

# INSTITUTO TECNOLOGICO DE CHIHUAHUA II



GRAFICACIÓN 9-10 AM  
PROJECT: SPACESHIP GAME

Author: Edith Ortiz Martinez #13550419

Adelaido Cano Lozano #14550322

Teacher: Alonso Salcido

Fecha: 07/10/17

## Introducción

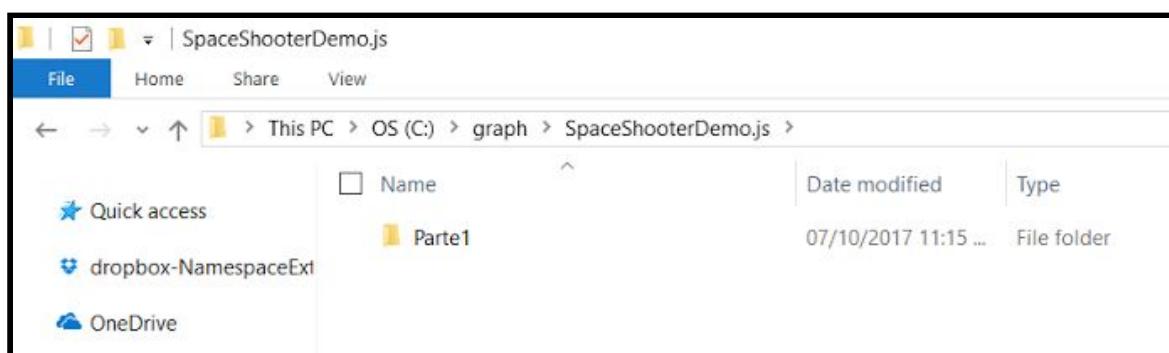
Este proyecto se basa en la creación de un juego en HTML5, llamado SpaceShip Demo, similar al juego de arcade Galaxian 1979 paso a paso.

Es básicamente un tutorial, el cual está dividido en 5 pasos, cada uno cubre una parte o aspecto del juego. Cubrirá todo, desde la configuración del juego hasta la optimización web en la codificación.

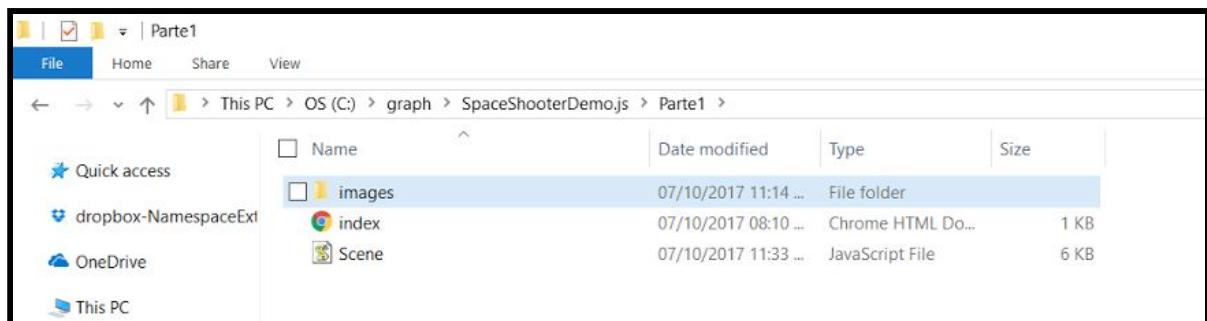
A continuación se explica el paso a paso para la realización del proyecto.

Primer paso:

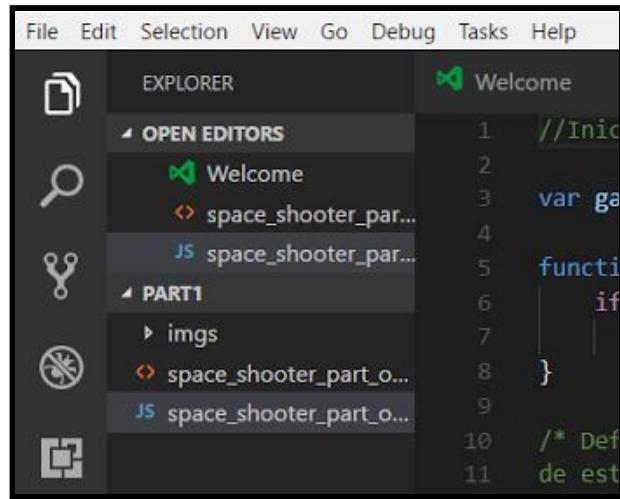
Empezamos creando nuestra carpeta para nuestra primera parte del proyecto:



Dentro de nuestra carpeta crearemos una segunda carpeta para guardar nuestra imagen con extensión .png, la cual utilizaremos para el fondo de nuestro canvas.



Además, enseguida se crean desde nuestro Visual Studio Code, un nuevo archivo JS llamado Scene, en el cual se utilizará sintaxis de JavaScript puro, para la realización del fondo de nuestro juego, además de nuestro index para la creación de nuestra webpage en implementando HTML5.



Empezamos con el código para la creación de nuestra webpage en HTML5 en nuestro index, el cual se vería de esta manera:

Se crea un canvas e incluimos algún texto dentro de él en caso de que el canvas no esté soportado en un navegador.

Por último, cargamos el recurso JavaScript.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Space Shooter Demo</title>
        <style>
            canvas {
                position: absolute; /*se toma como referencia la ventana del navegador*/
                top: 0px;
                left: 0px;
                background: transparent;
            }
        </style>
    </head>
    <body onload="init()">
        <!-- Este es el canvas para el fondo de nuestro juego -->
        <canvas id="background" width="600" height="360"> <!--se establece el tamaño del canvas-->
            Tu navegador no soporta el canvas, Por favor intenta con otro navegador no tan viejo.
        </canvas>
        <script src="space_shooter_part_one.js"></script>
    </body>
</html>
```

Ahora continuamos con nuestro código en JS:

```

1  /* Define un objeto para mantener todas nuestras imágenes en el juego,
2   de esta manera las imágenes solo son creadas una sola vez.
3   Este tipo de objeto es conocido como singleton.
4   singleton (instancia única): Garantiza que una clase tenga una sola instancia
5   y proporciona un punto de acceso global a ella.*/
6   var imageRepository = new function() {
7     //Definimos imágenes
8     this.empty = null;
9     this.background = new Image();
10
11    //Establecemos la fuente de la imagen
12    this.background.src = "imgs/is.png";
13  }

```

Dado que sólo nos centramos en el panorámico del fondo, sólo creamos la imagen de fondo y establecemos su atributo src. Ahora podemos hacer referencia al objeto del repositorio de imágenes para cualquier imagen que necesitamos.

El siguiente objeto que crearemos es el objeto Drawable.

```

1
2
3  /*Creamos el objeto para dibujar el cual sera la clase base para
4   todos los objetos dibujables en el juego. Se definen variables por
5   default que todos los objetos hijo van a heredar, asi como las
6   funciones por defecto*/
7  function Drawable() {//Este objeto es un objeto abstracto, el cual todos los objetos del juego lo van a heredar
8    this.init = function(x, y) {
9      //Variables por default
10     this.x = x;
11     this.y = y;
12   }
13
14   this.speed = 0;
15   this.canvasWidth = 0;
16   this.canvasHeight = 0;
17
18   //Se definen la función abstracta a implementar en los objetos hijo
19   this.draw = function() {
20     };
21   }
22 }

```

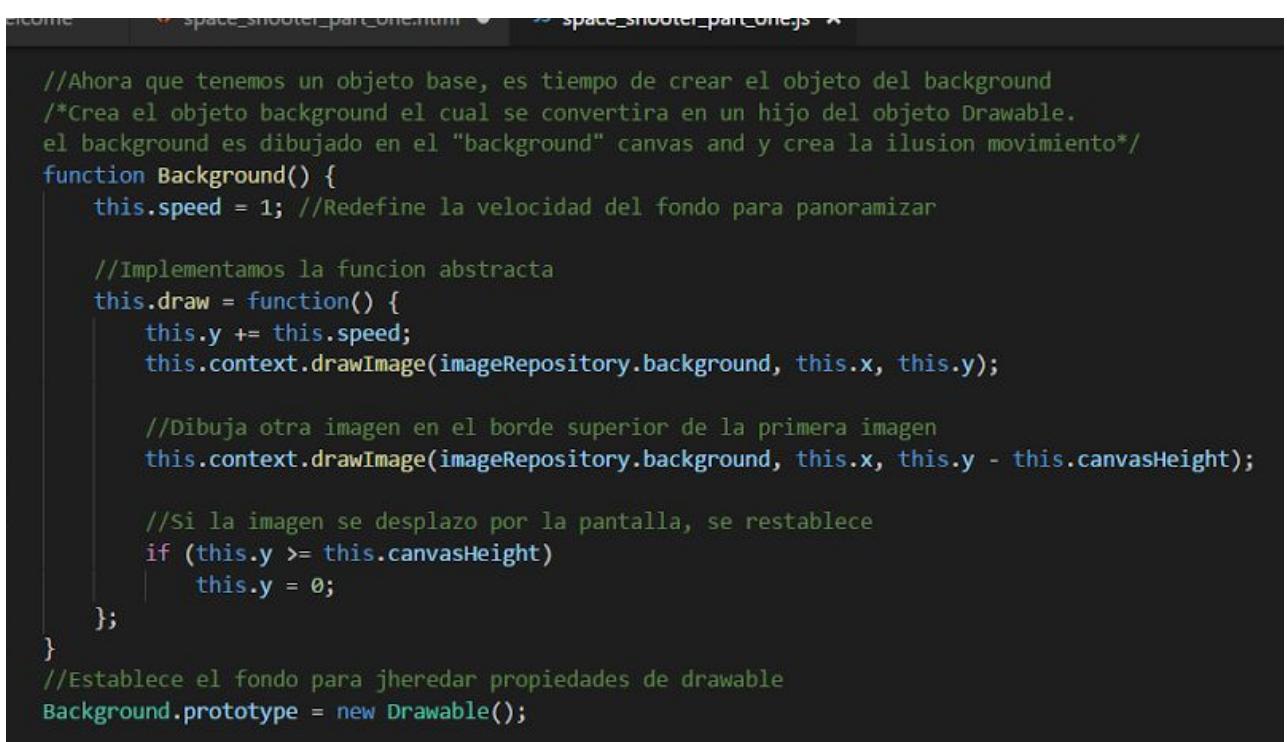
El objeto Dibujable es un objeto especial del que heredará todos los demás objetos de nuestro juego.

un objeto abstracto nos permite tener un objeto que define todas las variables y funciones que necesitamos para usarlo,luego da las mismas variables y funciones a todos los objetos que heredan de él, entonces no tenemos que duplicar ningun código.

Entonces, si alguna vez tenemos que hacer cambios en la función de un objeto, solo tenemos que cambiarlo en el objeto drawable y no a cada uno de ellos.

El objeto drawable tiene un método llamado init el cual nos permite poner a la X y la Y la posición del objeto cuando es creado. También define la velocidad del objeto, y la altura y el ancho del canvas que contiene el objeto.

El ultimo metodo "last", no necesariamente necesita ser definida ya que está vacío.



