# Un lenguaje para la especificación de la lógica de juegos

Carlos Marín-Lora[1], Miguel Chover[1], and Jose M. Sotoca[1]

Instituto de Nuevas Tecnologías de la Imagen (INIT), Universitat Jaume I, Castellón de la Plana, Spain
cmarin@uji.es, chover@uji.es, sotoca@uji.es

**Abstract.** The development of video games is a complex task due to the large amount of knowledge it requires in terms of design and programming. In particular, the specification of the behavior of the elements of a game is one of the factors that most increase its complexity. In this sense, there are a variety of game engines with numerous features and programming languages, including complex visual programming systems. With the intention of raising an alternative and reduce the complexity of current systems, presents a small programming language capable of defining any algorithm to specify the behaviors of the elements that make up the games. The dentition of the language has been based on the theorem of structured programming and its derived studies, taking as reference the While language. The ultimate goal of this work is to validate the expressiveness of language and demonstrate theoretically that allows to create the mechanics of any arcade game. For its validation, have been implemented multiple games with different mechanics, having successfully completed all the games that have been proposed so far.

**Keywords:** Programming Language · Game Development · Game Logic · Game Engine.

## 1 Introducción

El desarrollo de videojuegos implica el conocimiento de un lenguaje de programación. Actualmente existen múltiples entornos con los que desarrollar juegos: desde aplicaciones comerciales tan potentes como Unity o Unreal, pasando por proyectos colaborativos de código libre como Godot, hasta librerías de código como PlayCanvas, Threejs o Babylon [10, 13]. Estos entornos de desarrollo basan la construcción de los objetos de sus juegos y la definición de sus lógicas en lenguajes de programación de alto nivel como C++, C o JavaScript, y en librerías especializadas. Este entramado de software compone lo que se conoce habitualmente como motor de juegos.

Una característica que comparten todos estos motores es la tendencia a integrar nuevas técnicas y procedimientos en sus entornos de desarrollo haciendo

realmente complicado utilizar todo el potencial que proporcionan sin ser un programador experto. A esto se le añade la existencia de múltiples patrones de diseño y programación de juegos [23], y la inherente evolución de los lenguajes de programación de alto nivel, los cuales cuentan cada vez con más características y funcionalidades.

Con la intención de establecer una referencia para los lenguajes de especificación de la lógica de los videojuegos, este trabajo presenta un pequeño lenguaje de programación capaz de definir algoritmos que describan mecánicas clásicas en el desarrollo de videojuegos. Para ello se parte del teorema de programación estructurada [4] y se define un lenguaje que utiliza sólo la secuencia, la selección y la iteración. El lenguaje está basado en While, un pequeño lenguaje de programación utilizado para el análisis teórico de la semántica de lenguajes de programación imperativos [22].

El objetivo del trabajo es demostrar teóricamente que el lenguaje propuesto permite crear cualquier tipo de algoritmo asociado a los elementos de un videojuego, así cómo validar su expresividad en la práctica mediante la implementación de diversas mecánicas clásicas de juegos arcade.

A modo de resumen, este documento se organiza de la siguiente manera. En primer lugar, en la sección 2, se presenta el estudio realizado sobre el estado del arte en programación estructurada y programación de juegos. A continuación, en la sección 3, se detalla el contexto en el que ha sido definido el lenguaje. En la sección 4, se desarrolla su sintaxis y su semántica. Posteriormente, en la sección 5, se plantea la descripción de una serie de mecánicas de juego con el lenguaje a modo de caso de uso y de resultados del trabajo realizado. Finalmente, en la sección 6, se presentan las conclusiones al estudio y las posibles vías de trabajo en el futuro.

## 2 Estado del arte

Este trabajo trata de alcanzar un sistema mínimo para la definición de la lógica en videojuegos. Para ello, la investigación se centra en el estudio de la literatura sobre teoría de lenguajes de programación y la programación estructurada [9, 19], y particularmente sobre el teorema del programa estructurado.

