UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

HTML5 2D videogame programming

(Desarrollo de videojuegos 2D en HTML5)

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Álvaro Martínez de Miguel

Tutor: Ramón Pascual Mollá Vayá

Curso Académico 2015/2016

Dedication

To my beloved doctor, who makes my life funnier and happier than the game I make.

Acknowledgments

I want to thank my mother, my family, my friends and my coworkers at Hooptap.

HTML5 Videogame Programming

How to make a 2D videogame using HTML5 technologies

Alvaro Martinez de Miguel

Abstract

In these days where mobile and web gaming are on the rise, it's important to use a powerful and versatile environment like HTML5. This final project will cover the full development of a game wrapper library that allows anybody to see how JS game engines work on the inside.

The wrapper GW will contain interfaces for all core functionality an engine requires, it will be easy to extend and allow people to make their own games with it.

Resumen

En una época en la que los juegos web y para móviles están en alza, es importante usar un entorno potente y versátil como HTML5. Este tfg cubrirá el desarrollo completo de una libreria que actuara wrapper, permitiendo a cualquiera ver como los motores de juegos web funcionan por dentro.
El wrapper GW tendra interfaces para todas las funcionalidades basicas de un motor de juegos. Sera facil de extender y permitira a la gente hacer sus propios juegos con el.

Alvaro Martinez de Miguel
demipel8@gmail.com

Acronyms and initialisms

- **JS**: JavasScript
- **MVP**: Minimum Viable Product
- **ES5**: ECMAScript 5 and 5.1 edition
- **ES6**: ECMAScript 6 or 2015 or Harmony
- **API**: Application programming interface
- **SoC**: Separation of Concerns
- **MBB**: Minimum Bounding Box
- **CDN**: Content Delivery Network
- **GW**: Game Wrapper
- **CLI**: Command-line interface

Contents

Chapter 1

Introduction

1.1 Motivation

After working for more than 2 years with JavaScript and more specifically on JavaScript game development, I've gotten very fond of it. With a little study of this language you can find that a once thought of a toy language for DOM events and sticking AJAX petitions, is a very powerful tool that's present on every single browser in the world. Over the last years game developers have realized the power of these language and build a set of powerful tools for building from a small quiz with a question and a few buttons to a whole 3D MMO game.

Unfortunately not everything is joy. Due to the different browsers (and it's versions), the OS and the multiple devices, programming can turn to be a nightmare of hot-fixes to make sure your code will work as you intended everywhere.

Thus is my wish to collect all the experiences I've had making mini-games for my company and blend them into a wrapper that allows to learn how a section or an entire JS game engine works, and at the same time, to be able to make a game with it.

1.2 Objectives

The objective behind this final project is to offer a wrapper API that collects all the basic needs for an HTML5 game to be implemented. Leave the API opened so it can be extended or, make use of different implementation so students can try the wide variety of libraries available on the JS environment.

Make the Wrapper from a requirements specification, which will be extracted from studying game engines that produce web games.

Show the process and tooling of making the wrapper. Make extensive testing and documentation so it's solid and easy to extend. Upload it to the most used JS packet manager and publish a landing page.

Maintain all the project open source during and after its development.

1.3 Project structure

On the state of the art chapter, the JS environment will be analyzed from a game development perspective. After a brief introduction to the ecosystem, a SWOT analysis will explain the benefits and disadvantages of making web games.

The following chapter will discuss the most used features on game frameworks, the wrapper general architecture and the specification of the API .

In the implementation chapter, a section about the development environment and extension mechanisms of the wrapper. Deployment into production through a static objects pipeline is also discussed as it is very important for web technologies.

Chapter 2

State of the art

2.1 Review

The game industry is on the rise, producing games with budgets that rival some of the most expensive films. Teams of tens or hundreds are spending years to create astonishing games. The sector has been enough time to see its age spectrum wide enough to make game of all sorts of themes and all type of devices.

But also the indie game industry is expanding, development engines such as *Unity* or *Unreal* are making free licensing plans. Mobile games allow small teams to bring fun simple games, with original mechanics and not so focused on costly visual effects. Along this current of small indie development a new technology is gaining a lot momentum: **web**.

Web is present everywhere and with the rise of JavaScript server side a lot of developers are shifting from other languages, building a very powerful ecosystem. Included in this shift are game developers. Getting away from flash due to banning in mobile devices and the increasing number of browsers that are restricting its functionality, added to the fact that APIs for the same areas where flash is required are being deployed natively to HTML5, make sense for the change.

Setting a development environment for games in *HTML5* is pretty easy with the right tools and, projects like *CooconJS* and *NW.js* (previously known as node-webkit), allow to package our game into native mobile apps or desktop programs for every platform with little to non effort.

The main technology used for this project is HTLM5 and JavaScript.

2.2 HTML5

It's the fifth version of HTML (Hyper Text Markup Language) presented by the World Wide Web Consortium (W3C). It unifies previous versions of the standard such as *HTML 4*, *XHTML 1* and *DOM Level 2 HTML*. For the sake of complex applications it offers a new set of powerful APIs.

Some of these APIs give games applications a more flexible and stable base:

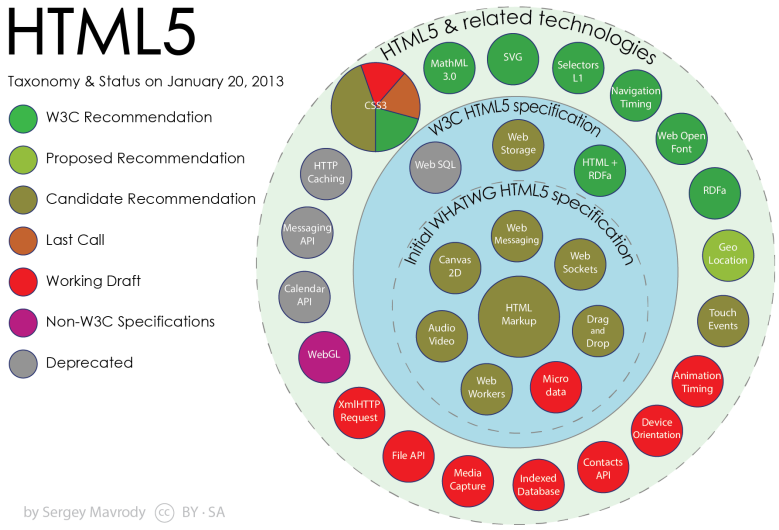


Figure 2.1: HTML5 new APIs

Canvas element: Allows for dynamic, scriptable rendering of 2D shapes and bitmap images. It also allow the use of WebGL (allowing the gpu to take load from the cpu).

Web Audio API: A high-level API for controlling and synthesizing audio on the web.

Web storage: Adds persistent data storage to the client browser.

2.3 JavaScript

2.3.1 History

JavaScript was originally developed by *Brendan Eich*, while he was working for *Netscape Communications Corporation* browser *Netscape navigator*. Netscape wanted a lightweight interpreted language that would complement Java by appealing to nonprofessional programmers, and Eich was commissioned to create it.

The first version was completed in ten days in order to accommodate the Navigator 2.0 Beta release schedule,and was called Mocha, which was later renamed LiveScript in September 1995 and later JavaScript in the same month.

In November 1996, Netscape announced that it had submitted JavaScript to Ecma International for consideration as an industry standard, and subsequent work resulted in the standardized version named ECMAScript. In June 1997, Ecma International published the first edition of the ECMA-262 specification. In June 1998, some modifications were made to adapt it to the ISO/IEC-16262 standard, and the second edition was released. The third edition of ECMA-262 was published on December 1999. Development of the fourth version was never completed and 2009 brought the 5 edition.

After realizing the long process that took to release a new version of ECMAScript and the fast rhythm of the industry, 2015 brought the partial implementation of ES6 or ES2015

edition and it has been proposed to change the release cycle to a year. Being ES2016 the first of the new series of the standard.

2.3.2 Characteristics of the language

- **C like syntax**
- **OOP but not class oriented**: An object-oriented program is a collection of individual objects that perform different functions. These objects are usually related in a hierarchical manner, in which new objects and subclasses of objects inherit the properties and methods of the objects above them in the hierarchy. JavaScript does not allow class instancing as it implements Prototype-based object-oriented programming. in this approximation behaviour reuse is performed via a process of cloning existing objects that serve as prototypes.
- **No variable typing**: JavaScript has typed values, not typed variables. The following built-in types are available:
  - string
  - number
  - boolean
  - null
  - undefined
  - object
  - symbol (new to ES6)
- **Scripting language** with JIT compilation
- **Functional programming**: It has arrays instead of lists and objects instead of property lists. Functions are first class. It has closures. You get lambdas through anonymous functions...
- **Event Driven**: A mechanism that handles executing multiple chunks of a program over time, at each moment invoking the JS engine, called the event loop. This introduces Asynchronous programming as an ordered queue of snippets of code to run, without JavaScript having a real sense of time.
- **Dynamic binding**: Object references checked at run time. This allow the same function to work over different contexts, helping the reuse of code.

2.3.3 Performance

For a long time javascript was treated as second class language for two main reasons:

1. Slow: The main idea was that JS taked 10 times to run a program than the same program written in C. Even though JavaScript is a complied language, it is compiled by the engine before execution trying to find possible better optimization, but it was complicated with characteristics such as the non typed nature of it's variables and the automatic memory management through the garbage collector. ASM.js proposal works on this areas, restraining javascript to a subset of the language that allows the engine for the best optimization possible.
2. Single threaded: JavaScript runs in only one single thread. The event driven language delivers incredible speed when it comes to I/O operations but spreading the event loop to different threads, for CPU demanding tasks, and maintaining coherence the whole time introduces more complexity. Some advances have been made in this area with the introduction of web workers.

A web worker runs independently of other scripts, in another thread. This way it wont affect the performance of the page. User interaction in the main thread can run uninterrupted, while the web worker runs in the background.

JavaScript engines

When a game is being developed in an environment such as javascript, it is impossible not to feel that some things escape your control. Even though you'll be able to launch your game on multiple OS systems, through different web browsers or even different types of devices, you have to take on account that each one of them will have a particular javascript engine. This engine will support the standard APIs up to some exchange, but still if two engines would support the exact same APIs, the underlying implementation would be diverse, and its areas of optimization won't be the same.

So we'll look at some points of interest:

Renderer

WebGL is the newest technology for web rendering an it offers the developers the possibility to send to the GPU repetitive work saving the CPU a lot of time, and has broaden the use of 3D graphics for the web. In a desktop environment it would be the preferred choice without a doubt.

But, mobile is a totally different world. IOS has a very optimized canvas component and has been known to give problems with WebGL. As for android, forget about it. All pre-Lollipop Android versions will give problems with WebGL. So canvas comes as the reasonable option for this type of environment. Also it has to be mentioned that Microsoft didn't supported WebGL until IE11, which can be a huge problem depending on the country you're planning to get your main audience from.

