UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Elaboración de una librería sobre *JavaScript* para el desarrollo de juegos 2D compatible con XNA

Víctor Hernández Molpeceres

Tutor: Carlos Aguirre Maeso

Mayo 2015

Resumen

El desarrollo de videojuegos se ha simplificado mucho en los últimos 25 años, antes se programaban en puro lenguaje ensamblador, ahora existen herramientas donde la mayor parte de la creación del juego se oculta tras una capa de alto nivel mucho más intuitiva. Esas herramientas son los motores gráficos y los *frameworks*. Uno de los más populares es el *framework* XNA, creado y actualmente descontinuado por Microsoft.

El objetivo de este trabajo es la creación de una librería JavaScript que imite la funcionalidad ofrecida por XNA para el desarrollo de videojuegos 2D, todo ello interactuará con el elemento HTML5 Canvas. Se ha elegido JavaScript por ser uno de los lenguajes más flexibles y populares actualmente, por su continua actualización y el inmenso contenido creado por los usuarios. Gracias a sus características, permite ejecutar juegos en cualquier navegador moderno, esto hace posible la ejecución de los juegos también en *smartphones*.

La librería debe ser capaz de renderizar imágenes en Canvas, cargar recursos en memoria, reproducir sonidos, gestionar animaciones y capturar eventos de teclado y ratón. Adicionalmente, para poder crear juegos más completos, deberá dar soporte para físicas 2D (gravedad, colisiones, etc.), gestionar la cámara de juego y manejar elementos geométricos y magnitudes físicas (puntos, vectores, etc.).

En el presente documento se analizarán las clases y elementos programados en JavaScript que permiten crear videojuegos con toda la funcionalidad anteriormente expuesta, así como las pruebas realizadas para la verificación del correcto funcionamiento de la librería y las demos incluidas que muestran pequeños juegos de ejemplo creados con la misma.

Palabras clave

Librería, *framework*, JavaScript, XNA, videojuegos, Canvas.

Abstract

Game development has been simplified so much in the last 25 years, they were programmed in pure assembly language, now there are tools where most of the creation of the videogame is hidden behind a high-level and much more intuitive layer. These tools are game engines and frameworks. One of the most popular is the XNA framework, created and currently discontinued by Microsoft.

The target of this work is the creation of a JavaScript library which imitates the functionality offered by XNA, for the creation of 2D games, the library will interact with the HTML5 element Canvas. JavaScript has been chosen because it's one of the most flexible and popular programming languages, because of his regular updating and for the huge content created by users. Thanks to its characteristics, it allows executing games in any modern web browser, this makes posible the execution of games in smartphones too.

The library must be able to render images on Canvas, load resources in memory, play sounds, manage animations and capture keyboard and mouse events. Aditionally, for the purpose of create more complete games, the library must offer support for 2D physics (gravity, collision detection, etc.), manage the game camera, and manage geometric elements and physic magnitudes (points, vectors, etc.).

In this document we will analyze the JavaScript programmed classes and elements that allow to create games with all the functionality previously exposed, as well as the tests done to verify the correct behaviour of the library and the demos included which show little game examples made with the library.

Key words

Library, framework, JavaScript, XNA, videogames, Canvas.

Índice de contenido

Índice de figuras

Glosario

2D y 3D: Dos y tres dimensiones respectivamente.

Android: Sistema operativo diseñado para dispositivos móviles.

API: Interfaz de programación de aplicaciones.

Autodesk Maya: Software de animación 3D.

Blender: Software *open source* de animación y creación de gráficos 3D

C++: Lenguaje de programación extendido a partir de C.

C#: Lenguaje de programación desarrollado por Microsoft que deriva de C/C++.

Canvas: Elemento HTML5 utilizado para pintar gráficos vía JavaScript.

Construct 2: Motor gráfico sencillo para creación de juegos 2D.

Drag & Drop: Se refiere a la acción de mover con el ratón objetos en pantalla.

Flag: En programación se refiere a bits usados para almacenar un valor binario.

Flash: Tecnología para crear animaciones gráficas vectoriales.

Framework: Tecnología de soporte definido con módulos de software concretos.

Game Maker Studio: Motor gráfico para creación de juegos 2D y 3D.

GPU: Unidad de procesamiento gráfico, del inglés *Graphic Processing Unit*.

***Hack'n slash**:* Género de videojuegos basados en los combates.

HTML5: Quinta revisión importante del lenguaje básico de la *World Wide Web*.

iOS: Sistema operativo diseñado por Apple para dispositivos móviles.

JavaScript: Lenguaje de programación interpretado, dialecto de ECMAScript

Juegos triple A o AAA: Videojuegos de gran presupuesto y popularidad.

Motor gráfico: rutinas de programación utilizadas para el desarrollo de videojuegos.

Open GL: Estándar que define una API para desarrollo de aplicaciones 2D y 3D

Open source: Hace referencia a *software* donde el código fuente es público.

Plugins: aplicación relacionada con otra que aporta una función nueva específica.

Renderizar: Proceso de generar una imagen o vídeo por cálculo computacional.

Shooter: Género de videojuegos basados en disparos en primera persona.

Smartphone: Teléfono construido sobre una plataforma informática móvil.

Sprite: Pequeño mapa de bits que se dibuja en una pantalla.

RPG: Género de videojuegos que usa elementos de los juegos de rol.

Tiled: Editor sencillo y flexible para crear mapas para juegos 2D basados en *tiles*.

Tilemaps: Conjunto de gráficos 2D utilizado para juegos 2D normalmente.

Videojuegos *indie*: Videojuegos creados por un individuo o pequeños grupos, sin apoyo financiero de distribuidores.

WebGL: estándar para mostrar gráficos en 3D en navegadores web.

1. Introducción

Hoy en día, con el gran avance de las nuevas tecnologías es mucho más sencillo desarrollar videojuegos que hace unos años atrás. La aparición de herramientas, como los motores gráficos Unity o el motor Unreal Engine, ha permitido que cualquiera con ganas y motivación pueda desarrollar videojuegos de diversa complejidad, a veces incluso sin tener que escribir ni una sola línea de código gracias a herramientas como Game Maker Studio o Construct 2.

Sin embargo, el desarrollo de videojuegos con estas herramientas, a pesar de simplificar en gran medida todo el proceso, no llegan a aportar una base muy sólida sobre cómo funcionan por dentro los videojuegos, ya que ocultan muchas de las mecánicas básicas de éstos. Esto es muy influyente a la hora de optimizar el rendimiento del videojuego, ya que normalmente, al no conocer el código autogenerado que hay por detrás, la optimización es complicada si la máquina tiene que gestionar y renderizar muchos elementos en pantalla.

Por supuesto no solo se han desarrollado herramientas de desarrollo de videojuegos con intuitivas interfaces gráficas donde el 70% del proyecto se realiza con sencillos métodos *Drag & Drop*, también existen una gran cantidad de librerías de diversa índole (enfocadas a físicas, a renderizado de modelos 3D, etc.) y *frameworks*, muchos de código libre, que ayudan a programar videojuegos con funciones predefinidas que permiten despreocuparse de tareas tan básicas como el dibujado de figuras 2D o modelos 3D por pantalla. Uno de estos *frameworks* es XNA *Game Studio*, desarrollado por Microsoft, creado para facilitar el desarrollo de videojuegos para el sistema operativo (SO) Windows y la consola Xbox.

1.1 Motivación

Actualmente existen diversas librerías creadas o relacionadas con el desarrollo de videojuegos 2D, que facilitan su programación sobre HTML5 y JavaScript. Esto se debe a las características de flexibilidad y portabilidad que ofrece (al poder ejecutarse en cualquier navegador moderno).

Algunas de estas librerías tienen una sintaxis y arquitectura de diseño similar a la de otras herramientas previas como por ejemplo Flash, lo que hace que sea fácil portar un videojuego desarrollado sobre la herramienta anteriormente mencionada a JavaScript.

Sin embargo, no hay ninguna librería JavaScript basada en XNA, a pesar de ser uno de los *frameworks* más utilizado en videojuegos *indie*, incluso a día de hoy, con XNA descontinuado.

1.2 Objetivos

El objetivo de este trabajo es el de desarrollar una librería JavaScript que permita una portabilidad sencilla de videojuegos 2D que hayan sido desarrollados mediante el *framework* XNA *Game Studio*, actualmente descontinuada por Microsoft. Es decir, la librería a desarrollar deberá contar al menos con las clases básicas de XNA y los métodos necesarios para poder programar un videojuego 2D sencillo, todo ello utilizando, en la medida de lo posible, la sintaxis (nombres de clases, métodos, etc.) y la estructura utilizada por XNA.

1.3 Estructura de la memoria

Este documento comenzará hablando sobre el estado del arte, en él se explicarán las diferencias entre Canvas y WebGL, se hablará de algunas de las herramientas para desarrollo de videojuegos que existen actualmente y sobre XNA.

Seguidamente entraremos en los detalles del diseño y desarrollo de la librería, explicando por qué se eligió XNA y JavaScript como lenguaje de programación, concretando la funcionalidad a implementar, hablando sobre

la orientación a objetos de JavaScript, explicando la estructura de XNA y viendo en detalle las clases que serán implementadas.

A continuación se expondrán las pruebas y demos realizadas para comprobar el correcto funcionamiento de las clases de la librería y finalmente se explicarán las conclusiones y las referencias que han ayudado al análisis y desarrollo del trabajo.

2. Estado del arte

Elegir cómo se va a desarrollar y para qué plataformas es una pregunta que todo diseñador de videojuegos se plantea al comenzar un nuevo proyecto. Para responder a esta pregunta se deben tener en cuenta también las necesidades de rendimiento, el público al que va dirigido el videojuego, los requerimientos de cada plataforma, etc. Dependiendo de la respuesta los programadores deberán utilizar un lenguaje de programación u otro, ya que no es lo mismo desarrollar un videojuego para Android que para iOS o PC.

Sin embargo, mediante HTML5 y JavaScript podemos crear videojuegos en cualquier sistema operativo que disponga de un navegador web. Esto hace que el videojuego desarrollado sea multiplataforma y portable, sin necesidad de que el código tenga que compilarse. Todo ello unido a la flexibilidad que brinda JavaScript a la hora de programar hace que sea muy cómodo desarrollar videojuegos.

2.1 Canvas vs WebGL

Es posible utilizar 2 tecnologías diferentes de renderizado a la hora de mostrar imágenes o gráficos en el navegador mediante HTML5, estas son Canvas y WebGL. En esta sección primero se definirán los dos términos y seguidamente se verá para qué se usa principalmente cada una de estas tecnologías.

Canvas

Es un elemento de HTML5 que permite renderizar imágenes y formas 2D de forma dinámica. Consiste en una región dibujable previamente definida en el código HTML, tiene atributos para especificar la anchura y el alto. Desde código JavaScript se puede acceder a esta región dibujable permitiendo el dibujado de gráficos de forma dinámica.

WebGL

Es una especificación estándar que actualmente está siendo desarrollada con el objetivo de mostrar gráficos y formas 3D acelerados por hardware (GPU) en navegadores web modernos. Hablando de forma técnica, WebGL sería una API para la implementación nativa de OpenGL 2.0 en navegadores. Herramientas como Blender o Autodesk Maya son capaces de crear y exportar, sin necesidad de programación, escenas y modelos para WebGL.

