

Project 3:

Minesweeper

COP3503C: Programming Fundamentals II

University of Florida

Laura Cruz Castro, Chaitanya Nulu, Cameron Brown, Joshua Fox

Spring 2024 — Revision 2

Welcome to the third and final project of the course: Minesweeper. Minesweeper is a classic game that has been around since the 1960s - you may have played it as a kid! In this project, you are going to design an entire Minesweeper game from scratch. Along the way, you will explore concepts like pointers, recursion, and displaying graphics on the screen. You'll be laying down the logic grid by grid, ensuring you don't blow up your chances of a bug-free game.

Contents

1 Overview	3
2 Minesweeper	3
3 SFML	4
3.1 About	4
3.2 Setup	4
3.2.1 Alternate Setup	4
3.3 Introduction to SFML	4
4 Windows	5
5 Welcome Window	5
5.1 Objective	5
5.2 Behavior and Validation	6
5.3 Helpful Classes and Functions	7
5.4 Additional Notes	7
6 Game Window	7
6.1 Objective	7
6.2 Images	8
6.3 Features	9
6.4 Related Concepts	11
6.5 Additional Notes	12
6.5.1 Config File	12
6.5.2 Buttons	12
6.5.3 Mouse Interactions	12
6.5.4 Adjacent Mines and Tiles	12
6.5.5 UI Locations	12
7 Leaderboard Window	13
7.1 Objective	13
7.2 Behavior	13
7.3 Related Concepts	14
7.4 Formatting and Output	14
8 Implementation Tips	15
8.1 Storing Resources	15
8.2 Global Variables	15
8.3 Paths	15
8.4 Managing Textures	16
8.5 Code Structure	16
9 Milestones	16
9.1 Milestone 1: Due April 12 (5 points)	17
9.2 Milestone 2: Due April 19 (15 points)	17
9.3 Milestone 3: Due April 24 (130 points)	18
10 Submission	20
11 Grading	21
A Sample Makefile	22
B Document Revisions	23

1 Overview

For this project, you will create a version of the classic game, Minesweeper. Minesweeper is a single-player puzzle game that has been around since the 1960s. The goal of the game is to clear a rectangular board containing hidden “mines” without detonating any of them. To help you avoid the mines, the game shows you the number of mines that are adjacent to each square. You can use this information to deduce the location of mines and avoid them.

If you’ve never played the game before, you can find several playable versions of this online:

- <http://minesweeperonline.com/>
- <http://www.freeminesweeper.org/minecore.html>

Your final version will look something like this:

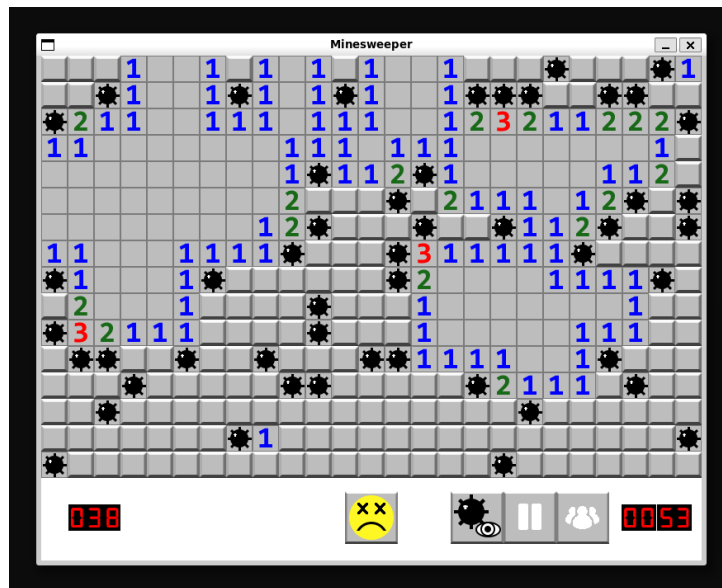


Figure 1.1: An example of what your final version will look like.

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.

2 Minesweeper

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!

3 SFML

3.1 About

The library you will use in this project is SFML, the Simple Fast Multimedia Library. SFML is a cross-platform library written in C++ with bindings available for many other languages. It provides a simple interface for drawing images to the screen, playing sounds, and getting input from the user.

Your first step is to compile an application using the library. Building an application using an external library can be difficult, 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 complete.

3.2 Setup

If you are using CLion, setting SFML can be done by configuring the CMakeLists.txt in your CLion project. You can find the CMakeLists.txt file [here](#).

Alex Johnson, a peer mentor for this course, has created a walkthrough video showing how to install SFML on CLion with this method, available [here](#). The video walks you through the following steps:

1. Copy the CMakeLists.txt from the link above and paste it into your project.
2. In the file, replace any instances of CMakeSFMLProject with the name of your project (which you chose when starting your project in CLion). You can find the name in the solution/file explorer to the left side of your screen.
3. In the call to add_executable, add in any project files you wish to compile when building your project.
4. If you are using a M-series Mac computer (M1, M2, etc.), set your compiler to the Clang compiler. You can find this setting in "Build, Execution, Deployment" → "Toolchains" in the CLion settings. Set your C++ compiler to Clang.

3.2.1 Alternate Setup

If you wish to install SFML manually on your computer, or you are not using CLion to develop your project, you can follow the steps below.

You want to download the appropriate version for the IDE or compiler that you are using. You can visit the [Download page on the SFML website](#) and choose the version of SFML that works for your development setup.

If you're working with Visual Studio or CLion, we also recommend watching these tutorials, as they may speed up the process of setting up your project:

- **Visual Studio 2019:** [YouTube Video: "best way to setup SFML in Visual Studio 2019"](#)
- **Visual Studio 2022:** [YouTube Video: "OpenGL SFML Visual Studio 2022 Installation"](#)
- **Command line via g++:** Install SFML using `brew install sfml`. Then use the Makefile linked in Listing A.1.

