

Trabajo final de grado

GRADO DE INFORMÁTICA

Facultad de Matemáticas
Universidad de Barcelona

**Modelado y visualización de
mercados distribuidos de energía**

Autor: Martin Azpillaga Aldalur

Director: Dr. Jesús Cerquides
Dr. Juan Antonio Rodríguez-Aguilar

Realizado en: Consejo superior de investigación científica
Barcelona, 21 de enero de 2016

Índice

1. Introducción y antecedentes	1
2. Objetivos	1
2.1. Funcionalidad	1
2.2. Requerimientos	1
2.3. Casos de uso	2
2.4. A continuación	2
3. Metodología y herramientas	3
3.1. Agile	3
3.2. Decisiones de diseño	4
3.3. Control de versiones: Git	4
3.4. Documentación: LaTeX	4
3.5. Modelado 3D: Blender	5
3.6. Entorno de visualización: Unity	6
3.7. Algoritmia: Java	6
3.8. Intercambio de datos: JSON	7
3.9. Trazado de gráficas: Gnuplot	7
3.10. A continuación	7
4. Especificación del mercado	8
4.1. Modelado de un mercado distribuido	8
4.2. Formato de entrada: Geolocalización de los prosumidores	8
4.3. Algoritmos para la creación de conectividad	9
4.3.1. Información completa: Steiner Trees	9
4.3.2. Información incompleta: Nearest neighbour problem	9
4.4. Visualización del mercado	10
4.5. Controles de usuario	10
4.5.1. hora	10
4.5.2. capacidad	10
4.5.3. perfil	10
4.6. Formato de salida: Mercado especificado	10
5. Simulación	11

5.1.	Principios de diseño	11
5.1.1.	Patrón MVC	11
5.1.2.	Patrón experto y la encapsulación	12
5.1.3.	Interficies y ampliación	12
5.2.	El paquete view	13
5.2.1.	Command Line Interface	13
5.2.2.	FileSystem	13
5.3.	El paquete control	14
5.3.1.	Main	14
5.3.2.	Simulation	14
5.4.	El paquete model.data	15
5.4.1.	ITemporalDistribution	15
5.4.2.	Data	15
5.5.	El paquete model.core	16
5.5.1.	Market	16
5.5.2.	Wire	16
5.5.3.	Weather	16
5.5.4.	Distributor	17
5.5.5.	Prosumer	17
5.6.	El paquete model.components	17
5.6.1.	IBiddingStrategy	17
5.6.2.	IBattery	17
5.6.3.	IGenerator	17
5.6.4.	IAppliance	17
5.7.	A continuación	17
6.	Visualización	18
6.1.	En este capítulo	18
6.2.	Filosofía de diseño de Unity	18
6.3.	Diseñando los paneles	19
6.4.	Control de la cámara	19
6.5.	Eligiendo la ciudad	20
6.6.	Seleccionando el perfil de las casas	20
6.7.	Seleccionando la franja horaria	20
6.8.	Ejecutando la animación	21

6.9. Mostrando la información relevante	21
6.10. Programando las animaciones	22
6.11. Controlando el flow	22
6.12. Parseando el JSON	23
6.13. A continuación	23
7. Futuro trabajo	24
7.1. Ampliar a Open Street Map, CityEngine.	24
7.2. OpenWeatherMap	24
7.3. Reinforcement learning en bidding	24

1. Introducción y antecedentes

2. Objetivos

2.1. Funcionalidad

El objetivo principal del proyecto es construir un software que permita visualizar simulaciones hechas sobre mercados distribuidos de energía. El software deberá cumplir con los requerimientos no funcionales especificados en 2.1 y posibilitar los casos de uso establecidos en 2.2.

2.2. Requerimientos

La aplicación debe ser usable, mantenible y libremente distribuible.

■ Requerimientos de Usabilidad

- La aplicación se ejecutará en un **entorno tridimensional interactivo** para resaltar la naturaleza tridimensional del problema.
- El usuario podrá interactuar con el programa mediante **controles minimalistas e intuitivos** para una rápida y fluida ejecución del programa.
- El usuario podrá **moverse libremente** por el entorno y **revelar o ocultar** las distintas partes de información de manera que pueda personalizar la información que percibe en cada momento.

■ Requerimientos de Mantenibilidad

Desde el comienzo del proyecto, se ha tenido presente la posibilidad de seguir trabajando en el proyecto después de la entrega ante los tribunales. Por ello, se ha hecho especial incapié en:

- Se seguirán **patrones de diseño** standares a la hora de diseñar e implementar el software de manera que el código sea fácilmente modificable.
- El programa estará **documentado** mediante **diagramas** que mejoren la comprensión del proyecto y una **memoria** que sirva como consulta sobre funcionalidades concretas de cada parte del proyecto.
- El programa contará con **mecanismos de ampliación** que permitan facilitar futuras modificaciones.

■ Requerimientos sobre la Distribución

- Todas las incorporaciones externas deberán contener licencias **gratuitas y libremente distribuibles** para mantener el proyecto asequible y accesible para todo usuario interesado en él.

- El producto final será **multiplataforma** de manera que se pueda ejecutar en los sistemas operativos más predominantes del mercado.
- El producto final será un **ejecutable autocontenido** de manera que no requiera configuración alguna para ser ejecutado.

2.3. Casos de uso

El proyecto debe contener tres partes: Especificación del mercado, simulación y visualización, siendo cada una ejecutable independientemente del resto. Las entradas y salidas de cada parte serán un único fichero formateado acorde a toda la información relevante. A continuación resumimos la funcionalidad y la interacción del usuario de cada parte:

Especificación del mercado

Se trata de una aplicación interactiva en el cual se genera un mercado a partir de la geolocalización de los prosumers proporcionada por el usuario. A partir de ese momento, el usuario podrá definir distintos parámetros sobre la simulación a ejecutar sobre el mercado. Se generará un fichero formateado que contenga todos los parámetros de la simulación.

Simulación del mercado:

Se generan y resuelven iterativamente problemas de CEAP modificando el estado de los datos a cada paso. Recibe los parámetros elegidos por el usuario y genera un archivo con los resultados de la simulación. El usuario no interviene en la ejecución del simulador.

