Call Of Pokemon: Memoria

Software de Comunicaciones

Borja Rodríguez
Joan Bru Catalán
Juan Carlos Morales
Daniel Sevilla

ÍNDICE

Objetivos del juego

El juego consiste básicamente en vencer a los otros jugadores en partidas online. Se inicia el servidor con las características deseadas (número de jugadores por partida, tamaño del mapa…) y luego cada usuario se conecta.

Lo primero que se debe hacer es registrarse o iniciar sesión, ya que el juego debe poder asociar los diferentes resultados de cada usuario en la base de datos. Seguidamente se llega al menú donde se le presentan diversas opciones para escoger.

Entre todas ellas, la primera Start New Game le permite al usuario elegir un personaje para empezar a jugar una partida siempre que no haya ninguna en curso. Una vez hecho esto, aparecerá una pantalla de espera mientras se espera a los otros usuarios por conectarse.

Una vez se han conectado todos, empieza la partida. En la pantalla de cada usuario se muestra el mapa con cada personaje inicializado en una casilla aleatoria. Los jugadores deberán ganar siendo los últimos en quedar de pie, o, dicho de otro modo, los últimos en quedar con toda la vida que se muestra encima del mapa.

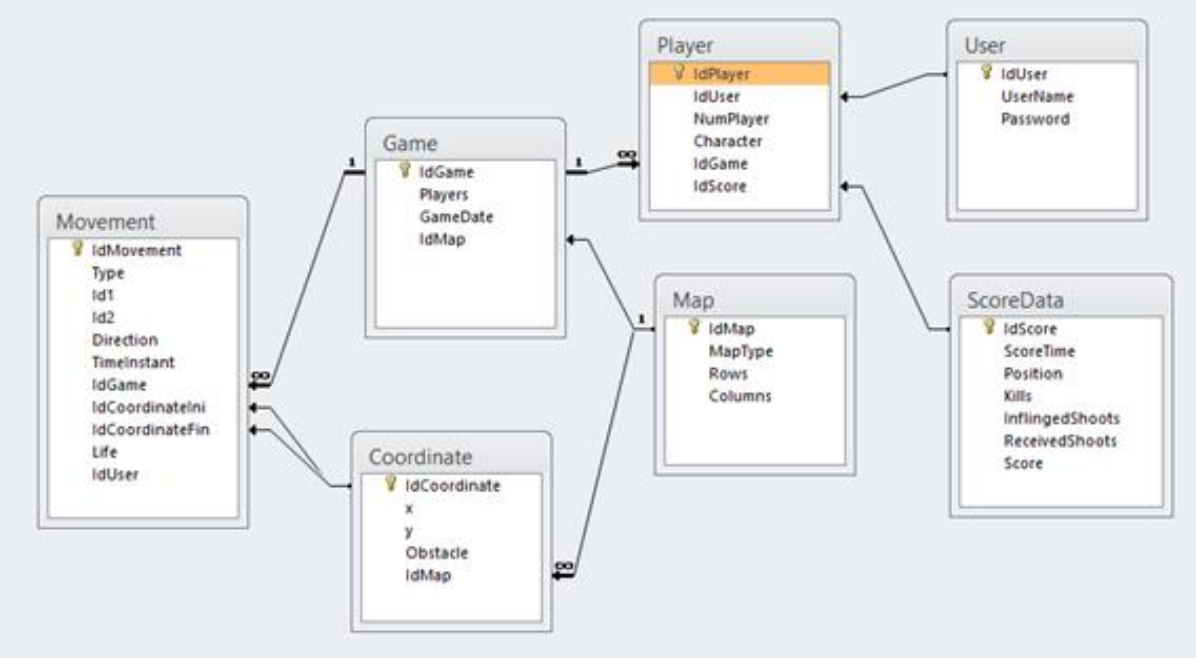
Para ello disponen de una serie de controles: para moverse y para atacar. Los movimientos varían con según qué personaje en cuanto a velocidad y obstáculos que pueden atravesar. Los ataques, que se dividen en cortos y largos, también varían en función de cada personaje en cuanto a velocidad, duración, daño y tipo.

A lo largo del juego se irán calculando parámetros de cada usuario como las veces que ha matado a otro, las veces que lo han dado… Todo para poder calcular una puntuación al final. Una vez acabado el juego, el ganador y solo este tendrá la opción de guardar la partida para poder verla en el reproductor de partidas del menú cuando quiera. Cada usuario solo puede tener una partida guardada.

A medida que se juega, se van acumulando puntos hasta que el usuario puede desbloquear nuevos personajes más fuertes.

Base de datos

El juego dispone de una base de datos para guardar diversos tipos de información en memoria. Esta fue hecha en Microsoft Access. A continuación, se presenta el esquema de relaciones de dicha base junto con sus tablas:



Para más información sobre cada campo, leer la explicación que se encuentra en el archivo "callofpokemondb.mdb" al lado de cada elemento.

Los tipos de datos que guarda la base se podrían clasificar de la siguiente forma:

- Información de usuario: la base necesita poder guardar el nombre y contraseña de los usuarios. Se trata de la tabla User.

- Información de una partida: la base requiere guardar los datos de un jugador durante una partida (puntuación, personaje escogido, fecha de la partida…) así como el resultado de esta. Se trata de las tablas Player, Game, ScoreData y Map.

- Información de una grabación: relacionada con la anterior, la base también deberá guardar información gráfica de según qué partida (a elección del usuario) para poder reproducirla de nuevo (se podría considerar información de una partida, pero no todas las partidas la tienen). Se trata de las tablas Movement y Coordinate.

En cuanto al código, la base dispone de dos clases principales: DataBase y DataBaseController. Estas clases se acceden solo desde el servidor, donde este opera cuando lo ve necesario.

