Escola Tècnica
Superior d'Enginyeria
Informàtica

Escuela Técnica Superior de Ingeniería Informática
Universitat Politècnica de València

The Traveler

Videojuego de plataformas empleando Unity3D y una API de IA en C#

Proyecto Final de Carrera

Ingeniería Informática

Autor: Guillermo Monfort Pardo

Director: Ramón Pascual Mollá Vayá

2015

Resumen

Este proyecto consiste en la realización de un videojuego de plataformas y exploración 2D con el motor de desarrollo Unity, con un énfasis en la gestión y creación de inteligencias artificiales con máquinas de estados. Se pretende crear un juego para plataformas de escritorio pulido, complejo y variado. Busca también aportar una visión general de la creación de un videojuego y de las distintas etapas de su desarrollo.

Se usará la API de gestión y creación de máquinas de estados finitos de José Alapont Luján en distintos elementos del juego. Se pondrá a prueba la herramienta y se comparará con otras alternativas.

Abstract

The goal of this project is to create a 2D platforming and exploration game using the Unity engine. There is an emphasis in the creation and management of artificial intelligences using finite state machines. The objective is a polished, complex and varied game for desktop platforms. It is meant to provide an overview of the creation of a videogame and all the stages of its development.

It will make use of José Alapont Luján's API for creating and managing finite state machines for different elements of the game. It will be put to the test and compared against other alternatives.

Palabras clave: videojuego, Unity, C#, FSM

Keywords: videogame, Unity, C#, FSM

Tabla de contenidos

1. Introducción ... 6
- 1.1. Motivación ... 6
- 1.2. Objetivos ... 7
- 1.3. Estructura de la obra ... 7

2. Estado del arte ... 9
- 2.1 Motores de juego ... 9
- 2.2 Selección de herramientas ... 10
- 2.3 Creación de inteligencia artificial ... 12
- 2.4 Videojuegos similares ... 13
- 2.5 Introducción a Unity ... 15

3. Diseño ... 18
- 3.1 Concepto ... 18
- 3.2 El jugador ... 18
- 3.3 Animales ... 20
- 3.4 Menú de pausa ... 24
- 3.5 Managers ... 25
- 3.6 *API* de gestión de máquinas de estado ... 26
- 3.6.1 Creación del documento *xml* con la máquina ... 26
- 3.6.2 Preparación de la clase *Tags* ... 28
- 3.6.3 Carga de la máquina con FSM_Parser ... 28
- 3.6.4 Uso de las máquinas de estado ... 29
- 3.7 Sistema propio de máquinas de estado ... 30
- 3.7.1 Estado ... 31
- 3.7.2 Máquina de estados ... 31

4. Planificación ... 32

5. Implementación ... 36
- 5.1 Colisiones ... 36
- 5.2 Protagonista ... 37
- 5.3 Cámara ... 39
- 5.4 Agua ... 40
- 5.5 Vegetación ... 43
- 5.6 Ojo del protagonista ... 43
- 5.7 Onda de interacción ... 44
- 5.8 Coger y lanzar objetos ... 44
- 5.9 Ciclo de día y noche ... 45
- 5.10 Paralaje ... 47
- 5.11 Puertas y transiciones entre niveles ... 48
- 5.12 Estrellas ... 48
- 5.13 Sistema de mensajes ... 48
- 5.13.1 Canvas ... 49

5.13.2 Message 49
5.14 Objetos rompibles 50
5.15 Lógica de los escenarios y de los puzzles 50
5.16 Expresiones del protagonista 51
5.17 Nubes de la isla 52
5.18 Animales 52
5.18.1 Pájaro 52
5.18.2 Pez 53
5.18.3 Medusa 53
5.18.4 Tortuga y Caracol 54
5.19 Música y efectos de sonido 54
5.20 Iluminación de escenarios 56
5.21 Menú de pausa 56
5.22 Animación de inicio 57
5.23 Detalles finales 57
6. Conclusiones 59
6.1 Relación con los estudios cursados 59
6.2 Conclusiones IA 60
6.3 Planificación 61
7. Trabajos futuros 63
8. Agradecimientos 64
9. Bibliografía 65
10. Anexo I: Uso de *2DToolkit* y *Ferr2D* 66
11. Anexo II: Documento de diseño 70

1. Introducción

1.1. Motivación

El deseo de trabajar profesionalmente en el diseño y creación de videojuegos en el futuro ha sido la principal motivación para realizar este proyecto. La industria de los videojuegos crece año tras año, constituyendo la principal industria de ocio en nuestro país y en el mundo entero [1], superando a las industrias del cine y la música.

Esto se debe en gran parte a la reducción de las barreras de entrada para los creadores, gracias a la introducción de herramientas como *Unity* o *Unreal Engine,* que posibilitan la creación de videojuegos de muy alta calidad con muy pocos recursos.

La contrapartida de esta democratización en las herramientas de creación es que la competencia ha aumentado y el número de juegos lanzados está aumentando constantemente. En los primeros 5 meses de 2014 se lanzaron más juegos en *Steam*[1] que en todo el año anterior. Debido a esto y dado que todo el mundo puede crear juegos pequeños de manera independiente usando estas herramientas, es requisito indispensable para conseguir un trabajo en el sector contar con algún proyecto personal a la espalda.

Así, este proyecto pretende servir para reforzar conocimientos y obtener experiencia con el motor *Unity* y el lenguaje *C#,* creando finalmente un videojuego pulido, jugablemente complejo y variado.

En un mercado saturado de juegos móviles para un público *casual* se ha optado por trabajar en un juego para plataformas de escritorio, por las posibilidades visuales y jugables que estas plataformas brindan, libres de las limitaciones inherentes a los dispositivos móviles.

La realización de un videojuego, además, precisa de conocimientos en muchos de los campos que un ingeniero informático debe dominar, como son la ingeniería y arquitectura del software, teoría de autómatas, inteligencia artificial o redes, además de otros más generales como son el diseño de juego, niveles, sonido y música, matemáticas y física. Por todo esto se considera un trabajo muy completo y una forma adecuada de culminar la carrera y probar y demostrar los conocimientos adquiridos a lo largo de la misma.

[1] Steam es la plataforma de distribución digital de juegos más grande en plataformas de escritorio

1.2. Objetivos

El objetivo principal del proyecto es el de diseñar e implementar un prototipo de un juego de aventuras 2D para plataformas de escritorio. El prototipo contará con la jugabilidad completa del personaje principal y con todos los sistemas principales del juego. Debe ofrecer una experiencia jugable completa y quedar preparado para su futura ampliación y compleción.

Se desea ahondar en las características del motor y del resto de herramientas elegidas, conocer el flujo de trabajo y las posibilidades de cada una de ellas. También se busca ganar experiencia y aprender el proceso de creación de un videojuego en general, prestando especial atención al diseño del juego y de los niveles que lo compondrán. Este conocimiento es independiente de las herramientas empleadas e imprescindible para crear un videojuego de calidad.

El uso y evaluación de la *API* de gestión de Inteligencia Artificial de José Alapont Luján es otro de los objetivos de este trabajo. Se busca analizar el uso de las máquinas de estados como técnica para la creación de inteligencias artificiales en los videojuegos. Así, se hará uso de la *API* para crear personajes no jugables controlados por distintos tipos de máquinas de estados y se mostrarán ejemplos de uso de la herramienta.

Además de emplear esta *API* se creará un sistema básico propio de máquinas de estado con el que se implementarán algunos objetos específicos para así poder realizar comparaciones exhaustivas entre esta dos herramientas de creación de inteligencias artificiales.

También se realizarán comparaciones más generales con otras herramientas ya existentes y distintos paradigmas de gestión de *IA*.

1.3. Estructura de la obra

En este apartado se pretende aclarar la estructura del documento, enumerar y describir brevemente los apartados más importantes. La obra arranca con una breve introducción en la que se explican los objetivos que se pretenden conseguir con el proyecto y las motivaciones que han propiciado su desarrollo. A continuación se realiza un análisis del estado del arte, se seleccionan las herramientas que se usarán en el desarrollo y se realiza una pequeña introducción a Unity. Seguidamente se aborda el diseño del videojuego y del sistema software que lo implementará de manera resumida y visual. La planificación constituye el siguiente apartado, en él se describe la distribución de tiempo que se ha establecido previamente a la implementación del proyecto. Tras ello se encuentra el apartado de implementación, en el que se explican todas las partes del desarrollo del proyecto, de manera general pero con suficiente detalle para poder

comprender el trabajo realizado. El documento sigue con las conclusiones, donde se analiza el resultado al final del desarrollo y la metodología de trabajo a lo largo del proyecto. Finalmente se encuentran los agradecimientos, la bibliografía y los anexos.

2. Estado del arte

2.1 Motores de juego

En la actualidad existe una gran cantidad de motores y *frameworks* de creación de videojuegos, cada uno con sus particularidades y características específicas. Algunos de los criterios que pueden servir para valorar un motor o *framework* concreto son:

- Orientación a 2D o 3D
- Lenguaje de programación que emplea
- Exportación y desarrollo multiplataforma (*cross-platform)*
- Código abierto
- Documentación de calidad y código de ejemplo
- Editor visual
- Precio y licencias

A continuación se presentan algunas de las herramientas más comunes y se analizan respecto a los criterios presentados:

GameMaker Studio, de *YoYo Games* es un motor de desarrollo con editor visual orientado tanto para novatos como para programadores experimentados. Permite crear la lógica del juego sin necesidad de conocimientos de programación y cuenta también con un lenguaje propio de *scripting.* Está principalmente orientado a dos dimensiones y es capaz de exportar a una gran cantidad de plataformas distintas, móviles o de escritorio, pero el editor funciona únicamente en Windows. La versión gratuita tiene funcionalidad limitada y sólo permite la exportación a Windows.

Godot es un motor gratuito y de código abierto desarrollado por *Okam Studio.* Cuenta con un entorno visual muy completo y permite la creación de juegos 2D y 3D, aunque está más orientado a las tres dimensiones. Hace uso de un lenguaje de *scripting* propio basado en *Python.* El editor funciona en Windows, Mac y Linux y permite exportar a las plataformas móviles y de escritorio más comunes. Es un motor muy joven y la documentación es limitada.

Unreal Engine es el motor profesional de *Epic Games.* Es gratuito y de código abierto, pero hay que pagar un *royalty* de un 5% al comercializar un producto realizado con el motor. Usa *C++* y cuenta también con un editor visual de *scripting*

(el sistema de *blueprints*). También posee un editor visual de *shaders.* Es usado por un gran número de estudios relevantes del sector y se ha utilizado para crear muchos juegos multiplataforma comercialmente exitosos y visualmente punteros.

LibGDX, de *Badlogic Games* es un *framework* Java muy eficiente orientado al desarrollo de juegos 2D, aunque también permite la creación de juegos 3D. Es gratuito y multiplataforma. Cuenta con una amplia comunidad y una documentación muy detallada y completa.

MonoGame es un *framework* C# de código abierto basado en la obsoleta *XNA.* Es gratuito, orientado al 2D y soporta todas las plataformas comunes. La documentación es muy escasa, pero cuenta con una gran comunidad de desarrolladores.

Unity es, probablemente, el motor de juegos más ubicuo en la actualidad. Es adecuado tanto para 2D como para 3D. La versión gratuita tiene muy pocas restricciones y puede usarse con juegos comerciales siempre y cuando no se superen unos ingresos de 100.000 dólares al año. El editor es multiplataforma y exporta a muchas plataformas distintas. Soporta C# y Javascript, tiene una documentación muy amplia y detallada y cuenta con la comunidad más amplia de entre todos los motores nombrados.

OpenFL es un *port* a Haxe de la *API* Flash. Es gratuito, multiplataforma y exporta a prácticamente todas las plataformas existentes. La documentación es deficiente pero, al estar basada en Flash, es posible hacer uso de la documentación existente de Adobe. Cuenta con una comunidad de desarrolladores bastante grande.

2.2 Selección de herramientas

Motor de juego

De entre las opciones presentadas anteriormente se decidió finalmente utilizar Unity debido a su facilidad de uso, la experiencia previa con la herramienta y el soporte que proporcionan la documentación y la comunidad de que dispone. Además era necesario, como mínimo, poder exportar el juego a las distintas plataformas de escritorio, algo trivial con Unity.

Una vez elegido Unity se decidió hacer uso exclusivo del lenguaje orientado a objetos C# en lugar de Javascript, dada la experiencia existente con el mismo y otros lenguajes similares como C++ o Java. Además, la documentación de C# es más amplia y la mayoría de *plugins* y proyectos de Unity están escritos usando este lenguaje.

Plugins y librerías

Una vez decidido el motor de juego se decidió buscar y hacer uso de varias librerías para facilitar algunas de las tareas de implementación que implicaba el proyecto o bien para rellenar huecos o mejorar algunas características de Unity. Dada la ubicuidad del motor escogido existe un gran número de *plugins* de muy alta calidad, tanto gratuitos como de pago. A continuación se comentan en detalle los escogidos y se comparan con las alternativas existentes:

2D Toolkit, de *Unikron Software,* es una herramienta que amplía, facilita y mejora el soporte 2D de Unity. Aporta muchas opciones de creación de colecciones de *sprites*[2]*,* imprescindibles en juegos 2D, permite crear *colliders* ajustados a las imágenes de manera automática, animaciones de *sprites,* fuentes de mapa de bits y mucho más. En este momento cuesta 75 dólares. Dada la experiencia previa con la herramienta y poseyendo una licencia de la misma, se decidió hacer uso de *2D Toolkit* desde el incio del proyecto. Existen alternativas como *Orthello 2D Pro*, o *Sprite Factory*, cada una con sus puntos fuertes y débiles y diferentes flujos de trabajo, pero ninguna cuenta con características específicas que no se puedan replicar con *2D Toolkit.*

Ferr2D Terrain Tool, de *Simbryo.* Debido a las especificaciones del proyecto era fundamental crear un mapeado orgánico, con curvas, huyendo de los mapas de tiles cuadriculados. *Ferr2D* permite crear terrenos de casi cualquier tipo a partir de una selección muy reducida de texturas iniciales. La herramienta se encarga de colocar, estirar y girar las texturas para que se adapten a las curvaturas, todo ello visualmente desde el propio editor de Unity. Además genera *colliders* optimizados y ajustados para los terrenos.

Comparándola con las alternativas existentes *2D Terrain Editor* de *Voodoo* y *2D Terrain Builder* de *Juoq*, *Ferr2D* resulta ser la herramienta más completa, ofreciendo un gran número de opciones de personalización para los terrenos generados. Unity planea ofrecer una funcionalidad parecida a la de estas herramientas (aunque previsiblemente más limitada) de forma nativa en la versión 5.4 [3] del motor.

DOTween es un motor de animación (por interpolación) muy eficiente y rápido creado por *Demigiant.* Un motor de este tipo permite realizar animaciones desde código de manera muy sencilla con una simple instrucción. Cuenta con una sintaxis muy clara y gran cantidad de atajos para tipos nativos y no nativos. *DOTween* mejora en eficiencia a motores similares como *GoKit, iTween, LeanTween o HOTween* [4].

[2] Mapa de bits

***Haste*,** de *Barking Mouse Studio,* un motor de búsqueda para Unity, que permite moverse por la jerarquía de la escena o del proyecto y seleccionar objetos con un atajo de teclado y una búsqueda muy rápida. Cuando un proyecto empieza a crecer la jerarquía de carpetas comienza a ser abrumadora y una escena puede contener centenares de objetos. *Haste* elimina en gran parte los problemas que esto ocasiona, aumentando la productividad del programador o diseñador. Se ha hecho uso de la versión gratuita de la herramienta, la de pago cuenta con atajos y comandos específicos para realizar acciones en el editor de Unity. Existe una alternativa llamada *ReUniter,* que tiene unas características similares pero no ofrece una versión gratuita.

UtilityKit es una pequeña colección de scripts creada por *prime31*. La manera de trabajar nativa de Unity obliga al programador a usar excesivamente *strings* en el código para hacer uso de funciones relacionadas con *Layers, Tags, SortingLayers, Resources* o *Escenas.* Esto no se considera una buena práctica porque los errores relacionados con esos *strings* sólo se pueden reconocer en ejecución, nunca en tiempo de compilación y terminan siendo una fuente de *bugs* muy común conforme la base de código aumenta. *UtilityKit* cuenta con un *script* llamado *ConstantsGeneratorKit* que genera una serie de clases estáticas con variables que contienen esos *strings* y las actualiza en cuanto se produce un cambio en alguna de ellos.

ObjectPool es un sistema de reutilización de objetos creado por *UnityPatterns.* Las operaciones de instanciación y destrucción de objetos son muy costosas, esta herramienta permite evitarlas reciclando y reutilizando objetos existentes en lugar de crear nuevos. Existen alternativas como *Pooling Toolkit* o *Easy Pool Manager* que ofrecen mayor facilidad de uso e integración con el editor, pero son de pago. Para los propósitos de este proyecto *ObjectPool* resulta más que suficiente.

2.3 Creación de inteligencia artificial

La inteligencia artificial en los videojuegos se emplea principalmente para generar comportamientos inteligentes en personajes no jugables. En la actualidad se usan dos técnicas predominantemente para modelar inteligencias artificiales, las máquinas de estados y los árboles de comportamiento. Esta última ha ido ganando cada vez más importancia en los últimos años, debido a la flexibilidad y expresividad que ofrecen los árboles de comportamiento [2], además de ser fácilmente comprensibles de un vistazo.

Aparte de las herramientas mencionadas en el apartado anterior se decidió emplear la *API* de inteligencia artificial de José Alapont. Esta *API* facilita la creación de comportamientos e inteligencias por medio de máquinas de estados finitos y soporta distintos tipos de máquinas (determinista, probabilista, inercial,

concurrente, basada en pilas). Se eligió esta herramienta por los siguientes motivos: En primer lugar, se buscaba modelar las inteligencias usando máquinas de estados debido a los conocimientos adquiridos durante la carrera sobre las mismas, mientras que no se contaba con experiencia con árboles de comportamiento. La herramienta es especialmente relevante por surgir como resultado de una tesina fin del máster IARFID, por lo que se deseaba probar su eficiencia y adecuación en un videojuego, campo para el cual fue concebida en primer lugar. Además, como consecuencia de lo anterior, se podía hacer uso de la herramienta sin coste alguno, mientras que la mayoría de alternativas existentes requieren de la adquisición de una licencia.

Seguidamente se comentan las alternativas disponibles más relevantes orientadas al motor Unity:

Behavior Designer, de *Opsive,* permite crear árboles de comportamiento por medio de un editor visual muy completo, así cómo depurarlos y observar su comportamiento de manera visual durante la ejecución. Es muy eficiente y ofrece integración con muchos otros plugins relevantes de Unity. Cuesta 75 dólares.

Behaviour Machine, de *Anderson Cardoso.* Esta herramienta está pensada para facilitar la creación de máquinas de estado y árboles de comportamiento de manera visual. Cuenta con una *API* extensa y muy bien documentada. Está disponible en tres versiones distintas, una gratuita, otra por 50 dólares y otra por 250, cada una con su conjunto de características y limitaciones.

NodeCanvas**,** de *Gavalakis Vaggelis,* es una herramienta de creación de comportamientos que permite añadir árboles de comportamiento, máquinas de estado jerárquicas y árboles de diálogo a Unity. Está pensada para que sea lo más intuitiva y fácil de usar tanto para programadores como diseñadores, pero sin sacrificar potencia ni expresividad. Tiene un precio de 75 dólares.

2.4 Videojuegos similares

En los videojuegos, como en cualquier otro campo artístico, una obra específica no suele surgir de manera aislada, sino que es el resultado, directa o indirectamente, de la unión de muchas obras o conceptos anteriores de la misma u otra disciplina, de acuerdo con el bagaje de los creadores. Este apartado busca analizar una serie de juegos existentes que han tratado con conceptos o se han enfrentado a problemas similares a los de ***The Traveler*** y explicar cómo se han resuelto tales problemas en cada caso.

Super Mario 64 fue el primer juego de plataformas en 3D de la saga Mario Bros. Los niveles tradicionalmente lineales de los juegos de Mario en 2D ahora se convertían en vastos mundos, colocando un énfasis en la exploración y la

rejugabilidad. Para distribuir estos mundos el juego cuenta con un gran escenario central dividido en varias partes. En este escenario central hay puertas que llevan al resto de niveles jugables, en los que transcurre toda la acción del juego. Así, el escenario central es solamente un nexo que permite acceso a los niveles, como los anteriores mapas del mundo de los juegos 2D, pero otorgando mucha más participación al usuario. Este nexo no debe formar un espacio físico lógico con el resto de niveles. Estos son completamente independientes, por lo que hay completa libertad creativa en lo que respecta al diseño de cada nivel.

Además Mario contaba en el juego con un conjunto de movimientos muy amplio, lo cual aportaba variedad y diversión al control y al movimiento a lo largo de los escenarios.

Animal Crossing es un simulador de vida muy relevante por su sistema de juego abierto y por la gran cantidad de actividades que puede realizar el jugador. El juego transcurre en un pequeño pueblo generado aleatoriamente repleto de vida. Se desarrolla en tiempo real, por lo que cuenta con un ciclo de día y noche y con el paso de estaciones. Estos ciclos transcurren sin pausas y los habitantes se adaptan también al paso del tiempo en tiempo real. Así, durante el día se podrá observar a los habitantes del pueblo realizando varias actividades en distintos puntos del mismo y, al llegar la noche, podrán caminar de vuelta a sus casas para dormir. El juego presenta una variedad muy amplia de posibilidades al jugador, que podrá realizarlas a su antojo, en el orden que desee según la experiencia que busque.

Fez es un juego de plataformas 2D con un gran énfasis en la exploración y los puzles. Cuenta con un gran mundo dividido en muchos niveles pequeños unidos de forma coherente y ofreciendo una continuidad. Tras los primeros minutos en los que se enseña al jugador los controles y particularidades del mundo del juego el jugador queda libre para recorrer los escenarios y resolver gran cantidad de puzles ambientales. Estos puzles se alejan de aquellos tradicionales que se centran en el uso de algunas mecánicas de juego puntuales y se apoyan más en la exploración y los escenarios, creando una experiencia muy orgánica y abierta.

The Traveler cuenta con varios elementos de los juegos anteriores e intenta combinarlos de manera satisfactoria en un juego de plataformas bidimensional. Así, el jugador tiene a su disposición un amplio conjunto de movimientos que le permiten desplazarse por un vasto mundo. Este mundo está dividido en escenarios y pretende conformar un espacio físico lógico en líneas generales, aunque se toman licencias en algunos casos, como puede ser el tamaño de espacios interiores respecto a su representación en el exterior.

El mundo ofrece al jugador una gran cantidad de actividades distintas y los espacios son rejugables. Existe paso del tiempo, aunque no equivalente al real, y el juego cuenta con un ciclo de día y noche. Sin embargo, este ciclo no será siempre sin transiciones, en el escenario central de la ciudad se realizará un corte al llegar el día o la noche, para ocultar y evitar las transiciones de estado de los habitantes en esos momentos específicos.

Habrá puzles ambientales centrados en la exploración y la observación del entorno y, en general, se limitará el uso de tutoriales para enseñar al jugador, intentando realizar esa función por medio del lenguaje visual y el *feedback*[3] del mundo.

En definitiva, la unión de estos conceptos y características existentes en juegos anteriores acaba creando un videojuego único del que no existe equivalente en el mercado actual.

2.5 Introducción a Unity

En este apartado se explicará cómo funciona Unity a un nivel básico y se comentarán los elementos básicos que ofrece a partir de los cuales se crearán las funcionalidades deseadas en videojuego. No pretende ser un tutorial detallado, simplemente se introduce la tecnología para facilitar la comprensión de la memoria.

En la base de Unity se encuentran los *GameObjects,* la clase base para todas las entidades de una *escena* de Unity. Así, los personajes, la cámara, cualquier objeto del escenario y los elementos de los menús son *GameObjects*[4]*.*

Una *escena* es simplemente una colección de objetos. Típicamente una *escena* representará un escenario independiente del juego, conteniendo todos los elementos que lo forman, incluyendo los menús.

