Lab 2 : A Graphical Tic-Tac-Toe

1 Objectives

The main objective of this assignment is to introduce you to the use of classes, objects, and methods, thus applying the related concepts presented in the lectures. You will do so by building a simple graphics-based Tic-Tac-Toe game. In the required part of the assignment, you will implement a class that represents the state of the game and a function that implements the logic of the game. Your code will link with instructor-provided code that implements the graphics of the game. In the optional (for extra marks) part of the assignment, you will implement the graphics of the game using the SFML graphics library. This optional part will introduce you to graphics programming and to writing event-driven programs that respond to mouse clicks.

2 Tic-Tac-Toe

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`.

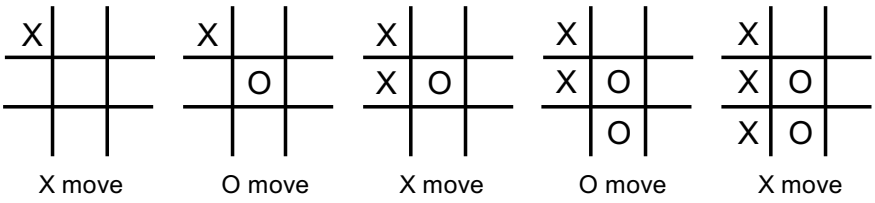

Figure 1: An example Tic-tac-toe game

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 Game Overview

The game consists of two main components: the *display server* or simply the *server* and the *game logic*. This is shown pictorially in Figure 2. The server is responsible for detecting mouse clicks and converting the window coordinates at which the mouse is clicked into game board coordinates. It is also responsible for graphically displaying the game board in a window. Finally, it is responsible for ending the game in response to the `Esc` key or a mouse click on the close icon at the top right corner of the window. The game logic contains a single function called `playMove`. This functions determines, for each mouse click by a player in a board location, if the move represented by the click is valid or not, if the game is over or not and accordingly "plays" the move.

The server and the game logic interact using an object of the type `gameState`. This object stores the state of the game, including the board coordinates at which the mouse is clicked, the

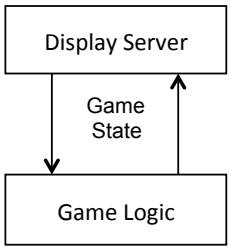

Figure 2: The game components

marks at each board location, whether the move is valid or not, whose turn it is, whether the game is over or not, etc.

The game operation is simple. The server displays the initial (empty) game board and waits for the user to click the mouse somewhere on the board. It then translates the window coordinates at which the mouse is clicked into game board coordinates. It updates the `gameState` object with these coordinates and invokes the `playMove` function of the game logic, passing to it (by reference) the `gameState` object. This function updates the `gameState` object. When the function returns to the server, the display is updated based on the updated `gameState`. The process repeats until there is a win or draw, until the `Esc` key is pressed or until the window is closed.

4 Problem Statement

This assignment consists of a required part and an optional part. In the required part, you will implement the methods of the class `gameState`, as defined in `gameState.h`. You will also implement a function `playMove`, which "plays" the move indicated by the user's click of the mourse. Your code will be linked with an instructor-provided display server to complete the game.

Once done, and *only after you are done* with the required part, you can attempt the optional part for extra credit. In this optional part, you will implement the display server.

The remainder of this section describes the `gameState` class, the methods of which you must implement, the `playMove` function that implements the game logic and some key files that make up the game. Section 7 details the optional part of the assignment.

4.1 The `gameState` Class

The state of a tic-tac-toe game is represented by an object of the `gameState` class. This object is created, initialized, and eventually destroyed by the display server. The definition of this class appears in the file `gameState.h`, which is located in the directory `/share/copy/ece244f/lab2/include`. You can only read but not modify this file. The class contains the following data members:

- `gameBoard` is a `boardSize` $\times$ `boardSize` (3×3 in this assignment) two-dimensional array that represents the game baord. 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 elements of the array are initialized by the server to `Empty`. The definitions of `boardSize`, `Empty`, `X` and `O` appear in the file `tictactoe.h` in the directory `/share/copy/ece244f/lab2/include`.

  In the array, `gameBoard[0][0]` represents the top-left corner cell of the game grid. Similarly, `gameBoard[boardSize-1][boardSize-1]` represents the bottom-right corner cell of the board.

