MINESWEEPER

Milestone 01

2805ICT

Carl Humphries
S5084150

Requirements

1. Functional requirements

1. Game Basics and Rules

The focus of the program is a grid layout of cells. These cells contain many properties being:

- Is the cell covered?
- Does the cell contain a bomb?
- How many neighbouring bombs are there?

All cells have neighbours, these are cells that share either a corner or an edge.

The program needs to simulate minesweeper, to do that the following requirements on rules are set. For a play to win all cells containing bombs must be marked with a flag, and all the flags placed must cover a bomb and nothing else. That is there are equal bombs to flags and all bombs are covered. To lose the game the user would have to left click on a cell containing a bomb that isn't covered. This immediately ends the game with a game over message.

2. Left Click Action

The program is very step based, meaning that for the game to progress it needs user input. The main action the user will take is left clicking. Left clicking a cell has different outcomes depending on what is under it, the cases are as follows- If the cell;

- is flagged do nothing;
- contains a bomb the game is over
- has 1 or many bombs neighbouring, display that number on the cell
- has no neighbouring bombs check if neighbours have been revealed; if they have not recursively reveal them following the same rules.

3. Right Click Action

Another main action for the game is right clicking. Right clicking will check if the cell is not revealed and if so will place a flag over it. This means the user thinks there is a bomb under it and will lock that cell from being revealed. This is crucial in how the game is won, because all bombs need to be marked with flags (see requirement 1). Right clicking a cell that already has a flag will remove the flag unlocking the cell to be revealed.

4. Game Variants

The program needs to support multiple variations on the minesweeper game. These include;

- using hexagons instead of square cells
- using colour patterns instead of bombs

These variants need to be supported in the final program as extensions of the game not new games entirely.

5. Timer and Scoring

While the main win or lose rules are defined in requirement 1, the user should also receive a score based on the time it took and the difficulty of the level. This score is to provide reusability of the program and help keep the user playing the game. The score will be based on pre-set levels and score will only be kept if they play those levels.

2. Non-Functional requirements

1. Speed

While the game is not inherently processor heavy, the recursive reveal of the cells should not take long at all and should be instant as far as the user sees. This should be done by not revealing something that already has been revealed, or something that does not need to be checked.

Carl Humphries
S5084150

2. Customisability

The user should be able to not only play pre-defined levels but to also customise their own x by y sized board. They should be able to define how big the board is and how many bombs it will contain. This will greatly improve reusability of the program and lead to a better game.

3. High scores

While the user can see their score after each game they should be put on the high scores if they do well. This should be stored in a small lightweight database (SQLite might be used), which they can view from the game menu. Only levels that have been pre-defined should store high scores otherwise it will make for unfair score or messy data storage.

4. Visually pleasing

While the game is quite basic in visual design it is necessary to make sure the user doesn't strain their eyes or not feel comfortable playing. This can be done by using colours that are a bit more pleasing on the eyes. The use of modern colour palettes may be useful to give the game a more up to date feel.

5. Colour coding

While the cells will display the number of neighbouring bombs, it is also important to make sure it is clear which areas contain more bombs. This can be done by colour coding the numbers using calmer colours for low numbers and more dangerous colours for high ones (example blue to red). This will not only look nicer but will help the user understand the game better and learn patterns.

3. Constraints

1. Works on latest version of Linux

The program needs to work on the latest version of Linux, while it was not specified exactly what OS (what version of Linux) it should work on most of them since Linux systems are similar.

2. Multiplatform

The program is required to work on at least two platforms and will be made for Windows and MacOS (should support Linux too). The foundation will be made with python due to its cross-platform support. And with the use of default packages bundled with the newest version of python, the system (given they have python) should be able to run the program.

3. Code reusability

The code needs to be reused on both systems not made independently. This is the main reason for using python since the code will virtually stay the same across both systems. Also, the main section of the code should be made into classes to enable reuse of the code in later projects or tasks.

Use Case Specification 1.2

Brief Description

This use case is used to determine what happens when an actor triggers a left click event on an uncovered cell. This case is only executed if the cell is covered. This use case can be triggered from itself recursively.

Actors

- Player (playing the game)
- Game

Entry Condition

- Cell needs to be covered.
- Player needs to left click the cell or celled recursively called.
- Cell needs to not be flagged.

Exit Condition

- More gameboard is revealed. Until no more can be revealed.