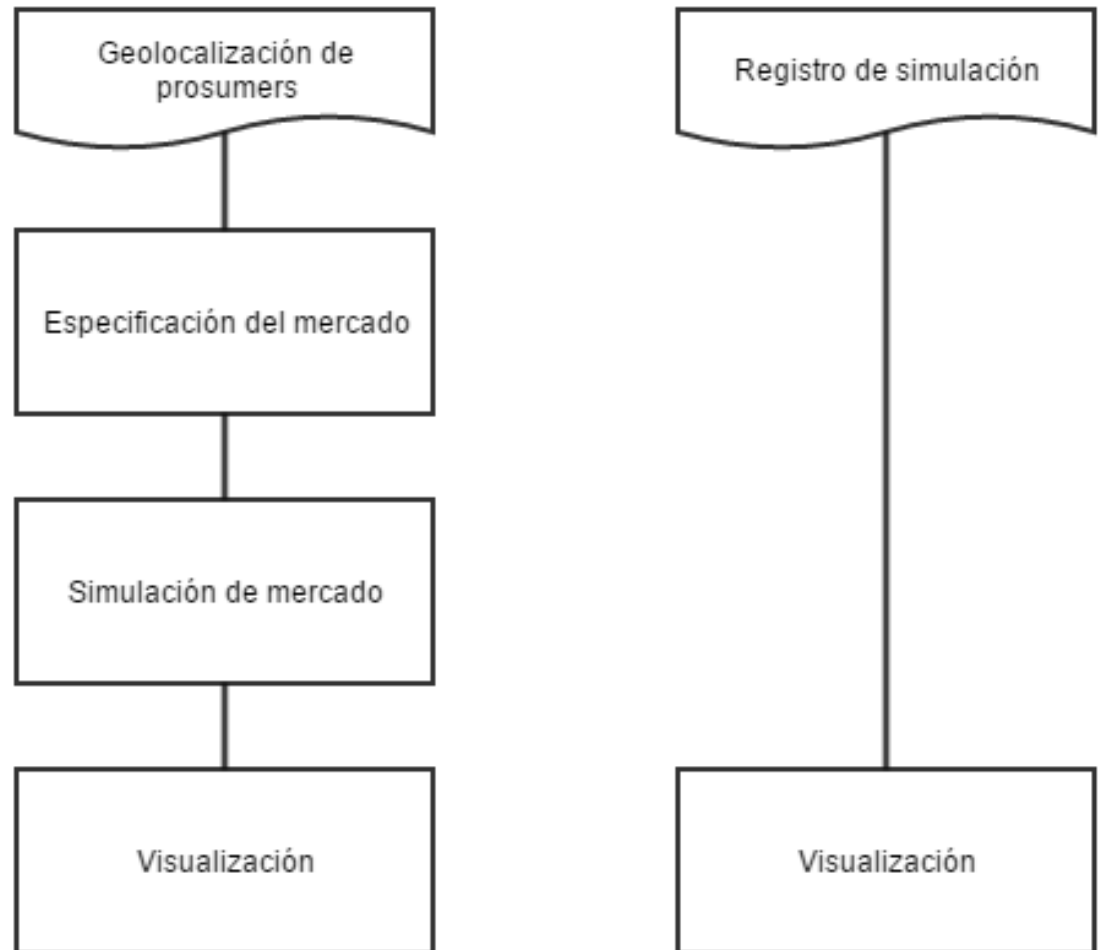
Visualización del mercado: Genera una visualización a partir de los resultados obtenidos. El usuario puede interactuar con la información relevante y controlar la reproducción de los resultados.

Diagrama de flujo general:

El producto final contiene todas las partes y proporciona dos modos de ejecución: Por una parte, debe permitir ejecutar todo el proceso de manera secuencial, por otra parte debe ser capaz de visualizar una simulación a partir de un registro de simulación creado externamente.

2.4. A continuación

Explicamos la metodología y herramientas utilizadas a lo largo del proyecto, razonando su elección ante las alternativas.



3. Metodología y herramientas

Explicaremos los programas y lenguajes más importantes usados durante todo el proyecto, razonaremos su uso ante sus equivalentes y citaremos sus autores así como la licencia bajo la que se distribuye cada una de ellas. Se busca que los programas elegidos sean libres y gratuitos y que preferentemente tengan la posibilidad de ser multiplataforma para mantener el proyecto asequible y accesible para cada usuario interesado en él.

3.1. Agile

Iteraciones y features.

3.2. Decisiones de diseño

filosofía de java y unity.

3.3. Control de versiones: Git

Todo proyecto informático que supere un mes de tiempo de producción requiere de un software de control de versiones. Son programas pensados para ayudar a los programadores a compartir código entre ellos e ir guardando versiones del proyecto, para poder volver a ellos ante un error inminente, sin que los programadores tengan que preocuparse de guardar versiones del proyecto localmente.

Los sistemas de control de versiones más utilizados actualmente son Git, Mercurial y SVN. Dado que todos cumplen las expectativas de ser gratuitos y multiplataforma, nos decantamos por Git, diseñado por Linus Torvalds y distribuido bajo la licencia

Bitbucket

Los proyectos de Git usualmente utilizan un repositorio on-line donde ir guardando la versión definitiva del proyecto en cada momento. Para ello es necesario encontrar un servicio de alojamiento de proyectos git en la web. Posiblemente el más reconocido sea GitHub, que es gratuito siempre y cuando se publique el código de forma abierta. También ofrece repositorios de pago con un gasto mensual. Por otro lado, Bitbucket es una alternativa que ofrece también la posibilidad de tener repositorios privados gratuitamente, siempre que no se supere el límite de 5 usuarios interventores del proyecto. Dado que la parte de RadPro se alojaba en un repositorio de BitBucket privado, se ha seguido la misma iniciativa en este proyecto.

3.4. Documentación: LaTeX

Así como un sistema de control de versiones, mantener una documentación clara del proyecto es esencial para su éxito y sobre todo para el futuro si alguien decide retomar el proyecto. Existen maneras específicas de documentar programas informáticos, como el JavaDoc de Java, pero debido a la naturaleza multiX del proyecto, se ha decidido usar una documentación común para todo el proyecto.

Los Typesetting systems son sistemas pensados para escribir un documento de manera formateada y ordenada. El más conocido entre ellos sea probablemente TeX. Creado en 1974 por Donald Knuth, ha sido utilizado en la escritura de múltiples documentos científicos.

A partir de TeX han salido derivados como LyX y LaTeX que pretenden facilitar su uso, así como presentar nuevas funcionalidades más específicas. Por otro lado, desde el año 2000 existe el proyecto NTS, de New Typesetting System, una reimplementación de TeX en Java, con la intención de proporcionar las ventajas que este lenguaje aporta como la facilidad para multiplataforma.

En este proyecto, se ha decidido usar LaTeX como typesetting system por su

robustez y previa experiencia del autor con el mismo. El resultado se encuentra en este documento, que pretende ser una guía para entender las ideas y razones principales detrás del diseño del programa.

LaTeX es software libre bajo licencia LPPL.

MiKTeX

MiKTeX es una distribución TeX/LaTeX para Microsoft Windows que fue desarrollada por Christian Schenk. Las características más apreciables de MiKTeX son su habilidad de actualizarse por sí mismo descargando nuevas versiones de componentes y paquetes instalados previamente, y su fácil proceso de instalación.

La versión actual de MiKTeX es 2.9 y está disponible en su página oficial bajo la licencia FSF/Debian.

Sublime Text

Por último, necesitamos un editor de texto donde escribir el documento de LaTeX. Existen editores propiamente creados para ello como TeXnicCenter y TeX-Maker, pero nos hemos decantado por el uso de Sublime Text. Sublime Text es un editor de texto plano genérico que aporta la posibilidad de añadir resalte de sintaxis, posibilidad de builds propias, snippets que agilizan el desarrollo del documento mediante plugins, por lo que resulta más cómodo de usar una vez se haya completado la configuración inicial.

Sublime Text se distribuye de forma gratuita, sin embargo no es software libre o de código abierto, se puede obtener una licencia para su uso ilimitado, pero el no disponer de ésta no genera ninguna limitación más allá de una alerta cada cierto tiempo.

3.5. Modelado 3D: Blender

Antes de que podamos simular una red eléctrica distribuida, deberemos crearla, y parte de ello consiste en modelar la malla 3D que representará el conjunto de edificios. Para ello necesitamos un programa de modelado 3D.

