DISEÑO E IMPLEMENTACIÓN DE UN LENGUAJE DE PROGRAMACIÓN DECLARATIVO PARA EL CONTROL DE AGENTES EN VIDEOJUEGOS

Universidad del Bío-Bío, Chile

Facultad Ciencias Empresariales

Departamento de Sistemas de Información

INGENIERÍA CIVIL EN INFORMÁTICA

By Christopher Cromer y Martín Araneda Acuña

DIRIGIDO POR CLEMENTE RUBIO-MANZANO

Enero 2023

Agradecimientos

From Chris,

I would like to thank my wife Elia. Her constant help and support is what has allowed me to go back to college and to achieve my goals and become the professional I am today. I owe her everything and am very lucky to have had someone on my side through this whole process during all these years of working, studying, and investigating.

I would also like to thank our thesis supervisor, Clemente Rubio-Manzano who inspired me to dig deeper into video game design and artificial intelligence. His support during this thesis was indispensable and helped us to bring this project to life.

De Martin,

Agradezco a mi Padre, por su constante apoyo, esfuerzo y sabiduría. A mi madre, por su cariño, paciencia y siempre la disposición a escucharme.

Por otro lado, quiero agradecer a nuestro profesor guía Clemente Rubio-Manzano, por darnos todo el apoyo necesario y tener siempre la disposición de ayudarnos para poder hacer de esta idea realidad.

Resumen

Los videojuegos se han convertido en el medio de entretenimiento más importante del mundo. Su diseño e implementación depende, en gran medida, de áreas como las Ciencias de la Computación o la Inteligencia Artificial ya que permiten dotarlos del realismo necesario para alcanzar niveles óptimos de inmersión tanto desde el punto de vista gráfico como de la jugabilidad. El uso de estas tecnologías ha permitido darle un lado más autónomo y humando, así como también la posibilidad de experimentar en bastantes aspectos, dado que los videojuegos contienen muchos subgéneros y por tanto, las elecciones son incontables.

La programación de la inteligencia artificial en los videojuegos puede resultar compleja debido a los conocimientos que se debe tener en la parte gráfica que van unidos estrechamente a la parte de comportamiento. Existe un gran interés en la comunidad científica en crear lenguajes de programación que permita separar la parte gráfica de la del comportamiento, esto es, programar ambas módulos de forma independiente.

En base a esto, el objetivo de este proyecto título ha sido investigar sobre el diseño e implementación de lenguajes declarativos aplicados al desarrollo de videojuegos. En particular, se ha creado un lenguaje de programación de tipo declarativo, donde sólo se indica qué se debe hacer, sin ir a niveles bajos de detalle. Todo esto es con la finalidad de que sea más afín a la lógica del ser humano y que también contribuya a que la implementación del agente sea más instintivo.

Para la creación del lenguaje, este ha sido escrito en C++ y también se utilizó LLVM, una tecnología que nos permitió generar y compilar nuestro lenguaje a uno máquina. Por otro lado, la parte lógica del lenguaje fue construida utilizando una base de conocimientos hecha en SQLite, que almacena todas los

elementos contenidos en las palabras claves implementadas, tales como acciones, hechos y reglas.

Los resultados de prueba obtenidos de la base de datos que almacena cada partida hecha por un jugador, sea humano o no, más el software de medición de rendimiento en R nos permitió elegir los mejores modelos estadísticos para que en un futuro contrastar estos valores con el agente y comprender que tan cercano es su comportamiento con el de un ser humano.

Finalmente, existen algunos aspectos importantes que quedan por terminar para que nuestro lenguaje de programación este completo, siendo estos el compilador, integración en el videojuego *Alai*, pruebas de rendimiento para evaluar su desempeño y por último un frontend que permita visualizar los resultados de las partidas.

Palabras Claves: Videojuegos, Inteligencia Artificial, Programación Declarativa, Base de Conocimiento, Compilación a Lenguaje Máquina

Índice general

1. Introducción **7**
- 1.1. Videojuegos y su Programación 7
- 1.2. Nuestra Investigación 10

2. Marco Teórico **12**
- 2.1. Compilación del Código Fuente 12
- 2.2. Motor de Videojuegos 13
- 2.3. Game I.A. 15
- 2.4. Lenguaje de Programación Compilado 17
- 2.5. Máquinas de Estados 17

3. Diseño e Implementación del Lenguaje **19**
- 3.1. Sintaxis 19
  - 3.1.1. Hechos 20
  - 3.1.2. Acciones 21
  - 3.1.3. Reglas 22
- 3.2. Semántica 23
- 3.3. Implementación del Lenguaje 23
  - 3.3.1. Arquitectura 23
  - 3.3.2. Incorporación del Lenguaje en el Motor de Videojuegos 26

4. Evaluación del Desempeño del Agente **28**
- 4.1. Sistema 28
  - 4.1.1. Monitor 28
  - 4.1.2. Servidor 28
  - 4.1.3. Estructura de Base de Datos 29

4.2. Análisis 32
4.2.1. Distribución Normal 32
4.2.2. Serie de Tiempo 34

Referencias **37**

Índice de figuras

2.1. Abstract Syntaxis Tree 13
2.2. Voxel Data Generation 14
2.3. Mesh Generation 15
2.4. Programación Lógica vs. Funcional 16
2.5. LLVM 17
2.6. Máquina de Estado Finito 18

3.1. Estructura Básica de una Palabra Clave Lógica 20
3.2. Estructura Básica de un Hecho 20
3.3. Estructura Básica de una Acción 21
3.4. Estructura Básica de una Regla 22
3.5. Diagrama Estructural del Compilador 24
3.6. Estructura del Base de Conocimiento 25
3.7. Videojuego Alai 27

4.1. Estructura del Base de Datos del Backend 30
4.2. Distribución Normal de Densidad de Probabilidad vs. Moneda . 32
4.3. Distribución Normal de Densidad de Probabilidad vs. Tiempo . 33
4.4. Serie de Tiempo de Monedas en un Partido 34

Índice de cuadros

3.1. Estructura del Base de Conocimiento 26

4.1. Estructura del Base de Datos del Backend 31

Capítulo 1

Introducción

1.1. Videojuegos y su Programación

Los videojuegos han superado al cine en cuota de mercado y se han posicionado como uno de los medios de entretenimiento más populares y complejos del mundo. Además, se han incorporado para apoyar al desarrollo en otras disciplinas como la educación o la investigación científica. Por ejemplo, en [1] se menciona que *“los videojuegos modernos se han convertido en una alternativa de calidad y de bajo coste para evaluar algoritmos de aprendizaje automático”*.

Una de las fases más importantes en el desarrollo de videojuego es el modelado y programación del comportamiento de los personajes. Estos personajes se mueven de forma automática y juegan un rol fundamental en la calidad final del videojuego; esto es, permiten juegos más desafiantes y divertidos [2].

El rol de los agentes tiene al menos dos dimensiones: cantidad y calidad. La cantidad se refiere a contar con un numero adecuado de actores que den la sensación al jugador de estar participando en el mundo real; la segunda es la habilidad de crear la ilusión de la credibilidad de comportamiento, directamente relacionada a la inteligencia artificial de los actores [3].

En la actualidad, los agentes se programan empleando técnicas deterministas como las máquinas de estado o los lenguajes de scripting. En la literatura actual se mencionan algunos desafíos y problemas abiertos relacionados a la programación de los agentes:

1. *”Desarrollar agentes autónomos que realicen tareas complejas es caro*

y consume mucho tiempo. La parte más costosa del desarrollo de estos agentes requiere extraer conocimiento de expertos humanos y adaptarlo al entorno” [4].

2. *”La I.A. de los videojuegos modernos todavía se basa en enfoques basados en reglas, y hasta ahora no se ha logrado desarrollar oponentes que resulten convincentes y sean similares a un humano.”* [5].