En el corazón de todos los motores de videojuegos se encuentra el llamado *game loop,* un patrón de diseño fundamental para los videojuegos. En esencia es un bucle infinito que en cada iteración recibe una entrada, actualiza los elementos del juego de acuerdo a esa entrada y finalmente muestra el resultado en pantalla. Unity no es ninguna excepción pues cuenta también con un *game loop*, pero éste está oculto al programador. Sin embargo Unity cuenta con la clase *MonoBehaviour*, la clase básica de la que heredarán todos los *scripts* que dependan del ciclo de ejecución del motor.

[3] Retroalimentación, respuesta o reacción. Información que el jugador recibe del juego.

[4] Existen excepciones, como en el caso de los objetos que descienden de *ScriptableObject.*

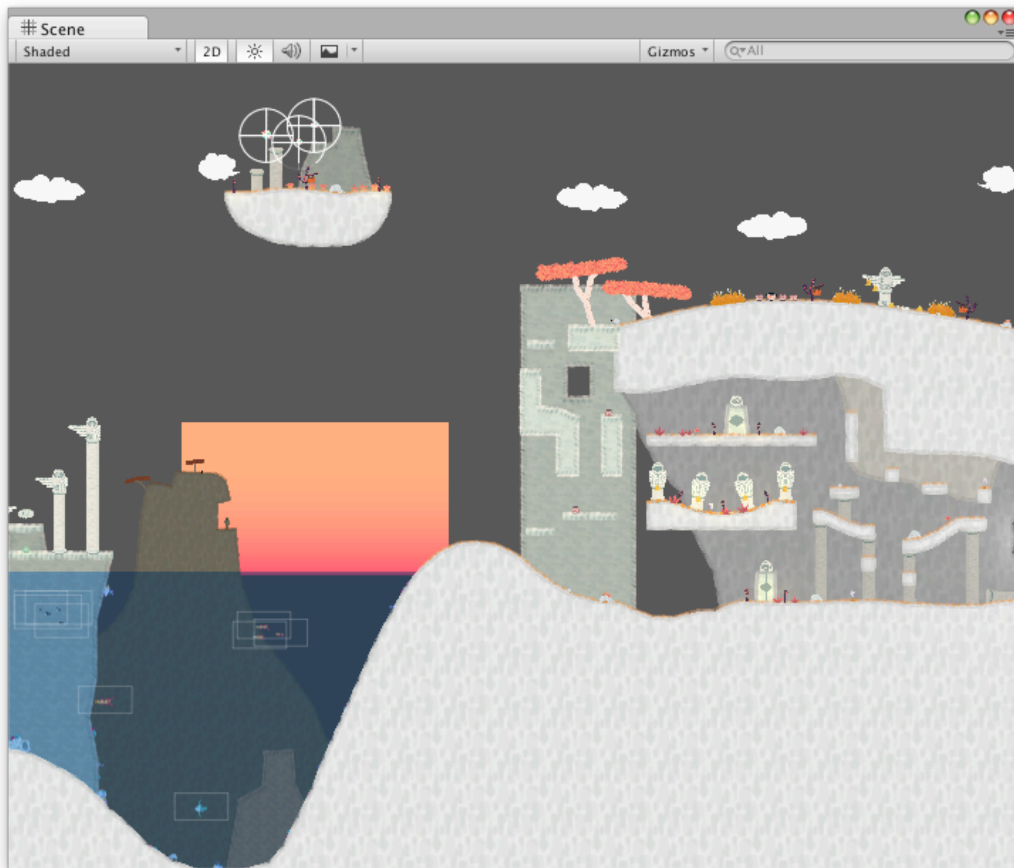


Ilustración 1. Una escena en el editor de Unity

De esta manera, todo *MonoBehaviour* cuenta con una serie de funciones que se llamarán automáticamente según el ciclo de vida propio del objeto y el del *game loop* del motor. La función *Awake()* se ejecuta en la instanciación del objeto, y *Start()* después de ésta. *Update()* se ejecutará una vez por cada ciclo del *game loop* mientras el *script* esté activo. Éstas son algunas de las funciones más importantes propias de los *MonoBehaviour*, el resto están explicadas en profundidad en la documentación de Unity [5].

El siguiente concepto fundamental para comprender el funcionamiento de Unity es el *Component.* Un *Component* es una pieza funcional de un *GameObject*. El *GameObject* no tiene funcionalidad propia, es un contenedor vacío por defecto al que se irán asignando *Components* con funciones específicas.

Al crear un *GameObject* se puede observar que contiene un *Component* llamado *Transform*, que dicta dónde está ubicado el objeto, así como su rotación y escala. Sin este componente el *GameObject* no estaría posicionado en el mundo y por lo tanto es imprescindible. Otros componentes que proporciona Unity son el *Collider*, que sirve para crear una zona de colisión en el objeto, el *Renderer,* que permite al

objeto tener una representación visual con unas determinadas características, o el *AudioSource,* que posibilita la emisión de sonidos desde el objeto.

Se pueden añadir tantos *Components* a un objeto como se desee, tanto nativos de Unity como propios diseñados desde cero. Desde el editor de Unity se puede ver los componentes asociados a un *GameObject* y obtener y editar información de cada uno de ellos haciendo uso del *Inspector.*

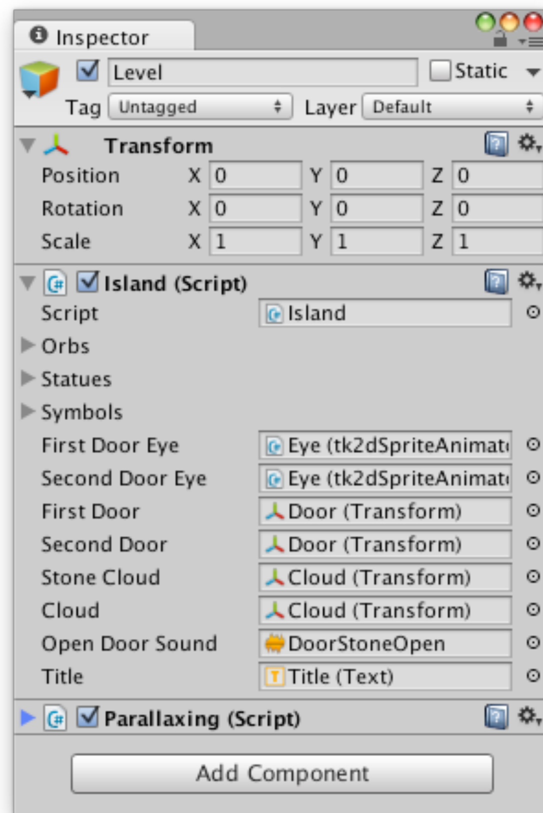


Ilustración 2. Inspeccionando un *GameObject*

Un ejemplo típico de las ventajas de un sistema por componentes sería, a la hora de crear un personaje jugable, separar la entrada del control y la lógica del movimiento en componentes distintos. De esa manera, bastaría con desactivar el *Component* que gestiona la entrada para impedir el movimiento por parte del jugador, o activar otro componente distinto de manera que el personaje pase a ser controlado por una inteligencia artificial.

Más allá de la funcionalidad básica, Unity cuenta con muchas características y editores visuales para, por ejemplo, crear animaciones complejas, importar y organizar *sprites* en *atlas*, crear máquinas de estado o gestionar la navegación automática de los personajes por el escenario.

3. Diseño

La fase de diseño es fundamental en el desarrollo de software. Sin diseño existe un alto riesgo de construir un sistema inestable, que falle al introducir cambios y sea difícil de probar y evaluar. El diseño es la base de todas las etapas posteriores del desarrollo, el tiempo invertido en este momento servirá para facilitar el trabajo futuro.

En la creación de un videojuego esta etapa contiene también el propio diseño del juego. Es aquí donde se define cómo va a ser el videojuego en detalle, así como todas las piezas que lo compondrán. De esta etapa se obtiene un documento de diseño de juego (*GDD, Game Design Document)*[5] que detalla el concepto, la historia, el arte, sonido y música, entre otros, del videojuego que se quiere construir.

El diseño de software permitirá, a partir de este *GDD,* traducir los requisitos especificados en un producto final de calidad.

3.1 Concepto

The Traveler es un juego de plataformas 2D de aventuras con escenarios abiertos y rejugables. No existe un menú principal ni de selección de pantallas, sino un amplio mundo con todos los niveles conectados físicamente entre sí.

El jugador controla a una criatura mágica que puede realizar diversas acciones y combinarlas para crear un *gameplay* complejo y variado. Esta jugabilidad permite al jugador desplazarse por el mundo rápida y eficazmente, motivando un deseo de exploración y descubrimiento. Además, es posible interactuar con la mayoría de elementos del mundo.

No es posible morir en el juego, no hay puntos de salud ni enemigos, no existe un riesgo real para el jugador en ningún momento, se busca que la experiencia sea relajada y fácilmente disfrutable.

3.2 El jugador

El protagonista dispone fundamentalmente de tres opciones básicas de movimiento, que se combinan entre sí y de acuerdo con el contexto y la situación para dar lugar a una gran variedad de acciones finales. Estos son los movimientos básicos:

- **Movimiento:** El protagonista se puede desplazar lateralmente por los escenario.

[5] Se puede consultar en el Anexo

- **Salto:** Es posible saltar verticalmente. El salto permite al jugador alcanzar lugares altos.
- **Transformación:** El personaje se puede transformar mágicamente en una bola. Así, puede rodar para moverse más rápido, rebotar en las paredes o flotar e impulsarse haciendo uso del agua.

Es posible combinar los movimientos para realizar acciones más complejas, por ejemplo se puede saltar varias veces seguidas para realizar el salto doble y triple, que permite al jugador llegar mucho más alto, o cambiar de dirección y saltar rápidamente para realizar un salto vertical muy veloz.

A continuación se muestra la máquina de estados del personaje principal. Se ha simplificado para facilitar su entendimiento y representación, la máquina final consta de 13 estados y más transiciones entre ellos.

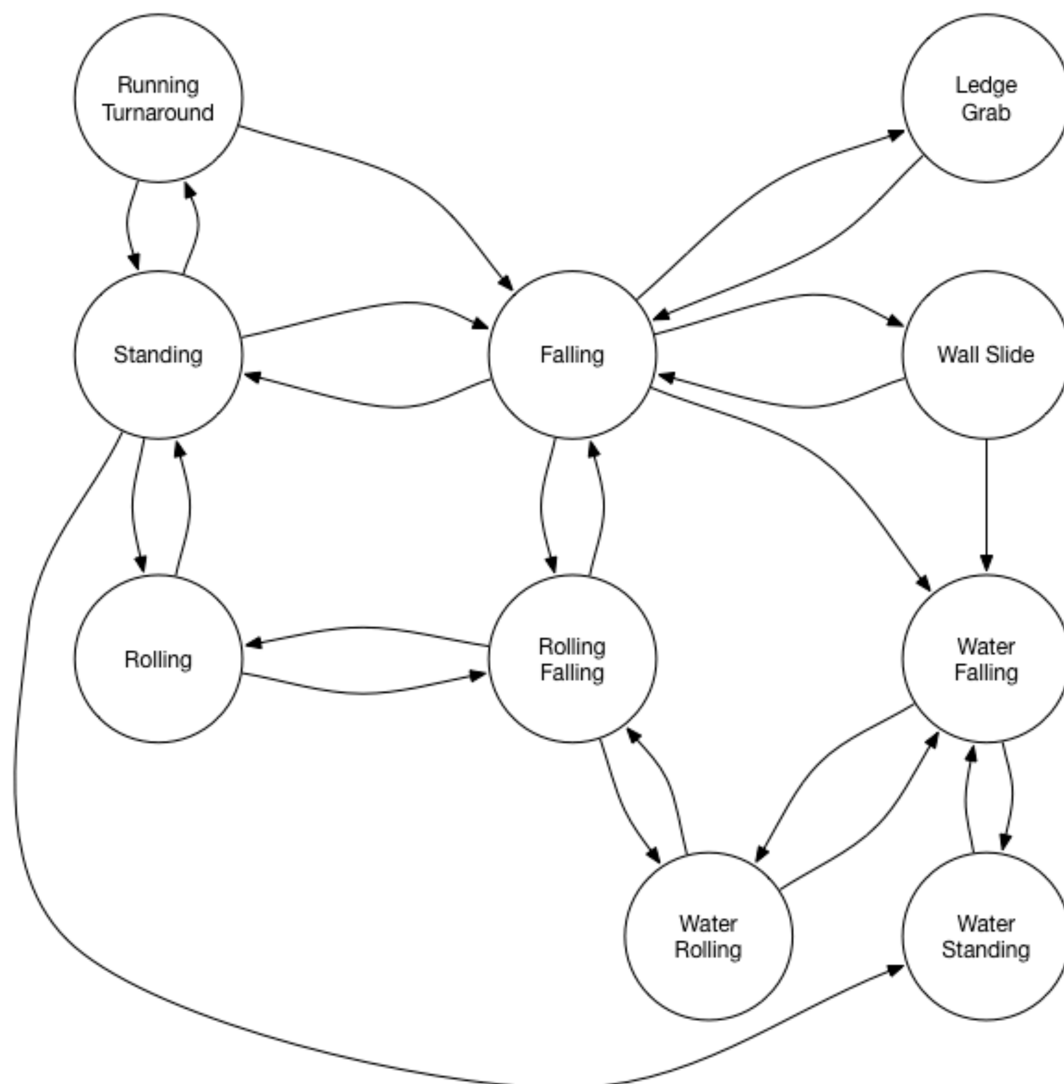


Ilustración 3. Máquina de estados del personaje jugable

Para implementar esta máquina de estados, dada su complejidad, se hará uso de un sistema propio que facilitará la tarea. Esto servirá, además, para poder comparar la *API* de gestión de máquinas de estado de José Alapont con una alternativa más simple y específica.

El siguiente diagrama muestra el objeto del juego del jugador, las clases que contiene y las relaciones que se dan entre ellas. De nuevo se ha simplificado y se omiten la mayoría de estados del protagonista.

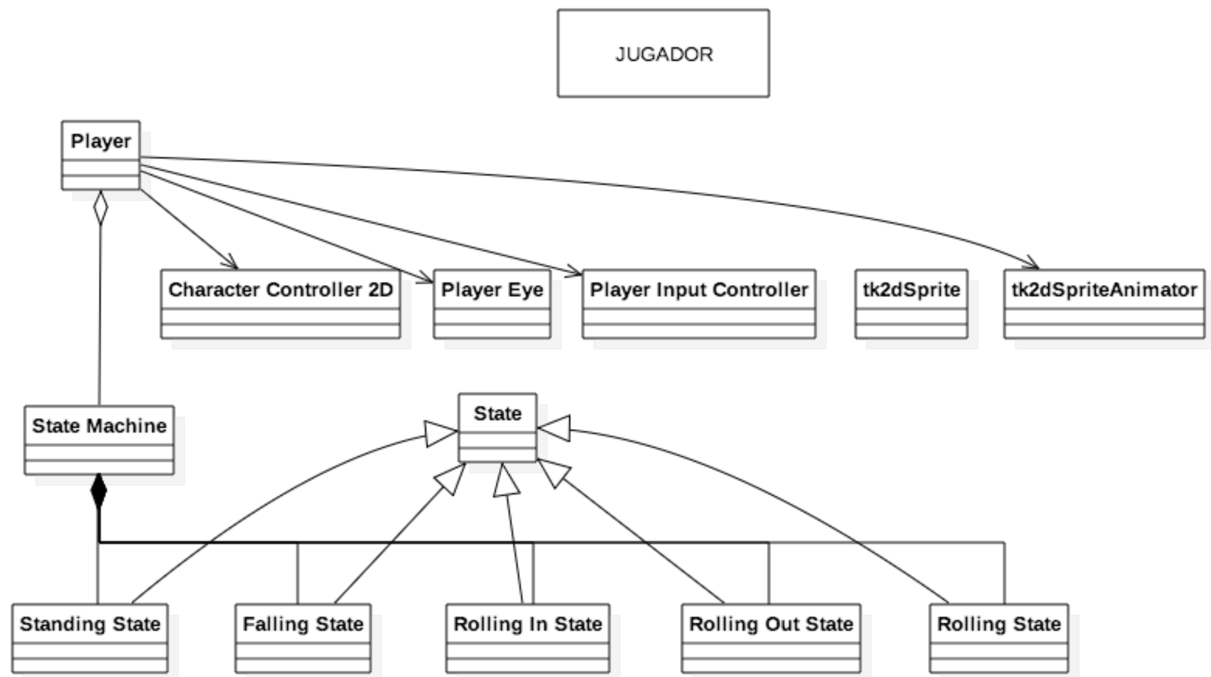


Ilustración 4. Diagrama de clases del jugador

3.3 Animales

Los escenarios del juego estarán poblados por seres vivos de todo tipo. Éstos poseerán una inteligencia artificial simple, de acuerdo con el tipo de animal que representan. Además podrán interactuar con el jugador y con su entorno.

Para definir el comportamiento de estos seres emplearemos la *API* de gestión de máquinas de estado.

A continuación se muestran los distintos animales y las máquinas correspondientes que describen su comportamiento:

Pájaro

El pájaro cuenta con una máquina de estados probabilista con un gran número de transiciones. Tiene dos variantes, la probabilista estándar y otra probabilista inercial que añade retardos entre los estados.

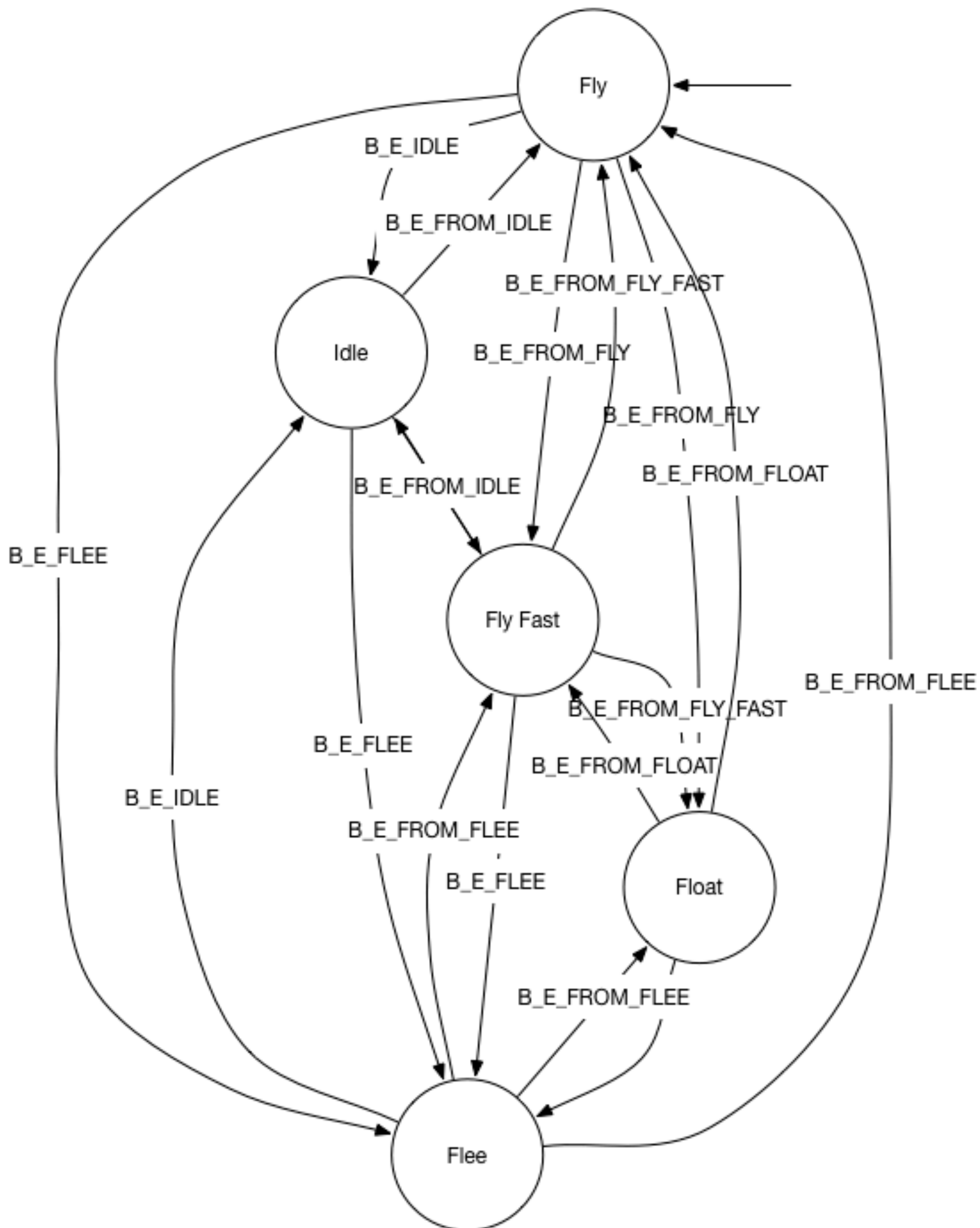


Ilustración 5. Máquina de estados del pájaro

Este animal empieza en el estado **Fly,** que describe un movimiento aleatorio en un área específica alrededor de su posición inicial. Una vez transcurrido un tiempo en este estado sale del estado inicial y se dirige hacia **Fly Fast** o **Float.** En **Fly Fast** realiza un comportamiento similar pero más rápido. En **Float** simplemente permanece estático en el aire durante un tiempo. En cualquiera de estos estados, si en algún momento toca el suelo pasará al estado **Idle,** reposando en el suelo unos instantes. Luego volverá a uno de los tres estados principales. Si el jugador se acerca al pájaro éste intentará huir en la dirección contraria transicionando al

estado **Flee.** Cuando se haya alejado lo suficiente volverá a su comportamiento habitual.

Pez

La máquina de estados que se encarga del comportamiento del pez es muy sencilla, contando únicamente con dos estados y dos transiciones. Es una máquina determinista.

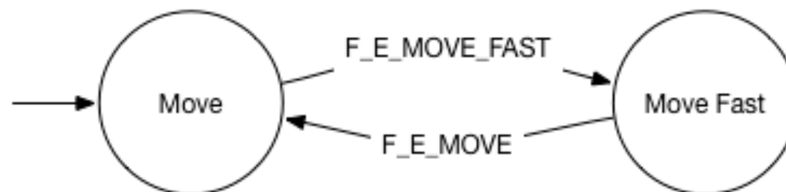


Ilustración 6. Máquina de estados del pez

El pez empieza en el estado **Move** y permanecerá en él hasta que el jugador interactúe con el animal. Así, el pez se desplazará lentamente y de manera aleatoria por un área rectangular alrededor de su posición inicial. Cuando el jugador interactúe pasará al estado **Move Fast**, en el que realizará un movimiento similar al de **Move** pero mucho más brusco y frenético. Al cabo de unos segundos volverá a su estado normal.

Medusa

La medusa tiene un comportamiento básico idéntico al del pez pero además añade un componente más. Se ha modelado usando una máquina concurrente por comodidad, pero podría simplemente haber constado de dos máquinas deterministas independientes.

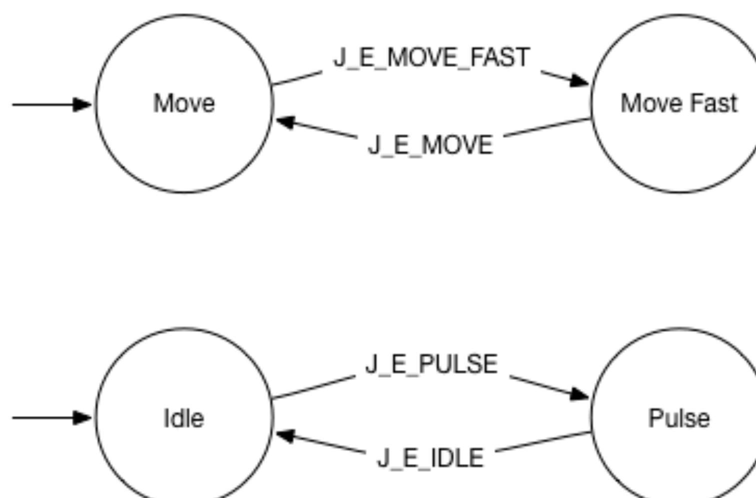


Ilustración 7. Máquina de estados de la medusa

Esta máquina cuenta con dos estados iniciales, **Move** e **Idle. Move** y **Move Fast** controlan el movimiento del animal, mientras que **Idle** y **Pulse** se encargan de su color de forma independiente. Cuando el jugador se encuentra cerca de una medusa esta pasa al estado **Pulse,** en el que cambiará de color rápidamente, con pulsaciones, como para alertar del peligro. Al alejarse el jugador volverá al estado **Idle,** retornando a su color normal.

