Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un videojuego con sistema modular de IA manipulable por el jugador en Unity3D

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Pedro Cazorla Martínez

Tutor: Jorge González Molla

2020-2021

Resumen

El objetivo de este trabajo es desarrollar un videojuego en el que los jugadores podrán crear sus propias criaturas definiendo su comportamiento usando un sistema modular de componentes predefinidos de IA. El juego fomentará la creatividad del jugador dejando que resuelva los problemas con los que se encuentra de forma libre.

Palabras clave: Unity, Videojuego, IA, modular, 3D.

Abstract

This project's objective is developing a videogame in which players can create their own creatures defining their behaviour using a modular system with pre-defined AI components. The game will foster creativity in the player by letting them resolve the problems they encounter freely.

Keywords: Unity, Videogame, AI, modular, 3D.

Índice

1. Introducción

Se ha analizado los posibles enfoques para el desarrollo de videojuegos de puzle y aventura satisfactorios y escogido una solución potencialmente viable. Los videojuegos de puzle se centran en desafíos lógicos y conceptuales. “Aunque muchos juegos de acción y de aventura incluyen elementos tipo puzle en el diseño de sus niveles, un verdadero juego de puzles se centra en la resolución de los puzles como la actividad principal de juego” [1] (Rollings, Andrew; Ernest Adams, 2006).

El videojuego desarrollado se trata de un videojuego en primera persona, es decir, donde la imagen vista en pantalla es la perspectiva desde los ojos del personaje que controla el jugador. El jugador es capaz de mover el personaje en un mundo tridimensional y girar la cámara para observar sus alrededores.

El jugador puede hacer aparecer y desaparecer una criatura aliada. El comportamiento de esta criatura es la inteligencia artificial que es compuesta por el jugador. Se entiende inteligencia artificial en el contexto de los videojuegos como el comportamiento de los personajes del juego no controlados directamente por el jugador.

En el juego desarrollado, los jugadores componen una máquina de estados simple y las transiciones entre estados, eligiendo qué acciones y en qué orden debe tomar la criatura. El jugador también elige sobre qué objetos del escenario tendrá que realizar dichas acciones la criatura.

A través de la criatura el jugador interactúa con distintos elementos del escenario para avanzar en este resolviendo desafíos lógicos basados en el comportamiento de la criatura.

El desarrollo de este sistema es de utilidad para lograr el objetivo principal de un videojuego, el entretenimiento, a través de permitir que el jugador resuelva los problemas que se le presentan con soluciones propias (al menos desde su punto de vista).

Se han realizado pruebas con jugadores en distintos puntos del desarrollo para validar la viabilidad de este enfoque y para corregir los problemas detectados a través de estas pruebas.

Finalmente, estos resultados han sido analizados para confirmar la eficacia de este enfoque y observar las posibles formas en las que se puede extender y aplicar este trabajo.

1.1 Motivación

Los videojuegos como pieza de software son interesantes ya que tienen requerimientos distintos al resto de las aplicaciones y por tanto ofrecen problemas a resolver distintos. En las aplicaciones comunes el objetivo suele ser el de llevar a cabo una tarea dada priorizando la eficiencia, eficacia y facilidad de uso. En los videojuegos

el objetivo principal es el entretenimiento y estímulo del jugador, existiendo muchos enfoques distintos (a menudo aplicados conjuntamente) con los que un videojuego consigue estos objetivos.

Dentro de las distintas formas de conseguir esta diversión en el jugador existen el género de puzles y el género de exploración / aventura. Los juegos de puzles estimulan al jugador ofreciéndole problemas lógicos a solucionar dentro del contexto del juego, mientras que los juegos de aventura presentan al jugador la habilidad de desplazarse por el entorno del juego libremente, dejando implícito el problema de qué camino escoger. Ambos géneros basan su entretenimiento en impedir el avance del jugador con obstáculos que debe aprender a atravesar; el juego debe conseguir que tanto llegar a la solución como resolver el problema sea satisfactorio para el jugador.

Existen muchas maneras de que la experiencia sea satisfactoria, pero una de las más interesantes es la de hacer sentir al jugador que el camino y/o solución escogidos han sido de su propia invención. Los videojuegos emplean todo tipo de técnicas para generar esta sensación: desde esconder todas las pistas que se le proporcionan de forma que el jugador no las note, hasta proporcionar sistemas que permitan al jugador realmente generar soluciones completamente nuevas a los problemas (por ejemplo, en los juegos de género *sandbox*).

En este trabajo se ha desarrollado un sistema propio y el juego que lo utiliza basado en ofrecer al jugador una herramienta para generar sus propias soluciones a los problemas presentados, en la forma de poder diseñar la inteligencia artificial básica de sus criaturas. Para complementar este sistema se ha desarrollado otros sistemas que interactúan entre ellos y pretenden hacer posible que surjan soluciones inesperadas en base a las acciones de los jugadores.

Los mecanismos desarrollados no son interesantes solo para el juego a desarrollar ni solo para la industria de los videojuegos en general, sino también para las aplicaciones de software encaradas a usuarios inexpertos. Permitir al usuario manipular el funcionamiento de una parte del sistema sin tener conocimientos de programación (scripting visual) es una funcionalidad ya encontrada en muchas aplicaciones, pero está aún en una etapa temprana de su desarrollo; por este motivo es interesante realizar exploraciones de este tipo de funcionalidad como lo es este proyecto.

1.2 Objetivos

En este trabajo se ha tomado como objetivo desarrollar un videojuego usando el motor Unity con un sistema de IA básico manipulable por el jugador y sistemas de juego emergentes (este concepto se explicará en el apartado 2. Estado del arte) que permitan al jugador solucionar los problemas planteados de forma propia. Se ha propuesto crear inicialmente dos niveles para este juego.

A su vez, se ha propuesto validar la efectividad de estos sistemas a través de pruebas con jugadores con el objetivo de comprobar la efectividad de los sistemas desarrollados para entretener a los usuarios.

1.3 Metodología

Se ha desarrollado el videojuego utilizando el marco de trabajo ágil Scrum. Scrum divide el desarrollo en periodos cortos de trabajo llamados *sprints*. "El *sprint* es un esfuerzo con un tiempo limitado; la longitud está acordada y fijada de antemano para cada *sprint* y suele estar entre una semana y un mes, con dos semanas siendo la más común "[2] (Ken Schwaber, 2004). Al final de cada uno de estos periodos se obtiene una nueva versión incremental del producto.

Para llevar la planificación se ha utilizado Jira[1], una aplicación que permite gestionar las historias (representan funcionalidad a desarrollar, requisitos) y tareas (representan trabajo a realizar necesario para el proyecto pero que no se traducirá directamente en funcionalidad). "Las historias de usuario son simples, cortas y describen en lenguaje natural la perspectiva de una persona (usuario o cliente) sobre un tema del proyecto" [3] (Mădalina MANOLE, Mihai-Șerban AVRAMESCU, 2017). Estas tareas e historias se definen e inicialmente asocian al *backlog*. "El *backlog* es la vista única y definitiva de todo lo que podría hacer el equipo de desarrollo, ordenado por prioridad "[4] (Pete Deemer, Gabrielle Benefield, Craig Larman, Bas Vodde, 2012). Antes de que un sprint comience se prepara moviendo las tareas e historias que se pretende desarrollar del *backlog* al *sprint*, definiendo el alcance de ese *sprint*.

Dentro de un sprint se cuenta con un tablero Kanban, que es una representación del estado actual de cada tarea/historia. El tablero que se ha usado ha consistido en cuatro posibles estados de una tarea: *To Do* (por hacer, tareas aún no empezadas), Diseñando, Programando/Haciendo y *Done* (tareas ya finalizadas).

Por último, con el objetivo de organizar el desarrollo desde un enfoque más amplio se ha contado con varias épicas: agrupaciones de tareas e historias con un objetivo común. Son de carácter orientativo, ya que en un desarrollo ágil el alcance del proyecto y de cada sprint pueden variar a lo largo del tiempo, por lo que no es posible planificarlos con seguridad antes del desarrollo. Las épicas del proyecto se explicarán en el apartado 3.6 Planificación.

1.4 Estructura

A continuación, se presenta la estructura general de la memoria con una explicación corta del contenido de cada apartado:

- **Estado del arte**: Se detalla el análisis del estado actual del mercado de videojuegos, los tipos de videojuegos más comunes, así como el nicho en el que cae el juego a desarrollar y por qué es interesante desarrollarlo.
- **Análisis del problema**: En esta sección se analiza el videojuego propuesto, detallando los requisitos que debe tener, así como los riesgos que puede conllevar su desarrollo, las posibles soluciones alternativas de llevarlo a cabo y finalmente se explica la solución escogida y la planificación del desarrollo.
- **Diseño de la solución**: Se explica el diseño del software del videojuego, empezando por la arquitectura, que define a grandes

[1] Página oficial de Jira: https://www.atlassian.com/software/jira

rasgos el comportamiento de bloques del software. Después se explica con más detalle el diseño de cada bloque y la utilidad de cada clase.

- **Desarrollo de la solución**: Se muestran ejemplos del código producido para el desarrollo del videojuego que puedan ser de interés, explicando la aplicación de ciertos conceptos de programación aprendidos que han sido útiles en el desarrollo, así como el funcionamiento de las mecánicas y sistemas principales del juego.
- **Pruebas**: Durante el desarrollo de este proyecto se han llevado a cabo dos series de pruebas con usuarios para comprobar la legibilidad de las mecánicas y el nivel de satisfacción de los jugadores. En este apartado se explican lo aprendido de estas pruebas y su utilidad durante el desarrollo.
- **Conclusiones**: Se resume lo logrado al final del desarrollo, además de explicar cómo han ayudado los conocimientos obtenidos durante la carrera al desarrollo.
- **Trabajos Futuros**: Se describen las posibilidades de futuro desarrollo sobre el videojuego creado, puntos a rematar, posibilidades de ampliación y otras aplicaciones de la tecnología.
- **Referencias**: Orígenes de las citas usadas en el texto de este documento y glosario con definiciones de varios términos usados en el texto.

2. Estado del arte

Los videojuegos pueden ser entendidos a través del enactivismo como una relación entre el jugador y su entorno, siendo el juego parte de este entorno. “El juego ofrece ciertas capacidades que actúan tanto como posibilidades como restricciones al jugador” [5] "(Jukka Vahlo, 2017). Aumentar las capacidades proporcionadas al jugador puede aumentar su tiempo de juego y satisfacción jugando.

En el lado más extremo de los juegos que ofrecen una gran cantidad de capacidades existen los juegos *sandbox*. Este género hace referencia a juegos “cuyo diseño enfatiza y fomenta el juego libre” [6] (Steve Breslin, 2009). El juego ofrece al jugador ciertos sistemas y un alto grado de libertad para manipularlos, usualmente ofreciendo pocos objetivos y si existen estos siendo muy generales. Un ejemplo de este tipo de juego es Minecraft[2], un juego en el que el espacio físico está dividido en bloques. El jugador tiene la capacidad de romper estos bloques y construir con ellos, pudiendo cambiar el entorno a voluntad. El juego ofrece algunos desafíos opcionales, pero en todo momento es el jugador el que decide lo que desea hacer. Este grado de libertad puede acabar perdiendo el interés del jugador, ya que este mismo es el que debe generar motivos para seguir jugando.

Alejándonos del extremo, existen otros juegos que ofrecen al jugador ciertos grados de libertad, pero también ofrecen objetivos más definidos. Uno de los ejemplos más

[2] Minecraft es uno de los videojuegos del género sandbox más populares: https://en.wikipedia.org/wiki/Minecraft

claros es The legend of Zelda: Breath of The Wild[3]. Este juego ofrece un mundo abierto además de ciertas habilidades y objetos utilizables por el jugador que tienen efectos simples en el mundo. Cuando estos efectos simples interactúan entre sí, sin embargo, se producen efectos más complejos que los jugadores pueden utilizar para afrontar los desafíos del juego de formas distintas y en muchos casos no predefinidas por los desarrolladores. A este tipo de juegos se los conoce como juegos con jugabilidad emergente: juegos donde “las mecánicas permiten al jugador crear nuevas estrategias y utilidades más allá de la intención o utilidad original” [7] (Josh Bycer, 2015).

En la figura 1 se muestra como un jugador ha tomado elementos del juego y los ha combinado para crear un tipo de carro volador y desplazarse por el aire, algo no existente de ninguna manera como elemento diseñado por los desarrolladores. Es este tipo de soluciones creativas inesperadas las que se buscan cuando se diseñan elementos simples que forman sistemas emergentes.

Figura 1 The Legend of Zelda: Breath of the Wild

Este tipo de interacciones son muy populares entre los jugadores, llevando a que se formen comunidades alrededor de encontrar y utilizar estas mecánicas emergentes.

En el caso de los videojuegos de puzles y exploración, estos suelen presentar tanto objetivos fijos y bien definidos, como soluciones únicas a los obstáculos. Esta rigidez es funcional, pero tiende a generar frustración en los jugadores cuando la dificultad de encontrar la solución es muy alta y no se adapta a la forma de pensar del jugador. Algo solucionable ofreciendo al jugador múltiples formas de afrontar los problemas.

