Program name: Minesweeper

Author: asd135hp1

Description: A recreation of classical Minesweeper game with a little bit renovation. Aside from three modes that presents in any kind Minesweeper games (Easy, Medium, Hard), a new mode is up to use, which is called "Multiplayer" mode. The mode will either create or join a game so that players will compete to each other through real-time communication to the server (or real-time multiplayer). What it means is that this program will take responsibilities in controlling data to be sent to the server also receiving only specific data needed for the multiplayer mode to be functional CONTINUOUSLY. It is done by using multi-threading system. Moreover, a little more specialty in this game is that each mode will have an end message, indicating how well the player played using number of flags used and the amount of time the player solved the puzzle.

More detailed description about some functions in the program:

-> ***Question:*** Hey, it is a bit messy (by a bit, I mean it is a total mess) in main *Gosu Window* function. What is going on there? An accident?

Answer: Oh, there is nothing to worry about, really. Multi-threading system is always messy in the first place.

-> ***Question:*** Did I hear 'multi-threading'? Why are you doing it? It is not even for beginner anymore...

Answer: Good question... The reason I do it is mainly because of how communicating to server always blocks main thread-

-> ***Question:*** Hey, that's what **Gosu::Window#update** function do! Why don't you use it instead?

Answer: Well, it was kind of my plan in the first place to go for multi-threading anyways, so I kept thinking that I must implement it no matter what. I must admit that it can be simplified with **Gosu::Window#update**, but at the end of the day, the nature of this program does not allow it to happen!

-> ***Question:*** What do you mean by that?

Answer: Well, even if I put everything inside **Gosu::Window#update** function, what will happen is that the multiplayer system will be delayed significantly! Could you imagine playing FPS game and you have to wait for one packet of data to be transmitted to your game, just to continue and send the next request to the server and then waiting repetitively for that packet to be received by your game? That just does not right and can be frustrating. So that I use multi-threading.

-> ***Question:*** Ok... You kind of hit off the mark, though... Where's the explanation?

Answer: Oh right! This is what will happen in this program when you launch it in multiplayer mode:

- First, you are in multiplayer chooser now and there are 3 buttons: *host a game*, *join a game* and *exit*
- When you choose to host/join a game, it will start another thread to *create*/*find* game id in the server through global *game id* + 1/*queue* through `***establish_game_session***`/`***join_available_game_session***`. As the result from those functions, it returns a bunch of game data, containing *board's width*, *board's height* and *game's number of bombs*.
- Then, it will merge that game data to current player's data storage of the program.
- After that, another thread will start to run to communicate to server endlessly and seamlessly, which I will explain later.
- Then, the thread will start to set initial opponent's data in current *Window* object (which is the game). Mostly, it defines opponent's margin to centralize his board to the center of right-hand screen
- Then, it will move from loading screen to multiplayer screen (from *UI stage* of -1 to *UI stage* of 8) #
- It will set number of bombs to that thread's hash value so that in the next update frame, number of bombs will be set to screen and player will see it when multiplayer screen is up.
- Then, main player's margin and square size is changed with some hard coded / calculated values, indicating maximum margin from screen that main player's board can spread (because if square size for each square on the board is a constant, the board will expand indefinitely to even outside of the screen, depends on that game's data; which is not a good thing. That thing is there to restrict the board's play field)

- Finally, it will initialize the game, which is the same as single player

-> ***Question:*** Good grief. I am already lost! So, what does that another thread do?

Answer: That another thread is what miracles happen and the evils also. Basically:

- At the beginning, it will define the thread's global variable of "*someone_wins*", indicating that someone has won the game, regardless which side so that the loop below will stop initiating threads that communicating to server.
- In that loop, to start off, it will merge an array of thread's data together to get the frame of the game to draw to the right-hand side of the main player's screen. Any threads that has been finished with status of <Thread #abcxyz dead> or false will be removed to free some memory.
- Then, immediately, it will start another thread inside that thread and append it to the mentioned array. The thread does the job of directly sending current player's board and current player's game state to the server and receiving opponent's board and that game's state from server. If someone wins, it will set its parent thread's global variable of *:someone_wins* to true to end server communication and also prepare to send end game message to the player.
- That child thread's reference is then pushed into that array of thread (mentioned in second dot point)
- The loop then finally sleep for 1 second to ready to start another communication thread
- Outside of the loop, it is when the game is all finished. All the appended child threads will be exited (not sure if it is safe enough but this is Ruby and everything is sure to be safe enough!)
- Finally, it will log a message, indicating that the thread is indeed finished.

-> ***Question:*** Ok... So how about `*initialize_game*`?

Answer: That function, huh? It does the following:

- Firstly, it will initialize both initial version of minesweeper board, containing 0s for underlying board and 'c's for upper surface board, which indicates covered squares.
- Then, it will initialize bomb positions, depending on the given number of bombs. With underlying board passed as a parameter, the bombs positions are there to spread -1 across the board (-1 is bomb)

- After that, `*distribute_bomb*` is called to increase every 8 squares' value by 1 around any bombs not including the bombs.
- Finally, some flags are set to make the game functional.

-> ***Question***: How about the game's mechanics?

Answer: It is simple. As you can see, every single time you click left mouse button, the game automatically reveals any white squares and some numbers, except for flags (a bit hard to explain); and calculate if the remaining covered squares are in the same number as bomb number. If that statement is true, it will indicate that the player wins. However, if the player clicks on the bomb square, it will be a losing experience instead. The right clicks are there to toggle flags and flag number. No more flags when its number reaches zero. This mechanic is applicable to both single player and multiplayer modes!