Four Colour Theorem C Program Documentation

Requirements to Build the Program

Software dependencies:

To compile and run the program, the following tools and libraries are required:

- C compiler

SDL2 library

- SDL2.dll (runtime)
- SDL2.lib / libSDL2.a (linking)
- SDL2main library if required by platform
- Header files: SDL.h.

#define and constants

```
#define Region_CountMax 100

#define Color_Count 4

#define Cell_Size 2

#define WINDOW_TITLE "Four Color Theorem"

#define HALLOFFAME "hall_of_fame.txt"


const int SCREEN_WIDTH = 800;

const int SCREEN_HEIGHT = 600;
```

- Region_CountMax —limit on how many Voronoi regions can exist.

- Color_Count — fixed number of available colors.

- Cell_Size — size of one rendered cell in pixels (2×2 in this case).

- WINDOW_TITLE — SDL window title.

- HALLOFFAME — filename used for storing results.

Constants are used for simplification of debugging or tuning. Allows editing only one place. Moreover, this is how Window Size is defined for SDL in most official documentation. And all of this values never going to be changed.

GameState enum

```
typedef enum {
  Menu,
  Game
} GameState;

typedef enum {
    Easy,
    Medium,
    Hard,
  }Difficulty;
```

I used enums for game states and and difficulty levels storing. Enum is usually self-explanatory because we use words like Easy, Medium, Hard instead of numbers like 0,1,2. Also can easily be used in tandem with other variables and arrays like here:

```
const int DIFF_REGION_COUNTS[] = {
  5,
  70,
  100
};
```

Meaning, regions amount easily connected to difficulty level. Easy-5, Medium-70, Hard-100.

Structures

```
typedef struct {
  SDL_Point point;
  int colorIndex;
} Region;
```

This struct keeps the center point of all the regions. Based on this point all the regions build their borders according to Voronoi.

Color index is current color of the region. -1 by default and then might be change to 0-3 and use colors according to *RGB_palette.*

```
struct Game {
    SDL_Window *window;
    SDL_Renderer *renderer;

    Region *regions;
    bool **adjucency;

    int chosenColor;
    int regionCount;

    Difficulty difficulty;
    GameState gameState;

    Uint32 startTimer;
    Uint32 finishTimer;

    bool winState;
};
```

The main struct of the game. Everything connected to the game logic is collected here.

window, renderer — SDL main parameters for drawing

regions — Dynamic array of regions

adjucency — dynamic matrix.

chosenColor — current color chosen by the user regionCount — how many regions are the in the game (depending on difficulty)

difficulty, gameState — self-explanatory

startTimer, finishTimer — start time and total time spent to finish the level in ms.

winState — win flag to end the game

When keeping everything in one struct, passing states to functions is very straightforward.

Adjacency matrix

adjucency[i][j] == true → region i touches region j

- Matrix size is Region_CountMax × Region_CountMax.
- Only border cells (right and bottom neighbor comparisons) are needed to determine adjacency.

Voronoi approach

Regions are not drawn as polygons. Instead, the screen is divided into small square cells (2×2 pixels) and each cell chooses the nearest region center

- No floating-point geometry
- Works reliably for color-filling and adjacency detection
- Easy to visualize and debug

Functions

sq2(x,y) – squared Euclidean distance

This function is a utility function that avoid calculating sqrt() and can be called in multiple places, which saves plenty line of code. Voronoi nearest-center comparisons do not require actual distance.

Input: x, y integer components.
Output: x*x + y*y.

main()

Game creation and initialisation happens here. Inside main the loop handles user input, switches between Menu and Game rendering stated, chechs wether the game is finished with winState, limits framerate.

Continuous rendering is required for interactive games or application in SDL, as it works entirely on event polling.

resultSave(name, time)

This function opens file *hall_of_fame.txt* in adding mode “a”, and writes string like this “Ivan 12.53” (Username and time needed to finish the game)

- Input: string name, integer milliseconds
- Output: none
- File mode: append (“a”)

hallOfFamePrint()

Opens file in reading mode and prints all entries to the console.

- **Input:** none
- **Output:** console output
- **File mode:** read (“r”)

setDifficulty(game, diff)

Role: Start or restart a level.

This function could basically be named level create/reload. It sets difficulty, takes required region amount, generates random points on the map, checks adjacency, nullifies victory flag and timers. This function is used when R is pressed to reload and when initial level is created.

game_cleanup(game)

Role: Final memory cleanup

This is a must-have function when working with SDL. It removes all SDL-objects like window and renderer and quits SDL. It is also a great place to free() adjacency check array and regions array.

mouse_input(game, x, y)

This function finds the closest region to the mouse click point. It sets the color of this concrete region to be the current chosen color.

- **Input:** mouse coordinates
- **Output:** modifies region color

regionsGenerator(game)

Easy implementation of rand() function. Every time regionsGenerator() is called it creates a specific number of dots on the plane. Later these dots serve as region “center” .

adjucencyCheck(game)

First, reset the entire adjacency matrix to false. Than iterates through all "cells": Determines which region is closest to the center of this cell— c. Looks at the right and bottom neighboring cells: If the closest region is different (cRight/cDown), then regions c and cRight/cDown are adjacent → set true in both directions.

Instead of complex geometry (intersecting region boundaries), "grid" approximation is used. Checking only the right and bottom is sufficient: If cells (A,B) are adjacent, then either adj[A][B] or adj[B][A] will already be set somewhere during the traversal. The code specifically sets both [c][cRight] and [cRight][c] symmetrically.

winCheck(game)

Iterates through every region. If color index is -1 returns false, if there is a conflict returns false as well. If each region is painted and no conflicts – return winState.

conflictCheck(game, regionIndex)

Function utilises adjucencyCheck() by checking whether any neighbouring regions is at conflict with another one

find_closest_region(game, x, y)

For point (x,y) find the closest center of the region. If this pixel is the closest to the center, than this pixel is a part of this region. This function is key in rendering the borders of the regions and mouse click logic.

game_renderer(game)

For each cell finds the closest region (closest). Determine whether this cell is a border: Compare with the cell to the right (closestRight) and below (closestBelow);

If at least one is different, it's a border draw it black. Otherwise, take the region's color by colorIndex. If the region is in conflict (conflictCheck), lighten the color (mix with white). Uncolored regions are gray (80,80,80).

Draws white dots at the centers of the regions (for debugging). Draw a color palette at the bottom of the screen:

SDL_RenderPresent — displays the frame.

Instead of calculating component-by-component for each pixel, speeds up rendering. Highlighting conflict areas makes the game visually clear.

menu_renderer(game)

This is the easiest function of all – just creates 3 rectangles in menuState.

sdl_initialise(game)

Initialises SDL and Dynamic memory. SDL initialisation is constant for most of the SDL projects and just has to be there in any case. Some parameters can be changed for different game setups like window types and console types.

```
game->regions = (Region*)malloc(Region_CountMax * sizeof(Region));
...
game->adjucency = (bool**)malloc(Region_CountMax * sizeof(bool*));
...
for (int i = 0; i < Region_CountMax; ++i) {
   game->adjucency[i] = (bool*)malloc(Region_CountMax * sizeof(bool));
}
```

Dynamic memory is initialised once and then reused when level is changed.

Resources I used to create this game:

https://rosettacode.org/wiki/Voronoi_diagram

https://solhsa.com/gp2/ch02.html

https://lazyfoo.net/tutorials/SDL/01_hello_SDL/index2.php