
# Aceleración de aplicaciones web con WebCL

**Proyecto de Sistemas Informáticos**

**Facultad de Informática**

**Universidad Complutense de Madrid**

Departamento de Arquitectura, Computadores y Automática (DACYA)

Curso 2014/2015

Autores:

Alejandro Ladrón de Guevara Cano

Yamandú Leandro Sotelo Souto

Miguel Alexander Maldonado Lenis

Directores:

Carlos García Sánchez

Guillermo Botella Juan

# Autorización

*Alejandro Ladrón de Guevara Cano, Yamandú Leandro Sotelo Souto y Miguel Alexander Maldonado Lenis, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluyen imágenes de los autores, la documentación y/o el prototipo desarrollado.*

*Alejandro Ladrón de Guevara Cano*

*Yamandú Leandro Sotelo Souto*

*Miguel Alexander Maldonado Lenis*

# Agradecimientos

*Este proyecto no habría sido posible si no hubiéramos contado con el apoyo de las personas que nos rodean. Por ello, queremos dedicarlo en primer lugar a nuestras familias y amigos, fundamentales para nuestra vida. También a nuestros directores de proyecto, por su aportación de ideas y el grado de confianza que nos han facilitado. Y a los profesores que durante estos años nos han enseñado todo lo que hemos aplicado en el proyecto.*

*A nuestras familias y amigos.*

# Índice general

# Índice de imágenes

# 1. Resumen del proyecto

## 1.1. En español

El rendimiento de las aplicaciones que se ejecutan en los navegadores web a menudo presentan diferencias significativas de rendimiento dependiendo de la máquina (ordenador, móvil, Tablet…) que lo soporta. Pero en muchos casos se podría solventar realizando un aprovechamiento adecuado de todos los recursos del equipo. Por ello, en este proyecto lo que se busca es que la aplicación web explote al máximo los recursos suministrados de forma efectiva.

Al ser el procesador (CPU) el encargado de ejecutar el código de las aplicaciones web y no poder en muchas ocasiones interferir en su rendimiento, en este proyecto explotaremos la potencia de computo de la tarjeta gráfica (GPU) para liberar de trabajo al procesador y poder alcanzar el mejor rendimiento de la aplicación en la máquina en donde se ejecute.

## 1.2. En Inglés

The performance of applications that run on Web browsers often show significant performance differences depending on the machine (computer, mobile, tablet ...) that supports it. But in many cases it could afford making adequate use of all computer resources. Therefore in this project what is sought is that the web application to fully exploit the resources provided effectively.

As the processor (CPU) responsible for implementing the code of web applications and often cannot interfere in performance, this project will exploit the computing power of the graphics card (GPU) to release the processor and work to achieve the best performance of the application on the machine where it is executed.

# 2. Palabras clave

Las palabras clave asociadas a este proyecto son:

- WebCL
- Nokia
- Juego de la vida
- Paralelización
- GPU
- Rendimiento computacional
- Mozilla Firefox
- Navegador web
- Liferinth

# 3. Lista de Acrónimos y Términos

En la siguiente lista se incluyen todos los acrónimos y términos empleados en el documento para facilitar al lector su comprensión del mismo.

| | |
|---|---|
| Antialiasing | La eliminación de información insignificante como para aparecer en la señal del monitor. |
| Array | Es la palabra en inglés usada en programación para referirnos a un vector de elementos colocados en una fila. |
| Canvas | Es un elemento de HTML que facilita el uso de gráficos. |
| Computación paralela | Es la ejecución de varias instrucciones al mismo tiempo, empleando hardware específico. Nos permite procesar más de una instrucción a la vez, pudiendo ser estas de un mismo proceso o de procesos distintos. |
| CPU | (Central Processing Unit) Es el hardware que ejecuta las instrucciones de un programa informático. Se vale del resto de componentes del sistema para interactuar con el usuario. |
| CUDA | (Compute Unified Device Architecture) Se identifica tanto a la arquitectura hardware integrada en las tarjetas gráficas de NVIDIA como a la extensión del lenguaje C para ejecutar código en ellas. |
| Framework | Base o infraestructura para el desarrollo de software. Puede ofrecer Bibliotecas u otras herramientas para ayudar a desarrollar un proyecto. |

| | |
|---|---|
| Gamepad | Mando con el que el usuario interactúa con el juego, tomando el control de las acciones del personaje. |
| GPU | (Graphics Processing Unit) Es la Unidad de procesamiento gráfico que encontramos en los ordenadores y en muchos dispositivos, que se encarga de procesar la información gráfica para mostrarla por pantalla. |
| Hardware | Componentes físicos que forman el sistema informático. |
| Kernel | Núcleo de código que se ejecuta encapsulado dentro de una función para ejecutarse en la GPU (terminología específica para el proyecto). |
| OpenCL | (Open Computing Language) Es un lenguaje de computación compuesto por una interfaz de programación que permite escribir código para que pueda ejecutarse tanto en CPU como en GPU. |
| OpenGL | (Open Graphics Library) Define un API para ayudar a producir gráficos 2D y 3D independientes de la plataforma y del lenguaje. |
| Paralelización | Tiene como objetivo la implementación de código susceptible de ser computado en paralelo. Normalmente partiendo de una versión secuencial se genera una versión equivalente que pueda ejecutarse en paralelo obteniendo el mismo resultado. |
| Plug-in | Es un complemento de una aplicación que sirve para extender la aplicación implementando una funcionalidad específica. |
| Profiling | Es el análisis del rendimiento de un programa informático usando la información recopilada de los datos obtenidos de la máquina. Su objetivo es detectar las partes del programa donde existen puntos problemáticos que ralentizan su ejecución, en velocidad o consumo de recursos. |
| Renderizar | El proceso de generación de la imagen que aparece en el monitor partiendo de modelos 3D. |

| | |
|---|---|
| Shaders | Los shaders se utilizan para realizar transformaciones y crear efectos especiales, como por ejemplo iluminación, sombras o niebla. Es una tecnología para llevar código a las GPU. |
| Sprite | Son dibujos realizados por hardware por medio de mapas de bits. |
| WebCL | *"(Web Computing Language) Es una tecnología que permite la integración de OpenCL en código JavaScript para poder usar programación paralela heterogénea y así aprovechar las ventajas de los CPUs y GPUs con varios núcleos."* [1] |
| WebGL | "*Es una especificación estándar que está siendo desarrollada actualmente para mostrar gráficos en 3D en navegadores web en cualquier plataforma que soporte OpenGL 2.0."* [2] |

# 4. Introducción y motivación

Con el auge de los procesadores multi-core y aceleradores gráficos, la industria ha ido ofreciendo diferentes paradigmas de programación como CUDA de NVIDIA o su homónimo más generalista OpenCL. OpenCL trata de instalarse como estándar de programación tanto arquitecturas multi y many-core, como en otro tipo de dispositivos como DSP, FPGAs, etc. Por otro lado, desde el punto de vista del consumidor final de dispositivos móviles como tabletas, teléfonos o vídeo-consolas, su utilización se reduce al consumo de contenidos multimedia y navegación por la web. Por este motivo, surge como iniciativa la incorporación de este tipo de tecnología a los navegadores web (concretamente Firefox y Chrome).

Bajo estas premisas, el proyecto propone la utilización de estas tecnologías para el desarrollo de aplicaciones web de procesamiento o en particular el desarrollo videojuegos online.

Este proyecto se centra en el ámbito de la computación híbrida CPU-GPU, la versatilidad que ofrecen los navegadores web (en este caso Firefox al ser el único que lo soporta completamente), el consumo de recursos de las máquinas pero pensando siempre en el mercado de los dispositivos móviles.

El proyecto propone el uso de WebCL como solución eficaz al reciente interés, tanto en el ámbito académico como en el industrial, por la computación gráfica. Hasta el momento los avances más notables se han asociado a OpenCL o CUDA. Apostando por el navegador como interlocutor del sistema.

Cada vez la tendencia en las tecnologías de la información mira hacia la portabilidad, facilidad y la eficiencia. Se necesita de tecnologías que funcionen en el

mayor número de sistemas, que puedan ser fácilmente implementadas y con un bajo consumo de los recursos del sistema. En este proyecto la abstracción que ofrece WebCL sobre la computación GPU es muy similar a la ofrecida en OpenCL, pero con la potencia de necesitar únicamente un navegador web que lo soporte. Es una clara abstracción a un nivel superior a la hora de explotar la tarjeta gráfica.

## 4.1. WebCL

WebCL nace para definir el enlace entre OpenCL y JavaScript, integrando el uno en el otro, permitiendo a las aplicaciones web acceder desde el navegador al procesamiento paralelo de las tarjetas gráficas (GPU) y los procesadores (CPU) multi-core (de varios núcleos) sin necesidad de ningún plug-in externo, siempre que el navegador lo soporte.

Esto posibilita aprovechar al máximo los recursos de los sistemas desde los navegadores web, acelerando en gran medida la ejecución de programas que contengan fragmentos de código paralelizable, especialmente cuando estos fragmentos son de un coste considerable (consumen mucho tiempo de CPU), lo cual permite a los navegadores ejecutar programas intensivos en cálculo, como editores de vídeo, motores de físicas, o videojuegos.

Desarrollado por el grupo Khronos, su primera especificación “WebCL 1.0” salió el 14 de marzo de 2014 en la cual se definen los métodos de la API (interfaz de programación) necesarios para utilizarlo.

A día de hoy, ningún navegador soporta nativamente WebCL, pero es previsible que lo hagan próximamente, ya que es una tecnología nueva, de la misma familia que las ya soportadas, como WebGL. Por tanto, por ahora es necesario el uso de una extensión para que funcione, como la desarrollada por Nokia, “Nokia WebCL” que funciona con Mozilla Firefox (versiones 30 a 35).

Para maximizar la portabilidad y vigencia de tecnologías nuevas como es el caso, son muy necesarios proyectos de estas características que apuestan por integrarlos en sus desarrollos. Contribuyendo a la comunidad informática mostrando al mundo académico el potencial y capacidad los más objetivamente posible.

Es importante tener en cuenta que provee de capacidades de renderizado como WebGL, sólo procesa datos.

### 4.1.1. Paralelismo

Para entender el impacto de la aplicación de WebCL hay que explicar en primer lugar lo que es el paralelismo y una ejecución paralela.

La programación de un dispositivo paralelo es un reto y, para muchos desarrolladores, puede requerir del aprendizaje de nuevas habilidades de programación. Por paralelo, nos referimos a que muchas unidades de hardware de procesamiento se ejecutan a la vez o, dicho de otra manera, muchos hilos de hardware están ejecutando se al mismo tiempo. Para las aplicaciones web basados en eventos secuenciales con el lenguaje JavaScript, se trata de un cambio radical.

El siguiente ejemplo muestra la idea de paralelizar un bucle secuencial:

```
function multiplica(a, b, n){
   var c=[];
   for(var i = 0; i < n; i++)
      c[i] = a[i] * b[i];
   return c;
}
```

**Imagen 1. Bucle secuencial.**

• Si en lugar del bucle tradicional ejecutamos un kernel con la opción de multiplicación con cada unidad de los datos en paralelo.

• Cada elemento de trabajo ejecuta una copia del kernel al mismo tiempo.

• Por lo que, con n elementos de trabajo en n núcleos obtenemos el resultado en una sola etapa, mientras que con el bucle tradicional necesitamos n pasadas al bucle.

```
__kernel void multiplica(__global const float *a,
                         __global const float *b,
                         __global float *c,
                         int n){
   int id = get_global_id(0);
   if(id >= n) return;
   c[id] = a[id] * b[id];
}
```

**Imagen 2. Bucle paralelo con WebCL.**

Con esto hemos conseguido realizar el mismo cálculo en una sola pasada. Pero para ello necesitamos que nuestro hardware soporte este tipo de ejecución. Mientras las CPUs tienden a tener 2, 4 u 8 núcleos, las GPU puede tener miles de núcleos. Incluso en los dispositivos móviles están llegando las GPU con cientos de núcleos.

Existen distintos tipos de paralelismo.

- A nivel de **bit**.

Este nivel de paralelismo afecta al tamaño de las líneas del procesador. En un procesador de 8 bits para sumar dos números de 16 bits necesitamos realizar la suma de la parte inferior, generar el acarreo, si hubiera, y usarlo en la suma de la parte superior. Sin embargo un procesador de 16 bits o más sólo necesitaría de una instrucción. Esta ventaja nos permite trabajar con registros más grandes además de reducir el número de instrucciones que debemos ejecutar.

- A nivel de **instrucción**.

Un programa lo componen miles de instrucciones, en ocasiones muchas de ellas pueden ejecutarse en distinto orden sin afectar el resultado final. Esto nos facilitará la optimización de los recursos requeridas por las instrucciones. El procesador se divide en varias etapas y cada una corresponde a una acción diferente a realizar por la instrucción. Así cada instrucción pasa por las etapas necesarias para realizar su función.

Pero también se pueden agrupar por conjuntos, siempre que no existan dependencias de datos, y ejecutar las instrucciones en un orden distinto al escrito por el programador. Conocido como ejecución fuera de orden gracias al *Scoreboarding* o el algoritmo de Tomasulo.

- A nivel de **datos**.

Este tipo de paralelismo reside en el uso de los datos. Este tipo de paralelismo lo podremos explotar en programas con ciclos de ejecución sobre un conjunto de datos. Para aprovecharnos de este tipo de paralelismo tendremos que analizar cuidadosamente el orden en el que se utilizan los datos en el algoritmo, para estar seguros de la corrección de la implementación paralela equivalente. En muchas ocasiones nos encontraremos con algoritmos que utilizan como entrada el resultado del ciclo anterior y será necesario modificar la estrategia con la que se aborda la solución, al ser imposible la paralelización de ese algoritmo.

- A nivel de **tareas**.

Se caracteriza por realizar los cálculos independientes como tareas que pueden realizarse en paralelo pero necesarias para obtener la solución de forma conjunta.

Los conceptos OpenCL se introducen en la siguiente sección.

### 4.1.2. OpenCL

Es un estándar multiplataforma para procesamiento paralelo, válido para, entre otros, CPUs, GPUs, y FPGAs, desarrollado también por el grupo Khronos. Ser multiplataforma le dota de una gran portabilidad, el código escrito para un procesador puede ser utilizado en otro hardware.

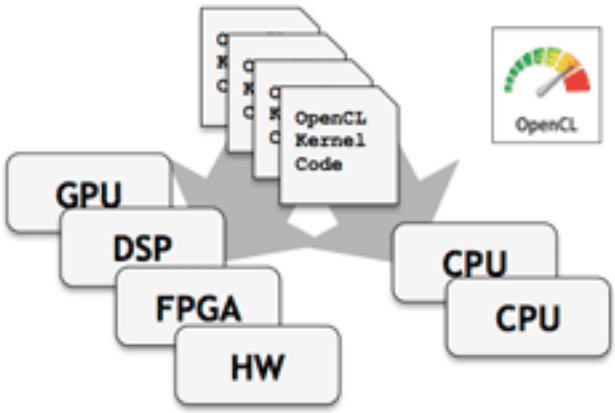



**Imagen 3. Portabilidad de código OpenCL [1]**

Con el auge de la telefonía móvil (con una amplia gama de dispositivos distintos) y los procesadores cada vez más y más potentes (incluso supercomputadores), era necesario disponer de un estándar que fuera portable, para funcionar en todos los dispositivos y tener la capacidad para explotar todos los nuevos recursos de que se dispone.

Su primera especificación “OpenCL 1.0” se publicó el 6 de octubre del 2009, y la última (hasta la fecha) “OpenCL 2.1 Provisional” el 29 de enero del 2015.

OpenCL utiliza paralelismo basado en tareas y en datos:

- El **paralelismo basado en tareas** se fundamenta en distribuir la ejecución de sub-procesos (tareas) entre los diferentes nodos de procesamiento. Si se tienen que realizar dos tareas, un procesador realiza una, mientras que el otro se encarga de la otra. Así, el tiempo de realizar ambas es menor, ya que se realizan las dos tareas simultáneamente.
- El **paralelismo basado en datos** se consigue cuando cada procesador ejecuta la misma instrucción, pero sobre diferentes conjuntos de datos. Si se tiene que realizar un bucle y tenemos dos procesadores, uno de ellos tratará la mitad del mismo, y el otro se encargará de la otra mitad, reduciendo así el tiempo de ejecución de esa tarea.