El teorema del programa estructurado, también llamado teorema de Böhm-Jacopini [4], establece la base de la programación estructurada evitando los comandos GO-TO y determinando que es posible calcular cualquier función computable mediante tres estructuras de control: la secuencia, la selección y la iteración. Entendiendo secuencia como la ejecución sucesiva de varios subprogramas (THEN), selección como la ejecución de un subprograma sobre otro en función del resultado de una expresión booleana (con estructuras de selección como el IF-THEN-ELSE), e iteración como la ejecución repetida de un subprograma siempre que la evaluación de una expresión booleana sea verdadera (WHILE-DO). Además, y a pesar de esta restricción, esta estructura contempla el uso de variables auxiliares para realizar un seguimiento de la información del programa.

En los años sucesivos, la programación estructurada ha dado lugar a multitud de trabajos de campo, incluyendo una aproximación del concepto a las máquinas de Turing [24]. Además, múltiples estudios complementaron el teorema de Böhm-Jacopini afirmando que mediante el uso de esas variables auxiliares a modo de variables booleanas, todo programa con bucles WHILE era equivalente a otro programa con un único bucle WHILE-DO [20, 6]. A partir de esto, se establece el denominado Teorema Folk, que indica que todo diagrama de flujo es equivalente a un programa While con una única ocurrencia de WHILE-DO, mediante el uso de variables auxiliares [12]. De hecho, algunos autores demostraron implícitamente que el uso de estas variables auxiliares era estrictamente necesario [2, 16, 17].

El contexto de este trabajo se centra sobre la aplicación de estos conocimientos sobre el desarrollo de juegos, y concretamente sobre el scripting. El scripting en videojuegos está considerado como un método de gestión para eventos y comportamientos específicos [27] que permite modificar la lógica del programa sin la necesidad de volver a compilar el juego completo [28].

Los lenguajes de scripting que suelen utilizarse para la definición de la lógica proporcionan una capa de abstracción sobre los sistemas que gestionan los juegos en los dispositivos, normalmente controlados por lenguajes como C o C++. Es decir, están orientados a facilitar la programación, a menudo a costa de afectar al rendimiento del juego en tiempo de ejecución [14].

Durante la última década, se ha observado una tendencia en los motores de juegos hacia el uso de lenguajes de scripting genéricos, desplazando a los lenguajes propios de los sistemas de desarrollo de juegos. Actualmente, los lenguajes más utilizados son C#, Python y JavaScript [1] con mención aparte a los sistemas de scripting visual como Scratch o Unreal Blueprints [25].

A continuación, se propone un lenguaje como punto medio entre la simbología abstracta de nivel formal y los lenguajes de scripting para juegos de alto nivel.

## 3 Contexto

El lenguaje de programación que se va a definir tiene propósito general. Sin embargo, ha sido descrito sobre un motor de juegos con un modelo de datos basado en sistemas multiagente [18, 5]. Estos sistemas se caracterizan porque sus agentes se encuentran en un espacio compartido (llamado entorno) con otros agentes, con los que comparten un conjunto de propiedades con características predefinidas: geométricas, render, físicas, audio, etc. Además, los agentes comparten unas propiedades generales del juego o entorno social (ver Figura 1). El comportamiento de cada agente se establece de forma autónoma en función de su propio estado y el de su entorno en el instante que se evalúa su lógica.

Para definir los comportamientos de los objetos del juego (agentes) tras sus interacciones con su entorno, se ha creado un lenguaje capaz de describir cualquier algoritmo, partiendo del teorema del programa estructurado [4, 12]. Este teorema establece que toda función computable puede ser implementada en un lenguaje de programación combinando sólo tres estructuras lógicas: la se-

