



UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Mención en Computación

**Generación de Escenarios de un Videojuego
2D mediante Programación Lógica**

Autor: Rafael Alcalde Azpiazu

Director: José Pedro Cabalar Fernández

A Coruña, 20 de septiembre de 2018

Especificación

Título del proyecto: Generación de Escenarios de un Videojuego
2D mediante Programación Lógica

Clase: Proyecto clásico de Ingeniería

Alumno: Rafael Alcalde Azpiazu

Director: José Pedro Cabalar Fernández

Miembros del tribunal:

Fecha de lectura:

Calificación:

DR. JOSÉ PEDRO CABALAR FERNÁNDEZ

Titular de universidad

Departamento de Computación

Universidade da Coruña

CERTIFICA

Que la memoria titulada **Generación de Escenarios de un Videojuego 2D mediante Programación Lógica** ha sido realizada por RAFAEL ALCALDE AZPIAZU, con DNI 47401974-D, bajo su dirección y constituye la documentación de su trabajo de Fin de Grado para optar a la titulación de Graduado en Ingeniería Informática por la Universidade da Coruña.

A Coruña, 20 de septiembre de 2018

*Cuando una afición en la infancia
se convierte en tu sueño de futuro...*

Agradecimientos

Primeramente me gustaría pedir disculpas, porque sé que se me dan mal estas cosas. A pesar de todo haré de tripas corazón para intentar recordar uno por uno aquellas personas que, ya sea por sus más y sus menos me han servido de pilar y me han ayudado para convertirme en lo que soy ahora y en llegar hasta aquí.

Empezaré por a dar las gracias a mi director, Pedro Cabalar, por el esfuerzo y la dedicación al guiarme en la construcción de este proyecto. Sobre todo agradecerle por las reuniones que tuvimos en los meses de verano, cuando todos solo estamos deseando tener vacaciones y descansar.

Quiero agradecer a mi familia, la cual me apoyó en todo momento, aparte de haber financiado mis caprichos y haberme dado una visión lo más abierta posible de la realidad que pueden dar unos padres. Entre todos creo que a la persona que tengo que agradecer más es a mi abuela materna, que es la persona con la que tengo muchísimo cariño y aprecio en esta vida.

También quiero agradecer a la comunidad de SceneBeta, que durante los años que cursé la ESO fue mi segundo hogar. Me ayudaron a pasar el tiempo pirateando la PSP para cargar, gracias a sus tutoriales, mis primeros pinitos en el mundo de la informática. Aún sigue en pie NekeOS, no me he olvidado de ello.

Luego no puedo olvidarme de agradecer a Mar, mi profesora de TIC de bachillerato, la cual me ayudó y guió a decidirme estudiar esta carrera.

Por otra parte me gustaría dar gracias a la Facultad de Informática de A Coruña, que durante estos último cuatro años de formación también ha sido mi segundo hogar, y si no que se lo pregunten a Lázaro.

Finalmente quiero agradecer a todas las amistades que me han acompañado, en mayor o menor medida, durante todos estos años, y con los que he vivido grandes experiencias, así que gracias a Eva, Iván, Dani, Santi, Diego, Borja, Fer, Jessie, Lara, y un largo etcétera de personas. Sois los mejores.

Rafael Alcalde Azpiazu
A Coruña, 20 de septiembre de 2018

Resumen

Este proyecto consiste en el desarrollo de una herramienta que, mediante el paradigma de la programación lógica, permita generar entornos jugables para un videojuego 2D táctico. Estos escenarios serán propuestos mediante un modelo declarativo creado en *Answer Set Programming*, el cual permite la representación del Conocimiento mediante lógica proposicional. Esto permitirá diseñar la construcción del entorno mediante una serie de reglas y restricciones, las cuales pueden ser modificadas fácilmente.

Palabras clave

ANSWER SET PROGRAMMING, PROGRAMACIÓN LÓGICA, REPRESENTACIÓN DEL CONOCIMIENTO, RESOLUCIÓN DE PROBLEMAS LÓGICOS, GENERACIÓN DE MAPAS POR ORDENADOR, FREECIV, LUA

Índice general

1. Introducción	1
1.1. Motivación	2
1.2. Objetivos	4
1.3. Estructura de la memoria	5
1.4. Plan de trabajo	6
2. Contexto	7
2.1. Juegos de estrategia por turnos	7
2.1.1. Sid Meier's: Civilization	8
2.1.2. Proyecto Freeciv	10
2.2. Tecnologías	11
2.2.1. Answer Set Programming	12
2.2.2. Lua	15
2.2.3. JSON	16
2.3. Herramientas	17
2.3.1. Atom	17
2.3.2. Git	18
2.3.3. LÖVE	19
3. Trabajo desarrollado	21
3.1. Propuesta	21
3.1.1. Formato del escenario de Freeciv	21
3.2. Proceso de ingeniería	26
3.2.1. Metodología de desarrollo	26

3.2.2. Gestión del proyecto	28
3.3. Análisis del software	31
3.3.1. Requisitos funcionales	31
3.3.2. Requisitos no funcionales	31
3.4. Diseño del sistema	32
3.4.1. Arquitectura software	32
3.4.2. Casos de uso	33
3.4.3. Pantallas del sistema	33
3.4.4. Diseño del programa lógico	38
3.4.5. Implementación	47
3.5. Ejemplos de uso	53
4. Evaluación	59
4.1. Tamaño de los mapas	60
4.2. Porcentajes de tierra y biomas	62
4.3. Prueba mixta tierra/biomas	65
5. Conclusiones	67
Apéndices	75
A. Bibliografía	77

Capítulo 1

Introducción

De un tiempo a esta parte, hemos visto como la industria de los videojuegos ha experimentado un cambio drástico, haciendo que actualmente sea uno de los principales motores económicos a nivel mundial, teniendo casos en los que un proyecto de este campo supera en nivel económico y de producción a muchas obras de la industria del cine. Esto se debe a todos los avances tecnológicos que nutren al mundo de los videojuegos, así como la madurez de un medio en el que muchos autores han visto su reconocimiento no por la diversión que plantean sus obras, si no por llevar al videojuego a su máxima expresión, realizándolo desde una forma más creativa o desarrollando el contenido y el alma del mismo de un modo que pueda llegar a un gran público. Uno de los puntos que hacen crecer esta industria es la Inteligencia Artificial, ya que el poder que aporta esta rama de la informática para crear sistemas eficientes o que se comporten de una manera inteligente permite llevar al videojuego a un nuevo nivel. Estos sistemas permiten crear desde interacciones humano-máquina más naturales al aplicarlas a las distintas entidades que pueden conformar el universo de un videojuego, como enemigos inteligentes que aprenden la forma de jugar del usuario o entidades aliadas que ajustan su comportamiento a la experiencia del jugador; como aplicar una variedad y diversidad al ecosistema al usar elementos de programación evolutiva a entidades [1] o generación procedimental [2] para generar escenarios. Incluso hemos visto este año como el aprendizaje profundo o *deep learning* ha permitido que empresas de *hardware* como Nvidia

han construido su nueva generación de tarjetas gráficas¹ ² para el mundo de los videojuegos con la premisa de un avance muy significativo en la calidad del renderizado de escenas generadas por computador, ya que permiten realizar una técnica muy costosa como puede ser el trazado de rayos o *raytracing* [3] en tiempo real [4].

Volviendo a la generación procedimental de entornos, muchos de estos sistemas se basan en una generación pseudo-aleatoria de puntos en donde se crean distintos patrones ya definidos por un ser humano. Esto se repite, incluso aplicando transformaciones de estos patrones hasta generar un mapa que parezca real en un alto grado. El problema de estos sistemas llega al momento de crear un entorno grande, resultando extremadamente lentos, por lo que otra aproximación muy usada es usar funciones matemáticas que definen el contorno del terreno, pudiendo incluso generarlo infinitamente en tiempo real a medida que el usuario avanza. Estas aproximaciones realizan un gran trabajo en cuanto a eficiencia y rapidez, mas resultan ineficientes a la hora de plantear requisitos y modificaciones concretas, ya que muchos de los elementos que definen estos sistemas están expuestos a la incertidumbre debido a su propia construcción. Este proyecto se centra en la aplicación de otro tipo de aproximación para la resolución de este problema, empezando por un caso concreto de generación de entornos para un sistema de entretenimiento.

1.1. Motivación

Como ya se ha comentado anteriormente, muchos de los campos de la Inteligencia Artificial se están aplicando cada vez más en la industria de los videojuegos, en especial para facilitar la tarea de crear sistemas y entornos que sean orgánicos y naturales para el consumidor de este tipo de *software*. Para ello se han aplicado muchas aproximaciones y optimizaciones de cara a tener

¹<https://www.nvidia.com/es-es/geforce/graphics-cards/rtx-2080-ti>

²<https://developer.nvidia.com/rtx>

sistemas lo más flexibles y con el objetivo de que respondan en un tiempo razonable consumiendo los mínimos recursos posibles. Este último requisito es imprescindible dado que un videojuego tiene que ser interactivo y con una tasa de respuesta lo más pequeña posible, así como poder ser ejecutado en sistemas con recursos muy escasos, como puede ser el caso de una consola portátil o un teléfono inteligente. A pesar de esto, muchos de los sistemas generadores de escenarios presentan problemas en el momento de ser modificados, ya que para influir en el resultado de la generación se necesita hacer una reprogramación completa del algoritmo generado. Esto hace que una modificación pequeña de los parámetros internos puede llegar a ocasionar que el resultado varíe enormemente, haciendo incluso que en ciertos videojuegos que requieren de una conexión a Internet para ser ejecutados, tengan que reiniciar el escenario, haciendo que sus usuarios pierdan el progreso explorado en el mapa. Ejemplos de esto los tenemos en muchos servidores del videojuego *Minecraft* o en las actualizaciones del videojuego *No Man's Sky*. Debido a esto, muchas empresas prescinden de realizar actualizaciones a las partes del código que controlan la generación del universo, lastrando consigo problemas que pueden ocasionar fallos por la mala gestión que pudo ocurrir a la hora de diseñar el sistema.

Como propuesta para esta problemática, en este proyecto se seguirá una aproximación distinta al resto de los sistemas. Para ello se partirá de un modelo declarativo formal con el que representar las reglas que describen el entorno. Para esta tarea se usará una herramienta basada en programación lógica, con la que se podrá definir un conjunto de reglas y restricciones con el que el sistema podrá completar y rellenar partes del entorno para generar un mapa final. Con la idea de restringir más este dominio, el sistema se centrará en la generación de un entorno en una rejilla 2D para el videojuego Freeciv, el cual al ser de formato libre, se puede encontrar fácilmente información de cómo funciona un mapa conforme a los requisitos del juego.

La gran ventaja que plantea el uso de las técnicas de programación lógica

es la independencia de la definición del modelo lógico con respecto al algoritmo de búsqueda o heurística que se use para hallar las posibles soluciones finales. Partiendo de esto, se usará el paradigma de *Answer Set Programming* [5], una variante de Programación Lógica que se ha convertido hoy en día en uno de los lenguajes de representación de conocimiento con mayor proyección y difusión debido tanto a su eficiencia en aplicación práctica para la resolución de problemas como a su flexibilidad y expresividad para la representación del conocimiento. La generación de escenarios de videojuegos supone un desafío como caso de prueba para *Answer Set Programming*, ya que el número de combinaciones posibles aumenta exponencialmente en función del tamaño del escenario. Cabe remarcar que ya existen antecedentes de generación declarativa para el diseño de espacios o entornos [6] usando este paradigma, así como el uso de *Answer Set Programming* en otros ámbitos donde también ocurre el mismo problema combinatorio, como puede ser la composición musical [7] [8].

1.2. Objetivos

Teniendo en cuenta lo explicado anteriormente, los objetivos finales de este proyecto son los siguientes:

- **Definición de un modelo declarativo para generación de escenarios:** Este proyecto se propone definir un modelo declarativo que dé respuesta a cómo construir un escenario del videojuego Freeciv usando las técnicas y paradigmas de *Answer Set Programming*, definiendo un conjunto de reglas, expresadas como restricciones en programación lógica, de modo que el usuario pueda variar sustancialmente la configuración de los escenarios obtenidos en función de la representación que se haga del problema.
- **Construcción de una pequeña herramienta gráfica:** Este pequeño editor gráfico permitirá al usuario crear y manipular manualmente el escenario, pudiendo fijar también algunas partes de la configuración del escenario y dejando que la herramienta complete las zonas no definidas.

- **Eficiencia:** Debido al problema que supone este tipo de aproximación, sobre todo al definir mapas de gran tamaño, la herramienta debe tener en cuenta la explosión combinatoria que puede ocasionar a la hora de obtener una solución. Así mismo, la herramienta ha de dar la respuesta en un tiempo tal que el usuario sienta razonable.

1.3. Estructura de la memoria

Para tener en cuenta la estructura que seguirá la memoria del presente proyecto, a continuación se explica los capítulos en los que se divide esta memoria:

- **Capítulo 1. Introducción:** Sirve como punto inicial a la lectura y conocimiento de este proyecto, describiendo la motivación que lo impulsa y detallando los objetivos a alcanzar.
- **Capítulo 2. Contexto:** Enmarca conceptos que son necesarios para entender el proyecto desarrollado, definiendo el plano tecnológico actual. Así mismo se describe y justifica las principales tecnologías empleadas en el desarrollo del mismo.
- **Capítulo 3. Trabajo desarrollado:** Explica las técnicas y el proceso de ingeniería llevado a cabo para la gestión, desarrollo y construcción del sistema propuesto para este proyecto. En este capítulo se indica el funcionamiento interno de la utilidad declarativa creada a la hora de plantear este proyecto.
- **Capítulo 4. Evaluación:** Realiza un análisis de las pruebas de rendimiento llevadas a cabo una vez acaba la construcción del proyecto, indicando la metodología y explicando los resultado obtenidos.
- **Capítulo 5. Conclusiones:** Ofrece una visión global de la viabilidad y calidad del sistema obtenido, así como se indican las vías de trabajo futura que se abre a la finalización del mismo.
- **Apéndices:** Adjunta las siguientes secciones complementarias:

- **Apéndice A. Bibliografía:** Recoge la documentación bibliográfica sobre la que se apoya este proyecto.

1.4. Plan de trabajo

Para el desarrollo del proyecto se han seguido las siguientes etapas:

- Estudio y análisis del estado del arte sobre la generación de escenarios en entornos similares, así como la investigación y estudio de la documentación del software elegido para el proyecto.
- Diseño e implementación del modelo declarativo para la generación de entornos en Freeciv.
- Diseño y construcción de una herramienta para la manipulación de entornos en FreeCiv.
- Diseño e implementación de un módulo que traduzca el resultado de ASP a Freeciv.
- Construcción de un componente capaz de evaluar el modelo lógico con distintos resultados.
- Evaluación de eficiencia para distintos casos de prueba.
- Redacción de la memoria del proyecto final. Esta fase se ha intentado realizar de forma paralela al resto de etapas.

Capítulo 2

Contexto

2.1. Juegos de estrategia por turnos

También conocidos como *turn-based strategy* (TBS), son juegos donde el tiempo de juego no transcurre de forma continua, si no que se divide en partes bien definidas llamadas turnos. En un turno, un jugador dispone de un periodo de análisis antes de realizar una acción, lo cual realiza un salto de turno al siguiente jugador. Cuando todos los jugadores han terminado su turno, se comienza una nueva ronda.

Existen varios géneros dentro de la estrategia por turnos:

- **Juegos clásicos de tablero:** Son los primeros juegos de este tipo, padres del resto de géneros. La idea es controlar la acción del propio juego mediante turnos. En este género tenemos como ejemplos el ajedrez, el Go o el Revesi.
- **TBS and RTT** (Estrategia por turnos y táctica en tiempo real): Añade un componente en tiempo real al permitir que en cada turno se generen batallas en tiempo real contra los oponentes. Un ejemplo de este género es la saga de videojuegos Total War.
- **Man-to-man:** Se basa en el uso de unidades muy pequeñas, en donde controlamos cada uno de los individuos de nuestra partida. En esta

categoría está la saga de videojuegos XCOM.

- **TRPG** (RPG táctico): También conocidos en Japón como Simulation RPG (SRPG), se basan en incorporar elementos de estrategia por turnos al combate clásico de los RPG. En la parte de tablero, un ejemplo clásico es Dragones y Mazmorras (*Dungeons & Dragons*), mientras que en el caso de los videojuegos, los más conocidos son Darkest Dungeon, algunas entregas de Megami Tensei como Shin Megami Tensei y Persona, la saga Disgaea o Fire Emblem.
- **4X**: Llamado así por la idea principal del género (*eXplore, eXpand, eX-ploit and eXterminate*, en castellano “Explora, expande, explota y extermina”). Se basan en el control de un imperio, profundizando en el desarrollo económico, tecnológico y militar, con la idea final de que crear a largo tiempo un imperio sostenible. Existe una profunda complejidad debido a la cantidad de elementos que hay que tener en cuenta. Un ejemplo en tablero de este género es Risk, mientras que en videojuegos destaca la saga Master of Orion y Sid Meier’s: Civilization [ver sección 2.1.1].

