Proyecto Final. Redes y Videojuegos en Red

Instrucciones para la entrega

- **Rellenar esta memoria** con la memoria del proyecto.
- Si el proyecto se realiza en **pareja** sólo es necesario **rellenar una memoria del proyecto**.
- El **código** del proyecto se entregará usando www.github.com.
- La **evaluación** del proyecto tendrá en cuenta el último commit antes de la fecha de entrega. No se considerarán commits posteriores.
- El **proyecto deberá defenderse en una presentación con el profesor**. En el acto de presentación los alumnos responderán preguntas sobre el proyecto realizado.

Formulario de entrega

Título	Magic Maze	
Alumno	Apellidos y Nombre	Eva Sánchez Muñoz
	Usuario GitHub	evsanc07@ucm.es
Alumno	Apellidos y Nombre	Emile de Kadt
	Usuario GitHub	edekadt@ucm.es
URL del repositorio		https://github.com/edekadt/totally-not-magic-maze

Memoria del Proyecto

0. Breve Descripción:

Magic Maze es un juego multijugador cooperativo, donde dos jugadores tendrán que conducir un grupo de personajes hacia las salidas del edificio donde están situados.

El proyecto está basado en un juego de mesa llamado Magic Maze[1] donde una serie de jugadores, cada uno con diferentes tipos de acciones, deben mover a los personajes por un laberinto hasta sus respectivas salidas sin comunicarse de forma verbal entre ellos. En esta versión, hemos decidido simplificar las reglas originales, eliminando el tiempo límite del que disponen los jugadores para finalizar la partida, las casillas especiales (como, por ejemplo, los portales o las escaleras) o la composición del mapa, con una mecánica similar a la presente en el dominó[2].

De forma dinámica, cada jugador deberá utilizar los comandos que tenga asignados para desplazar a todos los personajes en una dirección o, si la situación lo requiere, saltar el turno. Los comandos estarán divididos en dos, correspondiendo los del primer jugador al eje vertical, y los del segundo al eje horizontal.

Por una decisión de diseño, se decidió que los comandos de teclado moverían a todos los personajes a la vez, similar a la mecánica de desplazamiento que podemos encontrar en juegos como el 2048[3], en contraste con los movimientos individuales que podemos realizar en el juego original.

Para terminar una sala, los jugadores deberán conducir a cada personaje a la casilla de salida del mismo color, siendo el máximo número de personajes en una misma sala cuatro y el mínimo dos. Tras finalizar, el juego generará una nueva sala diferente, con un nuevo número de personajes aleatorio.

1. Arquitectura del Juego:

1.1 Modelo del juego:

En la versión original del proyecto planteamos un modelo basado en la topología de **Cliente-Servidor**[4], donde dos clientes estarían conectados a un servidor que controlaría todos los aspectos del juego. Sin embargo, durante el desarrollo del proyecto, se decidió que sería mejor solución implementar una de las variaciones de susodicha topología, siendo el propio servidor uno de los dos clientes.

Este modelo implica que el jugador que posea el rol de host (Es decir, el cliente encargado de funcionar como un servidor) deberá poseer una buena capacidad de red, ya que poseerá el conjunto de sistemas que desarrollarán la lógica interna del juego, además de los sistemas que compartirá con el jugador invitado (En este caso, el cliente conectado al servidor creado por el host)

1.2 Estado del juego:

El estado del juego vendrá dado por el número de jugadores conectados.

En el caso de haber dos jugadores conectados al mismo servidor, el estado del juego será el estado de **ejecución,** donde ambos jugadores podrán desplazar a los personajes a través del mapa con una serie de comandos establecidos según su rol (Host o invitado) Por otro lado, si solo hay un jugador conectado, dependiendo del rol que posea, podremos encontrarnos con una partida en el estado de **pausa** o inexistente.

