Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

MÁSTER UNIVERSITARIO EN DESARROLLO DE VIDEOJUEGOS

Trabajo Fin de Máster

Desarrollo de videojuegos serios para la rehabilitación de personas con daño cerebral

Development of serious videogames for the rehabilitation of people with brain damage

Autor: Juan Salvador Magariños Alba
Tutor: Jesús Miguel Torres Jorge
Cotutora: Silvia Alayón Miranda

D./Dña. **Jesús Miguel Torres Jorge**, profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D./Dña. **Silvia Alayón Miranda**, profesora Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutora

C E R T I F I C A (N)

Que la presente memoria titulada:

“Desarrollo de videojuegos serios para la rehabilitación de personas con daño cerebral”

ha sido realizada bajo su dirección por D. Juan Salvador Magariños Alba.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a julio de 2024.

Agradecimientos

Quiero agradecer a mi familia y amigos el apoyo constante a lo largo de todo este tiempo, en especial a mis padres, por darme los ánimos que necesitaba para finalizar mis estudios.

También quiero dar las gracias a mis tutores, Jesús Torres y Silvia Alayón, por toda la ayuda que me han prestado a lo largo del desarrollo del TFM.

Del mismo modo, agradezco también al equipo del Hospital Universitario, los investigadores Cristián Modroño, Rebeca Villaroel y Bárbara García, toda la ayuda y feedback que me han dado para sacar adelante el juego.

Índice

Resumen

El objetivo principal de este proyecto es desarrollar un videojuego para el Hospital Universitario de Canarias (HUC) que se pueda controlar con un eye tracker. Este programa se va a utilizar en la rehabilitación de pacientes con movilidad reducida o nula en las extremidades debido a una lesión cerebral. El equipo del HUC dispone de un proyecto de Unity al que otras personas han incorporado juegos para rehabilitación. Así, el objetivo final de este Trabajo Fin de Máster es incorporar un juego nuevo a dicho proyecto.

Concretamente, en el videojuego desarrollado, el jugador controlará con la mirada el movimiento de un pez en el fondo del mar atravesando diferentes escenarios. Estos escenarios son niveles del juego con distintos objetivos, como llegar a la meta antes de que se acabe el tiempo, o recoger un cierto número de premios. Los niveles se generan de manera aleatoria con un algoritmo sencillo que solo necesita la definición previa de algunos parámetros, como por ejemplo, los obstáculos que aparecerán o el número de premios a recoger.

Por último, para validar la aplicabilidad real del videojuego desarrollado en el ámbito de la rehabilitación, se realizarán estudios de usabilidad con personas sanas y con pacientes. Las personas que prueben el juego realizarán un cuestionario de satisfacción y usabilidad, cuya información se podrá utilizar después en futuras actualizaciones del juego.

Palabras clave: videojuego, rehabilitación, eye tracker, visión, movilidad

Abstract

The main objective of this project is to make a videogame for the University Hospital of the Canary Islands (HUC) that can be played with an eye tracker device. This program will be used to help rehabilitate patients with limited or null mobility in their limbs due to brain damage. The HUC team has a Unity project where people that have developed games for rehabilitation have uploaded their projects. For this reason, the final objective of this project is adding a new game to said project.

Specifically, in the game developed for this project, players will play as a fish at the bottom of the sea that must traverse multiple stages. These stages are levels with different objectives, like reaching the level's goal before a timer expires, or picking a certain amount of prizes up. The levels will be generated randomly using a simple algorithm that only needs some parameters, like the types of obstacles that will appear in the course and the number of prizes that the player must get.

FInally, the game will be tested with both disabled and healthy people in order to check whether the game can be effectively used as a rehabilitation tool. To do this, those who play the game will receive a usability test, whose information will be useful for future updates of the game.

Keywords: videogame, rehabilitation, eye tracker, sight, mobility

Capítulo 1
Introducción

1.1. Objetivo del Trabajo Fin de Máster

El objetivo principal de este Trabajo de Fin de Máster (TFM) es desarrollar un juego que se pueda usar en la rehabilitación de personas que sufren de alguna incapacidad motora por lesiones cerebrales. Más concretamente, el juego debe cumplir los siguientes requisitos:

- Debe ser completo, es decir, debe ser funcional y ofrecer varios niveles de complejidad.
- Debe ser integrado en el videojuego de Unity que contiene el resto de juegos actualmente usados en rehabilitación por el equipo del HUC.
- Debe ser controlado por un eye-tracker, es decir, un dispositivo capaz de detectar hacia dónde está mirando el jugador.
- Debe ser entretenido y motivador.

Además, el juego debe ser probado tanto con voluntarios sanos como por pacientes con algún tipo de daño cerebral.

La rehabilitación que se pretende realizar con el juego se basa en hacer que el paciente controle elementos en un entorno virtual -el juego- moviendo los ojos. Como se explicará más adelante, esto se debe a que las zonas del cerebro que controlan el movimiento de los ojos y las extremidades están ligeramente solapadas. Por ello, hacer que el paciente realice movimientos con los ojos puede ayudar a que mejore su capacidad para mover los brazos. Así, el juego está orientado a pacientes con lesiones cerebrales que les impiden usar los brazos, por lo que no pueden utilizar mandos convencionales.

1.2. Problema médico abordado

Este TFM se encaja en el proyecto NRET (Neurorrehabilitación basada en eye tracking) del HUC. En él se investiga cómo restablecer las funciones motoras en las extremidades superiores mediante el estímulo de las zonas del cerebro que se encargan de sus movimientos. La aproximación utilizada en el proyecto se basa en que las regiones del cerebro implicadas en el movimiento de los ojos y los brazos se solapan ligeramente. Por ello, aunque el paciente no pueda utilizar sus extremidades superiores con soltura, se pueden estimular las zonas del cerebro que se encargan de esas acciones moviendo los ojos, como se explica en [1].

La rehabilitación propuesta por el proyecto se basa en hacer que el paciente controle mediante los ojos un objeto presente en un entorno virtual. Al hacer esto, se activará parte de las zonas del cerebro que se encargarían del movimiento si este se realizase con las manos. La Figura 1 muestra las zonas del cerebro que se activan cuando el paciente controla el objeto virtual con las manos (A) y con los ojos (B). La última fila, C, muestra las regiones del cerebro que se estimulan en ambos casos.

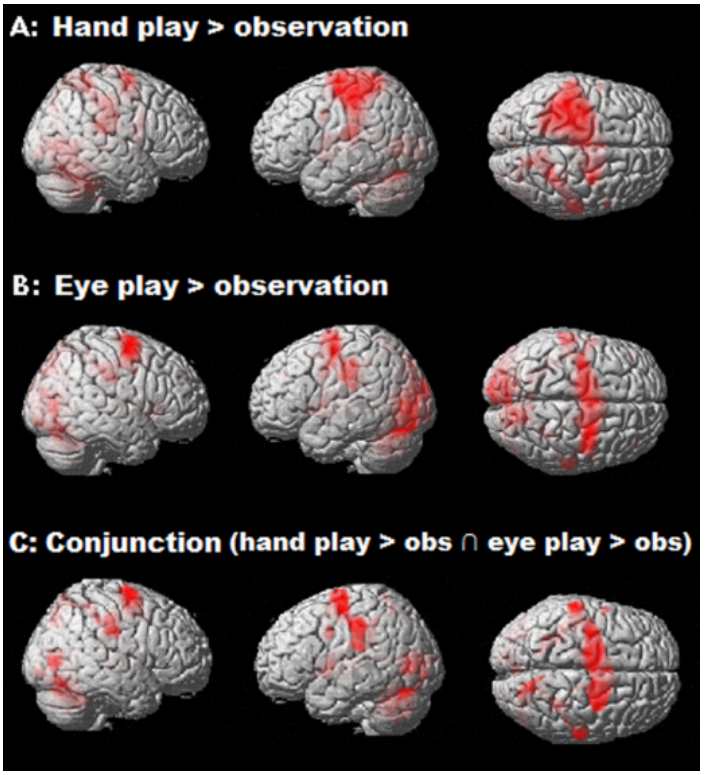

Figura 1. Regiones del cerebro que se activan al controlar el videojuego con las manos y los ojos. Imagen extraída de [2]

1.3. Antecedentes

Las nuevas tecnologías son una de las herramientas más utilizadas para la rehabilitación y monitorización de personas que sufren diferentes tipos de daños físicos y mentales. Así, a lo largo de los últimos años han surgido diferentes técnicas y metodologías aplicadas a la rehabilitación, algunas de las cuales se exponen en [3]. Un ejemplo es la telerehabilitación, que permite a los pacientes recibir sesiones de rehabilitación de forma telemática. Otras dos tecnologías

ampliamente utilizadas son la realidad virtual y los *wearables* o tecnología vestible.

Por un lado, la realidad virtual, que no tiene que estar vinculada necesariamente con los videojuegos, permite ayudar a personas con diversas dolencias, como por ejemplo, Parkinson o daños en la columna vertebral, a mejorar sus funciones motoras y su equilibrio. También facilitan a las personas con daño cerebral el restablecimiento de sus funciones cognitivas, como la memoria. No obstante, el uso de realidad virtual para rehabilitación no está exento de retos a los que enfrentarse. Algunos de ellos son el alto coste de la tecnología, y el riesgo de que los pacientes se mareen durante la sesión de juego.

Figura 2. Hombre usando gafas de realidad virtual. Imagen extraída de [4]

Por otro lado, los dispositivos tecnología vestible son aquellos que el usuario puede llevar puestos como si fuera un complemento, lo que facilita la monitorización de los pacientes. Dentro de este conjunto se encuentran dispositivos como los relojes de pulsera inteligentes, que pueden tomar medidas como la presión arterial de quien los lleva. También están las bandas de la marca Brainbit [5], que se llevan puestas en la cabeza y permiten medir la actividad eléctrica del cerebro de quien las lleva en todo momento.

Figura 3. Reloj de pulsera inteligente. Imagen extraída de [6]

Los videojuegos y las diferentes tecnologías relacionadas con ellos se han abierto paso como herramientas rehabilitadoras gracias a la gran variedad de géneros y dispositivos que existen. Por ejemplo, en [7] se exponen las ventajas del uso de videojuegos en los tratamientos, destacando cómo se puede hacer que el paciente se evada de sus dolencias y se interese por la rehabilitación. Esto es especialmente importante para la rehabilitación que atañe a este TFM, que utiliza el movimiento de los ojos como móvil rehabilitador, como se explicará al final de este apartado.

Existen varios ejemplos de juegos creados como herramienta de rehabilitación. Por ejemplo, Circus Challenge [8], que aparece en la Figura 4, es un videojuego para rehabilitación desarrollado por la universidad de Newcastle y Limbs Alive. En él, los jugadores deben realizar actividades en un circo utilizando controles por movimiento parecidos a los mandos de la Wii. Por otro lado, en [9] se expone cómo algunos juegos de realidad virtual que no han sido creados para rehabilitación pueden ser útiles para esta tarea. De entre los juegos mencionados destaca Beat Saber, especialmente por sus opciones de accesibilidad.

Figura 4. Circus Challenge. Imagen extraída de [8]

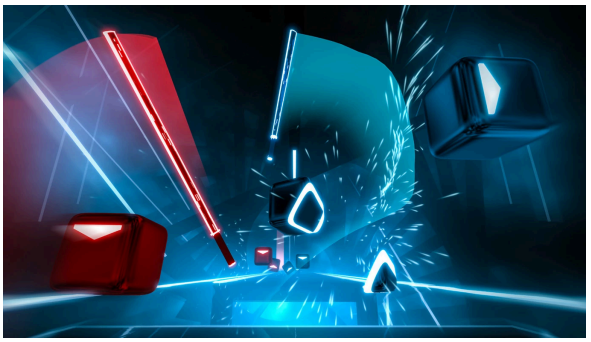

Figura 5. Beat Saber. Imagen extraída de [10]

El desarrollo de videojuegos terapéuticos es un tema en el que se basan varios trabajos hechos en la Universidad de La Laguna. Por ejemplo, en los Trabajos de Fin de Grado [11] y [12] se hicieron los videojuegos “Space Invaders” y “Breakout” para el HUC controlados por eye-tracker. Estos aparecen en las Figuras 6 y 7.

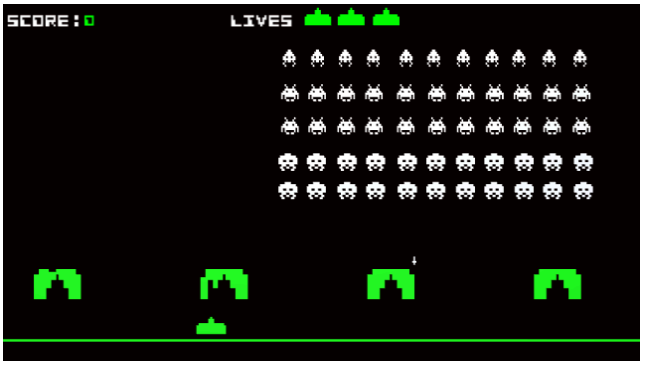

Figura 6. Space Invaders. Imagen extraída de [11]

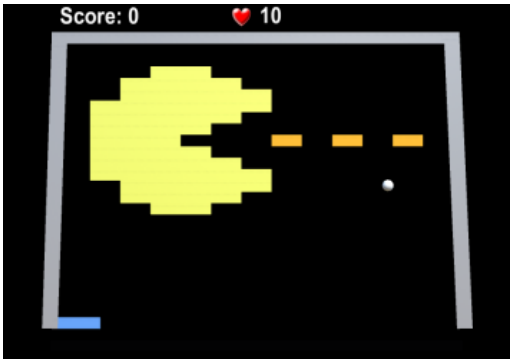

Figura 7. Breakout. Imagen extraída de [12]

Así, el proyecto en el que se encaja este TFM dispone de un videojuego hecho en Unity, también llamado NRET, que permite practicar juegos como el tiro al blanco y el "Golpea al topo" utilizando solo los ojos. Al ejecutar el juego, aparece la pantalla con los perfiles de los jugadores, que se muestra en la Figura 8, en la que el jugador puede seleccionar su perfil con la mirada.

Figura 8. Pantalla de selección de perfil del juego NRET

Al elegir un perfil, aparece el menú principal, que se muestra en la Figura 9. Aquí, el jugador puede elegir a qué jugar.

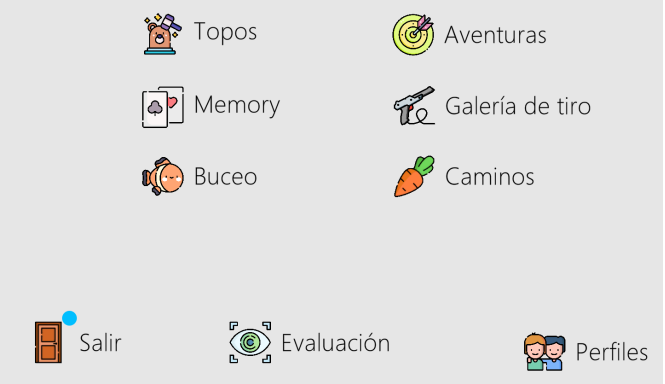

Figura 9. Pantalla de selección de juego

Finalmente, cuando el jugador selecciona un juego, aparece una última pantalla que muestra información relativa al progreso del jugador en el juego elegido. En el caso del juego desarrollado en este TFM, "Buceo", y de otros, esta información consiste en el nivel actual del jugador, y su puntuación. Esta pantalla se muestra en la Figura 10.

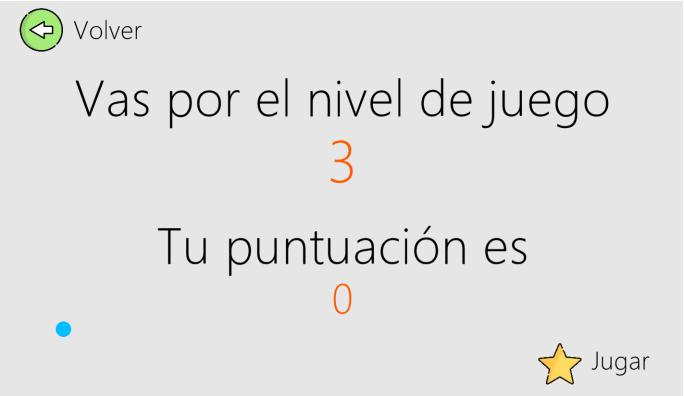

Figura 10. Pantalla previa al comienzo del juego

Capítulo 2
Herramientas utilizadas

En este TFM se han utilizado varias herramientas hardware y software. La herramienta hardware principal que se ha utilizado, aparte del ordenador en el que se ha desarrollado el juego, es un eye tracker de Tobii [13].

Tobii es una empresa especializada en desarrollo de tecnología de seguimiento de ojos (eye tracking) que ha desarrollado varios productos. Algunos ejemplos son las Tobii Pro Glasses 3 [14], unas gafas que incluyen eye trackers para poder realizar el seguimiento de la mirada de los usuarios de una forma más natural, y Tobii Horizon [15], un software que, si se conecta con un juego compatible, permite detectar la orientación y movimientos de la cabeza del jugador, y utilizar estos datos para controlar la cámara del juego. La lista de juegos que soportan el uso de Tobii Horizon incluye tanto juegos en primera persona, como es el caso de Microsoft Flight Simulator, como juegos en tercera persona, como Assassin's Creed Valhalla.

Concretamente, el eye tracker utilizado fue proveído por el HUC, y es el modelo 4C. Su modo de empleo es relativamente sencillo, ya que se conecta al PC mediante un cable USB y, para comenzar a utilizarlo, solo hay que calibrarlo utilizando el programa Tobii Eye Tracking Core Software [16].

Figura 11. Eye tracker 4C utilizado en el proyecto. Imagen extraída de [17]

Con respecto a las herramientas software, se han utilizado varios programas para la realización de las distintas partes del juego. La más importante es el motor de videojuegos Unity [18], que es, junto a Unreal Engine, uno de los programas para desarrollar videojuegos más populares. Entre sus características más destacables se encuentran su curva de aprendizaje sencilla y su soporte para exportar un mismo juego a varias plataformas usando un solo proyecto. No obstante, el motivo por el que se usó dicho motor es que el proyecto del HUC en el que se almacenará el juego desarrollado está hecho en Unity, aprovechando el SDK de Tobii disponible para este motor. También se utilizó la librería iTween para Unity [19] para implementar la vibración de algunos efectos del juego. No obstante, este recurso permite realizar algunos tipos de animaciones más fácilmente. Entre ellas,

destacan algunas cuya implementación puede ser más compleja, como el movimiento en una superficie no lisa, y las trayectorias parabólicas.

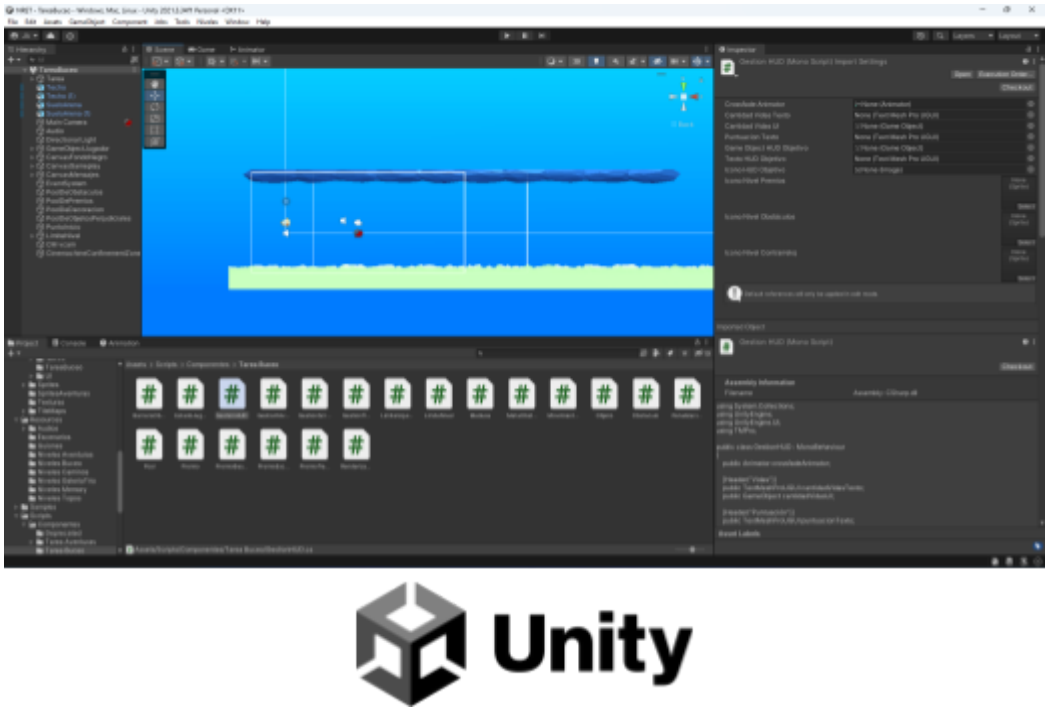

Figura 12. Entorno de desarrollo de Unity

Por otro lado, para implementar los scripts utilizados se usó Visual Studio Code [20], un IDE gratuito desarrollado por Microsoft. Este programa destaca por su gran cantidad de extensiones disponibles, así como por dar soporte a muchos lenguajes de programación.

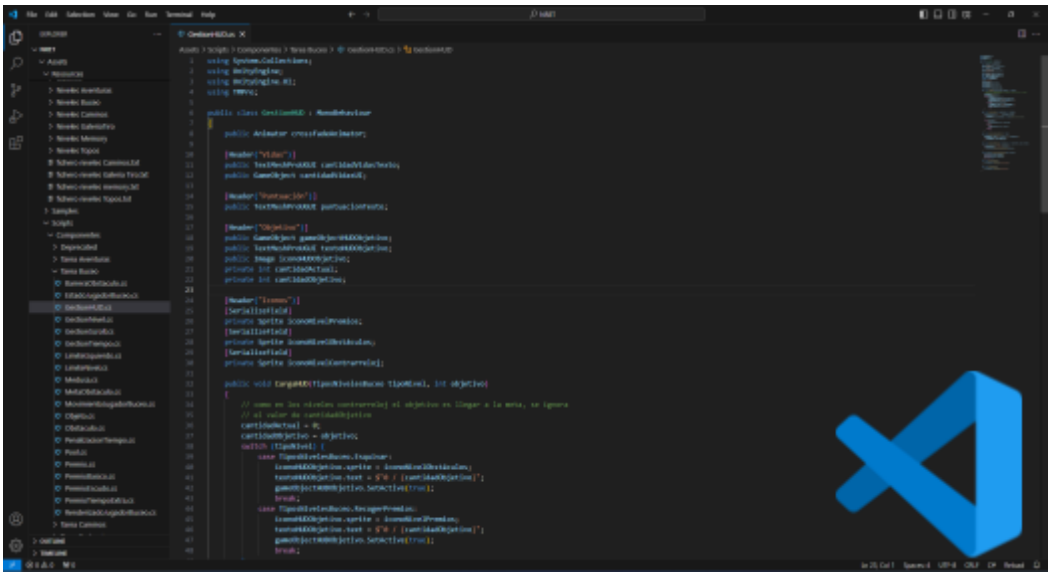

Figura 13. Interfaz de Visual Studio Code

Para el aspecto artístico del juego se utilizaron dos programas gratuitos: Blender y Krita. Por un lado, Blender [21] es un software open source y multiplataforma de modelado y animación 3D. Además, permite al usuario programar scripts en Python para automatizar tareas, y realizar simulaciones físicas como cabello, fluidos, y partículas.