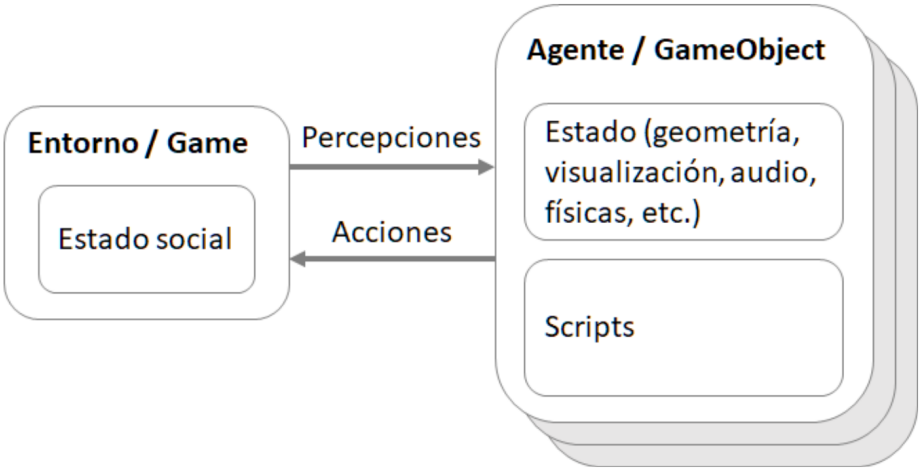


**Fig. 1.** Esquema del sistema de agentes.

cuencia, la selección y la iteración. Por lo tanto, estas serían las estructuras que deberían definirse, de forma similar a como se han definido para While [21].

Por otro lado, el teorema del programa estructurado tiene una versión que se adapta muy bien al patrón de diseño de juegos del bucle de juego, conocido como Game Loop [23]. Esta nueva versión realiza una demostración similar a la anterior pero utilizando únicamente un bucle while [12]. El nuevo teorema conocido como Folk Theorem especifica que "Cualquier diagrama de flujo es equivalente a un programa While con una ocurrencia del bucle while, incluyendo si es necesario variables adicionales". También hay resultados similares como el teorema de la forma normal que demuestra que cualquier programa While se puede convertir en un programa con un único bucle while [26].

De esta forma, el lenguaje que se presenta sólo va a incluir dos estructuras lógicas: la secuencia y la selección, ya que el bucle de juego forma parte del sistema. La eliminación de los bucles dentro del lenguaje no supone ningún problema en cuanto a la capacidad algorítmica de lenguaje y ya ha sido utilizada en otros contexto como en las álgebras de evolución donde se utiliza como constructores de sus reglas también, la secuencia y la condición [11]. Las máquinas de estados abstractos inventadas por Yuri Gurevich, constituyen uno de los modelos más generales de computación en la actualidad [8].

La definición del comportamiento de los agentes y la lógica del juego se realizará mediante scripts implementados en el nuevo lenguaje que se propone. El motor evaluará todos los scripts de los agentes del juego de manera secuencial mediante un único bucle iterativo WHILE-DO. Tras la evaluación de la lógica, el motor actualizará el estado del entorno y procederá a depurar las altas y bajas de agentes, así como el cambio de escenas o el final del juego. El bucle (y por lo tanto el juego) se ejecutará mientras la condición de salida del juego sea falsa[18].

# 4 Lenguaje

A partir de este modelo, el lenguaje resultante establece una estructura para la composición de la lógica que se evalúa en un bucle iterativo continuo, el game loop. Esta sintaxis seguirá el esquema de la notación de Backus-Naur [15].

## 4.1 Sintaxis

Haciendo una analogía con la sintaxis del lenguaje While [21], se procede a delimitar los elementos que compondrán las construcciones sintácticas en variables categóricas que indicarán el propósito de cada elemento del lenguaje:

- $n \in \mathbb{R}$, valores numéricos.
- $x$, como variables.
- $a$, como expresiones aritméticas.
- $b$, como expresiones booleanas.
- $S$, como comandos.

Estas variables categóricas pueden ser complementadas con subíndices del modo $a_1$, $a_2$, $a_3$ o $S_1$, $S_2$.

