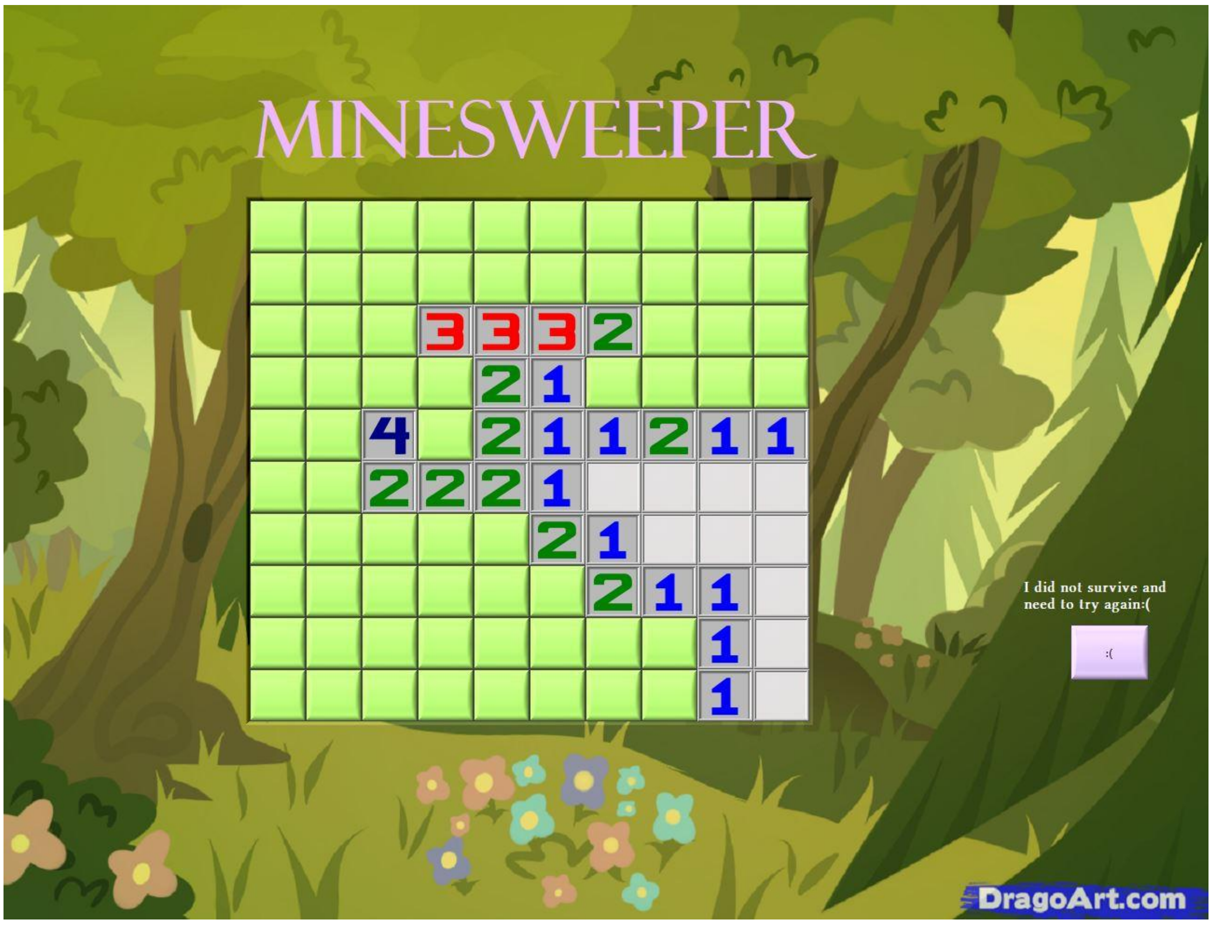
TFY4190 Instrumentation

# Making Minesweeper in LabVIEW

Group members: Ingrid Midtbust Hjelle and Simone Otelie Solstrand



## Abstract

In this project we discuss how the game Minesweeper has been made in LabVIEW by using a state machine. The state machine consists of *Initialize*, which places the mines randomly and completes the game logic where numbers are placed based on number of neighboring mines, then there is *Wait for state*, which takes in an action for the player, and *Game*, which checks if the game is won or lost, and finally *New*, which resets the game and lets the player start again. The game is functioning, and we had a lot of focus on the user experience by including pictures and making an interface which looked like a game. Most of the Minesweeper logic was successfully integrated, however, due to time issues, functionality such as placement of flags and recursion were unfortunately not completed.

## Table of contents

# 1 Introduction

In the subject TFY4190, a project over the course of six weeks is to be completed in LabVIEW. We have chosen to make the Minesweeper game, seeing as we found it to be an interesting and educational challenge. Our goal was to make a functioning Minesweeper game that was as close to the original game as possible. In addition, we wanted to add as many special effects as we had the time to.

## 1.1 The Minesweeper game

When starting a game of Minesweeper, the player is faced with a grid containing tiles that can be pressed. Behind the tiles there are hidden a certain amount of mines, and if you press one, you have lost. The goal of the game is to press all tiles that do not contain mines as fast as possible.

If you press a tile that is not a mine, two things can happen. If the tile has one or more mines in the neighboring tiles, the number of neighbor mines will appear on the tile. This number ranges between 1 and 8, where 8 is very unlikely. If the tile has no neighboring mines, it will open, and then check for mines in all the neighbors of all of the neighboring tiles. This happens until a tile containing a number is opened on all sides of the original tile. An example of how this looks after the first tile has been pressed is given in Figure 1.

