Universidad Zaragoza
1542

Trabajo Fin de Grado

Desarrollo de un videojuego en red con control compartido en tiempo real

Autor/es

Sergio Larrodera Arcega

Director/es

Francisco José Serón Arbeloa
Eduardo Mena Nieto

Escuela de Ingeniería y Arquitectura
2015

Desarrollo de un videojuego en red con control compartido en tiempo real

RESUMEN

Actualmente, los videojuegos están experimentando un gran auge tanto a nivel industrial como de volumen de jugadores. Uno de los formatos más populares es, sin duda, el de los videojuegos multijugador online. En este tipo de juegos, el modelo de interacción entre jugadores consiste en que cada jugador controla una entidad dentro del juego, normalmente independiente del resto de jugadores, por ejemplo, un personaje, un vehículo o un ejército.

Este trabajo explora la idea de que varios jugadores controlen de forma simultánea a la misma entidad mediante un sistema que ejecute en tiempo real la voluntad de la mayoría. Con este sistema de "control compartido", se persigue que los jugadores tengan que adaptarse al curso de acción decidido por la mayoría en cada momento y tengan que reevaluar constantemente su estrategia en función de lo que está sucediendo y no de los planes que tuvieran previamente.

Para lograr esto, se ha desarrollado un sencillo videojuego de plataformas en dos dimensiones, en el cual los jugadores controlan a un robot que recolecta mineral en un planeta desconocido. El mundo por el que se mueve el robot se genera de forma distinta en cada partida, para evitar que los jugadores puedan memorizar los caminos, de forma que no se pierda el componente de incertidumbre, que es esencial en este modelo de interacción.

Una vez se terminaron el videojuego y el sistema de control compartido, se realizaron pruebas con usuarios para comprobar si el sistema funciona de forma adecuada y si este tipo de interacción es atractiva para los jugadores.

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

D./Dª. Sergio Larrodera Arcega,

con nº de DNI 73007219K en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Desarrollo de un videojuego en red con control compartido en tiempo real,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 23 de junio de 2015

Fdo: Sergio Larrodera Arcega

ÍNDICE

1. INTRODUCCIÓN

La industria del videojuego ha crecido a gran ritmo en los últimos años. Uno de los formatos de videojuego más populares es el de los juegos multijugador online. Como ejemplo, *League of Legends*, el videojuego para PC más popular actualmente, alcanzaba a principios de 2014 los 27 millones de jugadores diarios[1].

Típicamente, en un juego multijugador la interacción entre jugadores consiste en que cada jugador controla una entidad diferente en el juego, ya sea un personaje, una nave espacial o una civilización, y se enfrenta o coopera con el resto de jugadores. También existen algunos ejemplos de juegos en los que varios jugadores controlan distintas partes de una misma entidad (por ejemplo, un jugador controla el movimiento de un helicóptero mientras otro dispara la ametralladora que lleva incorporada).

En este trabajo se plantea la idea de un nuevo tipo de interacción entre jugadores, en la que varios jugadores cooperan para controlar a un solo personaje en tiempo real mediante un sistema de gobierno de la mayoría, llamado en este documento "control compartido". El sistema recibe continuamente las acciones que desea realizar cada jugador y, cada pocos milisegundos, elige la acción o acciones mayoritarias y las aplica. Con este tipo de interacción, se busca que los jugadores intenten coordinarse para lograr un objetivo común, teniendo que reaccionar y adaptarse rápidamente a las decisiones que toma la mayoría.

1 Dato ofrecido por la empresa desarrolladora del videojuego, disponible en: http://www.riotgames.com/articles/20140711/1322/league-players-reach-new-heights-2014

2. OBJETIVOS

Para explorar la idea expuesta anteriormente, se ha propuesto desarrollar un sencillo videojuego de plataformas en dos dimensiones, con dos condiciones importantes. Por un lado, el jugador debe poder controlar al personaje usando un conjunto pequeño de acciones, de forma que la coordinación entre jugadores sea lo más sencilla posible. Por otro lado, cada partida debe ser diferente para que los jugadores no puedan memorizar una secuencia de acciones favorables y deban, por el contrario, tomar decisiones rápidas si quieren obtener un buen resultado.

El trabajo se ha organizado en dos partes. La primera parte consiste en desarrollar el videojuego para un solo jugador, con todas las características que debe tener un videojuego de este tipo: personaje, mundo que le rodea, elementos con los que interactúa, efectos de sonido, música, interfaz de usuario, etc. La segunda parte consiste en extender el videojuego, añadiendo un modo cooperativo online que permita a varios jugadores controlar al personaje simultáneamente, y realizar pruebas con usuarios para averiguar si el sistema de control compartido funciona, si es divertido y cómo se podría mejorar.

3. ESPECIFICACIÓN

En este apartado se describe el videojuego desde un punto de vista de diseño, explicando en detalle sus características y cómo se han desarrollado.

3.1. Historia y ambientación

El juego está ambientado en un planeta desierto con clima adverso, pero que contiene bajo su superficie un mineral desconocido con propiedades valiosas (representado en forma de cristales morados). El planeta estuvo habitado hace siglos por una civilización que se considera extinta. Se cree que dicha civilización se aprovechaba de las cualidades de este mineral para sobrevivir en las profundidades del planeta.

El jugador controla a un robot de exploración planetaria enviado por la humanidad para recoger la mayor cantidad posible de mineral. La orografía del planeta es de naturaleza montañosa y cavernosa, por lo que el robot necesita gastar una considerable cantidad de energía para sortear los desniveles del terreno y al mismo tiempo ser lo suficientemente pequeño para poder introducirse por cavernas y túneles en busca de mineral. Por ello, su batería es de duración muy limitada, por lo que el robot necesita recoger otros minerales que le sirvan de combustible (representados en forma de cristales azules) para continuar su viaje. Una vez el robot se queda sin energía, utiliza sus baterías de emergencia para salir del planeta y llevar a los humanos los valiosos minerales obtenidos.

La figura 1 muestra una imagen del robot en las profundidades del planeta. Debajo del robot se puede ver una de las muchas runas mágicas fabricadas por los antiguos habitantes del planeta, las cuales tienen diversos efectos sobre el robot.

Figura 1 – Protagonista del juego explorando una caverna, rodeado por runas y cristales

3.2. Jugabilidad

El objetivo del juego es conseguir el mayor número posible de cristales morados antes de quedarse sin energía. El mundo se genera proceduralmente (es decir, durante la ejecución del programa) a partir de un número entero aleatorio (llamado semilla), para que el jugador no pueda memorizar los caminos o la ubicación de los cristales. El mundo es infinito (hacia abajo y a los lados) y continuo (no dividido en niveles), por lo que se genera conforme el personaje avanza por el mismo.

La parte del planeta que interactúa con el robot está formada por bloques cuadrados de tamaño algo más grande que el robot. Algunos de estos bloques contienen runas que se activan al entrar en contacto con el robot, provocándole diversos efectos. Existen cuatro tipos de runas:

– Runa azul: al activarse, otorga al robot una mayor velocidad de movimiento durante unos segundos.

– Runa verde: mientras está activa, el robot obtiene una mayor potencia de salto.

– Runa amarilla: cuando se activa, el robot se queda pegado a la runa. Le permite apoyarse en las paredes (para saltar desde ellas) o caminar por el techo (en este último caso, al saltar se despega de la runa).

– Runa roja: al activarse, el robot pierde una considerable cantidad de energía.

Cerca de la superficie del planeta abundan los cristales azules que recuperan energía, pero no hay cristales morados. En las zonas profundas se encuentran los preciados cristales morados, pero también los cristales azules son más escasos y es más peligroso avanzar. De esta forma, el jugador deberá buscar cuevas profundas para encontrar más cristales morados y, de vez en cuando, regresar cerca de la superficie para reponer energía.

Sin embargo, los cristales azules no devuelven siempre la misma cantidad de energía. La cantidad devuelta es un valor aleatorio cuya media disminuye lentamente con el número de cristales azules obtenidos, limitando así la duración de la partida. Para que el jugador reciba un *feedback* de la cantidad de energía restaurada, al recoger un cristal azul se escucha una nota musical cuya mayor agudeza significa una mayor cantidad de energía.

Algunos aspectos relacionados con la energía, como la cantidad que restauran los cristales azules, lo rápido que disminuye la energía con el tiempo o la energía que se pierde al tocar una runa roja, están ajustados por un parámetro de dificultad. Este parámetro se utiliza para compensar la dificultad más elevada inherente a las partidas cooperativas.

A pesar de que la idea inicial fue que el robot dispusiera únicamente de un salto, de forma que debiera tocar el suelo antes de volver a saltar de nuevo, finalmente se decidió dotar al robot de un doble salto, pudiendo así saltar una vez más en mitad del aire. De esta forma, mejoró mucho la movilidad del robot, permitiéndole alcanzar los cristales con más precisión, evitar más fácilmente las runas rojas y escalar mayores pendientes.

3.3. Controles e interfaz

Los controles son sólo tres: izquierda, derecha y saltar, correspondientes a las teclas A, D y W en la

versión para PC y a distintas zonas de la pantalla táctil en la versión para Android. Además, existen algunas teclas de configuración: F10 para activar o desactivar el modo de depuración de errores, F11 para activar o desactivar el modo de pantalla completa y Escape (manteniéndolo pulsado 1 segundo) para cerrar la aplicación.

La interfaz es de estilo sencillo, pero cumple con su función. Al iniciar el juego aparece un menú donde se puede elegir entre jugar una partida para un jugador, crear una partida cooperativa, unirse a una partida cooperativa o modificar las opciones.

Durante la partida, se muestra un marcador en la parte superior izquierda de la pantalla en el que aparece la puntuación y la energía restante del robot. Al recoger cristales azules, cristales morados o al tocar una runa roja, se muestra sobre el robot la cantidad de energía restaurada, los puntos ganados o la energía perdida, respectivamente. Además, las luces del robot se iluminan según las teclas que está pulsando el jugador. Esto evita que el jugador sienta que sus acciones no tienen ningún efecto en la partida si no se corresponden con las ejecutadas por el sistema de control compartido.

En la figura 2 se puede ver una captura del juego con algunos de los elementos mencionados: el marcador, el indicador mostrado al perder energía y la luz verde del robot iluminada para indicar que el jugador está pulsando la tecla derecha.

Figura 2 – Muestra de algunos elementos de la interfaz

3.4. Gráficos

El juego utiliza proyección ortográfica, con tres planos superpuestos: el plano frontal, el plano intermedio y el fondo, como se puede ver en la figura 3. Estos planos representan un corte lateral del planeta, con su superficie y sus cuevas. El plano frontal es donde están el robot y los bloques que interactúan con éste: las paredes de las cuevas que se encuentran en el mismo plano que el robot. Se mueve con la cámara y colisiona con el robot. El plano intermedio representa el fondo

cercano al plano frontal, es decir, el fondo de las cuevas que recorre el robot. Se mueve con la cámara, pero no colisiona con el robot. Su función es decorar y dar sensación de profundidad. El fondo representa el fondo lejano al robot: las montañas en la superficie del planeta y el cielo. Sólo se mueve levemente con la cámara al subir o bajar, pero no lateralmente, y no colisiona con el robot.

