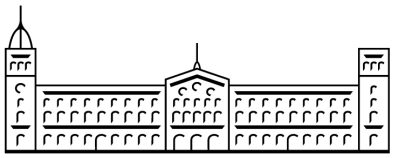


Trabajo final de grado

GRADO DE INFORMÁTICA

Facultad de Matemáticas
Universidad de Barcelona

Modelado, simulación y visualización de mercados distribuidos de energía

---

Autor: Martin Azpillaga Aldalur

Directores:	Dr. Jesús Cerquides
	Dr. Juan Antonio Rodríguez-Aguilar
Realizado en:	Consejo superior de investigación científica
Barcelona,	28 de enero de 2016

Abstract

Humanity needs energy. The industry, the educational and health institutious, even our own houses, all need energy to generate the goods they provide to us daily. But the raw materials historically used to produce energy is exhausting. Upon the drastical future, new models have been proposed. On one hand, new sources of energy have been found and developed, but none has overcome the use of fossil materials yet. On the other hand, there is ongoing reserch to find more efficient ways ways to use and deliver energy.

The energy markets we know are centralized markets: a distributor produces the energy and hands it out using the electrical grid. As a consequence, a big part of the energy is lost due to distribution expenses. An alternative model surges under the name of Smart Grids [1]: distributed markets that allow users to trade (buy or sell) energy between them. The discovery and improvement of secure and fair algorithms that find the optimal allocation of energy for a distributed market given the requirements of each user and the constraints of the grid is a major research problem. In [2], the RadPro algorithm is presented, stating that can be used to solve energy allocation problems up to some restrictions.

This project, intends to visualize the evolution of a distributed market where the users make trade offers depending on their necessities. Each user is provided by some amount of electrical components such as generators and appliances that produce or consume energy over time. As a result, the trading offer of a user changes over time, depending on the energy requirements that dictate the state of its components. The project gives the option to build personalizable distributed markets and runs simulations over them. The results can be visualized in a 3D environment where the user is able to analyze and interact with all the information.

Resumen

La humanidad necesita energía. La industria, los centros de enseñanza y de salud, incluso nuestras casas, todos necesitan energía para generar los bienes que nos proveen cada día. Pero la materia prima históricamente utilizada para producir energía se está agotando. Ante la alarmante situación, se han propuesto nuevos modelos de energía. Por un lado, se han descubierto y desarrollado nuevas fuentes de energía. Por otro, se investiga en nuevas maneras de utilizar y repartir energía de manera más eficiente.

Los mercados de energía que conocemos son mercados centralizados: una distribuidora produce la energía y la reparte utilizando la red eléctrica. Como consecuencia, gran parte de la energía producida se pierde en gastos de distribución. Un modelo alternativo surge bajo el nombre de Smart Grids [1]: mercados distribuidos que permiten a los usuarios intercambiar, comprar o vender, energía entre ellos. El hallazgo y desarrollo de algoritmos justos y fiables que permitan encontrar la asignación de energía para un mercado distribuido de energia dadas los requerimientos de cada usuario y las restricciones de la red es un gran problema de investigación.

En [2], se presenta el algoritmo RadPro, que afirma poder ser usado para resolver problemas de asignación de energía bajo unas condiciones.

Este proyecto, pretende visualizar la evolución de mercados distribuidos de energía en los que los usuarios definen ofertas de compraventa dependiendo de sus necesidades. Cada usuario dispone de un número de generadores y dispositivos que producen o consumen energía a lo largo del tiempo. Como resultado, la oferta de compraventa de un usuario va variando en el tiempo dependiendo del estado de sus componentes. El proyecto permite crear mercados distribuidos personalizables y ejecuta simulaciones sobre ellos. Los resultados pueden ser visualizados en un entorno tridimensional donde el usuario es capaz de analizar e interactuar con la información.

Índice

1. Introducción y antecedentes

La energía es una necesidad básica en nuestra sociedad. Prácticamente toda la industria requiere de energía para producir bienes. Incluso dentro de nuestras casas, hay mutitud de electrodomésticos que utilizan energía para facilitarnos nuestro día a día.

1.1. El dilema energético

A lo largo de los años, la energía se ha producido explotando sustancias fósiles de gran contenido energético como el carbón, el petróleo o el gas natural. Gracias al enorme consumo de energía en el planeta, las sustancias fósiles han ido disminuyendo en abundancia, haciendo cada vez más costosa su obtención y distribución, y poniendo potencialmente al planeta en una situación donde no haya suficiente materia prima para sustentar la demanda energética de la sociedad. Ante la alarmante situación, ha habido gran interés en investigar maneras de solventar este problema. Podemos separar las soluciones propuestas en dos vertientes principales:

- Una dirección es utilizar otras fuentes de energía, como la energía nuclear o las energías renovables.

  La energía nuclear genera grandes cantidades de energía, pero representa un riesgo de irradiación radioactiva peligrosa en caso de un error del sistema. Por esta razón, nunca ha sido totalmente aceptada por la sociedad y se ha disminuido su uso en los útimos años.

  La gran ventaja de las energías renovables es que son teóricamente inagotables: deberíamos ser capaces de generar energía mientras haya luz solar o viento sobre el planeta. Además, no presentan posibles riesgos de salud y son más respetuosos con el medio ambiente. Aún son una tecnología joven, pero ha tenido grandes avances en los últimos años. De hecho, ya no solo se hacen plantaciones industriales de placas fotovoltaicas en zonas de alta generación, sino que se ha llegado al punto de que un usuario particular puede permitirse comprar sus propias placas fotovoltaicas para autoabastecer el consumo de su vivienda.

- La otra dirección es disminuir el consumo energético. Con este objetivo en mente, cada año se abren múltiples campañas de concienciación que pretenden transmitir la importancia y los efectos que tiene ahorrar en energía. También se crean leyes y decretos que regulan el gasto energético de las empresas industriales.

  Una nueva idea es optimizar la manera en la que se distribuye la energía para evitar gastos inecesarios. El modelo de distribución más extendido actualmente es un modelo centralizado: Una o unas cuantas distribuidoras se encargan de producir la energía y la reparten entre los consumidores usando el cableado eléctrico. Como consecuencia, una gran parte de la energía producida se

pierde, disipándose en forma de calor. La alternativa es crear mercados distribuidos donde los usuarios estén interconectados y puedan intercambiarse energía entre sí, minimizando así gastos de distribución.

Las Smart Grid [1] unen la idea de que a partir de ahora, los usuarios podrán producir energía con la idea de utilizar un mercado distribuido, ofreciendo un modelo de mercado inteligente en el que cada usuario podrá intercambiar energía con otros usuarios. A estos usuarios se les asigna el nombre de prosumidores, ya que son productores y consumidores de energía simultaneamente.

La consideración de un modelo distribuido de mercado crea nuevas preguntas a responder: ¿cómo indica un usuario sus intereses de compraventa?, ¿cómo afecta la topología y el diseño de la red a los intercambios?, y sobre todo, ¿cómo se gestionan los intercambios?. En el siguiente apartado formalizaremos el problema para poder dar una posible resolución a estas preguntas.

1.2. El problema de la liquidación de mercados distribuidos de energía

Un mercado distribuido de energía puede modelarse como un grafo (P, E) donde cada nodo $p \in P$ representa un prosumidor y cada arista $\{p, p'\} \in E$ significa que el usuario p y el usuario p' están unidos mediante cableado eléctrico, de manera que pueden intercambiarse energía entre ellos.

Los intereses de cada prosumidor se representan mediante la oferta de compraventa que lanza al mercado. Esta oferta se modela utilizando una función $f : D \subset \mathbb{R} \to \mathbb{R}$ que indica el precio que está dispuesto a pagar dependiendo de la cantidad de energía que intercambie. El dominio de la función D puede contener un parte negativa que representa que el prosumidor pretende vender esa cantidad de energía, mientras que la parte positiva se utiliza para indicar los precios de compra.

Por ejemplo la función de compraventa:

$$\begin{aligned} f\colon [-5,10] &\to \mathbb{R} \\ x &\mapsto \begin{cases} -2x & x \in [-5,0] \\ 3x & x \in [0,10] \end{cases} \end{aligned}$$

representa que el usuario está dispuesto a vender cualquier cantidad $c \in [0, 5]$ de energía al precio de $2c$ unidades, y también está interesado en comprar una cantidad $c' \in [0, 10]$ al precio de $3c'$ unidades.

La topología de la red afecta a con quién puede intercambiar energía cada usuario. A la hora de resolver el problema, los prosumidores solo pueden intercambiar energía con prosumidores directamente conectados a ellos. No obstante, puede ocurrir que la energía intercambiada con un prosumidor adyacente llegue a un destinatario final externo.

Por ejemplo, consideremos un mercado de tres prosumidores A, B y C conectados secuencialmente como se muestra en la figura 1.

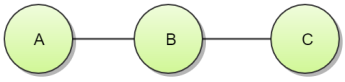


Figura 1: Ejemplo de mercado con tres prosumidores

En un momento de intercambio, el prosumidor A vende cuatro unidades de energía a su vecino B, el cual al mismo tiempo decide vender cuatro unidades de energía al prosumidor C. C no es vecino de A, pero dado que todos los intercambios se producen instantáneamente, el efecto final del intercambio es que el prosumidor A ha vendido cuatro unidades de energía a C. Sin embargo, para que esta operación sea posible, es indispensable que el prosumidor B haya mostrado interés en comprar y vender cuatro unidades de energía en su oferta de compraventa.

Como segunda restricción del mercado, se le añade a cada cable un máximo de capacidad de energía que puede transferir en cada instante. Ningún intercambio realizado podrá exceder la capacidad soportada por el cable.

Dada esta información, el problema de la liquidación de mercados distribuidos de energía consiste en encontrar la asignación de intercambios que optimice el beneficio total generado cumpliendo los requisitos de todos los prosumidores y de la red.

1.3. Algoritmos de resolución para mercados distribuidos de energía

Existen actualmente programas que resuelven el problema de la liquidación de energía simplificando alguno de sus aspectos. En [2], se define el problema EAP como el problema de liquidación de mercado donde las funciones de compraventa de los prosumidores son discretas. Para su resolución, se mapea el problema a un problema de programación lineal entera de manera que sea resoluble utilizando resolvedores comerciales como CPLEX y Gurobi. También se presenta el algoritmo RadPro: un método resolvedor basado en transpaso de mensajes que se aplica únicamente sobre grafos acíclicos, pero ofrece una gran mejora de eficiencia en estos casos.

A partir de entonces, se ha ampliado el problema a CEAP [3] que permite que las funciones de compraventa sean funciones lineales definidas a trozos.

El presente proyecto pretende utilizar este resolvedor de problemas CEAP para crear un entorno de simulación de mercados distribuidos de energía. Se amplía la expresividad de los prosumidores añadiéndoles componentes eléctricas como baterías, generadores y dispositivos de consumo. De esta manera, se puede generar una simulación donde las ofertas de compraventa de cada prosumidor van variando dependiendo del estado actual de sus componentes. Por último, se ofrece un entorno de visualización tridimensional donde poder analizar e interactuar con los resultados obtenidos.

