Box2D v2.1.2 which does not have edge shapes. I kinda forgot about that when I started making it. This subject will only be useful if you are using Box2D v2.2 onwards.)

Ghost vertices

In the ['jumpability'](#) topic I said that the winner of the title for most commonly asked questions asked by new Box2D users is "how can I tell if my player character is on the ground?". But I may have been wrong... we have another very strong contender for this title.

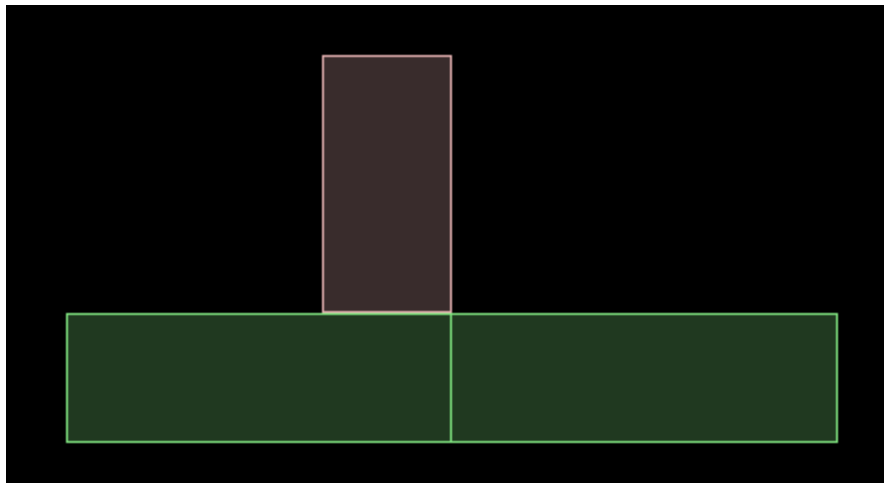
Perhaps at least as common is the question "why does my character get stuck when moving along flat ground?". This usually occurs as a result of the ground being made up of a series of fixtures (either polygon or edge shapes) placed next to each other to make up a flat surface, and a character body with a rectangular fixture trying to move along it.

The cause of this arrangement is in turn usually a side-effect of having level layouts defined as tiles in a fixed grid, which makes life easy in many ways but is somewhat sub-optimal when trying to represent a surface that should genuinely be one smooth piece (a dedicated [Box2D editor](#) would help a lot with this </shameless plug>).

Let's take a look at the classic case where this problem occurs and find out why, and then look at two ways to get around the problem. One way is to change the layout of the shapes involved, which is more of a hatchet job than a silver bullet but it is often good enough for many cases. The other way is to make use of 'ghost vertices' which will be the main focus of this topic.

Why does the character get stuck?

Here is the typical scenario where this problem shows up. The dynamic box is some kind of player or enemy moving towards the right. Just as it reaches the boundary between the two boxes in the ground below, it seems to get stuck on the corner of the box on the right.



From the [anatomy of a collision](#) topic, we know that Box2D responds to collisions between two fixtures by calculating how much they overlap, and finding the quickest way to push them apart. For this discussion, the key thing to note about this procedure is that little word 'two'. Collisions are not resolved in threes, neither are they resolved in fours. Of course, five is right out.

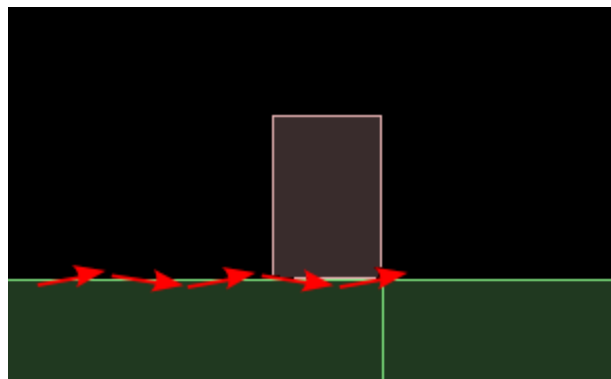
So there are actually two individual collisions going on which will be resolved separately:



In each of these the quickest way to push the overlapping fixtures apart is calculated. In the case on the left the solution is almost certainly to push the 'player' up and the ground down:



But the case on the right is not so cut and dried because this is a corner vs corner collision which are the most complicated. Recall again from the anatomy of a collision topic, that [a small change in the position of the fixtures could cause a big difference](#) in the resulting impulse that was calculated to separate them. As the 'player' body moves along the ground, it's actually moving up and down a tiny bit as the collision response works to keep it on top of the ground fixture. Here is an exaggerated image of this situation:



So at the moment the player collides with the new ground box, one of two things could happen. Zooming right in close to the corner we could see two possibilities...



... if the player was already further across the new box horizontally than it was submerged into it

vertically, we would get the outcome on the left. But if the player was overlapping more vertically than horizontally, we would get the outcome on the right, and this is how the player can momentarily get stuck on the ground that appears flat.

Fixing the problem - clipping polygon corners

As we can see above, having two corners at right angles to each other can result in a collision-correcting impulse which pushes the player back horizontally, which is the absolute worst direction. To improve on this we could clip off the corners of the player polygon, or the ground polygons, or both, to get a slightly better collision-correcting impulse direction. For example, an outcome like this would be much better:



For many cases, just making the player like this could be good enough. It's certainly an easy fix to try.

Of course, if you are able to make your player fixture into a circle, that's even better.

Fixing the problem - edge shapes

(Note: Box2D v2.1.2 does not have edge shapes)

Using edge shapes will give a huge improvement over polygon shapes. An edge shape is simply a line between two points, and these can be placed side by side to act as a ground in the same way as the boxes above. Theoretically, the same collision-correcting impulse problem applies with edges as it does for polygons, but in practise, for some reason it occurs much less frequently. I really don't know why this is, so if you do, please let us know :)

One nice thing about replacing polygons with edges is that you no longer need to worry about keeping them convex and within 8 vertices (for Box2D defaults) as you do with polygons. And when you have an area of ground which should be perfectly flat, it's trivial to replace the many edges with a single edge.

Really fixing the problem - ghost vertices

(Note: Box2D v2.1.2 does not have edge shapes)

Box2D v2.2 introduced the concept of 'ghost vertices' for edge shapes. Each edge shape obviously has two main vertices used to define its position and detect when the edge has collided with something. These main vertices can each have a ghost vertex on the 'outside', as in this diagram:



The ghost vertices get their name because they don't play a part in collision detection - other bodies can happily pass through the lines from v0-v1 and v2-v3 without colliding with the edge. So... what's the point of these ghost edges then? The point is that they play a part in collision *response*.

For example when a collision is detected at the v2 end of the main segment, both the v1-v2 and v2-v3 lines will be used to calculate the collision-response impulse. Remember how the whole getting stuck problem was caused by not being able to resolve collisions with more than one other fixture at a time? Well essentially, this is exactly what the ghost vertex system does for us. The result is that the collision response will act as if there was no discontinuous point at v2.

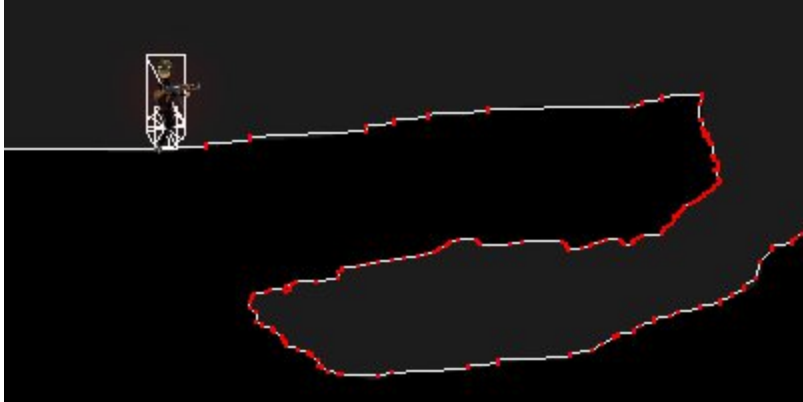
Using these ghost vertices is most commonly done by way of the `b2ChainShape` which sets up a string of edge shapes and takes care of all the ghost vertices automatically, placing the ghost vertices of each edge to match the main vertices of the edge on each side of it. If you need to set up a single edge with ghost vertices, you could do it like this:

```
1 b2EdgeShape edgeShape;  
2 edgeShape.Set( v1, v2 );  
3 edgeShape.m_vertex0.Set( v0 );  
4 edgeShape.m_vertex3.Set( v3 );  
5 edgeShape.m_hasVertex0 = true;  
6 edgeShape.m_hasVertex3 = true;
```

...where the vertex numbers are as in the diagram above. Note that the main edge is v1-v2 since it's in middle, and the ghosts being on the outside are v0 and v3.

