
# Universidad de Valladolid

## Escuela de Ingeniería Informática

**Trabajo Fin de Grado**

Grado en Ingeniería Informática

**Gamificación de Fundamentos de la Programación: Juegos serios para el aprendizaje de estructuras de código iterativas, métodos y funciones**

Autor:

**Mario Torbado de la Rosa**

Tutores:

**Alma María Pisabarro Marrón**

**Carlos Enrique Vivaracho Pascual**

# ÍNDICE GENERAL

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# RESUMEN

Esta es la memoria del Trabajo de Fin de Grado realizado por Don Mario Torbado de la Rosa, para la Facultad de Ingeniería Informática de la Universidad de Valladolid, en Julio de 2021.
En ella, se describe el desarrollo de 2 videojuegos serios para la gamificación de la asignatura Fundamentos de Programación, cuyo objetivo es ayudar a los alumnos de primer curso a interiorizar los conceptos de estructuras de control y de uso de métodos y funciones.

A mayores, se ha realizado la integración de dichos juegos con la plataforma web *Programa Jugando*[1] del grupo GREIDI[2], cuyo desarrollo formó parte de un Trabajo Final de Grado previo, realizado por Don Juan Rodríguez Sanz.

# 1

# INTRODUCCIÓN

Los juegos serios son aquellos cuyo objetivo principal es otro que el de la diversión del jugador. Existen diversos objetivos, formas y medios que puede tomar un juego serio, pero en cualquier caso, tienen como función motivar al receptor o mejorar su efectividad a la hora de realizar una tarea, cuyos resultados, en estos aspectos, podría ser inferior si se lleva a cabo de forma tradicional.

En este trabajo, abordaremos el diseño y realización de juegos serios, en la forma de videojuegos, para su aplicación en la docencia.
El uso de estos en el ámbito educativo es un tipo de gamificación en el aula, pues su propósito es servir de herramienta educativa, bien como refuerzo a las técnicas de enseñanza convencionales o sustituyendo a algunas de estas.

En concreto, este proyecto tiene como objetivo la creación de dos videojuegos educativos, tratando en cada uno algunos de los conceptos básicos de la programación.
Su público objetivo son los alumnos de la asignatura de Fundamentos de Programación, de Grado en Ingeniería Informática de la Universidad de Valladolid, aunque su aplicación podría expandirse a otros grados o niveles docentes.

A su vez, este trabajo forma parte de un proyecto mayor, la plataforma *Programa Jugando*[1], iniciativa del grupo GREIDI[2], que tiene como objetivo la gamificación completa del temario de la asignatura de Fundamentos de Programación.

## 1.1 MOTIVACIÓN

La programación es una materia que requiere un alto grado de abstracción para comprender muchos de sus conceptos, que pueden resultar complicados de transmitir de forma verbal.

Conceptos de este tipo pueden ser asimilados mejor una vez que se comienzan a poner en práctica, pero las herramientas de programación convencionales como compiladores o entornos de desarrollo, suelen ser complejas y poco intuitivas para un alumno que tiene su primera toma de contacto con las mismas. Esto puede resultar en una experiencia frustrante al no comprender por qué algo no funciona, no entender un error de compilación o no ser capaz de visualizar qué está haciendo realmente un fragmento de código.

La motivación de este proyecto es la de proporcionar al alumnado de la Escuela de una herramienta de aprendizaje que sirva de puente entre las lecciones teóricas del profesorado y las prácticas de programación.

Con esta herramienta, en forma de videojuegos serios, se motiva al alumno en el proceso de asimilación de conceptos de programación básicos, en un entorno controlado y que traslada la abstracción del código a una representación gráfica, facilitando su interpretación.

## 1.2 OBJETIVOS

El objetivo general del proyecto, es la asistencia a la formación del alumnado de primer curso en los conceptos de programación siguientes:

- Estructuras de código condicionales.
- Estructuras de código iterativas.
- Manejo de métodos y funciones.
- Paso por referencia y paso por variable.

Para su tratamiento, el proyecto dividirá la carga de aprendizaje en dos videojuegos, basados en las mismas mecánicas, pero tratando uno los dos primeros conceptos y otro los dos últimos.

Los objetivos específicos del proyecto, para poder cumplir su función correctamente, son los siguientes:

- Dotar a los juegos de un aspecto lúdico que motive a los alumnos en su participación.
- Enseñar la implementación de estructuras de código condicional, iterativo y métodos y funciones en la sintaxis del lenguaje de programación Java. También tratar indirectamente los conceptos de paso por referencia y paso por variable.
- Dotar a los juegos de un sistema de puntuación, que no promueva en exceso la competitividad entre alumnos, pero sirva como referencia a los profesores para conocer el grado de progresión del alumno en el aprendizaje de los conceptos indicados.
- Integrar los juegos en la plataforma *Programa Jugando*.

# 2

# HERRAMIENTAS UTILIZADAS

## 2.1 UNITY

Unity (figura 1) es un motor de videojuego multiplataforma creado por *Unity Technologies* y que está disponible como plataforma de desarrollo para Microsoft Windows, Mac OS, Linux [3].

Unity soporta *scripting* en el lenguaje C#, para el que incluye una extensa librería con la que trabajaremos, *UnityEngine*. También nos permite exportar el juego desarrollado a una gran variedad de plataformas objetivo de forma nativa (PC, consolas, Web, dispositivos móviles y de realidad aumentada).

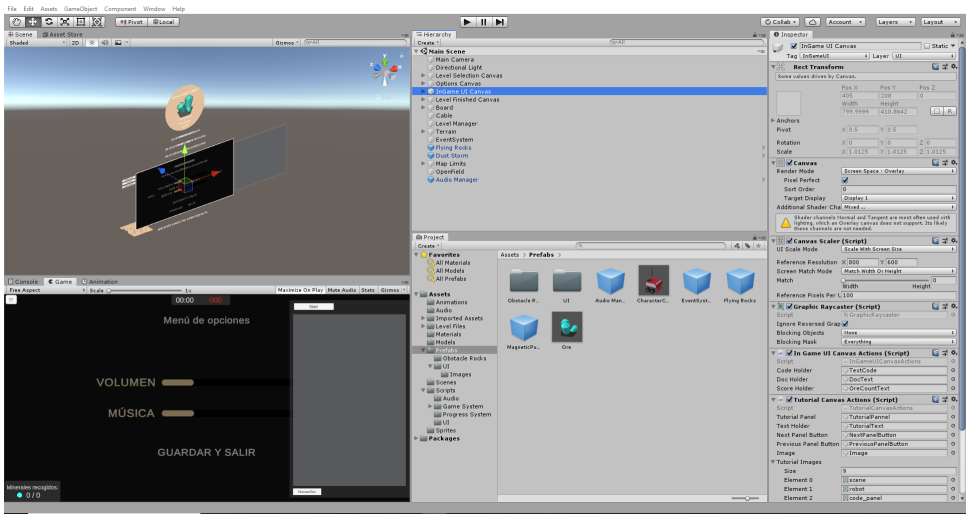


**Figura 1:** Editor de Unity.

Otras ventajas de Unity es que cuenta con una licencia gratuita (hasta un máximo de ingresos por su uso, a partir del cual tocaría pagar) y la extensa disponibilidad de documentación y formación gratuita en Internet.

La plataforma cuenta también con la denominada *Asset Store*. Esta es una tienda online en la que los desarrolladores de Unity pueden poner a la disposición de otros, bien mediante pago o de forma gratuita, cualquier elemento que hayan desarrollado para la plataforma.

Unity es una opción clara a la hora de elegir un motor de desarrollo para un proyecto de las características de éste. En cualquier caso, su uso es una decisión forzada por la plataforma *Programa Jugando*, que ha sido desarrollada con este motor, y que requiere que los juegos sean desarrollados con Unity y exportados como proyecto WebGL para su correcta integración.

## 2.2 GIT

Git es un sistema gratuito, de código abierto distribuido, para el control de versiones, diseñado para manejar desde pequeños proyectos a proyectos de gran escala de forma rápida y eficiente [4].

Control de versiones significa la gestión y almacenamiento de las distintas versiones del proyecto en desarrollo, junto a todo el historial de cambios, y los autores de los mismos, de tal forma que siempre podamos recuperar el estado en el que se encontraba el proyecto en un determinado momento.

Git es una herramienta indispensable para el desarrollo de software en equipo, pero también para el desarrollo en solitario, pues nos sirve para tener un historial de desarrollo y copias de seguridad a lo largo de todo el proceso.

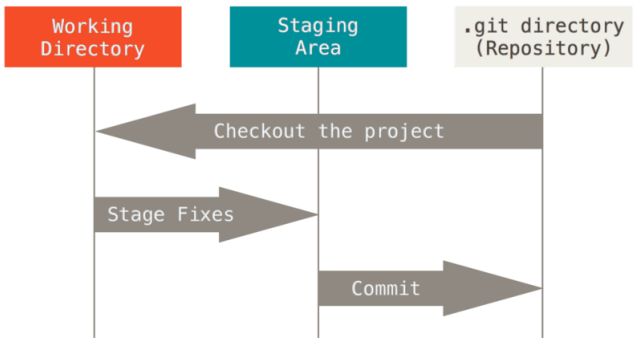


**Figura 2:** Flujo de trabajo en Git.

Para trabajar con Git, habitualmente tendremos dos copias del historial de desarrollo, una local, en nuestro ordenador, y otra remota, en un servidor. A estas copias nos referiremos como repositorio local y remoto, respectivamente (figura 2).

Mantener una copia remota sirve como mecanismo de seguridad, para que en caso de que perdiésemos la versión de nuestro ordenador, bien por un borrado accidental, bien por una avería del dispositivo, siempre tengamos la copia remota como respaldo.

Para este proyecto, utilizaremos la plataforma GitHub[5] como repositorio remoto.

Git permite la creación de varias líneas temporales de desarrollo, de tal forma que podemos tener varias versiones del proyecto en desarrollo sobre las que se realizan cambios de forma paralela. A estas líneas temporales se las denomina "ramas". Para este proyecto utilizaremos la siguiente organización en ramas del repositorio:

- Rama máster: Es la rama principal del proyecto. Aquí encontraremos la última versión ejecutable y probada del proyecto.
- Rama de desarrollo: Es la rama que contiene la versión en desarrollo actual del proyecto. Cuando en esta rama se completan los cambios de la iteración de desarrollo actual, se realizan las pruebas necesarias para confirmar que el funcionamiento es correcto, y entonces, los cambios se trasladan a la rama máster, "fusionando" esta rama en ella.
- Rama de tarea: Es la rama que contiene los cambios asociados a una tarea particular. Estos cambios pueden ser una nueva funcionalidad para el proyecto, una actualización de una funcionalidad existente o una corrección de un defecto. Las tareas son ítems definidos por la metodología de trabajo empleada en el desarrollo, que veremos más adelante, en el capítulo 3. Una vez completado el desarrollo de una tarea, los cambios se fusionan sobre la rama de desarrollo.

Aunque es posible albergar un proyecto Unity en un repositorio de Git, éste puede contener archivos de gran peso que sobrepasen el límite de capacidad establecido por nuestro repositorio remoto. Para evitar problemas, utilizaremos un archivo *.gitignore* que facilita el repositorio oficial de GitHub [6], que restringe de forma automática qué archivos no se guardarán en nuestro repositorio. Debido a esto, no bastará con descargar la versión de la rama máster del proyecto para tener una copia completa del proyecto, sino que se deberá entregar una copia comprimida completa también. Se acabará de detallar este aspecto en la sección de Entregables del capítulo 3.

## 2.3 BLENDER

Blender (figura 3) es una suite de creación 3D gratuita y de código abierto que soporta un *pipeline* completo de modelado 3D, *rigging*, animación, simulación, renderizado, comoposición, *motion tacking*, edición de vídeo y animación 2D [7].

Para el proyecto, usaremos Blender como herramienta de modelado 3D de algunos de objetos de la escena de juego.

Blender tiene la ventaja de tener compatibilidad con Unity en cuanto a que cuando modifiquemos un objeto 3D bien desde el editor de Blender, bien desde el editor de Unity, veremos los cambios reflejados correctamente desde el otro.

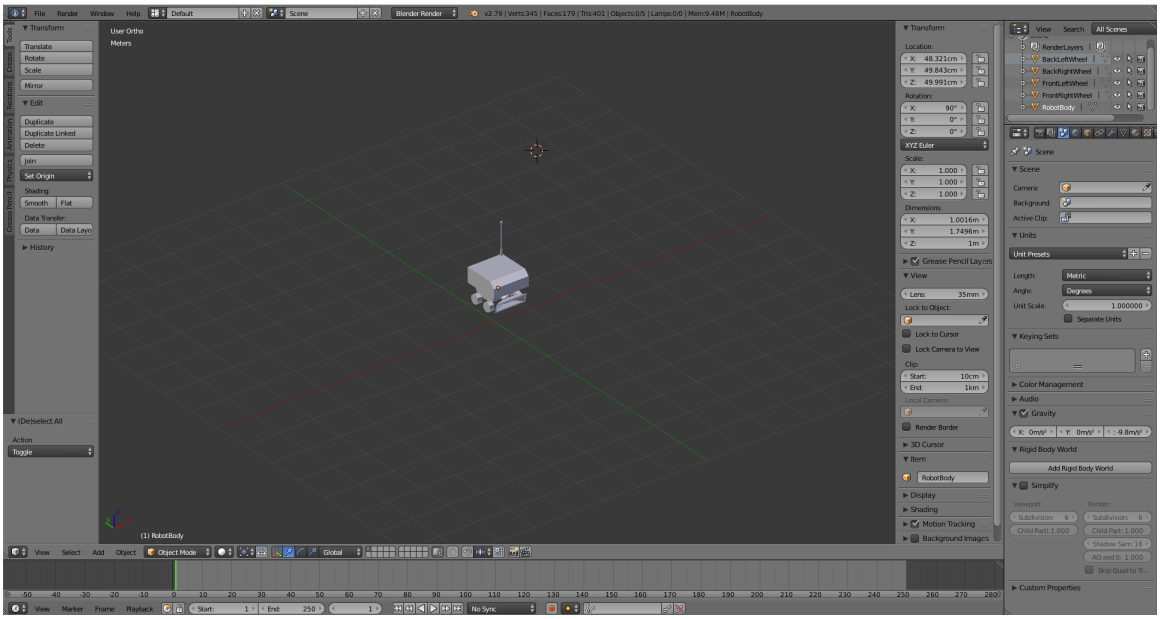

**Figura 3:** Editor de Blender.

# 3 PLAN DE PROYECTO

En este capítulo analizaremos los objetivos del proyecto, la metodología empleada durante su desarrollo, el seguimiento del mismo y, finalmente, un análisis de sus posibles riesgos.

## 3.1 ENTREGABLES

A continuación, se listan los objetos de desarrollo que deben ser entregados para considerar el proyecto finalizado:

- Documento de Diseño de Juego (GDD): Documento descriptivo que detalla las características de los videojuegos. Este documento se recoge en el capítulo 4 de esta memoria.
- Análisis de requisitos: Expone los requisitos técnicos que el proyecto deberá cumplir, de acuerdo a lo indicado en el documento de diseño. Este documento se recoge en el capítulo 5 de esta memoria.
- Análisis de riesgos: Recoge un listado de riesgos predichos para el desarrollo del proyecto, indicando su probabilidad de ocurrencia, y una serie de medidas preventivas para cada uno. Este documento forma parte del capítulo 3 de esta memoria.
- Código compilado de los videojuegos (*build*): Se entregará la versión final compilada de los dos videojuegos desarrollados, exportándola a la plataforma Programa Jugando.
- Código fuente de los videojuegos: Se entregará también el código fuente, para posibilitar su modificación en el futuro.

- Documentación para la modificación de puntuaciones: Se incluirá un documento que explique cómo modificar el cálculo de las puntuaciones en el código fuente. La modificación de las puntuaciones puede ser requerida por el cliente en caso de que, tras haber sido puesto el proyecto en fase de prueba por su usuario final (alumnos), resulte que la dificultad para superar los niveles sea demasiado alta o demasiado baja. El documento indica dónde y cómo modificar estos cálculos para ajustar la dificultad de los videojuegos. Esta documentación aparecerá como una sección del fichero *README.md* incluido en el repositorio del código fuente del proyecto.

- Documentación para la edición de niveles: La implementación de ambos videojuegos permite editar o crear nuevos niveles de manera eficiente, sin necesidad de usar el editor de Unity o conocer todos los detalles de la implementación. Esto da la posibilidad de expandir en un futuro el tamaño de los videojuegos entregados (Modificar el lenguaje de programación mostrado al jugador, añadir niveles para una dificultad, etc.). Esta documentación aparecerá como una sección del fichero *README.md* incluido en el repositorio del código fuente del proyecto.

## 3.2 METODOLOGÍA

Para el desarrollo del proyecto se ha utilizado la metodología *Kanban* [8], que se engloba dentro de las metodologías ágiles. La implementación de la metodología consiste en el uso de un tablero (figura 4), que será visible para todos los integrantes del proyecto, en el que se listan tareas, su estado de progreso, y las personas implicadas en dicha tarea. Cada estado de progreso será una columna del tablero, en el que podrá haber un número limitado o ilimitado de tareas:

- En reserva: Tareas que se han creado pero no se espera comenzar a trabajar en ellas en el futuro cercano.

- Por hacer: Tareas que están previstas iniciarse próximamente, ordenadas por prioridad.

- En progreso: Tareas actualmente en progreso. Como el equipo de desarrollo está formado por una sola persona, sólo podrá haber una tarea en esta columna simultáneamente.

- Bloqueado: Tareas cuyo desarrollo se ha pausado temporalmente, bien por un problema que hay que investigar, la necesidad de completar otra tarea antes de poder acabar esta u otra causa técnica.

- Completado: Aquí tendremos todas las tareas que se han realizado con éxito.

- Archivado: Tareas que se han descartado. Las mantenemos aquí en lugar de borrarlas, para poder tenerlas de referencia o “rescatarlas” dada la circunstancia.

Una tarea describe el trabajo que se debe realizar sobre el proyecto, normalmente enfocado a un área determinada del mismo. Al tratarse de un proyecto unipersonal, la generación de las tareas recae sobre el propio desarrollador. Las tareas serán definidas a partir de los requisitos del proyecto dados por el cliente y los requisitos internos del propio desarrollador.

Esta metodología se integrará también con el manejo del repositorio Git. Para cada tarea, se deberá crear una rama nueva, a partir de la rama de desarrollo. Una tarea no se podrá mover a la columna de 'completado' hasta que no se hayan realizado las pruebas unitarias que confirmen que el funcionamiento de los cambios implementados es correcto y se fusione la rama de la tarea con la rama de desarrollo.