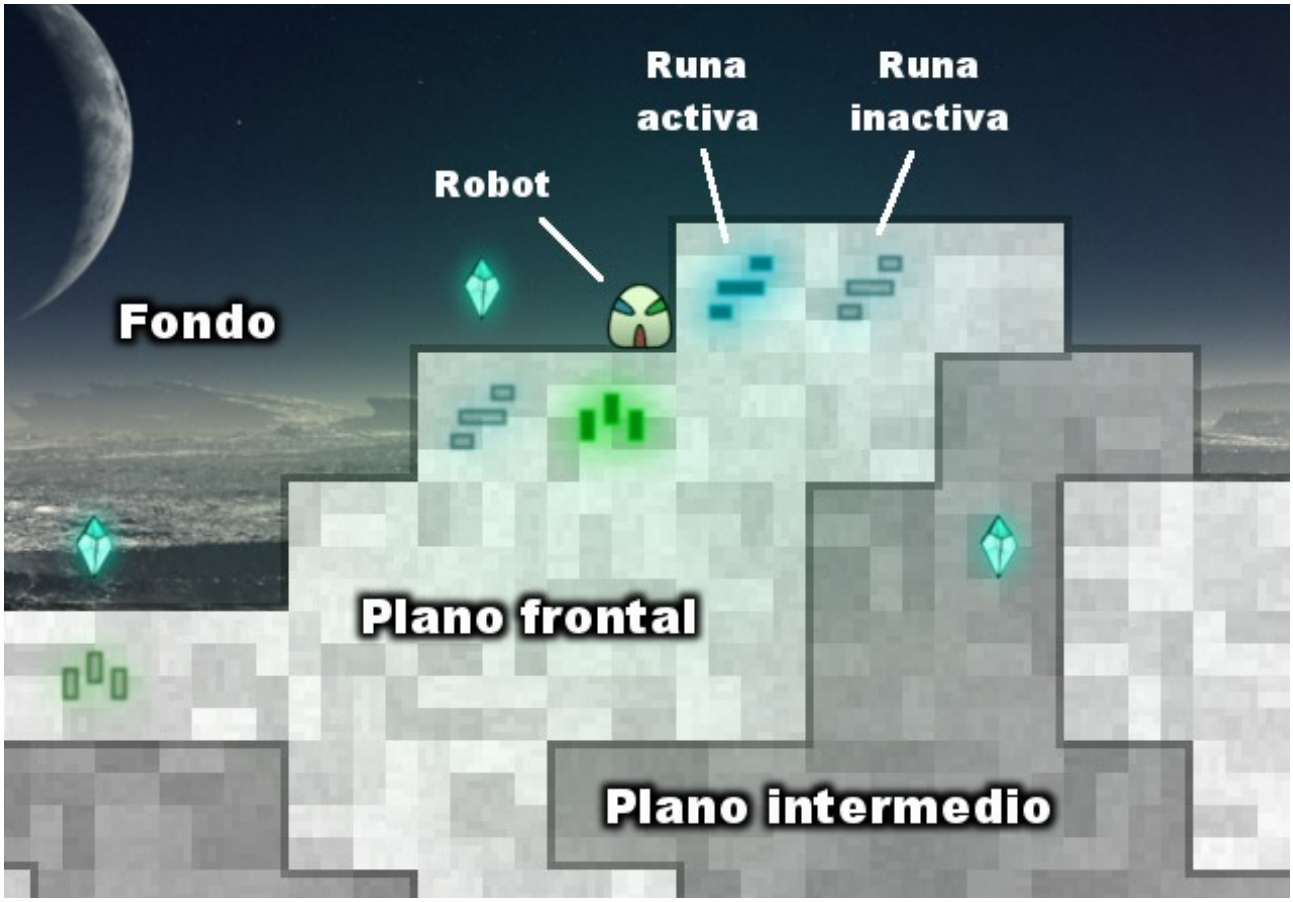

Figura 3 – Muestra de algunos elementos gráficos

El mundo representa un planeta árido y misterioso. Los bloques tienen una textura que se genera proceduralmente a partir de un conjunto de patrones (ver apartado 4.5), obteniendo un aspecto rocoso. Los bloques del plano frontal tienen un color blanco en la superficie, que se oscurece a medida que se va descendiendo. Los bloques del plano intermedio son de un color algo más oscuro que los frontales para distinguir entre las paredes y el fondo de la cueva.

Las runas grabadas en algunos bloques tienen un color característico según el tipo de runa. En el estado normal, el color es apagado, pero al activar estos bloques las runas brillan con un color más intenso durante el tiempo en el que están activas. La forma de cada runa ha sido diseñada para que sea fácil de diferenciar y a la vez insinúe sus propiedades. Se han utilizado formas rectangulares alargadas para que se asemejen a la textura diseñada para la roca.

El robot es de color blanco y textura pulida. Tiene forma de triángulo equilátero redondeado y en cada uno de sus lados tiene un *led*. Las luces de los *leds* se iluminan según la tecla pulsada y son de color azul, verde y rojo. La anchura y altura del robot son ligeramente superiores a medio bloque.

Para una mejor visualización de lo expuesto en este apartado, se pueden consultar más imágenes del videojuego en el apartado 5.

3.5. Música y sonido

La música se ha utilizado para crear un ambiente de misión espacial, reflejando que la acción transcurre en un planeta remoto y desierto. Se ha elegido un solo tema de extensa duración, considerando que era suficiente para el ámbito de este trabajo. Tanto esta música como los efectos de sonido usados en el juego se han obtenido del banco de sonidos Freesound[2] (algunos de ellos se han editado posteriormente).

El jugador recibe una respuesta sonora cuando el robot interactúa con algunos objetos del escenario. Si recoge un cristal azul, se escucha una nota de xilófono cuya agudeza depende de la energía restaurada. Para los cristales morados se utiliza una pequeña melodía de estilo robótico. Al tocar una runa roja, se reproduce un sonido corto de alerta, para avisar al jugador de que ha perdido energía. Por último, se añadió un sencillo sonido de cuenta atrás al iniciar la partida, que termina justo al aterrizar el robot en la superficie del planeta.

2 Disponible en: http://www.freesound.org

4. DESARROLLO

A continuación, se presenta la descripción del videojuego desde el punto de vista de la ingeniería del software: requisitos definidos, casos de uso de la aplicación, arquitectura del sistema y detalle de los principales algoritmos.

4.1. Requisitos

Los requisitos que se plantearon al comienzo del trabajo fueron los siguientes. En las tablas 1 y 2 se detalla la prioridad que se dio inicialmente a cada requisito y si se ha cumplido al final del trabajo.

Código	Requisito funcional	Prioridad	Cumplido
RF1	La aplicación permitirá comenzar una partida para un jugador	Alta	Sí
RF2	La aplicación permitirá crear una partida cooperativa personalizada	Alta	Parcial
RF3	La aplicación permitirá unirse a una partida cooperativa creada por otra persona	Alta	Parcial
RF4	La aplicación permitirá unirse a una cola de emparejamientos automáticos (*matchmaking*) para partidas cooperativas	Alta	Parcial
RF5	Al finalizar una partida, la aplicación ofrecerá al jugador o jugadores la posibilidad de guardar una imagen del mundo generado en esa partida	Baja	No
RF6	La aplicación permitirá modificar las opciones del juego, entre ellas el volumen de la música y los efectos de sonido y el idioma	Media	Sí
RF7	La aplicación permitirá al usuario ver las estadísticas de sus partidas	Baja	No

Tabla 1 – Requisitos funcionales

Código	Requisito no funcional	Prioridad	Cumplido
RNF1	La aplicación podrá ejecutarse en entornos Windows	Alta	Sí
RNF2	La aplicación podrá ejecutarse en entornos Android	Baja	Sí
RNF3	La aplicación estará disponible en inglés	Alta	Sí
RNF4	La aplicación estará disponible en español	Media	No

Tabla 2 – Requisitos no funcionales

Algunos de los requisitos con menos prioridad no han podido abordarse debido al tiempo del que se disponía. Caso especial son los requisitos RF2, RF3 y RF4. Debido a las limitaciones de tiempo y a las herramientas con las que se contaba, se ha optado por implementar una opción intermedia a estos requisitos:

– La aplicación permite al usuario organizar una partida cooperativa y añadirla a la lista de partidas

activas. La aplicación ejerce como servidor (no jugador) para esa partida.

– La aplicación permite al usuario unirse a una partida cooperativa. Se le asigna una partida automáticamente entre las que hay disponibles en la lista.

Este sistema no funciona ni como una partida personalizada tradicional ni como una cola de emparejamientos automáticos (ya que las partidas se crean de forma manual), pero se consideró que era la forma más rápida de poner en funcionamiento un sistema multijugador usando las herramientas proporcionadas por Unity, el motor gráfico usado para crear el juego.

4.2. Casos de uso

La figura 4 muestra los casos de uso de la aplicación. Posteriormente, se detalla cada caso de uso por separado (tablas 3 a 6).

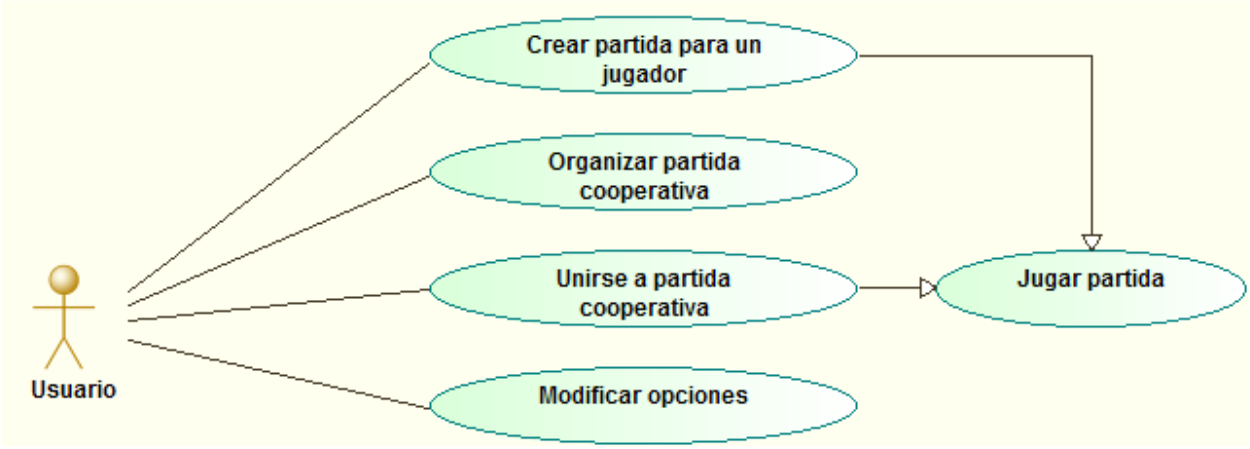

Figura 4 – Casos de uso

Caso de uso	Crear partida para un jugador
Descripción	Este caso de uso le permite al usuario crear una partida para un jugador
Precondiciones	El usuario se encuentra en el menú principal
Flujo de eventos	1. El usuario pulsa el botón “PLAY SINGLE PLAYER” 2. El sistema inicia la partida

Tabla 3 – Caso de uso “crear partida para un jugador”

Caso de uso	Organizar partida cooperativa
Descripción	Este caso de uso le permite al usuario organizar una partida cooperativa, ejerciendo como servidor de la partida
Precondiciones	El usuario se encuentra en el menú principal
Flujo de eventos	1. El usuario pulsa el botón “HOST MULTIPLAYER” 2. El sistema muestra una pantalla con las opciones para la partida: número de jugadores y puerto que se utilizará 3. El usuario selecciona las opciones deseadas, pulsa el botón “OK” y espera a que el resto de jugadores se una a la partida

	4. El sistema muestra una pantalla de espera con información de los jugadores que se han unido a la partida hasta el momento 5. Cuando todos los jugadores necesarios se han unido, el sistema inicia la partida

Tabla 4 – Caso de uso "organizar partida cooperativa"

Caso de uso	Unirse a partida cooperativa
Descripción	Este caso de uso le permite al usuario unirse a una partida cooperativa entre las disponibles
Precondiciones	El usuario se encuentra en el menú principal
Flujo de eventos	1. El usuario pulsa el botón "PLAY MULTIPLAYER" 2. El sistema muestra una pantalla de espera 3. Cuando todos los jugadores necesarios se han unido, el sistema inicia la partida

Tabla 5 – Caso de uso "unirse a partida cooperativa"

Caso de uso	Modificar opciones
Descripción	Este caso de uso le permite al usuario modificar las opciones del juego
Precondiciones	El usuario se encuentra en el menú principal
Flujo de eventos	1. El usuario pulsa el botón "OPTIONS" 2. El sistema muestra una pantalla con las opciones del juego: volumen de música y efectos, pantalla completa y modo de depuración de errores 3. El usuario modifica las opciones deseadas y pulsa el botón "OK" 4. El sistema se actualiza según las opciones modificadas y vuelve al menú principal

Tabla 6 – Caso de uso "modificar opciones"

4.3. Arquitectura del sistema

En este apartado se detalla la estructura de la aplicación, incluyendo las partes en las que está dividido el código y los componentes que se ejecutan en un momento dado.

4.3.1. Vista de módulos

El siguiente diagrama de clases (figura 5) presenta la organización interna del código y las relaciones más importantes entre las distintas partes. Posteriormente, se explica brevemente la función de cada clase del diagrama.

