Un Sistema de Control Autónomo para Personajes de Videojuegos Basado en el Modelo Cognitivo Creencia-Deseo-Intención

Alejandro Sánchez López
Fernando Romero de la Morena
Héctor Martín Solís

GRADO EN INGENIERÍA DEL SOFTWARE
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO
DE INGENIERÍA DEL SOFTWARE

Madrid, Junio de 2016

Director: Prof. Dr. D. Federico Peinado Gil

Autorización de difusión y utilización

Los abajo firmantes, alumno/s y tutor/es del Trabajo Fin de Grado (TFG) en el Grado en …… de la Facultad de ……..……, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el Trabajo Fin de Grado (TF) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

Periodo de embargo (opcional):

☐ 6 meses

☐ 12meses

TÍTULO del TFG:

……

Curso académico: 20….. / 20…..

Nombre del Alumno/s:

……

……

……

Tutor/es del TFG y departamento al que pertenece:

……

Firma del alumno/s Firma del tutor/es

Agradecimientos y dedicatoria

Queríamos agradecer, en primer lugar, a nuestro director de proyecto Federico Peinado Gil, que ha estado desde el primer momento volcado en el proyecto así como en el equipo de *Narratech*, siempre con buena actitud, tanto en el momento de felicitarnos con los logros que íbamos consiguiendo, como cuando nos regañaba para que no nos durmiésemos en los laureles, así como por toda la ayuda e ideas que nos ha proporcionado.

En segundo lugar, a los hermanos Pérez Colado, por haber desarrollado la herramienta que hemos utilizado para el desarrollo del entorno gráfico: *IsoUnity*; que también han prestado su ayuda al explicarnos su funcionamiento cuando estuvimos estancados.

También, dentro del mismo equipo, agradecer el buen trabajo en equipo y la actitud proactiva que siempre se ha tenido, ha hecho más fácil y ameno el desarrollo de todo el proyecto.

Por último, pero no menos importante, agradecemos a nuestros amigos y familia, su opinión y continua valoración de los resultados que íbamos obteniendo durante la realización del proyecto que nos ayudaron a ver nuestro propio trabajo desde otro punto de vista.

Gracias a todos por vuestra colaboración.

Índice

Índice de figuras

Resumen

La industria del videojuego ha avanzado a grandes pasos durante los últimos años respecto a la creación de 'inteligencia artificial' para sus personajes, afirmando siempre que la utilizan para dotar de realismo y credibilidad a sus personajes. Sin embargo este concepto ha variado sustancialmente año tras año, y aún hoy, la inteligencia que encontramos en los personajes está lejos de lo que uno podría esperar de ello, incluso lejos de lo ya estudiado y conocido en la correspondiente disciplina académica.

En el afán por desarrollar personajes que sean realmente autónomos y tomen sus propias decisiones tras razonar acerca de lo que ocurre en el juego, en este trabajo proponemos un sistema capaz de dotar de control autónomo a los personajes de un videojuego y con potencial para mostrar una mayor inteligencia. Para ello conectamos un armazón de desarrollo de videojuegos llamado *IsoUnity* , desarrollado sobre el entorno *Unity*, con un sistema multi-agente llamado *Jason* e implementado en *Java*, que utiliza el conocido modelo cognitivo Creencia-Deseo-Intención[1] para representar el estado interno de la mente de los agentes, que en nuestro caso serán personajes de videojuego.

A la hora de producir un videojuego, si se implementa mediante un sistema de agentes inteligentes, con información subjetiva sobre el mundo, objetivos propios y planes y tareas que realizar, el jugador tendrá una experiencia más plena. Nuestra visión es la de adoptar este sistema en el desarrollo de videojuegos independientes de perspectiva isométrica y recursos sencillos de estilo retro, de ahí el uso de *IsoUnity*.

En esta memoria, además de explicar en detalle nuestro sistema de control, documentamos las pruebas y las adaptaciones que proponemos para llevar a la práctica este concepto, sentando las bases tecnológicas para producir un videojuego completo utilizando este sistema. Siguiendo el camino iniciado en anteriores Trabajos de Fin de Grado de esta Facultad, queríamos continuar en esa línea de trabajo afinando más el concepto y abordando un tema nuevo, el de dotar a los personajes de videojuegos creados con *IsoUnity* de una autonomía mayor y mejores herramientas de toma de decisión para poder interactuar con su entorno y con otros personajes.

Palabras clave: Inteligencia Artificial para Videojuegos, Agentes Inteligentes, Sistemas Multi-Agente, Videojuegos en Perspectiva Isométrica, Desarrollo de Videojuegos Multiplataforma

[1] Del inglés, BDI: *Belief-Desire-Intention.*

Abstract

Video game industry has made great advances in recent years towards creation of an 'artificial intelligence' for their characters, telling that they use it to provide realism and credibility to their characters' behaviors. However, this concept has changed substantially year after year, and even today, intelligence found in the characters is far from what one might expect of it, and even far from what already studied and known in the relevant academic discipline.

In the rush to develop characters that are truly autonomous and make their own decisions, after reasoning about what happens in the game, in this paper we propose an intelligent system capable of providing autonomous control the characters in a video game and the potential to show greater intelligence. To do this, we connect a game development framework called *IsoUnity* (Pérez Colado & Pérez Colado, 2014), developed on the *Unity* environment, with a multi-agent system named *Jason* and implemented on *Java*, which uses the known cognitive model Belief-Desire-Intention (BDI) to represent the internal state of the minds of the characters, which in our case will be video game characters.

When making a video game, if implemented through a system of intelligent agents, with subjective information about the world, own goals and plans and tasks to perform, the player a more fulfilling experience. Our vision is to adopt this system in the development of independent video games isometric perspective and simple retro style resources, hence the use of *IsoUnity*.

In this paper, in addition to explain in detail our control system, documented tests and adjustments we propose to implement this concept, laying the technological base to produce a complete game using this system. Following the path initiated in previous Final degree project of this Faculty, we want to continue in that line of work further refining the concept and addressing a new topic, equip the video games characters created with *IsoUnity* greater autonomy and better decision-making tools to interact with their environment and other characters.

Keywords: Artificial Intelligence for Video Games, Intelligent Agents, Multi-Agent Systems, Video Perspective Isometric, Multiplatform Game Development.

1. Introducción

Actualmente se están utilizando cada vez más técnicas de inteligencia artificial para el desarrollo de videojuegos, con el propósito de presentar experiencias y personajes más realistas. Sin embargo muchas técnicas de decisiones de personajes están diseñadas para seguir un guión (*script*), y no dan el resultado esperado.

Actualmente, sino en todos, en la gran mayoría de videojuegos que se presentan al público, dan una sensación de que los personajes cuentan con una capacidad de decisión de sus acciones y de estar realizando tareas que realmente han decidido ellos, pero no es así.

Si nos fijamos en el modo de actuación de estos minuciosamente, seguramente podremos llegar a un patrón de acción. Por ejemplo, un enemigo que se cubre con una barricada y que cada cierto tiempo se asoma para disparar, u otro enemigo que no toma 'el camino principal' para poder alcanzarte por la retaguardia.

Seguramente cualquiera que haya jugado a un juego de disparos[2] se haya percatado de la actuación de sus enemigos en combate y ha esperado pacientemente a que estos volvieran a asomarse de su cobertura por el mismo sitio, para poder acabar con ellos.

Otro ejemplo podría ser *Hitman* (IO Interactive, 2000), donde lo principal es el sigilo y donde se da por hecho que el jugador va actuar así, pero, ¿qué pasa cuando decidimos jugar de manera alternativa? Probablemente el jugador llegue al siguiente punto de control 'a su manera', disparando escandalosamente a cualquier enemigo, y aun así veremos la siguiente cinemática con el personaje principal agachado sigilosamente y sin ninguna mancha de sangre o rastro del combate que ha tenido con sus enemigos.

Estos son dos simples ejemplos para demostrar que realmente gran parte de los videojuegos están *guionizados*, tanto en los comportamientos de los personajes como en la historia que estos cuentan. Es cierto que muchos dan una sensación de 'mundo abierto' donde puedes elegir realizar una acción u otra, y los demás actuarán en consecuencia, pero realmente no hay más que eso, una sensación.

La idea principal de este proyecto es diseñar y crear el prototipo de un sistema que permita a los personajes de un videojuego tomar sus propias decisiones, siendo realmente inteligentes, y llevar a cabo sus propias acciones sin necesidad de que el diseñador haya decidido de antemano su comportamiento.

Los personajes tendrán su propia inteligencia artificial que les permitirá interactuar con otros personajes del juego, y con el jugador, y en función de esta interacción perseguir sus objetivos y tomar decisiones diferentes. Esta inteligencia artificial será implementada mediante una serie de agentes inteligentes, uno por personaje de videojuego, los cuales permiten a cada personaje percibir su entorno, procesar tales

[2] Conocidos como *First Person Shooter* como es el caso de la saga *Call of Duty* (Infinity Ward, 2003)

percepciones y responder o actuar en su entorno de una manera más racional, es decir, acorde con su visión del mundo y sus objetivos, siguiendo lo que han establecido como sus 'planes' para hacer frente a la situación en que se encuentran.

Como explicaremos más adelante, nos vamos a inspirar en un videojuego clásico llamado *La Abadía del Crimen* (Opera Soft, 1987), en el cual había personajes autónomos muy primitivos cuyo comportamiento e interacciones ayudaban a hacer progresar la historia. Desarrollaremos un escenario de prueba inspirado en este juego, usando una herramienta llamada *IsoUnity* (Pérez Colado & Pérez Colado, 2014), que más adelante explicaremos cómo amplía la funcionalidad del entorno de desarrollo de videojuegos *Unity* (Unity Technologies, 2005), y probaremos en dicho escenario la utilidad de nuestro sistema.

La estructura de esta memoria es la siguiente: tras este primer capítulo de introducción, tenemos un segundo capítulo en el que revisamos el estado del arte, cómo se encuentra ahora mismo la investigación y el desarrollo en materia de personajes autónomos en videojuegos, junto con algunas técnicas e ideas que consideramos interesantes. En el Capítulo 3 especificamos los objetivos de este trabajo, mientras que en el Capítulo 4 indicamos cuál es nuestro método de trabajo y las herramientas con que contamos para realizarlo. En el Capítulo 5 explicaremos cómo fue el proceso de desarrollo, las actividades de análisis, diseño, implementación y prueba de este sistema de control propuesto, que permitirá dotar de autonomía a los personajes según un modelo cognitivo llamado 'Creencia-Deseo-Intención' que habrá sido explicado en el Capítulo 2. Finalmente, en el Capítulo 6 discutimos los resultados obtenidos y las ventajas y limitaciones del sistema propuesto; y en el Capítulo 7 damos nuestras conclusiones sobre lo que puede aprovecharse de este trabajo, junto a algunas propuestas de posibles líneas de actuación para darle continuidad.

2. Revisión del estado del arte

En este apartado se revisarán las actuales tecnologías aplicadas a videojuegos en el ámbito de la inteligencia artificial, así como determinados problemas que conllevan y se hará una explicación del modelo cognitivo de los agentes inteligentes.

2.1. Tecnología para crear personajes interactivos en videojuegos

Actualmente muchos videojuegos utilizan distintas técnicas de inteligencia artificial. Lo más común es la utilizarlas para el control de los personajes no jugadores[3], como por ejemplo para realizar búsqueda de rutas en el escenario evitando las superficies no transitables.

Otra utilidad destacable de la Inteligencia Artificial es usarla para aumentar la dificultad de los videojuegos. La mayoría de los videojuegos te dejan elegir entre distintos niveles de dificultad, los cuales, cuanto mayor es el nivel, más aumentan la capacidad de realizar decisiones mejores que permitan la derrota del jugador. Este tipo de inteligencia se puede aplicar tanto a juegos de disparos como a juegos de estrategia.

También cabe mencionar que existen concursos donde se premia por la mejor Inteligencia Artificial desarrollada. Un ejemplo de esto es *BotPrize* (Hingston, 2014), donde se premia al mejor bot (sistema de control autónomo de un personaje) desarrollado con *Unreal Engine* (Epic Games, 1998). Unos de sus sponsor más destacados de *BotPrize* (Hingston, 2014) es la compañía *2K Games*, conocida por videojuegos comerciales importantes como *Battleborn* (Gearbox Software, 2016) o la saga *Borderlands* (Gearbox Software, 2009).

2.1.1. Ejemplos de aplicación de Inteligencia Artificial en videojuegos

Actualmente en la industria del videojuego se busca el mayor realismo y la máxima similitud con la realidad. Es por esto que la Inteligencia Artificial está, si no en todos, en la gran mayoría de juegos que pretenden que los personajes no jugadores sean lo más 'reales' posibles y se adapten al entorno en el que están inmersos. A continuación, describiremos algunos ejemplos actuales.

Max Payne 3 (Rockstar Games, 2012) - Se trata de un videojuego de acción y disparos en los que los enemigos están provistos de una Inteligencia Artificial que los clasifica en varios tipos según sus acciones, pero que tienen en común un excelente manejo táctico de la situación. En la acción contra el jugador, los enemigos se mueven por el entorno intentando ocultarse del jugador detrás de cualquier elemento sólido para evitar ser alcanzado por disparos y mantenerse con vida; intentan flanquear al jugador por

[3] NPC's, del inglés *Non-Playable Characters.*

diferentes lados, en lugar de atacar ‘de frente’ al jugador, y ponerse así en una situación comprometida.

Fig. 2.1: Max Payne 3 (2012), enemigo divisando a jugador.

GTA V (Rockstar North, 2013) **-** Un clásico en el mundo de los videojuegos y una de sus franquicias más caras de producir y a la vez más rentables de la historia. En la última entrega de *Grand Theft Auto* (GTA) ha habido grandes innovaciones en la inteligencia artificial, tanto de los enemigos a los que nos enfrentamos, como en los ciudadanos que el jugador se encuentra paseando por la ciudad. Por ejemplo, apuntar con un arma a una persona hace que esta se asuste y reaccione según el tipo de personaje: tal vez huyendo, o atacando al jugador, o levantando las manos, etc.

Fig. 2.2: GTA V, personaje asustado por ver un arma

Splinter Cell Blacklist (Ubisoft Toronto, 2013) **-** Tanto en este videojuego en particular, como en todos los demás de la franquicia, el objetivo de infiltración y sigilo es lo que el jugador tiene que tener en cuenta, sabiendo que sus enemigos cuentan con una Inteligencia Artificial muy consciente de su entorno. Los personajes están siempre alerta a cualquier sonido que pueda ser sospechoso, y controlan las zonas con poca luminosidad intentando dar caza al jugador.

Fig. 2.3: Splinter Cell Blacklist (2013), enemigo en alerta

Killzone Shadow Fall (Guerrilla Games, 2013) - Esta franquicia hace que los enemigos tengan una Inteligencia Artificial que aprende con el desarrollo del juego, lo que hace que cuanto más avancemos en él, mejor sepan reaccionar los enemigos ante las situaciones, esquivando ataques, cubriéndose con su entorno, y realizando cualquier acción de combate.

Fig. 2.4: Killzone Shadow Fall (2013), enemigos utilizando escombros para protegerse

Son muchos ejemplos los que se podrían mostrar, ya que la Inteligencia Artificial está presente en prácticamente cualquier personaje que no esté enteramente *guionizado*, ya pertenezca a un juego de acción (*F.E.A.R.* (Monolith Productions, 2005), *Crysis* (Crytek, 2007), *Left 4 Dead* (Valve Software, 2008), etc.*)*, estrategia (*Age of Empires* (Ensemble Studios, 1997)*s*, *Starcraft* (Blizzard Entertainment, 2009), *Warcraft* (Blizzard Entertainment, 1994), etc.) o cualquier otro género que presente un enemigo con la 'necesidad' de pensar por sí mismo y tomar sus propias decisiones a cada momento.

2.1.2. Problemas actuales de la Inteligencia Artificial en videojuegos

Muchos videojuegos dan la falsa sensación de inteligencia artificial mediante un *script* que el personaje sigue a rajatabla. Este fenómeno se conoce en el mundo académico como efecto '*Eliza*', en el que el usuario interpreta la salida del sistema como un evento generado por el azar o la casualidad del momento, y otorga a las entidades con las que interactúa cualidades y habilidades humanas de manera errónea, ya que, al menos en el caso de los videojuegos, toda la situación generada está prevista por los desarrolladores y contemplada en el guion del videojuego. Aunque parezca que nuestras decisiones están influenciando al personaje, este simplemente está siguiendo las órdenes de un *script* y si acaso adapta algunas de sus acciones a distintas variables.

Fig. 2.5: Decisiones de The Wolf Among Us que modificarán el rumbo de los acontecimientos.

Otro problema de la tecnología actual de videojuegos es la falsa impresión de que, al realizar una acción, el hilo de la narración cambia completamente el hilo en el videojuego. Determinados videojuegos como *The Wolf Among Us* (Telltale Games, 2013) te dejan poco tiempo para realizar una decisión que consideres importante y supuestamente que va a cambiar completamente el curso de los acontecimientos. Sin embargo sólo hace falta volver a jugarlo tomando la otra decisión para comprobar que la historia no ha cambiado prácticamente en nada.

Uno de los retos que nos planteamos con nuestro proyecto es demostrar que las acciones del usuario, pueden afectar completamente al devenir de los acontecimientos, para que, efectivamente cada vez que se tome una decisión, todo cambie.

Actualmente se habla de un nuevo fenómeno en la inteligencia artificial aplicada a los videojuegos conocido como Inteligencia Artificial emergente. Hace referencia a aquel diseño que se basa en dotar a los personajes de muchos comportamientos simples pero potentes en cuanto a significado e implicaciones, de manera que, al combinarse entre sí, dan lugar a comportamientos del sistema global mucho más complejos y sofisticados. En cierto modo nosotros buscamos facilitar al desarrollador de videojuegos las herramientas para hacer que esta Inteligencia Artificial emergente sea posible.

También empieza a ser habitual encontrarse con personajes del juego que aprenden según las decisiones del jugador, ajustando sus comportamientos se modifica en consecuencia. Uno de los ejemplos más destacables es *Drivatar*^{TM} (*Driving Avatar*), es un ejemplo de Inteligencia Artificial desarrollada por Microsoft para el videojuego *Forza Motorsport* (Turn 10 Studios, 2005), que basado en técnicas de *Machine*

Learning[4], el conductor virtual consigue desarrollar una manera de actuar y pensar al volante, muy similares a las del jugador humano.

