TELECOMUNICACIÓN

Campus Sur

POLITÉCNICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Mejora de la lógica e incorporación de sistemas para el videojuego terapéutico *Phiby's Adventure 3D*.

AUTOR: Juan Ignacio Fuentes-Pila Martín

TITULACIÓN: Grado en Ingeniería de Sonido e Imagen

TUTOR: Martina Eckert

DEPARTAMENTO: Ingeniería Audiovisual y Comunicaciones

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: Francisco Javier del Río Martín

TUTOR: Martina Eckert

SECRETARIO: Enrique Rendón Angulo

Fecha de lectura:

Calificación:

El Secretario,

Agradecimientos

A mis profesores, por su esfuerzo en sacar lo mejor de mí. A mi familia, por todo el apoyo y cariño que me han dado. Y a ese "barquito" de amigos con el que zarpé hacia esta aventura. A todos ellos, por haberme hecho ser quien soy hoy.

Resumen

Este proyecto de fin de grado consiste en una mejora de la lógica y de la progresión del videojuego terapéutico para personas con movilidad reducida *Phiby's Adventure 3D*. Los objetivos de este proyecto están enfocados por un lado a los pacientes, para que tengan una experiencia más satisfactoria a la hora de jugar y, por otro, a los desarrolladores futuros del proyecto, con el fin de establecer un desarrollo más ágil y sencillo del videojuego.

Para llevar a cabo el proyecto se han utilizado diferentes tecnologías entre las que se encuentran los softwares de edición del proyecto (*Unity* y *Visual Studio Code*) una cámara de captación 3D y una plataforma web que es utilizada por los terapeutas para establecer los parámetros de los ejercicios integrados en el juego. El lenguaje de programación utilizado para su desarrollo ha sido C#.

Se trata de un proyecto que continúa el trabajo empezado por otros alumnos de la escuela y sobre el que se ha tratado de encontrar puntos de mejora del videojuego con el fin de ofrecer un mejor producto. Esto ha llevado a plantear mejoras en la programación, tratando de solucionar errores existentes e incorporando nuevos sistemas que fomenten aspectos clave de la jugabilidad típicos en un videojuego de aventuras.

Las características que se han incorporado consisten en: un sistema de misiones que ofrece nuevas formas de progreso en el juego, una interfaz visual (en forma de mapa) que otorga al usuario de una visión general del entorno y, por último, un sistema de control mediante gestos que permite al jugador moverse por los diferentes menús y que incorpora un sistema de calibración de los movimientos desarrollado por otro estudiante de la escuela, pudiendo adaptar la precisión de los movimientos en función de las necesidades de cada usuario. Estas nuevas características se han incorporado de manera complementaria, para hacer más satisfactoria la experiencia del usuario.

Abstract

The final degree project presented is an improvement of the logical behavior and the progression of the therapeutic video game designed for people with reduced mobility *Phiby's Adventure 3D*. The objectives of this project are, on one hand, focused on the patients, so they have a more satisfactory experience when they play the game. And, on the other hand, this project is focused on future developers, to create a more agile and simple development of the videogame.

To carry out the project, different technologies have been used, including the project's editing softwares (*Unity* and *Visual Studio Code*), a 3D capture camera and a web platform. The web is a tool for the therapists to determinate the parameters of the exercises integrated in the game, which are used for the rehabilitation of the patients. The code language used in this project has been C#.

This project continues the work started by other students, with the intention to offer a better product and a better experience for the users and developers. This has led to improvements in the logical structure of the video game, to the solution of errors and to the incorporation of new systems that promote key aspects of gameplay in an adventure video game.

The features that have been incorporated consist of: a quest system that offers new ways to progress in the game, a visual interface (a map) that gives the user an overview of the environment and, finally, a gesture control system that allows the player to move through the different menus and that incorporates a movement calibration system developed by another student of the school, being able to adapt the precision of the movements according to the needs of each user. These new features have been incorporated in a complementary way, to make the user experience more satisfactory.

Índice

Índice de figuras

Lista de acrónimos

GAMMA:	Grupo de Aplicaciones Multimedia y Acústica
CITSEM:	Centro de Investigación de Tecnologías Software y Sistemas Multimedia para la sostenibilidad
PFG:	Proyecto de Fin de Grado
Blexer:	Blender Exergames
K2UM:	Kinect to Unity Middleware
UI:	User Interface
HUD:	Head Up Display
NPC:	Non-Playable Character

1. Introducción

Los videojuegos se han asentado como una de las formas de entretenimiento más extendidas de la actualidad. No obstante, los videojuegos también se han utilizado como una vía de desarrollo para nuevas tecnologías y avances científicos. Es por ello por lo que organizaciones como el Grupo de Investigación de Aplicaciones Multi-Media y Acústica (GAMMA) de la Universidad Politécnica de Madrid (UPM) ubicado en el Centro de Investigaciones de Tecnologías Software y Sistemas Multimedia para la sostenibilidad (CITSEM) tienen una respetable trayectoria trabajando en el desarrollo de videojuegos y otros proyectos con objetivos más allá del mero entretenimiento, como es el caso del proyecto *Blender Exergames* (*Blexer*) cuyos fines son terapéuticos.

El proyecto de fin de grado (PFG) de esta memoria se incluye dentro del proyecto *Blexer*, particularmente dentro del videojuego terapéutico *Phiby's Adventure 3D* que inició su desarrollo en el año 2018 con el objetivo de ayudar a niños con problemas de movilidad reducida en sesiones de rehabilitación lúdicas. Tanto *Phiby's Adventure*, como el resto de los videojuegos dentro del entorno *Blexer*, hacen uso del dispositivo *Kinect* de captura de movimiento de la empresa *Microsoft* y de un software desarrollado por el equipo de GAMMA llamado *Kinect To Unity Middleware* (K2UM) para realizar los ejercicios desarrollados en el videojuego. Estos ejercicios son configurables por los terapeutas a través de la web *Blexer-med* y les permite obtener los resultados de sus pacientes durante las sesiones de rehabilitación.

En este proyecto se ha trabajado en la estructura lógica y el sistema de progresión de *Phiby's Adventure 3D*, ya que al empezar el proyecto se encontraron algunos puntos de mejora sobre los que trabajar para hacer más satisfactoria la experiencia de usuario. Por ello, se han incorporado nuevos sistemas que tratan de mejorar tanto el avance en la historia, como en el rendimiento del videojuego, de modo que las sesiones de rehabilitación de los usuarios sean más gratificantes. Además, se ha tratado de mejorar la experiencia de desarrollo del juego para futuros estudiantes, simplificando y mejorando ciertos aspectos de la programación.

Todo este trabajo se ha llevado a cabo con la colaboración de la alumna Laura Álvaro Gil [1] quien ha trabajado de manera paralela en el videojuego en su proyecto de fin de grado.

2. Antecedentes y entorno del proyecto

GAMMA [3] lleva años dentro de CITSEM [4] desarrollando videojuegos con fines terapéuticos, con el objetivo de ayudar a la rehabilitación de personas con discapacidades motrices y mediante la realización de ejercicios planteados como juegos interactivos, los cuales son supervisados por un equipo de terapeutas especializados.

Uno de los proyectos bajo el paraguas de GAMMA es el entorno *Blexer* [5], que consiste en una plataforma web con una base de datos que alberga los ajustes y los resultados de los jugadores, varios juegos realizados en *Blender* y *Unity*, diferentes cámaras *Kinect* de *Microsoft* [6] y dos *middleware* que transmiten los datos entre las cámaras, la base de datos y los juegos.

Uno de los videojuegos de *Blexer* es *Phiby's Adventure 3D* un juego del género de aventuras desarrollado en Unity en el que el jugador toma el control del personaje *Phiby* mediante el dispositivo *Kinect* y el software K2UM desarrollado por César Luaces Vela [7]. El proyecto ha pasado por un largo desarrollo hasta alcanzar su estado actual, llegando a un tamaño amplio en el que se han incorporado multitud de mecánicas y sistemas desde su primera versión realizada en *Blender*.

Phiby's Adventure se desarrolla en una isla (figura 1) donde el jugador puede explorar y avanzar en la historia del videojuego mientras va completando ejercicios que ayudan a su rehabilitación. Haoru Qin [8] fue la encargada de desarrollar un sistema de progreso estructurado en *checkpoints* en base a dicha historia. En esta estructura, el jugador avanza controlando al personaje principal, *Phiby*, cumpliendo objetivos y tareas que son dados por los NPCs de la isla. Estas tareas se realizan en forma de ejercicio físico, captándose los movimientos del jugador y siendo representados por *Phiby* en la escena. En el apartado 2.3 Estructura lógica de *Phiby's Adventure 3D* se especifica cuáles son los *checkpoints*, cuándo se actualizan, cuáles son los objetivos por cumplir y cuál es su relación con los ejercicios disponibles en el juego.

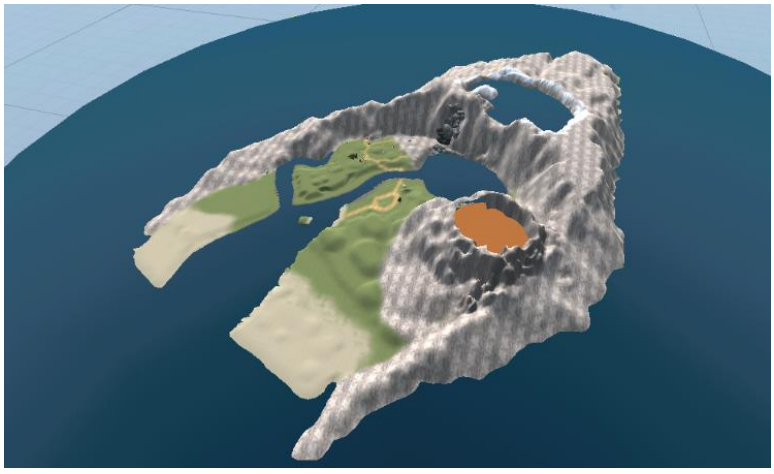


Figura 1. Captura del terreno de *Phiby's Adventure* en el editor de *Unity*.

2.1. Entorno Blexer

El entorno *Blexer* es un sistema de ejercicios físicos para personas con diversidad funcional, basado en videojuegos configurables a través de internet, el cual ha incorporado una variedad de videojuegos y tecnologías de captura de movimientos, permitiendo a terapeutas especializados trabajar con sus pacientes distintos ejercicios que son claves para el desarrollo de su rehabilitación.

Con el dispositivo de captura de movimiento, *Kinect One* de *Microsoft*, se realiza la captura de movimiento, pudiendo alternar entre un control por teclado (pensado para los desarrolladores) o por *Kinect* a través de la letra K. El *middleware* K2UM fue inicialmente creado para transmitir los datos de movimiento al juego. Pero, con el paso de tiempo, se le ha dado el propósito de servir de intermediario entre el videojuego y el servidor web creado por Mónica Jiménez [9] y Alba Aguilar López [10], de modo que los terapeutas puedan configurar la dificultad de los juegos y ver los resultados de los jugadores a distancia.

Uno de los requisitos a la hora de ejecutar las aplicaciones de *Blexer* es el de ejecutar previamente el *middleware* para poder utilizar el dispositivo. En la Figura 2 se presenta un diagrama que muestra cuáles son las conexiones servidor-ordenador de cada uno de los participantes (terapeuta y usuario) y su conexión con el dispositivo *Kinect*.

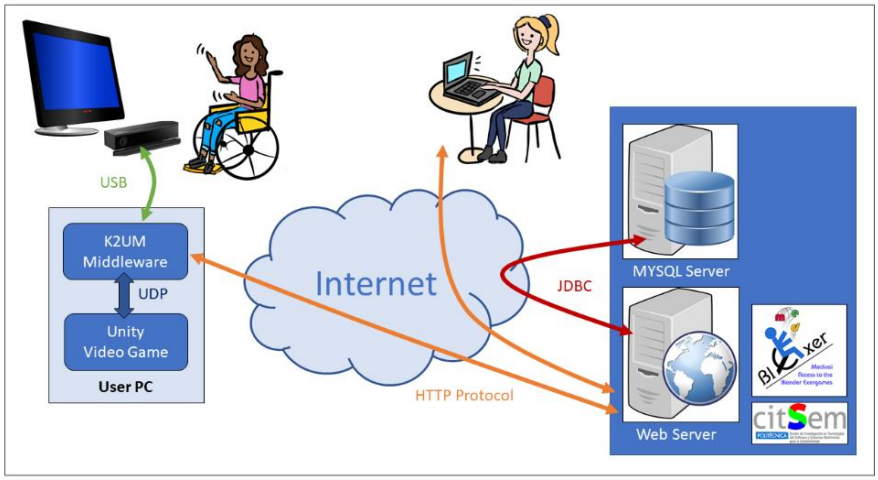


Figura 2. Diagrama de comunicación con el servidor desde los distintos dispositivos.

Al iniciar el *middleware* se abre la ventana de la Figura 3 donde el paciente o el terapeuta pueden introducir su nombre de usuario y su contraseña, cargar una configuración alojada en la web o continuar con una configuración por defecto. Una vez cargada la configuración, se muestra la interfaz de la Figura 4, donde el usuario puede iniciar la aplicación. También pueden enviarse los resultados de los ejercicios al servidor, visualizar el esqueleto que genera el dispositivo y ver otra información como los datos de envío UDP.

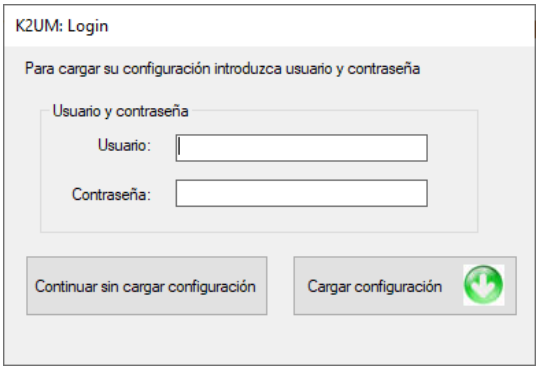


Figura 3. Ventana de *login* del *middleware* K2UM.

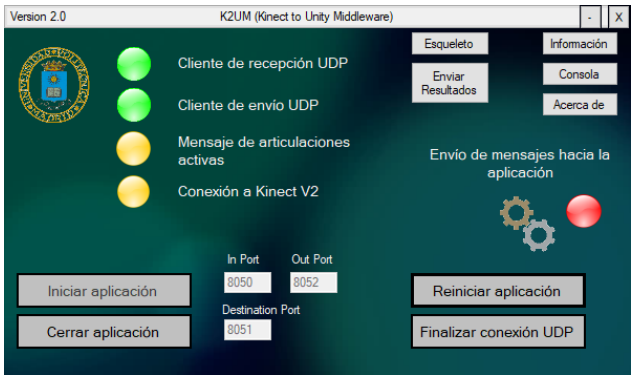


Figura 4. Interfaz de uso del *middleware* K2UM.

Por otro lado, en el servidor son modificables los parámetros de cada ejercicio y, al completar las distintas secciones del videojuego, los resultados son enviados de vuelta a la web a través del *middleware*, lo que permite almacenar el progreso hecho por el paciente en su rehabilitación y llevar un registro de su evolución en las diferentes actividades. La Figura 5 muestra un ejemplo de la interfaz del terapeuta con resultados reales, obtenidos con la versión anterior del videojuego.

Figura 5. Ejemplo de interfaz de terapeuta con una versión anterior del videojuego.

2.2. Unity, Visual Studio Code y GitHub

El entorno de desarrollo es el motor de videojuegos multiplataforma *Unity* [11], creado por la empresa *Unity Technologies* y desarrollado en C++ y C# para la creación de videojuegos. Se trata de una plataforma gratuita y fácil de usar que permite crear y editar entornos 3D (o 2D) mediante el uso de objetos que se instancian en escenas de ejecución (donde se desarrolla la acción del videojuego). Este software cuenta con multitud de herramientas que permiten manipular los elementos de cada escena al antojo del diseñador, pudiendo incorporar herramientas adicionales a través de la *Asset Store* y contando con soporte multiplataforma (*Windows, Mac OS, IOS, Android, Linux*, etc.).

En la Figura 6 se muestra un ejemplo de la interfaz de *Unity* con un proyecto abierto donde se puede ver cómo la interfaz se divide en ventanas de trabajo. En ella se observan la ventana de jerarquía (*Hierarchy*) donde se organizan y añaden los objetos de la escena, la ventana de proyecto (*Project*) donde se muestra un explorador de archivos con todos las carpetas y ficheros del proyecto, las ventanas de *Game* y *Scene* para visualizar el juego en ejecución y en modo edición respectivamente, las ventanas de *Console* (donde se muestra información de depuración de código) *Animation* y *Animator* (relacionadas con edición de animaciones) y finalmente la ventana de *Inspector*, donde se visualizan las propiedades de un determinado objeto o fichero, así como el valor de las variables públicas de un *script*.

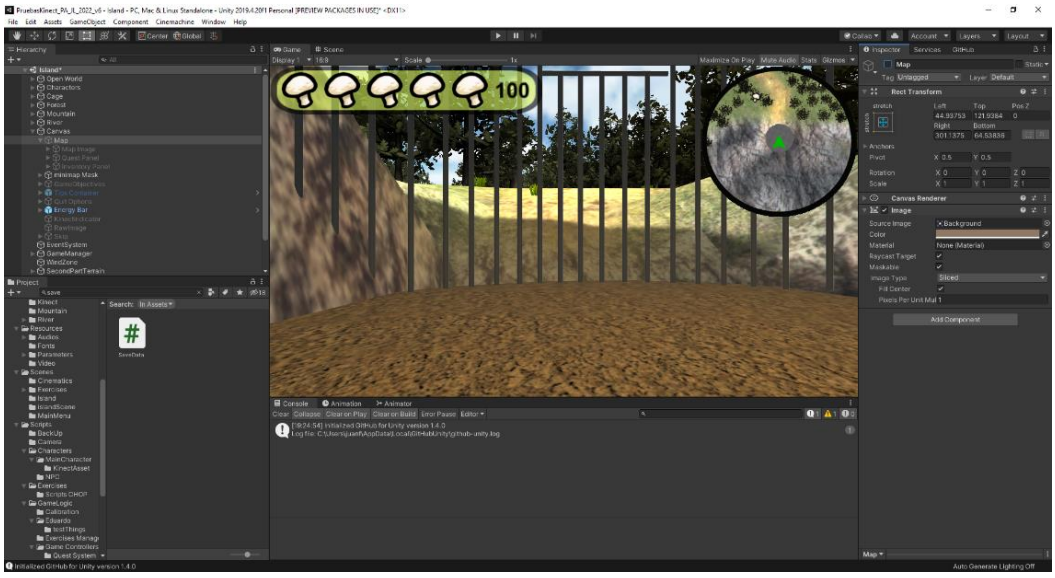


Figura 6. Ejemplo de la interfaz de Unity con una distribución de ventanas personalizada.

