A Game Engine Designed to Simplify 2D Video Game Development

Miguel Chover [1] Carlos Marín [2] [5] Cristina Rebollo [3] Inmaculada Remolar [4]

Institute of New Imaging Technologies - Universitat Jaume I. Castellón. Spain.

[1] chover@uji.es
[2] cmarin@uji.es
[3] rebollo@uji.es
[4] remolar@uji.es

[5] Corresponding Author. E-mail: cmarin@uji.es. Phone Number: +34 616 29 76 65

Abstract

In recent years, the increasing popularity of casual games for mobile and web has promoted the development of new editors to make video games easier to create. The development of these interactive applications is on its way to becoming democratized, so that anyone who is interested, without any advanced knowledge of programming, can create them for devices such as mobile phones or consoles. Nevertheless, most game development environments rely on the traditional way of programming and need advanced technical skills, even despite today's improvements. This paper presents a new 2D game engine that reduces the complexity of video game development processes. The game specification has been simplified, decreasing the complexity of the engine architecture and introducing a very easy-to-use editing environment for game creation. The engine presented here allows the behaviour of the game objects to be defined using a very small set of conditions and actions, without the need to use complex data structures. Some experiments have been designed in order to validate its ease of use and its capacity in the creation of a wide variety of games. To test it, users with little experience in programming have developed arcade games using the presented environment as a proof of its easiness with respect to other comparable software. Results obtained endorse the concept and the hypothesis of its easiness of use and demonstrate the engine potential.

Keywords

Game Engine, 2D Video Games, Game Editor, Game Logic.

1. Introduction

Video game creation is a very complex process where the participation of a multidisciplinary team is required, as well as the use of tools to assist the production of content. Game developers not only have to create games interesting enough to captivate players, but they also have to face the complex technical features required for today's computer games: graphical resources, interaction mechanisms and behaviour definition [23]. In order to simplify the problem, game development has evolved quickly since the mid-1990s, mainly because of the emergence of game engines. These tools aim to create reusable software to provide an easier way to generate games and reduce their production times. Although in their early stages the primary concern of these engines was the rendering system, other fields such as artificial intelligence, animation, physics, sound or networking were added over time.

Over the years, and with the purpose of making content creation accessible to more people, some game engines have been incorporating visual programming systems [3, 27] for the definition of game behaviour. In this way, the developers are now able to visually compose the scenes, set the interaction, determine the behaviours of the game characters, and even to debug errors. However, and despite the fact that since its conception the game engines have simplified the creation of video games, these tools are still too complicated to use. The set of functions and elements they have is still very large and is still difficult to handle for a non-programmer user.

As a starting point, the democratization of game development seems to be easier to achieve in 2D. As an example, one of the most popular environments for creating interactive 2D content through visual programming is called *Scratch* [31]. In this case, traditional programming is omitted by encapsulating code functions in block-shaped nodes that the user has to organize to create their algorithms. However, in this way, it does not eliminate the inherent complexity of traditional programming methodologies and although the environment is more attractive, complexity is similar to traditional programming. For these reasons, the development of tools that facilitate and make accessible the creation of video games for everyone remains an open problem and has a great interest.

This paper presents a Simplified Game Engine (*SGE*) designed to ease the game development process by providing tools oriented for non-programmers. This 2D game engine has been designed by simplifying each user-dependent process as much as possible to provide a most satisfying level of abstraction in terms of technology. The contributions of the work are aimed at the simplification of the video game specification, emphasizing on the simplification of the game logic definition, and can be summarized in the following ones:

- Simplification of the data model used to define games and a consequent specification of a simplified game engine architecture and editor.
- Elimination of hierarchical structures of game objects, common in most game engines to exploit the agent-based programming paradigm [57].
- Creation of a visual programming system using binary decision trees, used in several fields beyond computer science [26, 58].
- The engine does not include complex data structures such as vectors or matrices [15], common in other visual systems such as *Scratch*. Elimination of repetition statements, since the iteration is provided by the game loop itself.

Finally, in order to validate the capabilities and ease of use of the engine, two experiments have been carried out with children without great programming knowledge. The main objective is to compare *SGE* with *Scratch*, one of the most widespread visual programming environments. *Scratch* is used to program scripts in some game engines such as *Stencyl* [30, 47].

The rest of the paper is organized as follows. In section 2, the leading work in the area is introduced with the state of the art on game engines and visual tools to learn to programme. Next, in section 3, the technical conception of this work is developed, focusing on the game engine architecture and game specification along with the game editor and its behaviour specification system. Thereafter, a complete use case example is presented comparing the programming using *SGE* and *Scratch* in section 4. The experience gained from an experiment carried out with children along with its results are then presented and discussed in section 5. Finally, section 6 analyzes the main contributions of the proposal and its limitations together with an outline of the ongoing work.

2. State of the art

The video games industry, like any other, tries to minimize production costs in order to maximize profits [17, 54]. During the mid-90s, some companies, such as *IdSoftware*, added modularization to their main engines with the intention of reusing the software. They developed the First Person Shooter (FPS) game *Doom*, from where any further addition or change was quite easy to implement by modifying levels, characters, weapons or even creating new games. This led to the game engine concept and provided tools to develop new games in an easier way. In the late 90s, some games, such as *Id Software*'s *Quake III* and *Epic Games' Unreal*, were built on a modular and reusable conception. In this way, game engines improved the customization possibilities by adding coding features, for instance, scripting functionalities as *Quake C*. From this point on, game development companies became aware of the commercial interest of game engine licences and started looking towards an additional source of income.

As time passes, the improvements in graphics hardware, visualization technology and data structure are closing the gap between game engines for varying purposes. Today, it is possible to create 2D or 3D games with the same game engine. Even though specialization is still capital [23], creating a Massive Multiplayer Online Game (MMOG) is quite different from an FPS. The required features can be very different for each game genre. For instance, 2D animated sprites are pretty simpler to set up than realistic 3D visualization algorithms [48] or, in the same way, collision computations and physics simulations are far more complex in 3D [35]. An example of these game engines is *Unity* [33, 34, 50], a mighty platform to develop 2D or 3D games where deep knowledge about game engine concepts, features and advanced experience in the programming language *C#* is required in order to develop anything. Despite this, it is easy to perceive a trend towards simplification: a 3D engine of the highest level such as *Unreal Engine* [51] includes a visual programming system called *Blueprints Visual Scripting* [52] based on message passing, where programming is done by connecting game objects' components and functions. However, this engine has a high-level commercial purpose, so its use is still quite complex for non-technical people.

In response to the needs of these potential users, some companies have developed 2D game engines intended for creators without advanced programming knowledge. Its systems have visual editors to configure scenes, characters and even gameplay mechanics without writing a line of code. Most of them include visual programming methods, an approach that can bring simplicity to this process through one of its visual scripting methodologies: block-like, flowcharts-inspired, dataflow or message-passing programming, finite-states machines, event-based rules or behaviour trees [3, 27].

Table 1 presents a brief summary of the analysis of some of the current game engines that allow creating 2D games. The table details the platform where they are executed, the scripting system they use, their visual programming methodology and the number of functions or behaviour descriptors each one has to configure those gameplay mechanics. The elements of the table are arranged in ascending hierarchy based on the number of functions or behaviour descriptors and their ease of use.

