**Memoria del Proyecto CanarIO**

**Breve Descripción**

El juego CanarIO consiste en mover tu círculo por el mapa a fin de ganar el mayor tamaño posible. Cuando el tamaño del jugador aumenta de 100, este gana la partida. Para aumentar tu tamaño se deben “ingerir” los alimentos, pequeñas bolitas verdes esparcidas por el mapa. Así como a otros jugadores de menor tamaño. La partida se pierde si otro jugador te come.

1. **Arquitectura del Juego**
   1. **Modelo del juego**

El juego sigue un modelo cliente-servidor. El servidor contiene el estado actual del juego y guarda las conexiones con los clientes a los que les manda a renderizar cada vez que el estado del juego varíe. Por otro lado, estos clientes notifican al servidor cualquier cambio que hayan realizado para que actualice el estado del juego en consecuencia.

* 1. **Estado del juego, objetos y replicación**

Todos los elementos del juego, a excepción de los textos, a nivel lógico son Players. Un Player es la entidad básica del juego, contiene los datos necesarios para situar y renderizar correctamente círculos.

Al iniciarse el juego, el servidor genera aleatoriamente una determinada cantidad de alimentos que esparce por el mapa. Estos alimentos no son mas que Players que contienen únicamente una posición, tamaño y color. Además, no mueren, sino que cuando son consumidas cambian su posición a una nueva también aleatoria. Estos alimentos son guardados en un vector diferenciado dentro del servidor para tratarlos de manera independiente posteriormente.

Por otro lado, cuando el servidor recibe una petición de LOGIN por parte de un nuevo cliente, este crea un nuevo Player y le asigna el Socket correspondiente. Luego, mete el nuevo jugador en un vector de jugadores y lo instancia en el escenario en una posición aleatoria.

El servidor escucha constantemente a los jugadores. Cuando recibe un mensaje de alguno de ellos, lo procesa, actualiza el estado del juego y les envía el nuevo estado a todos los jugadores para que estos rendericen de nuevo. Este envío consiste en un mensaje por cada Player del juego (Jugadores y comida) que contiene la información necesaria para que el cliente lo dibuje en su pantalla.

**1.3 Protocolo de aplicación y serialización**

El servidor usa un protocolo de aplicación simple basado en el intercambio de mensajes con la siguiente estructura:

|  |
| --- |
| TIPO (1 byte, tipo uint8\_t). 0: LOGIN, 1:MOVE, 2:LOGOUT 3:DRAWPLAYER 4:RENDERCALL 5:WIN 6:LOSE |
| Estos dos apartados son el contenido de todos los mensajes |
| NICK (8 bytes, tipo char[8]). Contiene el nombre del jugador |
| MENSAGE (200 bytes, tipo char[200]). Contiene hasta 200 bytes de información |

Los mensajes se serializan en texto plano. En muchos de los mensajes solo interesa su tipo para que el receptor realice unas u otras acciones predeterminadas por lo que el apartado de mensaje se envía vacío.

La información requerida en los mensajes es serializada en texto, posteriormente se envía y el receptor se encarga de deserializar y parsear a estructuras de datos la información recibida en forma de texto.

|  |
| --- |
| Mensajes con el apartado mensaje vacío donde sólo importa el tipo |
| LOGOUT / RENDERCALL / WIN / LOSE |

|  |
| --- |
| Otros mensajes |
| DRAWPLAYER: Nick + msg (pos.x – pos.y – size - color) |
| MOVE: Nick + msg (key pressed) |

1. **Diseño del Servidor**

El servidor es la clase que se encarga de gestionar el estado del juego, así como de realizar comunicaciones con los clientes. Este está dotado de su propio display y realiza un ciclo completo cada vez que un cliente notifica un cambio de estado. (LOGIN, LOGOUT, MOVE). Mientras no haya modificaciones en el estado, el servidor permanece a la espera.

El servidor contiene dos vectores: El vector de jugadores y el vector de alimentos. Al ejecutarse por primera vez se inicia la ventana, se instancian los alimentos y se realiza el primer renderizado.

A partir de este momento el servidor entra en el bucle principal hasta que se cierra desde la consola. En este bucle el servidor espera mensajes de los clientes. Estos mensajes pueden ser de 3 tipos:

* **Login**: El servidor crea y asigna un nuevo Player al cliente.
* **Logout**: El servidor elimina todas las referencias a ese cliente y lo elimina del estado del juego.
* **Move**: El servidor mueve el jugador del cliente en función de la tecla que haya pulsado.

