Chapter 4

Integrating The Physics Engine

In Chapter 2, we developed the basic framework for our game, Froot Wars, and in Chapter 3, we looked at how to simulate a game world in Box2D. Now it is time to put together all the pieces to complete our game.

In this chapter, we will continue where we left off at the end of Chapter 2. We will add entities to our levels, use Box2D to simulate these entities, and then animate these entities within the game. We will use these entities to create a couple of working levels, and we will add mouse interactivity so that we can play the game. Once we have a working game, we will add sounds, background music, and a few other finishing touches to wrap up our game.

Now let's get started. We will be using the code from Chapter 2 as our starting point.

Defining Entities

So far, our game levels contain data for the background and foreground images and an empty array for entities. This entities array will eventually contain all the entities within our game: the heroes, the villains, the ground, and the blocks used to create the environment. We will then use this array to ask Box2D to create the corresponding Box2D shapes.

Typical entities will look like the examples shown in Listing 4-1.

Listing 4-1. Typical Entities

```
{type:"block", name:"wood", x:520,y:375,angle:90},
{type:"villain", name:"burger",x:520,y:200,calories:590},
```

The `type` property can contain values like `"hero"`, `"villain"`, `"ground"`, and `"block"`. We will use this property to decide how to handle an entity during creation and drawing operations.

The `x`, `y`, and `angle` properties are used to set the starting position and orientation of the entities. We can also store other custom properties (such as `calories`, which is the number of points scored for destroying a villain) within entities.

The `name` property tells us which sprite to use to draw the entity. All the images that we will use for the entities are stored in the `images/entities` folder.

The `name` property will also be used to refer to entity definitions. These definitions will include fixture data such as density and restitution, health data for destructible objects, and, in the case of heroes and villains, even details on the shape. Typical entity definitions will look like the examples shown in Listing 4-2.

Listing 4-2. Typical Entity Definitions

```
"burger":{
    shape:"circle",
    fullHealth:40,
    radius:25,
```

```
    density:1,
    friction:0.5,
    restitution:0.4,
},
"wood":{
    fullHealth:500,
    density:0.7,
    friction:0.4,
    restitution:0.4,
},
```

Now that we have decided how we will be storing the entities, we also need a way to create them. We will start by creating an `entities` object in `game.js` that will handle all entity-related operations in our game. This object will contain all the entity definitions as well as the methods for creating and drawing entities (see Listing 4-3).

Listing 4-3. The entities Object with Definitions for Entities

```
var entities = {
    definitions:{
        "glass":{
            fullHealth:100,
            density:2.4,
            friction:0.4,
            restitution:0.15,
        },
        "wood":{
            fullHealth:500,
            density:0.7,
            friction:0.4,
            restitution:0.4,
        },
        "dirt":{
            density:3.0,
            friction:1.5,
            restitution:0.2,
        },
        "burger":{
            shape:"circle",
            fullHealth:40,
            radius:25,
            density:1,
            friction:0.5,
            restitution:0.4,
        },
        "sodacan":{
            shape:"rectangle",
            fullHealth:80,
            width:40,
            height:60,
            density:1,
```

```
            friction:0.5,
            restitution:0.7,
        },
        "fries":{
            shape:"rectangle",
            fullHealth:50,
            width:40,
            height:50,
            density:1,
            friction:0.5,
            restitution:0.6,
        },
        "apple":{
            shape:"circle",
            radius:25,
            density:1.5,
            friction:0.5,
            restitution:0.4,
        },
        "orange":{
            shape:"circle",
            radius:25,
            density:1.5,
            friction:0.5,
            restitution:0.4,
        },
        "strawberry":{
            shape:"circle",
            radius:15,
            density:2.0,
            friction:0.5,
            restitution:0.4,
        }
    },
    // take the entity, create a Box2D body, and add it to the world
    create:function(entity){

    },
    // take the entity, its position, and its angle and draw it on the game canvas
    draw:function(entity,position,angle){

    }
}
```

The `entities` object contains an array with definitions for all the material types (glass, wood, and dirt) and definitions for all the heroes and villains that we will have in the game (orange, apple, and burger).

The values for some of these properties (such as `size`, `restitution`, and `fullHealth`) were decided based on feel, by constantly tweaking them in an effort to make the game as much fun as possible. The correct values for these properties will vary with each game you make.

We also have placeholders for the `create()` and `draw()` functions that we need to implement. However, before we can implement these, we need to add Box2D to our code.

Adding Box2D

The first thing we need to do is add a reference to `Box2dWeb-2.1.a.3.min.js` in the `<head>` section of `index.html` before the reference to `game.js`. The `<head>` section of the file will now look like Listing 4-4.

Listing 4-4. Adding Box2D to the index.html <head> Section

```
 <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Froot Wars</title>
    <script src="js/jquery.min.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/Box2dWeb-2.1.a.3.min.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/game.js" type="text/javascript" charset="utf-8"></script>
    <link rel="stylesheet" href="styles.css" type="text/css" media="screen" charset="utf-8">
</head>
```

One other thing that we will do is add references for all the commonly used Box2D objects to the beginning of `game.js` (see Listing 4-5).

Listing 4-5. Adding References to Commonly Used Box2D Objects

```
// 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;
```

Now that we have the references set up, we can start using Box2D from within our game code. We will be creating a separate `box2d` object inside `game.js` to store all our Box2D-related methods (see Listing 4-6).

Listing 4-6. Creating a box2d Object

```
var box2d = {
    scale:30,
    init:function(){
        // 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
        box2d.world = new b2World(gravity,allowSleep);
    },

    createRectangle:function(entity,definition){
            var bodyDef = new b2BodyDef;
            if(entity.isStatic){
                bodyDef.type = b2Body.b2_staticBody;
            } else {
                bodyDef.type = b2Body.b2_dynamicBody;
            }
```

```
            bodyDef.position.x = entity.x/box2d.scale;
            bodyDef.position.y = entity.y/box2d.scale;
            if (entity.angle) {
                bodyDef.angle = Math.PI*entity.angle/180;
            }

            var fixtureDef = new b2FixtureDef;
            fixtureDef.density = definition.density;
            fixtureDef.friction = definition.friction;
            fixtureDef.restitution = definition.restitution;

            fixtureDef.shape = new b2PolygonShape;
            fixtureDef.shape.SetAsBox(entity.width/2/box2d.scale,entity.height/2/box2d.scale);

            var body = box2d.world.CreateBody(bodyDef);
            body.SetUserData(entity);

            var fixture = body.CreateFixture(fixtureDef);
            return body;
    },

    createCircle:function(entity,definition){
            var bodyDef = new b2BodyDef;
            if(entity.isStatic){
                bodyDef.type = b2Body.b2_staticBody;
            } else {
                bodyDef.type = b2Body.b2_dynamicBody;
            }

            bodyDef.position.x = entity.x/box2d.scale;
            bodyDef.position.y = entity.y/box2d.scale;

            if (entity.angle) {
                bodyDef.angle = Math.PI*entity.angle/180;
            }
            var fixtureDef = new b2FixtureDef;
            fixtureDef.density = definition.density;
            fixtureDef.friction = definition.friction;
            fixtureDef.restitution = definition.restitution;

            fixtureDef.shape = new b2CircleShape(entity.radius/box2d.scale);

            var body = box2d.world.CreateBody(bodyDef);
            body.SetUserData(entity);

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

The box2d object connects an init() method that initializes a new b2World object, just like we did in Chapter 3. The object also contains two helper methods, createRectangle() and createCircle(). Both methods accept two parameters, the entity and definition objects that we described earlier. The entity object contains details such as its position, angle, and whether or not the entity is static. The definition object contains details about the fixture, such as restitution and density. Using these parameters, the methods create Box2D bodies and fixtures and add them to the Box2D world.

One thing to note is that both these methods convert the position and size using box2d.scale and convert the angle from degrees to radians before they can be used by Box2D.

One other thing that these methods do is attach the entity object to the body using the SetUserData() method. This enables us to retrieve any of the entity-related data for a Box2D body using its GetUserData() method.

Creating Entities

Now that we have Box2D set up, we will implement the entities.create() method inside the entities object that we defined earlier. This method will take an entity object as a parameter and add it to the world (see Listing 4-7).

Listing 4-7. Defining the entities.create() Method

