UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

A Survival Tale: Diseño e implementación de un videojuego con alta accesibilidad

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Zhihao Zhang

Tutor: Abad Cerdá, Francisco José

Curso 2019-2020

Resumen

Este TFG forma parte del desarrollo del juego **A Survival Tale**, que es un proyecto en el que trabajan 4 personas: tres diseñadores (del Grado de Diseño y Tecnologías Creativas de la Facultad de Bellas Artes) y un programador. **A Survival Tale** es un juego de aventuras en tercera persona, donde el jugador controla al personaje para resolver puzles, que consiste en esquivar zonas peligrosas e interactuar con el entorno para abrirse camino. Uno de los objetivos del proyecto es alcanzar una alta accesibilidad, para que personas con diversos tipos de discapacidad sean capaces de interaccionar y disfrutar del juego. A modo de ejemplo, planteamos:

- Diseñar un modo de juego con únicamente dos botones, para que se pueda controlar por personas con movilidad reducida.
- Permitir a jugadores con discapacidad visual seleccionar la gama de colores de los distintos tipos de objetos.

Este TFG consiste en realizar toda la implementación de código del juego. Se centra concretamente en el diseño e implementación de un sistema de entrada para personalizar el tipo de control, un manejador de navegación que permite navegar las opciones del menú con dos botones, una máquina de estados para controlar el jugador, un manejador de nivel para gestionar la información de cada nivel, una función de guardar y cargar datos, algunas funciones de accesibilidad y unas funciones básicas del juego. El motor y el lenguaje que vamos a usar es Unity y C#.

Palabras clave: Videojuego 3D, Unity, Accesibilidad

Abstract

This dissertation is part of the development of the game **A Survival Tale**, a project developed by 4 students: 3 designers (of the Degree in Design and Creative Technologies of the Faculty of Fine Arts) and one programmer. **A Survival Tale** is a third-person adventure game, where the player needs to control the character to solve puzzles avoiding dangerous zones and interacting with the environment for progressing in the game. One of the objectives of this project is to achieve a high accessibility, allowing users with various types of disabilities to play with this game. For example:

- Design a 2-button mode for people with reduced mobility.

- Allow users with visual disability to choose the color of different types of objects.

This document focuses on the programming of the game, specially in the design and implementation of an input system for personalizing the type of control, a navigation manager to navigate menus with only 2 buttons, a finite state machine to control the player, a level manager to manage the information of each level, a save and load system, some functions to improve accessibility of the game and some basic functions of game. The engine and programming language used are Unity and C#.

Key words: 3D video game, Unity, Accessibility

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

A Survival Tale es un juego de aventuras con una vista de tercera persona. El contexto del juego es, que un conjunto de personas llega por causas desconocidas a una isla y su objetivo es sobrevivir. El primer paso para sobrevivir es explorar este nuevo lugar. El jugador puede controlar a cualquier persona del pueblo para explorar la isla. El juego solo tiene tres tipos de objetos con los que se puede interactuar, pero cada objeto puede tener diferentes usos. Por ejemplo, la caja es el primer objeto con el que va a poder interactuar el jugador. La caja se puede empujar hacia un hueco y luego el jugador puede pasar por encima para ir al siguiente punto o el jugador también puede empujar la caja encima de un botón para activar la puerta. A parte de la caja, también hay otros dos objetos interactivos: el coco y el ave. El coco es un objeto pequeño que el jugador puede agarrar y lanzar. El ave es una especie de animal que puede ayudar al jugador a coger objetos a larga distancia o activar algún interruptor alejado.

Si bien la jugabilidad del juego no es muy innovadora, ya que es parecido a cualquier juego de resolver puzles, se han implementado características de accesibilidad que no suelen estar presentes en este tipo de juegos. Primero vamos a ofrecer tres tipos de control: el teclado, el ratón y el modo de dos teclas. Cada tipo de control va a poder controlar todo el juego, incluyendo la parte de navegación por los menús. Segundo vamos a implementar dos formas para cambiar la gama del color de los objetos del juego para que personas con diferentes discapacidades pueden elegir la forma más adecuada.

1.1 Motivación

El mercado del videojuego es un mercado muy grande, y cada año está creciendo más. Junto con ello, los motores de videojuegos también están creciendo, tanto en cantidad como en potencia, y muchos de ellos con licencias gratuitas. Asimismo esto ayuda mucho a los grupos pequeños o pequeñas empresas a crear juegos. Desde mi punto de vista, pienso que este TFG es una oportunidad, porque es un proyecto de grupo y voy a poder trabajar con los alumnos que se han especializado en diseño de videojuegos. Creo que la experiencia es muy cercana a un trabajo real. **A Survival Tale** tiene la posibilidad de ser el siguiente juego que aparezca en **Steam**[1](una plataforma para vender juegos digitales), además también puedo aprender cómo hacer un videojuego.

A parte del motivo anterior, existe otro motivo por el cual he elegido este TFG, y que es la accesibilidad. Gracias a la aparición del teléfono inteligente los juegos cada vez son más populares y más cercanos a cualquier persona, por lo tanto la accesibilidad se convierte en una característica muy importante. Cuando un juego es accesible significa que tiene más clientes potenciales. A parte del beneficio comercial pienso que hay otra ventaja. Tengo un amigo desde pequeño y él tiene discapacidad de movilidad reducida, solo puede mover una mano. Le he invitado a jugar muchos juegos, pero él ha rechazado la mayoría obviamente por no poder controlar bien el juego. La otra ventaja de añadir accesibilidad al juego pienso es que podemos ofrecer más entretenimiento a este tipo de personas.

[1] https://store.steampowered.com/

1.2 Objetivos

Este TFG es la parte desarrollo del **A Survival Tale**. En consecuencia el objetivo principal es implementar las ideas de los diseñadores y aportar otras ideas. Las ideas de los diseñadores son las siguientes:

1. Un sistema de entrada que permite al jugador elegir la forma de control entre teclado, ratón y modo de dos botones. Además se tiene que poder reemplazar las teclas dentro del juego.
2. El menú del juego, incluye el menú de configuración y sus funciones, así como:
   - Cambiar el volumen por categoría.
   - Cambiar el modo de sonido, mono o estéreo.
   - Cambiar a pantalla completa o al modo ventana.
   - Cambiar tipo y tamaño de letra, incluyendo letras del menú.
   - Cambiar tipo y tamaño del cursor.
   - Activar o desactivar la animación de diferentes tipos de objetos.
   - Una función de elegir color.
   - Ajustar el tiempo mínimo entre registro.
3. Una función que permite a cambiar la gama del color de diferentes tipos de objetos.
4. Tres tipos de objetos interactivos, la caja, el coco y el ave.

Aparte de estas ideas de los diseñadores, a continuación enumero algunas de mis ideas:

1. Una función básica de cualquier juego: guardar y cargar. Los datos guardados no se deben poder modificar fácilmente por otros programas.
2. Una función de reintentar el nivel actual. Esta funcionalidad se puede conseguir recargando la escena, pero se busca una alternativa más eficiente para cargar más rápido.

3. Diseñar una forma de cargar las escenas para que las escenas estén interconectados y el jugador no tenga que esperar a la pantalla de cargar al ir a otra escena.

4. Añadir un post proceso para los jugadores que son dicromáticos[2](un tipo de daltonismo).

5. Personalizar cada tipo de control para que se pueda utilizar con cualquier dispositivo de entrada soportado.

1.3 Metodología

Debido a que es un proyecto de grupo necesitamos algún programa para colaborar. Desafortunadamente el periodo que hemos desarrollado este juego ha coincidido justo con la época de la aparición del coronavirus, por esta razón todos las comunicaciones que hemos hecho es por internet. Se ha utilizado **Slack** para la comunicación. La causa de elegir esta plataforma es que podemos crear diferentes canales. Cuando surge un problema en alguna parte concreta podemos comunicarlo en dicho canal y no molestar a otros miembros. Otra ventaja de **Slack** es que podemos responder a cualquier mensaje y la respuesta aparece debajo del mensaje y así no perdemos ninguna pregunta o consejo.

Para guardar el proyecto hemos elegido **GitHub**. **GitHub** es una plataforma que usa el control de versiones git y se puede usar para guardar los proyectos. Cada vez que implementamos una función podemos realizar un **commit** y podemos volver a cualquier commit cuando queramos. **GitHub** tiene una desventaja y es que el tamaño del repositorio no puede superar 100 GB y el tamaño de un fichero no puede superar 100 MB, pero esto no influye mucho en nuestro proyecto.

A parte de las dos plataforma anteriores también hemos usado **Google Drive** para compartir los recursos. La mayoría de recursos usados en este proyecto han sido proporcionados por los diseñadores, y otros se han obtenido de **Google Images**.

[2] https://es.wikipedia.org/wiki/Daltonismo

La metodología que hemos usado es iterativa para cada función, pasando del análisis al diseño y a la implementación. Cuando encontramos problema por cualquier estas fases volvemos a la fase inicial y repetimos los pasos.

1.4 Estructura de la memoria

Esta memoria está compuesta por 7 capítulos. Empezamos por la introducción, que explica cómo es el juego **A Survival Tale**, y cuáles son los miembros del proyecto. Seguimos con el estado del arte, capítulo dividido en tres partes. Por una parte analizamos las características de los motores 3D gratuitos, cuáles son sus ventajas y para qué tipo de juegos están orientados, y por otra parte analizamos los juegos disponibles en el mercado más cercanos al juego propuesto (por temática y tipo) y los comentarios de sus usuarios. Por último, vamos a analizar juegos donde en el componente de accesibilidad es importante.

El capítulo 3 es donde empezamos a explicar el contenido del proyecto. Vamos a analizar los requisitos y problemas que presenta cada funcionalidad que queremos implementar. En el capítulo 4 mostramos el diagrama de clases por cada bloque y explicamos su implementación. El capítulo 5 presenta las limitaciones y los resultados de la última versión del juego.

En el capítulo 6 concluimos cuáles son los objetivos que no hemos conseguido y por qué, junto con qué hemos aprendido en este proyecto. Para acabar, en el capítulo 7 se propone cómo mejorar tanto como la jugabilidad como la escalabilidad del juego.

1.5 Colaboraciones

El proyecto se ha realizado por cuatro personas. Los miembros son:

Ariel Pascual Díaz. Ariel es quien ha propuesto la idea del juego. Se encarga de diseñar la parte de interfaces usuarios y generar los recursos de imágenes y sonidos.

María Nieves Martínez. Maria se encarga de los modelos 3D del juego, incluyendo el nivel 1 del juego.

Mario Ferrer Dénia. Mario se encarga de diseñar la parte jugabilidad, la animación del personaje y el diseño de tres tipos de objetos interactivos.

Zhihao Zhang, yo me encargo de integrar los recursos al juego e implementar las ideas de los diseñadores.

CAPÍTULO 2

Estado del arte

Este capítulo está dividido en tres partes. En la primera parte vamos a mencionar algunos motores gratis que existen en la actualidad y explicar sus características, en la segunda parte nos vamos a centrar en los juegos publicados con éxito de estilo parecido al nuestro, y en la última parte vamos a analizar juegos accesibles actuales.

2.1 Motores 3D de videojuegos

En el mercado actual existen muchos motores para crear videojuegos, pero como queremos hacer un juego 3D en este apartado solo vamos a discutir los motores 3D [1].

2.1.1. Godot

Godot[1] es un motor 2D y 3D multiplataforma. Está publicado bajo la licencia MIT, lo que significa que el usuario tiene todos los derechos del juego que ha hecho con **Godot**. Es un motor fácil de utilizar, con soporte de C++, C# o VisualScript (una forma de programar sin escribir código). Aparte de estos lenguajes tiene un lenguaje propio que es **GDScript** [2](Figura 2.1). Su sintaxis es parecida al Python no tipado, es decir, el usuario no tiene que preocuparse del tipo de variable. La característica principal de **Godot** es el sistema de nodos[3].

[1] https://godotengine.org/

```
1  extends Control
2
3  @export_range(0, 100, 1, "or_greater") var number: int
4
5  var _backing: int = 0
6  var property:
7      get:
8          return _backing + 1000
9      set(value):
10         _backing = value - 1000
11
12 func _ready():
13     super()
14     await $Button.button_down
15     $Label.text = "After first await"
16     $Label.text = await coroutine()
17
18 func coroutine():
19     await $Button.button_down
20     return "After second await"
21
```

Figura 2.1: GDScript

Un juego hecho con **Godot** está compuesto por nodos, donde un nodo puede ser una función, un objeto o cualquier recurso. El motor lleva cientos de nodos implementados para facilitar el desarrollo del videojuego.

La otra característica atractiva de **Godot** es que permite modificar el juego en ejecución, incluso cuando se está ejecutando en un dispositivo móvil. Asimismo permite compilar el juego para diferentes plataformas, PC, Página Web, móvil o consola.

La versión más actualizada es 3.2 pero a mediados de 2020 va a sacar la versión 4.0 [4]. Esta nueva versión añade un motor de renderizado Vulkan[2] (un api de gráfico que mejora la eficiencia del uso de CPU y GPU), también va a añadir algunas funcionalidades sobre la luz. Estas actualizaciones van a permitir a Godot competir con otros motores a nivel gráfico, la figura 2.2 es una demostración de la iluminación y la sombra de **Godot**.

2.1.2. Armory

Armory[3] es un plugin de **Blender**[4]. Para comprender mejor la ventaja de **Armory**vamos a explicar qué es **Blender**. **Blender** es un programa para crear mo-

[2] https://es.wikipedia.org/wiki/Vulkan
[3] https://armory3d.org/
[4] https://www.blender.org/

Figura 2.2: Iluminación global y reflejos

delos 3D. La primera versión gratis de blender fue publicada en el año 2002 bajo la licencia GPL[5]. Gracias a esa licencia **Blender** ha ido siendo mejorado por los equipos voluntarios de diversos lugares del mundo hasta hoy. **Blender** es un programa de modelado muy potente. No solo puede hacer modelos para los juegos, también puede crear modelos para las animaciones. En 2019 ha publicado un corto de animación maravilloso, que se llama **Spring**[6](Figura 2.3). El proyecto y los recursos de este corto también se ha publicado para usarse de forma gratuita. Los modelos y las animaciones de este TFG también está creado con **Blender**.

Figura 2.3: Spring

Armory es un plugin de código abierto, que permite al usuario crear juego en el editor de **Blender**. No es necesario realizar ningún paso de importar o exportar. Tiene un sistema lógico de nodos. Este sistema ayuda a los usuarios crear juego

[5] https://es.wikipedia.org/wiki/GNU_General_Public_License

[6] https://www.youtube.com/watch?v=WhWc3b3KhnY

sin escribir código. Cuando no existe el nodo que necesitas también puedes crear un nuevo nodo con el lenguaje Haxe. Haxe es un lenguaje de programación de alto nivel y multiplataforma. El código fuente de Haxe se puede traducir a otros lenguajes como JavaScript, C++, C#.... La desventaja de **Armory** es que es un motor bastante nuevo, no tiene muchas funciones y es posible encontrar bugs. A pesar de esto **Armory** sigue siendo un motor bueno para principiantes que tienen conocimiento de **Blender**.

2.1.3. CryEngine

CryEngine[7] era un motor comercial creado por Crytek, aunque había ofrecido la versión gratis para algunas escuelas pero generalmente para usar este motor había que pagar una cantidad de dinero. Pero por el crecimiento del mercado de los motores gratis como **Unity** y **Unreal**, Crytek decidió publicar la versión 5.0[8] junto con el código bajo la licencia pay what you want, es decir el usuario decide cuánto paga, pero dos años después, en 2018 han cambiado licencia por modelo de reparto de ingresos[9].

La característica principal [5] de **CryEngine** es su fotorrealismo gráfico. Las figuras 2.4 y 2.5 muestran dos juegos que están hechos con **CryEngine**. **CryEngine** controla muy bien el mundo abierto o escena con muchos detalles. El usuario no necesita crear muchos materiales por su cuenta para tener unos gráficos decentes, ya que como en **Unity**, el motor ya contiene muchos materiales realistas. En las versiones anteriores solo soportaba los lenguajes C++ y lua, sin embargo a partir la versión 5.4 también soporta C#, lo que ayuda mucho a los usuarios nuevos a usar **CryEngine**. Así como el motor está orientado para crear juegos fotorrealistas, **CryEngine** solo puede compilar el juego para PC y consolas en este momento. La desventaja de **CryEngine** es el tamaño de su comunidad, que es pequeña comparada con las de **Unreal** y **Unity** y por esta razón los usuarios nuevos tienen más difícil encontrar respuestas cuando tienen dudas.

[7] https://www.cryengine.com/
[8] https://en.wikipedia.org/wiki/CryEngine#Development
[9] https://www.cryengine.com/support/view/licensing

Figura 2.4: Kingdom Come: Deliverance

Figura 2.5: Prey

2.1.4. Unreal

Unreal[10] es un motor creado por Epic y también está orientado para crear juegos fotorrealistas. En un principio estaba diseñado para crear FPS pero hoy en día ya es un motor que puede hacer muchos tipos de juegos. Antes de **Unreal 4** había muchos tipos de licencia, y para usarlo había que pagar $99 por **Unreal Development Kit** pero a partir de **Unreal 4**(2014) la licencia cambió a $19 al mes. Junto a

[10] https://www.unrealengine.com/

ello y su código abierto, después de este cambio los usuarios de **Unreal** crecieron diez veces más, y por esta razón en 2015 decidieron cambiar la licencia al modelo de reparto de ingresos[11].

La característica principal de **Unreal** es la variedad de las funciones que tiene, como es un motor que ya tiene muchos años y han creado muchos juegos de AAA [6]. Existen muchas funciones que permiten a los usuarios crear cualquier tipo de juego. Algunas empresas, a pesar de tener su propio motor, también han usado **Unreal** para crear juego, como se puede ver en la figura 2.6 que es de Square Enix y la figura 2.7 que es de Microsoft. **Unreal** tiene otra característica atractiva que es el sistema **Blueprint**[7]. Es una forma de VisualScript, donde los usuarios pueden arrastrar los nodos para crear juegos, y es suficientemente potente para crear pequeños juegos y para los grupos grandes este sistema permite a los diseñadores crear las funciones que tienen relación con la jugabilidad. En la parte de lenguajes, **Unreal** solo permite C++ aunque puede usar C# usando plugin. Al igual que **CryEngine**, **Unreal** también está orientado para crear juegos grandes, y puede compilar el juego para PC y consolas, aunque igualmente puede compilar para Android, pero en cuando a dispositivos móviles es mejor **Unity**.

Figura 2.6: FINAL FANTASY VII Remake

2020 es un año importante en la industria del videojuego. Aparte de la publicación de las nuevas consolas de Sony y Microsoft, **Unreal** también ha publicado

[11] https://www.unrealengine.com/en-US/faq?active=unreal-studio

su nueva versión, **Unreal 5** [8]. Es una pena que **Unreal 5** se lanzará en año 2021, porque si no el motor que usamos en este proyecto podría haber sido **Unreal**. En **Unreal 5** saldrán dos funciones muy importante para los creadores: **Nanite** y **Lumen** [12](Figura 2.8). **Nanite** es una función que gestionar automáticamente el LOD del modelo[13] (Level of detail, una función que gestiona el detalle del modelo dependiendo de la distancia entre el objeto y la cámara), es decir, los usuarios ya no se tienen que preocuparse del número de polígonos de la escena. Además puede usar directamente modelos diseñados para cine. **Lumen** es una función sobre las luces y las sombras del juego, la luz es la característica más importante para tener unos buenos gráficos. Hasta ahora, para tener un buen efecto de iluminación había que realizar un paso de *bake* para generar las texturas y usar la luz indirecta para simular la luz principal de la escena, pero gracias a **Lumen**, ya no hace falta realizar estos pasos. La iluminación se convierte en dinámica y global. Los usuarios solo tienen que añadir y editar las luces, y la luz va a poder interactuar con la escena. Para concluir, **Nanite** y **Lumen** facilitan mucho el trabajo de los diseñadores y ayuda mucho a crear juegos espectaculares de forma fácil.

[12] https://vimeo.com/417882964

[13] https://en.wikipedia.org/wiki/Level_of_detail

Figura 2.7: Gears of War 4

Figura 2.8: Demostración de Nanite y Lumen

2.1.5. Unity

Unity[14] es un motor creado por Unity Technologies y publicado en el año 2005 [9]. En las versiones anteriores, Unity Technologies vendía el motor, a un precio mucho más barato que otros motores comerciales. En 2016, **Unity** publicó la versión gratis y cambió la forma de licencia a suscripción, es decir, cuando el ingreso supera una cantidad, tiene que suscribirse a la versión más alta, y el código es abierto a partir de la versión Pro[15].

La principal característica de **Unity** es que es un motor muy bueno para crear juegos pequeños o medianos. **Unity** soporta hasta 25 plataformas diferentes y se puede compilar el juego para cualquiera de ellas a partir del mismo proyecto. Gracias a esta propiedad la mayoría de juego indie y de móvil están hechos con **Unity**. En la figura 2.9 se muestra un juego muy famoso en año 2016 y la figura 2.10 muestra un juego creado por Blizzard, en principio sacado para PC pero en seguida publicado también para tablet y móvil.

[14] https://unity.com/

[15] https://store.unity.com/compare-plans

Figura 2.9: Pokémon Go

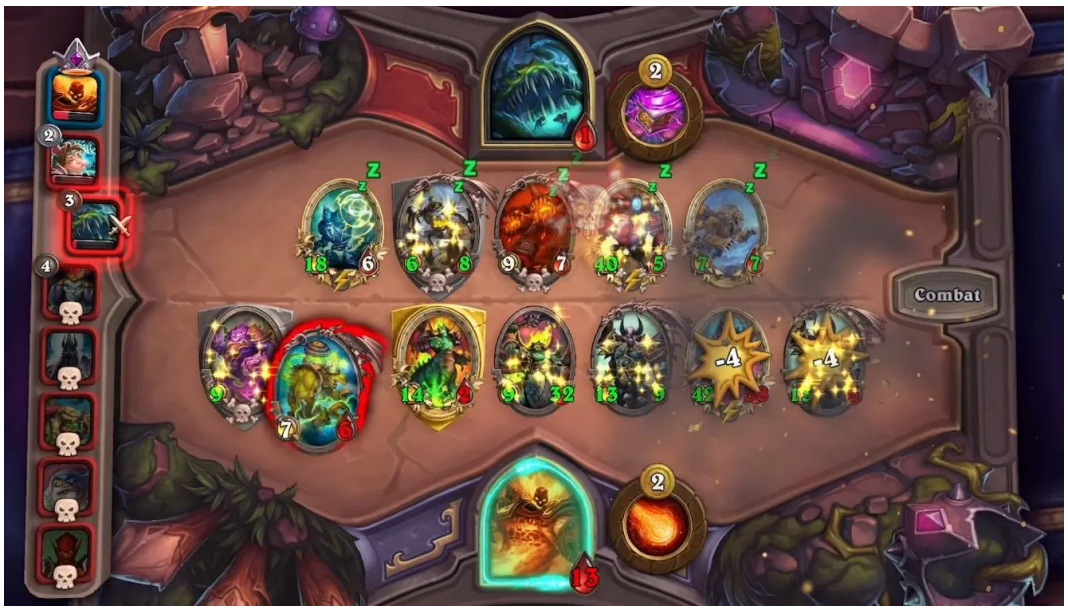


Figura 2.10: Hearthstone

La comunidad de **Unity** también es muy grande, **Unity** tiene usuarios en 195 países, lo que ayuda mucho a gente nueva para preguntar sus dudas. Con respecto a **Unreal**, **Unity** no tiene tantas funciones en su versión básica, pero se pueden encontrar más funciones en su **Asset Store**[16]. Igualmente, **Unity** no es tan amigable para los diseñadores como **Unreal**. **Unity** no tiene ninguna forma

[16] https://assetstore.unity.com/?gclid=Cj0KCQjwhIP6BRCMARIsALu9Lfm0g_547A1YeNFCY3PEVrFvN1LpPNXj6IBzMdqe6pvZ1PNMwYjsw7oaAh80EALw_wcB

de VisualScript por defecto, sin embargo la página oficial tiene muchos tutoriales de buena calidad. La mayoría de tutoriales tiene un proyecto para seguir paso a paso. Durante el confinamiento por el coronavirus **Unity** ha ofrecido los tutoriales premium gratuitamente. En la parte de lenguaje, **Unity** soportaba JavaScript, boo y C#, pero en la actualidad solo soporta C#.

