Lab 2 : A Graphical Tic-Tac-Toe

1 Objectives

The main objective of this assignment is to introduce you to the use of classes and objects, thus applying the related concepts presented in the lectures. You will do so by building a simple graphical Tic-Tac-Toe game using the object-oriented SFML multimedia library. In addition, the assignment will also introduce you to writing event-driven programs that respond to mouse clicks.

2 The Tic-Tac-Toe Game

Tic-tac-toe is a simple game commonly played by children. Two players, X and O, take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal line wins the game. Player X is always the first to place a mark. The following example shows the progression of a game won by player X.

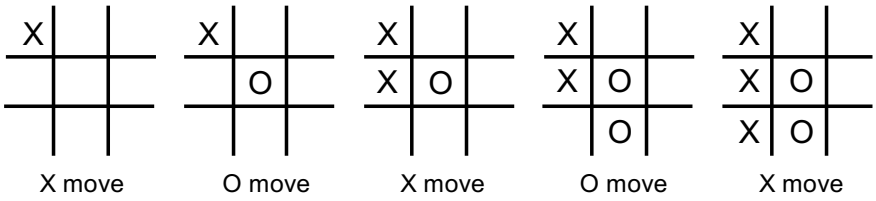

Figure 1: An example Tic-tac-toe game

The simplicity of the game makes it possible for each player to make a perfect move. Thus, the game often ends in a draw. A player wins only if the opponent makes a mistake (which is why the game is played only by children).

3 Problem Statement

You will write in C++ a graphical tic-tac-toe game. You will use SFML to display and update the game grid as two players take turns to click on empty cells in the grid. There is a working executable of the game (`tic-tac-toe-ref.exe`) in the lab material released on the course's web site. Thus, you can play the game yourself and use this executable to clarify the specifications of the assignment. If in doubt, run the game and see how it behaves. Please note that this executable will work only on ECF Linux machine (i.e., not Windows or Mac machines).

The remainder of this handout is organized as follows. Section 4 describes the code you must write to implement the "logic" of the game. Section 5 gives a quick overview of basic SFML graphics. Section 6 presents event-driven programming in SFML with some examples. Section 7 outlines the steps you must take to graphically display the tic-tac-toe game. Section 8 details how to compile and run your code. Finally Section 9 describes what you need to deliver and the marking scheme.

4 The Game

Implementing a tic-ta-toe game is relatively simple. The game grid is represented by a 3×3 two-dimensional array. It stores the marks of each player, and thus the state of the game. Each element of the array can be one of `Empty`, `X` or `O`. The array should initialized to `Empty`.

```
#define Empty  0
#define X      1
#define O     -1

int gameBoard[3][3] = {Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty};
```

In the array, `gameBoard[0][0]` represents the top-left corner cell of the game grid. Similarly, `gameBoard[2][2]` represents the bottom-right corner cell of the board.

Your main task in implementing the logic of the game is to write a function:

```
void playMove(  int board[3][3],      // The board 2-d array
                int row,              // Board row where mouse is clicked
                int col,              // Board column where mouse is clicked
                bool& turn,           // Whose turn is it?
                bool& validMove,      // Is the move valid?
                bool& gameOver,       // Is the game over?
                int& winCode,         // A code of winning sequence
              );
```

This function is to be implemented in the file `playMove.cpp` and it is called every time a player makes a move. It's goal to "play" the move and update the array representing the grid, among other variables. The `playMove` function takes the following parameters:

- `board` is the two-dimensional array representing the game board. Thea array is both input to and output of the function.
- `row` and `col` indicate the board row and column where the mouse is clicked. They are only inputs to the function (hence, they are passed by value).
- `validMove` is a Boolean output of the function and it should be set to true if the move is valid (i.e., the grid cell at `row` and `col` is empty) and to false otherwise. Note that since the variable is an output of the function, it is passed by reference.