When you install SFML, you may need to install or update other developer tools that allow SFML to work on your computer.

3.3 Introduction to SFML

There are many guides and tutorials on how to use SFML, but the key features that you will be utilizing for this project are concepts like `sf::Font`, `sf::Text`, `sf::Sprite`, `sf::Texture`, etc. All these concepts will be discussed in the later sections wherever they are being used.

This document will not replicate the wealth of information already out there about this library. The [Tutorials](#) page on the SFML website has several great tutorials on the various parts of SFML. We recommend reading (or at least, skimming) the following tutorials:

- [Opening and managing a SFML window](#): Explains how to open and manage the state of a window, where all of your sprites will be drawn.
- [Events explained](#): Explains events that are sent when the user interacts with your window, and how you should respond to them.

- [Sprites and textures](#): Explains sprites, or the objects in your window.
- [Texts and fonts](#): Explains how to write text in your window, and how to work with fonts.

Anything beyond that will not be applicable for this project (networking, audio, etc. will **not** be used). Everything you see on the screen (each tile, number, button, etc.) will 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, or if it's a text, we load a font once, and then create a Text object, set its size and display it at a certain position.

4 Windows

In this project, you will be working on three different windows: the welcome window, the game window, and the leaderboard window. A window in SFML is a graphical window that displays graphics and allows user interaction in a computer program.

We use the [sf::RenderWindow](#) to create a new object of the type render window. It is a subclass of [sf::Window](#) that provides a 2D rendering context. It allows you to draw graphics and handle user input events, such as mouse clicks and keyboard presses, using the SFML graphics module. It is a fundamental class for creating 2D games and interactive applications using SFML.

All the windows you will be creating will have fixed dimensions and cannot be resized. Here's how you can create a simple `sf::RenderWindow` of size 800x600 pixels with the title "SFML Window" and the resize option disabled.

```
1 sf::RenderWindow window(sf::VideoMode(800, 600), "SFML window", sf::Style::Close);
```

For all three windows, you will be setting their width and height according to the guidelines. Each of the three values below will be configured by in a local file named `config.cfg`. There are three lines representing the number of columns, the number of rows, and the number of mines, respectively. You can find more information about the config file in subsection 6.5.

Window	Background Color	Height (px)	Width (px)
Welcome Window	Blue	$(\text{rowCount} \times 32) + 100$	$(\text{colCount} \times 32)$
Game Window	White	$(\text{rowCount} \times 32) + 100$	$(\text{colCount} \times 32)$
Leaderboard Window	Blue	$(\text{rowCount} \times 16) + 50$	$(\text{colCount} \times 16)$

Table 4.1: Characteristics of Different Windows in the Game

5 Welcome Window

5.1 Objective

This will be the first thing the player should see when you run the game. It will prompt the user for their name. The user will type their name using the keyboard as input.



Figure 5.1: Example of the welcome window.

5.2 Behavior and Validation

In your window, you will need to implement the following behavior. Some of the behavior is meant to validate the user's name, and other behavior focuses on the interaction of the different windows to provide a great user experience.

- The user cannot enter more than 10 characters while typing their name.
- The user has to see the characters as they type and pressing backspace should remove the last character.
- You should also display a cursor indicator '|' as the user types in their name. This cursor is just to indicate where the text is being typed. You don't have to implement any blinking effect or use arrow keys to navigate between the text as this is not an actual cursor. But you can implement these features if you want to.
- The characters should only contain alphabetical characters (letters A through Z). So when a user types a number/special character, you shouldn't read it.
- No matter how the user enters their name, you should capitalize the first letter and set the other letters to lowercase. This correction should be visible as the user types not just when you are about to store the name. Here are some examples:
 - alex → Alex
 - bRUCe → Bruce
 - Cameron → Cameron
- When the user presses the Enter key, this window should close and the game window should open if at least one character has been input. Otherwise, the Enter key should not do anything.
- If the user manually closes this window (via pressing the X button, for example), the program should stop executing, and the game window should not load.
- You will be using certain SFML libraries for displaying text which will be discussed later in the next section.

5.3 Helpful Classes and Functions

Here are some helpful classes and functions you can use in this step:

Class/Function	Description
<code>sf::Font</code>	Loads and manages fonts for text rendering.
<code>sf::Text</code>	Displays and styles text (position, size, font, style).
<code>sf::Rect</code>	Defines a rectangle with floating-point dimensions, useful for positioning and bounding box calculations.
<code>sf::Color</code>	Manages RGBA colors for graphics elements.
<code>sf::Vector2</code>	Represents a 2D vector, used for positions and sizes.
<code>sf::Keyboard</code>	Handles keyboard input, like key presses.
<code>std::tolower()</code>	Converts characters to lower case.
<code>std::toupper()</code>	Converts characters to upper case.
<code>std::isalpha()</code>	Checks if a character is alphabetic.
<code>sf::Event</code>	Represents and handles user/system events (e.g., key presses, mouse clicks).

Table 5.1: List of helpful classes and functions for the welcome window.

