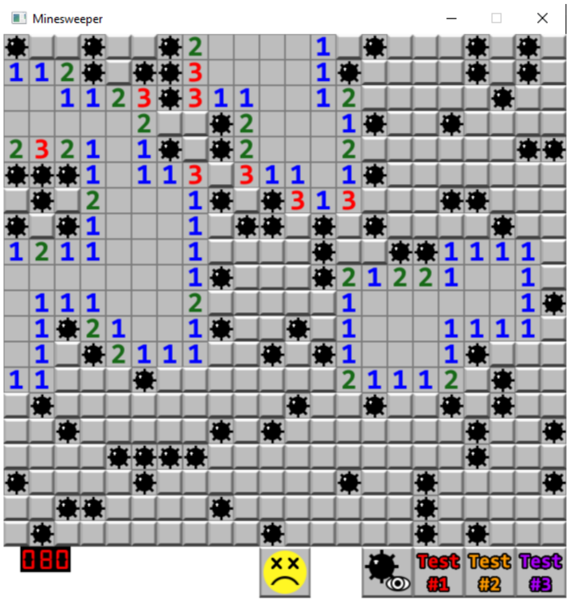
Project 3 – Minesweeper

Contents

Overview

For this project you are going to create a version of the classic game, Minesweeper. Your final version will look something like this:



If you've never played the game before, you can find a number of playable versions of this online:
http://minesweeperonline.com/
http://www.freeminesweeper.org/minecore.html

To create this project you are going to use SFML (the Simple and Fast Multimedia Library) to do the work of drawing images to the screen and getting mouse input, while you will be responsible for everything else.

Description

Rules Overview

The rules of the game are as follows:

There exists a board, which contains a grid of spaces. A space could be a mine, or not. The player clicks on a space, and it gets revealed. The goal of the game is to reveal all the spaces that are not mines, while avoiding the spaces that are.

When a space is revealed:

If it's a mine, the game ends
If it's not a mine, it shows the number of mines **adjacent to** that space (anywhere from 0 to 8, with 0 just showing as an empty space)
- If a space has no adjacent mines, all non-mine spaces adjacent to it are also revealed The player uses the numbers as clues to figure out where other mines may be located.

When all of the non-mine spaces have been revealed, the player wins!

Other features:

Flags: Right-clicking a **hidden** space puts a flag on that space, marking it as a possible mine. Flagged spaces cannot be revealed (with left-clicks or as a result of adjacent spaces being revealed), but another right-click will remove the flag.

Mine Counter: to track the number of mines that are on the board. Every time the player places a flag, the counter goes down by one. Every time they remove a flag, the counter goes up by one. The mine counter CAN go negative!

Restart Button: The smiley face centered at the top or bottom of the window lets you restart a new board with everything reset and mines randomized

Non-standard features for your version of this project

Debug Button: Clicking this button will toggle the visibility of the mines on the board. Use this to help you test/debug various features of the game. Having to play the game properly each time you want to test something is very time-consuming. Creating these developer shortcuts helps speed up the development process

Test Buttons #1-3: Another development shortcut, clicking on these loads a file with a specific board layout, details on this later.

Those are the features that your game will need to have. The rules are pretty simple, but even simple games like this can be challenging to implement.

Window / Board Setup

Each of the three values below will be configured by a text document, **config.cfg**. The first line being the **number of columns**, the second line being the **number of rows**, and the third line being the **number of mines**. There are also preset "test" boards that you can load to give predictable results, which will make testing some aspects of the program easier (these are described in more detail later in the document).

Window Size	Width = number of columns * 32
	Height = (number of rows * 32) + 88
	32 is the number of pixels on the side of a tile the extra 88 pixels is for the space for the menu

Mine count	Set by the config.cfg document, or equal to the number of mines in the premade grid
Tile count	Equal to the number of columns * the number of rows

Example:
If the text document is
25
16
50
Then the width is 25 * 32 = 800, the height is (16 * 32) + 88 = 600, mine count is 50,
and tile count is 25 * 16 = 800

SFML

The library that you will be using in this project is SFML—Simple Fast Multimedia Library. The first thing you would need to do is compile an application using this. Building an application using an external library can be a difficult thing, but it's something that you typically only have to do once at the start of a project, and then you're good to go until that project is finished.

To get started with SFML, visit this site:

https://www.sfml-dev.org/download/sfml/2.5.1/

You want to download the appropriate version for the IDE/compiler that you are using. If you are using something that isn't listed there, it is HIGHLY RECOMMENDED that you install and use one of the recommended tools. On Windows that would be Visual Studio, XCode on MacOS, etc. The links on that page are for already-compiled versions of the library which will work "out of the box" with the appropriate compiler.