2.1.1. Sid Meier’s: Civilization

Como ya comentamos, Civilization¹ es un videojuego de estrategia por turnos que cae dentro del género X4, lanzado por primera vez en 1991 por el programador y diseñador Sid Meier. El objetivo del juego es construir una civilización y avanzarla desde la prehistoria y hasta un futuro cercano. En cada turno, el jugador puede mover unidades por el mapa, construir o mejorar ciudades y unidades, y realizar negociaciones con el resto de jugadores. Una vez se realiza una ronda, el juego avanza unos años en la historia. Actualmente existen seis títulos principales (a la hora de escribir esta memoria el último juego lanzado fue Civilization VI en 2016) y varios spin-off que se centran en potenciar distintos aspectos de los títulos principales.

¹<https://civilization.com>

Al empezar la partida, el jugador se encuentra en un mapa basado en casillas generado procedualmente, el cual contiene distintos biomas (praderas, desierto, montañas, ríos, lagos, etc...) con una serie de recursos (vino, caballos, trigo, fuel, carbón, etc...). La posición inicial normalmente es aleatoria, y el jugador puede controlar con una serie de colonos. En el resto del mapa cercano a los colonos, existe una niebla de guerra para evitar que el jugador pueda ver que hay cerca de el.

Si el jugador decide fundar una ciudad con los colonos, esta empieza a generar recursos que pueden ser los propios del mapa que hay en las casillas cercanas a la ciudad o simplemente mano de obra, cultura, ciencia o dinero. La cantidad de recursos que se puede generar en una ciudad se basa en como de grande es esta. Con estos recursos, se puede producir nuevas unidades como colonos, guerreros, distintos elementos de guerra como barcos, aviones y maquinaria pesada, o unidades con las que mantener comercio con otras civilizaciones.

Con la ciencia y la cultura se puede aumentar el progreso tecnológico, el cual se expresa en un árbol de tecnologías a desarrollar. El jugador puede ir eligiendo que tecnología se desarrolla en todo momento. Con respecto al dinero se puede usar para mejorar a las ciudades y unidades, crear carreteras, aumentar el ritmo de producción o crear regalos para otras civilizaciones.

Las ciudades se pueden mejorar al crear nuevos edificios, que ayudan a aumentar la producción y la población. Estos edificios pueden ser graneros, universidades, distritos comerciales o incluso librerías. También pueden servir de fortaleza, como murallas, castillos de guardia o escuelas militares. Por último, existen edificios únicos que son las maravillas del mundo, que ayudan a aumentar el ritmo general de una civilización y que dan bonus al primero que las complete, de ahí que solo se pueda crear una vez en toda la partida.

Existen múltiples victorias por las que puede ganar un jugador: la victoria por conquista ocurre cuando el jugador toma el control o añade al imperio a todas las ciudades capitales del resto de civilizaciones; la victoria diplomática se obtiene cuando el jugador construye las Naciones Unidas al entablar amistades con todas las civilizaciones; la victoria tecnológica se alcanza cuando el jugador construye una nave para llegar a Alfa Centauri; por último, la victoria cultural se obtiene al acumular la suficiente cultura con respecto a las otras civilizaciones y construir los edificios necesarios para guiar al resto a conseguir un estado utópico con alta cultura.

Debido al éxito de las entregas de Civilization y las disputas legales sobre el derecho de explotación entre Activision y Avalon Hills, se han creado distintos proyectos y secuelas que descienden del oficial, como la saga Call to Power o los proyectos Openciv y Freeciv. Este último se explicará en la Sección 2.1.2].

2.1.2. Proyecto Freeciv

Freeciv² ³ es un videojuego de código abierto basado en la serie de Sid Meier's Civilization, sobre todo en Civilization II. Fue creado por tres estudiantes del departamento de computación de la universidad de Aarhus debido al bajo rendimiento de Openciv. Basándose en la arquitectura de X11, escribieron un sencillo cliente en donde se ejecutaría una prueba de concepto, la cual tuvo una enorme acogida y rápidamente se creó una comunidad, la cual pasó a gestionar el proyecto debido a que sus autores desearon la idea de continuar. Debido a esto expandiendo el juego al incluir elementos online, mejoras en la interfaz gráfica, mejor rendimiento, traducción para 30 idiomas distintos (entre ellos el castellano) y nuevos elementos al juego.

El juego está escrito en C y tiene compatibilidad con el estándar POSIX para ser ejecutado en el mayor número de sistemas posibles. Contiene un

²<http://www.freeciv.org>

³<https://github.com/freeciv>

intérprete de Lua, el cual se explicará en la Sección 2.2.2, que permite cargar scripts con el que dar un componente dinámico a las partidas (lo usa sobre todo para tutoriales) y desde el 2006 IANA asignó el puerto TCP/UDP 5556 a Freeciv [9].

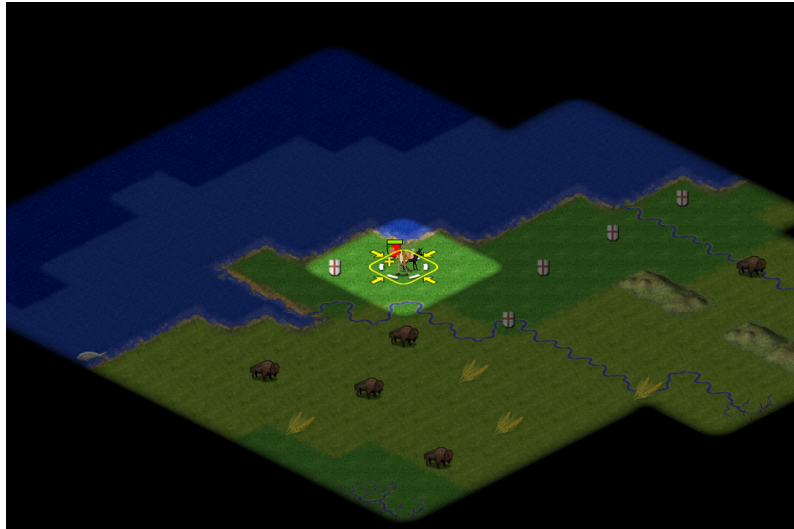


Figura 2.1: Ejemplo de escenario de Freeciv

Como el proyecto es de código abierto (tiene una licencia GPL [10]), cualquiera puede descargar el código fuente y modificarlo. Es por eso que se ha conseguido que el juego tenga diferentes tecnologías de interfaz gráfica, pasando por bibliotecas específicas como GTK⁴ y Qt⁵, bibliotecas gráficas generales como SDL⁶ o OpenGL⁷, e incluso bibliotecas del estándar web como WebGL⁸.

2.2. Tecnologías

Debido a las exigencias a la hora de desarrollar el proyecto, se ha optado por elegir un lenguaje de programación lógico sobre el que realizar la base declarativa del proyecto, ya que nos permitirá expresar las reglas de generación del mapa de forma matemática. También se ha escogido un segundo lenguaje

⁴<https://www.gtk.org>

⁵<https://www.qt.io>

⁶<https://www.libsdl.org>

⁷<https://www.opengl.org>

⁸<https://www.khronos.org/webgl>

multipropósito que nos servirá como soporte para crear un entorno gráfico con el que poder editar, guardar y cargar los mapas creados.

2.2.1. Answer Set Programming

Answer Set Programming (ASP) [5] es un paradigma enfocado a la resolución declarativa de problemas difíciles, combinando un lenguaje simple con el que modelar los problemas lógicos y herramientas de alto rendimiento para la resolución de estos. *Answer Set Programming* está basado en modelos estables [11], que usa para definir la semántica declarativa mediante programas lógicos normales. A esto se añade a que incorpora lógica no monótona [12], que añade razonamiento por defecto. Con esto, *Answer Set Programming* permite resolver problemas *NP-hard* de forma uniforme.

Una regla de programación lógica tiene el siguiente aspecto:

$$\underbrace{p}_{\text{Cabeza}} \leftarrow \underbrace{q_1, \dots, q_m, \text{not } q_{m+1}, \dots, \text{not } q_n}_{\text{Cola}}. \quad (2.1)$$

o, en formato texto:

$p \text{ :- } q_1, \dots, q_m, \text{not } q_{m+1}, \dots, \text{not } q_n.$

en donde p y todos los q_i son átomos, que son los elementos que pueden ser ciertos o falsos y los literales pueden ser tanto un átomo s y su negación ($\text{not } s$). Intuitivamente, una regla es una justificación que establece o deriva que p es verdad *si* todos sus átomos a la derecha de la flecha \leftarrow son ciertos. Por ejemplo, si suponemos la regla

$$\text{light_on} \leftarrow \text{power_on}, \text{not } \text{broken}. \quad (2.2)$$

informalmente significa que se puede afirmar que la luz está encendida si se puede establecer que hay electricidad y no hay razón de pensar que la lámpara. Puede existir reglas que no tengan cuerpo, como por ejemplo:

$$power_on \leftarrow . \quad (2.3)$$

Estas reglas se llaman hechos, ya que la cabeza es incondicionalmente cierta. Normalmente se omite la flecha \leftarrow . Los programas lógicos son una colección finita de reglas, en donde se expresa la “justificación” de un conjunto de átomos que pueden ser establecidos. Es importante señalar que *not* no es el operador estándar de negación (\neg), si no que significa que algo “no es derivable”. Esto es lo que se llama negación por defecto [13]. Pensando en las dos reglas presentadas en 2.2 y 2.3, *power_on* se puede derivar ya que es un hecho (2.2), mientras que *broken* no porque no hay ninguna regla que lo derive. Esto nos permite derivar *light_on* (2.3), el cual sería el modelo estable.

Existe otro tipo de reglas, las cuales no tiene cabeza, es decir,

$$\leftarrow B. \quad (2.4)$$

Estas reglas se llaman *constrains* o restricciones, y sirven para indicar que satisfacer *B* es una contradicción. Así mismo es común que programas lógicos contengan el par de reglas

$$a \leftarrow B, not \bar{a}. \quad (2.5)$$

$$\bar{a} \leftarrow B, not a. \quad (2.6)$$

en donde ni *a* ni \bar{a} aparecen en la cabeza de cualquier otra regla del programa, y *B* es una conjunción de literales. Esta regla aparece cuando se quiere referir tanto a un átomo *a* como a su negación (estándar). Para representar este último, se introduce el nuevo átomo \bar{a} y se incluye las dos reglas. Intuitivamente el rol de las reglas es seleccionar en caso de que *B* se satisfaga. Estas reglas se pueden escribir como una regla *choise* de la forma

$$\{a\} \leftarrow B. \quad (2.7)$$

En resumen, los problemas lógicos se reducen al cómputo de los modelos estables de un programa lógico dado, lo cual se hace mediante herramientas llamadas *solvers*, que se encargan de utilizar diferentes técnicas para obtener un modelo estable final.

Para ello, se necesita que los programas lógicos estén expresados con variables libres. Como la forma de expresar un programa lógico en ASP es mediante un lenguaje de alto nivel con ciudadanos de primer orden, muchas de las variables están ligadas. Es por eso que, antes de resolver el programa, se usa unas herramientas llamadas *grounders*, que permiten transformar el programa a su equivalente con variables libres. En la Figura 2.2.1 se muestra el ciclo de ejecución de un programa ASP.

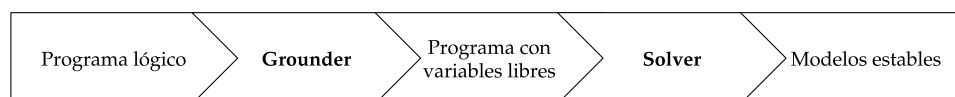


Figura 2.2: Ejemplo de ejecución de ASP

Unas de las herramientas más importantes de ASP son la de Potassco⁹, la cual son un conjunto de herramientas para ASP desarrolladas en la Universidad de Postdam. Contienen las herramientas fundamentales como un grounder llamado *gringo*; un solver que es *clasp*; y una herramienta que aglutina todo el sistema ASP, *clingo*. Así mismo, añade más funcionalidades al lenguaje ASP, como puede ser la resolución iterativa, debido a que permite embeber otros lenguajes como Lua, el cual está explicado en la Sección 2.2.2, o Python, que pueden interactuar con el programa escrito en ASP mediante la interfaz o *API built-in* de *clingo*.

Estas herramientas serán claves a la hora de realizar este proyecto, ya que, como veremos más adelante, debido a la naturaleza de ASP, serán la base

⁹<http://potassco.org>

sobre la que se construirá todo el proyecto.

2.2.2. Lua

Lua¹⁰ [14] (del portugués Luna), es un lenguaje de programación de propósito general, de scripting y multiparadigma (procedural, orientado a objetos basado en prototipos, funcional y *data-driven*). Fue pensado para ser embebido en cualquier aplicación independientemente de la plataforma sobre la que se ejecuta. Su intérprete esté escrito en ANSI C y contiene una API en C muy simple.

Uno de los puntos fuertes de Lua es que permite construir nuevos tipos en base a arrays asociativos, que también permiten extender la semántica del lenguaje al aplicar metadatos a estos. A parte de esto, también facilita la tarea al programador al tener un recolector incremental de basura que se encarga de gestionar automáticamente la memoria. Además, Lua es de tipado dinámico, y se ejecuta sobre un intérprete bytecode en una máquina virtual basada en registros, ejecutándose más rápido que otros lenguajes ejecutados sobre una máquina virtual basada en *stack*, como puede ser Java.

Aparte de esto, existen otras implementaciones y dialectos basados en Lua, como LuaJIT¹¹, que es una implementación de la máquina virtual de Lua que compilar en tiempo de ejecución y no antes de ejecutarse; y MoonScript¹², un lenguaje de scripting basado en Python que extiende el lenguaje de Lua al añadir orientación a objetos basados en clases, una forma más eficiente de lenguaje funcional y nuevas estructuras y elementos que no contenía Lua.

Existen otras implementaciones que lo único en lo que se diferencian de la implementación estándar es que expanden las funciones básicas o implementan

¹⁰<http://www.lua.org>

¹¹<http://lua.jit.org/lua.jit.html>

¹²<https://moonscript.org>

bibliotecas por defecto. A estas implementaciones se llaman *Lua Players* o *Lua Engines*, que principalmente son destinadas para la creación de videojuegos, ya que añaden la implementación de las bibliotecas gráficas para una plataforma concreta, como pueden ser SDL o OpenGL. Los más destacados son los intérpretes no oficiales para consolas de Sony y Nintendo, y los intérpretes para ordenador y móviles como Corona, Cocos-2D, Moai y LÖVE [ver sección 2.3.3].

Como ya explicamos en la Sección 2.2.1, las herramientas ASP que se usarán permiten usar este como un lenguaje embebido, lo que nos permitirá controlar la ejecución de la base. Así mismo, tomaremos el *engine* LÖVE para crear una interfaz gráfica para que los usuarios puedan trabajar mejor con la edición de los mapas.

2.2.3. JSON

JSON¹³ (acrónimo de JavaScript Object Notation) es un formato abierto [15] ligero de intercambio de información. Está pensado para la transmisión de objetos de datos usando texto en un formato legible por humanos, pudiendo ser leído y modificado fácilmente.

A demás, puedes ser fácilmente analizado y generado por máquinas, de ahí que a pesar de que en sus inicios fue pensado como un subconjunto del lenguaje de programación JavaScript/ECMAScript, actualmente es un lenguaje independiente debido a que muchos otros lenguajes incluyen actualmente herramientas para poder leerlo y modificarlo de forma nativa.

JSON se basa en dos tipos de estructuras:

- Un mapa asociativo que guarda pares (nombre, valor). Esta estructura se reconoce en el estándar como *object*.

¹³<https://www.json.org>

- Una lista ordenada que guarda valores. Esta estructura se reconoce en el estándar como *array*.

Estas estructuras se pueden mezclar independientemente, pudiendo generar listas de objetos u objetos que contienen listas de elementos.

Debido estas características, y a que Lua soporta este estándar mediante bibliotecas externas, se usará para el intercambio de datos entre el programa principal y la parte lógica, así como será utilizado para almacenar temporalmente los escenarios antes de ser exportados al software final.

2.3. Herramientas

Aparte de las tecnologías usadas para la construcción del proyecto, se ha usado distintas herramientas que nos han servido a la hora de elaborar y desarrollar el sistema planteado.