The table begins with *Flowlab* [16], which has fifty-three different elements to configure behaviours and ends with *Unity*, where the action is conducted by handmade *C#* scripts based on the complete language and some specific programming libraries and APIs. Many of them, like *Game Maker* [20] or *RPG Maker* [44], have systems based on lighter scripting tools, which keeps a dependency on code. Additionally, others such as *Construct 2* [46] or *Stencyl* [30, 47] rely on block-like interfaces, this latter case working with *Scratch* [31], a visual programming concept developed by the Massachusetts Institute of Technology (MIT). Usually, these block-like systems are based on events, in fact, *GDevelop* [11] attaches its behaviour definition system directly on events into a cross-platform engine with a visual programming interface. Finally, *GameSalad* [14, 43] presents a graphic interface which allows to visually configure functioning by arranging and connecting components.

Table 1 - Classification for state-of-the-art 2D game engines.

Game Engine	Platform	Scripting Method	Behaviour Specification	Elements*
Flowlab	Web	Visual Scripting	Message passing	53
Game Salad	Desktop	Visual Scripting	Components	63
GDevelop	Desktop	Visual Scripting	Event System	106
Game Maker	Desktop	Game Maker Language	Message passing	133
Construct 2	Desktop	JavaScript	Blocks	204
Stencyl	Desktop	Scratch	Blocks	C.L.
RPG Maker	Desktop	Ruby, C++, Java, JavaScript	Scripting	C.L.
Unreal	Desktop	C++, Visual Scripting	Message passing	C.L.
Unity	Desktop	C#	Scripting	C.L.

** Number of predefined set of functions or behaviour descriptors | C.L. Complete Language*

Even though these efforts are very significant steps forward towards the game development complexity reduction, the use of this kind of software still requires high technical profiles and specific training, thereby excluding most of the potential users of these technologies [38, 45]. One of the main reasons is related to the transition process between traditional programming and visual programming: it has generally been done by transforming programming functions into components, which avoids dependence on the code but maintains the huge variety of functions of traditional APIs. This way of proceeding inherits a systematic problem from the development of game engines: there is no generic game engines language, there is deep darkness about the essential requirements that a game needs to be executed and the limits between games, genres and engines of games are blurred [1, 2]. With this in mind, it is necessary to find a simpler way to create games, where inexperienced users could develop their own games by making use of graphical environments that do not require programming skills [24].

In the current literature, there are works that point out how complex can be for a beginner the approach to a problem through computational techniques [42, 7, 37] and the assistance that visual programming can provide [8, 5]. At the educational level, different methodologies have been studied to introduce programming concepts, both with traditional coding [28] and with visual programming [40]. It seems evident that visual programming can be an essential tool on the way towards the democratization of game development. In fact, some authors have carried out experiences with students associating visual programming and computer games. For instance, some authors [39] present a study conducted on programming students to test the learning of basic programming concepts by creating games with *Scratch* [31], and others [10] display a study to evaluate an object-oriented programming learning methodology through videogames programming.

At a more specific level, some works have proposed models that combine visual programming methodologies with the elements that a game engine requires to define the behaviours of a game. In this line, Furtado et al. [19] propose a description of game engines based on a more abstract and expressive set of layers. Furthermore, Zarraonandia et al. [59, 60] presented a conceptual model to organize the game features in a modular way, where the description and the definition required to create a combination of sub-games are based on a set of configurable elements and a basic vocabulary for each feature. Additionally, some software engineering methods have appeared as a possible plan to address this issue, proposing a systematization of the game development process [1, 41, 18]. All this analysis demonstrates that the creation of video games can be simplified and the development of new visual tools can make the creation of such content accessible to a large number of users.

3. The simplified game engine

The main hypothesis of this work is to demonstrate that the complexity of a game engine can be reduced in relation to the software architecture, the specification of the games and the editor itself while maximizing its potential in terms of creating different types of arcade games. In this section, the architecture of the proposed system is introduced below. This architecture has been designed to be able to create a wide variety of games, including physical games. Subsequently, the specification of the games is described. Each game can be composed of different scenes, with actors that can have different behaviour rules or scripts. All actors have the same properties which simplify both the specification of the games and the implementation of the engine. Finally, the game editor designed using the aesthetics of *Google Material Design* [22] is briefly described.

3.1. The game engine architecture

Essentially, most of the existing game engines have quite similar architectures and subsystems. This is because these modules are necessary for designing practically any game. A generalist system needs some modules that manage rendering, sound, logic evaluation, input system and physics. It is for this reason that this proposal has included a set of five modules: physical computing controllers, event management, logical evaluations, sound reproduction and scene representation. A representation of this architecture can be seen in Figure 1.

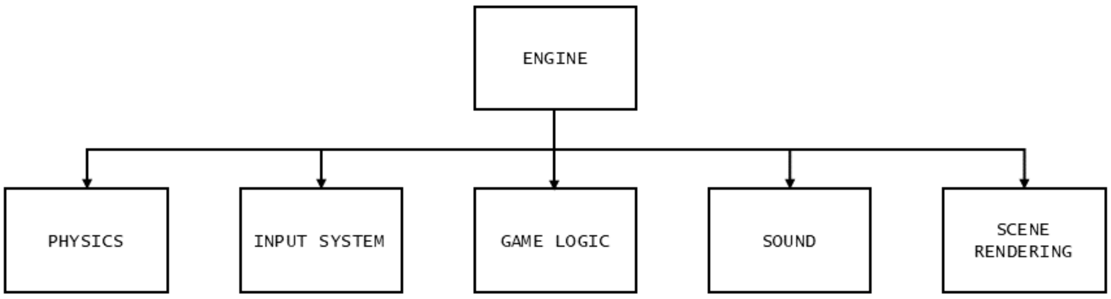


Figure 1. - Classic game engine architecture.

The most important module that has been developed is the game logic module. For the rest of the modules, a high-level layer has been created that allows different libraries to be incorporated. For example, in relation to the physics engine, both *Box2D* [6] and *Matter* [32] have been tested. In relation to the render engine, a 2D render has been developed from scratch and *Pixi* [53] has also been tested. In relation to the game logic module, a system has been implemented that allows building the behaviour rules using decision trees and a small set of conditions and actions (see section 3.2).

The *Game Loop* processes every action or condition in these five submodules on each iteration. It starts from the physics, goes through the event handling, continues with the game logic evaluation and, finally, it represents the state of the *Game* by the sound and rendering modules. The implementation of a *Game Loop* in a classic game engine may become a very challenging and very complex task. According to Deloura [15], this implementation usually absorbs the users in customized scripts for each game object to be executed in different stages of the *Game*, or even in a different number of times per frame, making it necessary for the user to have a vast knowledge of the game engine structure and operation. This process has been considerably simplified in *SGE*, the logic of the game is always computed after the physics and input evaluation in each game cycle. This simplification can be done since most 2D games do not have great performance needs.

3.2. The game specification

Basically, an *SGE Game* can be represented as a set of *Scenes* with a list of *Actors*, where the *Actors* are the elements of the game and the *Scene* is the stage used by the *Actors*. The concept behind this setup is to make *Actors* work as independent agents [4, 57] carrying out their roles and interacting both with each other and with the *Scene* configuration. Accordingly, a game specification has been developed, on the basis of the *Actor*, and through its *Properties* and *Rules*, any of these elements can be configured or modified. A diagram of this configuration is presented in Figure 2. The system has no hierarchy of actors like most 3D game engines, which simplifies the game engine architecture. This absence of hierarchy has not been a problem to implement any mechanics in the arcade games that have been created so far.

The *Game* is the central object of the system's data structure. Its properties are mainly related to the resolution of the screen and the orientation of the screen along with the sound linked to the entire game. The user also has the option of storing new custom variables to meet some game requirements, for example, to store the total score that players reach. In turn, the *Game* is composed of *Scenes*. These are responsible for storing camera properties, such as position, angle, zoom or gravity. The *Actors*, considered the main component of the game, are assigned in the *Scenes*. This is the most complex component since it is the cornerstone on which all interaction rests. They also have their own properties, related to their position, render, text, sound and physical properties. In the same way as the other components of the game, it can store custom variables to help the development of the game. In addition, they have a list of rules assigned to them that perform the gameplay by combining actions included in some programmable nodes. As they are considered the main structure of the *SGE*, they are explained in detail in the following.