- `clickedRow` and `clickedColumn` are two integers that store the board grid coordinates at which the mouse was last clicked.
- `moveValid` is a Boolean variable that is set to `true` when the mouse click represents a valid move for the current game. That is, it is set to `true` when the grid cell at `clickedRow` and `clickedColumn` is empty and is set to `false` otherwise.
- `gameOver` is a Boolean variable that should be set to `true` if the game is over as a result of the last mouse click (i.e., win or draw) and to `false` otherwise.
- `turn` is a Boolean variable that indicates whose turn it is, `X` (`true`) or `O` (`false`) for the current mouse click. If the move is valid, then the value of `turn` should be changed by the game logic from `true` to `false` or from `false` to `true` to reflect the change in turn. The server does not use this variable at all.
- `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 display server uses this code to draw a line on the game board.

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

The `gameState` class provides accessors and mutators to the above class data members. Please read the comments in the `gameState.h` file to find out what these methods do.

4.2 The `playMove` Function

The game logic implements a single function:

```
void playMove(gameState& game_state)
```

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 `gameState` object that is passed by reference to function. Upon completion, the function must update the game state object by updating:

- The game board at the appropriate location by either `X` or `O`.
- The `turn` value to reflect that the turn has changed.

- The Boolean variable `validMove`, described earlier.
- The Boolean variable `gameOver` to either `true` or `false` to reflect if the move ends the game in either a win or a draw.
- The variable `winCode` to either 0 if the game is not over or to the winning code (as described above) if the game is over.

The two game grid examples shown in Figure 3 are used to demonstrate the expected updates to the game state object.

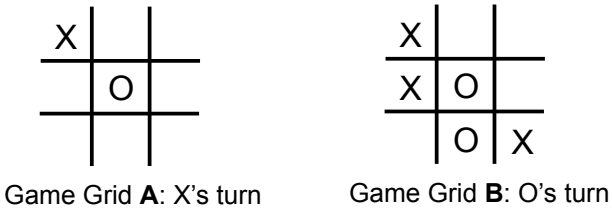

Figure 3: Input game boards

For game grid **A** on the left, the `turn` variable is `true`, indicating that it is X's turn to play. If `clickedRow = 1` and `clickedColumn = 2` when the `playMove` function is called, then an X is placed in the second row and third column of the game board. The move is valid (`validMove` is set to `true`), but the game is not over (`gameOver` is set to `false`). The variable `winCode` is set to 0. The variable `turn` is changed from `true` to `false` to indicate that it is now O's turn to play.

However, if `clickedRow = 1` and `clickedCol = 1` when the `playMove` function is called by the server, then the move is not valid (O's mark already occupies the cell). The game board is not updated. Further, `validMove` is set to `false`, `gameOver` is set to `false` and `winCode` is set to 0. The variable `turn` is **not** changed to indicate that it is remains X's turn to play with a valid move.

Similarly, for game grid **B** on the right, upon entry to `playMove`, `turn` is `false` (or `0`) indicating that it is O's turn to play. Thus, if `clickedRow = 0` and `clickedCol = 1`, the game state object should be updated to have `validMove` as `true`, `gameOver` as `true`, the game board updated with an `O` in row 0 and column 1, and `winCode` set to 5.

4.3 Include Files

In addition to the `gameState.h` file described above, there are two other include files that you should look at. They are:

- `tictactoe.h`, which has global definitions for `X`, `O` and `boardSize`. You should include this file in your `playMove.cpp`.
- `displayServer.h`: this file has the prototype of the display server function. You only need this file is you are implementing the display server for the optional part of the assignment.

These files are in the instructor's space at `/share/copy/ece244f/lab2/include`. You are allowed to read and include these file, but not to modify them. You can include the file by using only the file name. There is no need to specify the entire path to the file.

5 Procedure

Create directory called `lab2` in your `ece244` directory. Make sure that the permissions of this new directory is such that it is readable by none other than you (refer to lab assignment 1 for how to do so). Download the `zip` file containing the assignment release and place it in this `lab2` directory. Unzip the file, which will create the assignment files in the directory.