El mercado ofrece varias alternativas para este propósito. Desde centrados en arquitectura e ingeniería como AutoCad hasta modelado a través de escultura como ZBrush. Nosotros buscamos un programa de espectro genérico, ya que aparte de las ciudades, queremos modelar los menús y los controles de usuario usando dicho programa. Dentro de este ámbito seguimos encontrando varias alternativas: 3DS Max, Maya, Lightwave, Blender...

Nos decidimos por Blender por ser gratuito y multiplataforma, además de ser muy completo y ofrecer todas las funcionalidades que buscamos. Blender fue inicialmente distribuido de forma gratuita pero sin el código fuente, con un manual disponible para la venta, aunque posteriormente pasó a ser software libre.

3.6. Entorno de visualización: Unity

Unity es un motor de juegos, no está pensado para ser un entorno dedicado a la visualización como y Sin embargo, un juego no es más que una visualización interactiva compleja. Por ello, Unity ofrece un entorno en el que mostrar e interactuar con nuestros objetos tridimensionales de forma totalmente programable. Además cuenta con la ventaja de poder exportar el proyecto fácilmente a distintas plataformas como Windows, Mac y Linux, pero también Web, Android y iPhone OS.

Creado por Unity Technologies, Unity está disponible en dos versiones: Unity (totalmente gratuita) y Unity Pro que contiene más funcionalidades preimplementadas.

C# y Microsoft Visual Studio

Dentro de Unity los scripts pueden ser programados en tres posibles lenguajes de programación: C# Javascript, Boo.

Se ha elegido C# por su similitud a Java, el otro lenguaje importante de esta aplicación, y ventajas que ofrece ante los demás lenguajes de scripting, como ser fuertemente tipado (útil en proyectos de medida considerable) y la posibilidad de utilizar como editor Microsoft Visual Studio, un editor muy completo diseñado por Microsoft que cuenta, entre otras, con una licencia gratuita dirigida para uso de aplicaciones no empresariales.

3.7. Algoritmia: Java

Nos queda por cubrir la etapa de simulación. Podría usarse cualquier lenguaje de programación de propósito general para esta labor, pero dado que la base del simulador (radPro) ya está implementado en Java se ha decidido seguir usando el mismo lenguaje.

Además java sigue la filosofía de posibilidad de multiplataforma y licencia gratuita que pretende cumplir este proyecto.

Como información, Java fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems.

Editor Netbeans

Dada la gran fama de Java existen múltiples IDEs dedicados. Cualquiera de Eclipse o Android Studio serviría para nuestro propósito, pero usaremos NetBeans por preferencia personal del autor, que es un entorno de desarrollo integrado libre, hecho principalmente para el lenguaje de programación Java. Es un producto libre y gratuito sin restricciones de uso.

3.8. Intercambio de datos: JSON

Desde el momento en el que intervienen múltiples programas y lenguajes de programación en un proyecto es indispensable definir un formato de intercambio de datos.

Actualmente, los dos formatos más utilizados son XML y JSON. Por ser mucho más simple e inteligible, elegiremos JSON, sin darle importancia a las diferencias de rendimiento que podría haber entre ellas.

La mejor manera de visualizar un fichero JSON es utilizando una extensión para tu navegador preferido e internet. En nuestro caso, utilizaremos JSONView, escrito por Benjamin Hollis para Mozilla Firefox.

Por otra parte, necesitamos librerías que nos faciliten la creación y extracción de datos desde un archivo JSON. Cada lenguaje tiene librerías propias para esta función. Usaremos Gson de Google para Java y JsonObject escrito por Matt Schoen de Defective Studios para Unity.

3.9. Trazado de gráficas: Gnuplot

Por último, tendremos la necesidad de representar ciertos datos en forma de gráficas, por lo que necesitamos el acceso a un programa dedicado a crear gráficas de funciones.

Inicialmente, se buscó una librería propia para Java como JFreeChart y GRAL, pero por su falta de flexibilidad se descartaron y se decidió utilizar Gnuplot, un programa completo pensado para generar gráficas de funciones y datos.

Gnuplot se creó en 1986 y es compatible con los sistemas operativos más populares además de ser distribuido gratuitamente bajo licencia de software libre.

3.10. A continuación

Empezaremos a explicar el diseño e ideas de implementación de cada parte principal del proyecto, empezando por la creación de un red eléctrica distribuida hasta la visualización de su simulación.

4. Especificación del mercado

En 4.1 se define el modelo de mercado y sus parámetros como árbol dirigido ampliado, en 4.2 se define el formato del fichero de entrada que contiene la geolocalización de los prosumers, en 4.3 se detallan algoritmos para crear un árbol a partir de nodos situados sobre un plano. En 4.4, se explica el proceso de visualización del mercado. En 4.5 se describe el funcionamiento de los controles de usuario y por último en 4.6 se especifica el formate de datos de salida.

4.1. Modelado de un mercado distribuido

Para visualizar una simulación, lo primero que necesitamos es una ciudad sobre la que hacer los cálculos. Aprovechando que la visualización será en 3D podemos usar como ciudad un modelo 3D. Dado que queremos permitir que la ciudad pueda ser elegida por el usuario en tiempo de ejecución, tendremos que crear un mecanismo que permita importar un modelo 3D desde un fichero.

Los grandes programas de modelado como Blender o 3DS Max permiten crear un objeto tridimensional para después exportarla a un fichero en una variada de formatos: max, 3ds, obj... Usaremos el formato Wavefront OBJ por su simplicidad y por ser de licencia abierta. Además, OBJ es un formato muy extendido, por lo que la gran mayoría de programas de modelado 3D como blender o 3ds max lo soportan.

Wavefront OBJ guarda la información en líneas, donde cada línea esta precedida por una cadena de caracteres que indican el tipo de la información que sigue. Por ejemplo `v 1 1 1` indica un vértice en el punto (1,1,1) mientras que `f 1 2 3` indica una cara triangular que une los tres primeros vértices. Una ventaja de Wavefront OBJ es la simplicidad en el que se pueden diferenciar multiples objetos dentro del mismo modelo. Nuestra ciudad será un único modelo que tenga por submodelo cada una de las casas/parcelas que requieran ser conectadas a la red.

4.2. Formato de entrada: Geolocalización de los prosumidores

Quiero agradecer a [\[Referencia\]](#) por donar a la comunidad un importador de OBJ especialmente diseñado para Unity. El script coge como entrada un string a un fichero y devuelve una componente de tipo Mesh de Unity. El problema es que este script considera que todos los vertices pertenecen al mismo objeto. El código se ha ampliado de manera que detecte las líneas de formato `o nombre;` así crear una Mesh para cada parcela. El resultado es un array de Meshes. Crearemos un GameObject por cada uno que tenga como Mesh filter este mesh y por nombre su nombre y un GameObject empty llamado city. Por último el modelo de ciudad se situa en la escena y se escala de manera que quepa en el terreno. Ya estamos listos para proveer una red eléctrica a esta ciudad.

4.3. Algoritmos para la creación de conectividad

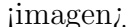
Hay muchos algoritmos que permiten unir todos los puntos dados en un espacio creando así un grafo. Recordamos que por limitaciones del RadPro nuestra red será un árbol, de manera que no podrá contener ningún ciclo. Dentro de los árboles podemos considerar dos posiciones: Creamos la red conociendo todos los puntos a unir o los puntos van añadiéndose a lo largo del tiempo y la red va ampliándose secuencialmente de manera que preserve la estructura de árbol.