La primera de ellas simplemente dispone de un método para iniciar y otro para acabar la conexión con la base desde java (usando ucanaccess) y toda una variedad de métodos donde mediante código SQL operamos en ella. La clase es muy densa pero sus métodos siguen el mismo patrón, lo cual la hace muy repetitiva.

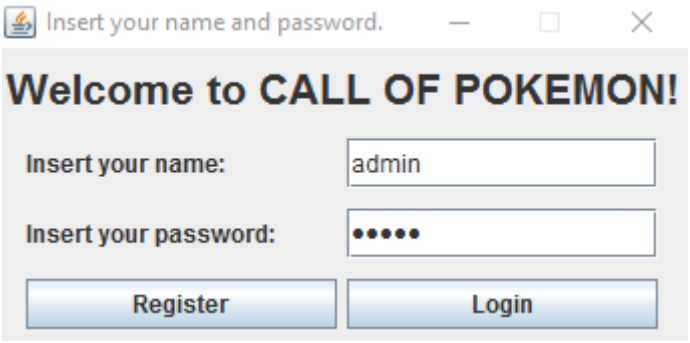
La segunda clase está diseñada para controlar que los accesos a dicha base sean en exclusión mutua. Dado que el servidor opera de forma secuencial en nuestro juego la

exclusión no supone ningún problema, pero en caso de que se quisiera ampliar (jugar varias partidas a la vez, por ejemplo) sería necesario.

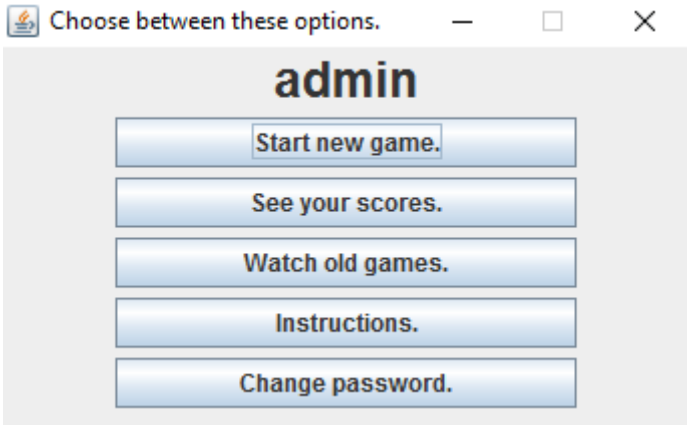
Interfaz gráfica de usuario

La interfaz gráfica de cada jugador se ha realizado mediante el patrón modelo-vista-controlador (MVC). Debido a que hay múltiples ventanas que se muestran por pantalla (registro, menú, selección de personajes ...) se ha considerado conveniente la partición en distintos módulos MVC:

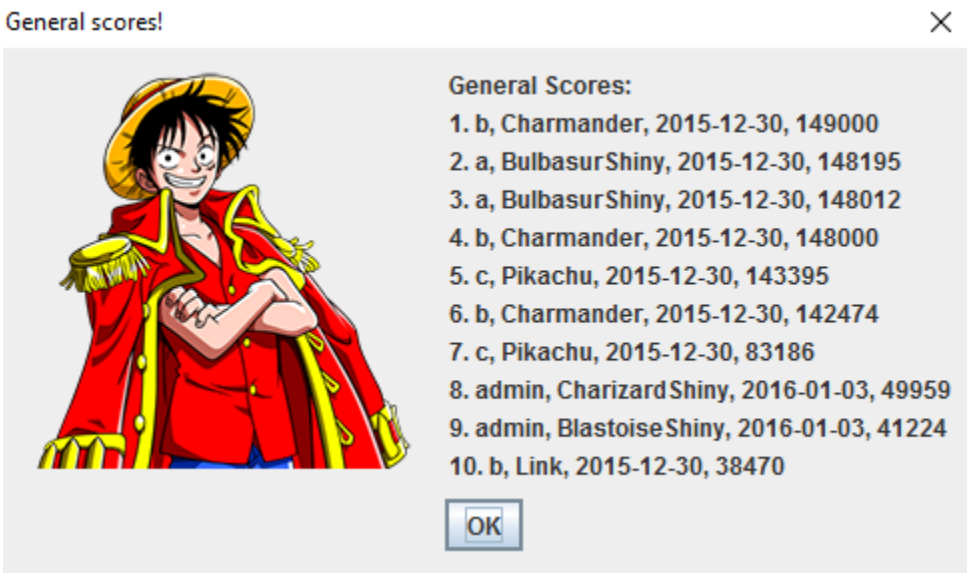
1. Login: Éste módulo muestra por pantalla una ventana donde el usuario puede conectarse al juego siempre y cuando tenga un usuario guardado en la base de datos, en caso contrario, el jugador puede registrarse en la misma ventana. Los usuarios o contraseñas vacías no están permitidas para eliminar problemas de suplantación de identidad.