El tablero utilizado para la gestión del proyecto, en este caso, ha sido un tablero digital de la aplicación Trello[9]. En la figura 4 se muestra una captura de pantalla del mismo, en un punto intermedio del desarrollo del proyecto.

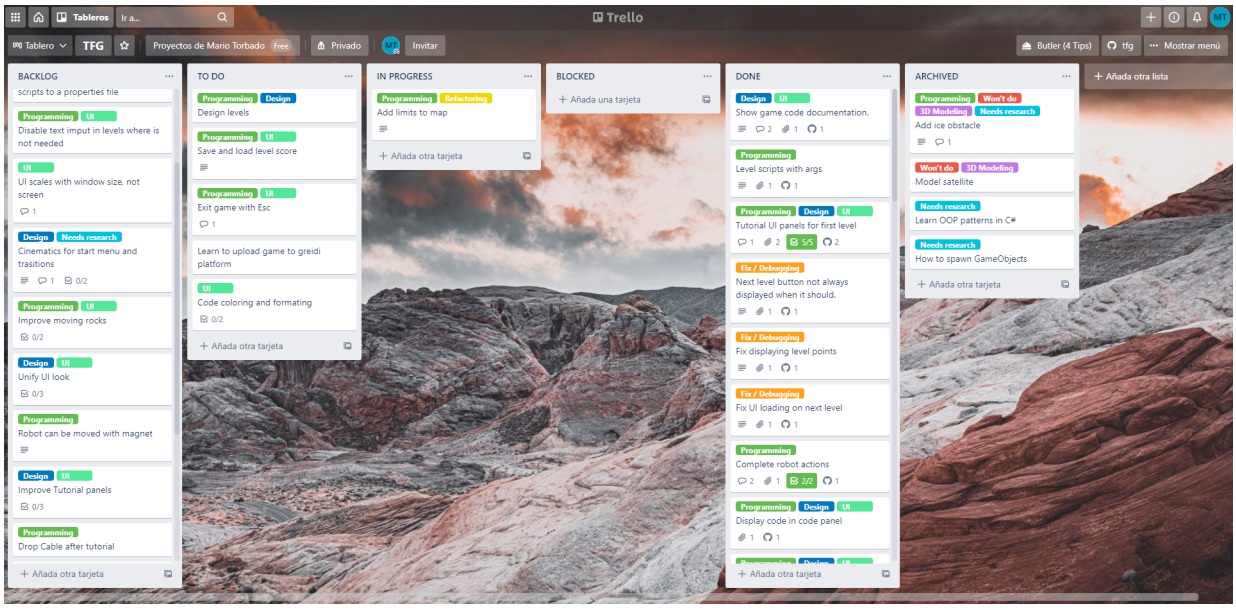


**Figura 4:** Tablero Kanban de Trello.

## 3.3 SEGUIMIENTO

*Kanban*, la metodología seleccionada y descrita en la sección anterior, no establece pautas de seguimiento de proyecto entre cliente y desarrolladores. Esto es así ya que *Kanban* está pensado para poder integrarse en un sistema de producción existente, u organizar a un equipo de desarrollo internamente, independientemente de la jerarquía de mando que exista por encima.

Los clientes del proyecto desarrollado son, en este caso, los mismos tutores, Doña Alma María Pisabarro Marrón y Don Carlos Enrique Vivaracho Pascual. Nos referiremos a ellos como el 'cliente' en este documento de aquí en adelante.

Como pauta de seguimiento informal, se establecerá la realización de reuniones del desarrollador con el cliente cada dos semanas, en la medida de lo posible. Dichas reuniones tienen como función informar al cliente de los cambios implementados en la última iteración, definir nuevos objetivos y requisitos para el proyecto y corregir la trayectoria del mismo en caso de desviarse del objetivo pretendido.

Estas reuniones se han llevado a la práctica con la regularidad predicha, desde el comienzo del proyecto en septiembre de 2019, a excepción del periodo entre junio de 2020 y marzo de 2021, meses durante los cuales el desarrollo del proyecto se detuvo, debido a la inviabilidad de congeniar su desarrollo con la vida laboral del desarrollador durante dicho periodo.

## 3.4 GESTIÓN DE RIESGOS

Un riesgo es la probabilidad de ocurrencia de un suceso en el futuro que afectaría negativamente a la capacidad para cumplir los objetivos del proyecto. Para gestionar los posibles riesgos a los que se enfrenta el proyecto, se ha seguido un proceso cuyos pasos se describen a continuación:

- Identificación: Se identifican y se enumeran los riesgos posibles para el desarrollo del proyecto.
- Análisis: En esta etapa se analiza cada riesgo y se formaliza un plan de actuación. Este plan consiste en la especificación de unas medidas de prevención, para evitar que la situación del riesgo se haga realidad, y unas medidas de contingencia, para que, en caso de que suceda, minimizar el impacto del mismo.
- Priorización: Se ordenan los riesgos según su importancia. A mayor probabilidad e impacto, mayor importancia.

En tablas 2 a 8 , se enumerarán los riesgos del proyecto, obtenidos mediante el proceso anteriormente indicado, utilizando la plantilla expresada en la tabla 1:

| | |
|---|---|
| Identificador | Identificador con forma R-0 |
| Riesgo | Descripción del riesgo |
| Probabilidad | Estimación de la probabilidad de que el riesgo suceda. Puede ser: raro, poco probable, posible, muy probable o casi seguro |
| Consecuencia | Descripción del resultado de suceder el riesgo |
| Impacto | Indicador de la gravedad de las consecuencias de la ocurrencia del riesgo. Puede ser: despreciable, bajo, moderado, alto o catastrófico. |
| Medidas de prevención | Acciones para evitar que el riesgo suceda. |
| Medidas de contingencia | Acciones para minimizar el impacto del riesgo. |

**Tabla 1:** Plantilla de riesgo.

| | |
|---|---|
| Identificador | R-1 |
| Riesgo | Disponibilidad insuficiente del desarrollador. |
| Probabilidad | Muy probable, por la conciliación con la vida laboral del desarrollador. |
| Consecuencia | El desarrollador no puede dedicar el tiempo necesario al desarrollo del proyecto para terminarlo en la fecha estimada de entrega inicial. |
| Impacto | Alto |
| Medidas de prevención | Ninguna. |
| Medidas de contingencia | Retrasar la entrega del proyecto, solicitar excedencia del puesto de trabajo para terminar el proyecto. |

**Tabla 2:** Riesgo R-1

| | |
|---|---|
| Identificador | R-2 |
| Riesgo | Pérdida del trabajo realizado por fallo del dispositivo de almacenamiento o borrado accidental. |
| Probabilidad | Posible |
| Consecuencia | Necesidad de rehacer el trabajo perdido y/o coste monetario para recuperar el material de trabajo. |
| Impacto | Alto |
| Medidas de prevención | Utilización de un repositorio remoto de una entidad de confianza (GitHub) y copias de seguridad en discos duros externos. |
| Medidas de contingencia | Retrasar la entrega del proyecto, incluir en el presupuesto del proyecto una reserva para averías técnicas. |

**Tabla 3:** Riesgo R-2

| | |
|---|---|
| Identificador | R-3 |
| Riesgo | Avería del equipo de trabajo |
| Probabilidad | Posible |
| Consecuencia | Parada temporal en el desarrollo del proyecto y/o coste monetario para recuperar el equipo de trabajo. |
| Impacto | Moderado. |
| Medidas de prevención | Disponer de un equipo secundario con el que poder continuar el desarrollo. |
| Medidas de contingencia | Retrasar la entrega del proyecto, incluir en el presupuesto del proyecto una reserva para averías técnicas, trasladar el desarrollo del proyecto a un equipo secundario. |

**Tabla 4:** Riesgo R-3

| | |
|---|---|
| Identificador | R-4 |
| Riesgo | Formación insuficiente. |
| Probabilidad | Posible |
| Consecuencia | Necesidad de pausar el desarrollo para aprender cómo implementar un requisito del proyecto. |
| Impacto | Moderado o bajo. |
| Medidas de prevención | Realizar una formación extensa en las herramientas necesarias para el desarrollo del proyecto de forma previa a iniciar su desarrollo, y reservar un margen de tiempo en la estimación del proyecto para completar el aprendizaje que fuera necesario. |
| Medidas de contingencia | Completar el aprendizaje necesario imprevisto, buscar una solución diferente, retrasar la entrega del proyecto. |

**Tabla 5:** Riesgo R-4

| | |
|---|---|
| Identificador | R-5 |
| Riesgo | Disponibilidad insuficiente del cliente (tutores). |
| Probabilidad | Poco probable. |
| Consecuencia | La comunicación es insuficiente y no se pueden establecer con precisión los requisitos del proyecto. La implementación puede no ajustarse a las necesidades del cliente. |
| Impacto | Alto |
| Medidas de prevención | Comunicación frecuente con el cliente y empleo de técnicas de desarrollo ágil e iterativo. |
| Medidas de contingencia | Ninguna. |

**Tabla 6:** Riesgo R-5

| | |
|---|---|
| Identificador | R-6 |
| Riesgo | Cambios críticos en los requisitos del proyecto. |
| Probabilidad | Poco probable. |
| Consecuencia | Parte del trabajo realizado debe descartarse y se necesitará más tiempo para implementar los cambios acordados. |
| Impacto | Moderado |
| Medidas de prevención | Comunicación frecuente con el cliente y empleo de técnicas de desarrollo ágil e iterativo. |
| Medidas de contingencia | Retrasar la entrega del proyecto. |

**Tabla 7:** Riesgo R-6

| | |
|---|---|
| Identificador | R-7 |
| Riesgo | Enfermedad o Incapacitación del desarrollador. |
| Probabilidad | Raro. |
| Consecuencia | El desarrollador no puede continuar trabajando el proyecto, de forma temporal, permanente o no al mismo ritmo que inicialmente. |
| Impacto | De moderado a catastrófico. |
| Medidas de prevención | Ninguna. |
| Medidas de contingencia | Retrasar la entrega del proyecto. |

**Tabla 8:** Riesgo R-7

## 3.5 PRESUPUESTO

El salario anual bruto medio de un programador de videojuegos en España es de 32.100€, según la web Jobted [10]. Para una estimación inicial de 300 horas de trabajo real para la realización del proyecto, si equiparamos esto a una jornada habitual de 40 horas semanales, necesitaríamos aproximadamente 2 meses para realizar el proyecto. Por tanto, tendríamos un coste de mano de obra de (32.100/12)*2 = 5.350€ brutos.

En cuanto al software utilizado, al haber optado por herramientas de código abierto o con licencias gratuitas, suficientes para cubrir el desarrollo de este proyecto, su coste total asociado es de 0€.

El hardware utilizado tiene un coste aproximado de 1.000€. Se trata de un equipo cuyo motivo de adquisición y uso no es exclusivamente para el desarrollo del proyecto. Contabilizaremos el coste de amortización para las 300 horas de trabajo de la siguiente manera:

Según la Tabla de coeficientes de amortización lineal de la Agencia Tributaria, para Equipos Informáticos de Procesos de Información, se establece un coeficiente lineal máximo del 25 % anual [11]. Considerando años de 236 días laborales, con jornadas de 8 horas durante las cuales el equipo estaría funcionando de forma continua, tendríamos un coste de 1000*0,25*(300/(236*8))= 39,72€ asociado al uso del hardware durante la realización del proyecto.

Para el cálculo del presupuesto se han desestimado otros costes, como el asociado al consumo eléctrico o el correspondiente a la tarifa de la compañía que suministra acceso a la Red.

Teniendo en cuenta todo lo mencionado, el presupuesto para la realización de este proyecto ascendería a 5.389,72€.

# 4

# DOCUMENTO DE DISEÑO DE JUEGO (GDD)

## 4.1 QUÉ ES UN GDD

GDD, del Inglés *Game Design Document*, o Documento de Diseño del Juego, es un documento estándar de la industria de los videojuegos que tiene como fin plasmar en texto el juego en desarrollo, desde su conceptualización general a detalles sobre mecánicas de juego, historia, personajes o diseño artístico [12].

Es un documento de carácter informal, sin estructura definida, y que se actualiza conforme avanza el desarrollo del juego.

La función principal de este documento es la de servir como guía y referencia entre los distintos departamentos y miembros de un equipo de desarrollo (diseñadores, programadores, artistas, marketing, etc.) y, por lo tanto, es un documento en el que todas las partes del equipo están involucradas en su edición.

Debido a la naturaleza dinámica del documento, es frecuente que sea utilizado como ítem de seguimiento en técnicas de desarrollo ágil: Una versión actualizada del documento es generada por el equipo de diseño, que debe ser aprobada por el editor o cliente. Cuando este da el visto bueno, el documento pasa a manos del equipo de desarrollo, que detalla la implementación de los cambios. Por último, el documento pasa de nuevo por el equipo de diseño, se revisa, y comienza una nueva iteración.

En este caso, al tratarse de un proyecto en el que el autor ocupa todos los roles del desarrollo del proyecto, el GDD es más un documento de presentación del mismo de cara al cliente, que una herramienta de comunicación interna para el desarrollo.

## 4.2 INTRODUCCIÓN

*Mars Miner* y *Venus Miner* son dos videojuegos educativos cuyo fin es el de mejorar el aprendizaje de los alumnos en conceptos básicos de programación y sus destrezas a la hora de utilizarlos.

En cada nivel de cada juego, el objetivo del jugador será guiar a un robot que se desplaza por la superficie de un planeta para que recoja un número de minerales valiosos.
Para que el robot consiga recoger los minerales, el jugador deberá analizar un fragmento de código que se mostrará en la pantalla, que representa las acciones que realizará el robot de manera automática. Después, el jugador podrá manipular la escena moviendo algunos obstáculos (rocas), para que el robot logre llegar a su objetivo. Si el robot colisiona con alguna roca, se sale de la zona de juego o completa la secuencia de código sin haber recogido los minerales necesarios, el jugador habrá perdido en dicha pantalla y no obtendrá puntos. Si por el contrario, logra recoger los minerales necesarios y completa la secuencia de código sin colisionar, el jugador habrá completado la pantalla y obtendrá una cantidad de puntos determinada por la dificultad del nivel y el tiempo e intentos que ha necesitado para completarlo.

*Mars Miner* y *Venus Miner*, aunque se entregan con el proyecto como dos juegos independientes, son en esencia el mismo, diferenciándose únicamente en aspectos estéticos y el tipo de código a analizar. Por este motivo se presentan ambos con el mismo Documento de Diseño y será en las últimas secciones donde se tratarán sus diferencias.

## 4.3 AUDIENCIA Y PLATAFORMA

*Mars Miner* y *Venus Miner* forman parte del proyecto de gamificación de la asignatura de Fundamentos de Programación, y su audiencia son Alumnos de primer curso de Grado de Ingeniería Informática.

Ambos juegos serán exportados para ser ejecutados desde la plataforma web *Programa Jugando* por lo que están destinados a ser jugados desde un navegador web en un ordenador de sobremesa o portátil.

## 4.4 TEMÁTICA

*Mars Mine*r y *Venus Miner* no cuentan con una historia o trasfondo explícito, pero sus diseños giran en torno a la exploración espacial y la explotación de recursos materiales de otros planetas. Del nombre, *Mars* y *Venus* hacen referencia respectivamente al planeta en el que se encuentra la escena de juego (Marte o Venus), y *Miner* (minero) a la labor de extraer minerales.

En el juego, el "protagonista" es un robot, de diseño inspirado en vehículos de exploración marciana reales como el *Curiosity* o el *Perseverance*. A dicho robot se referirá en el juego como el *Mars Hoover* o *Venus Hoover*, haciendo de nuevo referencia al planeta escenario, y *Hoover*, aspiradora en inglés. El robot por tanto será una especie de aspiradora autónoma que recoge minerales valiosos de la superficie de los planetas.

## 4.5 CÓMO SE JUEGA

Al comenzar el nivel, al jugador se le presentará la escena de juego (figura 5), que representa la superficie de un planeta, sobre la que se sitúa el tablero de juego. En él encontraremos los siguientes elementos de juego, ocupando una casilla:

- Mars/Venus Hoover: El robot que debe recoger los minerales.
- Rocas marrones: Son rocas fijas en el tablero, que no se pueden mover.
- Rocas plateadas: Son rocas que se pueden mover.
- Minerales: Gemas brillantes de minerales valiosos. El robot debe tratar de recogerlas.

Por otro lado, a la derecha de la pantalla de juego, se mostrará el código con las acciones del robot, y un botón para iniciar la secuencia.

En la esquina inferior izquierda aparecerá un marcador que indica el número de minerales que debe recoger el robot para superar la pantalla actual, y el número de minerales recogidos hasta el momento.

El jugador deberá analizar el código del robot y, viendo en la escena la posición y orientación inicial del mismo, prever cómo se moverá una vez iniciada la secuencia. El jugador entonces tratará de mover las rocas plateadas hasta llegar a una configuración que evite que el robot choque con alguna roca y que guíe al robot hasta los minerales.

Una vez el jugador esté convencido de que su configuración servirá para guiar correctamente al robot, pulsará un botón que inicie la secuencia del código del robot.

Una vez iniciada la secuencia, no se podrá parar. Esto es así para incrementar el énfasis en el análisis previo del código y no favorecer el realizar modificaciones sobre la marcha. Sin embargo, el jugador sí podrá mover rocas plateadas después de iniciar la secuencia, con el fin de evitar frustración por ver un fallo después de iniciar la secuencia o tener que esperar a que termine a sabiendas de que se fallará el nivel.

Como el jugador deberá corregir sus fallos a toda prisa mientras el robot se mueve, será más gratificante el completar una pantalla de esta forma debido a la mayor agilidad requerida para ello.

En la pantalla de juego aparecerá también un temporizador que marca el tiempo que llevamos en la pantalla actual, y un contador con la penalización de puntuación por tiempo. La penalización por tiempo y el hecho de que esta se muestre por pantalla sirve como incentivo a tomar riesgos y a no sobre-analizar el código, a la vez que incentiva la situación de que el jugador tenga que mover rocas después de iniciar la secuencia de código, todo con el objetivo de hacer la experiencia de juego más interesante.