3. *“Además, mientras los jugadores se familiarizan con la mecánica del juego y mejoran sus habilidades e idean nuevas estrategias, los agentes no cambian y eventualmente se vuelven obsoletos"* [1].

El lenguaje de scripting es diversamente usando en muchas áreas, sin embargo, en el campo de la inteligencia artificial, se utiliza mucho por la rapidez de desarrollo dado que no es necesario compilarlo y eso reduce el tiempo de iteración cuando se programa. A este respecto, la programación del comportamiento de agentes en videojuegos se ha realizado desde diferentes puntos de vista. Los trabajos se pueden agrupar en dos categorías: (i) basados en el lenguaje de programación lógico Prolog; (ii) basados en el lenguaje de programación lógico GOLOG.

En [6], se describe y explica un lenguaje de programación lógico llamado GOLOG que permite implementar agentes de videojuego del estilo FPS (Disparos en Primera Persona) y enfrentarlos a NPCs (Non-Player Characters). El videojuego en cuestión, llamado ”Unreal Tournament 2004”, es de tipo multijugador, donde los jugadores humanos compiten para alcanzar objetivos. Los oponentes pueden ser tanto humanos o controlados por el computador. Originalmente, la toma de decisiones de los bots fue hecha usando un lenguaje orientado a objetos llamado ”UScript”, que es propio del juego y es de tipo script, con lógica basada en un *”state machine”*, constando de nueve estados controlados por el motor del juego. Sin embargo, el manejo de los bots se realizó en *”ReadyLog”*, un lenguaje basado en GOLOG, que permite el razonamiento basado en acciones. Con esta información se agregó una interfaz al motor del juego que le permite transmitir información importante relacionada al mapa, tal como items (munición, vida y armas), ubicación de los jugadores y estilo del ambiente en general. Por tanto, la información de estos elementos sera transmitida al bot si este es capaz de percibirlos, lo que da posibilidad a cambiar el comportamiento del bot dentro del framework.

Por otro lado, en el articulo [7] se explica la creación e implementación de un script basado en el lenguaje Prolog dentro de un juego estilo FPS (Disparos en Primera Persona), con la finalidad de crear diferentes tácticas de comportamiento en un equipo de bots compuesto por agentes. La idea fue inspirada principalmente en la observación de tácticas de equipo presentes en torneos reales de Counter-Strike. Por otro lado, la construcción de este framework está basado en un proyecto anterior hecho por los autores para crear scripts para bots. El pilar de esta investigación se construye en la premisa de diseñar el framework para utilizar un creador de mapas y así personalizar el comportamiento del bot para mapas nuevos, pues con esto se le entregaría al agente conocimiento necesario del ambiente que le rodea para adquirir una ventaja táctica frente a oponentes humanos. Nos centramos en estudiar dos aspecto: el desarrollo del razonamiento lógica del agente y las características de su I.A. Cada bot contiene dos pilas, una de acciones y otra de tareas. Estas siempre ejecutan el bloque que esta en el tope de la pila. Para que un bot sea capaz de cumplir una tarea de manera exitosa, hará ciertas acciones asociadas a esa tarea con tal de cumplirla. Algunas condiciones causarán que se agreguen o quiten acciones dentro de esta pila. Por tanto, cuando se complete una tarea o una acción, se borrará el bloque y luego el bot intentará ejecutar la acción o tarea siguiente en la pila.Uno de los mecanismos de razonamiento utilizado es uno llamado ”Razonamiento de reflejos”, el cual sucede en cada actualización del estado actual del juego, prestando gran apoyo cuando ocurren cambios repentinos en un nivel, como en el ambiente o cuando el bot este siendo atacado. En consecuencia, este mecanismo posee una característica poderosa para agregar nuevas tareas o limpiar la pila de tareas, adaptándose a cada situación.

Los trabajos estudiados tienen algunas características relevantes:

- En [6] el tipo de I.A. esta basada en objetivos y emplea técnicas de planificación. Con respecto a la planificación de camino y la detección de colisiones debemos mencionar que la interfaz entre el juego, junto con el motor y los bots no permite modificar los algoritmos de la planificación de los caminos ni de la detección de colisiones para permitir el cumplimiento de los objetivos. El agente solo recibe información si está en su campo de visión, por tanto, no es omnisciente y no tiene información suficiente que le permita obtener ventaja contra otros agentes o jugadores.

- En [7] el agente usado en este proyecto es de tipo racional basado en objetivos, con base en un árbol de decisión con instrucciones condicionales. En los trabajos previos hechos por el autor, se implementó una base de datos que contiene puntos de navegación, con el fin de calcular y planificar el camino que debe seguir al moverse el bot. También se usan los puntos de navegación para tomar decisiones racionales de naturaleza táctica. El bot adquiere total conocimiento del mapa, lo que le permite tener una ventaja táctica y así aumentar sus posibilidades para ganar la partida.

1.2. Nuestra Investigación

Dado lo anterior, tiene un gran beneficio implementar una I.A. empleando un lenguaje declarativo ya que no se indica el cómo se debe realizar el comportamiento desde un punto de gráfico, solo indicamos el qué debe realizar el agente, es decir, cómo se tiene que comportar independientemente de los aspectos gráficos. Esto último es importante ya que conseguimos que la tarea de programación se aproxime a la lógica humana y el programador pueda implementar el comportamiento de los agentes de una forma más intuitiva y empleando reglas simples. En este sentido, uno de los lenguajes declarativos más importantes ha sido Prolog demostrando sus buenas propiedades en este ámbito. Por lo tanto, vamos a crear un lenguaje declarativo inspirado en Prolog que se pueda utilizar para programar el comportamiento de los personajes de un videojuego que tienen, al menos, que realizar las siguientes acciones: i) superar obstáculos, con la opción de saltar con velocidad; ii) recoger items que se agregan a la puntuación final; iii) evitar enemigos.

En este proyecto hemos desarrollado un lenguaje de programación de tipo declarativo inspirado en Prolog que permite modelar el comportamiento de los agentes de un videojuego empleando reglas lógicas.

Para llevar a cabo el objetivo anterior se han tenido que realizar los las siguientes tareas: i) Se revisó la bibliografía sobre Prolog, el motor Godot y programación de videojuegos; Se analizó la información recopilada de la bibliografía investigada; ii) Se creó el lenguaje de programación de tipo declarativo inspirado en Prolog, de nombre *"Obelisk"*; iii) Se implementó una inteligencia artificial basada en el lenguaje anteriormente creado en un videojuego; iv) Se evalúo del desempeño del lenguaje inventado, con verificación del cumplimiento exitoso de la superación de obstáculos por parte del agente.

El resto de la memoria se organiza como sigue: en el Capítulo 2 (Marco Teórico) explicaremos las herramientas que se van utilizar, análisis de rendimiento de software similares y descripción de conceptos abarcados en el proyecto; en el Capitulo 3 (Diseño e Implementación del lenguaje) presentamos el diseño del lenguaje de programación definiendo la estructura del lenguaje de programación *Obelisk*, incluyendo su sintaxis y semasiología respectiva. También se detalla el desarrollo del aspecto funcional del lenguaje, tales como el funcionamiento del compilador, base de conocimiento y librerías. Asimismo se describe el proceso para utilizar el lenguaje *Obelisk* con software externos; en el Capítulo 4 (Evaluación y Desempeño del Agente) realizamos una explicación de los elementos desarrollados para medir el rendimiento del agente en el videojuego *Alai*; Por ultimo, les explicamos las Conclusiones del proyecto y el trabajo futuro.

Capítulo 2

Marco Teórico

Este capítulo tiene como propósito describir los conceptos más importantes en el desarrollo del lenguaje de programación para control de la Inteligencia Artificial, así como también su implementación en un juego estilo plataformas.

2.1. Compilación del Código Fuente

La compilación del código fuente es el proceso en el cual se transforma un lenguaje de alto nivel a uno que se puede comprender una máquina. Este procedimiento esta compuesto de varios pasos, los que serán descritos a continuación.