Unity se complementa con el editor de código fuente desarrollado por *Microsoft* llamado *Visual Studio Code* [12] que incluye soporte de depuración para la edición de los *scripts* integrados en el proyecto de *Unity* en lenguaje C#, derivado de C/C++. En el editor es donde se crean y editan los scripts que serán usados en *Unity*, tal como se muestra en la Figura 7 a modo de ejemplo.

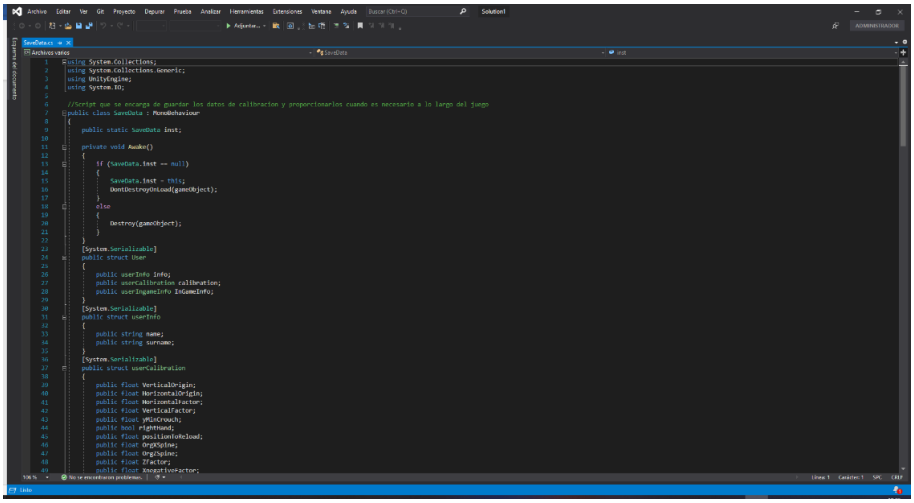


Figura 7. Ejemplo de la interfaz de *Visual Studio Code*.

También se ha hecho uso de la herramienta de repositorios de código *GitHub* [13] pensado para el desarrollo de aplicaciones en el que participan varios programadores para la actualización de scripts. Durante las pruebas realizadas, se comprobó que esta herramienta presentaba algunas incompatibilidades con este proyecto impidiendo el uso del *middleware* y el dispositivo *Kinect*. En el Anexo II se dan más detalles de cómo se ha tratado de integrar esta herramienta y se dan más especificaciones de su uso.

2.3. Estructura lógica de *Phiby's Adventure 3D*

En este apartado se dará una explicación de cómo venía desarrollándose la estructura lógica en *Phiby's Adventures 3D*. Esto implica hablar de los diferentes eventos que se producen en el juego, de cómo se dan paso los unos a los otros y de quienes son los agentes que se encargan de administrar dichos eventos.

Esta parte del videojuego fue programado por varios desarrolladores y ha ido creciendo de tal modo que se han llevado a cabo múltiples cambios en los diferentes métodos. Debido a esto, la lógica presenta una estructura compleja que contiene partes obsoletas o partes redundantes. En los siguientes apartados, se describe la estructura que tenía el juego al inicio del proyecto indicando los problemas detectados y, a partir del apartado 4, las soluciones que se han ido incorporando a lo largo del proyecto.

Para que la información de los siguientes apartados sea más sencilla de seguir, desde este punto en adelante se hará referencia a los objetos con la nomenclatura y formato *NombreDelObjeto* y a las clases o *scripts* con `NombreDeLaClase.cs`. El nombre del objeto no tiene por qué coincidir con el nombre del *script* que está contenido en ellos, bien porque el objeto contiene varios *scripts* o porque con el paso del tiempo su funcionalidad ha cambiado. Aunque se ha intentado que ambos nombres coincidan para evitar confusiones, en general, cuando se esté hablando de las funciones de un objeto, se hará referencia a sus *scripts*. Este caso ocurre con frecuencia al hablar del objeto

DataManager y la clase `GameState.cs`, la cual es la clase más importante para la progresión del jugador y está contenida dentro del objeto *DataManager*.

Es importante hacer esta diferenciación dado que puede haber varias instancias de un objeto en una misma escena (habiendo, por tanto, varias instancias de sus clases). Para que una clase acceda a las variables y métodos públicos de otra clase que esté contenida en otro objeto, es importante tener en cuenta si se está accediendo a la instancia específica que se está buscando (lo cual se puede hacer de diferentes maneras). En el apartado 4 se dan más detalles de esto al detallar los cambios que se han hecho en cada *script*.

2.3.1. *Sistema de progreso y checkpoints*

El progreso del personaje principal (*Phiby*) se controla a través de un sistema de *checkpoints*. Cuando el jugador (al cual, a partir de ahora, se denominará como *Phiby*, usuario o directamente como jugador) alcanza un cierto objetivo, el sistema debe recordar la información del juego en ese instante de la partida, de modo que, si se quiere modificar el resto de la información relevante, o simplemente se desea continuar la partida en otro momento, basta con comprobar cuál es el último *checkpoint* alcanzado por el jugador y actualizar el resto de las variables en consonancia. Tanto la historia, como el resto de los elementos lógicos que se desarrollan en cada escena dependen del orden cronológico en que se alcanza cada *checkpoint*. Se ha tratado de mantener el esquema dado por Haoru Qin [8], salvo por el *checkpoint* 4 donde, en la nueva versión, se ha incorporado un nuevo ejercicio diseñado por Laura Álvaro Gil. A continuación, se da un listado de los diferentes *checkpoints*.

- *Checkpoint* 0: Inicio del juego. *Phiby* se instancia en la escena *Island* dentro de una jaula.
- *Checkpoint* 1: El jugador completa el ejercicio *Escape from the cage* y es instanciado fuera de la jaula.
- *Checkpoint* 2: El jugador encuentra al NPC *Granny* y este le marca el objetivo de rellenar un bol con manzanas, teniendo que completar el ejercicio *Apple tree*.
- *Checkpoint* 3: *Phiby* se dirige al puente roto, con o sin el objeto cuerda en su inventario. Si no lo tiene permanece en este *checkpoint* hasta que lo encuentre.
- *Checkpoint* 4: Si el jugador tiene la cuerda, será enviado a buscar al NPC *Rocky*, teniendo que completar en el camino el ejercicio *Move rocks* incorporado por Laura.
- *Checkpoint* 5: *Phiby* ha liberado a *Rocky* y debe dirigirse hacia el puente para lanzar la cuerda que ha recogido previamente cerca del bosque.

- Checkpoint 6: *Phiby* deberá completar el ejercicio *Repair bridge* si tiene el número de troncos necesario. Si no, deberá realizar el ejercicio *Chop wood* para recolectar los troncos y volver al puente para completar *Repair bridge*.

Aunque el desarrollo de *Phiby's Adventure* contempla la adición de nuevos *checkpoints* organizados en forma de capítulos, y algunos de ellos se han empezado a implementar en otros PFG y prácticas, en este proyecto únicamente se ha trabajado con los 5 primeros de la lista anterior. A nivel de código, los *checkpoints* están representados por la variable entera pública llamada `checkpoint`, que está definida en la clase `GameState.cs` (ver apartado 2.3.2 Control de la lógica y clases principales, *GameState*) debiendo acceder a esta clase para modificar el valor de los *checkpoints*.

2.3.2. Control de la lógica y clases principales

El desarrollo lógico del juego viene administrado por tres tipos de clases. En primer lugar, una clase principal que se ocupa de almacenar los datos globales que son relevantes para la mayoría de los objetos o *scripts* durante la partida (en este caso será `GameState.cs`). En segundo lugar, se encuentran clases que se encargan de gestionar comportamientos que afectan a una o varias instancias de un objeto (en general se denominarán *managers*). Por último, se hablará de clases que ejecutan tareas específicas a partir de la información disponible, pudiendo haber varias instancias del mismo objeto simultáneamente (que normalmente recibirán el nombre de *controllers*). Esta distinción resulta de gran importancia para entender cómo se desarrolla la progresión del jugador en la historia a través de los *checkpoints*, ya que estas clases inician los diferentes eventos en función del *checkpoint* en el que se encuentre el jugador en cada instante de la partida. En el diagrama de la Figura 8 se muestra cuál era el flujo de ejecución de las principales clases involucradas en el juego recibido al inicio de este proyecto, qué *scripts* lo componen y qué información transmite.

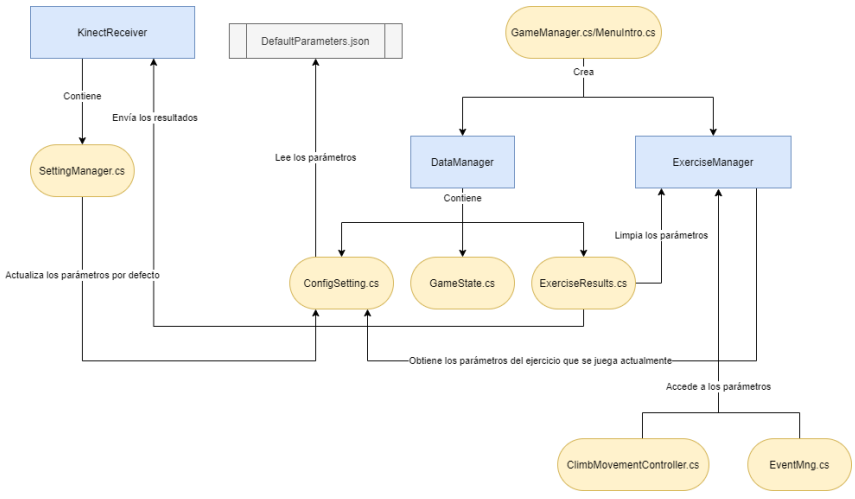


Figura 8. Diagrama de flujo de la estructura de las diferentes clases, extraído del *Game Design Document* v2.5 [13].

En un inicio, las clases `GameManager.cs` y `MenuIntro.cs` (dependiendo desde qué escena se iniciase la ejecución del juego) creaban el objeto *DataManager* que contiene a las clases `GameState.cs`, `ConfigSetting.cs` y `ExerciseResults.cs`, y el objeto *ExerciseManager* que contiene a la clase `ExerciseManager.cs`, aplicando la implementación que se muestra en la figura 9. *DataManager* era el objeto que contenía la información asociada al estado del juego y a la configuración de los ejercicios, mientras que *ExerciseManager* era el objeto que administraba la información y los parámetros del ejercicio que se va a jugar en ese momento. Sin embargo, algunas de estas clases, como `ExerciseResults.cs` o `ExerciseManager.cs` o `ConfigSetting.cs`, se han quedado obsoletas y su funcionamiento se ha vuelto difícil de entender con el paso del tiempo, por lo que esta configuración de clases y objetos se ha modificado, eliminado o sustituido por nuevas clases más simples, añadiendo la posibilidad de ejecutar el juego desde cualquier escena de modo que la realización de pruebas sea un proceso más ágil para los desarrolladores (ver apartado 4.1).

```
//TODO: Remove after completion of the game
private void Awake()
{
    /* It is called the CreateGameSetting method. */
    //Henar 26/04/2021 Here we create the DataManager
    if (GameObject.Find("DataManager") == null)
    {
        GameObject empty = new GameObject("DataManager");
        empty.AddComponent(typeof(ConfigSetting));
        empty.AddComponent(typeof(GameState));
        empty.AddComponent(typeof(ExerciseResults));
        DontDestroyOnLoad(empty);
    }
    if (GameObject.Find("ExerciseManager") == null)
    {
        GameObject empty = new GameObject("ExerciseManager");
        empty.AddComponent(typeof(exerciseManager));
        DontDestroyOnLoad(empty);
    }
}
```

Figura 9. Creación de los objetos *DataManager* y *ExerciseManager* en la versión inicial del proyecto.

Aunque en un inicio no había restricciones en cuanto al acceso a ciertas clases, en este proyecto se ha tratado de limitar este acceso a *scripts* y a variables importantes, de modo que se tenga un mayor control de todo lo que sucede en cada *frame* durante la ejecución del juego. Que muchas clases accedan a una misma variable puede dificultar el proceso de encontrar errores, alargando la implementación de otras características del videojuego. Este problema se ha encontrado a la hora de instanciar a *Phiby* según el valor de la variable pública `checkpoint`. Sobre este problema en concreto y su solución se hablará más adelante, en el apartado 4.2.

A partir de este punto, se da una explicación más detallada de cuáles son las características de las clases encargadas de la gestión de escenas y el progreso en la historia.

- **GameState**

`GameState.cs` es la clase encargada de administrar los datos globales del juego. Es decir, todas las variables que afectan al personaje principal o que determinan su progresión se encuentran en este *script*. Sus variables son públicas para otras clases (generalmente las clases *managers*) pudiendo modificarlas conforme sucedan los eventos del juego en cada escena o para poder administrar el comportamiento de varios objetos.

El objeto que contiene a la clase `GameState.cs` debe ser un objeto persistente, es decir, que no se destruya cuando el jugador pasa de una escena a otra. Para hacer esto, en anteriores versiones se utilizó el método `DontDestroyOnLoad()[14]`, el cual crea una escena *DontDestroyOnLoad* (que puede verse en la ventana de *Hierarchy* de la figura 10) donde los objetos contenidos en ella no se destruyen. En este proyecto se ha modificado la manera en la que esto se hace para ajustarse a la nueva estructura de objetos y clases.

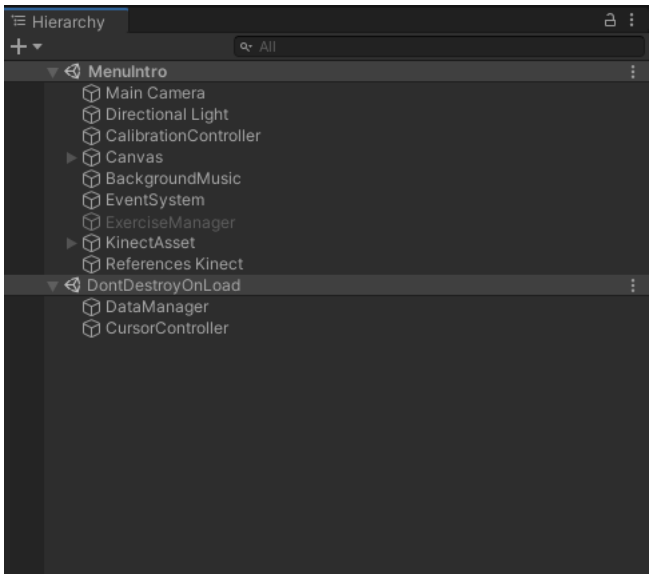


Figura 10. Ejemplo de escena *DontDestroyOnLoad* creada durante la ejecución del juego en la ventana *Hierarchy*.

El resto de los cambios están relacionados con la eliminación de variables obsoletas y la adición de algunas variables que se requieren en los nuevos sistemas que se han incorporado (que se explicarán en sus correspondientes apartados). Para comprender mejor esto último se añade un listado de las variables de `GameState.cs` en las que más se ha centrado la atención para este PFG y que se han mantenido para la versión actual del juego.

- Variables relacionadas con la progresión:
    - `public int checkpoint:` indica el valor del *checkpoint* en el que se encuentra el jugador. Es la variable más importante en términos de progreso del usuario.
    - `public int totalApplesNeed:` indica el número de manzanas necesarias para llenar el bol y poder pasar del *checkpoint* 2 al 3.
    - `public int bowlApples:` indica el número de manzanas que hay dentro del bol.
    - `public bool ropeCollected:` indica si se ha recogido la cuerda necesaria para pasar del *checkpoint* 3 al 4.
    - `public bool axeCollected:` indica si se ha recogido el hacha que se requiere para poder completar el ejercicio *Chop Wood*.
- Variables relacionadas con información del jugador:
    - `public int totalNumApples:` indica el número de manzanas recogidas por el jugador
    - `public int numLogs:` indica el número de troncos de madera recogidos por el jugador
    - `public float onScreenEnergy:` valor de la energía del jugador. Este valor se muestra en pantalla
    - `public Vector3 positionGame:` valor de la posición de *Phiby*. Se utiliza para instanciar a *Phiby* y para recordar su última posición en la isla.
    - `public Quaternion rotationGame:` valor de la rotación de *Phiby*. Se utiliza para instanciar a *Phiby* y para recordar su última posición en la isla.
- Variables de utilidad:
    - `public bool isKinect:` indica si el modo *Kinect* está activado o no. Su valor cambia al pulsar la letra K. Si es *true*, el modo *Kinect* está activado.
    - `public bool NoEnter:` indica si se puede entrar en un ejercicio. Se utiliza para bloquear el cambio de escenas al volver de un ejercicio. Se pone a *true* al regresar de un ejercicio y a *false* al salir de los *colliders* que activan el cambio de escena.

El resto de las variables están asociadas a otras características de menos relevancia para este PFG, ligadas con la realización de ejercicios, información de la web, etc.

- **Managers**

  Un *manager* se encarga de controlar comportamientos que implican a un grupo de clases u objetos. Por ejemplo, puede ser una clase encargada de determinar cómo y cuándo suceden los eventos dentro de una escena, implicando así a otros objetos y clases de esa escena. Si bien puede plantearse un diseño común para todos ellos, en *Phiby's Adventure 3D*, se diseñó un *manager* para cada escena, con funciones algo diferentes según el tipo de escena y sus características, ayudando a simplificar el código en algunos de los *managers*, pero dotando de muchas funciones a otros.

  Los *managers* que se diseñaron para administrar las escenas del videojuego fueron: `MenuIntro.cs` para la escena del menú principal, `GameManager.cs` para la escena de la isla y las clases `MiniGameManager.cs` y `EventMng.cs` para las escenas de ejercicios (cada clase se usa dependiendo de si está utilizando el modo teclado o el modo *Kinect*). Estas cuatro clases se conectan a través de la variable `checkpoint` de `GameState.cs` y de los *controllers* de sus escenas, que son quienes cargan una escena u otra a medida que el jugador avanza en la historia.

  Para dar más contexto, se va a dar una descripción de estas 4 clases explicando su funcionalidad junto a un listado de los métodos más relevantes para el proyecto y sobre los cuales se ha trabajado (ver apartado 4.1.2).

  - `MenuIntro.cs`: Es la clase encargada de la gestión de los eventos que se producen en la escena del menú principal del juego, lo que en este caso se traduce en mostrar los paneles del objeto *Canvas* (objeto donde se instancia esta clase) y que contiene las opciones del menú. Los principales métodos de esta clase son los relacionados con los botones del menú y dependiendo de la opción seleccionada, este *script* actualiza, carga o muestra la información correspondiente.
    - `public void StartAdventure()`: Inicializa las principales variables del *DataManager* y carga la escena de la isla desde el *checkpoint* 0.
    - `public void SelectCpToStart(int checkpoint)`: Inicia la partida desde el *checkpoint* pasado como parámetro.
    - `public void ContinueAdventure()`: Continua una partida cargada desde fichero.
    - `public void Quit()`: Finaliza la aplicación.