```
// take the entity, create a Box2D body, and add it to the world
create:function(entity){
    var definition = entities.definitions[entity.name];
    if(!definition){
        console.log ("Undefined entity name",entity.name);
        return;
    }
    switch(entity.type){
        case "block": // simple rectangles
            entity.health = definition.fullHealth;
            entity.fullHealth = definition.fullHealth;
            entity.shape = "rectangle";
            entity.sprite = loader.loadImage("images/entities/"+entity.name+".png");
            box2d.createRectangle(entity,definition);
            break;
        case "ground": // simple rectangles
            // No need for health. These are indestructible
            entity.shape = "rectangle";
            // No need for sprites. These won't be drawn at all
            box2d.createRectangle(entity,definition);
            break;
        case "hero":    // simple circles
        case "villain": // can be circles or rectangles
            entity.health = definition.fullHealth;
            entity.fullHealth = definition.fullHealth;
            entity.sprite = loader.loadImage("images/entities/"+entity.name+".png");
            entity.shape = definition.shape;
            if(definition.shape == "circle"){
                entity.radius = definition.radius;
                box2d.createCircle(entity,definition);
```

```
            } else if(definition.shape == "rectangle"){
                entity.width = definition.width;
                entity.height = definition.height;
                box2d.createRectangle(entity,definition);
            }
            break;
        default:
            console.log("Undefined entity type",entity.type);
            break;
    }
},
```

In this method, we use the entity type to decide how to handle the entity object and its properties:

- *Block*: For block entities, we set the entity `health` and `fullHealth` properties based on the entity definition, and set the `shape` property to `"rectangle"`. We then load the sprite, and call the `box2d.createRectangle()` method.
- *Ground*: For ground entities, we set the entity object's `shape` property to `"rectangle"` and call the `box2d.createRectangle()` method. We do not load a sprite because we will be using the ground from the level foreground image and won't be drawing the ground separately.
- *Hero and villain*: For hero and villain entities, we set the entity `health`, `fullHealth`, and `shape` properties based on the entity definition. We then set either the `radius` or the `height` and `width` properties based on the shape of the entity. Finally, we call either `box2d.createRectangle()` or `box2d.createCircle()` based on the shape.

Now that we have a way to create entities, let's add some entities to our levels.

Adding Entities to Levels

The first thing we will do is add a few entities inside our `levels.data` array, as shown in Listing 4-8.

Listing 4-8. Adding Entities to the `levels.data` array

```
data:[
 {   // First level
    foreground:'desert-foreground',
    background:'clouds-background',
    entities:[
        {type:"ground", name:"dirt", x:500,y:440,width:1000,height:20,isStatic:true},
        {type:"ground", name:"wood", x:180,y:390,width:40,height:80,isStatic:true},

        {type:"block", name:"wood", x:520,y:375,angle:90,width:100,height:25},
        {type:"block", name:"glass", x:520,y:275,angle:90,width:100,height:25},
        {type:"villain", name:"burger",x:520,y:200,calories:590},

        {type:"block", name:"wood", x:620,y:375,angle:90,width:100,height:25},
        {type:"block", name:"glass", x:620,y:275,angle:90,width:100,height:25},
        {type:"villain", name:"fries", x:620,y:200,calories:420},
```

```
        {type:"hero", name:"orange",x:90,y:410},
        {type:"hero", name:"apple",x:150,y:410},
    ]
 },
    {   // Second level
        foreground:'desert-foreground',
        background:'clouds-background',
        entities:[
            {type:"ground", name:"dirt", x:500,y:440,width:1000,height:20,isStatic:true},
            {type:"ground", name:"wood", x:180,y:390,width:40,height:80,isStatic:true},
            {type:"block", name:"wood", x:820,y:375,angle:90,width:100,height:25},
            {type:"block", name:"wood", x:720,y:375,angle:90,width:100,height:25},
            {type:"block", name:"wood", x:620,y:375,angle:90,width:100,height:25},
            {type:"block", name:"glass", x:670,y:310,width:100,height:25},
            {type:"block", name:"glass", x:770,y:310,width:100,height:25},

            {type:"block", name:"glass", x:670,y:248,angle:90,width:100,height:25},
            {type:"block", name:"glass", x:770,y:248,angle:90,width:100,height:25},
            {type:"block", name:"wood", x:720,y:180,width:100,height:25},

            {type:"villain", name:"burger",x:715,y:160,calories:590},
            {type:"villain", name:"fries",x:670,y:400,calories:420},
            {type:"villain", name:"sodacan",x:765,y:395,calories:150},

            {type:"hero", name:"strawberry",x:40,y:420},
            {type:"hero", name:"orange",x:90,y:410},
            {type:"hero", name:"apple",x:150,y:410},
        ]
    }
],
```

The first level contains two background `ground` entities—one for the floor and the other for the slingshot. These entities are meant to be static objects that are not drawn by us.

The level also contains four rectangular `block` entities (`glass` and `wood`). These are destructible elements that we have positioned using their `angle`, `x`, and `y` properties.

Finally, the level contains two `hero` entities (`orange` and `apple`) and two `villain` entities (`burger` and `fries`). Note that the villains have an extra property called `calories` that we will use to increase the player score when they are destroyed.

The second level has a similar design, except with a few more entities.

Now that we have defined entities for each level, we need to load these entities when we load the level. To do this, we will modify the `load()` method of the `levels` object (see Listing 4-9).

Listing 4-9. Modifying levels.load() to Load the Entities

```
// Load all data and images for a specific level
load:function(number){
   //Initialize Box2D world whenever a level is loaded
    box2d.init();

    // declare a new current level object
    game.currentLevel = {number:number,hero:[]};
    game.score=0;
```

```
    $('#score').html('Score: '+game.score);
    game.currentHero = undefined;
    var level = levels.data[number];


    //load the background, foreground, and slingshot images
    game.currentLevel.backgroundImage =
loader.loadImage("images/backgrounds/"+level.background+".png");
    game.currentLevel.foregroundImage =
loader.loadImage("images/backgrounds/"+level.foreground+".png");
    game.slingshotImage = loader.loadImage("images/slingshot.png");
    game.slingshotFrontImage = loader.loadImage("images/slingshot-front.png");

    // Load all the entities
    for (var i = level.entities.length - 1; i >= 0; i--){
        var entity = level.entities[i];
        entities.create(entity);
    };

      //Call game.start() once the assets have loaded
   if(loader.loaded){
       game.start()
   } else {
       loader.onload = game.start;
   }
}
```

The first change we have made is the addition of a call to `box2d.init()` at the very beginning of the method. The other change is the addition of a `for` loop where we iterate through all the entities for a level and call `entities.create()` for each entity. Now when we load a level, Box2D will get initialized and all the entities will get loaded into the Box2D world.

We still can't see the bodies we have added. Let's use the Box2D debug drawing method introduced in Chapter 3 to see what we created.

Setting Up Box2D Debug Drawing

The first thing we will do is create another `canvas` element inside the HTML file and place it just before the end of the `<body>` tag:

```
<canvas id="debugcanvas" width="1000" height="480" style="border:1px solid black;"></canvas>
```

This canvas is larger than our game canvas, so we can see the entire level without any panning. We will be using this canvas and debug drawing only to design and test our levels. We can remove all traces of debug drawing once the game is complete.

The next thing we need to do is set up debug drawing when we are initializing Box2D. We will do this by modifying the `box2d.init()` method so that it looks as shown in Listing 4-10.

Listing 4-10. Modifying box2d.init() to Set Up Debug Draw

```
init:function(){
    // 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
    box2d.world = new b2World(gravity,allowSleep);

    // Set up debug draw
    var debugContext = document.getElementById('debugcanvas').getContext('2d');
    var debugDraw = new b2DebugDraw();
    debugDraw.SetSprite(debugContext);
    debugDraw.SetDrawScale(box2d.scale);
    debugDraw.SetFillAlpha(0.3);
    debugDraw.SetLineThickness(1.0);
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit | b2DebugDraw.e_jointBit);
    box2d.world.SetDebugDraw(debugDraw);
},
```

This newly added code is the same as the code in Chapter 3. Before we can see the results of debug draw, we need to call the `world` object's `DrawDebugData()` method. We will do this in a new method called `drawAllBodies()` inside the `game` object, as shown in Listing 4-11. We will call this method from the `animate()` method of the `game` object.

Listing 4-11. Modifying animate() and Creating drawAllBodies()

```
animate:function(){
    // Animate the background
    game.handlePanning();

    // TODO: Animate the characters

    //  Draw the background with parallax scrolling
    game.context.drawImage(game.currentLevel.backgroundImage,game.offsetLeft/4,0
,640,480,0,0,640,480);

game.context.drawImage(game.currentLevel.foregroundImage,game.offsetLeft,0,640,480,0,0,640,480);

    // Draw the slingshot
    game.context.drawImage(game.slingshotImage,game.slingshotX-game.offsetLeft,game.slingshotY);

    // Draw all the bodies
    game.drawAllBodies();

    // Draw the front of the slingshot
    game.context.drawImage(game.slingshotFrontImage,game.slingshotX-game.offsetLeft,game.slingshotY);
```

