PRÁCTICA 1: GESTIÓN DE DATOS EN MEDIOS DIGITALES

Grado en Diseño y Desarrollo de Videojuegos, 3º A

Realizado por:

Kevin Carrasco Bertrán

Laura Hidalgo Rueda

ÍNDICE PRINCIPAL

1.1 Introducción **2**

1.2 Configuración inicial de la aplicación **3**

1.3 Generación del puzzle a partir de una imagen **4**

1.4 Manejo de eventos en Controlador **5**

1.5 Funcionamiento general con MVC **6**

1.6 Guardar y cargar partida **8**

Enlace a nuestro repositorio de desarrollo **8**

1.1 Introducción

La aplicación desarrollada durante la práctica se corresponde con un puzzle o rompecabezas (Imagen 1), en el cual el usuario es capaz de mover las piezas mediante *clicks* del ratón hasta su resolución.
Como opciones de interacción, el jugador dispone de botones para desordenar el rompecabezas tantas veces como desee, deshacer movimientos hasta la posición inicial del puzzle, rehacer movimientos hasta el último realizado, y un botón para resolver el rompecabezas de manera automática..
Además de estos botones, se dispone de 3 menús:

- El primer menú ***Archivo***, contiene la opción de ***"Load"***, que te permite personalizar tu rompecabezas con la imagen que elijas, y la opción de ***"Exit"***, que cierra la aplicación.
- El segundo menú es ***Help***, que contiene la opción ***"Info"***, la cual te muestra la información acerca de la aplicación.
- El tercer y último menú ***Partida*** contiene la opción de ***"Guardar"***, para continuar la partida en cualquier otro momento, y la opción de ***"Cargar"***, que te recupera la última partida jugada para que puedas continuar resolviendo el puzzle.

A la hora de cargar una imagen, el usuario puede introducir el número de piezas en el que quiere dividir el puzzle y, de forma adicional, se le permite seleccionar el tamaño de la imagen, por lo que se ha cambiado el tamaño del panel de la aplicación a unas dimensiones de 1500x800.
Para explicar el funcionamiento de la aplicación, lo dividiremos en apartados atendiendo a los requisitos de la práctica, los cuales se explican a continuación.

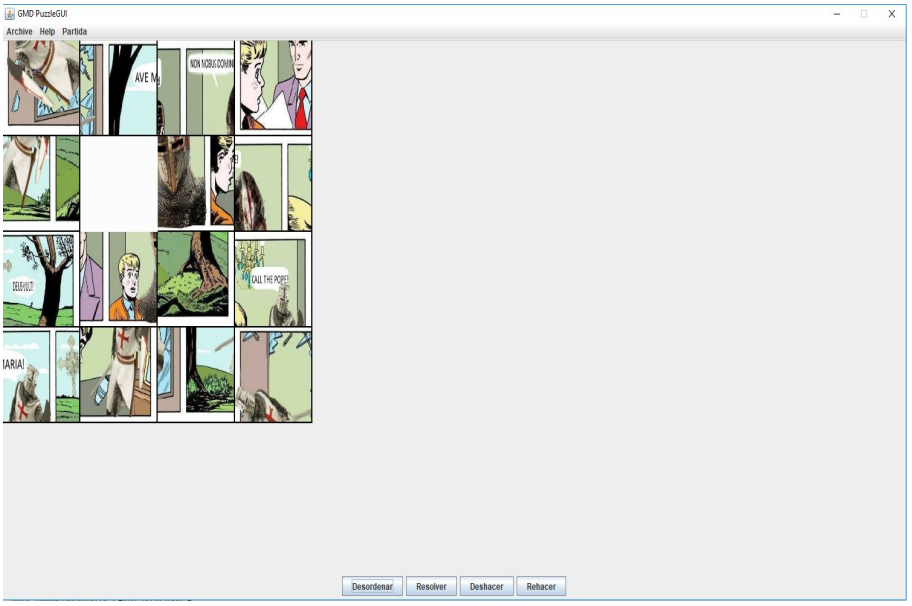

Imagen 1: Interfaz de la aplicación

1.2 Configuración inicial de la aplicación

Para abordar el problema de la configuración inicial de la aplicación, empleamos un fichero en formato .xml que incluye un .dtd interno, al tener una estructura más definida y regida por este último.

