Master thesis : Langage pour plate-forme de développement de jeux sur le web

Auteur : Chupin, Simon
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "computer systems and networks"
Année académique : 2016-2017
URI/URL : http://hdl.handle.net/2268.2/3160

University of Liège
Faculty of applied sciences

Master Thesis

Evolutive programming language for game development platform on the web

Author: Simon Chupin

Supervisor: Prof. L. Mathy

Master thesis submitted for the degree of
MSc in Computer Science and Engineering

Academic year 2016-2017

Acknowledgements

I would like to thank my supervisor, Prof. L. Mathy, who gave me the opportunity to work on such an interesting subject for my master thesis and who let me a lot of freedom on the conception of the application.

I would also express my special thanks to Renaud Hoyoux, who gave me useful advice.

Finally, I would like to express my deepest gratitude to all those who provided me time, support and encouragement during this year and all my years of university study, especially my family and my close friends.

Abstract

Evolutive programming language for game development platform on the web

Author: Simon Chupin
Supervisor: Prof. L. Mathy
Master thesis submitted for the degree of MSc in Computer Science and Engineering
Academic year 2016-2017, University of Liège

The continuous evolution of digitalization during the past decade has led to a developers and computer scientists shortage. In response to that trend, many tools have emerged to promote programming and particularly among children.

This works aims at developing a new impelling application to teach programming and encourage teenagers to get into informatics. The application concept is to provide simple tools and an adapted programming language to develop 2D games that run in the browser.

The challenge of this work is to bypass the tedious steps of usual languages and frameworks learning curves by giving the opportunity to produce complete games with little knowledge on programming and game development. The other key objective is to give an evolutive environment that allows children to progressively develop their skills as they gain experience.

This thesis presents the design choices made to achieves these goals and describes the application architecture and implementation.

Contents

Chapter 1

Introduction

1.1 Purpose

About fifteen years ago, the world has entered in the digital age, that is the point in time where human kind was able to store more information in digital than in analog format. And with this digital age, computer code has started to populate our everyday life. Not only the use of computer has exploded but also the development of new technologies, highly requiring programming, such as smart-phones, intelligence in cars, home automation and many more. We also find this digitalization in many companies with the use of automated processes and more recently with the development of artificial intelligence such as data mining to manage the company data and optimize their production.

This big change in the society has led to an increasing demand for computer programmers in the last decade. And more generally, coding skills has become a core skill requirement for a broad range of careers, not only for IT jobs. An American report [Tec16] states that 20% of well-paid job openings in 2015 was requiring coding skills. The same report assesses that coding jobs are growing faster than the overall job market with an estimation of a 12% faster growth.

Despite this increasing demand, the developer scarcity in the market is a reality. According to the 2016 edition of "Baromètre de la société de l'information du SPF Économie" [SE16], 46% of Belgian companies have encountered difficulties to fill ICT job vacancies in 2015. In Switzerland, a lack of $24,000$ computer scientists is forecasted for 2024 [Cha16].

However, the awareness of the coding skills importance has grown and many countries have put effort and investment to reverse the current trend. The idea is to sensitize and train more people to programming and a common thought is that it is both better and easier to learn it at an early age.

For example, the Obama administration has launched 'Computer Science for All' in 2016 [Smi16], a $4 billion initiative to give American students from kindergarten through high school the opportunity to develop programming skills.

In the UK, coding is on the national curriculum for primary and secondary school since 2014 as it is seen as a long-term solution to the "skills gap" between the number of technology jobs and the people qualified to fill them.

France also started to follow the flow. In high school, in addition to the final year specialization in informatics, optional courses are proposed to all other sections since 2016. The same year, basic informatics and algorithmic concepts have started to be taught in elementary school and real programming projects are made in high school [dS13].

In education, Belgium is still late in the informatics field [NJ13]. Most of the ICT and informatics courses focus more on how to use informatics tools instead of how to design and produce them.

Moreover, many websites propose programming languages on-line courses (Code.org, CodeAcademy,...) or learn-to-code applications (Tynker, Hopscotch, Scratch,...) and plenty of coding clubs have emerged such as Code Club, CoderDojo and much more. These clubs offer extracurricular free sessions to teach programming to children by helping them to use applications like Scratch or to develop their first website.

This work aims at encouraging teenagers to get into informatics by providing a new impelling tool to learn programming. The goal is to teach how to develop simple, but fully functional, 2D games that run on the browser. Video games are an exciting application of programming and is a real motivation to start learning it.

This tool tries to bypass the tedious steps of usual languages and frameworks learning curves by giving the opportunity to produce complete games with little knowledge on the subject. The other key concept of the work is to give an evolutive environment that allows children to progressively develop their skills as they gain experience.

The target audience is children from 12 to 18 years old, which means high school students.

1.2 Existing applications

Besides websites offering programming courses, a lot of interactive applications are available on-line to initiate kids to programming. This section briefly

reviews what those applications provide by presenting two of the most popular applications and their different approach to answer the problem. Their benefits and weaknesses are also discussed.

1.2.1 Scratch

1.2.1.1 What is scratch?

Scratch [1] is a block-based programming language for creating interactive stories, games and animations. The program conception is made by manipulating graphical elements instead of writing textual code. Scratch is intended to be a stepping stone for more advanced programming. For this reason, it is often used as a first approach to programming, and more particularly to promote computer science among kids aged from 8 to 16 years old.

1.2.1.2 Scratch functioning

Scratch uses multiple active objects called sprites to compose the user programs. Sprites are displayable objects, drawn from an editor or imported from the library or the user computer. Each sprite is programmable and have its own code.

Scratch is event-driven, users can define code to be executed in response to a particular event, e.g. *when this sprite clicked*. To code an interactive game, user usually places its logic code inside a *forever* loop attached to a start event to simulate the game loop.

To compose the code, the user can drag and drop code blocks from ten different categories:

- *Motion* - Move and rotate sprites, modify x and y position
- *Looks* - Control sprite appearance; attach speech or thought bubble, change of background, enlarge or shrink, transparency, shade
- *Sound* - Play sounds and notes
- *Pen* - Allow turtle graphics; control pen width, color, and shade
- *Data* - Create variables, assign value to variables; create lists
- *Events* - Event handlers, message manager

- *Control* - Control structures (repeat, forever, if-then, if-then-else, repeat-until), time manager, clone manager
- *Sensing* - Keyboard and mouse sensing, input manager, distance computing, collision manager
- *Operators* - Arithmetic, textual and boolean operators; random number generator
- *More blocks* - Custom procedure creation

Scratch works with 3 built-in types: text, number and boolean. The **Data** blocks allows to create and manipulate textual or numeric variables. A variable can be global or local to a sprite, and sprites can communicate through global variables. Another way of communication is using the messaging blocks in the **Events** category: when a sprite wants to trigger an event, it broadcasts a message that can be caught by any sprite.

Scratch allows to define procedures by creating custom blocks in the **More blocks** category, the user can specify the number and types of the procedure's parameters. However, Scratch does not provide a way to define functions with return value.

Scratch also allows to pause script execution at sprite level and offers the possibility to create sprite clones at runtime.

1.2.1.3 Advantages

Scratch is really easy to take in hand and its visual language removes the burden of learning a language syntax, what could be a brake for children when first programming. Scratch allows to create a wide variety of programs, from simple animations to interactive games. It also offers some powerful features such as its sprite collision detection.

Another strength of Scratch is its important internationalization. Both the user interface and the block instructions are translated in more than 50 languages.

1.2.1.4 Limitations

As it allows to create a large scale of different applications, from animations to interactive games; Scratch can be a bit difficult to approach for beginners. If the user has no precise idea of what type of program he wants to develop,

the code can quickly become messy and the result a bit precarious. This is a reason why Scratch learning is often supervised by teachers at school or by experienced programmers in coding clubs.

Scratch also suffers from a lack of features that could limit the development possibilities. As already stated, it misses the implementation of real functions with return value. Moreover, except for collision detection, Scratch does not expose a physics library. Managing sprite motion velocity and acceleration would be useful for certain type of program.

Scratch uses simple sprite images and the game rendering is not of high quality. Moreover, sprite motion is not smooth as the sprite position increases by steps. Even if Scratch provides block to *glide* sprites, the sprite motion is still jerky.

The fact that Scratch uses a block-based language can also be seen as a limitation for evolution towards a real programing language. However, the impact of block-based language on programming skills development is subject of debate.

A final disadvantage of Scratch implementation is that each sprite has its own code and there is no way to define logic applied for several sprite. This can result in code redundancy.

1.2.2 Code Combat

Code Combat[2] is a platform for students to learn programming while playing a game. The student has to solve role-playing based puzzle mini games by controlling the game hero with lines of code. For instance he would use control flow structures and call hero methods in order to move it towards the dungeon exit while checking if any monsters are in its sight and attacking them if so.

Code Combat proposes an evolving environment where new programming concepts are introduced as the user progresses through the different game levels, each level consisting on a new objective to be achieved with lines of code. The role-playing approach is enforced with the earnings of gems when solving a level and the possibility to spend them in order to buy new items that unlock programming functionalities.

1.2.2.1 Advantages

The learning is strongly gradual at two levels: on one hand, the user starts with a restricted API and will unlock new hero methods as he succeeds levels and buy new equipment; on the other hand, control flow statements and programming concepts (such as parameters, variables,...) are introduced every few levels. This avoids to overwhelm a beginner with a large documentation and language syntax and at the same time simulates its appetite to progress.

The game-based learning system also entertains the motivation and concentration of children. It involves challenges, encouraging students to master their skills, and avoids boredom by providing an enthralling storyline [Cam16][Alb15].