5.4 Additional Notes

The Welcome window will have the same width and height as the Game Window. The specific guidelines regarding the dimensions can be found in the subsection 6.5. The font you are going to use is `files/font.ttf`. Here are some details regarding the components you will be displaying in this window as shown in Figure 5.1:

Text Object	Bold	Underlined	Color	Size	Position
"WELCOME TO MINESWEEPER!"	Yes	Yes	White	24 px	$\frac{\text{width}}{2}, \frac{\text{height}}{2} - 150$
"Enter your name:"	Yes		White	20 px	$\frac{\text{width}}{2}, \frac{\text{height}}{2} - 75$
User-typed name	Yes		Yellow	18 px	$\frac{\text{width}}{2}, \frac{\text{height}}{2} - 45$

Table 5.2: Properties of Text Elements in the Welcome Window

Note: All text needs to be aligned to the center. By default, `sf::Text` objects have their origins located in the top left corner. You can use the code below to shift the origin to the center and set the text location to position (x, y).

```

1 void setText(sf::Text &text, float x, float y){
2     sf::FloatRect textRect = text.getLocalBounds();
3     text.setOrigin(textRect.left + textRect.width/2.0f,
4                   textRect.top + textRect.height/2.0f);
5     text.setPosition(sf::Vector2f(x, y));
6 }
```

Listing 5.1: Example of using the `setText` function.

6 Game Window

6.1 Objective

This is where you would actually implement the minesweeper game as shown in Figure 6.3. The rules of the game are outlined in section 2.

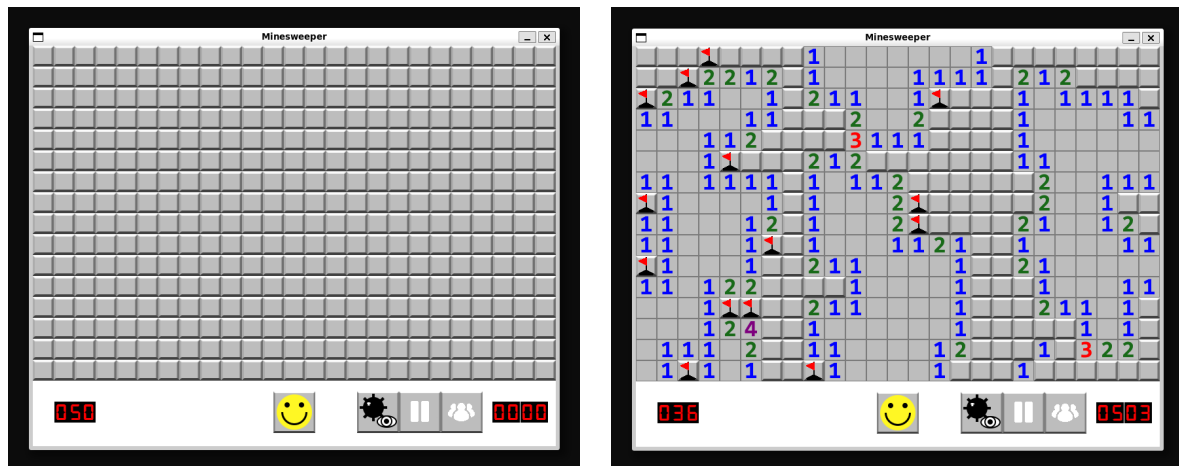


Figure 6.1: Example of the Game Window; Left: The window when the game just starts; Right: The window during the gameplay

6.2 Images

All the images shown below are available inside the `files/images` folder.

First, let's take a look at all the images you'll need to make your game:





Icon	Filename	Description
	<code>mine.png</code>	The star of the game (although if you play properly, you'll never see one!)
	<code>tile_hidden.png</code>	What all tiles look like before they are clicked/revealed
	<code>tile_revealed.png</code>	A revealed tile with no adjacent mines
	<code>number_#.png</code>	Tiles with the numbers 1-8 on them (replace # with the appropriate number). Used for tiles that have 1-8 adjacent mines

Table 6.1: Description of Game Images

Now, let's take a look at all the images you'll need to make the user controls for your game:









Icon	Filename	Description
	face_happy.png	Click this button to reset the map. New mines, everything hidden, it's like you restart 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 and the "timer". 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. You can do this by using <code>sf::IntRect</code> . See https://www.sfml-dev.org/tutorials/2.5/graphics-sprite.php for more information.
	debug.png	Used to toggle debug mode
	pause.png	Used to pause the game.
	play.png	Used to play the game when it's paused.
	leaderboard.png	Used to launch the leaderboard window.

Table 6.2: Description of UI Images

6.3 Features

There are many things to do in this window. But we can break them down into the following components:

- **Tiles:** By default all tiles use the `tile_hidden.png` sprite. A tile can be left-clicked to reveal it, or right-clicked to place a flag/remove a flag that's already placed. The `flag.png` will be placed on top of the `tile_hidden.png` sprite, while the `tile_revealed.png` sprite is displayed when the tile is left-clicked.
Depending on the following conditions, you should display another sprite on top of the revealed tile sprite:
 - If the tile has a mine, draw the `mine.png` sprite. Then you reveal all the tiles with mines and end the game. If there's a flag on a tile, then you need to draw the mine sprite on top of it i.e., the flag sprite stays in the background while the mine sprite is drawn in the foreground.
 - For tiles that have no mines, check all its neighbors for the number of mines around them and display this value using the `number_#.png` sprite.
 - When the revealed tile has no adjacent mines, you will just display a revealed tile and all non-mine tiles surrounding it will be revealed. This happens recursively.
 - When all non-mine tiles are revealed, the player wins. So you **MUST** place flags on top of all tiles with mines.
- **Happy Face Button:** This button lets you restart a new board with everything reset and mines randomized at any point in time. It also has two additional functions: the sprite changes to `face_win.png` when the player wins, and to `face_lose.png` when the player loses.
- **Counter:** Helps 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 counter **CAN** go negative!
- **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. The debug button shouldn't work when the game ends (victory/loss).