Optimization

As Kyle Simpson says on the fifth book of his You Don't Know JS saga, Async & Performance: *Many common performance tests unfortunately obsess about irrelevant microperformance details like x++ versus ++x. Writing good tests means understanding how to focus on big picture concerns, like optimizing on the critical path, and avoiding falling into traps like different JS engines implementation details.*

This means we will focus on the maintainability and readability of the code except on critical parts such as update and render methods, on which we will code the most efficient way possible, explaining it with comments if needed.

2.3.4 Data protection analysis

Cheating

There's a big problem for data consistency and avoiding data manipulation by evil users with technologies like JavaScript. The game runs in the client-side, the code will run on the users own device, so it's sensitive to manipulation. If the game your trying to do is multiplayer, you'll need to check server side if the actions one user is making are possible in the players context, that takes a lot of server computing, but helps to avoid cheaters.

The most common is this cases is a server that has an *authoritative state* of the game. That means the server receives all players inputs and calculates what the world should look like in order to prevent players from cheating.

Privacy

Private projects fear their code been stolen as it will be as simple as opening the browser console or scraping the game page. For this, and for optimization purposes, minimization and obfuscation are highly recommended, the code will weight less and be harder to read and modify.

Every task manager like *Gulp* or *Grunt* or a bundler like *webpack* will offer functionality to take this measures.

2.4 Justification

Lots of game engines are available to make games in HTML5 or have ways to export to JavaScript. These programs are powerful, complete and extensible pieces of software tested and ready for developing a professional game. So if there are so many engines. Why this final project?

2.4.1 Learning Purposes

It's true there are very good game frameworks but they are oriented to making games, not lo let learn people what it is required to make those same games. Some of them have private proprietary code bases, or are really bad documented inside, or can simply be overwhelming for someone who has just entered this world and wants to know how a simple game is made.

Right know, at the UPV, there are subjects and learning material oriented to game development, but none specific to the HTML5 field. This final project pretends to fill that gap.

2.4.2 Modularity

In object-oriented programming there is a say: *You wanted a banana but what you got was a gorilla holding the banana and the entire jungle*. Meaning that even though you can only want a small functionality, you have to carry around the whole environment. It can also be applied to this situation, to test a physics system you must know the whole objects system, the update loop, the renderers, ... Instead with a wrapper that has a defined interface you know what you have to implement to get it working without needing to know other subsystems.

2.4.3 SWOT Analysis

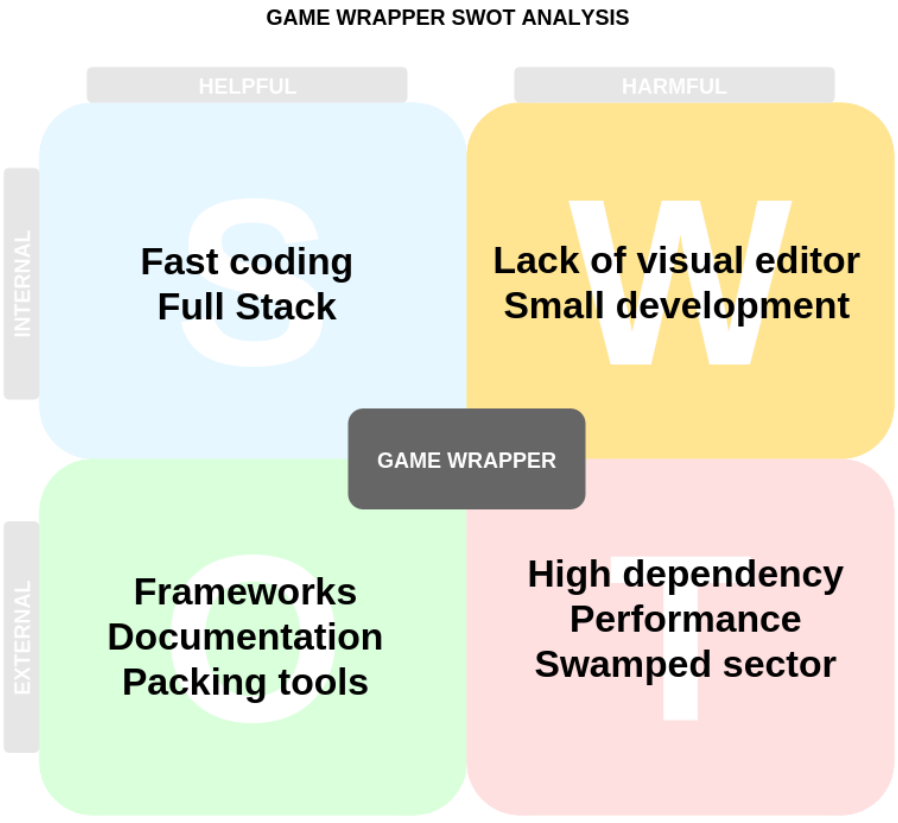


Figure 2.2: SWOT Analysis

Strengths and weaknesses

Strengths

Fast Coding: JavaScript is a very easy language to start with, lack of typed variables, dynamic memory manage via garbage collector, object management, etc. With the right understatement of its coding patterns and asynchronous programming a game can be prototyped fast and shipped to most types of hosting services or other Paas/Iaas [1] around the internet.

Full Stack: The whole game programmed and manage in the same language is a very big advantage and can speed up development. With a versatile team people can be switched between front-end, back-end, db or network teams. Models can be shared between the client program and the server side. Data can stay in json format through the whole flow, from the player input event to the data base and return.

Weaknesses

Lack of visual editor: Scene and animation editors are very handy in game development. The fact that we don't have it integrated with our environment is a major drawback, Some paid frameworks or IDEs already offer these tools for HTML5 development and more are getting into it, but, it's not the norm just jet.

Small development: Even width the best intentions, this wrapper is going to be developed, at least for a while, by one person. The development speed cannot compete with those projects that have tens or hundreds of contributors.

Opportunities and Threats

Opportunities

Frameworks: There are lots of options to choose, implemented with very diverse patterns and for multiple fields. With different renders, extension via plugins and very active in the open source community, getting lots of bug testing. The fact that professional studios use them on a daily bases can give an image of the state of maturity this young sector is getting.

Documentation: As commented on the opportunity above, some frameworks are very active in the open source community, having some members focused on the creation lots of learning material and quality docs. Although some times under valuated, this material can help smoothing the learning curve.

Packing tools: The *One development every platform* pattern is a very sought one. With the use of a couple packing tools it is possible for HTML5 to apply this technique with small changes.

Threats

High dependency: Even though the quantity of tools for building a good development environment is extraordinary it can backfire us. Editor, linter, code style checker, dependencies tool, template creator, task automation, local server,... Can make some errors hard to find and debugged if not mounted adequately.

[1] Paas: platform as a service. Iaas: Infrastructure as a service

Performance: Resources have to be transmitted through the internet, JavaScript compiled and processed at the client. This process hinders its performance when compared with other programming languages. But its getting closer, larger bandwidth, engines optimization and new technologies are making JavaScript up to a level that programs made in other languages are compiled into JavaScript to be run in the browser. A gaming example, in 2013 epic game's Unreal Engine 3 was fully ported to JavaScript with the use of asm.js, and only a year later the technology grew so much the Unreal Engine 4 was also ported. [2]

Swamped sector: HTML5 games are new to a lot of people, but some companies and programmers have been doing their job for some time. There are tens of frameworks to produce HTML5 games and some of them are very competitive and easy to use.

Going over all the analysis, the pros outdo the cons. The utility of the technology and the project has been established.

2.4.4 Proposal

Making JS games is a good option, how do we proceed to create our game wrapper?

First of all, get the most used features in JS gaming frameworks from a small study of some of the most used engines. Design a general architecture for the wrapper having in mind a modular design, which at first will be contained by the core features chosen before. Set an interface for each of the modules and implement it appropriately using a known library in the area.

How you build an app and the technologies or methods you decide to implement are as important as what you are building. We will make sure that our development process is smooth by using the adequate services and see what do we need to give the user the library in a way thatis best form him and the end user, the player.

Once a first version is build, we have to look for the easiest and most solid way of providing an extend procedure to create new modules or override existing ones. Create an automated task to develop modules, generate the interfaces documentation in order to be able to get a concise specification. Also create a testing set for each module so modifications are quickly validated.

Finally, publish GW to the most common package repository in JS and make a list of the task needed to do in the future to get to a more stable version.

[2] https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/

Chapter 3

Problem analysis

3.1 Project needs

At the introduction it was established through the objectives section that the API would be made based on a requirements specification, but before focusing on the web and game side. What does a software project of this type needs to have?

Coupling All the components forming this project must be able to change, so other people can test a specific library without having to compromise with a whole gaming framework. Loose coupling means that each of the system components has, or makes use of, little or no knowledge of the definitions of other separate components. This has to be met by always programming to an interface. The only immutable parts of the API will be its basic modules interfaces.

Open to changes This is going to be an open source project intended to be not just used by the user but modified by him so it has to have a maintainable code base. This will be achieved by setting a naming convention for all modules, methods and variables in the project and the use of documentation.

Robustness The change of a component of the game cannot make the other pieces fall even through an error. Robustness in software is defined as *the ability of a computer system to cope with errors during execution* so error handling and testing are crucial to the development of the library.

3.2 What it's needed to make a web game

To make a game a certain set of functionality is required in a framework. During this section a review of the basic needs is done.

3.2.1 Assets

Every game needs different resources, from a click adventure game that has all of its texts saved on a file, to top tier games that have models, filters, audio, meshes, sprites, animations...

The most basic ones are: *images*, *audio* and *data (JSON objects in JS case)*. The game is loaded through the internet so the assets need a component that takes care of caching the data into the client browser and binding a logical name inside the game with the actual resource, the **loader**.

Also a standard interface must define the assets properties so any section of the game knows how to manipulate an asset, lets call it the **resource** interface.

3.2.2 Visualization

For the user to interact and enjoy a game, he must see the graphics that the game is intended to show. The **render** takes care of the drawing tasks that are required. In web technologies there are 3 ways of rendering games, from the worst performance to the best: DOM, Canvas and WebGL.

3.2.3 DOM

The Document Object Model (DOM) is a programming interface for HTML, XML and SVG documents. It provides a structured representation of the document (a tree) and it defines a way that the structure can be accessed from programs so that they can change the document structure, style and content.

Basically is a technique that creates an html element for each rendered object in the game and it is rendered like any normal web would be. DOM rendering tends to slow on performance when a lot of objects are instantiated as it has to maintain an html element for each.

