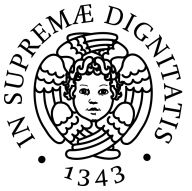

Università di Pisa

Dipartimento di Informatica
Corso di Laurea Specialistica in Tecnologie Informatiche

TESI DI LAUREA

A user-friendly authoring language and development environment for graphic adventure games

Relatore:
Prof. Antonio Cisternino

Candidato:
Andrea Serreli

Correlatore:
Dott. Andrea Canciani

Controrelatore:
Prof.ssa Chiara Bodei

Anno Accademico 2013/2014

A mamma, papà e Silvia

Abstract

This thesis discusses the implementation of a *Domain-Specific Language* (**DSL**) and a web IDE to visually create web applications focused on interactive storytelling, with particular attention to those users with little or no programming experience. The system has been implemented to validate the reactive programming model proposed by the evReact framework and its ability to easily represent control flow as a data structure. Moreover, we have been able to persist the current state of the workflow by serializing the networks underlying the evReact model. Being the whole system aimed to non-programmers, it was necessary to define intuitive ways of letting users bundle their resources and to carefully evaluate the minimal set of features they would need to express interactions, reactions and complex behaviors. This evaluation allowed to define a Domain-Specific Language that suits the problem domain and that the users can use to define game scripts at a higher level. Authored applications are compiled in JavaScript/HTML5/CSS3 and wrapped into a web page, ready to be finally uploaded.

Contents

1 Introduction

Nowadays the worldwide diffusion of Information Technology (**IT**) has heavily influenced the way of practicing any job. It is not unusual to see a worker being disoriented by the change of his traditional methods to exercise profession, or being forced to hire a team of computer scientists to develop *ad hoc* assistance tools because of inadequate programming competence, thus involving a significant expense.

Generally, there are two approaches aimed to facilitate the development of very specific classes of software: the first consists in the definition of *Domain-Specific Languages* (DSLs) which, in contrast to *General Purpose Languages*, are specific for the treatment of the domain of a given problem; the second consists in the implementation of *authoring systems* which provide preprogrammed features for the creation of certain categories of applications. These approaches are not discordant and in fact often converge into one authoring software built on top of a DSL.

In parallel to the constant changes brought by the integration of IT in everyday life, many new programming paradigms have been introduced during the years as candidates for replacing or enhancing the traditional ways of *thinking* how to design software. *Reactive programming* has recently gained popularity as a paradigm that is well-suited for developing event-driven and interactive applications. It facilitates the development of such applications by providing abstractions to express time-varying values and automatically managing dependencies between such values [1].

The **evReact** framework, developed by Andrea Canciani at the University of Pisa, proposes a reactive programming model based on *Petri* nets. Interestingly the decision to create an authoring system, which will be exposed

in the following subsection, turned out to be a very good testing ground for the validation of the model proposed by evReact. When developing an application belonging to a well-known problem domain, the use of DSLs and/or authoring systems created for such domain can dramatically cut the costs of production. Graphic and interactive development environments for the creation of applications are consequently very popular because they have proven useful both for experts and non-programmers.

The aim of this work is dual:

i. to provide an authoring system for the creation of web applications focused on *interactive storytelling* (**IS**) that is really portable to the highest possible amount of platforms, including mobile platforms as smartphones and tablets, bypassing the additional intermediate work just described;

ii. to exploit and validate the reactive programming model proposed by the evReact framework.

The authoring system will primarily be addressed to non-programmers and will adopt the approach of *visual programming*.

Authored applications will belong to a particular videogame genre known as *graphic adventure*, which not only is a general manifesto for IS, but also abounds with complex control flows and a structure which naturally fits very well the reactive programming paradigm and therefore can validate the evReact model.

The structure of this thesis is briefly described as follows:

In **Chapter 2** (Background), we will focus on the fundamental aspects of the problem. The main concepts of IS will be explained in detail, proceeding shortly after to a general analysis of the various approaches that have been adopted for it. We will see how rudimentary forms of IS were achieved long before technological progress, as well as how the interactivity degree in stories has gradually increased as technology and computer science advanced. Indeed, *gamebooks* and *interactive movies* will introduce the part of the chapter that describes different typologies of IS. Then, we will talk about a later form of IS, born thanks to the increasing popularity of computer games during late 70s / early 80s. More importantly, we will go on with a close and more detailed examination of a particular branch of IS, that is the branch of the so-called *graphic adventures*, which have been the main inspiration for this work and were hugely popular during the 90s. In particular, we will define what they actually are, what is their structure, why they were so hugely

popular and what caused their decline. The analysis will then explain why they are making a comeback and why they are still interesting nowadays. Finally, a brief explanation of authoring systems and DSLs will be provided, as well as their advantages, their disadvantages and the trade-offs that are often necessary to exploit them.

Chapter 3 (State of the Art) will study, under different points of view, several authoring systems for the creation of highly interactive applications. We will see that some of them are inherently dedicated to IS, while others *can* exploit it with a greater effort. We will also consider a milestone of the field, which is the *SCUMM* engine: while almost 30 years old and not "modern", it is widely considered as the main basis for comparison for modern engines that aim to offer similar features. The Chapter ends describing the main, most popular modern tools that are now used to facilitate the creation of graphic adventures: Adventure Game Studio, Wintermute Engine and Visionaire Studio. Considerations regarding their portability and the several features offered by these engines will be done.

Chapter 4 (Tools) overviews the tools and frameworks that have been used to implement the authoring system. The chapter is subdivided into five sections, each one introducing a particular *family* of tools. The first section is dedicated to the recent HTML5 standard, and describes thoroughly the main features that have been essential for our work, that is the canvas element and the SVG support. Of course, we will see advantages and drawbacks of both, and the reason why both of them have been adopted.
The following section describes one of the most popular JavaScript frameworks, called jQuery. In this section, we will understand what jQuery does, why it is so powerful and how it can be convenient and can shorten one's work by thousands of code lines. Later in this section, we will see two jQuery plugins that have been heavily utilized: jsTree and inputfile.js.
The third section focuses on the Twitter Bootstrap framework and its X-editable plugin. Particular attention will be given to Bootstrap's graphic widgets and its grid-based layout system.
The fourth section presents the evReact framework and many fundamental aspects of its approach to reactive programming. We will consider many different types of evReact's expressions, describing both primitive expressions and complex expressions obtained by their composition. We will also debate about the formalism of Petri nets, precisely why they are a popular formalism and what is its connection with evReact.
In the last section, there will be a brief description of two JavaScript frameworks that have been helpful for the realization of the system: the first is Snap.svg and facilitates the utilization of SVG resources within a web page,

the second is Paper.js, which provides many geometry classes and support for canvas drawing.

In **Chapter 5** (Design) the design of the application will be explained in detail, starting from the problem's domain analysis that led to the design of the DSL. Indeed, the first section describes the considerations that have been done about the general problem domain, and the conclusions that have been drawn which led to the real design of the language. The section goes on describing the phases of such design: the definition of a context-free grammar, its parsing, and the semantics of the various language constructs. Furthermore, we will understand the subdivision of generic valid scripts into a header part and a body part.
The second section of the chapter describes how the editor appears and behaves, as well as the functions performed by its many sections. For each section of the editor, we will find instructions and descriptions of what each field does, and how a section can communicate with the others, if possible.
The last section provides highly detailed explanations of the *conceptual* structure of the entire system, with the addition of UML class diagrams to clarify the hierarchy of the system entities, the communication between them and the role of single entities and groups of entities that cooperate together. Particular attention has been given to the description of the scripting part of the system. We will indeed see how the system manages three different representations of the DSL: a purely graphical representation inside the DOM view, a syntactic representation and a semantic representation. Starting from a visual representation, we are going to follow the process that obtains an intermediate syntactic representation of the language and a final semantic representation by exploiting the evReact framework and other core functions.

In **Chapter 6** (Implementation) we will see the implementation of the several parts of the system. Each part is implemented by a JavaScript module that provides both the needed data structures and the functions that allow to manipulate them properly and to make them cooperate. The classes are described more minutely than in the previous chapter, without omitting any method or member: for each method and for each member, there is a description of its role inside the class instances and, more extensively, inside the system. The primitive classes of the system will be described in the first section. The second section explains the implementation of the pathfinding by means of spacial partitioning with quadtrees and the application of a well known heuristic algorithm for shortest paths known as A*. A description of the general theory and the benefits of quadtree-based spacial partitioning and applied to A* will be provided.
The subsequent section explains the most important part of the system, that

is the part that manages the DSL and its various representations. Firstly, we are going to see the how the visual script is compiled into its syntactic counterpart and vice versa. Secondly, we are going to consider the compilation from syntactic representations of the scripts to evReact expressions, understanding how evReact is a substantial part of the run-time support of the exported games. The section finally explains how the script conditions are parsed by a recursive descent parser, factorizing the original context-free grammar into an LL(1) grammar, obtaining an abstract parse tree that is then passed as a function that evaluates the conditions.
The following section depicts the part of the run-time support that is not directly provided by evReact but, instead, by system core-functions that can only be called by evReact expressions. In particular we will see that, for each language construct that represents a game side effect, there is a core function that implements its semantics, possibly exploiting JavaScript closures when the side effect can be interrupted before reaching its completion.
The chapter proceeds with a section that is dedicated to the testing of the games from within the system, i.e. before their final exportation, with a *WYSIWYG* (What You See Is What You Get) approach.
Another section is fully dedicated to the canvas utilization, particularly to its event listeners and to their different event handling depending on the canvas state.
Two more brief sections illustrate the process of loading/saving scripts with the help of the well known JSON format and the process of exporting the games as stand-alone independent web pages. The last section of the chapter describes some parts of the DOM manipulation with the help of the jQuery framework: first of all we will consider the attachment of listeners and handlers of input events to the DOM elements. Then, we will describe the creation of complex DOM nodes as panels for game entities, including details about our usage of the Bootstrap features as well as the utilization of jQuery's selectors.
Lastly, a detailed description of the visual programming part of the IDE will be given from different points of view: what the visual constructs are and how they are implemented, how the jsTree framework was essential to develop the visual programming feature, and how we have guaranteed to always obtain certainly well-formed scripts by means of other specific event listeners and handlers for the trees that visually represent a script.

Chapter 7 (Conclusions and future works) will firstly present a brief summary of the obtained system and its most meaningful features, proceeding to drawing conclusions about the results that have been obtained and comparing the final product and its features to other common, similar products.

The last part of this chapter will explain how this work can be improved and what more features could be useful but could not be provided because of time constraints.

2 Background

2.1 Interactive Storytelling

Thanks to the aforementioned spread of IT, even teaching and learning methods have gone through a great transformation, becoming more and more interactive and less traditional. For example, *e-learning* for kids has proven astonishingly successful because it actually engages the subjects to actively participate to the educational process, often presented to the kids as a game they can play together. Moreover personality tests, either in the field of education or employment, have proven particularly efficient in the form of interactive programs where the candidates can make choices and have virtual conversations.

The field of *interactive storytelling* (**IS**) is a relatively new discipline and wraps both of these spheres of application. While it is impossible to unanimously give a definition to IS that would not result being trivial, it is sure its fulcrum is to merge the classic approach of narrative books with more modern, generally technological aspects that allow those who follow a story to interfere with it in some way. The *degree* of such interference is what makes the difference between different IS-focused applications. Interestingly, the type of interaction and its emphasis can result in totally different final products. It is important noticing that *not* any application with interactive components and a general plot belongs to the set of IS-based applications. In other words, a game of the type "Aliens have invaded the Earth, so kill'em all to save mankind" does not make IS: instead, every IS-based application has storytelling as its *main* objective. The following subsections describe several approaches of interactivity applied to storytelling throughout the years.

2.1.1 Gamebooks

For historical purposes, though not exploiting any type of computer technology, it is important to tell about the so-called *gamebooks*. Gamebooks are traditional printed books, with the peculiarity that they give readers the possibility to make choices and decisions during their reading. This is granted by explicitly asking readers to go to a specific page of the book or to another one in result to one or more choices they have to take at certain points of the narration. The plot follows a tree structure as it branches in different progressions, often having different endings based on the choices made. Even if this is a rudimentary way of including readers' participation, such interactivity is indeed what makes gamebooks the precursors and first example of IS. Moreover, apart from the interactivity itself, gamebooks were a good instrument to teach morality to young readers, as endings could also be "good endings" or "bad endings" according to the ethics of their decisions. It is not a case that gamebooks, during the 1970s, were very popular especially among kids. More modern examples of interactivity in books is the use of *hyperlinks* to switch between paths, with traditional pages being replaced by HTML pages.

2.1.2 Interactive Movies

Interactive movies are another notable branch in the field of IS. The approach they adopt is similar to that used in gamebooks, with a very reduced interactivity degree, but with the substantial difference that it is applied to *movies*. Such difference is significant since it involves a director, cameramen and real actors acting as in a traditional movie, with pre-filmed sequences that can be watched by the audience, again by means of interactive mechanics.
The first interactive movie was ***Kinoautomat*** (1967), and its interactivity consisted in a live moderator that appeared on stage at certain points to ask the audience to choose between two scenes. The chosen scene would play following an audience vote. The interactive element was achieved by switching a lens cap between two synchronized projectors, each with a different cut of the film.

Years later, the interactivity of these movies exploited less primitive methods thanks to technological progress. When *laserdiscs* were invented, the real innovation they brought was the fact that they allowed a *direct access* to their data, instead of the classic sequential access. This important property was immediately exploited to enhance interactive movies, resulting in the

birth of *laserdisc games*. Laserdisc games presented pre-filmed sequences, sometimes featuring real actors and sometimes animated cartoons. The most famous laserdisc game is *Dragon's Lair* (1983), originally released for arcade machines.

Figure 2.1: Screenshot from Cinematronic's *Dragon's Lair*, 1983

It featured animated sequences drawn by ex-Disney animator Don Bluth and allowed users to interact with the movie by a few buttons installed on the machine, to be pressed at the right time. Considering the very limited computational and graphic resources that "normal" videogames could count on at that time, it is easy to understand why Dragon's Lair had such a huge success, as it indeed had graphics ahead of its time of at least 15 years and players did not care about them being pre-rendered. However, as computers became more powerful later on, laserdiscs games began losing ground because of *costs* and *limitations*. The production of a laserdisc game actually required a very high budget compared to the one needed for developing a state-of-the-art videogame in the middle of the 1990s. Furthermore, the expressivity and the complexity of videogames was rapidly growing up and outclassed the very reduced interactivity offered by laserdisc games. The last nail in the coffin of laserdisc games was put when videogames could count on enough power to actually offer the same graphic quality and more entertaining and intense interaction mechanics.

2.1.3 Text Adventures

Text adventures are videogames in which the entire interface is, as the name suggests, *textual*, and the player can interact with the world by giving text *commands* as input. Text adventures brought IS further, basically adopting a diametrically opposite approach in comparison to laserdisc games: instead of reducing interaction to a minimum and relying on cutting edge graphics, they kept their appearance limited to simple text as in a book and relied on a level of interaction that was only limited by the programmers' fantasy. In fact, though the *gameplay* simply consisted in typing in commands, such commands could give players the illusion of actually being part of a complex story, to spatially move inside environments and even to feel in danger thanks to the well-known phenomenon of the *suspension of disbelief*. Obviously, even the players had an unlimited set of commands to type in, and this could stimulate their imagination much more than a gamebook or an interactive movie, where their choices were generally limited to a small set. Because of this reason, text adventures used a *parser* that analyzed the input strings and informed the player of the parsing result, that is if the command was unexpected or unrecognized or if it originated a transition of the game state. Text adventures quickly lost popularity as soon as similar applications with the addition of graphics were born.

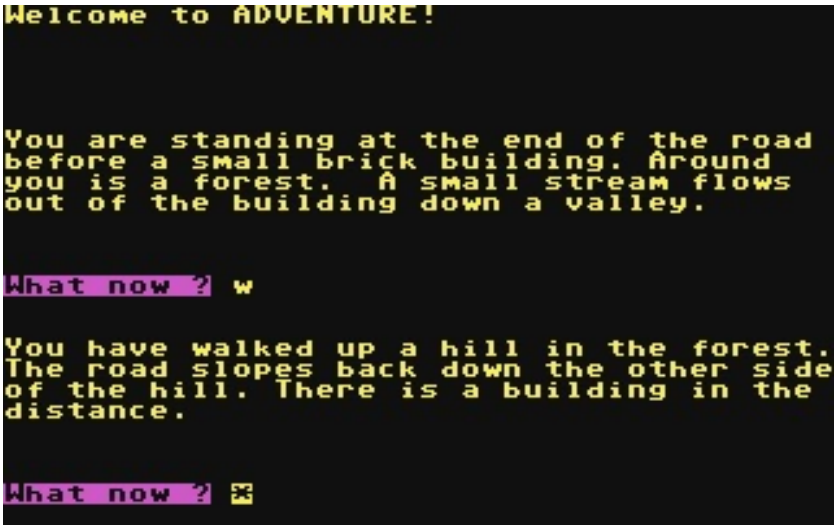

Figure 2.2: Screenshot from *Colossal Cave Adventure*, 1976

2.1.4 Graphic Adventures

In the following subsections, we will dive into the analysis of the cornerstone of the whole thesis and understand what graphic adventures are, what made them popular, what later made them unpopular, why they are making a comeback and why they are an interesting case of study.

2.1.4.1 What is a graphic adventure?

Graphic adventures belong to a category of videogames which puts a strong emphasis on narrative and storytelling. Generally there is a main, predefined plot, and a set of puzzles that the player has to solve to progress with the story; the player can explore places, talk to characters and combine objects in order to solve the puzzles.

In most cases, no quick reflexes or particular abilities are required to play through a graphic adventure: on the contrary, the game is all about parallel thinking and the player has usually unlimited time to think about possible solutions to the puzzles. At a given moment, the player may have different puzzles still to be solved which are independent to each other. Furthermore, solving a puzzle results in a transition of the game state, often unblocking a new puzzle to be solved: therefore some puzzles have a dependency relation on other puzzles, while others do not, guaranteeing a certain non-linearity of the story unfolding, despite it following a fixed plot. This particular structure can be clearly described by means of a *Puzzle Dependency Graph* (***PDG***) [2], where each node represents a puzzle and each arc a dependency relation, as shown in figure.

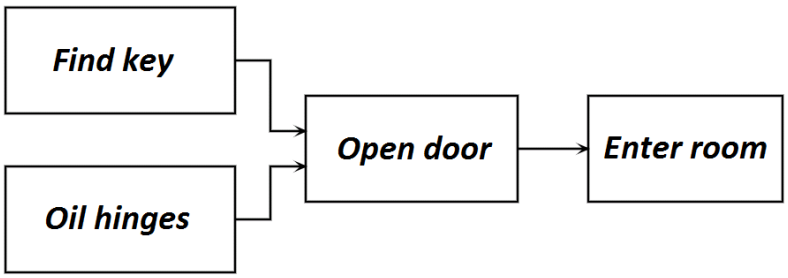

Figure 2.3: A puzzle dependency graph

The figure shows two independent puzzles, “Find key” and “Oil hinges”: when both of them are solved, a third puzzle can be solved, that is “Open door”. This puzzle depends on the resolution of **both** the two independent puzzles. Once the door has been opened, the puzzle “Enter room” can be solved, as it depends on the “Open door” puzzle. PDGs are essential not only for describing the flow of the game and the plot, but also to guarantee that the player won't fall into dead states of the game (e.g. to reach b the player needs to take a, but he forgot to do so and now a is unreachable). Graphic adventures are the natural evolution of the aforementioned text adventures, and initially almost coincided with them in every aspect but the presence of small, static illustrations that enhanced the overall experience by giving players a *visual* representation of the situation described by the text. The real advancement came with the employment of *animated* graphics that really “gave life” to the story. The characters could actually *move* within the environments and the situations once described by text could also be displayed in real time. In early graphic adventures, exactly as in text adventures, the commands had to be typed in by the user, then a parser evaluated them and, if they matched an expected sentence, the game flow would have proceeded in some way, or else the parser would have informed the user that there was no match. One of the biggest critics to this type of interaction was the rigidity of the parser: it was very common to type in an appropriate command and to have it rejected by the parser because it expected a synonym of a word, so the player had to face the frustration to try to guess synonyms and alternative ways of expressing the same command.

Figure 2.4: Screenshot from *King's Quest* by Sierra On-Line, 1983

A great innovation to the genre was brought by *Maniac Mansion* by Lucas-

Film Games (later LucasArts), which is universally considered as the ancestor of modern graphic adventures: the parser was put away in favor of a *point and click* interface, so that the player could explore the world by simply hovering the mouse over the scenery to get brief descriptions of the objects he could interact with and, by clicking the mouse, select from a fixed set of possible actions to be performed on the objects and characters which populated the environment. Such actions were contained inside a minimal GUI together with a textual inventory, as shown in figure:

Figure 2.5: Screenshot from *Maniac Mansion* by LucasFilm Games, 1987

2.1.4.2 The rise and fall of graphic adventures

Graphic adventures had their hayday from the end of the 80s up to the end of the 90s, when the notable advances in the field of computer graphics and the advent of GPUs led to a deep change in the way of making games. The rise of the third dimension in videogames is often considered to be the main reason of the decline of the graphic adventure genre, which had the two American companies LucasArts and Sierra On-Line as their main producers and undisputed leaders. Firstly, because the genre was tightly bounded to cartoon graphics which were impossible to achieve with the early 3D technology, since the amount of polygons a GPU could handle at that time was very limited. Secondly, because 3D graphics opened the door to improve other genres that couldn't be very popular before because of 2-dimensional limits as, for example, football simulators, and to create genres that didn't even exist before, as free roaming games. Furthermore, adventure games in

general have a common and inherent low *replay value*, that is the property of the game to be played more than once by players and to still entertain them. Unfortunately, once a story has been told, it is very difficult to gain the audience's attention by telling the same story once again and even if they can change some decisions the main plot will basically remain the same. Graphic adventures rapidly ended up exiled out of the mainstream market and became a niche product, kept alive by small groups of nostalgics [3].

Apart from a few software houses that later managed to adapt to 3D and to renew their way of making graphic adventures, the genre was completely buried by the game industry and "betrayed" by those companies which made it famous and popular. This situation remained stationary for the following decade, but something has changed since the diffusion of social networks and the appearance and spreading of mobile devices: social networks, as well as smartphone and tablets, have led to a massive comeback of 2D games. The former contributed to the birth of the so-called "social" games, in which the users are supposed to cooperate and share things together to progress in the game, while the latter started a huge wave of "casual" games, that is simple games in which interactivity is very limited and the efforts to play are minimal, in order to allow players to relax and have fun without the frustration of losing or getting stuck.

This second youth of 2D games, and the fact that the traditional point and click approach adopted by most adventure games inherently fits very well the touch gestures of mobile devices, brought back the interest in graphic adventures. Of course, their popularity cannot be comparable to their golden age's and it is very likely it will never be, but the genre has surely found a reason to exist again and to guarantee a financial return.

It is also important to consider that graphic adventures are not only limited to "videogamers", but have also proven very successful as learning methods for kids and as tools for developing negotiating capabilities and life skills, thanks to the massive presence of interactive dialogues.
For example, *Zapdramatic* was founded in July 2000 by filmmaker Michael Gibson and Negotiation and Alternative Dispute Resolution experts Allan Stitt, Frank Handy and Lisa Feld. This company claims its goal is to "popularize the art and science of negotiation to a world audience using interactive simulated adventure games" [4].

2.2 Authoring systems and DSLs

While traditional storytelling is all about the story, IS requires some confidence with computer science and programming, as said above. Psychologists, school teachers and alike don't generally have this kind of background to implement computer programs with the goal of interactive storytelling. As a consequence, they have to choose either to hire professionals or to use authoring systems to build the programs themselves.

