applied sciences

MDPI

Article

Design and Implementation of a Compiled Declarative Language for Game AI Control

Christopher Cromer †, Martin Araneda † and Clemente Rubio-Manzano *

Department of Information Systems, University of the Bío-Bío, Concepcion 4030000, Chile; chris@cromer.cl (C.C.); martin.araneda1501@alumnos.ubiobio.cl (M.A.)
* Correspondence: clrubio@ubiobio.cl
† These authors contributed equally to this work.

Abstract: Video games have become one of the most popular forms of entertainment around the world. Currently, agents (bots or non-player characters) are predominantly programmed using procedural and deterministic imperative techniques, which pose significant drawbacks in terms of cost and time efficiency. An interesting and alternative line of work is to develop declarative scripting languages which align the programming task closer to human logic. This allows programmers to intuitively implement agents' behaviors using straightforward rules. In this regard, most of these languages are interpreted, which may impact performance. Hence, this article presents the design and implementation of a new declarative and compiled scripting language called Obelysk for controlling agents. To test and evaluate the language, a video game was created using the Godot game engine, which allowed us to demonstrate the correct functionality of our scripting language to program the AIs participating in the video game. Finally, an analytics platform was also developed to evaluate the correct behavior of the programmed agents.

Keywords: game artificial intelligence; scripting programming language; computer games

Academic Editors: Maxim Mozgovoy and Paolo Burelli

Received: 5 September 2024
Revised: 20 November 2024
Accepted: 23 November 2024
Published: 27 December 2024

Citation: Cromer, C.; Araneda, M.; Rubio-Manzano, C. Design and Implementation of a Compiled Declarative Language for Game AI Control. *Appl. Sci.* **2025**, *15*, 157. https://doi.org/10.3390/app15010157

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Video games have surpassed cinema in market share, establishing themselves as one of the most popular and complex forms of entertainment worldwide. Furthermore, they have been integrated into various disciplines, including education and scientific research. For instance, modern video games are recognized as a quality and cost-effective alternative for evaluating machine learning algorithms [1].

A critical phase in video game development is the modeling and programming of character behavior. These characters, which move autonomously, play a fundamental role in the game's overall quality by making it more challenging and enjoyable [2]. The role of agents in video games has at least two dimensions: quantity and quality. Quantity pertains to having a sufficient number of actors to provide players with an immersive experience that mimics the real world. Quality involves creating the illusion of believable behavior, which is closely tied to the artificial intelligence of these actors [3].

One of the essential phases in the quality aspect is modeling and programming the characters' behavior. Currently, agents are programmed using deterministic imperative techniques such as behavior trees, state machines, or scripting procedural languages [4]. Procedural programming techniques have some disadvantages [5]: (i) the presence of control flow instructions limits the logical content of the program; (ii) non-mathematical use of variables and the fact that the assignment statement is sensitive to computational history makes reasoning about program properties difficult; and (iii) leaving control issues

to the programmer is not the most appropriate because sometimes the programmer must adapt programs to the underlying computer architecture.

In the context of computer games, some authors have highlighted the advantages of using declarative techniques: (i) they offer greater flexibility and maintainability and (ii) programmers do not need to hard-code all the transitions between each possible game situation [6]. Additionally, declarative programs allow for the creation of more complex Non-Player Character (NPC) behavior with fewer lines of code, making the overall program easier to understand [7].

On the other hand, the literature in computer games highlights several challenges and open problems related to agent programming: (i) *"Developing autonomous agents capable of performing complex tasks is both costly and time-consuming. The most significant expense in this process is the extraction of knowledge from human experts and its subsequent adaptation to the specific environment"* [8]; (ii) *"The artificial intelligence (A.I.) in modern video games still relies on rules-based approaches and has yet to create compelling, human-like opponents"* [9]; (iii) *"Additionally, as players become familiar with the mechanics of the game, improve their skills, and devise new strategies, the agents remain static and eventually become obsolete"* [1].

An interesting and alternative line of work is to develop declarative scripting languages which align the programming task closer to human logic, and the programmer can implement the agents' behavior more intuitively while using simple rules. Some authors have pointed out that: (i) *"smarter, more powerful scripting languages will improve game performance while making gameplay development more efficient"* [10]; (ii) *"logic-based approaches, can gain several valuable and useful suggestion when applied to such a broad and varied domain, for instance on how to improve the languages and the techniques in order to make them more affordable for people without a deeper knowledge of logic or more "productive", i.e., to develop specific extensions that make easier and faster some reasoning tasks"* [11]; (iii) *"knowledge bases are much more flexible and easier to change than a procedural algorithm [...] strategies can be easily encoded as a set of few rules, while the development of dedicated algorithms is time consuming and error prone"* [12].

In this sense, some interesting scripting has been proposed [12–16]. Scripting languages are widely used in many areas, particularly in artificial intelligence, due to its rapid development capabilities. Because it does not require compilation, iteration time during programming is significantly reduced. In the context of video games, the use of declarative languages for programming agents' behavior in video games has been carried out from different points of view. We can group the works into four categories: (i) based on the Prolog language [17–19]; (ii) based on the GOLOG language [15]; (iii) based on Planning [20]; and (iv) based on Answer Set Programming (ASP) [12,21].

Here, we review the studies most relevant to our work. In [15], the authors describe and explain a logical programming language called GOLOG that allows for the implementation of FPS (First Person Shooters) style game agents and to face them against NPCs. The type of A.I. used is goal-based and employs planning techniques. Additionally, the article [14] explains creating and implementing a script based on the Prolog language within an FPS (First Person Shooter) style game to create different behavioral tactics in a bot team made up of agents. The agent used in this project is of the objective-based rational type, based on a decision tree with conditional instructions. On the other hand, in [12], an ASP extension is used to model discrete knowledge about the game and the situations that can be experienced in a 2D physics-based game.