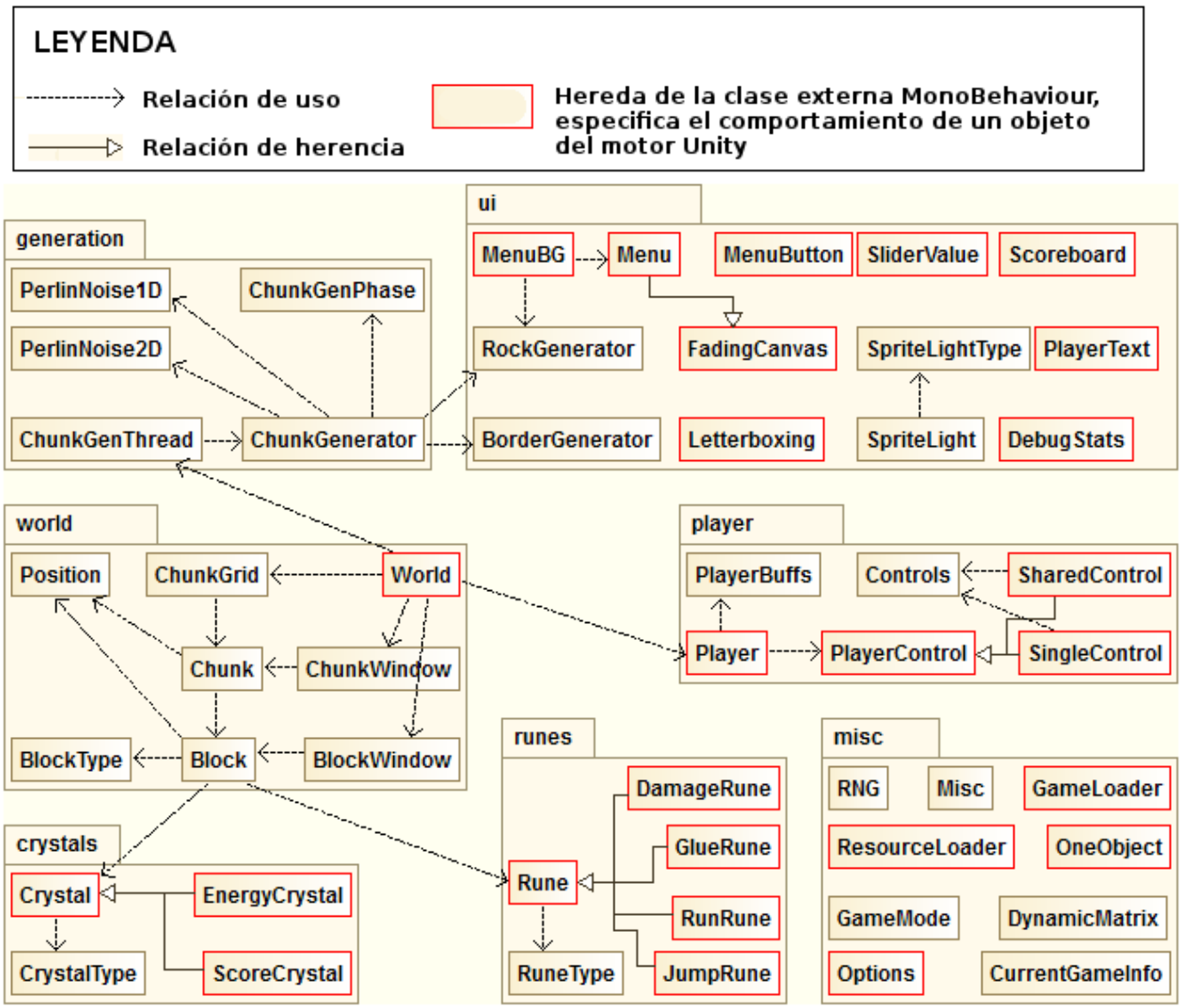

Figura 5 – Diagrama de clases

ChunkGenThread – Gestiona la generación asíncrona de *chunks* (partes en que se divide el mundo) según se necesiten en cada momento.

ChunkGenerator **–** Genera *chunks* siguiendo un proceso en varias fases.

ChunkGenPhase **–** Enumera las distintas fases de generación de *chunks*.

PerlinNoise1D **–** Genera de ruido de Perlin de una dimensión, utilizado para generar la superficie del planeta.

PerlinNoise2D **–** Genera de ruido de Perlin de dos dimensiones, utilizado para generar los sistemas de cuevas del planeta.

RockGenerator **–** Genera la textura rocosa usada para los bloques y el fondo de los menús.

BorderGenerator **–** Genera la textura del borde que separa los bloques frontales y los bloques del fondo.

***MenuBG* –** Crea el fondo usado para los menús e instancia el menú principal al inicio de la aplicación.

***Menu* –** Gestiona el comportamiento general de los menús y las características particulares de cada menú.

***MenuButton* –** Gestiona el comportamiento de los botones de los menús y contiene las funciones específicas que se ejecutan al pulsar cada botón.

***SliderValue* –** Programa el comportamiento de una opción de menú con *slider* (deslizador) para que actualice el valor correspondiente.

***FadingCanvas* –** Implementa el efecto de aparecer y desvanecerse progresivamente que utilizan algunos elementos de la interfaz.

***Letterboxing* –** Adapta la resolución del juego dependiendo de la proporción de la pantalla del dispositivo, añadiendo bandas negras si es necesario.

***SpriteLight* –** Contiene la lógica de la iluminación de los *sprites* (imágenes del juego), en función del tipo de *sprite* y su posición. En otras palabras, es el responsable de que el mundo se oscurezca lentamente conforme el jugador desciende por las cuevas.

***SpriteLightType* –** Enumera los tipos de iluminación que se aplican según el tipo de *sprite*.

***Scoreboard* –** Gestiona el marcador que muestra la puntuación y la energía restante del robot.

***PlayerText* –** Crea los indicadores que aparecen encima del robot cuando éste sufre algún efecto, como recuperar energía al recoger un cristal.

***DebugStats* –** Gestiona el texto que aparece en el modo *debug* (depuración de errores), que ofrece información sobre los *frames* por segundo, posición actual del robot, semilla del mundo actual, etc.

***Player* –** Controla toda la información relacionada con el robot: posición, velocidad, efectos activos, etc. Se encarga también de gestionar información sobre el estado y eventos de la partida. En el modo cooperativo, es el responsable de sincronizar toda esta información entre servidor y jugadores.

***PlayerBuffs* –** Contiene la información sobre algunos efectos que se aplican sobre el robot.

***PlayerControl* –** Proporciona una base común para las clases que controlan el comportamiento del robot.

***SingleControl* –** Controla el comportamiento del robot en el modo de un jugador, basándose en las acciones del usuario: mover izquierda, mover derecha, detenerse, saltar.

***SharedControl* –** Controla el comportamiento del robot en el modo cooperativo. Si la aplicación actúa como servidor, recoge las acciones enviadas por los jugadores y decide qué acción se ejecuta. Si la aplicación actúa como jugador, envía las acciones que el usuario quiere ejecutar.

***Controls* –** Recoge las acciones del usuario según la plataforma: del teclado si es PC o de la pantalla

táctil si es Android.

***World* –** Gestiona la información de los bloques que forman el mundo y coordina los algoritmos que deciden qué *chunks* deben generarse según el movimiento del personaje y qué bloques deben estar activos según si aparecen o no en pantalla.

***ChunkWindow* –** Decide qué *chunks* deben generarse según la posición actual del personaje y envía las peticiones correspondientes al generador de *chunks*.

***BlockWindow* –** Activa y desactiva los bloques según aparezcan o no en pantalla. Al activarse, el bloque genera un objeto de Unity a partir de la información del bloque. Al desactivarse, se elimina dicho objeto para liberar la memoria que ocupa.

***ChunkGrid* –** Almacena todos los *chunks* generados en una estructura de matriz infinita con coordenadas enteras. Además, contiene métodos para transformar las posiciones de bloques y *chunks* entre las coordenadas de la matriz y las de la escena, para calcular la posición de un bloque dentro de su *chunk*, el *chunk* al que pertenece un bloque, etc. En la figura 6 se muestra un ejemplo de la organización de la matriz de *chunks*.

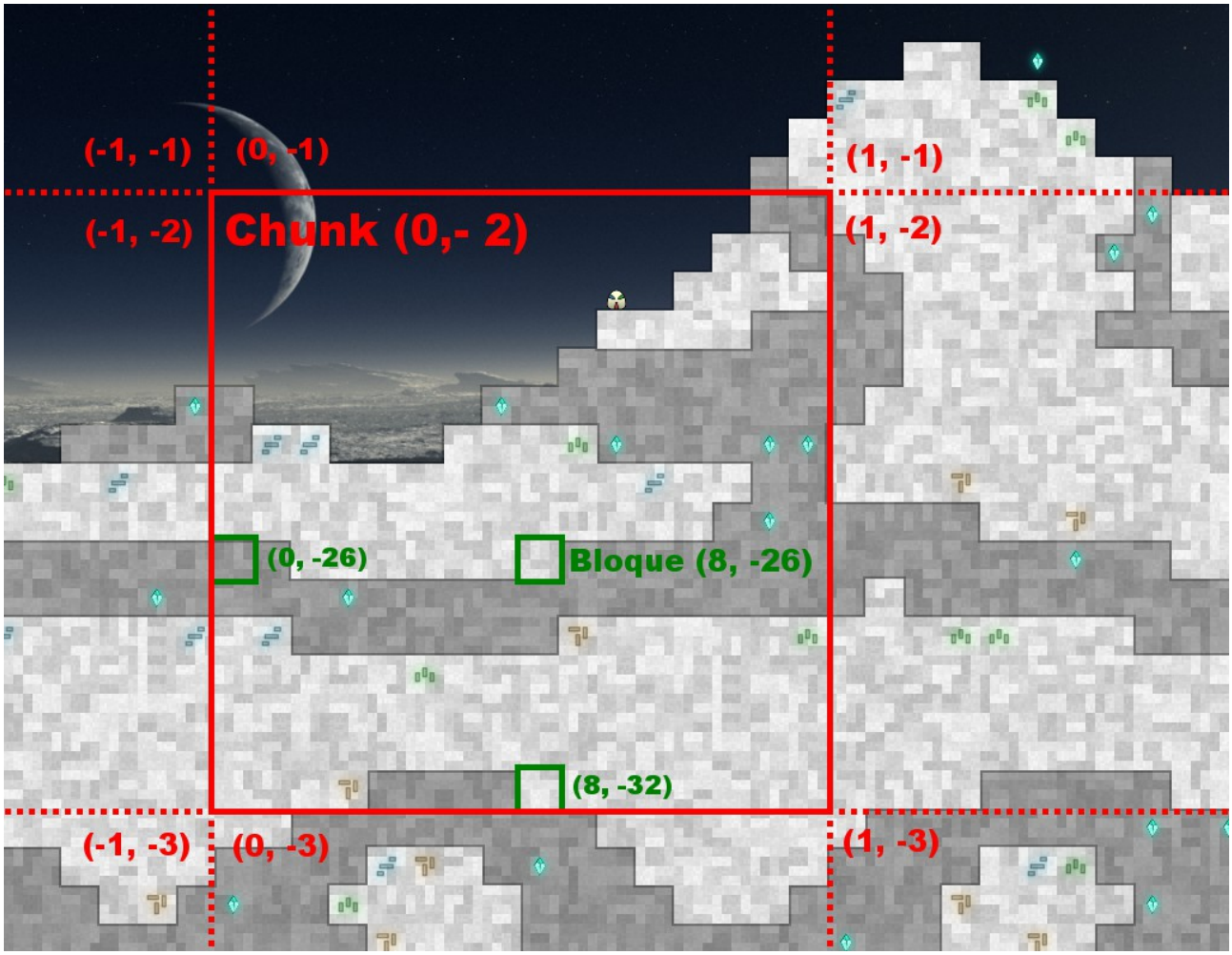

Figura 6 – Ejemplo de organización del mundo en chunks y bloques

***Chunk* –** Gestiona la información de un *chunk*: posición, bloques que contiene, *chunks* vecinos, etc.

***Block* –** Gestiona la información de un bloque: posición, tipo, runa y/o cristal que contiene, bloques vecinos, etc.

***BlockType* –** Enumera los tipos de bloques existentes.

***Position* –** Representa la posición de un bloque o *chunk* como coordenadas enteras de una matriz.

***Crystal* –** Implementa el comportamiento común a todos los cristales.

***CrystalType* –** Enumera los tipos de cristales existentes.

***EnergyCrystal* –** Implementa el comportamiento de un cristal de energía (cristal azul).

***ScoreCrystal* –** Implementa el comportamiento de un cristal de puntuación (cristal morado).

***Rune* –** Implementa el comportamiento común a todas las runas.

***RuneType* –** Enumera los tipos de runas existentes.

***DamageRune* –** Implementa el comportamiento de una runa de daño (runa roja).

***GlueRune* –** Implementa el comportamiento de una runa de pegamento (runa amarilla).

***RunRune* –** Implementa el comportamiento de una runa de correr (runa azul).

***JumpRune* –** Implementa el comportamiento de una runa de salto (runa verde).

***RNG* –** Genera números aleatorios de distintos tipos (entero, real, booleano) y distribuciones (uniforme, gaussiana).

***Misc* –** Contiene constantes y funciones de propósito general.

***GameLoader* –** Gestiona los procesos de la creación de una partida. En partidas cooperativas, se encarga de comunicar el servidor con los jugadores hasta que la partida está preparada para comenzar. Muestra información sobre estos procesos, como el porcentaje de carga, el número de jugadores que se han unido a la partida, mensajes de espera, etc.

***ResourceLoader* –** Inicializa algunas variables relacionadas con recursos del juego.

***OneObject* –** Realiza tareas iniciales, como la carga de las preferencias del usuario, y tareas que se mantienen durante toda la vida de la aplicación, como comprobar las teclas que permiten salir de la aplicación y activar/desactivar la opción de pantalla completa.

***GameMode* –** Enumera los diferentes modos de juego.

***DynamicMatrix* –** Estructura de datos que almacena una matriz dispersa de objetos con índices enteros. Está implementada como un diccionario de diccionarios (internamente tablas *hash*) con claves enteras. Se usa en *ChunkGrid* para almacenar los bloques y en *PerlinNoise2D* para almacenar los gradientes generados.

Options **–** Almacena las preferencias del usuario, las cuales se pueden modificar en el menú de opciones.

CurrentGameInfo **–** Almacena variables relacionadas con la partida actual: modo de juego, jugadores, puerto, dificultad, semilla, etc.