```
    if (!game.ended){
        game.animationFrame = window.requestAnimationFrame(game.animate,game.canvas);
    }
},
drawAllBodies:function(){
    box2d.world.DrawDebugData();
    // TODO: Iterate through all the bodies and draw them on the game canvas
}
```

For now, we have created a simple `drawAllBodies()` method that calls `box2d.world.DrawDebugData()`. We will eventually need to add code to iterate through all the bodies in the Box2D world and draw them on the game canvas. We call this new method from inside the `game` object's `animate()` method.

If we run our code now and load the first level, we should see the debug canvas with all the entities, as shown in Figure 4-1.

Figure 4-1. *First level drawn on the debug canvas*

The debug canvas view shows us all the game entities as circles and rectangles. We can also see the ground and slingshot blocks in a different color. We can use this view to quickly test our levels and make sure that all the entities are positioned correctly. Now that we can see that everything in the level looks alright, it's time to actually draw all the entities onto our game canvas.

Drawing the Entities

To draw an entity, we will define a method called `draw()` inside the `entities` object. This object will take the entity, its position, and its angle as parameters and draw it on the game canvas (see Listing 4-12).

Listing 4-12. The entities.draw() Method

```
// take the entity, its position, and its angle and draw it on the game canvas
draw:function(entity,position,angle){
    game.context.translate(position.x*box2d.scale-game.offsetLeft,position.y*box2d.scale);
    game.context.rotate(angle);
    switch (entity.type){
        case "block":
            game.context.drawImage(entity.sprite,0,0,entity.sprite.width,entity.sprite.height,
                    -entity.width/2-1,-entity.height/2-1,entity.width+2,entity.height+2);
        break;
        case "villain":
```

```
        case "hero":
            if (entity.shape=="circle"){

game.context.drawImage(entity.sprite,0,0,entity.sprite.width,entity.sprite.height,
                        -entity.radius-1,-entity.radius-1,entity.radius*2+2,entity.radius*2+2);
            } else if (entity.shape=="rectangle"){

game.context.drawImage(entity.sprite,0,0,entity.sprite.width,entity.sprite.height,
                        -entity.width/2-1,-entity.height/2-1,entity.width+2,entity.height+2);
            }
            break;
        case "ground":
            // do nothing... We will draw objects like the ground & slingshot separately
            break;
    }

    game.context.rotate(-angle);
    game.context.translate(-position.x*box2d.scale+game.offsetLeft,-position.y*box2d.scale);
}
```

This method first translates and rotates the context to the position and angle of the entity. It then draws the object on the canvas based on the entity type and shape. Finally, it rotates and translates the context back to the original position.

One thing to note is that the code stretches the image to a size of one pixel larger than the sprite definition in each direction when using `drawImage()`. This is so that small gaps between Box2D objects get covered up.

■ **Note** Box2D creates a "skin" around all polygons. The skin is used in stacking scenarios to keep polygons slightly separated. This allows continuous collision to work against the core polygon. When drawing Box2D objects, we need to compensate for this extra skin by drawing bodies slightly larger than their actual dimensions; otherwise, stacked objects will have unexplained gaps between them.

Now that we have defined an `entities.draw()` method, we need to call this method for every entity in our game world. We can iterate through every body in the game world by using the world object's `GetBodyList()` method. We will now modify the `game` object's `drawAllBodies()` method to do this, as shown in Listing 4-13.

Listing 4-13. Iterating Through All the Bodies and Drawing Them

```
drawAllBodies:function(){
    box2d.world.DrawDebugData();

    // Iterate through all the bodies and draw them on the game canvas
    for (var body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();

        if(entity){
            entities.draw(entity,body.GetPosition(),body.GetAngle())
        }
    }
}
```

The for loop initializes body using `world.GetBodyList()`, which returns the first body in the world. The body object's `GetNext()` method returns the next body in the list until it reaches the end of the list, at which point we exit the for loop. Within the loop, we check to see if the body has an attached entity; if it does, we call `entities.draw()`, passing it the body's `entity` object, position, and angle.

If we run our game and load the first level now, we should see all the entities drawn on the canvas, as shown in Figure 4-2.

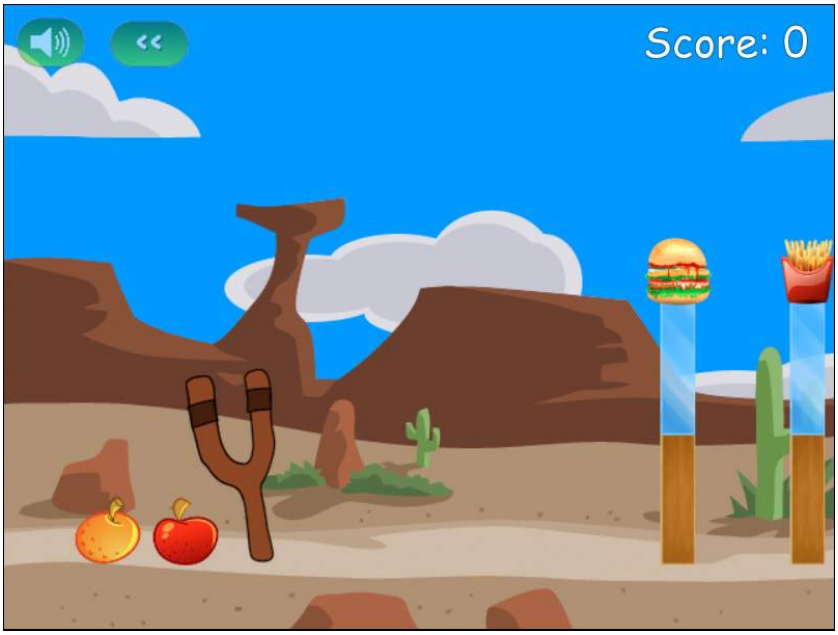

Figure 4-2. *Drawing the game entities on the canvas*

Once the level loads, the game pans to the right so that we can see the bad guys clearly, and then it pans back to the slingshot. We can see all the entities drawn properly at the same locations as on the debug canvas. The extra pixel we added in our `draw()` method ensures that all the stacked objects are positioned tightly next to each other. Note that the canvas preserves image transparencies when drawing images, which is why we can see the background through the glass block.

Now that we have drawn all the elements in the Box2D world, we need to animate the Box2D world.

Animating the Box2D World

As in the previous chapter, we can animate the Box2D world by calling the `world` object's `Step()` method and passing it the time step interval as a parameter. However, this is where things get a little tricky.

As per the Box2D manual recommendation, ideally, we should use a fixed time step for best results because variable time steps are hard to debug. Also as per the manual, Box2D works best with a time step of around 1/60th of a second, and you should use a time step no larger than 1/30th of a second. If the time step becomes very large, Box2D starts having problems with collisions, and bodies start passing through each other.

The `requestAnimationFrame` API can vary the frequency at which it calls the `animate()` method across browsers and machines. One way to get around this is to measure the time elapsed since the last call to `animate()` and pass this difference as a time step to Box2D.

However, if we switch tabs on the browser and then return to the game tab, the browser will call the `animate()` method less often, and this time step may become much larger than the upper limit of 1/30th of a second. To avoid problems due to a large time step, we will need to actively cap the time step if it becomes larger than 1/30th of a second.

Armed with this information, we will first define a `step()` method inside the `box2d` object that takes a time interval as a parameter and calls the world object's `Step()` method (see Listing 4-14).

Listing 4-14. The box2d.step() Method

```
step:function(timeStep){
    // velocity iterations = 8
    // position iterations = 3
    if(timeStep >2/60){
        timeStep = 2/60
    }

    box2d.world.Step(timeStep,8,3);
},
```

The `step()` method takes a time step in seconds and passes it to the `world.Step()` method. If `timeStep` is too large, we cap it at 1/30th of a second. We use the Box2D manual recommended values of 8 and 3 for velocity and position iterations. We will call this method from the `game.animate()` method after calculating the time step. The `game.animate()` method will now look like Listing 4-15.

Listing 4-15. Calling box2d.step() from game.animate()

```
animate:function(){
    // Animate the background
    game.handlePanning();

    // Animate the characters
    var currentTime = new Date().getTime();
    var timeStep;
    if (game.lastUpdateTime){
        timeStep = (currentTime - game.lastUpdateTime)/1000;
        box2d.step(timeStep);
    }

    game.lastUpdateTime = currentTime;

    //  Draw the background with parallax scrolling

game.context.drawImage(game.currentLevel.backgroundImage,game.offsetLeft/4,0,640,480,0,0,640,480);

game.context.drawImage(game.currentLevel.foregroundImage,game.offsetLeft,0,640,480,0,0,640,480);