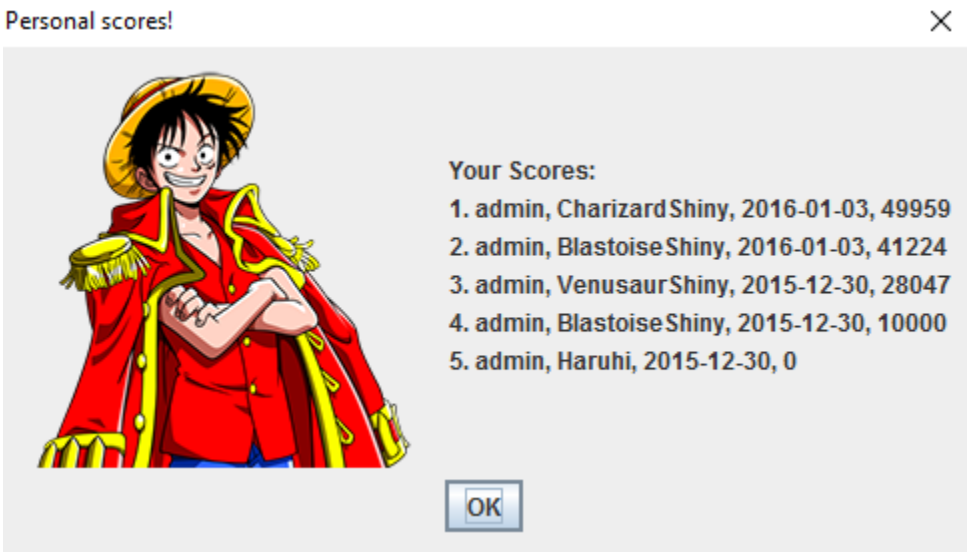


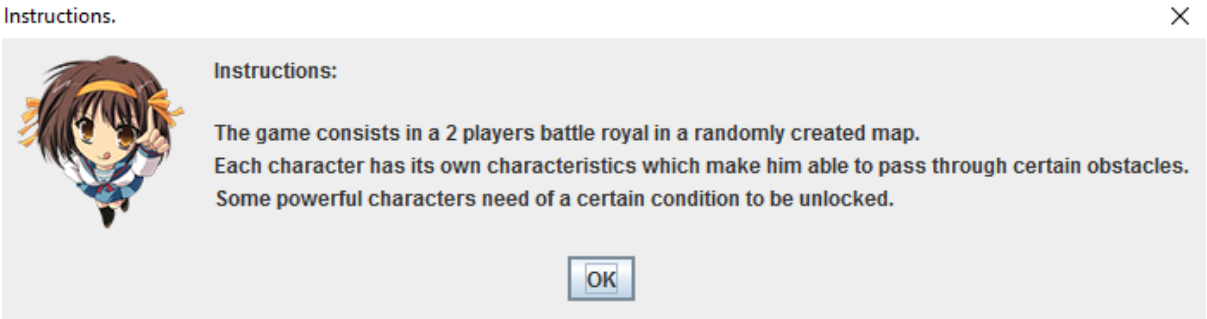
2. Menú: Éste módulo aparece justo cuando un usuario se conecta y muestra por pantalla una ventana con las distintas opciones disponibles para el jugador. El controlador (MenuController) es el encargado de regir qué se lleva a cabo según qué botón pulse el usuario.



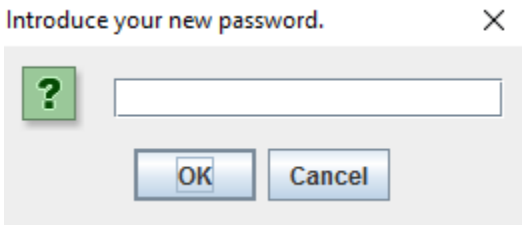
En el caso que se escojan las opciones de “See your scores” o “Instrucctions” aparecerá un Pop-Up con las puntuaciones generales y del personaje o con las instrucciones y controles del juego respectivamente.