4.3.2. Vista de Componentes y Conectores y Vista de Despliegue

Los siguientes diagramas de componentes y conectores (figuras 7 y 9) y diagramas de despliegue (figuras 8 y 10) representan de forma general los componentes que se encuentran en ejecución en el sistema en un momento dado y en qué máquina se ejecuta cada componente, tanto en una partida para un jugador como durante una partida cooperativa. Debajo de cada diagrama, se detallan los elementos más relevantes.

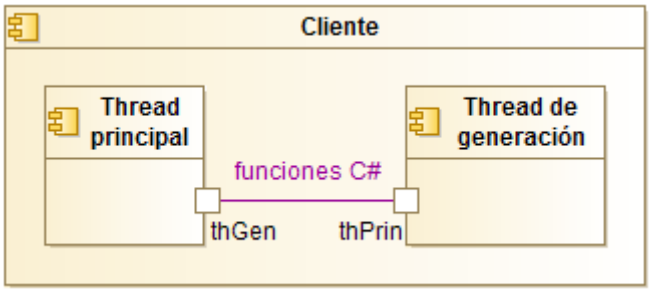

Figura 7 – Diagrama de componentes y conectores para partida de un jugador

Cliente **–** Componente que engloba a todo el código que se ejecuta en la máquina del usuario. Contiene un *Thread principal* y un *Thread de generación*, que se comunican entre sí mediante funciones C#.

Thread principal **–** Componente controlado por Unity, encargado de cargar las escenas, manejar los recursos, ejecutar los comportamientos programados para cada objeto de la escena, etc. Al iniciar la partida, crea un *Thread de generación*, al cual envía peticiones para que se vayan generando las partes del mundo que sean necesarias.

Thread de generación **–** Componente que se encarga de generar trozos del mundo según se vayan necesitando. Después de generar cada trozo, envía un aviso al *Thread principal*.

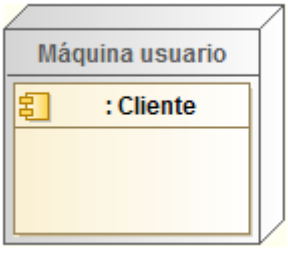

Figura 8. Diagrama de despliegue para partida de un jugador

Máquina usuario **–** Es la máquina en la que se ejecuta la aplicación.

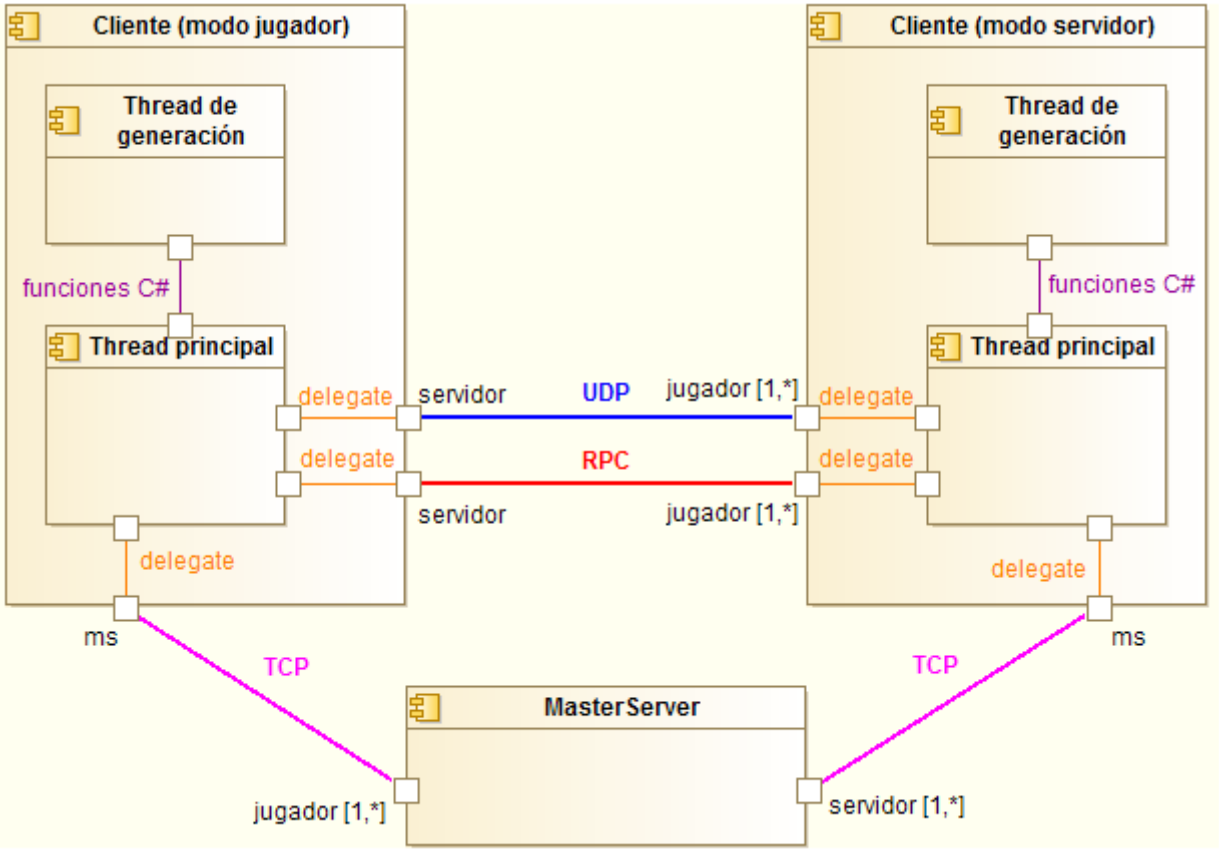

Figura 9 – Diagrama de componentes y conectores para partida cooperativa

***MasterServer* –** Componente externo, cuyo código lo proporciona Unity. Se ejecuta en una máquina remota y se encarga de manejar la lista de partidas y de poner en contacto a jugadores y servidores.

***Cliente (modo servidor)* –** Similar al componente *Cliente* descrito anteriormente, pero con código específico para servidor de partida cooperativa. Al crear una partida, la registra en un *MasterServer*, el cual le ofrece los datos de los jugadores que se conectan a la partida.

***Cliente (modo jugador)* –** Similar al componente *Cliente* descrito anteriormente, pero con código específico para jugador de partida cooperativa. Para unirse a una partida, se registra en un *MasterServer* como jugador y éste le asigna un servidor que haya creado previamente una partida cooperativa y le envía los datos del mismo.

***Conector UDP* –** Lo utiliza el servidor enviar periódicamente el estado de la partida a los jugadores.

***Conector RPC* –** Lo utilizan servidor y jugadores para enviar y recibir distintos mensajes, como las acciones de cada jugador o algunos eventos de la partida.

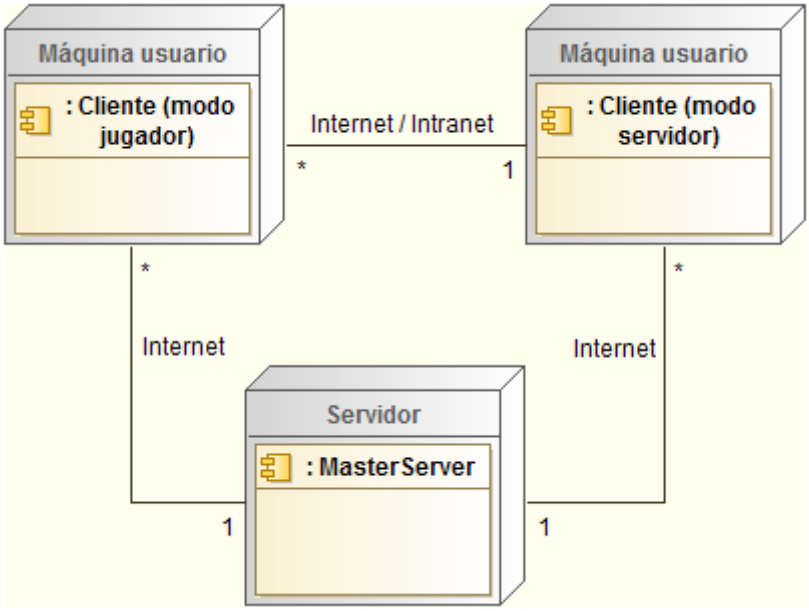

Figura 10 – Diagrama de despliegue para partida cooperativa

Máquina usuario **–** Son las máquinas en las que se ejecuta una instancia de la aplicación, ya sea como jugador o como servidor. Dichas máquinas pueden comunicarse a través de Internet o de red local (intranet).

Servidor **–** Es la máquina en la que se ejecuta el *MasterServer*. Está accesible al resto de máquinas a través de Internet.

4.3. Algoritmo de generación de mundo

Este algoritmo genera el planeta por el que se mueve el robot a partir de una semilla (número entero) que se elige aleatoriamente al inicio de la partida. El mundo creado está completamente definido por la semilla, lo que posibilita que en una partida cooperativa cada cliente pueda generar su mundo local sólo con saber la semilla y no tenga que recibirlo del servidor.

Al iniciar la partida, el algoritmo genera mundo en un radio alrededor del punto inicial del robot. Cuando éste se mueve, se genera poco a poco más mundo en la dirección de desplazamiento. El mundo está dividido en *chunks*, que son trozos de 16x16 bloques, los cuales son generados por un hilo separado, en concurrencia con el hilo principal.

El proceso para generar un *chunk* se divide en varias fases (ver figura 11):

Fase 0: Instanciación **–** Se construye el objeto que almacenará el *chunk* en la matriz. Se le asigna una semilla específica para el *chunk*, basada en la semilla principal y la posición del *chunk*. Esto asegura que el orden en que se generan los *chunks* no influya en el resultado.

Fase 1: Superficie **–** Se genera la forma de la superficie del planeta, es decir, la altura máxima en la que aparecen bloques en cada columna de la matriz. Se utiliza para ello ruido de Perlin 1D con 4 octavas (ver anexo A para más información sobre la generación y uso del ruido de Perlin). Las alturas generadas se guardan en una estructura de datos aparte, de forma que sean reutilizables al generar otros *chunks* de la misma columna.

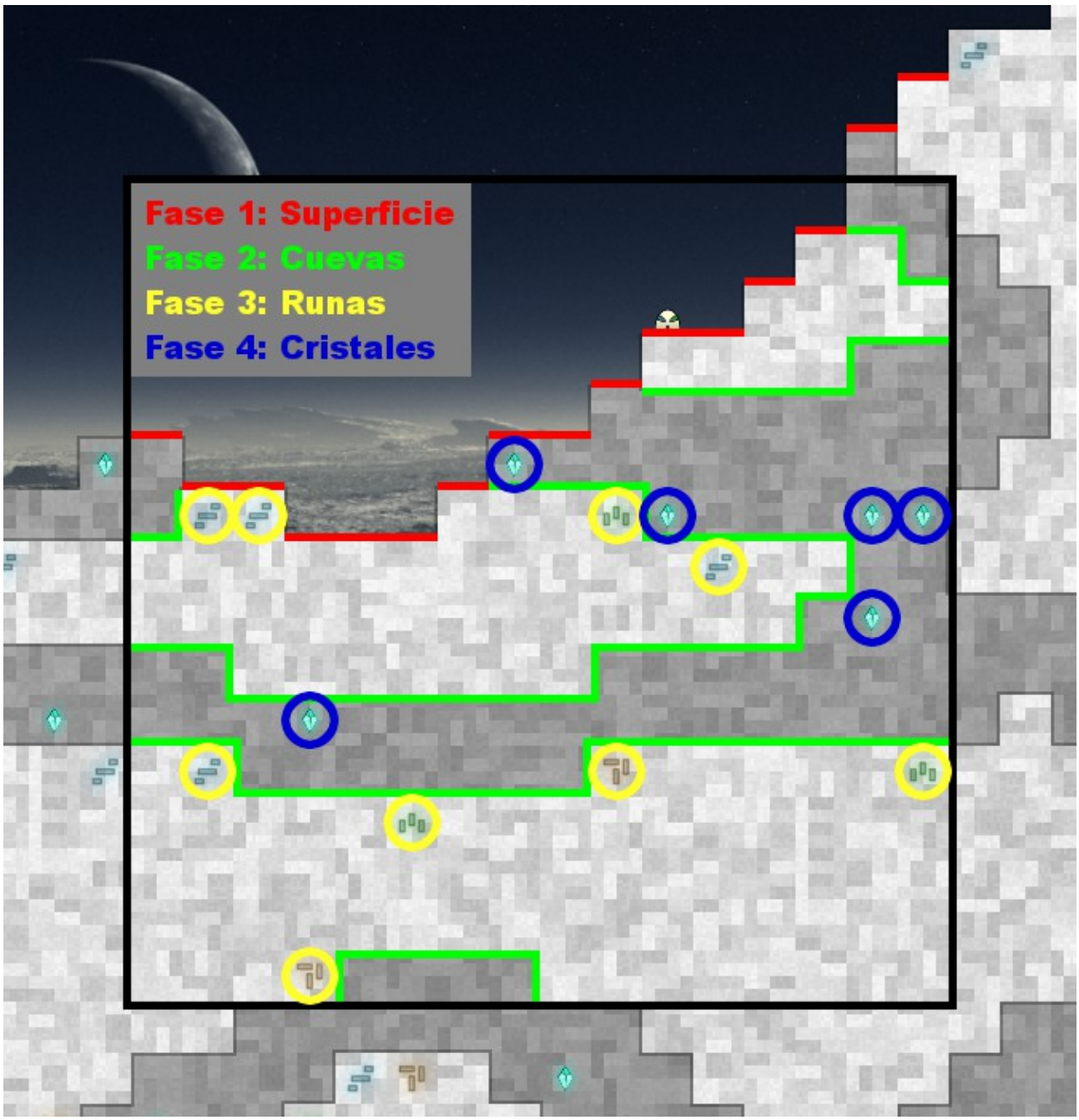

Figura 11 – Generación de un chunk dividida en fases

Fase 2: Cuevas **–** Se generan los sistemas de cuevas que recorre el robot y se rellena el *chunk* con bloques de frente de cueva, bloques de fondo de cueva y bloques vacíos (los que están por encima de la superficie). En la generación se utiliza ruido de Perlin 2D con sólo 1 octava, ya que este ruido es mucho más costoso de generar que el de una dimensión.