A continuación, calcula las posibles colisiones que se hayan podido producir a causa de este cambio en el estado. Se modifican consecuentemente los tamaños de los jugadores victoriosos en alguna colisión y se guardan en un vector de muertos a los que hayan perdido para su posterior tratamiento. Luego, se renderizan los jugadores y alimentos (en el servidor) y se les envía un mensaje de GAMEOVER a los jugadores en el vector de muertos.

Por último, se manda a todos los clientes la información necesaria para que rendericen el estado actual en sus pantallas.

**3. Diseño del Cliente**

El cliente se encarga de gestionar el input y renderizar lo que verá el jugador. Posee 2 hilos distintos, uno para mandar input al servidor y otro para recibir mensajes.

Al crear el cliente este hace una petición de LOGIN el cual avisa al servidor de la llegada de un nuevo jugador

El hilo que se encarga del input posee input\_thread(). Usa waitkey() para saber que tecla se ha pulsado y si esta no es la “q”, pues esta es la tecla de salida, le manda la tecla en el cuerpo del mensaje al servidor. Además, esta rama se encarga de comprobar si el jugador sigue vivo.

Esto se debe a que, a nivel de código, el jugador sólo “muere” cuando pulsa “q”. Su bucle se detiene cuando este muere y lanza logout() que se encarga de cerrar el programa del cliente y avisar al servidor de que el jugador ya no está conectado.

El hilo que se encarga de recibir mensajes también realiza el renderizado de la pantalla del cliente. Este posee un bucle infinito donde, primero, espera el mensaje; y según cual sea, puede realizar:

* DRAWPLAYER: Parsea la información recibida por el servidor con el método parseDraw(Vector2& position, uint16t& size, XLDisplay::XLColor& color\_, std::string message) y, con esta, dibuja tanto jugadores como alimentos.
* RENDERCALL: Bucle de renderizado. Flush() y clear() del display.
* WIN: El jugador gana, bloqueando su recepción de mensajes y renderizando la pantalla de victoria.
* LOSE: El jugador gana, bloqueando su recepción de mensajes y renderizando la pantalla de derrota.

Todas estas opciones implican un posterior proceso de renderizado y por eso se hacen en el mismo hilo.

El cliente no almacena casi información ya que de esto se encarga el Player que le corresponde y que se encuentra almacenado dentro del servidor. Por ello, el cliente lo solo posee un string con su nombre, un socket con la dirección del servidor, su display y un booleano que le permita saber de forma global si sigue vivo.

**4. Conclusiones**

Estuvimos mucho tiempo estos últimos días intentando implementar cosas que no conseguimos completar. La más destacable es el movimiento suave. En el formato en el que se encuentra actualmente, wait\_key() devuelve para la pulsación de una tecla un constante flujo de eventos keyDown keyUp alternativamente. Esto permite al cliente enviarle un mensaje de actualización de posición en cada keyDown. No obstante, esto limita mucho el movimiento, solo permitiendo que este exista en un eje a la vez y con una sensación de retardo al comenzar la pulsación.

Para mejorar este movimiento intentamos detectar los eventos keyUp y usarlos para, en un array de 4 booleanos, registrar el estado de las teclas WASD. El problema es que el servidor solo actualiza la posición de un cliente cuando recibe de este un mensaje de movimiento al que posteriormente le devuelve el nuevo estado del juego para que lo renderice. Esto implicaba que, aun manteniendo la tecla pulsada, el estado del juego solo se actualizaba en la primera pulsación.

Intentamos crear un thread paralelo en el cliente que mandara constantemente (60 veces por segundo) un mensaje MOVE al servidor con el estado de las teclas para que moviera al jugador. No obstante, el nuevo estado del juego derivado de esta modificación nunca llegaba de vuelta al cliente pues el thread que realizaba el MOVE no era capaz de recibir mensajes.

En un último esfuerzo por conseguir el suave y preciado movimiento, intentamos hacer el receive no bloqueante añadiéndole un timeout. Sin embargo, el receive nunca recibía mensajes y siempre hacía timeout.

También queríamos una cámara distinta para el jugador, donde todo se viera mucho más cerca para que este no viera el mapa completo.

Esto funcionaría de tal forma que todos los tamaños y posiciones mandadas por el servidor estarían escalados. Cada cliente tendría un proceso propio donde adecuaría todas las posiciones para situar su Player en el centro de la ventana. De esta manera se ignoraría el dibujado de todo aquello que se saliera de su pantalla.

Con esto podíamos abstraer el mapa y por lo tanto hacerlo mucho más grande, ya que al poder regular como queramos lo que ven los jugadores, los bordes actuales podrían no existir pues ninguno vería el mapa completo.

**Referencias**

[*https://agar.io/*](https://agar.io/)(Fácilmente confundible con el nuestro 😉)