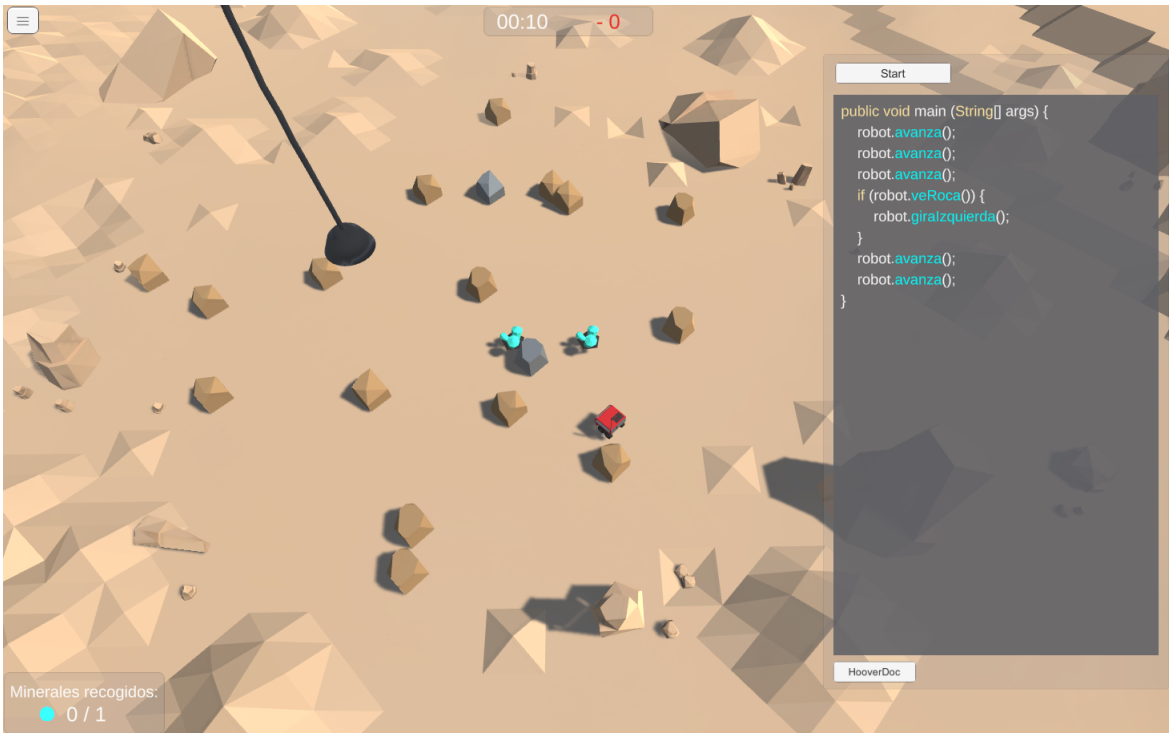


**Figura 5:** Escena de juego.

## 4.6 NIVELES

Los juegos contarán con 4 niveles de dificultad: fácil, medio, difícil y reto. Para cada nivel de dificultad, habrá un un número de escenarios de juego posibles. Entenderemos por escenario a una configuración concreta de elementos de juego y su posición en el tablero y de script de movimiento para el robot.

Los escenarios para cada nivel se cargarán de manera aleatoria al elegir una dificultad, a excepción de la primera vez que juguemos el nivel fácil, pues el primer nivel está pensado para servir como tutorial. La pantalla del tutorial muestra una serie de ventanas de texto que explican cómo se juega.

Cuando un jugador completa con éxito un escenario de juego, el jugador podrá continuar jugando escenarios de la misma dificultad, o si ha superado un umbral de puntuación, podrá pasar al siguiente nivel de dificultad. Si el jugador falla un nivel, podrá reintentar el mismo escenario o uno distinto de la misma dificultad. De esta forma, si el jugador se atasca, no se ve forzado a reintentar el mismo escenario, lo que podría frustrarle.

La primera vez que el jugador entra al juego, sólo podrá seleccionar el primer nivel, y se cargará la pantalla de tutorial. Conforme el jugador obtenga puntos, se desbloquearán los siguientes niveles.

## 4.7 MECÁNICAS DE JUEGO

En esta sección analizaremos las posibilidades creativas de los escenarios de juego, y cómo hacer que estas sean interesantes para el jugador. Para ello, empezaremos por analizar el conjunto de movimientos del robot, que se definirá en forma de métodos. Este conjunto es bastante limitado, pero suficiente para generar una variedad de niveles suficiente utilizando otros recursos, como veremos más a delante. Estos son los métodos disponibles del robot:

- robot.avanza(): Desplaza el robot 1 casilla hacia delante.
- robot.retrocede(): Desplaza el robot 1 casilla hacia atrás.
- robot.giraDerecha(): Gira el robot 90º hacia su derecha.
- robot.giraIzquierda(): Gira el robot 90º hacia su izquierda.
- robot.veRoca(): Comprueba si hay una roca en la casilla directamente frente al robot.

- robot.veRoca(int dist): Comprueba si hay una roca en una casilla frente al robot a la distancia indicada (dist).
- robot.cogeMineral(): Comprueba si el robot ha recogido un mineral en la casilla actual.

Los 4 primeros son procedimientos (*void*) que permiten el robot se desplace por el tablero, mientras que los 3 últimos son funciones permiten conjugar condiciones (*boolean*), para añadir complejidad al código. En la figura 6 se muestra un código de ejemplo, como se vería en la pantalla de juego, que hace uso de todos los elementos:


**Figura 6:** Código del robot.

Los distintos elementos que inicialmente añaden variabilidad al diseño de niveles de juego son: la secuencia de movimiento incluida en el código, el posicionamiento de los minerales, el posicionamiento de las rocas movibles y el posicionamiento de las rocas fijas. Sin embargo, estos elementos de por sí no hacen interesante la resolución del nivel, pues el jugador sólo tendrá que estimar el trayecto del robot y apartar las rocas necesarias. Si abusamos en el uso de niveles basados en este principio, el jugador se dará cuenta y podrá superar todos los niveles de la misma forma: apartando las rocas plateadas de la zona de movimiento entre el robot y los minerales, sin prestar

atención al código.

Con el fin de evitar este comportamiento, a continuación se enumeran una serie de recursos adicionales para dotar al diseño de niveles de creatividad, incrementar la dificultad o hacer la solución de un nivel menos obvia:

- Varias opciones: Si por ejemplo, en un nivel el robot debe recoger solo un mineral para superarlo, y solo hay un mineral en el tablero, el camino que tomará el robot resultará bastante obvio. Para motivar que el jugador se fije en el código, podemos añadir otros minerales, de forma que el jugador tenga que plantear la trayectoria a cada uno como una posible solución, y analizar el código para saber cuál es la correcta.
- Límite de movimientos: Podemos usar un contador y bucles para limitar las casillas que puede desplazarse el robot. El jugador no sólo deberá encontrar una configuración que haga que el robot alcance la roca, sino que también tendrá que ajustarse al límite de movimientos. Una variación de esta estrategia es utilizar la función robot.cogeMineral() para restaurar el valor del contador cada vez que el robot recoja un mineral.
- Escenas semejantes: Podemos repetir el tablero de un nivel variando el código de movimiento. Obliga al jugador a fijarse en los detalles del código, ya que, de lo contrario, podría pensar que es el mismo nivel y usar una configuración errónea.
- Código semejante: Podemos repetir el código de movimiento en varios niveles, modificando únicamente el tablero. El jugador estará familiarizado con la secuencia de acciones del robot, pero deberá aplicarla a una nueva situación.
- Marcha atrás: Haciendo que el robot vaya marcha atrás, los sentidos de giro se invierten con respecto a la dirección de avance. Puede resultar más complicado interpretar la trayectoria del robot, añadiendo dificultad.
- Anidamiento: Anidando estructuras de código condicionales e iterativas, el jugador deberá tener especial cuidado al leer el código para observar las diferentes profundidades (corchetes y tabulados). Aumenta la dificultad.
- Condiciones inalcanzables: Podemos incluir en el código condiciones (sentencias *if*) que sabemos a priori que nunca se van a ejecutar. El jugador deberá percatarse de ello para saber qué trayectoria seguirá el robot.
- Roca multifunción: Por lo general, utilizaremos el método robot.veRoca() en una sentencia condicional que haga cambiar de dirección al robot antes de colisionar. El jugador, por lo tanto, tendrá que recurrir en varias pantallas a situar una roca plateada en cada lugar que quiera hacer girar al robot. Para hacer esto más interesante, podemos diseñar la escena de tal forma que haya una roca plateada menos que los giros necesarios para la trayectoria que resuelve el nivel, y la solución consista en colocar una roca de tal forma que se use para realizar dos giros.

- Patrón en rocas fijas: Para resolver el nivel, el robot debe utilizar unas rocas fijas para realizar los giros que completen la trayectoria que recoja los minerales. El reto para el jugador es detectar este patrón y guiar el robot hasta él con las rocas plateadas.
- Rocas plateadas insuficientes: En la escena hay menos rocas plateadas de las necesarias para que el robot complete la pantalla. Por lo tanto, deberá recolocar alguna roca después de iniciar la secuencia de movimiento. Este recurso se reserva para los niveles de mayor dificultad, pues puede resultar menos intuitivo para el jugador y requiere de cierta agilidad motora para colocar la roca a tiempo en la posición correcta.

Utilizando estas estrategias, o la combinación de varias, y las distintas estructuras de código secuenciales (sentencias), iterativas (*while, for*) y condicionales (*if, else if, else*) podremos definir escenarios que se ajusten a los distintos niveles de dificultad.

## 4.8 SISTEMA DE PUNTUACIONES

Al tratarse de un juego serio, debemos tener especial cuidado con la forma de evaluar y puntuar el progreso del jugador. Este sistema debe motivar al jugador, pero sin generar comportamientos que propicien la ludopatía o comportamientos competitivos negativos entre jugadores.

Para servir de herramienta educativa, el sistema deberá ser capaz también de indicar cuándo el jugador ha realizado un progreso suficiente para comprender los conceptos tratados. A continuación se detalla la implementación del sistema de puntuaciones de acuerdo a estos principios.

Como se ha mencionado previamente, el juego contará con 4 niveles de dificultad. Cada dificultad, independientemente del número de posibles escenarios que contenga, tendrá una puntuación máxima fija, y de igual manera, habrá una puntuación máxima que se podrá conseguir en el juego. Esto sirve dos propósitos: Igualar el "valor" del juego al resto de los juegos de la plataforma, y evitar el "pique" o comportamientos demasiado competitivos entre los alumnos que haga que jueguen solo a aquellos juegos que den más puntos.

En el juego, también se define un valor de puntuación que indica a la plataforma que el juego se ha "superado". Esto quiere decir que el jugador ha superado los suficientes escenarios de diferentes niveles de dificultad como para tener una comprensión de los conceptos fundamentales tratados en el juego. Para estos dos juegos, dicho valor estará calculado de tal modo que se alcanzará al jugar lo suficiente para desbloquear todos los niveles de dificultad (el jugador habrá superado todos los niveles, excepto el último, que podrá haber probado) y al haber logrado la puntuación máxima en alguno de ellos. Lograr la puntuación máxima de un nivel no es especialmente difícil, ya que el jugador podrá jugar los escenarios de un nivel tantas veces como desee, y la puntuación incrementará con la puntuación lograda en el escenario independientemente de si se ha jugado o superado

previamente o no.

Cada escenario de juego superado proporcionará como máximo una porción de los puntos de su nivel, restando puntos de penalización por tiempo e intentos. Ambas penalizaciones cuentan con un valor máximo, y se define también una puntuación mínima, de tal modo que el jugador siempre obtenga puntos al superar un nivel, independientemente del tiempo o intentos. Estas consideraciones, de nuevo, tratan de evitar de frustrar al jugador cuando se atasca en un nivel.

La penalización por tiempo se incrementará a intervalos regulares (cada 30 segundos en *Mars Miner* y cada minuto en *Venus Miner*), de tal forma que el jugador cuenta con un margen para pensar sin ser penalizado.

La penalización por intentos, sólo tiene en cuenta las veces que el jugador intenta el mismo escenario de forma consecutiva, y se reinicia al salir del juego o cambiar de escenario. De esta manera evitamos que la penalización sea permanente y el jugador no pueda mejorar su puntuación tras haber aprendido, debido a errores del pasado.

Todos los parámetros mencionados en el cálculo de las puntuaciones tienen valores definidos en las versiones entregadas de los juegos. Dichos valores se han ajustado tras la realización de pruebas unitarias, pero debido a la imposibilidad de ser testeado por el usuario final (alumnos) antes de la entrega, se incluirá con el proyecto una guía para ajustar dichos parámetros en un futuro. Por esto no se detallan dichos valores en esta sección, dejándolo para la sección 5.3, Pruebas.

## 4.9 MULTIMEDIA

Para el diseño de los juegos se han creado una serie de elementos multimedia originales. Estos elementos son modelos 3D, animaciones, efectos de partículas [13], y clips de sonido.

Los modelos 3D empleados en el juego (figura 7) han sido creados en Blender y posteriormente exportados a Unity. A continuación se listan los modelos creados:

- Cable electroimán (figura 7, primera imagen).
- 4 Modelos de rocas en total (figura 7, segunda imagen). Los 4 modelos se usan tanto para las rocas fijas como las movibles (plateadas), aplicando diferentes materiales (color) a los distintos *Prefabs* para cada tipo de roca, dentro del editor de Unity.
- Robot (figura 7, tercera imagen). El cuerpo y las ruedas se importan como modelos separados para las animaciones.
- Mineral (figura 7, cuarta imagen).

- Escena de juego (figura 7, quinta imagen).

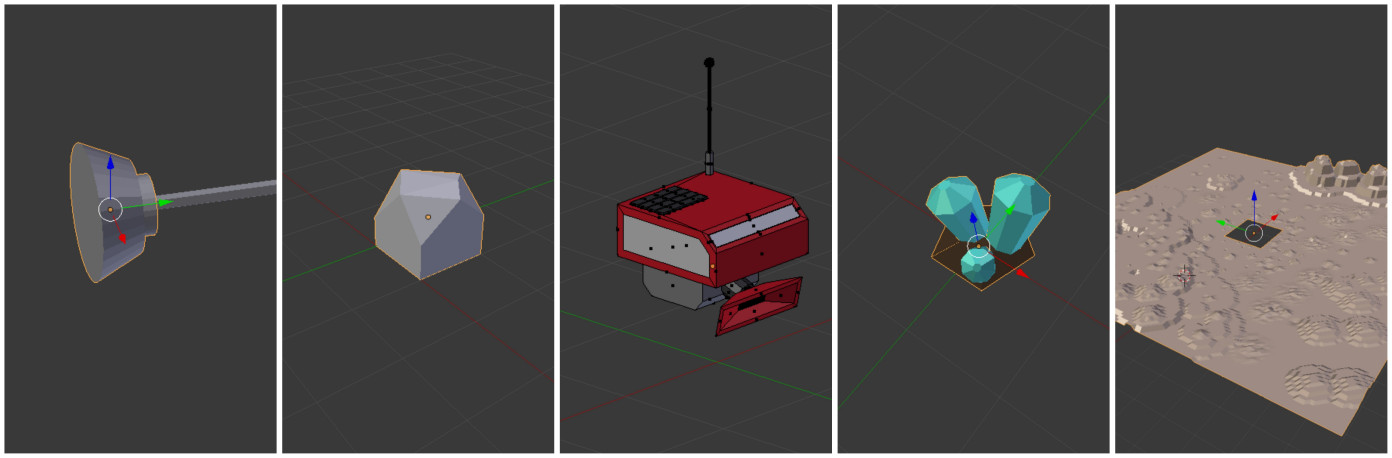

**Figura 7:** Modelos 3D.

Para las animaciones de movimiento y colisión del robot, se ha usado la herramienta de animación integrada en el editor de Unity.

Los efectos de movimiento de rocas, tormenta de arena y el humo de la animación de colisión del robot están creados con el sistema de partículas de Unity.

Los efectos de sonido provienen de clips extraídos de la plataforma *freesound.org* [14].

### 4.9.1 *Assets de terceros*

Para algunos elementos del juego se han utilizado los *Assets* de terceros listados a continuación:

- Rocas decorativas de la escena: *Low poly styled rocks*, de Daniel Robnik [15].
- Efecto tormenta de arena: *Stard Assets*, de Unity Technologies [16].
- Cuadrícula del tablero de juego: *SimpleGridShader*, de Revision3 [17].

La música del juego ha sido compuesta y producida por Don Óliver L. Sanz San José, antiguo alumno de la Escuela, permitiendo su uso de manera libre y gratuita en el juego. Desde aquí queremos agradecérselo.

## 4.10 MARS MINER

Este juego servirá para enseñar al alumno a entender estructuras condicionales e iterativas. El código del robot contendrá sólo un método *Main*, y podrá hacer uso de las siguientes instrucciones: *if, else, else if, while* y *for*.

Dada la ubicación del tema correspondiente en la asignatura, este juego será al que el alumno acceda primero, por lo que debe contar con un tutorial más completo, que detalle todos los aspectos necesarios para que el usuario aprenda a jugar.

Estéticamente, se diferenciará por la escena de juego, que emulará la superficie de Marte.

## 4.11 VENUS MINER

Este juego servirá para enseñar al alumno a utilizar métodos y funciones y a distinguir los conceptos de paso por referencia y paso por variable. El código del robot contendrá un método *Main* como antes, más las definiciones de métodos y funciones deseadas. A mayores, podrá hacer uso de las mismas instrucciones condicionales e iterativas que *Mars Miner*.

Otra diferencia clave es que, en el panel de código, aparecerá un campo de entrada, donde el jugador podrá introducir valores que corresponderán a algunos de los parámetros utilizados por el código. Esto añade otra variable que podremos tener en cuenta en el diseño de niveles.

Está versión se jugará después de *Mars Miner*, por lo que en el tutorial no será necesario explicar con tanto detalle los detalles técnicos del juego.

Estéticamente, se diferenciará por la escena de juego, que emulará la superficie de Venus.

## 4.12 REQUISITOS DE APRENDIZAJE PREVIOS

Ambos juegos están pensados para ser jugados después de que los conceptos tratados en ellos sean explicados en las aulas. Así pues, el jugador deberá conocer las sentencias condicionales e iterativas antes de jugar *Mars Miner*. Lo mismo se aplica para *Venus Miner* respecto al paso por referencia/variable, funciones y métodos.

Los juegos sirven como una herramienta para el refuerzo de estos conceptos y su puesta en práctica, pero en ningún momento sustituyen la labor docente de los profesores.

# 5

# ESPECIFICACIÓN TÉCNICA Y DESARROLLO

## 5.1 ANÁLISIS DE REQUISITOS

En esta sección se presentan las necesidades externas de cliente sobre los juegos, así como las necesidades internas, producto del diseño de la jugabilidad de los mismos.

Separaremos los requisitos en funcionales y no funcionales, siendo los primeros los que marcan el comportamiento del software, y los segundos, los que corresponden a atributos de calidad, características de funcionamiento o limitaciones.

Los requisitos enumerados a continuación corresponden a los aplicados a la versión final de los juegos, si bien estos han evolucionado a lo largo de las iteraciones de diseño llevadas al cabo durante la realización del proyecto.

