Memoria de Sistemas Informáticos

Visualización de Grafos planos

Autor	Guillermo Vayá Pérez
Matrícula	F980446
Curso	2012 - 2013

Índice de contenido

Introducción

El presente documento representa la memoria de trabajo sobre la asignatura Sistemas Informáticos desarrollada por Guillermo Vayá Pérez.

Descripción

El proyecto consiste en el desarrollo de un sistema de ayuda a la representación de varios algoritmos mediante los cuales se realiza una representación gráfica plana de un grafo.

Una representación plana de un grafo es aquella en la que el grafo no contiene aristas que se corten entre sí.

Herramientas utilizadas

Para desarrollar la aplicación se ha utilizado el lenguaje Scala junto con el *framework* libGdx.

Scala [3] es un lenguaje escrito sobre la JVM con una sintaxis más sencilla que la de Java y con una orientación más cercana a la Programación Funcional, facilitando y permitiendo la mezcla de programación orientada a objectos con la programación funcional.

LibGdx [4] es un *framework* orientado principalmente a la creación de videojuegos multiplataforma. Inicialmente se eligió por la posibilidad de portarlo a plataformas como iOS o Android, aunque finalmente no se ha llegado a desarrollar esa parte. A pesar de ello, permite utilizar el mismo código sobre las distintas plataformas principales (Linux, MacOS y Windows) sin necesidad de cambiarlo.

Github [5]. Se ha utilizado como repositorio del código y ha sido de tremenda utilidad ya que durante el desarrollo cambiaba frecuentemente de plataforma entre windows, mac os y linux para asegurar que el sistema funcionaba siempre en las 3 plataformas.

Por último, se utilizó **LibreOffice** [8] para generar la presente documentación.

Diseño

Algoritmos

Los algoritmos utilizados son los proporcionados por el tutor

Generación de un grafo DCEL

Esta es una estructura que facilita enormemente la triangulación del grafo. Para conseguirla, se duplican las aristas dándoles una orientación inversa a cada par. Partiendo de un vértice se va generando la cara hasta que se vuelve al inicial. En cada paso se elige siempre la siguiente arista más cercana en el sentido contrario a las agujas del reloj. Finalmente el último grupo de aristas que se forman son los de la cara exterior.

Triangulación

Para cada cara formada en el grafo DCEL, se comprueba si es triangular, si no lo es, se buscan 2 vértices separados por un tercero de manera que se generen 2 nuevas caras, siendo al menos una de ellas triangular. La arista trianguladora, se genera siguiendo la misma pauta que las originales, es decir, duplicada de manera que cada una pertenezca a una cara y esté orientada en sentido opuesto.

Ordenación canónica

Para la ordenación canónica se ha utilizado el algoritmo presentado por [CP90] y que se describiría de la siguiente manera:

Se toman 2 vértices v1 y v2, y se asigna la etiqueta -1 a todos los demás vértices. Inicialmente se procesa v1, por lo que se asigna la etiqueta 0 a todos los vértices adyacentes y después se procesa v2. A partir de v2 se siguen unas reglas cada vez que se procesa un vértice:

Sea v un vértice vecino del vK que está siendo procesado. Según el valor de la etiqueta de v se procederá de distintas maneras:

- La etiqueta de v vale -1: ahora la etiqueta de v pasa a valer 0.
- La etiqueta de v vale 0: v tiene un vecino procesado (llamémosle u). Se comprueba si vK es vecino de u. Si lo es, la etiqueta de v vale ahora 1, si no se marca con un 2.
- La etiqueta de v es mayor que 0: Se comprueban los 2 vértices adyacentes a vK para ver si son vecinos de v. Si ambos lo son, se reetiqueta restándole uno a la etiqueta actual, si uno si y el otro no, se mantiene y si ninguno lo es, se incrementa en 1 la etiqueta.

A cada paso, se elige uno de los vértices con etiqueta valor 1 y se procesa hasta que todos los vértices han sido procesados.

Colocación de vértices en la malla

Debido a los problemas con la ordenación canónica, este algoritmo es el que menos se ha estudiado y probado. El algoritmo sería:

Siguiendo el orden canónico anterior se colocan los 3 primeros vértices en las posiciones (0,0), (2,0) y (1,1)

Cada vértice siguiente se coloca por encima de los ya colocados con pendientes de -1 y 1 a cada lado, si las aristas cortaran habría que redimensionar el grafo desplazando parte de los vértices.