2.3.1. Atom

Atom¹⁴ es un editor de texto gratuito y de código abierto basado en Electron, que es un *Framework* para crear aplicaciones nativas con tecnología Web. Está desarrollado por GitHub Inc. e integra dentro de su interfaz la herramienta Git para el control de versiones. Se puede extender las funcionalidades de Atom mediante *plugins* escritos con Node.js, los cuales permiten desde resaltar la sintaxis de un determinado lenguaje de programación, cambiar el tema del editor, hasta añadir terminales, autocompletado de texto o nuevas herramientas para desarrollar/depurar código fuente. Entre estos plugins cabe destacar que para este proyecto podemos encontrar algunos que nos permiten resaltar la sintaxis de Lua y de ASP, así como iniciar LÖVE y ASP desde el propio editor. Debido a estas características, se usará este editor como una de las herramientas principales.

¹⁴<https://atom.io>

2.3.2. Git

Git¹⁵ es un sistema de control de versiones distribuido y de código abierto creado para la gestión de código fuente y desarrollo de software. Fue creado por Linus Trovalds como una alternativa libre y gratuita a BitKeeper para el desarrollo del kernel Linux por la comunidad. Está enfocado en la rapidez y eficiencia al procesar proyectos grandes, la integridad de datos, la seguridad mediante autenticación criptográfica y el fuerte soporte de un flujo de trabajo no lineal y distribuido.

Debido a estas características, Git es uno de los sistemas de control de versiones más importantes hoy en día, por lo que ha permitido que existan servicios de alojamiento en línea que incluyen esta herramienta:

- GitLab¹⁶, creado por dos programadores ucranianos y que hoy en día es gestionado por GitLab Inc. Es usado por organizaciones como Sony, IBM, NASA, CERN o GNOME Foundation. Permite realizar gratuitamente repositorios privados, gestionar grupos y realizar rastreo de *issues* y funcionalidades de CI/CD.
- Phabricator¹⁷, creado por Facebook como herramienta interna que integra también Mercurial y Subversion. Actualmente es de código abierto y lo usan empresas como Blender, Cisco Systems, Dropbox o KDE.
- Bitbucket¹⁸, creado por Atlassian. Este sistema integra Mercurial desde sus inicios y Git desde 2011, y está pensado para ser integrado con el resto de productos como Atlassian como Jira, Confluence y Bamboo.
- GitHub¹⁹, creado por tres estudiantes estadounidenses, hoy en día es gestionado por GitHub Inc. empresa que fue adquirida por Microsoft en

¹⁵<https://git-scm.com>

¹⁶<https://gitlab.com>

¹⁷<https://www.phacility.com>

¹⁸<https://bitbucket.org>

¹⁹<https://www.github.com>

2018. Permite realizar rastreo de *bugs*, wikis, gestión de tareas y petición de funcionalidades. Es usado por empresas como Microsoft, Google, Travis CI, Bitnami, DigitalOcean y Unreal Engine. Con la popularidad de GitHub nacieron varios servicios, como el Education Program, que permite a estudiantes el acceso gratuito a herramientas de GitHub y de partners; Gist, que permite usar GitHub como un hosting de *snippets* y GitHub Marketplace, que permite comprar servicios con los que aumentar las funcionalidades en los proyectos y que muchos de ellos son gestionados por partners.

2.3.3. LÖVE

LÖVE²⁰ (o también conocido como Love2D) es un motor de código libre multiplataforma para la creación de juegos en 2D mediante el lenguaje Lua. Está hecho en C++ y añade una API al lenguaje Lua para que el programador pueda usar las bibliotecas OpenGL ES y SDL de una forma muy simplificada mediante funciones y tipos creados en Lua. Con el paso del tiempo también fue incluyendo un motor de físicas 2D o bibliotecas para el manejo de fuentes FreeType, manejo de strings en UTF-8, funciones para usar sockets y una biblioteca especializada en conexión a videojuegos llamada ENet. Así mismo, debido al crecimiento de la comunidad, esta fue creando bibliotecas no oficiales que expanden el funcionamiento del motor, como bibliotecas para añadir interfaces inmediatas, soporte para técnicas usadas en videojuegos, orientación a objetos basada en clases o incluso portar el motor a otras plataformas o lenguajes.

Debido a la sencillez para poder dibujar en pantalla, así como poder integrar las bibliotecas gráficas creadas para LÖVE, este motor va a ser una de las herramientas principales que se usarán para la construcción de la interfaz de usuario en este proyecto.

²⁰<https://love2d.org>

Capítulo 3

Trabajo desarrollado

En este capítulo se detalla cada uno de los puntos llevados a cabo para la realización de este proyecto, empezando por definir la propuesta realizada, luego explicar el proceso ingenieril llevado a cabo y terminar desglosando el trazado de la ejecución de este proyecto.

3.1. Propuesta

El trabajo propuesto tiene como objetivo la creación de un elemento software funcional que, usando el paradigma lógico explicado en la Sección 2.2.1, permita la generación de un escenario jugable que pueda ser ejecutado en el juego Freeciv, como se indica en la Sección 2.1.2. Así mismo incluirá una interfaz gráfica interactivable que permitan al usuario marcar que zonas del terreno deben generarse y cuales no, indicando su contenido antes de lanzar el proceso de generación.

3.1.1. Formato del escenario de Freeciv

Como uno de los puntos fuerte de este trabajo tiene que ver con la generación de escenarios, este sistema tiene que tener como salida un mapa válido que sea leído correctamente por el juego Freeciv. Es por eso que explicaré en detalle el formato usado.

3. Trabajo desarrollado

Para empezar, el formato se basa en archivo de texto plano que contiene varios campos, haciendo que sea lo más simple posible y que en la teoría se pueda modificar a mano.

```
[scenario]
is_scenario=TRUE
name=_("My map")
description=_("This map is a example.")
players=TRUE

[savefile]
options=" +version2"
version=20
reason="Scenario"
rulesetdir="classic"
[...]

[settings]
set={"name", "value", "gamestart"
      "generator", "SCENARIO", "RANDOM"
      "mapsize", "FULLSIZE", "FULLSIZE"
      "maxplayers", "::PLAYERS::", "::PLAYERS::"
      "topology", "", "WRAPX|ISO"
      "xsize", 50, 12
      "ysize", 50, 12
      [...]
}
set_count=9

[map]
have_huts=FALSE
t0000="h+hf  aaaa  ff"
t0001="dgg  aat  ff"
t0002="hhh  pp"
t0003="hmp  ggghh"
t0004="hmh  s dghf"
t0005=" dg  phffpm "
t0006="  "
```

```

t0007="aa      sdg      aa"
t0008="aa      h      aa"
t0009="      jff p      "
t0010="mh      s pdp pm"
t0011="gf      pdhh      "
t0012="      gpsg      p"
t0013="      h  hh      "
t0014="hff      gg  s"
t0015="m+h    aaaa    m f"
startpos_count=5
startpos={"x","y","exclude","nations"
          0,2,FALSE,"Russian"
          9,11,FALSE,"Spanish"
          14,0,FALSE,""
          2,5,FALSE,""
          12,3,FALSE,""
        }
b00_0000="0000000000000000"
[... ]
r00_0000="0000000000000000"
[... ]

```

Listado 3.1: Ejemplo de formato de mapa.

Como se puede comprobar en el Listado 3.1, el archivo que contiene un mapa se divide en varias secciones de datos, los cuales se marcan con un título entre corchetes. De los definidos por Freeciv, se puede destacar algunos.

- **scenario**: Configura los ajustes básicos del mapa creado, como puede ser el nombre del mismo (con el campo **name**) o una breve descripción (con el campo **description**).
- **savefile**: Establece los valores por defecto de las opciones de juego, como puede ser la versión de Freeciv mínima (con el campo **options**), el conjunto de reglas de juego (con el campo **rulesetdir**) o las tecnologías disponibles (se establece en la lista guardada por **technology_vector** y se indica el número de elementos en **technology_size**).

- **settings**: Se puede definir una lista de valores del mapa y la topología del mismo (que se explica en detalle en la Sección 3.1.1) en el campo **set**. El número de valores definidos se guarda en el campo **set_count**.
- **map**: En esta sección se indica cada uno de los valores de terreno de la rejilla del mapa (en los campos **t00_XXXX**), así como la lista de puntos de inicio de jugadores (definidos en la lista **startpos** e indicado el tamaño de la lista en **startpos_count**), las capas con recursos o incluso las capas de ríos.

Topología del mapa

El mapa es siempre una rejilla de dos dimensiones en el que cada celda es una baldosa o *tile*. Esta rejilla puede estar configurada de varias maneras con la variable **topology** en la sección **settings**.

- **warpx**: La topología de escenario es como un mapa terrestre, es decir el eje Este-Oeste se junta.
- **warpy**: La topología del escenario junta el eje Norte-Sur.
- **warpx warpy**: La topología del escenario es un toroide, es decir, tiene forma de donuts.
- **iso**: La rejilla del escenario es isométrico.
- **hex**: La rejilla del escenario es hexagonal.
- **iso hex**: La rejilla del escenario es en forma de panel de abeja.

Terrenos disponibles por defecto

En Freeciv, cada celda de la rejilla del mapa contiene una baldosa de terreno único, que viene definido por un identificador único, el cual puede ser uno de los siguientes:

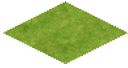


Descripción	ID	Imagen
<i>Pradera</i> : Es uno de los terrenos más comunes. Las unidades se pueden mover fácilmente.	g	
<i>Llanura</i> : Es otro terreno muy común. Se puede usar para crear carreteras.	p	
<i>Colinas</i> : Las unidades se mueven lentamente. Es uno de los terrenos con mayor bonus defensivo (200 %).	h	
<i>Bosque</i> : Produce una unidad de producción (madera) con la que construir edificaciones. Tiene un bonus defensivo de 150 %	f	
<i>Jungla</i> : Puede llegar a producir 4 unidades de producción si se encuentran con recursos de gemas o fruta.	j	
<i>Montañas</i> : Es el terreno con mayor bonus defensivo (300 %). Solo las unidades aéreas (aviones, cazas, etc) pueden atravesarla.	m	
<i>Desierto</i> : Normalmente solo se puede usar para crear carreteras, pero si hay un modificador de oasis puede generar hasta 3 unidades de producción.	d	
<i>Pantano</i> : Se puede irrigar rápidamente. Puede producir de 5 a 9 unidades de producción si se encuentra con recursos como tundra o con especias.	s	
<i>Tundra</i> : Solo se pueden crear carreteras.	t	
<i>Glacier</i> : Ninguna de las unidades puede cruzarlo.	a	

Tabla 3.1: Tipos de terreno.

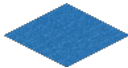
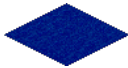
Descripción	ID	Imagen
<i>Mar</i> : Todas las unidades acuáticas pueden cruzarlo. Puede producir 2 unidades de producción si se encuentra con un banco de peces		
<i>Océano</i> : Solo las grandes embarcaciones y submarinos pueden pasar por encima.	:	

Tabla 3.2: Tipos de terreno.

3.2. Proceso de ingeniería

En este capítulo se explica el proceso que se ha seguido para realizar el proyecto descrito. Se empezará explicando la metodología de desarrollo escogida y a continuación se expondrá y se desgranará la gestión de este trabajo.

3.2.1. Metodología de desarrollo

A pesar de que en un primer se ha pensado que para la planificación de este proyecto se usaría la metodología de desarrollo Scrum [16] [17], finalmente se ha optado por usar una amalgama entre distintas metodologías de desarrollo en espiral. Esto es debido a que Scrum tiene unas características básicas que no se han tenido en cuenta a la hora de diseñar y desarrollar el proyecto, por lo que no sería correcto decir que se ha utilizado este tipo de metodología ágil:

- **Flujo de trabajo**: El trabajo se divide en varias iteraciones entre una y cuatro semanas llamadas *Sprint*, en donde en cada una de ellas se obtiene un incremento funcional del producto. Al comienzo de cada iteración se realiza una reunión en donde se identifican las tareas a realizar en esa iteración, realizando una estimación de todas; durante la realización de la iteración se realiza cada día una pequeña reunión en donde se comunica el estado actual; y, al finalizar una iteración se realiza una reunión que sirve como retrospectiva y alimentación para la próxima iteración.

A la hora de realizar este proyecto, a pesar de que se ha dividido en varias iteraciones de entre 1 semana y media, realizando una reunión al principio de estos, y en cada iteración se finaliza con un producto funcional, el resto de elementos de la metodología Scrum no se han llevado a cabo en la práctica.

- **Gestión de riesgos:** Uno de los puntos de desarrollo con una metodología ágil es la de mantener una gestión temprana de riesgos para evitar una gran desviación de la planificación. Esto se realiza mediante un panel que presenta listado ordenado de las tareas a realizar en el desarrollo total del proyecto, llamado *Product backlog*. Estas tareas son las que estiman su duración y se incluyen en otro listado de tareas a realizar en cada iteración, llamado *Sprint backlog*. Con esto se puede obtener una gráfica de evolución del proyecto llamado *Burn down chart*, pudiendo detectar desviaciones y sobreasignaciones de forma visual debido a que también se marca el trabajo ideal de proyecto.

Para este proyecto no se ha usado ninguno de estos elementos porque no se ha realizado un seguimiento exhaustivo del mismo a causa de la escasa experiencia de alumno a la hora de realizar una planificación y estimación del proyecto, sobre todo a la hora de seguir y diseñar un proyecto con una metodología de desarrollo ágil.

Debido a todos estos factores expuestos, se podría indicar que la metodología usada finalmente está más cerca de una basada en modelos de prototipos en donde se tiene en cuenta que cada prototipo resultante de una iteración es un modelo funcional del mismo, y en donde es revisado en la reunión de la siguiente iteración sin estar presente el cliente final, si no que solo con el director del proyecto. También en esta reunión se definen las tareas a realizar para el desarrollo del proyecto de cara a la nueva iteración.

3.2.2. Gestión del proyecto

A continuación se describirá la planificación llevada a cabo para cumplir con el tiempo estimado de realización del proyecto, así como el coste y los recursos necesarios para la construcción del mismo.

Planificación

Volviendo a lo comentado en la Sección 3.2.1, con respecto la planificación del desarrollo del trabajo en cuestión, se ha dividido en varias iteraciones que han ido incrementando las funcionalidades del producto final. Estas iteraciones vienen expresados de una forma gráfica en la Figura 3.1 y explicados en detalle en la Tabla 3.3.

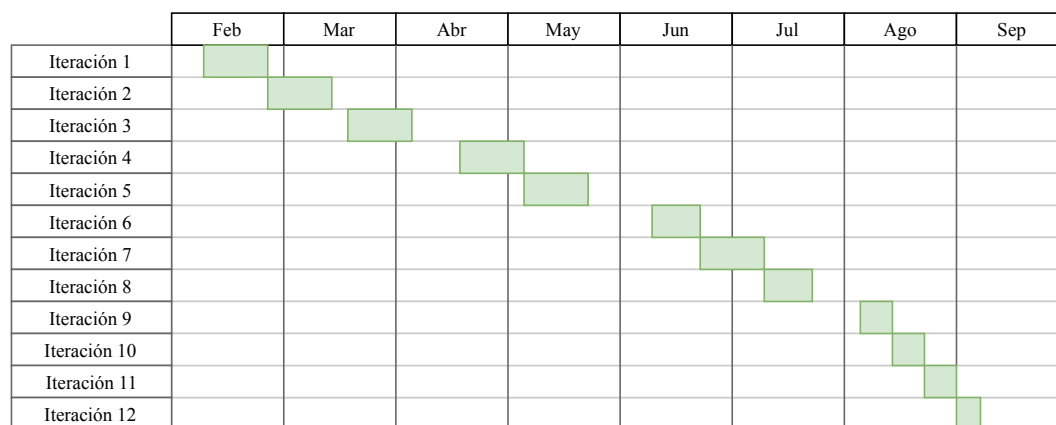


Figura 3.1: Diagrama de Gantt con las iteraciones del proyecto.

Cabe destacar que el proyecto en cuestión ha tenido dos aproximaciones diferentes. La primera aproximación tiene que ver con las iteraciones 1-3, en donde el sistema generaba todos los tipos de terrenos del mapa. El problema de esta aproximación es que ocasionaba que la generación de mapa tardase mucho más de lo que alguien puede considerar razonable. Esto es debido a la naturaleza de ASP, que tal como se describe en la Sección 2.2.1, provoca que el conjunto de reglas a tener en cuenta sea exponencial, dando como resultado una explosión combinatoria de hechos y que la herramienta no sea usable.

Iteración	Tiempo (h)
1	30
2	30
3	30
4	20
5	20
6	32
7	32
8	32
9	32
10	32
11	32
12	40
Total	362

Tabla 3.3: Desglose de las iteraciones.

Como segunda aproximación, la cual comprende el resto de iteraciones, se ha decidido dividir el programa en distintos módulos independientes que se encargan de generar distintas partes del mapa. Esto hace que el tiempo sea mucho más razonable y sea más manejable. Aún así, el hecho de cambiar de aproximación, como puede entenderse, hizo que se tuviera que cambiar la parte declarativa de la herramienta.