Several authoring systems focused on IS exist, but most of them suffer from *portability* issues, because they have a fixed *target platform* and, being not *cross-platform*, omit the others. This bad choice of design is generally due to the presence of virtual machines that allow to emulate different computer systems, thus permitting to run a program that is not compatible with the original machine. A similar approach is not admissible for several reasons:

- it forces users to have machines with enough computational power to emulate a different machine, thus it takes for granted they have enough funds to buy one
- it presumes users have enough confidence with computers to properly set up a virtual machine
- it presumes the program executed on a virtualized machine will run exactly as it would in the original machine, which instead is rarely true

The few authoring systems that are effectively cross-platform follow two different trains of thought for achieving *platform independence*. One consists in building and compiling the software separately for each target platform, resulting in a bigger amount of work, possible code duplication and an overall greater expense. The other approach exploits the use of interpreted languages or the compilation of the source program into an intermediate bytecode that will be interpreted by a different interpreter for each target platform. Again, the amount of work needed to implement an interpreter for each platform is not negligible at all.

Often authoring systems are built on top of a DSL. A common design issue when *creating* a DSL regards its expressivity: it is perfectly normal that this kind of language, while treating complex problems belonging to its domain with ease, may have serious difficulties to solve simpler problems that do *not* belong to it. This is natural because DSLs can generally treat complex data types as *native* types, but at the same time they are created to resolve a particular pool of problems and can hardly handle different problems, re-

gardless of their actual difficulty. This phenomenon often misleads designers to grant the language more expressivity than actually needed, losing the focus on the *simplicity* that a DSL should have and ending up with something much closer to a general-purpose language. The key to avoid falling into this "trap" is to accept making compromises after having carefully analyzed the problem's domain. This trade-off between *simple* and *expressive* is often the main challenge during the design phase of the language and plays a big role on its final efficiency.

2.3 Reactive programming

Reactive programming is a programming paradigm whose definition is still vague and, nowadays, is being formalized. *The Reactive Manifesto* [5] defines reactive systems as message-driven, elastic, responsive and resilient. The reactive programming paradigm has recently gained popularity, as it suits very well the development of highly interactive and event-driven applications. Generally, the traditional sequential programming approach forces the developers to manually manage data dependencies and changes of the application state, often resorting to function callbacks and event handling. Such management is very error-prone and requires great efforts from the programmers. Without losing generality, let us consider a simple case of expression dependency. Let $a = 3$, $b = a + 1$ and $c = 2*b - a$. In the classic sequential approach, a change of the value of a to, e.g. 5, must be explicitly managed by assigning the value 6 to b and 7 to c. With reactive programming, it is possible to automatically manage data dependencies so that each change of a will immediately reflect to the values of b and c. A similar approach can be found in spreadsheet systems like *Microsoft Excel*. Reactive programming provides abstractions to express programs as reactions to external events and having the language automatically manage the flow of time (by conceptually supporting simultaneity), and data and computation dependencies. This relieves programmers from having to care about the order of events and computation dependencies, thus allowing dynamic dataflows. Hence, reactive programming languages abstract over time management, just as garbage collectors abstract over memory management [1].

3 State of the art

Throughout the years many different engines, both free and commercial, have been released to allow aspiring developers of graphic adventures to concentrate on the story, the puzzles, the dialogues and the graphics, (almost) without having to consider the low-level aspects of the implementation. The development of this work could not exclude an examination of the state of the art of authoring system for the creation of graphic adventures. In the following sections, we will consider the most popular of them.

3.1 Dedicated authoring systems

3.1.1 SCUMM

One of the most famous and beloved engines is *SCUMM*, which was property of LucasArts and was developed by Ron Gilbert and Aric Wilmunder some time during 1986 to aid with the development of Maniac Mansion. The reason why it was included in this Chapter is that, even if it is almost 30 years old, it still is the term of comparison par excellence. SCUMM, acronym for *Script Creation Utilities for Maniac Mansion*, was both a domain-specific language and a set of tools to define locations, animations, puzzles, objects and their interaction with the player.

One of the main strengths of SCUMM was it relieved programmers of writing the game in the final target language in which the scripts would have been later compiled, and this was particularly convenient, especially considering

C64 versions of the first two games would have required them to program in pure Assembly.

Every script ran on a separate thread, with each thread cooperating with each other [6][7]. This was a very powerful approach since it allowed programmers to define a script for any game entity and then forget about it: every game entity *lived on its own*.

SCUMM games were sold as a set of game resources coupled with an executable file, which merely was the SCUMM interpreter. This approach made it possible, many years later, to run the games in an amazing variety of platforms with the *ScummVM* project, that is an interpreter for SCUMM-based games that replaces the original one, leaving the original game resources exactly as they were before.

3.1.2 Adventure Game Studio

Adventure Game Studio (AGS), by Chris Jordan, is a popular adventure game engine for Windows systems, free and open-source. It boasts a big and active community and can be somehow considered one of the main contributors to keeping the genre alive even through the darkest years. Countless games have been created with AGS, some have even been commercially released.

AGS comes up as an IDE with multiple features for defining environments, objects, characters, animations, sprites, pathfinding and GUIs; it also provides a Java/C# -styled scripting language with built-in editor and debugger, as well as a compiler [8]. The engine also supports a template mechanism for various aspects of the generated games, such as in-game GUIs, in order to have high flexibility and customization and to easily obtain the look and feel of old classics.

There are currently projects of porting the engine in more platforms.

3.1.3 Wintermute Engine

Wintermute Engine (**WME**) is another popular development kit dedicated to graphic adventures. Its features are very similar to AGS', but its approach is a bit less nostalgic and takes into account that years have passed and while some features were not essential during the 1990s, they are now common and desirable in a modern application. This can be noticed by its ability

to support real-time rendering of 3D models, modern resolutions, parallax scrolling, antialiasing and other up-to-date functions.

WME provides a graphic environment and an object-oriented scripting language with a *C-like* syntax, with the possibility to override and customize built-in methods [9].

It only works on Windows-based platforms.

3.1.4 Visionaire Studio

Visionaire Studio (VS) is a commercial engine, with which a lot of successful modern adventures have been programmed and sold in stores. It supports HD graphics, integrated LUA scripting language, particle systems and a variety of audio formats. Differently from AGS, VS does not look like a tool for nostalgic fans, but instead it seems to keep in mind the current market realities and to aim at being adopted not only by small groups of fans, but also by real software houses that want to have an economic return. This can also been noticed by the choice of a LUA scripting system, which is very popular in the professional game industry; moreover, this choice also suggests that a certain degree of programming knowledge is required to fully exploit VS's features.

Just like AGS and WME, VS comes as an IDE with various editors for game resources, game scripts and so on [10]. It also suffers from common portability issues, as the generated games are only playable by Windows systems.

4 Platform and Tools

The decision to adopt the web browser as the target platform for both the authoring system and the authored games has derived from two considerations. Firstly, as said before, the will of creating a tool which is really portable and cross-platform: since browsers are one of the most (if not *the* most) supported platforms by any device, it was easy to locate them as a the best possible choice. Secondly, because users tend to prefer applications that are ready to use over software that need to be installed locally on their machine, as they require time to be installed and properly set up. Of course, a direct consequence of said choice was to adopt JavaScript as the programming language to give life to the entire system. In this chapter, we will describe the tools that have been used for the implementation of the system, as well as the reasons behind their adoption.

4.1 HTML5

The recent introduction of HTML5 made this work possible. Without it, the implementation of the system would not have been possible in a reasonable time. HTML5 has added native support for *modern* webpages, with new DOM elements/attributes and with new APIs that can be used with JavaScript. The most notable new features include the support for canvases, video and audio files, drag&drop and the integration of *Scalable Vector Graphics* (**SVG**). In the next pages, we will see what parts of HTML5 have been used.

4.1.1 Canvas

The <*canvas*> element allows to dynamically render images, lines and other primitive shapes [**canvas**]. It was largely used within the editor to show the rooms' backgrounds and sprites and to let users define the vertices of polygons that mark up certain areas of interest within the environments they are creating. While a canvas is a data structure that suits very well the quick drawing and erasing of geometric primitives and images without even affecting the DOM, it also has some drawbacks.

First of all, it is raster-based, i.e. there's no direct correlation between the object sprites printed on the canvas and the object to which a sprite belongs : a canvas does not keep trace of anything but the blocks of pixels it receives. In other words, graphic elements printed on the canvas don't have a counterpart object of a scene-graph or the DOM, as a canvas is just a pixel-based drawing surface that stores pixel information. For this reason, object indexing by sprite is not native and it is necessary to obtain the object that "owns" a given sprite by checking its spacial coordinates inside a canvas and comparing it to the object position or bounding box. On the other hand, its raster-based nature allows direct pixel indexing and manipulation, which is not possible in vector-based images. In fact, we will later see that the pixel manipulation of a canvas has been used in the project for the spacial partitioning of game environments in order to implement pathfinding in a highly efficient way. Precisely, the color of a pixel has been used as information to distinguish zones where walking was allowed and where it was not.

Another problem is given by the layering of objects: z-ordering is also not native in canvas structures and can only be achieved by applying the *Painter's algorithm* or *priority fill*, that is drawing figures in order of distance from the observer, starting from the farthest up to the nearest and simply drawing the latter over the former. Consequently, a data structure for representing the z-order of the objects must be explicitly maintained. In addition, changing the position of an object implies the redrawing of the entire canvas or chunks of it.

One more drawback is that raster images are not resolution-independent, which means that scaling a raster image can give results that are not acceptable or that seriously damage its overall appearance. There are many scaling algorithms for raster images, some of which have surprisingly good results, but in fact, when using raster images, it is not possible to guarantee an acceptable final result without having the same image available in different native resolutions. On the contrary, vector images are inherently

multi-resolution and do not need any image processing but, as said, do not allow pixel manipulation, and this is potentially a drawback. Anyway, using raster images naturally means less reuse and more work for graphic designers. The figure below shows the larger scaling of two versions of the same image, one vector and one raster.

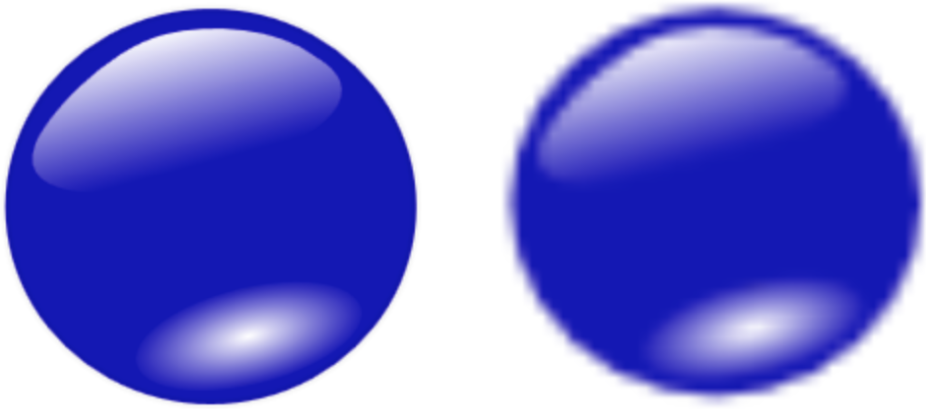

Figure 4.1: Zooming a vector image (left) and a raster image(right)

All considered, while the canvas element suits very well the *editing* phase, it implies several problems in the actual game. For this reason, as we will see later, it was decided to rely on the SVG technology for the run-time rendering of the game. In fact, our application uses a hybrid approach to exploit both data structures and their advantages.

4.1.2 Scalable Vector Graphics

Scalable Vector Graphics (**SVG**) is an XML-based markup language for the description of graphic objects. Graphic objects can be vector graphic shapes, raster images and text [11]. It provides support for interactive, dynamic images and animations.

SVG is supported by all major web browsers and is the core tool we have used for the rendering of every graphic entity of the games. Every SVG element belongs to the DOM, therefore any on-screen graphic element can be immediately indexed without further calculations on their bounding boxes and, furthermore, a modification to an element is immediately reflected graphically by the re-rendering of the browser.

Among the basic shapes offered by SVG, the <*polygon*> element has been

very convenient to the project, as it provides native support for the resolution of the *PIP* (Point-inativelyn-Polygon) problem. That is, event listeners attached to <*polygon*> elements can react at inputs, such as mouse events, that occurr *inside* the polygon, thus calculating if it contains a given point p. This feature has been exploited by representing *hotspots* as <*polygon*> elements. An hotspot H_I is defined as the fraction of space associated to an item I that defines its interactive zone, i.e. the zone that must react to user inputs.

Moreover, z-ordering is native, as it only requires the manipulation of the *z-index* CSS property and does not need explicit redrawing like a <*canvas*> element would.

4.2 jQuery

jQuery is a JavaScript library that provides countless features to help JavaScript programmers with the DOM manipulation, event handling and animations [12]. Nowadays it has become a standard *de facto* and is almost unavoidable for programmers of web applications and complex, modern websites. jQuery's motto is *Write less, do more* and it perfectly describes why this tool has become so important. It is officially supported by companies like Microsoft and Nokia, which bundle it on their platforms. Precisely, Microsoft includes jQuery in Visual Studio for the development with the ASP.NET Ajax and MVC frameworks. Google, which provides a hosting service for popular external open-source libraries, hosts jQuery and allows developers to include it by simply adding a snippet on their webpages. MediaWiki as well has been using jQuery since version 1.16.

Basically, the jQuery core function (*$*) acts as a wrapper for any DOM element and provides utility functions for the access and the manipulation of attributes and properties, appending and prepending of child nodes, modification of CSS style and so on. The wrapping occurs by simply passing an element as parameter to *$*, which will return the wrapped element. For example, *$(node)* will return a wrapper of the original node. Many jQuery functions work as getters *and* as setters, and this sensibly reduces the amount of methods that developers should remember to work conveniently. For example, *$(elem).text()* will return the element's inner text, but *$(elem).text('Hello, world!')* will set the element's inner text as the string "Hello, world!".

What really makes jQuery essential is one of its core features, that is the selector, which borrows its syntax from the CSS selectors and spares programmers *hundreds* of lines of code. In fact, while retrieving a node of the DOM tree might require at best several lines of code (depending on the property of interest of the element and the number of matching elements), jQuery's selector makes it very fast and simple to obtain it. jQuery's methods can be chained in an arbitrarily long sequence, that is complex selections can be potentially performed in one single command. Actually, the return value of a jQuery selection will be an Array of elements, which may be empty. This is due to the fact that a selection may match more than one element or no element at all, depending on the property that acts as a filter for the selection. For example, *$('#my_id')* will return, if present, a one-sized array containing the element with id *my_id*, if present, or an empty array. Furthermore, *$('div')* will return, if $N \geq 0$ * * elements are contained in the DOM, an N-sized array containing each one of them.

A notable feature, which has actually been very useful during the implementation of the project, is the *each* function. This method allows to apply a function to every element of a given list of elements returned by a jQuery selection and is indeed very powerful and time-sparing. Let us consider a more complex example to illustrate jQuery's benefits:
A webpage contains $2N$ $<div>$ elements with class C_0, N of which have also class C_1. Each $<div>$ element contains a $<span>$ element, which in turn contains a text string. The developer wants to set each string that is contained inside a $<div>$ element with class C_0 but *not* with class C_1 to the value "Hello, world!". In pure Javascript, he would write something similar to:

```
1 var a = document.getElementByClassName('c0');
2 for(var i = 0; i < a.length; i++)
3     if(a[i].className.match(/c1/) === null)
4         a[i].childNodes[0].innerText = 'Hello, world!';
```

The following snippet of code has the same result, but exploits jQuery:

```
1 $('.c0').not('.c1').find('span').each(function() {
2     $(this).text('Hello, world!'); });
```

As can be seen, chaining of method calls and selectors greatly simplify and reduce the developer's work. jQuery has also been used for quick CSS manipulation and for the attachment of *all* the event handlers of the project and also in some cases where a certain degree of asynchrony was required.

One of the strengths of jQuery is its support for plugins: as by now, thousands of useful plugins have been released by developers all over the world.

4.2.1 jsTree

jsTree is a jQuery plugin that provides features for the creation and the manipulation of tree data structures as a combination of native DOM elements that *visually* look like trees. This tool has been created to offer a graphical widget that has always been missing for web developers, often conventionally called *treeView* or *treeWidget*. With treeViews, it is possible to graphically represent data that inherently have a tree structure as, for example, a file system or a family tree. The tool supports drag and drop of tree nodes onto other tree nodes, node collapsing and expanding, customization of the look and feel of nodes and their icons, as well as keyboard navigation [**jsTree**]. Furthermore, it is possible give nodes different types and constraints. For example, one may decide that nodes of type T may not contain nodes of type Q or that nodes of type X must contain at least a node of type Y. This can be done by manually set handlers for native jsTree events, which are triggered every time a node is modified, created or deleted.

Node labels can be edited, created and deleted inline. One huge drawback of using this tool is the lack of a complete documentation, which is rather inadequate, and of a large community. jsTree has been used for the *visual programming* part of the project, that is for the graphic representation of the DSL within the editor. In fact, as said before, scripts inherently have a tree structure as they have a main root block that can contain an arbitrary amount of other blocks. Moreover, the explicit support for node icons and types has proven very appropriate to the project, as the DSL has different types of expressions. We will see that some expressions of the DSL act as containers for other expressions, while some other are terminal and cannot be further expanded. This means that the graphical tree representation of a script must consider that certain nodes have to be leaves, while others may be leaves or subtrees. Thanks to jsTree's support for node constraints, this behavior was easily implemented in the editor, and the association of a node type to a well-determined icon is an advantage to the users that can intuitively understand the meaning of a node.

4.2.2 inputfile.js

Another simple jQuery plugin that provides, as its name clearly remarks, replacement for the file input element of the DOM. The plugin has been used because input fields for files are well known for being rendered differently depending on the system language and being impossible to customize. *inputfile*

hides the original input fields and replaces them with standard elements that can be arbitrarily edited and that do not depend on the system language.

4.3 Bootstrap

Bootstrap, developed internally by *Twitter*, is an open source framework that provides features for responsive web design, grid-based layout, a set of basic HTML elements that have been styled to allow for easy enhancement via classes and user styles, as well a set of modern layout components as drop-down menus, button groups, navbars, panels, breadcrumbs and many others [13]. Its simplicity to create animated interfaces and to manage layouts has been extremely useful and has cut down on the time of realization of the project.

Thanks to Bootstrap, web developers can actually count on GUI elements that were normally present in *local* frameworks such as *Qt*, but that had to be implemented from scratch for web pages, thus requiring much time. Furthermore, web pages made with Bootstrap can adapt to the device with which they are visualized. As every browser has its own renderer, web pages generally appear different depending on which browser is viewing it. Pages created with Bootstrap instead get equally rendered in different browser, thanks to its heavy CSS utilization. Nonetheless, users can customize the build to suit their own needs.

Bootstrap's grid system for the page layout management has proven extremely efficient and has a practically null learning curve. In general, subdividing a page in rows and columns, possibly of different length and with other nested rows and columns, has always been a time consuming task for web developers which besides required a lot of knowledge and manipulation of CSS styles, *float* attributes and so on. Moreover, the task had to be repeated several times to guarantee that smaller devices as smartphones and tablets would correctly display the page, often with an *ad hoc* layout specific for different classes of devices.

The grid system provided by Bootstrap manages to make this process extremely quick and adaptive to different devices. The system is based on two different classes of <*div*> elements, that is the *.row* class and the *.col* class. A <*div*> element with class *row* must be placed inside another <*div*> with class *.container* or *.container-fluid*. A row can be subdivided into at most twelve parts of the same size, and a column within the row may occupy from one to twelve parts. The sum of the parts occupied by the columns must not exceed twelve. A column is a <*div*> element with at least one class with the following prefixes:

- *.col-xs-*, for extra-small devices such as smartphones

- *.col-sm-*, for small devices such as tablets
- *.col-md-*, for medium desktop devices
- *.col-lg-*, for large desktop devices

Prefixes are followed by an integer ranging from 1 to 9, which indicates how many spaces the column occupies. Columns must be contained inside of rows and rows can be nested inside of columns. When a * * element has more than one column class with different prefixes, the browser will properly render the column for the device that is currently displaying the page. Bootstrap's grid system has been used for the layout of the static part of the editor's page.

Another nice component offered by Bootstrap that has been used for this work is the *navbar* component, which is a ready-to-use navigation bar, very popular in modern web pages, especially in one-page web applications. Navbars help users through the navigation of the pages and are generally fixed at one side of the page. Bootstrap's navbars are responsive and adaptive, thus can collapse and expand.

Bootstrap comes with several jQuery plugins that add new components to the already vast amount of provided services. Moreover it has a huge community where developers can publish their projects, share information and require support.

4.3.1 X-editable

X-editable is a library based on Bootstrap that provides interactive popups that appear when editing certain strings as well as validation functions for editable data. It has been used not only to give a fresh and modern look to the web page, but also to validate some important text data contained in the editor. Precisely, every string that represents a unique identifier of a game entity can be modified by typing the new string inside of the popup. Of course, the popup will prompt an error to the user if it receives as input an already reserved identifier, an empty string or a string that contains illegal characters.

4.4 evReact

evReact is a framework that provides features for the exploitation of the *reactive programming* paradigm. It was developed by Andrea Canciani at the University of Pisa, originally in F#, and then ported to JavaScript. In evReact, expressions define event-driven systems. It is possible to define event-driven systems of arbitrary complexity by composing evReact expressions as sub-terms. Expressions can be active or inactive. Active expressions can complete and/or terminate. When an active expression terminates, it becomes inactive. *Completion* conceptually means that the expression has successfully listened to all the events it was waiting for, while *termination* means that the expression stops listening for events. In particular, evReact provides the following constructs:

- *SimpleExpr* is an atomic expression to which is associated an event and a predicate. This expression completes and terminates when the associated event is triggered and the predicate is true.

- *cat* is an operator used to compose a sorted list of expressions. Each subexpression is activated when the previous one completes. The whole expression completes when the last expression of the list completes, and terminates when every expression within the list is inactive.
  Let A and B be two expressions: we will write ***A ; B*** to indicate *cat(A, B)*.

- *any* is an operator used to compose a list of expressions. This expression completes when at least one expression of the list completes, and terminates when all of them become inactive.
  Let A and B be two expressions: we will write ***A | B*** to indicate *any(A, B)*.

- *all* is an operator used to compose a list of expressions. This expression completes when every expression of the list has completed, regardless of the order of completion. It terminates when every expression within the list becomes inactive.
  Let A and B be two expressions: we will write ***A & B*** to indicate *all(A, B)*.

- *iter* is an operator that forces its subexpression to restart listening for events once it has completed.
  Let A be an expression: we will write $^{+}$***(A)*** to indicate *iter(A)*

- *react* is an operator used to combine a subexpression and a callback

function. The expression completes and invokes the callback when the subexpression completes. It terminates when the subexpression terminates.
Let B be a subexpression and f a callback function. We will write ***B |-> f*** to indicate *react(B, f)*.

- *finally* is an operator very similar to *react*, with the only difference that the callback is invoked when the subexpression terminates (thus completion is not necessary). In this case, we use the annotation ***B |=> f***.
- *restrict* is an operator that combines a subexpression and a list of events. It is useful to force the termination of the subexpression when at least one of the events of the list has been ignored.
  Let A be an expression and L a list of events. We write ***A \ L*** to indicate *restrict(A, L)*.
- *cond* is an expression that combines a subexpression and a predicate. It will complete when the subexpression completes and the predicate is true.
  Let A be a subexpression and let P be a predicate. We will write ***A ? P*** to indicate *cond(A, P)*.

4.4.0.1 Petri nets

evReact is based on Petri nets, a model to describe distributed systems that are parallel and asynchronous. Petri nets are both a graphical model and a mathematical model. A Petri net is a a collection of *places*, *arcs* and *transitions*. Places are usually represented as circles, while transitions are represented as a bar. Arcs are directed and may connect either places to transitions or transitions to places, that is a Petri net is formally a bipartite graph with places and transitions that alternate on a path made up of consecutive arcs [14]. It is required that each arc has a node (a place or a transition) at both of its ends.

