Chapter 3

Physics Engine Basics

A physics engine is a program that provides an approximate simulation of a game world by creating a mathematical model for all the object interactions and collisions within the game. It accounts for gravity, elasticity, friction, and conservation of momentum between colliding objects so that the objects move in a believable way. For our game, we are going to be using an existing and very popular physics engine called Box2D.

The Box2D engine is a free, open source physics engine that was originally written in C++ by Erin Catto. It has been used in a lot of popular physics-based games, including *Crayon Physics Deluxe*, *Rolando*, and *Angry Birds*. The engine has since been ported to several other languages, including Java, ActionScript, C#, and JavaScript. We will be using a JavaScript port of Box2D known as Box2dWeb. You can find the latest source code and documentation for Box2dWeb at `http://code.google.com/p/box2dweb/`.

Before we start integrating the engine into our own game, let's go over some of the basic components of creating and simulating worlds using Box2D.

Box2D Fundamentals

Box2D uses a few basic objects to define and simulate the game world. The most important of these objects are as follows:

- *World*: The main Box2D object that contains all the world objects and simulates the game physics.
- *Body*: A rigid body that may consist of one or more shapes attached to the body via fixtures.
- *Shape*: A two-dimensional shape such as a circle or a polygon, which are the fundamental shapes used within Box2D.
- *Fixture*: Used to attach a shape to a body for collision detection. Fixtures hold additional, non-geometric data such as friction, collision and filters.
- *Joint*: Used to constrain two bodies together in different ways. For example, a revolute joint constrains two bodies to share a common point while they are free to rotate about the point.

When using Box2D in our game, we first need to define the game world. We then add bodies and their corresponding shapes using fixtures. Once this is done, we step through the world and let Box2D move the bodies around. Finally, we draw the bodies after each step. Most of the heavy lifting is done by the Box2D world object.

Now let's look at these steps in more detail as we use Box2D to create a simple world.

Setting Up Box2D

We will start with a simple HTML file just like in the previous chapters (`box2d.html`). The first thing we need to do is include a reference to the Box2dWeb library (`Box2dWeb-2.1.a.3.min.js`) in the head section of the HTML file (see Listing 3-1).

Listing 3-1. Basic HTML5 File for Box2D (box2d.html)

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-type" content="text/html; charset=utf-8">
        <title>Box2d Test</title>
        <script src="Box2dWeb-2.1.a.3.min.js" type="text/javascript" charset="utf-8"></script>
        <script src="box2d.js" type="text/javascript" charset="utf-8"></script>
    </head>
    <body onload="init();">
        <canvas id="canvas" width="640" height="480" style="border:1px solid black;">Your browser
does not support HTML5 Canvas</canvas>
    </body>
</html>
```

As you see in Listing 3-1, the `box2d.html` file consists of only a single `canvas` element that we will be drawing on. We refer to two JavaScript files: the Box2dWeb library file and a second file that we will use to store all our JavaScript code (`box2d.js`). Once the HTML file has loaded completely, it will call an `init()` function that we will use to initialize the Box2D world and start animating.

Referencing the Box2dWeb JavaScript file gives us access to the Box2D object in our JavaScript code. This object contains all the objects that we will need, including the world (`Box2D.Dynamics.b2World`) and the body (`Box2D.Dynamics.b2Body`).

It is convenient to define the commonly used objects as variables to save us some typing effort when we reference them. The first thing we will do in our JavaScript file (`box2d.js`) is to declare these variables (see Listing 3-2).

Listing 3-2. Defining Commonly Used Objects as Variables

```
// Declare all the commonly used objects as variables for convenience
var b2Vec2 = Box2D.Common.Math.b2Vec2;
var b2BodyDef = Box2D.Dynamics.b2BodyDef;
var b2Body = Box2D.Dynamics.b2Body;
var b2FixtureDef = Box2D.Dynamics.b2FixtureDef;
var b2Fixture = Box2D.Dynamics.b2Fixture;
var b2World = Box2D.Dynamics.b2World;
var b2PolygonShape = Box2D.Collision.Shapes.b2PolygonShape;
var b2CircleShape = Box2D.Collision.Shapes.b2CircleShape;
var b2DebugDraw = Box2D.Dynamics.b2DebugDraw;
var b2RevoluteJointDef = Box2D.Dynamics.Joints.b2RevoluteJointDef;
```

Once we define these variables as shortcuts, we can access the `Box2D.Dynamics.b2World` by using the `b2World` variable. Now, let's start defining our world.

Defining the World

The `Box2D.Dynamics.b2World` object is the heart of Box2D. It contains methods for adding and removing objects, methods for simulating physics in incremental steps, and even an option for drawing the world on a canvas. Before we can start using Box2D, we need to create the `b2World` object. We do this in an `init()` function that we create inside our JavaScript file (`box2d.js`), as shown in Listing 3-3.

Listing 3-3. Creating the `b2World` Object

```
var world;
var scale = 30; //30 pixels on our canvas correspond to 1 meter in the Box2d world
function init(){
    // Set up the Box2d world that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);
}
```

The `init()` function starts by defining `b2World` and passing the following two parameters to its constructor:

- `gravity`: Defined as a vector using a `b2Vec2` object, which takes two parameters, the x and y components. We set the world's gravity to be 9.8 meters per square second in the downward direction. The ability to set a custom gravity lets us simulate environments with different gravity fields, such as the moon or fantasy worlds with very low or very high gravity. We can also set `gravity` to 0 and use only the collision detection features of Box2D for games where we don't need gravity (space-based games or top-down view games like racing games).
- `allowSleep`: Used by `b2World` to decide whether or not to include objects that are at rest during its simulation calculations. Allowing objects that are at rest to be excluded from calculations reduces the number of unnecessary calculations and thus helps improve performance. Even if an object is asleep, it will wake up if a moving body collides with it.

One other thing that we do within our code is define a `scale` variable that we will use to convert between Box2D units (meters) and our game units (pixels).

■ **Note** Box2D use the metric system for all its calculations. It works best with objects that are between 0.1 meter to 10 meters large. Since we use pixels when drawing on our canvas, we will need to convert between pixels and meters. A commonly used scale is 30 pixels to 1 meter.

Now that we have a basic world, we need to start adding bodies to it. The first body we will create is a static floor at the bottom of our world.

Adding Our First Body: The Floor

Creating any body in Box2D involves the following steps:

1. Declare a body definition in a `b2BodyDef` object. The `b2BodyDef` object contains details such as the position of the body (x and y coordinates) and the type of body (static or dynamic). Static bodies are not affected by gravity and collisions with other bodies.

2. Declare a fixture definition in a `b2FixtureDef` object. This is used to attach a shape to the body. A fixture definition also contains additional information such as density, friction coefficient, and the coefficient of restitution for the attached shape.
3. Set the shape of the fixture definition. The two types of shapes that are used in Box2D are polygons (`b2PolygonShape`) and circles (`b2CircleShape`).
4. Pass the body definition object to the `createBody()` method of the world and get back a body object.
5. Pass the fixture definition to the `createFixture()` method of the body object and attach the shape to the body.

Now that we know these basic steps, we will create our first body inside the world: the floor. We will do this by creating a `createFloor()` method right below the `init()` function we created earlier. This is shown in Listing 3-4.

Listing 3-4. Creating the Floor