3.2.4 Canvas

HTML5 Canvas is an immediate mode bitmaped area of the screen that can be manipulated with JavaScript. Immediate mode refers to the way the canvas renders pixels on the screen. HTML5 Canvas completely redraws the bitmapped screen on every frame by using Canvas API calls from JavaScript.

3.2.5 WebGL

WebGL (Web Graphics Library) is a JavaScript API for rendering interactive 3D computer graphics and 2D graphics within any compatible web browser without the use of plugins. Based on OpenGL ES 2.0 standard it runs in the majority of desktop and mobile.

3.2.6 Update

If a game stays the same all the time, it's not a videogame, maybe even not a game. The game has to advance and change. This task is delegated to a **Game Loop**, which iterates all the time through certain aspect of the game like processing user input, updating object states and redrawing the screen. A game loop also helps to decouple I/O of the game from it's own advancement, making able to continue game execution while waiting for user input.

3.2.7 User Input

A game is intended to be interactive, the user must be able to communicate and manipulate the game. In a day with so many type of devices user input has to be supported in multiple ways: keyboard, mouse, touch and controller.

3.2.8 Physics

Lots of games try to imitate reality in a more or less realistic way. Physics libraries help to simulate forces on the objects of a game. This is a must in any ambitious game engine.

3.2.9 Audio

A game audio is more than a song. It need ambient sound, sound effects when events happen. Some games change the music depending on the situation of the player or even use sound to indicate the user of things that are about to take place. That's why an **audio manager** is recommended.

3.2.10 Tweening

Short for in-betweening, the process of generating intermediate frames between two images to give the appearance that the first image evolves smoothly into the second image. It's an easy way to move objects in simple ways without the use of a physic library which can over-complicate this task.

3.2.11 Spritesheets and Animations

A sprite, or image, is a very basic object of a game. A living object of the game often needs not just one image but a set of them to make one or more animations. Spritesheets are a good way to organize these animations, as it also help to save memory and work for the GPU.

3.3 Comparing Frameworks

To contrast the information posted before lets check the most used HTML5 Game frameworks:

3.3.1 Non free frameworks

Construct 2: The most popular framework according to HTML5gameengines.com. It has its own IDE, graphic level editor and it is extensible via plugin, also provides exportation to multiple platforms. It remembers to those types of frameworks that people who doesn't know how to program can use to make a videogame. Its prices go from 100 euros (personal license) to 330 euros (Business license) and a free license for education or other non profit projects.

Impact: Some years ago it seemed like a very popular solution with its level editor and cross-platform optimization but the fact that it is non free and that it hasn't been updated for almost a year have made the presence of this framework in the community almost inexistent.

3.3.2 Free frameworks

The are multiple ways to catalog game development frameworks, we will use the rendering technology, as graphics seem like the most basic and performance issue in our games.

Even thought some people might think that the're are types of games, like old text-based games, that don't need to worry about graphics you still need to render the text, and in some technologies is not as cheap, performance speaking, as people think.

DOM

CraftyJS: This is the framework I use at work. It's the only one I've come across that allows DOM rendering (It also works with canvas, and recently WebGl). It uses the ECS (Entity Component System) model to program games, which is a very comfortable and extensible pattern. With this pattern every instanced object you have in your game will be an entity. Entities can be formed mixing different components together. Offering a way to reuse and structure your code.

With a very simple API it gives you almost all the functionality needed to develop a 2D game. Particles are only allowed in canvas rendering mode and Physics are not implemented, only a Gravity component it's available which is not enough for most circumstances. As of this date (December 2015) my company has released an open plugin for MatterJS, a 2D physics engine written from a scratch in JavaScript. You can also use a couple ports of Box2D but their repositories have been abandoned for 3 years.

The biggest downside of this framework, aside from not having WebGl rendering is the slow update rate it has. Since July 2014 to June 2015 it has gone from 0.6.1 to 0.6.3. and its community, even though active, is pretty small compared to other frameworks.

Canvas

stage.js: It gives you a DOM-like tree data model to make your game. It abstracts all canvas dirty work (game loop and rendering). It also has physics via different plugins (MaterJS, p2.js, PhysicsJS, sat.js). It's a very simple framework to use, ideal for quick, simple games.

EaselJS: It's part of a suite library for HTML5 technologies known as CreateJS. Inside the library EaselJS take the role of simplifying canvas work. although it's not game specific solution, the fact that you can use the other libraries of the suite (all related to game programming: tweening, sound and preloading resources), and the flash-like interface. Attract some developers that come from other environments.

WebGL

Kiwi.js: It's a WebGL rendering based framework that uses the ECS model. It relies on a tight but still small community that make multiple plugins and game blueprints, which are whole games on github featuring a certain genre.

Phaser: It's game engine based on PixiJS renderer which gives it WebGL support with canvas fallback. It has probably the biggest HTML5 game developing community as it's founder also founded the html5gamedevs forum. It offers a lot of examples, 3 different Physics libraries and some awesome plugins, paid and free. Apart from tutorials, Phaser developers release minibooks explaining the insides of the framework trying to get people to know more the tool and make the most of it.

One of Phaser community more interesting projects is Mighty editor. An online game editor that lets you code , edit your scene graphically and preview your game.

Pandajs: Another framework based on PixiJS renderer. Less known and smaller than Phaser, also includes a game editor called Bamboo. Its advantage is that it's more lightweight than Phaser, who until recent versions has a monolithic structure.

Babylon: It's a WebGL based 3D games framework. It has an interesting performance on systems with good hardware. Also Microsoft has made some demos to promote their new browser, Microsoft Edge, and they show great potential.

One appealing feature for starters is that it allows you to play with the framework on a playground domain, being able to download the code once you've finished.

3.3.3 Other

Some other frameworks may not be specifically made for HTML5 programming, but they produce games for the web.

Cocos 2Djs: This framework is the HTML5 version of the C++ ultra famous Cocos-2D. It allows you to program in JavaScript and distribute to all kinds of platforms, web and native. This is achieved through important pieces: Cocos2d-html5 and Cocos2d-x JSB.

Cocos2d-html5 is a full HTML5 game engine, rendering on WebGL and Canvas. This allows games to run in browsers.

Cocos2d-x JSB serves as a middle layer between JavaScript and C++ Cocos2D-x code. With it you can communicate between both, allowing to target native platforms from JS code.

Haxe: An open source toolkit used to build cross-platform tools. It includes a programming language, a cross-compiler and a library. With this tools and the set of framework and libraries for gaming development built around haxe you can program once and deploy to multiple platforms. Including web.

Atomic: A new game engine in a very early stage that allows to program game with JS as well as C++. It has a very promising editor and its 2D module is free to use. It also deploys to multiple platforms.

3.3.4 The big guys

With the growth of HTML5 as a game programming language every one wants to be on board. This is the case of the next two engines. With a lot of experience on their backs, their working hard to deliver their product on WebGL based renderers.

Unity 3D: The favorite engine of indie studios of the last years. It's easy to learn and pretty powerful. A scene polished graphic scene, animation and audio editor with support for C# programming and a modified version of JavaScript.

With the popularity of Unity a really big community has formed around it, releasing lots of tutorials and learning material. A very good documentation and a spectacular asset store round up this product. Recently changing its license term have made viable for anyone to use all the power of this engine.

Unreal Engine 4: One of the most famous. With its origin in the all-around the world known *Unreal Tournament* game from epic games studios, used to make from casual to 3A games. The last version of the engine uses C++ for its programming language and it's experimenting with WebGL rendering. It has a bigger learning curve than Unity but in exchange you get the power of a much deeper control of the system, as it allows developers to go down to assembly language if required.

3.3.5 Comparative

After presenting all these engines, tables 3.1 and 3.2 check the use of the functionality presented on this chapter on them.

3.3.6 Conclusion

As it can be observed most of the studied frameworks have all the characteristics we were looking for, it's normal as they were the most basic ones. To get the fundamental pieces, the feature that will be implemented for the game wraper are the ones present in 10 out of the 14 studied framework.

The selected are Sprite, Loader, WebGL render, Game Loop, Physics, Audio, Tween and user Input. In the next section we will discuss the library architecture.

Afterwards we'll focus on each of the selected features one by one and try to resume a basic interface for each.

Table 3.1: Framework Features 1

	Assets	Loader	DOM	Canvas	WebGL	Game Loop
Construct 2	yes	yes		yes	yes	yes
Impact	yes	yes		yes		yes
CraftyJS	yes	yes	yes	yes	yes	yes
stage.js	yes	yes		yes		yes
EaselJS	yes	yes		yes		yes
Kiwi.js	yes	yes		yes	yes	yes
Phaser	yes	yes		yes	yes	yes
Pandajs	yes	yes		yes	yes	yes
Babylon	yes	yes			yes	yes
Cocos 2Djs	yes	yes			yes	yes
Haxe	yes	yes		yes	yes	yes
Atomic	yes	yes			yes	yes
Unity 3D	yes	yes			yes	yes
Unreal Engine 4	yes	yes			yes	yes
Total	14	14	1	9	11	14

Table 3.2: Framework Features 2

	Physics	Audio	Tween	User Input
Construct 2	yes	yes	yes	yes
Impact	yes	yes	yes	yes
CraftyJS	yes	yes	yes	yes
stage.js	yes	yes		yes
EaselJS	yes	yes	yes	yes
Kiwi.js	yes	yes	yes	yes
Phaser	yes	yes	yes	yes
Pandajs	yes	yes	yes	yes
Babylon	yes	yes	yes	
Cocos 2Djs	yes	yes	yes	yes
Haxe	yes	yes	yes	yes
Atomic	yes	yes	yes	yes
Unity 3D	yes	yes	yes	yes
Unreal Engine 4	yes	yes	yes	yes
Total	11	13	13	14

3.4 Architecture

We have an idea of the parts needed to make the wrapper but, how do we blend them together?

3.4.1 Modules

It has been commented before that the project has to be loosely coupled as it needs to be easy to modify. for the purpose we will apply the design principle *Separation of concerns* or SoC.

Separation of concerns

The SoC principal establishes that system elements should have exclusivity and singularity of purpose. No element should share in the responsibilities of another or encompass unrelated responsibilities. Separation of concerns is achieved by the establishment of boundaries.

A boundary is any logical or physical constraint which delineates a given set of responsibilities. This will increase maintainability and a more comprehensive codebase. Along the different types of separation we will apply vertical, horizontal and delegation of concerns.

Vertical separation

Each feature is going to be a standalone module which is able to function independently from the other, it will be a black box that will be accessed by others through the interface.