Installing and compiling your first "Hello World" program can be a bit tricky, especially if you've never done it before. There are guides here: https://www.sfml-dev.org/tutorials/2.5/, and certainly elsewhere online, but often the best source is from the developer's themselves.

SFML Basics

There are many guides and tutorials on how to use SFML, but the key features that you will be utilizing for this project are:

sf::Sprite	These objects will let you draw some or all of a texture to the screen, the primary object that you will use to represent all aspects of the game
sf:Texture	These store external images that you program needs
sf:Vector2f/i/u	A 2-dimensional vector (an X and Y position), a location on the screen; the f/i/u at the end indicates the type of data used for storage (float, int, unsigned int)
sf::Mouse	The mouse class, giving you information about where the cursor is located, and whether or not a mouse button is pressed

SFML Tutorials

This document will not replicate the wealth of information already out there about this library. The primary list of examples/tutorials can be found here: https://www.sfml-dev.org/tutorials/2.5/

From that page, a few in particular you will find useful for this project:

https://www.sfml-dev.org/tutorials/2.5/window-window.php https://www.sfml-dev.org/tutorials/2.5/window-events.php https://www.sfml-dev.org/tutorials/2.5/graphics-sprite.php

Anything beyond that will not be applicable for this project (networking, audio, etc will NOT be used). Everything you see on the screen (each space, numbers, buttons, etc) will all be created and drawn the same way: load a texture, create one or more sprites from that texture, and then draw them to the screen.

You will have to load and store the images listed in the next section in sf::Texture objects.

SF::Texture Objects and Global Variables

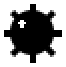
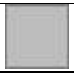
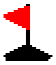
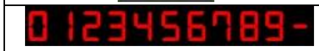
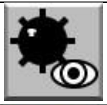
In this project you should not, under any circumstances, attempt to use sf::Texture objects in global space. Globals in general should be avoided, but if you try to load a sf::Texture in global space it may be initialized before other parts of SFML are initialized, causing it to crash before main() even starts.

To make matters worse, this problem might not occur on your machine, but it could on someone else's (i.e. the person grading your project).

Images

For this project you will have a folder, creatively called **images**, where all of the images for this project will be stored. These images will all be loaded as sf::Texture objects to be used in the creation of sf::Sprite objects.

The images are as follows:

Game Images		
	mine.png	The star of the game (although if you play properly, you'll never see one!)
	tile_hidden.png	What all tiles look like before they are clicked/revealed
	tile_revealed.png	A revealed tile with no adjacent tiles
	number_#.png	Tiles with the numbers 1-8 on them (replace # with the appropriate number. Used for tiles that have 1-8 adjacent mines
	flag.png	Draw this on top of hidden tiles when they are flagged by the player as possible mines.
UI Images		
	face_happy.png	Click this button to reset the map. New mines, everything hidden, it's like you restarted the program.
	face_win.png	Victory!
	face_lose.png	The opposite of victory! (It's cool, no smiley faces were harmed during the creation of this project)
	digits.png	Used for the digits on the "remaining mines" display. You can use this one texture for all the numbers, and change the "texture rect" of a sprite to draw a different portion of the image. The size of each digit (and the size of the texture rect you should use) is 21 x 32 pixels, and each digit would be offset by 21 (the width) times the digit you wanted. See https://www.sfml-dev.org/tutorials/2.5/graphicssprite.php for more information
	debug.png	Used to toggle debug mode
Test #1 Test #2 Test #3	test_1/2/3.png	Used to load test files from which the board will be set

Other Features

Buttons

A button is really just an image that you can click on to make something happen. A more complex UI system would use an event/messaging system, but on a basic level you just need a sf::Sprite to represent the button, and every time the player clicks on something, you need to check if that mouse click occurred inside the boundaries of the sf::Sprite you're using as the button.

