5-5-2024

Implementing Reinterpreted Requirements into an Existing Senior Design Project

Cooper James Bond
University of Alabama in Huntsville

Implementing Reinterpreted Requirements into an Existing Senior Design Project

by

Cooper James Bond

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

05/05/2024

Honors Capstone Project Director: Dr. Jacob Hauenstein

Cooper Bond *05/05/2024*

Student Date

Jacob Hauenstein *05/05/2024*

Project Director Date

Letha Etzkorn Digitally signed by Letha Etzkorn
Date: 2024.05.05 14:06:56 -05'00'

[Name Here] *[Date Here]*

Department Chair Date

Sean Lane Digitally signed by Sean Lane
Date: 2024.05.05 18:45:33 -05'00'

[Name Here] *[Date Here]*

Honors College Dean Date

Honors College

Frank Franz Hall

+1 (256) 824-6450 (voice)

+1 (256) 824-7339 (fax)

honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted with the final manuscript.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Cooper Bond

Student Name (printed)

Cooper Bond

Student Signature

05/05/2024

Date

Table of Contents

Abstract

The English Dart Premier League draws large crowds, and it is difficult for the audience to see the dartboard and keep track of the status of the match. The UAH CS 499 Senior Project Assignment “Dart Premier League” hopes to solve this issue through the implementation of thirteen features. For CS 499-02 FA23 Group 3, the thirteen features were implemented through their development of the Bullseye Scoreboard program. The purpose of “Implementing Reinterpreted Requirements into an Existing Senior Design Project” is to rework two of those requirements as implemented in the Bullseye Scoreboard to better perform for the intended audience. It was intended to implement the reworked requirements through existing features of PyQt: resizeEvent()and QStackedWidget. Ultimately the resizeEvent() implementation was successful, but the QStackedWidget implementation was found to be impractical given existing structure of the code, instead an alternative solution was found that while not meeting the explicit requirement, met the intended goal of the requirement.

Introduction

The UAH CS 499 Senior Project Assignment “Dart Premier League” has thirteen requirements that together result in an entity structured around two displays and an algorithmic implementation of a darts game. The first display, which whoever is recording the games of darts uses, will consist of two separate views: “Set up match view” through which the specifics behind the current match is input to the system, and “Match in progress view” through which the current match is recorded live, and must be resizable. The second display, through which the audience of the darts game will see information regarding the match and league, will consist of three separate views, all of which are required to be resizable:

- “Competitor Info View” which will display information about the current players and their statistics from previous matches in the league.
- “Match Info View” which will display information regarding the current players and their statistics from the current match.
- “Main Scoreboard” which will display an alternative version of “Match Info View” that includes information of the current leg of the match rather than statistics encompassing the entire match so far (which may consist of multiple legs). While not explicitly required, in Bullseye Scoreboard it is used to implement the requirement, “If the player is on track for the minimum number of throws to win the leg it will be displayed on the scoreboard.”

In addition to these visual requirements are algorithmic requirements designed to enable the scorer to accurately show the state of the darts game automatically, meaning that the program must be able to correctly emulate a game of darts, in addition to displaying information regarding

the league (information about matches and players). Strict requirements for these algorithms were:

- League management: maintain players and player information.
- Set-up for a match: Select competitors, scoring leg start value, number of legs, number of matches, initial running statistics to display, date of match, location of match, official names, layout for stadium scoreboard.
- Championship play, which consists of multiple matches.
- Ability to maintain data for darts thrown.
- Calculate and display if a player is on track for a perfect leg (where they are able to complete a leg in the minimum number of throws possible).
- Proper implementation of a dart match's leg, including scoring for a leg starting at either 801, 501, or 301, properly updating current score for the current player when recording their dart throws, and restrictions in a dart match like a winning dart has to be a double, winning is setting score exactly to zero, and a failsafe to account for a player getting their score to 1 where it is now impossible to win with a double.

These were all implemented in the Bullseye Scoreboard by the members of CS 499-02 FA23 Group 3, through PyQt, a set of Python bindings for C++'s Qt application framework, but there was a disjunct between implementation of some requirements, which while met the requirements, directly impeded the intention of the program, an interface to record and display the current state of a dart match. This disjunct is found in the following two requirements:

- "The stadium scoreboard ... can be modified by the scorer during the match without having to bring the stadium scoreboard down."