La desventaja de **Unity** es que no está orientado para crear juegos de gráficos fotorrealistas, pero eso no significa que no pueda. En el año 2020 el Unity's Demo Team ha publicado un corto de animación hecho con **Unity**, como se ve en la figura 2.11[17].

Figura 2.11: The Heretic

Para acabar queremos presentar el motivo por el que hemos elegido este motor. **Unity** es un programa muy popular, todos los miembros del proyecto han usado **Unity** en la asignatura **Introducción a la Programación de Videojuegos**, y todos sabíamos un poco de **Unity** antes de empezar este trabajo. Quizá no tener tantas funciones en su versión básica facilita aprenderlo, y cuando para centrarse en algunas partes se puede buscar información. Personalmente he aprendido muchas técnicas para hacer un videojuego desde este motor, por ejemplo renderizado, shader, iluminación. . .. Algunas de estas técnicas se han aplicado a este proyecto.

[17] https://www.youtube.com/watch?v=iQZobAhgayA

2.2 Juegos similares en el mercado

La creación de videojuegos es un arte, y cuando queremos saber si un juego es bueno o no, deberíamos mirar los comentarios de los usuarios que han jugado el juego. Por ello, los puntos fuertes y débiles de los juegos que vamos a explicar en este apartado se han obtenido a partir de comentarios de los jugadores. Los juegos seleccionados son juegos parecidos al nuestro y que tienen una valoración muy positiva.

2.2.1. ABZÛ

Abzû[18] es un juego desarrollado por Giant Squid en año 2016. La protagonista del juego es una submarinista y su objetivo es explorar un océano lleno de vida. En el océano existen muchos tipos de peces y el jugador puede interactuar con algunos de ellos. Además los peces reaccionan de diferentes formas ante el submarinista y los depredadores. El control del juego es muy simple, mover, interactuar y montar al pez. La principal característica del juego es que tiene una buena combinación de gráficos y música para dar sensación suave y relajada a los jugadores, como podemos ver en las figuras 2.12 y 2.13. Aunque los modelos son **low poly**[19], cuadran muy bien con el ambiente.

En Steam, el 92 % de 15194 reseñas han valorado el juego como muy positivo. En los comentarios positivos, a muchos de ellos les gusta el ambiente creado por este juego y piensan que es un juego que les permite relajarse, se pueden mirar los comportamientos de los peces, y aprender alguna sobre especies que no conocen.

Los comentarios negativos indican que el juego es aburrido. Aunque tiene algunos puzles, son demasiado simples. Pese a que la duración del juego es más o menos de tres horas, es demasiado si la jugabilidad consiste solo en avanzar hacia delante. El juego permite controlar con mandos o teclado y ratón, pero hay usuarios que dichos controles no son adecuados para jugar a este juego.

[18] https://store.steampowered.com/app/384190/ABZU/#app_reviews_hash
[19] https://en.wikipedia.org/wiki/Low_poly

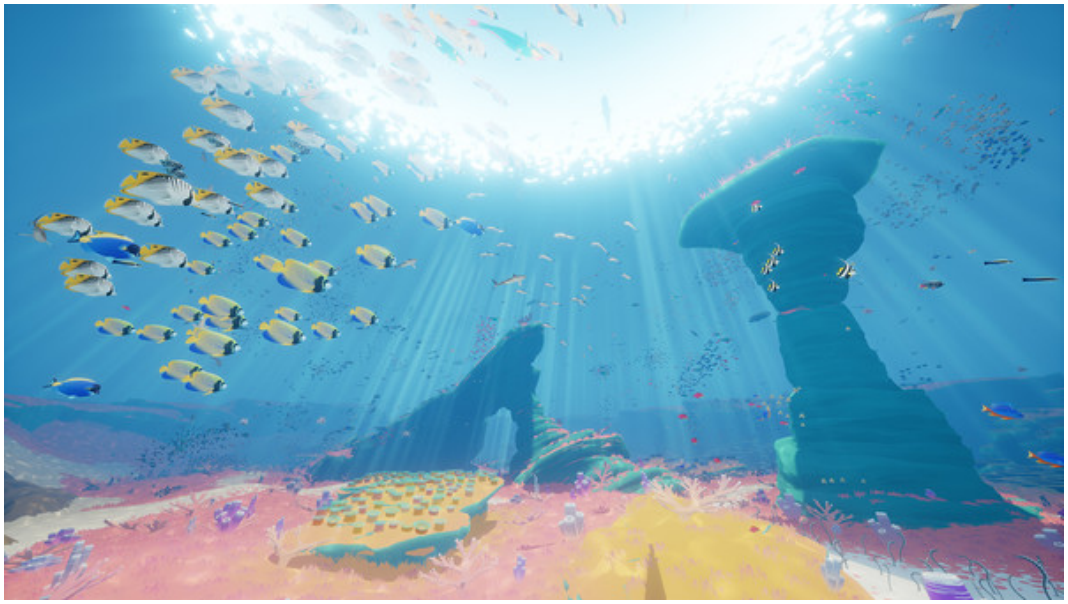


Figura 2.12: Abzû

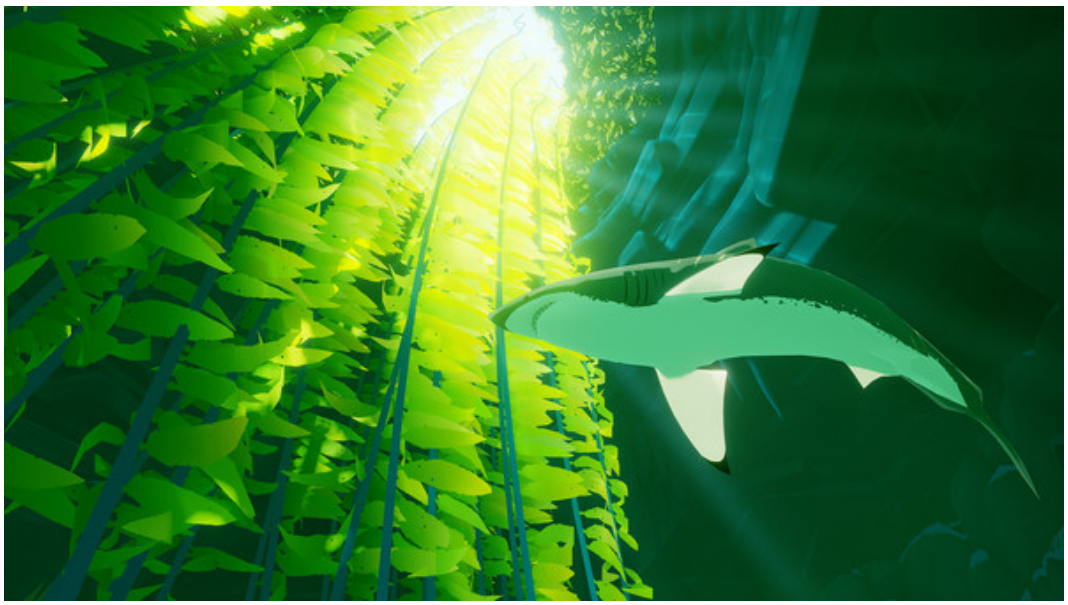


Figura 2.13: Abzû

2.2.2. RiME

RiME[20] es un juego de resolver puzles en tercera persona. Está creado por la empresa español Tequila Works. El protagonista es un chico que se despierta en una isla desconocida, y con la ayuda de un zorro mágico explora la isla para descubrir por qué está en esa isla. El juego permite al jugador subir, agarrar objetos, empujar objetos y cantar para activar algún obstáculo.

[20] https://store.steampowered.com/app/493200/RiME/

En Steam, el 90 % de 2537 reseñas son muy positivas. La mayoría ha valorado bien los gráficos (ver la figura 2.14) y muy bien el sonido. El aspecto mejor valorado de este juego es la historia. La historia en sí es normal, no tiene diálogos y se presenta por las cinemáticas. Sin embargo al final del juego se van a descubrir todos los misterios y convierte la historia del juego en una maravilla.

Figura 2.14: RiME

La mayoría de los comentarios negativos de este juego es por su optimización. Muchas personas no pueden jugar con su máquina, porque la tasa de fotogramas es tan baja que molesta la vista. Otro aspecto negativo es que el juego tiene partes de coleccionar objetos por el mapa, pero no hay ningún aviso cuando cambia de escena, debido a eso el jugador no puede ir a la escena anterior para recoger objetos.

2.2.3. Spirit of the North

Spirit of the North[21] es un juego creado por Infuse Studio en el año 2019. En principio solo estaba disponible para PS4, pero en 2020 también se ha publicado para PC y Nintendo Switch. El jugador va a controlar un hermoso zorro para explorar Islandia como podemos ver en la figura 2.16. Por el camino el jugador va encontrando otros zorros espíritus(Figura 2.15) para guiarle y ayudarle a resol-

[21] https://store.steampowered.com/app/1213700/Spirit_of_the_North/

ver los puzles. El juego no tiene historia, para comprender el contexto hay que resolver los puzles y estudiar los murales que hay en la escena.

Figura 2.15: Spirit of the North

Como el juego fue publicado en mayo de 2020 en PC, no hay muchas valoraciones en Steam por el momento. El 86 % de 418 han valorado muy positivamente el juego. **Spirit of the North** es parecido a otros juegos de exploración. A la mayoría de los jugadores le gustan los gráficos y el ambiente relajado. El aspecto más peculiar de este juego es que controlas un zorro para explorar, lo que atrae a muchos usuarios a jugar este juego.

La mayoría de los comentarios negativos son por falta de vida en la escena. El juego tiene un mapa grande, pero no tiene muchos decorados, lo que da una sensación de exploración en un lugar donde no hay nada. Otro aspecto valorado negativamente es que el control no es muy amable y el juego contiene errores. Algunos errores obligan al jugador a reiniciar el juego. Además el juego se guarda en los puntos de control, lo que obliga al jugador a volver a resolver los puzles ya resueltos.

Figura 2.16: Spirit of the North

2.3 Juegos accesibles en el mercado

Una vez analizados juegos de temática similar, vamos a estudiar algunos juegos accesibles que hay en el mercado.

2.3.1. The Last of Us Part II

Es un juego desarrollado por Naughty Dog y lanzado en junio de 2020. Es un juego de acción y aventura, que contiene contenido de sigilo, disparos, ataque cuerpo a cuerpo y puzles. En la figura 2.17 muestra una captura del juego. Todos estos contenidos hacen que, a primera vista, parezca que no es un juego accesible.

A pesar de que el juego ha tenido poco éxito en su historia, sin duda es un juego de siguiente generación, y no solo me refiero por la parte gráfica del juego, sino también la accesibilidad. El juego tiene más de 60 configuraciones que permiten al jugador personalizar su control preferido.

El juego tiene un modo de alto contraste que resalta al personaje como muestra en la figura 2.18, los enemigos y los objetos interactivos. También tiene audiodescripción, que se encarga de leer todos los textos que hay en el juego, e incluso los iconos. El juego también tiene una función de lupa que permite al jugador ampliar la pantalla pausando el juego o no.

Figura 2.17: The Last of Us Part II

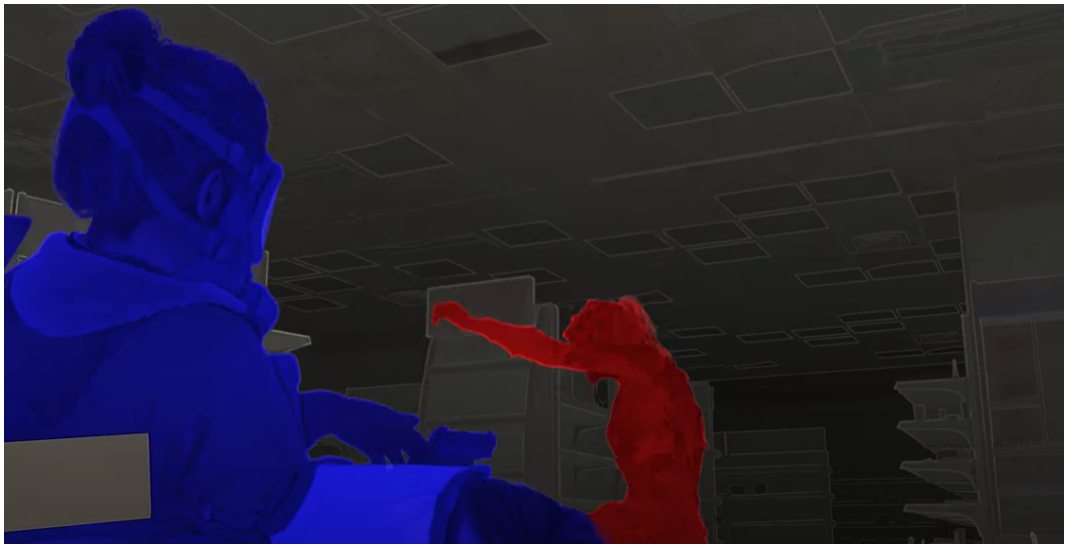


Figura 2.18: Modo alto contraste

Para personas con movilidad reducida o discapacidad intelectual, el juego tiene muchas opciones de ayuda. Por ejemplo, se puedes escanear qué hay cerca del jugador y se puede mover al lugar elegido usando un botón, pasar por los lugares que necesitan una secuencia de acciones compleja usando solo un botón, e incluso modificar la dificultad de la inteligencia artificial de los enemigos.

Es un juego donde su jugabilidad está completamente diseñada para personas que sin discapacidad y que añade opciones que permite a personas con discapaci-

dad poder disfrutar de igual manera (este es el objetivo principal de la Usabilidad Universal).

2.3.2. The Recycling Heroes

Es un juego educativo desarrollado por FUNDACION 3M ESPAÑA y lanzado en julio de 2020. Es un juego gratuito de PS4. La intención del juego es enseñar a las personas la importancia del reciclaje y ayudar a los niños a desarrollar las relaciones interpersonales.

Figura 2.19: The Recycling Heroes

Es un juego con muchas opciones accesibles: dispone de audiodescripción, que lee todos los elementos de la ventana, facilidades para la comprensión del contenido mediante iconos para los niños y lengua de signos(Figura 2.19) para los sordos. El juego también usa la tecnología **PlayLink**, que permite a los jugadores usar sus móviles como mando, y también pueden personalizar la forma de control con toques, con un botón deslizable o con el giroscopio.

CAPÍTULO 3

Análisis del problema

En este capítulo vamos a analizar los requisitos y problema de cada bloque del proyecto y mostrar una posible solución. Antes de entrar en el análisis, vamos a presentar los conocimientos básicos sobre **Unity** para poder entender mejor los problemas y las soluciones propuestas.

3.1 Conocimientos previos

Al empezar un nuevo proyecto, **Unity** crea automáticamente una escena. Un juego está compuesto por varias escenas. Cada escena está compuesta por objetos del juego (**GameObject**), que son los elementos básicos de **Unity**. En cada objeto del juego podemos añadir componentes para dar funcionalidad al objeto, como convertirse en una cámara, modelo, funciones, etc. Cada objeto debe tener al menos un componente Transform, que representa su posición, su rotación y su escala. Cuando el objeto es un elemento de interfaz, Transform se convierte en Rect Transform. Aparte de Transform, **Unity** tiene muchos otros componentes para facilitar el desarrollo del juego como **Collider**s (para el cálculo de colisiones), **Rigid Body** (para participar en el cálculo de física), **Audio Source** (para reproducir sonido). . . . El usuario también puede crear su propio componente. Cualquier clase C# que herede de **MonoBehaviour** se puede añadir al objeto como un componente. En la siguiente lista vamos a nombrar los componentes más importantes para el proyecto.

- **Event System**, cuando el usuario crea un canvas (un objeto para guardar los elementos de interfaces), si la escena no contiene un objeto con un Event System, entonces **Unity** crea un objeto con este componente. El componente se encarga de gestionar los elementos interactivos de la escena. Normalmente, **Event System** acompaña a otro componente que es **Input Module**, que por defecto es **Standalone Input Module**. Este componente se encarga de gestionar los eventos generados por la entrada, como por ejemplo si el usuario ha movido el ratón o ha pulsado una tecla. Por ejemplo, cuando el usuario hace clic con el ratón, **Input module** calcula si el usuario ha clicado en un botón, y si es el caso, pide a **Event System** enviar un mensaje al botón clicado para ejecutar la función asociada.

- **Cinemachine**, es un paquete de componentes que no viene acompañado con la versión básica de **Unity**. Para usarlo hay que importarlo desde la ventana de gestión de paquetes. Es un paquete que ya ha pasado el período de experimentación, es decir, es bastante estable. Como su nombre indica es un paquete para gestionar la cámara, y se puede usar para crear cinemáticas y también se puede usar para crear juego.

- **Post Process**, es otro paquete que hay que importar. Este componente sirve para añadir efectos especiales a la cámara. Es un componente que puede conseguir efectos visuales de forma fácil. Con este componente se pueden añadir muchos efectos. La figura 3.1 muestra una comparación antes y después de aplicar un postproceso. Hay que tener cuidado, ya que este componente solo aplica efectos a las cámaras donde está configurado. Los elementos de interfaz no se van a ver afectados.

- **Nav Mesh Surface**, es un componente de **Standard Assets**[1] que hay que buscar en la tienda de asset de **Unity**. Este componente genera las áreas caminables en la escena, por donde puede andar cada agente y facilita la búsqueda de caminos para la inteligencia artificial.

- **Projector**, es un componente que permite proyectar un material a los objetos que están en determinada zona.

[1] Standard Assets

Figura 3.1: Antes y después de aplicar el postproceso

3.2 Control del juego

La parte más importante de un juego es el control. Sin el control el jugador no puede interactuar con el juego, y el juego deja de ser un juego. Antes de analizar los problemas vamos a ver qué requisitos tenemos:

1. El jugador debe poder configurar las teclas dentro del juego.
2. Cada tipo de control debe poder controlar todo el juego.
3. Establecer un tiempo mínimo entre pulsaciones de teclado para evitar eventos repetidos (funcionalidad diseñada para personas con temblores en las manos).

En primer lugar vamos a estudiar el primer requisito, cambiar la configuración del teclado. Parece una función básica, pero después de una investigación nos dimos cuenta de que el **Input Manager** que tiene **Unity** por defecto no puede cambiar teclas ni puede saber qué tecla ha usado para una acción. Se puede cambiar la configuración del teclado en la ventana de inicio antes de empezar el juego(Figura 3.2), pero eso significa que cada vez que el jugador quiera cambiar las teclas, tiene que reiniciar el juego y eso no es accesible.

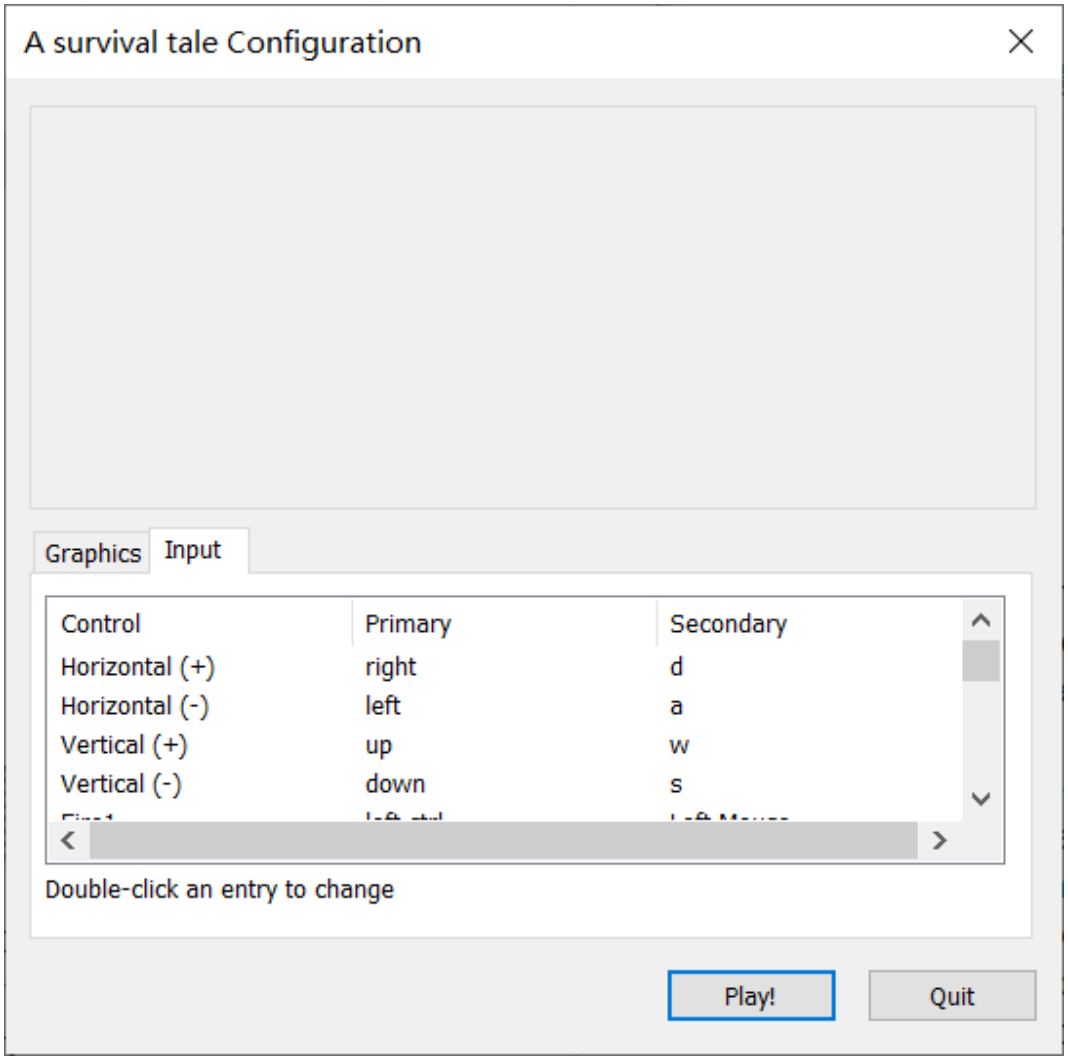


Figura 3.2: Ventana de configuración de Unity

Por la razón anterior hemos buscado si **Unity** tiene otro manejador de entrada y hemos encontrado un paquete que se llama **Input System**. Es un sistema complicado y tiene más funciones, aunque es un paquete en la fase experimental. El uso de este sistema es bastante fácil pero la configuración previa y para cambiar las teclas es muy complicado. Además, si queremos guardar las teclas tenemos que guardarlas en una estructura de datos.

Como **Input System** es un paquete en fase experimental, es posible que no sea estable. Como su forma de configurar el teclado es complicada, hemos decidido introducir cambios en **Input Manager**. Usamos **Input Manager** para leer las teclas, pero gestionamos las teclas nosotros, aunque eso introduce a un nuevo problema. En Event System hemos explicado que **Input Module** sirve para gestionar los eventos de entrada y por defecto es **Standalone Input Module**, por eso

solo reconoce los botones gestionados por Input Manager. Para resolver este problema hemos decidido modificar **Standalone Input Module** para que reconozca las teclas que hemos guardado, y así cumplimos los dos requisitos.

