Multimedia Systems (2025) 31:70
https://doi.org/10.1007/s00530-024-01574-8

REGULAR PAPER

GameScript: a simplified scripting language for video game development

Carlos Marín-Lora[1,2] · **Miguel Chover**[1]

Received: 1 March 2024 / Accepted: 7 November 2024 / Published online: 15 January 2025

Abstract
The process of video game development is complex, requiring proficiency in various technologies for design and implementation, including mastery of game engines and associated programming languages. Defining the behaviors of game elements within the necessary subprocesses poses a significant challenge. This challenge arises partly due to the utilization of general-purpose programming languages, which offer a wide range of options and allow tasks to be approached from different angles. This study introduces GameScript, a simplified scripting language designed for video game development. Based on principles of structured programming and inspired by the While language, GameScript focuses on essential game behavior definitions using a limited set of statements, including IF-THEN-ELSE selection and operations related to game elements to create and destroy them, or to modify properties. By using only numeric variables and simple arithmetic and relational operators, GameScript simplifies data handling and allows for the creation of various arcade games with straightforward syntax. Integrated into a multi-agent system game engine, GameScript has been validated through the successful implementation of diverse arcade game mechanics, demonstrating its effectiveness.

Keywords Scripting language · Game logic · Game engine · Game development

1 Introduction

Since the early 1990s, the video game industry has relied on the use of game engines for its development. Currently, a multitude of engines allow for the creation of both 2D and 3D games. Among the most popular are engines such as Unity, Construct, Game Maker, Twine, RPG Maker, Bitsy, PICO-8, Unreal Engine, Godot, and Ren'Py [1]. These environments incorporate a set of tools that facilitate content creation. These tools include organization through scene graphs, definition of game objects based on components, and specification of game element behavior through scripting systems [2].

✉ Carlos Marín-Lora
cmarin@uji.es

Miguel Chover
chover@uji.es

[1] Institute of New Imaging Technologies, Universitat Jaume I, Avda. Sos Baynat, s/n, 12071 Castellón, Spain

[2] Valgrai: Valencian Graduate School and Research Network of Artificial Intelligence, Camí de Vera s/n, 46022 Valencia, Spain

Scripting systems employed in game engines exhibit wide diversity and can be categorized into three main types: initialization scripts, event-driven scripts, and scripts defined through conventional programming languages [3]. Moreover, these scripts can be either loop scripts or regular scripts, depending on whether they are executed repeatedly throughout the game loop or only once. These scripts can be utilized to program games by applying different design patterns [4].

They offer a higher level of abstraction, making them a potent alternative to conventional programming languages. They reduce the effort required to create complex instructions [3] but may sometimes impact game performance at runtime [5]. Furthermore, there is a growing trend towards using different scripting languages within the same tool [6], as game engines continue to integrate diverse technologies. However, creating games using this type of scripting system still requires high technical expertise. Moreover, there is no standardization in terms of operation and language, as it depends on the available game engine.

In this context, this work addresses the complexity of current game development tools by introducing GameScript, a simplified scripting language aimed at defining video game

component behaviors. The main contributions of this study are:

- **Specification of syntax and semantics**. The language is based on the Structured Program Theorem [7] and its derived studies. The syntax and semantics of the language have been inspired by the While language [8], a small programming language used for theoretical analysis of imperative programming language semantics and validation of programming language prototype implementations [9].
- **Integration into a game engine.** The language has been incorporated into an engine based on multi-agent systems (MAS) [10]. However, it can serve as a conceptual foundation for logical specification in other game engines, following the research line proposed by Marín-Lora et al. [11].
- **Validation through game implementation**. The language has been tested for behavior specification in a large number of arcade games [12].

The document is structured as follows. Firstly, in Section 2, a summary of the different trends in structured programming and video game programming is presented. Next, Section 3 details the context and architecture of the game engine in which the language has been tested. Section 4 describes the syntax and semantics of GameScript. Following that, Section 5 outlines some examples of game mechanics that can be developed with this language. In Section 6, a comparative study against scripting methods of other game engines is provided. Finally, Section 7 presents the conclusions of the work carried out and potential paths for future research.

2 Scripting and Structured Programming

In the context of video games, scripting has become an essential tool for managing specific events and behaviors [13], allowing for the modification of game logic without requiring recompilation [14]. While many engines originally employed domain-specific languages, there has been a shift towards the use of general-purpose scripting languages. As a result, many commercial games have been developed using languages such as Lua, Python, AngelScript, GameMonkey Script, Tcl, Ruby, or other object-oriented languages like Squirrel, JavaScript, or ActionScript [3, 15]. Furthermore, visual scripting languages [16] have been widely adopted to facilitate programming learning [17].

However, the development of new scripting languages is ongoing [18]. As Wilcox points out, "*you're not reinventing the wheel. You're creating a way to express your thoughts concisely in a new language*." For this reason, this work focuses on developing a basic scripting language with a generalist approach, drawing upon ideas from the early days of programming and avoiding the complexity that current scripting languages have accumulated. To achieve this, the work has explored structured programming and its fundamentals [19, 20], particularly focusing on the Structured Program Theorem.

The Structured Program Theorem, also known as the Böhm-Jacopini Theorem [7], lays the foundation for structured programming. It demonstrates that any computable function can be computed using three control structures: sequence, selection, and iteration, while avoiding GO-TO commands. Sequence entails the serial execution of various subprograms (THEN). Selection involves the execution of one subprogram instead of another based on the result of a boolean expression (through selection structures like IF-THEN-ELSE), while iteration entails the repeated execution of a subprogram while the evaluation of a boolean expression remains true (WHILE-DO). Additionally, despite this restriction, this structure allows for the use of auxiliary variables to track program information.

Subsequently, the Structured Program Theorem led to several related works, including an approach to the concept of Turing machines [21]. Additionally, various studies supplemented the Böhm-Jacopini Theorem. They asserted that by utilizing these auxiliary variables as boolean variables, any program with WHILE loops was equivalent to another program with a single WHILE-DO loop [22, 23]. Building upon this notion, the Folk Theorem asserts that every flowchart is equivalent to a While program with a single occurrence of the WHILE-DO instruction, achieved through the use of auxiliary variables [24]. Some authors implicitly demonstrated that the use of these auxiliary variables was strictly necessary [25–27]. Additional works, such as the Normal Form Theorem, demonstrate that any While program can be converted into a program with a single while loop [28].

Given that loop-based scripting languages [3] are the most general and widely used, the Folk Theorem ensures that no additional loops are required in the language besides the game loop itself. This is because a single loop guarantees the creation of any computable function, provided the necessary auxiliary variables are considered. However, the current trend in scripting languages mirrors that of classical programming languages, where the use of loops is common. Additionally, there is a trend towards using complex data structures, such as arrays, lists, or graphs, beyond simple variables.

Before addressing the description of the syntax and semantics of the proposed language, the architecture of the game engine upon which the scripting language in question has been built is presented below.