- `public void CopyToClipboard()`: Copia la ruta donde se guarda el fichero.
- `public void GetGoal(string text)`: Devuelve la cantidad introducida en el *Input field*.
- `public void SetGoal()`: Establece el objetivo marcado por el usuario en el menú.
- `public void RandomGoal()`: Establece una cantidad aleatoria como objetivo.
- `public void ControlOption()`: Establece el modo de control (*Kinect* o teclado)

- `GameManager.cs`: Es la clase que administra los eventos de la escena *Island*. En esta escena es donde ocurre la mayor parte de los eventos importantes para el progreso del jugador, por lo que es el *script* más extenso de todos. Se encarga de tareas que van desde la instanciación del jugador en la isla, la gestión de eventos de teclado o mostrar consejos que sirvan de orientación para el usuario; hasta administrar cinemáticas o mostrar información en la UI (*User Interface*). Respecto a la instanciación del personaje principal en particular, en la versión recibida al inicio de este PFG, `GameManager.cs` posicionaba incorrectamente a Phiby al cambiar de escena durante del juego, error que se trata en el apartado 4.2.
  - `public void SetCheckPoint(int cp)`: establece el valor de la variable `checkpoint` del *DataManager* al valor pasado como parámetro.
  - `public int GetCheckPoint()`: devuelve el valor de la variable `checkpoint` de `GameManager.cs`
  - `private void SwitchPhiby()`: permite intercambiar el control del usuario entre *Kinect* y teclado.
  - `private void SetLocation()`: fija la posición de Phiby en la escena *Island* dependiendo del valor de la variable `checkpoint`.

- `MiniGameManager.cs`: Es la clase instanciada cuando se entraba en un ejercicio, pero estando en modo teclado (modo por defecto al iniciar el juego). La escena correspondiente cargada es *MiniGamesSimulator*. En ella, `MiniGameManager.cs` muestra en formato de texto las instrucciones necesarias para completar el ejercicio dependiendo de cuál de ellos se esté simulando.

- `EventMng.cs`: Controla la entrada en las escenas de ejercicios (*ClimbCage*, *AppleTrees*, *MoveRocks*, etc.), se instanciaba si se había activado el modo de control *Kinect* mediante la letra K. Esta clase se encargaba de instanciar a *Phiby* en la posición adecuada del ejercicio correspondiente según la variable `checkpoint` y finalizando el ejercicio si este se completaba o pasado un cierto tiempo.

En la solución propuesta, estas últimas dos últimas clases se han unificado en una sola para hacer menos confuso el seguimiento de variables importantes para la progresión del jugador y, además, porque comparten algunas funcionalidades como, por ejemplo, las de los métodos `public IEnumerator ExitMiniGame()` de `MiniGameManager.cs` y `public void exitScene()` de `EventMng.cs` que se encargan de volver a la escena *Island* (ver apartado 4.1.3) .

- **Controllers**

Un *controller* normalmente desempeña una tarea concreta para un cierto tipo de objetos. Puede haber varias instancias del mismo objeto, por lo que todos los objetos tendrán el mismo comportamiento, siendo su *manager* correspondiente el encargado de determinar cómo interactúa cada instancia con el resto del entorno. Entre sus funciones suele estar el movimiento de los objetos o el comportamiento de NPCs.

Como se mencionaba al inicio del apartado, una de las dificultades encontradas para poder llevar un seguimiento de qué ocurría en cada frame durante la ejecución del juego, es que algunos de estos *controllers* accedían de manera directa a las variables de la clase `GameState.cs`. Si a esto se le suma el hecho de que existían declaraciones redundantes de algunas variables (como era el caso de la variable `checkpoint`, que estaba definida en `GameState.cs` y en `GameManager.cs`) puede convertirse en un problema si no se trata con cuidado ya que, si muchos objetos dependen de una misma variable que, a su vez, puede ser modificada por todos ellos y está declarada en dos *scripts* diferentes, pueden producirse errores en el comportamiento de estos objetos. En la solución a este problema (ver apartados 4.1.2 y 4.1.3) se ha tratado de limitar la cantidad de objetos que acceden directamente a `GameState.cs`, utilizando de intermediario los *managers* de cada escena.

A continuación, se listan algunos de los *controllers* ya existentes sobre los que se ha trabajado y se explica cuál era su funcionalidad al inicio del proyecto.

- `PhibyMixedController.cs`: Es la clase encargada de administrar el movimiento de *Phiby*, dependiendo del modo de control activo (mediante la variable `isKinect`). Se ayuda de la clase `PhibySwitch.cs` que fundamentalmente informa a otras clases del modo de control que está activo.
- `EnergyController.cs`: Determina la energía que pierde el jugador en cada movimiento, tanto al desplazarse por la isla como en cada ejercicio. Actualmente se ha actualizado esta clase para incluir la funcionalidad que anteriores versiones se dividía en dos (`EnergyController.cs` y `MiniGameEnergyController.cs`).
- `Bowl.cs`: Define el estado del *bowl* (de vacío a lleno) para que `GameManager.cs` determine si se ha completado el objetivo marcado por el NPC *Granny*.
- `ShortBars.cs`: Controla el *collider* de las barras de la jaula que carga la escena del ejercicio *Escape from the cage* dependiendo de si *Phiby* ha recogido los lotes de paja.
- `Grandma.cs`: Controla la interacción del NPC *Granny* con *Phiby*. En un inicio este personaje controlaba el paso del *checkpoint* 1 al 2 y del 2 al 3.
- `Rocky.cs`: Controla la interacción del NPC *Rocky* con *Phiby* y su movimiento por la isla en el *checkpoint* 5.

En el apartado 4.1.4 se indicarán cuáles han sido los cambios que se han realizado sobre algunos de los *controllers* de esta lista y por qué ha sido necesario realizar estos cambios.

3. Objetivos del proyecto

Como se mencionaba en la introducción, el primer objetivo del proyecto es el de mejorar y rediseñar la estructura lógica y de progresión del videojuego. El avance del jugador se determina por un conjunto de *checkpoints* que determinan el punto de la historia que el jugador ha alcanzado en su partida. A medida que se alcanza un *checkpoint*, el juego actualiza y almacena la información que se tiene en ese momento de los datos que son más importantes para el progreso del jugador.

En este proyecto no se pretende cambiar radicalmente este sistema, si no optimizar o hacer más entendibles los elementos de código que están involucrados, directa o indirectamente, en el uso de los *checkpoints*.

Por otro lado, aunque se da la posibilidad al jugador de moverse libremente por el escenario, los *checkpoints* hacen que el progreso del juego sea muy lineal (ir de punto A a punto B, de punto B a punto C y así sucesivamente). Por esta razón, el segundo objetivo es crear una herramienta que permita implementar un progreso más flexible y no lineal. La herramienta escogida para llevar esto a cabo se conoce en el mundo de los videojuegos como **sistema de misiones**.

Un sistema de misiones consiste en una lista de tareas u objetivos a completar por parte del jugador que se van asignando a medida que este avanza en la historia o por el entorno, pudiendo entregar recompensas o premios por completar dichos objetivos. El enfoque con el que se quiere desarrollar esta herramienta es para que sirva como una nueva forma de progresión y jugabilidad que en futuras versiones del proyecto premie al jugador por completar las misiones (como monedas, puntos, objetos, etc.).

Por último, se pretende añadir otras dos características que complementen al sistema de misiones:

- La primera de ellas es una **Interfaz de Mapa** que sirva por un lado para darle al jugador una vista general de cuál es el terreno por el que se puede mover y, por otro, para informarle de las misiones que tiene que completar y en qué parte del terreno se completan.
- La segunda es el **sistema de control mediante gestos**, que permite al usuario moverse por algunos de los menús del juego a distancia, incluyendo la Interfaz de Mapa. Esto se hará moviendo el cursor utilizando del dispositivo *Kinect One*, por lo que, además, se pretende añadir la posibilidad de calibrar el desplazamiento del cursor dependiendo del grado de movilidad usuario. Este sistema de calibración se basa en funciones ya creadas por Eduardo Botija para el proyecto *Space Gun* [2] y que serán útiles para los ejercicios del videojuego.

En resumen, estas nuevas implementaciones (Sistema de misiones, Interfaz de Mapa y sistema de control mediante gestos) junto a las mejoras en el código, pretenden hacer más sencillo el desarrollo del videojuego, y más gratificante la experiencia del usuario.

4. Desarrollo y resultados

A continuación, se presentan los cambios realizados para esta nueva versión el videojuego con el fin de conseguir los objetivos planteados anteriormente, solucionando errores existentes en el juego y añadiendo nuevas funcionalidades.

En este apartado se comenta por qué se han implementado estas soluciones, las dificultades encontradas y los resultados que se han obtenido. En el apartado 4.1 se presentan los cambios hechos en los agentes lógicos del juego que determinan el progreso del jugador, desde cuál es la estructura hasta cambios realizados en cada uno de ellos para mejorar su comportamiento. En el apartado 4.2 se exponen los nuevos sistemas que se han incorporado al proyecto, cuál ha sido el proceso de su desarrollo y cuál ha sido el resultado conseguido. Por último, en el apartado 4.3, se comentan cambios menores que se han realizado en el juego para mejorar algunas características.

4.1. Rediseño de la estructura lógica

Partiendo de la estructura inicial, explicada en el apartado 2.3.2, se han modificado algunos de los elementos que formaban parte de ella. Se va a dar una explicación sobre qué elementos se han cambiado, cuáles han sido esos cambios respecto de la estructura inicial y por qué se han hecho. Después se comentan cuáles han sido los cambios específicos que se han realizado en cada uno de los agentes, tanto aspectos relacionados con la comunicación que existe entre ellos, como cambios relevantes para la progresión del usuario y cambios menores realizados para optimizar su comportamiento.

4.1.1. Reestructuración de los elementos involucrados

En este primer punto sobre el rediseño de la estructura de la lógica, se introducen los objetos más importantes, sus objetivos y los cambios que han sufrido respecto a la estructura inicial. Estos objetos son:

- *DataManager*
- Los objetos que contienen a los *managers* de cada escena (*GameManager*, *KinectGamesManager* y el *Canvas* del menú principal)
- *KinectReceiver*
- *CalibrationManager* añadido para este PFG para el sistema de calibración

Hay que mencionar que en este punto no se ha incorporado información sobre los *controllers* u otros objetos que no intervengan de manera directa en la nueva estructura que sustituye a la que había en versiones anteriores. Se da por hecho que estos objetos se comunican o bien con uno de los tres *managers* mencionados o con otros objetos de su misma escena.

A continuación, se aporta un listado de las funciones de cada objeto junto al diagrama de la nueva estructura (figura 11) donde quedan representados.

- *DataManager* sigue teniendo la función de manejar la información global del juego, hablando tanto de datos relevantes para la progresión del jugador en la partida, como de datos relevantes recibidos de o enviados a la plataforma *Blexer-med*, en relación con los ejercicios.
- Los objetos de los *managers* de escenas (*Canvas*, *GameManager* y *KinectGamesManager*) se encargan de acceder a los datos globales del juego contenidos en `GameState.cs` para poder activar los eventos que suceden en su correspondiente escena. En el caso particular de *KinectGamesManager*, también se necesita un acceso a los parámetros de la web para poder administrar los ejercicios adecuadamente.
- *KinectReceiver* es el objeto que establece una comunicación directa con el middleware en el envío de resultados y actualización de parámetros.
- *CalibrationManager* es un objeto cuyas clases son relevantes en el proceso de calibración que se ha incorporado en este proyecto y que se explica en detalle en el apartado 4.3.

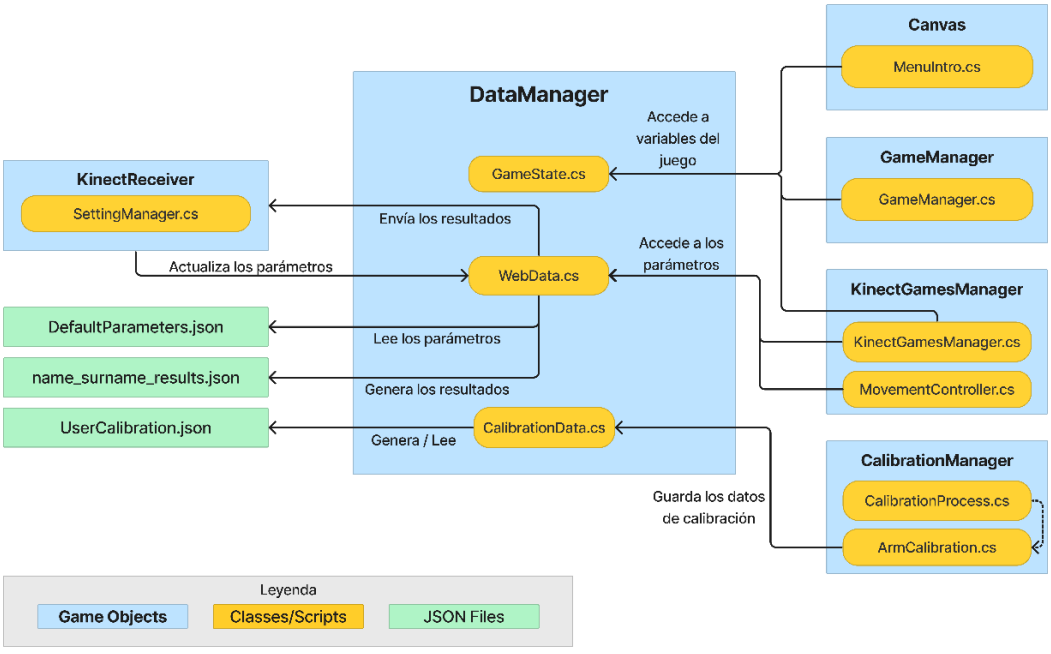


Figura 11. Diagrama de flujo de la nueva estructura de clases y objetos.

La nueva estructura gira en torno al objeto *DataManager*, que sigue conteniendo a la clase `GameState.cs`. Como se ha mencionado al principio del apartado 2.3, este objeto se comunicaba con el objeto *ExerciseManager* para enviarle información del ejercicio a realizar, utilizando las clases `ConfigSetting.cs` y `ExerciseResults.cs` que accedían

a la configuración de los parámetros de los ejercicios y envían los resultados obtenidos a la web (respectivamente). Con el fin de simplificar el proceso de acceso y envío de datos entre clases, se han sustituido las clases `ConfigSetting.cs`, `ExerciseResults.cs`, y `ExerciseManager.cs` por la clase `WebData.cs` (creado por Eduardo Botija [2]), que es la que se encarga ahora de enviar información sobre los parámetros y los resultados de los ejercicios a otras clases y al servidor web.

Además, a *DataManager* se le ha añadido la clase `CalibrationData.cs` para el guardado de calibraciones generadas en el menú de calibración, o para cargar una calibración previa (se trata en detalle en el apartado 4.3).

La incorporación de la clase `WebData.cs` implica que las clases que antes accedían a los *scripts* eliminados ahora accedan directamente a `WebData.cs`. Entre ellas están `KinectGamesManager.cs` (sustituto de `EventMng.cs`) la clase `MovementController.cs` (antes `ClimbMovementController.cs`) y la clase `SettingManager.cs` (sin cambios y contenida en el objeto `KinectReceiver.cs`).

El último cambio respecto a la estructura inicial es que el objeto *DataManager* ya no está creado por los *scripts* `MenuIntro.cs` y `GameManager.cs`. Esto permite a los desarrolladores trabajar en las escenas de ejercicios de manera más rápida sin tener que ejecutar el juego desde las escenas *MenuIntro* o *Island*.

4.1.2. Rediseño de GameState.cs

En cuanto al diseño de `GameState.cs` se han hecho cambios relativos a la persistencia de *DataManager* y cambios menores en la inicialización y definición de variables globales a las que acceden el resto de las clases. Se empieza hablando de la consecuencia de la modificación en la estructura del apartado anterior y después se comenta cuál es la situación de las variables en la versión actual de `GameState.cs`.

Para hacer persistente de nuevo a *DataManager*, se ha recurrido a modificar el código del *script* `GameState.cs` añadiendo el método `DontDestroyOnLoad()`. Sin embargo, esto no resultó ser suficiente, ya que al cambiar de escena se generaban dos objetos *DataManager* en la escena *DontDestroyOnLoad* (el ya existente y el de la nueva escena) debido a que *Unity* genera todo el contenido de la jerarquía al cargar una escena.

Para solucionar esto no resultó suficiente con comprobar si el objeto *DataManager* era nulo, por lo que se probaron dos estrategias dentro de la clase `GameState.cs`:

1. La primera estrategia que se probó se muestra en la figura 12. El código implementado busca todos los objetos que contengan una instancia de `GameState.cs` y destruye todos ellos, menos aquel que coincida consigo mismo (`this`). De este modo se evitan duplicados en la escena *DontDestroyOnLoad* y se consigue hacer persistente el objeto que contenga la clase `GameState.cs`, en este caso *DataManager*.

```
public void Awake()
{
    for (int i = 0; i < Object.FindObjectsOfType<GameState>().Length; i++)
    {
        if (Object.FindObjectsOfType<GameState>()[i] != this)
        {
            if (Object.FindObjectsOfType<GameState>()[i].name == gameObject.name)
            {
                Destroy(gameObject);
            }
        }
        else
        {
            DontDestroyOnLoad(gameObject);
        }
    }
}
```

Figura 12. Implementación del método *`DontDestroyOnLoad()`* en *`GameState.cs`*.

2. Una implementación más sencilla que se probó a raíz de estudiar las clases aportadas en [2], fue definir como *public* y *static* a una instancia de la clase `GameState.cs` y destruir cualquier objeto que no sea dicha instancia, tal y como se hace por ejemplo en la clase `WebData.cs` (figura 13). Esta implementación es útil para clases que van a estar contenidas en un solo objeto, lo cual presenta algunas ventajas ya que es más sencillo acceder a variables y métodos de la clase. Sin embargo, no se ha podido llevar a cabo ya que era necesario realizar modificaciones en demasiados *scripts* y, por cuestiones de tiempo se tuvo que abandonar esta estrategia.

```
public class WebData : MonoBehaviour
{
    public static WebData inst;
    private void Awake()
    {
        if(WebData.inst== null)
        {
            WebData.inst = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }
```

Figura 13. Ejemplo de uso de una instancia estática.

Un aspecto importante en `GameState.cs` está relacionado con la inicialización de las variables del juego. En los cambios realizados para esta versión del videojuego no se ha incluido una inicialización por código desde `GameState.cs` si no que se ha dejado que esto se haga a través de la herramienta *Inspector* con la intención de que sea más cómodo hacer pruebas en escenas distintas a la del menú principal.

