

BACK FROM THE KLONDIKE

2º Trabajo de Programación de Sistemas

[Memoria del proyecto del grupo V07](#)

Generación de mapas, resolución de algoritmo, visualización gráfica y modo interactivo

Óscar García Lorenz (15166)

Alejandro Redondo Ayala (15361)

Mario Musicò Cortés (15308)

Índice

1.	Introducción	2
a.	Ideas generales.....	2
b.	Reparto de tareas.....	2
1.	El programa	3
a.	Diagrama UML de clases	3
b.	Generación de mapas aleatorios	4
c.	El solver: Klondike	5
d.	El viewport y visualización con openGL	6
e.	El menú y los modos de interacción	7
2.	Anexos	8
a.	Indicaciones para la compilación	8
b.	Control de versiones con Git	8

1. Introducción

Este apartado resume los objetivos propuestos y alcanzados en el desarrollo del trabajo.

a. Ideas generales

Para diseñar el algoritmo de búsqueda de la salida más corta posible del mapa de Klondike, utilizamos un procedimiento que consta de dos fases: primero el cálculo de “distancias” (nº mínimo de saltos que se han tenido que dar hasta llegar a la casilla en cuestión) a través de una búsqueda en amplitud, y una vez que se alcanza una casilla periférica (el objetivo del juego), se reinicia la búsqueda sólo en las casillas que han sido visitadas hasta el momento y que por tanto tienen asignada una distancia, ésta vez en profundidad para encontrar el recorrido que en la etapa anterior ha conseguido alcanzar la meta (sólo hay uno, ya que el cálculo de distancias se ha interrumpido al alcanzar la meta en el mínimo número de saltos). Este procedimiento se detallará a continuación con mayor profundidad.

También hemos tomado la iniciativa de diseñar un generador de mapas aleatorios, para poder jugar a resolver el algoritmo y demostrar que funciona independientemente de los valores que haya en cada casilla.

Cabe destacar que resulta inviable calcular todas las salidas posibles que tenga un mapa, ya que el número de caminos posibles crece exponencialmente con la profundidad (“distancia”).

Para la visualización del procedimiento a través de OpenGL usando la librería glut, se dibujan diferentes elementos:

- Un pico (voxelizado al estilo de Minecraft) que indica la casilla en que el minero se encuentra en el momento, durante la primera etapa de cálculo de distancias.
- Cada casilla se solapa con un cuadrado de diferentes colores, semitransparentes: rojo si aún no ha sido visitada, verde si ha sido visitada incluida la actual, y azul si son las casillas a las que se puede saltar en el próximo paso (situadas en los puntos cardinales de la actual).
- Al final, y una vez se ha encontrado el camino más rápido de salida, se visualizan arcos de circunferencia entre casillas consecutivas, simulando los saltos que tendrá que dar el minero para salir del laberinto.

Diferentes botones en un menú para cambiar el modo en que se visualiza la solución, dando la opción al usuario para que realice la búsqueda él mismo de manera interactiva.

También se han integrado diferentes interacciones con la ventana gráfica de OpenGL a través de movimientos del ratón (zoom, rotación 3D y traslación).

b. Reparto de tareas

Las tareas se han repartido entre los diferentes miembros del equipo:

Óscar: Diseño de algoritmo de resolución, vistas de cámara, forma base del pico, visualización de arcos de saltos, generación de nuevos mapas aleatorios. Menú con botones mejorados.

Alejandro: implementación en c++ de la clase Figuras y visualización gráfica de figuras con OpenGL. Interacción entre algoritmo e interfaz gráfica. Instrucciones de uso (texto y recuadro).