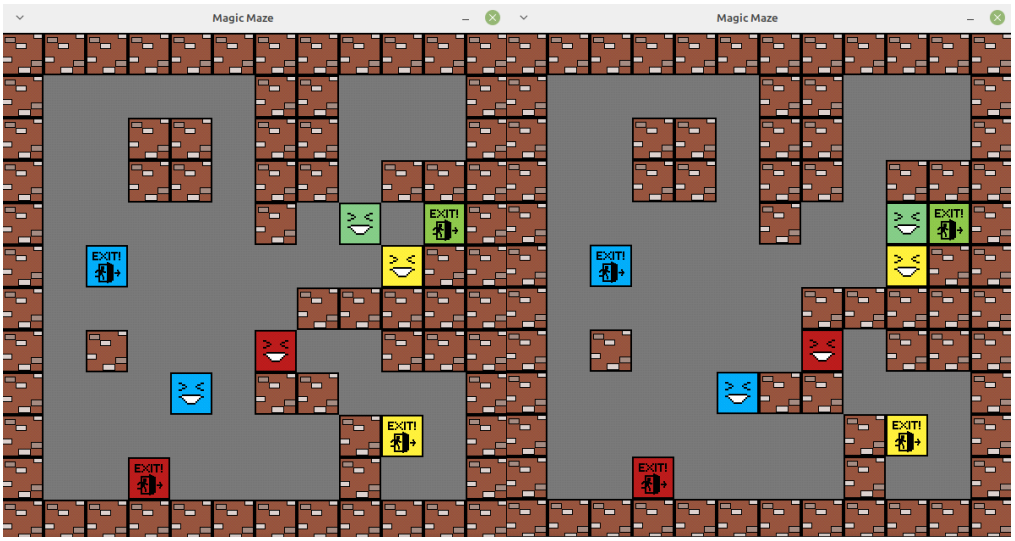
Tendremos dos ejecutables, uno para el cliente y otro para el servidor. Cuando se lanza el ejecutable del servidor, este abre un servidor en el puerto y la dirección que reciba como argumentos. Gracias a estos dos ejecutables, se decidirá qué jugador realizará el rol de host (es decir, el jugador que jugará con el cliente-servidor) y cuál el rol de invitado (el jugador que jugará como cliente).

Si el invitado decide desconectarse, la partida entrará en el estado de pausa hasta que otro cliente ingrese a la partida. Si, por el contrario, el host es quien decide desconectarse, la partida finalizará de forma inmediata.

1.3 Objetos:

En el juego podremos encontrar dos tipos diferentes de objetos: los héroes y el mapa.

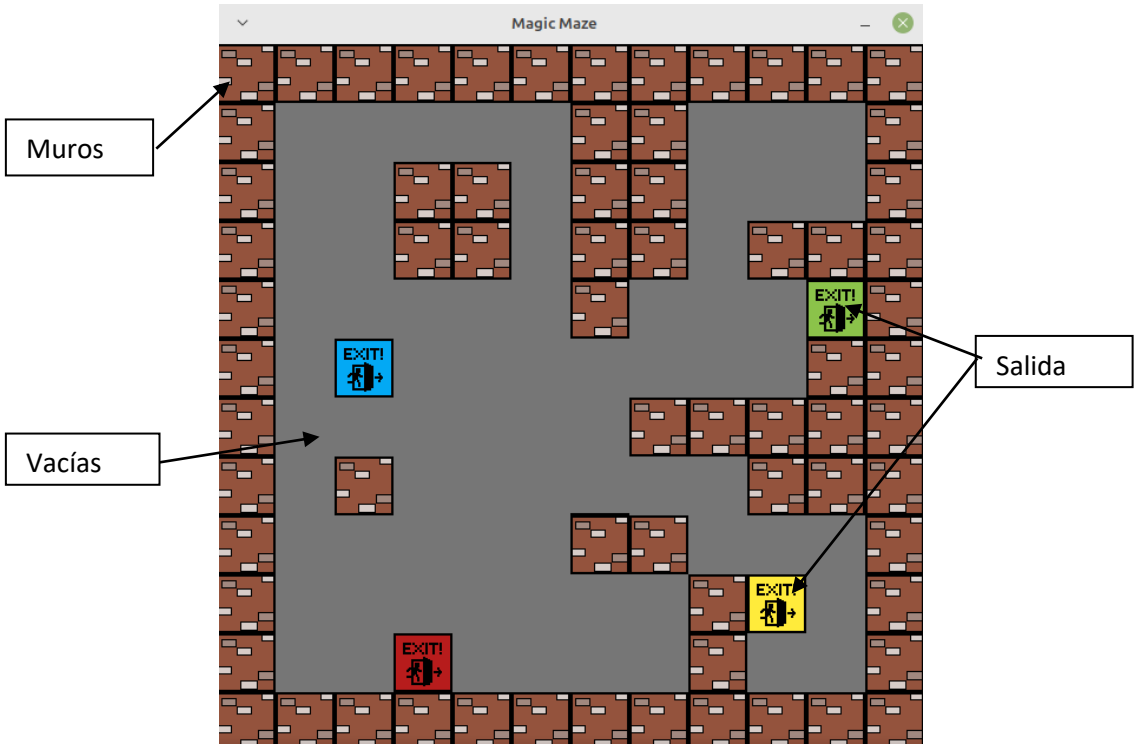
Los **héroes** son los objetos interactuables del juego. Los jugadores podrán controlar su desplazamiento con las flechas de dirección del teclado, moviéndose todos los héroes presentes en el tablero una casilla en la dirección seleccionada, siempre y cuando pueda realizar dicho movimiento.