    - Issue: this requirement is met as long as there is not a period where no windows are visible on the display, leading to an instance where changing the window causes the window to resize or move itself being acceptable. This actively takes away from the viewing experience of the audience.
- "The scorer's interface will consist of a resizable dartboard..." and "The stadium scoreboard is to be resizable..."
    - Issue: this requirement leads to a situation where the "Set up match view" for the scorer does not need to be resizable, even though it will be on the same display as the "match in progress view" that is required to be resizable. This could lead to situations where the "match in progress view" works on a display but the "set up match view" does not.

To combat these issues, which would enable an acceptable submission if following the strict requirements but may yield a program that is unwieldly to use or view, the following changes were made to those two requirements:

- All three stadium scoreboards will be part of the same window, rather than three separate windows that give the illusion of being one window.
- "All windows are to be resizable".

Process and Implementation

First Roadblock

When starting work on the project, one immediate concern was found, at some point between presenting it as a senior design and beginning work on this project, it had seemingly broken. All text white, making things like setting up a match impossible to do:

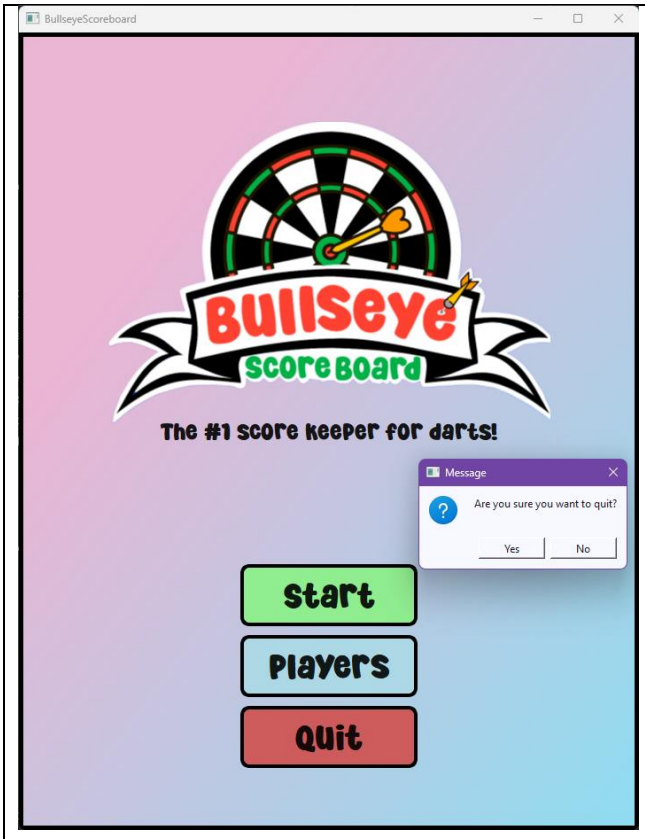

Figure 1: Intended Welcome Screen Display with System Exit Prompt

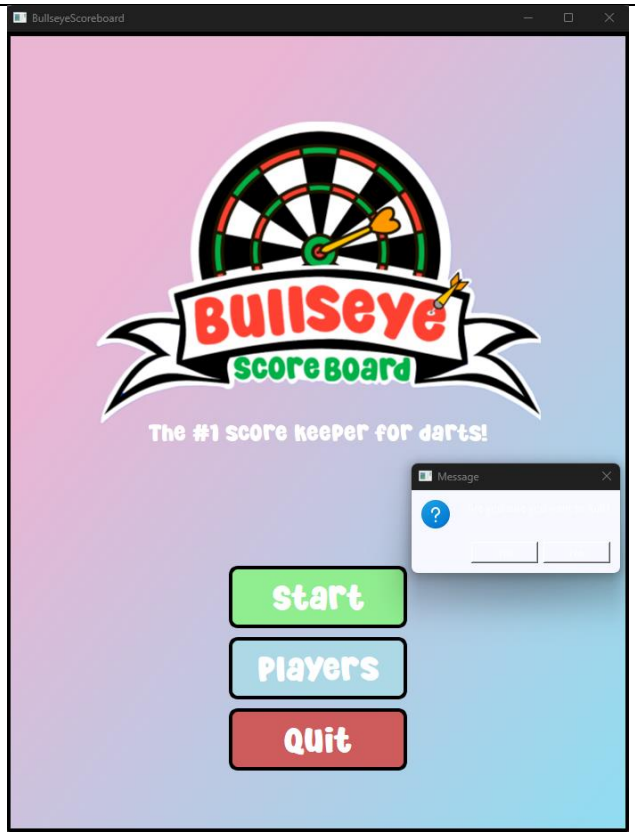

Figure 2: Actual Welcome Screen Display with System Exit Prompt

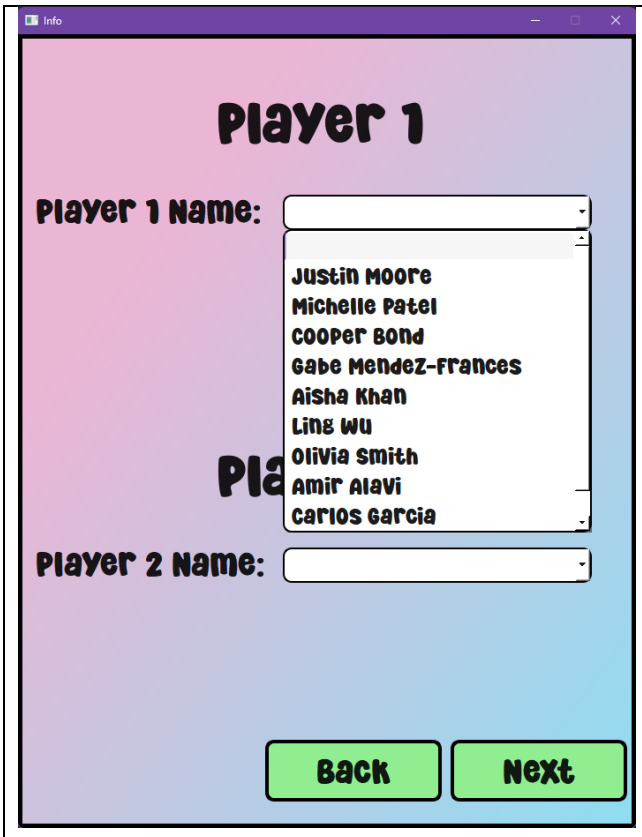

Figure 3: Intended Player Select Screen

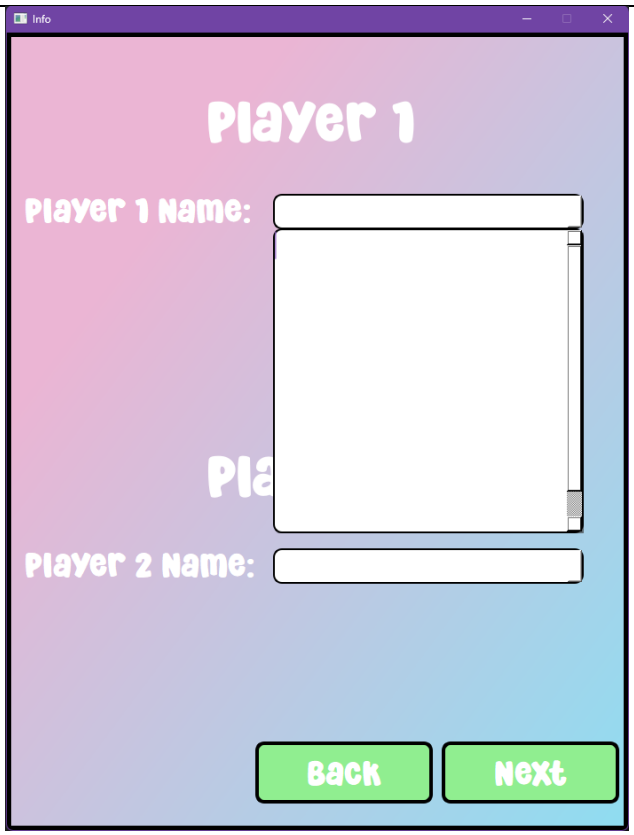

Figure 4: Actual Player Select Screen

This issue was assumed to be due to a difference in existing python packages on my system running it, as between the senior design and starting to work on it, I had to format it due to malware. Assuming that the issue was caused by a silent error in text formatting due to a missing package, an error that was not being recorded to the console, the attempted solution was to pull up the list of packages present on my original machine and individually download them. Doing this did not result in any changes. Ultimately, due to a serendipitous thought, it was determined the actual reasoning behind this was the fact my new system was using Dark Theme instead of Light Theme. Ultimately, a partial solution was found by adding the line

```
QApplication.instance().setPalette(app.style().standardPalette())
```

before executing the app in main. This resulted in a partial fix that while leaving the text on the Welcome Screen white, fixed the system exit prompt in the Welcome Screen and set the text on the Player Select Screen to black, making it visible.