- **Lexer:** El trabajo de un lexer es leer texto y hacer un análisis léxico para encontrar los símbolos y los tokens reconocidos. Un símbolo en el código, por ejemplo #, representa un comentario. Por otro lado, un token es un conjunto de símbolos que contiene un significado, es decir, puede ser el nombre de un variable o función, un número, una palabra clave como *if* o *while*, etc.

- **Parser:** El parser es responsable de interpretar y llevar a cabo operaciones basadas en los tokens y símbolos recibidos desde el lexer. Usando todos los tokens y símbolos encontrados, el parser construye un *Abstract Syntax Tree*.

- ***Abstract Syntax Tree(AST):*** El AST se utiliza para generar código intermedio(IR). Por otro lado, es útil para controlar el flujo de la lógica y generar el código de objeto en archivos de objeto que corresponde.

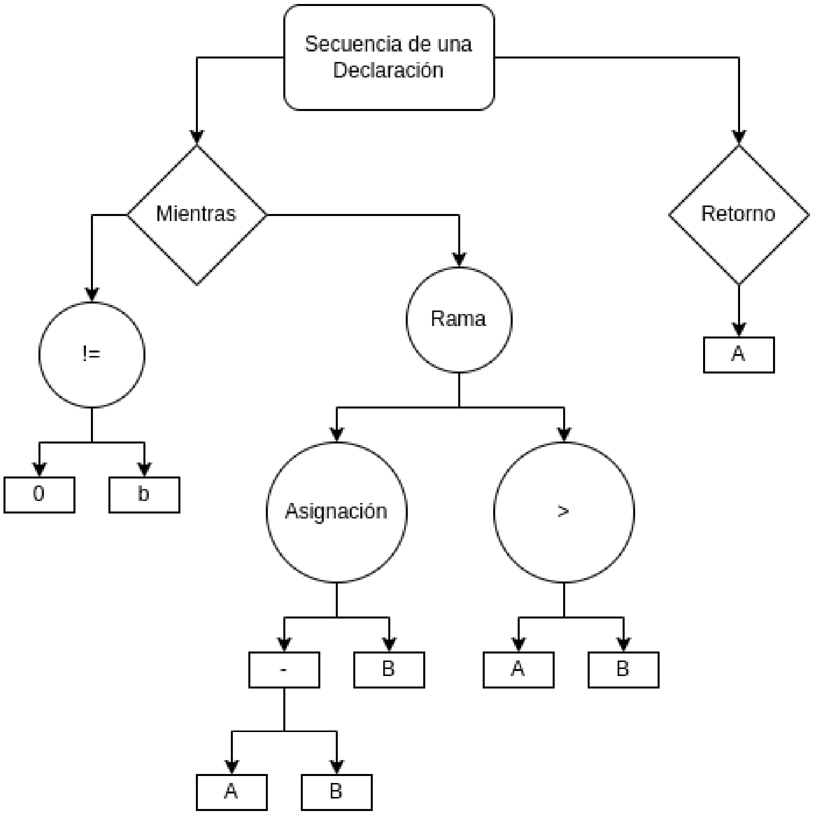

Figura 2.1: Abstract Syntaxis Tree

- **Linker:** El propósito del linker es tomar los archivos de objeto, unirlos y convertirlos en un ejecutable y/o librería que la máquina puede utilizar. Los binarios del ejecutable y librería contienen código de máquina basado en la arquitectura del sistema, por ejemplo x86_64 en caso de GNU/Linux de 64bit, que es usualmente encontrado en computadores de escritorio o notebooks usados hoy en día.

2.2. Motor de Videojuegos

Un motor de videojuegos es un conjunto de rutinas de programación que tiene como objetivo proveer facilidad en la utilización de elementos gráficos, sonidos, físicas, redes y varios otros aspectos que ayudan en el desarrollo de un videojuego. El motor que se utilizará en este proyecto tiene por nombre ”Godot”. [8, 9, 10]

El motor Godot tiene un gran valor en el desarrollo del proyecto, principalmente por su capacidad para implementar arte de píxel, en comparación con motores similares como Unity o Unreal, porque en Godot las distancias entre los elementos es medido en píxeles mientras que otros motores utilizan metros, lo que facilita en gran medida la eficacia al trabajar en juegos 2D, que usualmente para este género de videojuegos se usan píxeles y no metros.

Otro factor importante es que Godot posee un elemento llamado ”GDNative”. GDNative es una API que permite usar lenguajes de programación como C, C++ o Rust, siendo estos capaces de compilar código a lenguaje máquina. Lo anterior posee especial importancia para el proyecto, puesto que permite implementar otro lenguaje de programación que hará interacción con GDNative. [11]

También la API permite que la I.A tome decisiones con mayor rapidez dado que el código objeto de máquina se ejecuta con mayor velocidad que un lenguaje interpretado o que se compila a bytecode, como se puede notar en los siguientes gráficos que comparan GDScript(lenguaje interpretado similar a Python), C# y C++.

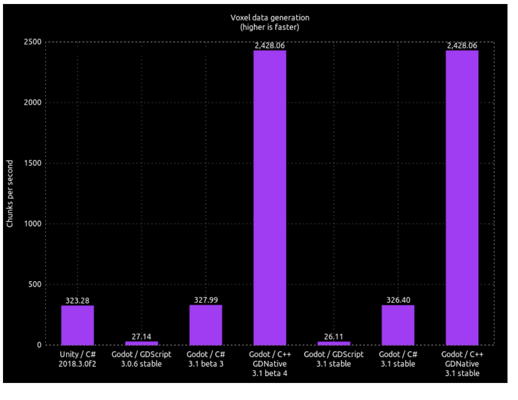

Figura 2.2: Voxel Data Generation
[12]

En el caso de generación de datos voxel, existe un aumento de 8,846.43 % en su rendimiento comparando GDNative con GDScript y un aumento de 651.07 % comparando GDNative con C#.

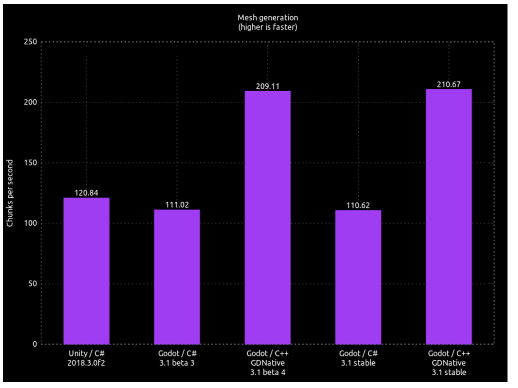

Figura 2.3: Mesh Generation
[12]

En la prueba de generación de mallas, GDNative tiene un aumento de 651.07 % en su rendimiento comparado con C#.

2.3. Game I.A.

La Inteligencia Artificial (I.A) es una disciplina científica relativamente nueva que nació de los avances en las ciencias de la computación y/o la informática. De forma intuitiva, la I.A puede entenderse como la capacidad que tiene una máquina para tomar decisiones de forma autónoma a partir de los estímulos que recibe del exterior. Una de las primeras veces que se hablo de la I.A como disciplina fue en el año 1956 cuando John McCarthy, importante matemático e informático, la definió como *"La ciencia e ingeniería de hacer máquinas inteligentes, con mayor énfasis en programas computacionales inteligentes"* [13].

Actualmente, la I.A puede considerarse también una tecnología moderna que ha ido evolucionando y se ha ido incorporando a nuestra vida cotidiana. Un ejemplo bastante importante es el reconocimiento facial introducido en teléfonos inteligentes, lo que permite agregar ciertos efectos a las imágenes, ajustar automáticamente el enfoque o balancear la exposición en condiciones de poca luz.