3 Game Engine

GameScript has been integrated into a 2D game engine based on MAS [10, 33]. MAS systems simplify game creation by focusing on agents and their interactions within a social environment [34]. Agents, in this setup, have variables to hold their state and scripts to define their actions and interactions with other agents and the environment (see Fig. 1). Because of these features, this architecture is seen as ideal for trying out the developed scripting language.

The analogy between defining a MAS and specifying a video game is undeniable [10, 11, 37, 38]: agents assume the role of game elements, often referred to as game objects. These agents interact with each other and convey essential information for game logic, detecting collisions or interactions with the user. Each agent in the game is assigned a specific task, guided by its own set of behavior rules defined through scripts. The behavior of each agent is autonomously determined based on its individual state and the environment state, and is evaluated sequentially [34].

The scripting language, like any other data management language, encompasses a set of operations known as CRUD (Create, Read, Update, Delete). This means that, in addition to enabling the creation and destruction of agents, the language also offers the capability to read and modify any properties of the agents and the social environment in response to player actions or events occurring in the system, such as collisions, state changes, or timers.

The variables storing agents and game states, managed from the scripting language, are numerical. Although the language primarily utilizes numerical, it can be easily extended to include other variable types, such as text strings. Additionally, it is possible to add new properties to expand both the initial state of the game and the agents of the system.

3.1 The game environment

The environment or social state of the game comprises properties that can be created, modified, or queried by any agent at any point during the game state, allowing for the addition of new knowledge as needed.The variables defining the game are grouped into the following categories:

- **Camera:** This category includes variables such as the position and rotation of the camera (cameraX, cameraY, cameraAngle).
- **Physics:** These variables define the forces interacting in the physical world, such as gravity force (gravityX, gravityY).
- **New:** Initially, new variables can be added that any agent can access, which are part of the social state of the game (for example, the overall game score).

Some variables, accessible only while the game is running, serve for interaction detection and game control:

- **Input**: These variables denote the events the game can manage, such as keyboard or mouse events, and signify whether the interaction has occurred or not (e.g., mouseDown, mouseUp, mouseOver, mouseClick, keyIDPressed, keyIDDown, keyIDUp, mouseX, mouseY).
- **Control**: Variables such as frames per second (FPS) at which the game runs or the delta time value (for example, FPS, deltaTime) or the management of collisions between objects.

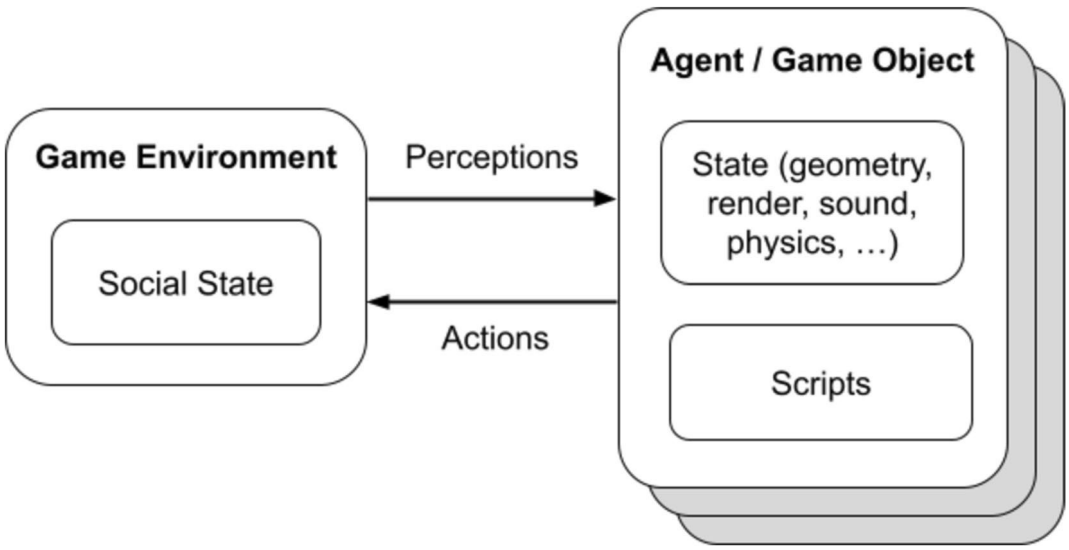

Fig. 1 Diagram of the agent system

Input or control variables are created in the engine's initialization module (see Fig. 2) after analyzing the scripts of each object and determining which variables are necessary, considering aspects of logic based on whether the game has keyboard and/or mouse events, and which keys are used for interaction or collision rules between object types.

3.2 The game object

The set of properties, shared by all game objects and defining the state of the engine's agents, are grouped into the following categories:

- **State**: A variable is established to activate and deactivate the agent (active), and another to define the object type in relation to collisions (tag).
- **Geometry**: It is represented through position variables (x, y), size (width, height), scale factor (scaleX, scaleY), rotation angle (angle), or collision shape (chosen between a box or a circle).
- **Rendering**: This category includes reference to the image representing the object, the opacity level, and the flipping state (image, opacity, flipX, flipY).
- **Sound**: Reference to the sound that the object can play (sound).
- **Physics**: Physical properties of the object, such as velocity (velocityX, velocityY) and angular velocity, as well as other material-related properties like friction, density, or restitution.
- **New**. Initially, new variables can be added that can also be accessed from any agent, to expand its internal state (e.g., character's life).

Some variables are only accessible during game execution and are grouped into collision, animation, and timer properties:

- **Collision.** These variables record when the collider of an agent detects a collision with the collider of another agent marked with a specific tag (collisionPlayer).
- **Animation**. In the case of animated images, the following variables are considered: the number of keyframes, the position of the current image in the sequence, and the speed at which the animation plays (numFrames, keyframe, FPS).
- **Timer**. Variables for counting the time elapsed since the game object was activated.

Collision, animation, and timer properties are created as needed in the engine's initialization module (see Fig. 2) after analyzing the scripts of each object.

3.3 The game loop

As a central component of any game from a programming perspective, the game loop [4] orchestrates the flow of the game's operations. The game engine accompanying the development of GameScript features three basic modules that sequentially assess the state of the game and its agents in each iteration: physics, logic, and rendering. However, as illustrated in Fig. 2, initialization takes place before the game begins. This process readies the system for evaluating the logic configured by the programmer in the agents' scripts, including the identification of necessary control variables for collision, animation, or timer events discussed in the preceding section. Initialization is responsible for creating and preparing these variables, and for converting the scripts into expressions (immutable at runtime) for subsequent evaluation by a mathematical library.

Upon completion of initialization, the game commences its execution, and the program enters a loop tasked with computing the physics, logic, and rendering of game objects (see Fig. 2) in sequential order. Focusing on the logic, when the game's operational flow reaches this juncture, the engine must interpret the behavioral specifications of the agents configured by the programmer through GameScript. Since the state of the game and all agents is stored in numerical variables, script execution evaluates them using a mathematical library that queries and modifies pertinent variables in each iteration of the game loop. Furthermore, to ensure proper data evaluation, the logic is evaluated multiple times within an internal loop, alongside the physics (see Fig. 2). This approach follows the scheme presented in Glenn Fiedler's article "*Fix your Timestep!*" [35], specifically the solution that includes a fixed time delta for the physical simulation along with a variable time delta for rendering.

