**EECE435L Final Report**

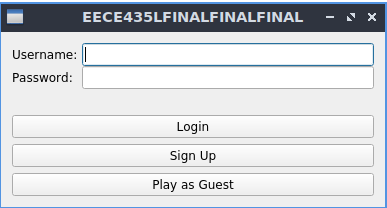
**Youssef Jaafar and Abbas Mahfouz**

**Introduction:**

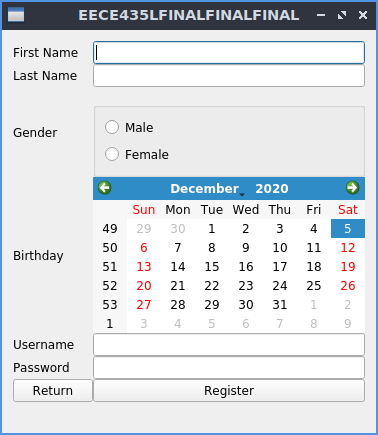
Over the course of this semester we had to get familiar with Qt Creator and different software tools and practices such as using makefiles, debugging, creating code documentation, checking for memory leaks and packaging, to complete our final project: a game platform where each user has their own account and access to two different games, Kill COVID-19 and Reversi (Othello).

**Implementation of Accounts and framework menu:**

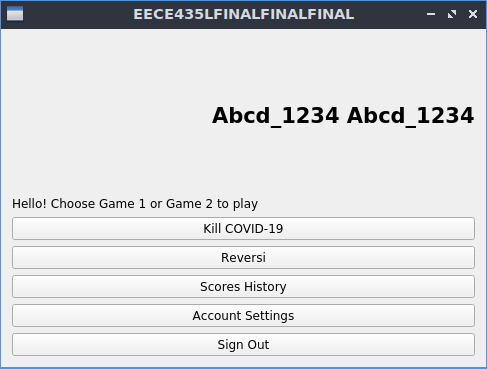
Every user either play as guest, or can have an account through which she/he can access the different games at the last reached level, and can check the performance and the history scores in every game.



In the sign-up form, a user is asked to enter their First and Last names, Gender, date of birth, a username and a password.