Conclusiones

Como se puede ver en las definiciones anteriores, la mayor diferencia entre Canvas y WebGL es básicamente que el primero ha sido creado principalmente para renderizar gráficos en 2D y el segundo gráficos en 3D. La complejidad intrínseca de los gráficos 3D hace que Canvas sea más fácil de utilizar y gestionar que WebGL. Sin embargo, al trabajar directamente con la GPU, WebGL cuenta con una eficiencia bastante aceptable (todo ello contando con que es un estándar cuya especificación y desarrollo no está acabado) y superior a la del Canvas.

En conclusión, no hay nada que se pueda hacer en Canvas que no se pueda hacer con WebGL, sin embargo, para la creación de juegos sencillos 2D sin demasiada carga gráfica es recomendable (al menos a día de hoy) utilizar Canvas, por su sencillez, su mejor soporte en navegadores y sobretodo en *smartphones*, pues WebGL aún no llega al nivel de soporte de Canvas en estos dispositivos.

2.2 Herramientas para desarrollar videojuegos

En este apartado se estudiarán las principales herramientas (motores gráficos y *frameworks*) así como algunas de las librerías JavaScript y HTML5 más utilizadas para el desarrollo de videojuegos y renderizado de gráficos en 2D y 3D. Al comienzo de cada subapartado se definirán los términos anteriormente mencionados.

2.2.1 Motores gráficos

Un motor gráfico está constituido de una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. Para ello, el motor provee como funcionalidad básica el renderizado de gráficos 2D y 3D, motor de físicas (aquí está incluida la detección de colisiones), gestión de sonidos, *scripting*, animación de objetos, gestión de redes, inteligencia artificial, administración de memoria y edición de escenarios. Por sí mismos, los motores gráficos son una solución integrada y completa para gestionar todos los sistemas o subsistemas de un juego.

En esencia, sirven para que los desarrolladores no tengan que "reinventar la rueda", de esta manera pueden ahorrar tiempo y esfuerzo en la creación de la funcionalidad básica de un videojuego y centrarse así en las mecánicas, la lógica y las características específicas del mismo.

A continuación, a modo de ejemplo se expondrán brevemente 3 de los motores gráficos más influyentes y utilizados en la industria actual de los videojuegos y se analizarán brevemente sus principales características:

CryEngine es un motor desarrollado por CryTek. Actualmente está en su versión tercera y es capaz de exportar juegos para 8 plataformas diferentes: MS Windows, Playstation 3, Playstation 4, Xbox 360, Xbox One, Wii U, Android e iOS. Tuvo un gran éxito principalmente con el lanzamiento de las sagas de videojuegos Crysis y Far Cry, debido a la calidad de sus gráficos y sus físicas de gran realismo.
CryEngine puede adquirirse para desarrollar videojuegos por aproximadamente 10$ al mes.

Unity es un motor creado por Unity Technologies. Está en su quinta versión y puede exportar videojuegos para 21 plataformas diferentes, lo que lo convierte en uno de los motores gráficos más versátiles. Entre estas plataformas están incluidas MS Windows, Playstation 3, Playstation 4, Xbox 360, Xbox One, Wii U, Android, iOS, Oculus Rift y Tizen, entre otras.

Alcanzó un gran éxito gracias a que por primera vez se ofrecían una gran cantidad de herramientas para desarrollar videojuegos en un solo programa y a un precio asequible, lo que propicio la creación de multitud de estudios independientes.

Es posible descargarlo y utilizarlo de forma gratuita sin algunas de sus características, su versión profesional cuesta 1500$ o 75$ al mes.

Unreal Engine es un motor creado por Epic Games. Se encuentra en su cuarta versión y se pueden exportar videojuegos para muchas plataformas diferentes, entre las que se encuentran MS Windows, GNU/Linux, Mac OS/OS X, Dreamcast, Gamecube, Wii, Xbox, Xbox 360, PlayStation 2, PlayStation 3, Xbox One y PlayStation 4. Algunos de los juego más conocidos desarrollados con Unreal Engine son los de las sagas BioShock, Batman: Arkham, Mass Effect o Gears of War entre otros.

Este motor actualmente es gratuito, sin embargo, después de ganar más de 3000$ con el lanzamiento de un juego desarrollado bajo este motor se llevarán un 5% de los beneficios.

2.2.2 *Frameworks* y librerías

A menudo la gente confunde el término *framework* con el término librería y viceversa, pero no son lo mismo. Antes de continuar es importante aclarar sus diferencias.

Diferencias entre *framework* y librería:

Indagando en internet resulta abrumadora la cantidad de librerías que son llamadas *frameworks* y la cantidad de éstos que son definidos por otros usuarios como librerías. Es difícil encontrar respuestas claras sobre este tema, por lo que se intentará explicar con la mayor claridad posible.

Un *framework* es un entorno de desarrollo completo que cuenta con un conjunto de librerías y aporta una estructura completa donde el usuario "encaja" su código. Impone unas condiciones a nuestra aplicación e incluso puede definir su arquitectura.

Una librería es un módulo *software* o conjunto de funciones/métodos que suministran una funcionalidad concreta para simplificar tareas complejas. Podemos utilizarlas desde nuestro código sin necesidad de adaptar/modificar nuestra estructura de aplicación.

En definitiva y simplificando mucho, nuestro código llama a las funciones de una librería mientras que un *framework* llamará a nuestro código, que sería algo así como piezas integradas en su estructura. En el caso de las librerías, el programador es el que utiliza las funciones de la librería cuando y como quiere, en el caso de los *frameworks* el programador debe encajar su código en su estructura.

Habiendo aclarado estos términos, podemos encontrar por internet un sinfín de librerías y *frameworks* sobre JavaScript que pueden ayudar al desarrollo de videojuegos en menor o mayor medida. Tienen diversas funcionalidades, algunos sirven para renderizado de modelos 3D, otros proporcionan físicas 2D realistas, etc. A continuación se enumerarán y describirán brevemente

algunos *frameworks* y librerías JavaScript con funcionalidad útil para el desarrollo de videojuegos:

Frameworks

- **Phaser** – es un *framework* de código libre para Canvas y WebGL basado en Flash. Cuenta con motor de físicas, gestión de *sprites*, partículas, gestión de sonidos, compatibilidad con *tilemaps*, etc.
- **LimeJS** – es un *framework* de código libre para HTML5 que puede ser utilizado para crear juegos tanto en pantallas táctiles como en navegadores.
- **BabylonJS** – es un motor 3D que funciona sobre WebGL. Tiene motor de físicas, de colisiones, de animaciones, soporta distintos tipos de cámaras, tiene partículas, soporte para diferentes tipos de texturas, etc. Ha sido desarrollado por trabajadores de Microsoft.
- **Pixi.js** – es un eficiente motor de renderizado 2D compatible con WebGL y Canvas. Es multiplataforma, soporta hojas de *sprites* (*sprite sheets*) y ofrece soporte para móviles y tabletas, entre otras características.
- **Canvas Engine** – es un motor para desarrollar juegos en el Canvas de HTML5. Es capaz de manejar juegos multijugador y funciona en navegadores y *smartphones*. Es compatible con el editor de mapas Tiled, tiene motor de colisiones, gestión de sonidos, animaciones, etc.

Librerías

- **MelonJS** – es una librería de código abierto basada en HTML5 para desarrollo de juegos 2D. Tiene gestión de animaciones, partículas, sonidos, efectos de transición, etc. Además es compatible con mapas generados mediante la herramienta Tiled.
- **Box2D JS** – es una portabilidad a JavaScript de la librería Box2D, desarrollada en C++. Esta librería ofrece una completa funcionalidad

para simulaciones físicas 2D de cuerpos rígidos. Incluye colisiones, aceleración, gravedad, fricción, polígonos convexos, etc.

- **Three.js** – es una librería de renderizado de modelos 3D sobre WebGL. Soporta diferentes tipos de cámaras, iluminaciones, texturas, animaciones, etc. Es la librería JavaScript más utilizada para mostrar gráficos 3D en HTML5.
- **Enchant.js** – es un motor simple de código abierto hecho en JavaScript para crear juegos y aplicaciones en HTML5 + JavaScript. Es multiplataforma, orientado a objetos y eventos, soporta WebGL, tiene motor de animación y cuenta con su propia biblioteca de imágenes.
- **Jaws** – es una sencilla librería para desarrollo de juegos 2D en el Canvas de HTML5. Soporta *sprites*, gestiona los estados del juego, y es capaz de precargar todas las imágenes antes del comienzo del juego.

2.3 XNA: el *framework* de *Microsoft*

XNA engloba un conjunto de herramientas que facilitan el desarrollo de videojuegos para Windows, Windows Phone y el sistema de Xbox 360 en un entorno administrado. El lenguaje utilizado para programar en XNA es C#.

XNA integra herramientas para la creación de contenidos tales como la plataforma para creación de audio (XACT). Proporciona apoyo para la creación de juegos tanto 2D como 3D y permite el uso de los controladores de Xbox 360. Microsoft proporciona XNA *Game Studio*, un entorno de desarrollo integrado (IDE). Actualmente está descontinuado, es decir, no tienen previsto seguir ampliándolo ni actualizándolo. Su última versión es la 4, la cual permitió desarrollar juegos y aplicaciones para el sistema Windows Phone 7 aparte de tener un mercado en línea propio. La versión 4.0.4, lanzada en 2014 hacía compatible XNA con el entorno *Visual Studio 2013*.

3. Diseño y desarrollo

Desarrollar una librería para facilitar la programación de videojuegos es un proyecto que podría alargarse casi hasta el infinito, el límite de la funcionalidad que se desea implementar está en la imaginación del propio desarrollador. Como hemos visto en el capítulo anterior, hoy en día se han creado algunos motores gráficos de carácter general, es decir, no están enfocados en ningún tipo de juego concreto, sino en todos, pues su objetivo principal es que cualquier persona pueda utilizarlo para crear juegos de todo tipo (plataformas, *hack'n slash*, *shooter*, RPG, etc.) y para cualquier plataforma moderna (idealmente).

Sin embargo, normalmente las compañías y empresas desarrolladoras de videojuegos crean motores gráficos propios que satisfacen exclusivamente las necesidades de los videojuegos que van a desarrollar, de tal manera que si el proyecto consiste en desarrollar un videojuego de tipo *shooter*, la empresa desarrollará un motor gráfico o un *framework* con funcionalidad enfocada a ese tipo de juego, y será tarea difícil hacer juegos de otro tipo con esa herramienta. De igual manera, en este trabajo se implementará funcionalidad adicional a la de XNA enfocada al desarrollo de videojuegos de plataformas en dos dimensiones. Más adelante, en este capítulo se concretarán las clases implementadas en la librería. De hecho, es posible que debamos hablar del desarrollo de un *framework* y no de una librería, puesto que realmente se recreará parte del *framework* de XNA en JavaScript.