### 5.1.1 *Requisitos Funcionales*

En esta sección, se define la lista de requisitos funcionales, que definen características requeridas en ambos juegos:

| RF-01 | Iniciar juego |
|---|---|
| Descripción | Al seleccionar el juego desde la plataforma, se deberá cargar el menú de inicio, que permitirá seleccionar un nivel de dificultad, ver las puntuaciones de cada nivel, cargar el menú de opciones y salir del juego. |

**Tabla 9:** Requisito Funcional RF-01

| RF-02 | Seleccionar un nivel |
|---|---|
| Descripción | El jugador podrá elegir entre 4 niveles de dificultad en el menú de inicio: fácil, medio, difícil y reto. El nivel Fácil siempre estará disponible, pero los siguientes niveles se desbloquearán al alcanzar una puntuación determinada en el nivel anterior. |

**Tabla 10:** Requisito Funcional RF-02

| RF-03 | Menú de opciones |
|---|---|
| Descripción | El menú de opciones permitirá ajustar los volúmenes de la música del juego y de los efectos de sonido por separado. A este menú se podrá acceder tanto desde la pantalla de juego como desde el menú de inicio. |

**Tabla 11:** Requisito Funcional RF-03

| RF-04 | Tutorial |
|---|---|
| Descripción | Al seleccionar el nivel fácil de dificultad, si el jugador no ha conseguido aún puntos en el juego (o lo que es lo mismo, no ha superado aún ninguna pantalla), se cargará una pantalla definida que contiene el tutorial de juego, en forma de paneles con texto explicativo e imágenes. |

**Tabla 12:** Requisito Funcional RF-04

| RF-05 | Pantallas aleatorias |
|---|---|
| Descripción | La pantalla mostrada cada vez que se juegue un nivel de dificultad, se seleccionará de manera aleatoria, a excepción de la pantalla para el tutorial. Al continuar jugando un nivel, la siguiente pantalla seleccionada nunca será la misma que la anterior. |

**Tabla 13:** Requisito Funcional RF-05

| RF-06 | Escena de juego. |
|---|---|
| Descripción | En la pantalla de juego, visualizaremos la escena que contiene el tablero de juego desde una perspectiva elevada, con el tablero situado en el centro de la pantalla. |

**Tabla 14:** Requisito Funcional RF-06

| RF-07 | Panel de código. |
|---|---|
| Descripción | En la pantalla de juego, visualizaremos al lado derecho un panel que contenga el código que representa las acciones del robot. En caso de que el código no cupiese en el espacio disponible, este panel podrá desplazarse verticalmente (*scroll*) mediante el uso de la rueda del ratón. |

**Tabla 15:** Requisito Funcional RF-07

| RF-08 | HooverDoc. |
|---|---|
| Descripción | En la pantalla de juego, habrá un botón para mostrar el *HooverDoc*, un panel que contiene todas las acciones del robot (métodos) y una explicación de las mismas. |

**Tabla 16:** Requisito Funcional RF-08

| RF-09 | Mover Roca Plateada. |
|---|---|
| Descripción | En la pantalla de juego, el jugador podrá mover la posición de una roca plateada en el tablero de la escena, haciendo *click* sobre la misma, arrastrando con el ratón, y soltando en la posición deseada. La roca plateada no podrá soltarse en una casilla del tablero que ya esté ocupada por otro elemento de juego. |

**Tabla 17:** Requisito Funcional RF-09

| RF-10 | Cronómetro. |
|---|---|
| Descripción | En la pantalla de juego, en la parte superior, aparecerá un panel que marca el tiempo desde que el jugador inició la pantalla actual, y un marcador con los puntos de penalización por tiempo. |

**Tabla 18:** Requisito Funcional RF-10

| RF-11 | Contador de minerales. |
|---|---|
| Descripción | En la pantalla de juego, en la esquina inferior izquierda, aparecerá un panel que indica el número de minerales que el robot debe recoger para superar la pantalla y los minerales recogidos hasta el momento. Este contador se actualizará en el instante que el robot recoja un mineral. |

**Tabla 19:** Requisito Funcional RF-11

| RF-12 | Iniciar movimiento del robot. |
|---|---|
| Descripción | En la pantalla de juego, habrá un botón para iniciar la secuencia de movimiento del robot. Una vez iniciada, no podrá detenerse. |

**Tabla 20:** Requisito Funcional RF-12

| RF-13 | Menú de la pantalla de juego. |
|---|---|
| Descripción | En la pantalla de juego, en la esquina superior izquierda, habrá un botón que desplegará un menú con las siguientes opciones: Reiniciar la pantalla actual, ir al menú de opciones y volver a la pantalla de inicio. |

**Tabla 21:** Requisito Funcional RF-13

| RF-14 | Pantalla superada. |
|---|---|
| Descripción | Una pantalla de juego se superará cuando, al terminar la secuencia de código del robot, éste haya recogido una cantidad de minerales igual o superior a lo indicado en el contador de minerales. El robot recogerá un mineral al pasar por la casilla en la que éste se encuentre. |

**Tabla 22:** Requisito Funcional RF-14

| RF-15 | Pantalla fallada. |
|---|---|
| Descripción | Una pantalla de juego se fallará cuando el robot choque con una roca, se salga de los límites del tablero o complete su secuencia de código sin recoger los minerales suficientes. |

**Tabla 23:** Requisito Funcional RF-15

| RF-16 | Menú de pantalla superada. |
|---|---|
| Descripción | En la pantalla de juego, al superar una pantalla con éxito, se mostrará un panel que indique los puntos conseguidos, el tiempo que se ha tardado y los intentos realizados hasta completar la pantalla, y las siguientes opciones: Jugar otra pantalla del mismo nivel y volver al menú de inicio. En caso de haber conseguido suficientes puntos para desbloquear el siguiente nivel de dificultad, aparecerá también la opción de iniciar una pantalla del siguiente nivel. |

**Tabla 24:** Requisito Funcional RF-16

| RF-17 | Menú de pantalla no superada. |
|---|---|
| Descripción | En la pantalla de juego, al fallar una pantalla, se mostrará un panel con las siguientes opciones: Reintentar la pantalla actual, jugar otra pantalla del mismo nivel de dificultad y volver al menú de inicio. |

**Tabla 25:** Requisito Funcional RF-17

### 5.1.2 *Requisitos No Funcionales*

En esta sección, se define la lista de requisitos no funcionales, que definen características requeridas en ambos juegos:

| RNF-01 | Lenguaje de programación mostrado. |
|---|---|
| Descripción | El código mostrado en las pantallas de juego deberá utilizar sintaxis Java. |

**Tabla 26:** Requisito No Funcional RNF-01

| RNF-02 | Aspecto del juego en la plataforma. |
|---|---|
| Descripción | El juego deberá mostrarse correctamente al ser ejecutado desde la plataforma, en un navegador web moderno, desde un ordenador, de tal forma que la interfaz sea clara, se distingan todos los elementos que componen el juego y ninguno de estos aparezca cortado o en una posición incorrecta. |

**Tabla 27:** Requisito No Funcional RNF-02

| RNF-03 | Legibilidad. |
|---|---|
| Descripción | Todo el texto del juego deberá estar diseñado para favorecer su legibilidad, y deberá tener un tamaño de fuente similar al utilizado en la plataforma. |

**Tabla 28:** Requisito No Funcional RNF-03

| RNF-04 | Claridad del tablero |
|---|---|
| Descripción | Los elementos del tablero de juego deberán distinguirse claramente unos de otros, y las casillas del tablero deberán poder distinguirse con facilidad cuando se mueva una roca plateada de posición. |

**Tabla 29:** Requisito No Funcional RNF-04

| RNF-05 | Tiempo de respuesta |
|---|---|
| Descripción | Todas las acciones dentro del juego, tanto con la interfaz de usuario como con la manipulación de la escena de juego, deberán responder inmediatamente, sin retardos. |

**Tabla 30:** Requisito No Funcional RNF-05

| RNF-06 | Motor de desarrollo |
|---|---|
| Descripción | Los juegos serán desarrollados usando el motor Unity. |

**Tabla 31:** Requisito No Funcional RNF-06

| RNF-07 | Integración con la plataforma |
|---|---|
| Descripción | Los juegos se integrarán con la plataforma, exportando éstos para la plataforma WebGL, e incorporando un sistema para leer y guardar las puntuaciones de cada juego para cada usuario. |

**Tabla 32:** Requisito No Funcional RNF-07

### 5.1.3 *Casos de Uso*

Los siguientes diagramas sirven para representar gráficamente la interacción que el jugador podrá tener con los juegos. Esta interacción se representará aislando 4 escenas diferenciadas, que limitan la interacción que es posible dentro del juego en cada una: El menú de inicio (figura 8), la pantalla de juego (figura 9), el menú tras superar una pantalla (figura 10) y el menú tras fallar una pantalla (figura 11).

