Creación de una biblioteca para desarrollo de videojuegos 2D en Javascript

Universidad de Murcia
Facultad de Informática

Trabajo Fin de Grado

Autor: Adrián Egea Comenge
Tutor: Juan Antonio Sánchez Laguna

adrian.egea2@um.es

7 de junio de 2017

DEDICADO A MIS PADRES,
María Isabel y Francisco José,
A MIS HERMANOS,
Jesús e Isabel María,
Y A MIS COMPAÑEROS,
M.

Palabras Clave

Videojuegos, Motor, API, 2D, 3D, Vector, Punto, Vértice, Geometría, Matemáticas, Render, Sprite, Script, Inteligencia Artificial, AI, OpenGL, WebGL, Hardware Gráfico, GPU, Física, Colisión.

Video Game, Game, Engine, Game Engine, Point, Vertex, Geometry, Math, Render, Sprite, Script, Artificial Intelligence, IA, OpenGL, WebGL, Graphic Hardware, Physics, Collision.

Índice general

Índice de figuras

Índice de Ecuaciones

Índice de código

Glosario

API
Application Programming Interface. Conjunto de subrutinas, protocolos y herramientas que simplifican el acceso a ciertas capacidades de un sistema.

Callback
Código ejecutable pasado como argumento a una función para poder ser ejectuado en algún momento.

Collider
Forma geométrica de un objeto compuesta por vértices destinada a la detección de colisiones.

Comportamiento
En videojuegos, se habla de comportamiento de una entidad cuando esta sigue unas instrucciones definidas por el programador, aportándole inteligéncia artificial.

Editor
Un editor es una herramienta gráfica que permite diseñar videojuegos de forma visual.

Entidad
Una entidad es cualquier objeto manipulable de la escena de un juego, un personaje, un obstáculo, un sonido.

Escena
Espacio virtual donde se ubican los objetos, como si de un escenario de teatro se tratara.

FPS
Frames Per Second. Frecuencia a la que son mostradas las imágenes del juego. A mayor frecuencia mayor fluidez de movimiento.

Game Engine
Biblioteca cuyo fin es simplificar la creación de videojuegos.

GLSL
OpenGL Shading Language. Sintaxis inspirada en *C*. Diseñado para programar ciertas fases programables de la tarjeta gráfica.

GPU
Graphic Processing Unit. Hardware especializado en el cálculo y generación de gráficos por pantalla.

Grafico
Cualquier imagen mostrada por pantalla. Fotos, videojuegos, vídeo, interfaces gráficas.

HTML5
Define tanto a la quinta versión del lenguage *HTML*, como al conjunto de tecnologías que potencian este lenguaje.

Input
En el ámbito de los videojuegos, se refiere a cualquier tipo de entrada introducida por el usuario mediante un periférico (teclado, ratón, mando).

Inteligencia Artificial
Simulación de una inteligencia mediante software. En el ámbito de los videojuegos, se utiliza para simular comportamientos en entidades no controladas por el jugador.

Matriz
Conjunto de números dispuestos en forma de cuadrícula, por filas y columnas.

Multijugador
Concepto que se refiere a videojuegos en los que participan varios jugadores, interconectados mediante internet.

OpenGL
Biblioteca para dibujado de gráficos 2D/3D acelerados por hardware.

OpenGL ES
Versión de OpenGL destinada a dispositivos móviles y embebidos.

Periferico
En el ámbito de los videojuegos, cualquier tipo de dispositivo externo que permite al jugador interactuar con el videojuego (teclado, ratón, mando).

Pipeline
En este contexto, conjunto de fases internas del hardware gráfico.

Render
Sinónimo de dibujar por pantalla.

Renderizado
Proceso de generación de una imagen. Desde que los datos son enviados al hardware gráfico, hasta que se procesa y se muestra por pantalla.

Script
En el ámbito de los videojuegos, pequeño programa que define el comportamiento de una entidad.

Shader
Programa, el cual, el hardware gráfico, puede compilar y ejecutar.

Sprite
Figura 2D con una textura. Pueden estar animados. Cualquier objeto de un videojuego 2D es un sprite.

Spritesheet
Imagen que almacena de forma consecutiva, cada instante de una animación.

TDD
Test Driven Development. Metodología que implica, crear pruebas, que debe pasar el código, para poder ser validado y poder continuar.

Texture Atlas
Imagen que almacena todas las texturas de un videojuego en una sola.

Tile
Pieza, por lo general rectangular, que compone un mapa. Entiéndase mapa, como el mundo 2D de un videojuego.

Vector
En este proyecto nos referimos al vector euclideano (x,y,z) y a su versión homogénea (x,y,z,w).

WebGL
Versión de OpenGL ES destinada a navegadores.

1
Resumen

Este *Trabajo de Fin de Grado* muestra el diseño e implementación de un *Motor de Videojuegos* o *Game Engine*, orientado a crear videojuegos 2D para navegador. Un *Game Engine* es una biblioteca que simplifica la creación de un videojuego, en este caso 2D.

En este proyecto, el problema planteado es crear un *Game Engine* capaz de dibujar, texturizar y animar sprites 2D mediante spritesheets, la simulación de físicas 2D, la gestión de escenas, capacidad de crear interactividad o IA básica mediante scripts, la entrada de usuario mediante teclado y ratón, así como funcionalidades básicas de vectores y matrices, junto con las operáciones aritméticas necesarias para estas estructuras matemáticas.

En este documento se expondrá el camino seguido para construir todos los módulos que componen el *Game Engine*, la metodología seguida, así como las decisiones de diseño tomadas, optimizaciones aplicadas, problemas encontrados y soluciones propuestas. También herramientas, tecnologías y bibliotecas analizadas para su uso en este proyecto, como *WebGL* y *Box2DWeb*.

Con el fin de aprovechar este proyecto para aprender todo lo posible sobre los conceptos de computación gráfica y arquitectura interna de los *Game Engines*, se han implementado todos los componentes del *Game Engine* desde cero. A excepción, por cuestiones de tiempo, del módulo de simulación física, que será una biblioteca externa llamada *Box2DWeb*.

2

Extended Abstract

The project presented here is a 2D *Game Engine* designed to create Web Browser games. A *Game Engine* is not a very complex concept to explain, it is a reusable code (usually it is distributed with a set of tools or a graphical editor) which is responsible for abstracting the details of most common tasks on game programming, such as graphics rendering, physics simulations, user input (keyboard, mouse, gamepad or another periferic), audio, animations, scenes management, network and memory management. So that developers (artists, designers, scripters and other programmers) can focus on the art, mechanics and development of the story of the game.

A *Game Engine* is usually composed of other code packages created specifically for each function such as rendering, physics simulation, AI, scripting, these packages are called middleware, since it is code that is placed between the programmer (video game) and the low-level features of the machine, abstracting the developer from the these low-level features.

A *Game Engine* encompasses many different fields, usually mathematics, computer graphics, and physics simulation. Relating to computer graphics we have geometry, linear algebra, arithmetic of vectors and matrices, linear transformations, projective space. And as for physics, integral and differential calculus, solid object dynamics, collision detection algorithms, space partitioning.

Obviously all these concepts must be implemented and we have to keep in mind programming issues such as organization and design of the code, so that it is modular and maintainable through the use of appropiate design patterns. A *Game Engine* must be executed in real time. That means the optimizations are a priority in this project.

There are many other fields that must be addressed when implementing a *Game Engine*, such as those mentioned above, audio, animation, scene management, network and memory management. In this project we focus on the core packages of a *Game Engine*, the rendering package and the physics simulation package. We will also address other extra packages such as the scene management package, the scripting package and the input package.

Note that this project was born as a desktop *Game Engine*, written in C++ and OpenGL. From that time (summer 2014) until today, the project has been modified, improved and optimized, finally, ported to JavaScript and WebGL.

Therefore, it is intended to develop a library able to handle 2D rendering, texture loading, sprite animation, spritesheets management, 2D physics simulation, keyboard and mouse user input. But it is not intended to cover areas such as audio or multiplayer connectivity. It is also offered a module for vector and matrix operations.

The library will be also able to manage scenes and entities, but this feature is not one of the main objectives for this project, only rendering and physics simulation will be assigned as main objectives.

The *Game Engine* is thought as a set of layers. The lower layers offer basic features to the upper layers, which use them to do more complex tasks. The three lower layers are the Graphics, Physics and Scripting layers. These layers provide a set of capabilities such as rendering, physics simulation and script execution.

Above these three layers, we find the Engine layer. This layer acts as the facade to the user and it is used by this to creating games. In this layer are classes for scenes and entities management, such as `Scene`, `GameObject` and `Component`.

Other auxiliar layers are Input, Resource Loader and Math. They offer user input processing, resource loading (such as textures) and math utils (vectors and matrices). These layers are transversal and they serve the rest of the layers.

This *Game Engine* is designed under the concept of entities. An entity represents any object in the scene. In this project, entities are called as `GameObjects`.

Now, the CES pattern can be introduced. CES or Component-Entity-System is a design pattern that gives priority to the composition over the inheritance. The data is organized as follows. The components are the pieces that contains the useful information. A component can be attached to an object (entity) and gives special features to this object, like rendering capabilities, AI, or physics properties. An entity (GameObject in this project) is an empty container that can be customized with the components.

Each type of component is managed by a specialized system. In this project, there are three systems, which are, `RenderEngine`, `PhysicsEngine` and `ScriptEngine`.

Now we will descend to the lowest layer of this *Game Engine*, the mathematics. Computer graphics is based on certain fields of mathematics, such as linear algebra and geometry. Specifically, vectors and matrices are the basis of calculations made in any videogame, mainly graphics and physics simulations. They are used to represent the position, velocity or acceleration, these are some of the magnitudes that can be representend by vectors.

In this project, the mathematics package offers optimized classes such as `Vector4`, `Vector3`, `Vector2` and `Matrix4`. Also, the basic operations for vectors and matrices.

This mathematics package is used by the graphics package, which is able to create sprites with a texture and animate them through spritesheets. The graphics package supports the creation of orthographic and perspective projection cameras too, as well as support fot texture atlas and materials.

The render system has been optimized with some techniques such as render batches, frustum culling and texture atlases. A batch is a set of entities than share the same texture, and can be drawn whitout change the texture, this reduce the GPU-CPU comunication. The frustum culling technique ignores the objects that are placed outside of the viewing volume. Finally, the texture atlas optimization allows that a lot of objects share the same texture.

So that the sprites can be drawn basically it has to execute three phases: `bind`, `update` and `render`. First phase, `bind`, creates a `WebGL` structure, named `VAO`. This structure stores the data that will be shared by all the instances of the same sprite, such as vertices, texture coordinates and indices. This `VAO` is stored in the graphic card memory. Note that the data is only sent once to the graphic card memory, then can be drawn many times without sending any data.

Once this data has been sent to the graphic card memory, the main loop starts, in which phases

`update` and `render` will be continuously repeated.

On the `update` phase, the `Transformation Matrices` are sent to the `WebGL shader`. The shaders are shared too, every sprite uses the same shader. This shader is designed to render 2D sprites with a texture. The texture can be cropped, static or animated.

On the next phase, `render`, the sprites are drawn. There are three conditions so that an object can be drawn. If the sprite is activated, the sprite belongs to the correct depth layer and the sprite lies within the frustum volume, then, the sprite is drawn.

If the sprite meets these conditions, then, the data that is not shared with the rest of the sprites, which makes the sprite unique, is loaded into the shader. These datas are the position, rotation, scale, the color, the selected region of the texture or the selected frame of the animation. The position, rotation, scale are combined in a matrix called `Model Matrix`, these three values are not sent to the shader, but the matrix.

Once these datas are loaded, the `draw call` is executed, and the internal render process within the graphic card starts. After the rendering, The screen will be cleared and returned to the `update` phase. On the `update` phase the position and rotation of the sprites can change because a physics simulation step is executed.

The physics simulation package is a wrapper for the `Box2DWeb` library. This facade simplifies the access to all the `Box2DWeb` features, so that the programmer can use the clases avaibles in this package to not deal directly with `Box2DWeb`. The programmer can also access all `Box2DWeb` features, through the wrapper classes.

A GameObject that only contains render or physics components is an object that represents a non interactive object. If objects are required to interact with each other, then, something else is needed. Here the `Script` class is introduced, the scripts are executable code that can be attached to a GameObject, so that the object can react to events such as collisions, the update and the creation/destruction of the GameObject itself.

Now the two main subsystems and the scene management package are presented, so we can explain how these subsystems, scenes and objects are related. At the beginning all `GameObjects` are created and customized with the components, then these `GameObjects` are added to the scene. When the *Game Engine* runs the main loop, the scene is loaded.

Loading the scene involves looping the list of `GameObjects` and extracting their components. Each component is sent to the corresponding subsystem (Render, Physics simulation or Scripting). When a scene is unloaded, the three subsystems clean their `GameObjects` and get back to an initial status.

In relation to the user input, the `Input` class has been implemented to act as the facade between the user and the native Javascript input. This class offers a serie of useful functions designed for the management of keyboard and mouse input. Functions such as `isKeyPressed`, `getButton` and `getCursorPosition` which returns a `Vector2` that represents the cursor position on the screen.

Since natively it is only possible to check if one key has been pressed at the same time, a list has been implemented, where pressed keys are stored. So when you ask if a key is pressed, the list is scrolled. This allows determining if one or more keys have been pressed. If the key is in the list, the answer is affirmative (true). When the key is released, the key is removed from the list. If the key is already in the list, it will not be reinserted. In this way, multiple keys can be asked for.

The proposed requirements for this project were to create a *Game Engine* able to draw, texturize and animate 2D sprites through spritesheets, physics simulation, scenes management, basic AI through

scripts, keyboard/mouse input management and vector and matrix operations.

All these objectives has been achieved through the following methodology. Before the code was written, external technologies were analized such as WebGL and Box2DWeb. Then a task list was created, following the Kanban methodology, a class diagram was created too. The implementation was tested through an integration test, which ensured the correctness of the code.

In this project I have been able to consolidate my knowledge about the WebGL library and computer graphics too. I have understood the importance of optimizing every chunk of code, it is not useful to simply draw on the screen, we need to do it in most efficient way, since a *Game Engine* is a tool that will be used to draw many elements on the screen.

On the other hand, we don't only have drawing functions, but also we have to take into account the physics simulation. This task also needs resources, so only an efficient implementation could handle the rendering and the physics simulation process at same time. Box2DWeb has been the selected library to do this job.

In terms of programming language, this project has allowed me to get very familiar with the JavaScript language. It is true, that the true power of JavaScript lies in libraries oriented to web programming, such as JQuery. In this project none of these libraries was needed. In fact the result would not have changed if it had been programmed in another language.

For this project, a large number of tools were not needed. It has been written in JavaScript, `HTML` and `GLSL` (`OpenGL` Shader Language). As code editor, Atom was used. Different addons were installed for JavaScript autocompletion. For execution, `Firefox` web browser was used, running on `Ubuntu 16.04`. Documentation was created by using the `js-doc` package. `Astah*` was the selected software for class diagram creation.

Finally, I have learned a lot about mathematics, computer graphics, rendering, physics simulation, game programming, *Game Engine* architecture and internals, design patterns, data structures, orders of magnitude, optimization. I have also been able to improve other skills such as project management, version control and agile methodologies.

3
Introducción

En este capítulo trataré de introducir al lector al dominio de este proyecto, los Motores de Videojuegos o *Game Engines* [1], trataré también de mostrar el *porqué* de este proyecto y sus orígenes.

El proyecto que aquí se presenta es un *Game Engine* 2D concebido para crear **juegos en un navegador web**. Un *Game Engine* no es un concepto muy complejo de explicar, es un código reutilizable (generalmente se distribuye con una serie de herramientas o junto con un editor con interfaz gráfica) que se encarga de abstraer los detalles de las tareas más comunes de la programación videojuegos, como el **dibujado de escenas por pantalla** (también denominado *renderizado*), la simulación de **físicas**, la **entrada** de usuario (teclado, ratón, mando o cualquier otro periférico), sonido, animación, gestión de escenas, redes y gestión de memoria, de modo que los desarrolladores (artistas, diseñadores, scripters y otros programadores) puedan **enfocarse en el arte, las mecánicas, el guión y el desarrollo de la historia**.

Un *Game Engine* suele estar compuesto por otros submódulos de código creados específicamente para cada función, renderizado, físicas, AI, scripts, estos submódulos se denominan *middleware*, ya que es código que se situa entre el programador (del videojuego) y las capacidades de la máquina, abstrayendo a este de su complejidad.

Un *Game Engine* engloba muchos campos diferentes, generalmente matemátias, computación gráfica y simulación de física. Referente a la computación gráfica tenemos geometría, álgebra lineal, aritmética de vectores y matrices, transformaciones lineales, espacio proyectivo. Y en cuanto a la física, cálculo integral y diferencial, dinámica de objetos sólidos, algoritmos de detección de colisiones, particionado del espacio.

Obviamente todos estos conceptos han de implementarse, teniendo en cuenta cuestiones del mundo de la programación, como organización y diseño del código, de modo que sea modular y mantenible mediante el uso de los patrones de diseño adecuados. Un *Game Engine* debe ejecutarse en tiempo real, de modo que las optimizaciones de los algoritmos implementados es una prioridad en un proyecto de este tipo.

Existen otros muchos campos que se deben tratar a la hora de implementar un *Game Engine*, mencionados anteriormente, sonido, animación, gestión de escenas, redes o gestión de memoria. Pero para este proyecto nos centraremos los módulos que conforman el núcleo un *Game Engine*, el módulo de renderizado y el módulo de físicas. Pero también trataremos algunos módulos extra como el módulo de gestión de escenas, el módulo de scripting y entrada de usuario.

Es importante comentar que este proyecto nació como un *Game Engine* orientado a videojuegos de escritorio, implementado en *C++* y *OpenGL*. Desde aquel entonces (verano de 2014) hasta el día de hoy, el proyecto ha sufrido incontables modificaciones, mejoras y optimizaciones, hasta ser portado a JavaScript y WebGL.

3.1 Motivación del proyecto

Este proyecto nace con la intención se ser una biblioteca, capaz de proporcionar una interfaz sencilla al programador para construir juegos sencillos (o no tanto) en 2D. Esta biblioteca ofrece y se encarga de todas las tareas de renderizado, simulación de físicas. Encapsulando toda la complejidad propiade APIs de más bajo nivel como pueden ser *WebGL* y *Box2D*.

No debo restar importancia a una motivación igual o mayor que la citada en el párrafo anterior, este proyecto inició sus andaduras gracias a la curiosidad y las ganas de aprender de este servidor, hace ya unos años que comencé a investigar por mi cuenta de forma autodidacta en el campo de los *Game Engines*, la computación gráfica y la simulación de físicas para videojuegos. No es sencillo introducirse en esta industria de modo que opté por el modelo *"Do the job before you get the job"*, es decir, *"Haz el trabajo antes de obtener ese puesto"*, de modo que este proyecto actuase a modo de portfolio, en forma de código y experiencia tangible.