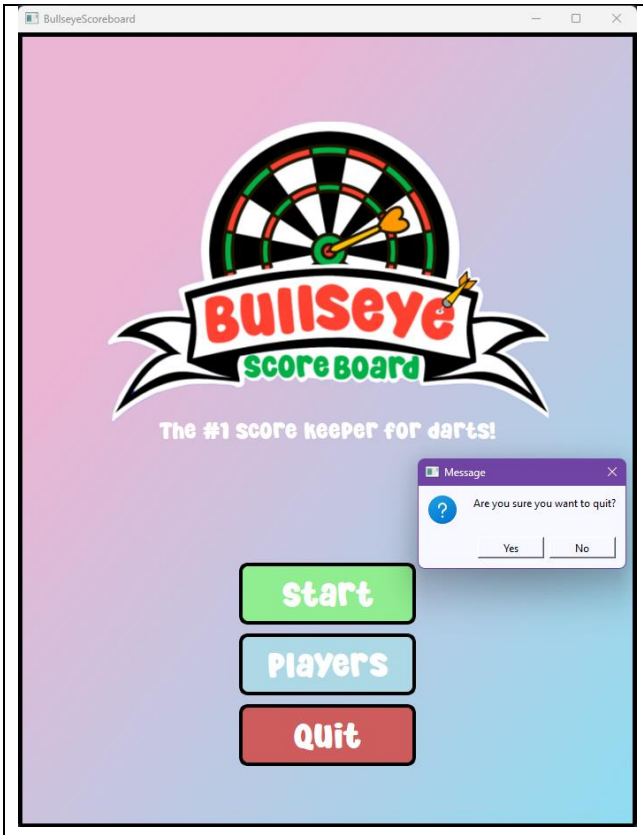

Figure 5: Acceptably "Fixed" Welcome Screen Display with System Exit Prompt

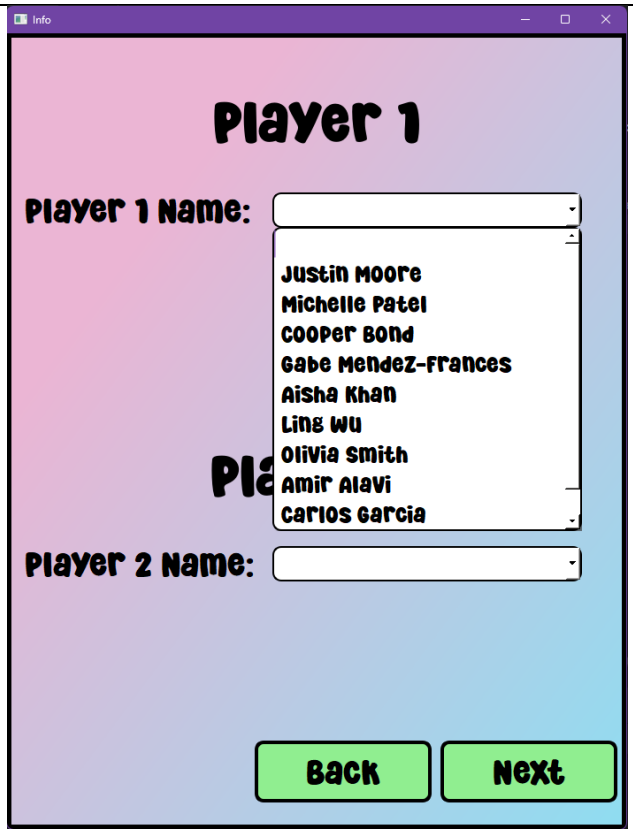

Figure 6: Fixed Player Select Screen (Visually identical to Figure 3)

