ITESO

Universidad Jesuita
de Guadalajara

PROYECTO FINAL

CONECTA “N”

Programación con Memoria Dinámica

Profesor: José Antonio Camarena Covarrubias

Equipo:

- **Kury Josafath Vázquez Correa ii727571**
- **Jorge Javier Guerrero Herrera si721790**

Fecha de entrega: 07/12/2021

Documentación del código del juego “ConectaN”

ConectaN.h :

Contiene las funciones y typedef que serán útiles para crear un tablero con fichas.

ConectaN.c :

El archivo .c tiene todas las funciones y estructuras necesarias para crear un tablero con fichas, ya que tiene las estructuras Movimiento (con información de coordenadas y jugador al que pertenece cada ficha) y Juego (con la información de cuántos movimientos hay en nuestro juego, cual es el valor de N, un arreglo que guarda todas los movimientos).
Las estructuras que crean una nueva estructura vacía no serán explicadas, sin embargo sí se explicará lo que hacen las funciones de nuevo movimiento y la lógica con la que trabajan las funciones que revisan si hay un ganador en el juego.

- nuevoMovimiento: básicamente es una función que crea un Movimiento nuevo vacío, después lo llena con la información recibida de coordenadas y jugador al que pertenece y por último agrega este movimiento al arreglo de movimientos de nuestra estructura intermedia de tipo Juego.
  En esta función además se llaman las funciones de buscar ganador para que si se corre desde un IDE o consola, se puedan ir monitoreando los resultados de dichas funciones.

- funciones que buscan ganador: lo que hacemos en esta función es asumir que es probable que la ficha que estamos metiendo es la que nos hace ganar, pero que probablemente no es la de un extremo de la línea de N fichas que necesitamos para ganar. Por este motivo lo que hacen todas las funciones de buscar ganador es primero buscar los límites de esta posible línea ganadora.
  Una vez que tenemos los límites mayores y menores de nuestras posibles líneas ganadoras hacemos una resta, y si la diferencia que hay entre las fichas de los dos extremos de la línea es mayor o igual a N-1 esto quiere decir que hay un ganador y el ganador es el dueño de esa última ficha ingresada.

Al final hay algunas funciones sencillas que se utilizan en el main ya que no se tiene acceso a las estructuras cómo tal desde ahí.

ConectaNgrafic.c :

Contiene la implementación de las funciones relacionadas a la inicialización y dibujo de la interfaz gráfica.
Contiene las funciones:

- DibujarTablero: mediante ciclos for y verificación de las coordenadas, se divide una espacio de trabajo para representar las casillas.
- InitGame: inicializa los valores de los structs FICHA y TURNO, qué representan los círculos qué van en el tablero, y el qué muestra el turno del jugador según el color, respectivamente.
- Update: cada qué el botón de alguna de las columnas es presionado, se actualiza la información de las structs FICHA y TURNO.
- DrawGame: se escribe todo el texto deseado, se llama a la función DibujarTablero, y se dibujan los botones bajo una condición para llamar a la función Update y actualizar los datos, para terminar ilustrando la figura qué representa el cambio de turno.

Para este archivo .c se incluyen el ConectaN.h qué contiene todas las funciones definidas necesarias, raylib.h para el manejo de figuras y texto, y raygui.h para el manejo de botones.

main.c

Contiene la función main, con el loop principal qué hace funcionar el programa junto con su representación gráfica. Se incluyen las librerías conectaN.h, qué encapsula todas las funciones utilizadas para el programa, raylib.h para el manejro de interfaz gráfica, la ventana emergente, etc., y raygui.h para el manejo qué se hace se botones cuando se dibuja el Juego.
Se inicializa la ventana, se establecen los FPS para correrlo, y en el loop principal “mientras la ventana no se cierre”, se llama únicamente a la función DrawGame.

raygui.h

Contiene el código de este módulo auxiliar a raylib.h para crear interfaces simples usando el estilo gráfico de raylib. Este código fue encontrado en la web para instalar dicho complemento de liberia y poder utilizar los botones.