4.3.1. Información completa: Steiner Trees

En el primer caso encontramos el conocido problema de Minimum Spanning Tree Problem, que trata de construir un camino que permita a un viajero pasar por todos los puntos recorriendo la mínima distancia posible. Sin embargo, en el entorno de nuestro problema, minimizar el tiempo o la distancia que ha de recorrer la electricidad para llegar de un extremo a otro no es relevante (ocurre en milésimas de segundo), sino que podemos plantearnos el siguiente problema:

Dado un conjunto de puntos P dentro de un espacio E , cual es la configuración que minimiza la distancia total de la red permitiendo crear nodos. Esto se traduciría en un menor coste de construcción, ya que se ahorraría en material y en tiempo requerido para montar la red. Este problema es conocido bajo el nombre de Steiner Tree Problem y ha sido muy estudiado por sus enormes aplicaciones en creación de redes. Los puntos añadidos se llaman Steiner points y cumplen ciertas propiedades curiosas como: Como máximo existen tantos steiner points como nodos iniciales menos dos. Cada steiner point es un nodo en el que se cruzan exactamente 3 caminos. Los tres caminos que intersectan en el punto se cortan en 120 grados entre sí.

Resulta que el problema es de complejidad NP-Hard y por lo que a partir de un número humilde de nodos (50 en un ordenador standard) la solución óptima es difícilmente alcanzable. Existen también varias maneras de aproximar steiner points, pero no he podido encontrar ninguna librería que los implemente más allá de artículos. El producto final contiene un jar llamado FindSteinerTree que recoge un listado de puntos en un archivo .txt y genera un .txt donde se guardan los steiner points y las conexiones entre los puntos.

Formato del txt: ;

4.3.2. Información incompleta: Nearest neighbour problem

en el caso anterior considerábamos que conocíamos la posición de todos los puntos que formarían el grafo. Sin embargo, si nos fijamos en la realidad, esta idea no sería práctica, ya que a lo largo del tiempo se van añadiendo y eliminando parcelas de manera que alteran los puntos del problema. Para crear un árbol en este caso usaremos el algoritmo Nearest Neighbour. Este algoritmo recorre cada uno de los puntos secuencialmente y une el punto con el punto más cercano a menos que ese punto ya esté unido a él. Como tal tiene una complejidad cuadrática respecto a la

cantidad de nodos y es resoluble hasta para un grafo enorme sin problemas.

Este algoritmo puede adaptarse correctamente a un entorno totalmente secuencial en el que los puntos van añadiéndose de forma continuada y a cada paso la red es un árbol completo.

Implementación en pseudocódigo: ¡código!

4.4. Visualización del mercado

4.5. Controles de usuario

4.5.1. hora

4.5.2. capacidad

4.5.3. perfil

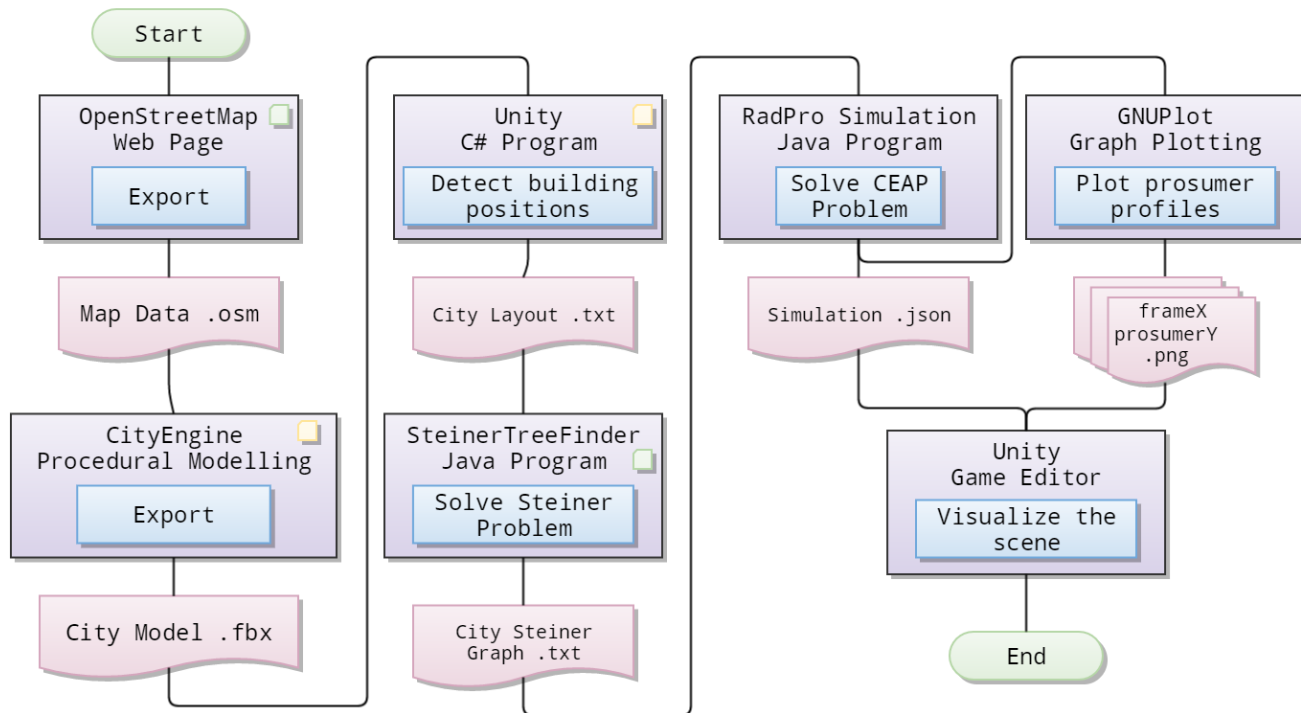
4.6. Formato de salida: Mercado especificado

Se utilizará un documento json para exportar los datos recopilados durante la especificación del mercado. El formato es el siguiente: imagen

Ahora somos capaces de crear modelos 3D, bien desde mapas reales o usando un software de modelado, guardarlos en un fichero obj, abrirlos desde una aplicación y crearles una red eléctrica a medida. Ya tenemos todo listo para poder simular como sería el trade de la energía en una ciudad así en el que cada parcela sea capaz de comprar y vender energía a toda la red. Veremos como se define el perfil de una casa y como afecta cada uno de sus parámetros a la simulación final, así como los detalles más importantes sobre el diseñado e implementación del simulador.

5. Simulación

En esta sección explicaremos más en detalle la parte de simulación. Para ello consideremos el siguiente diagrama: en cada apartado de esta sección se explica un



paquete del programa: 5.1 se habla del paquete Vista, 5.2 del paquete de control, y el modelo está dividido en 3 paquetes el núcleo 5.3, componentes 5.4 paquete componentes, 5.5 Distribuciones temporales.

5.1. Principios de diseño

5.1.1. Patrón MVC

El proyecto sigue el patrón modelo, vista y controlador. Se trata de dividir el código en tres partes:

- Vista: Se encarga de todas las interacciones externas del programa, bien sea capturar las interacciones del usuario o conexiones con otros programas o servicios como la red o el sistema de ficheros.
- Controlador: Controla el flujo principal del programa. La clase principal (main) se sitúa aquí. Funciona como enlace entre el modelo y resuelve los errores que se hayan podido generar a lo largo de la ejecución.
- Modelo: Contiene la representación de los datos usados en el programa ordenados de manera jerárquica.

