



UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA
DEPARTAMENTO DE COMPUTACIÓN

TRABAJO FIN DE MÁSTER
MÁSTER UNIVERSITARIO EN INGENIERÍA
INFORMÁTICA

**IndustryLP: Generador de polígonos
industriales para el videojuego Cities:
Skylines© mediante Programación Lógica**

Autor: Rafael Alcalde Azpiazu
Director: José Pedro Cabalar Fernández

A Coruña, 7 de septiembre de 2021

Especificación

Título del proyecto: IndustryLP: Generador de polígonos industriales para el videojuego Cities: Skylines© 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 **IndustryLP: Generador de polígonos industriales para el videojuego Cities: Skylines©** 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, 7 de septiembre de 2021

TODO...

Agradecimientos

Todo...

Rafael Alcalde Azpiazu
A Coruña, 7 de septiembre de 2021

Resumen

Este proyecto consiste en automatizar parcialmente la planificación urbanística de parques industriales en un videojuego de simulación mediante técnicas de razonamiento automático y representación del conocimiento. Para ello se construirá una herramienta declarativa gobernada por reglas que genere el entramado del polígono industrial y cuya salida pueda ser empleada por el simulador de ciudades Cities: Skylines©, que permite crear, modificar y ejecutar una simulación de un ambiente urbano, y del que se pueden extraer características que se pueden contemplar para la generación del parque, tales como información del tráfico, número de habitantes o zonas de viviendas cercanas.

Palabras clave

ANSWER SET PROGRAMMING, PROGRAMACIÓN LÓGICA, REPRESENTACIÓN DEL CONOCIMIENTO, RESOLUCIÓN DE PROBLEMAS LÓGICOS, GENERACIÓN DE POLÍGONOS INDUSTRIALES, CITIES: SKYLINES©, .NET FRAMEWORK, STEAM, MOD

Índice general

1. Introducción	1
1.1. Motivación	3
1.2. Objetivos	5
1.3. Estructura de la memoria	5
1.4. Plan de trabajo	6
2. Contexto	9
2.1. Videojuegos de construcción de ciudades	9
2.2. Tecnologías	10
2.2.1. Answer Set Programming	11
2.2.2. .NET Framework	14
2.3. Herramientas	16
2.3.1. Microsoft Visual Studio	16
2.3.2. Git	17
3. Trabajo desarrollado	19
3.1. Propuesta	19
3.2. Proceso de ingeniería	22
3.2.1. Metodología de desarrollo	23
3.2.2. Gestión del proyecto	24
3.3. Análisis del software	27
3.3.1. Requisitos funcionales	28
3.3.2. Requisitos no funcionales	29
3.4. Diseño del sistema	29

3.4.1. Arquitectura software	29
3.4.2. Casos de uso	31
3.4.3. Diseño del programa lógico	31
3.4.4. Implementación	37
4. Conclusiones	41
Apéndices	47
A. Manual de uso	47
A. Bibliografía	51

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 gráficas ^{1 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.

Es por ello que la idea central del proyecto es automatizar parcialmente la planificación de parques industriales en un videojuego de simulación urbanística mediante técnicas de representación del conocimiento. Para ello se construirá una herramienta declarativa gobernada por reglas que genere el entramado de un polígono industrial y cuya salida pueda ser empleada por el simulador de ciudades de Cities: Skylines©, que permite crear, modificar y ejecutar una simulación de un ambiente urbano, y del que se pueden extraer

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

²<https://developer.nvidia.com/rtx>

características que se puedan contemplar para la generación del parque, tales como información del tráfico, número de habitantes o zonas de viviendas cercanas, entre otras.

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 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, se desarrollará un generador que

pueda crear un conjunto de entramados de carreteras dentro del videojuego Cities: Skylines©. En este entramado se colocarán elementos del juego tales como naves, empresas, zonas de aparcamiento y zonas verdes, además de establecer una conexión de esta zona con el resto de carreteras del entorno (conexiones con autovías para el transporte de mercancías y la conexión a las zonas residenciales), y la generación de conexiones con los servicios mínimos que necesitan los edificios (traída de agua y de residuos, y electricidad). El generador a construir será gobernado por un conjunto de reglas, expresadas mediante restricciones lógicas y lineales representadas en un programa lógico. De este modo, el generador se puede adaptar con gran flexibilidad a distintos criterios de planificación urbanística, permitiendo que cualquier experto habituado al juego experimente modificaciones de dichos criterios mediante simples cambios en las restricciones del programa. Como ejemplo de prueba para estos criterios, se utilizarán reglas inspiradas en leyes y ordenanzas municipales reales [5] que regulen la forma del entramado y construcción del polígono en el propio juego, así como criterios propios del ámbito de arquitectura y planificación urbanística que pueden estar dados por un experto que tenga un gran conocimiento del simulador. Para codificar las reglas que definen el problema, se utilizará el paradigma de programación lógica conocido como Answer Set Programming [6]. Este paradigma 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 realistas [7] 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. El proyecto estudiará distintas técnicas para reducir esta explosión combinatoria manteniendo en la medida de lo posible, el carácter declarativo de la herramienta. A su vez, el generador podrá obtener información de Cities: Skylines© sobre tráfico e información de los habitantes para poder generar la configuración más adecuada, ya sea incluyendo paradas de transporte público con las zonas residenciales,

restricciones de horarios o tipos de vehículos que transitan las conexiones con el entorno. Cabe remarcar que ya existen antecedentes de generación declarativa para el diseño de espacios o entornos [8] [9] usando este paradigma, así como el uso de Answer Set Programing en otros ámbitos donde también ocurre el mismo problema combinatorio, como puede ser la composición musical [10][11].

1.2. Objetivos

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

- Obtener un programa lógico (conjunto de predicados y reglas lógicas) que permita incorporar criterios y restricciones de diseño (simplificados) de un parque industrial.
- Permitir la entrada de información de un mapa de ciudad de Cities: Skylines© en forma de hechos para predicados del programa lógico.
- Permitir recuperar la salida obtenida por el programa lógico (en forma de hechos) y visualizarla como mapa de polígono industrial en Cities: Skylines©.
- Proporcionar una interfaz gráfica (ya sea mediante un editor 2D a medida o bien usando características del juego Cities: Skylines©) para marcar zonas de construcción o generar automáticamente restricciones lógicas que se quieran aplicar a las soluciones.

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. 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 de la API de desarrollo de plug-ins de Cities: Skylines®.
- Análisis del estado del arte sobre generación de entornos urbanos.

- Diseño del programa lógico: elección de los predicados relevantes para el diseño del polígono industrial en un mapa de Cities: Skylines©, y las reglas que los relacionan.
- Diseño e implementación de un módulo que traduzca el estado del mapa de Cities: Skylines© en un momento concreto a hechos para predicados del programa en Answer Set Programming.
- Diseño e implementación de un módulo del programa lógico que genere entramados de carreteras en Answer Set Programming.
- Diseño e implementación de un plug-in (de tipo mod) de Cities: Skylines© que permita, durante la ejecución del juego, marcar zonas y/o restricciones, llamar a la herramienta de Answer Set Programming para generar un polígono industrial y recuperar la salida de dicha herramienta mostrando el polígono ya construido en el juego.
- Evaluación de eficiencia para distintos casos de prueba creados a priori.
- 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. Videojuegos de construcción de ciudades

Los videojuegos de construcción de ciudades es un género dentro de los videojuegos de simulación, en el que los jugadores actúan como planificadores y líderes generales de una ciudad o pueblo, mirándola desde arriba, y siendo responsables de su crecimiento y estrategia de gestión. Los jugadores eligen la ubicación de los edificios y las características de gestión de la ciudad, como los salarios y las prioridades de trabajo, haciendo que la ciudad se desarrolle en consecuencia.