Other things to remember

One thing to be careful of when making use of ghost vertices to get smooth collisions between edges, is making the edges too small in comparison to the 'player' or whatever will be colliding with them. For example if a collision occurs at the v2 end of the edge and the edges are so small that both v2 and v3 are completely inside the player fixture, the ghost system can fail. Here is a good example of a ground made up from edges that were too small and needed to be made larger (the red dots are the vertices making up the ground surface).



Sourcecode

Since this topic needs a more recent version of Box2D than the rest of the tutorial topics, I'll keep the source code here separately. This is a 'test' for the testbed. [iforce2d_ghost_vertices.h](#)

Finally, here is a video showing a comparison between polygons (bottom), edges without ghost vertices (middle), and edges with ghost vertices (top). Even with polygons, I found this 'sticking in the ground' to be very rare, in fact far too rare to make a video with! So in this example the ground has been made dynamic which slows down the rate of collision resolution, the player is made of a light lower body and a heavy upper body (50 times more dense!) and then strong forces are applied downwards on the upper body.

Even then the sticking was still rare, so I also added a downwards impulse just as the player got close to one of the 'sticky' points. Given that it took all this effort to get the player to stick on the non-ghost-vertex edges, they are probably just fine for most uses. But if you want a guaranteed solution you might want to try the ghost-vertex method - even with all this evil meddling, the player never got stuck once.

Joints

Box2D has a number of 'joints' that can be used to connect two bodies together. These joints can be used to simulate interaction between objects to form hinges, pistons, ropes, wheels, pulleys, vehicles, chains, etc. Learning to use joints effectively helps to create a more engaging and interesting scene.

Lets take a quick look at the available joints, then go over their characteristics, then finally we'll make an example using a few of the most commonly used ones. Here are the joints in Box2D v2.1.2:

- Revolute - a hinge or pin, where the bodies rotate about a common point
- Distance - a point on each body will be kept at a fixed distance apart
- Prismatic - the relative rotation of the two bodies is fixed, and they can slide along an axis
- Line - a combination of revolute and prismatic joints, useful for modelling vehicle suspension
- Weld - holds the bodies at the same orientation

- Pulley - a point on each body will be kept within a certain distance from a point in the world,
- where the sum of these two distances is fixed, kinda... (sheesh... there is no succinct way to describe this)
- Friction - reduces the relative motion between the two bodies
- Gear - controls two other joints (revolute or prismatic) so that the movement of one affects the other
- Mouse - pulls a point on one body to a location in the world

Joints added after v2.1.2:

- Wheel - the line joint, renamed
- Rope - a point on each body will be constrained to a maximum distance apart

In the source code, the actual joint classes are named like `b2RevoluteJoint`, `b2DistanceJoint`, etc.

Creating a joint

Joints are created in a similar way to bodies and fixtures, in that you setup a 'definition' which is then used to create the joint instance itself. The Box2D world manages the actual construction of the joint instance, and when you are done with it you tell the world to destroy it. This process is nothing new for us by now, but just to recap here is how the typical creation of a joint goes (the xxx will be replaced with one of the joint names above):

```
1 //set up the definition for a xxx joint
2 b2xxxJointDef jointDef;
3 jointDef.xxx = ...;
4
5 //create the joint
6 b2xxxJoint* joint = (b2xxxJoint*)world->CreateJoint( &jointDef );
```

Although the overall method is similar, there is one very important difference to creating bodies and fixtures. Let's recap some more: when creating a body, we used the function `CreateBody` which returns a `b2Body*`, and likewise `CreateFixture` returns a `b2Fixture*`. To get a body with different features, eg. static or dynamic, we set the appropriate field in the definition, and we got a `b2Body*` with those characteristics. Even with different characteristics, the behaviour of bodies is quite similar.

Joints on the other hand each have quite different behavior, in fact so much so that they are implemented as a separate classes, like `b2RevoluteJoint`, `b2DistanceJoint` etc. The class `b2Joint` is the parent of all the classes mentioned above, but it is an abstract class which is never used directly. So when creating joints, although the function `CreateJoint` does return a `b2Joint*` type as expected, this is really a pointer to an instance of one of the classes above.

That's why the returned value from `CreateJoint` has been cast to a `b2xxxJoint*` in the code above. Of course if you don't need to keep a reference to the joint, you can just call `CreateJoint` without even storing the returned value at all.

Joint definitions - common settings

Although each joint has a different behavior, you might have guessed from the fact that they have a common parent, they have some features in common too. Here are the fields which are common to every joint definition:

- `bodyA` - one of the bodies joined by this joint (required!)
- `bodyB` - the other body joined by this joint (required!)
- `collideConnected` - specifies whether the two connected bodies should collide with each other

The two body settings are obviously the bodies this joint will act on. In some cases it *does* make a difference which is which, depending on the joint type. See the discussion of each joint for details.

The `collideConnected` setting allows you to say whether the two bodies should still obey the normal collision rules or not. For example if you are making a rag-doll, you'll probably want to let the upper leg and lower leg segments overlap at the knee, so you would set `collideConnected` to `false`. If you are making an elevator platform, you'll probably want the platform to collide with the ground, so `collideConnected` would be `true`. The default value is `false`.

The code up to this point is quite simple, here's an example:

```
1 jointDef.bodyA = upperLegBody;  
2 jointDef.bodyB = lowerLegBody;  
3 jointDef.collideConnected = false;
```

Joint definitions - specific settings

After setting the common fields in a joint definition, you'll need to specify the details for the type of joint you are making. This commonly includes an anchor point on each body, limits on the range of movement, and motor settings. These need to be discussed in more detail for each joint since they are used slightly differently, but here is a general overview of what these terms mean.

- **anchor points**
 - Typically a point on each body is given as the location around which the bodies must interact. Depending on
 - the joint type, this point will be the center of rotation, the locations to keep a certain distance apart, etc.
- **joint limits**
 - Revolute and prismatic joints can be given limits, which places a restriction on how far the bodies can
 - rotate or slide.
- **joint motors**
 - Revolute, prismatic and line (wheel) joints can be given motor settings, which means that instead of spinning
 - or sliding around freely, the joint acts as if it had it's own power source. The motor is given

- a maximum force or torque, and this can be used in combination with a target velocity to spin or slide bodies in relation to each other.
- If the velocity is zero, the motor acts like a brake because it aims to reduce any motion between the bodies.

See the topics below for a discussion of the specific features of each joint.

Controlling joints and getting feedback

After a joint is created you can alter the parameters for it's behaviour such as changing the motor speed, direction or strength, enabling or disabling the limits. This can be very useful to build a range of interesting scenes and contraptions, for example wheels to drive a car, an elevator which moves up and down, a drawbridge that opens and closes.

You can also get information about what position the joint is at, how fast it is moving etc. This is useful if you want to use the activity of the joint in your game logic. You can also get the force and torque that the joint is applying to the bodies to keep them in the right place, which can be useful if you want to allow the joint to break when it sustains too much force.