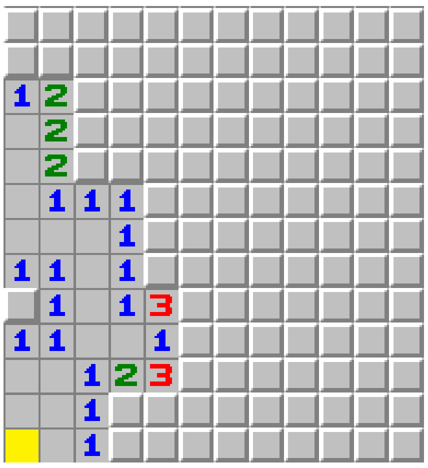


**Figure 1:** An example of how the board can look after the first tile is pressed. [1]

Here you can see how in total 38 tiles opened by this method even if the player only pressed one. Additionally, this picture also shows an example of tiles that are clearly mines. In order to mark this the player is given the option to place a flag at that tile instead of opening it. Then, if the player knows that there are for example 20 mines on the board, the player will know that the game is finished once 20 flags are placed.

[1] Taken from: http://datagenetics.com/blog/june12012/index.html

# 2 Method

In order to make the Minesweeper game, it was made a state machine with the following states: *Initialize*, *Wait for event*, *Game* and *New*. The purpose of these states and how they were implemented are explained in section 2.1.

Before the state machine is entered, the grid is initialized. It was found that the most effective way to make the grid was to have two separate grids and then place them on top of each other. The underlying grid, which we refer to as the *Game grid,* contains the logic of the game and is explained in section 2.1.1. On top of this grid it was placed a 2D array of boolean variables which we refer to as the *Boolean grid*. When a tile in the *Boolean grid* is opened, and thus takes the value True, the color of the tile will change to transparent, so that the player then can see the corresponding tile in the *Game grid*.

## 2.1 The state machine

### 2.1.1 Initialize

As the name suggests, in the Initialize state we want to initialize the game. The first step is to place the mines. For a 10x10 board we have chosen to have 20 mines. We start by hardcoding the board as a 10x10 array containing 20 mines symbolized by the letter M, and the remaining tiles are filled with the letter T, symbolizing the Norwegian word for empty. This array is then sent into a LabView's shuffle function *Shuffle 2D Array VI*. The result is then a grid in which the mines are at random locations.

The next step is to fill in the numbers by the logic explained in section 1.1. In order to do this we have to loop through all the tiles in the grid, and for each tile, loop through all the neighboring tiles. If the neighboring tile is a mine we take +1 to a constant. This should result in a number between 0 and 8 for each tile. If the resulting number is 0, no change is made to the tile. However, if the number is between 1 and 8, the letter T at the given position is changed to another letter according to Table 1.

**Table 1:** Letters assigned to tiles with 1-8 neighboring mines.

| Number of neighboring mines | Letter assigned |
|---|---|
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | D |
| 5 | E |

| | |
|---|---|
| 6 | F |
| 7 | G |
| 8 | H |

This seemed simple enough, but since Minesweeper does not have periodic boundary conditions (i.e. tiles at the border are connected to tiles on the opposite border), each tile does not have an equal amount of neighboring tiles. In fact, there are three possibilities. These are represented in Figure 2.

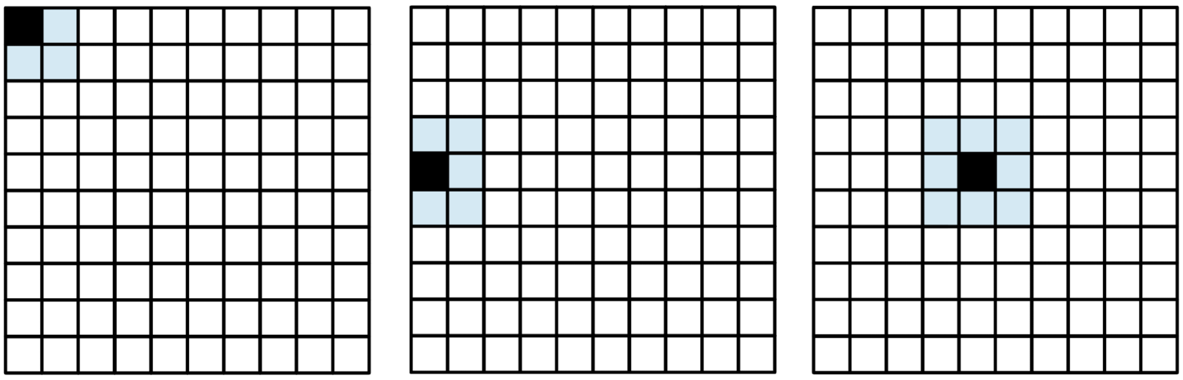


**Figure 2:** Figure showing how a tile can have 3, 5 or 8 neighboring tiles.

This means that only using one case structure to represent all tiles would not suffice as each case must have an equal amount of outputs. We therefore concluded that we needed to have at least three case structures, and which case structure we entered would depend on whether the tile was a corner tile, a border tile, or neither. The logic of our program is shown in Figure 3.