In a Petri net, the state of the system is represented by means of *tokens* or *marks*. Let P_{i} be a generic place of a Petri net PN and let the place marking $m_{\mathrm{i}} \geq 0$ indicate the amount of tokens contained in P_{i}, with $i = 0, \ldots, n-1$. The state of the system is, more precisely, the marking of the net m, that is the vector of the place markings $m = (m_0, \ldots, m_{\mathrm{n\text{-}1}})$. In case of *ordinary* Petri nets, arcs are not weighted, thus $m_{\mathrm{i}} \in \{0, 1\}\ \forall i = 0, \ldots, n-1$. evReact is based on ordinary Petri nets. The state will evolve as transitions are fired. Let the arc A_{ij} connect the place P_{i} to a transition T_{j}, and let the

arc A_{jk} connect T_j to the place P_k. In this case P_i is called *input* of T_j and P_k is called *output* of T_j . A transition with no input is called *source transition* and a transition with no output is called *sink transition*. In order to be fired, a transition needs to be enabled, that is all its input places must contain a token (or n tokens in case of arcs of weight n). Of course, source transitions are always enabled. When a transition is fired, every input place of the transition will have its token removed and a token will be added to every output place of the transition. More importantly, the process of firing a transition is indivisible.

4.5 Other JavaScript libraries

This section includes other stand-alone JavaScript libraries that have been useful to the project and which do not extend nor act as plugins for any other famous framework.

4.5.1 Snap.svg

Snap.svg is a JavaScript library which provides APIs for the manipulation and the animation of SVG content. It allows to work with already existing SVG data, differently from other similar frameworks which require that the data they work with must be generated by them. Every SVG element (that is, every graphic element) of the authored games has been treated with Snap. In particular, its utilities for the fast definition of polygons by simply passing an array of points as parameter, has been convenient and useful.

4.5.2 Paper.js

Paper.js is a framework for vector graphics scripting built on top of the HTML5 canvas. It provides a *scene graph* and a lot of features for the manipulation of geometric primitives. Actually, only a small part of Paper.js was used for the project, that is the utilities for bounding box calculations. Such utilities have been used for the *pick correlation* of the objects placed inside the editor canvas.

5 Design

5.1 The Domain-Specific Language

The design of a DSL must be based on a previous analysis of several issues. First of all, the *purpose* of the language must be very clear, so that the problem's domain can be well identified and problems *outside* of it can be easily excluded from the set of problems that really are interesting. Moreover it must be clear in advance to the designers, at least broadly, the kind of *syntax* and notations used by the language, as well as the programming paradigm (object-oriented, functional, etc.) that would best fit the treatment of the problem.

The purpose, though apparently very clear, had to be refined in order to restrict the generic term "graphic adventure" to a smaller set of existent graphic adventures that would become the *quality target* for the features they include and the approach they adopt. We chose to take LucasArts' graphic adventure games as reference model for their renowned quality, expressiveness and complexity.

This choice led to the definition of the minimal set of features that, combined to each other, would be able to emulate those games.
Some of these features are highly dynamic and must be easily treatable by the DSL. On the other hand, some other features are static and don't need to be explicitly supported by the language.
The set allowed to define the following specifications.

The game is modeled as a finite-state automaton, onto which is defined a

transition function and an *output* function. Each user input may or may not change the state of the game, and will surely result in some output. State transitions may be triggered by user inputs or by timing: factorizing this assumption, we decided that state transitions may only be triggered by *events*, which in turn can be triggered by user inputs and timing. Therefore, the language must provide easy ways to *fire* game events, to *wait* for them and to react to them in some way. Outputs consist of *computations* of core methods, that will be conventionally called *side effects*. Side effects can have different types and can affect graphics, properties or global variables.. These properties of the game entities can arbitrarily change, such as their spacial position, visibility, description and appearance. The language provides, as we well see later on, proper support for the manipulation of these properties. Game entities also have the ability to *speak* and to *walk* following a path. Playing characters must have an *inventory* that stores the objects that can be picked up during the game.

As for the programming paradigm, since the language is oriented to non-programmers, it was decided to make it as simple as possible: imperative, non-procedural and non-object-oriented. These considerations are the base for the context-free grammar described in the next paragraph.

5.1.1 Grammar

The following context-free grammar describes the structure of the implemented DSL:

⟨*Script*⟩ ::= ⟨*Statement*⟩

⟨*Statement*⟩ ::= ⟨*Aggregator*⟩ { ⟨*Statement*⟩* }
| if(⟨*BoolExpr*⟩) do ⟨*Statement*⟩
| SideEffect

⟨*SideEffect*⟩ ::= ⟨*Atomic*⟩
| ⟨*Interruptible*⟩

⟨*Aggregator*⟩ ::= Any
| All
| Seq

⟨*Atomic*⟩ ::= varSet(⟨*ID*⟩, ⟨*Expr*⟩)
| varIncr(⟨*ID*⟩, ⟨*Number*⟩)
| fireEvent(⟨*String*⟩)
| waitEvent(⟨*String*⟩)

| setPosition(⟨*ID*⟩, ⟨*Number*⟩, ⟨*Number*⟩)
| setDirection(⟨*Dir*⟩)
| show(⟨*ID*⟩)
| hide(⟨*ID*⟩)
| inventoryAdd(⟨*ID*⟩)
| inventoryRemove(⟨*ID*⟩)

⟨*Interruptible*⟩ ::= sayLine(⟨*ID*⟩, ⟨*Sting*⟩)
| walktoPos(⟨*ID*⟩, ⟨*Number*⟩, ⟨*Number*⟩)
| walkToObj(⟨*ID*⟩, ⟨*ID*⟩)

⟨*Dir*⟩ ::= Left
| Right
| Front
| Back
| FrontLeft
| BackLeft
| FrontRight
| BackRight

⟨*Expr*⟩ ::= BoolExpr
| NumExpr

⟨*BoolExpr*⟩ ::= (⟨*BoolExpr*⟩)
| true
| false
| ⟨*ID*⟩
| ⟨*BoolExpr*⟩ and ⟨*BoolExpr*⟩
| ⟨*BoolExpr*⟩ or ⟨*BoolExpr*⟩
| not ⟨*BoolExpr*⟩
| ⟨*NumExpr*⟩ > ⟨*NumExpr*⟩
| ⟨*NumExpr*⟩ = ⟨*NumExpr*⟩
| ⟨*NumExpr*⟩ < ⟨*NumExpr*⟩

⟨*NumExpr*⟩ ::= (⟨*NumExpr*⟩)
| ⟨*Number*⟩
| ⟨*ID*⟩
| ⟨*NumExpr*⟩ + ⟨*NumExpr*⟩
| ⟨*NumExpr*⟩ − ⟨*NumExpr*⟩
| ⟨*NumExpr*⟩ ∗ ⟨*NumExpr*⟩
| ⟨*NumExpr*⟩ / ⟨*NumExpr*⟩

5.1.2 Parsing

The scripts definition by the user follows the *visual programming* approach and is performed by dragging and dropping valid blocks of code. More formally, the user interface allows only to perform actions that correspond to the production rules of the language, i.e. the above grammar is used as a *generative grammar* for the definition of scripts. Whenever the productions lead to one or more nonterminal symbols, the user interface provides an appropriate default choice. This guarantees that users are allowed to only define scripts that are syntactically correct. Actually, keyboard typing is needed for the definition of boolean and numeric expressions: such decision was taken out of time constraints, but it would be desirable, in the future, to extend the graphical approach even to the guards. The parsing of the *if* guards has been implemented by a recursive descent parser that, taken as input the string representing the boolean expression to check, returns either a parse tree of the valid expression or false in case of string rejected.

5.1.3 Scripts and Run Time Support

5.1.3.1 Header

Scripts consist of a header and a body. While the body is a valid program written in the DSL just described, a header contains the event chains that will trigger its execution. Such event chains, or triggerers, can conceptually belong to three different types:

- *user-interaction* triggerers, that is chains of events fired in response of user input
- *event-occurrence* triggerers, that is chains of events programmatically fired by another user-defined script
- *timer* triggerers, that is chains of events fired at regular time intervals defined by the user

The script compiler receives a valid script as input and builds up an evReact expression that interprets its behavior. Headers are compiled as an *any* expression, meaning any chain of events contained inside the expression itself will be able to trigger the compiled body.

User-interaction triggerers are *cat* expressions: by design choice, the event chain for each of these triggerers consists of at least one and no more than

three events. The first event can be a natural mouse event, like a simple click, or an Action event belonging to the set of actions the player is allowed to perform while playing. In the second case, the expression waits for a sequence of one or two Object events, depending on the designer's choice: for example, *cat([Push, Button])* or *cat([Use, TV])* are perfectly legit triggerers, as well as *cat([Use, Batteries, Radio])*, which implies a combination of one Action and two Objects.

This pattern has been adopted for event-occurrence triggerers too, but as said before, the nature of events for this kind of triggerer is slightly different, since it does not depend on user input but on events explicitly fired by a script's body, for example to notify that a certain part of the plot has been completed and that the game state has to be updated in some way.

Timer triggerers consist of a simple evReact event that is fired at a regular time interval expressed in milliseconds by the user.

Another interesting and important aspect to consider is that a player might possibly change his mind while firing an action event, consequently firing two or more consecutive action events: it is clear that a coherent reaction to this use case is to immediately terminate the expressions that did not ignore events different from the last one triggered. As aforementioned, evReact provides the *restrict* operator to force an expression to terminate. Said considerations apply for the other two types of triggering as well.

Taking everything into account, the output for the headers' compiler is an evReact expression of the form:

$$\begin{aligned} &\mathit{any}([\mathit{restrict}(\mathit{cat}([\text{inputEv}_1, \ldots, \text{inputEv}_n]), [\text{restr}_1, \ldots, \text{restr}_n]),\\ &\mathit{restrict}(\mathit{cat}([\text{scriptedEv}_1, \ldots, \text{scriptedEv}_n]), [\text{restr}_1, \ldots, \text{restr}_n]),\\ &\quad \mathit{restrict}(\mathit{any}([\text{timeEv}_1, \ldots, \text{timeEv}_n)], [\text{restr}_1, \ldots, \text{restr}_n])]) \end{aligned}$$

5.1.3.2 Body

The script body consists of a single statement belonging to the DSL. The statements have been categorized into two main classes, that is *controllers*, which have statements as subterms, and *side effects*. Statements have a syntactically recursive structure, and their semantics is compositional: the semantics of an expression can be obtained from the semantics of its subexpressions and the semantics of the operator that is used to compose them, without any need to inspect their syntax. This kind of semantics is desirable for many reasons, as it makes it easier to understand the behavior of complex

expressions. Moreover, it is easily evaluated recursively by computing it on subexpressions and combining them as required depending on the operator.

Controllers are subclassed into *Aggregator* controllers and *If* controllers: Aggregators can have Sequence, Any or All type and semantically correspond to their aforementioned evReact counterparts. *If* controllers semantically correspond to *if* expressions in JavaScript. An aggregator works as a container of code and can be nested inside other aggregators.

Side effects represent the core features offered by the engine. *Atomic* side effects are fully executed instantly and their interpretation cannot be stopped once it started. Every side effect but *walkToPos*, *walkToObj* and *sayLine* belong to this class of side effects.

Interruptible side effects require a certain amount of time to be fully interpreted, and their execution can be stopped before completion.

In detail, the explanation of the semantics of all the available side effects is the following:

- *walkToPos(ID, xPos, yPos)* makes the game entity univocally identified by *ID* walk along the shortest path with starting point defined by its actual screen position and destination point (*xPos*, *yPos*). The side effect reaches completion when the position of the game entity equals the destination point.
- *walkToObj(entityID, objID)* makes the game entity univocally identified by *entityID* walk along the shortest path with starting point defined by its actual screen position and destination point defined by the property *walkingPoint* (set manually by the user from the graphical editor) of the game object univocally identified by *objID*. The side effect reaches completion when the position of the game entity equals the destination point.
- *sayLine(ID, sentence)* draws the *sentence* string over the game entity univocally identified by ID. The side effect reaches completion after the string has been on screen for a fixed time interval (150 ms).
- *delay(time)* delays the execution of the rest of the body of a lapse of time in milliseconds specified by the *time* argument.
- *waitEvent(eventID)* blocks the execution of the rest of the body until the event univocally identified by *eventID* is triggered.
- *fireEvent(eventID)* triggers the event univocally identified by *eventID*.

- *setRoom(roomID)* sets the location univocally identified by *roomID* as the current game location.
- *setDirection(characterID, dir)* makes the character univocally identified by *characterID* face the direction specified by the *dir* argument.
- *setPosition(ID, xPos, yPos)* places the game entity univocally identified by *ID* onto the screen coordinates (*xPos*, *yPos*).
- *show(ID)* sets as visible the game entity univocally identified by *ID* and its relative *hotspot*, if present.
- *hide(ID)* is dual to *show(ID)*.
- *inventoryAdd(invItemID)* adds the inventory object univocally identified by *invItemID* to the player's inventory.
- *inventoryRemove(invItemID)* is dual to *inventoryAdd(invItemID)*.
- *varSet(ID, val)* sets the game variable univocally identified by *ID* to the value specified by the *val* argument.
- *varIncr(ID, amount)* increments the numeric variable univocally identified by *ID* by the value specified by *amount*.

5.2 The Editor

The graphical editor is composed of different sections, each of them specialized in one particular aspect of the game. Each section is accessed by the relative button within the navigation bar, fixed on top of the web page. In turn, the navigation bar includes a *dropdown* menu for the management of projects, as seen in figure:

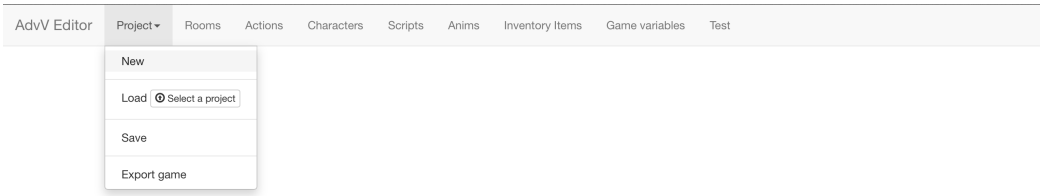

Figure 5.1: The *Anims* section

5.2.1 The Rooms section

The rooms section allows users to define new game locations, conventionally called *rooms*. Figure 5.2 shows the appearance of the section to the users. The section contains a single graphic panel which acts as a *container* for instantiated rooms, as well as a canvas that shows user-defined room properties and a canvas toolbar (*2.*) to choose which property to manipulate. The canvas and its toolbar are initially hidden. A small toolbar (*1.*) on top of the container panel permits to *add*, *delete* and *clear all* rooms.

When a user adds a new room, a subpanel associated to such room is added to the main container panel. A subpanel can be expanded and collapsed by a button located onto its header (*3.*) and, in case of multiple subpanels, the expansion of a subpanel triggers the collapsing of all of the others. The generic *click* on a subpanel's header sets such subpanel as *active*. A subpanel's header also contains a *title* (*4.*) which corresponds to the unique identifier assigned to the room with which it is associated. The expansion of a subpanel reveals its body, which in turn contains two graphical components for the customization of the room. The first component (*5.*) is a simple button which serves as a selector for the room's background image. When a local image is chosen for the room background, the canvas and its toolbar are

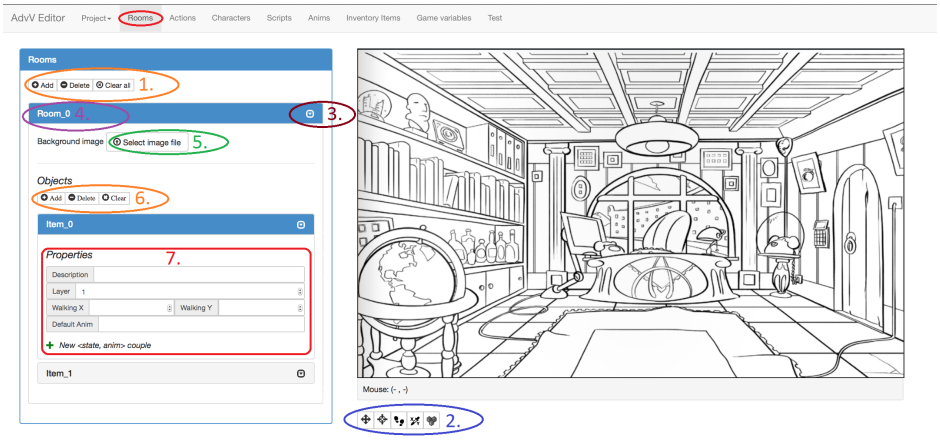

Figure 5.2: The *Rooms* section

shown and the background image is printed inside the canvas. The second component (*6.*) is another toolbar to manage the addition and the removal of objects to the room. Again, the addition of a room object by means of the *add* button of this second toolbar, involves the addition of an *object panel* to the current room subpanel.

Object panels behave the same way of room subpanels for what concerns expanding, collapsing and the title/identifier correspondence. An object panel's body (*7.*) contains several input fields to define properties of the object it is associated to. For a given room object, there are input fields for the definition of, in order:

- its on-screen *description*;
- its graphic *layer*;
- its "walking spot", that is the screen position to reach when walking to the object;
- its default *Anim* state *(see the Anims section)*;
- an arbitrary set of (*state*, *Anim*) couples;

As aforementioned, the canvas and its toolbar (*2.*) become visible as soon as the user chooses a background image for the currently selected room. The canvas toolbar consists of five buttons, each one of which changes the canvas state and its reaction to mouse events, in order to define more object properties for which simple input fields would be inconvenient to users.

Starting from the left, the first button allows users to change the position

of objects by clicking on their placeholder image printed inside the canvas over the background and dragging them around the room with the mouse, dropping the image when the desired position has been reached.

The second button permits to define an object ***hotspot***, described as a list of vertices that form a polygon. A vertex is added to the polygon by clicking inside the canvas. The polygon is closed by clicking a second time over the first vertex added to the list. The polygon can be cleared and the process started over by right-clicking inside the canvas. The figure below shows the process of definition of a polygon around a figure:

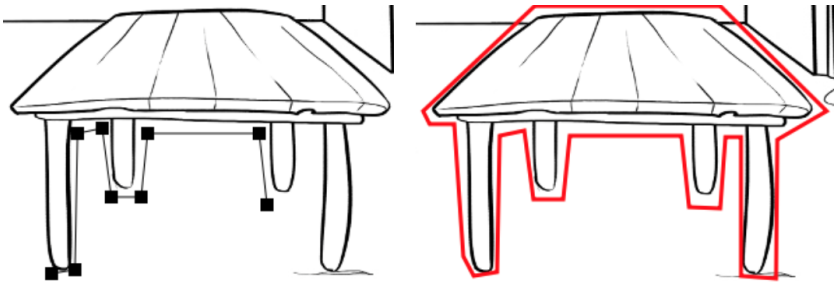

Figure 5.3: Adding vertices (left) and the closed polygon (right)

By pushing the third button, users can define the room's *walkable path* as a polygon in the same way hotspots are defined. Once the polygon has been created, it is possible to add holes to it, by keeping pressed the left shift key and tracing other polygons which will be considered as holes. Right-clicking on the canvas has the effect of clearing every polygon, both the walkable path and its holes, if any.

The fourth button is there for mere convenience, as it allows to save clicked coordinates of the canvas into a string displayed under the canvas itself. The fifth and last button allows to associate, for a given object, its *walk-behind* line. This straight, horizontal line defines the y on-screen coordinate such that, if a character's bottom y coordinate is lower than this line, then the character will be conceptually considered *behind* the object and thus will have a lower z-order (right hand rule).

5.2.2 The Characters section

The *characters* section has a simple panel which acts as container for user-defined characters. It contains another toolbar for creating and removing game characters, in the same way described for rooms and room items.

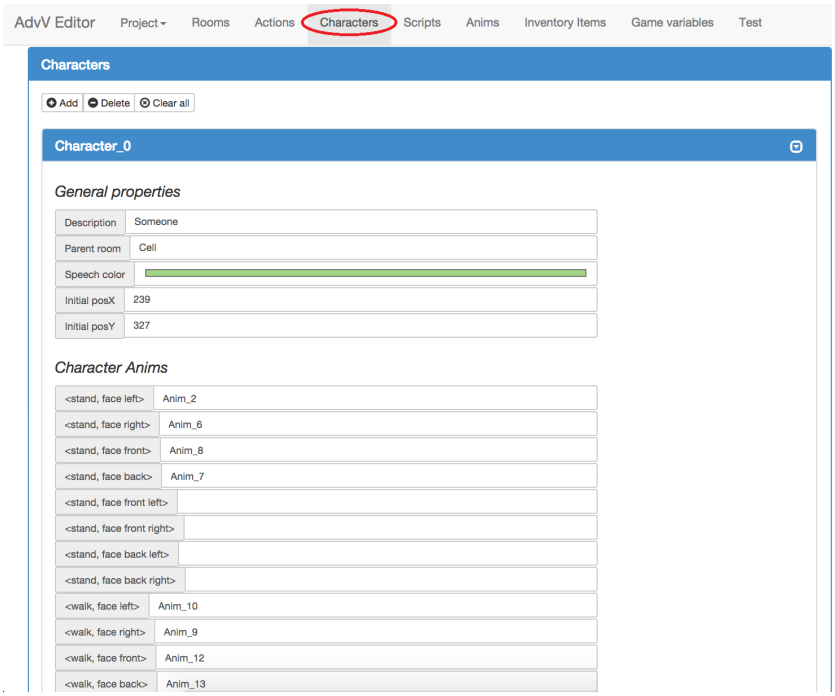

Figure 5.4: The *Characters* section

A character panel contains several input fields for the definition of character properties, which in turn are divided into *general* properties and *animation* properties.

General properties consist of the character's on-screen *description*, its *parent room*'s identifier, its *speech color* (which defines the color of the sentences possibly said by the character) and its initial on-screen *position* within its parent room.

Animation properties are a list of (*state*, *direction*, *Anim_ID*) triples, where the default state $s \in \{$ *stand*, *talk*, *walk* $\}$ and the generic direction d belongs to the set of admissible directions already specified by the DSL grammar in 5.1.1 while the third element of the triple is simply a unique identifier for an *Anim* instance. In addition to the three default states, users can define

custom states to associate to *Anim* identifiers, with no limits on the amount of states, by clicking the specific button on the panel bottom.