3.2 Objetivos del proyecto

Por tanto, se pretende aquí, desarrollar una biblioteca capaz de gestionar el renderizado 2D, carga de texturas, animación de sprites mediante *spritesheets*, la simulación de físicas 2D, la entrada de usuario mediante teclado y ratón, pero no se pretende abarcar áreas como el audio ni la conectividad multijugador. También se ofrece un módulo para trabajar con vectores y matrices, junto con las operáciones básicas necesarias para estas estructuras matemáticas.

La biblioteca también será capaz de gestionar escenas y entidades, pero esto no se tomará como uno de los objetivos principales del proyecto, solo el renderizado 2D y la simulación de físicas 2D serán tomados como los objetivos más importantes.

4

Estado del Arte

En este capítulo se expondrán cuales son algunas de las tecnologías actuales en el dominio de este proyecto, se dividirá en varias secciones. Primero, algunos de los *Game Engines* orientados a la web, más utilizados.

En segundo lugar, se verán las tecnología que usan estos *Game Engines* para generar los gráficos por pantalla. En concreto, las dos tecnologías más usadas actualmente para dibujar gŕaficos en el navegador: *WebGL* y *Canvas2D*

Y por último se hablará sobre la simulación de físicas realistas en los videojuegos.

4.1 Game Engines HTML5 actuales

En 1993 *IdSofware* lanzó su juego **Doom**, sobre este juego aparecieron ciertos artículos en los que se comenzaba a hablar del término **Doom Engine**. Esto permitía a IdSoftware no desarrollar sus juegos desde cero, sino desde un códidgo base reutilizable.

Figura 4.1: Doom.

Actualmente existen cientos de *Game Engines* diferentes, algunos son para uso general, otros están orientados a un género de videojuego concreto o a unas plataformas determinadas. En este proyecto se describirá un *GameEngine* orientado a la plataforma web, construido sobre el lenguaje *JavaScript*.

A continuación se analizarán algunos de los *Game Engines* más populares que pueden usarse para crear videojuegos para la web. Antes de entrar en detalle, en la tabla 4.1 se muestra un resumen de las capacidades de cada engine:

Nombre	Editor	2D	3D	Físicas	Audio
Construct2	Si	Si	No	Si	Si
ImpactJS	Si	Si	No	Si	Si
Phaser	No	Si	No	Si	Si
GameMaker	Si	Si	No	Si	Si
TurbulenZ	No	Si	Si	Si	Si
melonJS	No	Si	No	Si	Si

Cuadro 4.1: Comparativa de *Game Engines* HTML5.

4.1.1 Construct 2

- **Desarrollador:** Scirra
- **Plataformas:** HTML5, Windows
- **Coste:** Gratis y múltiples licencias disponibles
- **Títulos publicados:** The Next Penelope, Airscape: The Fall of Gravity, Cosmochoria, CoinOp Story, Mortar Melon, Super Ubie Land Remix, Hungry Hal

Construct2 [18] posee un editor, únicamente, para el sistema operativo Windows, pero los juegos creados pueden ser exportados a otras plataformas. Este editor permite *'Arrastrar y soltar'* elementos para crear las escenas de forma sencilla.

También ofrece una pestaña en la que editar todos comportamientos y eventos del juego, para poder confeccionar la lógica del juego sin necesidad de programar. En esta última característica reside la fuerza de este *Game Engine*, pues permite editar de forma visual, tareas que de forma general se harían desde el código.

Construct2 se sirve de **Box2DWeb** para manejar el apartado físico. Para crear objetos que se comporten como en un entorno realista, seleccionaremos el objeto al que se desea agregar propiedades físicas. En el menú *Properties*, haremos click en *Behaviors* y añadiremos el elemento *physics*.

Figura 4.2: Edición de variables y condiciones gráficamente.

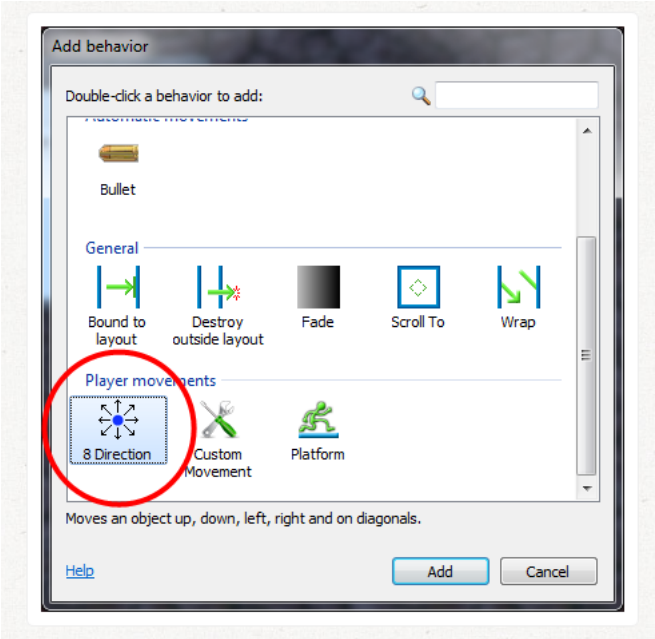

Figura 4.3: Una amplia gama de comportamientos predefinidos.

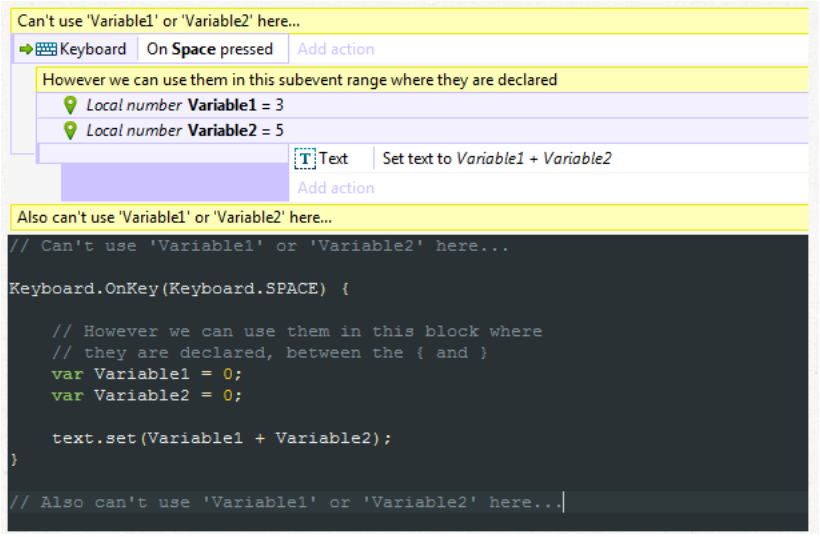

Figura 4.4: Definición de eventos, en este caso, al pulsar una tecla.

En la siguiente url se puede acceder a una serie de tutoriales entre los que se explican las características del editor: https://www.scirra.com/tutorials/.

4.1.2 ImpactJS

- **Desarrollador:** Impact
- **Plataformas:** HTML5
- **Coste:** 99$
- **Títulos publicados:** CrossCode, Super Flipside, Fire Emblem: Chronicles of the Abyss, Silent Hill 2 Demake, Elliot Quest

ImpactJS [19] es un *GameEngine* de pago escrito en JavaScript, también posee un editor construido en HTML5, llamado *Weltmeister*, este editor se ejecuta en cualquier navegador. Entre las funciones disponibles se encuentran, trabajo con capas de dibujado, capas de colisiones y manejo de las entidades. En la siguiente url podemos ver el editor en cuestión: http://impactjs.com/documentation/weltmeister.

El editor es bastante limitado, pues permite un buen control de las entidades del juego, pero no ofrece ningún manejo visual de los eventos del juego.

Este *Game Engine* también hace uso de la biblioteca de físicas *Box2DWeb*, pero no ofrece ningún wrapper para esta biblioteca, simplemente permite el uso completo de *Box2DWeb*.

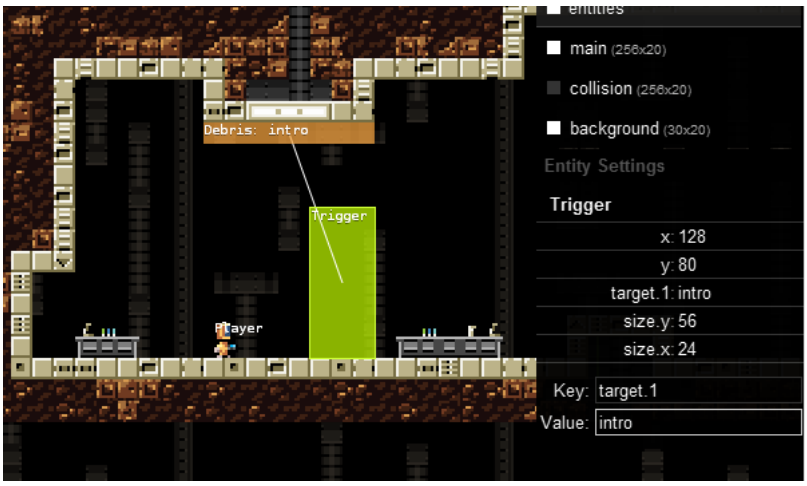

Figura 4.5: Weltmeister, editor de ImpactJS.

4.1.3 Phaser

- **Desarrollador:** Phaser
- **Plataformas:** HTML5
- **Coste:** Gratis
- **Títulos publicados:** The Townsfolk Cartel, Bowling Masters, My Swashbuckle Adventure, Wildwood Run

Phaser [20] es un *GameEngine* escrito en JavaScript, aunque los juegos creados con esta herramiente pueden ser escritos en JavaScript o Typescript. Este *Game Engine* no posee un editor oficial, pero se pueden encontrar varios proyectos independientes en internet. Aparte existe el llamado *Phaser Sandbox*, un editor de código que permite realizar pruebas y ver sus resultados en el mismo navegador.

En cuanto a las físicas, Phaser opta por una solución muy completa, la biblioteca simplifica la creación de objetos físicos y la selección del *physics engine* a elegir, como podemos ver en su documentación oficial [21].

Es decir, Phaser ofrece la posibilidad de usar uno o varios sitemas físicos, como *p2.js, arcade physics, NINJA physics, Box2d*. Esto se entiendo mejor con un ejemplo, pensemos que estamos diseñando un videojuego, el cual contiene un vehículo, como podría ser un coche, que dispara proyectiles a una serie de naves que sobrevuelan al vehículo en paralelo.

En *Phaser*, podríamos dejar la simulación del vehículo en manos de *p2.js*, un *physics engine* muy completo, pero cuyas simulaciones son más pesadas, mientras que la física de los proyectiles y las naves sería un trabajo para *arcade physics*, el cual es más ligero, pero para simular proyectiles y naves, nos sobra, ya que tienen un movimiento muy simple y no necesitan una simulación compleja.

El vehículo será simulado por *p2.js* ya que, puede ofrecer un mayor realismo, por tanto solo tendríamos una simulación pesada en marcha, mientras los proyectiles y naves serían simulaciones ligeras que no perjudican tanto el rendimiento general del videojuego.

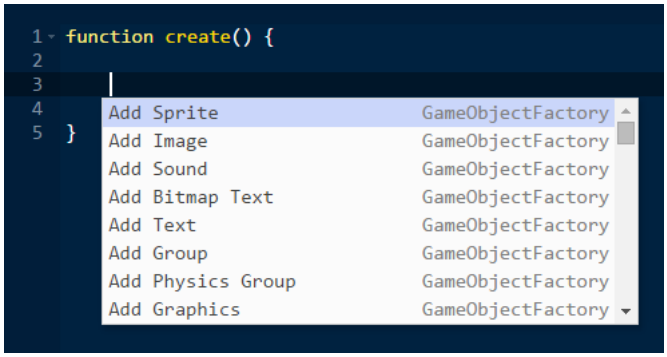

Figura 4.6: Autocompletado de código.

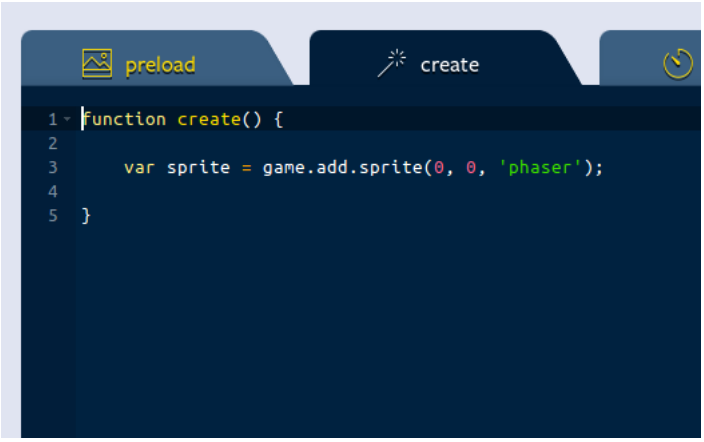

Figura 4.7: Editor Phaser Sandbox.

4.1.4 GameMaker

- **Desarrollador:** YoYo Games
- **Plataformas:** HTML5, Ubuntu(Linux), Tizen, Windows, Mac, Windows Phone, Android, iOS, Xbox One, PlayStation 3, PlayStation 4 y PlayStation Vita, Amazon Fire.
- **Coste:** Gratis y múltiples licencias disponibles
- **Títulos publicados:** Hyper Light Drifter, Maldita Castilla, Spelunky, Death's Gambit, Crashlands, Psebay, Orbit, Defenders of Ekron

GameMaker [23] puede exportar juegos a múltiples plataformas, entre ellas HTML5. Se distribuye con un editor oficial que permite crear juegos mediante el mecanismo de *'Arrastrar y soltar'*. Posee su propio lenguaje de programación inspirado en *C*, llamado *GameMaker Language* o ***GML***. Aunque también descata en la edición gráfica de eventos y entidades del juego. GameMaker también se apoya en *Box2D* para habilitar las propiedades físicas del mundo y los objetos.

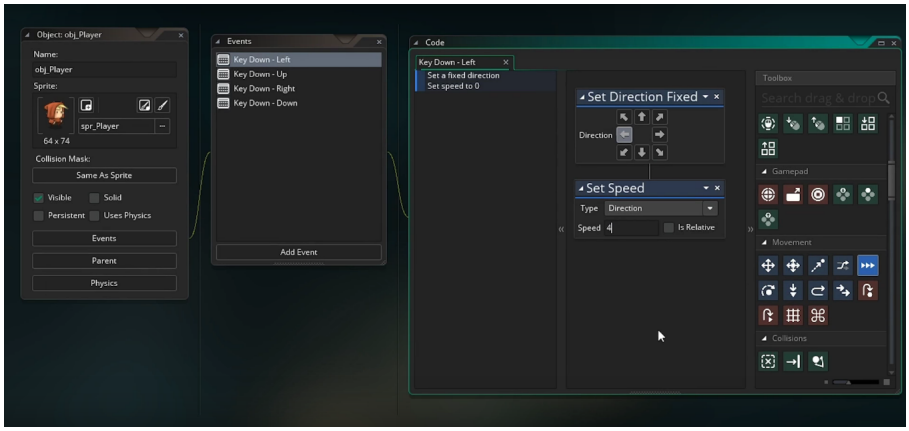

Figura 4.8: Entidades y eventos.

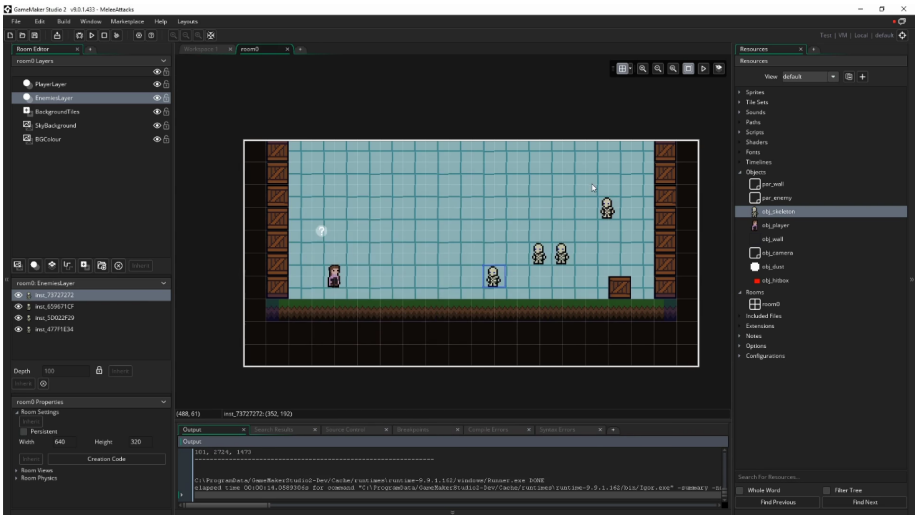

Figura 4.9: Editor de Game Maker.

4.1.5 TurbulenZ

- **Desarrollador:** Turbulenz
- **Plataformas:** HTML5
- **Coste:** Gratis
- **Títulos publicados:** Polycraft, The Marvellous Miss Take, Boundless

TurbulenZ [24] está programado en JavaScript principalmente, sus juegos son programados en JavaScript, TypeScript o CoffeScript. Este *Game Engine* no posee un editor oficial. Es uno de los pocos motores orientados a la web que soportan gráficos 3D.

Turbulenz implementa su propio *physics engine* para la gestión de las físicas 2D, para las físicas 3D, Turbulenz hace uso de un port a JavaScript de *Bullet Physics Library*, un *physics engine* 3D open source.

4.1.6 melonJS

- **Desarrollador:** melonJS
- **Plataformas:** HTML5
- **Coste:** Gratis
- **Títulos publicados:** MARS Type 1, GIK, Labyrinth survival, Schweinballons, Dead Can't dance

melonJS [25] es un *Game Engine* más ligero que los anteriores, escrito en JavaScript, sus juegos se programan también en JavaScript, además no posee un editor oficial. melonJS implementa su própio *physics engine*, diseñado especialmente para consumir poca CPU.

Una de sus ventajas es que provee integración con las siguientes tecnologías del ámbito de los videojuegos y las aplicaciones HTML5 híbridas:

- **Tiled** [26]: editor de mapas de *tiles*, de propósito general. Se denomina *tile* a cada pieza que compone un mapa.
- **TexturePacker** [27]: editor para la creación de *spritesheets* y *texture atlases*, ambos conceptos son similares, texturas independientes agrupadas en una textura mayor.
- **Shoebox** [28]: aplicación similar a *TexturePacker*.
- **PhysicEditor** [29]: Editor orientado a las propiedades físicas de los objetos y sus formas geométricas de colisión.
- **BitmapFont Generator** [30]: Generador de fuentes en imágenes *.bmp*.
- **PhoneGap** [31]: Framework de desarrollo de aplicaciones para móvil híbridas, mediante HTML, CSS3 y JavaScript.
- **CocoonJS** [32]: Plataforma para desplegar, testear. También permite acelerar aplicaciones híbridas mediante características nativas de los dispositivos.
- **Ejecta** [33]: Es una implementación en JavaScript, para *Canvas* y *Audio*, exclusivamente para *iOS* y *tvOS*. Puede verse como un navegador que solo puede mostrar un elemento *Canvas*.