Creamos una clase llamada ***Configuracion*** dentro del package ***config***, desde la que realizamos la lectura y el parse del fichero inicial a una nueva clase denominada ***Info***, que almacena la ruta de una imagen predeterminada, un número de filas (al tratarse de un puzzle cuadrado, el número de filas y columnas es el mismo), y el tamaño predeterminado de la imagen completa (sin dividir). De esta forma, al inicio de la aplicación, tratamos de leer este fichero:

- Si la lectura es posible, creamos la ***BoardView*** y el ***Modelo*** en base a los elementos almacenados en el archivo, llamando a un nuevo método ***initialize()*** en ***PuzzleGUI***, que recibe como parámetro de "path" la dirección de la imagen sin dividir.
- Si la lectura se realiza con errores o el fichero no está disponible, la configuración mostrada sería similar a la proporcionada dentro del código base de la práctica.

Cabe la posibilidad de que exista el fichero de configuración ***puzzle.xml***, pero el elemento defecto (ruta de la imagen predeterminada) referencie a una imagen no existente o no disponible. En este caso, se llama al método ***initialize()*** base de ***PuzzleGUI***, y como parámetro de "path" se le pasaría el array de paths aportado junto al código base.

1.3 Generación del puzzle a partir de una imagen

Para generar nuestro puzzle a partir de una imagen seleccionada, se siguen los siguientes pasos:

1. Se pulsa el botón “load” que genera un evento en ***Controller***.
2. Se muestra al usuario el panel de selección de archivos (imagen 2), permitiéndole elegir imágenes con formato de .jpg, .png, .gif y .bmp.
3. Si el usuario ha seleccionado una imagen, se procede a borrar todo lo relativo al tablero antiguo, y se crea una nueva ***BoardView*** a partir de su segundo constructor, que recibe como parámetro un fichero, y no un array de paths.
4. Desde ***BoardView*** se redimensiona la imagen completa al tamaño proporcionado por el usuario, haciendo una imagen cuadrada y, a continuación, se divide en tantas piezas como haya indicado el usuario. Como necesitamos una pieza blanca que haga de tope, redimensionamos la pieza básica proporcionada en el código base mediante el método ***resizeBlanca()***, y se almacena en el array de imágenes con id = 0 (posición inicial de la pieza blanca). Estas piezas serán guardadas en la carpeta “resources”, y sus direcciones se almacenarán en una nueva variable privada de la clase ***BoardView***, accesible mediante el método getter ***getPaths()***.
5. Una vez creado el tablero, se le añade todo lo necesario (***Controlador***, adición como ***Observer***, etc) y se resetea tanto el ***Modelo*** como las pilas de comandos dentro de ***Controller***.
6. Por último, se desordenan las piezas para que el usuario pueda proceder a su resolución.