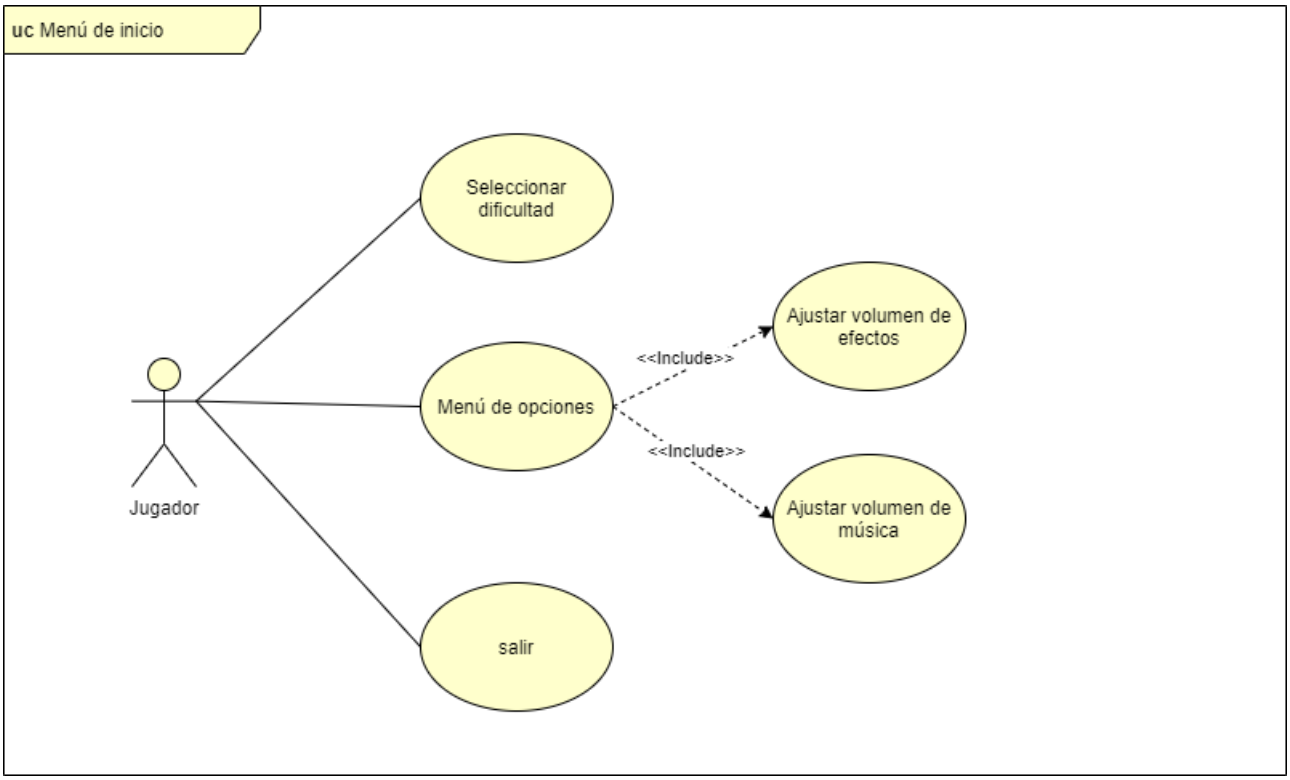


**Figura 8:** Diagrama de casos de uso: Menú de inicio.

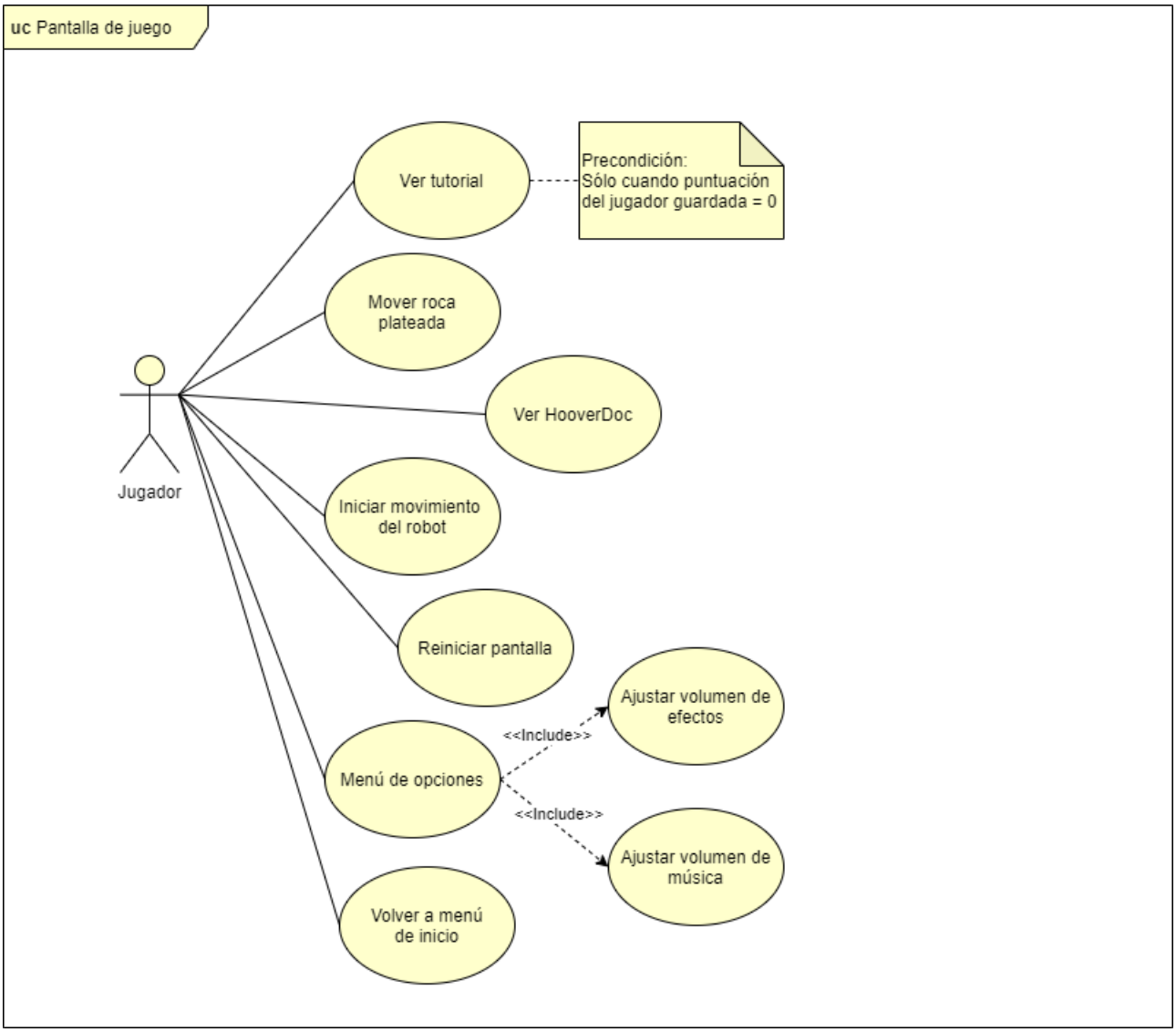


**Figura 9:** Diagrama de casos de uso: Pantalla de juego.

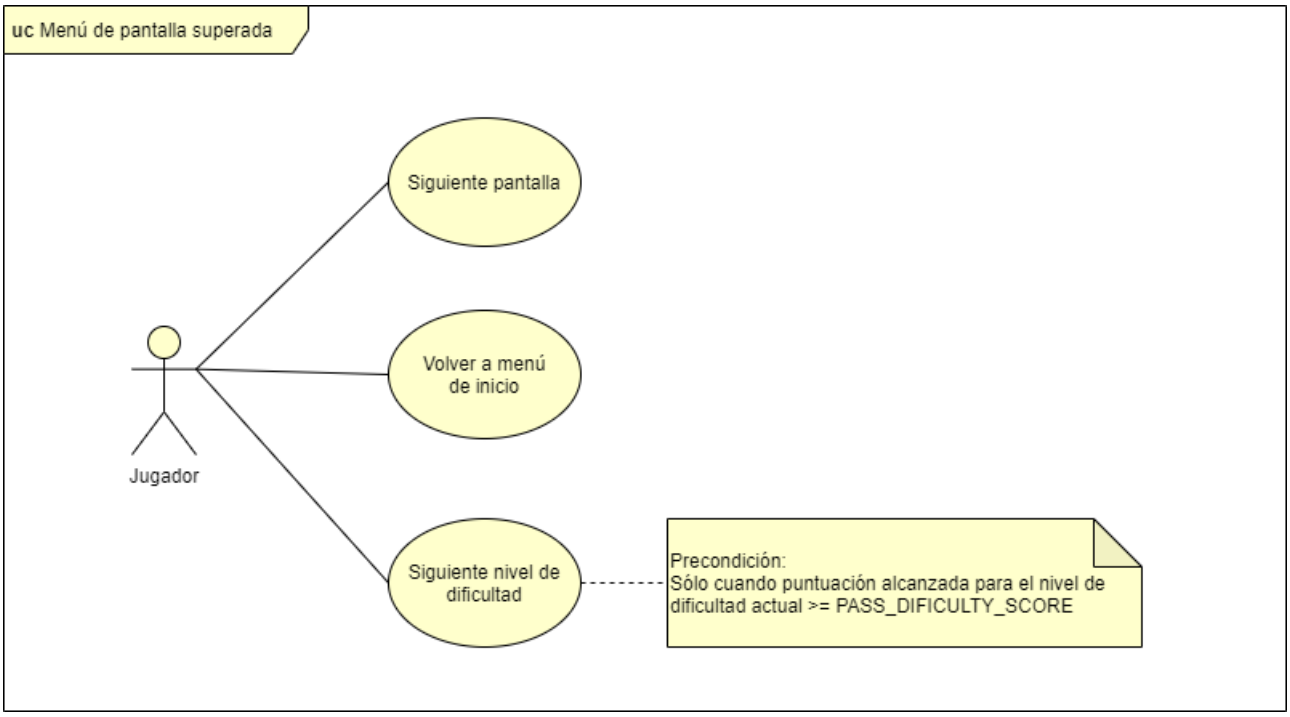


**Figura 10:** Diagrama de casos de uso: Menú de pantalla superada.

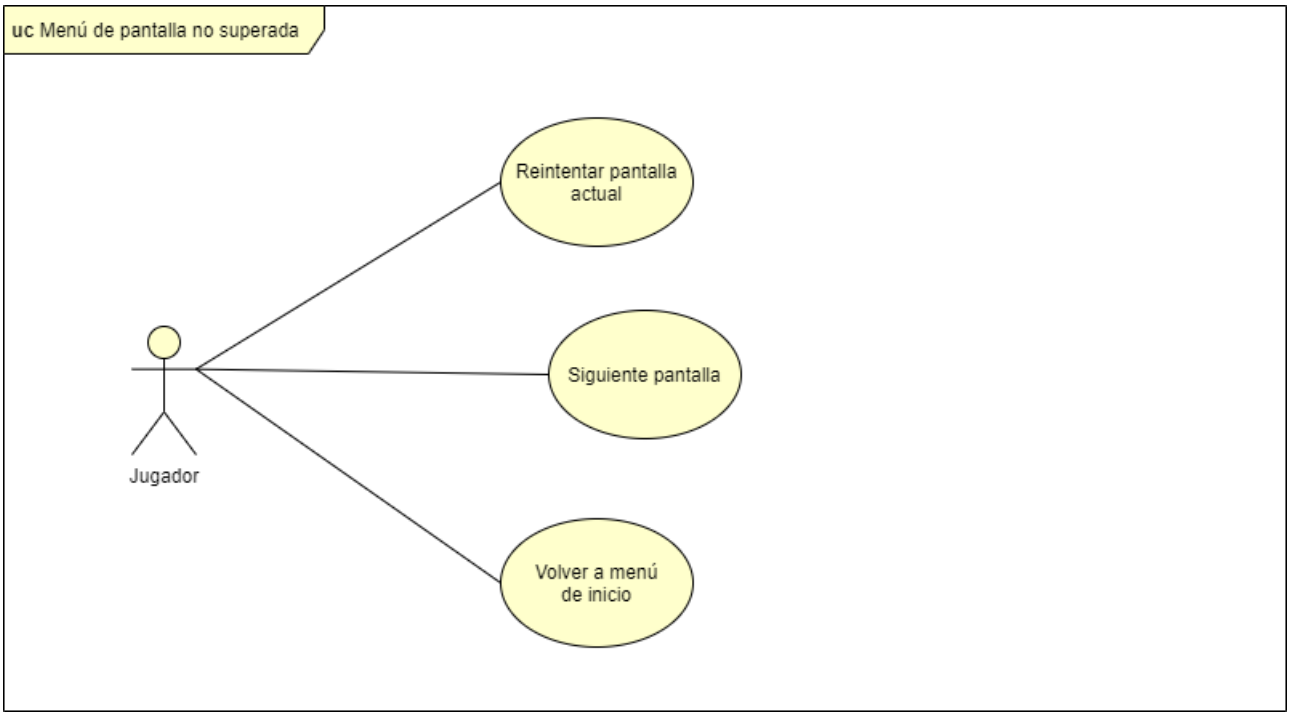


**Figura 11:** Diagrama de casos de uso: Menú de pantalla no superada.

## 5.2 DISEÑO E IMPLEMENTACIÓN

En esta sección analizaremos cómo se han diseñado y construido los videojuegos del proyecto utilizando las herramientas que disponemos dentro del motor de Unity.

Para una mejor claridad en la exposición, aislaremos los distintos sistemas que dotan de funcionamiento al juego, en cada una de las siguientes sub-secciones. También se detallarán los detalles de diseño no relacionados con la programación, como los modelos 3D, animaciones y efectos de sonido.

### 5.2.1 *Conceptos previos*

Antes de poder analizar la implementación de los videojuegos deberemos familiarizarnos con el siguiente vocabulario utilizado por la herramienta:

- *Asset* [18]: Un *Asset* es cualquier elemento que pueda ser usado dentro de un proyecto, desde una imagen, modelo 3D, fichero de audio o tipos exclusivos del editor de Unity.
- *GameObject* [19]: Un *GameObject* es un contenedor de Componentes de Unity. Todos los *GameObjects* contienen obligatoriamente un Componente *Transform*, que define la posición, rotación y dimensiones del *GameObject* dentro de la escena. Un *GameObject* también puede contener otros *GameObjects*. Es el elemento básico de Unity al que se le puede dotar de funcionalidad.
- Componente [20]: Son propiedades definidas que se pueden dar a los *GameObject*, como un cuerpo tridimensional (*mesh*), elementos de interfaz de usuario (botón, panel, etc.), propiedades físicas (es afectado por la gravedad), o un script que nosotros mismos redactemos.
- Escena [21]: Es un archivo que contiene *GameObjects*. Podemos entender la Escena como el espacio físico de nuestro juego. Un juego puede contener varias escenas, pero sólo podremos tener cargada una simultáneamente.
- *Prefab* [22]: Un *Prefab* es un *GameObject* reutilizable, que se guarda fuera de una escena. Esto nos sirve, entre otras cosas, para poder generar dinámicamente instancias del *GameObject*.
- *Canvas* [23]: Es el Componente Unity que sirve para definir un espacio para paneles de interfaz de usuario.

### 5.2.2 *MonoBehaviour*

*MonoBehaviour*[24] es una clase Unity que constituirá uno de los recursos principales a la hora de utilizar scripts dentro del proyecto. Cuando hacemos que una clase derive de *MonoBehaviour*, estamos conectando ésta con el funcionamiento interno del motor de Unity, y damos acceso a ésta a los eventos del ciclo de vida[25] de los scripts. Este ciclo, ilustrado en la figura 12, es un mecanismo que sirve al motor de Unity para indicar cuándo se debe ejecutar algo, mediante el uso de funciones. Cada función es llamada cuando su evento correspondiente del ciclo de vida se dispara, y estos eventos siguen siempre el mismo orden en cada iteración del ciclo.

A demás, la clase *MonoBehaviour* nos permite asignar un script como Componente a un *GameObject*, de tal forma que al comenzar la ejecución del juego, se genere una instancia única de la clase asociada al *GameObject*. Esto es especialmente útil para controlar dicho *GameObject* de forma aislada, pues podremos acceder a sus otros Componentes de forma directa, pero también nos servirá para poder hacer referencia a esta instancia concreta de la clase a través de su *GameObject*, desde otros *GameObjects* de la Escena.

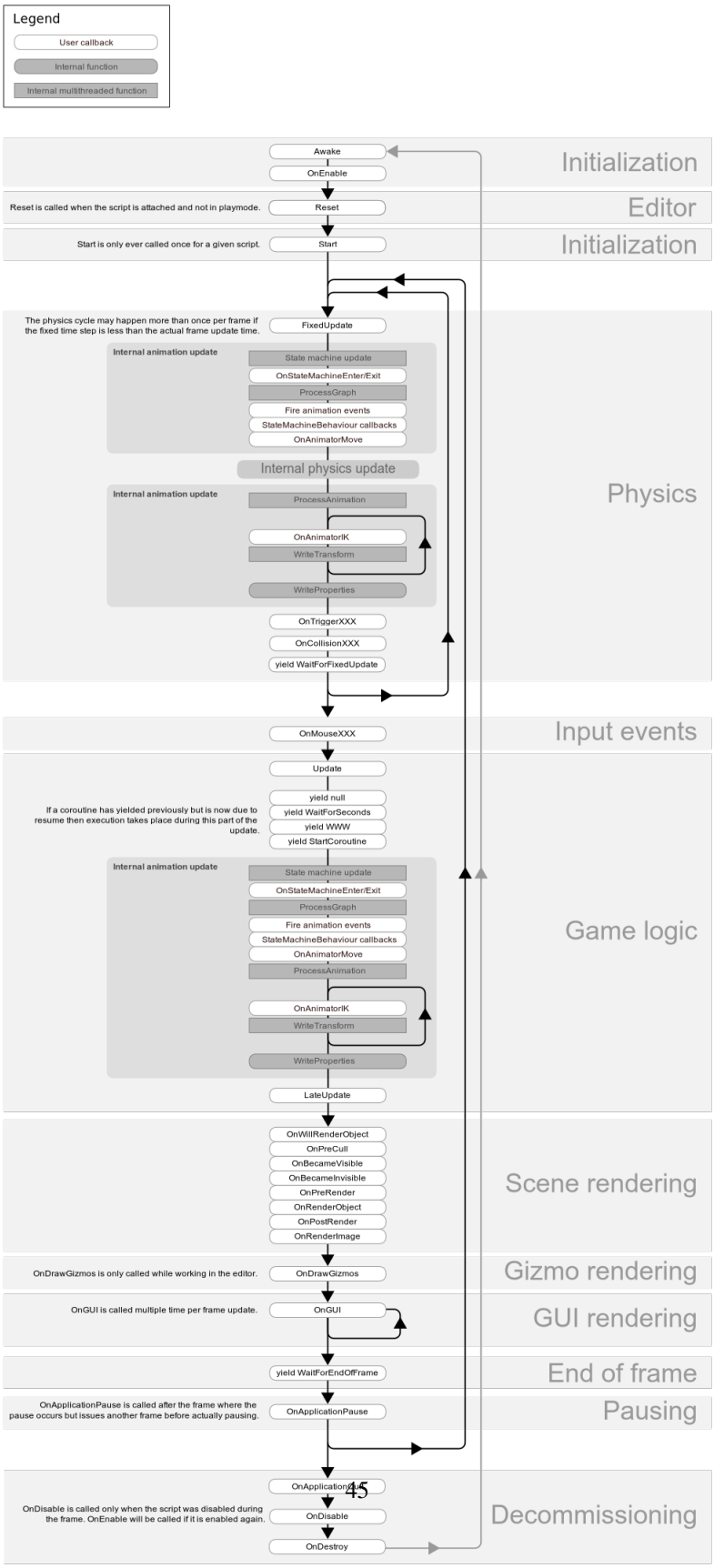




**Figura 12:** Ciclo de vida de script en Unity (MonoBehaviour).

### 5.2.3 *Eventos*

En Unity, cuando necesitemos que dos instancias de clases distintas, derivadas de *MonoBehaviour* y asignadas a diferentes *GameObjects*, interaccionen entre ellas, tendremos estas opciones para ello:

- Añadir una referencia en la clase al otro *GameObject* y establecer esta referencia mediante la interfaz del editor. Después, podemos obtener la referencia a la instancia a través del componente.
- Obtener una referencia al otro *GameObject* mediante uno de los métodos de Unity para buscar *GameObjects* dentro de la escena (*GameObject.Find/FindWithTag()*). Después, obtener la referencia a la instancia de igual forma.

Se trata de un mecanismo muy útil, pero que puede generar un "mar de dependencias" entre scripts debido a la necesidad de tener que comunicar desde una instancia de una clase a un conjunto de otras clases que "ha sucedido algo". Por ejemplo: Cuando el script del robot detecta que éste colisiona con una roca, el script que controla los elementos de la interfaz de juego deberá hacer que éstos se cierren, y el script que controla los menús de pantalla terminada, deberá hacer que aparezca en pantalla el menú de pantalla no superada.

Para evitar el problema indicado, dificultando la legibilidad del código, se ha implementado un sistema de eventos (figura 13), basado en una guía oficial de Unity [26].

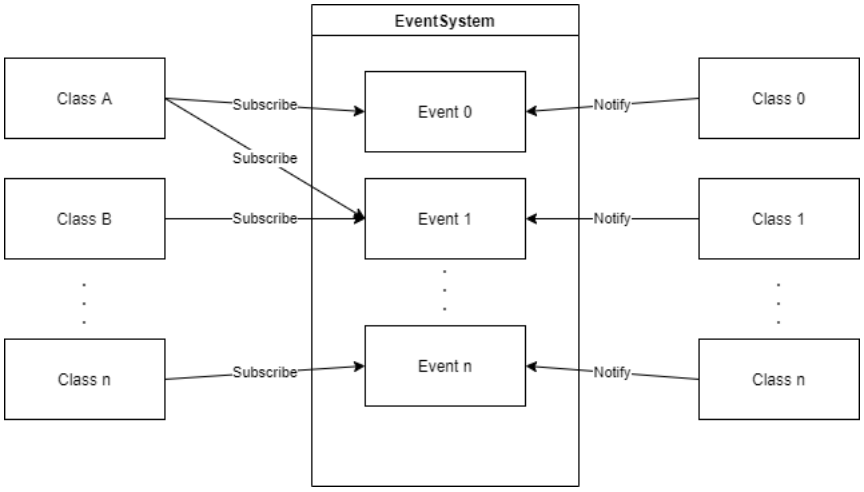


**Figura 13:** Sistema de Eventos.

Este sistema, consiste en una clase que llamaremos *EventSystem* (figura 13), que declara una lista de eventos:

```
public event Action onLevelLoad;
[...]
/// <summary>
/// (Event) A level is loaded and ready to be played
/// </summary>
public void LoadLevel() {
    if (onLevelLoad !=null) {
        onLevelLoad();
    }
}
```

El resto de clases, pueden suscribir métodos a uno de estos eventos:

```
GameEvents.current.onLevelLoad += ShowInGameUI;
```

O disparar el evento, haciendo que el resto de clases ejecuten los métodos suscritos:

```
GameEvents.current.LoadLevel();
```

Este sistema implementa los patrones de diseño *Singleton* y Observador.

### 5.2.4 *Generación y Carga de Niveles*

Cada Pantalla de juego de *Mars Miner* o *Venus Miner* está definido por 3 elementos:

- Configuración del tablero de juego: Los elementos de juego (rocas fijas, rocas movibles, minerales y el robot) y su disposición en el tablero al inicio de la pantalla.
- Código mostrado del robot: El texto del código Java mostrado en el panel de la derecha de la pantalla de juego, que representa las acciones que realizará el robot al iniciar su movimiento.
- Script de movimiento del robot: Es el código real, en lenguaje C#, que controla las acciones del robot al iniciar su movimiento.

En vez de definir una escena para cada pantalla de juego, en la cual deberíamos añadir "a mano" cada uno de los elementos del tablero de juego y asignar el código mostrado y el script de movimiento también de manera fija en cada escena, se ha diseñado un sistema para la carga dinámica de niveles. Este sistema consiste en utilizar 3 ficheros para cada uno de los 3 elementos anteriores, y un script que los interpreta y genera los elementos de juego necesarios en cada pantalla.

Para el tablero, se definirán archivos CSV (*comma separated values*). Esto es una tabla, con las mismas dimensiones que el tablero de juego, con la que indicamos qué elemento de juego debe haber en cada casilla. Un script leerá esta tabla y generará los *GameObjects* de cada elemento de juego a partir de *Prefabs*, y los colocará en la casilla correspondiente. Los posibles valores son los siguientes:

- 0: Casilla vacía.
- 1: Roca fija.
- 2: Roca movible (plateada).
- 3: mineral.
- d: robot orientado hacia abajo (coordenada X en la escena de Unity).
- u: robot orientado hacia arriba (coordenada -X en la escena de Unity).
- l: robot orientado hacia la izquierda (coordenada Z en la escena de Unity).
- r: robot orientado hacia la derecha (coordenada -Z en la escena de Unity).

Para el código mostrado en cada pantalla, se utilizará un fichero de texto. Para añadir coloreado, se podrán utilizar etiquetas XML de la forma $<color=\#FFFFFF>$, ya que este texto será interpretado por un Componente *TextMeshPro*[27], que soporta texto enriquecido[28].

Para el código de movimiento se han creado scripts en C#, con una clase que define la secuencia de movimientos del robot dentro de un método. Esta clase deberá derivar de una clase abstracta llamada *AbsLevel*, y definir en ella un método *Play()* que contiene la secuencia de acciones del robot. Las acciones posibles del robot se definen en una clase separada, *RobotActions*. Analizaremos en detalle este mecanismo más adelante.

Con la idea de que el cliente u otros futuros usuarios tengan la capacidad de crear niveles, el proyecto se entregará incluyendo una guía para ello, como ya se mencionó en la sección de entregables.

Este sistema de generación dinámica de niveles tiene varias ventajas:

- Rápida iteración en la creación de los niveles. Trabajar sobre estos archivos es más rápido que editar escenas en Unity a través del editor.
- Independencia del editor de Unity. Podemos crear niveles sin necesidad de instalar esta pesada herramienta.
- Precisión. En el editor de Unity, es fácil descolocar un objeto inintencionadamente dentro de la escena. Esto, para un juego de tablero, podría dar frecuentes problemas, al no coincidir la posición de los objetos con las casillas. Definiendo la posición de los objetos a través de código y utilizando una tabla, evitamos este problema.
- Expansibilidad. Con mínimos conocimientos sobre el funcionamiento interno del juego, podrán añadirse nuevos niveles en el futuro.
- Reutilización. El sistema ya ha servido para generar 2 variantes de juego, con tan solo modificaciones estéticas. En un futuro, podría reutilizarse, al menos parcialmente, para otros juegos de tablero.

En la figura 14 se muestra un diagrama de clases que refleja la implementación del sistema de lectura de los ficheros de configuración de pantalla. La clase *LevelLoader* es la encargada de tratar dichos ficheros. Esta clase interacciona con otras clases: *BoardManager*, *ScoreManager* y *InGameUICanvasActions*, que veremos más a delante en esta sección.