Unlike other scripting languages, Obelisk is a compiled and declarative programming language that gives programmers a double advantage: (i) it performs better as the compilation is completed before the code is executed and (ii) the declarative language allows the bots to be programmed, indicating how the behavior should be performed from a visual

point of view. Still, we only tell the agent what they should do, that is, how it behaves regardless of the computer graphics aspects. The Table 1 shows the key differences between Obelisk and the rest of the mentioned scripting languages for agent programming.

Table 1. A comparative table between scripting programming language for programming bots: imperative vs. declarative (NEA: None engine associated).

Scripting Laguage	Paradigm	Engine	Genres	Other Features
Lua [22]	Imperative	Love2D	All	Lua Virtual Machine programmed in C
Unreal script [23]	Imperative	Unreal	All	C++
ScriptEase [24]	Imperative	NEA	BioWare's Neverwinter Nights game	Scripting Version of C
BotL, TELL, TED [7]	Declarative	NEA	Strategy (City of Gansters)	C#
Golog [15]	Declarative	NEA	First-person shooter, Multiplayer	Prolog
Script in Prolog [14]	Declarative	NEA	First-person shooter	Swi-Prolog
Obelisk	Declarative	Godot	2D Platform	C++, SQLite

The primary innovation of our proposal is the skillful combination of several concepts, approaches, techniques and components, such as Game Artificial Intelligence, Scripting Programming Language, and Computer Games. In particular, the design, implementation, and evaluation of a declarative scripting language using C++, which translates instructions into a knowledge representation stored in SQLite. Additionally, we developed a video game using the Godot engine and created a library that enables communication between the language and the engine, facilitating practical application in real-world scenarios. In fact, we tested the language by using a video game that was created using the Godot game engine and it allowed us to demonstrate the correct functionality of our scripting language to program the AIs of the video game. Finally, for evaluating the behavior of the bots programmed with our language, we created an analysis platform using Go and R. This platform assesses the quality of the AIs based on the behavior of the entities compared to human players. While the nature of video games presents challenges in comparing our proposal to others—due to variations in games—our language is fully prepared for integration into any development project. Therefore, the highlights of the paper are as follows:

- A new declarative and compiled scripting language is presented
- We explain its main features from two points of view: its syntax and performance (computational and behavioral).
- We test it by using classical and statistical techniques
- We show that the process of programming is transparent and very interpretable
- A complete platform for game play analytics is also detailed

The rest of the paper is organized as follows: Section 2 introduces several general concepts related to computer games and provides a brief overview of ALAI, the computer game where our language was implemented. Section 3 presents the Obelisk language using an illustrative example, and we focus on syntax, semantics, and the architecture of the system, explaining each of its modules in detail. Section 4 explains how the analysis of games and the evaluation of the credibility of the agents have been carried out when they are programmed using Obelisk. Finally, Section 5 provides the conclusions and details of future work.

2. Preliminary Concepts

This chapter aims to explain the key concepts involved in developing a programming language for controlling Artificial Intelligence, and its implementation within a platform-style game.

2.1. Source Code Compilation

Source code compilation is the process of transforming a high-level programming language into machine-readable code. This procedure consists of several steps, which are described below:

- Lexer: The lexer reads the text and performs lexical analysis to identify recognized symbols and tokens. A symbol, such as '#', represents a comment. A token is a set of symbols that conveys meaning, such as variable names, function names, numbers, or keywords like 'if' and 'while'.
- Parser: The parser interprets and processes the tokens and symbols received from the lexer. It uses these tokens and symbols to construct an Abstract Syntax Tree (AST), which represents the hierarchical syntactic structure of the source code.
- Abstract Syntax Tree (AST): The AST is used to generate intermediate code, which helps control the flow of logic. It also plays a crucial role in producing the final object code, which is then compiled into corresponding object files.

2.2. Video Game Engine

A video game engine is a set of programming routines designed to simplify the use of graphic elements, sounds, physics, networks, and various other aspects essential for video game development. The engine chosen for this project is "Godot" [25,26]. Godot is highly valuable for project development, especially for its ability to implement pixel art. Unlike similar engines such as Unity or Unreal, where distances between elements are measured in meters, Godot measures these distances in pixels. This feature significantly enhances efficiency when working on 2D games, which typically use pixels rather than meters.

Another significant factor is the presence of "GDNative" in Godot. GDNative is an API that enables the use of programming languages such as C, C++, or Rust, which can compile code into machine language. This capability is particularly crucial for the architecture, as it facilitates the integration of additional programming languages to interact with GDNative [27]. The API also enables the AI to make decisions more swiftly, as machine object code executes faster than interpreted languages or those compiled to bytecode. This performance advantage is demonstrated in [28].

2.3. 2D Alai Computer Game

The Alai video game comprises five essential modules (see Figure 1). The first module, named "Coin", incorporates the necessary functionality to manage the acquisition and non-acquisition of coins. The "Goal" module facilitates the implementation of functionalities linked to achieving objectives. The "GUI" module is responsible for user interaction. The "Player" module, equipped with essential features, governs character control and the corresponding graphic motion actions. This module is intricately connected to the "State Machine" module, as its structure can be implemented based on it. For our initial proof of concept, we opted for the development of a 2D action game. In this context, the language facilitates the programming of video game characters, empowering them to execute essential actions, including:

1. Surmounting obstacles, with the added capability of executing speedy jumps.
2. Collecting items (coins) that contribute to the overall score.