Otros videojuegos permiten un grado intermedio de libertad. El juego Loadout permitía modificar el comportamiento de las armas del jugador de forma propia, centrándose en

[3] The Legend of Zelda: Breath of the Wild: https://en.wikipedia.org/wiki/The_Legend_of_Zelda:_Breath_of_the_Wild

esta mecánica. Se trataba de un juego multijugador que permitía combatir contra otros jugadores con armas diseñadas por el propio jugador. Esto resultaba en una sensación de originalidad y orgullo en el jugador.

Otro ejemplo son los juegos tipo rpg que permiten al jugador decidir qué habilidades tendrá su personaje, como por ejemplo Dark Souls. El jugador es capaz de elegir qué tipo de ataques tendrá a su disposición para lidiar con los enemigos, así como la agilidad del personaje. Estas decisiones aumentan la diversión del jugador y el tiempo que pasa jugando, al querer probar más posibilidades. Sin embargo, esta libertad viene restringida por el hecho de que las opciones son predefinidas, se trata de una selección entre posibilidades, no de creación de nuevas opciones.

2.1 Crítica al estado del arte

Existen pocos juegos como Breath of The Wild que permitan al jugador crear sus propias soluciones en entornos semi-controlados. Los juegos sandbox cumplen un rol distinto, ya que el grado de libertad es mucho mayor, los objetivos que se ofrecen son mucho más generales y desarrollan aspectos distintos de la relación juego-jugador.

La combinación de puzles y exploración en primera persona, en general, no ha visto demasiados juegos salir a la luz, especialmente en las últimas décadas. Asimismo, estos juegos suelen centrarse en soluciones únicas a los problemas planteados.

Existe entonces un nicho en el mercado de juegos que permitan al jugador desarrollar su creatividad dentro de un espacio prediseñado con objetivos concretos que ofrezca la posibilidad de que el jugador encuentre sus propias soluciones, consiguiendo que el juego se ajuste más al estilo de juego del jugador.

2.2 Propuesta

El juego desarrollado pretende existir dentro de este nicho. El juego ofrece al jugador una serie de herramientas para interactuar con el juego y una serie de sistemas que reaccionan de forma predecible, pero se complementan e interactúan entre sí. Esta combinación permite que el jugador actúe en base a su conocimiento del funcionamiento de los sistemas para usar las herramientas dadas para generar soluciones propias.

El sistema principal que ofrece el juego es un sistema por el cual, a través de una interfaz gráfica, el jugador puede diseñar una máquina de estados la cual se utiliza como inteligencia artificial de unas criaturas que crea el jugador a modo de ayudantes. Estas criaturas son capaces de llevar a cabo una serie de acciones o tareas que puede elegir el jugador, como moverse a un punto o atacar una entidad del entorno.

También se le permite al jugador definir las transiciones entre los estados de la inteligencia artificial. Estas transiciones se lanzan cuando se producen ciertos eventos, como que una tarea haya sido completada.

Esta mecánica permite que el jugador resuelva los problemas que le presenten a su modo. No simplemente ha de utilizar una habilidad dada ni encontrar una solución preexistente y rígida. En su lugar, el jugador debe ingeniar una combinación de

estados y transiciones que satisfaga sus necesidades, y pueden existir múltiples combinaciones que permitan al jugador sobrepasar un obstáculo dado.

Para complementar esta mecánica existe una serie de mecánicas simples que pretenden actuar conjuntamente como un sistema emergente. Estas mecánicas se basan en tres estados en los que pueden estar ciertas entidades: en llamas, húmedas o electrificadas. Mediante una serie de reglas que se detallan en el apartado 3.1 de requisitos funcionales, las entidades en estos estados afectan a otras entidades cercanas.

Las criaturas del jugador también pueden estar en estos estados, lo que permite que el jugador interactúe de forma semi libre con las entidades a través de estos sistemas y utilice sus propiedades de forma intuitiva para encontrar sus propias soluciones a los problemas en su camino.

Para llevar a cabo el propio desarrollo se ha utilizado la herramienta Unity 3D, un motor de videojuegos gratuito que ofrece muchas facilidades para el desarrollo. Algunas de estas facilidades se describirán brevemente en la sección 4.1 Arquitectura del sistema y la sección 5. Desarrollo de la solución.

3. Análisis del problema

Se ha realizado un análisis del videojuego a desarrollar, habiendo especificado los requisitos tanto funcionales como no funcionales. “Los requisitos se definen como una especificación de lo que debería ser implementado. Son descripciones de como el sistema debería comportarse, o de una propiedad o atributo del sistema. Pueden ser restricciones en el proceso de desarrollo del sistema” [8] (K. E. Wiegers, 1997). Con estos requisitos se han explorado distintas soluciones y se ha elegido una solución respecto al resto.

Al analizar nuestro juego como sistema observamos que solo existen dos actores: el reloj y el jugador. Esto se debe a que las personas que interactúen con el juego lo harán siempre desde el mismo rol de jugador, y el juego no interactuará con sistemas externos.

3.1 Requisitos Funcionales

Se han dividido los requisitos funcionales en 5 características:

- Control del personaje
- Sistema de habilidades del jugador
- Sistemas emergentes
- Menús
- Progreso del jugador

Los requisitos funcionales se han desarrollado como casos de uso. “Un caso de uso es todas las maneras de usar un sistema para conseguir un objetivo particular para un usuario particular. Combinados, el conjunto de todos los casos de uso proporciona todas las maneras útiles de usar el sistema, e ilustra el valor que ofrece.” [9] (Ivar Jacobson, Ian Spence, Kurt Bittner, 2011).

Se ha decidido utilizar casos de uso porque estos representan la interacción del usuario con el sistema. Otras formas de representar los requisitos, como las historias de usuario, solo describen la funcionalidad deseada. La interacción con el usuario es muy interesante en este proyecto al tratarse de un videojuego, donde el jugador espera que las acciones que realizan tengan cierto grado de realimentación, es decir, que cuando se realiza una acción que resulta en un cambio del estado del sistema que es de interés para el jugador, este debe ser alertado de alguna manera.

3.1.1 Control del personaje

La característica de control del personaje engloba los requisitos relacionados con capturar el *input* del jugador relacionado con el movimiento y desplazar al jugador por el mundo del juego, así como su capacidad de interactuar con objetos del escenario y de mover su cámara.

Referencia:	CU-01
Nombre:	Moverse horizontalmente
Descripción	Permite moverse horizontalmente dentro del espacio del juego. 1. El jugador presiona una tecla de movimiento horizontal 2. El sistema comprueba si el personaje puede moverse y, en ese caso, calcula la nueva velocidad del personaje y lo desplaza acorde a esta
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-02
Nombre:	Saltar
Descripción	Permite saltar dentro del espacio del juego. 1. El jugador presiona la tecla de salto 2. El sistema comprueba si el jugador puede moverse y si está en el suelo 3. Si el personaje puede moverse y está en el suelo, se calcula la nueva velocidad y se desplaza hacia arriba al personaje 4. Si el personaje no está en el suelo, se comprueba si hace poco que abandonó el suelo 5. Si hace poco que abandonó el suelo, se calcula la nueva velocidad y se desplaza hacia arriba al personaje 6. Si no hace poco que abandonó el suelo, el sistema guarda la petición de salto y, si el personaje toca el suelo dentro de un tiempo, se

	calcula la nueva velocidad y se desplaza hacia arriba al personaje
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-03
Nombre:	Correr
Descripción	Permite correr, lo que aumenta la velocidad del personaje. 1. El jugador mantiene presionado la tecla de correr 2. Si el personaje se está desplazando horizontalmente, el sistema aumenta la velocidad de movimiento
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-04
Nombre:	Agacharse
Descripción	Permite agacharse, lo que disminuye el tamaño del personaje y permite acceder a nuevos espacios. 1. El jugador presiona la tecla de agachado 2. Si el jugador no está agachado ya, el sistema pasa el personaje a agachado, se disminuye su velocidad, pero también se altura y la altura de su colisión dentro del juego 3. Si el jugador ya está agachado, el sistema pasa el personaje a no estar agachado, se revierten su altura y velocidad a los valores iniciales
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-05
Nombre:	Mover la cámara
Descripción	Permite mover la cámara por la que ve el jugador, permitiendo observar el entorno a voluntad. 1. El jugador desplaza el ratón 2. Si no hay abierto ningún menú, el sistema calcula la nueva posición y rotación de la cámara añadiéndole el movimiento del ratón 3. El sistema comprueba que la nueva posición vertical de la cámara está dentro del rango de valores aceptable y, de no estarlo, fija la cámara al valor límite más cercano

Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-06
Nombre:	Abrir puerta
Descripción	Permite abrir una puerta del nivel 1. El jugador se acerca a una puerta 2. El sistema muestra un diálogo de abertura de puerta 3. El jugador presiona la tecla de aceptar diálogo 4. El sistema comprueba si la puerta necesita una llave y, de necesitarla, comprueba que la llave existe en el inventario del personaje 5. Si la puerta puede ser abierta por el jugador, el sistema inicia la animación de apertura de la puerta y esta queda abierta
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-07
Nombre:	Añadir objeto a inventario
Descripción	Permite que se añada un objeto al inventario 1. El sistema recupera el nombre y cantidad de objeto de la entidad de la que se está añadiendo el objeto 2. El sistema añade el objeto con dicha cantidad al inventario
Actor	-
Relaciones	
Precondición	

Referencia:	CU-08
Nombre:	Coleccionar gotas
Descripción	Permite que el jugador añada gotas de agua coleccionables a su inventario al tocarlas 3. El jugador se desplaza hasta estar dentro del rango de una gota 4. El sistema añade el objeto gotas al inventario (realiza el CU-07 Añadir objeto a inventario). Se actualiza el número de gotas mostrado en pantalla 5. El sistema muestra un diálogo que comunica al jugador el número de gotas obtenidas
Actor	Jugador
Relaciones	Incluye CU-07 Añadir objeto a inventario

Precondición	

Referencia:	CU-09
Nombre:	Recoger objeto
Descripción	Permite que el jugador añada objetos del escenario a su inventario 1. El jugador se desplaza hasta estar dentro del rango de un objeto 2. El sistema muestra un diálogo que permite al jugador recoger el objeto 3. El jugador acepta el diálogo 4. El sistema añade el objeto al inventario (realiza el CU-07 Añadir objeto a inventario)
Actor	Jugador
Relaciones	Incluye CU-07 Añadir objeto a inventario
Precondición	

Referencia:	CU-10
Nombre:	Seleccionar objetivo
Descripción	Permite que el jugador seleccione un objetivo del entorno para un estado 1. El jugador mantiene la tecla de objetivos 2. El sistema muestra un contorno de color visible a través de paredes alrededor de los objetivos cercanos en el entorno 3. El jugador selecciona el estado para el que quiere definir el objetivo, apunta al objetivo y lo selecciona. 4. El sistema guarda el objetivo para ese estado y muestra el tipo de objetivo en la interfaz
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-11
Nombre:	Crear objetivo de movimiento
Descripción	Permite que el jugador cree un objetivo en una superficie que permite a una criatura intentar desplazarse a ese punto 1. El jugador mantiene la tecla de objetivos y apunta a una superficie 2. El sistema muestra un círculo que representa el objetivo de movimiento a crear 3. El jugador selecciona el estado para el que quiere definir el objetivo y acepta el objetivo 4. El sistema guarda el objetivo para ese estado y lo fija en el lugar
Actor	Jugador
Relaciones	

Precondición	

3.1.2 Sistemas de habilidades del jugador

Engloba los requisitos relacionados con las habilidades especiales del jugador, específicamente la capacidad de crear criaturas con distinto comportamiento.