Fig. 2.6: Conducción del DrivatarTM en Forza Motorsport

Hace muchos años ya se buscaban maneras de que una entidad interna pudiera ayudar o frenar al jugador en la medida que este avanzaba sobre el mismo videojuego. Tenemos videojuegos clásicos como *Pac-Man* (Namco, 1980), donde los fantasmas persiguen al jugador y le hacen difícil moverse por las diferentes galerías del mapa. En el *Pac-Man* original de hecho esa sensación de persecución era puro efecto Eliza. Otro ejemplo de ayuda son los numerosos personajes repartidos por los mundos de *Mario Bros* (Nintendo EAD, 1985), donde los personajes parecen estar implicados en las diversas aventuras del fontanero, aunque en realidad son totalmente reactivos a las acciones del jugador.

Finalmente podemos hablar de *La Abadía del Crimen* (Opera Soft, 1987), un videojuego español de los más clásicos, aquel en el que nos inspiramos para la realización de este proyecto. *La Abadía del Crimen* también se beneficiaba bastante del *efecto Eliza* para poder introducir al jugador en un mundo de asesinatos, intrigas y misterios, pese a la muy limitada inteligencia artificial creada por los desarrolladores del juego, acorde a los recursos de la época. La manera habitual de generar este efecto era programar personajes autónomos usando un modelo orientado a eventos, si el usuario hace algo o llega a ser tal momento del día, el personaje recibía el evento y caminaba hasta un nuevo lugar que tenía programado.

En el desarrollo de nuestro proyecto hemos utilizado herramientas realizadas en trabajos de años anteriores, concretamente *IsoUnity* (Pérez Colado & Pérez Colado, 2014) y hemos partido de proyectos similares a este pero que tenían personajes no jugadores muy básicos, que tan sólo se limitan a seguir al jugador o ejecutar acciones *guionizadas* como parte de una cinemática, como el trabajo de fin de grado de Reconstrucción de una video-aventura (Santamaría Barcina & Alexiades Estarriol, 2015). Un proyecto

[4] *Machine Learning* o Aprendizaje Automático: rama de la inteligencia artificial que trata de crear programas capaces de generalizar comportamientos a partir de una información no estructurada suministrada en forma de ejemplos.

similar que también conecta agentes inteligentes a *IsoUnity* es *TinyWorld* (Pérez Colado & Pérez Colado, 2014), aunque no utiliza ningún modelo cognitivo (véase 2.3).

2.2. Agentes autónomos

Los agentes inteligentes se describen como entidades capaces de percibir su entorno, procesar tales percepciones y responder o actuar sobre el mundo que habitan de manera racional, es decir, con coherencia y tratando de alcanzar un resultado esperado.

Los agentes inteligentes tienen las siguientes características (Incremental Learning in Intelligent Systems, 2013) :

- Aprender nuevos problemas e incrementar normas de solución.
- Capacidad de adaptación en línea y en tiempo real.
- Ser capaz de analizar condiciones en términos de comportamiento, el error y el éxito.
- Aprender y mejorar a través de la interacción con el medio ambiente (realización).
- Aprender rápidamente de grandes cantidades de datos.
- Deben estas basados en memoria de almacenamiento masivo y la recuperación de dicha capacidad.

Y pueden clasificarse en 6 grupos diferentes: Agentes reactivos, Agentes reactivos basados en modelo, Agentes basados en objetivos, Agentes basados en utilidad, Agentes que aprenden y Agentes de consultas.

2.2.1. Agentes basados en el modelo cognitivo Creencia-Deseo-Intención

En la arquitectura Creencia-Deseo-Intención[5] los agentes son vistos como entidades capaces de percibir su entorno, construir una creencias en base a esas percepciones, mantener una serie de deseos activos que guiarán sus decisiones y también unas determinadas intenciones en ejecución[6]. De esta manera el agente toma sus decisiones en función de sus propios estados, con un modelo que nos parece racional y humano.

Esta separación en creencias, deseos e intenciones se toma de la teoría del razonamiento humano que propone Michael Bratman, filósofo americano, considerado el autor de este modelo.

Para Bratman, creencias y deseos son actitudes mentales asociadas a la acción; sin embargo, la intención va asociada al control de la actitud de la persona (o agente software en este caso). Todo esto y mucho más relacionado con su modelo, en el que

[5] BDI, del inglés *Belief-Desire-Intention*.
[6] Actividades compuestas de varias acciones en secuencia sobre la que hay que hacer cierto seguimiento.

separa estos tres conceptos, está tratado en su libro *Intención, planes y razón práctica* (Bratman, 1999).

Posteriormente, las investigaciones en este área han conducido, por ejemplo, a la axiomatización de algunas implementaciones BDI, así como a las descripciones lógicas formales, como la de Anand Rao y Michael Georgeff: BDICTL propuesto en su libro *Deliberation and its Role in the Formation of Intentions* (Rao & Georgeff, 1991) o la extensión de esta por Michael Wooldridge para definir LORA: *The Logic Of Rational Agent* (Wooldridge, 2000), incorporando una lógica de acción. Por nuestra parte, para el manejo de los agentes BDI en este trabajo, hemos utilizado un lenguaje de programación orientado a agentes llamado *AgentSpeak* (abreviado como ASL) (Bordini, Hübner, & Wooldridge, 2007). Se basa en programación lógica y utiliza una arquitectura BDI de agentes autónomos que suele denominarse 'arquitectura cognitiva'.

Para el manejo de éste lenguaje, se utiliza un entorno adaptado a *Eclipse* llamado *Jason*: un intérprete para una versión extendida de *AgentSpeak*. Implementa las operaciones semánticas de ese lenguaje y aporta una plataforma para el desarrollo de sistemas multi-agentes con varias herramientas personalizables por el usuario. *Jason* está disponible como una librería software de código abierto y se distribuye bajo licencia GNU LGPL.

2.2.1.1. El modelo Creencia-Deseo-Intención

Para entender los agentes ASL hay que entender que se basan en el modelo BDI, que se compone de tres partes:

Creencias (*Beliefs*): Conocimiento que el agente tiene sobre el mundo. Esta información no tiene por qué ser veraz, podría estar desfasada o equivocada.

Deseos (*Desires*): Son todos los posibles estados del mundo que al agente le gustaría conseguir. De todos modos, tener un deseo no siempre implica que un agente actúe de forma directa para conseguirlo. Es simplemente un elemento que influye en las acciones del agente.

Intenciones (*Intentions*): Son la representación de las tareas que el agente se ha propuesto realizar. Las intenciones suelen estar asociadas con determinadas metas (*Goals*) y son a la vez el resultado de considerar opciones y de elegir. Las opciones elegidas de esta manera son intenciones, que a veces tendrán la forma de un plan complejo y otras veces de una simple acción.

El modelo BDI está basada en la lógica CTL*, dicha lógica es una lógica proposicional ramificada temporalmente, una extensión de la lógica modal, la cual es prácticamente usada en sistemas de reglas, donde está presente el tiempo. Existe una cierta relación con otras variedades de lógica, por ejemplo, la lógica modal. Esta llamada lógica modal, es la que intenta capturar el comportamiento deductivo de algún grupo de operadores modales. Los operadores modales son expresiones que califican la verdad de los juicios. (Zalta, 2009)

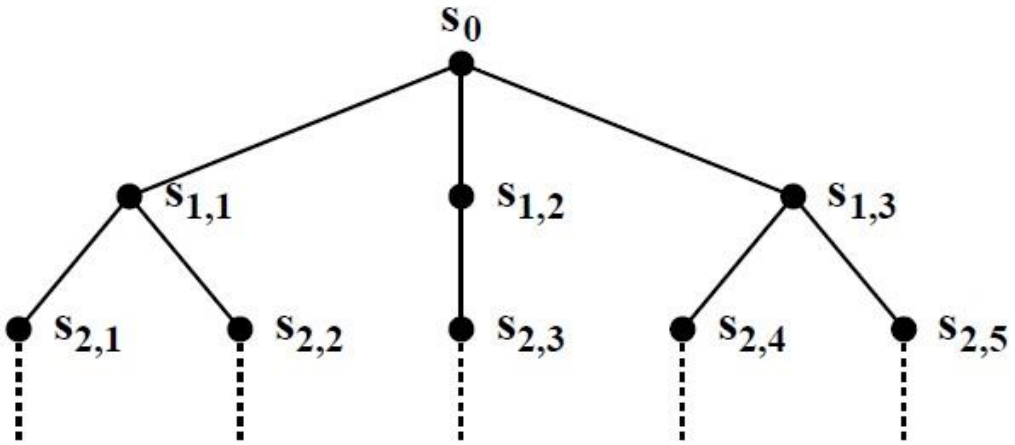

Fig. 2.7: Ejemplo de una lógica proposicional ramificada

El modelo BDI posee en realidad una arquitectura abstracta. La idea para implementarlo suele ser desarrollar una simplificación. Sólo las creencias del estado actual representan el conocimiento en forma de planes. Una intención es representada usando una pila de planes relacionados jerárquicamente, pudiendo coexistir múltiples pilas.

2.2.1.2. Agentes en lenguaje AgentSpeak

Los agentes implementados con el lenguaje *AgentSpeak* (ASL) son sistemas situados en algún entorno (*environment*). Estos agentes son capaces de percibir su entorno mediante sensores (*sensors* o *perceptions*), y tienen un repertorio de posibles acciones que pueden ejecutar (vía *effectors* o *actuators*) para modificar su entorno. Para decidir qué hacer se trabaja cambiando las percepciones del entorno y este según sus planes actuara en consecuencia y realizara las acciones pertinentes.

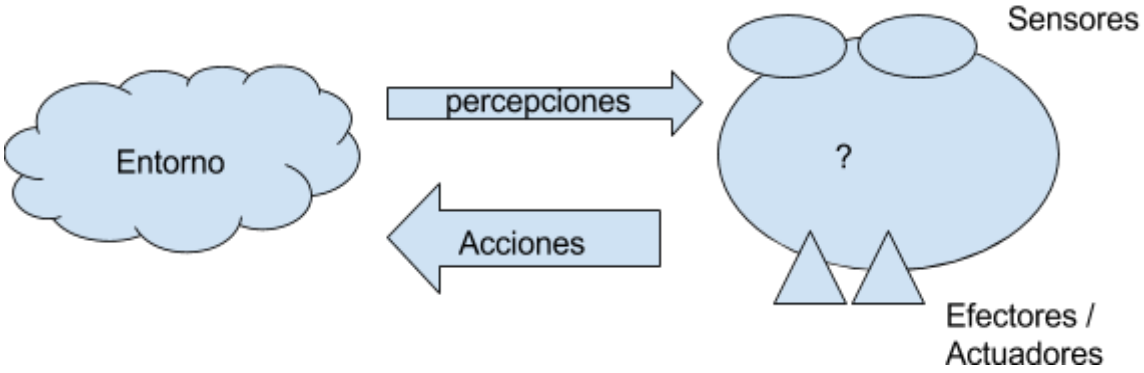

Fig. 2.8: Relación del entorno y los agentes

Por lo general, es más común que los agentes habiten en un entorno junto a otros agentes, dando lugar a un sistema multi-agente. Con lo cual, la modificación del entorno pasa a deberse a la interacción de varios agentes, haciendo la vida de estos más complicada, ya que para conseguir su meta en el entorno tendrá que tener en cuenta cómo otros agentes reaccionan a dichos cambios.

```
started.
+started <- .print('Hola mundo').
```

Fig. 2.9: Ejemplo de código ASL

La definición de un agente en *AgentSpeak* consiste en dos partes:

- Creencias iniciales (*initial beliefs)* y metas iniciales (*initial goals*) del agente
- Planes (*plans)* de actuación disponibles en el agente

En el código de ejemplo mostrado la primera línea define un *belief* inicial para nuestro agente[7]. *AgentSpeak* no tiene variables al estilo de lenguajes de paradigma imperativo como *C* o *Java*; se trabaja con otras nociones, como las creencias, metas y planes ya citadas.

La segunda línea define un plan. Se puede interpretar como: '*en el momento que 'empieces a creer'* 'started', *escribe el texto* 'Hola mundo''.

Un agente *AgentSpeak* se define por un conjunto de creencias *(beliefs)* que representan el estado inicial de su 'base de la creencias' *(belief base)*. Esta contiene un conjunto de fórmulas atómicas, y un conjunto de planes *(plans)* que conforman su 'biblioteca de planes' *(plan library)*. Para que el agente pueda empezar a actuar, se debe conocer cuáles son sus metas *(goals)* y cuales los eventos desencadenantes de sus cambios internos *(triggering events)*.

Hay dos tipos de metas (*Goals*):

- *Achievement goals*. Formadas por fórmulas atómicas precedidas del símbolo *[!]*. Estas metas se establecen al principio de la ejecución del agente, y por lo tanto empezará por intentar realizar estas metas.
- *Test goals*. Formadas por fórmulas atómicas precedidas del símbolo *[?]*, estas metas indican que el agente querrá probar si una determinada fórmula propuesta puede demostrarse, mediante unificación, con ayuda de algunas de sus creencias.

Se puede decir que un agente básico se comporta como un mero sistema de reacción. Estas reacciones se deben a eventos, que pueden ser cambios en las creencias provocados a su vez por la vía de la percepción del entorno, o cambios en las metas del agente originadas de la ejecución de planes previamente lanzados por otros eventos, como una especie de sub-metas.

El evento disparador *(triggering event)* define qué condiciones pueden hacer que se inicie la ejecución de un plan.

Los planes (*plans*) son establecidos en la programación del agente de tal modo que, como dijimos, son disparados por la adición *(+)* o eliminación *(-)* de creencias o metas. Los planes tienen dos partes:

[7] Aunque sólo hay uno inicial, podríamos haber dado una lista de ellos.

- Encabezado *(head)* que es formado a partir de un evento disparador, y un conjunto de 'creencias literales' que representan un determinado contexto. La conjunción de literales en el contexto debe ser una consecuencia lógica de las creencias actuales del agente si el plan se considera aplicable en ese instante.
- Cuerpo *(body)*: secuencia de acciones básicas o sub-metas que el agente tiene que conseguir o testear cuando se dispara el plan. El cuerpo incluye acciones básicas que representan operaciones atómicas que el agente puede ejecutar para cambiar su entorno.

```
+green patch(Rock)
     : not battery charge(low)
     <-   ?location(Rock,Coordinates);
           !traverse(Coordinates);
           !examine(Rock).
```

Fig. 2.10: Ejemplos de planes

Un ejemplo de uso de este sistema para solucionar problemas reales es el vehículo robótico *Rover*, que se utilizó en la exploración de Marte. Fue programado utilizando el lenguaje ASL y podemos ver un resumen de cómo se planteó el diseño para que este vehículo actuara ante una cierta situación.

En el código que hemos visto antes, el plan establece que cada vez que el *Rover* perciba una traza verde (*green patch*) en una roca determinada, debe intentar examinar dicha roca; sin embargo, ese plan solo puede ser usado en caso de que las baterías no estén a un nivel bajo. Para examinar la roca tiene que recopilar de su propia base de conocimiento las coordenadas asociadas con esa roca[8], entonces alcanzará esa meta de atravesar esas coordenadas y una vez allí, examinara la roca.

```
+!traverse(Coords)
     : safe path(Coords)
           <- move towards(Coords).
```

```
+!traverse(Coords) :
     : not safe path(Coords)
           <- ...
```

Fig. 2.11: Ejemplos de *goals*

Como es normal, cada una de estas *achievement goals* se dispararon en la ejecución de otro plan.

Los otros dos planes que hemos visto a continuación ofrecen secuencias alternativas de acciones que el *Rover* puede realizar según lo que cree que hay por el ambiente, especialmente cuando tiene que atravesar dando una ruta. Si el *Rover* cree que hay un camino seguro en esa dirección, entonces todo lo que tiene que hacer es realizar la

[8] Esta es la *test goal* al principio del cuerpo del plan.

acción de avanzar hacia esas coordenadas[9]. El plan alternativo no se muestra aquí; pero se trata de uno que debe proporcionar medios alternativos para que el agente pueda llegar a la roca evitando caminos inseguros.

2.2.2. Entornos

En un entorno multi-agente, este está compartido entre todos los agentes, por lo que las acciones de un agente pueden interferir con las de los otros que se encuentren en el mismo entorno. Por ello, tener una noción explícita del entorno, un modelo, pese a no ser obligatorio, es muy necesario para el desarrollo de un buen sistema multi-agente.

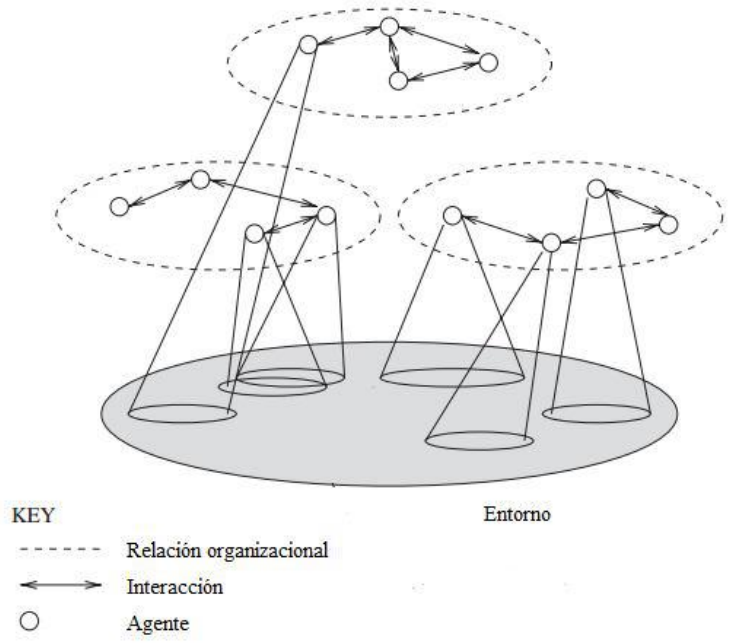

Fig. 2.12: Ejemplo gráfico de agentes comunicándose sobre un entorno

Hay algunos sistemas en los que el entorno está ya definido, pero en otros, como es nuestro caso, necesitaremos crear un nuevo modelo y el correspondiente entorno artificial, simulando dinámicamente los cambios del mismo. Una vez el entorno está definido es cuando se implementarán los agentes que operarán en él, percibiendo el tipo de las propiedades del entorno que se hayan definido en el modelo.

Suponiendo el entorno creado y los agentes definidos e implantaos sobre él, estos tendrán que interactuar tanto entre ellos mediante una comunicación directa agente-agente, como con el entorno, atendiendo a las percepciones que reciban sobre él e intentando efectuar cambios en el entorno. Esta relación se explica en la siguiente figura.