El resto de los cambios hechos en `GameState.cs` están relacionados con la definición de variables, eliminando aquellas que estaban en desuso o que presentaban redundancias. Además, se ha organizado el código para que sea más legible y se pueda encontrar la información de manera más rápida. A continuación, se da un listado de cuáles han sido estas modificaciones:

- `ExitMiniG` ha sido eliminada ya que presentaba redundancias con `NoEnter` y había perdido su sentido.
- Las variables relacionadas con el objeto *Bowl* y su estado, han sido trasladadas a su propio script (`Bowl.cs`). Se da algún detalle más en el apartado 4.1.3.
- Se ha eliminado la variable `ChopWood` `chopWoodGS` por no tener ninguna funcionalidad en el juego.
- Se han nuevas variables de control útiles para los nuevos sistemas y características incorporados en el proyecto como son:
    - `public int lastSceneID`: identificador para la última escena cargada. Se utiliza para actualizar la lista de misiones al cargar el juego desde un determinado *checkpoint*.
    - `public bool AllowKinect`: permite el uso del modo *Kinect* si se ha establecido conexión con el K2UM.
    - `public bool backToMenu`: se utiliza al cargar un ejercicio desde el menú, haciendo regresar al usuario al menú principal una vez se ha completado el ejercicio.
    - `public List<Quest> questList`: es la lista que contiene todas las misiones del juego y su estado (si están completadas por el jugador o no). La lista de misiones se inicializa en el método `Start()` de `GameState.cs` como se muestra en la figura 14.

```
public void Start()
{
    Debug.Log(Application.persistentDataPath);
    //List of main quests (QuestTye, id, title, description, completed)
    questList.Add(new Quest(Quest.QuestType.Main, 0, "Get out of here! Part 1", "Find a way to get out of the cage.", false, 0));
    questList.Add(new Quest(Quest.QuestType.Main, 1, "Get out of here! Part 2", "Grab some straws and climb up the bars.", false, 0));
    questList.Add(new Quest(Quest.QuestType.Main, 2, "Meeting Grandma", "Find Grandma in her house.", false, 1));
    questList.Add(new Quest(Quest.QuestType.Main, 3, "Where's my key?", "Find Grandma's key in the forest.", false, 2));
    questList.Add(new Quest(Quest.QuestType.Main, 4, "Mmmm... Apples!", "Check the bowl inside the house.", false, 1));
    questList.Add(new Quest(Quest.QuestType.Main, 5, "Climbing the apple trees", "Take a look behind the hut, there is one big and two s
    questList.Add(new Quest(Quest.QuestType.Main, 6, "Fill the bowl", "Back to Grandma's house and put the apples in the bowl.", false,
    questList.Add(new Quest(Quest.QuestType.Main, 7, "The Bridge", "Go to the bridge.", false, 5));
    questList.Add(new Quest(Quest.QuestType.Main, 8, "Looking for a rope", "Find a rope near the forest to fix the bridge.", false, 5));
    questList.Add(new Quest(Quest.QuestType.Main, 9, "Find Rocky", "Find Rocky and bring him back to help you.", false, 4));
    questList.Add(new Quest(Quest.QuestType.Main, 10, "Free Rocky", "Help Rocky to escape from the collapsed cave.", false, 4));

    //List of side quests (QuestTye, id, title, description, completed)
    questList.Add(new Quest(Quest.QuestType.Side, 11, "More Apples!", "Repeat climbing trees to get more apples", false, 0));
```

Figura 14. Inicialización de la lista de misiones en el método `Start()` de `GameState.cs`.

4.1.3. Rediseño de los managers

En el caso de los *managers*, se han hecho varias modificaciones en `MenuIntro.cs` y `GameManager.cs`. Por otro lado, las clases `EventMng.cs` y `MiniGameManager.cs` se han combinado en una sola clase. Los objetivos de estos cambios son:

1. Mejorar y simplificar los métodos y variables que utilizan, en especial aquellos relacionados con los datos del juego y la clase `GameState.cs`.
2. Solucionar algunos errores encontrados durante el desarrollo del proyecto.
3. Incorporar nuevas funcionalidades útiles para el proyecto en general y para los nuevos sistemas que se van a incorporar en el apartado 4.2.

En este apartado se van a presentar los managers de cada tipo de escena en su orden de aparición del videojuego: menú principal, la isla y los ejercicios. En cada una de ellas se indican los cambios aplicados para tratar de conseguir los objetivos anteriores.

- `MenuIntro.cs`

En primer lugar, se han actualizado los métodos de los botones del menú para que modifiquen adecuadamente todas las variables asociadas al jugador al iniciar una partida desde el inicio, desde un *checkpoint* o al continuar una partida en curso. En algunos métodos como `SelectCpToStart()`, que inicia el juego desde el *checkpoint* seleccionado, faltaba por actualizar valores importantes como la posición de *Phiby* para instanciar al personaje correctamente.

Se han eliminado los métodos `ShowPath()` y `SetSpecialCP()` por desuso y se ha mejorado el método `SetGoal()`, que ahora toma correctamente el valor introducido en el *InputField* con ayuda del método `GetGoal()`, que lee el valor introducido. Además, se ha incorporado en el *script* el método `MiniGameSelection()` que carga la escena del ejercicio que se pasa como argumento como se muestra en la figura 15.

```
/* Juan Ignacio 06/06/2026
 * Description: Start the chosen minigame. When the exercise is completed, back to the menu
 */
public void MiniGameSelection(int mg_selected)
{
    rotation = FindObjectOfType<Rotation>();
    rotation.finishAndDestroy();

    gameState.isKinect = true;
    gameState.backToMenu = true;

    switch (mg_selected)
    {
        case 1:
            gameState.gameType = GameState.GameType.climbingCage;
            SceneManager.LoadScene("ClimbCage");
            break;

        case 2:
            gameState.gameType = GameState.GameType.climbing;
            SceneManager.LoadScene("ClimbAppleTree");
            break;

        case 3:
            gameState.gameType = GameState.GameType.rockElimination;
            SceneManager.LoadScene("RockyMinigame");
            break;
    }
}
```

Figura 15. Método *`MiniGameSelection()`*.

Se ha añadido un método `Update()` para permitir activar o desactivar el modo *Kinect* desde el menú presionando la letra K. También, como se mencionaba en el apartado 2.1.1, se ha eliminado la creación del objeto *DataManager* ya que ahora dicho objeto se encuentra en todas las escenas.

Por último, se ha añadido el método `finishAndDestroy()` de la clase `Rotation.cs`, en las funciones que cambian de escena para evitar errores de "búsqueda nula" relacionados con el objeto *KinectReceiver* añadido en la escena *MenuIntro* para el control del cursor. En la figura 16 se muestra un ejemplo con el método `StartAdventure()`.

```
/* Haoru Qin 03/04/2020
 * Description: Create new game
 * Modified at 03/04/2022 by Juan Ignacio: now it also updates most of game variables
 */
public void StartAdventure()
{
    rotation = FindObjectOfType<Rotation>();
    rotation.finishAndDestroy();
    gameState.checkpoint = 0;
    gameState.onScreenEnergy = 100;
    gameState.totalNumApples = 0;
    gameState.numLogs = 0;
    gameState.positionGame = new Vector3(1257f, 59f, 903f);
    gameState.rotationGame = new Quaternion(0, 0, 0, 0);
    SceneManager.LoadScene("Island");
}
```

Figura 16. Ejemplo de uso del método *`finishAndDestroy()`* para evitar errores con el objeto *KinectReceiver*.

- `GameManager.cs`

Respecto a la relación de `GameManager.cs` con el progreso del juego se han hecho actualizaciones en los métodos de los *checkpoints* `SetCheckpoint()` y `GetCheckpoint()` de modo que ahora se acceda y se modifique directamente la variable `checkpoint` de `GameState.cs`.

Con el cambio anterior se ha eliminado la variable `checkpoint` evitar redundancias con `GameState.cs`. También se ha eliminado una gran cantidad de variables que estaban en desuso o que no tenían ninguna funcionalidad aparente (`phibyLoc`, `appleTreePoint`, `inConversation`, `initializeComponent` y algunas constantes de tipo `string`).

Para mejorar el rendimiento de esta clase, se ha modificado la búsqueda de la mayoría de objetos para ser asignados a través del *Inspector* o para ser buscados por su etiqueta, reduciendo el uso el método `Find()`en la medida de lo posible. Esto se debe a que el método `Find()` es un método que conlleva multitud de inconvenientes en escenas con una gran cantidad de objetos. Según la documentación ofrecida por Unity [15] si el juego está ejecutando múltiples escenas, el método `Find()` busca en todas ellas el objeto en cuestión, generando un alto consumo de recursos, razón por la que recomiendan buscar el objeto mediante etiqueta o utilizar el *Inspector*. En la figura 17 se muestra el listado de etiquetas que se han añadido desde el inicio del proyecto (desde el *tag* 20 *GameManager* hasta el *tag* 26 *QuestController*, aunque se han utilizado algunos ya existentes por defecto en *Unity* como *Player*).

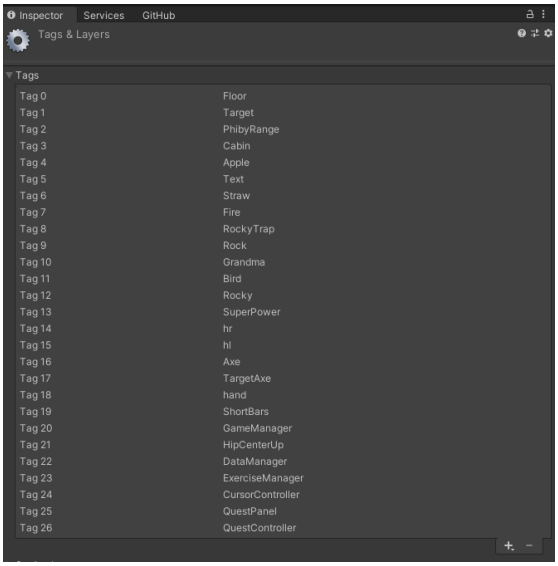


Figura 17. Lista de etiquetas creadas para *Phiby's Adventure*.

Se ha solucionado un error importante relacionado con el método `SetLocation()` (el cual se ha eliminado) que establecía incorrectamente la posición de *Phiby* cada vez que se cargaba la escena *Island*. El origen de este error se debe al incremento paulatino del

número de objetos y *scripts* creados en la escena *Island*, incluyendo al objeto *Main Character*. Dependiendo de la gestión de recursos que hiciera *Unity*, podían ocurrir dos situaciones expresadas en el diagrama de la figura 18.

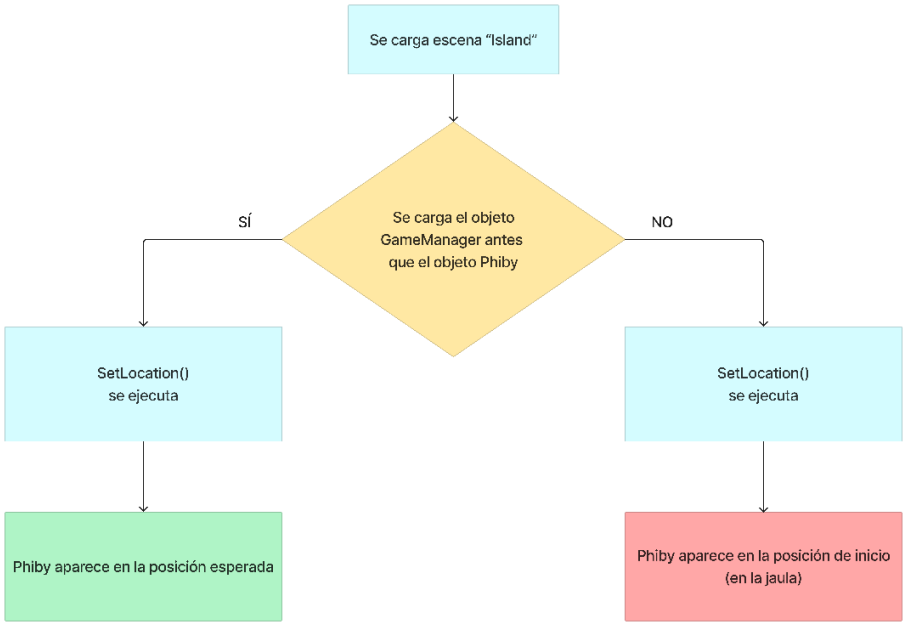


Figura 18. Diagrama de los posibles órdenes de ejecución de la escena Island.

Este problema fue uno de los primeros errores encontrados durante el proyecto y también fue el primero que se trató de corregir, recurriendo para ello al uso del método `Instantiate()` [16]. Este método coloca un *prefab* del objeto a instanciar tomado de la carpeta *Assets/Prefabs/Main Character* en la posición del escenario que se pasa como parámetro. De este modo se garantiza que la clase `GameManager.cs` se crea y sitúa siempre al objeto *Character* después de leer los parámetros adecuados de `positionGame` y `rotationGame` de `GameState.cs`. Aunque `Instantiate()` es una operación más costosa en términos de recursos, es preferible usar este método a que la posición del jugador dependa de la gestión de recursos que haga el software para cargar y ejecutar los elementos de una escena. En la figura 19 se muestra la función resultante que se ha implementado finalmente.

```
/* Juan Ignacio 30/3/2022
 * Description: Inizialices Phiby's prefab in the island, depending on the GameState position. This method replaces
 */
private void SpawnPhiby()
{
    if (gameState != null)
    {
        keyboardPhiby = GameObject.Instantiate(prefabPhiby, gameState.positionGame, gameState.rotationGame);
        cameraPhiby.FollowTarget(keyboardPhiby.transform.GetChild(0).gameObject.transform.GetChild(1).gameObject);
    }
}
```

Figura 19. Implementación del método *`Invoke()`* dentro del método *`SpawnPhiby()`* para instanciar a Phiby en la isla.

También se probó la herramienta de *Unity* ***Script Execution Order*** [17], que permite priorizar el orden de ejecución de los *scripts*. Esta herramienta no ha sido útil para solucionar el problema de la posición de *Phiby*, ya que se trata de un problema de generación de los objetos en una determinada escena y no un problema del flujo de ejecución de los scripts. Sin embargo, sí ha sido útil para permitir un correcto funcionamiento del juego a la hora de comenzar la ejecución desde otras escenas distintas al menú principal, utilizando la configuración que se muestra en la figura 20.

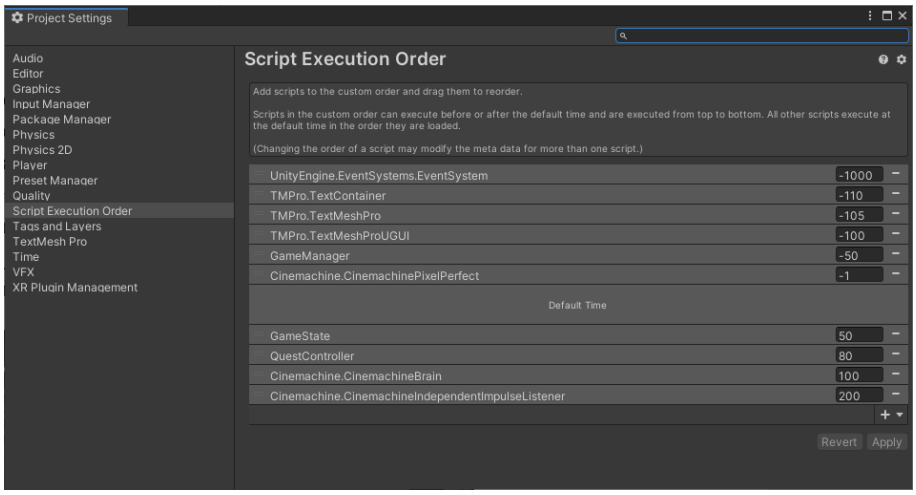


Figura 20. Configuración de la ventana *Script Execution Order* del menú de configuración del proyecto de Unity utilizada.

- `KinectGamesManager.cs`

`KinectGamesManager.cs` es el resultado de combinar las clases `EventMng.cs` y `MinigamesManager.cs`. Con este *script* se pretende simplificar la tarea de comprender qué ocurre en cada instante en las escenas de ejercicios/simulación en el futuro, además de unificar funciones comunes. Para juntar estos dos scripts se han realizado los siguientes cambios:

- Se ha dividido todos sus métodos en dos regiones:
  - Una región destinada al uso de Kinect que se ejecuta si `isKinect` es `true`.
  - Otra región destinada al uso del juego con teclado que se ejecuta si `isKinect` es `false`.
- Se ha unificado el método `ExitMiniGame()` de modo que la única diferencia entre el uso de Kinect o teclado es la destrucción (o no) del objeto que contiene a la clase `Rotation.cs`
- Se ha mantenido el método que en `EventMng.cs` se llamaba `spawn()` y que ahora se llama `SpawnMiniGame()` con la misma funcionalidad: situar a Phiby en la

posición adecuada según el ejercicio que se vaya a realizar, necesario en las secuencias de ejercicios.

Si bien estos cambios no deberían mejorar el rendimiento, sí resulta más simple consultar este *script* y entender su comportamiento.

4.1.4. Rediseño de los controllers

La mayoría de los cambios hechos en los *controllers* han sido cambios en cuanto al uso de la variable `checkpoint` de la clase `GameState.cs`. Ahora, la mayoría de los *controllers* de la escena *Island* acceden a esta variable a través de la clase `GameManager.cs` y el método `GetCheckpoint()` y, en general, no modifican su valor salvo en contadas excepciones (como en la clase `RepairBridge.cs` que controla el *collider* del puente de la isla y que controla el paso de los *checkpoints* 3 al 4 y del 5 al 6). En este apartado se van a centrar los comentarios en los *controllers* donde se han realizado los cambios más significativos para el proyecto.

- `EnergyController.cs` y `MiniGameEnergyController.cs`.

Estos dos *scripts* se han unificado ya que su objetivo es el mismo: reducir la energía del jugador según este completa ejercicios o se mueve, e incrementarla cuando *Phiby* recoge los *mushrooms* repartidos por la isla. Si bien `MiniGameEnergyController.cs` no contempla el hecho de incrementar la energía, ya que este *script* actúa en los ejercicios (donde no se ofrece la posibilidad de recuperar la energía) no hay ninguna necesidad de crear dos clases diferentes que controlen el valor de la energía cuyo comportamiento es similar. Por tanto, se ha eliminado la clase `MiniGameEnergyController.cs` y se ha mantenido el *script* `EnergyController.cs` añadiéndolo en las escenas de ejercicios en el objeto *MiniGameEnergyBar* (como se muestra en la figura 21) que muestra en el UI la energía del jugador en cada instante. Hay que mencionar que también se ha eliminado el método `Reset()` de `EnergyController.cs` por desuso.