El exponente de este género es la serie de juegos de SimCity^{TM1} (Figura 2.1), creados por el desarrollador de videojuegos Will Wright y padre de la saga The SimsTM. Fue lanzado en 1989 y desde entonces, ha plateado las bases y ha servido de inspiración al resto de videojuegos dentro de este ámbito. A pesar de su fama, algunas malas decisiones en su última gran entrega (SimCityTM 2013) han hecho que haya perdido parte de su popularidad en favor de otros títulos.

Otro título dentro de este género es Cities: Skylines©², desarrollado por la productora filandesa Colossal Order, y publicado por la editorial de videojue-

¹<https://www.ea.com/es-es/games/simcity>

²<https://www.citiesskylines.com/>



Figura 2.1: Pantalla del juego SimCity™

gos indie Paradox Interactive AB. Pertenecce a la serie de Cities in Motion©. Una de sus principales bazas con respecto a otros de su género es la simulación realista de tráfico, poniendo el foco en un sistema para entender los problemas en el diseño de la ciudad, y el uso de sistemas de transporte público para remediar problemas de congestión, que se ven potenciados en distintas expansiones del juego. Otra gran baza es que es altamente personalizable, pudiendo agregar desde mapas (Figura 2.1) hasta modelos de objetos realistas, como edificios que existen en la realidad, coches de marcas conocidas, etc. Esto hace que esté entre el top 10 de títulos con más contenido creado por la comunidad del Workshop de Steam³, la plataforma por referencia de compra de videojuegos en formato digital. En este proyecto vamos a utilizar esta característica de este título para poder crear un añadido que nos permita ejecutar nuestro programa lógico dentro del videojuego, y publicarlo posteriormente en Steam.

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 multipropósito que nos permitirá desarrollar el plug-in que conectará nuestro

³<https://steamcommunity.com/app/255710/workshop>



Figura 2.2: Recreación de A Coruña en Cities: Skylines©

programa lógico con el videojuego en cuestión.

2.2.1. Answer Set Programming

Answer Set Programming (ASP) 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 [12], que usa para definir la semántica declarativa mediante programas lógicos normales. A esto se añade a que incorpora lógica no monótona [13], 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:

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

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

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

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 [14]. 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.

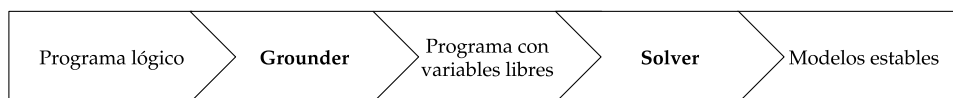


Figura 2.3: Ejemplo de ejecución de Answer Set Programing

Unas de las herramientas más importantes de Answer Set Programing son la de Potassco⁴, la cual son un conjunto de herramientas para Answer Set Programing 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 Answer Set Programing,

⁴<http://potassco.org>

Clingo. Así mismo, añade más funcionalidades al lenguaje Answer Set Programming, como puede ser al resolución iterativa, debido a que permite embeber otros lenguajes como Lua o Python, que pueden interactuar con el programa escrito en Answer Set Programming 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, qué debido a la naturaleza de Answer Set Programming, serán la base sobre la que se construirá todo el proyecto.

2.2.2. .NET Framework

.NET Framework⁵ es un Framework o entorno de desarrollo publicado por Microsoft para el sistema operativo Microsoft Windows. El software diseñado para ser ejecutado en .NET Framework está escrito en *Common Intermediate Language* o CIL, que es ejecutado en una entorno virtual llamado *Common Language Runtime*, tal y como se muestra en la Figura 2.2.2. Esta máquina virtual implementa técnicas de recolección de memoria, seguridad y manejo de hilos de ejecución, además de que el software es compilado a código máquina en tiempo de ejecución, obteniendo un mejor rendimiento que otros sistemas con máquina virtual como Java, en donde la implementación de referencia cuenta con una máquina virtual basada en stack.

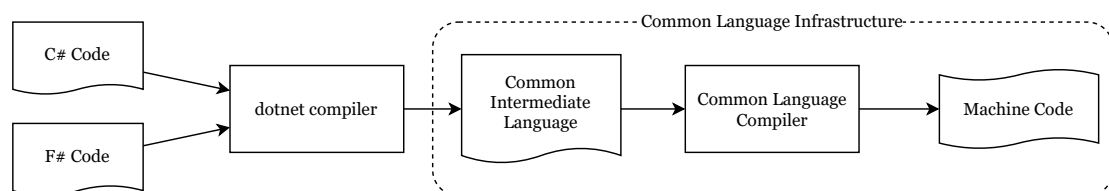


Figura 2.4: Arquitectura de .NET Framework

Uno de los principios de este Framework es que es independiente del lengua-

⁵<https://dotnet.microsoft.com/download/dotnet-framework>

je de programación usado, pudiéndose programar en C/C++, Java, C#, F# o J#. Estos tres últimos lenguajes también han sido diseñados por Microsoft, y fueron creados dentro del estándar de .NET Framework para cubrir diferentes necesidades (lenguaje imperativo y funcional en cada uno de los dos primeros casos, y una reimplementación del lenguaje Java en el último caso). En este proyecto usaremos en especial el lenguaje de programación C#, ya que al ser principalmente un lenguaje imperativo y de propósito general, nos permitirá desarrollar todo el código de este proyecto.

Debido a que .NET Framework es un entorno solo disponible para el sistema operativo Microsoft Windows, salieron varios proyectos que se ofrecen como alternativa para ser multiplataforma, como CrossNet, DotGNU⁶, o Mono⁷. En la actualidad el proyecto más usado es este último, que es desarrollado por el programador Miguel de Icaza, conocido por crear los proyectos GNOME y Xamarin (de la cual parte Mono). Es por eso que muchos proyectos que usan como lenguaje de programación C# y que quieren ser multiplataforma se basan en Mono y no en .NET Framework. Uno de ellos es el motor de desarrollo de videojuegos Unity, que es uno de los más importantes en la actualidad, y, en nuestro caso, sobre el que está programado Cities: Skylines®. Cabe destacar que en los últimos años, Microsoft desarrollo un Framework multiplataforma y de código abierto llamado .NET Core⁸, el cual implementa todas las características principales de .NET Framework. A partir de Noviembre de 2020 pasó a llamarse .NET, siendo este último el sucesor de .NET Framework.

Otra de las características de .NET Framework que nos servirán para este proyecto es la posibilidad de manejar código nativo (bibliotecas compiladas a código máquina) con código escrito en C#. Esto nos ha servido a la hora de poder usar las herramientas de Potassco, tal y como comentamos en la sección 3.4.1; ya que estas herramientas están escritas en C/C++, y en la actualidad

⁶<http://www.dotgnu.org/>

⁷<https://www.mono-project.com/>

⁸<https://dotnet.microsoft.com/download/dotnet>

no hay ninguna biblioteca que nos permita usarlas en código escrito para C#.