All Windows Are to be Resizable

While implementing the ability to resize the "Set up match view" it was determined three existing windows needed to be reworked to enable the ability to resize: Welcome Screen, Player Select Screen, and Match Setup Screen. The intent was to implement this through the resizeEvent function found in PyQt, a function that is automatically called on a window every

time that window is resized. For those windows to be resizable, the FixedSize each was assigned, which restricts the window to those specific dimensions and cannot be resized at all, had to be removed. Doing this did make the windows resizable, but not their components. Not only did the components of the window not move nor update, but it was possible to make the window itself smaller than the components:

Figure 7: Player Select Screen resized to be extremely small, cutting off components

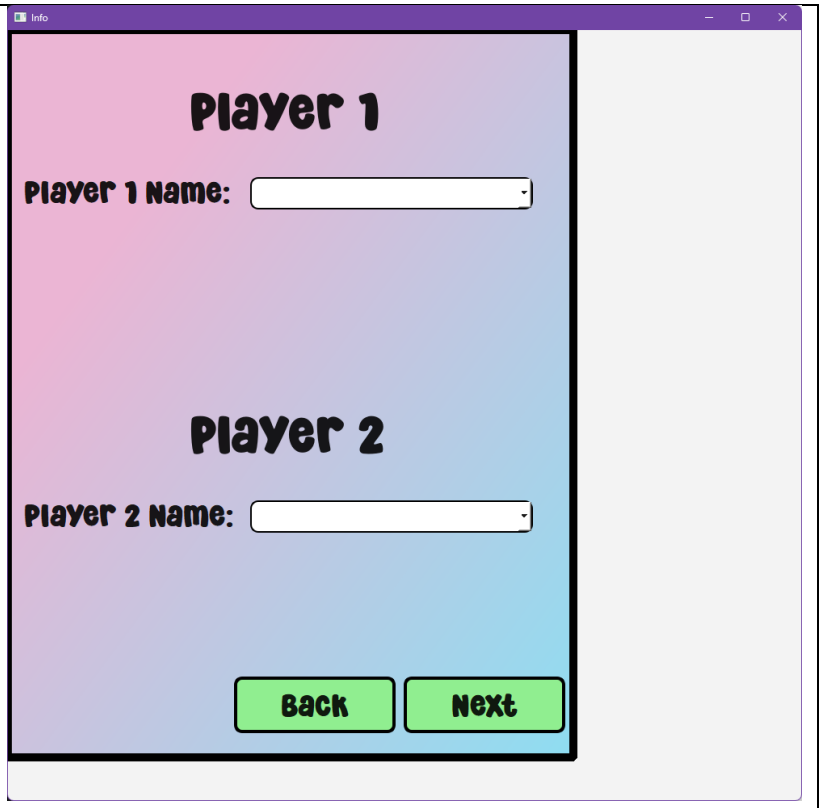

Figure 8: Player Select Screen not redrawing background when resized.

The solution to this was a combination of two actions, the addition of setting each windows MinimumSize equal to the original FixedSize, prevents a window from being smaller than those dimensions, and adding a resizeEvent function and a custom function in each window called adjust_widgets which is called in the resizeEvent function. This adjust_widgets function applies a custom geometric transformation on the Geometry value of each widget, which

modifies their x value, y value, width, and height in a specific manner depending on the “type” of widget, where a widget is one of Qt’s building blocks for GUIs (labels, dropdown menus, buttons, etc) and “type” is a custom identifier I made for the sake of categorization to classify how they should interact with the resizing screen,. These types are as follows:

- Centered Label: Centered Label (Label being Qt’s name for something that holds text or pixel maps) should always be in the center of the screen horizontally, and the same position (I.e. 2/3 the way) down the screen vertically regardless of how the window is resized.
- Centered Button: Centered Buttons will always be in the center of the screen horizontally, and same position down the screen vertically regardless of how the window is resized. Additionally, they should scale up in size as the window becomes larger.
- Corner Button: Corner Buttons should always be the same distance (I.e. 50 pixels) from whatever corner of the screen they are positioned in at default, regardless of how the window is resized. These buttons do not change in size.
- Left Labels: Left Labels will always be the same distance from the left side of the screen, and the same position down the screen vertically.
- Left Label’s Menus: These are the ComboBoxes (Qt’s name for drop down menus) and SpinBoxes (Qt’s name for the boxes you can type input in in a restricted range, and has arrows that increment up or down with a step size) that appear to the right of Left Labels. Their left side will be the same distance from the left side of screen, and their right side will be the same distance from the right side of the screen.