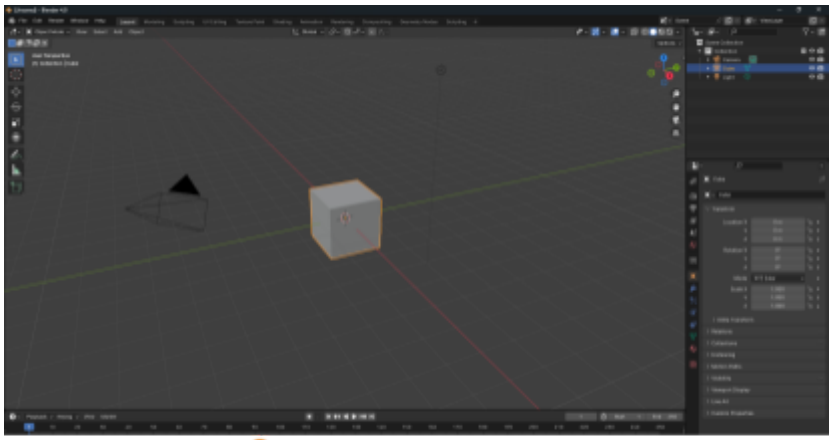

Figura 14. Interfaz de Blender

Por otro lado, Krita [22] es un programa gratuito y de código abierto de dibujo profesional. A pesar de ser un software libre, Krita destaca por ofrecer funciones que lo acercan a programas de dibujo más utilizados en la industria, como Photoshop. Algunas de estas funciones son la existencia de una gran variedad de pinceles y el soporte de dibujos tanto vectoriales como de mapa de bits. No obstante, el juego desarrollado utiliza, mayoritariamente, recursos en 3D, por lo que Krita solo se ha utilizado para realizar el degradado del skybox.

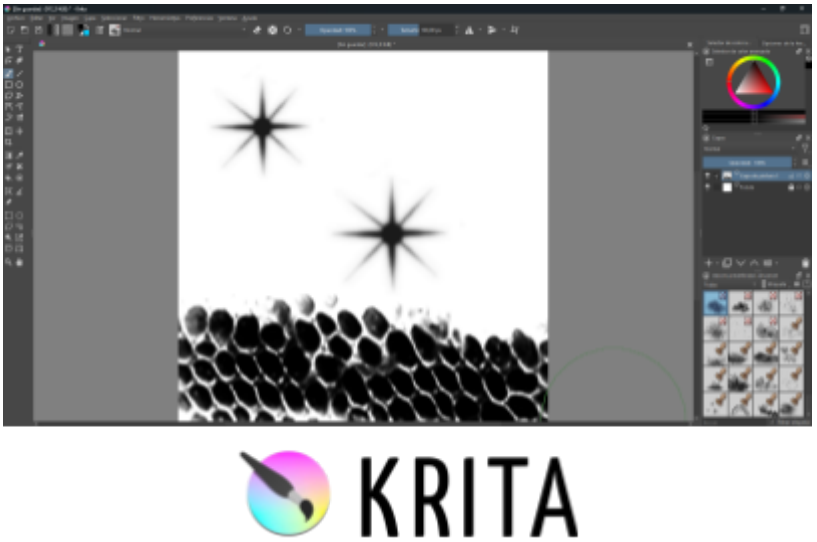

Figura 15. Interfaz de Krita

Por último, también se usó Plastic SCM [23]. Este software es un sistema de control de versiones hecho por una empresa española, Códice Software, y que forma parte de Unity DevOps. Por ello, Plastic SCM está integrado en Unity, y permite realizar el control de versiones de los proyectos hechos en dicho motor con más facilidad. Concretamente, se ha utilizado su aplicación de escritorio para poder acceder al repositorio del proyecto y crear una rama local donde desarrollar el juego.

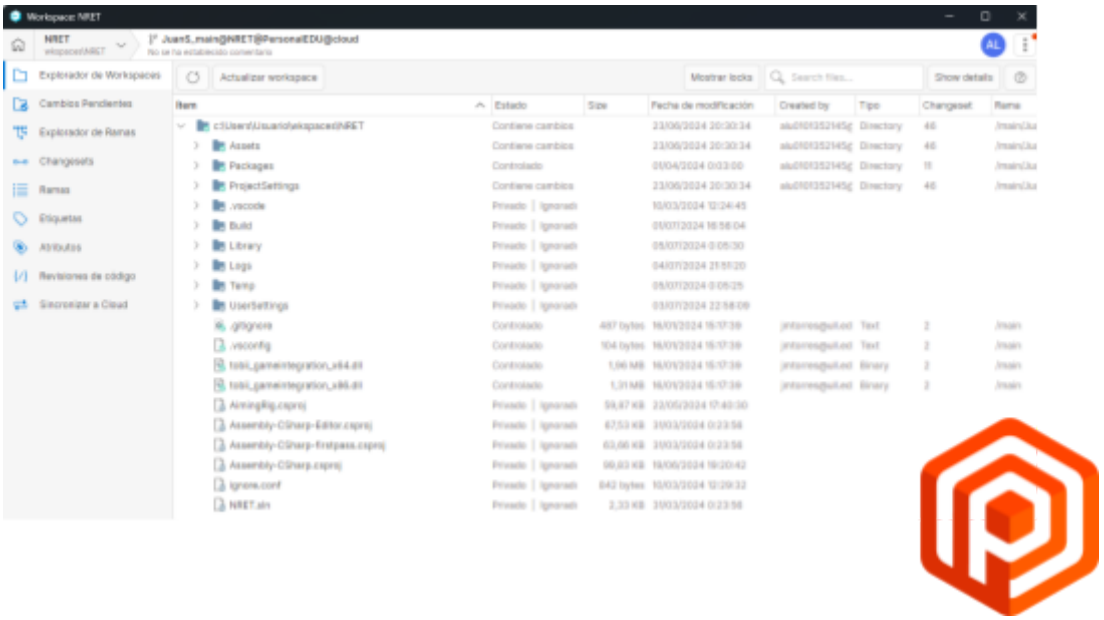

Figura 16. Interfaz de Plastic SCM

Capítulo 3
Características principales del juego

3.1. Bases del juego

Al igual que en los otros juegos del proyecto NRET, este también cuenta con una serie de niveles que el jugador debe ir superando para avanzar, de modo que el juego termina cuando se completa el último nivel.

Sin embargo, el núcleo principal del juego es el mismo en todos. En cada nivel, el jugador debe dirigir con la mirada a un pez para esquivar obstáculos y recoger los premios que se encuentra en su camino. El pez se mueve automáticamente hacia la derecha, por lo que es necesario que el jugador reaccione rápidamente a su entorno para subir y bajar el pez, acercándolo o alejándolo de los elementos que aparecen por la derecha.

Figura 17. Captura de pantalla del juego “Buceo”

Por otro lado, el jugador comienza cada nivel con tres vidas y pierde una cada vez que se choca con un obstáculo. No hay modo de recuperarlas. Si se gastan, la partida termina y el jugador vuelve al menú principal, desde donde puede elegir reintentar el nivel.

3.1.1. Tipos de niveles

Cada nivel tiene asociado uno de los siguientes objetivos:

- **Recoger X premios**: el jugador debe obtener X premios de cualquier tipo para superar el nivel.
- **Esquivar X obstáculos**: el jugador tiene que pasar a través de X muros sin chocarse para superar el nivel.
- **Contrarreloj**: para acabar el nivel, el jugador simplemente debe llegar a la meta, que está a una cierta distancia del comienzo del nivel, antes de que se acabe el tiempo.

Es importante destacar que, aunque el objetivo de cada nivel sea de un solo tipo, todos los niveles tienen tanto obstáculos como premios, que aumentan la puntuación del jugador cuando son esquivados y recogidos, respectivamente, como se ve en la Figura 18. La única excepción es el primer nivel, que no tiene obstáculos, pues es el tutorial.

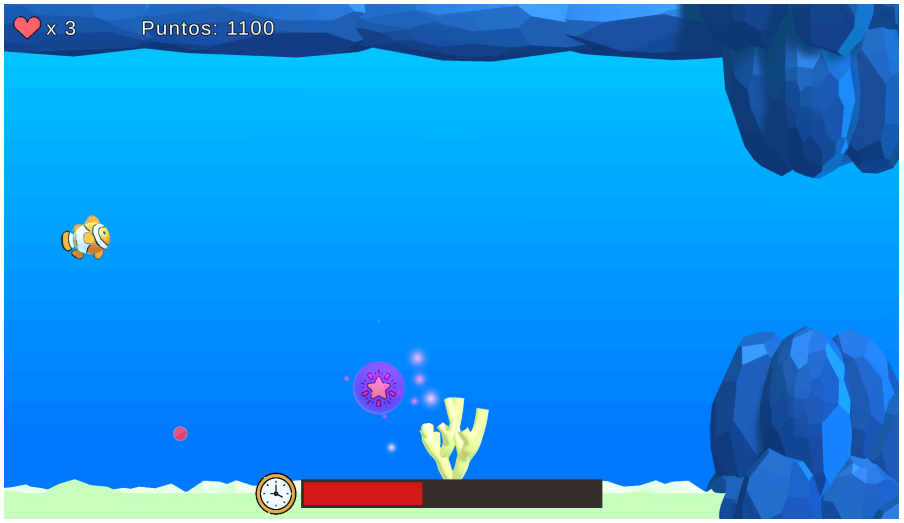

Figura 18. Captura de pantalla de un nivel contrarreloj

Además, los niveles contrarreloj son los únicos que tienen un tiempo límite para ser superados. La única forma de perder la partida en un nivel de uno de los otros dos tipos es quedarse sin vidas.

A lo largo de los niveles, el jugador se puede encontrar con elementos de dos tipos distintos: obstáculos y objetos.

3.1.2. Obstáculos

La principal característica de los obstáculos es que, cuando el jugador colisiona con ellos, este pierde una vida, se reduce un poco su velocidad durante un tiempo, y se le otorgan unos instantes de invulnerabilidad para que no se quede sin vidas inmediatamente. Por otro lado, los objetos pueden ser recogidos por el jugador al chocarse con ellos, lo que hace que desaparezcan y se aplique un efecto concreto al jugador, que puede ser beneficioso o perjudicial.

Existen varios tipos de obstáculos. Los más comunes son las rocas grandes y azules, que poseen un hueco por el que el jugador puede pasar. Si consigue atravesar el hueco sin chocarse con la roca, se considera que ha pasado el obstáculo exitosamente. Esto concede puntos al jugador y, si el objetivo del nivel es cruzar a través de X obstáculos, este contador se incrementa. Además, a medida que aumenta la dificultad de los niveles, los obstáculos cambian de forma para que sean más complicados de atravesar. Así, hay tres tipos de obstáculos:

- **Obstáculos de nivel 1**: son la versión más básica.
- **Obstáculos de nivel 2**: dejan un hueco ligeramente más estrecho entre las rocas, y son un poco más anchas, por lo que el jugador debe mantener la mirada fija para atravesarlas.
- **Obstáculos de nivel 3**: son mucho más gruesos que los anteriores, y su hueco puede ser horizontal o diagonal.

Además, en los últimos niveles comienzan a aparecer medusas moradas que también dañan al jugador, y que se mueven verticalmente. Sin embargo, estas no otorgan puntos si se evita chocar con ellas. La Figura 19 muestra ejemplos de los tres tipos de rocas, y una medusa. A la izquierda aparece el pez del jugador, para que se puedan comparar los tamaños de los huecos con respecto a él.

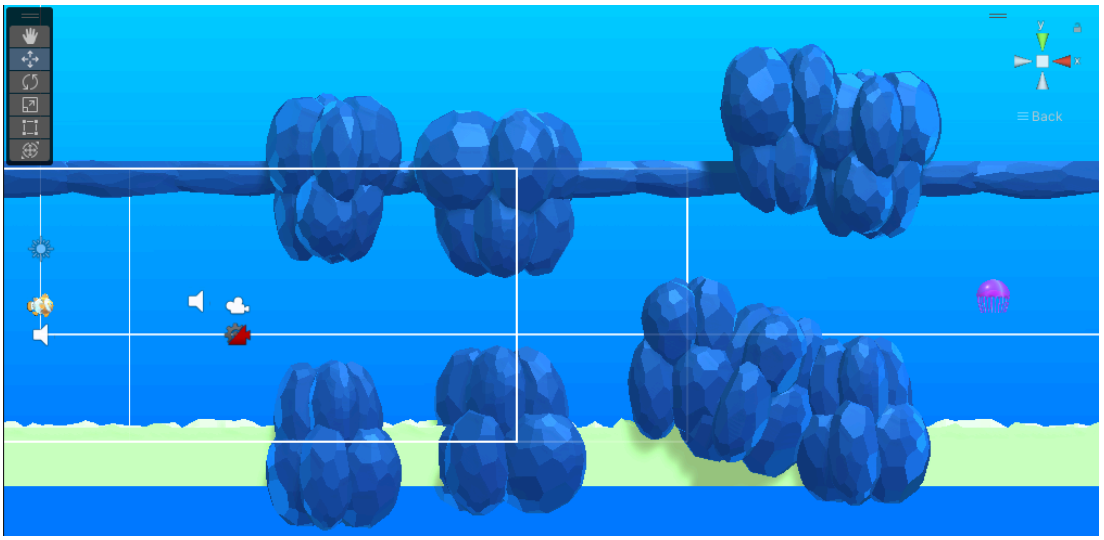

Figura 19. Ejemplos de obstáculos del juego. Las rocas son, de izquierda a derecha, obstáculos de nivel 1, 2 y 3. También aparece una medusa de los niveles avanzados

3.1.3. Objetos

Los objetos recolectables están formados por una esfera y una imagen que indica el efecto del mismo. Existen cuatro tipos de objetos:

- **Premio morado**: no tiene características especiales aparte de sumar 100 puntos al jugador, lo cual es el doble de lo que añaden los demás premios.
- **Premio verde**: suma 50 puntos y añade 5 segundos al contador de tiempo del jugador. Solo aparece en los niveles contrarreloj.
- **Premio amarillo**: suma 50 puntos y cubre al jugador con un escudo. Si el jugador choca con una roca o medusa y tiene una barrera, esta desaparece y el jugador recibe unos segundos de invulnerabilidad; pero no pierde velocidad ni vidas. Sin embargo, si el objeto con el que colisionó es un obstáculo, no se contabiliza como superado.
- **Penalización roja**: este es el único tipo de objeto perjudicial para el jugador. Su efecto es quitarle 100 puntos y 5 segundos de tiempo. Al igual que los premios verdes, solo aparecen en los niveles contrarreloj.

La Figura 20 muestra los cuatro tipos de objetos descritos anteriormente.

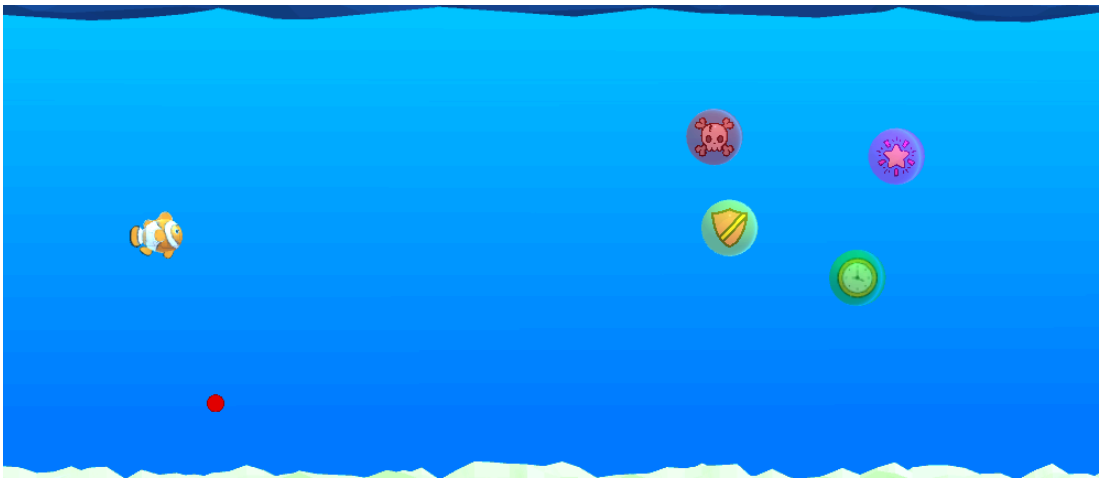

Figura 20. Objetos existentes en el juego

3.2. Niveles desarrollados

El juego cuenta con un total de 10 niveles:

- **Nivel 1**: el objetivo de este nivel es recoger 10 premios. No tiene obstáculos, ya que está más enfocado en ayudar al jugador a aprender a moverse. Los únicos premios que aparecen en él son los morados.
- **Nivel 2**: este nivel es el primero cuyo objetivo es esquivar un cierto número de obstáculos, concretamente, 10. Aquí comienzan a aparecer los premios que protegen al jugador.
- **Nivel 3**: el objetivo de este nivel es llegar a la meta, que está a 200 unidades de distancia de Unity (metros), antes de que pasen 70 segundos. Es el primero en el que aparecen los premios que suman tiempo al jugador.
- **Nivel 4**: en este nivel, el jugador debe recoger 12 premios. La principal característica de este nivel es que, a partir de aquí, la velocidad del jugador se incrementa en 1 m/s.
- **Nivel 5**: siguiendo la línea de aumentar la dificultad establecida por el nivel anterior, aquí dejan de aparecer los obstáculos de nivel 1 para dar paso a los de nivel 2. El objetivo del nivel es esquivar 15 obstáculos.
- **Nivel 6**: en este segundo nivel contrarreloj, el jugador debe alcanzar la meta, que está a 300 metros, en 75 segundos o menos. Como añadido, comienzan a aparecer los objetos que restan tiempo.
- **Nivel 7**: este es el tercer nivel cuyo objetivo es recoger premios. Concretamente, el jugador tiene que conseguir 15 objetos para ganar; pero

debe tener cuidado con las medusas, que aparecen a partir de este nivel. Además, la velocidad del jugador vuelve a incrementarse en 1 m/s.

- **Nivel 8**: el objetivo de este nivel es esquivar 20 obstáculos. A partir de aquí, pueden aparecer obstáculos tanto de nivel 2 como de nivel 3. Sin embargo, para aumentar la dificultad, ya no aparecen premios de escudo.
- **Nivel 9**: este es el último nivel contrarreloj del juego y, en él, el jugador debe llegar a la meta, que está a 400 metros, en 90 segundos o menos. No obstante, se debe tener especial cuidado, ya que no aparecen premios de tiempo, pero sí penalizaciones.
- **Nivel 10**: es el nivel final del juego. Para superarlo, el jugador debe recoger 20 premios.

3.3. Acceso al juego desde el menú principal

El juego NRET está organizado de forma que el jugador, tras ejecutarlo, debe elegir el perfil de jugador que quiere utilizar. Después, aparece el menú de la Figura 21, que muestra los juegos disponibles. Entre ellos se encuentra “Buceo”, el juego desarrollado para este TFM.

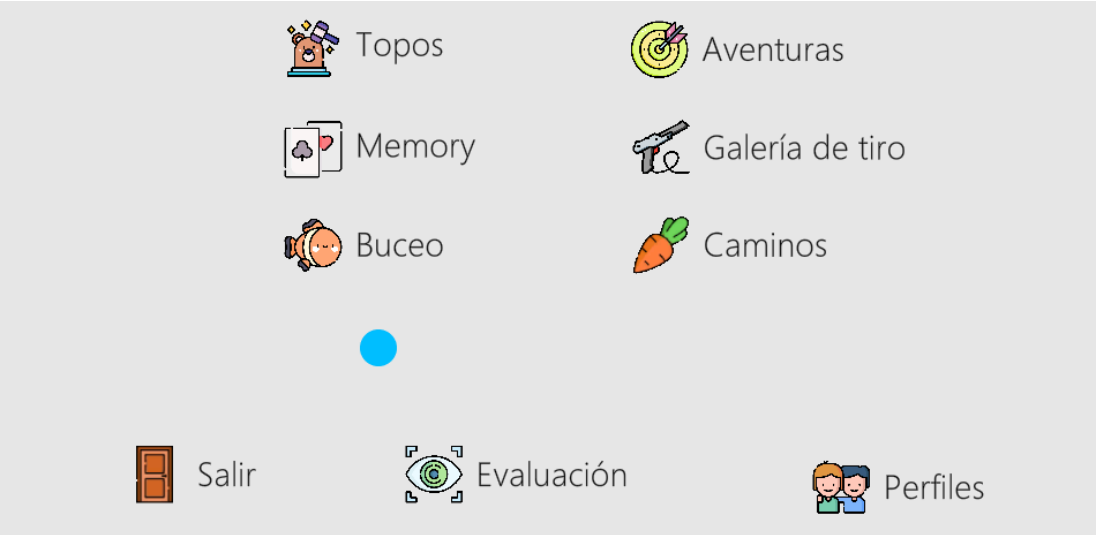

Figura 21. Menú principal del juego

Para comenzar, el jugador debe elegir seleccionar con la mirada el texto del juego “Buceo” o su icono. Al hacer esto, aparecerán en pantalla la puntuación actual del jugador y el nivel que debe superar a continuación, tal y como muestra la Figura 22. Una vez aquí, el jugador puede mirar a “Volver” para que se muestre de nuevo la pantalla de selección de juego, o puede pulsar “Jugar” para comenzar el nivel.

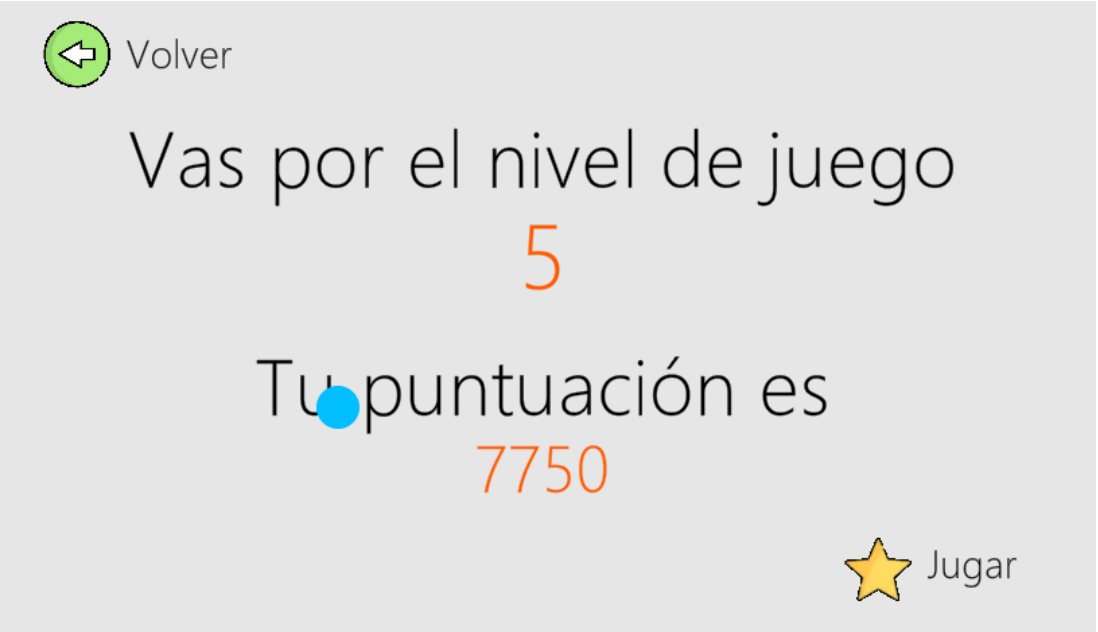

Figura 22. Pantalla previa al juego “Buceo”

3.4. Vídeos del juego

A modo de demostración, se han grabado dos vídeos en los que se muestran algunos de los niveles del juego. Para grabarlos, se ha utilizado una versión especial que permite controlar al pez utilizando el ratón en lugar del eye tracker.