- **Pause/Play Button:** Pauses or Plays the game. All functionality except the Happy Face button and Leaderboard button should be disabled. All tiles (regardless of debug mode status) should display `tile_revealed.png` sprite. The sprites for this button switch from `pause.png` to `play.png` and vice-versa depending on whether the game is paused or not. The timer will also stop when the pause button is pressed, and it will start again when the play button is pressed. The tiles must go back to their previous states once play is pressed. The pause/play button won't work when the game ends (victory/loss).
- **Leaderboard Button:** Will pause the game, and all tiles (regardless of debug mode status) should display `tile_revealed.png` sprite and display the leaderboard window. During this stage, the game window shouldn't be interactive, and the timer should stop. When the leaderboard window is closed, all tiles go back to their previous states, and the timer resumes. Note that if the pause button is pressed before the leaderboard button, then closing the leaderboard button should not resume the game.
- **Timer:** Displays the amount of time that has passed since the beginning of the game.

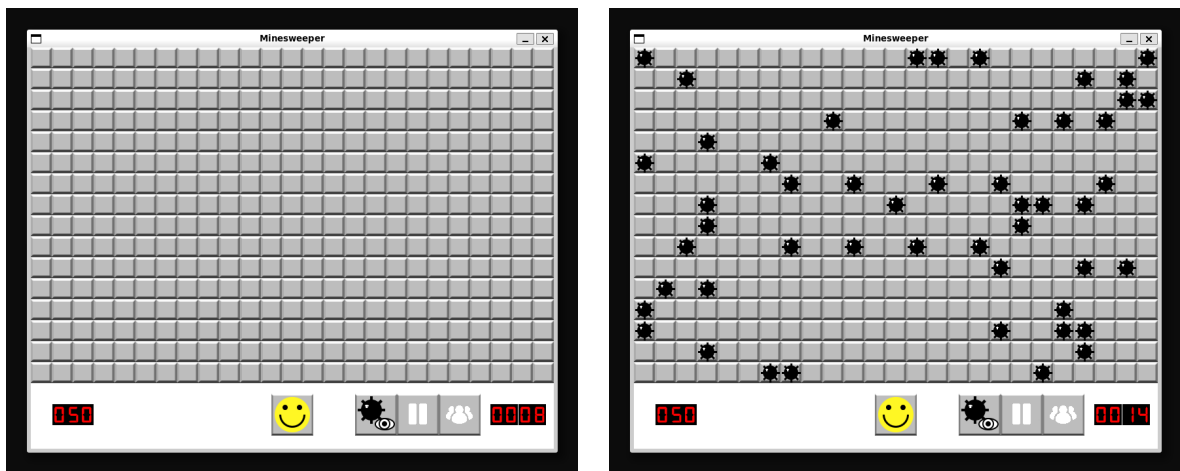


Figure 6.2: Example of the Debug button functionality; Left: Before pressing debug button; Right: After pressing debug button;

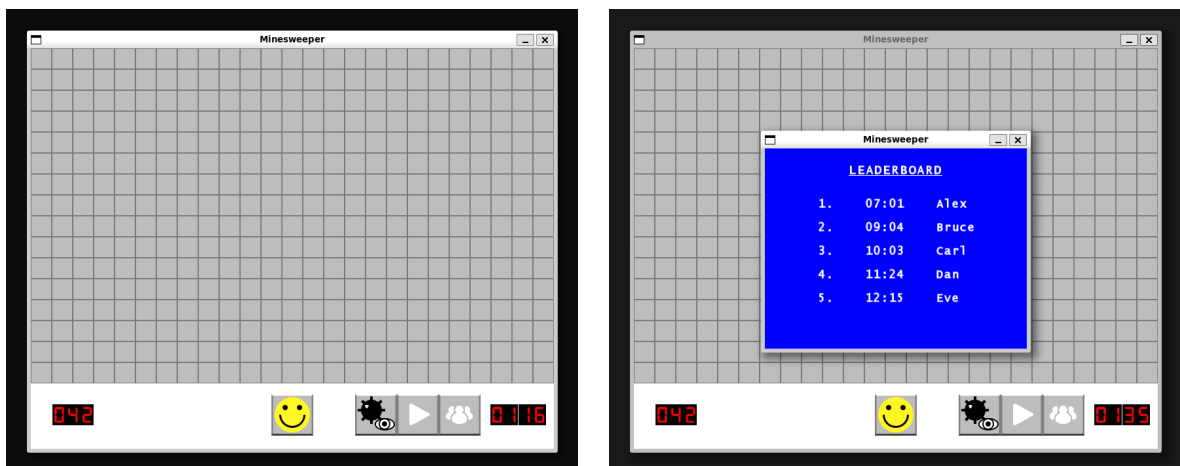


Figure 6.3: Example of the Paused Game Window; Left: The window when the pause button is pressed; Right: The window when the leaderboard button is pressed; (Even when debug mode is active, all tiles should display `tile_revealed.png` sprite)

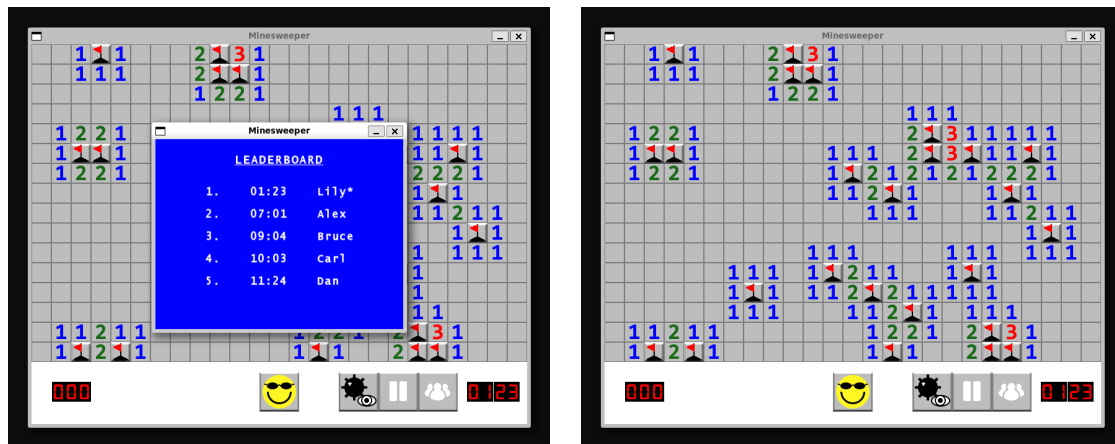


Figure 6.4: Example of the Victory Condition; Left: The leaderboard window pops up when the game is won; Right: The game board after the victory condition is met; Note: This is a 25x16 board and has 25 mines!