5.2.3 The Scripts section

The following figure shows how the *Scripts* page appears to the users:

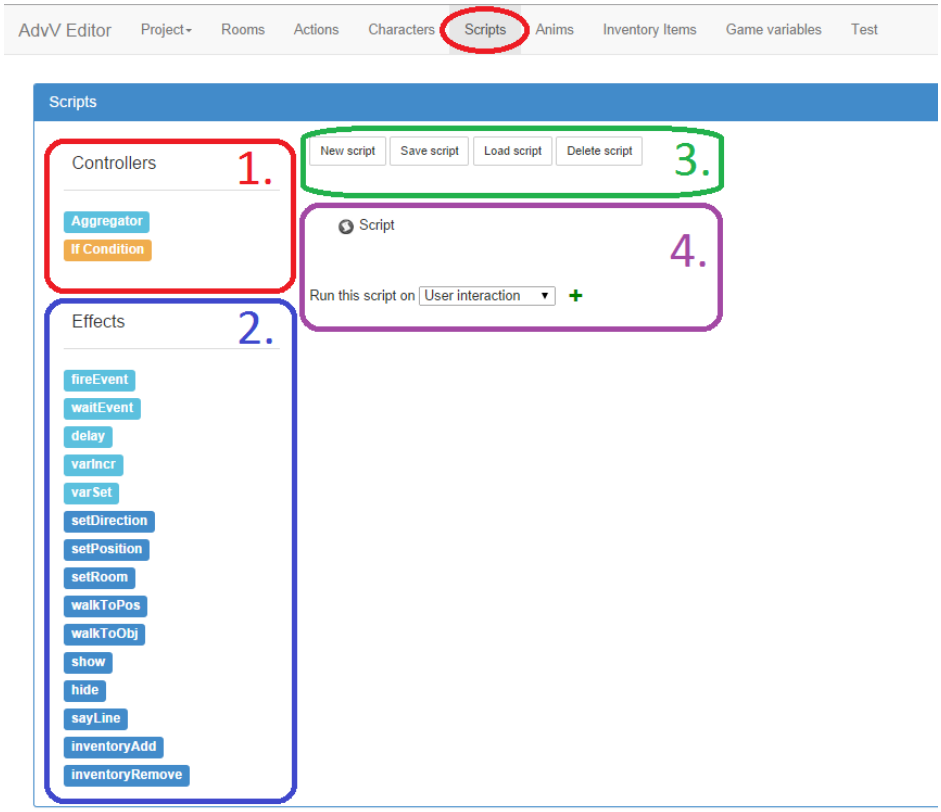

Figure 5.5: The *Scripts* section

1. and *2.* respectively contain all the *controllers* and all the *side effects* already described in 5.1.3.2. Each one of them can be added to the current script by dragging and dropping it onto the *script root* contained inside *4* or within a controller which is already present within the script. The toolbar in *3.* allows to define new scripts, loading existing scripts, deleting and saving them. Besides the script root, *4.* contains a dedicated button to add new triggerers and a dropdown menu to specify their type.

5.2.4 The Anims section

By the usual toolbar, users can create new *Anims*. An open Anim panel can be seen in the figure below:

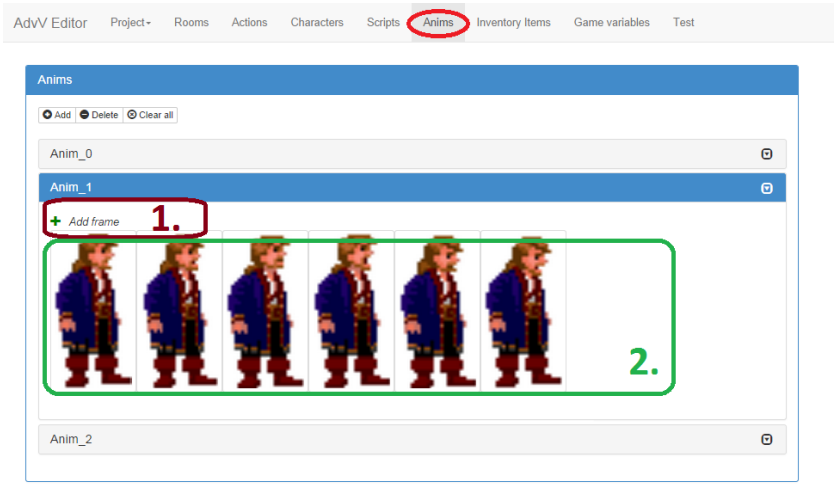

Figure 5.6: The *Anims* section

2. highlights the set of frames that compose the Anim. Each of these frames is added by clicking *1.* and can be removed by right clicking on it.

5.2.5 The Inventory Items section

As shown in the figure below, an *Inventory Item*'s panel consists of an input field for specifying its on-screen description and another for associating it to an existent Anim identifier.

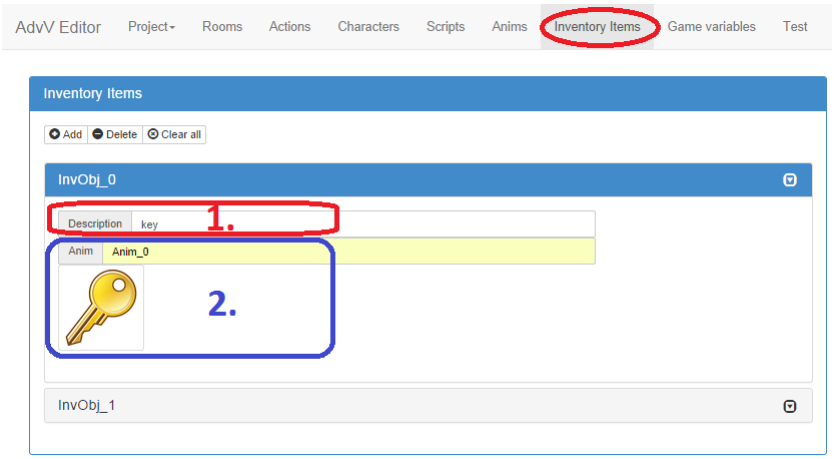

Figure 5.7: The *Inventory Items* section

5.2.6 The Game Variables section

The next figure shows the *Game Variables* section:

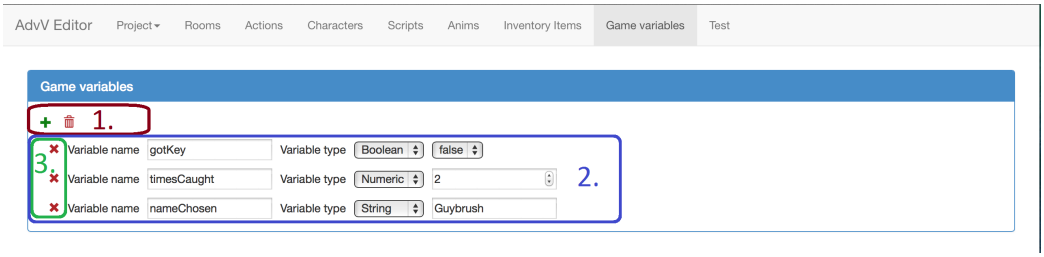

Figure 5.8: The *Game Variables* section

The main panel contains two icons (*1.*) to add new variables and to clear them all. Variables can be boolean, numeric and strings (*2.*). Each variable added can be removed by its relative icon for deletion (*3.*).

5.2.7 The Test section

This section is a simple wrapper of the game, useful to test it before exporting it as a simple HTML page. The *WYSIWYG* (what you see is what you get) philosophy of the entire editor can be clearly seen here, as wrapped games and exported games will be absolutely equivalent. The test can be started by clicking (*1.*).

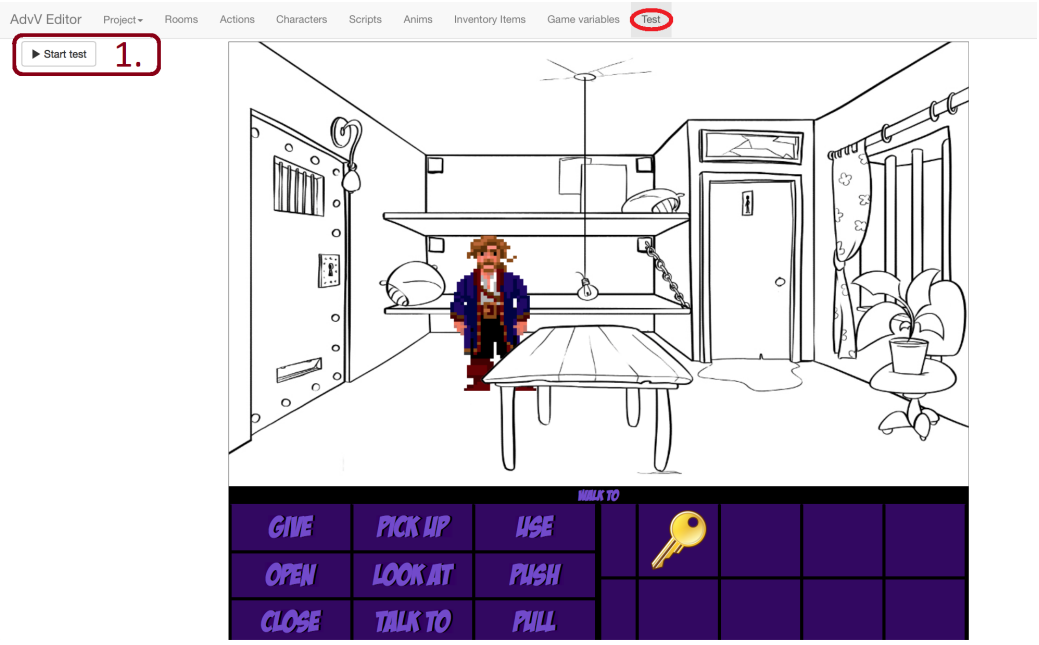

Figure 5.9: The *Test* section

5.3 Data Architecture

The editor just described is a mere way to graphically define properties of the various game components, but the interactions and behaviors of such components are yet to be described. In the following we will see a description of the system architecture, starting with a very high abstraction level and then proceeding to a more detailed illustration.

First of all, let us consider a game authored by the system: its high-level structure is very simple, as it simply consists of a graphic manager and a logic manager that interact with the rooms data, as can be seen in figure:

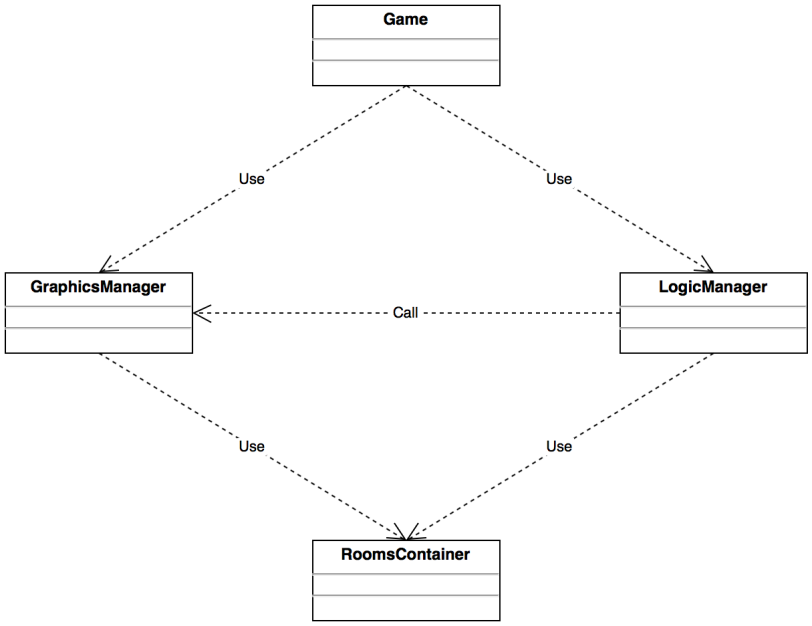

Figure 5.10: Abstract game architecture

Obviously, the graphic manager takes care of rendering the game entities on screen, while the logic manager is delegated to manipulate their properties and to call the graphics manager when necessary.

Let us now see the structure of these entities:

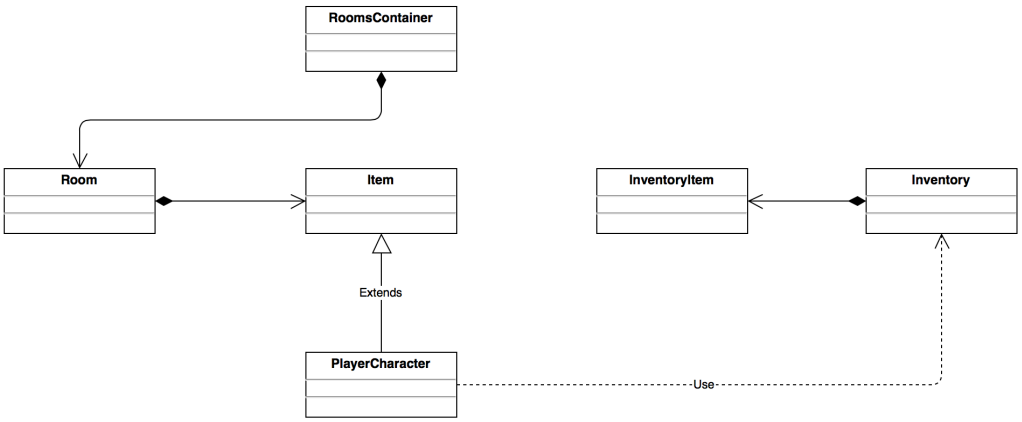

Figure 5.11: Logic game entities and their behavior

The *Room* class represents generic game locations, not necessarily in a strict sense: for example, a *Room* instance can be a close-up of an object, an open environment, a closed room. We can say that a *Room* represents a scene of the game that has a fixed background image and may or may not contain other game entities such as objects and characters. A *Room* instance may contain several *Item* instances, which represent general game entities that can *conceptually* behave as animated characters or as inanimate objects.

An *Item* class is an abstract representation of every logic entity in the game: player characters are as well a special case of *Item* and are indeed instances of the *PlayerCharacter* class, which extends *Item*. Since every player character has an inventory, the *PlayerCharacter* class uses the *Inventory* class. In turn, an *Inventory* instance acts as a container of *InventoryItem* instances.

The reason why generic items and inventory items are unrelated is that they're completely different from a conceptual point of view, that is the first are explicitly contained inside of a room, have a position that can be dynamically changed and can even be removed from their original room and placed into another one, while the latter can only be stored inside of a character's inventory, don't have a screen position and are simply defined just by a description and a graphic representation.

Any logic behavior is defined by scripts made within the editor. The scripts, written in the DSL, are then compiled into evReact expressions. The following figure illustrates the structure of the original scripts:

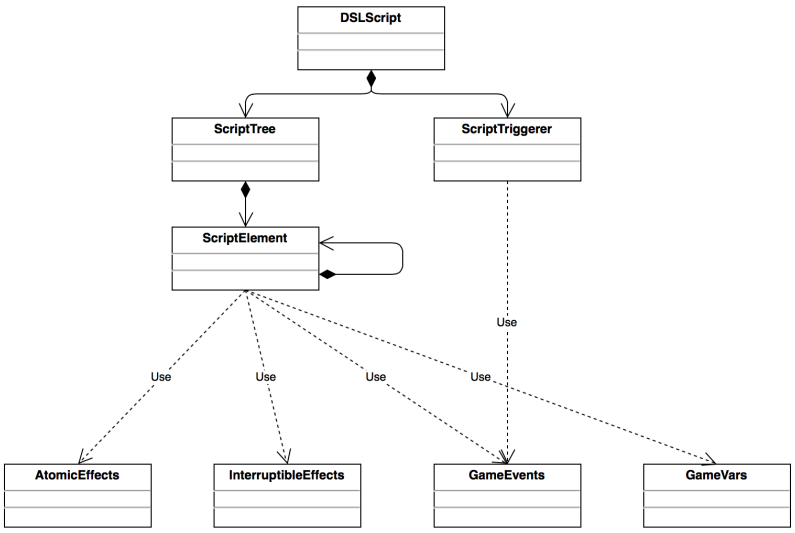

Figure 5.12: Structure of the DSL scripts

A script is represented by the *DSLScript* class, which in turn acts as a container for the script header, defined by the *ScriptTriggerer* class, and the script body, defined by the *ScriptTree* class. The latter describes the script as a tree structure, with its root being an instance of the *ScriptElement* class. This class represents a valid construct of the DSL, that is an aggregator or a side-effect, and may contain other instances of itself in the first case. A *ScriptElement* instance will make use of *syntactic representations* of the game's core data.
Syntactic representations refer to:

- side-effects functions, both atomic and interruptible, the first represented by the *AtomicEffects* class and the latter by the *InterruptibleEffects* class;
- every possible game event, represented by the *GameEvents* class;
- all the game variables, represented by the *GameVars* class.

The script header logically gathers ordered sequences of game events that will trigger the body execution, that is the *ScriptTriggerer* class uses the *GameEvents* class.

The scripts compiler, represented by the *ScriptCompiler* class, consists of two sub-compilers, one delegated to compile headers, represented by the *ScriptTriggererCompiler* class, and the other delegated to compile bodies, represented by the *ScriptTreeCompiler* class. A script compiler creates instances

of the EvreactExpr class, which represents a valid evReact expression that implements the behavior described by the DSL script, as shown in the figure below:

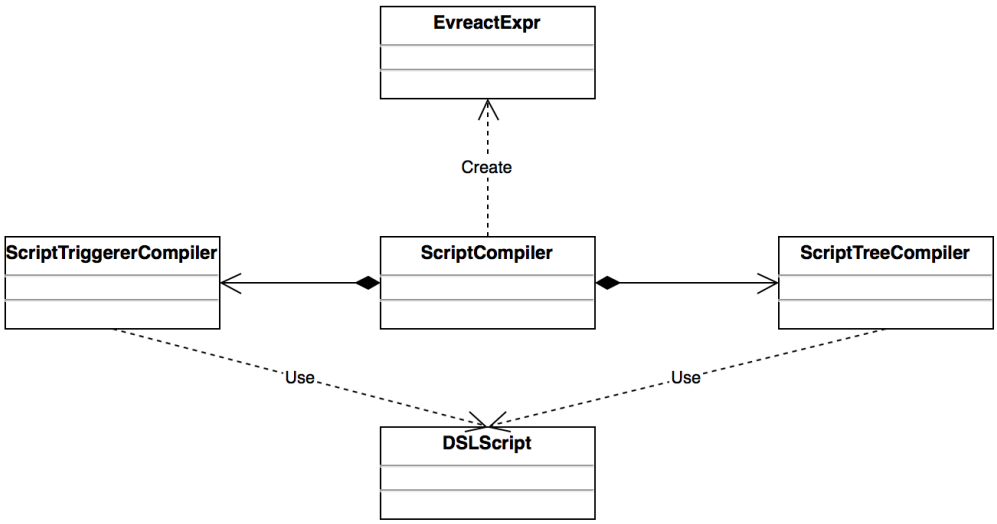

Figure 5.13: Compilation of the scripts

This class coincides with the generic *LogicManager* class mentioned earlier in this section. Its role is to *handle* every event that is fired throughout the game. The evReact networks are the only logic entities that may actually interact with the game core, which consists of the *interpreters* of the side-effects, both atomic and interruptible, the valid game events and the game variables.

By construction, only atomic side-effects may operate on the game variables, while both types of side-effects have access to the game events as they need them for synchronization. Both interpreters can make calls to the graphics manager to update the whole scene or parts of it. See the figure below:

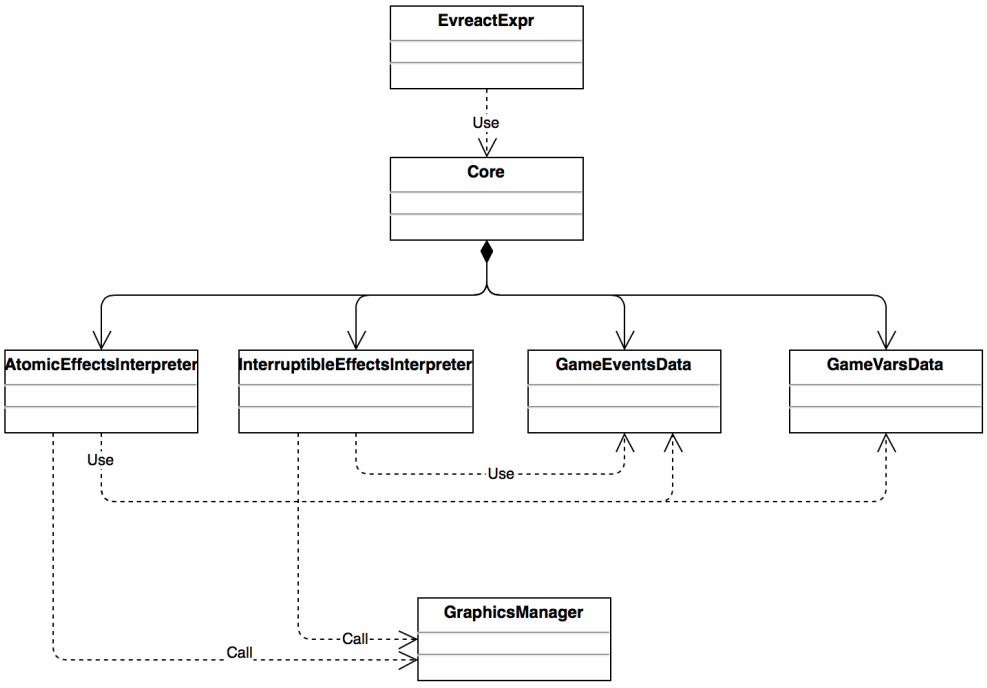

Figure 5.14: Interaction between evReact and the game core

While evReact networks handle events, the way user inputs can *fire* events is yet to be described. The most primitive event that can be fired by users is a simple mouse click. Every interactive entity of the game provides a listener for mouse events and, as it detects a click, it fires a more specific event, i.e. a valid and unique evReact event whose name matches its identifier. Let us now see how entities of very different nature can listen for click events and fire another event in response:

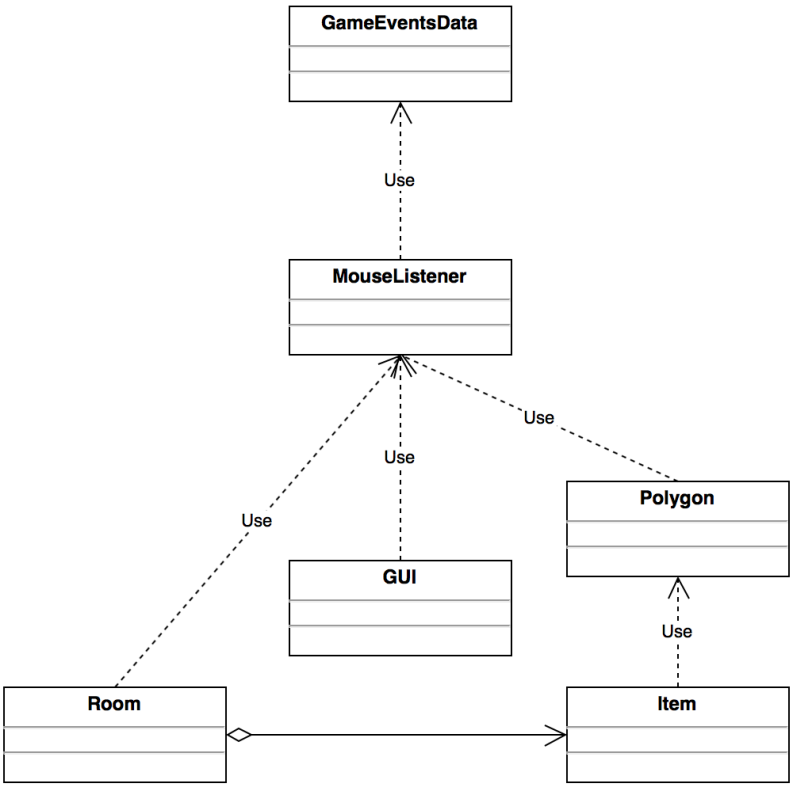

Figure 5.15: Mouse listeners for rooms, items and the GUI

Interactive items use a polygon to define the portion of space that must react to mouse inputs from users (**hotspots**). The *Polygon*, in turn, uses a *MouseListener*that, in response to a mouse event, fires the game event associated to the *Item* instance connected to the polygon.

A room's background image also has a mouse listener. Mouse clicks over a background will make its listener trigger a special event, i.e. a *background* event, which is a unique game event that will be handled contextually to the room where the click has occurred: if the room contains an area where the

player can walk, then the handler (precisely the *walkToPos* function) will compute the shortest path from its current position and the clicked spot and make him walk there. This computation will be described in Chapter 6.

As for the GUI, every cell that forms it has attached a mouse listener that will react by firing an appropriate game event depending on the part of the GUI that received the input, be it an action or an inventory object. Again, details on the implementation of this behavior will be provided in the next chapter.

6 Implementation

In this chapter, we will see in detail the implementation of the system. The project has been divided in several JavaScript modules, each one delegated to a particular part of the system.
The following sections conceptually divide the parts of the system and describe which modules contribute to each part.