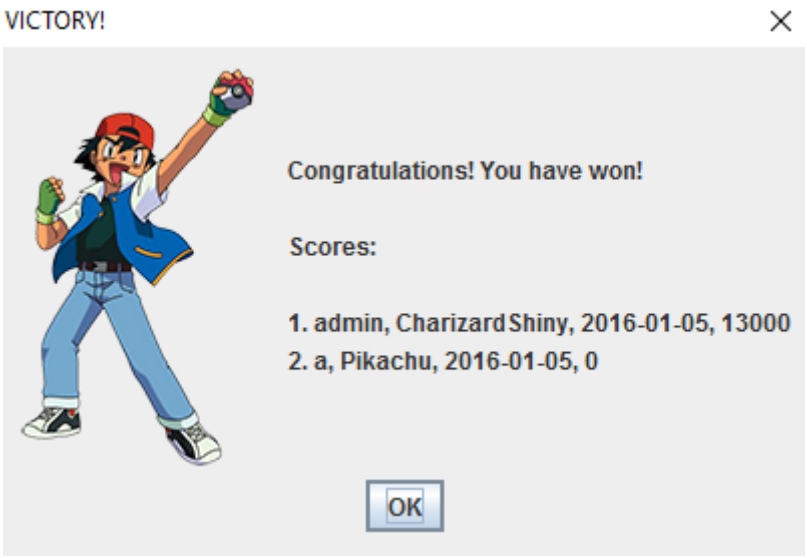


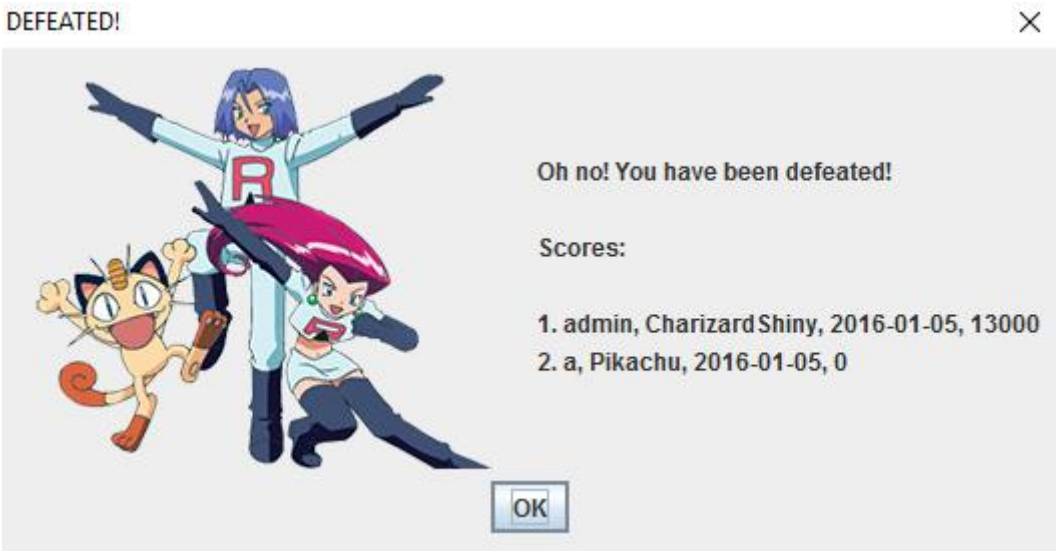


En el caso que se escoja “Change password.” se mostrará por pantalla un campo de texto donde se podrá introducir una nueva contraseña.



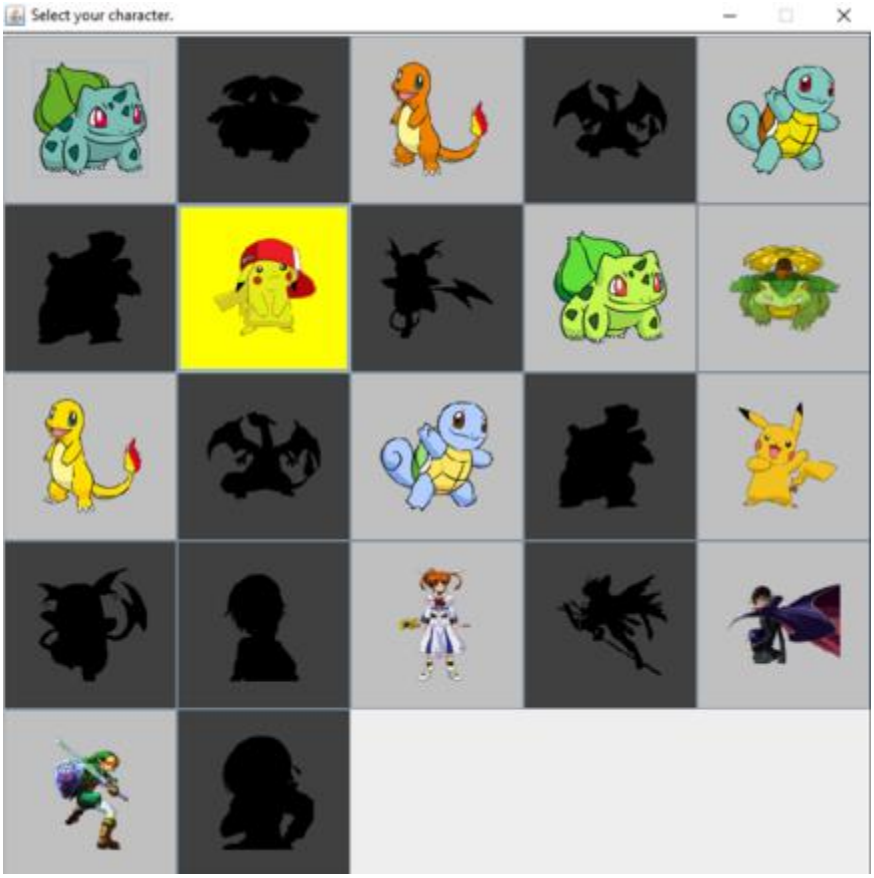
También se puede dar el caso que este módulo aparezca justo después de acabar una partida, en cuyo caso se le da la opción de volver a jugar una partida si lo desea. Antes de esto, pero, se le enseña al jugador su puntuación así como si ha perdido o ha ganado.



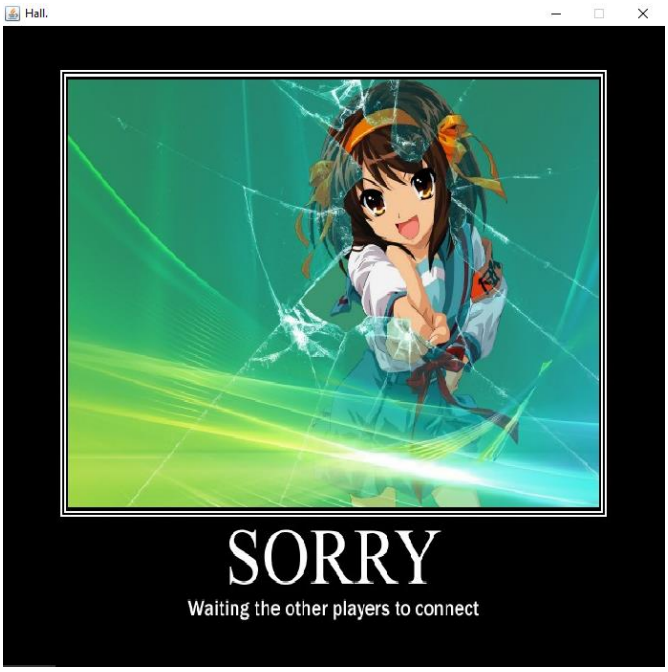


Otra opción es la de “Watch old games” que directamente mira si el jugador tiene alguna partida guardada, y en el caso que así sea muestra por pantalla esta partida junto con el tiempo que le queda a esta partida por acabar. Para su implementación otro módulo MVC ha sido creado. En este caso la vista recibe todos los movimientos que se han realizado durante la partida (lo que supone un espacio y tiempo de carga muy pequeños), y también tiene que hacer pequeños cálculos que en su momento hacía el cliente que ahora no está jugando.