     // Draw the slingshot
    game.context.drawImage(game.slingshotImage, game.slingshotX-game.offsetLeft, game.slingshotY);
    game.drawAllBodies();

    // Draw the front of the slingshot
    game.context.drawImage(game.slingshotFrontImage,game.slingshotX-game.offsetLeft,game.slingshotY);
```

```
    if (!game.ended){
        game.animationFrame = window.requestAnimationFrame(game.animate,game.canvas);
    }
},
```

We calculate `timeStep` as the difference between `lastUpdateTime` and `currentTime` and then call the `box2d.step()` method. We then save the current time into the game.lastUpdateTime variable.

The first time `animate()` is called, `game.lastUpdateTime` will be undefined, so we will not calculate `timeStep` or call `box2d.step()`.

Loading the Hero

Now that the animation and engine are in place, it's time to implement some more game states (a.k.a. game modes). The first state that we will implement is the `load-next-hero` state. When in this state, the game needs to count the number of heroes and villains left in the game, check how many are left, and act accordingly, as follows:

- If all the villains are gone, the game switches to the state `level-success`.
- If all the heroes are gone, the game switches to the state `level-failure`.
- If there are still heroes remaining, the game places the first hero on top of the slingshot and then switches to the state `wait-for-firing`.

We will do this by creating a method called `game.countHeroesAndVillains()` and modifying the `game.handlePanning()` method, as shown in Listing 4-16.

Listing 4-16. Handling the load-next-hero State

```
countHeroesAndVillains:function(){
    game.heroes = [];
    game.villains = [];
    for (var body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();
        if(entity){
            if(entity.type == "hero"){
                game.heroes.push(body);
            } else if (entity.type =="villain"){
                game.villains.push(body);
            }
        }
    }
},
handlePanning:function(){
    if (game.mode=="intro"){
        if(game.panTo(700)){
            game.mode = "load-next-hero";
        }
    }

    if (game.mode=="wait-for-firing"){
        game.panTo(game.slingshotX);
    }
```

```
    if (game.mode == "firing"){
        game.panTo(game.slingshotX);
    }

    if (game.mode == "fired"){
        // TODO:
        // Pan to wherever the hero currently is
    }

    if (game.mode == "load-next-hero"){
        game.countHeroesAndVillains();

        // Check if any villains are alive, if not, end the level (success)
        if (game.villains.length == 0){
             game.mode = "level-success";
            return;
        }

        // Check if there are any more heroes left to load, if not end the level (failure)
        if (game.heroes.length == 0){
            game.mode = "level-failure"
            return;
        }

        // Load the hero and set mode to wait-for-firing
        if(!game.currentHero){
            game.currentHero = game.heroes[game.heroes.length-1];
            game.currentHero.SetPosition({x:180/box2d.scale,y:200/box2d.scale});
             game.currentHero.SetLinearVelocity({x:0,y:0});
             game.currentHero.SetAngularVelocity(0);
            game.currentHero.SetAwake(true);
        } else {
            // Wait for hero to stop bouncing and fall asleep and then switch to wait-for-firing
            game.panTo(game.slingshotX);
            if(!game.currentHero.IsAwake()){
                game.mode = "wait-for-firing";
            }
        }
    }
},
```

The `countHeroesAndVillains()` method iterates through all the bodies in the world and stores the heroes in the `game.heroes` array and the villains in the `game.villains` array.

Inside the `handlePanning()` method, when `game.mode` is `load-next-hero`, we first call `countHeroesandVillains()`. We then check to see if the villain or hero count is 0 and, if so, set `game.mode` to `level-success` or `level-failure`, respectively. If not, we save the last hero in the `game.heroes` array into the `game.currentHero` variable and set its position to a point in the air above the slingshot. We set its angular and linear velocity to 0. We also wake up the body in case it is asleep.

When the body drops on to the slingshot, it will keep bouncing until it finally comes to rest and falls asleep again. Once the body goes back to sleep, we set `game.mode` to `wait-for-firing`. If we run the game and start the first level, we will see the first hero bounce on the slingshot and come to rest, as shown in Figure 4-3.

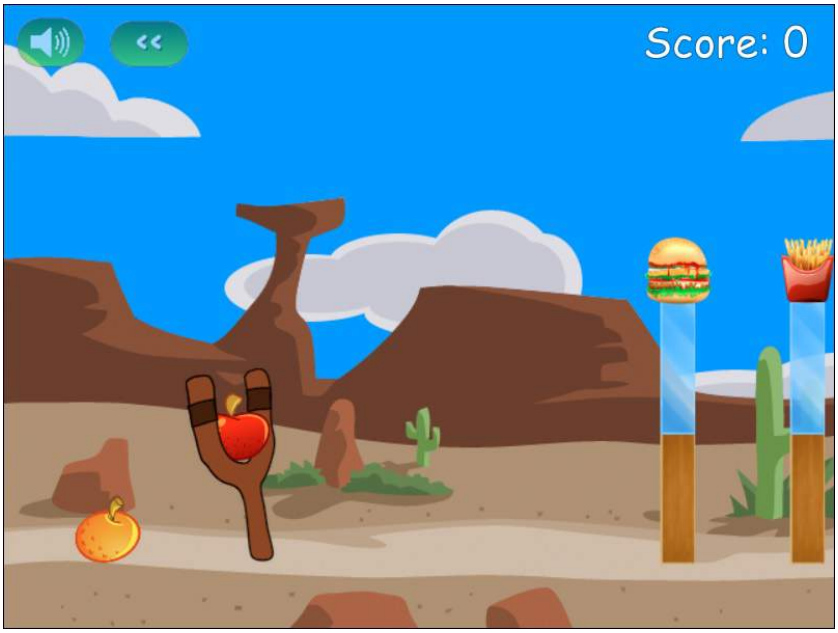

Figure 4-3. *First hero loaded on slingshot and waiting to be fired*

Now that we have the hero ready to be fired, we need to handle firing the hero from the slingshot.

Firing the Hero

We will implement firing the hero using three states:

- `wait-for-firing`: The game pans over the slingshot and waits for the mouse to be clicked and dragged while the pointer is above the hero. When this happens, it shifts to the `firing` state.
- `firing`: The game moves the hero with the mouse until the mouse button is released. When this happens, it pushes the hero with an impulse based on its distance from the slingshot and shifts to the `fired` state.
- `fired`: The game pans to follow the hero until it either comes to rest or goes outside the level bounds. Then game then removes the hero from the game world and goes back to the `load-next-hero` state.

We will first implement a method called `mouseOnCurrentHero()` inside the `game` object to test if the mouse pointer is positioned on the current hero (see Listing 4-17).

Listing 4-17. The mouseOnCurrentHero() Method

```
mouseOnCurrentHero:function(){
    if(!game.currentHero){
        return false;
    }
```

```
        var position = game.currentHero.GetPosition();
        var distanceSquared = Math.pow(position.x*box2d.scale - mouse.x-game.offsetLeft,2) +
Math.pow(position.y*box2d.scale-mouse.y,2);
        var radiusSquared = Math.pow(game.currentHero.GetUserData().radius,2);
        return (distanceSquared<= radiusSquared);
    },
```

This method calculates the distance between the current hero center and the mouse location and compares it with the radius of the current hero to check if the mouse is positioned over the hero. If the distance is less than the radius, the mouse pointer is positioned on the hero.

We can get away with using this simple check since all our heroes are circular. If you want to implement heroes with different shapes, you might need a more complex method.

Now that we have this method in place, we can implement the three states inside the `handlePanning()` method, as shown in Listing 4-18.

Listing 4-18. Handling the Firing States Inside the handlePanning() Method

```
if (game.mode=="wait-for-firing"){
    if (mouse.dragging){
        if (game.mouseOnCurrentHero()){
            game.mode = "firing";
        } else {
            game.panTo(mouse.x + game.offsetLeft)
        }
    } else {
        game.panTo(game.slingshotX);
    }
}

if (game.mode == "firing"){
    if(mouse.down){
        game.panTo(game.slingshotX);

game.currentHero.SetPosition({x:(mouse.x+game.offsetLeft)/box2d.scale,y:mouse.y/box2d.scale});
    } else {
        game.mode = "fired";
        var impulseScaleFactor = 0.75;
        var impulse = new b2Vec2((game.slingshotX+35-mouse.x-game.offsetLeft)*impulseScaleFactor,
(game.slingshotY+25-mouse.y)*impulseScaleFactor);
        game.currentHero.ApplyImpulse(impulse,game.currentHero.GetWorldCenter());
    }
}

if (game.mode == "fired"){
    //pan to wherever the current hero is...
    var heroX = game.currentHero.GetPosition().x*box2d.scale;
    game.panTo(heroX);

    //and wait till he stops moving or is out of bounds
    if(!game.currentHero.IsAwake() || heroX<0 || heroX >game.currentLevel.foregroundImage.width ){
```