6.1 Primitive structures

6.1.1 room-manager.js

The module called *room-manager.js* defines the most important entities of the system, that is rooms and room items. In this module are declared the global data structures that contain and map such entities, that is (variable names are self-explicative):

- *editorRoomsList[]*
- *editorMapIdRoom*{}
- *editorMapIdItem*{}
- *editorCharactersList*{}
- *editorMapIdCharacter*{}

Rooms are defined by the class *EditorRoom*, which is structured as shown below:

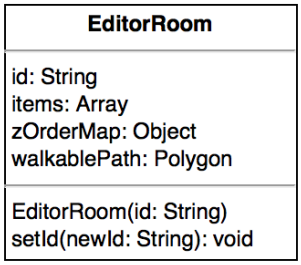

Figure 6.1

Each room has a unique identifier and the association between an *EditorRoom* instance and its *id* is stored in the relative global hashmap mentioned before. The *setId* method verifies that the new identifier for the room is unique among every other instance of *EditorRoom*. The *items* member is a heterogeneous Array such that *items[i]* stores, for $i = 0$, the base64 representation of its background image and $\forall i > 0$ an instance of the *EditorItem* class, which represents a generic item contained inside a room.

The *walkablePath* member is an SVG polygon that defines the area of the room where the playing character and the items are allowed to freely walk into. Finally, the *zOrderMap* class member is a hashmap that associates integer keys, representing a layer number, to Arrays of identifiers of *EditorItem* instances. This class represents both inanimated objects and alive characters, and has the following structure:

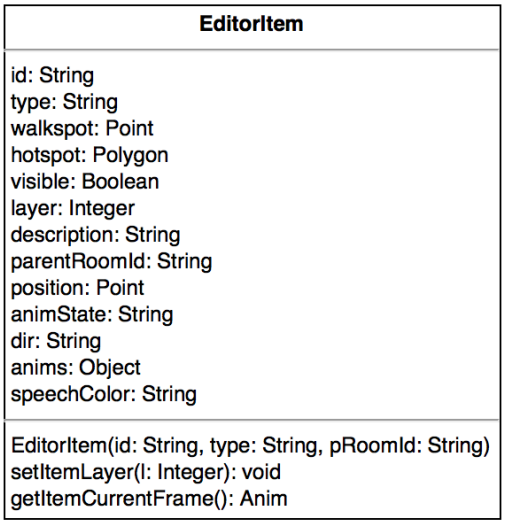

Figure 6.2

EditorItem instances are as well univocally identified by an *id* member and the associations between identifiers and *EditorItem* instances are stored in the relative global hashmap. As this is the standard behavior for most classes, from now on, whenever a class has an *id* field, it must be clear that the identifier is unique and that there is a global Object which acts as an id/instance hashmap. Therefore, only the name of the global Object will be specified.

It is important to notice that the uniqueness of identifiers is necessary and guaranteed only among instances of the same class, while instances of different classes may have the same identifier with no problem at all. The module contains a global function, $checkIdUniqueness(id, type)$, which checks for id conflicts among instances of the class described by the *type* parameter. Such parameter is a string belonging to the set { *"room", "item", "action", "character", "inventory-item"* }.

The *type* class member can either be "object" or "character". The *walkspot* member specifies the spacial coordinates on which the playing character must be to have conceptually "reached" the item. *hotspot* is a SVG polygon that pinpoints the spacial area of the room that listens to the player's mouse events and thus allows interaction with the item.

The *visible* member specifies whether the graphical representation of the item (and its *hotspot*) must be shown on screen or not. The *layer* member defines the z-order of the item on the scene: we will later see what are the effects of the manipulation of a layer. As mentioned earlier, the *description* member is a string that briefly describes the item and that will be printed inside the apposite section of the GUI when the item detects a mouse event. As one item can be contained inside at most a room at a given time, the member *parentRoomId* contains the identifier of the room that currently contains the item. The *position* member is a point with spacial coordinates that refer to the top-left position of the current graphic representation of the item, if one exists.

We will later see in the *anim-manager.js* section that every graphic component but the rooms' backgrounds is defined by a class called *Anim*. Avoiding for now to dive into details, it is enough to know that the *anims* member of the *EditorItem* class is a hashmap that associates possible states of the item to unique identifiers of the existing instances of *Anim*. As seen in the previous chapter, both inanimate objects and characters can have ***custom*** states that are associated to *Anim* instances by *<state, animId>* couples. Nonetheless, the default states of objects and characters are different: at construction time, the *type* parameter will determine the default state(s) of the item, which in turn will be keys of the *anims* class member. That is, if

type is "object" then the item will only have a default state precisely called *default*, but if type is "character" then the default states will be more than one and more specialized. Indeed, the item's default states will be *stand*, *walk* and *talk*, and each of these key states will have as value not an *Anim* identifier but an additional hashmap. Such additional hashmap contains eight keys, one for each of the cardinal points, and will have as value the identifiers of the *Anim* to which they are associated.

The *anim_state* member stores the current state of the item, *dir* stores the current direction (cardinal point) that the item is currently facing if the item is a character, or else it will be *null*. When an item speaks, its sentences will be displayed on screen as strings. The *speechColor* member stores the hexadecimal representation of the color that these strings must have.

Let us now consider the methods provided by the *EditorItem* class. The *setLayer* function takes as input an integer and has the following behavior: first, it retrieves the room containing the item by referencing *editorMapId-Room[parentRoomId]* and updates its *zOrderMap* member and, secondly, it sets the *layer* member to the proper value. It is important to remember that l is the function parameter while *layer* is a class member. Precisely, the function acts as following:

1. delete from the Array referenced by *zOrderMap[layer]* the element containing the item's identifier
2. if such Array is now empty, then delete the entire *zOrderMap[layer]* hashmap entry
3. multiply l by two
4. check for the existence of *zOrderMap[l]*: if such hashmap entry does not exist, then create it
5. add the item's identifier to the Array referenced by *zOrderMap[l]*
6. set *layer* to l.

The reason of step *3.* is that items can dynamically change layer at game execution time, thus it guarantees that any item can be set to a layer that is intermediate to the layers of two other items. More formally, let $item_i$ indicate an item with layer i, with $i > 0$: if a character walks **between** $item_i$ and $item_{i+2}$, a correct z-ordering of the scene must place the character at the layer *i+1*. This would not be possible without this simple multiplication, at least not as easily.

At editor level, users can visually drag and drop graphic representations of room items onto the canvas displaying the room's background. As already explained, a room item can have many graphic representations depending on its current state. For this reason, it was decided to define a *placeholder* for every room item which makes use of graphics. The global method *getItem-PlaceHolder(item)* takes as input a room item and returns its placeholder, defined as the first frame of the *Anim* instance associated to its default state. The last class method is *getItemCurrentFrame()*, which will retrieve the animation associated to the current item state and return the single frame it is displaying at the moment.

6.1.2 anims-manager.js

This module is delegated to the management of the game sprites. As explained in the previous chapter, the system does not handle single sprites but entire animations. A sprite is just an animation with a single frame. Animations are described by the *Anim* class, structured as shown in figure:

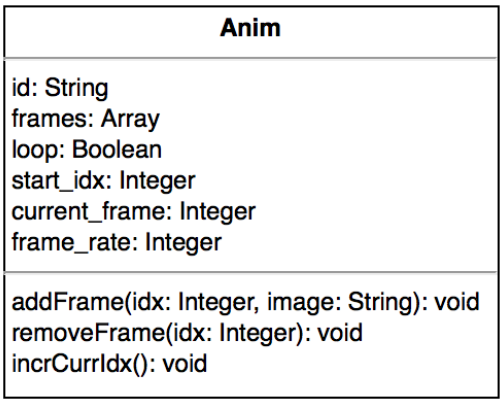

Figure 6.3

The global variables to contain and map animations are declared in this module and are respectively *editorAnimsList[]* and *editorMapIdAnim*{}. *Anim* instances store their frames into their *frames* property. Each element of the Array is a string containing the *base64* encoding of an image. The class members *start_idx* and *current_frame* respectively indicate the index of the *frames* Array that must be considered the first to display, and the one that must be displayed the next time the animation is played. The members *frame_rate* and *loop* indicate, respectively, the time interval in milliseconds after which displaying the next frame of the animation and whether the animation must be looped or not.

The class methods are really simple. *addFrame(idx, image)* stores the base64 encoding of the image contained in the *image* parameter at position *idx* of the *frames* Array. *removeFrame(idx)* will instead remove the string stored at position *idx*, while *incrCurrIdx()* will increment by *1 module n* the *curr_idx* member, where n is the length of the *frames* Array.

6.1.3 inventory-manager.js

This small module deals with a very particular type of game item, that is the inventory items. Inventory items can only be located inside of the playing character's inventory, which is displayed within a section of the game's GUI. The state of an inventory item is much less complex than "normal" items: indeed, they can either be in the inventory or not. Furthermore they do not have a spacial position, nor they can talk or walk. Nevertheless, they do have a graphic representation to be displayed within the GUI, so they are associated to an *Anim* instance. The figure shows the very basic structure of the class *InventoryItem*:

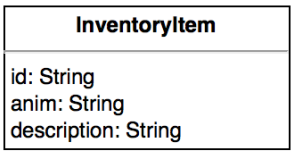

Figure 6.4

As usual, the module declares the global list and hashmap for the *Anim* instances, called respectively *editorAnimsList[]* and *editorMapIdAnim*{}. The *anim* class member contains the identifier of the *Anim* instance to which the inventory item is associated. The *description* member is a string containing a textual description of the inventory item, which will be displayed into the GUI when the user interacts with it.

6.1.4 variables-manager.js

In this tiny module, we can find the declaration of the global *editorGameVars*{} JavaScript Object, which maps the game variables names to their values. Moreover, it provides two global functions: *addGameVar(varName, varValue)* and *deleteGameVar(varName)*. The decision to create a dedicated module for so few code was taken in order to have a clear modular structure of the entire system.

6.2 Pathfinding

6.2.1 quadtree.js

As its name suggests, this module is delegated to the creation of quadtrees. The *Quadtree* class is rather rich, as can be seen below:

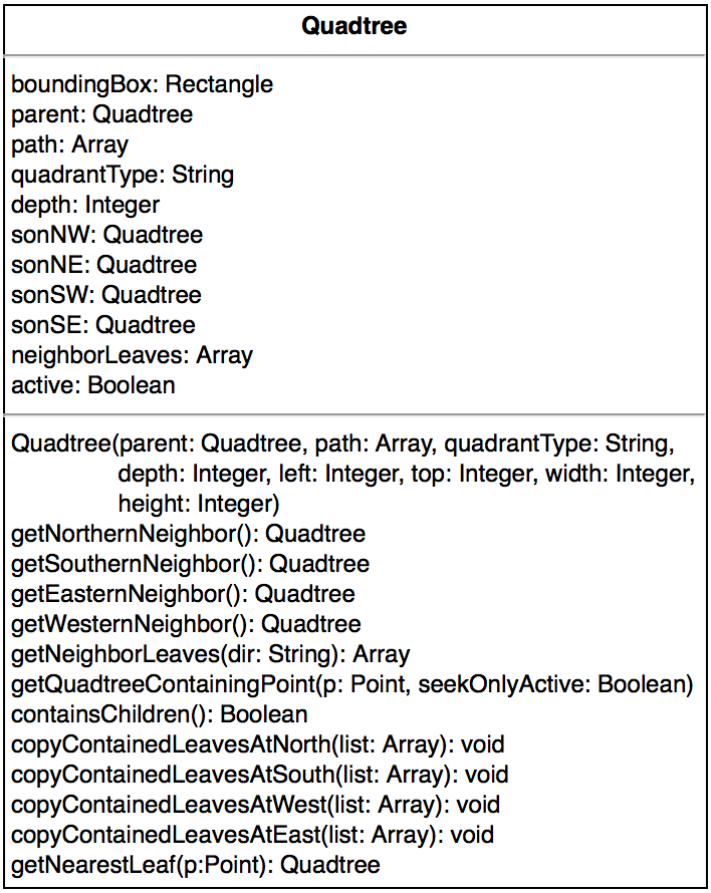

Figure 6.5

In this module, several global variables are declared:

- *walkableColor* is a string representing the color that defines areas where items can walk
- *walkableColorRGBA* is an Array of five elements, containing the numeric values of the red, green, blue and alpha components of *walkableColor*

- *pixels* is a JavaScript Object that will contain all the pixel data of the image that a quadtree will partition

From now on, pixels with a color matching *walkableColor* will be called *valid* pixels, while all of the others will be marked as *invalid*.

Let us start by examining the class constructor: it receives eight parameters, which will now be described. Parameter *parent* is a reference to the *Quadtree* instance which has called the constructor, thus is the parent of the quadtree that is being created. Obviously, the root quadtree will have this argument set as *null*. The second parameter, *path*, is an Array of strings that contains the entire path of the quadrants to traverse to reach the quadtree itself. The root quadtree will receive an empty array as parameter. The strings contained within the array are in the set { *"NW", "NE", "SW", "SE"*}. The *quadrantType* parameter is a string belonging to the same set, or is the string *"root"* for the root quadtree. The *depth* parameter is there for debugging purposes only, and describes the length of the quadtree being created, where the length of the root quadtree is 0. The last four parameters, *left*, *right*, *width* and *height* describe the bounding box of the spacial area that is wrapped by the quadtree.

Having described the constructor's parameters, we now know the purpose of most of the *Quadtree* class members, which have identical names. An **active** quadtree is a quadtree that:

a) does not have children, thus is a leaf

b) contains only valid pixels

If a quadtree is not active, its children will be stored inside the *childrenNW*, *childrenNE*, *childrenSW*, *childrenSE* members, each one dedicated to their relative quadrant. Finally, the *neighborLeaves* Array will contain, once the entire quadtree has been fully created, all the active quadtrees (if any) that are immediate neighbors of the *Quadtree* instance, at the four cardinal points.

The class methods have the following purpose: *getNorthernNeighbor()*, *getSouthernNeighbor()* , *getWesternNeighbor()* , *getEasternNeighbor()* will return the neighbor quadtree **at the same depth** (if present) and at the corresponding direction. Let qt_d be a quadtree of depth $d > 0$ and let us denote as $n_{dir}(qt_{\mathrm{d}})$ its neighbor at direction $dir \in \{ N, W, S, E \}$ of depth d.
The algorithm that retrieves $n_{dir}(qt_{\mathrm{d}})$ is the following:

1. $n_{dir}(qt_{\mathrm{d}})$ is a sibling if and only if:

- qt_d is a NW or NE quadrant and $dir = S$
- qt_d is a NE or SE quadrant and $dir = W$
- qt_d is a NW or SW quadrant and $dir = E$
- qt_d is a SW or SE quadrant and $dir = N$

2. otherwise, starting from qt_d, traverse the tree backwards, keeping track of the path that is being followed, until a quadtree T is found, such that:
   - for $dir = N$, T is a SW or SE child
   - for $dir = S$, T is a NW or NE child
   - for $dir = W$, T is a NE or SE child
   - for $dir = E$, T is a NW or SW child
3. if T exists, climb upon its parent and then traverse the path followed in step *2.* **in reverse order** and inverting, for each element of the path, the cardinal point described by dir

In the next figure, we can see an example of the northern neighbor (marked in green) lookup of a SW quadrant (marked in red):

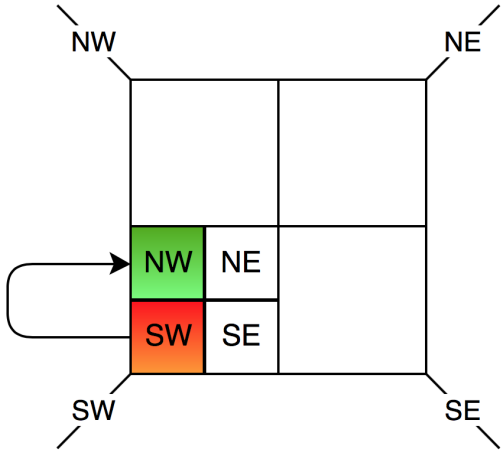

Figure 6.6: Simplest case of neighbor lookup

The same situation is represented as a clearer tree structure in figure 6.7.

As can be seen, this is the case in which the control in step *1.* is successful, thus the sought neighbor is a sibling. Let us now see a more complex case:

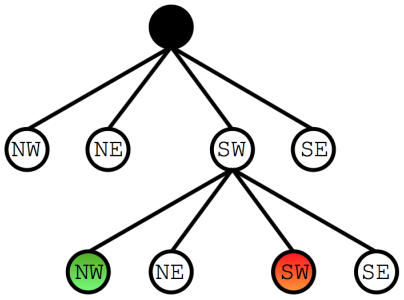

Figure 6.7: Tree representation of the previous quadtree

we are looking for the northern neighbor of a *NW* quadrant, marked in red in the next figure:

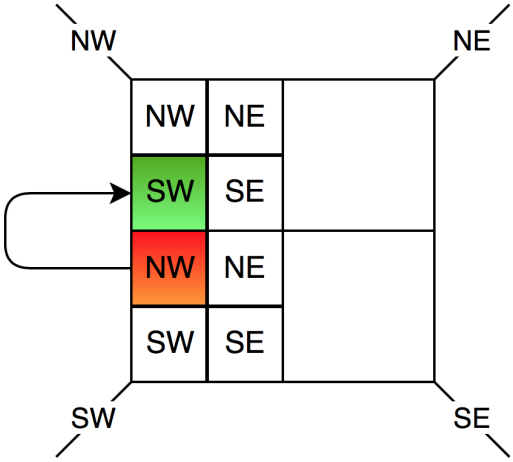

Figure 6.8: More complex case of neighbor lookup

The same situation is represented in a more clearly by the tree in figure 6.9. This time, the sought neighbor is not a sibling and steps *2.* and *3.* must be executed. The path to reach T is $<NW, SW>$. Once it has been found, we visit its parent, which in this case is the root. Starting from T's parent, we traverse down the tree again, following the path in reverse order and changing, for each element of the path, N with S and vice versa. Thus, the path that must be followed is $<NW, SW>$, as arrows 3 and 4 describe in figure 6.9.

Back to the description of the class methods, the *getNeighborLeaves(dir)* method will instead return all of the **active** quadtrees, no matter at which depth, that are adjacent to the quadtree at the direction described by the *dir*

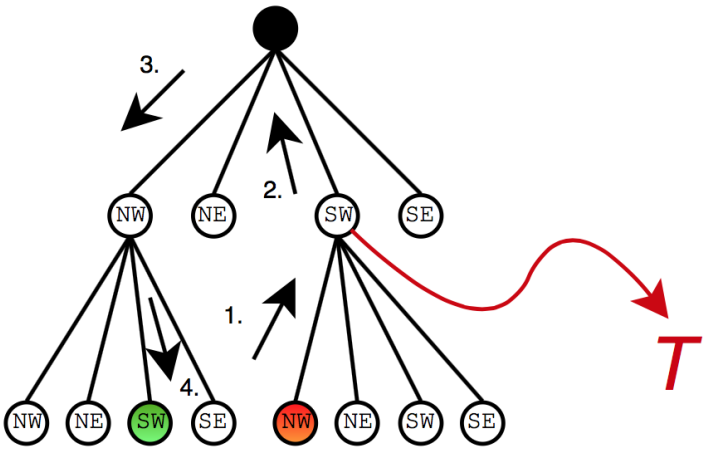

Figure 6.9: Application of the neighbor lookup algorithm

parameter. This search is a special case of the neighbor lookup just described. The method *getQuadtreeContainingPoint(p, seekOnlyActive)* will return a quadtree containing the point described by the *p* parameter. The return value will vary according to the value of the boolean parameter *seekOnlyActive*: if it is true, then the returned quadtree will either be an active quadtree containing the point or *null* if such quadtree does not exist. If it is false instead, it will return the smallest quadtree that contains *p*, possibly the root quadtree.

The *containsChildren()* method will tell whether the quadtree is a leaf or not: it is worth repeating that being a leaf is a necessary condition for a quadtree to be active, but is not sufficient, because its bounding box has to contain only valid pixels.

The four methods *copyActiveLeavesAtNorth(list)*, *copyActiveLeavesAtSouth(list)*, *copyActiveLeavesAtWest(list)*, *copyActiveLeavesAtEast(list)* take an Array as input and perform a depth-first visit of the quadtree at the direction specified by the method name, copying inside the Array all of the active quadtrees that are visited throughout the process.

Finally, the *getNearestActiveLeaf(p)* method takes as input the point *p* and returns the nearest active node to such point. The computed distance is the distance between *p* and the node's central point.

Once a user has traced a polygon that describes a room's walking area, a *Quadtree* instance is created to partition such area. The partitioning is performed by color, using an invisible canvas to hide the process to the user. After filling the canvas with white color, the polygon's surface is drawn onto

it with the color described by *walkableColor*. Then, the polygon's holes are re-filled with white color. The next figure shows the originally traced polygon and the content of the invisible canvas at the end of the process:

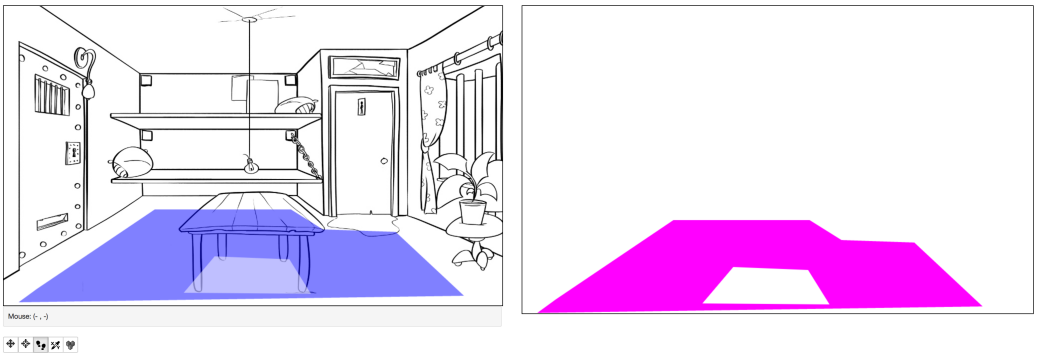

Figure 6.10: Original walking zone (left) and the invisible canvas (right)

The pixel data of the invisible canvas is stored inside the *pixels* global Object. The resulting image, which uses only two colors, is used as a bitmask for the generation of the quadtree. The generation starts with the creation of the root quadtree, which will have a bounding box of the same size of the entire background. For each quadtree *qt*, the construction process is the following:

1. examine the image pixels contained in the area of the bounding box: if both valid and invalid pixels are found, go to next step, else *qt* is a leaf, mark it as active node or inactive
2. if *qt* has a bounding box wide enough to be further subdivided, proceed to next step, else *qt* is an inactive leaf
3. split *qt* in four quadrants and assign each quadrant to a new quadtree, starting from *1.* for each one of them

The resulting quadtree is shown in figure 6.11.