Referencia:	CU-12
Nombre:	Crear criatura
Descripción	Permite que el jugador cree una instancia de criatura con el comportamiento y cuerpo seleccionados 1. El jugador apunta al lugar donde desea crear la criatura y la crea 2. El sistema comprueba que el lugar está lo suficientemente cerca y, de no estarlo, encuentra un punto dentro del rango en esa dirección donde crear la criatura. Entonces, crea una instancia del cuerpo seleccionado, construye el comportamiento dado por el jugador y se lo asigna al cuerpo
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-13
Nombre:	Seleccionar cuerpo criatura
Descripción	Permite que el jugador seleccione el cuerpo que tendrán las criaturas que cree 1. El jugador pide al sistema que seleccione el siguiente cuerpo 2. El sistema lo selecciona y actualiza la interfaz para mostrar esta selección
Actor	Jugador
Relaciones	
Precondición	El jugador tiene más de un cuerpo de criatura disponible

Referencia:	CU-14
Nombre:	Seleccionar comportamiento criaturas
Descripción	Permite que el jugador cambie el comportamiento de cada estado y las transiciones entre estados de la inteligencia artificial de una criatura 1. El jugador abre el menú de comportamiento de la criatura 2. El sistema muestra el menú, donde aparecen los estados y las transiciones entre estos representados gráficamente 3. El jugador selecciona uno de los estados 4. El sistema muestra una lista de comportamientos entre los que elegir 5. El jugador elige uno de los comportamientos

	6. El sistema guarda el comportamiento para ese estado y actualiza la interfaz del menú para mostrar el comportamiento en el estado 7. El jugador elige una de las transiciones entre estados 8. El sistema muestra una lista de tipos de transición 9. El jugador elige el tipo deseado 10. El sistema guarda ese tipo para la transición entre estados seleccionada y muestra gráficamente la selección en el menú
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-15
Nombre:	Comportamiento criatura
Descripción	Permite que la criatura se comporte tal como el jugador lo había especificado en el momento de su creación 1. Cada nuevo *frame*, el sistema comunica a la entidad criatura que debe actuar 2. La entidad actúa conforme al estado actual 3. Si el estado actual lanza una transición, cambia el estado para el siguiente *frame*
Actor	-
Relaciones	
Precondición	

Referencia:	CU-16
Nombre:	Estado ‘Moverse’
Descripción	Permite que la criatura se mueva hacia un objetivo 1. Cada frame, el sistema intenta desplazar a la criatura en la dirección del objetivo 2. El sistema comprueba si la criatura está dentro del rango aceptable para considerar que se ha movido hacia el objetivo correctamente. 3. Si está en el rango lanza la transición de ‘Completado’ si esta existe y se cambia el estado 4. Si la criatura no ha conseguido acercarse más al objetivo en un tiempo determinado, se lanza la transición de ‘Bloqueado’ si esta existe y se cambia el estado
Actor	-
Relaciones	
Precondición	La criatura está en el estado de Moverse

Referencia:	CU-17
Nombre:	Estado ‘Agarrar’

Descripción	Permite que la criatura agarre o desagarre un objetivo 1. Al iniciarse el estado el sistema comprueba si el objetivo ya está agarrado por la criatura 2. Si está agarrado ya por la criatura, el objetivo queda desagarrado y se lanza la transición de 'Completado' si esta existe y se cambia el estado 3. Si no estaba agarrado ya, cada *frame* si la acción no se ha llevado a cabo aún, el sistema comprueba si la criatura está lo suficientemente cerca al objetivo para agarrarlo 4. Si está en el rango el objetivo queda agarrado y su movimiento ligado al de la criatura. 5. El sistema lanza la transición de 'Completado' si esta existe y se cambia el estado 6. Si no está en el rango de agarre durante un tiempo determinado, se lanza la transición de 'Bloqueado' si esta existe y se cambia el estado
Actor	-
Relaciones	
Precondición	La criatura está en el estado de Agarrar

Referencia:	CU-18
Nombre:	Estado 'Romperse'
Descripción	Permite que la criatura se destruya automáticamente al llegar a este estado 1. Al iniciarse el estado el sistema destruye la criatura 2. El sistema devuelve al jugador la habilidad de crear una criatura
Actor	-
Relaciones	
Precondición	La criatura está en el estado de Romperse

Referencia:	CU-19
Nombre:	Estado 'Atacar'
Descripción	Permite que la criatura ataque un objetivo si está lo suficientemente cerca hasta que la vida de este llegue a 0 1. Cada *frame* si la acción no se ha llevado a cabo aún, el sistema comprueba si la criatura está lo suficientemente cerca al objetivo para atacarlo 2. Si está en el rango el objetivo lo ataca 3. Si la vida del objetivo llega a 0, el sistema lanza la transición de 'Completado' si esta existe y se cambia el estado 4. Si no está en el rango de ataque durante un tiempo determinado, se lanza la transición de 'Bloqueado' si esta existe y se cambia el estado
Actor	-
Relaciones	
Precondición	La criatura está en el estado de Atacar

Referencia:	CU-20
Nombre:	Estado ‘Esperar’
Descripción	Permite que la criatura espere una cantidad determinada de tiempo antes de continuar al siguiente estado 1. Tras una cantidad determinada de tiempo, el reloj comunica al sistema que ha pasado el tiempo de espera 2. El sistema lanza la transición de ‘Completado’ si esta existe y se cambia el estado
Actor	Reloj
Relaciones	
Precondición	La criatura está en el estado de Esperar

3.1.3 Sistemas emergentes

Referencia:	CU-21
Nombre:	Daño de fuego a alrededores
Descripción	Permite que el fuego de una entidad en llamas se extienda y haga daño a entidades cercanas periódicamente 1. Tras una cantidad determinada de tiempo, el reloj comunica al sistema que la entidad en llamas haga daño a sus alrededores 2. El sistema recoge todas las entidades dentro del rango del fuego 3. Por cada entidad, si esta puede estar en llamas, se pone en llamas 4. Si la entidad tiene vida, y si de tener humedad, está seca, será dañada un número determinado de puntos de vida 5. Si la entidad tiene humedad, deberá ser secada un número determinado de puntos de humedad. Estos puntos se le restarán al calor del fuego 6. Si el calor del fuego llega a 0, la entidad en llamas deberá dejar de estarlo
Actor	Reloj
Relaciones	
Precondición	La entidad está en llamas

Referencia:	CU-22
Nombre:	Humidificación de entidades
Descripción	Permite que una entidad húmeda humedezca las entidades a su alrededor 1. Tras una cantidad determinada de tiempo, el reloj comunica al sistema que la entidad humedezca a las entidades a su alrededor 2. El sistema recoge todas las entidades dentro del rango 3. Por cada entidad, si esta puede estar humedecida, se le añade una

	cantidad de puntos de humedad 4. Si la entidad que está dando esos puntos no es una fuente infinita, esta pierde tantos puntos como ha dado 5. Si una de las entidades pasa a tener más de cierta cantidad de humedad, pasa al estado húmedo 6. Si una de las entidades pasa a tener menos de cierta cantidad de humedad, pasa al estado seco
Actor	Reloj
Relaciones	
Precondición	La entidad está húmeda

Referencia:	CU-23
Nombre:	Apertura de barrera
Descripción	Permite que una barrera se levante dada una condición como estar electrificada 1. El sistema comprueba si se cumple la condición para que se levante la barrera 2. Si se cumple, la barrera es levantada hasta cierto punto predeterminado 3. Si la condición deja de cumplirse en algún momento, la barrera vuelve a bajar hasta el punto original
Actor	-
Relaciones	
Precondición	

Referencia:	CU-24
Nombre:	Objeto rompible
Descripción	Permite que un objeto se rompa cuando su vida llega a 0 1. Un elemento del sistema como una criatura ataca al objeto, restando una cantidad de puntos de vida 2. Si la cantidad de vida del objeto llega a 0, este pasa a romperse, dejando de actuar como un elemento físico capaz de bloquear un camino o sostener algo. Esto es acompañado por un efecto visual que muestra que el objeto se ha roto
Actor	-
Relaciones	
Precondición	

Referencia:	CU-25
Nombre:	Plataformas hidráulicas
Descripción	Permite que haya plataformas hidráulicas que se levanten cuando una entidad esté encima de otra plataforma

	1. Una entidad se desplaza encima de una plataforma pequeña 2. La o las plataformas grandes ligadas a la pequeña se levantan con una velocidad determinada hasta un punto determinado 3. Cuando una entidad se desplaza fuera de la plataforma pequeña, la o las plataformas grandes ligadas bajan con una velocidad determinada hasta su posición original
Actor	-
Relaciones	
Precondición	

3.1.4 Menús

Referencia:	CU-26
Nombre:	Acceder a partida
Descripción	Permite al jugador entrar en una partida ya empezada 1. El jugador selecciona la opción de continuar partida ya empezada desde el menú principal 2. El sistema muestra una pantalla con todas las partidas existentes 3. El jugador selecciona la partida 4. El sistema le da la opción de acceder a la partida o de eliminarla 5. El jugador selecciona acceder a la partida 6. El sistema carga la partida del jugador tal como estaba la última vez que se guardó (CU-35)
Actor	Jugador
Relaciones	Incluye CU-35 Cargado de partida
Precondición	Ya existe al menos una partida creada

Referencia:	CU-27
Nombre:	Eliminar partida
Descripción	Permite al jugador eliminar una partida ya empezada 1. El jugador selecciona la opción de continuar partida ya empezada desde el menú principal 2. El sistema muestra una pantalla con todas las partidas existentes 3. El jugador selecciona la partida 4. El sistema le da la opción de acceder a la partida o de eliminarla 5. El jugador selecciona eliminar la partida 6. El sistema elimina la partida del almacenamiento local y del estado y actualiza la interfaz para mostrar que ya no existe
Actor	Jugador
Relaciones	

Precondición	Ya existe al menos una partida creada

Referencia:	CU-28
Nombre:	Crear nueva partida
Descripción	Permite al jugador crear una nueva partida 1. El jugador selecciona la opción de crear nueva partida desde el menú principal 2. El sistema crea la nueva partida en el almacenamiento local y comienza la secuencia de inicio del juego
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-29
Nombre:	Abrir menú de pausa
Descripción	Permite al jugador abrir un menú que pausa la partida 1. El jugador abre el menú de pausa 2. El sistema paraliza el resto del juego y muestra el menú 3. Se muestran opciones de 'Continuar', 'Opciones' y 'Salir al menú principal'
Actor	Jugador
Relaciones	
Precondición	El jugador está dentro del juego, no en un menú

Referencia:	CU-30
Nombre:	Cerrar menú de pausa
Descripción	Permite al jugador cerrar el menú de pausa 1. El jugador abre el menú de pausa (CU-29) 2. El jugador selecciona 'Continuar' 3. El sistema reactiva el juego y esconde el menú. El juego está en el mismo estado que estaba en el momento de pausarlo
Actor	Jugador
Relaciones	Incluye CU-29 Abrir menú de pausa
Precondición	El jugador está dentro del juego, no en un menú

Referencia:	CU-31
Nombre:	Salir al menú principal
Descripción	Permite al jugador salir al menú principal desde la partida 1. El jugador abre el menú de pausa (CU-29)

	2. El jugador selecciona la opción ‘Salir al menú principal’ 3. El sistema detiene el juego, guarda el progreso (CU-33), descarga los niveles del juego y carga el menú principal 4. Se muestran opciones de ‘Continuar partida’, ‘Crear nueva partida’ y ‘Salir del juego’
Actor	Jugador
Relaciones	Incluye CU-29 Abrir menú de pausa y CU-33 Guardado de partida
Precondición	El jugador está dentro del juego, no en un menú

Referencia:	CU-32
Nombre:	Modificar opciones del juego
Descripción	Permite al jugador entrar al menú de opciones desde el menudo de pausa de la partida y modificar distintas variables a modo de opciones 1. El jugador abre el menú de pausa (CU-29) 2. El jugador selecciona la opción ‘Opciones’ 3. El sistema muestra un nuevo menú en el que se muestran las opciones de ‘Volumen Sonidos’, ‘Volumen música’, ‘Campo de visión’, ‘Punto de mira en pantalla’ y ‘Sensibilidad del ratón’ 4. El jugador selecciona los valores deseados para las opciones y acepta los cambios 5. El sistema guarda estos ajustes (CU-33) y devuelve al usuario al menú de pausa
Actor	Jugador
Relaciones	Incluye CU-29 Abrir menú de pausa y CU-33 Guardado de partida
Precondición	El jugador está dentro del juego, no en un menú

3.1.5 Progreso del jugador

Referencia:	CU-33
Nombre:	Guardado de partida
Descripción	Permite que se guarde el progreso del jugador en almacenamiento local para que el progreso no se pierda entre sesiones 1. El sistema obtiene los datos a guardar: La posición del jugador y de su criatura, la mente y estado actual de la criatura, así como los objetos que el jugador posee en su inventario, el último punto de reaparición alcanzado por el jugador y las opciones seleccionadas. Se guarda además el estado de las partes del nivel que se hayan resuelto 2. El sistema guarda estos datos en el almacenamiento local del ordenador
Actor	-
Relaciones	

Precondición	

Referencia:	CU-34
Nombre:	Cargado de partida
Descripción	Permite que se cargue el progreso del jugador del almacenamiento local 1. El sistema carga los datos desde el almacenamiento local: La posición del jugador y de su criatura, la mente y estado de la criatura, así como los objetos que el jugador posee en su inventario y el último punto de reaparición alcanzado por el jugador, las opciones seleccionadas y el estado de las partes del nivel 2. El sistema cambia el estado de las instancias correspondientes para que se ajusten a los datos cargados
Actor	-
Relaciones	
Precondición	

Referencia:	CU-35
Nombre:	Guardado automático de partida
Descripción	Permite que la partida se guarde automáticamente cada cierto tiempo para evitar la pérdida de datos en caso de cierre del juego de forma inesperada 1. El reloj pide al sistema que se guarden los datos 2. El sistema guarda los datos de la partida (CU-34)
Actor	Reloj
Relaciones	Incluye CU-33 Guardado de partida
Precondición	

Referencia:	CU-36
Nombre:	Activar punto de control
Descripción	Permite que se guarde el avance del jugador para que al morir aparezca en un punto cercano al que está 1. El jugador se desplaza sobre un punto invisible que hace de punto de control 2. El sistema guarda este punto como el último punto de control tocado
Actor	Jugador
Relaciones	
Precondición	

Referencia:	CU-37
Nombre:	Muerte y reaparición del jugador

Descripción	Permite que el jugador pueda morir y volver a aparecer en el último punto de control alcanzado 1. Una entidad ataca al jugador y el sistema comprueba si ha llevado los puntos de vida del jugador a 0 2. El sistema bloquea el movimiento del jugador y activa un efecto visual de fundido a negro. Se mueve el jugador al último punto de control activado, se destruyen las criaturas existentes en este momento de haberlas, se deshace el fundido a negro y se reactiva el movimiento del jugador.
Actor	-
Relaciones	
Precondición	

Referencia:	CU-38
Nombre:	Pantalla de final de juego
Descripción	Permite que al llegar al final de los niveles se muestre una pantalla que comunique al jugador que ha finalizado el juego 1. El jugador se desplaza sobre un punto invisible que hace de punto final del juego 2. El sistema muestra un mensaje que comunica al jugador que ha llegado al final y le muestra su puntuación (cantidad de coleccionables obtenidos) respecto a la puntuación máxima posible
Actor	Jugador
Relaciones	
Precondición	

3.2 Requisitos No Funcionales

Se han especificado los siguientes requisitos no funcionales:

RNF-01	El juego deberá funcionar en un ordenador con sistema operativo Windows
RNF-02	El juego deberá cargar el primer nivel en menos de 5 minutos
RNF-03	El juego deberá cargar el menú de inicio en menos de 1 minuto
RNF-04	El juego deberá estar disponible en español

3.3 Análisis de riesgos

El mayor riesgo que se advierte es el riesgo de aceptación por parte del usuario. El riesgo de aceptación conlleva que, a pesar de haber desarrollado el sistema tal como

exponen los requisitos, y a pesar de que este funcione correctamente, acabe dándose el caso de que este no era el sistema que necesitaba el usuario.