Or

- Game is over, and player has lost.

Basic Flow

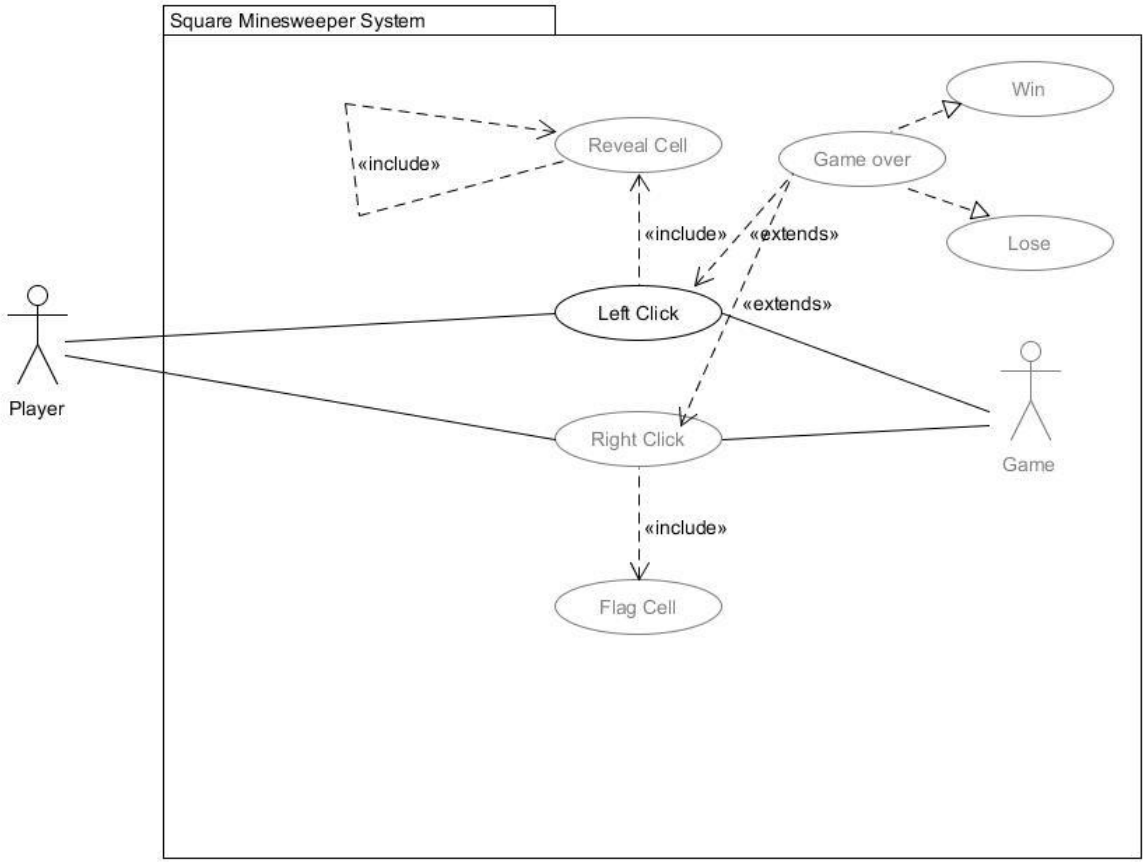
1. The use case starts when the player left clicks a cell that is covered and not flagged.
2. The cell is then checked to see if it contains a bomb; if there is a bomb go to alternative flow
3. Reveal the cell
4. The number of neighbouring bombs is checked;
    a. if there are no bombs surrounding the cell, recursively call this case on neighbouring cells
    b. if there is one neighbouring bomb change cell display number to 1 and set cell colour to blue.
    c. If there are two neighbouring bombs change cell display number to 2 and set cell colour to yellow
    d. If there are three neighbouring bombs change cell display number to 3 and set cell colour to red
    e. If there are more than three neighbouring bombs set colour to dark red and set cell display number to number of neighbouring bombs.
5. Use case ends

Alternative flow

1. If the cell contains a bomb the game is over, and the player loses.

Use Case Diagram

This diagram illustrates interactions from the player to the gameboard. This is the most important interaction because it is where the game can be played.



