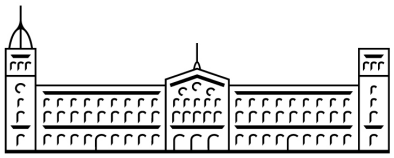


Trabajo final de grado

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,	25 de enero de 2016

Índice

1. Introducción y antecedentes

La energía es una necesidad básica en nuestra sociedad. Practicamente 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 y consumo de energía del planeta, las sustancias fósiles van decrementando en abundancia, lo cual hace cada vez más costoso encontrarlo y distribuirlo, y pone al planeta potencialmente en una situación donde no haya suficiente materia 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, como la energía solar o la energía eólica.

  La energía nuclear genera grandes cantidades de energía, pero representa un riesgo de irradiación radioactiva 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 otra alternativa son las energías renovables. La ventaja de estas fuentes es que son inagotables mientras haya luz solar o viento sobre el planeta por ejemplo, además de no presentar posibles riesgos y ser 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 llega el punto en el que un usuario particular puede permitirse comprar y mantener sus propias placas fotovoltaicas para abastecer 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 innecesarios. 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 de la red eléctrica. Según [referencia], alrededor de dos terceras partes de la energía producida se pierden en el reparto, al disiparse en forma de calor. La alternativa es crear mercados distribuidos de energía donde los

usuarios puedan intercambiarse energía entre sí, con él objetivo de minimizar los gastos de distribución.

Las Smart Grid [referencia] unen la idea de que a partir de ahora, los usuarios de la red podrán también producir energía con la idea de utilizar un mercado distribuido, para ofrecer un modelo de mercado intelegente en el que cada usuario podrá intercambiar (bien comprar o vender) energía con otros usuarios. A estos usuarios se les dota el nombre de prosumidores, ya que producen y consumen energía simultaneamente.

Al momento de considerar un modelo distribuido de mercado surgen nuevas preguntas a responder como: ¿Cómo se indica cuánta energía necesita cada usuario?, ¿Como 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 una función de oferta como:

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

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 con quién puede intercambiar energía cada usuario. A la hora de resolver el problema cada prosumidor intercambia energía con los prosumidores directamente conectados con él. No obstante, puede ocurrir, que la energía intercambiada con un prosumidor adyacente tenga como destinatario final un destinatario externo.

Por ejemplo: Consideremos un mercado de tres prosumidores A, B y C conectados secuencialmente. En un momento de intercambio, el prosumidor A vende 4 unidades de energía a su vecino B el cual al mismo tiempo decide vender 4 unidades de energía a un tercer prosumidor C que no es vecino de A. Dado que todos los intercambios se producen instantaneamente, el efecto final del intercambio es que el prosumidor A ha vendido 4 unidades de energía a C. En particular, el prosumidor B deberá proporcionar la posibilidad de transferir esa cantidad de energía, bien porque desea tanto comprar y vender esas cantidades de energía o bien porque permite transferir libremente la energía definiendo el valor de la función f en el 0 como 0.

Por otra parte, se le añade a cada cable del mercado 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 de entrada, 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 la red.

1.3. Algoritmos de resolución para mercados distribuidos de energía

Existen programas que resuelven el problema de la liquidación de energía simplificando alguno de sus aspectos. En [referencia], se define el problema EAP como el problema de liquidación de mercado donde las funciones f de los prosumidores son funciones discretas. Se da un mapping para plantear el problema de manera que se pueda resolver mediante programación lineal entera utilizando resolvedores comerciales como CPLEX y Gurobi. Tamién se presenta el algoritmo RadPro: Un método resolvedor basado en transpaso de mensajes que únicamente se aplica sobre grafos acíclicos, pero ofrece una gran mejora de eficiencia en estos casos.

A partir de entonces se ha ampliado el problema a $CEAP$ [referencia] que permite que las funciones f sean funciones lineales definidas a trozos. En el presente proyecto se usará la librería de Java [nombre] creada por [referencia] donde se incluyen la implementación de los diversos resolvedores.

La pretensión del proyecto es ampliar 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 cada prosumidor vayan variando dependiendo del estado de sus componentes. Por último, se da 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 define formalmente el modelo de mercado distribuido ampliado que in-

cluye las componentes eléctricas. En la quinta sección se explican los pasos que se siguen para ejecutar una simulación de un mercado distribuido ampliado mientras que en el sexto 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 los casos de uso establecidos en 2.2.

2.1. Requerimientos

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 **mecansimos de ampliación** que permitan facilitar futuras modificaciones.
- 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 más predominantes del mercado.
  - El producto final será un **ejecutable autocontenido** para que no requiera configuración alguna para poder ser ejecutado.

2.2. Casos de uso

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

- **Especificación del mercado**

  Se trata de una aplicación interactiva en el cual se genera un mercado a partir de la localización de los prosumidor proporcionada. 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 simlación.

- **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.

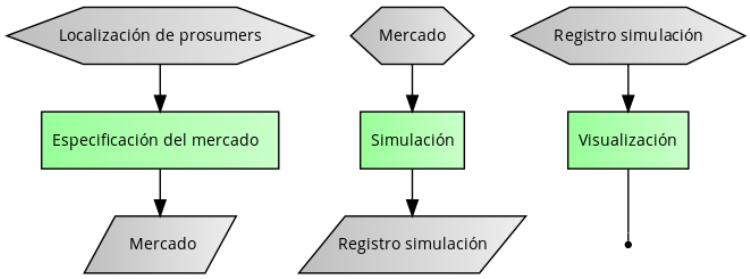
- **Visualización del mercado**

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

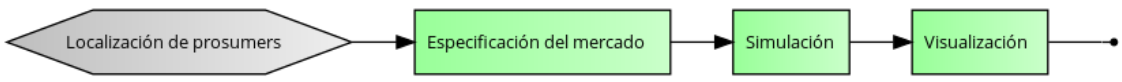
Las entradas y salidas de cada parte son un único fichero formateado que contiene toda la información relevante.