```
function createFloor(){
    //A body definition holds all the data needed to construct a rigid body.
    var bodyDef = new b2BodyDef;
    bodyDef.type = b2Body.b2_staticBody;
    bodyDef.position.x = 640/2/scale;
    bodyDef.position.y = 450/scale;

    // A fixture is used to attach a shape to a body for collision detection.
    // A fixture definition is used to create a fixture.
    var fixtureDef = new b2FixtureDef;
    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.2;

    fixtureDef.shape = new b2PolygonShape;
    fixtureDef.shape.SetAsBox(320/scale,10/scale); //640 pixels wide and 20 pixels tall

    var body = world.CreateBody(bodyDef);
    var fixture = body.CreateFixture(fixtureDef);
}
```

The first thing we do is define a `bodyDef` object. We set its type to be static (`b2Body.b2_staticBody`) since we want our floor to stay in the same place and not be affected by gravity or collisions with other bodies. We then set the position of the body near the bottom of our canvas (x = 320 pixels, y = 450 pixels) and use the scale variable to convert the pixels to meters for Box2D.

■ **Note** Unlike the canvas, where the position of rectangles is based on the top-left corner, the Box2D body position is based on the origin of the object. In the case of boxes created using `SetAsBox()`, this origin is at the center of the box.

The next thing we do is to define the fixture definition (`fixtureDef`). The fixture definition contains values like the density, the frictional coefficient, and the coefficient of restitution of its attached shape. The density is used to calculate the weight of the body, the frictional coefficient is used to make sure the body slides realistically, and the restitution is used to make the body bounce.

■ **Note** The higher the coefficient of restitution, the more "bouncy" the object becomes. Values close to 0 mean that the body will not bounce and will lose most of its momentum in a collision (called an inelastic collision). Values close to 1 mean that the body retains most of its momentum and will bounce back as fast as it came (called an elastic collision).

We then set the shape for the fixture as a `b2PolygonShape` object. The `b2PolygonShape` object has a helper method called `SetAsBox()` that sets the polygon as a box which is centered on the origin of the parent body. The `SetAsBox()` method takes the half-width and half-height (the extents) of the box as parameters. Again, we use the scale variable to define a box that is 640 pixels wide and 20 pixels high.

Finally we create the body by passing `bodyDef` to `world.CreateBody()` and create the fixture by passing the `fixtureDef` to `body.CreateFixture()`.

One other thing we need to do is call this newly created method from inside the `init()` function we declared earlier so that this body is created when the init() function is called. The `init()` function will now look like Listing 3-5.

Listing 3-5. Calling createFloor() from init()

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);

    createFloor();
}
```

Now that we have added our first body to the world, we need to learn how to draw the world so that we can see what we have created so far.

Drawing the World: Setting Up Debug Drawing

Box2D is primarily meant to be used only as an engine that handles physics calculations while we handle drawing all the objects in the world ourselves. However, the Box2D world object provides us with a simple `DrawDebugData()` method that we can use to draw the world on a given canvas.

The `DrawDebugData()` method draws a very simple representation of the bodies inside the world and is best used for helping us visualize the world while we are creating it.

Before we can use `DrawDebugData()`, we need to set up debug drawing by defining a `b2DebugDraw()` object and passing it to the `world.SetDebugDraw()` method. We do this in a `setupDebugDraw()` method that we will place below the `createFloor()` method inside `box2d.js` (see Listing 3-6).

Listing 3-6. Setting Up Debug Drawing

```
var context;
function setupDebugDraw(){
    context = document.getElementById('canvas').getContext('2d');

    var debugDraw = new b2DebugDraw();
    // Use this canvas context for drawing the debugging screen
    debugDraw.SetSprite(context);
    // Set the scale
    debugDraw.SetDrawScale(scale);
    // Fill boxes with an alpha transparency of 0.3
    debugDraw.SetFillAlpha(0.3);
    // Draw lines with a thickness of 1
    debugDraw.SetLineThickness(1.0);
    // Display all shapes and joints
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit | b2DebugDraw.e_jointBit);

    // Start using debug draw in our world
    world.SetDebugDraw(debugDraw);
}
```

We first define a handle to the canvas context. We then create a new `b2DebugDraw` object and set a few attributes using its `Set` methods:

- `SetSprite()`: Used to provide a canvas context for the drawing.
- `SetDrawScale()`: Used to set the scale to convert between Box2D units and pixels.
- `SetFillAlpha()` and `SetLineThickness()`: Used to set drawing styles.
- `SetFlags()`: Used to choose which Box2D entities to draw. We have selected flags for drawing all shapes and joints, and we use logical OR operators to combine the two flags. Some of the other entities we can ask Box2D to draw are the center of mass (`e_centerOfMassBit`) and axis-aligned bounding boxes (`e_aabbBit`).

Finally, we pass the `debugDraw` object to the `world.SetDebugDraw()` method. After creating the function, we need to call it from inside the `init()` function. The `init()` function will now look like Listing 3-7.

Listing 3-7. Calling setupDebugDraw() from init()

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);

    createFloor();

    setupDebugDraw();
}
```

Now that debug drawing is set up, we can use the `world.DrawDebugData()` method to draw the current state of our Box2D world onto the canvas.

Animating the World

Animating a world using Box2D involves the following steps that we repeat within an animation loop:

1. Tell Box2D to run the simulation for a small time step (typically 1/60th of a second). We do this by using the `world.Step()` function.
2. Draw all the objects in their new positions using either `world.DrawDebugData()` or our own drawing functions.
3. Clear any forces that we have applied using `world.ClearForces()`.

We can implement these steps in our own `animate()` function that we create inside `box2d.js` after `init()`, shown in Listing 3-8.

Listing 3-8. Setting Up a Box2D Animation Loop

```
var timeStep = 1/60;

//As per the Box2d manual, the suggested iteration count for Box2D is 8 for velocity and 3 for
position.
var velocityIterations = 8;
var positionIterations = 3;

function animate(){
    world.Step(timeStep,velocityIterations,positionIterations);
    world.ClearForces();

    world.DrawDebugData();

    setTimeout(animate, timeStep);