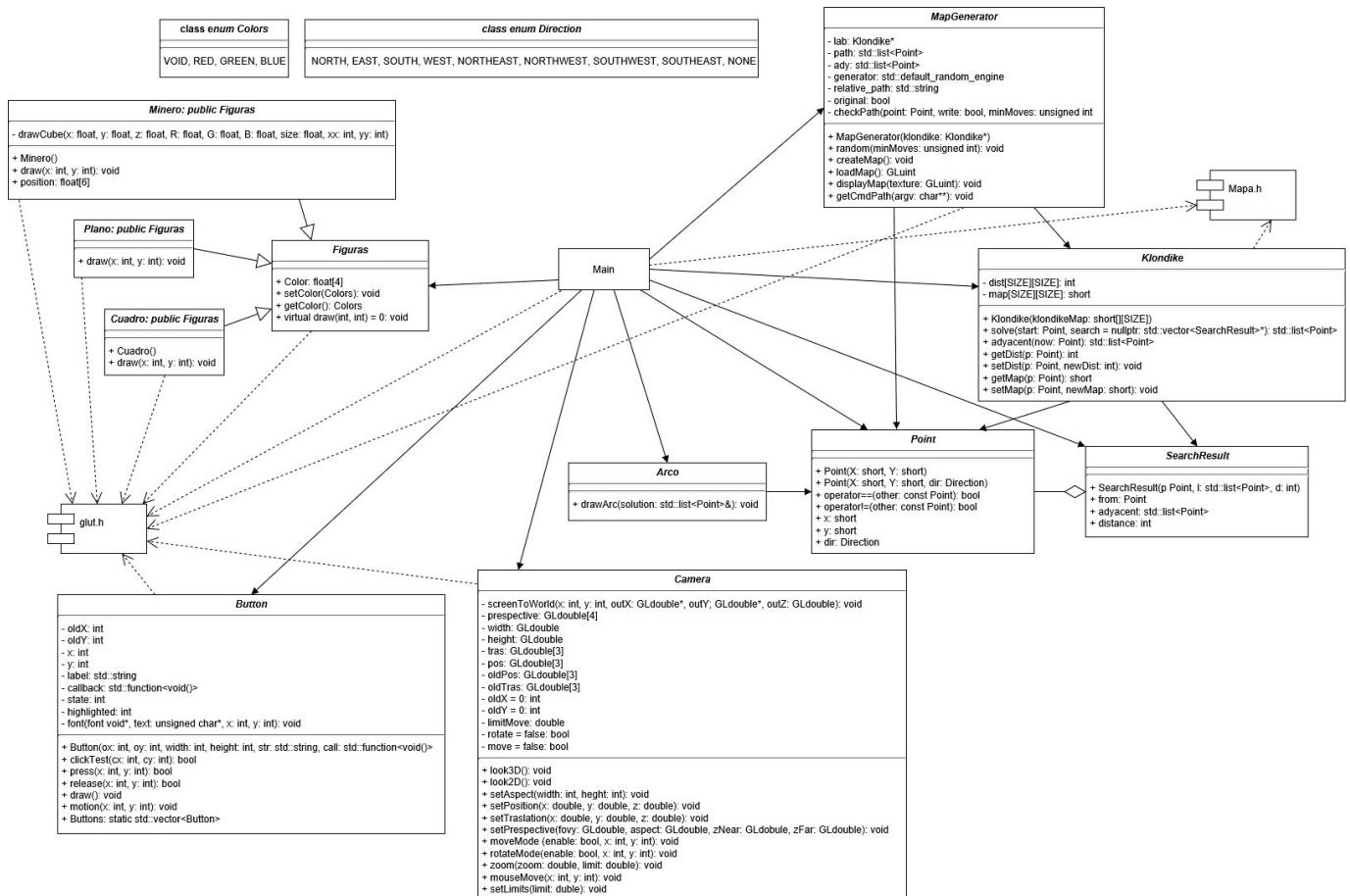
Mario: diagrama UML, documentación, prototipo de botones básicos y menú por consola (desestimado ya que Óscar ha propuesto una versión más completa). Considérese nota inferior.

1. El programa

En este apartado se explica cómo funciona el programa, añadiendo comentarios y aclaraciones para facilitar su comprensión.