[9] Esta es una acción básica a través de la cual el Rover puede efectuar cambios en su entorno

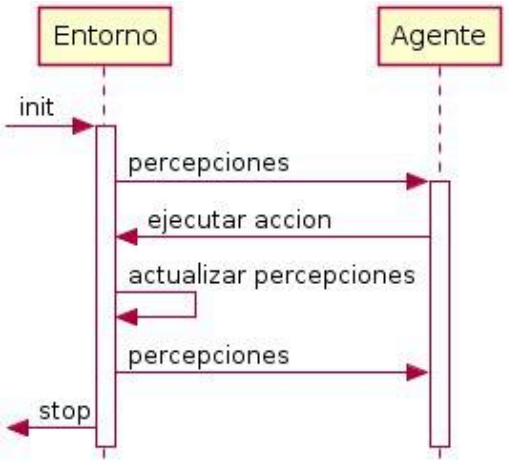

Fig. 2.13: Interacción entre entorno y agente.

2.2.2.1. Creación del Entorno

La clase *Java Environment* es proporcionada por *Jason* y permite percepciones individualizadas facilitando la tarea de asociar determinadas percepciones sólo a ciertos agentes. Esta clase mantiene la estructura para almacenar las percepciones que sólo unos agentes pueden consultar, así como las percepciones globales que existan.

Para ello, el *Environment* que tendrá que ser creado como una extensión de dicha clase, cuyos métodos tendrán que ser sobrescritos *(override)* para ajustarse a las necesidades de lo que se desee crear (véase en la Fig. 2.14Fig. 2.12).

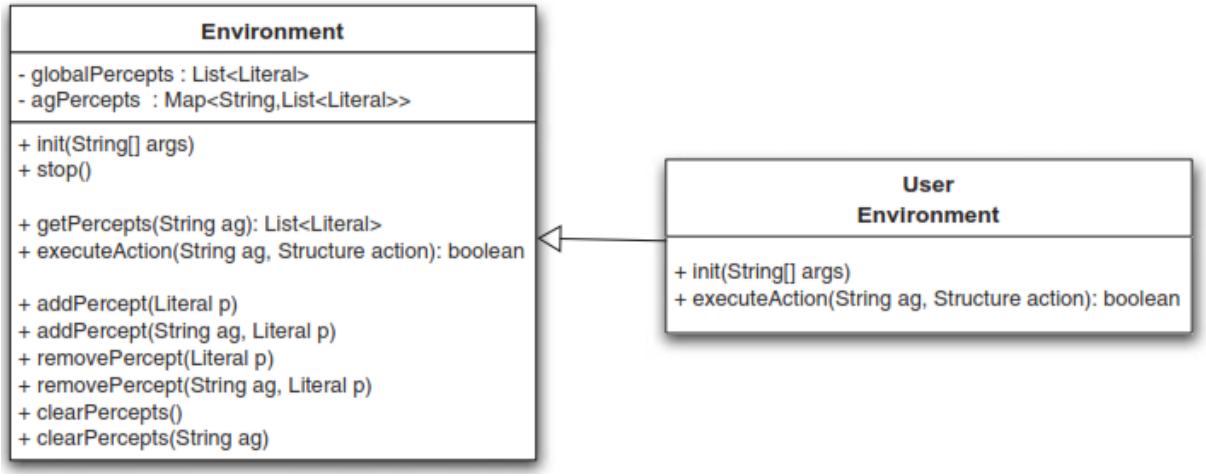

Fig. 2.14: Diagrama de clases para el entorno. El entorno personalizado del usuario heredará de la clase *Environment.*

2.2.2.2. Métodos del Entorno

- *init*: utilizado para recibir parámetros del entorno y producir la configuración principal. Se utiliza también para inicializar la lista de percepciones que tendrá el entorno cuando el sistema multi-agente empiece a funcionar.

- *addPercept(L)*: añade un literal L a la lista global de percepciones. Todos los agentes lo podrán percibir al ser global.

- *addPercept(A, L):* añadir un literal a la lista de percepciones del agente A. Solo este agente podrá percibirlo.

- *removePercept(L), removePercept(A, L):* mismo caso que los anteriores pero para eliminar una percepción global, o local a un agente.

- *clearPercepts() y clearPercepts(A):* en lugar de eliminar una percepción, elimina todas; puede hacerse para todos, o para un agente, respectivamente.

La lista de percepciones en la clase del entorno es automáticamente sincronizada por los métodos descritos arriba. Esto es importante dado que los agentes se ejecutan simultáneamente, y el la acción y las estructuras de las percepciones son globales para la clase del Entorno; aunque el trabajo básico de sincronización se realiza por *Jason*, dependiendo de cómo se lleva a cabo el modelo del entorno.

2.3. Agentes inteligentes en control de personajes de videojuego

Dentro del campo de los agentes inteligentes en los videojuegos podemos encontrar ejemplos tan interesantes como *Façade* (Mateas & Stern, 2005), una 'obra de teatro interactiva' sobre dos personajes que representan a un matrimonio con dificultades, y el personaje que interpreta el jugador, un amigo común. En este juego mediante una interacción entre los tres, que incluye comunicación mediante lenguaje natural, va surgiendo una historia que no sigue al pie de la letra el guion sino que son los propios personajes, sus pensamientos y emociones los que les hacen actuar en cada momento.

Fig. 2.15: Trip y Grace protagonistas de Façade

Otro ejemplo es *Prom Week* (kongregate, 2012), desarrollado en la Universidad de California en Santa Cruz, también un videojuego con un sistema de narración interactiva detrás que en esta ocasión hace las veces de 'simulador social'. En *Prom*

Week el jugador da vida a un grupo de estudiantes de secundaria en la semana más dramática de su carrera: el fin de curso de su etapa en el instituto. *Prom Week* (kongregate, 2012) es capaz de combinar la simulación dinámica de juegos al estilo *Los Sims* (Electronic Arts, 2000), con los personajes detallados y el diálogo de los mismos.

Fig. 2.16: logo de *Prom Week*

Para finalizar podemos hacer referencia a un proyecto financiado por la Unión Europea (red de excelencia, FP7-ICT-231824) y llamado IRIS[10] (André & Kurdyukova, 2009-2011), que comenzó en enero de 2009 y terminó en diciembre de 2011.

Entre los muchos desarrollos y teorías sobre narración interactiva que recopiló y propuso había una parte dedicada a los personajes creíbles (*Believable Characters*), dotados de autonomía y emociones. El lector puede recurrir a la base de datos de este proyecto para hacerse una idea de la gran cantidad de trabajos académicos (rara vez videojuegos comerciales) que exploran el uso de sistemas y agentes inteligentes para controlar personajes de videojuego.

Fig. 2.17: Ejemplo de narración interactiva, uno de las aplicaciones recopiladas en el proyecto IRIS

También podemos hacer una especial mención a *TinyWorld* (Pérez Colado & Pérez Colado, 2014), un proyecto de inteligencia artificial desarrollado por los hermanos Pérez Colado, al igual que *IsoUnity* (Pérez Colado & Pérez Colado, 2014).

[10] Del inglés, *Integrating Research in Interactive Storytelling*.

En este videojuego, se van proponiendo unos objetivos a cumplir a lo largo de la partida, donde el jugador es un ‘ser omnipotente’ que debe organizar a los seres para que se cumplan un máximo número de objetivos.

Pese a la simplicidad que puede parecer tener, el comportamiento de esos personajes lleva un desarrollo de inteligencia artificial donde, al principio, estos personajes desconocen el universo donde están ubicados, pero en el avance del juego y con las interacciones, van descubriendo nuevas capacidades e instrumentos del entorno.

El jugador, debe ser consciente de esto y hacer que los personajes que maneja, traten de evitar daños y que realicen tareas que estén al alcance de sus conocimientos, pudiendo hacer de manera escalada que vayan enfrentándose a retos mayores cuando amplíen sus conocimientos.

Los habitantes por su parte cuentan con un comportamiento básico que premia la supervivencia y la eficiencia, dividiendo las tareas siempre que sea posible y oportuno. Además, para la realización de algunas tareas los personajes pueden asociarse y colaborar uniendo fuerzas o distribuyendo roles que les permitan realizar la labor.

3. Objetivos

En este trabajo tratamos de explorar el uso de agentes BDI para crear personajes de videojuegos inteligentes, que sean capaces de interactuar entre ellos, comunicándose incluso para decir mentiras o información incompleta para manipular a otros agentes. Esto es lo que nos hizo plantearnos el uso del modelo cognitivo Creencia-Deseo-Intención porque alterar las creencias y los deseos de otros personajes hará que estos actúen de manera diferente, ya que sus acciones dependen de lo que creen y desean.

La visión original era poder combinar un simulador como el de *Los Sims* (Electronic Arts, 2000) originales y a la vez contar una historia desde el punto de vista de un protagonista, una experiencia más parecida a *La Abadía del Crimen* (Opera Soft, 1987). Tomamos como referencia un cruce de ideas entre los personajes autónomos de *Los Sims* (Electronic Arts, 2000) (habitantes de unas casas que tienen sus rutinas habituales como comer, dormir, ir al baño, etc.) con el Trabajo de Fin de Grado de Alexiades y Santamaría (Santamaría Barcina & Alexiades Estarriol, 2015), en el que comenzaron a modelar ese escenario donde unos monjes se comportan según lo que han oído acerca de los libros prohibidos de la biblioteca de la Abadía.

Para ilustrar el sistema propuesto, crearemos un escenario y unos personajes, ambientados también en *La Abadía del Crimen* que se mueven a su libre albedrío por ese entorno y hacen realizan acciones por su cuenta. El jugador será quien maneje a uno de esos personajes y podrá intervenir haciendo que su personaje interactúe con otros o realice ciertas acciones sobre los objetos del escenario para que de una manera indirecta cambie el comportamiento de todos los demás monjes.

El género no será de acción ni estrategia, tendrá un planteamiento cercano a la aventura, pero siendo esencialmente una simulación controlada en la que se podrán observar cómo actúan los personajes según sus objetivos cuando se ven envueltos en situaciones que interfieren en sus planes de acción. Con este escenario de ejemplo, analizaremos las ventajas y los inconvenientes de nuestro sistema.

3.1. Integración *Jason – IsoUnity*

Uno de los retos que hemos tenido desde el principio ha sido la comunicación entre dos plataformas tan diferentes como son *Jason* y *Unity*. Cada una de ellas utiliza su propio lenguaje. *Jason*, como dijimos, es una plataforma que permite compilar y ejecutar agentes programados mediante el lenguaje ASL, un lenguaje de paradigma declarativo muy diferente por tanto al más popular *Java*.

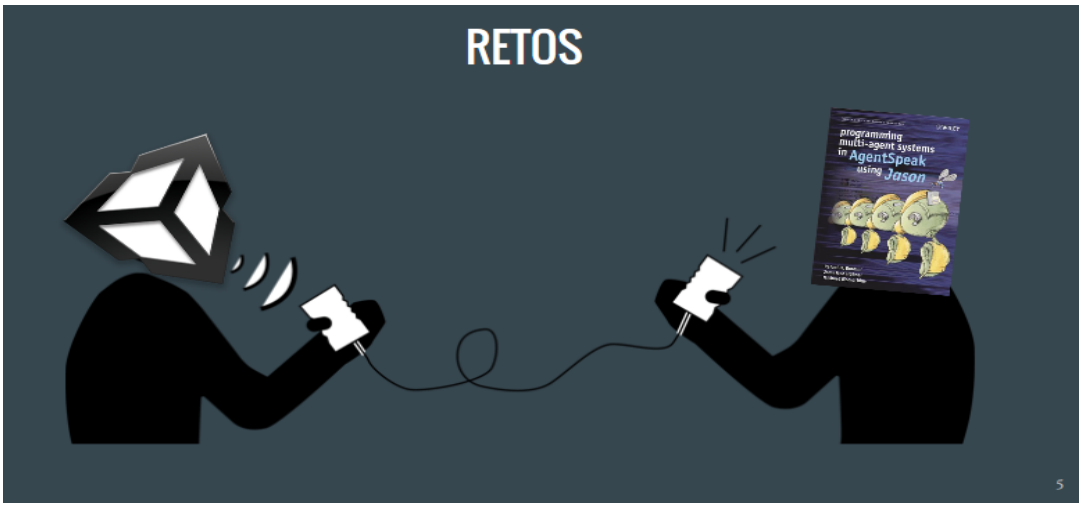

Fig. 3.1: Esquema que ilustra el primer objetivo de este trabajo. El primero de los retos a los que nos enfrentamos fue la comunicación entre ambos entornos de desarrollo

El sistema utiliza una arquitectura de red tipo cliente-servidor, donde el servidor contiene el modelo y está implementado en *Java* (junto a *Jason*) y su lenguaje de agentes ASL. El cliente está implementado en un proceso aparte, una aplicación hecha en *Unity* (junto a *IsoUnity* (Pérez Colado & Pérez Colado, 2014)) que hace las veces de la vista. Digamos que todo el conjunto sigue el Modelo Vista Controlador, donde el Modelo sería la parte más interna de *Java*, la Visa sería toda la parte de *IsoUnity* y el controlador, la parte del Entorno en *Java*, que se encarga de gestionar la comunicación entre ambas partes.

Por ello era necesario diseñar e implementar un protocolo que permitiera el envío de información entre estos dos nodos de la red, y que pudiera procesar de forma automática las distintas peticiones.

Para conseguir la comunicación entre ambos entornos de desarrollo, decidimos recurrir a una comunicación sencilla mediante sockets. Ambos extremos, cliente y servidor, serían capaces tanto de escuchar como de enviar contenido (objetos *Java*) a través de una dirección IP y un puerto acordado.

Para que la información fuese coherente en ambos entornos, esta se envía y recibe en forma de objetos encapsulados en formato JSON, sintaxis que ambos lenguajes son capaces de interpretar e integrar en sus estructuras de datos con facilidad para su tratamiento posterior.

3.2. Escenario de narrativa emergente con agentes BDI

La idea de utilizar agentes BDI surgió a través de la falta que vimos en los distintos videojuegos narrativos actuales de una narración que parezca real. Actualmente, la mayoría de los videojuegos están *guionizados* y siguen un patrón ya definido.

En este proyecto intentamos que los distintos personajes presentes tengan su propia historia que contar. Es decir, cada uno de ellos irá interactuando con el entorno y los distintos personajes sin que el jugador se dé cuenta, al no ser que esté en ese momento en la misma ubicación que los personajes.

De esta manera, si se aplicase esta metodología en los personajes de los videojuegos, pese a que este tuviese un principio, fin e hitos establecidos, habrá múltiples maneras de llegar a ellos debido a que las acciones de cada personaje serán acontecimientos significativos e influirán en el comportamiento de otros personajes. Esto daría mucho más dinamismo a la hora de avanzar en la historia, ya que habría muchas formas posibles de generar un cierto acontecimiento.

4. Metodología y herramientas

Para la realización del proyecto, consideramos que lo óptimo es seguir un modelo de desarrollo que sea capaz de afrontar muchos cambios en las especificaciones sin afectar demasiado al resultado final. Además, al comenzar a desarrollar con herramientas que no se conocen, habrá que invertir una importante cantidad de tiempo en la investigación y pruebas.

Una vez establecido los objetivos a desarrollar, se procuró seguir un modelo de desarrollo ágil, parcialmente basado en SCRUM, donde nos hemos encargado de llevar a cabo unas diferentes etapas, donde definimos objetivos, desarrollamos y observamos el valor que aportan al proyecto.

En las reuniones se planteaba:

Requisitos: Comentamos los cambios previstos para la próxima iteración de manera priorizada e intentamos resolver cualquier posible duda que estos puedan conllevar.

Planificación: Elaboramos una lista de tareas de la iteración necesarias para desarrollar los requisitos que hemos estimado a realizar. La estimamos, de manera conjunta, el esfuerzo y nos dividimos las tareas.

Demostración: Presentamos el proyecto para una prueba de evaluación donde veíamos los cambios realizados y pensábamos los que habría que realizar después.

Retrospectiva: Comentamos problemas que nos hayan podido retrasar o impedir el desarrollo de ciertos requisitos, para, entre nosotros o con ayuda del profesor, intentar resolverlos y poder mejorar nuestra productividad.

4.1. Hitos

Estableceremos los hitos propuestos inicialmente por el profesor, que luego sufrieron pequeñas modificaciones de fechas y de contenidos:

Fecha	Objetivos
30/09/2015	Tener una primera reunión para concretar método de trabajo y objetivos
15/10/2015	Se publican los trabajos asignados con preacuerdo
28/10/2016	Entregar el borrador del planteamiento del TFG
	Asistir al curso de escritura de trabajos académicos que da la Biblioteca
16/12/2016	Entrega de la prueba de concepto y del estudio previo del TFG
20/01/2016	Entrega de la primera versión funcional y borrador de su documentación técnica
10/03/2016	Entrega de la versión *alpha*, para probarla *in situ* nosotros
7/04/2016	Entrega de la primera versión *beta* con sus pruebas
12/05/2016	Entrega de la segunda versión *beta* con sus pruebas y el primer borrador completo de TFG
6/06/2016	Entregar el borrador completo final de todo el TFG al profesor y *release candidate* del software

14/06/2016	En caso de haber sido considerado APTOS según el profesor, llevar a encuadernar la memoria en rústica
17/06/2016	Depositar el TFG APTO finalmente en la Secretaría
20/06/2016	Ensayo de la exposición pública del TFG ante el grupo de investigación
28/06/2016	Realizar la exposición pública del TFG (fecha aproximada)

4.2. Metodologías

Hoy en día hay muchas metodologías para modelar y diseñar sistemas multi-agente. Algunos ejemplos son:

INGENIAS: es una metodología que incorpora una herramienta para desarrollar sistemas multi-agente que sean compatibles con dicha metodología. INGENIAS promueve un enfoque basado en modelos basados en el uso de INGENME. INGENME se utiliza para producir un editor visual para los Sistemas Multi-Agente (MAS, del inglés *Multi-Agent System*). Las especificaciones del MAS se procesan para producir código de programación, los documentos HTML, u otros productos. INGENIAS aborda las cuestiones de ingeniería de ida y vuelta, así, por una estructura de carpetas de concreto y una herramienta de migración de información de código de especificación.

PROMETHEUS: La metodología PROMETHEUS es ampliamente aceptada en la comunidad de investigación. Está destinada a ser una metodología práctica. Como tal, pretende ser completa: proporcionar todo lo que se necesita para especificar y diseño de sistemas de agentes, con un lenguaje bastante visual.

Esta metodología separa tres fases claramente identificadas.