Tortuga y Caracol

Esta máquina es compartida por dos animales distintos con comportamiento similar. Cuenta con dos versiones, una basada en pilas y otra determinista clásica.

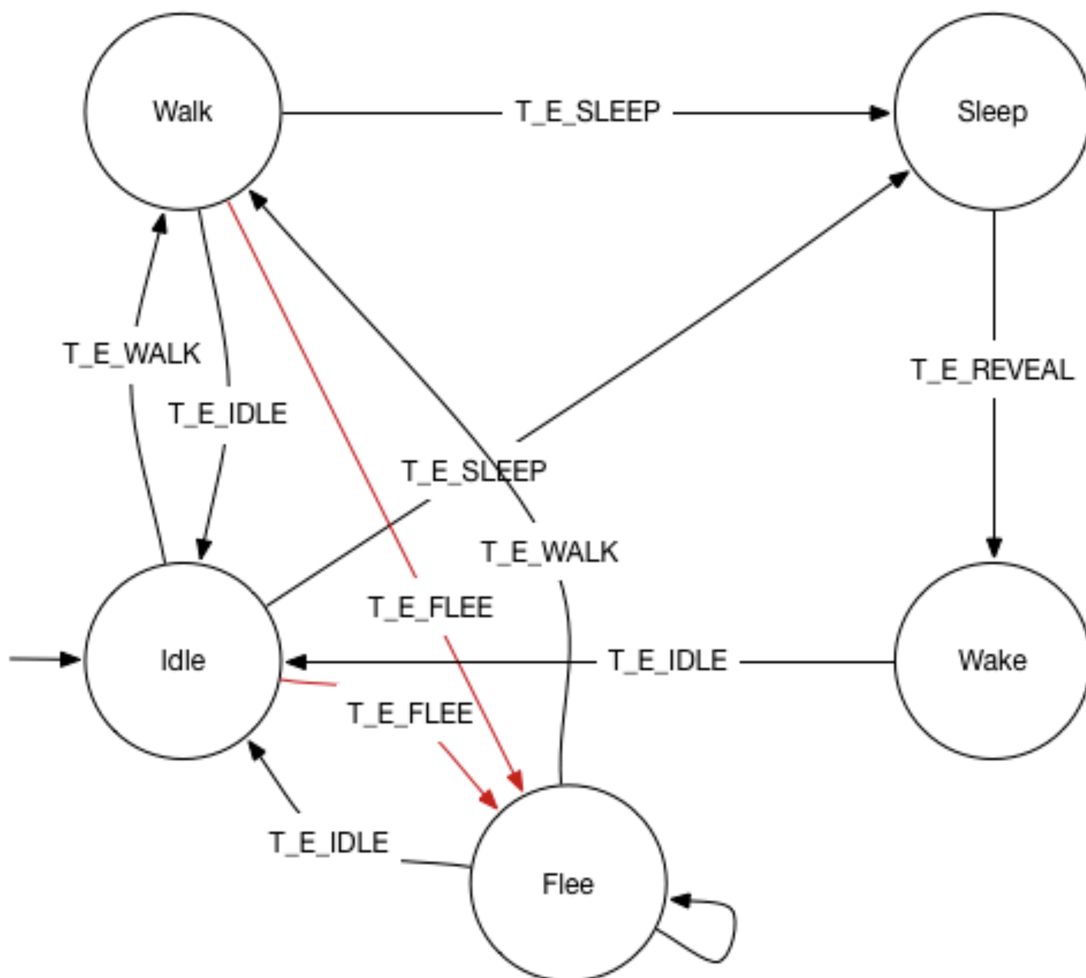


Ilustración 8. Máquina de estados basada en pilas de la tortuga y el caracol

El caracol empieza en el estado **Idle,** donde permanece quieto unos segundos. Luego comienza a andar en una dirección aleatoria durante un tiempo, para luego volver al estado **Idle.** Al cabo de un tiempo el caracol necesita dormir, por lo que pasará al estado **Sleep.** Después de reposar unos segundos se despertará en el estado **Wake** y a continuación volverá a **Idle.** Si el jugador se acerca al animal

mientras este se encuentra en los estados **Idle** o **Walk** se apilará el estado **Flee** (las transiciones rojas indican los eventos apilables) y la criatura tratará de huir y alejarse del jugador. Mientras el jugador permanezca cerca el estado seguirá apilado y el caracol no podrá transicionar a ningún otro estado, aunque necesite dormir. Cuando el jugador se aleje el estado se desapilará y el animal volverá automáticamente al estado en que se encontraba antes de comenzar huir.

En la máquina determinista alternativa únicamente hay una pequeña diferencia, cuando el animal ha pasado al estado **Flee** y el jugador se aleja, siempre volverá al estado **Idle**, pues no puede recordar el estado anterior como la basada en pilas.

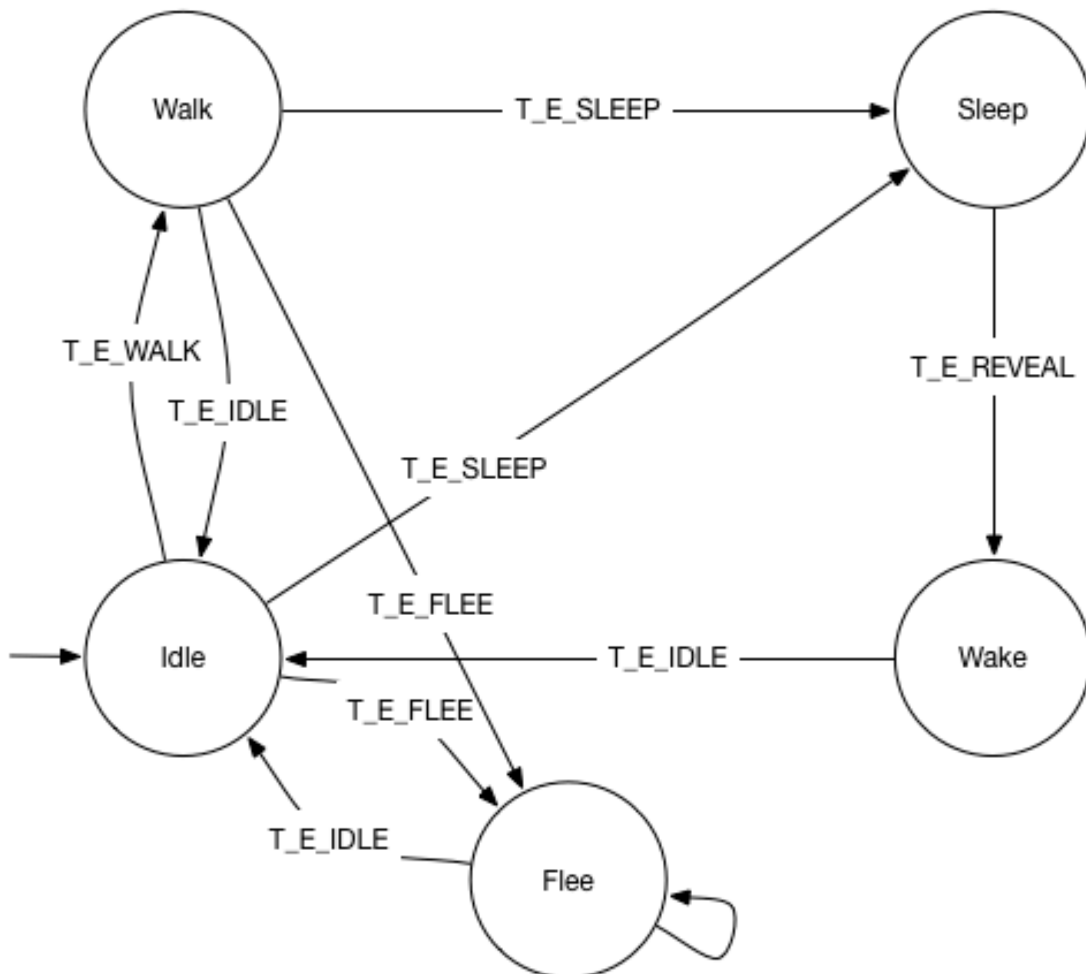


Ilustración 9. Máquina de estados determinista para la tortuga y el caracol

3.4 Menú de pausa

En cualquier momento de la partida se puede acceder al menú de pausa. Desde él se pueden configurar diferentes opciones de audio, acceder a los créditos del juego o salir de la partida. El diseño del menú es sencillo y flexible, de manera que

sea fácil añadir nuevos submenús y opciones en el futuro en caso de que sea necesario.

Se ha diseñado usando de nuevo la *API* de gestión de inteligencia artificial y se emplea una máquina de estados determinista para modelar su comportamiento.

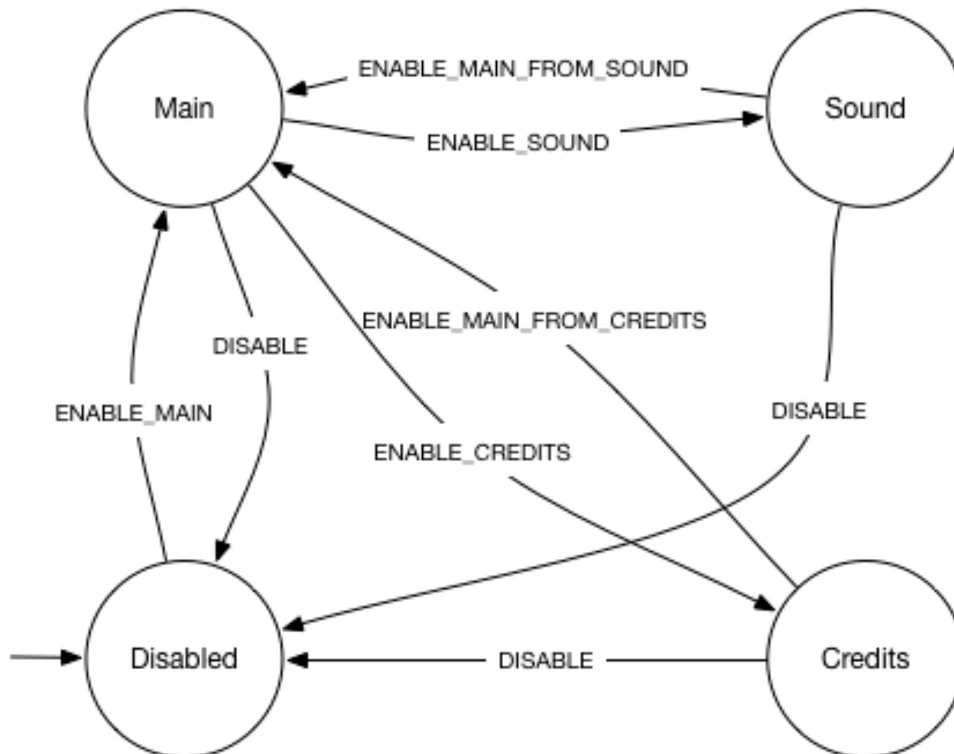


Ilustración 10. Máquina de estados del menú de pausa

El menú estará, en primer lugar, deshabilitado y oculto, esperando a ser invocado por el jugador. Una vez eso ocurre la máquina transita del estado **Disabled** a **Main,** mostrando el menú principal de pausa. En este momento el jugador puede elegir entre las opciones disponibles para acceder a otros submenús, reanudar la acción o salir del juego. Si el jugador escoge la opción de sonido la máquina pasará al estado **Sound**, ocultando el menú principal y mostrando el de audio. Desde aquí se podrá volver al estado **Main,** ocultando de nuevo el menú de sonido.

De igual manera, si selecciona la opción de créditos pasará al estado **Credits** y mostrará el submenú correspondiente con los créditos del videojuego. Desde cualquiera de los estados el jugador puede cancelar el menú para volver al juego, pasando al estado **Disabled** y ocultando todos los menús.

3.5 Managers

En la fase de diseño se detectaron cuatro clases importantes que permanecerán activas durante toda la ejecución del juego.

La primera es el **AudioManager**, que se encargará de gestionar la reproducción y reciclaje de los efectos de sonido. Dado el gran coste de instanciar y destruir objetos en ejecución [6], el **AudioManager** hará uso de un *pool*[6] de objetos, reciclando y reutilizando una serie de objetos para reproducir los sonidos, evitando instanciar nuevos. Este *manager* será accesible desde cualquier case, haciendo muy sencilla la reproducción de un sonido en cualquier contexto.

Por otra parte, debido a la gran cantidad de entidades que hará uso de máquinas de estados por medio de la *API* empleada, será necesaria otra clase que se encargue de la lectura de los archivos *xml* que contienen las descripciones de las máquinas, de manera que no tenga que realizarse la lectura para cada una de las entidades. **pbFSM_Manager** realiza esta función y es usada por el resto de clases para asignar la máquina deseada de entre las cargadas a sus objetos.

Para facilitar la gestión de la entrada, tanto de teclado como de mando, se ha diseñado la clase **CustomInput.** Esta clase da acceso a una serie de variables que contendrán el estado de los botones y *joysticks* de los que se hará uso en el juego, proporcionando más información sobre los mismos que la que se puede obtener por defecto con los métodos que ofrece Unity. El manager también tiene en cuenta una “zona muerta” para los *joysticks* de mando, dado que muchos no permanecen correctamente en su posición neutral y podía detectarse una entrada en casos no deseados.

En último lugar es necesaria una clase que se encargue de gestionar la lógica del juego. **GameLogic** guardará el progreso del jugador en la aventura y los puzles y se encargará de gestionar las transiciones entre los distintos escenarios por parte del jugador, así como del paso del tiempo virtual en el juego.

3.6 *API* de gestión de máquinas de estado

Usando la *API* de José Alapont se implementará la lógica de todos los animales del juego, así cómo la del menú de pausa.

La herramienta permite crear y gestionar máquinas de estado de distintos tipos de manera sencilla. Se ha hecho uso de la herramienta *FSM_Parser*, que facilita la carga de máquinas de estado desde un archivo *xml* como se describe a continuación.

3.6.1 Creación del documento *xml* con la máquina

En primer lugar es necesario elaborar un documento etiquetado con la descripción de la máquina de estados. Para ello partimos de una de las plantillas

[6] Un *pool*, o piscina de objetos, es un patrón de diseño para el desarrollo de software.

proporcionadas, según el tipo de máquina que queramos implementar. Este documento especificará los estados y las transiciones de que consta la máquina, así como las acciones que se realizarán en cada uno de ellos.

En primer lugar es necesario especificar el tipo de máquina de estados que se desea usar de entre los disponibles:

- **Clásica-Determinista:** Es la máquina más simple, enfocada a los comportamientos sencillos o muy generales.
- **Clásica-Probabilista:** Permite insertar probabilidades de activación entre las transiciones. Es una máquina indeterminista.
- **Inercial:** Es una versión alternativa de la máquina de estados clásica que soporta una latencia de ejecución de un estado.
- **Máquina basada en pilas:** Permite otorgar memoria a la máquina de estados, interrumpiendo la ejecución de un estado cuando la acción entrante es más prioritaria que la que estaba realizando. Al completar la acción nueva vuelve al estado en que se encontraba anteriormente.
- **Máquina de estados concurrentes:** Permite ejecutar varios estados al mismo tiempo, tantos como permita el diseñador.

Dentro del elemento con etiqueta “States” se encuentran los estados de la máquina. Cada uno de ellos estará definido con un elemento con etiqueta “State” hijo de “States”. El atributo “Initial” especificará si el estado es inicial o no. Los elementos que se encuentran dentro de “State” describen el resto de características del estado. Son los siguientes:

- **S_Name:** Nombre que identifica al estado.
- **S_Action:** Nombre de la acción que se realizará mientras la máquina se encuentre en el estado. Puede ser nulo.
- **S_inAction:** Nombre de la acción que se realizará al entrar al estado. Puede ser nulo.
- **S_outAction:** Nombre de la acción que se realizará al salir del estado. Puede ser nulo.
- **S_Fsm:** Enlace a una máquina de estados si se desea crear un estado jerárquico. En caso contrario se puede dejar vacío.

Una vez definidos los estados es momento de describir las transiciones entre los mismos. Éstas se encuentran dentro del elemento con etiqueta “Transitions”, cada una en un elemento “Transition” con los siguientes hijos:

- **T_Name:** Nombre que identifica a la transición.
- **T_Origin:** Nombre del estado origen de la transición.
- **T_Destination:** Nombre del estado destino de la transición.
- **T_Action:** Nombre de la acción que se realizará en el momento de la transición. Puede ser nulo.

Además, el elemento **Events** alberga una lista de eventos, cada uno definido con la etiqueta “Event” que tiene como hijos los siguientes elementos:

- **ID:** Nombre que identifica al evento.
- **Type:** Tipo de evento, puede ser “BASIC”, el evento estándar empleado en todas las máquinas de estados, o “STACKABLE”, usado en las máquinas de pilas para indicar que apila al estado origen.

3.6.2 Preparación de la clase *Tags*

La clase *Tags* es una clase estática que contiene las constantes numéricas que identificarán cada estado, transición, evento o acción de las máquinas de estados existentes. Para cada uno de estos elementos que creemos deberemos añadir una constante única y completar el método *StringToTag.*

3.6.3 Carga de la máquina con FSM_Parser

Tras los pasos anteriores la máquina de estados está preparada para ser cargada y usada. Para ayudar a realizar la carga se ha creado un manager llamado *pbFSM_Manager* que crea una instancia de *FSM_Parser*, busca todos los *xml* que contienen las máquinas de estado y los carga al principio de la ejecución.

```
xmltest.FSM_Parser parser = new xmltest.FSM_Parser();
fsm_manager = new FSM_Manager(parser);
DirectoryInfo dir = new DirectoryInfo(Application.streamingAssetsPath + "/AI/");
var files = dir.GetFiles("*.xml");

for (int i = 0; i < files.Length; i++)
    fsm_manager.addFSM(files[i].FullName);
```

Ilustración 11. Carga de las máquinas

El *script* simplemente accede a los archivos que se encuentran en una ruta determinada dentro de la carpeta *StreamingAssets* de Unity. Esta carpeta especial

permite acceder a los archivos en el equipo destino por medio del nombre de la ruta.

La clase que controla la IA de un objeto puede entonces acceder a la variable estática que contiene la instancia de *pbFSM_Manager* y crear una máquina del tipo elegido de entre las cargadas por el manager.

```
FSM = pbFSM_Manager.instance.fsm_manager.createMachine(this, Tags.CLASSIC, "BirdDeterministic");
```

Ilustración 12. Creación de la máquina de estados determinista del pájaro

3.6.4 Uso de las máquinas de estado

Buscando facilitar el uso y la preparación de las clases que se encargarán del control de la IA del juego se creó una clase abstracta, *AI_MonoBehaviour,* que desciende de *MonoBehaviour.* Esta clase proporciona todo lo necesario para hacer uso de una máquina de estados y define una serie de métodos que deberán ser implementados por las clases que hereden de ella.

```
public abstract class AI_MonoBehaviour : MonoBehaviour {

    public FSM_Machine FSM { get; protected set; }
    protected List<int> actionList;
    protected List<int> eventList = new List<int>();

    protected virtual void Start() {}

    protected virtual void Update()
    {
        actionList = FSM.UpdateFSM();

        for (int i = 0; i < actionList.Count; i++)
        {
            if (actionList[i] != Tags.UNKNOWN)
                ExecuteAction(actionList[i]);
        }
    }

    protected abstract List<int> CheckEvents();
    protected abstract void ExecuteAction(int action);
}
```

Ilustración 13. *AI_MonoBehaviour*

Dentro de *Start* se deberá crear la máquina de estados y asignarla a la variable *FSM. Update* actualizará la máquina en cada iteración y ejecutará el método *UpdateFSM* de la máquina. Este método acabará llamando a *CheckEvents,* el método que comprobará el estado de la máquina y generará eventos para realizar las transiciones y que deberá estar implementado teniendo en cuenta lo siguiente: Al principio del método será necesario limpiar la lista de eventos para evitar cargar con eventos de iteraciones anteriores y, en caso de que no se haya generado

ninguno durante la iteración actual, se añadirá el tag "UNKNOWN" (evento desconocido).

Una vez obtenida la lista con las acciones (identificadas con enteros), *Update* llamará a *ExecuteAction.* Esta función simplemente comprobará el identificador de la acción y realizará la acción correspondiente.

```
protected override void ExecuteAction(int action)
{
    switch (action) {
    case Tags.B_A_ENTER_IDLE:
        EnterIdle();
        break;
    case Tags.B_A_ENTER_FLY:
        SetTimer();
        Wander();
        break;
    case Tags.B_A_FLY:
        Fly(false);
        break;
    case Tags.B_A_FLY_FAST:
        Fly(true);
        break;
    case Tags.B_A_ENTER_FLOAT:
        SetTimer();
        break;
    case Tags.B_A_FLOAT:
        Float();
        break;
    case Tags.B_A_FLEE:
        Flee();
        break;
    default:
        break;
    }
}
```

Ilustración 14. Ejemplo de implementación de *ExecuteAction* en la clase del pájaro

3.7 Sistema propio de máquinas de estado

Se decidió implementar un sistema propio de máquinas de estado que sirviera de alternativa a la *API* de gestión de máquinas de estado de José Alapont en este proyecto, que facilitara la creación de unos objetos concretos que necesitaban de mucha flexibilidad en las transiciones y permitiera realizar comparaciones de rendimiento entre las herramientas.

Así, usando este sistema se implementará la lógica de personaje principal y también una versión alternativa de la tortuga, con el objetivo de facilitar las comparaciones de rendimiento.

Había dos objetivos fundamentales a la hora de crear el sistema, que eran:

- Implementará el patrón de diseño “estados como objetos”, es decir, cada estado estará contenido en una clase independiente, promoviendo la encapsulación del sistema.
- Cada estado conservará acceso al objeto original que contiene la máquina de estados.

El sistema estará compuesto por dos clases, una que define la máquina de estados y otra que representa un estado concreto.

3.7.1 Estado

Un estado debe pertenecer a una máquina de estados y conocer el objeto original que contiene dicha máquina.

La clase abstracta *State* define una serie de métodos para un estado que se describen a continuación:

- **onInitialized:** Se llamará al añadir el estado a una máquina de estados, para permitir realizar cualquier configuración inicial que sea necesaria.
- **Begin:** Método que se llama al transicionar a un estado.
- **Reason:** Método opcional que se ejecutará en cada iteración y que está pensado para permitir al estado comprobar las restricciones y cambiar a un nuevo estado.
- **Update:** Contiene el código con la lógica que se ejecutará cada iteración del estado.
- **End:** Se llamará al transicionar a un estado distinto al actual, antes de llamar a **Begin** en el nuevo.

3.7.2 Máquina de estados

La clase *StateMachine* contiene un diccionario con los estados de la máquina y se encargará de llamar a las funciones del estado cuando sea necesario. Además permite cambiar el estado actual y expone una variable de tipo *Action* llamada **onStateChanged** a la que se podrán suscribir otros objetos y que informará cuando se produzca un cambio de estado.

4. Planificación

Antes de comenzar la implementación del juego y tras estudiar las metodologías más empleadas en el desarrollo de software se decidió llevar a cabo una planificación del proyecto en base a iteraciones, haciendo uso del llamado desarrollo iterativo y creciente.