Students have the choice between four real-world programming languages to play the game: Python, JavaScript, CoffeeScript and Lua. This prepares the user to real programming experiences by teaching him to use strict syntax and code structure. Moreover, Code Combat provides useful UI features that ease the code production such as code completion, list of available methods, hints,...

1.2.2.2 Limitations

The Code Combat free version is only limited to basic programming concepts.

Even though the user interface is available in numerous languages, writing code is done in English. As it uses real-world programming languages, the language keywords obviously stay unchanged, but the exposed hero methods are not translated either.

Even if it incorporates a (paying) level for learning how to develop games, it doesn't provide a way to fully create a deployable game.

1.3 Problem-solving approach

This section describes the application approach to combine programming learning and game creation.

Game creation does not only involve logic coding and some steps of the development require complex configuration. The choices made in the application conception aim to reduce the amount of configuration and the number of repetitive and non educational tasks.

The life-blood of a game are its graphical elements, they have to be loaded, sometimes in different format depending on the needs, placed, resized, rotated, etc. Achieving those manipulations programmatically might not be an easy thing to do.

The application eases the assets management by providing an image gallery from which the user can load the desired graphic elements. It also provides a board initializer that gives the possibility to move and modify those elements with the mouse. Those graphical tools generates code internally that will be prepended to the user code afterwards. This results in an easier way to initialize the game stage and lets the user to focus more on logic.

Another difficulty for children in developing games might be the physics management as it could require advanced mathematical knowledge. The application is based on a game engine that already provides an API for physics management such as collision detection, gravity, velocity and much more. However, it could still be delicate to manipulate all these concepts on first approach with programming; and as the application purpose is to give the ability to develop functional games for real beginners, it provides a higher level of abstraction by exposing a core library implementing predefined behaviors. For instance, you can create a platformer hero that will automatically be subject to gravity, collide with platform and be controllable with the arrow controls.

The application intends to permit coding skills evolution and is then composed of different level of difficulty that provides different level of abstraction of the game engine, giving at the same time more freedom of implementation to the user. As the user progress in the difficulty levels, he also unlock new language functionalities.

In conclusion, this application is a game development platform adapted to teenagers that is composed of a graphical interface for tasks unrelated to game logic and that uses an adaptable language that fits the actual skills and knowledge of the user.

Chapter 2

Application design

This chapter deepens the discussion on the application's choices. It presents its main features and the designed programming language characteristics.

2.1 Application tools

This section presents the tools available to design games and explain how they can be used.

When the user connects to the web application, it reaches the landing page; this page's goal is to gather principal application's information. From the landing page, the user ends up on the playground page where he first has to choose which level of programming difficulty he wants to use. There are three different levels: *Beginner*, *Intermediate* and *Expert*; they all share the same functioning model but provide different level of abstraction and offer different access to some functionalities. Those differences are discussed through the next sections. After picking the desired level of difficulty, the user has access to the actual playground page where he will design, code, and run its game.

The playground is composed of two principal tools: the *board initializer* and the *code editor*; they are presented in the two following sections.

2.1.1 Board initializer

As briefly explained in section 1.3, the board initializer's purpose is to provide a graphical environment to build the game stage. It divides the window in two: on the left side is the part to add game elements, and on the right side is the board preview.

By manipulating inputs and buttons, the user can set a background, change the dimensions of the stage and add sprites on the board. Sprites are displayable elements that can interact with each other and respond to user inputs during the game; they are the main pieces that compose a game.

The user has access to a gallery of image from where he can select the desired images to represent its sprites. Each added sprite must have a unique name that follows some naming restriction, a validator checks if each name is unique and valid and warns the user otherwise. The sprite basic properties, namely its x and y position, width, height and orientation, are also displayed. Those properties can be modified by manipulating the sprite from the board preview.

For each sprite, the user choose its type from a list of predefined sprite types and can consult the corresponding documentation. It can also make some specific sprite initialization and for images that contains different costumes, he can select the default costume to be displayed. Predefined sprite types are discussed in section 2.2.5.2.

Figures A.1 and A.2 from Appendix A show an example of board initialization.

2.1.2 Code editor

The code editor is also divided into two parts. On the left are the actual editors and on the right is the game at its current state. In order to run the code and launch the game, the user has to save a board first from the board initializer. When saving a board, code for the game configuration and the sprite creation is generated; each sprite is added in a variable corresponding to the sprite name. Those variables can be manipulated in the different editors and more configuration can be defined.

There are 2 to 4 editors depending on the current difficulty level. Beginner level includes *Creation* and *Events* editors, intermediate introduces a third one: *Functions* editor; and Expert level adds the *Type* editor.

Once the user has placed its sprites on the board and made its configuration from the board initializer, he can press the *Save Board* button to generate initialization code. This auto-generated code ends up in the *Creation* editor in read-only mode but user can add more code initialization below.

The application uses an event-driven programming language and the user defines the code to trigger on particular events in the *Event* editor. The different types of events are discussed in section 2.2.3.2.

From the *Functions* editor, the user can define functions with parameters and return values. From the *Type* editor he defines its own sprite types.

To see the result of the current code, the user can press the *Run code* button. If the code contains any syntax or semantic errors, he is notified and the game is not launched. When executing an error free code, the game is launched in the right part of the window and the user can test it.

Figures A.3, A.4 and A.5 from Appendix A show an example of code edition and execution.

2.1.3 Other features

2.1.3.1 Internationalization

The whole application is available in English and in French. The programming language is also translatable; grammar keywords, core library global variables and predefined sprite types and their methods and fields have different naming depending on the current language.

2.1.3.2 Documentation

A complete documentation is described in the *Documentation* section of the web application. This documentation describes how to use the application and presents the programming language syntax. It explains the differences between each level of difficulty and also describes the core library global variables and the predefined sprite types.

The predefined sprite types are also briefly described on the board initializer when the user selects the type for a sprite.

2.1.3.3 Error notification system

An error notification system is implemented to prevent some wrong manipulation of the tools by the user. For instance, it avoids saving a board if there are sprite name conflicts and it prevent running a game without initializing the board first.

2.1.3.4 Code completion and highlighting

The editors provide basic code completion based on local variables and language keywords. They also handle brace completion and automatic indentation.

The syntax highlighting is also configured to suit the application language.

2.2 Programming language

The language is highly inspired by JavaScript. It follows most of its syntactic rules but presents some differences.

It uses different keywords, for instance the *repeat* keyword eases the implementation of basic loop control flow; the conditional *if* statement is also a bit different as it comes along with a *then* keyword.

The syntactic processing is also stricter. The syntax forces semicolons at the end of each instruction while they can be omitted in JavaScript; and the control flow bodies are always surrounded by curved brackets even if they are made of a single instruction. This implies a more rigorous syntax but involves less confusion for beginner.

The language implementation is based on the event-driven paradigm. The user defines several events by specifying their type, their condition and the action to trigger when they are fired. The reason why it uses this paradigm is to add a level of abstraction on game development by hiding the game loop from the code.

It also adopts an object-oriented approach by providing class-like sprite types with fields and methods.

The full description of the language is detailed in the next sections.

2.2.1 Lexical structure

2.2.1.1 Whitespace and comments

Whitespace characters are treated the same way as in JavaScript, they are ignored in the parsing process. The language defines single-line (//) and multiline (/* */) comments that are also ignored by the parser.

2.2.1.2 Keywords

The list of reserved keyword is given in Listing 2.1. Keywords are case-sensitive.

```
and          do          else          equals
function     forever     greater_than  if
let          lower_than  not           once
or           repeat      repeat_until  return
```

```
then            when            while
```

Listing 2.1: Language keywords

2.2.2 Syntax

A program consists in a set of `instructions` and `event`, `function` and `type` definitions. The syntax of the language is described by the following grammar using the extended Backus–Naur form (EBNF).

```
program = p { p };

p = instruction
  | event
  | function
  | type
;

instruction = expr ";"
            | assignment ";"
            | declaration ";"
            | control_flow
            | return ";"
;

event = ("when" | "while" | "once") "(" expr ")" "do" block;

function = "function" identifier "(" params ")" block;

expr = text
     | number
     | boolean
     | identifier { "." identifier } ["(" args ")"]
     | expr ("+" | "-") expr
     | expr ("*" | "/") expr
     | expr ("and" | "or") expr
     | "not" expr
     | expr ("greater" | "lower" | "equals") expr
     | "(" expr ")"
;

assignment = identifier "=" expr;
```

```
declaration = "let" identifier [ "=" expr ];

control_flow =
       "if" "(" expr ")" "then" block [ "else" block ]
     | "repeat" "(" expr ")" block
     | "forever" block
;

return = "return" expr;

block = "{" { instruction } "}";

args = [ expr { "," expr } ];

params = [ identifier { "," identifier } ];

identifier = letter { letter | digit | "_" };

text = ( '"' { all_characters - '"' } '"' | "'" {
   all_characters - "'" } "'");

number = [ "-" ] positive_integer ["." positive_integer];

boolean = "true" | "false";

lowercase_letter = "a" | "b" | "c" | "d" | "e" | "f" | "g"
                 | "h" | "i" | "j" | "k" | "l" | "m" | "n"
                 | "o" | "p" | "q" | "r" | "s" | "t" | "u"
                 | "v" | "w" | "x" | "y" | "z" ;

uppercase_letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                 | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                 | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                 | "V" | "W" | "X" | "Y" | "Z" ;

letter = lowercase_letter | uppercase_letter;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
   | "9";

positive_integer = digit { digit };
```

```
all_characters = ? all visible characters ? ;
```

Listing 2.2: Language syntax in EBNF form

2.2.3 Semantics

2.2.3.1 Instructions

An instruction can have four different semantic goals. It can be an expression, an assignment, a variable declaration or a control flow statement. Except for control flow, an instruction is always ended with a semicolon.

