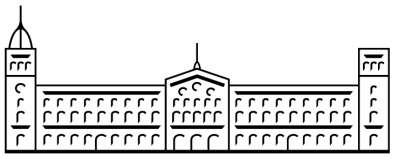

Trabajo final de grado

GRADO DE INFORMÁTICA

Facultat de Matemàtiques
Universitat de Barcelona

Visualización de redes inteligentes

Autor: Martin Azpillaga Aldalur

Director:	**Dr. Jesús Cerquides**
Realitzat a:	**IIIA**
	(nom del departament)

Barcelona, **13 de enero de 2016**

Abstract

Example abstract.

Resúmen

Resúmen

Índice

1. El entorno(1)

1.1. 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. Problemas: ¿Como se tradea la energía?

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.

2. Definiendo el problema(2)

2.1. El problema

Explicar qué se intenta resolver

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

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

2.3. Buscando tecnologías adecuadas

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

2.4. A partir de ahora

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

3. CEAP(5)

3.1. Se me presentó este proyecto.

3.2. Formalización en participant, ILV, PLV, Link.

3.3. RadPro.

3.4. Limitaciones y ampliaciones posibles.

4. Simulación(8)

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

4.2. Input/Output.

4.3. Diagramas.

4.4. Modelado de una casa

4.5. Battery,

4.6. Generator,

4.7. Appliance,

4.8. Bid.

5. Visualización(5)

5.1. Filosofía de Unity. Componentes/no.

5.2. Menú dinámico.

5.3. Animaciones.

6. Reports(2)

6.1. Aún por decidir.

7. Creación de la ciudad(5)

7.1. 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.

7.2. 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.

7.3. Eligiendo la ciudad

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.

7.4. 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.

7.5. 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.

7.6. 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¿