Software

Desde el punto de vista del software el diseño se ha orientado de manera parecida al desarrollo de un videojuego, por ello se han dividido los distintos algoritmos en fases, consiguiendo una separación visual y de comportamiento.

Mediante esta separación, se ha facilitado mucho el poder volver a un algoritmo anterior para volver a visualizarlo o incluso para poder corregir el grafo dibujado inicialmente y ver las diferencias con los pasos dados anteriormente.

No se ha seguido el paradigma de orientación a objetos de manera estricta, sino que se ha aprovechado la ventaja de usar Scala y poder entremezclarlo con la Programación Funcional, de la que se ha hecho extensivo uso en distintas partes del código.

A continuación se explica el diseño seguido en las distintas escenas:

General

Se ha intentado seguir un diseño lo más minimalista posible, dado que el objetivo principal es el seguimiento del algoritmo y un exceso de iconos podría distraer al usuario.

Adicionalmente, los botones con los que controlar la escena (volver, siguiente y salir) están incluidos en una caja que puede desplazarse a lo largo de la pantalla pudiendo así moverla si dificultara la visión del grafo en el punto actual del algoritmo.

Introducción del grafo

Para poder introducir un grafo se ha seguido el mismo principio que en el resto, eliminando botones que podrían generar confusión. Por ello se aprovecha de los dos botones del ratón para las posibles

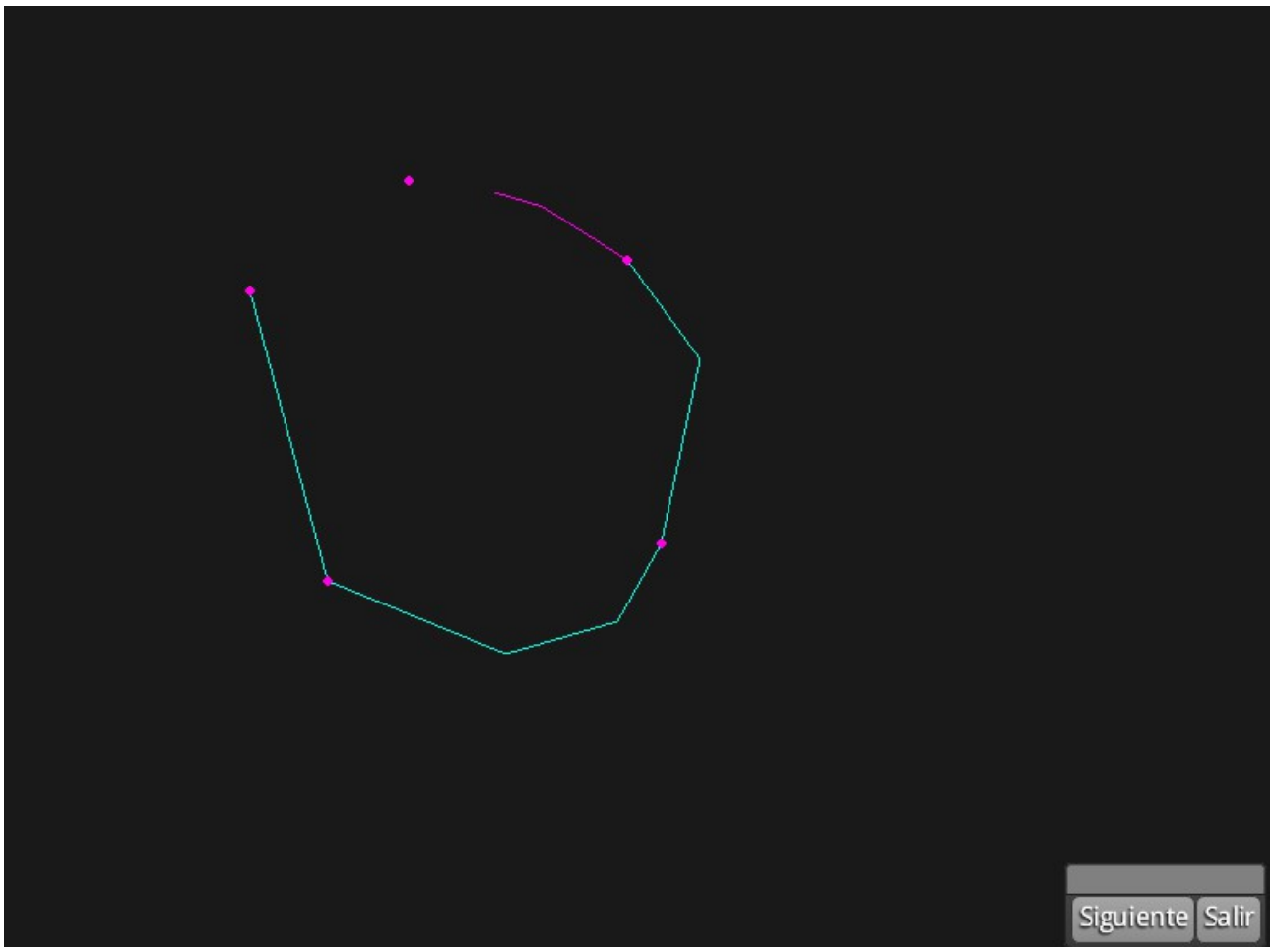
acciones. El botón izquierdo se utiliza para generar aristas mientras que el botón derecho se utiliza para generar los vértices. En el caso de iniciar una arista sin empezar en un nodo, el sistema creara automáticamente uno en el punto inicial donde se pulsó, aunque siempre intentará buscar un nodo cercano. Si encontrara un nodo cercano, el sistema asume que es ese nodo el que se quiere usar como inicio y modificará la arista para comenzar desde él. Si se pulsa de nuevo el botón izquierdo, se generará o bien un codo o si hubiera un vértice cercano, cerrará la arista considerando el nuevo vértice como el final. Otra opción es pulsar el botón derecho y el sistema ingresará un nuevo vertice finalizando así la arista. El resultado de esto es que se pueden incluso dibujar aristas curvas tan solo manteniendo el botón izquierdo pulsado, en el momento que nos acercáramos a un vértice, la arista terminaría y se incluiría en el grafo.