La estructura general del resto del documento es la siguiente: en la sección 2 se definen los objetivos y requerimientos del proyecto. En la sección 3 se razona el software utilizado para realizar las distintas partes del proyecto. En la cuarta sección, se detalla el modelo de mercado distribuido utilizado. En la quinta sección, se explican los pasos que se siguen para ejecutar una simulación sobre el mercado, mientras que en la sexta, se muestran las técnicas utilizadas para visualizar los resultados de forma intuitiva e inteligible. Por último la sección 7 contiene ideas para poder seguir ampliando el proyecto en el futuro.

2. Objetivos

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 las funcionalidades y los casos de uso establecidos en 2.2.

2.1. Requerimientos no funcionales

La aplicación debe ser usable, mantenibile 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
  - Se seguirán **patrones de diseño** estándares a la hora de diseñar e implemetar 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 extensión** que permitirán añadir nuevas funcionalidades fácilmente en el futuro.
- Requirimientos 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 predominantes del mercado.
  - El producto final será un **ejecutable autocontenido**, de manera que no requiera configuración alguna para poder ser ejecutado.

2.2. Funcionalidades y casos de uso

El proyecto debe contener tres partes: Modelado del mercado, simulación del mercado y visualización de mercado. A continuación, se resumen sus funcionalidades y se detallan las interacción del usuario con cada parte:

- **Modelado del mercado**

  El modelado incluye el diseño y la creación de mercados distribuidos de energía. El modelo de mercado se extenderá añadiendo componentes más complejas que aumenten la expresividad del problema y otorguen realismo a las simulaciones que se realizarán sobre ellas. El usuario podrá crear mercados personalizables que sigan el diseño establecido y ajustar parámetros para controlar la futura simulación.

- **Simulación del mercado**

  La simulación trata de simular la evolución del mercado creado a lo largo de un periodo de tiempo. Las necesidades de cada usuario variarán y generarán en cada momento ofertas de compraventa distintas. El simulador deberá resolver el problema CEAP presentado en cada momento. Los resultados del intercambio aftectarán al estado de los datos, generando nuevas ofertas en el futuro. El usuario no interviene en la ejecución del simulador.

- **Visualización del mercado**

  En la visualización se plasman los resultados obtenidos de la simulación en un entorno tridimensional. Se trata de la reproducción de la evolución del mercado observada en el simulador utilizando un entorno visual, de manera que sea más fácil interpretar los resultados. El usuario podrá interactuar con la información y controlar el flujo de la reproducción.

En las figuras 2 y 3 se muestran los casos de uso del proyecto. Por un lado, se exige que cada parte sea ejecutable independientemente del resto. Por otro lado, existe un producto final que facilita la ejecución secuencial de todas las partes.

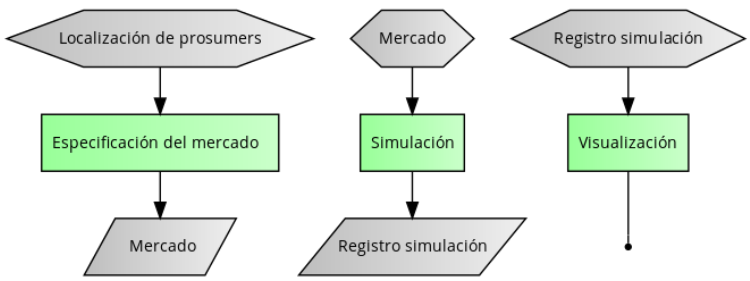


Figura 2: Cada parte ha de ser ejecutable de forma independiente

Figura 3: El producto final debe facilitar la ejecución secuencial de todas las partes

Considerando la naturaleza diversa del proyecto, se necesitan herramientas variadas para poder llevarlo a cabo. En la próxima sección, se explica el software utilizado a lo largo del proyecto, razonando su elección frente a posibles alternativas.

3. Metodología y herramientas

Explicaremos los programas y lenguajes más importantes usados durante todo el proyecto, razonaremos su uso ante programas equivalentes y citaremos a sus autores así como la licencia bajo la que se distribuyen. En el apartado 3.1 se explican las herramientas que ayudan a organizar el proyecto: control de versiones, documentación y diagramas. En el apartado 3.2 se encuentran los entornos de programación: Java y Unity, y en el apartado 3.3, se muestra el software utilizado para la generación de ficheros externos: Blender, JSON y Gnuplot.

3.1. Organización del proyecto

3.1.1. 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 versiones anteriores ante un error inminente.

Los sistemas de control de versiones más utilizados actualmente son Git, Mercurial y SVN. Git y Mercurial tienen la ventaja de ser sistemas de control de versiones distribuidos, lo cual permite ir guardando versiones localmente sin necesidad de un repositorio central remoto. Al ser el único usuario del proyecto, ni siquiera se ha creado un repositorio remoto y se ha trabajado localmente en un repositorio Git, valorando la experiancia previa del autor con dicho software ante Mercurial.

Git fue diseñado por Linus Torvalds y se distribuye bajo la licencia GNU GPL v2.

3.1.2. Documentación: LaTeX

Así como tener un sistema de control de versiones, mantener una documentación clara del proyecto es esencial para su éxito, sobre todo pensando en un futuro en el que se decida retomar el proyecto. Existen maneras específicas de documentar programas informáticos, como el JavaDoc de Java, pero debido a la variedad de herramientas de desarrollo del proyecto, se ha decidido unir toda la documentación comunmente en el presente documento.

Centrándonos en software genérico, la manera recomendable de escribir un documento técnico es utilizando los llamados 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. En este proyecto, se ha decidido usar LaTeX por su robustez y previa experiencia del autor con el mismo.

LaTeX es software libre distribuido bajo licencia LPPL.

3.1.3. Diagramas: Gliffy y Code2Flow

Los diagramas facilitan mucho la comprensión del proyecto, dado que pueden contener mucha información codificada mediante elementos visuales que la mente humana procesa rápida y eficazmente.

Los diagramas de modelo y diagramas de clase ayudan a entender la estructura interna del proyecto, así como las conexiones entre sus partes. La herramienta web Gliffy cuenta con una cuenta gratuita con el que se pueden crear múltiples diagramas de forma sencilla, siguiendo los estándares de notación.

Por otro lado, los diagramas de flujo explican el proceso que sigue el programa en un momento concreto. Son adecuados para detallar momentos enrevesados e importantes del programa. Normalmente, se utiliza pseudocódigo para acompañar estos diagramas. La herramienta web gratuita Code2flow convierte automáticamente el pseudocódigo en un diagrama de flujo acelerando el proceso de creación de ambos.

3.2. Entornos de programación

3.2.1. Entorno de simulación: Java

Podría usarse cualquier lenguaje de programación de propósito general para esta labor, pero dado que la base del simulador, el resolvedor de CEAP, está implementado en Java se ha decidido seguir usando el mismo lenguaje.

El uso de Java ayuda a conseguir el requerimiento de multiplataforma del proyecto y también se distribuye bajo licencia gratuita. Java fue originalmente desarrollado por James Gosling y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems.

3.2.2. Entorno de visualización: Unity

Unity es un motor de juegos, no está pensado para ser un entorno dedicado a la visualización como D3 o Processing. Sin embargo, un juego no es más que una visualización interactiva compleja. Por ello, Unity ofrece la posibilidad de mostrar e interactuar con objetos tridimensionales de forma totalmente programable.

Unity cuenta con la ventaja de poder exportar el proyecto a distintas plataformas mediante un click, bien sean sistemas operativos tradicionales como Windows, Mac y Linux, o bien plataformas móviles como Web, Android y iPhone OS.

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

3.3. Generación de ficheros externos

3.3.1. Modelado 3D: Blender

Dado que simularemos en un entorno tridimensional, necesitaremos una malla 3D que represente el conjunto de edificios. Para construirla, necesitamos usar 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 menús y controles de usuario.

Entre programas como 3DS Max, Maya y Lightwave nos decidimos por Blender por ser gratuito y multiplataforma, además de ser muy completo y ofrecer todas las funcionalidades que buscamos.

Blender es mantenido por BlenderFoundation y se distribuye como código abierto.

3.3.2. Intercambio de datos: JSON

Desde el momento en el que intervienen múltiples lenguajes de programación en un proyecto, es indispensable definir un formato de intercambio de datos común. De esta manera, un programa puede exportar sus resultados en dicho formato de manera que el siguiente programa los entienda y utilice fácilmente.

Actualmente, los dos formatos más utilizados son XML y JSON. Por ser mucho más simple e inteligible, nos decantaremos por el formato JSON. Necesitamos librerías propias de cada lenguaje que nos faciliten la creación y extracción de datos desde un archivo JSON. Usaremos Gson de google para Java y JsonObject [JSONObject] para Unity.

3.3.3. 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 donde crear gráficas de funciones en tiempo de ejecución.

Existen librerías propias de Java como JFreeChart y GRAL, pero por su falta de flexibilidad se han descartado y se ha decidido 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.

Vistas las herramientas necesarias, en la siguiente sección entramos en la implementación de las partes del proyecto empezando por el modelado del mercado

distribuido de energía.

4. Modelado del mercado

Esta sección cuenta con dos partes diferenciadas. Primero, en el apartado 4.1, se desarrollan las ideas del diseño de un nuevo modelo de mercado de energía. A continuación, se crea un entorno donde poder crear mercados distribuidos de energía. El usuario podrá elegir sus componentes y ajustar parámetros que afectarán a la simulación sobre el mercado.

Concretamente, en el apartado 4.2 se define el formato del fichero de entrada que contiene la localización de los prosumidores, en 4.3 se detallan algoritmos para crear un árbol de conectividad de la red a partir de las localizaciones. En 4.4, se explica la creación del entorno tridimensional donde el usuario podrá ajustar los parámetros del mercado. En 4.5 se describe el funcionamiento de los controles de usuario y por último en 4.6 se especifica el formato del fichero de salida.

4.1. Modelado de un mercado distribuido

Dado que se utilizará el algoritmo RadPro como resolvedor de problemas CEAP, el mercado ha de ajustarse a sus requerimientos. El mercado será un árbol dirigido (P, E) donde cada elemento $p \in P$ representa un prosumidor y cada elemento $(p, p') \in E$ indica que los prosumidores p y p' están unidos físicamente mediante cableado eléctrico, siendo p el origen y p' el final del enlace.

En la figura 4 se detalla el diagrama de modelo del mercado.

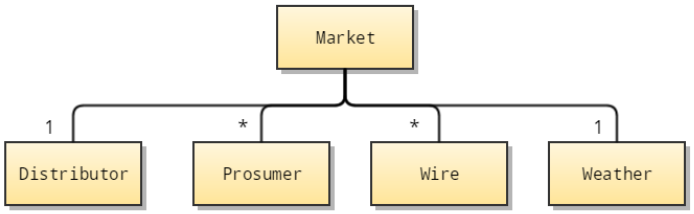


Figura 4: Diagrama de modelo del mercado

Además del árbol de conectividad, el mercado cuenta con un entorno consistente en una distribuidora y el tiempo meteorológico.

La distribuidora se encuentra conectada con cada prosumidor y actúa como un participante más del problema, vendiendo y comprando energía con los prosumidores. El precio de compraventa de la distribuidora es el mismo para todos los prosumidores, pero va variando cada hora. Las ofertas que los prosumidores lanzan al mercado deberán tener en cuenta el precio de la distribuidora para ser competitivos: un prosumidor deberá vender energía de forma más barata que la distribuidora. Por otra parte, aparece la posibilidad de intentar comprar energía de forma más barata que la distribuidora intentando acordar un intercambio entre prosumidores.