De esta manera, el proyecto se divide en bloques temporales, desarrollando una serie de características completas en cada bloque de manera secuencial. Para cada iteración se realiza un ciclo de desarrollo completo, incluyendo una fase de pruebas y documentación, antes de poder pasar a la siguiente iteración.

Esta metodología es especialmente adecuada para un videojuego por ser un tipo de proyecto en el que las fronteras entre análisis, diseño, implementación y testeo suelen ser difusas. Empleando el desarrollo creciente se obtiene, al final de cada iteración, un producto funcional y jugable, una especie de prototipo parcial en base al cual se basarán las siguientes iteraciones.

El desarrollo del proyecto se dividió en siete iteraciones, que se describen en detalle a continuación:

1ª iteración, 3 semanas

En esta primera iteración se trabajará en el movimiento del protagonista, implementando las siguientes características:

- **Script de gestión de colisiones y movimiento.** Se usará de base para el movimiento de todos los objetos que necesiten colisionar con el terreno y no estén gestionados directamente por el motor de físicas de Unity, es decir, el protagonista y los animales que poblarán el mundo.
- **Sistema propio de máquinas de estado**, del que hará uso el protagonista.
- **Scripts de gestión del *input.***
- **Código de movimiento del jugador, gestión de su máquina de estados.**
- **Arte provisional para el protagonista.**
- **Movimiento de la cámara**, que seguirá al jugador a lo largo del mundo.
- **Escenario provisional**, con los elementos necesarios para el testeo del movimiento del jugador.

2ª iteración, 2 semanas

Una vez finalizado el código relacionado con el jugador se creará un escenario cercano al diseño final y se actualizará el protagonista con arte y animaciones finales. La lista de tareas a realizar en esta iteración queda así:

- **16 animaciones del personaje principal**, usando más de 100 *sprites.*
- **Creación del sistema de *colliders* animados del agua.** El agua es un elemento fundamental en los escenarios del juego por sus implicaciones jugables y estéticas. En esta iteración se creará un sistema para animar una malla de agua bidimensional. Se implementará un *shader* simple para imitar un efecto de refracción, que desplace ligeramente la imagen de los objetos situados detrás de una masa de agua.
- **Creación del sistema de animación de plantas.** Se implementará un sistema para animar los *sprites* de vegetación cuando el jugador pasa por encima.
- **Escenario de cueva.** El escenario tendrá su diseño final pero contará en esta etapa con arte provisional.

3ª iteración, 2 semanas

Con la jugabilidad básica del jugador ya creada y un escenario bastante completo implementado en el editor, esta iteración servirá para completar las características del protagonista relacionadas con la interacción. Además se comenzará a trabajar en el ciclo de día y noche.

Así, en esta iteración se implementará:

- **Onda de interacción del protagonista y objetos interactuables.** El personaje puede interactuar con los objetos y animales cercanos y estos pueden realizar alguna acción personalizada en respuesta.
- **Coger y lanzar objetos.** El jugador podrá coger y lanzar los objetos que cuenten con un componente específico que incluya esta funcionalidad y los marque como objetos susceptibles de ser cogidos.
- **Ojo del jugador.** El protagonista tiene un gran ojo y será capaz de observar el entorno. El personaje dirigirá su mirada a objetos cercanos que posean un componente que los haga "observables".
- **Creación del cielo**, que cambiará de color según avance la hora del día. Para ello se creará un *shader* que permitirá pintar un objeto con un gradiente vertical, que luego se modificará en tiempo real desde código para reflejar la hora del día.

4ª iteración, 3 semanas

En este momento se completarán los escenarios y se incluirá el arte final de los mismos, entre otras cosas.

Tareas:

- **Creación del segundo escenario de cueva y de la isla principal.**
- **Actualización de la primera cueva con el arte final.**
- **Creación de un elemento puerta**, que permitirá desplazarse entre los niveles, incluyendo la lógica de fondo para que todo funcione sin problemas, incluyendo el llevar objetos de un escenario a otro.
- **Creación de un sistema de paralaje**, que permitirá a objetos de fondo lejanos desplazarse más lentamente que aquellos cercanos a la cámara.
- **Creación de las estrellas del cielo**, que se harán visibles durante la noche y se ocultarán al llegar el día.
- **Inclusión de un sistema de mensajes**, que permitirá mostrar texto al jugador cuando se acerca a elementos como carteles.
- **Objetos rompibles.** Algunos objetos que se pueden coger y lanzar se romperán al impactar contra una superficie a gran velocidad.

5ª iteración, 2 semanas y media

En esta iteración se llevarán a cabo las labores que se explican a continuación:

- **Creación de la lógica de los puzles.**
- **Expresiones del jugador.** El ojo del protagonista se usará para aportar más información al jugador sobre las acciones que está realizando. Así, el ojo mostrará cambios visuales al coger un objeto o resolver un puzle.
- **Añadido de detalles y decoración a los escenarios.** Se incorporarán plantas y objetos decorativos a los escenarios para darles más vida y hacerlos más atractivos para el jugador.
- **Creación de los animales y su inteligencia artificial.** Se crearán seis animales distintos, cada uno con sus propias animaciones.

6ª iteración, 2 semanas

Este bloque estará centrado en la implementación del audio y la iluminación, requiriendo la realización de las siguientes tareas:

- **Preparación del sistema de *AudioMixers* que gestionará el volumen de los efectos de sonido y la música.**
- **Creación del Audio Manager y del *pool* de sonidos.**
- **Inclusión de la canción principal y los efectos de sonido.**
- **Iluminación de escenarios y ajuste del ciclo de día y noche.** Se añadirán luces puntuales a lo largo de los escenarios para iluminarlos. La iluminación global de las escenas vendrá dada por la luz ambiental, que dependerá del ciclo de día y noche.
- **Creación del menú de pausa.**

7ª iteración, 1 semana y media

Con toda la funcionalidad básica del proyecto terminada, esta etapa se centrará en las mejoras y la optimización. Se testeará el proyecto globalmente, se añadirán más detalles y se realizarán los últimos retoques.

- **Más decoración en los escenarios.**
- **Más *feedback* para las acciones del jugador.** Por ejemplo, la pantalla temblará en ciertos momentos para dar información al jugador.
- **Animación inicial con el título del juego.**

El tiempo de desarrollo total del proyecto según esta estimación es de 16 semanas.

A continuación se muestra un diagrama de Gantt con la planificación temporal del proyecto:

	Nombre de la tarea	Duración	P2			P3			P4		
			Abr	May	Jun	Jul	Ago	Sep	Oct	Nov	Dic
1	1ª iteración	21 d			1ª iteración						
2	2ª iteración	14 d			2ª iteración						
3	3ª iteración	14 d				3ª iteración					
4	4ª iteración	21 d						4ª iteración			
5	5ª iteración	18 d						5ª iteración			
6	6ª iteración	14 d							6ª iteración		
7	7ª iteración	11 d								7ª iteración	

Ilustración 15. Diagrama de Gantt

5. Implementación

Tras las fases de diseño y planificación el proyecto estaba preparado para su implementación. Esta ha sido la fase más costosa y duradera del desarrollo del proyecto y ha servido para reforzar y ampliar los conocimientos de Unity y las distintas herramientas empleadas.

Dada la complejidad del proyecto, como resultado de la implementación se han obtenido más de 60 clases y alrededor de 8000 líneas de código que contienen las funcionalidades descritas durante la fase de diseño.

A continuación se describirán y documentarán los pasos que se llevaron a cabo durante la implementación en orden cronológico, de acuerdo con la planificación establecida anteriormente.

5.1 Colisiones

Antes de comenzar a trabajar en la implementación del protagonista era necesario crear un *script* que se encargara de resolver las colisiones del mismo con el terreno. Era vital separar esta funcionalidad en un componente independiente de la lógica propia del jugador para así poder reusarlo en todas las demás entidades que se moverán por el escenario, como los animales.

Tal componente deberá, conociendo la posición actual de la entidad y la velocidad de la misma (la intención de movimiento para una iteración), actualizar su posición teniendo en cuenta las colisiones con el escenario.

Los escenarios del juego debían ser orgánicos, conteniendo inclinaciones y curvas de todo tipo, por lo que se decidió emplear un sistema de *raycasts* para resolver las colisiones. Un *raycast* es simplemente un rayo de cierta longitud que se emite en una dirección y que comprueba si intersecta en algún punto con alguna superficie. Así, si se emite una serie de rayos en la dirección en que se esta moviendo el personaje y se iguala la longitud de los mismos a la velocidad actual del jugador, se puede conocer si éste va a colisionar con algún terreno durante la iteración actual y actuar en consecuencia.

Teniendo todo esto en cuenta se crea el *script CharacterController2D,* que realiza esta función y expone algunas variables para permitir configurarlo de manera sencilla según los requisitos de la entidad que lo va a utilizar.

Así, se puede elegir el número de rayos que se emitirán para el movimiento horizontal y vertical de manera independiente, según el tamaño del personaje o de las plataformas del terreno, o determinar el ángulo máximo de las pendientes que el personaje podrá escalar, entre otros.

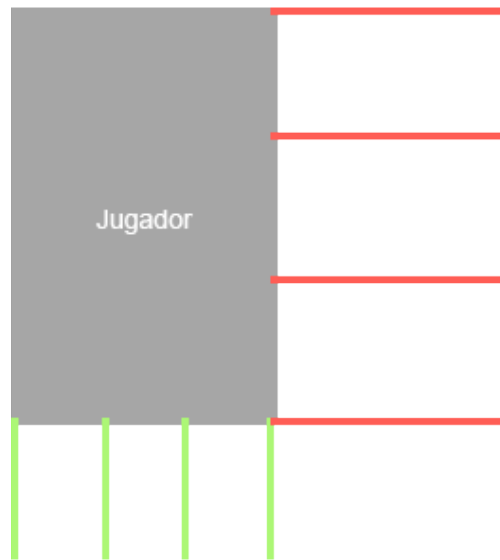


Ilustración 16. Representación visual del sistema de *raycasts*

Todas estas variables se pueden modificar desde el editor de Unity, incluso durante la ejecución del juego.

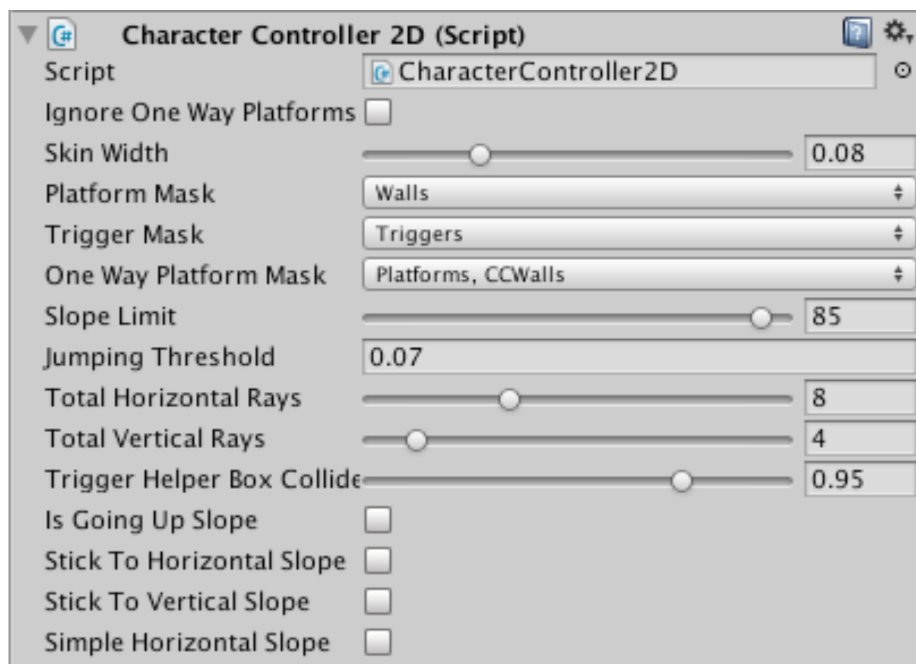


Ilustración 17. Componente *CharacterController2D* en el inspector

5.2 Protagonista

Antes de abordar la lógica propia del personaje principal era necesario encargarse de la gestión de la entrada de teclado y mando, para poder separarla en componentes independientes de la lógica.

Con este fin se crean dos *scripts*, *CustomInput,* que rellena algunos huecos existentes en la *API* de entrada de Unity, y *PlayerInputController,* que se apoya en el

anterior para gestionar la entrada específica para el protagonista. Su única función es la de permitir al script del jugador recibir la entrada mientras el componente *PlayerInputController* esté activo.

A continuación se creó la clase *Player,* que contiene la lógica general del protagonista, las constantes numéricas que definen el comportamiento del mismo y la máquina de estados. El protagonista cuenta con 17 estados distintos, contenidos en sus clases correspondientes.

El protagonista requiere de los siguientes componentes para funcionar correctamente:

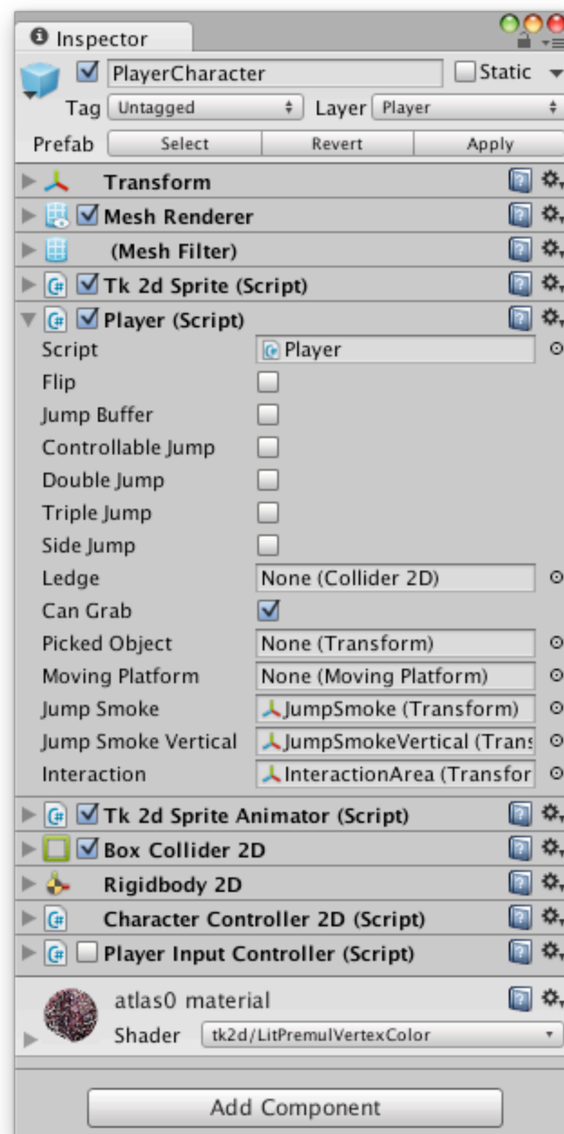


Ilustración 18. *GameObject* del protagonista en el inspector

Además de *Player, CharacterController2D* y *PlayerInputController,* que ya han sido descritos, se observan los componentes *tk2dSprite* y *tk2dSpriteAnimator,* que se encargan de mostrar y animar los *sprites* del personaje. El resto son componentes básicos de Unity: El componente *Transform,* como se explicó en el punto **2.5** contiene y gestiona la posición, escala y rotación del objeto. *MeshRenderer* y *MeshFilter* trabajan conjuntamente para renderizar la geometría del jugador en la posición dada por *Transform.* Por último, *BoxCollider2D* define una caja de colisión rectangular bidimensional que se apoya en *Rigidbody2D* para registrar las colisiones con otros objetos.

5.3 Cámara

En este momento se implementa un componente llamado *SmoothFollow* que se encarga de desplazar la cámara para que siga al jugador a lo largo del escenario. Esto se consigue haciendo uso de la función *SmoothDamp* de *Unity,* que permite cambiar un valor gradualmente a lo largo del tiempo para igualarlo a otro valor destino. Así, la cámara se desplaza describiendo un movimiento fluido, con un retardo muy pequeño respecto al movimiento del personaje.

Además, se define también una "ventana" dentro de la cual deberá permanecer siempre el protagonista, es decir, una distancia máxima entre la posición de la cámara y el jugador. Si no se aplicara esta restricción el personaje podría quedar fuera del campo de visión en algunos casos en que se está desplazando a gran velocidad.

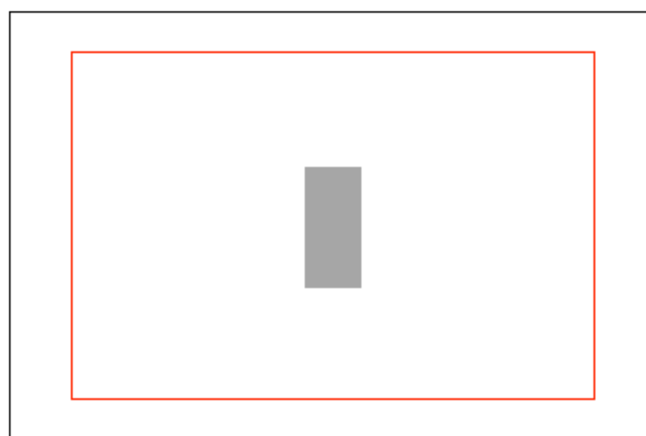


Ilustración 19. "Ventana" de la cámara

En la ilustración anterior se muestra este comportamiento. El rectángulo rojo representa el espacio de la pantalla en el que podrá encontrarse el jugador, en ningún momento podrá encontrarse fuera de esa zona.

5.4 Agua

Para las masas de agua de los escenarios se decidió realizar un modelado con muelles. Partiendo de una malla de segmentos rectangulares de agua, cada vértice superior de la malla actuará como un muelle, que se desplazará hacia abajo cuando un objeto entra en el agua y recuperará lentamente su posición inicial de reposo. Los scripts *Water2D* y *Water2DCollider* se encargan de modelar este comportamiento.

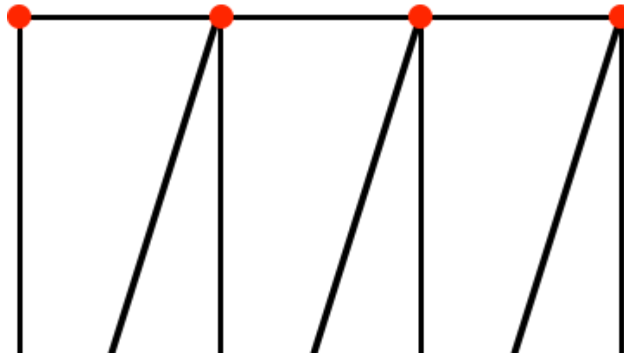


Ilustración 20. Malla de polígonos del agua

En primer lugar se crea la malla de segmentos y se prepara un *collider* para la masa de agua, que estará esperando a recibir la entrada de un objeto en el agua. Cuando esto ocurre se calculan los nodos (vértices) afectados por el impacto teniendo en cuenta la posición del objeto al penetrar en el agua y la anchura de los segmentos. Estos nodos afectados se desplazarán verticalmente proporcionalmente a la velocidad del objeto que ha colisionado.

En cada iteración a partir de este impacto inicial se propagará la fuerza a los nodos contiguos, disipándose lentamente. Parecerá que se están creando pequeñas olas en la superficie del agua.

Además, este script se encargará también de empujar a algunos objetos marcados como flotantes hacia arriba, desplazándolos a la superficie y haciendo que se mezan con las olas.

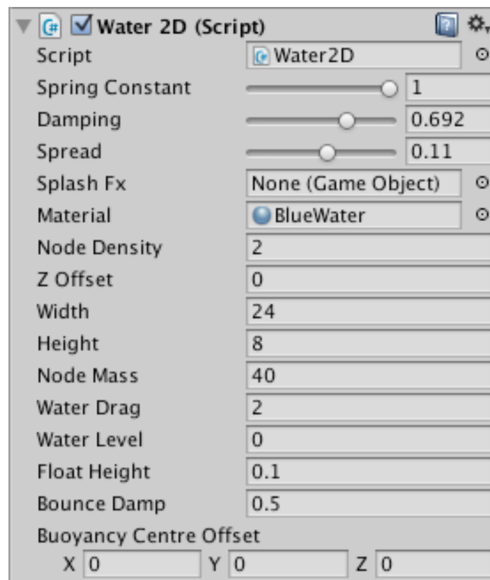


Ilustración 21. *Water2D* en el inspector.

El componente *Water2D* expone muchos parámetros que permiten configurar el comportamiento del agua y de los muelles, haciendo posible ajustar la disipación de los muelles, la anchura y densidad de los nodos, el rozamiento de los objetos en el agua, etc.

Para conseguir el aspecto visual deseado para el agua se diseñó un *shader* que recibe como parámetros un color y una textura. Este *shader* captura la imagen de todo lo que se encuentra detrás de la masa de agua, la distorsiona levemente de manera aleatoria a lo largo del tiempo haciendo uso de la textura que recibe (una imagen con ruido aleatorio) y tinta la imagen resultante con el color del agua.

A continuación se muestra la función que se aplicará a cada píxel del agua, este código se encarga de las operaciones descritas anteriormente:

```
float2 displacement = tex2D(_DispTex, i.scrPos).xy;
float t = i.scrPos.y + displacement.y * ((sin(_Time.y) + 0.5) * 0.01);

return tex2D(_GrabTexture, float2(i.scrPos.x, t)) * (_Color);
```

Ilustración 22. Contenido de la función de procesado de fragmentos

Con todas las piezas que componen el agua completadas, este es el resultado en ejecución:

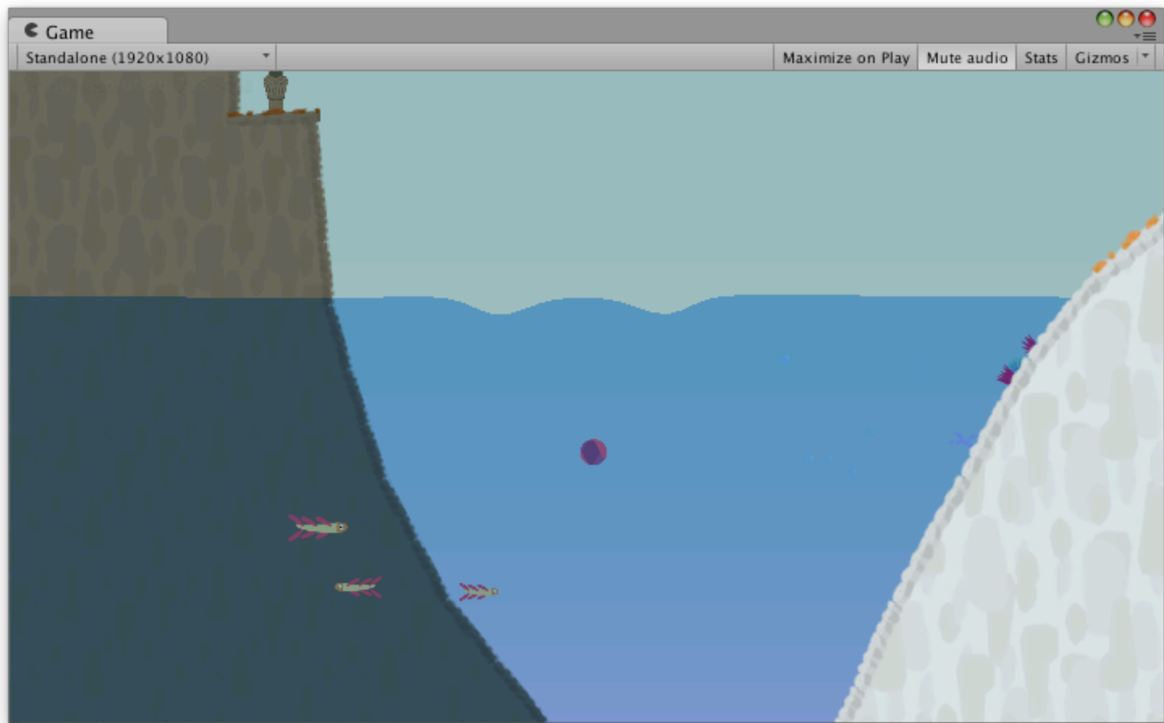


Ilustración 23. Resultado final en el juego

5.5 Vegetación

Para las plantas del juego se implementó un sistema que guarda muchas similitudes con el anterior empleado para el agua, aunque es menos complejo. En este caso se modelan de nuevo los vértices superiores del *sprite* de la planta como muelles, que se desplazarán horizontalmente cuando un objeto o personaje pase por encima de la misma.