Mientras la arista no esté completa, el sistema la marcará con un color fuxia y una vez cerrada tomará un color azul claro.

Al trazar la arista, el sistema no permitirá cruzarlas, obligando de esta manera a la introducción de un grafo plano, pero permitiendo suficiente complejidad en las aristas al tener los codos. En caso de intentar cortar una arista con otra, el sistema desechará el nuevo punto y mantendrá como referencia el anterior.

Una vez el usuario está satisfecho con el grafo dibujado, podrá pulsar en el botón “Siguiente” y con ello finalizará la introducción del grafo, pasando a la siguiente fase.

En la siguiente imagen se pueden ver los diversos conceptos comentados:



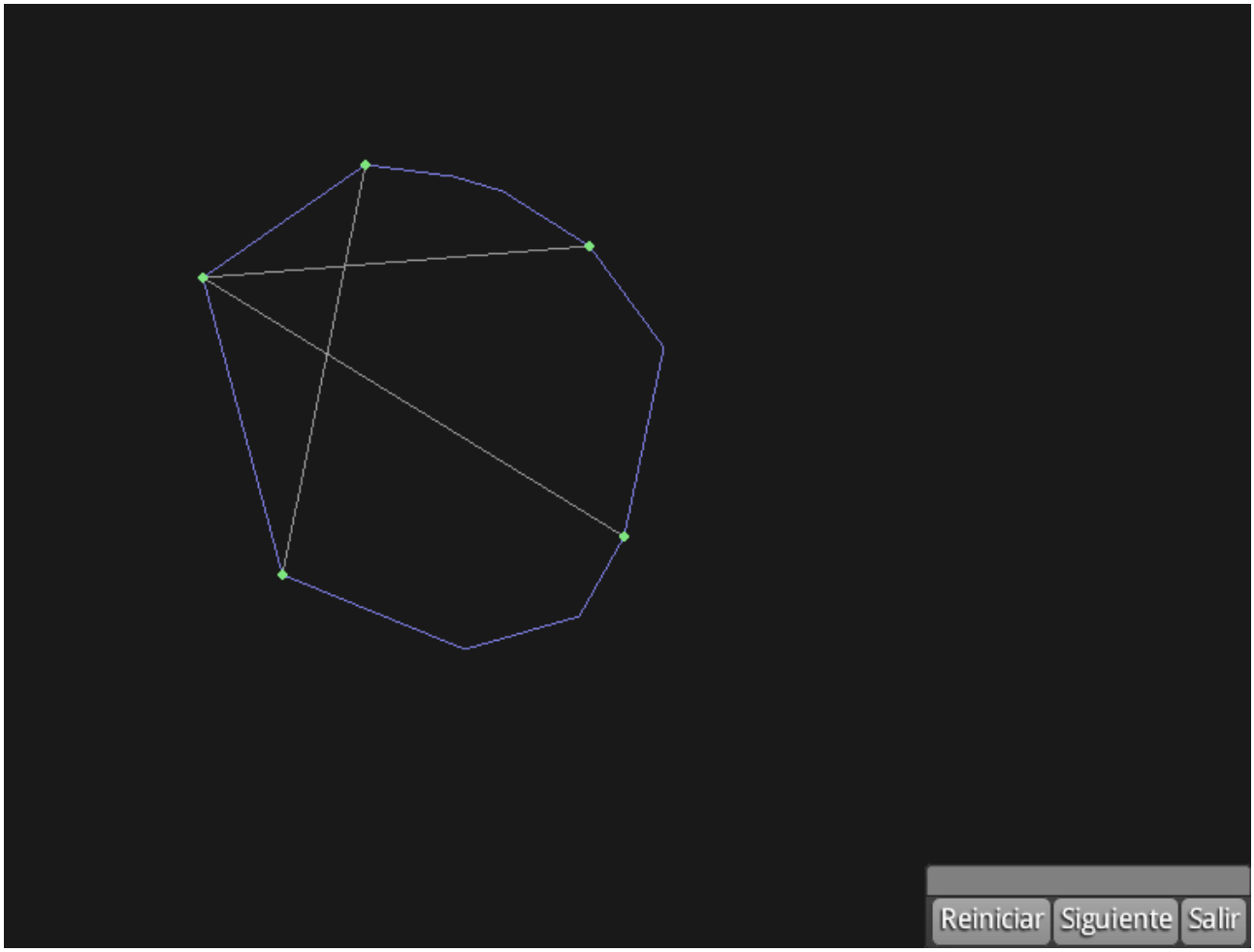
Generación de un grafo con estructura DCEL y triangulación

Aunque no se ha conseguido una representación de la estructura de DCEL (eso se deja como futura mejora del sistema) es el primer paso que sucede al cambiar de escena. A partir de aquí, cada vez que se pulse en siguiente se verá como el sistema calcula una nueva arista que cortará una de las caras haciendo el grafo triangular.

Para distinguir las aristas originales de las aristas de triangulación, se ha optado por utilizar colores diferentes durante esta fase, siendo azules las aristas originales y grises las utilizadas para la triangulación del grafo. En posteriores fases esta discriminación se elimina para no distraer al usuario con algo que ya no es el elemento principal del algoritmo de la escena en cuestión.

Una vez terminada la triangulación, se puede pulsar en el botón “Siguiente” para pasar a la siguiente escena.

En cualquier momento de esta escena se puede pulsar en el botón volver para ir a la escena anterior y poder modificar el grafo añadiendo nuevos vértices y aristas.