Este patrón intenta eliminar dependencias entre distintas partes del programa para que sean modificables independientemente, lo cual proporciona mantenibilidad al proyecto.

5.1.2. Patrón experto y la encapsulación

La encapsulación es uno de los grandes pilares del paradigma de Programación Orientada a Objetos. Se trata de ocultar los atributos que definen el estado de una clase de manera que solo sean modificables mediante los métodos que define la clase. De esta manera, el usuario de la clase se despreocupa de la representación interna de los datos y se centra únicamente en como usarlos. Esto previene que los datos sean modificados de forma incontrolada añadiendo robusteza y mantenibilidad al proyecto.

El patrón experto ayuda a implementar la encapsulación en un proyecto. Afirma que aquel quien debe modificar el estado de un objeto es aquel quien sabe más sobre la misma, es decir, la propia clase. Un ejemplo simple que ilustra la encapsulación a través del patrón experto podría ser:

- Una clase persona tiene como atributo su edad.
- Se quiere saber si una persona concreta es mayor de edad o no.
- Una posibilidad sería implementar un método `getEdad()` que devolviese la edad de la persona y evaluar `getEdad() >= 18`.
- Una alternativa que sigue el patrón experto sería crear un método `esMayorDeEdad()` que internamente evaluara `edad >= 18`.
- Observamos que por un lado, la responsabilidad sobre chequear el estado del objeto se delega a la propia clase, que es el experto.
- Como beneficio añadido, se elimina la dependencia sobre la representación de datos interna, ya que de este modo el usuario de la clase no debe saber si la edad se representa mediante un entero o una clase propia o podría incluso no existir ningún atributo y que el resultado dependiera de un proceso de cálculo interno.

La encapsulación facilita futuras modificaciones y aumenta la escalabilidad del programa, proporcionando mantenibilidad al proyecto.

5.1.3. Interfaces y ampliación

Una interficie es una colección de métodos abstractos: Cabeceras de métodos sin implementación de su cuerpo. Una clase que implemente una interfaz debe definir el cuerpo de cada método abstracto descrito en la interfaz.

Se permite guardar referencias a objetos a nivel de interficie, de manera que cualquiera clase que la implente, podrá ser asignada al objeto, sin importar la implementación concreta.

Esto proporciona separación entre distintas partes del proyecto, eliminando dependencias y proporcionando mantenibilidad al proyecto como hemos discutido previamente. Además profundiza en la idea de la encapsulación, ya que al no saber como se implementarán las interficies ni como serán sus atributos, se evita incluir métodos que accedan directamente a la representación del objeto.

Por último, las interficies proporcionan gran facilidad para ampliar el proyecto. Basta crear una nueva clase que implemente una interfaz del proyecto de otra manera para proporcionar una ejecución y resultados alternativos del proyecto.

A continuación analizaremos el objetivo y funcionamiento de los distintos paquetes del proyecto.

5.2. El paquete view

El programa interactúa con dos entes externos: La interfaz de línea de comandos y el sistema de ficheros. Cada una se representa en una clase propia que cuentan con métodos estáticos que proporcionan las funcionalidades deseadas.

5.2.1. Command Line Interface

La clase `CommandLineInterface` cumple dos funciones:

- Analiza los parámetros pasados a la ejecución del programa: Asegura que siguen el formato establecido e informa al usuario del formato en caso contrario. Toda la información de entrada está contenido en un único fichero JSON, de manera que el único parámetro es la ruta del fichero de entrada. Concretamente, el formato del comando para ejecutar el JAR distribuible es:

```
java -jar simulator.jar 'ruta del archivo de entrada'
```

- Ejecuta scripts de Gnuplot: A cada frame de la simulación, se crea un fichero que contienen órdenes de gnuplot para generar gráficas que representan las apuestas de mercado hechas por cada prosumer. A partir de la línea de comandos, se ejecuta un proceso de gnuplot que procesa estos scripts mediante el comando:

```
gnuplot 'ruta del archivo de script'
```

5.2.2. FileSystem

La clase `FileSystem` proporciona métodos para guardar y cargar ficheros desde la memoria. Concretamente:

- Procesa un fichero JSON ubicado en la ruta especificada y devuelve un objeto simulación con todos los parámetros inicializados.
- Guarda los comandos de gnuplot en un fichero de script en la ruta especificada.
- Guarda los resultados de la simulación en un archivo JSON en la ruta especificada. Para consultar el formato de salida dirigirse al apartado 4.8.

5.3. El paquete control

El flujo principal de programa se centra en crear una simulación ejecutando secuencialmente frames del mismo. La clase principal del programa es Main, el cual crea una instancia de simulación donde se encuentra el ciclo principal del programa. La simulación utiliza el CEAPSolver para resolver el problema CEAP a cada frame.

5.3.1. Main

- Parsea los parámetros recibidos utilizando Command Line Interface
- Genera una instancia de simulación utilizando FileSystem
- Ejecuta el bucle principal de la simulación
- Genera las gráficas de gnuplot usando FileSystem
- Guarda los resultados en un fichero JSON usando FileSystem

5.3.2. Simulation

Una simulación viene definida por los siguientes parámetros:

- Market: Árbol dirigido donde cada nodo representa un prosumer y las arestas cables de electricidad que unen dos prosumers
- Hour y minute: El momento del día en el que empieza la simulación
- Frames: La cantidad de frames a simular. Afecta linealmente al tiempo de ejecución
- minutesPerFrame: La cantidad de tiempo que se avanza entre frame y frame
- OutputFolder: Carpeta donde se guardarán las gráficas de gnuplot

El bucle principal del programa es el siguiente: [Pseudocódigo] Create JSON object; do ¡Market : Set bids ¡; CEAPSolver : Solve Market; Market : Process results ; Market : Generate script ; /Gnuplot Script /; FileSystem : Plot Bids ; Add Frame to JSON object ; while(More frames?) Return JSON object;

CEAPSolver se refiere a la librería importada.

5.4. El paquete `model.data`

Antes de describir la jerarquía del modelo de datos de arriba abajo, analizamos un nuevo tipo de dato esencial introducido en el proyecto, utilizado por distintas componentes principales del programa: `ITemporalDistribution`.

5.4.1. `ITemporalDistribution`

Los `TemporalDistribution` pretenden modelar valores que van cambiando a lo largo del tiempo, así como la cantidad de nubes o el gasto energético de una lavadora. Además proporciona una serie de facilidades que serán útiles a la hora de manipular datos que dependan del tiempo. La interfaz es la siguiente:

- `public Double getValue(Moment moment);`
Devuelve el valor en el momento dado.
- `public double getMeanBetween(Moment since, Moment until);`
Devuelve la media de los valores que coge la variable entre dos momentos.
- `public double getAddedValue(Moment since, Moment until);`
Devuelve la suma de todos los valores que coge la variable entre dos momentos.
- `public double getProgress(Moment moment);`
Devuelve el avance de progreso en la que se sitúa el momento dado respecto al dominio de variabilidad de la variable.

Existen dos implementaciones principales de la interfaz:

- `DiscreteTemporalDistribution`: Lo implementa extendiendo la clase `TreeMap<Moment, Double>`. Contiene un conjunto discreto y ordenado de entradas `Moment : Double`.
- `ContinuousTemporalDistribution`: Lo implementa utilizando dos `UnaryDoubleOperators`: Una para la función que modela el valor de la variable a lo largo del tiempo y una primitiva suya. De esta manera, la suma de todos los valores entre dos momentos que es la integral, se puede calcular restando la evaluación de la primitiva en el momento inicial a la evaluación de la primitiva en el momento final.