Es importante entender que OpenCL es un modelo de plataforma. Esto quiere decir que necesitaremos mover toda la información necesaria al dispositivo si queremos utilizarla en nuestro kernel. En OpenCL encontramos dos regiones de memoria muy diferenciadas, una referente al *Host* (en este caso la CPU) y otra del *Compute Device* (en este caso la GPU). Como ambas regiones de memoria son independientes tendremos que encargarnos de la transmisión de los datos.

En general, nos referiremos a *Host* para el dispositivo en el que se ejecuta el programa WebCL (es decir, el navegador). Nos referimos a un *Compute Device* al dispositivo en el que se ejecuta el kernel OpenCL (es decir, la GPU). Por lo tanto, una CPU puede ser tanto un *Host* y un *Compute Device*, pero para ello necesitaremos que la CPU lo permita y tener instalado en el sistema OpenCL.

OpenCL define 4 tipos de espacios de memoria dentro de un *Compute Device*:

• **Memoria Global** - Corresponde a la memoria RAM del dispositivo. Aquí es donde se almacenan los datos de entrada. Disponible para todos los grupos/elementos de trabajo. Al igual que la memoria del sistema es lenta. No está en caché.

• **Memoria Constante** - Caché de memoria global. Al mismo nivel que la memoria global.

• **Memoria Local** - Memoria de alta velocidad compartida entre trabajos de una unidad de cálculo (es decir, para un grupo de trabajo). Similar a la caché L1. Memoria razonablemente rápida.

• **Memoria privada** - Registros de un elemento de trabajo. Memoria muy rápida.

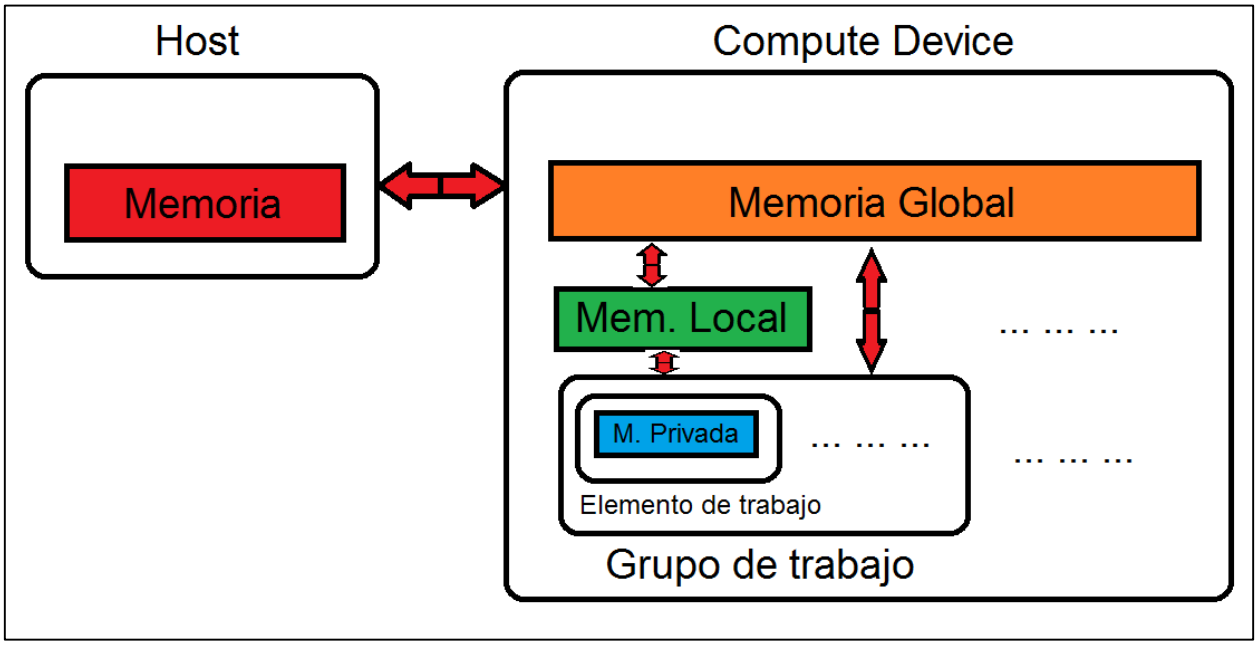



**Imagen 4. Espacios de memoria OpenCL.**

Sin embargo, la memoria privada es pequeña y la memoria local es a menudo cercana a los 64 KB. Como resultado de ello, el programador debe elegir con cuidado qué variables deja en cada espacio de memoria para obtener el mejor rendimiento de acceso a memoria.

Otro tipo de memoria es la llamada *Texture Memory*, que es similar a la memoria global salvo que se almacena en caché, optimizado para la localidad espacial en 2D, y diseñado para la transmisión. La latencia por su lectura es constante. En otras palabras,

si el dispositivo tiene soporte de imágenes y sus datos caben, obtendremos un buen rendimiento sin necesidad de realizar acciones adicionales.

## 4.2. El algoritmo del juego de la vida

El creador de este algoritmo es **John Horton Conway**, un matemático inglés formado en la Universidad de Cambridge. La idea del algoritmo fue publicada en octubre de 1970 por la revista Scientific American (en el número 223) [3]. Lo llamó algoritmo de la vida porque parece reflejar la evolución de un grupo de organismos vivos que nacen, se reproducen y mueren.

### 4.2.1. El algoritmo original

Se tiene un tablero dividido en celdas (como un tablero de ajedrez) en las cuales puede haber un "organismo vivo" (celda ocupada) o bien no haber nada o haber un "organismo sin vida" (celda libre). Con cada iteración del algoritmo, todas las celdas serán actualizadas en función de la configuración actual del tablero.

Para saber el estado de una celda en la siguiente iteración, hay que mirar a sus celdas "vecinas". Se usa una 8-vecindad para determinar las celdas que son adyacentes a otra, esto quiere decir que, para que una casilla sea adyacente a otra, tiene que estar a distancia 1 de otra, medida vertical, horizontal o diagonalmente (el cuadrado de celdas que envuelve a la casilla en cuestión).

En función del número de celdas vecinas que estén vivas se determina si una casilla estará viva o no en el siguiente paso del algoritmo:

- Toda celda ocupada, con 2 o 3 vecinas vivas, sobrevive.
- Toda celda ocupada, con 4 o más vecinas vivas, muere por inanición (a causa de la sobrepoblación).
- Toda celda ocupada, con 1 o menos vecinas vivas, muere por aislamiento.
- Toda celda vacía, con exactamente 3 vecinas vivas, nace (pasa a estar ocupada).

El objetivo de estas reglas es el siguiente:

- Que no haya ninguna configuración inicial del problema que haga que la población

crezca (se reproduzca) indefinidamente.

- Que haya configuraciones iniciales en las que '*parezca'* que una población vaya a crecer sin límites, pero que no lo haga finalmente.
- Que haya configuraciones iniciales que hagan que la población crezca y cambie durante un buen periodo de tiempo antes de acabar de tres maneras posibles:
    - La ***extinción***: Por sobrepoblación o aislamiento.
    - La ***estabilización***: En una configuración en la que no hay cambios.
    - La ***oscilación***: Entre dos o más configuraciones en un bucle infinito.

### 4.2.2. El algoritmo aplicado al proyecto

El objetivo de la aplicación de este algoritmo del juego de la vida en el proyecto es conseguir un algoritmo que requiera una gran cantidad de cálculos y que estos sean paralelizables:

- Al tener que recorrer todas las celdas (en las que se encuentra cada muro) mirando sus casillas vecinas, es un algoritmo de orden más que lineal. Aunque no es cuadrático (sería del orden de 9n), requiere, en cada vuelta del bucle, **mucho cálculo**.
- Al no depender una celda de si la vecina va a sobrevivir o no (sólo le importa si está viva o no en este momento), este algoritmo es totalmente **paralelizable**. Todos los cálculos se pueden hacer al mismo tiempo.

La idea de un laberinto con paredes que se mueven también hace pensar que el laberinto está vivo, y aplicar a éste un algoritmo de la vida como gestor de las paredes, refuerza la visión de un laberinto con mentalidad propia que intenta atrapar al que entra en él.

Las paredes serán las celdas del algoritmo, una celda viva implicará un muro levantado. Cuando esta celda muera (o estuviera vacía inicialmente), el muro bajará y permitirá el paso del jugador.

### 4.2.3. Detalles de la implementación

Como se ha explicado anteriormente, los organismos de este algoritmo son las paredes. Una pared puede tener cinco estados:

- **Viva.**
- **Pared permanente.**
- **Muerta.**
- **Hueco permanente.**
- **Inexistente.**

Al disponer de más estados que en el algoritmo original, hace falta una equivalencia entre estos y los originales.

- ❖ Viva y pared permanente se corresponden con el de celda **ocupada**.
- ❖ Los otros tres estados se corresponden con el de celda **vacía**.

Por ser un juego en el que el usuario tiene que poder moverse, el algoritmo no se está ejecutando constantemente, se usa un temporizador que ejecuta cada una de las iteraciones del mismo.

Las paredes no son exactamente celdas de un tablero de ajedrez en el espacio, lo cual hace que visualizar los 8-vecinos de una celda sea más difícil, y viene determinado por la manera en la que está hecha la matriz de muros, tal y como se muestra en la siguiente imagen:

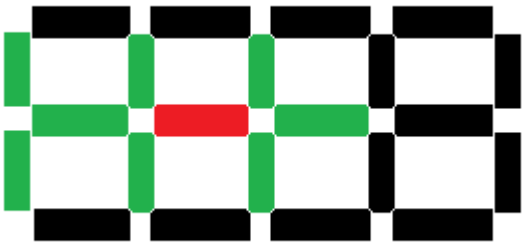


**Imagen 5. Vecindad de muros.**

Hay dos implementaciones del juego de la vida realizadas, una con código JavaScript y otra con WebCL.

### 4.2.4. Diferencias con el algoritmo inicial

A diferencia de lo que pretendió John Horton con su juego de la vida, en este caso el objetivo no puede ser alcanzar una configuración estable, ni una completa extinción de los organismos (paredes), ya que eso daría mucha facilidad al jugador para encontrar la salida, o incluso se la impidieran totalmente.

El objetivo es poner un desafío al jugador en el que tenga que moverse por un laberinto en constante cambio. Por consiguiente se han tomado medidas que intentan evitar que se produzcan estas situaciones:

- **Muros permanentes**.

  En la parte central del laberinto se colocan aleatoriamente al principio de la partida muros que nunca bajarán, aunque aplicando el algoritmo original les tocara bajar. Esta modificación impide que deje de haber muros en el laberinto, ya que siempre habrá alguno levantado en la parte central, y repercutirán en el comportamiento de los demás, generando nuevos muros.

- **Huecos permanentes.**

  Principalmente para impedir que se levante el muro de la salida del laberinto, aunque podrían usarse para dejar un camino abierto en caso de necesidad.

- **Número de paredes máximo y mínimo.**

  Se contabiliza el número de muros que está vivo en cada momento en el laberinto para introducir muros en el caso de que falten o eliminarlos en caso de que haya paredes de más.

Una buena forma de entender el funcionamiento del algoritmo es verlo en acción en el juego. La mejor forma es pulsar la tecla ‘J’ y visualizar la partida desde una posición cenital. Con las siguientes capturas mostramos al lector un ejemplo del movimiento de las paredes tras un ciclo de vida.

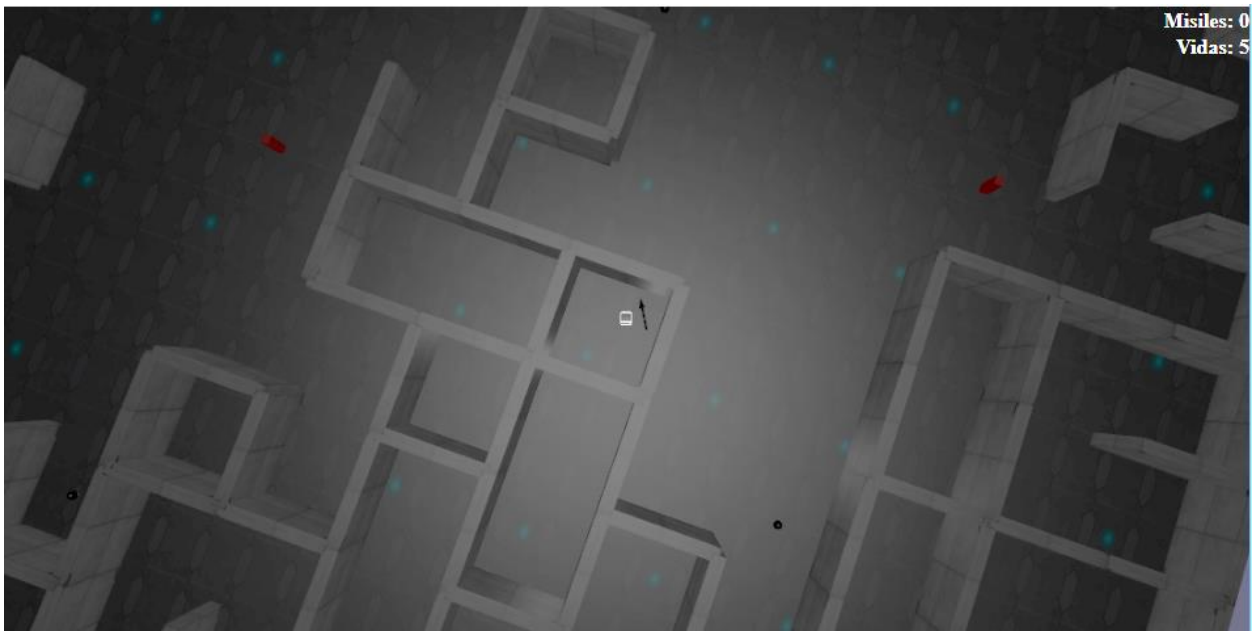



**Imagen 6. Situación inicial.**

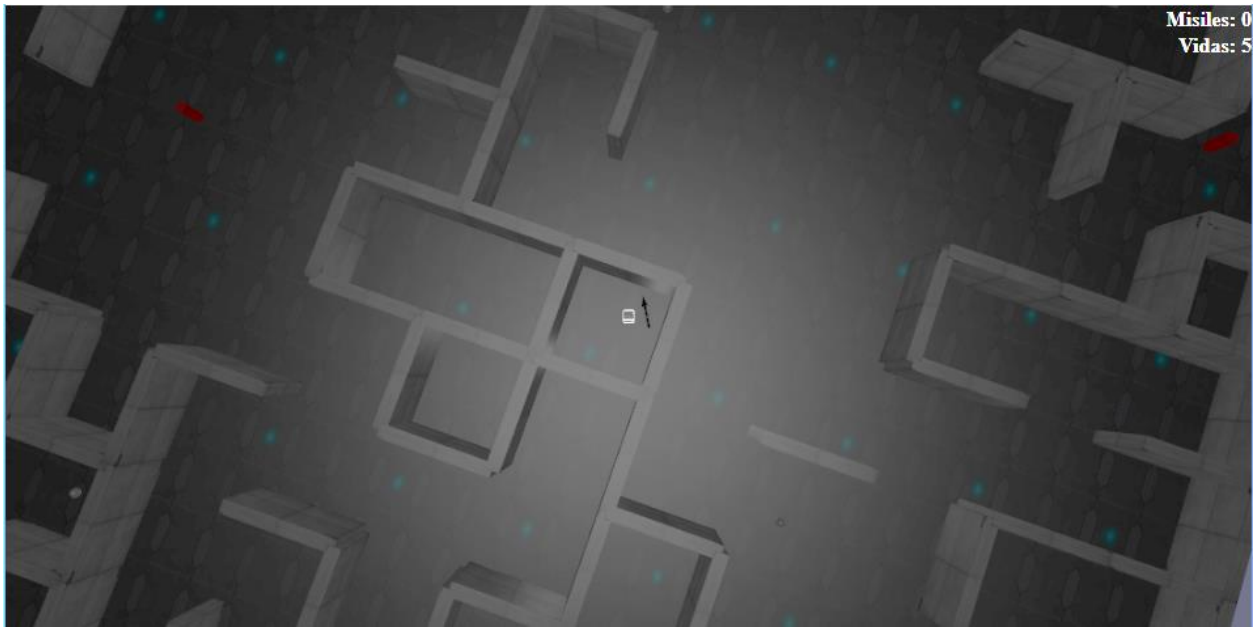



**Imagen 7. Situación en el siguiente ciclo.**

## 4.3. BabylonJS

El motor gráfico 3D del que este proyecto hace uso para mostrar todos los elementos 3D es BabylonJS[1]. Añadimos una breve descripción para el lector. [4]

### 4.3.1. ¿Qué es Babylon.js?

Es un *framework*[2] JavaScript de código abierto para el desarrollo de videojuegos 3D con HTML5 y WebGL, aunque se encuentra en una edad temprana el motor presenta una serie muy extensa de completas características, como el manejo de luces, materiales, cámaras, *sprites*[3], capas, mallas, motor de colisiones, texturas, motor de animaciones, sistemas de partículas, *antialiasing*[4], conversión de archivos de modelado 3D como **.**OBJ, .FBX y .MXB, entre otras, que permiten al desarrollador realizar sus creaciones sin problema.

