Documentación Tarea 2 Programada 1.

Lenguajes compiladores e intérpretes.

SpaCEInvaders

Paradigma Imperativo y Orientado a objetos

Profesor:

Marco Rivera Meneses.

Integrantes:

Carlos Andrés Mata Calderón

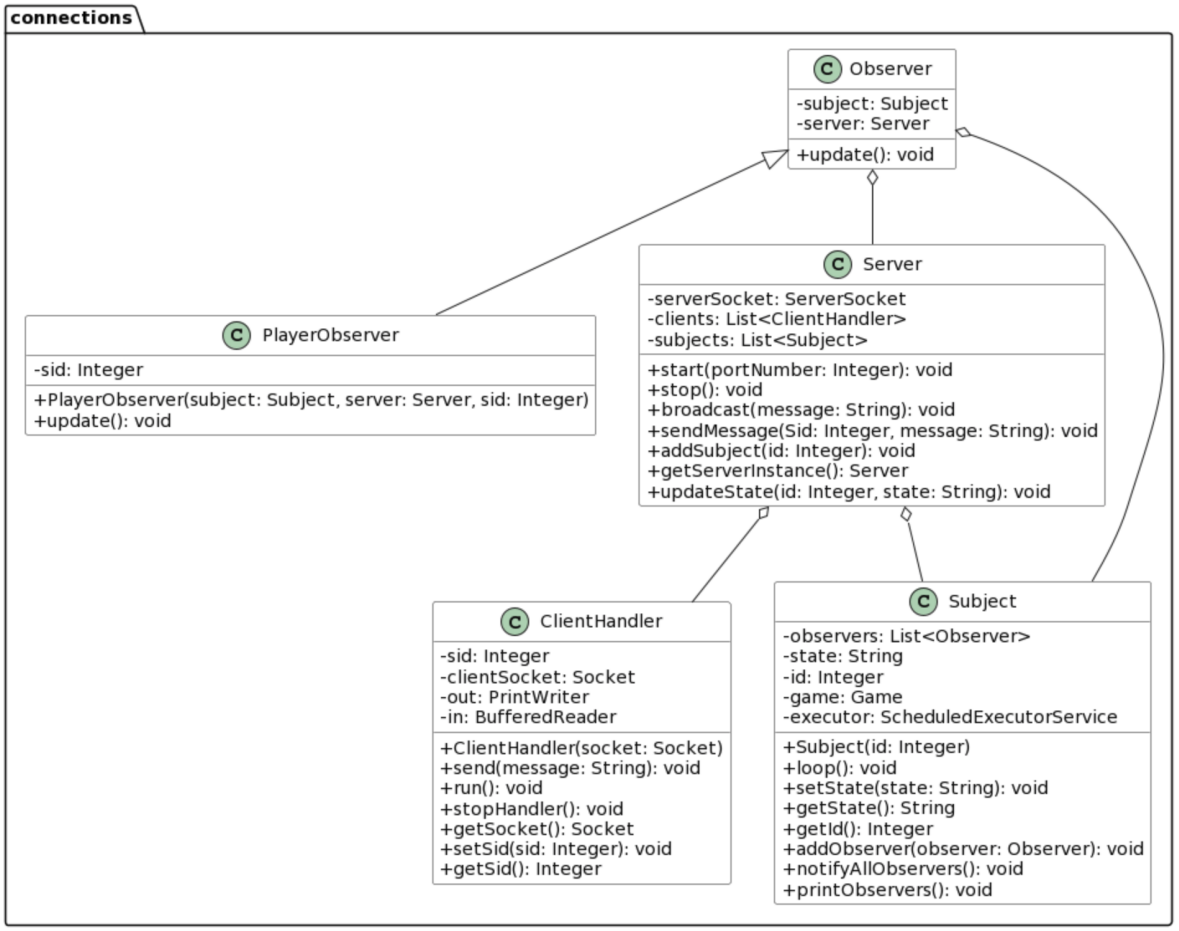
David Robles

Victor Cruz Jiménez.

Grupo 6.

Diagramas de Clases y Patrones de Diseño

Patrón Observer



El patrón Observer para implementar una comunicación entre los objetos Subject y Observer en un servidor de juegos. El patrón Observer permite a un objeto (Observer) ser notificado automáticamente cuando otro objeto (Subject) cambia su estado.