4.1.7 p5.js: Processing para la web

p5.js [34] es una biblioteca para JavaScript que trae el concepto de **Processing** a la web, esta biblioteca posee un conjunto completo de funcionalidades para el dibujo, siendo el lienzo de trabajo la propia página del navegador, pudiendo interactuar incluso con los elementos HTML5 (elementos del DOM) de la página, como el texto, video, input, imágenes, cámara, sonido.

Aquí podemos ver un pequeño ejemplo, en el que se dibuja un círculo a partir de un sencillo comando.

```
function draw() {
  ellipse(50, 50, 80, 80);
}
```

Figura 4.10: Hello World con p5.js.

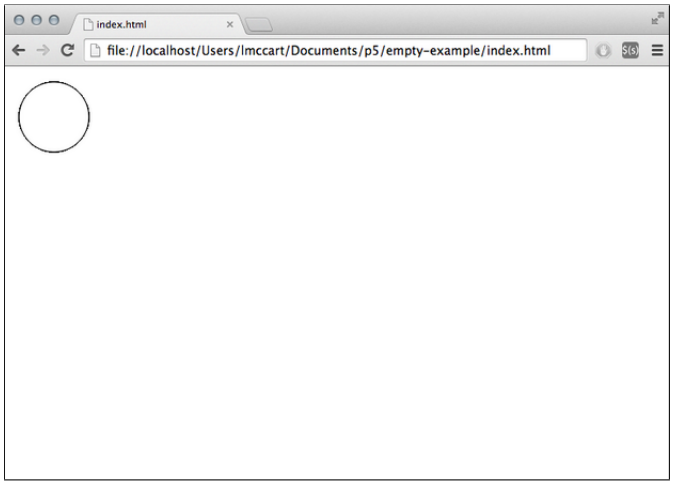

Figura 4.11: Resultado.

p5.js es un proyecto ambicioso que trata de llevar la programación visual a todo el mundo de la manera más cercana al Processing original. Sus ventajas son que, *p5.js* se programa en JavaScript, lenguaje con el cual posíblemente estés familiarizado, gracias a esto, es posible manipular los elementos del `DOM`, además, el proyecto es mantenido de manera oficial por la **Processing Foundation** [35].

La *Processing Foundation* fue fundada en 2012 tras más de una década de desarrollo de *Processing*. Su objetivo es promover la programación dentro de disciplinas artísticas, visuales e interactvas, que puedan tener alguna relación con algún campo tecnológico.

A partir de *p5.js* existe otra biblioteca que permite orientar *p5.js* a la creación de videojuegos sencillos, hablamos de la biblioteca **p5.play** [36], la cual provee, la clase `Sprite` para manejar objetos 2D con capacidades como, animaciones, detección y resolución básica de colisiones y cámara entre otros.

p5.play potencia la usabilidad y simplicidad, **no el rendimiento**, su motor de físicas no está basado en bibliotecas como *Box2D*, carece de eventos y tampoco soporta capacidades 3D.

4.2 Gráficos acelerados por hardware: WebGL

WebGL

- Web Graphics Library, es una API disponible para el lenguaje JavaScript que permite el renderizado de gŕaficos 2D y 3D en culaquier navegador compatible, a través del elemento HTML *<canvas>*.

La aparición de esta nueva API ha permitido a aquellos programadores que poseían conocimientos de OpenGL, renovarse y migrar a una nueva plataforma de desarrollo como es la Web, sin mucho esfuerzo.

La versión 1.0 de WebGL se ejecuta sobre la implementación nativa de OpenGL ES 2.0 (Un subconjunto de OpenGL orientado a dispositivos móviles o embebidos) en la máquina cliente, esto permite realizar el renderizado en el hardware gráfico (GPU) del cliente, lo que se denomina **client-based rendering**. Generar gráficos acelerados por hardware ofrece un mayor rendimiento y capacidades que si el renderizado se realizase por software (a través de la CPU).

De forma general, OpenGL, permite enviar información gráfica al hardware de renderizado para que este genere la imagen y la muestre por la pantalla. Esta información puede ser: vértices, coordenadas de texturas, matrices de transformación, puntos de vista de la cámara, vectores, normales, texturas, luces, etc.

Toda esa información pasa por un conjunto fases denominado **pipeline**, cada fase del pipeline se encarga de tratar esa información gráfica de una forma, generar nueva y enviarla a la siguiente fase. Al final de todas las fases, el resultado es una imagen generada y lista para mostrarse por pantalla.

En las GPUs antiguas, ese pipeline era fijo, denominado **fixed-pipeline**, esto signifca que esas fases no eran controlables, el método con el que se trataba la información en cada fase no se podía cambiar, era siempre el método predefinido y embebido en el hardware gráfico. Por esto las primeras versiones de OpenGL debía ofrecer una interfaz para un pipeline fijo.

En las GPUs modernas el pipeline es programable, esto significa que en ciertas fases del pipeline, el método con el que se trata la información (en esa fase), puede ser cambiado y definido por el programador. Esto se lleva a cabo mediante la creación de **Shaders**. Los shaders son programas escritos en un lenguaje que puede ejecutar el hardware gráfico en esas fases programables, para cada fase se puede escribir un shader.

Esto ofrece una mayor flexibilidad, una mayor variedad de efectos que se pueden conseguir con el hardware gráfico y que el programador pueda extraer todo el rendimiento que pueda gracias a este mayor control y personalización de las fases del pipeline.

Más adelante en la sección 5.3 se continua explicando y analizando el funcionamiento de esta API.

4.3 Canvas2D

`<canvas>` [3] es un elemento **HTML** el cual puede ser usado para dibujar gráficos y animaciones a través de JavaScript. Este elemento ofrece varios contextos de renderizado, podemos dibujar mediante un contexto *Canvas 2D*, o podemos dibujar mediante WebGL que usa un contexto 3D basado en OpenGL ES.

```
var canvas = document.getElementById('mycanvas');
var 2d-context = canvas.getContext('2d');
var webgl-context = canvas.getContext('webgl');
```

Los *Game Engines* para web basados en *WebGL*, se sirven de este componente `<canvas>` para mostrar los gráficos generados.

Por ejemplo, el siguiente código [4] genera una línea diagonal:

```
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.moveTo(0,0);
ctx.lineTo(200,100);
ctx.stroke();
```

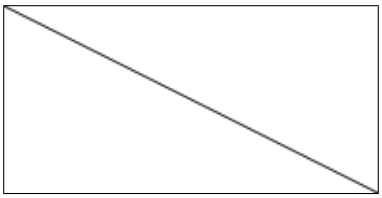

Figura 4.12: Línea en `<canvas>`.

Mientras que este otro código genera un círculo:

```
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.arc(95,50,40,0,2*Math.PI);
ctx.stroke();
```

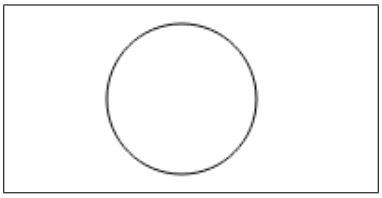

Figura 4.13: Círculo en `<canvas>`.

4.4 Simulación de Físicas

Physics Engine o Motor de físicas

- Un *physics engine* [9], simula la física de los objetos para otorgarles movimiento y reacciones realistas.

Un *physics engine* gestiona principalmente dos aspectos distintos, **Simulación de los objetos** y **Detección de colisiones**. Para comprender cómo trabaja un *physics engine*, expondré la estructura que sigen los *physics engines* de forma general.

4.4.1 Simulación: Integración numérica

Primero entendamos cómo se **simula el movimiento de un objeto**. En cada iteración o **paso** que realiza el *physics engine*, los objetos se mueven cierta **distancia**, a cierta **velocidad**, con cierta **aceleración** debido a la acción de una o varias **fuerzas** que actuan sobre el objeto.

- Para **simular** cada objeto:
  - Acumular las fuerzas que se están aplicando a un objeto, en una sola fuerza resultante.
  - Usar la **Segunda Ley de Newton** $F = m * a$ para calcular la aceleración del objeto.
  - Usar la aceleración del objeto para calcular la velocidad.
  - Usar la velocidad para calcular la posición.

Este paso se realiza en una **pequeña fracción de tiempo**, por ejemplo, en $1/60$ segundos. Este proceso en el que se resuelve la siguiente posición de los objetos se denomina **integración numérica** [5]. Existen varias técnicas de integración, y cada una posee unas ventajas e inconvenientes, dependiendo de la precisión con la que calculan la nueva posición o del coste computacional que conllevan: **Método de Euler**, **Integración de Verlet**, **Método del Punto Medio** y **Runge-Kutta**.

Hay que entender que todos estos métodos tienen cierto error de precisión, ya que dividen el tiempo, el cuál sabemos que es contínuo, en pequeños fragmentos de tiempo discreto, lo que introduce un fallo a la hora de calcular nuevas posiciones. La única forma de obtener un error nulo sería reduciendo esos fragmentos discretos a 0, lo cual no es posible.

4.4.2 Detección de Colisiones

Bien, una vez hemos calculado las nuevas posiciones de nuestros objetos, puede que hayan colisionado entre sí. Por tanto el *physics engine* debería detectar las colisiones, y crear unas respuestas físicas a esas colisiones, veamos como.

Primeramente, los objetos de un videojuego no colisionan con sus propios cuerpos, sino que son simplificados a figuras geométricas más simples, como rectángulos, cubos, císrculos, esferas o cilindros, entre otros.

Esto permite dividir la detección de colisiones en dos etapas, la **Broadphase** o *fase ancha* y la **Narrowphase** o *fase estrecha*.

4.4.2.1 Broadphase

En la *broadphase* se comprueba la detección de colisiones mediante figuras simples, generalmente rectángulos (*AABB* o *Axis-Aligned Bounding Box*), si algunos de estos rectángulos colisionan, entonces pasan a ser candidatos a posible colisión.

Cuando acaba la *broadphase*, se obtiene una lista de parejas candidatas a colisionar, y es entonces cuando en la *narrowphase* se comprueba si realmente colisionan o no, pero esta vez la comprobación se realiza con las figuras geométricas contenidas dentro del *AABB*.

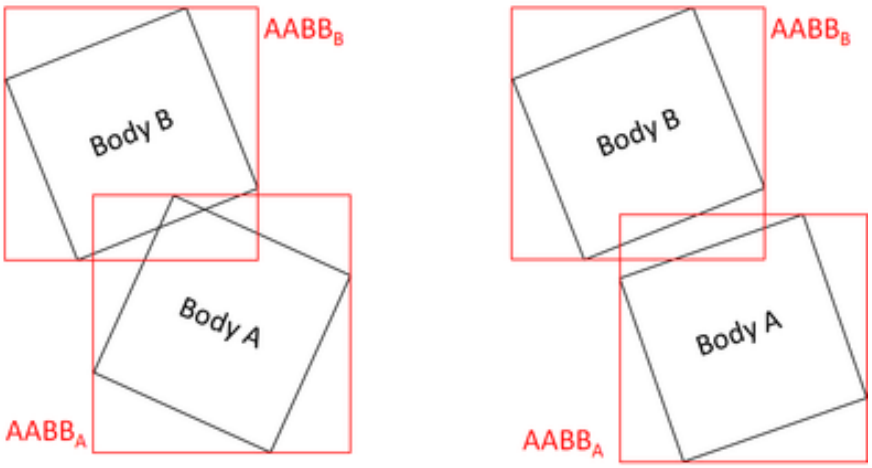

Figura 4.14: Dos parejas de rectágulos (en negro) contenidos dentro de sus respectivos *AABB* (en rojo).

En la figura 4.14 ambas parejas dan colisión en la *broadphase*, sin embargo, la primera pareja colisionará en la *narrowphase* pero no la segunda pareja.

4.4.2.2 Narrowphase

En la siguiente imagen (figura 4.15) podemos ver como dos círculos colisionan, cada círculo posee un vector velocidad, ν_A y ν_B. El vector **normal**, es perpendicular a la superficie de colisión, representada mediante la tangente de colisión, justo donde se cruzan estos dos vectores encontríamos el **punto de contacto**.

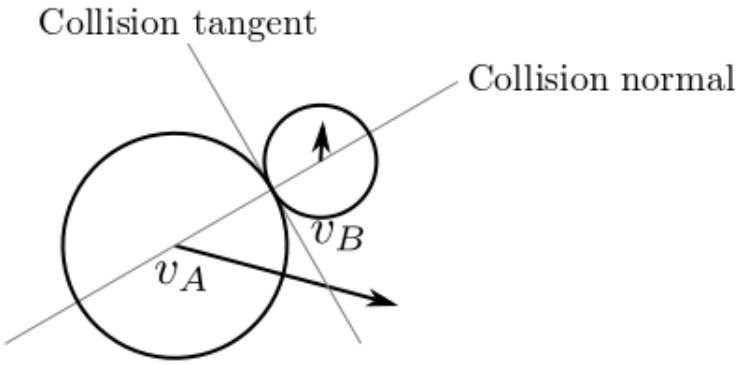

Figura 4.15: Anatomía de una colisión.

Una vez obtenidos estos datos se puede generar una respuesta física, como por ejemplo un rebote en direcciones opuestas.

Acabamos de ver un ejemplo con dos círculos, pero para figuras compuéstas por vértices como pueden ser, por ejemplo, rectángulos o cubos, existen ciertos algortimos. Algunos de los más usados son

GJK (*Gilbert–Johnson–Keerthi*) [16], **EPA** (*Expanding Polytope Algorithm*) [17] y **SAT** (*Separating Axis Theorem*) [15].

Estos algorimos se utilizan para encontrar el punto de contacto, la profundidad de penetración entre los objetos y la normal de colisión, **solo para figuras convexas**. SAT es usado en entornos 2D, mientras que una combinación de GJK+EPA es usada en entornos 3D. GJK necesita apoyarse en EPA, ya que, por sí solo, GJK solo mide la distancia entre dos figuras convexas.

4.4.2.3 Optimizando la detección de colisiones

Esta comprobación de colisiones se debe realizar para todos los objetos, una combinación de todos con todos, lo que para n objetos implica un orden de $O(n^2)$, más concretamente, $n * (n-1)/2$

Esto puede paliarse mediante técnicas de **Particionado del Espacio** [6]. Estas técnicas dividen el espacio en secciones para tratar de aislar los objetos que colisionan entre sí, de objetos alejados que no participan en la colisión, y así poder ahorrarse esas comprobaciones.

Estas técnicas se aplican a la *broadphase*. La primera técnica y la más obvia es la antes mencionada, detección de colisiones mediante **Fuerza Bruta**. Dejando de lado esta técnica, pasemos a hablar de algo más eficiente, las técnicas de *Particionado del Espacio*, como son, **Grids**, **BSP** o *Binary Spatial Partitioning* y **Quad/Octrees**.

La más simple es, crear una rejilla o **Grid** de tamaño fijo de modo que los objetos se generalicen con ciertas casillas de esa rejilla, hay que tener cuidado con el tamaño de rejilla que se escoge, ya que una casilla muy grande podría englobar demasiados objetos y casillas muy pequeñas podrían no generalizar bien el objeto original.

El **Particionado Binario** del espacio o **BSP**, divide el espacio mediante planos, de forma recursiva, un plano genera dos subespacios o dos nodos. Obviamente el espacio no puede subdividirse infinitamente.

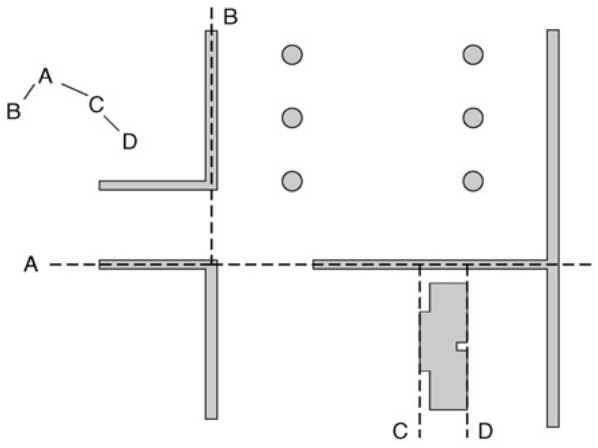

Figura 4.16: Particionado Binario del Espacio.

Los **Quadtree** y **Octree** guardan muchas similitud con los *BSP*, un árbol de este tipo está compuesto por nodos con **cuatro hijos**, cada vez que un nodo se subdivide, lo hace en cuatro. Difiere con *BSP*, en que, los nodos están alineados con los ejes del mundo. Un árbol de este tipo no subdivide un nodo, si no hay más de un objeto dentro, por tanto solo comprobará las colisiones de aquellos nodos que contengan

más de un objeto dentro, esto puede suceder porque los niveles de subdivisión son limitados, el árbol no puede subdividirse infinitamente.

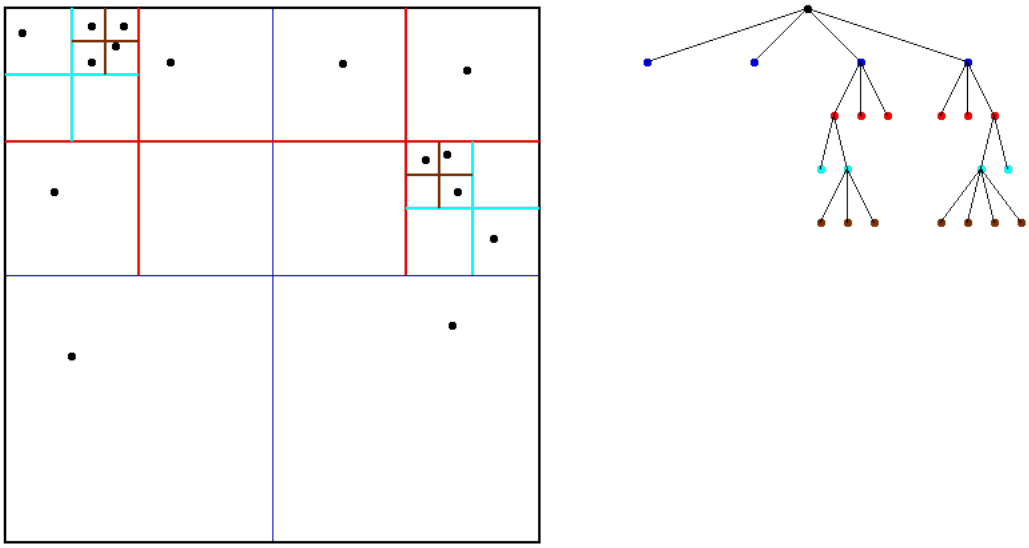

Figura 4.17: Ejemplo de Quadtree.

Nótese que en el **peor de los casos**, el caso en el que todos los objetos se encuentran agrupados dentro del mismo nodo (esto es válido para todos los algoritmos mencionados anteriormente), el algoritmo ofrecerá el mismo rendimiento que **Fuerza Bruta**, ya que deberá hacer las $n*(n-1)/2$ comprobaciones.

4.4.3 Box2D

Muchos de los *Game Engines* mencionados en la sección anterior 4.1, utilizan **Box2DWeb** [8] para simular físicas realistas en sus juegos.

Para explicar qué es *Box2DWeb*, primero tenemos que entender qué es **Box2D** [7]. Box2D es un *physics engine* o *motor de físicas*, ligero, robusto, eficiente y muy portable (a otros lenguajes), ha sido probado en muchas aplicaciones y plataformas, es gratis y totalmente open-source. Fue mostrado al mundo como una demo en la **GDC** de 2006.