3.3 Sistema de guardado

Esta también es una función muy importante en cualquier juego. En primer lugar vamos a analizar los requisitos:

1. Los datos guardados deben de ser correctos.
2. Se deben poder guardar diferentes tipos de datos.
3. El lugar para guardar el fichero debe ser personalizable, por lo menos para los desarrolladores.
4. Los datos no deben ser modificados fácil con editores de texto.

El primer requisito es relativamente fácil, en cuando guardar, debemos guardar los datos enteros y no debemos modificar los datos por partes. Antes de implementar el sistema de guardado, se ha hecho una búsqueda de implementaciones existentes, y hemos encontrado cuatro formas, que se pueden resumir en tres tipos diferentes.

La primera forma la lleva **Unity** por defecto. **Unity** tiene una clase **PlayerPrefs**, con unos métodos estáticos que permiten leer y escribir float, int y string. Aunque solo se soportan esos tres tipos de datos, podemos convertir las estructura de datos del juego a estos tres tipos básicos. El problema de esta forma es que no cumple el tercer requisito. El lugar donde guarda los datos lo decide **Unity**, aunque **Unity** lo guarda en diferentes lugares dependiendo del sistema operativo, pero el lugar de guardar es fijo, no podemos decidirlo por nosotros.

La segunda forma es guardar los datos como XML o JSON. Estos dos tipos de datos son muy populares, y existen librerías para poder usarlos en **Unity**. La idea principal de estas dos formas de guardar es convertir los datos a string, y delimita mediante etiquetas los valores y su tipo. Es muy útil en redes pero no

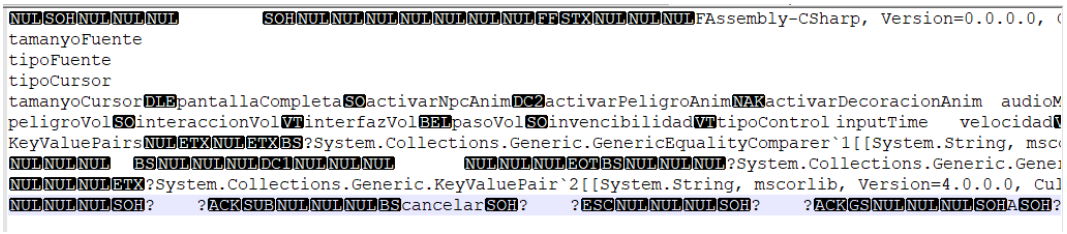


Figura 3.3: Datos de sistema del proyecto

cumple el cuarto requisito, porque los usuarios pueden modificar los valores de algunos campos fácilmente.

La tercera forma es usar la serialización de C#. En la asignatura de **Programación** hemos aprendido que serialización es convertir los objetos a un fichero binario, y la ventaja es que es una forma rápida de guardar y cargar, además de ahorrar memoria. La desventaja de la serialización es que no es legible para los usuarios, pero esto es justo que queremos para nuestro sistema de guardar. El problema de la serialización es que no soporta estructuras de **Unity** como Vector3, Color y Quaternion. Para resolver este problema hemos encontrado la clase **Surrogate**[10]. Cuando C# serializa el objeto necesita una clase **BinaryFormatter** para convertir los objetos a formato binario, y así podemos escribir nuestro **Surrogate** para decir a **BinaryFormatter** cómo convertir los estructuras de **Unity** a formato binario.

A pesar de que la serialización dificulta mucho leer y modificar los datos, siguen existiendo algunos campos que son legibles, como se puede ver en la figura 3.3 . Para mejorar un poco más este aspecto hemos usado la técnica que hemos aprendido en la asignatura **Redes De Computadores**. Cada vez que guardamos, calculamos el hash del dato y guardamos el valor junto con el fichero. Cuando cargamos el dato calculamos otra vez el valor del hash y comprobamos con el valor guardado, si los dos valores son diferentes entonces significa que el dato ha sido modificado, y se producirá un error.

Para acabar, se ha usado **FileStream** de C# para guardar el fichero y hemos decidido guardarlo en la carpeta que contiene el juego.

3.4 Interfaz de usuario

En este apartado vamos a dividir la descripción de la interfaz de usuario en diferentes subsecciones.

3.4.1. Sistema de navegación por el menú

En el apartado dedicado a los controles hemos resuelto cómo controlar los elementos de interfaces con las teclas definidas por nosotros, pero para navegar con dos botones vamos a necesitar un sistema de navegación.

La idea ha sido propuesta por Ariel, y se basa en la funcionalidad descrita en la página http://www.oneswitch.org.uk/page/switch-scanning-practice. La idea es cada cierto tiempo cambiar el elemento seleccionado, y así el jugador solo tiene que pulsar un botón para confirmar o cancelar. Después de una prueba se encontró un problema de usabilidad, y es que al intentar escribir mi nombre he tardado 147 segundos en conseguirlo, incluyendo el tiempo por falta de concentración y por no pulsar el botón de confirmar. Por ello se ha decidido añadir un requisito más para la función:

1. Se debe poder navegar todo los tipos de botones.

2. Se debe poder navegar un bloque que contiene muchos botones.

3. Se debe poder navegar los bloques junto con los botones.

Para analizar cómo se implementa esta función hemos buscado en la documentación de **Event System**[11], y hemos encontrado un método **SetSelectedGameObject** para seleccionar los elementos, y el parámetro es un **GameObject**, como todo elemento seleccionable es un objeto, solo necesitamos una lista de tipo **GameObject** para navegar y cumplimos el primer requisito.

Para evitar que el jugador tenga que esperar demasiado para elegir un botón hemos decidido organizar la interfaz en bloques navegables, donde cada bloque define un conjunto de botones.

La figura 3.4 muestra un ejemplo con las opciones de visualización del juego. El sistema de navegación se mueve inicialmente entre las opciones de pantalla completa, tamaño de letra, fuente, control animaciones, cursor y opciones contraste. Cuando el jugador pulsa el botón de confirmar, por ejemplo, en el bloque de pantalla completa, entonces podrá navegar entre el botón si y no. Si el jugador pulsa cancelar volverá a navegar entre los bloques anteriores.

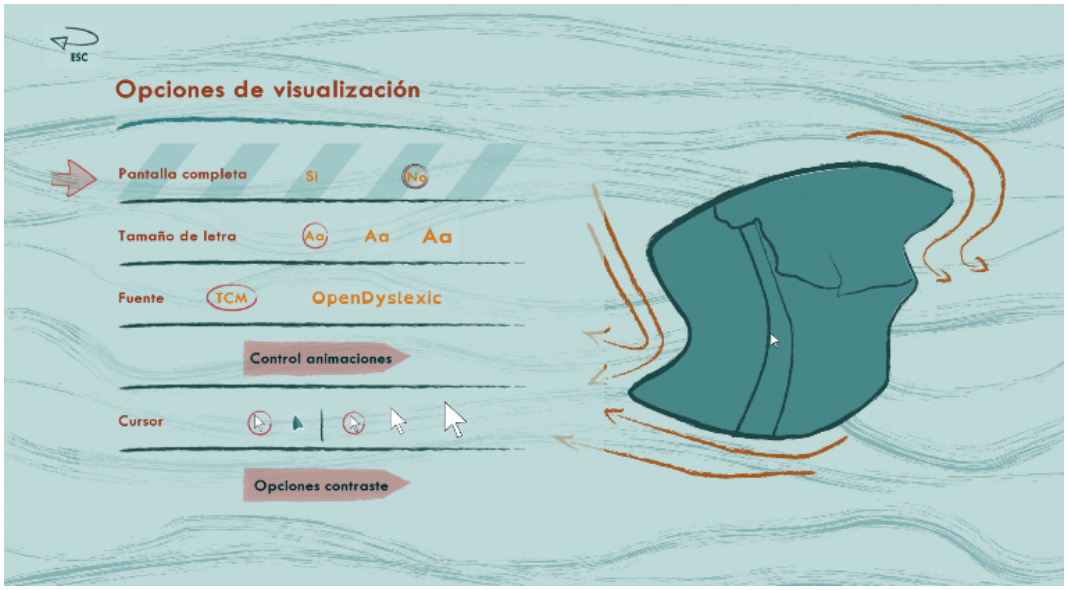


Figura 3.4: Opciones de visualización

Hay que tener en cuenta que el control de animaciones y las opciones de contraste son dos botones y no tienen bloque asociado. Cuando el usuario selecciona uno de estos dos botones, se debe comportar como un botón y no como un bloque.

En cuando a la navegación con teclado y ratón va a ser igual que con otros juegos:

- Teclado: se mueve el foco entre los botones usando las teclas de dirección. **Event System** decide qué elemento seleccionar. Cada elemento seleccionable tiene cuatro flechas para representar qué elemento hay que elegir cuando el jugador pulsa dicha dirección. Esta selección se puede generar automáticamente o definirse manualmente.

- Ratón: cuando el puntero del ratón pasa por encima de un elemento, se muestra un círculo que indica que está elegido. El componente **Button** y **Toggle** proporcionan esta funcionalidad.

Para acabar, como personalizar el control es uno de nuestros objetivos, el concepto de bloque sólo se necesita en el control de dos botones. En los otros dos tipos de control no es necesario, ya que baja la usabilidad, porque necesita más pulsaciones para elegir una acción. Por esta razón, los bloques solo deben funcionar para modo de dos botones.

3.4.2. Panel de color

Es un panel que permite al jugador elegir la gama de colores para cada tipo de objetos. Los elementos básicos de este panel son una paleta de colores y una ventana pequeña para mostrar el color que ha elegido. Antes de enumerar los requisitos vamos a analizar la forma de elegir color para cada tipo de control.

- Teclado, este tipo de control es relativamente fácil. Usamos las teclas de direcciones para mover el puntero y usamos la tecla de confirmar para elegir el color.

- Ratón, es más complicado que el teclado. Podemos hacer que el puntero siga a la posición del ratón y clicar para elegir color, pero esa forma no es usable para personas con temblor en las manos. Por lo tanto, cambiamos la forma a cuando el botón izquierdo está pulsado entonces el puntero sigue a la posición del ratón, y añadimos un botón para confirmar el color.

- Modo de dos botones, es el control más difícil de diseñar. Aparecen cinco barras de colores diferentes y el jugador puede elegir una de estas cinco barras y cambiar la saturación del color como podemos ver en la figura 3.5. Este método es un poco limitado, porque otras formas de control pueden elegir muchos más colores que el modo de dos botones. Para unificar la funcionalidad, hemos decidido que todos los tipos de control usen el mismo panel de color. El sistema de navegación solo sirve para elegir botones, pues

para que podamos navegar a través de los colores hemos tenido que crear una nueva clase para conseguirlo.

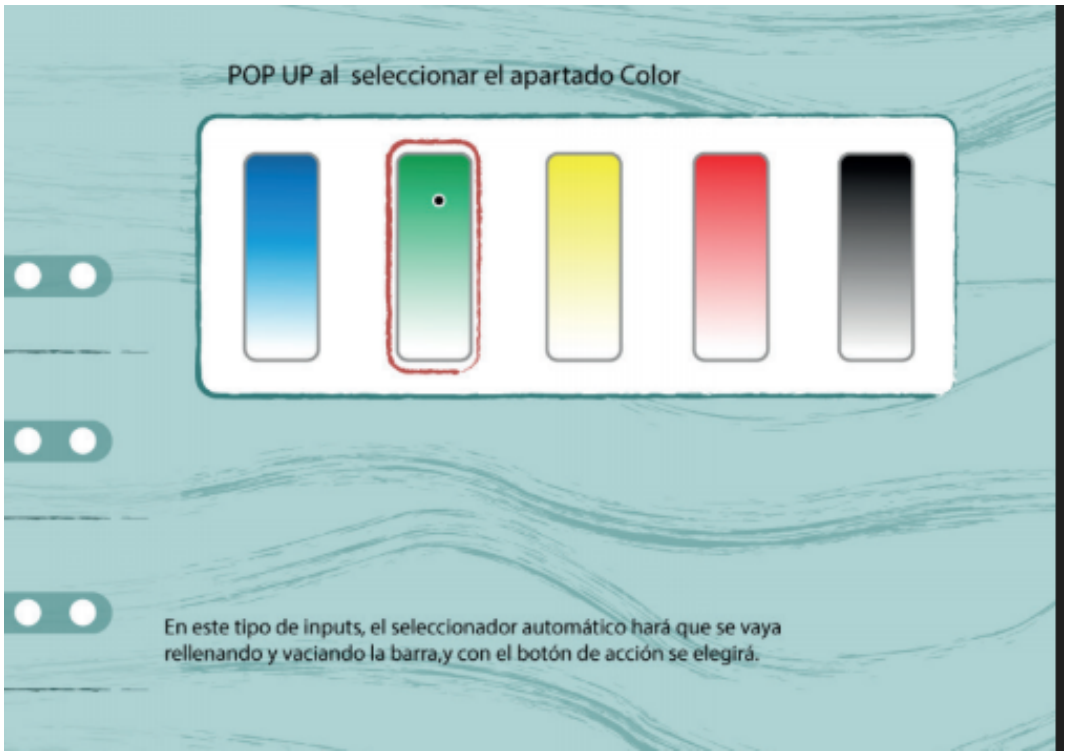


Figura 3.5: Panel de color para el modo de dos botones

Los requisitos se pueden resumir en los siguientes:

1. Una ventana para mostrar el color donde está el puntero.
2. Un botón del ratón puede confirmar el color.
3. Una clase que permite mover el puntero con teclas o navegar el puntero automáticamente.
4. Permitir cambiar la opacidad del color.

3.4.3. Cambiar fuente y tamaño

El diseñador quiere que el cambio de fuente y tamaño también afecte al menú. En el caso de los botones quiere que se escalen con respecto a la proporción predefinida. El diseñador ha seleccionado dos fuentes que podemos ver en la figura 3.6: una es **TCM**(Tw Cen MT) y otra es **OpenDyslexic**. **TCM** es la fuente por

TCM OpenDyslexic

Figura 3.6: Fuentes

defecto, que no presenta problemas, pero **OpenDyslexic** es una fuente que ocupa mucho espacio, por lo que para el mismo tamaño, cuando se usa **OpenDyslexic**, las letras se salen fuera de la imagen del botón. Para resolver el problema hemos diseñado el menú con **OpenDyslexic** y luego cambiado a **TCM**.

Había otro problema con los botones, y es que algunos tienen un texto muy largo y no es posible tener todo los textos al mismo tamaño de fuente. Para resolverlo hemos usado el componente **Text Mesh Pro** para los textos. Es un componente que tiene muchas configuraciones sobre textos y tiene una función de cambiar tamaño automáticamente. Gracias a esta función solo necesitamos definir el espacio del texto y el componente ajusta el tamaño automáticamente.

3.4.4. Flecha de indicación

Es una función para ayudar a los usuarios a localizar el elemento de interfaz que está activo. La idea es mostrar una flecha a la izquierda(Figura 3.7) cuando un botón o un bloque está seleccionado o cuando el ratón está dentro de la imagen del botón.

Figura 3.7: Flecha de indicación

El problema para implementar esta función es que por defecto, **Button** y **Toggle** de **Unity** no permite asignar un callback[2] a eventos *OnSelect* u *OnPointerEnter*. Para resolver esto hemos tenido que escribir una clase y heredar de **Button** (e igualmente para **Toggle**). Para facilitar su uso también hemos tenido que escribir una clase de editor para estas dos clases.

3.5 Función para cambiar el color de los objetos

Esta función es para los usuarios con problemas visuales. El requisito del diseñador es dividir los objetos en tres tipos: interactivos, jugador y fondos. El jugador deberá poder elegir la gama de color de estos tres tipos de objetos.

Es una función difícil de conseguir, porque tiene relación con la realización y no es una función que suelen tener los juegos. Se han estudiado tres soluciones, pero al final solo una es factible.

La primera opción es escribir todos los shaders (sombreador)[3]. Así cada material puede tener un color básico y solo tenemos que modificar este valor para cambiar el color. Pero el problema es que no sabemos tanto de shaders y no podemos reescribir todos los shaders que vamos a usar.

[2] https://es.wikipedia.org/wiki/Callback_(inform%C3%A1tica)
[3] https://es.wikipedia.org/wiki/Sombreador

La segunda opción, como los modelos pueden tener varios materiales, pues podemos crear tres materiales de cada tipo, y asignamos los materiales a cada tipo de objeto. La ventaja es que cuando modificamos el color solo tenemos que modificar el color de un material. El problema es cuando aparecen modelos que tienen varios materiales. En este caso tiene diferentes mapeados de las texturas(UV mapping)[4] de cada material, por lo tanto aunque añadamos un material, el material no envuelve a todo el modelo.

La última solución es que como la mayoría de los materiales tienen un atributo de color básico, podemos guardar todo los materiales separados por tipo, y cuando cambia el color, recorremos todo los materiales para cambiar su color básico. Ha aparecido un problema con el componente **Terrain**, que es un componente para crear terreno y también puede dibujar los tipos de suelos sobre él. El problema es que para dibujar los tipos de suelo se usa solo la textura. La textura es una imagen, y la hemos intentado modificar directamente, pero tarda demasiado y no es factible. Después de unas investigaciones hemos encontrado el shader que se ha usado para dibujar los suelos, y hemos descargado todos los shaders por defecto de **Unity**[5].

Antes de explicar cómo hemos resuelto el problema, queremos explicar una cosa. En este documento[12] hemos visto que el comportamiento de **Terrain** varía dependiendo del renderizado que se está usando. En nuestro proyecto hemos usado el renderizado por defecto (**Built-in Render Pipeline**). Con este renderizado **Terrain** realiza varios pasos si hay varios tipos de suelo.

Para resolver el problema del suelo hemos decidido modificar el shader para añadir un atributo de color básico, y en la función de dibujar hemos añadido el siguiente código:

```
1 o.Albedo = mixedDiffuse.rgb* _Color; // color_salida = color_salida *
    color_basico
```

Como estamos usando el renderizado por defecto, tenemos que cambiar dos shaders, uno es FirstPass.shader y otro es AddPass.shader. El resultado se puede ver en la figura 3.8, donde se ha aplicado al terreno un tinte violeta.

[4] https://es.wikipedia.org/wiki/Mapeado_de_texturas

[5] https://unity3d.com/cn/get-unity/download/archive

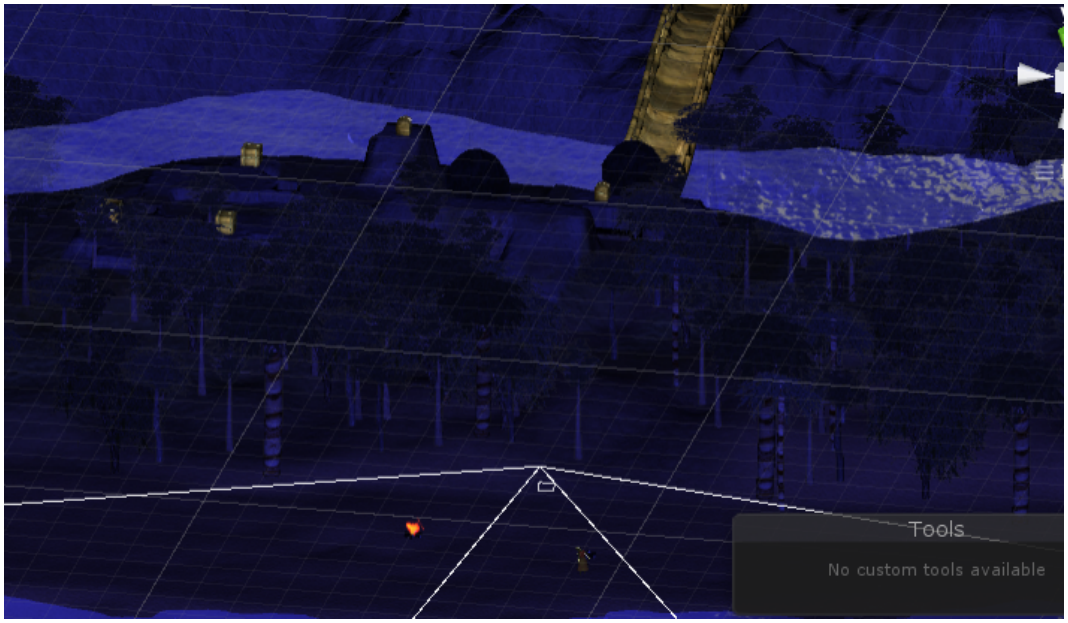


Figura 3.8: Función de cambiar color

En el panel del color hay un requisito, que es permitir cambiar la opacidad del color. En principio usábamos el panel RGB para elegir color y modificar el canal alfa para cambiar opacidad. Como se puede ver en código para añadir un color básico mostrado anteriormente, **o.Albedo** solo representa color. El valor del canal alfa no afecta al material. Para resolver el problema hemos decidido usar el panel HSV(Figura 3.9), donde el jugador modificar el valor H para elegir color y el valor S para modificar la claridad.

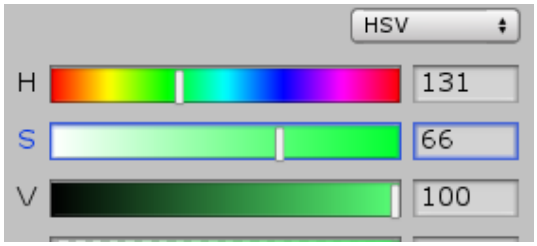


Figura 3.9: Color HSV

3.6 Corrección de los colores

En el apartado anterior hemos implementado una forma para cambiar los colores por tipo de objeto, pero el coste es bastante alto, tanto desde el punto de vista computacional como de desarrollo. Además reduce la calidad gráfica porque se pierde la variedad de color. Una alternativa que hemos encontrado es la función **ColorBlindCorrection**[6]. La idea de esta función es usar Post Process de **Unity**para cambiar los colores de la imagen. La ventaja de este método es que el coste depende de la resolución del juego y no del número de materiales del proyecto. Además no pierde tantas variedades de colores. En el proyecto no lo hemos implementado, pero este método también se puede usar para personas con alteraciones visuales como pronatopia, deuteranopía o tricromatismo anómalo.

3.7 Personaje

Después de analizar los elementos de la interfaz vamos a describir con el contenido del juego, empezando por el personaje. En este apartado vamos a dividir el análisis en dos partes: una sobre la funcionalidad básica del personaje y otra sobre la parte interactiva del personaje.

3.7.1. Funcionalidad básica

Es común utilizar una máquina de estados[7], para controlar a un personaje. Como su nombre indica, la máquina de estados es una estructura compuesta por varios estados, y dependiendo del estado actual va a reaccionar de distinta forma.

Para la implementación de la máquina de estados no vamos a utilizar estructuras if else, porque va a tener un coste lineal y va a limitar el número de estados que puede tener. Otro requisito es que las acciones puedan ser reutilizadas en diferentes estados. Por ejemplo, la acción de mover desde un estado parado y desde un estado en movimiento deben ser iguales y para reducir el gasto de memoria,

[6] https://gist.github.com/simonwittber/acfac14329312ecbae7e28f6ba9a1c5c

[7] https://es.wikipedia.org/wiki/Aut%C3%B3mata_finito

solo debemos tener un código de la acción mover. Para cumplir el objetivo de personalizar cada tipo de control, las acciones de distintos tipos de control deben ser diferentes y por eficiencia, el coste de seleccionar el tipo de control también tiene que ser constante.