La mecánica de juego principal no es directamente intuitiva para el usuario común de un videojuego como lo sería disparar. Al ser una mecánica de juego compleja y poco utilizada, es muy difícil determinar si va a producir una reacción positiva en los jugadores antes de ponerlo en sus manos. Además, es muy difícil saber si los usuarios van a saber cómo utilizarla correctamente, y qué grado de ayuda e información explícita necesitan para usarla en primer lugar. Los jugadores prefieren sentir que están aprendiendo cómo jugar por su cuenta en lugar de leer un manual de uso. Debido a esto, en el desarrollo del videojuego se busca encontrar el punto justo de información necesaria para que el jugador pueda hacer uso de las mecánicas del juego.

Es por estos motivos por lo que se debe realizar varias tomas de contacto lo antes posible en el desarrollo para observar cómo responden los jugadores a la mecánica, y se planea corregir los errores que se observen de los resultados de estas pruebas.

3.4 Identificación y análisis de soluciones posibles

Existen diversos motores de videojuegos para tener en cuenta para la realización de este proyecto. Una de las características que definen el juego es que es en 3D, lo que significa que necesitamos un motor capaz de soportar esto. Los motores 3D que vamos a considerar serán Unreal Engine 4, Untiy3D, Godot, y Armory3D.

El motor Godot está orientado a la facilidad de aprendizaje y de uso, sobre todo para usuarios sin mucha experiencia programando. Sin embargo, aún está en una fase temprana del desarrollo y no tiene la potencia ni las funcionalidades para competir con el resto de los motores gráficos teniendo en cuenta que en este proyecto la facilidad de aprendizaje no es un requerimiento.

Armory3D es un motor también en fase temprana de desarrollo. Su ventaja frente al resto de los motores reside en su integración de Blender en el proceso de desarrollo. Blender es una herramienta de modelado 3d con capacidades de animación y texturizado, que es usado a menudo para desarrollar recursos gráficos (*assets*) para su uso dentro de videojuegos. La integración directa de la herramienta con Armory3D permite un flujo de trabajo mucho más sencillo y rápido que la importación manual necesaria en los otros motores. Sin embargo, a Armory3D también le falta funcionalidad y potencia hallada en los dos motores siguientes.

Unreal Engine 4 es un motor gráfico muy potente enfocado a la creación de juegos de gran fidelidad gráfica. Cuenta con un sistema de programación visual para ayudar en la construcción de la lógica del juego y además cuenta con el apoyo financiero de *Epic*, la empresa desarrolladora, que otorga becas a muchos de los juegos realizados con el motor. Unreal Engine 4 también cuenta con ayuda online y un mercado de *assets*, aunque ambos están limitados respecto a Unity3D.

Unity3D es un motor capaz de producir juegos de alta fidelidad, pero más enfocado a la facilidad de uso. El motor es uno de los más populares entre los desarrolladores independientes, y por lo tanto es del que más información sobre su uso existe en la

web. Además, contiene un mercado de *assets* (elementos artísticos y herramientas que suplen funcionalidad específica) muy variado y completo, además de facilidad para la modificación de estos para adaptarlos a un juego. Esto permite realizar juegos de forma más sencilla y eficiente por equipos más pequeños o individuos, como se da el caso en este proyecto.

3.5 Solución propuesta

Como se ha comentado con anterioridad en la sección 2.2, el motor que se ha decidido utilizar es Unity3D. Esto permitirá la utilización de *assets* que ayudarán a concentrar el esfuerzo en el desarrollo de los sistemas del juego.

Además, este motor ya ha sido utilizado en asignaturas como PSW, donde se realizó un videojuego en 2D, IPV, donde se realizó un videojuego en 3D y ADV, donde se realizaron animaciones. Por esto, ya se dispone de la suficiente experiencia para que la facilidad de uso sea mucho mayor actualmente que la que proporcionaría otro motor.

Para complementar los *assets* de la tienda de Unity donde sea necesario se han decidido utilizar la herramienta Blender, una herramienta de modelado 3D, y PaintdotNet, una herramienta de edición de imágenes y dibujo básico.

Se ha decidido utilizar Jira como herramienta de planificación y gestión del desarrollo.

Teniendo en cuenta el análisis de riesgos, se ha decidido realizar pruebas de aceptación en varios puntos del desarrollo para comprobar la validez del juego.

3.6 Planificación

Se van a realizar tres *sprints* desde mayo hasta julio, durando cada *sprint* tres semanas. Las tareas del proyecto se han agrupado en seis Épicas, las cuales se muestran en la planificación global dentro de Jira en la Figura 2:

- Desarrollo del sistema modular de IA: Esta épica engloba las tareas relacionadas con la construcción del sistema modular por el cual el jugador es capaz de diseñar la inteligencia artificial que tendrán las criaturas que cree.
- Diseño y construcción de nivel(es) del juego: Tareas relacionadas tanto con el diseño de los niveles del juego, utilizando herramientas externas, como con la generación y búsqueda de recursos necesarios y el montaje propio de los niveles.
- Desarrollo sistemas de jugabilidad emergente: Tareas relacionadas con los sistemas emergentes explicados más detalladamente en el apartado 4.2.
- Desarrollo sistemas del jugador: Movimiento del jugador, menús, sistemas de inventario, muerte y reaparición, recompensa, etc. destinados a crear una experiencia de juego completa.
- Refinamiento: Retoques destinados a mejorar la calidad general del juego
- Validación del juego: Engloba la realización de pruebas de validación, centradas en comprobar que el juego que se está desarrollando es entretenido para los jugadores. Estas pruebas empezarán tan pronto como el sistema esté

desarrollado y exista una porción del juego jugable para obtener información lo antes posible.

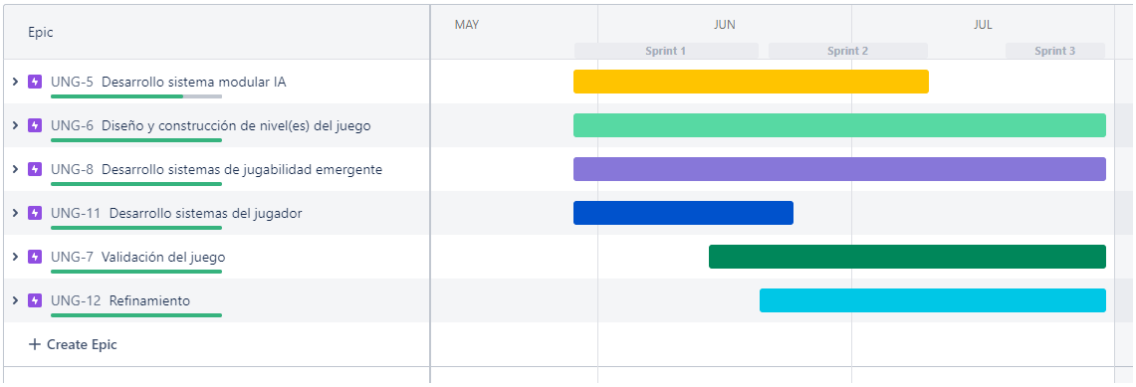


Figura 2

El primer *sprint* tiene como objetivo principal la implementación de la mecánica principal del juego y el desarrollo del primer nivel. Se ha de realizar un cuestionario para los jugadores que prueben el juego. Las respuestas al cuestionario serán usadas para generar puntos a arreglar en el siguiente sprint.

El segundo *sprint* tiene como objetivo principal el desarrollo de los sistemas emergentes del juego, así como de elementos de nivel que sean afectados por estos. También se llevarán a cabo los cambios diseñados a partir de las respuestas del cuestionario.

El tercer y último *sprint* tiene como objetivo la creación de los menús del juego, así como el sistema de guardado y la construcción del segundo nivel del juego. Se ha realizado una última tanda de pruebas para valorar la aceptación final de los jugadores.

4. Diseño de la solución

4.1 Arquitectura del sistema

Se ha utilizado una arquitectura basada en la arquitectura Model-View-Controller. “En una arquitectura Model-View-Controller los objetos de distintas clases manejan las operaciones relacionadas con el dominio de la aplicación (el modelo), la muestra del estado de la aplicación (la vista) y la interacción del usuario con el modelo y la vista (el controlador)”[10] (Stephen Pope, Glenn E. Krasner, 1988). En la Figura 3 se muestra el diseño de esta arquitectura:

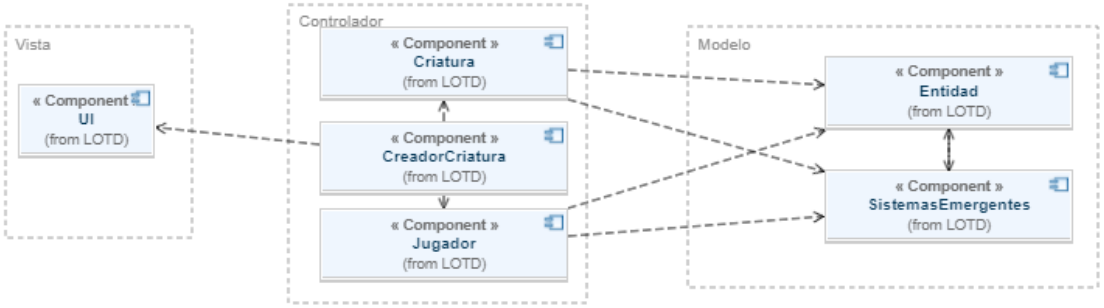


Figura 3 Diseño arquitectónico

Los elementos del modelo interactúan entre sí a través de la funcionalidad ofrecida por Unity. Los componentes del modelo y del controlador tienen funciones de actualización (*Update*) y comienzo (*Start*) que son llamadas por el motor de Unity al principio de cada *frame* y en el primer *frame* en el que está activo el objeto que tiene el componente dado, respectivamente. Los componentes hacen llamadas a través del motor de Unity para encontrar los objetos con los que quieren comunicarse.

Los componentes de la Criatura, el CreadorCriatura y Jugador son parte del controlador ya que manejan lógica para actualizar la vista y las entidades del modelo. Se explicarán en detalle en el siguiente apartado, pero las clases dentro de Criatura se encargan de la lógica de la máquina de estados de las criaturas creadas por el jugador, y a su vez realizan cambios sobre la entidad asociada a la criatura y las entidades objetivo.

Las clases de CreadorCriatura se encargan de manejar los inputs del usuario para obtener las características de la inteligencia artificial a crear. Además, habilitan y deshabilitan las partes de la UI necesarias para este proceso. Se encarga también de construir esta IA y crear la entidad de la criatura con ella.

Las clases de Jugador manejan los inputs de movimiento e interacción con el entorno del jugador, además de la lógica que gobierna el movimiento del personaje y los distintos estados en los que se puede encontrar.

4.2 Diseño detallado

La figura 4 muestra las clases más relevantes del videojuego. A continuación, se detallará el funcionamiento y diseño de cada uno de los componentes.