A continuación, se explica cómo se utiliza el patrón Observer en este código:

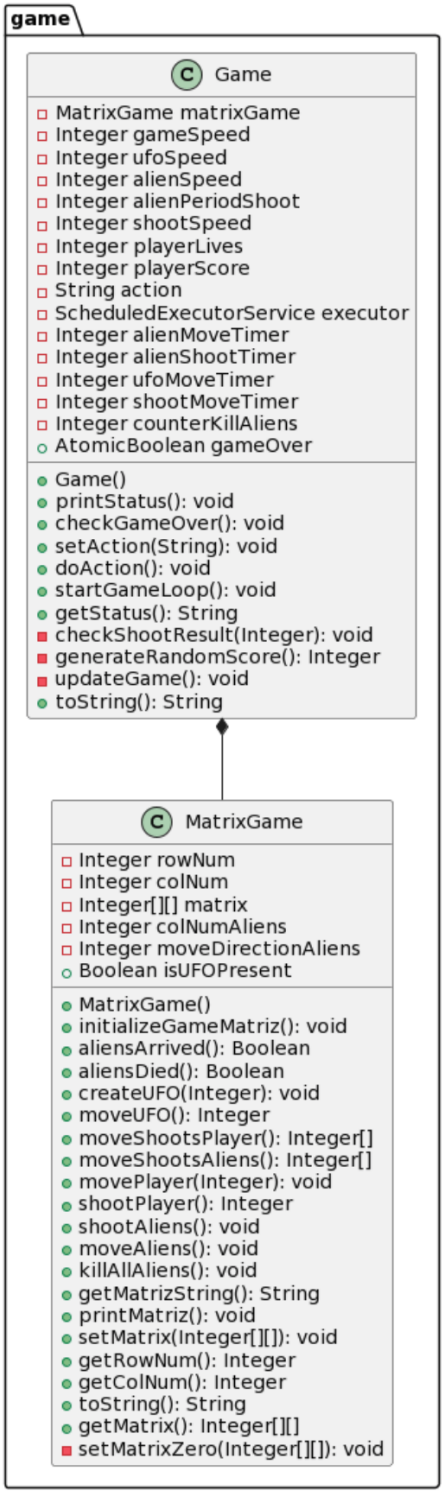
1. Clase Observer: La clase abstracta Observer define el método update() que será implementado por las subclases concretas de Observer. En este caso, la subclase PlayerObserver es un tipo concreto de Observer.
2. Clase PlayerObserver: Esta clase es una subclase concreta de Observer y define cómo debe ser notificada cuando el Subject al que está suscrita cambia de estado. En este caso, la clase PlayerObserver se actualiza enviando un mensaje al cliente asociado mediante el método update().
3. Clase Subject: La clase Subject es responsable de mantener una lista de objetos Observer (en este caso, objetos PlayerObserver) y notificarlos cuando su estado cambie. En el código proporcionado, el método setState() se utiliza para cambiar el

estado del Subject y, a continuación, se llama a notifyAllObservers() para notificar a todos los Observers suscritos. El método addObserver() se utiliza para agregar un nuevo Observer a la lista de Observers suscritos.

4. Clase Server: La clase Server es responsable de aceptar conexiones de clientes, crear objetos ClientHandler y mantener una lista de objetos Subject. En este código, los objetos ClientHandler pueden representar jugadores o espectadores. Cada vez que un jugador se conecta al servidor, se crea un nuevo objeto Subject, y el jugador es suscrito como Observer al Subject recién creado. Cuando un espectador se conecta al servidor, se suscribe como Observer al Subject asociado con el jugador que desea observar.

Cuando un objeto Subject cambia su estado (por ejemplo, cuando el estado del juego cambia), el método notifyAllObservers() se llama para notificar a todos los Observers suscritos al Subject. En respuesta, cada Observer llama a su método update() y realiza la acción correspondiente (en este caso, enviar un mensaje al cliente asociado con el estado actualizado del Subject).

Patrón Estructural Model-View-Controller



El patrón Model-View-Controller (MVC) es un patrón de diseño que separa la lógica de la aplicación en tres componentes interconectados: el Modelo, la Vista y el Controlador. Esto permite una mejor organización del código y facilita la escalabilidad y mantenimiento de la aplicación. A continuación, se explica cómo aplicar el patrón MVC en el servidor con las clases MatrixGame, Cliente y Game.