Diagramas de casos de uso:

Cada parte ha de ser ejecutable indendientemente del resto:



El producto 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 explican las 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 programas equivalentes y citaremos a sus autores así como la licencia bajo la que se distribuyen.

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 pasadas 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. Entre Git y mMercurial se ha utilizado Git por experiancia previa del autor con el software.

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

3.1.2. 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 variedad de herramientas de desarrollo 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 proyeto NTS, de New Typesetting System, una reimplementación de TeX en Java, con la intenión 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 robusteza 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.

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 del proyecto se dividen en dos tipos: Diagramas de modelo donde se contienen mucha informaciión A parte del texto, el proyecto cuenta con diversos diagramas que facilitan su comprensión. Para generar diagramas de modelo se ha utilizado la cuenta gratuita de la herramienta online gliffy.com. Para los diagramas de flujo, se utiliza la herramienta online Code2flow que automáticamente convierte pseudocódigo en un diagrama de flujo.

3.2. Entornos de programación

3.2.1. Entorno de simulación: 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.

3.2.2. 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.

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 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.

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

3.3. Generación de ficheros externos

3.3.1. 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.

3.3.2. Intercambio de datos: JSON

Desde el momento en el que intervienen múltipls 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 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 librerias propias para esta función. Usaremos Gson de google para Java y JsonObject esrito por Matt Schoen de Defective Studios para Unity.

3.3.3. Trazado de gráficas: Gnuplot

Por último, tendremos la necesidad de representar cciertos 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.

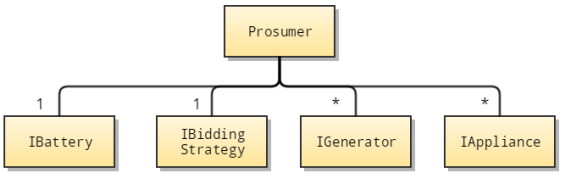
A continuación, se modela el mercado distribuido de energía sobre el cual se trabajará a lo largo del proyecto.

4. Especificación del mercado

En 4.1 se modela el mercado como árbol dirigido, ampliándolo con parámetros que permiten aumentar la expresividad de los prosumidores, en 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

El diagrama de modelo de un prosumidor es el siguiente:



Como se puede observar, un prosumidor cuenta con una batería y una estrategia de apuestas. Por otra parte puede contener cualquier número de generadores (productores de energía) y dispositivos (consumidores de energía).

4.2. Formato de entrada: Localización de los prosumidores

Para visualizar una simulación, lo primero que necesitmos 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 simplicitud 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.

La especificación del formato OBJ se puede consultar en: http://www.fileformat.info/format/wavef

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 ö ¡nombre¿ÿ 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.

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 FrameWrk 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.

4.3. Algoritmos de creación del árbol 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 nuesra 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 añadiendose a lo largo del tiempo y la red va ampliandose secuencialmente de manera que preserve la estructura de árbol.

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 e 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 maáximo existen tantos steiner points como nodos iniciales menos dos Cada steiner points 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 dificilmente 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.

4.3.1. 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 secuencialemnte 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 añadiendose de forma continuada y a cada paso la red es un árbol completo.

Diagrama de flujo:

4.4. Entorno tridimensional de ajuste de parámetros

Se necesita un entorno donde el usuario pueda personalizar los parámetros de la simulación. Aprovechando que la visualización de los resultados se hará en un entorno tridimensional, se ha implementado la capacidad de poder ajustar los parámetros de la simulación dentro del entorno de visualización.

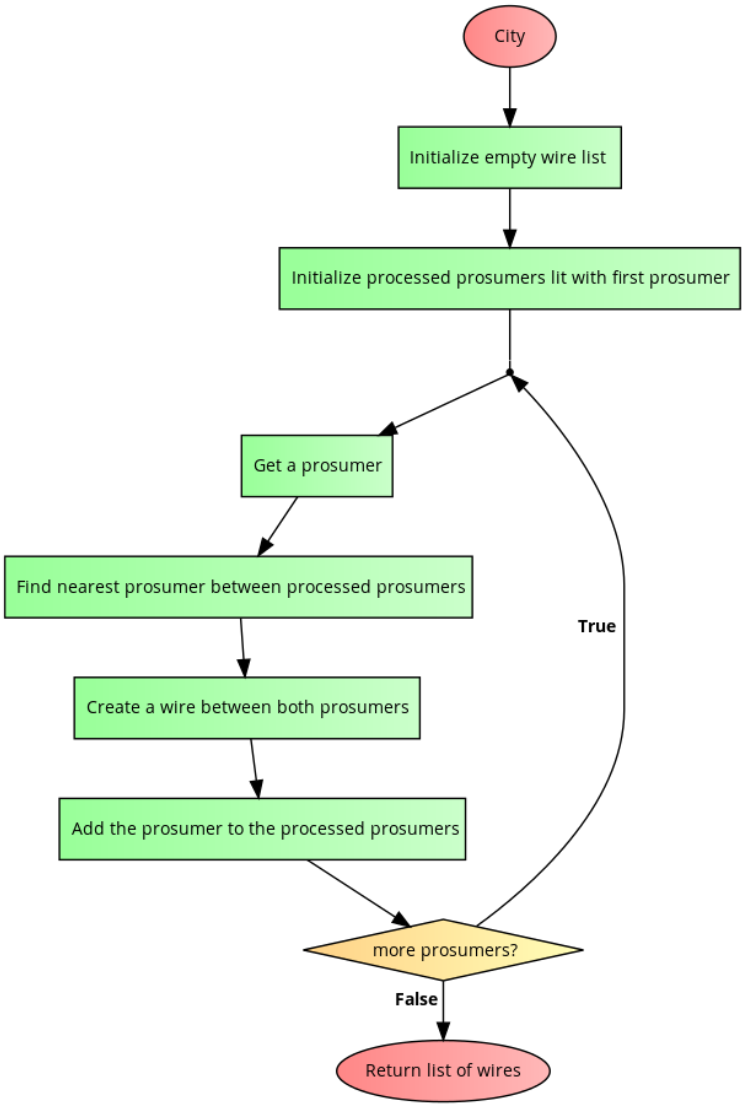
La construcción de las distintas partes de la escena es la siguiente:

- **Malla 3D del mercado**

  Se necesita un parseador de ficheros OBJ para extraer la información y crear una malla 3D. Existen plugins propios de unity que crean una componente Mesh a partir de un fichero OBJ. El proyecto utiliza el plugin ObjReader de Starscene Software (starscenesoftware.com) para este propósito. Se configura para que cada submalla sea un objeto propio en la escena que tenga como centro gravedad la media aritmética de sus vértices.

- **Reloj de simulación**

  A diferencia de la malla del mercado, el modelo del reloj no depende de la ejecuión del programa, por lo que se usa el editor de Unity para crear previamente un prefab con todas los comportamientos deseados. A La hora de

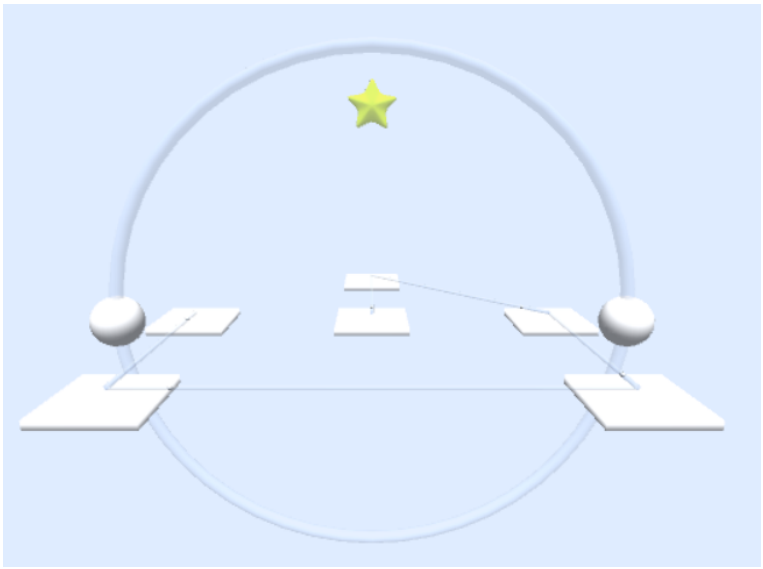


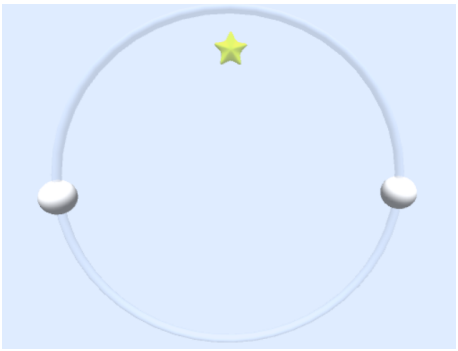
ponerlo en la escena basta con escalarlo a la medida de la malla del mercado para que quede rodeándoselo.

- **Red de conectividad**

  Los cables se modelan con un cilindro translúcido que contiene una esfera dentro. La esfera representa la energía que transcurre del cable y va recorriendo el cable desde el prosumer origen hasta el prosumer final. Se puede crear un prefab que contenga todos los comportamientos de un cable 3D.

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





4.5. Controles de usuario

4.5.1. 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 dondde 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).

4.5.2. Perfil de los prosumidores

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, está 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.

4.5.3. La franja horaria

Se permite 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 OnMouseDrag de Monobehaviour

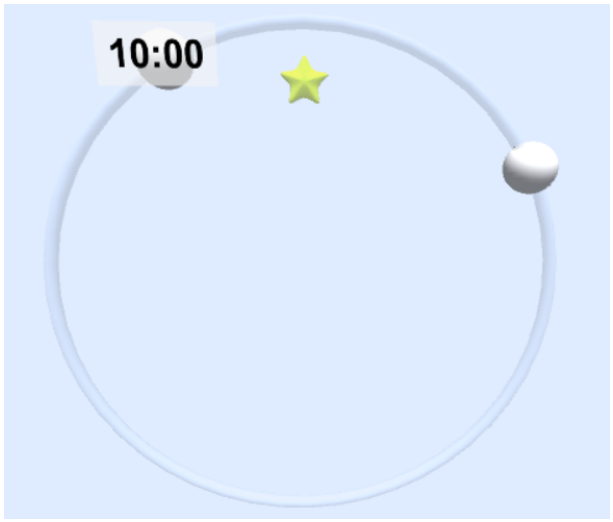
- Coger el punto que representa el ratón en la ventana

- 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.