3. Navigating away from enemies or employing the option to perform agile jumps over them.

The Alai computer game video can be found at the link https://alai.cromer.cl/ (accessed on 22 November 2024):

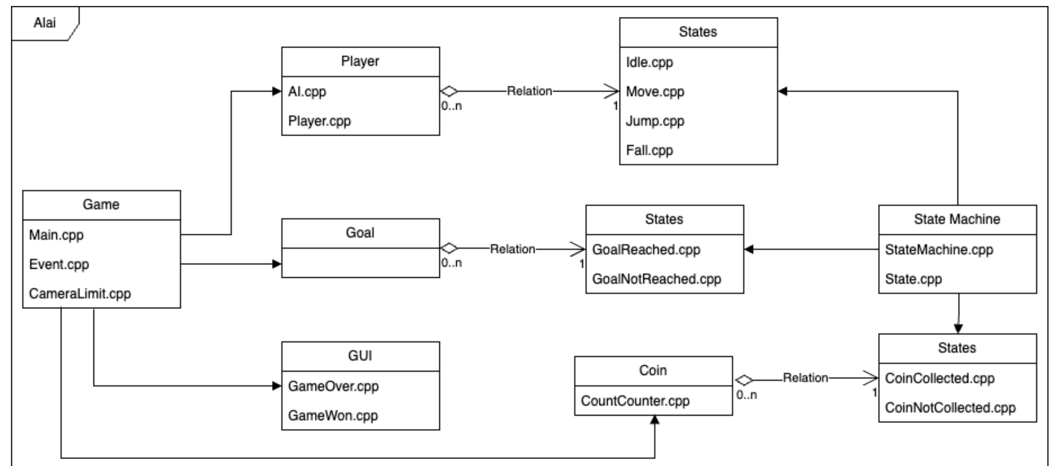

Figure 1. 2D Alai Computer Game Structure.

3. The Obelisk Language

Historically, AI has been associated with logic-based or symbolic methods such as Reasoning, Knowledge Representation, and Planning. In this section, we show the language's syntax by means of examples, it has been inspired in symbolic methods where the emphasis is on the representation of knowledge. Our language will be responsible for the AI's decision-making and will be custom-designed and inspired by Prolog, a declarative and logical programming language. The Obelisk language offers versatility, allowing for the programming of agent behavior through if-then rules in a broad spectrum of games, encompassing both 2D and 3D environments (see Figure 2).

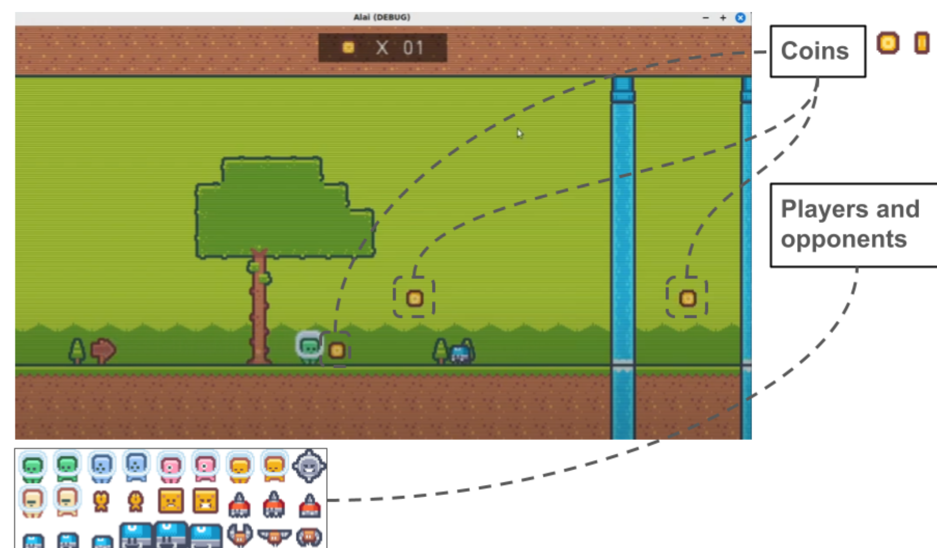

Figure 2. 2D Alai Computer Game: players, opponents and coins.

It is crucial to emphasize that the entity control within a video game is orchestrated by the game itself, facilitated by the underlying graphics engine. In essence, the availability of actions, like "Jump", is contingent upon the implementation of corresponding functions within the game's programming language. Consequently, the versatility of Obelisk extends

to virtually any video game, provided there exists a parallel action in the game that interfaces with a graphics engine. In other words, if "Jump" is a feature in Obelisk, it seamlessly translates to Alai, as long as the game is programmed using a compatible engine like Godot.

3.1. Syntax

The syntax of the language was designed to resemble human language, specifically English. This choice was made because English is widely used globally and is relatively easy to learn and use. The syntax structure of the "Obelisk" programming language consists of three parts: the name, the elements, and the semicolon (;). Together, these components form what the language recognizes as a keyword. The verb, referred to as the "semantic separation", is the element that divides the meaning of a linguistic expression containing a keyword. This separation is crucial for maintaining consistency in the written language when creating logical keywords. It is important to note that language keywords are restricted to English.

The language currently has three implemented keywords: actions, facts, and rules. These keywords share a similar structure, differing only in the content of their elements. In particular, it is formally defined by using the extended Backus–Naur notation as follows:

```
facts = ’fact’, "(", entities,  white space,  verb,
                     white space, entities, ")", ";" ;

fact = ’fact’, "(", entity,  white space,  verb,
                    white space, entity, ")", ";" ;

rule = ’rule’, "(", fact,  white space,
                    ’if’,  white space,
                    fact ")", ";" ;

action = ’action’, "(", ’if’, fact, ’then’,
                    white space, suggested_action,
                    white space, ’or’,  white space,
                    suggested_action ")", ";" ;

entities = ’"’, entity, ’"’, [ { white space, ’and’,
white space, ’"’, entity, ’"’ } ] ;

entity = ’"’, { all characters - ’"’ }, ’"’ ;

suggested_action = ’"’, { all characters - ’"’ }, ’"’ ;

verb = { all visible characters } ;

white space = ? white space characters ? ;

all visible characters = ? all visible characters ? ;

all characters = ? all characters ? ;
```

3.2. Semantics

The semantics of the language was designed to resemble human language, specifically English. Our language has three reserved words: actions, facts and rules. Its operation will be described below:

- Facts: They describe an assertion about the entities declared in the keyword elements. If a fact is not present in the knowledge base, it is implied that its boolean value is *false* by default.
- Rules: The rules are used to create conditions, and the result of these depends fundamentally on the facts. This means that a rule will have a boolean value *true* if a fact supports the rule. Otherwise, the rule will remain with the value *false*.
- Actions: Finally, the actions define which act will be carried out depending on whether the fact to which it alludes has a boolean value *true* or *false*.