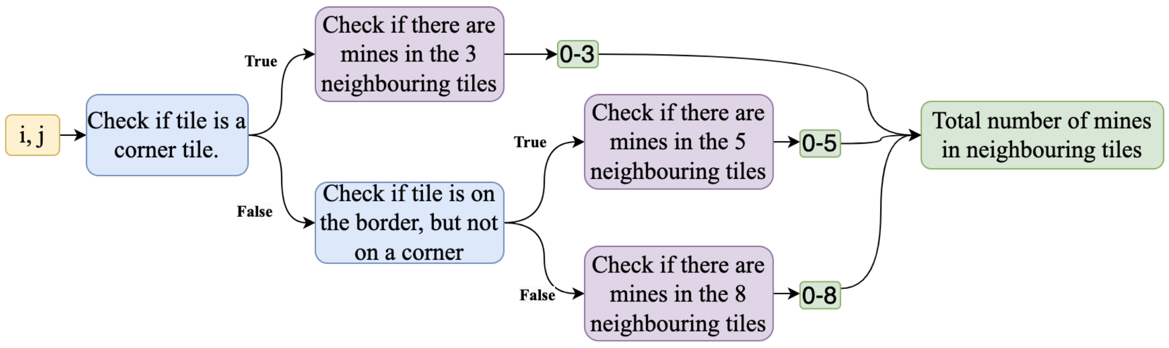

i, j
Check if tile is a corner tile.
True
Check if there are mines in the 3 neighbouring tiles
0-3
False
Check if tile is on the border, but not on a corner
True
Check if there are mines in the 5 neighbouring tiles
0-5
False
Check if there are mines in the 8 neighbouring tiles
0-8
Total number of mines in neighbouring tiles

**Figure 3:** Schematic of program structure in state Initialize. The blue boxes check which of the three cases are to be entered using boolean operators AND and OR. The purple boxes represent the three case structures included in our code, and finally, the green boxes represent the possible integer result from the different cases.

The indices i and j are incremented by two for-loops. In order to find the indices of neighboring tiles, these have to be incremented or lowered by 1 in 3, 5 or 8 different ways. The logic of how this was done is presented by the tables in appendix 2.

When the program was completed, each tile had a number ranging from 0 to 8 which represented how many of its neighboring tiles were mines. The tiles were then assigned a letter according to Table 1. However, we did not want a game where the player is only looking at letters. Therefore, pictures of numbers and mines were found, and an additional loop was added in which each letter got an assigned picture. The original array still contains letters, but it is now created a new 10x10 array which contains pictures so that the player gets a better experience playing the game. The 10 pictures used can be seen in Table 2.

**Table 2:** The pictures used to represent the letters in the original array. [2]

| | | | |
|---|---|---|---|
| | M | | D |
| | T | | E |
| | A | | F |
| | B | | G |
| | C | | H |

In order to place the pictures in the *Game grid* we had the pictures downloaded and used functions that already existed in LabView to read from the files and add them to the grid. In order to read from the file we used *Current VI's Path Function* to get the path file of the VI in

[2] The pictures are taken from the following sources.
Mine: https://www.shutterstock.com/image-vector/cartoon-comic-style-nuclear-mushroom-cloud-629206145
Open tile: https://commons.wikimedia.org/wiki/File:Minesweeper_0.svg
1: https://commons.wikimedia.org/wiki/File:Minesweeper_1.svg
2: https://commons.wikimedia.org/wiki/File:Minesweeper_2.svg
3: https://commons.wikimedia.org/wiki/File:Minesweeper_3.svg
4: https://commons.wikimedia.org/wiki/File:Minesweeper_4.svg
5: https://commons.wikimedia.org/wiki/File:Minesweeper_5.svg
6: https://commons.wikimedia.org/wiki/File:Minesweeper_6.svg
7: https://commons.wikimedia.org/wiki/File:Minesweeper_7.svg
8: https://commons.wikimedia.org/wiki/File:Minesweeper_8.svg

which we were working, and sent that path file along with the picture path file into the *Build Path Function*. Here, an appended path is created. Then *Read PNG File VI* was used to read the path, and *Draw Flattened Pixmap VI* was used to draw the picture as a RGB pixmap so that it had the right format to be allowed to be stored in the grid.

The full schematic of *Initialize* in LabView can be found in appendix 1.

### 2.1.2 Wait for event

The second state in the state machine is called *Wait for event*. This part of the state machine makes the game wait until the player does something that requires a new response from the game. This is done by using a structure in LabVIEW called *Event Structure*. This structure contains code which tells the program what to do when a specific event happens. The events contained in our event structure are *"I did not survive and need to try again:( : Value change"* and "*Array: Value change*". These are referred to as event 0 and event 1.

Event 0 is triggered when the player clicks on the button *I did not survive and need to try again :(*. The code then puts the game into the last case, *New*. The other event, event 1, is triggered when the player opens a tile. The code then sends the game into the third state,, *Game*.

The full LabVIEW schematic of *Wait for event* can be viewed in appendix 3.

### 2.1.3 Game

The third state in the state machine is called *Game.* In this part, the code checks if the player has opened a tile with a mine underneath. Every time the player opens a tile, the code checks if the element underneath is a mine. This is done by iterating through both of the grids that the player is interacting with. The player has opened a mine if an element in the *boolean grid* is True and the element with the same indices in the *game grid* is a mine. If the player has opened a mine a message is displayed to the user that the game is lost.

In addition, the code counts the number of opened tiles. If the player has opened 80 tiles without yet opening a mine, the 20 tiles remaining must all contain mines. Then another message is displayed to the user that the game has been won.

This state is sent back to the *Wait for event* state, and this therefore creates a loop as long as a tile the game is being played. The LabVIEW schematic of this can be viewed in appendix 4**.**

### 2.1.4 New

When event 0 is triggered, the game is put into the state *New.* Here the game is reset so that the player can start a new game. This is done by making all the elements in the *boolean grid* False, which corresponds to closed tiles. In addition, the *Game grid* is reset so that it only contains mines and empty tiles. . This state is then sent to the *Initialize* state, so that the *Game grid* can be initialized again. The LabVIEW schematics of this can be viewed in appendix 5**.**

## 2.2 User interface

The program ended up being much bigger than expected, and the Block Diagram is large and complicated. Luckily, the player of the game is only to see the Front Panel, and not the beautiful mess which hides behind it. We therefore put a lot of effort into making the application look like a game. As explained in the beginning of this section we stacked two grids on top of each other in order to create the illusion of a picture hiding behind a tile that is opened. The player only interacts with the *Boolean grid*. In addition we implemented actual pictures instead of using the letters which we began with.