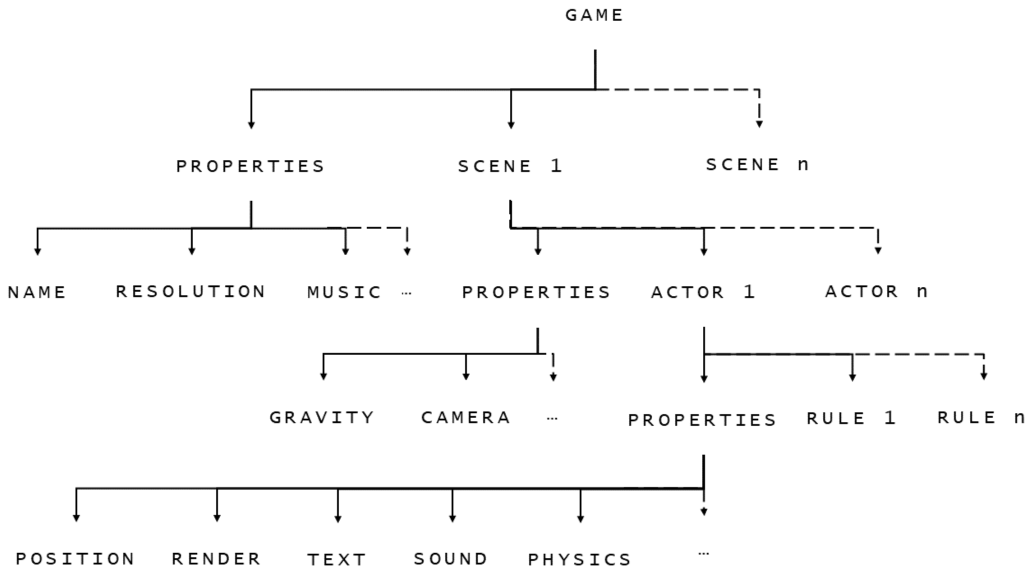


Figure 2. - Game data structure.

3.2.1. *Actor* properties

The *Actors* are the only element in the *Scene* and they are organized by an ordered list to determine their order of viewing. All the elements arranged in a *Scene*, such as characters, backgrounds, decoration, sound emitters, markers or HUD's, are *Actors*. They play a key role in the definition of the game. In a conventional game engine, the *Actor* concept would be expressed as game objects with different types of roles: lights, camera, sound, geometry, etc. However, this specification only requires the use of *Actors*, removing the reliance on game hierarchies, a central topic in traditional games implementation but, in fact, not essential to create most of the classic arcade games.

The *Actors* have a predetermined set of *Properties* by default, some of them related to the visual appearance: colour, line width, etc. Depending on their function in the game, they can be visible in the *Scene*, for example, when an *Actor* is assigned an image, it will be visible just as the gameplay designed using the

Actor's *Properties* and its *Rules* require. Besides images, they can represent other features in the game, such as *sounds*, *text*, *numbers* or *booleans*.

Finally, the established set of *Actor*'s properties can be classified and arranged in categories, in order to provide a better understanding for the final user. These properties are as follows:

- **General**. These properties deal with the position, scale and rotation of the *Actor*. They also include information about the collision shape profiling and if the object is enabled or disabled.
- **Graphic**. Related to visual and graphical properties: the *Actor* image and other transform options such as flip, repeat and displacement.
- **Text**. The *Actors* are able to show text in a certain position with a particular font and style.
- **Sound**. A sound can be associated with the *Actor*. It presents some properties to allow the modulation of its volume, some options to determine the moment it starts and if it is in loop.
- **Physics**. These properties are related to the type of physics body: dynamic, kinematic or static. Other characteristics such as the velocity and the properties that depend on the material, such as density or friction, are also included in this category.
- **Custom**. Additionally, it is possible to store information on variables to customize the games even further.

These properties can be modified from the behaviour rules of the actor itself or other actors of the same scene.

3.2.2. *Actor* rules

Actors are in charge of managing every event that takes place in the *Game*. For this purpose, a rule system has been devised in order to define the logic and the interactive behaviours of the *Game.* A behaviour rule is determined by a decision tree system [36] driven by a reduced set of *Actions* and *Conditions*, ready to be executed if the flow passes through them according to whether the *Conditions* are met or are not. An example of a rule is shown in Figure 3.

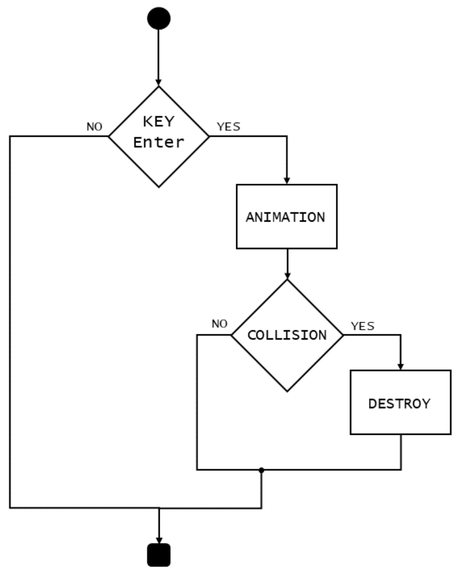


Figure 3. - Diagram of a decision tree.

The *Rule* structure is defined by two elements: *Actions* and *Conditions* arranged in a decision tree. Both *Actions* and *Conditions* are prepared to work with arithmetical expressions and with mathematical functions: *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *sqrt*, *random*, etc; and the data types supported by this system are *numbers* and *booleans*.

Actions are the defining elements of the specific behaviours of the *Actors*. A list of predefined actions is available to the game designer. In this sense, a thorough review has been carried out to generate a stable simplification in order to arrange a minimum set of predetermined *Actions* and thus to simplify the game logic. The study has resulted in fifteen *Actions*, some of which are described below:

- **Edit**. To change every *Property* by a specific value or by the result of an arithmetical expression. It works equally for the *Game*, the *Scene* or the *Actor*.
- **Animate**. To execute animations by arranging sprites on a specific frames-per-second rate.
- **Move / Move To**. To move the *Actor* a certain number of units on screen. It has a related *Action* called *Move To*, which causes it to travel towards a specific position or *Actor*.
- **Destroy**. This *Action* deletes the *Actor* from the *Scene* when is triggered.
- **Push / Push To / Torque**. To apply forces to the *Actor*. It also has a related *Action* called *Push To* and another pack called *Torque* that works with the same concept to apply angular forces.
- **Rotate / Rotate To**. To rotate the *Actor* a certain number of degrees. There is also *Rotate to Action* which allows rotating until reaching an angle.
- **Add Scene / Remove Scene / Go To Scene**. To handle the management of *Scenes* with *Add Scene*, *Remove Scene* and *Go to Scene*.
- **Spawn**. An automatic *Actor* copy generator.
- **Sound**. To start a sound by *Play sound*.

The conditions allow defining the execution flow for the actions. The scripting system includes the following types of conditions:

- **Compare**. This compares data values or expressions from any *Game*, *Scene* or *Actor Properties* with another value or expression.
- **Check**. To check if a boolean *Property* is met or it is not.
- **Collision**. This checks whether two *Actors* are colliding. It relies on the *Actor's* collision shape.
- **Keyboard**. To capture which key has been pressed on the keyboard.
- **Touch**. To manage the user interaction with mouse or touch events through tactile devices.
- **Timer**. To perform sets of *Actions* after a certain number of seconds.