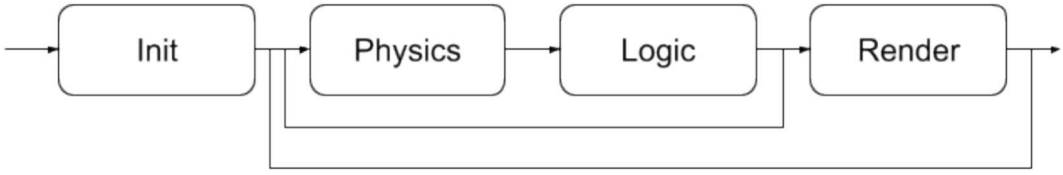

Fig. 2 The game loop

It is worth noting that there also exists a data input module that runs asynchronously. This module modifies event properties, such as keyboard interaction, which are subsequently queried from scripts in the logic module.

4 GameScript: scripting language

The proposed scripting language is based on the Structured Program Theorem [7, 24], which combines only three logical structures: sequence, selection, and iteration. In terms of iteration, the language does not require loops, as the game loop handles the repetition and continuous evaluation of game scripts. It is important to emphasize that eliminating loops within the language does not pose any issues in terms of its algorithmic capability. This strategy has already been utilized in other contexts, such as evolution algebras [29], or in abstract state machines invented by Yuri Gurevich, which constitute one of the most general models of computation today [30].

Therefore, GameScript will only include two logical structures: sequence and selection, as the game loop is part of the system.

4.1 Syntax

Drawing an analogy with the syntax of the While language [8] and following the Backus-Naur Form (BNF) notation scheme [31], the different syntactic categories and metavariables referring to the elements of each category are enumerated.

- n will vary in numbers, **Num**.
- x will vary in variables, **Var**.
- a will vary in arithmetic expressions, $\mathbf{A_{exp}}$.
- b will vary in boolean expressions, $\mathbf{B_{exp}}$.
- S will vary in statements, **Stm,** and
- g will vary in game objects, $\mathbf{G_{obj}}$.

Metavariables may include subscripts, such as n_1, n_2 representing numbers. Variables consist of strings of letters and numbers, beginning with a letter. Meanwhile, numbers are assumed to be structured as sequences of digits. Game objects are defined by a set of variables. From these elements, the abstract syntax is defined to generate variables, arithmetic expressions, boolean expressions, and language statements, yielding the following syntactic structure:

x ::= $g.x$ | $\#x$

a ::= n | x | $f(a)$ | $a_1 + a_2$ | $a_1 - a_2$ | $a_1 * a_2$ | a_1 / a | (a)

b ::= **true** | **false** | $a_1 > a_2$ | $a_1 >= a_2$ | $a_1 = a_2$ | $a_1 != a_2$

S ::= **create** g | **delete** | $x := a$ | **skip** | $S_1; S_2$ | **if** b **then** S_1 **else** S_2

To refer to variables from other game objects, a period is prefixed to the object's name (i.e., *Enemy.y*). If the variables do not belong to an object but to the game environment, they will be preceded by the character "#".

Arithmetic expressions "a" may consist of numeric literals "n," variables "x" as references to game objects or game variables, and a predefined set of mathematical functions "$f(a)$". These expressions are fundamental for performing calculations within the game environment. Specifically, they incorporate various mathematical functions such as square root, absolute value, random value generation, rounding, and trigonometric functions like sin, cos, tan, asin, acos, and atan.

Boolean expressions "b" are defined as having a basic element if their value is *true* or *false* (non-zero or zero) , or if they take one of the following relational forms: $a_1 > a_2$, $a_1 >= a_2$, $a_1 = a_2$, or $a_1 != a_2$, where a_1 and a_2 are arithmetic expressions. Note that relational operators are only included in one direction because they can be expressed by their opposite form.

Finally, language statements can be: **create** g which instantiates a new object, **delete** which destroys the object that executes the statement, assignment of an arithmetic expression to a variable "$x := a$", the **skip** statement, a composition of statements separated by semicolon, such as *"S1; S2"*, and the "**if** b **then** $S1$ **else** $S2$" statement.

This specification defines the abstract syntax of the language, outlining the construction of arithmetic expressions, boolean expressions, and statements. It closely resembles the language described by Nielson et al. [9], with a few modifications: the removal of the "while" statement, the extension of arithmetic expressions with functions, the removal of composite boolean expressions like $\neg b$ or $b_1 \wedge b_2$, and the addition of object creation and deletion statements, **create** and **delete**. Like in the specification of the While language [9], parentheses (...) are used here to resolve ambiguities, particularly in arithmetic expressions.

4.2 Semantics

The formal semantics of a language describes the meaning of the result of executing or evaluating a program. In the case of the proposed language, an operational approach is applied for defining the semantics [9]. In this context, the role of a language statement is to change the state of the system by modifying the value of some of its variables, adding the variables of a new object when it is created, or removing those of an object when it is destroyed. In particular, the natural operational semantics approach has been applied to describe how the program's execution results are achieved. According to this approach, the meaning of evaluating statements is specified using a transition system and can have two types of configurations:

<S, s> represents that statement S is executed from state s and s represents a terminal or final state

Therefore, the transition specifies the relationship between the initial state and the final state for each statement. This function, similar to the one described in the selected game engine for language integration, constitutes the state transformation function, introducing the effect of the actions of an agent on the environment [10]. The transition will be written as follows:

$$\langle S, s\rangle \rightarrow s'$$

This means that the execution of S from state s terminates and results in the final state s'. The definition of $\rightarrow$ is given by the rules in Fig. 3, where each rule has a number of premises (above the line) and a conclusion (below the line), as well as, when necessary, the condition that must be met if the rule is applied (to the right of the line). Rules with an empty set of premises are considered axioms, and the line has been omitted.

The axioms of the language, namely [*create*], [*delete*], and [*assign*] facilitate the addition, removal, and modification of game objects. These operations mirror the fundamental CRUD (Create, Read, Update, and Delete) operations found in persistent storage systems [32].

At this juncture, it should be noted that X represents the set of all variables in the system. The set is formed by the disjoint union of the variables of all game objects, in addition to the variables of the game environment.

$$X \backslash G_{Env} = U_{G \in P} G$$

where G_{Env} is the set representing the environment variables, and P is the family of sets representing the variables of game objects, constituting the partition of X. It is important to note that environment variables can also be read and modified.

