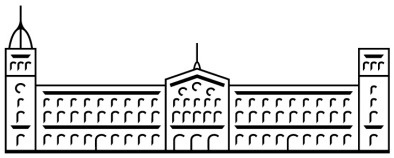

Trabajo final de grado

GRADO DE INFORMÁTICA

Facultat de Matemàtiques
Universitat de Barcelona

Visualización de Smart Grids

Autor: Martin Azpillaga Aldalur

Director:	Dr. Jesús Cerquides
Realitzat a:	Institut de Investigació de Intel.ligència Artificial
Barcelona,	15 de enero de 2016

Agradecimientos

Me gustaría agradecer a los siguientes personas por proporcionar material indispensable para la realización del proyecto: - iconos - usuarios de stackoverflow:

Abstract

Example abstract.

Resúmen

Resúmen

Índice

1. Introducción y antecedentes

1.1. El entorno: La humanidad necesita energía eléctrica

Hay gran interés por parte de goviernos en invertir en investigar maneras más eficientes de tratar con la energía.

1.2. La idea de los Smart Grids

Auge de energías renovables personales. Mencionar artículos donde se tratan smart grids.

1.3. El problema de la liquidación de mercados distribuidos de energía: CEAP

Quien controla todo el flujo, cuando se hacen los intercambios, como se hacen los intercambios, como pueden interferir los usuarios en estos intercambios. Mencionar reglamentos de otros países y estado actual de España.

1.4. Algoritmos de clearing para mercados distribuidos de energía: ILP, Radpro

1.5. RadPro.

1.6. A continuación

Definiremos en que manera queremos ampliar este trabajo.

2. Objetivos

2.1. En este capítulo

2.2. El problema

Explicar qué se intenta resolver

2.3. Partes del problema: Creación de grafo, CEAP, Simulación, Visualización, Reports

Explicar por que existe cada parte y por que está diferenciado del resto.

3. Metodología y herramientas

3.1. En este capítulo

Explicar programas adeuados para cada parte así como posibles distintas maneras de implementar cada aspecto a grandes rasgos

3.2. A continuación

Iremos explicando cada parte desde el núcleo (CEAP) hasta el exterior (Visualización)

4. Creación de la ciudad

4.1. En este capítulo

Empezaremos por crear una ciudad sobre la cual simular una red eléctrica inteligente. Para ello, primero deberemos definir que entendemos por ciudad dentro de nuestro programa, en que formato deberíamos guardar esta información, como acceder a esta información desde dentro del visualizador, como se interpreta y reproduce esta información dentro del visulizador, algoritmos que permiten crear una red que conecte todas las parcelas de nuestra ciudad y por último como usar herramientas como OpenStreetMap y CityEngine para crear ciudades realistas.

4.2. Definiendo la ciudad

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.

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

4.3. Leyendo el fichero

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.

4.4. Creando la red

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.

4.5. Información completa: Steiner Trees

En el primer caso encontramos el conocido problema de Minimum Spanning Tree Problem, que trata de construir un camino que permita a un viajero pasar por todos los puntos recorriendo la mínima distancia posible. Sin embargo, en 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.

Formato del txt: ¡imagen¿

4.6. Información incompleta: Nearest neighbour problem

en el caso anterior considerábamos que conocíamos la posición de todos los puntos que formarían el grafo. Sin embargo, si nos fijamos en la realidad, esta idea no sería práctica, ya que a lo largo del tiempo se van añadiendo y eliminando parcelas de manera que alteran los puntos del problema. Para crear un árbol en este caso usaremos el algoritmo Nearest Neighbour. Este algoritmo recorre cada uno de los puntos 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.

Implementación en pseudocódigo: ¡código¿

4.7. Ampliar a Open Street Map, CityEngine.

Por último se ha considerado como construir los modelos 3D de las ciudades. Y qué mejor manera que utilizando ejemplos existentes del mundo real. El mundo está repleto de ciudades y gracias a mapas online como OpenStreetMap es fácil conseguir un fichero que contenga toda la información relevante de una zona del mundo, como por ejemplo el corazón de Manhattan. En OpenStreetMap podemos exportar un trozo de mapa en 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 añadiendoles 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.