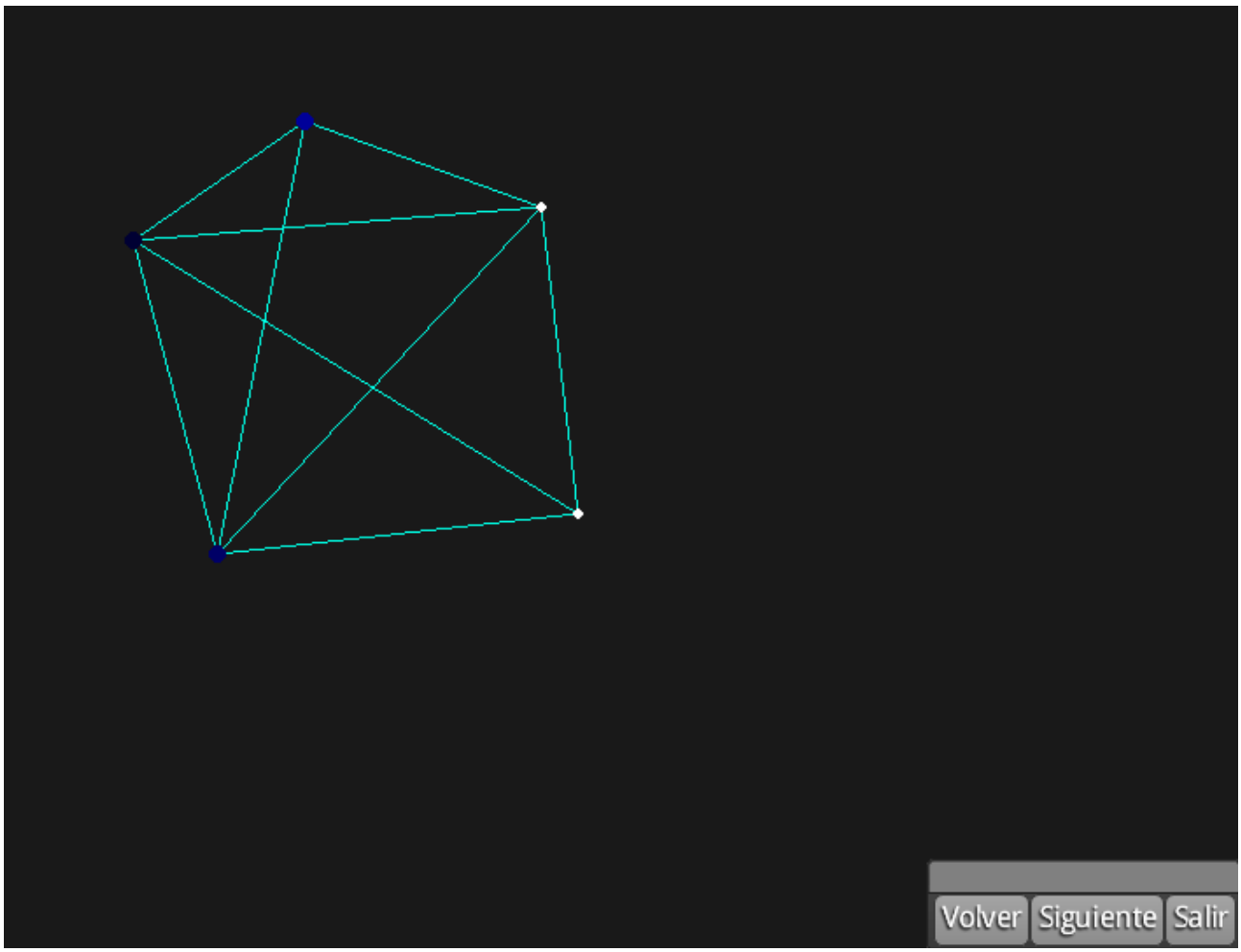


Ordenación canónica.

Inicialmente el grafo tendrá los vértices blancos y estos irán cambiando de color según se vaya ordenando el sistema. Para ello se ha utilizado un pequeño algoritmo que pasa el valor de orden a base 6 y cada dígito se multiplica por 51 (33 en hexadecimal) generando de esta manera una paleta de colores distinguibles con 216 colores validos.

Dado que el negro resultaría indistinguible y el blanco ya está usado por los vértices no ordenados aún se ha reducido la paleta a 214 colores, por lo que a partir del vértice número 215 se repetiría el patrón de colores. En cualquier caso, 214 se considera suficiente para los objetivos didácticos que se perseguían con el proyecto.