En este capítulo se hablará de por qué se ha elegido XNA y JavaScript y se analizará la funcionalidad básica y adicional que debe tener un videojuego de plataformas. Seguidamente, se estudiará la orientación a objetos en JavaScript, la estructura de XNA y se detallarán las clases y métodos implementados.

3.1 ¿Por qué XNA y por qué JavaScript?

Actualmente, XNA es un *framework* descontinuado, es decir, la empresa desarrolladora (Microsoft) no planea seguir ampliando, mejorando ni actualizando la plataforma. Sin embargo, aún sigue siendo uno de los *frameworks* más utilizados por desarrolladores *indies* y gente que quiere iniciarse en el mundo de los videojuegos, sobre todo si desarrollan para el sistema operativo Windows.

Por esta razón es interesante crear un sucesor con el que la gente pueda programar para dispositivos modernos. Aquí es donde entra en juego JavaScript, ya que su gran ventaja es que puede ser ejecutado cómodamente en cualquier navegador moderno que soporte HTML5, que es un estándar en el que cada día se está trabajando para mejorar sus características. Aparte del soporte continuo del lenguaje, se puede apreciar otra ventaja, es un lenguaje que no requiere ser compilado (puesto que es un lenguaje interpretado) o escrito mediante la ayuda de un IDE, es sencillo escribir código JavaScript con un editor de texto y ejecutarlo a través de un fichero HTML y el navegador, lo cual lo hace mucho más accesible a personas con menor conocimiento en programación.

Sin embargo, no todo son ventajas. Ni JavaScript ni HTML han sido diseñados para programar videojuegos. A pesar de que en los últimos años han evolucionado notablemente y la comunidad de desarrolladores ha generado mucho contenido, la carga de gráficos que puede soportar Canvas es reducida y aunque WebGL es más potente aún no está al nivel de eficiencia de las plataformas dedicadas al desarrollo de videojuegos. Por lo tanto, a día de hoy no es posible desarrollar juegos triple A programando en HTML5 y JavaScript, pero no es una mala alternativa si el juego a desarrollar no es muy complejo y se busca que sea multiplataforma.

3.1.1 Ventajas y desventajas

A continuación se analizarán y concretarán algunas de las ventajas y desventajas de utilizar JavaScript para desarrollar videojuegos.

Ventajas

- Completa interacción con los elementos HTML y con CSS. Mediante JavaScript es posible manipular las propiedades de los objetos y actualizar, establecer o eliminar ciertos valores para cambiar su comportamiento.
- JavaScript es un lenguaje ligero, es decir, el "peso" de los archivos contenidos en un juego programado en JavaScript es casi ridículo comparado con otros, lo que hace que los juegos carguen muy rápido.
- Los videojuegos desarrollados en JavaScript funcionan en cualquier plataforma (dependiendo de su programación). Llámese plataforma a Windows Mobile, Android, GNU Linux, iOS, etc.
- No son necesarios *plugins* adicionales en el navegador para ejecutar juegos desarrollados en JavaScript, es tan simple como acceder al sitio web donde esté alojado el juego.
- JavaScript es un lenguaje multiplataforma, lo que permite "desatar" al programador de un IDE o sistema operativo concreto.

Desventajas

- Uno de los grandes problemas de los desarrolladores no tiene que ver con el desarrollo en sí, sino con la incompatibilidad actual que existe entre los navegadores. Un mismo juego puede ejecutarse correcta y eficientemente en un navegador, pero en otros no.
- Debido a que JavaScript se ejecuta del lado del cliente (en el navegador) y a que es un lenguaje interpretado, cualquiera puede acceder al código con facilidad. Existen técnicas de ofuscación del código, aunque siguen sin ser realmente seguras.
- Se consume mucha memoria, lo que hace que el navegador se quede bloqueado, especialmente al ejecutar operaciones matemáticas pesadas (como por ejemplo *Math.sqrt()*).

- WebGL todavía no es está completamente soportado en todos los navegadores.

3.2 Funcionalidad a implementar

Los juegos en dos dimensiones pertenecen principalmente al género plataformas, el cual se hizo muy famoso en los años 80 gracias a juegos como Super Mario Bros (Nintendo) o Ghost'n Goblins (Capcom), aunque hoy en día sigue siendo un género que goza de gran popularidad.

Estos videojuegos se caracterizan por tener que realizar acciones como caminar, correr, saltar o escalar sobre una serie de plataformas, con obstáculos y enemigos, mientras se recogen una serie de objetos que nos permitirán avanzar en la trama del juego hasta completarlo. Generalmente, los videojuegos de plataformas también son característicos por utilizar vistas de desplazamiento horizontal, donde se ve al protagonista y los enemigos de perfil.

La funcionalidad de la librería a desarrollar debería ser capaz de cubrir los aspectos básicos de los videojuegos de plataformas 2D. A continuación se analizará la funcionalidad básica y adicional a implementar.

3.2.1 Funcionalidad básica

La funcionalidad básica son todos aquellos aspectos que hacen que pueda ejecutarse y visualizarse un juego 2D en el Canvas del navegador, concretamente la librería debe incluir soporte para:

- Bucle de juego, es decir, un bucle infinito que permita en cada iteración (aproximadamente 60 por segundo) actualizar la lógica del juego y realizar las demás acciones que procedan.
- Cargar recursos en memoria (imágenes, animaciones y sonidos).
- Dibujar imágenes en el Canvas.
- Dibujar texto en el Canvas.
- Reproducir canciones y efectos de sonido.

- Gestión de animaciones (cargarlas, iniciarlas y pararlas).
- Interacción con el ratón.
- Interacción con el teclado.

3.2.2 Funcionalidad adicional

La funcionalidad adicional es aquella que mejora o amplía las posibilidades de la librería para dar la oportunidad al desarrollador de crear videojuegos más completos de forma más sencilla. La librería desarrollada deberá ofrecer los siguientes soportes adicionales:

- Físicas 2D básicas:
  - Gravedad
  - Detección y gestión de colisiones
  - Aceleración/Deceleración
  - Fricción
  - Rebotes
- Manejo de elementos geométricos y magnitudes físicas en 2 dimensiones (vector, punto, segmento, rectángulo y circunferencia).
- Gestión de cámara, que sea capaz de seguir al personaje por el mapa de juego.
- Elementos básicos de los juegos de plataformas (jugador, proyectil, enemigo y objetos).

3.3 JavaScript orientado a objetos

JavaScript es un lenguaje de programación interpretado, principalmente basado en prototipos, esto significa que no es un lenguaje en el que los objetos se instancien mediante una clase, sino mediante la copia o clonación de otros objetos. Esto hace que sea un lenguaje muy flexible, donde los objetos pueden ser modificados y heredar métodos y valores de otros objetos o de su propio prototipo.

Existe un debate sobre si JavaScript es o no un lenguaje orientado a objetos, como Java o C#. Se podría decir que sí, ya que aunque no cuente con definición de clases como tal, es posible imitar su funcionamiento y los objetos cuentan con las principales características de los lenguajes orientados a objetos, estas características son herencia, polimorfismo, encapsulamiento y visibilidad entre otras.

Clases, atributos y métodos

XNA está programado en C#, un lenguaje orientado a objetos desarrollado por Microsoft que deriva de los lenguajes C y C++. Uno de los objetivos de este trabajo es hacer la experiencia de programación con la librería lo más parecida posible a XNA, por lo tanto, se han intentado imitar las clases C# del *framework* en JavaScript. A continuación se puede observar parte de la clase *Game1* de XNA, que hereda de *Game* comparándola con su equivalente en JavaScript:

```
/// <summary>
/// This is the main type for your game
/// </summary>
public class Game1 : Microsoft.Xna.Framework.Game
{
    public GraphicsDeviceManager graphics;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
    }

    /// <summary>
    /// Allows the game to perform any initialization it ne
    /// This is where it can query for any required servic
    /// related content.  Calling base.Initialize will enu
    /// and initialize them as well.
    /// </summary>
    public override void Initialize()
    {
        // TODO: Add your initialization logic here

        base.Initialize();
    }
```

```
/**
 * Provides basic graphics device initialization, game logic
 */
function Game() {
    "use strict";
    /*globals ContentManager, GraphicsDevice, document, Spri
    this.Content = new ContentManager();
    this.GraphicsDevice = new GraphicsDevice(document);
    this.spriteBatch = null;
    this.gameState = GameStates.Initializing;
    this.preloading = true;
}

/**
 * Allows the game to perform any initialization it needs to
 * This is where it can query for any required services and
 * related content.
 * @name Initialize
 * @function
 * @memberOf Game
 */
Game.prototype.Initialize = function () {
    "use strict";
    var self = this;

    // TODO: Add your initialization logic here

    self.gameState = GameStates.Loading;
};
```

Figura 1: Comparación de la clase Game en C# (izq.) y JavaScript (der.)

Como se puede apreciar, en JavaScript podemos imitar la creación de clases declarándolas como funciones (*function*), estas funciones contienen los atributos, los cuales se definirán como miembros de *this* sin son públicos o como *var* si se desea que sean privados. A la hora de instanciar una clase se hace con el método común de los lenguajes orientados a objetos, utilizaremos

la sentencia *new*. Por último, los métodos de la clase se definirán como prototipos (*prototype*) de la función. Es interesante también analizar el término *self*, podemos crear una variable local dentro de los métodos e igualarla a *this*, de esta forma accederemos a los atributos de una forma más eficiente. Esto se debe a que llamar continuamente a *this* para acceder a los atributos es más costoso que tenerla almacenada en una variable local, a la cual JavaScript accede de manera más rápida.

En JavaScript es posible también crear clases estáticas (*static*) que pueden ser accedidas desde cualquier parte del código. Un ejemplo es la clase *Keyboard*, que será definida como una variable global, y dentro de ella se declararán sus atributos y métodos. En este caso el código es algo diferente al de la clase *Game*, los métodos no se declaran como miembros de un prototipo, sino que se asignan a un atributo de la variable global. A continuación se puede observar un fragmento del código:

```
/**
 * Allows retrieval of keystrokes from a keyboard input device.
 * @global
 */
var Keyboard = {

    keyDownList: [],
    keyUpList: [],
    keyPressList: [],

    /**
     * Add a key to the list of pressed down keys.
     * @name addKeyDown
     * @function
     * @param {Event} event - This parameter represents a keyboard event.
     * @memberOf Keyboard
     * @static
     */
    addKeyDown: function (event) {
        "use strict";
        var keyCode = event.keyCode,
            indexOfKey = Keyboard.keyDownList.indexOf(keyCode);

        if (indexOfKey === -1) {
            Keyboard.removeKeyUp(keyCode);
            Keyboard.keyDownList.push(keyCode);
        }

        event.preventDefault();
    },
```

Figura 2: Fragmento de código de Keyboard

También podemos definir métodos estáticos asociados a clases que no lo son, veamos un ejemplo de la clase *Point*:

```
/**
 * Returns the point (0,0).
 * @name Zero
 * @function
 * @memberOf Point
 * @static
 */
Point.Zero = function () {
    "use strict";
    return new Point(0, 0);
};
```

Figura 3: Método estático de Point

Como se puede ver, la declaración de métodos estáticos de una clase es similar a la vista anteriormente, pero no se crea como miembro del prototipo de la clase.