These elements can provide basic coding knowledge without giving up complex game development, by only considering that the *Game Loop* implements the evaluation of the behaviour of every *Actor* on each iteration. Furthermore, in an attempt to enhance the simplification of the game development, the usage of logic expressions and complex data structures such as matrices, arrays or other complex structures like trees or graphs to create the *Actor*'s *Rules* have been discarded.

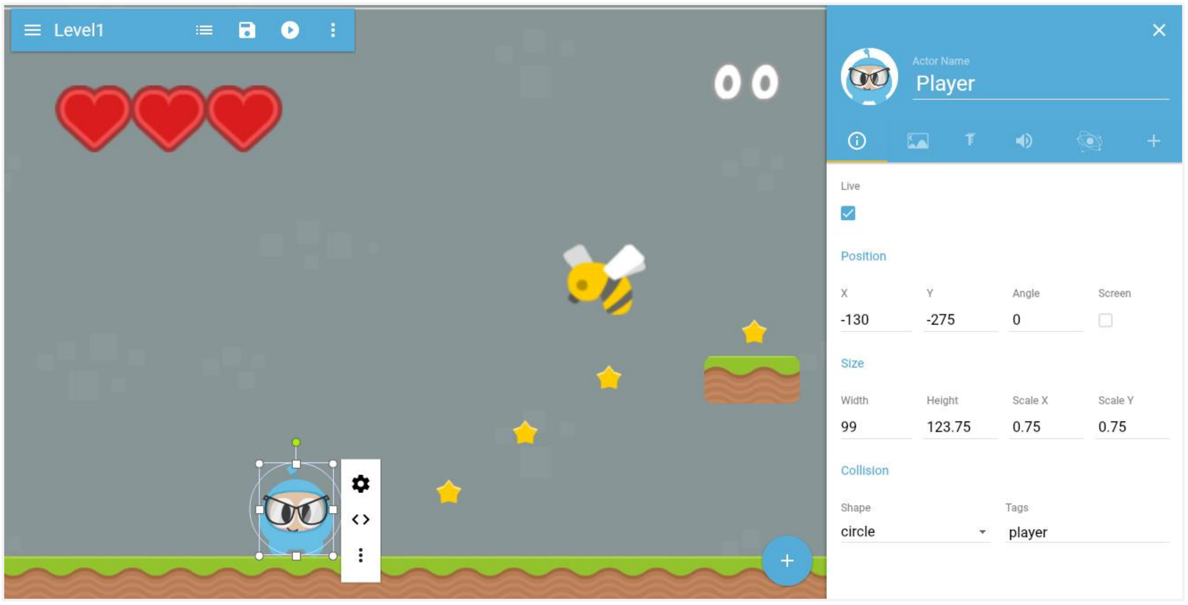
3.3. The game editor

In order to democratize game development, it is necessary to aim at the easiest and most accessible way to present the development tools for game developers. For this reason, a game editor has been created, taking as its starting point some of the main features of the user interface literature and their subsequent integration within the whole *SGE* environment.

3.3.1. The *Scene* editor

The game editor has been developed by adopting the user interface design and interaction concepts behind a slide-show software [49]. This kind of application is conceived to be easy to use and has a very similar structure to games: several slides that can be compared to the game levels or *Scenes*, object placement and properties such as *Actors* and their *Properties*, and interactive event animations as the behaviour *Rules*.

Another design feature is related to its visual appearance. In this sense, *Google Material Design* [22] has been the core of this interface specification, where the design definition is aimed at multi-device applications with fluid navigation. An example of this interface is shown in Figure 4 with a *Platform Game*. The *Scene* list for this game is available on the navigation menu. The canvas draws the environment designed for the first *Scene* called *Level 1*. The *Actor*, represented by the blue character, is selected and its gizmo and its associated graphical menu are visible. Through this menu, the user can access the *Actor's* properties and its *Rules* list. These features are also shown in Figure 4, where, after selecting the *Actor*'s properties icon, a modular panel comes out from the right-hand side of the screen showing all the properties arranged on tabs in accordance with the grouping presented in section 3.2.1.



Actor Name
Player

Live ✓

Position

X	Y	Angle	Screen
-130	-275	0	☐

Size

Width	Height	Scale X	Scale Y
99	123.75	0.75	0.75

Collision

Shape	Tags
circle	player

Figure 4. - The Game Editor.

3.3.2. The *Rule* editor

The roles performed by the *Actors* are defined in an edition tool environment, called *Rule* editor. This has been based on a simple decision tree, where the *Actions* and *Conditions* are arranged visually and configured until the desired behaviour is fulfilled.

Similarly to the *Actor*'s *Properties*, this *Rule* editor can be accessed via the *Actor*'s context menu. Figure 5 shows a decision tree editor, through which the behaviours can be composed by arranging the *Actions* and *Conditions*. These elements are accessible from two yellow and blue buttons, which serve as shortcuts to the set of *Actions* and *Conditions*, respectively. After selecting one of the elements, that panel comes out showing the *Properties* and the options determined for it. As an example of these features, Figure 5 shows a decision tree performing a *Jump* behaviour for a specific *Actor*. The Actor must be on the floor to be able to jump. Initially, a check is performed to determine whether the *Actor* is colliding with the floor and then, if the *Up* key is pressed, the jump action is produced by setting the *Actor*'s y-axis velocity at 300 pixels per second. If any of these *Conditions* are not met, the flow will travel through its left branch and no *Action* will be performed. Each action or condition is configured by setting their *Properties* in a panel which appears on the right when it is selected. In order to facilitate understanding of the *Rule* for Jump, a pseudocode version is annexed in Algorithm 1.

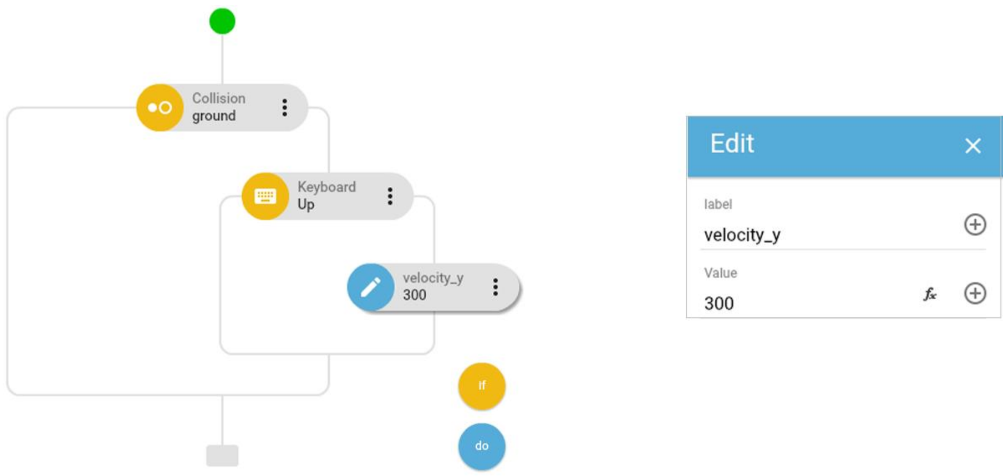


Figure 5. - Appearance of the Rule editor when the action Edit is being configured to perform the "Jump" role. This example establishes the property "velocity_y" at 300 pixels/second.