}
```

We first call `world.step()` and pass it three parameters: time step, velocity iterations, and position iterations.

Box2D uses a computational algorithm called an *integrator*. Integrators simulate the physics equations at discrete points of time. The time step is the amount of time we want Box2D to simulate. We set this to a value of 1/60th of a second.

In addition to the integrator, Box2D also uses a larger bit of code called a *constraint solver*. The constraint solver solves all the constraints in the simulation, one at a time. To get a good solution, we need to iterate over all constraints a number of times. There are two phases in the constraint solver: a velocity phase and a position phase. Each phase has its own iteration count, and we set these two values to 8 and 3, respectively.

■ **Note** Generally, physics engines for games work well with a time step at least as fast as 60Hz or 1/60 second. As per Erin Catto's original C++ *Box2D v2.2.0 User Manual* (available at `http://box2d.org/manual.pdf`), it is preferable to keep the time step constant and not vary it with frame rate, as a variable time step produces variable results, which makes it difficult to debug.

Also as per the Box2d C++ manual, the suggested iteration count for Box2D is 8 for velocity and 3 for position. You can tune these numbers to your liking, but keep in mind that this has a trade-off between speed and accuracy. Using fewer iterations increases performance, but accuracy suffers. Likewise, using more iterations decreases performance but improves the quality of your simulation.

After stepping through the simulation, we call `world.ClearForces()` to clear any forces that are applied to the bodies. We then call `world.DrawDebugData()` to draw the world on the canvas.

Finally, we use `setTimeout()` to call our animation loop again after the timeout for the next time step. We use `setTimeout()` for now because it is simpler for us to use the `Box2d.Step()` function with a constant frame rate. In the next chapter, we will look at how to use `requestAnimationFrame()` and a variable frame rate when integrating Box2D with our game.

Now that the animation loop is complete, we can see the world we have created so far by calling these new methods from the `init()` function. The updated `init()` function will now look like Listing 3-9.

Listing 3-9. Updated init() Function

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);

    createFloor();

    setupDebugDraw();
    animate();
}
```

When we open `box2d.html` in the browser, we should see our world with the floor drawn, as shown in Figure 3-1.

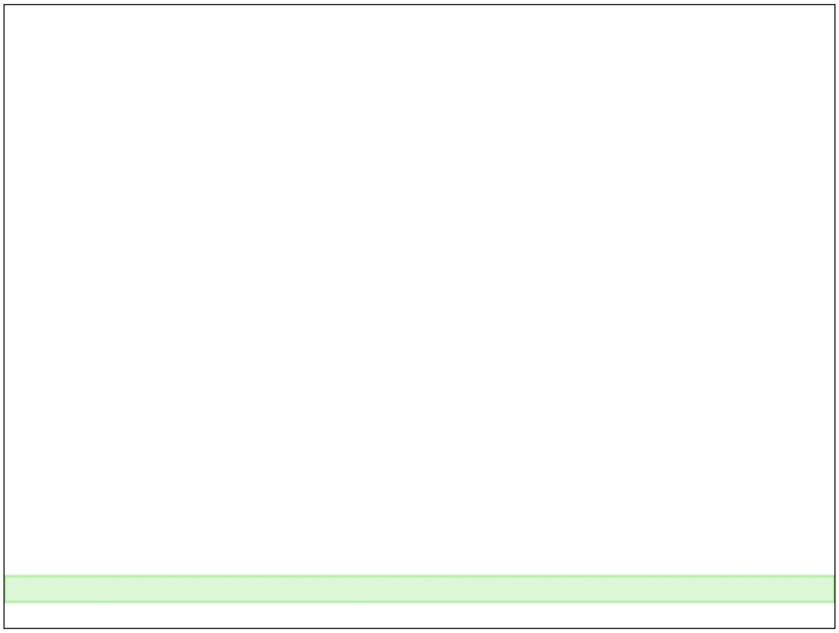


Figure 3-1. *Our first Box2D body: the floor*

This doesn't look like much yet. The floor is a static body that just stays floating at the bottom of the canvas. However, now that we have set up everything to create our basic world and display it on the screen, we can start adding some more Box2D elements to our world.

More Box2D Elements

Box2D allows us to add different types of elements to our world, including the following:

- Simple bodies that are rectangular, circular, or polygon shaped
- Complex bodies that combine multiple shapes
- Joints such as revolute joints that connect multiple bodies
- Contact listeners that allow us to handle collision events

We will now look at each of these elements in turn in more detail.

Creating a Rectangular Body

We can create a rectangular body just like we created our floor—by defining a `b2PolygonShape` and using its `SetAsBox()` method. We will do this within a new method called `createRectangularBody()` that we will add to `box2d.js` (see Listing 3-10).

Listing 3-10. Creating a Rectangular Body

```
function createRectangularBody(){
    var bodyDef = new b2BodyDef;
    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 40/scale;
    bodyDef.position.y = 100/scale;

    var fixtureDef = new b2FixtureDef;
    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.3;

    fixtureDef.shape = new b2PolygonShape;
    fixtureDef.shape.SetAsBox(30/scale,50/scale);

    var body = world.CreateBody(bodyDef);
    var fixture = body.CreateFixture(fixtureDef);
}
```

We create a body definition and place it near the top of the canvas at `x` = 40 pixels and `y` = 100 pixels. The one difference this time is that we define the body type as dynamic (`b2Body.b2_dynamicBody`). This means that the body will be affected by gravity and collisions. We then define the fixture with a polygon shape that is set as a box that is 60 pixels wide and 100 pixels tall. Finally, we add the body to our world.

We will need to add a call to `createRectangularBody()` inside the `init()` function so that it is called when the page loads. The `init()` function will now look like Listing 3-11.

Listing 3-11. Calling createRectangularBody() from init()

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);

    createFloor();
    // Create some bodies with simple shapes
    createRectangularBody();

    setupDebugDraw();
    animate();
}
```

When we run the code in the browser, we should see the new body that we just created, as shown in Figure 3-2.

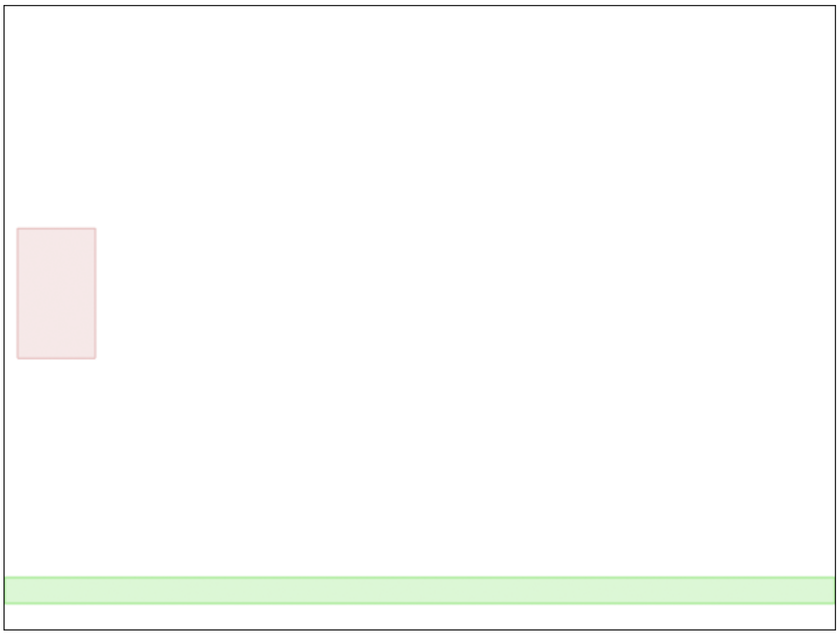


Figure 3-2. *Our first dynamic body: a bouncing rectangle*

Since this body is dynamic, it will fall downward because of gravity until it hits the floor, and then it will bounce off the floor. The body rises to a lower height after each bounce until it finally settles down on the floor. If we want, we can change the coefficient of restitution to decide how bouncy the object is.

■ **Note** Once the body comes to rest, Box2D changes the color of the body and makes it darker. This is how Box2D tells us that the object is considered asleep. Box2D will wake a body up if another body collides with it.

Creating a Circular Body

The next body we will create is a simple circular body. We can define a circular shape by setting the shape property to a `b2CircleShape` object. We will do this within a new method called `createCircularBody()` that we will add to `box2d.js`, as shown in Listing 3-12.

Listing 3-12. Creating a Circular Shape