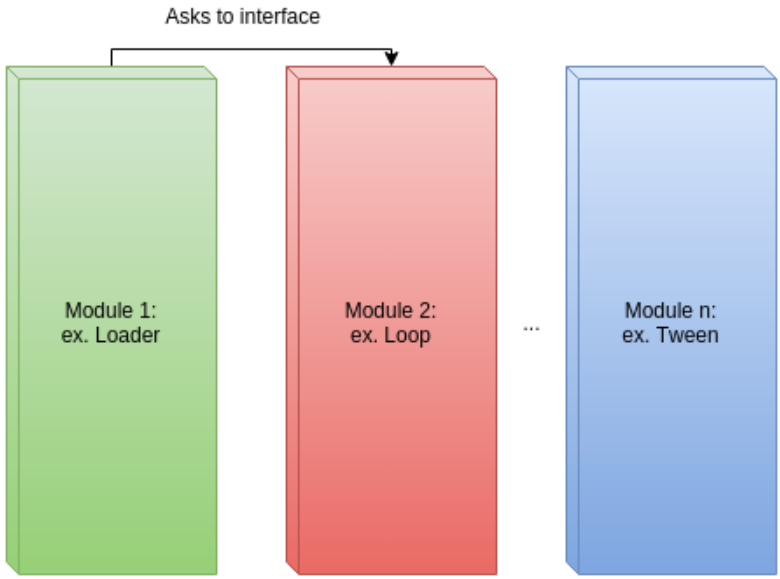


Figure 3.1: Vertical SoC

Horizontal separation

Most applications separate the code horizontally depending on their role. This boundaries are called layers and it's normal to have 3:

- **Presentation Layer**: Handles UI. Encapsulate all user interface concerns in order to allow the application domain to be varied independently.
- **Business Layer**: Handles the logic of the applications. Receives petitions from the presentation layer that transmits into querys for data to the RAL.
- **Resources Access Layer or RAL**: Provides access to data wherever it is, local storage or remote without the rest of the application having to care where it is.

Our core modules will have no user interface, nor data persistence, at least for the first version, also most will only require a single interface with a single default implementation. For this reasons layer boundaries won't be implemented. But a module has been said to be a black box that can implement different components of a module separately just with the condition of offering a unified interface. To allow this we will use the facade pattern to expose module functionality.

Facade pattern

The objective is to *Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use*

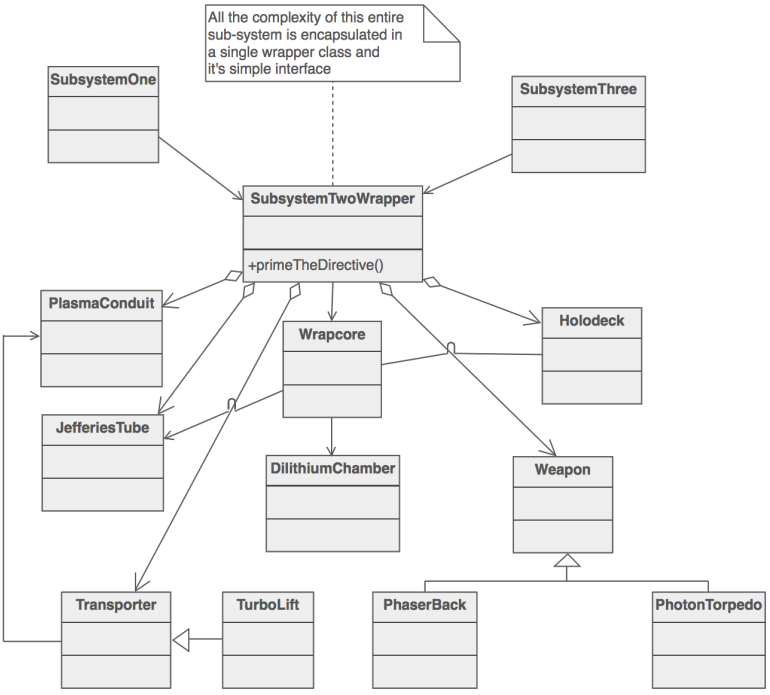


Figure 3.2: Facade Pattern schema

This means that the interfaces we will present individually on the next section must correspond to the facade we'll show, but that doesn't mean that a module is form by just one interface and one implementation. This comes handy when dealing with some system that

require sub systems of its own. For example the input system has to control keyboard input, mouse input and touch input. These functionalities can be modeled individually and brought together at the facade.

3.4.2 Object generators

Most of the features correspond to unique services that only are instanced once through the game but we have a special type different kind also. The modules that will generate the objects of the game that we will play with. They are responsible for creating all instances of them, we will use the factory function pattern. This pattern avoids the use of the new operator, that is controversial along the JavaScript community, instead is consists in a function that returns a new object each time it is invoked, like this:

```
function Factory( param ) {

  function feature() {
    obj.prop++;
  }

  var obj = {
    prop: 1
    feature: feature
  }

  return obj;
}
```

3.4.3 Overall structure

The first version of game wrapper modules will look like this:

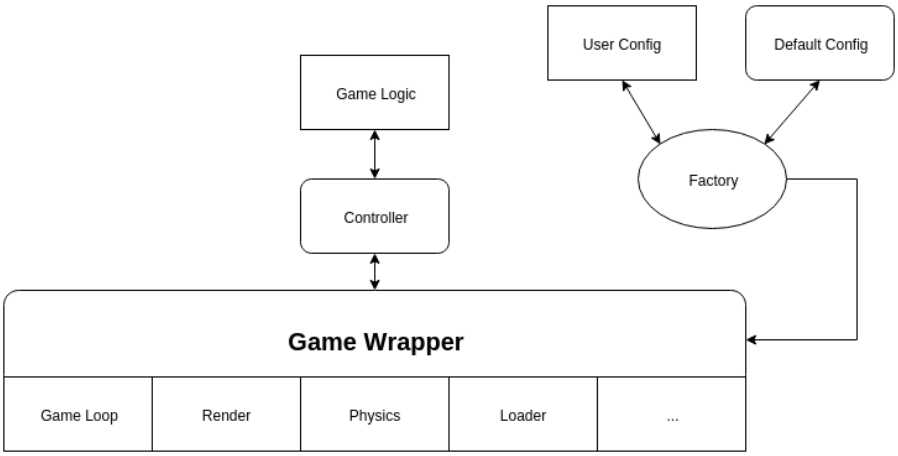


Figure 3.3: Architecture design

We have the modules but there are a couple of thing to clarify. Who will handle the flow of the game and how can the user communicate with the structure as a hole.

Controller

The controller will be a special module. When a game is instantiated it will pass the game config parameters to it and it will be responsible for getting all systems up, after that it will return control to the user so he can program the mechanic.

In the v1.0 case the controller will first load all assets, initialize the render with the required sizes and start the loop, then it will give control back to the user.

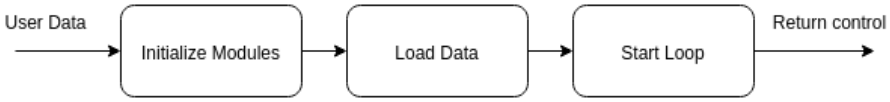


Figure 3.4: Controller flow

This modules is crucial as it will serve as a nexus between modules, when getting up the systems it will send them a reference to the game allowing them exchange data.

Launcher/Exposer

The Launcher has 2 fundamental tasks. Expose a global variable for other to access the game-wrapper and return an game generator with the required config.

Expose:

When ES2015 it's completely supported, most developers will use the JavaScript new module system to avoid exposing global variables that can pollute the namespace, override other libraries with the same name and open a window for a malicious user to mess around. But right now ES5.1 is the de facto standard and it forces to add a global variable to the namespace if we want to be independent from any module library. The launcher takes care of that.

It will also act as a factory for new game generators given a specific configuration, lets see an example of usage:

```
var engine = GW( customModules );
//if no parameter is send, it will use the default modules
engine( gameConfiguration ).then( function( game ) {
    ... //the game
});
```

As shown in the example, the launcher will send a promised that when resolved will send the instance of the game.

Default configuration object

To tell the wrapper which are the modules available by default, we need some way to locate them. A configuration module is built apart from the rest. It will contain a reference to each of the facades the modules provide and export an object containing them. This object will be loaded by the launcher and compared with the custom modules loaded by the user, giving priority to the user modules when names collide.

Module initialization

As commented at the controller, every module must have an initialization method which will receive the game instance. The name of the method will be initialized and it will have to do all work related getting the module running and subscribing to other modules events.

If a custom module doesn't have this method, it wont receive the game instance, making it difficult to connect with other pieces of the wrapper.

3.5 Specification

After discussing the general needs lets focus on each specific requirement. Briefly comment it and show what the interface would look like.

Let's start with those modules that provide core functionality.

3.5.1 Loader

In general, a loader is the part of a system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. In our case the loader gives 2 basic functionalities. The first and most important is loading the game assets from the web onto the client browser. The second is to assign a name to each resource, making it easier to work with it during the development of the game.

The loader interface will have methods to load the 3 most used types of files in a game: images, audio and data (json objects).

For the sake of controlling loading times, instead of loading one file at a time, loaders use to have a loading queue that only loads the files on it when an event is fired or the user specifies it. The interface will have a start method which will start loading the files and return a promise that will be resolved after all the elements are loaded.

Interface

```
loaderInterface = {
  initialize : function( game ){...},
  start: function(){...},
  image: function( name, url ){...},
  audio: function( name, url ){...},
  json: function( name, url ){...},
  getResource: function( name ){...},
```

```
}
```

3.5.2 Loop

The Loop is used to decouple the progression of game time from user input and processor speed. Almost every type of game needs one. A game loop runs continuously during gameplay. Each turn of the loop, it processes user input without blocking, updates the game state, and renders the game. It tracks the passage of time to control the rate of gameplay. This part of the program is really important as it will be the running code about 90% of the time the game is active.

In the first approximation after loading the resources, the loop will start running through 4 defined stages for each frame:

- **process**: The initial stage, its purpose is to take care of actions such as input management.
- **update**: Runs zero or more times per frame depending on the frame rate. It is used to compute anything affected by time - typically physics and AI movements.
- **render**: Update the screen, by changing the DOM or painting a canvas.
- **postRender**: Runs at the end of each frame and is typically used for cleanup tasks such as adjusting the visual quality based on the frame rate.

Each of the stages will contain a bucket of functions to execute, the user will add or remove functions from these buckets to add or delete functionality. The interface requires one add and remove function for every bucket present. The execution of the function buckets will be delegated to the implementation as added functionality may be needed at these steps.

