Un juego en 500 líneas: Galaga

Juan Antonio Recio García (jareciog@fdi.ucms)
Guillermo Jiménez Díaz (gjimenez@ucm.es)
Apuntes creados por Pedro Antonio González Calero (pedro@fdi.ucm.es)

Estructura del juego

El objetivo es hacer una versión simplificada de *Galaga*, que se muestra en la figura 1

Figure 1: Galaga

Se puede acceder al juego implementado en http://cykod.github.com/AlienInvasion

Un juego de este tipo consta esencialmente de los siguientes elementos:

- carga de recursos
- pantalla de bienvenida
- manejo de sprites
- gestión de la entrada del usuario
- detección de colisiones
- un bucle principal que lo ensambla todo

El fichero HTML que carga el juego básicamente lo que hace es definir un *canvas* que es la zona de la ventana del navegador donde se presentará el juego:

```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <meta charset="UTF-8"/>
  <title>Alien Invasion</title>
  <link rel="stylesheet" href="base.css" type="text/css" />
</head>
<body>
  <div id='container'>
    <canvas id='game' width='320' height='480'></canvas>
  </div>
  <script src='engine.js'></script>
  <script src='game.js'></script>
</body>
</html>
```

La hoja de estilo que se carga de `base.css` lo que hace es inicializar toda la información de estilo que pueda contener la página y establece las características de tamaño, márgenes y color del canvas:

```
/* Center the container */
#container {
  padding-top:50px;
  margin:0 auto;
  width:480px;
}
/* Give canvas a background */
canvas {
  background-color: black;
}
```

El canvas

Lo primero que hay que hacer para poder pintar en el canvas es obtener su *contexto* que es el objeto que incluye el API de llamadas que permiten pintar en el canvas. En el fichero `game.js` que se carga desde el anterior fichero HTML incluiríamos:

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

var ctx = canvas.getContext && canvas.getContext('2d');
if(!ctx) {
   // No 2d context available, let the user know
   alert('Please upgrade your browser');
} else {
  startGame();
}
function startGame() {
 // Let's get to work
}
```

Para pintar en el canvas empezamos dibujando un rectángulo amarillo (`#FFFF00`), y encima uno azul semitransparente, usando el esquema de colores *rgba* que permite especificar como cuarto parámetro el porcentaje de transparencia (*alpha*) :

```
function startGame() {
  ctx.fillStyle = "#FFFF00";
  ctx.fillRect(50,100,380,400);

  ctx.fillStyle = "rgba(0,0,128,0.5)";
  ctx.fillRect(25,50,380,400);
}
```

Para dibujar una imagen de mapa de bits (un *sprite*) en el canvas usamos el método `drawImage` del contexto:

```
drawImage( image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)
```

que copia el fragmento de la imagen que empieza en la posición `(sx,sy)` de ancho `sWidth` y alto `sHeight` y lo coloca en el canvas en las coordenadas `(dx, dy)` cambiando sus dimensiones para que tenga `dWidth` de ancho y `dHeight` de alto.

Como la carga de imágenes en HTML es asíncrona, la función de pintado la debemos incluir en una retrollamada que se asocia al evento `onload` de la imagen, y para evitar problemas de caché es importante que primero se asocie la retrollamada al evento y luego se indique la ruta de la imagen a cargar.

Carga de toda la imagen en la posición `(100, 100)` del canvas:

```
 var img = new Image();
  img.onload = function() {
    ctx.drawImage(img,100,100);
  }
  img.src = 'images/sprites.png';
```

o carga sólo de la nave del jugador:

```
ctx.drawImage(img,0,0,40,40,100,100,40,40);
```

como se muestra en la figura 2

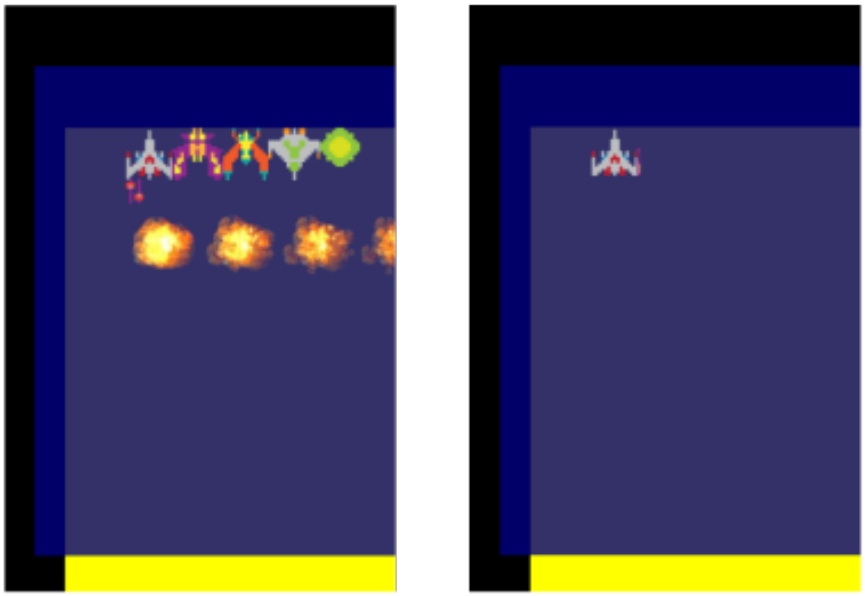

Figure 2: Carga de sprites

El evento `onload` es un evento HTML que permite asociar código JavaScript con eventos que tienen lugar en la página HTML que carga eso código. `onload` es un evento que se lanza cuando se ha cargado un objeto en la página, en este caso una imagen. También se generan eventos cuando se pulsan teclas o se utiliza el ratón y serán esos eventos los que utilicemos para recibir la entrada del jugador.

Arquitectura del juego

Objetos principales del juego

El juego lo organizamos en torno a 3 objetos básicos:

- un objeto `SpriteSheet` que gestiona la carga y dibujado de los sprites
- un objeto `Game` que se encarga de la gestión del juego a alto nivel: procesamiento de entrada, bucle principal y gestión de las pantallas
- un objeto `GameBoard` que gestiona el movimiento y colisión de los sprites

SpriteSheet: Carga de sprites

El objeto encargado de gestionar la carga y el dibujado de los sprites, que en este caso son los que se muestran en la figura 3

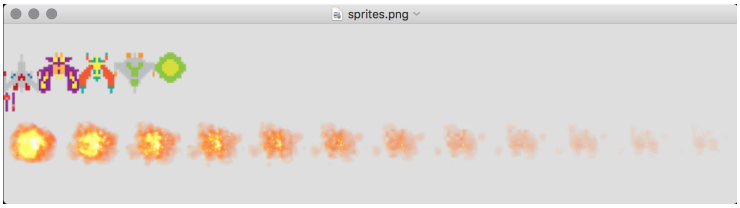

Figure 3: Sprite sheet del juego

Para pintar fácilmente los sprites utilizaremos el objeto SpriteSheet. Simplemente indicaremos sus coordenadas dentro de la hoja de sprites, su número de frames (para animaciones) y qué hacer cuando se cargue:

```
SpriteSheet.load({
  ship: {sx:0, sy:0, w:38, h:43, frames:3}
  }, function(){
    SpriteSheet.draw(ctx, "ship", 0,0);
    SpriteSheet.draw(ctx, "ship", 100,50);
    SpriteSheet.draw(ctx, "ship", 150,100,1);
  });
```

La funcionalidad anterior se implementa así (en engine.js):

```
var SpriteSheet = new function() {
  this.map = { };
  this.load = function(spriteData, callback) {
    this.map = spriteData;
    this.image = new Image();
    this.image.onload = callback;
    this.image.src = 'images/sprites.png';
  };
  this.draw = function(ctx,sprite,x,y,frame) {
    var s = this.map[sprite];
    if(!frame) frame = 0;
    ctx.drawImage(this.image,
                  s.sx + frame * s.w,
                  s.sy,
                  s.w, s.h,
                  x,   y,
                  s.w, s.h);
  };
}
```

Como sólo queremos que haya un objeto `SpriteSheet`, directamente le asignamos el resultado de invocar a la constructora con `new`. La función encargada de la carga, `SpriteSheet.load` recibe:

- Un objeto donde se le hace corresponder a cada nombre de sprite ("ship", "enemy_ship","explosion", ...) las coordenadas, el ancho y el alto dentro de la hoja de sprites, además del número de frames (fotogramas). Las animaciones las podemos generar como una secuencia de fotogramas, cada uno formado por un sprite diferente. En este caso sólo las explosiones tienen más de un fotograma y los distintos fotogramas de la misma entidad están alineados a la misma altura, colocando cada uno a la derecha del anterior, de forma que todos tienen la misma coordenada `y`, alto y ancho y sólo se diferencian en la coordenada `x`: `s.sx + frame * s.w`
- La función que será invocada cuando se produzca el evento (`onload`) de carga de la imagen, que es lo último que ocurre en la inicialización y carga de recursos

Para probarlo añadiremos el código que pinta el "ship" en tres posiciones dentro de nuestro método `startGame()`.

EJERCICIO: Consigue realizar la animación de la explosión mediante setInterval();

Game: gestión a alto nivel

El objeto `Game` carga, inicializa y coordina los elementos del juego encargándose de las siguientes tareas:

- Inicialización del canvas, obteniendo el contexto donde se dibujará el contenido del juego
- Inicialización de la gestión de la entrada del usuario
- Carga de los sprites
- Puesta en marcha del bucle principal del juego

```
var Game = new function() {

  // Inicialización del juego
  // se obtiene el canvas, se cargan los recursos y se llama a callback
  this.initialize = function(canvasElementId, sprite_data, callback) {

    this.canvas = document.getElementById(canvasElementId)
    this.width = this.canvas.width;
    this.height= this.canvas.height;

    this.ctx = this.canvas.getContext && this.canvas.getContext('2d');
    if(!this.ctx) {
```

```
      return alert("Please upgrade your browser to play"); }

    this.setupInput();

    this.loop();

    SpriteSheet.load(sprite_data,callback);
  };

  ...

  this.setupInput = ...

  ...

  this.loop = ...

}
```

Como sólo queremos que haya un objeto `Game`, directamente le asignamos el resultado de invocar a la constructora con `new`.

- La definición de `Game` y los demás objetos que gestionan los distintos elementos del juego los colocamos en el archivo `engine.js`.
- Al cargar `engine.js` se asignará un nuevo objeto a la variable `Game`.
- En `game.js` por ahora sólo incluimos las inicializaciones necesarias y la invocación al método `Game.initialize` que pone el juego en marcha:

```
// le asigna un nombre a cada sprite, indicando sus dimensiones
// en el spritesheet y su número de fotogramas
var sprites = {
 ship: { sx: 0, sy: 0, w: 38, h: 43, frames: 3 }
};

// Especifica lo que se debe pintar al cargar el juego
var startGame = function() {
    SpriteSheet.draw(Game.ctx,"ship",100,100,1);
}