Llegados a este punto conviene reiterar que en JavaScript no hay clases como tal, solo objetos. Lo que imita a la clase es una función con la capacidad de crear objetos que comparten un prototipo.

Herencia

La herencia en JavaScript es herencia entre objetos. Si un objeto deriva de otro heredará todo el comportamiento y estado definidos en el objeto base. Hay varias formas de que un objeto herede de otro en JavaScript, sin embargo, veremos cómo hacerlo de una forma parecida a la de un lenguaje con clases. Por ejemplo, queremos que la clase *AnimatedSprite* herede de *Sprite*, se haría de la siguiente manera:

```
/**
 * Represents an animated sprite, which is composed of various frames.
 * @constructor
 * @param {Texture2D} texture - The texture image.
 * @param {Vector2} scale - The scale of the image.
 * @param {Rectangle} sourceRect - The image source rectangle.
 * @param {Point} position - The sprite position in the canvas.
 * @param {Boolean} isVisible - Determines if the sprite is visible or not.
 * @param {Boolean} loop - Determines if the animation has to animate in loop or not (one time).
 * @param {Boolean} pingPongLoop - Determines if the animation should run in reverse when reachs the last
 *                                 frame or not (reset to initial frame).
 * @param {Number} millisecondsPerFrame - A number representing the milliseconds between each frame.
 * @param {Number} numberOfFrames - The number of frames in the sprite sheet.
 */
function AnimatedSprite(texture, scale, sourceRect, position, isVisible, animationType, numberOfFrames, millisecondsPerFrame) {
    "use strict";
    /*global Sprite, AnimationTypes, setInterval, clearInterval*/

    //Call the father constructor (super)
    Sprite.call(this, texture, scale, sourceRect, position, isVisible);

    this.AnimationType = animationType;
    this.NumberOfFrames = numberOfFrames;
    this.MillisecondsPerFrame = millisecondsPerFrame;
    this.CurrentFrame = 0;
    this.Forward = true; //to control the direction of the loop
    this.Columns = Math.floor(texture.Width() / sourceRect.Width);
    this.RunAnimation = true;
    this.VerticalOffset = 0;    //(optional) used when the animation sheet has different
                                //animations representing different states of the object
                                //which owns the sprite
    this.interval = null;
}

//Create the AnimatedSprite.prototype object, which inherits from Sprite.prototype.
//Then stablish the "constructor" property to reference to AnimatedSprite
AnimatedSprite.prototype = Object.create(Sprite.prototype);
AnimatedSprite.prototype.constructor = AnimatedSprite;
```

Figura 4: Definición de la clase AnimatedSprite, que hereda de Sprite

Como se puede ver, antes de declarar los atributos se llama al que sería el equivalente a la función constructora de *Sprite*, mediante *Sprite.call(…)*. Luego es necesario indicar a JavaScript que el prototipo de la clase hija debe crear una instancia de la clase padre, y además asignar manualmente su constructor, para que en tiempo de ejecución sepa que tiene que llamar al de *AnimatedSprite* y no al de *Sprite*.

Polimorfismo

Una definición de polimorfismo podría ser: los objetos de una clase derivada pueden tratarse como objetos de la clase base, pero manteniendo su comportamiento distintivo. Sin embargo se introduce el término “clase”, que son un mecanismo para implementar la programación orientada a objetos, pero no son el único. Una definición más adecuada sería: el polimorfismo es la posibilidad de enviar un mismo mensaje (es decir, de invocar a un método) a varios objetos de naturaleza homogénea.

Pongamos un ejemplo, supongamos que tenemos la clase Rectángulo, que hereda de la clase Figura. Disponemos de un método DibujarFigura que es capaz de llamar al método Dibujar, implementado en ambas clases, y el argumento de DibujarFigura es de tipo Figura. Podríamos pasar un objeto Rectángulo al método DibujarFigura y este ejecutaría el método Dibujar de Rectángulo, a esto se le llama polimorfismo. Como es un concepto algo complicado veremos un fragmento de código sencillo en JavaScript que ilustra el ejemplo:

```
var Figura = {
    Dibujar: function() {
        console.log("figura::dibujar");
    }
};

var Rectangulo = Object.create(Figura);
Rectangulo.Dibujar = function() {
    console.log("rectangulo::dibujar");
};

function DibujarFigura(figura) {
    figura.Dibujar();
}

DibujarFigura(Rectangulo);
```

Figura 5: Ejemplo de polimorfismo en JavaScript

Como se puede ver, tenemos polimorfismo automático en JavaScript, de hecho, al método DibujarFigura lo único que le interesa es que el objeto pasado por argumento tenga un método Dibujar.

3.4 Estructura de XNA

La estructura es bastante sencilla. El flujo de la aplicación comienza con una clase llamada *Program*, la cual contiene un método *Main* que es el punto principal de entrada a la aplicación. Este método solo hace dos cosas, crear la instancia de la clase *Game* y ejecutar su método *Run*, que es el bucle de juego el cual será llamado aproximada e idealmente 60 veces por segundo.

```
/// <summary>
/// Este es el punto de entrada al juego, empieza ejecutandose aquí.
/// </summary>
static void Main(string[] args)
{
    using (Game1 game = new Game1()) // Crea un objeto game de la clase Game1 (nuestro juego).
    {
        game.Run(); // Lanza el juego con el método Run().
    }
}
```

Figura 6: Método Main de la clase Program (C#)

En la librería JavaScript se tendrá lo mismo, *Program* será lo primero que ejecute el navegador al cargar el código JavaScript y además se inicializarán los *listeners* del teclado y del ratón necesarios para proporcionar datos de entrada al juego mediante las clases *Keyboard* y *Mouse*. Es necesario hacerlo en la clase *Program* porque es la única que tiene instanciada la clase *Game*, la cual contiene la referencia al Canvas, esta referencia es necesaria para adjuntarle algunos de los *listeners*. Vamos a explicar esto con un ejemplo, queremos que los eventos del ratón solo se tengan en cuenta cuando se produzcan dentro del Canvas y no fuera. Sin embargo, los eventos de teclado se los adjuntaremos a toda la ventana del navegador, el objeto que hace referencia a la ventana se denomina *window*.

La clase *Game* proporciona una estructura intuitiva para la inicialización básica de los elementos del juego, la carga de recursos, implementación de la lógica del juego y su renderización en pantalla. Primero llamará al método *Initialize*, cuyo objetivo es inicializar todas las variables necesarias. A continuación se ejecutará el método *LoadContent*, cuyo cometido es realizar la carga de recursos, como pueden ser imágenes y sonidos. Una vez cargados en memoria pueden ser utilizados para crear *sprites*, efectos de sonido, etc. A continuación comenzará el bucle de juego en el que se ejecutará el método *Update* y seguidamente *Draw*. El método *Update* se encarga de ejecutar y actualizar toda la lógica del juego, el método *Draw* ejecutado a continuación renderizará en pantalla todos los elementos gráficos requeridos en sus posiciones correspondientes.

Esta es básicamente la estructura del *framework*, a continuación se muestra un diagrama de flujo del mismo resumiendo de forma visual todo lo explicado anteriormente.

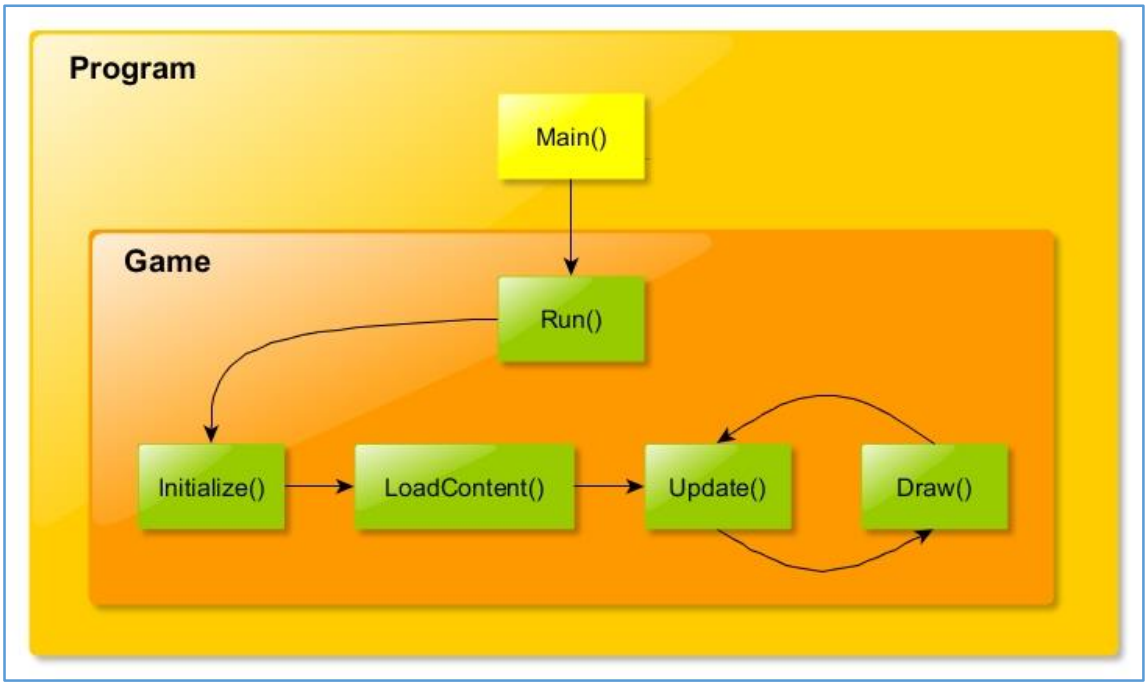


Figura 7: Flujo de ejecución básico de XNA

3.5 Clases implementadas en la librería

Como es obvio, no se puede implementar toda la funcionalidad de XNA, ya que es un *framework* lo suficientemente grande y trabajado como para ser reconstruido por una sola persona en JavaScript con todas sus características. Por ello, como ya se hablado con anterioridad en este documento se han implementado las clases básicas XNA para poder crear un juego 2D de plataformas. Además se han añadido clases adicionales que permitan cumplir con la funcionalidad básica y avanzada que fue analizada en el apartado 3.2.

En este apartado analizaremos de forma detallada y a la vez simplificada cada una de las clases implementadas en la librería JavaScript.

3.5.1 Clases de XNA adaptadas a JavaScript

Han sido 15 las clases de XNA adaptadas a JavaScript, a continuación se explicará brevemente el funcionamiento de cada una de ellas (en orden alfabético) y sus métodos asociados. Las clases que se mencionen aparecerán en negrita.

ContentManager

Se encarga de todo lo relacionado con la carga de recursos en memoria. Tiene métodos para para cargar imágenes (**Texture2D**), efectos de sonido (**SoundEffect**) y músicas (**Song**). Para ello se indica por argumento la ruta relativa del archivo y se agrega un *listener* "*load*" al elemento, esto hará que cuando el recurso haya sido cargado en memoria se ejecute una función (*loadHandler*). Dicha función se encargará de llevar la cuenta de todo lo que lleva cargado en memoria y de lo que falta, cuando todos los recursos hayan sido cargados se levantará un *flag* que lo indica. Este *flag* se utilizará para informar a la clase **Game** de que todos los recursos han sido cargados en memoria, por lo que pueden ser utilizados por el resto del código.