1. Especificación del sistema: primera fase en la que se identifican las funcionalidades básicas que va a necesitar el sistema, pensando en un futuro donde estarán instanciados los agentes. Se pensará cómo interactúa el sistema con el entorno, las percepciones y las acciones sobre el entorno, etc…
2. Diseño arquitectónico: fase intermedia en la que se determinan qué agentes contendrá el sistema y cómo va a ser sus interacciones. Aquí se consideran el acoplamiento y la coherencia entre agentes.
3. Diseño detallado: fase final donde se detallan más minuciosamente las características que va a tener cada agente y se comienza con su diseño. Se desarrolla la estructura interna de cada agente y se especifican sus planes, metas, eventos, etc.

Al ser una metodología muy básica e intuitiva, decidimos seguirla, y, aunque PROMETHEUS proporciona una herramienta de software (*Prometheus Design Tool*) (Padgham & Winikoff, 2004) que ayuda a gestionar esta metodología, no la hemos utilizado.

Respecto a la división en tres partes de esta metodología, esto no supone una división equitativa del tiempo, puesto que en nuestro caso, la primera fase de especificación del

sistema supuso un mayor tiempo, ya que hubo que pensar en un sistema comunicado entre dos plataformas.

4.3. Herramientas

Un proyecto software se apoya sobre múltiples herramientas, tanto para el propio desarrollo del proyecto, como para otras funciones secundarias.

El hecho de que nuestro videojuego requiere del desarrollo de inteligencias artificiales, así como de la creación 'más visual e interactiva' de un mapa en los que interactúan los personajes, ha hecho que sean necesarias diversas plataformas de desarrollo diferentes, con sus lenguajes correspondientes.

Dividimos las herramientas que hemos utilizado en dos grandes subgrupos: aquellas que han sido principales en el desarrollo, y aquellas que se han utilizado de apoyo, como secundarias.

4.3.1. Herramientas principales

A continuación detallamos aquellas que hemos considerado las principales herramientas para el desarrollo del sistema.

4.3.1.1. *Unity*

Unity es un software de desarrollo orientado a la creación de videojuegos. Este es el que utilizamos, como base, para la creación del mapa de nuestro proyecto.

Utilizamos *Unity* combinado con *IsoUnity* (Pérez Colado & Pérez Colado, 2014), que forman nuestra plataforma principal para el modelado del escenario de prueba, con su entorno y sus personajes, donde se muestran a los jugadores los elementos de los que consta el juego. En este lado también se utiliza una conexión con *Java* mediante *sockets*, empaquetados de igual manera mediante objetos JSON.

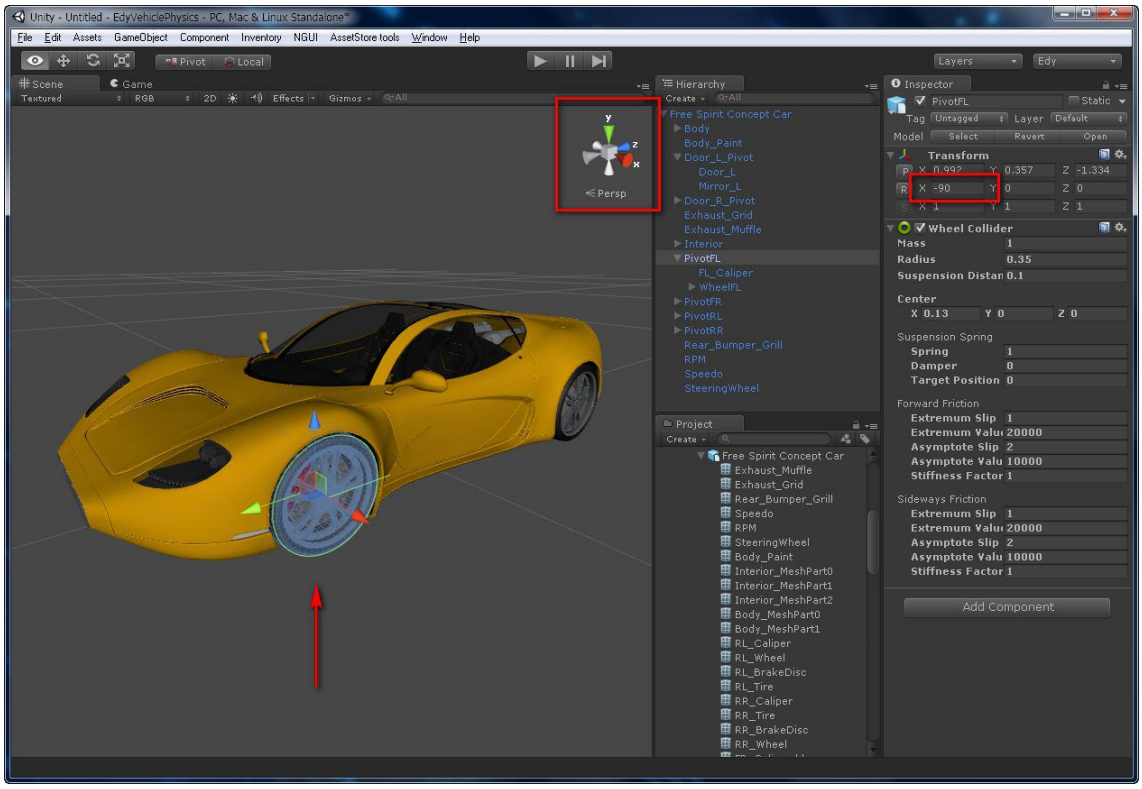

Fig. 4.1: Entorno de desarrollo de *Unity*

4.3.1.2. *IsoUnity*

IsoUnity (Pérez Colado & Pérez Colado, 2014), es una herramienta adaptada de *Unity* para el desarrollo de entornos de videojuegos en perspectiva isométrica. Desarrollada por los hermanos Pérez Colado en 2014 para su Trabajo Fin de Grado, esta herramienta permite la generación de mapas y entidades con una interfaz bastante sencilla e intuitiva.

Puesto que es una herramienta modificada de otra, podía tener fallos o carencias que nosotros pudiéramos necesitar, así que, utilizando la versión base que desplegaron en años anteriores, se le hizo unas modificaciones adaptando nuevas funcionalidades, que se detallarán en el apartado 5.

Con esta herramienta, además de la pura creación del mapa, con sus entidades, decoraciones, etc. también hay un desarrollo en la parte 'no visual' para gestionar el comportamiento de dichas entidades sobre el mapa. Agregado a eso, también está la implementación de la conexión mediante sockets para permitir la comunicación con el otro extremo de la comunicación en *Java*.

4.3.1.3. *Visual Studio*

La hemos utilizado como herramienta paralela a *MonoDevelop*, que cumple la misma función. Algunos de los integrantes del proyecto ya habían utilizado de antemano la integración de *Visual Studio* con *Unity*, lo que no hizo necesario recurrir a *MonoDevelop*.

Visual Studio ha sido utilizada como complemento a *Unity*/*IsoUnity*, que nos permite gestionar la parte trasera de *Unity*: entidades, scripts y comportamiento de la conexión. *Visual Studio* permite la incorporación del proyecto completo de *C#* para hacer más fácil su modificación e integración con los demás componentes del mismo.

4.3.1.4. *Eclipse*

Lo utilizamos como entorno de desarrollo principal para la programación en *Java*, que permite la abstracción del código distintas clases, para el desarrollo de la implementación de una conexión mediante *sockets*, por ejemplo, para poder enviar y recibir datos empaquetados en objetos, y comunicarnos por red. En el lado de *Java* de nuestra aplicación la información se empaqueta utilizando objetos JSON que pueden ser enviados a *Unity* fácilmente.

Eclipse es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma. Este es el IDE que hemos utilizado para el desarrollo de la parte de *Java* en la que se encapsula en código *Jason* y demás funcionalidades dependientes.

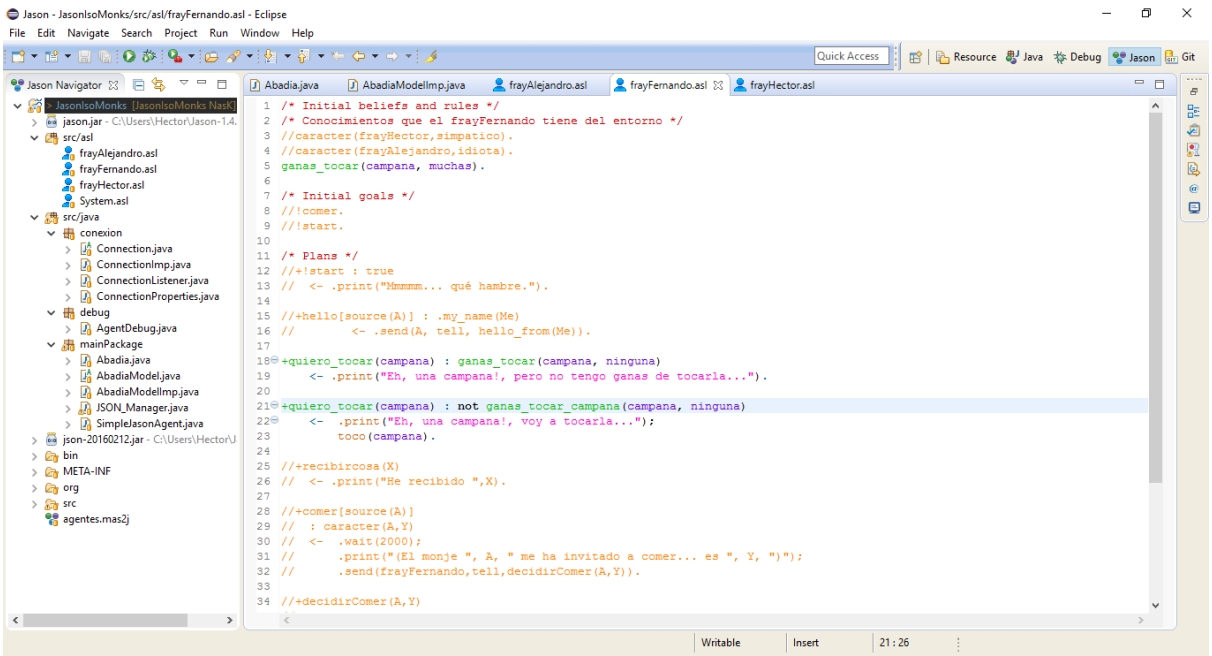

Fig. 4.2: Entorno de desarrollo de *Jason* en *Eclipse*

4.3.1.5. *Jason*

Una extensión para *Eclipse* que proporciona herramientas necesarias para la creación y gestión de agentes en un modelo BDI.

Jason es un sistema de desarrollo de agentes BDI para la creación y desarrollo de una inteligencia artificial para los personajes que se presentan en *Unity*. El *plugin* de *Jason* para *Eclipse*, permite el desarrollo de agentes BDI para esta plataforma.

Jason permite el desarrollo de agentes inteligentes en formato ASL, a los que se refiere a veces como sistemas reactivos de planificación. El lenguaje que utilizan estos agentes y que el *plugin* es capaz de interpretar.

Una vez escritos los planes, creencias e intenciones de cada agente en dichos ficheros, *Jason* los ejecutará sobre el entorno sobre el que interactuarán, tanto entre ellos como con lo que les rodea.

En nuestro caso, decidimos utilizar *Jason* ya que nos pareció ventajoso que se pueda desarrollar mediante un sencillo *plugin* de entorno, y que consista en una extensión del potente lenguaje ASL. Gracias a la unión con *Eclipse*, pudimos desarrollar fácilmente la comunicación con el entorno donde habitan los agentes y hacer que estos fueran capaces de percibir eventos que ocurren allí, en su entorno.

4.3.2. Herramientas secundarias

En este segundo subgrupo de herramientas entran editores de texto u otros entornos de desarrollo que han sido usados pero de manera muy puntual para el desarrollo de la aplicación, así como herramientas que no están relacionadas con el desarrollo, sino con la documentación, comunicación, etcétera.

***MonoDevelop*.** Herramienta proporcionada por el paquete de *Unity*, para el desarrollo de código, en *C#.* No llegó a utilizarse, salvo de manera muy esporádica.

***Sublime Text/Notepad*++**. Editor de texto que nos ayuda a perfilar el código y solventar algún problema que podamos tener en él de una manera rápida y sencilla.

JSON. Como formato textual y sencillo pero estructurado para la comunicación por red entre la parte de *Java* y la parte de *Unity*.

Dentro de estas secundarias, y orientadas más al ámbito de planificación y gestión de proyecto, comunicación y documentación utilizaremos:

***Google Drive*.** Será la principal vía de sincronización de archivos o elementos de información entre los diferentes miembros del grupo.

***Microsoft Word*.** Para maquetar y pulir la versión final de la memoria.

***Whatsapp*.** Para comunicación por mensajería instantánea entre los miembros del grupo. El uso del correo electrónico se redujo mucho gracias a esta herramienta.

***GitHub*.** Repositorio para llevar un control de las versiones y las diferentes partes o módulos del proyecto, y así que todos podamos trabajar simultáneamente en él y evitar desajustes entre las partes del código.

Estas son las direcciones donde se encuentran las dos partes del proyecto:

- *Java/Jason*: https://github.com/Kaerit/JasonIsoMonks
- *Unity/IsoUnity*: https://github.com/NasK1991/IsoMonks

Dia Diagram Editor. Editor de diagramas que nos ayudó a ver de una manera rápida las relaciones entre las diversas entidades del juego y poder desarrollar el código en sincronía con ello.

5. Contribución y resultados

Hemos tratado de construir un sistema que permita a los personajes estar dotados de inteligencia y que, dependiendo de la situación, tomen decisiones racionales en la que la Inteligencia Artificial valorará el tipo de respuesta a realizar, es decir, que le 'conviene' hacer o comunicar. Hemos tratado de abarcar todo lo posible, para comprobar cómo de útil puede ser esta tecnología, en el desarrollo de los acontecimientos planteados en un videojuego.

Los apartados de este capítulo son los siguientes. La sección 5.1 es una especificación del escenario de ejemplo que vamos a utilizar para ilustrar la utilidad del sistema.

La sección 5.2 es allí donde se explica cómo desde el punto de vista de la Ingeniería del Software cubrimos el análisis y diseño del sistema, siendo el foco principal el construir la arquitectura del mismo integrando dos sistemas que en principio están separados: sistema MAS de *Jason* y escenario de videojuegos isométrico y de estilo retro de *IsoUnity* (Pérez Colado & Pérez Colado, 2014).

Finalmente la sección 5.3 habla sobre cómo generamos el entorno y los personajes así como detalles de los mismos.

5.1. Especificación: La experiencia narrativa de ejemplo

A continuación se presenta el escenario principal de nuestro proyecto.

El ejemplo inicia con el personaje de '*frayHector*' en movimiento, iniciando una rutina diaria implementada a través de *Jason*. Este personaje, nada más empezar en su habitación, se dispondrá a ir a la rezar a la capilla, después de estar un tiempo 'rezando' en la misma, proseguirá yendo a la cocina a trabajar en ella, después irá al comedor y terminará su rutina volviendo a la habitación donde empezó.

El personaje de '*frayHector*' repetirá esta rutina indefinidamente, a no ser que el jugador, controlando a '*frayFernando*', interfiera en ella, es de esta manera donde se puede observar la lógica del comportamiento de *Jason*.

Si el jugador obstaculiza su comportamiento, de manera que este necesite ingeniárselas para volver a su rutina, en el ejemplo observamos como el personaje de '*frayFernando*' 'cerrara la puerta' que da acceso a la cocina, colocándose en la casilla que está al lado de la misma. De esta manera el personaje de '*frayHector*' interrumpirá su rutina y luego buscará la manera de volver a ella.

Para poder hacerlo, a '*frayHector*' se le añadirá una 'meta' de *Jason* que consistirá en encontrar la llave que abre la puerta de la cocina. Dicha llave la tiene '*frayFernando*' en su escritorio, de tal manera que se dispondrá en ir hacia allí y la buscará. Una vez conseguida, ya podrá abrir la puerta, por lo que volverá a ir a la cocina, pero esta vez con el fin de abrirla. Después de estar un rato intentando abrir, volverá a entrar en la cocina y retomará la rutina preestablecida desde un inicio.

Otro comportamiento de los agentes es por parte de '*frayAlejandro*'. Dicho agente está en su habitación, y cuando se entera de que '*frayFernando*' ha entrado en la cocina, él también irá. El agente estaba esperando a que el jugador entrase en la cocina para poder seguirle.

5.2. Análisis y Diseño: Integrando *Jason* con *IsoUnity*

Como se dijo anteriormente, uno de los principales retos fue la integración de las plataformas *Jason* y *Unity* para la comunicación entre ellas. Mientras que *Unity* sería el encargado de mostrar la parte visual del videojuego, *Jason* sería el 'cerebro' de los personajes. Por ello era importante dedicarle tiempo a buscar una buena implementación que permitiera el paso de información entre dichas plataformas.

Desde un primer momento se pensó que la mejor manera era implementar dos servidores y dos clientes UDP que transmitieran la información mediante *sockets.* Se implementó entonces un servidor y un cliente en *Java* y otros iguales en *C#*. Con ello, y mediante la asignación de puertos, se consiguió que se mandaran información simple. Esto nos permitía en un primer momento el paso de comandos simples desde *Jason* a *Unity*. Comprobamos que, cuando movíamos un personaje en *Unity*, en *Jason* recibía la información y nos la iba mostrando por pantalla. Todos los movimientos quedaban registrados.

Según fue avanzando el desarrollo, se iba necesitando más información, que con la implementación inicial del cliente-servidor UDP, no podríamos conseguir. Al menos no de la forma en la que lo teníamos pensado. Por ello, en vez de mandar mensajes simples, empezamos a mandar información más rica y a utilizar la sintaxis de JSON. Con él comunicábamos a *Jason* o a *Unity* todo sobre el personajes y su entorno: acción que trata de realizar, evento en el caso de que lo hubiera, etc. y así podríamos generar reacciones en *Jason* a las acciones recibidas. La comunicación por red genera otro problema. Al mandar eventos, el servidor quedaba a la espera de recibir algo que nunca llegaba, por ello se rediseñó el cliente-servidor para que fuera controlado por hilos. Al iniciar la aplicación se crean dos hilos paralelos que controlan la conexión y el paso de información entre plataformas sin que se queden esperando a algo que no llega nunca.

De forma muy esquemática mostramos a continuación el diagrama de secuencia que sigue el envío de un evento desde *Unity* a *Jason*.