El tiempo meteorológico refleja la cantidad de nubes y la velocidad del viento que hay en cada momento. Esto varía el funcionamiento de los generadores de los prosumidores, aumentando o disminuyendo su eficiencia. Los datos del tiempo se actualizan cada hora a partir de un fichero de datos meteorológicos de Barcelona y su valor se interpola linealmente entre hora y hora.

En la figura 5 se muestran las componentes eléctricas con las que cuentan los prosumidores.

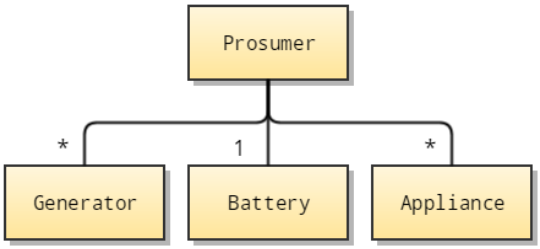


Figura 5: Diagrama de modelo de un prosumidor

Como se puede observar, un prosumidor cuenta con una batería y un número indefinido de generadores y dispositivos. Los generadores, como placas solares o aerogeneradores, producen energía a lo largo del tiempo; mientras que los dispositivos, como lavadoras, cocina o televisión, consumen energía en un periodo de tiempo.

La oferta de compraventa del prosumidor va variando según el estado de sus componentes. Generalmente hablando, si su gasto esperado en el siguiente tramo de tiempo es mayor que lo que espera producir, su oferta priorizará comprar energía, mientras que si se encuentra en el caso opuesto, podrá permitirse vender energía o guardarla en la batería para utilizar o venderla más adelante. Los detalles sobre la creación de las ofertas de compraventa se encuentran en el apartado 5.6.

4.2. Formato de entrada: Localización de los prosumidores

Aprovechando que la visualización será en tres dimensiones, usaremos como nodos del mercado mallas 3D: conjuntos de vértices, aristas y caras que forman figuras tridimensionales. En el caso de este problema, las mallas normalmente tendrán forma de edificio.

Los grandes programas de modelado como Blender o 3DS Max permiten exportar objetos tridimensional a ficheros formateados. Por ser un formato abierto, la entrada de datos del programa será un único archivo en formato OBJ que cumpla con las siguientes restricciones:

- Contendrá un único modelo con varios submodelos, cada uno representando un prosumidor del mercado.

- Cada submodelo tendrá su base en el plano $\{(x, y, z) \in \mathbb{R}^3 \mid z = 0\}$.

Se utilizará como posición del prosumidor el centro de masa del submodelo, es decir, la media aritmética de las coordenadas de sus vértices, proyectado sobre el plano $\{(x, y, z) \in \mathbb{R}^3 \mid z = 0\}$.

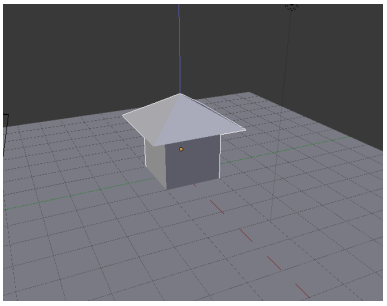


Figura 6: El punto interior indica el centro de masas de un submodelo. Se tomará como posición la intersección del eje vertical con el plano

4.3. Algoritmos de creación del árbol de conectividad

El siguiente paso es crear la red que une los prosumidores. Dados diversos puntos sobre un plano, hay muchos algoritmos que permiten unirlos todos creando un grafo. Sin embargo, por las limitaciones del algoritmo RadPro, solo nos sirven aquellos que den como resultado un árbol, esto es, un grafo acíclico. Podemos diferenciar los algoritmos que crean árboles dependiendo de la información de la que parten: (i) Información completa: se conoce de antemano la posición de todos los puntos, o (ii) Información incompleta: 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 caso de conocer la posición de todos los puntos nos encontramos con el conocido 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. No obstante, 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, sin embargo, podemos plantearnos esta variación del problema:

Dado un conjunto de puntos dentro de un espacio euclídeo, cuál es el mínimo camino que conecta todos los puntos, pudiendo añadir nuevos nodos si fuera necesario. Este problema es conocido bajo el nombre de Steiner Tree Problem y ha sido muy estudiado por sus enormes aplicaciones en la creación de redes. La solución consiste en encontrar los puntos llamados Steiner points, que cumplen curiosas propiedades:

- Como máximo existen tantos Steiner points como puntos 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.

Como era de esperar, el problema es NP-Hard y su solución es difícilmente alcanzable a partir de mercados que superen los 100 nodos.

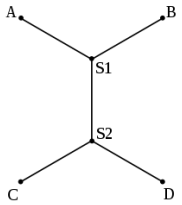


Figura 7: La solución para cuatro puntos radica en la creación de dos Steiner points

4.3.2. Información incompleta: Sequential nearest neighbour

En el caso anterior se consideraba que se conocían la posición de todos los puntos que formarían el grafo. Sin embargo, si nos fijamos en la realidad, esta idea no es práctica, ya que a lo largo del tiempo se van incorporando nuevos consumidores al mercado, añadiendo nuevos puntos al problema.

Para resolver este problema, debemos ser capaces de crear un árbol añadiendo nodos secuencialemente, preservando siempre la estructura de árbol. En la figura 8 se muestra el diagrama de flujo del algoritmo Sequential Nearest Neighbour que soluciona el problema.

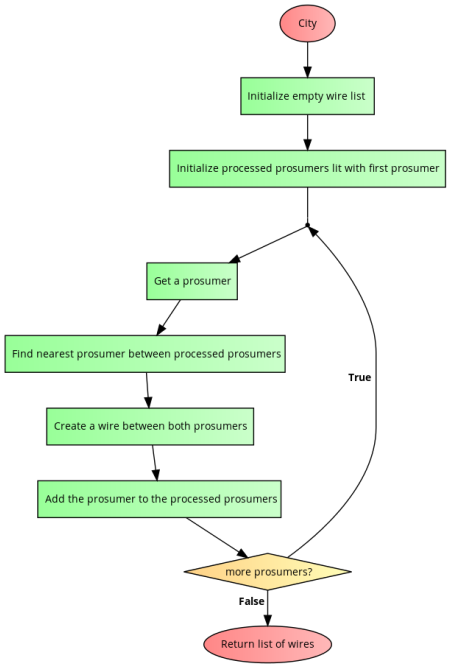


Figura 8: Diagrama de flujo del algoritmo nearest neighbour

El algoritmo recorre cada punto secuencialmente y lo une con el punto más cercano dentro de aquellos puntos que ya han sido procesados. Tiene complejidad cuadrática respecto a la cantidad de nodos. Por ello, es aplicable en grafos enormes sin problemas.

4.4. Entorno tridimensional de ajuste de parámetros

En este apartado se analiza la creación de un entorno tridimensional donde el usuario pueda personalizar los parámetros de la simulación. La figura 9 presenta un posible entorno ya creado.

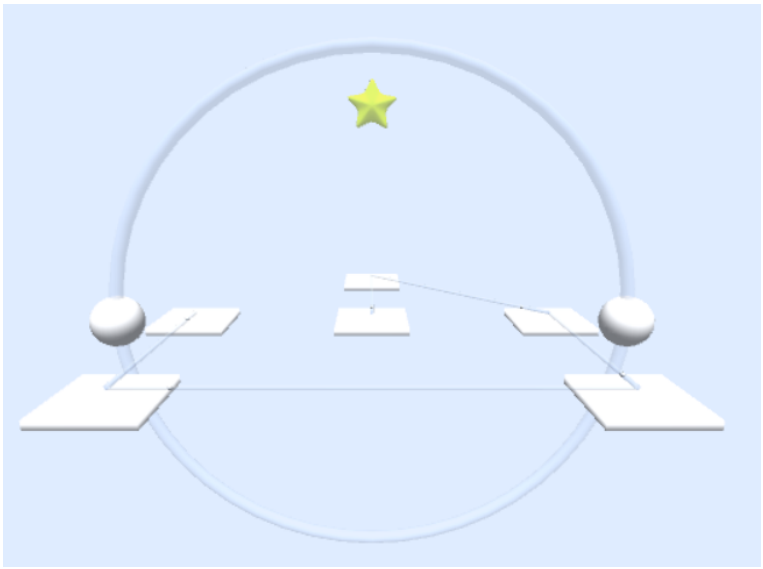


Figura 9: Entorno de ajuste de parámetros

Como se puede observar, el entorno cuenta con tres objetos principales: La malla 3D del mercado, la red de conectividad y un reloj de simulación.

- **Malla 3D del mercado**

  Importar una malla 3D en tiempo de ejecución en Unity es más complicado de lo que pueda parecer. El primer problema es abrir un cuadro de diálogo que permita al usuario elegir un archivo. Debido a que Unity usa Mono, que es un subconjunto de .NET FrameWork 2.0 especializado, no cuenta con la opción de crear ventanas propias del SO. La solución es añadir un DLL propio de .NET FrameWork 2.0 como plugin al proyecto. De todas maneras, el soporte no es completo y a veces salen avisos de seguridad ignorables al intentar abrir archivos.

  Por otra parte, se necesita un parseador de ficheros OBJ para extraer la información del archivo y crear una malla 3D. El proyecto utiliza el plugin OBJReader [OBJReader] para este propósito. Una vez se haya creado la malla, se hacen los cálculos del punto de gravedad y la posición sobre el plano de cada submodelo para situarlos correctamente.

- **Red de conectividad**

  Los cables se modelan con un cilindro translúcido que contiene una esfera dentro. La esfera representa la electricidad que fluye por el cable y va recorriendo el cable desde el prosumidor de origen hasta el prosumidor de destino.

  A la hora de la ejecución, el algoritmo de creación del árbol de conectividad devuelve una lista de pares de prosumidores a unir. Por cada par, se deberá crear una instancia del cable y situar y orientarlo de manera adecuada utlizando cálculos matemáticos apropiados.

- **Reloj de simulación**

  El reloj de simulación se trata de un *torus* fino orientado verticalmente con dos esferas sobre él.

  Lo único a tener en cuenta a la hora de la ejecución es la escala de la malla 3D del mercado, ya que el tamaño del reloj deberá ser siempre mayor que el mercado de manera que lo rodee.

4.5. Controles de usuario

Dentro del entorno de ajuste de parámetros, se añaden controles de usuario intuitivos que permitan ajustar parámetros del mercado y su simulación. En particular se pueden personalizar los perfiles de prosumidores, las capacidades de cableado y la franja horaria de la simulación.

4.5.1. Perfil de los prosumidores

El usuario puede seleccionar un perfil predefinido para cada prosumidor: Alto consumo, consumo medio o ahorrador. Estos perfiles afectan a la cantidad de generadores y dispositivos de consumo con los que contará el prosumidor.

Cuando el usuario pasa el ratón por encima de un prosumidor, éste se coloreará de diferente color dependiendo del perfil que tenga asignado. Por defecto, todos los prosumidores tienen asignado el perfil de consumo medio, que viene representado por el color naranja. Mientras se mantenga el ratón sobre el prosumidor, el usuario podrá elegir el perfil deseado pulsando las teclas 1, 2 y 3 del teclado, siendo 1 el de menor consumo y 3 el de mayor.

Por último, si se pulsa la tecla P, todos los prosumidores 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 del mercado.