// Indica que se llame al método de inicialización una vez
// se haya terminado de cargar la página HTML
// y este después de realizar la inicialización llamará a
// startGame
window.addEventListener("load", function() {
  Game.initialize("game",sprites,startGame);
});
```

El código de nuestro juego se sincroniza y recibe entrada de la página HTML

a través de los eventos que se generan en esta, y que pueden ser globales a la ventana del navegador o específicos de un elemento HTML. Se asocian funciones a los eventos con la función `addEventListener` cuya documentación aparece en la figura 4

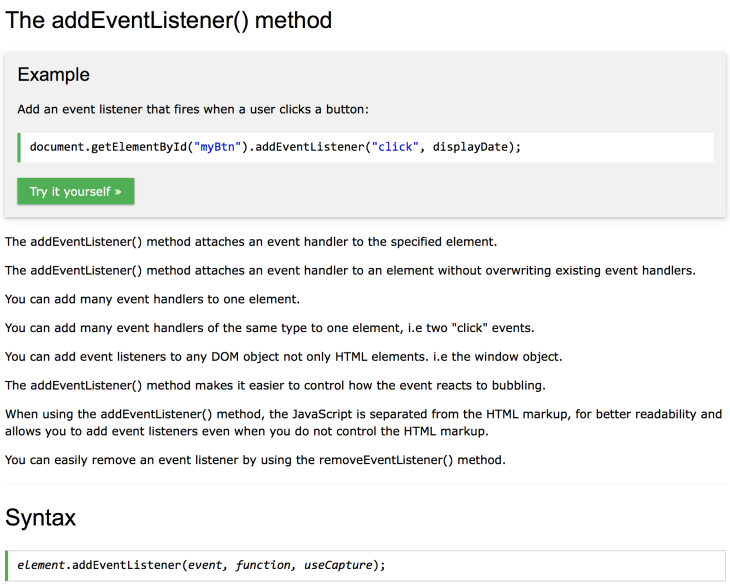

Figure 4: La función addEventListener

Game: Procesamiento de la entrada

La pulsación de las teclas generan los eventos `keydown` y `keyup` en el documento HTML, y es a esos eventos a los que asociamos funciones para procesarlas. En el objeto `Game` definimos un atributo `keys` donde almacenamos la información sobre el estado de las teclas que nos interesan: `true` si está pulsada y `false` si no lo está, cambiando de un estado a otro cuando se produzcan los eventos de pulsación o liberación de la tecla.

Para evitar que el navegador haga nada con las pulsaciones de las teclas que usamos en el juego, en la función que procesa los eventos enviamos el mensaje `preventDefault` al objeto evento para que no se ejecute el comportamiento por defecto.

Dentro de la clase `Game` añadimos el siguiente código:

```
// le asignamos un nombre lógico a cada tecla que nos interesa
var KEY_CODES = { 37:'left', 39:'right', 32 :'fire' };
```

```
this.keys = {};

this.setupInput = function() {
  window.addEventListener('keydown',function(e) {
    if(KEY_CODES[e.keyCode]) {
     Game.keys[KEY_CODES[e.keyCode]] = true;
     e.preventDefault();
    }
  },false);

  window.addEventListener('keyup',function(e) {
    if(KEY_CODES[e.keyCode]) {
     Game.keys[KEY_CODES[e.keyCode]] = false;
     e.preventDefault();
    }
  },false);
}
```

El último parámetro de `addEventListener` permite especificar si los eventos se procesan en el documento HtML desde el elemento más interno hacia el exterior (*bubbling*, con el parámetro a *false*) o viceversa (*capturing*, con el parámetro a *true*)

La función que se asocia al evento puede recibir un parámetro donde se le pasará un objeto con información general sobre el evento junto con información específica del tipo de evento, como se muestra para un evento de teclado en la figura 5

KeyboardEvent Object

Property	Description	DOM
altKey	Returns whether the "ALT" key was pressed when the key event was triggered	2
ctrlKey	Returns whether the "CTRL" key was pressed when the key event was triggered	2
charCode	Returns the Unicode character code of the key that triggered the onkeypress event	2
key	Returns the key value of the key represented by the event	3
keyCode	Returns the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key that triggered the onkeydown or onkeyup event	2
location	Returns the location of a key on the keyboard or device	3
metaKey	Returns whether the "meta" key was pressed when the key event was triggered	2
shiftKey	Returns whether the "SHIFT" key was pressed when the key event was triggered	2
which	Returns the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key that triggered the onkeydown or onkeyup event	2

Figure 5: Evento de teclado

Game: Bucle principal

Este es el *corazón* del juego. Un juego es una simulación interactiva en tiempo real que varias veces por segundo calcula el cambio de estado de las entidades que componen el juego y las vuelve a dibujar en su nuevo estado. Cuando se dibuja en el canvas, en cada pasada se vuelve a dibujar todo el contenido, por lo que debemos ser capaces de ejecutar el dibujado unas 60 veces por segundo. Para este primer juego utilizaremos una versión muy simple de bucle principal que no funciona bien en el caso general. El esquema lógico del bucle principal es el que se muestra en la figura 6

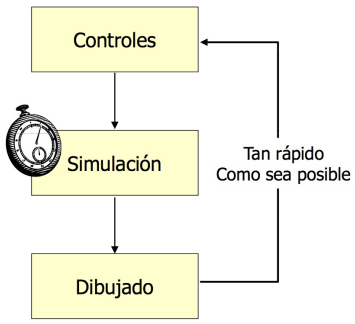

Figure 6: Bucle principal

El objeto `Game` almacena en `boards` una colección de entidades que requieren actualizar su estado y ser dibujadas por el bucle principal. Todas las entidades del juego tendrán que ser añadidas a `Game.boards` y deberán implementar los métodos `step` y `draw`, que se encargarán de actualizar su estado y dibujarlas respectivamente. Antes de cada dibujado se borra el contenido del canvas, además las entidades se dibujan siempre en el mismo orden por lo que podemos considerarlas como capas superpuestas y utilizar los atributos de transparencia si es el caso.

De nuevo en el objeto `Game` añadimos este código:

```
var boards = [];

this.loop = function() {
  var dt = 30 / 1000;

  // Cada pasada borramos el canvas
  Game.ctx.fillStyle = "#000";
  Game.ctx.fillRect(0,0,Game.width,Game.height);

  // y actualizamos y dibujamos todas las entidades
  for(var i=0,len = boards.length;i<len;i++) {
```

```
      if(boards[i]) {
        boards[i].step(dt);
        boards[i].draw(Game.ctx);
      }
    }

    setTimeout(Game.loop,30);
  };


  // Change an active game board
  this.setBoard = function(num,board) { boards[num] = board; };
};
```

Es habitual organizar las escenas de un juego por capas, como se muestra en la figura 7

Figure 7: Escenas organizadas en capas

La pantalla de título

Para escribir en un canvas se puede utilizar cualquier fuente cargada en la página. Si no queremos depender de las que puedan estar instaladas localmente en la máquina del usuario, podemos utilizar las fuentes que Google ofrece en la web http://www.google.com/fonts/, para lo cual es suficiente con incluir la carga de un css de Google, indicando qué fuente se quiere cargar:

```
<head>
  <meta charset="UTF-8"/>
  <title>Alien Invasion</title>
  <link rel="stylesheet" href="base.css" type="text/css" />
  <link href='http://fonts.googleapis.com/css?family=Bangers'
    rel='stylesheet' type='text/css'>
</head>
```

Para mostrar texto en el canvas se utiliza el contexto con

- el método `font` para especificar la fuente,
- el método `textAlign` para especificar la alineación y
- `fillText` para visualizar el texto en unas coordenadas dadas.

Creamos una clase `TitleScreen` que permite añadir una entidad de tipo pantalla de título al juego, con un método `step` que comprueba si se ha pulsado la barra espaciadora (para lo cual en algún momento debe haber estado 'arriba': `! Game.keys['fire']`) para, en su caso, iniciar una nueva partida, y un método `draw` que dibuja el contenido de la pantalla:

```
var TitleScreen = function TitleScreen(title,subtitle,callback) {
  var up = false;

  this.step = function(dt) {
    if( ! Game.keys['fire'] ) up = true;
    if( up && Game.keys['fire'] && callback ) callback();
  };

  this.draw = function(ctx) {
    ctx.fillStyle = "#FFFFFF";
    ctx.textAlign = "center";

    ctx.font = "bold 40px bangers";
    ctx.fillText(title,Game.width/2,Game.height/2);

    ctx.font = "bold 20px bangers";
    ctx.fillText(subtitle,Game.width/2,Game.height/2 + 140);
  };
};
```

y se añade al juego en la inicialización:

```
var startGame = function() {
  Game.setBoard(0,new TitleScreen("Alien Invasion",
                                  "Press fire to start playing",
                                  playGame));
}

var playGame = function() {
  Game.setBoard(0, new TitleScreen("Alien Invasion",
                                  "Game Started ..."));
}
```

con el resultado que se muestra en la figura 8

Figure 8: Pantalla de bienvenida

Recordemos que la invocación se inicia gracias a este código incluido en `game.js`:

```
window.addEventListener("load", function() {
  Game.initialize("game",sprites,startGame);
});
```

El jugador

Para añadir al jugador lo que tenemos que hacer es definir una nueva entidad que muestre el sprite `ship` cada vez que se invoque su método `draw` y desplace

su posición con cada ejecución de su método `step` en base a la pulsación de las teclas de desplazamiento:

```
var PlayerShip = function() {
   this.w =  SpriteSheet.map['ship'].w;
   this.h =  SpriteSheet.map['ship'].h;
   this.x = Game.width/2 - this.w / 2;
   this.y = Game.height - 10 - this.h;
   this.vx = 0;

   this.maxVel = 200;

   this.step = function(dt) {
     if(Game.keys['left']) { this.vx = -this.maxVel; }
     else if(Game.keys['right']) { this.vx = this.maxVel; }
     else { this.vx = 0; }

     this.x += this.vx * dt;

     if(this.x < 0) { this.x = 0; }
     else if(this.x > Game.width - this.w) {
       this.x = Game.width - this.w
     }
   }

   this.draw = function(ctx) {
     SpriteSheet.draw(ctx,'ship',this.x,this.y,0);
   }
}
```

y ahora cuando se pulsa la barra espaciadora en la pantalla de inicio la función `playGame` debe colocar el objeto `PlayerShip` entre las entidades gestionadas por `Game`:

```
var playGame = function() {
  Game.setBoard(0,new PlayerShip());
}
```

Gestión de entidades

GameBoard: Gestión de entidades

Hasta ahora hemos incluido la gestión de entidades en el propio objeto `Game`, que era el encargado de mantener la colección de entidades y recorrerla en el bucle principal. Para incluir más entidades de distintos tipos es mejor tener un objeto específicamente dedicado a su gestión: `GameBoard`. Este objeto además de la

carga, inicialización, dinámica y pintado de las entidades se ocupará también de gestionar las colisiones.

`GameBoard` gestiona una colección con todas las entidades y además lleva la cuenta del número de entidades de cada tipo. Considerando que no usamos herencia ni prácticamente tipado, el tipo de las entidades vendrá dado simplemente por un número que se almacenará en el atributo `type` de cada objeto:

```
var OBJECT_PLAYER = 1,
    OBJECT_PLAYER_PROJECTILE = 2,
    OBJECT_ENEMY = 4,
    OBJECT_ENEMY_PROJECTILE = 8,
    OBJECT_POWERUP = 16;
```

Creación de `GameBoard` e inserción de objetos:

```
var GameBoard = function() {
  var board = this;

  // The current list of objects
  this.objects = [];
  this.cnt = {};

  // Add a new object to the object list
  this.add = function(obj) {
    obj.board=this;
    this.objects.push(obj);
    this.cnt[obj.type] = (this.cnt[obj.type] || 0) + 1;
    return obj;
  };
```

nótese que cada objeto guarda una referencia al `GameBoard` en el que ha sido incluido, para, por ejemplo, borrarse a sí mismo cuando sea necesario o añadir una nueva entidad, por ejemplo cuando el jugador dispare una bala.

El borrado de una entidad de `GameBoard` tiene la complicación de que normalmente el borrado se realizará durante el bucle principal del juego cuando se está ejecutando el bucle que invoca el método `step` de cada entidad almacenada en `GameBoard.objects`. Si eliminamos un elemento del array mientras lo estamos recorriendo podemos tener resultados inesperados.

La solución es almacenar en un array auxiliar los objetos que han de ser borrados en cada pasada por el bucle principal, y una vez se haya ejecutado el `step` de todas las entidades, se borran los objetos que están almacenados en ese array:

```
// Reset the list of removed objects
  this.resetRemoved = function() { this.removed = []; };

// Mark an object for removal
  this.remove = function(obj) {
```

```
  var idx = this.removed.indexOf(obj);
  if(idx == -1) {
    this.removed.push(obj);
    return true;
  } else {
    return false;
  }
};