A partir de estos elementos se define la sintaxis abstracta para generar las expresiones aritméticas, las expresiones booleanas y los comandos del lenguaje, obteniendo la siguiente estructura sintáctica:

- $a ::= n \mid x \mid f(a) \mid a_1 + a_2 \mid a_1 \text{ - } a_2 \mid a_1 * a_2 \mid a_1 \text{ / } a_2$
- $b ::= true \mid false \mid x \mid a_1 < a_2 \mid a_1 \leq a_2 \mid a_1 = a_2 \mid a_1 \geq a_2 \mid a_1 > a_2 \mid a_1 \neq a_2 \mid \neg b$
- $S ::= x_1 := a_1 \mid x_1 := b_1 \mid S_1; S_2; \mid \textbf{if } b_1 \textbf{ then } S_1 \textbf{ else } S_2$

Por lo tanto, las expresiones aritméticas a se pueden componer mediante literales numéricos $n$, variables $x$ como referencias a propiedades del juego o de los objetos del juego, y por funciones $f(a)$ tales como el valor absoluto, valores aleatorios o operaciones trigonométricas.

En cuanto a las expresiones booleanas $b$ sólo tienen una de las siguientes formas. Diremos que tienen un elemento básico si es $true$, $false$ o $x$, siendo $x$ una variable booleana, o tiene la formas relacionales como $a_1 = a_2$ o $a_1 \leq a_2$ donde $a_1$ y $a_2$ son expresiones aritméticas. Diremos que tiene un elemento compuesto si tiene la forma $\neg b$, donde $b$ es una expresión booleana.

Esta especificación define la sintaxis abstracta del lenguaje y simplemente indica cómo construir expresiones aritméticas, booleanas y sentencias en el lenguaje.

## 4.2 Semántica

A un nivel semántico, el lenguaje se apoya fundamentalmente sobre las expresiones, tanto aritméticas como booleanas, y a su vez en comandos construidos mediante dichas expresiones. Los comandos se ejecutan en cada iteración del ciclo de juego evaluando el contenido de su expresión.

Estos comandos tienen la capacidad de transformar el estado del entorno del juego, así como de las propiedades del mismo y de los objetos del juego que lo componen. Estas transformaciones se realizan siguiendo el modelo de persistencia de información que se utiliza en bases de datos [7]. Es decir, con operaciones que realizan altas, bajas, modificaciones y consultas sobre el modelo. Estas operaciones son conocidas como las funciones CRUD (del inglés Create, Read, Update y Delete), pero aplicadas sobre el juego y la especificación de comportamiento del mismo. Para este lenguaje, se restringe el uso de las altas y bajas a su uso sobre objetos del juego, y el uso de modificaciones y consultas sobre las propiedades del juego (estado social) y de sus objetos (agentes). Del mismo modo, las altas de nuevos elementos se realizan mediante el operador **new**. El cual crea una instancia de un nuevo objeto a partir de otro existente en el juego.

**new** *gameObject*

Mientras que para realizar una baja, el lenguaje incluye el operador **delete**. Con esto, el sistema eliminará del modelo de datos el objeto que se le indique.

**delete** *gameObject*

Como cualquier otro lenguaje, las consultas a las propiedades se realiza mediante la llamada a las variables que representan la información del juego. Mientras que el caso de las modificaciones se realiza a partir de la asignación del resultado de la evaluación de una expresión a una variable que representa una propiedad del juego o de uno de sus elementos.

$$x_1 := x_1 + 1$$

Donde $x_1$ es una variable que representa una propiedad y $x_1+1$ una expresión aritmética.

Por otro lado, con el objetivo de elevar el nivel de abstracción del lenguaje, es posible generar conjuntos de comandos con propósitos específicos para el desarrollo de juegos. Por ejemplo para cambiar las propiedades de los objetos del juego, o para determinar si se ha producido una colisión entre dos objetos.