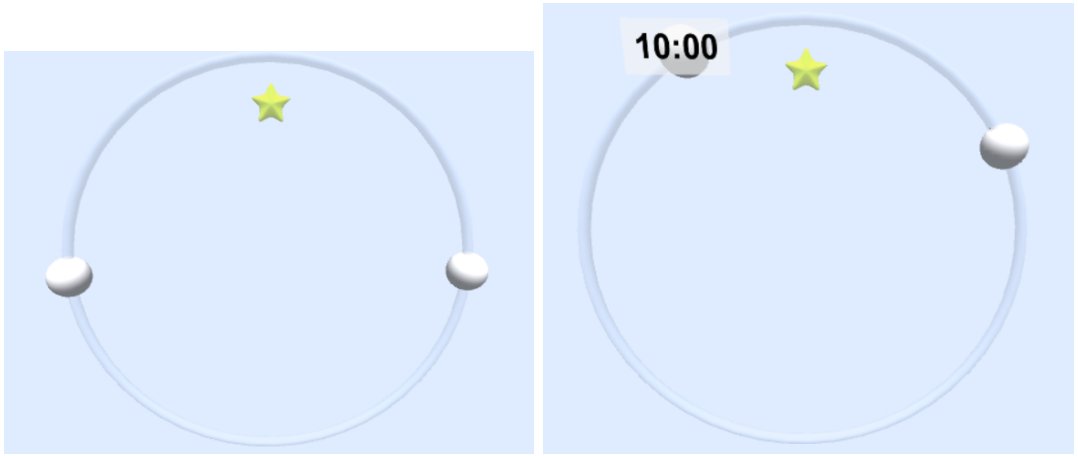
4.5.2. Capacidad del cableado

La capacidad de cada cable puede ser personalizada situándose encima de la misma y moviendo la rueda del ratón. Si el movimiento es de abajo a arriba, la capacidad aumenta y en caso contrario disminuye. El usuario puede observar los cambios ya que la anchura del cable se ajusta a la capacidad actual del mismo como se representa en la figura 10.

Por defecto, la capacidad de todos los cables empieza en 10 y varía entre 10 y 25 unidades.

4.5.3. La franja horaria

Se permite seleccionar la franja horaria en la que se ejecutará la simulación. Para ello, se deslizan las esferas situadas sobre el reloj, considerando que es un reloj solar,

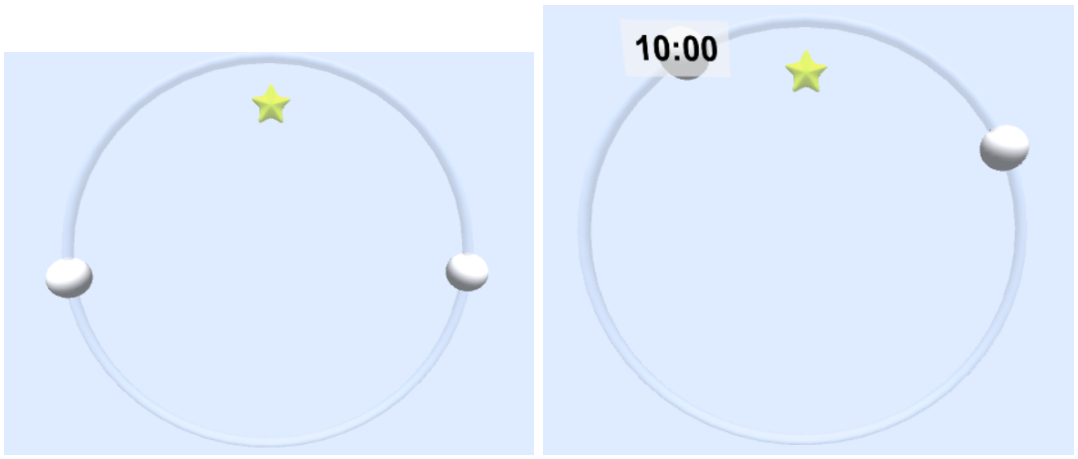


(a) Capacidad por defecto del cableado

(b) Aunque se aumente la capacidad, la esfera mantiene su tamaño

Figura 10: Ajuste de la capacidad del cableado

es decir, está dividido en 24 partes cada una representando una hora del día. El usuario percibe la hora que está ajustando mediante un panel que se revela mientras la esfera está en movimiento. Las figura 11 ilustra este efecto.



(a) Por defecto la franja horaria es de 06h a 18h

(b) Al mover las esferas se revela la hora que señalan

Figura 11: Ajuste de la franja horaria

4.6. Formato de salida: Mercado especificado

Se utilizará un documento JSON para exportar los datos recopilados durante la especificación del mercado. El formato se detalla en la figura 12.

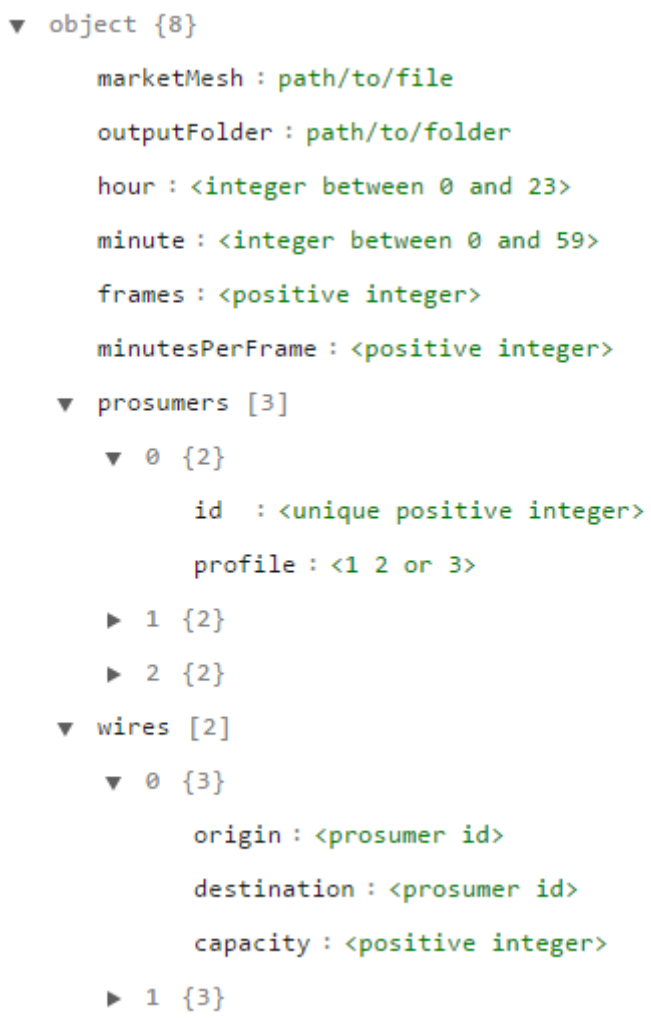


Figura 12: Formato del archivo JSON de salida

En esta sección, hemos visto como importar mallas 3D desde un fichero OBJ y dotarlos de una red de conectividad a medida, de manera que el usuario pueda controlar intuitivamente los parámetros de la simulación. En el próximo capítulo se explica el proceso de simulación y la generación del archivo de resultados.

5. Simulación

La simulación es un proceso en el que no interviene el usuario. Para entender su funcionamiento, en la figura 13 se muestra su diagrama de clases simplificado.

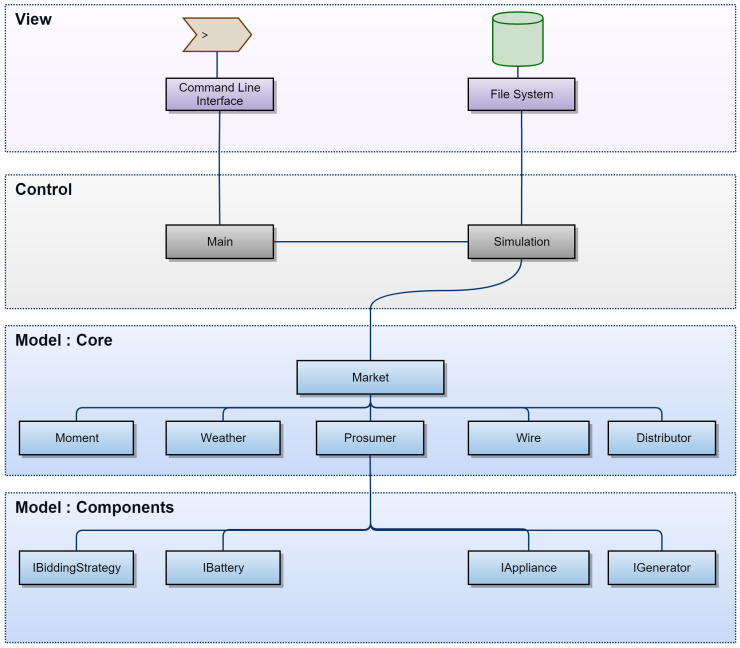


Figura 13: Diagrama de clases del simulador

En el apartado 5.1 se indican los principios de diseño seguidos en la implementación del simulador. A partir del apartado 5.2, se utiliza un apartado para explicar cada paquete del programa: 5.2 habla del paquete vista, 5.3 del paquete de control, y el modelo está dividido en 3 paquetes: en 5.4 se introducen las distribuciones temporales, un tipo de dato esencial para comprender el resto del modelo, el núcleo del modelo se explica en el apartado 5.5, y las componentes de un prosumidor en el apartado 5.6. Finalmente, en el apartado 5.7 se especifica el formato de salida de los resultados.

5.1. Principios de diseño

5.1.1. El 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. El 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 propia clase. De esta manera, el usuario de la clase se despreocupa de la representación interna de los datos y se centra únicamente en entender como interactuar con sus métodos. Esto previene que los datos sean modificados de forma incontrolada añadiendo robustez y mantenibilidad al proyecto.

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

- Una clase `Persona` tiene como atributo su `edad`.
- Desde una clase externa, se desea 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`.
- La alternativa que sigue el patrón experto sería crear un método `esMayorDeEdad()` que internamente evaluara `edad >= 18`.
- Observamos que por un lado, la responsabilidad sobre validar 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. Podría incluso no existir ningún atributo y que el resultado dependiera de un proceso interno.

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

5.1.3. Interficies y extensibilidad

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

Se permite guardar referencias a objetos a nivel de interficie, de manera que cualquier clase que la implemente, podrá ser asignada a la referencia, sin importar la implementación concreta.