Fase 3: Runas **–** Se generan runas en algunos bloques frontales. Para determinar la probabilidad de que una runa se genere se tienen en cuenta tanto la altura de cada bloque como el tipo de los bloques vecinos, aplicando ciertas reglas como que haya más probabilidad de generar runas rojas en bloques más profundos y que no sea posible generar una runa de salto en el techo de una cueva.

Fase 4: Cristales **–** Se generan los cristales. Al igual que en la fase anterior, se tiene en cuenta la altura del bloque para determinar la probabilidad de que se genere un cristal en ese bloque.

Cada fase requiere que la anterior esté finalizada, pero además, la fase 3 requiere que la fase 2 esté finalizada para todos los *chunks* vecinos. Esto es debido a que se necesita saber de qué tipo son los bloques vecinos a cada bloque, incluso para los bloques que están en el borde del *chunk*. Por ello, al iniciar la fase 3 se comprueba el estado de generación de los *chunks* vecinos, y se generan los *chunks* que hagan falta hasta que completen la fase 2.

En la figura 12 se muestra un diagrama de secuencia que describe cómo el hilo principal y el hilo de generación gestionan la generación de *chunks*.

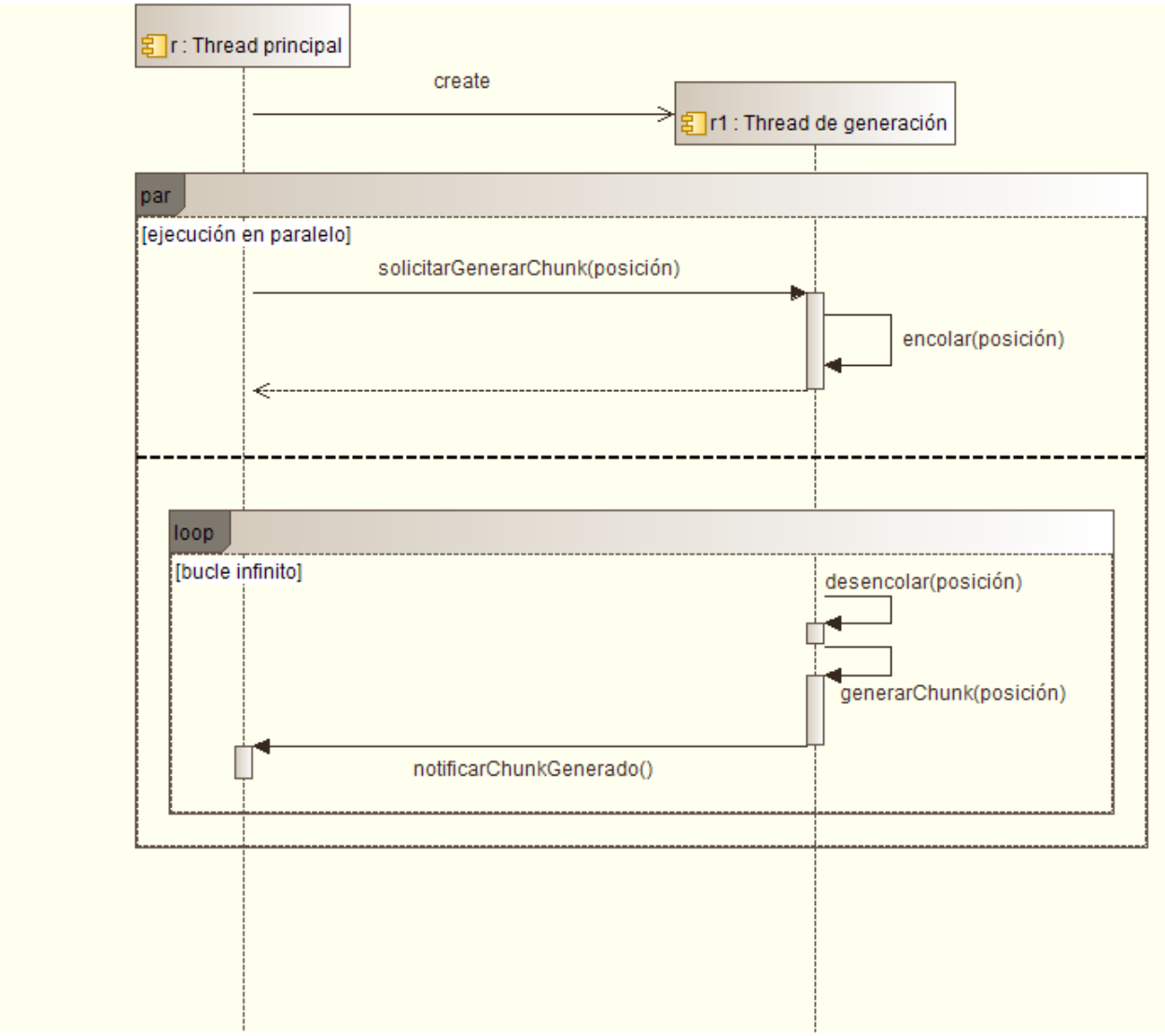

Figura 12 – Diagrama de secuencia para la generación de mundo

Como se ve en el diagrama, se ejecutan dos interacciones en paralelo. Cuando el hilo principal necesita generar un *chunk*, le envía una petición al hilo de generación, indicando la posición del chunk que se quiere generar. Por su parte, el hilo de generación almacena estas posiciones en una cola y de forma constante comprueba si hay nuevos *chunks* que generar. Cuando un *chunk* ha terminado de generarse, el hilo de generación lo notifica al hilo principal.

4.4. Algoritmo de control compartido

Este algoritmo permite a varios jugadores controlar a un único personaje. Los jugadores envían continuamente las acciones que desean realizar al servidor, el cual cada pocos milisegundos selecciona las acciones a realizar.

Para dicha selección, se consideran dos tipos de acciones independientes: dirección y salto. El tipo dirección tiene tres posibles valores: izquierda, derecha y no moverse, mientras que el tipo salto tiene dos valores: saltar y no saltar. Para cada tipo, se selecciona la acción más votada por los jugadores, y en caso de empate se selecciona la acción nula (no moverse, no saltar).

La figura 13 muestra los cuatro procesos que ocurren simultáneamente:

– Una vez por *frame* (imagen mostrada por el juego), cada jugador envía mediante una llamada RPC las acciones elegidas (dirección y salto), pero sólo si ha habido algún cambio respecto al *frame* anterior. El servidor las recibe y las guarda en una estructura de datos con las últimas acciones recibidas de cada jugador.

– Una vez cada 33 milisegundos, el servidor selecciona, mediante el proceso descrito anteriormente, las acciones que se ejecutarán durante los próximos 33 milisegundos.

– Una vez cada *frame*, el servidor ejecuta las acciones actuales (el salto sólo se ejecuta la primera vez, pero el movimiento lateral se ejecuta de forma continua).

– Una vez cada 33 milisegundos, el servidor envía vía UDP el estado actual de la partida a los jugadores, y éstos lo actualizan.

Según las pruebas realizadas con usuarios, el mínimo número de jugadores para lograr resultados aceptables es de 3. Con 2 jugadores se producen constantemente empates cuando uno de los jugadores quiere ir a la izquierda y el otro a la derecha, lo que deja al robot parado. Las pruebas con 5 jugadores (el máximo con el que se ha podido probar) son las que han obtenido mejores resultados. Sin embargo, aumentar mucho el número de jugadores, a pesar de que favorece la coordinación, proporciona al jugador menos sensación de control sobre el robot, por lo que se sospecha que el número óptimo debería estar entre 5 y 7 jugadores.

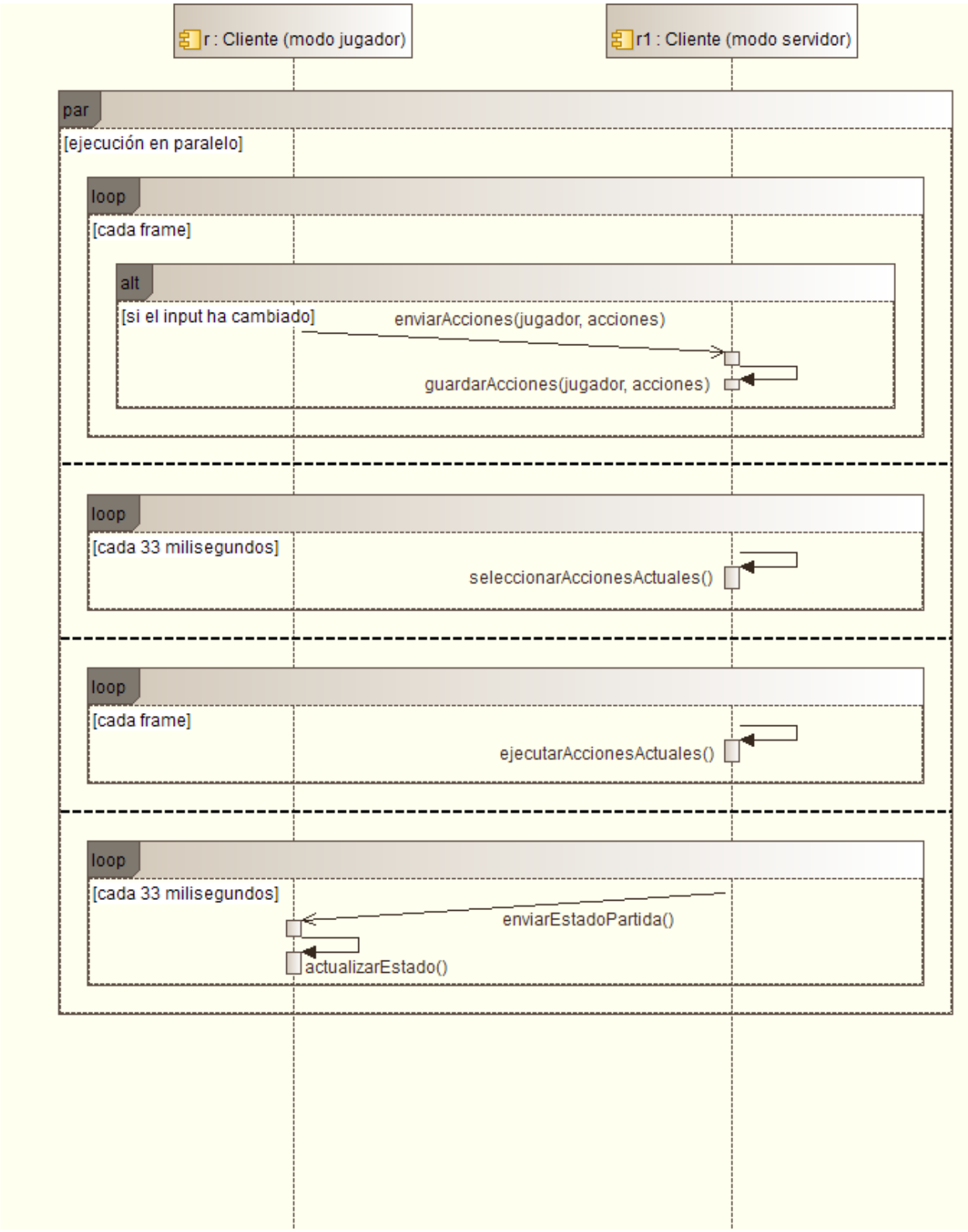

Figura 13 – Diagrama de secuencia de para el control compartido

4.5. Algoritmo de generación de la textura rocosa

La textura rocosa utilizada para los bloques y el fondo de los menús también se genera

proceduralmente, pero para ahorrar en tiempo y en memoria (ya que habría que almacenar por separado la textura de cada bloque) se generan 36 texturas diferentes al inicio de la partida, y se utilizan éstas con diferentes rotaciones y volteos (en total 288 posibilidades).