```
function createCircularBody(){
    var bodyDef = new b2BodyDef;
    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 130/scale;
    bodyDef.position.y = 100/scale;

    var fixtureDef = new b2FixtureDef;
    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.7;

    fixtureDef.shape = new b2CircleShape(30/scale);

    var body = world.CreateBody(bodyDef);
    var fixture = body.CreateFixture(fixtureDef);
}
```

The `b2CircleShape` constructor takes one parameter, the radius of the circle. The rest of the code, defining a body, defining the fixture, and creating the body, remains very similar to the code for the rectangular body.

One change we have made is to increase the restitution value to 0.7, which is much higher than the value we used for our previous rectangular body. We need to call `createCircularBody()` from inside the `init()` function. The `init()` function will now look like Listing 3-13.

Listing 3-13. Calling createCircularBody() from init()

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);

    createFloor();
    // Create some bodies with simple shapes
    createRectangularBody();
    createCircularBody();

    setupDebugDraw();
    animate();
}
```

Once we do this and run the code, we should see the new circular body that we just created (as shown in Figure 3-3).

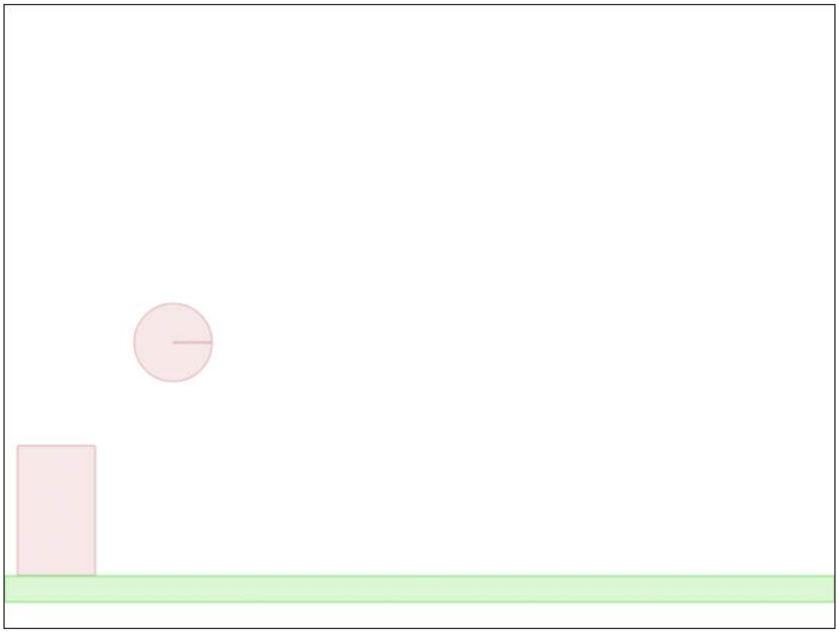


Figure 3-3. *A bouncier circular body*

You will notice that the circular body bounces much higher than the rectangular one, and takes a longer time to come to rest. This is because of the larger coefficient of restitution. When creating your own game, you can play around with these values until they feel right for your game.

Creating a Polygon-Shaped Body

The last simple shape we will create is the polygon. Box2D allows us to create any polygon we want by defining the coordinates of each of the points. The only restriction is that polygons need to be convex polygons.

To create a polygon, we first need to create an array of `b2Vec2` objects with the coordinates of each of its points, and then we need to pass the array to the `shape.SetAsArray()` method. We will do this within a new method called `createSimplePolygonBody()` that we will add to `box2d.js` (see Listing 3-14).

Listing 3-14. Defining a Polygon Shape with Points

```
function createSimplePolygonBody(){
    var bodyDef = new b2BodyDef;
    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 230/scale;
    bodyDef.position.y = 50/scale;

    var fixtureDef = new b2FixtureDef;
    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.2;
    fixtureDef.shape = new b2PolygonShape;
    // Create an array of b2Vec2 points in clockwise direction
    var points = [
        new b2Vec2(0,0),
        new b2Vec2(40/scale,50/scale),
        new b2Vec2(50/scale,100/scale),
        new b2Vec2(-50/scale,100/scale),
        new b2Vec2(-40/scale,50/scale),

    ];
    // Use SetAsArray to define the shape using the points array
    fixtureDef.shape.SetAsArray(points,points.length);

    var body = world.CreateBody(bodyDef);

    var fixture = body.CreateFixture(fixtureDef);
}
```

We defined a `points` array that contains the coordinates for each of the polygon points inside `b2Vec2` objects. The following are a few things to note:

- All the coordinates are relative to the body origin. The first point (0,0) starts at the origin of the body and will be placed at the body position (230,50).
- We do not need to close out the polygon. Box2D will take care of this for us.
- All points must be defined in a clockwise direction.

■ **Tip** If we define the coordinates in the counter-clockwise direction, Box2D will not be able to handle collisions correctly. If you find objects passing through each other, check to see whether you have defined points in the clockwise direction.

We then call the `SetAsArray()` method and pass it two parameters: the `points` array and the number of points. The rest of the code remains the same as it was for the previous shapes we covered.

Now we need to call `createSimplePolygonBody()` from the `init()` function. The `init()` function will now look like Listing 3-15.