Expression An expression is either a number, text or boolean value, a property access, a method call or an operation. Built-in types and the language type system are discussed in section 2.2.4.

Assignment An assignment assigns the evaluation value of the right hand side expression to the variable identified by the left hand side identifier. The type of the expression does not need to match the type of the variable. If they don't match, the type of the variable will dynamically change.

Declaration A variable is declared with the keyword *let* and the variable name should be unique. A declaration can be combined with an assignment in which case it forms an initialization.

Control flow Control flow statements modifies the order in which the instruction are executed. There are 5 control flow statements; 2 conditional branches: *if-then* and *if-then-else*, and 3 loop control: *repeat*, *repeat_until* and forever. *repeat* condition should evaluate to a positive integer number value and the loop body is executed accordingly to that value. *repeat_until* condition should evaluate to a boolean value and the loop body is executed while the condition is true.

2.2.3.2 Events

An event is composed of two parts: the condition and the action; and can be of 3 types: *when*, *while* and *once*.

The condition is an expression continuously evaluated and its value determine whether the action is triggered or not depending on the event type. The action is a block of instructions that are executed sequentially when triggered.

when An event of type *when* is triggered when the value of its condition switch from false to true. The action is only executed once while it is true and can't be triggered again before the condition value switches back to false.

while An event of type *while* continuously executes the action code when its condition is true.

once An event of type *once* behaves like a when event but the condition is never evaluated again after the first time the event is triggered.

2.2.3.3 Function

Functions definition is similar to JavaScript. Once a function is defined, it can be called anywhere in the code.

Functions definition are only introduced at the intermediate level of difficulty.

2.2.4 Type system

The language is composed of 3 built-in types: text, number and boolean. The text type strictly correspond to strings in JavaScript. Number types also strictly correspond to the numbers in JavaScript, they can be positive, negative, integer or decimal numbers. The boolean type can only take two values: *true* or *false*.

Types are not statically defined at variable declaration but inferred from the assigned value. As JavaScript, the language uses coercion to deal with values whose type doesn't fit the expecting type of an operation. This means it will implicitly cast the value to the correct type.

2.2.5 Core library

The application language comes with a standard library that includes global variables and predefined sprite types. This section draws up the list of these objects and describes their role. The complete library documentation can be found in the Documentation section of the application.

2.2.5.1 Globals

Game This object provide game configuration functions to resize, pause, resume the running game. It also handles background manipulation and provides function to create and add sprites to the game stage.

Keyboard This object monitors keyboard inputs and handles key press events. Each of its properties corresponds to a keyboard key and each key is composed of two boolean properties: *isDown* and *isUp* that expresses to the current key state.

Mouse This object monitors mouse left click and cursor position. Works also with any other pointer device.

Math This object has properties and methods for mathematical constants and functions.

Time Provides static properties for different unit of time, namely: a quarter of a second, half a second, a second and a minute. Also provides the *wait* method that allows to wait a delay before executing a function. The delay to wait and the function to execute are both given in parameters.

Direction Provides static properties corresponding to 8 angle orientation: up, down, left, right, north-east, north-west, south-east, south-west. This eases motion methods manipulation by abstracting the arbitrary angle values corresponding to these directions.

Color The object contains a set of 140 static fields corresponding to the 140 color names supported by all browsers.

2.2.5.2 Sprite types

This section explains the purpose of exposing predefined sprite types in the core library a describes the most important ones. The list of currently implemented sprite types is drawn up in Appendix **??**. This list does not mean to be exhaustive and is intended to be extended.

A sprite type defines a sprite with predefined logic adapted for a particular game role. It provides configurable properties, high-level methods to perform actions related to its role; and read-only boolean variables whose values becomes true on particular event. Those variables will be called *event variables* in the rest of the text.

Sprite This is the basic sprite type that defines properties and methods shared by every sprite type. When the user creates a new sprite type, it can use those properties and fields to implement its custom logic.

Hero This is the most advanced implementation of a sprite type. It defines a controllable sprite for platform games. A hero is subject to gravity and the player can control its movements with the arrow keys; it can jump and move to the left and to the right. It automatically collides with instances of sprite type *Platform* and can jump only if touching one. Its properties allows to change its motion speed, the force with which it jumps and the gravity force applied to him. It can equip a *Weapon* and shoot bullets with its *fire()* method.

Platform A platform is not subject to gravity and don't move on collision. It aims at holding sprites of type *Hero* or *Enemy*.

Object Sprites of *Object* type contains a *collected* event variable that evaluates to true when the object overlaps with a controllable sprite type.

Weapon A *Weapon* is a particular type of sprite, it is not visible on the board but can be equipped by compatible sprite types. Once equipped it defines the bullets appearance and properties of the weapon. *fireRate*, *bulletSpeed* and *fireAngle* are configurable properties.

Chapter 3

Application architecture and technologies

This chapter discusses the architecture details.

As the created games are conceived to be run in the browser, the application is developed as a web application. This allows to directly test and play the game while it is on development, without having to set up a local server and switch between a desktop application and the browser.

3.1 Languages and frameworks

In this section, the web technologies used to develop the application are described and the choices are justified.

3.1.1 Client side

3.1.1.1 Aurelia

Considering the importance of the front-end in this application, choosing a complete JavaScript client framework appeared to be relevant.

Aurelia[3] is a descendant of the popular AngularJS framework but offers some advantages. The key advantage is that Aurelia is unobtrusive; after configuration, developing a web application in Aurelia is basically writing in normal JavaScript and the templates look like absolutely sane HTML. It shortens the learning curve and is a benefit for maintenance. Moreover, Aurelia is more standard compliant than Angular.

Aurelia supports the last versions of javascript: ES5, ES 2015, ES 2016 and also supports TypeScript. Aurelia takes care of the transpilation and polyfills so the application can run on today's browsers.

Development in Aurelia is based on the *Model - View - View-Model* design pattern. User interface elements are composed of an HTML view and a JavaScript view-model. The view is rendered into the DOM and the view-model provides data and behavior logic to the view. Aurelia's powerful data-binding automatically synchronizes them together, allowing changes in the data to be reflected in the view and vice versa.

The Aurelia templating system lets the developer create custom HTML elements, custom attributes and control template generation with conditional or repeated DOM element creation.

Aurelia is used along with JSPM package manager and its associated SystemJS module loader. Those two tools make it really simple to install and import any JavaScript library from registries such as npm and GitHub. JSPM handles libraries dependency problems transparently to the developer.

Such technologies allow a highly modular JavaScript development with creation of components and services that can be reused and injected into each other. This results in a safer development process and helps to produce a more maintainable code.

3.1.1.2 TypeScript

Typescript[4] is a superset of JavaScript designed to improve and secure the production of JavaScript code. Typescript adds an optional static typing to JavaScript and provides additional language features such as interfaces and classes. It gives access to traditional object-oriented programming with Java-like class inheritance, access level modifiers on class members, decorators, and much more.

TypeScript supports the EcmaScript 6 specification that comes with many useful features such as block-scoped variables without hoisting, arrow functions and a more intuitive handling of current object context. This results in a cleaner and safer code production.

TypeScript supports definition files (.d.ts) that contains type definition of existing libraries written in vanilla JavaScript. This enables the use of values defined in the files as if they were statically typed, easing and securing the development when using external libraries.

3.1.1.3 Bootstrap

Bootstrap helps the design of the user interface and makes it responsive, it also provides useful JavaScript components.

In addition to the bootstrap classes, a traditional Cascading Style Sheet for custom styling is used. Some style rules are also directly defined in HTML tags' attributes in the views.

3.1.1.4 Gulp

During development, automated tasks are executed by the Gulp task manager. Gulp manages the build of the application and prepares client files for production. It executes the following tasks: transpile TypeScript into JavaScript, minify HTML and JavaScript, generate documentation from JsDoc and bundle the files.

3.1.1.5 Visual Studio Code

Visual Studio Code is a free powerful and lightweight code editor particularly adapted for the used technologies. It provides smart code completions with IntelliSense and is compatible with TypeScript definition files. VS Code includes enriched built-in support for Node.js development with JavaScript and TypeScript. It comes with a large set of extensions that enhance code production and ease web development.

3.1.1.6 Fabric.js

Fabric.js[5] is a powerful and simple Javascript HTML5 canvas library. It is used to implements the board initializer preview by adding and manipulating images on the canvas.

3.1.1.7 Ace editor

Ace[6] is an embeddable code editor written in JavaScript. It comes with a set of useful features: syntax highlighting creation, automatic indent and outdent, code completion,...

3.1.1.8 Phaser

The game engine was probably the most important choice in the architecture as it guides the way the application language and the core library are imple-

mented. Phaser is ranked 8th in the "Best 2D games engine" and is the first framework of the list using JavaScript [pha].

Phaser[7] is an open source framework for creating 2D browser games in JavaScript. It uses both canvas and WebGL renderer internally by means of the Pixi.js library. Phaser is a full-fledged game engine, it handles many features beyond rendering such as physics, inputs (keyboard, mouse), camera scrolling, collision, state management, spritesheets and animations, game loop, and more.

Phaser is shipped with support for 3 different physics systems: *Arcade*, *Ninja* annd *P2*.

Arcade Arcade Physics is for high-speed AABB collision only; it means the sprite hitboxes are non rotatable rectangles. This is cheap and fast to compute, but AABB doesn't guarantee that there really is a collision as it may be a completely transparent area overlapping.