There is no button the player can press which officially starts the game. Start is when the player presses the first tile. To create a better playing experience we wrote a message that welcomes the player to the game. This message appears on the screen when the game starts, and when the button called *I did not survive and need to try again :(* is pressed. As explained in section 2.1.2, pressing this button will also reset the board so that none of the tiles are pressed and the *Game grid* is reset so that all tiles are closed.

Finally, in order to make the game more presentable, a suitable background was found and put behind all the grids and the button. A title was also added on top. Colors and themes were decided in order to make a calm environment to juxtaposition the otherwise stressful game.

# 3 Results

The Minesweeper game is functioning, and is displayed in the figures below. The game contains mines, a reset button, numbers describing the numbers of neighboring mines and messages displayed to the player at fitting times. The figures show both the *Game grid* and the *Boolean grid* as well as the messages displayed to the player during the game.

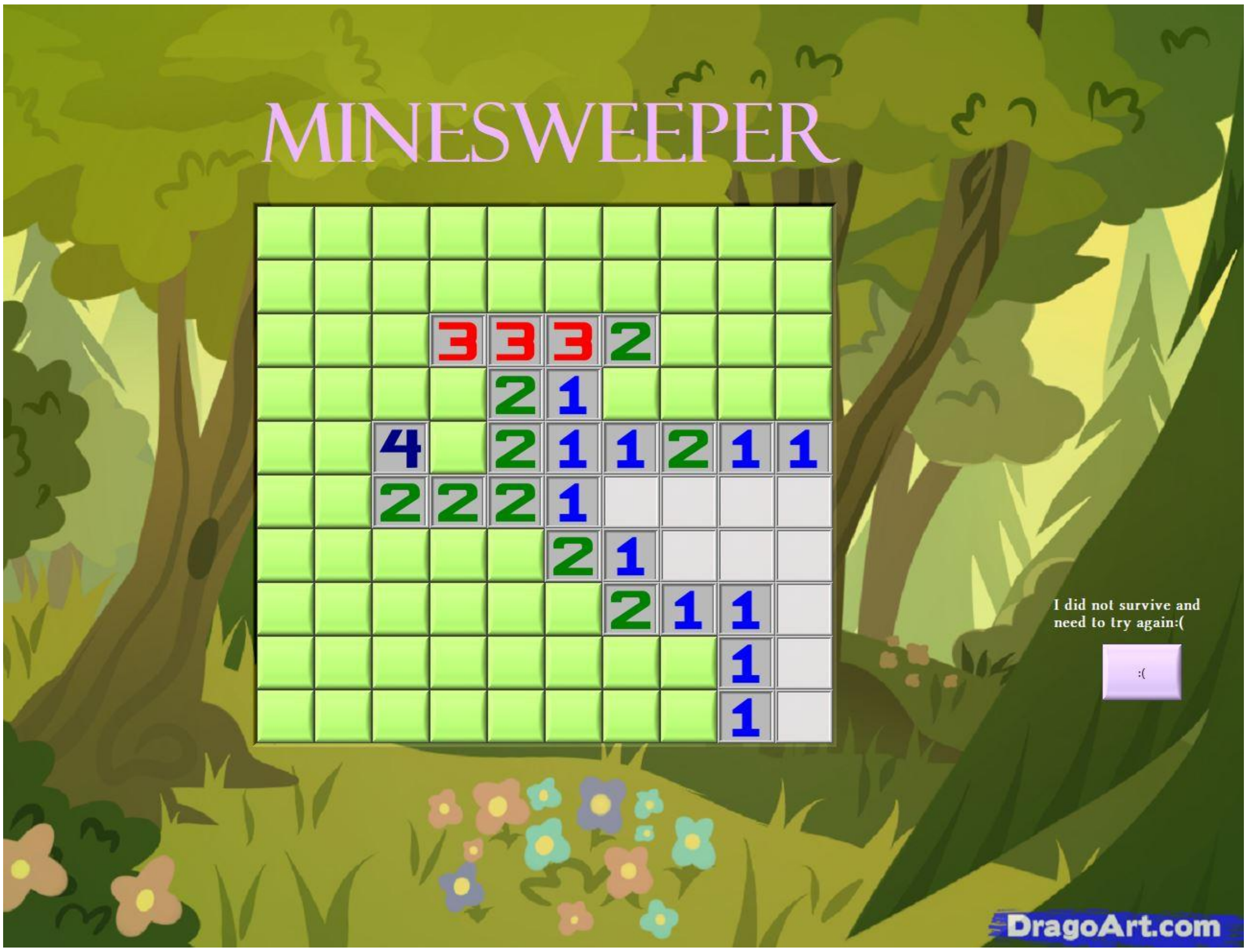


**Figure 4:** The user interface of the game. There is a pink title, "MINESWEEPER", over the grid. One can see that some of the tiles are opened so that it is possible to see some of the pictures that the *Game grid* consists of. In addition, one can see the background we placed behind the grid. [3] The button which resets the grid is to the left with a pink hue.

[3] Background of GUI taken from https://dragoart.com/tut/how-to-draw-forests-forest-backgrounds-13031.