Intuitively, the axiom [*create*] indicates that in state s, when **create** g is executed, a new state s' is obtained

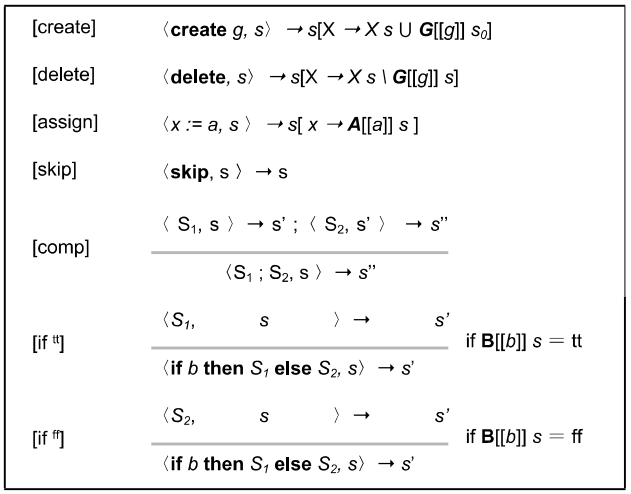

Fig. 3 Natural Semantics for the Scripting Language

$s[X \rightarrow X\, s \rightarrow G\left[\left[g\right]\right] s_0]$, which is equal to s except that the variables of the new game object g with their initial state values s_0 are added to the set of variables X. These variables are obtained using the semantic function G, which takes two arguments: the game object and the state. The functionality of G is as follows:

$$\mathrm{G} : \mathrm{G}_{\mathrm{obj}} \rightarrow (\mathrm{State} \rightarrow \mathrm{State})$$

This means that G takes a game object as a parameter and retrieves its initial state, which contains the variables of the game object g. It is also important to note that the position and rotation of the new game object are initialized with the values from the game object executing the **create** function.

The axiom [*delete*] similarly signifies that in state s, when **delete** is executed by a game object g, a new state $s[X \rightarrow X\, s \backslash G\left[\left[g\right]\right] s]$ is obtained. This state is equivalent to s, except that the variable set X no longer includes the variables of game object g, representing the result of applying the semantic function G.

Additionally, the axiom [*assign*] denotes that in a state s, when the assignment $x := a$ is executed, it result in a final state $s[\, x \rightarrow A[[a]]\, s\,]$, where the value of x is updated to $A[[a]]\, s$. This axiom serves to modify variables both of game objects and of the game environment. In this case, the semantic function A takes two arguments: the syntactic constructor and the state.

$$\mathrm{A} : \mathrm{A}_{\mathrm{exp}} \rightarrow (\mathrm{State} \rightarrow \mathrm{Z})$$

Similarly, the meaning of boolean expressions can be defined through the semantic function B.

$$\mathrm{B} : \mathrm{B}_{\mathrm{exp}} \rightarrow (\mathrm{State} \rightarrow \mathrm{T})$$

where T is defined by the values *tt* (for **true**) and *ff* (for **false**). For simplicity, in the language, variable values equal to zero are considered false, and non-zero values are considered true.

In order to illustrate some of the basic algorithms that can be executed using the proposed language, the derivation of certain language statements is detailed below.

Example 4.1. Statements required to move a game object in the direction specified by its *angle* property when the up key is held down. Let's assume the game object has a *speed* variable indicating its movement speed, and *#deltaTime* is the variable representing the elapsed time since the last frame (e.g., 0.02 seconds).

```
if #keyUpPressed then /* if up key is pressed */
    x := x + speed * sin(angle) * #deltaTime;
    y := y + speed * cos(angle) * #deltaTime
else skip;
```

Let s_0 be the initial state where the variables have the following values $[x \rightarrow 0, y \rightarrow 0, speed \rightarrow 5, angle \rightarrow \pi/4]$. When the key is not pressed, no state change will occur:

$$\frac{\langle skip, s_0\rangle \rightarrow s_0}{\langle if\#key\ Up\ \ Pressed\ then\ S\ else\ skip,\ s_0\rangle \rightarrow s_0}$$

In this case, the statement S is a compound statement composed of the **create** and **delete** statements. The following would be an instance of the rule:

$$\frac{\langle create\ g_1, s_0\rangle \rightarrow s_0[X \rightarrow X\, s_0 \rightarrow G\big[\big[g_1\big]\big]s_0]\ ;\langle delete\ g, s_1\rangle \rightarrow s_1[X \rightarrow X\, s_1 \backslash G\big[\big[g\big]\big]s_1]}{\langle create\ g_1\ delete\ g, s_0\rangle \rightarrow s_2}$$

However, in the event that the up key is pressed, the following instance of the [if ^{tt}] rule would occur:

$$\frac{\langle S, s_0\rangle \rightarrow s_2}{\langle if\#key\ Up\ \ Pressed\ then\ S\ else\ skip,\ s_0\rangle \rightarrow s_2}$$

where the statement S is a compound statement [comp], consisting of two assignments. The following would be an instance of the rule.

In this way, it can be observed that, upon executing the two statements within the **if** condition, a state transition occurs to s_1. Here, the system's variable set expands with the variables of game object g_1 in its initial state s_0. Subsequently, another state transition occurs, removing game object g and its variables from the system's variable set X. Where $s_1 = s_0[X \rightarrow Xs_0 \rightarrow G\big[\big[g_1\big]\big]s_0]$ and $s_2 = s_1[X \rightarrow Xs_1 \backslash G\big[\big[g\big]\big]s_1]$.It's worth noting that when

$$\frac{\langle x := x + speed * \sin(angle) * \#deltaTime, s_0\rangle \rightarrow s_0[\mathrm{x} \rightarrow 0.0707]\langle y := y + speed * \sin(angle) * \#deltaTime, s_1\rangle \rightarrow s_1[\mathrm{y} \rightarrow 0.0707]}{\langle x := x + speed * \sin(angle) * \#delta\,Time,\ s_0\rangle\langle y := y + speed * \cos(angle) * \#delta\,Time,\ s_1\rangle \rightarrow s_2}$$

Thus, it can be observed that after applying the two statements within the if statement, there is a change of state from s_0 to s_1, where a new value is assigned to x. Subsequently, there is another change of state from s_1 to s_2, where a new value is assigned to y. Specifically, $s_1 = s_0[x \rightarrow 0.0707]$ and $s_2 = s_1[y \rightarrow 0.0707]$.

Example 4.2. Sentences necessary to remove an object when it collides with another object and create a new object representing the explosion. The variable *bombCollision* pertains to the *Player* game object, taking the value false in the absence of collision and true upon collision.

if *bombCollision* **then** /* if it collides with the bomb */
    create *Explosion;*
    delete
else skip

While the collision does not occur [*bombCollision* →**false**], there is no change of state:

$$\frac{\langle skip, s_0\rangle \rightarrow s_0}{\langle if\ bomb\ Collision\ then\ S\ else\ skip,\ s_0\rangle \rightarrow s_0}$$

However, when a collision occurs and the *bombCollision* variable becomes **true**, the following instance of the rule [if ^{tt}] is triggered:

$$\frac{\langle S, s_0\rangle \rightarrow s_2}{\langle if\ bomb\ Collision\ then\ S\ else\ skip,\ s_0\rangle \rightarrow s_2}$$

creating a new object, the position and rotation variables are initialized with the values of the object executing the script where this action occurs.

5 Results

At this stage, to demonstrate the language's features and test its expressiveness, several games proposed by Becker et al. [12] have been implemented. Among them are Asteroids, Arkanoid, Frogger, Lunar Lander, or a platformer game (see Fig. 4). It should be noted that these games have been developed using mid-range to low-end computer models, and in no instance has there been a performance drop below 60 FPS. Subsequently, some of the most important mechanics of two of these games are described, along with the implementation of certain behaviors in a grid-based game such as Candy Crush. This approach aims to analyze the intricacies of the language's syntax and semantics.