En las dos imágenes de arriba, podemos ver un ejemplo del desplazamiento de los personajes por el mapa. En la primera imagen, podemos apreciar la posición de los héroes antes de que uno de los jugadores realice una pulsación de teclado, en este caso, la flecha de dirección derecha. Tras ejecutar la lógica del movimiento, todos los héroes han sido desplazados una casilla hacia la derecha, menos el héroe de color amarillo, que no puede realizar susodicha maniobra debido al obstáculo que hay en el camino y, por tanto, permanece en su posición actual.

Cada héroe poseerá un identificador propio, con el cual determinaremos su aspecto (es decir, el color que lo representa: azul, rojo, verde o amarillo) y cuál será su casilla de salida (esta casilla tendrá el mismo color del héroe, aunque formará parte del mapa). Además del identificador ya mencionado, los héroes pertenecerán a un grupo que permitirá identificarlos a la hora de realizar diferentes tipos de maniobras, ya sea indicando que han conseguido alcanzar la salida o bloqueando su movimiento, como podemos apreciar en el ejemplo anterior.

El otro objeto que podemos encontrar en el juego es el **mapa**, compuesto por casillas de tres tipos: muros, salidas y vacías. Los héroes podrán estar situados en las casillas vacías (que son casillas representadas con el color gris) y en las casillas de salida (en cada sala habrá el mismo número de salidas y héroes, habiendo una salida para cada héroe), pero no en las casillas de muros.



El mapa cambia en cada sala, habiendo implementado un algoritmo que genera de forma semialeatoria el mismo. Este algoritmo está encargado de posicionar a los personajes en el mapa, generando una serie de caminos de longitud aleatoria entre 20 y 40 pasos. El final de cada camino se utilizará para establecer el punto de inicio de cada héroe. Una vez establecidos todos los caminos, el algoritmo crea muros en las casillas restantes del mapa, terminando de formar el diseño del nivel.

Un punto que debemos resaltar respecto al mapa es la existencia de un nivel diseñado a mano, utilizada como primera sala en cada partida. Es un nivel sencillo, con cuatro personajes, que funciona como un pequeño tutorial para introducir las mecánicas del juego.

1.4 Replicación:

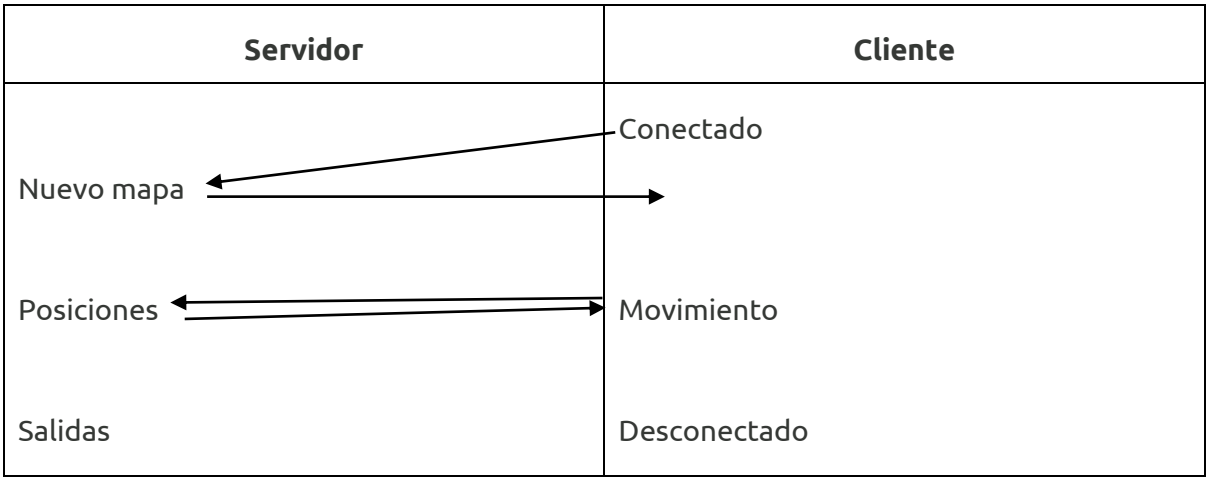
El juego utilizará un modelo **servidor-cliente UDP**, debido a que buscamos comprobar con cada envío de mensajes, quien es el jugador que ha realizado la pulsación de teclado, para comprobar si ha pulsado las teclas correspondientes o, en el caso de salir del juego, cual es el protocolo que debemos seguir.