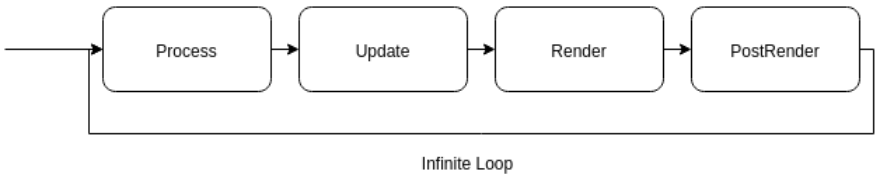


Figure 3.5: Game Loop stages order

FPS

Frames per second is a typical unit of control for performance of the loop. It sets how many times the loop will be executed every second, being crucial to the user experience. With the actual technology, 60 FPS are considered the standard for pc and consoles so the library will aim for this speed, however as web games can run in environmets with lower resources, the user has to be able to change the desired FPS value.

Start

Even though the loop will run for most of the game, there are some times when it is not necessary, at loading time for an instance (always speaking of a simple approximation, a more complex engine will have a specific loop through the loading that will show a loading splash). a Start method is needed.

Interface

```
loopInterface = {
  FPS : 60,
  start: function(){...},
  process: function( bucket ){...},
  update: function( bucket ){...},
  render: function( bucket ){...},
  postRender: function( bucket ){...},
  getFPS: function(){...},
  setFPS: function( number ){...},
  addProcess: function( func ){...},
  removeProcess: function( func ){...},
  addUpdate: function( func ){...},
  removeUpdate: function( func ){...},
  addRender: function( func ){...},
  removeRender: function( func ){...},
  addPostRender: function( func ){...},
  removePostRender: function( func ){...}
}
```

3.5.3 Render

Our render will literally show the game to the player as it's the module in charge of drawing the elements onto the screen.

All the hard work will be done by the implementation so the interface will be really simple.

The initialize method will generate the game canvas with the asked dimensions, it will add the renders refresh function to the loops render stage.

It will also need an add and remove function for all the elements visualized in our wrappers case images and text.

Interface

```
RenderInterface = {
  initialize: function( game ){...},
  addSprite: function( image, x, y){...},
  removeSprite: function( image, x, y){...},
  addText: function( text, x, y){...},
  removeText: function( text, x, y){...}
}
```

3.5.4 User Input

User Input will be formed by 3 micro systems consisting on the 3 technologies the game will accept input: keyboard, click and touch. But something special happens here. Even click and touch are applied to different interaction devices (one the mouse the other the hand), they can be treated equally, as you will want your desktops game click to be a touch button on a mobile environment and vice versa. But you can also require both as some laptops or computer screens provide touchscreen technology on platforms that usually are handled with a mouse. For this reason they are presented in the input facade as the pointer sub-object.

Interface

```
InputInterface = {
  initialize: function( game ){...},
  keyboard: {...},
  pointer: {...}
}
```

Keyboard

As the name says this component will handle the keyboard input. It will add a listener event to each key of the keyboard in order to track when is being pressed and when is being released. This will allow a method called isDown to check if a specified key is being pressed at a certain time in the game.

The initialize method will handle the initialization of all key related events.

```
Keyboard = {
  initialize: function( game ){...},
  isDown: function( key ){...}
}
```

Pointer

For each pointer on the screen a pointer sub object will be needed. In case of the mouse it will only be one but with touch screens it can be more. 3 types of events will be tracked, pressing releasing and moving, this way we can deliver not just clicking check but also drag and drop.

Every time one of this interactions is done, the handler will go through all the objects of the world (commented below) and check if any of the clicks is on an object, this way the object can add a callback.

A boolean property is added to control if we want multiple pointers or just one. Also a general method onClick is added in case we want to centralize click control.

```
Pointer = {
  initialize: function( game ){...},
  main: {...},
  second: {...},
  third: {...},
  fourth: {...},
  fifth: {...},
  all: [],
  allowMultiple:{...},
  onClick: function( callback ){...}
}
```

3.5.5 Tween

Tween is going to be a particular case as it needs a factory object to return the tween controller. Each time yo want to tween and object yo will call the tween factory with the object you want to modify and receive a controller object as a result.

```
objToTween = { x: 200};
var tween = game.tween.get( objToTween );
tween.to( { x: 1000}, 5000 );
tween.easing( game.tween.easing.Cubic.In )
tween.start();
```

This example gives us an idea of how it is going to work. Get will be the factory function, aside from creating the controller around the given object it will give useful functions.

- **to**: Sets the parameters to modify and its final value as a first parameter, for the second parameter it gets the time the tweening must be working until arriving to the desired state.
- **Delay**: Delays the starting time of the tweening from the moment the user starts it n millisecons, being n the parameters passed.
- **Start**: Launches the tweening.
- **onUpdate**: Each time the tweening modifies the value of the object it will launch the function passed here.
- **easing**: Sets an easing function for the tween.

Easing functions

Easing functions specify the rate of change of a parameter over time. This make the tween not just change the objective value the same amount each fraction of time. It adds realness and diversion to value changes.

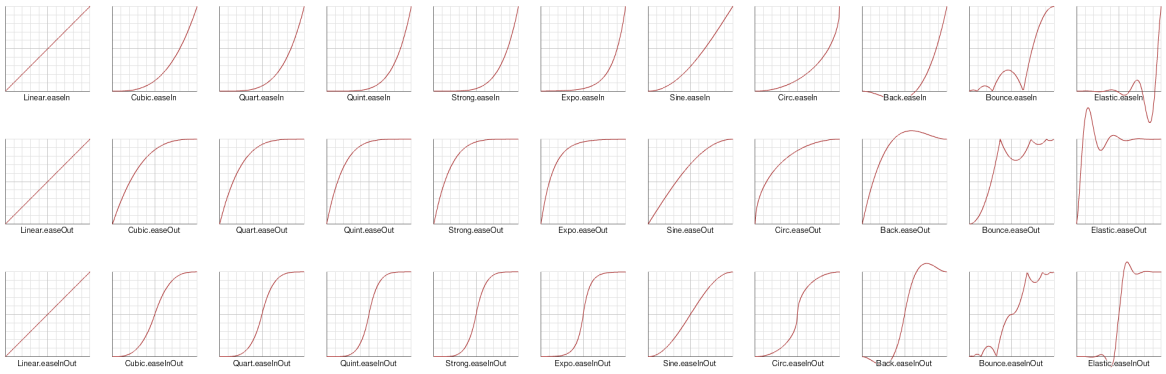


Figure 3.6: Examples of easing functions

The next easing functions will be available to use:

- **Back**: With sub types: In, InOut, Out.
- **Bounce**: With sub types: In, InOut, Out.
- **Circular**: With sub types: In, InOut, Out.
- **Cubic**: With sub types: In, InOut, Out.
- **Elastic**: With sub types: In, InOut, Out.
- **Exponential**: With sub types: In, InOut, Out.
- **Linear**: With sub type: None.
- **Quadratic**: With sub types: In, InOut, Out.
- **Quartic**: With sub types: In, InOut, Out.
- **Quintic**: With sub types: In, InOut, Out.
- **Sunisoidal**: With sub types: In, InOut, Out.

All this functions will be accessible through the function *easing*

Interface

As every module the initialize method will be needed, in this case, to tight the loop update method with tween updates. The resultant interface will be

```
TweenInterface = {
  initialize: function( game ){...},
  get: function( obj ){...},
  easing: {..}
}
```

3.5.6 Physics

Moving things from one point to another is good, but a lot of games need more than that. They need to simulate reality physics. This module will solve that. Unlike other modules, setting up the physics is a very resource demanding task. For that reason even though it will have the initialize method, it won't set up the physic world until it is explicitly demanded by the usePhysics method.

When physics are demanded, a world is created, this world contains all the elements of the physics simulation. Every object with physics will have a physics object related to him and every transformation that happens to one of the ends will have to be transmitted to the other, taking care not to make circular dependency as it will lead to awkward scenarios.

A method to set the gravity of the world is also required.

For the first version physical objects will only be rectangles. The type of physics applied to detect collision will be AABB or Axis-aligned minimum bounding box.

AABB

In geometry, the minimum bounding box for a point set in N dimensions is the box with the smallest measure within which all the points lie.

The axis-aligned minimum bounding box for a given point set is its minimum bounding box subject to the constraint that the edges of the box are parallel to the coordinate axes.

Axis-aligned minimal bounding boxes are used to approximate the location of an object in question and as a very simple descriptor of its shape.

It's very used as object intersection check is less expensive with this method than others, as it just has to check the coordinates to compare and discard pairs.

Signals

To notify collision events a signaling library will be used. A Signal is a type of Observer pattern implementation similar to an Event Emitter/Dispatcher or a Pub/Sub system, the main difference is that each event type has its own controller and doesn't rely on strings to broadcast/subscribe to events.

The signals will be introduced to the objects when the physic body is created. This way the user can set handlers for when this actions take place.

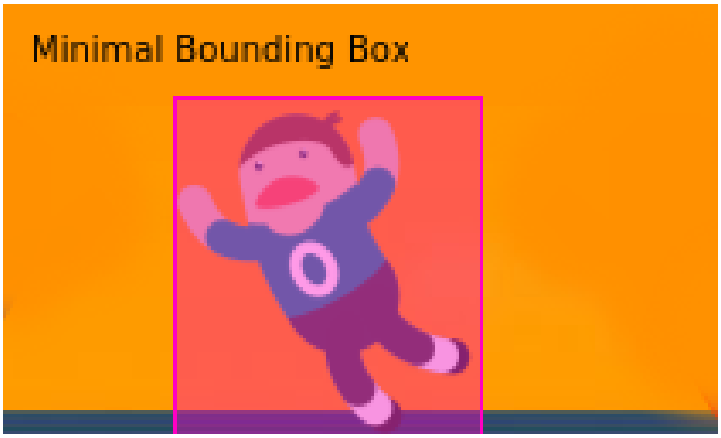


Figure 3.7: MBB surrounding sprite

Interface

```
PhysicsInterface = {
  initialize: function( game ){...},
  usePhysics: function(){...},
  setGravity: function( gravity ){...},
  rectangle: function( objects ){...}
}
```

3.5.7 World

This module serves as a bridge between game objects and modules like the game loop and the render.

It saves the task of having to monitor each object independently. Only games in the world will be rendered onto the screen or updated.

Interface

```
WorldInterface = {
  objects: [],
  initialize: function( game ){...},
  add: function( element ){...},
  remove: function( element ){...}
}
```

3.5.8 Audio

The audio module will be our audio controller. When a user wants to start, pause or stop an audio, it asks the audio module giving its id name. The id name is inserted when the audios are loaded through the loader.

A get method is added for those programmers that want to manage audios separately and a reference to the audios list is included on the interface.

Interface

```
AudioInterface = {
  initialize: function( game ){...},
  sounds: {...},
  play: function(name){...},
  pause: function(name){...},
  stop: function(name){...},
  get: function(name){...},
}
```

Now let's take a look at our object factories.