Algorithm 1 - *Example of the associated pseudocode for the rule in Figure 5.*

```
If (collision(Ground))
    If (KeyCode.Up)
        velocity_y = 300;
    End
End
```

4. Game Example: *Candy Crush*

Programming using *SGE* has some differences compared to the use of conventional programming paradigms. The creation of video games without the use of complex hierarchies and data structures such as vectors or matrices can be a challenge for expert programmers accustomed to using them in conventional programming languages. The impossibility of using loops when programming the rules within the actors, forces to assume the *Game Loop* as a way to perform the iterations. In this sense, programming video games with the proposed engine is more related to agent-oriented programming, where individual agents with similar properties interact with each other and with the environment to solve different types of problems. This way of programming allows to develop computational thinking in a more similar way to the interaction between people and can be easier for non-expert users. In order to illustrate the differences in the way a typical matrix problem can be solved, the creation of a *Candy Crush*-like game is proposed. It has been developed with SGE and *Scratch*, due to its orientation to the conventional learning of programming and its usage in some 2D game engines. This game is one of the 2D games with more complex mechanics that can be performed and its development with *SGE* demonstrates its great potential.

The *Candy Crush* game [9] is a match-three puzzle video game, in which the players have to swap candies on a game board to produce combinations of three or more with the same colour. This fact makes the candies disappear and allows the player to win points and to reach goals. Experimented computer developers usually use a matrix structure to store the game board, but a non-experienced user does not think in such complex data structures. Instead of using a matrix to handle the candies, each *Candy* becomes an *Actor* with its *Properties* and behaviour *Rules* that interacts with the rest of the *Actors* in the *Scene*. In this case, the *Candies* are organized in a matrix-like grid in the space, but there is no matrix as a data structure.

The size of the board for this implementation of the *Candy Crush* game is five elements in width and in height, and the *Candies* can display four different possible colours. Once the game board has been established with a random configuration, similar to the one shown in Figure 6, the *Candy Crush* game initialization is performed. The board is then filled with randomly coloured *Candies*.

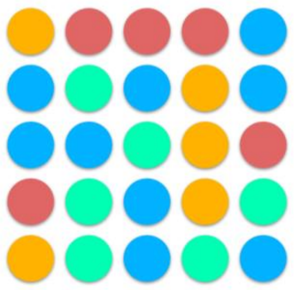


Figure 6. - Example of Candy Crush initialization.

Next, the implementation of the basic mechanics of the game using *Scratch* and later using *SGE* is considered.

4.1. *Scratch* solution

The initialization procedure establishes the game board assigning a random texture with a specific colour to each *Candy*. The required code would have an instruction sequence similar to the one shown in Algorithm 2.

Algorithm 2 - *Candy*'s colour initialization in pseudocode for *Scratch*.

```
i = 0;
j = 0;
Repeat Until (i < rows)
    Repeat Until (j < columns)
        color = random(1,4);
        Candy[i][j] = color;
        j = j + 1;
    End
    i = i + 1;
End
```

Continuing with the example, Algorithm 3 shows the function to remove three or more *Candies* in a row when they have the same colour. There, the algorithm checks each matrix row to see if there is a sequential combination of three or more *Candies* with the same colour. If that is the case, these *Candies* are removed.

Algorithm 3 - *Candy* checker and eraser in pseudocode for *Scratch*.

```
i = 0;
j = 0;
Repeat Until (i < rows)
    count = 1;
    remove = 0;
    Repeat Until (j < columns)
        If (Candy[i][j] == Candy[i][j+1])
            count = count + 1;
            If (j == columns - 1 && count >= 3)
                    remove = count;
                position = j + 1;
                End
            Else
                If (count >= 3)
                    remove = count;
                position = j;
                End
                count = 1;
            Repeat (remove > 0)
                remove = remove - 1;
                delete(Candy[i][pos - remove]);
            End
        End
        j = j + 1;
    End
    i = i + 1;
End
```

4.2. *SGE* solution

The same algorithm idea but performed with the *SGE* makes a significant change. Every *Candy* is represented by an *Actor*. Firstly, the developer has to set up the initial behaviour of *Candy* to create a random colour, essentially by making it choose between four images with four different colour textures. Then, it is only necessary to copy and paste the *Candy Actor* as many times as needed until the grid shape shown in Figure 6 appears. This sets an arrangement of *Actors* who, during their initialization, choose a random image at the beginning of the *Game*. As can be appreciated, a matrix data structure is no longer necessary to initialize each *Actor* in the *Game's* creation process. Instead, each *Candy* initializes itself as it is presented in Algorithm 4.

Algorithm 4 - Colour initialization of *Candy*.

```
If (initialization)
    image = image[random(0, 3)];
    initialization = false;   // Initially the actor has this custom property equal to true
End
```

The algorithm to remove three or more consecutive *Candies* with the same colour is set up with three different *Actors*, which are not visible in the execution because they require no image.

- **Tracker**. This *Actor* is responsible for travelling from left to right crossing over every *Candy* in each row, checking its colour properties and counting the ones arranged consecutively. Initially, its colour value is empty and it obtains the one from the first *Candy* to collide with it. Its logic is presented in Algorithm 5.

  Algorithm 5 - *Actor Tracker* behaviour rule.

  ```
  move(x + 1);
  If (colour == Candy.colour)
      counter++;
  Else
      counter = 0;
      colour = Candy.colour;
  End
  If (x > columns)
      destroy(Tracker);
  End
  ```

- **Eraser**. This *Actor* is spawned by *Candy* actors. If a *Candy* has a different colour than the one being tested and also the *Tracker* counter is 3 or greater, an *Eraser* is created. It receives information about how many *Candies* have to be destroyed. Its logic is presented in Algorithm 6.

Algorithm 6 - *Eraser Actor* behaviour rule.

```
move(x - 1);
If (collision(Candy))
    destroy = destroy - 1;
    If (destroy == 0)
        destroy(Eraser);
    End
End
```

- **Candy**. In addition to that of the initialization, this Actor has two extra Rules. The first one is the colour checker: if the Tracker collides with the Candy, it checks whether it has the same colour as the one being counted, as presented in Algorithm 7. If this is the case, the Tracker counter is increased by one. If not, a check is performed to see if the Tracker counter is equal to or greater than 3, and if it is true an Eraser is created. The second Rule is in charge of destroying the Candy if an Eraser collides with it, as also shown in Algorithm 8.

Algorithm 7 - *Candy Actor* behaviour *Rule* for the colour check.

```
If (Collision(Tracker))
    If (color == Tracker.colour)
        Tracker.counter += 1;
    Else
        If (Tracker.counter >= 3)
            Eraser.destroy = Tracker.counter;
            spawn(Eraser)();
        End
        Tracker.counter = 0;
        Tracker.colour = colour;
    End
End
```

As can be observed, in this way, complex data structures and their usage are avoided, breaking each role or *Action* in the *Actor*'s behaviours down and making them interact with each other. As can be seen, both ways of reaching a solution are two comparable approaches to solving an algorithm problem, but conceptually it is a closer model to the target user's way of thinking [55, 56].

Algorithm 8 - *Candy Actor* behaviour *Rule* for the elimination of the *Candy*.

```
If (collision(Eraser))
    destroy(Candy);
End
```

If the algorithms are analyzed in each case, using *Scratch* and *SGE*, it can be seen that the behaviours that appear in each of the actors in SGE exist the *Scratch* (see Algorithm 3), however, in *SGE*, they appear separately and therefore so much easier to understand.

5. User experience

To validate the game engine, an evaluation was carried out with children. The reason for using children as evaluators is that they are the perfect target user for these tools, that is, people familiar with technology but with the lack of the necessary knowledge to develop their own ideas in a videogame [25]. The procedure consisted in having them develop their own videogame and asking them some broader questions to generate a deeper and more specific understanding of the problems encountered during the creation of a videogame [21]. The children were organized by age into independent groups, each with a different arcade game and the evaluation was based on individual acceptance tests [12, 13]. In addition, a comparative test between SGE and Scratch was also performed. It was validated by the Wilcoxon Signed-Rank (WSR) test [29].