Game

Es probablemente la clase más importante, ya que es la que realiza el flujo básico de juego explicado en el apartado 3.4, además es la plantilla que el usuario de la librería deberá usar para comenzar a crear sus propios juegos con ella. Contiene las instancias a **ContentManager**, a **GraphicsDevice** y a **SpriteBatch**, además tiene una variable que indica el estado del juego (inicializando, cargando recursos y jugando). Como ya se ha explicado con anterioridad, esta clase contiene métodos para inicializar, cargar recursos, actualizar la lógica del juego y dibujarlo por pantalla.

En esta clase también se encuentra el método *Run*, que se ejecutará 60 veces por segundo (idealmente) actualizando y dibujando el juego haciendo uso del método *requestAnimationFrame*. Este método informa al navegador de que quieres realizar una animación y le solicita que 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, esta función es el propio método *Run*. También se utiliza un segundo argumento que especifica el elemento que limita visualmente toda la animación, en este caso, Canvas. Se ha incluido *requestAnimationFramePolyfill.js*, desarrollado por Erik Möller, que implementa una versión mejorada y compatible con todos los navegadores del método *requestAnimationFrame*.

El método *Run* quedaría de la siguiente manera:

```
/**
 * Call this method to initialize the game, begin running the game loop, and
 * start processing events for the game.
 * @name Run
 * @function
 * @memberOf Game
 */
Game.prototype.Run = function () {
    "use strict";
    /*globals requestAnimationFrame*/
    var self = this;

    //Call the game loop
    requestAnimationFrame(self.Run.bind(self), self.GraphicsDevice.canvas);

    //Choose the method to run according to the game state
    switch (self.gameState) {
    case GameStates.Initializing:
        self.Initialize();
        break;
    case GameStates.Loading:
        self.LoadContent();
        break;
    case GameStates.Playing:
        self.Update();
        self.Draw();
        break;
    }
};
```

Figura 8: Método Run de la clase Game

GraphicsDevice

Implementa una instancia al Canvas y maneja su configuración. Define el tamaño por defecto del Canvas (800x600) y el tamaño actual, además implementa métodos para establecer el tamaño del Canvas al tamaño de la ventana del navegador e incluso a pantalla completa. También contiene el método que se utilizará antes de repintar el Canvas en cada iteración, *Clear*, el cual limpia el Canvas dejándolo en blanco. Esto es necesario debido a que si no se hiciera el Canvas pintaría continuamente encima de las imágenes que hubiera anteriormente.

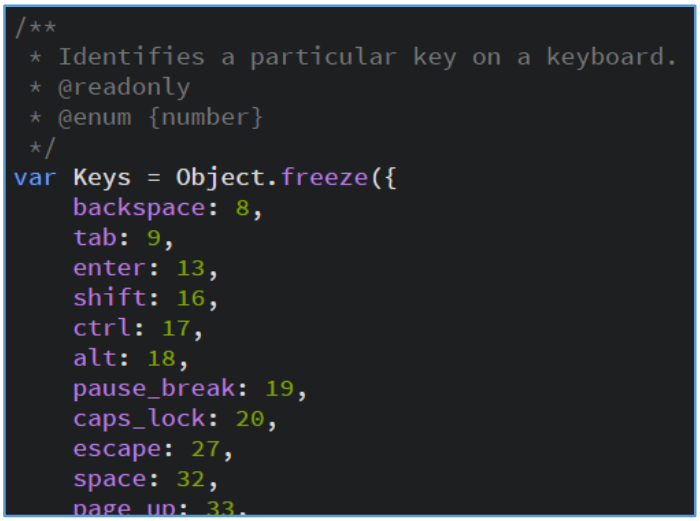
Keyboard

Esta clase estática mantiene un registro de las teclas pulsadas en el teclado. Es capaz de comprobar si una tecla concreta está siendo pulsada o si se ha levantado la pulsación mediante los métodos implementados. En la clase

Program se adjuntan *listeners* a los eventos "*keydown*" y "*keyup*" y se manejan con funciones implementadas dentro de la clase **Keyboard**.

Keys

Esto es una enumeración, contiene todos los códigos numéricos de las teclas para referenciarlos de forma intuitiva en el código. Una forma de imitar las enumeraciones en JavaScript es creando variables globales y asignándole unos objetos mediante *Object.freeze*. A continuación podemos ver un fragmento de la enumeración.



```
/**
 * Identifies a particular key on a keyboard.
 * @readonly
 * @enum {number}
 */
var Keys = Object.freeze({
    backspace: 8,
    tab: 9,
    enter: 13,
    shift: 16,
    ctrl: 17,
    alt: 18,
    pause_break: 19,
    caps_lock: 20,
    escape: 27,
    space: 32,
    page_up: 33,
```

Figura 9: Fragmento de código de la enumeración Keys

Mouse

Mantiene un registro de los botones pulsados en el ratón (izquierdo, central y derecho), la posición en coordenadas X Y del mismo en el Canvas y el valor de la rueda del ratón. Para ello, al igual que el teclado se han asignado *listeners* a los eventos "mousedown", "mouseup", "mousemove", "wheel" y "contextmenu" en la clase **Program**.

Point

Define un punto en el espacio 2D del Canvas, sus atributos son las coordenadas X e Y del punto. Contiene métodos estáticos para obtener el punto cero (0,0), el uno (1,1) y para comprobar si dos instancias de esta misma clase son iguales o no. Es utilizado a menudo para mantener la localización de cualquier otro objeto del juego.

Program

Es el punto de entrada principal al juego. Esta clase crea una instancia de la clase **Game** y ejecuta su método *Run*, que, como ya se ha explicado con anterioridad, es el bucle de juego. También agrega los *listeners* necesarios para gestionar los eventos de teclado y ratón pertinentes.

Rectangle

Define el objeto rectángulo, el cual mantiene unos atributos donde se especifica su posición en el Canvas y el ancho y alto del rectángulo. Contiene métodos para obtener la localización en forma de **Point**, el centro del rectángulo, las coordenadas de cada uno de sus lados, la mitad de su ancho y alto, comprobar si el rectángulo está vacío, comprobar si un punto específico está contenido dentro del rectángulo y aumentar el ancho y alto del rectángulo. Por último, se implementa un método capaz de comprobar la intersección del rectángulo con otro dado. Esta función devolverá *false* si no hay colisión o un **Vector2** que indica la superposición que hay entre los dos rectángulos si han colisionado.

Los rectángulos son ampliamente usados en videojuegos para comprobar colisiones entre objetos 2D de forma sencilla y poco costosa para la CPU.

Song

Representa un recurso de audio como canción, suelen utilizarse como fondo sonoro mientras el juego avanza. Esta clase implementa métodos para conocer la duración de la canción, para cambiar y/o saber si está silenciada o reproduciéndose en bucle, obtener la posición temporal de la canción en un momento dado y cambiar u obtener el valor del volumen. También implementa métodos para controlar su reproducción, es decir, se puede iniciar la canción, pausarla, reanudarla o pararla del todo. Los formatos de audio soportados actualmente por Google Chrome son: WAVE, WebM, Ogg, MP3 y MP4.

SoundEffect

Representa un recurso de audio como efecto de sonido, es decir, sonidos momentáneos y cortos que se reproducen debido a un evento que haya ocurrido en el juego. Por ejemplo, al destruir una nave sonará una explosión. Esta clase implementa los métodos necesarios para reproducir el efecto de sonido y para saber su duración. Los formatos de audio soportados son los mismos que los de la clase **Song**.

SpriteBatch

Esta clase contiene el contexto 2D del Canvas, necesario para dibujar las imágenes dentro de él. Implementa todos los métodos relacionados con el pintado en el Canvas, a continuación se listan de forma más detallada.

- Pintado de imágenes (**Texture2D**): se indicará la posición de la imagen, el rectángulo de imagen que debe recortar, la rotación, la escala y los posibles efectos (girar horizontal o verticalmente la imagen).
- Pintado de texto: se especificará el tipo de fuente, el texto a escribir, a posición, el color del texto y su escala.

- Pintado de rectángulo: este método pintará un rectángulo especificado por argumento con un grosor y color dados. Muy útil para comprobar de forma visual los rectángulos de colisión de los objetos.
- Pintado de segmento: este método dibujará una línea recta dado un segmento (**Segment**) con un grosor y color dados.

SpriteEffects

Es una enumeración de los posibles efectos que pueden asignarse a un *sprite*, estos por defecto no tendrán ninguno, pero puede cambiarse para que el *sprite* se dibuje volteado vertical u horizontalmente.

Texture2D

Esta clase representa cualquier textura o imagen que vaya a ser utilizada en el juego. Implementa métodos para obtener el ancho y alto de la imagen y el rectángulo en el que está contenida. También se puede obtener directamente la mitad de la anchura o de la altura de la imagen. Esta clase será utilizada por **Sprite** como su principal atributo.

Vector2

Define un vector 2D con coordenadas X e Y. Incluye métodos estáticos para obtener el vector cero (0,0), el vector uno (1,1), para realizar operaciones matemáticas con vectores (suma, resta, multiplicación y división), para calcular la distancia entre 2 vectores y para comprobar si son iguales. También implementa métodos no estáticos para conocer la longitud del vector y para obtener el vector normalizado o unitario.

3.5.2 Clases adicionales de la librería

Aquí se verán las clases adicionales implementadas que no están en XNA, pero que pueden servir de ayuda para programar videojuegos 2D. Algunas de estas clases están incompletas para que el usuario de la librería pueda personalizarlas e incluir su propia lógica.

AnimatedSprite

Esta clase representa un *sprite* animado, hereda de la clase **Sprite**. Este tipo de *sprites* están compuestos por varias imágenes, el dibujado de esas imágenes una detrás de otra a intervalos de tiempo regulares genera el efecto de animación. Es necesario indicar el número de imágenes que componen el *sprite*, el tipo de animación y los milisegundos que pasarán antes de que la animación se actualice a la siguiente imagen, además del resto de argumentos necesarios para generar el *sprite*, estos se explicarán más adelante. Contiene métodos para iniciar y parar la animación, esto se consigue mediante el método *setInterval*, que recibe dos argumentos, el primero es una función que se ejecutará pasados X milisegundos, indicados en el segundo argumento. Así, la función seleccionará la siguiente imagen de la animación mediante un rectángulo posicionado en las medidas exactas para escoger la siguiente imagen. Es algo complejo explicar esto solo con palabras, así que veremos un pequeño ejemplo donde se muestra una hoja de animación de un personaje y el rectángulo origen.