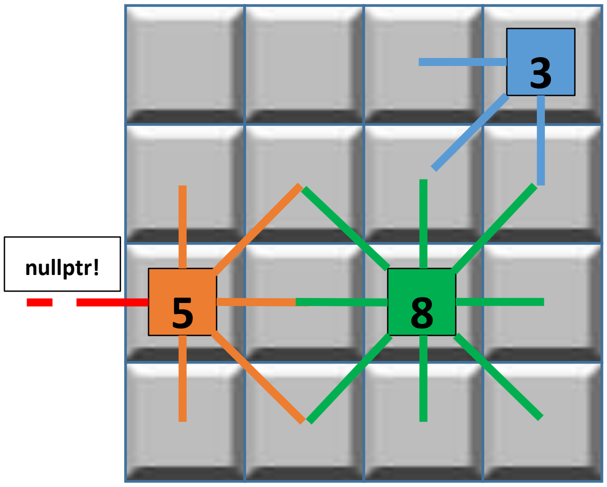
If you're drawing a sprite somewhere, you know its position (it's 0, 0 by default, or whatever you set it to). You can get the width/height of the sprite through its textureRect, and then it's just a matter of checking if the mouse position is inside that box.

Adjacent Mines and Tiles

In order to calculate the number of nearby mines, as well as when revealing tiles, each tile should store a list of neighboring tiles. A tile could have **UP TO** 8 neighbors. An easy way to do this is with pointers. Since the number is a variable, a dynamically sized container would be perfect for this. You could also use a fixed-length array, since no tile will ever have more than 8 neighbors.

```
vector<Tile*> adjacentTiles; // Store each tile near us, the size() of each vector will vary
Tile* neighbors[8]; // Always 8 pointers, some of which might be nullptr
```

Adjacent Mines



In this example, tiles on the corners have 3 adjacent neighbors. The tiles on the edges of the board have 5 neighbors, while tiles not on the corner or the edges have 8 neighbors.

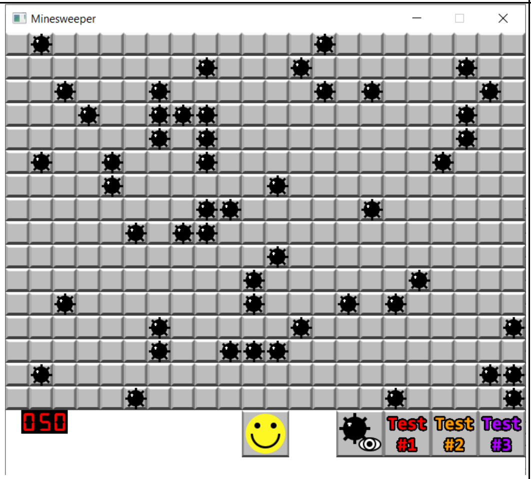
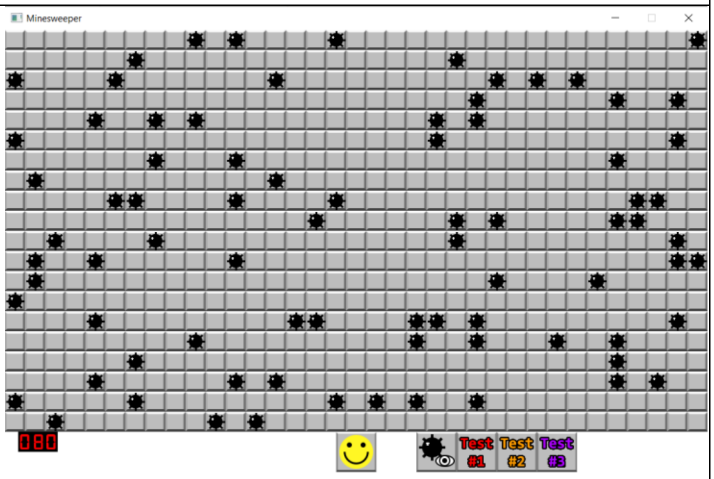
If you were to create an array of 8 pointers, some of them might not be valid, depending on where that particular tile is located

How you determine which tiles are nearby depends on how you set up the original board. 2D array? Each index will have neighbors that are +- 1 in the x/y axes.

Using a single array/vector? The calculations will be a bit different.

Config File

When starting the program, you should open the config.cfg in the “boards” folder. This is a plain text file, and you can open it in any text editor. This file contains three lines, the first being the number of columns, the second being the number of rows, and the third being the number of mines. This will not be changed during a game. The number of columns will never be less than 22, so that all of the menu buttons will be able to fit with the face centered and the buttons space correctly. Also the number of mines will be a valid number less than or equal to the number of tiles.