For example, we can control behavior using a declarative program comprising seven facts, only one rule, and four actions. The program is highly interpretable and easy to understand.

```
fact("coin" is "gold");

rule("gold coin" is "collectable" if "coin" is "gold");

fact("goal" is "touchable");

fact("shelly" is "dangerous");

fact("shelly" is "bouncable");

fact("shelly" can "walk");

fact("dreadtooth" is "dangerous");

fact("dreadtooth" can "walk");

action(if "shelly" is "bouncable" then "jump on" else "jump over");

action(if "dreadtooth" is "bouncable" then "jump on" else "jump over");

action(if "coin" is "collectable" then "collect" else "avoid");

action(if "goal" is "touchable" then "touch" else "avoid");
```

A video showing the final result can be found at the link https://youtu.be/wnXKkSL6B0Q (accessed on 22 November 2024):

The details about the implementation and its execution is shown in Table A1 (see Appendix A).

3.3. Obelisk Language Architecture

The language architecture comprises three modules (see Figure 3): compiler, knowledge base, and library. Obelisk is responsible for creating the game's AI by transforming the sets of facts, rules, and actions into a knowledge base which is implemented using SQLite, allowing queries during execution.

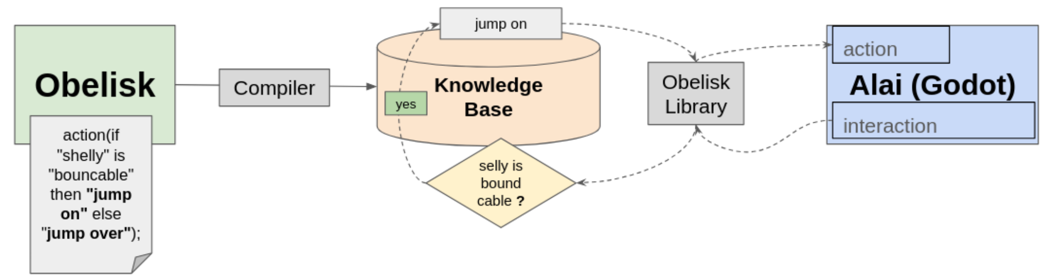

Figure 3. The software architecture comprises three modules: compiler, knowledge base, and library. It is also shown how the language communicates with the video game engine for a hypothetical case of jumping.

3.3.1. Compiler

The Obelisk compiler is designed to read source code using a Lexer. The Lexer generates tokens, which are then processed by the Parser to apply the appropriate operations. Finally, a knowledge base is created, allowing its information to be accessed by external software.

3.3.2. Knowledge Base

The knowledge base stores all the logic derived from facts, rules, and actions. It is implemented using SQLite, enabling queries to be made using the SQL language. The structure of the knowledge base consists of six tables (see Table 2) that store the information relevant to the control of the characters: names of the entities; names of the verbs used in the facts and rules; names of the possible actions that can be performed; facts that can be true or false; rules; actions that can be taken depending on whether the fact is true or false.

Table 2. Set of tables that implement the Knowledge Base.

Name of Table	Description
entity	The entity table is used to store the names of the entities present in the facts, rules and actions tions.
verb	The verb table is used to store the names of the verbs used in the facts and rules.
action	The action table stores the names of the possible actions that can be taken.
fact	The fact table has the facts that can be true. rosy or fake.
rule	The rule table contains rules. If a rule turns out to be true, the fact is inserted into the fact table.
suggest_action	The suggest_action table contains the two possible actions. tions that can be taken depending on whether the fact Is true or false.

3.3.3. Obelisk Library

Using external software, the Obelisk library allows you to interact and consult the Obelisk Knowledge Base. There are two types of data that the query can return. A string representing the action to take, or a number between 0 and 1 representing its associated truth value.

3.3.4. Incorporation of the Language in the Video Game Engine

To incorporate the language into external software, such as the video game *Alai*, the library can be used to query the compiled knowledge base and take action accordingly. It is the software's responsibility to set the action in motion. This can be accomplished

using various artificial intelligence techniques such as A*, state machine, and goal-oriented action planning.

Finally, the linker takes the object files, puts them together, and turns them into an executable library that the machine can use. The executable and library binaries contain machine code based on the system architecture, e.g., 86_64 in case of 64bit GNU/Linux, which is usually found on desktop or notebook computers used nowadays.

4. Bots Performance, Game Analysis and Credibility AI Assessment

There is a significant distinction between the algorithms used for programming gaming bots and those used in traditional applications. Classical algorithms often strive to simulate near-optimal intelligent behavior, such as finding the shortest path between two locations. In contrast, gaming bot algorithms focus on creating engaging and entertaining opponents for human players rather than optimal ones [29]. As noted by [30], the concept of AI in gaming is distinct: *"Artificial intelligence consists of emulating the behavior of other players or the entities they represent. The key concept is that the behavior is simulated. In other words, AI for games is more artificial and less intelligent. The system can be as simple as a rules-based system or as complex as a system designed to challenge a player as the commander of an opposing army."* This difference underscores the unique goals of gaming AI, which prioritize player experience over computational efficiency or accuracy.