1. Modelo (MatrixGame): La clase MatrixGame representa el Modelo en el patrón MVC. En este caso, se encarga de mantener y actualizar el estado del juego. Esto incluye la representación de la matriz del juego, la posición de los elementos en el juego (jugador, enemigos, disparos, etc.), así como las reglas de juego y la lógica de interacción entre estos elementos. El Modelo no tiene conocimiento directo de la Vista o del Controlador, pero puede notificar cambios en su estado a través de eventos o llamadas de retorno.
2. Vista (Cliente en C): La Vista es responsable de mostrar el estado actual del Modelo al usuario. En este caso, la Vista sería un cliente en C que se comunica con el servidor para recibir información sobre el estado del juego y mostrarla al jugador. La Vista podría recibir datos del Modelo a través del Controlador y actualizar la interfaz gráfica en consecuencia. La Vista también captura las acciones del usuario (por ejemplo, movimientos o disparos del jugador) y las envía al Controlador para su procesamiento.
3. Controlador (Game): La clase Game actúa como el Controlador en el patrón MVC. Su función principal es gestionar la interacción entre el Modelo y la Vista. El Controlador recibe eventos o acciones del usuario desde la Vista (por ejemplo, movimientos o disparos del jugador) y actualiza el Modelo de acuerdo con estas acciones. Luego, puede actualizar la Vista para reflejar los cambios en el Modelo. El Controlador también puede ser responsable de iniciar y detener el bucle de juego, así como de gestionar el estado general del juego (por ejemplo, verificar si el juego ha terminado o actualizar las puntuaciones).

Descripción de algoritmos utilizados.

Para llevar a cabo esta sección, se dividirá en dos secciones, una centrada en el servidor en Java y en sus algoritmos relacionados al correcto funcionamiento del programa, y otra centrada en el cliente en C y su funcionamiento.

En Java/Servidor

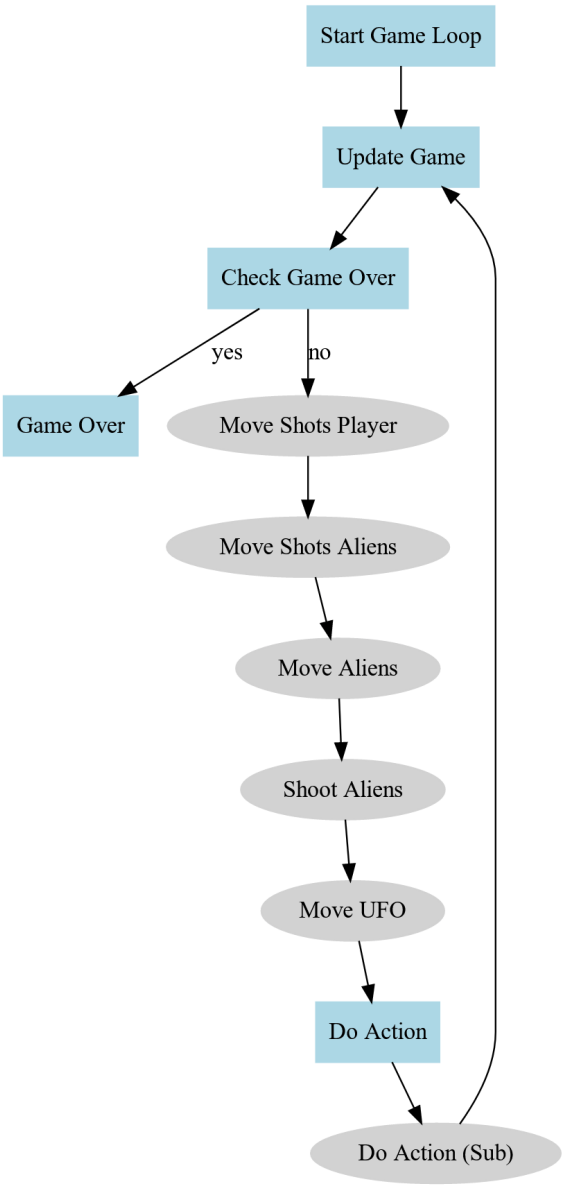
En la sección del Servidor, se explicará primero el funcionamiento de los Sockets y que tipo de datos son enviados a través de estos.