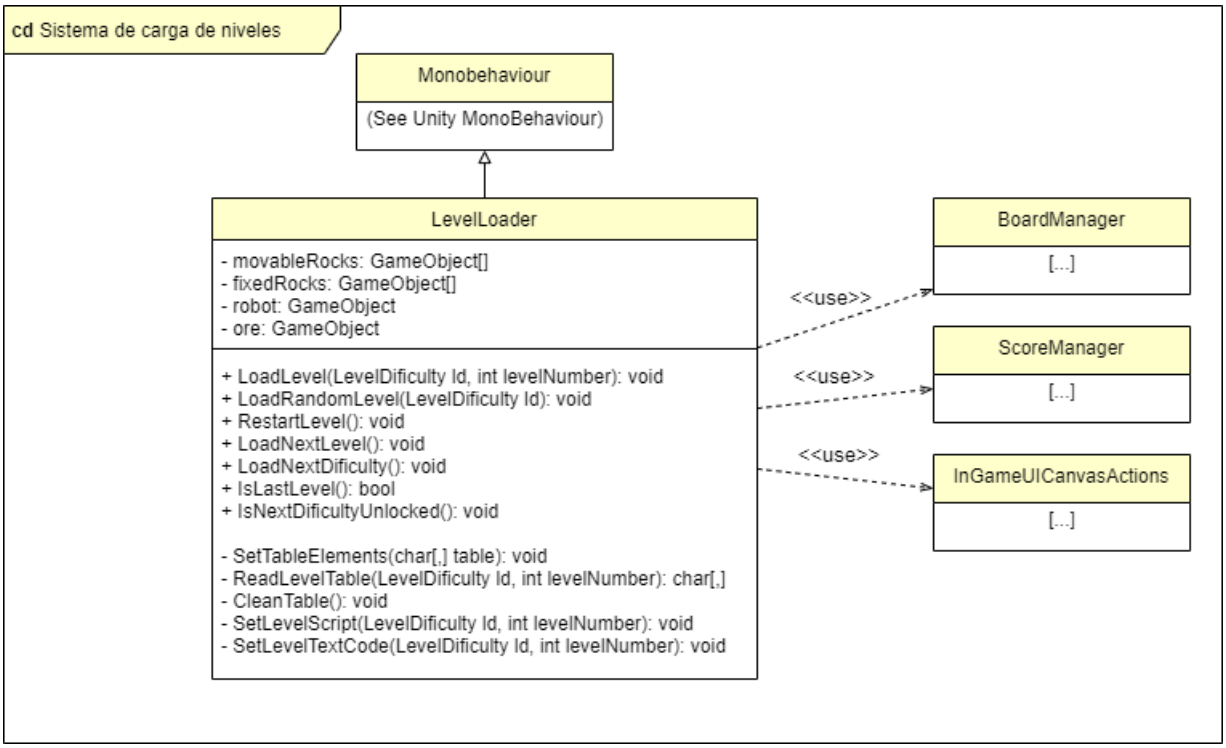


**Figura 14:** Diagrama de Clases: Carga de niveles.

### 5.2.5 *Gestión del tablero de juego*

Para controlar el desplazamiento de los elementos de juego dentro del tablero, se ha creado la clase *Tile* (figura 15). Esta clase sirve simplemente para representar una casilla del tablero, como unas coordenadas de 2 dimensiones. Aunque podría haberse utilizado una tupla, o la clase Vector2 de *UnityEngine*, de esta forma podemos diferenciar de forma más clara cuándo estamos tratando con una coordenada de nuestro tablero, y no un vector general.

La clase *BoardManager* (figura 15) sirve para definir propiedades del tablero, como sus dimensiones y el tamaño de cada casilla dentro del juego. También sirve para traducir las coordenadas de las casillas (*Tile*) a coordenadas reales de la escena de juego (Vector2, Vector3) cuando se desplaza el robot o una roca. Por último, utilizaremos también esta clase para asistir en el movimiento de rocas, indicando en la cuadrícula encima de qué casilla estamos situados y si se puede colocar la roca en la casilla o no.

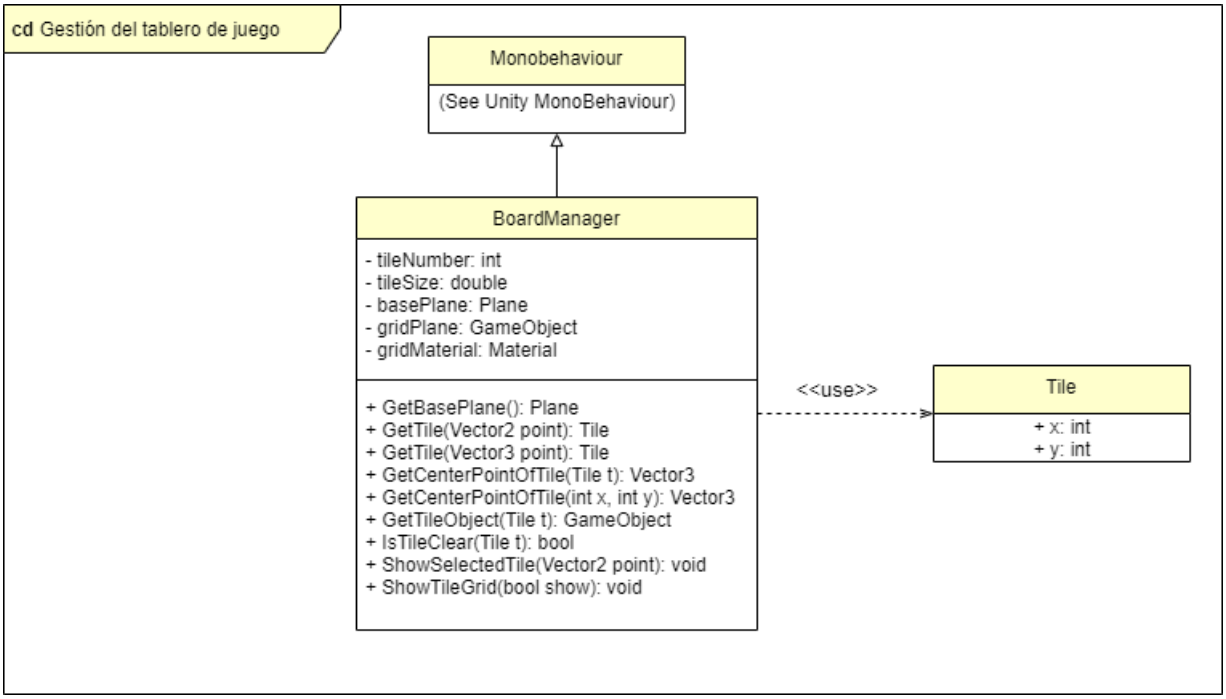


**Figura 15:** Diagrama de Clases: Tablero de juego.

### 5.2.6 *Niveles y acciones del robot. Corrutinas*

El sistema utilizado para la creación de niveles y las acciones del robot está representado en la figura 16, que contiene el diagrama de clases correspondiente.

Para programar las secuencias de acciones del robot, una clase definirá el conjunto de movimientos del robot, en forma de métodos. Después, para cada nivel, un script definirá la secuencia de movimientos concreta del nivel, utilizando dichos métodos y estructuras de código condicionales e iterativas. Esta secuencia estará encapsulada en la función *Play()*. Este script (clase *LevelN* en el diagrama de la figura 16) se asignará dinámicamente a cada instancia del *GameObject* del robot presente en la escena por la clase *LevelLoader* cuando carguemos una pantalla.

Para poder iniciar la secuencia de cualquier nivel, necesitaremos una interfaz que defina la función *Play()*. Esta es la interfaz *ILevel* que aparece en el diagrama de clases de la figura 16. La interfaz *ILevel* definirá también otras funciones requeridas para los scripts de nivel.

Dentro de los scripts de nivel necesitaremos definir también mecanismos para comprobar cuándo se ha completado la secuencia de movimiento del robot con éxito o cuándo el robot a colisionado y por tanto debe detenerse. Este código es común para todos los scripts de nivel, e irá definido en una clase Abstracta, *AbsLevel* (figura 16). Cada script de nivel deberá derivar de *AbsLevel*.

Las diferentes acciones del robot se definirán en el script *RobotActions* (figura 16) como funciones. Cada acción del robot deberá realizarse a tal velocidad que sea distinguible en la pantalla de juego. La clase *MonoBehaviour* no nos permite hacer esto a través de las funciones del ciclo de vida, por lo que deberemos utilizar otro mecanismo: corrutinas[29].

Las corrutinas nos permiten distribuir acciones en el tiempo, haciendo que su ejecución se pause en un punto, devolviendo el control a Unity, para luego retomar su ejecución en el punto donde se dejó.

Cada acción de movimiento o animación del robot se definirá como una función de tipo *IEnumerator*[30], una interfaz de C# que permite una iteración simple a través de una colección no genérica, y que devuelve *null* mientras no se halla completado:

```
public IEnumerator MoveFoward() {
    audioManager.Play("robot_motor2");
    Vector3 target = boardManager.GetCenterPointOfTile(GetFowardTile(1));
    float i = 0.0f;
    while (Vector3.Distance(transform.position, target) > 0.0f && canMove) {
        i += Time.deltaTime * moveSpeed;
        transform.position = Vector3.Lerp(transform.position, target, i);
        yield return null;
    }
    audioManager.Stop("robot_motor2");
}
```

El método *Play()* de cada script de nivel actuará como un contenedor de estas acciones y deberá ser declarado también de tipo *IEnumerator*. Para iniciar la secuencia de movimiento de un nivel, se hará a través de la función *StartCorrutine* de la siguiente manera:

```
GameObject[] characterCubes = GameObject.FindGameObjectsWithTag("CharacterCube");
foreach (GameObject characterCube in characterCubes) {
    StartCoroutine(characterCube.GetComponent<ILevel>()
    .Play(programInput.GetComponent<ProgramInputScript>().GetInputArrayStr()));
}
```

A continuación, en la figura 16 , se puede ver el diagrama de clase que muestra la relacióon entre las clases que implementan el sistema para la definicióon de los scripts de nivel del juego y la implementacióon de las acciones del robot, y sus dependencias con otras clases:

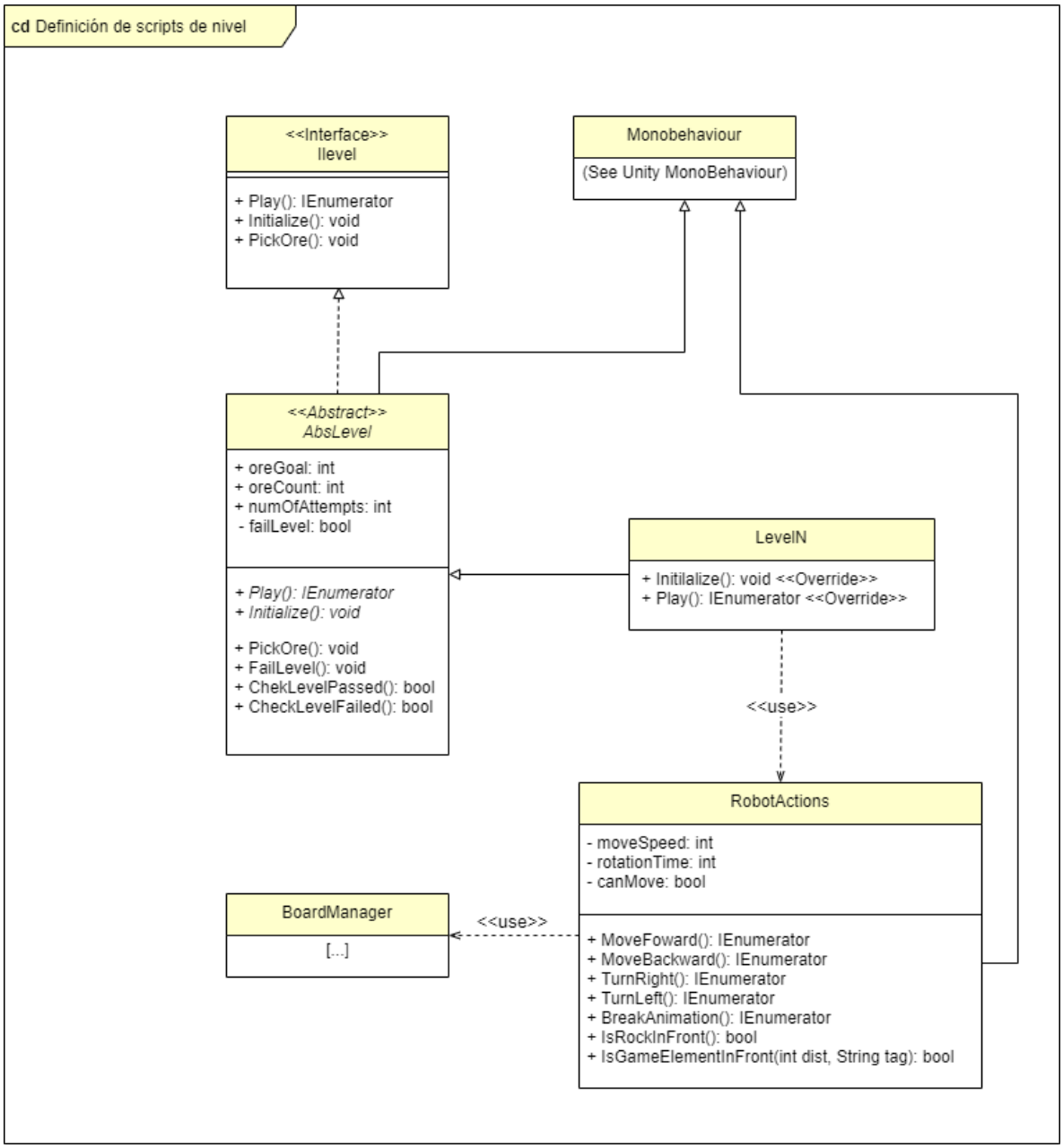


**Figura 16:** Diagrama de Clases: Scripts de nivel.

### 5.2.7 *Interfaz de usuario*

Para controlar la interfaz de usuario, la implementación del juego cuenta con un total de 12 scripts, que aparecen representados en el diagrama de clases de la figura 18. Es una implementación descentralizada, es decir, que no sigue una jerarquía en la que tengamos una clase desde la que se pueda manipular o acceder a todos los demás componentes o scripts que controlan la interfaz. Esta implementación descentralizada es resultado de la jerarquía de los *GameObjects* que componen la interfaz de usuario en la escena (figura 17).

Unity favorece una implementación de este tipo en cuanto a que, a más bajo el nivel en que se integren los scripts dentro de la jerarquía de *GameObjects*, más fácil es acceder a los Componentes de los mismos. Un Componente al mismo nivel que el Componente Script que lo controla, puede ser accedido directamente con *GetComponent<Tipo>()*, mientras que si debemos obtener una referencia a un Componente de distinto nivel, deberemos utilizar mecanismos menos eficientes para buscarlos, o indicar las referencias a través de la interfaz del editor.

A pesar de ello, se ha tratado de agrupar la funcionalidad de los distintos *Canvas* (figura 17) de tal forma que un script controle toda la funcionalidad del mismo, aunque con excepciones para funcionalidades aisladas. Esta es la jerarquía resultante de la implementación:

- Level Selection Canvas: Agrupa todos los elementos de la pantalla de inicio. Es controlado por el script *LevelSelector* (figura 18).
- Options Canvas: Agrupa todos los elementos del menú de opciones. Es controlado por el script *OptionsCanvasActions* (figura 18).
- InGame UI Canvas: Agrupa todos los elementos de interfaz de la pantalla de juego. Es controlado por el script *InGameUICanvasActions* (figura 18).
- Level Finished Canvas: Agrupa los elementos de los paneles mostrados al finalizar un nivel (nivel superado / nivel no superado). Es controlado por el script *LevelFinishedCanvasActions* (figura 18).

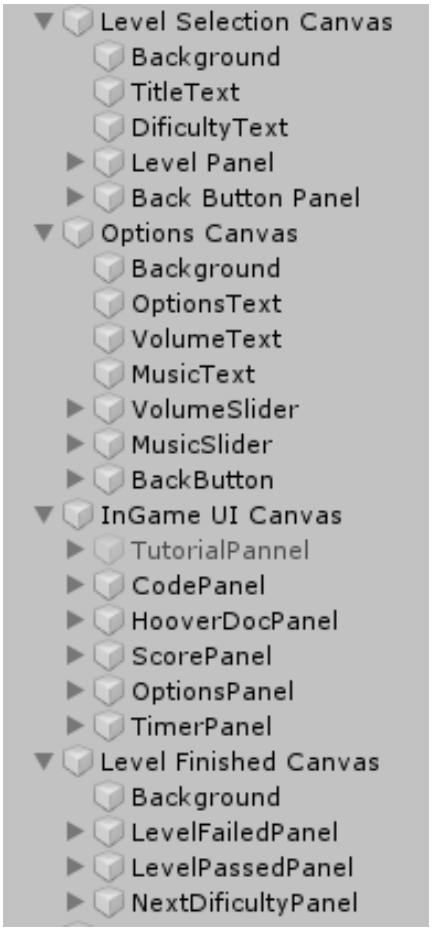


**Figura 17:** Jerarquía de *GameObjects* de la interfaz de usuario.

Para controlar cuándo se deberán mostrar u ocultar los distintos paneles de la interfaz de usuario, se utiliza el sistema de eventos previamente analizado. Cuando un elemento de interfaz requiera mostrar información manejada por otra clase, se utilizarán referencias a dichas clases.

La figura 18 contiene todas las clases empleadas para controlar la interfaz de usuario, y sus dependencias con las clases de los demás sistemas:

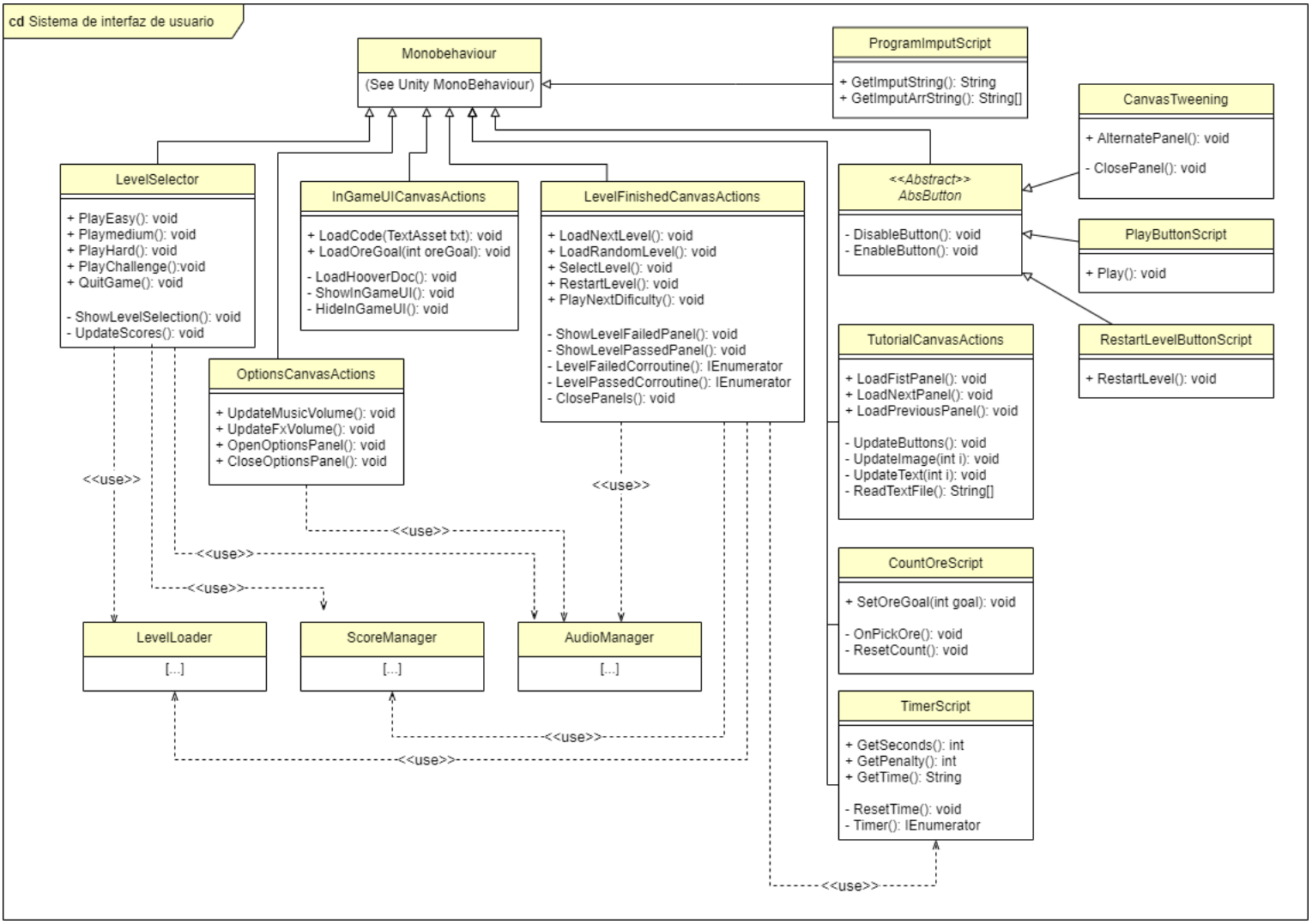


**Figura 18:** Diagrama de Clases: Interfaz de usuario.

### 5.2.8 *Guardado de puntuaciones e integración con la plataforma*

El sistema utilizado para el guardado de puntuaciones está representado en la figura 19, que contiene el diagrama de clases correspondiente.