In this context, a variation of the Turing test was proposed and designed in [31] to evaluate the capability of computer game bots to mimic human players. The concept is described as follows: *"Suppose we are playing an interactive video game with some entity. Could you tell, solely from the conduct of the game, whether the other entity was a human player or a bot? If not, then the bot is deemed to have passed the test"*.

Therefore, our primary objective in evaluating the bots is to assess the accuracy and credibility of the artificial intelligence's behavior within the video game. To achieve this, we compare the programmed agents to human players. In the context of our computer game, this assessment includes metrics such as the number of coins collected, the number of deaths, and other relevant performance indicators.

4.1. System

4.1.1. Monitor

We developed a monitor using the GDScript language, part of the Godot video game engine, to record the entire session of the agent and/or players. The recording process is synchronized with the Godot engine's update cycle. For instance, if the player's screen refresh rate is 60 Hz, 60 frames of information are recorded per second. At the end of a match, all recorded data are transmitted to a server for processing and storage for future analysis. Notably, all data are sent in JSON format to a REST server.

4.1.2. Server

The server was programmed in Go due to its high performance and low resource usage, such as CPU and RAM. This efficiency enables the simultaneous reception of large volumes of data from multiple games without issues. Additionally, the server offers a REST API with several endpoints for storing game data and retrieving stored information for future analysis.

4.1.3. Database

All the information on the games played are stored in a MySQL database (see Table 3).

Table 3. Set of tables that implement the Obelisk Analytics platform.

Name of Table	Description
frame	The frame table is used to store the number of points, coins, time passed, and positions of objects in the world each frame the game draws.
game	The game table contains information about the match and the team running that match, including operating system screen dimensions and similar data.
godot_version	The godot_version table is used to store the versions of Godot that have run a match of the game.
level	The level table is a parameter table that contains the names of the levels that can be played.
object	The object table is used to store the state, position and speed of an object in a specific frame of the match.
object_name	The object_name table is a parameter table that contains the names of all objects that exist in a game match.
object_state	The object_state table has all the names of the states of an object.
os	The os table is a parameter table that contains the possible operating systems that can play the game.
user	The user table contains the users and passwords of the people who have permission to query the API to obtain the data and use it.
player	The player table is an optional table where the data of the person who is playing a match can be stored to be able to do an analysis without being anonymous.

4.1.4. Web Page

We developed a web page using Vue.js capable of retrieving and displaying information from a database that stores played games. The site features a comprehensive login system, CRUD operations for essential tables, and a section displaying game data on a deterministic graph.

4.2. Analysis

Various analyses can be conducted using the data gathered from our agent. To streamline the study, we employ the R programming language for comprehensive game statistics calculations. R's capability to generate graphs based on data and formulas greatly enhances our ability to grasp and interpret agent/player behavior effectively. This kind of analysis could be used in the future for example to trains NPCs automatically [32].

4.2.1. Obelisk Analytics

We developed Obelisk Analytics to evaluate the correct behavior of the programmed agents within the video game. It is a web platform hosted at the URL https://alai.cromer.cl/ (accessed on 22 November 2024): which can receive information from the database that stores the games played. The site contains a fully functional login, CRUDs for essential tables and a web site to display the games played on a deterministic graph.

In particular, our platform is formed by three modules: (i) A monitor that records the entire session of the agent and players. All the game information is stored in a MySQL database. For each frame, the number of points, coins, the time elapsed, speed, and positions are saved. (ii) A server that allows us to receive data simultaneously from several

game sessions. The server also provides a REST-type API with several endpoints that will enable storing the game matches and consulting the information saved for future study. (iii) An R script that calculates all game statistics and generates graphs based on the stored data.

4.2.2. Games Analysis

In Figures 4 and 5, we analyze several games by using a normal probability distribution for coins and time. Utilizing the normal probability distribution involves analyzing the behavior of multiple games and subsequently computing the probabilities associated with specific events. This process enables a deeper understanding of the likelihood of various occurrences in the context of the games under consideration.

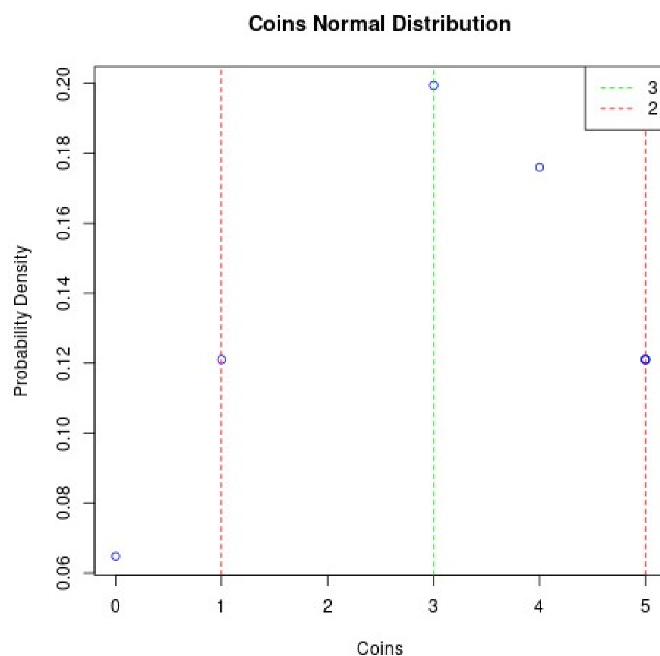

Figure 4. Normal probability distribution for coins.