Ninja Ninja Physics allows for complex tiles and slopes and handles rotations. This is a more flexible and accurate physics model but it's probably slower. Ninja physics has however a bigger flaw, it does not implement some basic features that can be found in the arcade physics.

P2 P2 is a full-body physics system, with constraints, springs, polygon support and more. It is allows real collision detection but require to define the polygon hitbox. As the application sprite creation is automated with the board initializer, complex polygon hitbox is not relevant as it would require to add a complex feature to define it. Moreover, constraints and springs physics might be superfluous for beginner.

From the above reflexion, the arcade physics has been chosen as the physics that runs the application core library.

Phaser handles events and event dispatching through its *Signal* manager. It provides methods to listen to signals by binding callback to it. Phaser sprites comes with a lot of predefined handy signals but the Signal manager also allows to create its own signals and dispatch them when desired. The Phaser *Signal* system is at the core of the application events management.

3.1.2 Server side

3.1.2.1 Node.js

The overall application development is based on the asynchronous and event-driven JavaScript runtime environment Node.js[8]. Node contains a built-in library to create http server and is served with a handy package manager NPM that provides easy installation of dependencies.

Node.js also provides the advantage to use JavaScript on server-side; this allows the web application development to unify around the same programming language, rather than rely on a different language for writing server side scripts.

3.1.2.2 Express.js

The Express.js[9] framework is used to create the web server based on Node. Express handles client requests with its routing system. It both serves client-side files and exposes the server RESTful API.

3.1.2.3 MongoDB

Application data are stored on a MongoDB[10] server. MongoDB is a cross-platform document-oriented database management system. It is classified as NoSQL database program and uses JSON-like documents to store data.

The Mongoose module is used to manipulate the database. It uses schemas to model data that allows to define types and structure data as if it was using SQL tables. Mongoose also handles database connections and data insertions with data validation and provides a useful query builder.

3.1.2.4 Jison

Jison[11] is a JavaScript parser generator. It is based on the parser generator Bison and includes its own lexical analyzer modeled after Flex.

The grammar is defined in a .jison file with a syntax similar to the one used with Bison.

3.2 Front-end

This section presents the main components of the application front-end.

Figure 3.1 presents the client directory structure.

Figure 3.1: Client directory structure

The following list explains the role of each subdirectory:

- **assets**: **media** contains the client images used for the UI design, **locales** regroups the localization files for i18n, **doc** contains the generated documentation in JSON format and **styles** includes all the custom style sheets.
- **build**: contains all the gulp tasks that automates development and deployment
- **dist**: contains the transpiled source for testing the application under development
- **export**: contains the bundled and minified transpiled source with the dependencies ready for deployment
- **jspm_packages**: contains the source of the installed dependencies
- **src**: contains the actual application source code
- **typings**: contains the typescript declaration files of the imported libraries

Figure 3.2 presents the client directory structure.

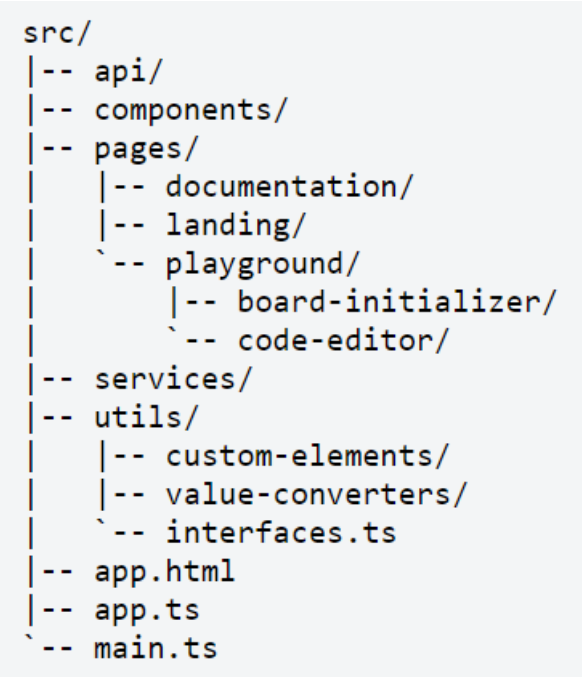

Figure 3.2: Source directory structure

The following list explains the role of each subdirectory:

- **api**: contains the application core library files; definition of global variables and predefined sprite types.
- **components**: common application components, such as game-container, language-switcher,...
- **pages**: implementation of the routed pages
- **services**: singleton service classes such as **backend-service** for back-end REST requests, **board-canvas** for Fabric.js canvas manipulation, and more.
- **utils**: contains reusable and configurable custom HTML elements and value converters; *interfaces.ts* contains typescript interfaces of important object, interfaces role is to enable "duck typing" during development

3.3 Back-end

This section presents the main components of the application back-end.

Figure 3.3 presents the server directory structure.

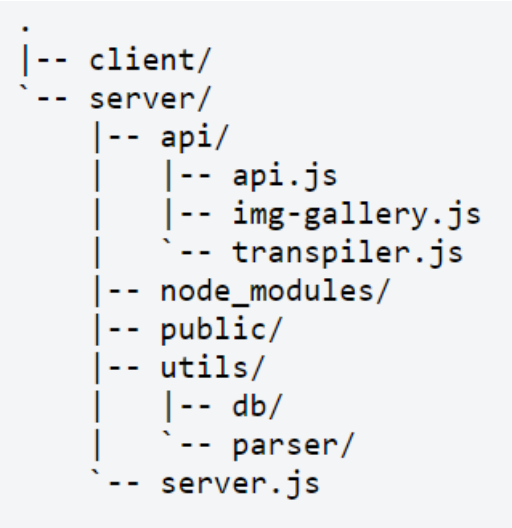

Figure 3.3: Server directory structure

The following list explains the role of each subdirectory:

- **api**: contains files related to the management of the RESTful API routes; `api.js` exposes the API, `img-gallery.js` and `transpiler.js` handle the requests for image fetching and code transpilation and construct the resulting json responses
- **node_modules**: contains the installed dependencies sources
- **public**: contains the sprite images and spritesheets
- **utils**: contains services; **db** contains database utility functions and **parser** contains scripts that handle the parser generation and provides utility functions to transpile code

Chapter 4

Implementation

This chapter discusses the application implementation details.

4.1 Front-end

4.1.1 Board initializer

The board initializer view implements navigation tabs to arrange initialization tools by category. Tabs separate the categories in different panes and each pane is viewable one at a time.

There are two categories: *Background and Dimensions* and *Sprites*.

From the first pane, game size can be modified and a background can be selected from a background gallery. The user can also select a type for the background among 3 possibilities: *Fixed*, *Camera* and *Scrollable*.

A *fixed* background scales to fit the game size and has no particular features.

With a *camera* background, only a portion of the game board is visible at any time. This portion is defined by the camera size and position. The camera can move around the game stage and can be set to focus or follow a particular sprite.

A *scrollable* background transforms itself into a repeating texture that can be scroll in any direction. The scrolling speed can also be controlled.

Sprites creation and management are discussed in the next section.

4.1.1.1 Sprite list

The sprites tab implements the tools to create and manage sprites. As it requires a lot of logic, this is implemented in a separated component: *sprite-list*.

This component consists on a button for adding sprite to the list and to the board preview. Clicking that button pops up an image gallery from where the user can select the image he wants to use for its sprite. A sprite can be selected either from the board preview or from the list. An example of a sprite list is shown in Figure 4.1

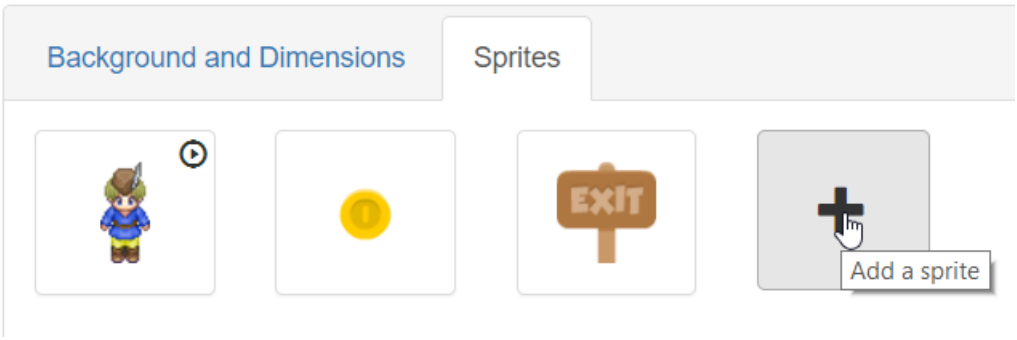

Figure 4.1: Sprite list example

When a sprite is selected, some action can be applied to the sprite and its information the displayed.

The sprite can be removed from the board preview and the list by pressing the *Delete Sprite* button.

An editable input is filled with a default name for the sprite, this will be used to name the variable corresponding to that sprite in the code. That name has to fulfill some variable naming convention, it has to start with a lower letter, can't contain special character except underscore, must not be empty and should be unique among all the sprites.

The plugin *aurelia-validation* is used to control that sprite names conform to those conditions. It detects errors and notifies them to the user in real time while he is editing the sprite name. The validator uses regular expression to control these requirements.

Figure 4.2 illustrates bad name validation errors and Listing 4.1 shows the validation plugin implementation.