El componente que se encarga de modelar las plantas se llama *Foliage.* No se entra en detalle en la implementación de este componente por seguir unos principios tan parecidos a los explicados en la sección anterior.

5.6 Ojo del protagonista

Para conseguir la expresividad que se buscaba en el personaje principal se decidió tratar el *sprite* general del protagonista y el de la pupila por separado, para que ésta pueda desplazarse independientemente. *2D Toolkit* posee una característica que permite definir puntos de anclaje en los *sprites* de una colección. Se ha establecido un punto de anclaje en todos los *sprites* en los que el ojo del jugador es visible, para así tener una posición base para la pupila en todos los estados posibles del personaje.

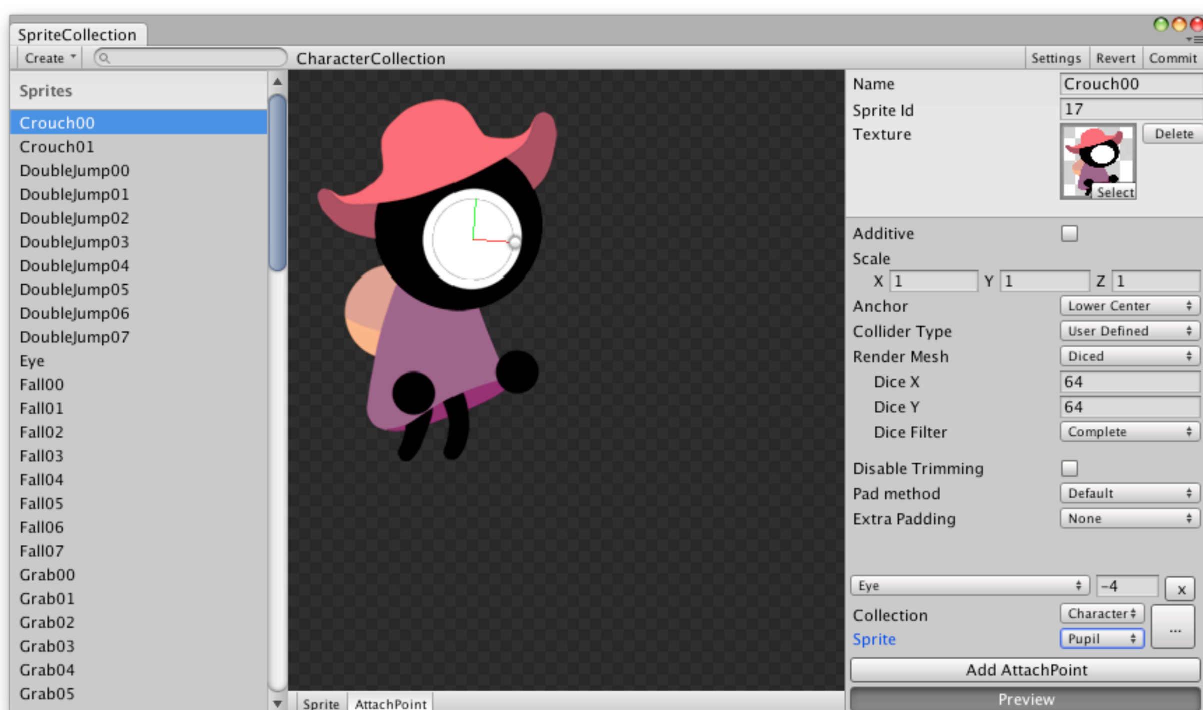


Ilustración 24. Punto de anclaje de la pupila en un *sprite*

Una vez definida esa posición base se crea un nuevo componente llamado *Player Eyes*, que se añadirá a la lista de componentes ya existentes para el protagonista. Este componente se encargará de mover la pupila a lo largo del ojo del personaje

de manera que parezca que el protagonista está mirando los objetos o personajes importantes que le rodean.

Para ello *Player Eyes* se apoyará en otro componente, *Trackable,* que se deberá asignar a aquellos objetos en que el protagonista puede centrar su atención. Además de gestionar el movimiento de la pupila, *Player Eyes* se encargará de hacer parpadear al personaje de manera natural a lo largo del tiempo.

5.7 Onda de interacción

El jugador podrá interactuar con el entorno por medio de una onda mágica. Esta onda estará representada en el juego con el objeto *InteractionArea*. Este objeto contiene un componente *Interaction*, que se encargará de realizar la animación de la onda mágica y destruirá el objeto tan pronto como ésta acabe.

Los objetos interactuables deberán implementar la interfaz *IInteractuable*, que requiere únicamente de la implementación del método *Interact.* Este método representa la acción que realizará el objeto cuando el jugador interactúe con él.

El jugador, sin importar el estado actual del personaje, podrá lanzar una onda pulsando una tecla. De esto se encarga también *Player.* En el momento en que se lance la onda se comprobará si alrededor del personaje existe algún objeto interactuable y, en caso de que así sea, se invocará *Interact* en ese objeto u objetos.

5.8 Coger y lanzar objetos

El jugador podrá también coger y lanzar objetos independientemente del estado en que se encuentre.

El componente *Throwable* describe el comportamiento de aquellos objetos que pueden ser cogidos por el jugador. Este tipo de objetos estarán en primer lugar controlados por el motor físico de Unity. Cuando el jugador pulsa el botón de coger, si no está sosteniendo un objeto ya y hay alguno disponible cerca de su posición, llamará a la función *PrepareToBePicked* de ese objeto. En este momento se modificará la variable *isKinematic* del objeto y dejará de depender de la física de Unity, pasando a estar controlado desde el código de *Throwable.* El objeto seguirá al personaje con un pequeño retardo, como si estuviera siendo controlado telepáticamente.

Si el jugador vuelve a pulsar el botón de coger se lanzará el objeto que se estaba sosteniendo. Este saldrá disparado en la dirección elegida y pasará de nuevo a ser independiente del jugador y a estar controlado por la física de Unity.

En este momento también se decide implementar el objeto *Sinkhole*, que se encarga de atraer y sostener objetos *Throwable*. Esto será útil más adelante en la implementación de los puzles.

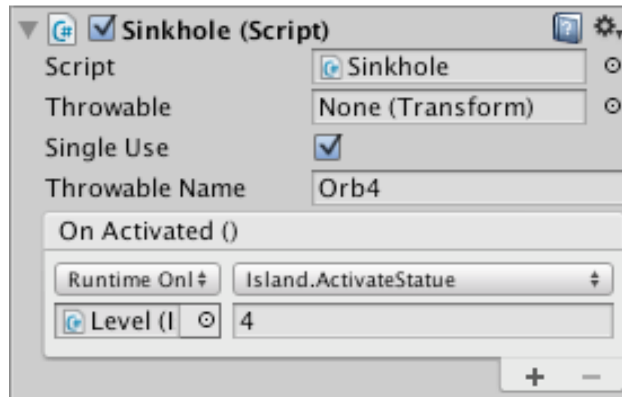


Ilustración 25. *Sinkhole* en el inspector

Un *Sinkhole* puede ser de un solo uso, es decir, una vez que ha absorbido un objeto este no se puede volver a coger. Además, puede especificarse un nombre concreto para que el agujero sólo pueda atraer el objeto que posee dicho nombre. Por último, se expone una variable de tipo *UnityEvent*, que permitirá llamar a un método especificado cuando el agujero haya absorbido un objeto.

5.9 Ciclo de día y noche

Para implementar el ciclo de día y noche hace falta un *script* que se encargue de registrar el paso del tiempo virtual del juego.

GameLogic es un componente que permanecerá activo durante toda la partida y que gestionará la lógica del juego. Contendrá una variable con el tiempo actual en el juego, que se incrementará en cada iteración siempre que el juego no esté pausado. Un ciclo completo dura 360 segundos, con el día ocupando 260 segundos y la noche solamente 100.

Este *script* contiene dos eventos a los que se pueden suscribir otros objetos para ser notificados en el anochecer y el amanecer, permitiendo realizar acciones en consecuencia (por ejemplo modificar el comportamiento de un *NPC*[7] o activar una fuente de luz cuando el escenario se vuelve oscuro al anochecer.

```
// Events
public event Action onDusk;
public event Action onDawn;
```

Ilustración 26. Eventos de *GameLogic*

[7] *Non-Player Character*, personaje no jugable

Además, era necesario un elemento que sirviera de cielo para los escenarios y cambiara de color para reflejar la hora del día. Esto se ha conseguido por medio de un rectángulo situado en el fondo de las escenas, que contará con un *shader* específico que "pinte" el rectángulo con un degradado, así como un *script* que se encargue de modificar los colores del *shader.* El fondo se moverá a la vez que la cámara y ajustará su tamaño para llenar completamente el campo de visión, que podrá variar según el nivel de zoom actual de la cámara.

El *shader GradientOffset* recibe como parámetros un color superior, un color inferior y un desplazamiento "offset" y crea un degradado fluido entre los dos colores. El desplazamiento describe la importancia de los colores al dibujar el degradado, la proporción de espacio que ocupará cada uno. Este parámetro se incluyó porque se deseaba que el color inferior estuviera limitado a una porción pequeña del rectángulo.

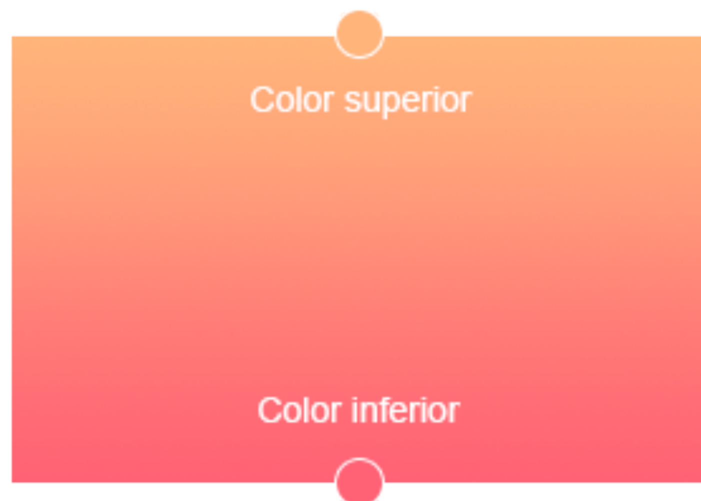


Ilustración 27. *GradientOffset* crea un gradiente a partir de dos colores

BackgroundTransition es el *script* que realiza el resto del trabajo descrito anteriormente. Este componente define una serie de paletas de color para las diferentes horas del día, así como una luz ambiental correspondiente para cada paleta. La luz ambiental definirá cómo se iluminan todos los objetos de la escena. Desde este *script* es posible también deshabilitar la luz ambiental si así se desea, como es el caso en escenarios cerrados en los que no se ve el cielo, como las cuevas.

Dadas las paletas para un ciclo, *BackgroundTransition* dividirá el tiempo total del ciclo entre el número de paletas para obtener la duración de cada sección. Luego comenzará a actualizar los colores del gradiente, interpolando entre el color de la paleta correspondiente a la sección actual y el de la siguiente, para que las transiciones sean siempre fluidas.

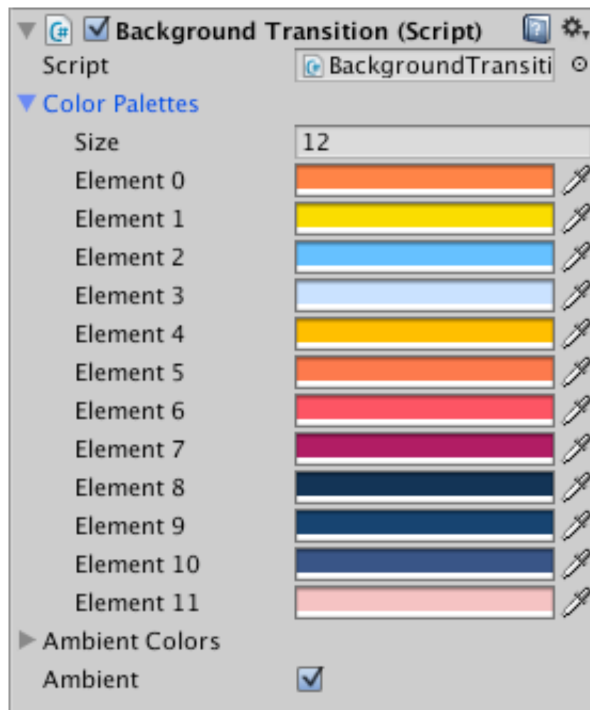


Ilustración 28. *BackgroundTransition* en el inspector

5.10 Paralaje

Para crear la sensación de profundidad en los escenarios se creó un sistema muy simple de paralaje que recibe una serie de objetos o grupos de objetos y asocia un valor a cada uno de esos grupos. Ese valor definirá la proporción de movimiento del objeto respecto a la cámara. Si el valor es igual a 0 no habrá diferencia en el desplazamiento del objeto. Cuando el valor aumenta el objeto actuará como si estuviera cada vez más cerca de la cámara. Y al contrario, parecerá que está más lejos cuanto menor sea el valor por debajo de 0.

El componente *Parallaxing* implementará este sistema.

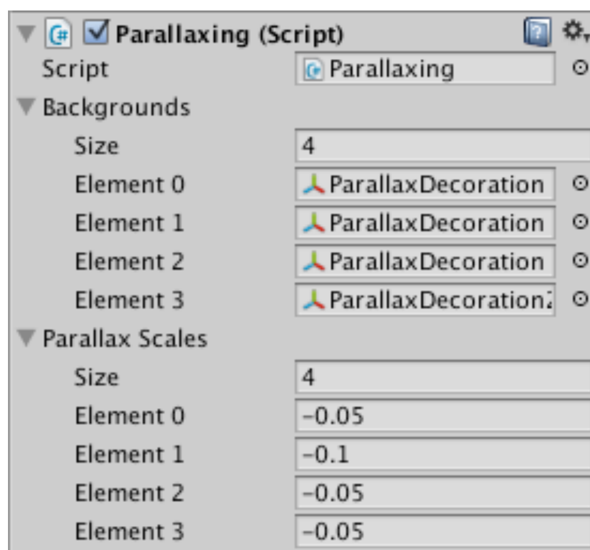


Ilustración 29. *Parallaxing* en el inspector

5.11 Puertas y transiciones entre niveles

Los niveles del juego están conectados entre sí por medio de puertas que el jugador puede atravesar pulsando una tecla.

El componente *Door* contiene la información de una transición entre niveles, es decir, el nivel destino y las coordenadas de aparición dentro de ese nivel. Cuando el jugador está situado delante de una puerta y usa la tecla adecuada se deshabilitarán los controles y se realizará la transición al nuevo escenario. *GameLogic* realiza las siguientes acciones al cambiar de nivel:

Realiza un fundido a negro, hace nulas las *actions* descritas anteriormente en el apartado del ciclo de día y noche (para evitar llamar a un objeto que se hubiera suscrito en el nivel actual y que desaparecerá con la transición), carga el escenario deseado, coloca al jugador en la posición especificada, funde desde negro y habilita de nuevo los controles en el jugador.

Además hay un caso especial que habrá que tener en cuenta, que se da cuando el jugador cambia de nivel mientras sostiene un objeto. En este caso, además de todo lo anterior, se asignará el jugador como padre del objeto que está sosteniendo en la jerarquía de Unity para que no se destruya automáticamente al cargar el nuevo escenario. Una vez cargado se volverá a establecer como objeto independiente.

5.12 Estrellas

Para crear las estrellas que pueblan el cielo durante la noche se ha empleado un sistema de partículas de Unity con un tiempo de vida infinito y una emisión muy alta, de manera que las partículas permanezcan estáticas durante la ejecución. Dado el gran número de estrellas es mucha mejor opción usar un sistema de partículas que añadir muchos *sprites* de estrellas distintos. Crear una textura tan grande como la resolución máxima de la pantalla con las estrellas tampoco es una solución muy eficiente.

El componente *Stars* está suscrito a los eventos ***onDusk*** y ***onDawn*** de *GameLogic* y hace aparecer y desaparecer las estrellas lentamente cuando estos eventos ocurren.

5.13 Sistema de mensajes

Para aportar información al jugador y poder enseñarle los controles del juego era necesario un elemento que mostrara texto. Se buscaba que este elemento estuviera integrado en el propio mundo del juego y que presentara la información de manera automática al acercarse el jugador.

Sin embargo, antes de hablar del propio sistema de mensajes es importante hablar del *canvas* de Unity, que nos permitirá mostrar el texto.

5.13.1 Canvas

El *canvas* es el elemento que dibuja los elementos *UI* del motor. Todo elemento de interfaz deberá ser hijo del *canvas* para poder ser dibujado durante la ejecución. Además, el *canvas* permite definir y configurar cómo se comportan estos elementos, por ejemplo puede establecer el sistema de escalado de los objetos de la interfaz según la resolución de la pantalla.

Cuenta con tres modos de *renderizado* distintos:

- **Screen Space – Overlay:** Coloca los elementos *UI* por delante de la pantalla.
- **Screen Space – Camera:** El *canvas* es colocado una distancia dada por delante de una cámara específica. Los elementos de la interfaz son *renderizados* por esa cámara y tienen en cuenta la configuración de la misma.
- **World Space:** El *canvas* se comporta como cualquier otro objeto de la escena, los elementos *UI* se dibujarán de acuerdo a su posición tridimensional en el mundo, por lo que pueden ser ocultados por otros objetos.

5.13.2 Message

Teniendo en cuenta el sistema de mensajes definido anteriormente se escogió usar un *canvas* con el modo de renderizado *World Space,* pues era preciso que el texto apareciera en el espacio físico del juego.

El componente *Message* se encarga de mostrar un mensaje especificado cuando el jugador entra en el *collider* asociado. Para ello crea una serie de elementos *UI* de texto, uno por cada letra del mensaje, y los muestra gradualmente, haciendo que parezca que el mensaje está siendo escrito sobre la marcha.

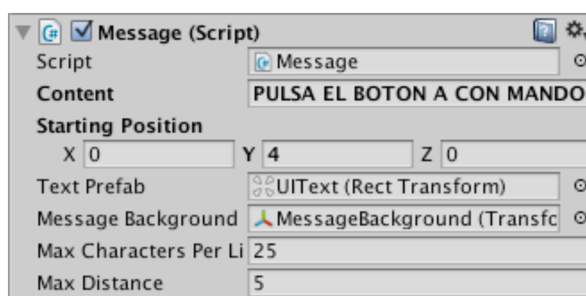


Ilustración 30. *Message* en el inspector

Además permite configurar varios parámetros desde el inspector de Unity, como el número máximo de caracteres por línea o la distancia máxima que puede alejarse el jugador antes de desaparecer el mensaje de manera automática.

5.14 Objetos rompibles

Para implementar los objetos rompibles se decidió usar un sistema muy simple: Cuando un objeto se rompe se sustituye por una versión alternativa en la que dicho objeto está dividido en trozos.

Ilustración 31. Jarrón normal y jarrón roto

BreakableObject se encarga de comprobar la velocidad de los objetos rompibles al colisionar con una superficie. Si la velocidad supera un valor específico se intercambia el objeto por la versión fragmentada y se transmite la velocidad a cada fragmento para que parezca que estos conservan la inercia del original.

Los fragmentos se autodestruyen al cabo de unos segundos gracias al componente auxiliar *SelfDestruct.*

5.15 Lógica de los escenarios y de los puzzles

Todos los escenarios que sufran cambios a lo largo del juego contarán con un *script* propio que se encargue de realizar las comprobaciones necesarias y realizar acciones en consecuencia.

Por ejemplo, si el jugador coge un objeto de la cueva y lo usa para resolver un puzle en la isla, cuando vuelva a la cueva el *script* del nivel sabrá que el puzle ha sido resuelto y ocultará ese objeto para que no pueda nunca estar duplicado en las escenas. Así, se crean tres *scripts* nuevos: *Island*, *Cave* y *Cave2*, uno para cada escenario del juego.

Los dos primeros puzles de la isla, en los que el jugador debe imitar una melodía interactuando con unas estatuas musicales para abrir las puertas que llevan a las cuevas, requieren de tres *scripts* específicos más: *Bell*, *SongStatueUp* y *SongStatueDown*.

Bell se usa en las campanas de las estatuas musicales y les permite emitir un sonido cuando el jugador interactúa con ellas. Se comunica con el *script* de la isla para comprobar si el jugador ha interpretado una melodía correctamente.

Los otros dos scripts hacen sonar, respectivamente, la melodía de las estatuas superior e inferior de la isla. Éstas son las melodías que el jugador debe imitar con las campanas. Cuando *Island* detecta que el jugador ha completado una melodía abre la puerta correspondiente.

El tercer puzle, que consiste en devolver unos orbes mágicos a sus respectivos pilares, no necesita de lógica adicional. Hace uso únicamente de los objetos que se pueden coger y lanzar y de los *Sinkhole*, ya implementados.

5.16 Expresiones del protagonista

Con el objetivo de hacer más expresivo al protagonista del juego y de dar más información al jugador respecto a las acciones que está realizando se implementaron una serie de estados para el ojo del personaje en *PlayerEyes:*

```
public enum EyeState
{
    Normal,
    HoldingObject,
    TryPickup,
    Happy
}
```

Ilustración 32. Estados del ojo en *PlayerEyes*

- **Normal:** Es el estado predeterminado del ojo, la pupila será de color negro y observará los objetos cercanos.
- **HoldingObject:** Cuando el jugador está sosteniendo un objeto la pupila se volverá de color verdoso para hacer ver que el personaje está usando magia.
- **TryPickup:** Al intentar coger un objeto la pupila se volverá verdosa rápidamente. Si no hay ningún objeto cerca que coger volverá al estado normal.
- **Happy:** Al resolver un puzle o realizar alguna acción que haga feliz al protagonista la pupila cambiará de forma durante un instante para reflejar el entusiasmo del personaje.

PlayerEyes expone el método público *SetEyeState* que actualiza el estado del ojo con aquel que recibe como parámetro.

5.17 Nubes de la isla

En la parte superior de la isla se quería que hubiera una serie de nubes desplazándose lentamente por el cielo. Para ello se ha asignado un componente muy simple a los *sprites* de las nubes llamado *SkyCloud.*

Este componente desplaza la nube con una velocidad aleatoria entre los límites especificados por el diseñador en la dirección dada hasta alcanzar unas posiciones límite, para evitar que se alejen infinitamente del escenario. Cuando una nube alcanza uno de esos límites se colocan de nuevo en el límite contrario, de manera que siempre se están reciclando las mismas nubes en un escenario.

Dado que permanecen la mayoría del tiempo fuera del campo de visión del jugador y no necesitan comportarse de manera muy precisa, los límites horizontales sólo se comprueban cada 5 segundos.

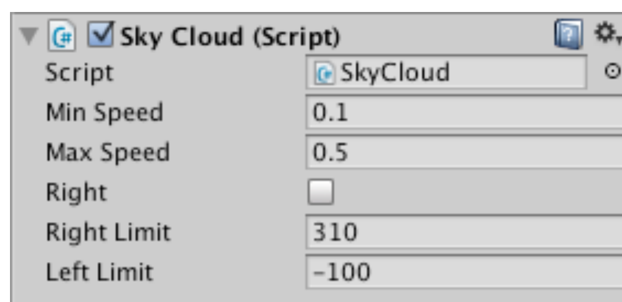


Ilustración 33. *SkyCloud* y sus parámetros

5.18 Animales

Para crear el comportamiento de los animales del juego se ha empleado la *API* de gestión de máquinas de estado de José Alapont. A continuación se explica de manera detallada cómo se ha implementado cada uno de ellos.