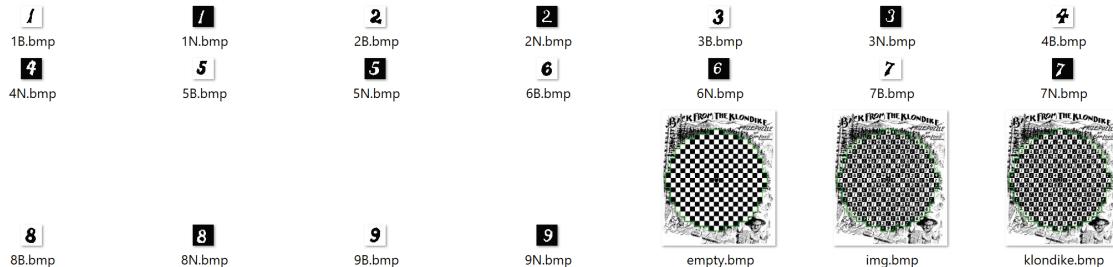
a. Diagrama UML de clases

A continuación, se adjunta el diagrama de clases que se han utilizado en el programa. Se han utilizado los recursos aprendidos durante las clases de la asignatura, mediante una programación orientada a objetos, utilizando diferentes clases que han permitido la encapsulación de sus métodos, agilizando el desarrollo del programa entre los tres. También hemos hecho uso de las herramientas que proporciona C++, como son la herencia (varias subclases de Figuras), el uso de clases abstractas (Figuras), el polimorfismo (funciones con mismo nombre pero distinto prototipo), las referencias, etc. Las diferentes relaciones entre las clases se muestran en el diagrama UML que se incluye a continuación. El diagrama ha sido creado con la webapp draw.io, y el archivo original se puede descargar [accediendo a este enlace](#).



b. Generación de mapas aleatorios

Para poder visualizar correctamente y de manera integrada diferentes mapas aleatorios, hubo que diseccionar el mapa original proporcionado como Klondike.bmp en las diferentes casillas, con todas las posibles combinaciones de números y colores, y generando un mapa base vacío, sin números.



Por defecto, al arrancar el programa, se carga el mapa original proporcionado. Cuando se ejecuta el comando de generar un nuevo mapa, el método de la clase MapGenerator `random()` genera una nueva matriz de saltos (valores que aparecen en las casillas del mapa), y en función de estos nuevos valores, `createMap()` genera un nuevo `img.bmp` utilizando las imágenes de los números correspondientes. A continuación, `loadMap()` carga el nuevo mapa en la textura del plano base de OpenGL, que visualiza en el viewport el nuevo mapa. El algoritmo de resolución utilizará entonces la nueva matriz de saltos generada para calcular los posibles movimientos.

Para que el mapa tenga solución única, `random()` tiene en cuenta el siguiente proceso:

1. Se vacía la matriz de saltos y se posiciona en la casilla de partida (11,11 si se desea que sea el centro), añadiéndola a la lista de la solución.
2. Se elige una longitud de salto aleatoria para la casilla actual, así como una dirección de salto aleatoria (N, NE, E, SE, S, SO, O, NO), y se efectúa el salto salvo que se cumpla alguna de las siguientes condiciones, en cuyo caso se recalculan ambos valores aleatorios:
 - a. Si se puede alcanzar desde la casilla actual la meta habiendo efectuado un número de saltos inferior al mínimo indicado.
 - b. Si se llega a una casilla que ya está en el camino de salida.
 - c. Si se llega a una casilla inmediatamente cercana (a las que se puede saltar desde ella) a una casilla que ya está en el camino de salida.
3. Se rellenan el resto de casillas que no pertenecen al camino de salida, asignándole un valor aleatorio que no cumpla que:
 - a. Esa casilla permite salir del mapa
 - b. Desde esa casilla se puede ir a una casilla del camino

c. El solver: Klondike

El solver trabaja en todo momento con la matriz de saltos que se pasa como argumento al constructor de la clase Klondike. En caso de generarse un nuevo mapa, se le volvería a pasar la nueva matriz de saltos sobre la que deberá trabajar.

Esta matriz de saltos tiene números enteros que indican:

- a. El valor es positivo: indica la longitud del salto que puede dar el minero cuando se encuentra en el punto x,y del mapa correspondiente con los índices del elemento de la matriz en cuestión
- b. El valor es 0: es una casilla periférica (meta)
- c. El valor es -1: es una casilla extra-periférica, por lo que no puede ser visitada (inválida).

La matriz de distancias en cambio parte con todas las casillas con valores negativos, y a medida que se visitan los puntos correspondientes, los elementos de esta matriz se irán sustituyendo por la distancia al punto de partida (mínimo número de saltos necesarios para llegar hasta ella, o “profundidad” del árbol de posibilidades).

Cada casilla del mapa está idealizada por un objeto de la clase Point, definido por su posición x,y y que por tanto tiene un valor asociado a él en la matriz de saltos.