3.5.9 Resource

Every asset the loader processes will have different properties depending on the implementation. The resource object specifies only the required objects for the wrappers and its implementation is an object factory that returns a correct formatted resource object from an arbitrary object data.

For the first version only the name and the path to the resource will be required, Both parameters will be strings.

Interface

```
Resource = {
  name: '',
  url: ''
}
```

3.5.10 Sprite

The most basic and used object, nearly all representations of the game use this object. Every sprite can change position, rotation or scale.

A only read shortcut to read x and y positions will be given.

Interface

```
Sprite = function( game, x, y, image ) {
 return {
   scale: function( x, y ){...},
    position: function( x, y){...},
    rotate: function( angle ){...},
    x : function(){...},
    y : function(){...}
}
```

3.5.11 Text

These days games are very visual but even the most extreme one, has some text. Our text object will be very simple and similar to sprite object, in fact it will have the same properties, changing the image with the text and adding a configuration object to set the style of the text on its invocation.

Interface

```
Text = function( game, x, y, text, style) {
 return {
   scale: function( x, y ){...},
    position: function( x, y){...},
    rotate: function( angle ){...},
    x : function(){...},
    y : function(){...}
}
```

The v1.0 style properties available will be:

- **font**: The style and size of the font.
- **Color**: Text color.
- **align**: In case of multiple lines: 'left', 'center' or 'right'.
- **stroke**: Text stroke color.

Chapter 4

Implementation

Now that all the required modules are listed and defined it's time make a default implementation of everything. On Annex A the're a list and short description of every library used for this purpose.

While reviewing the implementation process lets focus also on the tooling, several concepts and technologies are being used here. First and not very relevant, the project is being developed on an Antergos linux OS with the WebStorm IDE under an education license, This is not important because one of the aims is to be able to edit the project from any platform and/or editor. This is possible thanks to NodeJS and NPM:

NodeJS is a JavaScript runtime built on top of Chrome's V8 JavaScript engine. This allows JS to work server side. In our case all the tools used in the process are made with JS and runned on the environment.

NPM is nodes package manager. All services used for building, implementing the wrapper, even the wrapper itself, are available at this repository.

4.1 ES6/ES2015

In the JS introduction we talk about this near-future standard, but actual browsers don't support some or most of the new functionality. To surpass this obstacle we have two solutions that combined together can make most of the features available on today's browsers: polyfill and transpilers.

A polyfill, or polyfiller, is a piece of code that provides the technology that the browser is expected to provide natively. Allowing your code to be prepared for when this new syntax is available.

A Transpiler is a tool that transforms code with new syntax into old code.

Combining this 2 techniques we have Babel. Until a couple months ago it had the most complete set of ES6 features, it has been surpassed by Microsoft EDGE but the user still has to activate a flag, which most users won't know how to do it.

For actual info on browsers and platforms ES6 implementations check the web Kangax compatibility table which has up to date table charts.

Once we have the ES6 problem solve we can start working.

4.2 Folder structure

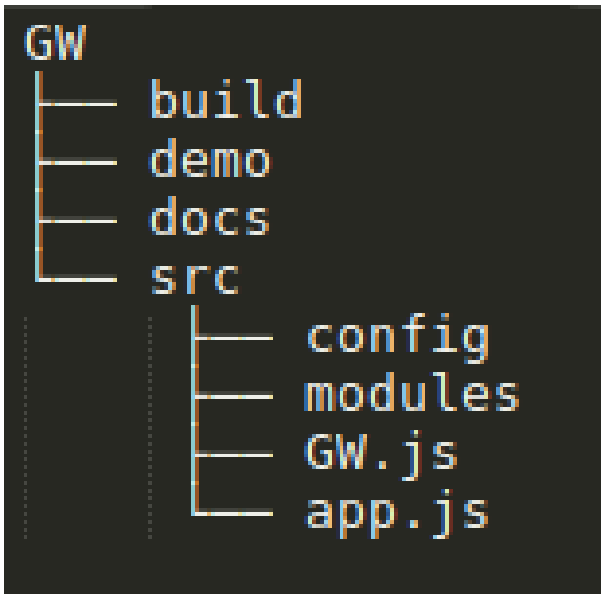


Figure 4.1: Folder Structure

- **build**: Will contain the production ready game wrapper.
- **demo**: A mini game demo to test the wrapper.
- **docs**: Wrapper auto-generated documentation.
- **src**: Were the library code is contained
- **src/config**: Default configurations folder.
- **src/modules**: Each sub folder of this directory will contain a module.
- **src/GW.js**: Base file, defines GW() factory.
- **src/app.js**: Exposes GW to the window for older browsers.

Module development

Each module is started with an interface full of empty functions. This interface is implemented in the implementation file. This way in case the implementation lacks some methods, nothing won't happen. This process can be done various times depending on the subcomponents each model has. Finally the facade wraps its functions around the implementations and exposes its public API.

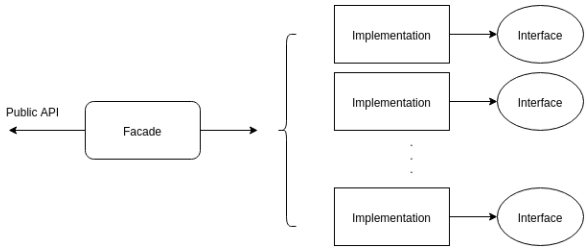


Figure 4.2: Module development schema

4.2.1 Loading the wrapper

Now that development and language is clear we have a version to work with but it still has important flaws. Loading the wrapper on a web looks like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Test Wrapper</title>
    <script type="application/javascript" src="GW/GW.js"></script>
    <script type="application/javascript" src="GW/config/defaults.js">
    </script>
    <script type="application/javascript"
    src="GW/modules/loader/interface.js"></script>
    <script type="application/javascript"
    src="GW/modules/loader/implementation.js"></script>
    <script type="application/javascript"
    src="GW/modules/loader/facade.js"></script>
</head>
```

And that is just one module, without even loading external dependencies. We would need to import more than 50 scripts to get the wrapper going, this is not manageable. Also its not good for the game users. Having to load so many files gets to longer loading periods which the user does not like, and it will make very easy for the players browser to cache some files and don't update them even though we have delivered a new update.

4.3 Static Assets Pipeline

Static files or assets are files you are going to serve from a CDN or a server, The pipeline is the process or series of process that compresses and minifies those file to be ready to deliver to the client. Babel is part of our assets pipeline as it transforms our code, but we are also going to need a program that mixes or bundles all our codebase together, making our wrapper an only file. For this task the selected program webpack.

Webpack is a very young program that's getting popular really fast. Bundling was usually achieved by task runners such as Gulp or Grunt, these programs are great and allow users to automatize lots of tasks but if you are only looking for a bundler, webpack is a good choice.

So we create the app.js file on the root of our code and require inside GW and configure webpack.config.js according to what we want it to do, then using a feature of npm that allows to run bash scripts webpack is launched. what we get is a file named gw.js. Finally a good and usable version of GW is available, what is next?

4.4 Extensible

GW is said to be easily extended, how is this going to be achieved? Through its design and two more crucial sections.

4.4.1 Documentation

Every facade is documented and available in detail at the doc pages. Anyone wanting to implement one of the existing modules can go to the docs an see every function they must implement, along with the input parameters received and the output it needs to give. This speeds up the development a lot, leaving any uncertainty away.

The documentation is generated through the use of JSDoc 3. An API documentation generator for JavaScript similar to the famous Java documentation JavaDoc.

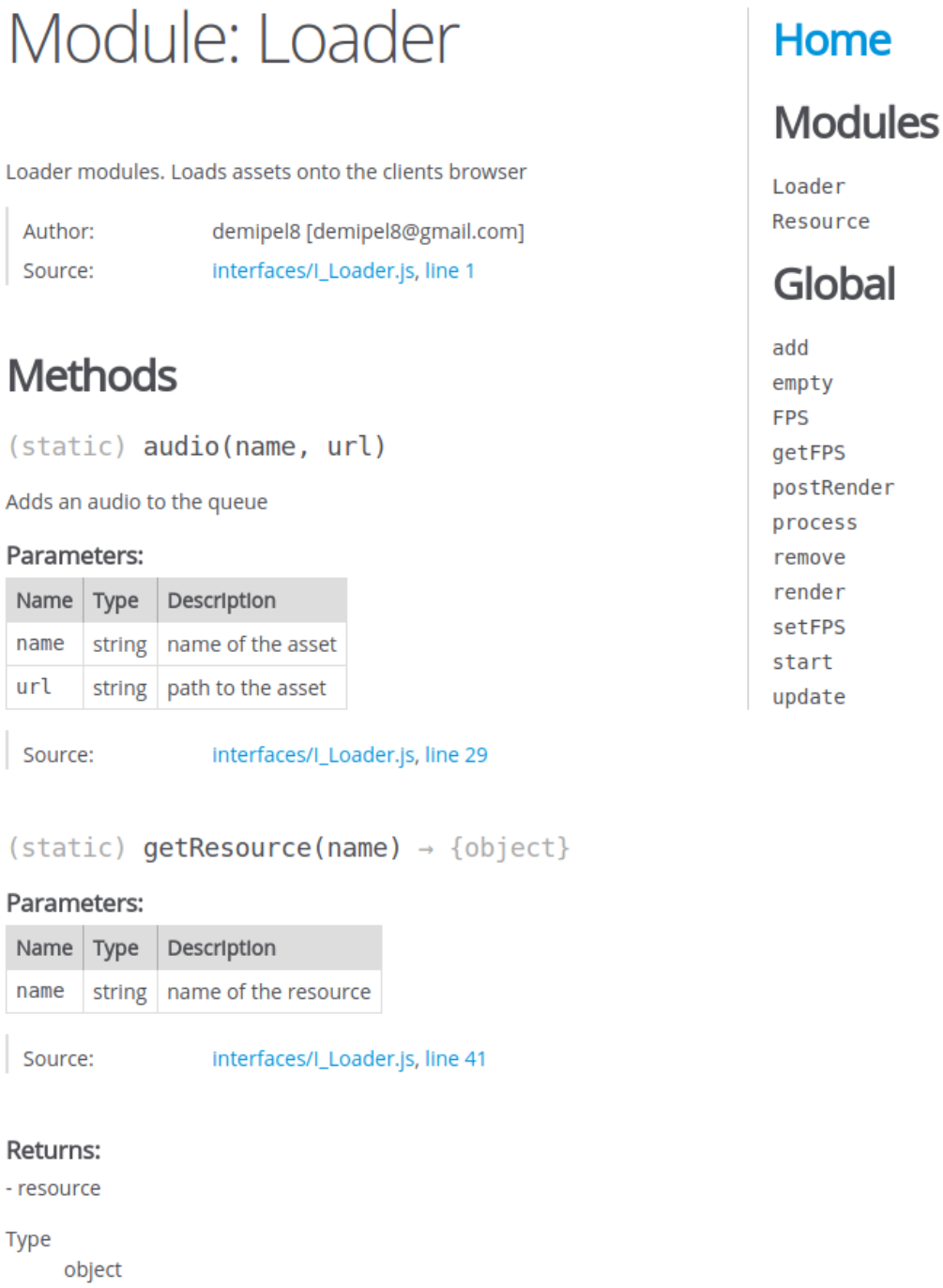
At the start of every Facade and its method there is a bloc of formatted comments that document aspects of the code. For example the Loader page on figure 4.3 comes from comments like this:

```
/**
 * Loader modules. Loads assets onto the clients browser
 * @module GW.Loader
 * @author demipel8 [demipel8@gmail.com]
 */

 import implementation from './implementation'

export default {
  /**
   * Used to configure any initial state required by the implementation
   * @param game - game object
   */
  initialize : implementation.initialize,
```

So this an easy way to implement, but it will be even easier if we could automatically create the files structure for a module. With the use of the task-runner gulp and npm script it's possible. the order npm run module – module name create a new folder on the modules directory with *interface.js, implementation.js and facade.js* empty files, ready to be filled. With generation and programming done the next step is validation. The docs are available but it's still a slow task to verify manually if everything is correct. Lets use test to solve this!



Module: Loader

Loader modules. Loads assets onto the clients browser

Author: demipel8 [demipel8@gmail.com]
Source: interfaces/I_Loader.js, line 1

Methods

(static) audio(name, url)

Adds an audio to the queue

Parameters:

Name	Type	Description
name	string	name of the asset
url	string	path to the asset

Source: interfaces/I_Loader.js, line 29

(static) getResource(name) → {object}

Parameters:

Name	Type	Description
name	string	name of the resource

Source: interfaces/I_Loader.js, line 41

Returns:

- resource

Type
object

Home

Modules

Loader
Resource

Global

add
empty
FPS
getFPS
postRender
process
remove
render
setFPS
start
update

Figure 4.3: Early version loader docs

4.4.2 Testing

Testing is essential to stable, maintainable code. Much more important if the project is maintained by several people because when something goes wrong, it's easier to trace and discover why it's failing. In our case it helps users validate their custom modules. Whenever you finish an implementation, you load the test file from the original implementation and run it with your code, if all tests pass it is ready to go into production. Lets see the technologies behind this.

Karma

Karma is a test runner for JavaScript. It sets an environment where tests can be done on multiple browsers and devices at once. It is very easy to use, in our case we have to make it work together with babel and webpack, so we create a *spec.bundle.js* file which will load together all tests. A configuration of this style allows us to also write tests in ES6.

Among all the configurations it has, we will ask it, to never stop once its running, to redo all tests when the tests or code files are modified and to run the tests in PhantomJS by defect. PhantomJS is a headless browser, which means it doesn't has a graphic user interface. It is ideal to run CLI tests.

Mocha

The test framework, allows us to evaluate the tests and show the results. It can output the results in various ways, we will see them at the console that invokes karma. Probably the best feature of Mocha is that you can use any assertion library that you want.

Chai

Chai is the assertion library used to test GW. An assertion is a statement of fact. Each assertion we make on a test must be true or the test will fail. we are using the BDD, Behaviour Driven Development, expect assertion. Every assertion will be formulated using expect

A sample of how all this programs work together is shown on this loader tests snippet.

```
describe ( 'Loader Module default implementation in GW - resource-loader',
 function() {

  beforeEach( () => {
    Loader.initialize();
  });

  describe( '#start', () => {
    it('returns a promise', () => {
      var promise = Loader.start();
      expect(promise).instanceOf( Promise );
    } );
```

At the test *return a promise* we expect Loader.start() invocation to return a promise. The results of the test are shown at the console.

```
Loader Module default implementation in GW - resource-loader
  ✓ should have correct public methods
  ✓ should work
  #initialize
    ✓ works based on the two tests above
  #start
    ✓ returns a promise
    ✓ works without loading anything
```

Figure 4.4: Karma CLI

With all this programs our testing suite seems complete but there is one thing remaining: Coverage. All tests can be passed, but we also need some context for the results. How much of the code composing the tested file is actually tested? coverage give the percentage of code tested. We use a small library, isparta, that is ready to be used on ES6 with babel. Results are shown at the console along with the tests.

```
PhantomJS 1.9.8 (Linux 0.0.0): Executed 22 of 22 SUCCESS (0.069 secs / 0.044 secs)
TOTAL: 22 SUCCESS