5.18.1 Pájaro

Como se explicó en el apartado de diseño el pájaro debía contar con dos máquinas de estados distintas, una inercial y probabilista y otra únicamente probabilista. *BirdInertial.xml* y *BirdProbabilistic.xml* implementan respectivamente dichas máquinas.

Desde la clase *Bird*, un booleano permite elegir la máquina de estados deseada desde el editor de Unity. Esta máquina se cargará al inicar la ejecución del juego. También se exponen las variables de velocidad y el radio del área de acción del animal, para que sea muy sencillo modificarlas y crear pequeñas variaciones entre animales similares.

Para mostrar el área por la que puede moverse el pájaro y facilitar el trabajo al diseñador se usa la función *OnDrawGizmos* de Unity.

```
void OnDrawGizmos()
{
    Gizmos.DrawWireSphere(transform.position, areaRadius);
}
```

Ilustración 34. *OnDrawGizmos* en *Bird*

Con *DrawWireSphere* se dibuja una esfera en la posición del pájaro con el radio de su área de acción. Esta esfera es visible desde la ventana de la escena de Unity.

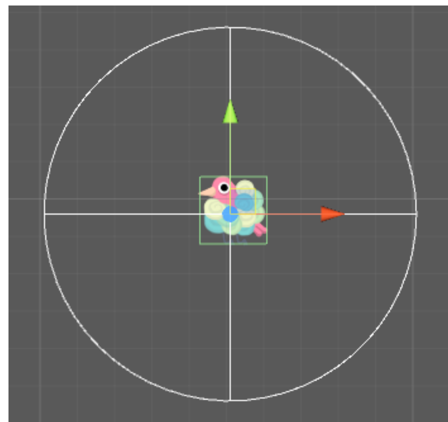


Ilustración 35. Pájaro seleccionado en la escena

La lógica del movimiento del animal es muy sencilla y se apoya fundamentalmente en la función *Wander*, que establece una posición dentro del área de acción del pájaro. El animal intentará acercarse a esa posición y, cuando lo consiga, se escogerá una nueva y se repetirá el proceso. Todos los demás animales contarán con una función similar adaptada a sus comportamientos y áreas de acción.

5.18.2 Pez

El pez necesita únicamente de una máquina determinista clásica, descrita en el archivo *FishDeterministic.xml.*

El *script* que contiene la implementación de este animal es *Fish.* Al contrario, que el pájaro, el pez puede desplazarse dentro de un área de acción rectangular, que se muestra en el editor de manera similar a la del pájaro, usando en este caso la función *DrawWireCube.*

5.18.3 Medusa

La medusa emplea una máquina con estados concurrentes descrita en *JellyfishConcurrent.xml.* Esta máquina puede ejecutar dos estados al mismo tiempo.

El comportamiento de este animal es idéntico al del pez en cuanto a movimiento se refiere y además incorpora su lógica específica para gestionar la reacción de la medusa cuando el personaje se acerca, momento en el que se pondrá a pulsar con un color.

5.18.4 Tortuga y Caracol

Estos animales cuentan con una máquina determinista clásica y otra basada en pilas, descritas en *TurtleDeterministic.xml* y *TurtleStackBased.xml.* De igual manera que con el pájaro, un booleano permite elegir la máquina que se desea usar.

Turtle contiene la implementación del comportamiento y la gestión de ambas máquinas. Dado que estas máquinas son compartidas por dos animales distintos y que *Turtle* está atado a la animación de los *sprites* de estos personajes, se ha tenido especial cuidado al crear dichas animaciones para que no haya que hacer ningún trabajo extra para usar los *sprites* del caracol o de la tortuga. Así, los dos animales cuentan con el mismo número de animaciones y se llaman de igual manera en los dos casos.

5.19 Música y efectos de sonido

Antes de incluir la música y los efectos en el juego era necesario crear un sistema que permitiera ajustar el volumen de las pistas desde el juego y las opciones. Para ello se ha hecho uso de los *Audio Mixer* de Unity. Un *Audio Mixer* permite mezclar varias fuentes de audio y aplicarles efectos. Se pueden organizar mediante una jerarquía en la que unos *mixers* están contenidos dentro de otros de manera que sean configurables a varios niveles.

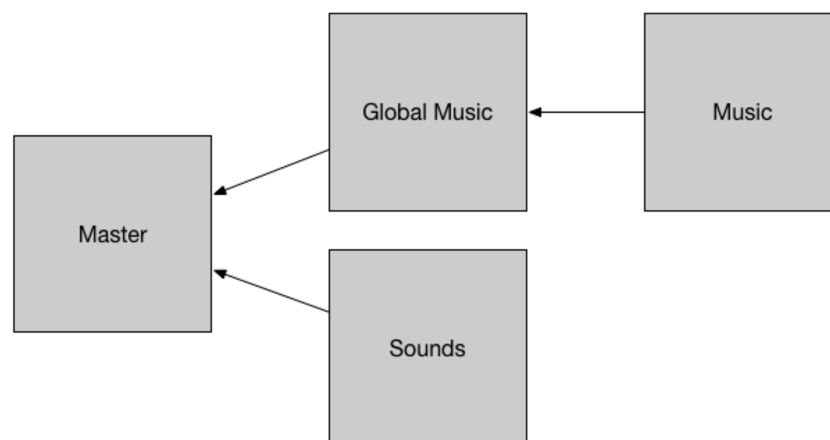


Ilustración 36. Jerarquía de *mixers*

En la ilustración anterior se observa la estructura de *mixers* del proyecto. Cada cuadrado representa un *Audio Mixer*, mientras que las flechas representan el direccionamiento de las salidas hacia otros mezcladores. Se ha decidido usar esta

estructura para poder configurar el volumen de los sonidos y la música, tanto por separado como a nivel global. Además, la música se ha dividido en dos *mixers* para que se puedan realizar ajustes de volumen desde el propio juego en los momentos que sea necesario independientemente de los ajustes del usuario.

Para gestionar las operaciones de audio de manera cómoda y sencilla se ha creado la clase *AudioManager*, que permite ajustar el volumen de los mezcladores de música y sonido, así como crear y hacer sonar nuevos efectos de sonido. Es importante saber que los *Audio Mixer* trabajan en decibelios, una unidad logarítmica, mientras que en el juego se muestra al usuario una escala lineal (de 0 a 100). *AudioManager* contiene una función que realizará la transformación a decibelios.

Para la creación de los efectos de sonido se ha empleado un *pool* de objetos. El juego podrá llegar a contar con gran cantidad de sonidos, que se estarán creando y destruyendo constantemente durante la ejecución, lo cual es indeseable debido al alto coste de la instanciación y destrucción de objetos. Se hará uso de un sistema que permite reutilizar objetos ya existentes y que sólo creara nuevos cuando no haya ninguno antiguo disponible.

Así, se ha creado el objeto *AudioObject*, un objeto muy simple que contiene un componente *AudioSource* y el *script AudioObject.* Este *script* se encargará de reproducir el efecto de sonido deseado por medio del *AudioSource* y de reciclar el objeto tan pronto como acabe de sonar para que pueda ser reutilizado con cualquier otro sonido.

AudioManager contará con una función sobrecargada llamada CreateSoundObject*,* que creará un *AudioObject* para reproducir el sonido especificado.

```
public static void CreateSoundObject(AudioClip clip)
public static void CreateSoundObject(AudioClip clip, float vol)
public static void CreateSoundObject(AudioClip clip, float vol, string objName)
public static void CreateSoundObject(AudioClip clip, float vol, string objName, Vector3 position)
```

Ilustración 37. Variantes de *CreateSoundObject*

Éstos son los parámetros que acepta la función:

- **Clip:** *AudioClip* con el sonido que se desea reproducir.
- **Vol:** Volumen del sonido.
- **ObjName:** Nombre que se dará al objeto en la jerarquía de Unity.

- **Position:** Posición del sonido. Únicamente en caso de que se especifique una posición el sonido se reproducirá como un efecto posicional situado en el espacio del juego.

Para reproducir la música simplemente se añadió un *AudioSource* al objeto que contiene el *script* con la lógica de juego, que permanecerá activo durante toda la ejecución. El *AudioSource* está configurado para que se ejecute al inicio y que reproduzca la canción en bucle.

5.20 Iluminación de escenarios

El escenario está iluminado globalmente usando la luz ambiental, que cambiará de color e intensidad según la hora del día. Todos los objetos que se desea que estén iluminados deberán usar un *shader* que soporte iluminación. En cambio, aquellos elementos que no deben ser afectados por la luz (por ejemplo los símbolos de los orbes mágicos, que deben dar la sensación de que brillan en la oscuridad), usarán un *shader Unlit,* sin iluminación.

Para resaltar zonas concretas se ha hecho uso de las *Point Light* de Unity, luces puntuales que iluminan un área circular. Además, para simular la luz que se recibe desde el cielo a través de las puertas u orificios en las paredes se ha creado el componente *BackgroundLight*, que obtiene el color medio actual del cielo y se lo asigna a una luz puntual. De esa manera, el foco emitirá siempre una luz del color del cielo.

Ilustración 38. Ejemplo de uso de una *BackgroundLight*

5.21 Menú de pausa

El menú de pausa está modelado usando de nuevo la *API* de máquinas de estados. Se usa una máquina determinista muy simple descrita en *PauseMenuDeterministic.xml,* con un estado para cada pantalla del menú y otro para cuando está deshabilitado.

Para mostrar los elementos del menú se usa otro *canvas* distinto al empleado para el sistema de mensajes, en este caso con el modo de renderizado *Screen Space-Overlay*, pues se desea que los elementos aparezcan delante de todos los objetos del escenario. Además se ha asignado al *canvas* un componente *Canvas Scaler*, que se asegura de que el menú escale correctamente para todas las resoluciones.

Al abrir el menú de pausa se debe congelar la acción del juego, para lo cual se modifica el valor de la variable estática *Time.timeScale*, que representa la escala según la cual transcurre el tiempo. Al hacerla igual a 0 se detiene el tiempo del juego. Además también se desactiva el *script* que controla el movimiento del jugador al entrar en pausa. Al desactivar el menú vuelve a igualarse a 1, el valor por defecto, y se vuelve a activar *Player.*

Para las animaciones de los elementos del menú se ha hecho uso de *DOTween*, que además permite realizar animaciones independientes del *timeScale* actual. En caso contrario los objetos no llegaría a moverse nunca cuando la escala de tiempo es igual a 0.

5.22 Animación de inicio

Al iniciar el juego se carga la escena de la isla, se muestra el título y se realiza un fundido desde negro al cabo de un instante, desapareciendo el título y otorgando el control al jugador.

Para realizar el fundido desde negro, de igual manera que entre las transiciones entre escenas, se usa un rectángulo que ocupa toda la pantalla y que está situado delante del escenario. Este rectángulo usa un material completamente negro que soporta transparencia. Desde código se modifica el valor de la transparencia durante la ejecución para realizar los fundidos.

El título aparece en un componente de texto, usando el mismo *canvas* que para el menú de pausa. Usando *DOTween* se anima la posición y la transparencia del texto para crear la animación.

5.23 Detalles finales

Una de las técnicas más comunes usada en videojuegos para dar información extra al jugador es el temblor de pantalla. En el caso de **The Traveler** se han usado temblores en varias situaciones distintas:

- Cuando el jugador se desplaza usando su transformación y choca contra una pared la pantalla tiembla ligeramente.
- Al resolver los puzles de las melodías, la pantalla tiembla durante un momento mientras suena un efecto de piedras desplazándose por el suelo

justo antes de abrirse la puerta, para así dar más importancia al momento y hacerlo más impactante.

- Al resolver el puzle de los orbes ocurre algo similar.

Para implementar este efecto se ha hecho uso de la función *DOShakePosition* de *DOTween*, que permite realizar un temblor de pantalla con una fuerza y duración específicas. En un primer lugar al realizar un temblor la cámara dejaba de seguir al jugador, temblando en el lugar en el que se había producido. Para evitar este problema y poder realizar temblores en movimiento se creó un objeto vacío, asignando la cámara del juego como hijo de dicho objeto. A la hora de realizar un temblor se ejecuta sobre el objeto padre vacío, mientras que el hijo continua siguiendo al jugador, consiguiendo el efecto deseado.

En el puzle de los orbes, además, se empleó otra técnica para dar más importancia al momento y además ocultar la sustitución de un objeto del juego por otro de manera elegante, sin que el jugador note el cambio. En este puzle hay una pequeña nube de piedra flotando sobre la plataforma central, al ser completado el puzle la nube debe perder su forma de piedra y volverse un objeto físico que el jugador podrá controlar.

De igual manera que con el fundido a negro, se creó un rectángulo situado delante del escenario con un material transparente, blanco en este caso. Al resolver el puzle se produce un temblor de pantalla y, justo al final, un destello blanco. Esto se consiguió haciendo que la transparencia del rectángulo pasara rápidamente a su valor máximo, revelando el color blanco durante un momento, y de nuevo a 0. El efecto es similar al de un *flash* de cámara. En el pequeño instante en que la pantalla es completamente blanca se sustituye la nube de piedra por la controlable por el jugador.

6. Conclusiones

En este proyecto se ha implementado un amplio prototipo jugable de un videojuego de plataformas y exploración en 2D. La creación de un videojuego conlleva una serie de obstáculos y dificultades, aparte de aquellos comunes al desarrollo de cualquier software, es una tarea difícil que requiere de conocimientos en muchos campos distintos.

A pesar de la dificultad inherente a este campo de desarrollo, el motor Unity ha facilitado notablemente el trabajo y ha permitido crear un videojuego complejo y multiplataforma en un periodo de tiempo pequeño en relación con otras tecnologías, sin sacrificar potencia ni flexibilidad.

6.1 Relación con los estudios cursados

Como se ha comentado en varias ocasiones anteriormente, un videojuego requiere de la aplicación de conocimientos muy variados y amplios de muchas ramas distintas.

Es por esto que los conocimientos adquiridos a lo largo de la carrera han sido de vital importancia para la realización de este proyecto, a continuación se enumeran algunas asignaturas que han sido especialmente útiles, de acuerdo con su campo:

Matemáticas y física: Los conocimientos de matemáticas y física son fundamentales en el desarrollo de videojuegos. Movimiento de entidades, aceleración, resolución de colisiones, *shaders*, todos tienen base en conocimientos matemáticos. En especial es imprescindible la destreza con trigonometría y álgebra. Asignaturas como *Análisis Matemático, Fundamentos Físicos de la Informática* o *Estructuras Matemáticas de la Informática* han sido de gran ayuda.

Desarrollo de software y programación: Un videojuego necesita de conocimientos no solamente de programación, sino también de diseño de software, estructuración de código, arquitectura de software, patrones de diseño. Debido a la interactividad inherente de los videojuegos y la necesidad de que estos respondan y muestren respuesta rápidamente a las acciones del usuario, la optimización es un aspecto clave que hay que tener en cuenta en cualquier proyecto. *Programación, Estructuras de Datos y Algoritmos* o *Algorítmica* enseñan programación pura, mientras que asignaturas como *Metodología y Tecnología de la Programación, Ingeniería de la Programación* o *Tecnología de Componentes, Patrones de Diseño y Generación de Código* educan al programador en el resto de disciplinas necesarias para crear código de calidad.

Autómatas y máquinas de estado: Dado el énfasis en la creación de inteligencias artificiales de este proyecto, los conocimientos adquiridos en las asignaturas *Teoría de Autómatas y Lenguajes Formales*, *Inteligencia Artificial* e *Ingeniería de Sistemas y Automática* han sido de gran ayuda.

Multimedia: La asignatura optativa *Fundamentos de Sistemas Multimedia* ha aportado conocimientos muy útiles de creación y manipulación de contenido multimedia, algo tremendamente útil en el desarrollo de videojuegos.

Estas son las asignaturas más relevantes de cara a la creación de este proyecto, pero sin duda hay muchas más que han ayudado a crear una base de conocimiento fundamental, además de preparar al estudiante y capacitarlo para seguir aprendiendo en el futuro, algo imprescindible en un entorno tan cambiante como el del desarrollo de software y videojuegos.

6.2 Conclusiones IA

En el proyecto se han usado dos sistemas distintos de gestión de máquinas de estado, uno propio para crear la máquina del protagonista y la *API* de José Alapont para crear las inteligencias de los animales. Cada uno tiene sus ventajas e inconvenientes, a continuación se presentan algunas conclusiones respecto al uso de estos sistemas:

La *API* IA soporta varios tipos de máquinas de estados distintas por defecto de manera muy sencilla, permite la definición de máquinas por medio de archivos *xml*, por lo que resulta muy fácil entender el comportamiento de una máquina con un simple vistazo a su definición. Esto resulta a la vez una ventaja y un inconveniente, por los pasos adicionales que implica la creación de una máquina con esta herramienta. En primer lugar hay que definir el archivo *xml* de la máquina, luego es necesario actualizar la clase *Tags* con los identificadores de los estados y transiciones, una tarea mecánica y tediosa. En último lugar hay que crear el *script* que se encarga de gestionar la máquina y su comportamiento.

Con el sistema propio, sin embargo, la definición de la máquina se da en el propio *script* del comportamiento, sobre la marcha, haciéndolo mucho más flexible en ese sentido. Además se pueden generar eventos y cambiar de estado desde cualquier lugar del código, mientras que la *API* IA sólo acepta los generados en la función *CheckEvents*. Esta flexibilidad hace que sea mucho más difícil comprender cómo se comporta una máquina creada con este sistema propio, siendo necesario leer todo el código y ver dónde se produce cada cambio de estado.

Para comprobar ambos sistemas a nivel de rendimiento se implementó la lógica de la tortuga dos veces, una con cada sistema y se creó una escena con un centenar de instancias. De este pequeña prueba se obtiene que ambos sistemas

resultan similares a nivel de rendimiento, la complejidad viene dada en última instancia por la propia lógica del comportamiento implementado más que por el sistema subyacente empleado.

En definitiva pienso que la *API* IA resulta muy útil y es recomendable para la definición de máquinas simples y reducidas, con comportamientos sencillos y permite realizar cambios en la máquina de estados fácilmente. Es, además, especialmente útil en la definición de máquinas de estados no deterministas.

El flujo de trabajo de la herramienta comienza a volverse en su contra a la hora de trabajar con máquinas más grandes y complejas, como es el caso del jugador, que necesita de cambios y revisiones constantes, así como de mucha flexibilidad en las transiciones. En esos casos resulta más adecuado utilizar un sistema alternativo como el presentado en esta memoria.

6.3 Planificación

La planificación de un proyecto de software siempre resulta complicada, con más razón aún tratándose de un videojuego, por el hecho de que está basado en la interacción con el usuario y en las sensaciones que ello transmite, algo fundamentalmente abstracto y subjetivo. El testeo es imprescindible para crear un videojuego, tanto por parte de los diseñadores y programadores como de personas no involucradas en el desarrollo, para conocer si el juego tiene el efecto deseado en los jugadores. Sin embargo, es muy difícil estimar las necesidades de tiempo de ese testeo, o si un requisito especificado en el documento de diseño no funciona en absoluto una vez implementado en el juego.

En el apartado de planificación se estimó una duración aproximada de 16 semanas para completar el proyecto, incluyendo todo el testeo y pruebas, pero la duración final fue de 20 semanas.

Notablemente, la implementación del código de gestión de colisiones así como el de movimiento del personaje ocupó la mayoría de este tiempo adicional, tareas que estaban planeadas para la primera iteración.

El movimiento del personaje por el escenario es la pieza más importante del juego y era necesario que funcionara de forma óptima y fuera lo más divertida posible. Si el bucle de interacción más básico del usuario con el juego, que ocupará la mayoría del tiempo del jugador, no es divertido, este juego no podrá funcionar nunca. No importa cuantas animaciones haya, la calidad de la música o de los gráficos, la base de **The Traveler** es la diversión que debe proporcionar el control del personaje y la exploración. Dada la complejidad del control del personaje, por lo orgánico de los escenarios, la gran cantidad de movimientos y velocidades distintas, esto ha resultado muy costoso. A lo largo del proyecto ha sido necesario

revisar y retocar partes del código del movimiento y las colisiones muchas veces, intentando mejorarlo poco a poco, puliendo detalles y eliminando los problemas que el usuario podía encontrar con los controles.

Este trabajo, por tanto, se ha ido repartiendo de forma orgánica en las distintas iteraciones del juego, que han crecido ligeramente.

Otro punto importante al que hubo que dedicar más tiempo es el *feedback*, la información que recibe el jugador y que le ayuda a entender el mundo del juego y los resultados de sus acciones en el mismo, a comprender su lógica interna y avanzar en la aventura. Este es un tema delicado sabiendo que la exploración y el descubrimiento eran piezas clave del juego, es necesario encontrar un equilibrio perfecto en el que el jugador recibe exactamente la información que necesita, ni más ni menos. Si no hay suficiente *feedback*, el jugador se siente perdido y se frustra, no siente deseo por explorar el mundo. Si recibe demasiado, por el contrario, la exploración se pierde, se puede volver demasiado artificial, el jugador puede pensar que el juego le está llevando de la mano y que no le da la opción de superar el juego por su cuenta, haciendo uso de su inteligencia y curiosidad.

Ha sido necesario revisar varias veces la información que se da al jugador acerca de los puzles, los elementos visuales que conectan los elementos con que el jugador tiene que interactuar para avanzar (orbes, símbolos, pilares, estatuas de las melodías, campanas, puertas). Se ha intentado dar siempre alguna pista que ayude al jugador a saber como resolver un puzle, pero la capacidad de observación y la curiosidad del jugador siguen siendo imprescindibles, las soluciones nunca se dan gratuitamente, el jugador debe poner de su parte.

A continuación se muestra el diagrama de Gantt con la distribución real de tiempo del proyecto tras su implementación.

	Nombre de la tarea	Duración	P2			P3			P4		
			Abr	May	Jun	Jul	Ago	Sep	Oct	Nov	Dic
1	**1ª iteración**	**27d**			1ª iteración						
2	**2ª iteración**	**18d**			2ª iteración						
3	**3ª iteración**	**16d**				3ª iteración					
4	**4ª iteración**	**26d**						4ª iteración			
5	**5ª iteración**	**24d**							5ª iteración		
6	**6ª iteración**	**19d**							6ª iteración		
7	**7ª iteración**	**11d**								7ª iteración	

Ilustración 39. Diagrama de Gantt final

7. Trabajos futuros

El juego realizado en este proyecto se concibió como un prototipo, una prueba de concepto de un posible videojuego más amplio y completo. Así sienta las bases del mismo y presenta una serie de mecánicas y una jugabilidad básicas que pueden servir de cimientos para desarrollar una experiencia jugable divertida y satisfactoria.

De esta manera, para crear un juego completo con base en las mecánicas descritas sería necesario añadir gran cantidad de escenarios nuevos, personajes no jugables, objetos interactuables, una base argumental coherente, rediseñar la interfaz y añadir menús, un sistema de guardado y carga, etc.

Ello constituye una tarea considerablemente grande y seguramente necesitaría de la colaboración de un equipo de desarrolladores, según la ambición de las ideas incluidas en el nuevo diseño del juego.

En lo que respecta a la API IA, considero que es posible realizar mejoras que hagan la herramienta más útil para el desarrollo de videojuegos de lo que lo es ahora. En especial pienso que habría que dedicar tiempo a repensar el flujo de trabajo, a buscar maneras de reducir la dificultad y el tiempo de la definición de las máquinas. Un par de posibles ampliaciones en este sentido se presentan a continuación:

- Creación de un editor visual para la creación y modificación de las máquinas de estado, que haga el proceso mucho más intuitivo y sencillo de comprender a simple vista.
- Automatizar la creación de la clase *Tags* o, por lo menos, parte de la misma, como la función *StringToTags.* Este paso constituye un proceso tedioso y repetitivo, teniendo que realizarlo con cada modificación de la máquina.

Con estas mejoras la productividad aumentaría de forma drástica y sería mucho más rápido implementar de manera satisfactoria una máquina de estados funcional.