  If the move is valid, the function should update the `board` array at the appropriate location.
- `turn` is a Boolean variable that indicates whose turn it is, X (true) or O (false) upon entry to the function. If the move is valid, then the value of `turn` should change from true to false or from false to true to reflect the change in turn.
- The `gameOver` is a Boolean output of the function that should be set to true if the game is over as a result of this move (win or draw) and to false otherwise.

- `winCode` is integer variable is set to a code that indicates which cells on the board have marks that form a line, as shown in Table 1. If `gameOver` is false, the code should be set to 0. If `gameOver` is true and the game is a draw, then `winCode` should also be set to 0. if `gameOver` is true and one of the players won, the code should be set to one of the integer values as indicated in the table. The main purpose of this code is to simplify drawing a line in the graphics part of the assignment.

Code	Sequence
0	No win
1	Row 0 of the grid, cell (0,0) to cell (0,2)
2	Row 1 of the grid, cell (1,0) to cell (1,2)
3	Row 2 of the grid, cell (2,0) to cell (2,2)
4	Column 0 of the grid, cell (0,0) to cell (2,0)
5	Column 1 of the grid, cell (0,1) to cell (2,1)
6	Column 2 of the grid, cell (0,2) to cell (2,2)
7	Left to right diagonal, cell (0,0) to cell (2,2)
8	Right to left diagonal, cell (2,0) to cell (0,2)

Table 1: `winCode` values

Upon completion, the function must print the following to `cout` on **single line**:

- The contents of the game board, printed one row after the other (i.e., the first row followed by the second row followed by the third row).
- The values of the `row` and `col` parameters.
- The output `turn` value.
- The output Boolean variables `validMove` and `gameOver`.
- The winning sequence code.

The values printed should be separated by single spaces. The two game grid examples shown in Figure 2 can be used to demonstrate the expected output.

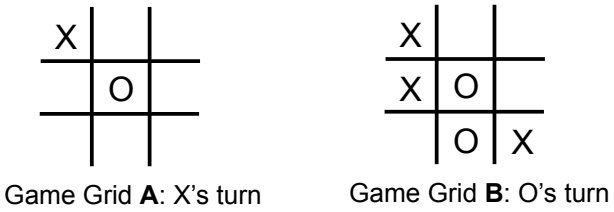

Figure 2: Input game boards

For game grid **A** on the left, the `board` array contains `1 0 0 0 -1 0 0 0 0` upon entry to the function. This sequence represents the first row of the array, followed by the second row and finally the third row. The `turn` variable is true (or `1`) indicating that it is X's turn to play.

If `row = 1` and `col = 2` when the `playMove` function is called, then an X is placed in the second row and third column of the grid. The move is valid but the game is not over. Thus, the output printed by `playMove` is:

```
1 0 0 0 -1 1 0 0 0 1 2 0 1 0 0
```

Notice that the output indicates how the `board` array has changed to reflect X's move and how the `turn` variable has changed from true to false to indicate that it is now O's turn to play.

However, if `row = 1` and `col = 1` when the `playMove` function is called, then the move is not valid (O's mark already occupies the cell). The `board` array is not affected and the output is:

```
1 0 0 0 -1 0 0 0 0 1 1 0 0 0 0
```

Similarly, for game grid **B** on the right, the `board` array contains `1 0 0 1 -1 0 0 -1 1` upon entry to the function. The `turn` variable is true (or `0`) indicating that it is O's turn to play. Thus, if `row = 0` and `col = 1`, the output printed by the function is:

```
1 -1 0 1 -1 0 0 -1 1 0 1 1 1 1 5
```

Notice how the output now indicates that the game is over and gives the code as 5, consistent with Table 1.

5 Basic Elements of SFML Graphics

Simple and Fast Multimedia Library (SFML) is a library that provides a simple application programming interface (API) for multimedia software development. In particular, it handles the creation of windows, detection of events such as keyboard presses and mouse clicks, displaying images and playing of music and sounds. SFML is written in C++ but has bindings for various other languages, such as C, Java, Python and Ruby. It runs under Windows, Linux, macOS, Android and iOS. In this course, we will use the Linux version 2.4.2. You are free to download the library to your own computer.