Cleaning up

There are two ways a joint can be destroyed. The first is the same way that bodies and fixtures are destroyed:

```
1 world->DestroyJoint( myJoint );
```

The other way which is not so obvious, is that joints are destroyed when one of the bodies they connect is destroyed. There is no point in having a joint which doesn't connect anything. This means that if you are changing joints during the simulation, you need to keep in mind the order in which you delete bodies and joints. For example note the differences between these situations:

```
1 // 'myJoint' connects body1 and body2
2
3 //BAD!
4 world->DestroyBody( body1 ); // myJoint becomes invalid here
5 world->DestroyJoint( myJoint ); // crash
6
7 //OK
8 world->DestroyJoint( myJoint );
9 world->DestroyBody( body1 );
```

Generally if you know the game logic, and when joints are created and how objects are likely to be destroyed, you can arrange your cleanup routines to avoid the first case above. However if you are working on a complex project you might find it easier to use the 'destruction listener' feature

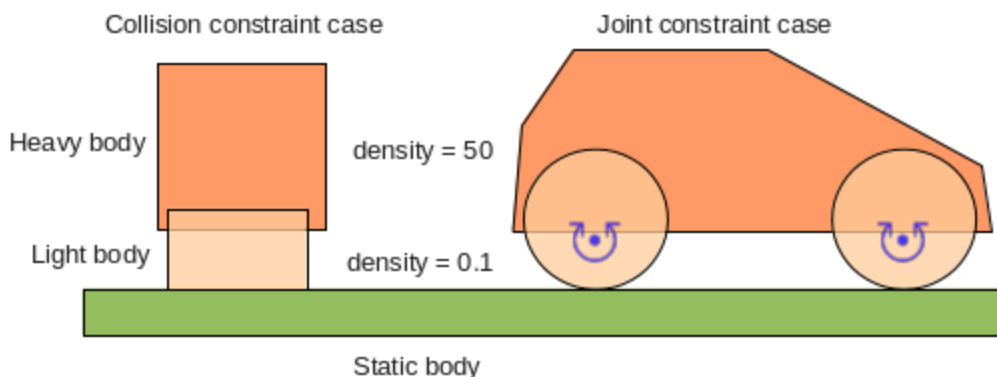
provided by Box2D. This allows you to receive a notification when any joint is destroyed so you know not to use it any more. Check out the section on 'Implicit destruction' in the 'Loose Ends' section of the [user manual](#).

Trouble with joint positions

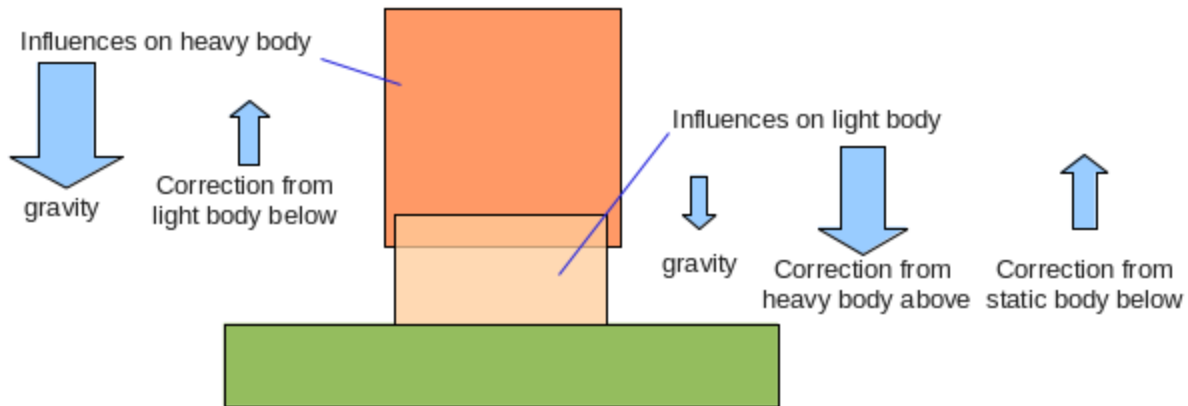
Sometimes you might find that a joint has difficulty keeping the two anchor points in the same place. This has to do with the way the bodies are corrected by the joint. The joint calculates the impulse necessary to push the bodies back into the right alignment, and this impulse is applied to each body with the heavier body being given less of the impulse, and the lighter body being given more.

If there are no other restrictions present (eg. where the two bodies are only interacting with each other) this is fine, and the bodies move as expected according to the impulses. However when another constraint is applied, this balancing of the impulse can be upset by the action of other impulses, usually those acting on the lighter of the two bodies. This is typically seen when a light body has two heavier bodies either joined to it or simply pressing against it. In fact, the same problem can be seen with simple collision resolution where a light body is sandwiched between two significantly heavier bodies, so this issue is not solely a problem with joints.

These are the typical cases:



Although in the left case the bodies are not joined, the same type of calculation is going on to resolve the overlapping fixtures. In each of these situations looking at the forces involved shows that the light body really is caught between a rock and a hard place.



Because it is light, it is affected most by the correcting impulse, and the heavier body on top is not corrected very much. The result is that it can take many time steps for the mis-alignment (or overlap in the case of a collision) to be resolved. Another very common situation where this occurs is when a chain is made from light segments, and then a heavy body is attached to the end - in that case the light bodies get 'stretched' rather than squashed as above, but the cause of the problem is the same.

The simplest way to overcome this problem is to make bodies that must interact with each other in this way have similar masses. If this seems unreasonable bear in mind that in the real world nothing is perfectly rigid so we never really see this situation occurring, and ideally I guess the object in the middle should be crushed somehow but that's a tricky thing to program. In the case where a heavy object hangs on a chain of light segments, this too in the real world results in either a deformation of the segments or a breakage of the chain altogether (fortunately that's not so tricky to program).

Details on how to use Box2D joints

While joints have some things in common, the differences between them are too great to cover in one topic. Check out the links below for a detailed discussion of specific joints and their usage.

- [Revolute](#)
- [Prismatic](#)

Revolute joints

The revolute joint can be thought of as a hinge, a pin, or an axle. An anchor point is defined on each body, and the bodies will be moved so that these two points are always in the same place, and the relative rotation of the bodies is not restricted.

Revolute joints can be given limits so that the bodies can rotate only to a certain point. They can also be given a motor so that the bodies will try to rotate at a given speed, with a given torque.

Common uses for revolute joints include:

- wheels or rollers
- chains or swingbridges (using multiple revolute joints)
- rag-doll joints
- rotating doors, catapults, levers

Creating a revolute joint

A revolute joint is created by first setting the desired properties in a `b2RevoluteJointDef`, then passing that to `CreateJoint` to get an instance of a `b2RevoluteJoint` object. We saw in the [joints overview](#) that all joint definitions have some properties in common - the two bodies to be joined, and whether they should collide with each other. So we'll set those first:

```
1 b2RevoluteJointDef revoluteJointDef;  
2 revoluteJointDef.bodyA = bodyA;  
3 revoluteJointDef.bodyB = bodyB;  
4 revoluteJointDef.collideConnected = false;
```

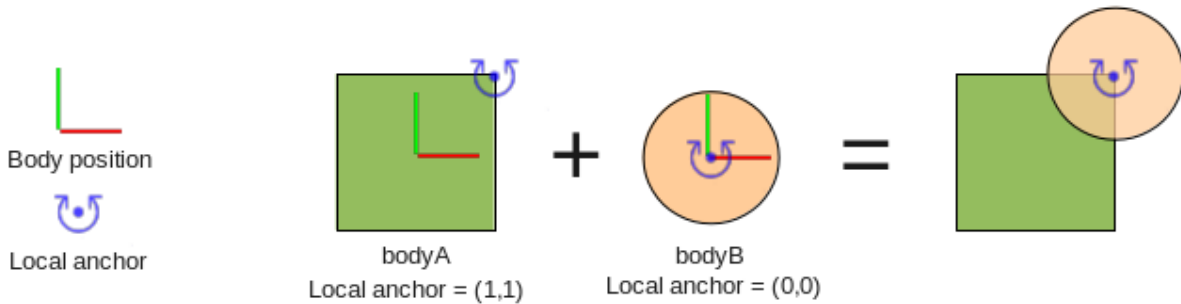
Then we come to a bunch of properties which are specific to revolute joints.

- `localAnchorA` - the point in body A around which it will rotate
- `localAnchorB` - the point in body B around which it will rotate
- `referenceAngle` - an angle between bodies considered to be zero for the joint angle
- `enableLimit` - whether the joint limits will be active
- `lowerAngle` - angle for the lower limit
- `upperAngle` - angle for the upper limit
- `enableMotor` - whether the joint motor will be active
- `motorSpeed` - the target speed of the joint motor
- `maxMotorTorque` - the maximum allowable torque the motor can use

Let's take a look at what these do in more detail.