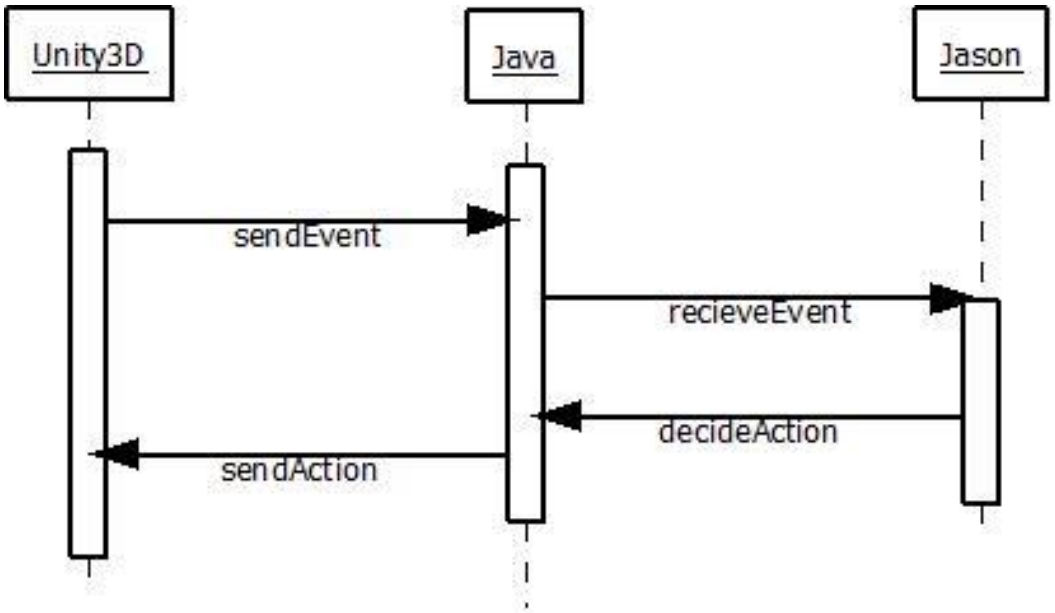

Fig. 5.1: Diagrama de secuencia que muestra el envío de eventos desde *Unity* a *Jason*

5.3. Implementación: Producción del escenario y los personajes del videojuego

El diseño y producción del escenario se realizó con la herramienta *IsoUnity* (Pérez Colado & Pérez Colado, 2014), la cual permite generar mapas isométricos de manera rápida. Dicha herramienta es importada a *Unity* como un proyecto nuevo, y a partir de ahí, aparecerán en el menú las nuevas opciones de sus herramientas.

Con esto preparado, se fue generando bloque a bloque todo el escenario. Con la herramienta *IsoUnity* se pueden crear distintas alturas para generar los muros por los cuales los personajes no pueden pasar. Dichas alturas debían ser superiores a 2, dado que los personajes podrían saltar los bloques de altura 1. Aclarar que los bloques de altura 0 no se utilizan, ya que no existía colisión con los personajes y estos no podían apoyarse en ellos. Por esto se diseñó el mapa con el suelo de bloques de altura 1 y los muros con bloques de altura 3.

IsoUnity genera los bloques, pero se quedan en un color básico. Una vez teníamos el modelo, habría que irle añadiendo las texturas a los bloques. Además de las texturas, se le añaden las decoraciones a ciertos elementos del mapa, tales como las lámparas o incluso los personajes.

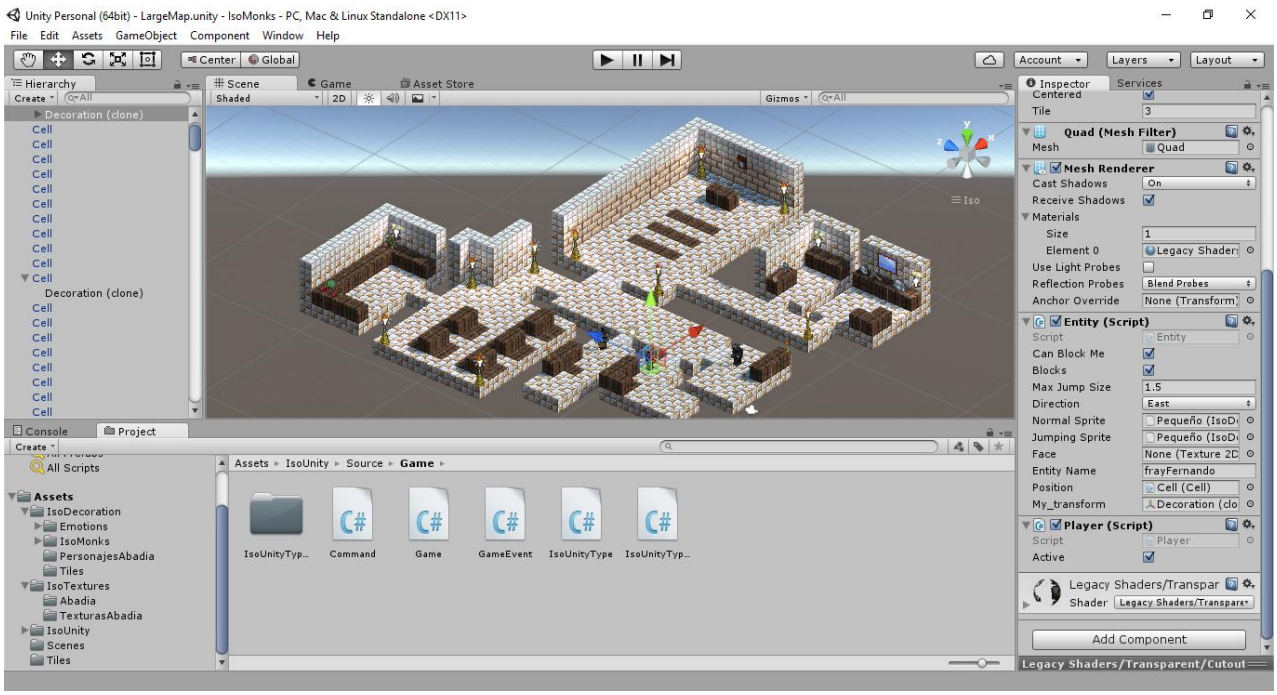

Fig. 5.2: Entorno de *IsoUnity* con el escenario de la Abadía

Los personajes son decoraciones, pero con entidades asociadas, lo que quiere decir que pueden tener funcionalidad. Además todas las decoraciones y entidades tienen un nombre, un identificador propio, que aprovecharemos para que *Jason* pueda reconocerlos rápidamente. De esta manera el código de *Java* y *Jason* pueden mandar órdenes para que *Unity* reconozca que personaje está hablando o hasta qué punto del escenario (justo donde una decoración) debe moverse.

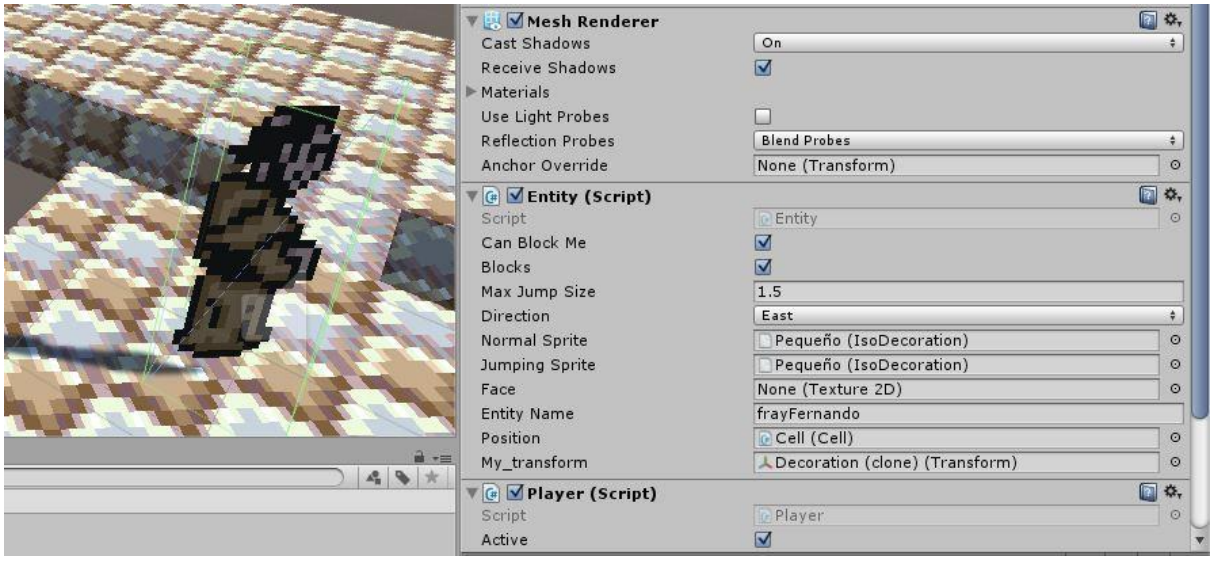

Fig. 5.3: Decoración de *Unity* con *Entity Name* '*frayFernando*'

En el mapa, hemos localizado diversos puntos de referencia para los personajes, que representan lugares en los que cada uno desarrolla su rutina, lugares en los que se producirán eventos, habrá puntos de luz, etc. A continuación, mostramos un listado para que el lector se pueda hacer una idea de cómo está estructurado el espacio.

Fig. 5.4: Planta de la abadía con las localizaciones destacadas

1. Puntos clave para '*frayAlejandro'* (de color rojo en el mapa)
   A. Localización donde trabaja en el taller (*fAlocation_taller*)
   B. Localización de su mesa del cuarto (*fAlocation_mesa_cuarto*)
   C. Localización donde reza en la capilla (*fAlocation_capilla*)
   D. Localización donde come (*fAlocation_comedor*)

2. Puntos clave para '*frayHector'* (de color en verde en el mapa)
   A. Localización donde reza en la capilla (*fHlocation_capilla*)
   B. Localización donde come (*fAlocation_comedor*)
   C. Localización donde come (*fHlocation_comedor*)

D. Localización de su mesa del cuarto (*fHlocation_mesa_cuarto*)
E. Localización de la llave de la cocina (*fHlocation_llave*)
F. Localización de la puerta de la cocina (*fHlocation_puerta_cocina*)

3. Puntos de eventos comunes para todos (de color azul en el mapa)
   A. Hacer sonar la campana para llamar a rezar (*campana_cerca*)
   B. Cerrar la puerta de la cocina (*cerrar_puerta*)
   C. '*frayAlejandro'* te sigue cuando entras al comedor (*entrar_comedor*)
   D. Hacer trabajar a los demás monjes (*hora_trabajar*)

En el ejemplo de código de a continuación tenemos algunos de los distintos métodos y propiedades de *Jason*. El agente '*frayFernando*' tiene un *initial belief* llamado *ganas_tocar*, el cual toma como parámetro la campana, y sus ganas, que son muchas. Como metas iniciales, no tiene ninguna, ya que se irán generando automáticamente según el agente vaya tomando decisiones.

Sus planes son *quiero_tocar* y como parámetro toma una campana. Los dos planes se llaman igual, pero uno es el procedimiento a seguir en el caso de que sus ganas de tocar la campana sean distintas de ninguna. Dichas ganas, como hemos dicho antes, van definidas en el *initial belief*.

Uno de los primeros pasos que dimos con el agente '*frayFernando'* era que tenía que tocar una campana. Pero el hecho de tocarla iba condicionado por las ganas que tuviese en ese momento de ello, a continuación está el fragmento de código que controla esa situación para los casos de que tenga ganas de tocarla, y cuando no lo tenga:

```
/* Initial beliefs and rules */
/* Conocimientos que el frayFernando tiene del entorno */
ganas_tocar(campana, muchas).


/* Initial goals */


/* Plans */


+quiero_tocar(campana) : ganas_tocar(campana, ninguna)
   <- .print('Eh, una campana!, pero no tengo ganas de tocarla...').

+quiero_tocar(campana) : ganas_tocar(campana, muchas)
   <-     .print('Eh, una campana!, voy a tocarla...');
        toco(campana).
```

Fig. 5.5: Ejemplo de código de *Jason* para '*frayFernando'*

5.4. Pruebas: Experimentos con varias ejecuciones de la experiencia

En esta sección exponemos algunas pruebas por las que hemos ido pasando para la realización e implementación final del proyecto, etapa que llegó una vez terminada toda la formación y la integración tecnológica realizada en el primer cuatrimestre.

Comunicación de entornos, Diciembre 2015 - Enero 2016: sopesamos diferentes maneras de comunicar los entornos, e intentamos utilizar *TinyWo*rld (Pérez Colado & Pérez Colado, 2014) para mandar información de *Unity* a *Eclipse*. Tras varios intentos fallidos de conexión con este modo, al ser mucho más desarrollado de lo que nosotros necesitábamos, optamos por hacer nuestras clases para el envío y recepción por *sockets*.

Estructuración de datos, Diciembre 2015 – Enero 2016: a la vez que fuimos trabajando en la comunicación, pensamos como estructurar las clases que contendrían todos esos datos. Primeramente se enviaban información de entidades, decoraciones, celdas… de manera independiente, pero daba problemas de concurrencia, así que finalmente decidimos agrupar todo ello en un mismo datagrama para enviarlo de una vez, y así almacenarlo en las estructuras eliminando el problema de concurrencia.

Eventos, Febrero 2016: La idea inicial era hacer que los agentes respondieran y ejecutaran acciones en base a unos eventos lanzados por el jugador. Para ello se prepararon varias casillas del mapa con *triggers* para que, cuando '*frayFernando'* pasara por encima de ellas, se lanzaran distintos eventos.

Agentes, Febrero – Marzo 2016: Para poder acceder a las propiedades de un agente desde *Java*, tuvimos que ejecutar distintas pruebas con las librerías de *Jason*. Entre estas pruebas estaba la actualización de la *Belief Base*. *Java* podía cargar el agente con una función, y así acceder a su *Belief Base* y las demás propiedades del agente. Sin embargo, a la hora de actualizar la *Belief Base*, tuvimos que investigar cómo funcionaba dicho procedimiento a bajo nivel. Finalmente averiguamos que la *Belief Base* se actualiza, no desde *Java*, si no al modificar las percepciones de los agentes.

Asociación eventos y agentes, Marzo 2016: Una vez finalizada la generación de eventos, intentamos introducir su asociación con los agentes. Se elaboró un esquema de fichero en JSON, en el cual se asocia a los agentes. Se les añade una percepción que les modifica la base de creencias y le hace actuar acorde a lo asociado en el JSON.

Creencias y percepciones, Marzo-Abril 2016: Para poder modificar la base de creencias a través de *Java* estuvimos intentando genera un agente fantasma en el cual gracias a sus métodos pudiéramos acceder a los cambios de los *percepts* del entorno. Esto era generar un objeto *Java* mediante el método que carga los ASL de *Jason*. Con dicho objeto, podríamos obtener sus propiedades, y comprobar que todo se iba actualizando. Finalmente se creó un *percept* asociado a cada agente en el cual, cada uno se modifica así mismo su base de creencias.

Rutinas, Abril – Mayo 2016: La creación de rutinas viene muy ligada a la generación de las percepciones y los cambios en su base de creencias. Mediante generaciones de eventos los agentes intentan conseguir sus metas marcas y es de esta manera por la cual ellos empiezan moverse y localizar objetos para satisfacer las metas impuestas. Una vez que completa su meta, se volverá a añadir al agente una nueva percepción, en la cual, ha conseguido algo anteriormente mediante la meta. Con ello, se volverán a ejecutar los planes, y esta vez de manera distinta, ya que sus percepciones y sus creencias, han cambiado.

Intercepción de rutinas, Mayo 2016: Desde el punto de vista del desarrollo, la intercepción de rutinas hay que tener en cuenta que la modificación de un agente implica la asociación de otro *percept* que modifica a su base de creencias y hay que tener en cuenta que cuando la rutina sea interrumpida puede generar estados en los agentes que hay que tener presente. Con esta idea el jugador puede interrumpir la rutina deseada y generar nuevas situaciones en el juego.

6. Discusión

En este capítulo se discuten los resultados obtenidos y expuestos en el último apartado del capítulo anterior (5.4), así como otras ideas en torno al sistema desarrollado, comparándolo con otras posibles soluciones.

6.1. Discusión sobre los resultados obtenidos

Los videojuegos no tienen por qué basarse en una historia lineal, prediciendo con anterioridad todos los caminos posibles por el cual ha de pasar el jugador, sino que son los personajes del videojuego y el propio jugador quien va descubriendo y creando su propia historia. A día de hoy poco se ha explotado este nuevo sistemas en videojuegos narrativos, y aunque es un objetivo ambicioso también es muy prometedor el hecho de preestablecer una entidad y poder dotarla de herramientas para que, junto con el jugador, vayan forjando una relación con los demás personajes y alrededor del escenario de juego, tratando de alcanzar unas metas hasta llegar al final de la historia.

Los finales de los videojuegos deberían depender siempre del transcurso del mismo, diversas maneras de actuación podrían desembocar en un mismo final, pero no debería ser lo habitual. Con estos resultados intentamos hacer una propuesta tecnológica para que en un futuro puedan conseguirse mejoras significativas en los videojuegos y hacer que el usuario sienta que verdaderamente forma parte de la historia. No queremos apoyarnos tanto en el llamado *efecto Eliza* sino trabajar en que los demás participantes del mismo, esos personajes no jugadores, actúen de manera lógica e independiente como entidades realmente autónomas, en definitiva, como si fueran otros jugadores humanos o al menos 'actores virtuales' que interpretan un papel en beneficio del argumento.

Como bien se exponía en las especificaciones, los personajes tienen una inteligencia artificial que les da la capacidad de reaccionar activamente frente a los acontecimientos que sucedan. Sin embargo, la historia que se puede contar en el videojuego está limitada a la que es, es decir, no es una historia abierta y dinámica.

La utilidad de este sistema sería en la generación de historias abiertas que hicieran que el juego fuese cambiante en función de las decisiones de los personajes. Un ejemplo podría ser el siguiente: situándonos en un marco medieval, el personaje llega a un pueblo donde es forastero y se le pregunta por qué está allí. Supongamos que la opción que elige es '*Vengo a derrotar al dragón que aterroriza estas tierras'*. Esta respuesta haría que se re-escribiese el futuro de lo que pasará en el videojuego: ahora el personaje se moverá en una historia donde los demás le tratan como un 'salvador' que ha venido a ayudarles, o le prometen riquezas y poder en recompensa. Tras recorrer cavernas y zonas de terrenos hostiles en búsqueda de ese enemigo, ahora el desenlace ideal de esta historia sería luchar contra ese dragón y vencerlo.

Pero imaginemos que esa no es la respuesta, sino que responde '*Vengo a desposarme con la princesa del reino*'. Ahora el futuro del juego no será ir en búsqueda del dragón

por diversas mazmorras, sino (asumiendo que no estamos ante una historia de amor) hacer ‘gestiones internas’ para conseguir ese matrimonio de conveniencia, tratando de demostrar al rey y a sus políticos de confianza que semejante enlace les conviene, ya sea por poder, dinero, alianzas estratégicas, etc.