Para generar una textura se utiliza uno de los patrones que hay definidos. Un patrón es una cuadrícula de 4x4 que se rellena con diferentes colores siguiendo una sola regla: la cuadrícula debe estar dividida en parejas de casillas adyacentes, en horizontal o vertical. Existen 36 patrones distintos que cumplen esta regla, algunos de los cuales se muestran como ejemplo en la figura 14, en la que los números representan la pareja a la que pertenece cada casilla.

El siguiente paso es colorear cada pareja con un color aleatorio escogido entre 16 tonos de gris. Las dos casillas de cada pareja deben tener el mismo color, pero no hay ninguna otra restricción en cuanto a selección de colores. En la figura 14 se muestra el resultado de este proceso, pero se han utilizado sólo 4 colores para mayor claridad.

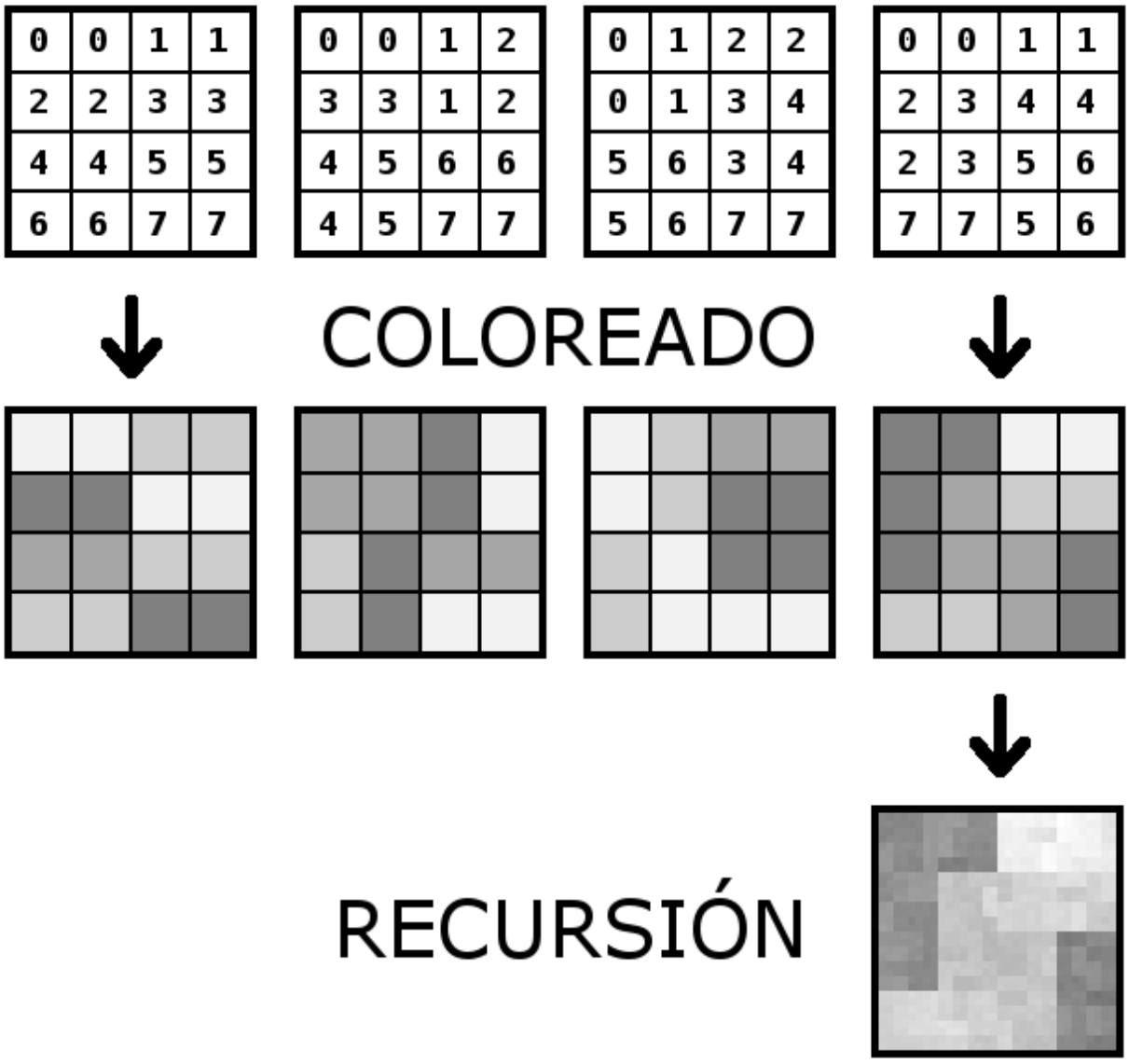

Figura 14 – Generación de la textura rocosa

El proceso que completa el algoritmo es la recursión, utilizada para crear un efecto fractal. Las texturas generadas son cuadrados de 64x64 píxeles, mientras que los patrones son de 4x4, por lo que cada casilla del patrón corresponde en la textura final a un "subcuadrado" de 16x16 píxeles. Con el algoritmo presentado hasta ahora estos cuadrados estarían rellenos de un solo color (ver figura 14). Para dotar a la textura de un aspecto más realista, se aplica el mismo algoritmo a cada subcuadrado, utilizando como color base el color asignado a ese subcuadrado por el algoritmo inicial y con una variación de color mucho[3] menor que el usado en el algoritmo inicial. Esta recursión se aplica de la misma forma una segunda vez para rellenar los subcuadrados de 4x4 píxeles dentro de cada subcuadrado 16x16 píxeles.

3 En concreto 16 veces menor, ya que se usan 16 grises distintos. De esta forma, los colores del subcuadrado son más cercanos a su color base que a los colores de otro subcuadrado con distinto color base.

5. RESULTADOS

En este apartado se presenta una serie de capturas del programa final, explicando debajo de cada captura a qué parte del programa corresponde y aspectos destacados de la misma.

Figura 15 – Menú principal

La figura 15 muestra el menú principal del juego, tal como se muestra al iniciar el programa. El encabezado muestra el nombre del juego y la versión. Las opciones que aparecen se explican en el apartado 4.2 (casos de uso). Observar que la tercera opción aparece resaltada porque se ha pasado el ratón sobre ella.

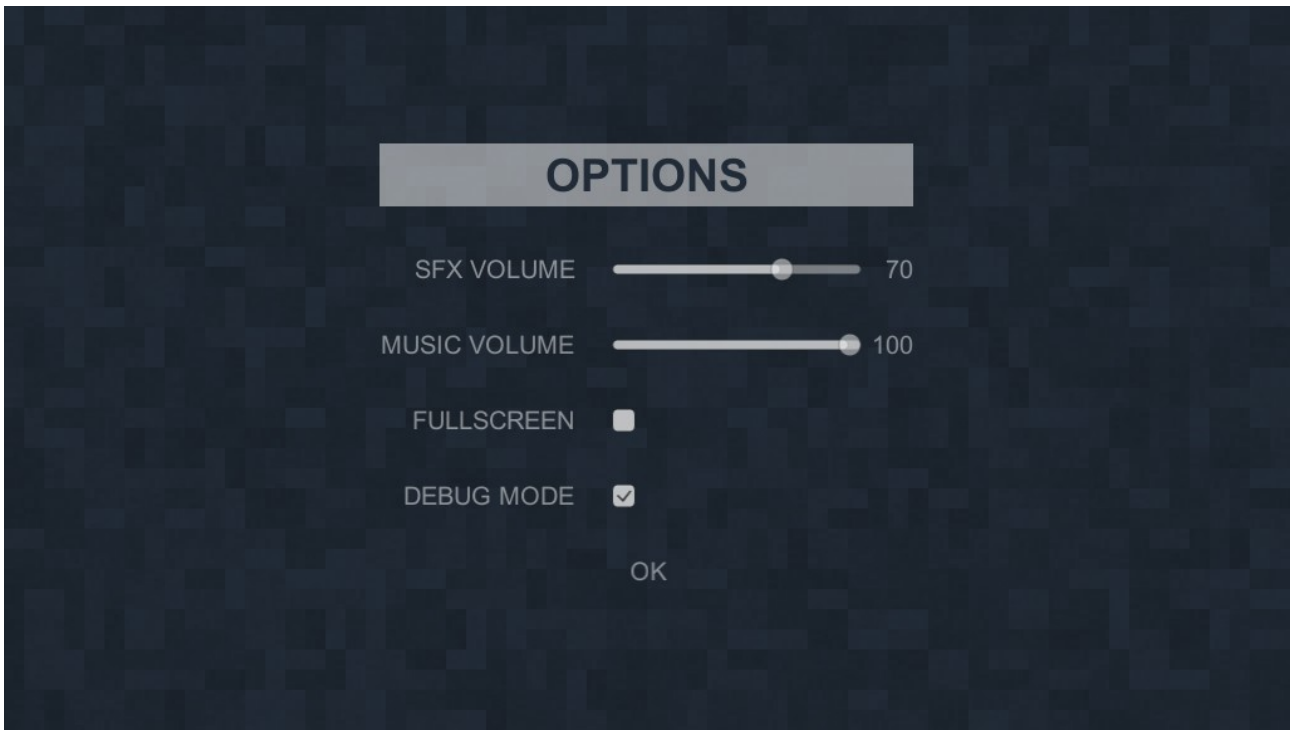

Figura 16 – Menú de opciones

La figura 16 muestra el menú de opciones. Las dos primeras controlan el volumen de efectos y música, y las restantes activan o desactivan la pantalla completa y el modo de depuración de errores.

Figura 17 – Inicio de la partida

La figura 17 muestra el aspecto habitual del inicio de una partida, justo después de que el robot aterrice en la superficie del planeta. Notar los tres planos diferentes: bloques frontales (roca blanca),

bloques de fondo (roca gris) y fondo. Arriba a la derecha aparece el marcador, con los valores iniciales de puntuación y energía.

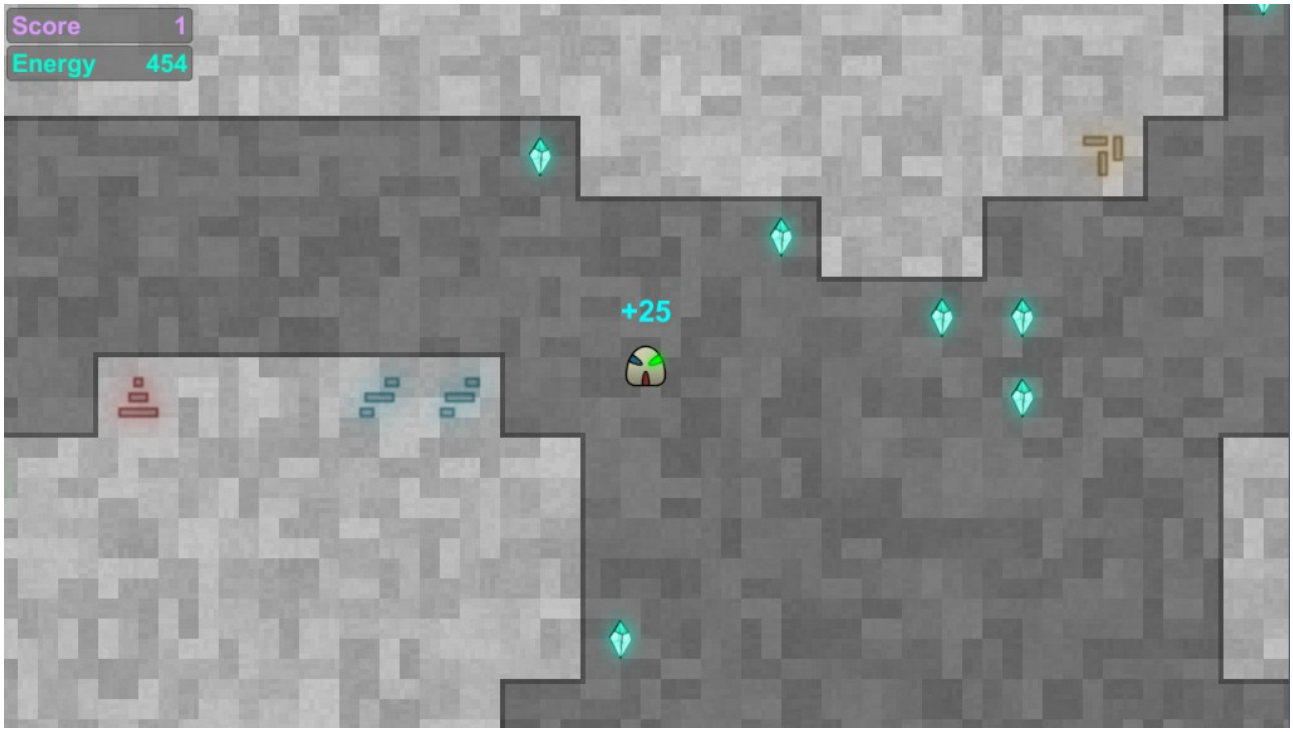

Figura 18 – Recogiendo cristales

La figura 18 muestra como el robot recoge un cristal de energía. Sobre el robot aparece la energía restaurada. Observar cómo se iluminan las luces del robot según las acciones del jugador (en este caso, ir hacia la derecha).