6.2.2 pathfinding.js

This module provides classes and functions for the implementation of the $A*$ (A star) algorithm. $A*$ is a pathfinding algorithm based on best-first search and heuristics which guarantees to find an optimal solution. Let S be the path's starting node and G the goal. Moreover, let $f(n) = g(n) + h(n)$ be the cost function of the generic node n, where $g(n)$ indicates the cost of the

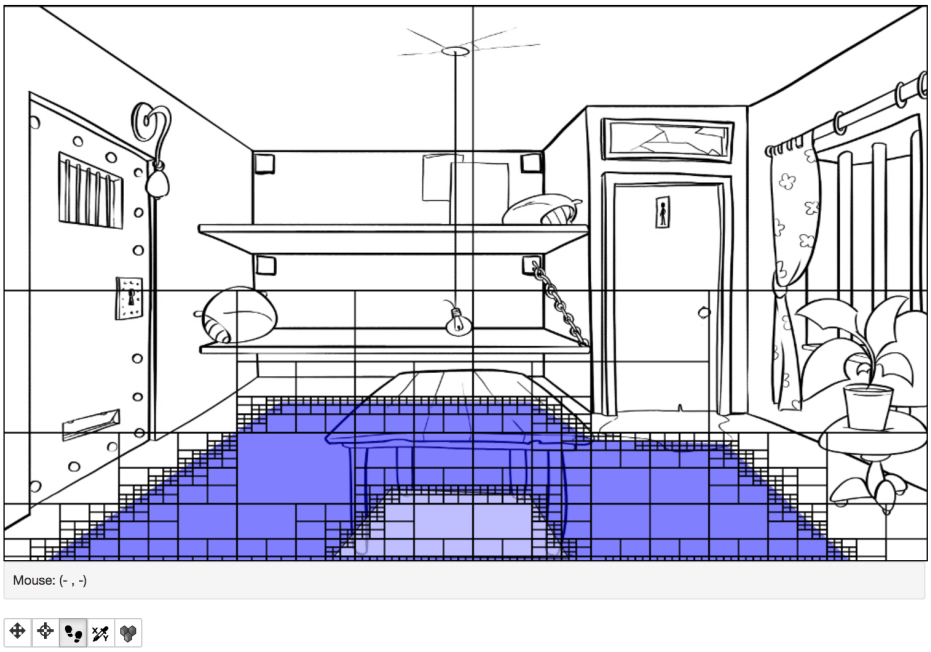

Figure 6.11: Spacial partitioning by quadtree

path to reach n and $h(n)$ indicates the **estimated** cost of the path from n to G. The following property holds:

$$h(S) = 0,\ h(G) = \infty,\ h(n) = Euclidean\_distance(n,\ G) \text{ for } n \neq S,\ n \neq G.$$

The algorithm uses two lists: *Open* is a priority list which will contain the nodes yet not visited, and *Closed* is a list that will store the visited nodes. The priority is given to the node with lowest cost function. Initially, $Open = \{\ S\ \}$ and $Closed = \{\ \emptyset\ \}$. At each step, the algorithm pops the highest priority node n from *Open*, puts it into *Closed* and adds to *Open* all the immediate neighbors of n that have not been visited yet, that is that are not already in the *Open* list. The algorithm will stop when the highest priority node of *Open* is G.

Figure 6.12 shows the classes that implement such behavior.

The *SearchGraphNode* class represents a generic search node of the path, with the relative g and h class members for the description of path cost and heuristic. Obviously, each instance of *SearchGraphNode* describes an active quadtree, i.e. has a one-to-one association with an active quadtree, whose reference is stored in the *quadtree* member. The *parentIdx* member is there to backtrack from G to S within the *Closed* list elements, once the goal has been reached. The *f()* method simply computes the function cost already

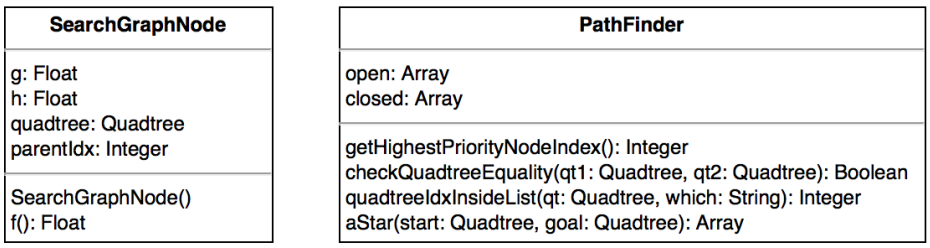

Figure 6.12: Classes for the implementation of $A*$ pathfinding

described and returns its value.

The actual implementation of the algorithm is provided in the *Pathfinder* class. We already know the purpose of its members, so we will examine the class methods. *getHighestPriorityIndex()* will return the index of the element contained in the *Open* list with the lowest cost function. *checkQuadtreeEquality(qt1, qt2)* will compare the bounding boxes of the two quadtrees passed as arguments, and will return a boolean. *quadtreeIdxInsideList(qt, which)* will check whether a search node associated to *qt* is already present in the *Open* or *Closed* list, depending on the value of the *which* parameter and will return its index within the list or *-1* if it is not present. The *aStar(start, goal)* method will return an Array of Points corresponding the the bounding box centers of the quadtrees that form the shortest path found between *start* and *goal*.

6.3 Scripting and compilation

6.3.1 From visual scripting to DSL scripts

The module *script-manager.js* defines the data structures needed to properly manage scripting in the DSL, whose grammar has already been described in Chapter 5. The class *ScriptElement* defines a **syntactic** representation of a valid *Statement* of the DSL: that is, *ScriptElement* instances may syntactically represent DSL's *Aggregators*, *If-controllers* and *SideEffects*. The class' structure can be seen in figure:

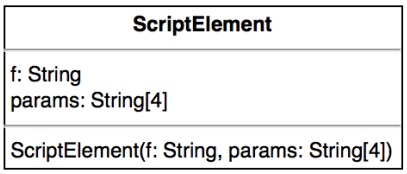

Figure 6.13: Class for the syntactic representation of a *Statement*

As seen earlier, *SideEffect* constructs have the form *Name(Params)*. Let *ValidFXNames* be the set of strings containing all of the possible *Name* parts of a *SideEffect* construct. The class constructor takes as input two parameters: the first one, *f*, is a String belonging to the set

$$\{ \text{“Aggregator”, “If”} \} \cup \textit{ValidFXNames}.$$

The second parameter is an Array of strings that describe, depending on the value of *f*, the *Aggregator* type (that is *Any*, *Sequence* or *All*), the *If-controller* guard, or the parameters of a *SideEffect*. The Array has a length ranging from 1 to 3, which is the maximum cardinality of the parameters passed to a *SideEffect* construct.

We know that the DSL is built in such a way that it is possible to nest any type of *Statement* into *Aggregators* and *If-controllers*. The process can be recursively repeated if the nested *Statement* is in turn of type *Aggregator* or *If-controller*. This means that a valid DSL script has a tree structure and that, of course, the *ScriptElement* class alone is not enough to express the structure of a script. The class called *ScriptTree* indeed implements this behavior:

As can be seen in figure, class instances will contain a *ScriptElement* instance and may contain an arbitrary amount of other *ScriptTree* instances.

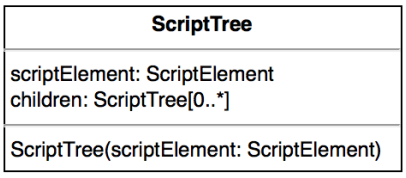

Figure 6.14: Class for the nesting of DSL *Statements*

This structure does not prevent a *SideEffect* from nesting other *Statements*, and this would be an undesired behavior. However, we will see that the *visual programming* part of the editor solves this problem before it can arise. Moreover, the *type* class member is a string that describes what is the type of the *ScriptElement* instance stored inside the *scriptElement* member. There are only two valid types: *game-controllers*, to which belong Aggregators and If-controllers, and *game-side-effects*, to which belong both interruptible and atomic SideEffects.

The next class represents the sequences of game events that will cause the execution of a script:

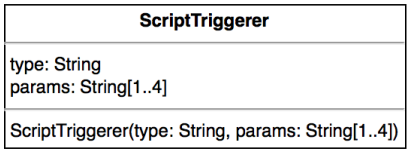

Figure 6.15: Class for representing script triggerers

The *type* member is a string belonging to the set { *"user-trigger", "event-trigger", "timer-trigger"* } and obviously refers to the type of triggerer among the three possible types described in Chapter 5. The *params* member is a string Array, with each element being the unique identifier of a game event. Its length ranges from 1 to 3, as for design choice a script can be triggered by a sequence of at most three events (for example: *Use*, *Key*, *Door*). Now that we have described all of its component, we may introduce the actual *Script* class:

Every DSL script is syntactically represented by instances of this class, whose members are in turn the syntactic representation of the script root and the script triggerers. Every instance of this class is stored into a global JavaScript Object declared in this module and called *editorScriptList*. This Object maps script names (identifiers) to scripts. Script names are chosen by the users when saving a script within the editor.

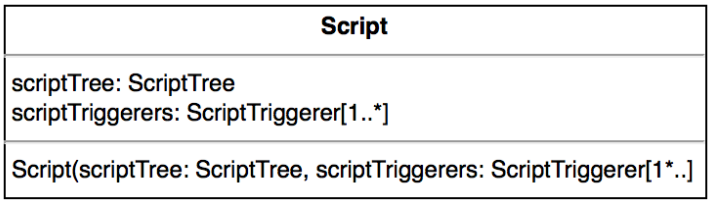

Figure 6.16: Class for the syntactic representation of a DSL script

Let us now see how the **visual** representation of a script, that is its *jsTree* graphic tree structure inside the editor, is then compiled into its **syntactic** counterpart with the *jsTree2ScriptTree* function:

```
1  var jsTree2ScriptTree = function(tree, parent)
2  {
3      var t = new ScriptTree(null);
4      t.type = parent == null?'game-controllers':tree.type;
5      t.id = tree.id;
6      t.parentId = parent;
7
8      var DOMnodes = tree.data.DOM;
9
10     if(DOMnodes)
11     {
12         var f = tree.text;
13         var params = new Array();
14         for(var i = 0; i < DOMnodes.length; i++)
15             params.push(DOMnodes[i].value);
16         t.scriptElement = new ScriptElement(f, params);
17     }
18     else t.scriptElement = new ScriptElement(
19                                     'Aggregator',
20                                      new Array('Sequence'));
21     var children = tree.children;
22     for(var i = 0; i < children.length; i++)
23         t.children[i] = jsTree2ScriptTree(
24                 editorScriptTree.get_node(tree.children[i]),
25                 t.id);
26     return t;
27 };
```

This recursive function takes as parameters a *jsTree* object and its *jsTree* parent's identifier. At its first call, the function will receive the root of the *jsTree* script and a *null* string. Each *jsTree* node will be compiled into a *ScriptTree* instance. By convention, the *jsTree* root will be compiled into a *ScriptTree* instance containing a *Sequence* aggregator *ScriptElement* object.

In order to allow the backwards process, i.e. the compilation from *ScriptTree* to *jsTree*, each id of a *jsTree* is dynamically saved onto its *ScriptTree* counterpart. This guarantees that the correspondence between graphic and syntactic representation of scripts is bijective. This occurs also for the parent id passed as second argument. The type of the *jsTree* node is either *game-controllers* or *game-side-effects* and will be stored as well into the relative *type* field of the *ScriptTree* instance being created.

Every *jsTree* node but the root contains a label to represent *Statement* names and at most three input fields to represent *Statement* params. This content is stored within a *data* object within the node and is, in the code above, temporarily stored into a *DOMNodes* variables. The content will be compiled into a *ScriptElement* instance, whose *f* field will contain the label and whose *params* field will contain the text data of the input fields. Finally, the function is recursively called for each *jsTree* child.

The following function compiles a *ScriptTree* instance into its graphic *jsTree* counterpart:

```
var scriptTree2jstree = function(tree, parentId)
{
    if(parentId == null)
    {
        parentId = 'j1_1';
        for(var i=0; i<tree.scriptTree.children.length;i++)
            scriptTree2jstree(tree.scriptTree.children[i],
                                parentId);
        var scriptTriggerers = tree.scriptTriggerers;
        for(var i = 0; i < scriptTriggerers.length; i++)
            view_CreateNewScriptRunnerTuple(
                scriptTriggerers[i].type,
                scriptTriggerers[i].params);
        return;
    }
    parentId = view_ScriptTreeAddNode(parentId,
        {text: tree.scriptElement.f,
        type:  tree.type,
        params : tree.scriptElement.params});
    for(var i = 0; i < tree.children.length; i++)
        scriptTree2jstree(tree.children[i], parentId);
};
```

The function *scriptTree2jstree* is somehow dual to *jsTree2ScriptTree*. The input params are respectively a *ScriptTree* instance and the identifier of its *jsTree* version. The first function call has the root *ScriptTree* and a *null* string. This will cause the recursive call of the function for every root child,

with the string "j1_1" as second parameter, which in *jsTree* is conventionally associated to the tree root. After every child has been set up, all of the *ScriptTriggerer* instances will as well be added to the DOM view by calling *view_CreateNewScriptRunnerTuple*, a function not belonging to this module that will later be explained in the proper section. Now, let us consider the second code branch, executed when the *ScriptTree* instance is not the root: the *jsTree* version of the tree node is added to the DOM by calling another external function and, then, the function is recursively called for each of the node's children. Again, the behavior of *view_ScriptTreeAddNode* will be explained in the *view.js* section.

6.3.2 From DSL scripts to evReact networks

Once a well-formed DSL script has been obtained, it must be compiled into a valid evReact expression to be properly interpreted during the game execution. The *test-manager.js* module is delegated to the creation and management of the game for test purposes and provides all the features that handle graphics, logic and inputs. A DSL script will be compiled at game creation time, with several functions that cooperate for this purpose. The function that actually returns compiled evReact networks is *ScriptCompiler*(script). This function is implemented as follows:

```
1  var scriptCompiler = function(script)
2  {
3      ...
4
5      var scriptBody = scriptTreeCompiler(script.scriptTree);
6      var scriptHeader = scriptTriggerersCompiler(
7                           script.scriptTriggerers);
8
9      var e = evreact.event.create('Finally');
10
11     return evreact.expr.iter(evreact.expr.cat(
12                  [evreact.expr.any([epsilon,
13                       evreact.expr.finallyDo(evreact.expr.cat
                            (
14                           [scriptHeader, scriptBody]),
15                           function() { e.trigger(); })]),
16                   evreact.expr.simple(e)]));
17 };
```

As can be seen, the first part of the code will call two other separate functions that compile, respectively, the script body and the script header. This is not a simple wrapper function, far from it. On the contrary, it makes sure that the

compiled evReact expression will be reactivated after termination and after completion. In other words, the game core will always be listening to the sequence of events that would trigger the execution of the script body: if the listened sequence matches, the script body will be executed until completion and termination, otherwise the evReact expression will simply terminate. In any case, the expression will be re-activated in order to repeat the process from start.

The first thing to notice is the custom evReact event that is created before actually manipulating the expressions returned by the two compilers (line 9). The e event will have, for debugging purposes, the string "Finally" as identifier. Now let us see the returned expression in detail: the expression is of the form

$$^{+}((\varepsilon \mid \boldsymbol{A} \mid\Rightarrow \boldsymbol{f}) \; ; \; \boldsymbol{B})$$

where

$$\begin{gathered} A = (scriptHeader \; ; \; scriptBody) \\ f = function() \; \{ \; e.trigger(); \; \} \\ B = e \end{gathered}$$

At the outermost level, it is an *iter* expression, which is the evReact construct that re-activates an expression after its completion. The first nested expression is a *cat* expression, which concatenates sorted sequences of (sub) expressions. The first subexpression of the sequence is an *any* of ε and A. As can be seen, when A completes or terminates, the f callback function will be called: such function simply triggers the event e. Back at the first nested expression (*cat*), the second subexpression B is a *simple* expression: as explained in section 4.4, a*simple* expression is atomic and is associated to a predicate and an event, and will complete and terminate when the event occurs and the predicate is true. In this case, its predicate is always true, since when a predicate is not explicitly specified, evReact will make this assumption.

Now, let us see the meaning of the whole expression. A is composed of subexpressions that will be explained later: for now, it is enough to know that it may terminate or complete, i.e. it may successfully listen to all the events it expects or stop listening without success. Whatever is the case, A will finally call f, thus it will trigger the event e. The ε expression is very important, as it makes sure that the expression is not deactivated if A terminates but does not complete. Indeed, if this is the case, ε completes, thus the first

subexpression of *cat* completes, and then B completes since the termination of A has triggered e. Instead, if A completes, the first subexpression of *cat* completes as well. When e is triggered, B completes, thus the whole expression at the outermost level completes and becomes inactive, then it will be immediately re-activated. The meaning of this expression is the following: "Wait for the events contained in *scriptHeader*. If such events occur, execute *scriptBody* and complete, else terminate. In any case, repeat the process.".

Now let us see the compilation of a script header. The compilation starts by calling the *scriptTriggerersCompiler* function, whose code is:

```
1 var scriptTriggerersCompiler = function(scriptTriggerers)
2 {
3     var exprs = [];
4     for(var i = 0; i < scriptTriggerers.length; i++)
5         exprs.push(scriptTriggererCompiler(
6                             scriptTriggerers[i]));
7     return evreact.expr.any(exprs);
8 };
```

As the code demonstrates, this function calls the actual compiler function for each script triggerer of the Array received as parameter, wraps the received expressions into an *any* expression and returns it. Far more interesting is the *scriptTriggererCompiler* function below:

```
1  var scriptTriggererCompiler = function(triggerer){
2      switch(triggerer.type){
3         case 'user-trigger':
4              var sequence = [];
5              sequence.push(evreact.expr.simple(
6                       testMapIdEvent[triggerer.params[0]]));
7              sequence.push(evreact.expr.simple(
8                       testMapIdEvent[triggerer.params[1]]));
9              if(triggerer.params[2])
10               sequence.push(evreact.expr.simple(
11                      testMapIdEvent[triggerer.params[3]]));
12
13             var restrictions = [];
14             for(var key in testMapIdEvent)
15                 restrictions.push(testMapIdEvent[key]);
16             return evreact.expr.restrict(
17                         evreact.expr.cat(sequence),
18                         restrictions);
19         case 'event-trigger':
20         ...
21         break;
22         case 'timer-trigger':
23         ...
```

```
24              break;
25          }
26      };
```

Firstly, the function will check for the type of triggerer that it is dealing with (see 5.1.3.1, page 46). In case of triggerers depending on user input, we know that a **sorted** sequence of two or three events must occur in order to fire the execution of the script body. For this reason, the function will create a *simple* expression for each event and put them into a *sequence* Array. This is surely done for the first two events of the triggerer and possibly for a third event (*triggerer.params[2]* is a boolean). It is worth noticing that a simple *cat* expression containing the sorted sequence is not enough to obtain the desired input handling. Indeed, we want the generic sequence of events e_i, e_{i+1} to be not only sorted but also **consecutive**: in other words, no other event rather than e_{i+1} must occur after e_i has occurred. The reason why this is the desired behavior is that the user is allowed to activate only one script at a time. Let us consider the following evReact expressions as compiled script headers:

i ***A ; C***

ii ***B ; C***

Moreover, let A, B and C be expressions that recognize a user input. The user might click on the GUI and trigger A, then click again on the GUI and trigger B, then finally click upon the room object that triggers C. In this case, the script body for i and for ii will **both** be executed, possibly resulting in an inconsistent game state. For this reason it is necessary to force an expression to be deactivated when an unexpected event of the sequence occurs. Such conduct is achievable by evReact's *restrict* expression. The following expressions actually work as desired:

i ***(A \ B) ; C***

ii ***(B \ A) ; C***

The function uses a *restrictions* Array that contains every game event and returns a *restrict* expression with *cat([sequence])* and the *restrictions* Arrays as parameters.

It is still to define how the script body is compiled by the function *scriptTreeCompiler*. Let us examinate its code in the next page.

```
var scriptTreeCompiler = function(scriptTree){
    if(!scriptTree)
        return epsilon;
    if (scriptTree.type == 'game-controllers') {
      if (scriptTree.scriptElement.f == 'Aggregator') {
        var evreactExprParamList = [];

        for (var i=0; i < scriptTree.children.length; i++)
           evreactExprParamList.push(
           scriptTreeCompiler(scriptTree.children[i]));

        switch (scriptTree.scriptElement.params[0]) {
         case 'Sequence':
         return evreact.expr.cat(evreactExprParamList);
         case 'All':
         return evreact.expr.all(evreactExprParamList);
         case 'Any':
         return evreact.expr.any(evreactExprParamList);  }}

    else if (scriptTree.scriptElement.f == 'If') {
       var expr = new Parser().parse(
              scriptTree.scriptElement.params[0]).expr;
       var exprEvaluationFun = evalExpr(expr, gameVars);
       var negatedExprEvaluationFun = function () {
           return !exprEvaluationFun(); };
       var e = evreact.event.create('ifEvent');
       var cond = evreact.expr.restrict(
          evreact.expr.cond(e, exprEvaluationFun), [e]);
       var negatedCond = evreact.expr.restrict(
       evreact.expr.cond(e, negatedExprEvaluationFun), [e]);
       var body = [cond];
       for (var i = 0; i < scriptTree.children.length; i++)
          body.push(
          scriptTreeCompiler(scriptTree.children[i]));
       return evreact.expr.cat([evreact.expr.react(epsilon,
       function () { e.trigger(); }), evreact.expr.any([
            negatedCond, evreact.expr.cat(body)])]);  }}

    else if (scriptTree.type == 'game-side-effects') {
      if (scriptTree.scriptElement.f in atomicEffectsMap)
        return atomicEffect(scriptTree.scriptElement.f,
          scriptTree.scriptElement.params);
      if (scriptTree.scriptElement.f in
         interruptibleEffectsMap)
        return interruptibleEffect(
          scriptTree.scriptElement.f,
          scriptTree.scriptElement.params);
      return null;    }};
```

The function is recursive and treats differently *Aggregators*, *If-conditions* and *SideEffects*. An *Aggregator* of type T is compiled into an evReact expression Exp_T such that:

- $T = Sequence \Rightarrow Exp_T = cat$
- $T = Any \Rightarrow Exp_T = any$
- $T = All \Rightarrow Exp_T = all$

The subexpressions that are nested within Exp_T are stored into a dedicated Array called *evReactExprParamList*, where each expression of the Array is obtained by recursively calling *scriptTreeCompiler* for each *scriptTree* child of the aggregator. Each recursive call will return a proper evReact expression that will then be stored into the Array.

Now, if the *scriptTree* instance to be compiled represents a *SideEffect*, the function checks whether it is *Atomic* or *Interruptible* by checking if the name of the side effect is contained into one global map or into the other one. These maps just associate a side effect's name to its interpreter function. In the first case, it will delegate the compilation to the function *atomicEffect(fId, params)*, while in the latter case it will call the *interruptibleEffect(fId, params)* function. Let us see how these two functions are implemented:

```
1 var atomicEffect = function(fId, params) {
2     var f = atomicEffects[fId];
3     var args = params.slice();
4     var reaction = function () { f.apply(null, args); };
5     return evreact.expr.react(epsilon, reaction);
6 }
```

atomicEffect retrieves the function with name *fId* and stores its reference into the *f* variable. Then it saves the *params* Array of function parameters into the *args* variable. This is done in order to exploit JavaScript's **closures**: indeed, the *reaction* variable is a function that will call *f(args)*, where *f* and *args* are *closed* variables, thus it will be able to access them even outside of their original scope.

The interesting part is the returned evReact expression: since evReact networks wait for events and react in some way as they occur, there is no native support for calling functions independently of event occurrences. The problem here is that we simply want the interpreters of the side effects to be called, because the events that triggered the script's body execution have already occurred at this point and we are not really expecting any more events.

This problem is bypassed by exploiting ε-transitions and the *react* expression. The function will return the following evReact expression: ε **|-> *f***. The semantics of this expression is very straightforward, as it waits for an ε event to occur and reacts to it by calling the f function. Since an ε event always occurs, this is equivalent to an imperative call of f.

The *interruptibleEffect* function behaves similarly and is implemented as follows:

```
1  var interruptibleEffect = function(fId, params) {
2      var completion = evreact.event.create(fId);
3      var f = interruptibleEffects[fId];
4      var args = params.slice();
5      args.push(function () { completion.trigger(); });
6      var reaction = function () { f.apply(null, args); };
7      return evreact.expr.cat(
8         [evreact.expr.react(epsilon, reaction),
9         evreact.expr.simple(completion)]);
10 };
```

This case is slightly different since, as the execution of the side effect interpreter is not atomic and might be interrupted before completion, it is mandatory to define an expression that takes completion into account. The function returns the evReact expression ε **|-> *f*** *; C*. C is a *simple* expression, thus it waits for a single event to occur in order to complete. This time the returned expression describes a sorted **sequence** of events that must occur in order to reach completion: the first event is ε, but the second is the *completion* event that is expected by the C expression. Semantically, this means that the entire *cat* expression will complete when both subexpressions complete. As explained earlier, ε always occurs, so C is used as a "barrier" that will be unlocked only after the *completion* event is fired. As can be seen, the *completion* event is added to the arguments of the f function: interruptible functions will trigger this event when every other computation is finished. In this way, the C expression will complete and the entire expression will complete as well.