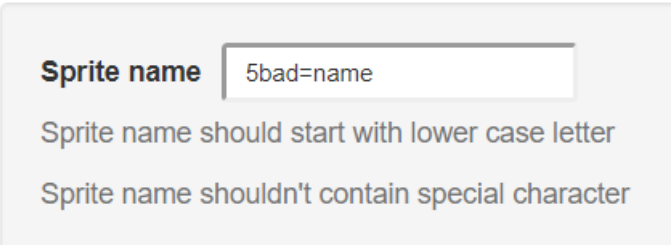

Figure 4.2: Bad name validation error

```
1  ValidationRules.ensure('name')
2  .required().withMessage('board-init.errorReq')
3  .matches(/^[a-z].*$/).withMessage('board-init.errorLow')
4  .matches(/^\w*$/).withMessage('board-init.errorSpec')
5  .satisfies((name: string, data: any) => {
6    return this.sprites.map(sprite => {
7      if (sprite !== newSprite)
8        return sprite.data.name;
9    }).indexOf(name) == -1;
10 }).withMessage('board-init.errorUniq')
11 .on(data);
```

Listing 4.1: Sprite name validation

The sprite information pane contains several tabs below the name input and delete button.

Type tab allows for picking the sprite type; it also also describes the currently selected type and exposes its properties and methods.

Properties tabs lists the sprite position, dimension and orientation properties.

Options tab sets allows to change default value of the sprite properties.

Costumes tab allows to pick a default costume to use when the game starts. This tab is only enabled for spritesheets. Such sprite, containing multiple costume, is identifiable by a play icon in the top right corner of its miniature.

4.1.1.2 Image gallery

The Aurelia *dialog-service* plugin is used to implement the different image galleries. The dialog service supports the use of dynamic content by calling custom view/view-model pair to construct the dialog and passing to it data from the parent view-model.

Both background and sprite image gallery use the same view/view-model pair: *image-gallery*. Depending on the model passed to *image-gallery*, it will query the database with different requests.

For the background gallery, this results in a set of background images while the sprite gallery is composed of several set of sprite images gathered in collapsible panels. Figure 4.3 shows the sprite gallery.

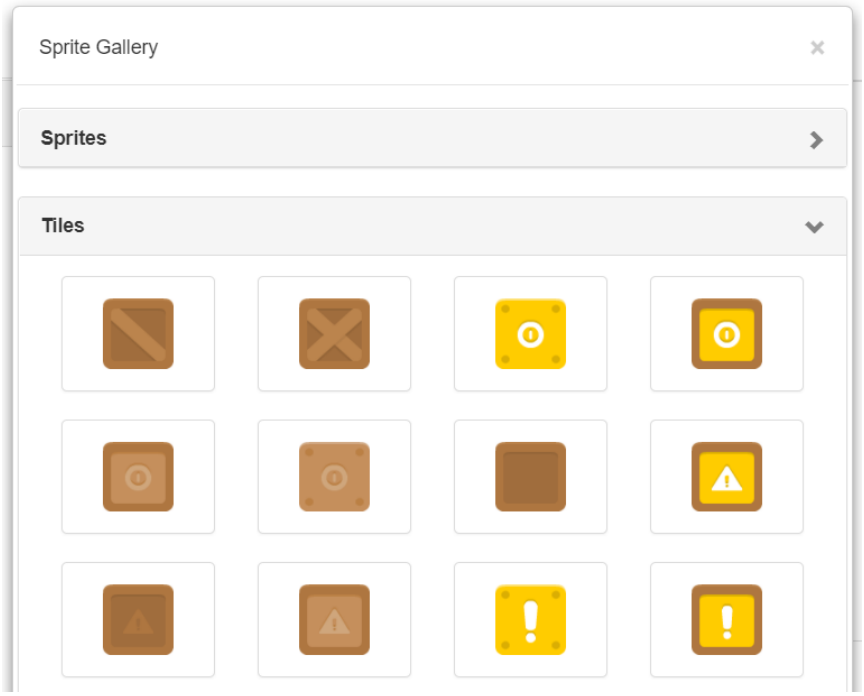

Figure 4.3: Sprite gallery

To pick an image from the gallery, the user has the possibility to double click on the desired item or to select it and then press the Ok button. In both cases, it triggers a callback with a response parameter containing the selected image information. This callback is then responsible for the image manipulation. Listing 4.2 illustrates the dialog service use.

```
1 private openBackgroundGallery() {
2   let model = {
3     title: 'board-init.bgnd-gallery-title',
4     sections: [{name: 'img-gallery.background', tag: 'Backgrounds'}]
5   }
6   this.dialogService.open({ viewModel: ImageGallery, model: model })
7                     .whenClosed(response => {
```

```
8       if (!response.wasCancelled && response.output != undefined) {
9         this.board.setBackground(response.output);
10        this.background = response.output;
11      }
12    });
13  }
```

Listing 4.2: Dialog service used for background gallery

4.1.1.3 Board preview

The board preview is implemented in the *BoardCanvas* class as a service. The class inherits from *fabric.Canvas* and initializes a fabric instance on the canvas element with id "board". *BoardCanvas* implements high-level public methods to perform canvas manipulation: resize canvas, add a background, add sprite or spritesheet, remove object,...

Adding an object to the canvas automatically places it in the center and selects it. Selection of an object enables border controls for resize and rotation.

Figure 4.4 shows a selected sprite with border controls on the board preview.

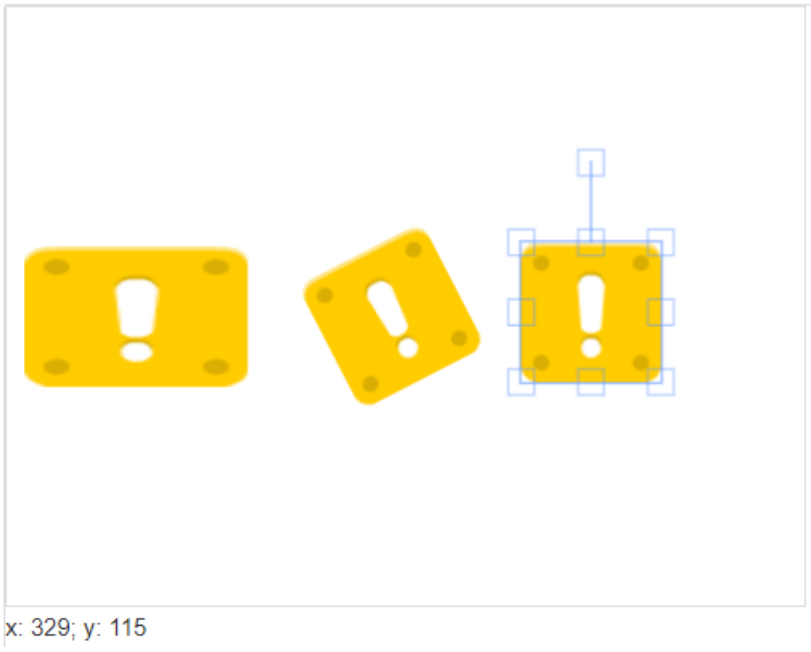

Figure 4.4: Board preview sprite manipulation

4.1.1.4 Board saving

In order to generate code from the board initialization, the user have to save the current state of the board by clicking on the *Save Board* button.

Saving the board gathers all information from the board initialization and use the Aurelia *Event Aggregator* to pass it in a message to the code editor.

The signature of the exchanged message is defined in the service file *messages.ts* by the *BoardInfo* class. It contains the game dimensions, the background information and an array of *SpriteInfo* objects containing sprite information.

4.1.2 Code editor

The ace editor module gives editors configuration possibilities such as indentation size, editor theme and syntax highlighting.

4.1.2.1 Code initialization generation

The *Editor* class subscribes to *BoardInfo* messages and defines a callback to execute when the event is triggered.

The board information is parsed using the *BoardInfoParser* service and corresponding code is generated. The *BoardInfoParser* generates two types of code from the board information.

The first one is directly Phaser code corresponding to the loading of sprite images and spritesheets. This code is not exposed to the user. The parser prevent image loading duplication by maintaining a list of key during the process. The resulting code is stored in a string and will be called in the *preload()* function of the game.

The second generated code consists of sprite creation and initialization. This code uses the application library and is inserted to the *Creation* editor. This code is set in readonly mode to prevent the user to induce incoherence between the code and the board initializer. However, the user can add more code initialization in the *Creation* editor below the generated code.

4.1.2.2 Code execution

The user can execute its code and see the results by clicking the *Run code* button. When he does so, if the application language was not set to english, its code is translated in english. The code translator is implemented using regular expressions generated from the map of keywords, variables and methods translations.

The resulting code is sent to the server to be properly parsed and transpiled. Server transpilation is explained in section 4.2.4.

The result of the transpilation contains instructions and function definitions and event information. The event information is processed to be transformed in phaser code. The resulting codes are surrounded by a *try...catch* clause in order to handle runtime errors.

Once code generation is complete, it is sent to the *game-container* component via *CodeUpdate* to launch the game.

4.1.2.3 Event precessing

The events are implemented using Phaser signals. For each event a new signal is created. The signal creation takes a callback that is triggered when the signal is dispatched, the callback contains the event action code.

The event condition is tested in the game loop and the event is dispatched whenever the condition evaluates to true. In order to implement the *when* event type, the signal is set inactive as soon as it is dispatched and get back to active only when the condition evaluates to false. This allows to execute the action code only once each time the event occurs.

4.1.3 Game container

The game container component subscribes to *CodeUpdate* messages. This message contains the generated phaser code from the code editor.

It contains 3 strings corresponding to the 3 phaser methods required to run a game: *preload()*, *create()* and *update()*. A new *Game* instance is created upon reception of a CodeUpdate message.

Game extends the *Phaser.Game* class and creates the actual phaser game instance. The 3 code strings are passed to the constructor of the Game class and are transformed into function via the JavaScript *Function* method. The global variables of the core library are passed to those function so they are in the same scope of the generated code. To actually launch the game, those functions are passed to an instance of the *Chapter* class that inherits from *Phaser.State*. The chapter is finally started and the game is launched and ready to be played.