Por el momento funciona adecuadamente en Firefox y Chrome. Aparentemente es una plataforma neutral, sin embargo, el sitio web del motor gráfico advierte que para Internet Explorer 11 aún está como *preview* (fase no final) y muchos *shaders*[5] no funcionan aún.

### 4.3.2. ¿Cómo funciona?

BabylonJS utiliza el elemento *canvas* de HTML5 como lienzo para dibujar la escena mediante WebGL. Para que la escena se pueda visualizar se debe instanciar el motor que se encargará de mostrar el lienzo (*canvas*). El motor es el eje entre babylon.js y WebGL, el encargado de enviar las órdenes y la creación interna de los objetos relacionados en WebGL.

---

[1] Ver explicación de la decisión en 6.1.2 Elección del motor gráfico.

[2] Ver definición en 3 Lista de Acrónimos y Términos.

[3] Ver definición en 3 Lista de Acrónimos y Términos.

[4] Ver definición en 3 Lista de Acrónimos y Términos.

[5] Ver definición en 3 Lista de Acrónimos y Términos.

Una vez creado el motor, podemos crear la escena. La escena puede ser vista como un contenedor para todas las entidades que trabajan en conjunto para crear una imagen en 3D.

Babylon.js soporta diferentes tipos de cámaras, luces, mallas e incorpora una biblioteca matemática completa con la cual se pueden manejar vectores, matrices, colores, rayas y *cuaterniones* (proporcionan una notación matemática para representar las orientaciones y las rotaciones de objetos en tres dimensiones) Se debe tener en cuenta que babylon.js utiliza un sistema de coordenadas de mano izquierda:

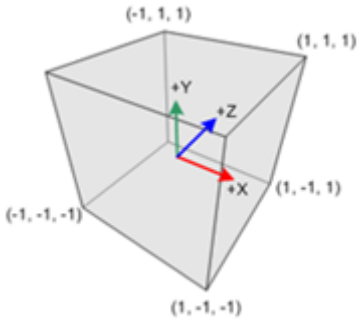



**Imagen 8. Sistema de coordenadas basado en el método de la mano izquierda.**[6]

Para que constantemente se esté actualizando nuestro lienzo es necesario definir un ciclo para el renderizado. Por lo tanto el último paso consiste en definir y registrar un ciclo render:

```
// Render loop
var renderLoop = function () {
 // Start new frame
 engine.beginFrame();

scene.render();

// Present
 engine.endFrame();

// Register new frame
 BABYLON.Tools.QueueNewFrame(renderLoop);
};

BABYLON.Tools.QueueNewFrame(renderLoop);
```

**Imagen 9. Ciclo render.**

[6] Imagen de David Catuhe [4].

Usando *QueueNewFrame*, se hace una llamada a *requestAnimationFrame*, esta función está apoyada en un *setTimeout* que le pide al navegador que llame a *renderloop* tan pronto como sea posible.

El *renderloop* en sí se basa en 4 partes:

1. engine.beginFrame: Determinar el inicio de un nuevo *frame*.
2. scene.render: Pide al *frame* hacer que todas las entidades sean de su propiedad.
3. engine.endFrame: Cierra el *frame* actual y lo presenta en el *canvas*.
4. QueueNewFrame: Registra un nuevo *frame* para la representación.

### 4.3.3. Características a destacar

**Materiales**

Un material es un objeto que define la forma de la malla. Babylon.js provee un objeto llamado *StandardMaterial* el cual cuenta con las propiedades que permiten definir el color de la malla, la transparencia, el color de reflexión recibida por la malla, etc.

**Luces**

Babylon.js cuenta con 4 tipos de luces diferentes (se pueden crear tantas luces como se quiera pero el *StandardMaterial* sólo puede tener en cuenta hasta 4 luces simultáneamente):

1. *PointLight*: Emite luz en todas las direcciones desde una posición específica.
2. *DirectionalLight*: Emite luz desde el infinito hacia una dirección específica.
3. *SpotLight*: Emite luz desde una posición a una dirección, como si fuera un cono.
4. *HemisphericLight*: Luz ambiental especial basada en una dirección para determinar si el color de la luz viene desde el suelo o desde el cielo.

Las luces tienen 3 propiedades principales:

1. *diffuse*: Color difuso que determina la parte reflejada de la luz.
2. *specular*: Color especular.
3. position: posición/dirección.

*SpotLights* también cuentan con:

1. *angle*: Ángulo del cono.
2. *exponent*: Exponente de atenuación.

*HemisphericLights* también cuentan con:

1. *groundColor*: Color de la parte del suelo de la luz.

**Cámaras**

Babylon.js soporta varios tipos de cámaras. Se pueden crear tantas cámaras como se quiera pero sólo una puede estar activa al tiempo, se pueden activar varias cámaras sólo si se usa multi-viewer.

1. *FreeCamera*: Cámara FPS que se puede controlar con las teclas y el mouse (Cámara en primera persona).
2. *TouchCamera*: Cámara controlada con eventos táctiles (Requiere un componente llamado hand.js para trabajar).
3. *ArcRotateCamera*: Cámara que gira alrededor de un pivote dado, se puede controlar con los eventos del ratón o toque (Requiere hand.js para trabajar).
4. *DeviceOrientationCamera*: Cámara que reacciona a los eventos de orientación del dispositivo, es decir cuando giras el dispositivo móvil.
5. *FollowCamera*: Cámara diseñada para seguir cualquier elemento de la escena desde cualquier ángulo.
6. *VirtualJoysticksCamera*: Cámara que reacciona a los eventos de un Joystick virtual, el cual es dibujado en el *canvas* mediante gráficos 2D para controlar las cámaras y otros elementos del escenario.
7. *OculusCamera*: Cámara de doble vista que reacciona a los eventos generados por el Oculus Rift (Casco de realidad virtual).
8. *AnaglyphCamera*: Cámara para gafas 3D de colores rojo y cian, que utiliza técnicas de filtrado de post-procesamiento.

9. *GamepadCamera*: Cámara para trabajar con *gamepads*[7].

Todas las cámaras pueden manejar automáticamente las entradas del *canvas* llamando a la función *attachControl*, se puede revocar mediante el uso de *detachControl*.

### Mallas

Entidades que se pueden crear a partir de formas básicas o de lista de vértices y caras. Las formas básicas son el Cubo, la Esfera, el Plano, el Cilindro, el Toro y el Nudo.

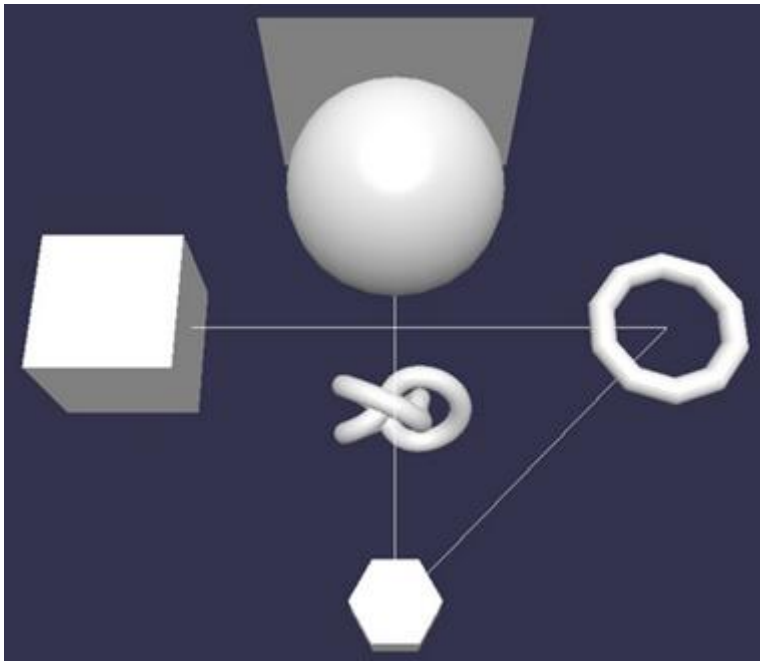


**Imagen 10. Formas básicas.[8]**

También se pueden crear mallas a partir de líneas, vértices e índices. Una vez creada la malla esta se puede modificar mediante las propiedades de posición, rotación y escala.

[7] Ver definición en 3 Lista de Acrónimos y Términos.

[8] Imagen de Andrew Toskin [23].

Un punto muy interesante es que se tiene la opción de poder establecer jerarquías entre las mallas, de esta forma todas las transformaciones en cuanto a la posición, rotación y escala del padre afectarán a los hijos.

Otras propiedades con las que se cuenta para la activación y la visibilidad de las mallas son:

1. *StandardMaterial*: Permite controlar la opacidad de un objeto.
2. *Visibility*: Controla transparencia por malla directamente sin usar un material.
3. *isVisible*: Activa la renderización de la malla. La malla se mantiene en la escena para otras operaciones y esta no es transmitida a los hijos.
4. *setEnabled*: Desactiva una malla y todos sus descendientes.

**Colisiones**

Babylon.js cuenta con un sistema de colisiones completo y muy fácil de utilizar. Una de las propiedades a destacar es la intersección entre objetos:

1. *intersectsMesh*: Se puede detectar si hay colisión entre objetos.
2. *intersectsPoint*: Se puede detectar si hay colisión con un vector en el espacio, es decir puntos específicos.

**Sistemas de partículas**

Mediante un sistema de partículas[9] se pueden añadir muy fácilmente efectos tales como explosiones, impactos, efectos mágicos, etc.

Los sistemas de partículas cuentan con un las siguientes propiedades más importantes:

1. *particleTexture*: Define la textura asociada con cada partícula.
2. *minAngularSpeed*/*maxAngularSpeed*: El rango de la velocidad de rotación angular de cada partícula.

[9] Un sistema de partículas se controla mediante las funciones start y stop.

3. *minSize*/*maxSize*: El rango de tamaños de cada partícula.
4. *minLifeTime*/*maxLifeTime*: El rango de vidas de cada partícula.

### 4.3.4. Beneficios de WebGL en BabylonJS

WebGL le da capacidades 3D al elemento *Canvas* para la mayoría de los navegadores (Mozilla Firefox y Google Chrome lo soportan actualmente). Es una de las herramientas 3D más interesantes referente al desarrollo de videojuegos 3D. Sin embargo, no es fácil de usar. Por ejemplo, para crear un juego se necesitan muchísimas capacidades adicionales como son: la detección de colisiones, las partículas y muchos efectos especiales.

En BabylonJS todas estas y algunas otras funcionalidades más, las encontraremos ya creadas implementadas en WebGL facilitándonos el uso de estas funciones tan costosas de desarrollar.

## 4.4. Objetivo del proyecto

El proyecto busca aprovechar las ventajas de la computación hibrida CPU-GPU desde el navegador web. En este caso, al tratarse de un videojuego tenemos de forma natural esa repartición de tareas por parte del navegador. Es decir, todo lo relacionado con la parte gráfica que se muestra en el monitor se ejecutara en la GPU[10], mientras el resto del código se ejecuta en la CPU.

Además, existe el valor añadido de enfrentarnos a una tecnología muy reciente y poco utilizada como es WebCL. Apostando por esta tecnología contribuimos al enriquecimiento del auge de la computación hibrida CPU-GPU.

Este proyecto se fundamenta en el estudio y análisis real en el desarrollo de aplicaciones web con gran carga computacional y necesitan del uso maximizado del

[10] Hay que mencionar que WebGL utiliza la GPU y es una forma de definir paralelismo de las tareas gráficas.

sistema. Es una demostración de viabilidad de la tecnología pero a la vez un análisis de sus capacidades.

Pero además de implementar una aplicación web y explotar el paralelismo con WebCL, se quiere dar un valor de base como estudio previo para mostrar el potencial de WebCL en dispositivos móviles. Los análisis de rendimiento tienen así una doble lectura, recabar datos sobre la propia aplicación y mostrar la viabilidad de su uso en dispositivos móviles.

# 5 Implementación y arquitectura del proyecto

Se ha seguido una estructura modular, separando las funcionalidades más importantes en macros diferentes, y se ha seguido la misma filosofía para los diferentes recursos que se han utilizado.

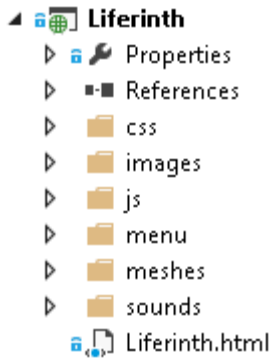



**Imagen 11. Estructura de ficheros.**

## 5.1. Estructura del proyecto

Por un lado tenemos la definición e implementación de la GUI representativa del menú del proyecto, en la cual hay enlaces a las páginas de las diferentes tecnologías que se han usado y una gestión de configuración para el juego que nos permite cambiar características del mismo.

- Carpeta *css*: Contiene el archivo Style.css en el cual están definidos los estilos de los menús.
- Carpeta *menú*: En esta carpeta están las definiciones e implementación de las funcionalidades de cada menú.

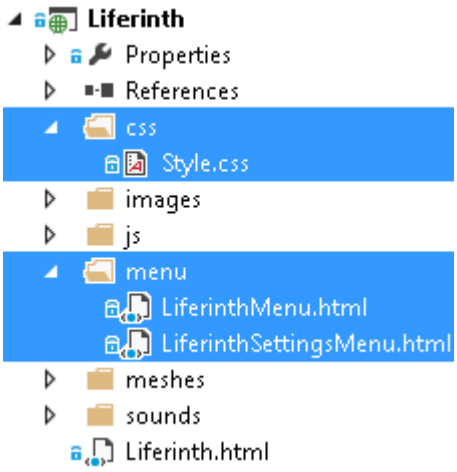



**Imagen 12. Ficheros de estilo y configuración.**

Por otro lado, tenemos los recursos externos que se han empleado en el proyecto, tales como imágenes, sonidos y objetos 3D importados.

- Carpeta *images*: En esta carpeta se encuentran las imágenes que se visualizan tanto en el juego como en los menús.
- Carpeta *meshes*: Contiene la malla de un objeto 3D, que se usa en el juego, creado con la herramienta de modelado 3D Blender.
- Carpeta *sounds*: Contiene los sonidos que se reproducen tanto en el juego como en los menús.