The last possible case is that the *scriptTree* instance passed to *scriptTreeCompiler* is an *If-condition*. This is a very special case, since, in the editor, conditions are manually typed in be the users and, therefore, there is no guarantee that they are well-formed, as they are subject to syntax errors. For this reason, the entire condition must be parsed before being processed and compiled. There is an entire module, called *condition-parser.js*, delegated to the parsing of conditions and to their evaluation. The implementation details will be treated in 6.3.3. At the moment being, it is enough to know that:

- the *Parser* class provides a method *parse(condition)* that will either return the condition parse tree or *null* if the condition was rejected.
- the *evalExpr(expr, gameVars)* is a function that takes as parameters a condition (parse tree) and a scope and returns another function that, when called, evaluates the condition inside such scope.

The code from line 21 to line 25 shows how the parser is instantiated and utilized in order to have two evaluation functions such that the first will evaluate the condition inside the *gameVars* scope and the latter will evaluate the same **negated** condition inside the same scope. The whole expression returned by the compiler is of the following type:

$$\varepsilon \text{ } \textbf{|-> e.trigger() ; (((e ? pos \ e) ; Body) | (e? neg \ e))}$$

where *pos* = *exprEvaluationFun*, *neg* = *negatedExprEvaluationFun* and *Body* is a sequence of evReact expressions corresponding to compiled *scriptTree* instances that are children of the *If-construct scriptTree* itself. The semantics of this expression is equivalent to an *if-then-else* construct. Let us consider the *any* subexpression: we have two *cond* expressions that complete if and only if the event *e* occurs and the associated predicate is true. The predicate *neg* is the negated of the predicate *pos*. This guarantees that exactly one of the two expressions will complete. In case *pos* is true the token passes to the *Body* expression, otherwise the whole expression completes. It can be noticed that the two *cond* expressions are put in restriction with the *e* event: this is not strictly necessary as the whole expression will surely complete. However, it would not **terminate** without using such restrictions, thus it would not be deactivated and would uselessly go on listening for events.

6.3.3 Parsing and evaluating conditions

The module *condition-parser.js* provides all the features needed to parse and evaluate the guards of the *if* constructs of the DSL. The *Parser* class defines a recursive descent parser and has the following structure:

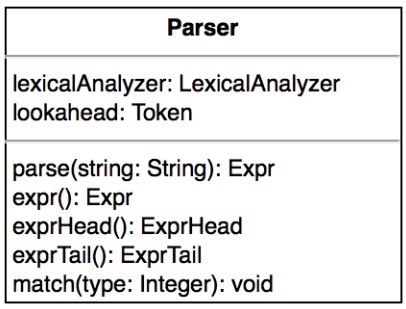

Figure 6.17: Class for recursive descent parsing of conditions

The parser is an LL(1) parser, which means that it will use only a token of lookahead to parse the received string. A parser performs the *syntax* analysis of strings, while the *lexical* analysis is done separately. The parser will indeed make use of a lexical analyzer, or **lexer**, which is delegated to read the characters that form the string and to split them into tokens. In turn, a **token** is a string of at least one character, called **lexeme**, which has an abstract type that represents a lexical unit. The classes that define lexers and tokens are shown in figure 6.18.

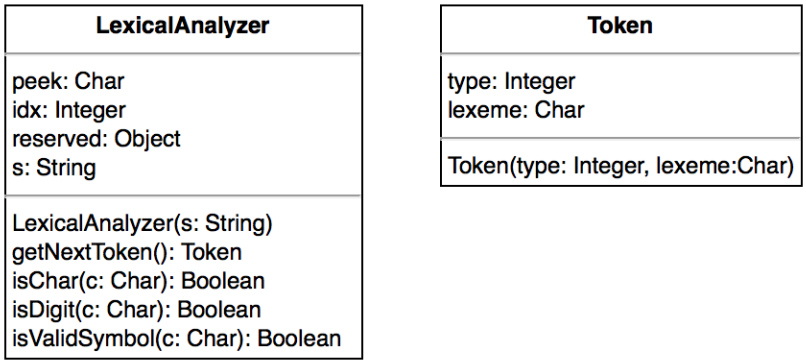

Figure 6.18: Lexer and Token classes

As the lexer scans the string, it creates *Token* instances and returns them to

the parser, which checks if the received token matches the expected type and proceeds or stops the parsing phase depending on the matching result.

The language grammar for expressing conditions has already been described in 5.1.1 at page 45 by the *BoolExpr* nonterminal. However, it has been factorized to obtain an equivalent LL(1) version, as follows:

⟨*Condition*⟩ ::= ⟨*Expr*⟩
| ε

⟨*Expr*⟩ ::= ⟨*ExprHead*⟩ ⟨*ExprTail*⟩

⟨*ExprHead*⟩ ::= ⟨*Id*⟩
| ⟨*Number*⟩
| not ⟨*Expr*⟩
| (⟨*Expr*⟩)

⟨*ExprTail*⟩ ::= ε
| and ⟨*Expr*⟩
| or ⟨*Expr*⟩
| > ⟨*Expr*⟩
| < ⟨*Expr*⟩
| = ⟨*Expr*⟩
| + ⟨*Expr*⟩
| − ⟨*Expr*⟩
| / ⟨*Expr*⟩
| ∗ ⟨*Expr*⟩

Naturally, every nonterminal is represented by a class, called with the same name for convenience. In the following we will use class names and nonterminal names interchangeably, as it will be clear by the context to which one we will be referring to. An *Expr* instance acts as a container of one *ExprHead* instance and one *ExprTail* instance. Instances of *ExprHead* contain a *type* member and a *body* member.

The **type** of an expression head is the type of the lexeme it contains, that is one among { *'Id', 'Number', 'not', '('* }, with corresponding expression types {*ID, NUM, OP_NOT, OPEN_BRACKET*}. The **body** of an expression may contain a variable name (*Id*), a number or another *Expr* instance.

The *parse* method receives a string as input and returns an instance of *Expr* if the string belongs to the grammar, or *null* otherwise.

The evaluation of conditions is performed by a dedicated function, whose code is as follows:

```
var evalExpr = function(expr, env)
{
    if (expr === null)
        return function () { return true; };

    var funHead = (function(head) {
        switch(head.type) {
            case Type.NUM: return parseInt(head.body);
            case Type.OPEN_BRACKET:
                return evalExpr(head.body, env);
            case Type.OP_NOT:
                var funBody = evalExpr(head.body, env);
                return function() { return !funBody(); };
            case Type.ID:
                return function() {
                    if (head.body in env)
                        return env[head.body];
                    else throw "Undefined variable.";  };
            default: throw "Syntax error, bad expression";
        };
    })(expr.head);

    return (function(tail) {
     var funTail = evalExpr(tail.body, env);
     switch(tail.type) {
       case Type.OP_AND: return function() {
            return funHead() && funTail(); };
       case Type.OP_OR: return function() {
          return funHead() || funTail(); };
       case Type.EPS: return funHead;
       case Type.OP_GT: return function() {
                  return funHead() > funTail(); };
       case Type.OP_LT: return function() {
         return funHead() < funTail(); };
       case Type.OP_EQ: return function() {
         return funHead() === funTail(); };
       case Type.OP_PLUS: return function() {
         return funHead() + funTail(); };
       case Type.OP_MINUS: return function() {
         return funHead() - funTail(); };
       case Type.OP_TIMES: return function() {
         return funHead() * funTail(); };
       case Type.OP_DIV: return function() {
         return funHead() / funTail(); };
       default: throw "Bad binary operator";
      }
    })(expr.tail);
};
```

The parameters *expr* and *env* are, respectively, an *Expr* instance and the environment where it must be evaluated. By convention, an empty condition is evaluated as *true*, as can be seen in lines 3-4. Let us examine the the assignment in line 6: the right-hand value of the assignment is the result of an anonymous recursive function that takes as input *expr.head* and returns a function that, when called, will return its evaluation. It can be seen in lines 8, 10, 13, and 17 that:

- if the expression's body is a number, the returned function returns its numeric integer value
- if the expression's body is an identifier, the returned function returns its value inside the scope
- if the expression's body is another expression, the function returns a recursive call to the anonymous evaluation function with its head as argument

The left-hand value of the assignment is a variable that will store the reference to the **result** of the aforementioned anonymous function applied to *expr.head*. In other words, *funHead* will contain a function returned by the call of the anonymous function with parameter *expr.head*.

It can be seen that if an *ExprTail* instance is not of type *EPS*, then its body will surely be another expression, thus an *Expr* instance. Otherwise, its body will be set to *null*. In line 23, another anonymous function is declared (and returned to the caller of the *evalExpr* function). This function takes as parameter an expression tail and returns a function which actually evaluates the full expression (condition). Before checking the type of the tail, a recursive call of the *evalExpr* function is performed, passing as parameter the tail body: the result of the recursive call is stored within the *funTail* variable (see line 24). After this step, the anonymous function is ready to return a proper function, whose computation depends on the tail type. Precisely, the returned function either applies the boolean operator to *funHead()* and *funTail()* or simply calls *funHead()* in case the tail type is *EPS*. Naturally, *funHead* and *funTail* are closed within the returned function's scope.

6.4 Run-time support

The module *sideEffects-manager.js* provides the run-time support of both atomic and interruptible game side effects. In the following, we will use the notation $entity_{entityId}$ to refer to the game entity with identifier *entityId*.

The following tables show the functions for the run-time support for game side effects, and their relative behavior.

function	**behavior**
walkToObj	Parameters: *walkingItemId*, *destItemId*, *callback*. Retrieves the walking spot *wSpot* of $item_{destItemId}$ and then calls *walkToPos(walkingItemId, wSpot.x, wSpot.y, callback)*
sayLine	Parameters: *itemId*, *sentence*, *callback*. 1) creates a DOM element that wraps the text defined by the *sentence* argument 2) if present, deletes the DOM element containing dialogue associated to $item_{itemId}$ 3) adds the element created in 1) to the DOM 4) after 2.5 seconds, deletes the element added in 3) and executes the callback function
walkToPos	Parameters: *itemId*, *xPos*, *yPos*, *callback*. 1) if $item_{itemId}$ has a graphic representation then proceeds to next step, else stops 2) retrieves the quadtree leaf containing the current position of $item_{itemId}$ and the nearest leaf to the point *(xPos, yPos)*. 3) uses the leaves in 2) as *start* and *goal* of the path, passes them to the pathfinder and stores the returned Array of points in *path* 4) pops a point p from *path* and sets the position of $item_{itemId}$ to p 5) if *path* is empty then executes the callback function, else waits 30 milliseconds and repeats step 4)

Table 6.1: Run-time support for interruptible side effects

function	behavior
fireEvent	Parameter: *eventId*. Retrieves $event_{eventId}$ from *testMapIdEvent* and then trigger it
setDirection	Parameters: *itemId*, *dir*. Sets the direction of $item_{itemId}$ as the *dir* parameter. Redraws the item on the scene.
show, *hide*	Parameter: *itemId*. Sets the *visibility* member of $item_{itemId}$ to true/false and retrieves the DOM element (if present) containing its graphic data, manipulating its attributes in order to show/hide it.
inventoryAdd, *inventoryRemove*	Parameter: *invItemId*. Adds/removes *invItemId* to the player character's inventory. Redraws the inventory inside the GUI.
varSet	Parameters: *varName*, *varValue*. Checks for the existence of the variable with name *varName* within the game scope, by accessing *gameVars[varName]*. If it exists, the function converts the string *varValue* (which is a **string** that represents a boolean, numeric or string value) into a proper value and assigns it to the variable.
varIncr	Parameters: *varName*, *incrAmount*. Checks for the existence of *gameVars[varName]* and verifies that it is numeric. If such conditions hold, it will try to parse *incrAmount* as a number: if the parsing is successful, then the variable will be incremented by the parsed value.
setPosition	Parameters: *itemId*, *xPos*, *yPos*. 1) sets the *position* member of $item_{itemId}$ to the coordinates *(xPos, yPos)* 2) if $item_{itemId}$ has a hotspot, translates each of its vertices in order to keep the same position relative to the item. 3) if $item_{itemId}$ is located inside the room being currently displayed, retrieves the DOM elements containing its graphical representation (if any) and its hotspot (if any) and manipulates their attributes so that they match the positions defined in 1) and 2)

Table 6.2: Run-time support for atomic side effects

6.5 Testing a game

Once the game resources have been properly set and the scripts have been defined, users can test the game before exporting it to a stand-alone HTML page. The *test* section of the editor wraps the game preview and allows to play it. An exported game will be completely equivalent to the game preview, so it can be said that the exportation process just changes the game wrapper and nothing else. The module *test-manager.js* contains all the core functions that allow to draw scenes, characters, items, GUI and so on. Since core functions are verbose and most of their code provides very common features in videogames, we will concentrate on the most interesting parts for our purposes, and will only briefly describe the least interesting functions. In the following, we will say that when a DOM node containing graphics is added to the DOM, it is "drawn" on screen. The term, although improper, is only used for convenience in order to avoid useless repetitions.

When the user pushes the *Play* button in the *test* section, the game resources will be initialized by calling the *initGame()* function. This function performs the following operations in order:

1) makes copies of all the data created during the editing phase, in order not to modify them while playing

2) calls the *startNets()* function, which is responsible for calling the scripts compiler described in 6.3.2 and for starting the obtained nets

3) creates three SVG layers:

   - *svg*, used to contain the current room's background image
   - *svg-hotspots-container*, used to contain the hotspots (polygons) of every item of the current room
   - *svg-items-container*, used to contain every other sprite of the room items

4) draws the current room and every room item (calls *drawScene()*) and the user interface (*calls drawGUI()).*

The function *drawScene()* acts as follows:

1) draws the character's sprite (calls *drawSprite(characterId)*) and sets its *z-index* properly (calls *setItemZIndex(characterId)*);

2) for each room item in order of layer, draws onto the scene its sprite (calls *drawSprite(itemId)* and its hotspot (calls *drawHotspot(itemId)*);

The function *drawHotspot* is very important because it activates the listening of basic mouse events by the polygons. In particular, the most important mouse event to be handled is the *mouseclick* event, which triggers an evReact event corresponding to the id of the item associated to the polygon. This allows the underlying evReact nets to progress and trigger the execution of scripts. This simple yet important task is carried out with the following line of code:

```
1 $(poly).click(function(e){testMapIdEvent[id].trigger();});
```

It should be noticed that polygons are drawn with no opacity, thus they are invisible to the users. It is mandatory to draw the polygons on screen, as their event listeners can be enabled only if they are present (although invisible) inside the web page.

The same applies for the *drawGUI()* function, which not only draws the user interface on screen, but also adds a listener to every GUI button in order to react to mouse events properly. A click on a table cell containing a game action (left half of the GUI) will trigger the relative action event, while a trigger on an inventory cell will trigger an object event with the same id of the inventory object it contains. Naturally, if the clicked inventory cell is empty, the click will be ignored.

The *drawSprite(itemId)* function instantly draws onto the scene the first frame of the current *Anim* instance of the item, and then defines an anonymous function which draws the successive frame and removes the old one. The anonymous function will be called at time intervals specified by the *frame_rate* property of the anim.

The remaining functions are setter functions:

- *setCurrentRoom(roomId)*, will update the scene by emptying all the graphic containers of the DOM and calling *drawScene()* again in order to have the new environment displayed to the user;
- *setItemLocation(itemId, roomId)* sets the *parentRoomId* member of the item to *roomId*;
- *setItemZIndex(itemId)* is normally called to set the current layer of characters or generic item rooms that can walk along the scene. The function will set the *z-index* CSS property of the DOM node containing the item sprite by checking the *walk-behind* lines (see page 52) associated to every other room item.

6.6 Editor canvas

In section 5.2.1 we described how the canvas and its toolbar can be utilized by users to define polygons and other properties of rooms and items. In this section we will see the module called *canvas-manager.js*, which provides the implementation of the different behaviors of the canvas. In the first few lines of code, some non-native members are added to the canvas and some global variables are declared and defined:

```
1 var canvas = $('canvas')[0];
2 canvas.state = 'idle';
3 canvas.mouseDown = {'clicked' : 'false', 'offset' : null};
4 canvas.selected = null;
5 var context = canvas.getContext('2d');
6 var MAX_WIDTH = 1000;
7 var MAX_HEIGHT = 600;
```

Code line 1 shows the result of a jQuery selector being assigned to the *canvas* variable. This is done in order to have a reference to the canvas data structure stored within a variable for all the duration of the session, thus reducing the number of jQuery queries to only one (for what concerns the canvas).

The *canvas.state* member reflects the toolbar button that is currently pressed, thus what kind of operation the user wishes to perform. Initially *idle*, the possible canvas states belong to the set { *idle, sprite, hotspot, pathfinding, xy-pick, walk-behind*}.

The *canvas.mouseDown* Object keeps track of whether the left mouse button is currently pressed and, in case the button has been pressed while hovering over a room item, the *(x, y)* offset from its top left position. The *canvas.selected* member stores the identifier of the room item the user has selected by pressing the left mouse button.

When drawing on a canvas, the programmer must refer to its drawing context, as the canvas itself is only a container for graphics that will be put over its *context* member. Code line 5 shows how the context reference is stored into a global *context* variable in order to have it available for all the duration of the session.

Finally, *MAX_WIDTH* and *MAX_HEIGHT* refer to the canvas and not to its drawing context: that is, the canvas will provide scrollers when its drawing context exceeds these sizes, but the context itself has no restrictions on the size of contained images.

The function *updateCanvas(roomId)* is called every time even a single pixel of the drawing context is modified. Firstly, the function will draw the background image of the room and then it will proceed by drawing every room item in order of layer, defined by the room's *zOrderMap* hashmap. Once all graphic data has been rendered onto the canvas, further context manipulation depends on the canvas state:

- if the state is *hotspot*, the currently selected item's hotspot polygon will be drawn
- if the state is *pathfinding*, the walkable area polygon surface and its holes will be drawn
- if the state is *walk-behind*, the "walk behind" line of the currently selected item will be drawn

In the following, we will examine the canvas' event listeners to user input. For convenience, we will use the notation *(x_{mouse}, y_{mouse})* to indicate the coordinates of the mouse pointer at the time a mouse event is fired.

6.6.1 *mouseclick* event listener

The *mouseclick* event will be handled differently, again depending on the canvas state. If the user is defining a polygon, i.e. the canvas state is either *hotspot* or *pathfinding*, then a new vertex will be added to the polygon, with the same coordinates of the clicked point.

In case of *pathfinding* state, a mouse click might add a new vertex to the surface or to the polygon holes, depending on whether the SHIFT key is pressed while clicking. Every time a vertex is added, the canvas is redrawn, with each vertex being visually represented by a small square handle. When the user clicks on the handle associated to the first vertex, the polygon is closed. When a polygon is closed, mouse clicks are ignored and the handles are naturally not drawn anymore, replaced by the polygon edges.

When the canvas state is *xy-pick*, a mouse click shows the canvas' clicked coordinates onto the clipboard below the canvas.
Finally, when the state is *walk-behind*, clicking on the canvas will result in drawing a line segment with endpoints *(0, y_{mouse})* and *(context.width, y_{mouse})*.

6.6.2 *mousedown* event listener

The *mousedown* event listener is useful in order to support drag and drop of items along the room. When this event is detected by the listener, the first thing it does is to set *canvas.mouseDown.clicked* to true. After this, it will check the canvas state and proceed only if it equals *sprite*. If this is the case, then the listener verifies if the mouse button was pressed over an item's bounding box and, if so:

- draws the item's bounding box onto the canvas in order to notify the user that he has selected an item
- sets *canvas.selected* to the corresponding item and *canvas.mouseDown.offset* to the x and y distance between the bounding box's top-left point and the mouse position

6.6.3 *mousemove* event listener

The *mousemove* event listener will react to events by updating the canvas clipboard to display the current mouse coordinates relative to the canvas. Moreover, if the canvas state is *sprite* or *walk-behind*, furhter operations are performed.

In the first case, the listener will check whether *canvas.mouseDown.clicked* is true and *canvas.mouseDown.offset* is not *null*. If such conditions hold, this means that the user is dragging an item image throughout the canvas, thus he is changing its position inside the room. Therefore, its *position* member will be updated and the canvas will be redrawn to graphically reflect its new position. Finally, for user convenience, the bounding box of the item will be drawn.

In the second case, *mousemove* events are handled by drawing a dashed segment line with endpoints *(0,* y_{mouse}*)* and *(MAX_WIDTH,* y_{mouse}*)*.

6.6.4 other event listeners

The *mouseup* event is handled, by simply resetting *canvas.mouseDown* to its initial state, already shown in code line 3. The *mouseout* event is handled the same way and, in addition, by updating the canvas clipboard mouse coordinates to *(- , -)*. Finally the *drop* event, which is handled when a user

drops an image onto the canvas from **outside** of it, updates the position of the dropped item and redraws the canvas.

6.7 Loading and saving projects

A project is a JSON representation of all the global variables that users manipulate through the editor. When a user saves a project, the function *saveProject()* makes copies of the important project variables, such as room lists, script lists, animation lists etc., and creates a *project* JavaScript Object that is defined as follows:

```
1 var project = {
2    rooms : JSON.stringify(backupRooms),
3    actions: JSON.stringify(editorActionsList),
4    characters: JSON.stringify(backupCharacters),
5    scripts: JSON.stringify(backupScriptList),
6    vars : JSON.stringify(backupGameVars),
7    anims: JSON.stringify(backupAnims),
8    inv_items : JSON.stringify(backupInvItemList)
9 };
```

As can be seen, every Object property is the JSON stringification of a variable copy. The *project* variable itself is then stringified in order to be ready for download. That is, a project is univocally represented by a string that the user may download. The download phase is managed by the *download(fileName, text)* function:

```
1 var download = function (filename, text)
2 {
3     var pom = document.createElement('a');
4     pom.setAttribute('href', 'data:application/json;charset=
          utf-8,'
5        + encodeURIComponent(text));
6     pom.setAttribute('download', filename);
7     pom.click();
8 };
```

The function creates an hyperlink which directly points to the JSON string of the project (line 4) and with *download* attribute equal to the name of the file which will be containing it (by default *project.json*, but it can be manually changed by the user before starting to download). Then, it fires the *click* event onto the link (line 7) in order to open the download window.

A project is loaded when the user clicks the relative button onto the navigation bar: the button is linked to an invisible *input* DOM node with type *file*, so that a click of the button triggers a click of the hidden node. The node is attached to an event listener for events of type *change* that, when an event is detected, calls the *loadProject(event)* function. This is necessary because the only way to upload files onto a web page is by *<input type="file">* DOM elements. The upload window only displays files with *.json* extension.

```
1  var loadProject = function(event)
2  {
3      var fileReader = new FileReader();
4      fileReader.onload = function(event)
5      {
6          var project = JSON.parse(event.target.result);
7          // reset global vars
8          ...
9          /* for each project property, parse its JSON
10            content and assign the resul to the relative
11            global variable */
12         ...
13     }
14     fileReader.readAsText(event.target.files[0]);
15 };
```

As a file is chosen for upload, the *change* event is triggered and detected by the listener, which passes it as argument to the *loadProject* function. This function unwraps the file content from the *event* argument (line 14): the resulting string is then parsed by the native JSON parser and stored into a *project* variable (line 6). Consequently, every resource of the loaded project is available as a JSON string in the relative property of the *project* variable. Before proceeding with the parsing of every property, the global variables are reset in order to clean them from manipulation previous to the project loading. At this point, each property of the *project* variable is then parsed and assigned to the relative global variable.