4.1.4 Core library

The core library implements an abstraction of the Phaser framework. Some Phaser features are simplified, others are combined to implement new func-

tionalities and the remaining is not exposed. Some library functionalities, particularly some sprite methods, are inspired from the Scratch block instructions.

4.1.4.1 Phaser functionalities

This sections explains how to use some Phaser features in order to help understanding the core library implementation.

State A Phaser game is composed of states. Each state implements 3 fundamental functions: *preload()*, *create* and *update()*.

The *preload* method contains the assets loading. It associates an image or spritesheet to a key that can be used to create a sprite object later.

The *create* function is where the sprite are created and placed on the board. Sprite properties can be initialized and a physics system can be instantiated.

The *update* function implements the game loop. It is called 60 times per seconds and all the game logic is implemented there.

Collision Phaser arcade physics does not provide a way to set collision in the *create()* function. Collision between two sprites has to be checked at each game loop iteration in the *update()* function with the *Physics.Arcade.collide()* function. Both sprites are passed as parameters to the function and an optional function callback can be passed as third parameter. This callback has two parameters corresponding to the two sprites, and is executed when the collision happens.

4.1.4.2 Sprite types

The predefined sprite types are classes that inherit from the Phaser.Sprite class. The sprite types implements high-level methods by manipulating parent class function and properties. They also overrides the Phaser.Sprite *update()* function, the logic implemented in this function is automatically added to the global game loop.

The collision management of sprite types is also managed in the overridden update function. Listing 4.3 shows the implementation of automatic collision between Hero and Platform sprites. At each game loop, the Hero-type sprite traverses all the sprites present in the game and checks their type. If it encounters a Platform-type sprite, it then calls the collision function. The code in Listing 4.3 also shows how an event variable automatically sets to true when an event occurs. All the event variables are always reset to false at the beginning of the update function.

```
1  // Set collision with platforms
2  let children = this.game.world.children;
3  for (let i = 0; i < children.length; i++) {
4    let child = children[i];
5    if (child instanceof Platform) {
6      this.game.physics.arcade.collide(this, child);
7    }
8    // Notify when touch enemy
9    if (child instanceof Enemy) {
10     this.game.physics.arcade.overlap(this, child, () =>
           this.touchEnemy = true);
11   }
12 }
```

Listing 4.3: Sprite type collision and event variables

4.1.5 Other features

4.1.5.1 Internationalization

Aurelia provides a plugin for internationalization support: *aurelia-i18n*. The plugin is configured in *main.ts*.

English is set to be the default and fallback language, this means the application starts in English and if a translation is missing in a language, the English translation is used instead.

The current available languages are English and French.

One JSON file is associated to each language and contains the key-value pairs for translation. Every translatable value is represented by its key and its translation is automatically updated from the view by applying the *translation binding behavior*.

The user can change the current application language at any time using the *language-switcher* located in the nav-bar. The language switcher implementation is based on the Bootstrap *select-picker* component and triggers a signal in the whole application when a different language is selected. The language-switcher is shown in Figure 4.5.

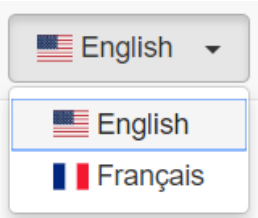

Figure 4.5: Language switcher

4.1.5.2 Documentation

The documentation generation is automated from the source code. Each global and sprite type properties and methods are documented in the source with JSDoc comments.

On application build, a gulp task reads the JSDoc in the typescript source files. This task is composed of two steps. The first step is a slightly modified version of the *comment-parser* module. For each multi-line comment, it reads the information of each tag and the global comment description. The result of this step is stored in a JSON object. The second step consists in parsing this JSON object and interpret the meaning of each tag to create a meaningful object describing each global and sprite type. This structured object is then injected in model-views that require to display documentation.

4.2 Back-end

4.2.1 Server creation and database connection

The server is launched by executing the *server.js* script with Node.js. This script instantiates an Express application, configures the Express middleware functions, connects to the database and then creates the server.

4.2.2 Routing

Client files are served by an Express application from the `export` folder in case of application deployment and from the `dist` folder during development. Image files contained in the `public` folder are also served by Express as static files.

The RESTful API is exposed by the means of an Express Router instance that creates a modular and mountable route handler. This Router is mounted

during the Express middleware configuration at server start-up and provides the 3 following URLs:

- **/api/img-gallery** - GET - List the different gallery categories names
- **/api/img-gallery/:id** - GET - Fetch the information of all images of the category *id*
- **/api/transpiler** - POST - Transpile the code given in the post body

4.2.3 Image storage

The server stores the images files used for sprite creation and background instantiation in its filesystem in *.png* or *.jpg* format. They are stored in the *public* directory and sorted by category in different subfolders.

In order to download the images, the client needs to be informed of their location on the server. This information is available in the database in the *imggalleries* collection.

The database is populated with the images information at server start-up; once the connection to the database is established, a procedure reads the server *public* directory and inserts for each images found its information in the collection.

An image is described in the database by a unique name and its url location. In the case of a spritesheet image, it also contains spritesheet information required to access the individual sprites separately. Image information are stored in the *imggalleries* collection by category, each entry of the collection consists of the category name and an array of images information.

The schema of an *imaggalleries* entry is given in Listing 4.4 using Mongoose schema definitions of the file `utils/db/schema.js`.

```
1  var mongoose = require('mongoose');
2
3  var imageSchema = mongoose.Schema({
4      name: String,
5      url: String,
6      spritesheet: {
7          type: {
8              sheetUrl: String,
9              spriteWidth: Number,
10             spriteHeight: Number,
11             horizontalNbr: Number,
12             verticalNbr: Number,
13             spriteNbr: Number,
```

```
14              defaultSpriteNo: Number
15          },
16          default: null
17      }
18  });
19
20  var imgGallerySchema = mongoose.Schema({
21      sectionName: String,
22      images: [imageSchema]
23  });
24  exports.ImgGallery = mongoose.model('ImgGallery', imgGallerySchema);
```

Listing 4.4: Typed declaration syntax suggestion

A spritesheet holds two different URLs, one is pointing to an individual image representing its default sprite, that is used for spritesheet miniature and the sheetUrl is pointing to the actual sheet containing the different sprites.

4.2.4 Transpiler

The grammar syntax is implemented in BNF form in a *.jison* file. When the server API receives a request for the transpiler, the code in the POST method body is parsed by the generated parser. The result of the transpilation is an object containing 4 fields. The first one is a string containing the instructions transpiled JavaScript. The second one is an array of event information. An event is described by 3 strings: its type, the expression of its condition and the instruction of its body. The last two fields are arrays containing the functions and types information.

Chapter 5

Conclusion

5.1 Main achievements

The contribution of this work offers a new approach to teach programming. It proposes a textual programming language intended to develop 2D games that run in the browser and graphical tools to handle game initialization.

The graphical tools, reduce the amount of tedious tasks required to make a deployable game.

The language is event-driven and allows to define intuitive event handlers. The user game logic thus no longer resides in a infinite loop as in traditional game development process.

It also exposes a core library that provides high-level types giving the opportunity to develop functioning game with relatively straightforward logic.

The language is evolutive and introduces more programming concepts as the level of difficulty increases. The library also implements different level of abstraction that allows the user to choose the trade-off between programming ease and game conception freedom.

5.2 Improvements & future works

The application at its current state does not pretend to be complete. Many more features can be added to the existing tools and the existing ones could be subject to improvements.

This section describes features that could be added or improved in a future work and explain some of the current application defects.

5.2.1 Type system

The language type system could be improved by providing typed declaration. This would allow for static type checking and result in more secure variable assignments and parameters passing in methods calls.

An example of a possible syntax for type declaration is given in Listing 5.1.

```
1 let bar: number = 42;
2
3 function foo(arg1: text, arg2: boolean) {
4   return arg1 + arg2;
5 }
```

Listing 5.1: Typed declaration syntax suggestion

5.2.2 Core library

As already discussed, the sprite types set does not pretend to be exhaustive and many more types could be defined in the library.

The current work behind the predefined sprite type implementation is to find reusable solution to abstract phaser logic in predefined classes. The concept implemented in the existing types, such as collision detection and event variables, might be exploited to define more sprite types.

Global variables can also be improved in a future work. For instance, the Mouse global could also handle the middle and right mouse buttons.

A *Debug* global variable could be introduced to provide code debugging functionalities.

5.2.3 Language features

5.2.3.1 Control flow

The language defines a reduced set of control flow statements with the intention of not overwhelming the beginner with too many syntax features to learn. However, the set of statements could be progressively extended as the user change the difficulty level.

Useful control flow statements that are not currently included in the language are break, continue, while-loop and switch-case.

5.2.3.2 Sprite type definition

Although presented in previous sections, the sprite type definition has been removed from the current version of the application as it was not working properly. A further reflection on the design and the implementation of this functionality is required in order to make it fully functional.

5.2.4 Editors autocomplete

The current code completion only predicts language keywords and local variables. Autocomplete should be based on the manipulated variable type and provide the set of its properties and method in order for it to become operationally meaningful.

A powerful tool to achieve this behavior has been considered during the application development. This tool is Tern.js, a stand-alone code-analysis engine for JavaScript. It provides many functionalities including autocompletion on variables and properties and function argument hints.

Unfortunately, the tern plugin for Ace editor is still in development and includes some issues. At the time of this work redaction, it was to complicated to be implemented.

A solution would be to change the text editors used in the application and for instance opt for *Code Mirror* whose tern plugin is more functional but still sparsely documented.