Carl Humphries
S5084150

Project risk

The first thing that was done was to outline and get a rough estimate on what is needed for the final design. This was to avoid redesigning core concepts once the project needs something new (like hexagons). Once a rough idea of what was needed, it was then segmented into smaller things. An example of this is noticing that a game consists of a board, which consists or many cells. This allowed the development of the board class and the cell class. Using these made for easier tweaking and independent workflows.

Once the classes were designed the work was split into smaller milestones, rough outline on what has been done and what needs to be done;

- CLI game
    - Implement the classes
    - Generate a valid board
    - Fill in the neighbouring data
    - Start applying reveal function (until one shows good results)
- GUI game (single app) Prototype (task 1)
    - Apply CLI game to square based GUI
    - Allow cells to be clicked
    - Change how the cells look once they are revealed
    - Debug reveal function
    - Add ability to place flags
    - Add win and lose conditions
- Build Hex game (task 2)
    - Apply square based logic to hex game
    - Change how neighbouring works
    - Update cell click events (to work with hexagons)
    - Debug reveal function alter to work with hex
- Build multigame program
    - Add both hex and square based boards into one program
    - Alter code to be more reusable
    - Add menu and ability to play either game type
    - Add scoring and high scores
    - Redesign how the game looks (first stage)
    - Debug any issues with the game
    - Allow custom levels and a level selector
- Update to include colour-based minesweeper (task 3)
    - Design how the neighbouring will work
    - Update the reveal function and game functionality
    - Debug these changes in its own program before applying to main game
    - Iron out bugs
    - Update UI to be closer to final design
    - Polish game

This workflow is to ensure each milestone is independent from the main program and is easily applied into when it is completed. While it is a rough outline on what needs to be done it is flexible enough to change if needed. It also allows the visualisation of progress through actual prototype and working versions. Each main phase is independent enough to run on its own allow for it to be polished before merging.

Feasibility Report

Minesweeper is a widely successful game consisting on such a simple concept. It is because of this simplicity it became so popular. It was widely available on old windows PC's and was something almost anyone could play. The project was tasked to create a Minesweeper like game but add a bit more to it.

The program needs to keep the same aspects of minesweeper but add two other game variants (requirement 1.4). These consisted of Hexagonal cells instead of square and a colour pattern variant. The project will be split into 4 main phases; development of original minesweeper, development of hexagonal minesweeper, development of colour-based minesweeper and merging into a central game.

The project is hopefully going to add more enjoyment to the original minesweeper. While incorporating the aspects that made the original game so fun. The project will be made using python 3.6 allowing it to be cross compatible for Windows and Unix (requirements 3.1 and 3.2).

The game targets younger ages allowing for the development of pattern following and problem solving (requirements 2.5 and 1.1). It also will be visually pleasing allowing for more replay ability and overall be more enjoyable for everyone (requirement 2.4).

The program will be developed in stages first using CLI input and output, then porting it over to a GUI based program. Each of the game variants will be fully working before they are merged into one program to allow for reuse of the games and workflow to be split up into smaller milestones.

Prototype

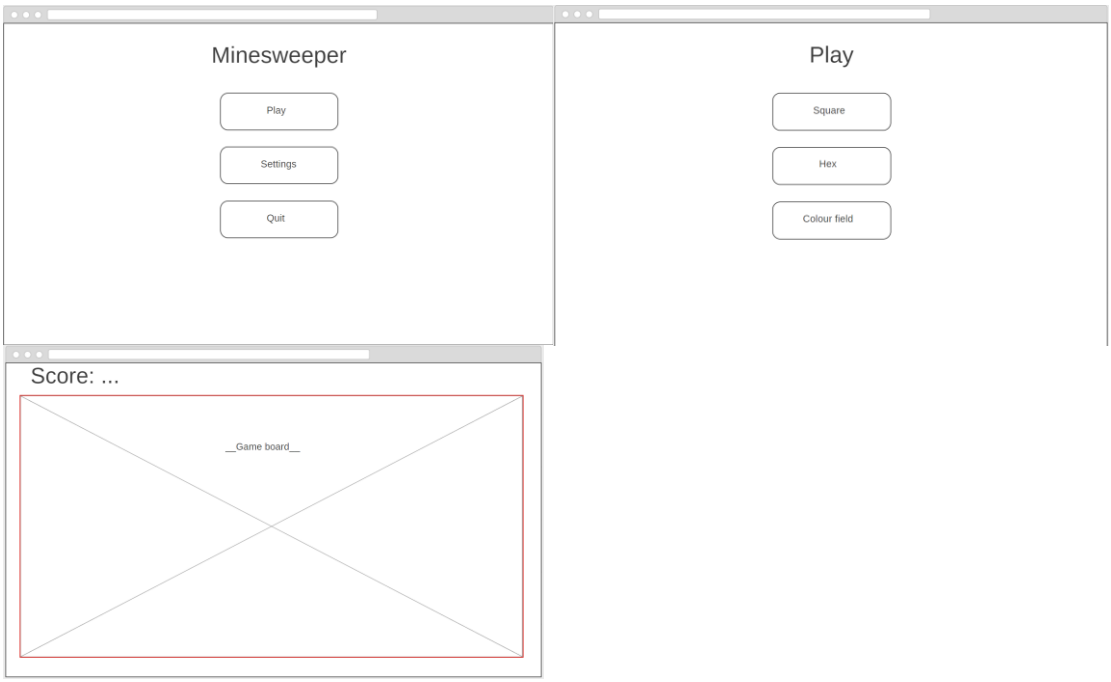
Download link (exp 1st sep 2018): https://ufile.io/zkrei