Local anchors

The local anchor for each body is a point given in local body coordinates to specify what location of the body will be at the center of the rotation. For example, if body A has a 2x2 square polygon fixture, and bodyB has a circular fixture, and you want the circle to rotate about it's center at one corner of the square...



...you would need local anchors of (1,1) and (0,0) respectively. So the code for this case would be:

```
1 revoluteJointDef.localAnchorA.Set(1,1);
2 revoluteJointDef.localAnchorB.Set(0,0);
```

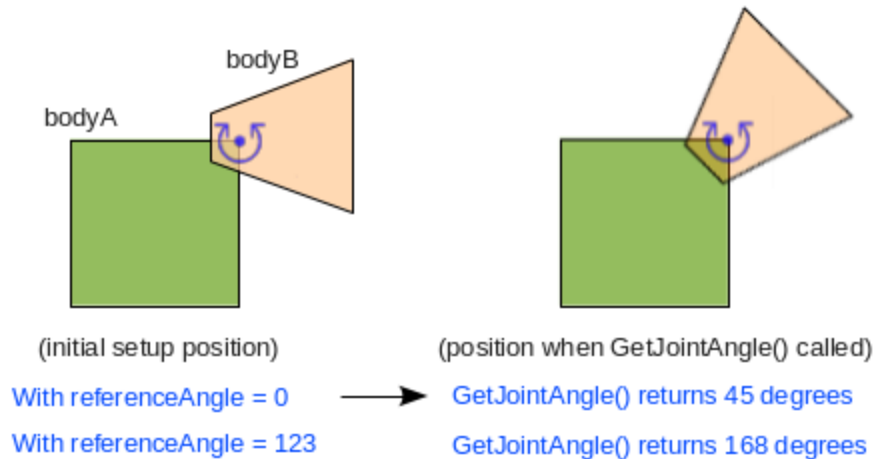
Note that it is not necessary to have actually have a fixture at the joint anchor position. In the example above, the anchor for bodyA could just as easily be made (2,2) or any other location without any problems at all, even if the circle and the box are not touching. Joints connect bodies, not fixtures.

You can access the anchor points after creating a joint by using `GetAnchorA()` and `GetAnchorB()`. Remember the returned locations are in local body coordinates, the same as you set in the joint definition.

Reference angle

The reference angle allows you to say what the 'joint angle' between the bodies is at the time of creation. This is only important if you want to use `GetJointAngle()` later in your program to find out how much the joint has rotated, or if you want to use joint limits.

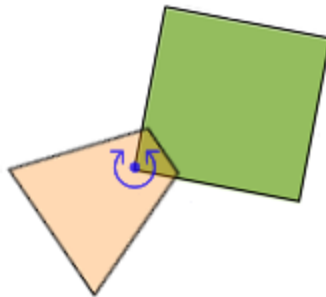
Typically, you will set up the bodies so that the initial rotation is considered to be the zero 'joint angle'. Here is an example of what to expect from `GetJointAngle()` in the typical case, and the case where a non-zero `referenceAngle` is given.



So a typical example is simply as below (the default value is zero anyway so this doesn't actually do anything).

```
1 revoluteJointDef.referenceAngle = 0;
```

Note that the joint angle increases as bodyB rotates counter-clockwise **in relation to bodyA**. If both bodies were moved or rotated in the same way, the joint angle does not change because it represents the relative angle between the bodies. For example, the case below will return the same values for GetJointAngle as the right-hand case above.

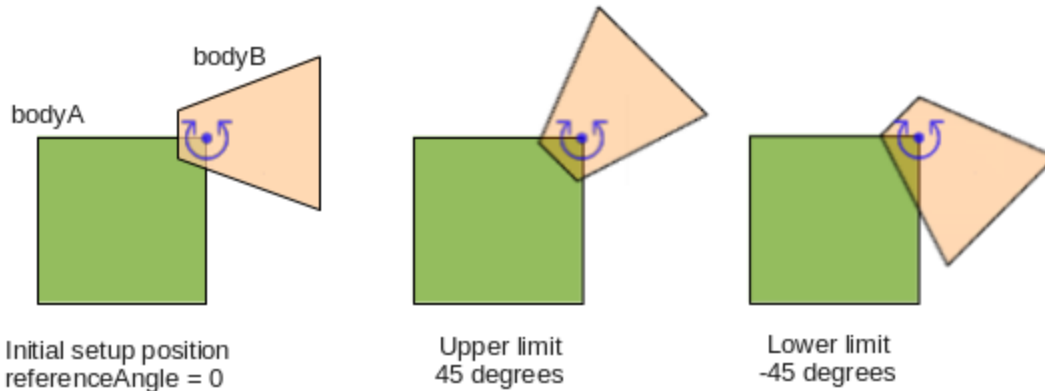


Also, please remember that all angles in Box2D are dealt with in radians, I am just using degrees here because I find them easier to relate to :p

Revolute joint limits

With only the properties covered so far the two bodies are free to rotate about their anchor points indefinitely, but revolute joints can also be given limits to restrict their range of rotation. A lower and upper limit can be specified, given in terms of the 'joint angle'.

Suppose you want the joint in the example above to be restricted to rotating within 45 degrees of it's initial setup angle.



The diagram should explain it fairly well - remember that counter-clockwise rotation of bodyB means an increase in the joint angle, and angles are in radians:

```
1 revoluteJointDef.enableLimit = true;
2 revoluteJointDef.lowerAngle = -45 * DEGTORAD;
3 revoluteJointDef.upperAngle = 45 * DEGTORAD;
```

The default value for enableLimit is false. You can also get or set these limit properties for a joint after it has been created, by using these functions:

```
1 //alter joint limits
2 void EnableLimit(bool enabled);
3 void SetLimits( float lower, float upper );
4
5 //query joint limits
6 bool IsLimitEnabled();
7 float GetLowerLimit();
8 float GetUpperLimit();
```

Some things to keep in mind when using joint limits...

- The enableLimits settings affects both limits, so if you only want one limit you will need to set
- the other limit to a very high (for upper limit) or low (for lower limit) value so that it is never reached.
- A revolute joint's limits can be set so that it rotates more than one full rotation, for example a
- lower/upper limit pair of -360,360 would allow two full rotations between limits.
- Setting the limits to the same value is a handy way to 'clamp' the joint to a given angle. This angle
- can then be gradually changed to rotate the joint to a desired position while staying impervious to
- bumping around by other bodies, and without needing a joint motor.
- Very fast rotating joints can go past their limit for a few time steps until they are corrected.
- Checking if a joint is currently at one of its limits is pretty simple:
- `bool atLowerLimit = joint->GetJointAngle() <= joint->GetLowerLimit();`

- `bool atUpperLimit = joint->GetJointAngle() >= joint->GetUpperLimit();`

Revolute joint motor

The default behaviour of a revolute joint is to rotate without resistance. If you want to control the movement of the bodies you can apply torque or angular impulse to rotate them. You can also set up a joint 'motor' which causes the joint to try to rotate at a specific angular velocity. This is useful if you want to simulate powered rotation such as a car wheel or drawbridge type door.

The angular velocity specified is only a target velocity, meaning there is no guarantee that the joint will actually reach that velocity. By giving the joint motor a maximum allowable torque, you can control how quickly the joint is able to reach the target velocity, or in some cases whether it can reach it at all. The behavior of the torque acting on the joint is the same as in the [forces and impulses](#) topic. A typical setting might look like this.

```
1 revoluteJointDef.enableMotor = true;
2 revoluteJointDef.maxMotorTorque = 20;
3 revoluteJointDef.motorSpeed = 360 * DEGTORAD; //1 turn per second counter-clockwise
```

The default value for enableMotor is false. You can also get or set these motor properties for a joint after it has been created, by using these functions:

```
1 //alter joint motor
2 void EnableMotor(bool enabled);
3 void SetMotorSpeed(float speed);
4 void SetMaxMotorTorque(float torque);
5
6 //query joint motor
7 bool IsMotorEnabled();
8 float GetMotorSpeed();
9 float GetMotorTorque();
```

Some things to keep in mind when using joint motors...

