03.05 - TicTacToe

Lenguajes Estructurados

Introducción

El objetivo de este trabajo práctico es desarrollar un juego de tres en línea utilizando la biblioteca gráfica SDL2.

Especificaciones

- El juego debe permitir jugar contra la computadora o contra otro jugador.
- El juego debe permitir elegir el símbolo que utilizará cada jugador (X, O, etc.).

Requisitos previos

Tener instalado un compilador de C, como GCC (GNU Compiler Collection), en tu sistema.
Instala las siguientes bibliotecas SDL:

- SDL2
- SDL2_image
- SDL2_ttf

Para instalar las bibliotecas en sistemas basados en Debian/Ubuntu, ejecuta los siguientes comandos:

```
sudo apt-get update
sudo apt-get install libsdl2-dev libsdl2-image-dev libsdl2-ttf-dev
```

Compilación del programa

Para compilar el programa, ejecuta el siguiente comando en la terminal desde el directorio donde se encuentra el archivo ttt.c:

```
gcc .\ttt.c -o ttt.exe -lSDL2 -lSDL2_ttf -lSDL2_image
```

Esto generará un ejecutable llamado ttt.exe.

Explicación del código

Inclusión de bibliotecas

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_ttf.h>
#include <stdio.h>
```

Se incluyen las bibliotecas necesarias para utilizar SDL2, SDL_image y SDL_ttf. También se incluye la biblioteca estándar de E/S para imprimir mensajes de error si los hubiera.

variables globales

```
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;

SDL_Window *window = NULL;
SDL_Renderer *renderer = NULL;

int board[3][3] = {0};

SDL_Texture *player1_texture = NULL;
SDL_Texture *player2_texture = NULL;
```

Se declaran las constantes para el ancho y alto de la pantalla, así como las variables globales para la ventana y el renderizador de SDL. También se declara una matriz 3x3 llamada board que representa el estado del tablero y las texturas para las fichas de los jugadores 1 y 2.

Funciones auxiliares

El programa incluye varias funciones auxiliares, como load_texture(), add_piece(), init(), clean_up(), draw_piece() y draw_board().
Estas funciones se utilizan para cargar texturas, agregar fichas al tablero, inicializar y limpiar recursos SDL, y dibujar las piezas y el tablero en la pantalla.

Función principal - WinMain()

La función principal WinMain() inicializa SDL y las bibliotecas auxiliares, carga las texturas de las fichas y entra en un bucle principal. En este bucle, se manejan los eventos de entrada del usuario y se dibuja el tablero en la pantalla.

El bucle principal maneja eventos de cierre de ventana y eventos de clic del ratón.

Cuando se detecta un clic del ratón, se calcula la posición del tablero en la que se hizo clic y se intenta agregar una ficha en esa posición.

Si se puede agregar una ficha, el jugador actual cambia al otro jugador.

Finalmente, se llama a clean_up() para liberar los recursos de SDL antes de que el programa termine

Función load_texture()

```
SDL_Texture *load_texture(const char *path)
{
    SDL_Surface *loaded_surface = IMG_Load(path);
    if (!loaded_surface)
    {
        printf("Error al cargar la imagen %s: %s\n", path, IMG_GetError());
        return NULL;
    }

    SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, loaded_surface);
    if (!texture)
    {
        printf("Error al crear la textura a partir de %s: %s\n", path, SDL_GetError());
    }

    SDL_FreeSurface(loaded_surface);

    return texture;
}
```

```
SDL_Texture *load_texture(const char *path)
```

Descripción: Carga una textura desde un archivo de imagen.

Argumentos:

- const char *path: Ruta al archivo de imagen.

Valor de retorno:

- Un puntero a la textura cargada o NULL si hay un error.

Detalles de implementación:

- Carga la superficie de la imagen utilizando IMG_Load().
- Crea una textura a partir de la superficie cargada utilizando SDL_CreateTextureFromSurface().
- Libera la superficie cargada utilizando SDL_FreeSurface().
- Retorna la textura creada.

Función add_piece()