Por un lado, en el primer vídeo, se recorren los tres primeros niveles, mostrando cada uno de los tres tipos de objetivos, los efectos de los premios, y lo que ocurre cuando el pez se choca, entre otros.

Por otro lado, en el segundo vídeo se enseña el noveno nivel del juego, que es también el último nivel contrarreloj del mismo. En él se muestra lo que ocurre al chocar con las medusas y las penalizaciones de tiempo, y cómo el jugador pierde la partida al quedarse sin vidas.

Los enlaces a ambos vídeos están disponibles en la Bibliografía.

Capítulo 4
Detalles de implementación

En este capítulo se explicarán con detalle cómo se han programado los diferentes sistemas del juego.

4.1. Gestión del juego

4.1.1. Datos del nivel

Todos los juegos incluidos en el proyecto NRET, basan los datos de sus niveles en `ScriptableObjects` creados a partir de la clase `NivelScriptable`. Esta clase incluye campos para definir el número del nivel, su dificultad y si es un nivel normal, un tutorial, o una demo. Como este script ya existía desde antes de la creación del juego desarrollado para este TFM, se creó un nuevo script que heredase de `NivelScriptable` para añadir las propiedades necesarias para definir cada nivel. Esta clase, llamada `NivelBuceoScriptable`, aparece en el Apéndice A.1, e incluye las siguientes variables:

- Tipo de nivel: si es de recoger premios, esquivar obstáculos, o contrarreloj. Esto se ha implementado con un enum llamado `TipoNivelesBuceo`.
- Espacio entre cada par de obstáculos: al igual que el anterior, los posibles valores de esta variable se han definido con un enum, `EspacioEntreObstaculosEnBuceo`, cuyos valores posibles son Poco, Medio, Mucho y Mixto.
- Probabilidades de que aparezcan ciertos elementos entre cada par de obstáculos: como se verá más adelante, cada dos obstáculos puede aparecer, un objeto beneficioso o perjudicial, o puede que no aparezca nada. Las probabilidades en porcentaje de que ocurran estos sucesos se almacenan en el ScriptableObject como tres variables de tipo float.
- Probabilidad de que aparezca un premio entre dos obstáculos.
- Distancia en X hasta el límite del nivel, desde la posición X = 0.
- Listas de obstáculos, premios, objetos perjudiciales, y elementos decorativos del nivel.
- Valores relacionados con el objetivo del nivel: la clase tiene tres atributos para guardar el número de premios a recoger, los obstáculos que hay que esquivar, y los segundos disponibles para alcanzar la meta. Se utiliza el que corresponda según el tipo de nivel.
- Velocidades base y reducida del pez: dado que se reduce la velocidad del pez durante un tiempo corto cuando se choca con un obstáculo, se

definen en los datos del nivel este nuevo valor de rapidez, y la que se usa por defecto.

4.1.2. Gestión de los eventos principales

El script que coordina a todos los demás y que contiene las funciones para controlar el estado del juego es `TareaBuceo`, disponible en el Apéndice A.2. Cuando el jugador selecciona "Jugar" en el menú principal, se cambia la escena actual a aquella donde se desarrolla toda la acción del videojuego. Una vez hecho esto, se ejecuta la corrutina `CorrutinaPartida`, que inicializa todos los datos necesarios para que comience el nivel. Para ello, guarda referencias a los scripts que gestionan elementos como la interfaz, el tiempo y la generación del nivel, y escribe la puntuación actual del jugador en la interfaz, pues esta se acumula a medida que este supera niveles.

Después, se muestran por pantalla una serie de mensajes sucesivos que indican al jugador datos sobre el nivel o lo que tiene que hacer. Para hacer que aparezcan los mensajes se ha utilizado la corrutina `MostrarMensaje`. Originalmente, esta fue utilizada para otros juegos, y permite mostrar mensajes como el de la Figura 23, definiendo el icono del aviso, el texto, la duración, y el audio reproducido. Los mensajes utilizados para este juego fueron grabados por Rebeca Villaroel, del equipo del HUC. Entre ellos se encuentran avisos para explicar al jugador los efectos de cada premio y los diferentes objetivos de los niveles. Para evitar abrumar al jugador con muchos mensajes, todos se muestran durante cuatro segundos, y los mensajes que indican el objetivo del nivel solo aparecen en la primera fase de ese tipo. En los siguientes niveles con ese objetivo, el objetivo pasa a indicarse en la interfaz con una flecha roja durante unos segundos.

Figura 23. Mensaje que se muestra en el primer nivel del juego

El único mensaje que aparece en todos los niveles es “Comenzamos”, que siempre es el último en mostrarse, y que marca el comienzo del nivel. Después de que se muestre este mensaje, se actualiza la interfaz del juego para mostrar el objetivo del nivel o la barra de tiempo según corresponda, e iniciar el temporizador si la fase es contrarreloj. Además, también se envían los datos del nivel al script que se encarga de la generación automática del mismo.

`TareaBuceo` también implementa las funciones que se encargan de actualizar el estado del juego cuando sucede algo importante, como que el jugador choque con un obstáculo o con un premio, así como lo que debe ocurrir cuando el jugador pierde todas las vidas, se queda sin tiempo, o supera el objetivo del nivel.

- Si el jugador supera un obstáculo, se ejecuta el método `GestionarObstaculoPasado`, que añade a su puntuación los puntos que se obtienen por obstáculo, y aumenta el número de obstáculos esquivados. Así, si el objetivo del nivel es esquivar obstáculos, se actualiza la interfaz, y se comprueba si la cantidad de barreras evitadas es la requerida, invocando la función `JuegoGanado` si es así.
- Por otro lado, si el jugador recoge un premio, se invoca el método `GestionarPremioRecogido`. De forma similar al anterior, este actualiza el contador de premios recogidos y, si el objetivo del nivel es obtener premios, la interfaz es actualizada y se comprueba si se ha superado la meta.
- `JuegoGanado` es un método sencillo que guarda la puntuación del jugador y gestiona la escritura del registro de posición ocular. Además, hace que el jugador vuelva a la escena principal, apareciendo en la pantalla que muestra el nivel actual y la puntuación en el juego “Buceo”.
- `TareaBuceo` también dispone de los métodos `JugadorSinVidas` y `TiempoExcedido`, que son invocados cuando el jugador se queda sin vidas y cuando se acaba el tiempo en los niveles contrarreloj, respectivamente. Estas funciones llaman a otra, `JuegoPerdido`, que, al igual que `JuegoGanado`, prepara el registro de la posición ocular y hace que el juego termine. Sin embargo, cuando el jugador pierde, no se guardan los puntos que ha obtenido en ese nivel.
- Finalmente, si el jugador llega al final del nivel, se ejecuta el método `GestionarFinalDelNivel`. Esta función llama a `JuegoGanado` si el nivel es contrarreloj, pues el jugador habrá llegado a la meta. Si el objetivo de la fase es otro, invoca a la corrutina `TransicionReinicioNivel`. Esta se encarga de oscurecer la pantalla, enviar al jugador de vuelta a la posición X = 0 del mundo, y aclarar de nuevo la pantalla.

Cabe destacar que, dado que `TareaBuceo` es el script que contiene las funciones que se invocan cuando el nivel termina, también es el responsable de realizar el

registro de posiciones oculares y resultados del nivel, como se explicará en el apartado 4.6.

4.1.3. Gestión y generación del nivel

Para entender cómo se gestiona el nivel, es importante mencionar que el GameObject de `TareaBuceo` tiene como hijos dos GameObjects muy importantes. Estos objetos, LimiteIzquierdo y GestionNivel, 1 y 3 en la Figura 24, contienen los scripts con los que comparten nombre, y que se explicarán más adelante. Además, el GameObject de `TareaBuceo` (esfera amarilla 2 en la Figura 24) actualiza en cada frame su coordenada X para que coincida con la del jugador. Esto hace que los otros dos objetos se muevan con ella, de modo que el nivel se genera a medida que avanza el jugador.

Figura 24. GameObjects encargados de generar el nivel. En el juego real, estos objetos no tienen malla ni material

El código que controla la generación del nivel está dividido en varios scripts. Uno de los más importantes es `Pool`, que implementa un repositorio de objetos clonados que se instancian cuando comienza el nivel y se sacan e introducen en el pool según sea necesario. La clase `Pool` se puede ver en el Apéndice A.3.

Al comienzo del nivel se ejecuta, como se verá más adelante, el método `CrearPool` de cada GameObject con este script. Esto creará una lista de GameObjects a partir de una lista de prefabs que recibe el método por parámetro. Concretamente, para cada objeto de dicha lista, se añaden X copias al pool, donde X es un parámetro configurable del script. Además, estos objetos son desactivados en cuanto son creados, ya que se considera que los objetos no visibles están dentro del pool; mientras que los visibles están fuera del mismo.

En la escena del juego hay cuatro objetos con este script. Cada uno almacena las instancias de un tipo de objeto diferente: premios, objetos perjudiciales, obstáculos, y elementos decorativos. Cuando el nivel comienza, se invoca el

método `CrearPool` cuatro veces, pasando cada vez una de las listas de objetos del `NivelBuceoScriptable` del nivel.

`Pool` también dispone de métodos para añadir y sacar objetos del repositorio. Por un lado, el método `SacarObjeto` comprueba si hay objetos en la lista y, si es así, extrae uno aleatorio. En cambio, si el pool está vacío, crea una nueva instancia de la lista de prefabs y la devuelve. Por otro lado, el método `AñadirObjeto` simplemente devuelve el GameObject que recibe como argumento al pool.

El script más importante relacionado con la generación del nivel es `GestionNivel`, visible en el Apéndice A.4, que se encarga de tres funciones principales:

- **Cargar los datos del nivel al principio del juego**: cuando comienza el nivel, `TareaBuceo` ejecuta el método `CargarNivel` de `GestionNivel`, que recibe como argumento el objeto `NivelBuceoScriptable` del nivel. Esta función crea los pools, determina los espacios máximo y mínimo entre obstáculos y coloca el límite del nivel a la distancia marcada. Además, dependiendo del tipo de nivel cargado, se coloca en el límite del nivel un tipo de meta. Concretamente, los niveles contrarreloj utilizan una línea de meta; mientras que los demás niveles emplean un cartel con una flecha, ya que, al cruzarlos, el jugador vuelve al principio del nivel. La Figura 25 muestra la línea de meta y el cartel mencionados.

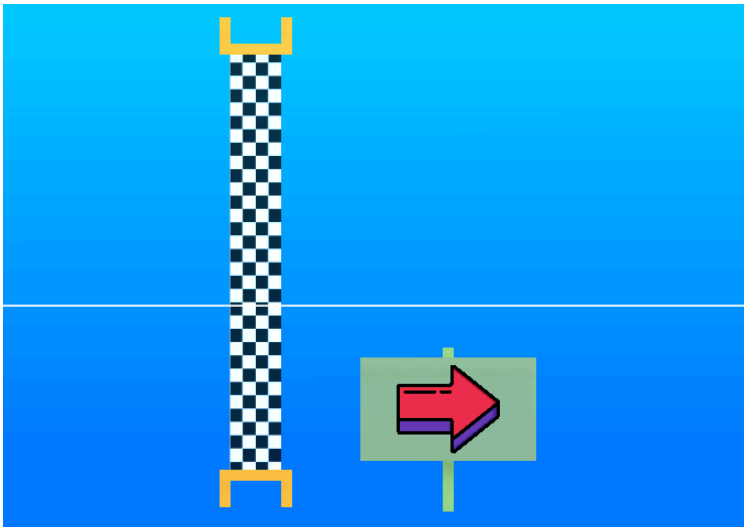

Figura 25. Metas del final del nivel

- **Generar el nivel proceduralmente**: esta es la función más importante del script, y se ejecuta mediante su `Update`. El nivel se genera por secciones que contienen un obstáculo, dos elementos decorativos, y un objeto. La

Figura 26 muestra un ejemplo de sección. Cuando la diferencia entre las coordenadas X del jugador y el obstáculo más cercano a él por la izquierda supera un umbral, se genera la siguiente sección siguiendo los siguientes pasos:

1. Se extrae un obstáculo aleatorio del pool, y se coloca en la posición del GameObject de `GestionNivel`.
2. Se calcula la distancia que debe recorrer el jugador tras superar este obstáculo para que se genere el siguiente. Este valor se obtiene como un valor aleatorio entre los espacios máximo y mínimo definidos por el nivel.
3. Se elige aleatoriamente el objeto especial que se colocará en la nueva sección. Para ello, se utiliza la función `ElegirTipoDeElementoDeSeccion`, que permite determinar la lista de la que se extraerá el objeto utilizando las probabilidades de que este sea un premio, un elemento perjudicial, o nada.
4. Si la función determina que se debe extraer un objeto, se saca del pool correspondiente y se coloca en el centro de la sección, con cierta variabilidad en X y en Y.
5. Finalmente, se extraen dos elementos decorativos del pool y se colocan en la sección.
6. Cabe destacar que el método `Update` de `GestionNivel` solo se ejecuta si el jugador está a la izquierda del límite del nivel y si el espacio entre estos dos objetos es menor que el espacio máximo entre obstáculos. De este modo, se impide que se generen secciones cuando el jugador está muy cerca de la meta.

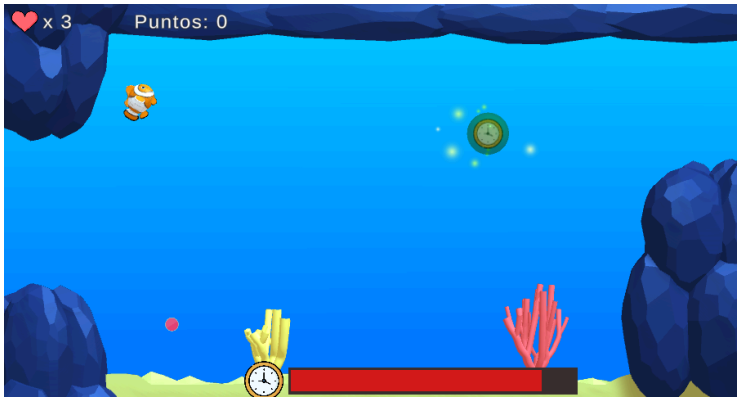

Figura 26. Ejemplo de sección del nivel

- **Devolver los objetos eliminados a su pool**: cuando un objeto debe desaparecer de la escena, por ejemplo, cuando el jugador recoge un premio, se invoca un método de `GestionNivel` para que se encargue de

ello. Así, las clases externas pueden invocar a `EliminarObstaculo`, `EliminarPremio`, `EliminarObjetoPerjudicial` o `EliminarDecoracion`. Todas las funciones desactivan el objeto eliminado para que desaparezca y lo devuelven al pool correspondiente. `GestionNivel` también tiene un método adicional para devolver a los pools todos los elementos fuera de ellos, llamado `EliminarTodo`.

Por otro lado, como se vio anteriormente, uno de los hijos de TareaBuceo tiene el script `LimiteIzquierdo`, visible en el Apéndice A.5. Este objeto es un Trigger que reacciona a los obstáculos, medusas, premios, elementos perjudiciales, y decoración que choca con él. Así, cuando un objeto colisiona con `LimiteIzquierdo`, este comprueba su tag, e invoca a una de las cuatro funciones de `GestionNivel` encargadas de eliminar objetos, según el tipo de GameObject.

Otro script importante es `LimiteNivel`, disponible en el Apéndice A.6, que es un componente del GameObject de la meta. Este objeto también es un Trigger, que reacciona cuando el jugador colisiona con él invocando a los métodos `GestionarFinalDelNivel` de `TareaBuceo` y `EliminarTodo` de `GestionNivel`. Como se ha explicado anteriormente, cuando el jugador llega al final del nivel, este termina si es un nivel contrarreloj; pero vuelve al principio si no lo es. Esto se hizo así para evitar que, si el jugador tarda demasiado en completar el nivel, su coordenada X crezca demasiado.

El último script de gestión del nivel es `GestionScroll`, disponible en el Apéndice A.7. Esta clase se encarga de hacer que los GameObjects que representan el suelo y el techo de la escena se coloquen adecuadamente para que la cámara vea siempre un techo y un suelo continuos. Concretamente, tanto el suelo como el techo tienen dos GameObjects, como se ve en la Figura 27, que son recolocados en el Update de `GestionScroll` según el movimiento de la cámara. Como el jugador siempre se mueve hacia la derecha, si los objetos no se mueven, el techo y el suelo se quedarán atrás. Por ello, lo que se hace es comprobar, en cada frame, si el jugador está lo suficientemente cerca del final del techo y el suelo. Si es así, el objeto que está más a la izquierda se coloca a la derecha del otro.

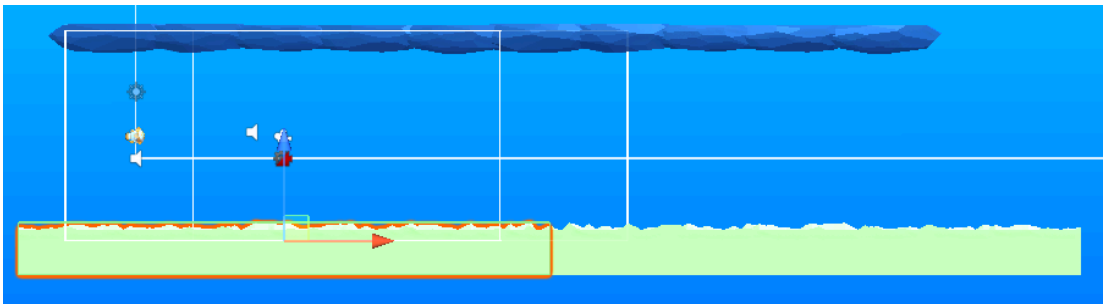

Figura 27. GameObjects del techo y el suelo. En naranja se marca el Collider de uno de ellos

4.1.4. Interfaz

La interfaz gráfica del juego tiene pocos elementos. Sólo la cantidad de vidas y la puntuación del jugador se muestran en todos los niveles. La información que no es común a todos los niveles es la que indica el objetivo del mismo, es decir, cuántos premios ha recogido el jugador de los que tiene que coger, cuántos obstáculos ha esquivado de los que se le asignaron, y cuánto tiempo le queda. Sin embargo, al comienzo de cada nivel, la parte de la interfaz que indica el objetivo de la fase se indica por una flecha roja, permitiendo al jugador saber qué debe hacer para superar el nivel sin decírselo explícitamente al principio en un mensaje. Las Figuras 28, 29 y 30 muestran la interfaz de cada tipo de nivel.

Figura 28. Interfaz de los niveles en los que hay que recoger premios

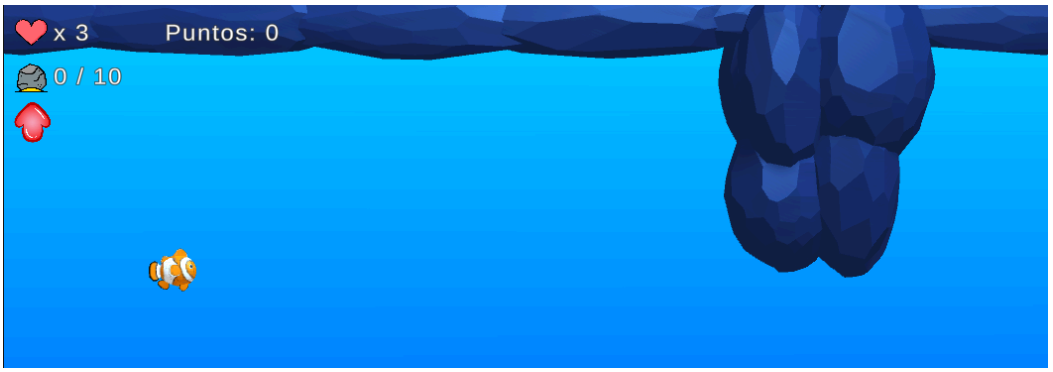

Figura 29. Interfaz de los niveles en los que hay que esquivar obstáculos

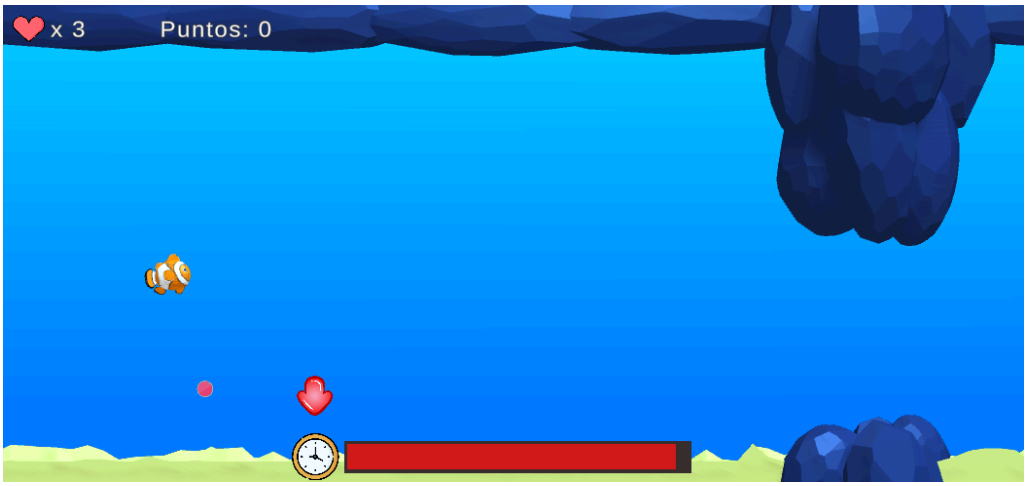

Figura 30. Interfaz de los niveles contrarreloj

El script encargado de actualizar la interfaz cuando es necesario es `GestionHUD`, que se puede ver en el Apéndice A.8. Su método `CargaHUD` es invocado por `TareaBuceo` cuando comienza el nivel para activar o desactivar los elementos de la interfaz, según corresponda al tipo del nivel actual. Así, este método recibe como argumentos un valor del enum de tipos de niveles, y el número objetivo del nivel. Es decir, recibe el número de premios a recoger o de obstáculos a esquivar. Luego, si el objetivo de la fase es conseguir objetos, se muestra la parte de la interfaz con el objetivo, se cambia el icono para que sea el que representa los premios, tal y como aparece en la Figura 28, y se indica que el jugador lleva 0 objetos recogidos en total. Si el objetivo de la fase es esquivar obstáculos, ocurre lo mismo, pero mostrando el icono de la roca que aparece en la Figura 29. Por último, si el nivel es contrarreloj, `CargaHUD` no hace nada, pues la barra de tiempo es gestionada por otro script que se explicará más adelante.

`GestionHUD` también dispone de tres métodos para actualizar la información que aparece en la parte superior de la pantalla:

- **`ActualizarVidas`**: recibe el nuevo número de vidas y un booleano que indica si el jugador ha recibido daño. Si este valor es true, se utiliza la función `ShakeScale` de iTween para sacudir el texto de las vidas. Este texto cambia de color dependiendo del número de vidas restantes, siendo blanco para dos o más vidas, amarillo para una sola, y rojo para ninguna.
- **`ActualizarProgresoObjetivo`**: este método recibe como parámetro el incremento a aplicar al número de obstáculos esquivados o de premios recogidos, que normalmente será uno. Con ello, actualiza el texto de la interfaz y, si esa cantidad es igual o mayor que el objetivo colorea el texto de verde para indicar que el jugador ha ganado.