The remainder of this section provides a very quick and brief tutorial on how to use SFML to create windows, display images and text, draw shapes and play sounds. It should suffice for you to complete the assignment. However, it is easy to get more documentation and plenty of examples on SFML at:

`https://www.sfml-dev.org/learn.php`.

5.1 Creating Windows

The code snippet below shows how to create a window object called `myFirstWindow`. It is of the type `sf::RenderWindow` and is constructed as, or initialized to be, a video window of 800 × 600 pixels. The window has the title `ECE244 Window`. It has two attributes specified: a title bar and a close button. Absent are attributes that make the window resizable, and hence this window is not.

```
// Create an 800x600 pixels window: it has a title bar and a close
// button, but is not resizable
sf::RenderWindow myFirstWindow(VideoMode(800, 600), "ECE244 Window",
                               Style::Titlebar | Style::Close);
```

When is window is minimized, an icon is displayed on the task bar. Thus, the following code load an image and makes it the icon for the window. The code first declares a variable called `myFirstWindowIcon` of type `Image` from the `namespace` called `sf`. It then loads an actual `jpg` image from the file `icon.jpg`, assumed to be in the working directory. If the load fails, an error code `EXIT_FAILURE` is returned. Once the image is loaded, it is set as the icon for the window, as shown.

```
sf::Image myFirstWindowIcon;
if (!myFirstWindowIcon.loadFromFile("icon.jpg")) {
    return EXIT_FAILURE;
}
myFirstWindow.setIcon(myFirstWindowIcon.getSize().x,
                myFirstWindowIcon.getSize().y,
                myFirstWindowIcon.getPixelsPtr());
```

Once a window is created, one can *draw* in this window using the `draw()` method of the `RenderWindow` class. Drawing does not immediately display what is drawn in the window. It rather only renders to the window internal buffer. In order to actually display what is drawn the `display()` method of the `RenderWindow` class must be used. Thus, a typical usage scenario to create the various shape, image and text objects to be displayed to the window, is to first use the `draw()` method to draw each object and then at the end use the `display()` method to display them all.

5.2 Drawing Images

In order to draw an image, the image is first loaded into a *texture*, then the texture is used to create a *sprite*. The sprite is then drawn to the window.

```
sf::Texture xTexture;
if (!xTexture.loadFromFile("x_image.jpg")) {
    return EXIT_FAILURE;
}
sf::Sprite xSprite(xTexture);

// Draw the image
myFirstWindow.draw(xSprite);

// Now actually update the display window
myFirstWindow.display();
```

It is possible to position the sprite in the window. This is accomplished using `the setPosition` method, which specifies the `x` and `y` pixel positions to position the top left corner of the sprit at. The origin is the top left corner of the window. The `x` axis is the horizontal, increasing to the right. The `y` axis is the vertical, increasing towards the bottom. For example,

```
xSprite.setPosition(15,30);
```

sets the top-left corner of the sprite at 15 pixels from left edge of the window and 30 pixels from the top edge of the window.

The same sprite can be drawn multiple times at different positions in the window. A sprite may also be scaled and rotated. Please refer to SFML documentations for details.

5.3 Drawing Shapes

A number of shapes, such as rectangles, circles, triangles, etc. can be created and drawn with SFML. For example, a rectangle can be created using the `sf::RectangleShape` class. Its constructors takes two-element vector that specify the width and height of the rectangle. Thus, the following code creates a rectangle shape whose width is 200 pixels and height is 50 pixels.

```
sf::RectangleShape myOwnRectangle(sf::Vector2f(200, 50));
```

After the shape is created, its size can be changed. Further, its position, and orientation in the window can be set. The following code shows some example.

```
// Change the size
myOwnRectangle.setSize(sf::Vector2f(10, 50));

// Set the position
myOwnRectangle.setPosition(10,60);

// Set the orientation in relation to the x axis
myOwnRectangle.rotate(-45);
```

You can also set the fill color of a shape. For example:

```
// Set the fill color to white
myOwnRectangle.setFillColor(sf::Color::White);

// Set the fill color to black
myOwnRectangle.setFillColor(sf::Color(0, 0, 0)); // RGB = 0 0 0, i.e., black
```

You may want to see the SFML documentations for other properties of shapes as well as other shapes that are available.