6.2. Discusión sobre el sistema propuesto

En el apartado 2.1.1 vimos ejemplos de videojuegos que utilizan métodos de inteligencia artificial para dar una mejor experiencia al usuario, sin embargo, vamos a compararlos con la tecnología de agentes inteligentes para ver hasta qué punto podrían mejorar.

Splinter Cell Blacklist (Ubisoft Toronto, 2013) **-** Este título, a pesar de estar centrado en el sigilo, tiene muchos puntos en los cuales se necesita de llaves u otros artilugios para avanzar en la trama. Supongamos pues que los NPC’s tienen una ‘mentalidad propia’ la cual les permite ir a conseguir sus metas. Cada persona jugará el juego de modo distinto. Si un jugador elimina a un NPC el cual tenía pensado ir a ver a nuestro objetivo, y a causa de eso el NPC no llega a la hora, el otro se enfadará y se irá a su habitáculo y cerrará con llave. Ahora nos encontramos en una situación en la cual tendremos que buscar una llave para poder abrir la habitación. Una decisión nuestra ha interferido en la meta de un NPC y este ha cambiado su objetivo, cambiando la partida por completo. Este es un ejemplo muy básico, pero dada la flexibilidad del sistema, se podría complicar mucho más.

Max Payne 3 (Rockstar Games, 2012) **-** Como dijimos es un *shooter*, el cual podría ver su jugabilidad incrementada con la implantación de un sistema de agentes BDI, que permitiera a los enemigos esquivar y alejarse de la línea de fuego del jugador. Como se puede apreciar en este título, el jugador va de frente disparando a los distintos enemigos, los cuales siguen disparando a su objetivo sin moverse demasiado. Si de una oleada de enemigos, algunos fueran capaces de ver de dónde vienen los disparos, analizar el peligro y dar un rodeo para disparar por detrás, incrementaría la dificultad y haría que los jugadores no pudieran ir tanto tiempo de frente, si no teniendo que vigilar sus espaldas.

GTA V (Rockstar North, 2013) **-** Una parte *guionizada* del *GTA* son las acciones de los personajes no jugables que nos encontramos en el mundo abierto. Además de repetirse su fisonomía y sus atuendos (por una cuestión de economía de recursos muy habitual en videojuegos), se repiten también sus acciones. Aunque son muy variadas y es difícil darse cuenta de ello, basta con seguir a uno de manera prolongada para darnos cuenta de ello. Si dichos personajes estuvieran implementados con agentes BDI, daría más dinamismo al juego, ya que podrían realizar distintas acciones, y hacer que dichas acciones interfieran en las acciones de otros personajes. De esta manera podría ocurrir que un personaje, por no haber podido conseguir su objetivo, esté en medio de nuestra misión principal y nos la haga un poco más fácil, o un poco más complicada. Diferente en cualquier caso.

Killzone Shadow Fall (Guerrilla Games, 2013) **-** Al igual que en *Max Payne 3*, al ser un *shooter*, los enemigos implementados con un sistema de agentes inteligentes, serán capaces de flanquear al jugador para intentar acabar con él. Podrían aprovechar elementos del entorno tal como bidones explosivos u otro tipo de objetos para maximizar el daño que le producen al jugador.

Si hablamos ahora sobre los problemas que hemos tenido a la hora de la realización de este trabajo ha sido la falta de información y ejemplos de *Jason* de entornos complejos por parte de su página oficial, a pesar de que tenían información de estructuras básicas resulto ser escasa para la hora de realizar ejemplo y pruebas para poder evaluar un código más bien complejo. Ejemplos con interfaces más elaboradas de 3D nos hubieran ahorrado mucho tiempo de pruebas.

Uno de nuestros grandes problemas debido a la falta de información fue el desarrollo de las rutinas de los personajes trabajando con su base de creencias. Mediante experimentos y pruebas, conseguimos interactuar con el agente para generar dinamismo en su comportamiento.

También el hecho de juntar dos entornos de desarrollo diferentes (uno mucho más visual y grande en cuanto a posibilidades, como *IsoUnity* (Pérez Colado & Pérez Colado, 2014); y otro más lógico y con un paradigma declarativo como con *Jason*) fue un gran reto, ya que teníamos que repasar y estudiar toda la estructuración de las clases y la gran cantidad de documentación imprescindible para manejarse con ambos entornos, lo que supone una inversión de tiempo considerable en formación.

Con respecto a la herramienta *IsoUnity* hubo que ampliarla y añadir ciertas funcionalidades para garantizar el correcto funcionamiento de los resultados obtenidos, como son el desarrollo de un generador de eventos y marcador de posiciones. Gracias al apoyo de nuestro director, Federico Peinado, de los desarrolladores de la herramienta de *IsoUnity*, los hermanos Pérez Colado, y de los desarrolladores Alexiades y Santamaría, pudimos implementar estas nuevas funcionalidades.

Algo que ha sido necesario mejorar constantemente fue el escenario del juego. A lo largo de toda la realización del trabajo esta tarea ha sido recurrente para poder adaptar el entorno a las nuevas funcionalidades y para poder realizar las pruebas lo más rigurosas posibles. El mapa fue evolucionando, desde un pequeño mapa sin ninguna decoración de 3 x 3 hasta el resultado final obtenido, el problema planteado no por *Unity* sino por *IsoUnity* es la ocupación en memoria de las imágenes digitales creadas a partir de un modelado 3D de cada objeto, la optimización por parte de *IsoUnity* de este aspecto todavía es algo precaria y primitiva. Por lo tanto había que tener cuidado para que el ordenador pudiera soportar el escenario, y por eso se decidió no hacerlo demasiado grande. Ahora mismo el tamaño del mapa con el espacio ocupado en disco no se relaciona de manera lineal, sino cuadrática así que había que tener en cuenta este hecho.

La generación de eventos fue algo también que necesitábamos mejorar y un desarrollo muy importante para empezar una relación entre *Unity* y *Java*. Uno de los eventos más

importante que tenemos es la inicialización del mapa en *Java*, que consiste en generar un evento en el momento de la inicialización de *Unity* en cual genera una estructura de JSON con la información necesaria del entorno. Esta estructura es enviada a través de un *socket* (que se explicara más adelante su desarrollo y funcionamiento) a *Java*, para que este tenga en cuenta los elementos del mismo y poder actuar acorde al entorno. Este generador de eventos también es utilizado para que los personajes puedan, en cierto modo, comunicarse entre sí, de tal manera que si uno hace sonar una campana (evento) el resto de personajes puedan percibir este cambio del entorno.

Para la generación de eventos necesitábamos una pequeña ampliación que no estaba contemplada de *IsoUnity* (Pérez Colado & Pérez Colado, 2014) y que supuso un problema y, por lo tanto, tuvimos que realizar nosotros. No hubiéramos podido seguir hacia delante sin esta pequeña implementación adicional que consistió en un marcador de posiciones, para poder mandar a un personaje a una ubicación exacta en el mapa, de tal manera que pudiera reconocer una ubicación a través de su nombre para así dirigirse a ella.

Como expresamos anteriormente por parte de *Jason* consideramos que la información fue algo escasa para hacer el tipo de desarrollo que nosotros buscábamos, y es donde más tiempo hemos tenido que invertir a la hora de empezar a ver resultados.

La sincronización de los ficheros de configuración de *Jason* supuso otro problema en la realización de este trabajo. Ya que *Eclipse* escribía las rutas absolutas de los ficheros con las dependencias a *Jason*[11], en sus archivos de configuración, entonces, a la hora de lanzar *Jason* en un ordenador, si lo había ejecutado otro antes, *Eclipse* leía una ruta incorrecta (la del otro ordenador) y al no encontrar los ficheros, daba lugar a errores y había que volverlo a modificar manualmente la configuración.

Se generaron clases para la extracción de información para el entorno y para los agentes, de esta manera pudimos observar más detenidamente los datos contenidos en estas estructuras para poder utilizarlas como realmente necesitábamos.

Cabe mencionar que la relación en sí de ambos entornos también supuso mucho trabajo. Para conseguir y mantener esta relación fue necesario mandar datos a través de una comunicación por red implementada mediante sockets. Realizamos una estructura de cliente-servidor en el cual *Unity* generaba información, la cual, se transfería a *Java*, que elaboraba una respuesta y reenviaba esta información de vuelta a *Unity*. En la conexión mediante *sockets*, por ejemplo había que tener en cuenta las reglas de entrada y salida del firewall del sistema, ya que algunos ordenadores por defecto traen filtros de seguridad para evitar la intromisión indeseada de programas ajenos desconocidos. Otro problema en hemos tenido a lo largo de todo el proyecto relacionado con el *socket* de información fue que el cierre de la aplicación en un momento crítico en la elaboración de la respuesta por parte de *Java*, comprometía el uso del socket, lo que nos hacía ver que este seguía en uso (no se había cerrado debidamente) cuando en realidad la

[11] Esto se refiere a las rutas del *classpath* de *Java*

aplicación no estaba haciendo uso de él, de esta manera la aplicación dejaba de funcionar con normalidad, la vuelta al estado inicial del *socket* implicaba el cierre de la sesión del ordenador o incluso el reinicio del mismo.

Como lo ya mencionado en *Unity* en relación al espacio ocupado en memoria principal y disco, el lanzamiento de nuestro proyecto implicaba ejecutar simultáneamente el entorno visual de *Unity* con el entorno lógico de *Java*. En comparación *Java* ocupa menos que *Unity* pero este lanzamiento simultaneo implicaba tener una buena capacidad de memoria para funcionamiento correcto y fluido de la aplicación, de lo contrario no solo se podían observar fallos visuales, como saltos de los personajes hacia posiciones más adelantadas a las que en realidad deberían ser, etc. sino que funciones internas de *Jason* las cuales utilizan contadores de tiempo real se veían afectadas, de tal manera, que debido a la latencia, si una función debía estar parada 10 segundos, y luego continuar, esta pausa no se realizaba o se reducía, y daba la sensación que el personaje no se paraba o no hacía lo que debía correctamente.

7. Conclusiones y trabajo futuro

En este capítulo exponemos cuáles son nuestras conclusiones y que posibles líneas de trabajo futuro planteamos para darle continuidad al proyecto.

7.1. Conclusiones

Una de las carencias que tienen muchos videojuegos hoy en día es la falta de inteligencia artificial en muchos de los personajes. Es esta falta de inteligencia lo que hace que los juegos se puedan volver repetitivos, e incluso aburridos. Nuestra motivación principal era desarrollar un sistema que, apoyado en la inteligencia artificial, diese más dinamismo a los NPC.

Es cuanto a esa 'carencia' de Inteligencia Artificial, es una falsa apariencia en muchos videojuegos, un truco que usa el diseñador para convencer al jugador. En la mayor parte de los videojuegos se pretende conseguir que, mediante *scripts* los personajes den la sensación de que 'piensan'. Sin embargo, al manejarlo todo mediante *scripts* nunca se puede llegar a cambiar el transcurso de la historia si siempre se ejecutan las acciones de la misma manera.

Con un modelo BDI se consigue que los agentes implementados sean capaces de conseguir sus metas establecidas mediante la ejecución de planes y la variación de sus creencias, de una manera mucho más similar a cómo funciona la cognición humana.

Nuestro proyecto aspira a facilitar el desarrollo de personajes verdaderamente inteligente, capaces de interactuar entre ellos y utilizar su entorno de manera planificada y lógica. Esto, como ya se comentó, hace que se dé dinamismo a la historia, haciendo que los personajes actúen de una manera similar a lo que podría ser la realidad, actuando conforme a sus objetivos, pudiendo mentir o actuar de una manera u otra con el fin de obtener su objetivo final.

Gracias a las pruebas de desarrollo que hemos llevado acabo y los resultados obtenidos en ellas, hemos podido ilustrar cómo la implementación de estos agentes como solución al control autónomo de personajes en videojuego puede ser una buena opción para una futura implementación en la Inteligencia Artificial de los personajes no jugables en videojuegos.

Habiendo conseguido los requisitos principales de nuestro proyecto, que eran proponer una solución de Inteligencia Artificial que pudiera integrarse en un entorno virtual isométrico y de estética retro, y habiendo construido un escenario ilustrativo de videojuego, nos damos cuenta de que todavía falta trabajo para poder tener un producto comercial que confrontar al público real. Nos hemos dado cuenta de posibles mejoras que podrían añadirse en un futuro.

Después de haber finalizado el proyecto pensamos que implementaciones de inteligencia artificial de este estilo, ya sean modelos BDI u otros, podrían dar mucho dinamismo al mundo de los videojuegos, haciendo que la experiencia de juego sea mucho más satisfactoria para el jugador. Sin embargo la implementación de este modelo, al estar desarrollado con un lenguaje declarativo (*AgentSpeak*), resulta bastante complicado al no ser tan intuitivo como otros lenguajes de programación. Además, estos lenguajes declarativos deben controlar las posibilidades en el que el estado del personaje este y suponen un control exhaustivo de este estado.

7.2. Trabajo futuro

La visión de futuro que planteamos nosotros en el mundo del videojuego es, además de integrar estas inteligencias en los personajes, crear historias generadas dinámicamente en ellos. Que la historia no esté escrita desde el principio sino que se vaya generando según el jugador avance en el tiempo, y según las acciones y respuestas que pueda dar a otros personajes que interactúen con él.

Esta idea podría ser muy interesante para juegos con más de un final posible. Los cuales dichos finales se irán acercando, no sólo en base a una serie de decisiones por parte del jugador, sino también a las decisiones que tomen los propios personajes no jugables. Es interesante ver como la historia se modifica, incluso a veces cuando el jugador no sepa qué ha ocurrido, teniendo que intentar averiguar qué ha cambiado para que pueda seguir con la historia. De esta manera cada vez que una persona juegue la historia, será distinta si toma alguna decisión diferente.

Además, por la toma de decisiones y las decisiones que tomen los personajes no jugables, es interesante ver que se podrían conseguir dos finales iguales, pero con decisiones distintas. Esto se conseguiría por los caminos que tomen en el árbol de decisiones.

Los ejemplos expuestos en el apartado 6.1 son ejemplos muy sencillos que simplemente parecen ofrecer una bifurcación en el camino, pero no es simplemente eso, ya que el sistema lo que permite es reestructurar todo lo que puede pasar en el juego y de esta manera acabar de una manera u otra. Incluso no necesariamente que el final sea distinto, si no que la historia cambie lo suficiente como para tener que cambiar la mentalidad del jugador para poder continuar por el camino que quiere.

Con una simple opción binaria se abren dos alternativas, pero por cada alternativa se pueden abrir otras más, y de esas otras muchas más, haciendo que cada vez que se juegue, lo que ocurre es diferente; no solo en la historia que se presenta, sino en los escenarios, que se podrían generar de manera también dinámica acorde a lo que fuese a pasar, algo que se está estudiando ya como propuesta para un Trabajo Fin de Máster.

Con nuestro trabajo damos un primer paso apostando por estos planteamientos y proponiendo posibles soluciones y distintas implementaciones para algunos de los

videojuegos comerciales que actualmente existen en el mercado. El siguiente paso podría ser integrar estos personajes tan dinámicos en un producto real, para abrir todo un mundo de posibilidades en el desarrollo de videojuegos.

Actualmente esto, podría ser muy interesante, además, con el actual desarrollo de distintas técnicas de realidad virtual, las cuales están modificando por completo el mundo de los videojuegos.

8. Referencias

André, P. D., & Kurdyukova, D. E. (2009-2011). IRIS. | http://iris.interactive-storytelling.de/.

Blizzard Entertainment. (1994). Warcraft. Blizzard Entertainment.

Blizzard Entertainment. (21 de Enero de 2009). StarCraft. *StarCraft*. Blizzard Entertainment.

Bordini, R. H., Hübner, J. F., & Wooldridge, M. (Octubre de 2007). *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley | http://io.acad.athabascau.ca/~oscar/ebook/Jason.pdf .

Bratman, M. E. (Enero de 1999). Modelo BDI. *Intention, Plans, and Practical Reason (The David Hume Series)*. CSLI.

Crytek. (16 de Noviembre de 2007). Crysis. Electronic Arts.

Electronic Arts. (Enero de 2000). Los Sims. Electronic Arts.

Ensemble Studios. (26 de Octubre de 1997). Age of Empires. Microsoft Studios.

Epic Games. (Marzo de 2016 de 1998). Unreal Engine. 31: Unreal Engine.

Gearbox Software. (Octubre de 2009). Borderlands. 2K Games.

Gearbox Software. (3 de Mayo de 2016). Battleborn. 2K Games.

Guerrilla Games. (15 de Noviembre de 2013). Killzone: Shadow Fall. Sony Interactive Entertainment.

Hingston, P. (2014). BotPrize. Perth, Oeste de Australia: Edith Cowan University | http://botprize.org/ .

Infinity Ward. (29 de Septiembre de 2003). Call of Duty. Activision.

IO Interactive. (2000). Hitman. Eidos Interactive, Square Enix.

kongregate. (17 de Febrero de 2012). Prom Week. kongregate | http://www.interactivestory.net/.

Mateas, M., & Stern, A. (5 de Julio de 2005). Façade. | https://promweek.soe.ucsc.edu/ .

Monolith Productions. (18 de Octubre de 2005). F.E.A.R. Vivendi.

Namco. (21 de Mayo de 1980). Pac-Man. Namco Bandai Games.

Nintendo EAD. (15 de Septiembre de 1985). Super Mario Bros. *Super Mario Bros.* Nintendo.

Opera Soft. (1987). La Abadía del Crimen. Opera Soft.

Ozawa, S., Kasabov, N. K., & Pang, S. (2013). Incremental Learning in Intelligent Systems. Springer-Verlag Berlin and Heidelberg GmbH & Co. K.

Padgham, L., & Winikoff, M. (2004). Developing Intelligent Agent System. Wiley.

Peinado, F., Cavazza, M., & Pizzi, D. (s.f.). *Revisiting Character-Based Affective Storytelling.* Facultad de Informática, Universidad Complutense de Madrid.

Pérez Colado, I. J., & Pérez Colado, V. M. (Junio de 2014). IsoUnity. *Un conjunto de herramientas para Unity orientado al desarrollo de videojuegos de acción-aventura y estilo retro con gráficos isométricos 3D*. Trabajo de Fin de Grado en la Facultad de Informática de la Universidad Complutense de Madrid.