- With a low max torque setting, the joint can take some time to reach the desired speed. If you make
- the connected bodies heavier, you'll need to increase the max torque setting if you want to keep
- the same rate of acceleration.
- The motor speed can be set to zero to make the joint try to stay still. With a low max torque setting
- this acts like a brake, gradually slowing the joint down. With a high max torque it acts to stop the
- joint movement very quickly, and requires a large force to move the joint, kind of like a rusted up wheel.
- The driving wheels of a car or vehicle can be simulated by changing the direction and size

- of the motor
- speed, usually setting the target speed to zero when the car is stopped.

Revolute joint example

Okay, let's make some revolute joints in the testbed to try out these settings and see how they work. First let's make as simple a joint as possible. We'll make one like in the first diagram on this page, using a square and a circle fixture. Since we have already covered many examples I will no longer be showing the full listing for all the code, but in most cases it's probably convenient to start with a basic 'fenced in' type scene as in some of the [previous topics](#) to stop things flying off the screen.

Into the scene we will first need to create the box and the circle bodies. It is very common to place the bodies at the position they will be joined in, but for the purpose of demonstration I will make the bodies start a little apart from each other so we can watch how the joint behaves in this situation.

```
1 //body and fixture defs - the common parts
2 b2BodyDef bodyDef;
3 bodyDef.type = b2_dynamicBody;
4 b2FixtureDef fixtureDef;
5 fixtureDef.density = 1;
6
7 //two shapes
8 b2PolygonShape boxShape;
9 boxShape.SetAsBox(2,2);
10 b2CircleShape circleShape;
11 circleShape.m_radius = 2;
12
13 //make box a little to the left
14 bodyDef.position.Set(-3, 10);
15 fixtureDef.shape = &boxShape;
16 m_world->CreateBody( &bodyDef );
17 m_world->CreateFixture( &fixtureDef );
18
19 //and circle a little to the right
20 bodyDef.position.Set( 3, 10);
21 fixtureDef.shape = &circleShape;
22 m_world->CreateBody( &bodyDef );
23 m_world->CreateFixture( &fixtureDef );
24
25
26
27
28
29
30
31
32
```

```
0
2
1
2
2
2
2
3
```

I have left out a few details such as the declaration of the class variables to store the bodies, but you are way smarter than to be scratching your head about that by now right? Next we make a revolute joint using the first few properties covered above:

```
1 b2RevoluteJointDef revoluteJointDef;
2 revoluteJointDef.bodyA = m_bodyA;
3 revoluteJointDef.bodyB = m_bodyB;
4 revoluteJointDef.collideConnected = false;
5 revoluteJointDef.localAnchorA.Set(2,2);//the top right corner of the box
6 revoluteJointDef.localAnchorB.Set(0,0);//center of the circle
7 m_joint = (b2RevoluteJoint*)m_world->CreateJoint( &revoluteJointDef );
```

Running this you should see the box and circle joined together with the center of the circle at one corner of the box, freely rotating. If you have the 'draw joints' checkbox checked, you will see blue lines are drawn between the body position and the anchor position. You might like to also add something like this in the Step() function so you can confirm the joint angle in real time:

```
1 m_debugDraw.DrawString(5, m_textLine, "Current joint angle: %f deg",
2 m_joint->GetJointAngle() * RADTODEG);
3 m_textLine += 15;
4 m_debugDraw.DrawString(5, m_textLine, "Current joint speed: %f deg/s",
5 m_joint->GetJointSpeed() * RADTODEG);
6 m_textLine += 15;
```

If you pause the simulation and restart it, then click 'single step', you will be able to see that for a brief moment, the two bodies are in their initial setup positions, and then the joint constraints



take effect.

Remember that since joints cannot create a perfect constraint like in the real world, in some cases you might find that the joined bodies can stray from their correct position, as mentioned in the [joints overview](#) topic.

Try moving the anchor positions to see how you can place the pivot point in different locations. For example, try these:

```
1 //place the bodyB anchor at the edge of the circle
2 revoluteJointDef.localAnchorB.Set(-2,0);
```

```

3
4 //place the bodyA anchor outside the fixture
5 revoluteJointDef.localAnchorA.Set(4,4);

```

Now lets set some joint limits. Following the example above, we'll make it able to move within the range -45 to 45 degrees.

```

1 revoluteJointDef.enableLimit = true;
2 revoluteJointDef.lowerAngle = -45 * DEGTORAD;
3 revoluteJointDef.upperAngle = 45 * DEGTORAD;

```

You should see the rotation of the bodies restricted. If you put the square body into a corner to keep it still, and pull on the circle body so that it pushes against one of the limits, you will be able to see how the joint angle can sometimes go a little bit over the limits when it has a strong force against it, as mentioned above.

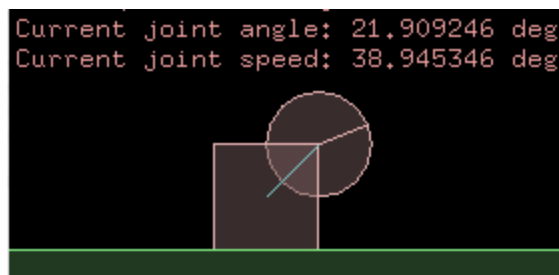
Next, let's add a joint motor, also as in the example above. It's more interesting to be able to switch the direction of the motor at runtime, so you could also add a class variable to hold the current motor direction, and use the Keyboard() function to switch it as in some of the [previous topics](#).

```

1 revoluteJointDef.enableMotor = true;
2 revoluteJointDef.maxMotorTorque = 5;
3 revoluteJointDef.motorSpeed = 90 * DEGTORAD;//90 degrees per second

```

This setting aims to rotate the joint 90 degrees per second, which means it should be able to get from one limit to the other in one second. However, the torque is a little low for the rotational inertia of the bodies involved. If you implemented the keyboard switching of the motor direction, switch directions back and forward and you will see that it takes a little time to speed up.



If you change the max torque setting to a higher value, around 60, you will see that the joint can accelerate fast enough so that it does actually reach 90 degrees per second before hitting the other limit. Try disabling the limit so you can see how the motor will keep the joint at a stable speed. Add another wheel and you have a simple car!

Simple chain example

Since chains are quite a common use for revolute joints, we'll give that a try too. A chain is just a

bunch of bodies joined together, so they are quite simple to make. First we'll try a loose chain flopping around in the world.

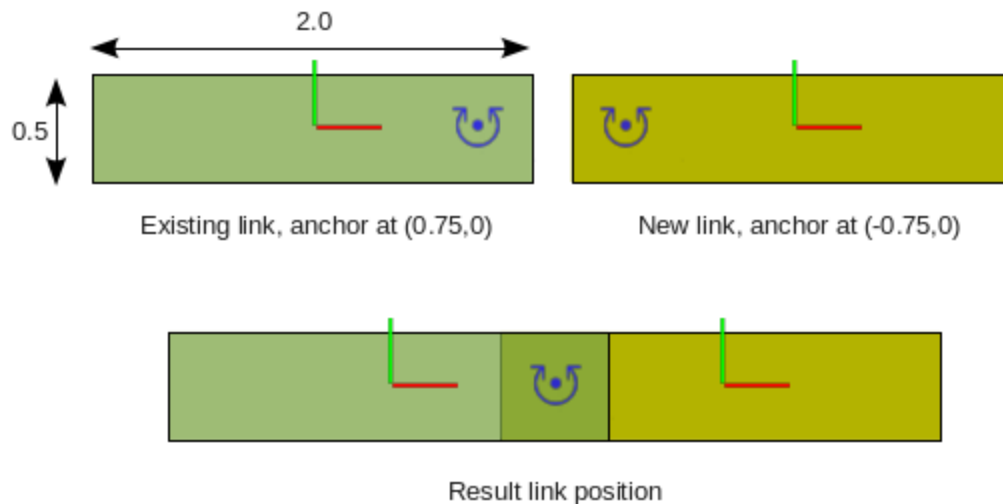
It's best to use a loop to make a chain because the contents are repetitive and with a loop we just change one number to get a longer chain. In one iteration of the loop we need to create a new body as a link in the chain, and attach it to the previous link. To start, let's create the bodies we'll need first:

```
1 //body and fixture defs are common to all chain links
2 b2BodyDef bodyDef;
3 bodyDef.type = b2_dynamicBody;
4 bodyDef.position.Set(5,10);
5 b2FixtureDef fixtureDef;
6 fixtureDef.density = 1;
7 b2PolygonShape polygonShape;
8 polygonShape.SetAsBox(1,0.25);
9 fixtureDef.shape = &polygonShape;
1
0 //create first link
1 b2Body* link = m_world->CreateBody( &bodyDef );
1 link->CreateFixture( &fixtureDef );
1
2 //use same definitions to create multiple bodies
1 for (int i = 0; i < 10; i++) {
3     b2Body* newLink = m_world->CreateBody( &bodyDef );
1     newLink->CreateFixture( &fixtureDef );
4
1     //...joint creation will go here...
5
1     link = newLink;//prepare for next iteration
6 }
1
7
1
8
1
9
2
2
0
2
1
2
2
2
2
3
```