Listing 3-15. Calling createSimplePolygonBody() from init()

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);
    createFloor();
    // Create some bodies with simple shapes
    createRectangularBody();
    createCircularBody();
    createSimplePolygonBody();

    setupDebugDraw();
    animate();
}
```

If we run this code, we should see our new polygon-shaped body (see Figure 3-4).

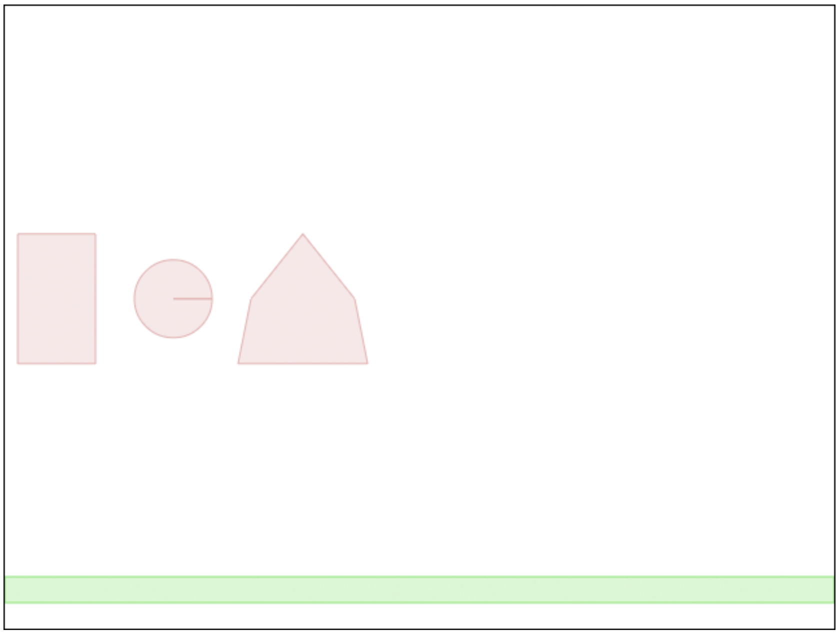


Figure 3-4. *A polygon-shaped body*

We now have created three simple bodies, with different shapes and properties. These simple shapes are usually enough to model a wide array of objects within our games (fruits, tires, crates, and so forth). Sometimes, however, these shapes are not enough. There are times when we need to create more complex objects that combine more than one shape.

Creating Complex Bodies with Multiple Shapes

So far we have been creating simple bodies with a single shape. However, as previously mentioned, Box2D lets us create bodies that contain multiple shapes.

To create a complex shape, all we need to do is attach multiple fixtures (each with its own shape) to the same body. Let's try to combine two of the shapes we just learned about into a single body: a circle and a polygon. We will do this within a new method called `createComplexPolygonBody()` that we will add to `box2d.js` (see Listing 3-16).

Listing 3-16. Creating a Body with Two Shapes

```
function createComplexBody(){
    var bodyDef = new b2BodyDef;
    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 350/scale;
    bodyDef.position.y = 50/scale;
    var body = world.CreateBody(bodyDef);

    //Create first fixture and attach a circular shape to the body
    var fixtureDef = new b2FixtureDef;
    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.7;
    fixtureDef.shape = new b2CircleShape(40/scale);
    body.CreateFixture(fixtureDef);

    // Create second fixture and attach a polygon shape to the body
    fixtureDef.shape = new b2PolygonShape;
    var points = [
        new b2Vec2(0,0),
        new b2Vec2(40/scale,50/scale),
        new b2Vec2(50/scale,100/scale),
        new b2Vec2(-50/scale,100/scale),
        new b2Vec2(-40/scale,50/scale),
    ];
    fixtureDef.shape.SetAsArray(points,points.length);
    body.CreateFixture(fixtureDef);
}
```

We first create a body, and then two different fixtures—the first for a circular shape and the second for a polygon shape. We then attach both these fixtures to the body using the `CreateFixture()` method. Box2D will automatically take care of creating a single rigid body that includes both these shapes.

Now that we have created `createComplexBody()`, we need to call it from inside the `init()` function. The `init()` function will now look like Listing 3-17.

Listing 3-17. Calling createComplexBody() from init()

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);

    createFloor();

    // Create some bodies with simple shapes
    createRectangularBody();
    createCircularBody();
    createSimplePolygonBody();
    // Create a body combining two shapes
    createComplexBody();

    setupDebugDraw();
    animate();
}
```

When we run this code, we should see our new complex body, as shown in Figure 3-5.

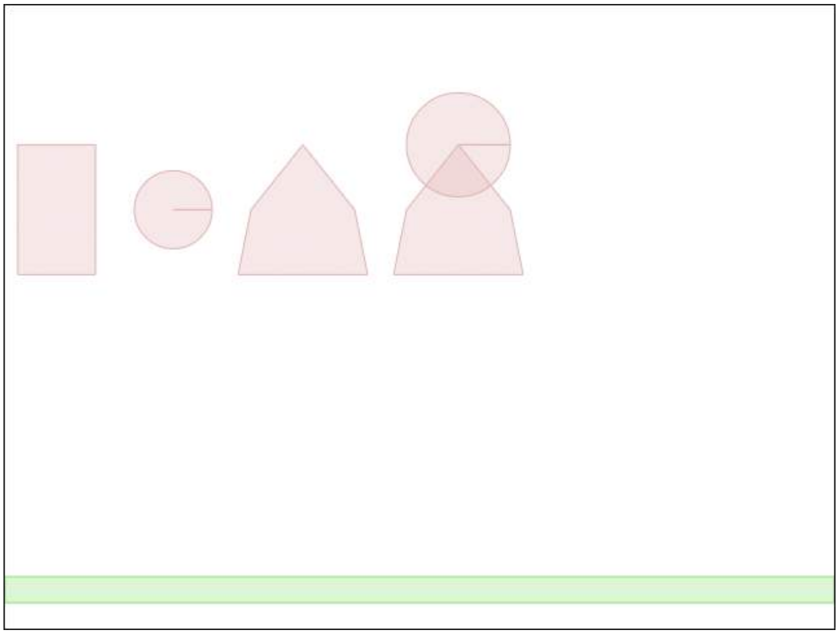


Figure 3-5. *A complex body with two shapes*

You will notice that the two shapes behave as one single unit. This is because Box2D treats these multiple shapes as a single rigid body. This ability to combine shapes allows us to emulate any kind of object we want, such as trees and tables.

It also allows us to get around the limitations on creating concave polygon shapes, since any concave polygon can be broken into multiple convex polygons.

Connecting Bodies with Joints

Now that we know how to make different types of bodies in Box2D, we will take a brief look at creating joints.

Joints are used to constrain bodies to the world or to each other. Box2D supports many different types of joints, including pulley, gear, distance, revolute, and weld joints.

Some of these joints restrict motion (for example, the distance joint and the weld joint), while others allow for interesting types of movement (for example, the pulley joint and the revolute joint). Some joints even provide motors that can be used to drive the joint at a specified speed. We will take a look at one of the simpler joints that Box2D offers: the revolute joint.

The revolute joint forces two bodies to share a common anchor point, often called a hinge point. What this means is that the bodies are attached to each other at this point, and can rotate about that point.

We can create a revolute joint by defining a `b2RevoluteJointDef` object and then passing it to the `world.CreateJoint()` method. This is illustrated in the `createRevoluteJoint()` method that we add to `box2d.js` (see Listing 3-18).