5.1 Asteroids

Asteroids, released by Atari in 1976, is a shooting game in which a spaceship navigates through an asteroid belt. The objective is to destroy as many asteroids as possible while avoiding collisions with them. For its implementation, four main agents are identified: the Spaceship, the Laser, the Asteroid, and the Asteroid Generator, each with their own behaviors.

The first script to analyze describes the behavior of the Spaceship (see Script 1). Specifically, the script defines the rotation and forward movement of the Spaceship, its repositioning when exceeding screen boundaries, its shooting capability, and its destruction upon colliding with an Asteroid.

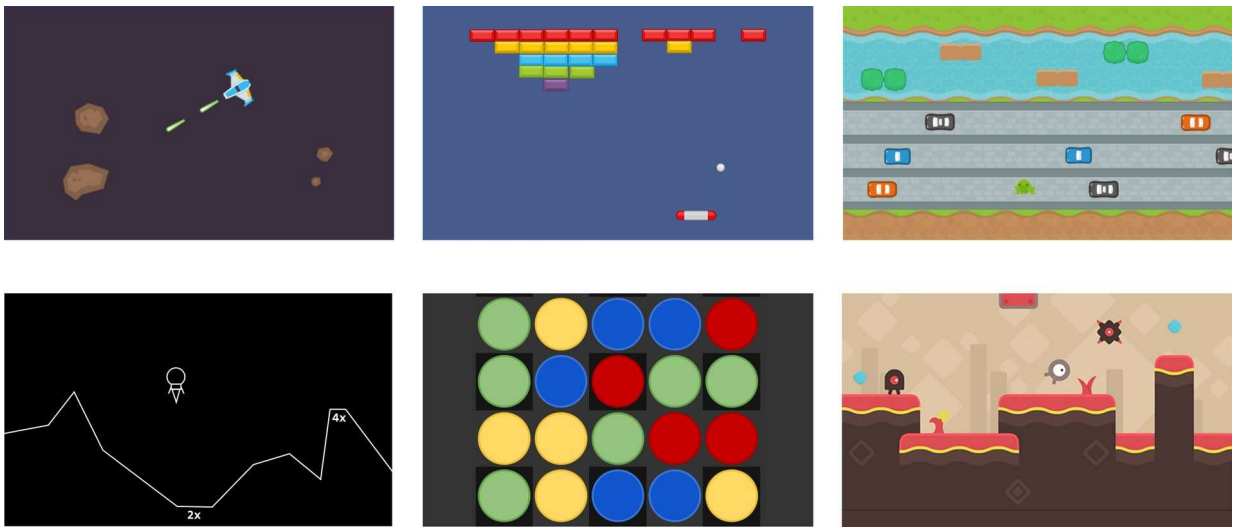

Fig. 4 Examples of some of the games implemented with the scripting language and the multi-agent game engine

Various syntactic elements are incorporated into the script. For instance, arithmetic expressions are utilized, as seen in line 2 with "*angle* + *speed* * *#deltaTime*". Boolean expressions are also present, as demonstrated in line 9 when the position of the Spaceship is checked using "*x* > (*#screenWidth* / 2)". Assignment statements are evident, such as in line 2 where the new angle value is calculated. Additionally, numerous **if** statements are featured, including nested selections, as observed in line 10. Moreover, **create** statements are used in lines 14 and 16 to initiate the Laser when the spacebar is pressed and to remove the Spaceship upon collision with the Asteroid.

Semantically, it's important to note that when a new agent is created, it inherits the transformation matrix from the agent that created it. This explains the assignments in lines 2 and 3 (refer to Script 2), where the Laser movement mirrors the direction of the Spaceship when it was created.

Also, it's crucial to emphasize the integration with the game engine, particularly concerning the input system, collision detection, and the physics engine. The input system dynamically updates game variables responsible for detecting user input. For instance, when the left key is held down, the variable *#keyLeftPressed* is updated to enable scripts to access it. Similarly, the Spaceship agent possesses a variable, *collisionSpaceshipAsteroid*, which toggles to true upon collision with objects of the Asteroid type (designated as 'asteroid'), and false otherwise. Additionally, two variables linked to the Spaceship enable the physics engine to exert force when the up key is pressed (refer to lines 4 to 6), with the magnitude and direction specified by the variables *ImpulseMagnitude* and *ImpulseAngle*.

Script 1. Spaceship Behavior

```
1   /* Spaceship Movement */
2   if #keyLeftPressed then angle := angle + speed * #deltaTime else skip
3   if #keyRightPressed then angle := angle - speed * #deltaTime else skip
4   if #keyUpPressed then /* physic movement */
5           ImpulseMagnitude := 0.1;
6           ImpulseAngle := angle
7   else skip
8   /* Repositioning Screen Limits */
9   if x > (#screenWidth / 2) then x := -#screenWidth / 2
10  else if (#screenWidth / 2) > x then x := #screenWidth / 2 else skip
11  if y > (#screenHeight / 2) then y := -#screenHeight / 2
12  else if (#screenHeight / 2) > y then y := #screenHeight / 2 else skip
13  /* Shoot */
14  if #keySpaceDown then create Laser else skip
15  /* Spaceship Destruction */
16  if collisionSpaceshipAsteroid then delete else skip
```

The following script (see Script 2) defines the behavior of the Laser. This game object appears when the space key is pressed and is always in motion until it collides with an Asteroid, at which point it is destroyed.

Script 2. Laser Behavior

```
1   /* Laser Movement */
2   x := x + speed * cos(angle) * #deltaTime;
3   y := y + speed * sin(angle) * #deltaTime;
4   /* Laser Destruction */
5   if collisionLaserAsteroid then delete else skip
```

The following script (see Script 4) illustrates the behavior of the Asteroid Generator agent. This agent employs a timer to create an Asteroid every 5 seconds. Objects employing timers in their rules have associated variables (in this case, *asteroidTimer*) that accumulate the elapsed time since the agent's creation. These variables are incremented by the value of *#deltaTime* in each cycle of the logic engine. The initialization or resetting of the timer is managed by the engine when the timer condition is met.

The behavior of the Asteroids (see Script 3) is based on calculating the initial position and the impulse applied for their physical movement. At the syntactic level, the utilization of the random() function is evident, as it is invoked in the assignment of the position, initial rotation, and direction of force (see lines 4-8). Additionally, instructions for repositioning the Asteroid when screen boundaries are exceeded (these lines are omitted in the script, as they are similar to those for repositioning the Spaceship) and its destruction (see line 12) upon colliding with the Laser (with the laser tag) are also provided. The destruction of the Asteroid results in the creation of two smaller Asteroids that behave similarly but no longer generate new Asteroids when destroyed.