Como se ha explicado en la introducción, nuestro modo de resolver el juego consta de dos fases principales:

1. Cálculo de distancias (BFS o búsqueda en amplitud):

- a. Se añade la casilla (“Point”) de partida a una lista FIFO de Points por visitar.
Distancia = nº de saltos = 0.
- b. En esta lista, se miran las casillas visitables (aún no visitadas y con valor de salto 0 o positivo) desde la casilla correspondiente al primer punto de la lista (teniendo en cuenta la longitud de salto indicada en el elemento con misma posición de la matriz de saltos), y se añaden estos puntos “visitables” al final de la lista. En la matriz de distancias, se indica la distancia de las casillas visitables, igual a la distancia de la actual incrementada en una unidad.
 - i. Si este valor es positivo, pasa al punto c.
 - ii. Si el valor es 0, ha alcanzado una casilla periférica (meta) y por tanto termina esta fase del algoritmo. Se vacía la lista y se añade la casilla a la lista de la solución.
- c. Vuelve al punto b hasta que se de el caso b.ii. La lista seguirá teniendo al menos una casilla (Point) visitable.

Después de esta primera fase, quedarán algunas casillas visitadas (valor positivo en la matriz de distancias), que quedarán coloreadas en verde, y otras que, antes de haber alcanzado una casilla meta, no hayan sido visitadas (por tanto, tendrán una mayor distancia que la de solución óptima, o posiblemente no haya combinación de saltos que llegue hasta ellas).

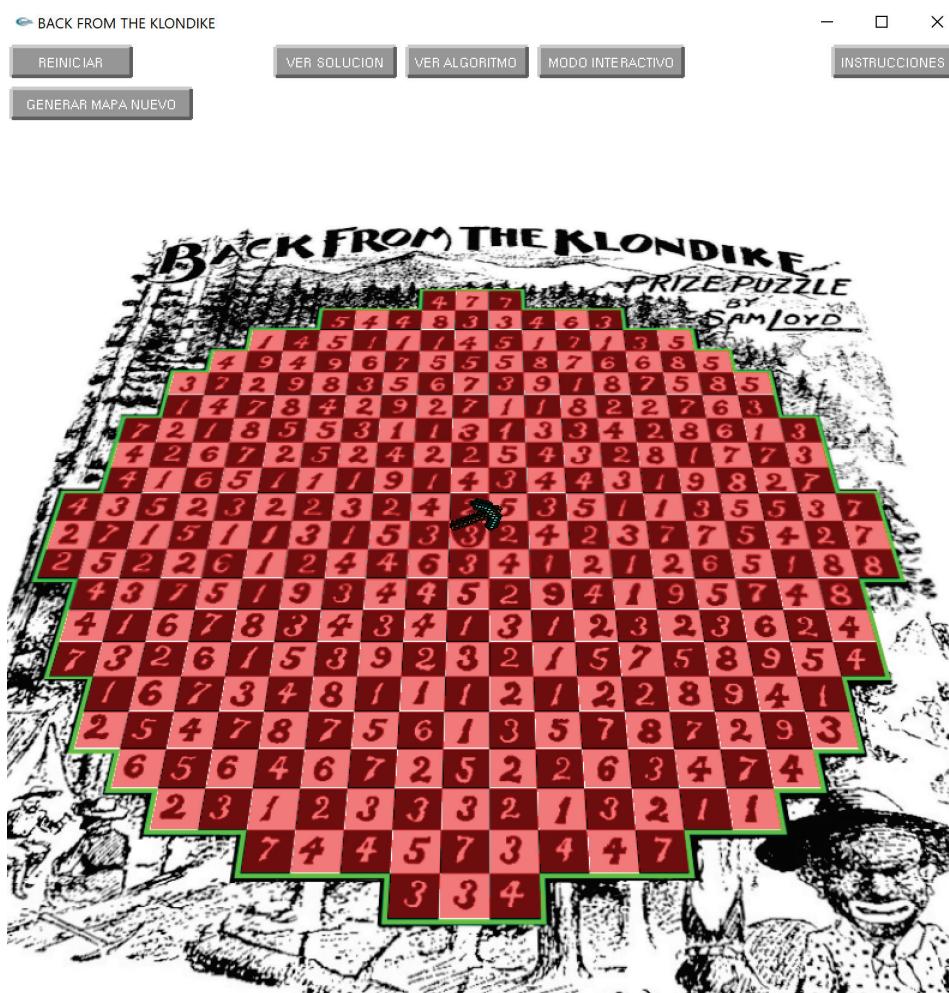
Por tanto, del árbol de posibilidades que contiene las casillas visitadas en la etapa del algoritmo anterior, sólo habrá una rama que haya alcanzado el “goal” o una casilla periférica (meta), ya que el resto no habrán sido visitadas antes de interrumpir la búsqueda. Como se ha indicado anteriormente, resulta ineficiente y superfluo calcular todos los caminos posibles, ya que pueden ser muchos, y se tardaría tiempo exponencial en visitar todos y también ocuparía memoria en exceso, cuando el objetivo planteado para el trabajo es encontrar el recorrido que requiera el menor número de saltos.