3. Character selection: Tras seleccionar la opción “Start new game.” del menú este módulo muestra una pantalla con los personajes disponibles para cada jugador, así como los que no están disponibles. Estos últimos se ocultan tras su silueta. El controlador de dicho módulo se encarga de reconocer sobre qué personaje se está encima (mostrando el fondo del personaje de un color distinto) y de reconocer el personaje que se ha escogido.



4. Hall: Cuando un jugador ha elegido personaje pero todavía no se han conectado los demás jugadores aparece una pantalla pidiendo al jugador en cuestión que se espere a que los demás jugadores se hayan conectado correctamente.



5. Game: Una vez se han conectado todos los jugadores este módulo muestra por pantalla una ventana con el mapa y los personajes, así como muestra los movimientos de estos como sus ataques y sus vidas en tiempo real. En el momento que uno de los personajes muere y están jugando más de 2 jugadores el jugador muerto queda sustituido por una tumba inmóvil.

Cómo se puede apreciar, este módulo es el más complejo y por ello requiere de una gestión un tanto distinta. Cuando este módulo recibe cambios no es él quien cambia las imágenes que se muestran sinó que lo hace mediante una clase llamada “graphics” que se comentará más adelante en el apartado “Entorno gráfico”.

Todas estas vistas están regidas por un controlador especial con un nombre bastante representativo - ViewsController - que se encarga de enseñar o esconder las pantallas según convenga.

Entorno gráfico

Todo el juego consta de un entorno gráfico formado por los gráficos y los sonidos que van acompañando al jugador en su partida. Para facilitar la implementación de este entorno nos hemos creado 4 clases encapsuladas en el paquete “utilities.image” y “utilities.sound”:

1. BottomButton: Esta clase hereda de la clase “JButton” y básicamente es una clase que facilita la incorporación y modificación de imágenes en botones. Es especialmente útil en el módulo MVC de la selección de personajes.

2. BottomPanel: Esta clase hereda de la clase “JPanel” y su función es la misma que “BottomButton” pero esta vez en paneles. Esta clase es crucial para el juego porque todo el mapa se muestra a través de unos paneles colocados en forma de parrilla con las distintas imágenes. Contiene un método setImage(image) para pintar dicha imagen.

3. TreatImage: Se podría decir que esta es la joya de la corona de las clases de apoyo ya que con solo dos métodos facilita la implementación de gran parte del código. Los métodos que contiene son “combine” - se encarga de combinar dos imágenes -, que se usa cada vez que un gráfico cambia respecto el mapa original. Y “setSiluete” -dada una imagen de un personaje obtiene la silueta de este -, que se usa en la selección de personajes.

4. TreatSound: Esta clase se encarga de empezar a reproducir una canción o de parar su reproducción.

Una vez introducidas las clases de ayuda pasamos a los gráficos del juego propiamente dichos. El juego consta de una clase y una interfaz importantes para la visualización de sus gráficos:

1. GraphicsGame: Se podría decir que esta es la clase más importante de todo el entorno gráfico, ya que es la encargada de gestionar todo lo relacionado con los gráficos (creación de mapa, movimientos del avatar del jugador, creación de ataques, movimientos del avatar de los ataques, muerte del jugador con la consiguiente muestra de su tumba, cambios de vida…). Por este motivo contiene todos los gráficos del juego (mapa, personajes y ataques) junto con las barras de vida de los personajes.

2. Graphics: Esta es una interfaz cuyo único método es “changeSprites”. Las clases que implementan esta interfaz contienen una serie de imágenes que se van modificando con el tiempo y mostrando por pantalla mediante este método con la intención de crear la sensación de una animación. Es el caso de los personajes y de los ataques (GCharacter, GLargeAttack y GShortAttack). Estas clases son todas abstractas y son padres de los diferentes gráficos de los personajes y ataques específicos facilitando así el patrón fábrica usado en las clases de los personajes y de los ataques.

3. GMap: Esta clase también hereda de “Graphics” pues es un gráfico, aunque ignora totalmente el método changeSprites, por ese motivo se la menciona a parte. Su única función es la de cargar las imágenes del terreno u obstáculos del mapa según convenga.

Mapa

El mapa del juego está distribuido en forma de casillas donde cada una dispone de un cierto obstáculo. Por él se mueven los personajes y los diferentes tipos de ataques durante la partida. En cuanto a código, la clase que crea y almacena el mapa es la clase OurMap.

La función principal de esta clase no es almacenar la información del mapa (guardar sus coordenadas, como veremos), sino que su potencial reside en sus métodos de creación de dicho mapa.

En el mapa, los diferentes obstáculos actúan como paredes para según qué personajes, por lo cual es evidente que un mapa con obstáculos elegidos aleatoriamente sería casi siempre imposible de jugar, ya que los personajes podrían quedar atrapados entre obstáculos y no encontrarse jamás.

Debido a este problema, el mapa sigue el proceso de creación siguiente:

1. Generación (método createRandomMap(int type)): en esta parte del proceso, se genera un mapa del tamaño especificado con obstáculos aleatorios en cada coordenada. También se guarda el tipo de mapa que será. Aquí, el mapa creado no es “jugable”, ya que aún hay muchos obstáculos y las casillas sin obstáculos no están conectadas (no hay caminos).

2. Verificación (método verify()): en esta parte se recorre cada una de las coordenadas creadas para con un método comprobar si cumple con nuestra condición para ser “jugable”. Dicha condición es que toda casilla sin obstáculo no puede estar rodeada por más de 2 obstáculos. De esta forma se consigue que cada casilla sin obstáculo tenga como mínimo una entrada y una salida. Aunque esto no garantiza un 100% de mapas perfectos, después de varias pruebas se vio que iba bastante bien.