CONCLUSIONES

Kury: para conseguir la funcionalidad de este proyecto nos enfrentamos a diversos retos. Se tomó la decisión de comenzar a programar sin el uso de la interfaz gráfica, conseguir el funcionamiento de la lógica del juego fue el primer paso, se hizo mediante arreglos dinámicos que iban registrando los movimientos de cada jugador y se iba analizando según columnas, renglones y diagonales para encontrar cuando se igualaran dichos movimientos consecutivos a la variable global CONECT_N. La idea fue evolucionando, desde simples ciclos de matrices que recorrían el juego

completo que usamos para darnos una idea más clara de las condiciones futuras que debíamos realizar, hasta la decisión de utilizar los arreglos antes mencionados. La interfaz gráfica fue lo que nos llevó más tiempo, conocer los comandos, las funciones y el comportamiento de las mismas para poder representar un tablero cambiante según los movimientos de los usuarios. Fue complicado tomar un camino para comenzar, ya que no era tan clara la manera de representar una matriz, sin poder acceder directamente a un espacio previamente definido. Por lo qué se decidió dividir un rectángulo según sus dimensiones, usando las columnas y filas seleccionadas,

```
void DibujarTablero(void) {

    DrawRectangleV( position: (Vector2) { .x: 100.2f, .y: 150.2f}, size: (Vector2) { .x: 625.2f, .y: 400.2f}, LIGHTGRAY);

    for (int i = 1; i < ROW; i++) {
        DrawLineV( startPos: (Vector2) { .x: 100.2f, .y: 150.2f + ((400.2f / ROW) * i)}, endPos: (Vector2) { .x: 725.2f, .y: 150.2f + ((400.2f / ROW) * i)}, BLACK
    }
    for (int i = 1; i < COL; i++) {
        DrawLineV( startPos: (Vector2) { .x: 100.2f + ((625.2f / COL) * i), .y: 150}, endPos: (Vector2) { .x: 100.2f + ((625.2f / COL) * i), .y: 550.2f}, BLACK);
    }
```

para posteriormente instalar botones arriba de cada columna que nos arrojaban el número para saber colocar la siguiente ficha, y con el uso de un arreglo con espacio para el número de columnas, se iba aumentando la frecuencia de clicks en los botones para conocer cuántas fichas ya se encontraban en la columna y obtener la fila válida para dejar caer la siguiente.

```
for (int j = 0; j < COL; j++) { //Imprime los botones y obtiene la columna al hacer click
    if (GuiButton( bounds: (Rectangle) { .x: 100.2f + ((625.2f / COL) * j), .y: 110, .width: (625.2f / COL), .height: (400.2f / ROW)}, text: "-") && (ficha.fi
        Update(j);
        nuevoMovimiento(prueba, ficha.color_int, ficha.filaValida[j], j);

    }
}
```

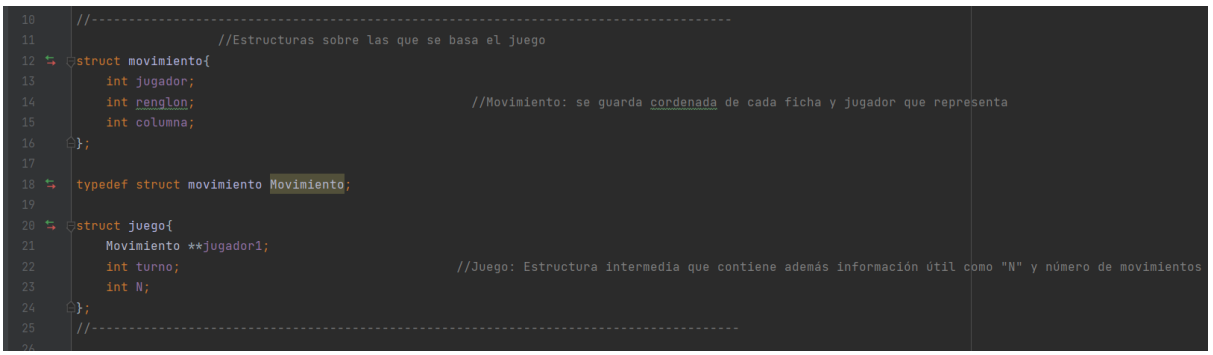
En esta parte se tiene un ciclo según las Columnas, se inicializan los botones que devuelve un bool si son presionados, y si es el caso, se actualiza la información de las estructuras para poder generar un movimiento.