```
int add_piece(int x, int y, int player)
{
    if (x < 0 || x > 2 || y < 0 || y > 2)
    {
        return 0;
    }

    if (board[x][y] != 0)
    {
        return 0;
    }

    board[x][y] = player;

    return 1;
}
```

```
int add_piece(int x, int y, int player)
```

Descripción: Agrega una ficha en la posición (x, y) del tablero.

Argumentos:

- int x: Coordenada X de la posición del tablero.
- int y: Coordenada Y de la posición del tablero.
- int player: Número del jugador (1 o 2).

Valor de retorno:

- 1 si se pudo agregar la ficha
- 0 en caso contrario.

Detalles de implementación:

- Verifica si las coordenadas (x, y) están dentro del rango del tablero y si la casilla está vacía.
- Si se cumplen las condiciones, coloca la ficha del jugador en la posición (x, y) y retorna 1.
- Si no se cumplen las condiciones, retorna 0

Función init()

```
int init()
{
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        printf("Error al inicializar SDL: %s\n", SDL_GetError());
        return 0;
    }
    window = SDL_CreateWindow("Tateti", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN);
    if (!window) {
        printf("Error al crear la ventana: %s\n", SDL_GetError());
        return 0;
    }
    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
    if (!renderer) {
        printf("Error al crear el renderizador: %s\n", SDL_GetError());
        return 0;
    }
    if (IMG_Init(IMG_INIT_PNG) != IMG_INIT_PNG) {
        printf("Error al inicializar SDL_image: %s\n", IMG_GetError());
        return 0;
    }
    if (TTF_Init() < 0) {
        printf("Error al inicializar SDL_ttf: %s\n", TTF_GetError());
        return 0;
    }
    return 1;
}
```

```
int init()
```

Descripción: Inicializa SDL y las bibliotecas auxiliares.

Valor de retorno:

- 1 si la inicialización fue exitosa
- 0 en caso contrario.

Detalles de implementación:

- Inicializa SDL con el subsistema de video.
- Crea la ventana SDL.
- Crea el renderizador SDL.
- Inicializa SDL_image con soporte para imágenes PNG.
- Inicializa SDL_ttf.

Función clean_up()

```
void clean_up()
{
    SDL_DestroyTexture(player1_texture);
    SDL_DestroyTexture(player2_texture);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    IMG_Quit();
    SDL_Quit();
}
```

```
void clean_up()
```

Descripción: Libera los recursos de SDL y las bibliotecas auxiliares.

Detalles de implementación:

- Destruye el renderizador y la ventana de SDL.
- Finaliza SDL_ttf, SDL_image y SDL.

Función draw_piece()

```
void draw_piece(SDL_Texture *texture, int x, int y)

{
    SDL_Rect dest_rect;
    dest_rect.x = x * (SCREEN_WIDTH / 3) + (SCREEN_WIDTH / 6);
    dest_rect.y = y * (SCREEN_HEIGHT / 3) + (SCREEN_HEIGHT / 6);
    dest_rect.w = 32;
    dest_rect.h = 32;

    SDL_RenderCopy(renderer, texture, NULL, &dest_rect);
}
```

```
void draw_piece(SDL_Texture *texture, int x, int y)
```

Descripción: Dibuja una ficha en la posición (x, y) del tablero.

Argumentos:

- SDL_Texture *texture: Textura de la ficha a dibujar.
- int x: Coordenada X de la posición del tablero.
- int y: Coordenada Y de la posición del tablero.

Detalles de implementación:

- Establece las coordenadas y el tamaño del rectángulo de destino en función de las coordenadas del tablero y el tamaño de la pantalla.
- Copia la textura en el rectángulo de destino utilizando SDL_RenderCopy().

Función draw_board()

```
void draw_board()
{
    SDL_SetRenderDrawColor(renderer, 0xFF, 0xFF, 0xFF, 0xFF);
    SDL_RenderClear(renderer);

    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0xFF);
    for (int i = 1; i < 3; ++i)
    {
        SDL_RenderDrawLine(renderer, i * SCREEN_WIDTH / 3, 0, i * SCREEN_WIDTH / 3, SCREEN_HEIGHT);
        SDL_RenderDrawLine(renderer, 0, i * SCREEN_HEIGHT / 3, SCREEN_WIDTH, i * SCREEN_HEIGHT / 3);
    }

    // Dibujar las fichas en el tablero
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            if (board[i][j] == 1)
            {
                draw_piece(player1_texture, i, j);
            }
            else if (board[i][j] == 2)
            {
                draw_piece(player2_texture, i, j);
            }
        }
    }