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

Por último, las interficies proporcionan gran extensibilidad al proyecto. Basta crear una nueva clase que implemente una interficie 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 uno se representa en una clase propia que cuenta 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 contenida 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 ofertas de compraventa 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 script de gnuplot'
  ```

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 las órdenes de gnuplot en un fichero de script ubicado en la ruta especificada.
- Guarda los resultados de la simulación en un archivo JSON en la ruta especificada.

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 `Simulation` donde se encuentra el bucle principal del programa. La simulación utiliza el resolvedor de problemas `CEAPSolver` para resolver el problema CEAP a cada frame.

5.3.1. Main

En la figura 14 se observa el diagrama de la clase principal.

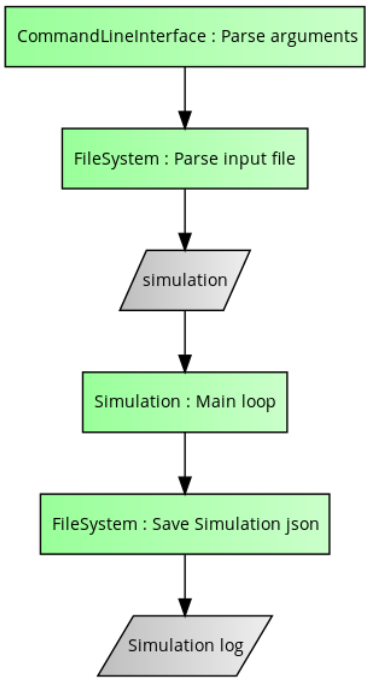


Figura 14: Diagrama de flujo de la clase principal

Primero se parsean los parámetros recibidos por la consola utilizando la clase `CommandLineInterface`. Si la ruta del archivo dado es correcta, se llama a `FileSystem` para generar una instancia de simulación. A continuación se ejecuta el bucle principal de la simulación, que se analiza en profundidad en el siguiente apartado. Por último, se crea un archivo JSON que contiene la información generada por la simulación.

5.3.2. Simulation

El bucle principal del programa se describe en la figura 15.

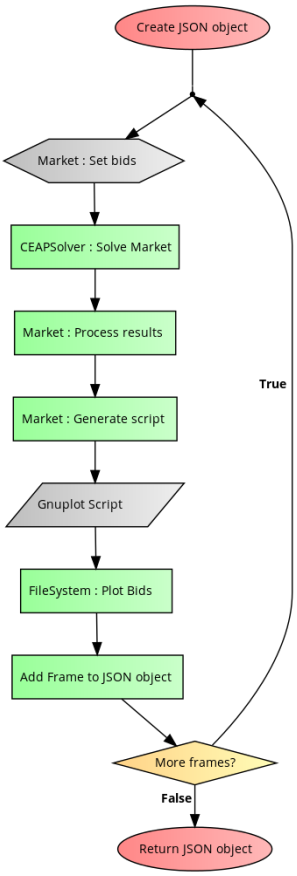


Figura 15: Diagrama de flujo de una simulación

Por cada frame de la simulación, se indica a cada componente del mercado que se sitúe en el momento adecuado. A continuación, se resuelve el mercado, encontrando los intercambios de energía que maximizan el beneficio total. Los prosumidores procesan los intercambios realizados y se genera el script de Gnuplot con las órdenes para dibujar la gráfica de cada prosumidor en este frame. Se llama a `FileSystem` para iniciar el proceso de Gnuplot y se guarda el estado del mercado en el objeto JSON de salida. Este objeto es devuelto como resultado de la simulación y será guardado en un fichero por la clase `Main`.

5.4. El paquete model.data

Antes de describir la jerarquía del modelo de datos en modo *top-down*, introcimos un tipo de dato esencial para comprender el resto del modelo: `ITemporalDistribution`.

5.4.1. ITemporalDistribution

Las distribuciones temporales pretenden modelar variables que van cambiando de valor a lo largo del tiempo. La cantidad de nubes en el cielo o el gasto energético de una lavadora son ejemplos de variables de este tipo. Además, esta clase proporciona una serie de facilidades que son útiles a la hora de manipular datos que dependan del tiempo. La figura 16 detalla la interficie.

```
// Devuelve el valor en un momento dado
public double getValue(Moment moment);
// Devuelve la media de los valores que coge la variable entre dos momentos
public double getMeanBetween(Moment since, Moment until);
// Devuelve la agregación de todos los valores que coge la variable entre dos momentos
public double getAddedValue(Moment since, Moment until);
// Devuelve el avance que lleva la variable en el momento dado respecto a su dominio de vida
public double getProgress(Moment moment);
```

Figura 16: Interficie de una distribución temporal

Existen dos implementaciones de la interficie:

- `DiscreteTemporalDistribution`

  La implementa extendiendo la clase `TreeMap<Moment, Double>`. Contiene un conjunto discreto y ordenado de entradas `Moment:Double` que representan medidas realizadas en un momento concreto. Los valores se interpolan linealmente entre dos medidas consecutivas para suavizar los saltos de valores.

- `ContinuousTemporalDistribution`

  La implementa utilizando dos `UnaryDoubleOperators`: uno describe la función que modela el valor de la variable a lo largo del tiempo y el otro es una primitiva suya. De esta manera, el valor agregado de todos los valores que coge la variable entre dos momentos se traduce a la integral definida de la función principal entre ambos momentos, la cual es fácilmente calculable gracias a la conocida regla de Barrow.

$$\forall F \in \int f : \int_a^b f = F(b) - F(a)$$

5.5. El paquete model.core

En el paquete core se encuentran los objetos más importantes que intervienen en la simulación, aquellos que conforman el modelo de mercado.

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 `Prosumer` y un conjunto `Wire` que modelan el mercado distribuido del problema. Además, existe un `Distributor` que está conectado a cada prosumidor y puede intercambiar energía con ellos y un `Weather` que controla el tiempo meteorológico, lo cual afecta a la eficiencia de los generadores.

Tiene tres funcionalidades principales:

- `setBids`: Sitúa el mercado en un momento concreto. Actualiza el tiempo que hace, el precio de la distribuidora y a continuación pide a cada prosumidor que realice su oferta de compraventa acorde a la nueva información.
- `processResults`: 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 acuerdos para que éstos los procesen.
- `generateScript`: 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 prosumidor. La escritura de las funciones a dibujar se delega a `IBiddingStrategy` dado que su expresión depende de la implementación elegida. La figura 17 muestra el formato general del script.

```
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 <acuerdo>, 0 to <acuerdo>, f_<i>(<acuerdo>) front
set arrow ...
// 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) <personlización de
pintado>, f_2(x) ...
```

Figura 17: Formato del script de Gnuplot

5.5.2. Wire

Guarda conjuntamente la información de un cable: El prosumidor de origen, el prosumidor de destino, la capacidad máxima de 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 una `DiscreteTemporalDistribution`. Gracias a ello ofrece facilidades, como obtener la cantidad media de nubes o la velocidad media del viento que ha habido en un periodo de tiempo.

5.5.4. Distributor

Los distribuidores tienen un `IBiddingStrategy` que indica la oferta de compraventa que hacen en todo momento. A diferencia de los `Prosumer`, esta oferta va variando según la hora en la que se encuentre la simulación, independientemente de cómo hayan sido los intercambios hechos en tandas previas. Cuentan con variables de tipo `ITemporalDistribution` que indican cómo 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

Modelamos los prosumidores como entes que tienen componentes eléctricas y presentan una oferta de compraventa dependiendo del estado de los mismos. Como se observa en la figura 18, cuentan en concreto con una batería y un número indefinido de generadores y dispositivos.

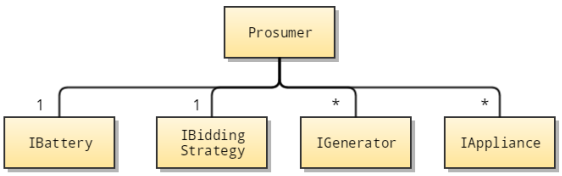


Figura 18: Diagrama de clases de un prosumidor

El siguiente apartado se dedica a explicar cada una de las componentes en detalle.

5.6. El paquete model.components

El último paquete que queda por analizar son las componentes de un prosumidor. Todas ellas cuentan con una interficie, de manera que si en un futuro se hiciera una implementación más profunda y compleja pudiérase integrar al proyecto sin problemas. Las interficies son minimales, se permite añadir más funcionalidades en las implementaciones, pero para poder utilizarlas se deberá castear la referencia de la interficie a la implementación concreta.

5.6.1. IBattery, IGenerator e IAppliance

Las figuras 19, 20 y 21 muestran respectivamente las interficies de las baterías, generadores y dispositivos, y cuentan con comentarios que explican la funcionalidad esperada de cada método.

```
// Revela la capacidad total de la batería
public double getCapacity();
// Revela el nivel actual de batería
public double getLevel();
// Intenta cambiar el nivel de batería una cantidad determinada
public void changeLevel( double amount );
```

Figura 19: Interficie de una batería

Como es de esperar, el nivel de la batería deberá mantenerse por debajo de su capacidad siendo siempre mayor igual a cero.

```
// Devuelve la energía producida entre los momentos preveídos teniendo en cuenta la meteorología
public double getGeneration( Moment since, Moment until, Weather weather );
```

Figura 20: Interficie de un generador

El programa cuenta con `SolarGenerator` y `EolicGenerator` que implementan la generación de forma distinta.

```
// Devuelve la energía requerida entre los momentos proveídos
public double getConsum( Moment since, Moment until );
// Devuelve el estado actual del dispositivo
public ApplianceState getState();
// Devuelve el momento en el está programado el inicio del dispositivo
public Moment getStartingTime();
// Los dispositivos pueden encontrarse en tres estados
public enum ApplianceState { Waiting, inExecution, Ended }
```

Figura 21: Interficie de un dispositivo

Cabe notar que la implementación de `getGeneration` como de `getConsum` utilizan variables de tipo `ITemporalDistribution` para calcular el resultado.

5.6.2. IBiddingStrategy

Por último, presentamos una interfaz importante de cara a futuras ampliaciones: `IBiddingStrategy`.