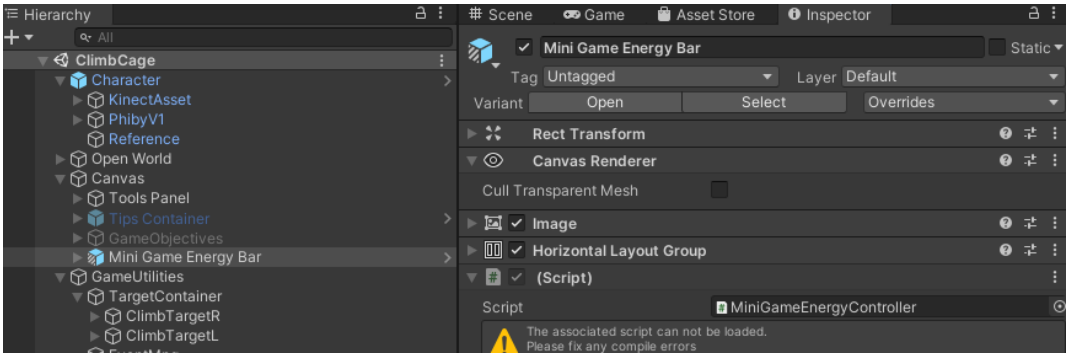


Figura 21. Captura de pantalla donde se muestra el objeto *MiniGameEnergyBar* que contiene el script `EnergyController.cs`

- `Bowl.cs`

En esta clase se ha incluido la variable `bowlState` que define el estado del bol para el NPC *Granny* (si está vacío, lleno o en un estado intermedio). Esta variable estaba definida en la clase `GameState.cs` y, si bien podría pensarse que es una información relevante para el estado del juego, lo cierto es que principalmente afecta al objeto *Bowl* para mostrar la animación en la que este se llena de manzanas y para informar al NPC *Granny* cuando esté lleno, razón por la cual se ha traído a esta clase.

- `PhibyMixedController.cs`

En este *script* se han incorporado dos gestos nuevos necesarios moverse por los diferentes menús, jugando con el modo *Kinect*: levantar el brazo izquierdo y el derecho.

Este *controller* registra los movimientos hechos en *Kinect* o las teclas pulsadas por el usuario para desplazar a *Phiby* por el terreno e interactuar con los ejercicios. Sin embargo, no se contaba con ninguna opción para acceder al menú de pausa desde *Kinect*, por lo que se ha añadido el gesto de levantar el brazo derecho por encima de la cabeza para activar o desactivar esta opción.

Dado que también es necesario mostrar la interfaz de mapa cuando el jugador lo desee (ver apartado 3.2.1) se ha añadido además el gesto de levantar la mano izquierda por encima de la cabeza para mostrar u ocultar esta interfaz (figura 22).

Puede verse la implementación de ambos gestos en el código de la figura 23 donde se accede a la clase `GameManager.cs` que contiene los métodos de `Pause()` y `ShowMap()`, que muestran el menú de pausa y la interfaz de mapa respectivamente. El motivo por el que estas funciones están en `GameManager.cs` es porque estas opciones son accesibles únicamente en la escena *Island* y es una manera de simplificar su implementación. En el apartado 5, se recomienda trasladar esto a un *manager* específico.

```
if (bones.SkeletonHead.transform.position.y < bones.SkeletonHandRight.transform.position.y && !stop)
{
    stop = true;
    GM.Pause();
}

if (bones.SkeletonHead.transform.position.y > bones.SkeletonHandRight.transform.position.y && stop)
{
    stop = false;
}

if (bones.SkeletonHead.transform.position.y < bones.SkeletonHandLeft.transform.position.y && !showMap)
{
    showMap = true;
    GM.ShowMap();
}

if (bones.SkeletonHead.transform.position.y > bones.SkeletonHandLeft.transform.position.y && showMap)
{
    showMap = false;
}
break;
```

Figura 22. Código de `PhibyMixedController.cs` de los gestos para mostrar/ocultar el mapa y pausar/continuar el juego.

```
public void Pause()
{
    pause = !pause;

    if (pause) //Stops the game
    {
        Time.timeScale = 0;
    }

    else //Plays the game
    {
        Time.timeScale = 1;
    }

    quitOptions.SetActive(pause); //Shows quit options panel depending on pause value
}
#endregion

#region Map and Quest System //Juan - 06/06/2022
/* Juan Ignacio 6/6/2022
 * Description: Shows or stop showing map interface
 */
public void ShowMap()
{
    showMap = !showMap;
    if (showMap)
    {
        if (!aS.isPlaying)
            aS.PlayOneShot(soundFX[0]); // Open map sound effect

        Map.SetActive(true);
        //Time.timeScale = 0;
    }

    else
    {
        if (!aS.isPlaying)
            aS.PlayOneShot(soundFX[1]); //Close map sound effect
        //Time.timeScale = 1;
        Map.SetActive(false);
    }

}
```

Figura 23. Código de `GameManager.cs` para pausar/continuar el juego y mostrar/ocultar el mapa.

4.2. Incorporación de nuevos sistemas

En este apartado se introducen nuevos sistemas que tratan de complementarse entre sí para ofrecer a futuros desarrolladores herramientas que ayuden a crear una experiencia más satisfactoria para el jugador a la hora de explorar el mundo e interactuar con el entorno. Hay que recalcar el hecho de que no se trata de sistemas completos y cerrados, si no que se trata de nuevas herramientas a disposición del diseñador, dejando la puerta abierta para que se amplíen y mejoren en futuras versiones.

4.2.1. Interfaz de mapa

En primer lugar, se propuso crear una interfaz nueva que aportase información al jugador del entorno por el que se mueve. La mayoría de los videojuegos modernos de mundo abierto y aventuras incorporan este sistema de diferente manera para guiar al jugador hacia donde tiene que ir.

Tomando como inspiración algunos de los títulos más populares de la industria, se ha tratado de diseñar una interfaz que se complemente con el sistema de misiones que se explica en el apartado 4.2.2, y que sea controlable tanto con *Kinect* como con teclado. Este apartado se centra en el proceso de creación del mapa de la isla y de su interfaz, tal y como se muestra en la figura 24.

Figura 24. Ejemplo de la distribución de la interfaz de mapa en una de sus últimas versiones, junto al resto de la UI.

En primer lugar, se buscó una manera de plasmar el terreno de la isla en una imagen 2D, probando con algunas posiciones de la cámara del editor de *Unity* y capturas de pantalla. Sin embargo, encontrar una posición adecuada de la cámara no resultaba fácil para obtener un buen plano a escala de la isla, por lo que se optó por extraer el mapa de alturas del terreno (figura 25).

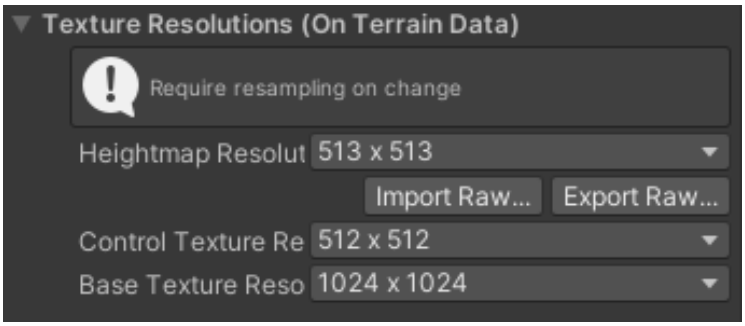


Figura 25. Ventana *Texture Resolutions* del terreno de la isla.

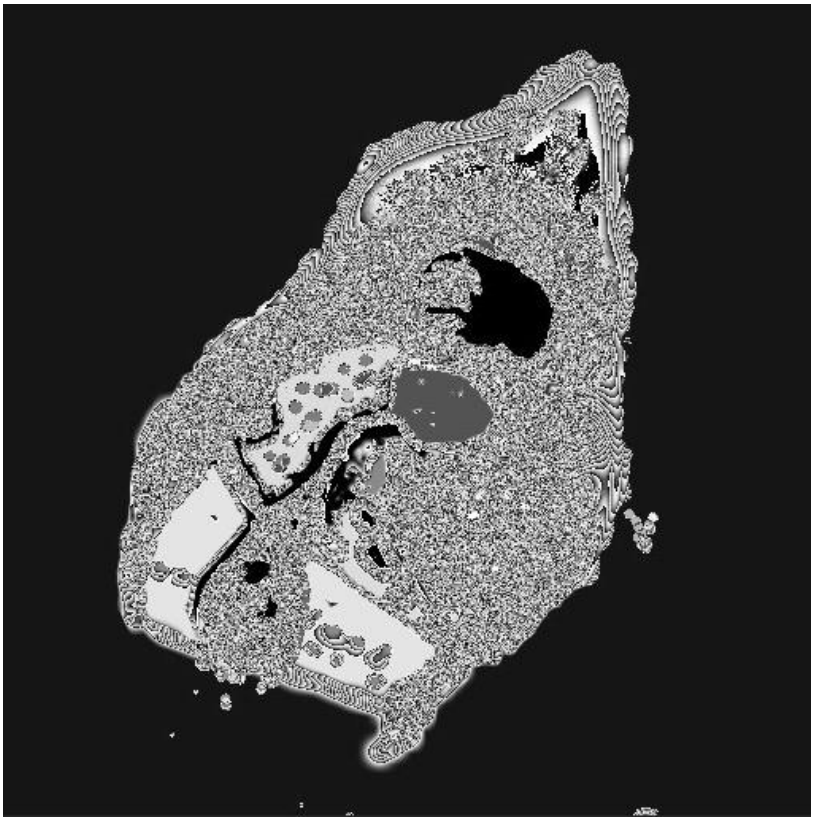
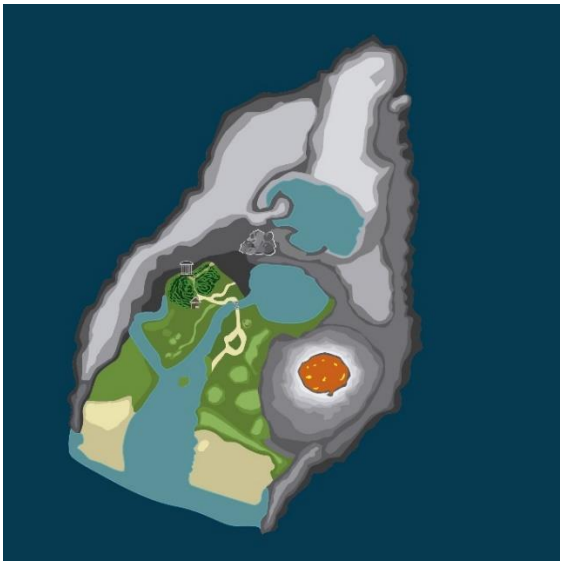


Figura 26. Mapa de alturas extraído del terreno del proyecto.

El mapa de alturas es accesible a través de los ajustes del terreno (*Terrain Settings*) en el apartado *Texture Resolutions*, pudiendo importar o, en este caso, exportar una imagen en formato *.raw* con una cierta resolución, tal y como se deja ver en la figura 25 mostrada anteriormente. Sin embargo, en la figura 26 se puede apreciar cómo el mapa de alturas resultó ser una imagen con mucho ruido en el terreno además de ser una imagen en blanco y negro. Por esta razón, en una primera versión del diseño del mapa, utilizando el software gratuito Paint, se hizo una versión a color del mapa, marcando lugares importantes y marcando los contornos y niveles del terreno (figura 27). Esta versión se utilizó como *placeholder* (un "sustituto temporal") del mapa y poder empezar a diseñar el resto de los elementos de la interfaz. La base de la interfaz ha consistido en un panel sobre el que se han situado otros tres paneles: uno para el mapa, otro para las misiones y otro para el inventario, tal y como se ha mostrado en la figura 24.

Figura 27. Primera versión a color del mapa de la isla.

Una de las ideas que se plantearon en un inicio era poder ampliar diferentes regiones del mapa para poder ver con mayor detalle lo que hay en ellas. Para llevar esto a cabo se utilizó el software *Illustator* [18] en un lienzo de tamaño 2000x2000 píxeles, con el fin de mejorar el dibujo del terreno y poder extraer una imagen de cada región con una resolución adecuada, añadiendo además iconos ilustrativos de las zonas más importantes, como pueden ser la jaula, la cueva o la casa. *Illustrator* hace uso de imágenes vectorizadas permitiendo ampliar imágenes sin producir *pixelado*, útil para extraer cada región del terreno a mayor resolución. En las figuras 28.b, 28.c y 28.d se muestra una ilustración de las tres regiones junto a la figura 28.a donde se muestra una vista global del mapa.



(a)

(b) (c) (d)

Figura 28. Mapa completo y regiones de la isla. Imagen (a) mapa completo, imagen (b) región 1, imagen (c) región 2, imagen (d) región 3.

Una vez se han creado las imágenes del mapa y de cada región, estas se han llevado *Unity* y se han incorporado al objeto *Canvas* de la escena *Island*, donde se usará el mapa. En dicho objeto se ha añadido la estructura de la figura 29, que incluye componentes *button* para ampliar cada región al "*clickar*" sobre ellas y un botón para regresar al mapa general, como se muestra en las imágenes de la figura 30.

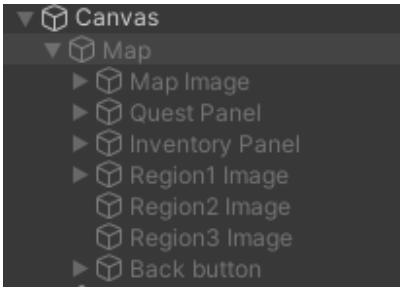


Figura 29. Resultado de la organización de los objetos del mapa en el objeto *Canvas*.

Figura 30. Izquierda: imagen completa del mapa con los botones de selección de cada región. Derecha: imagen de la *Región 1* ampliada junto al botón *Back* para volver al mapa completo.

Por último, dado que esta interfaz incorpora muchos elementos se ha modificado el modo de escalado (*UI Scale Mode*) de la componente *Canvas Scaler* (figura 31) del objeto *Canvas* para adaptarse a diferentes tamaños de pantalla sin afectar a la escala de los elementos de la interfaz.

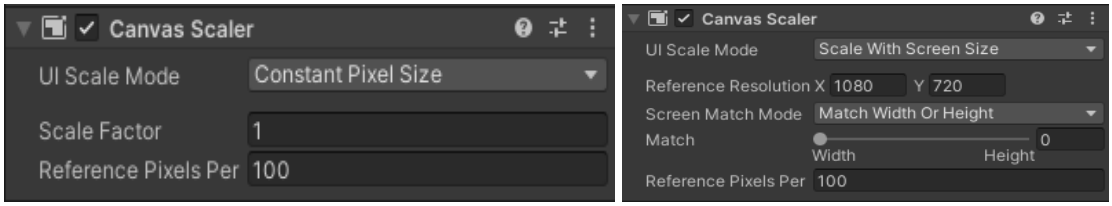


Figura 31. Componente *Canvas Scaler* del objeto *Canvas*. Izquierda: configuración antigua. Derecha: configuración nueva.

El control del mapa (mostrar u ocultar) se realiza a través de la tecla M en el control por teclado o mediante el gesto de levantar la mano izquierda por encima de la cabeza en el control por *Kinect* en el `Update()` de la clase `PhibyMixedController.cs`.

Para mostrar los objetos que se recogen por el mundo en esta nueva interfaz, se ha incluido el panel de herramientas (*Tools Panel*) existente en la versión anterior del UI y es necesario comentar una modificación hecha en el panel de herramientas respecto de versiones anteriores. El panel *Tools Panel* antes se disponía de manera vertical utilizando una componente que ofrece *Unity* llamada *Vertical Layout Group*, que permite equiespaciar los hijos del objeto padre que tiene dicha componente en la dirección vertical. Además este objeto rellena el espacio delimitado por el objeto padre a medida que se activan los objetos hijo. Sin embargo, dado que la disposición del nuevo inventario ya no es vertical, si no que es horizontal, esta componente se ha cambiado por una de tipo *Horizontal Layout Group*, cuyas funciones son las mismas pero en una disposición

horizontal. En la figura 32 hay una comparación de la configuración de los parámetros del *Vertical Layout Group* y del *Horizontal Layout Group*, así como una muestra del resultado en la la interfaz en la figura 33.

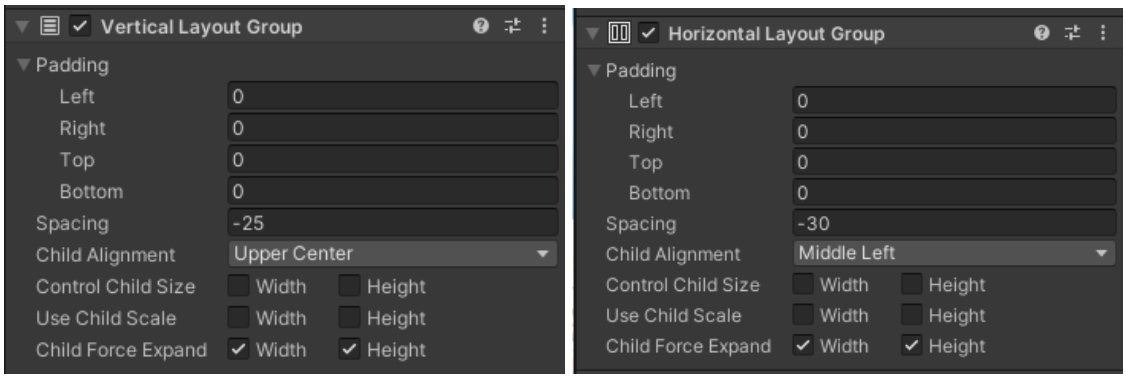


Figura 32. Izquierda: configuración del *Vertical Layout Group*. Derecha: configuración del *Horizontal Layout Group*.

Figura 33. Izquierda: disposición del *Tools Panel* en versiones anteriores. Derecha: disposición del *Tools Panel* en la interfaz de mapa.

Por último, antes de pasar al sistema de misiones, para la última versión de la interfaz se han añadido sonidos cada vez que se abre y se cierra el mapa (carpeta *Assets/Resources/Audios/Sounds/Interface*, sonidos *Map_open* y *Map_close*) grabados con el software gratuito *Reaper* [19], y se ha cambiado el diseño artístico de la interfaz, incorporando imágenes de mapas difuminadas, cambiando el color y la fuente de los textos [20] e incorporando un borde al panel del inventario. Estos cambios son visibles en la figura 34. Para el diseño de las fuentes se ha recurrido a la web *DaFont* [21].

Figura 34. Interfaz del mapa con el panel de misiones y el panel de inventario.

4.2.2. Sistema de misiones

Hacer mapas de grandes dimensiones puede dejar zonas del mapa que queden en desuso. Este problema de diseño sucede en la isla ya que parte de su extensión es inalcanzable por el jugador o está falta de contenido y mecánicas. Con el fin de ayudar a solucionar el problema, se ha añadido un sistema de misiones en el que los programadores puedan incorporar nuevas tareas y recompensas para el jugador, de modo que se premie su curiosidad por explorar la isla y cumplir diferentes objetivos.

Para ello en primer lugar se ha creado una clase que contiene las características propias de una misión llamada `Quest.cs`. Para definir esta clase se pensaron varios diseños, escogiendo al final el mostrado en la figura 35, donde sus principales atributos son:

- `QuestType type`: atributo de clase `QuestType`, tipo enumerado que indica si la misión es de tipo *main* para las misiones principales (*main quests*) o *side* para las misiones secundarias (*side quests*).
- `int questID`: número identificador único para cada misión.
- `int markID`: identificador que permite asignar la misión a una posición del mapa.
- `string title`: título de la misión.
- `string description`: explicación breve del objetivo de la misión.
- `bool completed`: indica si la misión se ha completado o no. Es el único atributo público.

```
public enum QuestType { Main, Side }            // Types of quests: Main for most important quests / Side for optional or less important quests
[SerializeField] private QuestType type;        // Indicates if a quest belongs to the main quests chain or a it's a side quest
[SerializeField] private int questID;           // Identification number of each quest
[SerializeField] private int markID;            // Mark icon to be active on the map
[SerializeField] private string title;          // Title of the quest
private string description;                     // A summary of quest's objectives
public bool completed;                          // True if quest is completed, false if not


public Quest(QuestType type, int id, string title, string description, bool completed, int markID)
{
    this.type = type;
    this.questID = id;
    this.title = title;
    this.description = description;
    this.completed = completed;
    this.markID = markID;

    //TODO: Set a reward
}
```

Figura 35. Declaración de atributos y constructor de la clase *`Quest.cs`*.