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. Microsoft Visual Studio

Microsoft Visual Studio⁹ es un Entorno de Desarrollo Integrado o IDE creado por Microsoft y disponible solo para el sistema operativo Microsoft Windows. Fue pensado en su inicio para desarrollar con las tecnologías Visual de Microsoft, como Visual C++, Visual J++ o Visual Basic, pero finalmente se ha convertido en el IDE de facto para desarrollar en las tecnologías .NET. Implementa tecnologías de autocompletado inteligente (IntelliSense™), así como de desarrollo de software (Azure DevOps mediante un plugin llamado Team Explorer), debugging, despliegue de sistemas basados en web, herramientas para el diseño de interfaces gráficas y diseño de esquemas de base de datos. Incluido a esto, se puede extender las funcionalidades de Microsoft Visual Studio mediante *plug-ins*, que están escritos en CIL, y que se pueden descargar desde Visual Studio Gallery (un repositorio centralizado de Microsoft).

Existe varias versiones de este editor: Enterprise, que está pensada para empresas que necesiten el despliegue en varios equipos; Professional, que está pensada para grupos con hasta 5 equipos; y Community, que solo permite una licencia única y es la gratuita. Además, existe una versión gratuita, de código libre y multiplataforma llamada Visual Studio Code, que está basado en Electron, un Framework para crear aplicaciones nativas con tecnologías Web.

⁹<https://visualstudio.microsoft.com/es>

Es debido a la fuerte integración que tiene con las tecnologías .NET, y que tanto .NET Framework como Microsoft Visual Studio está disponible en Microsoft Windows; que usaremos este IDE como una de las herramientas principales para el desarrollo de este proyecto.

2.3.2. Git

Git¹⁰ 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:

- GitLab¹¹, 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.
- Phabricator¹², 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.
- Bitbucket¹³, creado por Atlassian. Este sistema integra Mercurial desde

¹⁰<https://git-scm.com>

¹¹<https://gitlab.com>

¹²<https://www.phacility.com>

¹³<https://bitbucket.org>

sus inicios y Git desde 2011, y está pensado para ser integrado con el resto de productos como Atlassian como Jira, Confluence y Bamboo.

- GitHub¹⁴, creado por tres estudiantes estadounidenses, hoy en día es gestionado por GitHub Inc. empresa que fue adquirida por Microsoft en 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*; GitHub Marketplace, que permite comprar servicios con los que aumentar las funcionalidades en los proyectos y que muchos de ellos son gestionados por partners; y GitHub Actions, un sistema de automatización *CI/CD* para construir, lanzar test y desplegar proyectos dentro de GitHub.

Nosotros usaremos este último servicio como servidor de control de versiones y para montar el sistema de automatización *CI/CD*.

¹⁴<https://www.github.com>

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 área que contenga una planificación de un polígono industrial que pueda ser generado y jugado dentro del mapa de Cities: Skylines©. Así mismo incluirá una interfaz gráfica interactuable que permitan al usuario marcar que zonas del terreno deben generarse y cuales no, indicando su contenido antes de lanzar el proceso de generación.

Antes de proceder a explicar como se ha llevado a cabo, procederemos a explicar como funciona la industria en Cities: Skylines©.

Definición de Industria

Una de las principales mecánicas en Cities: Skylines© es la de crear zonas de los diferentes tipos de edificios que se construir en las zonas adyacentes a las

3. Trabajo desarrollado

carreteras, tal y como se muestra en la Figura 3.1. Estas zonas se dividen en una cuadrícula de 5 de ancho a lo largo de la carretera, en donde cada celda es de 8x8 metros. En estas zonas se pueden definir los siguientes tipos descritos a continuación.



Figura 3.1: Ejemplo de definición de zonas. En la azul las zonas comerciales y en verde las residenciales.

- **Residencial:** Corresponde a las casas donde la gente vivirá.
- **Comercial:** Se refiere a las tiendas y otros servicios que venden los productos creados por la industria o que son exportados.
- **Industrial:** Las zonas industriales proveen trabajo a los ciudadanos y crean los productos de consumo para los edificios comerciales.
- **Oficinas:** Las zonas de oficina dan trabajo solo a ciudadanos con alto nivel de estudios. Además no producen ni contaminación, ni tráfico, ni productos de consumo.
- **Servicios:** En esta categoría entran los edificios que no se pueden crear mediante zonas, si no que se fijan su construcción. Pertenecen al gobierno

y dan servicios a los ciudadanos, como parques, cuarteles de policía, estaciones de buses, etc.

En este proyecto nos centraremos en las zonas industriales. Son una de las zonas que aumentan el tráfico intensamente, por lo que su planificación debe ser rigurosa. Las zonas industriales se dividen en varios tipos dependiendo de los recursos naturales que puedan usar, que pueden ser renovables (suelo fértil) o no renovables (petróleo y minerales).

- **Genérica:** No tiene una especialización, por lo que corresponde a edificios genéricos. Es una de las más contaminantes pero es la base de la cadena de suministro en Cities: Skylines®. Proveen de trabajos para personas de estudios bajos.
- **Granjas:** Necesitan de suelo fértil (que es renovable) para producir recursos sin producir contaminación. Solo da trabajo para ciudadanos con nivel de estudios bajo.
- **Bosques:** No contamina el suelo pero genera un nivel de ruido significativo. Consume también más electricidad que la industria genérica. También da trabajo para gente con nivel de estudios bajo.
- **Petróquímica:** Usa el suelo que contiene petróleo (que no es renovable) para producir combustible. Genera ingresos por impuestos altos, así como altos niveles de contaminación y consumo alto de energía.
- **Minera:** Usa zonas con minerales (recurso no renovable) para producir carbón. Produce menos contaminación e ingresos que la petroquímica pero usa más energía que la industria genérica.

Desde el lanzamiento de la expansión *Industries*, se agregó un sistema producción, en donde se pueden procesar los recursos naturales hasta obtener productos de consumo de lujo. Para ello existen cinco tipos de edificios: extractores, procesadores, factorías, auxiliares y almacenes. Los edificios auxiliares dan zonas para que los trabajadores vivan y ofrecen mantenimiento, mientras



Figura 3.2: Interfaz de gestión de recursos de la expansión *Industries*

que los almacenes sirven para guardar cada uno de los productos de cada una de las etapas.

Además, desde la expansión de *Sunset Harbor* existe la industria pesquera, que agrega como recurso natural el pescado. Siguiendo el sistema de producción agregado en *Industries*, este se puede procesar hasta obtener productos de consumo que se puedan exportar o vender en comercios. Dependiendo de la profundidad y el flujo del agua, se generará uno de los cuatro tipos de pescados: anchoas, atún, salmón y marisco. La polución en el agua provoca que desaparezcan este recurso del agua.

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[15][16], 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, 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 un desarrollo en Sprint porque no se ha realizado un seguimiento exhaustivo del mismo. En el caso de gestión de riesgos se ha usado el Product backlog como backlog de historias para hacer, sin hacer estimación.