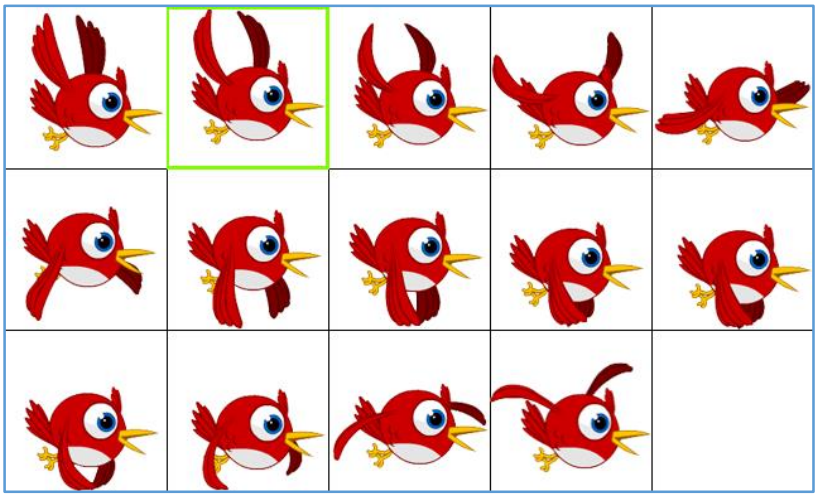


Figura 10: Hoja de animación de ejemplo

En la imagen podemos observar que la animación está compuesta por 14 imágenes. El rectángulo (en color verde) estaría ahora seleccionando la imagen 1 (empezando desde 0) y al pasar el intervalo de tiempo avanzaría a la imagen 2. El propio método que actualiza la animación avanzará a la segunda fila automáticamente cuando llegue al final de la primera. Para parar la animación simplemente se utiliza la función *clearInterval* pasando como argumento la variable que almacenó el retorno de la función *setInterval* al iniciar la animación.

AnimationTypes

Enumeración que define los 3 tipos de animación contemplados en esta librería:

- *PlayOneTime*: Ejecutará la animación una única vez, esto se utiliza para animaciones que solo deban ocurrir en un instante concreto, como por ejemplo una explosión.
- *Loop*: La animación se ejecutará desde la imagen 0 hasta la última y luego volverá al principio, es decir, se reproducirá la animación íntegra en bucle.
- *PingPongLoop*: Similar a la anterior con una diferencia, al llegar a la última imagen de la animación no volverá a la imagen 0, sino que volverá hacia atrás. Es decir, si tenemos una animación con 10 imágenes, al llegar a la 10 volverá a la 9, luego a la 8… Al llegar a la imagen 0 volverá a ir hacia delante.

Estos tipos de animación son utilizados por el método *UpdateAnimation* de la clase **AnimatedSprite** para animar el *sprite* de una forma u otra.

Ball

Esta clase representa un objeto bola o pelota, emulando sus características físicas básicas, como son la gravedad, el rozamiento del suelo y los rebotes con las paredes y con otras bolas. Es posible también arrastrar la bola con el ratón y soltarla en cualquier punto del Canvas (*Drag & Drop*). Los ajustes en las constantes físicas de la clase están definidos en la misma para ofrecer una experiencia realista, no obstante es posible modificarlos a gusto del usuario.

Camera

Uno de los elementos más importantes en cualquier videojuego es la cámara, normalmente mantiene al jugador en el centro de la pantalla mientras este explora el mapa. El tamaño de la cámara es el mismo que el del Canvas, y es posible hacer *zoom*, acercándola o alejándola mediante métodos creados en la clase.

En los videojuegos normalmente el personaje no llega hasta el final de la cámara para explorar el mapa, sino que se define un rectángulo más pequeño no visible que una vez alcanzado por el personaje hará que la cámara se desplace. De este modo, la experiencia de juego es más cómoda ya que se evitan movimientos innecesarios de la cámara y el jugador puede ver más parte del mapa sin que el personaje esté realmente posicionado ahí. Sin embargo, estos límites tienen menos prioridad que los límites del mapa, por lo que si la cámara ha alcanzado el final del mapa no avanzará más, aunque el personaje haya pasado el rectángulo límite establecido. La clase tiene fijados los límites de este rectángulo a un 25% del rectángulo del Canvas, sin embargo, estos valores pueden ser modificados.
A continuación se verá un ejemplo que representa el rectángulo interno de la cámara:

Figura 11: Rectángulo interno de la cámara

Esta imagen muestra lo que mostraría el Canvas, la cámara está posicionada en los límites izquierdo e inferior del mapa, por lo que a pesar de que el personaje está por debajo de los límites del rectángulo interno, la cámara no se mueve.

Circle

Representación geométrica de una circunferencia, clase de la cual está formada la clase **Ball**. Define el radio de la circunferencia y su posición. Tiene métodos para obtener su centro, las coordenadas del rectángulo exterior que lo encerraría, para comprobar si un punto (**Point**) está contenido en la circunferencia, si una circunferencia está dentro de la otra, si una circunferencia está colisionando con otra o si está colisionando con una línea o segmento (**Segment**). También es posible "inflar" la circunferencia, es decir, aumentar o reducir su radio.

Enemy

Representa un personaje enemigo en el juego. Los enemigos suelen cambiar de comportamiento dependiendo de los eventos que ocurran, la clase está preparada para cambiar el estado del enemigo pasado cierto tiempo. Al igual que otras, esta clase sirve como plantilla para que el usuario la personalice a su gusto. Básicamente cuenta con un método *Update* para actualizar su comportamiento y de un método para obtener su rectángulo de colisión.

GameStates

Es una enumeración de los posibles estados del juego, de forma predeterminada se incluyen los 3 básicos: inicialización (*Initializing*), cargado de recursos (*Loading*) y juego ejecutándose (*Playing*). Esta enumeración será utilizada por el método *Run* de la clase **Game** para saber que método debe ejecutar en cada iteración.

GravityDirections

Enumeración que define las 4 direcciones básicas de la gravedad: arriba, abajo, izquierda y derecha. Será utilizado por los elementos que hagan uso de la gravedad, como por ejemplo la clase **Ball**.

PickableObject

Representación sencilla de un objeto de juego recogible por el jugador, como por ejemplo vidas o habilidades nuevas. Consta de un *sprite* y un tipo de objeto, estos tipos no están definidos, será el usuario quien los defina. Tiene un método para obtener el rectángulo de colisión del objeto.

Player

Representa el personaje manejable por el usuario, sus atributos y métodos son prácticamente iguales que **Enemy**, el usuario deberá definir el comportamiento del mismo.

Polygon

Representa un polígono formado por cualquier número de segmentos (**Segment**). Dispone de un método para obtener los vértices del polígono.

Projectile

Representa un objeto proyectil, como por ejemplo un misil o una bala. Está formado por su *sprite*, el poder del proyectil, su velocidad relativa y el vector de velocidad. Cuenta también con un método *Update* donde el usuario de la librería deberá incluir su lógica.

Segment

Representa una línea recta o segmento, formado de dos puntos (**Point**). Tiene métodos para calcular la distancia del segmento y para obtener el vector dirección del mismo.

Sprite

Clase formada por una imagen (**Texture2D**) y otras propiedades, como la escala de la imagen, su posición, si es visible o no, efectos del *sprite*, su rotación y su rectángulo origen, el cual será utilizado para recortar la imagen. Contiene métodos para calcular su ancho y alto, su centro, las coordenadas de cada lado del rectángulo que encierra el *sprite* y su rectángulo de colisión. Esta es la clase padre de **AnimatedSprite** y una de las más utilizadas en videojuegos.

4. Pruebas

Todo sistema informático requiere de un banco de pruebas que verifique la funcionalidad implementada. En el caso de esta librería, se han creado varios entornos de pruebas interactivos para comprobar que todos los métodos implementados de las clases XNA funcionan a la perfección. Las clases adicionales se probarán mediante dos demos interactivas diferentes que representan el uso conjunto de toda la librería, por lo que bastarán para comprobar la funcionalidad básica y adicional especificada en el apartado 3.2.

4.1 Pruebas unitarias de las clases XNA

En las pruebas unitarias se comprueba la lógica y la correcta funcionalidad de los módulos que forman la librería. Se han creado 4 entornos diferenciados que agrupan las clases con funcionalidades relacionadas, de tal manera que puedan probarse varias en el mismo entorno. Algunas de las clases y métodos se prueban de forma implícita con todos los entornos o con algunos de ellos, principalmente las clases **Program** y **Game**, pues son la entrada inevitable hacia la aplicación. El resto de clases no especificadas se prueban de una u otra manera en los entornos.

A continuación se explican los 4 entornos de pruebas creados.

Test de Point, Vector2 y Rectangle

Aquí se prueban todos los métodos y la funcionalidad de estas clases, el código hace comparaciones internas comprobadas de antemano manualmente y escribe en el Canvas el resultado de las pruebas, si es verde es que la prueba ha sido pasada satisfactoriamente.

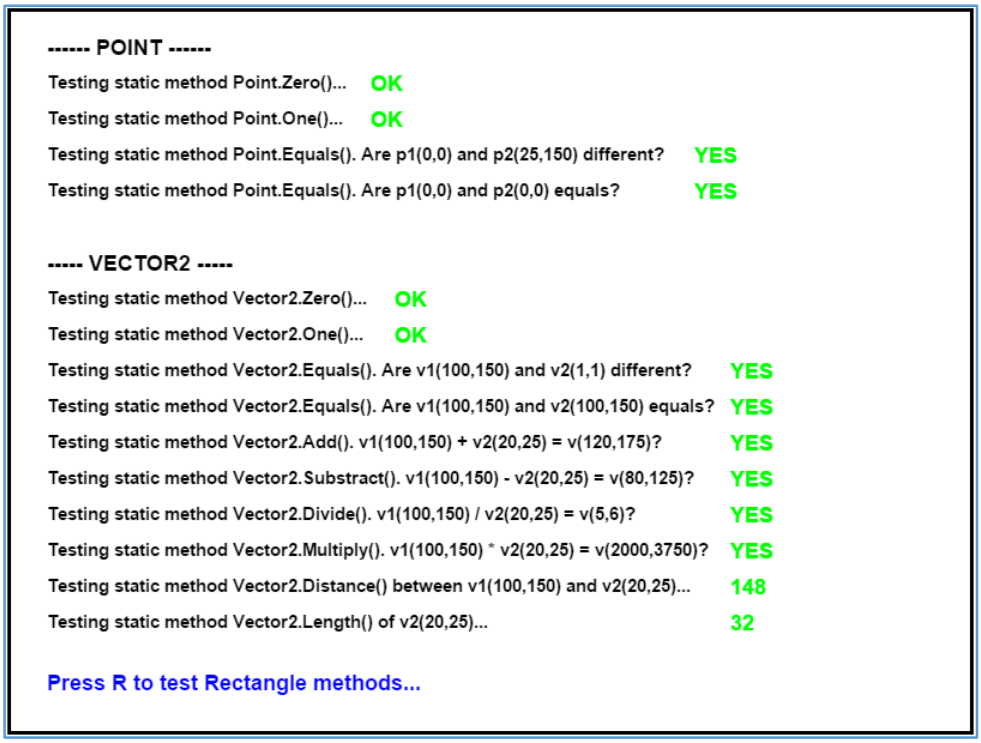


Figura 12: Pruebas unitarias de las clases Point y Vector2

Como se puede ver, esta primera fase del entorno de pruebas no es interactiva, pero si pulsamos la tecla 'r' del teclado pasaremos a la siguiente fase, donde el usuario puede probar la funcionalidad de la clase **Rectangle**.