Listing 3-18. Creating a Revolute Joint

```
function createRevoluteJoint(){
    //Define the first body
    var bodyDef1 = new b2BodyDef;
    bodyDef1.type = b2Body.b2_dynamicBody;
    bodyDef1.position.x = 480/scale;
    bodyDef1.position.y = 50/scale;
    var body1 = world.CreateBody(bodyDef1);

    //Create first fixture and attach a rectangular shape to the body
    var fixtureDef1 = new b2FixtureDef;
    fixtureDef1.density = 1.0;
    fixtureDef1.friction = 0.5;
    fixtureDef1.restitution = 0.5;
    fixtureDef1.shape = new b2PolygonShape;
    fixtureDef1.shape.SetAsBox(50/scale,10/scale);

    body1.CreateFixture(fixtureDef1);

    // Define the second body
    var bodyDef2 = new b2BodyDef;
    bodyDef2.type = b2Body.b2_dynamicBody;
    bodyDef2.position.x = 470/scale;
    bodyDef2.position.y = 50/scale;
    var body2 = world.CreateBody(bodyDef2);

    //Create second fixture and attach a polygon shape to the body
    var fixtureDef2 = new b2FixtureDef;
    fixtureDef2.density = 1.0;
    fixtureDef2.friction = 0.5;
    fixtureDef2.restitution = 0.5;
    fixtureDef2.shape = new b2PolygonShape;
    var points = [
        new b2Vec2(0,0),
        new b2Vec2(40/scale,50/scale),
        new b2Vec2(50/scale,100/scale),
        new b2Vec2(-50/scale,100/scale),
        new b2Vec2(-40/scale,50/scale),
    ];
    fixtureDef2.shape.SetAsArray(points,points.length);
    body2.CreateFixture(fixtureDef2);

    // Create a joint between body1 and body2
    var jointDef = new b2RevoluteJointDef;
    var jointCenter = new b2Vec2(470/scale,50/scale);
    jointDef.Initialize(body1, body2, jointCenter);
    world.CreateJoint(jointDef);
}
```

In this code we first define two bodies, a rectangle (body1) and a polygon (body2), that are positioned on top of each other, and then we add them to the world.

We then create a b2RevolutionJointDef object and initialize it by passing three parameters to the Initialize() method: the two bodies (body1 and body2), and the joint center, which is the point around which the joints rotate.

Finally, we call world.CreateJoint() to add the joint to the world.

We need to call createRevoluteJoint() from our init() function. The init() function will now look like Listing 3-19.

Listing 3-19. Calling createRevoluteJoint() from init()

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);

    createFloor();

    // Create some bodies with simple shapes
    createRectangularBody();
    createCircularBody();
    createSimplePolygonBody();

    // Create a body combining two shapes
    createComplexBody();

    // Join two bodies using a revolute joint
    createRevoluteJoint();

    setupDebugDraw();
    animate();
}
```

When we run our code, we should see our revolute joint in action. You can see this in Figure 3-6.

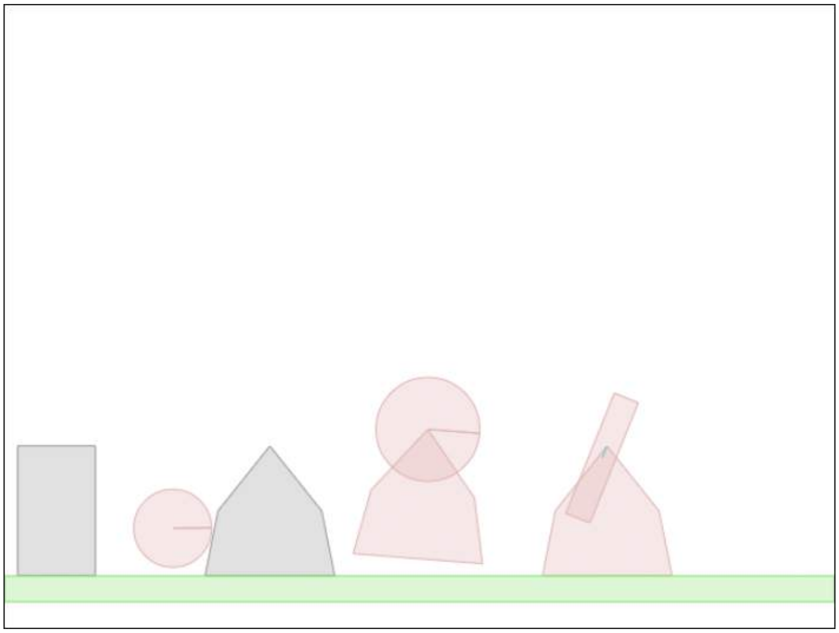


Figure 3-6. *A revolute joint in action*

As you can see, the rectangular body rotates about its anchor point, almost like a windmill blade. This is very different from the complex body we created earlier, where the shapes acted like a single body.

Each of the joints in Box2D can be combined in different ways to create interesting motions and effects, such as pulleys, ragdolls, and pendulums. You can read more about these other types of joints in the Box2D reference API, which you can find at `http://www.box2dflash.org/docs/2.1a/reference/`. Note that this is for the Flash version of Box2D that our JavaScript version is based on. We can still refer to the method signatures and documentation in this Flash version when developing for the JavaScript version because the JavaScript version of Box2D was developed by directly converting the Flash version, and the method signatures remain the same across the two.

Tracking Collisions and Damage

One thing that you may have noticed in the previous few examples is that some of the bodies were colliding against each other and bouncing back and forth. It would be nice to be able to keep track of these collisions and the amount of impact they cause, and simulate a body getting damaged.

Before we can track the damage to an object, we need to be able to associate a life or health with it. Box2D provides us with methods that allow us to set custom properties for any body, fixture, or joint. We can assign any JavaScript object as a custom property for a body by calling its `SetUserData()` method, and retrieve the property later by calling its `GetUserData()` method.

Let's create another body that will have its own health unlike any of the previous bodies. We will do this inside a method called `createSpecialBody()` that we will add to `box2d.js` (see Listing 3-20).

Listing 3-20. Creating a Special Body with Its Own Properties

```
var specialBody;
function createSpecialBody(){
    var bodyDef = new b2BodyDef;
    bodyDef.type = b2Body.b2_dynamicBody;
    bodyDef.position.x = 450/scale;
    bodyDef.position.y = 0/scale;

    specialBody = world.CreateBody(bodyDef);
    specialBody.SetUserData({name:"special",life:250})

    //Create a fixture to attach a circular shape to the body
    var fixtureDef = new b2FixtureDef;
    fixtureDef.density = 1.0;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.5;

    fixtureDef.shape = new b2CircleShape(30/scale);

    var fixture = specialBody.CreateFixture(fixtureDef);
}
```

The code for creating this body is similar to the code for creating a circular body that we looked at earlier. The only difference is that once we create the body, we call its `SetUserData()` method and pass it an object parameter with two custom properties, `name` and `life`.

We can add as many properties as we like to this object. Also, note that we saved a reference to the body in a variable called `specialBody` that we defined outside the function. This way, we can refer to this body outside of the function.

If we call `createSpecialBody()` from the `init()` function, we won't see anything exceptional—just another bouncing circle. We still want to be able to track collisions happening to this body. This is where contact listeners come in.

Contact Listeners

Box2D provides us with objects called contact listeners that let us define event handlers for several contact-related events. To do this, we must first define a `b2ContactListener` object and override one or more of the events we want to monitor. The `b2ContactListener` has four events we can use based on what we need:

- `BeginContact()`: Called when two fixtures begin to touch.
- `EndContact()`: Called when two fixtures cease to touch.
- `PostSolve()`: Lets us inspect a contact after the solver is finished. This is useful for inspecting impulses.
- `PreSolve()`: Lets us inspect a contact before it goes to the solver.

Once we override the methods that we need, we need to pass the contact listener to the `world.SetContactListener()` method. Since we want to track the damage a collision causes, we will listen to the `PostSolve()` event, which provides us with the impulse transferred during a collision (see Listing 3-21).

Listing 3-21. Implementing a Contact Listener