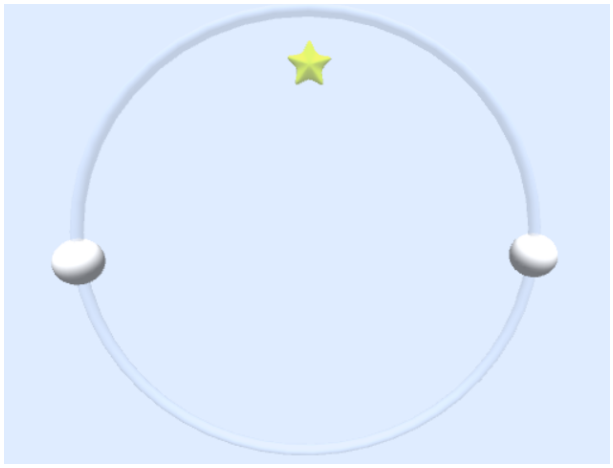
- escalarlo por el radio mayor del torus para situarlo encima del mismo

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.



4.5.4. Capacidad del cableado

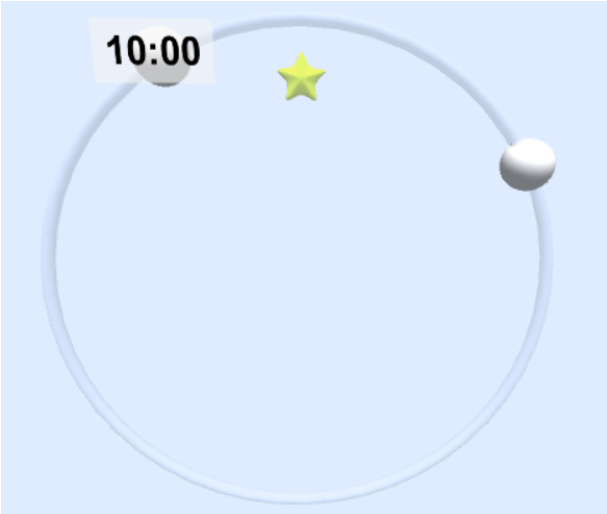
La capacidad de cada cable puede ser personalizada situándose encima de la misma y moviendo la rueda del ratón. Si el movimiento es de abajo a arriba, la capacidad aumenta y en caso contrario disminuye. El usuario puede observar los cmbios ya que la anchura del cable se ajusta a la capacidad actual del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



Pr defecto, la capacidad de todos lo cables empieza en 10 y varía entre 10 y 25.

4.5.5. Revelación/Ocultación de 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 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.

4.6. Formato de salida: Mercado especificado

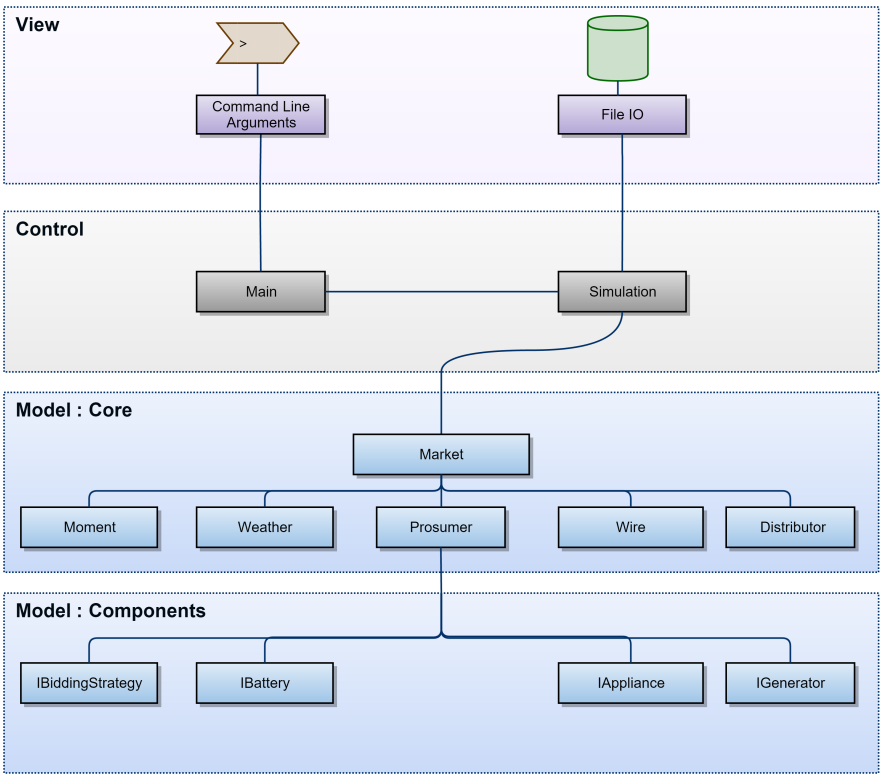
Se utilizará un documento json para exportar los datos recopilados durante la especificación del mercado. El formato es el siguiente:

El programa es capaz de crear modelos 3D desde un fichero .obj y dotarlo 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 del mercado utilizando los parámetros proveidos.

```
▼ object {8}
    marketMesh : path/to/file
    outputFolder : path/to/folder
    hour : <integer between 0 and 23>
    minute : <integer between 0 and 59>
    frames : <positive integer>
    minutesPerFrame : <positive integer>
  ▼ prosumers [3]
    ▼ 0 {2}
        id  : <unique positive integer>
        profile : <1 2 or 3>
    ▶ 1 {2}
    ▶ 2 {2}
  ▼ wires [2]
    ▼ 0 {3}
        origin : <prosumer id>
        destination : <prosumer id>
        capacity : <positive integer>
    ▶ 1 {3}
```

5. Simulación

En esta sección explicaremos en detalle la parte de simulación. Para ello consideremos el siguiente diagrama:



El apartado 5.1 indica 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: 5.1 se habla del paquete vista, 5.2 del paquete de control, y el modelo está dividido en 3 paquetes: las distribuciones temporales que se explican en el apartado 5.3, el núcleo del modelo en el 5.4, y las componentes de un prosumidor en el apartado 5.5.