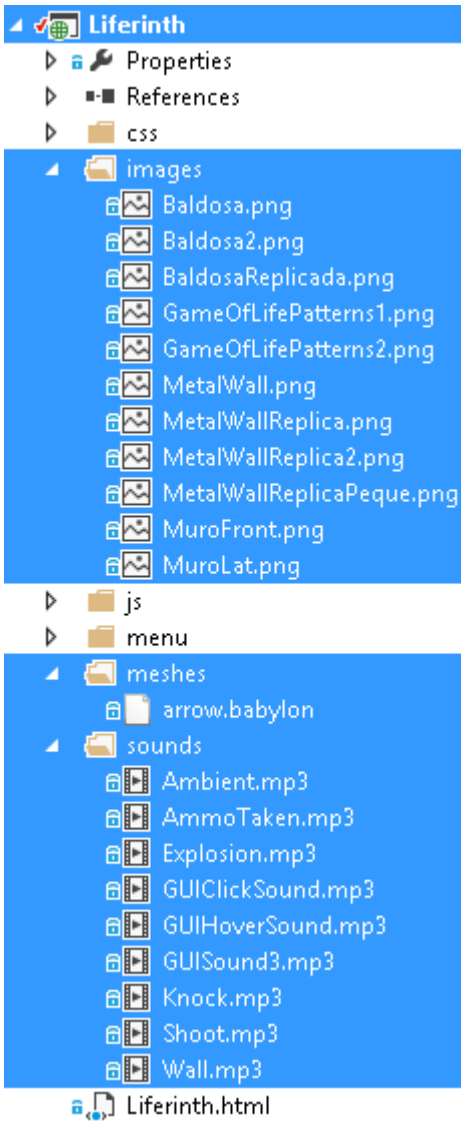



**Imagen 13. Estructura del proyecto, recursos externos.**

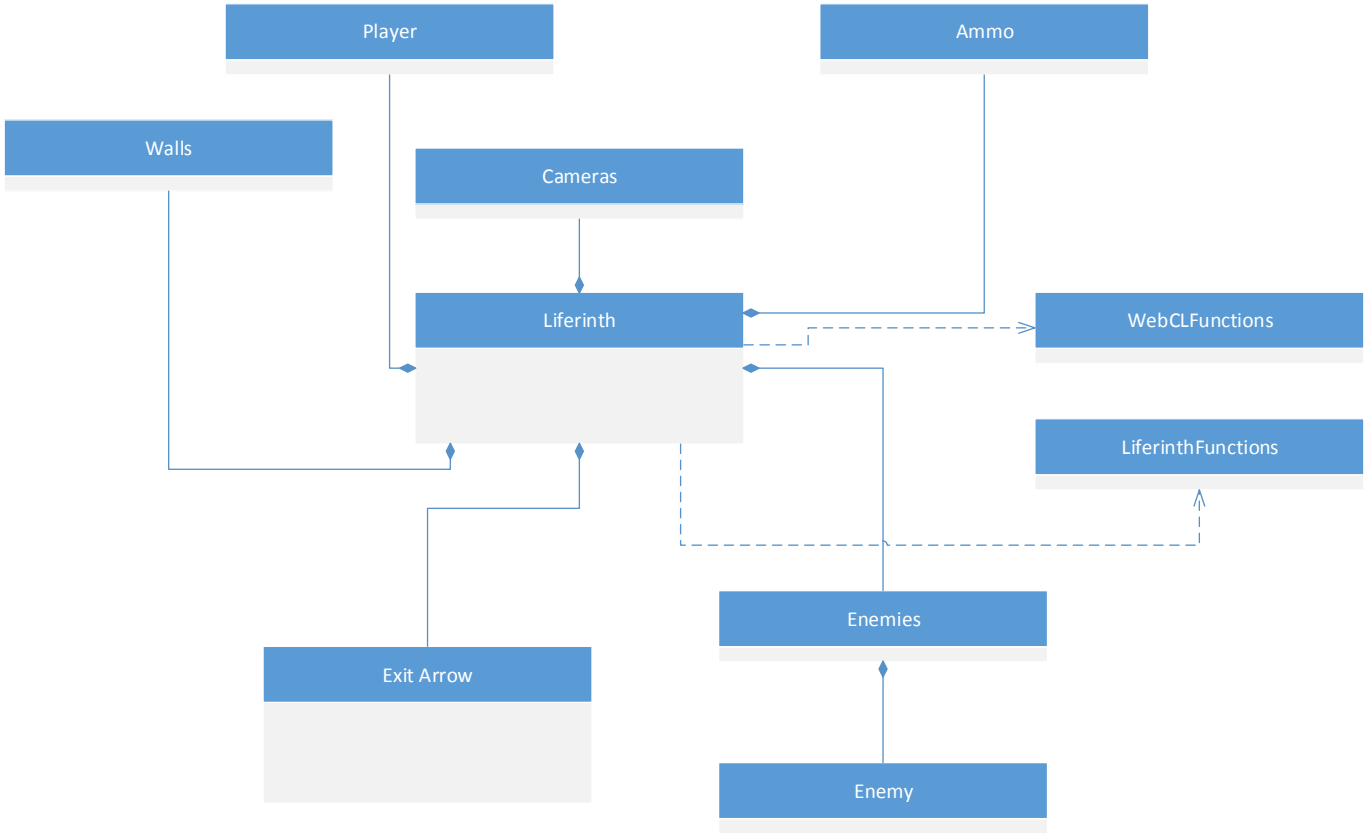
Y, por último, las funcionalidades y el *main* del proyecto, el cual es el encargado de cargar y guiar la interacción entre los diferentes componentes del proyecto.

- Carpeta *js*: Esta carpeta contiene el fichero del *framework* (babylon.js), así como ficheros extra para algunas características del *framework* y uso de funcionalidades especiales de los menús (hand.js y jQuery.js) .Y, por último, ficheros en los que se han definido las funcionalidades de características específicas del juego.
- Fichero *Liferinth.html*: Representa, a grandes rasgos, la estructura principal a nivel de código del proyecto, en la cual se define de forma jerarquizada el correcto uso del *framework* BabylonJS, así como de todas las funcionalidades o características del juego y la carga y gestión de los múltiples recursos externos que se utilizan.


**Imagen 14. Estructura del proyecto, funcionalidades principales.**

## 5.2. Estructura de la aplicación

El siguiente diagrama muestra la relación entre las clases JavaScript utilizadas.




| WebCLFunctions |
|---|
| **public**<br>queue : WebCL.CommandQueue<br>platform : WebCL.Platform<br>device : WebCL.Device<br>globalWorkSize : int<br>localWorkSize : int<br>kernel : WebCL.Kernel<br>kernelSource : Object<br>results : int[][]<br><br>**private**<br>program : WebCL.Program<br>input : WebCL.Buffer<br>output : WebCL.Buffer |
| **public**<br>listDevices(Platform[]) : void<br>getKernel(String) : WebCL.Kernel<br>initWebCLKernel() : void<br>runKernel() : void |

| LiferinthFunctions |
|---|
| No attributes |
| **public**<br>CellContainsPlayer(int, int) : bool<br>AliveNeightbours8(WallState[][], int, int) : int<br>ChangeWallsLife() : void<br>ChangeWalls() : void |

| Exit Arrow |
|---|
| **private**<br>arrow : Babylon.Mesh |
| **public**<br>Initialize() : void |


Liferinth
public
scene : Babylon.Scene
engine : Babylon.Engine
canvas : Object
gravity : Vector3
maxWalls : int
minWalls : int
lastCol : int
lastRow : int
exitRow : int
exitCol : int
entranceRow : int
entranceCol : int
floorHeight : float
plane : Babylon.Mesh
ceiling : Babylon.Mesh
Player : Player
Walls : Walls
Enemies : Enemies
Ammo : Ammo
Cameras : Cameras
Arrow : ExitArrow
private
planeSize : float
activeFlashLight
timeChange : int
volume : float
changeExitFrequency : float
numAmmo : int
light1 : Babylon.HemisphericLight
light2 : Babylon.HemisphericLight
flashLight : Babylon.SpotLight
angFlashLight : float
fogStart : float
fogEnd : float
Ammo
private
MISSILE_SIZE : float
wallWidth : float
planePosXini : float
planePosZini : float
ammo : Babylon.Mesh[]
public
Initialize() : void
Enemies
public
enemies : Enemy[]
numEnemies : int
public
-Nombre del miembro
CreateEnemy(int, int) : void
ManageEnemy(Object) : void
GetCollider(int) : Babylon.Mesh
KillEnemy(int) : void
Cameras
private
numCameras : int
activeCameraNum : int
topDownCameraHeight : float
distanceCamera : float
camera : Babylon.FollowCamera
topDownCamera : Babylon.FollowCamera
completView : Babylon.Viewport
smallView : Babylon.Viewport
public
Initialize() : void
Enemy
public
height : float
width : float
length : float
private
ellipsoid : Vector3
i : int
j : int
position : Vector3
mesh : Babylon.Mesh
collider : Babylon.Mesh
orientation : bool
direction : bool
diagonal : bool
act : int
offsetX : float
offsetZ : float
public
Initialize() : void
Move(Object) : void
GetCollider() : Babylon.Mesh
Dispose() : void
private
RefreshEnemy() : void

## Walls

**public**
WallState : enum
wallScale : Vector3
numWalls : int
totalWalls : int
rows : int
cols : int
walls : WallState[][]
paintedWalls : Babylon.Mesh[][]
binWalls : bool[][]
posMatrix : Vector3[][]

**private**
wallWidth : float
wallHeight : float
wallDepth : float
timeToNoCollisionOnHide : float

---

**public**
CreateDefaultWall() : Babylon.Mesh
CreateWall(int, int, bool) : void
PaintWalls(int, int) : Babylon.Mesh[][]
CopyWalls() : WallState[][]
GeneratePermaWalls() : void
AnimateWalls() : void
CreateMatrix(int, int) : int[][]
RandomWallMatrix() : WallState[][]
PositionsMatrix(int, int) : Vector3[][]
linealToXY(int) : int
fillBinWalls() : void
updateWalls() : void
CreateLabyrinthBounding(int, int) : void
ChangeExit(bool, bool, int) : void
ShowAnimation(int, int) : void
ShowNoAnimation(int, int) : void
HideWall(int, int) : void
ShowWall(int, int) : void
SetPermaWall(int, int) : void
GetExitRow() : int
GetExitCol() : int
UnderneathPlayer(int, int) : bool
CellContainsPlayer(int, int) : bool

**private**
CreateBorder() : void
CreateEntranceAndExit(int, int) : void
AnimateWall(int, int) : void
HideAnimation(int, int) : void
HideNoAnimation(int, int) : void
PlayerTooFarFromWall(int, int) : bool

## Player

**public**
meshPlayer : Babylon.Mesh
collidingBox : Babylon.Mesh
turnLeft : bool
turnRight : bool
forward : bool
backwards : bool
jumping : bool
crouching : bool

**private**
jumpHeight : float
jumpTime : float
defaultSpeed : float
speed : float
backSpeed : float
speedReduceCrouching : float
playerHeight : float
playerWidth : float
playerLength : float
ellipsoidPlayer: Vector3
numLives : int
maxNumLives : int
numLivesDisplay : Object
labelLivesDisplay : Object
MISSILE_SPEED : float
MISSILE_SIZE : float
MISSILE_OFFSET : float
missiles : Mesh[]
directions : Vector3[]
numMissiles : int
maxNumMissiles : int
numMissilesDisplay : Object
labelMissilesDisplay : Object

---

**public**
Initialize() : void
Reset() : void
AnimatePlayer() : void
JumpAnimation() : void
LandAnimation() : void
CrouchAnimation() : void
StrandUpAnimation() : void
TurnLeft() : void
TurnRight() : void
MoveForward() : void
MoveBackwards() : void
DecrementLives() : void
GetNumLives() : int
DisplayInfo() : void
HideInfo() : void
IncrementMissiles() : void
GetMaxMissiles() : int
GetNumMissiles() : int
GetMissiles() : Mesh[]
GetDirections() : Vector3[]

**private**
SetNumOfLives() : void
SetNumOfShots() : void
DisplayLabelInfo() : void
DecrementMissiles() : void

## 5.3. Ejecución del código en la maquina

El encargado de ejecutar el código JavaScript de nuestra aplicación es el navegador y es gracias a los complementos WebGL y WebCL (desarrollado por Nokia) los encargados de gestionar las secciones de código que se ejecutarán en la GPU. El código ejecutable de la aplicación se distribuye entre la CPU y la GPU.

Gran parte de la responsabilidad de la ejecución recae en el programador, pero también es crucial la optimización de las implementaciones de WebCL, WebGL, BabylonJS y del propio navegador Firefox.

Si desde un principio el navegador se ejecuta sólo en la CPU, la razón por la que puede utilizar la GPU son los complementos. Pero hay que tener presente que problemas como las regiones de memoria empleadas por el proceso del navegador pueden ralentizar el trasiego de los datos.

Si para poder ejecutar el código de JavaScript en la CPU no tenemos que preocuparnos de dónde se encuentran nuestros datos y de las dependencias de ejecución, cuando queramos lanzar ejecuciones en la GPU habrá que tener presente que la región de memoria que se tiene acceso desde la CPU es distinta que la GPU.[11] Esto nos obliga a enviar a la GPU los datos que utilice nuestro código, antes de poder ejecutarlo. Y por lo tanto también tendremos que traernos los datos una vez ejecutado el código.

[11] Existen arquitecturas que soportan la unificación del mapa de memoria. No obstante, sigue siendo necesario mover los datos de memoria principal a las memorias de la GPU para minimizar el impacto de las latencias de acceso a memoria.

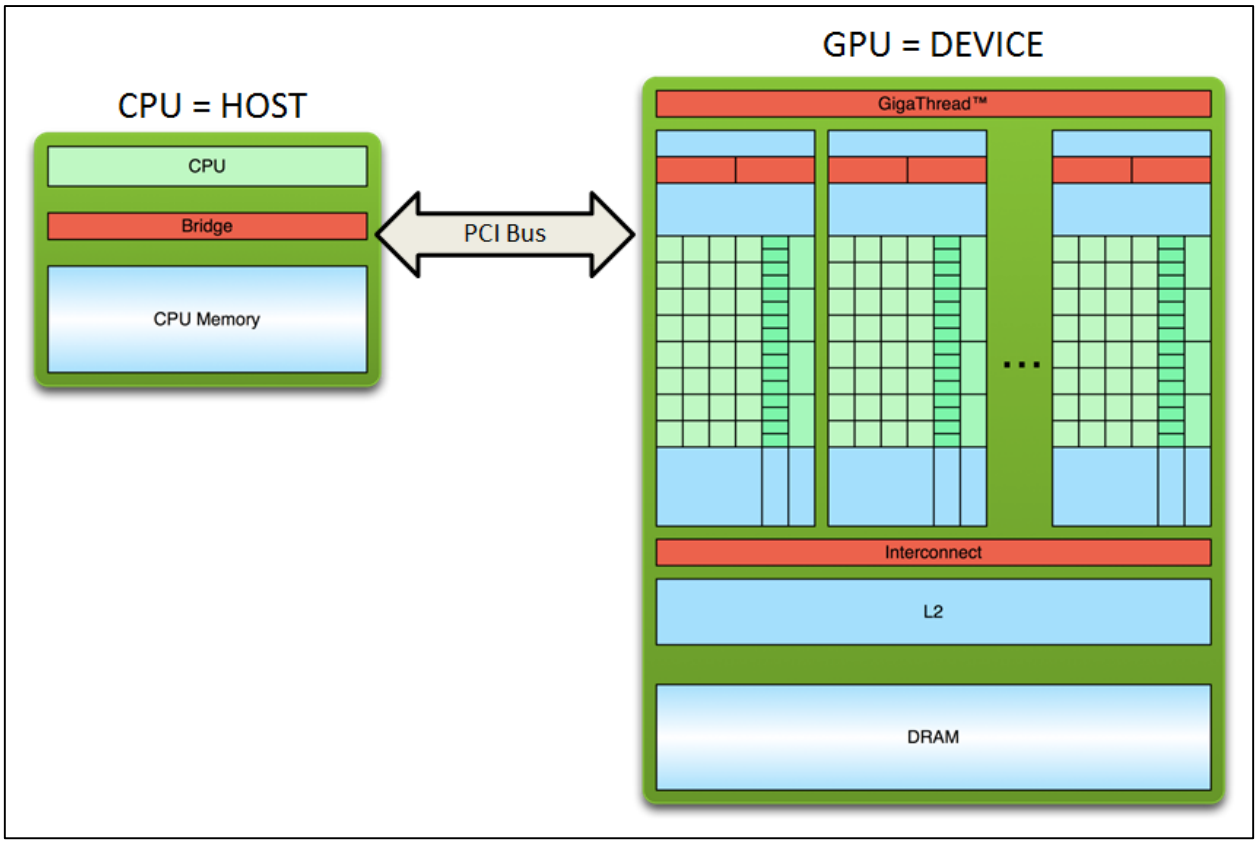



**Imagen 15. Arquitectura habitual del sistema.**

El procedimiento con WebCL sería así:

1. Envío de los datos a la GPU.

   ```
   enqueueWriteBuffer(input, ...);
   ```

2. Fijar el número de hilos que se van a lanzar.

   ```
   globalWorkSize = x;
   localWorkSize = y;
   ```

3. Lanzar la ejecución del kernel.

```
enqueueNDRangeKernel(kernel, globalWorkSize, localWorkSize, ...);
```

4. Esperar a que todos acaben.

   ```
   finish();
   ```

5. Traer los datos de la GPU.

   ```
   enqueueReadBuffer(output, ...);
   ```

# 6 Metodología

Desde la propia gestación del proyecto como tal, la deliberación de posibles aplicaciones qué desarrollar hasta el desarrollo del videojuego pasando por la paralelización.