```
//Ahora que tenemos un objeto base, es tiempo de crear el objeto del background
/*Crea el objeto background el cual se convertira en un hijo del objeto Drawable.
el background es dibujado en el "background" canvas and y crea la ilusion movimiento*/
function Background() {
    this.speed = 1; //Redefine la velocidad del fondo para panoramizar

    //Implementamos la funcion abstracta
    this.draw = function() {
        this.y += this.speed;
        this.context.drawImage(imageRepository.background, this.x, this.y);

        //Dibuja otra imagen en el borde superior de la primera imagen
        this.context.drawImage(imageRepository.background, this.x, this.y - this.canvasHeight);

        //Si la imagen se desplazo por la pantalla, se restablece
        if (this.y >= this.canvasHeight)
            this.y = 0;
    };
}
//Establece el fondo para jheredar propiedades de drawable
Background.prototype = new Drawable();
```

El objeto background establece la velocidad del fondo panorámico a un pixel por fotograma, luego define el método draw en el método draw, actualizamos la posición Y del objeto(desde que nuestro fondo de panorama a la imagen de arriba a abajo) luego lo dibuja en el canvas.

Luego dibujamos la misma imagen para crear la ilusión de un fondo infinito. Por último, el método comprueba si la posición y de la imagen ha desaparecido de la pantalla y lo restablecerá si tiene que continuar con el panorámico. Para configurar el objeto Background a heredar del objeto Drawable, usamos la funcionalidad prototípica.

Prototípico es un poco confuso al principio, pero no es demasiado difícil de conseguir. Básicamente, estamos diciendo al objeto de fondo copiar toda la información del objeto Drawable.

Así es como podemos implementar la herencia en JavaScript.

```

/*con la estructura basica del juego completo, es hora de crear el objeto final
que manejará el juego entero*/
//se crea el objeto Game, el cual tomara todos los objetos y datos para el juego.
function Game() {
    /*obtiene la informacion del canvas y context y actualiza todos los objetos del juego
    -regresa true si el canvas es soportado y falso si no, esto para parar
    la animacion de correr constantemente en navegadores mas viejos*/
    this.init = function() {
        //obtiene el elemento canvas
        this.bgCanvas = document.getElementById('background');

        //Testeo para saber si el canvas es soportado
        if (this.bgCanvas.getContext) {
            this.bgContext = this.bgCanvas.getContext('2d');

            //inicializa los objetos para contener la informacion de su context y canvas

            Background.prototype.context = this.bgContext;
            Background.prototype.canvasWidth = this.bgCanvas.width;
            Background.prototype.canvasHeight = this.bgCanvas.height;

            //inicializa el objeto background
            this.background = new Background();
            this.background.init(0,0); //Establece el punto de inicio de dibujo como (0,0)
            return true;
        } else {
            return false;
        }
    };

    //comienza el bucle de animacion
    this.start = function() {
        animate();
    };
}

```

El objeto Game tiene sólo dos métodos. La función init captura primero todos los elementos de la página web. A continuación, comprueba si el lienzo se admite mirando la función canvas.getContext. La función devuelve true si el navegador actual admite el lienzo y devuelve false si no lo hace.

Si se admite el lienzo, el juego establecerá información sobre el lienzo en el objeto de fondo. De esta manera, el objeto de fondo sabe qué lienzo utilizar y sus dimensiones. A continuación, crea un objeto de fondo y le da su posición inicial. Por último, el juego devuelve verdadero, lo que significa que el lienzo es de hecho apoyado y que el juego puede continuar.

El método start sólo inicia el bucle de animación y se llama después de que la función init devuelve true. Si el lienzo no está soportado en navegadores antiguos, este método nunca se llama para que los recursos innecesarios del sistema no se utilicen en una función que no puede hacer nada.

Ahora, por último, con el objeto Game terminado, lo único que queda por hacer es crear el bucle de animación.

```
/* Con el objeto Game terminado, lo único que queda por hacer es crear el bucle de animación.*/
/* el bucle de animacion. llama a requestAnimationFrame para optimizar
el bucle del juego y dibuja todos los objetos del juego. esta funcion tiene que ser
una funcion global y no puede estar sin un objeto*/
function animate() {
    requestAnimationFrame( animate );
    /* informa al navegador que quieres realizar una animación y solicita que el navegador programe el repintado de la ventana para el próximo
    ciclo de animación. El método acepta como argumento una función a la que llamar antes de efectuar el repintado
    Debes llamar a este método cuando estés preparado para actualizar tu animación en la pantalla para pedir que se programe el repintado.
    Esto puede suceder hasta 60 veces por segundo en pestañas en primer plano, pero se puede ver reducido a velocidades inferiores en
    pestañas en segundo plano
    */
    game.background.draw();
}

/*Finds the first API that works to optimize the animation loop,
otherwise defaults to setTimeout()
*/
window.requestAnimationFrame = (function(){
    return window.requestAnimationFrame ||
    window.webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame ||
    window.oRequestAnimationFrame ||
    window.msRequestAnimationFrame ||
    function( callback, element){
        window.setTimeout(callback, 1000 / 60);
    };
})();
//La pieza final del código es crear el objeto juego y correrlo
```

La función animate es el bucle de juego y todo lo que hace es dibujar el objeto de fondo.

Para animar el juego, la función animada llama el calce requestAnimationFrame creado por Paul Irish. No se debe crear un bucle de juego con window.setTimeout, ya que no está optimizado para 60FPS.

Los navegadores modernos han creado sus propias funciones de bucle de juego altamente optimizadas, y este shim encuentra el primero que funciona y lo usa.

Por defecto aparecerá window.setTimeout si nada funciona.

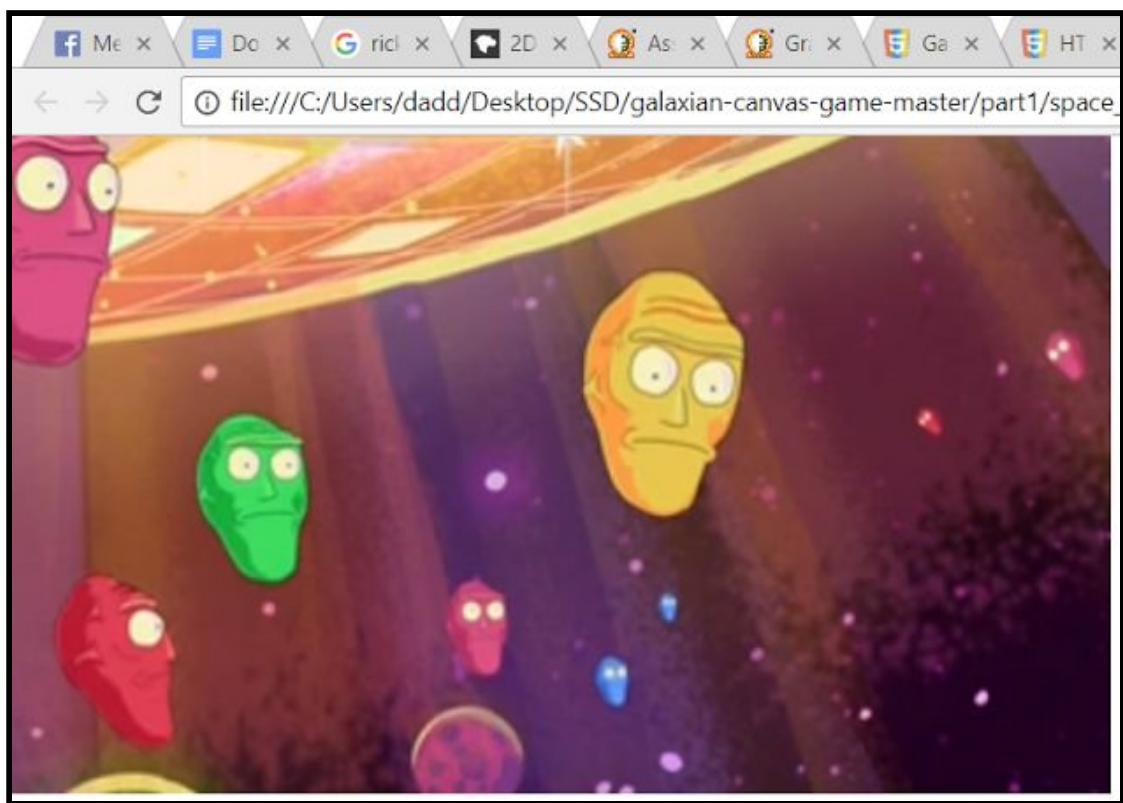
La última pieza de código es crear el objeto de juego y ejecutarlo:



The screenshot shows a code editor interface with the following details:

- OPEN EDITORS:** 2 UNSAVED
- Files:** Welcome, space\_shooter\_part\_01.js, space\_shooter\_part\_02.js
- PART1:** imgs, space\_shooter\_part\_01.js
- Code Preview:** The code shown is:

```
//Inicializar el juego y comenzar
var game = new Game();
function init() {
    if(game.init())
        game.start();
}
```



Este sería el background de nuestro juego final, y ahora empezamos con el segundo paso de la creación de nuestro juego.

Segundo paso:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Space Shooter Demo</title>
    <style>
      canvas {
        position: absolute; /*se toma como referencia la ventana del navegador*/
        top: 0px;
        left: 0px;
        background: transparent;
      }
      #background{
        z-index:-2;
      }
      #main{
        z-index:-1;
      }
      #ship{
        z-index:0;
      }
    </style>
  </head>
  <body>
    <!-- Este es el canvas para el fondo de nuestro juego -->
    <canvas id="background" width="600" height="360"> <!--se establece el tamaño del canvas-->
      Tu navegador no soporta el canvas, Por favor intenta con otro navegador no tan viejo.
    </canvas>
    <!-- el canvas para todos los enemigos y balas-->
    <canvas id="main" width="600" height="360">
    </canvas>
    <!-- el canvas que la nave usa (solo puede moverse un cuarto de la pantalla hacia arriba)-->
    <canvas id="ship" width="600" height="360">
    </canvas>
    <script src="space_shooter_part_one.js"></script>
  </body>
</html>
```

Algunas cosas han cambiado, se agregaron dos canvas más a la página y los styles para apilarlos en su orden correcto en la pantalla. Se utilizarán 3 canvas para nuestro juego, ya que nos permite no tener que redibujar el juego completo de cada frame.

Un canvas será para el fondo de desplazamiento. Otro canvas se utilizará para la nave del jugador. El último canvas se utilizará para las balas que mueven cada fotograma una vez que están en la pantalla.

También eliminamos el evento onload = "init ()" de la etiqueta body.

Ahora actualizamos la función init () del objeto Drawable:

```
/*Creamos el objeto para dibujar el cual sera la clase base para
todos los objetos dibujables en el juego. Se definen variables por
default que todos los objetos hijo van a heredar, asi como las
funciones por defecto*/
function Drawable() {//Este objeto es un objeto abstracto, el cual todos los objetos del juego lo van a heredar
    this.init = function(x, y, width, height) {
        //Variables por default
        this.x = x;
        this.y = y;
        this.width = width; //como vamos a utilizar imagenes que no seran |
        this.height = height;//del tamaño del canvas, tenemos que definir alto y ancho de esas imagenes
    }
}
```

Puesto que estamos añadiendo imágenes que ya no llenan el canvas completo, necesitamos establecer la altura y el ancho de cada objeto a su imagen asociada. Esto nos ayudará más adelante mientras intentamos dibujar y luego borrar cada objeto de la pantalla.