Una vez conocidos ambos datos, ya se podía jugar con las coordenadas de las fichas que iban entrando. También había surgido la idea de recurrir a los cuadros de texto que indicará directamente la columna, pero era aún más complicado utilizar esa herramienta. Puedo concluir que los detalles de implementación imaginados para resolver cualquier tipo de código determinan el camino qué se seguirá durante el resto de la programación, y definen también el grado de dificultad que se desee manejar durante el trabajo, mientras mejor esté todo organizado, encapsulado, referenciado, mayor facilidad para conocer el status del código y la funcionalidad que va tomando. También destaco la importancia de tomar el tiempo y organización suficiente para trabajar día a día en proyectos de este tipo, mientras más tiempo, más detalles puedes agregar para mejorar el código. Al final logramos resolver la mayoría de las cosas que nos habíamos propuesto, un detalle qué nos faltó solucionar fue que en la interfaz fueran visibles tableros mucho mayores, aunque la

implementación si dirá con certeza cuando alguien conecta fichas muy grandes, en el tablero gráfico no se podrá observar de la mejor manera.

Jorge:

En general se presentaron varios retos al realizar este proyecto, sobre todo con la parte de utilizar la librería de reylib con la que apenas y habíamos tenido un pequeño contacto anteriormente.
Pero siendo más específicos y hablando de la implementación cómo tal creo que algunas partes importantes son las estructuras, desde que se comienza a pensar en un proyecto así es muy importante saber hacia dónde queremos ir y qué cosas nos facilitarán el trabajo para llegar allí, o lo que también es válido, empezar descartando las ideas que solamente te alejan o complican tus objetivos o simplemente prácticamente te imposibilitan el cumplirlos.
Hablando de este caso, la primer idea que se tuvo para implementar un conectaN fue en una matriz, así cómo se haría un contecta4 simple, una matriz en la que cada espacio del tablero fuera una posición en la matriz y se guardara un número entero según el estado de dicho espacio de tablero. Pero al darnos cuenta de que esta idea sería muy complicada cuando se trataran de valores variables de renglones, columnas y N, además de las restricciones propias del trabajo por ser un proyecto escolar tuvimos que empezar a pensar en cuáles serían las estructuras e implementaciones que se usarían para que fuera eficiente en procesamiento y utilizara memoria dinámica.
La primera idea fue utilizar árboles binarios de búsqueda que se comparan u ordenaran por el número de columna y almacenar en cada nodo stacks con movimientos. Lo cual pudo haber funcionado pero sin duda era mucho más difícil que la solución final, que fue solo hacer una estructura para guardar movimientos con toda la información necesaria para imprimirlos en el tablero y una estructura intermedia con un arreglo dinámico de movimientos.