```
function listenForContact(){
    var listener = new Box2D.Dynamics.b2ContactListener;
    listener.PostSolve = function(contact,impulse){
        var body1 = contact.GetFixtureA().GetBody();
        var body2 = contact.GetFixtureB().GetBody();

        // If either of the bodies is the special body, reduce its life
        if (body1 == specialBody || body2 == specialBody){
            var impulseAlongNormal = impulse.normalImpulses[0];
            specialBody.GetUserData().life -= impulseAlongNormal;
            console.log("The special body was in a collision with impulse", impulseAlongNormal,"and
its life has now become ",specialBody.GetUserData().life);
        }
    };
    world.SetContactListener(listener);
}
```

As you can see, we create a `b2ContactListener` object and override its `PostSolve()` method with our own handler. The `PostSolve()` method provides us with two parameters: `contact`, which contains details of the fixtures that were involved in the collision, and `impulse`, which contains the normal and tangential impulse during the collision.

Within `PostSolve()`, we first extract the two bodies involved in the collision and check to see if our special body is one of them. If it is, we extract the impulse along the normal between the two bodies, and subtract life points from the body. We also log this event to the console so we can track each collision.

Obviously, this is a rather simplistic way of handling object damage, but it does what we need it to do. The greater the impulse in a collision, and the higher the number of collisions, the faster the body loses health.

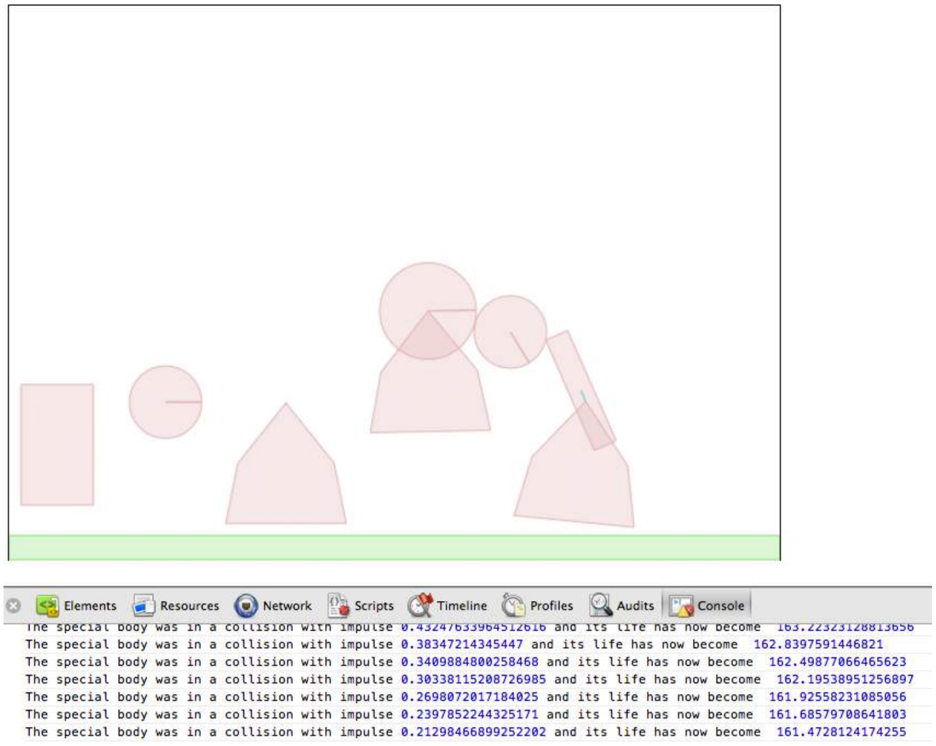
■ **Note** The `PostSolve()` method is called for every collision that takes place in the Box2D world, no matter how small. It is even called when an object is rolling on another. Be aware that this method will be called a lot.

Next we call both `createSimpleBody()` and `listenForContact()` from `init()`. The `init()` function will now look like Listing 3-22.

Listing 3-22. Calling createSpecialBody() and listenForContact() from init()

```
function init(){
    // Set up the box2d World that will do most of the physics calculation
    var gravity = new b2Vec2(0,9.8); //declare gravity as 9.8 m/s^2 downward
    var allowSleep = true; //Allow objects that are at rest to fall asleep and be excluded from
calculations
    world = new b2World(gravity,allowSleep);

    createFloor();

    // Create some bodies with simple shapes
    createRectangularBody();
    createCircularBody();
    createSimplePolygonBody();
    // Create a body combining two shapes
    createComplexBody();

    // Join two bodies using a revolute joint
    createRevoluteJoint();

    // Create a body with special user data
    createSpecialBody();

    // Create contact listeners and track events
    listenForContact();

    setupDebugDraw();
    animate();
}
```

If we run our code now, we should see the circle bouncing about, with a message in the browser console after each collision telling us how much the body's health has dropped, as shown in Figure 3-7.



```
The special body was in a collision with impulse 0.43247633964512616 and its life has now become  163.22323128813656
The special body was in a collision with impulse 0.38347214345447 and its life has now become  162.8397591446821
The special body was in a collision with impulse 0.3409884800258468 and its life has now become  162.49877066465623
The special body was in a collision with impulse 0.30338115208726985 and its life has now become  162.19538951256897
The special body was in a collision with impulse 0.2698072017184025 and its life has now become  161.92558231085056
The special body was in a collision with impulse 0.2397852244325171 and its life has now become  161.68579708641803
The special body was in a collision with impulse 0.21298466899252202 and its life has now become  161.4728124174255
```

Figure 3-7. *Watching collisions with Contact Listeners*

It is nice to be able to track the life of our special body, but it would be nicer if we could do something when it runs out of life.

Now that we have access to `specialBody` and the `life` property, we can check after every iteration to see if the body life has reached 0 and, if so, remove it from the world using the `world.DestroyBody()` method. The easiest place to do this check is in the `animate()` method. The `animate()` function will now look like Listing 3-23.

Listing 3-23. Destroying the Body

```
function animate(){
    world.Step(timeStep,velocityIterations,positionIterations);
    world.ClearForces();

    world.DrawDebugData();