```
// Actualiza la oferta de compraventa dependiendo de las componentes del prosumidor
// y el precio de la distribuidora
public void setBid( Moment moment, IBattery battery, IDistributor distributor,
                    ArrayList<IGenerator> generators, ArrayList<IAppliance> appliances);
// Linealiza la función de compraventa para quel problema sea resoluble mediante el CEAPSolver
public PiecewiseLinearValuation toPLV();
// Procesa los intercambios que se han efectuado el última tanda
public void processResults( HashMap<Double, TraderType> trades );
// Escribe las órdenes de Gnuplot en el fichero dado para crear la gráfica de la oferta de compraventa
public void writePlotData( String plotFile, PrintWriter writer );
// Se diferencia entre dos tipos de intercambios: efectuados con particulares o con la distribuidora
public enum TraderType{ House, Distributor }
```

Figura 22: Interficie de una estrategia de apuestas

Para entender mejor el funcionamiento de las ofertas de compraventa, en el próximo apartado, se detalla una implementación de la interficie.

5.6.3. Implementación de IBiddingtrategy: LogBid

Primero extraemos las propiedades comunes que debe cumplir cualquier oferta de compraventa. La lógica de mercado indica que una oferta ha de ser creciente respecto a la cantidad comprada. Además queremos que la función que buscamos sea siempre inferior al precio de la distribuidora para que la oferta sea competitiva. Por último, es deseable que la curva sea ajustable mediante parámetros que permitan representar la necesidad del prosumidor en ese momento. Resumiendo, buscamos una función continua, creciente, acotada por una función lineal, la oferta de la distribuidora, y que pueda ser ajustada mediante parámetros.

Es bien conocido que las funciones logarítmicas son continuas, crecientes y acotadas por cualquier función lineal. De hecho, se cumple la desigualdad:

$$\log(x+1) \leq x$$

donde la igualdad se produce únicamente en $x = 0$.

Si suponemos que la función de la distribuidora tiene pendiente $k \in \mathbb{R}^+$:

$$k\log(x+1) \leq kx$$

Ahora, podemos añadirle un parámetro $a \in \mathbb{R}^+$ que siga preservando la desigualdad:

$$k\log(\frac{x}{a}+1) \leq \frac{x}{a}$$

Desarrollando, obtenemos una función con un parámetro que podemos utilizar para modelar la necesidad:

$$ak \log(\frac{x}{a} + 1) \leq kx$$

Podemos extender la desigualdad a los negativos, para modelar ventas, utilizando valores absolutos:

$$\left|ak \log(\frac{x}{a} + 1)\right| \leq |kx|$$

Tanto a como k como el logaritmo de un valor mayor a uno son siempre positivos. Por la desigualdad triangular podemos acotar superiormente $\left|\frac{x}{a} + 1\right| \leq \frac{|x|}{a} + 1$:

$$ak \log(\frac{|x|}{a} + 1) \leq \text{sign}(x)kx$$

Por último, podemos controlar el punto de contacto de la curva con la función lineal aplicando una translación a la curva. Añadiendo un parámetro $b \in \mathbb{R}$:

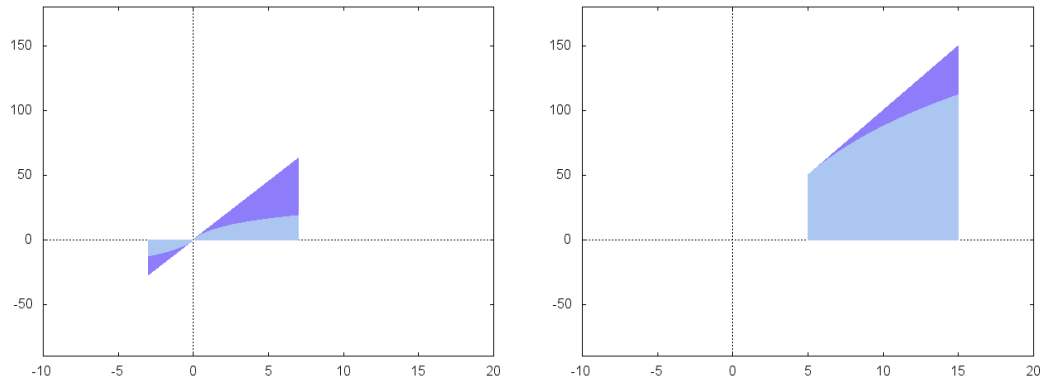
$$sign(x)k(b + a \log(\frac{|x - b|}{a} + 1)) \leq kx$$

el punto de contacto entre las curvas se translada al punto (b, kb).

Obtenemos la expresión utilizada por `LogBid` para expresar las ofertas de compraventa de los prosumidores:

$$sign(x)k(b + a \log(\frac{|x - b|}{a} + 1))$$

Las figura 23 muestra posibles diferentes formas que toma esta curva dependiendo de sus parámetros. La función pintada en morado representa la oferta de la distribuidora mientras que la que se encuentra pintada en azul se refiere a la oferta del prosumidor.



(a) El prosumidor puede comprar y vender energía: k=9, a=1, b=0

(b) El prosumidor únicamente compra energía: k=10, a=8, b=5

Figura 23: Ejemplos de funciones de compraventa

Observamos que estas curvas tienen un dominio delimitado $[lower, upper]$ que depende de la cantidad total de energía que puede vender o comprar un prosumidor.

Queda por explicar, cómo se decide actualizar el valor de estos parámetros dependiendo del estado de las componentes del prosumidor. En concreto debemos definir el dominio $[lower, upper]$ de la curva, obtener la pendiente lineal k, calcular la necesidad del prosumidor a y definir el punto de contacto b.

1. Lo primero es calcular la expectativa de balance de energía esperada de cara al siguiente frame. Para ello, se suma el valor de generación esperado de cada generador y se resta el consumo esperado de cada dispositivo en marcha.

$$expected = \sum_{gen} production(gen) - \sum_{app} consum(app)$$

2. El dominio de la curva indica cuánta energía puede llegar a intercambiar el prosumidor. El prosumidor debe abastecer su consumo esperado. Por ello, puede vender como máximo energía como tenga almacenada en la batería menos la expectativa de energía calculada. La cantidad máxima que puede comprar el prosumidor depende de la capacidad de la batería, una vez se haya aplicado la expectativa de energía. Esto es, el nivel actual de batería menos la expectación más la capacidad de la batería. O dicho de otro modo, el mínimo del dominio más la capacidad de la batería.

$$lower = battery.level - expected$$

$$upper = lower + battery.capacity$$

3. La pendiente de la función lineal es el precio de la distribuidora en ese momento.

$$k = distributor.rate$$

4. Para calcular la necesidad, se calcula el tiempo que falta para que empiece cada dispositivo en espera. La necesidad aumenta inversamente proporcional a los minutos que falten para el comienzo de cada dispositivo.

$$a = \sum_{app} \frac{k}{app.minutesToStart}$$

5. El punto de contacto representa la cantidad de energía que el prosumidor debe comprar necesariamente. Por este motivo, coincidirá con *lower* cuando éste sea positivo y valdrá cero en caso contrario.

$$b = max(lower, 0)$$

5.7. Formato de salida: Registro de simulación

En las figuras 24 y 25 se detalla el formato del fichero JSON de salida.

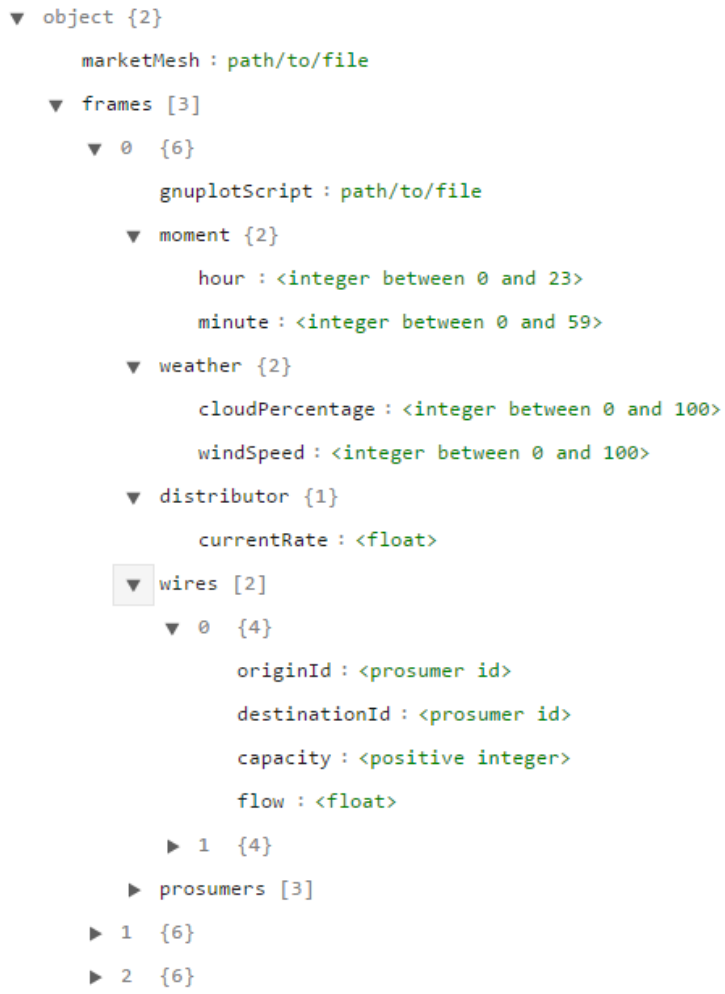


Figura 24: Formato del archivo JSON de salida

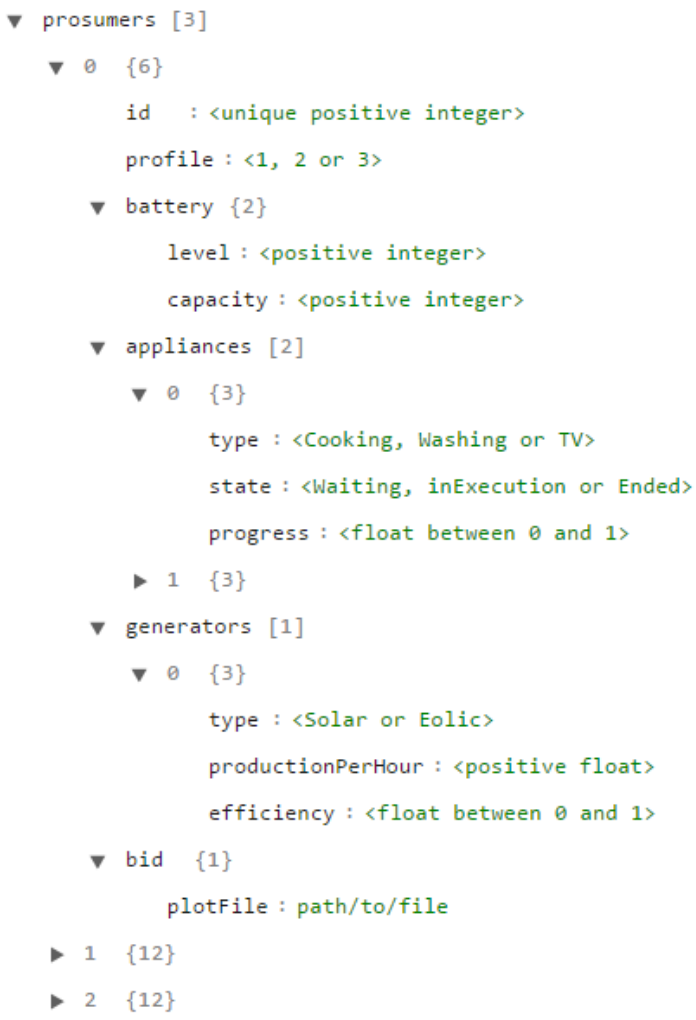


Figura 25: Formato de los prosumidores en el archivo JSON de salida

En la siguiente sección, nos adentramos en la tercera y última parte del proyecto: la visualización.

6. Visualización y producto final

La visualización se encarga de generar un entorno tridimensional donde poder analizar e interactuar con los resultados de la simulación. La estructura simplificada de las componentes que la conforman se encuentra en la figura 26.

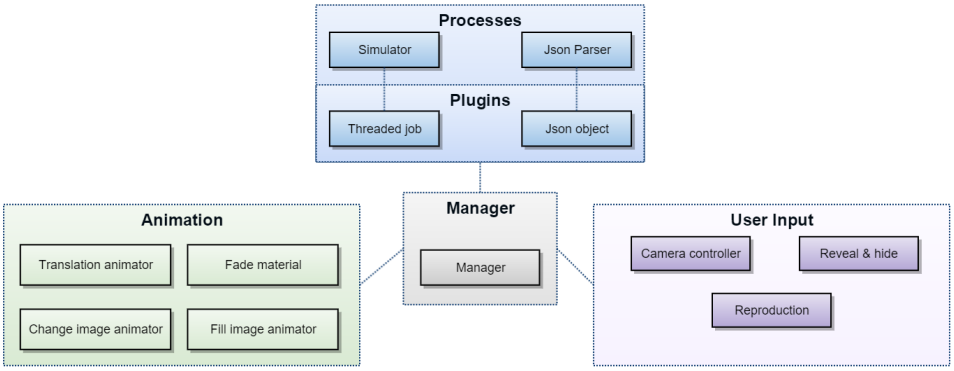


Figura 26: Diagrama de las componentes del visualizador

Manager es la componente que controla el flujo del visualizador y funciona como puente para unir las distintas componentes. El resto de componentes se dividen en tres bloques: procesos que se ejecutan de forma transparente al usuario, animaciones de los distintos objetos de la escena y componentes que controlan las interacciones on el usuario.

Esta sección comienza explicando los principios de diseño distintivos de Unity en el apartado 6.1. El apartado 6.2 se utiliza para describir los objetos que conforman el entorno de visualización. A partir de este momento se resume el contenido de cada bloque: el apartado 6.3 se encarga de los procesos transparentes al usuario; el apartado 6.4 explica cómo se animan los distinos elementos de la escena; y finalmente, en el apartado 6.5 se muestran los controles de los que dispondrá el usuario dentro del visualizador.

6.1. Principios de diseño de Unity

Desde el punto de vista de la programación, Unity es una librería más que cuenta con clases y métodos que facilitan la creación y el control de objetos comúnmente encontrados en juegos, como colisiones o la gravedad.

Esta librería se encuentra disponible en tres lenguajes de programación: C#, Java y Boo. Todos ellos son lenguajes dentro del paradigma Programación Orientada a Objetos. Sin embargo, el diseño de una aplicación en Unity se aleja de lo que uno normalmente espera de un entorno de POO.

La diferencia clave es que en Unity lo que se programa son comportamientos, en lugar de entes que constan de datos y métodos como en la programación usual.

Por ejemplo, consideremos una escena que consta de personas y perros. Se quiere lograr que las personas sean capaces de caminar por la escena y hablar con otras personas, mientras que los perros se limitan únicamente a caminar por la escena.

El diseño propuesto por el paradigma general de Programación Orientado a Objetos trataría de crear dos clases `Persona` y `Perro`, donde el primero contaría con los métodos `hablar()` y `caminar()` mientras que la clase `Perro` contaría con un método dedicado a implementar su manera de caminar.