5.1. Objectives and hypothesis

The main assumption of this work is based on the perception that the game development process has the potential to be made easier, on the estimation that an arcade game can be made with a reduced set of *Actions* and *Conditions*, and also on the notion that the system's usage has to match the profiles of non-technical users. Implicitly, it is assumed that if these statements are met, then the *SGE* system has to be perceived as easy to use. It is necessary to verify that a tool such as the one presented here facilitates game creation for the users and it is found to be useful, along with the hypothesis that the rule editor is easy to understand. A summary of the objectives and hypothesis is presented in Table 2.

Table 2 - Objectives and hypotheses for the experiment.

Objectives	**Hypotheses**
O1 - The tool has to make game development easier.	H1 - This tool makes the creation of games easier for non-programmers.
	H2 - The tool is found to be useful.
O2 - The tool has to be able to create arcade games using a reduced set of *Actions* and *Conditions*.	H3 - The system of rules is easy to use.
	H4 - The tool is easy to understand.
	H5 - The tool is easier to use than other similar tools.

5.2. Protocol

A total of one hundred and twenty children attending a summer camp related to technology served as the subjects for this experiment. Their ages ranged between seven and fourteen years. As regards gender, seventy-four of them were boys and forty-six were girls. Previously, their parents were informed about the aim and the method of the experiment, and they gave their informed consent for their sons and daughters to take part in this study. Regarding the level of previous experience of programming, all the children stated that they had never created a computer game.

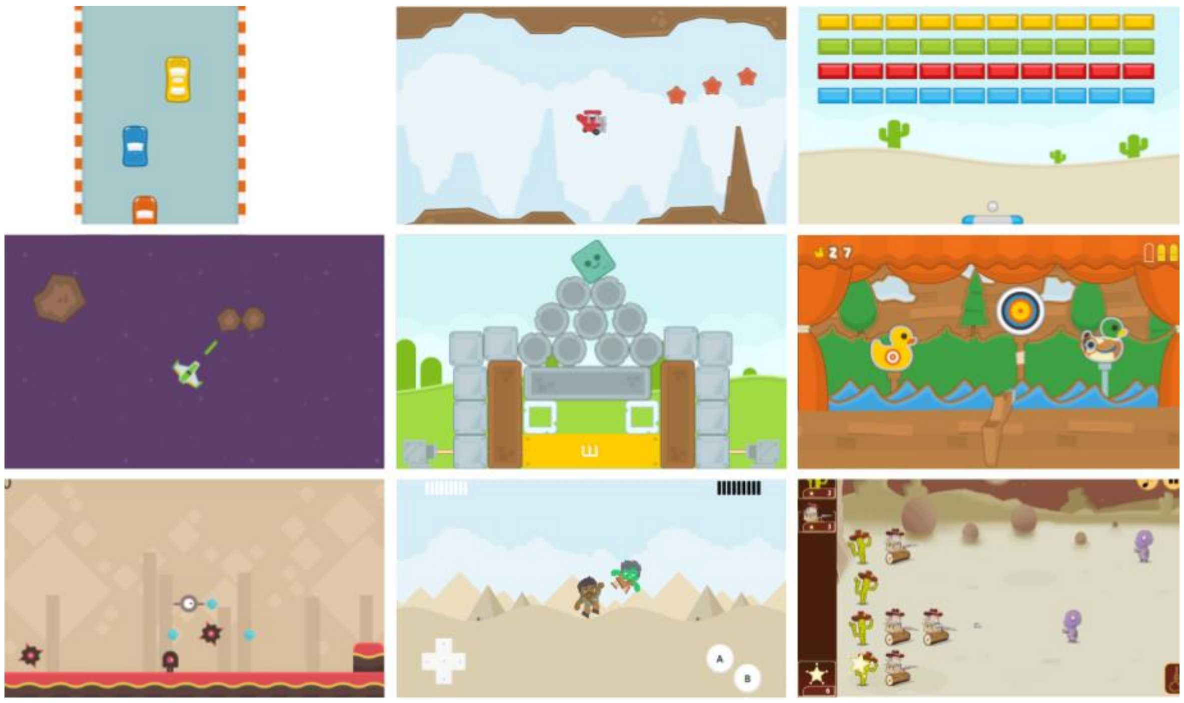


Figure 7. - Example of arcade games performed during the tests (sorted by rows according to their difficulty level).

The children had exactly twenty hours to work on their games. They were asked to design one of nine different arcade games according to their age. Figure 7 shows nine captures taken on the final versions of the games, the images are organized in the same order as described below, starting from the upper-left corner:

- **Asteroids**. A classic space and third-person shooter where the asteroids are subdivided after being hit.
- **Arkanoid**. A classic puzzle game where the ball has to bounce until there are no bricks left.
- **Cowboys vs Aliens**. Tower defence game where the Cowboys defend their land from the Aliens.
- **Car Racing**. A vertical scroll game where the goal is to advance as far as possible avoiding collisions with other cars and the road boundaries.
- **Tappy Plane**. A horizontal scroll game, the goal of which is to travel for as long as possible avoiding collisions with the environment.
- **Ducks**. A first-person shooter game based on the classic ones found in fairgrounds.
- **Abstract Adventure**. A classic platform game where the goal is to elude enemies and traps at the same time as every coin in the scene is gathered.
- **Blocks**. A physics game where the goal is to keep the player on the platform.
- **Combat**. A fighting game between two players.

The video games were developed individually. The children were arranged in groups of between ten and fifteen members each group working on one of the nine games that had previously been assigned to them. At the end of the experiment, they were asked to answer some simple questions about comfort with the tool and the understanding of the method. The response to each question was evaluated on a 5-point Likert scale [29]. The questions were adapted to their age and were focused on the complexity of the game concept, in order to obtain a measurement of the Perceived Usefulness (PU) and Perceived Ease-of-Use (PEOU) ratio. Questions asked if they believed that using a particular system would require less effort and would enhance their job performance. The questions are presented in Table 3.

Moreover, twenty-three of these children were asked to develop another video game using *Scratch* instead of *SGE*. At the end of the twenty hours' work, they filled in a questionnaire to compare *SGE* and *Scratch*.

They answered a survey with three options: 1 represents a preference for *SGE*, -1 indicates that they prefer *Scratch*, and finally 0 indicates that they did not answer the question. The questions are shown in Table 4. These tests were assessed with the average and rank based on the WSR test.

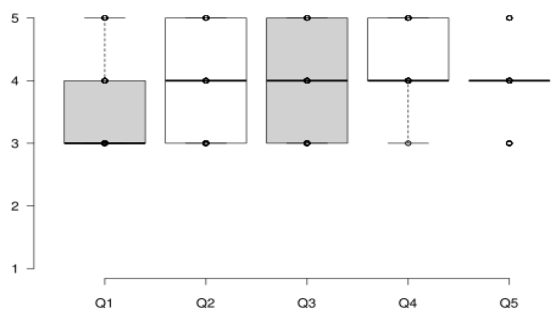
5.3. Results

First, the evaluation of the *SGE* is presented in Table 3 and it is supported by the survey average and the standard deviation values. In addition, the distribution of the data can be observed graphically in Figure 8. Concerning the PU analysis, it can be seen that all the values vary in the range between 3 and 5, which represents a satisfactory evaluation for this game engine. Question Q1, concerning ease of use, is the one that has the worst average value: 3.66. This is because it is the first time the user tackles the design of a video game and considers that it is not such an easy task. However, they generally found the *SGE* system to be useful to perform this task (Q2) and quite easy to learn (Q5). The most remarkable data was obtained regarding question Q4, which indicates that they felt comfortable using this game engine. They also rated the program workflow of the SGE as easy to understand (Q3).