    //Kill Special Body if Dead
    if (specialBody && specialBody.GetUserData().life<=0){
        world.DestroyBody(specialBody);
        specialBody = undefined;
        console.log("The special body was destroyed");
    }

    setTimeout(animate, timeStep);
}
```

Once we finish calling `world.Step()` and drawing the world, we check to see whether `specialBody` is still defined and whether its life has reached 0. Once the life does reach 0, we remove the body from the world using `DestroyBody()` and then set `specialBody` to `undefined`.

This time when we run the code, the special body bounces around with its life dropping until it finally disappears. A message appears in the console telling us that the body was destroyed.

■ **Note** We can track all the bodies and elements in a game using a similar principle by iterating through an array of objects. The point where we destroy a body is the perfect place for us to add explosion sounds or visual effects in a game and maybe update the score.

Drawing Our Own Characters

We have played with a lot of Box2D features so far. However, we have only been drawing using the default `DrawDebugData()` method. While this method is fine when testing code, we can't really write an amazing game looking like this. We need to know how to draw our own characters using all the drawing methods we learned in the first chapter.

Every `b2Body` object has two methods, `GetPosition()` and `GetAngle()`, that provide us with the coordinates and rotation of the body inside the Box2D world. Using the scale variable we defined in this chapter and the canvas `translate()` and `rotate()` methods we explored in Chapter 1, we can draw our characters or sprites on the canvas at the location that Box2D calculates for us.

To illustrate this, we can draw the special body that we have been playing with so far inside a `drawSpecialBody()` method that we will add to `box2d.js` (see Listing 3-24).

Listing 3-24. Drawing Our Own Character

```
function drawSpecialBody(){
    // Get body position and angle
    var position = specialBody.GetPosition();
    var angle = specialBody.GetAngle();

    // Translate and rotate axis to body position and angle
    context.translate(position.x*scale,position.y*scale);
    context.rotate(angle);

    // Draw a filled circular face
    context.fillStyle = "rgb(200,150,250);";
    context.beginPath();
    context.arc(0,0,30,0,2*Math.PI,false);
    context.fill();

    // Draw two rectangular eyes
    context.fillStyle = "rgb(255,255,255);";
    context.fillRect(-15,-15,10,5);
    context.fillRect(5,-15,10,5);

    // Draw an upward or downward arc for a smile depending on life
    context.strokeStyle = "rgb(255,255,255);";
    context.beginPath();
    if (specialBody.GetUserData().life>100){
        context.arc(0,0,10,Math.PI,2*Math.PI,true);
    } else {
        context.arc(0,10,10,Math.PI,2*Math.PI,false);
    }
    context.stroke();

    // Translate and rotate axis back to original position and angle
    context.rotate(-angle);
    context.translate(-position.x*scale,-position.y*scale);
}
```

We start by translating the canvas to the body's position and rotating the canvas to the body's angle. This is very similar to the code we looked at in Chapter 1.

We then draw a filled circle for the face, two rectangular eyes, and a smile using an arc. Just for fun, when the body life goes below 100, we change the smile to a sad face.

Finally, we undo the rotation and translation.

Before we can see this method in action, we will need to call it from inside `animate()`. The finished `animate()` method will now look like Listing 3-25.

Listing 3-25. The Finished animate() Method

```
function animate(){
    world.Step(timeStep,velocityIterations,positionIterations);
    world.ClearForces();

    world.DrawDebugData();

    // Custom Drawing
    if (specialBody){
        drawSpecialBody();
    }

    //Kill Special Body if Dead
    if (specialBody && specialBody.GetUserData().life<=0){
        world.DestroyBody(specialBody);
        specialBody = undefined;
        console.log("The special body was destroyed");
    }

    setTimeout(animate, timeStep);
}
```

What we have done here is check whether `specialBody` is still defined and call `drawSpecialBody()` if it is. Once the body dies, `specialBody` will become undefined and we will stop trying to draw it. You will notice that we draw after `DrawDebugData()` has completed, so we end up drawing on top of the debug drawing.

When we run this finished code, we see our new version of `specialBody` with a smiley face that becomes sad after a while before finally disappearing (see Figure 3-8).

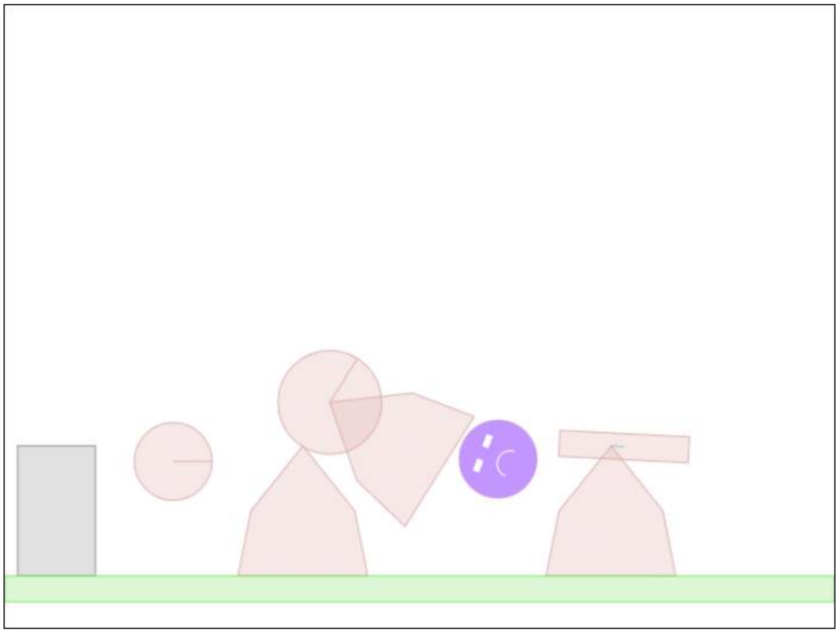


Figure 3-8. *Drawing our own character*

We have just animated our own character using the Box2D engine. This may not seem like much, but we now have all the building blocks that we need to build games using Box2D.

When you create your own game, you won't just be playing with boxes and circles. You will still use simple shapes that are similar in appearance to your game elements so that they seem to move realistically. However, you will be drawing all the characters yourself instead of using debug drawing.

Summary

In this chapter we took a crash course on the Box2D engine. We created a world in Box2D and drew different kinds of bodies within it. We made simple circular and rectangular shapes, polygons, and complex bodies that combined multiple shapes, and we used joints to combine shapes.

We animated the world realistically by letting Box2D handle the physics computations and drawing the world using `DrawDebugData()`. We used contact listeners to track collisions and slowly damage and destroy objects within the world. Finally, we drew our own character that was moved by Box2D.

We covered most of the elements of Box2D that we will be using in our game. If you would like to dive deeper into the Box2D API, you can look at the API reference available at `http://www.box2dflash.org/docs/`. You can also read the Box2D guide available at the same site.

In the next chapter, we will combine everything that we have learned so far to integrate Box2D into our game. We will create a framework to handle creation of our game entities inside Box2D. We will then use images and sprites to draw our characters over the parallax scrolling backgrounds that we built in Chapter 2. After that, we will spend some time polishing up our game by adding sound effects, and then wire everything together to create a finished, physics-based puzzle game.