- **`ActualizarPuntuación`**: recibe la nueva puntuación del jugador, ya agregada al valor anterior, un booleano para indicar si se debe resaltar el texto de un color, y el color elegido para ello. Después de actualizar el texto de la puntuación, el método comprueba si el booleano es true y, si es así, invoca la corrutina `ResaltarTextoPuntuacion`, que recibe como parámetro el color elegido. Dicha corrutina cambia el color del texto por el elegido, espera 0'3 segundos, y lo devuelve a su color original.

Los últimos dos métodos de `GestionHUD` son `OscurecerPantalla` y `AclararPantalla`, que se encargan de activar los triggers "StartCrossfade" y "EndCrossfade", respectivamente. Los triggers se utilizan para disparar las transiciones de la animación del crossfade, que sucede cuando el jugador llega al final del nivel y debe volver al principio. En la escena hay un Canvas con un panel negro que cubre toda la pantalla, de modo que se modifica su transparencia con un Animator para implementar el crossfade. El Animator se muestra en la Figura 31, y consta de tres estados :

- **Start**: es el estado inicial y no tiene animación.
- **crossfade_start**: reproduce una animación que incrementa gradualmente el valor de opacidad del fondo negro de 0 a 1.
- **crossfade_end**: usa una animación que, al contrario que la anterior, decrementa gradualmente la opacidad desde 1 hasta 0.

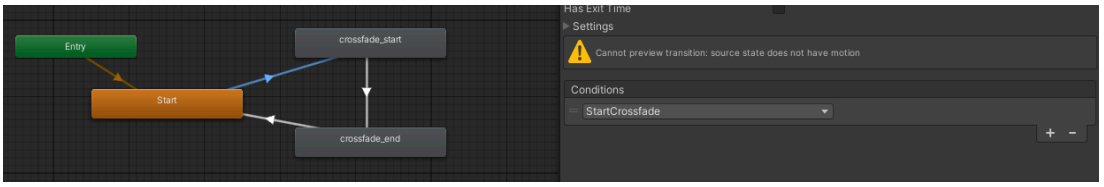

Figura 31. Animator del crossfade

4.1.5. Gestión del tiempo

`GestionTiempo`, que se puede ver en el Apéndice A.9, es el script que se encarga tanto de gestionar el temporizador del nivel como de la barra de tiempo. Contiene las variables necesarias para almacenar los segundos restantes y el tiempo máximo.

En primer lugar, cuando el nivel actual es contrarreloj, `TareaBuceo` llama al método `IniciarTemporizador` de `GestionTiempo`, que recibe como parámetro el tiempo máximo, para inicializar las variables mencionadas anteriormente, mostrar la barra de tiempo, y activar un booleano que indica si el temporizador está activo o no. Así, en el método Update de `GestionTiempo`, si dicho booleano es true, se

resta Time.deltaTime a los segundos restantes. Si este valor es inferior o igual a cero, quiere decir que el jugador se ha quedado sin tiempo, así que se invoca al método `TiempoExcedido` de `TareaBuceo` para terminar el juego. Por otro lado, si todavía queda tiempo, se calcula el porcentaje de tiempo restante, dividiendo los segundos que quedan entre el tiempo máximo, y se asigna este valor a la propiedad `fillAmount` de la barra de tiempo. Esto se hace así porque el relleno rojo de la barra es una imagen de tipo Filled, como se muestra en la Figura 32, por lo que se puede definir qué porcentaje de la imagen se muestra.

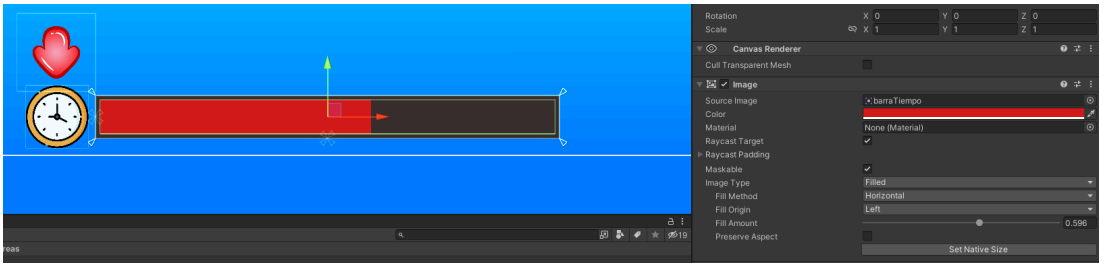

Figura 32. Propiedades de la barra de tiempo

`GestionTiempo` también tiene dos métodos adicionales, `ParарTemporizador` y `AñadirTiempo`. El primero permite asignar false al booleano, permitiendo que se reste `Time.deltaTime` en `Update`. El segundo sirve para agregar al contador de segundos restantes un float que se pasa por parámetro. Después de sumar este valor, se asegura que no supera el número de segundos con los que se comienza el nivel, para que no se pueda acumular tiempo sin límite.

4.2. Implementación del pez

Una vez han sido presentados los sistemas principales del juego, a continuación se explicará en detalle cómo funciona el pez que controla el jugador.

4.2.1. Estructura

El GameObject del pez está formado por cuatro objetos principales:

- **GameObjectJugador**: es un objeto sin malla que contiene el SphereCollider que envuelve el pez, y el Rigidbody, usado para moverlo y evitar que atraviese el techo y el suelo. Además, contiene los scripts que se utilizan para gestionar el funcionamiento del pez. Estos scripts se describirán más adelante y son `MovimientoJugadorBuceo`, `EstadoJugadorBuceo`, y `RenderizadoJugadorBuceo`.

- **Pez**: este objeto agrupa todas las partes del cuerpo del pez y contiene un Animator, que se utiliza para ejecutar la animación que permite que el pez mueva las aletas.
- **Escudo**: es el GameObject del escudo que obtiene el jugador al recoger un premio de escudo. Se trata de una esfera transparente y naranja que lo envuelve. Sin embargo, no tiene Collider y solo está para indicar que el escudo está activo, pues siempre se utiliza el SphereCollider de GameObjectJugador para detectar las colisiones. El escudo permanece desactivado hasta que se recoge un premio de escudo, desapareciendo de nuevo cuando el jugador choca con un obstáculo.
- **PartículasEscudo**: es el sistema de partículas que se emite cuando el escudo se rompe, que se muestra en la Figura 34.

La jerarquía que forman estos cuatro objetos se muestra en la Figura 33.

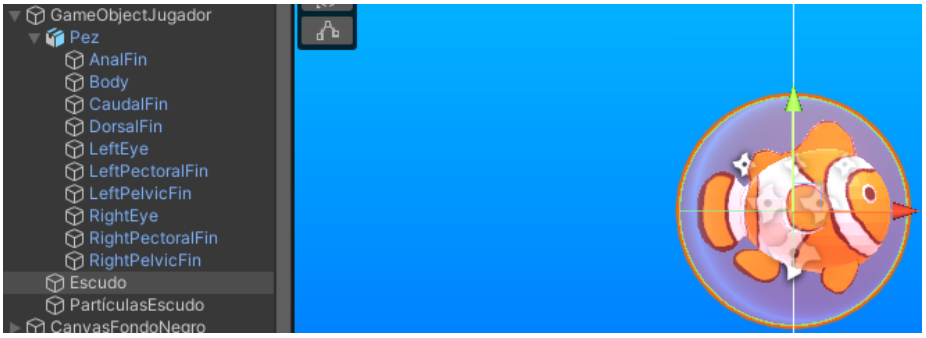

Figura 33. Estructura del GameObject del pez

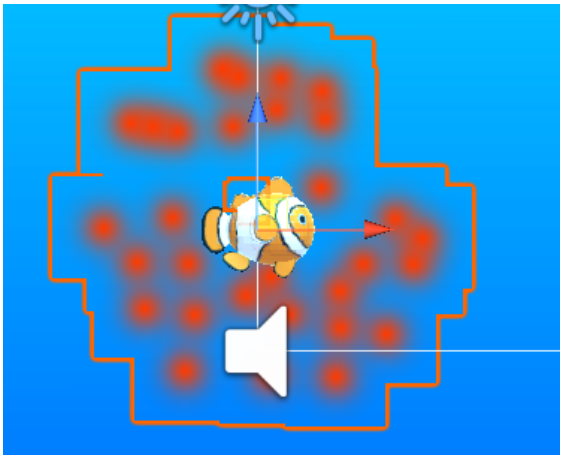

Figura 34. Sistema de partículas del escudo

4.2.2. Movimiento

El movimiento del jugador se basa en el avance automático, pues el pez se mueve siempre en la dirección en la que esté mirando. De este modo, lo que hace el jugador con la mirada es cambiar la dirección de movimiento del pez, ya que está programado para dirigirse siempre hacia el punto donde el jugador está centrando la mirada.

El script `MovimientoJugadorBuceo` se encarga de actualizar la posición del jugador en cada `FixedUpdate`. Este script se puede ver en el Apéndice A.10. En el método `Start` se obtiene de `TareaBuceo` los valores de las velocidades base y reducida del nivel. A partir de ese momento, usa la posición registrada de la mirada del jugador para dirigir el pez.

Como el pez utiliza un Rigidbody, el código que se encarga de su movimiento se debe ejecutar en `FixedUpdate`. Al igual que en otras funciones del programa, al principio de `FixedUpdate` se comprueba si la tarea está bloqueada, saliendo de la función si es así. Si no, se obtiene de la API de Tobii una instancia de la estructura `GazePoint`, que almacena la información procedente del eye tracker acerca del punto al que está mirando el jugador. Después, se comprueba si esa información es válida y, si lo es, esa posición se convierte al espacio de coordenadas del mundo. Con esto se crea un vector, `posicionAMirar`, que indica la posición a la que debe mirar el pez. Para hacerlo, se toma como coordenada X la suma de la posición en X del pez y un desplazamiento, mientras que la coordenada Y es la de `posicionAMirar`, y la Z se ignora por completo.

Después el pez se orienta hacia `posicionAMirar` utilizando la función `Lerp` de la clase `Vector3`, que permite interpolar entre dos vectores. En este caso, el vector inicial es el `right` del jugador, y el objetivo es la resta normalizada del vector que une `posicionAMirar` con la posición del jugador. Además, la interpolación se realiza a una velocidad marcada por el producto de la velocidad actual del pez y `Time.fixedDeltaTime`. Por último, el pez se mueve en la dirección en la que mira utilizando el método `MovePosition` de su Rigidbody.

4.2.3. Estado

El script `EstadoJugadorBuceo`, visible en el Apéndice 11, se encarga de gestionar el estado interno del jugador, que incluye las vidas, la velocidad, y el escudo, entre otros elementos.

En primer lugar, cuando el jugador se choca con un objeto que debe hacer que pierda una vida, como una roca, se utiliza el método `RecibirGolpe`. Al igual que en otras funciones, esta también se cancela si la tarea está bloqueada; pero, si no

lo está, actúa de una forma u otra dependiendo de si el jugador está utilizando el escudo. Si es así, se invoca la función `RomperEscudo`, que se verá más adelante. Si no lo está usando y el jugador no tiene activado el flag que indica que es invulnerable, quiere decir que debe recibir el impacto y sufrir sus efectos. Concretamente, se utiliza `iTween.ShakeScale` para hacer que el pez tiemble. También se inicia la corrutina `CorrutinaSacudirCamara` para que la cámara que sigue al pez también se mueva, con el objetivo de aumentar la sensación recibida por el impacto. Después se utiliza el método `PerderVida` para procesar la resta de una de las vidas del jugador y el posible fin de la partida. Finalmente, se invocan las corrutinas `CorrutinaInvulnerabilidad` y `CorrutinaPerdidaVelocidad` para hacer que el jugador sea invulnerable y pierda velocidad durante un tiempo. De este modo, se consigue que los impactos afecten negativamente en los niveles contrarreloj, y también se evita que en un solo impacto se pierdan todas las vidas, pues, como se verá más adelante, el jugador puede atravesar las rocas.

El método `RomperEscudo` comienza reproduciendo el audio que indica que el escudo se ha roto y desactivando el flag `EscudoActivo`. Después, se reproduce el sistema de partículas del escudo y se desactiva su GameObject para hacer que desaparezca. Por último, se invoca la corrutina `CorrutinaInvulnerabilidad` para que el jugador sea invencible durante un periodo de tiempo. El motivo por el que esto ocurre también cuando el jugador tiene el escudo puesto es que, si no fuera así, el jugador chocaría de nuevo al perder el escudo, pues la colisión con las rocas se comprueba en todos los frames. Por otro lado, `EstadoJugadorBuceo` también tiene una función para activar el escudo, llamada `ActivarEscudo`. Este método es invocado por los premios de protección cuando son recogidos.

La función `PerderVida` mencionada anteriormente simplemente decrementa en uno el número de vidas del jugador e invoca al script `GestionHUD` para que actualice la interfaz con la nueva cantidad de vidas, pasando `true` como parámetro para indicar que el jugador ha recibido daño. Después, se reproduce el efecto de sonido del choque, y se comprueba si el número de vidas restantes es cero, llamando al método `JugadorSinVidas` de `TareaBuceo` si es así para que termine la partida.

Por otro lado, `EstadoJugadorBuceo` tiene tres corrutinas ya mencionadas:

- **`CorrutinaSacudirCamara`**: aplica ruido Perlin a la cámara Cinemachine que sigue al jugador durante unos segundos.
- **`CorrutinaInvulnerabilidad`**: al principio de esta función se activa el flag de invencibilidad. Luego se calcula el número de parpadeos, es decir, ciclos en los que el pez desaparece y aparece para indicar que no puede recibir daño, a partir de la duración de la invencibilidad y la de cada parpadeo. Así, en cada parpadeo, ejecuta la función CambiarVisibilidad

que se verá más adelante para ocultar y mostrar el pez esperando un tiempo entre cada llamada. Cuando se han realizado todos los parpadeos, se desactiva el flag de invulnerabilidad para que el pez pueda recibir daño.

- **`CorrutinaPerdidaVelocidad`**: fija el valor de la propiedad `VelocidadActual` del script `MovimientoJugadorBuceo` al de la velocidad reducida en ese nivel, espera una cierta cantidad de segundos, y luego la devuelve a su valor original.

La función del método `CambiarVisibilidadPez` es recorrer la lista de partes del cuerpo del pez y activar o desactivar su renderizado según indique el booleano pasado como parámetro. Es el único método de `RenderizadoJugadorBuceo`.

4.2.4. Modelo y animación

Como muestra la Figura 35, tanto el modelo del pez como su material fueron creados en Blender. Las diferentes partes del cuerpo del pez permanecieron como mallas separadas dentro del fichero FBX para facilitar la animación del pez en Unity. Esto también simplificó la creación del mapa UV del pez y su posterior pintado, ya que permitió que cada pieza fuera separada del resto en el mapa UV. Al importar el fichero FBX a Unity, se creó automáticamente un material para la malla del pez.

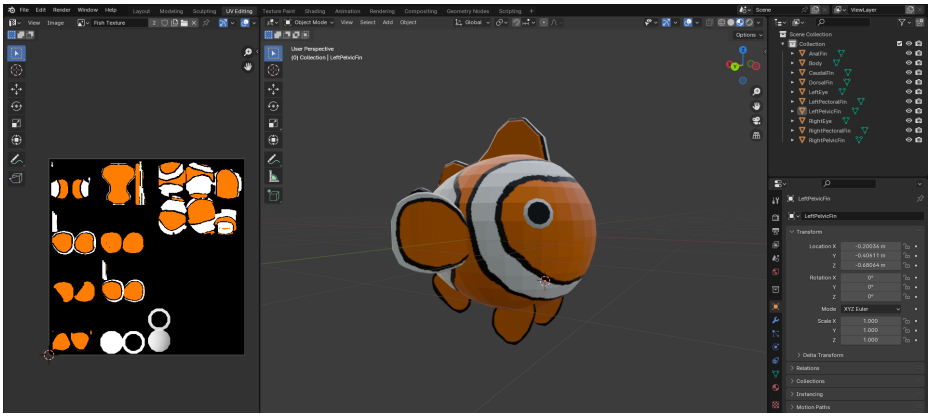

Figura 35. Ventana de Blender con el modelo y el material del pez

En cuanto a la animación, esta se creó directamente en Unity, asignando al GameObject que contiene las partes del cuerpo del pez un componente Animator. De este modo, se creó una pista en la animación para cada aleta, y se hizo que cada una realizase un movimiento que se repite en ciclos de un segundo, que es lo que dura la animación.

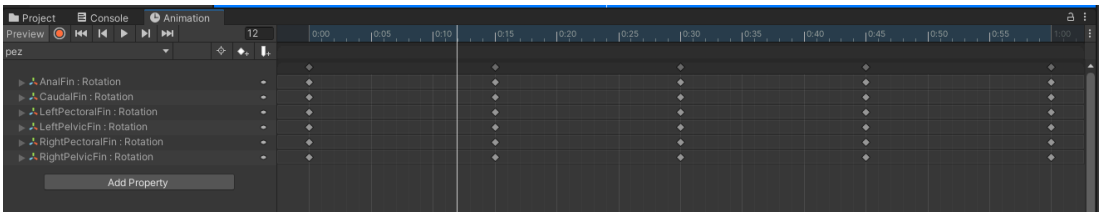

Figura 36. Animación del pez en Unity

4.3. Obstáculos y medusas

4.3.1. Obstáculos

Los obstáculos del juego se componen de las siguientes partes, que están marcadas en la Figura 37:

1. **Entrada del obstáculo**: es un objeto vacío que marca el punto que se considera la entrada del obstáculo. No influye en nada en el juego, sino que se colocó por si acaso, en un futuro, interesa registrar las coordenadas de la entrada de los obstáculos en el registro de datos visuales.
2. **Salida del obstáculo**: al igual que la entrada, la salida del obstáculo tampoco se usa, y solamente se incluye por si acaso es necesaria en un futuro para el registro de datos visuales.
3. **Meta**: es un GameObject que solo tiene un Collider y el script `MetaObstaculo`. Cuando el jugador colisiona con él, se comprueba si se ha chocado con una roca. En caso contrario, considera que ha superado el obstáculo.
4. **Barreras**: son las rocas que se deben evitar para superar el obstáculo.

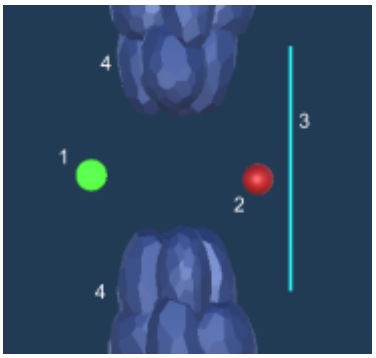

Figura 37. Partes de un obstáculo

El funcionamiento interno de los obstáculos se basa en tres scripts: `Obstaculo`, `BarreraObstaculo`, y `MetaObstaculo`. Estas clases se pueden ver en los Apéndices 12, 13 y 14, respectivamente.

En primer lugar, `Obstaculo` es el script asignado al GameObject padre que contiene todos los elementos de la Figura 37. Su funcionamiento se basa en el uso de un flag, `jugadorHaChocado`, para indicar si el jugador ha colisionado en algún momento con alguna de las rocas del obstáculo. Este flag se puede activar o desactivar con los métodos `Reiniciar` y `ObstaculoGolpeado`, respectivamente. Cuando el jugador alcanza la meta del obstáculo, se invoca el método `ObstaculoSuperado`, que comprueba si el flag está desactivado. Si es así, se considera que el obstáculo ha sido superado, por lo que se ejecuta el método `GestionarObstaculoPasado` de `TareaBuceo`.

Por otro lado, cada una de las rocas del obstáculo tiene un componente `BarreraObstaculo`. Con él, las barreras, que actúan como Triggers, pueden reaccionar a las colisiones con otros objetos. Si el jugador choca con una de ellas, esta invoca al método `ObstaculoGolpeado` del script `Obstaculo`, y a la función `RecibirGolpe` del componente `EstadoJugadorBuceo` del jugador, para registrar que ese obstáculo ya no puede ser superado, y que el pez debe perder una vida. Además, la clase implementa tanto `OnTriggerEnter` como `OnTriggerStay`, para que lo único que pueda evitar que el jugador choque constantemente con las rocas sea el periodo de invencibilidad que obtiene tras recibir daño o al romperse su escudo. Cabe destacar que, como las rocas son Triggers, el jugador las atraviesa. El motivo por el que esto se hizo así es que, si el jugador no pudiese pasar a través de las rocas, podría perder varias vidas si no es lo suficientemente habilidoso para entrar en el hueco antes de que pierda la invulnerabilidad.

Por último, las metas de los obstáculos utilizan el script `MetaObstaculo` para reconocer cuándo el jugador las atraviesa. Cuando esto ocurre, invocan al método `ObstaculoSuperado` del script `Obstaculo` para comprobar si el jugador ha chocado con una barrera. Si no es así, se considera que se ha superado de verdad.

4.3.2. Medusas

Las medusas que aparecen en los niveles más avanzados hacen que el jugador pierda una vida si choca con ellas. No cuentan como obstáculos que el jugador deba superar, si ese es el objetivo del nivel, ni le otorgan puntos por evitarlas. Por este motivo, su script `Medusa` se limita a invocar al método `RecibirGolpe` de `EstadoJugadorBuceo` si el pez choca con una medusa. Sin embargo, las medusas no están formadas por una jerarquía de varios GameObjects. El script `Medusa` se puede ver en el Apéndice 15.

Por otro lado, las medusas se mueven constantemente hacia la dirección marcada por su propiedad `direccionMovimiento`, que comienza siendo hacia arriba. Este valor se cambia en el método `Update` de `Medusa` para que se mueva constantemente entre las posiciones Y = -1 e Y = 4, invirtiendo el sentido de su movimiento cuando llega a uno de esos dos extremos. Después de comprobar si es necesario cambiar el sentido del movimiento, la medusa se desplaza.

Además, al igual que el pez, las medusas también están animadas y fueron modeladas en Blender. En su caso, no tienen varios componentes animados por separado, sino que la animación hace que la medusa rote sobre sí misma y cambie su escala de forma cíclica. El modelo de la medusa y la línea temporal de su animación se muestran en la Figura 38.

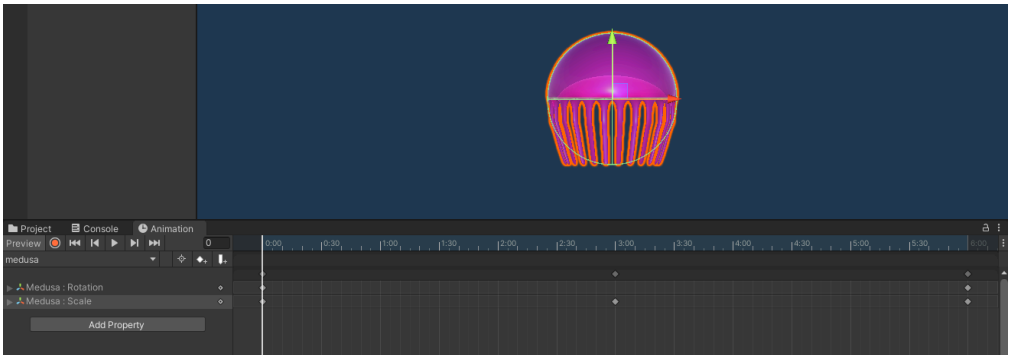

Figura 38. Animación y modelo de las medusas

4.4. Objetos

Al igual que los obstáculos, los objetos del juego están formados por varios GameObjects. La Figura 39 muestra cómo están organizados los que solamente dan puntos, pero esta estructura se repite en todos los objetos.