Once a shape is created is its properties are defined, it can be drawn to the window and then displayed. For example, the following code draws then displays the rectangle defined above.

```
// Draw the rectangle
myFirstWindow.draw(myOwnRectangle);

// Now actually update the display window
myFirstWindow.display();
```

6 Event-Driven Programming

Event-driven programming refers to a programming paradigm in which the flow of a program is dictated by *events* external to the program. It is commonly used for graphical user interfaces where a program is written to respond to user actions, such mouse clicks or keyboard presses.

An event-driven program typically has a loop that "listens" for events. Once an event occurs, the type of event is determined and execution flows to a code segment that responds to or handles the event. Thus, this segment of code is often referred to as the *event handler*. The remainder of this section describes how events are handled in SFML, focusing mostly on mouse events.

The main loop that listens for events often looks like:

```
while (myFirstWindow.isOpen()) {
    // The event
    sf::Event event;

    // Process the events
    while (myFirstWindow.pollEvent(event)) {
        // Handle the events
        :
        :
    }
}
```

The outer `while` loop iterates as long as the window `myFirstWindow` is open. An object of type `sf::Event` is defined. The method `myFirstWindow.pollEvent` checks for events and if there is at least one event, it loads the next event into the object `event` (passed by reference to the method) and returns true. Thus, the inner `while` loop iterates as long as there are events to process. The body of this loop handles the events, one at a time.

The handling of the events is done by checking for the type of the event and taking the appropriate action. For example, the following code snippet handles the Escape key press, which is intended to close the window.

```
if (event.type == Event::KeyPressed &&
    event.key.code == Keyboard::Escape) {
    myFirstWindow.close();
}
```

Thus, if the event is that a key is pressed (event.type is `Event::KeyPressed`) and if the key pressed is the `Escape` key (`event.key.code == Keyboard::Escape`) then the window is closed. It is also possible to detect when the window is closed by the user, as follows.

```
if (event.type == Event::Closed) myFirstWindow.close();
```

A mouse button press can be similarly detected and handled. In the code below, the method `Mouse::isButtonPressed` returns true is the event is that a button of the mouse is pressed. The argument to the method (`Mouse::Left`) indicates button is of interest, the let one. There is a similar method for detecting when the button is then released.

Often, it is desired to determine where the cursor was when the button is pressed. This can be determined using the `getPosition` method. It returns a 2-element vector containing the `x` and `y` pixel coordinates at which the mouse was clicked. You can then use these coordinates as needed by your code.

```
if (Mouse::isButtonPressed(Mouse::Left)) {
   // left mouse button is pressed
   // Get the coordinates in pixels.
   sf::Vector2i localPosition = Mouse::getPosition(window);

   // The Vector2i is a type defined in SFML that defines a
   // two element integer vector (x,y). This is how the
   // elements of the vector are accessed
   int xPosition = localPosition.x;
   int yPosition = localPosition.y;

   // Important to keep in mind that the x axis is the
   // horizontal one (i.e., columns) while the y axis is
   // the vertical one (i.e., rows)
   :
   :
}
```

7 The Graphical Tic-Tac-Toe

You will implement the code for the graphical part of the game in the file `main.cpp`. A skeleton of the code you must implement is in the `main.cpp` file contained in the released `zip` file. The file contains comments that will guide you towards what you have to do. In summary, your `main` function must:

- Declare and initialize the various variables that represent the state of the game. Most of them are already declared for you in the skeleton code.
- Create a window and initialize its icon (see Section 5.1.
- Load the images representing a blank cell as well as the X's and the O's into textures and then create sprites for them (see Section 5.1).
- Create rectagular shapes to represent the borders of the cells. A line is a rectangle that is "thin" in one dimension.
- Write an event handling loop that monitors mouse click and window close events (see Section 6).
- When a left mouse button click even is detected, determine the pixel coordinates the mouse is clicked at and translate them into row and column values on the game grid. Mouse clicks outside the window or not inside a cell of the grid should be ignored.
- If the mouse click is on the window close icon, close the window.