El lenguaje responsable que se encargará de la toma de decisiones por parte de la I.A será de carácter propio e inspirado en Prolog, un lenguaje declarativo y lógico. [14]

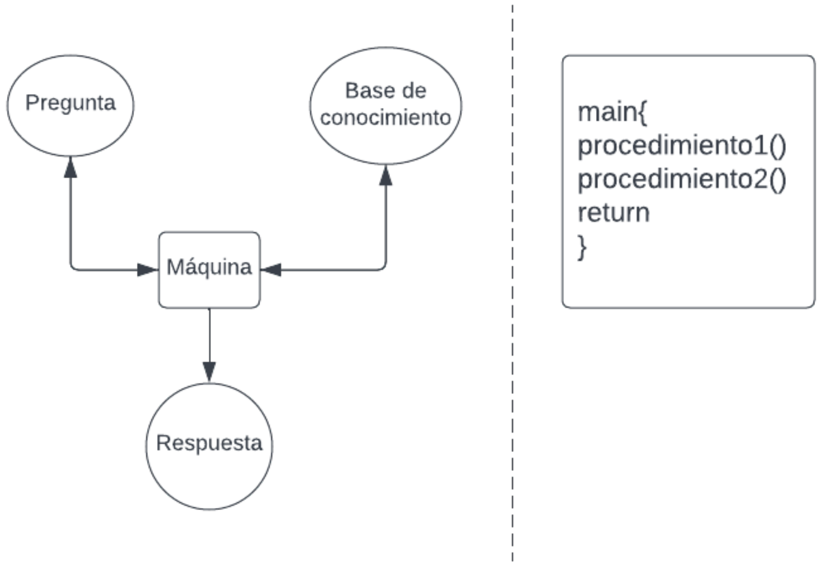

Figura 2.4: Programación Lógica vs. Funcional

Al hablar de programación funcional, la idea principal es que todos los elementos sean funciones y estos sean capaces de poder ejecutarse de manera secuencial, por lo cual se utiliza la lógica paso por paso para resolver el problema. Por otro lado, la programación lógica utiliza una base de conocimiento para hacer preguntas y recibir respuestas que se utilizarán para resolver el problema [15].

El lenguaje tipo Prolog debe tener 3 elementos importantes para funcionar:

1. **Hechos:** Son datos verdaderos, como por ejemplo ”el español es una idioma”.

2. **Reglas:** Son cláusulas condicionales que conectan los hechos. Un ejemplo es: ”si vives en Chile hablas el español”.

3. **Preguntas:** Son necesarias para tener una respuesta por parte de la base de conocimiento. Un ejemplo sería ”¿El español es una idioma?”.

2.4. Lenguaje de Programación Compilado

El proyecto LLVM es un conjunto de tecnologías de compilador y toolchain, el cual permite crear un lenguaje propio de programación [16].

Figura 2.5: LLVM

LLVM consiste de varios sub-proyectos, pero el que será utilizado principalmente es ”LLVM Core”. Este sub-proyecto contiene un optimizador y generador de código, siendo este último llamado LLVM Intermediate Representation(LLVM IR). La funcionalidad es similar a una Virtual Machine de bytecode que es portátil y se puede correr en cualquier sistema que posee el LLVM.

Otro aspecto importante de LLVM es que se puede utilizar el LLVM IR, que fue generado anteriormente y así compilarlo a lenguaje máquina para la arquitectura computacional que se desee. Luego, el código objeto generado se puede utilizar con un linker para crear librerías y binarios, lo que tendrá importancia al querer integrar el código que compila nuestra compilador en el motor Godot.

2.5. Máquinas de Estados

Una máquina de estados se utiliza para controlar el estado de un objeto que tiene una ramificación compleja y estados mutables. [17] La máquina de estados finita es parte de una rama de las Ciencia de la Computación llamado *”teoría de autómata”*, la cual incluye la famosa máquina de Turing. La máquina de estados es usualmente usada en programación de I.A., videojuegos y en la creación de compiladores de código, sin embargo fuera de estas 3 áreas, posee muy poca adopción y uso.

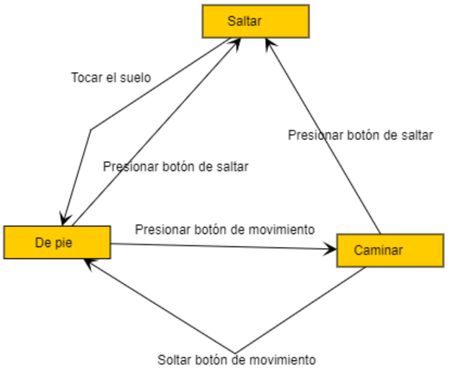

Figura 2.6: Máquina de Estado Finito

La máquina de estado finita posee un conjunto concreto de estados en la que es capaz de estar, como ”saltar” o ”caminar”, la cual también posee una restricción importante que consiste en que esta máquina solo puede presentar un solo estado en un instante de tiempo, es decir, el jugador no es capaz de saltar y caminar en un mismo momento.

El funcionamiento de la máquina de estados finita consiste en una secuencia de entradas y eventos que son enviadas a esta, los cuales se usan para cambiar entre estados. Cada máquina tiene un conjunto de transiciones y cada una de ellas está asociada con una entrada o un evento, lo que finalmente apunta a otro estado. Por tanto, cuando llega la entrada o evento y este coincide con una transición dentro del estado actual, la máquina cambiará al estado al que apunta la transición, por lo que si un jugador se encuentra ubicado en el estado ”caminar”, se puede presionar el botón de saltar para cambiar al estado de ”saltar”.

Capítulo 3

Diseño e Implementación del Lenguaje

Nuestro lenguaje de programación, de nombre *"Obelisk"*, tiene como pilar fundamental en su diseño el paradigma declarativo, es decir, posee un fuerte enfoque en definir el resultado al que se desea llegar, en contraste a describir el flujo de control para obtener la solución. En consecuencia, el idioma posee por naturaleza un nivel de abstracción alto.

Por otro lado, las palabras claves utilizadas están inspiradas en el lenguaje Prolog, el cual posee hechos y reglas. Esto es para luego agregar una palabra propia de las acciones a tomar, dependiendo de los hechos y las reglas.

Debido a esto, se abren posibilidades para ser perfeccionado con facilidad ya que al escribir código no es necesario determinar el procedimiento al cual se desea llegar, dándole la poderosa cualidad de ser bastante flexible.

"Obelisk" contiene palabras claves que cuando sean utilizadas permitan que ciertas operaciones lógicas sean declaradas y ejecutadas, y por lo tanto reconocidas para su posterior uso, las cuales serán descritas a continuación.

3.1. Sintaxis

La sintaxis del lenguaje fue diseñado teniendo en mente el asemejarse al lenguaje humano, específicamente el idioma inglés. La razón es debido a que el inglés es utilizado en todas partes del mundo y es muy fácil de aprender y usar.

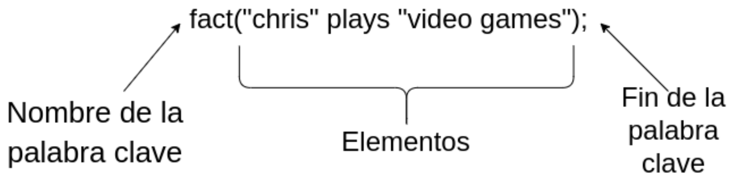

Figura 3.1: Estructura Básica de una Palabra Clave Lógica

La estructura de la sintaxis del lenguaje de programación *”Obelisk”* se desglosa en tres partes. El nombre, los elementos y el punto y coma (;) componen en su totalidad lo que nuestro idioma comprende como una palabra clave del lenguaje. Cabe señalar que las palabras claves del lenguaje están restringidas solo en el idioma inglés.

En este caso, el idioma posee tres palabras claves implementadas, las cuales son las acciones, los hechos y las reglas. Cada una de ellas tiene un orden ligeramente similar y difieren sólo en el contenido de los elementos, el cual se describirá a continuación.