Para resumirlo, los requisitos de la máquina de estado son los siguientes:

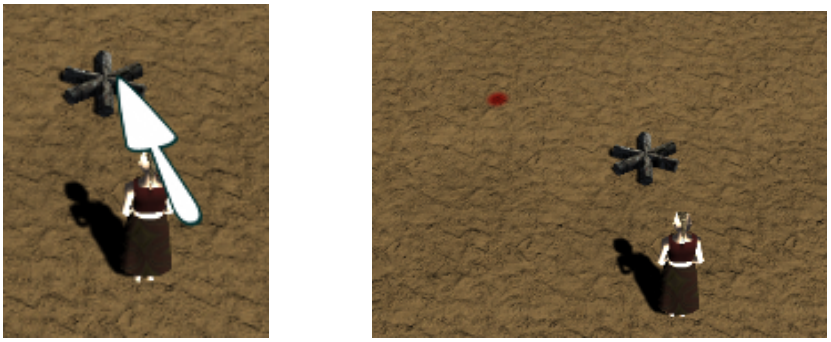
1. El coste de seleccionar el estado debe ser constante.

2. Se puede utilizar acciones de otros estados.

3. Personalizar las acciones para diferentes tipos de control y el coste de seleccionar control también tiene que ser constante.

Ya tenemos la máquina de estado, ahora vamos a analizar cómo mover el personaje. Para el control de teclado, dicha implementación es fácil, ya que podemos usar **RigidBody** o **CharacterController** para mover el personaje. Para que el jugador que usa el ratón se pueda mover fácilmente por la escena necesitamos una función de búsqueda de caminos, para la que se usado el componente **Nav Mesh**.

Nav Mesh resuelve el cálculo de rutas al utilizar el ratón, pero para el modo de dos botones necesitamos otra solución. Cuando el usuario está usando el control mediante dos botones, se muestra una flecha encima del personaje girando, y cuando el jugador pulsa confirmar, entonces la flecha se para y a continuación se muestra un punto hacia donde apunta la flecha para que el jugador pueda decidir a dónde se quiere mover. La flecha y el punto podemos ver en la figura 3.10.



Elegir dirección Elegir Distancia

Figura 3.10: Forma de controlar con modo de dos botones

La visualización del punto de destino es problemática cuando se dibuja sobre el suelo, y hay objetos más altos que lo tapan. Para resolver este problema hemos estudiado qué hacen otros juegos. Por ejemplo, hemos encontrado que Final Fantasy XIV tiene una función parecida, que se puede ver en la figura 3.11. Parece que el círculo está proyectado sobre el suelo. En consecuencia hemos decidido usar el componente **Projector**.

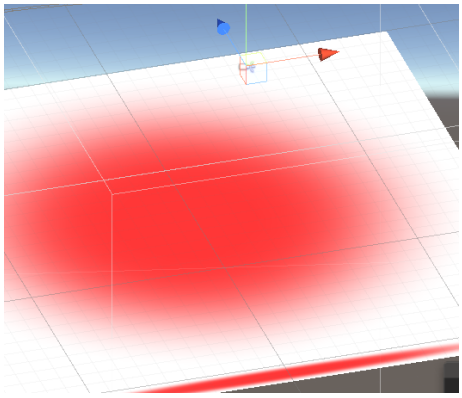
Figura 3.11: Final Fantasy XIV

En conocimiento previo hemos explicado que **Projector** proyecta un material, es decir, si lo usamos con un material por defecto, proyectará dicho material como una textura, como podemos ver en la figura 3.12. Afortunadamente hemos encontrado en los **Standard Assets** la funcionalidad necesaria, donde un **Projector** usa un material especial que no dibuja la parte de textura con color blanco. Gracias a ello, hemos podido cambiar la textura de este objeto y hemos conseguido que proyecte el punto.

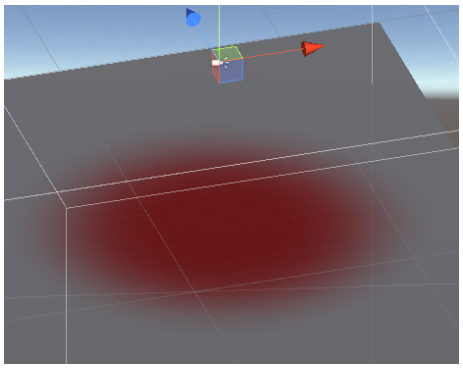
Los requisitos para mover el personaje se pueden resumir en dos:

1. Se debe poder calcular la ruta desde la posición actual hasta un punto indicado por el usuario.

2. Se debe poder definir el destino del personaje mediante dos botones.



Proyección de un material normal



Proyección de un material especial

Figura 3.12: Projector

Una vez podemos mover al personaje, tenemos que pensar cómo implementar el movimiento de la cámara. En la introducción hemos dicho que es un juego de tercera persona, pero existen muchos tipos de cámara de tercera persona. En principio hemos decidido que la cámara siga al personaje, pero hay algunos lugares del nivel donde los árboles tapan la vista del jugador. Para resolverlo, hemos decidido usar Cinemachine. En **Cinemachine** existe un tipo de cámara que se llama **Dolly Track**, que permite diseñar una ruta y hacer que la cámara se mueva siguiendo la ruta diseñada.

El diseñador quiere que la rotación de la cámara no cambie, es decir, que no mire hacia el jugador, pero con Dolly Track, como la cámara se mueve en una ruta fija, el jugador puede salir fuera de la pantalla o puede aparecer en las esquinas como si no fuera un elemento importante del juego. Al final hemos decidido usar una mezcla de **Dolly Track** y cámara fija: cuando el jugador está resolviendo un puzle entonces la cámara se queda fija y cuando se está desplazando por el nivel usa **Dolly Track**.

Para acabar, enumeramos otras funciones necesarias para el juego, pero que se pueden implementar con los componentes de **Unity** fácilmente:

1. Animación del personaje.

2. Sonido de pisadas del personaje.

3. El sonido de la pisada debe cambiar dependiendo del material del suelo.

4. Cambiar el color de la ropa del personaje cuando muere para dar sensación que es otra persona del pueblo.

5. Informar al jugador de cuándo puede interactuar con los objetos.

3.7.2. Interacción

Para empezar, tenemos que analizar cómo se controla el juego. Hay tres acciones en cada uno de los tres tipos de controles, aunque usan diferentes teclas: confirmar, cancelar y reintentar. Vamos a analizarlo por separado.

- Modo de dos botones, como el botón de confirmar lo hemos usado para mover, para interactuar tenemos que usar el botón de cancelar. Pero este botón también hay que usarlo para abrir el mapa, para no tener conflicto hemos decidido que si el jugador está cerca de un objeto interactivo, entonces el botón cancelar sirve para interactuar sino, para abrir el mapa.

- Teclado, en este tipo de control, el botón de confirmar no está ocupado como el modo de dos botones, por lo tanto usamos el botón confirmar para interactuar, y para abrir el mapa se usa el botón de cancelar.

- Ratón. Se usa interacción directa con el botón principal del ratón para accionar los objetos interactivos. El botón secundario se usa para abrir el mapa. Para soportar el uso de ratones de un sólo botón, también se puede abrir el mapa mediante un icono que se encuentra en la esquina superior derecha dela ventana como podemos ver en la figura 4.26.

En los objetivos se han presentado los tres tipos de objetos interactivos que hay en el juego: la caja, el coco y el ave. Además de estos objetos, el jugador también pueden interactuar con el entorno.

La figura 3.13 muestra el diseño de la caja, que define además los siguientes requisitos:

- La caja se puede empujar desde las cuatro direcciones.

- Hay una variante de la caja, donde se puede añadir un asa a cualquier dirección.

- Cuando estamos empujando, la caja debe ser influida por la física, porque la caja puede rellenar un agujero en el suelo para que el jugador pueda pasar.
- La caja debe poder interactuar con el interruptor que se encuentra en el suelo del nivel.

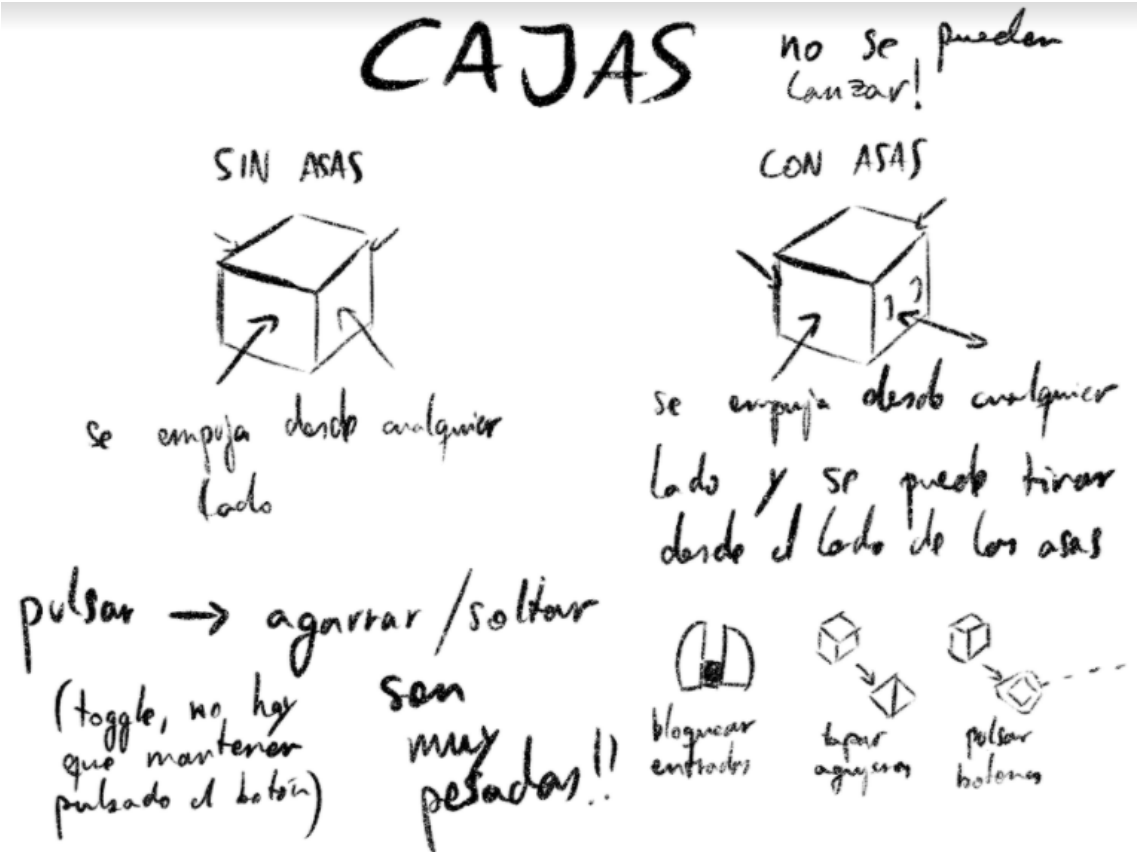


Figura 3.13: Diseño de la caja

La figura 3.14 muestra un detalle del diseño del primer nivel del juego. Aquí encontramos dos requisitos más:

- Cuando no estamos interactuando, la caja no participa en la simulación física.
- El jugador puede caer en el hueco cuando está empujando la caja.

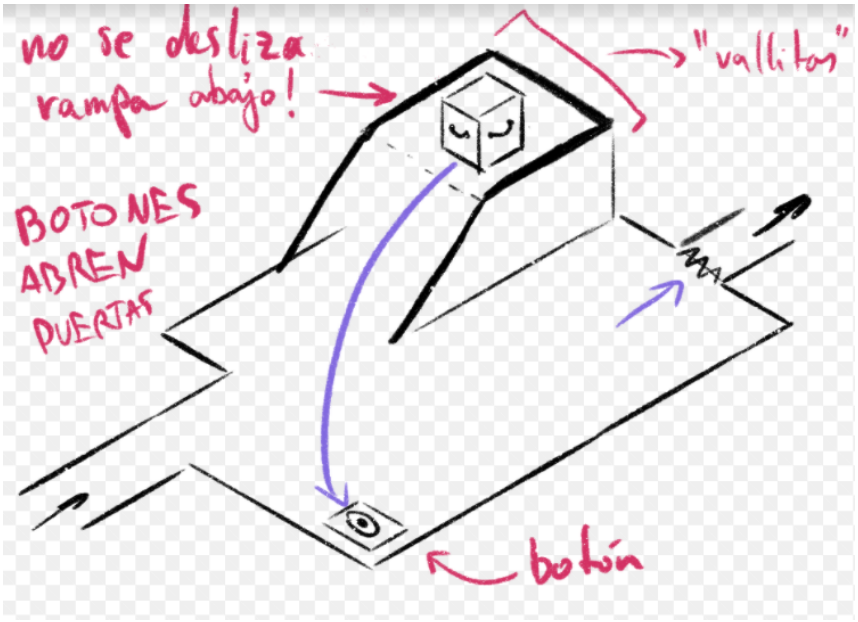
Una vez descritos los requisitos básicos, vamos a analizar el control. En el apartado funcionalidad básica hemos decidido mover el personaje con **Nav Mesh Agent**. Una propiedad de este componente es que ignora la física. Además siempre hace que el personaje mire hacia donde se mueve pero, cuando tiramos la caja, el personaje debe mirar hacia la caja y moverse hacia atrás. Por esa razón vamos a usar el componente **Transform** para empujar y tirar la caja.

Para los modos de entrada de teclado y de dos botones es fácil acercarse a la caja e interactuar con ella desde las cuatro direcciones. Sin embargo, como el jugador no puede moverse donde mira la cámara, es más difícil acercarse a la parte trasera de la caja utilizando el ratón. Así pues, hemos decidido que cuando se usa el ratón, al clicar sobre la caja, se muestran unas flechas para elegir con qué lado se quiere interactuar, como se ve en la figura 3.15.

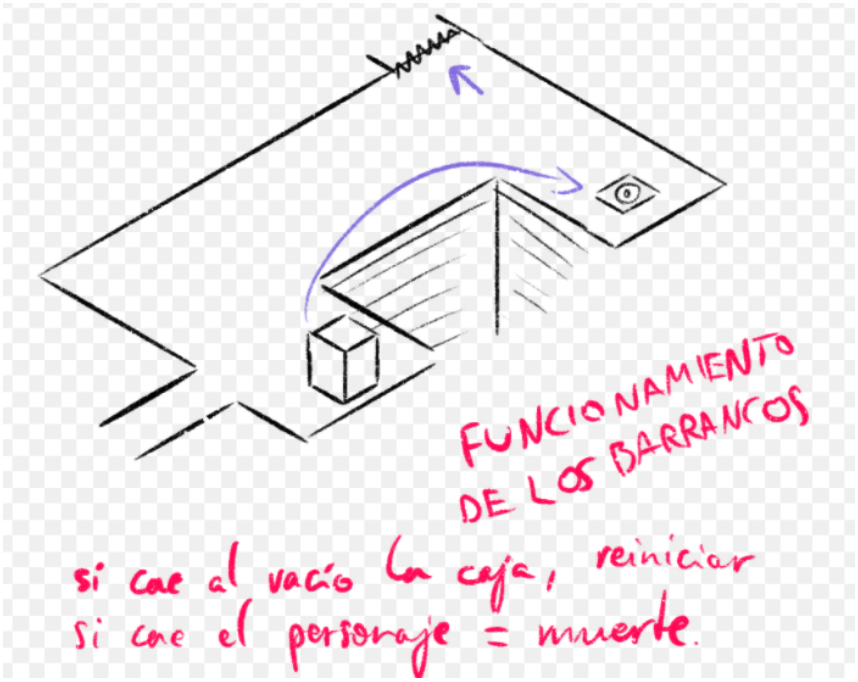
La figura 3.16 muestra el diseño del coco. Los requisitos son:

- Debe participar de la simulación física.
- Se debe poder coger por el jugador o por el ave.
- Se puede lanzar.

El control del coco también plantea problemas. Después de coger el coco, el jugador tiene que elegir la dirección en la que lanzarlo. Tanto con el ratón como con el modo de dos botones se puede elegir cualquier dirección, usando la misma forma usada para mover al personaje, pero el teclado solo puede lanzar a ocho direcciones. Para que el teclado tenga la misma precisión que los otros dos modos de interacción, hacemos que el personaje gire poco a poco hacia la dirección donde ha pulsado el jugador.



Diseño 1



Diseño 2

Figura 3.14: Detalle del diseño del nivel 1

La figura 3.17 muestra el diseño del ave y podemos encontrar siguientes requisitos:

- El ave puede agarrar un objeto pequeño.
- Cuando el ave tiene una cosa en su boca entonces el botón confirmar es para lanzar.

Figura 3.15: Selección del lado con el que se quiere interactuar

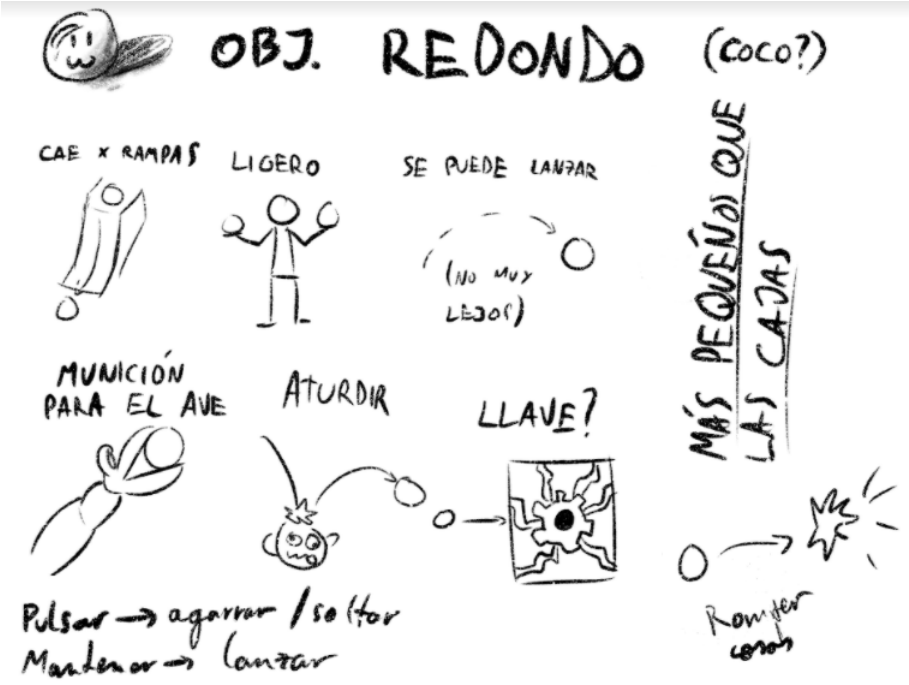


Figura 3.16: Diseño del coco

- Cuando el ave no tiene ninguna cosa en su boca entonces el botón confirmar es para agarrar.

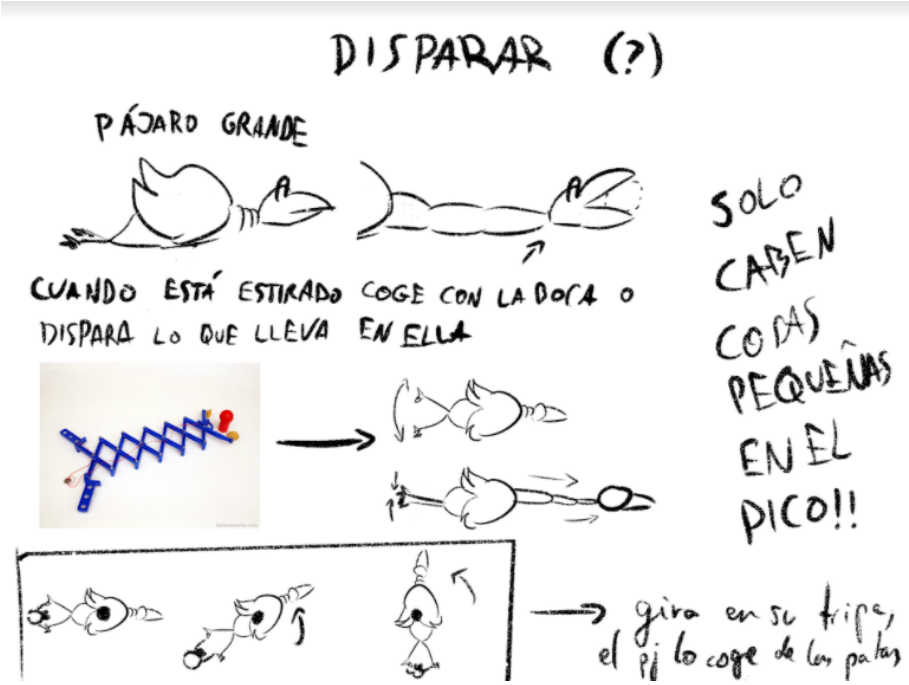


Figura 3.17: Diseño del ave

CAPÍTULO 4

Diseño e implementación

En este capítulo vamos a presentar la estructura del juego y explicar cómo hemos implementado cada clase. A causa de que el proyecto es grande, se ha dividido el capítulo en varios bloques.

4.1 Control

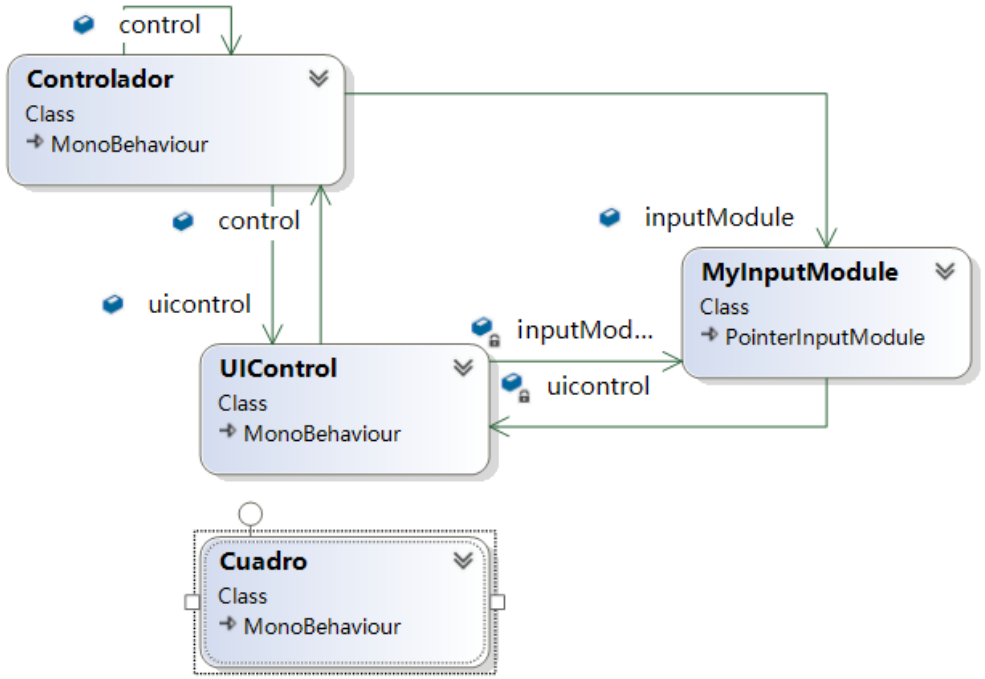


Figura 4.1: Diagrama de control

La figura 4.1 muestra la diagrama de clases del control, empezamos por **Controlador**, que es una clase que sirve para comunicarnos con las otras clases. El objeto **Controlador** puede estar en diferentes escenas. Si usamos el método *Find* para obtener la referencia el coste sería muy alto, por este motivo vamos a crear una variable estática, y asignamos la referencia al iniciar el juego. Así las demás clases pueden acceder a **Controlador** a través por esta variable estática.