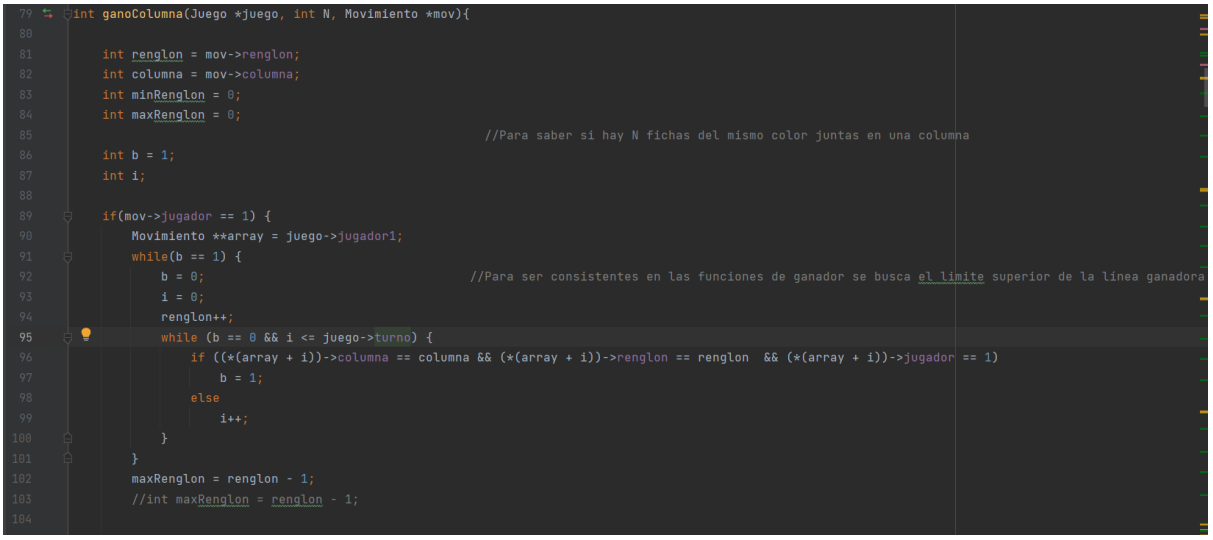


```
//-----------------------------------------------------------------------------
                //Estructuras sobre las que se basa el juego
struct movimiento{
    int jugador;
    int renglon;                                //Movimiento: se guarda cordenada de cada ficha y jugador que representa
    int columna;
};

typedef struct movimiento Movimiento;

struct juego{
    Movimiento **jugador1;
    int turno;                                  //Juego: Estructura intermedia que contiene además información útil como "N" y número de movimientos
    int N;
};
//-----------------------------------------------------------------------------
```