    SDL_RenderPresent(renderer);
}
```

```
void draw_board()
```

Descripción: Dibuja el tablero y las fichas en la pantalla.

Detalles de implementación:

- Establece el color de fondo y borra la pantalla utilizando SDL_RenderClear().
- Establece el color de las líneas del tablero y dibuja las líneas utilizando SDL_RenderDrawLine().
- Itera a través de todas las posiciones del tablero y dibuja las fichas de los jugadores utilizando draw_piece():
- Si board[i][j] es 1, dibuja la ficha del jugador 1 en la posición (i, j).
- Si board[i][j] es 2, dibuja la ficha del jugador 2 en la posición (i, j).
- Actualiza la pantalla utilizando SDL_RenderPresent().

Función WinMain()

```
int WinMain(int argc, char *args[])
{
    if (!init()) {
        printf("Error al inicializar.\n");
    }
    else {
        player1_texture = load_texture("rojo.png");
        player2_texture = load_texture("verde.png");
        if (!player1_texture || !player2_texture) {
            printf("Error al cargar las texturas de las fichas.\n");
        }
        else {
            int quit = 0;
            SDL_Event e;
            int currentPlayer = 1;

            while (!quit)
            {
                while (SDL_PollEvent(&e) != 0)
                {
                    if (e.type == SDL_QUIT) {
                        quit = 1;
                    }
                    if (e.type == SDL_MOUSEBUTTONDOWN) {
                        int x = e.button.x / (SCREEN_WIDTH / 3);
                        int y = e.button.y / (SCREEN_HEIGHT / 3);
                        if (add_piece(x, y, currentPlayer)) {
                            currentPlayer = (currentPlayer == 1) ? 2 : 1; // Cambiar el jugador actual
                        }
                    }
                }
                draw_board();
                SDL_RenderPresent(renderer);
            }
        }
    }
    clean_up();
    return 0;
}
```

```
int WinMain()
```

Descripción: La función principal del programa.

Detalles de implementación:

- Llama a init() para inicializar SDL y las bibliotecas auxiliares.
- Carga las texturas de las fichas de los jugadores 1 y 2 utilizando load_texture().

- Si las texturas se cargaron correctamente, entra en un bucle principal:
    - Maneja los eventos de entrada del usuario utilizando SDL_PollEvent():
        - Si el evento es de tipo SDL_QUIT, establece quit en 1 para salir del bucle principal.
        - Si el evento es de tipo SDL_MOUSEBUTTONDOWN, calcula la posición del tablero en la que se hizo clic y llama a add_piece() para agregar una ficha en esa posición.
        - Si se pudo agregar una ficha, cambia el jugador actual al otro jugador.
    - Llama a draw_board() para dibujar el tablero y las fichas en la pantalla.
    - Actualiza la pantalla utilizando SDL_RenderPresent().
- Llama a clean_up() para liberar los recursos de SDL antes de que el programa termine.

Algorigmo de Minimax

El algoritmo Minimax es un algoritmo de búsqueda recursiva que se utiliza para determinar el mejor movimiento en un juego con dos jugadores que se turnan (como el tres en línea).

Minimax asigna un valor a cada movimiento posible. El jugador que maximiza (Max) intenta obtener el mayor valor posible, mientras que el jugador que minimiza (Min) intenta obtener el menor valor posible.

En el tres en línea, se puede asignar un valor de +10 a un estado de ganar para el jugador Max, -10 a un estado de ganar para el jugador Min y 0 a un empate.
El algoritmo decidirá el mejor movimiento basándose en estos valores.

Primero, hace falta implementar tres funciones adicionales:

- is_winner(int player) verifica si un jugador ha ganado el juego.
- is_full_board() verifica si el tablero está lleno (es decir, si no quedan movimientos posibles).
- minimax(int depth, int isMaxPlayer) evalúa recursivamente los movimientos posibles y devuelve el mejor valor posible.
- is_empty(int x, int y) verifica si una casilla está vacía
- remove_piece(int x, int y) para deshacer un movimiento

```c
int is_winner(int player)
{
    // Verifica filas, columnas y diagonales para ver si el jugador ha ganado
    // Retorna 1 si el jugador ha ganado, 0 en caso contrario
}

int is_full_board()
{
    // Verifica si el tablero está lleno
    // Retorna 1 si el tablero está lleno, 0 en caso contrario
}

int is_empty(int x, int y)
{
    // Verifica si la casilla (x, y) está vacía
    // Retorna 1 si la casilla está vacía, 0 en caso contrario
}

void remove_piece(int x, int y)
{
    // Elimina la pieza en la posición (x, y)
}
```

