Table of Contents

[Displaying the data 2](#_Toc39765140)

[GUI 2](#_Toc39765141)

[Data fetching 3](#_Toc39765142)

[GUI updates 3](#_Toc39765143)

[Losing 4](#_Toc39765144)

[Winning 4](#_Toc39765145)

[Editing the data 5](#_Toc39765146)

[“Stepping” a tile 5](#_Toc39765147)

[Marking a tile 5](#_Toc39765148)

[Creating the grid 5](#_Toc39765149)

[Optional Extras 6](#_Toc39765150)

[Custom Game Parameters 6](#_Toc39765151)

[Spritesheet 6](#_Toc39765152)

[Sounds 6](#_Toc39765153)

# Displaying the data

## GUI

When the program is started, the MinesweeperGUI class initializes a JavaFX environment.

It configures the Stage, setting its title, setting it as resizable, and then creates some additional components, such as the menu bar (shown right), for laying out the game & its controls.

The game is initialized in the newGame function, which takes rows, columns and max mines parameters. This is fed into the Minesweeper.newGame function (a part of the main Minesweeper class created in Part 1 of this assignment) which then initializes the game’s core under the hood for the GUI to interface with.

In the newGame function, I call setupGrid, which creates the JavaFX components required to show the Minesweeper grid on the screen.

In setupGrid, I firstly check whether a previous game’s grid exists and destroy it if so. Then, I create a GridPane, which is the perfect component for a grid-based game, allowing me to define the number of rows, columns and the width and height of these rows and columns. I base the default tile size as 24px, and the window is resized to fit exactly to fit the size of the grid.

Notably, the window is resizable, and the GridPane will resize accordingly to accommodate the player’s requested size. The player cannot resize the window to be any smaller than the default grid size. I must make a technical note here and state that due to two bugs in JavaFX, calling setWidth and setHeight does not actually set the window’s size correctly. In fact, there is a constant offset of -16px for the window width, and -38px for the window height, meaning I must compensate for this by adding 16px and 38px to the width and height respectively. I tested whether this was an OS-specific problem but my macOS test had exactly the same issue.

Each tile on the grid is generated one-by-one based on the game’s rows and columns parameters. I additionally generate an extra two columns and two rows to represent the game’s border. Each tile listens to mouse click events (left clicks and right clicks) which updates the GUI and edits the data accordingly, which I will elaborate on further into this writeup.

Every tile is an object instantiated from the GridTile class. This class takes a parameter of either a TILE enumeration, or a MineTile object created in the main Minesweeper class (in Part 1 of the assignment.) This parameter determines the image that the tile displays to the player on the GUI. The TILE enumeration is as follows:

public static enum TILE {

TILE,

REVEALED,

MARKED,

BOMB,

BOUNDARY

}

Each TILE has an image associated with it, which you will notice is represented by a Minecraft block:

|  |  |  |
| --- | --- | --- |
| **Image** | **Enum** | **Description** |
|  | TILE | An unmarked, unrevealed tile, represented by a Minecraft Grass block. |
|  | REVEALED | An unmarked, revealed tile, represented by a Minecraft Dirt block. This signifies to the player that the tile has been revealed; the grass has been “removed” to expose what’s underneath. |
|  | MARKED | A marked tile, represented by a Minecraft Pumpkin block. I think the facial expression on the pumpkin represents the facial expression of someone stepping on a land mine. |
|  | BOMB | A land mine tile, represented by a Minecraft TNT block. This is never shown to the player unless they have revealed a land mine tile and therefore have lost the game. |
|  | BOUNDARY | The boundary/border, which does nothing and cannot be interacted with, represented by a Minecraft Bedrock block, which cannot be destroyed, interacted with or acquired by the player in Minecraft. |

Lastly, a revealed GridTile will show the number of neighbouring mines to the player using a JavaFX Label, like shown left.

Left: An image of the generated Minesweeper window, grid and tiles, showing the menu bar, revealed tiles, unrevealed tiles, the boundary/border, and a marked tile.

## Data fetching

Every GridTile, except for those representing the boundary/border, internally store a corresponding Minesweeper.Minefield.MineTile (created in Part 1) object which the GridTile represents on the GUI.

Each tile has an update function, which queries the MineTile it represents for the following data:

* Whether the MineTile has been marked, using MineTile.isMarked()
* If the MineTile has not been marked, whether the MineTile has been revealed, using MineTile.isRevealed()
* And finally, if the MineTile has been revealed, not marked, the number of mine neighbours, using MineTile.getMineNeighbours()

## GUI updates

The GUI does not asynchronously “update” every frame per-say, as this would be a very inefficient method of updating the UI.

Because the game has no background processes (such as AI) it only needs to respond based on the player’s actions.

As previously stated, each tile on the grid listens to the player’s mouse events: what tile was clicked, and whether the click was a left click or a right click.