Del análisis de las expresiones y sus combinaciones se obtienen dos categorías: de acción y de condición. La primera derivada de las estructuras de asignación o modificaciones, y la segunda de las estructuras de selección. Los comandos de acción son aquellos que realizan acciones sobre los datos del juego. Los comandos de condición son aquellos que evalúan la relación entre parámetros. Un ejemplo de una expresión de acción es el siguiente:

$$x_1 + n_1 + abs(n2 + sqrt(x_2))$$

Mientras que una expresión de condición podría tener la siguiente apariencia:

$$x_1 \geq n_1 + abs(n2 + sqrt(x_2))$$

A partir de estas operaciones, el lenguaje presenta una semántica que se organiza en base a dos estructuras: las secuencias y las condiciones.

- **Secuencias**: de la forma $S_1$; $S_2$; $S_3$; Donde primero se ejecuta el comando $S_1$, después el $S_2$ y así sucesivamente.
- **Condición**, si la expresión $b$ es cierta entonces se ejecuta $S_1$, sí es falsa se ejecuta $S_2$.

Con esta estructura, el flujo de la evaluación de los comandos se puede definir como una estructura secuencial de árboles binarios. Donde un comando puede contener un nodo condición que dirija el flujo sobre los comandos situados en una de sus dos ramas. En la Figura 2 se puede ver un ejemplo de esta construcción.

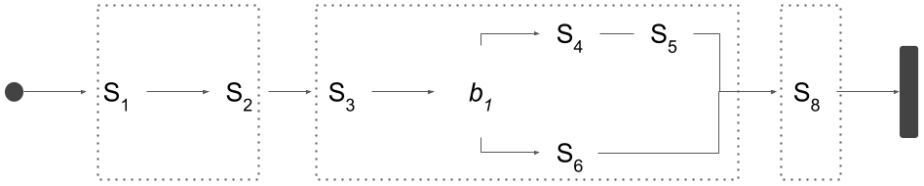


**Fig. 2.** Diagrama de la estructura del lenguaje compuesta por secuencias y condiciones.

Donde el conjunto más a la izquierda que contiene los comandos $S_1$ y $S_2$ representa los scripts de un objeto del juego, el del centro con la condición $b_1$ representa los de otro, y el bloque de la derecha representa los scripts del otro objeto diferente, el cual contiene el comando $S_8$.

## 5 Resultados y casos de uso

Con el objetivo de mostrar las características y las capacidades del lenguaje, a continuación se exponen los resultados obtenidos en la implementación de juegos.

Para probar la expresividad del lenguaje se ha implementado una gran cantidad de juegos entre los que destacan títulos arcade clásicos tales como Asteroids, Blocks, Frogger, Street Racer, PacMan, Mario Bros o el Space Invaders. El motivo tras la utilización de juegos arcade como demostradores y casos de uso reside en estudios que afirman que la programación de juegos presenta un contexto favorable para el aprendizaje de métodos genéricos de programación [3]. Se considera que si el lenguaje es capaz de implementar mecánicas de juegos arcade, también podrá implementar mecánicas más avanzadas derivadas de ellos, como las que se pueden encontrar en cualquier juego actual.

Durante el desarrollo de estos juegos a modo de demostradores, se ha pulido la especificación del lenguaje, llegando a agregar elementos necesarios y eliminando otros redundantes o innecesarios. Finalmente, el método ha demostrado ser robusto, ya que ha permitido completar con éxito la programación de todos