Con el objeto Drawable actualizado, actualizaremos el objeto image Repository:

```
0  /* Define un objeto para mantener todas nuestras imagenes en el juego,
1   de esta manera las imagenes solo son creadas una sola vez.
2   Este tipo de objeto es conocido como singleton.
3   singleton (instancia unica): Garantiza que una clase tenga una sola instancia
4   y proporciona un punto de acceso global a ella.*/
5  var imageRepository = new function() {
6      //Definimos imagenes
7      //this.empty = null;
8      this.background = new Image();
9      this.spaceship = new Image();
10     this.bullet = new Image();
11     //Asegurarse de que todas las imagenes se han cargado antes de empezar el juego
12     var numImages = 3; //tenemos 3 imagenes ahora
13     var numLoaded = 0;
14     function imageLoaded(){ //funcion para las imagenes cargadas
15         numLoaded++;
16         if(numLoaded === numImages){//el numero de imagenes cargadas|
17             window.init();
18         }
19     }
20     this.background.onload = function(){ //fondo
21         imageLoaded();
22     }
23     this.spaceship.onload = function(){ //nave
24         imageLoaded();
25     }
26     this.bullet.onload = function(){ //balas
27         imageLoaded();
28     }
29     //Establecer las imagenes src
30     //Establecemos la fuente de la imagen
31     this.background.src = "imgs/is.png";
32     this.spaceship.src = "imgs/nave.png";
33     this.bullet.src = "imgs/bllt.png";
34 }
```

Primero cargamos las dos nuevas imágenes usadas para la nave y las balas que dispara la nave, y luego establecemos sus atributos src. El cambio principal es la adición de la función imageLoaded () .

La función imageLoaded () se usa para determinar cuándo todos los activos del juego (en este caso imágenes) se han cargado correctamente. Una vez que todos los activos de los juegos se cargan, se iniciará el juego. Hay dos razones por las que añadimos esta función.

La primera razón es que puede utilizar esta función para ayudar a producir una pantalla de carga con una barra de carga. Cuando cada nuevo recurso llama a esta función, puede avanzar la barra de carga por el porcentaje de numLoaded / numImages.

La segunda (y primaria) razón es para arreglar un error que se encontró. Cuando el juego ejecutaría la función game.init (), las imágenes no estarían completamente cargadas. Esto causaría la llamada a imageRepository.ship.width para devolver 0 ya que no había ninguna imagen. Este tipo de error se conoce como condición de carrera. La carrera se realizó entre la carga de la imagen y la llamada a imageRepository cuando se necesitaba la imagen.

Al implementar la función imageLoaded (), podemos asegurar que todas las imágenes se cargan antes de que se usen, negando la condición de carrera del código original. Es por esta razón que el evento onload = "init ()" se sacó de la etiqueta <body> (porque el evento onload sólo tiene en cuenta los activos cargados en el HTML, no a través de JavaScript) y se trasladó a la imagenLoaded () función.

A continuación, escribiremos el código para manejar un jugador moviéndose y disparando. Para comenzar, presentaremos una nueva estructura de datos llamada conjunto de objetos.

"Una agrupación de objetos es una estructura de datos que reutiliza objetos antiguos para no crear o eliminar continuamente otros nuevos". Cuando se crean o eliminan muchos objetos en sucesión rápida (como nuestras balas), puede causar que el juego se retrase intenta liberar la memoria utilizada por esos objetos. Estaremos usando un pool de objetos para ayudar a manejar todas las balas que el jugador puede disparar para que el juego no se quede retrasado por tener que crear y eliminar cada bala.

```

function Pool(maxSize){
    var size = maxSize; //Max balas permitidas en el pool
    var pool = []; //se guardan en un arreglo

    this.init = function(){
        for(var i=0; i<size; i++){
            //inicializamos el objeto bala
            var bullet = new bullet();
            bullet.init(0,0, imageRepository.bullet.width,
                imageRepository.bullet.height);
            pool[i]= bullet;
        }
    };

    //toma el ultimo articulo en la lista y lo inicializa
    //y luego lo pone al principio del arreglo
    this.get = function(x, y, speed){
        if(!pool[size - 1].alive){
            pool[size - 1].spawn(x, y, speed);
            pool.unshift(pool.pop());
        }
    };

    /*Usado para que la nave sea capaz de lanzar dos balas a la vez
    solo si la funcion get() es usada dos veces, la nave es capaz de disparar
    y solo lanza una bala en vez de 2*/
    this.getTwo = function(x1, y1, speed1, x2, y2, speed2){
        if(!pool[size - 1].alive &&
            pool[size - 2].alive){
            this.get(x1, y1, speed1);
            this.get(x2, y2, speed2);

            this.get(x1, y1, speed1),
            this.get(x2, y2, speed2);
        }
    };

    /*Dibuja cualquier bala en uso, si alguna bala se sale de la pantalla
    la limpia y la manda al principio del arreglo */
    this.animate = function(){
        for(var i=0; i<size; i++){
            //solo dibuja hasta que encontramos una bala que no esta viva
            if(pool[i].alive){
                if(pool[i].draw()){
                    pool[i].clear();
                    pool.push((pool.splice(i,1))[0]);
                }
            }else
                break;
        }
    };
}

```

Cuando se inicializa el pool (agrupación), rellena un arreglo con objetos Bullet vacíos que se volverán a utilizar cada vez que se cree una bala. Cuando el pool necesita crear una nueva bala para su uso, mira el último elemento del arreglo y comprueba si está actualmente en uso. Si está en uso, el pool está lleno y no se pueden generar balas. Si no está en uso, el pool genera el último elemento del arreglo y, a continuación, se despliega desde la parte posterior y lo empuja hacia el frente del arreglo. Esto hace que el pool tenga balas en la espalda y balas usadas en la parte delantera.

Cuando el pool anima las balas, comprueba para ver si la bala está en uso (no hay necesidad de dibujar balas no utilizadas) y si lo es, lo dibuja. Si la función draw () devuelve true, la bala está lista para ser reutilizada y el pool borra la bala y la elimina el arreglo y la empuja hacia la parte posterior. Hacer esto hace que crear / destruir balas en el pool tenga tiempo constante.

Con el pool de objetos en la mano, ahora podemos crear el objeto bullet:

```
function Bullet(){
    this.alive = false; //es verdadero si la bala esta actualmente en uso
    //da los valores a la bala
    this.spawn = function(x, y, speed){
        this.x = x;
        this.y = y;
        this.speed = speed;
        this.alive = true;
    };
    /*utiliza un rectangulo para borrar la bala y la mueve
    regresa true si la bala se movio fuera de la pantalla, indicando que
    la bala esta lista para ser limpiada por el pool, de otra manera
    dibuja la bala */
    this.draw = function(){
        this.context.clearRect(this.x, this.y, this.width, this.height);
        this.y -= this.speed;
        if(this.y <= 0 - this.height){
            return true;
        }else{
            this.context.drawImage(imageRepository.bullet, this.x, this.y);
        }
    } ;
    //reestablece los valores de la bala
    this.clear = function(){
        this.x = 0;
        this.y = 0;
        this.speed = 0;
        this.alive = false;
    };
}
Bullet.prototype = new Drawable();
```

El objeto Bullet establece el estado inicial de la bala, estableciendo alive como false. Luego define tres métodos: spawn, draw y clear. El método spawn se utiliza para crear una nueva bala y toma un valor x y y, así como la velocidad de la bala, estableciendo variables en consecuencia.

### Rectángulos sucios

En el método de dibujo usamos una técnica conocida como un rectángulo sucio para borrar sólo el área inmediata alrededor de la bala. Si la bala se sale de la pantalla, el método devuelve verdadero, lo que significa que está listo para ser reutilizado, de lo contrario lo vuelve a dibujar.

Los rectángulos sucios son un gran aumento de rendimiento, ya que la limpieza de todo el canvas es caro, mientras que el borrado de una pequeña sección del canvas es barato. Así, cada vez que dibujemos objetos en la pantalla, usaremos rectángulos sucios para borrarlos del canvas antes de volverlos a dibujar, y sólo volveremos a dibujar los objetos que se han movido desde el último fotograma.

Cuando se trabaja con animación en canvas, el rendimiento puede ser un desafío, ya que las operaciones de mapa de bits son muy costosas, especialmente en altas resoluciones. Una regla de optimización importante a seguir es reutilizar tantos píxeles como sea posible entre frames. Un buen ejemplo de esto es cuando se borran los píxeles con el método clearRect (x, y, w, h), es muy beneficioso borrar y volver a dibujar sólo los píxeles que han cambiado y no el canvas completo. A diferencia de las regiones de dibujo de Flash Player, esta gestión de "rectángulos sucios" debe realizarse manualmente para el canvas. - Hakim El Hattab

Los rectángulos sucios son la razón por la que dividimos el juego en tres canvas. Si sólo tuviéramos un canvas, tendríamos que pintarlo en orden de abajo a arriba. El fondo se dibujaría primero, luego la nave y las balas. Hacerlo de esta manera nos llevaría a tener que redibujar todo lo que es cada cuadro, lo cual es caro.

Si sólo tuviéramos dos canvas, el fondo volvería a dibujar cada trama, pero la nave podría ser dibujada sólo cuando se haya movido. Las balas se volverían a dibujar cada cuadro usando un rectángulo sucio, pero si la bala se superpone a la nave, parte la nave también se borraría. Si el jugador no se había movido desde el último cuadro, la nave no se volvería a dibujar por lo que la sección borrada permanecería hasta que el jugador se moviera. No es el mejor efecto visual.

Así que usamos tres canvases para que podamos ensuciar las balas sin tener que limpiar la nave si la bala se superpone. El rendimiento se mantiene y visualmente nada parece torcido.

El último método del objeto de bala es el método clear. Este método simplemente restablece todas las variables establecidas por el método spawn para que el objeto pueda ser reutilizado.

El último objeto por crear es el objeto nave:

```
function Ship(){
    this.speed = 3;
    this.bulletPool = new Pool(30);
    this.bulletPool.init();
    var fireRate = 15;
    var counter = 0;
    this.draw = function(){
        this.context.drawImage(imageRepository.spaceship, this.x, this.y);
    };
    this.move = function(){
        counter++;
        //determina si la accion es mover
        if(KEY_STATUS.left || KEY_STATUS.right ||
        KEY_STATUS.down || KEY_STATUS.up){
            //la nave se movio, entonces se borra la imagen actual para que pueda
            //ser redibujada en su nueva locacion
            this.context.clearRect(this.x, this.y, this.width, this.height);
            //actualiza x y y de acuerdo con la direccion de movimiento
            //y redibuja la nave, cambia the else if's to id=f statements
            //para tener movimientos diagonales
            if(KEY_STATUS.left){
                this.x -= this.speed
                if(this.x <=0)//continua jugando
                this.x = 0;

            }else if (KEY_STATUS.right){
                this.x += this.speed
                if(this.x >= this.canvasWidth - this.width)
                this.x = this.canvasWidth - this.width;
            }else if (KEY_STATUS.up){
                this.y -= this.speed
                if(this.y <= this.canvasHeight/4*3)
                this.y = this.canvasHeight/4*3;
            }else if(KEY_STATUS.down){
                this.y += this.speed
                if(this.y >= this.canvasHeight - this.height)
                this.y = this.canvasHeight - this.height;
            }
        }
    }
}
```

```

        }else if(KEY_STATUS.down){
            this.y += this.speedif(this.y >= this.canvasHeight - this.height)
            this.y = this.canvasHeight - this.height;
        }
        //finalizando el redibujado de la nave
        this.draw();
    }
    if(KEY_STATUS.space && counter >= fireRate){
        this.fire();
        counter = 0;
    }
};

//dispara 2 balas
this.fire = function(){
    this.bulletPool.getTwo(this.x+6, this.y, 3, this.x+33, this.y, 3);
};

Ship.prototype = new Drawable();

```

El objeto Ship fija la velocidad de la nave a 3 píxeles por trama y crea el objeto pool para las balas de las naves. A continuación, establece la tasa de fuego de las naves para poder disparar una vez cada 15 fotogramas, utilizando el contador para determinar cuándo puede disparar. La nave también tiene tres métodos: dibujar, mover y disparar.