Tras inicializar el juego, el servidor enviará un mensaje de *nuevo mapa* y *salidas* al cliente, que contiene toda la información que corresponde a la distribución de las diferentes casillas del mapa y la posición inicial de los personajes.

Durante el transcurso del juego, el cliente mandará mensajes de *movimiento* al servidor. El servidor calculará los resultados y enviará un nuevo mensaje de *posiciones* al cliente, que contendrá solo las nuevas posiciones de los personajes, tanto las que han cambiado como las que no. De este modo, el cliente no necesita realizar los cálculos detrás de sus movimientos, encargándose de ello el servidor, que sincronizará ambos clientes.

En el caso de llegar a su correspondiente casilla de salida, el personaje será eliminado y desaparecerá del mapa. Si todos los héroes han sido eliminados, se generará un nuevo mapa de juego y se realizará el envío del mensaje de *nuevo mapa,* llamándose también al mensaje que actualizará las *salidas*.

Aparte de los mensajes mencionados, tendremos mensajes de *desconectado* y *conectado,* que solo serán enviados por el cliente cuando este se conecta o desconecta al servidor.



El mensaje de nuevo mapa solo se envía dos veces, una cuando se inicia el juego y otra cuando todos los personajes han llegado a la salida. Junto a nuevo mapa enviaremos el mensaje de actualizar las salidas. Además, los mensajes de movimiento y posiciones son enviados de forma constante entre cliente y servidor hasta finalizar el juego, ya que no existe un límite de movimientos que pueda realizarse.

1.5 Protocolo de aplicación:

El protocolo de aplicación que ha sido utilizado en el proyecto es el protocolo UDP, debido a que buscamos que haya un envío constante de mensajes entre cliente y servidor, sin necesidad de que el programa realice comprobaciones del contenido del mensaje.

1.6 Serialización:

El envío de los mensajes se realiza de forma binaria, en bytes. Los mensajes transmitidos luego son deserializados en otro mensaje.

A pesar de que el cliente y el servidor no pueden enviar todos los mismos tipos de mensajes, es importante que ambos los reconozcan, ya que deben deserializar aquellos que no puedan enviar y viceversa.

Los métodos empleados para la realización de los procesos de serialización y deserialización podemos encontrarlos en Chat, con el nombre de **to_bin** y **from_bin**.

2. Diseño del Servidor:

El servidor del juego está encargado de gestionar la lógica de este, creando los algoritmos para generar el mapa, calcular las posiciones de los personajes y detectar el final del juego.

En el servidor podemos encontrar los sistemas creados para gestionar las operaciones mencionadas: HeroSystem y MapSystem. Además, tanto el cliente como el servidor tendrá su propia clase de Game, que gestionará su bucle principal.

GameMessage hereda de la clase Serializable, y está encargada de traducir los mensajes que llegan o decidimos enviar, usando para ello los métodos de serialización **to_bin** y **from_bin**. En ambos métodos se han utilizado *union types*, para reducir el tamaño máximo de memoria que ocupan los mensajes.

Estos mensajes son utilizados en el bucle principal del juego, donde hay implementado un thread que conecta con el método **do_messages,** mientras los sistemas realizan sus actualizaciones durante cada vuelta del bucle**:**

- **CLIENTJOINED:** El servidor recibirá el aviso de que un cliente se ha conectado.
- **CLIENTLEFT:** El servidor recibirá el aviso de que un cliente se ha desconectado, eliminando la información referente al cliente.
- **MOVEMENT:** Dependiendo de la dirección recibida, el servidor desplazará los personajes en una dirección u otra, y dará el aviso de que hubo movimiento.
- **NEWMAP:** En el caso de recibir este mensaje, el servidor lo ignorara, ya que solo el servidor puede enviar este mensaje.
- **UPDATEPOS:** En el caso de recibir este mensaje, el servidor lo ignorara, ya que solo el servidor puede enviar este mensaje.
- **UPDATEEXITS:** En el caso de recibir este mensaje, el servidor lo ignorara, ya que solo el servidor puede enviar este mensaje.