Script 3. Asteroid Behavior

```
1   /* Asteroid Movement */
2   if firstTime then /* initial position and physics impulse */
3           firstTime := 0;
4           x := random(-#screenWidth /2, #screenWidth / 2);
5           y := #screenHeight / 2 + height;
6           angle := random (1, 360);
7           ImpulseMagnitude := 2;
8           ImpulseAngle := random(1,360);
9   else skip
10  /* Repositioning Screen Limits, same code as for the Spaceship (@ Script 1)*/
11  /* Asteroid Destroy*/
12  if collisionAsteroidLaser then
13          create HalfAsteroid; /* small Asteroid agent with the same behavior */
14          create HalfAsteroid;
15          delete
16  else skip
```

Script 4. Asteroid Spawner Behaviour

```
1    if asteroidTimer = 5 then
2            create Asteroid
3    else skip
```

5.2 Platformer

A platformer game can include various types of agents such as the main character controlled by the player, platforms, surfaces, enemies, etc. This example focuses on mechanics related to the main character's movements and the accompanying camera tracking. These movements are physical, altering the character's velocity. The character's animations consist of running, standing still, jumping, and falling. Additionally, sound effects are implemented in the engine by modifying variables.

The first script (see Script 5) governs the movements of the Player agent (the main character), changes its animations, and detects collisions with enemies. Movements are executed by adjusting the character's velocity in the x and y coordinates. Note that jumps are only allowed when the character is in contact with the ground (see line 4). Different image sequences can be included as variables of the agent for animations. When a script updates the image (see line 10), it assigns the reference to the current image of the specified sequence. The rendering engine is responsible for changing the image of the animation sequence in each frame according to its playback speed.

Regarding sound playback, each of the agents has a sound variable that serves as a reference to the sound to be played. When the logic engine detects this variable, it initiates sound playback asynchronously and resets the variable value.

Script 5. Player Behaviour

```
1    /* Player Movement */
2    if #keyRightPressed then velocityX := speed else skip
3    if #keyLeftPressed then velocityX := -speed else skip
4    if collisionPlayerGround then
5            if #keyUpPressed then velocityY := speed else skip
6    else skip
7    /* Player Animations */
8    image := runAnimation
9    if velocityX > 0 then flipX := 0 else flipX := 1
10   if velocityY > 0 then image := jumpAnimation else image := idleAnimation
11   if 0 > velocityY then image := fallingAnimation else skip
12   /* Collision with enemies */
13   if collisionPlayerEnemy then
14           lives := lives -1;
15           sound := soundHurt
16   else skip
```

Finally, the behavior of the Camera agent, responsible for controlling the game camera that follows the player, is described (see Script 6). The script manages the camera to replicate the player's movements as long as it is within the boundaries defined by *#screenWidth* and *#screenHeight*.

Script 6. Camera Behaviour

```
1   /* Camera Movement */
2   if Player.x > (- #screenWidth / 2) then
3       if (#screenWidth / 2) > Player.x then #cameraX := Player.x else skip
4   else skip
5   if Player.y > (- #screenHeight / 2) then
6       if (#screenHeight / 2) > Player.y then #cameraY := Player.y else skip
7   else skip
```

5.3 Candy crush

Candy Crush Saga is a widely popular tile-matching game developed by King. In this game, colored candy pieces are swapped on a board by players to create lines of three or more candies of the same color, subsequently removing them from the board and replacing them with new ones. While a matrix structure is typically used to manage board information in such games, our language lacks such structures. Therefore, an alternative approach is required to implement similar mechanics.

For simplicity, the focus here is on describing the detection and removal of three consecutive red candy pieces along a single horizontal line. Three candy colors are assumed: red, green, and blue. Extending this behavior to the full game involves replicating it across rows, columns, or newly generated pieces. The agents involved in this mechanic are the Candy, the Counter, and the Destroyer.

The script (see Script 7) begins by spawning the Counter agent at the start of its row after each player moves. Once spawned, the Counter traverses its row horizontally, moving from Candy to Candy using the step variable. At each position, consecutive red candy pieces are checked for through collision detection, and the count is stored in an auxiliary variable, *redCandy*. Upon detecting three consecutive red candies, a Destroyer agent is spawned. If the count falls short, the Counter self-destructs upon reaching the screen's end.

Script 7. Detection of three consecutive pieces.

```
1   x := x + step
2   if collisionRedCandy then redCandy := redCandy + 1; else skip
3   if collisionGreenCandy then redCandy := 0 else skip
4   if collisionBlueCandy then redCandy := 0 else skip
5   if redCandy = 3 then
6       create Destroyer
7   else skip
8   if x >= #screenWidth then
9       delete
10  else skip
```

If a Destroyer agent has been spawned, it triggers a behavior outlined in scripts 8 and 9. The first script governs the movement of the Destroyer agent, which moves three positions in the opposite direction to the Counter agent, traversing the three consecutive pieces. After completing these movements, the agent self-destructs.

Script 8. Removal of three consecutive pieces.

```
1   x := x - step
2   if collisionRedCandy then
3           redCandy := redCandy + 1
4   else skip
5   if redCandy = 3 then
6           delete
7   else skip
```

Simultaneously, considering that an agent can only destroy itself, script 9 governs the reaction of the red candy agents to the collision with the Destroyer agent, resulting in their removal from the game environment.

Script 9. Removal of red candies.

```
1   if collisionDestroyer then
2           delete
3   else skip
```

6 Comparative analysis

6.1 Performance comparison with traditional scripting

As mentioned in Sect. 3.3, the evaluation of logic within a game engine is an integral step in the game loop execution. This process involves assessing the game's state in various aspects such as physics, collision detection, user interaction, and rendering, among others. In terms of operation, a standard game loop typically has a refresh rate of 17 milliseconds (60 FPS), and the execution of scripts controlling game element behaviors occupies a percentage of this time.

Evidently, the paradigm behind GameScript may present different occupation times compared to a traditional scripting language that uses data structures such as arrays or matrices, or control structures like loops. To compare these differences in occupation time, a comparative study was conducted by implementing two mechanics from the Candy Crush game presented in Sect. 5.3: the random initialization of candies to one of the available colors and the traversal of rows to check for three or more consecutive candies of the same color following the logic of Script 7.

Table 1 Comparison in milliseconds of the execution time of two Candy Crush mechanics using GameScript and traditional data structures for N scripts

	Initialization		Consecutive candies	
Scripts	Traditional	GameScript	Traditional	GameScript
10	0.009 ms	0.006 ms	0.004 ms	0.015 ms
100	0.047 ms	0.051 ms	0.013 ms	0.043 ms
1000	0.453 ms	0.500 ms	0.108 ms	0.350 ms
10000	4.999 ms	5.335 ms	2.141 ms	2.941 ms

Regarding the implementation differences, for the random initialization algorithm, the main distinction is that a traditional implementation would traverse the list of elements in the first iteration of the game, whereas GameScript requires a script per agent that includes an auxiliary variable to check if it is the first iteration. For checking consecutive candies, in the traditional version, it would not be necessary to implement the same algorithm as in Script 7; it would suffice to traverse the matrix row with a loop and check if three consecutive candies are found.

In this context, and to obtain comparable data for the comparison, Python was selected as the general programming environment on an Ubuntu 22.04.4 LTS system with an

Table 2 Comparison of game engine scripting languages in aspects such as control structures, game loop, and components.

