Minesweeper

Project 3 for CMPE230 - Spring 2024 Course.

About

Minesweeper A QT Widget Application written in C++.

The application supports the following operations:

- Start / Restart Game
- Reveal cells
- See Score
- Hint
- Flag/Unflag cells

In Minesweeper, only unrevealed cells are clickable. Left-clicking reveals numbers indicating neighboring mines, empty cells, or mines, while right-clicking toggles flags on unrevealed cells. Once revealed, cells become unclickable. The restart button starts a new game with a different mine layout, and a label shows the number of revealed cells as the score. The game's grid and mine count are configurable.

The game ends when a mine is clicked (loss) or all non-mine cells are revealed (win). After the game ends, all mines are revealed, a notification appears, and all cells become unclickable until a new game starts. The restart button remains functional.

The hint button suggests a safe, unrevealed cell that does not contain a mine. If no safe move exists, it offers no suggestion. If the suggested cell isn't revealed by the player, a second click reveals it.

Usage

Example Game Flow:

> M = 20 N = 20 K = 45

Drive Link

Program Structure (QT Creator)

```
Headers/
   cell.h
   utils.h
Sources/
   cell.cpp
   utils.cpp
   main.cpp
Resources/
   images.qrc
minesweeper_game.pro
```

Program Structure (Directory)

```
cell.h
cell.cpp
utils.h
utils.cpp
main.cpp
minesweeper_game.pro
minesweeper_game.pro.user
report.md
report.pdf
images/
```

Classes and Methods

Cell Class

This class is responsible for managing a single cell in an application. A cell can be either of the following:

```
- Empty
- Flag
- Mine
- Num0
- Num1
- Num2
- Num3
- Num4
- Num5
- Num6
- Num7
- Num8
- Hint
- WrongFlag
```

Some Important Methods:

- `updateImage()`: Updates the cell's image based on its mode and revealed status.
- `checkWinCondition()`: Checks if the player has won the game. If all non-mine cells are revealed, the player wins and a win message is shown.
- `reveal()`: Reveals the cell, updating its image and emitting the clicked signal. If the cell is empty or contains a zero, it reveals its neighboring cells. Also checks for win condition after revealing the cell.
- `revealEmptyNeighbors(int row, int col)`: Reveals all neighboring cells that are not yet revealed. This function is called recursively for empty cells to reveal large empty areas.

Signals

Signals were utilized for various features including:

- Restart button
- Hint button
- Right-click action on a cell.
- Left-click action on a cell.

This implementation ensured smooth and responsive interactions within the game interface.

Utilities & Usages

- `void updateSafeAndMineCells(Cell ***cells, int numRows, int numCols, bool &changed)`: Decides whether cells are guaranteed to be safe or mines from the perspective of a user. Used in the hint algorithm. If the number of guaranteed mines around a cell equals the cell's number, the remaining hidden cells around it are marked as safe. If the number of guaranteed mines plus hidden cells around a cell equals the cell's number, the remaining hidden cells are marked as mines.

- `void giveHint(Cell ***cells, int numRows, int numCols)`: Provides a hint to the player by identifying a safe cell that hasn't been revealed. It updates the status of cells using the updateSafeAndMineCells function until no changes occur. If a safe cell is found, it is marked as a hint. If no safe cells are available, a message is displayed.

- `void lockAllCells(Cell ***cells, int numRows, int numCols)`: Locks all cells on the game board, preventing any further interactions. This is useful for ending the game or preventing changes during certain operations.

- `void placeMines(Cell ***cells, int numRows, int numCols, int numMines)`: Randomly places mines on the game board, ensuring that each cell can only contain one mine. The number of mines placed is determined by the numMines parameter.

- `void setNumbers(Cell ***cells, int numRows, int numCols)`: Sets the number for each cell based on the number of surrounding mines. Cells without mines are assigned a number indicating how many mines are adjacent to them.
- `void openAllMines(Cell ***cells, int numRows, int numCols)`: Reveals all mines on the game board, typically used when the game is lost. Temporarily disconnects the clicked signal to prevent interaction during the reveal process.
- `void resetHintState(Cell ***cells, int numRows, int numCols)`: Resets the hint state for all cells on the game board. This is useful for clearing any hint markers before a new hint is provided.
- `void cleanup(Cell ***cells, int numRows, int numCols)` : Cleans up the dynamically allocated memory for the game board. Deletes each cell and frees the memory allocated for the rows and the cell array.
- `void updateScoreLabel(QLabel *scoreLabel, int revealedCount, int *score)` : Updates the score label with the current score. The score is incremented by the number of revealed cells and displayed on the score label.

Program Design and Rationale

- The core of the Minesweeper game is a grid layout containing cells. Each cell is an interactive component that the player can click to reveal whether it is a mine or a safe spot. The grid is dynamically created based on the game configuration (`N` x `M` cells).
- The gameplay is governed by several functions that manage the game's logic, such as placing mines, calculating adjacent mines, and handling user interactions.
- Each cell is connected to a signal that updates the game's state and the score based on the user's actions. Clicking on a cell will either reveal a mine, ending the game, or show a safe spot, possibly revealing adjacent safe areas automatically if they are free of mines.
- The restart button resets the entire game state, including re-randomizing the placement of mines and resetting the score. The hint button utilizes game logic to provide non-destructive guidance to help players advance in the game.