You will write code in two files: `gameState.cpp` and `playMove.cpp`. The first contains the implementation of the methods of the class `gameState`. The second contains the "logic" of the game in the `playMove` function. Both files are in the `tictactoe/` directory in `lab2`. Since you are not allowed to modify any of the include files needed for the assignment (i.e., the `.h` files), they are not part of the release. Instead they are placed in the directory `/share/copy/ece244f/lab2/include`, where they are directly accessed. If you wish to view the files, you can use NetBeans to navigate to the files, or read them directly by specifiying the full path of the file.

The `tictactoe` directory also contains a pre-configured project for `NetBeans` 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/tictactoe`. You can then compile and run the code in `Netbeans`.

To use the command line to compile, make `~/ece244/lab2/tictactoe` your current working directory and type `make`. This will utilize the provided `Makefile` and place the executable in `~/ece244/lab2/exe/tictactoe.exe`. You can directly run this file.

The `~ece244i/public/exercise` command is helpful in testing your code. First change directory to where the executable of your assignment code is:

```
cd ~/ece244/lab2/exe/
```

Then run exercise as follows:

```
~ece244i/public/exercise 2 tictactoe.exe
```

The `exercise` command will and run your game in a special "text" mode that does not display graphics nor use mouse input. Instead, it accepts row and column grid coordinates from the keyboard and prints to the screen the game stare, including the board. It will let you know if your code 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.

6 Marking and Deliverables

You must submit your code for autotesting and style marking. The autotester will be used to check the correctness of your `gameState` implementation and your `playMove` function. This part is worth 80% of the mark of the assignment. The TAs will examine your code and mark your programming style (structure, descriptive variable names, useful comments, consistent indentation and code readability). This part is worth 20% of the assignment mark.

To submit your code, *make sure your are in the ditectory that contains the source files* (i.e., `~/ece244/lab2/tictactoe`). Submit your `gameState.cpp` and `playMove.cpp` files as lab 2 using the command:

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

7 OPTIONAL: The Display Server

The optional part of this assignment is to implement the display server. You should **NOT** attempt this optional part until you are completely done with the required part of the assignment and submitted you code. You should first read Appendix A on basic SFML graphics and Appendix B on event driven programming before proceedings with this optional part.

You will implement the code for the graphical part of the game in the functions `main` and `displayServer`. Skeletons of the code you must implement appear in two files: `main.cpp` and `displayServer.cpp`. These files are contained in the optional release `zip` file (see Section 5 for how to get this file). The skeletons contain comments that guide you towards what you have to do. In summary, your `main` function should only invoke the `displayServer` function. The `displayServer` function must:

- Declare and initialize the various variables needed for the graphics. Most of them are already declared for you in the skeleton code.
- Create a window and initialize its icon (see Appendix A.1 and the comments in the released code).
- 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 Appendix A.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 Appendix B).
- 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 object `game_state`.
- Examine the object `game_state` after `playMove` returns 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 stop accepting mouse clicks after this. The same happens if the game ends in a draw.

7.1 Procedure

Create directory called `lab2-opt` in your `ece244` directory. Make sure that the permissions of this new directory is such that it is readable by none other than you (refer to lab assignment 1 for how to do so). Download the `zip` file containing the release files for the optional part of the assignment and place it in this `lab2-opt` directory. Unzip the file, which will create the necessary files in the directory.

You will implement your code in four files. The first two are `playMove.cpp` and `gameState.cpp`, which you have already written for the required part of the assignment. You should just copy them over. The this file is the `displayServer.cpp` and this should contain all the code needed to display

the game graphically and interact with the user. The last is the tictactoe.cpp file, which contains the `main` function of the program and this function simply invokes the `displayServer` and returns when the `displayServer` function returns. The files have comments to guide you towards what you have to implement.

A reference executable (`tictactoe-reference.exe`) is released with the assignment. You can use this executable to clarify the graphical specifications of the assignment. If in doubt, run the reference solutions and see how it behaves. Please note that this executable will work only on ECF Linux machine (i.e., not Windows or Mac machines).

7.2 Marking and Deliverables

There are no `exercise` or `submit` commands for the optional part of the assignment. Instead, 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 an *additional* 30% of the mark for the assignment.

A 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`.

A.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.

A.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.

A.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();
```

B 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)
   :
   :
}
```