	GameScript	Gdevelop	Construct 3	Game Maker	Stencyl
Selection	If then else	If then	If then	If then else	If, otherwise, otherwise if,
Loops	No specific sentences	Repeat, while, foreach object, foreach child	For, foreach, repeat, while	Repeat, while, for, do until	Repeat n times, repeat until, while
Game loop	No specific functions	No specific functions	Every tick, on suspended, on resume, on start of layout	No specific functions	When creating, when drawing, always
Main entity and components	Agent and five components including geometry, physics, audio, render, and behavior rules	Object and purposes for display, ui or visual effects.	Objects and add ones such as plugins, effects and behaviors	Objects and properties to handle visuals, physics, hierarchy and events	Actor and components related to appearance, behaviors and physics

AMD Ryzen 7 7840HS processor and 32GB of RAM, both for the GameScript version (through an implementation of the logic evaluation module in the game engine) and for the traditional implementation.

For the study, executions of both implementations were performed with sets of scripts, and the execution times required for each case were measured. Table 1 presents the results obtained for various orders of magnitude of rules in the scene. It should be noted that, in the case of initialization, both implementations include a call to the same random number generation method, which has been observed to slow down execution, although this equally affects both cases.

As shown in Table 1, the recorded times are generally lower in the traditional implementation compared to the GameScript implementation. Only from the fourth order of magnitude of evaluated rules in the scene do times start to approach a third of the available time at 60 FPS. However, these volumes are extremely high for the type of games developed with this system, which usually have around 100 rules running in each game loop. In this sense, it can be stated that although the performance of GameScript is lower than that of a traditional implementation, the occupation percentage of the total time for 100 rules, and even 1000 rules, is sufficient for use in video game development.

6.2 Comparison with other 2D game engines

At the semantic level, the features of the chosen programming language play a crucial role in the development of any project. At this point, a comparison will be made between the system proposed in this paper and the programming languages of four of the most prominent engines in the development of 2D video games: GDevelop, Construct 3, Game Maker and Stencyl. This comparison is framed within the context of the multi-agent engine, highlighting key variations across various fundamental dimensions.

The comparison process among 2D game development environments presented in Table 2 reveals significant differences regarding the available control structures. GameScript, for instance, provides a simple option with its "**if then else**" structure, whereas GDevelop offers a wide range of structures including conditions and loops. Conversely, Construct 3 and Game Maker present more conventional options with "**if else**" and "**repeat**", "**while**" respectively, while Stencyl adds complexity with its variety of structures, including "**otherwise if**" and additional logical operators.

Loops, another crucial feature, also showcase a variety of approaches among the environments. While GameScript lacks this functionality, GDevelop, Construct 3, Game Maker, and Stencyl offer loops such as "**repeat**", "**while**", "**foreach**", and more, enabling developers to control the repetition of actions or events in their games.

Regarding the game loop and associated functions, each environment offers different options. While GameScript and GDevelop do not specify specific functions for the game loop, Construct 3 includes events for various situations such as "at layout start" or "on each tick" of the game clock. Game Maker provides events for object creation and drawing, as well as functions that run continuously, while Stencyl offers events for specific situations like creation and drawing, ensuring detailed control over the game's behavior in different states.

6.3 Limitations

In terms of analyzing capabilities and limitations in video game development, the primary aspect that stands out compared to a conventional implementation is its inferior performance due to the alternative approach in algorithm implementation. GameScript employs a

distributed organization of agents that execute their scripts in each iteration of the game loop. Although this methodology simplifies behavior specification, a traditional implementation is typically more efficient. According to the performance analysis in Sect. 6.1, despite using a single loop and auxiliary variables instead of more efficient data structures like arrays, current hardware allows for the creation of 2D arcade games without significant issues.

In general, it is important to note that scripts in GameScript tend to be more extensive compared to conventional programming languages. This is because, in cases where logical operators or data structures like lists are used, the simplicity of GameScript's operators and structure increases the amount of code required to implement behaviors. For example, in the implementation of "Consecutive Candies" in Sect. 6.1, Python allows for the direct evaluation of three consecutive positions in a list, while in GameScript, auxiliary variables are needed to count and manage consecutive colors. However, it is important to remember that the philosophy behind the development of GameScript is not focused on achieving maximum performance, but on simplifying behavior definition and offering alternatives for solving mechanics, promoting new approaches to computational thinking.

Another aspect impacting performance is memory organization. Compared to data structures like arrays or matrices, which are designed to optimize memory usage, GameScript's approach does not guarantee this, resulting in lower performance by distributing elements across the game space. Additionally, efficiency is influenced by the compiler or interpreter used. In the case of the game engine, the language is developed in JavaScript, utilizing a variant of *math.js* for script evaluation, which aligns with the characteristics of the engine in which it has been implemented. Furthermore, the interpreted nature of the language affects its performance relative to compiled languages; however, this design allows for implementation in any conventional programming language, such as JavaScript, C++, C#, or Python, as it employs a restricted set of statements and data structures. Finally, in terms of scalability, it has been observed that the complexity of games developed with GameScript can range from a few scripts to more than 1000 without significantly affecting performance, as long as the complexity of the scripts is considered.

These limitations should be taken into account when using GameScript, as although it offers advantages in terms of flexibility and ease of implementation, its performance and scalability may not be suitable for developments requiring high performance and efficient memory management. Nonetheless, for arcade games, GameScript is sufficient to provide complete development and a satisfactory gaming experience.

7 Conclusions and future work

This document proposes a scripting language that allows the definition of game logic with a simple and compact syntax. The language comprises only two logical structures: sequence and selection, as the game loop is an integral part of the system. The syntax enables the specification of object creation and deletion, as well as the modification of their variables. Furthermore, language conditions are defined using relational boolean expressions, eliminating the need for compound boolean operations involving logical operators. Additionally, arithmetic expressions play a fundamental role in the language and have been expanded to facilitate the use of mathematical functions. Thus, a language is introduced in which all variables are of numeric type.

The language has been successfully tested on a MAS-based game engine. The engine has been adapted so that different modules operate on system variables. Thus, the initialization module creates and prepares all necessary variables for use in the scripts, the input system modifies the game variables, the physics engine acts on the variables to detect collisions and apply forces to game objects, the logic engine updates the variables defining the timers of its rules, and finally, the rendering engine modifies the variables related to sprite animation.

As outlined in the document, with the specification of the proposed language, a large number of arcade games have been implemented. It is worth noting that eliminating control structures such as loops requires designing games with MAS in mind, as demonstrated by the removal of pieces in the Candy Crush game.

As part of future work and projects, the capabilities of the multi-agent game engine architecture and the scripting language will continue to be tested to evaluate their potential. Additionally, potential performance optimization options will be explored, with a focus on improving memory management and overall efficiency, for instance, by studying the possibility of integration into parallel or distributed architectures. Regarding the language, it is intended to be used to solve classic programming problems, such as sorting lists or arrays, and even for programming pathfinding algorithms. Furthermore, the utility of the language for 3D game creation will be explored, either through the expansion of the multi-agent engine to its 3D version or its integration into commercial game engines. Efforts will also be made to enhance certain features, such as adding support for peripheral controllers like console gamepads. Regarding the engine, aside from its planned expansion into 3D, analysis and profiling functions are being considered to verify its performance in complex gaming contexts. Currently, work is underway on integrating the language into the Unity ecosystem for

3D game development, based on C# and leveraging Unity's capabilities and functionalities, which could address some of the current limitations and expand the applications of GameScript.