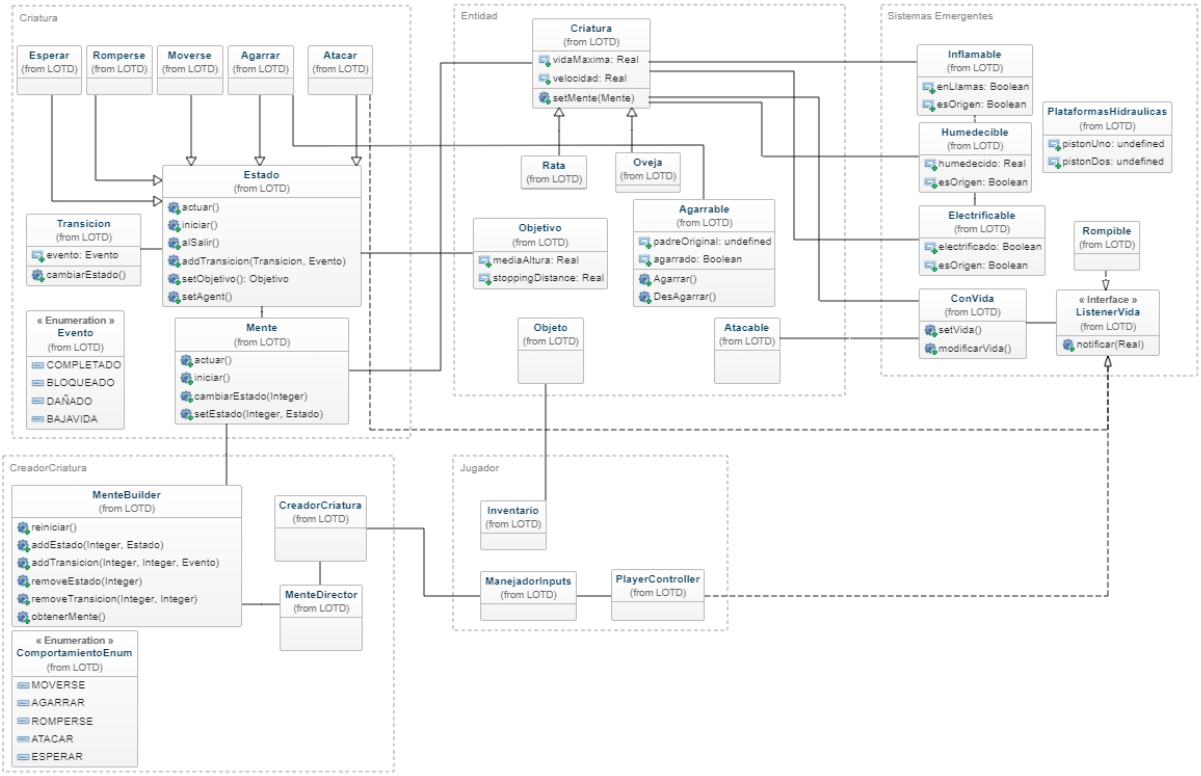


Figura 4 Diseño y relaciones de todas las clases relevantes

Criatura:

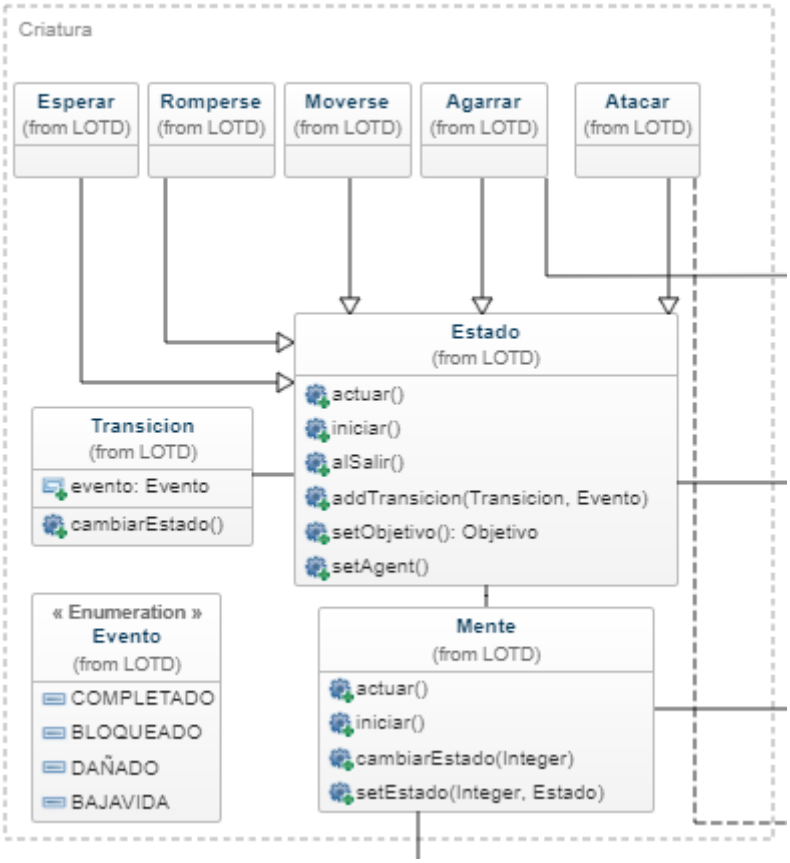


Figura 5 Diseño de clases del componente Criatura

Estas clases, mostradas en la figura 5, se encargan de manejar la lógica de la inteligencia artificial de las criaturas del jugador. Disponen de un objeto Mente que utiliza el patrón *State* para ejecutar distinta funcionalidad con el mismo código llamando a los métodos abstractos de Estado. La funcionalidad depende de qué descendiente de Estado es el estado actual, ofreciéndose ahora mismo cinco alternativas: Esperar, Romperse, Moverse, Agarrar y Atacar.

Estas clases son clases que heredan de la clase abstracta Estado e implementan los métodos abstractos de esta. La funcionalidad definida por cada estado es una implementación de los métodos actuar (lanzado cada frame), iniciar (lanzado al cambiar a este estado por primera vez) y alSalir (lanzado cuando se cambia desde este estado o la criatura muere).

Cada Estado puede tener una lista de Transición, que a su vez están relacionadas con un Evento específico. Cuando un Estado lanza un Evento dado, se ejecuta la Transición asociada a ese Evento, si existe. Esta Transición cambia el Estado actual en el objeto Mente, permitiendo que cambie la funcionalidad a ejecutar por la Mente.

CreadorCriatura:

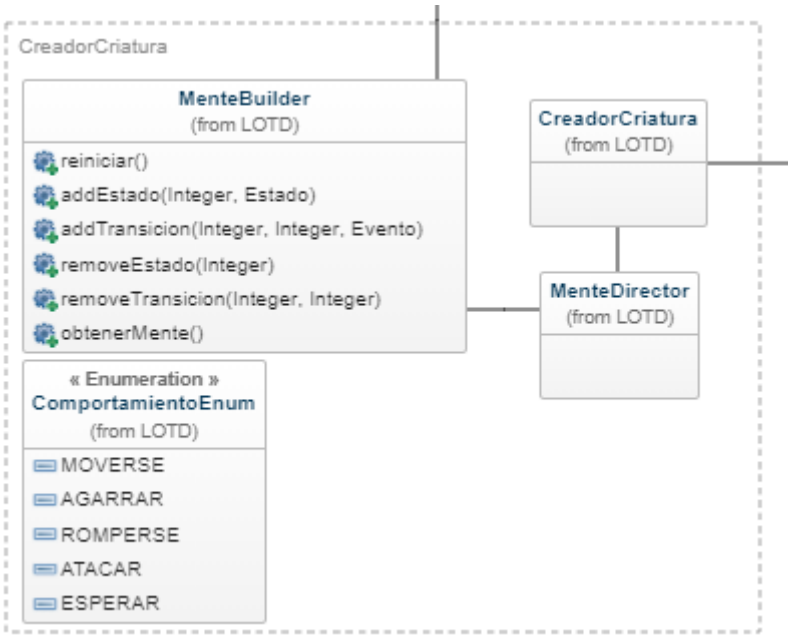


Figura 6 Diseño de clases del componente CreadorCriatura

CreadorCriatura toma los inputs del usuario y recaba los datos de la inteligencia artificial que el usuario está creando. Cuando el jugador crea una criatura, esta clase llama a MenteDirector con estos datos, la cual utiliza los métodos de MenteBuilder para construir un objeto Mente con las características deseadas haciendo uso del patrón Builder. Estas clases y sus relaciones se muestran en la figura 6.

“El patrón Builder separa la construcción de un objeto complejo de su representación, para que el mismo proceso de construcción pueda crear diferentes representaciones” "[11] (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994). Se basa en definir la construcción de un objeto como una serie de pasos e instrucciones que tras su ejecución producen nuevas instancias de un objeto con diferentes propiedades dependiendo de la implementación del Builder.

Otro de los objetivos del patrón Builder es el de simplificar la creación de objetos complejos, como se da el caso en este proyecto donde la clase Mente a construir con sus estados es compleja. Si no se utilizara el patrón Builder sería necesario crear muchos métodos constructores para todas las combinaciones posibles de Mente, y sería mucho más costoso añadir nuevas implementaciones de Estado. Con el patrón Builder, el código que construye el objeto Mente no necesita cambiar cuando se implementan nuevos estados.

El patrón Builder se ha complementado con un director. El director se encarga de llamar a los métodos del Builder en el orden necesario dados unos datos de entrada para que la mente se construya y funcione correctamente. Sin el director, el código de llamada a los métodos del Builder estaría repetido en cada lugar donde se instancie un objeto Mente. Además, el director permite que si es necesario cambiar cómo se llama a los métodos del Builder, esto solo necesita afectar a la clase del director; las clases que usan el director para instanciar el objeto Mente no necesitan conocer los cambios.

Entidad:

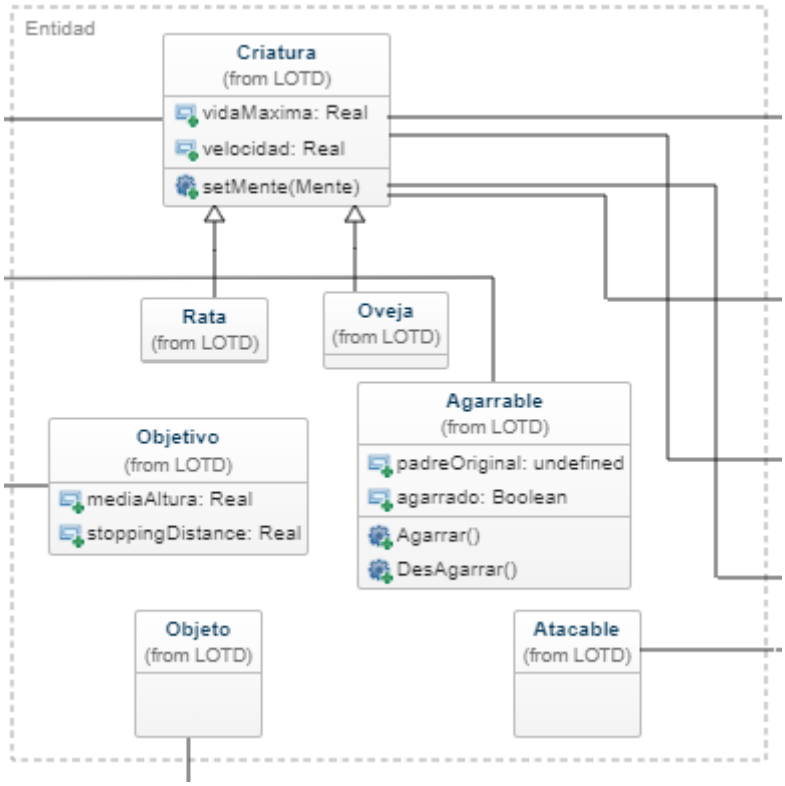


Figura 7 Diseño de clases del componente Entidad

Se trata de clases que representan la información de una entidad dada. Estas clases y sus relaciones se muestran en la figura 7. La clase Criatura contiene la vida y velocidad de la criatura creada, y tiene dos clases que heredan de ella, Rata y Oveja, que especifican los atributos para los dos tipos de criatura.

La clase Objetivo representa un objetivo para un estado de la mente de la criatura, con datos necesarios para el correcto funcionamiento de los estados respecto a este. Por ejemplo, el atributo ‘stoppingDistance’ de Objetivo permite a la criatura conocer como de cerca debe de estar del objetivo para considerar que ya lo ha alcanzado cuando su estado es moverse hacia este. Esto varía de objetivo a objetivo debido a su tamaño y/o localización en el nivel. Además, una entidad que no tiene el componente Objetivo asociado no puede ser seleccionada como objetivo.

Agarrable y Atacable son componentes que, cuando usados junto a Objetivo, permiten saber si una criatura puede agarrar o atacar respectivamente al objetivo. Además, contienen información relevante para el correcto funcionamiento del estado como por ejemplo la clase Atacable que contiene una referencia al componente ‘ConVida’ de la entidad y permite que el estado Atacar se suscriba y ataque a este.

La clase Agarrable contiene información sobre el padre original del objetivo en la jerarquía, además de si el objetivo ya está agarrado. Cuando un objetivo es agarrado, su padre pasa a ser la criatura que lo ha agarrado. Debido al funcionamiento de Unity, una entidad hija se desplaza y rota en función a la posición y rotación de su padre. Dicho de otro modo, al ser padre del objetivo, cuando la criatura se desplaza la posición del objetivo se actualiza automáticamente para estar en la misma posición relativa a la criatura.

Sin embargo, para que la criatura deje de agarrar el objetivo es necesario que se deshaga este emparejamiento, ya que ya no se busca que la criatura y el objetivo se muevan conjuntamente. Además, cuando una criatura es destruida, todos los objetos descendientes de la entidad en la jerarquía son destruidos automáticamente, y esto no debería ocurrir para un objeto agarrado. Por estos motivos es necesario que la clase Agarrable guarde el padre original, para poder ser restituido cuando la criatura desaparezca o lo desagarre.