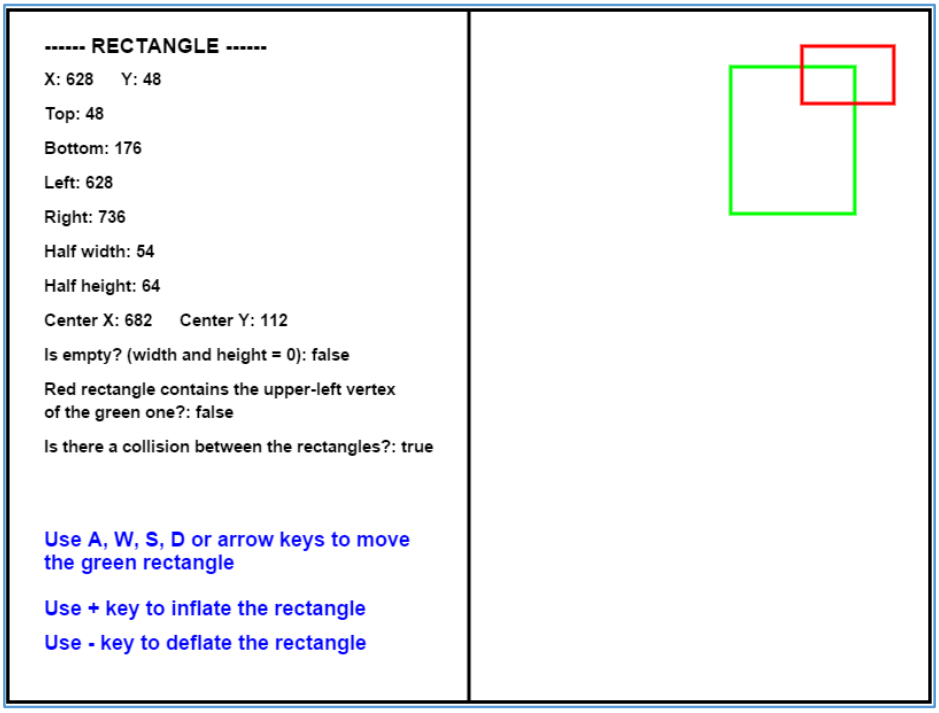


Figura 13: Pruebas unitarias de la clase Rectangle

En este entorno podemos mover el rectángulo verde dentro de los límites y aumentar o reducir su tamaño. Mientras se interactúa con él podemos comprobar diversos datos, como su posición en el Canvas, sus medidas o si está colisionando con el rectángulo rojo.

Test de Mouse y Keyboard

En este entorno probaremos la interacción del usuario con los dispositivos de entrada soportados, que son el teclado y el ratón. Como se verá a continuación, podemos comprobar la posición del ratón en el Canvas, si se está pulsando alguno de sus 3 botones o el valor de la rueda del ratón (sumará 1 cada vez que se gire hacia delante y restará 1 si se mueve hacia atrás). También podemos ver los códigos de las teclas del teclado que están siendo pulsadas y las que han sido ya pulsadas.

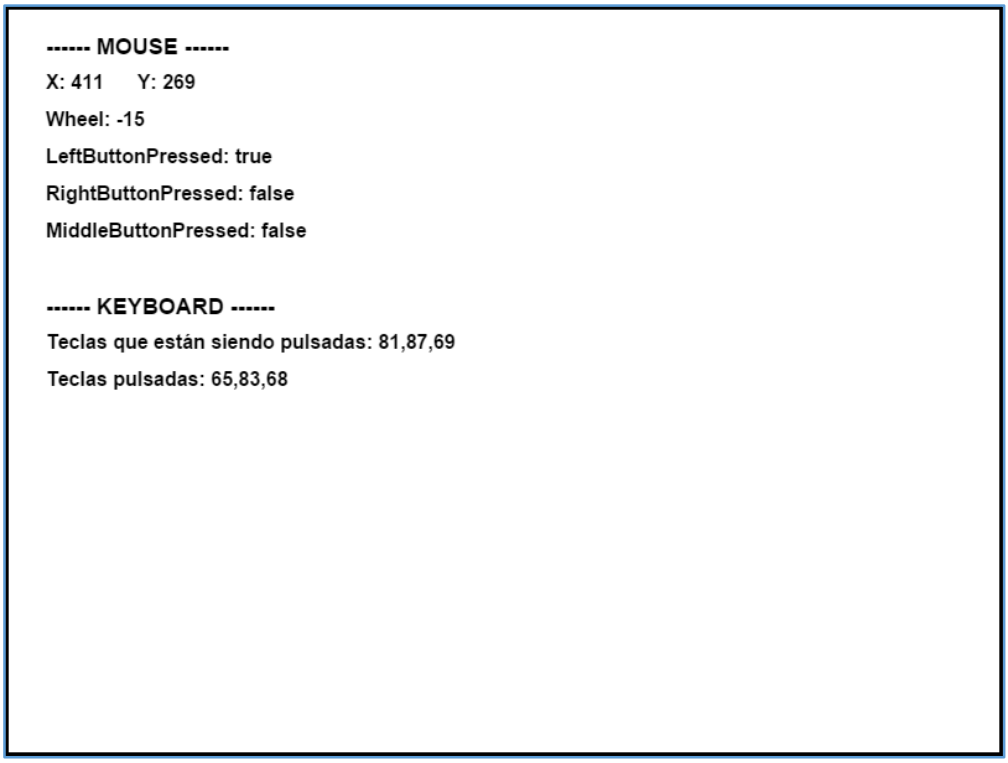


Figura 14: Pruebas unitarias de las clases Mouse y Keyboard

En este ejemplo se estaba pulsando el botón izquierdo del ratón y las teclas 'q', 'w' y 'e' del teclado. Anteriormente fueron pulsadas las teclas 'a', 's' y 'd'.

Test de Song y SoundEffect

En este test comprobaremos el funcionamiento de las dos clases relacionadas con el audio. Se probarán los métodos para reproducir/parar la canción, aumentar su volumen, silenciarla y activar el modo bucle (*loop*). También se probará la reproducción de 3 efectos de sonido diferentes al pulsar las teclas adecuadas.

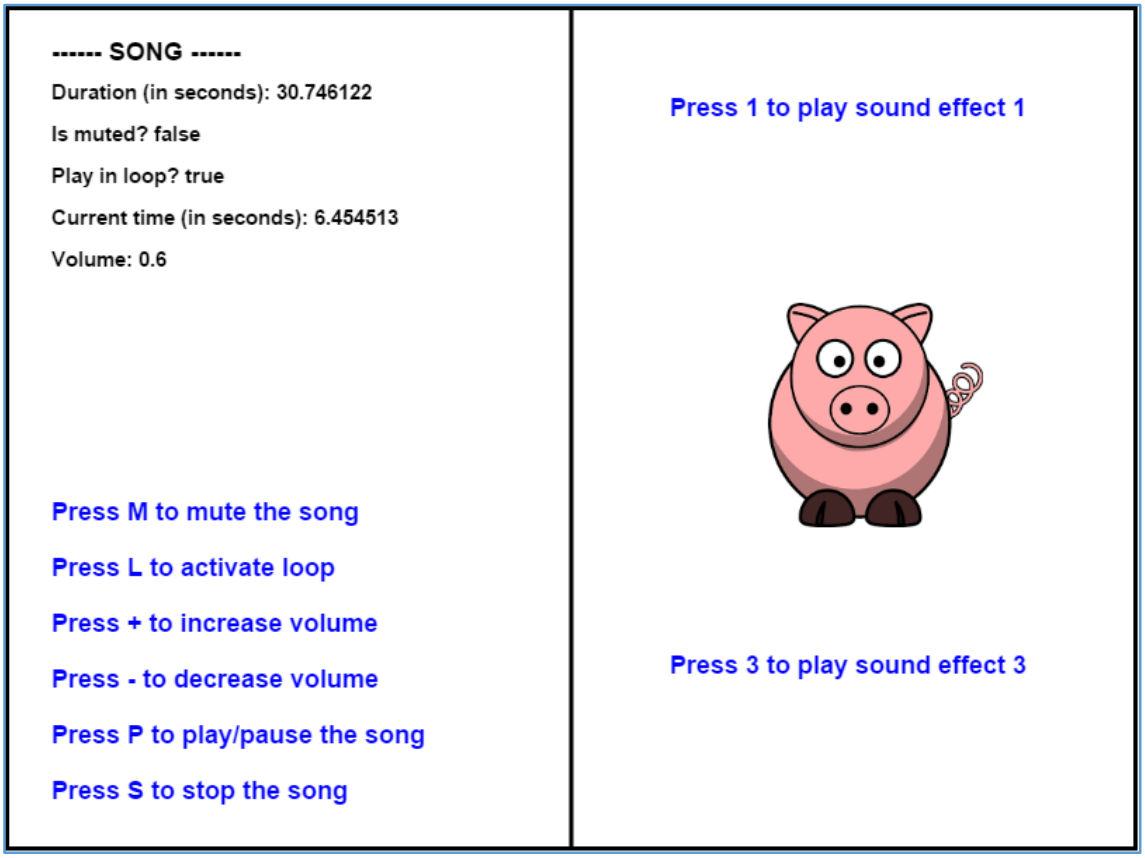


Figura 15: Pruebas unitarias de las clases Song y SoundEffect

Como en el resto de entornos de prueba, las instrucciones para interactuar con el mismo están en color azul, al presionar las teclas '1', '2' o '3' se reproducirán los efectos de sonido asociados, apareciendo una imagen relacionada con el sonido durante el tiempo que este dure.

Test de GraphicsDevice y AnimatedSprite

En este entorno se testearán los métodos relacionados con estas clases, podremos comprobar las medidas del Canvas y pasar a modo de pantalla completa o al modo en el que el Canvas ocupará toda la página. A la vez podremos ver un *sprite* animado controlable. Es posible mover el *sprite*, rotarlo, mostrar su rectángulo de colisión, aumentar o reducir su escala, invertirlo vertical u horizontalmente, comenzar/parar la animación, incrementar los milisegundos entre cada imagen de la animación (*frame*) y cambiar el tipo de la misma.