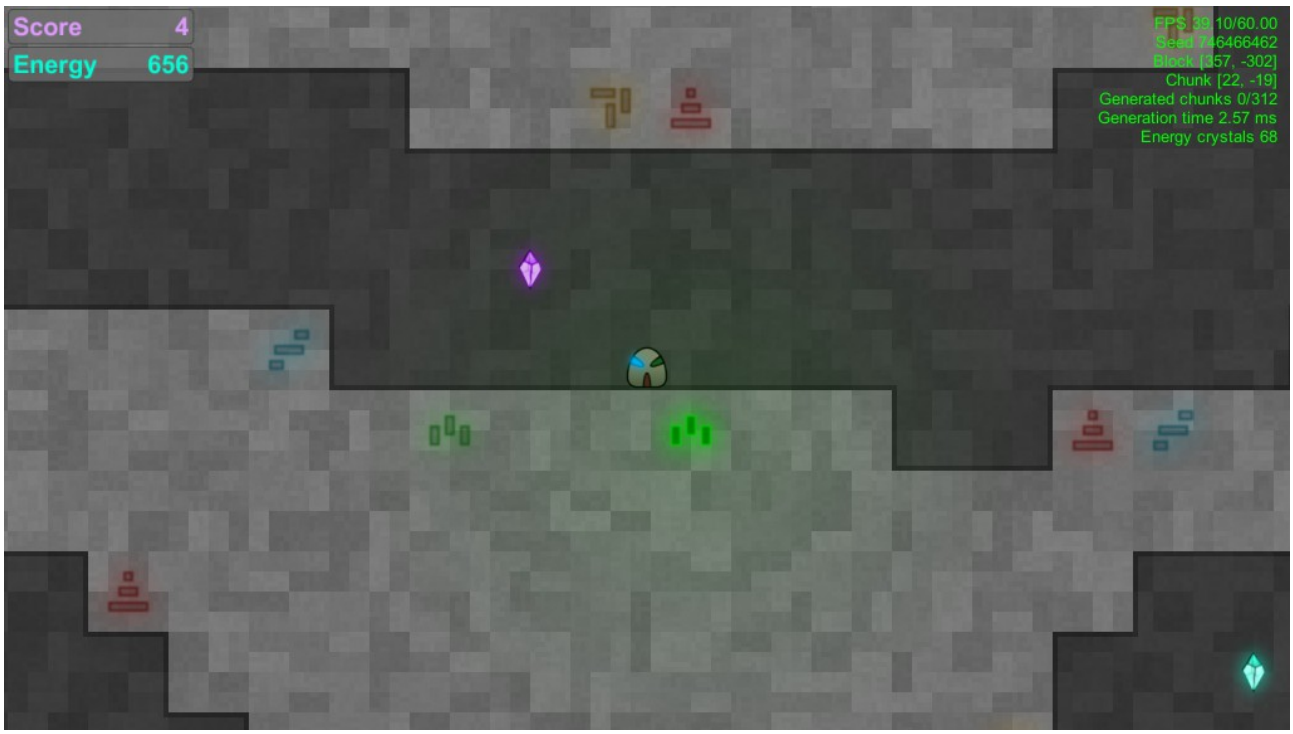

Figura 19 – En las profundidades

La figura 19 muestra al robot activando una runa de salto al pasar sobre ella, y a punto de recoger un cristal de puntuación. Se ha activado el modo de depuración de errores, por lo que puede verse información de interés en la esquina superior derecha de la pantalla, incluyendo la posición actual del robot: bloque (357, -302) y *chunk* (22, -19). Notar también que el ambiente es más oscuro que en la anterior captura, ya que el robot se encuentra a una altura mucho más profunda.

6. CONCLUSIONES

Como se expone en el apartado 2, el objetivo general del trabajo era desarrollar el prototipo de un videojuego que explorara la idea de control compartido. Dentro de las limitaciones del proyecto, el objetivo se ha completado, desarrollando un sencillo videojuego con todas las características necesarias: personaje, mundo que le rodea, efectos de sonido, música e interfaz, etc.

A este videojuego se ha añadido un modo multijugador cooperativo que utiliza el sistema de control compartido para dirigir las acciones del personaje. En las pruebas realizadas con este modo el sistema ha funcionado de forma estable y con una latencia aceptable. Los jugadores han logrado coordinarse de manera satisfactoria y han considerado la experiencia como divertida, especialmente cuando los jugadores se encontraban en la misma sala y podían comunicarse entre ellos fácilmente.

7. TRABAJO FUTURO

A partir de los resultados de este trabajo, se pretende continuar con el desarrollo, con el objetivo futuro de lanzar el juego comercialmente en la plataforma de distribución digital Steam. Para ello será necesario refinar algunos de los apartados del proyecto y añadir nuevas características. Las mejoras proyectadas a medio plazo son:

– Generación de mundo más variada y compleja, con zonas del planeta diferenciadas, que detecte los posibles caminos que puede tomar el jugador y coloque las plataformas, las runas y los cristales de forma inteligente en función de la dificultad de la partida.

– Mejoras en la interfaz del juego, en algunos de los recursos gráficos y en la variedad de la música.

– Cola de emparejamientos automáticos para partidas cooperativas, que permita formar partidas agrupando a los jugadores en función de su experiencia y habilidad.

– Sistema de *pings* con el que los jugadores puedan usar el ratón para señalar al resto de jugadores una parte de la pantalla, como modo de comunicación.

– Otras mejoras: tutorial, más tipos de cristales, zonas secretas, animaciones del robot, traducción a distintos idiomas, posibilidad de ver estadísticas de las partidas, posibilidad de guardar el mundo generado como una imagen, etc.

8. DIAGRAMA TEMPORAL

La realización del trabajo ha llevado un total de 340,83 horas. En el siguiente diagrama de Gantt (figura 20) se muestra el desarrollo de las tareas realizadas a lo largo del tiempo. En el anexo B se explican en detalle dichas tareas, así como otros aspectos relacionados con la gestión del trabajo.

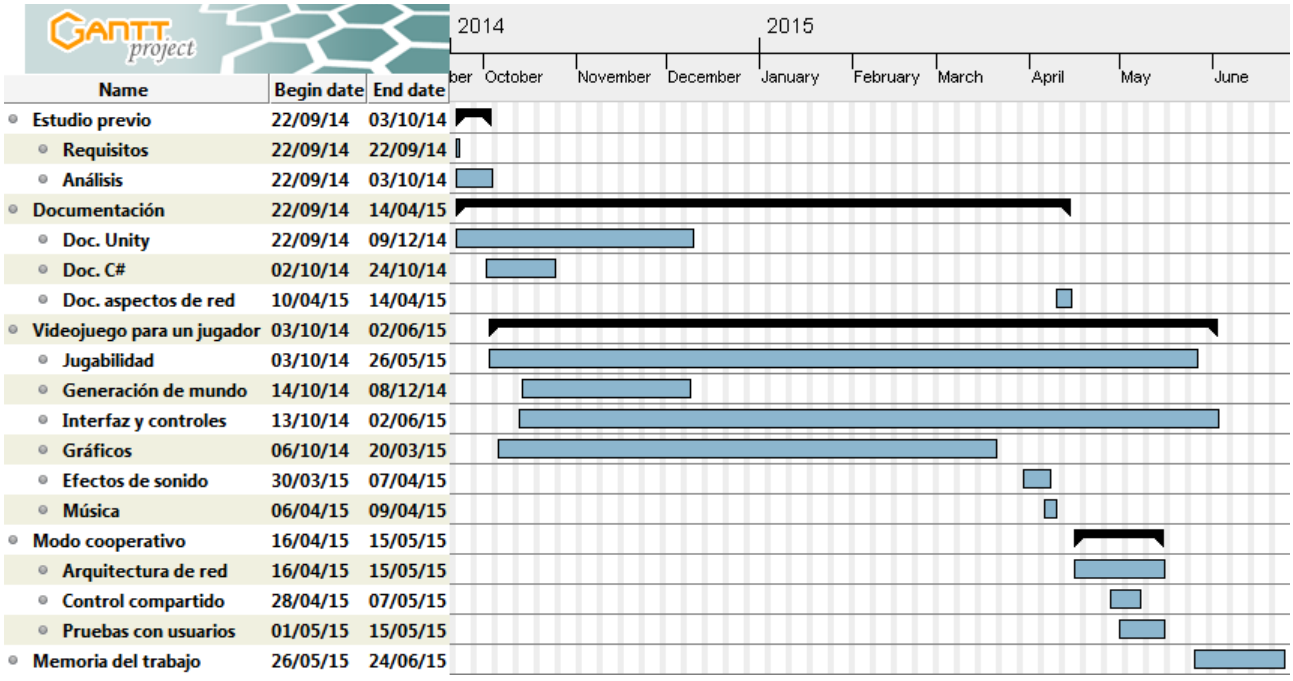

Figura 20 – Distribución temporal de las tareas realizadas durante las 340,83 horas

9. BIBLIOGRAFÍA

Toda la bibliografía aquí enumerada se encuentra disponible en inglés. Las páginas web de este apartado han sido accedidas por última vez el 15 de junio de 2015.

Estudio previo

– *Introducing Shared Character Control to Existing Video Games*, Anna Loparev, Walter S. Lasecki, Kyle I. Murray y Jeffrey P. Bigham, 2014. Disponible en: http://repository.cmu.edu/hcii/277

– *Real-time Crowd Control of Existing Interfaces*, Walter S. Lasecki, Kyle I. Murray, Samuel White, Robert C. Miller y Jeffrey P. Bigham, 2011. Disponible en: *http://www.cs.rochester.edu/hci/pubs/pdfs/legion.pdf*

Documentación

– Tutoriales sobre Unity Engine. Disponibles en: *https://unity3d.com/learn/tutorials/modules*

Generación de mundo

– *How to Make Insane, Procedural Platformer Levels*, Jordan Fisher, 2012. Disponible en: *http://www.gamasutra.com/view/feature/170049/How_to_Make_Insane_Procedural_Platformer_Levels_.php*

– Artículo "Perlin Noise" en Wikipedia. Disponible en: *https://en.wikipedia.org/wiki/Perlin_noise*

– Glosario de la librería de ruido coherente "libnoise". Disponible en: *http://libnoise.sourceforge.net/glossary*

Arquitectura de red

– Artículo "Source Multiplayer Networking" en la wiki de desarrolladores de Valve. Disponible en: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

– *Legacy Network Reference Guide*, Unity Engine. Disponible en: *http://docs.unity3d.com/Manual/NetworkReferenceGuide.html*

Optimización

– Reducing Memory Usage in Unity, C# and .NET/Mono, Andrew Fray, 2013. Disponible en: *https://andrewfray.wordpress.com/2013/02/04/reducing-memory-usage-in-unity-c-and-netmono*

ANEXO A – UTILIZACIÓN DE RUIDO PARA LA GENERACIÓN DE MUNDO

Como se explicó en el documento principal, se ha utilizado ruido de Perlin para generar el mundo. Un ruido es una función matemática (o algoritmo) que para cada punto del espacio (n-dimensional) devuelve un valor real, generalmente entre -1 y 1. El ruido de Perlin deriva de los llamados ruidos coherentes[4], los cuales tienen tres propiedades:

1. El mismo valor de entrada genera siempre el mismo valor de salida.
2. Un cambio pequeño en el valor de entrada produce un cambio pequeño en el valor de salida.
3. Un cambio grande en el valor de entrada produce un cambio aleatorio en el valor de salida.

La primera propiedad permite que se pueda generar el ruido a trozos (*chunks*), sin que existan discontinuidades entre trozos generados por separado. La segunda asegura precisamente dicha continuidad (coherencia) del mundo generado. La tercera otorga una apariencia de aleatoriedad y no predictibilidad al mundo.

El ruido de Perlin se genera sumando varias muestras de ruido coherente (generado en tiempo de ejecución, utilizando un algoritmo sencillo[5]) de distintas frecuencias, llamadas octavas. Cada octava tiene el doble de frecuencia que la anterior y la mitad de peso en el ruido final. El resultado es un ruido fractal que, como se muestra en las figuras 21 y 22, genera formas más parecidas a las naturales: sistemas montañosos, humo, etc.