El método de dibujo sólo dibuja la nave a la pantalla. El método de movimiento primero incrementa la variable del contador, luego determina si la nave se ha movido basándose en la entrada del teclado (que se describe a continuación). Basado en la entrada del teclado, mueve la nave y luego se asegura de que no salga de la pantalla. La nave en nuestro juego sólo será capaz de moverse 1/4 del camino hasta la pantalla, por lo que se utilizará como la frontera más arriba.

Lo último que hace el método de draw es determinar si el jugador está disparando usando la entrada del teclado y puede disparar basado en la tasa de fuego. Si la nave puede disparar, llama al método fire () y restablece el contador a 0.

El método de fuego llama al método getTwo () del objeto pool para obtener dos balas a la vez y las coloca justo encima de las dos armas de la nave, estableciendo su velocidad en 3 píxeles por fotograma.

Para obtener el teclado de entrada para el juego, vamos a utilizar una técnica desarrollada por Doug McInnes cuando creó asteroides para el canvas.

```

KEY_CODES = [
    32: 'space',
    37: 'left',
    38: 'up',
    39: 'right',
    40: 'down',
]

//se crea el arreglo para los KEY_CODES y se le asignan sus valores
//chemando true/false es la manera mas rapida de checar el
//status de una tecla presionada y cual fue presionada y en que direccion

KEY_STATUS = {};
for(code in KEY_CODES){
    KEY_STATUS[KEY_CODES[code]]= false;
}
/* configura el documento para escuchar eventos onekeydown(cuando
   alguna tecla este presionada) cuando una tecla es presionada, le da
   la direccion apropiada para saber que tecla fue*/

document.onekeydown = function(e){
    var keyCode = (e.keyCode) ? e.keyCode : e.charCode;
    if(KEY_CODES[keyCode]){
        e.preventDefault();
        KEY_STATUS[KEY_CODES[keyCode]] = true;
    }
}

/* configura el documento para ecuchar eventos onekeyup
   (cuando una tecla es liberada) cuando una tecla es liberada, asigna
   la direccion apropiada a falso para darnos a saber cual tecla fue*/

document.onekeyup = function(e){
    var keyCode = (e.keyCode) ? e.keyCode : e.charCode;
    if(KEY_CODES[keyCode]){
        e.preventDefault();
        KEY_STATUS[KEY_CODES[keyCode]]=false;
    }
}

```

El código primero crea un objeto JSON que contiene los códigos de teclado para cada clave del teclado y lo que significa para nuestro juego (ex. left arrow = left). También crea un arreglo de estados de verdadero / falso para que sepamos cuándo se pulsa una tecla. A continuación, crea los eventos onkeydown y onkeyup para activar el cambio de estado de clave de una clave.

Lo último que tenemos que hacer es actualizar el objeto de juego y la función de animación:

```
function Game() {
    /*obtiene la informacion del canvas y context y actualiza todos los objetos del juego
    -regresa true si el canvas es soportado y falso si no, esto para parar
    la animacion de correr constantemente en navegadores mas viejos*/
    this.init = function() {
        //obtiene el elemento canvas
        this.bgCanvas = document.getElementById('background');
        this.shipCanvas = document.getElementById('ship');
        this.mainCanvas = document.getElementById('main');

        //Testeo para saber si el canvas es soportado
        if (this.bgCanvas.getContext) {
            this.bgContext = this.bgCanvas.getContext('2d');
            this.shipContext = this.shipCanvas.getContext('2d');
            this.mainContext = this.mainCanvas.getContext('2d');

            //inicializa los objetos para contener la informacion de su context y canvas

            Background.prototype.context = this.bgContext;
            Background.prototype.canvasWidth = this.bgCanvas.width;
            Background.prototype.canvasHeight = this.bgCanvas.height;
            Ship.prototype.context = this.shipContext;
            Ship.prototype.canvasWidth = this.shipCanvas.width;
            Ship.prototype.canvasHeight = this.shipCanvas.height;
            Bullet.prototype.context = this.mainContext;
            Bullet.prototype.canvasWidth = this.mainCanvas.width;
            Bullet.prototype.canvasHeight = this.mainCanvas.height;

            //inicializa el objeto background
            this.background = new Background();
            this.background.init(0,0); //Establece el punto de inicio de dibujo como (0,0)
            //Iniciar el objeto ship(nave)
            this.ship = new Ship();
            // establecer la nave para empezar cerca del medio del canvas
            var shipStarX = this.shipCanvas.width/2 - imageRepository.spaceship.width;
            var shipStarY = this.shipCanvas.height/4*3 + imageRepository.spaceship.height*2;
            this.ship.init(shipStarX, shipStarY, imageRepository.spaceship.width,
                imageRepository.spaceship.height);
        }
    }
}
```

```

        var shipStarX = this.shipPosition.x + this.shipWidth/4 > imageRepository.spaceship.height/2,
            this.ship.init(shipStarX, shipStartY, imageRepository.spaceship.width,
                           imageRepository.spaceship.height);
        return true;
    } else {
        return false;
    }
};

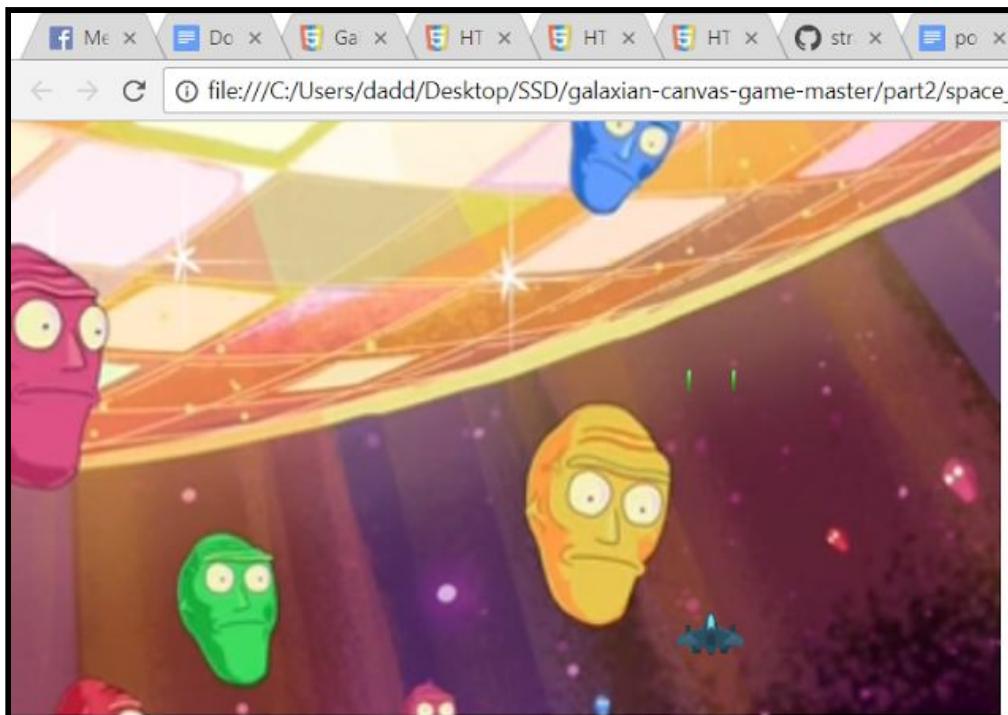
//comienza el bucle de animacion
this.start = function() {
    this.ship.draw();
    animate();
};
}

/* Con el objeto Game terminado, lo único que queda por hacer es crear el bucle de animación.*/
/* el bucle de animacion. llama a requestAnimationFrame para optimizar
el bucle del juego y dibuja todos los objetos del juego. esta funcion tiene que ser
una funcion global y no puede estar sin un objeto*/
function animate() {
    requestAnimationFrame( animate );
    /* informa al navegador que quieres realizar una animación y solicita que el navegador programe el repintado de la ventana para el próximo
    ciclo de animación. El método acepta como argumento una función a la que llamar antes de efectuar el repintado
    Debes llamar a este método cuando estés preparado para actualizar tu animación en la pantalla para pedir que se programe el repintado.
    Esto puede suceder hasta 60 veces por segundo en pestañas en primer plano, pero se puede ver reducido a velocidades inferiores en
    pestañas en segundo plano
    */
    game.background.draw();
    game.ship.move();
    game.ship.bulletPool.animate();
}
}

```

A continuación, inicializa el fondo y el objeto ship(nave), colocándolo cerca del centro inferior del canvas para iniciar.

La función animate sólo tiene que llamar a la función move () y la función animate () del conjunto de objetos para que se animen.



### Paso 3:

En este paso, no hay nada que agregar a nuestro código HTML, así que continuamos implementando nuestras naves enemigas de esta manera:

```
var imageRepository = new function() {
    // Define images
    //this.empty = null;
    this.background = new Image();
    this.spaceship = new Image();
    this.bullet = new Image();
    this.enemy = new Image(); //Se agregan las imágenes para el enemigo y las balas enemigas
    this.enemyBullet = new Image();

    //Asegurarse de que todas las imágenes se han cargado antes de empezar el juego
    var numImages = 5; //tenemos 5 imágenes ahora
    var numLoaded = 0;
    function imageLoaded() {//funcion para las imágenes cargadas
        numLoaded++;
        if (numLoaded === numImages) {
            window.init();
        }
    }
    this.background.onload = function() {//fondo
        imageLoaded();
    }
    this.spaceship.onload = function() {//nave
        imageLoaded();
    }
    this.bullet.onload = function() {//balas
        imageLoaded();
    }
    this.enemy.onload = function() {//nave enemiga
        imageLoaded();
    }
    this.enemyBullet.onload = function() {//balas enemigas
        imageLoaded();
    }

    //Establecer las imágenes src
    //Establecemos la fuente de la imagen
    this.background.src = "imgs/bg.png";
    this.spaceship.src = "imgs/ship.png";
```

```
    this.bullet.src = "imgs/bullet.png";
    this.enemy.src = "imgs/enemy.png";
    this.enemyBullet.src = "imgs/bullet_enemy.png";
}

/*Creamos el objeto Drawable para dibujar, el cual se
```

En este paso solo se carga la nave enemiga y las balas enemigas a nuestro juego y usando la función `imageLoaded()` para asegurarnos de que todas hayan cargado antes de que empiece el juego (nos aseguramos de cambiar `numImages` a 5).

Lo siguiente que tenemos que hacer para añadir enemigos a nuestro juego es actualizar los objetos Bullet y Pool. La razón de esto es doble.