## 6.1. Plan de desarrollo

La hoja de ruta que se ha seguido en el desarrollo de este proyecto es la siguiente.

### 6.1.1. Bases del proyecto y decisión de la aplicación.

Este proyecto desde sus inicios supuso un reto, tanto para los directores como para nosotros. Una vez acordado el proyecto y el objetivo del mismo, nos quedaba decidir qué aplicación desarrollar. Esta fue la primera decisión más importante. Tras varias posibles propuestas nos decidimos por la realización de un videojuego. Nos pareció la forma más entretenida de afrontar este tipo de proyecto, centrado en el análisis y estudios de rendimiento. También es la forma más amigable de llevar nuestras investigaciones al mayor número de personas.

El videojuego es un laberinto del que se debe salir, pero la dificultad reside en que las paredes del laberinto aparecen y desaparecen modelando el “juego de la vida”.

### 6.1.2. Elección del motor gráfico

La toma de decisión del motor gráfico a utilizar fue un punto muy importante que ha condicionado el rendimiento gráfico el proyecto (al ser el renderizado uno de los puntos más importantes en las aplicaciones 3D). Al estar centrado el proyecto en la computación GPU como encargada de descongestionar el uso de la CPU, era muy

importante escoger un motor gráfico que integrase en su implementación tecnologías de paralelización optimizadas y ejecución GPU, en este caso integrando WebGL.

El primer mes investigamos diferentes motores que probamos ligeramente antes de tomar la decisión definitiva. Los 3 motores considerados comparten las mismas características, son motores 3D de código abierto que usan WebGL y están escritos en JavaScript.

**PlayCanvas**

PlayCanvas es un proyecto de código abierto que es un motor de aplicación 3D y una plataforma de creación en la nube que permite la edición simultánea a través de una interfaz basada en el navegador web. Se ejecuta en los navegadores modernos que soporten WebGL, incluyendo Mozilla Firefox y Google Chrome. El motor permite simulación de la física, la manipulación de audio tridimensional y animaciones 3D.

El motor es compatible con el estándar WebGL para usar la GPU como acelerador de gráficos en 3D y permite el uso de programación JavaScript. Los proyectos pueden ser distribuidos a través de un enlace web URL o dispositivos móviles, por ejemplo, para Android, utilizando CocoonJS.

Las pruebas que hicimos necesitaron de conectividad online y se notaba a la vista un rendimiento bajo entrecortando el movimiento.

**Three.js.**

Es una biblioteca liviana escrita en JavaScript para crear y mostrar gráficos animados por ordenador en 3D en un navegador Web y puede ser utilizada en conjunción con el elemento *canvas* de HTML5, SVG o WebGL.

Se ha popularizado como una de las más importantes para la creación de las animaciones en WebGL. Permite crear complejas animaciones 3D que se muestran en el navegador sin el esfuerzo que se requiere para una aplicación independiente tradicional con un plug-in.

Este motor nos pareció muy bueno. Las pruebas iniciales fueron rápidas y fáciles. Pero pareció ser más pesado que BabylonJS.

**BabylonJS.**

Explicado anteriormente.

Se apostó por este *framework* porque tanto los ejemplos de la web como las pruebas iniciales resultaron muy satisfactorias. Aunque es el más joven y entrañaba un riesgo al no estar tan extendido, nuestra decisión se fundamenta en su agilidad para desarrollar porque el rendimiento fue similar al de Three.js.

### 6.1.3. Versión inicial del videojuego

Después pasamos a desarrollar una primera versión del videojuego. En esta primera versión no se integró toda la funcionalidad final porque lo que se buscaba era obtener un punto de partida estable con el que empezar a paralelizar integrando WebCL y realizar los primeros análisis de rendimiento.

### 6.1.4. Paralelización con WebCL

Con una versión estable, el siguiente paso era buscar las secciones de código que podríamos optimizar. En un estudio inicial fundamentado en la comprensión del flujo de ejecución, se consideró el movimiento de las paredes como un buen candidato a ser optimizado. Tras un análisis donde se veía que el tiempo invertido en esa sección podría reducirse, se pasó a paralelizar el bucle de creación y movimiento de las paredes del laberinto.

### 6.1.5. Añadir funcionalidad completa, ver rendimiento

Continuar integrando todas las funcionalidades. Realización de test para ver las latencias, el uso de recursos o el consumo de la aplicación. Identificar problemas y buscar optimizaciones.

### 6.1.6. Comparativa entre proyecto con WebCL frente otro sin WebCL

Con el proyecto materializado se decidió deshacer la paralelización generando una nueva rama del proyecto, igual a la raíz con la salvedad de no tener secciones de código WebCL. Finalizando con un análisis comparativo entre ambas ramas que justifique el esfuerzo en desarrollar con WebCL.

## 6.2. Áreas de desarrollo

Identificamos 3 áreas que engloban las tareas del proyecto. Una tarea de un área no tiene sentido sin su réplica en las otras dos. Es decir si se implementa el movimiento de las paredes (perteneciente al área de Aplicación) tendremos otra tarea de análisis de rendimiento (perteneciente al área de Métricas) y otra que buscará la optimización, de ser posible, con WebCL (perteneciente al área de Optimización).

### 6.2.1. Área de Aplicación

En nuestro caso el videojuego. Aquí entran aspectos como la funcionalidad, la apariencia y estética, el objetivo del personaje, la estrategia de movimiento de las paredes, diseño de la aplicación. También podemos encuadrar: las habilidades del personaje, los enemigos, del desarrollo del videojuego en definitiva…

### 6.2.2. Área de Optimización

Se puede ver como una reescritura del código para sacar el máximo partido de la máquina. En este caso ayudados con WebCL para explotar el paralelismo residente en nuestro código.

### 6.2.3. Área de Métricas

Estudios de Rendimiento y análisis de los datos. Con los que nos ayudábamos para tomar decisiones sobre las secciones de código a paralelizar o qué mejoras de rendimiento obteníamos, viendo así si se obtenían mejoras significativas por las acciones realizadas.

## 6.3. Herramientas de desarrollo

Estas han sido las distintas herramientas utilizadas.

### 6.3.1. Visual Studio

Es un entorno de desarrollo de aplicaciones que soporta multitud de lenguajes de programación; C++, C#, Java, JavaScript o PHP entre otros. Es una herramienta de

Microsoft para sistemas Windows que permite la programación, compilación y ejecución bajo una apariencia de ventanas.

### 6.3.2. GitHub

Es un repositorio para proyectos que utiliza el sistema de control de versiones Git. Se ha desarrollado bajo una licencia de código abierto GPLv3. El proyecto se puede encontrar en el repositorio de GitHub[12] con el nombre de “Liferynth”[13].

### 6.3.3. Blender

Programa para el modelado, animación y creación de gráficos 3D. Es un proyecto de software libre aunque inicialmente no se distribuyó el código fuente. Muy útil para la creación de elementos 3D. Además permite la creación de videojuegos gracias a un motor de juegos interno.

[12] https://github.com/alejandroladrondeguevara/Liferynth

[13] Aunque el nombre de la aplicación es Liferinth al crear el proyecto hubo cierta confusión y quedo equivocado.

# 7 Resultados

Como ya se ha mencionado este proyecto se fundamenta en el estudio y análisis de las capacidades de la computación híbrida. En primer lugar el uso de la GPU para la aceleración de la ejecución del videojuego. Pero también con la intención de migrar los resultados a los dispositivos móviles.

El estudio y análisis del rendimiento ha sido la piedra angular de este proyecto. Han sido frecuentes las verificaciones de las métricas con cada nueva actualización o modificación importante.

Nos hemos ayudado de complementos de Firefox tanto para realizar los estudios de rendimiento como para depurar la ejecución del código.

## 7.1. Herramientas

Para el estudio y análisis continuado, y en paralelo, que hemos ido llevando a cabo durante el desarrollo del proyecto, se han seguido distintas técnicas explicadas a continuación.

### 7.1.1. Temporizadores empotrados en el código

Esta es la manera más rudimentaria que existe de analizar el rendimiento de la aplicación. Se trata de incorporar en el código fuente temporizadores para calcular los retardos que intervienen en una sección de código determinada. Es una medida utilizada como doble comprobación tras los datos obtenidos con otras herramientas. Por ejemplo añadiendo:

```
performance.now();
```

### 7.1.2. Firebug

Es un complemento de Firefox para la depuración de código desde el propio navegador. Permite muchas formas de analizar la ejecución del código pero su mayor utilidad radica en la búsqueda de errores en el código.

### 7.1.3. Cleopatra

Es un complemento de Firefox que nos muestra, gracias a un análisis en ejecución del tiempo, la latencia y los recursos que se lleva cada sección del código que se ejecuta en el navegador. Podemos diferenciar entre: pestañas, complementos instalados o, como en nuestro caso, únicamente el código JavaScript de la aplicación.

## 7.2. Gráficas de rendimiento del movimiento de las paredes

El código relacionado con las paredes, tanto la generación como el movimiento y en igual manera el renderizado, al afectar el número de muros, tiene un peso importante en nuestra aplicación. Y los siguientes análisis buscan estudiar el impacto de cada característica.

Como se puede apreciar en esta gráfica, el tiempo de ejecución del algoritmo del “juego de la vida” haciendo uso de código JavaScript (en verde) es proporcional al número de muros del laberinto. Se puede ver la semejanza de la curva con una función lineal representada en rojo (y = número de muros / 10). Este resultado coincide con lo esperado de una ejecución secuencial: a mayor número de cálculos, mayor tiempo de ejecución, ya que tiene que acabar un cálculo antes de realizar el siguiente (sumándose sus tiempos de ejecución).

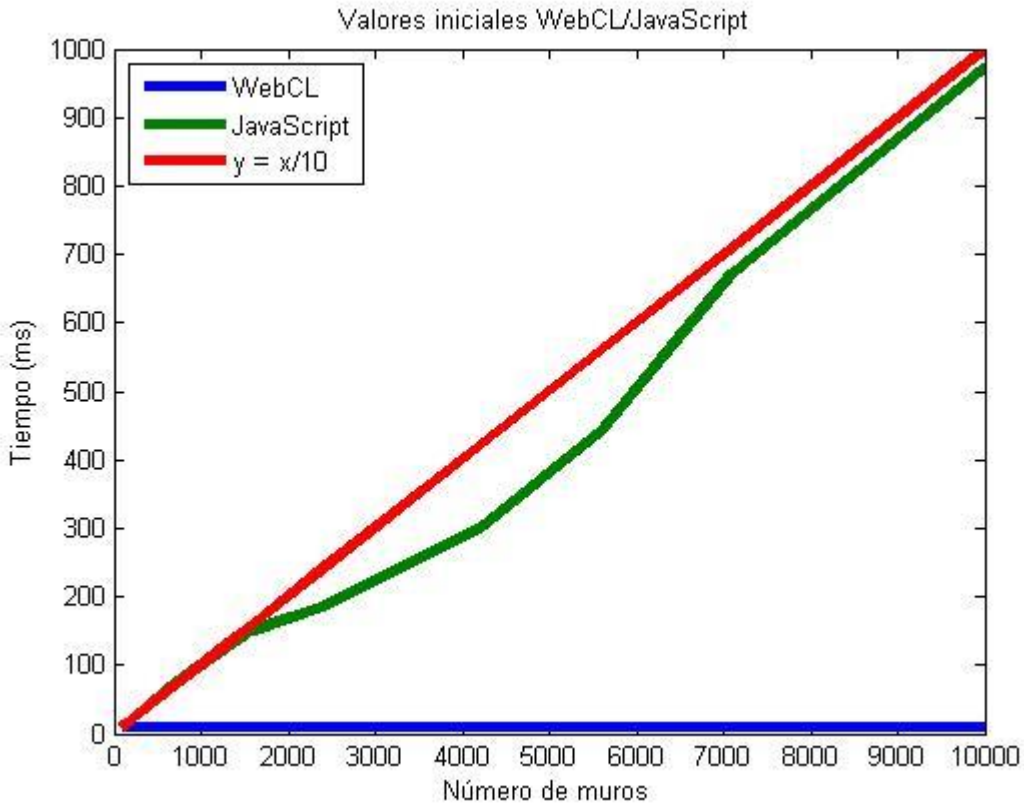



**Imagen 16. Valores iniciales WebCL/JavaScript.**

En contraposición, se observa también el tiempo de ejecución del mismo algoritmo pero, en este caso, con código WebCL (en azul). El resultado es diferente: a mayor número de muros, mismo tiempo de ejecución (alrededor de 9 ms[14]). Esto también concuerda con lo esperado al ejecutar código de forma paralela. Al realizarse todos los cálculos al mismo tiempo, es indiferente el número de cálculos que haya que hacer (siempre que no se exceda el número máximo de cálculos simultáneos soportados).

Cabe destacar además que, cuanto mayor es el número de muros (más cálculos), la ejecución con WebCL produce mayor aumento de rendimiento.

## 7.3. Características de los recursos que se pueden explotar

Para un análisis más exhaustivo en tiempo real de la ejecución de la aplicación hemos recurrido a un complemento de Firefox llamado Cleopatra [5]

[14] Considerando sólo el computo en GPU, no el envío de datos CPU-GPU.

La siguiente captura muestra  los porcentajes y cargas de trabajo que se envían a la GPU usando WebCL. Es importante notar que estamos viendo exclusivamente el uso de la GPU por parte del código que hemos paralelizado. Para entender cómo se reparten entre ellos la carga de trabajo. Aislar las pruebas es muy importante a la hora de identificar errores en la ejecución o ideas equivocadas[15].

Si en las gráficas anteriores comparábamos el retardo modificando el número de muros del laberinto, en la siguiente grafica podemos ver el impacto que tiene el código WebCL sobre el navegador. Podemos ver que el 74.4% del tiempo dedicado al movimiento de los muros se lo lleva la ejecución del kernel en la GPU y que el resto es el retardo por el trasiego de datos entre la CPU y la GPU.

[15] Ideas equivocadas provocadas por errores de concepto relacionados con la ejecución WebCL o aplicar suposiciones incorrectas.