config	config
22 16 50	35 20 50
	

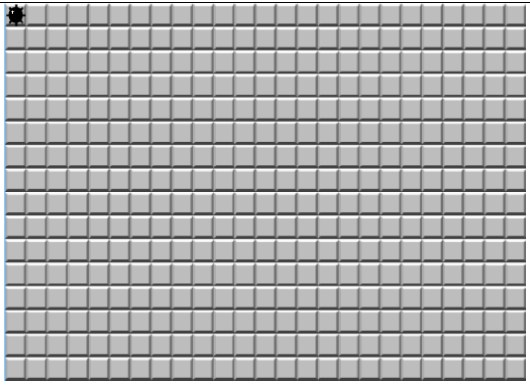
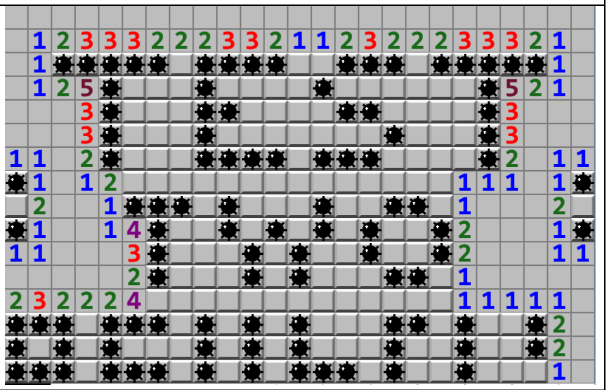
We will provide one example, but we can test it with A DIFFERENT ONE THAN THE PROVIDED SAMPLE.

Loading board from files

When clicking any of these buttons:

You should open up one of the three files located in the “boards” folder. For Test #1, you should open “testBoard1.brd”, test #2 is “testBoard2.brd”, etc. These are plain text files, and you can open them in any text editor.

Those files contain a bunch of 0s and 1s to represent the layout of a particular map. Why use these? When developing any project, having some sort of test data, some known value, is essential. How you do you know if hundreds of randomly generated values are correct or not? That’s really difficult. Using specific patterns can help you determine if you’re code is working correctly.

testBoard1	**testBoard2**
1000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000 0000000000000000000000000	0000000000000000000000000 0000000000000000000000000 0011111011111001110111100 0000100010000100000010000 0000100011000011000010000 0000100010000000100010000 0000100011110111000010000 1000000000000000000000001 0000011101000100110000000 1000001001010101001000001 0000001000101001001000000 0000001000101000110000000 0000000000000000000000000 1110111010101000110100100 1010100010101000100100100 1110111010101110100100000
	
testboard1 loaded, no revealed tiles	**testboard2** loaded, one tile on the edge revealed (which caused a cascaded of revealed tiles)

Your code will be tested with boards that are DIFFERENT THAN THE PROVIDED SAMPLES. While you may hard-code tests in your own projects, using external data makes your program more flexible. Two additional boards, **recursion_test** and **lots_o_mines** can be used to test the recursive reveal algorithm and the mine counter.

NOTE: The test boards must match the dimensions in the config.cfg file. For example, if the dimensions in the config file are 20 x 32, the .brd file must have 20 columns and 32 rows, otherwise you will get unusual results! The provided test boards all have a size of 25 x 16

Code Structure

With larger programs, you can accomplish the goal in any number of ways. There isn't a single way to write this that works better above all others. From the outside perspective (i.e. that of a player), your application needs to DO various things:

- When the player clicks a space, reveal it.
- When the player clicks the restart button, reset the board.
- If a mine is revealed, end the game. And so on.

HOW you choose to accomplish those things is up to you. If you want to write a single, gigantic main() function, you are free to do so—that approach is not recommended, however. A few suggestions:

A class to represent the board. This represents the core data object in the game.

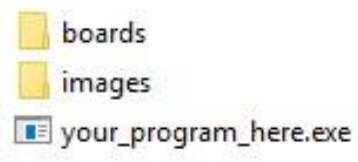
A class for tiles/spaces. The board is made up of a whole lot of these things. Each one of these can be a mine, have a flag, some number of adjacent tiles/mines, etc.

Many programs (games or otherwise) do the same things over and over again while the application is running. The ability to easily (in code) reset everything is critical. Think about what sorts of helper functions you might want to make that happen. Things like:

Restarting the board
Setting or clearing tiles of flags
Setting or clearing mines (singly or in large quantities)
Recalculating the number of adjacent mines Etc…

Paths

In this project, any operations involving files (loading textures and boards) should use RELATIVE paths, NOT absolute paths. Your code should be based around a folder structure like that the image on the right. When you load a texture, it should be from the images folder, like "images/mine.png", and when you load a board file or the config file, it should be from the boards folder, such as "boards/testboard2.brd"



Be sure to use FORWARD slashes in your paths for compatibility. For example:

"images\\mine.png" – this will work on Windows, but not elsewhere. Don't use this.
"images/mine.png" – this will work everywhere. Do use this!