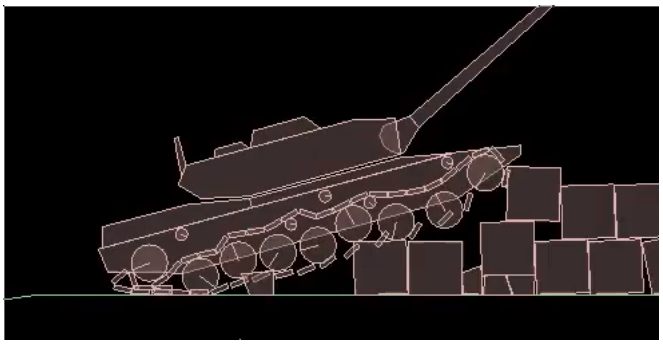

Figura 4.18: Simulación de un vehículo en Box2D.

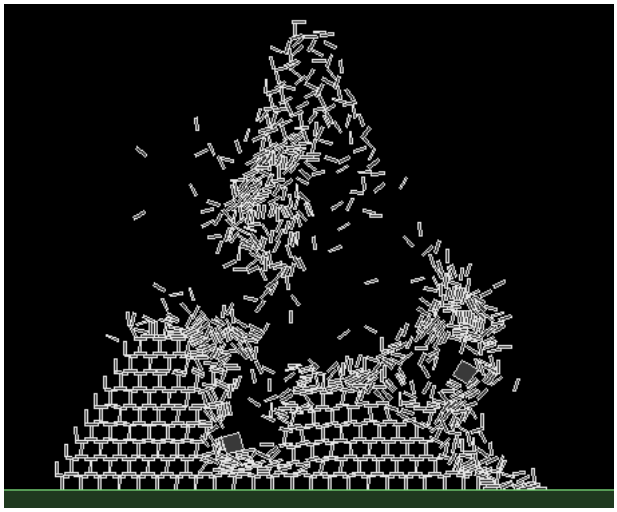

Figura 4.19: Simulación de partículas en Box2D.

Box2D está orientada principalmente para ser integrada en videojuegos 2D, ha sido creada y es mantenida por una sola persona, *Erin Catto*. También ha sido portada a múltiples lenguajes como **C#, Java, JavaScript, Flash, Delphi, Python** y dispositivos como **Nintendo DS y BlackBerry**. En este proyecto, se utiliza el port de JavaScript llamado **Box2DWeb**. *Box2DWeb* fue portado a JavaScript desde el port de *Flash*, *Box2DFlash* mediante un transpilador *Actionscript3-to-JavaScript*.

Más adelante en la sección 5.4 se continua analizando esta biblioteca de forma un tanto más exhaustiva.

5

Análisis de objetivos y metodología

En este capítulo se verán cual es el problema que plantea este proyecto, la tecnología usada y la metodología llevada a cabo para alcanzar los objetivos. En cuanto al análisis de bibliotecas necesarias solo nos centraremos en analizar las soluciones para **renderizado**, **simulación de físicas**, **creación de scripts** y **gestión de escenas** que son los cuatro bloques de requisitos que el proyecto necesita cubrir.

5.1 Análisis de los requisitos

Nos situamos frente a un proyecto que trata de ofrecer una fachada al usuario (programador), que le permita crear juegos 2D de una manera cómoda sin tener que lidiar con bibliotecas de bajo nivel como *WebGL* o *Box2D*. Por tanto los requisitos que este proyecto necesita son los siguientes:

- Operaciones básicas para *Vectores* y *Matrices*.
- Renderizado 2D sobre un elemento `<canvas>`.
- Texturizado de objetos 2D.
- Recorte de texturas.
- Soporte de *Texture Atlas*.
- Animación de *spritesheets*.
- Simulación de objetos físicos 2D.
- Detección y Respuesta de colisiones 2D.
- Colisiones para Polígonos y Círculos.
- Comportamiento ante colisiones personalizable mediante *callbacks*.
- Creación de comportamientos mediante *Scripts*.
- Abstracción de los objetos del juego mediante entidades y componentes.
- Gestión básica de escenas. Crear y cambiar de escena.
- Inserción dinámica de objetos en la escena. (Mientras se ejecuta el juego).

5.2 Análisis de posibles soluciones

5.2.1 Renderizado

En este apartado corresponde realizar una comparativa entre las dos tecnologías predominantes en cuanto al renderizado en navegadores se refiere: **WebGL** y **Canvas 2D**. Ambas introducidas en la sección 4.

Primero hablemos de *Canvas 2D*, como ventajas, esta *API* ofrece una fachada más sencilla para el usuario y está soportada por todos los navegadores. Y como inconvenientes, al tratar de renderizar muchos objetos, estamos hablando de más de 100 objetos, *Canvas 2D* reduce significativamente su *framerate*, este descenso se agrava si tratamos de renderiar más elementos.

Por otro lado *WebGL*, al ser acelerado mediante el hardware gráfico de la máquina cliente, ofrece un mayor rendimiento, pudiendo renderizar cientos de objetos sin sufrir ningún deterioro en su framerate. *WebGL* también goza de una buena tasa de soporte, aproximadamente el **96 %** de los navegadores soportan esta *API*.

El pipeline de *WebGL* es programable, lo que nos ofrece un mayor control sobre el proceso de renderizado, abriendo la posibilidad a introducir optimizaciones que beneficien el framerate. Pero es este mayor grado de libertad el que introduce un mayor grado de complejidad. *WebGL* es más complejo de leer y comprender que *Canvas 2D*, pero es esta falta de control la que provoca que *Canvas 2D* falle En cuanto a rendimiento. Aunque es mencionable que para pequeños juegos, siempre es más práctico optar por la *API* más sencilla, pero en este caso nos interesa ofrecer algo que apueste por el rendimiento, por tanto, **elegiremos WebGL** para este proyecto.

Para finalizar con esta comparativa, me gustaría añadir un par de ventajas un tanto personales, *WebGL* me atrajo para este proyecto porque yo ya conocía y había trabajado con **OpenGL**, su hermano mayor de escritorio, por tanto, podía portar las ideas desarrolladas en *OpenGL* a *WebGL* con poco esfuerzo (relativamente). También, si realizaba este proyecto en *WebGL*, todo lo aquí aprendido, podía volver a transladarlo a *OpenGL*.

WebGL se analiza a fondo más adelante en la sección 5.3.

5.2.2 Simulación Física

Para este proyecto se han barajado diferentes bibliotecas para simulación de físicas, entre ellas, *Box2D*, *matter.js*, *p2.js* y *Physics.js*. Los tres últimos son bibliotecas modernas, escritas desde cero en JavaScript, están terminadas, pero aún siguen encontrando algunos bugs o mejoras que aplicar. Todas estas bibliotecas, ofrecen prácticamente las mismas *features* y satisfacen de sobra los requisitos del proyecto.

En cambio *Box2D* lleva casi once años, dando soporte a muchos juegos famosos de la industria por tanto ha sido testeada bajo muchas situaciones.

Box2DWeb es la versión en JavaScript de esta biblioteca, por tanto me parece la mejor opción. *Box2DWeb* se analiza más adelante en la sección 5.4.

5.2.3 Gestión de escenas

Para implementar la gestión de escenas no se usarán bibliotecas externas, sino que se programará desde cero. La idea consistirá en poder tener una clase `Scene` compuesta por los objetos de la escena, a los que denominaremos `GameObject`. Un `GameObject` puede representar cualquier objeto dentro de nuestra escena.

Los `GameObject` se podrán añadir y eliminar de la escena. Un `GameObject` podrá tener diferentes propiedades, como propiedades físicas (posición, velocidad, geometría de colisión), propiedades gráficas (textura, color) o propiedades como *scripts* para dar interactividad e inteligencia artificial al objeto.

La solución diseñada se explica en las secciones 6.3 y 6.10.

5.2.4 Scripts

Un *script* puede verse como un comportamiento asociado a un objeto de la escena. Este *script* define como se comporta ese objeto, por ejemplo, como interactua ante una colisión con otro objeto, o qué acción realiza en cada actualización del bucle del juego.

Esta será la forma de proporcionar **inteligencia artificial** a los objetos de una escena. Este sistema será implementado desde cero.

5.3 Solución escogida para Renderizado: WebGL

En esta sección analizaremos *WebGL* para su mayor comprensión.

5.3.1 Concepto matemáticos básicos

Para poder entender y usar una API como OpenGL (o su versión web en este caso) es necesario entender una serie de conceptos matemáticos, sobre los cuales, se basa OpenGL. Estos conceptos se encuentran dentro de las ramas de las matemáticas denominadas **álgebra lineal** y **geometría** (especialmente **geometría proyectiva)**. En estas ramas se estudian conceptos como los vectores, matrices, espacios vectoriales, transformaciones lineales, proyecciones ortogonales y en perspectiva, etc. A continuación se explican estos conceptos.

5.3.1.1 Sistemas de coordenadas

Sistema de coordenadas

- En geometría, un sistema de coordenadas [40] es un sistema que utiliza uno o más números (coordenadas) para determinar unívocamente la posición de un punto o de otro objeto geométrico.

En OpenGL, se definen un sistema en el que las coordenadas **x**, **y**, **z**, están acotadas en el rango **[-1.0,1.0]**, estas coordenadas serán mapeadas a la ventana en la que se esté dibujando, por tanto los objetos situados fuera de esas coordenadas no serán visibles. Este tipo de coordenadas se conoce en OpenGL como **normalized device coordinates** o **NDC**.

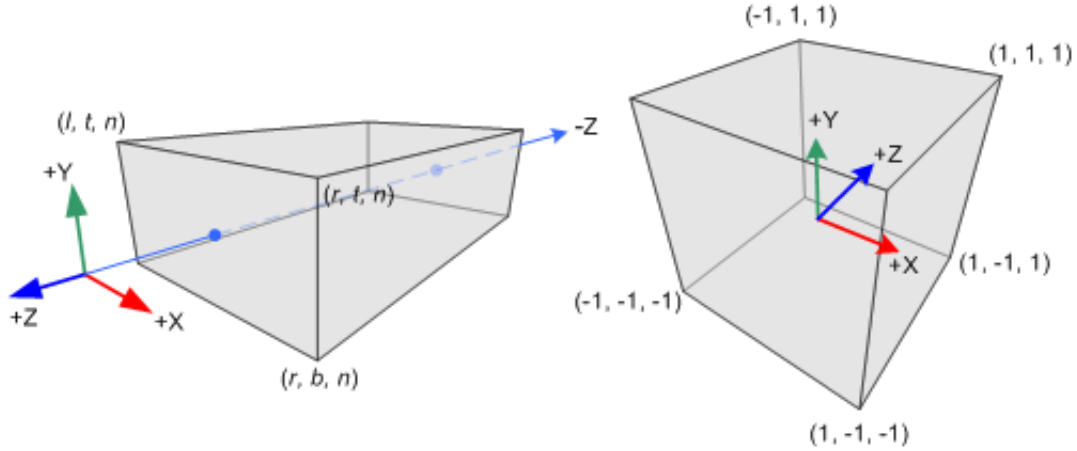

Figura 5.1: Normalice Device Coordinates o NDC

Pero en OpenGL el usuario puede trabajar en el rango de coordenadas que prefiera, será más tarde, en una fase del pipeline de renderizado, dónde los elementos definidos por el usuario serán transformados al sistema NDC (quedando acotados en [-1.0, 1.0]).

En OpenGL se tratan con varios sistemas de coordenadas diferentes, en concreto, los siguientes (en inglés, para no desviarnos de la nomenclatura de OpenGL): **Local space, World space, View space, Clip space y Screen space.**

Para pasar de un sistema a otro, se transforman los vértices de los objetos mediante una matríz de transformación, este concepto se tratará más adelante. Por el momento basta con entender que permiten cambiar de sistema de coordenadas.

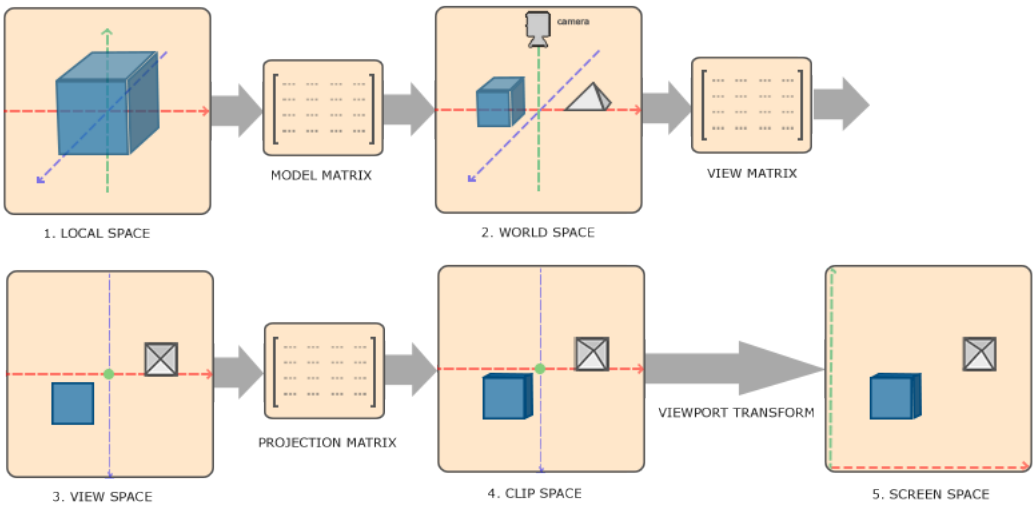

Figura 5.2: Diferentes espacios de coordenadas a los que son transformados los objetos.

- El *Local space*, representa las coordenadas que son **relativas** al origen de coordenadas del objeto. Por lo general, los objetos se suelen crear con el centro en la coordenada origen (0,0,0).

- El *World space*, representa las coordenadas del mundo, en este caso todas las coordenadas son relativas al origen del mundo. Por tanto todos los objetos del mundo están situados en relación con el centro de este mundo que suele ser el (0,0,0).

- El *View space*, representa las coordenadas desde el punto de vista del observador del mundo o escena, este observador se suele conocer como la *cámara*.

- En el *Clip space*, las coordenadas del *View space* son procesadas al rango [-1.0,1.0].

- En el *Screen space*, las coordenadas del *Clip space* transformadas a los rangos de la ventana o pantalla (resoluciones de pantalla que nos son familiares pueden ser 800x600 o 1024x768, por ejemplo).

5.3.1.2 Coordenadas homogéneas

Dado que en OpenGL estamos trabajando en un espacio denominado **espacio proyectivo**, para describir puntos en este espacio vamos a necesitar más coordenadas que las clásicas **x,y,z**. A cada punto del espacio cartesiano, se le añade una coordenada extra **w** obteniendo **coordenadas homogéneas**[43]. Esta coordenada *w* se denomina factor de escala, y permite traducir un punto homogéneo a su versión original cartesiana.

Esta traducción se realiza mediante la siguiente operación :
$(x, y, z, w) => (x/w, y/w, z/w)$.

Esto provoca que podamos tener infinitas representaciones del punto (x,y,z). Puede que esto ya nos suene al concepto de tener varios observadores en una misma escena observando el mismo punto 3D, cada uno lo proyectará de una forma diferente en su pantalla 2D, cada uno tendrá su representación del mismo punto 3D.

5.3.1.3 Vectores, Matrices y Transformaciones

> *I understand how the engines work now. It came to me in a dream. The engines don't move the ship at all. The ship stays where it is and the engines move the universe around it.*
>
> *(-Cubert Farnsworth, Futurama)*

Representación de un punto en el espacio mediante un vector
- Cada punto del espacio es representado mediante un vector de 4 componentes (x,y,z,w).

Es importante comprender que en el ámbito de OpenGL el término punto y vértice pueden ser usados como sinónimos, siendo estrictos, un punto es una posición en el espacio y un vértice es un punto que pertenece a un modelo geométrico como un rectángulo, un triángulo o al modelo 3D del protagonista de

tu juego de estrategia en tiempo real favorito. No obstante ambos están representados por un *vector* y especifican una posicion en el espacio.

Transformaciones afines

- Las transformaciones son lo que hacen posible mover, rotar y escalar objetos. También son las encargadas de la proyección de coordenadas 3D sobre superficies 2D.

Podemos pensar en una transformación como en una matriz que representa un *espacio*, si multiplicas un vector (o punto) por esa matriz, entonces el punto se transofrma al espacio representado por la matriz. Imaginemos que creamos una matriz de rotación, que indica una rotación de 20º en el eje x. Si multiplicamos esa matriz por un punto, el resultado será un nuevo punto, equivalente al primero, solo que rotado 20º en el eje x.

Existen varias transformaciones con las que trabajamos en OpenGL: **model transformations, viewing trasnformations y projection transformations**. Cada transformación tiene **asociada** una **matriz** específica.

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix} \tag{5.1}$$

Ecuación 5.1: Multiplicación de matriz por vector.

Las **model trasformations** son las encargadas de **mover, rotar y escalar** los objetos, para conseguir esto, se **aplica la matriz de transformación a todos los vértices del objeto**. Si bien es cierto, existe una estructura matemática mejor para representar rotaciones que las matrices. Los llamados **cuaterniones**, su principal ventaja es que evitan el llamado *Gimbal Lock*, esto es, la pérdida de un grado de libertad en un entorno tridimensional al colocar en una configuración paralela. En este proyecto, usaremos matrices para representar rotaciones, pues nuestro entorno es 2D y no nos afecta el *Gimbal Lock*.

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

Ecuación 5.2: Matrices para mover y escalar.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_y = \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_z = \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

Ecuación 5.3: Matrices para rotar en los ejes x,y,z. Una matriz por eje.

Las **viewing trasformations** son las encargadas de transformar el objeto en relación a la posición y rotación de la cámara. La matriz para esta transformación se obtiene de la **posición** y los vectores **up, right y forward** de la cámara.

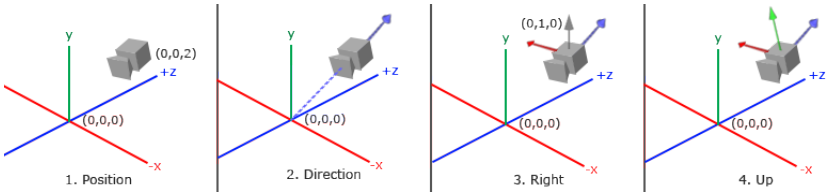

Figura 5.3: Vectores con los que se crea la matriz view.

$$\begin{bmatrix} right_x & up_x & forward_x & pos_x \\ right_y & up_y & forward_y & pos_y \\ right_z & up_z & forward_z & pos_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

Ecuación 5.4: La matriz para Viewing transformation.

Las **projection trasformations** definen el volumen de dibujado y los planos de clipping. Todo esto se conoce como el **frustum**. Existen dos proyecciones diferentes: **orthogonal y perspective**. La proyección ortogonal ignora la profundidad de los objetos por lo que se suele usar para aplicaciones o juegos en 2D, mientras que la perspectiva si tiene en cuenta la lejanía de los objetos, por tanto se suele usar en aplicaciones 3D.