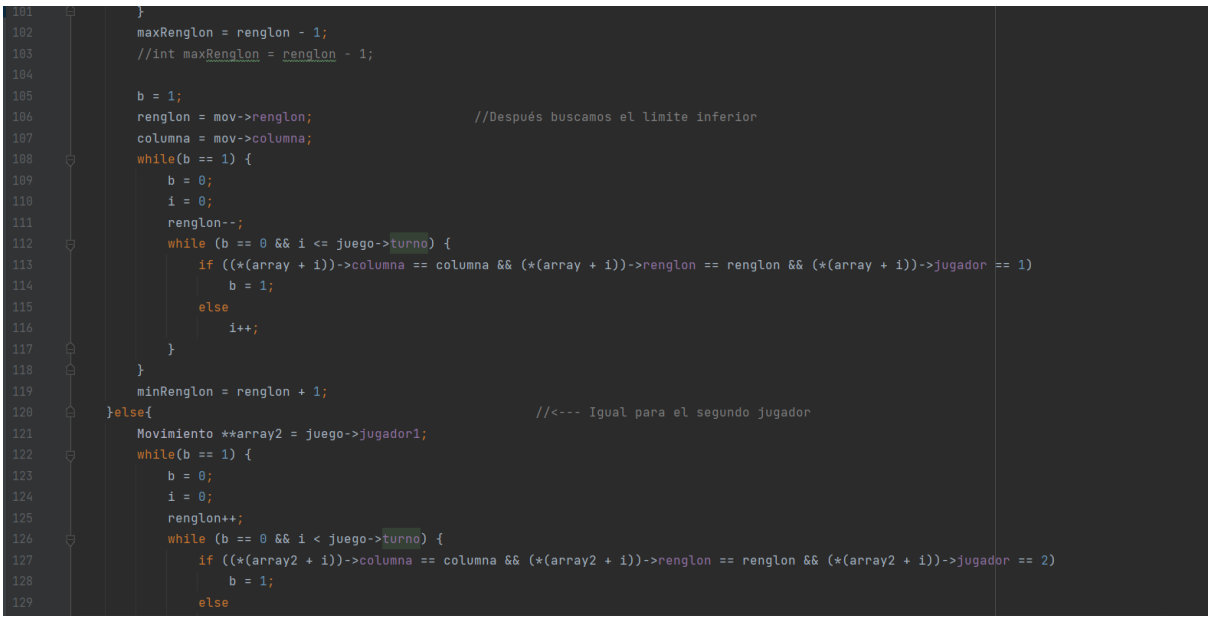
La primer idea de la estructura de Juego, era que tuviera un arreglo para las fichas de cada jugador, es por eso que bastante partes del código tienen una lógica un poco extraña para tratarse de solo un arreglo cómo las funciones que revisan si hay ganadores que se muestran a continuación:



```
int ganoColumna(Juego *juego, int N, Movimiento *mov){

    int renglon = mov->renglon;
    int columna = mov->columna;
    int minRenglon = 0;
    int maxRenglon = 0;
                                                    //Para saber si hay N fichas del mismo color juntas en una columna
    int b = 1;
    int i;

    if(mov->jugador == 1) {
        Movimiento **array = juego->jugador1;
        while(b == 1) {
            b = 0;                                  //Para ser consistentes en las funciones de ganador se busca el limite superior de la linea ganadora
            i = 0;
            renglon++;
            while (b == 0 && i <= juego->turno) {
                if ((*(array + i))->columna == columna && (*(array + i))->renglon == renglon  && (*(array + i))->jugador == 1)
                    b = 1;
                else
                    i++;
            }
        }
        maxRenglon = renglon - 1;
        //int maxRenglon = renglon - 1;
```



```
        }
        maxRenglon = renglon - 1;
        //int maxRenglon = renglon - 1;

        b = 1;
        renglon = mov->renglon;                     //Después buscamos el límite inferior
        columna = mov->columna;
        while(b == 1) {
            b = 0;
            i = 0;
            renglon--;
            while (b == 0 && i <= juego->turno) {
                if ((*(array + i))->columna == columna && (*(array + i))->renglon == renglon && (*(array + i))->jugador == 1)
                    b = 1;
                else
                    i++;
            }
        }
        minRenglon = renglon + 1;
    }else{                                          //<--- Igual para el segundo jugador
        Movimiento **array2 = juego->jugador1;
        while(b == 1) {
            b = 0;
            i = 0;
            renglon++;
            while (b == 0 && i < juego->turno) {
                if ((*(array2 + i))->columna == columna && (*(array2 + i))->renglon == renglon && (*(array2 + i))->jugador == 2)
                    b = 1;
                else
```

estas decisiones respondieron a la primera idea de tener dos arreglos, se puede notar en el nombre del arreglo “array2”, y sobre todo en que primero compara fichas cuando vienen de jugador1 y en el else compara cuando vienen de jugador2.
Esta idea original de los dos arreglos se basaba en mejorar el desempeño de la aplicación, reduciendo a la mitad el número de comparaciones a la hora de realizar cualquier búsqueda o impresión. Pero tuvimos problemas por alguna razón con una de ellas y se optó por solo trabajar con una y modificar un poco las funciones existentes para que funcionaran con un solo arreglo de movimientos.

A manera de conclusión podría decir que me gustó el resultado obtenido del proyecto, aunque es verdad que no es perfecto hicimos un buen trabajo con los recursos que tuvimos, de tiempo y conocimiento. Me gustó los aprendizajes que nos

dejó el desarrollar esta aplicación y creó que cumplió con el objetivo de hacernos ver un poco hacia el contenido del curso y darnos cuenta lo que hemos avanzado. Una última nota a tomar en cuenta sería que la parte que más nos costó trabajo y nos costó funcionamiento en nuestra aplicación fue la librería gráfica que al final de cuentas no es parte del curso.

git de referencia:
https://github.com/GuerreroHJorge/conectaN

git de repositorio completo:
https://github.com/Kuryvc/CONECTAN