Mouse Interactions

The application can really do "nothing" until the user clicks their mouse. Typically games and many other applications are clearing/redrawing the screen on a regular basis (often dozens of times a second). Until the player clicks the mouse somewhere in the window, the program will appear to just sit there, idle.

Once the player has clicked, however (you can check for this in the event loop), you then need to do some checks about that click.

Where did they click?
Is that a valid space on the board anywhere? If so, should you do anything in response to this?

Did the player just click a mine?
Boom, game over!
If not a mine, reveal the tile (and possibly reveal adjacent tiles, which could reveal more tiles…)

Revealing a tile
If the number of adjacent mines is 0, reveal any neighboring tiles as well (as long as they aren't mines)
In revealing those, do the same sort of check for any neighbors to that tile… (sounds a bit like recursion here!)

Storing Resources

While a program is running, it needs RESOURCES to get the job done—things like icons, textures, sound files, etc. Many of the resources need to be stored for long-term use, as they may be called upon time and time again… but you don't always know when they'll be needed when you compile your code.

A great storage container for assets that you want to reference by name is the **map**<>. Storing something that you can access by its name with container["NameOfAsset"] is vastly preferable to that of dealing with arrays—was "GameOver.png" stored in array[25], or array[26]?

You may find it helpful to create a single storage container for all of the sf::Texture objects, and then pass that around to any class which might need those files.

Using Documentation

Reading through documentation, help files, guides, tutorials, etc is an absolutely critical skill that you must develop. The problem you are currently trying to solve, the exact combination of variables for your scenario might not have existed before now.

The One True Answer to your problem might not be out there on the Internet, in a StackOverflow.com question, or in a video on YouTube. However, the information to help you figure out PARTS of your problem are almost certainly out there. You will have to figure out how to make sense of those smaller bits of information and decide on a proper course of action.

For example, the data referenced sf::Texture objects disappears when the object is deleted, or falls out of scope. You can't create a sf::Texture inside a function, create a sf::Sprite from that texture, and then use the sprite outside the function. An example of this (and what not to do) in the documentation:

https://www.sfml-dev.org/tutorials/2.5/graphics-sprite.php#the-white-square-problem

Tips

Any libraries or APIs that you work with will have some sort of documentation. READ IT! You absolutely MUST get used to being able to sift through information to find the answers that you are looking for.

Don’t be afraid to experiment! When getting access to new code, you have to figure out how it works. Documentation is all fine and good, but at some point you have to actually DO IT yourself. Learning by doing is the most effective way. Write some code, screw it up, fix that code, do it all over again.

Don’t try to write the entire program all at once. Hard-code test values if you need to. Try to get one single tile working on a basic level (position, responding to mouse clicks, etc) before creating dozens/hundreds of them.

Think about what types of classes or functions you might want to have for this project. There is a board, a board has tiles, tiles have various properties or states… How do you want to store that data? An array? A vector<>? A 2-dimensional array?

You’re the one writing the code! Write it in a way that makes sense to you. Everyone tackles problems a bit differently, find an approach that works for you.

Submissions

You are going to turn in your source code, and your source code only. No images, no SFML libraries, only the .h and .cpp files you wrote to complete the project. Zip up **JUST YOUR SOURCE CODE** and name it **LastName.FirstName.Project3.zip.**

Partial Credit for Features that Partially Work

The features you need to implement for this project are listed in the rubric on the next page, along with point values. For each feature you can get full points, half points, or no points.

1. Full points – Feature works perfectly, no bugs of any kind
2. Half points – Feature has any bugs at all, or is partially implemented
3. No points – Feature not implemented at all, or so minimally implemented that no functionality exists (for example: drawing a button to the screen that you can’t click on at all doesn’t count as partial implementation)

Point deductions:

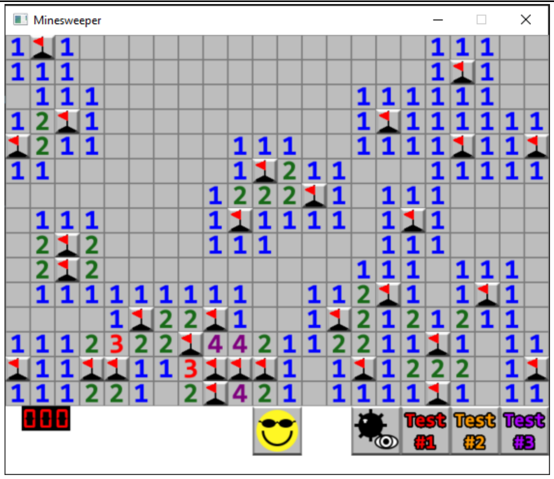
-10 for using global variables (sf::Texture objects or otherwise).
-10 for not using relative paths (or for not using forward slashes in your paths), or submitting anything other than your source code