Youtube: https://youtu.be/M5UAreDsxWE
Note: there is a win case, but it isn't shown because the video needed to be short.

Conceptual Design

GUI design

The GUI should be quite simple allowing for easy navigation and a draft for the screen has been created as a starting point and is subject to change.



The idea is to use the classic old minesweeper style but include modern colouring and try to modernise the game layout. The final screen is where the user will interact with the game itself.

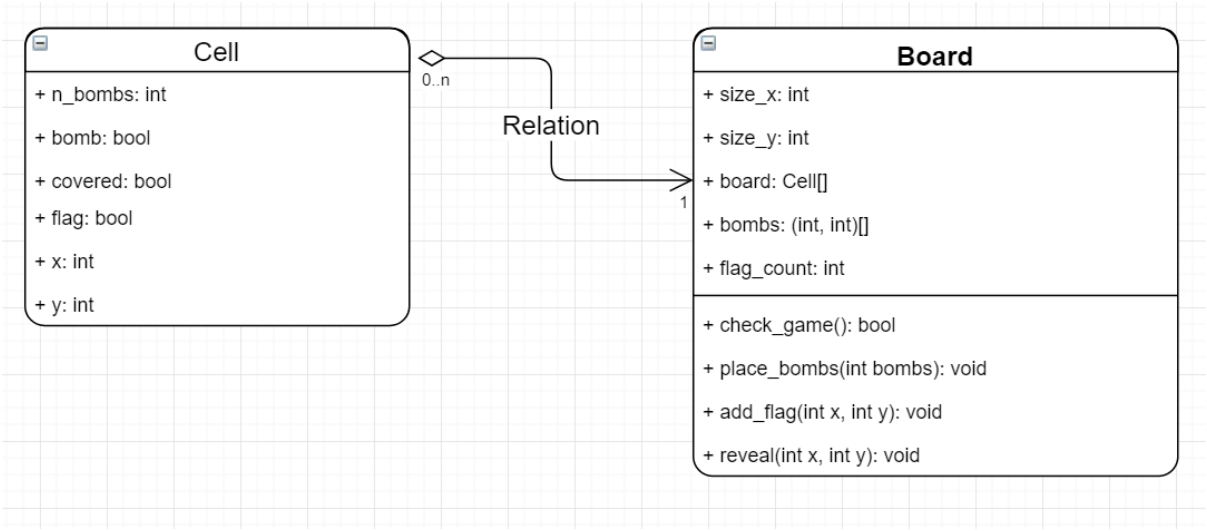
Carl Humphries
S5084150

Use cases cross-reference

The game board will house most of the functionality specified in requirements 1.1, 1.2 and 1.3. The Use Case diagram illustrates how the user interacts with the game board and the game itself. If the player left clicks the game will result in either a revealing cells or loss. If the player right clicks it will result in a flag being placed or a win.

Class Design

As mentioned before under “project risk” the classes that came to mind were one for the board and one for the cells. Below is the UML for cell and board as well as a relation showing how the board uses cell:



Note: Because of python lacking private variables all members are left “+” (public), this would not be the case if another programming language was used. But it was initially left public because of the similarity to the actual prototype. Also because of the choice of python doxygen wasn’t used to document classes. It was difficult to find a way to generate it through doxygen so the UML was done manually.