Acknowledgments This work has been developed within the framework of research project PID2023-149976OB-C21, funded by "MCIN/AEI/10.13039/501100011033/FEDER, UE", and grant CIAICO/2021/037 from the Generalitat Valenciana. It has also received support from ValgrAI, the Graduate School and Research Network in Artificial Intelligence, and the Generalitat Valenciana, with co-funding from the European Union. In addition, this work has benefited from the graphic resources of Kenney.nl. We would like to acknowledge Professor Jose Martinez-Sotoca. He was part of the initial conception of the idea and development of the model. Sadly, he passed away in 2022. However, we would like to acknowledge him for being part of this research.

Author Contribution The authors C.ML and M.C. collaborate jointly, building upon prior work conducted by C.ML under the guidance of M.C. Both authors played an integral role in all stages of the process, including the conception of the idea, analysis of the state of the art, development of the model and test sets. Additionally, responsibilities for drafting the article and reviewing the manuscript were shared between them, reflecting an equitable collaboration in the execution of this study.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Toftedahl, M., & Engström, H. (2019). A taxonomy of game engines and the tools that drive the industry. In DiGRA 2019, The 12th Digital Games Research Association Conference, Kyoto, Japan, August 6-10, 2019. Digital Games Research Association (DiGRA).
2. Gregory, J. (2018). Game engine architecture. crc Press.
3. Anderson, E. F. (2011, May). A classification of scripting systems for entertainment and serious computer games. In 2011 Third International Conference on Games and Virtual Worlds for Serious Applications (pp. 47-54). IEEE.
4. Nystrom, R. (2014). Game programming patterns. Genever Benning.
5. Kernighan, B.W., Wyk, C.J.V.: Timing trials, or the trials of timing: experiments with scripting and user-interface languages. Software. Pract. Exp. **28**(8), 819–843 (1998)
6. Phelps, A.M., Parks, D.M.: Fun and Games: Multi-Language Development: Game development can teach us much about the common practice of combining multiple languages in a single project. Queue **1**(10), 46–56 (2004)
7. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. Commun. ACM **9**(5), 366–371 (1966)
8. Nielson, H.R., Nielson, F.: Semantics with applications, vol. 104. Wiley, Chichester (1992)
9. Nielson, F., Nielson, H. R., & Hankin, C. (2004). Principles of program analysis. Springer Science & Business Media.
10. Marín-Lora, C., Chover, M., Sotoca, J.M., García, L.A.: A game engine to make games as multi-agent systems. Adv. Eng. Softw. **140**, 102732 (2020)
11. Marín-Lora, C., Cercós, A., Chover, M., & Sotoca, J. M. (2020, April). A First Step to Specify Arcade Games as Multi-agent Systems. In the World Conference on Information Systems and Technologies (pp. 369-379). Springer, Cham.
12. Becker, K., & Parker, J. R. (2005). All I ever needed to know about programming, I learned from re-writing classic arcade games. In Future play, the international conference on the future of game design and technology.
13. Sweetser, P., Wiles, J.: Scripting versus emergence: issues for game developers and players in game environment design. Int. J. Intell. Games. Simulat. **4**(1), 1–9 (2005)
14. Varanese, A. (2002). Game Scripting Mastery (Premier Press Game Development (Paperback)). Course Technology Press.
15. Garcés, D.: Scripting language survey. Game. Programm. Gems. **6**(2006), 323–340 (2006)
16. Sewell, B. (2015). Blueprints Visual Scripting for Unreal Engine. Packt Publishing Ltd.
17. Rebollo, C., Marín-Lora, C., Remolar, I., & Chover, M. (2018). Gamesonomy vs Scratch: Two Different Ways to Introduce Programming. International Association for Development of the Information Society.
18. Wilcox, B. (2007, April 12). Reflections on building three scripting languages. Retrieved January 31, 2024, from https://www.gamedeveloper.com/programming/reflections-on-building-three-scripting-languages
19. Dijkstra, E. W. (1970). Notes on structured programming.
20. Mills, H. D. (1972). Mathematical foundations for structured programming.
21. Prather, R.E.: Structured turing machines. Inf. Control. **35**(2), 159–171 (1977)
22. Mirkowska, G. (1972). Algorithmic logic and its applications. Doctoral diss., Univ. of Warsaw.
23. Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. (Eds.). (1972). Structured programming. Academic Press Ltd.
24. Harel, D.: On folk theorems. Commun. ACM **23**(7), 379–389 (1980)
25. Ashcroft, E., & Manna, Z. (1979). The translation of "go to"programs to "while" programs. In Classics in software engineering (pp. 49-61).
26. Kosaraju, S.R.: Analysis of structured programs. J. Comput. Syst. Sci. **9**(3), 232–255 (1974)
27. Kozen, D., & Tseng, W. L. D. (2008, July). The Böhm–Jacopini theorem is false, propositionally. In the International Conference

on Mathematics of Program Construction (pp. 177-192). Springer, Berlin, Heidelberg.
28. Solin, K.: Normal forms in total correctness for while programs and action systems. J. Logic Algebraic Program. **80**(6), 362–375 (2011)
29. Gurevich, Y., & Börger, E. (1995). Evolving algebras 1993: Lipari guide. Evolving Algebras, 40.
30. Dershowitz, N. (2012). The generic model of computation. arXiv preprint arXiv:1208.2585.
31. Knuth, D.E.: Backus normal form vs backus naur form. Communicat. ACM. **7**(12), 735–736 (1964)
32. Daissaoui, A. (2010, May). Applying the MDA approach for the automatic generation of an MVC2 web application. In 2010 Fourth International Conference on Research Challenges in Information Science (RCIS) (pp. 681-688). IEEE.
33. Chover, M., Marín-Lora, C., Rebollo, C., & Remolar, I. (2020). A game engine designed to simplify 2D video game development. Multimedia Tools and Applications, 1-22.
34. Wooldridge, M. (2009). An introduction to multiagent systems. John Wiley & sons.
35. Fiedler, G. (2004, June 10). Fix your timestep! Retrieved January 31, 2024, from https://gafferongames.com/post/fix_your_timestep
36. Wilensky, U. (2001). NetLogo Tetris model. http://ccl.northwestern.edu/netlogo/models/Tetris. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL
37. Marín-Lora, C., & Chover, M. (2023, September). A Multi-agent Sudoku Through the Wave Function Collapse. In the European Conference on Multi-Agent Systems (pp. 381-395). Cham: Springer Nature Switzerland.
38. Marín-Lora, C., Chover, M., & Sotoca, J. M. (2022). A Multi-agent Specification for the Tetris Game. In Distributed Computing and Artificial Intelligence, Volume 1: 18th International Conference 18 (pp. 169-178). Springer International Publishing.