Objeto contiene el nombre del objeto y la cantidad a otorgar cuando el jugador entre en contacto con la entidad. Este objeto se añade al Inventario del jugador.

SistemasEmergentes:

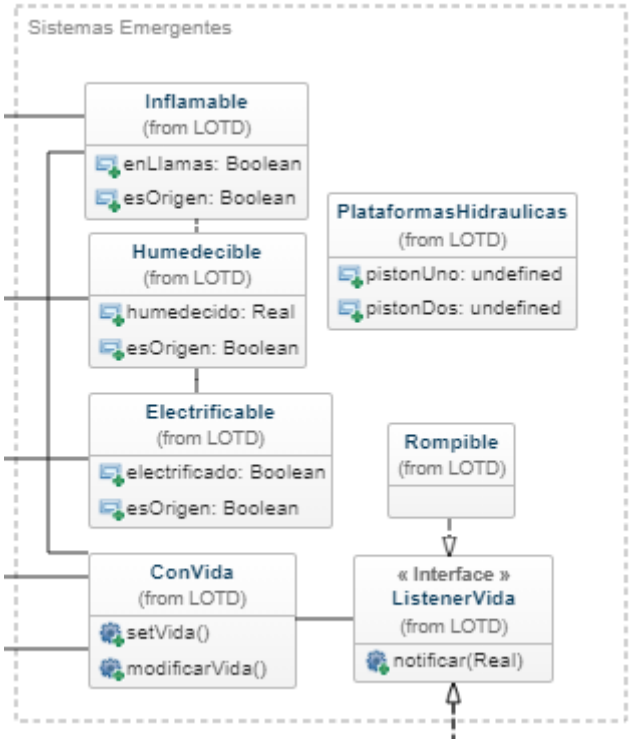


Figura 8 Diseño de las clases del componente Sistemas Emergentes

Estas clases, mostradas en la figura 7, contienen los datos y lógica necesarios para el funcionamiento de sistemas que interactúan entre sí.

La clase ConVida controla los puntos de vida actuales de una entidad. Esta clase sigue el patrón 'Observer', teniendo una lista de objetos que implementan ListenerVida a los que notifica cuando se modifican los puntos de vida con la nueva cantidad. Cada implementación de ListenerVida define qué debe ocurrir cuando recibe la notificación. Por ejemplo, la clase PlayerController implementa ListenerVida y cuando la vida llega a 0 el jugador muere y reaparece en un punto anterior. Rompible es otra implementación que se usa en elementos del escenario que deben romperse cuando la vida llegue a 0.

PlataformasHidraulicas es una clase que hace subir un set de plataformas que se levantan cuando una entidad se posa encima de un botón en el suelo del escenario, y vuelven a bajar cuando la entidad ya no está encima del botón.

Inflamable, Humedecible y Electrificable definen unos sistemas directamente relacionados entre sí. Cada uno de estos objetos pueden estar o no en los estados que describen, y las clases que heredan de estas definen qué ocurre dependiendo del estado en el que esté la entidad. También disponen de un atributo ‘esOrigen’ que indica que la entidad siempre estará en el estado dado.

Las entidades inflamables disponen de un valor de calor. Varias veces por segundo, estas entidades buscan otras entidades dentro de su rango. Si la entidad es humedecible esta entidad inflamable reduce su propio calor y a su vez la humedad de la otra entidad. Si la entidad es atacable y no está húmeda o no es humedecible, la entidad recibe daños. Por último, si la entidad es inflamable, no está inflamada actualmente y está seca o no es humedecible, la entidad es inflamada.

Las entidades humedecibles disponen de un valor de humedad. Cuando este valor es menor a un umbral, la entidad se considera seca y cuando es mayor a otro umbral, se considera húmeda. En el caso intermedio no se considera de ninguna de las dos maneras. Varias veces por segundo busca otras entidades dentro de su rango. Si estas son humedecibles, aumenta su humedad disminuyendo la de esta entidad.

Las entidades electrificables pueden estar en estado electrificado o no. De estar electrificadas, buscan otras entidades dentro de su rango. Si estas entidades son electrificables, las electrifican, creando una corriente visual que demuestra la conexión entre ambas. Las entidades electrificables tienen dos rangos distintos en los que buscar, uno normal y otro, más grande, que busca entidades húmedas. Debido a esto, una entidad húmeda puede recibir electricidad desde mayor distancia que una no húmeda.

Jugador:

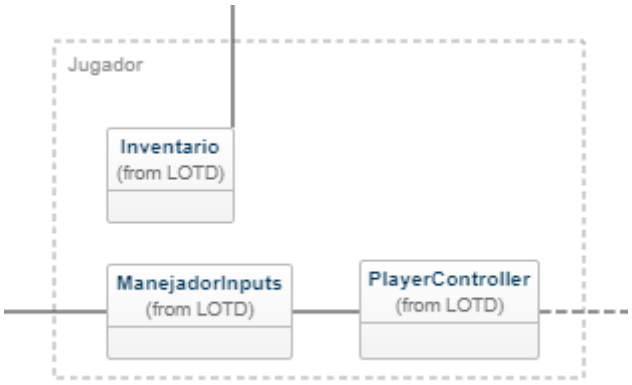


Figura 9 Diseño de las clases del componente Jugador

En la figura 8 se muestran las clases del componente Jugador. La clase ManejadorInputs define las entradas de teclado y ratón del jugador y los métodos para comprobar su estado. Estos métodos son usados por las clases PlayerController y CreadorCriatura.

La clase PlayerController utiliza estos métodos para permitir al jugador realizar acciones como saltar, desplazarse o agacharse. Define la lógica de cómo deben comportarse estas acciones y como debe afectar la gravedad al jugador. Además,

controla el movimiento de la cámara en primera persona del juego. Por último, está suscrito al objeto ConVida del jugador y cuando este llega a 0 activa la secuencia de muerte y reaparición en un punto anterior del mapa.

La clase Inventario maneja un diccionario de objetos que tiene en posesión el jugador. Cuando un nuevo objeto es añadido, notifica a ciertas clases que necesiten actualizarse a través del patrón Observer. Un ejemplo es un contador de 'gotas de agua pura', un coleccionable del juego. Este contador se actualiza cada vez que se recibe una de estas gotas para mostrar la puntuación actual del jugador. Otro de los usos del Inventario es comprobar si el jugador posee un objeto en específico, como una llave o la habilidad de crear criaturas de tipo Oveja.

4.3 Tecnología utilizada

La tecnología principal utilizada ha sido Unity y C#. Unity3D ha proporcionado el motor base sobre el que se ha desarrollado y en el que se han apoyado la mayoría de las otras clases del juego.

Para la generación de algunas imágenes se ha utilizado la herramienta gratuita Paint.Net.

Para la construcción de los niveles del juego se ha usado una herramienta que extiende a Unity llamada 'ProBuilder'. Esta herramienta ha permitido la edición básica de los vértices de modelos simples en el escenario, ayudando a crear la escena de forma más sencilla. También se ha utilizado Blender, una herramienta de modelado 3D, para algún modelo que no podía realizarse fácilmente con ProBuilder.

5. Desarrollo de la solución

A continuación, se va a detallar la implementación de las clases más importantes para el funcionamiento del juego. Cuando una clase es referenciada como un componente, se hace referencia a un concepto de Unity. En el motor de Unity, un juego está dividido en escenas, que representan distintas partes del juego. Una escena a su vez está compuesta por *GameObject*s en una jerarquía, cada uno de los cuales representa una entidad del juego, ya sea un elemento estático de la geometría del nivel o un personaje que se mueve por la escena.

Los *GameObject* pueden tener asociados scripts escritos en C# que heredan de la clase MonoBehaviour proporcionada por Unity. Estos scripts son llamados también componentes, y Unity proporciona funcionalidad especial para que estos componentes interactúen con el *GameObject*, con otros componentes asociados al *GameObject* e incluso con componentes de otros *GameObject* en el juego. Las partes específicas de esta funcionalidad que han sido de utilidad en el desarrollo del juego se explicarán a medida que se expliquen las clases que las utilizan.

Vamos a comenzar por la implementación de la mecánica principal del juego:

La clase Criatura.cs, como se ha explicado en el apartado 4, contiene estadísticas específicas de la criatura, como la velocidad. Además, como componente, se encarga de pasar los eventos de Update y OnDestroy, mostrados en la figura 10, a la instancia de Mente de la criatura.

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    if(mente != null)
    {
        mente.actuar();
    }
}

Unity Message | 0 references
private void OnDestroy()
{
    try
    {
        if (mente != null)
        {
            mente.destruirse();
        }
    }catch(Exception ) {}

    if(creadorCriatura != null)
    {
        creadorCriatura.criaturaMuerta();
    }
}
```

Figura 10 Métodos Update y OnDestroy de clase Mente

El evento OnDestroy se lanza cuando el objeto al que está asociado el componente es destruido, lo que permite liberar recursos. Esto ocurre cuando el jugador reclama la criatura o esta muere.

La clase Mente se encarga de hacer actuar el Estado actual cada *frame*, manejar los cambios de estado que se produzcan, inicializar correctamente los estados y asegurar que salen correctamente cuando la criatura sea destruida, en la figura 1 s1e muestran los métodos de actuar e iniciar.

```
1 reference
public void actuar()
{
    if(!destruida && estado != null)
    estado.actuar();
}
1 reference
public int iniciar(NavMeshAgent agent, Objetivo[] objetivos, Transform transform)
{
    for(int i = 0; i< numEstados; i++)...
    cambiarEstado(0);
    estado.inicializar();
    return 0;
}
```

Figura 11 Métodos actuar e iniciar de clase Mente

Como se verá más adelante, cada instancia de Estado implementa métodos de actuar e inicializar.

```
public void cambiarEstado(int posicion)
{
    if(estado != null)
    {
        try
        {
            estado.salir();
        }
        catch(Exception ) { }
    }
    Estado estadoObjetivo = arrayEstados[posicion];
    if(estadoObjetivo != null)
    {
        estado = estadoObjetivo;
        estado.inicializar();
    }
}
1 reference
public void destruirse()
{
    destruida = true;
    for(int i = 0; i < numEstados; i++)
    {
        Estado est = arrayEstados[i];
        if(est != null)
        {
            est.alDestruirse();
        }
    }
}
```

Figura 12 Métodos cambiarEstado y destruirse de Mente

En el cambio de estado, mostrado en la figura 12, se llama al método ‘salir’ del estado actual, y el método ‘inicializar’ del nuevo estado. Al destruirse la criatura, se llama al

método ‘alDestruirse’ de todos los estados para evitar errores por la influencia de la criatura en otros objetos.

La clase abstracta Estado define una serie de métodos abstractos además de la funcionalidad general que se aplica a todas las instancias de un estado.

```
6 references
public abstract void actuar();
6 references
public abstract void iniciar();
6 references
public abstract void alSalir();
7 references
public abstract void alDestruirse();
```

Figura 13 métodos abstactors actuar, iniciar, alSalir y alDestruirse de la clase Estado

Estos métodos abstractos mostrados en la figura 13 permiten implementar la funcionalidad de cada clase que hereda de estado de forma distinta, lo cual permite a la clase Mente actuar de forma independiente al tipo de estado actual. Esto significa que no es necesario hacer cambios en Mente o Estado para implementar nuevos comportamientos para la inteligencia artificial, sino que simplemente estos comportamientos deben implementar los métodos abstractos de Estado.