Existe una debilidad de sostenibilidad del código en lo concerniente a los mensajes: al estar declarados independientemente en el código del servidor y el cliente, cualquier cambio a los mensajes debe escribirse en ambos sitios para garantizar una serialización correcta. Este problema (declaración doble de código común) se extiende a todo el proyecto, pero es más problemático en los mensajes.

Dejando a un lado el apartado relacionado con la red, HeroSystem está encargado de gestionar el comportamiento de los personajes del juego. Posee un método público llamado **move,** que puede activarse al recibir un mensaje de MOVEMENT. En este método, añadimos a las posiciones de todos los héroes del tablero los valores x e y recibidos. Sin embargo, antes de cambiar la posición, comprobamos que la nueva posición no corresponde con un muro u otro personaje con el método privado **checkMove.** Para comprobar si los héroes han llegado a su casilla de salida, utilizamos el método update de MapSystem, eliminando los héroes y generando una nueva sala si todos han sido eliminados.

Además, en el HeroSystem podemos encontrar el método que tiene como propósito codificar los movimientos que han realizado los personajes. Esto será enviado a través de otro método en Game, quien a su vez se lo dará al socket.

MapSystem es el otro sistema principal del servidor, donde generaremos las salas al comienzo de cada nivel. Para ello habrá dos métodos: **load**, que leerá el nivel de tutorial de un archivo de texto; y **generateLevel**, que utilizará el algoritmo de creación aleatoria para diseñar la sala. Además, en MapSystem podemos encontrar un método para limpiar los grupos de entidades tras finalizar con cada sala, consiguiendo que no queden almacenadas las posiciones de objetos entre salas.

En MapSystem tendremos un método que creará una cadena de valores de texto con la información del mapa, siendo esta enviada a un método de Game. Este método estará encargado de mandar la información a través del socket. De la misma forma, nos encontraremos con un caso similar en las salidas.

3. Diseño del Cliente:

El cliente del juego tan solo podrá enviar mensajes al servidor y renderizar su propia pantalla de juego.

De forma similar a lo que sucede en el servidor, el cliente posee una clase Game propia donde realizará el bucle principal del juego y gestionará los mensajes con el método **net_thread**:

- **CLIENTJOINED:** En el caso de recibir este mensaje, el servidor lo ignorara, ya que solo el servidor puede enviar este mensaje.
- **CLIENTLEFT:** En el caso de recibir este mensaje, el servidor lo ignorara, ya que solo el servidor puede enviar este mensaje.
- **MOVEMENT:** En el caso de recibir este mensaje, el servidor lo ignorara, ya que solo el servidor puede enviar este mensaje.
- **NEWMAP:** El cliente recibirá este mapa, con el cual podrá cargar la pantalla de juego y el nivel en el que está jugando el servidor.
- **UPDATEPOS:** El cliente recibirá las posiciones actuales de los personajes, en la mayoría de los casos tras realizar una pulsación de teclado.
- **UPDATEEXITS:** El cliente recibirá la posición de las casillas de salida, renderizándolas junto con el mapa de la sala.

El cliente del juego utilizará un sistema llamado RenderSystem, para renderizar en pantalla el mapa de juego y los personajes, situando estos en las posiciones que recibe por mensaje. Por ello tendremos un método para dibujar a los héroes, otro para las salidas y otro para el mapa, actualizándose todos durante cada vuelta del bucle principal con el objetivo de prevenir la repetición de imágenes en pantalla.

El HeroSystem del cliente, a diferencia del que podemos encontrar en el servidor, tan solo estará encargado de enviar la dirección de la tecla pulsada, dejando que el servidor se haga cargo de los cálculos. También tendremos un método que, tras recibir el mensaje del servidor con los movimientos actualizados, lo traducirá para mostrarlo en pantalla.

De la misma forma, el cliente contará con un método encargado de recibir el mensaje de nuevo mapa del servidor, traduciendo la información recibida para poder renderizar la pantalla de juego actual.

4. Desarrollo del juego:

El proyecto que había diseñado en un inicio posee bastantes diferencias con el resultado final.

En primer lugar, el juego estaba diseñado con un modelo cliente-servidor*: dumb terminal*, con la esperanza de maximizar la sincronización entre ambos jugadores. Después de comenzar, se decidió implementar una versión con un cliente que funcionaría como servidor, ya que resultaba más sencillo de configurar con el código realizado durante la práctica 2 de Redes[5].