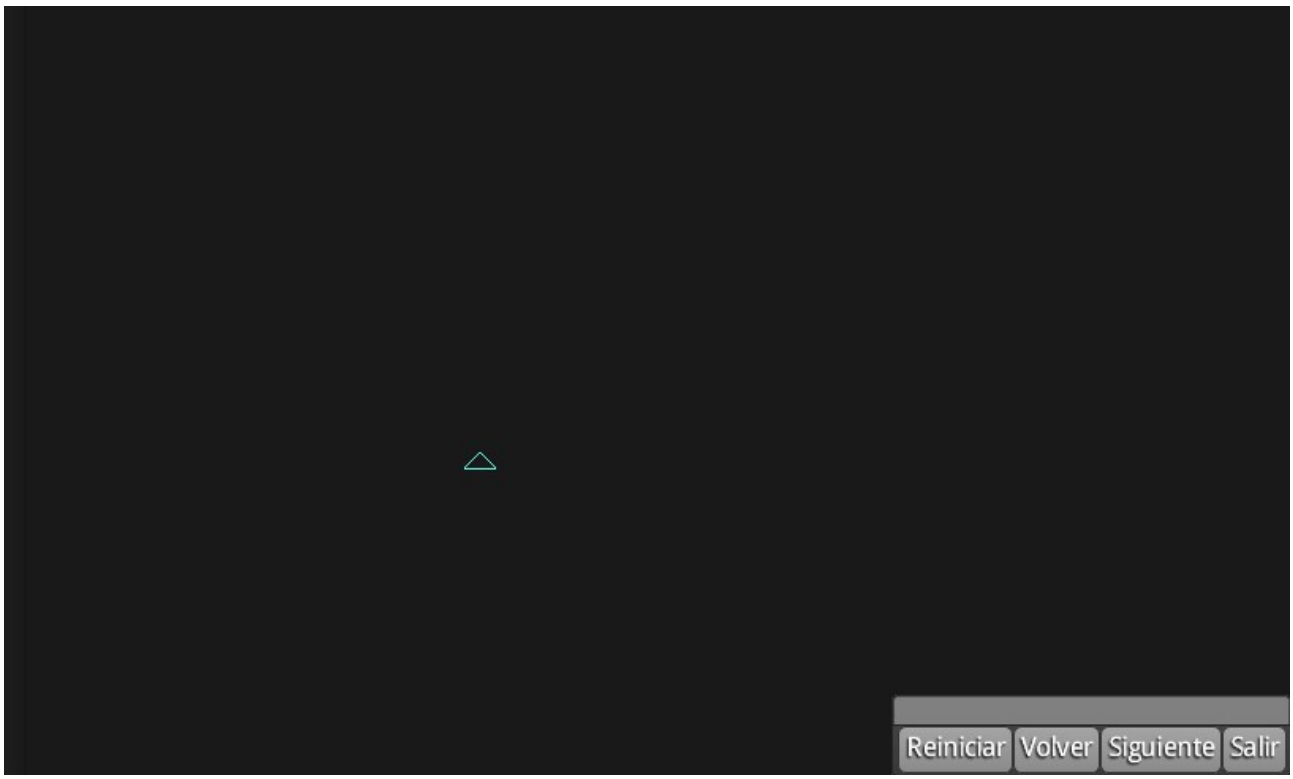
Una vez ordenados todos los vértices, se pasa a la escena final pulsando en el botón “Siguiente”



Dibujado del grafo

Durante esta fase se va incluyendo el siguiente vértice en función de la ordenación canónica previa. Respecto a las aristas, estas se dibujan únicamente si los dos vértices que la componen están siendo mostrados. Una vez se han dibujado todos los vértices, el último paso elimina las aristas triangulares dejando el resultado final del grafo.

En esta escena, se puede volver a la anterior pulsando en volver o se puede empezar de cero pulsando en reiniciar, lo que eliminará todos los datos del grafo actual y volverá a la pantalla inicial esperando un nuevo grafo.


Reiniciar
Volver
Siguiente
Salir

Obteniendo el proyecto

Jar compilado

Para obtener el .jar con todas las dependencias incluidas, se puede descargar de:

https://dl.dropboxusercontent.com/u/3776107/grafo_ssii_2013.jar

Con ese fichero y una instalación de JVM se puede ejecutar en cualquier plataforma con el siguiente comando:

java -jar grafo_ssii_2013.jar

Construyendo el proyecto

Para construir el proyecto, primero ha de instalarse SBT (el sistema de compilación de Scala), este está disponible tanto en las distribuciones más habituales como en la propia web del proyecto SBT.

Para obtener el código puede bajarse de Github [1] bien mediante el clonado del respositorio o pulsando en los enlaces de descarga de la propia web.

Una vez está todo lo anterior, seguir estos pasos:

1. Abrir una consola de comandos
2. Ir al directorio donde se ha puesto el código
3. Ejecutar sbt: sbt
4. Cambiar el proyecto de android a desktop mediante: project desktop
5. Pedirle a sbt que consiga las dependecias: update
6. Pedirle a sbt que descargue la ultima version de libgdx: libgdx-update
7. Ejecutar: run

Esto mostrara una ventana como las mostradas más arriba. Si en vez de eso se quiere conseguir un .jar para poder distrbuirlo, las instrucciones son iguales pero cambiando el paso 7 por ejecutar: assembly. Esto generará un archivo “.jar” con todo lo necesario para ejecutarlo sin tener siquiera Scala instalado, aunque si requerirá una JVM

Dificultades encontradas

Las principales dificultades encontradas son:

- **Algoritmo de ordenación canónica.** Todas las descripciones dadas asumen un lector con un conocimiento de grafos que aunque inicialmente poseía, ha ido disminuyendo con el paso del tiempo al entrar en el mundo laboral y no poder usarlo en el día a día. Por ello la lectura de documentación relacionada con este algoritmo se hizo ciertamente difícil.
- **Documentación**. A la hora de buscar documentación, sobretodo sobre el algoritmo de ordenación canónica, se descubre una falta de material importante, siendo apenas unos pocos *papers* y algún artículo los únicos que hablan del tema. Fue especialmente frustrante ver como además muchos de ellos eran copias del resto que incluso reaprovechaban el ejemplo presentado en el artículo original haciendo que su aporte fuera escaso (por no decir nulo) y acentuando el problema de la falta de documentación.
- **Herramientas**. Las herramientas también fueron un problema en dos ocasiones. Inicialmente se usó Python como lenguaje y Kivi como *framework*, buscando unas capacidades parecidas a las encontradas en Scala y libGdx, pero por desgracia Kivi en el momento de su uso era un proyecto muy reciente que no permitía hacer algunas cosas necesarias para el proyecto, por lo que hubo que abandonar el código usado y comenzarlo de nuevo con otras herramientas. Posteriormente se consideró utilizar JavaScript y HTML5 que quizás habrían sido mucho mejor elección, pero en el momento de reescribir el proyecto no disponía de los mismos conocimientos que tengo ahora sobre ambos por lo que el inicio hubiera sido complicado.
- **Versionado de herramientas.** A lo largo del desarrollo de este proyecto, tanto Scala como libGdx sufrieron un rediseño de algunas de las partes de que se estaban utilizando, por lo que hubo que actualizar las herramientas y adecuar el código para evitar que el proyecto fuera imposible de construir más allá de la versión inicial.

Mejoras

La primera mejora sería corregir el algoritmo de ordenación ya que es el problema principal que ha impedido que el sistema esté completo, ya que no siempre ordena correctamente y genera grafos que se cortan.

Una vez el sistema funcione, las mejoras considerables serían:

Etiquetado de grafos. Si bien estaba en mis planes iniciales etiquetar tanto los vértices como las aristas (y en el código todos los vértices y aristas tienen una etiqueta única) no se ha mostrado para evitar que el texto se cruce con otras aristas o que incluso tapara a vértices próximos entre sí, dificultando su visualización. Un estudio de las posibles técnicas a utilizar sería necesario para poder implementarlo.

Generalizar el sistema para poder utilizarlo con diferentes algoritmos. Aunque buena parte del código está separado (Interfaz – grafos) habría que refactorizar las escenas para que fueran más genéricas y que con pocas lineas de código más se pudiera utilizar casi cualquier algoritmo de representación de grafos.

Contemplar el algoritmo con grafos con aristas que se corten entre sí. Realizar un estudio de planaridad del grafo previo a los algoritmos aquí presentados y si es aplanable, continuar el algoritmo.

Representación de un grafo en su estructura DCEL. Aunque tenía la idea de representarlo, los primeros intentos que hice al respecto quedaban muy difíciles de leer por lo que se descartó y se ocultó esta representación al usuario en aras de la claridad. Era especialmente confuso si alguna cara era pequeña o si había aristas con pocos grados de diferencia.

Conclusión

El proyecto, aunque interesante y altamente educativo, ha sufrido mucho debido al algoritmo de ordenación canónica y a mi falta de tiempo debido a los horarios de mi trabajo. Aun así no quería dejar de presentarlo por si en un futuro se decide continuar o utilizar partes de él para mostrar otros algoritmos igualmente interesantes.

Bibliografía y enlaces

[1] Código fuente: https://github.com/Willyfrog/grafo

[2] Jar compilado: https://dl.dropboxusercontent.com/u/3776107/grafo_ssii_2013.jar

[3] Scala: http://www.scala-lang.org/

[4] libGdx: http://libgdx.badlogicgames.com/

[5] Github: http://www.github.com

[6] Python: http://www.python.org

[7] Kivi: http://kivy.org/

[8] LibreOffice: http://libreoffice.org

[9] Paleta de colores distinguibles: http://www.hitmill.com/html/color_safe.html

[CP90] Chrobak, M. and T.H. Payne, "A Linear Time Algorithm for Drawing a Planar Graph on a Grid"