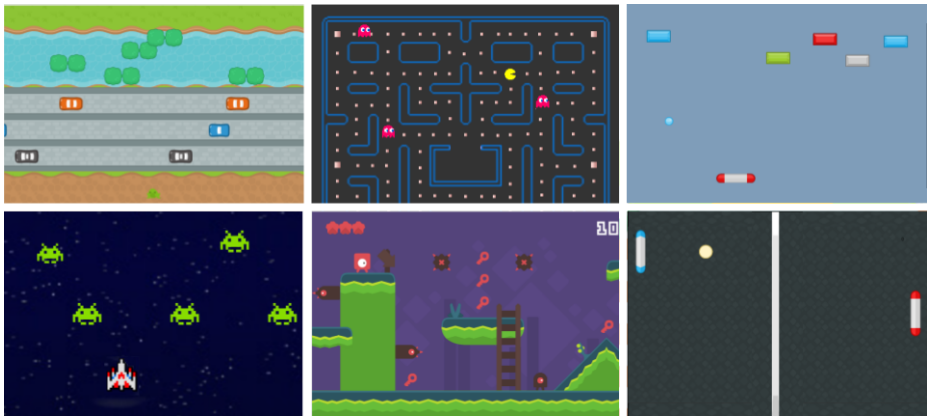

**Fig. 3.** Muestra de juegos implementados con el lenguaje.

los juegos propuestos hasta la fecha. En la Figura 3 se muestran algunos de los juegos arcade desarrollados.

A continuación, se expone como caso de uso el desarrollo del juego arcade Asteroids publicado por Atari en 1976. Un juego de disparos en el que se atraviesa un cinturón de asteroides con una nave espacial. El objetivo es destruir tantos asteroides como sea posible, evitando colisionar con ellos. La selección de este juego se debe a que incluye mecánicas muy comunes en juegos. En la Figura 4 se puede observar una captura del juego durante su ejecución. La definición del juego se va a centrar en cuatro objetos del juego: Nave, Láser, Asteroide y Generador de Asteroides.

**Nave**

El elemento principal del juego es la Nave, controlada por el jugador. Puede desplazarse hacia delante en función de la orientación de su morro. A continuación, se muestra el script que controla el movimiento de la Nave con tres eventos que permiten la rotación a la izquierda y a la derecha, y el empuje de la nave.

```
if keyLeft then Me.angle := Me.angle + 0.1;
if keyRight then Me.angle := Me.angle - 0.1;
if keyUp then
        Me.x := Me.x + 100 * cos(Me.angle);
        Me.y := Me.y + 100 * sin(Me.angle);
```

Donde las variables *keyLeft*, *keyRight* y *keyUp* corresponden a eventos del teclado que forman parte del estado social del juego. Por otro lado, la variables *Me.x*, *Me.y* y *Me.angle* hacen referencia a las propiedades de posición y rotación

de la nave. Además en el script también se utilizan las funciones trigonométricas seno y coseno que permiten crear expresiones aritméticas.

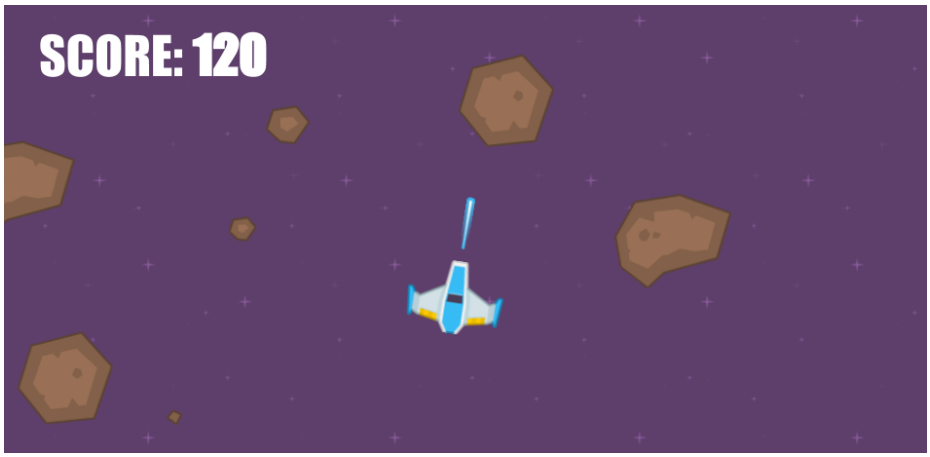


**Fig. 4.** Captura del juego Asteroids durante su ejecución.

Por otro lado, se contempla un script adicional que controla el comportamiento de la Nave al alcanzar los límites del escenario del juego. En ese caso, se reposiciona el objeto en el punto contrario del escenario en función del tamaño del mismo y de sus propias dimensiones.