- Call the `playMove` function, passing to it the appropriate variables.
- Examine the results of the output variables and then re-draw the **entire** board again to reflect the move that was just played. If the game is over and one player wins, you must draw a line across the marks based on the `winCode`. The program should just stop accepting mouse clicks after this. The same happens if the game ends in a draw.

8 Procedure

Download the `zip` file containing the assignment release and place it in your `ece244` directory. Unzip the file, which will create a directory called `lab2` in your `ece244` directory. This is where all the assignment files are.

A working version of the game (called `tic-tac-toe-ref.exe`) appears in the `lab2` directory. It runs only on only for ECF machines and you should run it to see how the game behaves.

You will write code in two files: `playMove.cpp` and `main.cpp`. The first contains the "logic" of the game and it implements the `playMove` function as described in Section 4. The second contains the `main` function of your code and it performs all the actions needed to display the game graphically and interact with the user, as described in Section 7. Both files are in the `src/tic-tac-toe` directory in `lab2`. The files have comments to guide you towards what you have to implement. Pleaae note that **you are not allowed to add, remove or rename files, or to modify any files other than** `main.cpp` **and** `playMove.cpp`.

The `lab2` directory also contains pre-configured projects for both `NetBeans` and `CodeLite` as well as a `Makefile` should you want to directly use the command line. To use the supplied `NetBeans` project, start `NetBeans`, open a project through the menus: `File -> Open Project` and select `~/ece244/lab2`. You can then compile and run the code in `Netbeans`.

If you would like to use `CodeLite`, then start `CodeLite` and create a new workspace at your `ece244` directory using the menus: `File->New->New Workspace`. In the resulting dialog box, select `C++` and uncheck `Create the workspace under a separate directory`. Provide a workspace name of `ece244` and then press the `...` to specify the workspace path. Select `~/ece244`. Press `OK`. Finally, add the project to your workspace through the menus: `Workspace->Add an existing project` and select `~/ece244/lab2/tic-tac-toe.project`.

To use the command line to compile, make `~/ece244/lab2/src` your current working directory and type `make`. This will utilize the provided `Makefile` and place the executable in `~/ece244/lab2/build/EXE/tic-tac-toe`. Copy this file to `~/ece244/lab2` and run it there.

The `~ece244i/public/exercise` command will also be helpful in testing your `playMove` function. Since only the text output of `playMove.cpp` can be tested, the usage of `exercise` is slightly different for this assignment: you will submit your `playMove.cpp` to exercise instead of the binary executable.

First change directory to where the sources of your assignment code are:

```
cd ~/ece244/lab2/src/tic-tac-toe
```

Then run exercise as follows:

```
~ece244i/public/exercise 2 playMove.cpp
```

Note how you are submitting to `exercise` the `playMove.cpp` file as opposed to an executable, as you did in the first lab. In this case, `exercise` will compile and run your function with a test `main` function and will let you know if your output has any errors in it. Please note that some of the `exercise` test cases will be used by the autotester during marking of your assignment. However, we will not provide all the autotester test cases in `exercise`, so you should create additional test cases yourself and ensure you fully meet the specification listed above.

9 Marking and Deliverables

The assignment is marked in three parts. In the first part, you will demonstrate your code to one of the TAs in the lab, who will assess the graphics components of the code and assign you a mark. The successful demonstration of the graphics is worth 30% of the mark. In the second part, you will submit your code for autotesting and style marking. Another 50% of your mark is based on the correctness of your `playMove` function. Finally, the programming style (structure, descriptive variable names, useful comments, consistent indentation, use of functions to avoid repeated code and general readability) shown in your code will be examined and marked. This part is worth 20% of the total assignment mark.

Submit the `main.cpp` and `playMove.cpp` files as lab 2 using the command:

```
~ece244i/public/submit 2.
```