3. Rediseño (método lookAround(Coordinate c)) : esta parte consiste en arreglar dichas coordenadas que no cumplen con la condición anterior. Simplemente, cuando encuentra una casilla rodeada, cambia todas sus casillas por casillas sin obstáculo. Una vez que realiza un cambio, avisa de nuevo a que se verifique el mapa hasta que todas las coordenadas cumplan con la condición.

Coordenada

Cada casilla del mapa contiene una coordenada, que tiene toda la información sobre dicha casilla: dónde está, qué obstáculo tiene, si hay algún ataque o personaje allí…

La clase Coordinate se encarga de guardar dicha información en sus atributos, que son los siguientes:

1. Dos integers, “x” e “y”, que indican, cómo no, donde se encuentra la coordenada gráficamente.

2. Un integer “o”, que representa el obstáculo natural que contiene.

3. Dos Object “inside” e “inside2”, que representan si contienen algún elemento activo en ese momento (ataque, personaje…).

El primero “inside” hace referencia a elementos “sólidos” que no pueden atravesar. Sería el caso de los personajes y los ataques largos. Estos colisionan al encontrarse con una roca u otro personaje o ataque.

El segundo “inside2” se usa para los ataques cortos, ya que estos sí pueden sobreponerse. Esta diferenciación era necesaria ya que los ataques cortos no se mueven y afectan solo a una casilla.

Ataques

En este juego actualmente hay dos tipos de ataques, los largos y los cortos. Ambos tienen una clase propia que los rige y esta es abstracta y “serializable”. Los ataques contienen dos métodos destacables:

1. hasCollision (int x, int y): Este método se encarga básicamente de controlar las colisiones de los ataques con obstáculos y otros ataques o personajes.

2. getGraphics(): Este método se encarga de cogerlos gráficos de cada ataque específico. Como la clase es abstracta, esta consta de múltiples hijos cada uno con unos gráficos distintos, entonces, este método funciona como patrón fábrica de gráficos de ataques. Del mismo modo, el tener múltiples clases hijas facilita la implementación de un patrón fábrica de ataques en los personajes. Todos los ataques deben conocer la coordenada en la que están, la dirección en la que miran, el mapa que los contiene y el personaje que los ha invocado.

Personajes

La clase personaje es una clase abstracta llamada OurCharacter y contiene múltiples atributos como puede ser una coordenada para la posición, el tipo del personaje (agua, fuego, eléctrico, etc), la vida, el daño base, el nombre, el tiempo entre movimientos, el tiempo entre ataques, etc. Todos los personajes del juego (22 en el momento de escribir esto) heredan de esta clase.

El método initialize() reinicia todas las variables y coloca al personaje en una posición random permitida.

Lo más remarcable de esta clase es que actúa como una fábrica de ataques largos, de ataques cortos y de gráficos, lo que hace completamente innecesario el uso de ifs para escoger el ataque o los gráficos del personaje en cuestión. Por supuesto, cada uno de estos métodos es sobrescrito en cada una de las subclases para devolver la instancia de la clase correspondiente en cada caso.

También se pueden destacar los métodos abstractos de recibir ataque largo y ataque corto y el método de comprobar colisión, que se calculan de forma diferente en cada personaje.

Clase DataPacket

Esta clase es, tal como su nombre indica, un paquete de datos. Cualquier mensaje entre cliente y servidor se realiza intercambiando objetos instancia de esta clase.

Consta de un código de control para indicar la operación, tres enteros con información relevante según el tipo de paquete y un objeto que usualmente es un movimiento.

Clase MovementsModel

A pesar de su nombre, no es estrictamente un modelo, pero dado que el servidor tiene un MovementController es razonable llamarla así.

Esta clase almacena la información de cualquier movimiento individual que se produce en el juego (moverse, cambio de vida de personaje, crear nuevo ataque largo, mover ataque largo, etc) y el tiempo en el que se produce respecto al tiempo en el que empezó el juego. En el servidor, el controlador del juego los crea y los almacena en un buffer del SendController para procesarlos y crear el paquete correspondiente a enviar a los jugadores. En el OutputController del jugador, se crea un movimiento dependiendo de la información del paquete de datos y se añade al modelo.

La principal causa del diseño de esta clase es la implementación del mecanismo de guardar partidas, del cual se habla en otro apartado. Cada movimiento es guardado en la base de datos a medida que se va reproduciendo con toda la información necesaria para poder luego reproducir partidas a partir de ellos.

Conexión cliente-servidor

La conexión cliente-servidor se realiza con el método NIO visto en clase (clases ClientNIO y ServerNIO). Cada operación de enviar y recibir mensaje del cliente se ejecuta en el controlador que toque según la vista en la que esté en ese momento. El servidor también tiene un controlador para recibir y otro para enviar.

Al abrir un cliente, este espera a poder conectarse con el servidor. Si se conecta satisfactoriamente, el servidor le envía un DataPacket diciéndole si acepta la conexión o si la rechaza. En caso de que la conexión sea rechazada el cliente se cierra.

En la primera pantalla debes hacer login o registrarte en el caso de que no tengas un usuario. Se le manda un paquete al servidor con nombre de usuario y contraseña. El servidor consulta la base de datos y le devuelve la información pertinente al cliente en función de eso (usuario ya existente, contraseña incorrecta, etc).

Una vez hecho login puedes decidir si empezar a jugar, ver tus puntuaciones, ver partidas guardadas, leer las instrucciones o cambiar la contraseña. El caso que interesa ahora mismo es el de jugar (los otros se gestionan igual que el login: mandar petición al servidor y recibir la información de la base de datos).