3.1.1. Hechos

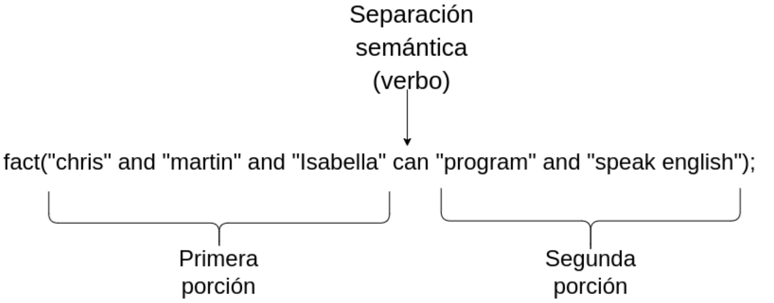

Figura 3.2: Estructura Básica de un Hecho

Los hechos tienen su contenido dividido en dos porciones. Sin embargo, la separación semántica es hecha por un verbo.

Por lo tanto, tenemos:

- 1°ra Entidad(es) (primera porción): Deben estar en comillas dobles, pueden ser más de uno y deben estar separados por la conjunción *and*. Ej. *”chris and martin...etc”*.

- Verbo: Representa la división semántica y la relación entre las dos porciones. Ej. *"can"*

- 2°da Entidad(es) (segunda porción): Deben estar en comillas dobles, pueden ser más de uno y deben estar separados por la conjunción *and*. Ej. *"program and speak english...etc"*

3.1.2. Acciones

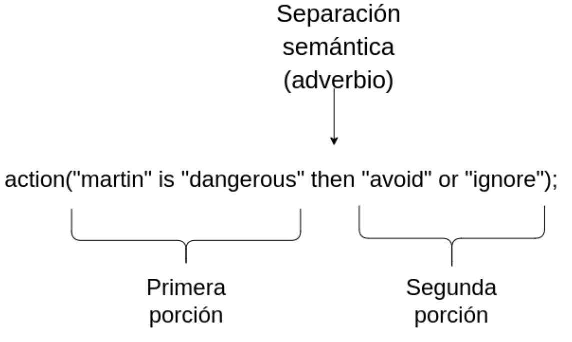

Figura 3.3: Estructura Básica de una Acción

La sintaxis de las acciones destaca principalmente, como las demás palabras clave, en los parámetros. El contenido esta compuesto por dos porciones, divididos por el adverbio. Usando como referencia el ejemplo anterior, la estructura es:

- 1°ra Entidad (primera porción): Debe estar en comillas dobles. Ej. *"martin"*.

- Verbo: Representa la relación entre las dos entidades de la primera porción. Ej. *"is"*.

- 2°da Entidad (primera porción): Debe estar en comillas dobles. Ej. *"dangerous"*.

- Separación semántica: El adverbio describe la implicación que tiene la primera porción sobre la segunda. . Ej. *"then"*.

- Sugerencia verdadera(segunda porción): Presenta la opción a sugerir en caso en que el hecho sea verdad. Ej. *"avoid"*.

- Sugerencia falsa (segunda porción): Manifiesta la elección a suscitar si es que el hecho sea falso. Ej. *"ignore"*.

3.1.3. Reglas

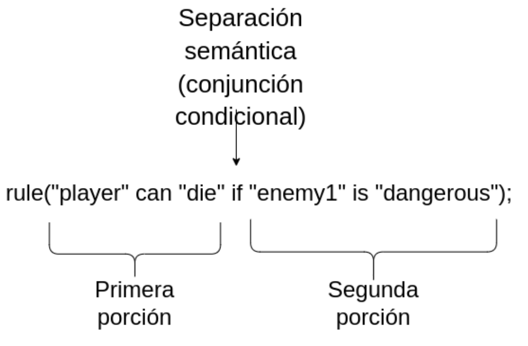

Figura 3.4: Estructura Básica de una Regla

La sintaxis de las reglas, como en el resto de las palabras claves, posee una división semántica que divide el contenido de los elementos en dos fragmentos. Cabe mencionar que las reglas deben contener si o si dos entidades por cada porción, separados por un verbo. Por tanto, su estructura es:

- 1°era Entidad (primera porción): Debe estar entre comillas dobles. Ej. *"player"*.
- Verbo (primera porción): Es la relación entre ambas entidades de la primera porción.
- 2°da Entidad (primera porción): Debe estar entre comillas dobles. Ej. *"die"*.
- Conjunción condicional: Indica la división semántica entre las dos porciones. En español expresa "cuando, a ser, etc.". Ej. *"if"*.
- 1°ra Entidad (segunda porción): Debe estar entre comillas dobles. Ej. *"enemy1"*.
- Verbo (primera porción): Es la relación entre ambas entidades de la segunda porción.

- 2°da Entidad (segunda porción): Debe estar entre comillas dobles. Ej. *"dangerous"*.

3.2. Semántica

La semasiología de *Obelisk* es en base a las palabras claves contenidas en el lenguaje y en consecuencia, describen la lógica y funcionamiento que tendrán.

Cabe mencionar que cada vez que se agreguen nuevas palabras claves, quedará un registro de estos en la base de conocimientos, el cual almacena toda la información contenida en el lenguaje. Este último será explicado con detalle más adelante.

Finalmente, nuestro lenguaje posee tres palabras claves y su funcionamiento será descrito a continuación:

- **Hechos:** Los hechos, como su nombre indica, describen una afirmación en referencia a la entidad declarada en los elementos de la palabra clave. Es importante mencionar que si un hecho no esta presente en la base de conocimiento, se deja a entender que por defecto su valor booleano es *false*.

- **Reglas:** Las reglas se utilizan para crear condiciones y el resultado de estas dependen fundamentalmente de los hechos. Las reglas se utilizan para crear condiciones y el resultado de estas dependen fundamentalmente de los hechos. Esto quiere decir que una regla tendrá un valor booleano *true* si existe el hecho que respalde tal regla. En caso contrario la regla quedará con valor *false*.

- **Acciones:** Finalmente, las acciones definen que acto(s) se realizará(n) dependiendo si el hecho al cual alude tenga valor booleano *true*.

3.3. Implementación del Lenguaje

3.3.1. Arquitectura

La arquitectura del lenguaje esta formada por tres módulos: compilador, base de conocimiento y librería.

- **Compilador:** El compilador de Obelisk tiene como propósito leer el código fuente usando un *Lexer*. Luego se utiliza el *Parser* para analizar y aplicar operaciones en base a los tokens que encontró el *Lexer*. Tras esto, se crea una Base de Conocimiento, el cual posee la característica de poder consultar su información utilizando un software externo. Finalmente, el código relevante se transforma en código intermedio(IR) que luego será transformado en binario.

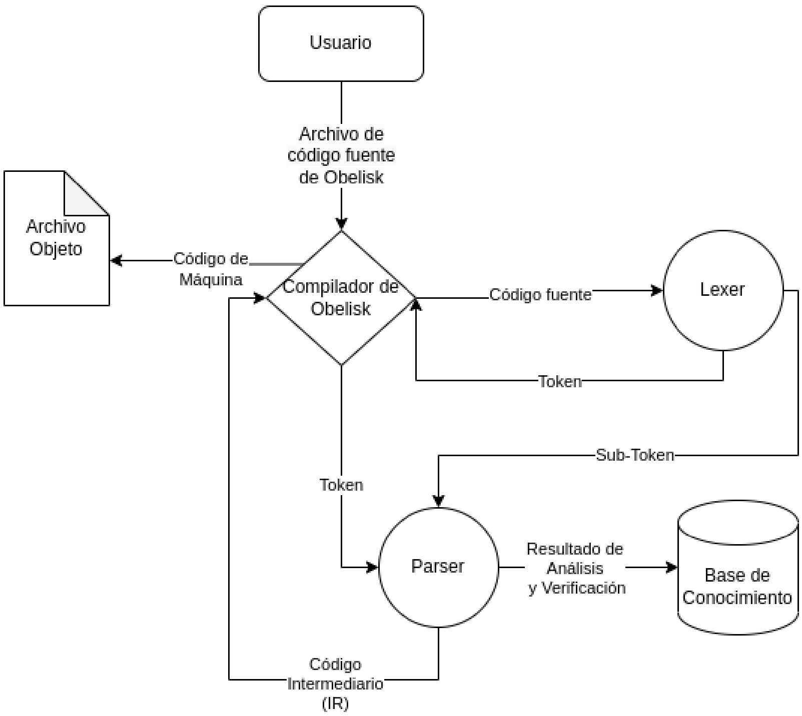

Figura 3.5: Diagrama Estructural del Compilador

- **Base de Conocimiento:** La Base de Conocimiento es donde se almacena toda la lógica proveniente de los hechos, reglas y acciones. Su implementación fue hecha utilizando SQLite, el cual permite hacer consultas a la base de conocimiento utilizando el lenguaje SQL.