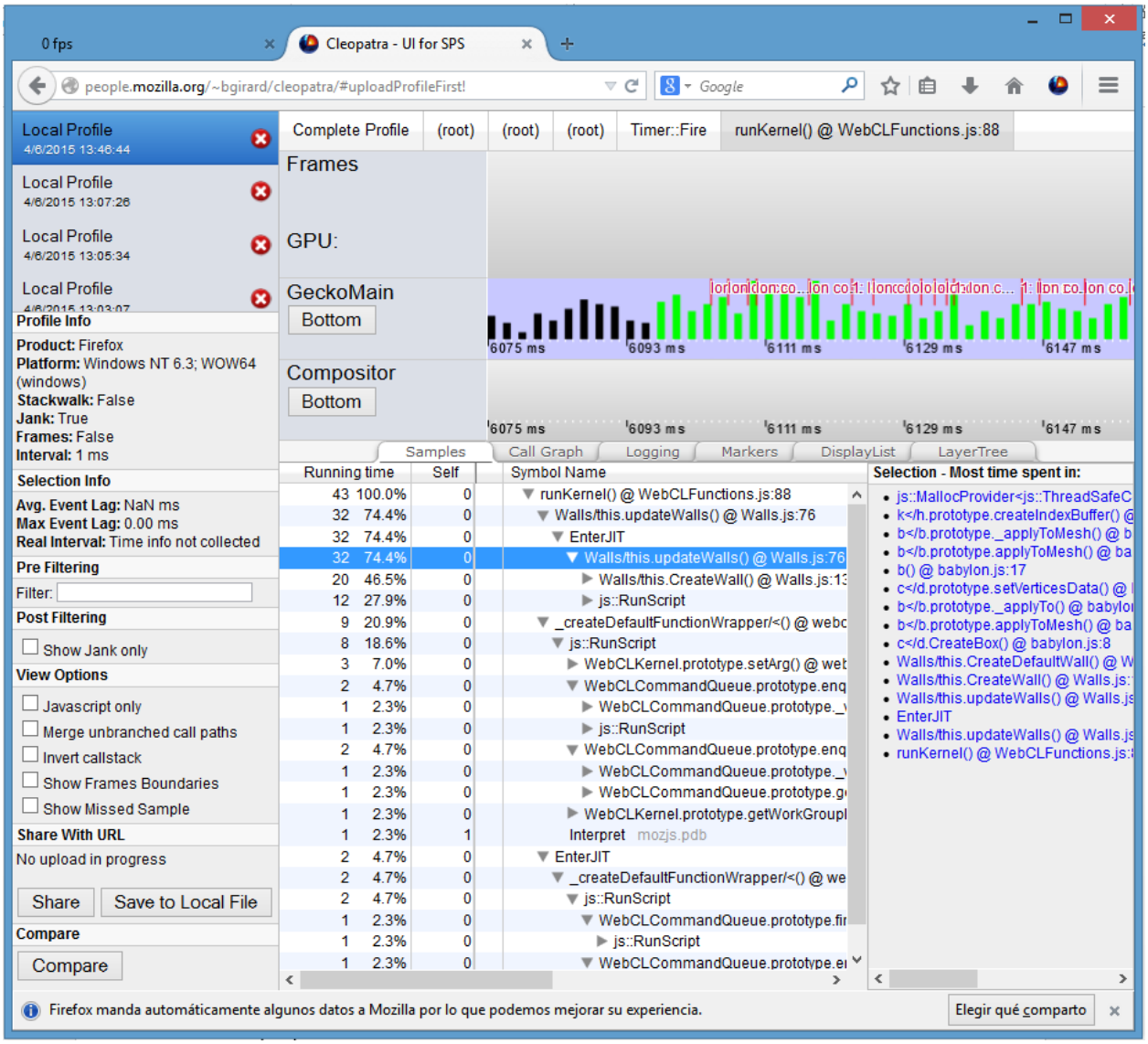



**Imagen 17. Cleopatra carga de trabajo WebCL.**

De ahí la importancia de llevar ejecuciones muy costosas en cálculo y que no requieran de mucho trasiego de datos. En nuestro caso tiene un porcentaje elevado pero frente a la CPU su ventaja será la localidad de los datos.

En esta otra gráfica vemos el efecto que produce en la ejecución secuencial localizada en la CPU (JavaScript, gráfica verde) la jerarquía de memoria. Hay que notar también que la ejecución paralela en la GPU (WebCL, gráfica azul) no es constante. En este caso es la sincronización final de los hilos en la GPU la que lo produce, ya que la ejecución puede entrelazarse con otras tareas residentes en la GPU.

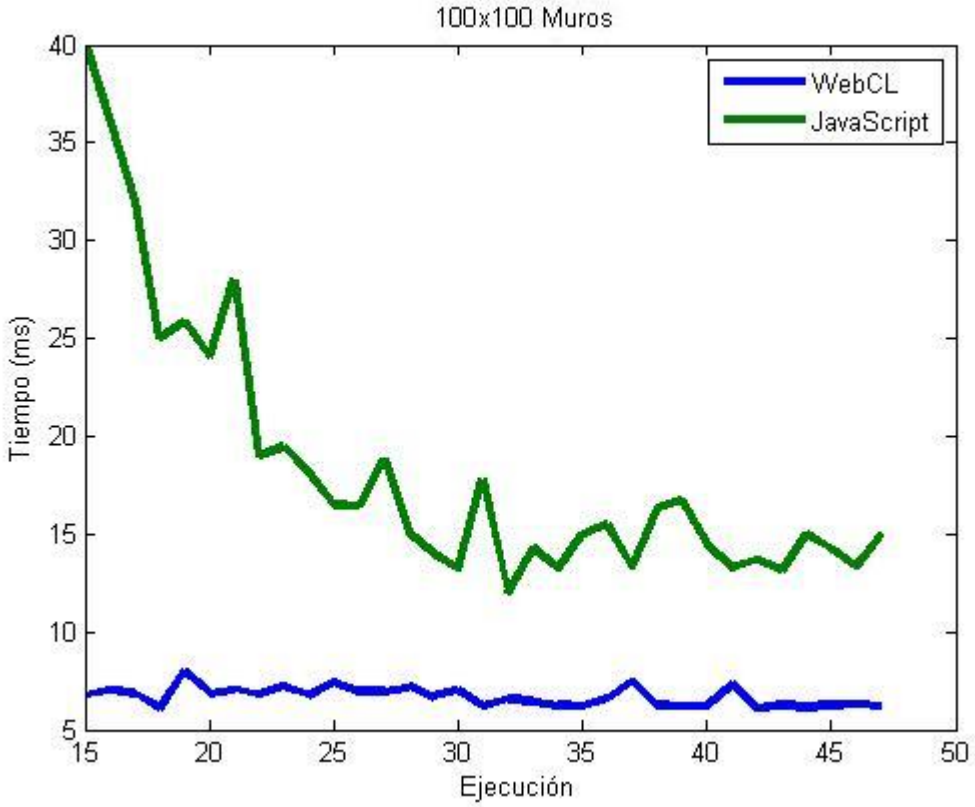



**Imagen 18. 100x100 Muros efecto jerarquía de memoria en la CPU.**

También podemos ver el uso respectivo de forma global. En este caso contabilizando la carga total existente en el navegador. Podemos ver el impacto de los distintos módulos de código JavaScript. En este ejemplo vemos que el renderizado interviene en un 2.2 % y es que hay que considerar que durante esta prueba no se movió el personaje salvo como puede apreciarse, al final de la prueba. Y cada pico de cómputo corresponde al cálculo de la variación de los muros. En este caso localizada en la GPU y con una latencia media de 359 ms[16].

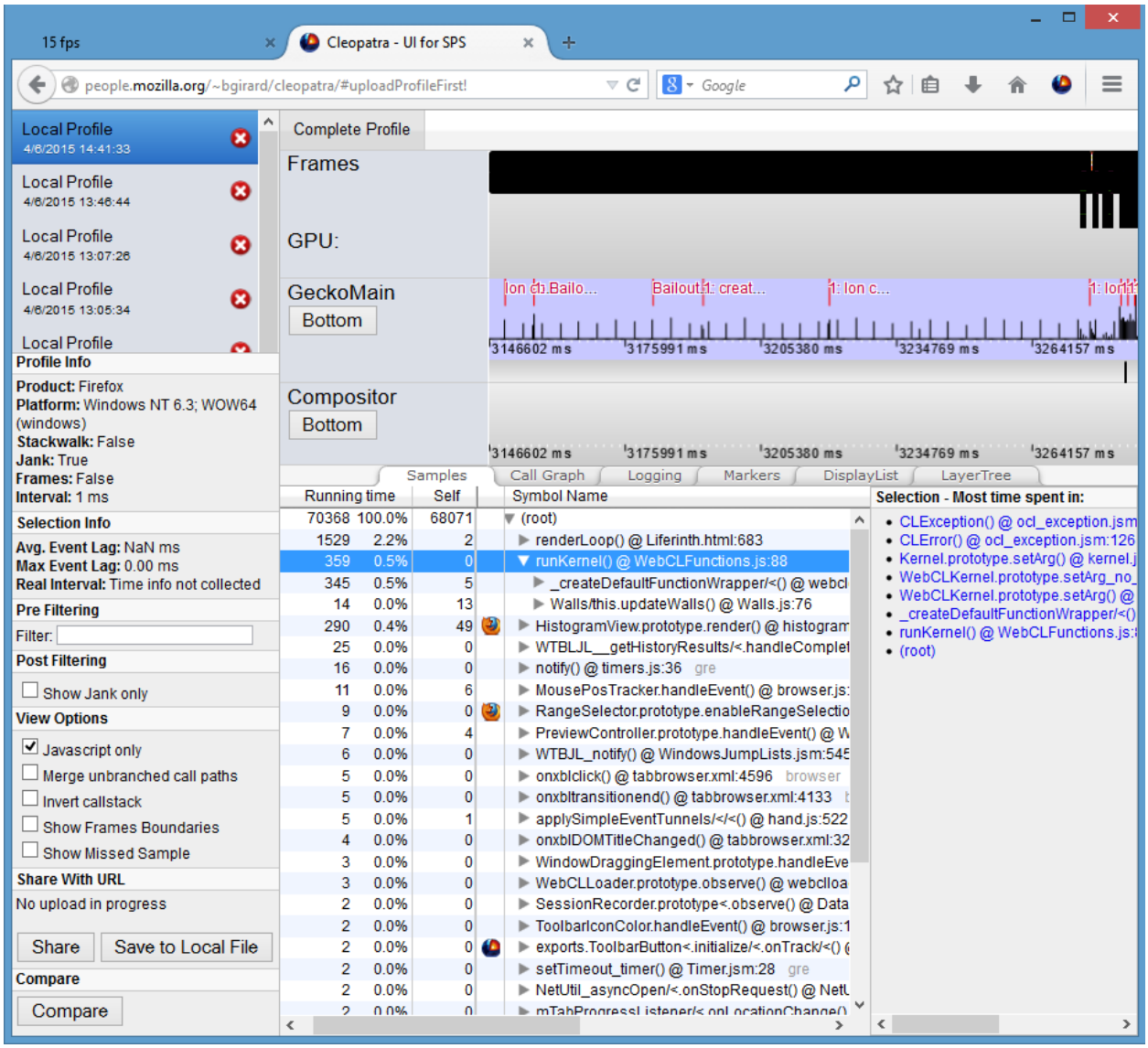



**Imagen 19. Cleopatra tiempo invertido en el movimiento de los muros del laberinto.**

[16] Contabilizando el envío de los datos CPU-GPU.

En la siguiente podemos ver una prueba más realista donde se ve el impacto del renderizado en la aplicación, el 74.6% del tiempo.

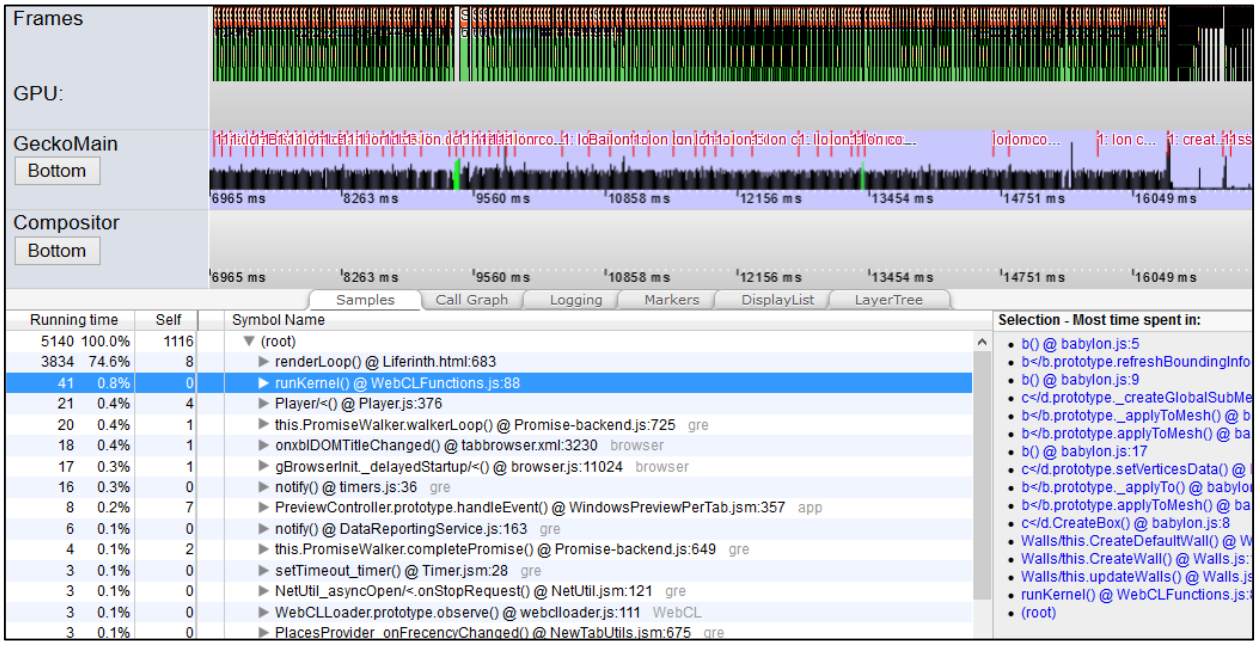



**Imagen 20. Cleopatra carga real del renderizado frente al movimiento de los muros.**

En este caso, si nos fijamos, la región de WebCL residente en la GPU tiene un impacto global del 0.8% traducido en 41 ms[17]. Tras estos datos podemos ver que la mayor carga computacional reside en el renderizado del videojuego. El problema lo producen el número tan alto de muros que se encuentran en la escena.

## 7.4. Paralelización

La paralelización ya explicada anteriormente es una de las piedras angulares de la actualidad. En este proyecto ha sido pieza fundamental y que aparece recursivamente. Pero este proyecto no se centra exclusivamente en el uso de algoritmos de paralelización, sino también en mostrar la versatilidad de la paralelización en cualquier tipo de aplicación.

Hay que tener presente que el navegador web tiene ciertas limitaciones a la hora de utilizar los recursos del sistema. No se tiene un acceso a los dispositivos tan directamente como pudiera hacerse desde un programa escrito en C y compilado para

[17] Contabilizando el envío de los datos CPU-GPU.

ese sistema. En nuestro caso, el navegador hace uso de los APIs instalados previamente en el propio navegador (WebCL, WebGL) y que soporte el equipo (OpenCL si se tuviese CPU many-core).

Hemos abordado la paralelización tras versiones completas de la aplicación a las que hemos realizado estudios con diversas herramientas de *profiling*.

## 7.5. Algoritmo paralelizado

La modelización del movimiento de las paredes, empleando una versión propia del algoritmo del "juego de la vida".

Partiendo de la primera versión del algoritmo, dedicada a analizar de forma secuencial, pared por pared, consultando el valor de vida de sus adyacentes, hemos elaborado una versión equivalente integrando WebCL.

Siguiendo las reglas cada pared podría modificar su estado en el tablero dependiendo de la configuración de sus vecinas. Además, sin necesidad de conocer cuál será el estado al que pasarán en el siguiente ciclo. Esto nos dice que el estado del tablero se utiliza sólo como lectura por cada pared.

Si a cada pared le proporcionamos la información de su vecindario. Esta puede decidir cuál será su estado en el próximo ciclo. Siendo totalmente independientes.

Con la información del tablero actual como entrada lanzamos un kernel que compute el algoritmo. En este caso consideraremos el número total de paredes y la capacidad de cómputo de la GPU para calcular el número de hilos que tendremos que configurar en el kernel para poder lanzarlo.

```
        workGroupSize = kernel.getWorkGroupInfo(device,
    webcl.KERNEL_WORK_GROUP_SIZE);

        globalWorkSize[0] = count + workGroupSize - (count
    % workGroupSize);

        localWorkSize[0] = workGroupSize;
```

**Imagen 21. WebCL tamaño del grupo de trabajo.**

Una vez calculado podemos lanzar la ejecución del kernel.

El kernel computa el movimiento de las paredes como el “juego de la vida”. Comprobando el estado de su vecindario para decidir su estado. El algoritmo es el mismo que se ejecuta de forma secuencial aplicado a una pared.

```
int x = floor((float) (i / cols));
int y = i % cols;
for ( int r = x - 1; r <= x + 1; r++)
    for ( int c = y - 1; c <= y + 1; c++)
        if ( r != x && c != y )
            if ( r >= 0 && r <= lastRow && c >= 0 && c < lastCol ) {
                int aux = c + r * cols;
                if ( input[aux] == 1) n++;
            }
if( n == 3 && input[i] == 1 ) output[i] = 1;
else if ( n > 3 && input[i] == 1 ) output[i] = 0;
else if ( n < 2 ) output[i] = 0;
else if ( n == 2 ) output[i] = 1;
```

**Imagen 22. Código movimiento de las paredes.**

# 8 Conclusiones y trabajo futuro

Tras el trabajo de todo un año invertido en este proyecto hemos elaborado una serie de conclusiones sobre los análisis realizados y una lista de posibles ampliaciones.