Rubric

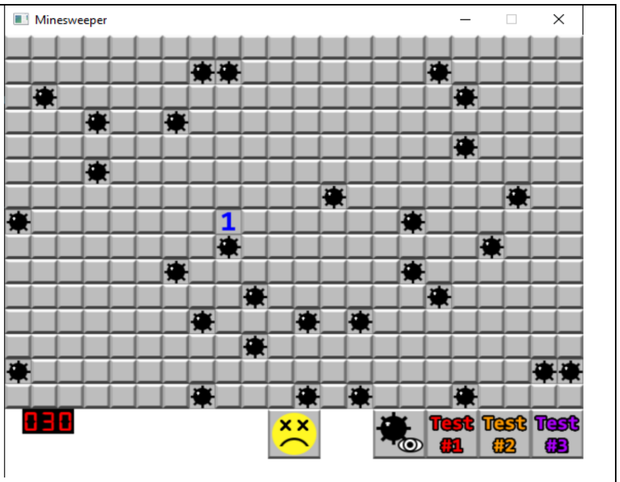
config.cfg file	Board changes size and its number of mines based on values set in config.cfg file. **Your code will be tested with DIFFERENT CONFIG FILES than the sample provided. (Same file format, just different values.)**	8
Tile Revealing	Clicking on a tile reveals it. If it is a mine, game over. If it has 0 adjacent mines, reveal all neighboring tiles which are **not currently revealed**, **not mines** and **not flagged**, and then each of those neighbors go through this process as well. Depending on the board layout, a single click could reveal nearly the entire board!	16
Tile Display	Tiles display depending on their state: Unrevealed (the default state) Revealed, and empty (no adjacent mines) Revealed, and near 1-8 mines (showing the appropriate number) Revealed, and showing a mine	8
Flags	Right-clicking on a hidden tile sets a flag on it. Right-clicking a flagged tile removes the flag. Left-clicking a tile with a flag has no effect. Flagged tiles **cannot be revealed in any way** (by the player, or by a revealing algorithm). The flag **must** be removed first. The number of flags on the board affects the counter (see below).	12
Mines Remaining	A counter of how many mines are on the board, minus the number of flags placed. Adding/removing flags from tiles affects this. Remaining flags CAN go negative!	12
End Conditions - Victory	Revealing all **non-mine** tiles ends the game, and marks all remaining tiles (i.e. the mines) with flags. Flagging mines will affect the counter, so the final counter will be a 0 after you win. Smiley face changes to sunglasses version. - No further interactions with the game board are possible (you did it all already!) - The player CAN click the smiley face to start a new game, or use the testing buttons to load one of the test boards. (The debug button shouldn't do anything at this point… game's over, you know where the mines are!)	8
End Conditions - Defeat	Clicking on a mine ends the game. What should happen: - All tiles with mines are revealed (and display on top of any flags you may have place) - The smiley face changes to the dead face (he's just acting, don't worry!) - No further interactions with the game board are possible. - The player CAN click the dead smiley face to start a new game, or use any of the testing buttons. (The debug button shouldn't do anything at this point… game's over, you know where the mines are!)	8
Random Mine Placement	When the game **starts** and when the board is **reset** (by clicking the smiley face button), a number of mines equal to the number specified in the .cfg file are randomly placed on the map--no more, no less.	8

Test Buttons (1, 2 and 3)	3 test buttons to load testBoard1.brd, testBoard2.brd, testBoard3.brd. Each button updates all board spaces according to the data in the file, resets all flags and the counter – it's like you just started the program, with a specific set of data instead of random tiles. **Your code will be tested with DIFFERENT BOARD FILES than the samples provided. (Same file format, just different 1s and 0s—don't hard-code results!)**	12
Debug Button	Clicking the debug button toggles whether or not to show the mines on the map. Since the intent of this feature is to help YOU, the programmer, see the mines, draw the mines OVER anything else. Don't stop drawing anything else. These are in addition to whatever is normally being displayed.	8
	Total	100