6.4 Related Concepts

Here are some of the concepts you will be using for this window:

Class/Function	Description
<code>std::chrono</code>	<p>It is a part of the C++ standard library that provides support for date and time manipulation, including clocks, timers, time points, and durations. You will use this for implementing the timer functionality.</p> <ul style="list-style-type: none"> <code>chrono::high_resolution_clock::now()</code> is a C++ function that returns the current time point of the high-resolution clock. <code>chrono::duration_cast</code> is a C++ function that converts a duration value from one unit to another, using the rules of the specified duration type. It allows you to convert between different time units, such as seconds, milliseconds, and microseconds, with precision and type safety.
<code>sf::Texture</code>	It is an object that represents an image loaded into memory and allows you to create sprites that display the image. It is a fundamental class for working with graphics in SFML.
<code>sf::Sprite</code>	It is a 2D object that displays a texture or a portion of a texture on the screen. It allows you to manipulate the position, scale, and rotation of the texture and is commonly used to display game characters and objects.
<code>sf::Rect</code>	It is a simple rectangle class that represents a rectangular area of pixels, defined by its top-left corner and its width and height in pixels. It is often used to specify a portion of a texture that should be displayed by a sprite.
<code>sf::Vector2</code>	It is a class in SFML that represents a 2D vector with floating-point coordinates.
<code>sf::Mouse</code>	It is a static class that provides access to the current state of the mouse, such as its position, buttons, and wheel and is used to handle mouse input.
<code>sf::Event</code>	It is an SFML data structure that represents user or system events, like mouse clicks or key presses, and enables you to handle them in your program. You will be using events like <code>"sf::Event::Closed"</code> and <code>"sf::Event::MouseButtonPressed"</code> .

Table 6.3: Related concepts for the game window.

6.5 Additional Notes

6.5.1 Config File

First, you will be working on creating the board. For that, you will be reading a text document `files/config.cfg`. There are three lines representing the number of columns, the number of rows, and the number of mines, respectively. You read these values and create a board of the given dimensions. These values will not be changed during a game. Also, the dimensions will never be less than 22 columns x 16 rows, so that all the menu buttons will be able to fit with the face-centered and the buttons space correctly. The number of mines will be a valid number less than or equal to the number of tiles. We will provide an example, but we will test with different values. Feel free to modify the config file to test out things. For example, setting the "mine count" value to 1 in the config file will make it really easy to test.

```
1 25
2 16
3 50
```

Listing 6.1: Example config file.

For example, if your game was given the config file in Listing 6.1, then the width of the welcome and game window is $25 \times 32 = 800$, the height is $(16 \times 32) + 100 = 612$, mine count is 50, and the tile count is $25 \times 16 = 400$.

6.5.2 Buttons

A button is 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.

6.5.3 Mouse Interactions

The application can do "nothing" until the user clicks their mouse. Typically, games and many other applications clear/redraw the screen regularly (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. You won't see anything moving other than the timer.

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.

6.5.4 Adjacent Mines and Tiles

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

6.5.5 UI Locations

Here is a table with all positions for the buttons and other UI components. Use these values as they have been tested to work on all board dimensions.

Component	Position
Happy Face Button	$\left(\frac{\text{number of columns}}{2.0} \times 32\right) - 32, 32 \times (\text{number of rows} + 0.5)$
Debug Button	$(\text{number of columns} \times 32) - 304, 32 \times (\text{number of rows} + 0.5)$
Pause/Play Button	$(\text{number of columns} \times 32) - 240, 32 \times (\text{number of rows} + 0.5)$
Leaderboard Button	$(\text{number of columns} \times 32) - 176, 32 \times (\text{number of rows} + 0.5)$
Counter	Start drawing the digits from $(33, 32 \times (\text{number of rows} + 0.5) + 16)$. For a negative counter, draw the '-' sprite in digits.png at location $(12, 32 \times (\text{number of rows} + 0.5) + 16)$. All digits are 21px wide, draw the next digit sprite after 21px.
Timer (Minutes)	Start drawing two digits from $((\text{number of columns} \times 32) - 97, 32 \times (\text{number of rows} + 0.5) + 16)$.
Timer (Seconds)	Start drawing two digits from $((\text{number of columns} \times 32) - 54, 32 \times (\text{number of rows} + 0.5) + 16)$. All digits are 21px wide, draw the next digit sprite after 21px.

Table 6.4: Positioning of Various Components

7 Leaderboard Window

7.1 Objective

The Leaderboard window, appearing when the "Leaderboard" button is clicked or after a game victory, displays the top player scores. This window fetches data from the `leaderboard.txt` file, listing players' names and their scores in a formatted manner. The file contains a list of comma-separated values: the first value contains the user's play time, and the second contains their name.

```
1 07:01,Alex
2 09:04,Bruce
3 10:03,Carl
4 11:24,Daniel
5 12:15,Eve
```

Listing 7.1: The default configuration of `leaderboard.txt` you are provided with.

You need to read the `leaderboard.txt` file and display its contents according to the format shown in Figure 7.1. Do note that you will be also printing the ranks on the left-hand side as shown in the figure.