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

Ecuación 5.5: La matriz de proyección ortogonal.

*Variables para expresar la matriz de proyección en perspectiva en función de **near, far, top y bottom**:*

$$top = near * \tan(PI/180) * (fov/2)$$
$$bottom = -top$$
$$right = top * aspect$$
$$left = -right$$

$$\begin{bmatrix} \frac{2*near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2*near}{top-bottom} & -\frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{5.6}$$

Ecuación 5.6: La matriz de proyección perspectiva.

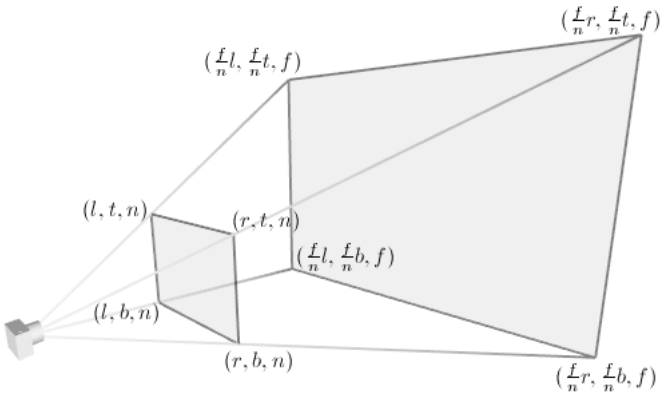

Figura 5.4: Ejemplo de Frustum de la proyección perspectiva.

5.3.2 Pipeline de renderizado

Ahora ya sabemos como transformar los vértices de un objeto geométrico. Pero, **¿cúando usar estas transformaciones en OpenGL?**.

Estas transformaciones se aplicarán en una fase concreta del pipeline, llamada **vertex shader**, a continuación se explican todas las fases.

Pipeline

» El pipeline[46] son una serie de etapas, en las cuales, la salida de una etapa sirve de entrada para la siguiente, como si de una cadena de montaje se tratara. El objetivo de esta cadena de etapas es construir una imagen a partir de los datos de entrada. Entendiendo los datos de entrada como los objetos de una escena, sus vértices, texturas, posiciones, etc.

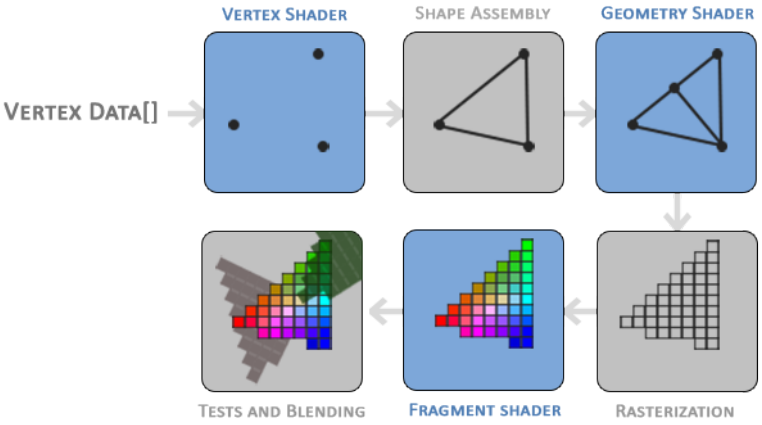

Figura 5.5: Fases del pipeline de renderizado.

Todo comienza enviando la infomación de los vértices a la memoria del hardware gráfico a través de OpenGL, esto se hace creando unos **buffers** con todos los vértices y pasándoselos a cierta función de OpenGL que veremos más adelante. Pero de momento basta con entender que pasamos una colección de vértices al hardware gráfico.

- **Vertex shader**: Esta fase es **programable** y es donde **se aplican las transformaciones a los vértices**.
- **Shape assembly**: Esta fase usa los vértices transformados de la fase *Vertex shader* y los ensambla en una forma primitiva como puede ser **punto, línea, o triángulo**.
- **Geometry shader**: Esta fase es **programable** y no profundizaremos mucho en ella, pero básicamente sirve para crear nuevos vértices en la figura sin necesidad de haberlos introducido en el buffer original.
- **Rasterization**: Convierte cada primitiva en un conjunto de fragmentos. Los fragmentos están alineados con los píxeles de la pantalla.
- **Fragment Shader**: Esta fase es **programable** y es donde se procesa cada fragmento de forma individual. Aquí es dónde se le aplica al fragmento una textura o color.
- **Test and blending**: Por último esta fase comprueba los fragmentos que tapan a otros así como las transparencias y los mezcla para crear la imagen final 2D proyectada en la pantalla.

Si el lector desea completar su información acerca de como trabajar con un **pipeline programable** en WebGL, puede dirigirse al Anexo 8.1.

5.4 Solución escogida para la Física: Box2DWeb

En esta sección trataré de explicar los **conceptos básicos** de *Box2D* (también váildos para *Box2DWeb*). Como características generales de *Box2D* podemos encontrar, **CCD** o *detección continua de colisiones*, **callbacks** en cada momento de la de colisión (begin, end, pre-solve, post-solve), `Shapes` para polígonos convexos y círculos, múltiples `Fixtures` por `Body`, **Dynamic AABB Tree** (árbol de jerarquía de figuras generalizadas mediante AABBs) para la *Broadphase*, **SAT** para la *Narrowphase*, grupos de colisiones y categorias (en caso de querer restringir qué objetos colisionan con cuáles otros).

Pero para explicar la esencia de *Box2D* considero mejor realizar un recorrido a través de las clases más importantes (para el usuario) de *Box2D*: `World`, `Body`, `Fixture`, `Shape`, `Constraint` y `Joint`.

5.4.1 World

World

- La clase principal de *Box2D* es la clase `World` [11]. Se compone de una colección de `Body`, `Fixture` y `Constraint`, es decir, de cuerpos, formas geometricas de colisión y restricciones para los cuerpos.

Estos tres tipos de elementos interactuan entre sí como ahora veremos. Todo cuerpo debe ser creado o eliminado desde la clase `World`, de modo que esta clase controla todos las reservas de memoria de estos objetos.

Desde esta clase podemos crear y destruir cuerpos, definir la gravedad, buscar fixtures concretas o lanzar *Ray Cast* para averiguar con qué fixtures interseca un rayo.

5.4.2 Body

Body

» Esta clase [12] representa una entidad con propiedades físicas. No tiene ninguna forma geométrica asignada, ni propiedades de colisión (Esto se hará mediante la clase `Fixture`). Puede ser de tres tipos diferentes: *Static*, *Dynamic* y *Kinematic*.

Las propiedades físicas que podemos elegir para un `Body` son:

- **mass**: La masa del objeto.
- **velocity**: Vector velocidad del objeto, indica cómo de rápido se mueve y en qué dirección.
- **rotational inertia**: Cuanto esfuerzo hay que hacer para que el objeto comience a rotar.
- **position**: La posición en el espacio 2D del objeto.
- **angle**: Orientación del objeto.

Los diferentes tipos de objetos se definen de la siguiente manera:

- **Static**: Objeto estático. No se mueve, la propiedad de velocidad no tiene efecto en este objeto.
- **Dynamic**: Objeto dinámico. Este objeto se verá afectado por la física de las fuerzas que se le apliquen. Por tanto se moverá con una velocidad y una aceleración.
- **Kinematic**: Objetos cinéticos. Estos objetos no se ven afectados por las fuerzas que se le apliquen. En cambio si tienen movimiento.

5.4.3 Fixture

Fixture

» Esta clase [13] es usada para definir el tamaño, la forma y las propiedades del material de un objeto, como la restitución, la fricción y la densidad.

Enlaza un `Shape` con un `Body` y le añade propiedades físicas como fricción, restitución y densidad. Un `Body` puede poseer diferentes `Fixture`, estos afectan al centro de gravedad del `Body`.

Para definir la forma del objeto se puede elegir entre diferentes valores de la clase `Shape`. Esta clase representa la forma geométrica 2D, como un círculo o un rectángulo. Sus valores son: `ChainShape`, `CircleShape`, `EdgeShape`, `PolygonShape`.

5.4.4 Joint

Joint

» Esta clase [14] es usada para *atar* dos o más objetos juntos. Existen varios tipos de Joints: `Revolute`, `Distance`, `Prismatic`, `Line`, `Weld`, `Pulley`, `Friction`, `Gear`, `Mouse`, `Wheel`, `Rope`.

Para que se entienda la finalidad de un `Joint`, aquí expongo algunas definiciones.

- **Revolute**: crea un punto, un pivote, sobre el que rotan los objetos.
- **Distance**: mantiene dos objetos a cierta distancia fija.
- **Prismatic**: fija la rotación relativa de dos cuerpos.
- **Weld**: mantiene dos cuerpos con la misma orientación.

5.5 Metodología

5.5.1 Kanban

Para el control de las tareas en las que se subdividió este proyecto, se hace uso de la herramienta *Trello*. La cual permite crear tableros en los que clasificar unas tareas u otras. Esta aplicación se ajusta muy bien a la metodología *Kanban*, pero en vez de adoptar el clásico tablero compuesto por las columnas *ToDo*, *Doing* y *Done*, se ha querido adoptar un sistema, que, aún siendo un poco más complejo, permite más flexibilidad.

En concreto, a mi, me ha permitido aumentar el número de estados por el que pasan las tareas y a su vez, clasificar estas tareas, según el módulo al que pertenecen (render, físicas, scripts, matemáticas...).

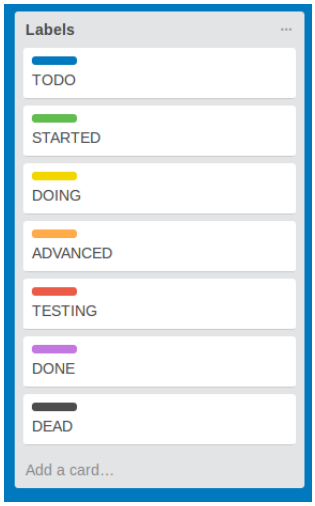

Figura 5.6: Control de tareas mediante Trello. Cada color indica un estado diferente de la tarea.

Como se puede apreciar en las imágenes, cada tarea posee una serie de etiquetas que indican el estado en el que se encuentra. Conforme la tarea avanza, se le van añadiendo etiquetas de estado hasta que se termina. Las tareas con la etiqueta azul son las que están por hacer, mientras que las que han llegado hasta la etiqueta lila, son las finalizadas. La etiqueta negra, significa que la idea ha sido descartada.

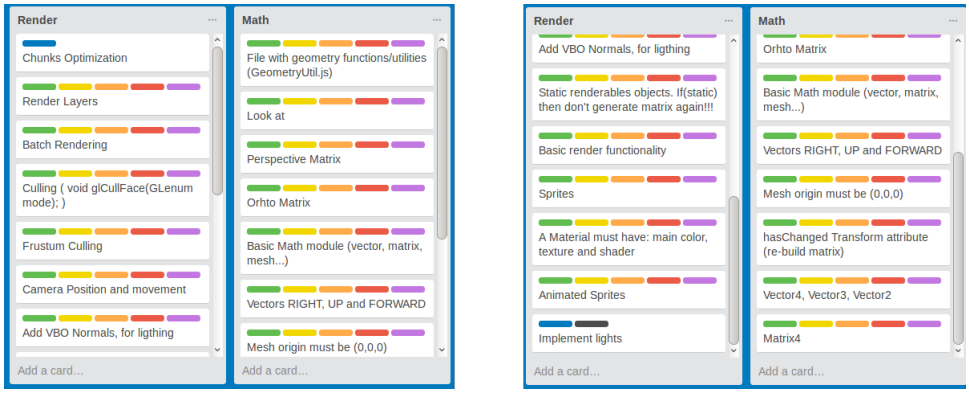

Figura 5.7: Tareas del módulo de Render y Matemáticas.

Figura 5.8: Tareas del módulo de Scripting y GameObjects.

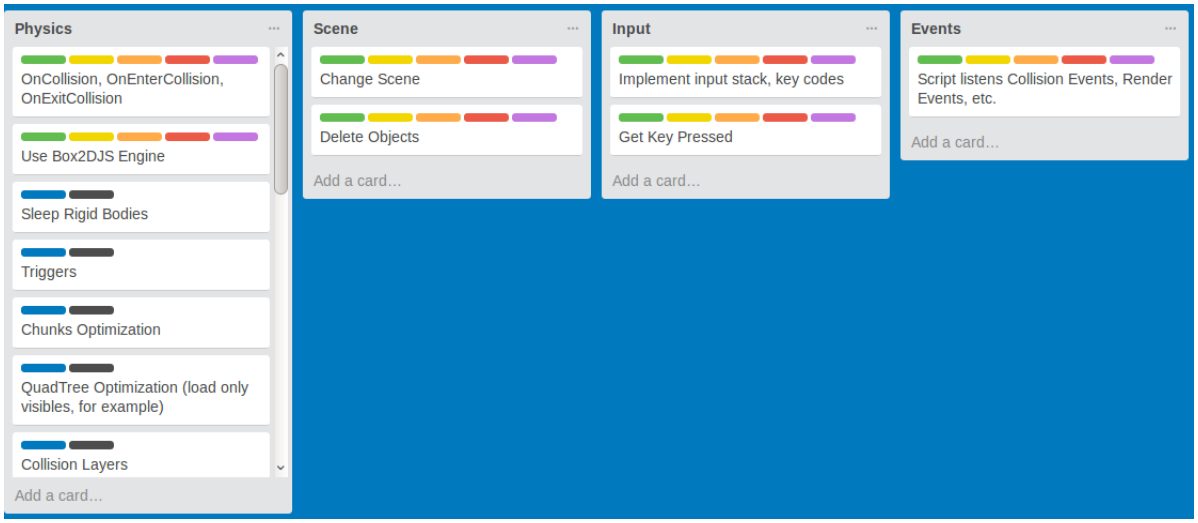

Figura 5.9: Tareas de Física, Gestión de escenas, Entrada de usuario y Eventos.

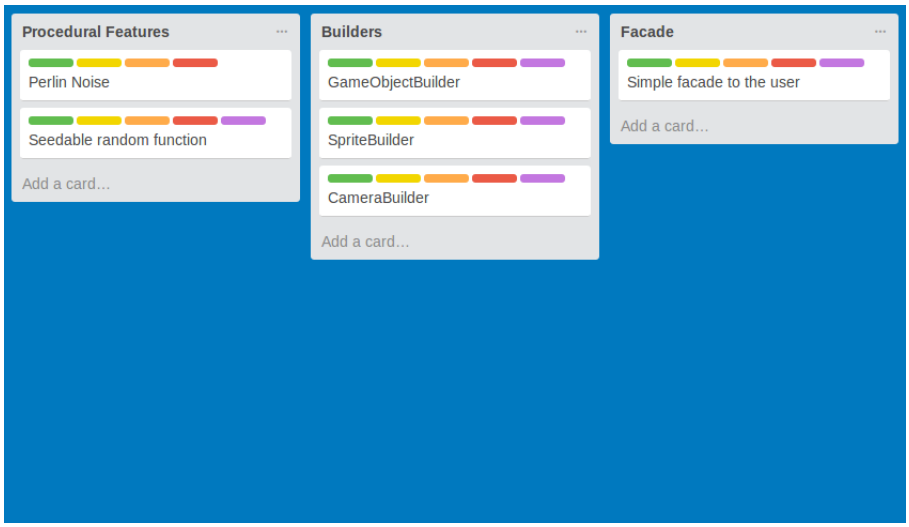

Figura 5.10: Tareas sobre Generación procedural, Builders y Fachada al usuario.

Figura 5.11: Tareas varias.

5.5.2 Control de Versiones: git y GitHub

Se hace uso de la herramienta *git*, para el control de versiones, así como del servicio *GitHub* para el alojamiento web del repositorio remoto. Puede consultarse a través de esta *url* https://github.com/AdrianECom/ThiefEngine.

5.5.3 Testing

En este proyecto no se ha seguido un *TDD* estricto, no se ha construido un conjunto de pruebas como tal, que el sistema debe pasar tras cada cambio. Sino que se ha construido una **demo**, la cual debe funcionar correctamente tras cada cambio en el *Game Engine*. Esta demo ha ido creciendo al mismo tiempo que crecían las capacidades del *Game Engine*.

El test en cuestión puede encontrarse en el repositorio del proyecto, dentro de la carpeta *test*, llamado `test.js`. En este programa, se crea una escena en la que se comprueba el correcto funcionamiento del *Game Engine*.

Figura 5.12: Test en ejecución.

En el test se crea un jugador, objetos dinámicos destructibles y un objeto estático, los cuales, se indican en la imagen 5.13. El jugador puede ser controlado mediante las flechas de dirección del teclado, mientras que los objetos desctructibles, son destruidos al colisionar con el jugador. El objeto estático no se destruye ni se mueve aunque colisione con el jugador. Al pulsar la tecla espacio, se crean más soldados dinámicamente, esto sirve para crear una masiva cantidad de estos y comprobar el rendimiento del *Game Engine* al añadir una gran cantidad de objetos a la escena.

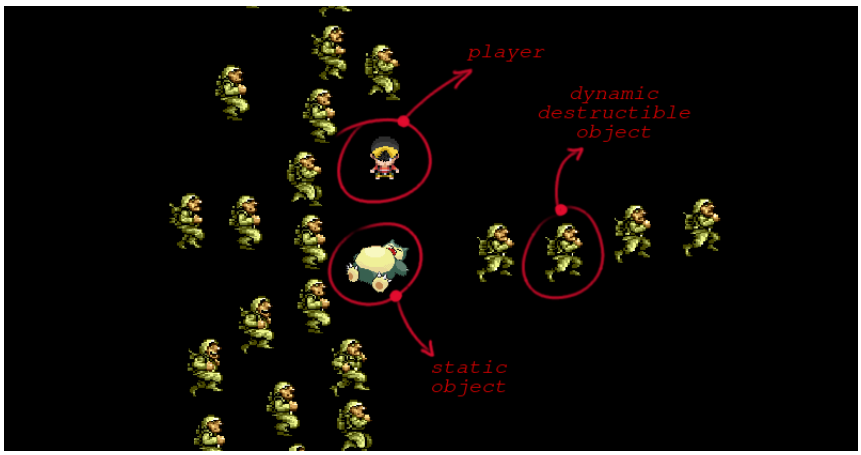

Figura 5.13: Objetos del test.

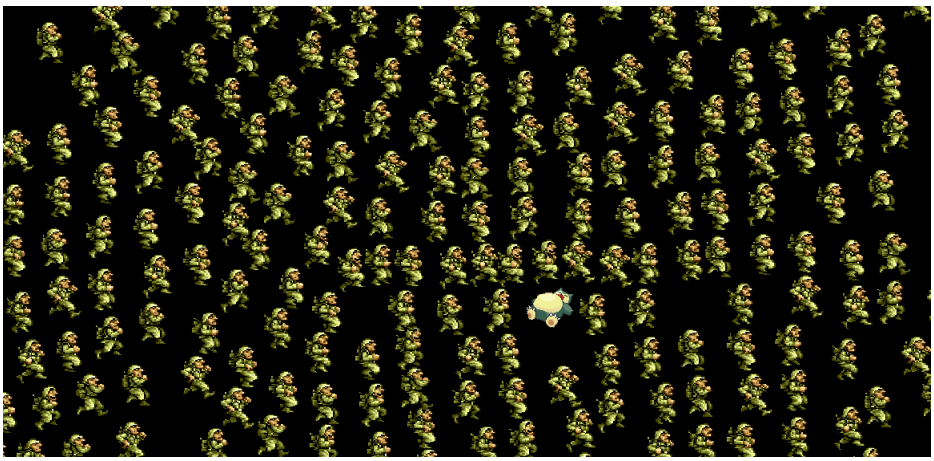

Figura 5.14: Test de rendimiento.