Al darle a jugar, el servidor comprueba si ya hay una partida empezada. Si no hay ninguna partida en curso comprueba cuanta gente hay en la pantalla de selección de personajes. En caso de no llegar al máximo, se le manda al cliente (al cual a partir de este punto le llamaremos jugador) un mensaje de aceptar juego, este mensaje contiene el número de jugador asignado al solicitante. Además, el servidor registra al jugador en una lista de SocketChannel donde están los usuarios que van a jugar (a los que se les va a hacer broadcast durante el juego). Cuando el jugador recibe el aceptar juego, abre la pantalla de selección de personajes. Los personajes que puede escoger cada jugador los determina la base de datos y esta información también es enviada.

Al elegir un personaje se le manda al servidor el nombre del personaje escogido. Y se pasa a la pantalla de espera.

Una vez que todos los jugadores han escogido personaje, el servidor inicializa el juego: Crea el mapa y los personajes que han elegido los jugadores (asignándole a cada uno su número de identificación) y hace un broadcast de cada uno de estos datos a los jugadores de la lista comentada anteriormente (metidos en su correspondiente DataPacket). Cuando se han mandado todos estos datos necesarios para poder iniciar el juego, se les manda un mensaje a los jugadores para que finalmente inicialicen la vista del juego.

Siempre que se cierre una ventana, se le manda al servidor un paquete de conexión terminada para que éste cierre el canal correspondiente. Además, según el punto en el que se cierre la ventana, se transmitirán más paquetes para no dejar el juego colgado dado que no es lo mismo cerrarla en el menú, en la pantalla de selección de personajes o en mitad del juego.

Mecanismo del juego y mensajes cliente-servidor durante el juego

La clase DataInputController, que gestiona los inputs del teclado, consta de tres threads distintos: Uno que escanea las teclas de movimiento (w, a, s, d), otro que escanea la tecla de ataque largo (y) y otro que escanea la tecla de ataque corto (u). De esta forma puede atacar mientras te mueves.

Mensajes que puede enviar el jugador y gestión por parte del servidor:

- Moverse o cambiar de dirección (teclas w, a, s, d, que equivalen a arriba, izquierda, abajo, derecha):

  En el caso de una pulsación corta que dure menos que 0.1 s, se le envía al servidor un paquete de “girar personaje” con la id del jugador y la dirección a la que intenta girar en función de la tecla pulsada. Si la tecla se mantiene pulsada más tiempo, se le envía al servidor un paquete de “mover personaje” con la id del jugador. El servidor automáticamente mueve el personaje en la dirección en la que está mirando, así que no es necesario mandar una dirección.

  La forma usada para crear este comportamiento en el DataInputController ha sido usar un timer que se inicie al pulsar la tecla y se cancele al soltarla. Este timer es el que manda los paquetes de mover personaje mientras no se cancele y se empieza a ejecutar a partir de 0.1 s al pulsar la tecla con una periodicidad que depende de la velocidad del personaje escogido.

  Cuando al servidor le llega un cambio de dirección al ReceiveController, cambia la dirección del personaje en el controlador del juego (MovementsController) y luego manda desde el SendController un paquete “girar personaje” con la id del personaje que ha cambiado de dirección y la dirección correspondiente mediante un broadcast.

  Cuando al servidor le llega un paquete de movimiento, el controlador del juego comprueba si el movimiento a realizar está permitido o no (si hay colisión con otro personaje, con alguna casilla del mapa, si se sale fuera de límites, etc). Las condiciones de colisión de cada personaje son distintas (función hasCollision(int x,int y)). En caso de ser posible el movimiento, el controlador del juego mueve una casilla el personaje y el controlador de enviar manda un paquete de “mover personaje” con la id del personaje que se ha movido y la nueva coordenada mediante un broadcast.

- Ataque largo (tecla y):

  Cada vez que se pulsa esta tecla se comprueba cuando fue la última vez que se pulsó. Si fue hace un tiempo menor que un cierto tiempo mínimo que depende del personaje (tiempo mínimo entre ataques largos) no ocurre nada. En caso contrario se manda un paquete al servidor de “crear nuevo ataque largo” con la id del personaje que lo crea.

  Al recibir este paquete, el servidor crea un nuevo ataque largo que será diferente según el personaje que lo esté creando (cada personaje es una fábrica de ataques largos). A dicho ataque se le asigna la dirección a que mira el personaje y otro número de identificación debido a que cada personaje no puede tener en pantalla tantos ataques como quiera (respetando el tiempo mínimo entre ataques al lanzarlos) y, puesto que cada ataque se gestiona de forma diferente, no pueden tener todos el mismo id del personaje que los ha lanzado.

  Una vez creado el ataque, el controlador del juego pasa a ejecutarlo mediante un timer que se ejecuta según la velocidad de movimiento del ataque largo del personaje (es diferente para distintos personajes). A cada vencimiento del timer el controlador intenta mover el ataque una casilla y para ello comprueba toda una casuística de colisiones (colisión con terreno, con borde de pantalla, con jugador, con otro ataque largo y con un ataque corto).

  En el caso de suceder una colisión con terreno (cada ataque puede colisionar con terrenos distintos), con el límite de la pantalla o con un ataque corto, el ataque es destruido sin más y se detiene el timer. En el caso de colisión con otro ataque largo, ambos son destruidos y se detienen sus timers (es una buena forma de aumentar la experiencia de juego al permitir protegerse mediante ataques largos o cortos). En el caso de colisión con un personaje que no sea el mismo que los ha lanzado se procede a calcular el daño realizado al personaje, que es diferente dependiendo del personaje que reciba el ataque (función receiveLargeAttack (LargeAttack a) en OurCharacter).