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 clsae 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 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 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. Interficies y extensibilidad

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 propocionando 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

El siguiente diagrama muestra el flujo principal:

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 prosumidores
- 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:

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 progresa en la que se situa el momento dado respecto al dominio de variabilidad de la variable.

Existen dos implementaciones principales de la interaz:

- 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 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 dependan 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 prosumidores 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 indicaa 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 hcha 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

El diagrama de modelo de un prosumer es el siguiente: Cada una de las componentes se explica en detalle en el siguiente apartado.

Para crear un prosumer se necesita proveer un identificador único y opcionalmente un perfil de prosumer. Si se proporciona un perfil, se le añadirán IAppliances y IGenerators preconstruidos automáticamente. En cualquier momento se le podrán añadir componentes llamando a los métodos adecuados.

Los prosumidores recalculan su IBidStrategy dependiendo de los tratos que hayan hecho en la tanda anterior. Para ello proporcionan la informacionan la información del resto de sus componentes (batería, generadores y dispositivos) y es la misma estrategía la que decide como actualizar la apuesta. Este proceso se discute al final de este capítulo, en el apartado 6.89.

5.6. El paquete model.components

El último paquete que queda por analizar son las componentes de un prosumer. Todas ellas tienen una interficie que implentan, de manera que si en un fuuro se hiciera una implementación más profunda y compleja de las mismas pudierase 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 siguientes imágenes muestran las interficies IBattery, IGenerator e IAppliance Una batería ha de ser capaz de revelar el nivel de de energía que guarda, revelar la máxima cantidad de energía que puede albergar y modificar su nivel en una cantidad establecida. Un generador ha de calcular la energía que produce en un intérvalo de tiempo. Ejemplo: SolarGenerator. Eficiencia y tal. Una appliance revela el estado en el que se encuentra y la hora de inicio y calcula la energía consumida La mayoría de los métodos presentes son autoexplicativos. Tanto getConsum como getGeneration utilizan ITemporalDistributions para calcular el resultado. Get startingTime devuelve el momento en el que empieza la appliance

5.6.2. IBiddingStrategy

Una IBiddingStrategy requiere la implementación de cuatro métodos. A continuación se explica la funcionalidad esperada de cada método y su implementación en la estrategia LogBid del proyecto.

- setBid() : Es donde se actualiza la expresión que define la apuesta a partir de la información del entorno.
- toPLV() : Una PLV es un función lineal definida a trozos. El resolvedor de mercados CEAP recibe como entrada PLV por lo que es necesario linealizar en trozos la apuesta para poder resolver el mercado.
- setTrades() : Una vez se haya resuelto el mercado se reciben los tratos que se hanaceptado para poder procesarlos internamente
- wirtePlotData() : Añade las órdenes de Gnuplot necesarias para dibujar la gráfica que representa la apuesta en un escritor de ficheros poporcionado.

5.6.3. Implementación de IBiddingtrategy: LogBid

La lógica de mercado dice que una apuesta ha de ser creciente respecto a la cantidad comprada. Además queremos que la apuesta siempre sea inferior al precio de la distribuidora para ser competitivos. Por último, si la curva depende de parámetros, podrá ser ajustada a las necesidades del prosumer en ese momento.

Buscamos una función continua, creciente, acotada por una función lineal (precio de distribuidora) que pueda ser ajustada mediante dos parámetros: La cantidad que necesita comprar el prosumer y la necesidad de energía del prosumer.

Es bien conocido que una función logarítica es continua, creciente y acotada por cualquier función lineal. De hecho, se cumple la desigualdad:

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

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

Aplicado a una función lineal de pendiente k:

$$klog(x+1) \leq kx$$

Podemos añadirle un parámetro $a > 0$ que siga preservando la desigualdad:

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

Y utilizar este parámetro para modelar la necesidad. Desarrollando obtenemos una función con un parámetro :

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

Podemos extender la desigualdad a los negativos utilizando valores absolutos:

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

Como tanto a como k y el logaritmo de un valor mayor a uno son positivos

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

Por último, la cantidad mínima de compra controla el punto de contacto entre nuestra función y la lineal. añadamos un parámetro b que controle el punto de contacto:

$$k * (b + a * log(\frac{|x-b|}{a}+1)) = kx \leftrightarrow x = b$$

Y obtenemos la expresión de la curva LogBid: $k * (b + a * log(\frac{|x-b|}{a}+1))$ que cumple con los criterios que buscábamos.

Para actualizar la expresión dependiendo del estado de prosumer deberemos calcular el dominio de la función, el precio del distribuidor, y la necesidad del prosumer. El cálculo de estos valores se explica en el siguiente diagrama:

- Por cada generador, se suma el valor esperado que va a producir
- Por cada dispositivo, se suma el valor esperado que va a consumir
- La expectativa es la resta entre estos dos valores
- El mínimo del dominio es la expectativa menos el nivel de batería
- El máximo del dominio es el mínimo más la capacidad total de la batería
- El precio del distribuidor se consigue consultando el precio actual del mismo

- Por cada dispositivo que esté a la espera, se calcula el tiempo que falta para que empiece
- la necesidad aumenta proporcionalmente al precio de la distribuidora e inversamente proporcionalmente a los minutos que falten por comenzar

La linealización de la curva se hace dividiendo el dominio en n puntos equidistantes (n controla la precisión de la linealización) y se encuentra la recta que pasa por dos puntos consecutivos. Cada una de estas recas es un ILV que alunirlas todas se obtiene un PLV.