Las puntuaciones del juego se manejarán a través de un objeto *Score* (figura 19). Esta clase simplemente guardará los valores de las puntuaciones de los cuatro niveles de dificultad por separado, y proporcionará unas funciones básicas para actualizar estos valores y calcular la puntuación total. Esta clase se declarará como *[System.Serializable]*, que permite su conversión a *json* para poder ser enviada y guardada en la plataforma.

La clase *ScoreManager* (figura 19) es la encargada de calcular la puntuación de cada pantalla, obtener las puntuaciones anteriormente obtenidas por el jugador desde la plataforma y enviar a la plataforma las puntuaciones actualizadas después de superar una pantalla.

El cálculo de la puntuación de cada pantalla se realiza a partir de su dificultad (campo *LevelDificulty*), el número de intentos seguidos, el tiempo que se ha tardado en completar la pantalla, y los diferentes parámetros de referencia, mostrados en el diagrama (figura 19) en mayúsculas (constantes). Este cálculo se realiza en la función *LevelPoints()* de *ScoreManager* (figura 19).

Para obtener la dificultad de la pantalla actual y el número de intentos, necesitaremos una referencia a la clase *LevelLoader* (figura 14), mientras que el tiempo se obtiene de la clase *TimerScript* (figura 19).

Para guardar las puntuaciones en la plataforma, debemos utilizar llamadas externas. Se trata de una solución que Unity ha declarado como obsoleta, aunque la mantiene, pero que se usa en la plataforma *Programa Jugando* porque no se encontró ninguna otra alternativa para comunicar aplicaciones independientes en tiempo de ejecución.

Para actualizar las puntuaciones del jugador en la plataforma usaremos esta llamada externa:

```
string jsonScore = JsonUtility.ToJson(finalScore);
#pragma warning disable CS0618
Application.ExternalCall("setCampoLibre", jsonScore);
#pragma warning restore CS0618
```

También tendremos otra llamada externa para indicar a la plataforma la puntuación total obtenida hasta el momento y si el jugador ha obtenido los suficientes puntos como para marcar el juego como superado en la plataforma:

```
int completed = 0;
if (finalScore.TotalScore() >= PASS_GAME_SCORE) completed = 1;
```

```
int [] data = {finalScore.TotalScore(), completed};
#pragma warning disable CS0618
Application.ExternalCall("guardar", data);
#pragma warning restore CS0618
```

Para solicitar el objeto *Score* guardado en la plataforma (en forma de *json*), deberemos realizar la siguiente llamada:

```
#pragma warning disable CS0618
Application.ExternalCall ("getCampoLibre");
#pragma warning restore CS0618
```

Y para poder recuperar el objeto, tendremos que incluir un *GameObject* en la escena con nombre "OpenField" que contenga un script *OpenField.cs* que defina un método *public void getCampoLibre(string campoLibre)*. Esta implementación debe ser así por requisito de la plataforma. Como la llamada a la función *getCampoLibre* por parte de la plataforma es asíncrona, utilizaremos el sistema de eventos para redirigir los datos de puntuación a *ScoreManager* (figura 19):

```
public class OpenField : MonoBehaviour {

    public void getCampoLibre(string campoLibre) {
        GameEvents.current.LoadGameData(campoLibre);
    }
}
```

En ScoreManager:

```
GameEvents.current.onLoadGameData += LoadScore;
[...]
private void LoadScore(string scoreStr) {
    try {
        if (!scoreStr.Equals("") && scoreStr != null) {
            finalScore = JsonUtility.FromJson<Score>(scoreStr);
            GameEvents.current.UpdateScores();
        }
    } catch (Exception e) {
        Debug.Log("Could not read GameData string: " + e.ToString());
    }
}
```

La figura 19 contiene el diagrama de las clases que forman el sistema de puntuaciones descrito y sus dependencias con las clases de los demás sistemas:

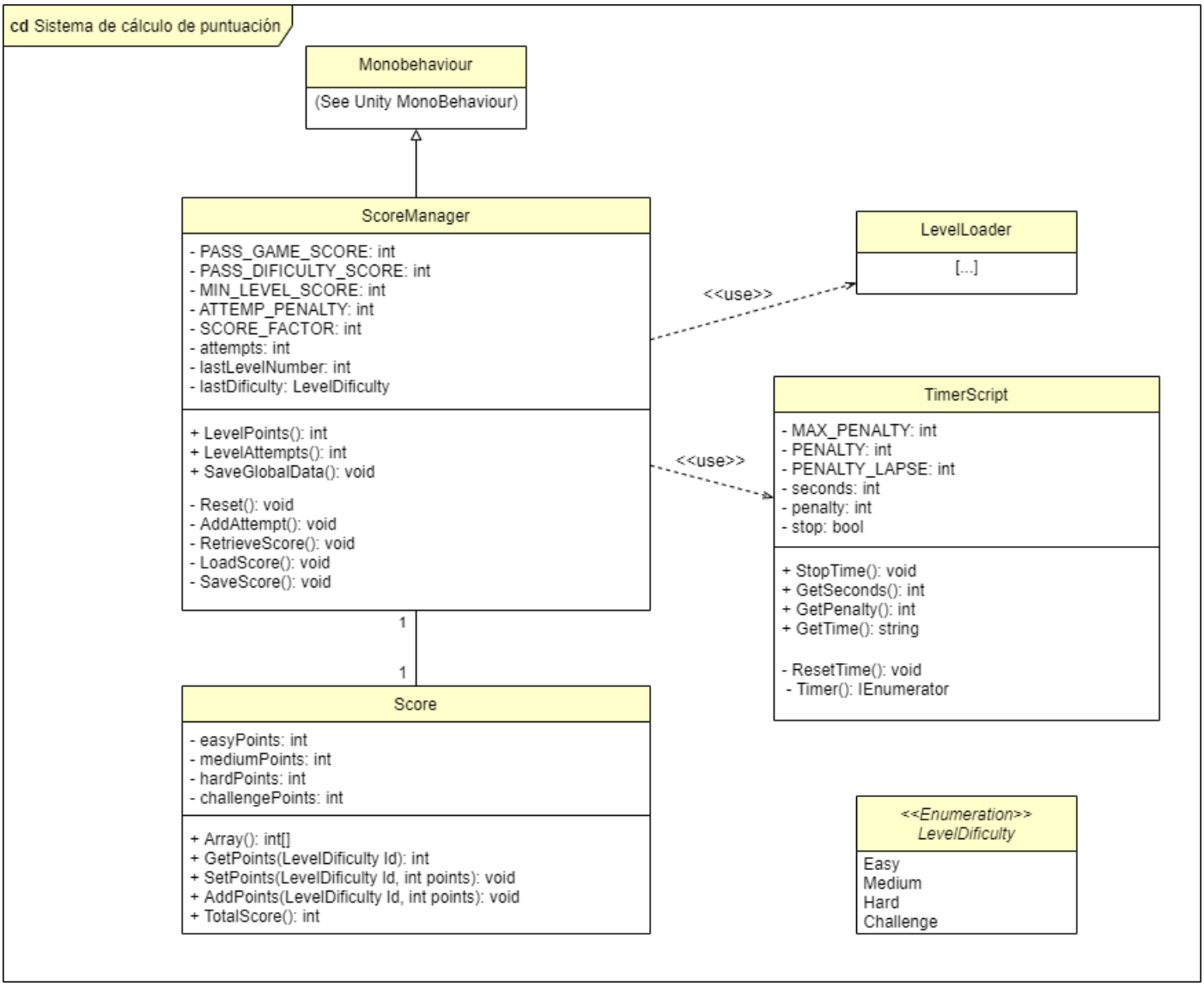


**Figura 19:** Diagrama de Clases: Puntuaciones.

### 5.2.9 *Sonido*

En Unity, para hacer que un *GameObject* emita un sonido podemos utilizar el Elemento *AudioSource*[31]. Este Elemento nos permite añadir un clip de sonido, modificar algunos parámetros del mismo y hacer que se reproduzca cuando queramos.

Para centralizar y simplificar el uso de los distintos sonidos del juego, se ha creado la clase *AudioManager* (figura 20). Esta clase se encarga de generar dinámicamente los *AudioSource* necesarios para los sonidos del juego, y proporciona métodos para ajustar y reproducir cada uno de los sonidos.

También, utilizaremos la clase *Sound* (figura 20) para agrupar en un sólo objeto el clip de sonido, su *AudioSource* y sus parámetros.

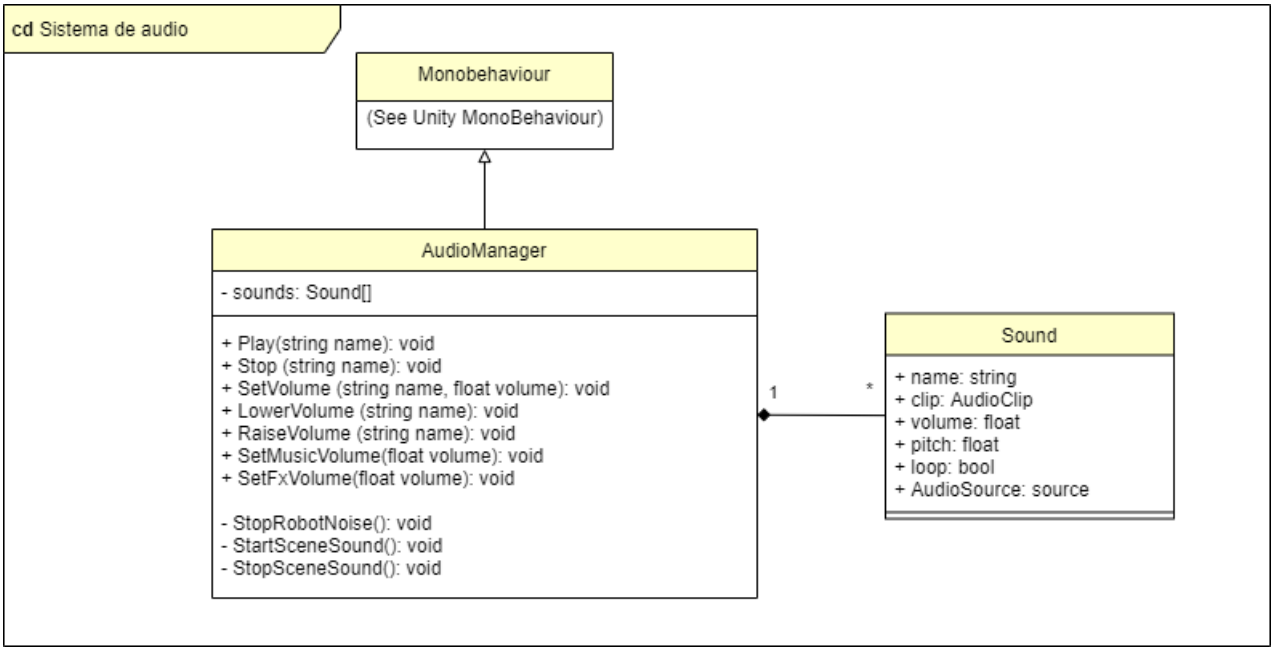


**Figura 20:** Diagrama de Clases: Sistema de audio.

## 5.3 PRUEBAS

Para verificar que la implementación de los diferentes aspectos del proyecto es correcta, se deberán realizar múltiples pruebas a lo largo del desarrollo del proyecto.

El proceso de prueba consiste en todas las actividades del ciclo de vida, estáticas o dinámicas, del proyecto, para determinar si se han satisfecho los requisitos establecidos, determinar que estos se ajustan al propósito del proyecto y para detectar fallos [32].

Para este proyecto definiremos los siguientes tipos de prueba que se deberán llevar al cabo:

- Pruebas unitarias: Prueban el funcionamiento correcto de un componente del proyecto de forma aislada. Serán realizadas por el desarrollador. Antes de poder fusionar el trabajo realizado de una rama de tarea a la rama de desarrollo, se deberán probar todos los componentes afectados por los cambios, mediante una serie de pruebas que irán asociadas a la tarea.
- Pruebas *Alpha*: Son las pruebas realizadas sobre una versión completa del proyecto aún en desarrollo. Esta prueba se realiza por miembros internos del equipo del proyecto, normalmente por un equipo de probadores ajeno al desarrollo. Para el proyecto, adaptaremos esta definición y estas pruebas se realizarán entre el desarrollador y el cliente de forma conjunta.
- Pruebas *Beta*: Son realizadas por una selección de usuarios finales del proyecto, los alumnos en este caso, sobre una versión completa del proyecto cercana a la versión final. A partir de esta etapa de prueba, se obtiene retroalimentación del jugador sobre aspectos mejorables del juego, o errores que no se han localizado anteriormente, para su ajuste y corrección en la versión final.

En las siguientes sub-secciones se listan las pruebas realizadas.

### 5.3.1 *Pruebas Unitarias*

Utilizaremos la siguiente plantilla de la figura 33 para describir cada una de las pruebas unitarias realizadas, , que se muestran en las tablas 34 a 69.

| | |
|---|---|
| Identificador | Identificador de la prueba, en la forma PU-00. |
| Elemento | Parte de la implementación proyecto sobre la que se realiza la prueba . |
| Descripción | Descripción de la prueba, detallando el comportamiento esperado. |
| Resultado | Salida de la prueba, puede ser ‘pasada’ o ‘fallida’. |

**Tabla 33:** Plantilla de prueba unitaria.

| Identificador | PU-01 |
|---|---|
| Elemento | Menú de inicio - General. |
| Descripción | El menú de inicio se muestra correctamente al cargar el juego en la plataforma. |
| Resultado | Pasada. |

**Tabla 34:** Prueba Unitaria PU-01

| Identificador | PU-02 |
|---|---|
| Elemento | Menú de inicio - Selección de nivel. |
| Descripción | En el menú de inicio se muestran 4 botones, uno por nivel dificultad. Al pulsar un botón de nivel, se carga la pantalla de juego, con una pantalla seleccionada aleatoriamente entre las pantallas disponibles para la dificultad. |
| Resultado | Pasada. |

**Tabla 35:** Prueba Unitaria PU-02

| Identificador | PU-03 |
|---|---|
| Elemento | Menú de inicio - Desbloqueo de niveles. |
| Descripción | Al iniciar por primera vez el juego, aparecen bloqueados todos los botones de selección de nivel de dificultad a excepción del primero. Al lograr una puntuación determinada en un nivel, se desbloquea el botón de la siguiente dificultad. |
| Resultado | Pasada. |

**Tabla 36:** Prueba Unitaria PU-03

| Identificador | PU-04 |
|---|---|
| Elemento | Menú de inicio - Mostrar puntuaciones. |
| Descripción | Debajo de cada botón de selección de dificultad que esté desbloqueado, se muestra la puntuación obtenida hasta el momento por el jugador en esa dificultad. |
| Resultado | Pasada. |

**Tabla 37:** Prueba Unitaria PU-04

| | |
|---|---|
| Identificador | PU-05 |
| Elemento | Menú de inicio - Ajustes. |
| Descripción | En el menú de inicio aparece un botón para acceder al menú de ajustes. Este menú permite ajustar el volumen de la música y los efectos de sonido independientemente, mediante 2 sliders. |
| Resultado | Pasada. |

**Tabla 38:** Prueba Unitaria PU-05

| | |
|---|---|
| Identificador | PU-06 |
| Elemento | Menú de inicio - Salir. |
| Descripción | En el menú de inicio aparece un botón para cerrar el juego. En la plataforma, al pulsar el botón, el juego se cierra y se vuelve a la pantalla de selección de juego. |
| Resultado | Pasada. |

**Tabla 39:** Prueba Unitaria PU-06

| | |
|---|---|
| Identificador | PU-07 |
| Elemento | Pantalla de juego - carga del tutorial. |
| Descripción | Al cargar por primera vez una pantalla de juego, aparece el primer panel del tutorial superpuesto a la escena de juego, y se bloquea la interacción con el resto de elementos de la pantalla. |
| Resultado | Pasada. |

**Tabla 40:** Prueba Unitaria PU-07

| | |
|---|---|
| Identificador | PU-08 |
| Elemento | Pantalla de juego - cerrar tutorial. |
| Descripción | Al cerrar el último panel del tutorial, los elementos de la pantalla de juego bloqueados previamente se desbloquean. |
| Resultado | Pasada. |

**Tabla 41:** Prueba Unitaria PU-08

| Identificador | PU-09 |
|---|---|
| Elemento | Pantalla de juego - Desplazar roca. |
| Descripción | En la escena de juego, se podrá hacer click sobre las rocas plateadas, iniciando la animación de 'levitación'. Al arrastrar el ratón se desplaza la roca. Al soltar la roca, ésta se coloca en la casilla sobre la que levitaba en el último instante. Si se suelta la roca sobre una casilla ya ocupada, esta vuelve a su casilla original. |
| Resultado | Pasada. |

**Tabla 42:** Prueba Unitaria PU-09

| Identificador | PU-10 |
|---|---|
| Elemento | Pantalla de juego - Mostrar cuadrícula. |
| Descripción | Mientras se arrastra el ratón para desplazar una roca, se muestra la cuadrícula del tablero, con la casilla sobre la que la roca levita coloreada. Si la casilla sobre la que levita está libre, la cuadrícula se muestra de color verde, y roja en caso contrario. |
| Resultado | Pasada. |