```
protected void notificar(Evento evento)
{
    bool existeTransicion = transiciones.TryGetValue(evento, out Transicion transicion);
    if (existeTransicion)
    {
        transicion.cambiarEstado();
    }
}
```

Figura 14 método notificar de la clase Estado

Dentro de la funcionalidad común, Estado cuenta con un *Map* de transiciones, siendo cada Transición un objeto que guarda el estado al que se debe llegar y que tiene un método ‘cambiarEstado’ que realiza la transición.

Estado dispone de un método ‘notificar’ mostrado en la figura 14 que es usado por cada comportamiento para avisar de ciertos eventos, por ejemplo, el evento de tarea completada. Cuando esto ocurre, se busca la transición que corresponde a ese evento, si es que existe, y se ejecuta ‘cambiarEstado’. Esta es la diferencia del patrón Estado respecto al patrón Estrategia: el Estado decide cuándo y cómo cambia el comportamiento del objeto, sustituyéndose a sí mismo por el siguiente Estado.

A continuación, se describe la clase Agarrar como ejemplo de uno de los estados. Su método actuar se muestra en la figura 15

```
6 references
public override void actuar()
{

    float distToObjetivo = (piesObjetivo(objetivo) - transform.position).magnitude;

    if (!completado && distToObjetivo - distanciaAgarrar < objetivo.stoppingDistance)
    {
        agarrar();
    }

    if (tiempoActualBloqueado >= tiempoTopeBloqueado)
    {
        tiempoActualBloqueado = 0;

        if (!completado)
        {
            notificar(Evento.BLOQUEADO);
        }
    }

    tiempoActualBloqueado += Time.deltaTime;
}
```

Figura 15 método actuar de la clase Agarrar

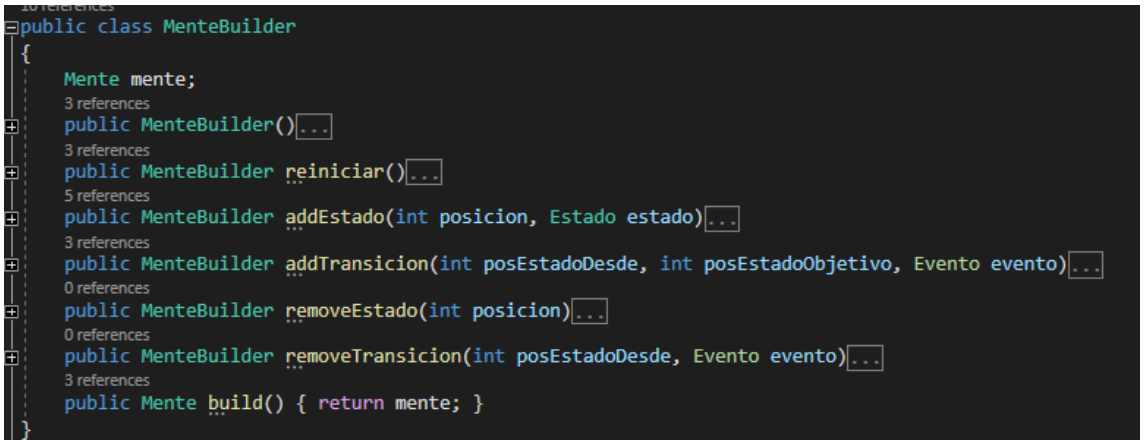
En este estado la criatura intenta agarrar un objeto, para lo que debe estar dentro del rango. Si pasa un tiempo predefinido (dado por 'tiempoTopeBloqueado') sin estar lo suficientemente cerca para agarrar el objeto, emite el evento Bloqueado.

Dentro del método agarrar el objeto se pone por debajo de la criatura en la jerarquía de Unity, permitiendo que cuando la criatura se mueva el objeto se mueva con esta. Tras realizar este agarre correctamente, se lanza el evento Completado.

```
public override void alDestruirse()
{
    desAgarrar();
}
```

Figura 16 Método alDestruirse de la clase Agarrar

También es interesante como ejemplo de la utilidad del método 'alDestruirse' que se llama sobre cada estado cuando la criatura es destruida. Como se muestra en la figura 16, la implementación de agarrar de este método desagarra el objetivo. De no desagarrar el objeto, este quedaría también destruido al estar en ese momento bajo la criatura en la jerarquía, debido al funcionamiento de Unity, lo cual no es deseado.



```
public class MenteBuilder
{
    Mente mente;
    3 references
    public MenteBuilder()...
    3 references
    public MenteBuilder reiniciar()...
    5 references
    public MenteBuilder addEstado(int posicion, Estado estado)...
    3 references
    public MenteBuilder addTransicion(int posEstadoDesde, int posEstadoObjetivo, Evento evento)...
    0 references
    public MenteBuilder removeEstado(int posicion)...
    0 references
    public MenteBuilder removeTransicion(int posEstadoDesde, Evento evento)...
    3 references
    public Mente build() { return mente; }
}
```

Figura 17 MenteBuilder

```
public class MenteDirector
{
    1 reference
    public static Mente menteDesdePasos(MenteBuilder builder, PasosMente pasos)
    {
        builder.reiniciar();
        if (pasos.estados[0] == null)
        {
            pasos.estados[0] = ComportamientoEnum.MOVERSE;
        }
        for (int i = 0; i < pasos.numEstados; i++)
        {
            Estado estado = comportamientoDesdeEnum(pasos.estados[i]);
            if (estado != null)
                builder.addEstado(i, estado);
        }
        for (int i = 0; i < pasos.numEstados; i++)
        {
            for (int j = 0; j < pasos.numEstados; j++)
            {
                if (pasos.eventos[i][j] != null)
                {
                    Evento evento = (Evento)pasos.eventos[i][j];
                    builder.addTransicion(i, j, evento);
                }

            }
        }
        return builder.build();
    }
```

Figura 18 MenteDirector

MenteBuilder contiene los métodos para crear una mente, mientras que MenteDirector contiene la lógica de cómo deben ser llamados estos, dado un objeto PasosMente que contiene los estados y transiciones que debe tener la mente. Parte de estas dos clases se muestran en las figuras 17 y 18.

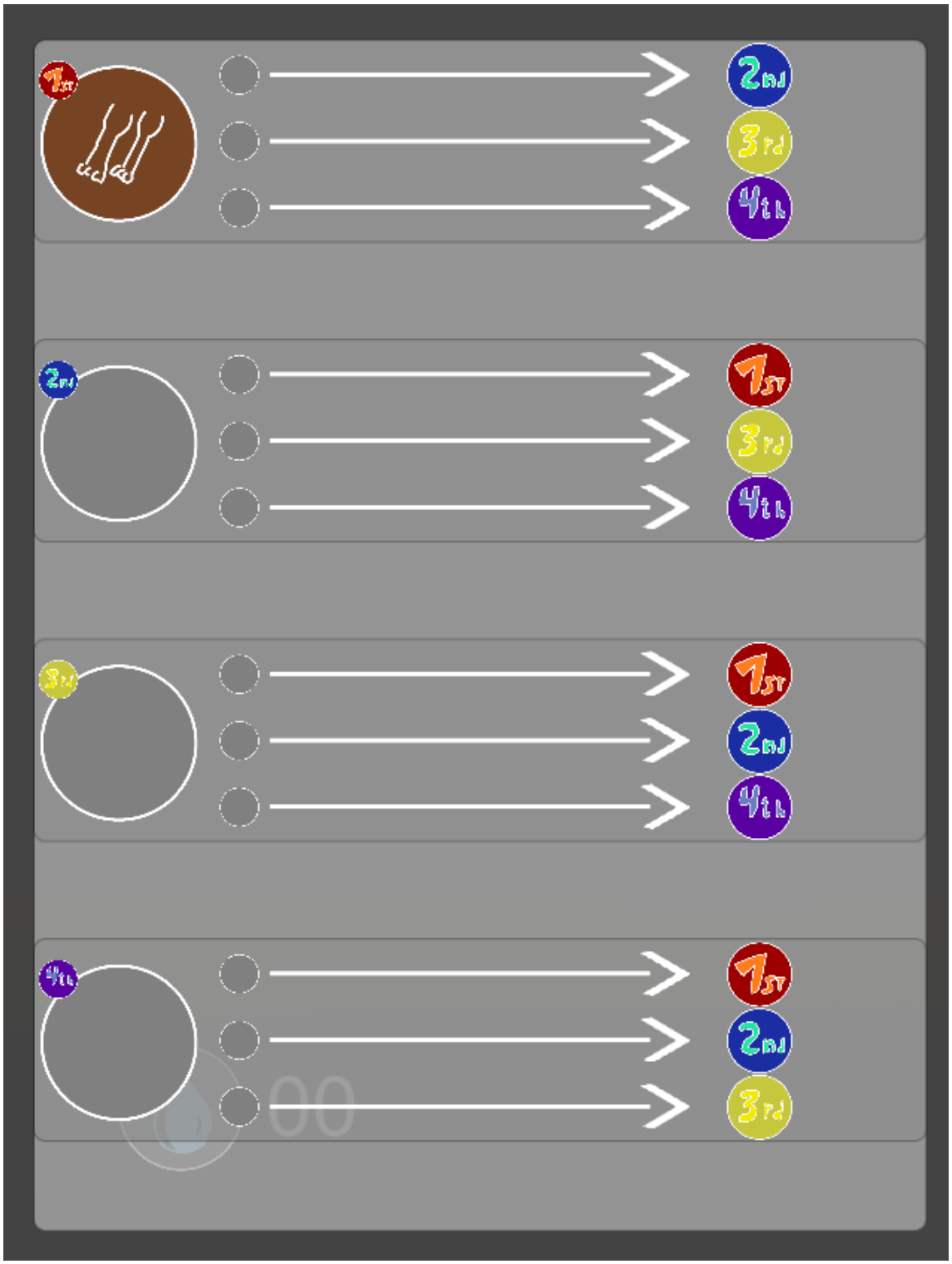


Figura 19 Menú de inteligencia artificial

En la figura 19 se muestra el menú que permite al jugador diseñar el comportamiento de las criaturas. A la izquierda se encuentran los cuatro estados en los que puede estar la máquina de estados de la criatura. Cuando se clica sobre estos estados aparece una lista con los tipos de estado para elegir. A la derecha de cada estado se encuentran tres flechas que representan transiciones hacia otros estados. Clicando en una de estas flechas se puede seleccionar el evento que debe ocurrir para pasar a el estado al que apunta la flecha.

En cuanto a los sistemas emergentes, ha sido especialmente útil una de las funciones de Unity dentro del espacio de nombres ‘Physics’ llamada ‘OverlapBox’. Esta función permite obtener todas las entidades en un área dando las dimensiones y el lugar de la caja imaginaria donde buscar dichas entidades. Esto permite por ejemplo que un objeto en llamas busque otros objetos a su alrededor para prenderlos.

A continuación, se describe la funcionalidad de movimiento del jugador, definida en la clase PlayerController.cs, una parte de la cual se muestra en la figura 20.

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    canMove = true;
    inputs = GetComponent<InputsHandler>();
    controller = GetComponent<CharacterController>();
    Cursor.lockState = CursorLockMode.Locked;
    centerCamRot = camPivot.localRotation;
    lastCheckPoint = transform.position;
    ConVida conVida = GetComponent<ConVida>();
    conVida.Suscribirse(this);
    fadeToBlack.Suscribirse(this);
}

// Update is called once per frame
Unity Message | 0 references
void Update()
{
    if (canMove)
    {
        handleHorizontalMovement();
        handleJump();
        handleCamera();
        handleCrouch();
    }
}
```

Figura 20 clase PlayerController

Como podemos ver en el método Update() se llama a varias funciones que controlan distintos aspectos del movimiento, en caso de que el jugador pueda moverse actualmente. El método Update() es un método especial que pueden tener los componentes. Unity se encarga de llamar al método Update de los componentes de los *GameObject* activos en cada *frame* de ejecución.

Start es otro método especial que es llamado en el primer *frame* en el que la entidad que tiene la instancia de este componente está activa. En este caso, así como en la mayoría de las clases, el método Start se utiliza para inicializar las variables del componente, así como acceder a los otros componentes del objeto. Esto último se

hace a través del método proporcionado por Unity GetComponent<T>(), donde T es la clase del componente al que se quiere acceder.

Cabe destacar que no siempre es necesario acceder así a los componentes. Unity permite que a los campos públicos o con el tag '[SerializeField]' se les agregue una referencia desde el editor visual de la aplicación. Si el campo es primitivo, como un float, puede recibir un valor desde el editor. Esto permite que distintas instancias del mismo componente en distintos objetos tengan valores y referencias distintas.

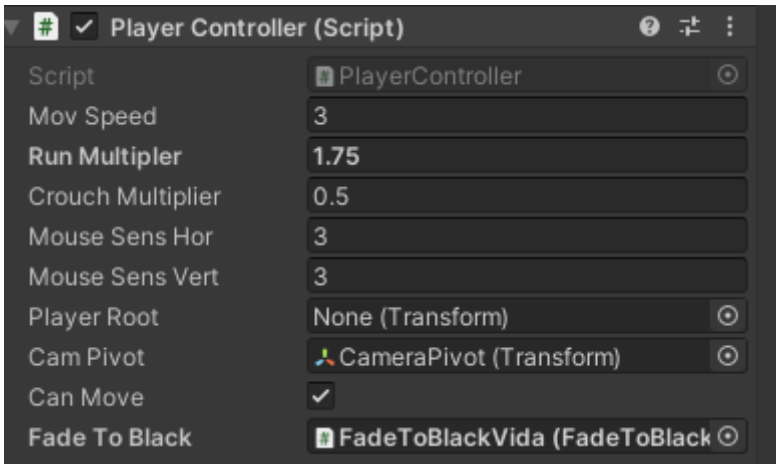


Figura 21 componente PlayerController en inspector de Unity

En otros casos, buscar el objeto con un componente dado en la jerarquía de la escena sería costoso y, cuando no es necesario, es más sencillo y eficiente gestionar la referencia de esta manera. Como se muestra en la figura 21, este es el caso con la referencia al componente 'FadeToBlack', que se encarga de hacer que la pantalla pase a negro y vuelva a la normalidad, proporcionando notificaciones a través del patrón 'Observer'. En el método Start podemos ver como PlayerController se suscribe tanto al componente ConVida como a FadeToBlack, a pesar de que FadeToBlack nunca es obtenido mediante código en Start.