En el controlador tenemos una variable de tipo booleano que se llama *controlable*. Cuando está variable está en falso, se ignoran los eventos generados por el jugador, y sirve para controlar el tiempo entre eventos. La función que usa la variable es **registrarControl**, que empieza una corrutina que espera *inputTime* (una variable para guardar cuánto tiempo hay que esperar después de que el jugador ha pulsado una tecla) y vuelve a poner la variable controlable a verdadero. Cuando *inputTime* es 0 entonces se espera hasta el final de un frame para no registrar ninguna entrada más.

La clase **MyInputModule** modifica a **Standalone Input Module** y sirve para manejar los eventos de entrada y para enviarlos a **Event System** para controlar los elementos de la interfaz. No tiene que ver con el control del jugador. Vamos a añadir variables de tipo KeyCode para guardar nuestras teclas. Además, también tiene un controlable para activar y desactivar el control a interfaz.

Para que el módulo funcione tenemos que modificar o añadir las siguientes funciones:

- *Process*, es la función que se ejecuta cuando se recibe cualquier evento de entrada. Tenemos que añadir una comprobación de si las variables controlables son verdadero.

- *GetMove*, esta función calcula la dirección que ha elegido el jugador. El valor que devuelve es un Vector2 para saber la dirección horizontal y vertical. Aquí lo modificamos para que tener en cuenta laa configuración del teclado.

- *ProcessMousePress*, es la función que gestiona si se ha pulsado un botón del ratón o no. Cuando está pulsado el botón derecho se llama a **Controlador** para que cierre el menú. Cuando el botón se suelta hay que llamar regis-

trarControl, ya que para que funcionen los botones deben recibir tanto los eventos de pulsación como de liberación.

- *SendMoveEventToSelectedObject*, cuando **Event System** no está seleccionando ningún botón, entonces elige el primer botón que hay en la pantalla.
- *Mappear*, es una función que hemos añadido para cambiar las teclas.

  La clase **Cuadro** es el bloque que se ha descrito en la función de navegación, y tiene un array de **GameObject**s para guardar los botones. También puede guardar otros cuadros.

  La clase **UIControl** es la clase que hace la función de navegación. Tiene una variable para guardar el objeto que está seleccionado. Cuando el módulo de entrada recibe un evento de confirmar o cancelar también informa a esta clase. En caso de confirmar, si el objeto que está eligiendo ahora es un **Cuadro**, entonces guarda la lista que está navegando ahora en una pila junto con el índice y navega el contenido del **Cuadro**. Como **Cuadro** no es un elemento de interfaz, para mostrar el fondo se tiene que hacer por código. En caso de cancelar, se mira en la pila, y si hay elementos entonces se navegará por la lista sacada de la pila. Si la pila está vacía, se informa a **Controlador** para pedir que cierre el menú o cambie la **Pagina**.

  Para la navegación, hemos usado una corrutina que, cada cierto tiempo, pide a **Event System** seleccionar un objeto. También se ha creado un método *apilarNavegacion*, para facilitar insertar algunas elecciones, por ejemplo, el jugador puede elegir cerrar juego, en este caso hay que navegar las opciones de si o no quiere cerrar el juego.

4.2 Sistema de guardado

La figura 4.2 es la diagrama de clases del sistema de guardar, la clase **Controlador**, aparte de para comunicar con otras clases, también sirve para iniciar los objetos, por lo tanto la información de configuración se almacena en **Controlador**.

DatosSistema es una clase para guardar los datos de las configuraciones y **Datosjuego** es para guardar valores del nivel y del jugador. En el apartado de

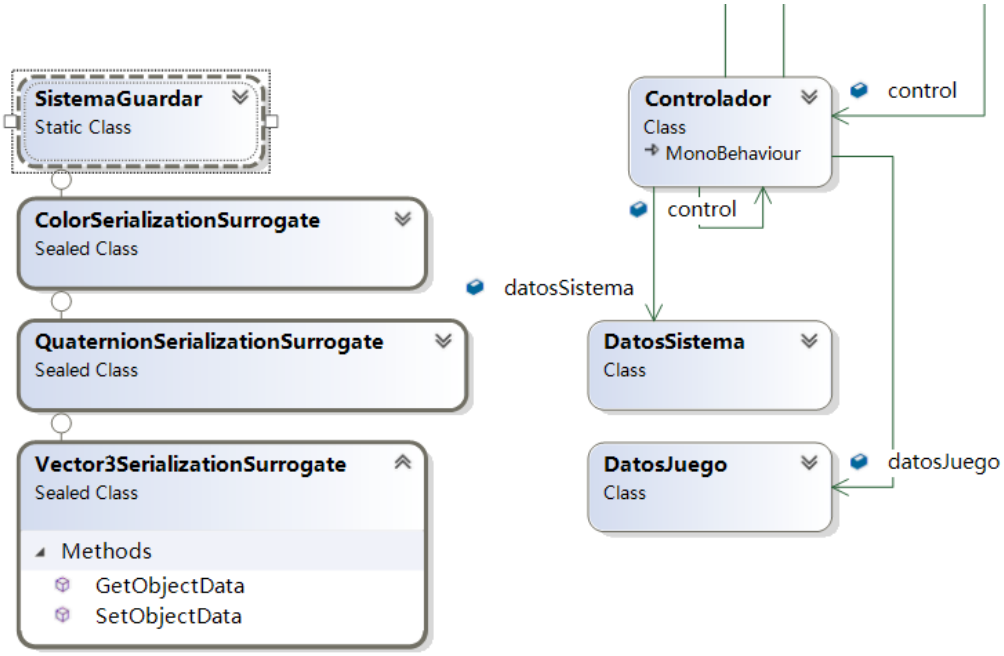


Figura 4.2: Diagrama de guardar

análisis hemos dicho que la serialización de C# no puede serializar las estructuras de **Unity**, por eso, hemos necesitado crear tres clases que heredan de **ISerializationSurrogate** y escribir en *GetObjectData* y *SetObjectData* qué datos hay que guardar y cómo leer los datos.

Para calcular el hash, hemos usado la forma fácil que es sumar el valor o hash de cada campo. Hay que tener cuidado con los objetos que no tiene un *GetHashCode* reescrito, porque van a usar el de **System.Object** y lo que devuelve es la dirección de la referencia. Como este valor es variable, cambiará de una ejecución a otra. Por ello hay que reescribir el método *GetHashCode* de los objetos que se deseen serializar.

SistemaGuardar es una clase estática a la que solo accede la clase **Controlador**. Sirve para guardar y cargar los datos, y también hace una comprobación de si la información ha sido modificada, y su caso, crea un objeto nuevo con los valores por defecto.

4.3 Menú principal

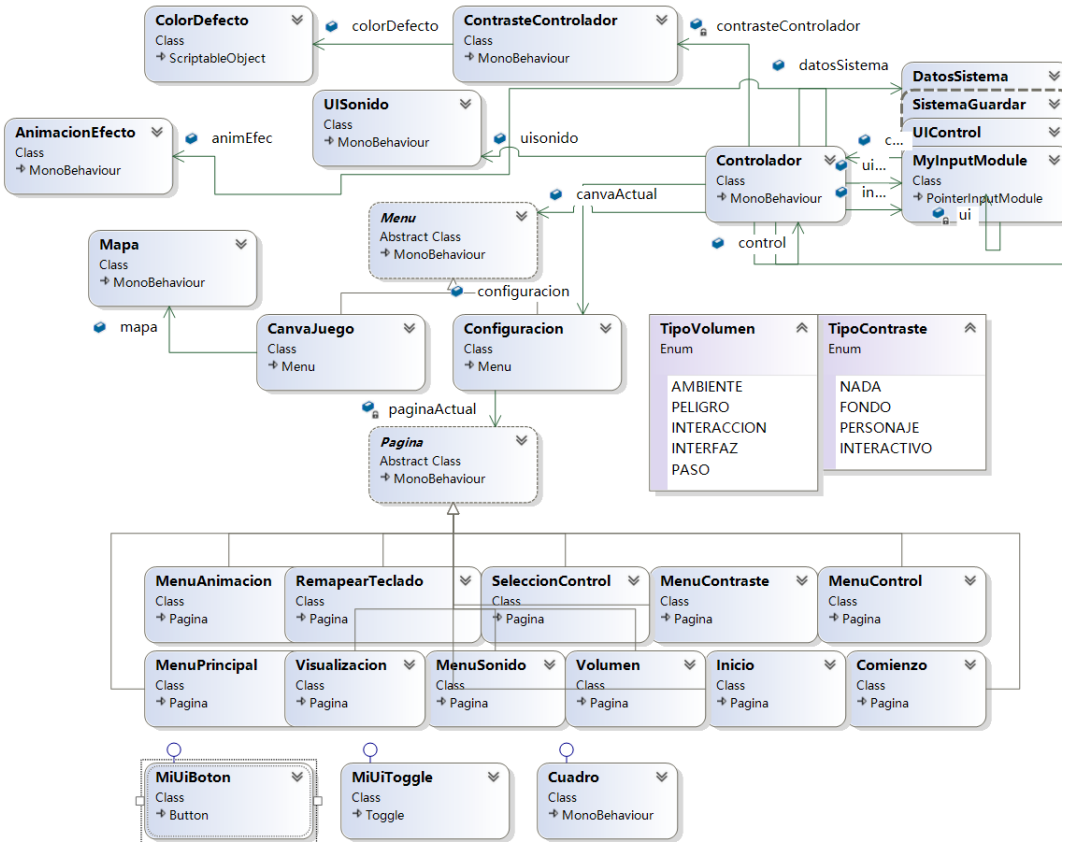


Figura 4.3: Diagrama de interfaz

En el juego vamos a tener dos interfaces: el menú principal y el HUD[1] del juego que podemos encontrar en la figura 4.3. En el apartado de los controles hemos explicado el método *registrarControl* que es una función que pone la variable controlable a falso durante un periodo de tiempo para evitar capturar entrada consecutiva, pero el jugador no sabe que no puede controlar, para dar una indicación al jugador hemos decidido añadir un icono de cargar(Figura 4.4) para informar al jugador que cuando hay este icono las teclas que ha pulsado por jugador no va a afectar al juego.

Hemos dicho que tenemos dos menús, esto significa que tenemos dos canvas diferentes, para no duplicar el mismo icono en dos canvas hemos creado un panel que guarda este icono y lo dejamos como hijo de **Controlador**. En **Controlador** guardamos qué menú está mostrando ahora en *canvaActual*, cada vez que

[1] https://es.wikipedia.org/wiki/HUD_(inform%C3%A1tica)

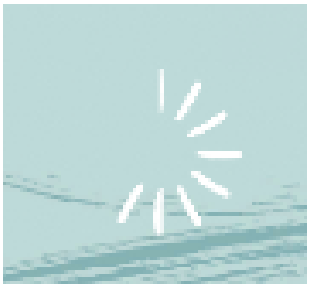


Figura 4.4: Icono de cargar

necesitamos mostrar el icono entonces lo ponemos en *canvaActual* y así evitamos duplicar recursos innecesarios.

A parte del panel de icono cargar, existen varios paneles adicionales:

- *panelCargar*, un panel negro con una animación de un barco, que se muestra cuando el juego está cargando.
- *panelElegirColor*, este panel contiene el panel de selección de color. Aunque por el momento solo se usa en el menú de configuración, lo hemos dejado en **Controlador** por si en el futuro se necesita seleccionar color en otros lugares del juego.
- *panelMuerto*, es el panel que se muestra cuando el jugador muere.
- *panelSalir*, es un panel que pregunta al jugador si realmente quiere cerrar el juego.
- *panelTutorial*, es un panel para mostrar las imágenes y textos del tutorial.

UISonido es una clase para guardar los sonidos de la interfaz. Cuando el módulo de entrada recibe los eventos de mover el foco entre botones, pide a **UIControl** reproducir el sonido a través de **Controlador**.

Menu es una clase abstracta, con cuatro funciones:

- *abrirMenu*, este método recibe un parámetro de tipo int, porque el menú principal se puede abrir de varias formas. La primera vez que se inicia el juego se abre la ventana de configuración para seleccionar el tipo de control. En caso de que no sea la primera ejecución, se abre el menú de inicio.

Cuando se abre el menú dentro del juego, hay que abrir la página de configuración.

- *cancelar*, el método cancelar de **Controlador** va a preguntar a **UIControl** si existen elementos en su pila. En caso de que esté vacía, devuelve falso, ya que **Controlador** va llama a cancelar de **Menu** para volver a la página anterior.

- *cerrarMenu*, este método también recibe un parámetro de tipo int, porque para empezar el juego y cerrar la configuración va a tener diferentes comportamientos.

- *getAcutalNavegable*, es un método para cuando **UIControl** no sabe qué control hay que navegar, por ejemplo, cuando se cambia el control al modo de dos botones.

Menu tiene dos hijos, uno es **CanvaJuego** que es el HUD y explicamos después, y otro es **Configuracion**, que es el menú principal del juego, y todas las páginas de configuraciones. Tiene un método *abrirPagina* que se usa para cambiar la página de configuración. Guarda la página activada en una pila, cuando **Controlador** llama a *cancelar* comprueba si existe una página en la pila, y si está vacía cierra el menú y no hace nada.

Se ha decidido permitir al jugador controlar el menú con los tres métodos de control, ya que si el jugador cambia el tipo de control sin querer, no podría volver al control anterior, sobre todo personas con movilidad reducida. Cuando se hace clic con el ratón en un lugar donde no hay nada, **EventSystem** establece la variable del objeto seleccionado a nulo. Se ha hecho que, cuando no hay nada seleccionado, se seleccione el primer objeto, y el primer objeto también lo asigna en *abrirPagina*.

Pagina es una clase abstracta para implementar las páginas de configuración. Tiene una variable *menu* para guardar objetos de la clase **Configuracion**, una lista de **GameObject** para guardar los elementos que se pueden navegar con **UIControl** y también tiene un *primerElegido* para guardar el primer botón de la página.

Pagina tiene un método que se llama *inicializar*, donde se inicializa qué botón debe estar activado y donde se modifican los tamaños y fuentes de texto. Para no crear copias para cada página, las fuentes se guardan en **Controlador**. Cuando una página lo necesita, accede a la variable estática para pedirla.

Como se ve en la figura 4.3 hay muchas clases que se derivan de **Pagina**. A continuación las presentamos una por una y explicamos sus funciones.

- **Inicio**(Figura 4.5), es la página que se muestra cuando el jugador inicia el juego. Tiene las acciones de jugar, opciones y salir. Jugar y opciones consiste en ir a otras páginas, pero salir lanza una llamada a la función *cerrarJuego*. Como cerrar el juego es una función que se puede realizar en cualquier momento del juego, ponemos la función en **Controlador**.

  Para que se pueda cerrar el juego cerrando la ventana, al iniciar el juego tenemos que añadir la siguiente llamada a [13], que se ha situado en el método Start de Controlador.

```
1 Application.wantsToQuit += cerrarJuego; //agregar una callback a
    wantsToQuit
```

  cerrarJuego es un método que no tiene parámetros y devuelve un booleano. Cuando devuelve verdadero, entonces **Application** cerrará el juego. Como queremos que se muestre una ventana de confirmación como se ver en la figura 4.6, entonces mostramos el panel de salir con la fuente y el tamaño definidos en la configuración y llamamos a *apilarNavegacion* de **UIControl** para que el usuario pueda navegar entre los botones de sí y no. Como el jugador puede intentar volver a cerrar la ventana, creamos una variable que se llama *cerrandoJuego*, que está en verdadero cuando el panel de salir está activado. Por lo tanto, *cerrarJuego*, antes de abrir el panel de salir va a comprobar el valor de *cerrandoJuego* y, en caso de que sea cierto, directamente devuelve falso para acabar la función.

  El siguiente paso es implementar otra función que se encargue de cerrar realmente el juego, y que enlazamos a los botones de si y no. Si el jugador elige si, se guardan los datos del juego para no perder el progreso del jugador y se llama a **Application.Quit**. Esta función llamará otra vez a *cerraJuego*

sin parámetro. Por lo tanto vamos a añadir otro booleano más, que se llama cerrar, y que cuando sea verdadero entonces **Application** cierre el juego. El valor de la variable lo establecen los botones de sí y no.

Figura 4.5: Inicio

Figura 4.6: Panel de cierre del juego

- **Comienzo**(Figura 4.7), es la página para elegir empezar una nueva partida o continuar. En su función de inicialización pregunta a **SistemaGuardar** si existen datos del juego, y en caso de que no existan, entonces desactiva la opción de continuar. En esta clase se va a llamar a *NuevaPartida* y a *Continuar* de **Controlador**, pero el contenido de las funciones se explicarán más adelante.

Figura 4.7: Comienzo

- **MenuPrincipal**(Figura 4.8), es la página principal de configuración. Desde aquí se puede ir a cualquier al resto de páginas de configuración.

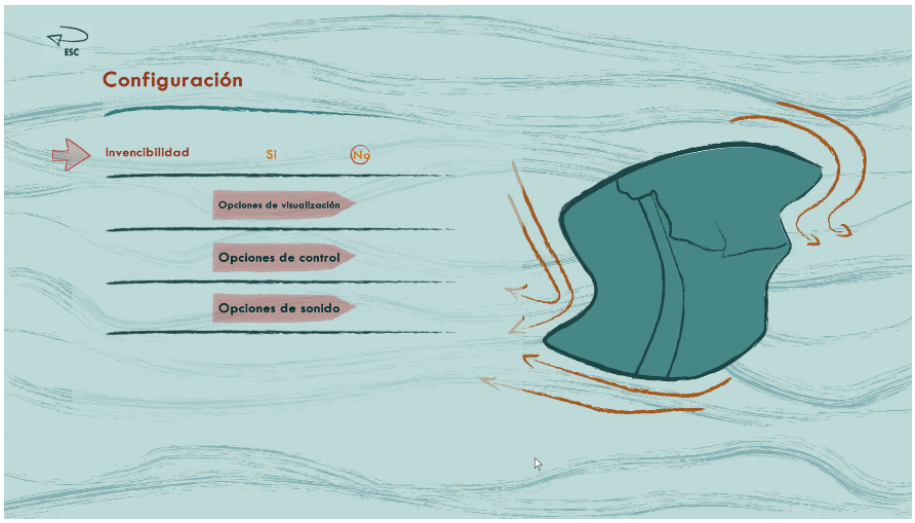


Figura 4.8: Página de configuración principal

- **Visualizacion**(Figura 4.9), es la página de las opciones relacionadas con los gráficos. Pantalla completa, como es una función que necesita reiniciar el juego, se implementa en Controlador. Se modificará la propiedad *fullScreen* de **Screen** y se guardará el valor.

  Cómo el cambio del tamaño y fuente ya se ha implementado en la función inicializar, aquí solo vamos a modificar los datos que hay en **DatosSistema**, y volvemos a inicializar otra vez la página actual.

Para cambiar el cursor también vamos a llamar a una función de **Controlador**. Se ha dividido el nombre de los ficheros con las imágenes de los cursores en dos partes: un prefijo para indicar el tipo de cursor y un sufijo para indicar el tamaño. Gracias a eso, se puede calcular el nombre del fichero a cargar a partir de las propiedades seleccionadas, una para indicar el tipo y otra para indicar el tamaño. Para cambiar el cursor se usa *SetCursor* de **Cursor**, y se ha seleccionado el modo a *ForceSoftware* para poder cambiar de tamaño.

La imagen del cursor debe ser múltiplo de 32, ya que si no, la imagen se deforma.

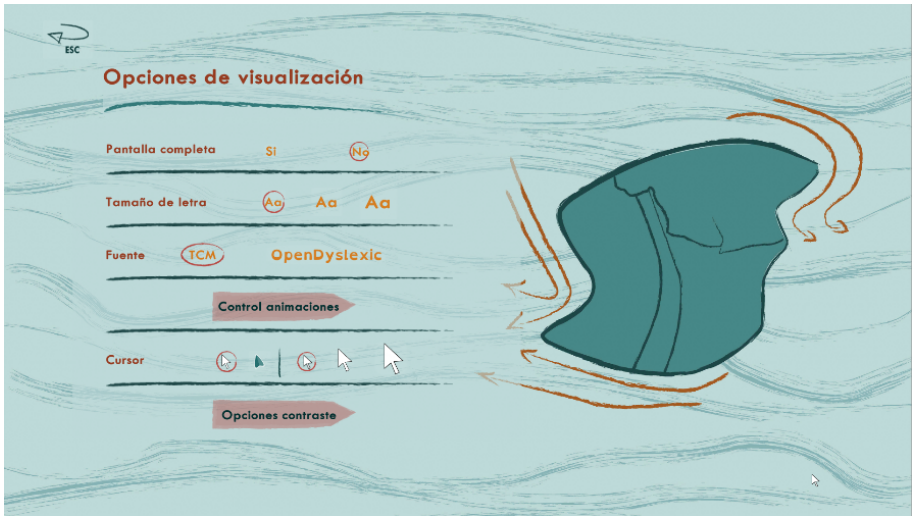


Figura 4.9: Visualización

- **MenuContraste**(Figura 4.11), es la página para cambiar la gama de color de los objetos. Para abrir el panel llamaremos a la función *ElegirColor* de Controlador. El método recibe dos parámetros, uno es la posición donde debe aparecer el panel y otro es una **Action** que recibe *Color* como parámetro, es decir, pasamos una función como argumento, que recibirá el color seleccionado por el jugador.

  La navegación(**UIControl**) que hemos diseñado solo sirve para navegar entre los objetos. Para poder elegir color en el modo de dos botones, tenemos que implementar otra función. Como el panel de elegir color es un objeto hijo de **Controlador**, vamos a incluir el código en *Update* de **Controlador**.

Figura 4.10: Panel de selección de color

Antes de la implementación vamos a analizar el panel. En la figura 4.10 podemos encontrar cuatro elementos:

1. Una marca de verificación representa el botón de confirmar que solo es necesario para ratón.
2. Un cuadrado para mostrar el color que está elegido.
3. Una paleta que muestra el canal H de la escala de color HSV.
4. Un puntero para elegir color.

La forma de elegir color la podemos dividir en dos partes: mover el puntero y mostrar el color donde está el puntero. La segunda parte es igual para todo los tipos de controles: tenemos que obtener el píxel donde está el puntero, y para que podamos leer los píxeles de la imagen tenemos que activar el atributo **Read/Write Enabled**.

Como el panel solo muestra un canal del color, solo tenemos que mover el puntero horizontalmente y cuando el puntero llega al final del panel, volver al principio.

La paleta de colores la hemos creado con **OpenCV**.

Para los controles con ratón y teclado hay que tener cuidado con los bordes para que el puntero no se salga del panel.

Como cambiar el color no es una función que se suele usar a menudo y para cambiar el color tenemos que cargar todos los materiales, hemos creado un prefab[2] que solo se instancia cuando se va a cambiar el color, y una vez cambiado se borra el objeto y se libera la memoria.

El prefab tiene un componente **ContrasteControlador** que va a guardar todos los materiales por tipo. También hemos reescrito la representación del **Editor** de este componente, para tener un botón que puede guardar todos los colores actuales en un **ColorDefecto** que es un **ScriptableObject** (objetos que se pueden guardar en un fichero)[3]. También hemos añadido un botón para cambiar el color por defecto, porque los cambios hechos en los materiales en el editor se van a guardar, pero en el juego compilado no. Por lo tanto cada vez que se usa esta función en el editor tenemos que deshacer los cambios.

El método que realiza esta función es *cambiarContraste*. Como cada vez vamos a instanciar y borrar los objetos, para que solo necesite llamarse una vez hemos creado una clase *enumerada* **TipoContraste** que tiene activado el atributo *Flags*, lo que nos permite usar máscaras para saber qué color hay que cambiar.