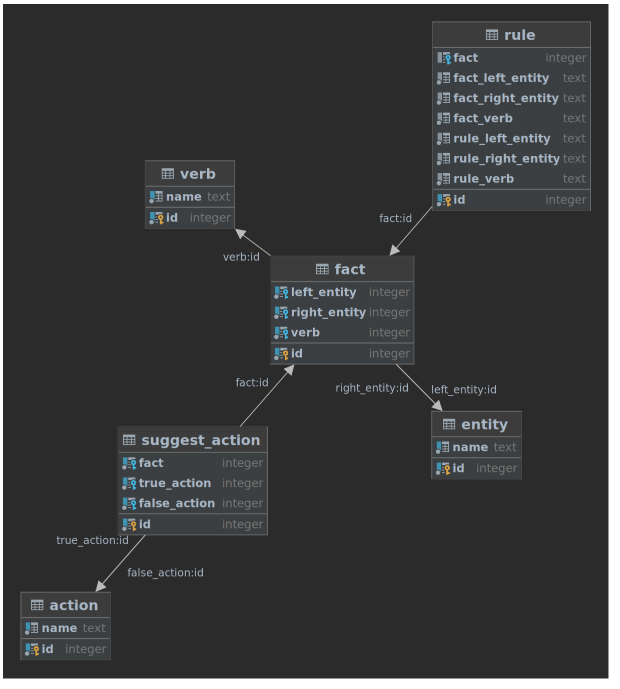

Figura 3.6: Estructura del Base de Conocimiento

La estructura de la base de conocimientos consiste en seis tablas principales las cuales se describen en el siguiente cuadro:

Nombre de Tabla	Propósito
entity	La tabla entity es usada para almacenar los nombres de las entidades presentes en los hechos, reglas y acciones.
verb	Se utiliza la tabla verb para almacenar los nombres de los verbos usados en los hechos y reglas.
action	La tabla action almacena los nombres de las posibles acciones que se pueden tomar.
fact	La tabla fact tiene los hechos verdaderos. Si existe una fila en esta tabla, la relación entre las dos entidades es verdadera.
rule	La tabla rule contiene reglas. Si una regla resulta ser verdad, se inserta el hecho en la tabla fact.
suggest_action	La tabla suggest_action contiene las dos posibles acciones que se pueden tomar dependiendo si el hecho es verdadero o falso.

Cuadro 3.1: Estructura del Base de Conocimiento

- **Librería:** La librería de Obelisk permite interactuar y consultar a la Base de Conocimiento de Obelisk, con el uso de un software externo. Hay dos tipos de datos que puede devolver la consulta. Un string que representa la acción a tomar o un numero entre 0 y 1 que representa su valor de verdad.

3.3.2. Incorporación del Lenguaje en el Motor de Videojuegos

Para incorporar el lenguaje en un software externo, como por ejemplo nuestro videojuego *Alai*, se utiliza la librería de Obelisk, la cual permite hacer consultas a la base de conocimientos y tomar acciones respectivamente.

Tomando en cuenta la utilización de un software *Third-Party*, cuando uno de estos utilice el idioma *Obelisk*, se hacen consultas a la base de conocimientos, la cual maneja la lógica proveniente de las palabras claves implementadas. Es importante mencionar que la base de conocimientos es compilada antes de ejecutar cualquier consulta a esta.

Después de esto, son dos los posibles resultados que pueden tener las consultas, los cuales son las acciones que debe tomar el agente o un valor numérico que represente su resultado y que se puede interpretarse en el software externo.

Basado en las decisiones elegidas en relación a la consulta de la base de conocimiento, es la responsabilidad del software poner la acción en marcha. Esto se puede llevar a cabo utilizando varias técnicas de inteligencia artificial tales como A*, máquina de estado, planificación de acciones orientados a metas, etc.

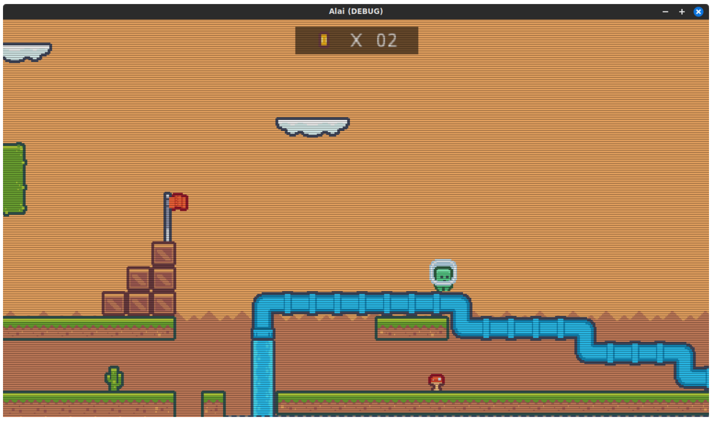

Figura 3.7: Videojuego Alai

Capítulo 4

Evaluación del Desempeño del Agente

La evaluación del agente tiene como propósito estimar el correcto comportamiento de la inteligencia artificial dentro del videojuego, como por ejemplo la cantidad de monedas recolectadas, veces que murió, etc.

4.1. Sistema

4.1.1. Monitor

Desarrollamos un monitor hecho en el lenguaje GDScript, que es parte del motor de videojuegos Godot y que graba todo la sesión del agente y/o jugadores. La grabación del partido es basado en el ciclo de update del motor Godot, es decir que si la pantalla del jugador tiene 60Hz como su taza de refresco, se grabará 60 cuadros de información cada segundo. Al terminar un partido toda esta información se envía a un servidor donde se procesa y guarda para futuro análisis. Cabe mencionar que se envía todo los datos en formato JSON a un servidor de REST.

4.1.2. Servidor

El servidor fue programado en el lenguaje Go. La razón es por su alto nivel de rendimiento y bajo nivel de uso de recursos, tales como CPU y RAM. Esto nos permite recibir muchos más datos simultáneamente de varias partidas al mismo tiempo sin complicaciones. También el servidor provee una API de tipo REST con varios endpoints que permiten almacenar las partidas del juego y consultar la información guardada en ello para futuro estudio.

4.1.3. Estructura de Base de Datos

Todo la información de las partidas jugadas se almacenan en un base de datos MySQL.