This is a good example of how useful it can be to define a body or fixture once, and then use the definition multiple times :) Notice that all the bodies are in the same place, so when the test starts you'll get a wacky situation where they push each other out of the way initially:

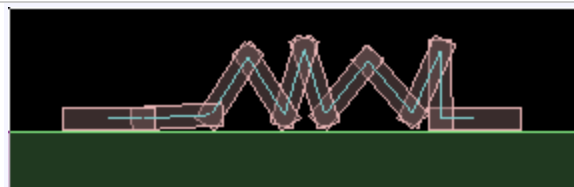


Now let's think about where the joint anchors will go. Suppose we want them to be at the end of each chain link, preferably centered rather than on a corner, and inset a little from the end so that when the link bends it doesn't make any large gaps appear on the outer side.



The code to do this is pretty easy, it's just a few extra lines:

```
1 //set up the common properties of the joint before entering the loop
2 b2RevoluteJointDef revoluteJointDef;
3 revoluteJointDef.localAnchorA.Set( 0.75,0);
4 revoluteJointDef.localAnchorB.Set(-0.75,0);
5
6 //inside the loop, only need to change the bodies to be joined
7 revoluteJointDef.bodyA = link;
8 revoluteJointDef.bodyB = newLink;
9 m_world->CreateJoint( &revoluteJointDef );
```

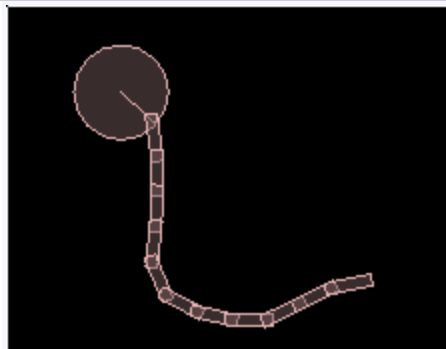


Right at the beginning of the simulation, the link bodies are still all piled on top of each other so for a proper game you would want to place the links in a more sensible position to start with, but the anchor positions will be the same. Notice that the `collideConnected` property (default is false) means that each link in the chain will not collide with it's neighboring links, but it will still collide with all the other links.

Finally, let's try attaching one end of the chain to a grounded body. Make a dynamic body with a circle fixture, and set up a revolute joint to connect it to a static body at it's center. The testbed

already has a static body at (0,0) which we can use for this.

```
1 //body with circle fixture
2 b2CircleShape circleShape;
3 circleShape.m_radius = 2;
4 fixtureDef.shape = &circleShape;
5 b2Body* chainBase = m_world->CreateBody( &bodyDef );
6 chainBase->CreateFixture( &fixtureDef );
7
8 //a revolute joint to connect the circle to the ground
9 revoluteJointDef.bodyA = m_groundBody;//provided by testbed
10 revoluteJointDef.bodyB = chainBase;
11 revoluteJointDef.localAnchorA.Set(4,20);//world coords, because m_groundBody is at (0,0)
12 revoluteJointDef.localAnchorB.Set(0,0);//center of circle
13 m_world->CreateJoint( &revoluteJointDef );
14
15 //another revolute joint to connect the chain to the circle
16 revoluteJointDef.bodyA = link;//the last added link of the chain
17 revoluteJointDef.bodyB = chainBase;
18 revoluteJointDef.localAnchorA.Set(0.75,0);//the regular position for chain link joints, as above
19 revoluteJointDef.localAnchorB.Set(1.75,0);//a little in from the edge of the circle
20 m_world->CreateJoint( &revoluteJointDef );
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```



Prismatic joints

The prismatic joint is probably more commonly known as a slider joint. The two joined bodies have their rotation held fixed relative to each other, and they can only move along a specified axis. Prismatic joints can be given limits so that the bodies can only move along the axis within a specific range. They can also be given a motor so that the bodies will try to move at a given speed, with a given force. Common uses for prismatic joints include:

- elevators
- moving platforms
- sliding doors
- pistons

Creating a prismatic joint

A prismatic joint is created by first setting the desired properties in a `b2PrismaticJointDef`, then passing that to `CreateJoint` to get an instance of a `b2PrismaticJoint` object. We saw in the [joints overview](#) that all joint definitions have some properties in common - the two bodies to be joined, and whether they should collide with each other. So we'll set those first:

```
1 b2PrismaticJointDef prismaticJointDef;  
2 prismaticJointDef.bodyA = bodyA;  
3 prismaticJointDef.bodyB = bodyB;  
4 prismaticJointDef.collideConnected = false;
```

Then we come to a bunch of properties which are specific to prismatic joints.

- `localAxis1*` - the axis (line) of movement (relative to bodyA)
- `referenceAngle` - the angle to be enforced between the bodies
- `localAnchorA` - a point in body A to keep on the axis line
- `localAnchorB` - a point in body B to keep on the axis line
- `enableLimit` - whether the joint limits will be active
- `lowerTranslation` - position of the lower limit
- `upperTranslation` - position of the upper limit
- `enableMotor` - whether the joint motor will be active
- `motorSpeed` - the target speed of the joint motor
- `maxMotorForce` - the maximum allowable force the motor can use

* Changed to `localAxisA` in Box2D v2.2.0

Let's take a look at what these do in more detail.

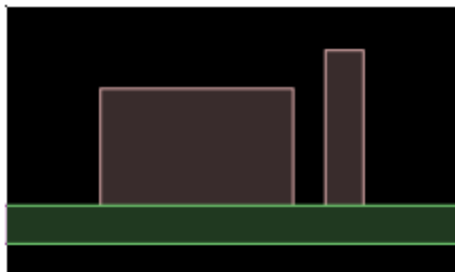
As an example of setting up prismatic joints, we'll make a simple forklift. This will make use of joint limits and motors. Here are the bodies we'll use for this - one large box for the chassis and a smaller one for the lift slider. Since we have already covered many examples I will no longer be showing the full listing for all the code, but it's probably convenient to start with a basic 'fenced in' type scene as in some of the [previous topics](#) to stop things flying off the screen.

