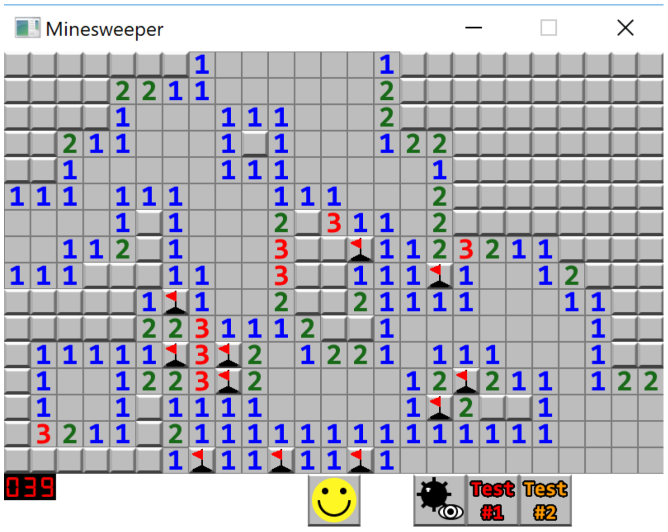
UNIVERSITY of FLORIDA
Herbert Wertheim
College of Engineering

COP3504C

P4: Minesweeper

Overview

This project will provide students with practice building and working within an object-oriented by building classes to represent the elements of the classic PC game "Minesweeper" and implement a working version.



Gameplay

Many free variants of Minesweeper are out there. It's a good idea to play and become familiar with the game!

The goal is to reveal all and *only* tiles that are free of mines. The board is made up of a grid of tiles, each of which may or may not have a mine hidden underneath. When the player left clicks on a tile, it is revealed.

- Right clicking a tile displays a flag on the tile to mark it as a possible mine
- If a mine tile is revealed, the player loses / game ends. If all non-mine tiles are revealed, the player wins.
- If non-mine tile is revealed, it displays how many mines adjacent to that tile (1-8, or empty for 0)
    - If there are no adjacent mines, all non-mine tiles adjacent to the tile are also revealed
- The player uses the numbers as clues to deduce where other mines are located

Note that *flagged spaces cannot be revealed*, but right clicking again removes the flag.

Mine Counter

The counter tracks the number of mine remaining. When a flag is placed, counter decreases by one (whether or not it is placed on a mine). Whenever a flag is removed, the counter increases. *This can be negative*!

Buttons

New Game – restarts the game – including randomly reassigning mines
Debug Mines – toggles visibility of mines on the board. This facilitates testing / debugging
Test Buttons – load predefined test boards (detailed in later sections)

Structure

This section outlines important structural elements the facilitate implementation of the specification.

Window Management

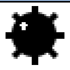
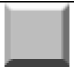
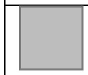
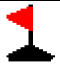
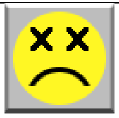
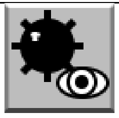
The SFML window must be `800x600` and use this title bar text: "`P4 – Minesweeper, <Student name>`". The window must be able to be closed via the ✕ button in the upper-right corner.

Event Management

Like many interactive toolkits, SFML uses events to signal input from the user. Events must be captured and processed via polling and filtering.

Images

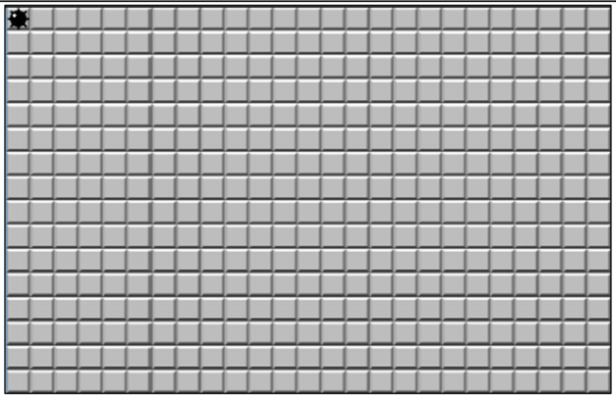
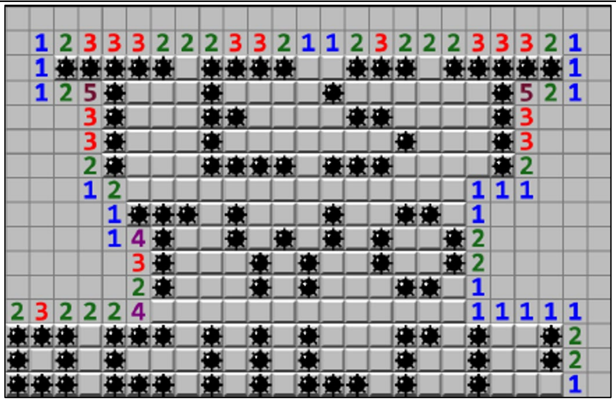
All images needed for this project can be found in the `images` folder. *Do not submit this folder.* Images should be loaded as `sf::Texture` objects and used to create `sf::Sprite` objects. Images are as follows:

Game Images		
	mine.png	Star of the game (but if played properly, we'll never see one!)
	tile_hidden.png	An unrevealed tile
	tile_revealed.png	A revealed tile with no adjacent tiles
	number_#.png	Tile digits (where **#** is replaced with 1:8) for adjacent mines
	flag.png	Drawn over tiles when flagged by player as possible mines
UI Images		
	face_happy.png	Button to reset/start new game; new mines, nothing revealed. The new state should be a default game state (25x16 with 50 randomly placed mines).
	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	Digits for the mine counter. The size of each digit is 21 x 32 pixels. (**Hint**: can use just one texture, but several sprites!)
	debug.png	Toggle mine debugging mode
	test_1.png	Loads a test file from which the board will be set
	test_2.png	Loads a test file from which the board will be set

Board Files

Pre-generated game boards can be loaded from board files (`brd`) stored in plain text as a sequence of digits, where **`‘0’`** represents an empty tile and **`‘1’`** represents a mine. This can be used to build and test configurations.

`1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
	`0 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 0 0`
	`0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0`
	`0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0`
	`0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0`
	`0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 1 0 0 0 0 1 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
	`0 0 0 0 0 1 1 1 0 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 1 0 0 1 0 0 0 0 0 0`
	`0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 0 0`
	`0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0`
	`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
	`1 1 1 0 1 1 1 0 1 0 1 0 1 0 0 0 1 1 0 1 0 0 1 0 0`
	`1 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 0 1 0 0`
	`1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 0 1 0 0 0 0 0`

	
No revealed tiles	One time on edge revealed (causing cascade)

Requirements

Students use SFML v2.5.1 for this project. They will construct several classes and functions as part of this assignment. ***All attributes / methods must be private unless noted in the specification.***

Toolbox Class

A toolbox class is often used to contain variables that would otherwise be accessed throughout an application; it is a mechanism to avoid truly global variables. The **`Toolbox`** class will be a *singleton* (class with only one instance) and will contain *at least* the following attributes and methods:

```
public sf::RenderWindow window; // SFML application window
public GameState* gameState;    // Primary game state representation
public Button* debugButton;     // Reveals mines in debug mode
public Button* newGameButton;   // Resets / starts new game
public Button* testButton1;     // Loads test board #1
public Button* testButton2;     // Loads test board #2
```

`public static Toolbox&` **`getInstance()`**

Returns a reference to the singular `Toolbox` instance.

`private` **`ToolBox()`**

Default constructor; should be accessible only from within the class itself. This method initializes the buttons, window, game board, and any other elements necessary to play the game.

Button Class

This class will be used to implement each button *widget*. Widgets are responsible for rending themselves, originating interaction (e.g., clicking), and conveying state to the user. It will have these public methods:

`public Button(sf::Vector2f _position, std::function<void(void)> _onClick)`
Constructs a new object at the specified **`_position`** which invokes the **`_onClick`** callback when clicked.

`public sf::Vector2f getPosition()`
Returns the position of the button.

`public sf::Sprite* getSprite()`
Returns the current sprite of the button.

`public void setSprite(sf::Sprite* _sprite)`
Sets this button's visualization to the specified **`_sprite`**.

`public void onClick()`
Invokes the button's callback method (usually called when clicked).

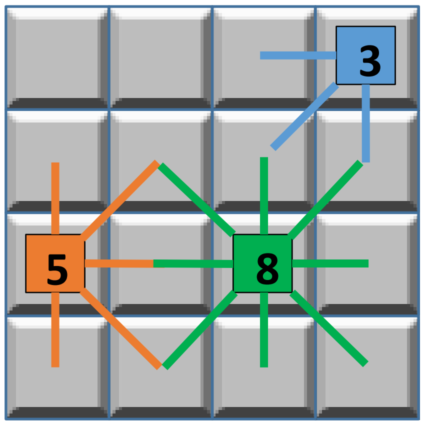


Figure 1. Mine configuration examples.

Tile Class

This class implements the tile widgets that make up the board. The class can be *optionally* subclassed to further encapsulate special traits. Each tile's neighbor configuration can vary (see Figure 1). When a tile is adjacent to the edge of the board, the neighbor pointer should be a **`nullptr`** value.

`public enum State { REVEALED, HIDDEN, FLAGGED, EXPLODED }`
Represents tile's current UI state (visualization).

`public Tile(sf::Vector2f position)`
Constructs a new tile object at the designated **`_position`**.

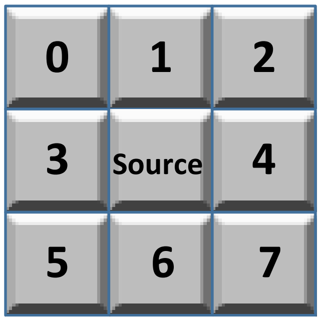


Figure 2. Neighbor ordering.

`public sf::Vector2f getLocation()`
Returns the position of this tile.