En writePlotData se escriben las órdenes de gnuplot siguiendo el formato especificado en 6.89

5.7. Formato de salida: Registro de simulación

El archivo de salida es un archivo JSON que tiene el siguiente formato:

El objeto prosumidores cuenta con el siguiente formato:

A ontinuación analizaremos la tercera parte del proyecto: La visualización.

6. Visualización y producto final

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 resolvedor 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.

6.1. Principios 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 cdad 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.

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.

6.2. Diseño de los paneles de prosumidor

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 to da 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, cisualizaremos 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 sepas 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 las 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.

6.3. Ejecutar la simulación

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 metodo update de ThreadedJob, este actualizará su estado y en caso de que haaya 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 proveidos: 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 distribuible, 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 ejcutable ,exe. Por eso situaremos el jar dentro de esta carpeta y la aplicación será distribuible.

6.4. Cargado de simulación ejecutada previamente

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 adptación 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. Nosotro usaremos JSONObject de ¡referencia¿.

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 bine 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.

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.

6.5. Controles de usuario

El control de la cámara y las teclas de H y G que revelan los prosumidores y el cableado respectivamente, se heredan de la primera parte del proyecto. Además se añaden las siguiente funcionalidades:

6.5.1. Revelar/Ocultar paneles de prosumidor y flujos de energía

Una vez se hayan procesado los resultados de la simulación, cada prosumidor cuenta con un panel informativo con toda la información del estado. Este panel se sitúa justo encima de cada prosumidor y permanece invisible por defecto. El usuario puede controlar su estado de visibilidad haciendo click sobre el prosumidor deseado.

Por otra parte, las esferas que circulan por los cables representando la energía, cuentan con otro panel invisible por defecto que muestran la cantidad de energía que se está transfiriendo cuando el usuario hace click sobre ellas. Para ocultar esta información, basta con clickar la esfera de nuevo.

6.5.2. Controlador de la reproducción

El usuario puede controlar la reproducción de los resulados mediante este submenú que aparece en la visualización. Dada su importancia para poder analizar correctamente los resultados, este menú es siempre visible y se renderiza en en la HUD del programa, ocupando la parte superior de la pantalla.

Cuenta con tres textos que representan la hora de inicio, la hora actual y la hora final de la simulación, una barra que va acumulando el progreso de la simulación y tres botones interactuables:

- El botón play es un toggle que controla si los elementos se animan o no
- El botón pause es un toggle que pausa la reproducción mientra está activo, mostrando únicamente la evolución entre dos frames consecutivos para poder analizar mejor ese momento.
- El botón siguiente solo puede pulsarse cuando pause está activado y avanza la simulación al siguiente intérvalo de tiempo.

6.6. Animación de los resultados

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 máuina 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 correcamente 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.

A continuación, mostraremos posibles rutas de avance para este proyecto.

7. Futuro trabajo

7.1. Open Street Map y 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 formator .osm muy parecido a xml que contine 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 lso 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 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 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 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 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 útimo el proyecto proporciona una plataforma donde poder testear estrategias de bidding en un mercado de energía. Tal y como se describe n el artículo

[referencia], 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.