Coste y recursos

Para calcular el coste de los recursos humanos que ha supuesto este proyecto, además de las 362 horas totales del proyecto, también se han llevado a cabo reuniones de una hora a la hora de iniciar cada iteración, así como una al inicial el proyecto y otra al terminar el mismo, dando un total de 14 horas a mayores al proyecto. Teniendo esto en cuenta y siguiendo la Guía Salarial del Sector TI en Galicia [18], se ha estimado los costes que se pueden ver en la Tabla 3.4.

Recurso	Tiempo (h)	Coste (€/h)	Total
Doctor Investigador	14	24,00	336,00
Analista Programador	376	9,00	3384,00
Total	376		3720,00

Tabla 3.4: Coste de los recursos humanos del proyecto.

Recurso	Vida (mes)	Uso (mes)	Coste (€)	Total (€)
Portátil	48	7	800,00	116,67
Freeciv	∞	7	-	0
Clingo	∞	7	-	0
LÖVE	∞	7	-	0
HUMP	∞	7	-	0
SUIT	∞	7	-	0
json.lua	∞	7	-	0
busted	∞	7	-	0
luaproc	∞	7	-	0
LDoc	∞	7	-	0
Github	∞	7	-	0
Travis-CI	∞	7	-	0
Coveralls	∞	7	-	0
LaTeX	∞	7	-	0
Total				116,67

Tabla 3.5: Coste de los recursos humanos del proyecto.

Por otro lado, en la 3.5 se recoge la lista de recursos materiales y lógicos necesarios para el desarrollo de este proyecto. Para el caso de los recursos *hardware* se ha tenido en cuenta el tiempo de uso para este proyecto con respecto a su vida útil, mientras para los recursos *software*, se ha preferido por usar herramientas y bibliotecas de uso libre y gratuitas, de ahí que el coste no esté contado para el resultado final.

Con todo esto se puede concluir que el coste total del proyecto asciende hasta 3836,67 € en total.

3.3. Análisis del software

Una vez definido el sistema y planificada su construcción, se ha realizado un análisis en donde se identifican los requisitos que debe cumplir el software una vez terminado el proyecto.

3.3.1. Requisitos funcionales

Los requisitos funcionales son aquellas condiciones indispensables que estipulan las funcionalidades que debe proporcionar el sistema. Para este proyecto se ha recogido los diferentes requisitos:

- Generación de un mapa que sea legible por el videojuego Freeciv.
- Permitir añadir restricciones sobre ciertas zonas del mapa.
- Poder guardar y recuperar el mapa en un formato sencillo.

3.3.2. Requisitos no funcionales

Los requisitos no funcionales, por su contra, son aquellas condiciones indispensables que debe cumplir el sistema a la hora de diseñar e implementar. Para este proyecto se han tenido en cuenta estos requisitos:

- Eficiencia y eficacia: El generador debe responder en el menor tiempo posible arrojando una respuesta óptima.
- Escalabilidad: El generador debe trabajar con mapas de diferentes tamaños, por lo que el sistema debe poder soportar cualquier tamaño de entrada.
- Usabilidad: La interfaz gráfica debe ser lo más sencilla posible, evitando que el usuario tenga que realizar tareas tediosas a la hora de construir mapas.

3.4. Diseño del sistema

Una vez definidos los requisitos se ha procedido a realizar el diseño software del sistema en cuestión, empezando a concretar la arquitectura propuesta y luego desarrollando los casos de uso y diagramas de clases.

3.4.1. Arquitectura software

Debido a que el sistema cuenta con una interfaz gráfica y un módulo que se encargará de generar el mapa, el sistema estará dividido en dos partes concretas tal y como se puede ver en la Figura 3.2:

- Una parte que será la aplicación gráfica, que actuará en todo momento como *Front-end* de cara al usuario. Esta parte sigue la estructura Modelo-Vista-Controlador (MVC), en donde el controlador se encargará de hacer de puente entre la interfaz gráfica, que es lo que manipulará el usuario, y los datos guardados en memoria. Así mismo proporcionará las funcionalidades básicas del sistema como guardar o cargar el mapa y crear un mapa en blanco.
- La segunda parte actuará como *Back-end*, que se encargará de generar el mapa mediante un programa lógico escrito en ASP que se ejecutará Clingo. Contiene un controlador que se encargará de hacer la llamada a Clingo y de obtener su resultado, y luego otro controlador que se encargará de llamar primeramente al programa lógico que genere las regiones y luego, dada una región, rellene las casillas de la región. Para ello primeramente define que zonas son tierra y que zonas son agua, y con las zonas con tierra se rellenan con cordilleras y con áreas bióticas (las cuales son zonas grandes que contienen un solo tipo de terreno). Una vez realizado terminará generando los mares y los puntos de inicio para los jugadores.

Tanto el controlador de la interfaz de usuario como el controlador del programa ASP se ejecutarán en paralelo, pudiendo enviarse información de un controlador a otro para saber cuando hay que empezar una generación o si

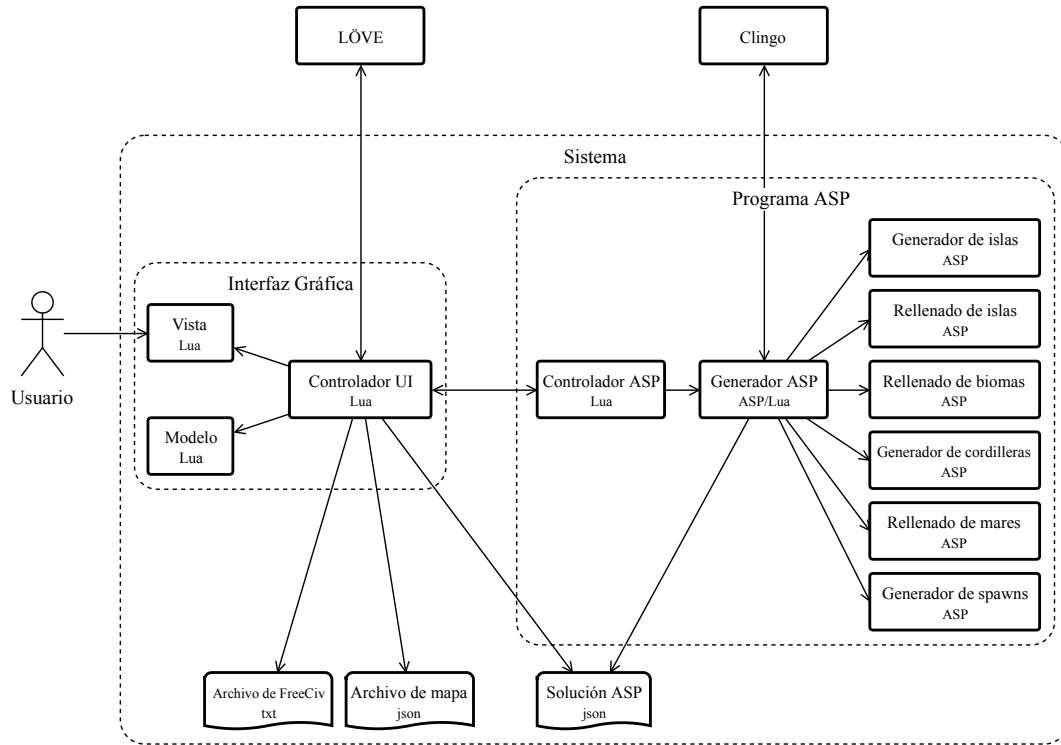


Figura 3.2: Arquitectura del sistema.

esta terminó. A pesar de esto, los resultados de la generación de ASP se guardarán en un archivo intermedio para evitar enviar gran cantidad de datos entre los elementos.

3.4.2. Casos de uso

El sistema tiene en cuenta que se usará en todo momento por un único usuario, el cual llevará a cabo todas las funcionalidades propuestas en la Sección 3.3.1 a través de una interfaz gráfica que se explica en la Sección 3.4.3. Estos requisitos funcionales se transforman, por tanto, en los casos de uso del sistema que se proponen en la Figura 3.3.

3.4.3. Pantallas del sistema

El proyecto propuesto está pensado para ser usado a través de una interfaz gráfica de usuario, la cual es modelada mediante un patrón MVC como se indica en la Sección 3.4.1. Esta interfaz está dividida en varias vistas, las

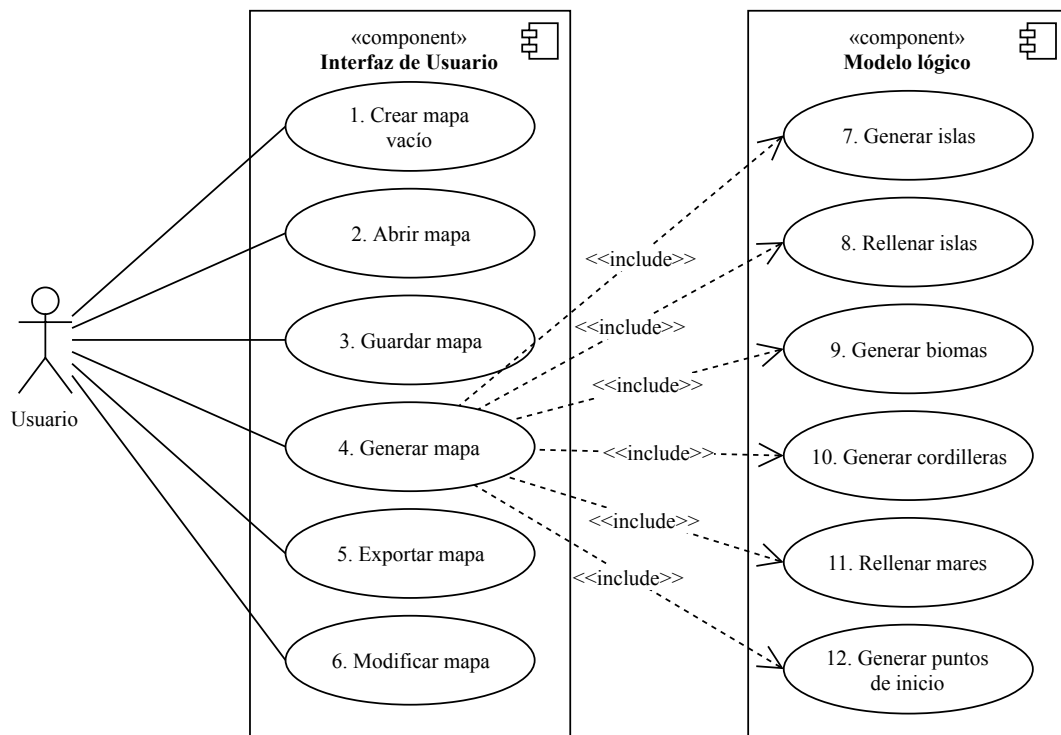


Figura 3.3: Casos de uso del sistema propuesto.

cuales corresponden con los casos de uso propuestos contra los que el usuario interacciona directamente en la Sección 3.4.2.

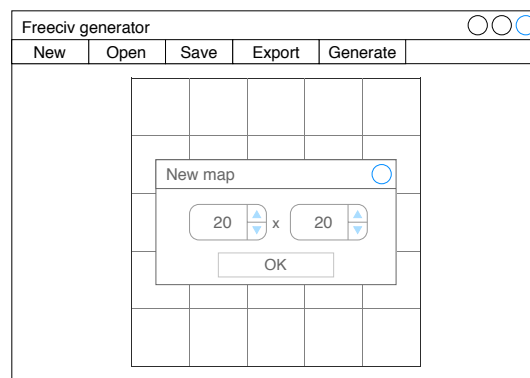


Figura 3.4: Pantalla de nuevo mapa.

- 1. Crear mapa vacío: Se acciona cuando el usuario presiona en el botón de “Crear mapa” en la barra de herramientas, lo que despliega una ventana emergente parecida a la Figura 3.4. Aquí el usuario puede indicar el tamaño del mapa a crear y pulsar el botón de “Aceptar”. Con esto el sistema muestra una rejilla vacía.

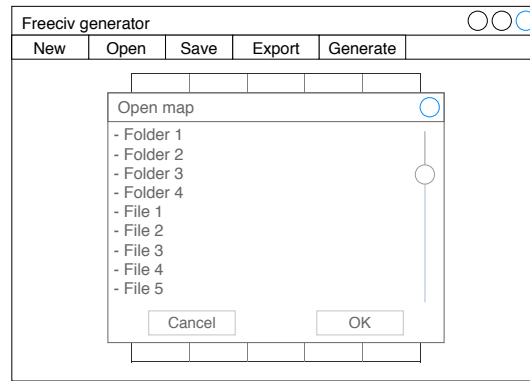


Figura 3.5: Pantalla de abrir archivo de mapa.

- 2. Abrir mapa guardado en disco: Se inicia cuando el usuario presiona en el botón de “Abrir mapa” en la barra de herramientas, lo que despliega una ventana emergente parecida a la Figura 3.5. Aquí el usuario navega a través de una lista que representa los archivos guardados en disco y selecciona un archivo anteriormente guardado por la aplicación que representa un mapa. El sistema procede a cargarlo y mostrar los datos del mapa en la rejilla principal.

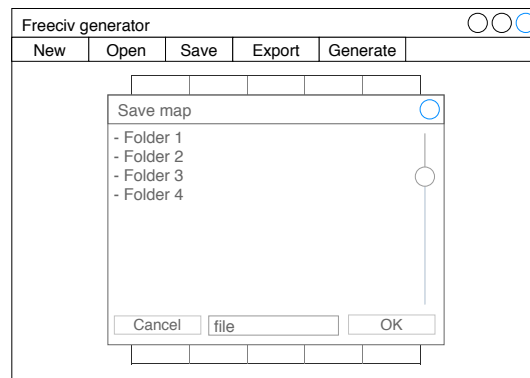


Figura 3.6: Pantalla de guardar mapa.

- 3. Guardar mapa en disco: Se inicia cuando el usuario presiona en el botón de “Guardar mapa” en la barra de herramientas, mostrando una ventana emergente como la de la Figura 3.6. Aquí el usuario navega por las carpetas guardadas en disco a través de una lista y selecciona una donde se guardará. Además, escribe el nombre del nuevo archivo en un campo de texto debajo de esta lista y acepta. El sistema procede

a transformar los datos del mapa en un fichero que se guarda en disco donde el usuario indic .

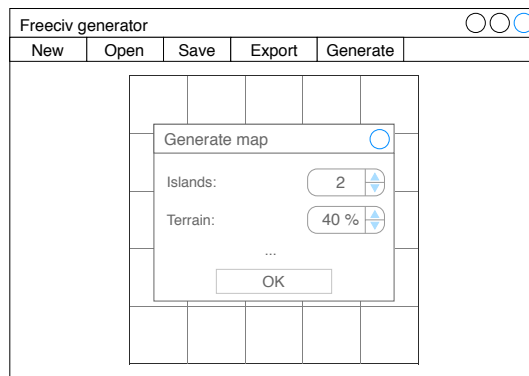


Figura 3.7: Pantalla de generar mapa.

- 4. Generar mapa: Se inicia cuando el usuario acciona el bot n de “Generar mapa” en la barra de herramientas, haciendo que el sistema muestre una ventana emergente parecida a la Figura 3.7. Aqu  el usuario indica los par metros que prefiera en la generaci n y acepta. En sistema procede a llamar a la parte del programa l gico creado en ASP, la cual va ejecutando cada uno de los casos de uso. Entre medias, el sistema lanza una vista como la de la Figura 3.8 para proporcionar retro-alimentaci n al usuario. Una vez acabada la generaci n, el sistema muestra en la rejilla principal con el mapa que se ha generado finalmente.

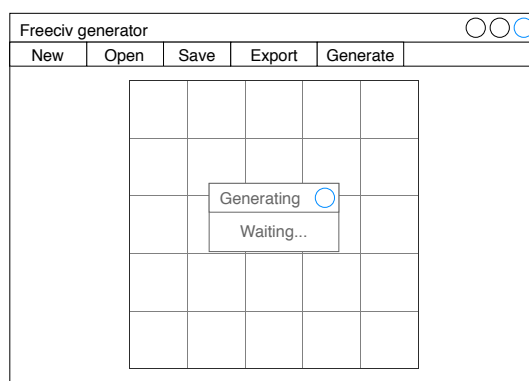


Figura 3.8: Pantalla con el mensaje de espera.

- 5. Exportar mapa al formato de Freeciv: Se inicia cuando el usuario presiona en el bot n de “Exportar mapa” en la barra de herramientas,

mostrando una ventana emergente como la de la Figura 3.9. Aquí el usuario navega por las carpetas guardadas en disco a través de una lista y selecciona una donde se guardará. Además, escribe el nombre del nuevo archivo en un campo de texto debajo de esta lista y acepta. El sistema procede a transformar los datos del mapa en un fichero reconocible por Freeciv y se guarda en disco donde el usuario indicó.