Pérez Colado, I. J., & Pérez Colado, V. M. (Junio de 2014). TinyWorld. *TinyWorld*. Trabajo de Fin de Grado en la Facultad de Informática de la Universidad Complutense de Madrid | https://github.com/Victormafire/TinyWorld .

Rao, A. S., & Georgeff, M. P. (Enero de 1991). Deliberation and its Role in the Formation of Intentions. Cornell University Library.

Rockstar Games. (15 de Mayo de 2012). Max Payne 3. Rockstar Games.

Rockstar North. (14 de Abril de 2013). Grand Theft Auto V. Rockstar Games.

Santamaría Barcina, A., & Alexiades Estarriol, A. (2015). Reconstrucción de una video-aventura. *Reconstrucción de una video-aventura isométrica clásica utilizando IsoUnity como herramienta de producción multiplataforma*. Trabajo de Fin de Grado en la Facultad de Informática de la Universidad Complutense de Madrid.

Telltale Games. (11 de Octubre de 2013). The Wolf Among Us. *The Wolf Among Us*. Vértigo, Warner Bros. Interactive Entertainment.

Turn 10 Studios. (3 de Mayo de 2005). Forza Motorsport. Microsoft Studios.

Ubisoft Toronto. (20 de Agosto de 2013). Tom Clancy's Splinter Cell: Blacklist. Ubisoft.

Unity Technologies. (30 de Mayo de 2005). Unity. Unity Technologies.

Valve Software. (18 de Noviembre de 2008). Left 4 Dead. Valve Software, Electronic Arts.

Wooldridge, M. (Junio de 2000). Reasoning about Rational Agents. Cambridge University.

Zalta, E. N. (2009). *Modal Logic.* Metaphysics Research Lab.

9. Anexos

Anexo 1. Title
An Autonomous Control System for Video Game Characters Based on the Belief-Desire-Intention Cognitive Model

Anexo 2. Introduction

Artificial Intelligence techniques are being used more every day in videogame development with the objective of creating more realistic experiences or characters. Anyway, a lot of decision techniques are designed to follow a *script* and are not showing the expected result.

Almost all videogames are giving the impression of characters being able to decide about their actions, but it's not like that.

If we take a look in the way they act, we will surely be able to find an action pattern: an enemy hiding behind a wall and every so often leaning out to shoot, or other enemy not taking the 'main road' just to catch you from behind.

Probably anyone who had played a shooter, they may had notice their enemy's interaction in combat, so the player had wait patiently until they show up again so they can finish them.

Another example could be *Hitman* (IO Interactive, 2000); where stealth is the main objective, but what happens if the player decides to play in a different way? Probably the player will reach the waypoint, shooting outrageously to every enemy, and then get the cinematic with the main character crouched down with a single drop of blood in his clothes.

Those are two simple examples to show up that a big part of videogames are scripted, both in characters behavior as in the story. A lot of videogames gives a sensation of open world where you can choose between two actions, and then, every other character will react in consequence, but it is just that, a sensation.

The idea of this project is to design and prototype a system which allows videogame characters to take their own decisions, being truly intelligent, and being able to take their own actions even if it are not scripted.

The characters will have their own artificial intelligence which will allow them to interact with other characters, and based on this decision, follow their objectives and take different decisions. This intelligence will be designed thanks to a series of intelligent agents, one for every character in the videogame, which will allow them to *percept* their environment, process those *perceptions* and act based on their reasoning, it means, according their beliefs, their wishes and following what the established as their intentions facing the world.

As explained below, we will inspire a classic video game called *The Abbey of Crime*(Opera Soft, 1987), in which there were very primitive autonomous characters who behaved differently according to the story progressed. We will develop a test scenario inspired by this game, using a tool called *IsoUnity* (Pérez Colado & Pérez Colado, 2014) (plugin which extends the functionality of the development environment *Unity* game) and there will prove the usefulness of our system.

The structure of this report is as follows: after this first introductory chapter, we have a second chapter in which we review the state of the art, how is now research and development of autonomous characters in video games, with some ideas and techniques which we consider interesting. In Chapter 3 we specify the objectives of this work, in Chapter 4 indicate what our working method and the tools we have. In Chapter 5 we explain how the analysis, design, implementation and testing of this proposed intelligent system which controls autonomous form of video game characters based on the cognitive model Belief-Desire-Intention (called BDI) was. Finally, in Chapter 6 we discuss the results and in Chapter 7 we give our conclusions with our proposals for possible lines to continue our work.

Anexo 3. Conclusions

One of the many lacks videogames have this days is the lack of artificial intelligence on its characters. Is this lack of intelligence what makes games become repetitive or even boring. It aims to develop a kind of artificial intelligence giving more dynamism to NPCs.

Is both the lack of artificial intelligence, as the fake appearance of games having one implemented, being one of the highlighted problems in videogames. The intention is to, using *scripts*, characters give the sensation they are 'thinking'. Nevertheless, having all *scripted* it will never change the story if actions are performed in the same way every time.

With a BDI model is achieved that implemented agents are able to reach their established goals using their plans and varying their beliefs, in a more similar way to how human cognitive works.

Our project reaches the limit where characters are intelligent and interact between them and their environment. This makes the story more dynamic, making NPC act in a similar way to what would be in real life, acting according to their objectives, being able to lie or act in a way or another with the final purpose of getting their final objective.

Thanks to development test which we have done and the results obtained, we have been able to illustrate how the agent's implementation as a solution to autonomous control of videogame characters can be a good option to future implementation in artificial intelligence of non-playable characters in videogames.

Having done an achievement with principal requirements of our project, which were propose an artificial intelligence solution which could be integrated in an isometric virtual environment, with retro style, and having built an illustrative scenario of videogame, we realized there is still work needed to have an commercial product. We have realized of possible improvements which could be added in a future.

After finishing this project, we believe that this kind of implementation of artificial intelligence, even BDI model or another one, could give many dynamism to the videogame world, making game experience a lot more satisfying to player. Anyway, the implementation of this model, being developed in a declarative language (ASL), it is quite difficult because of not being as intuitive as other programing languages. Also, these declarative languages must control possibilities of having the character's state, and it suppose an exhaustive control of this state.

Anexo 4. Manual de instalación
Instalación de los entornos de desarrollo

Para poder ejecutar el juego y hacer interactuar ambas plataformas, estas deben estar correctamente instaladas y configuradas. A continuación proponemos una guía de configuración e instalación para preparar nuestra máquina, PC con *Windows*, y que todo funcione correctamente.

En primer lugar, se deben tener instalados ambos entornos. Para ello, accederemos a sus respectivas páginas para descargar los instaladores:

- *Unity*: https://unity3d.com/es
- *Eclipse*: https://www.*Eclipse*.org/downloads/

La instalación de ambos entornos de desarrollo viene guiada paso a paso en los respectivos instaladores.

Instalación de *Jason*

Para adaptar *Eclipse* para que sea compatible con *Jason*, debemos instalar un *plugin*, que obtendremos aquí: https://sourceforge.net/projects/Jason/files/Jason/

Recomendamos descargar la versión 1.4.2 ya que es la que utilizamos en este trabajo, y podemos garantizar que funciona perfectamente. Aunque las versiones más recientes no deberían dar problema, no podemos asegurar que el comportamiento sea el deseado.

Una vez descargado, debemos descomprimir el fichero y colocarlo en cualquier carpeta de nuestro equipo.

Si es la primera vez que se ha utilizado *Jason* en el equipo, se deberá ejecutar el fichero */lib/Jason.jar* que añadirá *Jason* al *PATH* del sistema.

Para la instalación de *Jason* en *Eclipse*, debemos arrancar el entorno y abrir el menú *'Help > Install New Software'* como podemos ver en la siguiente figura:

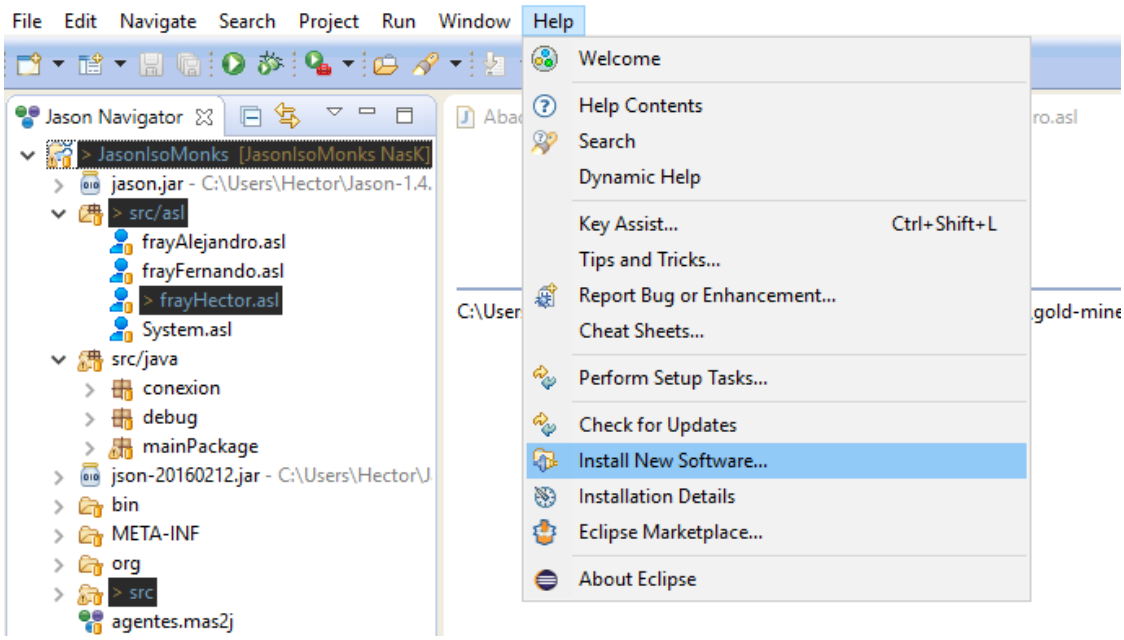

Fig. 9.1: Proceso de instalación de *Jason* en *Eclipse*. Instalar *plugin*.

Aparecerá una nueva ventana donde pondremos la dirección del *plugin*, que variará en función de la versión del entorno que estemos utilizando:

- *Eclipse Indigo:* http://Jason.sourceforge.net/*Eclipse*plugin/
- *Eclipse Juno, Kepler, Luna o Mars:* http://Jason.sourceforge.net/*Eclipse*plugin/juno/

Seleccionamos '*Jasonide'* para proceder a su instalación

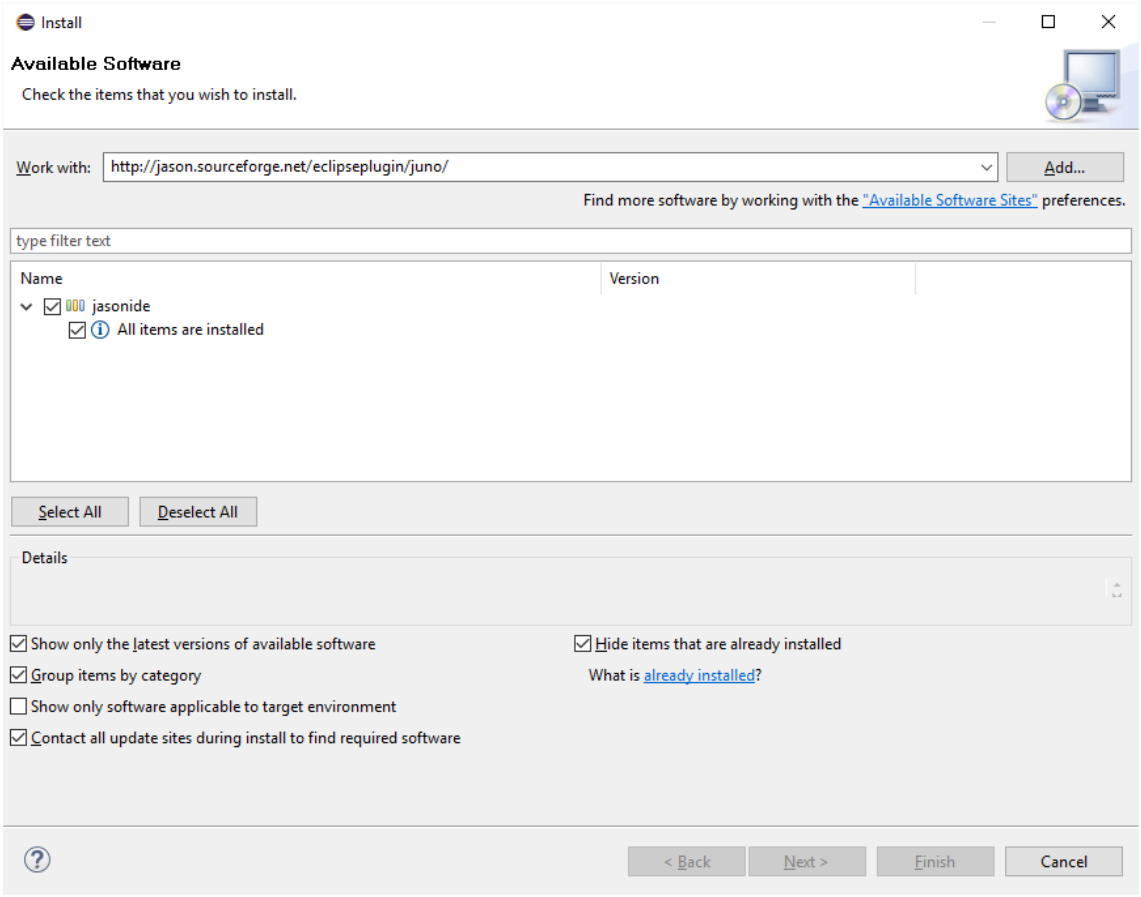

Fig. 9.2: Proceso de instalación de *Jason* en *Eclipse*. Instalar *plugin*. Paso final.

Continuamos con el proceso hasta que esté instalado, nos pedirá el reinicio de *Eclipse*, y una vez hecho, tendremos disponible la vista de *Jason* así como la importación y creación de proyectos *Jason*.

Importación de proyectos

Para importar los dos proyectos software relativos a este trabajo, denominados *IsoMonks* (el cliente) y *JasonIsoMonks* (el servidor), a sus respectivos entornos de desarrollo se deberán descargar de los repositorios donde están alojados.

- Proyecto *Unity: IsoMonks*
  https://github.com/NasK1991/IsoMonks
- Proyecto *Jason: JasonIsoMonks*
  https://github.com/Kaerit/JasonIsoMonks

Una vez descargados, se debe descargar también el mapa que necesitará *Unity*, que está alojado aquí:

https://drive.google.com/open?id=0B1lBgv6_d4joODgxdWdEX2Y1YXM

Este fichero se debe colocar en el proyecto *IsoMonks* en la carpeta '*\IsoMonks\Assets\Scenes'*. Después de esto, se deberán importar en los respectivos entornos de la siguiente manera:

En *Unity* es tan sencillo como abrir el entorno, y seleccionar *Open* y elegir la carpeta con el proyecto clonado de *GitHub*. Una vez importado, abriremos el mapa previamente colocado en '*\IsoMonks\Assets\Scenes'* desde el entorno.

En *Eclipse*, teniendo seleccionada la vista de *Jason*, en el menú contextual seleccionamos '*File > Import'* y seleccionamos el tipo de proyecto *Jason*

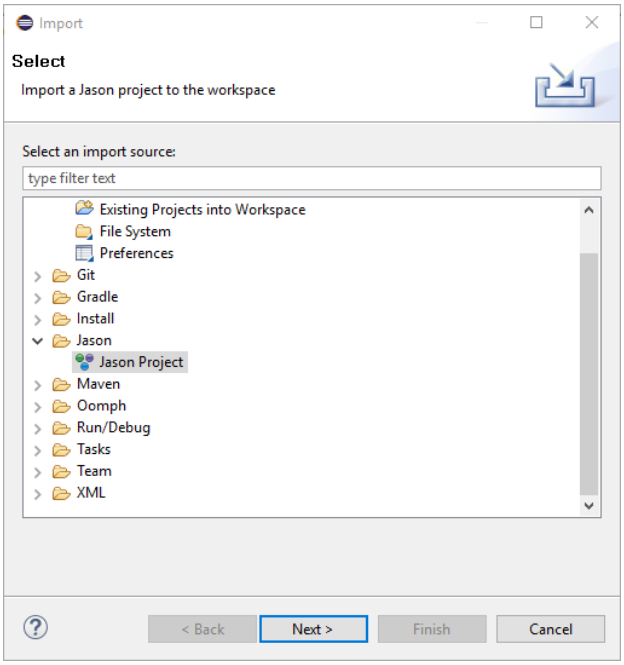

Fig. 9.3: Importar proyecto *Jason* en el entorno.

Se deben añadir las dependencias de archivos *.jar* externos que se utilizan, como el de JSON, para ello, primero debemos tenerlo descargado en nuestro equipo.

Accedemos a http://www.java2s.com/Code/Jar/j/DownloadjavaJSONjar.htm y descargamos el archivo *.jar*. Una vez en nuestro equipo, volvemos a *Eclipse*, y hacemos *click* derecho sobre el proyecto y accedemos a *Properties*.

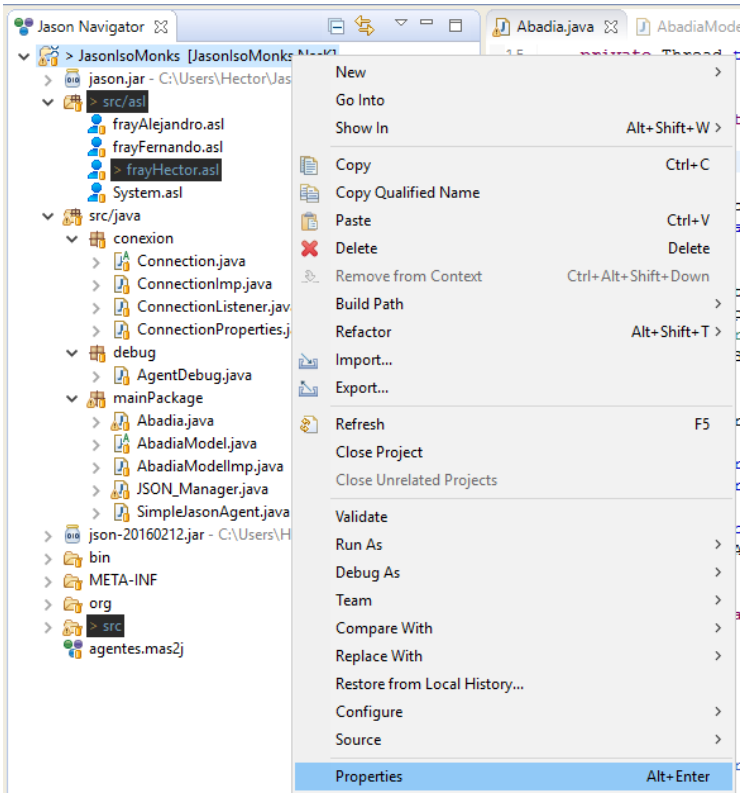

Fig. 9.4: Configurar librerías del proyecto *Jason*.