A left click causes the tile to be revealed. This calls the Minesweeper.Minefield.step (created in Part 1) function specifying which tile was clicked on, which then progressively reveals neighbouring un-mined tiles. When the step function is finished, it will return true if the game is still ongoing, or false if the player has just stepped on a mine and has lost the game (see the “[Losing](#_Losing)” section.) Regardless of what the function returns, the game iterates over every tile on the grid and calls its update function, which updates the image each GridTile shows to the player. Finally, if step returned false, the game informs the player they’ve lost (see the “[Losing](#_Losing)” section), otherwise, the game then calls Minesweeper.Minefield.areAllMinesRevealed(), which will return true if all un-mined tiles are revealed and all unrevealed mines are marked, which then informs the player they’ve won (see the “[Winning](#_Winning)” section), ending the game.

A right click toggles whether the tile is marked or unmarked, which also calls Minesweeper.Minefield.areAllMinesRevealed() and begins the game win process if applicable.

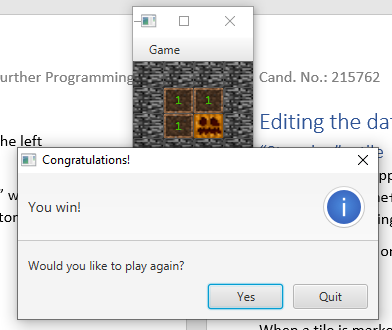
The GridTiles are updated via their GridTile.update() function, which, using the data described in the “[Data Fetching](#_Data_fetching)” section, updates its image based on the state of the MineTile object it represents.

## Losing

When the game has been lost, it will present the top-left commiseration dialogue.

“Quit” will close the entire program, and “Yes” will prompt the player for custom game parameters (see “[Custom Game Parameters](#_Custom_Game_Parameters).”)

## Winning

When the game has been won, it will present the bottom-left congratulations dialogue. Like with “[Losing](#_Losing)”, “Quit” will close the entire program, and “Yes” will prompt the player for custom game parameters (see “[Custom Game Parameters](#_Custom_Game_Parameters).”)

# Editing the data

## “Stepping” a tile

When a tile is “stepped on” by the player (left clicked; see the “[GUI updates](#_GUI_updates)” section), the Minesweeper.Minefield.step function is called, specifying which tile was stepped on, which then progressively reveals neighbouring un-mined tiles under the hood inside the Minefield object.

If the player steps on a land mine, the step function returns false which means the game has been lost, see the “[Losing](#_Losing)” section.

## Marking a tile

When a tile is marked by the player (right clicked; see the “[GUI updates](#_GUI_updates)” section), the Minesweeper.Minefield.markTile function is called, specifying which tile was marked/unmarked by the player, which updates the marked state of the tile under the hood inside the Minefield object.

## Creating the grid

The player can actually specify custom game parameters to create custom games. See the “[Custom Game Parameters](#_Custom_Game_Parameters)” section.

# Optional Extras

## Custom Game Parameters

Shown left, the game allows for the player to specify custom game parameters. This dialogue is shown when the player confirms they want to play another game after winning or losing, and when the player selects Game -> New game… in the menu bar.

Notably, the game will not accept any input from the player that is 0 or under or is not a valid integer. If the player does not enter any values the game will use the default ones provided and shown left.

## Spritesheet

The Spritesheet is a simple class that pre-loads and fetches the images for each tile.

Each GridTile will call the Spritesheet.getTile function, which returns an Image object representing the corresponding tile image to the TILE enumeration of the GridTile. Then, the GridTile, which contains an ImageView object, updates the ImageView using the returned Image object.

This is an optimized method of updating the images; instead of constantly loading the image from the disk, simply load it once, store it in memory and then reference back to that every time it’s needed.

## Sounds

The game also includes some sounds which are played from various events.

Each sound is an MP3 file, and there can be multiple sounds for each event, which are picked randomly when played. The GameSounds class controls the playback of these sounds, and alike to the Spritesheet class, pre-loads each sound into memory for when they are needed. It also stops any other sounds from playing simultaneously to another.

Each sound is described below.

|  |  |  |
| --- | --- | --- |
| **Event** | **File name** | **Description** |
| Tile revealed | reveal[1..4].mp3 | Played when a tile has been revealed (but not when any neighbouring tiles are also revealed as a result), it is the sound of the shovel in Minecraft digging up grass. |
| Tile marked | mark[1..3].mp3 | Played when a tile has been marked by the player, it is the sound of the “Villager” in Minecraft, representing the sound of someone thinking. |
| Tile unmarked | unmark[1..4].mp3 | Played when a tile has been unmarked by the player, it is the sound of when dirt has been interacted with in Minecraft, representing a flag being taken out of the dirt below. |
| Game lost/Mine revealed | explode[1..4].mp3 | Played when the game has been lost by the player revealing a land mine below them, it is the sound of a TNT explosion in Minecraft. |
| Game won | win1.mp3  win2.mp3 | Played when the game has been won by the player, this is actually two separate sounds taken from Minecraft – the first is the sound of a firework flying into the sky, and the second is the sound of firecrackers popping, representing a firework explosion to celebrate the player’s win. |