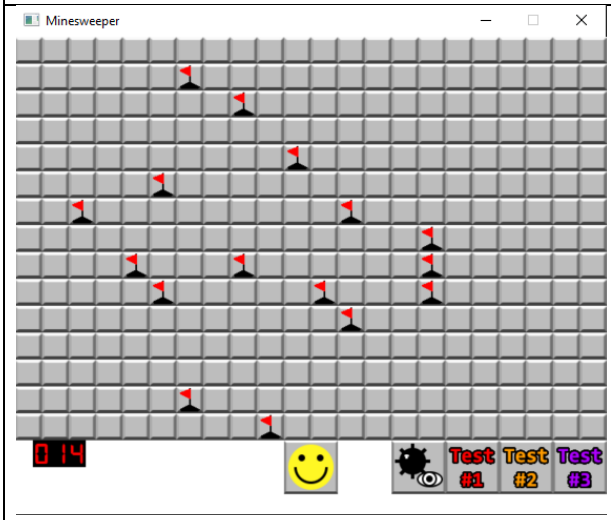
Sample Images



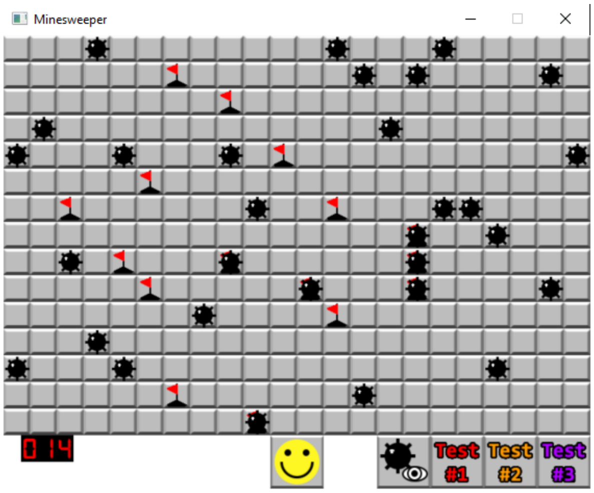
Victory!



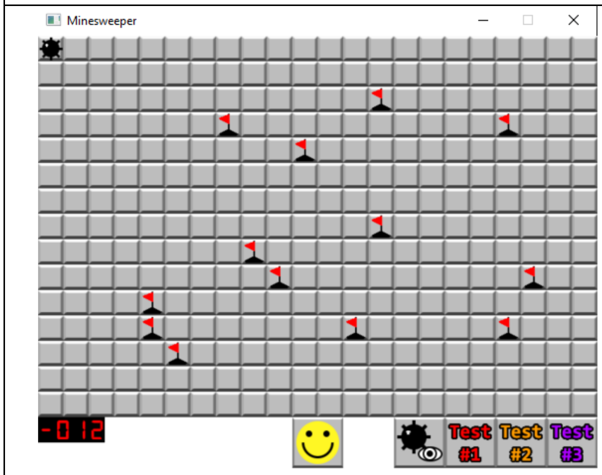
Defeat...



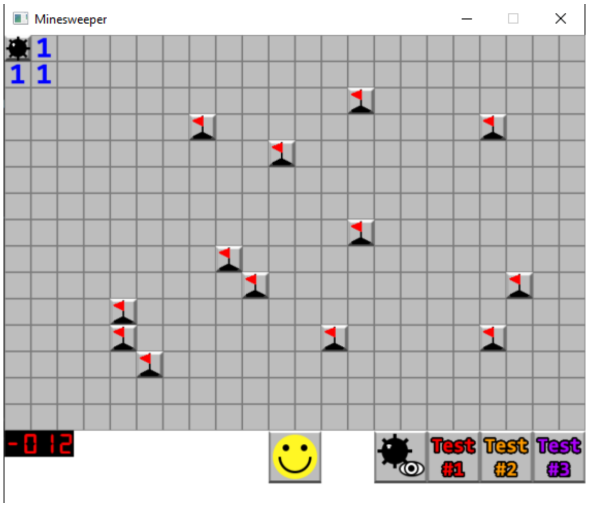
Flagged tiles (on a board with 50 mines)