La conexión del servidor se lleva a cabo en el puerto 25565, este estará a la espera hasta que se conecten los clientes, una vez se lleva a cabo la conexión de los clientes, el servidor comenzará con la generación de los elementos necesarios para el correcto funcionamiento del juego, el principal algoritmo para el funcionamiento del juego se llama **startGameLoop.**

startGameLoop es un algoritmo en el servidor Java que se encarga de iniciar la creación e inicio de las funciones relacionadas con el correcto funcionamiento del juego. Este algoritmo maneja el movimiento del jugador a través de la función **doAction**, el movimiento de los enemigos en pantalla y la comprobación del estado del juego, incluyendo elementos como

la puntuación, tiempo de juego, vidas y enemigos restantes en pantalla. Las funciones **updateGame** y **checkShootResult** son las principales encargadas de llevar a cabo estas acciones.

La función **doAction** verifica las entradas en el teclado del jugador y realiza una acción específica para cada tecla pulsada. Por otro lado, **updateGame** actualiza el estado del juego, la cantidad de enemigos en pantalla y realiza el movimiento lateral de los enemigos, comprobando los márgenes de la pantalla para realizar un cambio de dirección cuando llegan al borde. Finalmente, **checkShootResult** verifica si la bala disparada por el jugador impactó a un enemigo, utilizando un switch que cambia de valor dependiendo del enemigo impactado y dando una cantidad de puntos en específico según el enemigo.



Game Class Flowchart

En C/Cliente.

En C, el cliente se conecta al mismo puerto que el servidor, y una vez establecida la conexión, el servidor comienza a enviar datos al cliente. La representación gráfica de estos datos se lleva a cabo mediante la interpretación de cadenas de caracteres.

Para procesar estas cadenas de caracteres, se utiliza la biblioteca SDL de C, que incluye una función llamada **renderChar**. Esta función se encarga de leer las cadenas de caracteres enviadas por el servidor y luego llama a la función **showPicture** para cargar las imágenes correspondientes en la interfaz gráfica del cliente. De esta manera, el cliente puede recibir y mostrar información enviada por el servidor en tiempo real.

Descripción de estructuras utilizadas.

Para iniciar esta sección, se va a hablar por separado de cada lenguaje, es decir, se explicará primero lo relacionado a Java y después lo relacionado a C.

En Java:

La principal estructura utilizada en Java fue una matriz de datos.

En dicha matriz llamada “matrix” se crean los elementos del juego, tanto jugador, los “aliens” que van a ser los enemigos del juego, enemigos aleatorios y todo lo relacionado con el mismo, elementos tales como puntuación, vidas, etc.

En C.

Dicha matriz tiene que ser enviada a través de sockets al cliente, convirtiéndose en una cadena de chars, para que una vez enviada en su totalidad al cliente, una vez recibida la lista en su totalidad, esta se aloje en una lista de chars, en formato CSV, en la memoria del cliente, de modo que este sea capaz de comprenderla y cargarla en una interfaz gráfica para que sea llevado a cabo el juego.

Problemas conocidos con solución.

Uno de los principales problemas que se encontraron a la hora de realizar este trabajo, fue el hecho que de se tenía que realizar la conexión por medio de sockets, lo que dificulto hasta cierto punto el avance en el mismo, ya que ocurrían errores a la hora de intentar enviar la información por medio de sockets, pero este error se llegó a solucionar de forma efectiva.

Problemas sin solución.

En cuanto al servidor, cuando se estaban haciendo las pruebas iniciales de sockets, era posible tener un “jugador” y un “espectador” a la vez, pero una vez el código avanzó en complejidad, el espectador dejó de recibir las actualizaciones con los cambios en el juego.

En el cliente, parece que C no siempre interpreta los mensajes recibidos de la forma esperada, por lo que la interfaz puede mostrar posiciones erróneas para algunas naves u objetos que no estaban allí, aunque suelen desaparecer al recibir la próxima actualización.

Plan de actividades.

Estudiante a cargo:	**Tarea asignada:**	**Descripción de tarea**	**Tiempo estimado:**	**Fecha de entrega:**
Carlos Andrés Mata Calderón	lógica Servidor	Encargado de la creación del servidor	2 Horas	14/4/2023
		Encargado de llevar a cabo lo relacionado con la lógica del juego.	8 horas	19/4/2023
David Robles	lógica Cliente	Encargado del correcto funcionamiento del cliente	8 horas	19/4/2023
	Creación del Exe	Encargado de llevar a cabo la creación del archivo. Exe para la ejecución del programa	1 hora	20/4/2023
Victor Cruz Jiménez	lógica Cliente	Encargado de la conexión del cliente.	2 horas	14/4/2023
	Administración	Encargado de llevar a cabo la documentación	3 horas	20/4/2023

		necesaria del proyecto.		