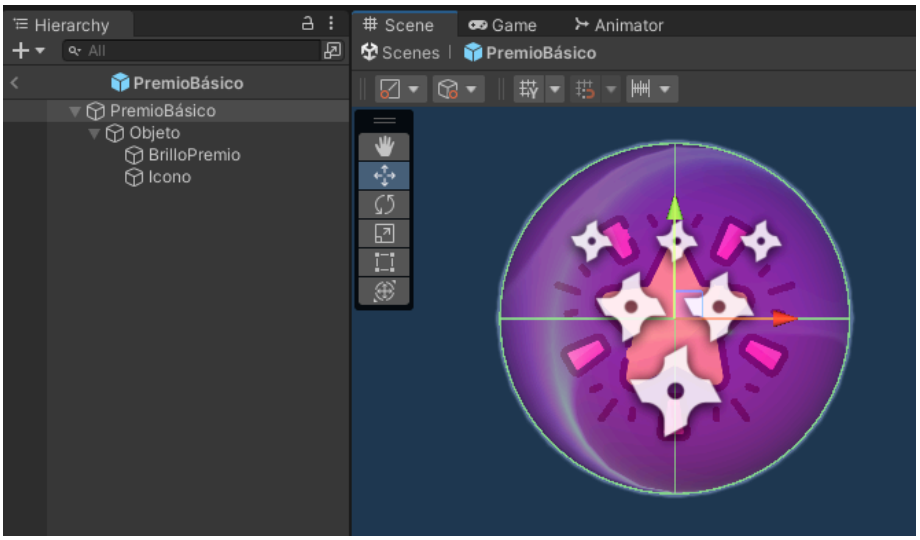

Figura 39. Estructura de los premios de puntuación

Así, los GameObject que forman el objeto son:

- **Raíz**: este objeto, cuyo nombre corresponde al nombre del objeto concreto, es un GameObject vacío que actúa como raíz del objeto. En la Figura 39, este objeto es PremioBásico.
- **Objeto**: es el GameObject que contiene la malla y el SphereCollider del objeto, además del script que implementa su funcionamiento. El motivo por el que todos estos componentes no están en el objeto raíz es que los objetos tienen una animación que hace que se muevan cíclicamente de arriba a abajo, al igual que las medusas. Como esto modifica su posición, se añadieron el Animator y el resto de componentes a un objeto hijo de la raíz para que la posición en el mundo del objeto no dependiera de su animación.
- **BrilloPremio**: es un sistema de partículas que emite destellos constantemente. Su color es el mismo que el del premio.
- **Icono**: es un plano con un material creado a partir de una textura con transparencias. Permite identificar el efecto del premio. Las imágenes fueron obtenidas de la página web Flaticon [24].

En cuanto a los scripts que utilizan los objetos, se ha implementado la jerarquía de clases que aparece en la Figura 40.

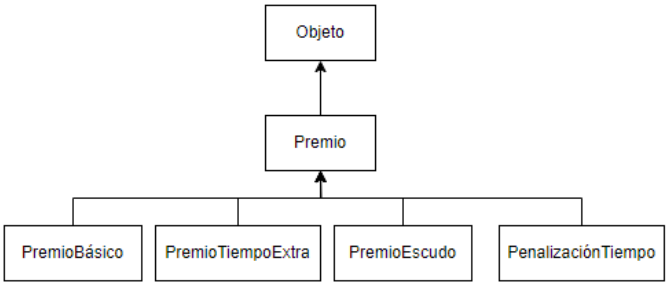

Figura 40. Diagrama de clases de los scripts de los objetos

En primer lugar, la clase `Objeto` existe sólo para obtener referencias a los scripts `TareaBuceo` y `GestionNivel` al principio del juego; pero no tiene otras funciones. Esta clase se muestra en el Apéndice 16.

En segundo lugar, la clase `Premio` introduce una referencia al GameObject raíz del objeto y un getter para obtenerlo. También almacena el número de puntos obtenidos al recoger el premio, y un método abstracto, `AplicarEfecto`, que se debe ejecutar cuando el jugador colisiona con el objeto. Además, declara el método `OnTriggerEnter` que utilizarán todos los objetos que hereden de `Premio`. Este comprueba si el GameObject que ha chocado con el objeto es el jugador y, si es así, se invoca la función `AplicarEfecto`, y luego se ejecuta el método `EliminarPremio` del script `GestionNivel` para devolver el premio al pool. Es importante mencionar que cada tipo de premio tendrá un prefab asociado, por lo que, como las propiedades `objetoRaizPrefab` y `puntos` aparecen en el Editor, se les asignan sus valores definitivos desde ahí. La clase `Premio` se puede ver en el Apéndice 17.

La clase `PremioBasico`, disponible en el Apéndice 18, es la que utiliza el premio que solo otorga 100 puntos. Por ello, su implementación de `AplicarEfecto` solo debe invocar los métodos `AgregarPuntuacion` y `GestionarPremioRecogido` de `TareaBuceo` para que se añadan los puntos y se realicen las acciones adicionales que haga falta.

Por otro lado, la clase `PremioEscudo` es prácticamente igual a `PremioBasico`, pues la única diferencia entre ellas es que `PremioEscudo` también ejecuta la función `ActivarEscudo` del script `EstadoJugadorBuceo` para activar el escudo del pez. Esta clase se muestra en el Apéndice 19.

La clase `PremioTiempoExtra`, visible en el Apéndice 20, implementa su propio `Start` para obtener una referencia al script `GestionTiempo` al crearse. Además, almacena una variable que indica cuántos segundos extra se obtienen al recoger

un premio de este tipo, que, en este caso, son cinco segundos. Así, `AplicarEfecto` invoca al método `AñadirTiempo` de `GestionTiempo`, además de agregar la puntuación usando el método `AgregarPuntuacion` de `TareaBuceo`.

Por último, aunque los objetos que restan tiempo no se pueden considerar premios, la clase `PenalizacionTiempo` hereda de `Premio`. Esto se debe a que las penalizaciones solo se diferencian de los premios en la implementación de `AplicarEfecto`. Así, `AplicarEfecto` comprueba si el jugador tiene el escudo activo y, si es así, lo rompe y este evita que el jugador sufra los efectos del objeto. Si el pez no está protegido, se invoca a `AgregarPuntuacion` y `AñadirTiempo` con valores negativos para que se le resten puntos y segundos. Concretamente, al recoger estas penalizaciones, el jugador pierde 100 puntos y 5 segundos. Además, se reproduce el mismo efecto de sonido que se usa cuando se choca con un obstáculo; pero no pierde ninguna vida. Esta clase se puede ver en el Apéndice 21.

4.5. Decoración del escenario

Los diferentes modelos que forman el escenario del juego –es decir, el suelo, el techo, las rocas de los obstáculos y los corales del fondo– fueron modelados con Blender. Por un lado, los corales fueron modelados utilizando el plugin de Blender "Add Curve: Sapling Gen Tree". El motivo por el que se usó este add-on, a pesar de que está pensado para crear árboles y personalizar la distribución de sus ramas y hojas, es que se quiso dar una apariencia más realista a la ramificación de los corales. Por otro lado, el resto de elementos decorativos del escenario se modelaron con las herramientas que ya están presentes en la versión base de Blender. Los corales modelados son los que aparecen en la Figura 41.

Figura 41. Corales del escenario

En cuanto al skybox, fue creado a partir de tres texturas hechas en Krita. Como Unity permite definir un material para skybox asignando una textura a cada una de las caras de su cubo, se crearon dos texturas azules, una cian y otra azul marino, para las caras superior e inferior del cubo, respectivamente. Para el resto de caras se usó un degradado. Como la cámara del juego es ortográfica y su ángulo pitch es cero, no se pueden ver en el juego las partes superior e inferior del

cubo que forma el skybox; pero sí se pueden ver en el editor, como muestra la Figura 42.

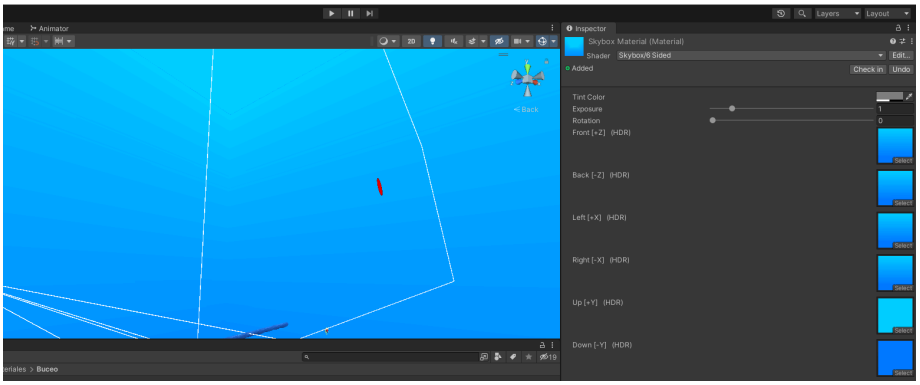

Figura 42. Material creado para el skybox del juego

4.6. Registros

Al igual que en el resto de juegos del proyecto NRET, cada vez que el jugador gana o pierde en una fase de "Buceo" se guarda en un fichero .txt información sobre su rendimiento y el punto de la pantalla al que estaba mirando en diferentes instantes. Esto último se conoce como registro de posición ocular. Los datos que se añaden sobre el juego desarrollado son, para cada instante de tiempo:

- Tiempo desde que comenzó el intento
- Puntuación actual (acumulada con los niveles superados)
- Vidas
- Coordenadas X e Y de la mirada, en el espacio original de `GazePoint`
- Premios obtenidos hasta el momento en el nivel
- Premios que se deben recoger para superar el nivel (-1 si no es el objetivo)
- Obstáculos esquivados hasta el momento en el nivel
- Obstáculos a evitar para superar el nivel (-1 si no es el objetivo)
- Distancia a la meta desde la posición X = 0
- Tiempo restante en segundos (-1 si el nivel no es contrarreloj)

Cada cierto tiempo se invoca el método virtual `NuevoRegistro`, que es definido en la clase `Tarea`, madre de `TareaBuceo`. Este método, cuyo código se encuentra en el Apéndice 22, permite crear una nueva entrada a partir de los datos anteriores y añadirla a la lista con la que se creará el archivo final. Como `Tarea` ya estaba presente en el proyecto, para posibilitar la creación de registros del juego "Buceo" se ha redefinido el método `NuevoRegistro` en `TareaBuceo`. Su función es obtener

los datos mencionados anteriormente y devolver una instancia de la clase `RegistroPosicionOcularTareaBuceo`.

`RegistroPosicionOcularTareaBuceo` es una clase hija de `RegistroPosicionOcular`, que ya estaba presente antes de la creación del juego de este Trabajo Fin de Máster. `RegistroPosicionOcular` define las propiedades que son comunes a los registros de todos los juegos, como las coordenadas de la mirada y el instante de tiempo. Esta clase es donde se creó inicialmente el método virtual `RegistroFormateadoParaEscribirEnDisco`, que devuelve una string con la información formateada para añadirla directamente al txt como fila. Así, `RegistroPosicionOcularTareaBuceo` sobreescribe tanto el constructor como dicho método para incluir los parámetros necesarios. Esta clase se muestra en el Apéndice 23.

La Figura 43 muestra la cabecera de un registro. Esta incluye, además de la fecha, el código del paciente, y el nivel actual, todos los datos recabados por la clase `RegistroPosicionOcularTareaBuceo`.

```
codigo del paciente: 002
fecha de registro: martes, 18 junio 2024 19-25-56
Tarea de buceo
Nivel: 0
Leyenda: Tiempo desde que comenzó el intento; Puntuación; Vidas; Posición X de la mirada; Posición Y de la mirada;
Posición X del jugador; Posición Y del jugador; Premios obtenidos; Premios por recoger; Obstáculos esquivados;
Obstáculos por esquivar; Distancia a la meta; Tiempo restante
```

Figura 43. Cabecera de un registro

Concretamente, este registro corresponde al primer nivel, que es el tutorial. En este nivel hay que recoger 10 premios, por lo que la columna “Premios por recoger” tiene ese valor en todas las filas. En cambio, las columnas “Obstáculos por esquivar” y “Tiempo restante” siempre valen -1, pues no son los objetivos del nivel. En la Figura 44 se ven algunas entradas del mismo registro, pero en un momento posterior. Por ejemplo, es posible que el paciente tuviera dificultades con el control del juego, pues después de 53 segundos, solo había conseguido 3 premios.

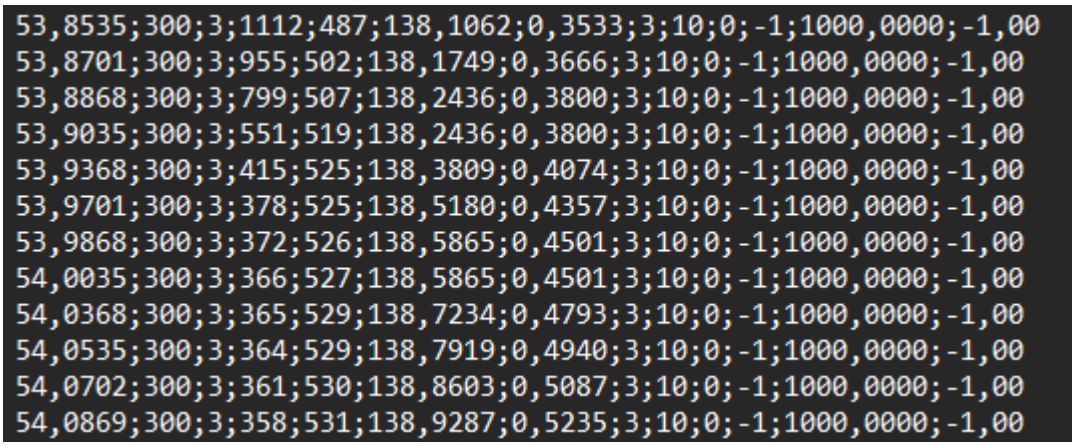

Figura 44. Entradas de un registro

Capítulo 5
Pruebas realizadas. Estudio de usabilidad

El objetivo de este capítulo es exponer las pruebas que se han realizado con personas sanas y pacientes, y los resultados obtenidos en el estudio de usabilidad.

5.1. Pruebas realizadas

El proceso de las pruebas consta de dos fases. En la primera, el usuario debe jugar al juego desde el principio. En la segunda, debe rellenar un cuestionario de usabilidad. Las preguntas del cuestionario se dividen en varias secciones:

1. **Información sobre el usuario**: se centran en obtener datos sobre la edad y educación de la persona que está probando el juego, así como el uso que le da a la tecnología. Las preguntas de esta sección son:
    - **Edad**: cada una de las respuestas disponibles corresponde a un rango de edad: [18, 25], [26, 35], [36, 45]. [46, 55], [56, 65], [66, 75] y 76 o más.
    - **Nivel de estudios**: ESO, Bachillerato, Grado, o Máster y/o Doctorado
    - **¿Usa el ordenador?**: sí o no.
    - **Si ha contestado sí a la anterior, ¿con qué frecuencia?**: a diario, varias veces a la semana, una vez a la semana, algunas veces al mes, o algunas veces al año.
    - **Si ha respondido a la anterior, ¿qué uso le da?**: se puede elegir varias respuestas de entre trabajo, redes sociales, y jugar.
    - **¿Ha jugado a videojuegos?**: sí o no.
    - **Si ha contestado sí a la anterior, ¿con qué frecuencia?**: a diario, varias veces a la semana, una vez a la semana, algunas veces al mes, o algunas veces al año.
2. **Experiencia de juego**: se trata de una serie de preguntas que el jugador debe contestar eligiendo, para cada una, si está de acuerdo o no con la afirmación que se presenta en la pregunta. Para ello, las respuestas disponibles utilizan la escala Likert [25], que permite al usuario definir lo de acuerdo o en desacuerdo que está con una afirmación presentando las respuestas posibles en un rango en el que “De acuerdo” y “En desacuerdo” están en extremos opuestos, y “Ni uno ni otro” está en medio. Las preguntas de esta sección son:
    - Calibro el sistema de control ocular al inicio del juego sin dificultad.
    - Controlo el pez mediante el control ocular de forma intuitiva y fácil.
    - Los niveles que he jugado son desafiantes.
    - Puedo identificar claramente hacia dónde dirijo la mirada.

- La velocidad de respuesta del control ocular es adecuada para jugar.
- La velocidad del pez al moverse con el eye tracker es adecuada.
- El control ocular es cómodo y no causa malestar durante las sesiones de juego.
- Entiendo las instrucciones del juego fácilmente.
- En la pantalla aparece toda la información necesaria para jugar.
- El feedback audiovisual del juego mientras utilizaba el eye tracker es útil.
- La mayoría de la gente aprendería a usar este juego en forma muy rápida.
- Este juego controlado por los ojos ofrece mecánicas que resultan emocionantes y atractivas.
- Este juego me ha aportado diversión y entretenimiento.

3. **Ideas y sugerencias sobre cómo mejorar la experiencia de juego**: está formada por dos preguntas de respuesta corta en las que el sujeto puede exponer sus ideas de mejora para el juego. Estas preguntas son:
    - ¿Hay características adicionales que te gustaría ver implementadas en el juego?
    - ¿Qué aspectos específicos te gustaría que se mejoraran en futuras actualizaciones?

5.2. Resultados obtenidos

El juego fue probado por cinco pacientes y nueve personas sanas. De ellos, tres pacientes y ocho personas sanas rellenaron la encuesta, lo que da un total de once respuestas. A partir de ellas, se puede inferir información acerca de los jugadores y su opinión acerca del juego.

Para empezar, de la primera sección, la información más importante es la relación del sujeto con los videojuegos, ya que seguramente aquellos que sean jugadores frecuentes puedan aprender a jugar con más facilidad. En las respuestas de las preguntas relacionadas con esto, visibles en la Figura 45, se observa que la gran mayoría ha jugado a videojuegos alguna vez, pero solo un 30% juega más de una vez a la semana.

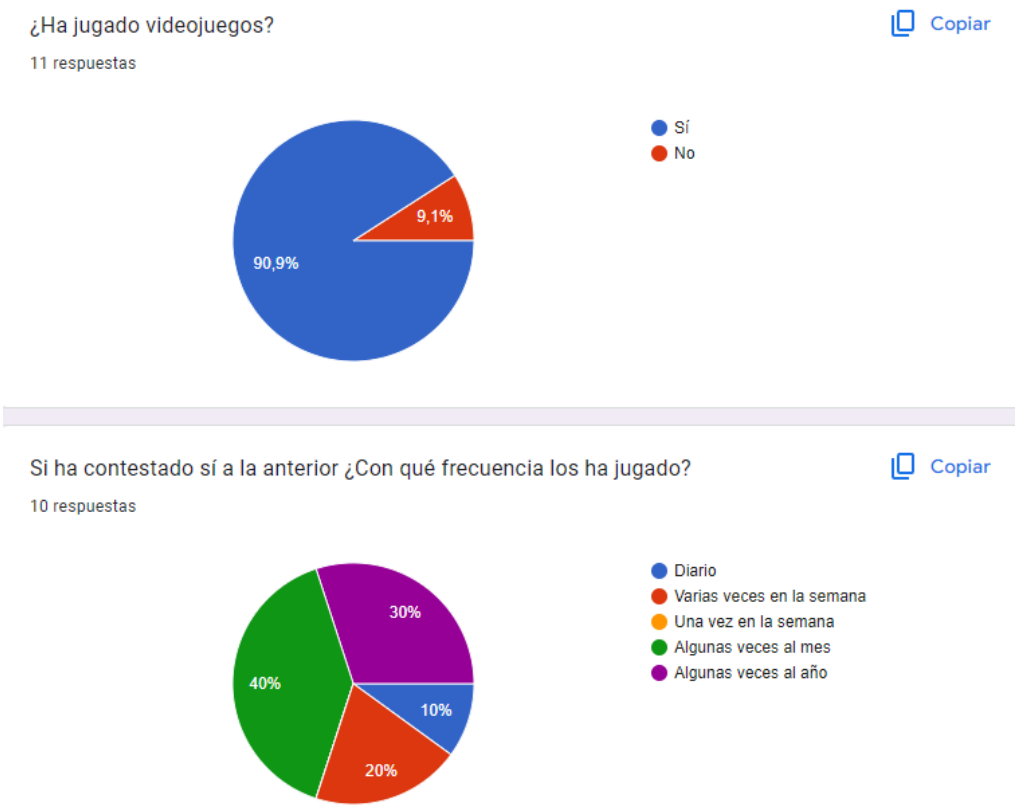

Figura 45. Respuestas a las preguntas sobre si el sujeto juega a videojuegos

Si se analizan las respuestas a las preguntas sobre el control del pez en la siguiente sección, se observa que las opiniones de los sujetos están mucho más divididas con respecto a este aspecto. Por ejemplo, las Figuras 46 y 47 muestran que, en general, los sujetos opinan que el control del pez es intuitivo; pero muchos tienen problemas para identificar hacia dónde dirigen la mirada. Actualmente, el punto de mira del jugador se marca en la pantalla con un punto rojo; pero probablemente se pueda mejorar esa parte de la interfaz para que sea más fácil de reconocer.

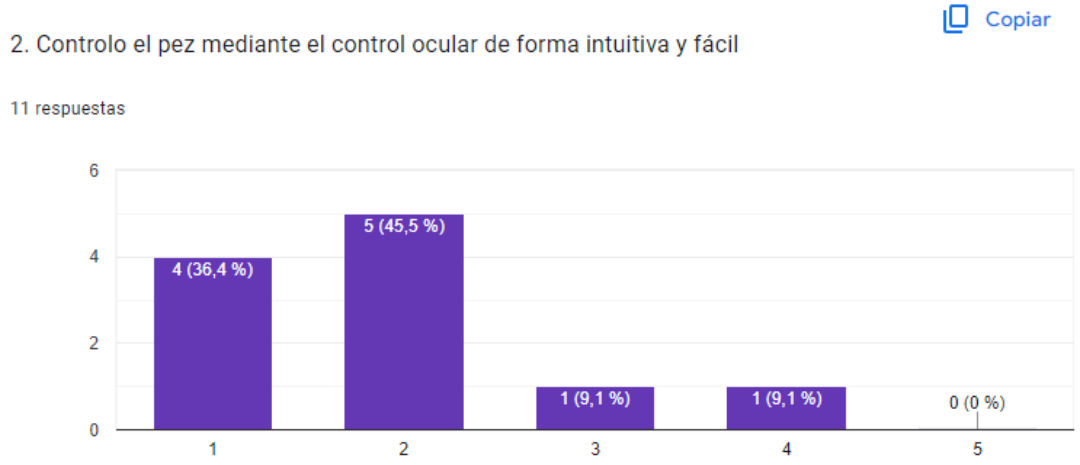

Figura 46. Resultados de la pregunta sobre la sencillez del control

Figura 47. Resultados de la pregunta sobre la facilidad para identificar el punto de mira

En cambio, como muestra la gráfica de la Figura 48, los sujetos creen que la velocidad del pez es apropiada, aunque hay personas que no están de acuerdo con esta afirmación. Esto podría significar que hay personas que consideran que el pez va demasiado rápido o lento, o que, en ocasiones, la velocidad del pez no concuerda con el punto de mira. Si se analizan las respuestas a las preguntas de respuesta libre, se encuentran propuestas como hacer que la velocidad horizontal

del pez dependa de la posición horizontal de la mirada, de modo que el pez fuera más rápido si el jugador mira hacia la derecha.