**Tabla 43:** Prueba Unitaria PU-10

| Identificador | PU-11 |
|---|---|
| Elemento | Pantalla de juego - HooverDoc. |
| Descripción | Al pulsar el botón 'HooverDoc', aparece en pantalla el panel con la documentación del código del robot. Al volver a pulsarlo, éste se cierra. |
| Resultado | Pasada. |

**Tabla 44:** Prueba Unitaria PU-11

| Identificador | PU-12 |
|---|---|
| Elemento | Pantalla de juego - Cronómetro. |
| Descripción | Al iniciar una pantalla, el cronómetro empieza contar. Cada 30 segudos, el contador de penalización incrementa en 50 puntos, hasta un máximo de 200. |
| Resultado | Pasada. |

**Tabla 45:** Prueba Unitaria PU-12

| | |
|---|---|
| Identificador | PU-13 |
| Elemento | Pantalla de juego - Marcador de minerales. |
| Descripción | Al iniciar una pantalla, el marcador de minerales muestra el número de minerales requerido para superar la pantalla. Al recoger un mineral, el número de minerales recogidos se actualiza. |
| Resultado | Pasada. |

**Tabla 46:** Prueba Unitaria PU-13

| | |
|---|---|
| Identificador | PU-14 |
| Elemento | Pantalla de juego - Botón start. |
| Descripción | Al pulsar el botón start, éste se bloquea y comienza la ejecución de la secuencia de acciones del robot. |
| Resultado | Pasada. |

**Tabla 47:** Prueba Unitaria PU-14

| | |
|---|---|
| Identificador | PU-15 |
| Elemento | Pantalla de juego - Panel de opciones. |
| Descripción | Al pulsar el botón de la esquina superior izquierda de la pantalla de juego, aparece un panel con los botones 'Reiniciar nivel', 'Opciones' y 'Volver a inicio'. Volviendo a pulsar el botón, el panel se cierra. |
| Resultado | Pasada. |

**Tabla 48:** Prueba Unitaria PU-15

| | |
|---|---|
| Identificador | PU-16 |
| Elemento | Pantalla de juego - Reiniciar nivel. |
| Descripción | Al pulsar el botón de reiniciar nivel, la escena de juego vuelve a su estado inicial, y el botón 'start' se desbloquea si estaba bloqueado. |
| Resultado | Pasada. |

**Tabla 49:** Prueba Unitaria PU-16

| | |
|---|---|
| Identificador | PU-17 |
| Elemento | Panel de pantalla superada - General. |
| Descripción | Al completarse la secuencia de acciones del robot y haber recogido todos los minerales necesarios para pasar la pantalla, aparece el panel de pantalla superada y los demás elementos de interfaz de usuario de la pantalla de juego desaparecen. Se muestra en esta pantalla los intentos, tiempo y puntos de la pantalla correctamente. |
| Resultado | Pasada. |

**Tabla 50:** Prueba Unitaria PU-17

| | |
|---|---|
| Identificador | PU-18 |
| Elemento | Panel de pantalla superada - Continuar. |
| Descripción | Al pulsar el botón 'continuar', se carga otra pantalla de juego aleatoria de la misma dificultad. |
| Resultado | Pasada. |

**Tabla 51:** Prueba Unitaria PU-18

| | |
|---|---|
| Identificador | PU-19 |
| Elemento | Panel de pantalla superada - Seleccionar otra dificultad. |
| Descripción | Al pulsar el botón 'seleccionar otra dificultad' se vuelve a la pantalla de inicio. |
| Resultado | Pasada. |

**Tabla 52:** Prueba Unitaria PU-19

| | |
|---|---|
| Identificador | PU-20 |
| Elemento | Panel de pantalla superada - Siguiente dificultad. |
| Descripción | Cuando se supera la pantalla obteniendo suficientes puntos para desbloquear la siguiente dificultad, aparece el botón 'siguiente dificultad'. Al pulsarlo, se carga una pantalla aleatoria de la siguiente dificultad. No ocurre si ya se está jugando en la última dificultad (reto). |
| Resultado | Pasada. |

**Tabla 53:** Prueba Unitaria PU-20

| | |
|---|---|
| Identificador | PU-21 |
| Elemento | Panel de pantalla no superada - General. |
| Descripción | Al colisionar el robot contra una roca o contra los límites del tablero, aparece el panel de pantalla no superada y los demás elementos de interfaz de usuario de la pantalla de juego desaparecen. |
| Resultado | Pasada. |

**Tabla 54:** Prueba Unitaria PU-21

| | |
|---|---|
| Identificador | PU-22 |
| Elemento | Panel de pantalla no superada - Reintentar pantalla . |
| Descripción | Al pulsar el botón 'reintentar pantalla, se vuelva a cargar la última pantalla de juego. |
| Resultado | Pasada. |

**Tabla 55:** Prueba Unitaria PU-22

| | |
|---|---|
| Identificador | PU-23 |
| Elemento | Panel de pantalla no superada - Continuar. |
| Descripción | Al pulsar el botón 'continuar', se carga otra pantalla de juego aleatoria de la misma dificultad. |
| Resultado | Pasada. |

**Tabla 56:** Prueba Unitaria PU-23

| | |
|---|---|
| Identificador | PU-24 |
| Elemento | Panel de pantalla no superada - Seleccionar otra dificultad. |
| Descripción | Al pulsar el botón 'seleccionar otra dificultad' se vuelve a la pantalla de inicio. |
| Resultado | Pasada. |

**Tabla 57:** Prueba Unitaria PU-24

| | |
|---|---|
| Identificador | PU-25 |
| Elemento | Guardado y carga de puntuaciones. |
| Descripción | En la plataforma, al salir del juego tras conseguir puntos y volver a entrar, se muestra la puntuación actualizada correctamente. |
| Resultado | Pasada. |

**Tabla 58:** Prueba Unitaria PU-25

| | |
|---|---|
| Identificador | PU-26 |
| Elemento | Efectos de sonido - Música. |
| Descripción | Al iniciar el juego, comienza a reproducirse el clip de sonido de la música del juego. |
| Resultado | Pasada. |

**Tabla 59:** Prueba Unitaria PU-26

| | |
|---|---|
| Identificador | PU-27 |
| Elemento | Efectos de sonido - Viento. |
| Descripción | Al seleccionar una pantalla, comienza a reproducirse el clip de sonido del efecto de viento. |
| Resultado | Pasada. |

**Tabla 60:** Prueba Unitaria PU-27

| | |
|---|---|
| Identificador | PU-28 |
| Elemento | Efectos de sonido - Ruidos del robot. |
| Descripción | Al seleccionar una pantalla, comienza a reproducirse el clip de sonido del efecto ruido del motor del robot. Cuando este se mueve, el tono del efecto de sonido cambia. |
| Resultado | Pasada. |

**Tabla 61:** Prueba Unitaria PU-28

| | |
|---|---|
| Identificador | PU-29 |
| Elemento | Efectos de sonido - Roca levitando. |
| Descripción | Mientras se arrastra una roca, se reproduce el clip de sonido del efecto de ondas magnéticas. |
| Resultado | Pasada. |

**Tabla 62:** Prueba Unitaria PU-29

| | |
|---|---|
| Identificador | PU-30 |
| Elemento | Efectos de sonido - Soltar roca. |
| Descripción | Al soltar una roca que se estaba desplazando, se reproduce el clip de sonido del efecto de golpe. |
| Resultado | Pasada. |

**Tabla 63:** Prueba Unitaria PU-30

| | |
|---|---|
| Identificador | PU-31 |
| Elemento | Efectos de sonido - Colisión del robot. |
| Descripción | Al colisionar el robot, se reproduce el clip de sonido del efecto de golpe. |
| Resultado | Pasada. |

**Tabla 64:** Prueba Unitaria PU-31

| | |
|---|---|
| Identificador | PU-32 |
| Elemento | Efectos de sonido - Recoger mineral. |
| Descripción | Al recoger un mineral, se reproduce el clip de sonido del mieral recogido. |
| Resultado | Pasada. |

**Tabla 65:** Prueba Unitaria PU-32

| | |
|---|---|
| Identificador | PU-33 |
| Elemento | Efectos de sonido - Pulsar botón. |
| Descripción | Al pulsar cualquier botón, se reproduce el sonido de botón pulsado. |
| Resultado | Pasada. |

**Tabla 66:** Prueba Unitaria PU-33

| | |
|---|---|
| Identificador | PU-34 |
| Elemento | Animaciones - Colisión del robot. |
| Descripción | Al colisionar el robot, se reproduce la animación de colisión del robot. |
| Resultado | Pasada. |

**Tabla 67:** Prueba Unitaria PU-34

| | |
|---|---|
| Identificador | PU-35 |
| Elemento | Animaciones - Tormenta de arena. |
| Descripción | Al iniciar una pantalla de juego, comienza la animación de efecto tormenta de arena. |
| Resultado | Pasada. |

**Tabla 68:** Prueba Unitaria PU-35

| | |
|---|---|
| Identificador | PU-36 |
| Elemento | Animaciones - Ondas magnéticas. |
| Descripción | Mientras se desplaza una roca, se reproduce la animación de ondas magnéticas. |
| Resultado | Pasada. |

**Tabla 69:** Prueba Unitaria PU-36

### 5.3.2 *Pruebas Alpha*

Las pruebas *Alpha* de los juegos consistirán en la demostración de su funcionamiento frente al cliente. En estas pruebas, realizadas durante las reuniones de seguimiento una vez que se haya completado el desarrollo suficiente, el cliente propone escenarios de interacción con el juego, poniéndose en la piel del jugador. Se simulan dichos escenarios, y se anota cualquier fallo observado o comentarios del cliente a cerca de la mejora o modificación de la interacción con el juego.

Estas pruebas se han realizado de manera informal y cuando se han considerado oportunas por el desarrollador o el cliente. A continuación, se listan los cambios realizados en los juegos a raíz de estas reuniones:

- Rediseño de la pantalla de inicio: Inicialmente, en la pantalla de inicio, aparecía un menú en el que el jugador seleccionaba qué pantalla quería jugar individualmente. Tras una prueba con el cliente, se decidió cambiar este diseño, para que el jugador sólo tuviera que seleccionar un nivel de dificultad.
- Selección de pantalla aleatoria: Después del cambio anterior, cuando el jugador seleccionaba una dificultad, la primera pantalla de juego que se cargaba para cada dificultad era siempre la misma. Se decidió cambiar este diseño, de tal forma que la pantalla cargada es siempre aleatoria, sin poder repetirse. La única excepción es la pantalla cargada para el tutorial, que siempre será la primera pantalla cargada cuando sea la primera vez que se juega.
- Opacidad de los paneles de interfaz de usuario. Tras otra prueba con el cliente, se aumentó la opacidad del fondo de varios de los paneles de la interfaz de los juegos, para mejorar la lectura del texto.

### 5.3.3 *Pruebas Beta*

Debido a la fecha de entrega del proyecto, no ha podido ser probado aún por su usuario final, los alumnos. Por este motivo, se delegará la responsabilidad de la realización de las pruebas beta en el cliente.

La entrega del código fuente junto a una documentación para el ajuste de los parámetros del cálculo de puntuaciones y la creación de niveles, como se indicó en la sección de entregables, permitirán al cliente ajustar estos elementos tras obtener retroalimentación de alumnos al probar los juegos.

Se estima que dichos elementos serán los que más probablemente necesiten modificarse, pues se deberán ajustar en base a las capacidades observadas en los alumnos para superar con éxito cada uno de los niveles de dificultad.

# 6

# CONCLUSIÓN

Al finalizar el proyecto, se han cumplido los objetivos propuestos inicialmente. Esto es, la creación de dos videojuegos para la gamificación de la asignatura de Fundamentos de Programación, su integración con la plataforma Programa Jugando, y una documentación para el ajuste de dichos juegos.

Con estos juegos, se pretende ayudar a los alumnos a asimilar con más facilidad cómo funcionan las estructuras condicionales, iterativas, funciones y métodos en Java, y los conceptos de paso por valor y paso por referencia.

Enseñándoles a analizar y descifrar código a través de juegos, tratamos de convertir una tarea de aprendizaje normalmente pesada y que puede resultar frustrante para muchos, en un pasatiempo divertido.

La gamificación también nos puede aportar el valor añadido de motivar al alumnado y hacer que su interés por la materia aumente, pudiendo repercutir en una mejora de sus resultados académicos.

A demás, con la implementación realizada de los juegos, no sólo hemos cubierto las necesidades del cliente, sino que a demás hemos creado una serie de sistemas para poder ampliar el contendido de dichos juegos en el futuro, o crear otros diferentes a partir de ellos.

## 6.1 LÍNEAS DE TRABAJO FUTURAS

En el plano de los juegos *Mars Miner* y *Venus Miner*, podríamos ampliar estos dos juegos de diversas formas, con el objetivo de aumentar el tiempo de juego de cada uno de ellos, y añadir variedad a su jugabilidad. A continuación enumeramos algunas de estas posibilidades:

- Añadir más pantallas de juego, usando el mecanismo de generación implementado.
- Añadir otros lenguajes de programación, en otras nuevas versiones del juego.
- Añadir modos de juego, para añadir diversidad a la forma de jugar las mismas pantallas. Por ejemplo, podríamos añadir un modo de juego en el cuál se establezca un temporizador de X minutos, y el jugador deba resolver el mayor número posible de pantallas en dicho tiempo, aumentando progresivamente la dificultad.
- Añadir más elementos del tablero de juego, como otros tipos de obstáculo, u obstáculos con movimiento. Añadiría un plus de complejidad y podría propiciar más creatividad en el diseño de las secuencias de movimiento.

Por otro lado, alguno de los sistemas implementados en este juego, como la generación dinámica de pantallas de juego o el sistema de control de tablero, podrían aprovecharse para crear otros juegos con mecánicas de juego completamente distintas.

Una proposición de segundo juego que se hizo durante el desarrollo del proyecto, en lugar de *Venus Miner*, fue la de aprovechar el sistema de tablero para crear un juego en el que, el jugador, controlaría directamente el movimiento del robot mediante teclas. La mecánica de juego consistiría en recoger materiales que representarían tipos de datos, con el objetivo de aprender las diferentes estructuras de almacenamiento de datos disponibles en Java y qué tipo de datos pueden albergar cada una. Finalmente, este concepto de juego fue descartado en favor de la implementación de *Venus Miner*, debido a las limitaciones de tiempo del proyecto.

# BIBLIOGRAFÍA


[1] Alma María Pisabarro Marrón, Carlos Enrique Vivaracho Pascual, "Plataforma programa jugando." https://greidi.infor.uva.es/programajugando/, consultado en: 17-06-2021.

[2] Alma María Pisabarro Marrón, Carlos Enrique Vivaracho Pascual, "Grupo de estudio en innovación docente en informática." https://greidi.infor.uva.es/, consultado en: 17-06-2021.

[3] Wikipedia, "Unity (motor de videojuego)." https://es.wikipedia.org/wiki/Unity_(motor_de_videojuego), consultado en: 15-06-2021.

[4] "Git." https://git-scm.com/, consultado en: 15-06-2021.

[5] "Github." https://github.com/, consultado en: 15-06-2021.

[6] "Unity gitignore." https://github.com/github/gitignore/blob/master/Unity.gitignore, consultado en: 19-06-2021.

[7] Blender Foundation, "Blender." https://www.blender.org/, consultado en: 15-06-2021.

[8] Willy-Peter Schaub, "How does kanban relate to devops?." https://web.archive.org/web/20200401130304/https://opensource.com/article/20/4/kanban-devops, consultado en: 15-06-2021.

[9] "Aplicación trello." https://trello.com/, consultado en: 17-06-2021.

[10] Jobted, "Sueldo del programador de videojuegos en españa." https://www.jobted.es/salario/programador-videojuegos, consultado en: 17-06-2021.

[11] Agencia Tributaria, "Tabla de coeficientes de amortización lineal." https://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml, consultado en: 17-06-2021.

[12] Tracy Fullerton, "Game design workshop, chapter 14: The design document," consultado en: 20-06-2021.

[13] Unity Technologies, "Particle system." https://docs.unity3d.com/Manual/class-ParticleSystem.html, consultado en: 19-06-2021.

[14] Freesound, "freesound." https://freesound.org/, consultado en: 12-06-2021.

[15] Daniel Robnik, "Low poly styled rocks." https://assetstore.unity.com/packages/3d/props/exterior/low-poly-styled-rocks-43486, consultado en: 11-05-2021.

[16] Unity Technologies, "Standard assets (for unity 2018.4)." https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-for-unity-2018-4-32351, consultado en: 02-06-2021.

[17] Revision3, "Simple grid shader." https://assetstore.unity.com/packages/tools/simple-grid-shader-119988, consultado en: 03-11-2019.

[18] Unity Technologies, "Asset workflow." https://docs.unity3d.com/Manual/AssetWorkflow.html, consultado en: 19-06-2021.

[19] Unity Technologies, "Gameobjects." https://docs.unity3d.com/Manual/GameObjects.html, consultado en: 19-06-2021.

[20] Unity Technologies, "Using components." https://docs.unity3d.com/Manual/UsingComponents.html, consultado en: 19-06-2021.

[21] Unity Technologies, "scenes." https://docs.unity3d.com/Manual/CreatingScenes.html, consultado en: 19-06-2021.

[22] Unity Technologies, "Prefabs." https://docs.unity3d.com/Manual/Prefabs.html, consultado en: 19-06-2021.

[23] Unity Technologies, "Canvas." https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html, consultado en: 19-06-2021.

[24] Unity Technologies, "Important classes - monobehaviour." https://docs.unity3d.com/Manual/class-MonoBehaviour.html, consultado en: 19-06-2021.

[25] U. Technologies. https://docs.unity3d.com/Manual/ExecutionOrder.html.

[26] Unity Technologies, "Eventos." https://learn.unity.com/tutorial/eventos-w#5e419557edbc2a0a62170fe6, consultado en: 14-02-2021.

[27] Unity Technologies, "Textmeshpro." https://docs.unity3d.com/Manual/com.unity.textmeshpro.html, consultado en: 19-06-2021.

[28] Digital Native Studios, "Rich text." http://digitalnativestudios.com/textmeshpro/docs/rich-text/, consultado en: 19-06-2021.

[29] Unity Technologies, "Coroutines." https://docs.unity3d.com/Manual/Coroutines.html, consultado en: 19-06-2021.

[30] Microsoft, "Ienumerator interfaz." https://docs.microsoft.com/es-es/dotnet/api/system.collections.ienumerator?view=net-5.0, consultado en: 20-06-2021.

[31] Unity Technologies, "Audiosource." https://docs.unity3d.com/Manual/class-AudioSource.html, consultado en: 19-06-2021.

[32] ISTQB Foundation, "Testing." https://glossary.istqb.org/app/en/search/testing, consultado en: 19-06-2021.