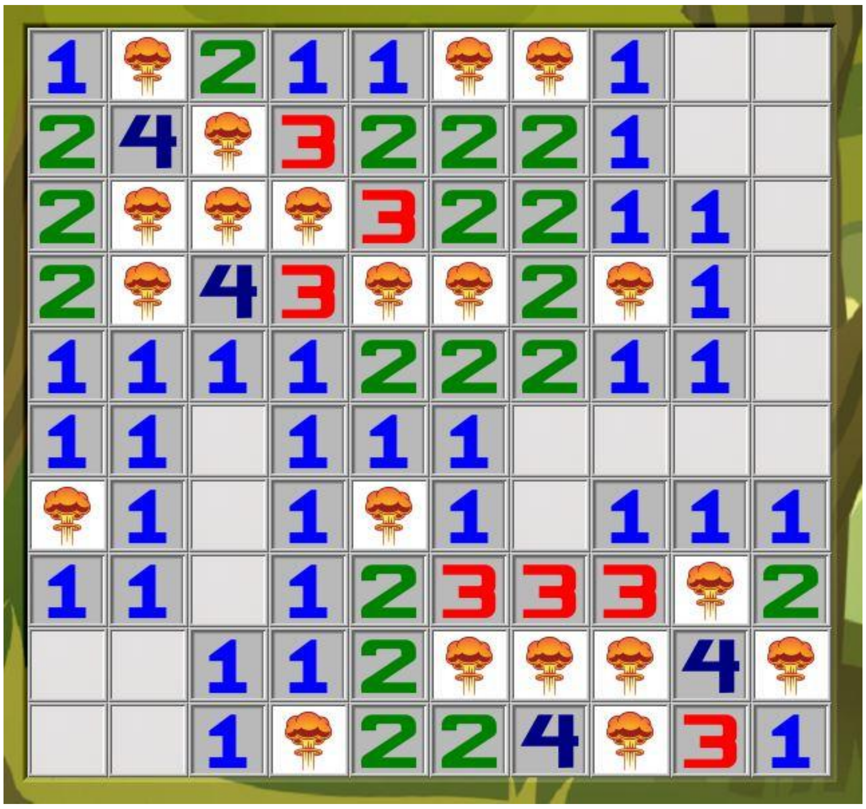
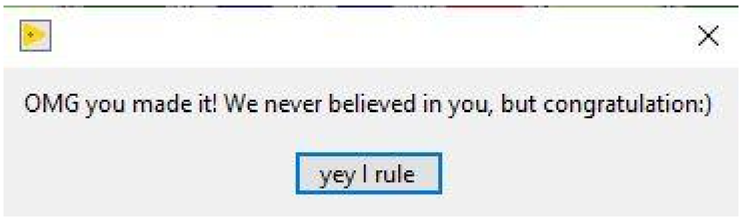


**Figure 5**: In this figure one can see that the mines are shuffled around. Furthermore, the numbers describing the numbers of neighboring mines are correct. This shows that the Initialize case in the game works.

The messages in the figures below, which are to be displayed to the player, were thought through carefully with respect to the players motivation and overall experience of the game.



OMG you made it! We never believed in you, but congratulation:)

**Figure 6:** The figure contains the message displayed to the user when managing to open all the tiles that do not hide a mine. Furthermore, one can see the button the player can push after having read the message.

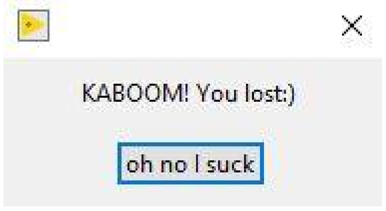
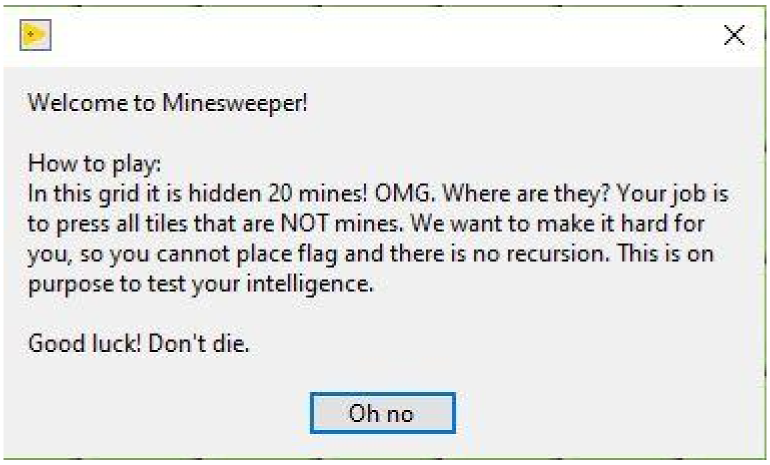

KABOOM! You lost:)

**Figure 7:** The figure shows the message displayed to the user when opening a mine, and losing the game. In addition, one can see the button that the player can click after having read the message.


Welcome to Minesweeper!

How to play:
In this grid it is hidden 20 mines! OMG. Where are they? Your job is to press all tiles that are NOT mines. We want to make it hard for you, so you cannot place flag and there is no recursion. This is on purpose to test your intelligence.

Good luck! Don't die.

**Figure 8:** This is the message displayed to the user when starting the game. The message explains how the game is played. In addition, there is a button that can be pushed when the player is done reading.

## 4 Discussion

There were several unforeseen problems while working on the project. Perhaps the most time-consuming problem was LabVIEW itself. There were several times where the logic had to be changed, or we had to start all over again, because the functionality needed did not exist in LabVIEW. This was especially challenging when initializing the grid. We had to find multiple loops to get around our problems, and as a result our program took up a huge amount of space in the block diagram. Unfortunately, LabView has no zoom function, and therefore it became hard to work with and hard to debug the code. A possible solution to this is to create more VI's and have several files instead of having everything in one file as we did. This is something we would do if we had more time. These problems forced us to learn LabVIEW properly, so ultimately we appreciate them.