Game Setup

We've set up basic variables to be consumed by the objects and methods outside the main for global access and ease of use:

The variables include:

```
// Config Variables
const int N = 16;
const int M = 16;
const int K = 50;

// UI Variables
const int buttonWidth = 54;
const int cellSize = 15;
const int paddingX = 22;
const int paddingY = 52;

// State Variables
bool gameOver = false;
int score = 0;
```

Window & UI

On the head of the main, we declare basic window and ui settings

```
// Global application-level configuration
QApplication app(argc, argv);
QWidget mainWindow;
QIcon appIcon(":/images/mine.png");
mainWindow.setWindowTitle("Minesweeper");
app.setWindowIcon(appIcon);
```

```
// Window configuration
mainWindow.setFixedSize(cellSize * N + paddingX, cellSize * M + paddingY);
QVBoxLayout *mainLayout = new QVBoxLayout(&mainWindow);
QHBoxLayout *topLayout = new QHBoxLayout;
mainLayout->addLayout(topLayout);
```

as well as other styling related definitions:

```
gridLayout->setSpacing(0);
gridLayout->setColumnStretch(0, 0);
gridLayout->setRowStretch(0, 0);
gridLayout->setColumnMinimumWidth(15, 15);
gridLayout->setRowMinimumHeight(15, 15);
gridLayout->setContentsMargins(0, 0, 0, 0);
for (int i = 0; i < N; ++i) {
    gridLayout->setRowStretch(i, 0);
    gridLayout->setRowMinimumHeight(
        i, 15);
}
for (int j = 0; j < M; ++j) {
    gridLayout->setColumnStretch(j, 0);
    gridLayout->setColumnMinimumWidth(
        j, 15);
}
```

State

We keep state variables as following and alter throughout the program:

```
bool gameOver = false;
int score = 0;
```

OOP

We did not need an exhaustive use of classes in this project. The objects in the applications are mere buttons and cells.

The only class we have implemented is `Cell`

Cleanup and Resource Management

Upon closing the application, all dynamically allocated resources, particularly the grid of cells, are properly cleaned up to prevent memory leaks. This ensures that the application runs efficiently and maintains resource integrity throughout its lifecycle.

Logging and Debugging

- While we were implementing class method, we have used `qDebug` utility to trace the execution flow of the program. An example is the overrided `mousePressEvent()` method on the `Cell::QWidget` method where we have checked for the type
- To track the hint algorithm, the safe cells and guaranteed mine cells were debugged via the console. This debugging process facilitated the identification and verification of cells, ensuring the accuracy and functionality of the hint mechanism.

Challenges Encountered

- One of the difficulties in the project was that the cells had to be reset while calculating the score. Before I realized that I needed to reset, although the scores were calculated properly in the first round, the cells revealed in the next round were not included in the score again. Resetting the signal coming to the cells was sufficient to solve this situation.

- Initially, the game grid was covered with custom buttons to manage both left and right click events, enabling gameplay through these buttons. However, this approach led to disruptions in the user interface

(UI) component of the game. Consequently, it was found to be more practical and comprehensible to overload the `mousePressEvent` function to handle these events, providing a more effective solution.

- Ensuring that a cell is revealed upon the second click of the hint button proved to be challenging. This issue was addressed by using a boolean method named isHint(), which tracked the state of the cell. By implementing this method, it was possible to reveal the cell upon its second click when marked as a hint.
- Adjusting the distance between the grid and the container was somewhat challenging. This issue was resolved by utilizing various functions of QT's default UI API. The functions used were as follows:

```
setSpacing
setColumnStretch
setRowStretch
setColumnMinimumWidth
setRowMinimumHeight
setContentsMargins
```

- We did not comply with C++ formatting best practices from the getgo. This had only ever slowed down the process of development. Another problem was with not splitting the code into meaningful modules. We had focused solely on the functionality, ignoring -or rather lacking- the discipline needed.
- Considerable effort was dedicated to refactoring out unused code, a necessary step due to having explored different approaches for certain functionalities. These functionalities included handling right-click events, displaying the hint button, and managing cell boolean variables. By removing redundant code, the codebase was streamlined, which not only improved readability and maintainability but also enhanced overall performance and reduced potential sources of bugs.

Conclusion

Overall, this project helped us not only using C++ but also being able adapt 3rd party libraries without a prior knowledge. Apparently, a significant portion of the QT features were unintuivite, especially event handling and ui styling.

It also played an important role on understanding the tradeoffs of algorithms such as speed & accuracy, thoroughness & ease of implementation. All these compensation were visible while we were implementing the hint algorithm.

Contributors

- Eray YÜKLÜ - 2021200273
- Ümit Can EVLEKSİZ - 2020200114