Debido a todos estos factores expuestos, se podría indicar que la metodología usada finalmente está más cerca de una metodología basada en kanbanflow, en donde las tareas del product backlog se mueven a un tablero con columnas según el estado de la tarea; o 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 la 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, para la planificación se ha usado una metodología más parecida a un kanbanflow o un modelo de prototipos, dividiendo en varias tareas incrementales que, una vez terminadas, dan nuevas funcionalidades al producto final. Debido a la naturaleza del proyecto, estas tareas se han dividido en 3 épicas:

1. **Generador del polígono industrial:** Las tareas presentes tienen que dar como producto final de esta épica un mod funcional para Cities: Skylines©. Dentro de esta épica tenemos las siguientes historias de usuario:

1. **Crear prototipo de complejo industrial:** El complejo debe tener dos carreteras que se unan en la mitad con un cruce y en cada uno de los cuatro espacios se debe situar una nave.
 2. **Generar rejilla simple:** Se seleccionará una sección del mapa y se creará un conjunto de carreteras que formen una cuadrícula.
 3. **Conectan ClingoSharp con IndustryLP:** Se incluirá el proyecto ClingoSharp como dependencia de IndustryLP y se procederá a hacer una generación sencilla.
 4. **Funcionalidades extra:** Se añadirán a la selección de la región operaciones de movimiento de la región, cambio de tamaño y rotación de la misma.
 5. **Añadir restricciones/preferencias:** Para cada parcela, el usuario podrá proponer un edificio o prohibir la generación del mismo.
 6. **Establecer plantillas:** Se incluirá una lista de las distintas topologías de carreteras que el usuario podrá seleccionar, y que se usarán en vez de la cuadrícula.
 7. **Construir polígono:** Una vez seleccionado la solución de Clingo, el sistema deberá construir objetos servibles dentro del entorno de Cities: Skylines©.
2. **Interfaz C# con Clingo:** El resultado de esta épica es crear un producto final que sea una librería que actúe como binding de Clingo para C#, que será usado por el mod creado en Cities: Skylines©:
8. **Generar biblioteca dinámica:** Como Clingo está escrito en C/C++, la idea es que se pueda crear una biblioteca dinámica nativa (.dll en Microsoft Windows, .so en GNU/Linux) que se pueda usar para ClingoSharp.
 9. **Cargar programa lógico sencillo:** Se deberían crear los módulos básico para poder invocar Clingo desde un programa en C# y permitir que cargue un programa lógico.

10. **Conectan ClingoSharp con IndustryLP:** Se incluirá el proyecto ClingoSharp como dependencia de IndustryLP y se procederá a hacer una generación sencilla.
 11. **Completar módulo Control:** El objetivo es poder leer cada uno de los átomos resultantes de un modelo estable. Además poder incorporar llamadas de forma asíncrona a Clingo.
 12. **Implementar módulo de modelos estables:** El objetivo es poder leer cada uno de los átomos resultantes de un modelo estable. Además poder incorporar llamadas de forma asíncrona a Clingo.
 13. **Añadir tests unitarios:** Se definirán los tests básico para las funcionalidades principales de Clingo.
3. **Documentación:** El objetivo de esta épica es la de crear las diferentes documentaciones necesarias del proyecto. Se divide en:
14. **Memoria del proyecto:** Crear este documento que tendrá toda la información del proyecto.
 15. **Documentación del IndustryLP:** Generará los diagramas y la documentación de código necesaria.
 16. **Documentación de ClingoSharp:** Se crearán los diagramas y documentación del código necesaria.

Considerando las tareas descritas anteriormente, en la Tabla 3.1 podemos ver la dedicación en horas llevada a cabo para cada proyecto. Debido a que no se ha realizado una estimación antes de la realización, no tenemos un dato previsto, si no que es directamente el valor final tangible.

Coste y recursos

Para calcular el coste de los recursos humanos que ha supuesto este proyecto, aparte de las 363 horas totales se ha añadido un 10 % a mayores que incluye las reuniones de seguimiento por parte del director del proyecto.

Tarea	Tiempo (h)
01	30
02	6
03	30
04	30
05	40
06	45
07	15
08	6
09	15
10	30
11	30
12	15
13	15
14	40
15	8
16	8
Total	363

Tabla 3.1: Desglose de la dedicación a las historias de usuario.

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. Eso suma 37 horas al proyecto, dando un total de 400 horas total. Teniendo esto en cuenta, y siguiendo la Guía Salarial del Sector TI en Galicia [5], se han estimado los costes que se pueden ver en la Tabla 3.2.

Por otro lado, en la 3.3 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

Recurso	Tiempo (h)	Coste (€/h)	Total
Doctor Investigador	37	24,00	888,00
Analista Programador	363	9,00	3267,00
Total	400		4155,00

Tabla 3.2: Coste de los recursos humanos del proyecto.

Recurso	Vida (mes)	Uso (mes)	Coste (€)	Total (€)
Portátil	48	10	800,00	166,67
Cities: Skylines©	∞	10	27,99	27,99
Clingo	∞	10	-	0
.NET Framework	∞	10	-	0
Visual Studio	∞	10	-	0
Github	∞	10	-	0
LaTeX	∞	10	-	0
Total				194,66

Tabla 3.3: Coste de los recursos humanos del proyecto.

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 4349,66€ en total.

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 polígono industrial que sea legible por el videojuego Cities: Skylines©.

- Permitir añadir restricciones sobre ciertas zonas del mapa.
- Poder mostrar los resultados de la generación dentro del videojuego.

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

Este proyecto se ha dividido en dos tipos de arquitectura dada la naturaleza del problema. Esto es debido a que estamos implementando diferentes paradigmas, y cada una tiene su forma de diseñarse.

Como primera parte vamos a analizar la arquitectura del software que se ejecuta dentro de Cities: Skylines©. Tal y como se puede ver en la Figura

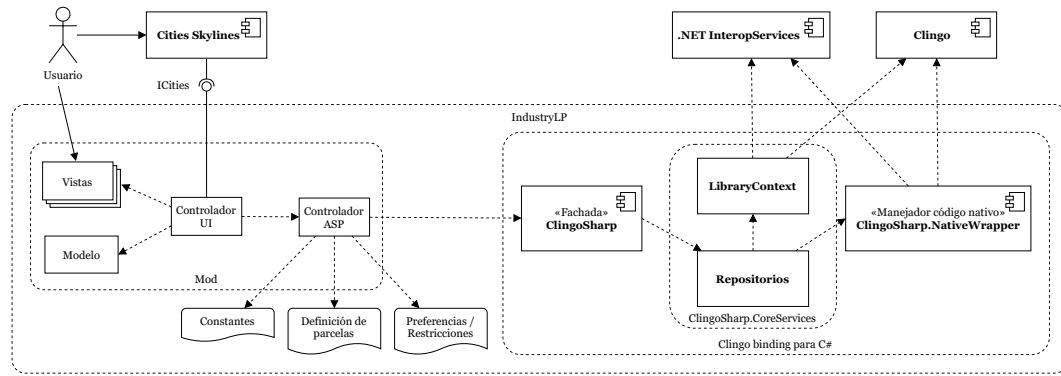


Figura 3.3: Arquitectura del mod de Cities: Skylines©

3.3, este se ha dividido en dos capas para permitir sus reusabilidad en distintos proyectos. En la primera capa tendríamos la interfaz del mod en si mismo. Este se basa en una arquitectura Modelo-Vista-Controlador (MVC), en donde el controlador se encargará de reaccionar antes los eventos de las vistas, que son la parte que el usuario manipula, y modificar el modelo, que son los datos que se guardan en memoria en este caso.