Por ejemplo, **MenuContraste** tiene una variable *colorCambiado* de **TipoContraste**. Inicialmente está a nulo, y cada vez que el jugador modifica un color, vamos a realizar una operación **OR** a la variable *colorCambiado*. Cuando cerramos esta página, pasamos *colorCambiado* a *cambiarContraste*, y la función puede comprobar con el siguiente código si ha cambiado:

```
1 if ((TipoContraste.FONDO & mascara) == TipoContraste.FONDO) //
    comprobar si el bit de FONDO es 1
```

Para activar la función dicromática tenemos que llamar *modoDicromatico* de **Controlador**. Este método simplemente activa o modifica el modo de la clase **ColorBlindCorrection** de **Post Process Volume**.

[2] https://docs.unity3d.com/Manual/Prefabs.html

[3] https://docs.unity3d.com/Manual/class-ScriptableObject.html

Figura 4.11: Contraste por categoría

- **MenuAnimacion**(Figura 4.12), es la página donde se pueden desactivar las animaciones, pero solo vamos a dejar desactivar las animaciones auxiliares. En el nivel 1 no tenemos ninguna animación de peligro o NPC (non-player character) y por lo tanto solo vamos a cambiar el valor en **DatosSistema**. Tenemos una animación de fondo que es el movimiento del agua, implementada con un shader, por lo que hemos hecho lo mismo que para cambiar el color: guardamos el shader en la clase **AnimacionEfecto**, y en caso de haya que desactivar la animación, ponemos el campo del shader a 0.

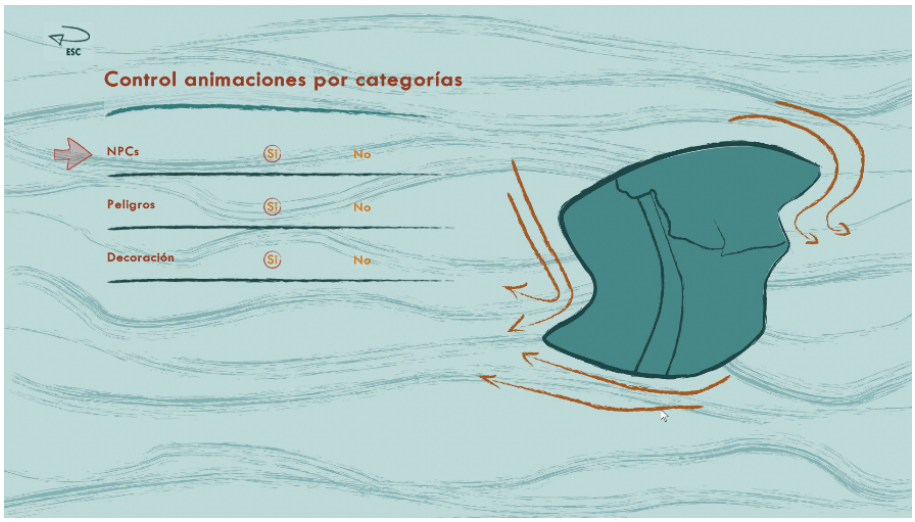


Figura 4.12: Control animaciones por categorías

- **MenuControl**(Figura 4.13), aquí están las opciones relacionadas con los controles. Para cambiar la forma de control hay que llamar *CambiarControl* de

Controlador. Esta función también se necesita en la para inicialización, por lo que la dejamos en Controlador. Vamos a cambiar el valor de **DatosSistema**, y activar **UIControl** en caso de que el modo de entrada actual sea el de dos botones. El control del jugador se gestiona en la clase **Player**. En caso de que haya empezado la partida, hay que pedir a **Player** que actualice los controles.

Cambiar el tiempo mínimo entre eventos solo necesita cambiar el valor de **DatosSistema**, por lo que se implementará en esta misma clase. Cambiar velocidad solo afecta si el jugador ha empezado la partida, por lo que en ese caso hay que avisar a **Player**.

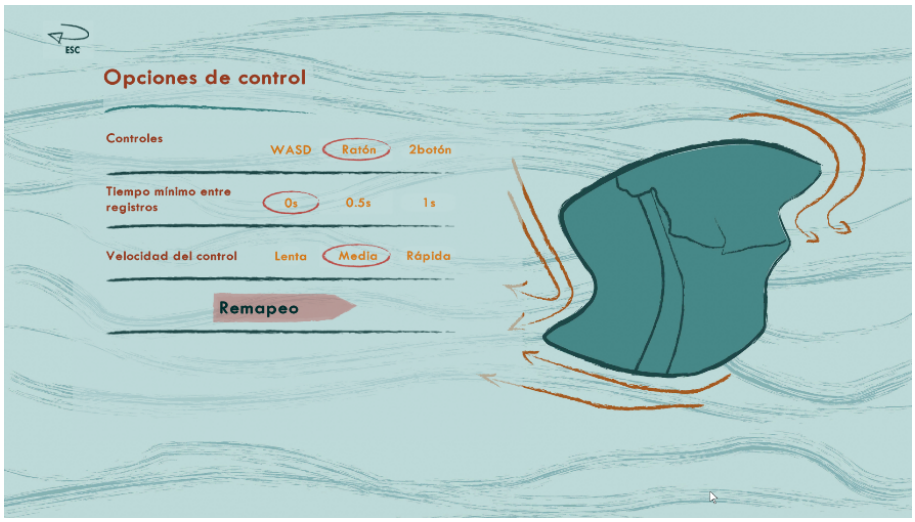


Figura 4.13: Opciones de control

- **Remapear**(Figura 4.14), aquí en realidad hemos creado dos páginas: una para el teclado y otra para el modo de dos botones.En **Controlador** hemos creado un diccionario para guardar las teclas. En la inicialización se leen las teclas y se muestran los botones. En esta clase tenemos dos diccionarios: uno para comprobar si la tecla ya está asignada, y en su caso hay que intercambiar dicha tecla. El otro guarda la referencia de los textos de los botones para luego actualizar los textos.

  Para capturar la tecla hemos usado el evento **OnGUI**, pero hay que tener cuidado, ya que este evento es costoso porque puede ejecutarse varias veces en un ciclo [14]. Si el jugador pulsa dos teclas entonces se va a ejecutar

dos veces. Por ello vamos a crear una variable para controlarlo. Una vez obtenida la tecla ponemos esta variable a falso para no capturar más.

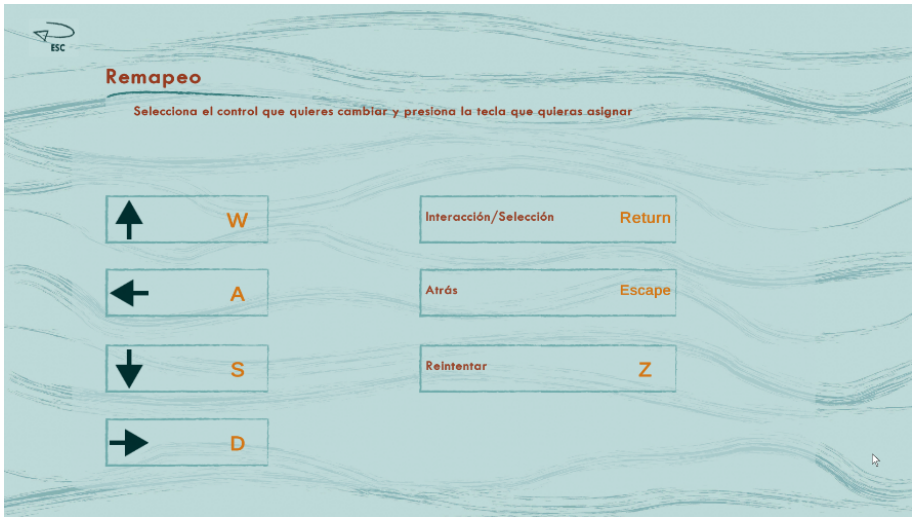


Figura 4.14: Remapeo

- **MenuSonido**(Figura 4.15), esta página solo tiene una opción para cambiar el modo de sonido entre mono o estéreo. Llama a *CambiarModoSonido* de **Controlador**, y el método cambia la propiedad *speakerMode* de **AudioSettings**

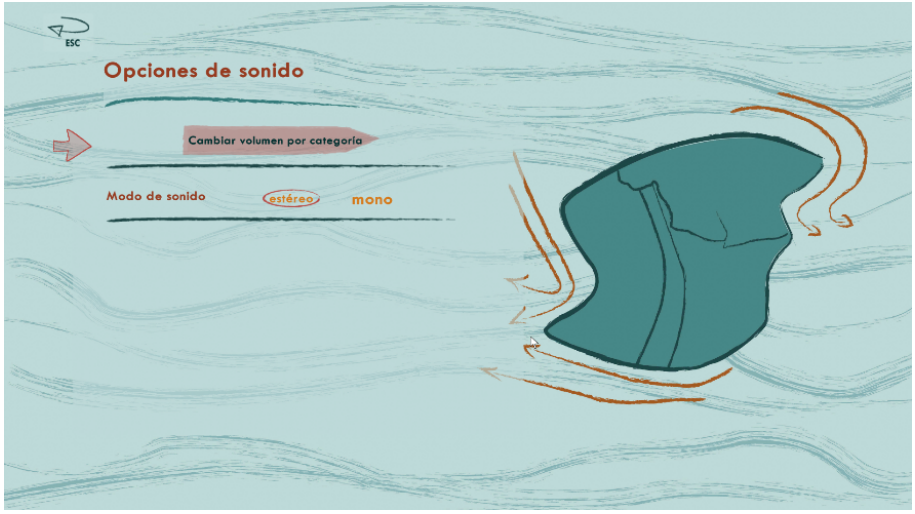


Figura 4.15: Opciones de sonido

- **Volumen**(Figura 4.17), esta página permite al jugador cambiar el volumen de cada tipo de sonido por separado. El componente que produce el sonido es **AudioSource**, y la salida del sonido puede pasar por un Audio Mixer.

Se ha creado un grupo *Master* y cinco subgrupos como se puede ver en la figura 4.16.

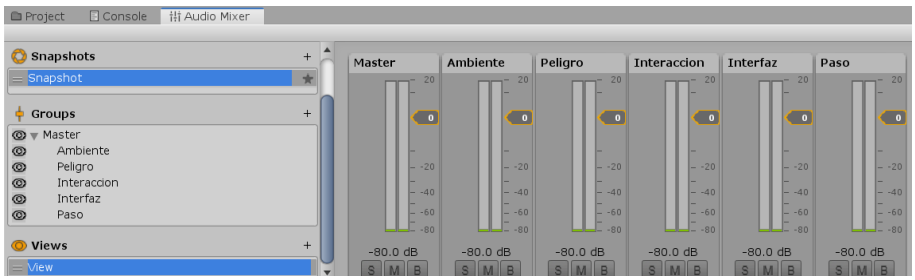


Figura 4.16: Audio Mixer

Para cambiar el volumen hay que llamar a *ModificarVolumen* de **Controlador**. El método necesita dos parámetros: uno para indicar qué grupo y otro para indicar el volumen. El editor de **Unity** no permite asignar una *callback* con dos parámetros, por lo que en *ModificarVolumen* de **Volumen** vamos a recibir un string, y vamos a separar ambos parámetro con una coma, y antes de llamar a Controlador separar la cadena, convertirla en un int y en un float (tipo y volumen).

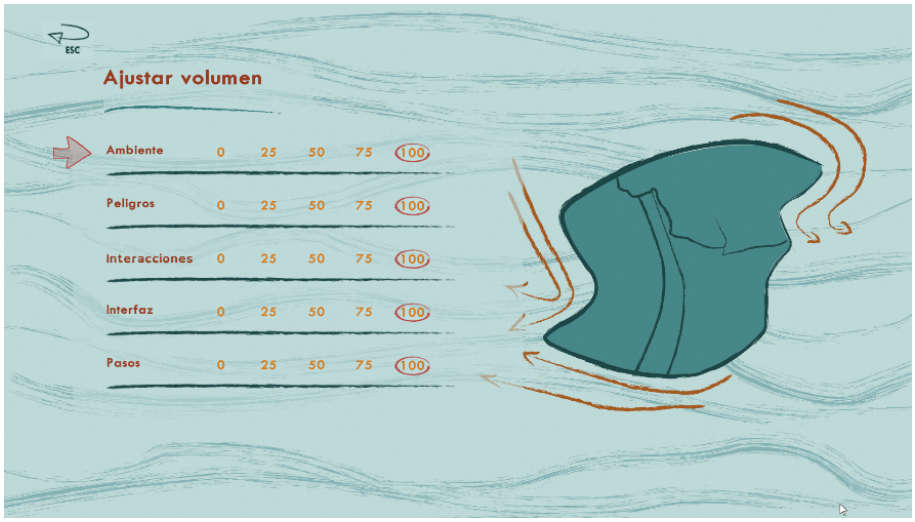


Figura 4.17: Ajustar volumen

- **SeleccionControl**(Figura 4.18), es la página que se muestra la primera vez que se inicia el juego. El juego empieza con el control de modo de dos botones, y en esta página el jugador puede elegir la forma de control preferida. Una vez elegida ya no aparecerá esta página. Se llama a la misma función que llama **MenuControl** para cambiar control.

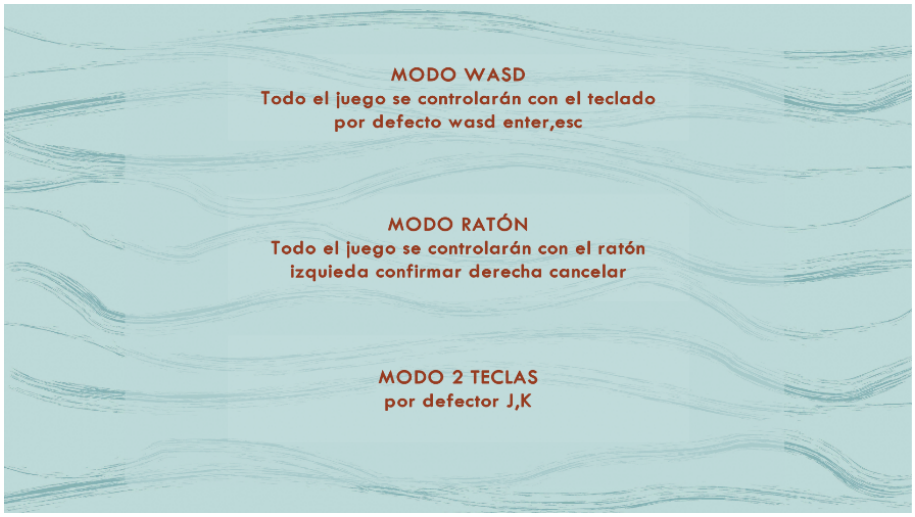
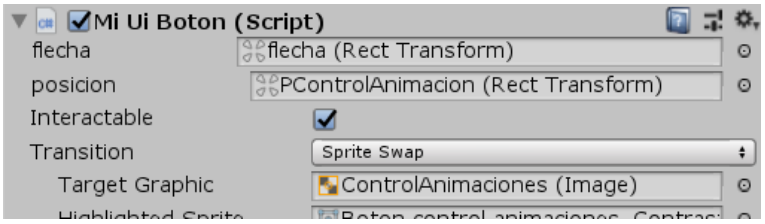
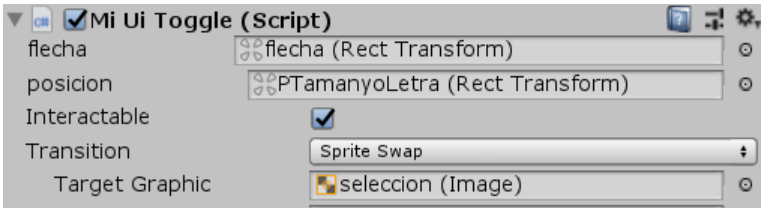


Figura 4.18: Configuración inicio



Editor de MiButton



Editor de MiToggle

Figura 4.19: Editor modificado

En interfaz de usuario hemos dicho que necesitamos una flecha de indicación, y por defecto los botones de **Unity** no lo permiten, por lo que hemos creado **MiUiBoton** y **MiUiToggle** que hereda de **Button** y **Toggle**. Nuestras clases tratan los eventos *OnSelect*, *OnDeselect*, *OnPointerEnter* y *OnPointerExit*. Hay que implementar cuatro clases: **ISelectHandler**, **IDeselectHandler**, **IPointerEnterHandler** e **IPointerExitHandler**. A las nuevas clases se añaden dos variables **RectTransform**, una para guardar la flecha y otra para mover la flecha a esa posición. También se ha aumentado el editor para trabajar con estas dos clases, como podemos ver en la figura4.19.

4.4 Jugador

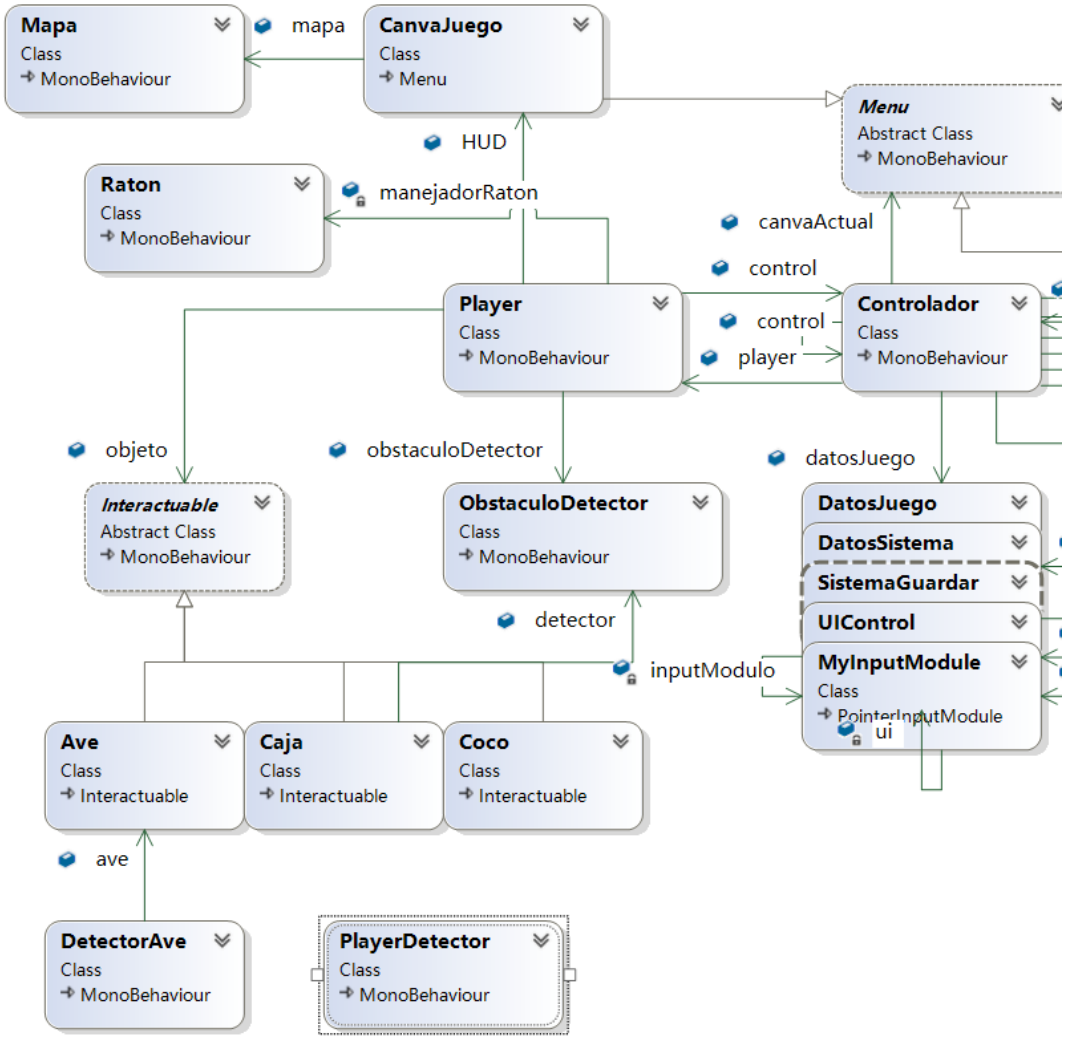


Figura 4.20: Diagrama del jugador

En este apartado vamos a explicar las clases relacionadas con el jugador(Figura 4.20). La clase **Player** es la clase principal. En funcionalidad básica hemos dicho que queremos que la selección de estado y selección de control tengan un coste constante. Para ello se han usado las clases **Func** y **Action** de C#. **Action** se usa para las funciones que no devuelve valores, y la usamos para los estados. Vamos a mantener una variable *estadoActual* y *controlable*, siendo esta última para pausar los movimientos del jugador cuando se abre el mapa. *estadoActual* es una variable de tipo **Action**. En el método de **Update**, si los controlables están a true se ejecutará *estadoActual*.

También tenemos creado un **Action** para cada estado. Cuando cambia el estado solo tenemos que asignar el estado al que queremos cambiar a *estadoActual*. Así podemos implementar tantos estados como se desee.

Para que la selección de control también sea de coste constante hemos creado un **Func<bool>** para cada acción. Una acción puede ser mover, cancelar, elegir dirección, etc. Las acciones no reciben ningún parámetro pero devuelve un booleano, este booleano va indicando si hay que cambiar el estado o no. Para inicializar hemos creado un método *actualizarControl* que, dependiendo del tipo de control, va a asignar acción correspondiente. Esta función sí que tiene un coste lineal y depende del número de tipos que hay. Como no hay muchos tipos y este método solo se llama al inicializar el jugador o cuando se cambia el tipo de control, el coste no es tan importante.

Como el número de estados no influye en el coste temporal, se van a crear muchos estados para poder organizar mejor el juego. Aparte de *estadoActual*, también hemos creado una clase enumerada dentro de la clase **Player** para saber en qué estado está el jugador.

Antes de analizar cómo implementamos cada estado, vamos a diseñar la estructura de la máquina. Como podemos ver en la figura 4.21, tenemos un estado parado que es el estado de entrada. Desde este estado podremos iniciar las acciones disponibles. Antes de interactuar con cualquier objeto pasamos por un estado preparar interacción. En este estado podemos realizar cualquier configuración previa. Para añadir más objetos interactivos solo tenemos que añadir más estados después del estado de preparar.