----------|----------|----------|----------|----------|----------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------|----------|----------|----------|----------|----------------|
----------|----------|----------|----------|----------|----------------|
All files |      100 |      100 |      100 |      100 |                |
----------|----------|----------|----------|----------|----------------|
```

Figure 4.5: Coverage test

Now users can trusts the test suite to give meaningful results, and only worry why their code is failing the tests. The wrapper is ready to be made a public package.

4.5 Validation

With everything on board and working, it is time to test our library and validate its model. A small demo test that makes sure every subsystem of GW is up and running. A very simple mechanic but representative of the potential this tool can offer.

4.5.1 Concept

We have a closed stage with some platforms. Our player starts at the top platform and has to go through a whole at the bottom. To try and stop us there are some enemies that kill us the moment we touch them.

How do we check all the wrapper features:

- **Loader**: It loads a set of images and audio.
- **Controller**: After loading the assets, the controller will return the programmer control of the game.
- **Render**: Must draw on the screen our game objects.

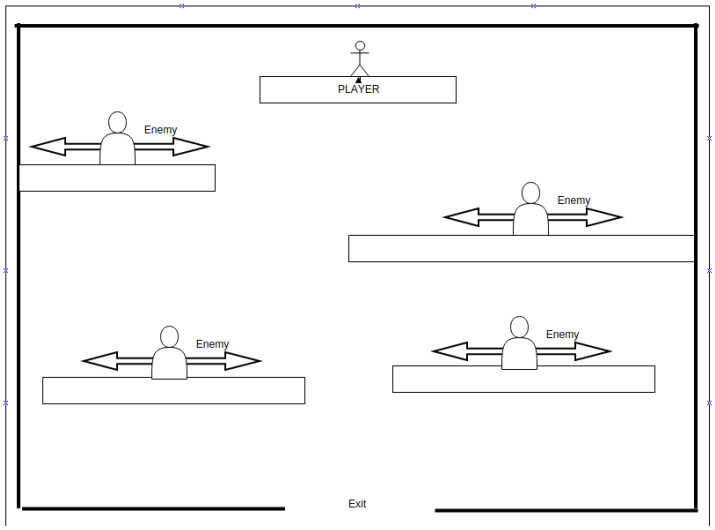


Figure 4.6: Visual representation

- **Loop**: If any object changes its states it will be working.
- **Input**: The user will be controlled in two ways:
  - **Keyboard**: Left and right arrows will handle horizontal movement and the up arrow will allow it to jump.
  - **Click**: When a click is done the player will automatically move to the x coordinate of the click event.
  - **Touch**: Exactly the same as click.
- **Tween**: The enemies will move from left to right and vice versa with a tween function.
- **Physics**: Walls and platforms will have a physic body, also the player and enemies. This way they will be able to move on top of platforms and get stuck when trying to surpass a wall. Also collision will be checked between enemies and the player.
- **Audio**: Some actions of the game like, winning, loosing or jumping have audios attach that sound when that event is triggered.
- **Sprite**: All images of the game are sprite objects.
- **Text**: A countdown timer will be shown at the top left corner of the screen.
- **Resource**: Every time a sprite object is created it asks the loader for the resource path, which is returned as a Resource object.

.

4.5.2 Ending conditions

Win: The player will win if he arrives to the hole on the floor without any problem.
Loose:

- The countdown gets to 0.
- The Player collides with an enemy.

If the game can be played without errors and gets right this 3 scenarios. The wrapper passes the test successfully.

4.6 Publishing

At the start of this chapter, npm was said to be the biggest repository for JS programs here is were are going to deploy the wrapper.

4.6.1 NPM

To update a package to npm the project must be converted into a valid package. This is obtained through the creation of the *package.json* file. This file already exists in the project as it contains all the dependencies of the project. We will add some configuration so that npm recognizes it as a public and valid package. The whole package.json file forms the Appendix B.

To publish the project we get a console and go to the root path of our project. make sure we are ont the master branch of the repository and everything is commited. we log onto npm with the command *npm login* and then *npm publish*, after the process is done our package is publised.

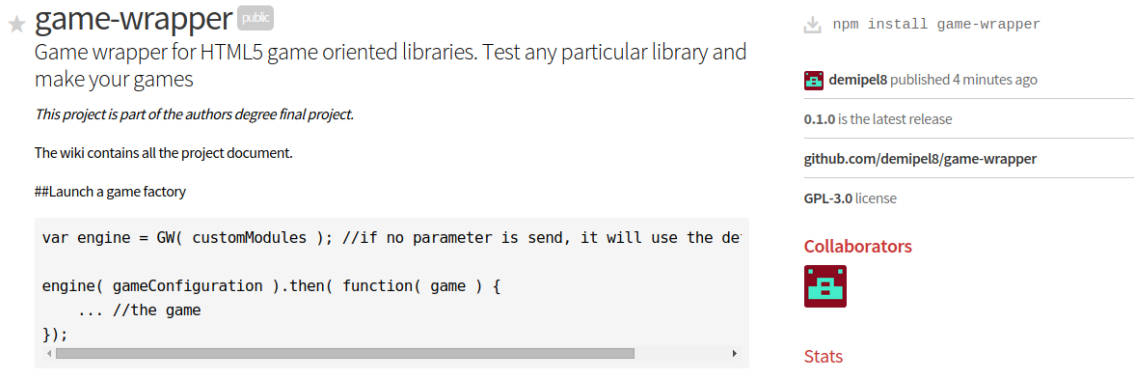


Figure 4.7: The wrapper at npm

4.6.2 GitHub

The project is controlled through a git repository and saved remotely using the GitHub platform. The working flow used is called git-flow. With git flow only 2 branches are published remotely, Master and develop. Master has the ready to production version of the library while development has the cutting-edge stable version. Master branch should only be used to read or use directly in a project, for any feature development develop is the one.

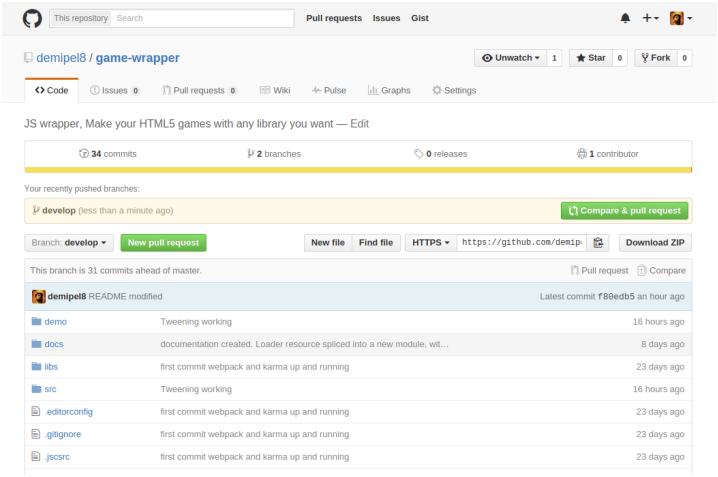


Figure 4.8: GitHub page

All the issue reporting will be done through GitHub. And the updated version of this memory will be on the wiki of the project.

4.6.3 Landing Page

As version number 1 it's still not released. We want to generate a landing page to generate some noise, were people can subscribe to get news of GW development. Using LaunchRock app free membership, we have created a simple but elegant landing page for people to see.

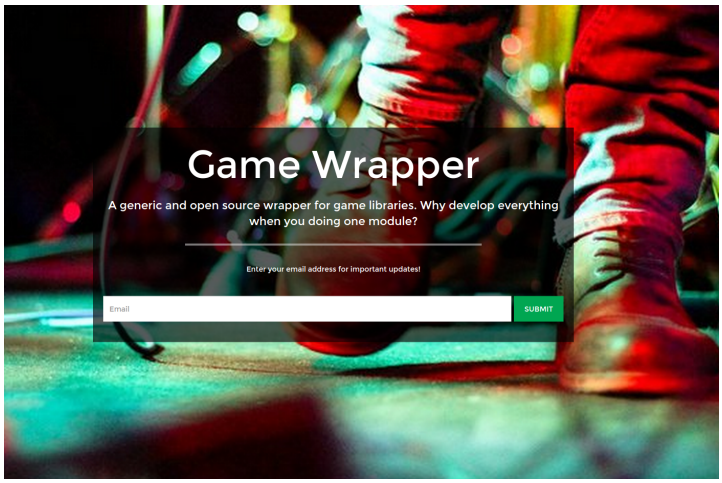


Figure 4.9: Landing Page

With this tool we can get statistics of the people that access our web and engage them with a periodic newsletter, telling the advancements of GW.

4.7 Future Work

This is a very young project, there are lots of thing to do. On a business level, the model has to be validated by watching user usage of the product and the feedback they make.

A proper web page has to be made and a sub domain to contain the docs.

In the technical side here are a couple of things.

- **GW**: Value the use of a config object instead of controller fixed parameters in order to spread the initial parameters through all models
- **Input**: Offer support for the new gamepad API.
- **Loader**: Spritesheet loading.
- **Render**: Animation support.
- **Sprite**: More options. Refactor default implementation.
- **Physics**: Add circle and polygonal shapes. Also constraints.
- **Particles**: Particles support.
- **Audio**: Search for multi-channel support.
- **Controller flow**: As game stop being mini demos and turn to bigger project they need a way to organize its code. A stage/scene component will be required.
- **Package**: Create a complementary tool make easy packaging games for deployment on multiple platforms .
- **Demo**: Have a decent demo online via Heroku, so continuous integration tools like Travi CI can be tested.

Version 1.0 is months away as development it's only done in free time after work. But it will eventually come out.

4.8 Conclusion

4.8.1 General

The development have restrained a bit the objectives of the project, but have also delimited it from parts that wasn't as critical as in an initial moment it was thought. Through the whole development no game has been made but just a series of demos to proof the features included. So small that it wasn't even worth it to add them here.

As we have just said, the limitation has made shift the concept from creating a small tool and a game to create a decent wrapper that allows other people to learn what happens inside a web game, and not just that, but also experiment and play with it.

Most of the tools used in the project have been new. I use related services but a bundler like webpack is something new. The whole testing on which I have made a lot of emphasis, it completely new to me, as testing at my company wasn't done until mid-November of 2015.

The scope of the whole wrapper feels too big for a degree final project when tried to compare with mature game engines. The extend a specification of their interfaces is way beyond.

The general feeling is accomplishment and feeling just at the start of a very long term project.

4.8.2 Personal

Making a standard interface is not easy. Every time you face a problem on the development, a quick fix is available, but it screws all the premises you have established. So sometimes it seems like you advance really slow or that you have to change half of the design for a small detail. But I have learned and enjoyed making GW. It's the first time I can properly design my software without anyone telling me that it needs to be done by Friday. This gives me a perspective of how software must be done, independently from the purpose of that program.

I also have learned to integrate all the 8 different libraries of the default implementation, each one with its quirks and perks, but mostly very well done and structured pieces of software. All of them allow very easily to be integrated.

The wide basic knowledge acquired on the degree mixed with the knowledge of JS gaming achieved at my work place have helped a lot on the design phase and made implementation go smoother than anticipated.

Bibliography

Boissière, Alexandrine (2015). *Efficient Static Assets Pipeline with Webpack*. URL: https://www.youtube.com/watch?v=w1dAb_Umt8o.

Facade Design Pattern. URL: https://sourcemaking.com/design_patterns/facade.

Foundation, Mozilla (2005-2015). *Mozilla Development Network*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

Greer, Derek (2008). *The Art of Separation of Concerns*. URL: https://sourcemaking.com/design_patterns/facade.

HTML5 Game Devs (2013-2015). URL: http://www.html5gamedevs.com/.

Jína, Vojta (2012). *Testacular:JavaScript test runner*. URL: https://www.youtube.com/watch?v=MVw8N3hTfCI.

Seibel, Peter (2006). *Coders at Work: Reflections on the Craft of Programming*. Apress.

Simpson, Kyle (2014a). *You Don't Know JS: Scope Closures*. O'Reilly Media.

— (2014b). *You Don't Know JS: this Object Prototypes*. O'Reilly Media.

— (2015a). *You Don't Know JS: Async Performance*. O'Reilly Media.

— (2015b). *You Don't Know JS: ES6 Beyond*. O'Reilly Media.

— (2015c). *You Don't Know JS: Types Grammar*. O'Reilly Media.

Index

Appendices

Appendix A

Implementation libraries

A.1 Loader - Resource-loader

A middleware-style generic resource loader built with web games in mind.

github: `https://github.com/englercj/resource-loader`
npm: `https://www.npmjs.com/package/resource-loader`

A.2 Loop - MainLoopJS

MainLoop.js provides a well-constructed main loop useful for JavaScript games and other animated or time-dependent applications.

github: `https://github.com/IceCreamYou/MainLoop.js`
docs: `https://icecreamyou.github.io/MainLoop.js/docs/#!/api/MainLoop`
npm: `https://www.npmjs.com/package/mainloop.js`

A.3 Render - PixiJS

MainLoop.js provides a well-constructed main loop useful for JavaScript games and other animated or time-dependent applications.

github: `https://github.com/pixijs/pixi.js`
docs: `https://pixijs.github.io/docs/`
npm: `https://www.npmjs.com/package/pixi`

A.4 User Input - Keypress

Keypress is a robust keyboard input capturing JavaScript utility focused on input for games.

github: `https://github.com/dmauro/Keypress`
docs: `https://dmauro.github.io/Keypress/`
npm: `https://www.npmjs.com/package/keypress`

A.5 Tween - tween.js

JavaScript tweening engine for easy animations, incorporating optimised Robert Penner's equations.

github: `https://github.com/tweenjs/tween.js`
docs: `https://github.com/tweenjs/tween.js/blob/master/docs/user_guide.md`
npm: `https://www.npmjs.com/package/tween.js`

A.6 Physics - matter-js

JavaScript 2D rigid body physics engine for the web.

github: `https://github.com/liabru/matter-js`
docs: `http://brm.io/matter-js-docs/`
npm: `https://www.npmjs.com/package/matter-js`

A.7 Audio - howler.js

Audio library for the modern web. It defaults to Web Audio API and falls back to HTML5 Audio.

github: `https://github.com/goldfire/howler.js`
docs: `http://goldfirestudios.com/blog/104/howler.js-Modern-Web-Audio-Javascript-Library`
npm: `https://www.npmjs.com/package/howler`

Appendix B

Package.json

```
{
  "name": "game-wrapper",
  "version": "0.1.0",
  "description": "Game wrapper for HTML5 game oriented libraries.
  Test any particular library and make your games",
  "scripts": {
    "start": "webpack-dev-server",
    "web": "webpack",
    "test": "./node_modules/.bin/karma start",
    "docs": "rm -r ./docs/; ./node_modules/.bin/jsdoc -c jsdoc.config.json",
    "module": "./node_modules/.bin/gulp module -0"
  },
  "author": "demi ( Alvaro Martinez de Miguel )",
  "license": "GPL-3.0",
  "keywords": [
    "game",
    "wrapper",
    "HTML5",
    "javascript",
    "2d",
    "UPV"
  ],
  "bugs": {
    "url" : "https://github.com/Hooptap/HooptapMechanics/issues",
    "email" : "demipel8@gmail.com"
  },
  "main": "./build/gw.js",
  "repository": {
    "type": "git",
    "url": "https://github.com/demipel8/game-wrapper.git"
  },
  "homepage": "http://game-wrapper.launchrock.com/",
  "devDependencies": {
    "babel-core": "^5.2.6",
    "babel-loader": "^5.0.0",
```

```
    "babel-polyfill": "^6.0.16",
    "chai": "^3.4.0",
    "colors": "^1.1.0",
    "expose-loader": "^0.7.1",
    "gulp": "^3.9.0",
    "gulp-file": "^0.2.0",
    "isparta-loader": "^2.0.0",
    "jsdoc": "^3.4.0",
    "jsdocs": "^0.1.0",
    "json-loader": "^0.5.3",
    "karma": "^0.13.15",
    "karma-chai": "^0.1.0",
    "karma-coverage": "^0.2.7",
    "karma-mocha": "^0.2.0",
    "karma-phantomjs-launcher": "^0.1.4",
    "karma-spec-reporter": "0.0.18",
    "karma-webpack": "^1.7.0",
    "mocha": "^2.3.3",
    "mocha-loader": "^0.7.1",
    "phantomjs": "^1.9.19",
    "phantomjs-polyfill": "0.0.1",
    "webpack": "^1.8.11",
    "webpack-dev-server": "^1.8.2"
  },
  "dependencies": {
    "howler": "^1.1.28",
    "keypress.js": "^2.1.0-1",
    "mainloop.js": "^1.0.1",
    "matter-js": "^0.8.0",
    "pixi.js": "^3.0.8",
    "tween.js": "^16.3.1",
    "signals": "^1.0.0"
  }
}
```