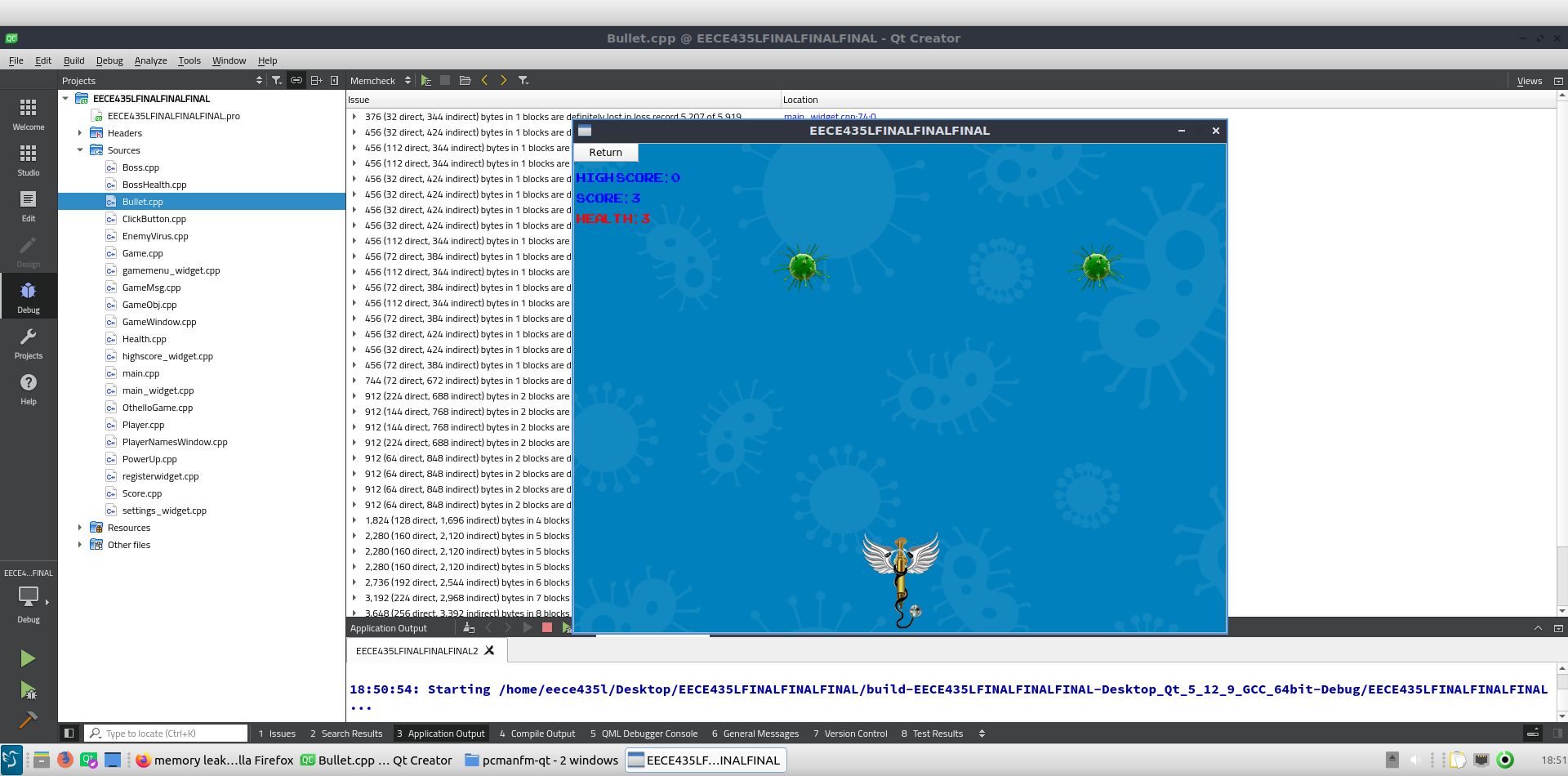


After logging in or playing as guest, the user is directed to the game menu where they can select to play game 1 or game 2, in addition accessing scores history, and account settings.



**Kill COVID-19:**

The Kill COVID-19 game we implemented was modeled after class shooter games like Space Invaders where the player can move left and right on a vertically scrolling screen where the goal is to shoot the enemies before they reach the bottom of the screen.

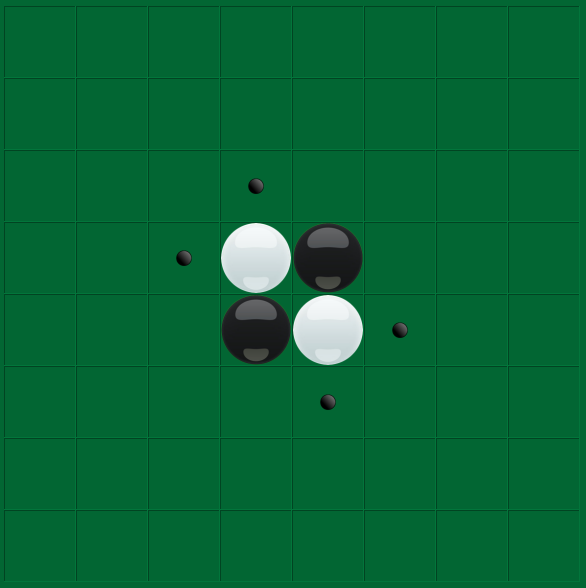


The implementation was done by creating a player class which has horizontal movement defined by either decrementing or incrementing the horizontal position while either left or right key is pressed until the player reaches the end of the screen. Shooting the enemies is done by spawning a bullet that has an increasing vertical position (actually done by decrementing the y value since the top of the screen is the beginning of the axis) until touching an enemy, determined by a built in function for checking collisions, at which point both colliding items are deleted in the case of regular enemies or it calls a function decrementing boss health. When enemies are defeated the player score is increased.



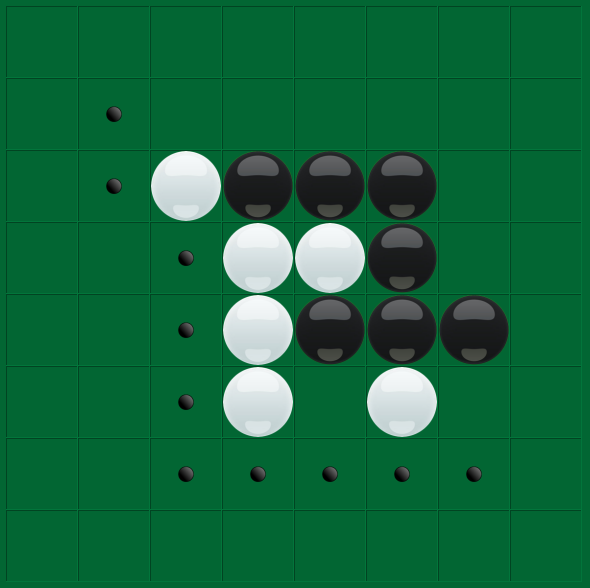
Enemies are spawned on screen randomly by reading different possible positions from a text file and then creating them in the set places on screen. The speed at which enemies descend towards the player increases as the player defeats more enemies to increase the challenge by doubling the enemy speed variable until it reaches a maximum of 16. If the player reaches a score limit they win the game but if their health drops to 0 they lose. Either way, they are taken to a screen that lets them either restart or exit.

**Othello:**



Our implementation of Othello attempts to implement the rules of the game in which the 8x8 grid has 4 pieces at the start in the center and only moves which can flip pieces, which means sandwiching a piece or more between two pieces of the opposite color, are valid.