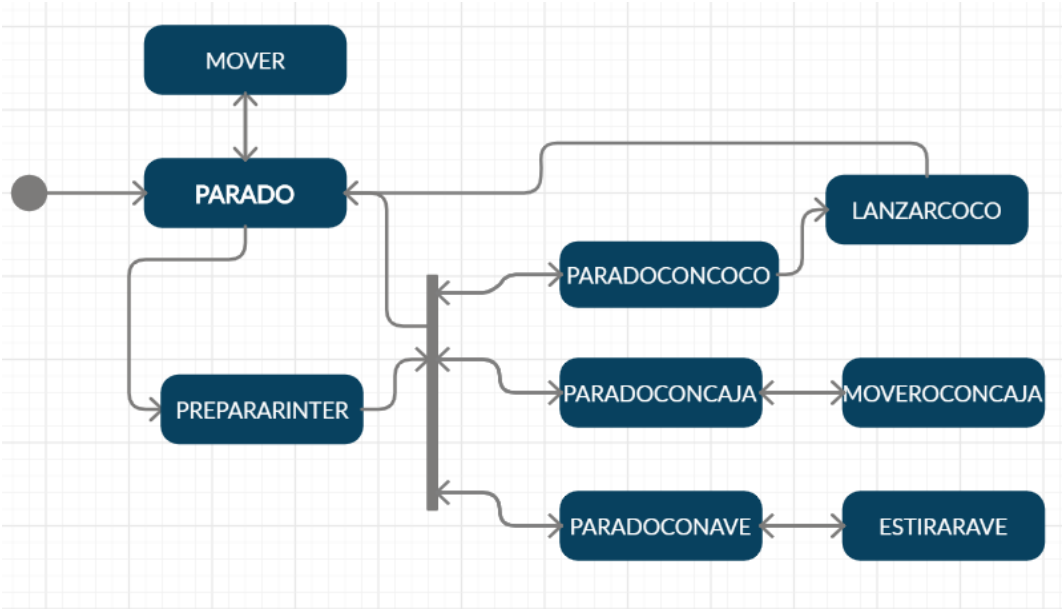


Figura 4.21: Máquina de estados del jugador

En la figura parece que elegir un tipo de interacción tiene un coste lineal según el número de objetos interactivos, pero en realidad también es de coste constante. Vamos a ver en concreto cómo es cada estado:

- *PARADO*, es el estado cuando el jugador está parado. En este estado tenemos cuatro acciones, mover, interactuar, abrir el mapa y reintentar.

  accionMover, es la acción que sirve para mover el jugador. Para el movimiento con el teclado, *moverJugadorWASD*, primero tenemos que ver qué dirección ha pulsado el jugador. El método *obtenerMovimientoWASD* devuelve un *Vector3* donde el campo x representa el movimiento en horizontal y el campo z en vertical.

  Esta acción es reutilizable, si el movimiento no es 0, entonces el jugador entra en el estado de mover. Si el jugador está en el estado de parado, hay que devolver true para cambiar el estado, y si el jugador está en el estado mover, hay que devolver un falso para no cambiar el estado. Es decir, cuando el movimiento no es 0 hay que devolver si estado es igual al *Estado.PARADO* y cuando el movimiento es 0 hay que devolver si el estado es igual a *Estado.MOVER*.

  Para mover con el agente tenemos que asignar un destino, que se calcula como **la posición del jugador + dirección que ha pulsado el jugador ***

una distancia La distancia puede ser cualquier valor suficientemente grande, porque si el valor es muy pequeño el jugador se va a mover y parar repetidamente.

La posición se obtiene directamente de Transform.

El principal problema en este apartado es calcular la dirección en el mundo a la que quiere ir el jugador, porque esto depende de la rotación en el eje Y de la cámara. Es decir, hay que rotar la dirección seleccionada por el usuario en el mismo ángulo de la cámara en el eje Y. En el editor de **Unity** podemos modificar la rotación de cada eje y esto se llama **EulerAngle**, pero eso solo funciona en el editor. La rotación de **Unity** se representa con **Quaternions**[15], por lo tanto si sacamos rotación del eje Y de la cámara no va a funcionar. Para resolver el problema hemos creado una propiedad *rotacionCamaraXZ* para entender mejor veamos directamente el código.

```
1  public Quaternion rotacionCamaraXZ
2  {
3  // solo calcula la rotacion si alguien le ha llamado
4      get
5      {
6      // para saber donde esta mirando la camara
7          Vector3 dst = mainCamera.transform.forward;
8          // proyectamos el vector dst en un plano de xz
9          dst.y = 0;
10
11 //cuando vector up de la camara menor que 0 significa la camara
12 //esta al reves y tenemos que invertir su forward
13         if (mainCamera.transform.up.y < 0)
14         {
15             dst.z = -dst.z;
16             dst.x = -dst.x;
17         }
18         // generar una rotacion desde (0,0,1) a donde esta mirando
               la camara
19         return Quaternion.FromToRotation(Vector3.forward,dst);
20     }
21 }
```

Una vez tenemos la propiedad anterior, para girar solo tenemos que normalizar el movimiento (porque para movimientos en diagonal (1,0,1), el vector no es normal) y luego multiplicar *rotacionCamaraXZ* por el vector de movimiento normalizado.

Para gestionar los eventos del ratón hemos creado una clase **Raton**, porque cada vez que el jugador pulsa el botón izquierdo hay que lanzar un *Raycast* para ver si ha pulsado un objeto o no. Con esta clase podemos desactivar el cálculo cuando el jugador está usando otros dos tipo de control y así no gastar recursos de computación extra. En la clase vamos a tener una variable *posiciónPantalla* para guardar la posición del ratón en la pantalla. Esto no es necesario en el editor, pero en la versión compilada y si el jugador ha cambiado la resolución del juego hay que proyectar a la resolución 1920x1080 para que algunas funciones se comporten correctamente. Una variable *posicionEscena* guarda la posición de la escena donde ha pulsado el jugador. Una variable *interactuable* indica si se puede interaccionar con el objeto, porque el jugador no va a poder interaccionar con el coco que, poro ejemplo, ha agarrado un ave. Una variable *objeto* de tipo **Interactuable** guarda el objeto seleccionado.

moverJugadorRaton, cuando el jugador pulsa el botón izquierdo del ratón, comprueba si no se ha pulsado en un objeto interactivo, y entonces asigna *posicionEscena* como destino de agente.

moverJugadorDosBoton, esta acción se divide en dos pasos: uno para elegir la dirección y otro para elegir la distancia. Como el punto para elegir la distancia es un proyector ortogonal, lo dejamos a bastante altura, y cuando el jugador ha confirmado la distancia, realiza un *Raycast* hacia abajo para calcular el destino del agente.

accionInteractuar, esta acción devuelve true cuando el jugador puede interactuar. Todos los objetos interactivos tienen un objeto hijo con un componente **PlayerDetector**. Cuando el jugador está cerca del objeto, llama a *asigObjeto* de **Player**. Esta función se encarga de asignar el objeto a **Player**, eligiendo el objeto que está más cerca al jugador. Además, para reducir la posibilidad

de error y ahorrar cálculos, solo se asigna cuando está en estado parado o mover.

interactuarWASD e *interactuarDosBoton* en el caso de que el jugador pulse el botón de cancelar o confirmar y el objeto seleccionado sea distinto al nulo, en caso de poder interactuar se cambiar al estado preparar interactuar.

interactuarRaton, las acciones con ratón son un poco diferentes. Se va a implementar las cuatro direcciones mencionadas en el apartado de Interacción. Cuando el objeto pulsado no es la caja, entonces se hace lo mismo que con los otros tipos de control, pero en caso de caja vamos a llamar *mostrarDireccion* de **CanvaJuego**. Esta función muestra cuatro direcciones y cada dirección está enlazada con *preparaInteractuarCaja* de **Player**. Para cambiar a estado de preparar. Para trabajar con las direcciones de interacción con la caja también tenemos que usar la misma técnica que usada para mover el jugador con teclado, pero esta vez tenemos que proyectar al plano x e y, y también hay que aplicar una rotación de la caja.

accionCancelar y *accionReintentar* son dos acciones que detectan si el jugador ha pulsado el botón o no. *accionCancelar* es una acción reutilizable, en estado parado se usa para abrir el mapa.

- *MOVER*, es el estado cuando el jugador se está moviendo. En este estado solo tenemos una acción, que es la acción de cancelar el movimiento. Hay que determinar si el jugador quiere dejar de moverse o ha llegado su destino.

  En el control del teclado podemos reutilizar *moverJugadorWASD*, añadiendo que cuando el jugador no pulsa ninguna tecla hay que restaurar el agente para que pare directamente.

  paraJugadorRaton, en modo de ratón comprueba si se ha pulsado el botón derecho y si ha llegado al destino. En su caso, restaurar el agente y devolverla true. En otro caso se llama otra vez a moverJugadorRaton para dejar el jugador en estado de mover.

  paraJugadorDosBoton, como la forma de moverse con dos botones es difícil realizar en movimiento, aquí solo comprobamos si ha pulsado cancelar o ha llegado el destino.

- *PREPARARINTER*, antes de interactuar con los objetos tenemos que acercarnos. En el caso de la caja tenemos que estar en el lado correcto y a una distancia correcta para que la animación cuadre. Los objetos interactivos heredan de la clase **Interactuable**, y tienen un float que se llama *distanciaInteractuable* que se usa en un método virtual que es *obtenerPosicion*, que es un método que devuelve el punto más cercano entre el objeto y el jugador, y además es menor que la *distanciaInteractuable*. También tiene dos funciones abstractas: *finPreparar* y *finInteractuar*.

  El estado solo tiene una acción, que es la acción de cancelar, pero el estado tiene otras comprobaciones. Una es comprobar si está atascado en un camino y otra si la diferencia de altura es grande. Para ello creamos un float *contadorAtascado*, comprobamos la velocidad del agente, aquí comprobamos *sqrMagnitude* de la velocidad, porque esta forma es más eficiente. En caso es de que la velocidad sea baja miramos si ha llegado el destino. Si ha llegado al destino comprobamos la diferencia de la altura, si la diferencia es alta volvemos al estado parado, en caso de no haber llegado al destino sumamos *deltaTime* (diferencia entre actual y tiempo del ciclo anterior) al *contadorAtascado*. Si este contador pasa de un segundo entonces el jugador está atascado y volvemos al estado parado.

  solo en caso de llegar al destino y la diferencia de alturas no es grande se llama a *finPreparar* del objeto para que el objeto pida el cambio de estado, así evitamos comprobar qué estado tenemos que cambiar.

  Si el jugador no ha llegado al destino hay que actualizar *destinoPrepararInt*, porque el objeto interactivo se puede mover, como en el caso del coco.

- *PARADOCONCAJA*, antes de explicar el estado vamos a ver la clase **Caja**. Lo primero que tenemos que hacer es reescribir *obtenerPosicion*. Tenemos cuatro variables **Transform** para guardar las posiciones donde puede estar el jugador para interactuar con cada cara de la caja. También creamos cuatro booleanos para que los desarrolladores puedan elegir qué lado tiene un asa para tirar. *obtenerPosicion* de **Interactuable** recibe un **Transform** como parámetro pero aquí vamos a crear una función sobrecarga que recibe un Vector3, porque así, cuando el ratón ha seleccionado la dirección,

podemos pasar la posición de la dirección elegida. *obtenerPosicion* va a calcular qué posición de las cuatro direcciones es la más cercana. También se va a configurar la variable *conAsa* (para saber si se puede tirar), *dir* (para saber la dirección de movimiento) y *detectorPos* (**ObstaculoDetector**, para detectar si la caja ha chocado con algo). Una vez el jugador llama *finPreparar* de la caja, llama *prepararConCaja* de **Player** para cambiar al estado parado con caja. Como hemos dicho en el apartado de interacción que no podemos implementar el movimiento de la caja con agente, se ha implementado con **Transform** en una corrutina. Para la acción de mover caja solo tenemos que intentar asignar destino, y la parte de mover se va a hacer de igual forma para tres tipos de control. Cuando entra en el estado, el jugador también va a activar **ObstaculoDetector**. Para mover la caja necesitamos los valores *dirMoverCaja* y *disPendiente*, La corrutina comprueba cada ciclo si el jugador y la caja han chocado con otros elementos y compruba la distancia entre el jugador y la caja. Si la distancia es grande significa que el jugador o la caja han caído y hay que volver al estado parado.

Al usar *Translate* de **Transform**, tenemos que mover el jugador y la caja, porque son dos objetos separados. Una vez movido, tenemos que restar la distancia desplazada de *disPendiente*, y la corrutina solo funciona si *disPendiente* es un valor positivo.

Las acciones de movimiento con la caja son parecidas a mover, porque las dos formas asignan un destino, solo en este caso si el jugador quiere tirar de la caja hay que comprobar que la variable *conAsa* de la caja está activada, y la distancia es la distancia entre el jugador y el punto proyectado en la dirección de mover. Para saber cuando el jugador quiere tirar hacemos un producto escalar de la dirección hacia delante con la dirección que ha pulsado el jugador, y si el resultado es mayor que 0, entonces hay que empujar y si es menor que 0 hay que tirar.

accionMoverConCajaDosBoton, aquí solo vamos a tener una posición, porque la dirección no tiene sentido. Cuando el lado de la caja tiene asa, se desplaza desde detrás hacia delante.

- *MOVERCONCAJA*, este estado también es parecido al estado mover. Cuando la corrutina detecta que está parado, entonces ha llegado al destino o ha chocado, y hay que volver al estado de parado con caja. Tiene dos acciones: cancelar y mover caja, que es la misma acción que parado con caja.

- *PARADOCONCOCO*, coco es un poco diferente a caja, porque se lleva en la mano, y para que cuadre con la animación, en la mano del modelo del jugador hemos dejado un *posCogerCoco*. La función *finPreparar* de **Coco** va a asignar su padre a *posCogerCoco*, así el coco siempre aparece en la mano del jugador. Este estado tiene dos acciones: cancelar, para dejar el coco y volver al estado parado, y lanzar. Esta acción también hace que el jugador gire.

  Para mejorar un poco la indicación, al elegir la dirección aparece la flecha de elegir dirección.

  accionLanzarCocoWASD, en el apartado de interacción hemos dicho que vamos a girar al personaje poco a poco. Para conseguirlo se usa *RotateTowards*. Esta función recibe dos rotaciones y un float para indicar cuánto hay que girar. Cuando la acción captura el botón devuelve verdadero para cambiar el estado a lanzar.

  accionLanzarCocoRaton, en esta acción vamos a calcular un punto alrededor al que mirará el jugador. Para crear el punto primero transformamos la posición del personaje en la posición de la pantalla, y calculamos la dirección donde está apuntando el ratón. Segundo intercambiamos el campo z y el campo y para poner la dirección en el plano XZ. Por último giramos la dirección calculada con *rotacionCamaraXZ* y sumamos la posición del jugador. Una vez tenemos calculado el punto llamamos a LookAt de **Transform** para mirar a donde apunta el ratón. También se podría haber hecho con *Raycast* y quitar el campo y, pero *Raycast* es costoso y esta forma es muchísimo más eficiente.

- *LANZARCOCO*, es un estado que no tiene acción. Se utiliza para sincronizar con la animación de lanzar. En **Player** creamos un booleano lanzar, este atributo se pone en verdadero dentro de la animación de lanzar coco. El estado

solo comprueba el valor, y cuando está en verdadero llama a *finInteractuar* de **Coco** y añade una fuerza a su **Rigid Body**.

- *PARADOCONAVE*, el estado es simple, porque solo tiene la acción de lanzar y elegir dirección, que son dos acciones que ya tenemos implementadas. Como el ave tiene que interactuar con el coco, hemos creado un **DetectorAve** y lo hemos dejado en el pico del ave. Una vez que el jugador ha elegido la dirección, empieza una animación para estirar el cuello. Hemos creado un booleano para controlar el movimiento de expansión y contracción del cuello. Existen dos condiciones para contraer el cuello: cuando el pico del ave toca algo y cuando ha llegado a su distancia límite. Estas dos condiciones las vamos a comprobar con una corrutina. Para ver si ha llegado a la distancia límite comprobamos si la animación ha finalizado. **Unity** no tiene un atributo para comprobar si ha finalizado una animación. La forma que vamos a usar es mediante *normalizedTime*. Este tiempo se restaura cuando se produce un cambio en la animación. Cuando es mayor que 1 significa que ha cumplido un ciclo. Cuando no ha finalizado la animación miramos si tenemos un objeto en la boca y si hemos chocado con algo. Al chocar con algo, si tiene un objeto en la boca lo lanza y si no entonces llama a *interactuarConAve* de **Coco**. De momento solo podemos interactuar con el coco, si tuviéramos más objetos pequeños, tendríamos que crear una interfaz para representar objetos interactivos pequeños.

- *ESTIRARAVE*, este estado es parecido al estado de lanzar coco. Esperamos cuando el cuello de ave ha retrocedido y entonces volvemos al estado de parado con el ave.

Para finalizar el apartado vamos a presentar la configuración del **Animator** del jugador(Figura 4.22). En la figura se muestran las animaciones del jugador. Tiene una estructura muy parecida a los estados del **Player**, por lo tanto solo tenemos que activar los valores correspondientes en cada estado para hacer funcionar las animaciones.

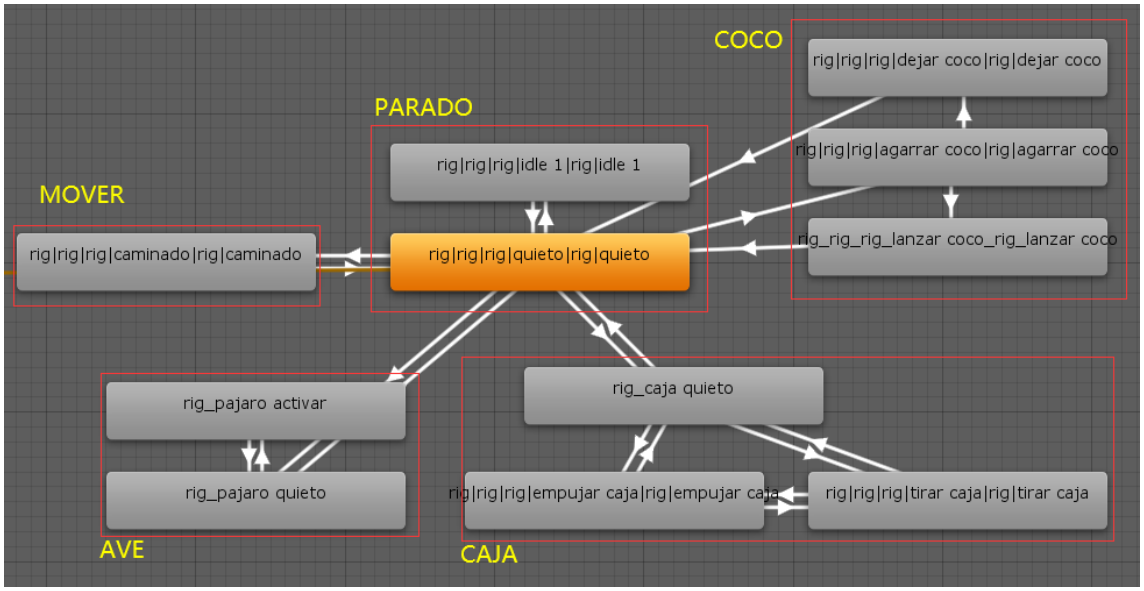


Figura 4.22: Estado de animación del jugador

Como es frecuente modificar los parámetros de las animaciones, en fase de inicialización hemos obtenido el identificador de todos parámetros con *Animator.StringToHash*.

4.5 Nivel

En este apartado vamos a explicar cómo hemos gestionado los niveles y la escena y cómo hemos conseguido la función reintentar(Figura 4.23). Empezamos por la clase interfaz **Nivel**, que tiene dos variables: *id* que sirve para identificar el nivel, y *completado*, que sirve para guardar si el jugador ha pasado el nivel, porque si lo ha completado ya no puede reintentar. La función de esta clase es guardar la información del nivel. Podemos ver que su clase hija **Nivel1**, tiene diferentes objetos. Para reintentar o cargar la escena tenemos que guardar la información de estos objetos en **NivelConfig**. Los datos a guardar se pueden dividir en cuatro tipos: *int, bool, Vector3 y Quaternion*, en el método *generarConfig* vamos a especificar qué hay que guardar de cada tipo. Para facilitar el desarrollo, aquí también hemos reescrito el editor de nivel, y hemos añadido un botón que autogenera un fichero de **nivelConfigObj** que es igual que **NivelConfig** solo que es un **ScriptableObject**. Este objeto va a ser la configuración inicial del nivel.

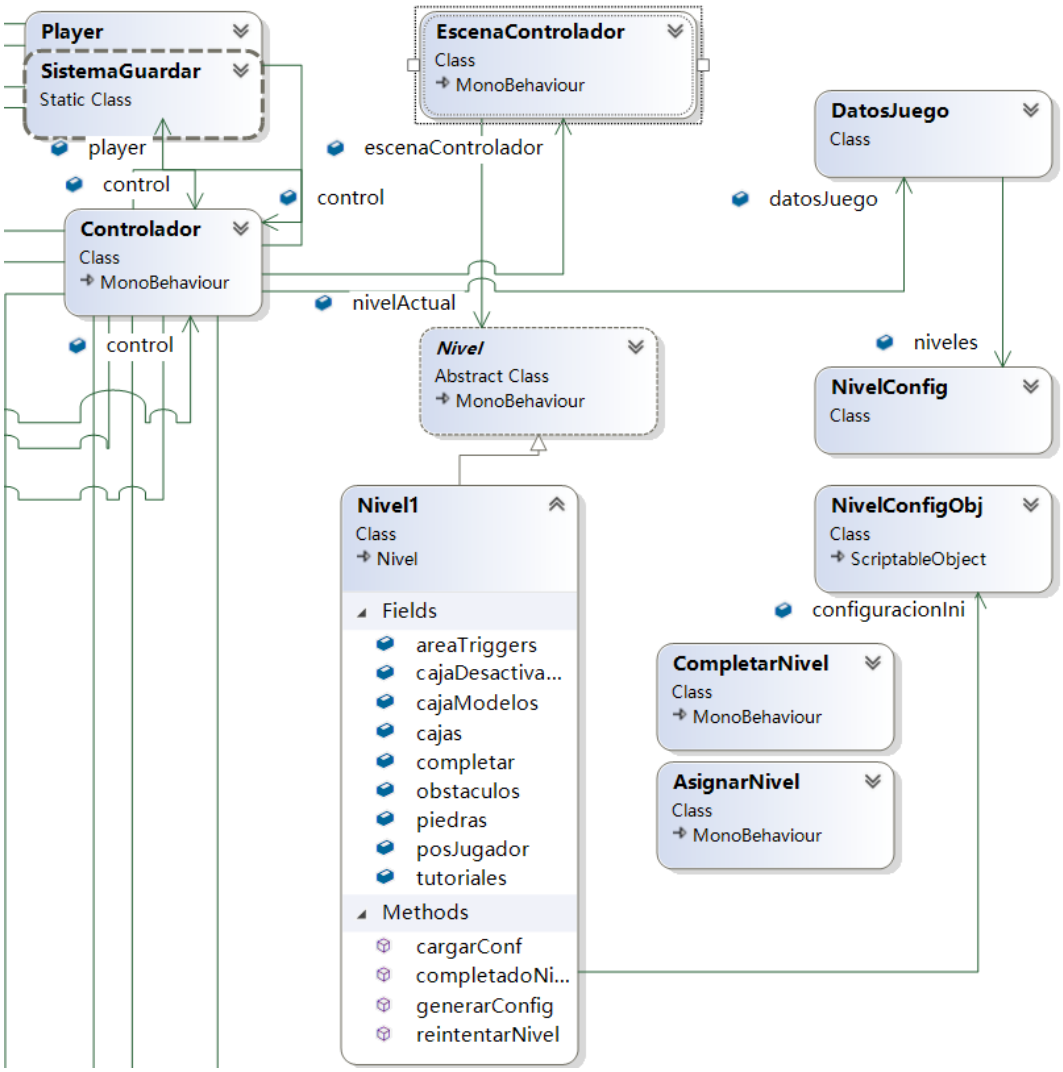


Figura 4.23: Diagrama de nivel

Cada vez que el nivel está cargado va a ver si **DatosJuego** existe y en su caso ejecutar *cargarConf*, que es un método abstracto porque cada objeto se va a cargar de diferente forma. La forma de asignar un valor al nivel es cuando el jugador llega a algún punto del juego. Entonces el método **AsignarNivel** pasará el *id* y la referencia a **EscenaControlador**. También puede pasarle nulo, por ejemplo, cuando el jugador está en una zona intermedia que conecta dos niveles.

Aquí tenemos que aclarar la diferencia entre escena y nivel. La escena tiene su id en la ventana **Build** de **Unity**, y una escena puede contener varios niveles. La forma de cargar la escena es igual que al cambiar de nivel, cuando se llega a un punto del juego hay que cargar o descargar la escena. Por lo tanto tiene el

mismo problema que nivel, en **EscenaControlador** tiene un array de booleanos para recordar qué escenas están activas.