```
int minimax(int depth, int isMaxPlayer)
{
    // Si alguien ganó o si el tablero está lleno, retorna el valor correspondiente
    if (is_winner(1))
        return 10 - depth;
    if (is_winner(2))
        return -10 + depth;
    if (is_full_board())
        return 0;

    int bestValue;
    if (isMaxPlayer)
    {
        bestValue = -1000; // Inicializa el mejor valor para Max en un valor muy bajo
        // Itera a través de todos los movimientos posibles
        // ... Realiza el movimiento
        // ... Llama a minimax() recursivamente con el siguiente jugador
        // ... Deshace el movimiento
        // ... Compara y actualiza el mejor valor
    }
    else
    {
        bestValue = 1000; // Inicializa el mejor valor para Min en un valor muy alto
        // Itera a través de todos los movimientos posibles
        // ... Realiza el movimiento
        // ... Llama a minimax() recursivamente con el siguiente jugador
        // ... Deshace el movimiento
        // ... Compara y actualiza el mejor valor
    }
    return bestValue;
}
```

Luego, modificar la función ia_move() para utilizar el algoritmo Minimax:

```
void ia_move(int currentPlayer)
{
    int bestValue = 1000; // Inicializa el mejor valor en un valor muy alto
    int bestMoveX = -1;
    int bestMoveY = -1;

    // Itera a través de todos los movimientos posibles
    for (int x = 0; x < 3; x++)
    {
        for (int y = 0; y < 3; y++)
        {
            // Comprueba si la casilla (x, y) está vacía
            if (is_empty(x, y))
            {
                // Realiza el movimiento
                add_piece(x, y, currentPlayer);

                // Calcula el valor minimax del movimiento
                int moveValue = minimax(0, 1); // El jugador humano es el jugador Max (1)

                // Deshace el movimiento
                remove_piece(x, y);

                // Compara y actualiza el mejor valor y la mejor posición
                if (moveValue < bestValue)
                {
                    bestValue = moveValue;
                    bestMoveX = x;
                    bestMoveY = y;
                }
            }
        }
    }

    // Realiza el mejor movimiento encontrado
    add_piece(bestMoveX, bestMoveY, currentPlayer);
}
```

Implementación de funciones `ia_move_minimax` y `minimax`

Explicamos paso a paso la implementación de las funciones del algoritmo, comenzando con algunos conceptos básicos del algoritmo Minimax y luego describiendo cómo se implementan en el código.

Fundamentos del algoritmo Minimax

El algoritmo Minimax es una técnica de búsqueda en árbol utilizada en juegos de dos jugadores con información perfecta, como Tic-Tac-Toe.

La idea detrás de Minimax es simular todos los posibles movimientos que pueden hacer tanto el jugador como el oponente, y luego elegir el movimiento que maximice la posibilidad de ganar y minimice la posibilidad de perder.

El algoritmo funciona de manera recursiva, donde cada llamada a la función Minimax representa un movimiento en el árbol de juego.

Los nodos en el árbol de juego representan diferentes estados del tablero de juego.

La función Minimax devuelve un valor que indica qué tan "bueno" es un movimiento dado para el jugador actual.

Implementación de `ia_move_minimax`

La función `ia_move_minimax` se utiliza para encontrar el mejor movimiento para la IA utilizando el algoritmo Minimax.

```
void ia_move_minimax(int currentPlayer)
{
    int best_value = -10000;
    int best_x = -1;
    int best_y = -1;
```

Inicialmente, establecemos `best_value` en un valor muy pequeño (-10000) para asegurarnos de que cualquier valor que obtengamos del algoritmo Minimax sea mayor que este valor inicial.

Las variables `best_x` y `best_y` almacenarán las coordenadas del mejor movimiento encontrado por la IA.

```
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            if (board[i][j] == 0)
            {
                board[i][j] = currentPlayer;
                int value = minimax(board, 0, 0);
                board[i][j] = 0;
```

Recorremos todas las casillas del tablero y, si encontramos una casilla vacía (sin ficha), simulamos un movimiento colocando la ficha del jugador actual (IA) en esa casilla.

Luego, llamamos a la función `minimax` para evaluar el valor de este movimiento. Después de evaluar el movimiento, revertimos el cambio en el tablero (quitamos la ficha de la casilla).

```
                if (value > best_value)
                {
                    best_value = value;
                    best_x = i;
                    best_y = j;
                }
            }
        }
    }
```

Si el valor obtenido de la función `minimax` es mayor que el mejor valor actual, actualizamos el `best_value` y las coordenadas `best_x` y `best_y`.