Conclusiones.

- El proyecto se benefició enormemente del uso de los paradigmas de Programación Orientada a Objetos (POO) e Imperativa, ya que permitieron estructurar el código de manera más organizada y modular. Gracias a la implementación de patrones de diseño como Observer y Model-View-Controller (MVC), se pudo mantener una clara separación de responsabilidades entre las clases y sus interacciones. Esto resultó en un proyecto más fácil de entender, mantener y escalar.

- La implementación de estos paradigmas y patrones de diseño en otros proyectos puede mejorar la calidad del código y facilitar la colaboración entre equipos de desarrollo. Al utilizar enfoques comprobados como Observer y MVC, los desarrolladores pueden centrarse en la lógica de negocio específica del proyecto, sabiendo que la arquitectura general es sólida y escalable.

- Conocer los paradigmas de POO e Imperativa y sus patrones asociados es esencial para cualquier desarrollador de software moderno. No sólo permiten construir aplicaciones más eficientes y escalables, sino que también facilitan la comunicación entre desarrolladores y la adopción de mejores prácticas de desarrollo en toda la industria.

- En este proyecto, se enfrentaron y superaron varios desafíos técnicos, como la comunicación entre el servidor y el cliente a través de sockets y la interpretación de mensajes en tiempo real. A pesar de algunos problemas conocidos sin solución, el proyecto demuestra el poder y la flexibilidad de los paradigmas de POO e Imperativa y la importancia de utilizar patrones de diseño adecuados para gestionar la complejidad del código en aplicaciones distribuidas.

Recomendaciones.

- Utilizar patrones de diseño y arquitecturas comprobadas: Para futuros proyectos, se recomienda investigar y adoptar patrones de diseño y arquitecturas probadas, como

Observer y MVC, desde el inicio del desarrollo. Al hacerlo, los equipos de desarrollo pueden centrarse en la lógica específica del negocio y asegurarse de que el proyecto sea más fácil de mantener, escalar y entender.

- Priorizar la comunicación y documentación: Es fundamental que los desarrolladores se comuniquen de manera efectiva entre sí y documenten adecuadamente sus decisiones y enfoques. Esto facilitará la comprensión del código por parte de otros miembros del equipo, así como la identificación y solución de problemas. La documentación debe incluir información sobre las clases, métodos, algoritmos y estructuras de datos utilizadas, así como detalles sobre problemas conocidos y posibles soluciones.
- Realizar pruebas y revisiones de código continuas: Para garantizar la calidad y estabilidad del software, es crucial llevar a cabo pruebas y revisiones de código de forma regular durante todo el ciclo de desarrollo. Esto ayudará a identificar y solucionar problemas antes de que se conviertan en desafíos mayores, mejorando la eficiencia y la calidad del producto final. Además, alentar a los desarrolladores a revisar el código de sus compañeros de equipo puede fomentar el aprendizaje y la colaboración, lo que a su vez puede resultar en un código más robusto y eficiente.

Referencias.

Cabanes, N. (s.f.). *Curso C*. Obtenido de Juegos Multiplataforma: SDL: https://www.nachocabanes.com/c/curso/cc10e.php

González, J. D. (s.f.). *Usando sockets en Java. Una simple aplicación cliente servidor usando sockets*. Obtenido de https://www.programarya.com/Cursos-Avanzados/Java/Sockets

Wikibooks. (11 de 09 de 2021). *Programación en C/Sockets*. Obtenido de https://es.wikibooks.org/wiki/Programación_en_C/Sockets

Bitácora.