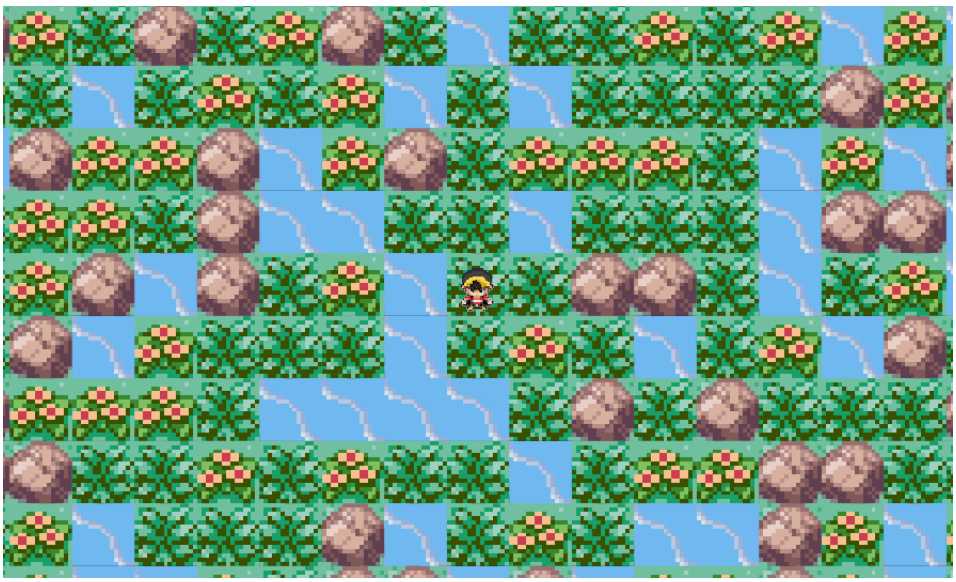

Figura 5.15: Test de múltiples texturas.

6

Diseño y resolución del trabajo realizado

En este capítulo se mostrará la estructura completa del *Game Engine*, se estudiarán las capas de este, desde un enfoque *Bottom-Up*, desde las capas bajas hasta las superiores. Pero antes de comenzar, debemos comprender como, de forma general, funciona un *Game Engine*.

Para este proyecto me he inspirado en muchos *Game Engines* famosos como pueden ser **Unity 3D**, **Unreal Engine**, **CryEngine** o bibliotecas como **LibGDX** u **Ogre3D**. Todos estos *Game Engines* se basan en los conceptos de **GameObject** y **Component**.

`GameObject` no es más que un **contenedor** que representa a los objetos de nuestro juego y a este contenedor, se le van incluyendo componentes que le proporcionan diferentes capaciadades, como renderizado, colisiones, IA, efectos, audio, entre otros.

Cada componente será gestionado por el **subsistema** correspondiente, es decir, los componentes de renderizado serán gestionados por el *Graphic Engine* (subsitema que se encarga de los gráficos), los componentes de colisión, serán gestionados por el *Physics Engine* y así con todos los tipos de componente. Esto se explicará más exhaustivamente en las siguientes secciones, pero ya tenemos suficiente como para poder comenzar a desgranar el *Game Engine*.

6.1 Herramientas utilizadas

Para este proyecto no se ha necesitado una gran cantidad de herramientas. El proyecto ha sido programado en JavaScript, HTML y GLSL (lenguaje de shader, ver anexo 8.1) Como editor de código, se ha utilizado el editor *Atom*, con diferentes *plugins* para el autocompletado del lenguage JavaScript.

Para la ejecución se ha usado el navegador web *Firefox*, bajo el sistema operativo *Ubuntu 16.04*. Para crear la documentación del código se ha hecho uso del paquete de *NodeJS*, *js-doc*. Para concatenar los diferentes ficheros de clases en un solo fichero, se ha utilizado un script *Shell* creado para este fin.

Para el diseño y generación de los diagramas de clases se ha utilizado el programa *Astah**.

6.2 Diseño por capas

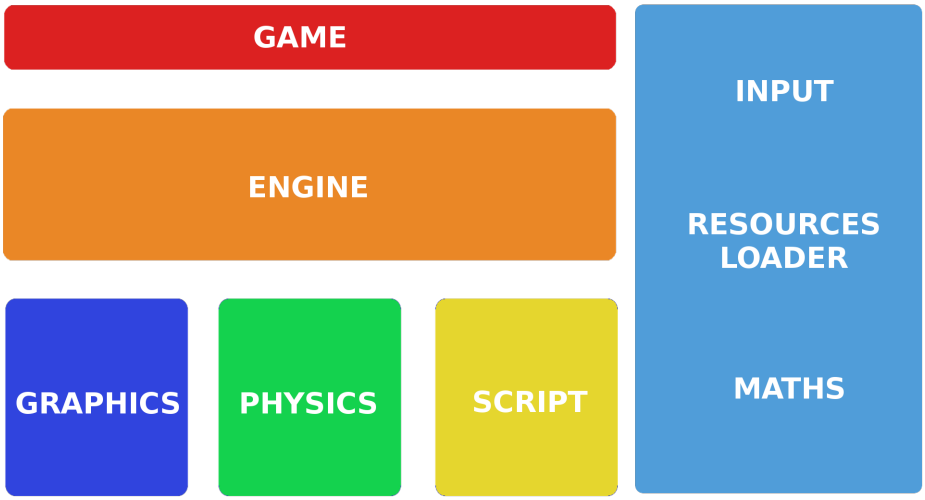

Figura 6.1: Capas del engine.

Game Engine por capas

- El *Game Engine* ha sido ideado como un conjunto de capas, las capas más bajas ofrecen funcionalidades más básicas hacia las capas superiores, las cuales aprovechan para realizar tareas más complejas.

Empezando desde abajo, las capas **Graphics, Physics** y **Script**, son los tres **subsistemas** del *Game Engine*, estos ofrecen diferentes capacidades a los objetos de la escena, capacidades de dibujado, de simulación física y de ejecución de scripts, respectivamente.

La capa **Engine** es la fachada al usuario, usada por este para crear sus juegos. En esta capa se encuentran las clases destinadas a la **gestión de escenas y entidades** como: `Scene`, `GameObject` y `Component`.

Las capas **Input**, **Resource Loader** y **Math** ofrecen procesamiento de la entrada del usuario, carga de recursos (por ejemplo, imágenes) y utilidades matemáticas, respectivamente. Estas tres últimas capas son transversales y sirven al resto de capas del motor.

La capa **Game**, es independiende del *Game Engine*, aquí es dónde el usuario crea su juego usando la fachada proporcionada.

6.3 Component-Entity-System

En el diseño de un *Game Engine* podemos encontrarnos clases como `Sprite` o `RigidBody`, siendo un `Sprite` un objeto que puede dibujarse y `RigidBody` un objeto que puede simular físicas y colisiones.

Imaginemos también, que esas clases son abstractas, y que de ellas heredan muchos tipos de *sprites* y muchos tipos de *rigid bodies*. Mientras esas jerarquías no se entrelacen, todo estará bien. Pero, **¿que**

pasa si queremos tener tipos de objetos que se puedan dibujar y que también reaccionen ante la física y las colisiones?.

En ese caso deberíamos hacer que clases que heredan de `Sprite` también lo hiciesen de `RigidBody`, de esta forma tendríamos objetos que se dibujan, y aparte, objetos que se dibujan y reaccionan ante las físicas. Pero esto provoca que la jerarquía de herencia explote a lo ancho de forma exponencial.

El ejemplo aquí expuesto, solo tenía dos jerarquías separadas, pero en el diseño de un *Game Engine* puedes encontrarte más de dos jerarquías, y esa explosión exponencial de clases se convierte en una explosión exponencial, prohibitiva.

CES [52] o *Component-Entity-System*, soluciona este problema. Es un patrón de diseño que prima la composición sobre la herencia. Los datos se organizan de la siguiente forma. Los **componentes** son la pieza que contiene la información importante. En un *Game Engine*, cada clase que proporciona propiedades a un objeto, como `RigidBody`, que proporciona capacidades físicas, o `Script` que proporciona reacción a eventos, serán componentes.

Estos componentes se añadirán a una **entidad** que hará de contenedor, en un *Game Engine*, las entidades serán los objetos del juego, es decir, los `GameObject`. Por tanto los `GameObject` serán contenedores vacíos a los que les añadiremos componentes para porporcionarles capacidades de renderizado, físicas o IA, entre otros.

Ahora, **aquí se demuestra como el patrón *CES* resuelve el problema que planteaba la herencia**. Si queremos tener objetos que posean varios tipos de propieddades (dibujado, físicas, IA, ...), solo tenemos que añadir los componentes pertinentes para agregar esas propiedades al objeto, sin necesidad de heredar de ninguna clase.

Cada componente, es manejado de forma separada por un **sistema** especializado. En nuestro caso hablamos del **RenderEngine**, **PhysicsEngine** y **ScriptEngine**. Cada uno de estos sistemas se encarga de los componentes que tienen que ver con el renderizado, la física y los scripts respectivamente.

En la imagen 6.2 podemos ver cómo se organizan estas clases. La clase `GameObject` estará compuesta por objetos `Component`. Ambos heredan de la clase `BaseObject`, esta clase lo único que proporciona es un **id** único. Bastante útil a la hora de identificar a los objetos y a los componentes. La clase `Scene` representa una escena y está compuesta por objetos `GameObjects`.

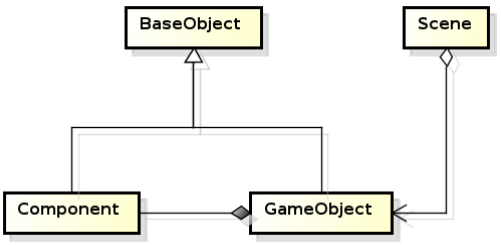

Figura 6.2: Módulo de GameObject.

6.4 Bucle principal y el manejo del Tiempo

El control del tiempo es vital, en ello se basa que un *Game Engine* se ejecute a un número fijo de **FPS** o *Frames per second*. Una buena forma de explicar el uso del tiempo, mediante el **main loop**[42] o *bucle principal* del *Game Engine*.

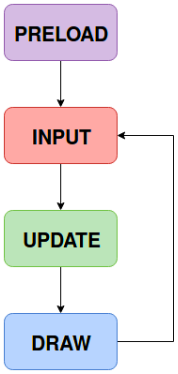

Figura 6.3: Esta es la forma general de representar un game loop.

Código 6.1: Game Loop simple en pseudocódigo.

```
1
2 while(true){
3   processInput();
4   update();
5   render();
6 }
```

El *Game Engine* diseñado en este proyecto, también sigue este esquema, no obstante, este esquema peca de ser demasiado general, un código real de un game loop es mucho más complejo, a continuación podemos ver el pseudocódigo del game loop usado en este proyecto.

Código 6.2: Game Loop complejo en pseudocódigo.

```
1
2 SECONDS_PER_FRAME = 1/FPS;
3
4 while(true){
5
6   deltaTime = getElapsedTime();
7
8   scriptEngine.update();
9
10  while( deltaTime > SECONDS_PER_FRAME ){
11
12    deltaTime = deltaTime - SECONDS_PER_FRAME;
13
14    physicsEngine.update(SECONDS_PER_FRAME);
15
16  }
17
18  renderEngine.render();
19
20 }
```

Analicemos como funciona este prodecimiento, lo primero que hace es obtener, el tiempo transcurrido durante el frame anterior, `deltaTime = getElapsedTime();` . Este tiempo se cononce como el **Delta Time**.

Después actualiza la lógica del juego, `scriptEngine.update();`, el **Script Engine**, es el subsistema que se encarga de manejar toda la lógica que el usuario crea en su juego, mediante los llamados **Scripts**, lo cual se explicará más adelante. Pero lo importante de este punto, es que es aquí, donde se **comprueba la entrada del usuario**, ya sea teclado, ratón, gamepad, etc.

Ahora le llega el momento de actualizar la física del juego, aquí es dónde se actualizarán las posiciones, velocidades y aceleraciones de los objetos del juego.

Código 6.3: Actualizar las físicas.

```
1
2 while( deltaTime > SECONDS_PER_FRAME ){
3
4   deltaTime = deltaTime - SECONDS_PER_FRAME;
5
6   physicsEngine.update(SECONDS_PER_FRAME);
7
8 }
```

Este bucle lo que hace es evitar que se actualicen las físicas, si el tiempo del que disponemos no es suficiente. Se comprueba si el tiempo que ha pasado es mayor que el tiempo que necesita un frame. Si la condición se cumple, se actualizan las físicas un paso.

Al ser un bucle, si seguimos teniendo tiempo, seguimos actualizando las físicas hasta quedarnos con un tiempo inferior al que necesitamos para actualizar las físicas. Una vez actualizadas las físicas, se procede, por último, a dibujar la escena.

6.5 Módulo de Matemáticas : Vectores y Matrices

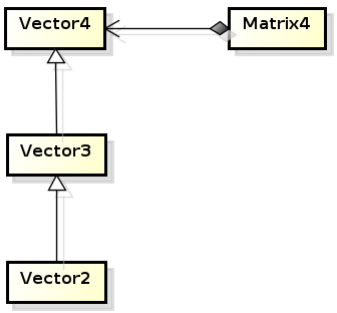

Figura 6.4: Módulo de Matemáticas.

La **computación gráfica** se fundamenta en ciertas áreas de las matemáticas, como son el **álgebra lineal** y la **geometría** para representar objetos y proyectarlos en el monitor. En concreto, estructuras matemáticas como los **vectores y las matrices** son la base de los cálculos realizados en cualquier videojuego o aplicación gráfica. Esto se analiza en la sección 5.3.

Más allá de los gráficos, vectores y matrices son también usados en los cálculos para la simulación

de físicas. Posiciones, fuerzas, velocidad o aceleración son algunas de las magnitudes representadas mediante vectores.

Es por ello que cualquier *Game Engine* debe traer consigo ciertas clases o estructuras de datos que permitan modelar estos cálculos.

En este proyecto, el módulo de matemáticas, es la base de todo el *Game Engine* y ofrece las clases `Vector4`, `Vector3`, `Vector2` y `Matrix4`, así como aritméticas de vectores y de matrices, que, como he dicho, son la base de todas las operaciones realizadas en los engines de gráficos y físicas.

La clase `Vector4` es un vector de **4 componentes**, internamente posee cuatro variables, que son, `x`, `y`, `z` y `w`. La clase `Vector3` hereda de `Vector4` fijando el valor de `w` a `1`. A su vez, `Vector2` hereda de `Vector3` fijando el valor de `z` a `0`. De modo que las tres clases de vectores pueden tratarse como si fueran `Vector4`.

Código 6.4: Clases de vectores.

```
1
2 var Vector4 = function (x,y,z,w){
3     this.x = x;
4     this.y = y;
5     this.z = z;
6     this.w = w;
7 };
8
9 var Vector3 = function (x,y,z){
10     Vector4.call(this,x,y,z,0);
11 };
12
13 var Vector2 = function (x,y){
14     Vector3.call(this,x,y,0);
15 };
```

La clase `Matrix4`, representa una matriz de 4x4, estas matrices se utilizan en las **transformaciones lineales** tal y como se explica en el apartado 5.3.1.3. *WebGL* requiere que las matrices sean enviadas, a los *shaders*, **por columnas**, por ello la clase `Matrix4` ha sido implementada por columnas, para tener que evitar la operación de transposición antes de enviar la matriz al shader.

Esta decisión de diseño supone una optimización importante en el *Game Engine*, pues cada objeto del juego poseerá una matriz asociada, si la clase `Matrix4` se hubiera implementado por filas, cada matriz de cada objeto tendría que ser transpuesta, pero gracias a implementar la matriz por columnas, los datos de la matriz pueden enviarse tal cual, ahorrándonos así una operación.

Internamente `Matrix4` es un **array de 16 elementos**, en lugar de ser un **array de 4 vectores**. Esto beneficia tanto la localidad espacial como la localidad temporal, pues los 16 elementos de la matriz se almacenarán en memoria de forma consecutiva.

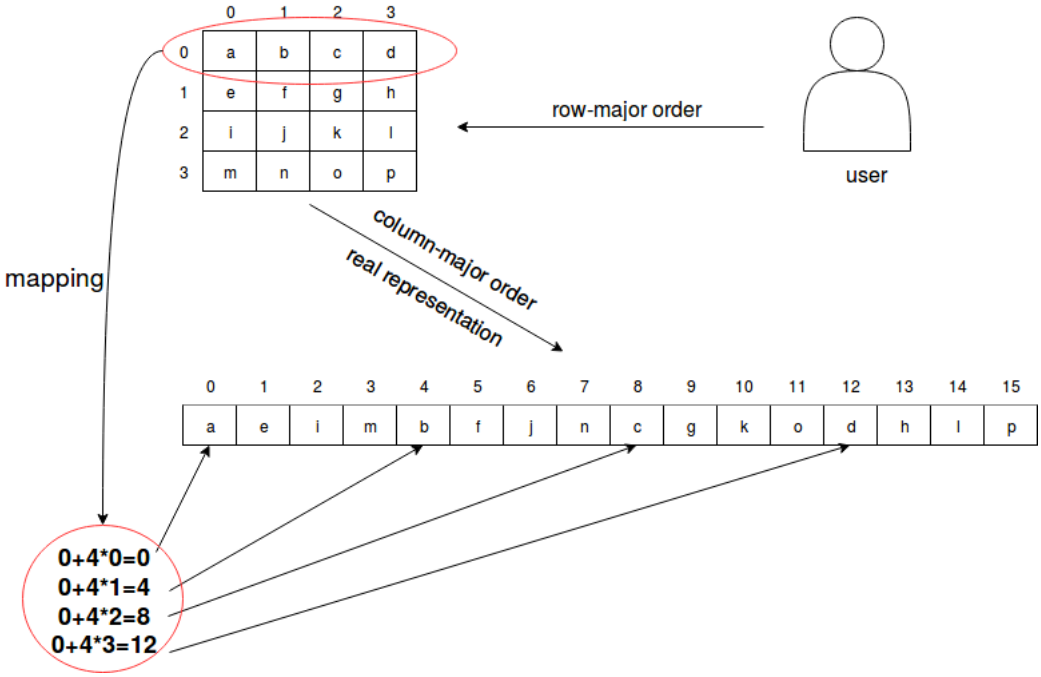

Figura 6.5: El usuario introduce una matriz por filas, pero se representa internamente por columnas.