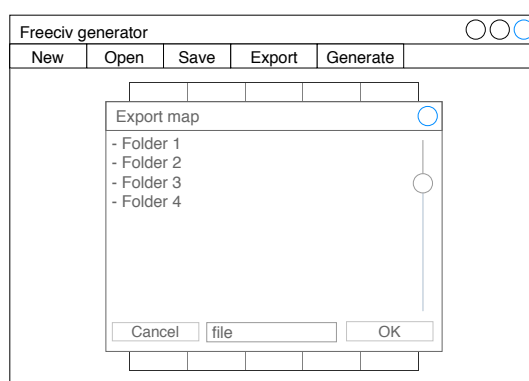


Figura 3.9: Pantalla de exportar mapa.

- 6. Modificar mapa: Se inicia cuando el usuario presiona sobre una de las celdas de la rejilla expuesta en la Figura 3.10. El sistema pasará a cambiar el terreno de la celda por otro, realizando un ciclo en el momento en el que no quede ninguna opción nueva.

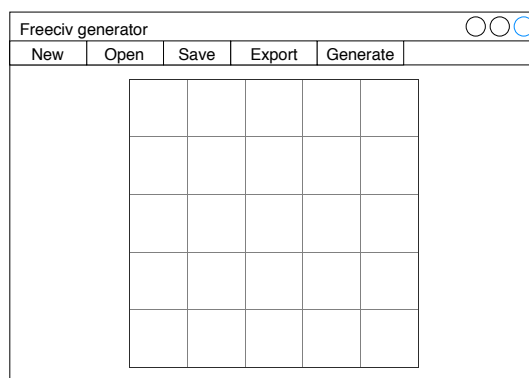


Figura 3.10: Pantalla principal de la aplicación.

3.4.4. Diseño del programa lógico

Retornando a lo comentado en la Sección 3.4.1, una de las piezas fundamentales de este proyecto es la definición de un generador de mapas en formato declarativo. Este módulo está pensado mediante el paradigma de programación lógica que usa la tecnología ASP, tal y como se expone en la Sección 2.2.1.

El objetivo de este generador es intentar dar un modelo declarativo basado en reglas que corresponda en mayor o menor medida con la definición de un mapa de Freeciv. Es por esto que, como se ha contado en la Sección 3.2.2, para evitar los problemas que puede suponer este proyecto en cuanto a evitar tener una explosión combinatoria de soluciones válidas, se ha decidido finalmente realizar una aproximación dividiendo este módulo en varios programas independientes que generan partes concretas del escenario.

Generación de regiones

Como primer paso para la generación del mapa, se ha pensado en usar un programa lógico que, dada una división grande del mapa (la cual llamaremos cuadrante), identifica para cada una de ellas a que región o isla pertenece, tal y como se puede ver en la Figura 3.11.

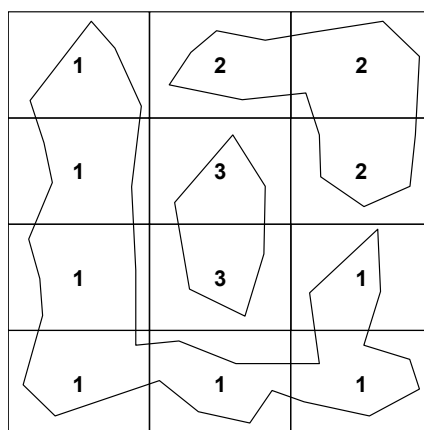


Figura 3.11: Ejemplo de mapa dividido en cuadrantes

Para ello se elige un cuadrante único para isla, el cual va a ser la raíz desde

donde se vaya expandiendo toda la región. Esto se ha implementado mediante siguientes reglas, en donde primeramente se selecciona una raíz ($\text{q_root}(\text{C}, \text{I})$) para la región $\text{region}(\text{I})$ dado un cuadrante $\text{q_pos}(\text{C})$ mediante una *choice* rule. A esta *choice* rule se le indica la cardinalidad 1-1, que se expresa como $\text{min } \{p\} \text{ max}$ e indica que solo puede haber un átomo $\text{q_root}(\text{C}, \text{I})$ por cada región I . Luego se añade una regla de restricción para que dos raíces de distintos cuadrantes tengan la misma posición. Para ello ASP tiene el operador != que significa distinto:

```
1 { q_root(C, I) : q_pos(C) } 1 :- region(I).
:- q_root(C, I), q_root(C, J), I!=J.
```

Para que cada raíz se expanda por el mapa, se ha planteado que las celdas contiguas sean alcanzables ($\text{q_reached}(\text{C}, \text{I})$), es decir, que si está adyacente (se tiene en cuenta 4-adyacente mediante el átomo $\text{q_adj}(\text{C}, \text{D})$) a la raíz o a otra celda conectada a la raíz. Esta última puede o no ser alcanzada siempre que no se haya escogido para otra región, la cual viene definido en el átomo $\text{existanother}(\text{I}, \text{C})$:

```
q_reached(C, I) :- q_root(C, I).
{q_reached(C, I)} :- q_reached(D, I), region(I),
    not existanother(I, C), q_adj(D, C).
existanother(I, C) :- q_reached(C, J), region(J), region(I),
    J!=I.
```

Finalmente, para que todo el mapa esté completo y no haya cuadrantes sin definir a una región, se ha añadido una regla de restricción que evita las soluciones en donde el número de elementos $\text{q_reached}(\text{C}, \text{I})$ no es igual al máximo número de cuadrantes (dados por el átomo $\text{n_quadrants}(\text{Z})$). Para ello se usa una función agregada o *aggregate* [19], que son muy similares a las correspondientes funciones de SQL en bases de datos. En el caso de $\text{\#count}(\text{P}$

: $q(P)$), cuenta el número de átomos P en $q(P)$ que son ciertos:

$n_quadrants(Z) :- \#count\{C, I: q_reached(C, I)\} = Z.$
 $:- n_quadrants(Z), max_quadrants(MAX), Z \neq MAX.$

Definición de regiones

Como segundo paso que se realiza, se procede a detallar cada una de las regiones por separado, indicando qué celdas del mapa que corresponde a esa región son de agua y cuales son de tierra. Para ello se realiza una aproximación parecida explicada en la generación de regiones, en donde se toma una celda que cae dentro de la región como celda raíz $rootcell(P)$ que contendrá tierra. Esto se realiza mediante una regla *choice* con cardinalidad 1-1:

1 $\{rootcell(P) : c_pos(P)\} \quad 1.$

Con esto la celda raíz de la isla intenta alcanzar al resto de celdas adyacentes usando una definición similar a la explicada en el apartado anterior, para ello se usa la relación $reached(Q)$:

$reached(P) :- rootcell(P).$
 $\{reached(P)\} :- reached(Q), adj(Q, P).$

Aún así, en este caso no queremos que la tierra rellene completamente la isla, por lo que el número de las celdas que se pueden alcanzar viene condicionado por la variable $LAND$ del átomo $dims(ROWS, COLS, LAND)$, que indica la cantidad de tierra que se tiene que alcanzar en tanto por cien. Por defecto está establecida para un 20 % del número de celdas totales de la región, valor que viene establecido en el átomo n_cells :

$cells_reached(Z) :- \#count\{P: reached(P)\}=Z.$


```
:- cells_reached(Z), n_cells(MAX), dims(ROWS, COLS, LAND),  
   Z!=MAX*LAND/100.
```

Cabe destacar que en este apartado también se han añadido restricciones evitar la generación de formas o zonas que no sean del todo creíbles. Estas restricciones son las siguientes y pueden comentarse en cualquier momento sin peligrar el resto del programa:

```
%Forbidden create lakes with 1 cell  
:- c_pos(P), not_reached(P), top(P, T), T!=w, left(P,L), L!=w,  
   bottom(P,B), B!=w, right(P,R), R!=w.  
  
%Forbidden create bridges  
:- reached(P), top(P, w), left(P, L), L!=w, bottom(P, w),  
   right(P, R), R!=w.  
:- reached(P), top(P, T), T!=w, left(P, w), bottom(P, B), B!=w,  
   right(P,w).
```

Finalmente hay que destacar que, como la generación de regiones es independiente, para evitar que las islas de tierra se solapen, el programa principal genera unas restricciones que corresponden con las celdas adyacentes a las que están en borde de una región y han sido seleccionadas como tierra. Estas restricciones se tienen en cuenta en la siguiente generación, evitando que esto suceda.

Rellenado del terreno

Una vez se ha terminado la ejecución del apartado anterior, se procede a la selección de los biomas, es decir, las zonas de tierra con un tipo de terreno en concreto, que contendrá esa región. Para ello, de un modo similar al que se viene explicando, se elige una celda que será la raíz del bioma y se extiende mediante las celdas adyacentes. Aún así hay que destacar que, por defecto, el bioma de una celda de tierra es de hierba, por lo tanto aquí caben dos

afirmaciones que se han supuesto en esta generación:

- La hierba es el único bioma que no se genera, si no que todas las celdas contienen hierba a siempre y cuando no se exprese lo contrario.
- Puede existir islas que no tengan todos los tipos de biomas, por lo que puede o no generarse un tipo concreto de bioma.

Con esta información se han llevado a cabo las siguiente reglas, en donde $\text{root}(P, T)$ es la celda inicial del bioma y $\text{reached}(P, T)$ son las celdas que se alcanzan para el bioma T :

```
% The root of a bioma
{root(P, T) : cell(P, 1)} :- biomas(T).
:- root(P, T), root(P, U), T!=U.

% Defines which cells could be reached
reached(P, T) :- root(P, T).
{reached(P, T)} :- biomas(T), cell(P, 1), adj(P, Q),
    reached(Q, T).
:- reached(P, T), reached(P, U), T!=U.
```

Por otra parte, pueden existir ciertos biomas que solo se generen en una latitudes concretas, como los casquetes polares. Para ello se pueden crear ciertas reglas especiales que generen estos terrenos en función de la fila del mapa en la que se encuentra. En el siguiente ejemplo se ha recogido las reglas que generarían los casquetes en la zona norte y en la zona sur del mapa, siendo de ancho un 10 % del alto del mapa:

```
ice(ROWS*10/100) :- dims(ROWS, COLS, TERRAIN).

land(p(X, Y), glacier) :- cell(p(X, Y), 1), ice(N), X < N-1.
land(p(X, Y), glacier) :- cell(p(X, Y), 1), ice(N),
    dims(ROWS, COLS, TERRAIN), X > ROWS-N-1.
```

Para terminar con este apartado, se ha añadido varias restricciones de tamaño. Las primeras reglas tienen que ver con la cantidad de tipos de terrenos generados, la cual sigue con el esquema propuesto en el apartado anterior (por defecto está establecida para un 20 % del número de celdas totales de la región):

```
n_cells(Z) :- #count{C, I : reached(C, I)} = Z.
:- n_cells(Z), max_cells(MAX), dims(ROWS, COLS, TERRAIN),
   Z!=MAX*TERRAIN/100.
```

Finalmente se ha decidido que un bioma tiene que ocupar al menos dos casillas. Debido a eso, para los biomas generados se cuenta el número de celdas alcanzadas y se añade una regla de restricción que evita que la suma total de celdas alcanzadas sea menor a dos, la cual viene dada en el átomo `size_bioma(T,N)`:

```
size_bioma(T, N) :- root(Q, T), cell(Q, 1),
   #count{P: reached(P, T)} = N.
:- size_bioma(T, N), N < 2.
```

Definición de cordilleras

Para la definición de cordilleras (el tipo de terreno montaña), esta se realiza una vez se ha completado la definición de los biomas de una isla, evitando así que el módulo anterior tenga una explosión combinatoria fuerte de resultados. Así mismo, esto nos permite tener en cuenta que, para dar un mayor realismo a la generación, en la vida real puede existir un sistema montañoso que divida ecosistemas con el mismo tipo de bioma debido al acercamiento de dos placas tectónicas.

Después de esta breve explicación, para la definición de las reglas de generación de cordilleras se ha propuesto que esta generación tenga dos puntos, un

punto inicial **start(P)** y un punto final **start(Q)** que comprenda el inicio y el final de la cordillera:

```
% Generate the start and the end position
1 {start(P) : cell(P, 1)} 1.
1 {end(P) : cell(P, 1)} 1 :- start(Q).
% Forbidden that start and end points are the same.
:- start(P), end(Q), P=Q.
```

Con esto lo que se pretende es hacer un camino que vaya de un punto a otro, para ello, de la misma forma que se ha generado las islas, se parte del punto inicial y se van alcanzando las celdas de tierra adyacentes mediante el átomo **reached(P)**:

```
% The start cell is reached always.
reached(P) :- start(P).
% A cell next to a reached cell could be reached
{reached(P)} :- cell(P, 1), reached(Q), adj(P, Q).
```

Lo siguiente es evitar todas las soluciones en donde no se haya alcanzado el punto final, para ello se define el átomo **complete** en función de esto y se prohíbe las soluciones que no tienen este átomo mediante una regla de restricción:

```
% The path is complete if the end is reached.
complete :- end(P), reached(P).
% Forbidden the paths that don't reach the end point.
:- not complete.
```

Con esto generará un camino de cordilleras con cualquier ancho. Aún así se ha preferido acotar este camino a un ancho máximo. Para ello se han añadido unas reglas para obtener el número de vecinos adyacentes, en donde se parte

de un borde del camino para el que el átomo $\text{size}(P,D,N)$, $N = 0$ y se avanza en un sentido D (arriba, abajo, izquierda, derecha). Por ejemplo, partiendo de un borde inferior, sus vecinos adyacentes en el sentido up serían:

```
size(p(X, Y), up, 0) :- reached(p(X, Y)), not reached(p(X-1, Y)).
size(p(X, Y), up, N+1) :- reached(p(X, Y)), size(p(X-1, Y), up, N)
.
```

Esto se realiza para todos los sentidos correspondientes a la definición de 4-adyacencia. Con esto se puede obtener el ancho del camino en cualquier punto sumando los elementos en una dirección concreta mediante los átomos $\text{height}(P,N)$ y $\text{width}(P,N)$, los cuales se obtienen con las reglas:

```
height(P, N+M+1) :- size(P, up, N), size(P, down, M).
width(P, N+M+1) :- size(P, left, N), size(P, right, M).
```

Dadas estas reglas, se puede construir restricciones para delimitar el tamaño máximo de la cordillera. Para ello el tamaño viene impuesto en la variable **SIZE** del átomo $\text{dims}(\text{ROWS}, \text{COLS}, \text{SIZE}, \text{LENGHT})$:

```
:- height(P, N), dims(ROWS, COLS, SIZE, LENGHT), N>SIZE.
:- width(P, N), dims(ROWS, COLS, SIZE, LENGHT), N>SIZE.
```

Rellenado de agua

Con respecto a este módulo, lo que hace es añadir una plataforma continental cerca de la costa y el resto dejarlo como mar profundo. Para ello, en vez de usar un 4-adyacente se usa un 8-adyacente para que la plataforma bordeé todas las islas. Las reglas que forman los átomos de agua $\text{water}(P,C)$ son las siguientes:

```
%Draw sea if there are near land
water(P, sea) :- cell(P, w), adj(P, Q), cell(Q, 1).
```

```
%Draw ocean if there are near sea or near ocean
water(P, ocean) :- cell(P, w), adj(P, Q), water(Q, sea),
    not around_land(P).
water(P, ocean) :- cell(P, w), adj(P, Q), water(Q, ocean),
    not around_land(P).
```

Generación de puntos de inicio

Para generar los puntos iniciales desde donde partirán los jugadores se tiene en cuenta lo cerca que están de las montañas y lo cerca que están del agua para intentar crear asentamientos lejos de cordilleras y cercanas al mar.

Para ello se realiza primero la generación de un punto de inicio en una posición aleatoria mediante la siguiente regla *choice*:

```
N { player(P) : cell(P, 1) } N :- players(N).
```

Luego se procede a obtener la distancia desde un punto de inicio a la celda más próxima con agua. Para ello primero se ha planteado una regla que obtiene el átomo *greater_dist_water(J,N)*, que dado un jugador *player(J)*, todas las distancias en donde existe dos átomos *cell(P1,w)* y *cell(P2,w)*, en donde uno tiene una distancia desde el jugador más pequeño a este. Con esto se plantea otra regla que define el átomo *water_distance(J,D)*, que dado un jugador *player(J)*, obtiene la distancia más cercana al agua mediante el átomo *cell(P,w)* que no tiene una distancia más pequeña:

```
greater_dist_water(J, N) :- player(J), cell(P1, w), cell(P2, w),
    manhattan(J, P1, N), manhattan(J, P2, M), N > M.
water_distance(J, D) :- player(J), cell(P, w),
    manhattan(J, P, D), not greater_dist_water(J, D).
```

Esto mismo se realiza con la distancia a la celda de montaña mas próxi-

ma. Con esto, se ha utilizado un par de funciones que corresponden con una extensión de ASP, que está basada en preferencias. Estas preferencias se pueden expresar mediante reglas $\#maximize(P: q(P))$ y $\#minimize(P: q(P))$, que están soportadas por Clingo, y permiten obtener el modelo que optimice las preferencias dadas en P , descartando el resto de modelos. Para este caso se quiere optimizar la menor distancia al agua por un jugador y la mayor distancia de un jugador a la montaña:

```
#minimize{D : player(J), water_distance(J,D)}.  
#maximize{D : player(J), mountain_distance(J,D)}.
```

Finalmente, teniendo esto en cuenta, para interconectar todos estos programas se ha creado un controlador que usa la API *built-in* de clingo¹, la cual permite usar Lua dentro de un programa lógico. Este controlador sigue el flujo de información que se describe en la Figura 3.12, en donde podemos ver como se va realizando las llamadas a los diferentes módulos explicados anteriormente.