Originalmente se planteó utilizar una clase auxiliar `QuestGoal.cs` que definiera los objetivos, según el tipo de tarea a realizar. Sin embargo, aunque esta clase puede ser útil en futuras versiones, por simplicidad se ha prescindido de ella, ya que la mayoría de misiones pensadas tenían el mismo objetivo: completar un minijuego o encontrar a un personaje u objeto. Esto hizo que resultase más sencillo marcar como completada la misión a través de los controladores de los objetos o al salir de un minijuego. En la figura 36 se muestra un ejemplo de una versión de prueba de esta clase (que finalmente ha sido descartada) con algunos de sus los atributos en los que se pensó para su diseño. El más importante de ellos era el tipo de objetivo, permitiendo clasificar las misiones según el tipo de requisitos que esta tendría para completarse. En el ejemplo mostrado, se pensó en tres tipos de objetivos: de recolección (`Gathering`), de búsqueda de un objeto o personaje (`Find`) y de realización completa de un ejercicio (`CompleteAnExercise`).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/* Author: Juan Ignacio Fuentes-Pila
 * Date: 21/02/2022
 * Functionality: Defines de quest types of goal or how will be completed.
 * Changes:
 *  -
 * Notes:
 *  -
 */
[System.Serializable]
public class QuestGoal
{
    public GoalType goalType;
    public int requiredAmount;
    public int currentAmount;
    public bool founded;
    public bool exerciseDone;
    public enum GoalType { Gathering, Find, CompleteAnExercise }
}
```

Figura 36. Definición y atributos de la clase QuestGoal.cs.

Regresando a la clase `Quest.cs` se ha añadido un constructor para crear instancias de esta clase, en la que cada misión es una instancia diferente. Las misiones se guardarán en una lista global de misiones accesible por el resto de *scripts*, que se definirá al inicio del juego en el método `Start()` de la clase `GameState.cs`, tal y como se explica más adelante en este apartado. Para obtener los parámetros de cada misión se han aportado algunos métodos tipo *get* de los principales parámetros (figura 37), donde la variable *completed* es la única modificable por ser pública, ya que es necesario que todas las clases involucradas (en especial `GameState.cs` y aquellas clases que entregan una misión a *Phiby*) sepan qué misiones están completadas y cuáles no. Este diseño permite añadir fácilmente nuevos atributos como el que se propone en el apartado 5. Propuestas a futuro, como el de añadir uno o varios atributos de recompensas que premien al jugador al completar objetivos.

```
public int GetQuestID()
{
    return questID;
}

public int GetMarkID()
{
    return markID;
}

public QuestType GetQuestType()
{
    return type;
}

public string GetQuestTitle()
{
    return title;
}

public string GetQuestDescription()
{
    return description;
}
```

Figura 37. Definición de métodos *get* de la clase Quest.cs.

Todas las misiones están contenidas en una lista que debe ser persistente para saber qué misiones se han completado y cuáles no durante la ejecución del juego. Esta lista se define en la clase `GameState.cs`, y en su método `Start()` se han definido y añadido a dicha lista un total de 11 misiones, siendo una de ellas secundaria, tal y como se muestra en la figura 38. Esta lista es fácilmente ampliable por futuros desarrolladores, pudiendo incluso crear listas separadas para las misiones principales y las secundarias, o incluso realizar cadenas de misiones principales diferentes según el progreso del jugador en las diferentes regiones de la isla.

```
public void Start()
{
    Debug.Log(Application.persistentDataPath);
    //List of main quests (QuestTye, id, title, description, completed)
    questList.Add(new Quest(Quest.QuestType.Main, 0, "Get out of here! Part 1", "Find a way to get out of the cage.", false, 0));
    questList.Add(new Quest(Quest.QuestType.Main, 1, "Get out of here! Part 2", "Grab some straws and climb up the bars.", false, 0));
    questList.Add(new Quest(Quest.QuestType.Main, 2, "Meeting Grandma", "Find Grandma in her house.", false, 1));
    questList.Add(new Quest(Quest.QuestType.Main, 3, "Where's my key?", "Find Grandma's key in the forest.", false, 2));
    questList.Add(new Quest(Quest.QuestType.Main, 4, "Mmmm... Apples!", "Check the bowl inside the house.", false, 1));
    questList.Add(new Quest(Quest.QuestType.Main, 5, "Climbing the apple trees", "Take a look behind the hut, there is one big and two s
    questList.Add(new Quest(Quest.QuestType.Main, 6, "Fill the bowl", "Back to Grandma's house and put the apples in the bowl.", false,
    questList.Add(new Quest(Quest.QuestType.Main, 7, "The Bridge", "Go to the bridge.", false, 5));
    questList.Add(new Quest(Quest.QuestType.Main, 8, "Looking for a rope", "Find a rope near the forest to fix the bridge.", false, 5));
    questList.Add(new Quest(Quest.QuestType.Main, 9, "Find Rocky", "Find Rocky and bring him back to help you.", false, 4));
    questList.Add(new Quest(Quest.QuestType.Main, 10, "Free Rocky", "Help Rocky to escape from the collapsed cave.", false, 4));

    //List of side quests (QuestTye, id, title, description, completed)
    questList.Add(new Quest(Quest.QuestType.Side, 11, "More Apples!", "Repeat climbing trees to get more apples", false, 0));
```

Figura 38. Inicialización de la lista de misiones del juego.

Las misiones y su interacción con el resto de objetos deben ser administradas por `QuestManager.cs` , clase creada con este fin. Sus principales variables son:

- `List<Quest> playerQuests`: que almacena las misiones que tiene el jugador en ese momento. `QuestManager.cs` modifica y muestra esta lista en el panel de misiones de la interfaz del mapa.
- Variables `Text[] title` y `Text[] description` para mostrar el título y la descripción de cada misión.
- Los *arrays* de `GameObjects mainMarks` y `sideMarks` que contienen las marcas del mapa que se asignarán a cada misión.
- `GameObject QuestMark` que sirve de alerta para informar al jugador de que hay una nueva misión en su lista.

La gestión de estos parámetros se hace a través de cinco métodos:

- `public void NewQuest (Quest quest, int pos)`: añade la misión pasada como parámetro a la lista del jugador y pasa la posición del panel donde se quiere colocar a `ShowQuest()` para mostrarla en el *Quest Panel*.
- `public void CompleteQuest (Quest quest, int pos)`: elimina la misión pasada de la lista del jugador, la marca como completada y pasa la posición dada a `ShowQuest()` para eliminarla del *Quest Panel*.
- `public void ShowQuest(Quest quest, int pos, bool show)`: muestra o deja de mostrar la misión pasada como parámetro en la posición del panel indicada dependiendo del valor de *show* pasado (`true` = mostrar, `false` = ocultar).
- `private void CompletePreviousQuests(int lastQuest)`: método auxiliar que marca como completadas las misiones previas a la misión dada. Este método se ha utilizado para completar las misiones previas a un determinado *checkpoint* cuando se carga el juego desde el menú de *Editor's Tool* del menú principal.

- `private IEnumerator QuestNoticeShowing()`: corrutina que activa el icono de misión para informar al jugador de que hay una nueva misión en su lista. Está basado en el método `StartingEffect()` hecho por Haoru.

El objetivo es que los NPC's y otros objetos de la isla den cada misión al jugador al interactuar con ellos, para lo cual acceden al método público `NewQuest()`. Actualmente, el sistema está pensado para que las misiones se recojan y se completen de manera automática, por lo que siempre que se añade una misión, se completa la anterior. En la figura 39 se muestra un ejemplo de cómo se incorpora esto en el controlador del NPC "Granny" (`Grandma.cs`) al entrar en su *collider*, completando la misión 2 que está en la posición 0 del panel de misiones, y entregando la siguiente misión para mostrarla en la misma posición.

```
IEnumerator OnTriggerEnter(Collider other)
{
    if (other.tag == "Player")
    {
        switch (gameManager.GetCheckPoint())
        {
            case 0: // At this point, player shouldn't have access to Grandma
                if (!aS.isPlaying)
                {
                    aS.PlayOneShot(clips[0]);
                    gameManager.ShowTip("You are not supposed to talk to me...");
                }
                break;