## 8.1. Conclusiones

Todo el trabajo invertido en este proyecto tenía gran parte de investigación, mirando como objetivo futuro la posible implantación de esta tecnología en dispositivos portátiles. La intención fundamental de la obtención del rendimiento era el estudio de su viabilidad. Y por las pruebas realizadas es una alternativa muy interesante que considerar que nos permitiría ahorrar energía además de ganar en velocidad.

El desarrollo de este proyecto es un ejemplo del potencial que se puede exprimir desde el navegador web utilizando WebCL. Entendiendo bien las características específicas de cada aplicación, se puede sacar mucho rendimiento desde el navegador sin necesidad de crear aplicaciones nativas para cada tipo de dispositivo.

Uno de los aspectos más críticos en la realización de proyectos de este estilo, es el colapso de la GPU. Hay que tener en cuenta que el tratamiento gráfico y parte de nuestro código lo vamos a llevar a ejecutar a la GPU y no todos los proyectos se pueden enfocar en este sentido. Un videojuego es un ejemplo en el que hay que tener muy presente el porcentaje que va a necesitar para las actividades gráficas propias, como el renderizado, para no llevarnos a engaños si buscamos enviar mucho cómputo a la GPU.

Para el renderizado sería conveniente realizar un estudio más exhaustivo sobre las necesidades requeridas por la aplicación. Como en nuestro caso fue un videojuego, se convirtió en un punto fundamental. Pero en otro tipo de proyectos, que no requieran de animación 3D, se podría usar en mayor medida la GPU sin tener pérdidas notables de rendimiento.

Pero hay que tener muy presente las limitaciones que conlleva el uso de aplicaciones para navegadores, como es la necesidad de tener conectividad a internet, el uso del navegador concreto bajo las versiones específicas junto con los complementos necesarios instalados. Aunque es cierto, que se han dado varios avances en la integración de estas tecnologías en los navegadores, actualmente puede no ser una alternativa viable para proyectos que busquen llegar al gran público.

Una buena implementación con WebCL hace posible la migración de código a otras tecnologías paralelas con poco esfuerzo. La integración de estándares de paralelización está cada vez más demandado y permite extender la vida útil de nuestras aplicaciones al poder escalar junto con las máquinas que las ejecutan.

## 8.2. Trabajos futuros

Actualmente no todos los dispositivos móviles cuentan con unidades GPU, pero el consumo eléctrico es uno de los puntos fundamentales de su utilización. Como la GPU tiene un consumo mucho menor que la CPU, comienza a integrarse, cada vez más, en todos los dispositivos. Y el reto actual es el uso por parte de los programadores. No sólo que implementen secciones de código paralelizable sino su correcta integración sin dependencias arquitectónicas.

La implementación optimizada de funciones de BabylonJS u otro motor gráfico como la detección de colisiones, sería uno de los aspectos más destacables para un proyecto futuro. Aunando el potencial de la paralelización y haciéndolo trasparente para el programador.

Siempre desde una perspectiva de la utilización, es muy importante obtener implementaciones optimizadas y tener presente en todo momento el uso de los recursos que estamos haciendo de la máquina. La toma de decisiones por parte

de la propia aplicación de ejecutar el código en CPU o en GPU dependiendo de la demanda en ese momento es una característica que se quiso entregar en la aplicación pero cuyo impacto era mayor del esperado y se decidió no implantar. Sería una característica muy interesante a desarrollar en el futuro.

# Apéndice A. Instalación WebCL en el navegador Mozilla Firefox[18]

Para poder ejecutar la aplicación del proyecto es necesario contar con un Mozilla Firefox entre las versiones 30 a 35, la utilizada por nosotros es la 34, utilizada durante todo el proyecto.

## 9.1. Instalación Mozilla Firefox Portable v.34

Acceda a la dirección online:

http://mozilla-firefox-portable.en.uptodown.com/download/92825

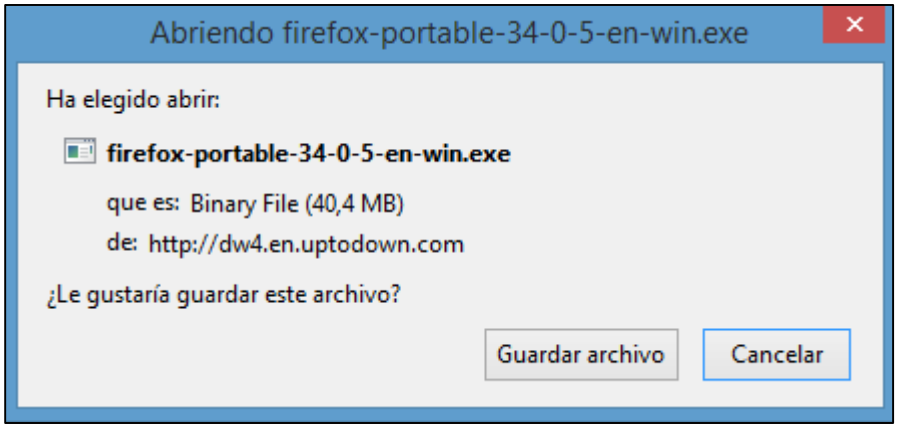



**Imagen 23. Ventana para guardar el archivo de instalación.**

Seleccione “Guardar archivo”. Se descargará un ejecutable para instalar el navegador.

[18] Estas instrucciones se pueden encontrar en la web de Nokia de WebCL: http://webcl.nokiaresearch.com/index.html. [6]

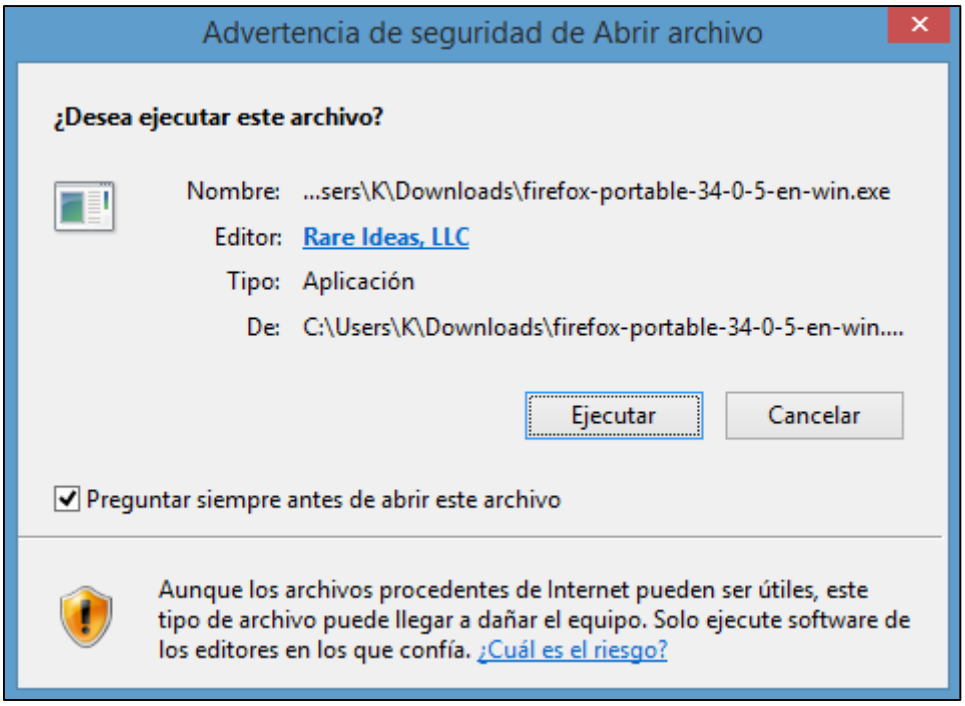



**Imagen 24. Mensaje de advertencia.**

Ejecuta el archivo descargado y sigue los pasos de instalación.

## 9.2. Instalación de WebCL

Una vez instalado el navegador tendremos que instalar el plug-in de WebCL con la implementación de Nokia. Acceda a la siguiente url:

http://webcl.nokiaresearch.com/extensions/firefox/multiplatform/latest/webcl-1.0.xpi

Le saltara un mensaje similar a este:

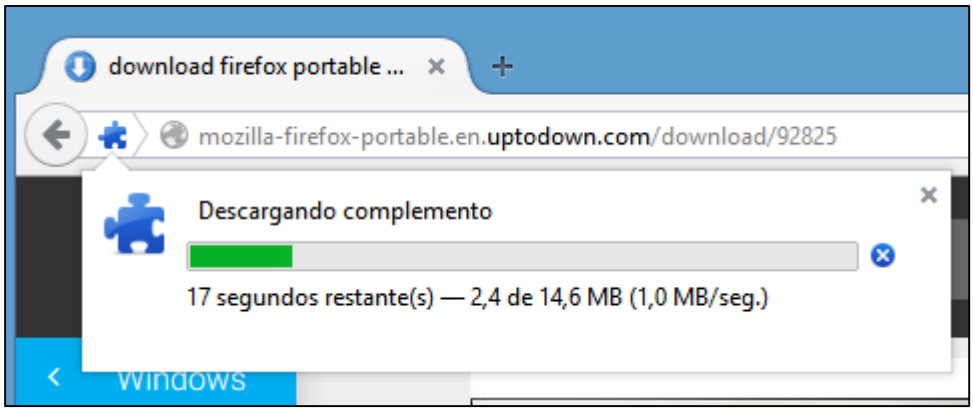



**Imagen 25. Descarga del complemento WebCL.**

Cuando finalice.

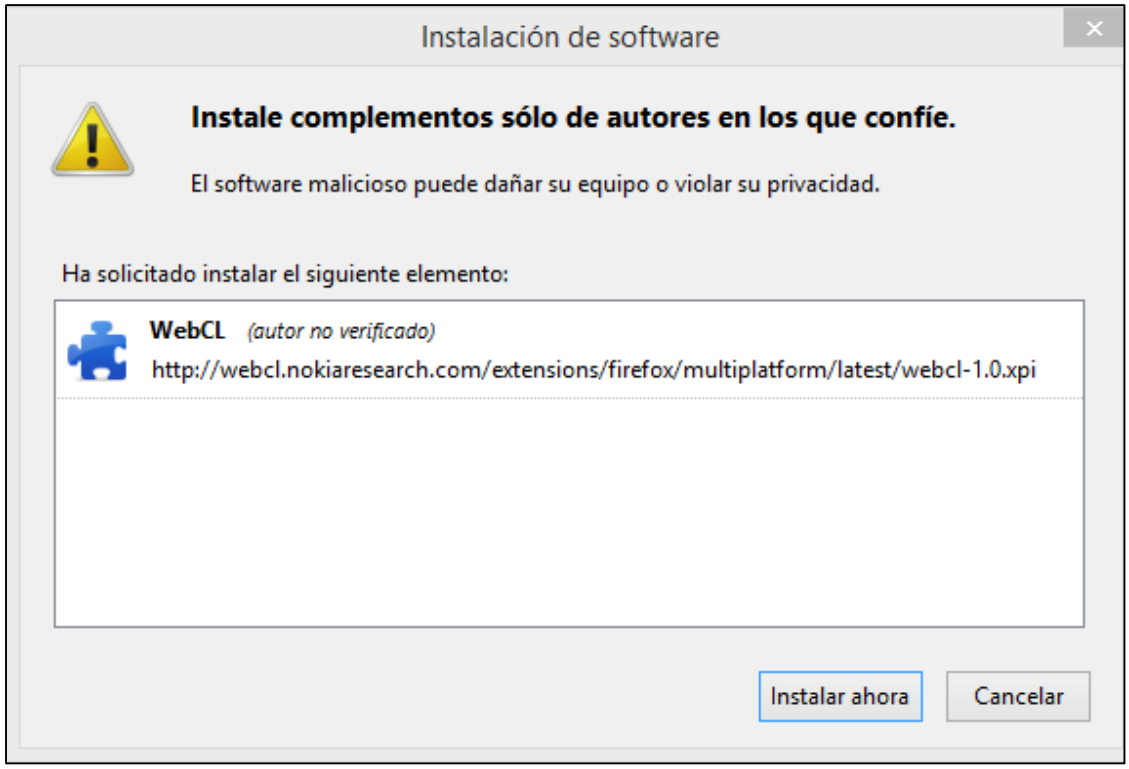



**Imagen 26. Instalación del complemento.**

Le pedirá permisos para instalar, pulse “Instalar ahora”.

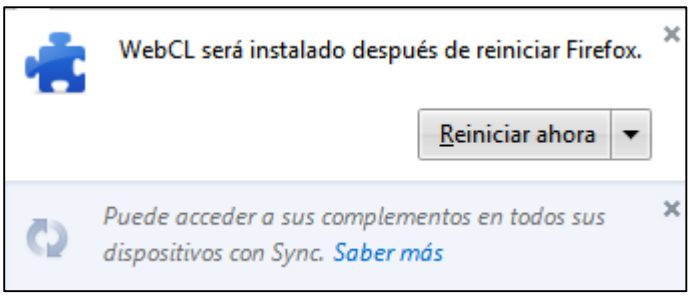



**Imagen 27. Mensaje de reinicio del navegador.**

Después reinicie el navegador.

En este caso el navegador tiene preinstalado WebCL, por lo que no es necesario realizar ninguna acción más. Ya se puede utilizar la aplicación del proyecto sin problemas.

# Apéndice B. Tutorial de la aplicación

La aplicación se puede encontrar en el repositorio de GitHub con el nombre de “Liferynth”[19].

## 10.1. Requisitos

- Mozilla Firefox v34.[20]
- Complemento WebCL implementación Nokia.
- Complemento WebGL (Integrado en Firefox v34).
- Tarjeta Gráfica.

## 10.2. Videojuego

La aplicación consiste en un videojuego al que hemos llamado “**Liferinth**”. El usuario controla un personaje en tercera persona cuyo objetivo es salir de un laberinto dinámico, modelado como el “juego de la vida”.

El usuario cuenta con una flecha que señala la salida del laberinto. Pero, ¡Ojo! La salida también puede cambiar de ubicación. El personaje cuenta además, con la capacidad de disparar misiles que le permiten matar a los enemigos y derribar paredes dinámicas.

---

[19] Aunque el nombre de la aplicación es Liferinth al crear el proyecto hubo cierta confusión y quedo equivocado.

[20] Ver tutorial de instalación en Apéndice A. Instalación WebCL en el navegador Mozilla Firefox.

Debe evitar ser tocado por los enemigos. Estos, si le tocan, le quitarán una vida. Al quedarse sin vidas se acaba el juego.

Además, el número de proyectiles es limitado, pero el personaje puede recoger munición que se encontrará en el laberinto.

### 10.2.1. Controles del videojuego:

| Tecla | Acción |
|---|---|
| W/w/↑ | Adelante |
| S/s/↓ | Atrás |
| A/a/← | Izquierda |
| D/d/→ | Derecha |
| E/e | Disparar |
| J/j | Cambiar cámara |
| F/f | Activa/desactiva linterna |
| “Espacio” | Saltar |
| “Control” | Agacharse |

### 10.2.2. Comprobación WebCL instalado

Antes de comenzar a jugar veremos un mensaje que nos informará si nuestro navegador soporta WebCL.

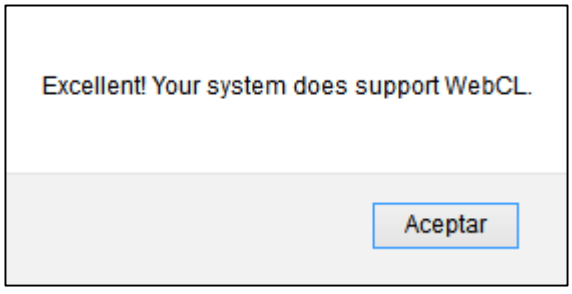



**Imagen 28. Mensaje Sí soporta WebCL.**

O no lo soporta.

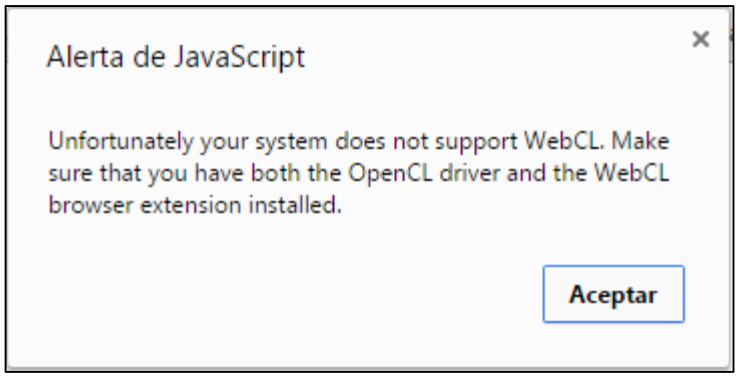



**Imagen 29. Mensaje NO soporta WebCL.**

### 10.2.3. Menú

A continuación se muestra una captura del menú de la aplicación:


**Imagen 30. Menú de la aplicación.**

Este menú nos permite acceder de forma rápida a información relativa al proyecto: el repositorio del proyecto o las páginas más importantes de las tecnologías que forman parte del proyecto.