3.4.5. Implementación

Una vez analizado el proyecto en cuestión, pasaré a detallar el diseño de cada uno de los diferentes componentes del sistema en su construcción, indicando mediante diagramas como se integran los diferentes elementos.

Como ya se indicó en la Sección 2.2.2, el lenguaje de programación Lua no tiene un paradigma de programación orientado a objetos basados en clases, más se ha preferido que en la realización de este sistema se use el módulo *class.lua* de la biblioteca *HUMP*² para una organización lo más estructurada posible. Así mismo, también hay que destacar que el lenguaje tampoco soporta la manipulación de archivos en formato JSON, por lo que se ha usado el

¹<https://potassco.org/clingo/python-api/current/clingo.html>

²<https://github.com/vrld/hump>

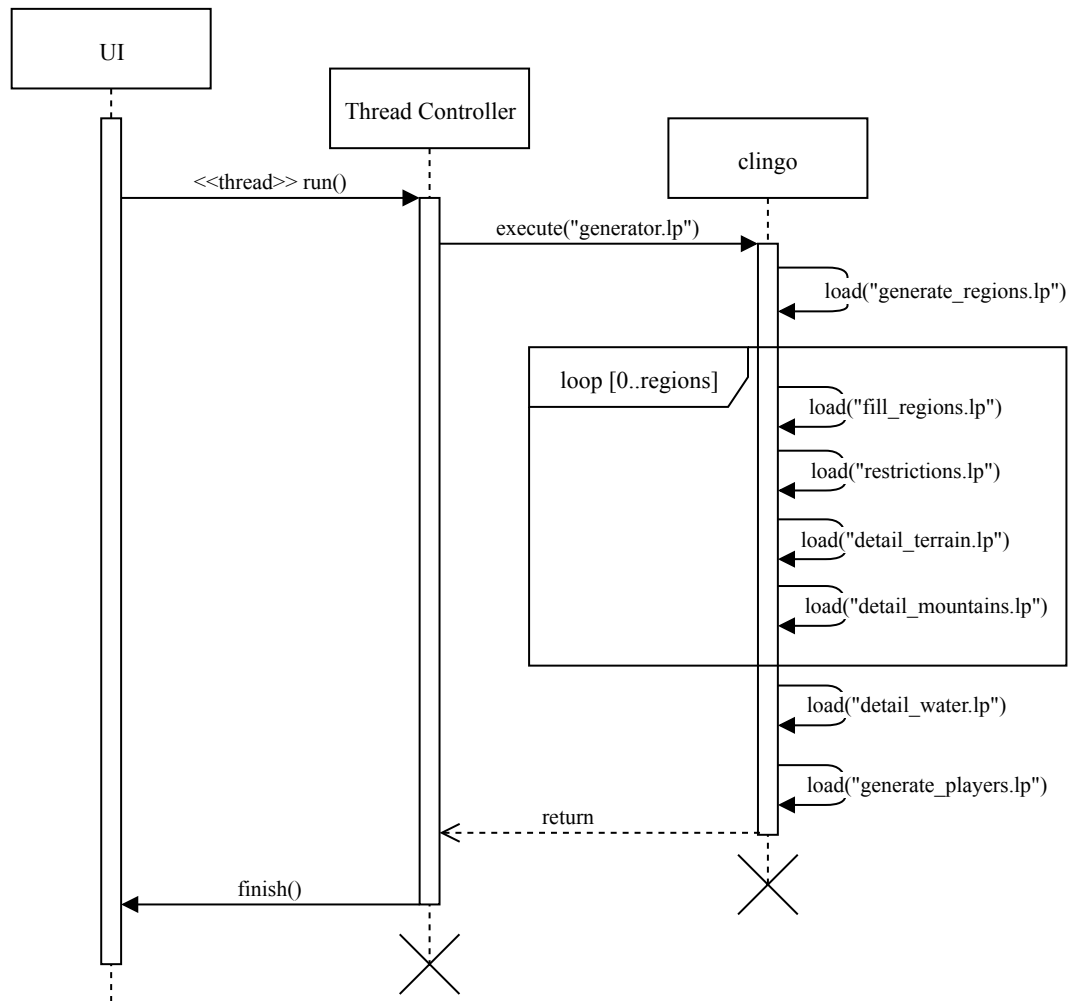


Figura 3.12: Diagrama de secuencia de la ejecución del generador.

módulo *json.lua*³, el cual permite transformar un texto en formato JSON a tipos compatibles en Lua.

Por último, indicar que para la realización de las vistas de la interfaz de usuario se ha usado la biblioteca *SUIT*⁴, que permite generar una interfaz gráfica en modo inmediato de forma sencilla sobre LÖVE, pudiendo realizar un prototipo de los elementos más rápidamente y con más flexibilidad que con otro tipo de bibliotecas gráficas, como puede ser GTK+⁵. Por otra parte, habrá elementos gráficos más complejos (como pueden ser diálogos de selección de

³<https://github.com/rxi/json.lua>

⁴<https://github.com/vrld/SUIT>

⁵<https://www.gtk.org/>

archivos o ventanas emergentes) que se tendrán que generar a mano, tal y como se expone en la Sección 3.4.5.

Clase principal

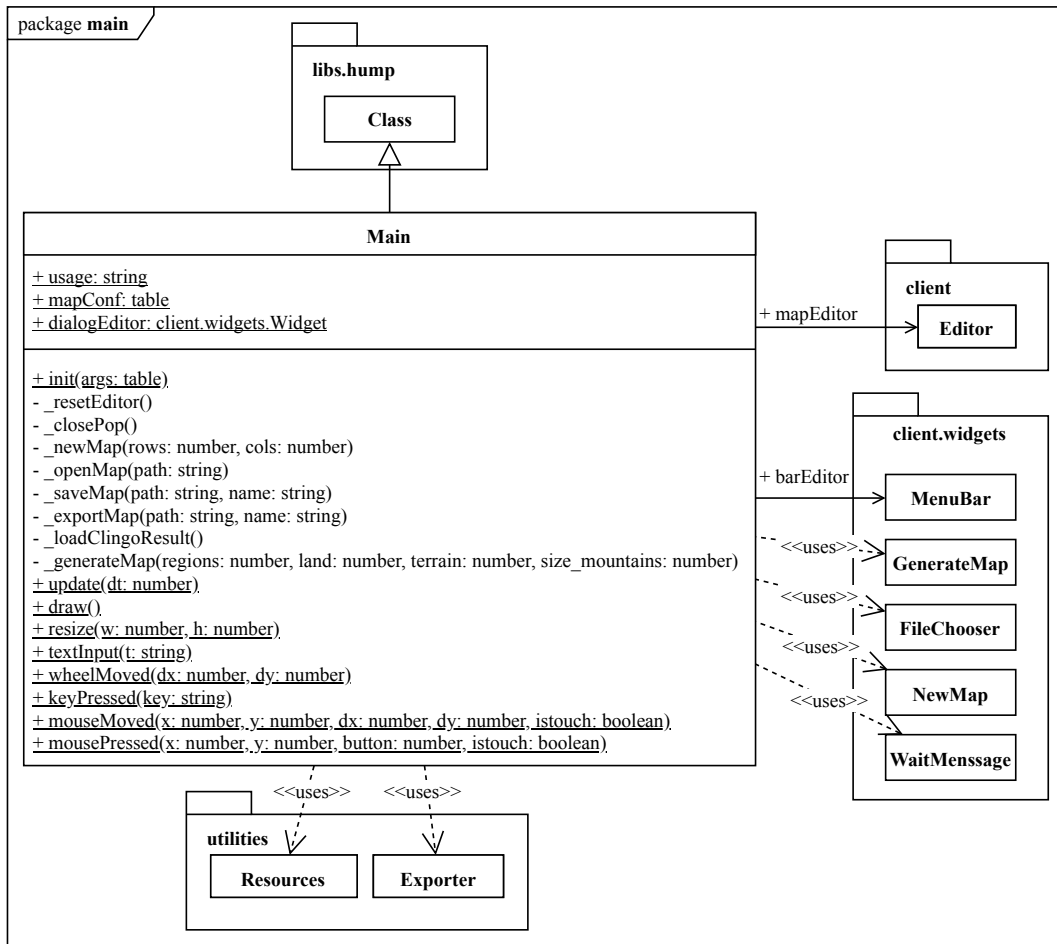


Figura 3.13: Diagrama de clases del paquete principal.

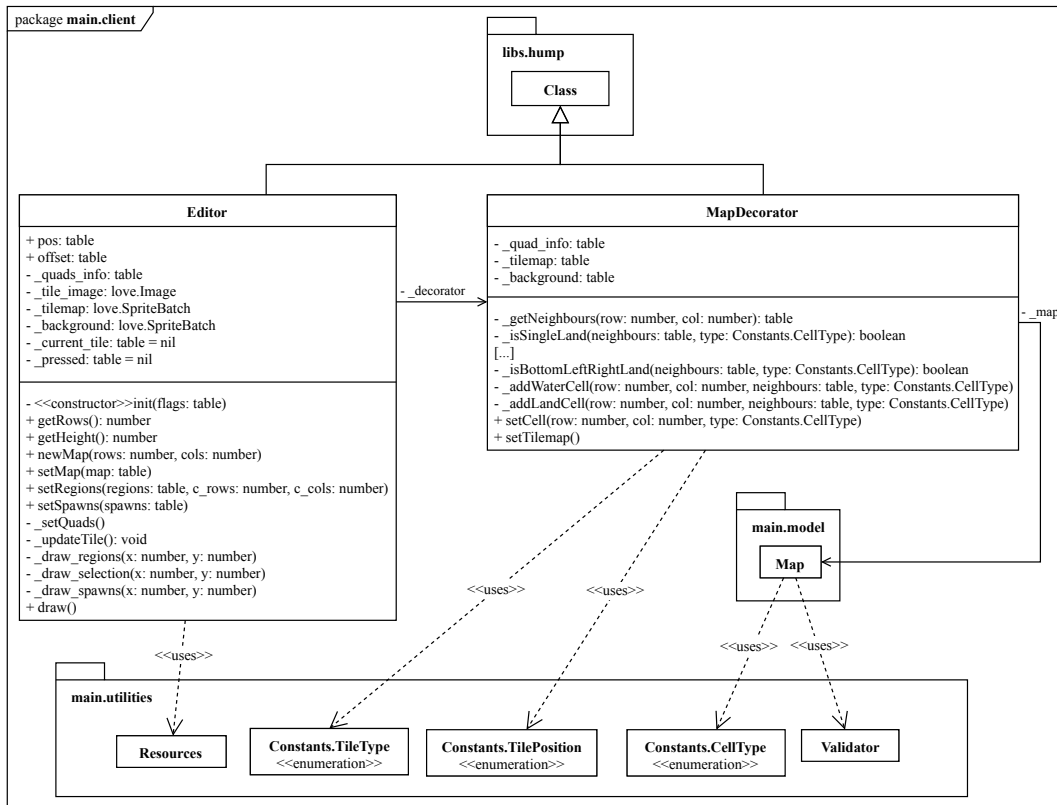
Como se puede ver en la Figura 3.13 se ha diseñado la clase principal **Main**, la cual es una clase estática que sirve de controlador para las vistas y el modelo. Implementa los casos de uso de la interfaz descritos en la Sección 3.4.2 en los métodos `_newMap`, `_openMap`, `_saveMap`, `_exportMap` y `_generateMap`, que son eventos que se llaman desde los diálogos emergentes tal y como se explica desde la Sección 3.4.5. Para el caso de uso de modificar el mapa, la tarea recae en la clase **Editor**, que se explica en detalle en la Sección 3.4.5.

Con respecto al motor gráfico LÖVE, este usa varias funciones predefinidas cuando se activan distintos eventos, los cuales llaman a los métodos públicos de la clase principal:

- El método `init` es llamado en la función `love.load` la primera vez que se ejecuta el programa. Sirve para cargar los elementos gráficos e iniciar el resto de clases que serán usadas por el programa.
- Como LÖVE es un motor para programación de videojuegos, la función `love.update` es llamada en un bucle de eventos internos antes de actualizar la pantalla. Esta función llama al método homónimo de la clase principal, el cual prepara las vistas de la interfaz para su pintado en pantalla y llama a los métodos correspondientes según los eventos producidos en las vistas.
- Una vez actualizada la lógica, se procede al pintado de pantalla mediante la función `love.draw`, la cual llama al método con el mismo nombre de la clase principal, que se encarga de pintar las vistas en pantalla.
- Existen otros eventos, los cuales LÖVE tiene contempladas varias funciones por defecto adicionales que se lanzan para contestar a estos. Es el caso de cuando se redimensiona la ventana (`love.resize`), se introduce un texto mediante un teclado virtual (`love.textinput`), se realiza un movimiento de la rueda del ratón (`love.wheelmoved`), se presiona una tecla (`love.keypressed`), se mueve el ratón (`love.mousedmoved`) o se presiona un botón del ratón (`love.mousepressed`). Estas llaman a los métodos correspondientes de la clase principal.

Editor gráfico

La clase `Editor` sirve como un controlador para responder a las modificaciones y cambios de representación del mapa, ya sea cuando se actualiza este mediante uno de los casos de uso que recoge la clase principal [ver Sección 3.4.5] o cuando el usuario procede a la modificación del mapa, tal y como se

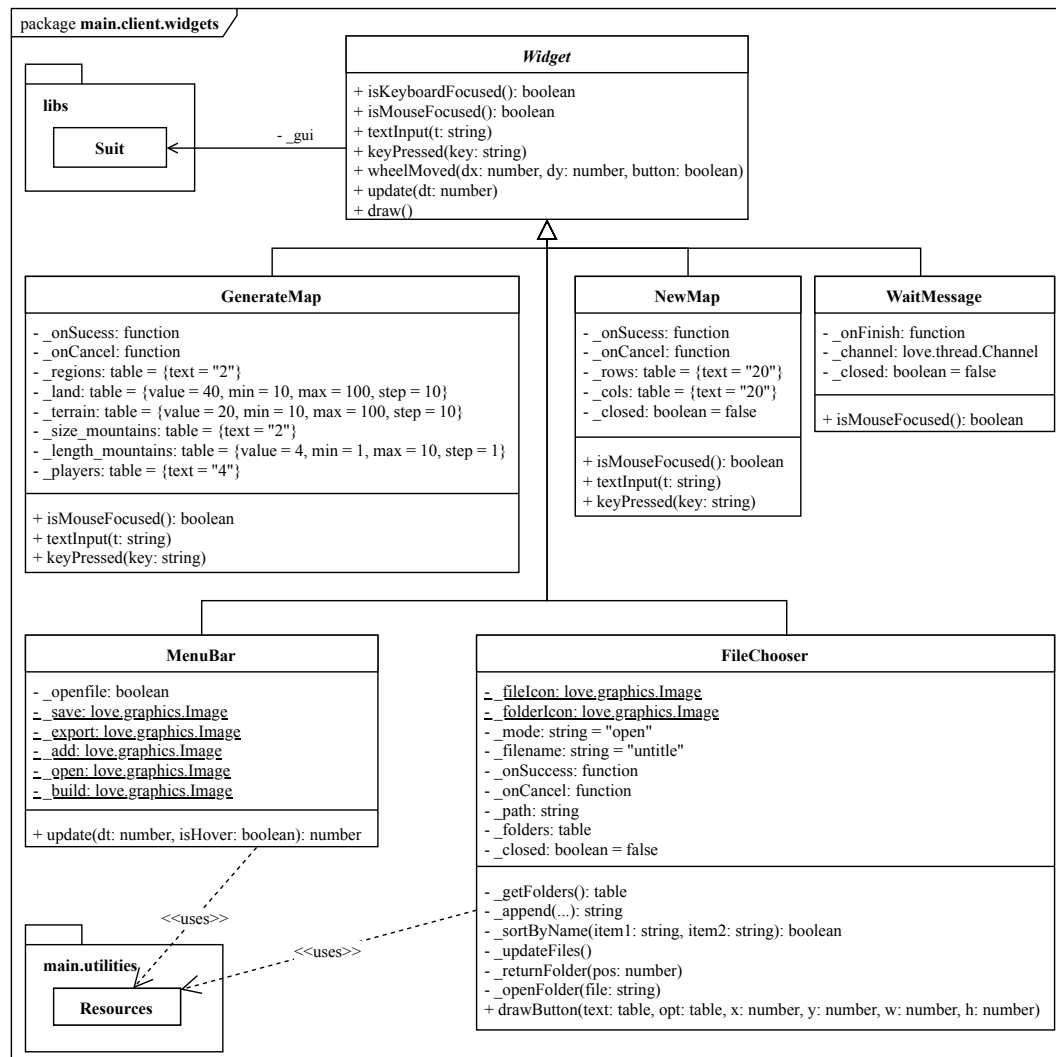
Figura 3.14: Diagrama de clases del paquete `Client`.

explica en la Sección 3.4.2.