En primer lugar, los enemigos disparan una bala diferente que el jugador, y estas balas también tienen comportamientos diferentes a la bala del jugador. Segundo, la nave enemiga necesitará su pool de balas para manejar todas las balas que dispara, así que necesitamos actualizar el objeto Pool para manejar balas enemigas. El grupo de objetos también mantendrá las naves enemigas para que podamos continuar generando nuevas olas cuando se destruye la vieja ola.

```
function Bullet(object) {
    this.alive = false; //es verdadero si la bala esta actualmente en uso
    var self = object;
    //da los valores a la bala

    this.spawn = function(x, y, speed) {
        this.x = x;
        this.y = y;
        this.speed = speed;
        this.alive = true;
    };

    /*utiliza un rectangulo para borrar la bala y la mueve
    regresa true si la bala se movio fuera de la pantalla, indicando que
    la bala esta lista para ser limpiada por el pool, de otra manera
    dibuja la bala */
    this.draw = function() {
        this.context.clearRect(this.x-1, this.y-1, this.width+1, this.height+1);
        this.y -= this.speed;
        if (self === "bullet" && this.y <= 0 - this.height) {
            return true;
        }
        else if (self === "enemyBullet" && this.y >= this.canvasHeight) {
            return true;
        }
        else {
            if (self === "bullet") {
                this.context.drawImage(imageRepository.bullet, this.x, this.y);
            }
            else if (self === "enemyBullet") {
                this.context.drawImage(imageRepository.enemyBullet, this.x, this.y);
            }
        }
        return false;
    };
}
```

```

        return false;
    }
};

//reestablece los valores de la bala
this.clear = function() {
    this.x = 0;
    this.y = 0;
    this.speed = 0;
    this.alive = false;
};
Bullet.prototype = new Drawable();

```

El objeto Bullet recibe primero una cadena que indica qué tipo de bala será: un jugador "bullet" o un "enemyBullet". En la función draw (), determinamos si la bala ha salido de la pantalla basándose en qué bullet es (las balas del jugador salen de la pantalla en la parte superior, las bolas enemigas salen de la pantalla en la parte inferior). Si la bala no ha salido de la pantalla dibujamos la imagen apropiada en el canvas.

```

/*
llena el arreglo con los objetos dados
*/
this.init = function(object) {
    if (object == "bullet") {
        for (var i = 0; i < size; i++) {
            //inicializamos el objeto bala
            var bullet = new Bullet("bullet");
            bullet.init(0,0, imageRepository.bullet.width, imageRepository.bullet.height);
            pool[i] = bullet; //arreglo para las balas
        }
    }
    else if (object == "enemy") {
        for (var i = 0; i < size; i++) {
            var enemy = new Enemy();
            enemy.init(0,0, imageRepository.enemy.width, imageRepository.enemy.height);
            pool[i] = enemy;
        }
    }
    else if (object == "enemyBullet") {
        for (var i = 0; i < size; i++) {
            var bullet = new Bullet("enemyBullet");
            bullet.init(0,0, imageRepository.enemyBullet.width, imageRepository.enemyBullet.height);
            pool[i] = bullet;
        }
    }
};

```

Sólo tenemos que cambiar la función init () del objeto Pool. La función recibe una cadena que indica qué tipo de agrupación debe ser: una agrupación "bullet", una agrupación "enemy" o una agrupación "enemyBullet" y luego rellena el arreglo con el objeto designado.

### La nave enemiga

Los enemigos de nuestro juego imitarán el patrón de movimiento de los enemigos en Space Invaders, moviéndose de un lado a otro de la pantalla. La diferencia será que no se mueven por una fila cada vez que golpean el borde de la pantalla, sino que permanecen en su fila actual todo el tiempo. También haremos que aparezcan fuera de la pantalla y luego se mueven hacia abajo de la pantalla para llegar a sus filas.

```
*Crea el objeto de la nave enemiga.  
*/  
function Enemy() {  
    var percentFire = .01;  
    var chance = 0;  
    this.alive = false;  
  
    /*  
     * Establece los valores del enemigo  
     */  
    this.spawn = function(x, y, speed) {  
        this.x = x;  
        this.y = y;  
        this.speed = speed;  
        this.speedX = 0;  
        this.speedY = speed;  
        this.alive = true;  
        this.leftEdge = this.x - 90;  
        this.rightEdge = this.x + 90;  
        this.bottomEdge = this.y + 140;  
    };  
  
    /*  
     *Mueve al enemigo  
     */  
    this.draw = function() {  
        this.context.clearRect(this.x-1, this.y, this.width+1, this.height);  
        this.x += this.speedX;  
        this.y += this.speedY;  
        if (this.x <= this.leftEdge) {  
            this.speedX = this.speed;  
        }  
        else if (this.x >= this.rightEdge + this.width) {  
            this.speedX = -this.speed;  
        }  
        else if (this.y >= this.bottomEdge) {  
            this.speed = 1.5;  
            this.speedY = 0;  
        }  
    };  
};
```

```

        }
        else if (this.y >= this.bottomEdge) {
            this.speed = 1.5;
            this.speedY = 0;
            this.y -= 5;
            this.speedX = -this.speed;
        }

        this.context.drawImage(imageRepository.enemy, this.x, this.y);

        // El enemigo tiene chance de disparar cada momento
        chance = Math.floor(Math.random()*101);
        if (chance/100 < percentFire) {
            this.fire();
        }
    };

    /*
     * Dispara una bala
     */
    this.fire = function() {
        game.enemyBulletPool.get(this.x+this.width/2, this.y+this.height, -2.5);
    }

    /*
     * Reestablece los valores del enemigo
     */
    this.clear = function() {
        this.x = 0;
        this.y = 0;
        this.speed = 0;
        this.speedX = 0;
        this.speedY = 0;
        this.alive = false;
    };
}

Enemy.prototype = new Drawable();

```

Los enemigos tendrán una pequeña posibilidad de disparar una bala cada fotograma. Hacer esto ayuda al juego a sentirse diferente cada vez y no un patrón fijo que el jugador podría memorizar. .01 puede parecer un número muy pequeño, pero cuando se considera que se mueven 60 veces cada segundo, las probabilidades son que disparan al menos una vez cada dos o tres segundos. Y cuando estamos planeando tener 18 enemigos en la pantalla, cada uno disparando cada dos o tres segundos sigue siendo un montón de balas.

Debido a que los enemigos se mueven en un grupo y se mueven hacia adelante y hacia atrás, cada uno necesita saber hasta qué punto a la izquierda y la derecha puede moverse antes de que cambie de dirección. Este cambio es relativo a cada nave individual. Así, la función spawn () toma la posición x y y del inicio de la nave y luego le da tres límites: un borde izquierdo, un borde derecho y un borde inferior (ya que se moverán hacia abajo para comenzar, queremos que ellos final en sus filas apropiadas). Así que cada nave se moverá hacia adelante y hacia atrás, con un rango de 180px.

La función draw () es directa, indicando a la nave que cambie de dirección cuando golpea el borde izquierdo o derecho, y deje de moverse hacia abajo y comience a moverse hacia la izquierda cuando toque el borde inferior. Golpear el borde inferior también disminuirá la velocidad de la nave un poco de su velocidad inicial. Utilizaremos la función Math.random () para generar un número aleatorio entre 1 y 100 para ver si el enemigo disparará este marco.

Las balas que se mueven dependen en gran medida de la nave enemiga que está viva, por lo que perder el enemigo también afecta a las balas. Lo que queremos en cambio es un sistema flojo acoplado donde si una nave muere las balas seguirán funcionando correctamente. Para hacer esto, tendremos un objeto de pool enemigo que maneja todas las balas para todas las naves enemigas y es controlado por el objeto Game.

## El paso final

El último paso es de nuevo actualizar el juego y las funciones animadas para implementar las naves enemigas y las balas.

```
/*con la estructura basica del juego completo, es hora de crear el objeto final
que manejará el juego entero*/
//se crea el objeto Game, el cual tomará todos los objetos y datos para el juego.
function Game() {
    /*obtiene la información del canvas y context y actualiza todos los objetos del juego
    -regresa true si el canvas es soportado y falso si no, esto para parar
    la animación de correr constantemente en navegadores más viejos*/
    this.init = function() {
        //obtiene el elemento canvas
        this.bgCanvas = document.getElementById('background');
        this.shipCanvas = document.getElementById('ship');
        this.mainCanvas = document.getElementById('main');

        //Testeo para saber si el canvas es soportado

        if (this.bgCanvas.getContext) {
            this.bgContext = this.bgCanvas.getContext('2d');
            this.shipContext = this.shipCanvas.getContext('2d');
            this.mainContext = this.mainCanvas.getContext('2d');

            //inicializa los objetos para contener la información de su context y canvas

            Background.prototype.context = this.bgContext;
            Background.prototype.canvasWidth = this.bgCanvas.width;
            Background.prototype.canvasHeight = this.bgCanvas.height;

            Ship.prototype.context = this.shipContext;
            Ship.prototype.canvasWidth = this.shipCanvas.width;
            Ship.prototype.canvasHeight = this.shipCanvas.height;

            Bullet.prototype.context = this.mainContext;
            Bullet.prototype.canvasWidth = this.mainCanvas.width;
            Bullet.prototype.canvasHeight = this.mainCanvas.height;

            Enemy.prototype.context = this.mainContext;
            Enemy.prototype.canvasWidth = this.mainCanvas.width;
            Enemy.prototype.canvasHeight = this.mainCanvas.height;
        }
    }
}
```

```

        Enemy.prototype.context = this.mainContext;
        Enemy.prototype.canvasWidth = this.mainCanvas.width;
        Enemy.prototype.canvasHeight = this.mainCanvas.height;

        //Inicializa el objeto background
        this.background = new Background();
        this.background.init(0,0); //Establece el punto de inicio de dibujo como (0,0)
        //Iniciar el objeto ship(nave)
        this.ship = new Ship();
        // establecer la nave para empezar cerca del medio del canvas
        var shipStartX = this.shipCanvas.width/2 - imageRepository.spaceship.width;
        var shipStartY = this.shipCanvas.height/4*3 + imageRepository.spaceship.height*2;
        this.ship.init(shipStartX, shipStartY, imageRepository.spaceship.width,
                      imageRepository.spaceship.height);

        // Inicializa el objeto pool del enemigo
        this.enemyPool = new Pool(30);
        this.enemyPool.init("enemy");
        var height = imageRepository.enemy.height;
        var width = imageRepository.enemy.width;
        var x = 100;
        var y = -height;
        var spacer = y * 1.5;
        for (var i = 1; i <= 18; i++) {
            this.enemyPool.get(x,y,2);
            x += width + 25;
            if (i % 6 == 0) {
                x = 100;
                y += spacer
            }
        }

        this.enemyBulletPool = new Pool(50);
        this.enemyBulletPool.init("enemyBullet");

        return true;
    } else {
        return false;
    }
}