`public State getState()`
Returns current state of this tile.

`public std::array<Tile*, 8>& getNeighbors()`
Returns pointer to array of `Tile` pointers (see Figure 2 for ordering).

`public void setState(State _state)`
Sets the state of this tile. Should trigger other behaviors related to the state change (including visualization).

`public void setNeighbors(std::array<Tile*, 8> _neighbors)`
Populates / replaces the neighboring tile container.

`public void onClickLeft()`
Defines the reveal behavior for a tile when the left mouse button is clicked inside its boundaries.

`public void onClickRight()`
Toggles this tile's state between **`FLAGGED`** and **`HIDDEN`**.

`public void draw()`
Render this tile to the screen according to is state.

`protected void revealNeighbors()`
Based on State and mine content of the tile neighbors, set their state to **`REVEALED`**.

GameState Class

The `GameState` object should contain the `Tile` objects that represent the locations in the game and play status.

`public enum PlayStatus { WIN, LOSS, PLAYING }`
Tracks the play status of the game, which is reflected in the behavior of the user interface and visualizations.

`public GameState(sf::Vector2i _dimensions = Vector2i(25, 16), int _numberOfMines = 50)`
Constructs a new random game state with specified tile **`_dimensions`** and the specified **`_numberOfMines`**.

`public GameState(const char* filepath)`
Constructs a game state based on the size, mine placement, and mine number specified at **`filepath`**.

`public int getFlagCount()`
Current count of the number of flags placed on the screen.

`public int getMineCount()`
Current count of the number of mines actually on the board.

`public Tile* getTile(int x, int y)`
Returns a pointer to the **`Tile`** at the specified coordinates, or **`nullptr`** if out of bounds.

`public PlayStatus getPlayStatus()`
Returns the play status of the game.

`public void setPlayStatus(PlayStatus _status)`
Sets the play status of the game.

Global Functions

A proper implementation will also include the following functions in ***global scope*** (not in a class). Except for **`main()`**, these functions should be prototyped in the **`minesweeper.h`** header file:

`int launch()`
This method is invoked directly by **`main()`** and is responsible for the game's launch. It should be possible to directly invoke this function after including the submitted source and header files in the test suite.

`void restart()`
Resets all states/objects and generates a default game state (random board) and turns off debug mode if active. The new state should be a default game state (25x16 with 50 randomly placed mines).

`void render()`
Draws the all UI elements according to the current **`gameState`** and debug mode.

`void toggleDebugMode()`
Flips the debug mode on/off. (Debug mode should initially be off/**`false`**.)

`bool getDebugMode()`
Returns the **`true`** if debug mode is active, and **`false`** otherwise.

Students may add functions, as needed, to implement the program. For example, this function may be helpful:

`public int gameLoop()`
Encapsulates event-polling, rendering, and other logic necessary to handle input and output for the game.

Finally, the **`main()`** function should be defined exactly as follows:

```
int main() { return launch(); } // Just invokes launch()! (You can leave off the comment)
```

Submissions

NOTE: Your output must match the example output *exactly*. If it does not, ***you will not receive full credit for your submission***! (Note that matching sample output is necessary, but not sufficient, for full credit.)

Files: minesweeper.zip
Method: Submit on Canvas

Compressed Archive (minesweeper.zip)

We do not list required source files, only headers. You should include additional source or header files in addition to those listed – based on your design – but you must have the listed files at a minimum.

Your compressed file should have the following directory/file structure:

```
minesweeper.zip
  ↳ minesweeper (directory)
        → CMakeLists.txt
        → minesweeper.h
        → Buton.h
        → GameState.h
        → Tile.h
        → Toolbox.h
        → (Other sources / folders)
```

Helpful Links

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

Tips & Tricks

Here are some things to keep in mind as you work on this project:

- Most platforms will have some sort of documentation. ***Read it***. It's critical to get used to sifting through technical documentation to find answers.
- Don't be afraid to try things; when working with new toolkits and platforms, "playing" an "dabbling" help us understand these unfamiliar systems. Learning by doing (and failing!) is a critical part of problem-solving.
- Don't try to do everything at once. It's OK to hard code a few things and build a small piece, expanding as you go. For example, try to get one tile working, and one button; then a few buttons; etc.
- Think about what types of classes or functions we might want to add. How do we want to store the board data? What types of containers will we use? What will make our lives easier (or harder?)
- This is your task; Write in a way that makes sense *to you*. Everyone tackles problems differently – that's OK!

Using Documentation

The One True Answer to a problem might not be out there on the Internet, in a StackOverflow.com question, or on YouTube. However, the information to help us figure out PARTS of the problem is almost surely out there. We must find out how to make sense of the smaller bits of information and decide on a course of action.

For example, data in `sf::Texture` objects disappears when the object is deleted or falls out of scope. We can't create a local `Texture`, make a `Sprite` from it, return it from the function, and use it. Documentation helps here: https://www.sfml-dev.org/tutorials/2.5/graphics-sprite.php#the-white-square-problem