            case 1:
                if (firstTime)
                {
                    if (!gameManager.GetQuest(2).completed)
                    {
                        QC.CompleteQuest(gameManager.GetQuest(2), 0);
                        QC.NewQuest(gameManager.GetQuest(3), 0);
                    }
```

Figura 39. Ejemplo de implementación de finalización y entrega de misiones en el controller del NPC "Granny".

Tanto al añadir una nueva misión como al completarla, se activa el método `ShowQuest()` que añade la correspondiente misión a la posición del panel indicada, asignando el texto y la descripción de la misión a los objetos de tipo `Text` en las variables de la clase. Este método distingue además entre el tipo de misión (si es principal o secundaria) mostrando el texto de un color diferente dependiendo de su tipo (amarillo para principales y verde para secundarias). Cada vez que se añade una nueva misión se muestra un icono que parpadea gracias a la corrutina `QuestNoticeShowing()` creada con el fin de avisar al jugador cuando esto sucede. Además, al mostrar el icono se reproduce el sonido *quest_sound.wav* incluido en la carpeta *Assets/Resources/Audios/Sounds/Interface* y que ha sido grabado con *Reaper*.

En la figura 40 se muestra un ejemplo en una versión intermedia de la interfaz de mapa de cómo se ve el panel de misiones cuando el jugador tiene una misión activa en su lista de misiones, apareciendo en el panel el título de la misión en color amarillo (por ser una misión principal) y una descripción del objetivo a realizar, junto a un contador de las misiones activas cerca del título del panel de misiones.

En la figura 41 se muestra un ejemplo con 2 misiones, siendo una de ellas una misión secundaria, y en la figura 42 un ejemplo del icono que informa al jugador de que se ha añadido una nueva misión a su lista de misiones.

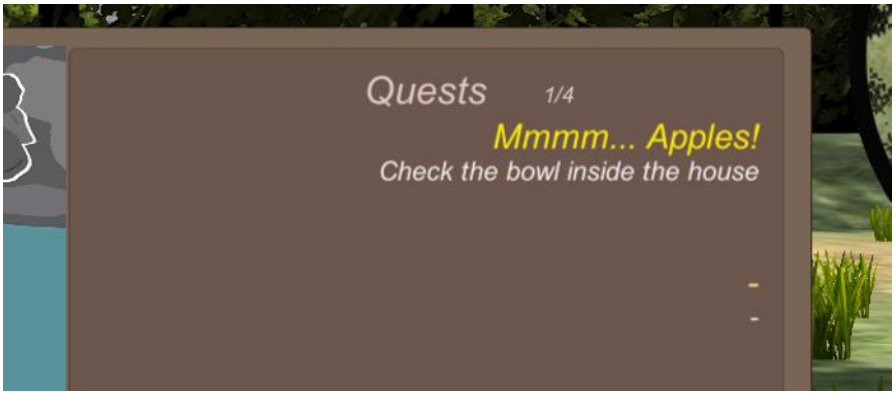


Figura 40. Ejemplo del panel de misiones con una misión activa.

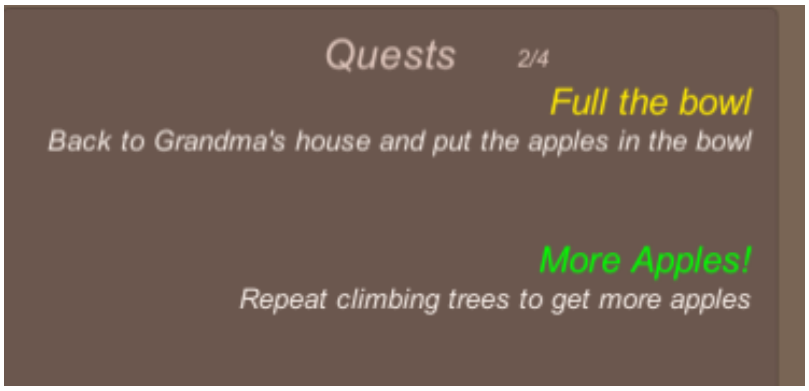


Figura 41. Ejemplo del panel de misiones con dos misiones activas: una misión principal (en color amarillo) y una misión secundaria (en color verde).

Figura 42. Ejemplo de icono de misión.

Para que el jugador tenga una pista de dónde debe completar la misión, a cada una se le asigna una marca en el mapa que indica la posición donde se completa dicha misión. Estas marcas están incluidas en el *Canvas* y se asignan a cada misión según la zona de la región a la que se quiera enviar al jugador. En el caso de la figura 43 se muestran las marcas creadas en la jerarquía para esta versión del juego en la región 1 del mapa (donde se desarrolla actualmente la mayor parte del juego). En la figura 44 se muestra un caso hipotético en el que se han marcado varias misiones principales y una secundaria con marcas diferentes en el mapa.

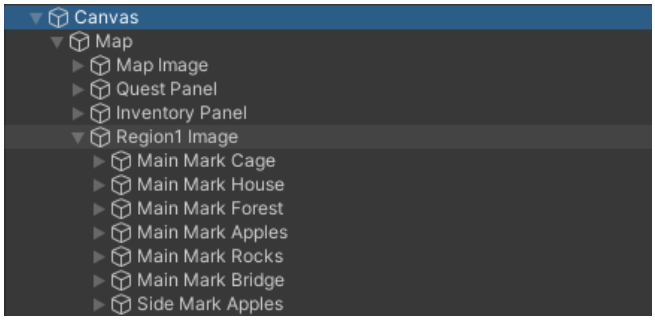


Figura 43. Distribución de las marcas de misión en la jerarquía de Unity dentro del objeto *Region 1 Image*.

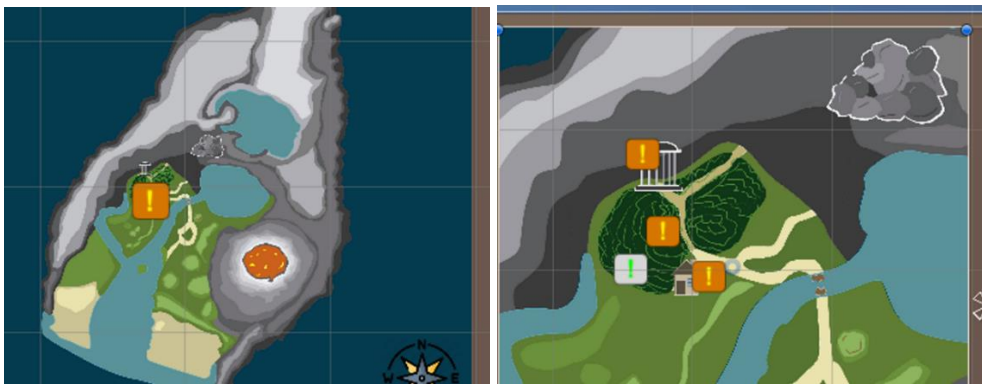


Figura 44. Ejemplo de iconos de misión en el mapa. Izquierda: icono de misión en la *Región 1*. Derecha: iconos de misiones activas en la *Región 1*.

Para completar su funcionamiento, y atender algunas necesidades respecto de otras clases, `QuestManager.cs` cuenta con un método `Start()` y un método `Update()`.

- El método `Start()` se usa fundamentalmente para completar y asignar las misiones al volver de un ejercicio y para completar las misiones anteriores al iniciar el juego desde un *checkpoint*. Esto se debe a que *QuestManager*, por cuestiones de diseño, no es un objeto persistente, de modo que no se guarda en esta clase la información sobre qué misiones se han completado.
- Por otro lado el método `Update()` se utiliza para actualizar el contador de misiones en el panel de misiones, de modo que el jugador pueda ver cuantas

misiones tiene y cuántas más puede recoger. En la figura 45 se muestra una captura de un ejemplo del panel de misiones con todas sus características del sistema de misiones activas, integrado en una versión intermedia de la interfaz del mapa.

Figura 45. Resultado del sistema de misiones en funcionamiento e integrado en la interfaz de mapa.

4.2.3. Sistema de calibración y control del cursor en modo Kinect

Debido a la imposibilidad de interactuar con los menús y con algunas características del juego en el modo *Kinect*, se ha incorporado un control del cursor junto al sistema de calibración desarrollados por Eduardo Botija [2] que permiten adaptar los controles del juego a la movilidad de cada usuario. Estas clases únicamente se han adaptado para ser usadas en *Phiby's Adventure*, sufriendo leves cambios como por ejemplo el acceso a la variable `isKinect` de la clase `GameState.cs`.

Para implementar esta característica se ha creado un objeto *CursorController* en la escena *MenuIntro* que tiene los scripts: `CursorController.cs`, `ClickController.cs` y `SmoothMovement.cs`. Cada uno cumple, respectivamente, la función de determinar la posición del cursor, definir el "área de *click*" sobre la cual se produce un *click* pasado un cierto tiempo, y suavizar el movimiento del curso según la calibración realizada. Estas clases necesitan del *KinectAsset* para conectarse con el *middleware* y recibir los movimientos del usuario captados por *Kinect* [7], siendo la tecla K la que activa o desactiva el control del cursor mediante el dispositivo.

Para calibrar el movimiento del cursor se ha incorporado un objeto *CalibrationManager* que contiene las clases `CalibrationProcess.cs` y `ArmCalibration.cs`. `CalibrationProcess.cs` indica al usuario los pasos a seguir durante el proceso de

calibración en el objeto *Canvas*, al cual se le han incorporado el objeto *CalibrationGuide* que contiene los paneles que muestran las instrucciones a seguir (figura 46). Mientras, `ArmCalibration.cs` determina el área de movimiento disponible del cursor, la amplificación necesaria para cubrir esa área y que mano se está controlando (izquierda o derecha). Para iniciar el proceso de calibración, el usuario debe acceder a él a través del menú de *Setting Options* > *Calibration* (figura 47*).* Esto abre el menú de calibración donde se ofrece la posibilidad cargar una calibración ya creada en un fichero del formato *.json* o iniciar una calibración nueva (figura 48).

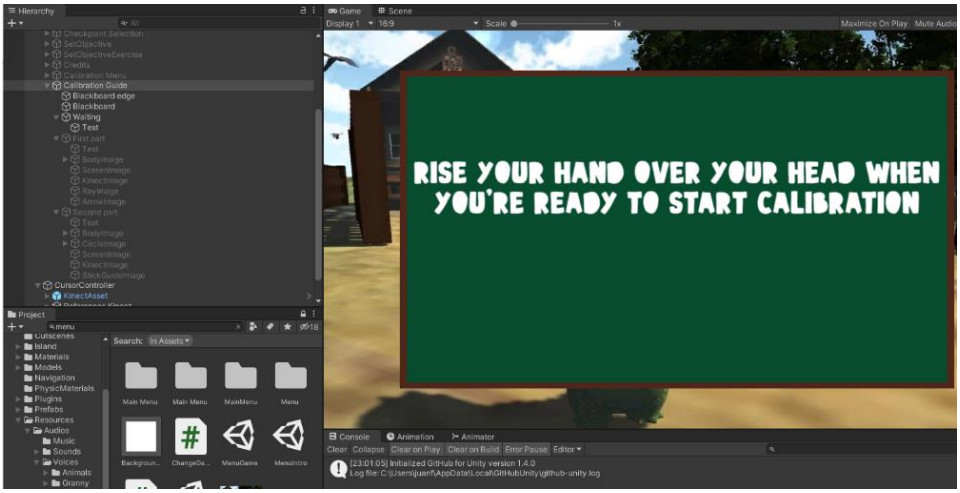


Figura 46. Modificación realizada en el objeto *Canvas*.

Figura 47. Opción de calibración incoporada en el menú *Settings*

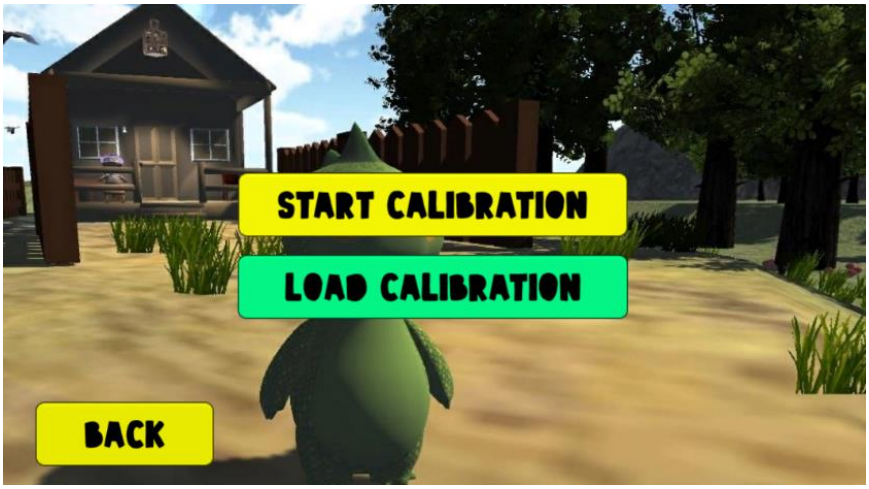


Figura 48. Menú de calibración.

Al iniciar el proceso de calibración se muestran las instrucciones en tres pasos.

- Paso 1 (figura 49): Se espera a que el usuario levante uno de sus dos brazos por encima de la cabeza. El brazo alzado es identificado por `ArmCalibration.cs` como el brazo que controlará el cursor.

Figura 49. Primer paso del proceso de calibración. Iniciar el proceso levantando el brazo a calibrar.

- Paso 2 (figura 50): Se pide al usuario apuntar al centro de la pantalla. `ArmCalibration.cs` define el punto de referencia del área de movimiento del cursor.

Figura 50. Segundo paso del proceso de calibración, apuntar a la pantalla

- Paso 3 (figura 51): Se pide al usuario realizar círculos concéntricos, teniendo un elemento que sirve de guía para comprobar que se está realizando el giro. En este punto `ArmCalibration.cs` define el área de movimiento del cursor, el cual depende de la pantalla.

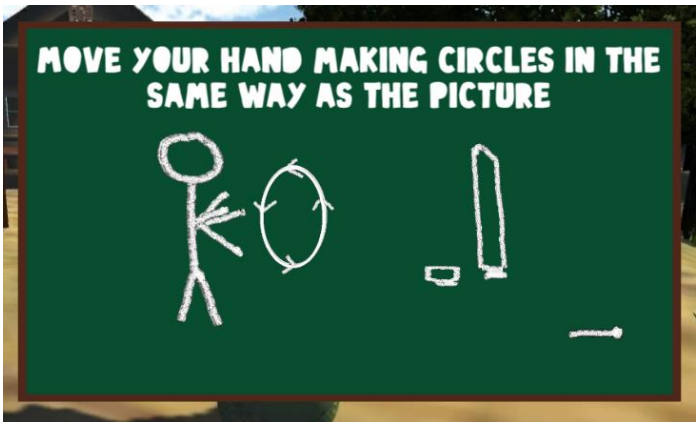


Figura 51. Tercer paso del proceso de calibración, realizar círculos.

Una vez completada la calibración, esta se guarda a través del script `CalibrationData.cs`, una modificación del script `SaveData.cs` de Eduardo hecha específicamente para guardar la información de calibración del brazo que se va a utilizar y que se guarda en el objeto *DataManager* para no perder la información de calibración durante la ejecución del juego.

Si bien se ha conseguido implementar el sistema de modo que el cursor responda a los movimientos del brazo y se produzca un *click* al situarlo un cierto tiempo sobre un área "clickable" (como un objeto *button*), los resultados obtenidos son algo mejorables. Dado que se está accediendo a funciones del sistema operativo, el movimiento del cursor depende de la resolución de cada pantalla, y no hay una traducción precisa entre el movimiento generado y el número de píxeles a trasladar el cursor.

Además es necesario confinar el cursor dentro de la ventana del juego por medio de `CursorLockMode.Confined` [22]. De no hacerse, dado que se están realizando llamadas al sistema operativo (a través de `[DllImport("user32.dll", EntryPoint = "SetCursorPos")]` [23]) y no se está utilizando una funcionalidad propia de Unity, pueden producirse comportamientos indeseados, especialmente si se utiliza más de un monitor (como por ejemplo, que el "clickado" automático cierre la ventana del juego o seleccione otra ventana que el usuario tenga activa en su ordenador, bloqueando el cursor en un punto de la pantalla y haciendo imposible desactivar el modo Kinect) .

4.3. Cambios menores

Para finalizar el apartado 4, se enumeran algunos cambios menores realizados en uno o varios *scripts* para mejorar el uso de otras características o hacer más sencillo el desarrollo.

- Uso del *Inspector*

  El *Inspector* es una herramienta que ofrece *Unity* para acceder y editar la información de los objetos de cada escena, incluidas las instancias de sus *scripts*. Esta herramienta estaba siendo infrautilizada, lo que implica un aumento de las líneas de código necesarias para cumplir toda la funcionalidad de los *scripts* y empeorando el rendimiento del videojuego.

  Si bien esta herramienta no es indispensable, puesto que existen ciertas limitaciones que son mejor abordadas desde el código, como por ejemplo al buscar instancias persistentes, sí es recomendable combinar su uso con la programación de *scripts*. Lo más importante es que los métodos `Update()` contengan la menor cantidad de procesos costosos en recursos (como el método `Find()`), ya que hasta que no se completen, no se pasa al siguiente frame, lo que ralentiza la ejecución del juego.

- Regiones de uso

  Se ha comenzado a estructurar el código en regiones de uso mediante las instrucciones `#region` y `#endregion`. Esto permite asimilar cuáles son las funciones de un determinado script con un rápido vistazo, lo que aporta mayor agilidad en el desarrollo y búsqueda de errores de *scripts* con un elevado número de líneas y funciones.

- Reducción de corrutinas

  Se ha cambiado algunos métodos de tipo `IEnumerator` a tipo `void`. La clase `IEnumerator` permite pausar o reanudar un determinado método y obtener un grado de paralelización ciertas acciones. Si bien este tipo de métodos es útil, no siempre en necesario, ya que utilizado de manera inadecuada puede hacer más difícil el seguimiento de variables del juego.

- Eliminación de código redundante y en desuso

  Ya se ha mencionado en algunos apartados de la memoria que se han eliminado secciones del código y variables que habían quedado en desuso, que no se había terminado de dar una funcionalidad o que incluso eran redundantes.

Esta tarea se ha realizado de forma general al consultar *scripts* en colaboración con Laura Álvaro Gil durante el desarrollo del proyecto, tomando precauciones a la hora de hacerlo.

- Características adicionales

  Se han añadido algunas características extra relacionadas con el uso de los menús y los sistemas del juego.

  - Se ha añadido la variable `bool allowKinect` en `GameState.cs` que habilita la posibilidad de cambiar el modo de control entre teclado y Kinect cuando el dispositivo está conectado y el *middleware* se ha iniciado. Esto se ha aplicado para evitar errores como los del apartado 3.2.3 relacionados con el cursor.
  - Se ha añadido la variable `bool backToMenu` en `GameState.cs` que permite regresar al menú principal cuando se carga un ejercicio desde el menú principal.
  - Se han añadido métodos de pausa que detienen la ejecución natural del juego cuando se pulsa la tecla ESC en la escena *Island*. Estos métodos se han añadido en `GameManager.cs` para simplificar su implementación, pero es posible que se puedan incorporar en un *script* que agrupe este tipo de características.

5. Conclusiones

Para finalizar la memoria de este PFG se va a hacer un resumen del trabajo hecho, concluyendo con si se han cumplido los objetivos del proyecto y cuáles han sido las dificultades que se han encontrado a lo largo del desarrollo.

El proyecto se ha basado en fases de diseño, planificación de tareas, implementación y fases de pruebas, de modo que se modificasen los objetivos a corto plazo a medida que avanzasen las semanas y se cumpliesen las tareas y objetivos previos. Es por ello por lo que se han realizado reuniones semanales con la tutora del proyecto Martina Eckert y con la compañera Laura Álvaro Gil con el fin de extraer versiones y actualizaciones periódicas del videojuego.

En las primeras reuniones se comenzó con la introducción al entorno de trabajo y se conoció la cronología del proyecto y su desarrollo por parte de los alumnos que han trabajado en él en sus contribuciones de PFG y prácticas. Se hicieron pruebas de la versión que había en ese momento del videojuego para familiarizarse con las mecánicas y sistemas que había en ese momento con el fin de aportar nuevas ideas al proyecto. Se empezó a profundizar en el proyecto, leyendo el código y probando el comportamiento de *scripts* con el fin de encontrar errores y puntos de mejora. Esta fue la parte más compleja del proyecto, ya que se trata del resultado de la contribución de muchos alumnos, lo que complica la comprensión de los algoritmos y sistemas que se han diseñado a lo largo de los años. Este problema ha persistido hasta las fases finales del desarrollo ya que la lista de *scripts* que se han tenido que entender a medida que se profundizaba en el proyecto ha sido cada vez mayor.

Teniendo en cuenta esto, se ha mejorado el comportamiento de algunos de los *scripts* más importantes para el funcionamiento del videojuego, se han dado más indicaciones en el propio código y se ha explicado su comportamiento para facilitar el trabajo de nuevos desarrolladores. De cara al jugador se han incorporado sistemas que hacen más entretenido el videojuego y que aportan funcionalidades y herramientas que hoy en día son tendencia en los videojuegos de aventuras de la industria.

Hay que mencionar que el uso del dispositivo *Kinect*, el cual es el elemento central del juego, ha añadido un grado más de complejidad al proyecto. Por ejemplo, a la hora de testear los cambios que se aplicaban en busca de errores, ha habido que realizar un mayor número de pruebas de las habituales: las pruebas correspondientes al comportamiento por teclado más las correspondientes al comportamiento con *Kinect*.

En conclusión, se ha tratado de un proyecto en el que se ha trabajado con diferentes tecnologías y herramientas para conseguir una versión más robusta del videojuego, con características útiles que permiten un desarrollo más rápido e interesante de Phiby’s Adventure 3D.

6. Propuestas a futuro

En este apartado se pretende dar una visión a futuro de los posibles cambios y mejoras en diferentes aspectos del videojuego que podrían ayudar tanto a la jugabilidad y experiencia de usuario, como al desarrollo por parte de nuevos diseñadores.