6.6 Componente Transform

Antes de continuar y una vez explicado el módulo de matemáticas, hablaré sobre el componente más importante, `Transform`. Este componente alberga la **posición, rotación y escala** del `GameObject`, de modo que esta información queda centralizada y accesible para el resto de componentes. Gracias a esto todos los componetnes obtienen la misma información de posición, rotación y escala, en lugar de albergar cada uno su propia copia de estos datos.

La posición, rotación y escala están representadas mediante vectores, por tanto, se puede operar sobre estos datos mediante aritmética de vectores. También es la clase que se encarga de generar las **matrices** de translación, rotación y escalado, a partir de esos valores. Estas matrices serán usadas por *WebGL* en sus cálculos. (Ver sección 6.7.6).

Está diseñada para generar las matrices única y exclusivamente cuando los datos de posición, rotación o escala cambian, en vez de hacerlo cada *frame* del juego. Esto es una mejora de rendimiento muy importante, ya que cuantas menos matrices se (re)generen, mejor tasa de frames podrá ofrecer el *GameEngine*.

Esto sirve para objetos estáticos, estos objetos solo necesitan generar su matriz, una vez, no cada *frame*.

6.7 Módulo de Gráficos : Render Engine

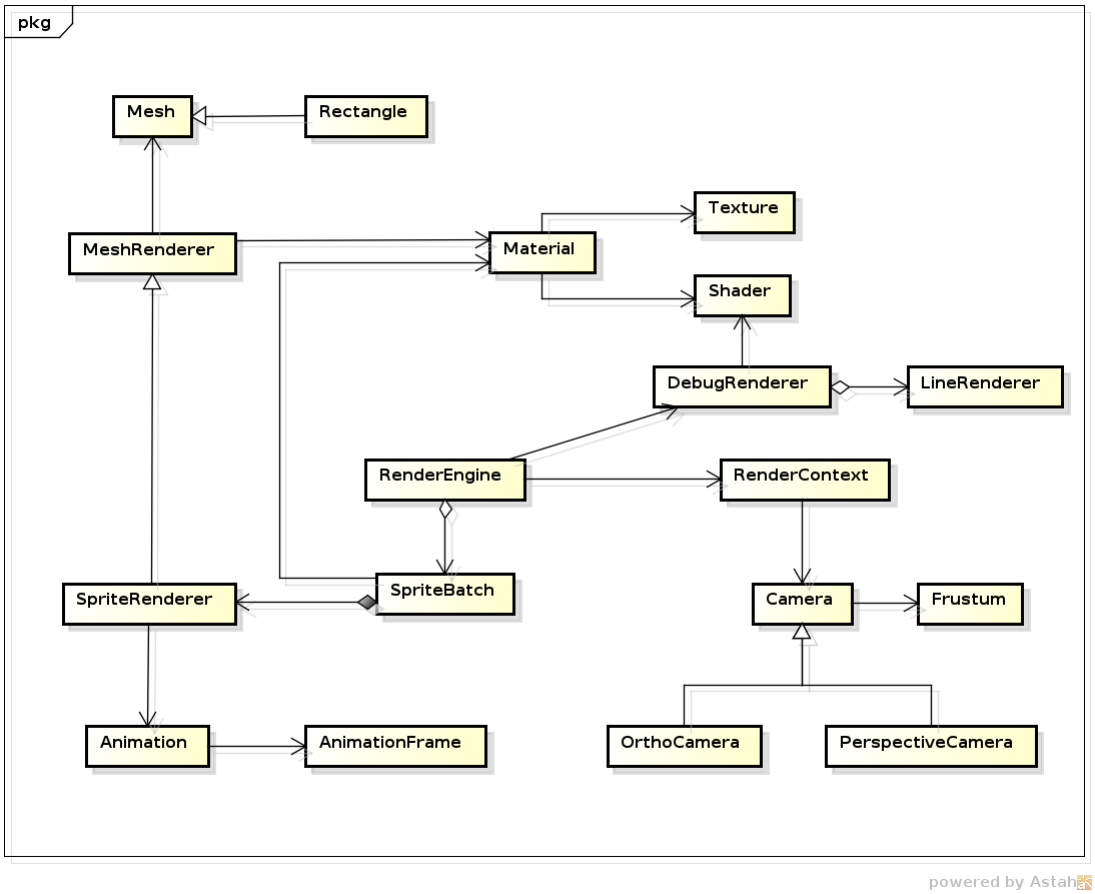

Figura 6.6: Módulo de Gráficos.

Este *GameEngine* ofrece la posibilidad de dibujar *sprites* (figuras rectangulares), texturizarlos (con un color o una textura o ambos combinados) y animarlos mediante *spritesheets*. Soporta también la creación y control de cámaras en proyección ortográfica y en perspectiva, así como soporte de *TextureAtlases* y materiales.

La clase `RenderEngine` es la que se va a encargar de gestionar, todos los componentes de la clase `SpriteRenderer`. Esta última clase, es el componente que proporciona a un `GameObject` todas las capacidades para poder dibujarse por pantalla en forma de ***sprite* rectangular**.

La clase `RenderContext` representa el contexto en el cual se dibujarán los *sprites*, la única función de esta clase es albergar la cámara. Existía la idea de que, en el caso de que este *GameEngine* adquiriese capacidades 3D, estea clase albergaría también las luces de la escena. Esta clase ofrece un nivel más de indireción, ya que así la cámara no se debe compartir directamente entre los objetos que se use, sino que se comparte el objeto `RenderContext` y de esta forma podemos cambiar dinámicamente el típo de cámara.

6.7.1 Batches o Lotes

Dentro de `RenderEngine`, los *sprites* se agrupan en **batches** o lotes. Los *sprites* se agrupan mediante un criterio, por ejemplo, en este caso, es que posean la misma textura, de modo que cuando se tenga que dibujar un *batch* de mil *sprites*, solo se active una vez, la textura, desactivándose una vez se hayan dibujado todos los *sprites*.

Los *batches* son objetos de la clase `SpriteBatch` que agrupan los *sprites* compartiendo la textura, pero no solo la textura. Estos *batches* están diseñados específicamente para agrupar *sprites*. Almacenan una sola copia de una malla rectangular `RectangleMesh`, ya que todos los *sprites* son cuadrados y comparten los mismos cuatro vértices, aunque cada *sprite* se dibuje en **posiciones diferentes**.

Los *batches* son una **optimización** vital a la hora de poder dibujar muchos objetos por pantalla. De otro modo, habría que activar cada textura de forma independiente, dibujar el *sprite* y luego volver a desactivar la textura, añadiendo una sobrecarga de cómputo debido a la constante comunicación entre la CPU y la GPU.

6.7.2 Capas de profundidad

Los *sprites* también se organizan por capas de profundidad, es decir los que se dibujen en la capa 0, serán los que se dibujen al fondo, los sprites de la última cápa, la capa n, se dibujarán encima de todos los demás.

Si no se usasen capas, los *sprites* se dibujarían, según el orden en el que han sido añadidos a la escena. De esta forma el bucle de renderizado queda de la siguiente forma.

Código 6.5: Bucle de renderizado

```
1 for (var i = 0; i <= this.numLayers; i++) { // PARA CADA CAPA
2
3     if(this.textureBatches !== null) // SI HAY BATCH DE SPRITES CON TEXTURA
4       for (var j in this.textureBatches) // PARA CADA BATCH
5         this.textureBatches[j].render(i); // DIBUJAR LOS QUE PERTENECEN A LA CAPA i
6
7     if(this.noTextureBatch !== null) // SI HAY SPRITES SIN TEXTURA (SOLO COLOR)
8       this.noTextureBatch.render(i); // DIBUJAR LOS QUE PERTENECEN A LA CAPA i
9 }
```

Esta organización por capas, puede **restar rendimiento** a la mejora de los *batches*, ya que los *batches* posiblemente se renderizarían por partes. El mejor caso, en cuanto a rendimiento, se obtendría cuando todos los objetos están en la misma capa, pues no se produciría esta partición de los *batches*.

Para paliar esta caida de rendimiento, los sprites se añaden a los batches ordenados por la capa a la que pertenecen, de modo que se rendericen todos juntos.

6.7.3 Texture Atlas

Para mejorar el rendimiento, el usuario puede agrupar sus texturas individuales en lo que se denomina **Texture Atlas**, que no es más que una combinación de todas las texturas en una sola imagen más grande. El usuario puede aplicar ese *Texture Atlas* al `GameObject` como si fuese una textura normal, y seleccionar que región de esa textura desea seleccionar, mediante coordenadas de textura.

Esto beneficia en que los *sprites*, se van a crear con el mismo *Texture Atlas*, por lo que se **agruparán bajo el mismo batch**, pero, a la hora de dibujarse, cada uno se dibujará con la región seleccionada, por tanto parecerá que cada *sprite* tiene su textura independiente.

Figura 6.7: Ejemplo *Texture Atlas* agrupando texturas.

6.7.4 Frustum culling

En la sección 5.3, explico qué es un **frustum**, un volumen, compuesto por planos, dentro de los cuales se renderiza todo lo visible. La clase `Camera` posee un objeto `Frustum`, que representa este volumen, cada frame, es actualizado si la cámara se ha movido, para regenerar ese volumen en la posición correcta. Si la cámara no se ha movido, el *frustum* no cambiará.

Jugando con el *frustum*, podemos aplicar una optimización bastante básica, **Frustum Culling**, que consiste en lo siguiente. Si podemos detectar qué *sprites* se encuentran dentro del frustum, entonces podemos hacer que, solo los *sprites* que se encuentren dentro, sean renderizados, ya que los que se encuentran fuera no serán vistos por nadie. La idea básica es, ***"no dibujes lo que nadie ve"***. Esto reduce la carga de la tarjeta gráfica, ya que solo tiene que renderizar un subconjunto del total de *sprites*.

6.7.5 Animaciones mediante SpriteSheets

La clase `SpriteRenderer` puede dibujar un *sprite* simple o animado, si se le proporciona la animación a reproducir. Las animaciones se definen mediante la clase `Animation` que, a su vez, está compuesta por objetos de la clase `AnimationFrame`, que representa cáda uno de los frames de la animación.

Una animación parte de una imagen, un *spritesheet*, el concepto es similar al de un *Texture Atlas*, todas las animaciones de un personaje se encuentran agrupadas en una imagen. En la figura 6.8, podemos ver, la primera fila, animación de andar hacia abajo, segunda línea, andar hacia la izquierda, tercera, andar hacia la derecha y cuarta, andar hacia arriba.

Cada fila representa una animación y se reproducen (en este caso) de izquierda a derecha, cada animación posee cuatro frames, por tanto necesitaremos cuatro `AnimationFrame` por animación.

Figura 6.8: *SpriteSheet* de un personaje.

La clase `AnimationFrame` posee información de qué región de textura se debe recortar para ese frame. Las animaciónes se actualizan cada frame, y se pueden reproducir a una velocidad, a mayor velocidad más rápido pasará al siguiente frame.

6.7.6 Comunicación con WebGL

Para que los *sprites* puedan dibujarse básicamente hay que llevar a cabo tres fases: **bind** (enlazar), **update** (actualizar) y **render** (dibujar).

Estas tres fases han sido implementadas como tres funciones dentro de la clase `SpriteBatch`, ya que es la que almacena los *sprites* y se encarga de dibujarlos de forma eficiente.

La primera fase, **bind**, crea una estructura propia de *WebGL* llamada *VAO* (ver anexo 8.1) con la infomación que compartirán todos los sprites. De forma general, lo que se hace en esta fase es **enviar a la memoria de la tarjeta gráfica** toda la información necesaria. En nuestro caso se enviarán los vértices de un rectángulo, el mismo rectángulo que comparten todos los *sprites*. También se envían las coordenadas de las texturas y los índices. Lo importante aquí es comprender que esta información solo se envía **una vez** a la memoria gráfica.

Una vez esta información ha sido enviada a la memoria gŕafica, comienza el bucle del juego, en el que se sucederán contínuamente las fases **update** y **render**.

En la fase **update**, se envían las matrices `ProjectionMatrix` y `ViewMatrix` a los *shaders* (ver anexo 8.1) de *WebGL*. Los *shaders* tampoco se duplican, todos los *sprites* usan el mismo *shader*, uno diseñado para dibujar *sprites* 2D con texturas estáticas, recortadas o animadas.

Concrétamente la clase `Material` es la que almacena el `Shader`, también almacena otros dátos relacionados con el material que adopta el *sprite*, como la textura y el color.

En la siguiente fase, **render**, se procederá con el **dibujado** o renderizado. El *batch* activará la textura y dibujará aquellos *sprites* que cumplan las siguientes condiciones:

- El *sprite* está activado. (Los componentes se pueden activar y desactivar, el usuario podría haberlo desactivado a voluntad por algún motivo).
- El *sprite* pertenece a la capa de profundidad que se pretende dibujar ahora.
- El *sprite* se encuentra dentro del volumen de dibujado (*frustum*).

Si el *sprite* pasa estas condiciones, entonces se cargan en el *shader* los datos propios del *sprite*, es decir los que no pueden ser compartidos por el resto de *sprites* y hacen a este único. En este caso son, el color que se quiera añadir a la textura (se puede optar por no añadir ninguno) y la posición, rotación y escala del *sprite*. Estos tres datos relacionados con la geometría del objeto (posición, rotación y escalado), se obtienen del componente `Transform`, y van combinados en forma de matriz, la denominada `ModelMatrix`. Esta matriz es la que se le envía al *shader*.

Una vez cargados estos datos, se procede a ejecutar el comando que inicia el proceso de dibujado en la tarjeta gráfica, acto seguido se visualizarán los *sprites* en pantalla. Luego se borrará la pantalla y se volverá a la fase *update*.

6.7.7 DebugRenderer: Dibujar líneas

Adicionalmente se ha implementado la clase `DebugRenderer` y `LineRenderer`. Estas dos clases están destinadas al renderizado de líneas mediante *WebGL*. Siendo *LineRenderer* una estructura que contiene toda la información necesaria para dibujar.

`DebugRenderer` almacena la lista de `LineRenderer` y se encarga de crear diréctamente los objetos `LineRenderer`, a partir de los parámetros de punto origen, punto destino, color de la línea.

6.8 Módulo de Físicas : Box2DWeb

La finalidad de este módulo no es otra que enmascarar la biblioteca *Box2DWeb* y ofrecer una interfaz más sencilla y acorde con el resto de módulos, para poder crear entidades físicas de forma más sencilla.

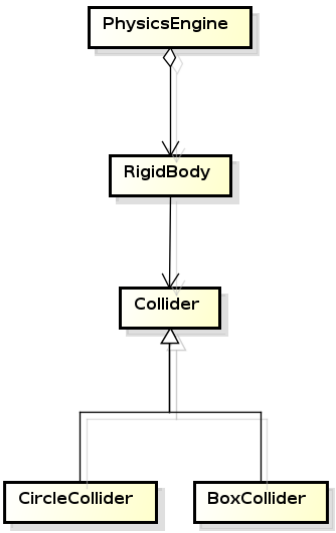

Figura 6.9: Módulo de Físicas.

La función principal de la clase `PhysicsEngine`, es la de ser una fachada sobre la clase `World` de *Box2DWeb*. Se encarga de gestionar los **componentes** `RigidBody` y `Collider`.

`PhysicsEngine` implementa por defecto características que siempre interesa tener cuando se trabaja con *Box2DWeb*, pero no vienen implementadas por defecto, como el **mapeo de las posiciones y ángulos** de rotación de los objetos físicos, a los objetos renderizables. Esto es algo vital, pero es algo de lo que se tiene que encargar el programador cuando trabaja *Box2DWeb al desnudo*.

También implementa un `ContactListener` (clase que ya viene en *Box2DWeb*) para detectar los **eventos de colisión** y una gestión correcta del **borrado** de los objetos.

La clase `RigidBody` enmascara el acceso a la clase `Body` de *Box2D*. Y la clase `Collider` gestiona la creación de fixtures. `CircleCollider` para crear círculos y `BoxCollider` para cajas.

Para añadir un componente `Collider` a un *Game Object* es obligatorio que este posea un componente `RigidBody`.

Obviamente todo esto simplifica y limita las capacidades del *Game Engine*, por eso, desde `PhysicsEngine` y `RigidBody`, se puede seguir accediendo a todas las capacidades de *Box2D*, mediante las funciones `getBox2DBody` y `getBox2DWordl` respectivamente, en caso de necesitar todas las capacidades de *Box2D*.

6.9 Módulo de Lógica : Scripts

Un `GameObject` que contenga solo componentes de renderizado o físicas representa un objeto bastante básico de un juego, pero si queremos que esos objetos interactuen ante eventos del juego, necesitamos algo más. Aquí es donde entra la clase `Script`, un *script*, es un componente que el programador puede crear e insertar en el `GameObject`, para que este, reaccione a eventos tales como una colisión, la creación y destrucción del objeto o una actualización del *game loop*.

`ScriptEngine` no es más que una clase que contiene una lista con todos los `componentes Script` del juego y cada iteración del *game loop*, se recorre la lista para actualizar los scripts.

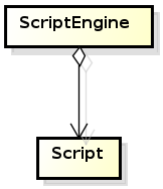

Figura 6.10: Módulo de Script.

`Script` es una **clase abstracta** de la que debe **heredar** el programador para crear sus propios scripts. Al heredar se deberán redefinir las siguientes funciones:

- **`start`**: esta función se ejecuta cuando el `GameObject` es creado en la escena.
- **`update`**: esta función se ejecuta cada frame del juego.
- **`onEnterCollision`**: esta función se ejecuta cuando el `GameObject` colisiona con otro objeto.
- **`onExitCollision`**: esta función se ejecuta cuando el `GameObject` deja de colisionar con algún otro objeto.
- **`onDestroy`**: esta función se ejecuta cuando el `GameObject` es destruido de la escena.

6.9.1 ContactListener y Scripts

El `ContactListener` implementado en el módulo de física detecta cuando un `GameObject` colisiona con otro objeto. Recordemos que un `GameObject` es un contenedor de componentes, cuando se detecta la colisión, se accede a los componentes del `GameObject` y se buscan los componentes `Script`. Para cada script, se ejecuta su función `onEnterCollision`. Lo mismo ocurriría cuando la colisión finaliza, pero se ejecutaría la función `onExitCollision`.

Las funciones `onEnterCollision` y `onExitCollision` reciben dos parámetros que el usuario puede consultar: `otherGameObject` y `contact`. El primer parámetro es el `GameObject` con el que nuestro `GameObject` está colisionando. El segundo parámetro es una estructura de *Box2D*, de la clase `Contact`. De esta estructura se puede extraer información como el punto de contacto, la normal de colisión, los dos fixtures involucrados en la colisión, la tangente de colisión, la penetración y el impulso que se aplicará.

6.10 Combinando los módulos: Engine

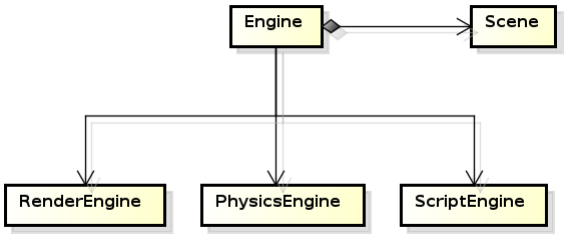

Figura 6.11: La clase Engine hace uso de `Scene`, `RenderEngine`, `PhysicsEngine` y `ScriptEngine`.