```
    if (best_x != -1 && best_y != -1)
    {
        add_piece(best_x, best_y, currentPlayer);
    }
}
```

Una vez que hemos explorado todos los posibles movimientos, realizamos el mejor movimiento encontrado colocando la ficha en las coordenadas `best_x` y `best_y`.

Implementación de `minimax`

La función `minimax` es la implementación del algoritmo Minimax propiamente dicho.

```
int minimax(int depth, int is_maximizing, int player)
{
```

La función `minimax` toma tres argumentos: `depth`, `is_maximizing` y `player`. `depth` representa la profundidad actual en el árbol de búsqueda, `is_maximizing` es un booleano que indica si el jugador actual está maximizando o minimizando, y `player` es el jugador para el que estamos buscando el mejor movimiento (1 o 2).

```
    int winner = check_winner();
    if (winner != 0)
    {
        return winner == player ? 1 : -1;
    }
```

Se llama a la función `check_winner` para verificar si hay un ganador en el estado actual del tablero.

Si hay un ganador, la función devuelve un valor que indica qué tan bueno es el estado actual del tablero para el jugador `player`.

Si `player` es el ganador, la función devuelve 1; si el oponente es el ganador, devuelve -1.

```
    if (is_full())
    {
        return 0;
    }
```

Se llama a la función `is_full` para verificar si el tablero está lleno y no hay más movimientos posibles.

Si el tablero está lleno, entonces el juego ha terminado en empate y se devuelve un valor de 0.

Si el juego no ha terminado, evaluamos todos los movimientos posibles para el jugador actual.

La función Minimax se llama recursivamente para cada movimiento posible, y la profundidad aumenta en 1 en cada llamada recursiva.

- Si el jugador actual es el jugador maximizador (IA), buscamos el movimiento con el mayor valor posible.
- Si el jugador actual es el jugador minimizador (oponente), buscamos el movimiento con el menor valor posible.

Para cada movimiento, actualizamos el tablero, llamamos a la función Minimax y luego revertimos el cambio en el tablero.

Al final, la función Minimax devuelve el mejor valor encontrado para el jugador actual en función de si es un jugador maximizador o minimizador.

```
    int best_value = is_maximizing ? -1000 : 1000;
    int opponent = player == 1 ? 2 : 1;
```

Se inicializa la variable `best_value` con un valor extremo, dependiendo de si el jugador actual es maximizador o minimizador. También se calcula el número del oponente (1 o 2).

```
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            if (board[i][j] == 0)
            {
```

Se itera a través de todas las casillas del tablero en busca de casillas vacías.

```
                board[i][j] = is_maximizing ? player : opponent;
                int value = minimax(depth + 1, !is_maximizing, player);
                board[i][j] = 0;
```

Para cada casilla vacía, se coloca temporalmente la ficha del jugador actual en la casilla y se llama a la función `minimax` recursivamente, aumentando la profundidad y alternando entre el jugador maximizador y minimizador.

Luego, se deshace el movimiento temporal en el tablero.

```
                best_value = is_maximizing ? fmax(best_value, value) : fmin(best_value, value);
            }
        }
    }
```

Se actualiza el valor de `best_value` utilizando la función `fmax` o `fmin` en función de si el jugador actual es maximizador o minimizador.

```
    return best_value;
}
```

Finalmente, se devuelve el valor de `best_value`, que representa el valor óptimo del estado actual del tablero para el jugador `player`.

La función `minimax` explora exhaustivamente todas las posibles secuencias de movimientos en el juego de Tic-Tac-Toe hasta que encuentra un estado terminal (es decir, un estado donde el juego ha terminado), y luego asigna un valor a ese estado según si es favorable para el jugador `player` (1), favorable para el oponente (-1) o un empate (0).

El algoritmo Minimax se basa en la suposición de que ambos jugadores actuarán de manera óptima para maximizar su propio beneficio y minimizar el beneficio de su oponente.

Por lo tanto, durante la exploración del árbol de búsqueda, la función `minimax` alterna entre maximizar y minimizar los valores de los estados del tablero.

Al final de la exploración, la función `minimax` devuelve el valor óptimo del estado actual del tablero para el jugador `player`.

Este valor se utiliza en combinación con la función `ia_move_minimax` para determinar el mejor movimiento que el jugador controlado por IA debe realizar.