7.2 Behavior

Your window should have the following behavior:

- If the player clicked on the "Leaderboard" button, all you have to do is display the contents of the file.
- If the player just won, you need to check if their time beat any other time in the list. If so, you will need to insert the player details (name and time) into the leaderboard. Indicate this change by displaying the new entry with an asterisk (*) after the player's name. You also need to update `leaderboard.txt` with the new record. Remember to only keep the top five players.
- When the leaderboard window is open, it appears just above the actual game window.
- If the leaderboard window is open, you cannot interact with the game window.
- Unlike in other games, the ranking system here is independent of the number of rows, columns, and mines. No matter how you modify the config file, you can update the same `leaderboard.txt` file.



Figure 7.1: Example of the Leaderboard Window.

Left: The window when the player clicks the leaderboard button

Right: The window when the player overtakes someone, after winning the game.

7.3 Related Concepts

Here are some of the concepts you will be using for this window:

Class/Function	Description
sf::Font	You will need to load a font before you can display text.
sf::Text	This will be used to draw text onto the screen. It will allow you to change a lot of things like the text position, size, style (bold/underlined/italic), etc.
sf::Color	It is a simple class that represents an RGBA color. It allows you to define colors for drawing shapes, sprites, and text in your SFML application. You will mainly use this to set the text and background color.
sf::Rect	It is a class in SFML that represents a rectangular area with floating-point coordinates, defined by its top-left corner and its width and height in pixels. You can use this to store the local bounds of a <code>sf::Text</code> , which is helpful in setting the text to the center.
sf::Vector2	It is a class in SFML that represents a 2D vector with floating-point coordinates.
sf::Event	It is an SFML data structure that represents user or system events, like mouse clicks or key presses, and enables you to handle them in your program. You will be using events like <code>"sf::Event::Closed"</code> .
File I/O	In C++, file I/O operations are performed using the standard library classes and functions, such as <code>std::fstream</code> and <code>std::ifstream</code> for reading from files, and <code>std::ofstream</code> for writing to files.
std::ostringstream	Allows you to read and write formatted data to and from strings.
std::string::substr	<code>substr()</code> is a member function of the <code>std::string</code> class that allows you to extract a substring from a string, defined by a starting index and a length or by a starting index and an ending index.

Table 7.1: SFML concepts you will be using for the Leaderboard Window.

7.4 Formatting and Output

When displaying text in the leaderboard window, you should continue to use the same font family provided to you (ie, `files/font.ttf`). Table 7.2 has some more necessary characteristics of the text you will be displaying.

Text Object	Bold	Underlined	Color	Size	Position
"LEADERBOARD"	Yes	Yes	White	20px	$\frac{\text{width}}{2}, \frac{\text{height}}{2} - 120$
Content of <code>leaderboard.txt</code>	Yes		White	18px	$\frac{\text{width}}{2}, \frac{\text{height}}{2} + 20$

Table 7.2: Properties of Text Elements in the Leaderboard Display

It is recommended to store all the rows in a string where each entry is separated by two new line characters (`\n\n`). Also, you can separate each value in a row with a tab character (`\t`). Once you have your string, you can proceed with making your `sf::Text` instance. As in earlier sections, remember to use the `setText()` method given to you in Listing 5.1 to ensure the text is center-aligned.

8 Implementation Tips

8.1 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 a `container["NameOfAsset"]` is vastly preferable to that of dealing with arrays—was “GameOver.png” stored in an `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.

8.2 Global Variables

In this project, you should not, under any circumstances, attempt to use `sf::Texture` and `sf::Font` objects in global space. Globals in general should be avoided, but if you try to load a `sf::Texture`/`sf::Font` 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 be on someone else’s (i.e. the person grading your project).

8.3 Paths

Here’s the "files" folder structure you will be provided with:

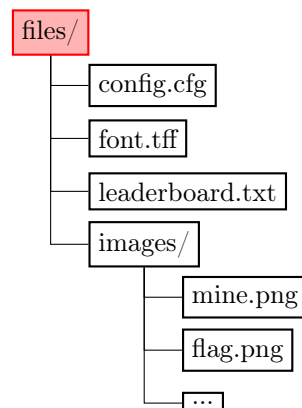


Figure 8.1: The appropriate tree structure for your files.

In this project, any operations involving files (loading textures and boards) should use relative paths, not absolute paths. Your code should be based on the folder structure shown above. When you load a texture, it should be from the images folder, like “files/images/mine.png”, and when you load the config file, it should be from `files/config.cfg`.

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

- “files\\images\\mine.png” – this will work on Windows, but not elsewhere. Don’t use this.
- “files/images/mine.png” – this will work everywhere. Do use this!

8.4 Managing Textures

Textures need to be kept alive in some sort of data structure to ensure that their memory isn’t deleted. Because sprites just store a pointer to a texture, **if the texture falls out of scope, its memory will be erased and the sprite will appear as a white square.**

Therefore, it’s recommended to develop a class or data structure to manage your textures. This class would store dynamic memory for each texture so that you can quickly access it throughout your program. You can store these textures in a map, where the name of the file will lead to the appropriate SFML texture.

8.5 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 than all others. From the outside perspective (i.e., that of a player), your application needs to DO various things. 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. 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.
- You can have a separate Texture Manager class for loading textures and getting them at any point in time.
- 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.
- Separate classes for the implementation of the Welcome window and the Leaderboard window.

9 Milestones

In order to space out your progress for this assignment, you will be required to complete a series of milestones. These milestones will be checked by your peer mentor. You will be required to show your peer mentor your progress on the milestones during your weekly meeting. You will not be required to submit anything to Gradescope for these milestones.

Milestones are not optional or extra credit. You must complete the specified milestone by the specified date, or you will be unable to earn a portion of the points for this project.