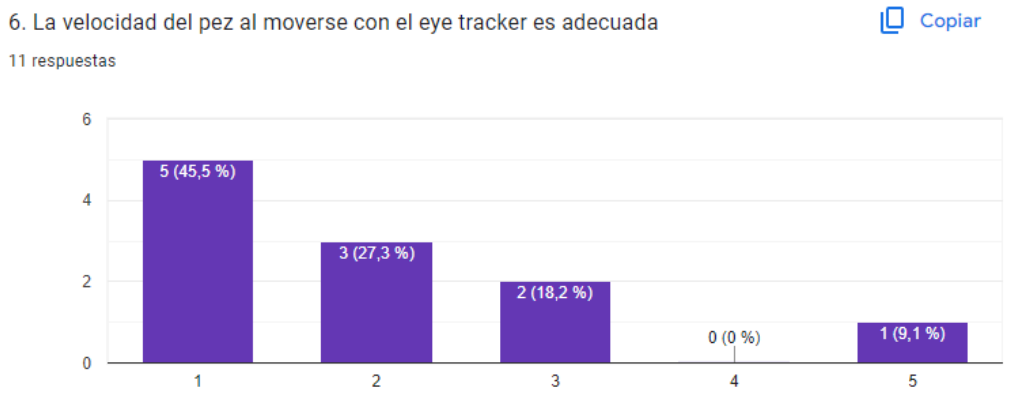

Figura 48. Resultados de la pregunta sobre la velocidad del pez

Si se analizan las respuestas a las preguntas sobre la diversión aportada por el juego y su dificultad, se aprecia que las opiniones también están bastante divididas. Por un lado, como muestra la Figura 49, la gran mayoría de jugadores piensa que el juego es entretenido. Además, también se puede ver en la Figura 50 que, por lo general, las mecánicas del juego son interesantes para los jugadores; pero no todos están completamente de acuerdo en esto. Por ello, es probable que, si se añadiesen más tipos de niveles o mecánicas al juego, se debería dar prioridad a intentar que fueran más interesantes.

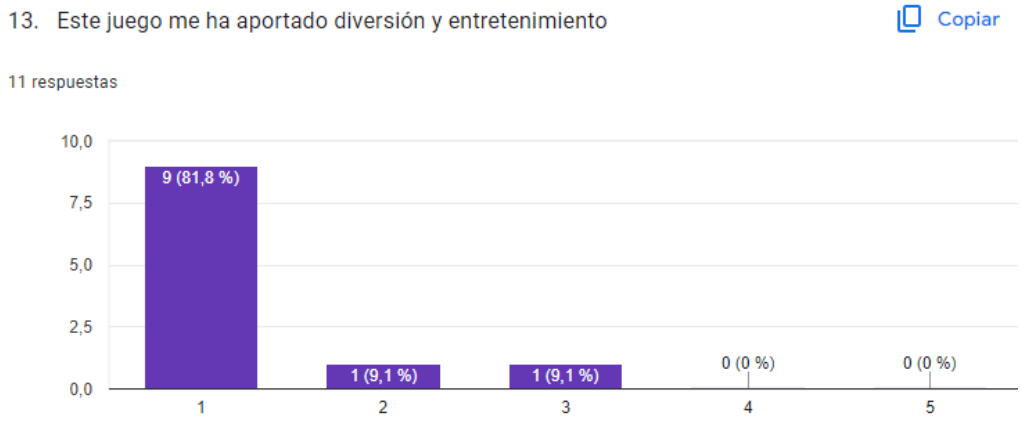

Figura 49. Resultados de la pregunta sobre la diversión aportada por el juego

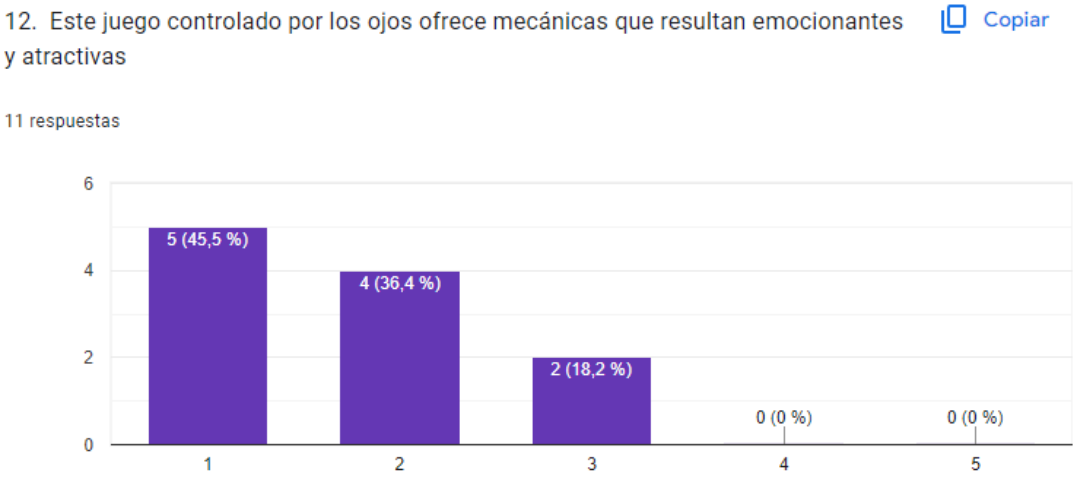

Figura 50. Resultados de la pregunta sobre el interés generado por las mecánicas

Para finalizar con esta sección, también cabe destacar que, como muestra la Figura 51, la mayoría de los sujetos piensan que el juego es desafiante; pero también hay un porcentaje importante que opina que no lo es, o incluso que es fácil. Al igual que con la diversión aportada por el juego, si se hicieran más niveles o se introdujeran nuevas mecánicas, probablemente sea recomendable añadir niveles más complicados.

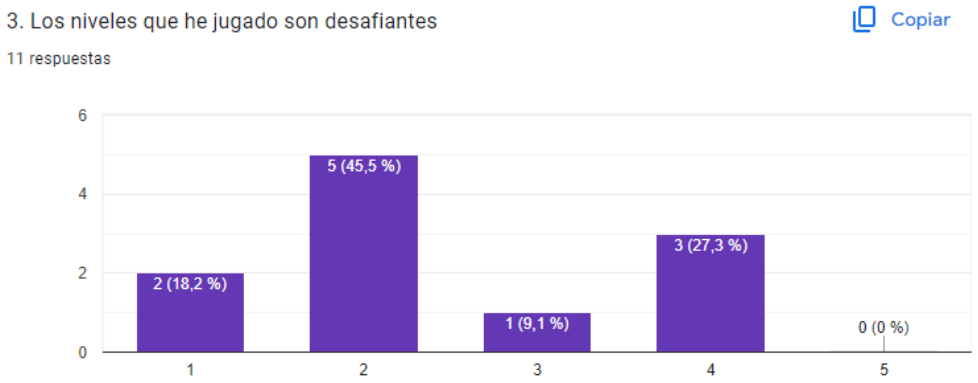

Figura 51. Resultados de la pregunta sobre la dificultad del juego

Finalmente, en cuanto a las preguntas de respuesta libre, se pueden extraer algunas conclusiones generales sobre la opinión de los sujetos:

- El juego debería ofrecer más opciones de accesibilidad, por ejemplo, permitir cambiar los colores utilizados en el juego para mejorar la experiencia de personas con discromatopsia. También se podría permitir cambiar el tamaño del puntero, para que fuera más fácil verlo.
- Como se vio anteriormente, las respuestas a las preguntas libres apuntan a que los jugadores querrían que el juego tuviera más variedad de obstáculos y actividades más complejas, entre otros elementos.
- Dado que el pez se controla con la mirada, podría ser apropiado hacer que la información del número de vidas y el progreso en el nivel se moviera con el cursor.

Capítulo 6
Presupuesto

Para finalizar, en este capítulo se estima un presupuesto necesario para cubrir los gastos del desarrollo del presente TFM.

Tabla de materiales

Recurso	**Coste del recurso (sin IGIC)**	**Precio total (con 7% IGIC)**
PC	1.700 €	1.819 €
Eye tracker Tobii 4C	280 €	299,60 €
Unity 2021 (licencia Personal)	0 €	0 €
Blender 4.0	0 €	0 €
Krita 5.2.2	0 €	0 €
Visual Studio Code 1.90.2	0 €	0 €
Plastic SCM	0 €	0 €

Tabla de recursos humanos

Recurso	**Coste del recurso (sin IGIC)**	**Precio total (con 7% IGIC)**
Horas de trabajo (300 h)	40 €/ h	12.840 €

Al sumar todos los importes calculados, se obtiene que el presupuesto necesario es **14.958,60 €**.

Conclusiones y Líneas futuras

Conclusiones

El objetivo de este TFM ha sido desarrollar un juego completamente jugable controlado por un eye tracker que se pueda utilizar para rehabilitar a pacientes reales, así como realizar pruebas con pacientes y personas sanas. Por tanto, dado que se ha creado exitosamente el juego “Buceo” dentro del proyecto de Unity utilizado por el HUC, y también se han realizado las pruebas esperadas, se puede afirmar que se ha cumplido este objetivo.

No obstante, si bien el proyecto se ha completado con éxito, durante el desarrollo del juego se presentaron algunos problemas menores relacionados con la implementación del juego. Muchos de ellos estaban relacionados con bugs existentes en la generación de los niveles, de modo que fue necesario modificar varias veces el algoritmo y los parámetros para obtener un resultado satisfactorio. Estos problemas retrasaron el comienzo de las pruebas con los pacientes, pero igualmente pudieron llevarse a cabo sin problemas.

También cabe mencionar que, originalmente, se usaba un solo pool para almacenar tanto los premios como los objetos perjudiciales. Además, cada vez que se generaba una sección, se elegía un objeto de ese pool; pero también era posible que no se extrajese ninguno. Por todo esto, cuando se alcanzaban los niveles que introducen los objetos perjudiciales, la probabilidad de que apareciera un premio disminuía, lo cual resultaba frustrante en los niveles de recoger objetos. Este fue uno de los principales inconvenientes detectados por el equipo del HUC cuando comenzaron las pruebas con pacientes. Por ello, se hizo posible que se pudiera definir un valor de probabilidad para cada tipo de objeto.

La etapa más complicada fue el comienzo del proyecto, pues implicó estudiar partes del código de los otros juegos desarrollados para saber cómo conectar el nuevo juego con el sistema ya existente. Por ejemplo, fue necesario modificar la implementación del menú principal para añadir los botones que permiten acceder al nuevo juego y los registros. Por otro lado, el código que escribe los registros en ficheros .txt ya estaba hecho, así que lo que se añadió fue la clase `RegistroPosicionOcularTareaBuceo` para que recabase los datos necesarios sobre el juego nuevo. El resto de partes de la implementación no resultaron complicadas, a excepción de la generación del nivel, como se mencionó anteriormente.

Por otro lado, el modelado 3D de los recursos del juego fue una de las partes más complicadas, debido a la falta de experiencia en ello. Por este motivo, las animaciones del juego también son sencillas.

Por último, comentar que la experiencia adquirida gracias a este TFM ha sido muy positiva, pues me ha permitido colaborar con un equipo que trabaja para ayudar a los demás como el del HUC. Además, los resultados de la encuesta indican que las personas que probaron el juego lo encontraron entretenido, lo que también es importante porque, al final, un videojuego que se use para rehabilitación no tiene por qué no ser divertido. Aparte de esto, los resultados de la encuesta apuntan a que, en general, el juego gustó; pero hay mucho margen de mejora en apartados como la accesibilidad y el control.

Líneas futuras

Con respecto a las posibles líneas de trabajo futuras, a pesar de que el juego ya es perfectamente jugable y está completo, cabe la posibilidad de ampliarlo con más niveles, obstáculos y objetos. Hacer esto no debería ser difícil, ya que el juego está construido de forma que se pueden introducir nuevos elementos con facilidad. Sin embargo, también sería interesante modificar el algoritmo de generación de niveles para introducir nuevos tipos de secciones. Por ejemplo, se podrían añadir secciones que no sean horizontales, sino diagonales o incluso verticales, pues esto permitiría crear niveles muy distintos de los que se han desarrollado hasta ahora. También se podría adaptar el juego para que se pueda jugar con otros tipos de dispositivos, como realidad virtual.

Otra posible línea de trabajo futura es el desarrollo de juegos adicionales para NRET con el conocimiento adquirido gracias a la creación de este. Si bien no se utilizaron muchas funciones de la API de Tobii, para un futuro juego se podría investigar si esta posee funcionalidades que puedan resultar interesantes, como la detección de los parpadeos y movimientos de la cabeza del jugador. Sin embargo, siempre hay que tener en cuenta que el público objetivo de estos juegos son pacientes con daño cerebral, por lo que lo ideal sería desarrollar un videojuego accesible para la mayor cantidad de pacientes posible.

Conclusions and future work

Conclusions

The main goal of this master's final project was to develop a game that could be played using an eye tracker device so that it could be used to rehabilitate real patients, as well as testing the game with both patients and healthy people. Since the "Diving" game has been created successfully as part of the Unity project used by the HUC, and the game has been tested as expected, too, the project's goal has been certainly achieved.

However, while the project has been completed successfully, some minor problems appeared during the development of the game that were related to its implementation. Many of them were related to bugs in the level generation algorithm, so said procedure and its parameters had to be modified multiple times in order to achieve a good result. These problems delayed the start of the tests with the patients but, fortunately, they were still carried out without adversities.

It should also be noted that both prizes and harmful items used to be stored in the same pool. Besides, each time the algorithm generated a section, it could pick a random item from that pool, but it could also not do so. For these reasons, the probability of prizes appearing diminished upon reaching the levels where harmful items begin appearing, which caused frustration in levels whose goal was to pick prizes. This was one of the main complaints that the HUC reported when the testing phase started. Because of this, the game was modified to allow setting the probability of each type of item.

The project's most difficult phase was its beginning, since I had to study parts of the games that were already developed in order to understand how to connect the new game with the rest of the program. For example, I modified the game's main menu to add the buttons that allow users to access the new game. Furthermore, while the code that writes the registers as .txt files was already developed, I had to add the `RegistroPosicionOcularTareaBuceo` class, which collects the data of the new game needed for the registers. The rest of the implementation was not difficult, except for the algorithm that generates the level, as mentioned before.

On the other hand, modeling the game's 3D assets was one of the most difficult tasks of the project due to the lack of experience in that field. For this reason, the game's animations are also simple.

Lastly, the experience that I have acquired thanks to this master's thesis is quite positive, since this project has allowed me to cooperate with the HUC team, which works to help others. Besides, the survey's results indicate that people who played the game found it entertaining, which is also important, because even video games that are not created explicitly to entertain should also be fun. The survey also indicates that, even though people had positive opinions about the game, there is still room for improvement in areas like accessibility and the game's controls.

Future work

As for possible future plans for the project, even though the game is already playable and finished, it could be expanded with more levels, obstacles and items. Doing this should not be difficult, since the game has been built in a way that could ease the introduction of new elements. However, another interesting possibility would be modifying the algorithm that generates the levels to add new types of sections. For example, adding vertical and diagonal sections would be interesting, because this would allow the creation of levels that are very different to the ones developed until now. The game could also be adapted so that it can be played using other devices, like a virtual reality headset.

Another future plan could be to develop more games for the NRET project using the knowledge acquired while making this one. While the Tobii API was not used too much in this game, if I were to develop another one in the future, it would be interesting to research if said API has functionalities that could be used to create more complex interactions, like blink detecting and head movement tracking. However, taking into account that these games' main target are patients with brain damage is paramount, so any new games should be developed in a way that can be played by as many people as possible.

Bibliografía

Enlaces a los vídeos de Youtube:

Niveles 1-3: https://youtu.be/45ujmo3orCY

Nivel 9: https://youtu.be/hKA27U6fE8Q

[1] García-Ramos, B. R., Villarroel, R., González-Mora, J. L., Revert, C., & Modroño, C. (2023). Neurofunctional correlates of a neurorehabilitation system based on eye movements in chronic stroke impairment levels: A pilot study. Brain and Behavior. Recuperado de https://doi.org/10.1002/brb3.3049

[2] Modroño C, Plata-Bello J, Zelaya F, García S, Galván I, Marcano F, et al. (2015) Enhancing Sensorimotor Activity by Controlling Virtual Objects with Gaze. PLoS ONE 10(3): e0121562. https://doi.org/10.1371/journal.pone.0121562

[3] Emerging Technologies in Rehabilitation for Complex Injuries and Conditions. (2023, Septiembre 11). Physiopedia, . Recuperado el 29-06-2024 de https://www.physio-pedia.com/index.php?title=Emerging_Technologies_in_Rehabilitation_for_Complex_Injuries_and_Conditions&oldid=341321

[4] Pexels (s.f.) Recuperado el 02-07-2024 de https://www.pexels.com/es-es/

[5] Brainbit (s.f.) Recuperado el 29-06-2024 de https://brainbit.com/

[6] pxhere (s.f.) Recuperado el 02-07-2024 de https://pxhere.com/

[7] Exploring the Video Games-Physical Therapy Amalgamation: Benefits and Considerations (s.f.) Recuperado el 29-06-2024 de https://www.empoweremr.com/blog/video-games-physical-therapy

[8] Rao, V. (2012, Mayo 23) Circus Challenge: Video Games for Stroke Victims, Assistive Technology Blog de https://assistivetechnologyblog.com/2012/05/image-source-engadget-is-often.html

[9] Gustavsson, M., Kjörk, E. K., Erhardsson, M., & Alt Murphy, M. (2021). Virtual reality gaming in rehabilitation after stroke – user experiences and perceptions. Disability and Rehabilitation, 44(22), 6759–6765. Recuperado de https://doi.org/10.1080/09638288.2021.1972351

[10] Márquez, R. (2019, Mayo 5) Análisis de Beat Saber: una de las experiencias que justifican tener un casco de realidad virtual. Recuperado el 02-07-2024 de

https://www.vidaextra.com/analisis/analisis-beat-saber-experiencias-que-justifican-tener-casco-realidad-virtual

[11] Rodríguez Santana, M. A. (2016) Implementación de videojuegos para neurorrehabilitación a través de eye-tracking, Trabajo Fin de Grado, Universidad de La Laguna. Recuperado de https://riull.ull.es/xmlui/bitstream/handle/915/2621/Implementacion%20de%20video%20juegos%20para%20neurorrehabilitacion%20a%20traves%20de%20eye-tracking.pdf?sequence=1&isAllowed=y

[12] Martínez Galán, R. (2015) Desarrollo de videojuegos para la neurorrehabilitación mediante eye-tracking, Trabajo Fin de Grado, Universidad de La Laguna. Recuperado de https://riull.ull.es/xmlui/bitstream/handle/915/2623/Desarrollo%20de%20videojuegos%20para%20la%20neurorrehabilitacion%20mediante%20eye-tracker.pdf?sequence=1&isAllowed=y

[13] Tobii (s.f.). Recuperado el 29-06-2024 de https://www.tobii.com/

[14] Tobii Pro Glasses 3. Real Insights from the real world (s.f.) Recuperado el 29-06-2024 de https://www.tobii.com/products/eye-trackers/wearables/tobii-pro-glasses-3

[15] Tobii Horizon (s.f.) Recuperado el 29-06-2024 de https://gaming.tobii.com/horizon/

[16] Eye Tracking Software (s.f.) Recuperado el 29-06-2024 de https://gaming.tobii.com/getstarted/

[17] Tobii Help Center (2016) Recuperado el 06-07-2024 de https://help.tobii.com/hc/en-us/articles/213414285-Specifications-for-the-Tobii-Eye-Tracker-4C

[18] Unity (s.f.) Recuperado el 02-07-2024 de https://unity.com/es

[19] Berkebile, B. (s.f.) iTween. Recuperado el 29-06-2024 de https://www.pixelplacement.com/itween/index.php

[20] Visual Studio Code (s.f.) Recuperado el 05-07-2024 de https://code.visualstudio.com/

[21] Blender (s.f.) Recuperado el 05-07-2024 de https://www.blender.org/

[22] Krita (s.f.) Recuperado el 05-07-2024 de https://krita.org/es/

[23] Plastic SCM (s.f.) Recuperado el 05-07-2024 de https://www.plasticscm.com/

[24] Flaticon (s.f.) Recuperado el 29-06-2024 de https://www.flaticon.es/

[25] Bhandari, P. y Nikolopoulou, K. (2023, Junio 22). What is a Likert Scale? | Guide & Examples. Scribbr. Recuperado el 29-06-2024 de https://www.scribbr.com/methodology/likert-scale/

Apéndice A
Scripts

Apéndice A.1 Clase `NivelBuceoScriptable`

```
[CreateAssetMenu]
public class NivelBuceoScriptable : NivelScriptable
{
    [Header("Generación del nivel")]
    public TiposNivelesBuceo tipoNivelBuceo;
    public EspacioEntreObstáculosEnBuceo espacioEntreObstáculos;

    public float probabilidadAparición Premio;
    public float probabilidadAparición ElementoPerjudicial;
    public float probabilidadEspacioVacío;
    public float distanciaALaMeta;

    public List<GameObject> premios;
    public List<GameObject> elementosPerjudiciales;
    public List<GameObject> obstaculos;
    public List<GameObject> decoracion;

    [Header("Objetivos del nivel")]
    public int obstaculosAEsquivar;
    public int premiosARecoger;
    public float segundosDisponibles;

    [Header("Características del pez")]
    public float velocidadBase;
    public float velocidadReducida;
}
```