```
        // then delete the old hero
        box2d.world.DestroyBody(game.currentHero);
        game.currentHero = undefined;
        // and load next hero
        game.mode = "load-next-hero";
    }
}
```

When the state is `wait-for-firing` and the mouse is being dragged, we change the state to `firing` if the mouse pointer is positioned on the hero; if the mouse pointer is not positioned on the hero, we pan the screen toward the cursor. If the mouse is not being dragged, we pan toward the slingshot.

When the state is `firing` and the mouse button is down, we set the position of the hero to the mouse position and pan toward the slingshot. When the mouse button is released, we set the state to `fired` and apply an impulse to the hero using the `b2Body` object's `ApplyImpulse()` method. This method takes the impulse as a parameter in the form of a `b2Vec2` object. We set the `x` and `y` values of the impulse vector as a multiple of the x and y distance of the hero from the top of the slingshot. (The impulse scaling factor is a number that I came up with by experimenting with different values.)

When the state is `fired`, we pan the screen toward the hero and wait for the hero to either come to rest or fall outside of the game bounds. When it does either, we remove the hero from the world using the `DestroyBody()` method and change the state back to `load-next-hero`.

When we run this finished code and load the level, we should be able to fire the hero at the blocks and knock them down, as shown in Figure 4-4.

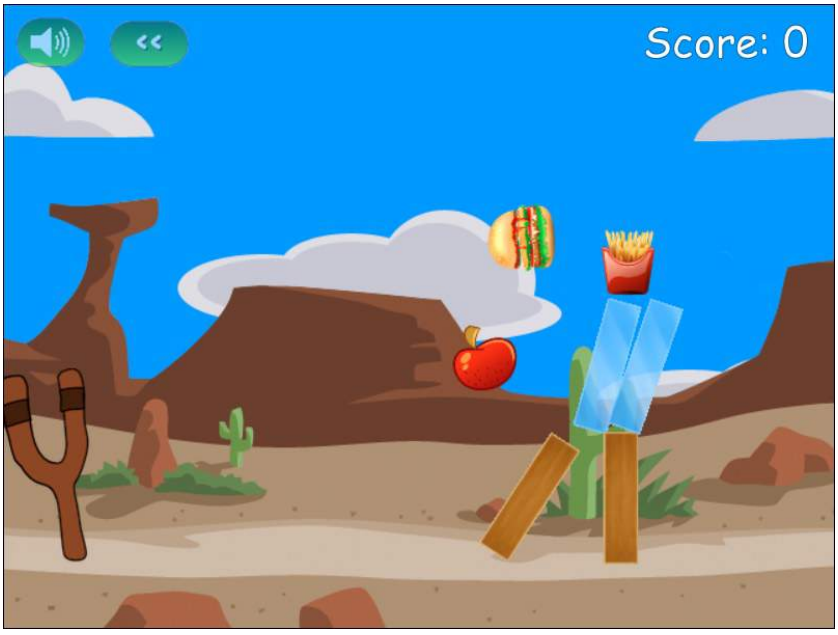

Figure 4-4. *Firing the hero at the blocks and knocking them over*

Once the hero either stops rolling or goes outside the bounds of the level, it is removed from the game and the next hero is loaded onto the slingshot. At this point, once all the heroes are gone, the game just stops and waits. So, the last thing that we need to do is implement ending the level.

Ending the Level

Once a level ends, we will stop the game animation loop and display a level ending screen. This screen will give the user options to replay the current level, proceed to the next level, or return to the level selection screen.

The first thing we need to do is add `onclick` event handlers to the `endingscreen div` element and place it in the `index.html` file after the other game layers. The final markup will look like Listing 4-19.

Listing 4-19. Finished endingscreen div Element

```
<div id="endingscreen" class="gamelayer">
    <div>
        <p id="endingmessage">The Level Is Over Message</p>
        <p id="playcurrentlevel" onclick="game.restartLevel();"><img src="images/icons/prev.png">
Replay Current Level</p>
        <p id="playnextlevel" onclick="game.startNextLevel();"><img src="images/icons/next.png">
Play Next Level </p>
        <p id="returntolevelscreen"onclick="game.showLevelScreen();"><img
src="images/icons/return.png"> Return to Level Screen</p>
    </div>
</div>
```

We also need to add the corresponding CSS into `styles.css`, as shown in Listing 4-20.

Listing 4-20. CSS for the endingscreen div Element

```
/* Ending Screen */
endingscreen {
    text-align:center;
}

#endingscreen div {
    height:430px;
    padding-top:50px;
    border:1px;
    background:rgba(1,1,1,0.5);
    text-align:left;
    padding-left:100px;
}

#endingscreen p  {
    font: 20px Comic Sans MS;
    text-shadow: 0 0 2px #000;
    color:white;
}

#endingscreen p img{
    top:10px;
    position:relative;
    cursor:pointer;
}
```

```
#endingscreen #endingmessage  {
    font: 32px Comic Sans MS;
    text-shadow: 0 0 2px #000;
    color:white;
}
```

Now that the ending screen is ready, we will implement a method called `showEndingScreen()` inside the `game` object that will display the `endingscreen div` element (see Listing 4-21).

Listing 4-21. The game.showEndingScreen() Method

```
showEndingScreen:function(){
    if (game.mode=="level-success"){
        if(game.currentLevel.number<levels.data.length-1){
            $('#endingmessage').html('Level Complete. Well Done!!!');
            $("#playnextlevel").show();
        } else {
            $('#endingmessage').html('All Levels Complete. Well Done!!!');
            $("#playnextlevel").hide();
        }
    } else if (game.mode=="level-failure"){
        $('#endingmessage').html('Failed. Play Again?');
        $("#playnextlevel").hide();
    }

    $('#endingscreen').show();
},
```

The `showEndingScreen()` method shows different messages based on the value of `game.mode`. The option to play the next level is shown if the player was successful and the current level was not the final level of the game. If the player was unsuccessful or the current level was the final level, the option is hidden.

We will now handle `level-success` and `level-failure` within the `handlePanning()` method of the `game` object by adding the code shown in Listing 4-22 to the bottom of the method.

Listing 4-22. Implementing the Level Ending States

```
if(game.mode=="level-success" || game.mode=="level-failure"){
    if(game.panTo(0)){
        game.ended = true;

        game.showEndingScreen();
    }
}
```

When `game.mode` is either `level-success` or `level-failure`, the game first pans back to the left, then sets the `game.ended` property to `true`, and finally displays the ending screen shown in Figure 4-5.

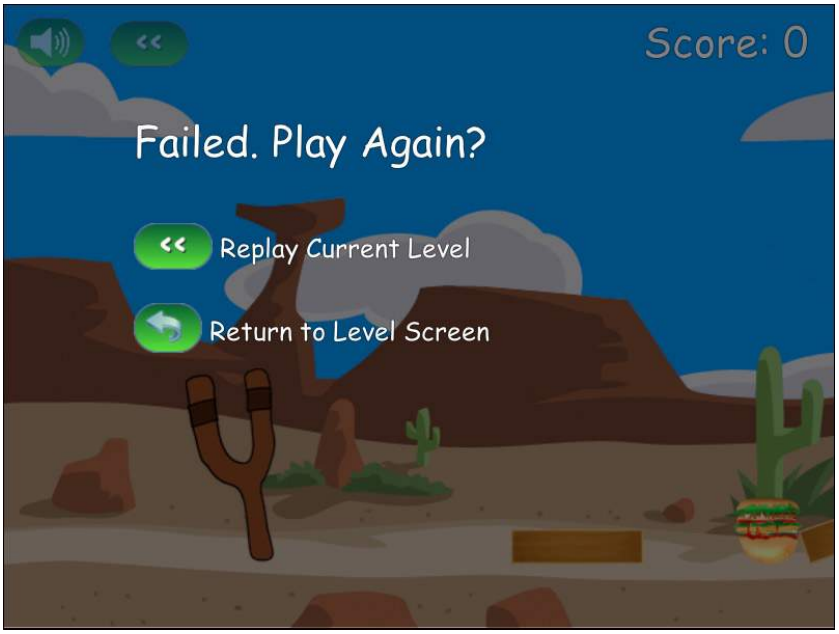

Figure 4-5. *The level ending screen*

Of course, since we haven't yet implemented collision damage, the villains cannot die and we can never win. Therefore, the next thing we will implement is collision damage.

Collision Damage

The first thing we need to do is track collisions by using a contact listener and overriding its `PostSolve()` method, just like we did in Chapter 3. We will add this listener to the world immediately after it has been created in the `init()` method of the `box2d` object, as shown in Listing 4-23.

Listing 4-23. Handling Collisions Using a Contact Listener

```
init:function(){
    // 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
    box2d.world = new b2World(gravity,allowSleep);

    var debugContext = document.getElementById('debugcanvas').getContext('2d');
    var debugDraw = new b2DebugDraw();
    debugDraw.SetSprite(debugContext);
    debugDraw.SetDrawScale(box2d.scale);
    debugDraw.SetFillAlpha(0.3);
    debugDraw.SetLineThickness(1.0);
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit | b2DebugDraw.e_jointBit);
    box2d.world.SetDebugDraw(debugDraw);
```

```
    var listener = new Box2D.Dynamics.b2ContactListener;
    listener.PostSolve = function(contact,impulse){
        var body1 = contact.GetFixtureA().GetBody();
        var body2 = contact.GetFixtureB().GetBody();
        var entity1 = body1.GetUserData();
        var entity2 = body2.GetUserData();

        var impulseAlongNormal = Math.abs(impulse.normalImpulses[0]);
        // This listener is called a little too often. Filter out very tiny impulses.
        // After trying different values, 5 seems to work well
        if(impulseAlongNormal>5){
            // If objects have a health, reduce health by the impulse value
            if (entity1.health){
                entity1.health -= impulseAlongNormal;
            }

            if (entity2.health){
                entity2.health -= impulseAlongNormal;
            }
        }
    };
    box2d.world.SetContactListener(listener);
},
```

Within the `PostSolve()` method, if either of the bodies involved in the collision has a `health` property, we reduce the health by the value of the impulse along the normal. Since the `PostSolve()` method is called for every little collision, we ignore any collision where `impulseAlongNormal` is less than a threshold.

The next thing we will do is add some code in the `game` object's `drawAllBodies()` method to check if a body's `health` property is less than zero or the body has gone outside the level bounds. If either is true, we will remove the body from the world. The `drawAllBodies()` method now looks like Listing 4-24.

Listing 4-24. Removing Dead Bodies from the World

```
drawAllBodies:function(){
    box2d.world.DrawDebugData();

    // Iterate through all the bodies and draw them on the game canvas
    for (var body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();

        if(entity){
            var entityX = body.GetPosition().x*box2d.scale;
            if(entityX<0|| entityX>game.currentLevel.foregroundImage.width||(entity.health &&
entity.health <0)){
                box2d.world.DestroyBody(body);
                if (entity.type=="villain"){
                    game.score += entity.calories;
                    $('#score').html('Score: '+game.score);
                }
```

```
            } else {
                entities.draw(entity,body.GetPosition(),body.GetAngle())
            }
        }
    }
}
```

If the code finds that the body has gone outside the level bounds or the entity has lost all its health, we use the `world` object's `DestroyBody()` method to remove the body. Additionally, if the entity is a villain, we add the entity's `calorie` value to the game score.

When we run the game, the villains do get destroyed and the score increases, as shown in Figure 4-6.

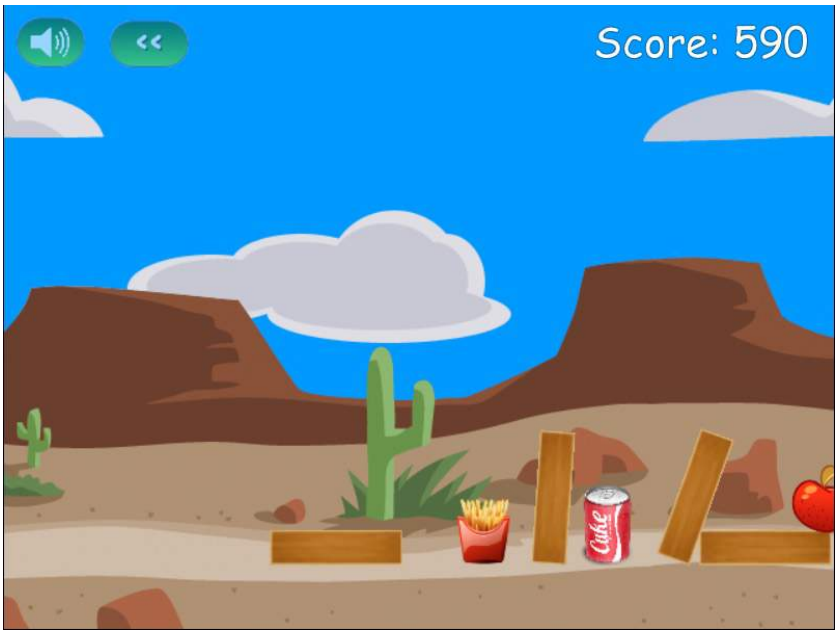

Figure 4-6. *The score increases after a bad guy gets destroyed*

Now that we have a working level, let's add a few finishing touches. The first thing we will do is draw a slingshot band when the hero is being fired.

Drawing the Slingshot Band

The slingshot band is going to be a thick brown line from the end of the slingshot to the extreme end of the hero. We will draw the band only when the game is in `firing` mode. We will do this in a `drawSlingshotBand()` method inside the `game` object, as shown in Listing 4-25.

Listing 4-25. Drawing the Slingshot Band

```
drawSlingshotBand:function(){
    game.context.strokeStyle = "rgb(68,31,11)"; // Darker brown color
    game.context.lineWidth = 6; // Draw a thick line

    // Use angle hero has been dragged and radius to calculate coordinates of edge of hero wrt.
hero center
    var radius = game.currentHero.GetUserData().radius;
    var heroX = game.currentHero.GetPosition().x*box2d.scale;
    var heroY = game.currentHero.GetPosition().y*box2d.scale;
    var angle = Math.atan2(game.slingshotY+25-heroY,game.slingshotX+50-heroX);

    var heroFarEdgeX = heroX - radius * Math.cos(angle);
    var heroFarEdgeY = heroY - radius * Math.sin(angle);

    game.context.beginPath();
    // Start line from top of slingshot (the back side)
    game.context.moveTo(game.slingshotX+50-game.offsetLeft, game.slingshotY+25);

    // Draw line to center of hero
    game.context.lineTo(heroX-game.offsetLeft,heroY);
    game.context.stroke();

    // Draw the hero on the back band

entities.draw(game.currentHero.GetUserData(),game.currentHero.GetPosition(),game.currentHero.
GetAngle());

    game.context.beginPath();
    // Move to edge of hero farthest from slingshot top
    game.context.moveTo(heroFarEdgeX-game.offsetLeft,heroFarEdgeY);

    // Draw line back to top of slingshot (the front side)
    game.context.lineTo(game.slingshotX-game.offsetLeft +10,game.slingshotY+30)
    game.context.stroke();
},
```

We start by setting the drawing color to a dark brown using the `strokeStyle` property. We next set the line drawing width to 6 pixels using the `lineWidth` property. We then draw a band from the back of the slingshot to the hero, draw the hero on top of the band, and, finally, draw a band from the front of the slingshot to the edge of the hero furthest from the slingshot.

We will call this method from the `game.animate()` method right after we draw all the other bodies. The final `game.animate()` method will look like Listing 4-26.

Listing 4-26. Calling the drawSlingshotBand() Method from animate()

```
animate:function(){
    // Animate the background
    game.handlePanning();

    // Animate the characters
        var currentTime = new Date().getTime();
```

```
        var timeStep;
        if (game.lastUpdateTime){
            timeStep = (currentTime - game.lastUpdateTime)/1000;
            if(timeStep >2/60){
                timeStep = 2/60
            }
            box2d.step(timeStep);
        }
        game.lastUpdateTime = currentTime;

    //  Draw the background with parallax scrolling

game.context.drawImage(game.currentLevel.backgroundImage,game.offsetLeft/4,0,640,480,0,0,640,480);

game.context.drawImage(game.currentLevel.foregroundImage,game.offsetLeft,0,640,480,0,0,640,480);

    // Draw the slingshot
    game.context.drawImage(game.slingshotImage,game.slingshotX-game.offsetLeft,game.slingshotY);

    // Draw all the bodies
    game.drawAllBodies();

    // Draw the band when we are firing a hero
    if(game.mode == "firing"){
        game.drawSlingshotBand();
    }

    // Draw the front of the slingshot
    game.context.drawImage(game.slingshotFrontImage,game.slingshotX-game.offsetLeft,
game.slingshotY);

    if (!game.ended){
        game.animationFrame = window.requestAnimationFrame(game.animate,game.canvas);
    }
},
```

When we run this code, we should see a brown band around the hero, as shown in Figure 4-7.

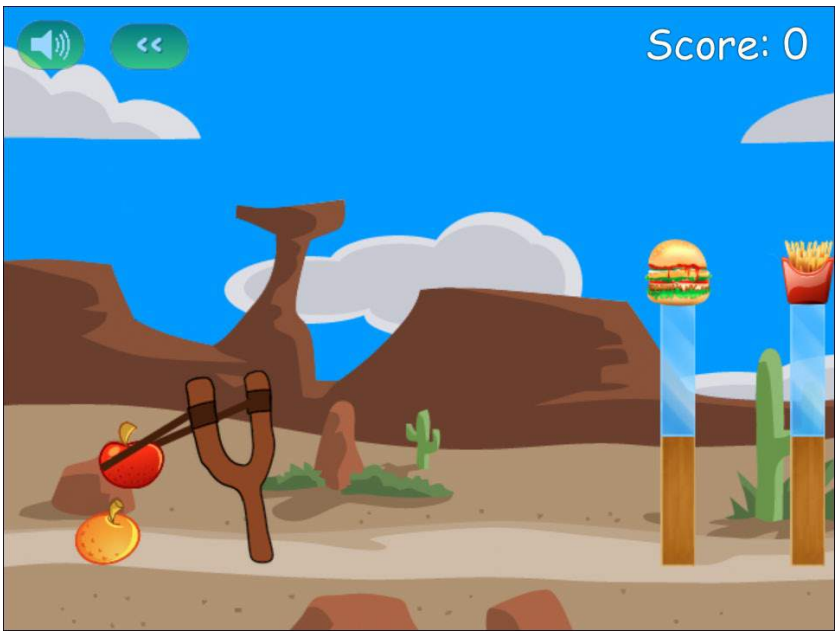

Figure 4-7. *Drawing the slingshot band*

This isn't a complete solution. The band might look a little unnatural at certain extreme angles. You might consider improving this method by superimposing some extra images on top of the band to cover up these edge effects. For now, this simple implementation will suffice.

Now that we have the artwork for the level wrapped up, let's implement the buttons for changing and restarting levels.

Changing Levels

We have already implemented one way to traverse levels, using the level selection screen. Now we will implement the buttons for restarting a level and proceeding to the next level.

We start by implementing the `restartLevel()` and `startNextLevel()` methods inside the `game` object, as shown in Listing 4-27.

Listing 4-27. Implementing restartLevel() and startNextLevel()

```
restartLevel:function(){
    window.cancelAnimationFrame(game.animationFrame);
    game.lastUpdateTime = undefined;
    levels.load(game.currentLevel.number);
},
startNextLevel:function(){
    window.cancelAnimationFrame(game.animationFrame);
    game.lastUpdateTime = undefined;
    levels.load(game.currentLevel.number+1);
},
```

The methods are fairly simple. Both of them cancel any existing `animationFrame` loops, reset the `game.lastUpdateTime` variable, and finally call the `levels.load()` method with the appropriate level number. We need to call these methods from the `onclick` event of the corresponding images in the `scorescreen` and `endingscreen` layers, as shown in Listing 4-28.

Listing 4-28. Setting the onclick Events for Changing Levels

```
<div id="scorescreen" class="gamelayer">
    <img id="togglemusic" src="images/icons/sound.png">
    <img src="images/icons/prev.png" onclick="game.restartLevel();">
    <span id="score">Score: 0</span>
</div>

<div id="endingscreen" class="gamelayer">
    <div>
        <p id="endingmessage">The Level Is Over Message</p>
        <p id="playcurrentlevel" onclick="game.restartLevel();"><img
src="images/icons/prev.png"> Replay Current Level</p>
        <p id="playnextlevel" onclick="game.startNextLevel();"><img
src="images/icons/next.png"> Play Next Level </p>
        <p id="returntolevelscreen"onclick="game.showLevelScreen();"><img
src="images/icons/return.png"> Return to Level Screen</p>
    </div>
</div>
```

If we run the game, we should now be able to restart a level or proceed to the next level using the provided buttons.

We now have a working game with complete levels. We also have a simple way to build new levels. However, there is still one last element missing: sound.

Adding Sound

Adding sound makes a game much more immersive. We will start by adding a few sound effects for when the slingshot is released, for when a hero or villain bounces, and for when one of the blocks gets destroyed. We will also add some background music, along with the capability to turn it off if we want.

The sounds files for each of these effects are available in the `audio` folder (in both MP3 and OGG format).

We will start by loading these sound files in the `game` object's `init()` method, as shown in Listing 4-29.

Listing 4-29. Loading Sound and Background Music

```
init: function(){
    // Initialize objects
    levels.init();
    loader.init();
    mouse.init();

    // Load All Sound Effects and Background Music

    //"Kindergarten" by Gurdonark
    //http://ccmixter.org/files/gurdonark/26491 is licensed under a Creative Commons license
    game.backgroundMusic = loader.loadSound('audio/gurdonark-kindergarten');
```

```
    game.slingshotReleasedSound = loader.loadSound("audio/released");
    game.bounceSound = loader.loadSound('audio/bounce');
    game.breakSound = {
        "glass":loader.loadSound('audio/glassbreak'),
        "wood":loader.loadSound('audio/woodbreak')
    };

    // Hide all game layers and display the start screen
    $('.gamelayer').hide();
    $('#gamestartscreen').show();

    //Get handler for game canvas and context
    game.canvas = document.getElementById("gamecanvas");
    game.context = game.canvas.getContext('2d');
},
```

The code loads the different sound files using the `loader.loadSound()` method and saves them for later reference. We store the break sounds in an associative array so that we can easily add sounds for more entities and reference them by name. The background music is an excellent Creative Common–licensed tune called "Kindergarten" by Gurdonark.

■ **Tip** You can find some amazing free music for your own games at the ccMixter website, located at http://www.ccmixter.com.

Adding Break and Bounce Sounds

Now that we have loaded these sounds, we need to associate these sound effects with the entities and play them at the right time. We will modify the `entities.create()` method and set the break and bounce sounds in the entity definitions, as shown in Listing 4-30.

Listing 4-30. Assigning Sounds to Entities During Creation

```
create:function(entity){
    var definition = entities.definitions[entity.name];
    if(!definition){
        console.log ("Undefined entity name",entity.name);
        return;
    }
    switch(entity.type){
        case "block": // simple rectangles
            entity.health = definition.fullHealth;
            entity.fullHealth = definition.fullHealth;
            entity.shape = "rectangle";
            entity.sprite = loader.loadImage("images/entities/"+entity.name+".png");
            entity.breakSound = game.breakSound[entity.name];
            box2d.createRectangle(entity,definition);
            break;
```

```
        case "ground": // simple rectangles
            // No need for health. These are indestructible
            entity.shape = "rectangle";
            // No need for sprites. These won't be drawn at all
            box2d.createRectangle(entity,definition);
            break;
        case "hero":    // simple circles
        case "villain": // can be circles or rectangles
            entity.health = definition.fullHealth;
            entity.fullHealth = definition.fullHealth;
            entity.sprite = loader.loadImage("images/entities/"+entity.name+".png");
            entity.shape = definition.shape;
            entity.bounceSound = game.bounceSound;
            if(definition.shape == "circle"){
                entity.radius = definition.radius;
                box2d.createCircle(entity,definition);
            } else if(definition.shape == "rectangle"){
                entity.width = definition.width;
                entity.height = definition.height;
                box2d.createRectangle(entity,definition);
            }
            break;
        default:
            console.log("Undefined entity type",entity.type);
            break;
    }
},
```

The advantage of attaching sounds to entities during creation like this is that every entity can have its own custom "break" sound and "bounce" sound. Now, all we need to do is play the sounds when the events actually occur. First, we play the bounce sound whenever we detect a collision, inside the `b2ContactListener` object we defined earlier, as shown in Listing 4-31.

Listing 4-31. Playing the Bounce Sound During a Collision

```
var listener = new Box2D.Dynamics.b2ContactListener;
listener.PostSolve = function(contact,impulse){
    var body1 = contact.GetFixtureA().GetBody();
    var body2 = contact.GetFixtureB().GetBody();
    var entity1 = body1.GetUserData();
    var entity2 = body2.GetUserData();

    var impulseAlongNormal = Math.abs(impulse.normalImpulses[0]);
    // This listener is called a little too often. Filter out very tiny impulses.
    // After trying different values, 5 seems to work well
    if(impulseAlongNormal>5){
        // If objects have a health, reduce health by the impulse value
        if (entity1.health){
            entity1.health -= impulseAlongNormal;
        }
```

```
        if (entity2.health){
            entity2.health -= impulseAlongNormal;
        }

        // If objects have a bounce sound, play the sound
        if (entity1.bounceSound){
            entity1.bounceSound.play();
        }

        if (entity2.bounceSound){
            entity2.bounceSound.play();
        }
    }
};
```

During a collision, we check if the entity has a `bounceSound` property defined and, if so, we play the sound. If we define bounce sounds for any entity, this code will automatically play it. Next, we play the break sound any time an object gets destroyed, inside the `drawAllBodies()` method of the `game` object (see Listing 4-32).

Listing 4-32. Playing the Break Sound when an Object Is Destroyed

```
drawAllBodies:function(){
    box2d.world.DrawDebugData();

    // Iterate through all the bodies and draw them on the game canvas
    for (var body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();

        if(entity){
            var entityX = body.GetPosition().x*box2d.scale;
            if(entityX<0|| entityX>game.currentLevel.foregroundImage.width||(entity.health &&
entity.health <0)){
                box2d.world.DestroyBody(body);
                if (entity.type=="villain"){
                    game.score += entity.calories;
                    $('#score').html('Score: '+game.score);
                }
                if (entity.breakSound){
                    entity.breakSound.play();
                }
            } else {
                entities.draw(entity,body.GetPosition(),body.GetAngle())
            }
        }
    }
},
```

Again, we check to see if the entity being destroyed has a `breakSound` property and, if so, we play the sound. So far we have defined break sounds for the glass and wood blocks, but we can easily extend the code to add sounds for the other entities.

Finally, we play the `slingshotReleasedSound` when `game.mode` changes from `firing` to `fired` inside the `handlePanning()` method (see Listing 4-33).

Listing 4-33. Playing the Slingshot Released Sound when the Hero Is Fired

```
if (game.mode == "firing"){
    if(mouse.down){
        game.panTo(game.slingshotX);

game.currentHero.SetPosition({x:(mouse.x+game.offsetLeft)/box2d.scale,y:mouse.y/box2d.scale});
    } else {
        game.mode = "fired";
        game.slingshotReleasedSound.play();
        var impulseScaleFactor = 0.75;
        // Coordinates of center of slingshot (where the band is tied to slingshot)
        var slingshotCenterX = game.slingshotX + 35;
        var slingshotCenterY = game.slingshotY+25;
        var impulse = new b2Vec2((slingshotCenterX -mouse.x-game.offsetLeft)*impulseScaleFactor,
(slingshotCenterY-mouse.y)*impulseScaleFactor);
        game.currentHero.ApplyImpulse(impulse, game.currentHero.GetWorldCenter());

    }
}
```

Now when you run the game, you should hear sound effects when the hero is fired, when it bumps against something, or when the blocks get destroyed. The last thing we will be adding is the background music.

Adding Background Music

We have already loaded the background music file along with the other sound files in the `game.init()` method. Now we need to create a few methods for starting, stopping, and toggling the background music. We will add these methods to the `game` object, as shown in Listing 4-34.

Listing 4-34. Methods for Controlling Background Music

```
startBackgroundMusic:function(){
    var toggleImage = $("#togglemusic")[0];
    game.backgroundMusic.play();
    toggleImage.src="images/icons/sound.png";
},
stopBackgroundMusic:function(){
    var toggleImage = $("#togglemusic")[0];
    toggleImage.src="images/icons/nosound.png";
    game.backgroundMusic.pause();
    game.backgroundMusic.currentTime = 0; // Go to the beginning of the song
},
toggleBackgroundMusic:function(){
    var toggleImage = $("#togglemusic")[0];
    if(game.backgroundMusic.paused){
        game.backgroundMusic.play();
        toggleImage.src="images/icons/sound.png";
    } else {
        game.backgroundMusic.pause();
```

```
        $("#togglemusic")[0].src="images/icons/nosound.png";
    }
},
```

The `startBackgroundMusic()` method first calls the `backgroundMusic object's play()` method and then sets the toggle music button image to show a speaker with sound coming out.

The `stopBackgroundMusic()` method sets the toggle music button image to show a speaker with no sound. It then calls the `backgroundMusic` object's `pause()` method and sets the audio back to the beginning of the song by setting its `currentTime` property to 0.

Finally, the `toggleBackgroundMusic()` method checks to see whether or not the music is currently paused, calls either the `pause()` or `play()` method, and then sets the toggle image appropriately.

Now that we have these methods in place, we need to call them. We will call the `startBackgroundMusic()` method when the game starts from inside the `game.start()` method, as shown in Listing 4-35.

Listing 4-35. Starting the Background Music

```
start:function(){
    $('.gamelayer').hide();
    // Display the game canvas and score
    $('#gamecanvas').show();
    $('#scorescreen').show();

    game.startBackgroundMusic();

    game.mode = "intro";
    game.offsetLeft = 0;
    game.ended = false;
    game.animationFrame = window.requestAnimationFrame(game.animate,game.canvas);
},
```

Next, we will call the `stopBackgroundMusic()` method whenever the level ends by adding it to the `showEndingScreen()` method, as shown in Listing 4-36.

Listing 4-36. Stopping the Background Music

```
showEndingScreen:function(){
    game.stopBackgroundMusic();
    if (game.mode=="level-success"){
        if(game.currentLevel.number<levels.data.length-1){
            $('#endingmessage').html('Level Complete. Well Done!!!');
            $("#playnextlevel").show();
        } else {
            $('#endingmessage').html('All Levels Complete. Well Done!!!');
            $("#playnextlevel").hide();
        }
    } else if (game.mode=="level-failure"){
        $('#endingmessage').html('Failed. Play Again?');
        $("#playnextlevel").hide();
    }

    $('#endingscreen').show();
},
```

Finally, we will call the `toggleBackgroundMusic()` method from the `onclick` event of the toggle music button inside the `scorescreen` layer, as shown in Listing 4-37.

Listing 4-37. Toggling the Background Music

```
<div id="scorescreen" class="gamelayer">
    <img id="togglemusic" src="images/icons/sound.png" onclick="game.toggleBackgroundMusic()">
    <img src="images/icons/prev.png" onclick="game.restartLevel();">
    <span id="score">Score: 0</span>
</div>
```

Now when we run the game, the background music starts playing every time a level starts. If we click the toggle button, the music pauses and the button changes to the no-sound icon, as shown in Figure 4-8.

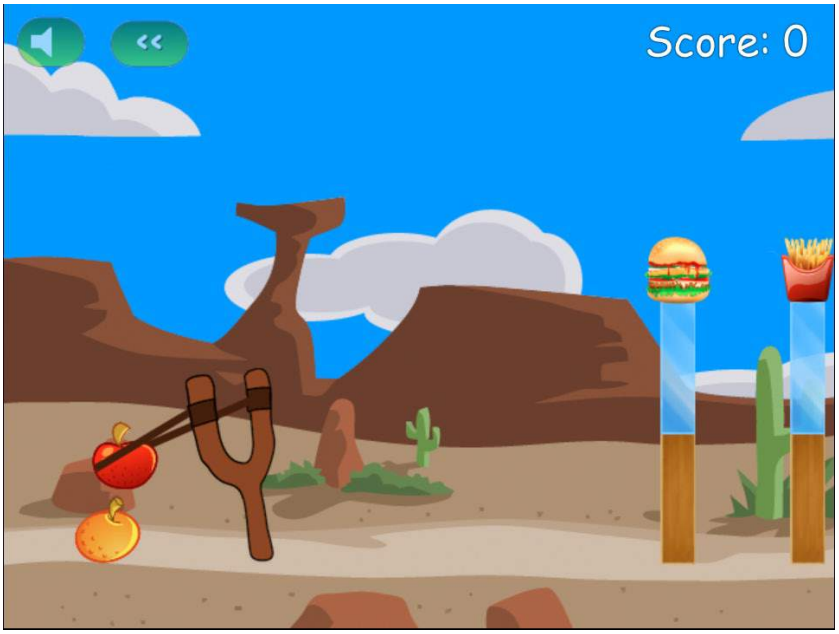

Figure 4-8. *The finished game with the background music switched off*

With this last change, our game is finally complete. We now can select a level and play the game by slinging across the hero fruits to attack the evil junk food while listening to sound effects and background music. Take some time to enjoy the game and come up with your own ideas for levels.

Summary

Over the past three chapters, we created our first physics engine–based HTML5 game. We started in Chapter 2 by creating a basic game framework with menus, a level system, and an asset loader and setting up game animation. We then covered the basics of Box2D in Chapter 3. Finally, in this chapter we integrated Box2D into our existing game framework and wrapped up our game by adding menu options, sounds effects, and music.

Of course, there is still a lot of room for us to expand the functionality of this game. Some of the obvious next steps would be to add animations for different entities, add more levels, tweak the game physics parameters, and add more heroes and villains with different characteristics.

However, the game has all the essential elements that people have come to expect from a good HTML5 game. You can use the code in this game as a starting point for any of your own physics engine–based games and take it wherever you would like.

Now that we have made our first game, we will take on a slightly more challenging project in the next few chapters: building a complete, multiplayer, real-time strategy game. So let's keep going.