Para ello, siguiendo el patrón decorador, esta se apoya en la clase `MapDecorator`, la cual se encarga de actualizar la vista del mapa, que es representada mediante una rejilla con los distintos terrenos mediante el objeto `love.SpriteBatch`, el cual es un mapa de *tiles* bidimensional. Debido a que hay terrenos que contienen diferentes imágenes para las posiciones de un *tile*, `MapDecorator` contiene varios métodos que permiten discretizarlas conociendo los vecinos de una celda. Finalmente esta clase llama al modelo en si, representado por la clase `Map`, la cual tiene una representación del mapa en forma de tabla.

Finalmente todas estas clases usan constantes que están definidas el módulo `Constants`.

Elemento gráficos complejos

Figura 3.15: Diagrama de clases del paquete `Widget`

Como ya expliqué en la entrada de la Sección 3.4.5, debido a las limitaciones de la biblioteca *SUIT*, hay varios elementos gráficos que no se incluyen en ella debido a que son complejos y no es del ámbito de esta biblioteca, es por eso que se han realizado a mano. Estos elementos tienen su propio controlador para poder usarlos más fácilmente dentro del programa.

- Para la barra de herramientas se ha realizado la clase `MenuBar`, la cual controla y muestra la lista de botones que accionan los principales casos de uso. Al constructor se le pasa las funciones para los eventos de aceptar

y cancelar diálogo. Tiene un método `update` que devuelve un número que se corresponde con el botón pulsado en la barra.

- La clase `NewMap` controla y muestra la ventana emergente que se puede ver en la Figura 3.4. Al constructor se le pasa las funciones para los eventos de aceptar y cancelar diálogo. Contiene el método `isMouseFocused` que indica si el ratón está encima de la ventana, y los métodos `textInput` y `keyPressed` para introducir texto en las entradas de la ventana.
- La clase `GenerateMap` controla y muestra la ventana emergente con el diálogo de generar mapa, tal y como se puede ver en la Figura 3.7. Contiene los mismos métodos que la clase `NewMap`.
- La clase `WaitMessage` controla y muestra la ventana emergente que se puede ver en la Figura 3.8. Al constructor se le pasa el canal del *thread* que se usa en la generación del mapa, tal y como se expone en la Sección 3.4.4, y la función que se lanza cuando la generación termina. Contiene el método `isMouseFocused` que indica si el ratón está encima de la ventana.
- La clase `FileChooser` controla y muestra la ventana emergente correspondiente a un diálogo de selección de archivo, tal y como se puede ver en las Figuras 3.5, 3.6 y 3.9. Contiene varios métodos privados para la ordenación y la apertura de carpetas en disco, así como el método `isMouseFocused` que indica si el ratón está encima de la ventana, los métodos `textInput` y `keyPressed` para introducir texto en las entradas de la ventana y el método `wheelMoved` que permite subir y bajar la lista de carpetas y ficheros mediante la rueda del ratón.

3.5. Ejemplos de uso

Ahora pasaré a explicar como usar la utilidad creada. Para ello hay que tener en cuenta que previamente tiene que estar instalado `clingo` 5.2, `LÖVE` 11.0 y la biblioteca de `LuaFilesystem`. Se recomienda usar un sistema GNU/Linux, ya que es donde se han realizado las pruebas. Para empezar a usar la

3. Trabajo desarrollado

interfaz, vamos a abrir un terminal y ejecutar LÖVE en el directorio donde esté instalada la herramienta con el comando:

```
love .
```

Se nos abrirá la pantalla 3.16, en donde podemos usar las funciones de crear un mapa en blanco (+) o abrir un fichero (📁).

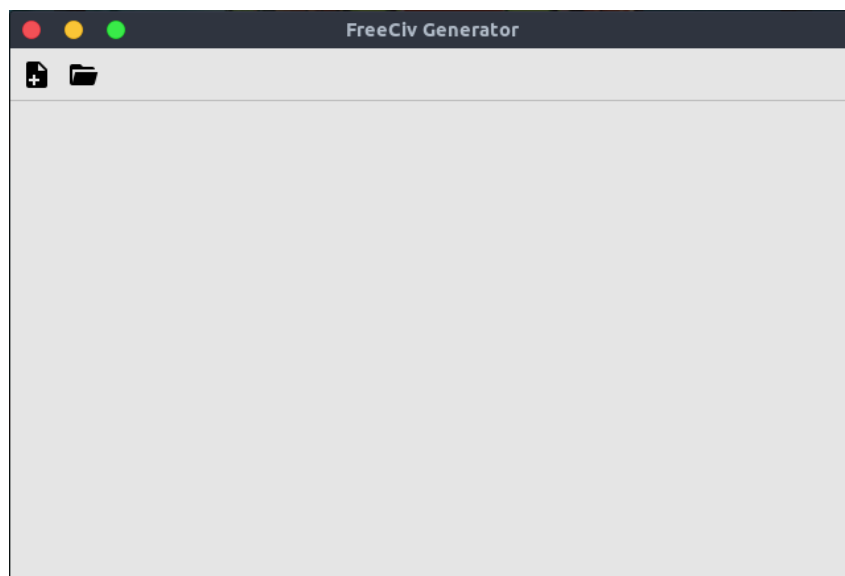


Figura 3.16: Pantalla principal de la aplicación.

Si seleccionamos crear un nuevo mapa, nos preguntará por las dimensiones y aparecerá un nuevo mapa con todas las casillas vacías. Se nos activarán las opciones para guardar el mapa (📁), exportar el mapa a formato Freeciv (📤) y generar un nuevo mapa (🔧).

Podemos seleccionar ahora generar mapa. Nos aparecerá el diálogo de la Figura 3.18 preguntando por parámetros para la generación. Una vez los hayamos ajustado, aceptamos y empezará a generar el mapa.

Una vez terminado nos mostrará el resultado de la generación rellenando todas las celdas. Con esto, podemos exportar para usar en Freeciv.

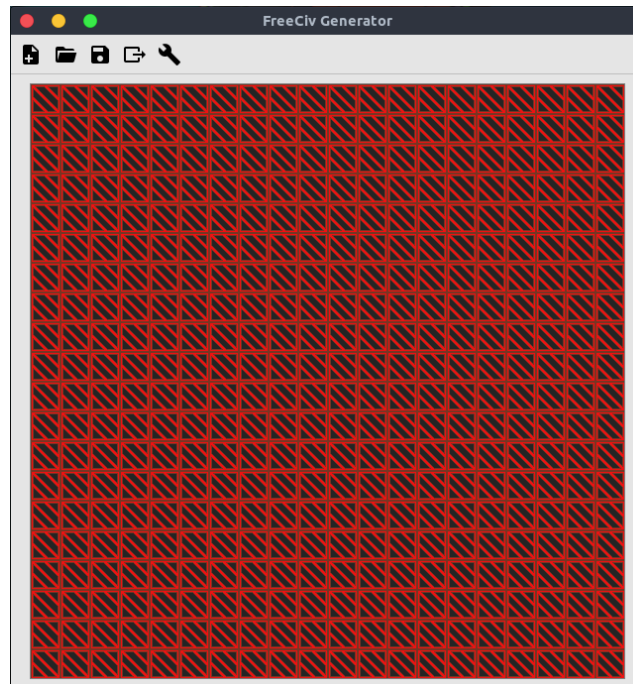


Figura 3.17: Pantalla con un mapa vacío

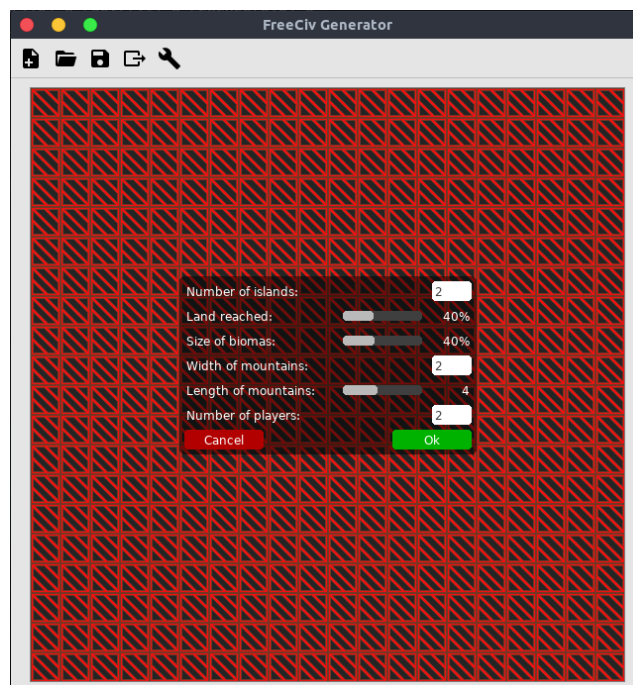


Figura 3.18: Diálogo con los parámetros de generar mapa.



Figura 3.19: Pantalla con un mapa vacío

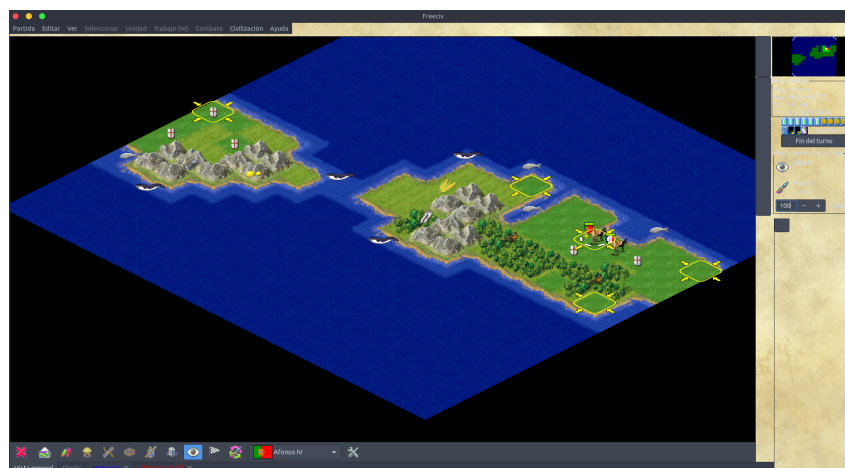


Figura 3.20: Pantalla con un mapa vacío

Podemos también pensar en modificar el modelo lógico del programa para añadir restricciones a mano. Para ello vamos a proponer que no nos gusta que el sistema genere los puntos iniciales y finales de las montañas pegadas a la costa. Para ello vamos a añadir dos restricciones al archivo `main/asp/detail_mountains.lp` en donde usaremos los átomos `start(P)` y `end(P)` junto con las celdas adyacentes que se pueden obtener con el átomo `adj(P,Q)` y negar que tenga el átomo `land(P)`. Quedaría de la siguiente forma:

```
:- start(P), adj(P, Q), pos(Q), not land(Q).  
:- end(P), adj(P, Q), pos(Q), not land(Q).
```

Sin cerrar el programa principal podemos volver a pulsar el botón de generar, el cual podrá dar como resultado el mapa de la Figura 3.21.



Figura 3.21: Ejemplo de ejecución con restricciones en la generación de montañas.

Capítulo 4

Evaluación

En este capítulo se analizará los resultados obtenidos en el proceso de evaluación del sistema. Para la planificación de las pruebas se ha optado por el uso de una herramienta de integración continua en donde se corren tanto pruebas de unidad a las partes del modelo de la interfaz gráfica, así como a los distintos módulos de los programas lógicos.

Además se ha usado un programa escrito en Lua para la ejecución de pruebas de rendimiento o *benchmarks* el cual solo usa el módulo *clingo* para evitar el uso de la interfaz gráfica. En todas estas pruebas se ha añadido a las pruebas la restricción de que el cómputo del mapa solo puede durar como máximo cinco minutos, matando el proceso en caso de superar este tiempo y pasando a la siguiente prueba. Con esto podemos indicar cuales son las configuraciones que dan una experiencia pobre al usuario.

A continuación se describen los resultados obtenidos de estas pruebas de rendimiento con distintos parámetros.

4.1. Tamaño de los mapas

Estas pruebas se han realizado usando distintos tamaños de mapas cuadrados, dejando el resto de variables con valores por defecto (cantidad de tierra y de biomas al 20 %, tamaño de las cordilleras en 2 casillas, longitud de cordilleras en 3 casillas, 2 jugadores y 2 casillas de distancia mínima entre jugadores).

Debido a que el módulo pide el número de cuadrantes y el tamaño en casillas de un cuadrante, se han realizado las pruebas calculando primero el tamaño total del mapa y con esto se ha obtenido tres números:

$$islands = \lfloor \sqrt{size} - 1 \rfloor \quad (4.1)$$

$$n_1 | size, \text{ en donde } n_1 > islands. \quad (4.2)$$

$$n_2 = size / n_1 \quad (4.3)$$

Con esto se procede a obtener el número de cuadrantes y el tamaño del cuadrante mediante:

$$q_{num} = \min(n_1, n_2) \quad (4.4)$$

$$q_{size} = \max(n_1, n_2) \quad (4.5)$$

$$(4.6)$$

Con esto nos aseguramos de que cada uno de los módulos tengan porciones parecidas para computar. En la Tabla 4.1 podemos ver los resultados de la realización de esta prueba de rendimiento ejecutando 25 ejecuciones cada una de las medidas, en donde se recoge el tiempo total de todas las ejecuciones, el tiempo total que ha usado clingo para obtener una solución y el tiempo medio de una ejecución. Por otra parte en la Figura 4.1 podemos ver una comparativa entre el tiempo medio entre ejecuciones y el tiempo medido arrojado por clingo.

Como se puede observar, el tiempo de ejecución sigue una tendencia casi exponencial, haciendo que a partir de mapas de 45x45 celdas en ningún mo-

Tamaño	Islas	q_{num}	q_{size}	Total	Ejecución
10 x 10	2	2	5	6 s	117 ms
15 x 15	2	3	5	17 s	145 ms
20 x 20	3	4	5	88 s	296 ms
25 x 25	4	5	5	200 s	675 ms
30 x 30	4	5	6	1137 s	731 ms
35 x 35*	4	5	7	-	-
40 x 40	5	5	8	1791 s	1050 ms
45 x 45	5	5	9	4627 s	1047 ms

Las filas indicadas con - se refieren a las pruebas en donde una ejecución no se han podido completan en menos de 5 minutos.

Tabla 4.1: Resultado del *benchmark* con diferentes tamaños de mapa.

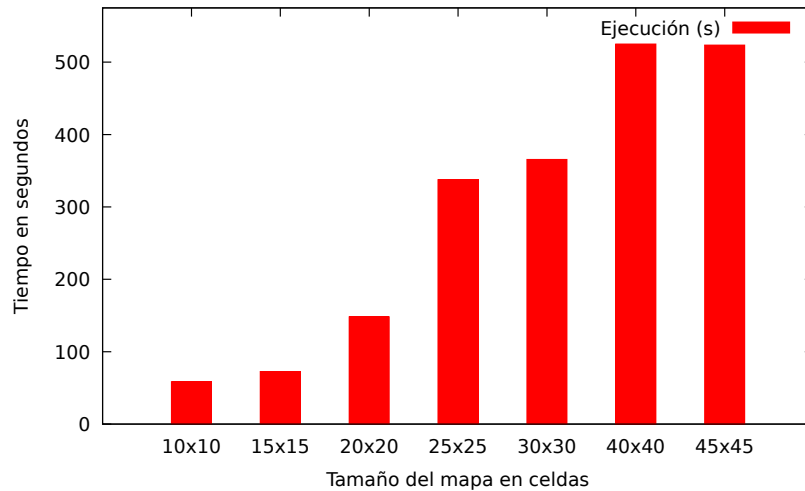


Figura 4.1: Tiempos del *benchmark* con diferentes tamaños de mapa.

mento la ejecución tarde menos de 5 minutos. Aún así, hay que destacar un caso concreto, ya que la prueba de un mapa de 35x35 celdas con los valores propuestos se realiza en más de 5 minutos, cosa que tanto en las pruebas anteriores como en las siguientes no ocurre. Esto puede deberse a que el mapa para esta prueba no esté bien repartido para los distintos módulos.