Apéndice A.2 Clase `TareaBuceo`

```
public class TareaBuceo : Tarea
{
    public GameObject jugador;
    public GameObject puntoInicio;
    public EstadoJugadorBuceo estadoJugadorBuceo;
```

```
private Coroutine corrutinaJuego;
private GestionNivel gestionNivel;
private GestionTiempo gestionTiempo;

private int obstaculosEsquivados;
private int premiosRecogidos;
private int numeroNiveles;

private GestionHUD gestionHUD;

public NivelBuceoScriptable Nivel {
    get { return (NivelBuceoScriptable) Configuracion.nivelActual;}
}

[SerializeField]
private int puntosPorObstaculo;

private void Update()
{
    gameObject.transform.position = new Vector3(
      jugador.transform.position.x,
      transform.position.y,
      transform.position.z);
}

public override void CrearRutasMensajesAudio()
{
    rutasMensajesAudio = new string[]
    {
        "Comenzamos",
        "Muy bien",
        "Has perdido",
        "Esquiva los obstáculos",
        "Recoge los premios",
        "Llega a la meta antes de que se acabe el tiempo",
        "Si tardas demasiado perderas vidas",
        "Cuidado ahora el pez va mas rapido",
        "Cuidado ahora se pondra mas dificil",
        "Cuidado con las medusas",
        "Evita los circulos rojos perderas tiempo",
        "Recoge los premios verdes para ganar tiempo",
        "Recoge premios amarillos para protegerte",
        "Recoge premios morados para obtener puntos",
        "Sube y baja el pez con la vista"
```

```
    };
}

public override string ObtenerNombreTarea()
{
    return "Tarea buceo";
}

protected override string ObtenerCabeceraTarea()
{
    string cabecera = string.Empty;
    cabecera += "Tarea de buceo\n";
    cabecera += "Nivel: " + Configuracion.nivelActual.numeroDelNivel +
                "\n";
    cabecera += "Leyenda: Tiempo desde que comenzó el intento;
                Puntuación; Vidas; Posición X de la mirada; " +
                "Posición Y de la mirada; Posición X del jugador;
                Posición Y del jugador; Premios obtenidos; Premios por
                recoger; Obstáculos esquivados; " +
                "Obstáculos por esquivar; Distancia a la meta; Tiempo
                restante";
    return cabecera;
}

protected override RegistroPosicionOcular NuevoRegistro(float tiempo,
                                                        int x, int y)
{
    // Vidas
    int vidas = estadoJugadorBuceo.Vidas;

    // Posición X del jugador
    float posicionXJugador = jugador.transform.position.x;

    // Posición Y del jugador
    float posicionYJugador = jugador.transform.position.y;

    // premios por recoger
    int premiosObjetivo = (Nivel.tipoNivelBuceo ==
                           TiposNivelesBuceo.RecogerPremios) ?
                           Nivel.premiosARecoger : -1;

    // obstáculos por esquivar
    int obstaculosObjetivo = (Nivel.tipoNivelBuceo ==
                              TiposNivelesBuceo.Esquivar) ?
```

```
                                Nivel.obstaculosAEsquivar : -1;

        // tiempo restante
        float segundosRestantes = (Nivel.tipoNivelBuceo ==
                                   TiposNivelesBuceo.Contrarreloj) ?
                                   gestionTiempo.SegundosRestantes : -1f;

        return new RegistroPosicionOcularTareaBuceo(
            tiempo, x, y, this.puntuacion, vidas, posicionXJugador,
            posicionYJugador, premiosRecogidos, premiosObjetivo,
            obstaculosEsquivados, obstaculosObjetivo,
            Nivel.distanciaALaMeta, segundosRestantes
        );
    }

    public override void AgregarPuntuacion(int cambioPuntuacion)
    {
        // no modificar la puntuacion si el juego ya ha terminado
        if (TareaBloqueada) {
            return;
        }

        this.puntuacion += cambioPuntuacion;
        if (this.puntuacion < 0) {
            this.puntuacion = 0;
        }
        Color colorTexto = (cambioPuntuacion >= 0) ? Color.green :
                           Color.red;
        gestionHUD.ActualizarPuntuacion(this.puntuacion, true, colorTexto);
    }

    protected override void Inicio()
    {
        numeroNiveles = 10;
        corrutinaJuego = StartCoroutine(CorrutinaPartida());
    }

    protected override bool TodosLosNivelesCompletados()
    {
        return
              Configuracion.pacienteActual.ultimoNivelDesbloqueadoTareaBuceo
              >= numeroNiveles;
    }
```

```
protected override IEnumerator TareaCompletada()
{
    base.TareaCompletada();
    yield return StartCoroutine(
        MostrarMensaje("Has completado todos los niveles del juego",
                       "¡Has completado todos los niveles del juego!",
                       0, null,
                       Mensaje.TipoMensaje.Record)
    );
}

protected override bool GuardarProgreso(bool partidaGanada)
{
    Debug.Log("Guardando progreso de partida de buceo");

    if (partidaGanada) {
        // guardar la puntuacion
        if (puntuacion > 0) {
            Configuracion.pacienteActual.puntuacionTareaBuceo +=
              puntuacion;
        }

        // progresar

       Configuracion.pacienteActual.ultimoNivelDesbloqueadoTareaBuceo++;

        if (TodosLosNivelesCompletados()) {
            Debug.Log("Todos los niveles de la tarea completos");
            // El juego se ha terminado, no hay mas niveles

       Configuracion.pacienteActual.ultimoNivelDesbloqueadoTareaBuceo =
          numeroNiveles;
        }

        // serializar los datos en disco
        Aplicacion.instancia.GuardarDatosPaciente(
          Configuracion.pacienteActual);

    } else {
        // no cambiamos el progreso ni guardamos datos
        Debug.Log("La partida se ha perdido, no se guarda el progreso");
    }

    // devolvemos falso porque no se conceden premios adicionales
```

```
        return false;
    }

    private IEnumerator CorrutinaPartida()
    {
        Debug.Log("Inicio de tarea");
        obstaculosEsquivados = 0;
        premiosRecogidos = 0;
        gestionNivel = FindObjectOfType<GestionNivel>();
        gestionTiempo = FindObjectOfType<GestionTiempo>();
        gestionHUD = FindObjectOfType<GestionHUD>();
        gestionHUD.ActualizarPuntuacion(this.puntuacion, false,
                                        Color.white);

        switch (Nivel.numeroDelNivel) {
            case 0:
                yield return
                  StartCoroutine(MostrarMensaje("Recoge los premios",
                                $"Recoge {Nivel.premiosARecoger} premios",
                                4, null, Mensaje.TipoMensaje.PremioBuceo));
                yield return
                  StartCoroutine(MostrarMensaje("Recoge premios morados para
                                obtener puntos", "Recoge premios morados
                                para obtener puntos", 4, null,
                                Mensaje.TipoMensaje.Estrella));
                break;
            case 1:
                yield return
                  StartCoroutine(MostrarMensaje("Esquiva los obstáculos",
                                $"Esquiva {Nivel.obstaculosAEsquivar}
                                obstáculos", 4, null,
                                Mensaje.TipoMensaje.ObstaculoBuceo));
                yield return
                  StartCoroutine(MostrarMensaje("Recoge premios amarillos
                                para protegerte", "Recoge premios amarillos
                                para protegerte", 4, null,
                                Mensaje.TipoMensaje.Escudo));
                break;
            case 2:
                yield return
                  StartCoroutine(MostrarMensaje("Llega a la meta antes de
                                que se acabe el tiempo", "Llega a la meta
                                antes de que se acabe el tiempo", 4, null,
                                Mensaje.TipoMensaje.Tiempo));
```

```
            yield return
              StartCoroutine(MostrarMensaje("Recoge los premios verdes
                             para ganar tiempo", "Recoge los premios
                             verdes para ganar tiempo", 4, null,
                             Mensaje.TipoMensaje.Tiempo));
            break;
        case 5:
            yield return
              StartCoroutine(MostrarMensaje("Evita los circulos rojos
                            perderas tiempo", "Evita los círculos
                            rojos, perderás tiempo", 4, null,
                            Mensaje.TipoMensaje.Calavera));
            break;
        case 6:
            yield return
              StartCoroutine(MostrarMensaje("Cuidado con las medusas",
                            "Cuidado con las medusas", 4, null,
                            Mensaje.TipoMensaje.Medusa));
            break;
        default:
            break;
    }
    yield return
      StartCoroutine(MostrarMensaje("Comenzamos", "Comenzamos", 4, null,
                    Mensaje.TipoMensaje.Comienzo));

    switch (Nivel.tipoNivelBuceo) {
        case TiposNivelesBuceo.Contrarreloj:
            gestionHUD.CargaHUD(Nivel.tipoNivelBuceo, 0);
            break;
        case TiposNivelesBuceo.Esquivar:
            gestionHUD.CargaHUD(Nivel.tipoNivelBuceo,
                                Nivel.obstaculosAEsquivar);
            break;
        case TiposNivelesBuceo.RecogerPremios:
            gestionHUD.CargaHUD(Nivel.tipoNivelBuceo,
                                Nivel.premiosARecoger);
            break;
        default:
            Debug.Log("Tipo de nivel desconocido");
            break;
    }

    if (Nivel.tipoNivelBuceo == TiposNivelesBuceo.Contrarreloj) {
```

```
            gestionTiempo.IniciarTemporizador(Nivel.segundosDisponibles);
        }
        DesbloquearTarea();
        gestionNivel.CargarNivel(Nivel);
        yield return new WaitForSeconds(0.1f);
    }

    public override void TiempoExcedido()
    {
        if (Nivel.tipoNivelBuceo == TiposNivelesBuceo.Contrarreloj) {
            gestionTiempo.PararTemporizador();
            jugador.GetComponent<Rigidbody>().isKinematic = true;
            JuegoPerdido();
        }
    }

    public void JugadorSinVidas()
    {
        gestionTiempo.PararTemporizador();
        jugador.GetComponent<Rigidbody>().isKinematic = true;
        JuegoPerdido();
    }

    protected override void JuegoGanado()
    {
        gestionTiempo.PararTemporizador();
        base.JuegoGanado();
    }

    public void GestionarFinalDelNivel()
    {
        if (Nivel.tipoNivelBuceo == TiposNivelesBuceo.Contrarreloj) {
            // terminar el juego aquí, pues el jugador ha ganado
            // (si se hubiera acabado el tiempo, el juego ya habría
            // terminado)
            JuegoGanado();
        } else {
            StartCoroutine(TransicionReinicioNivel());
        }
    }

    private IEnumerator TransicionReinicioNivel()
    {
        gestionHUD.OscurecerPantalla();
```

```
        yield return new WaitForSeconds(1f);
        jugador.transform.rotation = Quaternion.identity;
        jugador.transform.position = new Vector3(
          puntoInicio.transform.position.x,
          jugador.transform.position.y,
          jugador.transform.position.z);
        gestionNivel.SetJugadorAlPrincipioDelNivel(true);
        gestionHUD.AclararPantalla();
    }

    public void GestionarObstaculoPasado()
    {
        obstaculosEsquivados++;
        AgregarPuntuacion(puntosPorObstaculo);
        if (Nivel.tipoNivelBuceo == TiposNivelesBuceo.Esquivar) {
            FindObjectOfType<Audio>().FeedbackBarreraPasada();
            gestionHUD.ActualizarProgresoObjetivo(1);
            if (obstaculosEsquivados >= Nivel.obstaculosAEsquivar) {
                JuegoGanado();
            }
        } else {
            FindObjectOfType<Audio>().FeedbackPremioObtenido();
        }
    }

    public void GestionarPremioRecogido()
    {
        premiosRecogidos++;
        if (Nivel.tipoNivelBuceo == TiposNivelesBuceo.RecogerPremios) {
            FindObjectOfType<Audio>().FeedbackBarreraPasada();
            gestionHUD.ActualizarProgresoObjetivo(1);
            if (premiosRecogidos >= Nivel.premiosARecoger) {
                JuegoGanado();
            }
        } else {
            FindObjectOfType<Audio>().FeedbackPremioObtenido();
        }
    }
}
```

Apéndice A.3 Clase `Pool`

```
public class Pool : MonoBehaviour
{
    private List<GameObject> listaPrefabs;
    public List<GameObject> gameObjects;
    private int siguientePrefabAInstanciar;

    [SerializeField]
    private int cantidadInicialPorObjeto;

    public int TamañoPool {
        get { return gameObjects.Count; }
    }

    public void CrearPool(List<GameObject> objetos)
    {
        listaPrefabs = objetos;
        gameObjects = new List<GameObject>();
        siguientePrefabAInstanciar =
          UnityEngine.Random.Range(0, listaPrefabs.Count);

        foreach (GameObject prefab in objetos) {
            for (int i = 0; i < cantidadInicialPorObjeto; ++i) {
                GameObject nuevoObjeto =
                  Instantiate(prefab, gameObject.transform.position,
                              Quaternion.identity);
                nuevoObjeto.SetActive(false);
                gameObjects.Add(nuevoObjeto);
            }
        }
    }

    public GameObject SacarObjeto()
    {
        if (gameObjects.Count > 0) {
            int indice = UnityEngine.Random.Range(0, gameObjects.Count);
            GameObject objetoSacado = gameObjects[indice];
            gameObjects.RemoveAt(indice);
            return objetoSacado;
        } else {
            if (listaPrefabs.Count == 0) {
```

```
                return null;
            }
            GameObject nuevoObjeto =
              Instantiate(listaPrefabs[siguientePrefabAInstanciar],
                          gameObject.transform.position,
                          Quaternion.identity);
            nuevoObjeto.SetActive(false);
            siguientePrefabAInstanciar = UnityEngine.Random.Range(0,
              listaPrefabs.Count);
            return nuevoObjeto;
        }
    }

    public void AñadirObjeto(GameObject nuevoObjeto)
    {
        gameObjects.Add(nuevoObjeto);
    }
}
```

Apéndice A.4 Clase `GestionNivel`

```
public class GestionNivel : MonoBehaviour
{
    public GameObject limiteNivel;
    public GameObject meta;
    public GameObject cartel;
    public GameObject indicadorRepetirNivel;

    [SerializeField]
    private Pool poolDeObstaculos;

    [SerializeField]
    private Pool poolDePremios;

    [SerializeField]
    private Pool poolDeElementosPerjudiciales;

    [SerializeField]
    private Pool poolDeDecoracion;

    private TareaBuceo tareaBuceo;
```

```
private NivelBuceoScriptable nivel;
private float minEspacioEntreObstaculos;
private float maxEspacioEntreObstaculos;

private float espacioHastaElSiguienteObstaculo;
private float posicionXUltimoObstaculo;
private bool jugadorAlPrincipioDelNivel;

public List<Tuple<float, float>> rangos;

void Start()
{
    tareaBuceo = FindObjectOfType<TareaBuceo>();
    rangos = new List<Tuple<float, float>>();
}

void Update()
{
    if (tareaBuceo.TareaBloqueada) {
        return;
    }

    if (((gameObject.transform.position -
          limiteNivel.transform.position).magnitude <=
          maxEspacioEntreObstaculos + 5f) ||
        (gameObject.transform.position.x >
        limiteNivel.transform.position.x)) {
        return;
    }

    float distanciaHastaElUltimoObstaculo;
    if (jugadorAlPrincipioDelNivel) {
        distanciaHastaElUltimoObstaculo = maxEspacioEntreObstaculos;
    } else {
        distanciaHastaElUltimoObstaculo =
          Math.Abs(gameObject.transform.position.x -
          posicionXUltimoObstaculo);
    }
    jugadorAlPrincipioDelNivel = false;

    if (distanciaHastaElUltimoObstaculo >=
        espacioHastaElSiguienteObstaculo) {
        GameObject nuevoObstaculo = poolDeObstaculos.SacarObjeto();
        if (nuevoObstaculo != null) {
```

```
    nuevoObstaculo.transform.position =
      new Vector3(gameObject.transform.position.x,
                  nuevoObstaculo.transform.position.y,
                  nuevoObstaculo.transform.position.z);
    nuevoObstaculo.SetActive(true);
    Obstaculo componenteObstaculo =
      nuevoObstaculo.GetComponent<Obstaculo>();
    componenteObstaculo.Reiniciar();
    posicionXUltimoObstaculo =
      componenteObstaculo.Meta.transform.position.x;
    espacioHastaElSiguienteObstaculo =
      UnityEngine.Random.Range(minEspacioEntreObstaculos,
                               maxEspacioEntreObstaculos);
} else {
    posicionXUltimoObstaculo = gameObject.transform.position.x;
    espacioHastaElSiguienteObstaculo =
      UnityEngine.Random.Range(minEspacioEntreObstaculos,
      maxEspacioEntreObstaculos);
}

float centroUltimaX = posicionXUltimoObstaculo +
                      espacioHastaElSiguienteObstaculo / 2;
float desplazamiento;
int elementoSeccion = ElegirTipoDeElementoDeSeccion();
switch (elementoSeccion) {
    case 0: // premio
        GameObject nuevoPremio = poolDePremios.SacarObjeto();
        if (nuevoPremio != null) {
            desplazamiento =
              espacioHastaElSiguienteObstaculo / 8;
            nuevoPremio.transform.position = new
            Vector3(UnityEngine.Random.Range(
              centroUltimaX - desplazamiento,
              centroUltimaX + desplazamiento),
              UnityEngine.Random.Range(-2f, 4f),
              gameObject.transform.position.z);
            nuevoPremio.SetActive(true);
        }
        break;
    case 1: // elemento perjudicial
        GameObject nuevoElemento =
          poolDeElementosPerjudiciales.SacarObjeto();
        if (nuevoElemento != null) {
            desplazamiento =
```

```
                    espacioHastaElSiguienteObstaculo / 8;
                nuevoElemento.transform.position = new
                  Vector3(UnityEngine.Random.Range(
                    centroUltimaX - desplazamiento,
                    centroUltimaX + desplazamiento),
                    UnityEngine.Random.Range(-2f, 4f),
                    gameObject.transform.position.z);
                nuevoElemento.SetActive(true);
            }
            break;
        case 2: // nada
            break;
        default:
            break;
    }

    // se extraen dos elementos decorativos del pool y se colocan a
    // la derecha del nuevo obstáculo
    desplazamiento = espacioHastaElSiguienteObstaculo / 3;
    GameObject nuevoObjeto = poolDeDecoracion.SacarObjeto();
    if (nuevoObjeto != null) {
        nuevoObjeto.transform.position = new Vector3(
          centroUltimaX - desplazamiento, -3.4f, 1f);
        nuevoObjeto.transform.RotateAround(
          nuevoObjeto.transform.position, Vector3.up,
          UnityEngine.Random.Range(-90f, 90f));
        nuevoObjeto.SetActive(true);
    }

    desplazamiento = espacioHastaElSiguienteObstaculo / 6;
    nuevoObjeto = poolDeDecoracion.SacarObjeto();
    if (nuevoObjeto != null) {
        nuevoObjeto.transform.position = new Vector3(
          centroUltimaX + desplazamiento, -3.4f, 1f);
        nuevoObjeto.transform.RotateAround(
          nuevoObjeto.transform.position, Vector3.up,
          UnityEngine.Random.Range(-90f, 90f));
        nuevoObjeto.SetActive(true);
    }
  }
}

public int ElegirTipoDeElementoDeSeccion()
{
```

```
        float probabilidadTotal =
          nivel.probabilidadAparicionPremio +
          nivel.probabilidadAparicionElementoPerjudicial +
          nivel.probabilidadEspacioVacío;
        float numeroElegido =
          UnityEngine.Random.Range(0f, probabilidadTotal);
        int indiceElegido = 2;
        for (int i = 0; i < rangos.Count; ++i) {
            if (rangos[i].Item1 < rangos[i].Item2) {
                if ((numeroElegido >= rangos[i].Item1) &&
                    (numeroElegido < rangos[i].Item2)) {
                    indiceElegido = i;
                    break;
                }
            }
        }

        return indiceElegido;
    }

    public void SetJugadorAlPrincipioDelNivel(bool valor)
    {
        jugadorAlPrincipioDelNivel = valor;
    }

    public void CargarNivel(NivelBuceoScriptable nuevoNivel)
    {
        nivel = nuevoNivel;
        poolDeObstaculos.CrearPool(nivel.obstaculos);
        poolDePremios.CrearPool(nivel.premios);
        poolDeDecoracion.CrearPool(nivel.decoracion);
        poolDeElementosPerjudiciales.CrearPool(
          nivel.elementosPerjudiciales);
        jugadorAlPrincipioDelNivel = true;

        switch (nuevoNivel.espacioEntreObstáculos) {
            case EspacioEntreObstáculosEnBuceo.Mucho:
                minEspacioEntreObstaculos = 20f;
                maxEspacioEntreObstaculos = 25f;
                break;
            case EspacioEntreObstáculosEnBuceo.Medio:
                minEspacioEntreObstaculos = 15f;
                maxEspacioEntreObstaculos = 20f;
                break;
```

```
    case EspacioEntreObstáculosEnBuceo.Poco:
        minEspacioEntreObstaculos = 12f;
        maxEspacioEntreObstaculos = 15f;
        break;
    case EspacioEntreObstáculosEnBuceo.Mixto:
        minEspacioEntreObstaculos = 8f;
        maxEspacioEntreObstaculos = 20f;
        break;
    default:
        Debug.LogError(
          "ERROR: Espacio entre obstáculos no reconocido");
        break;
}
switch (nuevoNivel.tipoNivelBuceo) {
    case TiposNivelesBuceo.Contrarreloj:
        meta.SetActive(true);
        cartel.SetActive(false);
        break;
    default:
        meta.SetActive(false);
        cartel.SetActive(true);
        break;
}

indicadorRepetirNivel.SetActive(true);
espacioHastaElSiguienteObstaculo =
  UnityEngine.Random.Range(minEspacioEntreObstaculos,
                           maxEspacioEntreObstaculos);
limiteNivel.transform.position =
  new Vector3(nivel.distanciaALaMeta, 1f, 0f);

float probabilidadTotal =
  nivel.probabilidadAparicionPremio +
  nivel.probabilidadAparicionElementoPerjudicial +
  nivel.probabilidadEspacioVacío;
float numeroActual = 0.0f;
rangos.Add(
  new Tuple<float, float>(numeroActual, numeroActual +
                          nivel.probabilidadAparicionPremio));
numeroActual += nivel.probabilidadAparicionPremio;
rangos.Add(
  new Tuple<float, float>(numeroActual, numeroActual +
    nivel.probabilidadAparicionElementoPerjudicial));
numeroActual += nivel.probabilidadAparicionElementoPerjudicial;
```

```
        rangos.Add(
          new Tuple<float, float>(numeroActual, numeroActual +
            nivel.probabilidadEspacioVacío));
    }

    public void EliminarObstaculo(GameObject objetoEliminado)
    {
        objetoEliminado.SetActive(false);
        poolDeObstaculos.AñadirObjeto(objetoEliminado);
    }

    public void EliminarPremio(GameObject objetoEliminado)
    {
        objetoEliminado.SetActive(false);
        poolDePremios.AñadirObjeto(objetoEliminado);
    }

    public void EliminarElementoPerjudicial(GameObject elementoEliminado)
    {
        elementoEliminado.SetActive(false);
        poolDeElementosPerjudiciales.AñadirObjeto(elementoEliminado);
    }

    public void EliminarDecoracion(GameObject objetoEliminado)
    {
        objetoEliminado.SetActive(false);
        poolDeDecoracion.AñadirObjeto(objetoEliminado);
    }

    public void EliminarTodo()
    {
        // Devolver todos los premios al pool de premios
        GameObject[] premios = GameObject.FindGameObjectsWithTag("Premio");
        foreach (GameObject premio in premios) {
            if (premio.activeInHierarchy) {
              EliminarPremio(premio.GetComponent<Premio>().GetObjetoRaiz());
            }
        }

        // Devolver todos los objetos decorativos al pool de decoración
        GameObject[] decoracion =
          GameObject.FindGameObjectsWithTag("Decoracion");
        foreach (GameObject objeto in decoracion) {
            if (objeto.activeInHierarchy) {
```

```
                EliminarDecoracion(objeto);
            }
        }

        // Devolver todas las medusas al pool de objetos perjudiciales
        GameObject[] medusas = GameObject.FindGameObjectsWithTag("Medusa");
        foreach (GameObject medusa in medusas) {
            if (medusa.activeInHierarchy) {
                EliminarElementoPerjudicial(medusa);
            }
        }

        // Devolver todos los obstáculos al pool
        Obstaculo[] obstaculos =
          FindObjectsOfType(typeof(Obstaculo)) as Obstaculo[];
        foreach (Obstaculo obstaculo in obstaculos) {
            if (obstaculo.gameObject.activeInHierarchy) {
                EliminarObstaculo(obstaculo.gameObject);
            }
        }

        // Devolver todas las penalizaciones al pool
        GameObject[] penalizaciones =
          GameObject.FindGameObjectsWithTag("Penalizacion");
        foreach (GameObject penalizacion in penalizaciones) {
            if (penalizacion.activeInHierarchy) {
              EliminarElementoPerjudicial(
              penalizacion.GetComponent<Premio>().GetObjetoRaiz());
            }
        }
    }
}
```

Apéndice A.5 Clase `LimiteIzquierdo`

```
public class LimiteIzquierdo : MonoBehaviour
{
    public GestionNivel gestionNivel;

    private void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.tag == "Obstaculo") {
```

```
            // en las barreras, el GameObject marcado como "Obstaculo" es el
            // collider llamado "MetaObstaculo"
            gestionNivel.EliminarObstaculo(
              collider.gameObject.GetComponent<MetaObstaculo>().
              objetoObstaculo.gameObject);
        } else if (collider.gameObject.tag == "Premio") {
            gestionNivel.EliminarPremio(
              collider.gameObject.GetComponent<Premio>().GetObjetoRaiz());
        } else if (collider.gameObject.tag == "Decoracion") {
            gestionNivel.EliminarDecoracion(collider.gameObject);
        } else if (collider.gameObject.tag == "Medusa") {
            gestionNivel.EliminarElementoPerjudicial(collider.gameObject);
        } else if (collider.gameObject.tag == "Penalizacion") {
            gestionNivel.EliminarElementoPerjudicial(
              collider.gameObject.GetComponent<Premio>().GetObjetoRaiz());
        }
    }
}
```