Reconocimientos

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

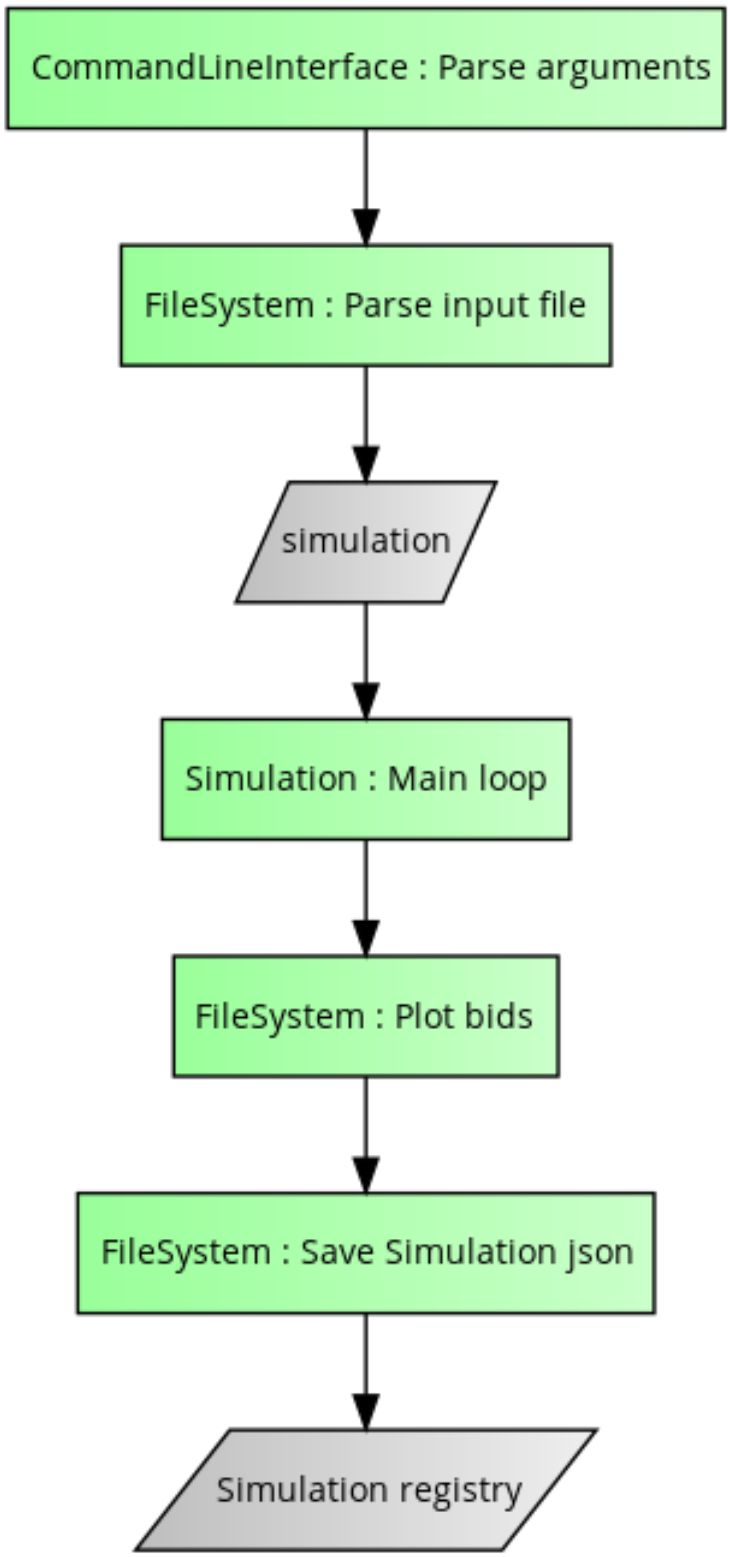
- Icono 2D de la visualización: Icons made by Freepik (www.freepik.com) and Google ("http://www.flaticon.com/authors/google") from Flaticom (www.flaticon,com) licensed by Creative Commons BY 3.0 ("http://creativecommons.org/licenses/by/3.0/")
- Efecto de cargando: LoadingEffect by OneManArmy (http://armedunity.com/user/1-onemanarmy/)
- 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

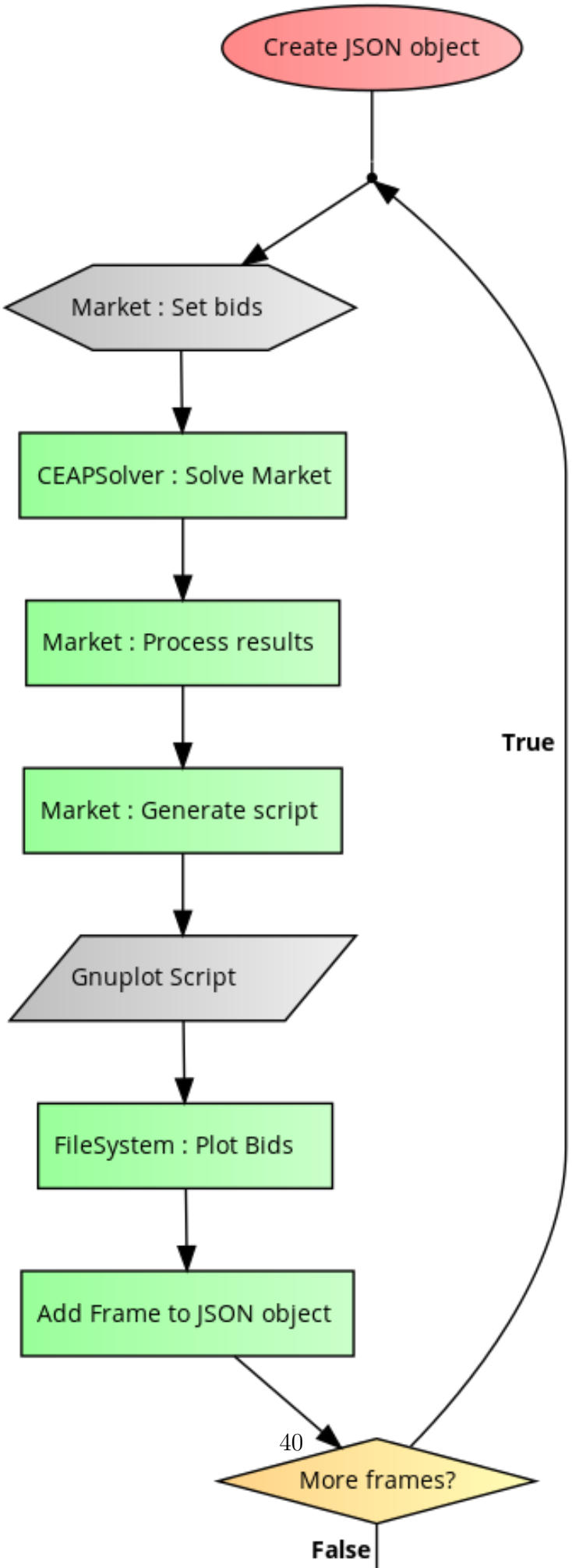
Referencias

[1] RadPro

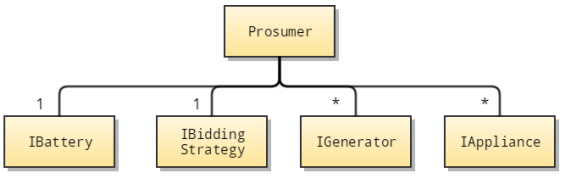
[2] Reinforcement learning

[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.