Además, el controlador de la interfaz se conectará con el controlador del modulo ASP para hacer llamadas a Clingo y obtener el resultado de las preferencias, tal y como se explica en la Sección 3.4.3. Este módulo a su vez llama a la segunda capa, que se corresponde con un *binding* o adaptación de la API en C de Clingo para el lenguaje de programación C#. Este binding usa como referencia final la API en Python de Clingo¹, la cual está dividida en módulos. Es por eso que nos hemos basado en una arquitectura de repositorios, en donde hay una fachada que llama a cada una los distintos componentes, que contienen como contexto la referencia a la biblioteca de Clingo, y los cuales a su vez llaman a otro componente que se encarga de hacer la llamada a la API en C de Clingo.

En la segunda parte analizaremos la arquitectura del programa lógico. Como ya comentamos en la Sección 2.2.1, para resolver un programa lógico nece-

¹<https://potassco.org/clingo/python-api/5.4>

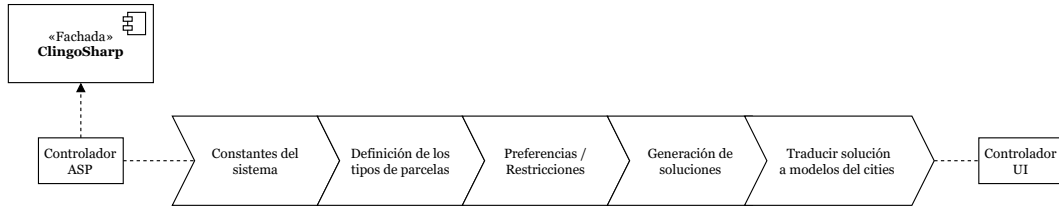


Figura 3.4: Arquitectura del programa lógico

sitamos transformarlo a su equivalente con variables libres mediante un grounder, y obtener los modelos estables mediante un solver. Debido a que es un proceso de transformación se ha ideado el usar una arquitectura en *pipeline*, en donde la salida de cada proceso es la entrada del siguiente. Tal y como se puede ver en la Figura 3.4, se ha dividido en una etapa de definición de constantes del sistema lógico, una etapa de definición del los tipos de parcelas (obteniendo previamente la información de los modelos existentes en Cities: Skylines©), una etapa de preferencias o restricciones del tipo o posición de las parcelas, una etapa de obtención de los modelos estables, y una última etapa de traducción de cada átomo del modelo estable a un modelo que pueda comprender Cities: Skylines©. Cada una de las etapas se explicarán en la Sección 3.4.3.

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. Estos requisitos funcionales se transforman, por tanto, en los casos de uso del sistema que se proponen en la Tabla ??

3.4.3. 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 polígonos industriales en formato declarativo. Para ello usamos el paradigma de programación lógica, que usa la tecnología Answer Set Programming, tal y como se expone en la

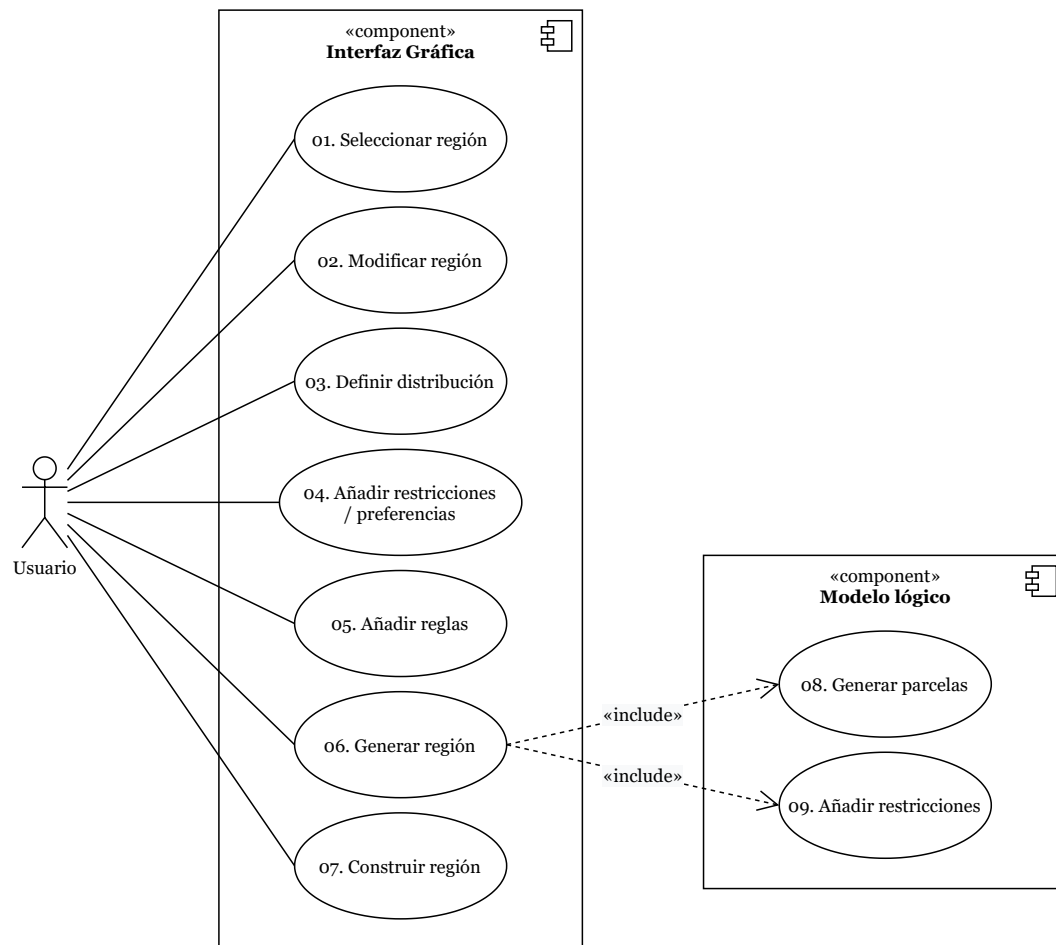


Figura 3.5: Diagrama de casos de uso del sistema propuesto

CU	Nombre	Descripción
01	Seleccionar región	El usuario puede pinchar en cualquier zona del mapa de Cities: Skylines© y marcar una región.
02	Modificar región	Una vez marcada una región, el usuario puede cambiar el tamaño y la rotación de esta mediante movimiento del ratón.
03	Definir distribución	Una vez establecida la región, el usuario puede indicar que tipo de topología de carreteras se va a generar en la región.

Tabla 3.4: Descripción de casos de uso 1

Sección 2.2.1.

CU	Nombre	Descripción
04	Añadir restricciones / preferencias	Dada una distribución, el usuario puede marcar que edificios prefiere generar y cuales quiere descartar de la generación.
05	Añadir reglas	Antes de empezar a generar la región, el usuario puede modificar de forma opcional la generación indicando nuevas reglas escribiendolas en un campo de texto.
06	Generar región	El sistema deberá generar una región válida y mostrarla. Para eso se apoyará de los casos de uso 08 y 09.
07	Construir región	Una vez seleccionada la región que el usuario quiere construir, se invocará a Cities: Skylines© para que cree los objetos necesarios en el juego para que se pueda usar por la IA.
08	Generar parcelas	El sistema deberá seleccionar un tipo edificio para cada tipo de parcela.
09	Añadir restricciones	Una vez seleccionado el tipo de edificio, el sistema añadirá las restricciones de cada parcela para que el sistema lógico genere una solución que satisfaga dichas restricciones.