6.8 Exporting games

The exportation of games is performed by the functions defined in the *exporter.js* module. As aforementioned, the approach of the system is *WYSIWYG* (What You See Is What You Get), i.e. an exported game will behave under every point of view as the game that can be tested from within the editor in the dedicated section. This means that, when a game is ready to be tested, it is also ready to be exported. The function *exportGame()* creates a very large string called *program*, that represents the whole source code of the web page to be generated. This string is obtained by concatenating three strings, each one containing a different part of the page:

- *header*, containing the page's fixed HTML
- *globalVars*, containing the definition of all the game variables for graphic resources, scripts etc.
- *coreFunctions*, containing the string representation of all the functions that are needed to run the game properly (core functions)

Given a generic function *f*, a string representing its source code can be obtained by calling *f.toString()*. For each core function *cf*, the exporter appends *cf.toString()* to the *coreFunctions* variable. Once the *program* string is obtained, the function lets the user download an HTML file containing such string, by calling the *download* function described in 6.7. By default, the file name will be *game.html*.

6.9 HTML View

The module *view.js* is delegated to the management of the DOM manipulation: features for event handling, addition and removal of DOM nodes are provided by this module. In this section, we will examine the creation of non-trivial sets of nodes, the handling of user input and, more importantly, the management of the visual programming part of the editor.

6.9.1 Event listeners and handlers

Event listeners for static input fields are initialized at page loading time, as can be seen below:

```
<body onload="init()">
```

As soon as every DOM element of the page has been loaded, the *init()* function is called. The function contains a long list of event listener attachments, such as:

```
$('#project-uploader').change(loadProject);
$('#project-downloader').click(saveProject);
$('#export-game').click(exportGame);
$('.add.room').click(function()
{
   var newRoomName = 'Room_' + editorRoomsCount++;
   createNewEditorRoom(newRoomName);
   view_CreateNewRoomPanel(newRoomName);
});
$('#start-test').click(initTest());
...
```

Without losing generality, we will now consider the part of the module that handles the scripts from the perspective of the DOM view, i.e. the part that invokes the compilation of visual scripts into their syntactic representation. When a user decides to save its visual scripts, a listener attached to the *Save script* button handles the mouse click event as follows:

```
$('#save-script').click(function()
{
    var scriptId = $('#script-name')[0].value;
    var isValid = validateScriptId(scriptId);
    if(!isValid)
        return;
    saveScript(scriptId);
});
```

Line 3 shows a typical jQuery selection, which retrieves the input element with id *script-name*, unwraps it from jQuery's *$* object and obtains the input string typed in by the user as the name for the script being saved. The id is then validated and the script can finally be saved by calling the relative function, whose code is as follows:

```
1  var saveScript = function(scriptId)
2  {
3      var scriptTree = jsTree2ScriptTree(
4         editorScriptTree.get_node('j1_1'), null);
5
6     var triggerers = new Array();
7     $('.tuple').each(function()
8     {
9       var tupleData = $(this).find('input');
10      var params = new Array();
11      for(var i = 0; i < tupleData.length; i++)
12         params.push(tupleData[i].value);
13      triggerers.push(new ScriptTriggerer(
14        $(this).attr('class').split(' ')[2], params));
15    });
16
17    editorScriptList[scriptId] = new Script(
18       scriptTree, triggerers);
19 };
```

In section 6.3.1 we have seen that the function *jsTree2ScriptTree* compiles a script's graphic representation within the DOM as its syntactic counterpart, but we have not explained where the triggerers are taken from in order to create a *Script* instance, which contains both a *ScriptTree* instance and an Array of *ScriptTriggerer* instances. The code shows how each triggerer's graphic representation, which is of a list of input fields, is contained within a DOM node with class *.tuple*. For each of these nodes (line 7), the value of every input field is extracted and saved into an Array, then an instance of *ScriptTriggerer* is created and added to a *triggerers* Array. When every triggerer has been processed, a *Script* instance is added to the global scripts hashmap (line 17).

6.9.2 Complex DOM nodes

The most complex functions of the module handle the creation of subpanels (see section 5.2, page 50). Indeed, event listeners and handlers for non-static DOM elements such as subpanels must be created and removed on the fly,

as soon as a user clicks the *Add* or *Delete* buttons. For completeness, we will consider the creation of a *Room* subpanel:

```
1  var view_CreateNewRoomPanel = function(roomId){
2    var roomPanel = $(document.createElement('div'));
3    var roomPanelHeading = $(document.createElement('div'));
4    var roomPanelTitle = $(document.createElement('h4'));
5    var roomPanelToggler = $(document.createElement('a'));
6    var roomPanelCollapse = $(document.createElement('div'));
7    var roomPanelBody = $(document.createElement('div'));
8    var bgSelector = $(document.createElement('input'));
9
10   roomPanelBody.append(bgSelector);
11   bgSelector.attr(
12         {
13             'type' : 'file',
14             'accept' : 'image/ *',
15             'id' : roomId + '-bg-selector'
16         });
17     ...
18     roomPanelCollapse.attr(
19     {
20         'id' : roomId + '-panel',
21         'class' : 'panel-collapse collapse'
22     });
23     roomPanelCollapse.append(roomPanelBody);
24
25     roomPanelToggler.attr(
26     {
27         'data-toggle' : 'collapse',
28         'data-parent' : '#room-accordion',
29         'href' : '#' + roomId + '-panel',
30         'class' : 'glyphicon glyphicon-collapse-down
31             pull-right',
32         'style' : 'color: inherit; text-decoration: none'
33      });
34       ...
35     $('#room-accordion').append(roomPanel);
36     ...
37     view_CreateEditorItemPanelGroup(roomPanelBody);
38 }
```

The function takes as parameter the id of a new room, which is guaranteed to be automatically valid. Lines 3 to 9 show the declaration and definition of the DOM nodes that form a subpanel to be added to the static *Rooms* panel. As shown in line 11, some of these nodes are put inside the subpanel's body, in this case the input field that allows users to load a background image for the room being created. Lines 26 to 33 show the exploitation

of Bootstrap's *accordions*: given a hyperlink such as *roomPanelToggler*, it is enough to manipulate its attributes in order to obtain an animated collapsible element: precisely, it is necessary to specify the hyperlink's destination (in this case, the *roomPanelCollapse* element), and its data parent, which is the main panel that acts as a container for all the defined rooms and is a static DOM element, so its life cycle will endure for the entire session. Once the subpanel has been created, it is appended to the main panel, as shown in line 36. The creation of a room subpanel will trigger the creation of an item panel to be added to its body, as the function call at line 38 shows.

6.9.3 Visual scripting

We have already seen that the *Scripts* section of the editor presents several labels that visually represent game side effects and aggregators from the DSL. From the perspective of the DOM, there are two HTML unordered lists, one dedicated to aggregators and one to game side effects. Each label is actually an HTML list element (*<li>*) contained inside one of the two lists, as shown below:

```
1  <ul class="game-controllers">
2     <li><span class="label label-info" value="Aggregator">
          Aggregator</span>
3     </li>
4     <li><span class="label label-warning" value="If">If
          Condition</span>
5     </li>
6     <li><span class="label label-warning" value="Repeat">Loop<
          /span>
7     </li>
8  </ul>
9  <ul class="game-side-effects">
10    <li><span class="label label-info" value="fireEvent">
          fireEvent</span>
11    </li>
12    <li><span class="label label-primary" value="setPosition">
          setPosition</span>
13    </li>
14    ...
15    <li><span class="label label-primary" value="inventoryAdd"
          >inventoryAdd</span>
16    </li>
17    <li><span class="label label-primary" value="
          inventoryRemove">inventoryRemove</span>
18    </li>
19 </ul>
```

The *init()* function makes the labels draggable by attaching to each one a proper handler for *dragstart* events:

```
1 $('.game-side-effects li, .game-controllers li').on(
2    'dragstart', function(event) {
3         event.originalEvent.dataTransfer.setData(
4            'data', JSON.stringify({
5               'type' : $(this).parent()[0].className,
6               'text' : $(this).find('span').attr('value')
7            })
8        );
9    }).attr({'draggable' : 'true'}).;
```

The jQuery selector retrieves each <*li*> child of the DOM nodes with class *.game-side-effects* and *.game-controllers* (notice that the two unordered lists are the only nodes of the DOM with these classes), and attaches to each one a listener for *dragstart* events. The anonymous function that acts as event handler, sets the *data* property of the input event as a JSON stringification of an Object with two properties:

- *type*, corresponding to the class of the <*ul*> node that is parent of the <*li*> element;
- the *value* attribute of the <*span*> children of the <*li*> element.

This manipulation of the event is essential in order to have trace of the originally dragged label once it is be dropped.

Each jsTree node has class *.jstree-node*. The support for dropping labels onto them is provided by the following code:

```
1 $('.jstree-node').on('drop', function (event) {
2    var data = JSON.parse(
3         event.originalEvent.dataTransfer.getData('data'));
4    var thisType = editorScriptTree.get_node($(this)).type;
5    if(thisType == 'game-side-effects')
6        return;
7    if (data.text.length > 0) {
8         view_ScriptTreeAddNode($(this), data);
9         event.stopImmediatePropagation(); }});
```

As soon as a label is dropped over a jsTree node, the Object stringification previously assigned to the event's *data* property is parsed (line 2) and stored into a variable. Then, the jsTree node checks its own type (line 5), in order to verify whether it can contain nested children or not. If this is the case, the function *view_ScriptTreeAddNode(parent, data)* is called to add the new jsTree node to the DOM. The importance of this function is critical, as it

visually represents every construct of the DSL, thus is the core of the **visual programming** part of the system. Its implementation is the following:

```
1  var view_ScriptTreeAddNode = function(parent, data)
2  {
3      var newNode = editorScriptTree.get_node(
4        editorScriptTree.create_node(
5          parent, {'text' : data.text}, 'last');
6      editorScriptTree.open_node(parent);
7      newNode.type = data.type;
8      var icon;
9      switch(newNode.type) {
10         case 'game-controllers':
11           icon = ...
12           newNode.original.max_children = Infinity;
13        break;
14         case 'game-side-effects':
15           icon = ...
16           newNode.original.max_children = 0;
17         break;
18     }
19     editorScriptTree.set_icon(newNode, icon);
20
21
22      newNode.data = { DOM : []};
23      switch(newNode.text.toLowerCase()) {
24         ...
25         case 'sayline':
26             newNode.data.DOM[0] = // datalist
27             newNode.data.DOM[1] = // input field
28             break;
29         ...
30     }
31     // add to the DOM all the elements of newNode.data.DOM
32     if(data.params) // if loading a script
33         view_PopulateTreeNodeDOM(
34            newNode.data.DOM, data.params);
35 }
```

The function takes as parameters the jsTree parent node of the node that is going to be created, and a JavaScript Object containing information about the type of node (aggregator or side effect), its label (**name** of the aggregator or side effect) and possibly its parameters. In fact, as seen in 6.3.1, page 85, code line 16, this function is also called when converting a *scriptTree* instance to a jsTree node: this happens when a previously saved script is loaded and shown into the editor. In this case, the *data* argument will also contain a *params* property, which stores the parameters typed in by the user inside the

relative input fields. The function performs several operations:

1) creates the jsTree node, specifying its parent node and its label (line 3);
2) visually expands its parent node, in order to show the new node on the page (line 6);
3) assigns *data.type* to the new node (line 7);
4) depending on the node type, assigns a different icon to the the node (lines 8-19);
5) adds an Array to the new node, intended to store its additional input fields (line 22);
6) depending on the label of the node, assigns different input fields to the Array (lines 23-30);
7) adds the input fields just created to the DOM;
8) if the parameters for this node have already been specified (i.e. the function has been called to ***load*** an already existing script), fill the input fields with each parameter (lines 32-34).

The last part of the *view.js* module we are going to see is the function that manages the visual representation of script headers. Under the part of the page that contains the (visual representation of the) script body, there is a small section with a dropdown menu that allows the user to specify the type of script triggerer:

```
1 <div class="row">
2    Run this script on
3    <select id="script-runner">
4       <option value="user-trigger">User interaction</option>
5       <option value="event-trigger">Event occurrence</option>
6       <option value="timer-trigger">Timer</option>
7    </select>
8   <span id="add-execution-triggerer" class="glyphicon
      glyphicon-plus"></span>
9 </div>
```

The <*span*> element is the *plus* icon that must be clicked in order to create a new script triggerer. This is done by the usual attachment of an event listener to the icon:

```
1 $('#add-execution-triggerer').click(function() {
2    view_CreateNewScriptRunnerTuple(
3       $('#script-runner')[0].value), null });
```

When the user clicks the *plus* icon, the current option of the dropdown menu is passed as first argument to the function *view_CreateNewScriptRunnerTuple(type, data)*. Said function can also be called when the user is loading a script that has already been created, and in that case the second argument will contain the data that must fill the triggerer's input fields. In our case, the second argument is *null* since a trigger is being created from scratch and not loaded, thus its input fields still have to be filled in. Let us examine how the function is implemented:

```
1  var view_CreateNewScriptRunnerTuple = function(type, data) {
2      var row = $(document.createElement('div')).addClass(
3         'row tuple ' + type)
4      var eraser = $(document.createElement('span')).attr(
5         'class', 'glyphicon glyphicon-remove')
6      eraser.click(function() { row.remove();});
7      row.append(eraser);
8
9      switch(type)  {
10         case 'user-trigger':
11         /* append to the row:
12                1) a list of game actions
13                2) a list of game items
14                3) a checkbox
15                4) a list of game items  */
16        row.append(...);
17        row.append(...);
18        row.append(...);
19        row.append(...);
20         break;
21         case 'event-trigger':
22         // append to the row an input field of type string
23         row.append(...);
24         break;
25         case 'timer-trigger':
26             /* append to the row:
27                1) an input type of type number
28                2) a checkbox  */
29        row.append(...);
30        row.append(...);
31     }
32
33     $('#script-container').append(row);
34     if(data)
35         view_PopulateTuple(type, row, data);
36 };
```

Firtsly, the function creates a row, that is a * * container with class *.row .tuple* (see Bootstrap's layout system, section 4.3, page 37), as shown

in line 2. A responsive icon for the deletion of the entire row is created and appended to the row itself (lines 4-7). After this preliminary phase, the function determines which input fields it must add to the row by checking the *type* parameter (lines 9-33). Such input fields visually represent the event chain that will trigger a script's body execution. In case of *user-trigger*, the event chain consists of an action event, an item event (target object 1) and possibly another item event (target object 2). The corresponding input fields are:

- a <*select*> element, with <*option*> elements corresponding to the identifiers of the actions that the player can perform
- a <*select*> element, with <*option;*> elements corresponding to the identifiers of all the items that the player can interact with (including inventory items);
- a checkbox which specifies if the event chain contains a third event or not;
- same as *1)*, but disabled if the checkbox is not checked.

In case of *event-trigger*, the event chain consists of a single event, that is the event that, once fired, will provoke the execution of the script body. The corresponding input field is a simple <*input*> element of type string, intended to contain the identifier of the event.

Finally, if the triggerer is a *timer-triggerer*, there is no explicit chain of events, as the event triggering the script is simply the passing of a time lapse. However, it is necessary to specify what is the time lapse and if, once it has passed, the timer should start over in order to trigger the execution periodically. Indeed, the input fields are an <*input type="number"*> element that specifies the time lapse in milliseconds, and a checkbox that, if checked, signals that the timer must periodically restart.

Once every input field has been added to the row, it is appended to the part of the scripts section that is intended to contain the script body and header (line 35).
Finally, if the *data* parameter is not *null* (thus the function is loading a previously created triggerer), another utility function will be called, which will fill in the input fields of the row coherently with the triggerer parameters specified by *data* (lines 36-37).

7 Conclusions and future works

We have been able to define and implement an authoring system for graphic adventures. The system is highly portable since it has been built as a web application compatible with the most popular cross-platform browsers. The created scripting language is able to describe all the main typologies of interactions and the typical situations that are generally found in graphic adventures, especially those made with the SCUMM engine, which was our comparison term.

The visual programming environment greatly simplifies the work of scripters with little programming experience. In addition, the underlying evReact networks make their work much easier, managing the whole control flow and the application state in such a way that the scripters only have to care about firing (and waiting for) game events .

The obtained system demonstrates how the evReact approach for modeling reactive programming suits very well the management of complex control flows that are typically present in story-driven games. Furthermore, saving the state of a given game by serializing the underlying evReact networks has relieved us of explicitly creating data structures to represent a consistent game state, which would have required much work and time.

The serialization of evReact networks potentially allows to simulate a full replay of the game, from its start to the actual state, by firing all the events that have been detected and stored as tokens within the evReact inner structures. Moreover we have been able to achieve almost full portability of the generated games, with limited conflicts with some browsers that can anyway be easily worked out.

The system performs all of the tasks that we considered interesting, but is far for complete. Many features have been cut out during the early stages of the work, as they were not essential and would have took away a lot of time from the implementation of much more interesting and fundamental features.

The first enhancement for a possible future version is to extend the support to other popular browsers, as Internet Explorer: at the moment being, the system is fully compatible with Chrome and Firefox, and partially compatible with Safari.
As explained earlier, the if-guards must be typed in manually by the scripters because of time constraints. It would actually be natural to extend the visual programming approach of the editor in order to also include such conditions. On the other hand, another desirable feature is to also include a non-visual programming environment, in such a way that expert programmers can be able to write their own code without being forced to rely on the graphical programming environment.

More importantly, the system handles graphical resources but does not support other forms of media, such as audio and video data. Including sounds and other media would surely result in increasing the quality of interactivity and storytelling, so their support by the system is surely enticing. Of course, this would require a careful extension of the scripting language and of the system classes, in order to have *ad hoc* supporting features.

Another nice addition would be the support of real-time scaling of moving items and characters, in order to have a more coherent sense of perspective of the environments, thus an overall higher suspension of disbelief from the players.

Finally, it would be very interesting to distribute the game control flow between server side and client side. Indeed, this would have interesting implications, e.g. it would make possible to reduce the control flow of the client side to a minimum by keeping locally the evReact networks that listen to input events and storing on the server side all the other networks (that is, the networks that manage the plot unfolding and the game progress). Furthermore, such distributed control flow would prevent users from "cheating", as forcing the game progress locally would not be possible thanks to the server side control.

We have tested the distribution of the control flow between a server and a client running an exported game. Precisely, the server stores its own evReact networks that keep track of the game's plot unfolding, while the client stores the evReact networks that handle inputs from the user. In this setting, users

can play the game from a device, close their session and let the server serialize its networks. Once they restart the game (from the same device or even from another one), the server deserializes the networks that track their progress and restores the game state that was reached during the previous session. In other words, the game progress is preserved by serializing and deserializing the server-side networks. This makes it very easy to keep the game state coherent, without particular efforts even when it is modified from multiple devices.

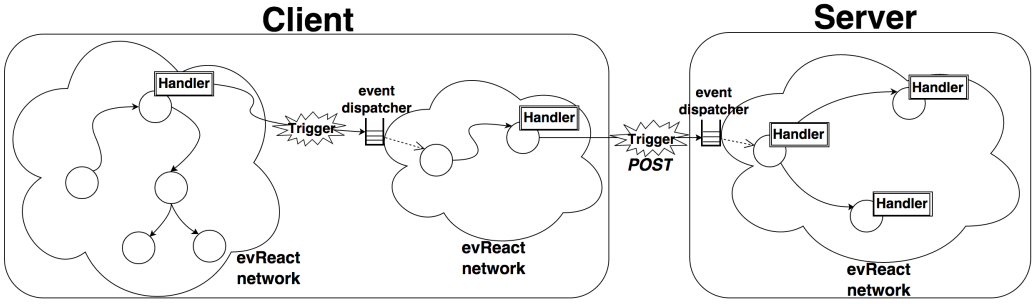

Figure 7.1: Control flow distributed between client and server

Figure 7.1 shows an example of distributed control flow, with different evReact networks stored in the client and in the server. The evReact networks are represented as clouds: the client contains two networks, while the server only has one network. The plain circles represent evReact expressions and the circles with handlers represent reactions. The **Handler** icons represent the functions that are executed as reactions to events. As these functions can contain arbitrary code, a handler can basically perform any kind of operation as soon as it receives an expected event. This aspect is very important, as it makes it possible to exploit communication between different networks, even between client and server. In the figure, the **Trigger** icons describe the situation in which a handler has actually triggered an evReact event. The two triggers in figure have a slightly different behavior: the one on the left activates a network local to the client, while the second one communicates with the server. Indeed, the second handler performs a **POST** or **GET** request, which triggers an event on the server. When the server receives such request, it triggers a local event and its evReact networks will possibly react to it. This distribution of the control flow is very powerful, as the state of a client-server application is implicit within the evReact networks. Normally, keeping and ensuring consistency of the state of a client-server web application is one of the aspects that requires most work and efforts. The possibility of having an implicit state managed by evReact is one of the aspects that make this framework particularly appealing.

Bibliography

[1] Engineer Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666.

[2] *Puzzle Dependency Charts*. Grumpy Gamer. URL: http://grumpygamer.com/puzzle_dependency_charts.

[3] Kurt Kalata et al. *Hardcoregaming101.net Presents: The Guide to Classic Graphic Adventures*. S.l.: CreateSpace Independent Publishing Platform, May 17, 2011. 772 pp. ISBN: 9781460955796.

[4] *Zap Dramatic - Life Experience Through Simulations*. Nov. 24, 2014. URL: http://www.zapdramatic.com/about.htm.

[5] *The Reactive Manifesto*. URL: http://www.reactivemanifesto.org.

[6] *The SCUMM Diary: Stories behind one of the greatest game engines ever made*. URL: http://www.gamasutra.com/view/feature/196009/the_scumm_diary_stories_behind_.php.

[7] Lloyd Rosen. *SCUMM - The Infernal Machine*. URL: http://www.mts.net/~kbagnall/commodore/scumm/scumm%20overview.txt.

[8] *Adventure Game Studio | AGS*. AGS is an Adventure Engine to create graphical point-and-click adventure games - Adventure Game Studio. URL: http://www.adventuregamestudio.co.uk/site/ags/.

[9] *Wintermute Engine ■ features*. URL: http://dead-code.org/home/index.php/features/.

[10] *Visionaire Studio*. Nov. 24, 2014. URL: http://wiki.visionaire-tracker.net/wiki/Main_Page.

[11] *SVG 1.1 (Second Edition) – 16 August 2011*. W3C. URL: http://www.w3.org/TR/SVG11/intro.html.

[12] *jQuery*. URL: http://jquery.com/.
[13] Jake Spurlock. *Bootstrap*. 1 edition. Beijing: O'Reilly Media, May 22, 2013. 128 pp. ISBN: 9781449343910.
[14] René David and Hassane Alla. "Bases of Petri Nets". In: *Discrete, Continuous, and Hybrid Petri Nets*. Springer Berlin Heidelberg, Jan. 1, 2010, pp. 1–20. ISBN: 978-3-642-10668-2, 978-3-642-10669-9. URL: http://link.springer.com/chapter/10.1007/978-3-642-10669-9_1.

8 Acknowledgments

Per prima cosa ringrazio di cuore la mia famiglia, per il supporto totale in ogni momento della mia vita e in particolar modo durante questo lungo percorso universitario. Grazie per essere la mia forza!

Ringrazio il Prof. Cisternino per avermi dato la possibilità di sviluppare questo lavoro di tesi, per i consigli, la pazienza e le sonore risate riecheggianti per tutto il dipartimento che mi hanno accompagnato in questi mesi.

Grazie a tutti i nerd del laboratorio 304, dai quali ho imparato un casino di cose interessanti in mesi di sedute intensive di nerdaggine.

Ringrazio tutti i miei amici, non posso elencarli tutti perché vorrei tenere il numero di pagine sotto il migliaio!

Un grazie speciale a Giovanna.

Beh, credo sia tutto, pare che alla fine anche io ce l'abbia fatta!
So long, and thanks for all the fish!