  Después de recalcular la vida del personaje se manda un paquete con la id del personaje que ha recibido daño y su nueva vida mediante un broadcast, se comprueba si ha muerto y, de ser cierto, se manda un nuevo paquete con la id del personaje muerto mediante un broadcast y se comprueba si el número de personajes vivos es 1 para acabar el juego. Si la partida acaba se manda un paquete de acabar partida a todos los jugadores mediante un broadcast.
  Cada vez que se mueve un ataque largo se manda un paquete de “crear nuevo ataque largo”, “mover ataque largo” o “terminar ataque largo” según si es la creación del ataque, si es un movimiento normal o si ha habido colisión. Dichos paquetes contienen la id del ataque largo y, en el caso del paquete para crear un nuevo ataque, también se manda la id del personaje que crea el ataque.

- Ataque corto (tecla u):

  Es parecido al ataque largo con la diferencia de que es estacionario y tiene una duración determinada, por lo que el paquete de “mover ataque corto” no existe.

Al igual que en el ataque largo, existe un tiempo mínimo entre ataques cortos que, usualmente, es mayor que el tiempo mínimo entre ataques largos.

Cuando el servidor recibe un paquete para crear un ataque corto, primero obtiene el ataque correspondiente del personaje (los personajes también son fábricas de ataques cortos) y se le asigna una id. Luego el controlador del juego lo ejecuta mediante un timer que se ejecuta N veces con intervalos T entre llamadas. N es un parámetro que depende del ataque en cuestión mientras que T vale 0.1 s.

En la primera ejecución del timer, el controlador comprueba si no se está intentando ejecutar el ataque corto en una posición prohibida. Si es una posición permitida manda un “crear nuevo ataque corto” con la id del personaje y la id del ataque corto. En caso contrario se elimina el ataque y se cancela el timer antes de tiempo.

En todas las ejecuciones del timer, comprueba si en la misma casilla del ataque corto hay un personaje distinto al que ha lanzado el ataque y se calcula el daño de la misma forma que en el ataque largo. Los paquetes referentes al daño, a la muerte y al fin del juego también son idénticos.

- Instakill (cerrar ventana):

  Realmente no es un comando, pero si un jugador cierra su ventana, se le manda un mensaje al servidor que mata a su personaje instantáneamente, mandando los paquetes correspondientes como se ha explicado en el ataque largo, y lo borra de la lista de jugadores a los que se le hace broadcast.

El jugador después de recibir cada paquete setea los cambios correspondientes en el modelo, añade el movimiento y avisa a la vista para pintar.

De vez en cuando ocurre que al hacer un broadcast hay algún jugador al que no le llega el DataPacket. Esto no es un problema en el caso de moverse o cambiar de dirección (al volver a moverte se actualiza la nueva posición y es como si nunca se hubiese perdido el paquete). Sin embargo, en el caso de perder el paquete de “crear nuevo ataque largo” o “terminar ataque largo” puede ocurrir que se intente pintar un ataque que no existe o se quede un ataque fantasma en pantalla, por lo que hay una par de funciones de control en el controlador del jugador que le preguntan al servidor si hay un ataque activo o no cuando venza un timer o cuando le llegue un paquete de “mover ataque largo” con una id que no exista.

Mecanismo de guardar y reproducir partidas

Esta parte del proyecto es una de las más interesantes y complejas y en la cual participan tres de sus grandes partes: el servidor, la base de datos y la interficie gráfica. El trabajo de esta parte consiste en guardar de alguna forma las partidas que se juegan para poderlas reproducir de nuevo si se quisiese. Todo esto se consigue gracias a la clase MovementsModel.

Dicha clase nos permite guardar en ella toda la información necesaria que contiene un movimiento en la partida, por lo cual el servidor solo tiene que ir almacenandolos en orden (les pasa un parámetro con el instante de tiempo en que se producen) y luego guardarlos en la base.

Para limitar la implementación de dicho mecanismo, se ha diseñado con las siguientes condiciones:

- Solo el jugador que gana una partida recibe la opción de guardarla para reproducirla en cualquier momento.

- Cada jugador solo puede tener una partida guardada (para facilitar la asociación de movimientos en la base y limitar su tamaño). En caso de querer guardar otra teniendo ya una guardada, se sobreescribirán.

Una vez aclarado esto, el proceso seguido es el siguiente:

1. Empieza una partida. El servidor va almacenando en un array todos los movimientos que va recibiendo hasta el movimiento final (incluido) que lleva el prefijo GameEnded.

2. Acaba la partida. El vencedor recibe un popUp preguntándole si quiere guardar o no la partida justo después de mostrarle los scores.

3. Caso afirmativo. El servidor mira si el usuario tiene ya alguna partida guardada. Si es así, elimina de la base todos los movimientos de dicha partida y, además, tambíen las coordenadas del mapa en dónde se jugó la partida. Esta última eliminación es posible debido a que cada partida solo puede tenerla guardada un jugador, por lo que si ninguno la está guardando, nos podemos ahorrar de guardar también la información del mapa, que no se tendrá que reproducir nunca. Finalmente, el servidor guarda en la base los movimientos junto con la información del usuario y del juego necesarias.

4. Caso negativo. El servidor vacia el array de movimientos y elimina las coordenadas del mapa de dicha partida de la base.

Una vez guardada la partida, solo es cuestión de tiempo hasta que el usuario decida reproducirla. Para ello tiene la opción “Watch old games” en el menú, que se encarga de ello.

Cómo se anunció en el apartado de interfaz gráfica de usuario, un módulo MVC se ha creado para la realización de la vista de los juegos antiguos. Este módulo es muy parecido al de “Game” con la pequeña diferencia que ahora no hay interacción directa del usuario, sino que se consta de un array de movimientos secuencialmente guardados con el tiempo en el que se ejecutaron. De este modo, la vista sólo se encarga de ejecutar estos movimientos en el momento que precisa como si de un jugador en tiempo real se tratase. Además, debe ejecutar algunos “seteos” de posiciones que en su momento ejecutaba el jugador y que ahora por razones obvias no lo hace.