Tabla 3.5: Descripción de casos de uso 2

Con esto, 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 polígono industrial. A continuación explicamos el diseño de cada una de las etapas que consta el sistema de generación.

Constantes del sistema lógico

Como primer paso para la generación del polígono industrial, definimos unos átomos que nos servirán para indicar el tipo de algunas variables. En concreto hemos definido el átomo `row(X)`, que contiene la definición de cada fila, y `column(X)` que es la contraparte para columnas. En el caso de los tipos de edificio disponibles, se ha creado el átomo `str_parcel(X)`, el cual guarda como string el nombre de cada uno de los edificios disponibles en Cities: Skylines®.

Definición de los tipos de parcela

Con las constantes ya definidas, se ha definido una regla para la generación de cada parcela `parcel(X, Y, B)` con una *choice rule*. La *choice rule* tiene cardinalidad 0 - 1, que se expresa como $\min \{p\} \max$ e indica que puede existir o no un átomo `parcel(X, Y, B)` dado un tipo de edificio `str_parcel(B)` para cada celda `row(X)`, `column(Y)`.

Con esto le decimos a Answer Set Programing que puede elegir cualquier combinación para cada parcela, incluyendo no generar una parcela de un tipo dato. Esto puede generar soluciones donde no existe parcelas, o donde para una parcela existen varias soluciones. Para restringir las soluciones a modelos estables en donde exista un solo átomo `parcel(X, Y, B)`, vamos a añadir un esquema de comprobación-restricción. Este esquema se basa en definir un átomo auxiliar que marque una comprobación concreta, y luego una restricción (una regla sin cabeza) para este átomo.

```
% Generacion %  
0 { parcel(X, Y, B) : str_parcel(B) } 1 :- row(X), column(Y).  
  
% Comprobacion %  
sell_parcel(X, Y) :- row(X), column(Y), str_parcel(B), parcel(X, Y  
    , B).  
  
% Restriccion %  
:- row(X), column(Y), not sell_parcel(X, Y).
```

```
:- parcel(X, Y, S1), parcel(X, Y, S2), str_parcel(S1), str_parcel(
    S2), row(X), column(Y), S1 != S2.
```

Listado 3.1: Código para la generación de parcelas

En nuestro caso usaremos como átomo auxiliar `sell_parcel(X, Y)`, que nos indicará que existe una parcela en una posición, y como restricciones definiremos una regla para que no existan celdas sin el átomo `sell_parcel(X, Y)`, y otra regla para que no existan celdas con tipos de edificio distintos.

Preferencias/Restricciones de parcelas

Con la definición de cada parcela a una posición dada hecha, añadiremos más reglas para delimitar la generación de los modelos estables a las preferencias del usuario. Para ello, desde la interfaz podemos indicar si queremos que un edificio se genere en una posición dada, o si queremos que un edificio no se genere nunca en una posición.

```
% Preferencia de colocacion de parcelas
parcel(1, 1, "Cargoyard").
parcel(2, 3, "Wind turbine").
parcel(2, 2, "Cargoyard").
parcel(3, 1, "General Factory 3x2").
...

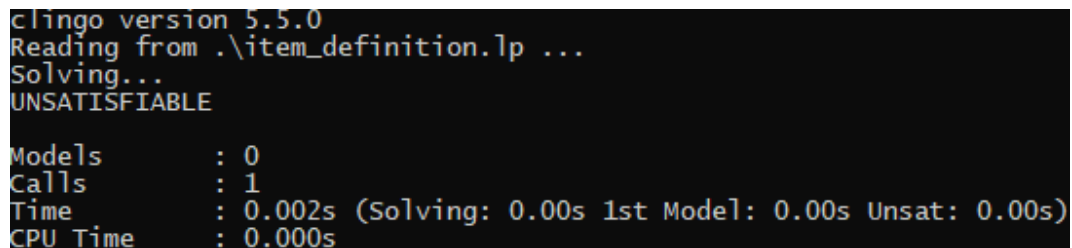
% Restricciones del tipo de edificio
:- parcel(4, 1, "General Factory 2x2").
:- parcel(1, 2, "Forestry 4x4").
:- parcel(2, 2, "Forestry 4x3").
```

Listado 3.2: Preferencias de parcelas

En la primera parte vamos a traducir las preferencias a hecho, que son los propio átomo, o también se pueden ver como reglas sin cola y actúan como valores de verdad dentro del modelo lógico. En el segundo caso, se traducirán a restricciones o reglas sin cabeza, tal y como vimos en la sección anterior. En

el Listado 3.2 vemos ejemplos de estas preferencias.

Uno de los resultados lógicos que podemos tener aquí es que el usuario ponga tanto como preferencia como restricción el mismo tipo de edificio. Aquí el resultado que nos devuelve clingo es que el modelo es insatisfactible, es decir, no existe ningún modelo estable que cumpla con el programa lógico, tal y como se muestra en la Figura 3.6.



```
clingo version 5.5.0
Reading from .\item_definition.lp ...
Solving...
UNSATISFIABLE

Models      : 0
Calls       : 1
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
```

Figura 3.6: Ejemplo de un programa lógico sin solución

Otras formas de delimitar la generación es mediante las distancias y la topología de la generación. Esto nos permite hacer reglas más complejas en donde indiquemos restricciones más avanzadas, como que no queremos que dos tipos de edificios estén cerca, por ejemplo en el caso de que sea industria contaminante con industria verde. Para ello podemos usar algún algoritmo, como distancia de manhattan (tal y como se muestra en el Listado ??) o k-vecino más cercano.

```
distance(S1, S2, |X1-X2|+|Y1-Y2|) :- parcel(X1, Y1, S1), parcel(X2
    , Y2, S2), X1 != X2, Y1 != Y2.
neighbour(S1, S2) :- distance(S1, S2, 1).
```

Listado 3.3: Cálculo de la distancia del taxista en Answer Set Programming

Para delimitar la generación con este tipo de reglas, podemos usar los átomos que acabamos de crear como una estrategia de *divide y vencerás*, introduciendo estos átomos en hechos o en prohibiciones. Esto se ve reflejado en el Listado

```
neighbour("Forestry 3x3", "Forestry 3x2").
neighbour("Forestry 3x4", "Forestry 3x2").
```

```
neighbour("Forestry 3x3", "Forestry 3x2").
```

```
:- distance("Forestry 3x3", "Farm Village", D), D < 4.
```

```
:- distance("Forestry 3x4", "Farm Village", D), D < 4.
```

Listado 3.4: Prohibiciones de generación de edificios

3.4.4. Implementación

Una vez analizado el proyecto en cuestión, y explicado como funciona el programa lógico, pasaremos a detallar el diseño de los componentes principales de los sistemas creados para usar en Cities: Skylines®.

Primeramente empezaremos explicando como funcionan las diferentes acciones en IndustryLP. Pensando en su reusabilidad, y que en un futuro el número de acciones pueda crecer, se ha pensado en implementarlas usando un patrón estrategia. Tal y como se muestra en la Figura 3.7, cada acción deriva de la clase `ToolAction`.