A lot of time was spent on getting *Initialize* to work. The state machine had been made beforehand, but since *Initialize* took so long we had a discussion if we should remove it as there was little time to add buttons and more functionality. The alternative would have been to create for example a case structure in a while loop, where the condition of the while loop decided whether the game was being played or not. Ultimately we decided to keep the state machine as what we initially thought would be simpler turned out to be more work. In hindsight this was a good choice as we at the end had time to add several small functionalities easily in the state machine.

Due to time limitations our program has some bugs that we were not able to fix. These are not bugs due to mistakes, but due to not having time to make functionality that would eliminate them. When a player opens a tile behind which it is a mine, the message that the game is lost appears on the screen. However, this incident does not force the game to quit. The natural step for the user is of course to start a new game, but it is also possible to keep playing. Two things can then happen. The user can keep trying to open tiles. Since a tile with a bomb is then already opened, the message which states that the game is lost will keep appearing after each opened tile. Another possibility, unfortunately, is that the player can close the tile containing the bomb. Then the game will go on as normal. Hence, if the player knows about this bug, winning the game becomes trivial. There are two ways to fix this. We can either make the grid so that once a tile is opened, it cannot be closed again. This should be implemented anyways, but is not at this point. Additionally, one could make the code so that once a mine is opened, the game is automatically reset.

Another problem was the recursion for both of the grids, and the placing of flags. The game began to work sometime during our fifth or sixth week of working. At this point,the game did not have a functioning reset button in addition to some bugs. We discussed what was necessary to do in order to implement both the recursion, flags and the reset button. There was not enough time to manage all of it, and we wanted the game to be as fully functioning as possible. Therefore, we prioritized implementing a reset button, messages and more to make the players experience as good as possible. Hence, there was simply not enough time to start implementing both recursion and flags into the game. If we had the time, this would be the natural next step.

# 5 Conclusion

The Minesweeper game is possible to play, and is rather similar to the original Minesweeper game, as was our goal. However, due to a lack of time, there is no recursion in the grid, and it is not possible to place flags either. In conclusion, we are satisfied with our work, and we have created a product which is presentable to a player.

# 6 References

When it comes to the code in this project we have not used any sources. All the logic presented in this report is completely made by us and does not come from a separate source. We have however taken pictures from various places. Since pictures do not have an author in the same way as a web page, we decided to present the sources of the pictures in footnotes throughout the text as the pictures were presented. Thus, no references will be listed here.

## 7 Acknowledgments

We found this project quite challenging, and would not have been able to come as far as we have without the help of our lab coordinator Lise Kvalø.