5.4.2. `Data`

En `Data` se definen variables estáticas que son las instancias de `ITemporalDistribution` que se usarán en el proyecto.

5.5. El paquete `model.core`

En el paquete `core` se encuentran los objetos más importantes que intervienen en la simulación: Aquellos que no dependen de otros.

5.5.1. Market

Market es la cúspide de la jerarquía de datos del programa y contiene referencias a todos los datos del modelo. Está constituido por un conjunto de Prosumers y un conjunto Wires que forman el mercado interno del problema, un único Distributor que está conectado a cada prosumer y puede intercambiar energía con ellos y un Weather que controla el tiempo que hace, que afecta a las tasas de generación de los generadores.

Tiene tres funcionalidades principales:

- Set Bids : Sitúa la ciudad en un momento concreto. Actualiza el tiempo que hace, el precio de la distribuidora y a continuación, pide a cada prosumer que realiza su apuesta acorde a la nueva información del entorno. Para analizar la realización de las apuestas consultar apartado [6.89]
- Process Results : Asigna a cada cable la corriente que transcurre por ella y a continuación indica a cada prosumer los intercambios que ha realizado en la última tanda de tratos.
- Generate Script : Genera un único fichero que contiene las órdenes de gnuplot para generar las gráficas que representan la oferta hecha y los resultados obtenidos de cada prosumer. La escritura de las funciones a dibujar se delega a IBiddingStrategy dado que su expresión dependerá de la implementación elegida.

El formato genérico de estos scripts es:

5.5.2. Wire

Guarda conjuntamente la información de un cable: El prosumer de origen, el prosumer de destino, la capacidad máxima energía que puede transportar, y el flujo de energía en el momento actual. Los tres primeros valores se mantienen constantes a lo largo de la ejecución del problema mientras que el flujo se actualiza después de cada tanda de intercambios.

5.5.3. Weather

El tiempo se modela usando dos variables, nubes y viento, que van actualizándose a lo largo de la simulación. La variabilidad de estos valores viene dado por un ITemporalDistribution que se explica en el apartado [6.89], gracias a ello ofrece

facilidades como obtener la cantidad media de nubes o la velocidad media del viento que ha habido entre un periodo de tiempo.

5.5.4. Distributor

Los distribuidores tienen un `IBiddingStrategy` que indica la apuesta que hacen en todo momento. A diferencia de los Prosumer, esta apuesta va variando según el momento en el que nos encontros, independientemente de como hayan sido los intercambios hechos en tandas previas. Para ello tienen `ITemporalDistributions` que indican como deben variar sus parámetros dependiendo del momento. En la implementación que se muestra, las distribuidoras usan una estrategia de apuesta lineal que depende de un parámetro, el ratio del precio, que va variando según la hora del día.

5.5.5. Prosumer

5.6. El paquete `model.components`

Contiene los objetos que forman un prosumer

5.6.1. `IBiddingStrategy`

5.6.2. `IBattery`

5.6.3. `IGenerator`

5.6.4. `IAppliance`

5.7. A continuación

6. Visualización

6.1. En este capítulo

Dada la naturaleza tridimensional del problema, se optó por usar un programa que permitiera visualizar los datos en 3D e interactuar con ellos. Unity, como editor de juegos, permite esta interacción y tiene la gran ventaja de poder crear un ejecutable para las plataformas más reconocidas como Windows, Mac, Linux, pero también web o android. Dicho ejecutable será el producto final del proyecto, ya que incluirá el simulador, mientras que este a su vez incluye el resolvidor RadPro. Veremos primero la filosofía de diseño de Unity que cambia bastante en referente a paradigmas de POO comunes como Java o C++. Explicaremos el esquema básico de las componentes que forman el visualizador, y nos adentraremos en los detalles más interesantes que esconden. Por último veremos como crear el ejecutable para poder distribuir nuestro programa fácilmente.

6.2. Filosofía de diseño de Unity

La programación en Unity puede efectuarse en los lenguajes C++, JavaScript y Boo, aunque este último no se utiliza habitualmente. Todos ellos son lenguajes dentro del paradigma Programación Orientada a Objetos, pero en Unity cambia la manera en el que se organizan los datos. El cambio más importante es que lo que se programa son comportamientos, no entes. Los objetos centrales de Unity son los prefabs: Packs de componentes que definen como se comporta ese objeto dentro de la escena. Por ejemplo, todos los objetos tienen una componente Transform que indica la posición, rotación y escala del objeto. Unity ofrece varias componentes comunes predefinidas listas para usar en tus prefabs tal como colliders para detectar colisiones entre objetos o mesh renderers para definir la geometría (los vértices) que forman el objeto. Por ello la implementación de funcionalidades se centra en encontrar acciones/comportamientos de los objetos de la escena y definir cada uno de ellos en un fichero aparte. Por ejemplo: De manera que un mismo objeto que camine y hable tendrá, en vez de un fichero persona en el que se especifican ambos comportamientos como métodos, dos comportamientos definidos en ficheros distintos Caminar y Hablar. La idea detrás de esto es que si a posteriori se crea un objeto que solo camine, baste con añadirle la componente de caminar y ajustar sus parámetros desde dentro del editor. Las componentes son clases especiales que heredan de la clase MonoBehaviour de la librería de Unity. Esto permite que los comportamientos sean añadidos a modelos creando lo que se conoce como gameObjects o prefabs, packs de modelo + comportamientos. Comportamientos comunes como colisiones o animaciones ya vienen implementadas en Unity y basta con añadirle la componente correspondiente a

6.3. Diseñando los paneles

Una vez ejecutada la simulación, contamos con un json que describe toda la información relevante que ha ocurrido en cada frame. La información sobre el momento del día lo mostraremos en la GUI principal de manera que sea apreciable desde cualquier punto de la escena. Aprovechando la naturaleza 3D de nuestro visualizador intentaremos mostrar toda la información posible en 3D. La información referente a una casa la mostraremos en un canvas (panel 2D) pero situado dentro de la escena 3D sobre la casa en cuestión. Por último visualizaremos los cables como líneas que van desde la casa origen hasta la casa de destino, su tamaño representará la capacidad de ese cable, visualizaremos el flujo de electricidad moviendo unas partículas en la dirección del flujo sobre el cable y el flujo en si en un panel 3D sobre las partículas.

Diseño del panel de una casa: 

- En la barra izquierda se muestran los generadores que contiene la casa. Un icono representa el tipo de generador que se trata, bien solar o eólica y sobre la imagen se sobrepone un relleno que representa la eficiencia que está teniendo ese generador en ese momento en referencia al máximo rendimiento que podría tener.
- En el panel central se muestra la gráfica de la apuesta realizada por esta casa.
- En el panel derecho se añaden los dispositivos. De manera similar a los generadores, están representados por un icono referente al tipo del dispositivo y sobre ellos se sitúa un relleno que muestra su progreso.
- En el panel inferior se encuentra la batería, que también contiene una imagen que se rellena indicando el porcentaje de batería actual respecto al total.

Tanto los generadores como los dispositivos son variables y se deben añadir en tiempo de ejecución, una vez se sepa cuantos y de que tipo son. Por ello, crearemos un prefab para cada uno de ellos.