9.1 Milestone 1: Due April 12 (5 points)

For this milestone, you must show your peer mentor that you have installed SFML successfully on your own machine. You can submit an image of the circle and your code side-by-side to Canvas to earn 5 points for this milestone.

In order for you to generate the window shown in 9.1, run the code in Listing 9.1 (from the SFML website):

```

1  #include <iostream>
2  #include <SFML/Graphics.hpp>
3  int main()
4  {
5      sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
6      sf::CircleShape shape(100.f);
7      shape.setFillColor(sf::Color::Red);
8
9      while (window.isOpen())
10     {
11         sf::Event event;
12         while (window.pollEvent(event))
13         {
14             if (event.type == sf::Event::Closed)
15                 window.close();
16         }
17
18         window.clear();
19         window.draw(shape);
20         window.display();
21     }
22
23     return 0;
24 }
```

Listing 9.1: Code to demonstrate that SFML is installed correctly.

You should see a big red circle appear in your IDE, as shown in Figure 9.1.

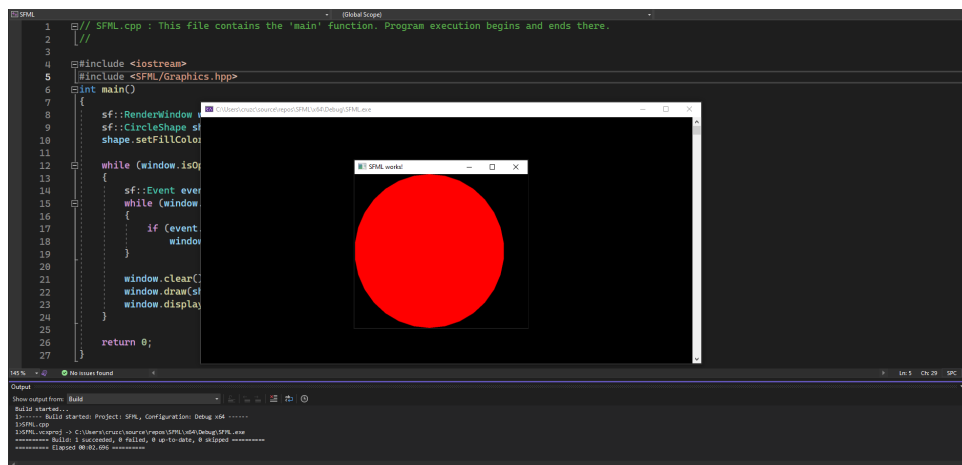


Figure 9.1: Milestone 1

9.2 Milestone 2: Due April 19 (15 points)

You will have until April 19 at 11:59 pm to show your peer mentor that you have already implemented at least one of the following behaviors from the following table. This submission can be made through Canvas via submitting a video showing your code and the behavior you have implemented.

Window	Necessary Implementation
Welcome Window	<ul style="list-style-type: none"> • Player can enter a name up to 10 characters (alphabet only, no numbers/special characters). • Characters appear as typed; backspace removes the last character. • Cursor is visible. • Names are formatted correctly. • Pressing "enter" closes the welcome window and launches the game window. • Closing the welcome window prevents the game window from opening. • The "enter" key has no effect if no characters are entered.
Game Window	<ul style="list-style-type: none"> • All tiles on the board are visually discernible. • Tile number and dimensions change according to the configuration file. • At least one tile shows correct functionality: <ol style="list-style-type: none"> 1. Clicking a number tile reveals the number. 2. Right-clicking a tile sets a flag. 3. Right-clicking a flagged tile removes the flag.
Game Window	<ul style="list-style-type: none"> • Tiles can be visually seen and revealed. • Mines are randomly assigned to tiles. • At least two buttons (debug and face) are implemented, including the restart game function.
Game Window	<ul style="list-style-type: none"> • All tiles are visible, with numbers and dimensions based on the configuration file. • The number of mines is displayed on the counter. • The counter updates to reflect the increase or decrease in mines.
Leaderboard Window	<ul style="list-style-type: none"> • The leaderboard reads from <code>leaderboard.txt</code> and displays its contents. • Closing the leaderboard window returns to the game window with the timer displayed. • The functionality of the timer is not required at this point.

Table 9.1: Summary of features to be implemented for Milestone 2.

Other combinations of features might be considered for this milestone; however, as a general, at least a 10-point and a 15 points item of the Milestone 3 rubric will be required for your to achieve full points in Milestone 2.

9.3 Milestone 3: Due April 24 (130 points)

For this milestone, you must submit a zip folder to Gradescope. This folder will have all .cpp and .h files. This project is manually graded, and you must ensure that your project complies with all rubric requirements before submitting your folder.

Item	Points	Description
<code>config.cfg</code>	10	Board changes size and its number of mines based on values set in <code>config.cfg</code> file.
Tile Revealing	15	Clicking on a tile reveals it. If it's 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 goes through this process as well. A single click could reveal nearly the entire board depending on the layout.
Tile Display	10	<p>Tiles display depending on their state:</p> <ul style="list-style-type: none"> • Unrevealed (default state). • Revealed and empty (no adjacent mines). • Revealed, near 1-8 mines (showing the appropriate number). • Revealed, showing a mine.
Flags	10	Right-clicking a hidden tile sets a flag. Right-clicking a flagged tile removes the flag. Left-clicking a flagged tile has no effect. Flagged tiles cannot be revealed unless the flag is removed first.
Mines Remaining	10	A counter shows how many mines are on the board minus the number of flags placed. Flags added or removed affect this count. The remaining flags can go negative.
Victory	10	Revealing all non-mine tiles ends the game. Mines are automatically flagged. Smiley face changes to sunglasses version. No further interactions with the board are possible. The player can start a new game or check the leaderboard. A player's winning time is compared to the leaderboard, and new records are indicated with a '*'.
Defeat	10	Clicking a mine ends the game. All mines are revealed. The smiley face changes to a dead face. No further game board interactions are possible. The player can start a new game or check the leaderboard. Pause-play buttons become inactive.