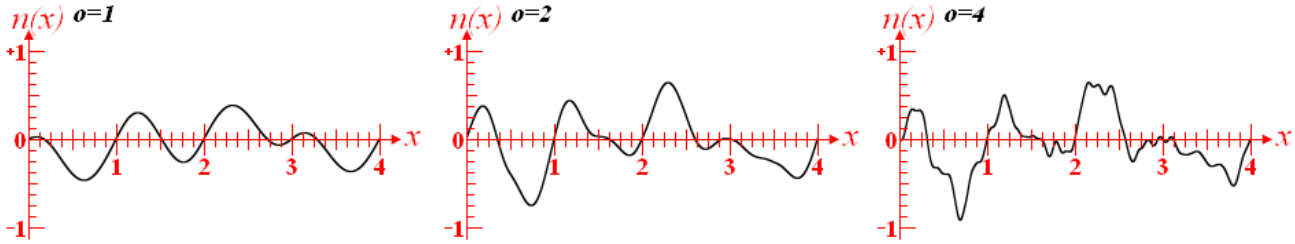

Figura 21 – Ruido de Perlin de una dimensión con 1, 2 y 4 octavas

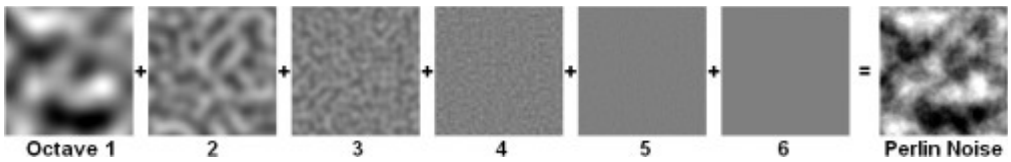

Figura 22 – Ruido de Perlin de dos dimensiones, obtenido al sumar 6 octavas

En la figura 23 se muestra un ejemplo simplificado de cómo se generan las cuevas a partir de ruido de Perlin de dos dimensiones. Si el valor del ruido para la posición del bloque está entre -0.1 y 0.1, se genera un bloque de fondo (color oscuro). En caso contrario, se genera un bloque frontal (color claro).

4 Más información en la página web de la librería de ruido coherente "libnoise": http://libnoise.sourceforge.net/glossary

5 Puede verse la idea básica del algoritmo utilizado en Wikipedia: http://en.wikipedia.org/wiki/Perlin_noise

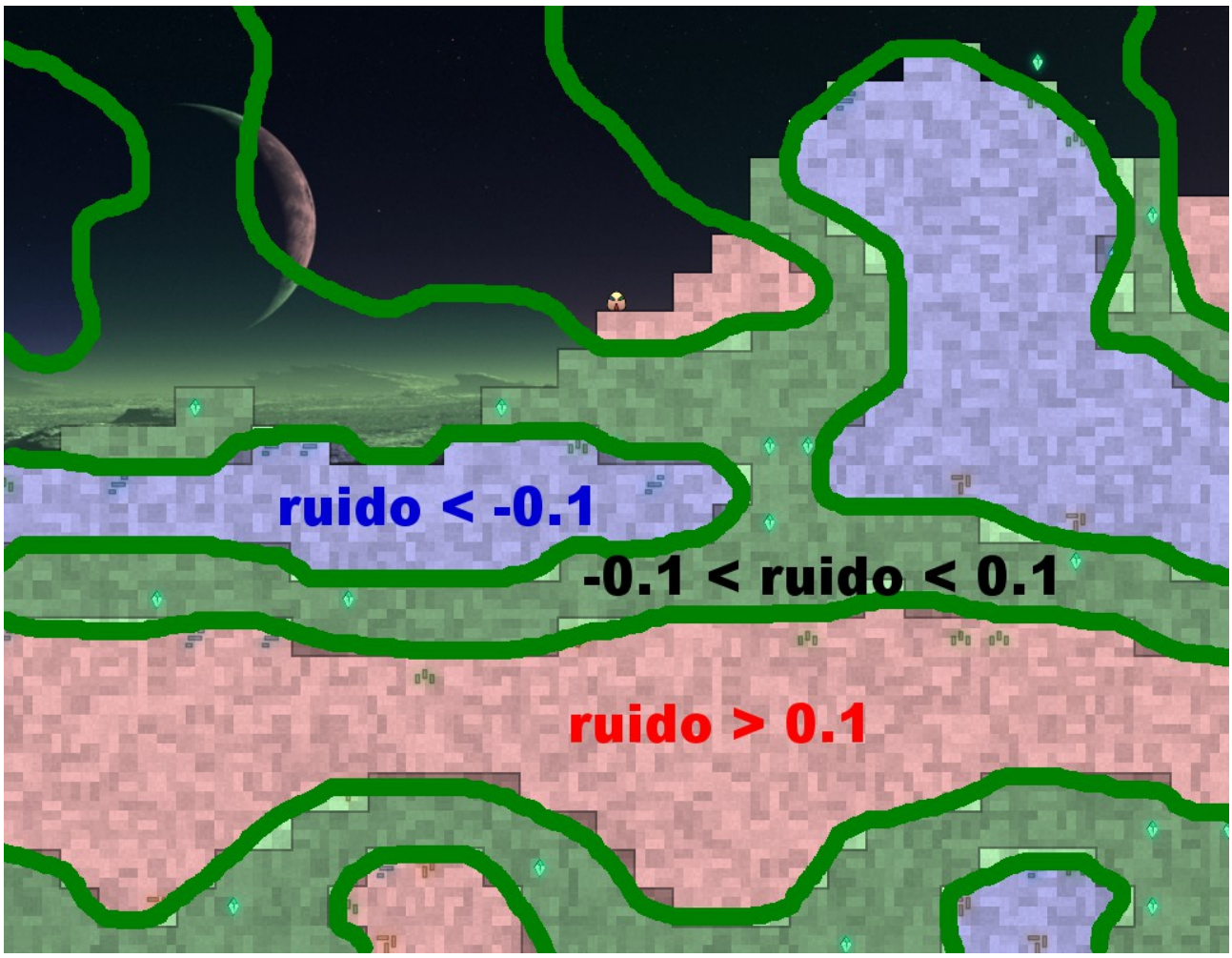

Figura 23 – Generación de cuevas a partir de ruido de Perlin 2D

ANEXO B – GESTIÓN DEL TRABAJO

En este anexo se exponen aspectos relacionados con la gestión del trabajo realizado.

B.1. Planificación

Al inicio del trabajo se diseñó una lista de tareas generales a realizar, asignando un tiempo estimado a cada una, teniendo en cuenta un objetivo de 300 horas para la totalidad del trabajo. La tabla 7 recoge la lista de tareas, junto con su tiempo estimado y su tiempo real (expresados en minutos y horas). Las tareas marcadas con un asterisco se realizaron parcialmente en un trabajo para la asignatura de Videojuegos, el cual fue la base para el presente trabajo. El tiempo dedicado a esas tareas en el anterior trabajo no se incluye en la tabla.

TAREA	ESTIM. (m)	ESTIM. (h)	REAL (m)	REAL (h)
Estudio previo del problema	**1320**	**22**	**950**	**15,83**
Requisitos *	120	2	90	1,5
Análisis *	1200	20	860	14,33
Documentación sobre las tecnologías a utilizar	**3480**	**58**	**2440**	**40,67**
Documentación sobre Unity	2400	40	1590	26,5
Documentación sobre C#	600	10	530	8,83
Documentación sobre aspectos de red *	480	8	320	5,33
Desarollo del videojuego para un jugador	**8700**	**145**	**11620**	**193,67**
Jugabilidad	1200	20	2270	37,83
Generación de mundo	2400	40	4430	73,83
Interfaz y controles	1500	25	1390	23,17
Gráficos	2100	35	2490	41,5
Música	1200	20	270	4,5
Efectos de sonido	300	5	770	12,83
Desarollo del modo cooperativo	**3000**	**50**	**3360**	**56**
Arquitectura de red	2100	35	2370	39,5
Algoritmo de control compartido	600	10	510	8,5
Pruebas con usuarios	300	5	480	8
Memoria del trabajo	**1500**	**25**	**2080**	**34,67**
TOTAL	**18000**	**300**	**20450**	**340,83**

Tabla 7 – Distribución de tareas

El trabajo comenzó con un estudio (parcialmente ya realizado) sobre las características del videojuego que se quería desarrollar y los problemas más importantes que se deberían afrontar: generación de mundo, control compartido, etc. A partir de ahí, se generaron unos requisitos y se plantearon ideas generales para la resolución de estos problemas.

El segundo paso fue consultar la documentación disponible sobre el motor Unity y el lenguaje C# (usado por Unity para programar comportamientos de los objetos). Cuando se tuvo cierto dominio de las herramientas a utilizar, se comenzó a desarrollar el videojuego, centrándose en el modo para un jugador.

Cuando ya se tuvo el primer prototipo con características mínimas de jugabilidad (personaje,

bloques, movimiento, saltos, etc.) se comenzó el diseño del sistema de generación procedural del mundo, el cual llevó mayor tiempo del esperado debido a su complejidad, hasta que finalmente se decidió dar por terminado, no habiendo dado tiempo a introducir todas las características que se plantearon en un principio.

Una vez se completaron las características principales del videojuego, se comenzó a extender el mismo para permitir jugar en red. Se adaptó el juego para que admitiera varios jugadores en red, de forma que cualquiera de ellos pudiera controlar al personaje. Más tarde, se implementó el sistema de control compartido y, cuando estuvo terminado, se realizaron pruebas con usuarios.

En el siguiente diagrama de Gantt (figura 24) se muestra el desarrollo de las tareas anteriores a lo largo del periodo de realización del trabajo.

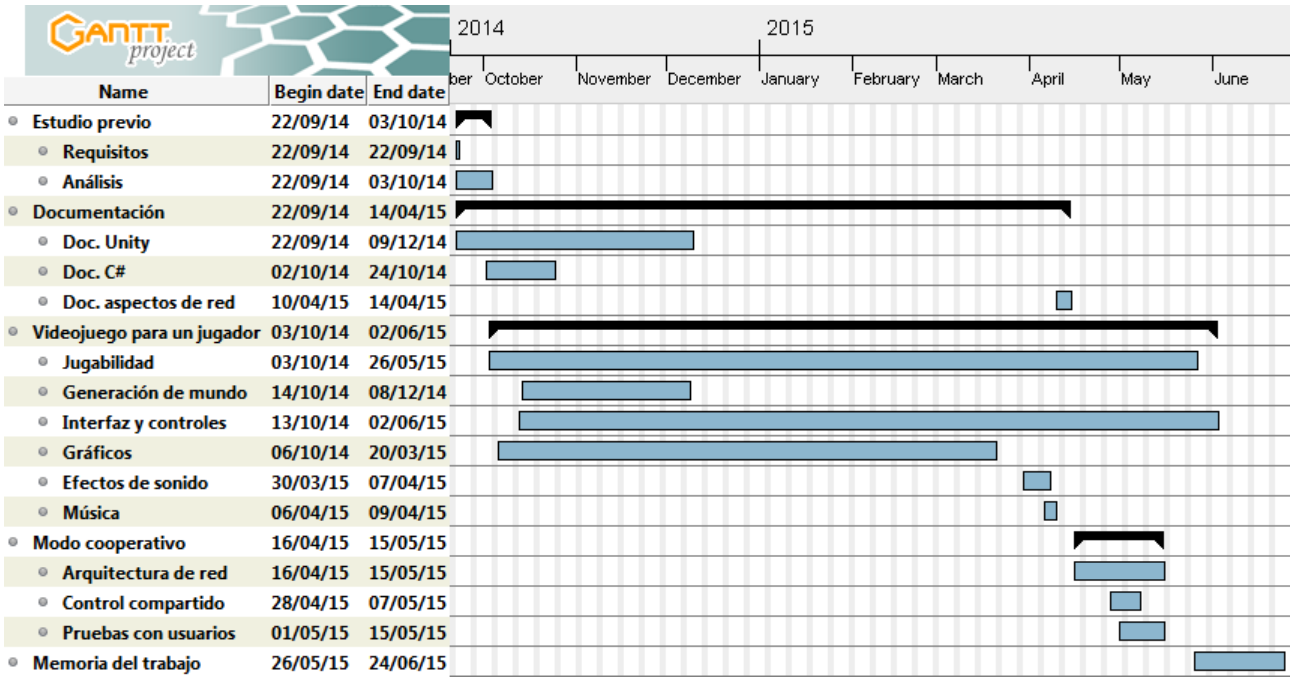

Figura 24 – Diagrama de Gantt del trabajo

Como se puede apreciar en el diagrama, algunas tareas, como la documentación sobre tecnologías o la creación de los recursos gráficos del videojuego, se han realizado cuando parecía adecuado o era necesario para avanzar en el resto de tareas. Por otro lado, tareas como la implementación del sistema de generación de mundo o de la arquitectura de red se han realizado en periodos más cortos de tiempo, pero ocupando la mayor parte del esfuerzo durante estos periodos.

B.2. Herramientas utilizadas

Se han empleado las siguientes herramientas durante la realización del trabajo. Todas ellas son gratuitas y algunas son software libre.

Unity Engine – Desarrollo del videojuego (motor gráfico)
Visual Studio C# Express – Desarrollo del videojuego (programación)
GIMP – Creación de recursos gráficos para el videojuego y de figuras para la memoria
Audacity – Edición de efectos de sonido
Modelio – Creación de diagramas usados en análisis y en la memoria

LibreOffice – Creación de la memoria y otros documentos
GanttProject – Creación del diagrama de Gantt para la memoria
Dropbox – Copia de seguridad y sincronización entre distintas máquinas de trabajo
Facebook – Grupo dedicado para la coordinación de las pruebas con usuarios