5.3 Current deployment and source code

The application is currently deployed on EvenNode, a Node.js web hosting platform, and is accessible at the following URL:

`http://codeyourgames.eu-4.evennode.com`.

The source code can be browsed at the following GitHub repository:

`https://github.com/sChupin/codeyourgames.git`

Appendix A

Application screenshots

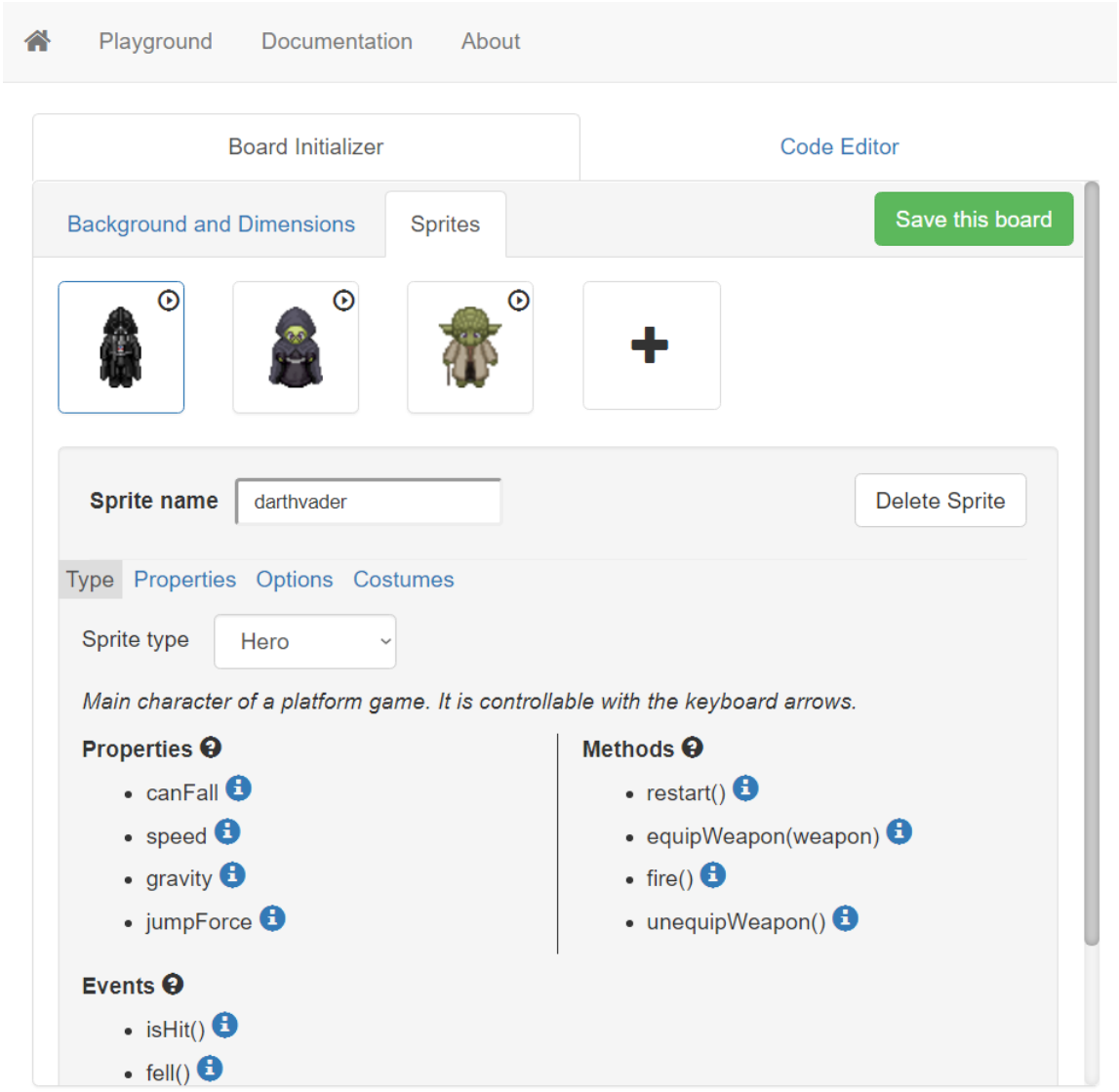

Figure A.1: Board initializer sprites list (left part)

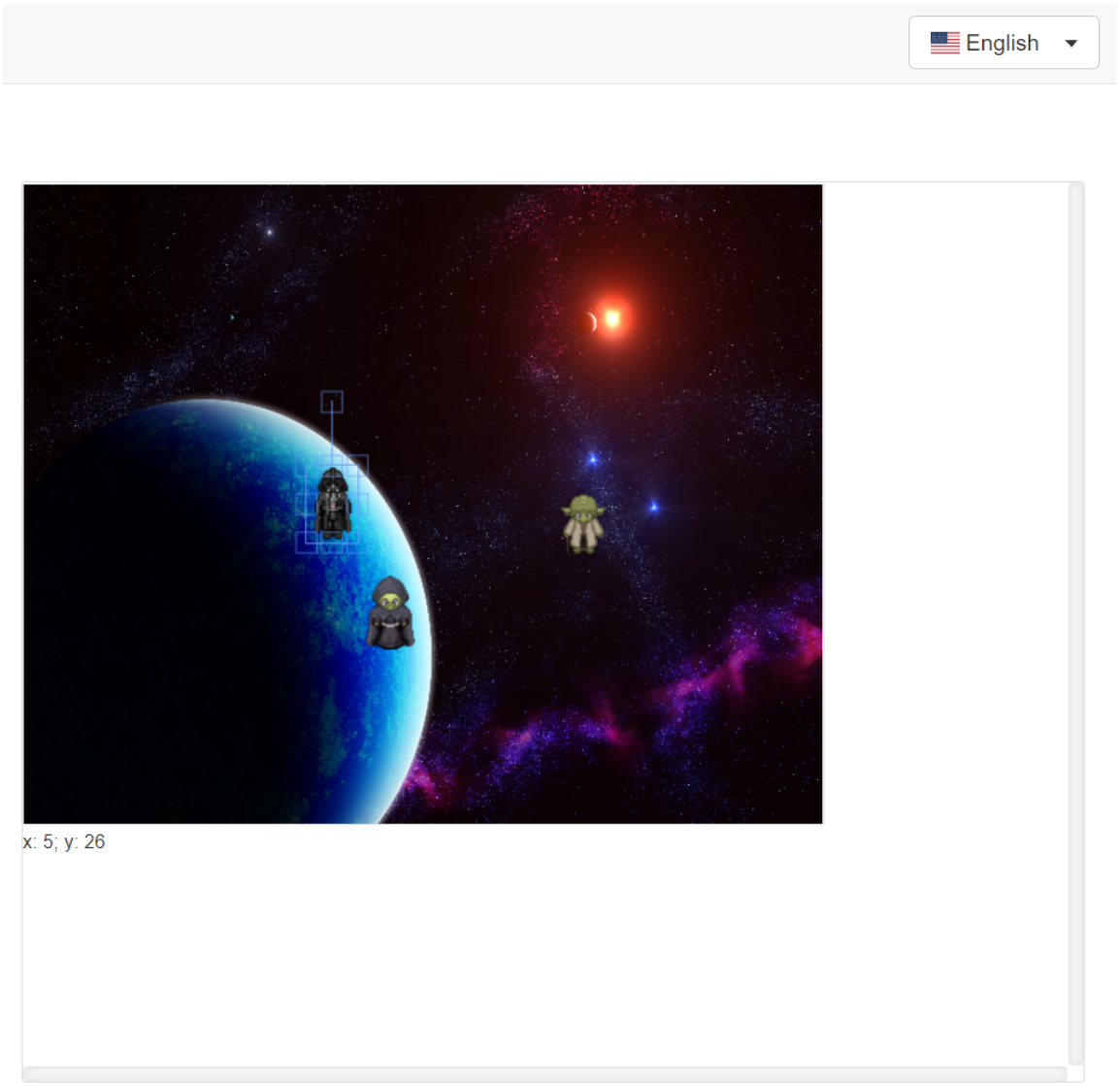

Figure A.2: Board initializer sprites list (right part)