CommandLineInterface : Parse arguments
FileSystem : Parse input file
simulation
Simulation : Main loop
FileSystem : Plot bids
FileSystem : Save Simulation json
Simulation registry


Create JSON object
Market : Set bids
CEAPSolver : Solve Market
Market : Process results
Market : Generate script
Gnuplot Script
FileSystem : Plot Bids
Add Frame to JSON object
More frames?
True
False

```
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 ...

// 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) ...
```



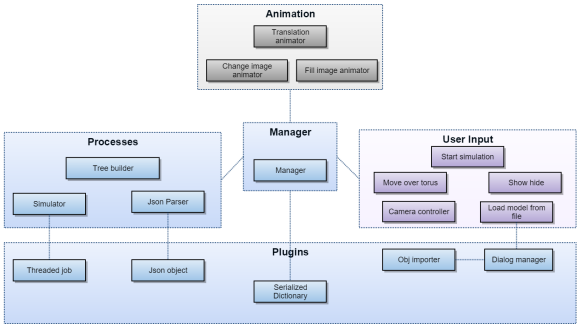

```
public double getCapacity();
public double getLevel();
public void changeLevel( double amount );
```

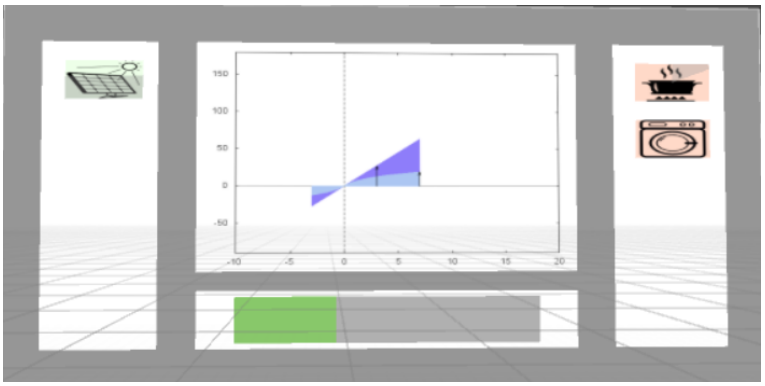
```
public void setStartingMoment( Moment moment, Weather weather );
public double getGeneration( Moment since, Moment until, Weather weather );

public enum GeneratorType
{
   Solar, Eolic
}
```

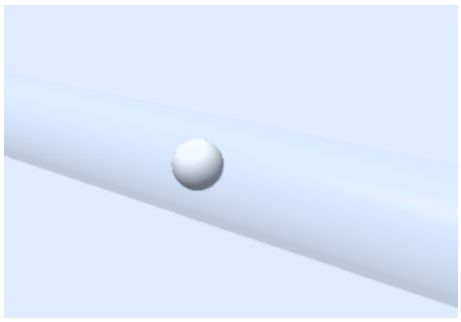
```
public void setStartingMoment( Moment moment );
public double getConsum( Moment since, Moment until );
public ApplianceState getState();
public Moment getStartingTime();

public enum ApplianceState
{
   Waiting, inExecution, Ended
}
```

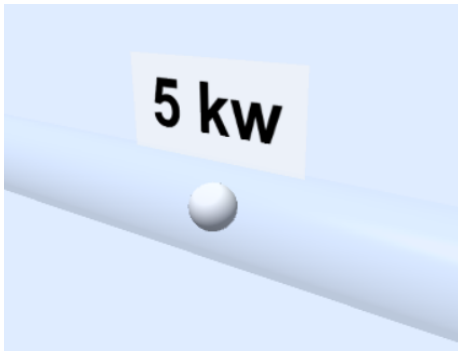
```
▼ object {2}
    marketMesh : path/to/file
  ▼ frames [3]
    ▼ 0  {6}
        gnuplotScript : path/to/file
      ▼ moment {2}
            hour : <integer between 0 and 23>
            minute : <integer between 0 and 59>
      ▼ weather {2}
            cloudPercentage : <integer between 0 and 100>
            windSpeed : <integer between 0 and 100>
      ▼ distributor {1}
            currentRate : <float>
      ▼ wires [2]
        ▼ 0  {4}
              originId : <prosumer id>
              destinationId : <prosumer id>
              capacity : <positive integer>
              flow : <float>
        ▶ 1  {4}
      ▶ prosumers [3]
    ▶ 1  {6}
    ▶ 2  {6}
```

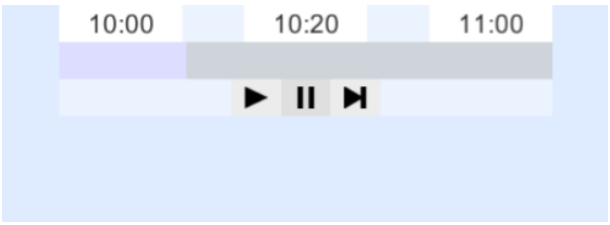
```
▼ prosumers [3]
  ▼ 0  {6}
      id   : <unique positive integer>
      profile : <1, 2 or 3>
    ▼ battery {2}
        level : <positive integer>
        capacity : <positive integer>
    ▼ appliances [2]
      ▼ 0  {3}
          type : <Cooking, Washing or TV>
          state : <Waiting, inExecution or Ended>
          progress : <float between 0 and 1>
        ▶ 1  {3}
    ▼ generators [1]
      ▼ 0  {3}
          type : <Solar or Eolic>
          productionPerHour : <positive float>
          efficiency : <float between 0 and 1>
    ▼ bid  {1}
        plotFile : path/to/file
  ▶ 1  {12}
  ▶ 2  {12}
```








5 kw


10:00
10:20
11:00