8. Agradecimientos

A Marcos Vendrell Añó por la ayuda prestada con el diseño de juego y de escenarios, así como por el excelente trabajo en todo el apartado artístico visual, realizado enteramente por él.

A Juan María Prieto Iborra por su colaboración con la música y los efectos de sonido, que no podían ser más adecuados para el proyecto.

A Ramón Mollá, tutor del proyecto, por la libertad que me ha dado a la hora de realizar un videojuego tal como mis compañeros y yo deseábamos, ofreciendo ayuda y asistencia en todos los ámbitos. Agradezco también la oportunidad de usar la API IA de José Alapont, gracias a lo cual he adquirido nuevos conocimientos de inteligencia artificial y máquinas de estados, así como reforzado los ya existentes.

9. Bibliografía

[1] Aevi, La industria del videojuego http://goo.gl/r5XYg8

[2] Opsive, Behaviour Trees or Finite State Machines http://www.opsive.com/assets/BehaviorDesigner/documentation.php?id=49

[3] Unity, Roadmap https://unity3d.com/es/unity/roadmap

[4] DOTween, Engines Comparison http://dotween.demigiant.com/ - enginesComparison

[5] Unity, Execution Order http://docs.unity3d.com/es/current/Manual/ExecutionOrder.html

[6] Unity, Object Pooling http://goo.gl/cOikVp

10. Anexo I: Uso de *2DToolkit* y *Ferr2D*

2DToolkit

Las dos características fundamentales de *2DToolkit* son la creación de colecciones de *sprites* y la creación de animaciones. Para ambas funciones la herramienta cuenta con un editor visual muy completo y sencillo de usar.

Para crear una colección de *sprites* basta con crear una colección vacía, abrir el editor de colecciones y arrastrar todas las imágenes deseadas a la parte izquierda del editor. Desde el editor se puede modificar un gran número de ajustes globales de la colección o específicos de cada *sprite.* Por ejemplo, dentro de los ajustes globales, se puede ordenar la generación de vectores normales para que sea posible iluminar los *sprites,* cambiar el formato y el modo de filtrado de las texturas o cambiar el modo de los *colliders.*

En los ajustes del *sprite* se puede elegir el tipo de *collider,* cambiar el centro posicional o dividir el *sprite* en trozos pequeños que luego la herramienta reconstruirá en ejecución, para ahorrar espacio.

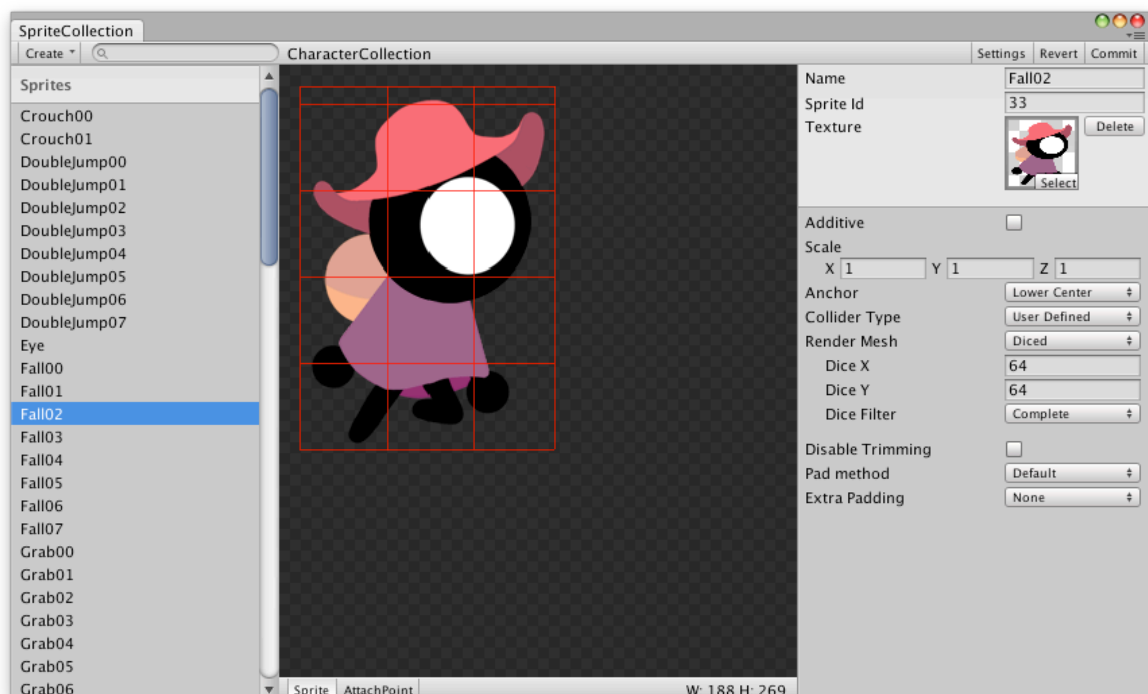


Ilustración 40. Editor de colecciones de *2DToolkit*

Gracias a las distintas opciones de personalización se consigue minimizar el tamaño de los *atlas* resultantes de manera que el motor, durante la ejecución del

juego, tenga que cargar muchas menos texturas independientes, mejorando la eficiencia.

Ilustración 41. Atlas del personaje principal

Una vez los *sprites* están contenidos en una colección es posible crear animaciones a partir de los mismos. Para ello *2DToolkit* ofrece otro editor en el que será posible establecer la secuencia de imágenes de la animación y el número de imágenes por segundo con que se reproducirá. También es posible crear *triggers*, es decir, establecer puntos específicos en la animación que luego ejecutarán eventos desde código.

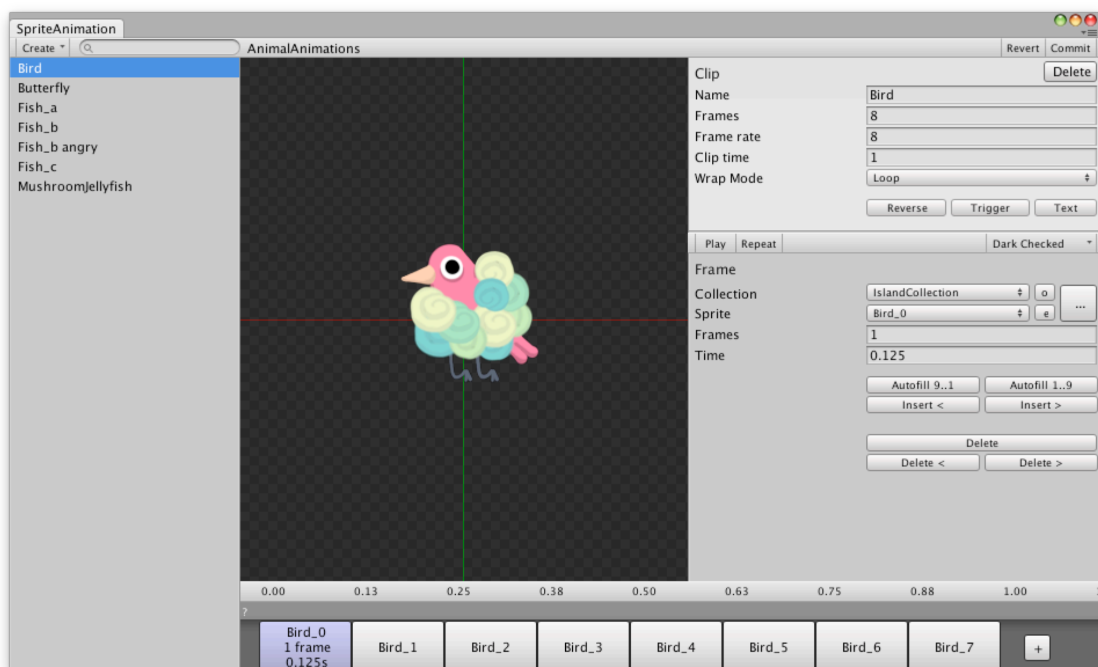


Ilustración 42. Editor de animaciones de *2DToolkit*

Ferr2D

Para crear un terreno con *Ferr2D* es necesario primero hacer uso del editor de materiales que incorpora para crear el objeto que contendrá las especificaciones con las texturas del terreno. Un material de *Ferr2D* consta de un suelo, un techo, paredes izquierda y derecha y una textura para rellenar la malla. Cada una de las paredes o suelo puede contar con tantas texturas intercambiables para el centro como se quiera, para dar variedad al terreno, así como de, opcionalmente, texturas para las esquinas.

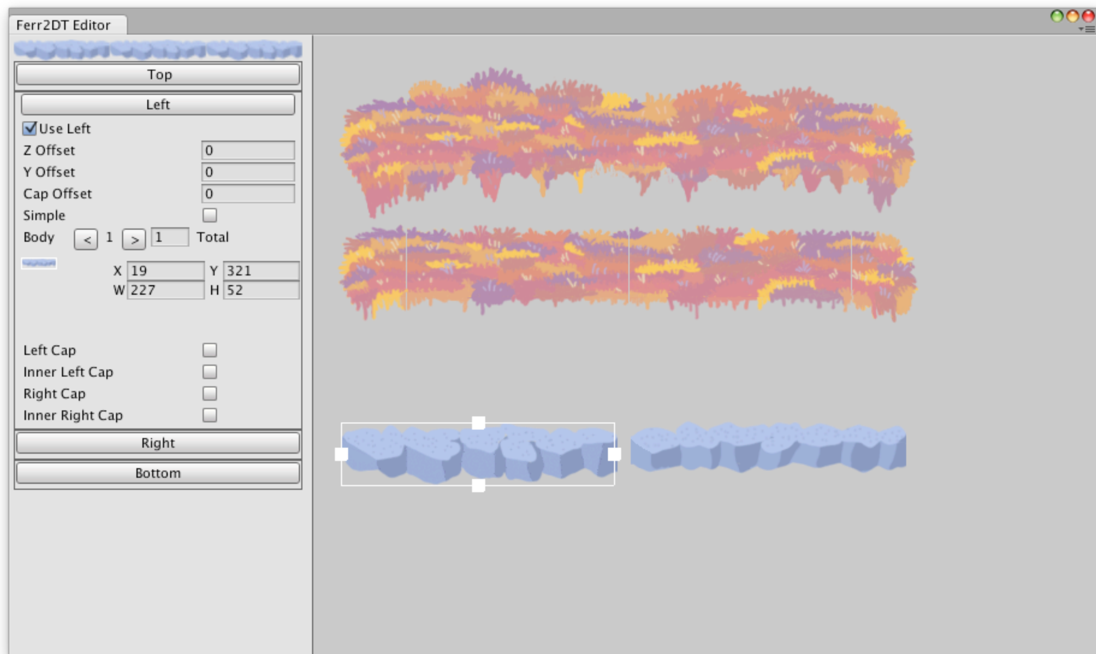


Ilustración 43. Editor de materiales de *Ferr2D*

Una vez creado un material es posible generar un nuevo terreno desde el editor de escenas de Unity con dicho material. Al seleccionar un nuevo terreno se observa que está formado por una serie de vértices, que se pueden arrastrar para modificar la apariencia del terreno. También se pueden crear tantos vértices nuevos como se desee. Desde el inspector se pueden modificar todas las propiedades del terreno, para así ajustar la apariencia y el comportamiento del mismo.

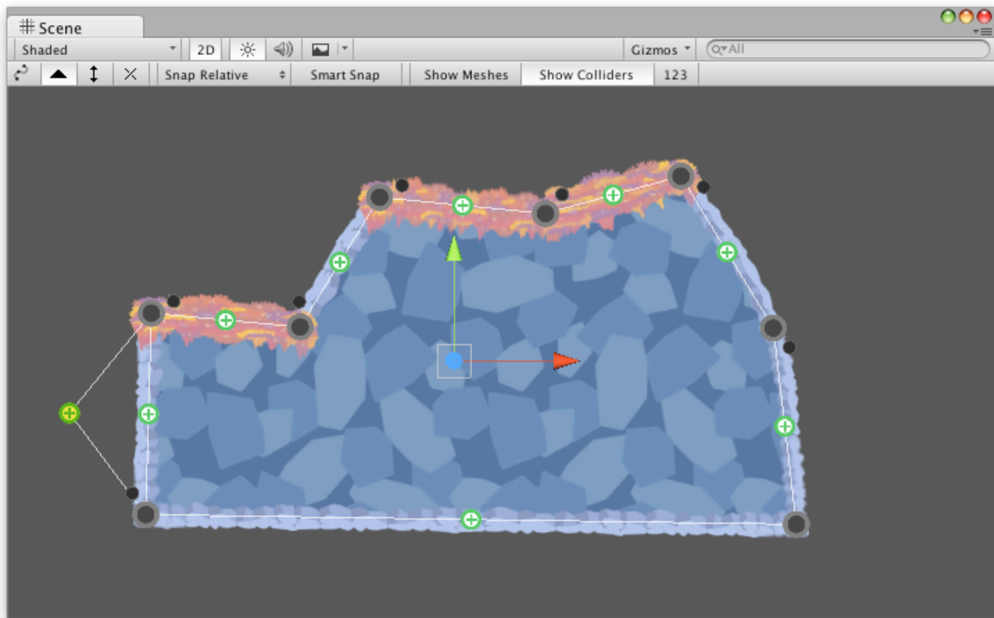


Ilustración 44. Edición del terreno desde la ventana *Scene*

11. Anexo II: Documento de diseño

VISIÓN GENERAL DEL JUEGO

DESCRIPCIÓN GENERAL

The Traveler es un juego de plataformas y exploración 2D con un claro énfasis en la exploración y la rejugabilidad. El jugador controla a una criatura que, justo después de su cumpleaños, se marcha de casa con la intención de conocer, explorar y asentarse en un nuevo lugar, con un deseo de ser útil para la comunidad.

El juego contiene un arco argumental principal y una progresión a lo largo de los dos mundos del mismo, así como una gran cantidad de actividades secundarias y secretos ocultos. Este arco forma la columna vertebral del juego, pero el contenido y la diversión se encuentran realmente en el resto de actividades que se podrán realizar durante la aventura e incluso después de haber completado esa historia principal.

Las secciones de plataformas pondrán a prueba al jugador y requerirán de gran destreza y habilidad en el control del personaje, mientras que para resolver los puzles ambientales que se encontrarán repartidos por el escenario será esencial la exploración y la atención al entorno.

GAMEPLAY

Para moverse por los vastos escenarios que forman el juego el jugador contará con varias opciones de movimiento y un sistema de control fácil de aprender pero difícil de dominar a la perfección.

El protagonista podrá moverse rápidamente tanto por tierra como por agua y aprovecharse de ciertos elementos del escenario para impulsarse y acceder a lugares nuevos. También puede coger y lanzar los objetos que se encuentren desperdigados por el escenario. Algunos de ellos modifican ligeramente la jugabilidad o propiedades físicas del personaje.

Será posible hablar con el resto de habitantes del mundo y también interactuar con animales, criaturas y objetos del escenario.

MENTALIDAD

El juego pretende despertar en el jugador un interés por explorar y entender el mundo en el que se encuentra. No hay un sentido de urgencia ni riesgos que pongan en peligro la vida del personaje. Se busca evocar una sensación de calma y tranquilidad, que la voluntad de experimentar todo lo que ofrece el juego nazca del propio jugador y no se sienta impuesta por retos artificiales. En definitiva, que el jugador se sienta inmerso completamente en el universo del juego y parte fundamental del mismo.

ASPECTOS CLAVE

- El juego tiene una historia principal y un gran número de actividades secundarias y secretos.

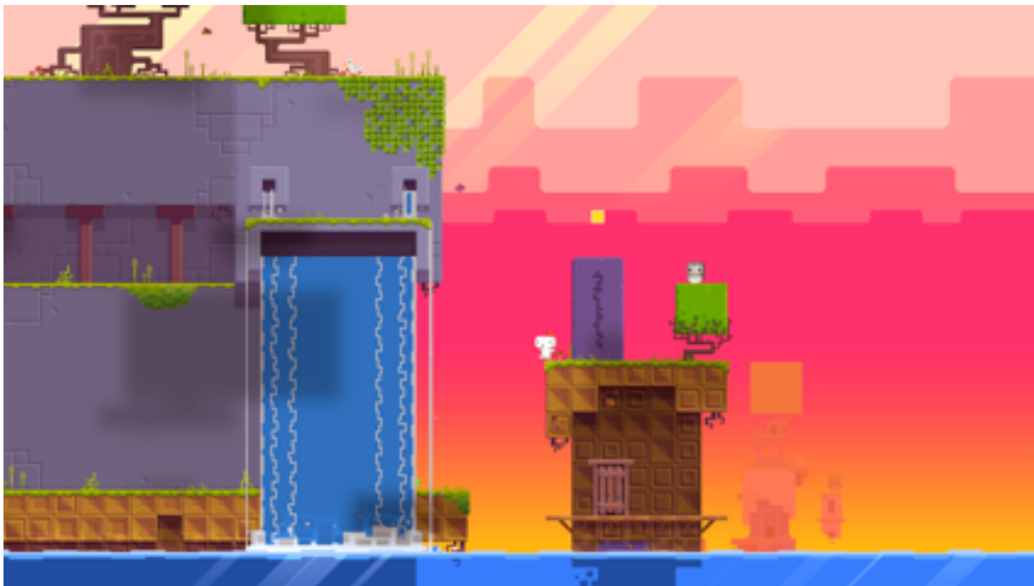
- Muchos escenarios que evolucionarán y se transformarán con el paso del jugador.
- Exploración, plataformas y puzles son piezas fundamentales del juego y se unen para crear una experiencia única.
- El jugador puede interactuar con otros personajes, animales y objetos.
- Los objetos del escenario se pueden coger y lanzar, pueden funcionar como elementos de puzle, modificar el comportamiento del personaje, etc.
- El jugador podrá recibir dinero vendiendo objetos o ayudando a los habitantes del mundo que luego podrá utilizar para comprar otros ítems o incluso adquirir una casa.

REFERENCIAS

Los siguientes videojuegos han servido de inspiración para el proyecto:

FEZ

Juego de plataformas, puzles y exploración con una estética muy particular. El aprendizaje guiado se limita al principio del juego y el jugador queda libre para experimentar y explorar. Tiene muchos puzzles abiertos e imaginativos, alejándose de los juegos de puzzles clásicos centrados alrededor de una mecánica concreta.



ANIMAL CROSSING

El jugador controla a un niño que acaba de llegar a un pueblo nuevo y debe realizar distintas actividades para ganarse la vida y terminar pagando la hipoteca de su casa. Este juego ha servido de inspiración por la gran cantidad de posibilidades de entretenimiento que ofrece.

SUPER MARIO 64

El juego que estableció un nuevo arquetipo para los juegos 3D, como **Super Mario Bros** hizo para los juegos de plataformas bidimensionales. En vez de niveles únicos lineales se presentan grandes escenarios rejugables con muchos retos distintos y se eliminan los límites de tiempo de juegos anteriores. De esta manera, se da mucha importancia a la exploración y el descubrimiento.

Por otra parte, las películas del estudio de animación **Studio Ghibli** han servido de inspiración por su estética y capacidad de crear mundos únicos y mágicos con unas reglas y lógica propias y específicas para cada mundo.

CONTROLES

El jugador puede:

- Moverse lateralmente
- Saltar (salto simple, doble, triple, salto largo, salto lateral, salto pared…)
- Interactuar
- Coger y lanzar objetos
- Desplazarse rodando
- Mover la cámara para observar los alrededores del personaje

El juego está pensado principalmente para ser jugado con un mando, pero se puede jugar sin problemas también con un teclado. De primeras sólo está soportado el mando de Xbox 360. En el futuro habría que tener en cuenta distintos mandos para

soportarlos nativamente o permitir el remapeado de botones desde las opciones del juego.

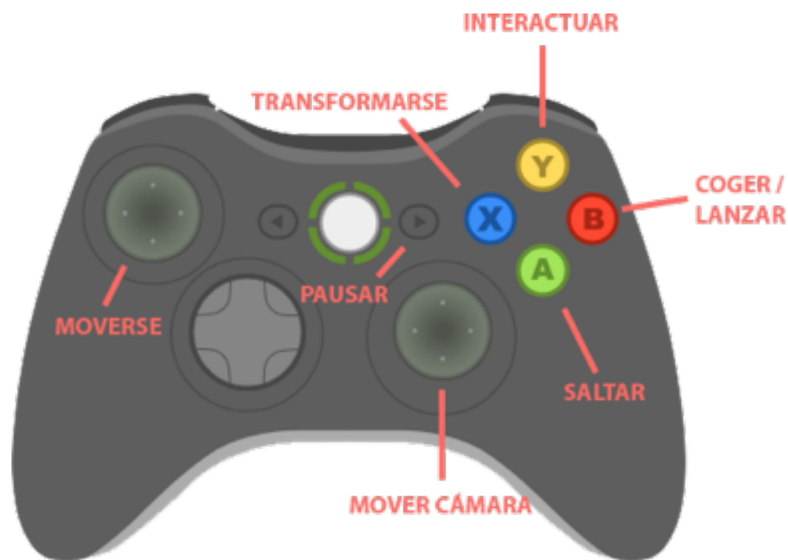


Ilustración 45. Controles con el mando de Xbox 360

La variedad a la hora de desplazarse por los escenarios es uno de los puntos clave del juego. Los saltos y la transformación del personaje pretenden complementarse para hacer que el simple hecho de desplazarse por el mundo sea divertido, profundo y variado. Haciendo uso de todos los sistemas y mecánicas disponibles el jugador se podrá mover por tierra, agua y aire de manera eficiente y rápida.

Además, una vez el jugador consiga algo de maestría con los controles podrá descubrir nuevos caminos y atajos que antes no parecían posibles, fomentando la rejugabilidad y haciendo de la mecánica más básica del movimiento algo divertido y duradero a lo largo del juego.

EL MUNDO DEL JUEGO

DESCRIPCIÓN GENERAL

El mundo en el que tiene lugar el juego pretende ser cotidiano pero con toques mágicos y reglas propias. Estará dividido en varios escenarios distintos, que contendrán lugares diversos como colinas y acantilados al borde del mar, pequeñas islas, cuevas, grandes picos, praderas, lagos y pequeños poblados. En general el ambiente será rural y campestre, de acuerdo con la filosofía del juego.

EL MUNDO FÍSICO

Viajes

Los escenarios del juego deben dar la sensación de formar parte de un único mundo grande, pero estarán divididos para facilitar la creación de los mismos. La forma de alcanzar dichos lugares no siempre será obvia o accesible de primeras, puede que sea necesario resolver algún puzle o interactuar con determinado objeto antes de poder acceder a una zona o escenario.

Aparte de las opciones de movimiento propias del personaje para desplazarse por los escenarios existirán otros elementos que faciliten el movimiento o sirvan de atajo entre partes del mundo muy alejadas entre sí (por ejemplo un tranvía para desplazarse por la ciudad).

Biología

Dentro del mundo del juego habrá gran cantidad de animales y plantas. Los insectos y pájaros podrán ser capturados por el jugador con un cazamariposas, y también existirá la posibilidad de pescar distintos tipos de peces con una caña, así como recolectar plantas.

Clima

En cada escenario puede haber un clima distinto, o incluso variar dinámicamente. Esto puede servir para cambiar las criaturas o plantas que aparecen en un escenario según las condiciones actuales del tiempo.

Día y noche

El juego cuenta con un ciclo de día y noche sin transiciones. El color e intensidad de la luz ambiental se ajusta de acuerdo con la hora del día para cambiar el ambiente de los escenarios. El ciclo no transcurrirá a tiempo real. Puede que en el futuro haya que limitar este sistema o hacerlo más restrictivo debido a su complejidad por los diferentes comportamientos que los NPCs puedan tener según la hora actual y la dificultad de realizar las transiciones entre esos comportamientos.

OBJETOS ROMPIBLES

El jugador puede coger y lanzar algunos objetos del mundo. Esto puede servir para distintos fines dentro del mundo del juego, por ejemplo llevar un objeto a un NPC, romper una vasija para descubrir algo en su interior, etc.