Los cables los modelamos usando la componente LineRenderer de unity, que dados dos puntos genera una línea personalizable entre que los une. Sobre ella moveremos un objeto con la componente TrailRenderer, que deja una estela detrás del objeto mientras esta se mueva. Por último, sobre esta partícula situaremos un texto que indica la energía que se transfiere en ese instante.

6.4. Control de la cámara

En cualquier momento del programa, el usuario puede controlar la posición y rotación de la cámara. Los controles imitan el funcionamiento que ofrece el editor de unity por defecto: Con las teclas WASD se controla la posición de la cámara mientras que el movimiento del ratón mientras se tenga el click derecho presionado controla la rotación.

MonoBehaviour contiene un método Update() que se llama a cada frame de la simulación. Cabe notar que entre llamadas al método update pasa un tiempo variable. Por ello hay que ir con cuidado a la hora de hacer animaciones etc. hay

que considerar la variable `Time.deltaTime` de unity en donde se guarda el valor de tiempo que ha transcurrido entre el frame actual y el anterior. Usando los métodos estáticos de la clase `Input` de `UnityEngine` `OnMouseDown` y `GetKeyDown()` se detecta si alguna de las teclas del control de cámara ha sido pulsada. A partir de aquí, se implementa la funcionalidad deseada (`Translate` para las teclas de control y `Rotate` para la mouse).

6.5. Eligiendo la ciudad

Añadir un modelo a una escena previamente a su ejecución es relativamente sencillo, basta con crear un prefab que tenga las componentes que te interesan. Sin embargo, importar un modelo en tiempo de ejecución en Unity es más complicado de lo que puede parecer. El primer problema es abrir un cuadro de diálogo que permita al usuario elegir un archivo. Unity usa Mono que es un subconjunto de `.NET Framework 2.0` especializado para juegos, que por defecto no incorpora la opción de crear ventanas propias del SO. La solución ha sido añadir el `dll Windows.Forms.dll` de `.NET Framework 2.0` en una carpeta llamada `Plugins` dentro del proyecto de unity de manera que el compilador de unity comprenda que queremos usar tal extensión y permita crear las ventanas. El soporte de `.NET Framework 2.0` no está garantizado por Unity por lo que pueden saltar errores de seguridad al intentar abrir la ventana, pero ignorando estos defectos podemos permitir que el usuario abra un fichero. Quiero agradecer a [¡Referencia!](#) por la solución prestada.

6.6. Seleccionando el perfil de las casas

Una vez la ciudad se encuentre en la escena el usuario podrá ver y modificar el perfil de la gente que vive en esa casa en tres niveles: Alto consumo, consumo regular o alto ahorro.

Cuando el usuario pasa el ratón por encima de una casa, esta casa se coloreará de un color dependiendo del perfil que tenga asignado. Por defecto, todas las casas tienen asignado el perfil de consumo regular que viene representado por el color naranja. Una vez situado el ratón sobre la casa el usuario podrá elegir el perfil deseado para esa casa simplemente pulsando las teclas 1, 2 o 3 siendo 1 el de menor consumo y 3 el mayor.

Por último si se pulsa la tecla P, las casas se colorearán con el color correspondiente a su perfil hasta que se vuelva a pulsar P. Esto permite visualizar la distribución de perfiles a lo largo de una ciudad.

6.7. Seleccionando la franja horaria

El segundo paso es seleccionar de que hora a que hora se ejecutará la simulación. Para ello contamos con un torus fino con dos bolas sobre ella. Estas bolas se pueden seleccionar y arrastrar sobre el torus y muestran un lienzo al moverse que indica la

hora que señalan. La idea bajo esta implementación es: - Detectar si el mouse está en modo drag con el método `OnMouseDown` de `Monobehaviour`

- Coger el punto que representa el ratón en la ventana
- Hacer un cambio de base para que deje el pixel 0,0 esté situado en el centro de la screen
- normalizar el vector que va desde el centro hasta el punto del ratón.
- escalarlo por el radio mayor del torus para situarlo encima del mismo

Por defecto estos valores empiezan de 06:00h a 18:00h. Cabe considerar que para que el funcionamiento del programa sea el esperado, la hora inicial debería ser inferior a la hora final

6.8. Ejecutando la animación

Como tercer paso, el usuario hace click en la estrella de acuerdo con que está contento con los parámetros elegidos hasta ahora. Esto crea un nuevo thread que se encarga de correr la simulación. La creación del thread se consigue gracias a la clase `ThreadedJob` que es aportación de [referencia]. En particular este thread tiene un método que se ejecutará al ser creado. Además, cada vez que el thread principal llame al método `update` de `ThreadedJob`, este actualizará su estado y en caso de que haya acabado generará un evento que será tratado por el thread principal. Por último el thread se destruirá porque ya ha completado su trabajo. El thread al mismo tiempo crea un `Process` para abrir una ventana de terminal y ejecuta el simulador con los parámetros proveídos: `java -jar simulator.jar`. Cabe decir que si se utilizan rutas absolutas para decir a la terminal donde se encuentra el jar, nuestra aplicación no será distributable, porque no funcionará en otros ordenadores, por ello debemos poder acceder al simulador usando rutas relativas. Unity provee con la variable `Application.dataPath` que referencia en tiempo de ejecución a la carpeta de datos que está utilizando Unity en ese momento. Mientras estamos en el editor de Unity, esta carpeta es la carpeta `Assets` del proyecto mientras que una vez se haya hecho la build, es la carpeta que acompaña al ejecutable .exe. Por eso situaremos el jar dentro de esta carpeta y la aplicación será distributable.

6.9. Mostrando la información relevante

Durante toda la ejecución del visualizador, se minimiza la información mostrada de manera que sea más intuitivo y disfrutable para el usuario final. En particular diversos objetos comparten el comportamiento de revelar/ocultar cierta información al ser pulsados. Por ejemplo, al pulsar sobre las casas, veremos el informe de qué está ocurriendo en esa casa mientras se ejecuta la simulación, mientras que si pulsamos sobre los focos de energía que pasan sobre los cables, veremos o ocultaremos un texto con cuanta energía transportan en cada momento.

Por otro lado, mediante las teclas H y G podemos controlar qué partes de la ciudad se ven en cada momento. Con H (de house) podemos revelar/ocultar

las casas mientras que con G (de grid) podemos revelar/ocultar los cables. Por último, con la tecla Esc podremos salir de la aplicación. El objeto manager es el que contendrá estos comportamientos.

6.10. Programando las animaciones

La simulación nos da frames discretos en los que sabemos como es el estado del sistema. Nuestra intención será interpolar linealmente estos valores para que parezca que el movimiento es continuo.

Unity cuenta con 2 modos por defecto que permiten hacer animaciones: -La componente Animation permite ejecutar animation clips guardados en una lista, es la manera original de hacer las animaciones. -La componente Animator también conocida como Mecanim, que genera una máquina de estados que tiene en cada estado el clip de la animación y contiene transiciones entre estados que se encargan de controlar como cambian las animaciones.

Sin embargo, ninguna de ellas permite cambiar los clips de animación en tiempo de ejecución. Los clips han de estar guardados en ficheros .anim y ser referenciados en estas componentes, no se permite añadir clips en tiempo de ejecución. Es una limitación del sistema actual que han anunciado que añadirán en una próxima versión. Por ello, la solución es crearnos nosotros un algoritmo sencillo y simple que se encargue de simular animaciones básicas.