```

```

    //comienza el bucle de animacion
    this.start = function() {
        this.ship.draw();
        animate();
    };
}

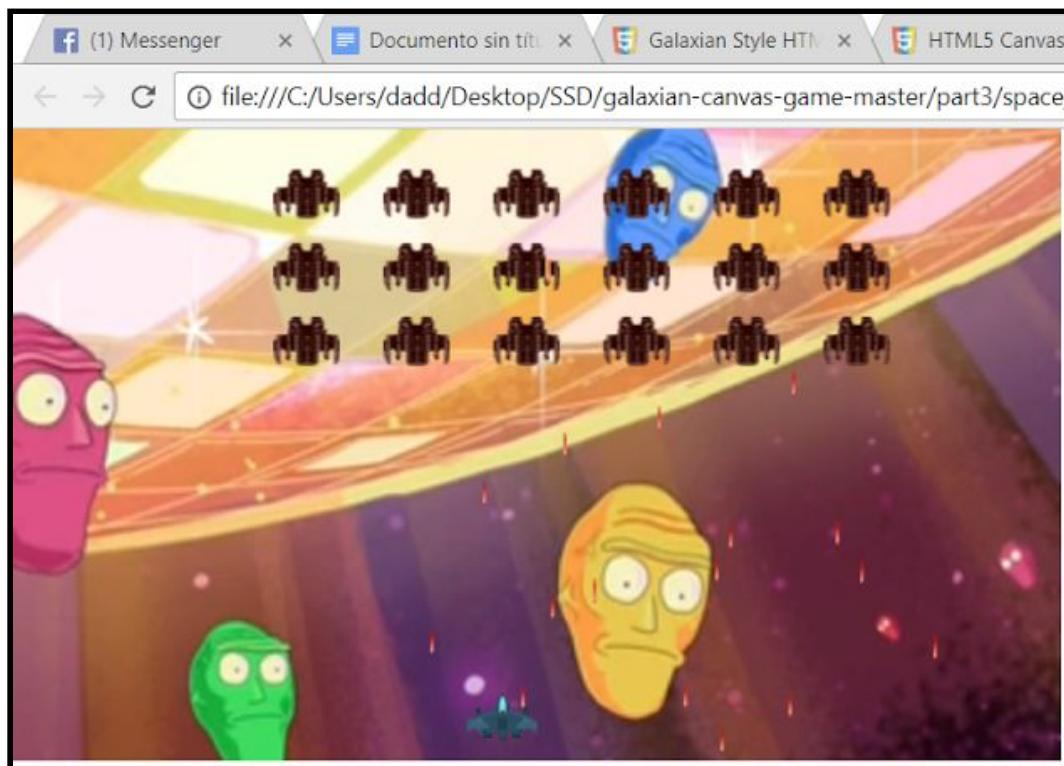
/* Con el objeto Game terminado, lo único que queda por hacer es crear el bucle de animación.*/
/* el bucle de animacion. llama a requestAnimationFrame para optimizar
el bucle del juego y dibuja todos los objetos del juego. esta funcion tiene que ser
una funcion global y no puede estar sin un objeto*/

function animate() {
    requestAnimationFrame( animate );
    /* informa al navegador que quieras realizar una animación y solicita que el navegador programe el repaintido de la ventana para el próximo
    ciclo de animación. El método acepta como argumento una función a la que llamar antes de efectuar el repaintido
    Debes llamar a este método cuando estés preparado para actualizar tu animación en la pantalla para pedir que se programe el repaintido.
    Ésto puede suceder hasta 60 veces por segundo en pestañas en primer plano, pero se puede ver reducido a velocidades inferiores en
    pestañas en segundo plano
    */
    game.background.draw();
    game.ship.move();
    game.ship.bulletPool.animate();
    game.enemyPool.animate();
    game.enemyBulletPool.animate();
}

```

Una vez más, nada demasiado nuevo aquí. El mayor cambio es la implementación de las naves enemigas. Tendremos 18 naves enemigas: 3 filas de 6 naves. Debido a que queremos que comience la pantalla, comenzamos la fila inferior justo por encima del borde superior ( $y = -height$ ) y el enemigo más a la izquierda en 100px. La variable espaciadora determina la distancia entre cada fila. Dentro del bucle, creamos 18 enemigos, cada uno se coloca 25pxs aparte de la anterior en la fila. Cuando tenemos seis enemigos en una fila, subimos una fila y reiniciamos los enemigos a 100px.

Este es nuestro resultado final del paso 3:



Paso 4:

#### Detección de colisiones 2D

Para comenzar, hablaremos sobre cómo manejar la detección de colisiones 2D. Hay muchos algoritmos para detectar cuando dos objetos están tocando en el espacio 2D. Para nuestro juego simple, no tenemos que implementar un algoritmo de detección complejo ya que los videojuegos sólo tienen que estimarse razonablemente bien.

Por lo tanto, usaremos una prueba de caja delimitadora para nuestra detección de colisión.

```
// ALGORITMO DE DETECCION DE COLISION
if (objects[x].collidableWith === obj[y].type &&
    objects[x].x < obj[y].x + obj[y].width &&
    objects[x].x + objects[x].width > obj[y].x &&
    objects[x].y < obj[y].y + obj[y].height &&
    objects[x].y + objects[x].height > obj[y].y) {
    objects[x].isColliding = true;
    obj[y].isColliding = true;
}
};

};
```

Un algoritmo de detección de colisión de caja delimitadora toma dos objetos y comprueba si los límites del primer objeto están dentro de los límites del segundo objeto. Requiere cuatro controles, uno para cada borde del cuadro delimitador.

Para implementar la detección de colisiones, podríamos simplemente poner este algoritmo en un bucle y comprobar cada objeto contra todos los demás objetos. Sin embargo, piense en cuántos objetos están presentes en nuestro juego. En cualquier momento podría haber 18 enemigos, una nave de un jugador, hasta 30 balas de jugador y hasta 50 balas enemigas. Eso es casi 100 objetos que podrían estar en la pantalla a la vez.

Si tuviéramos que ejecutar este algoritmo en un bucle como se describió anteriormente, el juego tendría que realizar unos 5.000 cheques sólo para comprobar si dos objetos chocan. Y como tenemos que ejecutar nuestro algoritmo de detección de colisiones en cada fotograma, puede realmente ralentizar el juego. Así que lo que necesitamos hacer es ayudar a acelerar la detección de colisiones.

## Mejora de la detección de colisiones 2D

Simplemente mejorar el algoritmo de caja delimitadora no nos ayudará a acelerar la detección de colisión porque el principal problema radica en cuántas comprobaciones el algoritmo tiene que realizar. Por lo tanto, la mejor manera de acelerar la detección de colisiones es reducir el número de comprobaciones que el algoritmo debe realizar. Para ello, utilizaremos una técnica conocida como partición espacial.

## Particionamiento espacial

La división espacial toma un espacio confinado (en este caso nuestro juego) y lo subdivide en múltiples espacios más pequeños. Al subdividir el espacio, podemos determinar rápidamente qué objetos pertenecen a la misma área y sólo tenemos que ejecutar la detección de colisión contra esos objetos. Cualquier objeto que no pertenezca al mismo espacio no puede colisionar, por lo que no tenemos que desperdiciar un cheque en él.

Al dividir el espacio, somos capaces de reducir nuestros controles de detección de colisión de 5.000 a alrededor de 100 por cuadro, eso es una gran mejora.

Para implementar esta técnica, usaremos una estructura de datos llamada quadtree.

```
/**  
 * QuadTree object.  
 *  
 * The quadrant indexes are numbered as below:  
 *  
 *   1 | 0  
 *   -+---  
 *   2 | 3  
 *       |  
 */  
function QuadTree(boundBox, lvl) {  
    var maxObjects = 10;  
    this.bounds = boundBox || {  
        x: 0,  
        y: 0,  
        width: 0,  
        height: 0  
    };  
    var objects = [];  
    this.nodes = [];  
    var level = lvl || 0;  
    var maxLevels = 5;  
  
    /*  
     * limpia el cuadrante y todos los nodos de los objetos  
     */  
    this.clear = function() {  
        objects = [];  
  
        for (var i = 0; i < this.nodes.length; i++) {  
            this.nodes[i].clear();  
        }  
  
        this.nodes = [];  
    };
```

```
/*
 * obtiene todos los objetos en el cuadrante    */
this.getAllObjects = function(returnedObjects) {
    for (var i = 0; i < this.nodes.length; i++) {
        this.nodes[i].getAllObjects(returnedObjects);
    }

    for (var i = 0, len = objects.length; i < len; i++) { ... }

    return returnedObjects;
};

/*
Devuelve todos los objetos con los que el objeto podría colisionar   */
this.findObjects = function(returnedObjects, obj) { ... };

this.insert = function(obj) {
    if (typeof obj === "undefined") {
        return;
    }

    if (obj instanceof Array) {
        for (var i = 0, len = obj.length; i < len; i++) {
            this.insert(obj[i]);
        }
    }

    return;
}

if (this.nodes.length) {
    var index = this.getIndex(obj);
    // Sólo agregue el objeto a un subnodo si puede caber completamente dentro de un
    if (index != -1) {
        this.nodes[index].insert(obj);
```

```
        this.nodes[index].insert(obj);

        return;
    }
}

objects.push(obj);

// Evitar la división infinita
if (objects.length > maxObjects && level < maxLevels) {
    if (this.nodes[0] == null) {
        this.split();
    }

    var i = 0;
    while (i < objects.length) {

        var index = this.getIndex(objects[i]);
        if (index != -1) {
            this.nodes[index].insert((objects.splice(i,1))[0]);
        }
        else {
            i++;
        }
    }
}
};

/*
 * Determine a qué nodo pertenece el objeto. -1 medios
 * el objeto no puede encajar completamente dentro de un nodo y es parte
 * del nodo actual
 */
this.getIndex = function(obj) {

    var index = -1;
    var verticalMidpoint = this.bounds.x + this.bounds.width / 2;
    var horizontalMidpoint = this.bounds.y + this.bounds.height / 2;
```

```
// El objeto puede caber completamente dentro del cuadrante superior
var topQuadrant = (obj.y < horizontalMidpoint && obj.y + obj.height < horizontalMidpoint);
// El objeto puede caber completamente dentro del cuadrante inferior
t
var bottomQuadrant = (obj.y > horizontalMidpoint);

// El objeto puede caber completamente dentro de los cuadrantes izquierdos

if (obj.x < verticalMidpoint &&
    obj.x + obj.width < verticalMidpoint) {
    if (topQuadrant) {
        index = 1;
    }
    else if (bottomQuadrant) {
        index = 2;
    }
}
// Objeto puede arreglar completamente dentro de los cuadrantes derechos

else if (obj.x > verticalMidpoint) {
    if (topQuadrant) {
        index = 0;
    }
    else if (bottomQuadrant) {
        index = 3;
    }
}

return index;
};

/*
 * Divide el nodo en 4 subnodos
 */
this.split = function() {
    // Bitwise or [html5rocks]
    var subWidth = (this.bounds.width / 2) | 0;
```

```
/*
 * Divide el nodo en 4 subnodos
 */
this.split = function() {
    // Bitwise or [html5rocks]
    var subWidth = (this.bounds.width / 2) | 0;
    var subHeight = (this.bounds.height / 2) | 0;

    this.nodes[0] = new QuadTree({
        x: this.bounds.x + subWidth,
        y: this.bounds.y,
        width: subWidth,
        height: subHeight
    }, level+1);
    this.nodes[1] = new QuadTree({
        x: this.bounds.x,
        y: this.bounds.y,
        width: subWidth,
        height: subHeight
    }, level+1);
    this.nodes[2] = new QuadTree({
        x: this.bounds.x,
        y: this.bounds.y + subHeight,
        width: subWidth,
        height: subHeight
    }, level+1);
    this.nodes[3] = new QuadTree({
        x: this.bounds.x + subWidth,
        y: this.bounds.y + subHeight,
        width: subWidth,
        height: subHeight
    }, level+1);
};

}
```

Un quadtree indexará todos los objetos del juego en nodos y subnodos. Todos los objetos dentro de un nodo podrían colisionar, por lo que solo verificamos aquellos objetos para la colisión. Con el quadtree en la mano, estamos casi listos para implementar la detección de colisiones en nuestro juego.