# 8 Appendices

## Appendix 1 - LabView schematic of Initialize state

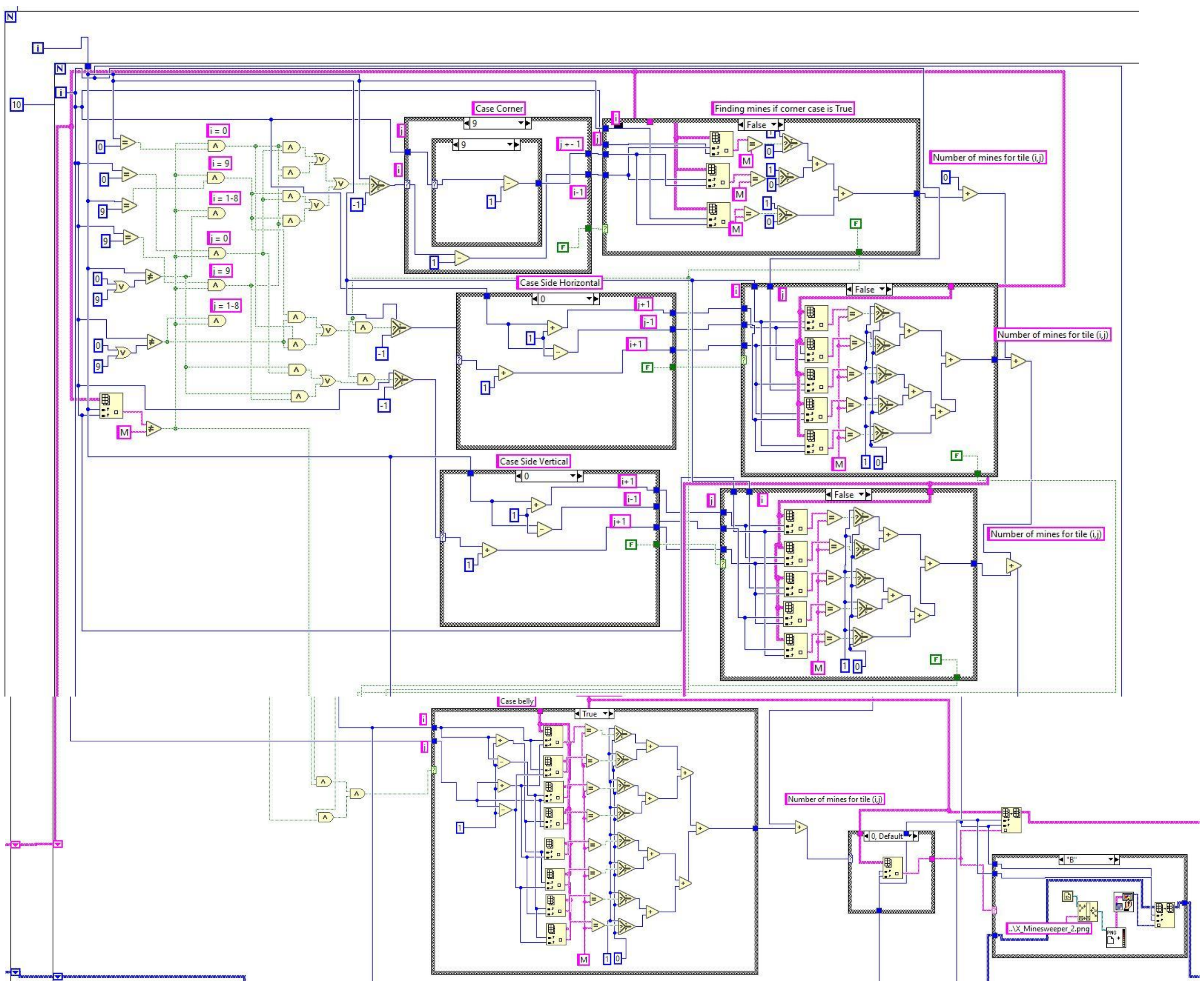


**Figure 9:** The full schematic of the *Initial* state. Note that since LabView does not have a zoom function, we had to take two separate screenshots. Some of the wires may therefore not match in the picture.

## Appendix 2 - Logic to find neighboring tiles

**Table 3:** The three indices representing neighboring tiles for a corner tile.

| i | j |
|---|---|
| i ± 1 | j |
| i ± 1 | j ± 1 |
| i | j ± 1 |

**Table 4:** The five indices representing neighboring tiles for the border tiles. We had to separate whether the tile was on the horizontal or vertical border. These two cases resulted in different indices being sent into the purple check.

| **Horizontal border** | | | **Vertical border** | |
|---|---|---|---|---|
| i | j | | i | j |
| i | j + 1 | | i + 1 | j |
| i | j - 1 | | i + 1 | j ± 1 |
| i ± 1 | j | | i | j ± 1 |
| i ± 1 | j + 1 | | i - 1 | j ± 1 |
| i ± 1 | j - 1 | | i - 1 | j |

**Table 5:** The eight indices representing neighboring tiles for the tiles which are neither border or corner tiles.

| i | j |
|---|---|
| i | j + 1 |
| i | j - 1 |
| i - 1 | j |
| i + 1 | j |
| i + 1 | j + 1 |
| i + 1 | j - 1 |
| i - 1 | j + 1 |
| i - 1 | j - 1 |

## Appendix 3 - Wait for event

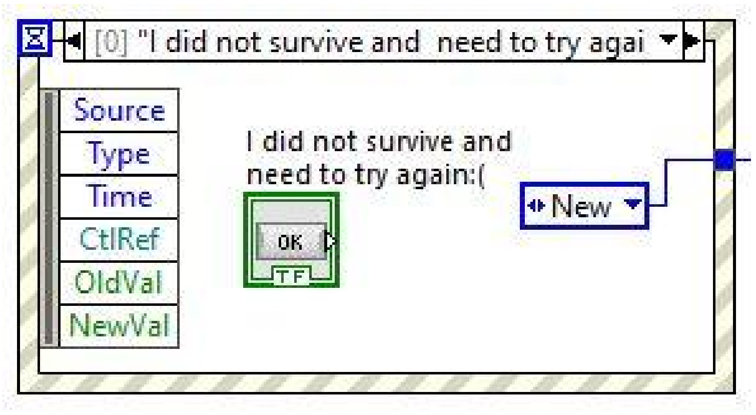

[0] "I did not survive and need to try agai
Source
Type
Time
CtlRef
OldVal
NewVal
I did not survive and need to try again:(
OK
TF
New

**Figure 10:** The *Event Structure* with the event "*I did not survive and need to try again:(: Value change.*". In the event structure one can see the button, and that the code sends the game to the state *New* when triggered.

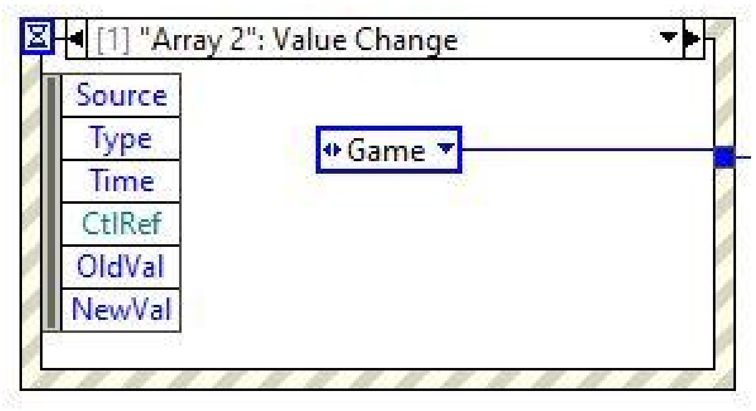


**Figure 11:** The *Event Structure* with the event "*Array 2: Value change*". This is the event that is triggered whenever the player opens a tile. The code then sends the game into the state *Game.*