Our implementation is based on the combination of the game’s basic elements: squares and pieces. The board is a vector of pieces. Each piece has a position defined by its horizontal and vertical coordinates on the 8x8 board. In order to check valid moves we created a function that checks each square in the entire grid to see if a flip can be made in a given direction (up, down, left, right, upper diagonal, lower diagonal…) by finding whether the square contains a stone and if its neighbors are of the opposite color in each direction and then checking whether at the end of that direction (before reaching the end of the board) there is a piece of the same color as the active player. Possible moves are shown on the board by highlighting the squares in a different color.



The player moves by clicking on the squares at which point the game checks whether the square clicked is one of the valid ones found before. If a move is valid and a row, diagonal or diagonal can be flipped the game does so by checking all the possible moves for the square clicked (meaning the different directions pieces can be flipped from the square) and then flips then by parsing the initial position of the square and the end position given from the list of possible moves (meaning where it meets a piece of the same color).

The game is encapsulated in a menu which allows the player different options like setting the starting color, restarting and viewing their high scores. High scores and point histories are implemented through SQL. Before starting each player has the ability to enter an identifier as well.

**Division of Tasks:**

* Both of us collaborated on most parts of the code in order to complete the project efficiently. This is especially true for larger files such as the classes responsible for handling the game processes like Game.h/cpp in Kill COVID-19 and GameWindow.h/cpp in Othello because they require implementing functions in which many of the classes we defined interact, for example Game.h/cpp has a function that needs to spawn the enemies that were defined in another class and then another function to make sure they are moving correctly on screen. In all phases, we divided up the different functionalities needed and each of us worked on defining a class to meet the requirements.
* Here are some highlights of each of our work:
  + Abbas

The spawning of enemies randomly on the screen in Kill COVID-19 was handled by Abbas through a text file containing different binary strings where 1s meant that an enemy would be spawned at that position (e.g.: 00100). The game knows where to spawn the enemy thanks to a function that attributes each position of 1s to place on the screen.

* Youssef

Tracking scores in Othello through the implementation of SQL Tables which are initialized once and then save the scores of each player and different measures taken such as the start time of the game

**Challenges Faced:**

* In our implementation of each phase and the different games we focused on creating functional code first and created each part alone. While this approach was helpful in focusing it became a challenge later when trying to link the different parts together. The issue was solved by modifying each game to be able to link back to the menu by having the user identifier included but it also posed difficulties in the ways the files and projects were connected. This issue could have been avoided by planning and thinking of the final implementation from the start and we will definitely pay attention to that in future projects.
* There were problems with the settings options. While it initially did not work well, we were able to fix it by improving our method of reading from files and fixing a bug in the initial code. However, there was still some difficulty in loading profile pictures.
* Saving the high scores to a txt file posed a lot of problems especially for Othello but that was fixed using SQL.
* The implementation of the makefile was difficult as the Qt dependencies required a different approach.

**Memory Leaks:**

* Qt Creator integrates Valgrind code analysis tools for detecting memory leaks and profiling function execution.

To analyze applications using valgrind, we did the following: (Qt documentation)  
- In the **Projects** mode, select a debug build configuration.  
- Select **Debug** to open the **Debug** mode, and then select **Memcheck** on the toolbar.  
- Select the "Start button" button to start the application.  
- Use the application to analyze it.  
- Select the "Stop button" button to view the results of the analysis in the **Analysis** view.

We can also use valgrind by clicking on analyze, then on valgrind memory analyzer.

* We ran the Valgrind tool locally on Light-Ubuntu Virtual machine, the results are below:

****

**Handling Memory Leaks:**

One way we can manage memory in Qt is using QPointer. We may use QPointer on all classes based on QObject. We would also use QPointer in a situation where the pointer is being created outside of a QObject class such as in the main function or other none Qt based classes we may have created in our code.

Here is how we can use it:

Instead of using: MyObject \* obj = new MyObject;

We can now use : QPointer<MyObject> object = new MyObject;

The Qpointer wraps obj as a pointer to MyObject. QPointer deals with dynamically destroying the obj pointer should there no longer be any references to it.

We can also manually delete all initialized pointers (objects created on the heap) like we did in class.

**Profiling:**

For profiling, we used the Callgrind tool to profile functions execution. We can also load the data generated by Callgrind (The result will be stored in a callgrind.out.XXX file where XXX will be the process identifier.) into the KCachegrind profile data visualization tool for browsing the performance results.

To run Callgrind, we did the following: (Qt documentation)

1. In the **Projects** mode, select a release build configuration.
2. Select **Debug** to open the **Debug** mode, and then select **Callgrind** on the toolbar.
3. Select the "Start button" button to start the application.
4. Use the application to analyze it.
5. Select the "Stop button" button to view the results of the analysis in the **Profile** view.

Callgrind (Valgrind function profiler) records the call history of functions that are executed when the application is run. It collects the number of instructions that are executed, their relationship to source lines, the relationships of the caller and callee between functions, and the numbers of such calls.

Below is some of the results:

