

CREACIÓN DE UN JUEGO CON HTML5

Liliana Patricia Santacruz Valencia

Edificio Ampliación Rectorado, 2015B

liliana.santacruz@urjc.es



Contenido

- Creación del mundo básico del juego.
- Fundamentos del motor de la física.
- Integración del motor de la física.

CREACIÓN DEL MUNDO BÁSICO DEL JUEGO

Introducción

- Auge de los dispositivos que soportan juegos basados en rompecabezas simples y física.
- Se pueden jugar en cortos periodos de tiempo.
- *Angry Birds (Rovio Entertainment)*:
 - Rompecabezas/juego de estrategia.
 - Honda para disparar a los pájaros y a los cerdos enemigos.
 - Utiliza motor de la física para modelar las colisiones, los disparos, etc.

Básicos (I)

- Canvas

"El elemento lienzo proporciona scripts con un lienzo de mapa de bits dependiente de la resolución, que se puede usar para renderizar gráficos, gráficos de juegos u otras imágenes visuales sobre la marcha".

<https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element>

- Permite dibujar formas primitivas como **líneas**, *círculos* y **rectángulos**, así como **imágenes** y **texto**, optimizado para un dibujo rápido.
- Los navegadores han comenzado a habilitar la **renderización acelerada** para que los juegos y animaciones se ejecuten rápidamente.

Básicos (II)

```

1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta http-equiv="Content-type" content="text/html; charset=utf-8">
5      <title>Ejemplo de página HTML</title>
6      </script>
7    </head>
8    <body onload="pageLoaded();">
9      <canvas width="600" height="400" id="testcanvas" style="border:1px solid black;">
10        Tu navegador no soporta la etiqueta Canvas de HTML5. Por favor, utiliza otro navegador.
11      </canvas>
12    </body>
13  </html>

```

Básicos (III)

```

3      <head>
4          <meta http-equiv="Content-type" content="text/html; charset=utf-8">
5          <title>Ejemplo de archivo HTML5</title>
6          <script type="text/javascript" charset="utf-8">
7              function pageLoaded(){
8                  // Obtener el manejador del objeto canvas
9                  var canvas = document.getElementById('testcanvas');
10
11                  // Obtener el contexto 2D para este canvas
12                  var context = canvas.getContext('2d');
13
14                  // Nuestro código de los dibujos ...
15              }
16          </script>
17      </head>
18      <body onload="pageLoaded();">
19          <canvas width="600" height="400" id="testcanvas" style="border:1px solid black;">
20              Tu navegador no soporta la etiqueta Canvas de HTML5. Por favor, utiliza otro navegador.
21          </canvas>
22      </body>
23  </html>

```

```

16 // Dibujar rectángulos
17
18 // Rectángulos rellenos
19 // Dibujar un cuadrado sólido con ancho y alto de 100 pixels en (200,10)
20 context.fillRect(200,10,100,100);
21 // Dibujar un cuadrado sólido con ancho de 90 y alto 30 pixels en (50,70)
22 context.fillRect(50,70,90,30);
23
24 // Contornos rectangulares
25 // Dibujar un contorno rectangular se ancho y alto de 50 pixels en (110,10)
26 context.strokeRect(110,10,50,50);
27 // Dibujar un contorno rectangular se ancho y alto de 50 pixels en (30,10)
28 context.strokeRect(30,10,50,50);
29
30 // Otros rectángulos
31 // Rectángulo de 30 píxeles de ancho y 20 de alto en (210,20)
32 context.clearRect(210,20,30,20);
33 // Rectángulo de 30 píxeles de ancho y 20 de alto en (260,20)
34 context.clearRect(260,20,30,20);
35
36
37 // Dibujar formas complejas
38 // Triángulo relleno
39 context.beginPath();
40 context.moveTo(10,120); // Comenzar a dibujar en 25,25
41 context.lineTo(10,180);
42 context.lineTo(110,150);
43 context.fill(); // Cerrar la forma y rellenarla
44
45 // Contornos triangulares
46 context.beginPath();
47 context.moveTo(140,160); // comenzar a dibujar en 110,160
48 context.lineTo(140,220);
49 context.lineTo(40,190);
50 context.closePath();
51 context.stroke();
52
53 // Otras líneas complejas ...
54 context.beginPath();
55 context.moveTo(160,160); // comenzar a dibujar en 110,160
56 context.lineTo(170,220);
57 context.lineTo(240,210);
58 context.lineTo(260,170);
59 context.lineTo(190,140);
60 context.closePath();
61 context.stroke();

```



```

63 // Dibujar arcos
64
65 // Dibujar semicírculos
66 context.beginPath();
67 // Dibujar un arco en (400,50) con radio 40 de 0 a 180 grados, contrareloj
68 context.arc(100,300,40,0,Math.PI,true); // (PI radianes = 180 grados)
69 context.stroke();
70
71 // Dibujar un círculo completo
72 context.beginPath();
73 // Dibujar un arco en (500,50) con radio de 30 de 0 a 360 grados, contrareloj
74 context.arc(100,300,30,0,2*Math.PI,true); // (2*PI radianes = 360 grados)
75 context.fill();
76
77 // Dibujar un arco de tres cuartos
78 context.beginPath();
79 // Dibujar un arco en (400,100) con radio de 25 de 0 a 270 grados, sentido reloj
80 context.arc(200,300,25,0,3/2*Math.PI,false); // (3/2*PI radianes = 270 grados)
81 context.stroke();
82
83 // Dibujar texto
84 context.fillText('This is some text...',330,40);
85
86 // Modificar fuentes
87 context.font = '10pt Arial';
88 context.fillText('This is in 10pt Arial...',330,60);
89
90 // Dibujar contorno de texto
91 context.font = '16pt Arial';
92 context.strokeText('This is stroked in 16pt Arial...',330,80);
93
94 // Dibujar con colores y transparencias
95 // Fijar el color a rojo
96 context.fillStyle = "red";
97 // Dibujar un rectángulo relleno de color rojo
98 context.fillRect (310,160,100,50);
99
100 // Fijar el color del contorno a verde
101 context.strokeStyle = "green";
102 // Dibujar un rectángulo relleno con verde
103 context.strokeRect (310,240,100,50);
104
105 // Fijar el color de relleno a rojo utilizando rgb()
106 context.fillStyle = "rgb(255,0,0)";
107 // Dibujar un rectángulo relleno rojo
108 context.fillRect (420,160,100,50);
109
110 // Fijar el color a verde con transparencia a 0.5
111 context.fillStyle = "rgba(0,255,0,0.6)";
112 // Dibujar un rectángulo verde semitransparente
113 context.fillRect (450,180,100,50);

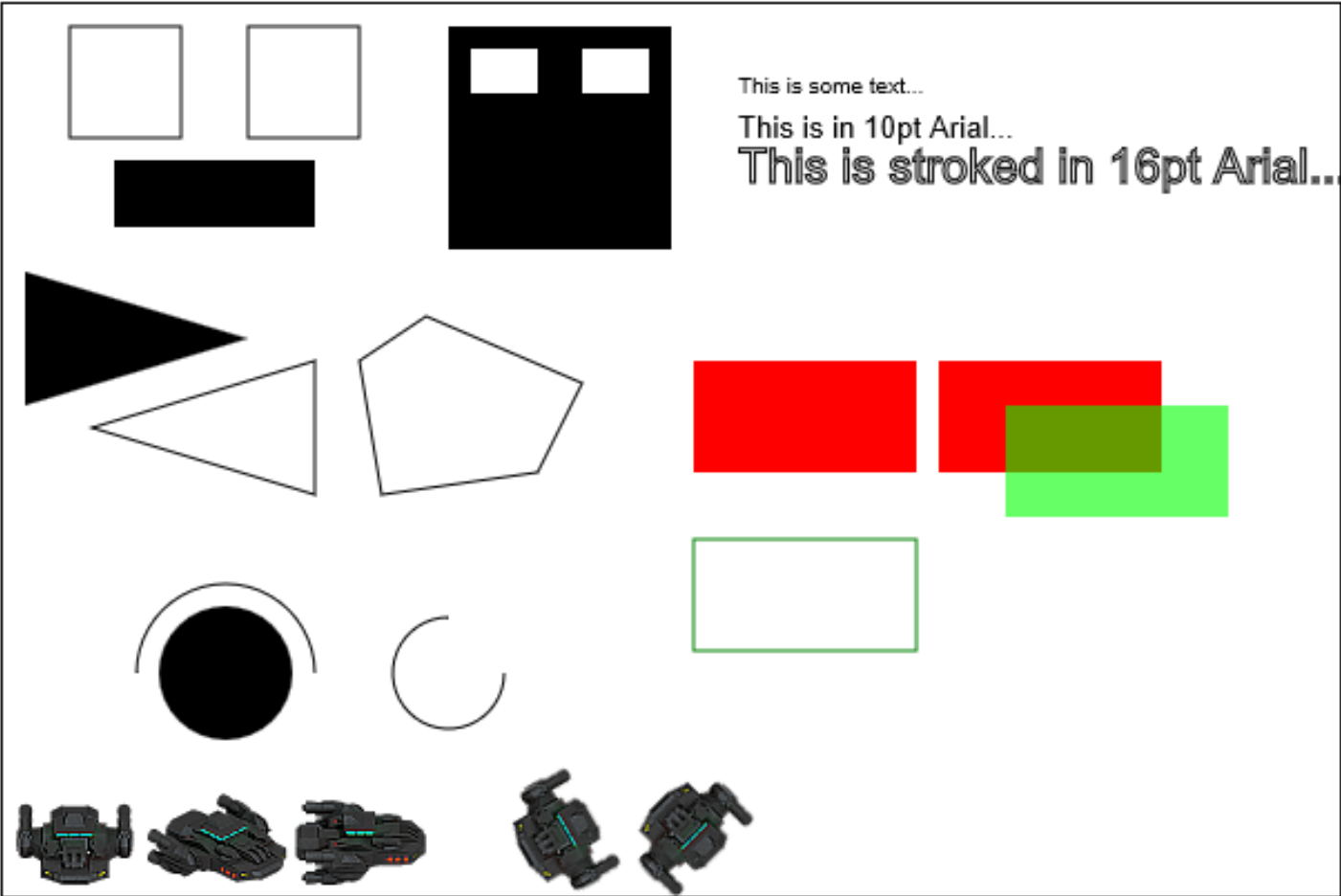
```

```

115 // Dibujar imágenes
116
117 // Obtener un manejador para el bojeeto imagen
118 var image = document.getElementById('spaceship');
119
120 // Dibujar la imagen en (0,350)
121 context.drawImage(image,0,350);
122
123 // Escalar la imagen a la mitad del tamaño original
124 context.drawImage(image,0,400,100,25);
125
126 // Dibujar parte de la imagen
127 context.drawImage(image,0,0,60,50,0,420,60,50);
128
129
130
131 //Transformación
132
133
134 //Transladar el origen a la ubicación del objeto
135 context.translate(250,370);
136 //Rotar al rededor del origen 60 grados
137 context.rotate(Math.PI/3);
138 context.drawImage(image,0,0,60,50,-30,-25,60,50);
139 //Restablecer el estado original
140 context.rotate(-Math.PI/3);
141 context.translate(-240,-370);
142
143 //Transladar el origne a la ubicación del objeto
144 context.translate(300,370);
145 //Rotar al rededor del nuevo origen
146 context.rotate(3*Math.PI/4);
147 context.drawImage(image,0,0,60,50,-30,-25,60,50);
148 //Restablecer el estado original
149 context.rotate(-3*Math.PI/4);
150 context.translate(-300,-370);
151
152 }
153 </script>
154 </head>
155 <body onload="pageLoaded();">
156   <canvas width="600" height="400" id="testcanvas" style="border:1px solid black;">
157     Tu navegador no soporta la etiqueta Canvas de HTML5. Por favor, utiliza otro navegador.
158   </canvas>
159   
160 </body>
161 </html>

```

Visualización



Objetivo

- Construir un juego de rompecabezas basado en física con niveles.
- Las frutas son las protagonistas.
- La comida chatarra es el enemigo.
- Algunas estructuras rompibles dentro del nivel.
- **Implementar:**
 - ✓ Pantalla de bienvenida.
 - ✓ Pantalla de carga y pre-cargadores.
 - ✓ Pantallas de menú.
 - ✓ *Parallax scrolling* (desplazamiento).
 - ✓ Sonido.
 - ✓ Motor de la física.
 - ✓ Marcador.

Diseño básico (I)

- **Pantalla de bienvenida:** Muestra cuando se carga la página del juego.
- **Pantalla de inicio:** Un menú que permite al jugador comenzar el juego o modificar las opciones de configuración.
- **Pantalla de carga/progreso:** Muestra cuando el juego el juego está cargando los *Assets* (imágenes y sonidos).
- **Canvas del juego:** La capa actual del juego.
- **Marcador:** Se muestra sobre el Canvas, con unos pocos botones y el puntaje.
- **Pantalla final:** Una pantalla que se muestra al final de cada nivel.

Diseño básico (II)

- Cada capa se implementa definiendo un `<div>` o un `<canvas>` que se muestra o se esconde según se requiera.
- Se utilizará:
 - JQuery
 - JavaScript
 - CSS

Pantalla de bienvenida y menu principal

```
<!DOCTYPE html>
<html>
  <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/game.js" type="text/javascript" charset="utf-8"></script>
    <link rel="stylesheet" href="styles.css" type="text/css" media="screen" charset="utf-8">
  </head>

  <body>
    <div id="gamecontainer">
      <canvas id="gamecanvas" width="640" height="480" class="gamelayer">
      </canvas>

      <div id="scorescreen" class="gamelayer">
        
        
        <span id="score">Score: 0</span>
      </div>

      <div id="gamestartscreen" class="gamelayer">
        <br>
        
      </div>

      <div id="levelselectscreen" class="gamelayer">
      </div>

      <div id="loadingscreen" class="gamelayer">
        <div id="loadingmessage"></div>
      </div>

      <div id="endingscreen" class="gamelayer">
        <div>
          <p id="endingmessage">The Level Is Over Message</p>
          <p id="playcurrentlevel"> Replay Current Level</p>
          <p id="playnextlevel"> Play Next Level </p>
          <p id="showLevelScreen"> Return to Level Screen</p>
        </div>
      </div>
    </div>
  </body>
</html>
```


**Estilos para el contenedor
del juego y pantalla de inicio**

```
#gamecontainer {
  1 width:640px;
  height:480px;
  2 background: url(images/splashscreen.png);
  border: 1px solid black;
}
```

```
.gamelayer {
  1 width:640px;
  height:480px;
  3 position:absolute;
  display:none;
}
```

/ Pantalla de inicio del juego */*

```
#gamestartscreen {
  padding-top:250px;
  text-align:center;
}
```

```
#gamestartscreen img {
  margin:10px;
  cursor:pointer;
}
```

1. Definir el contenedor del juego y las dimensiones de todas las capas

2. Fijar la imagen de pantalla de bienvenida y el fondo del contenedor

3. Asegurar que todas las capas del juego se ubican en posición absoluta

- Una encima de otra
- Ocultas por defecto

Añadir estilo a la pantalla de inicio

Pantalla de bienvenida



Figura 1. Pantalla de bienvenida del juego ***Froot Wars***

**JavaScript para mostrar el
menu principal, la pantalla de
carga y el juego**

game.js

Crear un archivo llamado **game.js**

```
$(window).load(function() {  
    game.init();  
});
```

3

3. El método **game.init()** es necesario para evitar comportamientos inesperados, tales como errores JavaScript

```
var game = {  
    // Comienza inicializando los objetos, precargando los assets y mostrando la pantalla de inicio  
    1 init: function(){  
        // Inicializa objetos  
        2 levels.init();  
        loader.init();  
        mouse.init();  
  
        // Oculta todas las capas del juego y muestra la pantalla de inicio  
        1 $('.gamelayer').hide();  
        $('#gamestartscreen').show();  
  
        // Obtener el controlador para el canvas y el contexto del juego  
        game.canvas = $('#gamecanvas')[0];  
        game.context = game.canvas.getContext('2d');  
    },  
}
```

2. Llamamos a **level.init()** desde **game.init()** para generar los botones de la pantalla de selección

Pantalla de bienvenida y menú de opciones



Figura 2. Pantalla de bienvenida y menú de opciones

Selección del nivel

Proceso

- Una vez que el usuario pulse sobre el botón **Play** debe acceder a la pantalla de selección de nivel, para ver los niveles disponibles.
- Previamente, es necesario crear un objeto **levels** para manejar los niveles.
 - Datos y funciones para controlar la inicialización.
 - Se crea dentro de **game.js** y antes del objeto **game**.
- Dentro de **init()** cambiar:

//Obtener el manejador para el canvas y el contexto

```
game.canvas = document.getElementById('gamecanvas');
```

por:

```
game.canvas = $('#gamecanvas')[0];
```



```

var levels = {
  // Nivel de datos
  1 data:[
    { // Primer nivel
      foreground:'desert-foreground',
      background:'clouds-background',
      entities:[]
    },
    { // Segundo nivel
      foreground:'desert-foreground',
      background:'clouds-background',
      entities:[]
    }
  ],
  // Inicializa la pantalla de selección de nivel
  2 init:function(){
    var html = "";
    for (var i=0; i < levels.data.length; i++) {
      var level = levels.data[i];
      html += '<input type="button" value="'+(i+1)+'">';
    };
    $('#levelselectscreen').html(html);

    // Establece los controladores de eventos de clic de botón para cargar el nivel
    $('#levelselectscreen input').click(function(){
      3 levels.load(this.value-1);
      $('#levelselectscreen').hide();
    });
  },

  // carga todos los datos e imágenes para un nivel específico
  load:function(number){
  }
}

```

1. El objeto `levels` tiene un *array* con información acerca de cada nivel

2. Genera dinámicamente los botones para cada nivel

3. Los controladores de eventos de clic de botón de nivel llaman, al `load()` para cada nivel y luego esconden la pantalla de selección de nivel

Init()

```
init: function(){
    // Inicializar objetos
    levels.init();
    loader.init();
    mouse.init();

    // Ocultar todas las capas del juego y mostrar la pantalla de inicio
    $('.gamelayer').hide();
    $('#gamestartscreen').show();

    //Obtener manejador para el canvas del juego y el contexto
    game.canvas = $('#gamecanvas')[0];
    game.context = game.canvas.getContext('2d');
},
```

Estilos para pantalla de nivel de selección

```
/* Pantalla de selección de nivel*/
#levelselectscreen {
    padding-top:150px;
    padding-left:50px;
}

#levelselectscreen input {
    margin:20px;
    cursor:pointer;
    background:url(images/icons/level.png) no-repeat;
    color:yellow;
    font-size: 20px;
    width:64px;
    height:64px;
    border:0;
}
```

game.showLevelScreen()

- Dentro del objeto **game**, crear el método **game.showLevelScreen()**
 - Oculta la pantalla del menú principal y muestra la pantalla de selección de nivel.

```
showLevelScreen:function(){
  1  $('.gamelayer').hide();
  2  $('#levelselectscreen').show('slow');
},|
```

1. Primero oculta todas las otras capas del juego

Muestra la capa de **levelselectscreen** utilizando una animación **slow**

game.showLevelScreen()

- Finalmente se llama a `game.showLevelScreen()` cuando el usuario pulsa el botón **Play** desde el evento **onclick** de la imagen.

```

```

- De esta forma, al pulsar el botón Play:
 - El juego detectará el número del nivel.
 - Ocultará el menú principal.
 - Y mostrará los botones para cada uno de los niveles.

Pantalla de selección de nivel



Figura 3. Pantalla de selección de nivel

Cargar imágenes

Proceso (I)

- Antes de implementar los niveles en sí, es necesario ubicar el cargador de imágenes y la pantalla de carga.
 - Cargar las imágenes para un nivel.
 - Comenzar el juego una vez cargados los *assets*.
- Se diseña una pantalla de carga que contiene un GIF animado con una imagen de la barra de progreso y un texto indicando el número de imágenes cargadas.
- Para ello se añade el siguiente código al archivo **styles.css**.

CSS pantalla de carga

```
/* Pantalla de carga */  
#loadingscreen {  
    background: rgba(100,100,100,0.3);  
}
```

Añade un color gris oscuro sobre el fondo para permitirle al usuario saber que actualmente hay algo en proceso y que no está listo para recibir ninguna entrada del usuario

```
#loadingmessage {  
    margin-top: 400px;  
    text-align: center;  
    height: 48px;  
    color: white;
```

Muestra un mensaje de carga en color blanco

```
background-color: #333; color: white; text-align: center;  
font-size: 12px; font-family: Arial;  
}
```

Cargador de *assets* (I)

Definir el objeto loader dentro de game.js

```
var loader = {
  loaded:true,
  loadedCount:0, // Assets que han sido cargados antes
  totalCount:0, // Número total de assets que es necesario cargar
```

```
  init:function(){
    // Comprueba el soporte para sonido
    var mp3Support,oggSupport;
    var audio = document.createElement('audio');
    if (audio.canPlayType) {
      // Actualmente canPlayType() devuelve: "", "maybe" o "probably"
      mp3Support = "" != audio.canPlayType('audio/mpeg');
      oggSupport = "" != audio.canPlayType('audio/ogg; codecs="vorbis"');
    } else {
      //La etiqueta de audio no es soportada
      mp3Support = false;
      oggSupport = false;
    }

    // Comprueba para ogg, mp3 y finalmente fija soundFileExtn como undefined
    loader.soundFileExtn = oggSupport?".ogg":mp3Support?".mp3":undefined;
  },
```

```
  loadImage:function(url){
    this.totalCount++;
    this.loaded = false;
    $('#loadingscreen').show();
    var image = new Image();
    image.src = url;
    image.onload = loader.itemLoaded;
    return image;
  },
```

loadImage() carga las imágenes

Incrementa la variable totalCount

Muestra la pantalla de carga cuando se invoca

Cargador de *assets* (II)

loadSound() carga los sonidos

Incrementa la variable totalCount

Muestra la pantalla de carga cuando se invoca

```
soundFileExtn:".ogg",
loadSound:function(url){
    this.totalCount++;
    this.loaded = false;
    $('#loadingscreen').show();
    var audio = new Audio();
    audio.src = url+loader.soundFileExtn;
    audio.addEventListener("canplaythrough", loader.itemLoaded, false);
    return audio;
},
itemLoaded:function(){
    loader.loadedCount++;
    $('#loadingmessage').html('Loaded '+loader.loadedCount+' of '+loader.totalCount);
    if (loader.loadedCount === loader.totalCount){
        // El loader ha cargado completamente..
        loader.loaded = true;
        // Oculta la pantalla de carga
        $('#loadingscreen').hide();
        //Y llama al método loader.onload si este existe
        if(loader.onload){
            loader.onload();
            loader.onload = undefined;
        }
    }
}
```

Inicializar el cargador

- Después de utilizar el `loader`, es necesario llamar al método `loader.init()` desde el `game.init()` para que el cargador se inicialice cuando el juego esté inicializado.

```
init: function(){  
    // Inicializa objetos  
    levels.init();  
    loader.init();  
    mouse.init();  
  
    // Oculta todas las capas del juego y muestra la pantalla de inicio  
    $('.gamelayer').hide();  
    $('#gamestartscreen').show();  
  
    // Obtener el controlador para el canvas y el contexto del juego  
    game.canvas = $('#gamecanvas')[0];  
    game.context = game.canvas.getContext('2d');  
},
```

- Se utilizar el `loader` al llamar a cualquiera de los dos métodos `loadImage()` o `loadSound()`.
- La pantalla mostrará el progreso de carga, hasta que se carguen todas las imágenes y sonidos.

Pantalla de carga



Figura 4. Pantalla de carga

Cargar niveles

Proceso

- Cargar el fondo, el primer plano y las imágenes de la honda.
- Mediante la definición `load()` dentro del objeto `levels`.

Carga de fondo, primer plano y honda

// Cargar todos los datos e imágenes para un nivel específico

```
load:function(number){
```

```
    // declarar un nuevo objeto de nivel actual
    game.currentLevel = {number:number,hero:[]};
    game.score=0;
    $('#score').html('Score: '+game.score);
    var level = levels.data[number];
```

//Cargar el fondo, el primer plano y las imágenes de la honda

```
    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");
```

//Llamar a game.start() cuando los assets se hayan cargado

```
    if(loader.loaded){
        game.start()
    } else {
        loader.onload = game.start;
    }
}
```

La función `load()` crear el objeto `current level` para almacenar los datos del nivel cargado. La utilizaremos también para cargar los demás elementos del juego

`start()` es donde el juego actual será dibujado

Animación del juego

Proceso (I)

- Se llama al código de dibujar y animar muchas veces por segundo utilizando **`requestAnimationFrame`**.
- Se debe colocar al comienzo del archivo **`game.js`**.

requestAnimationFrame

```
// Preparar requestAnimationFrame y cancelAnimationFrame para su uso en el código del juego
(function() {
    var lastTime = 0;
    var vendors = ['ms', 'moz', 'webkit', 'o'];
    for(var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x) {
        window.requestAnimationFrame = window[vendors[x]+'RequestAnimationFrame'];
        window.cancelAnimationFrame =
            window[vendors[x]+'CancelAnimationFrame'] || window[vendors[x]+'CancelRequestAnimationFrame'];
    }

    if (!window.requestAnimationFrame)
        window.requestAnimationFrame = function(callback, element) {
            var currTime = new Date().getTime();
            var timeToCall = Math.max(0, 16 - (currTime - lastTime));
            var id = window.setTimeout(function() { callback(currTime + timeToCall); },
                timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };

    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
            clearTimeout(id);
        };
})();
```

Proceso (II)

- Se utiliza `game.start()` para configurar el bucle de animación y luego dibujar el nivel dentro de `game.animate()`.

```
// Modo Game
mode:"intro",
// Coordenadas X & Y de la honda
slingshotX:140,
slingshotY:280,
start:function(){
    $('.gamelayer').hide();
    // Mostrar el canvas del juego y la puntuación
    $('#gamecanvas').show();
    $('#scorescreen').show();

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

start() inicializa
variables necesarias en el
juego

mode almacena el
estado actual del juego

Fija el intervalo de animación del
juego para llamar a la
función animate() utilizando
window.requestAnimationFrame

```
handlePanning:function(){
    game.offsetLeft++; //Marcador de posición temporal- mantiene la panorámica a la derecha
};
```

```
animate:function(){
    // Anima el fondo
    game.handlePanning();

    // Anima los personajes
```

animate() realiza toda
la animación y el dibujo
dentro del juego

Comprueba si el juego ha
terminado, si no es así utiliza
requestAnimationFrame para
llamar de nuevo a anim

```
// Dibuja el fondo con desplazamiento (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);
```

Se mueven a diferentes velocidades

```
// Dibuja la honda
game.context.drawImage(game.slingshotImage,game.slingshotX-game.offsetLeft,game.slingshotY);

game.context.drawImage(game.slingshotFrontImage,game.slingshotX-game.offsetLeft,game.slingshotY);

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

CSS para puntuación

```
/* Pantalla de puntuación */
#scorescreen {
    height:60px;
    font: 32px Comic Sans MS;
    text-shadow: 0 0 2px #000;
    color:white;
}
```

La capa de puntuación es una banda delgada que aparecerá en la parte superior derecha de la pantalla

```
#scorescreen img{
    opacity:0.6;
    top:10px;
    position:relative;
    padding-left:10px;
    cursor:pointer;
}
```

Se añade un poco de transparencia para asegurar que no distrae del resto de elementos del juego.

```
#scorescreen #score {
    position:absolute;
    top:5px;
    right:20px;
}
```

```
/* Fin de pantalla */
endingscreen {
    text-align:center;
}
```

Pantalla del nivel 1 con puntuación

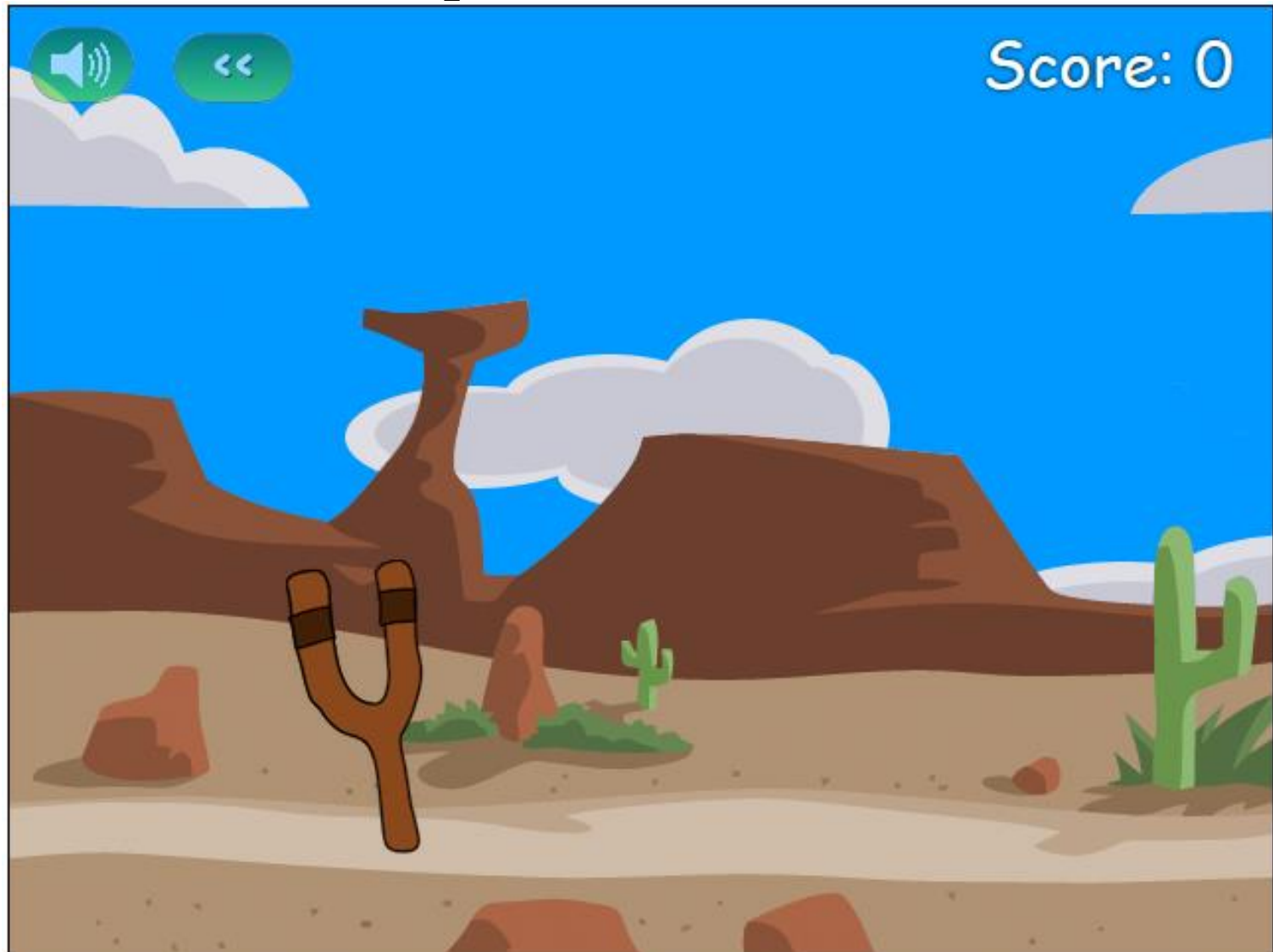


Figura 4. Pantalla del nivel 1 con puntuación inicial

Manejo de entradas por ratón

Proceso (I)

- Se identifican los eventos JavaScript que permiten capturar las entradas de ratón.
 - mousedown
 - mouseup
 - mousemove
- Se utiliza JQuery para simplificar la escritura del código.
 - Se crea un objeto **mouse** separado dentro de **game.js** .

```

var mouse = {
  x:0,
  y:0,
  down:false,
  init:function(){
    $('#gamecanvas').mousemove(mouse.mousemovehandler);
    $('#gamecanvas').mousedown(mouse.mousedownhandler);
    $('#gamecanvas').mouseup(mouse.mouseuphandler);
    $('#gamecanvas').mouseout(mouse.mouseuphandler);
  },
  mousemovehandler:function(ev){
    var offset = $('#gamecanvas').offset();

    mouse.x = ev.pageX - offset.left;
    mouse.y = ev.pageY - offset.top;

    if (mouse.down) {
      mouse.dragging = true;
    }
  },
  mousedownhandler:function(ev){
    mouse.down = true;
    mouse.downX = mouse.x;
    mouse.downY = mouse.y;
    ev.originalEvent.preventDefault();
  },
  mouseuphandler:function(ev){
    mouse.down = false;
    mouse.dragging = false;
  }
}

```

Init() fija los eventos para cuando se mueve, se presiona o se libera el botón del ratón

mousemovehandler()

- Utiliza `offset()` de JQuery.
- Las propiedades `pageX` y `pageY` del evento del objeto, para calcular las coordenadas X,Y del ratón respecto a la esquina superior izquierda del canvas y almacenarlas.
- Comprueba si el botón del ratón está presionado, entonces devuelve `true`.

mousedownhandler()

- Fija la variable `mouse.down` como `true`.
- Almacena la ubicación donde el botón del ratón fue presionado.

mouseuphandler()

- Fija las variables `down` y `dragging` como `false`.

Proceso (II)

- Igual que con los anteriores `init()`, se llama a este método desde `game.init()`.

```
init: function(){  
    // Inicializa objetos  
    levels.init();  
    loader.init();  
    mouse.init();  
  
    // Oculta todas las capas del juego y muestra la pantalla de inicio  
    $('.gamelayer').hide();  
    $('#gamestartscreen').show();  
  
    // Obtener el controlador para el canvas y el contexto del juego  
    game.canvas = $('#gamecanvas')[0];  
    game.context = game.canvas.getContext('2d');  
},
```

Definición de los estados del juego

Proceso (I)

- Se almacena el estado actual del juego en la variable `game.mode`.
- Algunos de los estados esperados en el juego son:
 - **intro**: El juego se desplazará alrededor del nivel para mostrarle al usuario todo lo que hay en él.
 - **load-next-hero**: El juego comprueba si hay otro héroe que cargar en la honda. El juego termina cuando ya no hay héroes o los villanos han sido destruidos.
 - **wait-for-firing**: El juego se desplaza hacia el área donde está la honda, en espera de que el usuario dispare al héroe.
 - **firing**: Tiene lugar cuando el usuario pulsa el héroe, pero antes de que libere el botón del ratón. Se prepara el ángulo de tiro y la altura a la que se lanzará el héroe.
 - **fired**: Sucede después de que el usuario libera el botón del ratón, entonces se habrá lanzado al héroe y el motor de la física actuará sobre todo, mientras el usuario sigue la trayectoria del héroe.

Proceso (II)

- Se pueden implementar más modos o estados.
- Solo es posible un estado a la vez.
- Se definen algunas condiciones para el código de desplazamiento.
- Dicho código debe ir dentro del objeto **game**, después de **start()**.

```

// Velocidad máxima de panoramización por fotograma en píxeles
maxSpeed:3,
// Mínimo y Máximo desplazamiento panorámico
minOffset:0,
maxOffset:300,
// Desplazamiento de panorámica actual
offsetLeft:0,
// La puntuación del juego
score:0,

//Desplegar la pantalla para centrarse en newCenter
panTo:function(newCenter){
    if (Math.abs(newCenter-game.offsetLeft-game.canvas.width/4)>0
        && game.offsetLeft <= game.maxOffset && game.offsetLeft >= game.minOffset){

        var deltaX = Math.round((newCenter-game.offsetLeft-game.canvas.width/4)/2);
        if (deltaX && Math.abs(deltaX)>game.maxSpeed){
            deltaX = game.maxSpeed*Math.abs(deltaX)/(deltaX);
        }
        game.offsetLeft += deltaX;
    } else {

        return true;
    }
    if (game.offsetLeft <game.minOffset){
        game.offsetLeft = game.minOffset;
        return true;
    } else if (game.offsetLeft > game.maxOffset){
        game.offsetLeft = game.maxOffset;
        return true;
    }
    return false;
},
handlePanning:function(){
    if(game.mode=="intro"){
        if(game.panTo(700)){
            game.mode = "load-next-hero";
        }
    }

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

    if (game.mode=="load-next-hero"){
        // TODO:
        // Comprobar si algún villano está vivo, si no, terminar el nivel (éxito)
        // Comprobar si quedan más héroes para cargar, si no terminar el nivel (fallo)
        // Cargar el héroe y fijar a modo de espera para disparar
        game.mode="wait-for-firing";
    }

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

    if (game.mode == "fired"){
        // TODO:
        // Hacer una panorámica donde quiera que el héroe se encuentre actualmente
    }
},

```

panTo()

-Desplaza suavemente la pantalla para dar la coordenada X y devuelve true si está cerca de la pantalla o si la pantalla se ha desplazado a la izquierda o a la derecha.

-Modela la velocidad de desplazamiento mediante maxSpeed.

Resumen

- Hasta este punto se ha:
 - Definido e implementado la pantalla de bienvenida y el menú de juego.
 - Creado un sistema de nivel simple y un cargador de *assets* para cargar dinámicamente las imágenes utilizadas por cada nivel.
 - Montado el lienzo de juego y el bucle de animación. Además hemos implementado el desplazamiento de paralaje para dar la ilusión de profundidad.
 - Utilizado estados de juego para simplificar el flujo y facilitar el movimiento alrededor del nivel.

Fundamentos del motor la física

Introducción

- El motor de la física es un programa que:
 - Proporciona una **simulación** aproximada del mundo de un juego.
 - Mediante la creación de un **modelo matemático** de todas las **interacciones** de los objetos y las **colisiones** dentro del juego.
 - Explica la gravedad, elasticidad, fricción y conservación del momentum entre objetos que chocan, demo do que los objetos se muevan en una forma creíble.
- Para **Froot wars**
 - Se utiliza el motor de la física Box2D
 - Box2DWeb
<http://code.google.com/box2dWeb>

Box2D

Fundamentos Box2D

- Box2D utiliza algunos objetos básicos para definir y simular el mundo del juego:
 - **World:** Es el objeto principal de Box2D, el cual contiene todos los objetos del mundo y simula la física del juego.
 - **Body:** Es un cuerpo rígido que puede consistir de una o más formas fijadas al cuerpo a través de accesorios.
 - **Shape:** Es una forma bidimensional, como un círculo o un polígono, los cuales son formas básicas.
 - **Fixture:** Se utiliza para unir una forma a un cuerpo para la detección de la colisión. Los accesorios mantienen datos adicionales no geométricos tales como fricción, colisión y filtros.
 - **Joint:** Se utiliza para limitar dos cuerpos en formas diferentes, por ejemplo, una articulación en la que dos cuerpos comparten un punto común mientras que son libres de rotar alrededor de ese punto. 60

Box2D en el juego

- Definir el mundo del juego.
- Añadir cuerpos y sus correspondientes formas utilizando accesorios.
- Mover los cuerpos alrededor utilizando Box2D.
- Dibujar los cuerpos después de cada paso.
- El objeto del mundo Box2D realiza la mayor parte del trabajo pesado.

Configurando Box2D

Crear un archivo nuevo llamado 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>
```

init()

-Inicializa el mundo Box2D y comienza la animación.

Box2DWeb-2.1.a.3.min.js

-Da acceso al objeto Box2D.

-Contiene todos los objetos necesarios, incluidos el mundo (Box2D.Dynamics.b2World) y el cuerpo (Box2D.Dynamics.b2Body).

Proceso (I)

- Crear un archivo llamado **box2d.js**.
- Declarar los objetos más comúnmente utilizados como variables.

```
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;
```

Proceso (II)

- El objeto **Box2D.Dynamics.b2World** es el corazón de **Box2D**.
- Contiene métodos para:
 - Añadir y eliminar objetos.
 - Simular la física a través de pasos incrementales.
 - Y una opción para dibujar el mundo sobre el canvas.
- Antes es necesario crea el objeto **b2World**.
 - En la función **init()** creada dentro del archivo **box2d.js**

Proceso (III)

```
var world;  
var scale = 30; //30 pixeles en el canvas equivalen a 1 metro en el mundo  
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cál  
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 ./s^2  
    var allowSleep = true; //Permite que los objetos que están en reposo  
  
    world = new b2World(gravity,allowSleep);
```

- La función `init()` comienza definiendo `b2World` y pasando los siguientes dos parámetros a su constructor:
 - **gravity**: Se define como un vector mediante el objeto `b2Vec2`.
 - Se fija a 9.8m/s^2 hacia abajo, lo que permite simular entornos con diferentes campos de gravedad.
 - Se fija a `0` en juegos que no necesitan la gravedad, se utilizan las características de detección de colisiones de **Box2D**.

Proceso (IV)

- **allowSleep**: Utilizado por **b2World** para decidir si incluye o no objetos que están en reposo durante los cálculos de simulación.
 - Al excluir los objetos en reposo reduce el número de cálculos innecesarios y mejora el desempeño.
 - Si un objeto está dormido, se despertará si un objeto choca con él.
- También se define la variable **scale** utilizada para convertir **metros** (en **Box2D**) a **pixeles** (las unidades del juego).
 - Box2D trabaja mejor con objetos entre 0,1m a 10m.
 - 30 pixeles equivalen a 1 metro.

Añadiendo el primer body:
El piso

Proceso (I)

- Para crear un **body** en **Box2D** se requiere:
 1. Declarar una definición **body** en un objeto **b2BodyDef**
 - Contiene la posición (**x,y**) y el tipo de **body** (estático –no le afecta la gravedad ni las colisiones con otros cuerpos- o dinámico).
 2. Declarar una definición **fixture** en un objeto **b2FixtureDef**
 - Se utiliza para unir una forma a un **body**.
 - Contiene información como la densidad, el coeficiente de fricción y el coeficiente de restitución para la forma añadida.
 3. Fijar la forma de la definición **fixture**. Los dos tipos de formas utilizado en **Box2D** son:
 - Polígonos (**b2PolygonShape**).
 - Círculos (**b2CircleShape**).

Proceso (II)

1. Declarar una definición **body** en un objeto **b2BodyDef**
 - Contiene la posición (x, y) y el tipo de **body** (estático –no le afecta la gravedad ni las colisiones con otros cuerpos- o dinámico).
2. Declarar una definición **fixture** en un objeto **b2FixtureDef**
 - Se utiliza para unir una forma a un **body**.
 - Contiene información como la densidad, el coeficiente de fricción y el coeficiente de restitución para la forma añadida.
3. Fijar la forma de la definición **fixture**. Los dos tipos de formas utilizado en **Box2D** son:
 - Polígonos (**b2PolygonShape**).
 - Círculos (**b2CircleShape**).

Proceso (III)

4. Pasar el objeto de definición **body** al método **createBody()** del mundo y recuperar el objeto **body**.
 5. Pasar la definición **fixture** al método **createFixture()** del objeto **body** y añadir la forma al **body**.
- Ahora ya se puede crear el primer **body** dentro del mundo, el piso.
 - Crear el método **createFloor()**, justo después de **init()**

createFloor()

```
function createFloor(){  
  //Una definición Body que tiene todos los datos necesarios para construir un cuerpo rígido  
  var bodyDef = new b2BodyDef;  
  bodyDef.type = b2Body.b2_staticBody;  
  bodyDef.position.x = 640/2/scale;  
  bodyDef.position.y = 450/scale;  
  
  // Un accesorio se utiliza para unir una forma a un cuerpo para la detección de colisiones  
  // La definición de un accesorio se utiliza para crear un 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 pixeles de ancho por 20 pixeles de alto  
  
  var body = world.CreateBody(bodyDef);  
  var fixture = body.CreateFixture(fixtureDef);  
}
```

- Primero se define el objeto **bodyDef**.
- Se fija como **estático** y cerca de la parte inferior del canvas.
- Se utiliza una **escala variable** para convertir de **pixeles** a **metros** para Box2D.

createFloor()

- Luego se define **FixtureDef**, que contiene los valores necesarios para añadir la forma.
- La **densidad** se utiliza para calcular el peso del cuerpo.
- El **coeficiente de fricción** para asegurar que el cuerpo se escurre de forma realista.
- La **restitución** se utiliza para hacer que el cuerpo rebote.
 - Cuanto mayor sea el valor más rebota el objeto.
 - Valores cercanos a **0** hacen que el cuerpo no rebote y perderá su momentum en una colisión (**colisión inelástica**).
 - Valores cercanos a **1** hacen que el cuerpo mantenga más su momentum y se recuperará tan rápido como llegó (**colisión elástica**).
- A continuación se fija la forma del objeto como polígono
 - **b2PolygonShape** tiene un método **helper** llamado **SetAsBox()** que fija el polígono como una caja centrada en el origen del cuerpo padre y toma el ancho y alto de la caja como parámetros.
- Finalmente se crea el **body** pasando **bodyDef** a **world.CreateBody()** y se crea el accesorio pasando el **fixtureDef** a **body.CreateFixture()**

Llamando createFloor()

- Es necesario llamar este método desde `init()`, de modo que `body` se creará cuando se llama a dicho método.

```
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo de la física.  
    var gravity = new b2Vec(0,9.8); //Declara la gravedad como 9.8 ./s^2.  
    var allowSleep = true; //Permite que los objetos que están en reposo se queden dormidos  
    //y se excluyan de los cálculos.  
    world = new b2World(gravity,allowSleep);  
  
    createFloor();  
}
```

- Ahora es necesario aprender cómo dibujar el mundo, para poder ver o que hemos creado.

Dibujando el mundo

Proceso (I)

- **Box2D** proporciona el método **DrawDebugData()** para dibujar el mundo en un canvas
 - **DrawDebugData()** dibuja una representación simple de los cuerpos y permite visualizar el mundo mientras se va creando.
- Antes de utilizarlo es necesario configurar los dibujos de depuración definiendo el objeto **b2DebugDraw()** y pasándolo al método **world.SetDebugDraw()**
 - Esto se hace en el método **setupDebugDraw()**, que se coloca debajo del método **createFloor()**, dentro del archivo **box2d.js**

setupDebugDraw()

```
var context;  
function setupDebugDraw(){  
    context = document.getElementById('canvas').getContext('2d');  
  
    var debugDraw = new b2DebugDraw();  
  
    // Utilizar este contexto para dibujar la pantalla de depuración  
    debugDraw.SetSprite(context);  
    // Fijar la escala  
    debugDraw.SetDrawScale(scale);  
    // Rellenar las cajas con transparencia de 0.3  
    debugDraw.SetFillAlpha(0.3);  
    // Dibujar líneas con espesor de 1  
    debugDraw.SetLineThickness(1.0);  
    // Mostrar todas las formas y uniones  
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit | b2DebugDraw.e_jointBit);  
  
    // Empezar a utilizar el dibujo de depuración en el mundo  
    world.SetDebugDraw(debugDraw);  
}
```

Proceso (II)

- Primero se define un manejador para el contexto del canvas.
- Se crea un nuevo objeto **b2DebugDraw** y se fijan algunos atributos utilizando sus métodos **Set**:
 - **SetSprite()**: Proporciona el contexto del canvas para el dibujo.
 - **SetDrawScale()**: Utilizado para conversiones entre unidades de **Box2D** y pixeles.
 - **SetFillAlpha()** y **SetLineThickness()**: Utilizados para dibujar estilos.
 - **SetFlags()**: Utilizado para escoger las entidades de **Box2D** a dibujar:
 - Se utilizan flags para dibujar todas las formas y uniones.
 - Y el operador lógico OR para combinar las dos flags.
 - Otras entidades a dibujar son el centro de masa (**e_centerOffMassBit**) y las cajas delimitadoras alineadas con eje (**e_aabbBit**).
- Finalmente se pasa el objeto **debugDraw** al método **world.SetDebugDraw()**

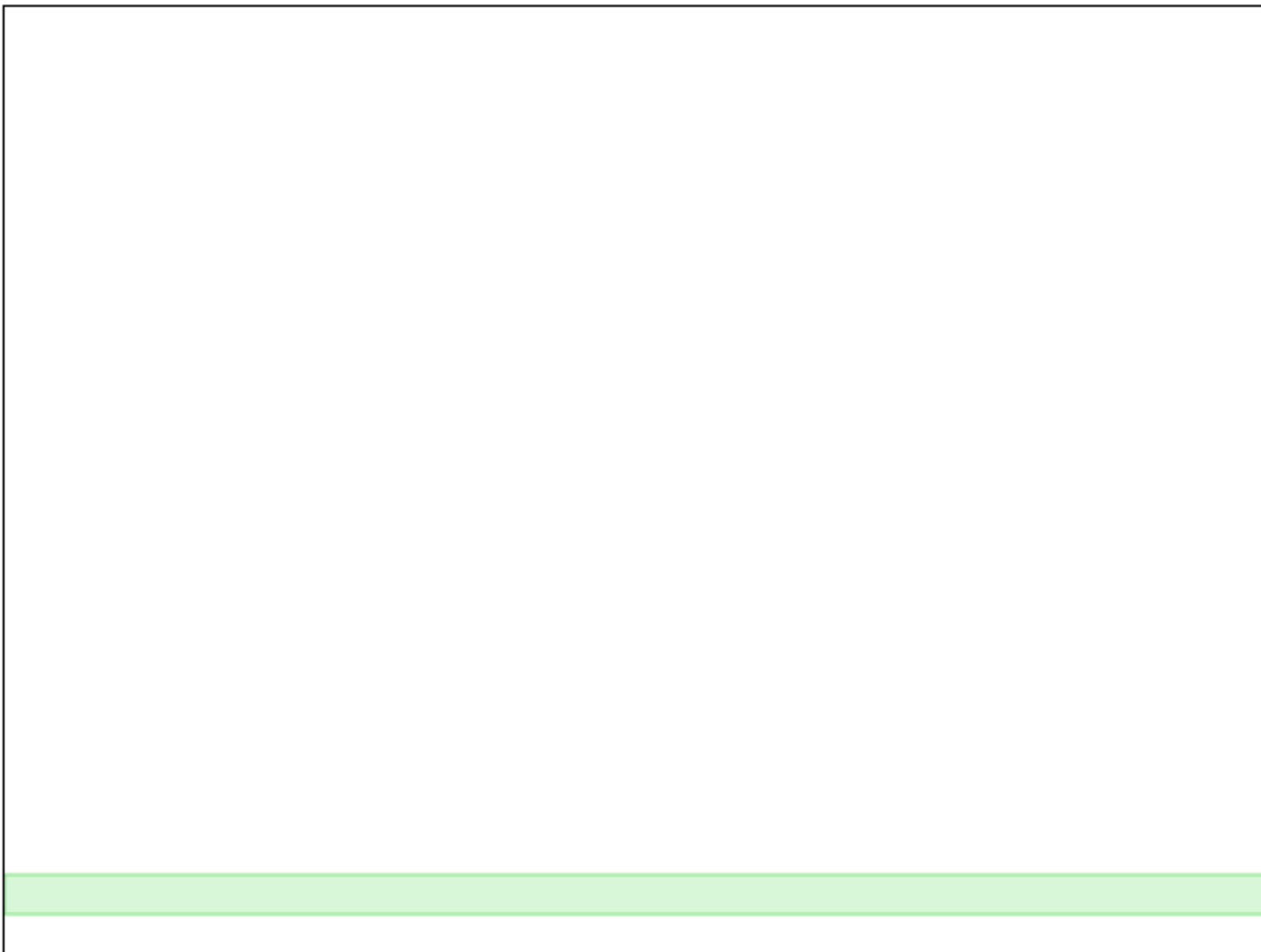
Proceso (III)

- Después de crear la función, es necesario llamarla desde `init()`:

```
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo de la física.  
    var gravity = new b2Vec(0,9.8); //Declara la gravedad como 9.8 ./s^2.  
    var allowSleep = true; //Permite que los objetos que están en reposo se queden dormidos  
    //y se excluyan de los cálculos.  
    world = new b2World(gravity,allowSleep);  
  
    createFloor();  
  
    setupDebugDraw();  
}
```

- Ahora se puede utilizar el método `world.DrawDebugData()` para dibujar el estado actual del mundo Box2D en el canvas.

Visualizar



Animando el mundo

Proceso (I)

- Animar un mundo utilizando **Box2D** seguir los pasos que se repiten dentro del bucle de animación:
 1. Decirle a **Box2D** que ejecute la simulación para un espacio pequeño de tiempo (1/60 de segundo). Se hace esto utilizando la función **world.Step()**.
 2. Dibujar todos los objetos en sus nuevas posiciones utilizando **world.DrawDebugData()** o funciones propias para dibujar.
 3. Borrar cualquier fuerza aplicada, utilizando **world.ClearForces()**.
- Dichos pasos se implementan a través de la función **animate()**, creada dentro de **box2d.js**, después de **init()**.

animate()

```
var timeStep = 1/60;  
  
//La iteration sugerida para Box2D es 8 para velocidad y 3 para posición.  
var velocityIterations = 8;  
var positionIterations = 3;  
  
function animate(){  
    world.Step(timeStep,velocityIterations,positionIterations);  
    world.ClearForces();  
  
    world.DrawDebugData();  
  
    setTimeout(animate, timeStep);  
}
```

TimeStep es la cantidad de tiempo que se quiere simular en Box2D (se fija a un intervalo de 1/60 segundos).

- *Integrator*
 - Algoritmo computacional
 - Simula las ecuaciones físicas en puntos discretos de tiempo.
- *Constraint solver*
 - Resuelve todas las restricciones en la simulación, una a la vez.
 - A través de dos fases: velocidad y posición, fijadas a 8 y 3.

animate()

- Después de la simulación se llama al método **world.ClearForces()** para limpiar cualquier fuerza aplicada a los cuerpos.
- Luego se llama a **world.DrawDebugData()** para dibujar el mundo en el canvas.
- Finalmente se utiliza **setTimeout()** para llamar el bucle de la animación de nuevo, después del fin del tiempo para el próximo paso de tiempo.
- Ahora que el bucle de animación está completo, podemos ver el mundo creado, llamando llamando al nuevo método desde la función **init()**.

Actualización init()

```
function init(){
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 m/s^2.
    var allowSleep = true; //Permite que los objetos que están en reposo se qu

    world = new b2World(gravity,allowSleep);

    createFloor();

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

Más elementos Box2D

Otros tipos de elementos

- Cuerpos simples como rectangular, circular o poligonal.
- Cuerpos complejos que combinan múltiples formas.
- Articulaciones tales como aquellas que conectan múltiples cuerpos.
- Escuchadores de contacto que permiten manejar eventos de colisión.

Cuerpos simples

RectangularBody()

- Se define un `b2PolygonShape` utilizando `SetAsBox()`.
- Luego se crea el rectángulo utilizando `createRectangularBody()`.
- Se añade al código de `box2d.js`.

RectangularBody()

```
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);
}
```

El cuerpo se
verá afectado
por las colisiones

Actualización init()

```
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo de la f.  
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 m/s^2.  
    var allowSleep = true; //Permite que los objetos que están en reposo se queden do  
  
    world = new b2World(gravity,allowSleep);  
  
    createFloor();  
    //Crear algunos cuerpos con formas simples  
    createRectangularBody();  
  
    setupDebugDraw();  
    animate();  
}
```

- Al ser un cuerpo dinámico caerá hacia abajo debido a la gravedad hasta que toque el suelo y luego saltará del suelo.
- El cuerpo se eleva a una altura inferior después de cada rebote hasta que finalmente se establece en el suelo.
- Cambiando el coeficiente de restitución se hará que el objeto rebote más o menos, según se quiera.

CircleBody()

- Se define una forma circular configurando la propiedad **shape** para un objeto **b2CircleShape**.
- Luego se crea el círculo utilizando **createCircleBody()**.
- Se añade al código de **box2d.js**.

CircleBody()

```
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);  
}
```

El constructor toma un parámetro, el radio del círculo.

Actualización init()

```
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo de la  
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 m/s^2.  
    var allowSleep = true; //Permite que los objetos que están en reposo se queden e  
  
    world = new b2World(gravity,allowSleep);  
  
    createFloor();  
    //Crear algunos cuerpos con formas simples  
    createRectangularBody();  
    createCircularBody();  
  
    setupDebugDraw();  
    animate();  
}
```

PolygonBody()

- Box2D permite crear un polígono definiendo las coordenadas para cada uno de los puntos.
- La única restricción es que los polígonos sean convexos.
- Primero se debe crear un *array* de objetos **b2Vec2** con las coordenadas para cada uno de los puntos.
- Luego se pasa el *array* al método **shapeSetAsArray()**.
- Para ello se utiliza el método **createSimplePolygonBody()**.
- Se añade al código de **box2d.js**.

PolygonBody()

```
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;
    // Crear un array de puntos b2Vec2 en la dirección de las agujas del reloj.
    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),
    ];
    // Usar SetAsArray para definir la forma utilizando el array de puntos.
    fixtureDef.shape.SetAsArray(points,points.length);

    var body = world.CreateBody(bodyDef);

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

PolygonBody()

- Todas las coordenadas son relativas al origen del cuerpo.
- El primer punto comienza en el origen (0,0) del cuerpo y se coloca en la posición del cuerpo (230,50).
- Es necesario cerrar el polígono, Box2D se encarga de ello.
- Todos los puntos se deben definir en la dirección de las agujas del reloj.
 - Si se definen en sentido contrario no se podrá manejar las colisiones.
- Luego se llama al método **SetAsArray()** y se le pasa dos parámetros: los puntos del *array* y el número de puntos.
- Se llama a **createSimplePolygon()** desde la función **init()**

Actualización init()

```
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo de la física.  
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 m/s^2.  
    var allowSleep = true; //Permite que los objetos que están en reposo se queden dormidos  
  
    world = new b2World(gravity,allowSleep);  
  
    createFloor();  
    //Crear algunos cuerpos con formas simples  
    createRectangularBody();  
    createCircularBody();  
    createSimplePolygonBody();  
  
    setupDebugDraw();  
    animate();  
}
```

Cuerpos complejos

Crear cuerpos complejos con formas múltiples

- Para crear formas complejas es necesario juntar múltiples accesorios (**fixtures**), cada uno con su propia forma al mismo cuerpo (**body**).
- Para ello se utiliza el método `createComplexPolygonBody()`
- Se añade al código de `box2d.js`

createComplexBody()

```
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);

    //Crear el primer accesorio y añadir una forma circular al cuerpo.
    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);

    // Crear el segundo accesorio y añadir una forma poligonal al cuerpo.
    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);
}
```

createComplexBody()

- Se crea un cuerpo y luego dos accesorios diferentes, el primero para un círculo y el segundo para un polígono.
- Se unen ambas formas al cuerpo mediante el uso del método **createFixture()**.
- **Box2D** crea automáticamente un cuerpo rígido que incluye ambas formas.
- Se llama **createComplexBody()** desde la función **init()**.

Actualización init()

```
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo de la física  
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 m/s^2.  
    var allowSleep = true; //Permite que los objetos que están en reposo se queden dormidos  
  
    world = new b2World(gravity,allowSleep);  
  
    createFloor();  
    //Crear algunos cuerpos con formas simples  
    createRectangularBody();  
    createCircularBody();  
    createSimplePolygonBody();  
    //Crear un cuerpo combinando dos formas  
    createComplexBody();  
  
    setupDebugDraw();  
    animate();  
}
```

**Conectar cuerpos con
artefactos**

createRevoluteJoint()

- Las articulaciones se utilizan para restringir los cuerpos al mundo o a otras articulaciones.
- **Box2D** soporta muchos tipos de articulaciones como `pulley` (polea), `gear` (engranaje), `distance` (distancia), `revolute` (revolución) y `weld` (soldadura).
- Algunas de estas articulaciones restringen el movimiento (`pulley` y `revolute`).
- Otras proporcionan motores que se pueden utilizar para manejar la articulación a una velocidad determinada.

createRevoluteJoint()

```
function createRevoluteJoint(){
    //Definir el primer cuerpo
    var bodyDef1 = new b2BodyDef;
    bodyDef1.type = b2Body.b2_dynamicBody;
    bodyDef1.position.x = 480/scale;
    bodyDef1.position.y = 50/scale;
    var body1 = world.CreateBody(bodyDef1);

    //Crear el primer accesorio y añadir na forma rectangular al cuerpo.
    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);

    // Definir el segundo cuerpo
    var bodyDef2 = new b2BodyDef;
    bodyDef2.type = b2Body.b2_dynamicBody;
    bodyDef2.position.x = 470/scale;
    bodyDef2.position.y = 50/scale;
    var body2 = world.CreateBody(bodyDef2);

    //Crear el segundo accesorio y añadir un polígono al cuerpos
    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),
```

createRevoluteJoint()

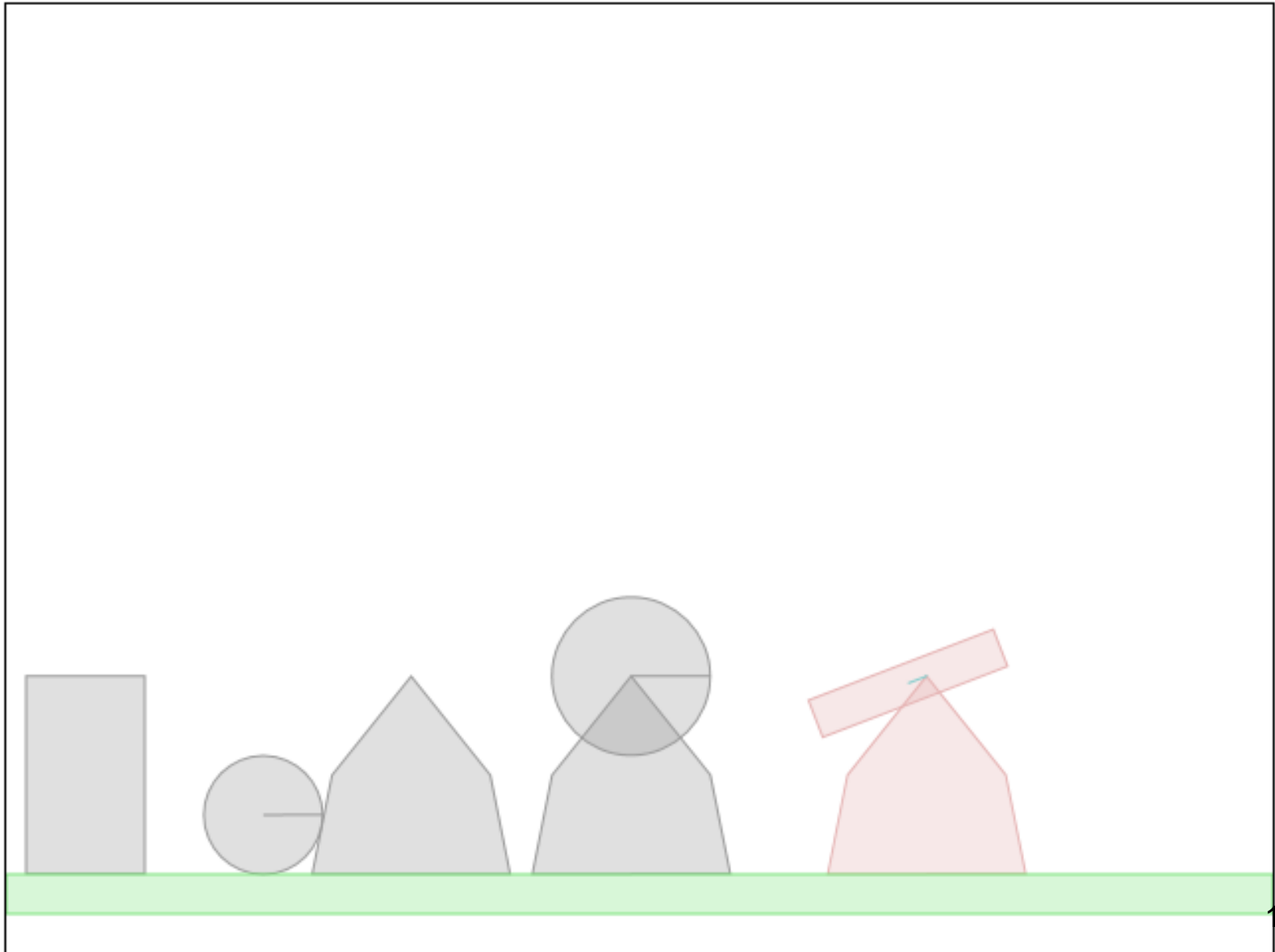
```
new b2Vec2(50/scale,100/scale),  
new b2Vec2(-50/scale,100/scale),  
new b2Vec2(-40/scale,50/scale),  
];  
fixtureDef2.shape.SetAsArray(points,points.length);  
body2.CreateFixture(fixtureDef2);  
  
// Crear una articulación entre el body1 y el body2  
var jointDef = new b2RevoluteJointDef;  
var jointCenter = new b2Vec2(470/scale,50/scale);  
  
jointDef.Initialize(body1, body2, jointCenter);  
world.CreateJoint(jointDef);  
}
```

- Se llama a `createRevoluteJoint()` desde `init()`.

Actualización init()

```
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo de la física.  
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 m/s^2.  
    var allowSleep = true; //Permite que los objetos que están en reposo se queden dormidos  
  
    world = new b2World(gravity,allowSleep);  
  
    createFloor();  
    //Crear algunos cuerpos con formas simples  
    createRectangularBody();  
    createCircularBody();  
    createSimplePolygonBody();  
    //Crear un cuerpo combinando dos formas  
    createComplexBody();  
    //Unir dos cuerpos mediante una articulación (revolute joint)  
    createRevoluteJoint();  
  
    setupDebugDraw();  
    animate();  
}
```

Visualizar



Seguimiento de colisiones y daños

Introducción

- Es interesante poder hacer el seguimiento de una colisión, del impacto que causa y simular el daño sufrido por un objeto.
- Para ello es necesario primero asociarle vida o salud.
- Se puede asignar cualquier propiedad personalizada a un cuerpo llamando al método **SetUserData()** y recuperarla después llamando a su método **GetUserData()**.
- Creamos un cuerpo que tenga su propia salud a diferencia de los anteriores.
- Esto se hace dentro de un método llamado **createSpacialBody()** que se añade al código de **box2d.js**

createSpecialBody()

```
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});
```

```
    //Crear un accesorio para unir una forma circular al cuerpo
```

```
    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);
```

```
}
```

Se salva una referencia a **Body** en una variable llamada **specialBody**, fuera de la función.

Después de crear **Body** se llama a **SetUserData()** al que se le pasan dos parámetros **name** y **life**.

Actualización init()

```
function init(){  
  //Configuración del mundo Box2d que realiazará la mayor parte del cálculo de la física.  
  var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 m/s^2.  
  var allowSleep = true; //Permite que los objetos que están en reposo se queden dormidos y se excluyan de los  
  
  world = new b2World(gravity,allowSleep);  
  
  createFloor();  
  //Crear algunos cuerpos con formas simples  
  
  createRectangularBody();  
  createCircularBody();  
  createSimplePolygonBody();  
  
  //Crear un cuerpo combinando dos formas  
  createComplexBody();  
  
  //Unir dos cuerpos mediante una articulación  
  createRevoluteJoint();  
  
  // Crear un cuerpo con datos especiales del usuario  
  createSpecialBody();  
  
  setupDebugDraw();  
  animate();  
}
```


Contact listeners

Introducción

- **Box2D** proporciona objetos llamados **contact listeners** para definir deferentes manejadores de eventos relacionados con los contactos.
- Para ello es necesario primero definir un objeto **b2ContactListener** y sobre escribir uno o más eventos que se quieran supervisar.
- **b2ContactListener** tiene cuatro eventos que se pueden utilizar de acuerdo a las necesidades particulares:
 - **Begincontact()**: Llamado cuando dos **fixtures** entran en contacto.
 - **EndContact()**: Llamado cuando dos **fixtures** cesan el contacto.
 - **PostSolve()**: Permite inspeccionar un contacto después de que el solucionador termina. Útil para hacer seguimiento de impulsos.
 - **PreSolve()**: Permite inspeccionar un contacto antes de pasar al solucionador.
- Una vez sobre escritos los métodos necesarios se le pasa el **contact listener** a **world.SetContactListener()**.

Procedimiento (I)

- Para hacer el seguimiento que causa una colisión es necesario escuchar al evento **PostSolve()** el cual proporciona el impulso transferido durante una colisión.

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

    // Si cualquiera de los cuerpos es el special body, reduzca su vida
    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);
}
```

Se crea un objeto **b2ContactListener** y se sobre escribe su **PostSolve()** con nuestro propio manejador.

Contiene detalles de los **fixtures** involucrados en la colisión.

Contiene el impulso normal y tangencial durante la colisión.

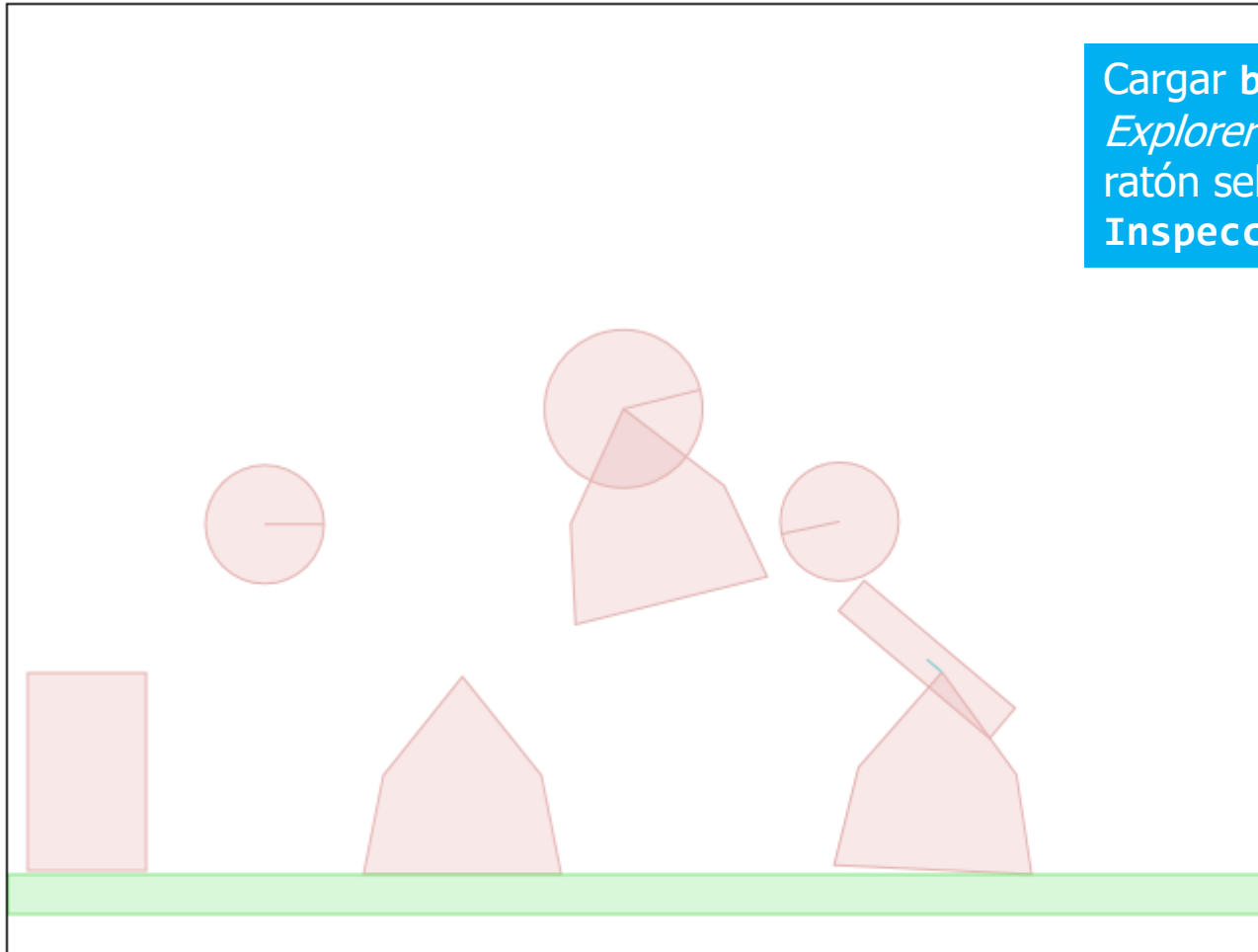
- Dentro de **PostSolve()** se extraen los dos cuerpos involucrados en la colisión y se verifica si uno de ellos es un **special body**.
- Si es así, se extrae el impulso normal entre los dos cuerpos y se restan puntos de vida del cuerpo.
- Se registra este evento en la consola para hacer el seguimiento de cada colisión.
- Cuanto más impulso en la colisión y más alto el número de colisiones, menos vidas tendrá el cuerpo.






Actualización init()

```
function init(){  
    //Configuración del mundo Box2d que realizará la mayor parte del cálculo de la física.  
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9.8 m/s^2.  
    var allowSleep = true; //Permite que los objetos que están en reposo se queden dormidos y se excluyan de los cálculos.  
  
    world = new b2World(gravity,allowSleep);  
  
    createFloor();  
    //Crear algunos cuerpos con formas simples  
    createRectangularBody();  
    createCircularBody();  
    createSimplePolygonBody();  
  
    //Crear un cuerpo combinando dos formas  
    createComplexBody();  
  
    //Unir dos cuerpos mediante una articulación  
    createRevoluteJoint();  
  
    // Crear un cuerpo con datos especiales del usuario  
    createSpecialBody();  
  
    // Crear contact listeners y registrar los eventos  
    listenForContact();  
  
    setupDebugDraw();  
    animate();  
}
```

Visualización

Cargar `box2d.html` en *Internet Explorer* y con el botón derecho del ratón seleccionar del menú la opción **Inspeccionar elemento > Consola**



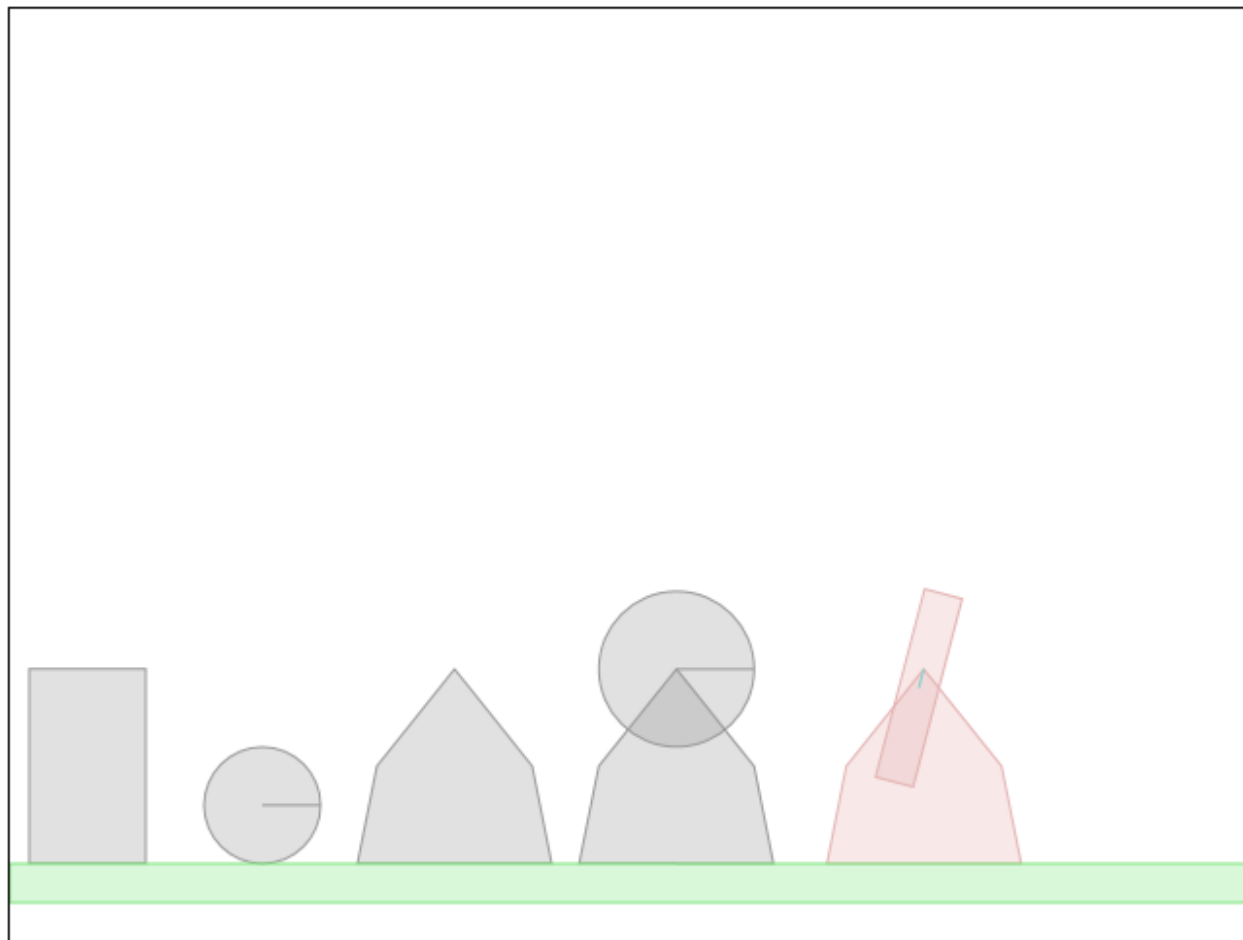
F12	Explorador DOM	Consola	Depurador	Red	Capacidad de respuesta de la IU	Generador de perfiles	Memoria	Emulación
 0	 1	 1						
<pre>The special body was in a collision with impulse 0 and its life has now become 161.4728124174255</pre>								
<pre>The special body was in a collision with impulse 17.500390855275892 and its life has now become 143.9724215621496</pre>								
<pre>The special body was in a collision with impulse 0 and its life has now become 143.9724215621496</pre>								
<pre>The special body was in a collision with impulse 5.230892675743804 and its life has now become 138.7415288864058</pre>								
<pre>The special body was in a collision with impulse 0 and its life has now become 138.7415288864058</pre>								






Procedimiento (II)

- Ahora es posible comprobar si después de cada iteración la vida del cuerpo ha llegado a 0 en cuyo caso se puede eliminar del mundo utilizando `world.DestroyBody()`.
- El lugar más sencillo para hacer esta comprobación es en el método `animate()`.

```
function animate(){  
    world.Step(timeStep, velocityIterations, positionIterations);  
    world.ClearForces();  
  
    world.DrawDebugData();  
  
    //Matar Special Body si muere  
    if (specialBody && specialBody.GetUserData().life<=0){  
        world.DestroyBody(specialBody);  
        specialBody = undefined;  
        console.log("The special body was destroyed");  
    }  
  
    setTimeout(animate, timeStep);  
}
```

Visualización



F12	Explorador DOM	Consola	Depurador	Red	Capacidad de respuesta de la IU	Generador de perfiles	Memoria	Emulación
 0	 1	 1						
<p>The special body was in a collision with impulse 0 and its life has now become 36.29063046468919</p> <p>The special body was in a collision with impulse 22.251023599781263 and its life has now become 14.039606864907924</p> <p>The special body was in a collision with impulse 17.83647116336406 and its life has now become -3.7968642984561356</p> <p>The special body was in a collision with impulse 21.504168861335394 and its life has now become -25.30103315979153</p> <p>The special body was in a collision with impulse 7.436621337829014 and its life has now become -32.73765449762055</p> <p>The special body was destroyed</p>								

Consideraciones

- Es posible hacer seguimiento de todos los cuerpos y elementos del juego, utilizando un principio similar iterando a través de un *array* de objetos.
- El punto donde destruimos un cuerpo es el lugar perfecto para añadir:
 - Sonidos de explosiones
 - Efectos visuales en un juego
 - Actualizar la puntuación

**Dibujar nuestros propios
caracteres**

Introducción

- El método **DrawDebugData()** definido en **Box2D** se utiliza para dibujar.
- Para dibujar caracteres propios es necesario utilizar otros métodos.
 - Cada objeto **b2Body** tiene dos métodos **GetPosition()** y **GetAngle()** que proporcionan las coordenadas y la rotación del cuerpo en el mundo **Box2D**
 - Utilizando la variable **scale** y los métodos del canvas **translate()** y **rotate()** es posible dibujar caracteres o *sprites* sobre el canvas en una ubicación que **Box2D** calcula por nosotros.
- Para ilustrar lo anterior podemos dibujar el **special body** dentro del método **drawSpecialBody()** del archivo **box2d.js**

drawSpecialBody()

```
function drawSpecialBody(){
    // Obtener la posición y el ángulo del cuerpo
    var position = specialBody.GetPosition();
    var angle = specialBody.GetAngle();

    // Traducir y girar el eje a la posición y al ángulo del cuerpo
    context.translate(position.x*scale,position.y*scale);
    context.rotate(angle);

    // Dibujar una cara circular llena
    context.fillStyle = "rgb(200,150,250)";
    context.beginPath();
    context.arc(0,0,30,0,2*Math.PI,false);
    context.fill();

    // Dibujar dos ojos rectangulares
    context.fillStyle = "rgb(255,255,255)";
    context.fillRect(-15,-15,10,5);
    context.fillRect(5,-15,10,5);

    // Dibujar un arco hacia arriba o hacia abajo para una sonrisa dependiendo de la vida
    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(0,0,10,Math.PI,2*Math.PI,false);
    }
    context.stroke();

    // Traducir y girar el eje de nuevo a la posición original y el ángulo
    context.rotate(-angle);
    context.translate(-position.x*scale,-position.y*scale);
}
```

Procedimiento

- Antes de ver `drawSpecialBody()` en acción es necesario llamarlo desde `animate()`

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

    world.DrawDebugData();

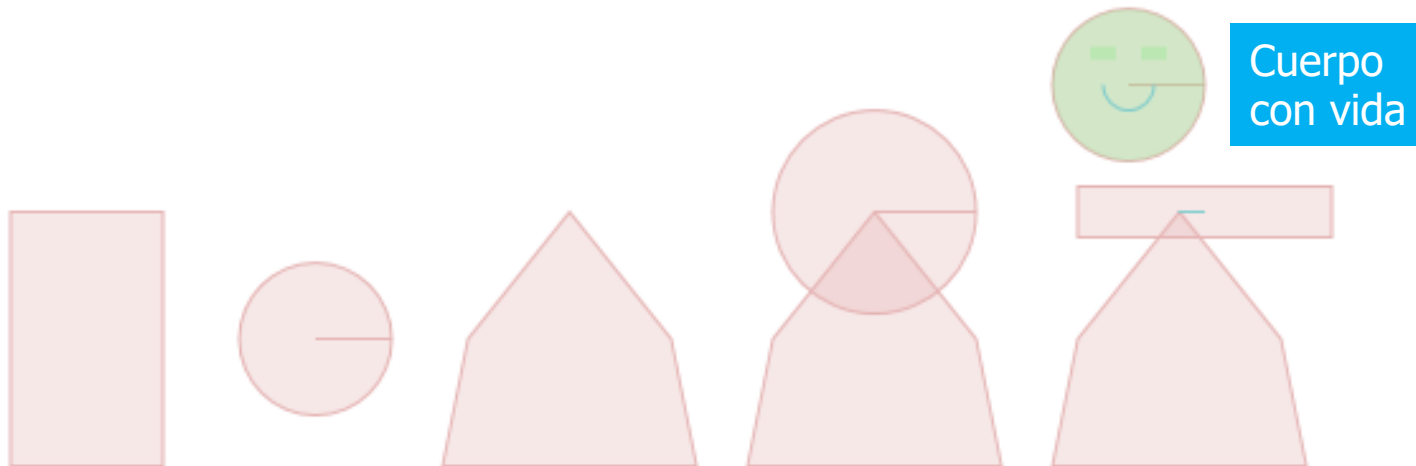
    // Dibujo personalizado
    if (specialBody){
        drawSpecialBody();
    }

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

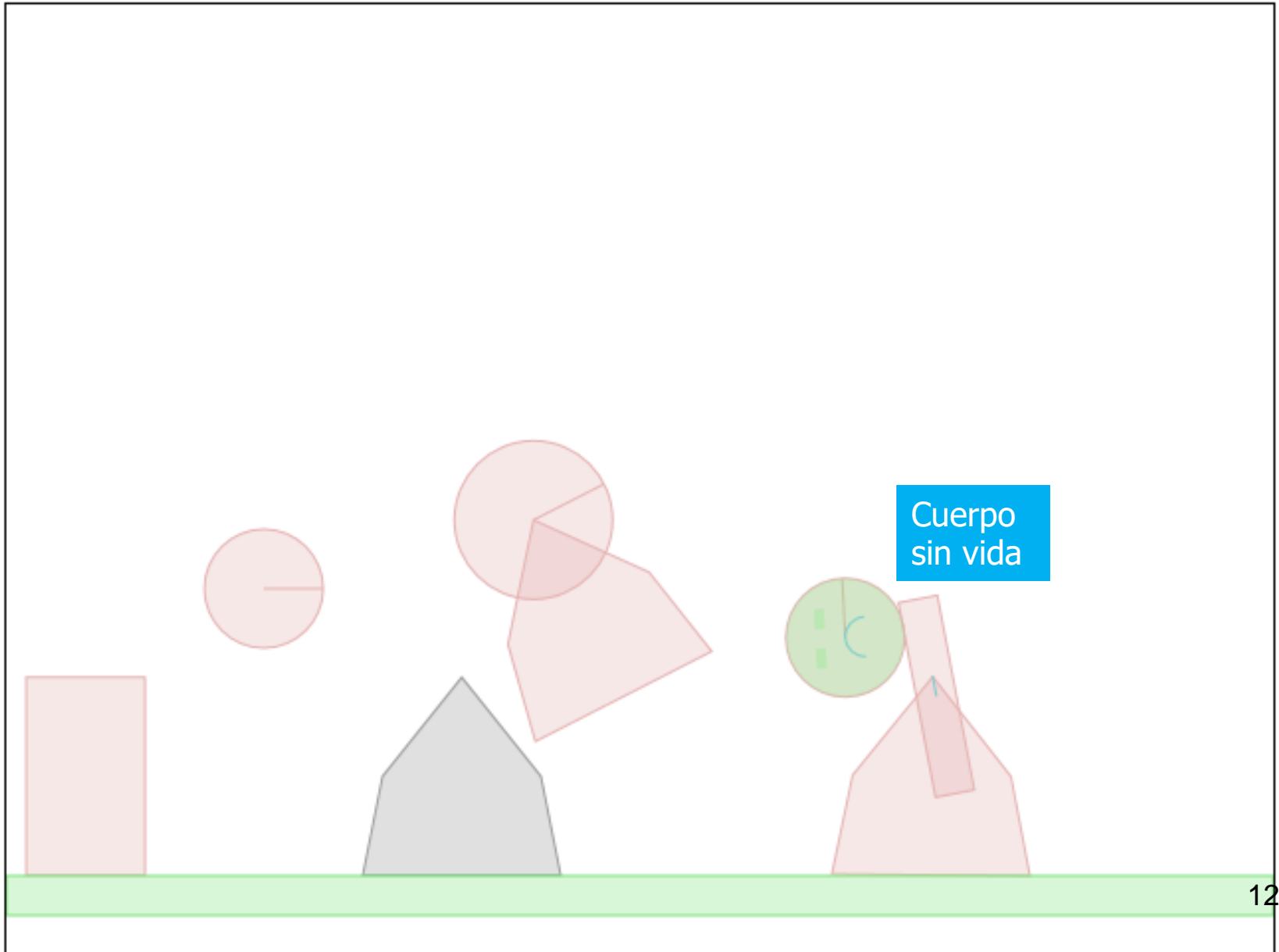
    setTimeout(animate, timeStep);
}
```

Se comprueba si `specialBody` aún está definido, en cuyo caso se llama a `drawSpecialBody()`.

Visualización (I)



Visualización (II)



Resumen

- Hasta este punto se ha:
 - Animado el mundo de forma realista permitiendo a **Box2D** realizar los cálculos físicos y dibujando el mundo mediante **DrawDebugData()**.
 - Utilizado **contact listeners** para hacer el seguimiento de las colisiones, dañar y destruir objetos dentro del mundo.
 - Dibujado nuestro propio carácter que se movía gracias a **Box2D**.
- Para profundizar en el uso de **Box2D** visitar:
<http://www.box2dflash.org/docs/>, que contiene una guía de uso del motor.

Integración del motor de la física

Proceso

- Añadir **entidades** a los niveles.
 - Añadir interactividad al ratón para poder jugar.
- Utilizar **Box2D** para **simular** dichas entidades.
- **Animar** las entidades dentro del juego.
- Añadir **sonidos, música de fondo** y algunos otros detalles para finalizar el diseño e implementación del juego.

Definir entidades

Proceso (I)

- Los niveles del juego contienen datos para las imágenes del fondo y primer plano y un **array** vacío para las entidades.
- Ese **array** puede contener todas las entidades del juego:
 - Villanos
 - Suelo
 - Bloques
- Y se utiliza para preguntar a **Box2D** para crear las formas correspondientes

Es posible definir entidades personalizadas como **calories**, es el número de puntos anotados por destruir un villano.

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

Type se utiliza para decidir cómo manejar una entidad durante las operaciones de creación y dibujo.

x, y y angle se utilizan para fijar la posición inicial y la orientación de las entidades.

Proceso (II)

- La propiedad **name** indica cuál es el **sprite** a utilizar para dibujar la entidad. Las imágenes utilizadas para las entidades se almacenan en el folder **images/entities**
 - Name también se puede utilizar para hacer referencia a las definiciones de las entidades, las cuales pueden incluir datos del accesorio (**fixture**) como:

```
"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,  
},
```

Detalles de la forma
(héroes y villanos)

Datos sobre la salud de
los objetos destructibles

Densidad

Restitución

Proceso (III)

- Una vez decidido cómo se almacenarán las entidades, es necesario definir cómo se van a crear.
- Comenzar por crear un objeto **entities** dentro de **game.js**.
- El objeto **entities** permite:
 - Manejar todas las operaciones relacionadas con entidades dentro del juego.
 - Contener todas las definiciones de entidades, así como también los métodos para crearlas y dibujarlas.

var 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,
    },
  },
}
```

entities contiene un array con las definiciones de todos los tipos de materiales y de todos los héroes y villanos del juego

```
//Tomar la entidad, crear un cuerpo Box2D y añadirlo al mundo
create:function(entity){
};
//Tomar la entidad, su posición y su ángulo y dibujarlo en el canvas del juego
draw:function(entity,position,angle){
}
```

Proceso (IV)

- Definir marcadores de posición para las funciones **create()** y **draw()**.
- Antes de implementarlas es necesario añadir **Box2D** al código del archivo **index.html**.

```
<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>
```

Proceso (V)

- Crear un objeto box2d dentro del archivo game.js.

```
var box2d = {
  scale:30,
  init:function(){
    // Configurar el mundo de box2d que hará la mayoría de ellos cálculo de física
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9,8 m / s ^ 2 hacia abajo
    var allowSleep = true; //Permitir que los objetos que están en reposo se queden dormidos y se excluyan de los cálculos
    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;
    },
  },
}
```


Proceso (VI)

```
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;
},
}
```

Proceso (VII)

- El objeto **box2d()** conecta un método **init()** que inicializa al nuevo objeto **b2world**.
- Además contiene dos métodos *helper* **createRectangle()** y **createCircle()** ambos reciben dos parámetros, los objetos:
 - **entity**: Contiene detalles como la posición, el ángulo y si la entidad es estática o no.
 - **definition** : Contiene detalles sobre los accesorios (**fixture**) como densidad o restitución.
- Utilizando estos parámetros los métodos crean los cuerpos (**body**) y accesorios (**fixture**) **Box2D** y los añade también al mundo **Box2D**.

Proceso (VIII)

- Estos métodos convierten la posición y el tamaño utilizando **box2d.scale** y convierten el ángulo de grados a radianes antes de que sean utilizados por **Box2D**.
- Finalmente estos métodos añaden el objeto **entity** al cuerpo (**body**) mediante el método **SetUserData()**.
 - Hace posible recuperar cualquier dato relacionado con la entidad para el cuerpo (**body**) **Box2D** utilizando su método **GetUserData()**.

Crear entidades

Proceso (I)

- Una vez configurado **Box2D**, implementar el método **entities.create()** dentro del objeto **entities** definido anteriormente.
- Este método tomará un objeto **entity** como un parámetro y lo añadirá el mundo.

entities.create ()

```
//Tomar la entidad, crear un cuerpo box2d y añadirlo al mundo
create:function(entity){
    var definition = entities.definitions[entity.name];
    if(!definition){
        console.log ("Undefined entity name",entity.name);
        return;
    }
    switch(entity.type){
        case "block": // Rectángulos simples
            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": // Rectángulos simples
            // No necesitan salud, son indestructibles
            entity.shape = "rectangle";
            // No hay necesidad de sprites. Éstos no serán dibujados en absoluto
            box2d.createRectangle(entity,definition);
            break;
        case "hero": // Circulos simples
        case "villain": // Pueden ser círculos o rectángulos
            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;
    }
},
```

Proceso (II)

- Este método utiliza el tipo **entity** para decidir cómo manejar el objeto **entity** y sus propiedades:
 - **Block**: Para entidades *block*, se fija la entidad **health** y las propiedades **fullHealth**.
 - Basada en la definición de la entidad y fija la forma de la propiedad a **rectangle**.
 - Se carga el **sprite** y se llama a **box2d.createRectangle()**.
 - **Ground**: Para entidades *ground*, establecemos la propiedad de forma del objeto entidad a **rectangle** y se llama a **box2d.create.Rectangle()**
 - No se carga el **sprite** porque se utilizará el suelo de la imagen del primer plano y no se dibujará el suelo separadamente.

Proceso (III)

- **Hero y villian:** Para las entidades *hero* y *villain*, se fija la entidad **health** y las propiedades **fullHealth** y la forma de las propiedades basadas en la definición de la entidad.
 - Se fija el **radius** o las propiedades **height** y **width** basadas en la forma de la entidad.
 - Finalmente se llama a **box2d.createRectangle()** o **box2d.createCircle()** basado en la forma.
- Una vez que se sabe cómo crear entidades, se añaden algunas entidades a los niveles.

**Añadir entidades
a los niveles**

Proceso (I)

Se añade dentro de var levels

```
data: [
  { // Primer nivel
    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: 185, y: 390, width: 30, height: 80, isStatic: true},

      {type: "block", name: "wood", x: 520, y: 380, angle: 90, width: 100, height: 25},
      {type: "block", name: "glass", x: 520, y: 280, angle: 90, width: 100, height: 25},
      {type: "villain", name: "burger", x: 520, y: 205, calories: 590},

      {type: "block", name: "wood", x: 620, y: 380, angle: 90, width: 100, height: 25},
      {type: "block", name: "glass", x: 620, y: 280, angle: 90, width: 100, height: 25},
      {type: "villain", name: "fries", x: 620, y: 205, calories: 420},

      {type: "hero", name: "orange", x: 80, y: 405},
      {type: "hero", name: "apple", x: 140, y: 405},
    ]
  },
  { // Segundo nivel
    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: 185, y: 390, width: 30, height: 80, isStatic: true},

      {type: "block", name: "wood", x: 820, y: 380, angle: 90, width: 100, height: 25},
      {type: "block", name: "wood", x: 720, y: 380, angle: 90, width: 100, height: 25},
      {type: "block", name: "wood", x: 620, y: 380, angle: 90, width: 100, height: 25},
      {type: "block", name: "glass", x: 670, y: 317.5, width: 100, height: 25},
      {type: "block", name: "glass", x: 770, y: 317.5, width: 100, height: 25},

      {type: "block", name: "glass", x: 670, y: 255, angle: 90, width: 100, height: 25},
      {type: "block", name: "glass", x: 770, y: 255, angle: 90, width: 100, height: 25},
      {type: "block", name: "wood", x: 720, y: 192.5, width: 100, height: 25},

      {type: "villain", name: "burger", x: 715, y: 155, calories: 590},
      {type: "villain", name: "fries", x: 670, y: 405, calories: 420},
      {type: "villain", name: "sodacan", x: 765, y: 400, calories: 150},

      {type: "hero", name: "strawberry", x: 30, y: 415},
      {type: "hero", name: "orange", x: 80, y: 405},
      {type: "hero", name: "apple", x: 140, y: 405},
    ]
  }
],
```

Proceso (II)

- El **primer nivel** contiene dos entidades de fondo **ground**, una para el suelo y otra para la onda.
- Estas entidades están destinadas a ser objetos estáticos que no son dibujados por nosotros.
- También contiene cuatro entidades **block** (**glass** y **wood**). Son elementos destructibles que se posicionan utilizando sus propiedades **angle**, **x** e **y**.
- Además contiene dos entidades **hero** (**orange** y **apple**) y dos entidades **villian** (**burger** y **fries**). Los villanos tienen una propiedad extra llamada **calories**, utilizada para incrementar la puntuación del jugador al ser destruidas.

Proceso (III)

- El **segundo nivel** tiene un diseño similar, excepto con unas pocas entidades más.
- Una vez definidas las entidades para cada nivel, es necesario cargar las entidades cuando se cargue el nivel.
- Para hacer esto se modifica el método **load()** del objeto **levels**.

Proceso (IV)

```
// Cargar todos los datos e imágenes para un nivel específico
```

```
load:function(number){
```

```
  //Inicializar box2d world cada vez que se carga un nivel
```

```
  box2d.init();
```

```
  //Declarar un nuevo objeto de nivel actual
```

```
  game.currentLevel = {number:number,hero: []};
```

```
  game.score=0;
```

```
  $('#score').html('Score: '+game.score);
```

```
  game.currentHero = undefined;
```

```
  var level = levels.data[number];
```

```
  //Cargar las imágenes de fondo, primer plano y honda
```

```
  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");
```

```
  //Cargar todas las entidades
```

```
  for (var i = level.entities.length - 1; i >= 0; i--){
```

```
    var entity = level.entities[i];
```

```
    entities.create(entity);
```

```
  };
```

```
  //Llamar a game.start() una vez que todos los assets han sido cargados
```

```
  if(loader.loaded){
```

```
    game.start();
```

```
  } else {
```

```
    loader.onload = game.start;
```

```
  }
```

```
}
```

Se añade box2dinit()

Con el bucle for se itera a través de todas las entidades para un nivel y llamar a `entities.create()` para cada entidad.

Proceso (V)

- Cuando se cargue el nivel **Box2D** será inicializado y todas las entidades serán cargadas en el mundo **Box2D**.
- Aún no es posible ver los cuerpos añadidos, para ello utilizaremos el método de depuración del dibujo visto anteriormente.

Configurar el dibujo de depuración de Box2D

Proceso (I)

- Crear un nuevo `<canvas>` dentro de el archivo HTML y ponerlo antes del final de `<body>`

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

- Este canvas es más grande que el nuestro, así que podemos ver el nivel completo sin panoramización.
- Utilizamos este canvas y depuramos el dibujo, solo para diseñar y probar los niveles.
- Es posible borrar todas las trazas de la depuración del dibujo una vez se completa el juego.
- Configurar la depuración del dibujo cuando se inicialice **Box2D**, modificando `box2d.init()` dentro de `var box2d.`

Proceso (II)

```
init:function(){
    // Configurar el mundo de box2d que hará la mayoría de ellos cálculo de física
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9,8 m / s ^ 2 hacia abajo
    var allowSleep = true; //Permitir que los objetos que están en reposo se queden dormidos y se excluyan de los cálculos
    box2d.world = new b2World(gravity,allowSleep);

    // Configurar la depuración del dibujo
    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);
},
```

- Antes de ver los resultados de la depuración del dibujo es necesario llamar al método `Draw.DebugData()` del objeto `world`.
- Lo haremos en un nuevo método llamado `drawAllBodies()`, dentro del objeto `game`.
- Se puede llamar a este método desde `animate()` del objeto `game`.

Proceso (III)

```
animate: function(){
    // Animar el fondo
    game.handlePanning();

    // TODO: Animar los personajes

    // Dibuja el fondo con desplazamiento de paralaje
    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);

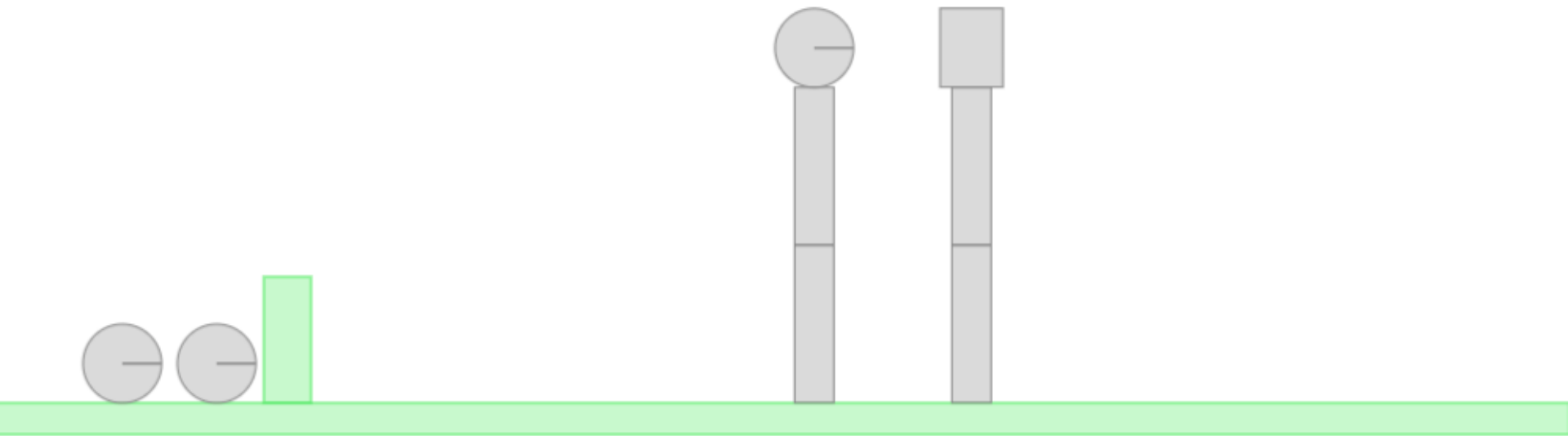
    // Dibuja la honda
    game.context.drawImage(game.slingshotImage, game.slingshotX-game.offsetLeft, game.slingshotY);

    // Dibuja todos los cuerpos
    game.drawAllBodies();

    // Dibuja el fente de la onda
    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: Iterar a través de todos los cuerpos y dibujarlos sobre el canvas del juego
}
```

Visualización



Dibujar las entidades

Proceso (I)

Definir `draw()` dentro del objeto `entities`

```
// Tomar la entidad, su posición y ángulo y dibujar en el lienzo de juego
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":
            // No hacer nada ... Vamos a dibujar objetos como el suelo y la honda por separado
            break;
    }

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

Toma estos parámetros y los dibuja en el lienzo del juego

Traduce y rota el contexto a la posición y el ángulo de la entidad y dibuja el objeto en el canvas basado en el tipo y la forma de la entidad

Rota y traduce y el contexto a la posición original

Proceso (II)

- Llamar `entities.draw()` desde cada entidad en el mundo del juego.
- Iterar a través de cada cuerpo en el mundo del juego, utilizando el método `GetBodyList()` del objeto `world`.
- Modificar `drawAllBodies()` del objeto `game`.

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

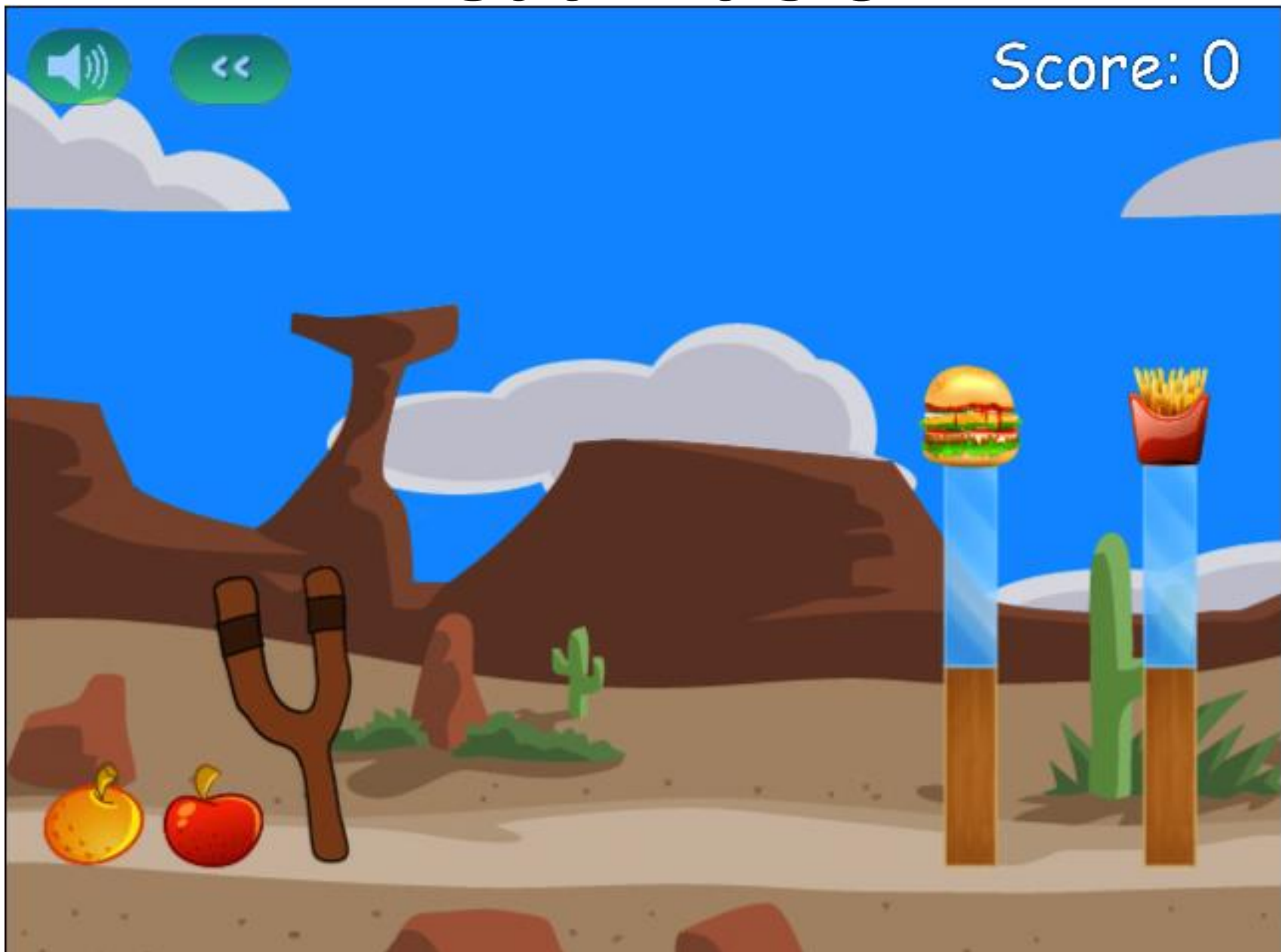
    // Iterar a través de todos los cuerpos y dibujarlos en el lienzo del juego
    for (var body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();

        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())
            }
        }
    }
}
```

For inicializa `body` a través de `GetBodyList()` que devuelve el primer cuerpo en el mundo hasta terminar la lista, entonces sale del bucle

Comprueba si el cuerpo tiene una entidad adjunta y llama a `entities.draw()` pasándole el objeto `entity`, la posición y el ángulo

Visualización



Proceso (III)

- Una vez cargado el nivel, el juego muestra una vista panorámica de modo que se puede ver perfectamente a los chicos malos y luego la vista regresa a la honda.
- Así se ven todas las entidades dibujadas adecuadamente como aparecen en el canvas depurador.
- El pixel extra que se añade al método `draw()` asegura que todos los objetos apilados estrechamente uno al lado del otro.

Animar el mundo Box2D

Proceso (I)

- Es posible animar el mundo **Box2D** llamando al método **step()** del objeto **world** y pasándole el intervalo de paso de tiempo como parámetro, lo cual es un poco complicado.
- Siguiendo las recomendaciones del manual de **Box2D**:
 - Idealmente se debe utilizar un paso de tiempo fijo para obtener mejores resultados ya que las variables de paso de tiempo son difíciles de depurar.
 - **Box2D** trabaja mejor con un paso de tiempo de alrededor de **1/60** de segundo y no se debe utilizar un paso de tiempo más grande de **1/30** de segundo.
 - A mayor paso de tiempo más problemas con las colisiones y los cuerpos empiezan a pasar a través de uno al otro.
- La API de **requestAnimationFrame** puede variar la frecuencia a la cual se llama al método **animate()** a través de los navegadores y máquinas.
 - Una forma de evitarlo es medir el tiempo transcurrido desde la última llamada de **animate()** y pasar esa diferencia como un paso de tiempo para **Box2D**.

Proceso (II)

- Sin embargo, al cambiar de pestaña del navegador y volver a la pestaña del juego, el navegador llamará a `animate()` con menos frecuencia y este paso de tiempo podría ser más grande que el límite superior de $1/30$ de segundo.
- Para evitarlo es necesario limitar activamente el paso de tiempo si este es más grande que $1/30$ de segundo.
- Definir `Step()` dentro del objeto `box2d` que toma un intervalo de tiempo como parámetro y llama a `Step()` del objeto `world`.

```
step: function(timeStep){  
    // velocidad de las iteraciones = 8  
    // posición de las iteraciones = 3  
    if(timeStep > 2/60){  
        timeStep = 2/60  
    }  
    box2d.world.Step(timeStep,8,3);  
},
```

Toma el paso en segundos y lo pasa a `world.Step()`.

Se llama a este método desde `game.animate()` después de calcular el paso de tiempo.

Proceso (III)

```

animate: function(){
    // Animar el fondo
    game.handlePanning();

    // Animar los personajes
    var currentTime = new Date().getTime();
    var timeStep;
    if (game.lastUpdateTime){
        timeStep = (currentTime - game.lastUpdateTime)/1000;
        box2d.step(timeStep);
    }
    game.lastUpdateTime = currentTime;

    // Dibuja el fondo con desplazamiento de paralaje
    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);

    // Dibujar la honda
    game.context.drawImage(game.slingshotImage, game.slingshotX-game.offsetLeft, game.slingshotY);

    // Dibujar todos los cuerpos
    game.drawAllBodies();

    // Dibujar el frente de la honda
    game.context.drawImage(game.slingshotFrontImage, game.slingshotX-game.offsetLeft, game.slingshotY);

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

```

Se calcula `timeStep` como la diferencia entre `lastUpdateTime` y `currentTime` y luego se llama a `box2d.step()`

Se guarda el tiempo actual en la variable `game.lastUpdateTimeCurrentTime`

La primera vez que se llama a `animate()` `game.lastUpdateTimeCurrentTime` es indefinido, por tanto no se calcula `timeStep` ni se llama a `box2dstep()`

Cargar el héroe

Proceso (I)

- En este punto es posible implementar más estados o modos del juego:
- El primer estado es **load-next-hero**, en el cual el juego necesita contar y verificar el número de héroes y villanos que quedan en el juego y actuar en consecuencia:
 - Si todos los villanos se han ido, el juego cambia al estado **level-success**.
 - Si todos los héroes se han ido, el juego cambia al estado **level-failure**.
 - Si aún quedan héroes, el juego coloca al primer héroe encima de la honda y luego cambia al estado **wait-for-firing**.
- Esto se hace creando un método llamado **game.countHeroesAndVillians()** y modificando el método **game.handlePannig()**

Proceso (II)

```
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:
        //Vista panorámica donde el héroe se encuentra actualmente es...
    }
}
```

Interactúa a través de todos los cuerpos y almacena los héroes en el *array* `game.heroes` y a los villanos en el *array* `game.villains`

Proceso (III)

```

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

    // Comprobar si algún villano está vivo, si no, termine el nivel (éxito)
    if (game.villains.length == 0){
        game.mode = "level-success";
        return;
    }

    // Comprobar si hay más héroes para cargar, si no terminar el nivel (fallo)
    if (game.heroes.length == 0){
        game.mode = "level-failure";
        return;
    }

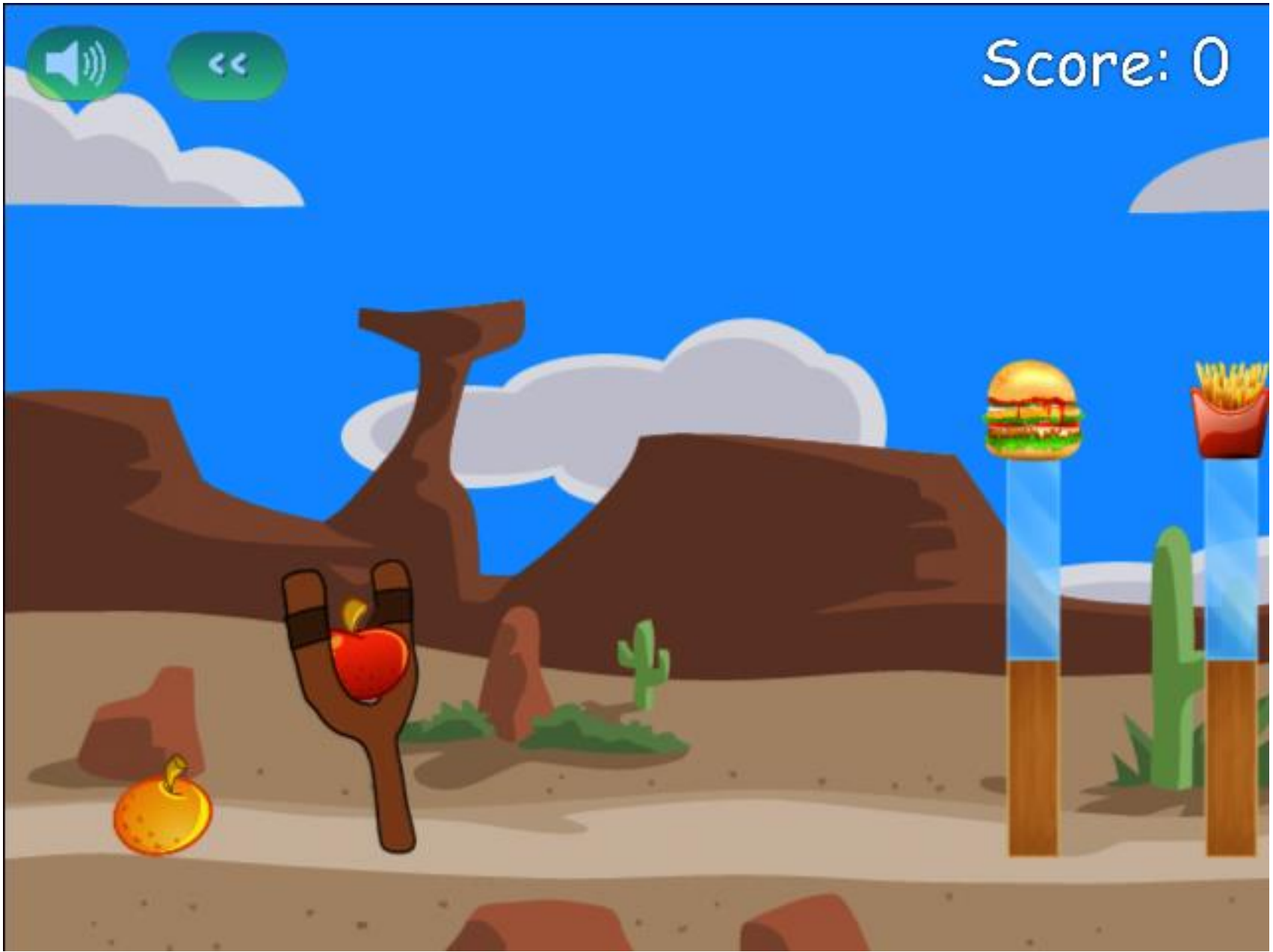
    // Cargar el héroe y establecer el modo de espera para disparar (wait-for-firing)
    if (!game.currentHero){
        game.currentHero = game.heroes[game.heroes.length-1];
        game.currentHero.SetPosition({x:180/box2d.scale,y:200/box2d.scale});
        game.currentHero.SetLinearVelocity({x:0,y:0});
        game.currentHero.SetAngularVelocity(0);
        game.currentHero.SetAwake(true);
    } else {
        // Espere a que el héroe deje de rebotar y se duerma y luego cambie a espera para disparar (wait-for-firing)
        game.panTo(game.slingshotX);
        if (!game.currentHero.IsAwake()){
            game.mode = "wait-for-firing";
        }
    }
}
},

```


Proceso (IV)

- El método `count.HeroesAndVillians()` itera a través de todos los cuerpos en el mundo y almacena los héroes en el *array* `game.heroes` y a los villanos en el *array* `game.villians`.
- Dentro del método `handlePanning()` cuando `game.mode` es `load.next.hero` hacer lo siguiente:
 - Llamar a `countHeroesandVillians()`.
 - Verificar si la cuenta de villanos o héroes es `0`, si es así fijar `game.mode` a `level-success` o `level-faillure` respectivamente.
 - En caso contrario salvar el último héroe en el *array* `game.heroes` dentro de la variable `game.currentHero` y fijar su posición a un punto en el aire por encima de la honda.
 - Fijar su velocidad angular y lineal a `0`.
 - Despertar al cuerpo en caso de que esté dormido.

Visualización



Disparar al héroe

Proceso (I)

- Para disparar el héroe se utilizan tres estados:
 - **wait-for-firing**: El juego hace una vista panorámica sobre la honda y espera a que el ratón sea presionado y arrastrado mientras que el puntero está sobre el héroe. Cuando esto sucede cambia al estado **firing**.
 - **firing**: El juego mueve el héroe con el ratón hasta que el botón del ratón es liberado, cuando esto sucede empuja el héroe con un impulso basado en la distancia desde la honda y cambia al estado **fired**.
 - **fired**: El juego hace una panorámica para seguir al héroe hasta que vaya a descansar o salga de los límites del nivel. Entonces el juego elimina al héroe del mundo del juego y regresa al estado **load-next-hero**.

Proceso (II)

- Ahora se implementa el método `mouseOnCurrentHero()` dentro del objeto `game` para probar si el puntero del ratón está posicionado sobre el héroe actual.

```
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);
},
```

- Calcula la distancia entre el centro del héroe actual y la ubicación del ratón y lo compara con el radio del héroe actual para comprobar si el ratón está ubicado sobre el héroe.
- Esto funciona ya que todos los héroes son circulares, para implementar otro tipo de héroes es necesario utilizar un método más complejo.
- A continuación implementamos los tres estados dentro del método `handlePanning()`



Proceso (III)

```
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";
        game.slingshotReleasedSound.play();
        var impulseScaleFactor = 0.75;

        // Coordenadas del centro de la honda (donde la banda está atada a la honda)
        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());
    }
}

if (game.mode == "fired"){
    //Vista panorámica donde el héroe se encuentra actualmente...
    var heroX = game.currentHero.GetPosition().x*box2d.scale;
    game.panTo(heroX);

    //Y esperar hasta que deja de moverse o está fuera de los límites
    if(!game.currentHero.IsAwake() || heroX<0 || heroX >game.currentLevel.foregroundImage.width ){
        // Luego borra el viejo héroe
        box2d.world.DestroyBody(game.currentHero);
        game.currentHero = undefined;
        // y carga el siguiente héroe
        game.mode = "load-next-hero";
    }
}
```

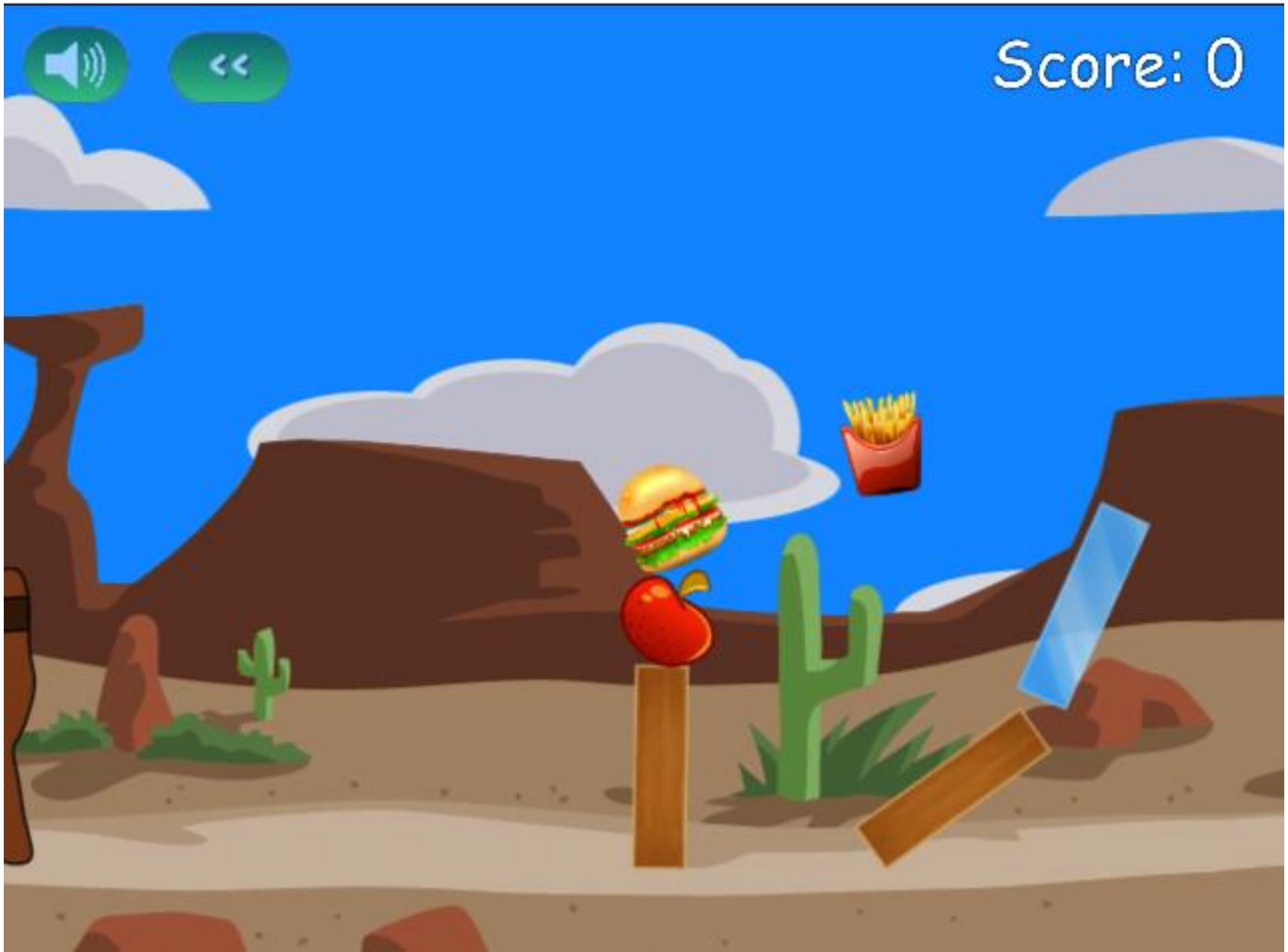
Proceso (IV)

- Cuando el estado es **fired** se hace una panorámica de la pantalla hasta el héroe y se espera a que el héroe vaya a descansar o salga de los límites del nivel. Entonces el juego elimina al héroe del mundo del juego utilizando el método **DestroyBody()** y regresa al estado **load-next-hero**.
- Una vez que el héroe deja de rodar o va fuera de los límites del nivel es eliminado del juego y se carga el próximo héroe en la honda.
- Cuando todos los héroes se han ido, el juego se detiene y espera.
- Por tanto lo último que hay que implementar es finalizar el nivel.

Proceso (IV)

- Cuando el estado es **wait-for-firing** y el ratón está siendo arrastrado se cambia de estado a **firing** si el puntero del ratón está posicionado sobre el héroe.
 - Si no es así se hace una vista panorámica de la pantalla hacia el cursor.
 - Si el ratón no está siendo arrastrado se hace una vista panorámica hacia la honda
- Cuando el estado es **firing** y el botón del ratón está presionado se fija la posición del héroe a la posición del ratón y se hace una vista panorámica hacia la honda.
 - Cuando el botón del ratón es liberado, se fija el estado a **fired** y se aplica un impulso al héroe utilizando el método **ApplyImpulse()** del objeto **b2Body**.
 - Este método toma el impulso como un parámetro en forma de un objeto **b2Vec2**.
 - Se fijan los valores de **x** e **y** del vector del impulso como un múltiplo de la distancia **x** e **y** del héroe desde arriba de la honda. Este factor de escala de impulso resulta de experimentar con varios valores.

Visualización



Finalizar el nivel

Proceso (I)

- Una vez que el nivel termina se parará el bucle de animación del juego y se muestra la pantalla de fin de nivel.
- Dicha pantalla le dará al usuario opciones para volver a jugar el nivel actual, ir al siguiente nivel o volver a la pantalla de selección de nivel.
- Lo primero que hay que hacer es añadir los manejadores de evento **onclick** para el elemento **div endingscreen** y añadirlo al archivo **index.html** después de las otras capas del juego.

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

Proceso (II)

```
/* Pantalla final */
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;
}
```

Añadir las hojas de estilo
en el archivo `styles.css`

Proceso (III)

- Ahora que la pantalla de fin de nivel está lista se implementa el método `showEndingScreen()` dentro del objeto `game` que mostrará el elemento `div endingscreen`.

```
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();
},
```

Proceso (IV)

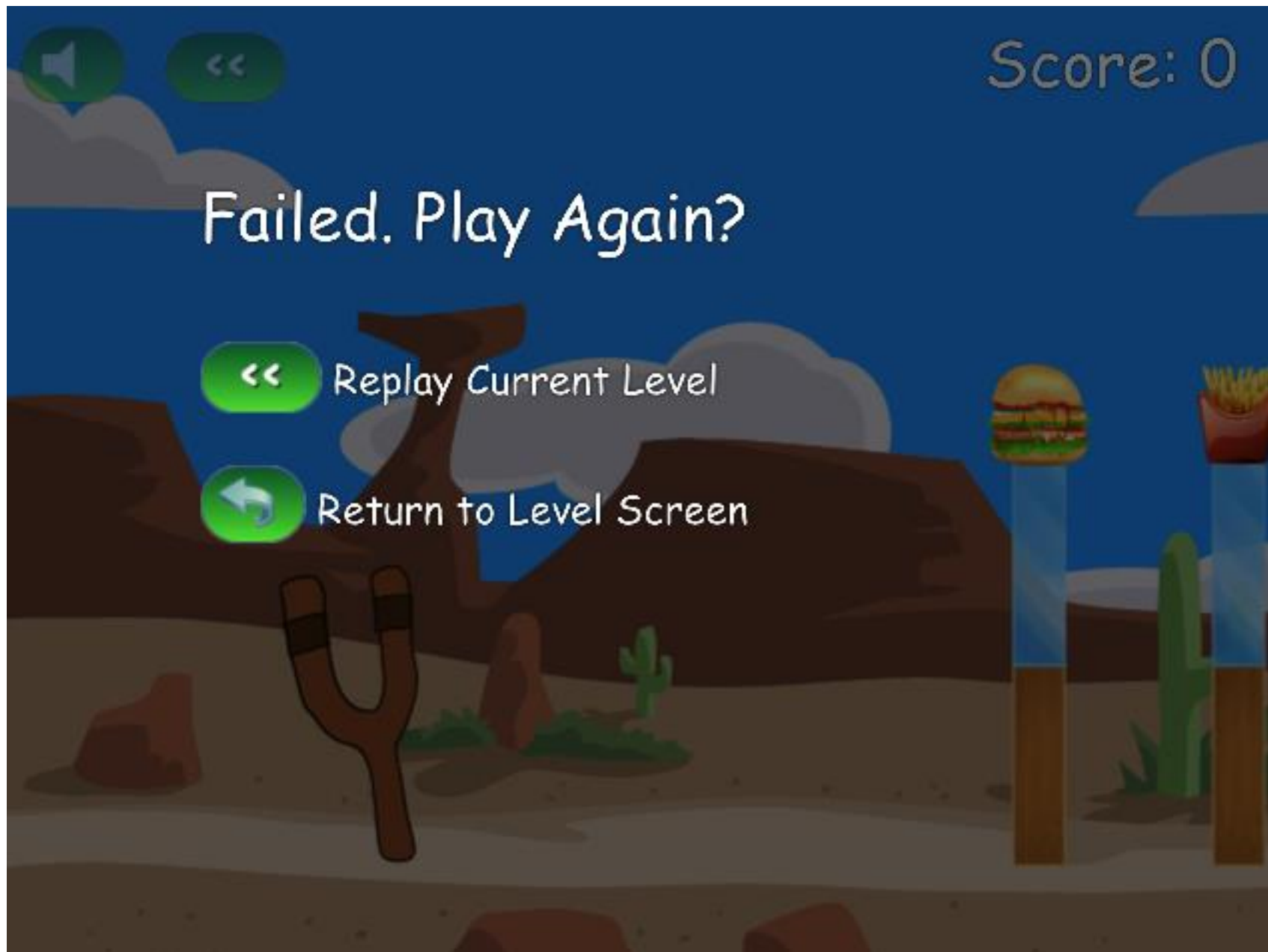
- El método `showEndingScreen()` muestra diferentes mensajes basados en el valor de `game.mode`.
- La opción para jugar el próximo nivel se muestra si el jugador tuvo éxito y el nivel actual no fue el último nivel del juego.
- Si el jugador no tuvo éxito o el nivel actual fue el nivel final la opción se esconde.
- Ahora se maneja `level-success` y `level-failure` dentro de `handlePanning()` del objeto `game`.

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

Proceso (V)

- Cuando el método `game.mode` es `level-sucess` o `level-failure` el juego hace una vista panorámica hacia atrás y a la izquierda y luego fija la propiedad `game.ended` a `true` y finalmente muestra la pantalla final.

Visualización



Daños por colisión

Proceso (I)

- Grabar las colisiones utilizando un escuchador de contacto (**contact listener**) y sobrescribir su método **PostSolve()**.
- Se añade el escuchador al mundo inmediatamente después de haber sido creado en el método **init()** del objeto **box2d**.

Proceso (II)

```
init:function(){
    // Configurar el mundo de box2d que hará la mayoría de los cálculos de la física
    var gravity = new b2Vec2(0,9.8); //Declara la gravedad como 9,8 m / s ^ 2 hacia abajo
    var allowSleep = true; //Permita que los objetos que están en reposo se queden dormidos y se excluyan de los cálculos
    box2d.world = new b2World(gravity,allowSleep);

    // Configurar depuración de dibujo
    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]);
        // Este listener es llamado con mucha frecuencia. Filtra los impulsos muy pequeños.
        // Después de probar diferentes valores, 5 parece funcionar bien
        if(impulseAlongNormal>5){
            // Si los objetos tienen una salud, reduzca la salud por el valor del impulso
            if (entity1.health){
                entity1.health -= impulseAlongNormal;
            }

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

PostSolve() es llamado en cada colisión.

Si impulseAlongNormal es menor que un límite, se ignora la colisión.

Proceso (III)

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

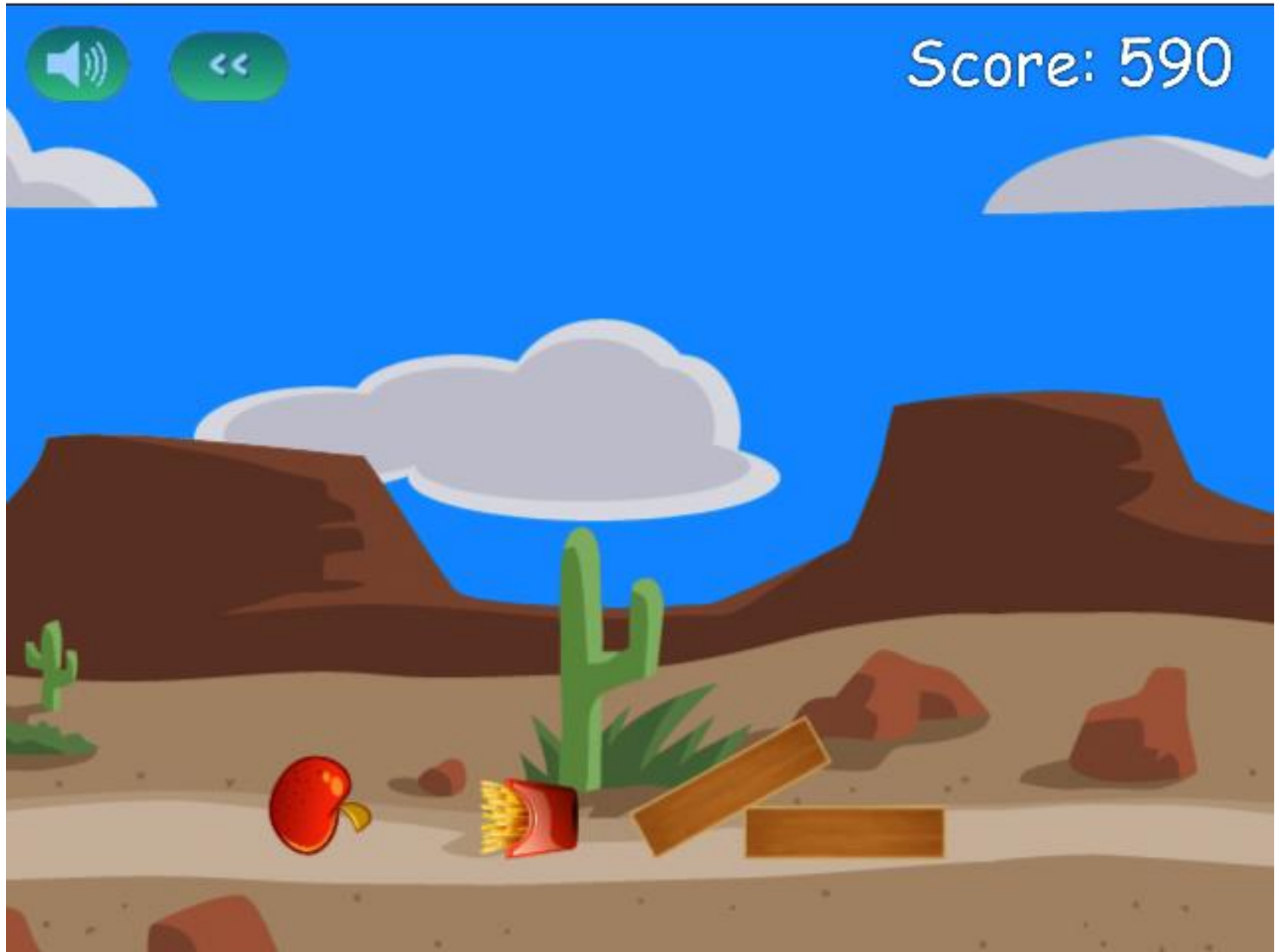
    // Iterar a través de todos los cuerpos y dibujarlos en el lienzo del juego
    for (var body = box2d.world.GetBodyList(); body; body = body.GetNext()) {
        var entity = body.GetUserData();

        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())
            }
        }
    }
}
```

Si el cuerpo va fuera de los límites del nivel o la entidad pierde toda su salud, se utiliza `DestroyBody()` para eliminar el cuerpo.

Si la entidad es un villano, se añade el valor de sus calorías a la puntuación del juego.

Visualización



Dibujar la banda de la honda

Proceso (I)

- La banda de la honda será una línea marrón delgada desde el final de la honda hasta el extremo del héroe.
- La banda se dibuja solo cuando el juego está en modo **firing**.
- Esto se hace utilizando el método `drawSlingshotBand()` dentro del objeto `game`.

Proceso (II)

```
drawSlingshotBand:function(){
    game.context.strokeStyle = "rgb(68,31,11)"; // Color marrón oscuro
    game.context.lineWidth = 6; // Dibuja una línea gruesa

    // Utilizar el ángulo y el radio del héroe para calcular el centro del héroe
    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();
    // Iniciar la línea desde la parte superior de la honda (la parte trasera)
    game.context.moveTo(game.slingshotX+50-game.offsetLeft, game.slingshotY+25);

    // Dibuja línea al centro del héroe
    game.context.lineTo(heroX-game.offsetLeft,heroY);
    game.context.stroke();

    // Dibuja el héroe en la banda posterior
    entities.draw(game.currentHero.GetUserData(),game.currentHero.GetPosition(),game.currentHero.GetAngle());

    game.context.beginPath();
    // Mover al borde del héroe más alejado de la parte superior de la honda
    game.context.moveTo(heroFarEdgeX-game.offsetLeft,heroFarEdgeY);

    // Dibujar línea de regreso a la parte superior de la honda (el lado frontal)
    game.context.lineTo(game.slingshotX-game.offsetLeft +10,game.slingshotY+30)
    game.context.stroke();
},
```

drawSlingshotBand()
lo llamamos desde
game.animate(),
después de dibujar
todos los otros cuerpos.

Proceso (III)

```
animate:function(){
    // Animar el fondo
    game.handlePanning();

    // Animar los personajes
    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;

    // Dibuja el fondo con desplazamiento de paralaje
    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);

    // Dibujar la honda
    game.context.drawImage(game.slingshotImage,game.slingshotX-game.offsetLeft,game.slingshotY);

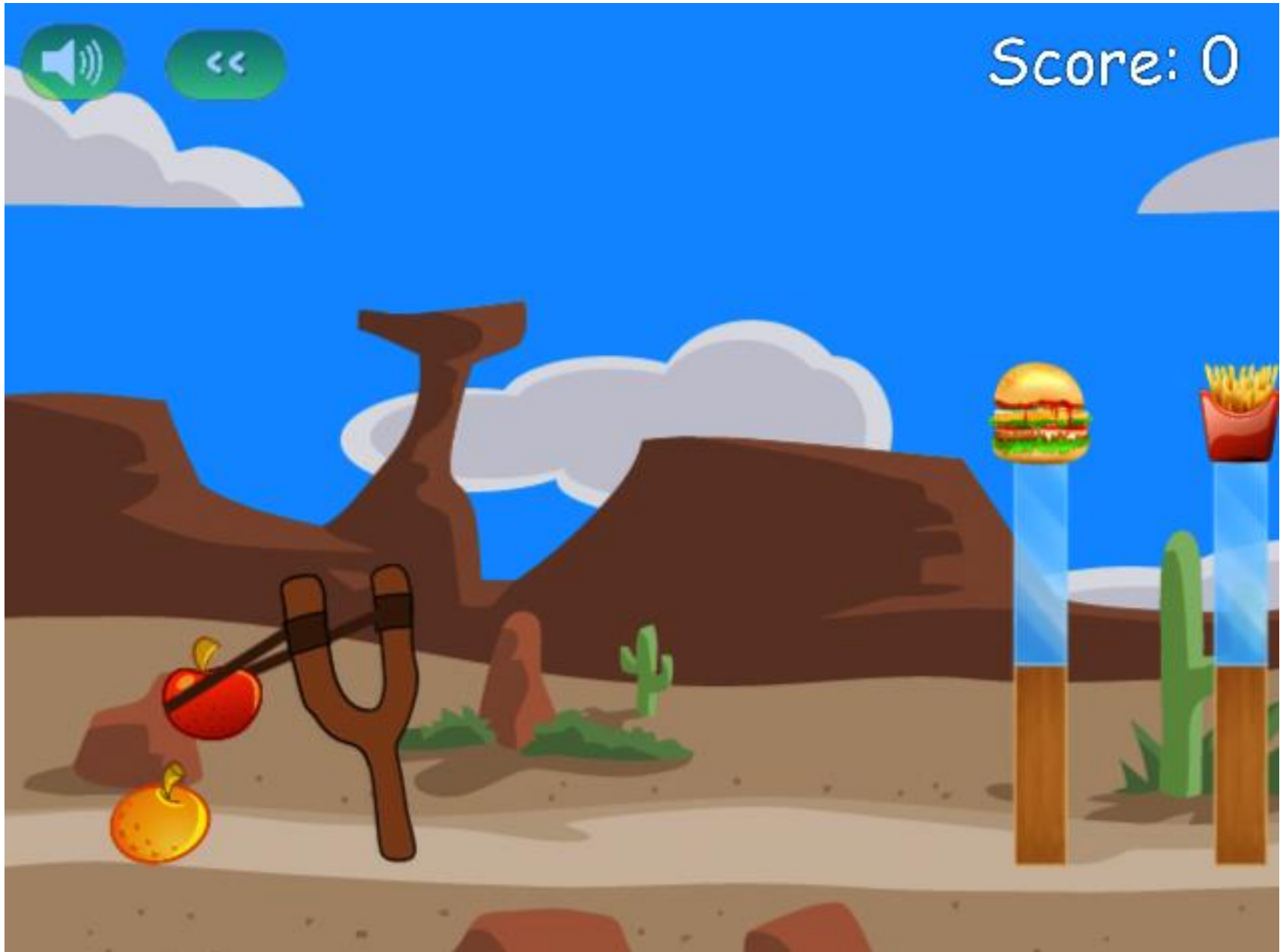
    // Dibujar todos los cuerpos
    game.drawAllBodies();

    // Dibujar la banda cuando estamos disparando un héroe
    if(game.mode == "wait-for-firing" || game.mode == "firing"){
        game.drawSlingshotBand();
    }

    // Dibujar el frente de la honda
    game.context.drawImage(game.slingshotFrontImage,game.slingshotX-game.offsetLeft,game.slingshotY);

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

Visualización



Cambiar niveles

Proceso (I)

- Implementar los botones para reiniciar un nivel y pasar al siguiente nivel.
- Esto se hace utilizando los métodos `resetLevel()` y `startNextLevel()` dentro del objeto `game`.

```
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);
},
```

Ambos métodos cancelan cualquier bucle `animationFrame`, reinician el juego, la variable `lastUpdateTime` y llaman al método `levels.load()` con el número de nivel apropiado.

Proceso (II)

- Llamar a los métodos anteriores desde el evento **onclick** de las imágenes correspondientes en las capas **scorescreen** y **endingscreen**.
- Esto se hace desde el archivo **index.html**.

```
<div id="scorescreen" class="gamelayer">
  
  
  <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();"> Replay Current Level</p>
    <p id="playnextlevel" onclick="game.startNextLevel();"> Play Next Level </p>
    <p id="returntolevelscreen" onclick="game.showLevelScreen();"> Return to Level Screen</p>
  </div>
</div>
```

Añadir sonido

Proceso (I)

- Añadir sonido hace el jugador se sienta más inmerso en el juego.
- Utilizar efectos de sonido para cuando:
 - La honda es liberada.
 - Un héroe o un villano rebota.
 - Uno de los bloques es destruido.
- Añadir música de fondo, con la posibilidad de apagarla si se desea.
- Los sonidos (en formato **MP3** y **OGG**) están disponibles en la carpeta de Recursos en el aula virtual.
- Cargarlos en el método **init()** del objeto **game**.

Proceso (II)

```
init: function(){
    // Inicialización de objetos
    levels.init();
    loader.init();
    mouse.init();

    // Cargar todos los efectos de sonido y música de fondo

    //"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')
    };

    // Ocultar todas las capas del juego y mostrar la pantalla de inicio
    $('.gameLayer').hide();
    $('#gamestartscreen').show();

    //Obtener el controlador para el lienzo de juego y el contexto
    game.canvas = document.getElementById('gamecanvas');
    game.context = game.canvas.getContext('2d');
},
```

`loader.loadSound()`
permite cargar diferentes
sonidos y los guarda para
referencias posteriores

**Añadir sonido para rotura y
rebote**

Proceso (I)

- Asociar los efectos de sonido a las entidades y ejecutarlas en el momento adecuado.
- Modificar el método `create.entities()` y fijar los sonidos de rotura y rebote en la definición de las entidades.
- Adjuntar sonidos a las entidades durante la creación para que cada entidad pueda tener su propio sonido de rotura y rebote personalizados.

```

create:function(entity){
    var definition = entities.definitions[entity.name];
    if(!definition){
        console.log ("Undefined entity name",entity.name);
        return;
    }
    switch(entity.type){
        case "block": // Rectángulos simples
            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": // Rectángulos simples
            // No hay necesidad de salud. Estos son indestructibles
            entity.shape = "rectangle";
            // No hay necesidad de sprites. Éstos no serán dibujados en absoluto
            box2d.createRectangle(entity,definition);
            break;
        case "hero": // Círculos simples
        case "villain": // Pueden ser círculos o rectángulos
            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;
    }
},

```

Proceso (II)

- Ejecutar los sonidos cuando los eventos tengan lugar, por ejemplo el sonido de rebote cuando se detecte una colisión dentro del objeto `b2ContactListener`.

```
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]);
    // Este listener es llamado con mucha frecuencia. Filtra los impulsos muy pequeños.
    // Después de probar diferentes valores, 5 parece funcionar bien
    if(impulseAlongNormal > 5){
        // Si los objetos tienen una salud, reduzca la salud por el valor del impulso
        if (entity1.health){
            entity1.health -= impulseAlongNormal;
        }

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

        // Si los objetos tienen un sonido de rebote, reproducirlos
        if (entity1.bounceSound){
            entity1.bounceSound.play();
        }

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

Durante la colisión se verifica si la entidad tiene una propiedad `bounceSound` y si es así ejecuta un sonido

Proceso (III)

- Ejecutar el sonido de rotura cada vez que un objeto sea destruido.
- Esto se hace dentro del método `draw2AllBodies()` el objeto `game`.

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

    // Iterar a través de todos los cuerpos y dibujarlos en el lienzo del juego
    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())
            }
        }
    }
},
```

Se verifica si la entidad que ha sido destruida tiene una propiedad `breakSound` y si es así ejecuta un sonido

Proceso (IV)

- Ejecutar `slingshotReleasedSound` cuando cambie `game.mode` de `firing` a `fired` dentro del método `handlePanning()`.

```
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;

        // Coordenadas del centro de la honda (donde la banda está atada a la honda)
        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());
    }
}
```

- Al ejecutar el juego se escucharán los efectos de sonido cuando el héroe dispare, cuando vuelva a chocar contra algo.

Añadir música de fondo

Proceso (I)

- Crear algunos métodos para iniciar, detener y alternar la música de fondo.

```
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; // Ir al comienzo de la canción
},
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";
    }
},
```


Proceso (II)

- **startBackgroundMusic()** llama primero al objeto **play** de **backgroundMusic** y luego fija la imagen del botón de alternar la música para mostrar un altavoz con sonido.
- **stopBackgroundMusic()** fija la imagen del botón de alternar la música para mostrar un altavoz sin sonido. Este llama a **pause()** del objeto **backgroundMusic** y fija el audio de nuevo al principio de la canción fijando su propiedad **currentTime** a 0.
- **toggleBackgroundMusic()** comprueba si la música está actualmente en pausa y llama a **pause()** o **play()** y luego fija la imagen de alternar apropiadamente.
- Llamar al **startBackgraoundMusic()** cuando el juego comienza desde **game.start()**

Proceso (III)

- Llamar a `toggleBackgroundMusic` desde el evento `onclick` del botón conmutador de música dentro de la capa `scorescreen`.
- Añadir el siguiente código al archivo `index.html`

```
<div id="scorescreen" class="gameLayer">
  
  
  <span id="score">Score: 0</span>
</div>
```

Resumen

- A través de estas diapositivas se ha creado:
 - Un marco para el desarrollo de un juego basado en HTML5
 - Un sistema de nivel
 - Un cargador de *assets*
 - La animación del juego
- Se ha explicado el motor de la física.
- Se ha llevado a cabo la integración añadiendo menú de opciones, efectos de sonido y música.
- Se han utilizado todos los elementos esenciales esperados de un buen juego en HTML5.