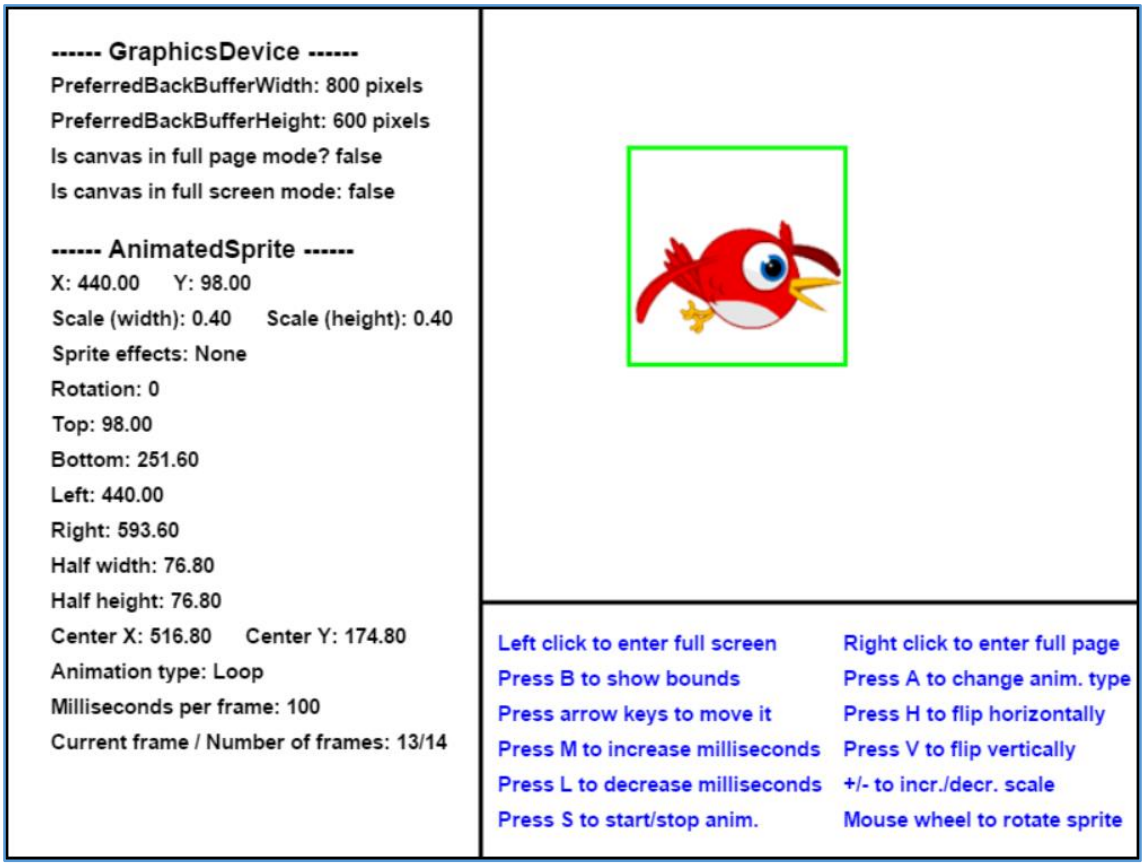


Figura 16: Pruebas unitarias de las clases GraphicsDevice y AnimatedSprite

Con este último entorno de pruebas, las clases de XNA y todos sus métodos quedan testeados.

4.2 Pruebas de la librería en conjunto

Los entornos creados para este apartado tienen como objetivo probar la implementación de las clases XNA adaptadas a JavaScript junto con el resto de las clases de la librería, de este modo se probará que el conjunto de la librería cumple con la funcionalidad básica y adicional.

Mini-juego de Rayman

El videojuego Rayman, creado por Ubisoft, fue uno de los videojuegos de plataformas con más éxito en su época, a día de hoy todavía se siguen desarrollando juegos de Rayman. La demo creada intenta imitar el aspecto y las mecánicas del juego utilizando sus mismos *sprites* (encontrados en internet). La demo incluye un pequeño nivel donde se puede controlar a Rayman, también hay enemigos dañinos si nos tocan y que se mueven de forma aleatoria por el mapa y objetos que se pueden recoger. Para completar el nivel es necesario recopilar todas las perlas y llegar a la meta sin morir. También hay objetos que permitirán a Rayman obtener el poder de volar, mejorar su puño para que sea más rápido y reponer su escudo vital.

Para la demo se han modificado las clases **Player**, **Enemy** y **Projectile** y se han creado otras 2, una contiene una enumeración con los tipos de objetos y la otra es la clase **Level**, creada para definir mediante *arrays* las posiciones en el mapa de los enemigos, los objetos, la posición inicial de Rayman, la meta y las plataformas donde caminará Rayman. Para ello se ha utilizado la herramienta Tiled, con la que se ha generado el mapa de juego y se han definido en formato JSON las posiciones de los elementos anteriormente mencionados. Los datos en JSON se han añadido a *arrays* en la clase **Level** que posteriormente serán interpretados para construir el mapa de juego.

En la demo se puede incrementar o disminuir el zoom de la cámara, mover a Rayman, hacer que salte, hacer que lance su puño, poner el juego a pantalla o página completa y mostrar la ayuda, que informa sobre cómo realizar todas estas acciones.

Figura 17: Demo jugable de Rayman hecho con la librería

Demo de físicas con pelotas

En esta pequeña demo se ponen a prueba las físicas que pueden llegar a generarse mediante la librería. El entorno de pruebas es simple, consta de una habitación vacía y mediante unos botones situados fuera del Canvas, a la derecha pueden generarse pelotas de tamaño aleatorio (dentro de unos límites) y activar la gravedad hacia arriba, abajo, izquierda o derecha. De esta manera las pelotas colisionarán y rebotarán unas con otras, generando una sensación aproximadamente realista. También es posible mediante el ratón seleccionar una pelota y moverla a cualquier sitio de la habitación. Para ello el ratón se debe situar en la pelota que se desea mover, pulsar el botón izquierdo del ratón y soltarlo cuando se desee soltar la pelota, como ya se ha visto anteriormente, esta técnica se conoce como *Drag & Drop.*

Figura 18: Demostración de físicas con pelotas

5. Conclusiones

Este ha sido un proyecto personalmente motivador, pues el mundo de los videojuegos es muy interesante dadas las infinitas posibilidades creativas que permite. Recrear un *framework* tan importante como XNA me ha proporcionado la posibilidad de experimentar dificultades técnicas que ayudan a comprender como está hecho un videojuego por dentro, lo cual es algo importante para saber cómo programarlos de una forma más eficiente.

Otro aspecto interesante ha sido el hecho de imitar el *framework* con JavaScript, pues es un lenguaje muy diferente a C#, pero que está evolucionando y mejorándose constantemente. Esto, ligado a su flexibilidad lo hace uno de los lenguajes actuales más populares. Además, no es un lenguaje que se dé muy en profundidad en la carrera y trabajar con ello me ha proporcionado un buen nivel que personalmente considero importante en mi aprendizaje como programador, ya que el desarrollo web está a la orden del día.

En vistas al futuro, la librería debe mejorarse mucho aún, pues XNA es *framework* muy grande con muchas posibilidades. Podría implementarse toda la parte 3D utilizando WebGL en vez de Canvas. Sería interesante implementar todo un motor de físicas de forma más eficiente y preparar clases que puedan adaptarse a cualquier tipo de juego, para que la librería fuera más versátil. Otro de los puntos interesantes sería añadir soporte para gestión de menús y de la interfaz del juego. También podría contemplarse el soporte para *smartphones* ya que JavaScript proporciona métodos para trabajar con la tecnología táctil, de esta manera se podrían crear juegos multiplataforma.

En resumen, este trabajo ha sido muy enriquecedor en mi aprendizaje como informático, enfrentarme solo a este reto ha llevado su tiempo, tanto investigando como programando. Al final, la librería cumple con las expectativas iniciales y es posible crear videojuegos con ella, aunque como ya se ha dicho antes puede mejorarse en muchos aspectos.

Referencias

Libros

- *Foundation Game Design With HTML5 And JavaScript*
  Rex van der Spuy

Recursos en la web

Páginas de interés

- http://www.crazyteam.es/html/javascript/escribiendo-codigo-javascript-eficiente/ - [Accedida el 24/02/2015]
- http://www.project-cyan.com/2012-03-crear-un-grid-de-tiles-con-tiled-y-javascript/ - [Accedida el 05/03/2015]
- http://www.ibm.com/developerworks/library/wa-build2dphysicsengine/ - [Accedida el 06/03/2015]
- http://www.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/ - [Accedida el 24/02/2015]
- http://bl.ocks.org/milkbread/11000965 - [Accedida el 13/04/2015]
- http://h3manth.com/content/html5-canvas-full-screen-and-full-page - [Accedida el 01/05/2015]
- http://www.sitepoint.com/html5-javascript-mouse-wheel/ - [Accedida el 18/03/2015]
- http://es.wikipedia.org/wiki/Motor_de_videojuego - [Accedida el 11/03/2015]
- http://www.anaitgames.com/status/606402 - [Accedida el 05/04/2015]
- http://www.adwe.es/general/conferencias-adwe-frameworks-para-el-desarrollo-de-juegos-para-dispositivo-moviles - [Accedida el 05/04/2015]
- http://es.wikipedia.org/wiki/WebGL - [Accedida el 03/04/2015]
- http://es.wikipedia.org/wiki/Elemento_canvas - [Accedida el 03/04/2015]
- http://stackoverflow.com/questions/21603350/is-there-any-reason-for-using-webgl-instead-of-2d-canvas-for-2d-games-apps - [Accedida el 03/04/2015]

- http://www.lewebmonster.com/desarrollo-de-juegos-con-jquery-memoria/ - [Accedida el 10/05/2015]
- http://es.wikipedia.org/wiki/Programaci%C3%B3n_basada_en_prototipos - [Accedida el 16/05/2015]
- http://es.wikipedia.org/wiki/JavaScript - [Accedida el 19/03/2015]
- http://es.wikipedia.org/wiki/C_Sharp - [Accedida el 19/03/2015]
- http://geeks.ms/blogs/etomas/archive/2013/10/25/191-es-javascript-orientado-a-objetos.aspx - [Accedida el 02/04/2015]
- http://www.programania.net/diseno-de-software/entendiendo-los-prototipos-en-javascript/ - [Accedida el 02/04/2015]
- https://developer.mozilla.org/es/docs/Web/JavaScript/Introducci%C3%B3n_a_JavaScript_orientado_a_objetos - [Accedida el 03/04/2015]
- http://ovaraksin.blogspot.com.es/2011/09/quick-practical-introduction-to-jsdoc.html - [Accedida el 04/03/2015]
- https://developer.tizen.org/dev-guide/2.2.0/org.tizen.web.appprogramming/html/guide/w3c_guide/graphics_guide/performance_comparison.htm - [Accedida el 22/04/2015]
- https://msdn.microsoft.com/es-es/library/ie/dn265058(v=vs.85).aspx - [Accedida el 29/03/2015]
- http://notasjs.blogspot.com.es/2014/09/diferencia-entre-libreria-y-framework.html - [Accedida el 28/03/2015]
- http://www.anaitgames.com/status/606402 - [Accedida el 25/04/2015]
- http://www.4rsoluciones.com/framework-sdk-biblioteca-api-cuales-son-las-diferencias/ - [Accedida el 19/05/2015]

Frameworks

- LimeJS - http://www.limejs.com/ - [Accedida el 01/04/2015]
- Phaser - https://phaser.io/ - [Accedida el 02/04/2015]
- Canvas Engine - http://canvasengine.net/ - [Accedida el 02/04/2015]
- Babylon - http://www.babylonjs.com/ - [Accedida el 06/04/2015]
- Pixi - http://www.pixijs.com/ - [Accedida el 01/04/2015]

Librerías

- Jaws - http://jawsjs.com/ - [Accedida el 07/04/2015]
- Three.js - http://threejs.org/ - [Accedida el 07/04/2015]
- Enchant - http://enchantjs.com/es/ - [Accedida el 10/04/2015]
- Melon - http://melonjs.org/ - [Accedida el 07/04/2015]
- Box2D JS - http://box2d-js.sourceforge.net/ - [Accedida el 10/04/2015]