The results regarding PEOU were even better, confirming that the system's environment encourages the child’s creativity for new game designs and their satisfaction with the games produced. The means of the data obtained in questions related to their attitude towards use or intention of use are in a range between 4 and 5. This fact confirms the proposed hypothesis because most of the children felt that the experience was worthwhile, rating each question in this category with a mean value of 4.48.

Table 3 - Items and results for the PU and PEOU surveys.

Questions		**Average**	**S.D.**
PU			
Q1	Has it been easy to create games?	3.66	0.77
Q2	Have you found it useful?	3.95	0.86
Q3	Have you understood the program's workflow?	4.00	0.77
Q4	Have you felt comfortable using the program?	4.44	0.56
Q5	Has it been easy to learn how to use the program?	3.98	0.61
PEOU			
Q6	Do you feel capable of making new games with the program?	4.31	0.80
Q7	Are you happy with the games you have created?	4.69	0.46
Q8	Would you like to continue making games like these ones?	4.45	0.74



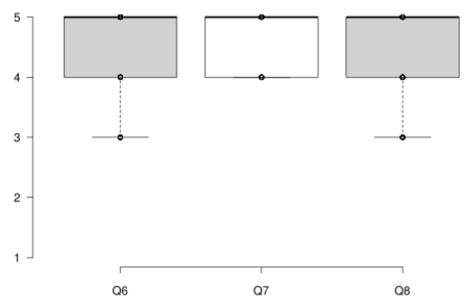


Figure 8. - Graphical distribution of the data obtained for PU (left) and PEOU (right).

Concerning the evaluation of the tests that compare *SGE* and *Scratch,* all the data obtained are shown in Table 4. The average of this data is also shown. Regarding the WSR test, the signed-ranks of the data are calculated and shown in the rightmost column. These values have been analysed to evaluate the WSR test. Taking into account that twenty-three children (*n=23*) answered these questions and using its proper significance level (*a=0.05*), the result of the test confirms that the data are statistically significant and so there is a difference between the two programming tools.

After analysing the results obtained, it can be said that the users feel the *SGE* be a more useful and comfortable tool, thus highlighting the result of question S4 about the satisfaction with the game that was finally designed. Question S7 also confirms that they prefer to work with the *SGE* system than to use *Scratch*. However, the results obtained in question S6 show that *Scratch* has been perceived as an easier tool for creating loops in the video game. It was said that this concept is not explicitly included in the proposed game engine although it can be programmed by configuring the flowcharts. The fact that the users can develop their own loops in the Scratch code made the code that they built more understandable to them.

Table 4 - Comparative test on user acceptance between *SGE* and *Scratch*.

Questions		**Average**	**Signed Rank**
S1	Which one has been the fastest to learn?	0.46	1.00
S2	Which one do you think you could create a new game with?	0.54	2.00
S3	Which one have you found easier to understand?	0.71	5.00
S4	Which one makes you feel happiest with the results?	0.83	6.00
S5	Which one do you think has been most useful to you?	0.58	4.00
S6	Which one has given you more facilities to perform a loop?	-0.54	-2.00
S7	Which would you like to continue working with?	0.88	7.00

5.4. Discussion

The hypotheses set out regarding the proposed system have all been confirmed with the data obtained from the user tests. After the twenty hours of the experiment, each child was able to go home with his/her own video game completely finished. This fact confirms hypothesis H1 because the users were children without any knowledge of programming and they were capable of developing a complete 2D arcade video game. The fact that every game was completed on time along with the test results makes it possible to claim that the tool is easy to learn and to comprehend, which at the very least enforces hypotheses H3 and H4. Furthermore, the results also show that they generally agreed that the experience was perceived as helpful to them, which lends support to the validation of hypothesis H2. On another note, no significant usability problems were detected, although some syntax and vocabulary comprehension issues have been recognized as obstacles hindering the comprehension of the system. Despite this, the children said that it was easy to use and understand. Most of them even stated that they were thrilled with their creations, and they were looking forward to starting their own ideas.

Furthermore, the results from the comparative study performed with *Scratch* and *SGE* proved that the *SGE* system is widely perceived as easier to use and to understand than other visual behavioural specification environments such as *Scratch*. These results reinforce the validation of H1 and support the validation of H5.

6. Conclusions and future work

This work presents a game engine, *SGE*, which addresses the aim of making the video game development process easier for people with non-technical profiles. Essentially, the way these interactive applications are developed has been changed from the traditional method of programming. Instead of using the data structures of some programming language, the *SGE* proposes a reduced set of *Actions* and *Conditions* to develop arcade games. These elements are combined in flow charts that determine the actions carried out by the actors, the main component of this game development system.

The game specification has been restricted in order to achieve a reduced game engine architecture and a straightforward game editor. The game development has been performed without relying on matrices, loops and other complex data structures and through a reduced set of *Actions* and *Conditions*. Some of the tests performed have been conducted in order to evaluate the simplicity of *SGE* in performing this task. The results demonstrate that, by using this game engine, users are able to develop games in an easier way. The game engine proposed in this research has been tested by children without any knowledge of programming, who develop games using this system and with the popular visual programming tool Scratch. Results obtained from these experiments show that *SGE* is perceived as a useful and easy-to-use tool for game development and for learning to programme, even when compared with *Scratch*.

The experience acquired during this work will push us in the future to continue the development of this project on a 3D game engine. Similarly, it will have to be easy to use and ready to enable non-programming people to develop games with visual behaviour descriptors. Currently, it is already easy to create 3D environments or very simple games with editors such as *Minecraft* but it is still necessary to have advanced knowledge to be able to create games with development frameworks such as *Unity*. In this sense, bridging the gap between these two types of editors is going to be one of the main research topics from now on, in order to allow users to build 3D arcade games without any knowledge of coding.

Acknowledgement

This work has been supported by the Ministry of Science and Technology (TIN2016- 75866-C3-1-R) and the Universitat Jaume I research project (UJI-B2018-56). Furthermore, this work has been made possible thanks to the graphics resources created by Kenney from Kenney.nl.

References

1. Ampatzoglou A, Stamelos I (2010) Software engineering research for computer games: A systematic review, Inf. Softw. Technol. 52 (9).
2. Anderson, E. F., Engel, S., McLoughlin, L., & Comninos, P. The case for research in game engine architecture, 2008; 228-231.
3. Bácsi, S., & Mezei, G. (2019). Towards a Classification to Facilitate the Design of Domain-Specific Visual Languages. Acta Cybernetica, 24(1), 5-16.
4. Biswas P. K (2008) Towards an agent-oriented approach to conceptualization, Appl. Soft Comput. 8 (1).