4.8. A continuación

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

5. Simulación

5.1. En este capítulo

5.2. Filosofía de java. POO, encapsulación, interfaces.

5.3. Input/Output.

5.4. Diagramas.

5.5. Modelado de una casa

5.6. Battery,

5.7. Generator,

5.8. Appliance,

5.9. Bid.

5.10. A continuación

6. Visualización

6.1. En este capítulo

Dada la naturaleza tridimensional del problema, se optó por usar un programa que permitiera visualizar los datos en 3D e interaccionar con ellos. Unity, como editor de juegos, permite esta interacción y tiene la gran ventaja de poder crear un ejecutable para las plataformas más reconocidas como Windows, Mac, Linux, pero también web o android. Dicho ejecutable será el producto final del proyecto, ya que incluirá el simulador, mientras que este a su vez incluye el 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 nustro programa fácilmente.

6.2. Filosofía de diseño de Unity

La programación en Unity puede efectuarse en los lenguajes C++, JavaScript y Boo, aunque este último no se utiliza habitualmente. Todos ellos son lenguajes dentro del paradigma Programación Orientada a Objetos, pero en Unity cambia la manera en el que se organizan los datos. El cambio más importante es que lo que se programa son comportamientos, no entes. Los objetos centrales de Unity son los prefabs: Packs de componentes que definen como se comporta ese objeto dentro de la escena. Por ejemplo, todos los objetos tienen una componente Transform que indica la posición, rotación y escala del objeto. Unity ofrece varias componentes comunes predefinidas listas para usar en tus prefabs tal como colliders para detectar colisiones entre objetos o mesh renderers para definir la geometría (los vértices) que forman el objeto. Por ello la implementación de funcionalidades se centra en encontrar acciones/comportamientos de los objetos de la escena y definir 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 a

6.3. Diseñando los paneles

Una vez ejecutada la simulación, contamos con un json que describe toda la información relevante que ha ocurrido en cada frame. La información sobre el momento del día lo mostraremos en la GUI principal de manera que sea apreciable desde cualquier punto de la escena. Aprovechando la naturaleza 3D de nuestro visualizador intentaremos mostrar 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:¡imagen¿

- 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.

6.4. Control de la cámara

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

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

que considerar la variable Time.deltaTime de unity en 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).

6.5. Eligiendo la ciudad

Añadir un modelo a una escena previamente a su ejecución es relativamente sencillo, basta con crear un prefab que tenga las componentes que te interesan. Sin embargo, importar un modelo en tiempo de ejecución en Unity es más complicado de lo que puede parecer. El primer problema es abrir un cuadro de diálogo que permita al usuario elegir un archivo. Unity usa Mono que es un subconjunto de .NET FrameWork 2.0 especializado para juegos, que por defecto no incorpora la opción de crear ventanas propias del SO. La solución ha sido añadir el dll Windows.Forms.dll de .NET FrameWork 2.0 en una carpeta llamada Plugins dentro del proyecto de unity de manera que el compilador de unity comprenda que queremos usar tal extensión y permita crear las ventanas. El soporte de .NET 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.

6.6. Seleccionando la franja horaria

El segundo paso es seleccionar de que hora a que hora se ejecutará la simulación. Para ello contamos con un torus fino con dos bolas sobre ella. Estas bolas se pueden seleccionar y arrastrar sobre el torus y muestran un lienzo al moverse que indica la hora que señalan. La idea bajo esta implementación es: - Detectar si el mouse está en modo drag con el método 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

6.7. Ejecutando la animación

Como tercer paso, el usuario hace click en la estrella de acuerdo con que está contento con los parámetros elegidos hasta ahora. Esto crea un nuevo thread que se encarga de correr la simulación. La creación del thread se consigue gracias a la clase ThreadedJob que es aportación de ¡referencia¿En particular este thread tiene un método que se ejecutará al ser creado. Además, cada vez que el thread principal

llame al 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.

6.8. Mostrando la información relevante

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

Por otro lado, mediante las teclas H y G podemos controlar qué partes de la ciudad se ven en cada momento. Con H (de house) podemos revelar/ocultar las casas mientras que con G (de grid) podemos revelar/ocultar los cables. Por último, con la tecla Esc podremos salir de la aplicación. El objeto manager es el que contendrá estos comportamientos.

6.9. Programando las animaciones

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

Unity cuenta con 2 modos por defecto que permiten hacer animaciones: -La componente Animation permite ejecutar animation clips guardados en una lista, es la manera original de hacer las animaciones. -La componente Animator también conocida como Mecanim, que genera una 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.

6.10. Controlando el flow

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

6.11. Parseando el JSON

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

6.12. A continuación

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

7. Conclusiones y futuro trabajo

7.1. Conclusiones

7.2. Futuro trabajo

Referencias

[1] 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.