En la clase `Engine` es dónde se explica cómo se relacionan los tres subsistemas principales junto con la escena y sus objetos. Al principio, todos los `GameObject` serán creados junto con sus componentes (a gusto del usuario-programador) y añadidos a la `Scene`. Cuando el `GameEngine` inicia su bucle principal, la escena es **cargada**.

La carga de la escena implica recorrer todos los `GameObject` y extraer sus componentes. Cada componente será enviado al subsistema correspondiente.

Cuando una escena es **des-cargada**, los tres subsistemas limpian sus objetos y vuelven a un estado inicial. Hecho esto se vuelve a producir el proceso de carga, pero esta vez con la nueva escena.

6.11 Módulo de Entrada de usuario: Teclado y Ratón

La clase `Input`, hace de fachada entre el manejo de la entrada nativo de JavaScript y el programador, ofreciendo símplemente una serie de funciones útiles para el manejo de la entrada de teclado, como son: `isKeyPressed`, para saber si cierta tecla ha sido pulsada, `getButton` para saber qué botón del ratón ha sido pulsado y `getCursorPosition` que devuelve un `Vector2` con la posición del cursor en pantalla.

Dado que de forma nativa solo se puede comprobar si se ha pulsado una tecla a la vez, se ha implementado una lista, dónde las teclas pulsadas se van acumulando. De modo que cuando se consulta si una tecla se ha pulsado, se recorre esa lista. Esto permite indicar si una o más teclas han sido pulsadas, ya que si están en la lista, la respuesta será afirmativa. Cuando una tecla se libera (deja de pulsarse) es eliminada de la lista.

Si la tecla ya está en la lista no se volverá a insertar. Esto se debe a que, la detección de teclas tiene una frecuencia muy alta y la lista crecería de forma incontrolable. Así como mucho, la lista crecerá, un número igual a la cantidad de teclas que pueda pulsar el usuario a la vez, suponiéndose este número diez o menor.

7

Conclusiones y vías futuras

7.1 Conclusiones

Los requisitos propuestos para este proyecto fueron crear un *Game Engine* capaz de dibujar, texturizar y animar sprites 2D mediante spritesheets, la simulación de físicas 2D, la gestión de escenas, capacidad de crear interactividad o IA básica mediante scripts, la entrada de usuario mediante teclado y ratón, así como funcionalidades básicas de vectores y matrices, junto con las operáciones aritméticas necesarias para estas estructuras matemáticas.

Todos esos objetivos han sido cumplidos mediante la metodología seguida. Antes del desarrollo del código, se han analizado las dos tecnología externas usadas en este proyecto, WebGL y Box2DWeb. Tras esto, se crea una lista de tareas bien definidas, siguiendo una metodología Kanban y un diseño previo de las clases y módulos mediante diagramas de clases. Seguido de una implementación controlada mediante pruebas de integración.

En este proyecto he podído afianzar mis conocimientos sobre la biblioteca WebGL y en general, la computación gráfica, orientada a videojuegos. He comprendido la importancia de optimizar cada paso, pues, no sirve que simplemente podamos dibujar en pantalla, sino que necesitamos dibujar de la forma más eficiente posible, ya que los *Game Engine* van a ser herramientas usadas para dibujar muchos elementos en pantalla.

Por otra parte, no tenemos solo las funciones de dibujado, sino también tenemos que tener en cuenta la simulación física, tarea que también necesita recursos, más razón aún para realizar implementaciones eficientes. Para esta funcionalidad he trabajado y aprendido con la biblioteca Box2DWeb.

En cuanto al lenguaje, JavaScript, este proyecto me ha permitido familiarizarme mucho con él, ya que todo el proyecto está escrito en este lenguaje. Bien es cierto, que la verdadera potencia de JavaScript reside en bibliotecas orientadas a la programación web, como JQuery. En este proyecto no se ha necesitado nada de ello, de hecho el resultado no hubiera cambiado de haberse programado en otro lenguaje. La única diferencia es la plataforma en la que se ejecuta, escritorio o navegador.

Como ingeniero he podido nutrirme muchísimo ya que un proyecto tan grande ofrece la posibilidad de aplicar muchos conocimientos como matemáticas, computación gráfica, simulación física, así como técnicas específicas de cada área, arquitectura interna de los *Game Engine*, patrones de diseño, estructuras de datos, órdenes de magnitud, optimización. Y no solo conocimientos técnicos, sino que he podido mejorar mis habilidades en cuanto a gestión de proyectos se refiere, control de versiones, metodologías ágiles, diseño de software.

También otras habilidades que, aún pudiendo pertenecer al campo de la ingeniería, pertenecen realmente a uno ligeramente más general, el ámbito personal. Enfrentarme a un proyecto de este tamaño me ha proporcionado una experiencia que hubiera sido difícil de conseguir de otra manera. Este proyecto ha llevado mi capacidad de esfuerzo un paso más adelante, debido claramente a la motivación que este proyecto me infundía. Sin duda, he descubierto como ser un buen autodidacta, y creo que es la herramienta más valiosa que me llevo de este trabajo.

Esto me lleva a mencionar, que, aún habiendo sido una actividad principalmente de trabajo autónomo, no habría podido llevarla acabo, de no ser por la base teórica recibida a lo largo de la carrera, ya que realmente los temas tratados en este proyecto no se ven en la carrera, pero sin duda, he podido adentrarme en este mundo y defenderme, gracias a esa base aprendida en la carrera. También dentro de la intensificación de Computación, he podido enfrentarme a varios proyectos que también han contribuido a solidificar mi conocimiento sobre gráficos y videojuegos.

7.2 Vias Futuras

Como líneas de trabajo futuras, quedan contempladas las siguientes. La primera y más simple tal vez sería la de mejorar la biblioteca con funcionalidades extra que sigan facilitando el trabajo a los programadores. Como expansión de la fachada con más funciones útiles para el desarrollo de videojuegos.

La siguiente consistiría en expandir las capacidades de 2D a 3D, tanto de dibujado como de simulación física. Aunque esto puede conllevar una reestructuración completa del código, por no hablar de la mayor complejidad que la simulación física 3D tiene sobre la 2D.

Otra posible vía de trabajo sería construir un videojuego a partir de esta biblioteca, se podrían construir todo tipo de videojuegos 2D, ya que la biblioteca posee bastante grado de genericidad, pudiendo optar por plataformas, puzzles, mundo abierto.

Por último, la vía de trabajo que más interesante me parece sería crear un editor, mediante HMTL5, que permita diseñar videojuegos con la biblioteca aquí implementada, de forma visual, lo cual acelera mucho el trabajo, de hecho, considero vital poseer un editor visual, pero debido a que la envergadura del proyecto ya es demasiado grande, esta opción quedó descartada.

8

Anexos

8.1 Anexo 1: Trabajar con un pipeline programable

8.1.1 Shaders

Desde OpenGL 2.0, los shaders se han convertido en la herramienta que permite al programador personalizar ciertas fases del pipeline, en nuestro caso veremos las dos fases programables más importantes **vertex shader** y **fragment shader**.

> **Shader**
> - Un shader de OpenGL es un programa escrito en **GLSL**[47] (OpenGL Shading Language), que puede compilarse y ejecutarse en el hardware gráfico.

8.1.1.1 Vertex Array Objects y Vertex Buffer Objects

La primera fase del pipeline, **vertex shader**, recibe como entrada un array de vértices o, en terminología de OpenGL, un **Vertex Stream**.

Esos vértices están almacenados de una forma especial, en dos estructuras de datos, denominadas **VAO** (Vertex Array Object)[50] y **VBO** (Vertex Buffer Object)[51].

> **Vertex Buffer Object**
> - Un **VBO** es un buffer el cual es usado para enviar información de los vértices al pipeline.

Para ser enviados a la fase **vertex shader**, los vértices son encapsulados en un VBO. El VBO no es más que un array, en el que los componentes de los vértices se suceden de foma consecutiva de la forma: `[x0,y0,z0,x1,y1,z1,x2,y2,z2,...]`.

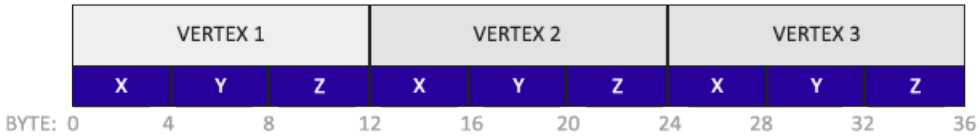

Figura 8.1: Esquema de un VBO.

Los VBO se usan para enviar otra información a parte de las posiciones de los vértices, también se usan para enviar las componentes de **color**, **coordenadas de texturas**, **normales**. Estos datos se explicarán más adelante en este documento. De momento nos quedaremos con que los VBO sirven para transmitir información al hardware gráfico en forma de *stream*.

Posición, **color**, **coordenadas de texturas**, **normales**, estos datos son conocidos como los **atributos** de los vértices. Podemos verlos como las propiedades que tiene un vértice.

Para cada atributo se deberá crear un VBO, tal y como se ha hecho para la posición en la imagen 8.1. A la hora de dibujar, deberemos activar estos VBO uno por uno (esto también se explicará más adelante). Para evitar tener que activar los VBO uno por uno, podemos agruparlos dentro de otra estructura superior, el VAO.

Vertex Array Object

- Un **VAO** es una **colección de VBOs**. Esto permite agrupar todos los atributos de un objeto bajo una misma estructura. Por ejemplo, las posiciones de los vértices de un objeto, sus colores, sus coordenadas de textura, sus normales. Es decir, hay una relación directa entre un objeto de nuestra escena y el VAO, por así decirlo, el VAO puede representar un objeto completo, con todas sus características.

En la figura 8.2 podemos ver un VAO que contiene múltiples VBOs. El VBO1 contiene todas las posiciones de los vértices mientras que el VBO2 contiene todos los colores de los vértices, los colores tienen cuatro componentes **r,g,b,a** por tanto el VBO2 se vería como sigue: `[r0,g0,b0,a0,r1,g1,b1,a1,r2,g2,b2,a2,...]`

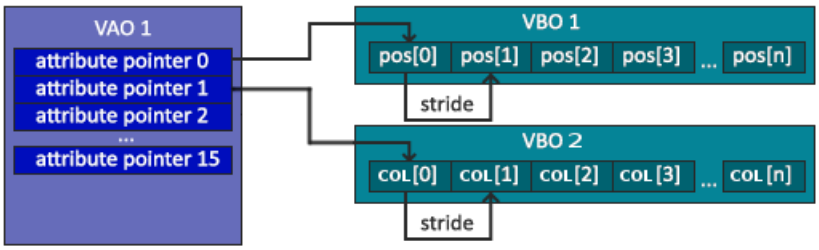

Figura 8.2: Esquema de un VAO.

8.1.1.2 Vertex Shader

El vertex shader es un programa escrito en GLSL que recibe la información de un vértice del VAO y devuelve ese vértice transformado. A continuación podemos ver un ejemplo de vertex shader:

Código 8.1: Vertex Shader

```
attribute vec4 position;
attribute vec4 color;
attribute vec2 texcoord;

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;

varying lowp vec4 vColor;
varying vec2 vTexcoord;

void main() {
  gl_Position = projectionMatrix*viewMatrix*modelMatrix*position;

  vTexcoord = texcoord;
  vColor = color;
}
\caption{Vertex Shader}
```

Analicemos el shader paso por paso. Lo primero con lo que nos encontramos son los atributos: **posición, color y coordenada de textura**. Estos son los atributos del vértice, anteriormente ya mencionados.

```
attribute vec4 position;
attribute vec4 color;
attribute vec2 texcoord;
```

Las variables **uniform** son variables que mantienen su valor para todos los vértices, es decir son compartidas por todos los vértices. Ejemplos de ello son las matrices *proyectionMatrix y viewMatrix*.

```
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 transformationMatrix;
```

Las variables **varying** son la forma de comunicar el vertex shader con el fragment shader, en este caso creamos dos variables varying, *vColor y vTexcoord*, estas variables serán asignadas con el color y la coordenada de textura respectivamente y serán enviadas al fragment shader.

```
varying lowp vec4 vColor;
varying vec2 vTexcoord;
```

Este es el cuerpo principal del shader.

```
void main() {
  gl_Position = projectionMatrix*viewMatrix*modelMatrix*position;

  vTexcoord = texcoord;
  vColor = color;
}
```

La variable **gl_Position** es una variable que viene **predefinida en GLSL**, es necesario asignarla para que OpenGL lea de ahí la posición final de nuestro vértice.

Aplicación de las matrices de transformación

» Este punto del shader es muy importante pues **es aquí donde se usan las matrices de transformación** para transformar la posición del vértice.

La múltiplicación de matrices se realiza en el shader y no en el código del programa, básicamente esa es la diferencia entre realizar la multiplicación de matrices con la **GPU** o con la **CPU**. Al realizar la multiplicación de matrices con la GPU nos aseguramos un mejor rendimiento.

```
gl_Position = projectionMatrix*viewMatrix*modelMatrix*position;
```

Por último enviamos el color y la coordenada de textura al fragment shader.

```
vTexcoord = texcoord;
vColor = color;
```

8.1.1.3 Fragment Shader

El fragment shader es un programa escrito en GLSL que trabaja con **fragmentos**, nótese que ya no son vértices. Un fragmento es un elemento que posee un color y una posición. Y se corresponde con un píxel en pantalla. O dicho de otra forma un fragmento es aquello que se pintará finalmente en la pantalla del usuario.

Código 8.2: Fragment Shader

```
varying vec2 vTexcoord;
varying lowp vec4 vColor;

uniform sampler2D uSampler;

void main() {
  vec4 texColor = texture2D(uSampler, vTexcoord);
  gl_FragColor = texColor + vColor;
}
```

Como mencionábamos antes, las variables varying reciben los datos enviados por el vertex shader.

```
varying vec2 vTexcoord;
varying lowp vec4 vColor;
```

Esta variable uniform, es por la cual, se pasa la información de la textura 2D al shader.

```
uniform sampler2D uSampler;
```

El cuerpo del shader. Aquí básicamente, se obtiene el color de la textura para ese fragmento y se almacena en **gl_FragColor**, otra variable predefinida de GLSL, de la cual, se leerá el color del fragment.

También se le suma *vColor*, GLSL nos permite hacer este tipo de operaciones, así, por ejemplo, combinamos el color de la textura con otro color.

```
1 void main() {
2   vec4 texColor = texture2D(uSampler, vTexcoord);
3   gl_FragColor = texColor + vColor;
4 }
```

Bibliografía

[1] What is a Game Engine? http://www.gamecareerguide.com/features/529/what_is_a_game_.php

[2] Middleware http://www.develop-online.net/analysis/the-rise-of-middleware-2-0/0115974

[3] Elemento HTML <canvas> https://html.spec.whatwg.org/#canvas

[4] Ejemplos <canvas> https://www.w3schools.com/html/html5_canvas.asp

[5] Different Numerical Integration techniques http://gafferongames.com/game-physics/integration-basics/

[6] Spatial Partitioning https://en.wikipedia.org/wiki/Space_partitioning

[7] Box2D http://www.box2d.org/

[8] Repositorio Box2DWeb https://github.com/hecht-software/box2dweb

[9] Ian Millington *Game Physics Engine Development*, Morgan Kaufmann, 2007.

[10] Ian Parberry *Introduction to Game Physics with Box2D*, CRC Press, 2013.

[11] Wordl - Box2D http://www.iforce2d.net/b2dtut/worlds

[12] Body - Box2D http://www.iforce2d.net/b2dtut/bodies

[13] Fixture - Box2D http://www.iforce2d.net/b2dtut/fixtures

[14] Joint - Box2D http://www.iforce2d.net/b2dtut/joints-overview

[15] Separating Axis Theorem (SAT) http://www.dyn4j.org/2010/01/sat/

[16] Gilbert–Johnson–Keerthi (GJK) http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?&arnumber=2083

[17] Expanding Polytope Algorithm (EPA) http://www.dyn4j.org/2010/05/epa-expanding-polytope-algorithm/

[18] Construct2 https://www.scirra.com/construct2

[19] ImpactJS http://impactjs.com/documentation/weltmeister

[20] Phaser https://phaser.io/

[21] Phaser Physics Documentation http://phaser.io/docs/2.4.4/Phaser.Physics.html

[22] Explaining phaser physics systems http://www.html5gamedevs.com/topic/4518-explaining-phaser-2s-multiple-physics-systems/

[23] GameMaker http://www.yoyogames.com/gamemaker/

[24] TurbulenZ http://biz.turbulenz.com

[25] MelonJS http://melonjs.org/

[26] Tiled www.mapeditor.org

[27] TexturePacker https://www.codeandweb.com/texturepacker

[28] Shoebox http://renderhjs.net/shoebox/

[29] PhysicEditor https://www.codeandweb.com/physicseditor

[30] BitmapFont Generator www.angelcode.com/products/bmfont/

[31] Cordova/PhoneGag https://cordova.apache.org

[32] CocoonJS https://www.ludei.com/cocoonjs/

[33] Ejecta https://github.com/phoboslab/Ejecta

[34] p5.js https://p5js.org/

[35] Processing Foundation https://processingfoundation.org/

[36] p5.play http://p5play.molleindustria.org/

[37] Game Engines History https://prezi.com/6h9-3usp4nb-/the-history-of-game-engines/

[38] Game Engines History 2 http://www.kinephanos.ca/2014/game-engines-and-game-history/

[39] ¿Qué es un Game Engine? https://prezi.com/dswuxovmsk_t/que-es-un-game-engine/

[40] Weisstein, Eric W. «Coordinate System». http://mathworld.wolfram.com/CoordinateSystem.html

[41] Coordinate Systems http://learnopengl.com/#!Getting-started/Coordinate-Systems

[42] Luke Benstead, Dave Astle, Kevin Hawkings *Beginning Opengl Game Programming*, Course Technology 2nd edition, 2009.

[43] David C., Lay (2007). Álgebra lineal y sus aplicaciones (3 edición). México: Pearson. pp. 159, 162. ISBN 9789702609063.

[44] García Alonso, Fernando Luis; Pérez Carrió, Antonio; Reyes Perales, José Antonio. Ampliación de fundamentos de matemática aplicada. España: Club Universitario. p. 110. ISBN 9788484549772.

[45] Santaló, Luis A. Geometría Proyectiva (3ª edición). Buenos Aires, Argentina: Eudeba. pp. 88-92.

[46] Pipeline, OpenGL Basic Theory https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

[47] GLSL https://www.opengl.org/documentation/glsl/

[48] Vertex Shader https://www.opengl.org/wiki/Vertex_Shader

[49] Fragment Shader https://www.opengl.org/wiki/Fragment_Shader

[50] Vertex Array Object https://www.opengl.org/wiki/Vertex_Specification#Vertex_Array_Object

[51] Vertex Buffer Object https://www.opengl.org/wiki/Vertex_Specification#Vertex_Buffer_Object

[52] Component-Entity-System https://www.gamedev.net/resources/_/technical/game-programming/understanding-component-entity-systems-r3013