## Actualización de objetos para la detección de colisiones

Todavía hay una cosa más que hacer antes de implementar la detección de colisiones 2D. No todos los objetos pueden chocar con cualquier otro objeto. Por ejemplo, una nave no chocará con su propia bala, lo mismo ocurre con una nave enemiga. Así que debemos comprobar si los dos objetos deben colisionar antes de comprobar si están chocando. Esto significa que cada objeto contiene una lista de objetos con los que puede chocar y el algoritmo de detección de colisión comparará esta lista con el tipo del otro objeto.

Para comenzar a implementar la detección de colisiones, editaremos el objeto Dibujable:

```
/*Creamos el objeto Drawable para dibujar, el cual sera la clase base para
todos los objetos dibujables en el juego. Se definen variables por
default que todos los objetos hijo van a heredar, asi como las
funciones por defecto*/
function Drawable() {//Este objeto es un objeto abstracto, el cual todos los objetos del juego lo van a heredar
    this.init = function(x, y, width, height) {
        //Variables por default
        this.x = x;
        this.y = y;
        this.width = width;//como vamos a utilizar imagenes que no seran
        this.height = height;//del tamaño del canvas, tenemos que definir alto y ancho de esas imagenes
    }

    this.speed = 0;
    this.canvasWidth = 0;
    this.canvasHeight = 0;
    this.collidableWith = "";
    this.isColliding = false;
    this.type = "";

    //Se definen la función abstracta a implementar en los objetos hijo
    this.draw = function() {
    };
    this.move = function() {
    };
    this.isCollidableWith = function(object) {
        return (this.collidableWith === object.type);
    };
}
```

El objeto recibe dos nuevas variables: collidableWith y isColliding, así como una nueva función isCollidableWith () para probar si el objeto puede colisionar con el objeto dado. La variable isColliding nos dice cuándo un objeto está chocando y no debe ser redibujado al canvas.

Con la necesidad de tener una lista de objetos colidibles, cada objeto en nuestro juego tiene que implementar este cambio. Empezaremos con el objeto Bullet.

```
/*
 * El código es lo que se movió hacia un punto más tarde, indicando que
 * la bala está lista para ser limpiada por el pool, de otra manera
 * dibuja la bala */
this.draw = function() {
    this.context.clearRect(this.x-1, this.y-1, this.width+2, this.height+2);
    this.y -= this.speed;

    if (this.isColliding) {
        return true;
    }
    else if (self === "bullet" && this.y <= 0 - this.height) {
        return true;
    }
    else if (self === "enemyBullet" && this.y >= this.canvasHeight) {
        return true;
    }
    else {
        if (self === "bullet") {
            this.context.drawImage(imageRepository.bullet, this.x, this.y);
        }
        else if (self === "enemyBullet") {
            this.context.drawImage(imageRepository.enemyBullet, this.x, this.y);
        }

        return false;
    }
};

/*
//reestablece los valores de la bala
*/
this.clear = function() {
    this.x = 0;
    this.y = 0;
    this.speed = 0;
    this.alive = false;
    this.isColliding = false;
};
```

Sólo se deben cambiar las funciones draw () y clear (). La función draw agrega la comprobación para ver si el objeto está chocando. Si es así, devuelve true al pool de objetos para que sepa que el bullet puede ser reutilizado. La función clear simplemente restablece la variable isColliding a false.

El siguiente objeto que actualizaremos es el objeto Ship:

```
al rededor de la pantalla */
function Ship() {
    this.speed = 3;
    this.bulletPool = new Pool(30);
    var fireRate = 15;
    var counter = 0;
    this.collidableWith = "enemyBullet";
    this.type = "ship";

    this.init = function(x, y, width, height) {
        //Variables por Default
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.alive = true;
        this.isColliding = false;
        this.bulletPool.init("bullet");
    }

    this.draw = function() {
        this.context.drawImage(imageRepository.spaceship, this.x, this.y);
    };
    this.move = function() {
        counter++;
        //determina si la accion es mover
        if (KEY_STATUS.left || KEY_STATUS.right ||
            KEY_STATUS.down || KEY_STATUS.up) {
            //la nave se movio, entonces se borra la imagen actual para que pueda
            //ser redibujada en su nueva locacion
            this.context.clearRect(this.x, this.y, this.width, this.height);

            //actualiza x y y de acuerdo con la direccion de movimiento
            //y redibuja la nave, cambia el else if's to id=f statements
            //para tener movimientos diagonales
            if (KEY_STATUS.left) {
                this.x -= this.speed
                //redibujando la nave, cambia el else if's to id=f statements
                //para tener movimientos diagonales
                if (KEY_STATUS.left) {
                    this.x -= this.speed
                    if (this.x <= 0) //mantiene al jugador dentro de la pantalla
                        this.x = 0;
                } else if (KEY_STATUS.right) {
                    this.x += this.speed
                    if (this.x >= this.canvasWidth - this.width)
                        this.x = this.canvasWidth - this.width;
                } else if (KEY_STATUS.up) {
                    this.y -= this.speed
                    if (this.y <= this.canvasHeight/4*3)
                        this.y = this.canvasHeight/4*3;
                } else if (KEY_STATUS.down) {
                    this.y += this.speed
                    if (this.y >= this.canvasHeight - this.height)
                        this.y = this.canvasHeight - this.height;
                }
            }
            //finalizando el redibujado de la nave
            if (!this.isColliding) {
                this.draw();
            }
            else {
                this.alive = false;
                game.gameOver();
            }
        }
        if (KEY_STATUS.space && counter >= fireRate && !this.isColliding) {
            this.fire();
            counter = 0;
        }
    };
}
```

El objeto de la nave fijará su tipo y será collidable con las balas enemigas. En la función draw (), sólo redibujaremos la nave y la dejaremos disparar si no está chocando con nada.

El siguiente es el objeto Enemigo:

```
function Enemy() {
    var percentFire = .01;
    var chance = 0;
    this.alive = false;
    this.collidableWith = "bullet";
    this.type = "enemy";

    /*
     * Establece los valores del enemigo
     */
    this.spawn = function(x, y, speed) {
        this.x = x;
        this.y = y;
        this.speed = speed;
        this.speedX = 0;
        this.speedY = speed;
        this.alive = true;
        this.leftEdge = this.x - 90;
        this.rightEdge = this.x + 90;
        this.bottomEdge = this.y + 140;
    };

    /*
     * Mueve al enemigo
     */
    this.draw = function() {
        this.context.clearRect(this.x-1, this.y, this.width+1, this.height);
        this.x += this.speedX;
        this.y += this.speedY;
        if (this.x <= this.leftEdge) {
            this.speedX = this.speed;
        }
        else if (this.x >= this.rightEdge + this.width) {
            this.speedX = -this.speed;
        }
        else if (this.y >= this.bottomEdge) {
            this.speed = 1.5;
            this.speedY = 0;
        }
    };
}
```

```

        if (this.isColliding) {
            this.context.drawImage(imageRepository.enemy, this.x, this.y);

            // El enemigo tiene chance de disparar todo el tiempo
            chance = Math.floor(Math.random()*101);
            if (chance/100 < percentFire) {
                this.fire();
            }

            return false;
        }
        else {
            game.playerScore += 10;
            game.explosion.get();
            return true;
        }
    };

/*
 * Dispara una bala
 */
this.fire = function() {
    game.enemyBulletPool.get(this.x+this.width/2, this.y+this.height, -2.5);
};

/*
 * Reestablece los valores del enemigo
 */
this.clear = function() {
    this.x = 0;
    this.y = 0;
    this.speed = 0;
    this.speedX = 0;
    this.speedY = 0;
    this.alive = false;
    this.isColliding = false;
};

```

El objeto enemigo establecerá su tipo y será colindable con las balas del jugador. Sólo se redibujará y disparará si no está colisionando, y la función clear () restablece la variable isColliding.

El último objeto a editar es el objeto Pool:

```

this.init = function(object) {
    if (object == "bullet") {
        for (var i = 0; i < size; i++) {
            // Initialize the object
            var bullet = new Bullet("bullet");
            bullet.init(0,0, imageRepository.bullet.width,
                        imageRepository.bullet.height);
            bullet.collidableWith = "enemy";
            bullet.type = "bullet";
            pool[i] = bullet;
        }
    }
    else if (object == "enemy") {
        for (var i = 0; i < size; i++) {
            var enemy = new Enemy();
            enemy.init(0,0, imageRepository.enemy.width,
                      imageRepository.enemy.height);
            pool[i] = enemy;
        }
    }
    else if (object == "enemyBullet") {
        for (var i = 0; i < size; i++) {
            var bullet = new Bullet("enemyBullet");
            bullet.init(0,0, imageRepository.enemyBullet.width,
                        imageRepository.enemyBullet.height);
            bullet.collidableWith = "ship";
            bullet.type = "enemyBullet";
            pool[i] = bullet;
        }
    }
};

//toma el ultimo articulo en la lista y lo inicializa
//y luego lo pone al principio del arreglo
this.get = function(x, y, speed) {
    if(!pool[size - 1].alive) {
        pool[size - 1].spawn(x, y, speed);
        pool.unshift(pool.pop());
    }
}

```

Para cada tipo de bala, las balas del jugador y las balas enemigas, estableceremos el tipo y las variables colindableWith en consecuencia. También agregamos una nueva función, getPool () para devolver todos los objetos vivos en la agrupación como un arreglo que luego se insertará en el quadtree.

Con todos nuestros objetos actualizados, estamos casi listos para implementar la detección de colisiones.

### El paso final

Para implementar el árbol quad en nuestro juego, agregamos sólo una línea al final del objeto Game, justo antes de devolver true:

```
// Start QuadTree
this.quadTree = new QuadTree({x:0,y:0,width:this.mainCanvas.width,height:this.mainCanvas.height});
```

A continuación, podemos actualizar la función animate () para usar el quadtree:

```
/*
function animate() {
    document.getElementById('score').innerHTML = game.playerScore;

    // inserta objetos en el quadtree
    game.quadTree.clear();
    game.quadTree.insert(game.ship);
    game.quadTree.insert(game.ship.bulletPool.getPool());
    game.quadTree.insert(game.enemyPool.getPool());
    game.quadTree.insert(game.enemyBulletPool.getPool());

    detectCollision();

    // no mas enemigos
    if (game.enemyPool.getPool().length === 0) {
        game.spawnWave();
    }

    // Animate game objects
    if (game.ship.alive) {
        requestAnimationFrame( animate );

        game.background.draw();
        game.ship.move();
        game.ship.bulletPool.animate();
        game.enemyPool.animate();
        game.enemyBulletPool.animate();
    }
}
```

Lo primero que hacemos es despejar el quadtree y añadir todos los objetos a él. Una vez que todos los objetos se agregan al quadtree, ejecutamos nuestro algoritmo de detección de colisiones en detectCollision (). Terminamos moviendo y animando todos los objetos en la pantalla. Esta orden asegura que los objetos que chocan en el marco no se vuelvan a dibujar al final del marco.

El último bit de código es la función detectCollision ():

```

function detectCollision() {
    var objects = [];
    game.quadTree.getAllObjects(objects);

    for (var x = 0, len = objects.length; x < len; x++) {
        game.quadTree.findObjects(obj = [], objects[x]);

        for (y = 0, length = obj.length; y < length; y++) {

            // ALGORITMO DE DETECCION DE COLISION
            if (objects[x].collidableWith === obj[y].type &&
                (objects[x].x < obj[y].x + obj[y].width &&
                 objects[x].x + objects[x].width > obj[y].x &&
                 objects[x].y < obj[y].y + obj[y].height &&
                 objects[x].y + objects[x].height > obj[y].y)) {
                objects[x].isColliding = true;
                obj[y].isColliding = true;
            }
        }
    }
};

```

La función primero obtiene todos los objetos en el quadtree y los guarda en una sola matriz. A continuación, realiza un bucle sobre cada objeto de la matriz y recupera otra matriz de objetos con los que el objeto podría colisionar. Por último, ejecuta el algoritmo de detección de colisiones contra estos objetos para ver si chocan.