## Appendix 4 - Game

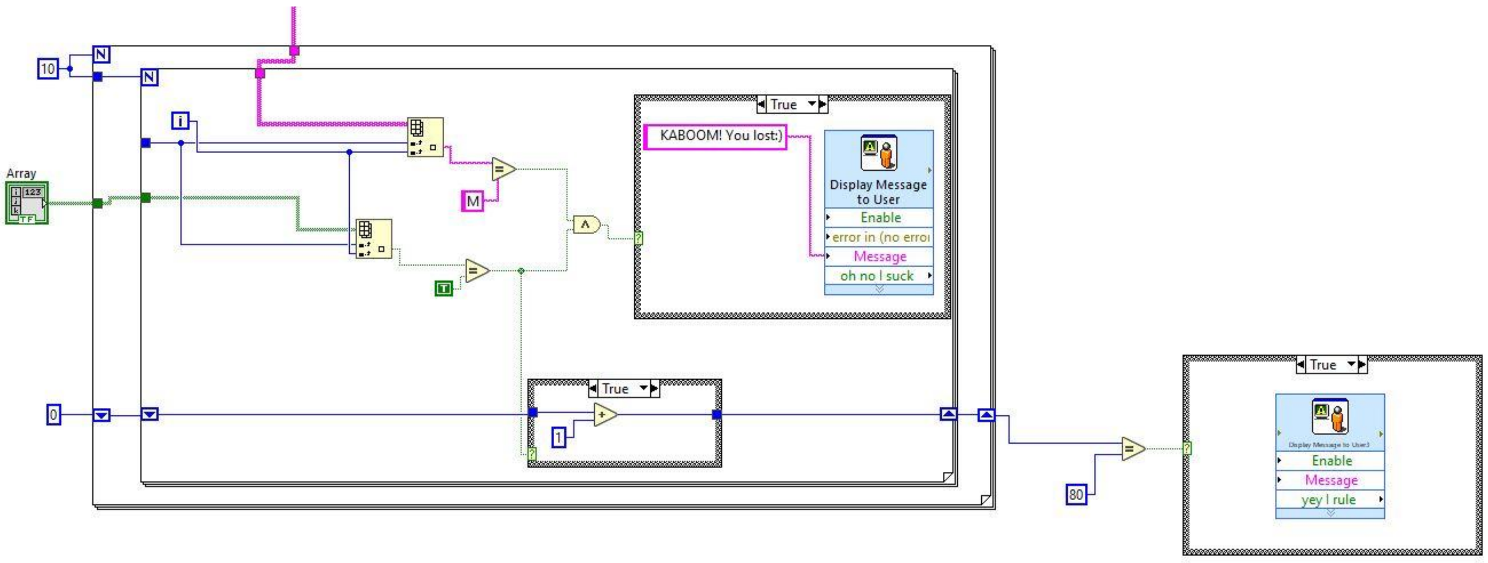


**Figure 12:** This figure contains the LabVIEW schematics of the case *Game.* Here one can see how the code checks if the player has opened a tile with a mine underneath, and the messages that are displayed. In addition, the code checks if the player has won by opening all the tiles which do not have a mine underneath.

## Appendix 5 - New

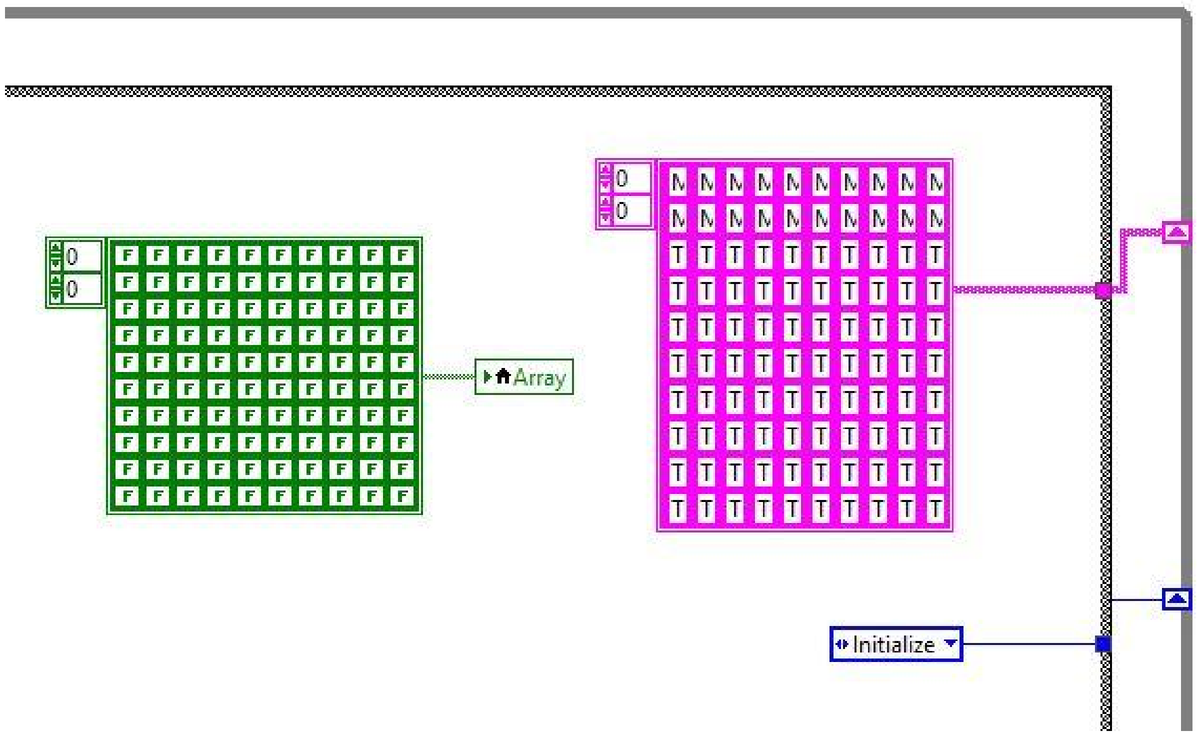


**Figure 13:** This figure contains the LabVIEW schematics for the state *New.* Here one can see how the *Boolean grid* and the *Game grid* is reset after the "reset" button is clicked. In addition, the code sends the game to the state *Initialize.*