4.2. Porcentajes de tierra y biomas

Para la primera parte de la prueba se ha usado distintos porcentajes de tierra, usando la cantidad de islas y los parámetros de cantidad y tamaño de los cuadrantes para mapas de 25x25 celdas de la prueba anterior y dejando el resto de variables con valores por defecto (cantidad de biomas al 20 %, tamaño de las cordilleras en 2 casillas, longitud de cordilleras en 3 casillas, 2 jugadores y 2 casillas de distancia mínima entre jugadores).

Porcentaje	Total	Ejecución
10 %	174 s	6,96 s
15 %	178 s	7,12 s
20 %	178 s	7,12 s
25 %	179 s	7,16 s
30 %	181 s	7,24 s
35 %	178 s	7,12 s
40 %	179 s	7,16 s
45 %	178 s	7,12 s
50 %	181 s	7,24 s
55 %	179 s	7,16 s
60 %	178 s	7,12 s
65 %	179 s	7,16 s
70 %	178 s	7,12 s
75 %	180 s	7,20 s
80 %	180 s	7,20 s

Tabla 4.2: Resultado del *benchmark* con diferentes porcentajes de tierra.

En la Tabla 4.2 podemos ver los resultados de la realización de esta prueba de rendimiento ejecutando 25 ejecuciones cada una de las medidas, en donde se recoge el tiempo total de todas las ejecuciones, el tiempo total que ha usado clingo para obtener una solución y el tiempo medio de una ejecución. Por otra

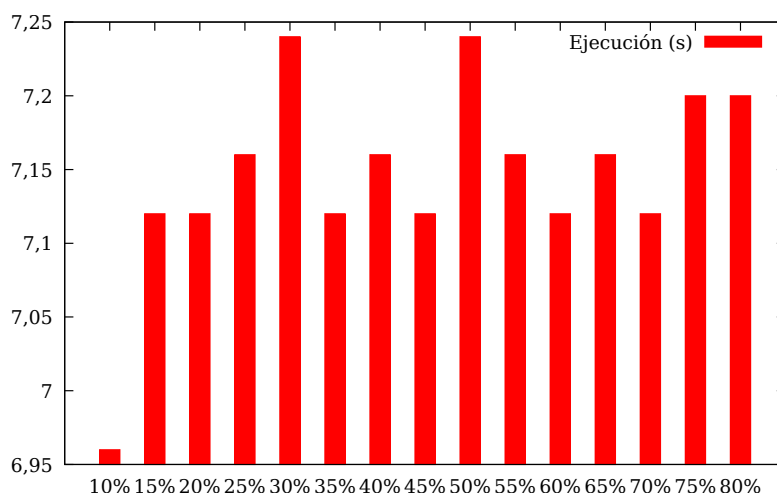


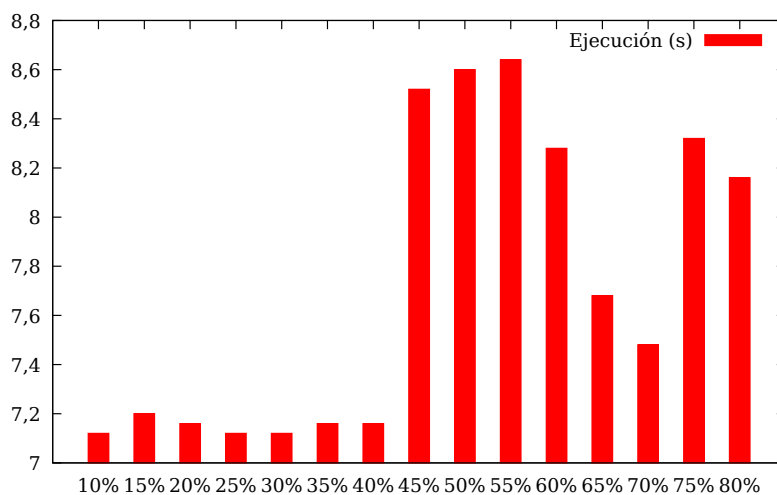
Figura 4.2: Tiempos del *benchmark* con diferentes porcentajes de tierra.

parte en la Figura 4.2 podemos ver una comparativa entre el tiempo medio entre ejecuciones y el tiempo medido arrojado por clingo.

En el caso de la segunda parte de la prueba se ha usado distintos porcentajes de cantidad de bioma que alcanza una isla, usando los mismos datos que para la prueba anterior. En la Tabla 4.3 podemos ver los resultados de la realización de esta prueba de rendimiento ejecutando 25 ejecuciones cada una de las medidas, en donde se recoge el tiempo total de todas las ejecuciones, el tiempo total que ha usado clingo para obtener una solución y el tiempo medio de una ejecución. Por otra parte en la Figura 4.3 podemos ver una comparativa entre el tiempo medio entre ejecuciones y el tiempo medido arrojado por clingo.

Como se puede observar, en estas pruebas no se haya bastante diferencia al modificar uno de los valores, dando resultados muy similares entre si. Es por eso que se ha decidido combinar las dos pruebas en una sola para observar algún cambio efectivo. Esto se recoge en la Sección 4.3.

Porcentaje	Total	Ejecución
10 %	178 s	7,12 s
15 %	180 s	7,20 s
20 %	179 s	7,16 s
25 %	178 s	7,12 s
30 %	178 s	7,12 s
35 %	179 s	7,16 s
40 %	179 s	7,16 s
45 %	213 s	8,52 s
50 %	215 s	8,60 s
55 %	216 s	8.64 s
60 %	207 s	8,28 s
65 %	192 s	7,68 s
70 %	187 s	7,48 s
75 %	208 s	8,32 s
80 %	204 s	8,16 s

Tabla 4.3: Resultado del *benchmark* con diferentes porcentajes de tierra.Figura 4.3: Tiempos del *benchmark* con diferentes porcentajes de tierra.

4.3. Prueba mixta tierra/biomas

En esta última prueba se ha tenido en cuenta los datos de la prueba relatada en la Sección 4.2, es decir, usando un mapa de 25x25 dividido en 5 cuadrantes de 5 celdas cada uno, con 4 islas totales y dejando el resto de variables con valores por defecto (tamaño de las cordilleras en 2 casillas, longitud de cordilleras en 3 casillas, 2 jugadores y 2 casillas de distancia mínima entre jugadores).

	10 %	15 %	20 %	25 %	30 %	35 %
10 %	42	41	35	88	117	214
15 %	41	37	36	88	133	247
20 %	37	43	37	105	123	224
25 %	29	51	43	92	123	220
30 %	42	43	138	95	-	-
35 %	41	40	38	90	-	241
40 %	43	46	36	-	-	217
45 %	69	49	36	737	-	220
50 %	100	-	1064	-	-	-
55 %	55	956	-	-	-	-
60 %	164	740	-	-	-	-
65 %	97	838	-	-	-	-

Tabla 4.4: Resultado del *benchmark* con diferentes porcentajes de tierra (las columnas) y de biomas (las filas). Los tiempos totales son en segundos.

Como se puede observar en los resultado mostrados en la Tabla 4.4, en donde se han ejecutado 5 iteraciones por cada valor, el módulo de clingo obtiene resultados decentes cuando se establecen valores bajos para el porcentaje de tierra y de biomas, más el tiempo de ejecución se dispara en el momento de poner valores altos. Esto se debe a, como ya se ha comentado a lo largo de la memoria, el paradigma ASP puede llegar a tener en cuenta una gran combinación de datos y producirse una explosión combinatoria de soluciones,

4. Evaluación

retardando en buena medida la procura de una solución.

Capítulo 5

Conclusiones

En este proyecto se construido una herramienta declarativa con la que elaborar, mediante programación lógica, nuevos escenarios para el videojuego Freeciv. Para ello se usó el paradigma de *Answer Set Programming*, una variante de Programación Lógica de uso frecuente para la Representación del Conocimiento y la resolución de problemas. La principal ventaja de *Answer Set Programming* para este caso fue la facilidad que otorgaba el uso de predicados simples a la hora de definir ciertas propiedades dentro de un escenario concreto, así como añadir directamente reglas de restricción para ciertos elementos del mapa bajo la forma de reglas de programación lógica. Esto proporcionó una gran flexibilidad, ya que con ello sólo se necesita realizar la especificación del problema sin importar el método de resolución que se aplicará.

También se ha reducido el problema que surge al tener en cuenta todas las reglas y restricciones del mapa, lo cual produce una explosión combinatoria de soluciones, y ocasionaba que el sistema tardase en encontrar. Para ello se ha separado el problemas de búsqueda del modelo declarativo en distintos módulos independientes: un generador de regiones, un módulo para rellenar las regiones, un generador de biomas, un módulo que define cordilleras, un módulo para definir las celdas de agua y por último un módulo para definir los puntos iniciales de los jugadores.

A pesar de todo esto, existen algunas limitaciones de cara la finalización de este proyecto, las cuales pueden ser resueltas en un futuro inmediato:

- El sistema no tiene en cuenta todos los elementos que proporciona el videojuego, por lo que no soporta actualmente la generación de ríos, ni la generación de recursos ni la generación de puntos de inicio para bárbaros o aldeas perdidas. Esto se podría solucionar añadiendo nuevos módulos a la generación del mismo.
- El sistema presenta graves problemas de eficiencia a la hora de generar grandes porciones de mapas, mermando la funcionalidad de la herramienta. Esto se puede observar en la Sección 4, en donde había problemas con mapas mayores de 45x45 celdas en cuanto a tiempo de ejecución. Esto se podría resolver intentando pulir las reglas actuales de los módulos o replanteando la forma en la que se han definido, en algunos casos dividiendo más para reducir la carga de estos.

Para finalizar, se puede marcar una serie de líneas, tanto para solucionar y mejorar el sistema propuesto como para ampliar y extender las funcionalidades actuales de cara a largo plazo:

- Mejorar la interfaz gráfica, incluyendo distintos elementos gráficos con los que manejar de forma más efectiva el mapa, como puede ser herramientas para ampliar y disminuir el mapa, mover el mapa o marcar de forma más visual el tipo de terreno marcar.
- Indicar de forma más visual restricciones en el mapa, así como zonas que tendrá en cuenta el generador para completar. Estas zonas incluso pueden tener restricciones de que piezas no poner o preferencias de los tipos de terrenos a generar.
- Publicar y anunciar la herramienta para tener una buena base de usuarios que puedan producir un *feedback* del uso y características de la misma.

-
- Añadir a la base de conocimiento estilos personalizados, ya sea incluyéndolos dentro de las reglas o importándolos como perfiles. Estos permitirían que un usuario pueda tener preferencias de cara a la generación (que no le guste generar cordilleras cerca del agua, que los usuarios estén lo más cercano a bosques o recursos, etc).

Índice de tablas

3.1. Tipos de terreno.	25
3.2. Tipos de terreno.	26
3.3. Desglose de las iteraciones.	29
3.4. Coste de los recursos humanos del proyecto.	30
3.5. Coste de los recursos humanos del proyecto.	30
4.1. Resultado del <i>benchmark</i> con diferentes tamaños de mapa. . . .	61
4.2. Resultado del <i>benchmark</i> con diferentes porcentajes de tierra. .	62
4.3. Resultado del <i>benchmark</i> con diferentes porcentajes de tierra. .	64
4.4. Resultado del <i>benchmark</i> con diferentes porcentajes de tierra (las columnas) y de biomas (las filas). Los tiempos totales son en segundos.	65

Índice de figuras

2.1. Ejemplo de escenario de Freeciv	11
2.2. Ejemplo de ejecución de ASP	14
3.1. Diagrama de Gantt con las iteraciones del proyecto.	28
3.2. Arquitectura del sistema.	33
3.3. Casos de uso del sistema propuesto.	34
3.4. Pantalla de nuevo mapa.	34
3.5. Pantalla de abrir archivo de mapa.	35
3.6. Pantalla de guardar mapa.	35
3.7. Pantalla de generar mapa.	36
3.8. Pantalla con el mensaje de espera.	36
3.9. Pantalla de exportar mapa.	37
3.10. Pantalla principal de la aplicación.	37
3.11. Ejemplo de mapa dividido en cuadrantes	38
3.12. Diagrama de secuencia de la ejecución del generador.	48
3.13. Diagrama de clases del paquete principal.	49
3.14. Diagrama de clases del paquete <code>Client</code>	51
3.15. Diagrama de clases del paquete <code>Widget</code>	52
3.16. Pantalla principal de la aplicación.	54
3.17. Pantalla con un mapa vacío	55
3.18. Diálogo con los parámetros de generar mapa.	55
3.19. Pantalla con un mapa vacío	56
3.20. Pantalla con un mapa vacío	56

3.21. Ejemplo de ejecución con restricciones en la generación de mon-	
tañas.	57
4.1. Tiempos del <i>benchmark</i> con diferentes tamaños de mapa.	61
4.2. Tiempos del <i>benchmark</i> con diferentes porcentajes de tierra.	63
4.3. Tiempos del <i>benchmark</i> con diferentes porcentajes de tierra.	64

Bibliografía

- [1] E. J. Hastings, R. K. Guha, y K. O. Stanley, “Evolving content in the galactic arms race video game,” en *2009 IEEE Symposium on Computational Intelligence and Games*, Sept 2009, pp. 241–248.
- [2] S. Parkin, “A science fictional universe created by algorithms,” May 2016. [Online]. Disponible en: <https://www.technologyreview.com/s/529136/no-mans-sky-a-vast-game-crafted-by-algorithms/>
- [3] T. Whitted, “An improved illumination model for shaded display,” *Commun. ACM*, vol. 23, n.º 6, pp. 343–349, Jun 1980. [Online]. Disponible en: <http://doi.acm.org/10.1145/358876.358882>
- [4] S. G. Parker, H. Friedrich, D. Luebke, K. Morley, J. Bigler, J. Hoberock, D. McAllister, A. Robison, A. Dietrich, G. Humphreys, M. McGuire, y M. Stich, “Gpu ray tracing,” *Commun. ACM*, vol. 56, n.º 5, pp. 93–101, May 2013. [Online]. Disponible en: <http://doi.acm.org/10.1145/2447976.2447997>
- [5] G. Brewka, T. Eiter, y M. Truszczyński, “Answer set programming at a glance,” *Commun. ACM*, vol. 54, n.º 12, pp. 92–103, Dic 2011. [Online]. Disponible en: <http://doi.acm.org/10.1145/2043174.2043195>
- [6] A. M. Smith y M. Mateas, “Answer set programming for procedural content generation: A design space approach,” vol. 3, pp. 187 – 200, 10 2011.
- [7] R. Martín Prieto, “Herramienta para armonización musical mediante answer set programming,” Universidade da Coruña, 2017. [Online].

- Disponible en: https://github.com/Trigork/haspie/blob/master/docs/tech_report/root.pdf
- [8] G. Boenn, M. Brain, M. D. Vos, y J. ffitich, “Automatic music composition using answer set programming,” *CoRR*, vol. abs/1006.4948, 2010. [Online]. Disponible en: <http://arxiv.org/abs/1006.4948>
 - [9] “Service name and transport protocol port number registry,” IANA, Jan 2006. [Online]. Disponible en: <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=5556>
 - [10] “Gnu general public license,” Free Software Foundation. [Online]. Disponible en: <http://www.gnu.org/licenses/gpl.html>
 - [11] M. Gelfond y V. Lifschitz, “The stable model semantics for logic programming.” en *Proceedings of International Logic Programming Conference and Symposium*, vol. 88, 1988, pp. 1070–1080.
 - [12] S. Hanks y D. McDermott, “Nonmonotonic logic and temporal projection,” *Artificial Intelligence*, vol. 33, n.º 3, pp. 379 – 412, 1987. [Online]. Disponible en: <http://www.sciencedirect.com/science/article/pii/0004370287900439>
 - [13] K. L. Clark, “Readings in nonmonotonic reasoning,” M. L. Ginsberg, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, cap. Negation As Failure, pp. 311–325. [Online]. Disponible en: <http://dl.acm.org/citation.cfm?id=42641.42664>
 - [14] R. Ierusalimschy, *Programming in Lua, Fourth Edition*. Lua.Org, 2016.
 - [15] “The JSON Data Interchange Format,” ECMA, Tech. Rep. Standard ECMA-404 1st Edition / October 2013, Oct 2013. [Online]. Disponible en: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

- [16] K. Schwaber y M. Beedle, *Agile Software Development with Scrum*, 1º ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.
- [17] K. Schwaber, *Agile Project Management With Scrum*. Redmond, WA, USA: Microsoft Press, 2004.
- [18] “Guía salarial del sector ti en galicia,” Vitae Consultores, 2016.
- [19] W. Faber, G. Pfeifer, N. Leone, T. Dell’armi, y G. Ielpa, “Design and implementation of aggregate functions in the dlν system*,” *Theory Pract. Log. Program.*, vol. 8, n.º 5-6, pp. 545–580, Nov 2008. [Online]. Disponible en: <http://dx.doi.org/10.1017/S1471068408003323>