// Removed an objects marked for removal from the list
this.finalizeRemoved = function() {
  for(var i=0,len=this.removed.length;i<len;i++) {
    var idx = this.objects.indexOf(this.removed[i]);
    if(idx != -1) {
      this.cnt[this.removed[i].type]--;
      this.objects.splice(idx,1);
    }
  }
};
```

Donde se han utilizado algunos métodos de los arrays:

- `indexOf` que devuelve la posición de un elemento en un array o -1 si no está
- `push` que añade un elemento a un array
- `splice` que elimina a partir de la posición indicada el número de elementos que se especifica

Además, para la gestión de las entidades, `GameBoard` incluye dos funciones de orden superior que permiten aplicar una función dada a todos los objetos y encontrar el primer objeto de la colección que hace cierta una función dada:

```
// Call the same method on all current objects
this.iterate = function(funcName) {
   var args = Array.prototype.slice.call(arguments,1);
   for(var i=0,len=this.objects.length; i < len; i++) {
     var obj = this.objects[i];
     obj[funcName].apply(obj,args);
   }
};

// Find the first object for which func is true
this.detect = function(func) {
  for(var i = 0,val=null, len=this.objects.length; i < len; i++) {
    if(func.call(this.objects[i])) return this.objects[i];
  }
  return false;
};
```

En la implementación de `iterate` hay un patrón habitual en JavaScript. En `arguments` se reciben todos los parámetros de la función, incluyendo el valor de `this` en la primera posición. El método `slice` de los arrays devuelve los elementos a partir de un índice dado, pero como `arguments` no es un verdadero array (un fallo de diseño en JavaScript) no se le puede enviar el mensaje `slice`, pero sí podemos usarlo como `this` de la llamada a `slice` a través de `call`.

Y la actualización y el pintado de todos los objetos de `GameBoard`:

```
// Call step on all objects and them delete
// any object that have been marked for removal
this.step = function(dt) {
  this.resetRemoved();
  this.iterate('step',dt);
  this.finalizeRemoved();
};

// Draw all the objects
this.draw= function(ctx) {
  this.iterate('draw',ctx);
};
```

De forma que el bucle principal llama a los `step` y `draw` de todos los `GameBoard` almacenados en la variable `boards`, quienes a su vez llaman a los `step` y `draw` de los objetos que almacenan:

```
var boards = [];

this.loop = function() {
  var dt = 30 / 1000;

  // Cada pasada borramos el canvas
  Game.ctx.fillStyle = "#000";
  Game.ctx.fillRect(0,0,Game.width,Game.height);

  // y actualizamos y dibujamos todas las entidades
  for(var i=0,len = boards.length;i<len;i++) {
    if(boards[i]) {
      boards[i].step(dt);
      boards[i].draw(Game.ctx);
    }
  }

  setTimeout(Game.loop,30);
};
```

```
  // Change an active game board
  this.setBoard = function(num,board) { boards[num] = board; };
};
```

Y así podremos inicializar el juego con la nave del jugador de esta forma:forma:

```
var playGame = function() {
  var board = new GameBoard();
  board.add(new PlayerShip());
  Game.setBoard(0,board);
};
```

Colisiones: la eficiencia importa

`GameBoard` se encarga además de detectar las colisiones entre las entidades que gestiona. Una técnica habitual en la detección de colisiones es utilizar *cuadros delimitadores* (*bounding boxes*). El cuadro delimitador de un objeto es el rectángulo (en 2D) o el paralelepípedo (en 3D) más pequeño que contiene totalmente a la figura. Aunque esto es una aproximación a las verdaderas colisiones, es mucho más rápido de ejecutar y en la mayoría de los casos proporciona suficiente calidad. Es muy sencillo comprobar si dos rectángulos intersectan:

```
  this.overlap = function(o1,o2) {
    return !((o1.y+o1.h-1 < o2.y) || (o1.y > o2.y+o2.h-1) ||
             (o1.x+o1.w-1 < o2.x) || (o1.x > o2.x+o2.w-1));
  };
```

Equipados con la función `overlap` podemos recorrer la lista de entidades del juego y comprobar cuáles colisionan entre sí, con cada vuelta del bucle principal. Este cálculo puede ser costoso computacionalmente y podemos hacerlo más eficiente si tenemos en cuenta que no nos interesan las colisiones entre todos los tipos de entidades. Por ejemplo, los enemigos no pueden chocar entre sí, por lo tanto implementamos una función `collide` que permite especificar el tipo de objetos sobre los que queremos hacer las comprobaciones:

```
  this.collide = function(obj,type) {
    return this.detect(function() {
      if(obj != this) {
       var col = (!type || this.type & type) && board.overlap(obj,this);
       return col ? this : false;
      }
    });
  };
```

donde nos aprovechamos de que los identificadores de los tipos de los objetos los hemos definido como potencias de 2:

```
var OBJECT_PLAYER = 1,
```

```
    OBJECT_PLAYER_PROJECTILE = 2,
    OBJECT_ENEMY = 4,
    OBJECT_ENEMY_PROJECTILE = 8,
    OBJECT_POWERUP = 16;
```

lo que nos permite comprobar y especificar varios tipos con operaciones a nivel de bit:

```
  board.collide(enemy, OBJECT_PLAYER | OBJECT_PLAYER_PROJECTILE)
```

Los disparos

Para implementar los disparos:

- se define una constructora `PlayerMissile` que permite instanciar balas y que incluye los métodos para desplazarlas y detectar colisiones (`step`) y dibujarlas (`draw`)
- se modifica el método `step` del `PlayerShip` para que instancie un par de balas en la punta de sus cañones cada vez que se pulsa la barra espaciadora

Para los métodos de la constructora `PlayerMissile` usamos el prototipo porque es más eficiente al mantener una sola copia del código, independientemente del número de balas, que será grande.

Se añaden las coordenadas del sprite de la bala

```
var sprites = {
  missile: { sx: 0, sy: 30, w: 2, h: 10, frames: 1 },
  ...
```

y se añade la constructora y los métodos de `PlayerMissile`:

```
var PlayerMissile = function(x,y) {
  this.w = SpriteSheet.map['missile'].w;
  this.h = SpriteSheet.map['missile'].h;
  // El misil aparece centrado en 'x'
  this.x = x - this.w/2;
  // Con la parte inferior del misil en 'y'
  this.y = y - this.h;
  this.vy = -700;
};