**if** Me.x > screenWidth - Me.width **then** Me.x := Me.width;
**if** Me.x < Me.width **then** Me.x := screenWidth - Me.width;
**if** Me.y > screenHeight - Me.height **then** Me.y := Me.height;
**if** Me.y < Me.height **then** Me.y := screenHeight - Me.height;

Para completar el objetivo del juego, la Nave debe ser capaz de disparar objetos de tipo Láser hacia los Asteroides. Esta mecánica produce nuevos objetos Láser bajo la Nave y los desplaza en función de la orientación de la Nave en el momento de su creación y de un parámetro de velocidad predefinido. El código que controla esta mecánica se compondría de la siguiente manera:

**if** keySpace **then new** Laser(*params*);

Donde **new** Laser(*params*) genera un nuevo objeto Láser con la posición y la rotación de la Nave como parámetros.

Por otro lado, el jugador debe evitar que la Nave colisione con los Asteroides. Sí este evento ocurriera, el sistema detectará el evento de colisión y procederá a destruir el objeto Nave y a activar la condición de salida del juego.

```
if Me.collisionWithAsteroid then
    destroy Me;
    exit := true;
```

**Láser**
Tras su creación el Láser tiene dos scripts que controlan su comportamiento. El primero controla el desplazamiento. En función de su rotación, establecida por la Nave al crear el objeto Láser, y de un parámetro de velocidad arbitrario, el objeto se desplaza continuamente por el escenario.

```
Me.x := Me.x + 300 * cos(Me.angle);
Me.y := Me.y + 300 * sin(Me.angle);
```

El segundo script gestiona la colisión del Láser contra un Asteroide. Cuando el sistema detecta la colisión, se procede a la destrucción del Láser.

```
if Me.collisionWithAsteroid then delete Me;
```

**Generador de Asteroides**
Los asteroides son los objetos del juego que realizan el papel de antagonista. Son creados por un elemento auxiliar encargado de generar arbitrariamente en el escenario nuevos objetos Asteroide mediante un script que controla su creación en función del tiempo.

```
if spawnTimer ≥ 1
    then
        new Asteroide(params);
        spawnTimer := 0;
    else
        spawnTimer := spawnTimer + deltaTime;
```

**Asteroide**
Por último, el Asteroide se desplaza arbitrariamente por el escenario y únicamente actúa cuando colisiona con un objeto Láser. En este momento, según el juego original, aumenta la puntuación de la partida, procede a dividirse en dos nuevos objetos Asteroide de menor tamaño, y por último se destruye.

```
if Me.collisionWithLaser then
    score := score + 1;
    new Asteroide(params);
    new Asteroide(params);
    delete Me;
```

## 6 Conclusiones

En este trabajo se presenta un lenguaje que permite definir lógicas de juegos a partir de una semántica que contiene un conjunto reducido de acciones y condiciones. Los ejemplos de juegos elegidos se corresponden con mecánicas implementadas en base al desarrollo de juegos clásicos. El lenguaje se ha demostrado eficaz en la tarea de especificar juegos, ya que ha sido capaz de completar todas las mecánicas que se han propuesto hasta la fecha.

La simplicidad del lenguaje se fundamenta en los teoremas de la programación estructurada y sus derivaciones, como el teorema Folk. Se ha demostrado que es posible crear un lenguaje para la creación de juegos donde sólo existen estructuras condicionales y comandos compuestos por expresiones, sin bucles. La evaluación de esta semántica se fundamenta en la iteración continua del juego a través del ciclo de juego. Además, se ha observado que operadores lógicos como AND y OR no son necesarios. Del mismo modo que tampoco es necesario el uso de estructuras de datos complejas como vectores o matrices, aunque esto deriva del modelo de datos basado en el paradigma de los sistemas multiagente.