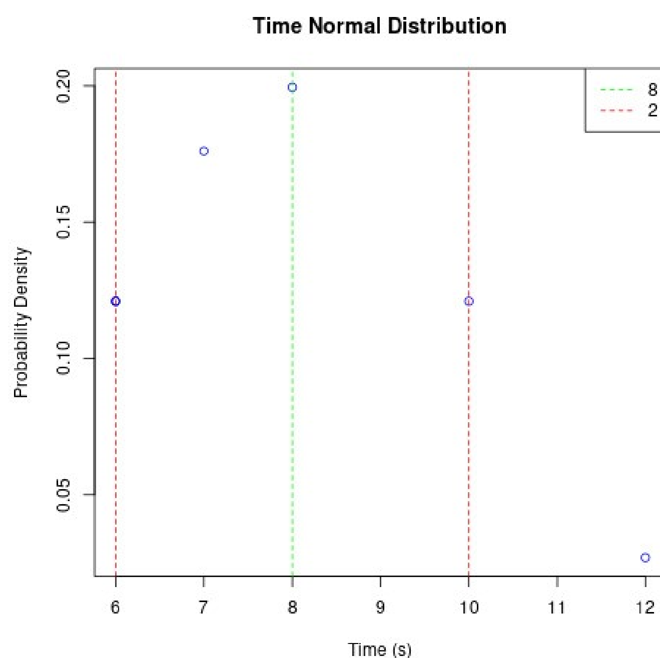

Figure 5. Normal probability distribution for time.

In the Figures 5 and 6, the X-axis depicts the number of coins acquired by the agent and/or player, while the Y-axis illustrates the probability density. Probability density is the relative likelihood that a random variable's value matches the corresponding sample. The green line signifies the average value denoted by the formula $\bar{X} = \frac{\Sigma_{i=1}^{n} X_i}{n}$, where $\bar{X}$ is the mean value, n is the number of games, and $\Sigma_{i=1}^{n} X_i$ is the sum of coins in each game. The red lines, known as standard deviations, are crucial indicators. Calculating the standard deviation involves first obtaining the variance, represented by the formula $\sigma^2 = \frac{\Sigma_{i=1}^{n} (X_i)^2}{n}$, where σ^2 is the variance. Subsequently, the standard deviation is derived through the formula $\sigma = \sqrt{\sigma^2}$, where σ represents the standard deviation.

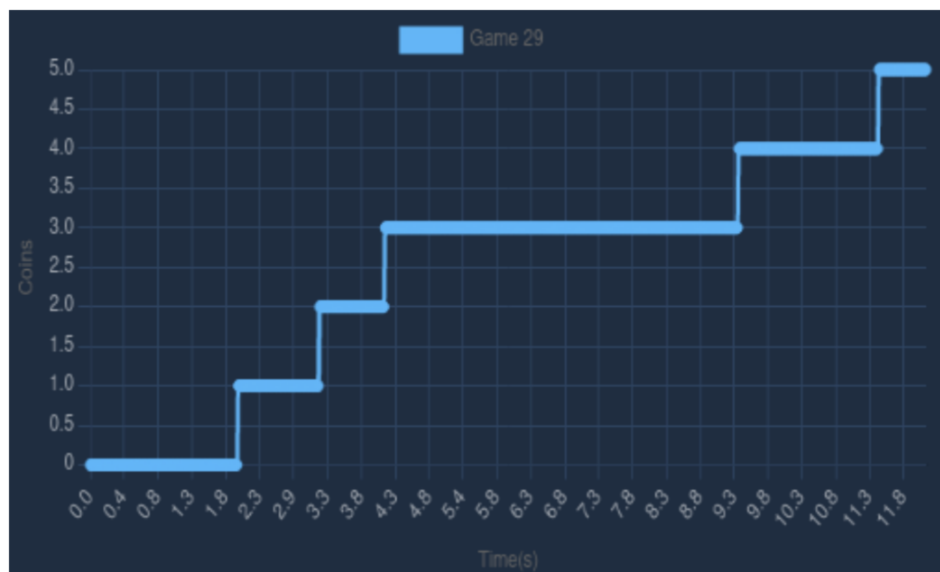

Figure 6. Time series summarizing the game session of an agent.

After 7 games, we had an average value of 3 coins represented on the green line. When calculating the standard deviation, we obtained the value 3; therefore, the first red line is placed at 1 coin because three minus 2 is 1. And in the case of the second red line, 3 plus 2 is 5. On the other hand, we obtained an average value of 8 s and a standard deviation of 2 s. We can conclude that, in the value of the variable, it is pretty likely that a game lasts between 6 s and 10 s, with a value very close to 8 s.

4.2.3. Time Series

We used the time series to analyze the agent's behavior during a game. This allowed us to understand better the decisions and how long it took to achieve the goals. In the Figures 6 and 7, we can see two different game types: agent and human. We compared the human and the AI, and the behavior was similar, but there was a difference when approaching an enemy, avoiding the enemy, and going back for the coin it missed while avoiding the enemy. At the same time, the human managed to avoid the enemy and obtain the coin in the same jump without the need to go back for it.