En primer lugar, se aconseja describir cada uno de los métodos y atributos que se implementen o modifiquen en nuevos PFGs o prácticas de la forma más clara y precisa posible, incluyendo encabezados y anotaciones adicionales (tarea que, si bien se ha hecho en parte, estaba en parte incompleta).

Otra tarea que se propuso hacer en fases iniciales del proyecto era la de incorporar un *sound manager* y *sound controllers* para cada NPC, de modo que estuviesen mejor controlados los sonidos del juego y que, por falta de tiempo, esto no se ha podido realizar.

También se recomienda continuar la tarea de diferenciar *managers* de *controllers* de modo que se delimiten las funciones de cada *script* de manera clara. Un ejemplo es el caso de funciones que están en `GameManager.cs` como el control de cinemáticas y el control de menús (entre otras funcionalidades) que, para simplificar el desarrollo, se han mantenido en esta clase ya que son funciones que se dan en la escena *Island*, pero que podrían estar en managers más especializados, reservando para `GameManager.cs`. las funciones relacionadas con la progresión del juego en la escena *Island*.

Una propuesta interesante para motivar al jugador para completar misiones es la de diseñar recompensas que se entreguen al finalizar cada misión, que puedan ser útiles para el progreso en el videojuego o bien puntos que motiven al usuario a seguir jugando y tratar de superar una marca personal. Además, se propone diseñar nuevas misiones, tanto principales como secundarias, que ofrezcan una mayor variedad de tareas o que fomenten la repetición de los ejercicios con el fin de hacer más interesante el movimiento por el mundo.

Se propone mejorar el control de la interfaz de mapa y los menús del juego en el control por *Kinect*, incorporando el trabajo realizado por Eduardo Botija en su PFG y con la finalidad con la que se diseñó: permitir que los ejercicios se adapten al rango de movimientos de los usuarios.

Por último, se propone un listado de tareas pendientes que pueden hacer de la siguiente versión del proyecto un videojuego mucho más completo:

- Finalizar la implementación del movimiento del cursor para todas las escenas del videojuego. Debido al comportamiento del Kinect Asset actual, no se ha podido implementar esta característica de manera satisfactoria para ser usada por el jugador. Con las nuevas versiones del Kinect Asset se espera que esta característica sea más sencilla de implementar.

- Completar el resto de los capítulos pendientes de Phiby's Adventure 3D. El proyecto cuenta con un mayor desarrollo de la historia en el que no se ha podido profundizar ya que el trabajo se ha centrado en desarrollar y mejorar otros aspectos del videojuego.
- Ampliar el listado de misiones e incorporar recompensas que motiven al usuario. Aunque el sistema cuenta con un conjunto de misiones, la idea de este es que el listado se amplie de acuerdo al desarrollo de la historia, ofreciendo una ventana abierta a la creatividad de los desarrolladores para mejorar la experiencia del usuario.

7. Referencias

[1] Laura Álvaro Gil, «Mejora de la experiencia y del contenido del videojuego terapéutico “Phiby’s Adventure 3D”» Escuela Técnica Superior de Ingeniería de Sistemas de Telecomunicación, UPM, Madrid, 2022

[2] Eduardo Botija «Revisión y Mejora de Videojuego West Gun Therapy» Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación, UPM, Madrid, 2022.

[3] GAMMA, «Grupo de Aplicaciones Multimedia y Acústica» [En línea]. Available: https://www.citsem.upm.es/es/quienes-somos/grupos-de-investigacion/grupo-de-aplicaciones-multimedia-y-acustica-gamma.

[4] CITSEM, «CITSEM,» [En línea]. Available: https://www.citsem.upm.es/es/.

[5] CITSEM, «“Medical Access to the Blender Exergames, Blexer» [En línea]. Available: https://blexer-med.citsem.upm.es/index.php.

[6] W. Community, «Kinect for Widows», 19 Mayo 2022. [En línea]. Available: https://docs.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows.

[7] C. Luaces Vela, «Diseño e implementación de un entrono virtual de ejercicios físicos» Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación, UPM, Madrid, Febrero 2018.

[8] Haoru Qin, «Diseño e implementación de la historia, las mecánicas y el flujo del videojuego serio “Phiby’s Adventures v2”» Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicaciones, UPM, Madrid, 2020.

[9] Mónica Jiménez, «Plataforma médica para el entorno de videojuegos terapéutico "Blexer"» Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación, UPM, Madrid, Julio 2017.

[10] A. A. López, «Implementación de medidas de seguridad en plataforma médica para entorno de videojuegos terapéuticos» Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación, UPM, Madrid, Julio 2019.

[11] Unity, «Unity,» [En línea]. Available: https://unity.com/es.

[12] Visual Studio Code, «Visual Studio Code» [En línea]. Available: https://code.visualstudio.com.

[13] GitHub, «GitHub» [En línea]. Available: https://github.com.

[14] GAMMA, «Phiby’s Adventure Game Design Document v2.5»

[15] Unity, «Object.DontDestroyOnLoad» [En línea] Available: https://docs.unity3d.com/ScriptReference/Object.DontDestroyOnLoad.html.

[16] Unity, «GameObject.Find» [En línea]. Available: https://docs.unity3d.com/es/530/ScriptReference/GameObject.Find.html.

[17] Unity, «Object.Instantiate» [En línea]. Available: https://docs.unity3d.com/ScriptReference/Object.Instantiate.html.

[18] Unity, «Script Execution Order» [En línea]. Available: https://docs.unity3d.com/Manual/class-MonoManager.html.

[19] Adobe, Adobe Illustator [En línea]. Available: https://www.adobe.com/es/products/illustrator.html.

[20] Reaper , «Reaper» [En línea]. Available: https://www.reaper.fm/.

[21] Fuentes de textos, «dafont» [En línea]. Available: https://www.dafont.com/es/

[22] Unity, «CursorLockMode.Confined» [En línea]. Available: https://docs.unity3d.com/2019.4/Documentation/ScriptReference/CursorLockMode.Confined.html.

[23] Microsoft, «SetCursorPos» [En línea]. Available: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setcursorpos.

Anexo I. Presupuesto

Este anexo presenta los recursos (humanos y materiales) que se han utilizado para desarrollar el proyecto y el coste que requeriría invertir en él.

En primer lugar se considera que el trabajo ha sido realizado por al menos un ingeniero junior de telecomunicaciones, habiendo trabajado un tiempo estimado de 325 horas (285 horas para el desarrollo del proyecto y 40 horas para la elaboración del informe). Tomando como referencia el salario medio de un Ingeniero recién titulado [1], el cual ronda un valor bruto de 20.600 €/año, y considerando unas 40 horas semanales de trabajo, se tiene un salario por hora de 10,73 €/hora y, por tanto un coste total de 3486,98 € brutos.

En cuanto al coste material del proyecto, este viene marcado fundamentalmente por el ordenador de desarrollo, las licencias *software* y el dispositivo *Kinect One* de *Microsoft*. A modo de ejemplo, se da un listado del coste de cada elemento (considerando los componentes del ordenador) y el cómputo total. En el coste del ordenador se consideran los requisitos mínimos para el funcionamiento del editor de *Unity* [2] y se presenta una configuración de componentes posible para desarrollar el proyecto.

Es importante tener en consideración que los precios del material pueden variar a lo largo del tiempo según a el contexto económico del momento.

Tabla 1. Presupuesto del coste humano y material del proyecto.

Descripción	**Unidades**	**Coste parcial (€)**
Coste ingeniero junior (325h)	1	3486,98
Licencia OEM Windows 10 Home [3]	1	19,75
Procesador AMD Ryzen 5600G, 6 núcleos 4.4GHz [4]	1	179,89
Módulo memoria RAM 8GB [5]	2	63,98
Placa Base 52,52€ [6]	1	52,52
Disco duro SATA 1TB [7]	1	39,99
Fuente de alimentación 650W [8]	1	64,99
Caja/Torre contenedora [9]	1	36,49
Ratón [10]	1	1,98
Teclado [11]	1	4,99
Monitor full HD [12]	1	118,99
Kinect One [13]	1	144,95
COSTE TOTAL (€)		**4.204,50**

Las licencias software utilizadas en el proyecto de *Unity*, *GitHub*, *Visual Studio Code*, GIMP y *Reaper* son gratuitas. *Adobe Ilustrator* cuenta con una versión de prueba gratuita.

7.1. Referencias

[1] Jobted, «Sueldo del Ingeniero de Telecomunicaciones en España» [En línea]. Available: https://www.jobted.es/salario/ingeniero-telecomunicaciones [Último acceso: septiembre 2022]

[2] Unity, «System requirements for Unity 2020.1» [En línea]. Available: https://docs.unity3d.com/2020.1/Documentation/Manual/system-requirements.html [Último acceso: septiembre 2022]

[3] GVGmall, «Licencia de Windows 10 OEM» [En línea]. Available: https://www.gvgmall.com/microsoft-windows-10-home-oem-cd-key-global.html [Último acceso: septiembre 2022]

[4] PcComponetes.com «Procesador AMD Ryzen 5600G» [En línea]. Available: https://www.pccomponentes.com/amd-ryzen-5-5600g-440ghz [Último acceso: septiembre 2022]

[5] PcComponetes.com «Memoria RAM Kingston Fury Beast DDR4» [En línea]. Available: https://www.pccomponentes.com/kingston-fury-beast-ddr4-2666-mhz-8gb-cl16 [Último acceso: septiembre 2022]

[6] PcComponetes.com «Placa Base Gigabyte B450» [En línea]. Available: https://www.pccomponentes.com/gigabyte-b450m-s2h [Último acceso: septiembre 2022]

[7] PcComponetes.com «Disco SATA Seagate Barracuda 1 TB» [En línea]. Available: https://www.pccomponentes.com/seagate-barracuda-35-1tb-sata3 [Último acceso: septiembre 2022]

[8] PcComponetes.com «Fuente de slimentación Corsair CV series 650W Plus Bronze» [En línea]. Available: https://www.pccomponentes.com/corsair-cv-series-cv650-650w-80-plus-bronze-v2 [Último acceso: septiembre 2022]

[9] PcComponetes.com «Torre Nox Kore USB 3.0» [En línea]. Available: https://www.pccomponentes.com/nox-kore-usb-30 [Último acceso: septiembre 2022]

[10] PcComponetes.com «Mouse Hama MC-100» [En línea]. Available: https://www.pccomponentes.com/hama-mc-100-raton-con-cable-1000dpi-negro [Último acceso: septiembre 2022]

[11] PcComponetes.com «Teclado Owlotech Office K300» [En línea]. Available: https://www.pccomponentes.com/owlotech-office-k300-teclado-usb-negro [Último acceso: septiembre 2022]

[12] PcComponetes.com «Monitor Philips 226E9QHAB 21,5''LED IPS FullHD Freesync» [En línea]. Available: https://www.pccomponentes.com/philips-226e9qhab-215-led-ips-fullhd-freesync [Último acceso: septiembre 2022]

[13] Game «Dispositivo Kinect One» https://www.game.es/kinect-xbox-one-xbox-one-102796 [Último acceso: septiembre 2022]

Anexo II. Uso de la plataforma GitHub en Unity

GitHub es una plataforma de desarrollo colaborativo que permite alojar proyectos y usa el sistema de control de versiones *Git*. A pesar de ser más conocido por ser usado principalmente para la creación y desarrollo de código fuente, en la actualidad *GitHub* también puede ser un buen aliado para el desarrollo de proyectos en *Unity*. En las figuras 52, 53, 54 y 55 se muestran ejemplos de la interfaz de usuario en la web de *GitHub* en un proyecto.

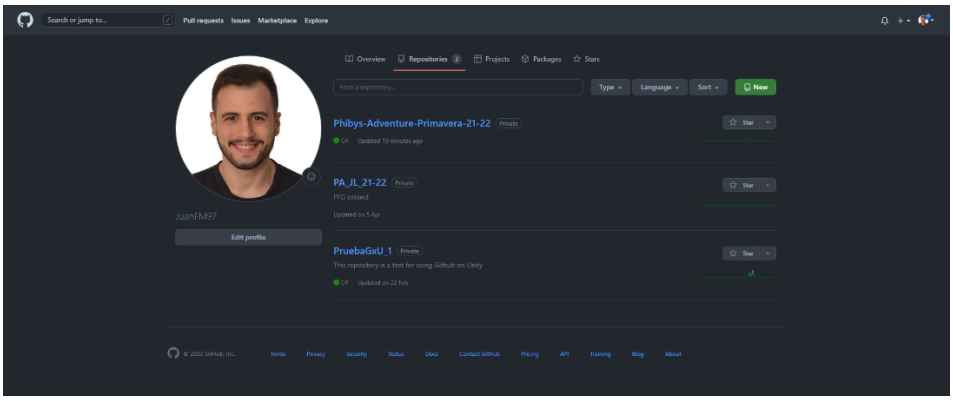


Figura 52. Interfaz de *GitHub* en su versión web

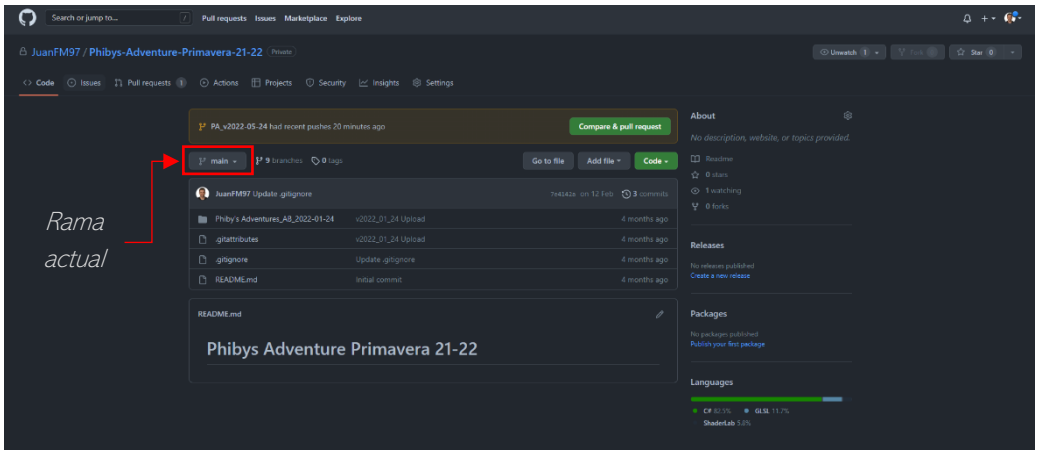


Figura 53. Interfaz de un repositorio en *GitHub* web

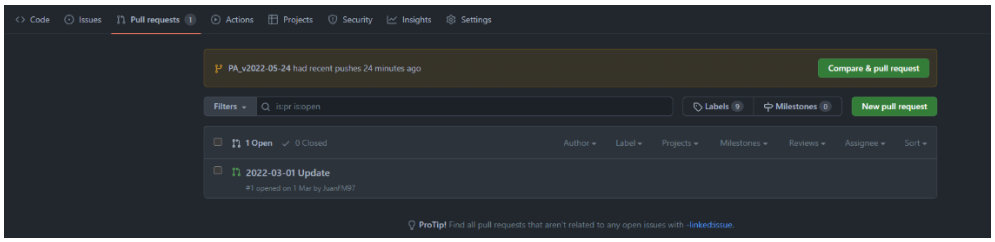


Figura 54. Pestaña de *pulls* o actualizaciones

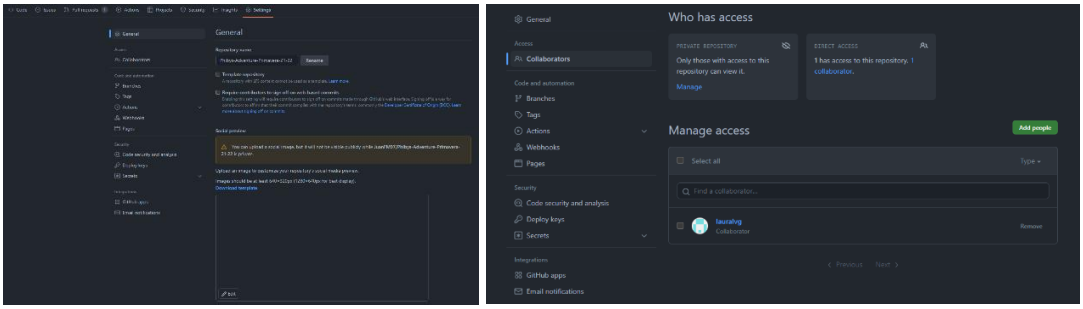


Figura 55. Pestaña de opciones (izquierda). Panel de colaboradores dentro de la pestaña de opciones (derecha).

Para empezar a usar *GitHub*, lo primero, tras darse de alta en la plataforma con una cuenta, es crear un repositorio, pudiendo asignarle un nombre y, opcionalmente, incluir una descripción. Este es el sitio en el que se almacenará el proyecto y donde se reflejarán los cambios que se hagan en él de forma remota, y donde cualquiera de los usuarios incluidos en el repositorio puede descargar dichos cambios para actualizar su proyecto. Sólo es posible crear repositorios públicos si se usa la plataforma de forma gratuita, pero se permite incluir algunos ficheros de texto como un *readme* o un *gitignore* de forma automática (figura 56). Un fichero *gitignore* permite despreciar los archivos cuyas extensiones figuren en él. Por defecto hay un *gitignore* (figura 57) para *Unity*, útil para ignorar los archivos temporales que este utiliza.

Nombre	Fecha de modificación	Tipo	Tamaño
Phiby's Adventures_JL_2022	25/05/2022 14:16	Carpeta de archivos	
.gitattributes	12/02/2022 12:02	Archivo GITATTRIB...	1 KB
.gitignore	12/02/2022 12:03	Archivo GITIGNORE	1 KB
Phiby's Adventures_AB_2022-01-24.rar	06/02/2022 14:29	Archivo RAR	946.347 KB
README.md	12/02/2022 11:07	Archivo MD	1 KB

Figura 56. Ficheros gitignore y README creados en el respositorio junto al proyecto.

Figura 57. Información fichero gitignore.

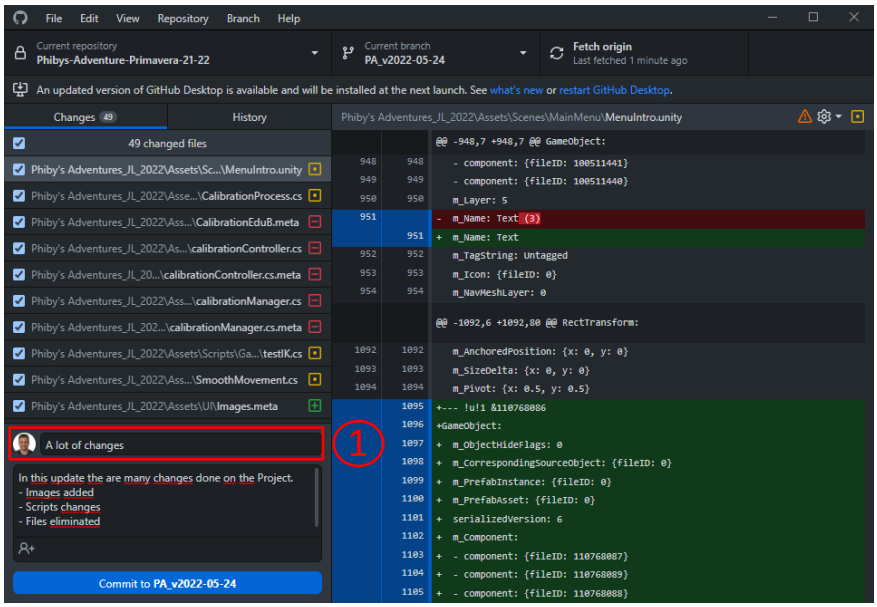
Una vez subido (o creado) el proyecto de *Unity* a ese repositorio se debe compartir al resto de desarrolladores para que puedan clonar ese repositorio a través de un enlace, creándose una copia local en su ordenador. Es necesario realizar los siguientes ajustes en *Unity* para que el control de versiones sea más eficaz:

- *Edit* > *Project Settings* > *Editor* > *Version Control* > *Visible Meta Files*. Estos archivos guardan información sobre los ajustes y la configuración de cada objeto creado en *Unity*. Deben estar activados para que sean detectados.

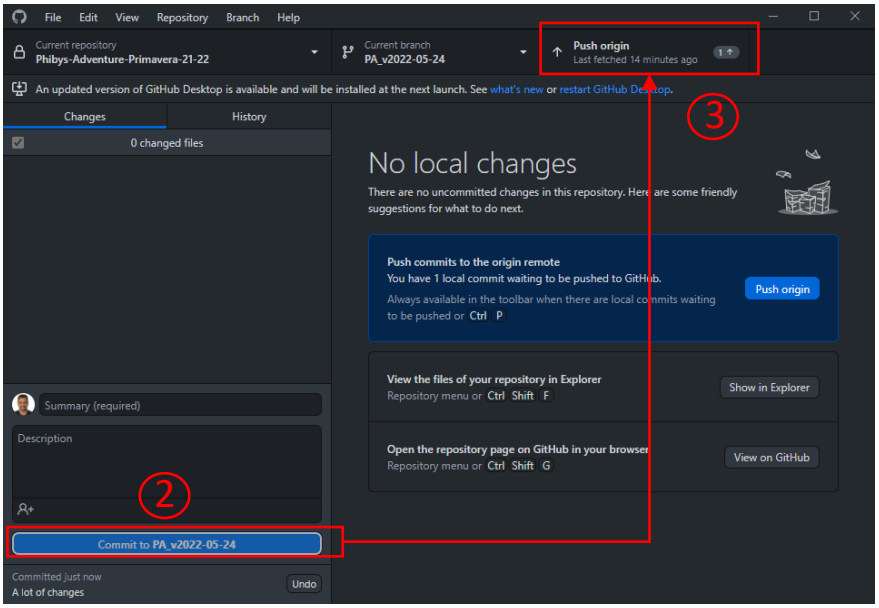
- *Edit* > *Project Settings* > *Editor* > *Asset Serialization* > *Force Text*. Este ajuste hace que muchos de los archivos creados por *Unity* sean más fáciles de leer.

Cuando un desarrollador haga cambios en su versión local del proyecto, deberá subirlos al repositorio de *GitHub*, y el resto de los contribuidores recibirán una notificación para actualizar sus copias de proyecto con las modificaciones nuevas. *GitHub* permite trabajar con diferentes versiones o *branches* (ramas) en el que cada desarrollador puede trabajar con una versión diferente o generar actualizaciones del proyecto, teniendo por defecto una rama principal o *main* donde se irán reflejando los cambios.

Para realizar esta tarea de forma más sencilla, es recomendable usar el cliente *GitHub Desktop*, aunque también se puede hacer a través de líneas de comando o editar código desde la aplicación web. En la figura 58 se pueden observar un ejemplo de uso de la de la aplicación de escritorio de *GitHub* para actualizar un proyecto.



(a)



(b)

Figura 58. Ejemplo de actualización de cambios en la aplicación de *GitHub*. Imagen (a): lista de cambios y paso 1 Introducir resumen de cambios. Imagen (b): pasos 2 y 3, actualizar rama y publicar cambios.

En la zona superior de la interfaz se encuentra el nombre del proyecto, la rama de trabajo y el botón de actualización, donde se puede comprobar si alguien ha actualizado el proyecto. A la izquierda figuran las actualizaciones hechas en el proyecto, con tres categorías de colores: verde para ficheros añadidos, amarillo para ficheros cambiados y rojo para ficheros eliminados. Este patrón de colores se repite para el panel derecho donde se puede observar con más detalles cuales son esos cambios realizados.

Antes de publicar los cambios, es necesario añadir un resumen de los cambios y, opcionalmente, una descripción más detallada donde se puede explicar en qué consisten dichos cambios. Una vez hecho, pulsando en *Commit to <branch>*, se añadirán los cambios a la rama, siendo el último paso clickar en el botón *Push origin* para que el resto de usuarios puedan ver y descargar los cambios (figura 59). En ese momento, otro usuario puede decidir si actualizar su proyecto o conservar su versión.

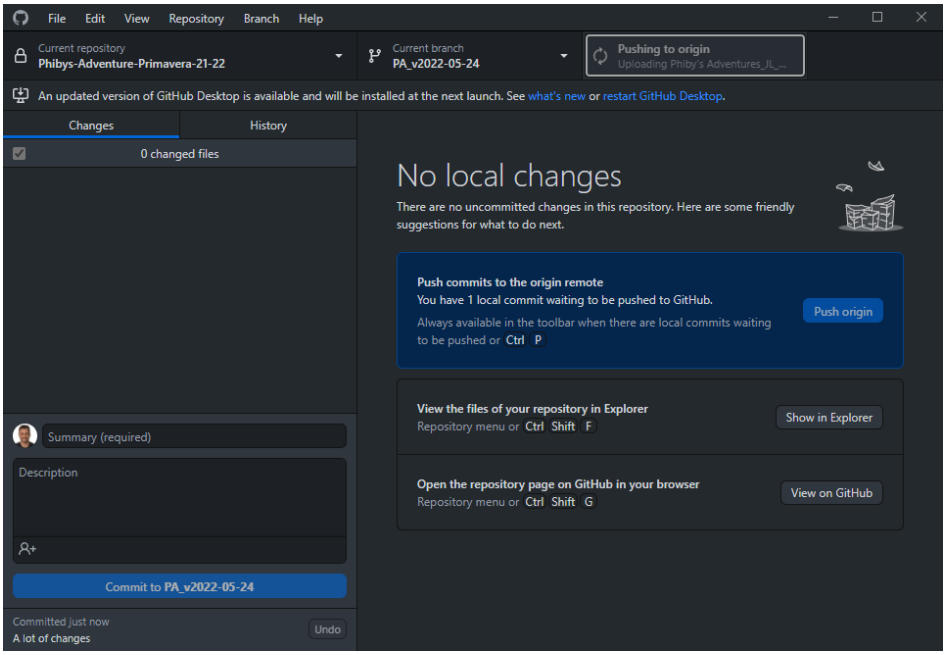


Figura 59. Ejemplo de publicación de cambios en el proyecto.