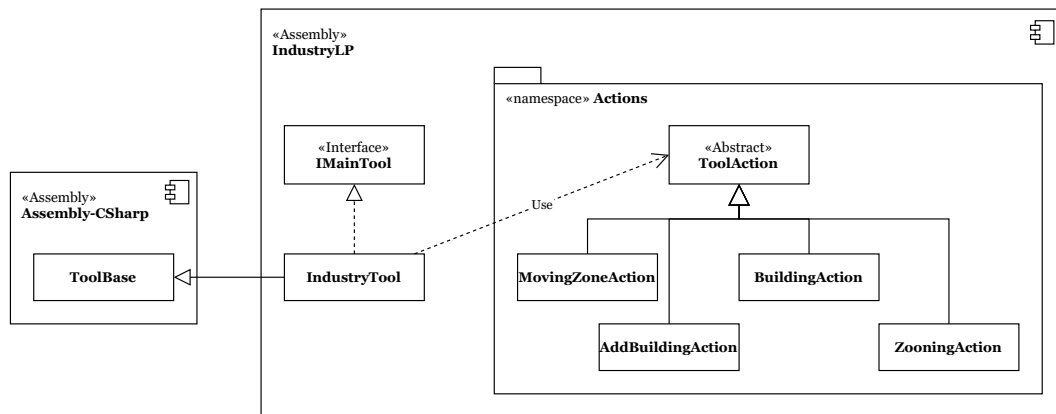


Figura 3.7: Acciones de la interfaz

Esta clase abstracta define una serie de métodos comunes que corresponden al ciclo de vida pensado para cada una de las acciones, representados en la Figura 3.8. El ciclo de vida de una acción es controlado por la clase `IndustryTool`, que a su vez deriva de `ToolBase`, una clase abstracta que nos proporciona la interfaz de Cities: Skylines® y que también tiene un ciclo de vida, pero que es controlado por el propio juego.

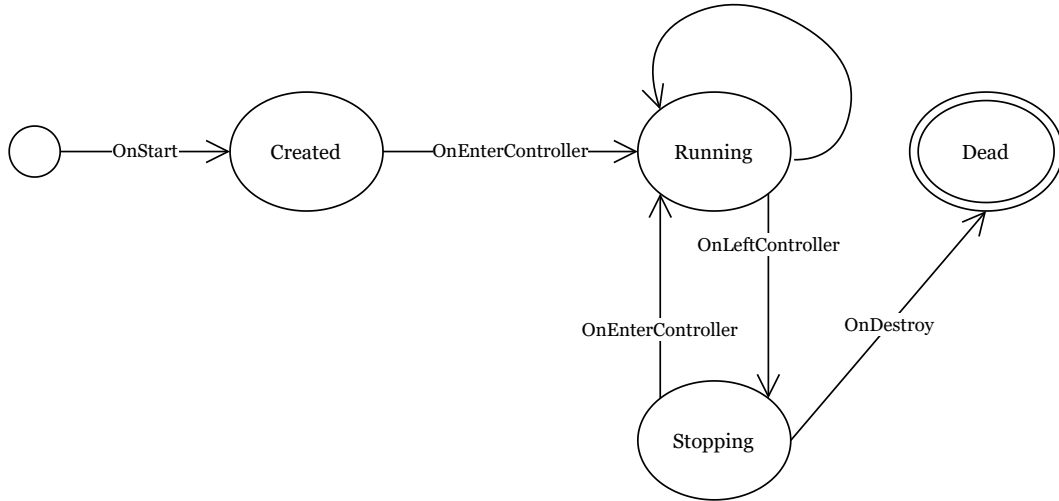


Figura 3.8: Ciclo de vida de una actividad

Es por eso que, dependiendo del estado en el que se encuentre el juego y la interfaz, la clase `IndustryTool` delegará en cada una de las acciones. Además, `IndustryTool` implementa la interfaz `IMainTool`, que define una serie de métodos que pueden usar las acciones para cambiar de estado o ejecutar otra acción, evitando así el acoplamiento entre `IndustryTool` y cada una de las acciones concretas.

Con respecto a la generación de distribuciones, pensando también en la reusabilidad y en la implementación de nuevas topologías de parques industriales en un futuro, también se optó por usar un patrón de diseño, en este caso de *Template Method*. El diseño de la generación de distribuciones se muestra en la Figura 3.9.

Aquí, existen dos clases abstractas: `DistributionThread`, que define los métodos para la ejecución del algoritmo de generación de la distribución, el cual a diferencia del problema principal, este está implementado de forma imperativa; y `DistributionInfo`, que contiene la información resultante de la generación del algoritmo, y que define métodos de navegación de parcelas según la distribución.

En el caso del sistema que se encarga de conectar y ejecutar la bibliote-

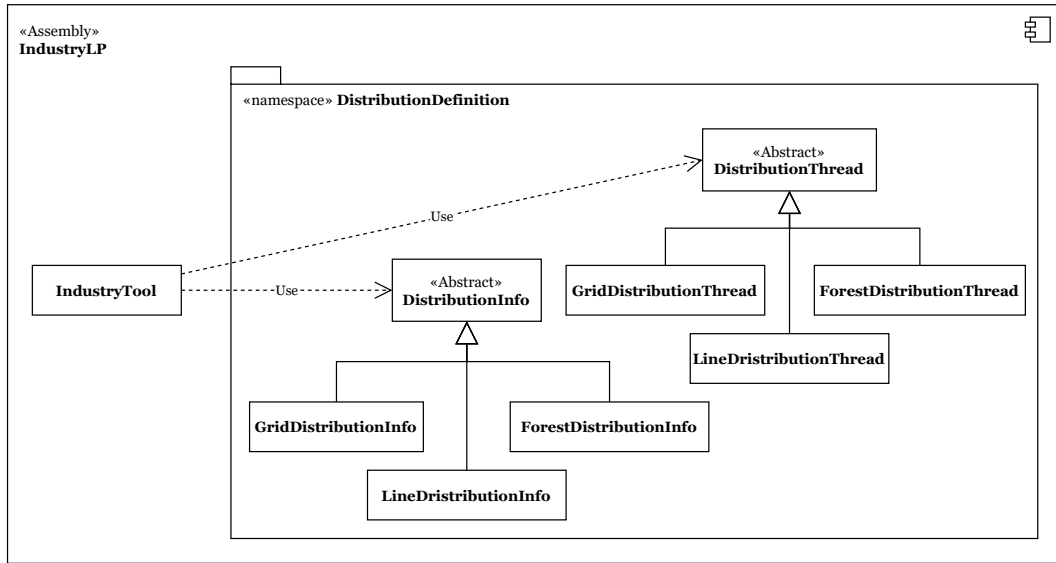


Figura 3.9: Definición de las distribuciones

ca de Clingo, se ha optado por una arquitectura de tipo repositorio, tal y como se muestra en la Figura 3.10. Cada módulo de Clingo conecta con un módulo del componente *ClingoSharp.NativeWrapper*, que se encargar de enlazar dinámicamente las funciones de la API en C con métodos internos de cada módulo. Estos módulos a su vez exponen funciones con tipos de datos manejados por C# mediante interfaces, que derivan de una interfaz general llamada *IClingoModule*.