2. Búsqueda del camino alcanzado:

- Se lee la distancia del último elemento Point de la lista de la solución. En primera instancia, el único elemento corresponderá al punto “goal” encontrado en la fase anterior.
 - Se añaden todas aquellas casillas que tenga una distancia 1 unidad inferior a la del elemento en cuestión, a la lista de casillas visitables.
 - Se lee el primer elemento disponible de la lista de casillas visitables. Siempre habrá al menos uno ya que se ha tenido que saltar desde alguna casilla de distancia n a la casilla de distancia n-1 que estamos leyendo en este momento de la lista de la solución.
 - Si desde este punto, y a través de su correspondiente valor de salto en la matriz de saltos, es posible alcanzar el último punto de la lista solución, se añade este nuevo punto al final de esta lista, y se vacía la lista de casillas visitables. Si este nuevo punto tiene distancia 0, significa que se ha regresado a la casilla de partida. Fin de la búsqueda.
 - Si desde este punto no hay salto posible que llegue al último punto de la lista solución, se borra de la lista de visitables y se vuelve al punto c.
3. **Recorrido de la solución:** para visualizar la solución encontrada, basta con recorrer la lista de solución en sentido inverso, ya que el último elemento añadido será el de distancia 0, y el primero que se añadió en la primera fase de búsqueda es la meta.

d. El viewport y visualización con openGL

Se ha integrado en el viewport (ventana gráfica de OpenGL) una serie de elementos que permiten interactuar con el programa. Aquí se puede ver una captura de la interfaz que se muestra al iniciar el ejecutable.



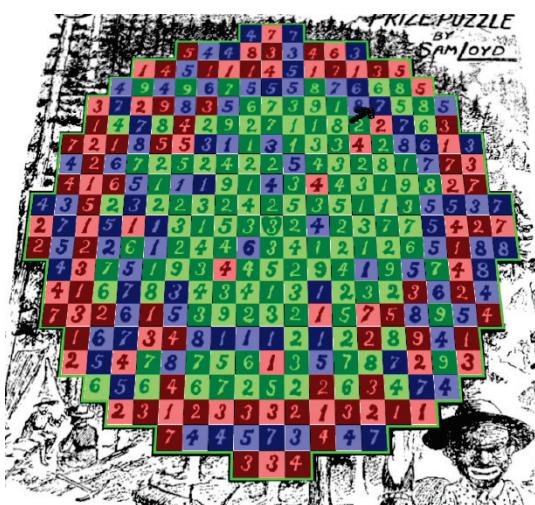
En la imagen anterior se pueden observar diferentes elementos:

- **Botones** (se detallará su funcionalidad en el siguiente apartado).
- **Plano base del mapa**, sobre el que se dibuja la textura del mapa vacío, superpuesto con las imágenes de los números correspondientes.
- **Estado de las casillas**: en este caso, al ser el comienzo del juego, están todas en rojo ya que aún no han sido visitadas.
- **Pico del minero**: indica dónde se encuentra el jugador en cada momento. Hemos decidido usar el pico de Minecraft para que tuviera cierta coherencia con el contexto del juego. Se construye voxel a voxel, y luego se dibuja en la posición que corresponda. Además, se ha animado en modo oscilante.

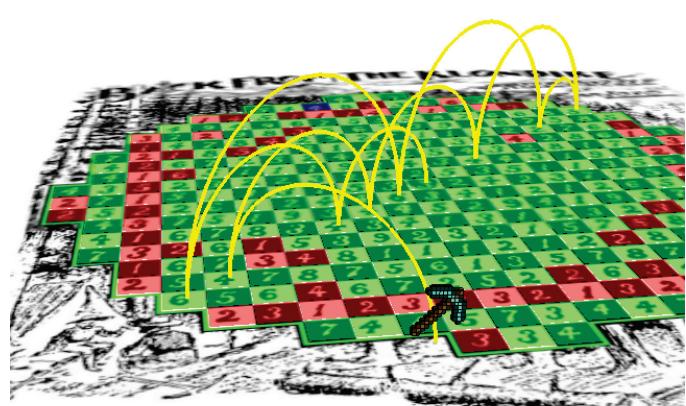
e. El menú y los modos de interacción

El menú consta de los siguientes botones, de la clase estática Button.