Como trabajo futuro, se pretende el desarrollo de este modelo de lenguaje aplicado a algoritmos típicos en lógica de juegos y en la aplicación del lenguaje a un contexto programación concurrente.

## Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Ciencia, Innovación y Universidades (PID2019-106426RB-C32, RTI2018-098651-B-C54) y por proyectos de investigación de la Universidad Jaume I (UJI-B2018-56, UJI-B2018-44). Además, este trabajo ha sido posible gracias a los recursos gráficos creados por Kenney.nl.

## References

1. Anderson, E. F. (2011). 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.
2. Ashcroft, E., Manna, Z. (1979). The translation of go-to programs to while-programs. In Classics in software engineering (pp. 49-61).
3. Becker, K., Parker, J. R. (2005). All I ever needed to know about programming, I learned from rewriting classic arcade games. In Future Play, The International Conference on the Future of Game Design and Technology.

4. Bohm, C., Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. Communications of the ACM, 9(5), 366-371.
5. Chover, M., Marín, C., Rebollo, C., Remolar, I. (2020). A game engine designed to simplify 2D video game development. Multimedia Tools and Applications, 1-22.
6. Dahl, O. J., Dijkstra, E. W., Hoare, C. A. R. (Eds.). (1972). Structured programming. Academic Press Ltd.
7. 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.
8. Dershowitz, N. (2012). The generic model of computation. arXiv preprint arXiv:1208.2585.
9. Dijkstra, E. W. (1970). Notes on structured programming.
10. Gregory, J. (2018). Game engine architecture. crc Press.
11. Gurevich, Y., Börger, E. (1995). Evolving algebras 1993: Lipari guide. Evolving Algebras, 40.
12. Harel, D. (1980). On folk theorems. Communications of the ACM, 23(7), 379-389.
13. Ho, X., de Joya, J. M., Trevett, N. (2017). State-of-the-art webGL 2.0. In SIGGRAPH Asia 2017 Courses (pp. 1-51).
14. Kernighan, B. W., Wyk, C. J. V. (1998). Timing trials, or the trials of timing: experiments with scripting and user-interface languages. Software: Practice and Experience, 28(8), 819-843.
15. Knuth, D. E. (1964). Backus normal form vs. backus naur form. Communications of the ACM, 7(12), 735-736.
16. Kosaraju, S. R. (1974). Analysis of structured programs. Journal of Computer and System Sciences, 9(3), 232-255.
17. Kozen, D., Tseng, W. L. D. (2008, July). The Böhm–Jacopini theorem is false, propositionally. In International Conference on Mathematics of Program Construction (pp. 177-192). Springer, Berlin, Heidelberg.
18. Marín-Lora, C., Chover, M., Sotoca, J. M., García, L. A. (2020). A game engine to make games as multi-agent systems. Advances in Engineering Software, 140, 102732.
19. Mills, H. D. (1972). Mathematical foundations for structured programming.
20. Mirkowska, G. (1972). Algorithmic logic and its applications. Doctoral diss., Univ. of Warsaw.
21. Nielson, H. R., Nielson, F. (1992). Semantics with applications (Vol. 104). Chichester: Wiley.
22. Nielson F., Nielson H. R. (1999). Principles of Program Analysis. Springer.
23. Nystrom, R. (2014). Game programming patterns. Genever Benning.
24. Prather, R. E. (1977). Structured turing machines. Information and Control, 35(2), 159-171.
25. 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.
26. Solin, K. (2011). Normal forms in total correctness for while programs and action systems. The Journal of Logic and Algebraic Programming, 80(6), 362-375.
27. Sweetser, P., Wiles, J. (2005). Scripting versus emergence: issues for game developers and players in game environment design. International Journal of Intelligent Games and Simulations, 4(1), 1-9.
28. Varanese, A. (2002). Game Scripting Mastery (Premier Press Game Development (Paperback)). Course Technology Press.