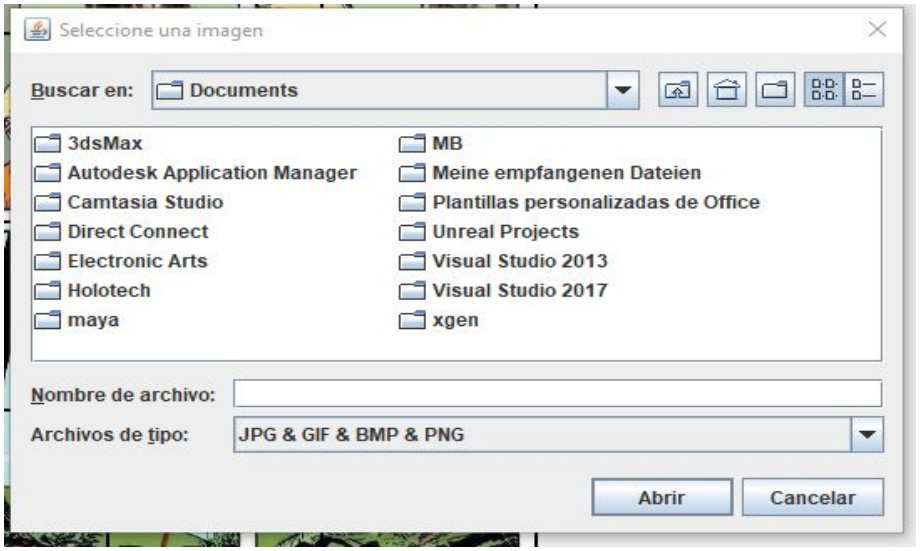

Imagen 2: Selector de imágenes

1.4 Manejo de eventos en Controlador

El objetivo principal de la clase Controlador es el manejo de eventos, que se realiza mediante dos métodos: ***actionPerformed()*** (Que maneja eventos de botones) y ***mouseClicked()*** (Que maneja los eventos de ratón en la ventana).

La acción principal y básica de la aplicación, sería el movimiento de piezas. Esto implica los eventos de ratón, junto a distintos eventos de botones.

- Al pulsar sobre la ventana, ***mouseClicked()*** recibe un evento con información de la posición del ratón en pantalla. A partir de esta información, creamos un ***MovCommand*** (Que hereda de Command), cuyo constructor se encarga de llamar al método ***movePiece()*** de ***BoardView***, comprobando qué pieza ha sido pulsada (Método ***locatePiece()***) y si el movimiento es posible o no, devolviendo un array con la posición en la lista de posiciones de la pieza actual y la pieza blanca. ***MovCommand*** se encarga de, si el movimiento no es null, llamar a ***notifyObservers()*** del ***Controlador***, actualizando tanto la ***BoardView*** como el ***Modelo*** de datos. El ***Modelo*** de datos es el que se encarga de comprobar si un movimiento lleva a la victoria o no.
- Todos estos ***MovCommand*** son almacenados en una pila en ***Controlador***, permitiendo posteriormente el funcionamiento de los botones deshacer, desordenar… Junto con otra pila que almacena los movimientos deshechos, si deseamos rehacerlos uno a uno. De esta forma, estamos implementando los requisitos de movimiento de piezas, desordenar y resolver automáticamente, junto con el mensaje de victoria.

Por otro lado, tenemos los eventos por comandos creados en PuzzleGUI, que son gestionados por el método ***actionPerformed()*** en Controller:

- Load: Permite cargar una imagen en el rompecabezas.
- Exit: Permite salir de la aplicación.
- Info: Muestra un archivo con la información relativa a la aplicación.
- Clutter: Desordena las piezas del puzzle.
- Solve: Resolución automática del puzzle.
- Deshacer: Deshace el último movimiento realizado.
- Rehacer: Rehace el último movimiento realizado.
- Guardar: Guarda la partida actual en un archivo Json.
- Cargar: Recupera la última partida jugada.

1.5 Funcionamiento general con MVC

Hemos mantenido la estructura y funcionamiento central de la aplicación, ciñéndonos al patrón MVC, apoyado por los patrones Command y Observer.

Cuando el usuario realiza el movimiento de una pieza, el procedimiento a seguir a nivel interno de la aplicación es el siguiente:

1. Se crea un ***MovCommand***:
    a. Si es a partir de un evento de ratón, se crea con la posición del ratón en la pantalla, y su constructor se encarga de obtener los índices de pieza blanca, pieza pulsada y certifica si el movimiento es posible mediante el método ***movePieze()*** en ***BoardView***.
    b. Si es un movimiento aleatorio, se crea directamente con un array de índices, obtenido del método ***getRandomNextMovement()*** en ***Modelo***. Este método obtiene las piezas vecinas a la pieza blanca desde el método ***Vecinos()***, y conoce la posición anterior de esta. Desordena de forma aleatoria el ArrayList de vecinos, y comprueba el primero en la lista que sea un movimiento posible y que no envíe a la pieza blanca de vuelta a su posición previa.
2. Se ejecuta el método ***redoCommand()*** de este ***MovCommand***, el cual llama al método ***notifyObservers()*** de Controlador, que llamará a los métodos ***update()*** tando de ***BoardView*** como de ***Modelo***.
3. Estos métodos ***update()*** intercambian las piezas de posición dentro de sus respectivos ArrayList; en ***BoardView*** una lista de ***PieceView*** y en ***Modelo*** una lista de ***PieceModel***.
    a. El método ***update()*** de ***Modelo*** se encarga a su vez de llamar al método ***isPuzzleSolved()***, que comprueba si el id de los ***PieceModel*** coincide con su posición en la lista, lo que indicaría que el puzzle ha sido resuelto. En caso afirmativo, el método update() muestra en pantalla un ***JOptionPane*** con mensaje de “¡Has ganado!”.
4. Si el movimiento resultante en ***MovCommand*** es posible (No ha almacenado un valor “null”), este es almacenado en una pila de “deshacer movimientos” en ***Controlador***.
    a. Esta pila puede ser usada en los métodos invocados por los eventos ***“resolver”*** y ***“deshacer”***: extraerían todos o el primer elemento de la pila, ejecutarían su método ***undoCommand()*** (Realiza el movimiento inicial con las posiciones del array invertidas) y los almacenarían en una segunda pila “rehacer movimientos”.
    b. Esta segunda pila puede ser usada en los métodos invocados por el evento “rehacer”: extraería el primer elemento de la pila,

ejecutaría su método ***redoCommand()*** y lo almacenaría de nuevo en la pila “deshacer movimientos”.

- Esta segunda pila se vacía cuando se realiza cualquier movimiento de desordenar o mover, dado que en caso contrario, permitiría mover piezas lejanas a la pieza blanca.
- Ambas pilas son vaciadas ante cualquier cambio de puzzle, ya sea una nueva imagen y distribución, o la carga de una partida anterior.

1.6 Guardar y cargar partida

Se ha añadido a la aplicación la posibilidad de guardar y cargar partida.
Para ello, desde la nueva clase ***Configuracion*** (Ya comentada en el apartado 1.2) con métodos estáticos, se almacena la información en formato Json, empleando la librería gson.

Para hacer más fácil el guardado y acceso a la información, creamos una clase auxiliar: ***PartidaLD***, que almacena el ***Modelo*** y la pila de ***MovCommand*** necesaria para resolver el puzzle.

Así, el almacenamiento en fichero se constituye en una única clase directa, y la lectura se realiza parseando a esta clase, e invocando a un nuevo método ***CargarPartida()***, que reinicia el ***Modelo*** en ***Controlador*** y la ***BoardView*** en el nuevo método de ***PuzzleGUI initCarga()*** (Con un funcionamiento similar a ***updateBoard()*** pero empleando los parámetros del ***Modelo*** de la partida guardada) y volcando la pila de ***MovCommand*** de nuevo sobre la pila actual de ***Controlador***. A estos métodos de la clase ***Configuracion*** se accedería desde el manejador de eventos de ***Controlador,*** mediante el evento “cargar”, añadido adicionalmente en ***PuzzleGUI*** junto al evento “guardar”, dentro de un menú desplegable en el panel superior denominado “Partida”.

Implementamos un nuevo método en ***BoardView*** llamado ***imagenesGuardadas()***, que almacena cada una de las imágenes del array de PieceView en una ruta distinta, y actualiza el array de rutas con estas direcciones. Este método es invocado antes de almacenar una nueva ***PartidaLD*** en fichero, permitiendo que estas imágenes ya recortadas no sean sobreescritas y pudiendo actualizar las rutas del array de ***PieceModel*** en ***Modelo***, haciendo posible el generar puzzles de distintas imágenes sin corromper los datos de la partida guardada.

Enlace a nuestro repositorio de desarrollo

Por si se desea, se facilita el enlace al repositorio de Github donde se ha ido desarrollando el código del programa: Repositorio Práctica GDDMD.