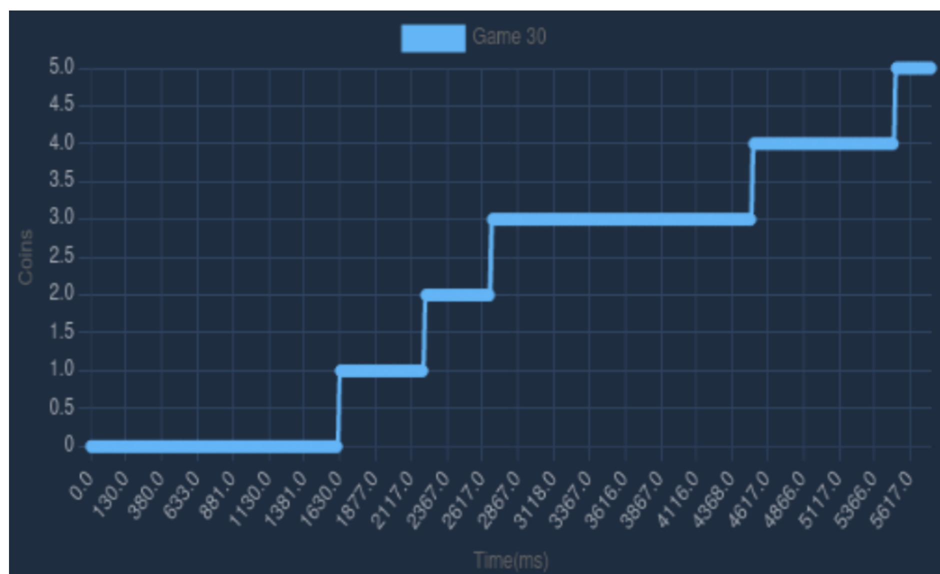

Figure 7. Time series summarizing the game session of an human player.

5. Conclusions and Future Works

In this work, we have introduced a new declarative programming language called Obelisk, inspired by symbolic methods with a focus on knowledge representation. Obelisk offers versatility in programming agent behavior through if-then rules across a wide range of games, including both 2D and 3D environments. We have demonstrated its main features from multiple perspectives:

- Syntax and Semantics: Through a series of examples, we showcased the syntax and semantics of Obelisk, which was designed to resemble human language, specifically English. The language includes three reserved words: actions, facts, and rules.
- Architecture: Obelisk is composed of three key modules: the compiler, knowledge base, and library. The compiler was implemented using C++, while the logical aspects of the language were built using an SQLite-based knowledge base, which stores elements like actions, facts, and rules.
- Evaluation: Through statistical techniques, we demonstrated that Obelisk allows for easy control of players in a manner similar to human players. The transparency and interpretability of this process are enhanced by the system's reliance on declarative rules, making AI behavior highly traceable.

Future work will focus on several areas to enhance the functionality and scope of Obelisk. These include refining the language to achieve Turing completeness, conducting advanced analyses of agent behavior using data from various software environments, and leveraging GPUs with libraries such as CUDA from Nvidia. We also plan to perform comprehensive comparisons with other approaches regarding operation and performance and extend our framework to other engineering applications [33,34].

Despite these future directions, Obelisk is currently ready for use by programmers specializing in AI for video games, offering a powerful tool for developing sophisticated and interpretable AI behavior.

Author Contributions: Conceptualization, C.R.-M.; methodology, C.C. and M.A.; software, C.C. and M.A.; validation, C.C. and M.A.; formal analysis, C.C., M.A. and C.R; investigation, C.C., M.A. and C.R; resources, C.C., M.A. and C.R; data curation, C.C., M.A. and C.R; writing—original draft preparation, C.C., M.A. and C.R.-M.; writing—review and editing, C.R.-M.; visualization, C.R.-M.; supervision, C.R.-M.; project administration, C.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Software Repository, Metadata Description

Table A1. Code metadata description, all needed information for using the project.

Nr.	Code Metadata Description	Version-URL
C1	Current code version	1.0.3
C2	Permanent link to code/repository used for this code version	https://github.com/cromerc/obelisk/releases/tag/1.0.3 (accessed on 22 November 2024)
C3	Code Ocean compute capsule	N/A
C4	Legal Code License	BSD-3-Clause
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	C++, C, SQLite, and Meson
C7	Compilation requirements, operating environments & dependencies	C++ 17, C 17, Meson, SQLite C bindings, doxygen, LLVM 14, and Linux Mint 21
C8	If available Link to developer documentation/manual	https://alai.cromer.cl/docs (accessed on 22 November 2024) and https://alai.cromer.cl/docs/refman.pdf (accessed on 22 November 2024)
C9	Support email for questions	chris@cromer.cl

References

1. Gemine, Q.; Safadi, F.; Fonteneau, R.; Ernst, D. Imitative learning for real-time strategy games. In Proceedings of the 2012 IEEE Conference on Computational Intelligence and Games (CIG), Granada, Spain, 11–14 September 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 424–429.
2. Feng, S.; Tan, A.H. Towards autonomous behavior learning of non-player characters in games. *Expert Syst. Appl.* **2016**, *56*, 89–99. [CrossRef]
3. Borovikov, I.; Harder, J.; Sadovsky, M.; Beirami, A. Towards interactive training of non-player characters in video games. *arXiv* **2019**, arXiv:1906.00535.
4. Uludağlı, M.Ç.; Oğuz, K. Non-player character decision-making in computer games. *Artif. Intell. Rev.* **2023**, *56*, 14159–14191. [CrossRef]
5. Julián, P.; Alpuente, M. *Logic Programming. Theory and Practice*; Pearson Education: Madrid, Spain, 2007.
6. Lapeyrade, S.; Rey, C. Non-Player Character Decision-Making with Prolog and Ontologies. In Proceedings of the 2023 IEEE Conference on Games (CoG), Boston, MA, USA, 21–24 August 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 1–2.
7. Zubek, R.; Horswill, I. Logic programming in commercial games: Experiences and lessons learned. In Proceedings of the Game Developer Conference, San Francisco, CA, USA, 17–21 March 2023. Available online: https://ianhorswill.github.io/game-resources/ (accessed on 22 November 2024) .
8. Van Lent, M.; Laird, J. Learning hierarchical performance knowledge by observation. In Proceedings of the Machine Learning-International Workshop Then Conference, Bled, Slovenia, 27–30 June 1999; Morgan Kaufmann Publishers, Inc.: San Francisco, CA, USA, 1999; pp. 229–238.
9. Thurau, C.; Paczian, T.; Sagerer, G.; Bauckhage, C. Bayesian imitation learning in game characters. *Int. J. Intell. Syst. Technol. Appl.* **2007**, *2*, 284–295. [CrossRef]
10. White, W.; Koch, C.; Gehrke, J.; Demers, A. Better scripts, better games. *Commun. ACM* **2009**, *52*, 42–47. [CrossRef]
11. Germano, S.; Leone, N.; Ianni, G. Logic Programming in Non-Conventional Environments. Ph.D. Thesis, Università della Calabria, Arcavacata, Italy, 2018.