La filosofía de diseño de Unity propone crear clases que definan comportamientos, en lugar de entes abstractos. En este ejemplo, se crearían dos clases `Caminar` y `Hablar` que implementarían cada uno de los métodos de manera genérica. Igual que las clases, estos comportamientos pueden recibir parámetros, como clips de animación, para modificar su funcionamiento dependiendo del objeto en el que nos encontremos. El resultado son dos tipos de objetos: Unos que cuentan con los comportamientos `Caminar` y `Hablar` y otros que cuentan únicamente con `Caminar`.

A los objetos que constan de una colección de comportamientos se les conoce con el nombre `prefab` en Unity, y son los objetos básicos para crear cualquier escena. Unity ofrece una variedad de componentes predefinidas listas para ser añadidas a los prefabs del proyecto. Si deseamos que las personas no choquen entre ellas, basta con añadir una componente `Collider` al `prefab` de persona. Si queremos que los perros se representen mediante una malla 3D, añadiremos la componente `Mesh Renderer` al prefab de perro, y así consecutivamente con todos los comportamientos.

Como en la mayoría de proyectos, se necesita normalmente un objeto que controle el estado del programa y decida qué opciones están disponibles al usuario en cada momento. En el presene proyecto, la componente `Manager` es la que se encarga de esta función.

La componente `Manager` sigue el patrón de diseño Singleton: se limita a una la cantidad de instancias que puedan existir de este objeto en la escena. Esto permite crear una variable estática dentro de la clase que referencia a la única instancia actual de la clase. Como resultado, se facilita enormemente la conexión entre las diferentes componentes del proyecto utilizando `Manager` como enlace, ya que cualquier comportamiento puede enviar sus datos al `Manager` utilizando esta referencia estática de `Manager`, que se encargará de hacer llegar el mensaje al destinatario especificado.

Estrictamente hablando, las componentes del bloque de procesos que se explican en el apartado 6.3 no siguen la idea de comportamientos, porque son simplemente algoritmos que se ejecutan de forma transparente al usuario. El resto de componentes son comportamientos que forman parte de prefabs. Analizaremos su funcionalidad en los apartados 6.4 y 6.5.

6.2. Entorno de visualización

Antes de analizar los comportamientos, analizaremos los nuevos objetos que se añaden a la escena cuando entramos al visualizador. Como se puede observar en la figura 27, comparado con el entorno de ajuste de parámetros, se elimina el reloj y se añaden dos nuevos objetos: paneles informativos de los prosumidores y el menú de control de reproducción.

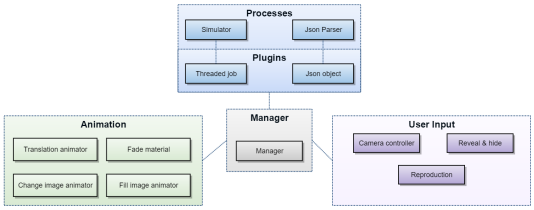


Figura 27: Entorno de visualización

- **Paneles informativos de los prosumidores**

  Una vez procesados los resultados de la simulación, se añade a cada prosumidor un panel imbuido en 3D que representa su estado a lo largo de la reproducción. En la figura 28 se encuentra una instancia del diseño del panel informativo.

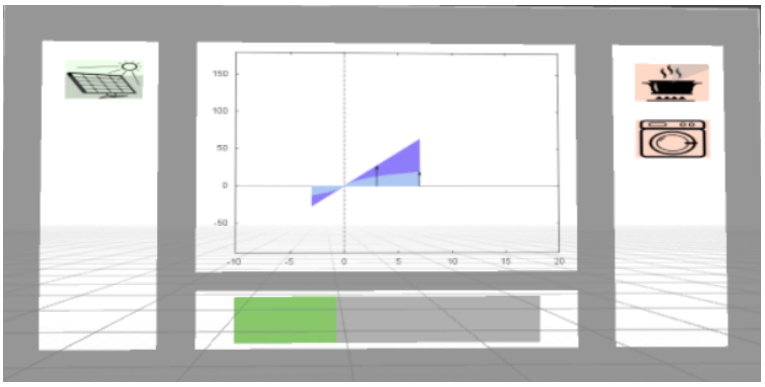


Figura 28: Diseño del panel informativo de un prosumidor

  En la barra de la izquierda se muestran los generadores que contiene la casa. Un icono [Icons] representa el tipo de generador del que se trata, y sobre la imagen se superpone un relleno que representa la eficiencia que está teniendo ese generador en ese momento en referencia al máximo rendimiento que podría tener.

  En la de la barra derecha se añaden los dispositivos. De manera similar a los generadores, están representados por un icono referente al tipo de dispositivo y sobre ellos se superpone una imagen que va rellenándose de forma horaria mostrando el progreso de completitud del dispositivo.

El panel central alberga la gráfica de la función de compraventa efectuada por el prosumidor, con flechas que indican los acuerdos realizados en la tanda anterior.

Por último, en el panel inferior se encuentra la batería, que también cuenta con una imagen que se rellena horizontalmente indicando el porcentaje de batería actual.

- **Menú de reproducción**

  El menú de reproducción sirve para controlar el tempo en el que se muestran los resultados de la simución. Su diseño se encuentra en la figura 29.

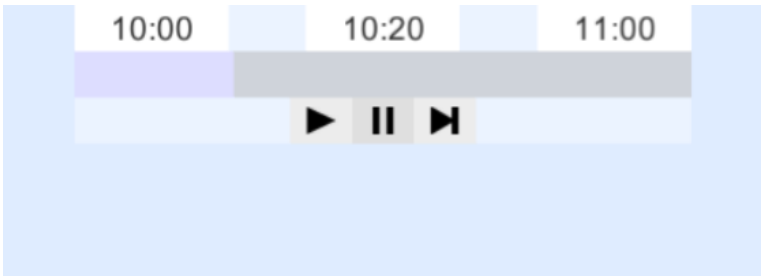


Figura 29: Diseño del menú de reproducción

Consta de tres paneles de igual altura situados uno encima del otro. El panel del primer nivel cuenta con tres textos: el primero revela la hora de inicio de la simulación; el segundo, la hora en la que se encuentra la reproducción y el tercero, la hora de finalización.

El segundo panel es una barra de progreso que se va completando mientras avanza la reproducción. El tercero cuenta con los controles de usuario. Concretamente, los botones de *play*, pausa y *step* que se explican en detalle en el apartado 6.5.

6.3. El bloque de procesado

Como se indica en los requerimientos del proyecto, el producto final ha de facilitar la ejecución secuencial de cada una de las partes. En este apartado explicamos cómo se ejecuta el simulador desde dentro del entorno de visualización y cómo se procesan los resultados para poder mostrarlos adecuadamente en pantalla.

- **Ejecución de la simulación**

  La figura 30 recuerda el entorno de ajuste de parámetros.

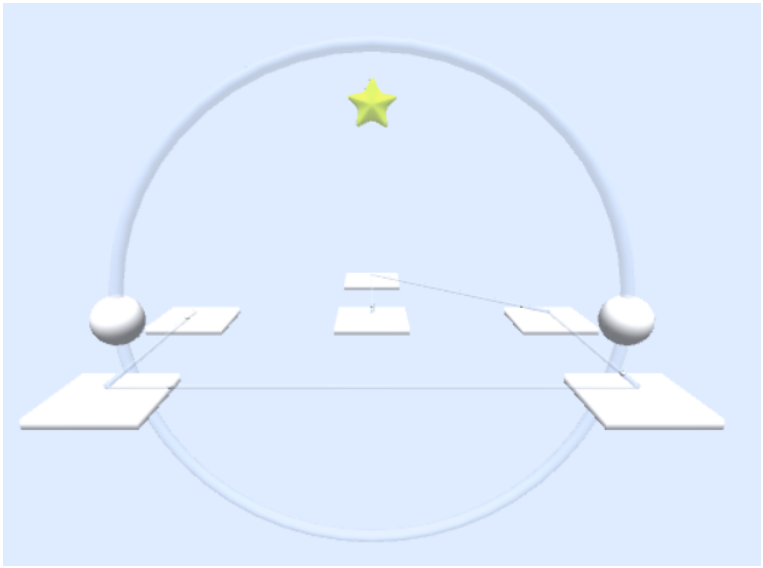


Figura 30: Entorno de ajuste de parámetros

Una vez el usuario esté contento con los parámetros de simulación elegidos siguiendo las indicaciones de la sección 4, puede pulsar sobre la estrellita que se encuentra sobre el mercado para inicializar el simulador.

Esto crea internamente un nuevo hilo que se encarga de correr la simulación. Mientras tanto, en la escena se muestra un icono de carga, pero sigue animándose como anteriormente. La creación del hilo es posible gracias a la clase `ThreadedJob` [ThreadedJob]. Cuando el hilo acaba, genera un evento que será tratado por el hilo principal y se autodestruye.

Este hilo se encarga de crear un `Process` para abrir una ventana de terminal y ejecutar el simulador con los parámetros proveídos. Cabe mencionar, que para su correcto funcionamiento, deberemos incluir el JAR distribuible del simulador en una ruta relativa al visualizador. De esta manera, el hilo será capaz de encontrar la ruta absoluta hasta el simulador y ejecutar el comando satisfactoriamente en cualquier plataforma.

- **Análisis de los resultados de la simulación**

  Los resultados de la simulación se guardan en un archivo JSON que contiene toda la información necesaria para replicar la simulación.

  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, la librería no se soporta. No obstante, existen librerías de JSON creadas especialmente para Unity. En el presente proyecto se utiliza la librería JSONObject [6].

El parseado del fichero se hace en dos fases. Primero se detecta con cuántos generadores y dispositivos cuenta cada prosumidor y se generan los paneles informativos acordemente. En la segunda fase, se coge la componente de animación de cada objeto de la escena y se popula con los datos del fichero.

Lamentablemente, Unity no permite hacer llamadas a su librería fuera del hilo principal, por lo que el parseado del fichero se debe hacer en el hilo principal. Esto provoca que la escena se congele hasta que se complete el proceso y pueda verse el resultado final.

6.4. El bloque de animación

La simulación contiene *frames* que indican el estado en el que se encuentra el mercado en un instante concreto. El objetivo es interpolar linealmente los valores que definen el estado del mercado para conseguir una animación continua y fluida.

Unity cuenta con dos sistemas por defecto para hacer animaciones:

- La componente `Animation` permite ejecutar clips de animación guardados previamente en una lista, y es la manera tradicional de conseguir animaciones en Unity.
- A partir de la verión 4, Unity incorpora la componente `Animator` que consta de una máquina de estados donde cada nodo es un clip de animación y cada arista define los parámetros de la transición entre dos clips.

Sin embargo, ninguna de las dos maneras permite aún crear y asignar clips de animación en tiempo de ejecución. Estos sistemas solo funcionan si los clips están guardados en ficheros ANIM y son referenciados correctamente en las componentes mencionadas. Dado que en el proyecto no podemos saber como serán las animaciones hasta que se haya ejecutado el simulador, crearemos nosotros un algoritmo sencillo que se encargue de simular las animaciones.

Definiremos dos modos de animación:

- **Modo repetitivo**

  La simulación se congela entre dos frames y se anima la evolución de un frame al siguiente una y otra vez. Esto permite analizar concretamente qué ha ocurrido entre estos instantes de tiempo.

- **Modo continuo**

  En el modo continuo, la animación va recorriendo todos los frames y vuelve a repetirse cuando haya llegado al final de la simulación.