Apéndice A.6 Clase `LimiteNivel`

```
public class LimiteNivel : MonoBehaviour
{
    [SerializeField]
    private TareaBuceo tareaBuceo;

    private void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.tag == "Player") {
            tareaBuceo.GestionarFinalDelNivel();
            FindObjectOfType<GestionNivel>().EliminarTodo();
        }
    }
}
```

Apéndice A.7 Clase `GestionScroll`

```
public class GestionScroll : MonoBehaviour
{
```

```
public Camera camara;
public GameObject objetoPrincipal;
public GameObject objetoSecundario;

private float anchoCamara;
private float anchoObjeto;
private float diferenciaDeAncho;

void Start()
{
    anchoCamara = 2f * camara.orthographicSize * camara.aspect;
    anchoObjeto = objetoPrincipal.GetComponent<BoxCollider>().size.x;
    diferenciaDeAncho = anchoObjeto - anchoCamara;
}

void Update()
{
    // se desplaza el objeto principal
    float diferenciaXCameraYObjetoPrincipal =
      camara.gameObject.transform.position.x -
      objetoPrincipal.transform.position.x;
    bool camaraALaDerechaDelObjetoPrincipal =
      diferenciaXCameraYObjetoPrincipal >= 0;
    diferenciaXCameraYObjetoPrincipal =
      Mathf.Abs(diferenciaXCameraYObjetoPrincipal);

    if (diferenciaXCameraYObjetoPrincipal > (diferenciaDeAncho / 2)) {
        if (camaraALaDerechaDelObjetoPrincipal) {
            // si hay un hueco por la derecha, hay que colocar el objeto
            // secundario a la derecha del principal
            objetoSecundario.transform.position =
              objetoPrincipal.transform.position +
              anchoObjeto * Vector3.right;
        } else {
            // si hay un hueco por la izquierda, hay que colocar el
            // objeto secundario a la izquierda del principal
            objetoSecundario.transform.position =
              objetoPrincipal.transform.position +
              anchoObjeto * Vector3.left;
        }
    }

    // si el objeto principal sale del frustum de la cámara, el
    // secundario pasa a ser el principal (da igual el lado por el que
```

```
            // salga)
            if (diferenciaXCameraYObjetoPrincipal >=
                (anchoCamara / 2 + anchoObjeto / 2)) {
                GameObject aux = objetoPrincipal;
                objetoPrincipal = objetoSecundario;
                objetoSecundario = aux;
            }
        }
    }
```

Apéndice A.8 Clase `GestionHUD`

```
public class GestionHUD : MonoBehaviour
{
    public Animator crossfadeAnimator;

    [Header("Vidas")]
    public TextMeshProUGUI cantidadVidasTexto;
    public GameObject cantidadVidasUI;

    [Header("Puntuación")]
    public TextMeshProUGUI puntuacionTexto;

    [Header("Objetivo")]
    public GameObject gameObjectHUDObjetivo;
    public TextMeshProUGUI textoHUDObjetivo;
    public Image iconoHUDObjetivo;
    private int cantidadActual;
    private int cantidadObjetivo;

    [Header("Iconos")]
    [SerializeField]
    private Sprite iconoNivelPremios;
    [SerializeField]
    private Sprite iconoNivelObstáculos;
    [SerializeField]
    private Sprite iconoNivelContrarreloj;

    public void CargaHUD(TiposNivelesBuceo tipoNivel, int objetivo)
    {
        // como en los niveles contrarreloj el objetivo es llegar a la meta,
        // se ignora el valor de cantidadObjetivo
```

```
        cantidadActual = 0;
        cantidadObjetivo = objetivo;
        switch (tipoNivel) {
            case TiposNivelesBuceo.Esquivar:
                iconoHUDObjetivo.sprite = iconoNivelObstáculos;
                textoHUDObjetivo.text = $"0 / {cantidadObjetivo}";
                gameObjectHUDObjetivo.SetActive(true);
                break;
            case TiposNivelesBuceo.RecogerPremios:
                iconoHUDObjetivo.sprite = iconoNivelPremios;
                textoHUDObjetivo.text = $"0 / {cantidadObjetivo}";
                gameObjectHUDObjetivo.SetActive(true);
                break;
        }
    }

    public void ActualizarVidas(int numeroVidas, bool recibeDaño)
    {
        cantidadVidasTexto.text = "x " + numeroVidas.ToString();

        if (recibeDaño) {
            iTween.ShakeScale(cantidadVidasUI, new Vector3(0.5f, 0.5f,
                              0.5f), 1f);
        }

        switch (numeroVidas) {
            case 1:
                cantidadVidasTexto.color = Color.yellow;
                break;
            case 0:
                cantidadVidasTexto.color = Color.red;
                break;
            default:
                break;
        }
    }

    public void ActualizarProgresoObjetivo(int cambio)
    {
        cantidadActual += cambio;
        textoHUDObjetivo.text = $"{cantidadActual} / {cantidadObjetivo}";
        if (cantidadActual >= cantidadObjetivo) {
            textoHUDObjetivo.color = Color.green;
        }
```

```
    }

    public void ActualizarPuntuacion(int puntuacion, bool resaltarTexto,
                                     Color colorTexto)
    {
        puntuacionTexto.text = $"Puntos: {puntuacion}";
        if (resaltarTexto) {
            StartCoroutine(ResaltarTextoPuntuación(colorTexto));
        }
    }

    private IEnumerator ResaltarTextoPuntuación(Color colorTexto)
    {
        puntuacionTexto.color = colorTexto;
        yield return new WaitForSeconds(0.3f);
        puntuacionTexto.color = Color.white;
        yield return null;
    }

    public void OscurecerPantalla()
    {
        crossfadeAnimator.SetTrigger("StartCrossfade");
    }

    public void AclararPantalla()
    {
        crossfadeAnimator.SetTrigger("EndCrossfade");
    }
}
```

Apéndice A.9 Clase `GestionTiempo`

```
public class GestionTiempo : MonoBehaviour
{
    public GameObject interfazTiempo;
    public Image barraTiempo;

    private float segundosRestantes;
    private float tiempoMaximo;
    private bool temporizadorActivo = false;
    private TareaBuceo tareaBuceo;
```

```
    public float SegundosRestantes {
        get {
            return (temporizadorActivo) ? segundosRestantes : -1f;
        }
    }

    private void Start()
    {
        tareaBuceo = FindObjectOfType<TareaBuceo>();
    }

    private void Update()
    {
        if (temporizadorActivo) {
            segundosRestantes -= Time.deltaTime;
            if (segundosRestantes <= 0) {
                tareaBuceo.TiempoExcedido();
            }

            float porcentaje = segundosRestantes / tiempoMaximo;
            barraTiempo.fillAmount = porcentaje;
        }
    }

    public void IniciarTemporizador(float tiempo)
    {
        temporizadorActivo = true;
        segundosRestantes = tiempo;
        tiempoMaximo = tiempo;
        interfazTiempo.SetActive(true);
    }

    public void PararTemporizador()
    {
        temporizadorActivo = false;
    }

    public void AñadirTiempo(float segundosAñadidos)
    {
        segundosRestantes += segundosAñadidos;
        segundosRestantes = (segundosRestantes > tiempoMaximo) ?
                            tiempoMaximo : segundosRestantes;
    }
}
```

Apéndice A.10 Clase `MovimientoJugadorBuceo`

```
public class MovimientoJugadorBuceo : MonoBehaviour
{
    [SerializeField]
    private float velocidadActual;
    private float velocidadBase;
    private float velocidadReducida;

    [SerializeField]
    private float offsetXObjetivoAMirar;

    private TareaBuceo tareaBuceo;
    private new Rigidbody rigidbody;

    public float VelocidadActual {
        get { return velocidadActual; }
        set { velocidadActual = value; }
    }

    public float VelocidadBase {
        get { return velocidadBase; }
    }

    public float VelocidadReducida {
        get { return velocidadReducida; }
    }

    void Start()
    {
        rigidbody = gameObject.GetComponent<Rigidbody>();
        tareaBuceo = FindObjectOfType<TareaBuceo>();
        velocidadBase = tareaBuceo.Nivel.velocidadBase;
        velocidadReducida = tareaBuceo.Nivel.velocidadReducida;
        velocidadActual = velocidadBase;
    }

    void FixedUpdate()
    {
        if (tareaBuceo.TareaBloqueada) {
            return;
        }
```

```
        // movimiento con eye tracker
        GazePoint gazePoint = TobiiAPI.GetGazePoint();
        if (gazePoint.IsValid) {
            Vector3 posicionVista =
              Camera.main.ScreenToWorldPoint(gazePoint.Screen);
            Vector3 posicionAMirar =
              new Vector3(transform.position.x + offsetXObjetivoAMirar,
                          posicionVista.y, 0f);

            gameObject.transform.right =
              Vector3.Lerp(gameObject.transform.right,
                           (posicionAMirar - transform.position).normalized,
                           velocidadActual * Time.fixedDeltaTime);
            rigidbody.MovePosition(transform.position + transform.right *
                                   velocidadActual * Time.fixedDeltaTime);
        }
    }
}
```

Apéndice A.11 Clase `EstadoJugadorBuceo`

```
public class EstadoJugadorBuceo : MonoBehaviour
{
    public TareaBuceo tareaBuceo;
    public GameObject escudo;
    public GameObject cuerpoPez;
    public ParticleSystem particulasEscudo;

    private int vidas;
    private bool invulnerable;
    private bool escudoActivo;
    private RenderizadoJugadorBuceo rendererPez;
    private MovimientoJugadorBuceo movimientoPez;
    private GestionHUD gestionHUD;

    private CinemachineVirtualCamera cinemachineVirtualCamera;
    private CinemachineBasicMultiChannelPerlin
      cinemachineBasicMultiChannelPerlin;
    private float duracionSacudidaEnSegundos = 0.3f;
    private float intensidadSacudida = 3f;
    private float duracionInvulnerabilidadEnSegundos = 2f;
```

```
private float duracionParpadeo = 0.1f;

private float duracionPerdidaVelocidad = 2f;

public bool EsInvulnerable {
    get { return invulnerable; }
    set { invulnerable = value; }
}

public bool EscudoActivo {
    get { return escudoActivo; }
    set { escudoActivo = value; }
}

public int Vidas {
    get { return vidas; }
}

void Start()
{
    vidas = 3;
    EsInvulnerable = false;
    EscudoActivo = false;
    rendererPez = gameObject.GetComponent<RenderizadoJugadorBuceo>();
    movimientoPez = gameObject.GetComponent<MovimientoJugadorBuceo>();
    cinemachineVirtualCamera =
      FindObjectOfType<CinemachineVirtualCamera>();
    cinemachineBasicMultiChannelPerlin =
      cinemachineVirtualCamera.
      GetCinemachineComponent<CinemachineBasicMultiChannelPerlin>();
    gestionHUD = FindObjectOfType<GestionHUD>();
    gestionHUD.ActualizarVidas(vidas, false);
}

public void RecibirGolpe()
{
    if (tareaBuceo.TareaBloqueada) {
        return;
    }

    if (escudoActivo) {
        // el jugador pierde el escudo al golpear el obstáculo
        RomperEscudo();
    } else if (!invulnerable) {
```

```
            iTween.ShakeScale(cuerpoPez, new Vector3(0.5f, 0.5f, 0.5f), 1f);
            StartCoroutine(CorrutinaSacudirCamara());
            PerderVida();
            StartCoroutine(CorrutinaInvulnerabilidad());
            StartCoroutine(CorrutinaPerdidaVelocidad());
        }
    }

    public void RomperEscudo()
    {
        FindObjectOfType<Audio>().FeedbackChoqueUsandoEscudo();
        EscudoActivo = false;
        particulasEscudo.Play();
        escudo.SetActive(false);
        StartCoroutine(CorrutinaInvulnerabilidad());
    }

    public void PerderVida()
    {
        if (tareaBuceo.TareaBloqueada) {
            return;
        }

        --vidas;
        gestionHUD.ActualizarVidas(vidas, true);
        FindObjectOfType<Audio>().FeedbackError();
        if (vidas == 0) {
            // terminar partida
            tareaBuceo.JugadorSinVidas();
        }
    }

    public void ActivarEscudo()
    {
        escudo.SetActive(true);
        EscudoActivo = true;
    }

    private IEnumerator CorrutinaSacudirCamara()
    {
        cinemachineBasicMultiChannelPerlin.m_AmplitudeGain =
          intensidadSacudida;
        yield return new WaitForSeconds(duracionSacudidaEnSegundos);
        cinemachineBasicMultiChannelPerlin.m_AmplitudeGain = 0f;
```

```
    }

    private IEnumerator CorrutinaInvulnerabilidad()
    {
        EsInvulnerable = true;

        float numeroParpadeosInvulnerabilidad =
          duracionInvulnerabilidadEnSegundos / (float)duracionParpadeo;
        for (int i = 0; i < numeroParpadeosInvulnerabilidad; ++i) {
            rendererPez.CambiarVisibilidadPez(false);
            yield return new WaitForSeconds(duracionParpadeo / 2f);
            rendererPez.CambiarVisibilidadPez(true);
            yield return new WaitForSeconds(duracionParpadeo / 2f);
        }

        rendererPez.CambiarVisibilidadPez(true);
        EsInvulnerable = false;
    }

    private IEnumerator CorrutinaPerdidaVelocidad()
    {
        movimientoPez.VelocidadActual = movimientoPez.VelocidadReducida;
        yield return new WaitForSeconds(duracionPerdidaVelocidad);
        movimientoPez.VelocidadActual = movimientoPez.VelocidadBase;
    }
}
```

Apéndice A.12 Clase `Obstaculo`

```
public class Obstaculo : MonoBehaviour
{
    private TareaBuceo tareaBuceo;
    private bool jugadorHaChocado;

    [SerializeField]
    private GameObject meta;

    [SerializeField]
    private GameObject entrada;

    [SerializeField]
    private GameObject salida;
```

```
    public GameObject Meta {
        get { return meta; }
    }

    public GameObject Entrada {
        get { return entrada; }
    }

    public GameObject Salida {
        get { return salida; }
    }

    void Start()
    {
        tareaBuceo = FindObjectOfType<TareaBuceo>();
    }

    public void Reiniciar()
    {
        jugadorHaChocado = false;
    }

    public void ObstaculoSuperado()
    {
        if (!jugadorHaChocado) {
            // si el jugador no ha chocado con ninguna de las barreras,
            // recibe los puntos que le corresponden
            tareaBuceo.GestionarObstaculoPasado();
        }
    }

    public void ObstaculoGolpeado()
    {
        jugadorHaChocado = true;
    }
}
```

Apéndice A.13 Clase `BarreraObstaculo`

```
public class BarreraObstaculo : MonoBehaviour
{
```

```
    public Obstaculo objetoObstaculo;

    private void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.tag == "Player") {
            objetoObstaculo.ObstaculoGolpeado();
            EstadoJugadorBuceo estadoJugadorBuceo =
              collider.gameObject.GetComponent<EstadoJugadorBuceo>();
            estadoJugadorBuceo.RecibirGolpe();
        }
    }

    private void OnTriggerStay(Collider collider)
    {
        if (collider.gameObject.tag == "Player") {
            objetoObstaculo.ObstaculoGolpeado();
            EstadoJugadorBuceo estadoJugadorBuceo =
              collider.gameObject.GetComponent<EstadoJugadorBuceo>();
            estadoJugadorBuceo.RecibirGolpe();
        }
    }
}
```

Apéndice A.14 Clase `MetaObstaculo`

```
public class MetaObstaculo : MonoBehaviour
{
    public Obstaculo objetoObstaculo;

    private void OnTriggerExit(Collider collider)
    {
        if (collider.gameObject.tag == "Player") {
            objetoObstaculo.ObstaculoSuperado();
        }
    }
}
```

Apéndice A.15 Clase `Medusa`

```
public class Medusa : MonoBehaviour
{
    private float velocidad = 0.1f;
    private Vector3 direccionMovimiento;

    private void Start()
    {
        direccionMovimiento = Vector3.up;
    }

    private void Update()
    {
        if (direccionMovimiento.y == -1) {
            if (gameObject.transform.position.y < -1f) {
                direccionMovimiento = Vector3.up;
            }
        } else if (direccionMovimiento.y == 1) {
            if (gameObject.transform.position.y > 3f) {
                direccionMovimiento = Vector3.down;
            }
        }

        gameObject.transform.Translate(
          direccionMovimiento * velocidad * Time.deltaTime);
    }

    private void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.tag == "Player") {
            EstadoJugadorBuceo estadoJugadorBuceo =
              collider.gameObject.GetComponent<EstadoJugadorBuceo>();
            estadoJugadorBuceo.RecibirGolpe();
        }
    }

    private void OnTriggerStay(Collider collider)
    {
        if (collider.gameObject.tag == "Player") {
            EstadoJugadorBuceo estadoJugadorBuceo =
              collider.gameObject.GetComponent<EstadoJugadorBuceo>();
```

```
            estadoJugadorBuceo.RecibirGgolpe();
        }
    }
}
```

Apéndice A.16 Clase `Objeto`

```
public class Objeto : MonoBehaviour
{
    protected GestionNivel gestionNivel;
    protected TareaBuceo tareaBuceo;

    protected void Start()
    {
        gestionNivel = FindObjectOfType<GestionNivel>();
        tareaBuceo = FindObjectOfType<TareaBuceo>();
    }
}
```

Apéndice A.17 Clase `Premio`

```
public abstract class Premio : Objeto
{
    [SerializeField]
    protected GameObject objetoRaizPrefab;

    [SerializeField]
    protected int puntos;

    protected abstract void AplicarEfecto(GameObject jugador);

    public GameObject GetObjetoRaiz() {
        return objetoRaizPrefab;
    }

    private void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.tag == "Player") {
            AplicarEfecto(collider.gameObject);
```

```
            gestionNivel.EliminarPremio(objetoRaizPrefab);
        }
    }
}
```

Apéndice A.18 Clase `PremioBasico`

```
public class PremioBasico : Premio
{
    protected override void AplicarEfecto(GameObject jugador)
    {
        tareaBuceo.AgregarPuntuacion(puntos);
        tareaBuceo.GestionarPremioRecogido();
    }
}
```

Apéndice A.19 Clase `PremioEscudo`

```
public class PremioEscudo : Premio
{
    protected override void AplicarEfecto(GameObject jugador)
    {
        tareaBuceo.AgregarPuntuacion(puntos);
        jugador.GetComponent<EstadoJugadorBuceo>().ActivarEscudo();
        tareaBuceo.GestionarPremioRecogido();
    }
}
```

Apéndice A.20 Clase `PremioTiempoExtra`

```
public class PremioTiempoExtra : Premio
{
    protected float tiempoAñadido = 5f;
    private GestionTiempo gestionTiempo;

    protected new void Start()
    {
```

```
        base.Start();
        gestionTiempo = FindObjectOfType<GestionTiempo>();
    }

    protected override void AplicarEfecto(GameObject jugador)
    {
        tareaBuceo.AgregarPuntuacion(puntos);
        gestionTiempo.AñadirTiempo(tiempoAñadido);
        tareaBuceo.GestionarPremioRecogido();
    }
}
```

Apéndice A.21 Clase `PenalizacionTiempo`

```
public class PenalizacionTiempo : Premio
{
    protected int tiempoRestado = 5;
    private GestionTiempo gestionTiempo;

    protected new void Start()
    {
        base.Start();
        gestionTiempo = FindObjectOfType<GestionTiempo>();
    }


    protected override void AplicarEfecto(GameObject jugador)
    {
        EstadoJugadorBuceo estadoJugadorBuceo =
            jugador.GetComponent<EstadoJugadorBuceo>();
        if (estadoJugadorBuceo.EscudoActivo) {
            estadoJugadorBuceo.RomperEscudo();
        } else if (!estadoJugadorBuceo.EsInvulnerable) {
            tareaBuceo.AgregarPuntuacion(puntos);
            gestionTiempo.AñadirTiempo(-tiempoRestado);
            FindObjectOfType<Audio>().FeedbackError();
        }
    }
}
```

Apéndice A.22 Método `NuevoRegistro`

```
protected override RegistroPosicionOcular NuevoRegistro(
  float tiempo, int x, int y)
{
    // Vidas
    int vidas = estadoJugadorBuceo.Vidas;

    // Posición X del jugador
    float posicionXJugador = jugador.transform.position.x;

    // Posición Y del jugador
    float posicionYJugador = jugador.transform.position.y;

    // premios por recoger
    int premiosObjetivo =
      (Nivel.tipoNivelBuceo == TiposNivelesBuceo.RecogerPremios) ?
      Nivel.premiosARecoger : -1;

    // obstáculos por esquivar
    int obstaculosObjetivo =
      (Nivel.tipoNivelBuceo == TiposNivelesBuceo.Esquivar) ?
      Nivel.obstaculosAEsquivar : -1;

    // tiempo restante
    float segundosRestantes =
      (Nivel.tipoNivelBuceo == TiposNivelesBuceo.Contrarreloj) ?
      gestionTiempo.SegundosRestantes : -1f;

    return new RegistroPosicionOcularTareaBuceo(
        tiempo, x, y, this.puntuacion, vidas, posicionXJugador,
        posicionYJugador, premiosRecogidos, premiosObjetivo,
        obstaculosEsquivados, obstaculosObjetivo,
        Nivel.distanciaALaMeta, segundosRestantes
    );
}
```

Apéndice A.23
Clase `RegistroPosicionOcularTareaBuceo`

```
public class RegistroPosicionOcularTareaBuceo : RegistroPosicionOcular
```

```
{
    protected float xJugador, yJugador, xMeta, segundosRestantes;
    protected int puntuacion, vidas, premiosRecogidos, premiosObjetivo,
      obstaculosEsquivados, obstaculosObjetivo;

    public RegistroPosicionOcularTareaBuceo(
      float tiempo, int x, int y, int newPuntuacion, int newVidas,
      float newXJugador, float newYJugador, int newPremiosRecogidos,
      int newPremiosObjetivo, int newObstaculosEsquivados,
      int newObstaculosObjetivo, float newXMeta, float newSegundosRestantes
      ) :
    base(tiempo, x, y)
    {
        xJugador = newXJugador;
        yJugador = newYJugador;
        puntuacion = newPuntuacion;
        vidas = newVidas;
        premiosRecogidos = newPremiosRecogidos;
        premiosObjetivo = newPremiosObjetivo;
        obstaculosEsquivados = newObstaculosEsquivados;
        obstaculosObjetivo = newObstaculosObjetivo;
        xMeta = newXMeta;
        segundosRestantes = newSegundosRestantes;
    }

    public override string RegistroFormateadoParaEscribirEnDisco()
    {
        string tiempoString = tiempo.ToString("0.0000");
        string xJugadorString = xJugador.ToString("0.0000");
        string yJugadorString = yJugador.ToString("0.0000");
        string xMetaString = xMeta.ToString("0.0000");
        string segundosRestantesString = segundosRestantes.ToString("0.00");
        return  $"{tiempoString};{puntuacion};{vidas};{x};{y};
                {xJugadorString};{yJugadorString};{premiosRecogidos};
                {premiosObjetivo};{obstaculosEsquivados};
                {obstaculosObjetivo}; {xMetaString};
                {segundosRestantesString}";
    }
}
```