Table 9.2: Detailed Rubric for Milestone 3 Part I

Feature	Points	Description
Random Mine	10	At game start and when resetting (via smiley face button), the specified number of mines from the .cfg file are randomly placed on the map.
Welcome Window	15	<p>Player name entry requirements include:</p> <ol style="list-style-type: none"> 1. Limit to 10 characters (alphabet only, no numbers/special characters). 2. Visible characters as typed; backspace removes the last character. 3. Visible cursor. 4. Names converted to the right format when typing (not just when stored). 5. Pressing "Enter" closes the window and launches the game. 6. Closing the welcome window prevents game window opening. 7. "Enter" key has no effect if no characters entered.
Timer and Pause Button	15	The timer functions correctly. Clicking the pause button pauses the counter and shows all tiles with <code>tile_revealed.png</code> . Only leaderboard and face buttons work in pause mode. Clicking play resumes the game, reverts tiles to prior states, and continues the counter.
Leaderboard and Button	15	Clicking the leaderboard button shows the top 5 leaders. This action pauses the timer and changes all tiles to <code>tile_revealed.png</code> . Closing this window returns to the game with tiles in their prior states (including the paused mode).
Total	130	

Table 9.3: Detailed Rubric for Milestone 3 Part II

10 Submission

You will turn in your source code and a `README.md` file only. No images, no SFML libraries, only the `.h` and `.cpp` files you wrote to complete the project and a `README.md` file with the following information:

README.md

```

1 Name:
2 Section:
3 UFL email:
4 System:
5 Compiler:
6 SFML version:
7 IDE:
8 Other notes:

```

Note: Zip up your code along with this above file filled with all the information. Please use the following folder name: **lastname_firstname_project3_Spring2024**

Include all fields in the `README.md` file, if you have nothing to mention in "Other notes", just type "None".

Then submit this zip folder to Gradescope. This project is manually graded, and you must ensure that your project complies with all rubric requirements before submitting your folder.

! If the name or your folder does not comply with the naming convention, you will have a -10 deduction in your project.

11 Grading

In general, the features you need to implement for this project are listed in the feature summary in the Milestone 3 section 9.2 and 9.3, along with point values. You can get full, half, or no points for each feature.

- **Full points:** Feature works perfectly, no bugs of any kind
- **Half points:** Feature has any bugs at all or is partially implemented (if your program is slow, you might get deductions here)
- **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)

The general breakdown of the Project three grade can be found in 11.1

Standard Points		
Item	Points	Description
Milestone 1	5 points	Successful completion of Milestone 1.
Milestone 2	15 points	Successful completion of Milestone 2.
Milestone 3	130 points	Successful completion of Milestone 3.
Total		150
Deductions		
Using Global Variables	-10	Using global variables (including <code>sf::Texture</code> objects).
Not Using Relative Paths	-10	Not using relative paths for file referencing.
Unrequested Files	-10	Existence of files in the submitted folder that were not requested.
Missing README.md	-10	The README.md file was not included in the submission.

Table 11.1: Rubric for the Entire Project

A Sample Makefile

This section is completely optional, and only serves as a reference for those who want to build their project via make. You do **not** need to use this Makefile, write your own Makefile, or submit a Makefile as part of your project.

To execute your program via terminal, you can use the following Makefile:

```

1 build:
2     g++ src.cpp \
3     -I<PATH_TO_INCLUDE> \
4     -L<PATH_TO_LIB> \
5     -lsfml-graphics \
6     -lsfml-window \
7     -lsfml-system \
8     --std=c++11 \
9     -o sfml-app
10
11 # This is an example of what the build rule might look like on a Mac, with the relevant
12 # paths filled in:
13 example:
14     g++ src/*.cpp \
15     -I/opt/homebrew/Cellar/sfml/2.6.1/include \
16     -L/opt/homebrew/Cellar/sfml/2.6.1/lib \
17     -lsfml-graphics \
18     -lsfml-window \
19     -lsfml-system \
20     --std=c++11 \
21     -o sfml-app

```

Listing A.1: Makefile to compile your project through the command-line.

Once you have the Makefile in your project directory, navigate to the directory in your terminal and use the appropriate Make command based on your environment:

- For Unix-based systems (such as Linux or macOS), use the `make` command.
- For Windows systems with Microsoft Visual Studio installed, use the `nmake` command.
- For Windows systems with MinGW installed, use the `mingw32-make` command.

After running the appropriate Make command, an executable file named `project3.out` will be generated. You can then run this file to start the game. Note that this Makefile assumes that all your source files are located inside the "src" folder.

B Document Revisions

Table B.1 shows the edits have been made to the document since its initial release.

Revision	Date	Section	Changes
2	April 10th, 2024	subsection 3.2	Added alternate SFML installation process and associated video (cred: Alex Johnson, Justin Lopez)
		subsection 5.2, subsection 6.4, subsection 7.3	Fixed links to <code>sf::Vector2f</code> , <code>sf::IntRect</code> , and <code>sf::IntRect</code> , which are now template classes. (cred: Caleb Dalton)
		subsection 8.4	Added a reference to the "white square problem".
		subsection 9.1	Milestone 1 can be submitted through Canvas. Removed questionnaire requirement. (cred: Jason Lin)
		subsection 9.2	Milestone 2 can be submitted through Canvas. Removed questionnaire requirement. (cred: Jason Lin)
1	April 8th, 2024	Appendix A	Created section by moving Makefile from earlier in the document. Clearly stated that Makefile is not a requirement. (cred: Gabby Taboada)
		—	Initial release.

Table B.1: Changes that have occurred in the document since its initial release.