Carlos:

```
❯ git log --graph
* commit e55ed40d3af4d95328ac476f66367b31311918fe (HEAD -> add-last-features)
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 23:49:25 2023 -0600
|
|     docs: add comments
|
* commit adab5f3c204c426df9f00fc7240a70c27301af99
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 23:49:09 2023 -0600
|
|     docs: add comments
|
* commit 3a4e2295df412cd9e77bba864c83479f6b8f6a3e (origin/add-last-features)
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 23:44:13 2023 -0600
|
|     chore: two players
|
* commit df6267f36a2946c168c5305e2df57f3fc704a440
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 23:22:34 2023 -0600
|
|     feat: add events keyboard
|
* commit 61080bea371466483c30fb61136f173e7cc85f35
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 23:08:18 2023 -0600
|
|     docs: add comments
|
* commit 750cda00728856a40c8f9f87185dda71fcb987c7
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 23:00:32 2023 -0600
|
|     docs: add comments
|
```

```
* commit 3ff4dae82a90cfcc4f387fffced92e6e423f8e74 (origin/unified, origin/master, unified, master)
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 22:34:12 2023 -0600
|
|     feat: show the game in gui perfect
|
* commit 20575b90521188c4f6dfdb3033199768c00654d8
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 22:33:12 2023 -0600
|
|     feat: show the game in gui perfect
|
* commit 4adb891da100fd630e59ebe80ce0b390a98a9524
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 19:22:21 2023 -0600
|
|     feat: show the game in console client
|
* commit 36e045c895471cd02ad92d80b79e9fc75206aa65
| Author: Carlos12001 <carlos.andres12001@gmail.com>
| Date:   Thu Apr 20 18:55:32 2023 -0600
|
|     feat: show the game
|
*   commit 68560a3b72d05af4ad9148968eae31ae8acc664e
|\  Merge: 6891d10 fc27445
| | Author: Carlos12001 <carlos.andres12001@gmail.com>
| | Date:   Thu Apr 20 17:27:19 2023 -0600
| |
| |     Merge branch 'sockets' into unified
| |
| |     # Conflicts:
| |     #       server/src/Main.java
```

David

```
| * commit fc2744501c97dbf7aad30d91e95474ff3ec8ad2c (sockets)
| | Author: David Robles <47421546+diabloget@users.noreply.github.com>
| | Date:   Thu Apr 20 16:16:18 2023 -0600
| |
| |     fix: \n message
| |
| |     \n fixed my whole life :D
| |     PD: I hate winsock
| |
| * commit 71d50a10ca076a47f2b4556cf1254275cdf70249
| | Author: David Robles <47421546+diabloget@users.noreply.github.com>
| | Date:   Thu Apr 20 15:38:43 2023 -0600
| |
| |     update: socket int
| |
| |     Int Socket is now a unique variable located in the csocket.c file, there is no multiple declarations any more :D
| |
| * commit 77a66604ef9251a7c8c9b937030e3c7c42dafb29
| | Author: David Robles <47421546+diabloget@users.noreply.github.com>
| | Date:   Thu Apr 20 14:09:34 2023 -0600
| |
| |     Fix: Cmakelist correction
| |
| * commit 8766368125c4b1b349ac4abaccd737015f63cb3c
| | Author: David Robles <47421546+diabloget@users.noreply.github.com>
| | Date:   Thu Apr 20 07:07:04 2023 -0600
| |
| |     fix: Minor change in colour
| |
| |     Background colour changed for a better black/gray
| |
| * commit d61bcb8d42b10b3adba5758fe090eb4c11835e83
| | Author: David Robles <47421546+diabloget@users.noreply.github.com>
| | Date:   Thu Apr 20 07:01:50 2023 -0600
| |
| |     fix: Memory leak fixed
| |
| |     There was a huge memory leak D:
| |
| * commit c87022bee473584641da54c713a7e6f1ec54f846
| | Author: David Robles <47421546+diabloget@users.noreply.github.com>
| | Date:   Thu Apr 20 05:53:05 2023 -0600
| |
| |     Feat: image load
| |
| |     Now the code is able to load icons and show them in the gui :D
| |
```

```
| * commit b70f0838c42a4c971bf7ef987aa83a58e30d4fb6
| | Author: David Robles <47421546+diabloget@users.noreply.github.com>
| | Date:   Mon Apr 17 11:46:58 2023 -0600
| |
| |     fix: corrected index
| |
| |     Fixed client list index to match the programming index.
| |
| * commit 03b96adfdcc9daaf78d5bdd9e577306ae80f0e47
| | Author: David Robles <47421546+diabloget@users.noreply.github.com>
| | Date:   Mon Apr 17 11:43:38 2023 -0600
| |
| |     feat: add multiple client functionallity
| |
| |     Now it works with 2 players, or 1 player + 1 spectator.
| |
```