Hay dos modos de animación: Repetitivo y continuo. En el repetitivo, el mismo frame se anima una y otra vez para analizar correctamente que ha pasado concretamente entre esos instantes de tiempo. En el continuo, la animación va recorriendo todos los frames y se repite una vez haya llegado al final. Todos los animadores tienen como entrada un array de datos. En el caso de la batería estos datos representan el porcentaje de llenitud, mientras que en el caso de los dispositivos representan su progreso. La idea es llamar a un método un número concreto de veces cada segundo donde se recalcule el valor actual y se actualize la información necesaria correspondiente. Al finalizar el segundo, se llama a otro método en el que o bien se pasa al siguiente estado o bien se reinicia el estado actual dependiendo de si la animación está en modo repetitivo o no. Una vez que hayamos calculado el valor actual basta con actualizar Image.fillAmount con el valor correcto.

6.11. Controlando el flow

Incluso en un entorno dirigido por componentes como unity, necesitamos un objeto que controle el estado del programa y decida que opciones están disponibles al usuario en cada momento. En nuestro proyecto, la componente Manager se encarga de esta función. Además sirve como puente sobre las diferentes componentes. Si un objeto quiere comunicar algo a otro pero no tiene guardada una referencia a él, puede pasar la información al manager y este será el que haga llegar la información al destinatario. Funciona similarmente al método main de los programas Java.

no puedo hacerlo con ThreadedJob porque no puedo hacer Instantiate dentro.

La solución sería devolver una struct con toda la información recopilada por el parseador y hacer las instanciaciones en el thread principal.

6.12. Parseando el JSON

Así como crear el json era muy sencillo, parsearlo lleva más tiempo. Primero debemos encontrar una librería adecuada que sirva para Unity. Probablemente la herramienta más popular utilizada para controlar archivos JSON en C# sea Json.NET creado por James Newton-King. De todas formas, dadas las limitaciones de compatibilidad de Unity con .NET, se necesita adaptar el código a Unity. Existe esta adaptación a la venta en la Asset Store de Unity, pero para mantener este proyecto gratuito, se ha evitado usarlo.

De todos modos existen parseadores de JSON creados por la comunidad especialmente para Unity. Nosotros usaremos JSONObject de preferencia.

El primer paso es detectar con cuantos generadores y dispositivos cuenta cada casa e instanciar un prefab por cada uno en el panel de la casa correspondiente. Una vez todos los objetos de la escena existan popularemos sus componentes de animación con los datos guardados en el JSON. Para cada tipo de dato deberemos identificar cual es su animador, o bien FillImageAnimator en el caso de la batería, los generadores o los dispositivos, o bien ChangeImageAnimator en el caso de las apuestas, o bien TranslationAnimator en el caso

6.13. A continuación

Mostraremos las conclusiones obtenidas y posibles rutas de avance para este proyecto.

7. Futuro trabajo

7.1. Ampliar a Open Street Map, CityEngine.

Por último se ha considerado como construir los modelos 3D de las ciudades. Y qué mejor manera que utilizando ejemplos existentes del mundo real. El mundo está repleto de ciudades y gracias a mapas online como OpenStreetMap es fácil conseguir un fichero que contenga toda la información relevante de una zona del mundo, como por ejemplo el corazón de Manhattan. En OpenStreetMap podemos exportar un trozo de mapa en formato .osm muy parecido a xml que contenga toda la información sobre calles, parcelas, parques etc. que se encuentran en ella. Ahora solo queda usar un programa de modelado procedural como CityEngine para crear una ciudad en 3D a partir del mapa! CityEngine soporta nativamente el tipo de archivos .osm y es capaz de crear modelos 3D a partir de mapa extruyendo las parcelas de edificaciones y añadiéndoles materiales adecuados.

OpenStreetMap, como el nombre lo indica, es una iniciativa de software libre en el que son los mismos usuarios los que van completando el mapa y aportando cada vez información más detallada como donde se encuentran fuentes de agua o ciertas señales de tráfico. Sin embargo, CityEngine es un programa con licencia de pago, por lo que se ha dejado fuera del proyecto final. De todas maneras, podremos visualizar algunos resultados obtenidos usando la aplicación de prueba de 30 días que ofrece la empresa, que soporta importar archivos .osm y exportar el modelo en formato OBJ.

7.2. OpenWeatherMap

OpenWeatherMap es una API pública que ofrece datos históricos y predicciones futuras sobre el tiempo a lo largo de todo el planeta. inspirada en la filosofía de OpenStreetMap y Wikipedia ofrece la información gratuitamente de manera que sea accesible para todos. Cuenta con datos de más de 250.000 ciudades repartidos a lo largo del mundo y su uso va creciendo superando ya un billón americano de consultas de predicciones al día.

Aplicado al proyecto, podría hacerse un pequeño cliente HTTP que hiciera queries a la API de OpenWeatherMap y utilizar el JSON de respuesta para modelar el tiempo en la simulación. Además, unido con OpenStreetMap, podría usarse la longitud y la latitud del mapa seleccionado para consultar el tiempo en esa zona del mundo y utilizar dicha información en el simulador. Para empezar a consultar la API, basta con crear una API Key en <http://openweathermap.org/appid> y consultar el formato de las queries en su documentación en <http://openweathermap.org/api>.

7.3. Reinforcement learning en bidding

Por último el proyecto proporciona una plataforma donde poder testear estrategias de bidding en un mercado de energía. Tal y como se describe en el artículo

[referencia], se pueden implementar procesos de reinforcement learning de manera que los prosumers vayan variando su oferta dependiendo de la información que vayan acumulando a lo largo del tiempo.

Reconocimientos

Me gustaría agradecer a los siguientes personas por proporcionar material indispensable para la realización del proyecto:

- Icono 2D de la visualización: Icons made by Freepik (www.freepik.com) and Google ("http://www.flaticon.com/authors/google") from Flaticon (www.flaticon.com) licensed by Creative Commons BY 3.0 ("http://creativecommons.org/licenses/by/3.0/")
- Efecto de cargando: LoadingEffect by OneManArmy (<http://armedunity.com/user/1-onemanarmy/>)
- Parseador de ficheros OBJ: ObjReader by Starscene Software (starscenesoftware.com)
- Visor de diccionarios en el editor de Unity: SerializedDictionary by Vexe
- Pseudocódigo y diagramas de flujo: Code2Flow

Referencias

- [1] RadPro
- [2] Reinforcement learning
- [3] Batut, C.; Belabas, K.; Bernardi, D.; Cohen, H.; Olivier, M.: User's guide to *PARI-GP*,
`pari.math.u-bordeaux.fr/pub/pari/manuals/2.3.3/users.pdf`, 2000.

```

set terminal png

set output 'ruta al fichero de salida'

// Expresiones de funciones

f_1(x) = x >= <minX> && x <= <maxX> ? <expresión> : 1/0
f_2(x) = ...

// Indicadores de tratos establecidos (flechas)

unset arrow

set arrow from <trato>, 0 to <trato>, f_<i>(<trato>) front
set arrow w ...

// Configuración de parámetros de gnuplot

set samples <precisión>

set nokey

set xzeroaxis

set yzeroaxis

// Pintado de las funciones en una gráfica

plot [<dominio de la gráfica>] f_1(x) <personalización de
pintado>, f_2(x) ...

```