- **Reiniciar**: mantiene el mapa cargado, pero devuelve el minero a la posición inicial, y borra la visualización de cualquier trayecto (visitas a casillas y arcos de solución), y por tanto todas las casillas vuelven a colorearse de rojo.
- **Generar mapa nuevo**: reinicia y, además, genera un nuevo mapa aleatorio según el procedimiento descrito, y lo carga en la textura del plano base, apareciendo los nuevos números correspondiente a la longitud de salto en cada casilla. Está configurado que el mínimo número de saltos para la salida más corta sea 10.
- **Ver solución**: calcula la lista de la solución mediante el algoritmo de búsqueda, y la recorre visualizando los arcos de circunferencia que emulan los saltos entre casillas.
- **Ver algoritmo**: lo mismo que el botón anterior pero, además, a medida que va construyendo la matriz de distancias para generar la lista de la solución, visualiza las casillas que se visitan en la primera etapa de búsqueda del camino de salida, por lo que se van coloreando las casillas visitadas de verde translúcido (para que se pueda seguir viendo el número correspondiente a la casilla), y las azules son las que se pueden visitar saltando desde la que se encuentra en cada momento.
- **Modo interactivo**: cuando se encuentra activo, inhabilita el modo de búsqueda automático, y visualiza una roseta de botones para que el usuario indique el siguiente salto que desea que el minero haga, sustituyendo la aleatoriedad del algoritmo. A medida que el usuario va indicando los saltos que desea ejecutar, las casillas se van coloreando siguiendo el mismo criterio.
- **Instrucciones**: muestra un cuadro informativo con estas instrucciones resumidas.



Fase de cálculo de distancias



Visualización de la solución calculada por el algoritmo para el mapa original

2. Anexos

a. Indicaciones para la compilación

Para facilitar la portabilidad entre sistemas operativos y diferentes entornos de programación (IDEs) y compiladores, se ha comprobado la compatibilidad del programa tanto en Linux (usando *make*), como Mac (usando Xcode), como Windows (usando Visual Studio Enterprise).

Se ha estructurado el código en diferentes carpetas:

- **include** para archivos de cabecera (headers)
- **src** para código fuente (cpp). El archivo que contiene al main se llama Explorador.cpp
- **GL** incluye la librería glut.h
- **build** es el directorio donde el compilador guardará los archivos objeto, si se usa el Makefile
- **bin** es el directorio donde make guardará el ejecutable
- **graphics** contiene todas las imágenes *.bmp necesarias para cargar las texturas del mapa en openGL. klondike.bmp es la imagen original, empty.bmp es el canvas sin números, img.bmp es el mapa que usa el programa en el momento (generado por el propio programa, si le pedimos que genere uno nuevo). Por defecto, se encuentra dentro del directorio **bin** para que se pueda ejecutar directamente el programa tras el comando make.

Para usar el Makefile, hay dos comandos disponibles:

- a) **make**: compila el programa principal, el main debe encontrarse en un archivo fuente de la carpeta src. Archivos objeto se generan en carpeta build y el ejecutable en la carpeta bin.
- b) **make clean**: borra el directorio build con todos los archivos objeto generados, así como el ejecutable.

Para usar Visual Studio en Windows es necesario construir un proyecto a partir del código fuente proporcionado. No hemos visto conveniente añadir todos sus archivos y bases de datos que genera ya que ocupan mucho espacio. Indicaciones:

- Para que se ejecute correctamente el programa, la carpeta **graphics** deberá situarse en el directorio donde se genere el ejecutable con V.S.
- Se deberán incorporar al proyecto todos los archivos de **src**, **include** y **GL**, así como incluir en los directorios de VC++ del proyecto estos directorios.
- También será necesario incluir en el directorio del proyecto los archivos **glut32.lib** y **glut32.dll**.
- *Nota: la opción de zoom con la rueda del ratón no es compatible con Windows.*

b. Control de versiones con Git

Para poder trabajar cada miembro en su ordenador y luego poder fusionar fácilmente su parte del código, utilizamos el control de versiones Git, creando un repositorio privado en www.bitbucket.org que nos permitiera compartir los archivos y mantener un control de versiones para retroceder en caso de que no funcionara algo. Para clonar este repositorio en nuestros ordenadores usábamos GitKraken como interfaz gráfica, que permite ver los cambios del código entre versiones, y cambiar de rama fácilmente.