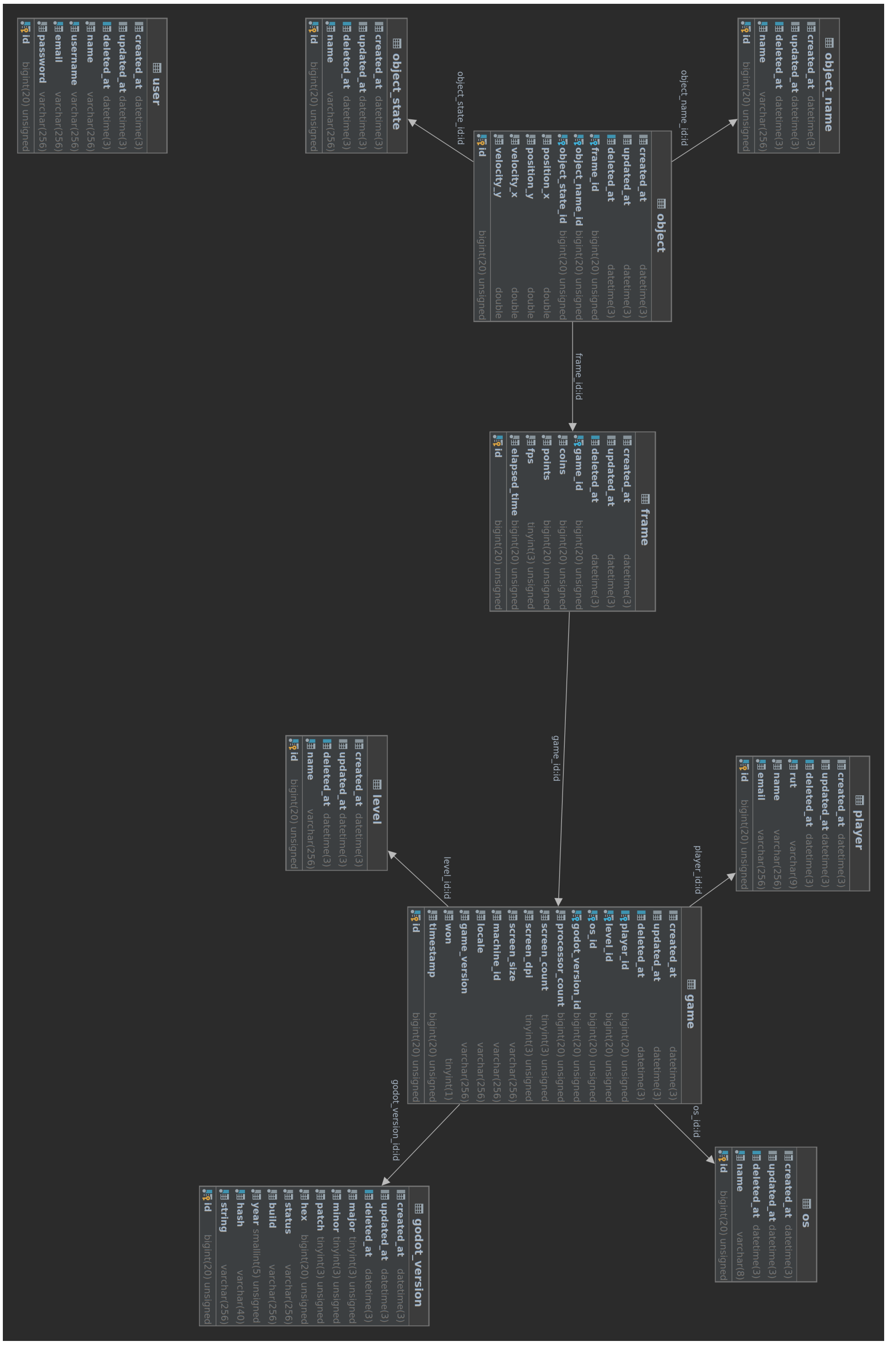

Figura 4.1: Estructura del Base de Datos del Backend

Nombre de Tabla	Propósito
frame	La tabla frame se utiliza para almacenar la cantidad de puntos, monedas, tiempo que pasó y posiciones de los objetos en el mundo cada cuadro que dibuja el juego.
game	La tabla game contiene información sobre el partido y el equipo que corre tal partido, incluyendo sistema operativo dimensiones de la pantalla y datos similares.
godot_version	La tabla godot_version se usa para almacenar las versiones de Godot que han corrido un partido del juego.
level	La tabla level es una tabla de parametros que contiene los nombres de los niveles que se puede jugar.
object	La tabla object se utiliza para almacenar el estado, posición y velocidad de un objeto en un cuadro especifico del partido.
object_name	La tabla object_name es una tabla de parámetros que contiene los nombres de todos los objetos que existen en un partido del juego.
object_state	La tabla object_state tiene todos los nombres de los estados de un objeto.
os	La tabla os es una tabla de parámetros que contiene los posibles sistemas operativos que pueden jugar el juego.
player	La tabla player es una tabla opcional donde se puede almacenar los datos de la persona que está jugando un partido para poder hacer un análisis sin ser anónima.
user	La tabla user contiene los usuarios y contraseñas de los personas que tienen permiso hacer consultas al API para obtener los datos y usarlos.

Cuadro 4.1: Estructura del Base de Datos del Backend

4.2. Análisis

Son muchos los tipos de análisis que son posibles con los datos que podemos recolectar de nuestro agente. Por tanto, para facilitar el estudio, utilizamos el lenguaje de programación R para calcular todas las estadísticas de las partidas. El idioma R genera gráficos basados en los datos y formulas para facilitar la comprensión del comportamiento del agente/jugador.

4.2.1. Distribución Normal

En pocas palabras, el proceso de uso de la distribución de probabilidad normal es básicamente analizar el comportamiento de varias partidas dentro del juego para luego calcular las probabilidades de que un evento ocurra.

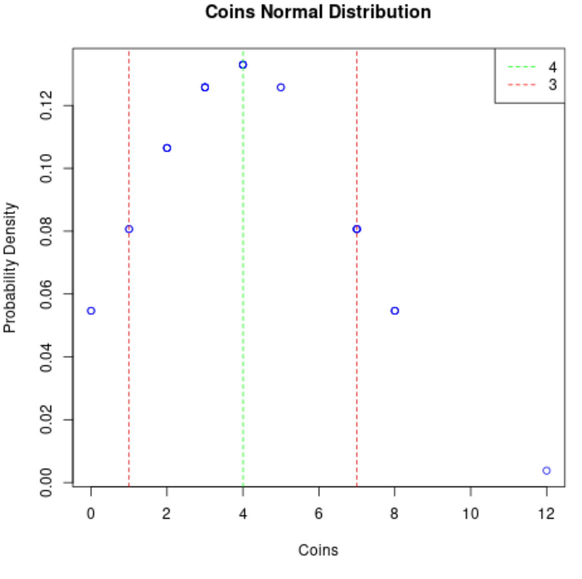

Figura 4.2: Distribución Normal de Densidad de Probabilidad vs. Moneda

En la figura anterior podemos ver en el eje X la cantidad de monedas obtenidas por el agente y/o jugador. En el eje Y tenemos la densidad de probabilidad. La densidad de probabilidad se puede interpretar como la probabilidad relativa en que un valor de un variable al azar seria igual a la muestra correspondiente.

La linea verde es el valor medio representado por la formula $\bar{X} = \frac{\Sigma_{i=1}^{n} X_i}{n}$ donde $\bar{X}$ es el valor medio, n es la cantidad de partidas y $\Sigma_{i=1}^{n} X_i$ es la sumatoria de los monedas en cada partida. Las lineas rojas se llaman desviación estándar. Para obtener la desviación estándar el primer paso es obtener la varianza que es representado por la formula $\sigma^2 = \frac{\Sigma_{i=1}^{n}(X_i)^2}{n}$ donde σ^2 es la varianza. Finalmente obtenemos la desviación estándar con la formula $\sigma = \sqrt{\sigma^2}$ donde σ es la desviación estándar.

De 9 partidas del juego, tuvimos un valor medio de 4 monedas representadas en la linea verde. Al calcular la desviación estándar obtuvimos el valor 3, por lo tanto la primera linea roja se coloca en 1 moneda porque 4 menos 3 es 1. Y en el caso de la segunda linea roja 4 mas 3 es 7.

Si la densidad de probabilidad en un punto X es muy grande, eso significa que un valor de un variable es probablemente cerca el punto X. En el caso de la figura 4.2, se puede interpretar eso en que es muy probable que el valor de un variable sería entre 1 a 7 acercando a 4.