12. Calimeri, F.; Fink, M.; Germano, S.; Humenberger, A.; Ianni, G.; Redl, C.; Wimmer, A. Angry-HEX: An artificial player for angry birds based on declarative knowledge bases. *IEEE Trans. Comput. Intell. AI Games* **2015**, *8*, 128–139. [CrossRef]
13. Chandra, B.; Cheslack-Postava, E.; Mistree, B.F.; Levis, P.A.; Gay, D. Emerson: Scripting for federated virtual worlds. In Proceedings of the 15th International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational and Serious Games (CGAMES), Louisville, KY, USA, 28–30 July 2010.
14. Jaśkiewicz, G. Prolog-scripted tactics negotiation and coordinated team actions for Counter-Strike game bots. *IEEE Trans. Comput. Intell. AI Games* **2014**, *8*, 82–88. [CrossRef]
15. Jacobs, S.; Ferrein, A.; Lakemeyer, G. Controlling unreal tournament 2004 bots with the logic-based action language golog. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, Lexington, KY, USA, 18–22 November 2005; Volume 1, pp. 151–152.
16. Abbadi, M.; Di Giacomo, F.; Cortesi, A.; Spronck, P.; Costantini, G.; Maggiore, G. Casanova: A simple, high-performance language for game development. In Proceedings of the Serious Games: First Joint International Conference, JCSG 2015, Huddersfield, UK, 3–4 June 2015; Proceedings 1; Springer International Publishing: Berlin/Heidelberg, Germany 2015; pp. 123–134.
17. Świechowski, M.; Mańdziuk, J. Prolog versus specialized logic inference engine in General Game Playing. In Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games, Dortmund, Germany, 26–29 August 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 1–8.
18. Kissmann, P.; Edelkamp, S. Instantiating general games using prolog or dependency graphs. In Proceedings of the Annual Conference on Artificial Intelligence, Sanya, China, 23–24 October 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 255–262.
19. Vittaut, J.N.; Méhat, J. Fast instantiation of GGP game descriptions using prolog with tabling. In Proceedings of the ECAI, Prague, Czech Republic, 18–22 August 2014; IOS Press: Amsterdam, The Netherlands, 2014; pp. 1121–1122.
20. Long, E. Enhanced NPC Behaviour Using Goal Oriented Action Planning. Master's Thesis, School of Computing and Advanced Technologies, University of Abertay Dundee, Dundee, UK, 2007.
21. Angilica, D.; Ianni, G.; Pacenza, F. Declarative AI design in unity using answer set programming. In Proceedings of the 2022 IEEE Conference on Games (CoG), Beijing, China, 21–24 August 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 417–424.
22. Brumbaugh, Z.; Leithner, M. The Lua Programming Language. In *Scripting Farming Simulator with Lua: Unlocking the Virtual Fields*; Apress: Berkeley, CA, USA, 2023; pp. 45–83.
23. Sanders, A. *An Introduction to Unreal Engine 4*; AK Peters/CRC Press: Boca Raton, FL, USA, 2016.
24. McNaughton, M.; Redford, J.; Schaeffer, J.; Szafron, D. Patttern-Based AI Scripting Using ScriptEase. In Proceedings of the Advances in Artificial Intelligence: 16th Conference of the Canadian Society for Computational Studies of Intelligence, AI 2003, Proceedings 16, Halifax, Canada, 11–13 June 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 35–49.
25. Adams, E.; Dormans, J. *Game Mechanics: Advanced Game Design*; New Riders: Indianapolis, IN, USA, 2012.
26. Bradfield, C. *Godot Engine Game Development Projects*; Packt: Birmingham, UK, 2018.
27. Manzur, A.; Marques, G. *Godot Engine Game Development in 24 Hours, Sams Teach Yourself: The Official Guide to Godot 3.0*; Sams Publishing: Carmel, IN, USA, 2018.
28. Royal Donut Games. CPU Voxel Benchmarks of Most Popular Languages in Godot. Abr. de 2022. Available online: http://www.royaldonut.games/2019/03/29/cpu-voxel-benchmarks-of-most-popular-languages-in-godot/ (accessed on 22 November 2024).
29. Soni, B.; Hingston, P. Bots trained to play like a human are more fun. In Proceedings of the 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), Hong Kong, China, 1–8 June 2008; IEEE: Piscataway, NJ, USA, 2008; pp. 363–369.
30. Kehoe, D. Designing Artificial Intelligence for Games. 2009. Available online: https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-gamespart-1 (accessed on 22 November 2024).
31. Hingston, P. A turing test for computer game bots. *IEEE Trans. Comput. Intell. AI Games* **2009**, *1*, 169–186. [CrossRef]
32. Rubio-Manzano, C.; Lermanda, T.; Martínez-Araneda, C.; Vidal, C.; Segura, A. Teach me to play, gamer! Imitative learning in computer games via linguistic description of complex phenomena and decision trees. *Soft Comput.* **2023**, *27*, 3023–3035. [CrossRef]
33. Ganin, Y.; Bartunov, S.; Li, Y.; Keller, E.; Saliceti, S. Computer-aided design as language. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 5885–5897.
34. Wu, Y.; He, F.; Zhang, D.; Li, X. Service-oriented feature-based data exchange for cloud-based design and manufacturing. *IEEE Trans. Serv. Comput.* **2015**, *11*, 341–353. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.