Originalmente íbamos a importar varios niveles diseñados a mano, los cuales leeríamos en el programa; pero al final optamos por crearlos de forma aleatoria, para dar una sensación de diversidad y desconocimiento de la salas que saldrán.

Otra de las diferencias que hay entre el diseño inicial en comparación con el final podemos encontrarla en el modo de juego. Al inicio iba a implementarse un modo contrarreloj de juego o un límite de movimientos por sala, pero se acabó abandonando por motivos de alcance.

Para nuestra comodidad, primero desarrollamos la versión funcional del juego para un solo jugador en Visual Studio[6] en Windows, ya que es un ámbito donde estamos más cómodos programando, partiendo de una plantilla de SDL. Después, importamos el proyecto a una máquina virtual, donde comenzamos a adaptar el código para poder utilizar clientes y servidores.

Durante el transcurso del proyecto tuvimos diferentes problemas que caben destacar, como por ejemplo el cambio de máquina de virtual por motivos de rendimiento e imposibilidad de compilar C++ 17, necesario para poder utilizar las librerías SDL. Otro problema surgió a la hora de detectar el JSON de los recursos en uno de los ordenadores, impidiendo trabajar en paralelo.

5. Makefile:

```
SRC_DIR:= server / client

COMMON_DIR := common

SRC_FILES := $(wildcard $(SRC_DIR)/*.cpp) $(wildcard $(SRC_DIR)/*/*.cpp)

COMMON_FILES:= $(wildcard $(COMMON_DIR)/*.cpp) $(wildcard
```

```
$(COMMON_DIR)/*/*.cpp)

OBJ_FILES:= $(patsubst $(SRC_DIR)/%.cpp,$(SRC_DIR)/%.o,$(SRC_FILES)) $(patsubst
$(COMMON_DIR)/%.cpp,$(COMMON_DIR)/%.o,$(COMMON_FILES))

EXEC_FILE := MagicMazeServer.out / MagicMazeClient.out

LDFLAGS := -L/usr/lib/x86_64-linux-gnu -lSDL2_image -lSDL2_mixer -lSDL2_ttf -lSDL2
-lpthread

CPPFLAGS := --std=c++17 -I/usr/include/SDL2

$(EXEC_FILE): $(OBJ_FILES)

        g++ -o $@ $^ $(LDFLAGS)

$(OBJ_DIR)/%.o: $(OBJ_DIR)/%.cpp

        g++ $(CPPFLAGS) -c -o $@ $<

clean:

        rm -f $(EXEC_FILE) $(OBJ_FILES)
```

En el proyecto podemos encontrar dos ejecutables diferentes, uno para el servidor y otro para el cliente. Las diferencias entre los dos son mínimas, pero tener dos ejecutables sirve para poder establecer mejor los elementos del proyecto que pueden utilizar cada jugador.

Estos dos ejecutables son creados gracias a un makefile, encargado de hacer *make* de ambos ejecutables cuando entra el comando por la terminal y de eliminarlos cuando recibe un *make clear*.

6. Posibles mejoras:

Una de las mejoras que podrían implementarse está en el procesado de mensajes, ya que tanto servidor como cliente leen los mensajes a la vez, habiendo por ello una debilidad en el código del programa. Esta mejora implicaría establecer métodos propios para leer los mensajes, sin necesidad de recurrir al mismo.

Otra posible mejora está en las mecánicas del juego, pudiendo implementarse una cuenta atrás en las salas. Esto además implicaría desarrollar un menú de pausa entre niveles, donde aparezca un mensaje de victoria o derrota y el tiempo empleado para finalizar el nivel.

Además, existe la posibilidad de diseñar modos de juegos para más clientes, pudiendo haber cuatro jugadores al mismo tiempo, controlando cada jugador una flecha de dirección.

Una mejora que podría tenerse en cuenta es la disminución de recursos necesarios para ejecutar el juego, habiendo numerosos scripts sin alterar que poseen tanto cliente como servidor.

7. Referencias:

1. Magic Maze (Juego de mesa)
2. Dominó (Juego de mesa)
3. 2048 (Videojuego)
4. Topología cliente-servidor (Apuntes de la asignatura – Tema 3.1)
5. Práctica 2 de Redes (Proyecto de la asignatura)
6. Visual Studio (Programa)