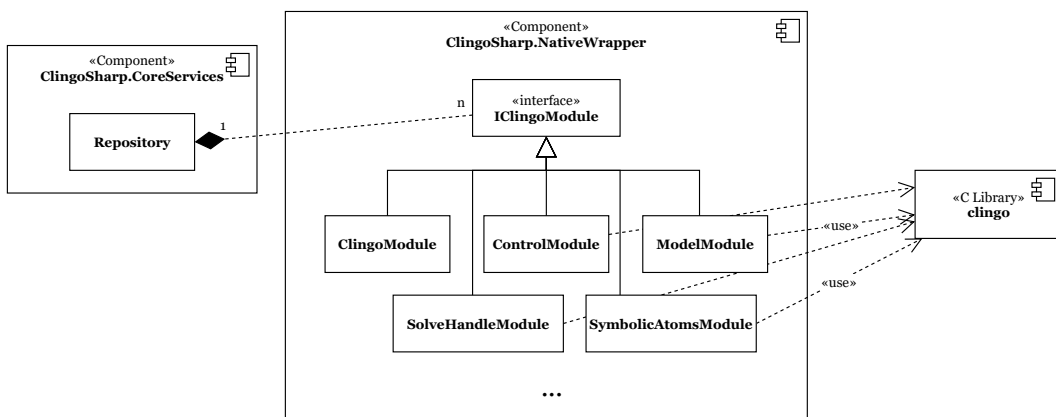


Figura 3.10: Arquitectura de ClingoSharp

En el caso del componente *ClingoSharp.CoreService*, este implementa la clase *Repository*, que contiene una composición de cada uno de los módu-

los. Esto es debido a que antes se pasa a memoria la biblioteca en Clingo, haciendo uso de los servicios disponibles en cada sistema operativo y de .NET Framework. Una vez realizada la carga en memoria, se procede a cargar dinámicamente cada uno de los módulos marcados con la interfaz `IClingoModule` del componente *ClingoSharp.NativeWrapper*, y que finalmente son expuestos por nuestro repositorio.

En la Figura 3.11 se muestran las dependencias de cada uno de los distintos módulos. A pesar de lo complicado que parece, la arquitectura en repositorio nos permite abstraer la parte de enlazado de nuestro programa con la biblioteca nativa de Clingo, del enlazado de cada uno de los diferentes punto de control de la biblioteca, además de exponer una API limpia de Clingo en C#. Otro punto a favor de esta arquitectura es que también actúa como wrapper, permitiendo cambiar entre distintas versiones de la biblioteca nativa de Clingo sin cambiar nuestro código, así como de implementar otra biblioteca de Answer Set Programming en C#.

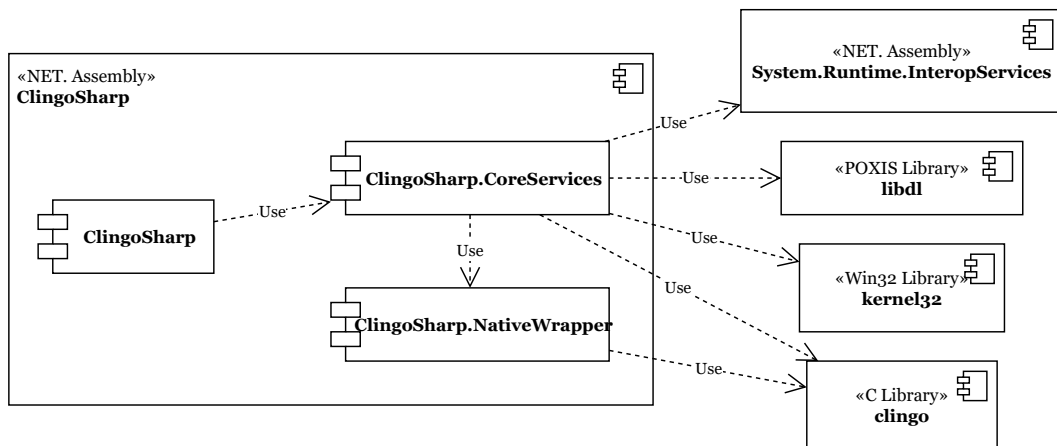


Figura 3.11: Dependencias de ClingoSharp

Para finalizar, cabe destacar que cada una de las capas (IndustryLP y ClingoSharp) se ejecuta en paralelo, pudiendo el primero enviar mensajes a ClingoSharp sin causar que la interfaz se congele, además de servir como recolector de errores que se puedan originar en nuestro programa lógico sin afectar a Cities: Skylines©.

Capítulo 4

Conclusiones

TODO

Índice de tablas

3.1. Desglose de la dedicación a las historias de usuario.	27
3.2. Coste de los recursos humanos del proyecto.	28
3.3. Coste de los recursos humanos del proyecto.	28
3.4. Descripción de casos de uso 1	32
3.5. Descripción de casos de uso 2	33

Índice de figuras

2.1. Pantalla del juego SimCity™	10
2.2. Recreación de A Coruña en Cities: Skylines©	11
2.3. Ejemplo de ejecución de Answer Set Programing	13
2.4. Arquitectura de .NET Framework	14
3.1. Ejemplo de definición de zonas. En la azul las zonas comerciales y en verde las residenciales.	20
3.2. Interfaz de gestión de recursos de la expansión <i>Industries</i>	22
3.3. Arquitectura del mod de Cities: Skylines©	30
3.4. Arquitectura del programa lógico	31
3.5. Diagrama de casos de uso del sistema propuesto	32
3.6. Ejemplo de un programa lógico sin solución	36
3.7. Acciones de la interfaz	37
3.8. Ciclo de vida de una actividad	38
3.9. Definición de las distribuciones	39
3.10. Arquitectura de ClingoSharp	39
3.11. Dependencias de ClingoSharp	40

Apéndice A

Manual de uso

En esta sección mostraremos un pequeño manual de uso del mod creado para Cities: Skylines©, así como su instalación.

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] S. L. Fontán, C. L. García de Leaniz, y R. G. García, *Guía de recomendaciones: Técnicas para el proyecto integral de parques empresariales en Galicia*, 1º ed., ser. 1. Xunta de Galicia. Instituto galego da vivenda, 2009.
- [6] 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>

- [7] A. M. Smith y M. Mateas, “Answer set programming for procedural content generation: A design space approach,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, n.^o 3, pp. 187–200, 2011.
- [8] R. Alcalde Azpiazu, “Scenario generation for a 2d videogame using logic programming,” Universidade da Coruña, 2018. [Online]. Disponible en: <https://github.com/NEKERAFA/FreeCiv-Editor/blob/master/docs/report/AlcaldeAzpiazu.Rafael.TFG.2018.pdf>
- [9] A. M. Smith y M. Mateas, “Answer set programming for procedural content generation: A design space approach,” vol. 3, pp. 187 – 200, 10 2011.
- [10] 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
- [11] G. Boenn, M. Brain, M. D. Vos, y J. ffitch, “Automatic music composition using answer set programming,” *CoRR*, vol. abs/1006.4948, 2010. [Online]. Disponible en: <http://arxiv.org/abs/1006.4948>
- [12] 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.
- [13] 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>
- [14] 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>

- [15] K. Schwaber y M. Beedle, *Agile Software Development with Scrum*, 1^o ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.
- [16] K. Schwaber, *Agile Project Management With Scrum*. Redmond, WA, USA: Microsoft Press, 2004.