5. Blackwell, A. F. (1996, September). Metacognitive theories of visual programming: what do we think we are doing?. In Proceedings 1996 IEEE Symposium on Visual Languages (pp. 240-246). IEEE.
6. Catto, E. (2011). Box2D: A 2D physics engine for games.
7. Chang, S. E. (2005). Computer anxiety and perception of task complexity in learning programming-related skills. Computers in Human Behavior, 21(5), 713-728.
8. Chao, P. Y. (2016). Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. Computers & Education, 95, 202-215.
9. Chen C, Leung L (2016) Are you addicted to candy crush saga? An exploratory study linking psychological factors to mobile social game addiction, Telemat. Inf. 33 (4).
10. Chen, W. K., & Cheng, Y. C. (2007). Teaching object-oriented programming laboratory with computer game programming. IEEE Transactions on Education, 50(3), 197-203.
11. Correa, J. D. C (2015). Digitopolis II: Creation of video games GDevelop. Correa, J. D. C (ed).Bogotá.
12. Davis F. D (1989) Perceived usefulness, perceived ease of use, and user acceptance of information technology, MIS Q. 13 (3)
13. Davis F. D, Venkatesh V (2004) Toward pre-prototype user acceptance testing of new information systems: implications for software project management, IEEE Transactions on Engineering Management 31(46).
14. Dekhane, S., & Xu, X. (2012). Engaging students in computing using GameSalad: a pilot study. Journal of Computing Sciences in Colleges, 28(2), 117-123.
15. Deloura M (2000) Game Programming Gems, Charles River Media, Inc., Rockland, MA, USA.
16. Flowlab. https://flowlab.io/ [Online; Last accessed: 2019-1-8] (January 2019).
17. Folmer E (2007) Component-based game development - a solution to escalating costs and expanding deadlines? In: Schmidt, H.W - Component-Based Software Engineering. Springer Berlin Heidelberg, pp 66-73.
18. Furtado A. W, Santos A. L (2006) Using domain-specific modelling towards computer games development industrialization. In: The 6th OOPSL. A workshop on domain-specific modelling.
19. Furtado A.W. B, Santos A. L. M, Ramalho G. L, de Almeida E. S (2011) Improving digital game development with software product lines, IEEE Software 28 (5).
20. Game Maker. YoYo Games. http://www.yoyogames.com [Online; Last accessed: 2019-1-8] (January 2019).
21. Gilchrist, V. J (1992) Key informant interviews. In B. F. Crabtree & W. L. Miller (Eds.), Research methods for primary care, Thousand Oaks, CA, US: Sage Publications, Inc. 3:70-89.
22. Google Design. https://design.google [Online; Last accessed: 2019-1-8] (January 2019).
23. Gregory J (2014) Game Engine Architecture, 2nd Edition, A. K. Peters, Ltd., Natick, MA, USA.
24. Guo B, Fujimura R, Zhang D, Imai M (2012) Design-in-play: improving the variability of indoor pervasive games, Multimedia Tools and Applications 59 (1).
25. Hanks K, Odom W, Roedl D, Blevis E (2008) Sustainable millennials: Attitudes towards sustainability and the material effects of interactive technologies, In: Conference on Human Factors in Computing Systems - Proceedings, pp 333-342
26. Kamadi, V. V., Allam, A. R., & Thummala, S. M. (2016). A computational intelligence technique for the effective diagnosis of diabetic patients using principal component analysis (PCA) and modified fuzzy SLIQ decision tree approach. Applied Soft Computing, 49, 137-145.
27. Kiper, J. D., Howard, E., & Ames, C. (1997). Criteria for evaluation of visual programming languages. Journal of Visual Languages & Computing, 8(2), 175-192.
28. Koulouri, T., Lauria, S., & Macredie, R. D. (2015). Teaching introductory programming: A quantitative evaluation of different approaches. ACM Transactions on Computing Education (TOCE), 14(4), 26.
29. Lazar J, Feng J. H, Hochheiser H (2010) Research Methods in Human-Computer Interaction, Wiley.
30. Liu C.-H, Lin J, Wilson D, Hemmenway E, Hasson Z, Barnett Y. (2014) Making games a "snap" with stencyl: A summer computing workshop for k-12 teachers. In: Proceedings of the 45th ACM Technical Symposium on Computer Science Education. ACM, New York, NY, USA.

31. Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. ACM Transactions on Computing Education (TOCE), 10(4), 16.
32. Matter.js. http://brm.io/matter-js [Online; Last accessed: 2019-8-24] (August 2019).
33. Menard M (2011) Game Development with Unity, 1st Edition, Course Technology Press, Boston, MA, US.
34. Messaoudi, F., Simon, G., & Ksentini, A. (2015, December). Dissecting games engines: The case of Unity3D. In 2015 International Workshop on Network and Systems Support for Games (NetGames) (pp. 1-6). IEEE.
35. Millington I (2010) How to Build a Robust Commercial-Grade Physics Engine for Your Game, In: Game Physics Engine Development, 2nd Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
36. Millington I, Funge J (2009) Artificial Intelligence for Games, 2nd Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
37. Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming—views of students and tutors. Education and Information Technologies, 7(1), 55-66.
38. Nuñez-Valdez E. R, Sanjuan-Martinez O, Bustelo C. P. G, Lovelle J. M. C, Infante-Hernández G (2013) Gade4all: Developing multi-platform video games based on domain specific languages and model driven engineering, International Journal of Interactive Multimedia and Artificial Intelligence 2 (2).
39. Ouahbi, I., Kaddari, F., Darhmaoui, H., Elachqar, A., & Lahmine, S. (2015). Learning basic programming concepts by creating games with Scratch programming environment. Procedia-Social and Behavioral Sciences, 191, 1479-1482.
40. Powers, K., Gross, P., Cooper, S., McNally, M., Goldman, K. J., Proulx, V., & Carlisle, M. (2006, March). Tools for teaching introductory programming: what works?. In ACM SIGCSE Bulletin (Vol. 38, No. 1, pp. 560-561). ACM.
41. Reyno E. M, Cubel J. A. C (2008) Model-driven game development: 2d platform game prototyping. In: GAMEON.
42. Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. Computer science education, 13(2), 137-172.
43. Roy, K., Rousse, W. C., & DeMeritt, D. B. (2012, October). Comparing the mobile novice programming environments: App Inventor for Android vs. GameSalad. In 2012 Frontiers in Education Conference Proceedings (pp. 1-6). IEEE.
44. RPG Maker. https://www.rpgmakerweb.com [Online; Last accessed: 2019-1-8] (January 2019).
45. Salen K, Zimmerman E (2006) The Game Design Reader: A Rules of Play Anthology. MIT Press.
46. Stemkoski L, Leider E (2017) Game Development with Construct 2: From Design to Realization, Apress.
47. Stencyl. http://www.stencyl.com [Online; Last accessed: 2019-1-8] (January 2019).
48. Treglia D (2002) Game Programming Gems 3, Game Programming Gems Series, Charles River Media.
49. Tufte E. R (2006) The Cognitive Style of PowerPoint: Pitching Out Corrupts Within, 2nd Edition.
50. Unity 3D Engine. Unity. http://www.unity3d.com [Online; Last accessed: 2019-1-8] (January 2019).
51. Unreal Engine, Epic games. http://www.unrealengine.com [Online; Last accessed: 2019-1-8] (January 2019).
52. Valcasara N (2005) Unreal Engine Game Development Blueprints, Packt Publishing.
53. Van der Spuy, R. (2015). Learn Pixi. js. Apress.
54. Williams D (2002) Structure and competition in the u.s. home video game industry. International Journal on Media Management. 4 (1).
55. Wing, J. M. (2008). Computational thinking and thinking about computing. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 366(1881), 3717-3725.
56. Wing, J. M. (2006). Computational thinking. Communications of the ACM, 49(3), 33-35.
57. Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: Theory and practice. The knowledge engineering review, 10(2), 115-152.

58. Yao, J., Pan, Y., Yang, S., Chen, Y., & Li, Y. (2019). Detecting Fraudulent Financial Statements for the Sustainable Development of the Socio-Economy in China: A Multi-Analytic Approach. Sustainability, 11(6), 1579.
59. Zarraonandia T, Diaz P, Aedo I (2017) Using combinatorial creativity to support end-user design of digital games, Multimedia Tools and Applications 76 (6).
60. Zarraonandia T, Diaz P, Aedo I, Ruiz M. R (2015) Designing educational games through a conceptual model based on rules and scenarios, Multimedia Tools and Applications 74 (13).