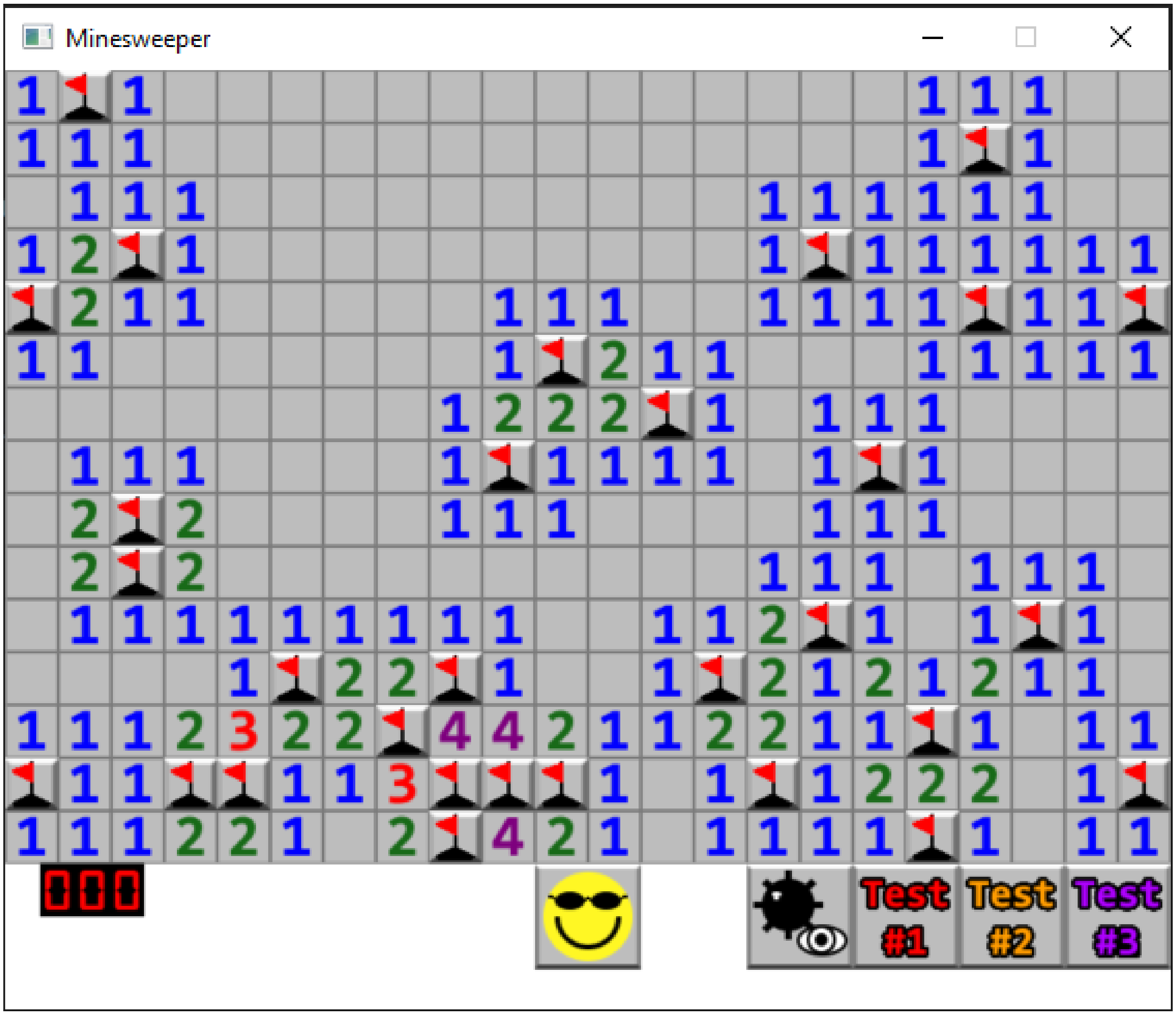
Same scenario, debug mode turned on



Negative counter (Flags on a board with only 1 mine)



Same board, after just one click



Minesweeper

1	🚩	1															1	1	1		
1	1	1															1	🚩	1		
	1	1	1											1	1	1	1	1	1		
1	2	🚩	1											1	🚩	1	1	1	1	1	1
🚩	2	1	1						1	1	1			1	1	1	1	🚩	1	1	🚩
1	1								1	🚩	2	1	1				1	1	1	1	1
								1	2	2	2	🚩	1		1	1	1				
	1	1	1					1	🚩	1	1	1	1		1	🚩	1				
	2	🚩	2					1	1	1					1	1	1				
	2	🚩	2											1	1	1		1	1	1	
	1	1	1	1	1	1	1	1	1			1	1	2	🚩	1		1	🚩	1	
				1	🚩	2	2	🚩	1			1	🚩	2	1	2	1	2	1	1	
1	1	1	2	3	2	2	🚩	4	4	2	1	1	2	2	1	1	🚩	1		1	1
🚩	1	1	🚩	🚩	1	1	3	🚩	🚩	🚩	1		1	🚩	1	2	2	2		1	🚩
1	1	1	2	2	1		2	🚩	4	2	1		1	1	1	1	🚩	1		1	1

000