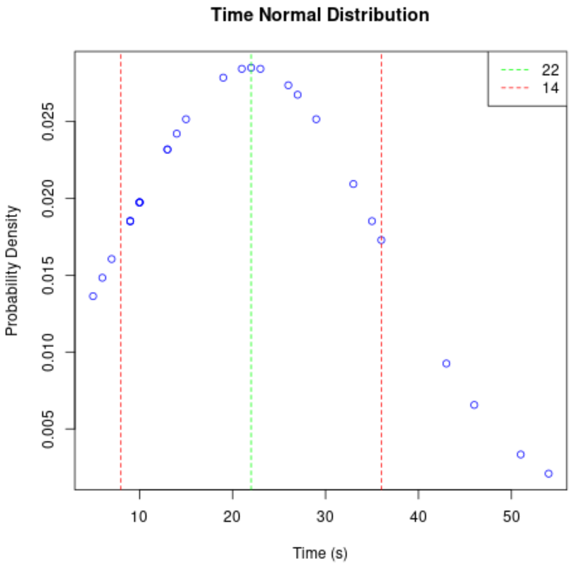

Figura 4.3: Distribución Normal de Densidad de Probabilidad vs. Tiempo

En la figura anterior analizamos 28 partidas con un valor medio de 22 segundos y un desviación estándar de 14 segundos. Podemos concluir que, en el valor de la variable, es bastante probable que una partida dure entre 8 segundos y 36 segundos, con un valor muy cercano a 22 segundos.

4.2.2. Serie de Tiempo

Utilizamos la serie de tiempo para analizar el comportamiento del agente durante una partida. Eso nos permite entender mejor sus decisiones y cuanto demoró en lograr los objetivos.

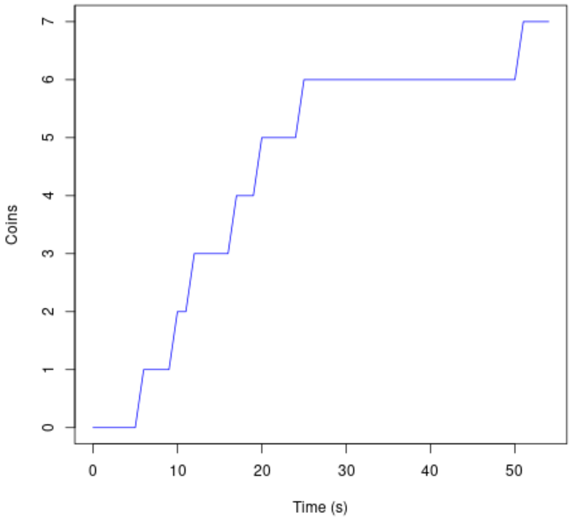

Figura 4.4: Serie de Tiempo de Monedas en un Partido

En la figura 4.4 analizamos una partida de un jugador humano que demoró 58 segundos y recolectó 7 monedas. En el eje X tenemos el tiempo medido en segundos, y en el eje Y tenemos la cantidad de monedas que tiene el agente y/o jugador en cualquier momento.

La mejor forma de utilizar un gráfico de serie tiempo en el análisis del agente y/o jugador es compararlo con otras partidas para ver que tan cerca el gráfico es con los demás. Idealmente el agente bien desarrollado debe demorar menos tiempo en completar sus objetivos y estar similar a un partido de jugador humano por cantidad de monedas obtenidas y en el tiempo que demoró en llegar a la meta final.

Conclusiones y Trabajo Futuro

En base al desarrollo del lenguaje, hemos probado con éxito el funcionamiento básico de la palabra clave lógica ”hechos”, llevando los elementos contenidos en el último a la base de conocimientos, siendo así guardados apropiadamente para futuras consultas de ser necesario.

Con respecto al videojuego *Alai*, este funciona en su totalidad y con todas las mecánicas respectivas que usaremos para nuestro agente, como movimiento en dos dimensiones, saltar, doble salto, recolección de monedas, meta para finalizar el nivel y recolección de datos por cada partida.

Por otro lado, en el área de pruebas de desempeño, hemos realizado satisfactoriamente pruebas con datos de un jugador real utilizando nuestro software estadístico escrito en R, dando así paso a aplicar modelos descriptivos, como distribución normal y serie de tiempo, a nuestro agente y compararlo con un ser humano.

Además, el servidor backend que almacenará los datos recopilados del juego *Alai* está completo, para en un futuro ser consultados por el frontend y visualizarlos en un formato presentable.

Sin embargo, tenemos varios elementos que proponemos desarrollar en un futuro, con la intención de complementar nuestro lenguaje de programación y su funcionalidad. Estos ya poseen una base y solo requieren su finalización.

- Perfeccionar nuestro lenguaje de programación para que sea *Touring-complete*.

- Análisis del comportamiento del agente dentro de cualquier software, utilizando los datos obtenidos del videojuego.

- Pasar la lógica de consultas proveniente de la base de conocimientos y así utilizar la GPU, con librerías tales como CUDA de Nvidia.

- Implementar lógica difusa en los hechos, reglas y acciones.

Referencias

[1] Quentin Gemine y col. «Imitative learning for real-time strategy games». En: *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2012, págs. 424-429.

[2] Shu Feng y Ah-Hwee Tan. «Towards autonomous behavior learning of non-player characters in games». En: *Expert Systems with Applications* 56 (2016), págs. 89-99.

[3] Igor Borovikov y col. «Towards interactive training of non-player characters in video games». En: *arXiv preprint arXiv:1906.00535* (2019).

[4] Michael Van Lent y John Laird. «Learning hierarchical performance knowledge by observation». En: *ICML*. 1999, págs. 229-238.

[5] Christian Thurau y col. «Bayesian imitation learning in game characters». En: *International journal of intelligent systems technologies and applications* 2.2-3 (2007), págs. 284-295.

[6] «Controlling Unreal Tournament 2004 Bots with the logic-based action language Golog / Jacobs, Stefan ; Ferrein, Alexander ; Lakemeyer, Gerhard». En: *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE).* (2011), págs. 151-152.

[7] Grzegorz Jaśkiewicz. «Prolog-Scripted Tactics Negotiation and Coordinated Team Actions for Counter-Strike Game Bots». En: *IEEE Transactions on Computational Intelligence and AI in Games* 8.1 (2016), págs. 82-88. DOI: 10.1109/TCIAIG.2014.2331972.

[8] David Vallejo y Cleto Martín. *Desarrollo de Videojuegos: Un Enfoque Práctico*. CreateSpace, 2015. ISBN: 978-1-51730-955-8.

[9] Ernest Adams y Joris Dormans. *Game Mechanics: Advanced Game Design*. New Riders, 2012. ISBN: 0-32182-027-4.

[10] Chris Bradfield. *Godot Engine Game Development Projects*. Packt, 2018. ISBN: 978-1-78883-150-5.

[11] Ariel Manzur y George Marques. *Sams Teach Yourself, Godot Engine Game Development in 24 Hours*. Pearson, 2018. ISBN: 0-13483-509-3.

[12] Royal Donut Games. *CPU Voxel Benchmarks of Most Popular Languages in Godot*. Abr. de 2022. URL: `http://www.royaldonut.games/2019/03/29/cpu-voxel-benchmarks-of-most-popular-languages-in-godot/`.

[13] John McCarthy. «What is artificial intelligence?» En: (2007).

[14] Max Bramer. *Logic Programming with Prolog*. Springer, 2013. ISBN: 978-1-44715-486-0. DOI: `10.1007/978-1-4471-5487-7`.

[15] tutorialspoint. *Prolog Tutorial*. Abr. de 2022. URL: `https://www.tutorialspoint.com/prolog/`.

[16] Mayur Pandey y Suyog Sarda. *LLVM Cookbook*. Packt, 2015. ISBN: 978-1-78528-598-1.

[17] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. ISBN: 0-99058-290-6.