PlayerMissile.prototype.step = function(dt)  {
  this.y += this.vy * dt;
  if(this.y < -this.h) { this.board.remove(this); }
};

PlayerMissile.prototype.draw = function(ctx)  {
```

```
  SpriteSheet.draw(ctx,'missile',this.x,this.y);
};
```

Más adelante se añade al `step` de `PlayerMissile` el código que detecta las colisiones de las balas con los enemigos. Esencialmente se comprueba si la bala colisiona con algún otro objeto, `collision`, y si es así se le resta vida `collision.hit( this.damage )`

El objeto `PlayerShip` es el responsable de crear las balas cuando se pulsa el botón de disparo. Para evitar que se lancen balas de manera continua, incluimos un atributo `reload` que lleva la cuenta desde el último disparo y no deja que se dispare otra bala antes de que transcurra el tiempo especificado en `reloadTime`. A la constructora de `PlayerShip` se añade:

```
var PlayerShip = function() {
  ...
  this.reloadTime = 0.25; // un cuarto de segundo
  this.reload = this.reloadTime;
  ...
```

y al método `step` de `PlayerShip`:

```
  this.step = function(dt) {
    ...

    this.reload-=dt;
    if(Game.keys['fire'] && this.reload < 0) {
      Game.keys['fire'] = false;
      this.reload = this.reloadTime;

      this.board.add(new PlayerMissile(this.x,this.y+this.h/2));
      this.board.add(new PlayerMissile(this.x+this.w,this.y+this.h/2));
    }
  };
```

Los enemigos

Creación de los enemigos: parametrización vs. herencia

Viendo los tipos de enemigos que hay en el juego ¿será necesario definir distintas constructoras o se podrán construir todos con la misma?

Aunque habrá distintos tipos de enemigos en el juego, esencialmente sólo se diferenciarán en el sprite que se usa para presentarlos y en los parámetros de la ecuación en que se basa su movimiento.

Los sprites se gestionan a través del `SpriteSheet` que se configura a partir de un objeto `sprites`:

```
var sprites = {
 ship: { sx: 0, sy: 0, w: 37, h: 42, frames: 1 },
 missile: { sx: 0, sy: 30, w: 2, h: 10, frames: 1 },
 enemy_purple: { sx: 37, sy: 0, w: 42, h: 43, frames: 1 },
 enemy_bee: { sx: 79, sy: 0, w: 37, h: 43, frames: 1 },
 enemy_ship: { sx: 116, sy: 0, w: 42, h: 43, frames: 1 },
 enemy_circle: { sx: 158, sy: 0, w: 32, h: 33, frames: 1 }
};
```

Y para el movimiento usaremos la misma ecuación de movimiento para todos los enemigos, variando sólo algunos parámetros:

$$vx = A + B * \sin(C * t + D)$$
$$vy = E + F * \sin(G * t + H)$$

que permite definir trayectorias sinusoidales para los enemigos, según van descendiendo por la pantalla, y donde A es la velocidad constante, B es la velocidad sinusoidal, C es el periodo y D es el desfase de la velocidad sinusoidal, en su componente horizontal, mientras que E, F, G y H son los equivalentes para la componente vertical.

Los enemigos vendrán por oleadas donde un número de ellos, configurados de forma similar, se irán creando sucesivamente con un cierto desfase y se moverán coordinadamente por la pantalla. Para la creación lo que hacemos es configurar un objeto 'blueprint' (*boceto*) con los atributos iniciales de todos los enemigos de un determinado tipo:

```
var enemies = {
  basic: { x: 100, y: -50, sprite: 'enemy_purple',
           B: 100, C: 2, E: 100  }
};
```

y asi podemos crear cada enemigo combinando los valores del blueprint con los valores por defecto:

```
var Enemy = function(blueprint,override) {
  var baseParameters =  { A: 0, B: 0, C: 0, D: 0,
                          E: 0, F: 0, G: 0, H: 0 }

  // Se inicializan todos los parámetros a 0
  for (var prop in baseParameters) {
    this[prop] = baseParameters[prop];
  }

  // Se copian los atributos del blueprint
  for (prop in blueprint) {
    this[prop] = blueprint[prop];
  }
  // Se copian los atributos redefinidos, si los hay
```

```
  if(override) {
    for (prop in override) {
      this[prop] = override[prop];
    }
  }
  this.w = SpriteSheet.map[this.sprite].w;
  this.h = SpriteSheet.map[this.sprite].h;
  this.t = 0;
}
```

El atributo t inicializado a 0 marca el tiempo que lleva el objeto en pantalla y es el valor que se utiliza en la ecuación de movimiento.

Y de esta forma podemos añadir un par de enemigos al juego:

```
var playGame = function() {
  var board = new GameBoard();
  board.add(new PlayerShip());
  board.add(new Enemy(enemies.basic));
  board.add(new Enemy(enemies.basic, { x: 200 }));
  Game.setBoard(3,board);
};
```

Movimiento y dibujado

Para el desplazamiento se aplican las ecuaciones de movimiento para obtener el valor de la velocidad y en base a ese valor y el tiempo transcurrido se calcula la nueva posición. La función de dibujado es similar a las que hemos visto antes:

```
Enemy.prototype.step = function(dt) {
  this.t += dt;
  this.vx = this.A + this.B * Math.sin(this.C * this.t + this.D);
  this.vy = this.E + this.F * Math.sin(this.G * this.t + this.H);
  this.x += this.vx * dt;
  this.y += this.vy * dt;
  if(this.y > Game.height ||
     this.x < -this.w ||
     this.x > Game.width) {
       this.board.remove(this);
  }
}

Enemy.prototype.draw = function(ctx) {
  SpriteSheet.draw(ctx,this.sprite,this.x,this.y);
}
```

Refactorización de los distintos tipos de sprites

Según la "regla de las 3 veces" de la refactorización, cuando repetimos el mismo código tres veces ha llegado el momento de refactorizar el código para no ir dejando "deudas técnicas". Las entidades que representan al jugador, los misiles y los enemigos tienen parecido el código de la creación y el dibujado. Creamos un nuevo objeto `Sprite` que servirá de prototipo a los tres tipos de entidades:

```
var Sprite = function() { }

Sprite.prototype.setup = function(sprite,props) {
  this.sprite = sprite;
  this.merge(props);
  this.frame = this.frame || 0;
  this.w =  SpriteSheet.map[sprite].w;
  this.h =  SpriteSheet.map[sprite].h;
}

Sprite.prototype.merge = function(props) {
  if(props) {
    for (var prop in props) {
      this[prop] = props[prop];
    }
  }
}
Sprite.prototype.draw = function(ctx) {
  SpriteSheet.draw(ctx,this.sprite,this.x,this.y,this.frame);
}
```

y se introducen algunos cambios (refactorizaciones) en los distintos tipos de entidades:

```
var PlayerShip = function() {
  this.setup('ship', { vx: 0, frame: 1, reloadTime: 0.25, maxVel: 200 });

  ...
}

PlayerShip.prototype = new Sprite();

var PlayerMissile = function(x,y) {
  this.setup('missile',{ vy: -700 });
  this.x = x - this.w/2;
  this.y = y - this.h;
};
```

```
PlayerMissile.prototype = new Sprite();

var Enemy = function(blueprint,override) {
  this.merge(this.baseParameters);
  this.setup(blueprint.sprite,blueprint);
  this.merge(override);
}
Enemy.prototype = new Sprite();
Enemy.prototype.baseParameters = { A: 0, B: 0, C: 0, D: 0,
                                   E: 0, F: 0, G: 0, H: 0,
                                   t: 0 };
```

y se pueden eliminar los métodos `draw` de las 3 entidades que ahora lo heredan de `Sprite`.

Colisiones

En `GameBoard` ya hemos incluido las funciones `overlap` que determina si dos objetos dados colisionan y `collide` que devuelve el primer objeto de un cierto tipo (o tipos) que colisiona con un objeto dado. Para solo evaluar colisiones entre objetos de un cierto tipo, se añaden tipos y se asigna tipo a cada tipo de entidad:

```
var OBJECT_PLAYER = 1,
    OBJECT_PLAYER_PROJECTILE = 2,
    OBJECT_ENEMY = 4,
    OBJECT_ENEMY_PROJECTILE = 8,
    OBJECT_POWERUP = 16;


  PlayerShip.prototype = new Sprite();
  PlayerShip.prototype.type = OBJECT_PLAYER;

  PlayerMissile.prototype = new Sprite();
  PlayerMissile.prototype.type = OBJECT_PLAYER_PROJECTILE;

  Enemy.prototype = new Sprite();
  Enemy.prototype.type = OBJECT_ENEMY;
```

Cuando se produzca una colisión, el comportamiento por defecto es destruir el objeto con el que se colisiona haciéndolo desaparecer del tablero:

```
Sprite.prototype.hit = function(damage) {
  this.board.remove(this);
}
```

Colisiones de las balas con los enemigos

Las balas hacen un cierto daño cuando alcanzan a un enemigo y desaparecen:

```
var PlayerMissile = function(x,y) {
  this.setup('missile',{ vy: -700, damage: 10 });
  this.x = x - this.w/2;
  this.y = y - this.h;
};

PlayerMissile.prototype.step = function(dt)  {
  this.y += this.vy * dt;
  var collision = this.board.collide(this,OBJECT_ENEMY);
  if(collision) {
    collision.hit(this.damage);
    this.board.remove(this);
  } else if(this.y < -this.h) {
      this.board.remove(this);
  }
};
```

Le añadimos un atributo de salud a los enemigos y modificamos la función `hit` heredada de `Sprite`

```
var enemies = {
  basic: { x: 100, y: -50, sprite: 'enemy_purple',
           B: 100, C: 4, E: 100, health: 20 }
};

Enemy.prototype.hit = function(damage) {
  this.health -= damage;
  if(this.health <= 0)
    this.board.remove(this);
}
```

Colisiones de los enemigos con el jugador

Si algún enemigo colisiona con el jugador, desaparece del tablero y envía el mensaje `hit` al objeto jugador:

```
Enemy.prototype.step = function(dt) {
  ...

  var collision = this.board.collide(this,OBJECT_PLAYER);
  if(collision) {
    collision.hit(this.damage);
    this.board.remove(this);
```

```
  }

  ...
}
```

y cuando el jugador es golpeado, se acaba el juego:

```
PlayerShip.prototype.hit = function(damage) {
  if(this.board.remove(this)) {
    loseGame();
  }
}
```

al eliminarlo del tablero, hay que comprobar que no se haya eliminado antes, por eso la función `GameBoard.remove` devuelve un valor booleano para indicar si se ha borrado en esta invocación o ya estaba borrado:

```
var GameBoard = function() {
  ...

  this.remove = function(obj) {
    var idx = this.removed.indexOf(obj);
    if(idx == -1) {
      this.removed.push(obj);
      return true;
    } else {
      return false;
    }
  };

  ...
}
```

Explosiones

Añadimos un nuevo tipo de sprite que se encargará de visualizar una explosión cada vez que explote un enemigo:

```
var sprites = {
  explosion: { sx: 0, sy: 64, w: 64, h: 64, frames: 12 },
  ...
}
```

Los fotogramas que componen el sprite de la explosión se muestran en la figura 9

recuérdese que el método `draw` heredado de `Sprite` tiene en cuenta el fotograma:

```
Sprite.prototype.draw = function(ctx) {
  SpriteSheet.draw(ctx,this.sprite,this.x,this.y,this.frame);
```

Figure 9: Explosión

```
}
```

y una explosión deberá ir incrementando el contador de fotograma con cada vuelta del bucle principal (en realidad, cada 3 vueltas para que la explosión dure un poco más y se vea mejor):

```
var Explosion = function(centerX,centerY) {
  this.setup('explosion', { frame: 0 });
  this.x = centerX - this.w/2;
  this.y = centerY - this.h/2;
  this.subFrame = 0;
};

Explosion.prototype = new Sprite();

Explosion.prototype.step = function(dt) {
  this.frame = Math.floor(this.subFrame++ / 3);
  if(this.subFrame >= 36) {
    this.board.remove(this);
  }
};
```

y la explosión se crea cuando se destruye un enemigo:

```
Enemy.prototype.hit = function(damage) {
  this.health -= damage;
  if(this.health <=0) {
    if(this.board.remove(this)) {
      this.board.add(new Explosion(this.x + this.w/2,
                                   this.y + this.h/2));
    }
  }
}
```

Niveles

Representación de los niveles

Lo que falta es añadir el código que se encargue de instanciar los distintos tipos de enemigos y los vaya secuenciando adecuadamente de acuerdo con el reto que el diseñador quiera plantear.

Empezamos definiendo varios tipos de enemigos (en realidad sus `blueprints`):

```
var enemies = {
  straight: { x: 0,   y: -50, sprite: 'enemy_ship', health: 10,
              E: 100 },
  ltr:      { x: 0,   y: -100, sprite: 'enemy_purple', health: 10,
              B: 200, C: 1, E: 200  },
  circle:   { x: 400,   y: -50, sprite: 'enemy_circle', health: 10,
              A: 0,  B: -200, C: 1, E: 20, F: 200, G: 1, H: Math.PI/2 },
  wiggle:   { x: 100, y: -50, sprite: 'enemy_bee', health: 20,
              B: 100, C: 4, E: 100 },
  step:     { x: 0,   y: -50, sprite: 'enemy_circle', health: 10,
              B: 300, C: 1.5, E: 60 }
};
```

Para especificar los datos de instanciación de los enemigos podríamos utilizar un array donde especificásemos para cada enemigo: su tipo, su posición inicial y el instante de tiempo en el que debe crearse. Sin embargo, es menos laborioso (para el diseñador, aunque no así para el programador), especificar cada oleada por: el tipo de enemigos de esa oleada, el instante en que se crea el primero, cuándo se crea el último y cuánto se debe esperar entre la creación de dos consecutivos (lo que indirectamente determina el número de enemigos de la oleada):

```
var level1 = [
 // Start,    End, Gap,  Type,   Override
  [ 0,       4000, 500, 'step' ],
  [ 6000,   13000, 800, 'ltr' ],
  [ 12000,  16000, 400, 'circle' ],
  [ 18200,  20000, 500, 'straight', { x: 150 } ],
  [ 18200,  20000, 500, 'straight', { x: 100 } ],
  [ 18400,  20000, 500, 'straight', { x: 200 } ],
  [ 22000,  25000, 400, 'wiggle', { x: 300 }],
  [ 22000,  25000, 400, 'wiggle', { x: 200 }]
];
```

El objeto `Level`

El objeto que se encarga de instanciar a los enemigos se construye a partir del array con la información de las oleadas y la función a la que hay que llamar cuando ya no haya más enemigos para crear. El objeto `Level` guarda una copia de los datos del nivel en el atributo `levelData`, y como a medida que vaya creando enemigos irá modificando ese atributo, se hace una copia profunda del array recibido como parámetro, para así no modificar la definición global del nivel almacenada el `level1` (en JavaScript todos los parámetros se pasan por referencia):

```
var Level = function(levelData,callback) {
  this.levelData = [];
  for(var i = 0; i < levelData.length; i++) {
    this.levelData.push(Object.create(levelData[i]));
  }
  this.t = 0;
  this.callback = callback;
}
```

Como el objeto `Level` se añade al tablero, tendrá un método `draw`, que no ha de dibujar nada:

```
Level.prototype.draw = function(ctx) { }
```

y un método `step` que será el encargado de ir creando las oleadas y comprobar cuando ya no hay nada más que crear:

```
Level.prototype.step = function(dt) {
  var idx = 0, remove = [], curShip = null;

 // Update the current time offset
  this.t += dt * 1000;

  //  Example levelData
  //   Start, End,  Gap, Type,   Override
  // [[ 0,     4000, 500, 'step', { x: 100 } ]
  while((curShip = this.levelData[idx]) &&
        (curShip[0] < this.t + 2000)) {
    // Check if past the end time
    if(this.t > curShip[1]) {
      // If so, remove the entry
      remove.push(curShip);
    } else if(curShip[0] < this.t) {
      // Get the enemy definition blueprint
      var enemy = enemies[curShip[3]],
          override = curShip[4];

      // Add a new enemy with the blueprint and override
      this.board.add(new Enemy(enemy,override));

      // Increment the start time by the gap
      curShip[0] += curShip[2];
    }
    idx++;
  }
  // Remove any objects from the levelData that have passed
  for(var i = 0, len = remove.length; i < len; i++) {
    var idx = this.levelData.indexOf(remove[i]);
```

```
    if(idx != -1) this.levelData.splice(idx,1);
  }

  // If there are no more enemies on the board or in
  // levelData, this level is done
  if(this.levelData.length == 0 && this.board.cnt[OBJECT_ENEMY] == 0) {
    if(this.callback) this.callback();
  }
}
```

El programa principal

Cada partida viene controlada por estas tres funciones:

```
var playGame = function() {
  var board = new GameBoard();
  board.add(new PlayerShip());
  board.add(new Level(level1,winGame));
  Game.setBoard(3,board);
};

var winGame = function() {
  Game.setBoard(3,new TitleScreen("You win!",
                                  "Press fire to play again",
                                  playGame));
};

var loseGame = function() {
  Game.setBoard(3,new TitleScreen("You lose!",
                                  "Press fire to play again",
                                  playGame));
};
```

recuerda que 'loseGame' se invoca cuando el objeto del jugador es alcanzado por un enemigo:

```
PlayerShip.prototype.hit = function(damage) {
  if(this.board.remove(this)) {
    loseGame();
  }
}
```

Referencias

Referencias

- Pascal Rettig. Professional HTML5 Mobile Game Development
- Douglas Crockford. JavaScript: The good parts. O'Reilly Media, 2008. Los principales detalles se pueden ver en una de las charlas que Douglas Crockford dio en Google
- John Resig. Secrets of the JavaScript Ninja. Manning Publications, 2012
- Professional JavaScript for Web Developers, 3rd Edition
- Frequently Misunderstood JavaScript Concepts