En este proyecto tenemos las siguientes escenas:

- Escena principal, es la primera escena del juego, aquí está el controlador, el menú principal, la cámara y el **Event System**.
- Escena intermedia. Es la primera escena que se carga cuando **Controlador** se ejecuta *NuevaPartida* o *Continuar*. Esta escena tiene el jugador, **CanvaJuego** y **Nav Mesh Surface**, con lo cual tiene datos de todas las escenas que existen en el juego. Una vez cargada esta escena, **EscenaControlado** empieza a cargar datos y cargar las escenas que se indica en **DatosJuego**.
- Isla, es la escena donde tiene el nivel 1, es la única escena del juego que tenemos hecha.

4.6 Funciones del nivel 1

En este apartado vamos a explicar las funciones del nivel 1.

4.6.1. Caja

Cuando recibimos la escena del diseñador nos encontramos un problema, y es que los huecos tienen forma de cono son pequeños, por lo que no cabe la caja. Por ello hemos decidido intentar que se caiga utilizando código. La clase que realiza esta función es **AreaTrigger**, nuestro objetivo es que cuando la caja tape todo el hueco, generar un evento para que la caja caiga ignorando la física. Para saber si la caja ha tapado todo el hueco vamos a necesitar varios activadores, por eso **AreaTrigger** va a tener un contador para guardar cuántos activadores tenemos y un *array* de **AreaTriggerComponente**. Cuando un objeto toca el componente, entonces el componente llamará a la función *Activar* de **AreaTrigger** y cuando el objeto se aleja, llamará a *Desactivar*.

De momento solo tenemos un hijo de **AreaTrigger** que es **DetectorCajaCaer**. Cada vez que llama a su *Activar* va a decrementar el contador, y cuando el conta-

dor llega a 0, comprueba que todos los componentes han tocado el mismo objeto, y entonces llama a caer de Caja.

El método *Desactivar* de esta clase simplemente incrementa el contador. Una vez que ha caído la caja, tenemos que desactivar **DetectorCajaCaer** para no gastar recursos computacionales, su estado de activo también tenemos que guardarlo.

Una vez la caja ha caído, el jugador debe poder pasar por el hueco, lo que impide que el jugador realmente pase es un **Nav mesh Obstacle**. Como puede interactuar con el hueco cualquier caja, este obstáculo tenemos que enlazarlo con el detector. El método *caer* de **Caja**, recibe una función que es finalizar y la ejecuta cuando la animación de caer ha finalizado. Como las cajas que pueden rellenar un hueco tienen una animación, el estado de la animación también tenemos que guardarlo.

4.6.2. Tutorial

En el libro[16] podemos encontrar que el jugador puede frustrarse si no sabe cómo controlar el juego, y por eso hemos decidido crear el tutorial. El tutorial puede iniciarse por varias condiciones. Por ejemplo, cuando empieza el juego se muestra el tutorial para señalar cómo mover al personaje. Cuando aparece la caja se muestra un tutorial para decir cómo interactuar con ella, y al interactuar con una caja se muestra un tutorial sobre cómo empujar o tirar. **Tutorial** tiene un método *mostrarTutorial* que recibe un int, que determina qué textos e imágenes tiene que mostrar.

En **Tutorial_Nivel1** hay dos estructuras de datos, un *array* de **Sprite**, que sirve para guardar imágenes y otro es un diccionario. La idea es que el método *cargarTexto* cargue el diccionario desde un fichero local. El juego permite diferentes idiomas, pero no tenemos hecho el autómata por lo tanto aquí lo inicializamos dentro del método y no va a permitir cambiar el idioma, pero en un futuro se podrá.

Para no confundir al jugador, solo se muestra el tutorial del tipo de control seleccionado. *mostrarTutorial* de **Controlador**, recibe dos parámetros: un *array* de **Sprite**s y un *array* de **string**. Cada vez muestra un elemento de los arrays. El

contenido de **Sprite** puede ser nulo, y en este caso no se muestra ninguna imagen. Cuando no queda nada más se cierra el panel. En la figura 4.24 podemos ver una tutorial de enseñar como interactuar con la caja.

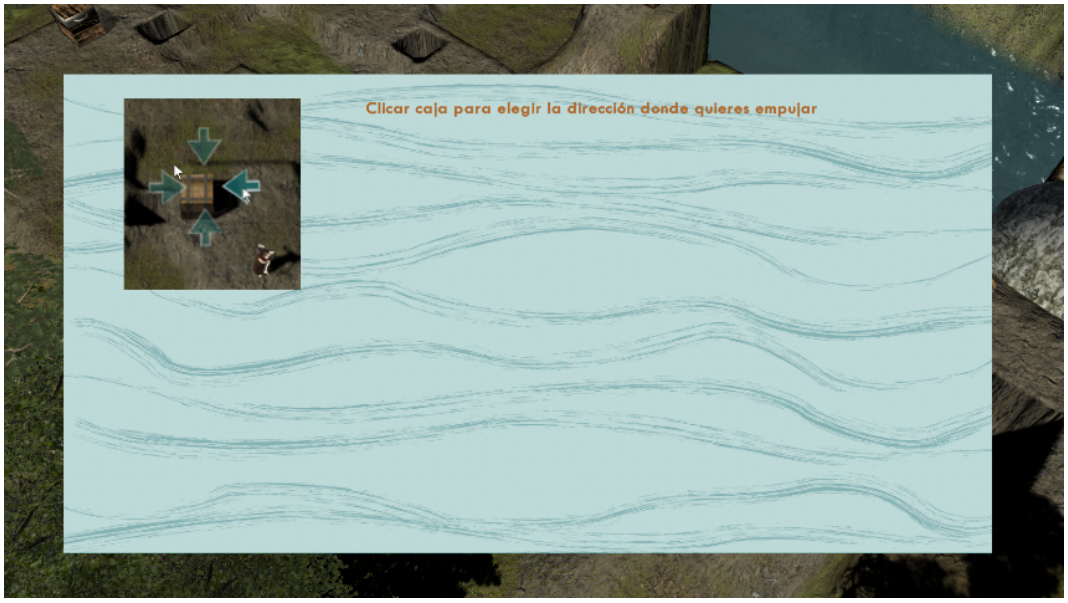


Figura 4.24: Tutorial de cómo interactuar con la caja

4.6.3. Obstáculo

En el nivel 1 vamos a encontrar una piedra grande que nos impide avanzar. El jugador tiene que empujar la caja encima de un botón para que la piedra se caiga hacia un lado y abrir el camino. Para conseguirlo, hemos creado una clase **Puerta** que representa los obstáculos. Tiene un método *Abrir*, que recibe un booleano para indicar si el botón está pulsado. Como el jugador puede salir del juego cuando la piedra está produciendo la animación de caer, hasta que no acabe la animación no va a guardar la rotación y posición de la piedra.

4.6.4. Cámara

En funcionalidad básica hemos dicho que se usaría **Cinemachine** para implementar la cámara. Para ello vamos a necesitar crear una cámara virtual. Vamos a crear una variable para guardar la referencia de la cámara virtual, y cuando el jugador está dentro de una zona, **ActivaCamara** pasará la cámara virtual a Controlador, y si el Controlador ya tiene una entonces se desactiva.

La cámara virtual tiene dos atributos importantes: *Follow*, para seguir al jugador y *LookAt*, para mantener al jugador en el centro. A estos dos atributos hay que pasarles el objeto para que funcionen. El desarrollador puede elegir activar o desactivar esta dos atributos.

4.6.5. Sonido de pisadas

La forma de implementar esta función es similar a la forma de activar la cámara. En player creamos un variable *sonidoAndar* para guardar el **AudioClip** las pisadas. Para cambiar el sonido de la pisada dependiendo de la posición en el nivel, en cada zona de la escena ponemos un activador. Cuando el jugador entra en esta zona entonces **SonidoPaso** llama a *cambiarSonidoPaso* de **Player** para cambiar el sonido.

4.7 Nivel 1

Figura 4.25: Isla de nivel 1

En este apartado vamos a presentar cómo es el nivel 1 de nuestro proyecto(Figura 4.25). Empezamos en la arena(Figura 4.26), aquí es donde va a vivir el pueblo, de momento como no tenemos animaciones y modelos hecho no hemos añadido nada, pero depende del tiempo que ha pasado(número de veces que muere el jugador), va a tener más infraestructuras y más personas.

Figura 4.26: Arena

Seguimos entrando al bosque(Figura 4.27), hemos puesto que el **Dolly Track** siempre sigue un poco detrás del jugador, eso ayuda al jugador ir hacia delante y es fácil de encontrar el camino que hay que seguir.

Figura 4.27: Bosque

Después de atravesar el bosque encontramos la primera parte de nuestro nivel 1(Figura 4.28), como primer nivel hay que ser fácil y que solo sirve para familiarizar el control, solo hay una caja para cada hueco.

Figura 4.28: Primera parte del nivel 1

Una vez resuelto la primera parte vamos a encontrar la segunda funcionalidad de la caja, aquí tenemos que poner la caja encima de la roca roja(que es un botón) para activar el camino. Aquí también encontramos nuestro primer peligro, que es el hueco grande que podemos ver en la figura 4.29. Cuando el jugador está tirando la caja es posible caer al hueco grande y si se cae se muere.

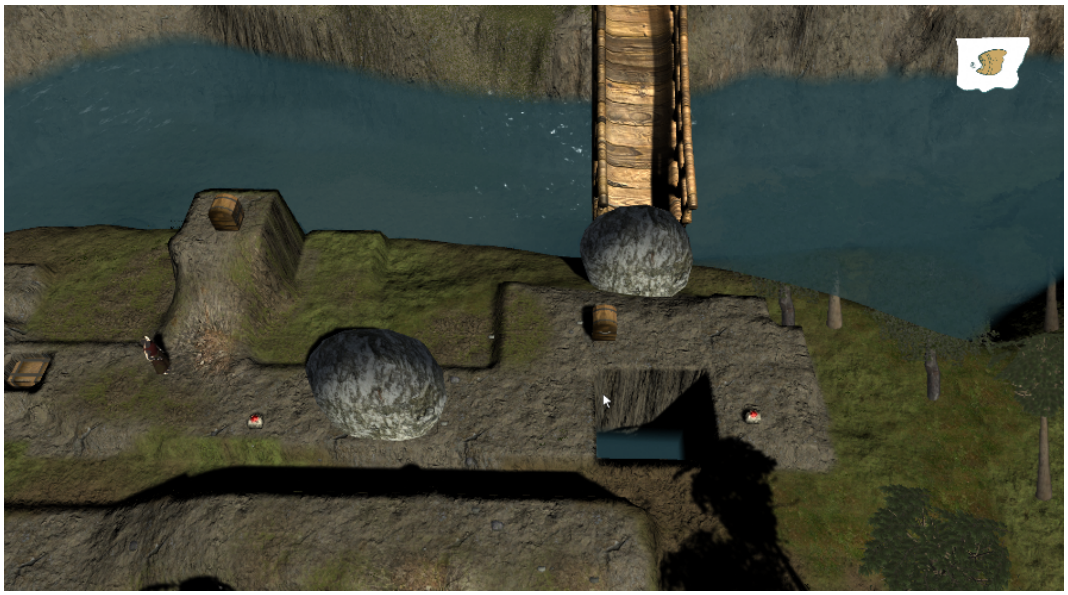


Figura 4.29: Segunda parte del nivel 1

El primer nivel acaba cuando el jugador resuelve todo puzle y llega al puente(Figura 4.30), aquí acaba el juego de momento.

Figura 4.30: Completado

CAPÍTULO 5

Pruebas y Resultados

La metodología que hemos usado es la iterativa. Esto significa que ya hemos hecho muchas pruebas y las soluciones ya están implementadas. En este capítulo solo vamos a presentar la última prueba. Hemos realizado dos pruebas: el juego en el editor y el juego compilado. Primero describiremos los problemas comunes. El primer problema es que nos dimos cuenta de que falta cambiar el color de la hierba en la función de cambiar el color de los objetos. Para resolver este problema tenemos que añadir **TerrainData** a **ContrasteControlador**.

El segundo problema es un problema que pasa a veces. No sabemos la condición que produce el problema, pero hemos identificado hay veces que en la colisión del personaje y **Terrain**, por una condición desconocida el jugador sale volando.

El tercer problema es que los árboles van creciendo cuando nos acercamos a ellos. Es por el LOD de los modelos que no está bien configurado. Es un problema grave tanto visualmente como con respecto a la eficiencia, porque los detalles del modelo cambian constantemente y el colector de basura está constantemente recolectando. Este problema lo debe resolver el diseñador, cambiando la configuración de los LOD de los modelos.

A parte de estos tres problemas hay otro problema que solo pasa en la versión compilada, que es que algunos árboles son blancos. Esto es porque **Unity** no ha incluido los shaders que hemos usado, y tenemos que incluirlos manualmente. Como el juego aún solo tiene un nivel, y no vamos a sacar la versión final del juego, de momento no vamos a incluir estos shaders.

La resultado del proyecto es casi todas las funciones del juego tienen un coste constante, a veces lineal. El resultado obtenido se puede medir analizado el juego con la herramienta **Profile** de **Unity**.

Aparte de las funciones de inicialización y de carga de la escena, nuestro juego solo tiene dos partes que son costosas cuando el jugador está jugando. Una es cargar o descargar la escena, porque va a guardar información de varios niveles. En este caso usamos métodos asíncronos y así el jugador no percibe una pausa en el juego. Sin embargo, reintentar sí que genera una pausa en el juego. Otra función costosa es la de cambiar colores, porque tiene que recorrer todos los materiales.

El juego se a probado con un portátil de MSI con siguientes características:

CPU: i7-7700HQ

GPU: GTX 1060 3GB

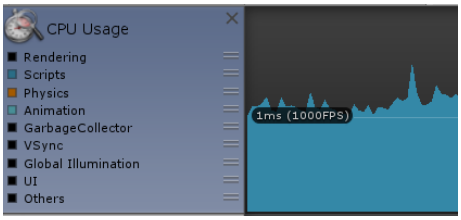
RAM: 16GB

SO: Windows 10 64bit

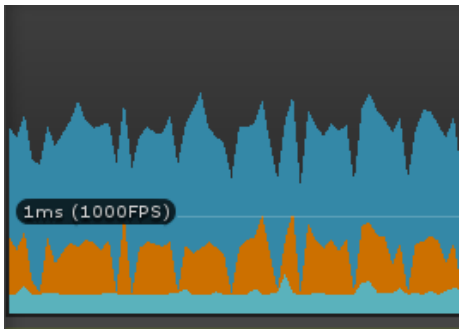
Con la herramienta **Profile** se han obtenido las siguientes medidas:

- Navegar por las opciones del menú tiene un tiempo de ejecución de alrededor de 1 ms que podemos ver en la figura 5.1, es decir, más o menos 1000 FPS,
- En la figura 5.2 podemos ver los tiempos cuando el jugador juega el juego. Cuando el jugador se mueve, interactúa o se muestra el tutorial, el tiempo ejecución se mantiene más o menos en 2 ms (más o menos 500 FPS),
- al seleccionar reintentar o cambiar los colores ha costado 4 ms.

Los valores mencionados depende de la complejidad del nivel y del número de materiales, pero el tiempo de espera del jugador sigue siendo bajo.

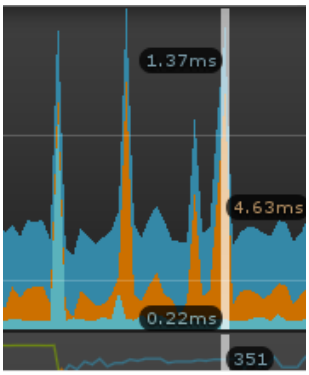


Tiempo de modificar opciones

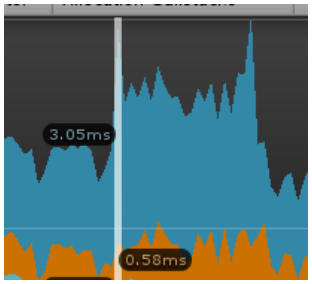


Tiempo de interactuar

Figura 5.1: Tiempos constantes



Tiempo de reintentar



Tiempo de cambiar color

Figura 5.2: Tiempos de las funciones

CAPÍTULO 6

Conclusiones

Después de tres meses de desarrollo del juego, los objetivos del juego se han cumplido casi en su totalidad, aunque no se haya acabado el juego. Todas las propuestas de los diseñadores se han implementado casi en su totalidad. Las dos funciones que no se han podido implementar son diseñar un modo para jugadores ciegos e implementar la acción de deshacer. La primera propuesta se hizo a mitad del desarrollo y no sabemos cómo podemos implementarlo. Con respecto a la función de deshacer, el objetivo es permitir al jugador deshacer las acciones mal hechas para poder avanzar en el juego. Como no podemos deshacer infinitos pasos se ha decidido sustituir la función de deshacer por la función de reintentar, que sí se ha implementado.

Mis ideas sí se han implementado en su totalidad, pero como el juego no está acabado y no se ha publicado, las formas de control del jugador que hemos diseñado no sabemos si realmente son usables o no.

Durante los tres meses de desarrollo, hemos usado dos meses para implementar las interfaces del juego y las interacciones del jugador, y un mes para el primer nivel. El sistema básico del juego está casi acabado. Hay muchas ideas que hemos sacado de este libro[16]. Es un libro que habla sobre distintos tipos de discapacidad, pero nuestro juego está centrado en personas con movilidad reducida y problemas visuales. Hay muchas ideas sobre accesibilidad pero pocas implementaciones. Esperamos que este TFG pueda ayudar a otros desarrolladores que quieran añadir funciones accesibles a su juego, porque tenemos algunas

funciones que son fáciles de aplicar a otros juegos como navegación, la función de cambiar color, estado de la máquina, etc.

Los conocimientos que hemos adquirido durante la carrera han influido mucho en el proyecto. Aparte de las técnicas descritas, hay otros aspectos que no hemos mencionado en la memoria. Hay un término que nos ha acompañado durante la carrera que es el coste. Esto nos ha llevado a buscar documentación sobre optimización del juego [17], y siempre estamos pensando cómo optimizar, tal como elegir la estructura adecuada para guardar datos (**Estructuras De Datos Y Algoritmos**), diseñar un **Controlador** que sirva para conectar con otras clases para facilitar el diseño y mantenimiento (**Ingeniería Del Software**), evitar división de coma flotante mediante una multiplicación o en algunos casos, usar if y return (**Arquitectura E Ingeniería De Computadores**), pensar bien cuál es el problema y cuál es su coste mínimo (**Computabilidad Y Complejidad**), pensar qué código va a ser optimizado por el compilador y cuál es posible que no se optimice, etc.

Durante el desarrollo hemos conocido mejor el ciclo de vida del desarrollo de videojuegos y hemos aprendido funcionalidades básicas de **Unity**, como renderizado y los controles de eventos. Igual saber más paradigmas de programación hubiera ayudado a analizar el problema desde diferentes puntos de vista. Esto también pasa con el motor de videojuegos, en los siguientes proyectos queremos probar con otros motores para tener más conocimiento sobre juegos.

CAPÍTULO 7

Trabajos futuros

El principal futuro trabajo va a ser implementar más niveles. Como tenemos ya implementada la interacción con el coco y el ave, podemos jugar con estos dos objetos.

También tenemos que añadir más personajes al pueblo, y cambiar las infraestructuras del pueblo cada vez que el jugador muera para dar una sensación de paso del tiempo.

Aparte de diseñar más niveles, también tenemos que añadir más funciones sobre la accesibilidad, tal y como avisar al jugador cuando se acerca a un peligro y alertas visuales.

Respecto a la escalabilidad del juego, de momento el juego no tiene problema de escala, pero tiene dos aspectos que limitan la escala del juego: el tamaño de una parte de la isla y el número de materiales.

La primera limitación es porque **Nav Mesh Surface** necesita la información de toda la isla para que el agente pueda moverse de una escena a otra como si fueran una. Para resolver esto podemos dividir la isla en varias partes, pero para cambiar entre la escena se necesita cargar. Otra solución es crear un camino para que el jugador pueda saltar a la otra parte, usando **Nav Mesh Link** (el agente solo puede pasar este enlace saltando) para conectar dos **Nav Mesh Surface**.

La segunda limitación es que ahora cada vez que cambiamos el color cargamos todos los materiales y los recorremos de uno en uno. Para resolverlo hay dos formas: la primera es usar otra implementación, que no he mencionado en el

análisis porque no sé si puede implementar en la versión gratuita de **Unity**. Consiste en modificar la parte de renderizado del motor para que antes de dibujar se calcule qué tipo de objeto es y se aplique el color correspondiente. Esta solución tendría un coste de computación y de desarrollo parecido o menor a la función de corrección de colores.

Otra forma es crear un **ContrasteContrador** para cada escena, y así cambiar el color solo al cargar la escena.

Bibliografía

[1] J. Petty, "Top 12 free game engines for beginners & experts alike," *Concept Art Empire*, 2020. [Online]. Available: https://conceptartempire.com/free-game-engines/

[2] *GDScript basics*. [Online]. Available: https://docs.godotengine.org/en/stable/getting_started/scripting/gdscript/gdscript_basics.html

[3] *NODES: GODOT'S BUILDING BLOCKS*. [Online]. Available: https://kidscancode.org/godot_recipes/g101/start/101_03/

[4] D. Buckley, "Godot 4.0 tutorials – complete guide," *GameDev Academy*, May 2020. [Online]. Available: https://gamedevacademy.org/godot-4-0-tutorial/

[5] M. Dealessandri, "What is the best game engine: is cryengine right for you?" *gamesindusty*, Jan 2020. [Online]. Available: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-cryengine-the-right-game-engine-for-you

[6] ——, "What is the best game engine: is unreal engine right for you?" *gamesindusty*, Jan 2020. [Online]. Available: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-cryengine-the-right-game-engine-for-you

[7] *Introduction to Blueprints*, Unreal. [Online]. Available: https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted/index.html

[8] "A first look at unreal engine 5," 2020. [Online]. Available: https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5

[9] M. Dealessandri, "What is the best game engine: is unity right for you?" *gamesindusty*, Jan 2020. [Online]. Available: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you

[10] *SurrogateSelector Class*, MicroSoft. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.surrogateselector?view=netcore-3.1

[11] *EventSystem*, Unity. [Online]. Available: https://docs.unity3d.com/2018.1/Documentation/ScriptReference/EventSystems.EventSystem.html

[12] *Terrain Layers*, Unity. [Online]. Available: https://docs.unity3d.com/Manual/class-TerrainLayer.html

[13] *Application.wantsToQuit*, Unity. [Online]. Available: https://docs.unity3d.com/ScriptReference/Application-wantsToQuit.html

[14] *Order of Execution for Event Functions*, Unity. [Online]. Available: https://docs.unity3d.com/Manual/ExecutionOrder.html

[15] *Rotation and Orientation in Unity*, Unity. [Online]. Available: https://docs.unity3d.com/Manual/QuaternionAndEulerRotationsInUnity.html

[16] J. L. González, N. Padilla, M. Cabrera, F. L. Gutiérrez, and P. Paderewski, "Favoreciendo la jugabilidad en videojuegos accesibles." in *Buenas prácticas de accesibilidad en videojuegos*, 2012, p. 73. [Online]. Available: http://www.ceapat.es/InterPresent1/groups/imserso/documents/binario/accesvideojuegos.pdf

[17] *Performance recommendations for Unity*, MicroSoft. [Online]. Available: https://docs.microsoft.com/en-us/windows/mixed-reality/performance-recommendations-for-unity