Esto último es posible dado que habrá objetos que se podrán romper al lanzarlos a gran velocidad contra una superficie.

TERRENO

Para transmitir lo que queremos el terreno del juego debe ser orgánico y variado, por lo que no nos basta con un sistema típico de tiles o un número arbitrario de rampas con distintas inclinaciones. Buscamos un terreno que soporte cualquier tipo de inclinación y reajuste sus texturas automáticamente para que el resultado sea siempre visualmente aceptable, sin texturas excesivamente estiradas ni cortes.

SISTEMA DE RENDERIZADO

DESCRIPCIÓN GENERAL

El juego utiliza una proyección ortográfica con una cámara situada en el eje Z apuntando a la parte positiva del mismo.

RENDERIZADO 2D/3D

Unity, el motor usado en el juego, es un motor de renderizado 3D, pero usando la proyección ortográfica y orientando la cámara hacia los ejes x e y se limita a esos dos ejes sin percibir profundidad y se puede usar como un motor 2D.

CÁMARA

DESCRIPCIÓN GENERAL

La cámara, por lo general, seguirá al personaje principal, permaneciendo este siempre cerca del centro de la pantalla.

DETALLE DE CÁMARA #1

Al moverse el personaje la cámara lo seguirá suavemente con cierto retardo. De esta manera se evita que el personaje esté siempre centrado y el movimiento resulta más fluido.

DETALLE DE CÁMARA #2

Existirá una “ventana” de la cual nunca saldrá la cámara mientras está siguiendo al jugador para evitar que este pueda quedar fuera del encuadre en algunos momentos, por ejemplo cuando está desplazándose a mucha velocidad.

DETALLE DE CÁMARA #3

En ciertos momentos la cámara se moverá automáticamente sin interacción por parte del usuario para revelar o dar más importancia a ciertos elementos del escenario. El zoom también se ajustará automáticamente en algunos casos para crear momentos visualmente atractivos o mostrar varios objetos importantes en pantalla.

DETALLE DE CÁMARA #4

El usuario podrá controlar manualmente la cámara para revelar los alrededores del personaje. Una vez deja de pulsar los botones de control de la cámara esta volverá automáticamente a seguir al jugador.

FONDOS Y PARALAJE

CIELO

Dado que el juego precisa de un ciclo dinámico de día y noche, el cielo (un cuadrado bidimensional situado en el fondo de la escena) será un degradado con un color superior e inferior, que irá variando por código para reflejar las distintas horas del día y la noche. Para ello crearemos un shader propio.

De noche aparecerán estrellas o constelaciones en el cielo por delante de esta primera capa con el gradiente.

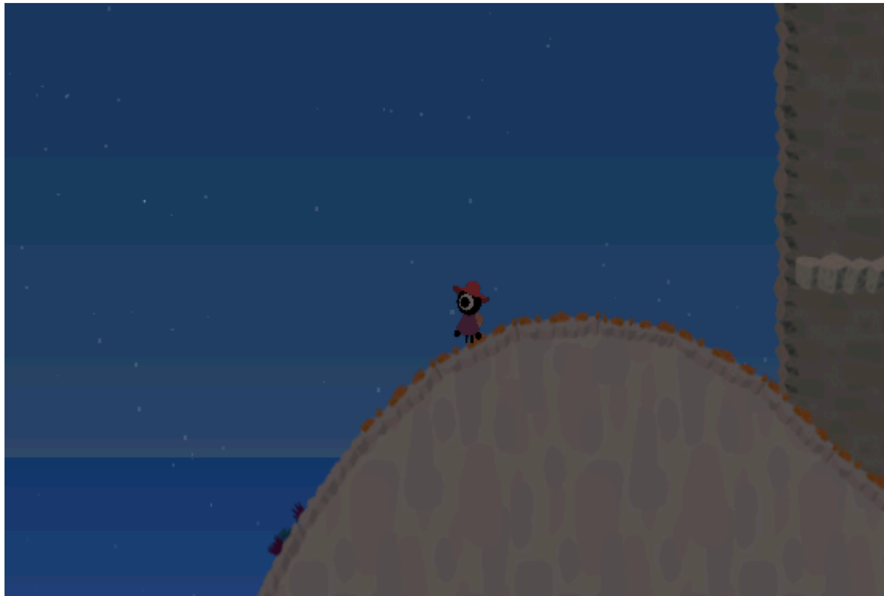


Ilustración 46. El cielo de noche

Además, en los escenarios abiertos podrán verse nubes desplazándose horizontalmente por el cielo a distintas velocidades y alturas.

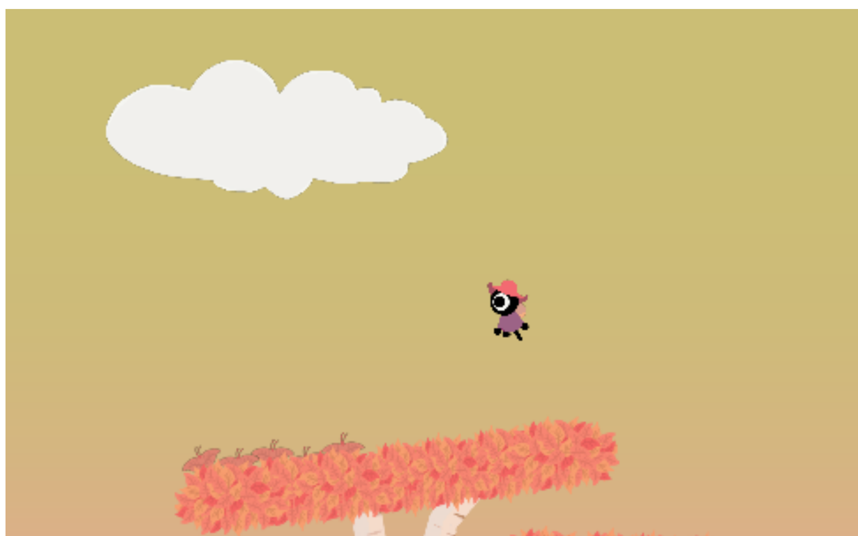


Ilustración 47. Nube desplazándose por el cielo

PARALAJE

A lo largo de las escenas habrá varias capas en distintas profundidades que añadirán detalle al escenario aunque no afecten a la jugabilidad por no tener colisión. Para dar sensación de profundidad usaremos un sistema de paralaje que, teniendo en cuenta el movimiento de la cámara desplace también esas capas de fondo, más lentamente cuanto más lejos se encuentren.

MENSAJES

En determinados puntos del mundo habrá carteles con información importante y consejos para el jugador. Cuando el protagonista se acerque a estos carteles la información se mostrará automáticamente encima de los mismos, en el propio mundo del juego, ocultándose al alejarse unos pasos. Así, los mensajes de ayuda se encuentran perfectamente integrados en el juego y evitan ser intrusivos o confusos.

VEGETACIÓN

En los escenarios habrá plantas que reaccionaran al movimiento del personaje. Así, cuando el personaje camine a través de un arbusto, este se moverá levemente. Para ello habrá que desplazar los vértices superiores del arbusto, de manera que parezca que se inclina hacia los lados.

AGUA

Debido a las implicaciones jugables que el agua tiene en este juego y ya que el personaje va a estar constantemente entrando y saliendo de la misma, el agua estará animada usando una malla de objetos, que reaccionarán a la entrada y salida de personajes y objetos en la masa de agua y propagarán olas a lo largo de la malla.

Además, el agua será semitransparente y permitirá ver los objetos que están situados detrás. Usaremos un shader para simular un efecto simple de refracción y distorsionar levemente la imagen de esos objetos que están detrás.

MOTOR DEL JUEGO

DESCRIPCIÓN GENERAL

El motor Unity se encargará de realizar el renderizado de los sprites 2D del juego, de calcular la iluminación y las físicas y colisiones de los objetos.

DETECCIÓN DE COLISIONES

Para resolver las colisiones del personaje principal y el resto de objetos no físicos (no controlados directamente por el motor de físicas de Unity) se hace uso de Raycasts teniendo en cuenta la velocidad actual del objeto para encontrar las paredes y suelo con los que pueden colisionar.

MODELOS DE ILUMINACIÓN

DESCRIPCIÓN GENERAL

El modelo de iluminación se basará principalmente en una luz ambiental, dependiente de la hora del día, que afectará a todos los objetos del escenario. Además, ciertos objetos del mundo emitirán su luz propia, por lo que se hará uso de luces puntuales en esos casos.

DISPOSICIÓN DEL MUNDO

DESCRIPCIÓN GENERAL

El mundo estará a priori dividido en tres amplias zonas, que constarán a su vez de varios escenarios.

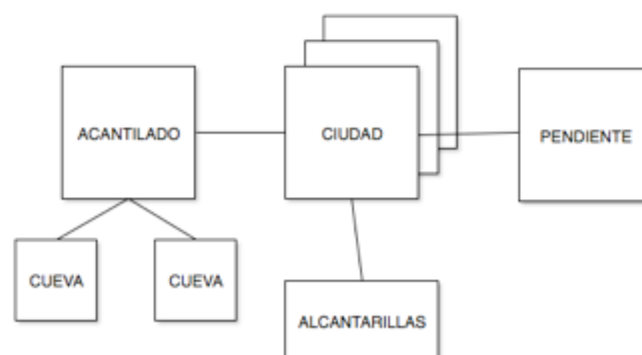


Ilustración 48. Disposición de los escenarios y zonas

En la imagen se puede ver un esquema muy general del mundo del juego. Los cuadrados representan escenarios y las flechas indican qué escenarios están conectados entre sí.
La ciudad sirve de nexo entre todas las zonas del juego y está organizada a base de

capas, esto es, distintos escenarios superpuestos y conectados entre sí que simularán la profundidad y complejidad de una ciudad tridimensional con la limitación de nuestro punto de vista bidimensional.

TRANSICIONES ENTRE ZONAS

Las transiciones entre las distintas zonas del juego podrán ser de distintos tipos:

- Puertas o entradas con las que podrá interactuar el jugador una vez se sitúe delante de ellas.
- Transiciones en los límites horizontales o verticales de la pantalla. Cuando el jugador atraviese uno de esos límites quedará fuera de la cámara y se cargará una nueva zona.
- Momentos específicos que activarán una transición. Por ejemplo el jugador podrá interactuar con una barca y se mostrará una animación con la barca alejándose del escenario actual, para cargar uno nuevo. ste

ACANTILADO

El acantilado es la primera zona del juego y debe permitir al jugador habituarse a los controles del juego y entender cuál es la dinámica que este seguirá. Hay algunos puzles ambientales simples, zonas con agua y en general es asequible moverse con los movimientos básicos del personaje. No hay apenas NPCs con los que interactuar.

Los retos serán limitados al principio pero habrá zonas y detalles para ser descubiertos una vez el jugador tenga más información sobre el mundo y más control sobre el personaje, haciendo imprescindible el backtracking en el futuro.

CIUDAD

La zona más amplia del juego, repleta de NPCs, casas y tiendas, con muchos niveles de plataformas y en mucha vida en general. Pretende servir de hub al que el jugador acudirá para recibir misiones y recados, o bien recolectar información y escuchar rumores que puedan abrir nuevas posibilidades de exploración. El jugador podrá comerciar en las tiendas y realizar todo tipo de actividades variadas.

La exploración será fundamental también en esta zona debido a la densidad de los escenarios, el ritmo será distinto al de las otras dos zonas, mucho más centradas en la naturaleza y los espacios abiertos.

TERCERA ZONA

La última zona del juego está pendiente de diseñar, pero hay que tener en cuenta que, debido a la premisa del juego, no necesariamente deberá ser más compleja o difícil que las anteriores, como sería común en un juego lineal organizado por niveles. Las tres zonas deben empujar al jugador a moverse y explorar, volviendo constantemente a visitar las zonas anteriores. Deben ser un cajón de arena con muchas posibilidades no necesariamente lineales a nivel argumental y dar más bien la sensación de ser un mundo único a través del cual el jugador se puede mover para ir descubriendo nuevas cosas.

PERSONAJES

FILOSOFÍA Y PREMISAS

Los personajes del juego pretenden ser muy variados. En el mundo de nuestro juego pueden convivir seres mágicos con humanoides típicos, así como animales imaginarios con otros presentes en nuestro mundo. Esto es una constante a lo largo de los diferentes apartados del juego y de su historia.

PROTAGONISTA

DESCRIPCIÓN GENERAL

El protagonista de nuestro juego es un ser peculiar de raza desconocida, con un único ojo y un cuerpo completamente negro. El personaje hace uso de una magia para realizar saltos y movimientos complicados, y se puede transformar en una bola para desplazarse más rápido y flotar en el agua. Puede hacer levitar objetos en el aire e interactuar con el entorno por medio de una onda mágica.

ANIMACIÓN

El personaje está hecho a partir de una serie de imágenes y se emplea animación tradicional a base de fotogramas para dar la sensación de movimiento.

Arriba se puede observar un fotograma de la animación de salto. El ojo se dibuja por separado para poder desplazarlo durante la ejecución, como se explica a continuación.

OJO

Para facilitar la inmersión, dar más vida al protagonista y aportar feedback al jugador, el personaje reaccionará por medio de su ojo.

Cuando un objeto importante interactuable o npc se encuentre cerca del jugador, éste mirará hacia el objeto.

Tras determinadas acciones (el jugador resuelve un puzle, habla con un npc, etc) el personaje reaccionará mostrando distintas expresiones mediante el ojo.

Al coger un objeto el ojo cambiará de color para mostrar que el personaje está usando su magia para hacer flotar dicho objeto.

Además el personaje parpadeará regularmente.

ONDA DE INTERACCIÓN

El jugador podrá, en cualquier momento, pulsar un botón para interactuar con el entorno. Al interactuar se mostrará una onda de color verdoso, que se expanderá y desaparecerá rápidamente. De esta manera se interactuará con todos los objetos que se encuentren dentro del radio de la onda.

Al interactuar con un objeto que se puede coger y lanzar este realizará una pequeña animación para hacer saber al jugador que es susceptible de ser cogido.

NPCs

DESCRIPCIÓN GENERAL

Para los personajes no jugables se empleará, en principio, animación tradicional, de igual manera que con el protagonista. Sin embargo, podría ser útil utilizar animación vectorial esqueletal para algunos de estos personajes, especialmente aquellos que sólo necesitan de una animación simple de caminado y estático. De esta manera, se podría diseñar cada animación una única vez y crear personajes enteros cambiando únicamente las piezas base que forman la animación. Es decir, sólo habría que dibujar cada personaje una única vez.

COMPORTAMIENTO

Los personajes no jugables deberán desplazarse por el mundo y podrán reaccionar a distintos eventos (e.g. proximidad del jugador). Cada personaje o animal podrá tener un comportamiento específico, que se describirá usando máquinas de estados.

DETALLES DE PERSONAJES

PÁJARO

El pájaro se desplazará volando por el aire, variando su velocidad aleatoriamente para darle algo de vida y hacerlo un poco impredecible. De vez en cuando podrá posarse sobre una superficie y reposar unos instantes, para luego volver a iniciar el vuelo.

Si el jugador se acerca demasiado intentará huir y alejarse del mismo, una vez se encuentre a una distancia razonable del mismo retornará a su comportamiento habitual.

MARIPOSA

La mariposa estará reposando en una superficie indefinidamente hasta que el jugador se acerque lo suficiente, momento en el que comenzará a volar en dirección contraria al jugador.

PECES

Los peces se comportan todos de manera similar pero cuentan con tres sprites distintos para dar variedad.

Se desplazarán lentamente por el agua, de manera aleatoria dentro de un área rectangular especificada. Cuando el jugador se acerque e interactúe con ellos los peces se moverán agitadamente durante unos instantes, con una velocidad muy superior a la habitual. Al cabo de unos segundos se calmarán y retomarán su comportamiento estándar.

PEZ 1

PEZ 2

PEZ 3

MEDUSA

La medusa se desplazará de manera idéntica al pez y además cambiará de color cuando el jugador se acerca.

CARACOL Y TORTUGA

Estos dos animales se desplazarán por el suelo lentamente, eligiendo una nueva dirección aleatoria cada cierto tiempo. Si el jugador se acerca intentarán huir en dirección contraria, siempre que exista una superficie por la que puedan moverse. Después de un cierto tiempo necesitarán dormir, encerrándose dentro de sus conchas. Mientras los animales duermen no se darán cuenta de si el jugador está cerca y no podrán huir. Al cabo de un tiempo despertarán y comenzarán de nuevo a desplazarse por el suelo.

CARACOL

TORTUGA

INTERFAZ DE USUARIO

DESCRIPCIÓN GENERAL

Se pretende reducir la interfaz al mínimo y hacer aquellos elementos necesarios minimalistas y sobrios. Todo aquella información que pueda estar integrada en el juego deberá estarlo a menos que deteriore la calidad del feedback que el jugador recibe a cambio. Así, por ejemplo, una lista de misiones que podría estar en un menú, puede estar integrada en un objeto físico del juego, por ejemplo un diario o un mural en una casa del juego.

MENÚ PRINCIPAL

MENÚ DE PAUSA

Al acceder al menú de pausa desde la partida se congelará el juego, se oscurecerá la imagen y se mostrará por delante un menú simple con las opciones generales, que llevarán a más submenús.

Ilustración 49. Menú principal de pausa

La primera opción, “REANUDAR” permite salir del menú de pausa y volver al juego. “SONIDO” muestra un nuevo submenú que permite ajustar el volumen por separado de la música y los efectos de sonido. Pulsando “VOLVER” se muestra de nuevo el menú principal de pausa.

Ilustración 50. Menú sonido

De la misma manera, pulsando “VIDEO” llegamos a un nuevo menú que permite ajustar la resolución, cambiar la modalidad de la ventana y probablemente ajustar algunas opciones gráficas.

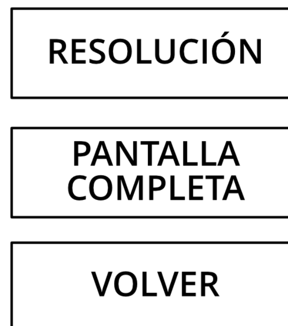


Ilustración 51. Menú vídeo

“CRÉDITOS” muestra un menú simple con los créditos del juego.

Y “SALIR” finaliza la ejecución del juego y nos devuelve al escritorio.

El menú será muy sencillo y minimalista, utilizando simplemente texto básico en color blanco sobre un fondo negro semitransparente. La opción seleccionada se distinguirá con dos guiones a los lados.

Se verá así en el juego:

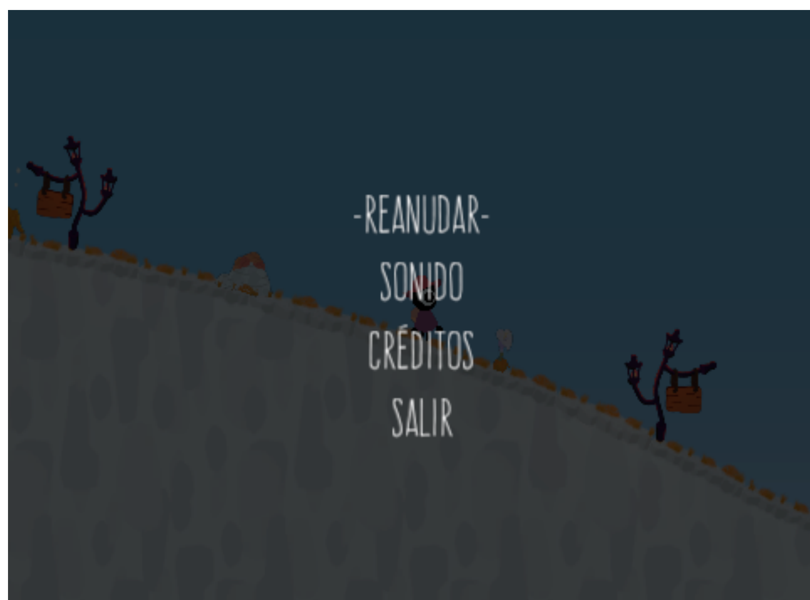


Ilustración 52. Menú de pausa *ingame*

INTERFAZ DURANTE LA PARTIDA

Durante el juego normal se pretende no mostrar ningún elemento de interfaz en pantalla. Dado que no hay vida no hace falta mostrar ningún contador de salud y los controles son lo suficientemente simples para no necesitar mostrarlos constantemente en pantalla.

Aquella información necesaria para el jugador se intentará mostrar sólo cuando sea pertinente, por ejemplo si el jugador acude a una tienda y desea comprar algo, sólo entonces se mostrará el dinero de que dispone. O, al conversar con un NPC, aparecerá un botón pequeño que indique cual es el control para avanzar en la conversación.

MÚSICA Y EFECTOS DE SONIDO

MÚSICA

Para la música del juego se buscan bucles relajados, que inviten a la aventura y a la exploración sin llegar a ser cargantes, pues estarán sonando constantemente de fondo y no deben distraer, sino acompañar. Por lo tanto, se desea música ambiental ligera, muy instrumental y no demasiado guiada por la melodía.

La música podrá cambiar según la hora del día para, por ejemplo, tener un aire más misterioso por la noche. También puede llegar a desvanecerse completamente y quedar en silencio, depositando toda la carga en los efectos de sonido en algunos momentos puntuales.

Unity 5 cuenta con un sistema de mixers que facilita la mezcla de diferentes pistas de sonido dinámicamente. Así, se podría almacenar diferentes pistas de una canción según la instrumentación y gestionarlas durante el juego para cambiar el ambiente que buscamos transmitir.

EFECTOS DE SONIDO

El juego necesita de muchos efectos de sonido variados, unos realistas y ambientales para completar los escenarios y hacerlo más vivo y creíble (sonidos de agua, viento, pájaros, etc…) y otros más específicos y particulares para dar feedback al jugador respecto a sus acciones (sonidos de interfaz, melodía al resolver o fallar un puzzle).

Habrá dos tipos de sonido distintos, aquellos posicionados en el espacio físico del juego, que sonarán con más intensidad cuanto más cerca esté la cámara, y otros globales que sonarán siempre con su máxima intensidad.

Usaremos efectos de sonido libres obtenidos de bancos de sonidos y sitios web y también algunos efectos propios.

AVENTURA DE UN JUGADOR

DESCRIPCIÓN GENERAL

El juego cuenta con un único modo, la aventura de un jugador.

La aventura comienza en la zona del acantilado, con una serie de misiones sencillas para habituarse con los controles, pero el jugador podrá desplazarse libremente por todo el escenario. Sin embargo, no será posible acceder al escenario central de la ciudad hasta superar esa serie de misiones, por lo que ésta primera parte de la aventura es la más lineal y guiada.

Una vez el jugador alcanza la ciudad todos los escenarios serán accesibles y el juego adquiere una dinámica mucho más abierta y libre. El jugador podrá ir completando misiones en las distintas zonas, realizar recados y participar en muchas actividades secundarias. Así, podrá por ejemplo capturar un pez que sólo aparece en una zona concreta a una hora del día específica gracias a la información que ha obtenido de un habitante de la ciudad. O realizar una carrera a lo largo de la ciudad con otro de los habitantes. Recolectar plantas para ayudar a un tendero y buscar secretos ocultos por todos los escenarios.

CONDICIONES DE VICTORIA

No existen condiciones de victoria en el juego para completar la aventura. Aunque hay una línea argumental principal el juego no termina ahí. Una vez completadas las misiones principales el jugador podrá seguir explorando el mundo y participando en el resto de actividades que ofrecen los escenarios.

HORAS DE GAMEPLAY

Es muy difícil estimar las horas de juego debido a la naturaleza abierta y libre del mismo. La historia principal se espera que dure alrededor de 4 horas, pero dado que no es necesario completarla de manera lineal podría alargarse mucho más.

Una particularidad de este tipo de juego es que, una vez están creados los escenarios, es muy sencillo añadir más contenido y posibilidades (por ejemplo más peces para ser pescados, más misiones pequeñas que no requieran de arte nuevo ni mecánicas específicas), alargando la duración del juego de manera muy económica, dotando de aún más vida a los escenarios y aumentando la densidad de los mismos.