En "*Settings*" podremos configurar aspectos técnicos que afectan al rendimiento del juego, tal y como puede verse a continuación:

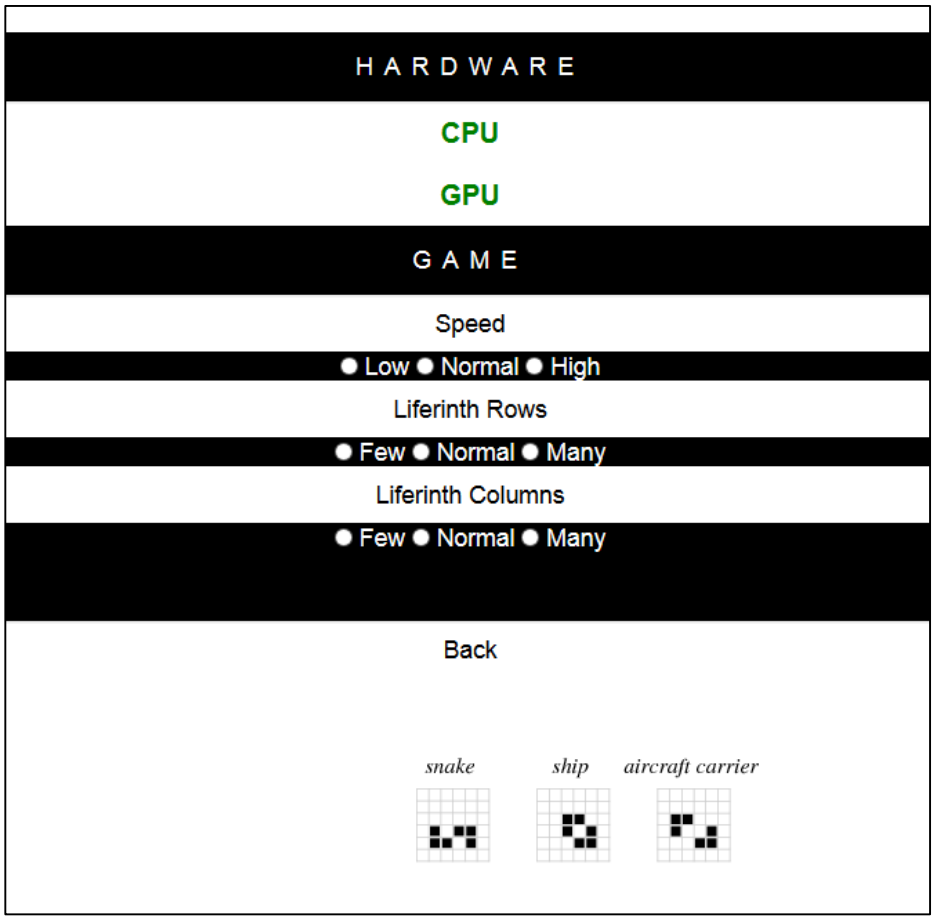



**Imagen 31. Menú "Settings".**

### 10.2.4. Enemigos

En el laberinto te encontrarás con enemigos que tendrás que esquivar. Si te tocan te quitaran una vida y si te quedas sin vidas se acaba el juego. A continuación mostramos una captura donde vemos un enemigo.

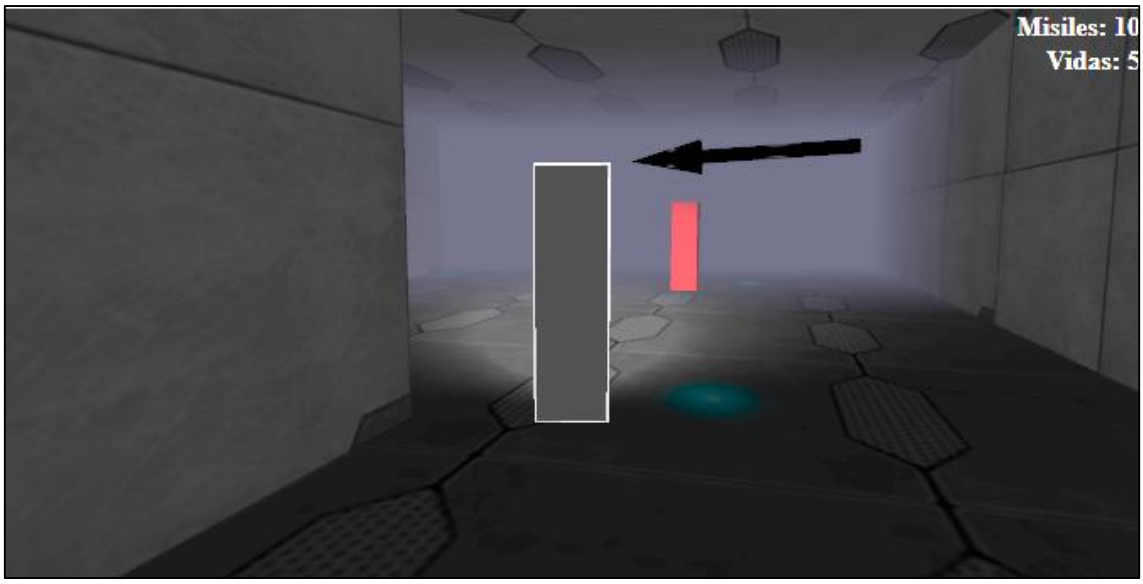



**Imagen 32. Captura del videojuego. Enemigo.**

### 10.2.5. Munición

Pero el personaje puede matar a los enemigos disparándoles un misil. Comenzamos la partida con 10 misiles que es el número máximo de misiles que podemos tener. Podemos disparar los misiles para matar a los enemigos o para tumbar paredes dinámicas. No existe diferencia visual entre paredes dinámicas y estáticas pero tras varios ciclos de vida de las paredes, notaremos que existen paredes que no suben ni bajan. Sera inútil disparar contra ellas.

Podremos recargar munición, cogiendo misiles que aparecen y desaparecen por el laberinto a su antojo. En la siguiente imagen se muestra un misil de munición a la derecha del personaje:

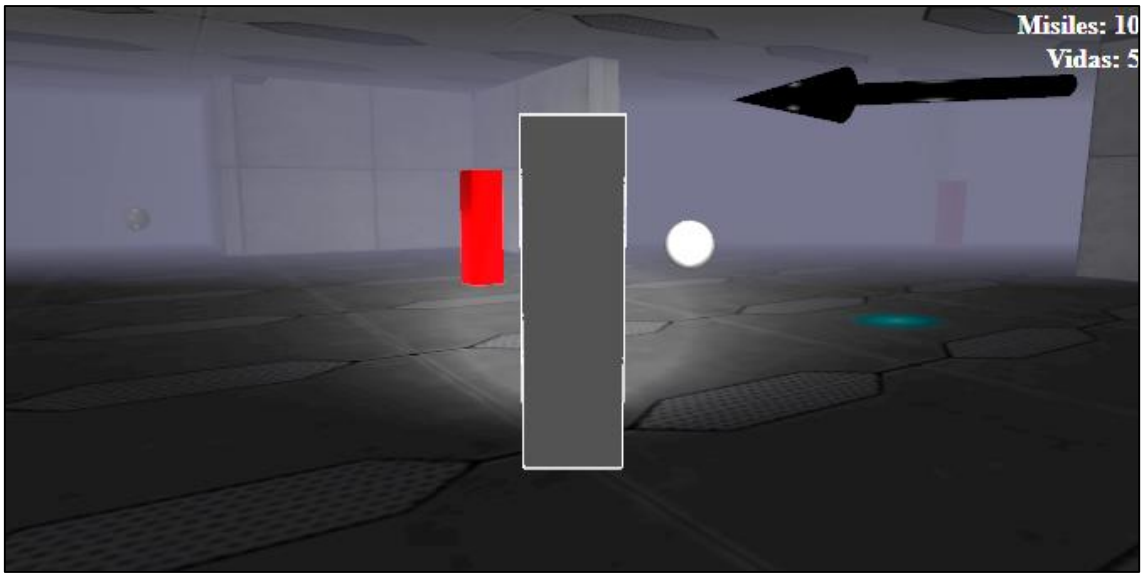



**Imagen 33. Captura del videojuego. Munición.**

### 10.2.6. Cámara

Aunque la visión inicial sea en tercera persona podemos cambiar la cámara a una posición más cenital donde tendremos una mejor visión del laberinto.

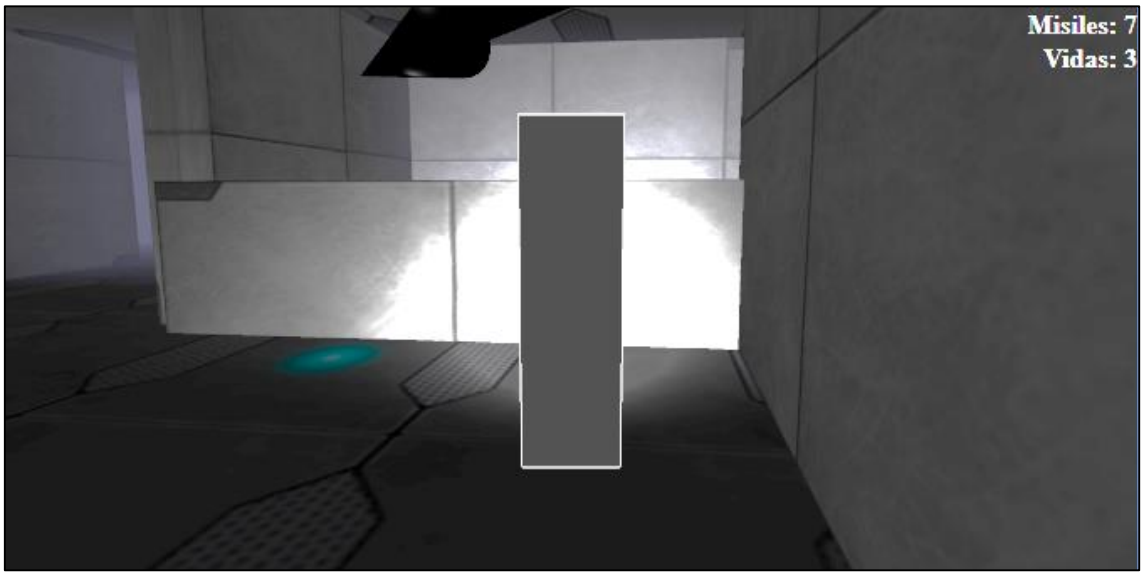



**Imagen 34. Captura del videojuego. Cámara en 3º persona.**

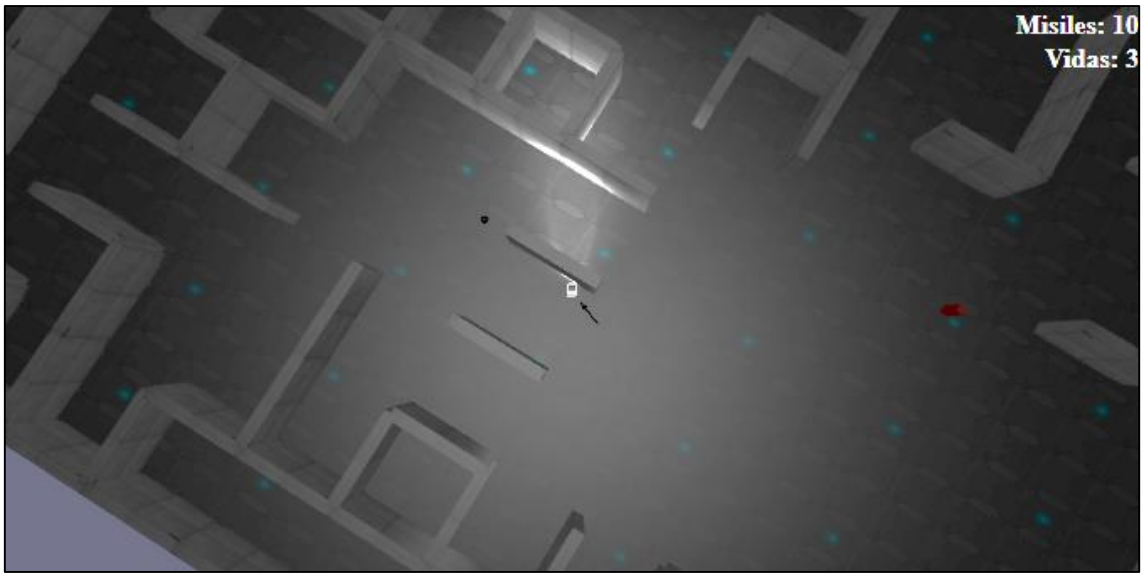



**Imagen 35. Captura del videojuego. Cámara cenital.**

Recuerda que la salida también se va moviendo por todo el laberinto, así que se rápido o no podrás salir por ahí.

# Referencias y Bibliografía


[1] K. group, «Khronos WebCL,» [En línea]. Available: https://www.khronos.org/webcl/.

[2] K. group, «Khronos WebGL,» [En línea]. Available: https://www.khronos.org/webgl/.

[3] J. H. Conway, «Juegos matemáticos,» *Scientific American,* nº 223, 1970.

[4] D. Catuhe, "Babylon.js caracteristics," 27 jun 2013. [Online]. Available: http://blogs.msdn.com/b/eternalcoding/archive/2013/06/27/babylon-js-a-complete-javascript-framework-for-building-3d-games-with-html-5-and-webgl.aspx.

[5] Mozilla Developer Network and individual contributors, «Profiling with the built-in profiler,» [En línea]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler.

[6] N. Tampere, «WebCL nokia implementation portal,» Nokia, [En línea]. Available: http://webcl.nokiaresearch.com/.

[7] «Mozilla Firefox WebCL proyect,» [En línea]. Available: http://hg.mozilla.org/projects/webcl/.

[8] SRA-SiliconValley, «Samsung WebKit WebCL,» [En línea]. Available: https://github.com/SRA-SiliconValley/webkit-webcl.

[9] M. D. N. a. i. contributors, «Profiler Add-on Cleopatra,» [En línea]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler.

[10] «NVIDIA,» [En línea]. Available: http://www.nvidia.es/object/cuda-parallel-computing-es.html.

[11] A. Gottlieb and G. S. Almasi, Highly parallel computing, Redwood City, California: Benjamin/Cummings, 1989.

[12] G. Fox, R. Williams y G. Messina, Parallel Computing Works!, Morgan Kaufmann, 1994.

[13] A. Munshi, "OpenCL Parallel Computing on the GPU and CPU," 2008.

[14] «OpenGL,» [En línea]. Available: https://www.opengl.org/.

[15] S. G. I. Corp., «SGI,» [En línea]. Available: http://www.sgi.com/tech/opengl/?/overview.html.

[16] BENOITGIRARD, "CallGraph Added to the Gecko Profiler," 13 January 2015. [Online]. Available: https://benoitgirard.wordpress.com/2015/01/13/callgraph-added-to-the-gecko-profiler/.

[17] «BabylonJS,» [En línea]. Available: http://www.babylonjs.com/.

[18] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry y D. Schaa, Heterogeneous Computing with OpenCL, Morgan Kaufmann, 2011.

[19] M. Scarpino, OpenCL in Action: How to Accelerate Graphics and Computations, Manning Publications, 2011.

[20] A. G. B. M. T. F. J. a. G. D. Munshi, OpenCL Programming Guide, Addison-Wesley Professional, 2011.

[21] D. Kirk y W.-M. Hwu, Programming Massively Parallel Processors, Morgan Kaufmann, 2010.

[22] W. Hillis y G. Steele, Data parallel algorithms, 1986.

[23] A. Toskin, "GitHub BabylonJS project," 8 january 2015. [Online]. Available: https://github.com/BabylonJS/Babylon.js/wiki/02-Basic-elements.