```
1 //body and fixture defs - the common parts
```

```

2  b2BodyDef bodyDef;
3  bodyDef.type = b2_dynamicBody;
4  b2FixtureDef fixtureDef;
5  fixtureDef.density = 1;
6
7  //two boxes
8  b2PolygonShape squareShapeA;
9  squareShapeA.SetAsBox(5,3);
1 b2PolygonShape squareShapeB;
0 squareShapeB.SetAsBox(1,4);
1
1 //large box a little to the left
1 bodyDef.position.Set(-10, 10);
2 fixtureDef.shape = &squareShapeA;
1 m_bodyA = m_world->CreateBody( &bodyDef );
3 m_bodyA->CreateFixture( &fixtureDef );
1
4 //smaller box a little to the right
1 bodyDef.position.Set( -4, 10);
5 fixtureDef.shape = &squareShapeB;
1 m_bodyB = m_world->CreateBody( &bodyDef );
6 m_bodyB->CreateFixture( &fixtureDef );
1
7
1
8
1
9
2
0
2
1
2
2
2
3

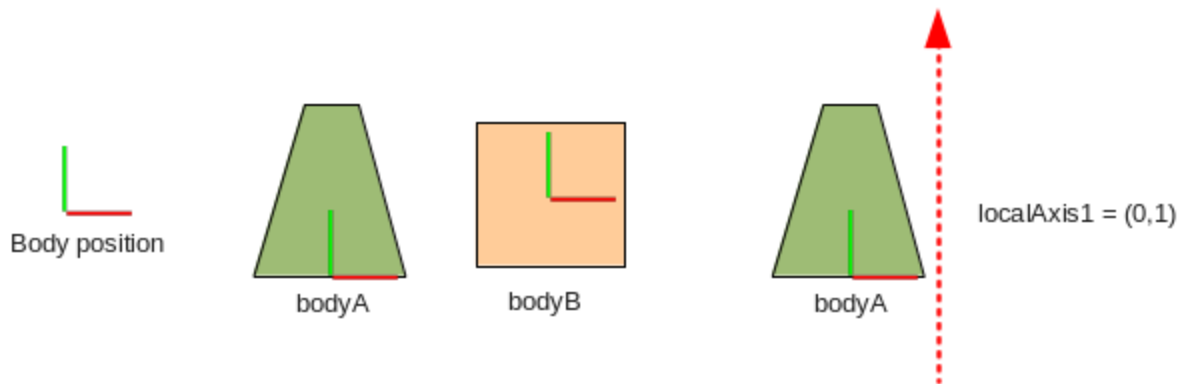
```



Forklift main body and lift slider

Joint axis

The joint axis is the line along which the bodies can move relative to each other. It is specified in bodyA's local coordinates, so you could think of it as the direction bodyB can move from bodyA's point of view. For example if bodyA has a trapezoidal fixture, and bodyB has a square polygon fixture as below, and you want bodyB to be able to slide along the direction the trapezoid is 'pointing'...

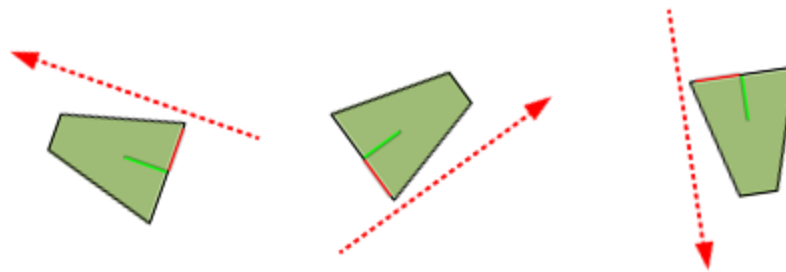


...you would need a local axis of (0,1). So the code for this case would be:

```
1 prismaticJointDef.localAxis1.Set(0,1);
```

* Changed to localAxisA in Box2D v2.2.0

This means that as bodyA moves around in the world, the line along which bodyB can slide will move with it, for example:



Note that the axis itself is not related to any particular point in the body, it only specifies a direction for the sliding movement. That's why I have intentionally shown it outside the body's fixture in the diagram :) The axis given should be a unit vector, so before you create the joint remember to normalize the vector if it had a length other than 1:

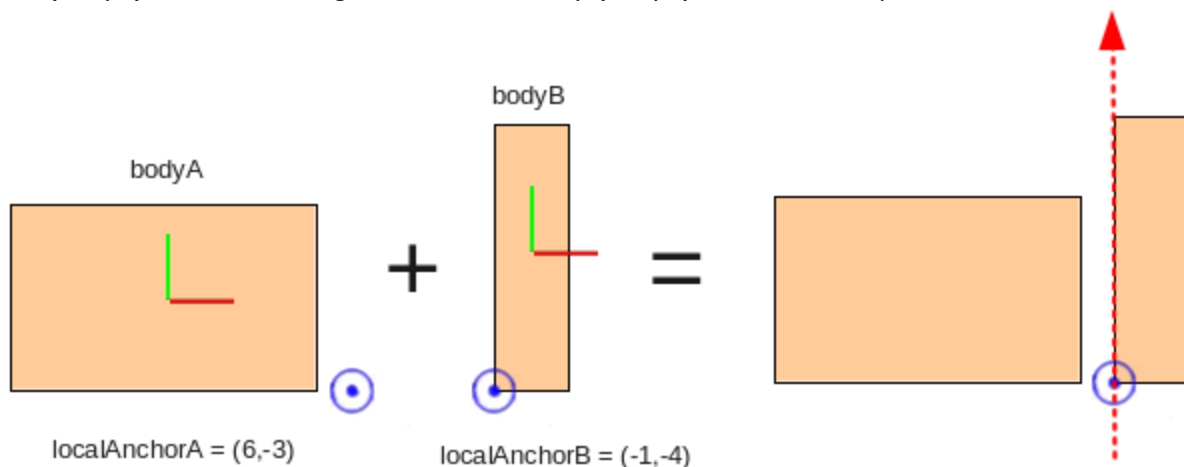
```
1 prismaticJointDef.localAxis1.Normalize();
```

Also note that since this only specifies a direction for sliding, the negative of this vector is an equivalent direction, eg. in the example above we could also have used (0,-1). For setting joint limits and motors this will become important.

Let's make the axis for our forklift joint (0,1) so that a positive movement in the axis raises the lift slider.

Local anchors

Now that we have established the direction along which the two bodies should move with respect to each other, we can specify points on each body that should stay on the axis line. These are given in local coordinates for each body, so you need to remember to look from the point of view of the body in question. Getting back to the forklift example, let's say we want to have the lift slider (bodyB) a little to the right of the main body (bodyA). We could use positions like this:

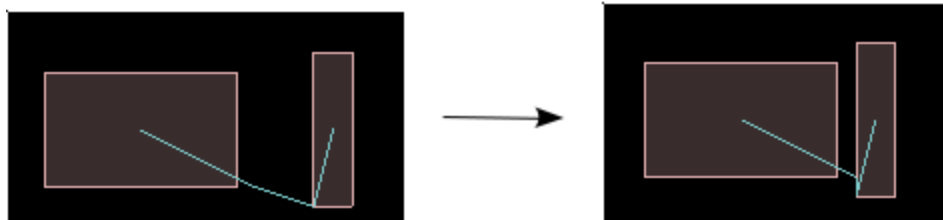


```
1 prismaticJointDef.localAnchorA.Set( 6,-3);//a little outside the bottom right corner  
2 prismaticJointDef.localAnchorB.Set(-1,-4);//bottom left corner
```

Now that we have some sensible values in place for a basic prismatic joint, we can create the joint itself (here I am assuming there is a class variable to store the joint pointer in to access it later).

```
1 m_joint = (b2PrismaticJoint*)m_world->CreateJoint( &prismaticJointDef );
```

When you run this you will see if you pause the simulation right at the beginning, the bodies are initially at their defined positions before the prismatic constraint takes effect in the first time step.

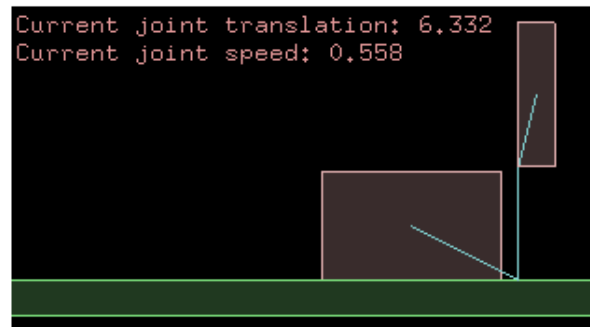
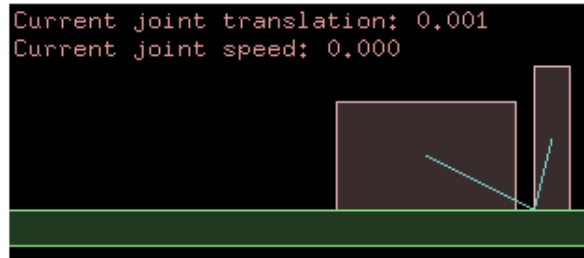


Since the axis of the joint is (0,1) and none of the bodies are rotated, we could actually have used any old value for the y-value of these anchor points and the bodies would still slide along the same line. However, when we want to set limits for the joint we'll need to be aware of where

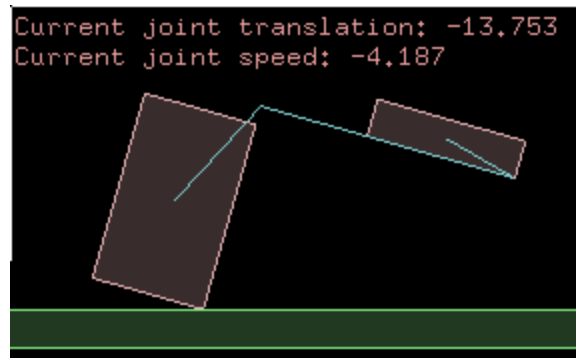
these points are on the line in order to get the limits in the right place. To make it easier to understand the values for joint limits in the sections below, let's add some output on the screen to show the current joint translation and speed:

```
1 //in Step()
2 m_debugDraw.DrawString(5, m_textLine, "Current joint translation: %.3f",
3 m_joint->GetJointTranslation());
4 m_textLine += 15;
5 m_debugDraw.DrawString(5, m_textLine, "Current joint speed: %.3f",
6 m_joint->GetJointSpeed());
7 m_textLine += 15;
```

Here are a couple of example positions. In the left one, the anchor points we specified are at the same location, so the translation of the joint is considered to be zero. In the right example, the joint anchor of the lift slider (bottom left corner of small body) has moved so that it is about the same height as the top of the large body, which as you'll recall is 6 units high, as confirmed by the on-screen display.



If you pick the bodies up and move them around you'll see how the axis is always the same relative to the two bodies, and the translation is only measured along this axis.

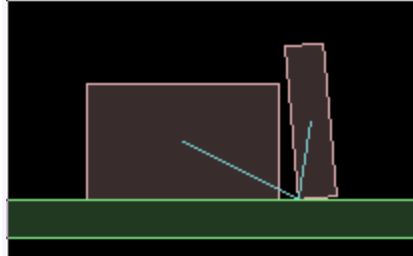


Reference angle

The angles of the bodies in this example started at the default angle of zero, and when we constrain them with the prismatic joint they cannot rotate to any other angle (relative to the other body). If we want to have a different angle between the bodies, we can set that with the reference angle property of the prismatic joint definition. The reference angle is given as the angle of bodyB, as seen from bodyA's point of view.

As an example, let's say we want the lift slider body to be tilted back a little to help keep the cargo from falling off the forks. We can give the joint a reference angle of 5 degrees, which will make the joint hold bodyB at 5 degrees **counter-clockwise** to bodyA.

```
1 prismaticJointDef.referenceAngle = 5 * DEGTO RAD;
```

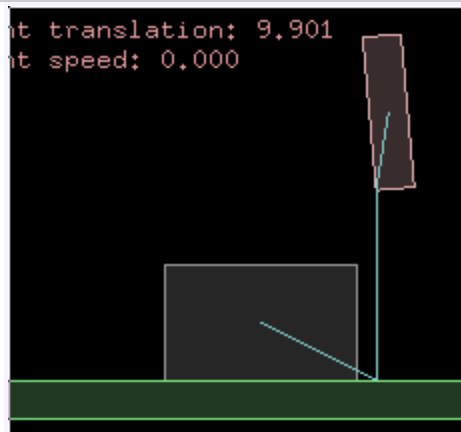


Note that since the local anchor point of each body is constrained to be on the axis line, the reference angle effectively causes bodyB to rotate around the local anchor point (at the bottom left corner).

Prismatic joint limits

With the settings made so far the two bodies are free to slide along the line indefinitely, but we can limit this range of motion by specifying limits for the joint. Joint limits define a lower and upper bound on the joint translation within which the bodies will be kept. This is where it comes in handy to have the joint translation displayed on the screen, so we can easily see where the limits should be set by moving the bodies around a bit. For this forklift example, we could set the lower limit at zero (this is where the lift slider body touches the ground) and the upper limit at hmm... about 10 looks fine.

```
1 prismaticJointDef.enableLimit = true;  
2 prismaticJointDef.lowerTranslation = 0;  
3 prismaticJointDef.upperTranslation = 10;
```



The default value for enableLimit is false. You can also get or set these limit properties for a prismatic joint after it has been created, by using these functions:

```

1 //alter joint limits
2 void EnableLimit(bool enabled);
3 void SetLimits( float lower, float upper );
4
5 //query joint limits
6 bool IsLimitEnabled();
7 float GetLowerLimit();
8 float GetUpperLimit();

```

Some things to keep in mind when using joint limits...

- The enableLimits settings affects both limits, so if you only want one limit you will need to set
- the other limit to a very high (for upper limit) or low (for lower limit) value so that it is never reached.
- Setting the limits to the same value is a handy way to 'clamp' the bodies at a given translation. This value
- can then be gradually changed to slide the bodies to a desired position while staying impervious to
- bumping around by other bodies, and without needing a joint motor.
- Very fast moving bodies can go past their limit for a few time steps until they are corrected.
- Checking if a joint is currently at one of its limits is pretty simple:
- `bool atLowerLimit = joint->GetJointTranslation() <= joint->GetLowerLimit();`
- `bool atUpperLimit = joint->GetJointTranslation() >= joint->GetUpperLimit();`

Prismatic joint motor

The default behaviour of a prismatic joint is to slide without any resistance. If you want to control the movement of the bodies you could either apply force or impulse to them as normal, or you can also set up a joint 'motor' which causes the joint to try to slide the bodies at a specific speed relative to each other. This is useful if you want to simulate powered movement such as a piston or elevator. Or a forklift.

The speed specified is only a target speed, meaning there is no guarantee that the joint will actually reach that speed. By giving the joint motor a maximum allowable force, you can control how quickly the joint is able to reach the target speed, or in some cases whether it can reach it at all. The behavior of the force acting on the joint is the same as in the [forces and impulses](#) topic. As an example, try setting the joint motor like this to move the lift slider body upwards.

```

1 prismaticJointDef.enableMotor = true;
2 prismaticJointDef.maxMotorForce = 500; //this is a powerful machine after all...
3 prismaticJointDef.motorSpeed = 5; //5 units per second in positive axis direction

```

The default value for enableMotor is false. Note that here we need to take into account the direction of the axis we specified at the beginning of this topic. A positive value for the motor

speed will move bodyB in the axis direction. Alternatively you could think of it as moving bodyA in the negative axis direction because the motor doesn't really move either body specifically, it just sets up a force between them to push or pull them in the appropriate direction, so you can use a prismatic joint motor to pull things together as well as push them apart.

You can also get or set these motor properties for a joint after it has been created, by using these functions:

```
1 //alter joint motor
2 void EnableMotor(bool enabled);
3 void SetMotorSpeed(float speed);
4 void SetMaxMotorForce(float force);
5
6 //query joint motor
7 bool IsMotorEnabled();
8 float GetMotorSpeed();
9 float GetMotorForce();
```

Some things to keep in mind when using prismatic joint motors...

- With a low max force setting, the joint can take some time to reach the desired speed. If you make
- the connected bodies heavier, you'll need to increase the max force setting if you want to keep
- the same rate of acceleration.
- The motor speed can be set to zero to make the joint try to stay still. With a low max force setting
- this acts like a brake, gradually slowing the bodies down. With a high max force it acts to stop the
- joint movement very quickly, and will require a large external force to move the joint, like a rusted
- up uh... sliding thing.

Example

You have probably noticed that prismatic joint concepts are very similar to the revolute joint, and the properties and functions are basically a linear version of their revolute counterparts. Since we've worked through a simple example while covering the main points above, I will leave it at that for this topic.

If you are interested in seeing a little more of prismatic joints in action, grab the [source code](#) and take a look at the source for the 'Joints - prismatic' test, which shows a second prismatic joint to simulate a laterally sliding cargo tray for the forklift, which you can control with the keyboard.

