**Object-oriented programming ex4- Circle the Frog.**

Submitted by:

Ely Asaraf, ID: 319027355

Nadav Malul, ID: 206762973

**Intro**

In this exercise we recreated a board game called “Circle the Cat”.

The board contains an 11x11 matrix of tiles, each tile can be either "blocked" or free.

In the middle of the board there is a frog, your mission is to circle the frog with blocked tiles so that the frog will not be able to move.

Rules:

It’s a game of turns between the user and the frog.

The user starts with choosing one unblocked tile and blocking it.

Afterwards, it's the frog's turn. The frog will calculate the closest path to escape the board and will take the first step of this path.

If the user successfully blocked the frog from all sides: the user will level up to the next level.

If the frog managed to escape the board: the user will lose, after losing, the user will gain a random new level with the same amount of "BLOCK\_NUMBER" of tiles.

Note: It is not enough to trap the frog in a certain space. The stage is won only if the frog has nowhere to go.

**Classes and design:**

**Class Controller:**

Members:

Board m\_board – game board.

Utility m\_utility – draws buttons, highlights buttons etc.

Sound m\_sound – plays background music, sounds effects, dec/inc volume.

Controller is used to connect between different parts of the code.

Whenever mouse is clicked, controller sends the click position to class Board in order to handle the mouse click, after handling the click, controller will also play the relevant sound effect for the click by using class Sound.

Once the user clicks on a button, controller will handle the click by using class Utilities.

**Class Board:**

Members:

Frog m\_frog – The frog game object is within the board.

Graph m\_graph – Finds the path to escape the board.

vector<vector<Tile>> m\_tiles – contains the board data.

This class contains the data structure which represents the board ( vector<vector<Tiles>>)

Board's job is to update changes made by the user (for example blocking a tile) and to check the validity of each mouse click on the board.

Also, board creates the graph data structure since the board. Since the board is the only one that contains the data of the matrix its his responsibility to connect between the graph (which contains all vertices and edges between non occupied tiles) and the frog that needs to find the next step.

Class graph:

Members:

int V – numbers of vertices in the graph.

vector<pair<int, int>>\* adj – contains a pair of vertex and weight.

The graph finds the closest escape route for the frog.

If the frog has no routes to escape from the board, the graph will return an invalid step, indicating that the frog cannot escape the board.

The graph updates the distance between vertices when the user blocks a tile (by doing so edges are removed).

Class GameObject:

Members:

sf::Sprite m\_sprite;

GameObject represents the different in-game objects.

Class Frog: public GameObject :

Members:

Animation m\_frogAnim – for the frog's move animation.

sf::Sprite m\_newPosition – frog's next step.

This class represents the frog.

Class Tile : public GameObject:

Members:

bool m\_blocked - to know if the frog can step on this tile.

sf::Vector2u m\_index – Tile's index on the board.

This class represents a tile on the board.

Class Animation:

For handling animations.

Class Buffers:

Single load buffer for each game sound, this class is used by class Sounds.

Class Textures:

Single load texture for each game texture, this class is used by classes GameObject and Utility.

Class Utility:

Members:

std::vector<sf::Sprite> m\_buttons – contains all buttons sprites

This is a Utility class used by Controller to add effects and manage utilized details.

**Worth Mentioning data structures:**

Board::vector<vector<Tiles>>: each tile contains a Boolean which represents a blocked/unblocked tile.

Graph::vector<pair<int, int>>\* adj - contains a list of all adjected vertices and the weight of the track between those.

**Worth Mentioning Algorithms:**

Convert 2d matrix to 1d array: (Graph and board)

Inspiration: the board contains elements in a 2d matrix, the graph should contain elements as indexes of vertices.

Solution: By building the graph from the board, each element located in [x][y] was converted to a 1d index by this formula: index = y + (x \* BOARD\_SIZE).

(This function is 1:1 and onto)

By receiving the next step from the graph, we needed to convert the formula the other way around, we used the opposite function so that:

x = index / BOARD\_SIZE

y = index % BOARD\_SIZE

This indexing system is the way that the board and the graph transfer data between them.

**Functions that use this method**:

From 2d to 1d : Graph::handleEdges

From 1d to 2d: Board::relocateFrog

Graph::Shortest path:

We used Dijkstra's algorithm (with changes).

From Dijkstra's algorithm we received the shortest path from a source vertex to each other vertex in the graph.

Adjustments: Added a vector in the size of the vertices in the graph, the goal is for each vertex index discovered, we wanted to keep the index of its predecessor.

For example, if vector[0] = 11 that means that the vertex with index 11 discovered vertex 0 and the shortest path to vertex 0 is via vertex 11.

In this way we were able to find the shortest path from frog's position to any edge-tile, and by checking the predecessor of each index we were able to find the frog's next step towards the grand escape.

Board::shuffleBlocked:

This algorithm is responsible for randomizing blocked tiles for each level.

Input: Current number of level.

The algorithm starts by inserting the number of wanted blocked tiles (by current level – as decided in the macros) into the beginning of a vector of Booleans.

For example, at the first stage the algorithm inserts 14 true elements into the beginning of the vector (the rest are set as false). Afterwards, the algorithm shuffles between all the Booleans by using std::shuffle to randomize the blocked tiles’ locations.

**If the user wants to add a level or modify the number of blocked tiles in the beginning of the level:**

First: go to macros.h.

Adding a level: Change the integer "NUM\_OF\_LEVELS" to be the number of levels you want.

Changing a level's difficulty: increase/decrease the co-responding index value in the array "BLOCKED\_NUMBER" which is in macros.

Undo algorithm:

After each valid step by the user, the board keeps (as a pair) – the location of the tile which was blocked by the user and the previous frog location (before moving). This pair is kept in a vector of pairs as a member of board.

By pressing undo – the algorithm will take the last pair inserted to the vector, unblock the given tile, restore edges which was removed from the graph and will set the frog's position into the previous frog's position.

**Known bugs:**

Currently there are no known bugs in the current version.

**Other notes:**

None.