A continuación, se describe la clase Inventario.cs que maneja los objetos obtenidos por el usuario:

```
private List<ListenerInventario> listenersInventario;
Dictionary<string, CuentaObjeto> objetos;
Unity Message | 0 references
private void Start()
{
    objetos = new Dictionary<string, CuentaObjeto>();
    listenersInventario = new List<ListenerInventario>();
}
1 reference
public void addObjeto(string nombre, int cantidad)
{
    if (objetos.TryGetValue(nombre, out CuentaObjeto cuenta))
    {
        cuenta.cantidad = cuenta.cantidad + cantidad;
        notificar(nombre, cuenta.cantidad);
    }
    else
    {
        CuentaObjeto cuentaNueva = new CuentaObjeto();
        cuentaNueva.cantidad = cantidad;
        cuenta.nombreObjeto = nombre;
        objetos.Add(nombre, cuentaNueva);
        notificar(nombre, cuentaNueva.cantidad);
    }
}
```

Figura 22 addObjeto de la clase Inventario

En la figura 22 encontramos parte del código de la clase Inventario. Para guardar los objetos se utiliza un *Map* de *string* y un objeto CuentaObjeto que guarda información sobre el objeto y la cantidad actual. El *Map* permite buscar objetos por la clave *string* que se corresponde con el nombre del objeto, y recuperar el valor CuentaObjeto.

Cuando añadimos un objeto se busca en objetos y, de no ser encontrado, se crea una nueva instancia de CuentaObjeto para guardarlo. El método notificar itera sobre la lista de ListenerIventario y los notifica de la adquisición del nuevo objeto.

6. Pruebas

Se han realizado pruebas de aceptación con usuarios para asegurar que el juego es entretenido y satisfactorio para los jugadores y encontrar puntos a tratar para mejorar la calidad del juego.

La primera tanda de pruebas se realizó tras el primer *sprint*, en el que se construyó la mecánica principal del juego que permite a los jugadores crear las criaturas y darles una IA. La primera tanda tenía como objetivo comprobar las dificultades que tenían los jugadores respecto a entender esta mecánica.

Tal como se esperaba, se encontró que los jugadores tenían mucha dificultad para entender la mecánica principal y como usarla, a pesar de que disfrutaron de la experiencia generalmente.

De esta tanda de pruebas se extrajeron una serie de puntos a tratar:

Primero, una serie de cambios en la interfaz del juego eran necesarios para que se entendiera mejor qué estaba ocurriendo y qué podía o no hacer en cada momento el jugador. Por ejemplo, el jugador solo puede tener una criatura en el escenario en todo momento, lo cual es indicado por un cambio de color de un elemento de la interfaz a gris. Este elemento representa una rata, pero varios de los jugadores no discernían la figura, y tuvo que cambiarse.

Otro cambio era necesario en el menú de construcción de la inteligencia artificial; este era difícil de entender, tanto en propósito como el significado de elementos específicos de la interfaz. Para resolverlo, se añadió un panel con información relevante al lado del panel de construcción de la IA, mostrado en la Figura 23. En este panel se ofrece una descripción breve de la interfaz y la mecánica principal del juego.

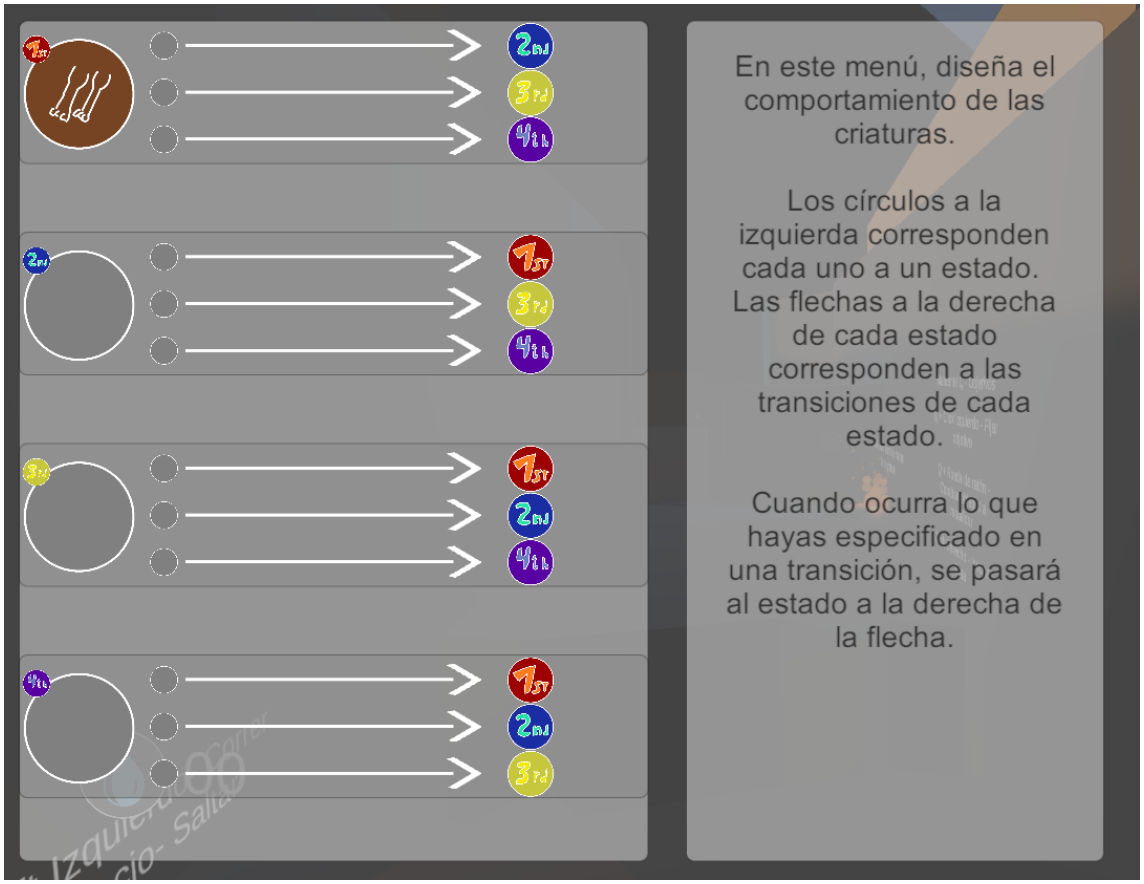


Figura 23 Menú de creación de criaturas con instrucciones

Más importantemente, las pruebas demostraron que los puzles presentados al jugador desde un principio eran demasiado complejos, y era necesario rebajar la dificultad inicial del juego y dejar que el jugador aprendiera las mecánicas principales una a una con puzles muy sencillos.

Tras realizar estos cambios y los dos últimos *sprints*, se realizó una segunda tanda de pruebas para comprobar si los cambios habían surtido efecto y cómo reaccionaban los jugadores al estado final del juego tras el desarrollo.

En la segunda tanda de pruebas, los jugadores fueron más capaces de entender la mecánica principal del juego, pero existían aún problemas de comprensión respecto al funcionamiento de algunos aspectos de esta. Sin embargo, los jugadores llegaron a entender la mecánica y alcanzar el final del juego.

7. Conclusiones

Se han desarrollado dos niveles de un videojuego en primera persona basado en la exploración y la resolución abierta de puzles mediante el diseño por parte del usuario del comportamiento de sus aliados. El juego está dotado además de sistemas de jugabilidad emergente que interactúan entre sí con reglas simples y permiten al jugador encontrar soluciones distintas a los puzles.

Se han realizado dos tandas de pruebas para validar la recepción del juego por los usuarios y se ha obtenido información valiosa de estas. El resultado final de las pruebas ha determinado que aún es necesario mejorar la comunicación del funcionamiento de las mecánicas del juego al jugador, pero la recepción ha sido mayoritariamente positiva.

7.1 Relación del trabajo desarrollado con los estudios cursados

El desarrollo de este juego no hubiera sido posible sin muchos de los conocimientos adquiridos en las diferentes asignaturas de ingeniería informática.

Desde un punto de vista de diseño, el funcionamiento del sistema de estados por mediante del patrón Estado no hubiera sido posible diseñarlo sin la ayuda de la asignatura de Diseño Del Software, donde se aprendió gran cantidad de patrones, algunos de los cuales han sido aplicados en este proyecto y que han simplificado el desarrollo.

Por supuesto, el desarrollo no hubiera sido posible sin los conocimientos generales de programación adquiridos en asignaturas como ‘Introducción a la Programación’, ‘Programación’, y ‘Lenguajes, Tecnologías y Paradigmas de la programación’.

En la asignatura de ‘Interfaces Persona-Computador’ se trabajó con C# y Visual Studio, además de la herramienta de control de versiones Git, las cuales se han utilizado en este proyecto. Además, IPC proporcionó conocimientos sobre el diseño de interfaces usables que se han aplicado en la creación de la interfaz del juego.

De ‘Estructuras de Datos y Algoritmos’, se ha utilizado un HashMap, enseñado en la asignatura.

El sistema de inteligencia artificial diseñado por el jugador se trata de una máquina de estados de las cuales se ha aprendido en la asignatura de ‘Teoría de autómatas y lenguajes formales’.

La asignatura de ‘Introducción a la Programación de Videojuegos’ ha ofrecido muchos conocimientos sobre todas las partes del proceso de creación de un videojuego, desde el diseño hasta la implementación. Además, la experiencia práctica de la creación de un juego en la asignatura junto a los consejos y conocimientos sobre videojuegos y también específicamente sobre Unity han sido indispensables para la realización de este proyecto.

También se han aplicado algunos de los conocimientos de carácter general obtenidos en la asignatura de ‘Animación y Diseño de Videojuegos’, además de haberse utilizado una herramienta recomendada por el profesor de la asignatura.

Tanto ‘Proceso de Software’ como ‘Proyecto de Ingeniería de Software’ también fueron fundamentales al proporcionar experiencias de desarrollo, específicamente en un entorno ágil usando herramientas similares a Jira.

Como se puede apreciar, este proyecto no se podría haber realizado sin todo lo aprendido a lo largo de la carrera.

8. Trabajos Futuros

Primero se ha de decir que no se han llegado a trabajar todos los requisitos, particularmente los de menú de opciones y gestión de guardado y partidas guardadas. Sería especialmente interesante desarrollar esta parte del juego que ayudaría a que se comportara como se espera que se comporte un videojuego real. También sería muy interesante centrarse en transmitir las mecánicas del juego de forma más entendible al jugador y realizar más tandas de testeo hasta conseguir que el jugador entienda las mecánicas con facilidad.

Se han desarrollado tan solo dos niveles del juego, y por ello las mecánicas aún están poco exploradas. Existen muchas más posibilidades de presentar al jugador nuevas situaciones y problemas solo con los sistemas desarrollados actualmente.

También existe la posibilidad de ampliar los sistemas y la inteligencia artificial que puede diseñar el jugador, tanto en número de estados como funcionalidad. Esto daría pie a aún más posibilidades de puzles en el juego.

Aunque se encuentra en un estado jugable, aún existen muchos puntos de mejora, tanto visualmente, como en términos de narrativa, opciones proporcionadas al usuario, y usabilidad general. Futuramente podría ser interesante pulir estas partes del juego.

Otra posibilidad sería utilizar el sistema de creación de una máquina de estados para otros tipos de construcciones del jugador. En lugar de diseñar una inteligencia artificial, podría diseñar la secuencia de efectos producida por un arma al disparar, o el comportamiento de un elemento del juego.

Por último, el sistema podría ser mejorado y utilizado en otros contextos. Por ejemplo, podría utilizarse para desarrollar una forma de diseñar la inteligencia artificial de un

personaje de forma intuitiva y visual para parte del equipo de desarrollo de un juego que no está familiarizado con la programación.

9. Referencias

9.1 Bibliografía

[1] A. Rollings y E. Adams, Fundamentals of Game Design. 2006.

[2] K. Schwaber, Agile project management with Scrum. 2004.

[3] M. Manole y M.-Șerban Avramescu, «A Comparative Analysis of Agile Project Management Tools», Economy Informatics, vol. 17, n.º 1, 2017, [En línea]. Disponible en: http://www.economyinformatics.ase.ro/content/EN17/03%20-%20manole,%20avramescu.pdf

[4] P. Deemer, G. Benefield, C. Larman, y B. Vodde, «A Lightweight Guide to the Theory and Practice of Scrum», The Scrum Primer, 2012, [En línea]. Disponible en: https://scrumprimer.org/scrumprimer20_small.pdf

[5] J. Vahlo, «An Enactive Account of the Autonomy of Videogame Gameplay. Game Studies», vol. 17, n.º 1, 2017, [En línea]. Disponible en: http://gamestudies.org/1701/articles/vahlo

[6] S. Breslin, «The History and Theory of Sandbox Gameplay», 2009, [En línea]. Disponible en: https://www.gamedeveloper.com/design/the-history-and-theory-of-sandbox-gameplay

[7] J. Bycer, «Examining Emergent Gameplay», 2009, [En línea]. Disponible en: https://www.gamedeveloper.com/design/examining-emergent-gameplay

[8] K. E. Wiegers, Software requirements practical techniques for gathering and managing requirements throughout the product development cycle. 1997.

[9] I. Jacobson, I. Spence, y K. Bittner, USE-CASE 2.0 The Guide to Succeeding with Use Cases. 2011. [En línea]. Disponible en: https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf

[10] S. Pope y G. E. Krasner, A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk80 System. 1988.

[11] E. Gamma, R. Helm, y J. Vlissides, Design patterns: elements of reusable object-oriented software. 1994.