En la sección *Java Build Path*, accedemos a la pestaña *Libraries* y añadimos el *.jar* externo de JSON, debiendo quedar finalmente de la siguiente manera:

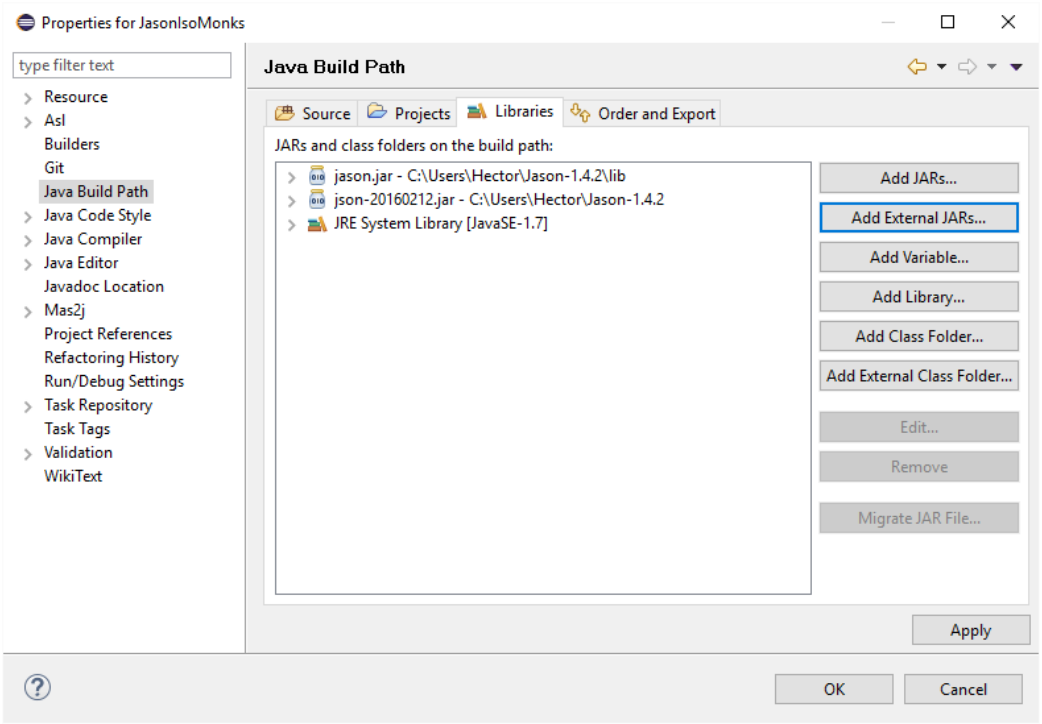

Fig. 9.5: Añadir librerías externas al proyecto *Jason*.

Ejecución

Ahora que tenemos ambos entornos correctamente instalados y configurados, así como los proyectos importados, podremos hacerlos funcionar. Es tan sencillo como tener ambos abiertos y lanzarlos en orden.

Primero se ejecuta el proyecto *JasonIsoMonks* (el servidor), que quedará a la espera hasta que *IsoMonks* (el cliente) esté listo. Será entonces cuando se ejecute el proyecto en *Unity* y comenzará la comunicación entre ambos entornos, pudiendo ver el escenario de ejemplo funcionando con normalidad.

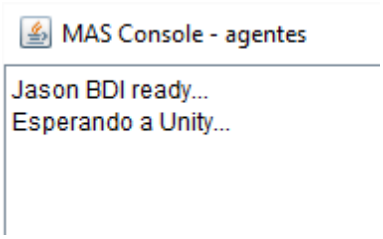

Fig. 9.6: *Jason* esperando a la ejecución de *Unity*

Fig. 9.7: *Unity* y *Jason* ejecutándose simultáneamente

Anexo 5. Contribuciones de los participantes

Alejandro Sánchez López

Mi principal aportación al proyecto ha sido la referenciada a *Unity*, e *IsoUnity*, encargado de revisar el código de los hermanos Pérez Colado y encargado de las modificaciones del mismo.

Me encargué de desarrollar dentro de *IsoUnity* unas modificaciones acerca de mandar eventos a través del socket de información, se generó para ello una clase asociada a uno de los objetos generados en *IsoUnity* que consistía en encapsular cierta información para poder mandarla.

También hubo que modificar la clase principal de ejecución del juego '*Game.cs'*, en la funcion encargada de actualizar la interfaz '*tick()*', se añadió unas líneas de código para poder aceptar los eventos que provenían a través del socket.

```
//Eventos llegados por el socket
GameEvent temp = Connection.getInstance().ReceivedEvent();
if (temp.Name == "action") {
    this.actionEvent(temp);
} else {
    this.events.Enqueue(temp);
}
```

Fig. 9.6: código de *game.cs*

En este fragmento de código se observa como recibe un evento y se encola en la cadena principal de eventos.

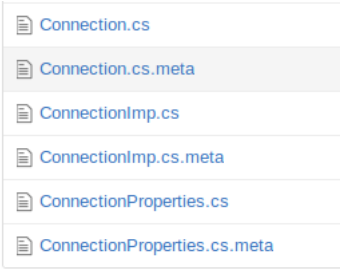

Fig. 9.7: paquete *connection*

A continuación, el paquete *conection* (véase en la figura antes mencionada) (los fichero meta, son creado por Unity, asi que no se pasaran a explicación), la clases:

Connection.cs: Fichero con la interfaz que contiene las cabeceras de las funciones, quse pasaran a explicar en su implementación.

ConnectionImp.cs: Fichero con la clase implementadora de las funciones.

Las funciones principales son las siguientes:

sendEvent(bool send, object ev): función que recibe como parámetro, un booleano para hacer el envío del segundo parámetro, un evento del juego, este primero se utiliza para poder visualizar la información sin tenerla que mandar.

GameEvent ReceivedEvent(): función que recibe la información del socket, es la utilizada en el bucle de la funcion principal, '*tick()'* de '*Game.cs*', la antes mencionada.

ConnectionProperties.cs: Fichero con los detalles de la conexión, que se extrajeron a un fichero aparte por tema de comodidad si hubiera que modificar esta información

```
public ConnectionProperties() {
    string IP = "127.0.0.1";
    int exitPort = 9876;
    int enterPort = 9877;
    int ttl = 10;

    socketClient = new UdpClient(IP, exitPort);
    socketServer = new UdpClient(new IPEndPoint(IPAddress.Any, enterPort));
    socketServer.Client.ReceiveTimeout = ttl;
    sender = new IPEndPoint(IPAddress.Any, exitPort);
}
```

Fig. 9.8: datos del fichero *ConnectionProperties.cs*

Además, los objetos a modificar en el mapa eran las celdas, que hubo que adaptarlas y modificar sus atributos de clase para poder introducir las modificaciones necesarias, y añadirlas un *collider*[12] para la generación del evento.

El evento consistía en un JSON, que contenía la id numérica de la celda generadora del evento y un mensaje con el tipo de evento, que era un id de tipo *string* que comprobábamos desde el lado de *Jason* y una cadena de texto, también de tipo *string*. El lanzador del evento consistía en un elemento de tipo *collider* de *Unity*, que al entrar en 'colisión' con otro producía que la clase asociada a la celda mandara el evento al socket.

Los objetos encargados de enviar la colisión eran los *colliders* del propio personaje principal '*frayFernando'* y de la celda generadora del evento. Al personaje también hubo que modificarlo para poder adaptarlo a esta nueva implementación, modificando la estructura del objeto creado por *IsoUnity* y por ende el código de los hermanos Pérez Colado.

También para poder entender la jerarquía de las clases y el funcionamiento de las mismas se hicieron pruebas con mapas, en los que se introducían marcadores para ver el funcionamiento y las trazas de la ejecución del código.

Durante la parte de aprendizaje del funcionamiento del lenguaje ASL, se creó también unas clases de '*Debug*' para ver la información que podía contener los agentes y el entorno y poder observar mejor como interactuaban entre ellos, y ver si se podrían

[12] Objetos en Unity encargados de los eventos de las colisiones.

generar sobre código *Java*, después de las pruebas que hicimos utilizando esa clases se determinó que los agentes pertenecían al entorno y este una vez generado a través del fichero de carga, no se podían añadir más.

También fui encargado de la realización de las pruebas de fluidez y consumo de memoria sobre la ejecución de los entornos de *Jason* e *IsoUnity*, cuando se lanzaban conjuntamente.

Fernando Romero de la Morena

He realizado diversas tareas durante el proyecto. De las primeras que realicé fue la instalación y preparación de la herramienta *IsoUnity* en el entorno de desarrollo de videojuegos *Unity3D*, además de los primeros pasos con dicha herramienta. Fue necesario leer la documentación aportada por los autores de la herramienta y comenzar a realizar pruebas de diseño con ella. Se probaron los bloques que *IsoUnity* va generando, asignándoles las decoraciones pertinentes para empezar a crear un mapa visual, además de los personajes.

Una de las grandes contribuciones y desarrollos que hice fue la preparación, análisis y desarrollo del sistema de comunicación entre *Unity3D* y *Java*. Esto fue llevado a cabo mediante un desarrollo de un cliente-servidor *UDP* que, mediante sockets intercambian la información de los distintos entornos, todo esto almacenado en JSON. Este desarrollo tuvo que ir mejorando con el paso del tiempo, ya que según se avanzó en el proyecto, salían a la luz problemas que no pude ver venir en un principio. Uno de estos problemas fue que el servidor dejaba de responder ya que se quedaba esperando a algo que nunca llegaba, un evento. Esto provocaba que se parara por completo la ejecución, y fuera necesario reiniciarlo. Esto se arregló con la creación de los *hilos* de procesos.

Una vez terminada la comunicación, había que empezar a implementar los métodos que harían funcionar a los agentes en el lenguaje ASL. Estudié los distintos ejemplos que se encuentran en internet sobre dicho lenguaje, y empecé a implementar nuestras ideas. Los primeros pasos fueron cómodos y fáciles, pero según se avanzó en el desarrollo de rutinas más intrincadas, hubo que adentrarse más a fondo en cómo funcionaba a bajo nivel *Jason*.

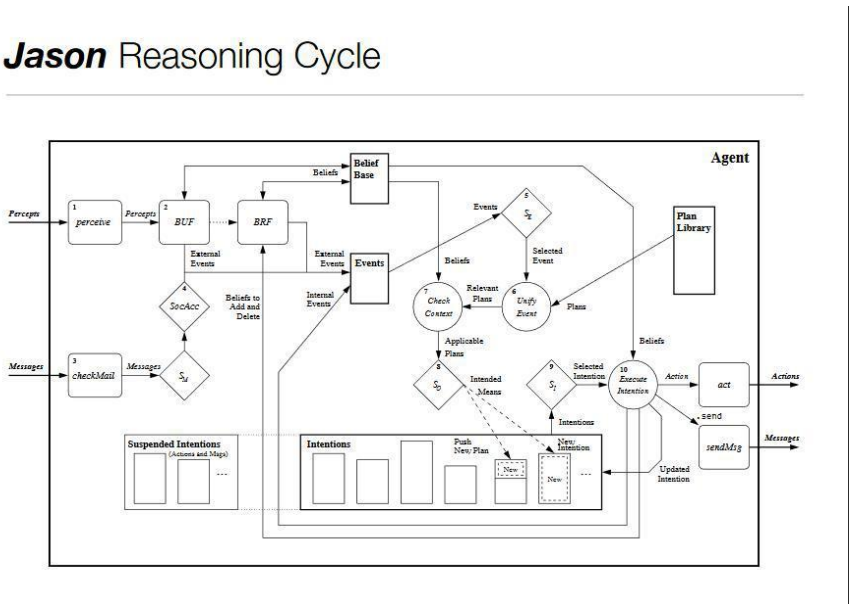

Fig. 9.9: funcionamiento interno de agentes ASL

Teníamos un error de concepto en cuanto a cómo funcionaba *Jason*, ya que pensábamos que la *BeliefBase* se actualizaba directamente desde código Java, y esto no era así. Viendo la imagen de arriba pude comprender que realmente al agente sólo le llegan las percepciones, y no actualizaciones directas de la *BeliefBase*. Son dichas percepciones las que actualizan dicha *BeliefBase*. Con esto desarrollado pude crear una rutina simple para un agente y que, al provocar un "bloqueo" en dicha rutina, por parte del jugador, el agente modificara su creencia sobre algo, y tuviera que modificar sus planes y sus metas para poder continuar con dicha rutina.

Una vez que se consiguió comprender cómo funcionaba y qué se estaba haciendo mal, fui capaz de desarrollar un método que, dado un agente, generara sus propios planes en función de sus creencias.

Es importante destacar el trabajo de investigación que tuve que hacer sobre el lenguaje de programación *Jason* y cómo funciona a bajo nivel, ya que la documentación y ejemplos que hay en internet, además de ser escasos, son bastante pobres en cuanto a información. Todo lo que se pudo implementar fue por la unión de conocimientos de distintas fuentes.

Para finalizar, los últimos pasos que di en el desarrollo de este proyecto, fue el desarrollo de esta memoria, que junto a mis compañeros, hemos realizado. El trabajo fue repartido para que todos pudiéramos contar en los distintos apartados las partes que más conocimientos teníamos. Es por ello que me he encargado de las partes de comunicación entre entornos y determinadas partes de la memoria que hablan de *Jason* y su comportamiento, o su funcionamiento. Otra de las partes que me encargué en la memoria fue el análisis del actual mercado de videojuegos y cómo funcionan en cuanto a mecanismos de inteligencia artificial se refiere, lo que me daba ideas de implementación para nuestro propio proyecto, las cuales, algunas se han podido llevar a cabo y otras no por falta de tiempo, pero que sin duda serían muy interesantes de implementar en un futuro.

Héctor Martín Solís

Durante el desarrollo de este proyecto, se ha intentado hacer que el reparto de tareas fuese lo más equitativo posible, siempre intentando hacer que cada uno realizase una parte similar de trabajo. De igual manera, puesto que nuestro Trabajo Fin de Grado consistía en dos entornos de desarrollo diferentes, con diferentes lenguajes, algunos teníamos más experiencia que otros en un entorno u otro.

No obstante, hemos tratado de aprender a manejar ambos entornos y lenguajes, pero obviamente, nos dividimos las tareas de manera que cada uno estuviese más desenvuelto en su entorno.

Por mi parte, en un principio me encargué de investigar cómo funcionaba *Jason*, puesto que era algo nuevo para todos nosotros. Yo, personalmente había trabajado en alguna ocasión con algún lenguaje declarativo como *Prolog*, por lo que me resulto un poco menos difícil de entender.

En los primeros pasos, tanto Fernando como yo, fuimos indagando en los ejemplos que proporcionaba la página de *Jason* y tratando con su herramienta. Con los ejemplos que más tratamos fueron con los *Gold Miners* y con *Domestic Robot*, que nos ayudaron mucho a entender cómo se comunicaban estos agentes.

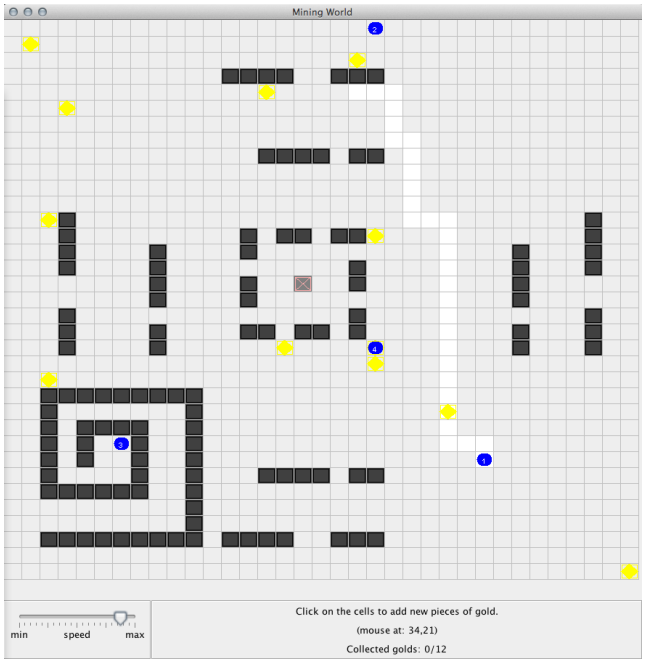

Fig. 9.10: ejemplo de *Gold Miners* en funcionamiento

Una vez entendido que estos agentes se 'desplegaban' sobre un entorno común a todos, tratamos de importar los ejemplos que proporciona la documentación de *Jason* a *Eclipse* para ejecutarlos y tener una idea más práctica del funcionamiento de estos agentes en el entorno, y cómo se conectaba con *Java* puro.

Tras esta investigación con los ejemplos, empezamos con la instalación de *Jason* en *Eclipse* e intentando lanzar los primeros agentes a interactuar. Creamos un entorno personalizado de la Abadía donde iban a estar nuestros agentes 'frailes'.

Mientras que Alejandro configuraba la parte de la conexión en la parte de *IsoUnity*, y Fernando hacía esa misma parte para *Java*, yo continué avanzando en el desarrollo de la parte *Jason* dentro de *Java*. Fue entonces cuando conseguimos comprender la comunicación entre Jason y Java, cuya función *main* era ahora la función *init* que proporcionaba *Jason*, y apartir de ese pequeño, pero vital 'descubrimiento', conseguimos configurar mediante *Java* el entorno donde se iban a desplegar los agentes.

Cuando la parte de *IsoUnity* no requería tanto trabajo como la de *Jason*, nos centramos todos un poco más en esta última sin olvidarnos de la primera, avanzando en la intercomunicación de agentes y aumentando poco a poco sus rutinas, control de acciones, creencias, planes, etcétera.

Toda la parte de *Jason* es la que más complicaciones nos ha supuesto, puesto que en su página, pese a que explican el funcionamiento interno de los agentes, así como el lenguaje *AgentSpeak* en el que se desarrollan, no se habla apenas de su integración y funcionamiento en entornos de desarrollo como *Eclipse*.

Por mi parte, modifiqué pequeños fragmentos de código de *IsoUnity* para configurar las entidades que manejaba. También aporté bastante al desarrollo visual, construyendo una de las últimas ampliaciones finales del mapa.