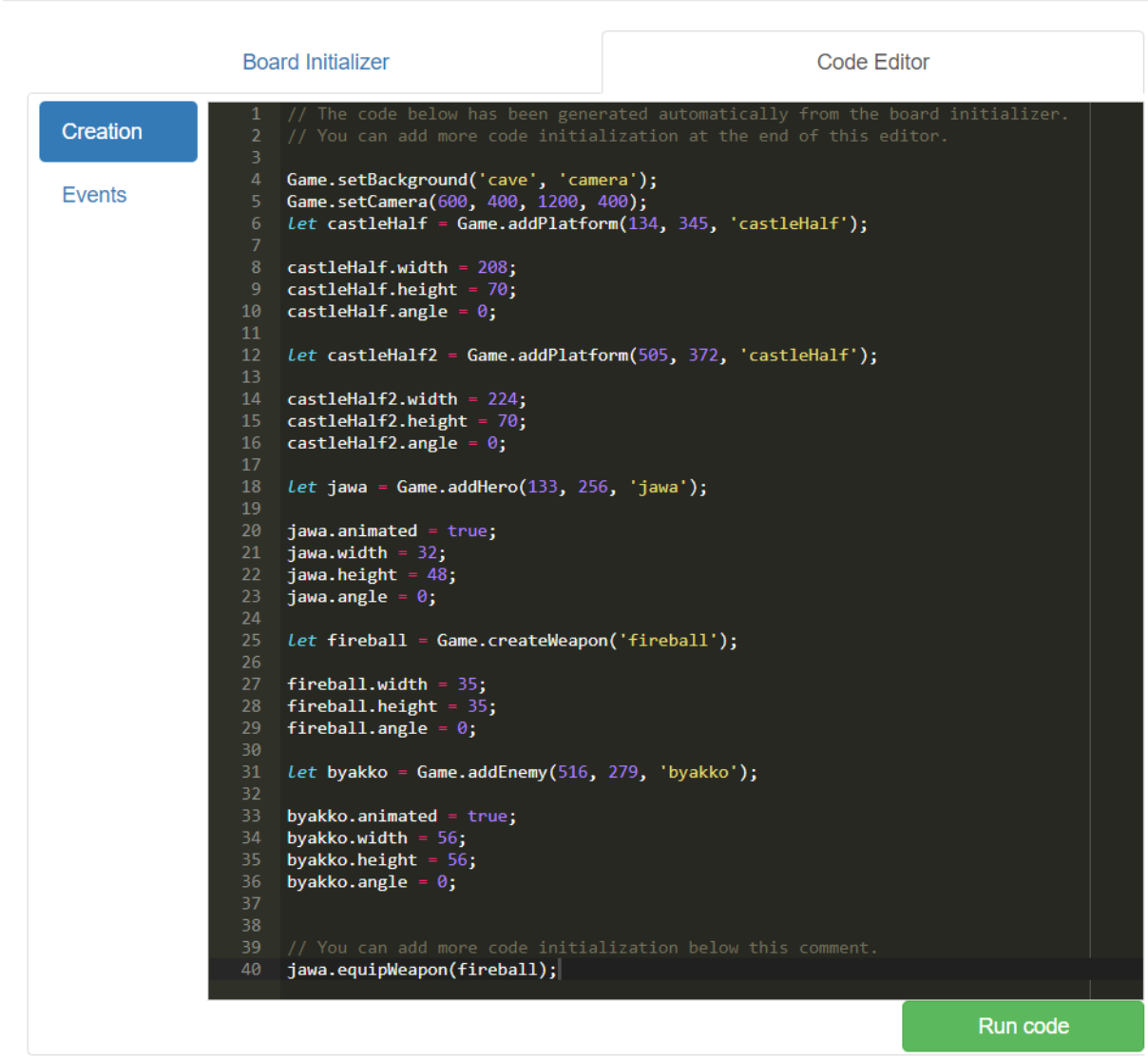

Figure A.3: Code edition - Creation editor

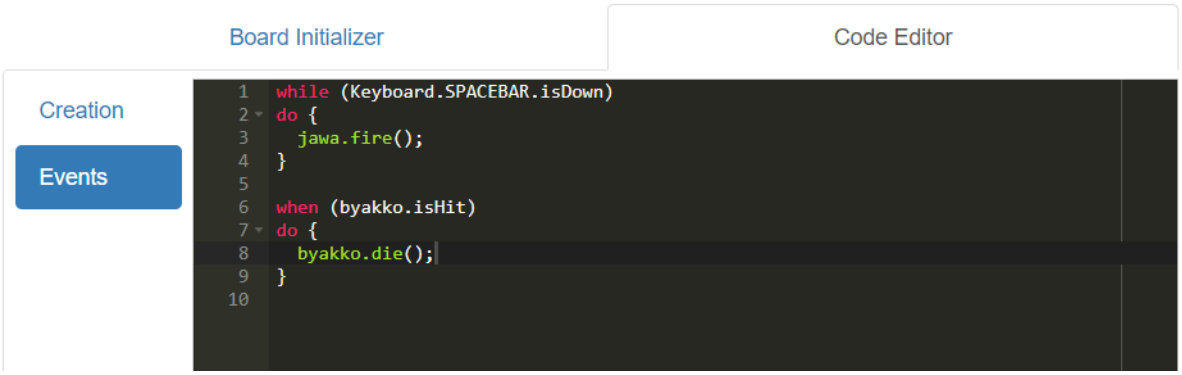

Figure A.4: Code edition - Events editor

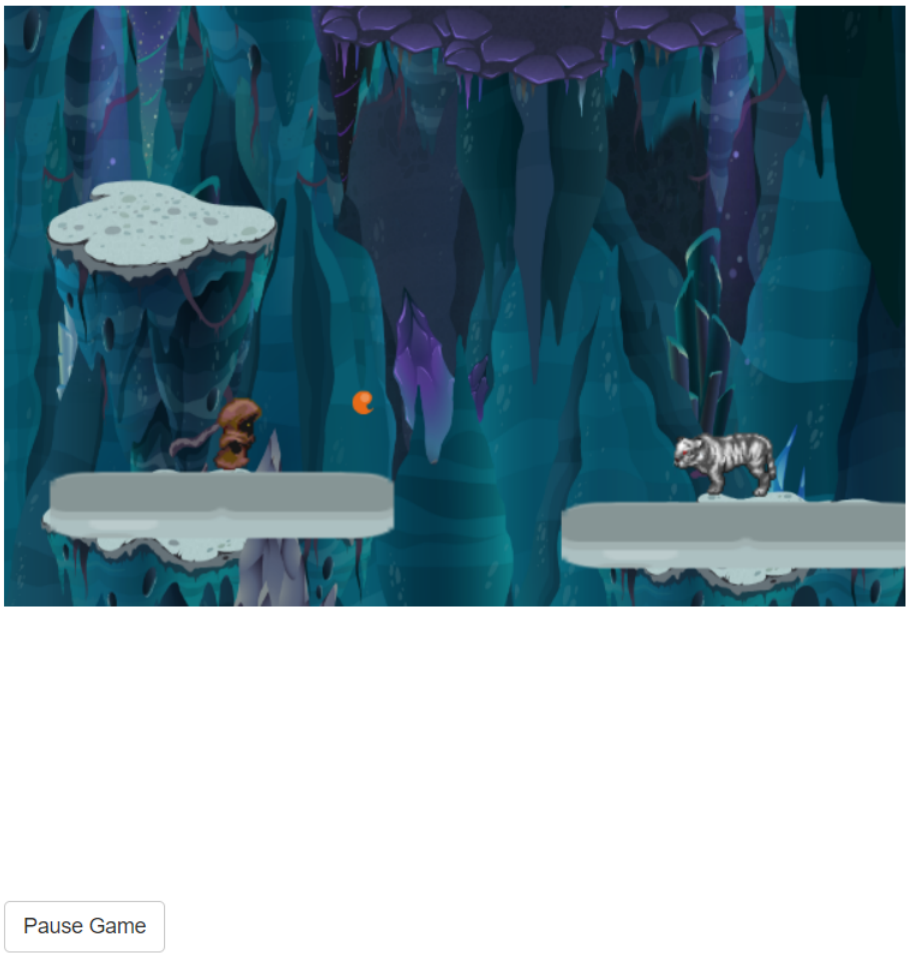

Figure A.5: Running game

Appendix B

Web resources

[1] **Scratch** https://scratch.mit.edu

[2] **CodeCombat** https://codecombat.com

[3] **Aurelia** http://aurelia.io

[4] **TypeScript** https://www.typescriptlang.org

[5] **Fabric.js** http://fabricjs.com

[6] **Ace** https://ace.c9.io

[7] **Phaser** https://phaser.io

[8] **Node.js** https://nodejs.org/

[9] **Express.js** http://expressjs.com

[10] **MongoDB** https://www.mongodb.com

[11] **Jison** https://zaa.ch/jison/

List of Figures

List of Listings

Bibliography

[Alb15] Brie Albert. Infographic: How game-based learning can support strong mathematical practices. *MIND Research Institue Blog*, 2015. http://blog.mindresearch.org/blog/game-based-learning-infographic-strong-math-practices.

[Cam16] Billy Camden. Gaming is "good for children's brains", study suggests. *Schools Week*, 2016. http://schoolsweek.co.uk/gaming-is-good-for-childrens-brains-study-suggests/.

[Cha16] Yannick Chavanne. Tendances 2017: Pénurie promise de spécialistes it. *ICT journal*, December 2016. http://www.ictjournal.ch/articles/2016-12-07/tendances-2017-penurie-promise-de-specialistes-it.

[dS13] Académie des Sciences. L'enseignement de l'informatique en france, il est urgent de ne plus attendre. Technical report, Académie des Sciences, 2013. http://www.academie-sciences.fr/pdf/rapport/rads_0513.pdf.

[NJ13] Julie Henry Noémie Joris. L'enseignement de l'informatique en belgique francophone : état des lieux. In *Drot-Delange, B. ; Baron, G-L. & Bruillard, E. Sciences et technologies de l'information et de la communication (STIC) en milieu éducatif, 2013, Clermont-Ferrand, France.*, 2013. <edutice-00875646v1>https://edutice.archives-ouvertes.fr/edutice-00875646v1/document.

[pha] 100 best 2d game engines as of 2017. https://www.slant.co/topics/341/~best-2d-game-engines. Accessed: 2017-08-05.

[SE16] Classes moyennes et Energie SPF Economie, P.M.E. Baromètre de la société de l'information. Technical report, SPF Economie, P.M.E., Classes moyennes et Energie, 2016. http://economie.fgov.be/fr/binaries/Barometre_de_la_societe_de_l_information_2016_tcm326-278973.pdf.

[Smi16] Megan Smith. Computer science for all. *obamawhithehouse.com blog*, 2016. https://obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all.

[Tec16] Burning Glass Technologies. Beyond point and click. Technical report, Burning Glass Technologies, 2016. http://burning-glass.com/wp-content/uploads/Beyond_Point_Click_final.pdf.