La escena cuenta con diversos objetos que han de ser animados de forma distinta. Las imágenes sobrelapadas sobre los generadores, los dispositivos, la batería y el progreso total de la simulación en el menú de reproducción han de animar el

atributo llamado `fillAmount` de la componente `Image` que tienen. La componente `FillImageAnimator` es la que se encarga de esta función.

El panel donde se muestran la función de compraventa ha de actualizarse cada frame mostrando la nueva imagen. Esto lo controla la componente `ChangeImageAnimator`. Finalmente la componente `TranslationAnimator` se encarga de animar el recorrido que efectúan las esferas de electricidad según el flujo que recorre por su cable.

Todas estas componentes tienen como entrada un array de datos que se rellena a la hora de parsear el fichero JSON generado por la simulación y se basan en llamar repetidamente a una función de actualización cada cierto intervalo de tiempo que indique el nuevo valor que ha de tomar la variable en cuestión.

La componente `FadeMaterial` es diferente, ya que utiliza *lerping*, una técnica implementada internamente en Unity, para animar el material de un objeto desde opaco o translúcido a uno transparente y viceversa. Esto se utiliza para permitir al usuario controlar la visibilidad de los distintos elementos de la escena, como se detalla en los controles de usuario explicados en el siguiente apartado.

6.5. El bloque de interacción

La visualización de los resultados de la simulación es interactiva y cuenta con diversos controles que permiten personalizar lo que ve el usuario en todo momento.

6.5.1. Control de la cámara

El usuario cuenta con esta funcionalidad a lo largo de toda la ejecución del producto final. Permite controlar la posición y rotación de la cámara imitando los controles que se utilizan dentro del editor de Unity. Las teclas WASD controlan la posición de la cámara mientras que el movimiento del ratón controla la rotación mientras se tenga el click derecho presionado controla la rotación.

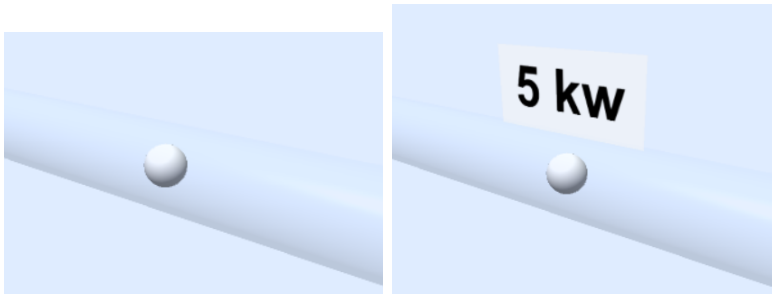
La implementación de esta funcionalidad se encuentra en `ControlCamera` que se basa en utilizar los métodos estáticos de la clase `Input` de Unity para detectar la entrada del usuario y recolocar y rotar la cámara utilizando cálculos matemáticos adecuados.

6.5.2. Revelar y ocultar información

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, varios objetos cuentan con la funcionalidad de revelar cierta información al hacer click o interactuar con ellas.

Por ejemplo, una vez se hayan procesado los resultados de la simulación, el usuario puede pulsar sobre un prosumidor para revelar u ocultar su panel informativo. Del mismo modo, si se clicka sobre las esferas de electricidad que viajan por los cables, se podrá mostrar o ocultar un panel donde se indica la cantidad de energía

que está fluyendo en el momento actual. En la figura 31 se ejemplifica este comportamiento.



(a) Por defecto, el flujo permanece oculto

(b) Al hacer click se muestra el panel

Figura 31: Ejemplo de revelación de información

Por otra parte, el usuario puede ocultar o revelar completamente todos los objetos que sean de un tipo concreto. Mediante la tecla `M`, de *market*, se controla la visibidad de todos los prosumidores, mientras que con la tecla `G`, de *grid*, podremos decidir si se muestra el árbol de conectividad o no.

6.5.3. Control de la reproducción

Por último, el visualizador cuenta con un menú para controlar la reproducción de la simulación.

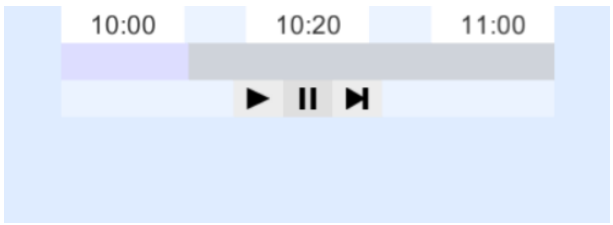


Figura 32: Diseño del menú de reproducción

Dada su importancia para poder analizar correctamente los resultados, este menú es siempre visible y se renderiza en la HUD del programa, ocupando la parte superior de la pantalla. Su comportamiento se define en `Reproduction`

Cuenta con tres botones interactuables, siendo los botones de *play* y pausa de tipo *toggle*. Mientras el botón de *play* esté desactivado, los elementos de la escena permanecerán estáticos. Si se activa, se recorrerán todos los *frames* consecutivamente reproduciendo la simulación completa.

Sin embargo, si el usario hace click sobre el botón de pausa, el transcurso de la animación parará y se centrará en animar la evolución desde el *frame* actual hasta el siguiente una y otra vez. Esto permite analizar concretamente qué ha ocurrido entre estos instantes de tiempo.

Una vez nos encontremos en estado de pausa, podremos avanzar al siguiente frame pulsando el botón de *step*.

Acabada la explicación de la implementación de las distintas partes del proyecto, la siguiente sección resume los propone posibles rutas de avance en el futuro.

7. Conclusiones y futuro trabajo

7.1. Conclusiones

Este proyecto consta de tres grandes aportaciones. Extender el modelo distribuido de energía para incluir extender la expresibilidad de los mercados distribuidos de energía incluyendo generadores baterías y dispositivos eléctricos a

7.2. Futuro trabajo

Los proyectos nunca se terminan. Siempre hay lugar para posibles ampliaciones y mejoras. De ahí que se llamen proyectos.

7.2.1. Open Street Map y CityEngine

El proyecto no se ha centrado en la creación de mallas 3D de mercados, pero hay mucho por investigar al respecto. 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 el corazón de Manhattan. En OpenStreetMap podemos exportar un fragmento de mapa en formato OSM, un formato parecido a XML, que contine información sobre calles, parcelas parques etc. que se encuentran en ella. Al ser de contribución abierta, cada vez alberga información más detallada incluyendo señales de tráfico incluso fuentes públicas de agua.

Una vez conseguido el trozo de mapa que nos interesa, basta usar un programa de modelado procedural para reproducir la ciudad entera en 3D. CityEngine es un programa comercial que soporta nativamente el formato OSM y es capaz de crear modelos 3D a partir del mapa extruyendo las parcelas de edificaciones y añadiéndoles materiales adecuados. Incluso permite modificar los detalles de los edificios pudiendo indicar su año de construcción o zona de edificación, bien sea rural, residencial o industrial.

Debido a las limitaciones del tiempo y el desorbitado precio del programa CityEngine, solo se ha experimentado utilizando la licencia de prueba de 30 días que ofrece la empresa.

7.2.2. OpenWeatherMap

Otra manera de obtener resultados más realistas es utilizar datos actuales de meteorología para modelar el tiempo que hace en el mercado.

OpenWeatherMap es una API pública que ofrece datos históricos y prediciones 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 repartidas a

lo largo del mundo y su uso va creciendo, superando ya las mil millones 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 a OpenStreetMap, podría usarse la longitud y la latitud del mapa de seleccionado para consulta 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 y consultar el formato de las queries en su documentación .

Los datos de meteorología que se utilizan en el presente proyecto se han extraído de OpenWeatherMap, pero son estáticos, no dependen del entorno real del mercado.

7.2.3. Estrategias de reinforcement learning

El proyecto actual ofrece una plataforma donde poder testear estrategias de bidding sobre un mercado de energía. Tal y como se describe en el artículo [10], se pueden implementar procesos de reinforcement learning de manera que los prosumidores vayan variando su oferta dependiendo de la información que vayan acumulando a lo largo del tiempo. Como resultado, se obtendrían agentes inteligentes, capaces de adaptarse al mercado y encargarse de gestionar todas las necesidades energéticas que pueda tener un usuario particular.

Reconocimientos

Me gustaría mostrar mi agradecimiento a las siguientes personas y entidades por proporcionarme material indispensable para la realización del proyecto:

[JSONObject] JSON file parser for Unity, Matt Schoen
`http://wiki.unity3d.com/index.php?title=JSONObject`

[OBJReader] OBJ file parser for Unity, StarScene software
`http://www.starscenesoftware.com/objreader.html`

[Icons] 2D Icons, made by Freepik at FlatIcon
`www.flaticon.com`

[ThreadedJob] Thread generator for Unity, user Bunny83 at UnityAnswers
`http://answers.unity3d.com/users/6612/bunny83.html`

Referencias

[1] European Technology Patform: Update of the SmartGrids SRA 2007 for the needs by the year 2035, 2012.

[2] J. Cerquides; G. Picard; J. A. Rodríguez-Aguilar: Designing a Marketplace for the Trading and Distribution of Energy in the Smart Grid, 2015.

[3] J. Cerquides; G. Picard; J. A. Rodríguez-Agilar: Defining and solving the energy allocation problem with continuous prosumers, 2015.

[4] D. Urieli and P. Stone: TacTex'13: A Champion Adaptive Power Trading Agent, 2013.