Ahora tenemos un juego que es capaz de comprobar las colisiones en 60FPS. Se tardó un poco de refactorización para llegar allí, pero todo valió la pena al final.

Paso 5:

El juego ahora tiene sonido (en el fondo, cuando disparas un láser, cuando golpeas a un enemigo, y cuando mueres) y sigue la pista de tu puntuación. También te permitirá reiniciar el juego si te golpean y generan otra ola de enemigos cuando matas al último en la pantalla.

La página HTML5

Comenzaremos este tutorial agregando la puntuación del jugador y el juego sobre la pantalla. Ambos se realizan modificando la página HTML:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Space Shooter Demo</title>
    <style>
      canvas {
        position: absolute;
        top: 0px;
        left: 0px;
        background: transparent;
      }
      #background {
        z-index: -2;
      }
      #main {
        z-index: -1;
      }
      #ship {
        z-index: 0;
      }
      .score {
        position: absolute;
        top: 5px;
        left: 480px;
        color: #5fce1e;
        font-family: Helvetica, sans-serif;
        cursor: default;
      }
      .game-over {
        position: absolute;
        top: 150px;
        left: 120px;
        color: #5fce1e;
        font-family: Helvetica, sans-serif;
        font-size: 50px;
        cursor: default;
        display: none;
      }
    </style>
  </head>
  <body>
    <!-- The canvas for the panning background -->
    <canvas id="background" width="600" height="360">
      Your browser does not support canvas. Please try again with a different browser.
    </canvas>
    <!-- The canvas for all enemy ships and bullets -->
    <canvas id="main" width="600" height="360">
    </canvas>
    <!-- The canvas the ship uses (can only move up
    one forth the screen. -->
    <canvas id="ship" width="600" height="360">
    </canvas>
    <div class="score">SCORE: <span id="score"></span></div>
    <div class="game-over" id="game-over">GAME OVER<p><span onclick="game.restart()">Restart</span></p></div>
    <div class="loading" id="loading">LOADING...<p>Please wait</p></div>
    <script src="space_shooter_part_five.js"></script>
  </body>
</html>

```

Añadimos dos divs nuevos debajo del canvas, uno para mostrar la puntuación de los jugadores y el otro para mostrar el juego sobre la pantalla. También añadimos

nuevos estilos para colocar estos elementos en sus ubicaciones correctas en la pantalla y hacer que se vean bien.

### La puntuación del jugador

Para implementar la puntuación del jugador, agregamos una variable en el objeto Juego y modificamos la función draw () del objeto Enemigo para agregar puntos a la puntuación cuando el barco está chocando.

```
        this.speedX = -this.speed;
    }
    else if (this.y >= this.bottomEdge) {
        this.speed = 1.5;
        this.speedY = 0;
        this.y -= 5;
        this.speedX = -this.speed;
    }

    if (!this.iscolliding) {
        this.context.drawImage(imageRepository.enemy, this.x, this.y);

        // El enemigo tiene chance de disparar todo el tiempo
        chance = Math.floor(Math.random()*101);
        if (chance/100 < percentFire) {
            this.fire();
        }
    }

    return false;
}
else {
    game.playerScore += 10;
    game.explosion.get();
    return true;
}
};

/*
 * Dispara una bala
 */
this.fire = function() {
    game.enemyBulletPool.get(this.x+this.width/2, this.y+this.height, -2.5);
};

/*
 * Reestablece los valores del enemigo
 */
this.clear = function() {
```

### Adición de audio HTML5

Para añadir audio HTML5 al juego, comenzaremos creando la agrupación de sonidos. Es muy similar a nuestra piscina de objetos, pero necesita ser ligeramente modificado para manejar el sonido.

```

/*
 * A sound pool to use for the sound effects
 */
function SoundPool(maxSize) {
    var size = maxSize; // Max balas permitidas en el pool
    var pool = [];
    this.pool = pool;
    var currSound = 0;

    /*
     * llena el arreglo con los objetos dados
     */
    this.init = function(object) {
        if (object == "laser") {
            for (var i = 0; i < size; i++) {
                // Inicializa el objeto
                laser = new Audio("sounds/laser.wav");
                laser.volume = .12;
                laser.load();
                pool[i] = laser;
            }
        } else if (object == "explosion") {
            for (var i = 0; i < size; i++) {
                var explosion = new Audio("sounds/explosion.wav");
                explosion.volume = .1;
                explosion.load();
                pool[i] = explosion;
            }
        }
    };
}

```

La agrupación de sonidos es diferente de la agrupación de objetos que usamos para los objetos de juego, ya que no necesita empujar y hacer estallar los sonidos de la matriz. Un elemento de audio se reproducirá hasta el final de la pista, a continuación, establecer su estado finalizado a verdadero. Por lo tanto, sólo podemos hacer un bucle sobre la matriz de sonidos, jugando sólo si es currentTime = 0 o si ha terminado. Sólo necesitamos asegurarnos de que tenemos suficientes objetos de sonido en el pool que nunca nos topamos con el problema de no poder reproducir un sonido. Llamar a la función load () de un elemento Audio le dice al navegador que cargue el archivo de audio (ayuda con los navegadores más antiguos).

Con el pool de sonido lista, ahora podemos agregar el audio del juego a nuestro objeto de juego:

```

        this.enemyBulletPool = new Pool(50);
        this.enemyBulletPool.init("enemyBullet");

        // Start QuadTree
        this.quadTree = new QuadTree({x:0,y:0,width:this.mainCanvas.width,height:this.mainCanvas.height});

        this.playerScore = 0;

        // Archivos de audio
        this.laser = new SoundPool(10);
        this.laser.init("laser");

        this.explosion = new SoundPool(20);
        this.explosion.init("explosion");

        this.backgroundAudio = new Audio("sounds/kick_shock.wav");
        this.backgroundAudio.loop = true;
        this.backgroundAudio.volume = .25;
        this.backgroundAudio.load();

        this.gameOverAudio = new Audio("sounds/game_over.wav");
        this.gameOverAudio.loop = true;
        this.gameOverAudio.volume = .25;
        this.gameOverAudio.load();

        this.checkAudio = window.setInterval(function(){checkReadyState()},1000);
    }
}

```

La función init () ahora configura los objetos de audio que necesitamos para los láseres, las explosiones y los audios de fondo para el juego y el juego a través de la pantalla. El indicador de bucle le indica al elemento de audio que se reproducirá cuando haya terminado (creando un bucle de sonido infinito). Diez sonidos láser es suficiente con nuestra tasa de fuego actual de la nave. Si aumenta la tasa de fuego de la nave, asegúrese de agregar más láseres a la piscina de sonido. Dado que la nave puede disparar dos láseres a la vez, agregaremos el doble de sonidos de explosión que los sonidos láser a la piscina de sonido.

Para garantizar que los sonidos del juego se hayan cargado completamente para evitar errores de excepción de DOM, creamos un intervalo de tiempo para comprobar el estado listo de nuestros archivos de audio más grandes: game\_over.wav y kick\_shock.wav.

## Últimos retoques

Para que el juego sea más desafiante, agregaremos ondas infinitas al juego. Para ello, movemos el código para inicializar el objeto de grupo enemigo en el objeto Juego en su propia función y lo llamamos cada vez que queremos generar una nueva ola.

```

// Genera una nueva ola de enemigos
this.spawnWave = function() {
    var height = imageRepository.enemy.height;
    var width = imageRepository.enemy.width;
    var x = 100;
    var y = -height;
    var spacer = y * 1.5;
    for (var i = 1; i <= 18; i++) {
        this.enemyPool.get(x,y,2);
        x += width + 25;
        if (i % 6 == 0) {
            x = 100;
            y += spacer
        }
    }
}

//comienza el bucle de animacion

```

Luego dentro del bucle animado, comprobamos si no hay más enemigos y generamos una nueva ola si no lo hay.

```

// no mas enemigos
if (game.enemyPool.getPool().length === 0) {
    game.spawnWave();
}

// Animate game objects
if (game.ship.alive) {
    requestAnimFrame( animate );
}

```

La última cosa a implementar es el juego y reiniciar la funcionalidad. Cuando el jugador muere, queremos que aparezca el texto "Game Over" en la pantalla, el juego para detener la reproducción y la capacidad del jugador para reiniciar el juego desde el principio. Para ello, crearemos dos nuevas funciones en el objeto Game: gameOver () y restart ().

```

    // Reinicia el juego
    this.restart = function() {
        this.gameOverAudio.pause();

        document.getElementById('game-over').style.display = "none";
        this.bgContext.clearRect(0, 0, this.bgCanvas.width, this.bgCanvas.height);
        this.shipContext.clearRect(0, 0, this.shipCanvas.width, this.shipCanvas.height);
        this.mainContext.clearRect(0, 0, this.mainCanvas.width, this.mainCanvas.height);

        this.quadTree.clear();

        this.background.init(0,0);
        this.ship.init(this.shipStartX, this.shipStartY,
                      imageRepository.spaceship.width, imageRepository.spaceship.height);

        this.enemyPool.init("enemy");
        this.spawnWave();
        this.enemyBulletPool.init("enemyBullet");

        this.playerScore = 0;

        this.backgroundAudio.currentTime = 0;
        this.backgroundAudio.play();

        this.start();
    };

    // Game over
    this.gameOver = function() {
        this.backgroundAudio.pause();
        this.gameOverAudio.currentTime = 0;
        this.gameOverAudio.play();
        document.getElementById('game-over').style.display = "block";
    };
}

```

La función gameOver () simplemente detiene el audio de fondo de la reproducción e inicia el juego a través de audio, así como muestra el juego sobre el texto que escondió. La función restart () sólo toma todos los elementos del juego de nuevo a su estado inicial y comienza el juego de nuevo.

Para detener la reproducción del juego cuando el jugador es golpeado, solo añadimos una condición en el bucle animado para no recordarla si el jugador ya no está vivo.

```
840 // Esta función debe ser una función global y no puede estar dentro de un objeto.
841 */
842 function animate() {
843     document.getElementById('score').innerHTML = game.playerScore;
844
845     // inserta objetos en el quadtree
846     game.quadTree.clear();
847     game.quadTree.insert(game.ship);
848     game.quadTree.insert(game.ship.bulletPool.getPool());
849     game.quadTree.insert(game.enemyPool.getPool());
850     game.quadTree.insert(game.enemyBulletPool.getPool());
851
852     detectCollision();
853
854     // no mas enemigos
855     if (game.enemyPool.getPool().length === 0) {
856         game.spawnWave();
857     }
858
859     // Animate game objects
860     if (game.ship.alive) {
861         requestAnimFrame( animate );
862
863         game.background.draw();
864         game.ship.move();
865         game.ship.bulletPool.animate();
866         game.enemyPool.animate();
867         game.enemyBulletPool.animate();
868     }
869 }
870
871 function detectCollision() {
872     var objects = [];
873     game.quadTree.getAllObjects(objects);
874
875     for (var x = 0, len = objects.length; x < len; x++) {
```