The code implementation to achieve these goals, using width/height_factor being the current width/height of the screen divided by default width/height of screen, and the text

appearing between “self.” and “.setGeometry”/”.move" being the name of the corresponding type of object is as follows:

- Centered Label:

```
self.label.setGeometry(self.label_x * width_factor, self.label_y * height_factor,
        min(int(self.label_width * width_factor), self.width()),
        self.label_height * height_factor)
```

- Centered Button:

```
self.quitBtn.setGeometry(self.quitBtn_x * width_factor, self.quitBtn_y *
    height_factor, self.quitBtn_width * width_factor, self.quitBtn_height *
    height_factor)
```

- Corner Button:

```
self.finishBtn.move(self.width()-(self.defaultWidth - self.finishBtn_x),
        self.height() - (self.defaultHeight - self.finishBtn_y))
```

- Left Labels:

```
self.matchDateLabel.move(self.matchDateLabel_x, self.matchDateLabel_y *
        height_factor)
```

- Left Label’s Menus:

- Due to a quirk in the visual size of labels and menus compared to their actual size, additional code needed to be written so that each Left Label's Menu would appear centered on the appropriate Left Label instead of shifting away from center as manually implemented in the default size.

```
matchDateYConversion = (self.matchDateLabel.geometry().y() + (1 / 2) *
        self.matchDateLabel_height - (1 / 2) * self.matchDate_height)
self.matchDate.setGeometry(self.matchDate_x,
        matchDateYConversion,self.width() - self.matchDate_x - 50,
        self.matchDate_height)
```

For the background of the windows to update, the part that draws the background found in paintEvent needed to be updated to reflect the new screen size, and update() would need to be called in resizeEvent as doing so would end up calling paintEvent. The original implementation of paintEvent drew a gradient that transitions from a shade of pink originating from 25, 400 to a shade of blue originating from the bottom right corner of the screen, then drew a rectangle with that gradient encompassing the entire window (0,0 to 700, 900 at default). To keep the gradients original design the input for width/height factor was used again as follows:

```
grad1 = QLinearGradient(25 * self.width() / 700, 400 * self.height() / 900, self.width(),
    self.height())
```

This ends up drawing a gradient originating from the same proportion down and across the screen as default to the same bottom corner of the screen, both as found in the default

window size. This results in windows that are resizable, and do not take away from the scorers' ability to input things regardless of the new window size:

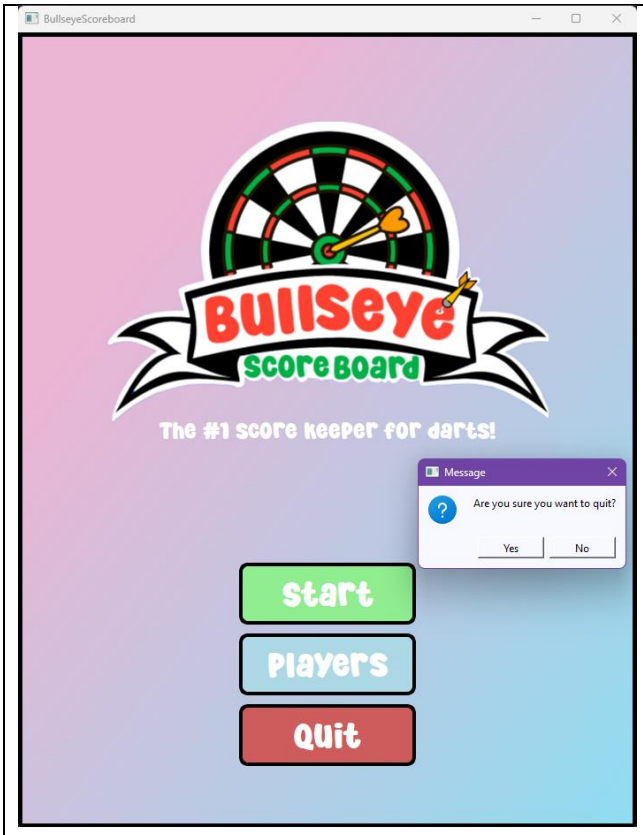

Figure 5: Acceptably Fixed Welcome Screen Display with System Exit Prompt (Default Size)

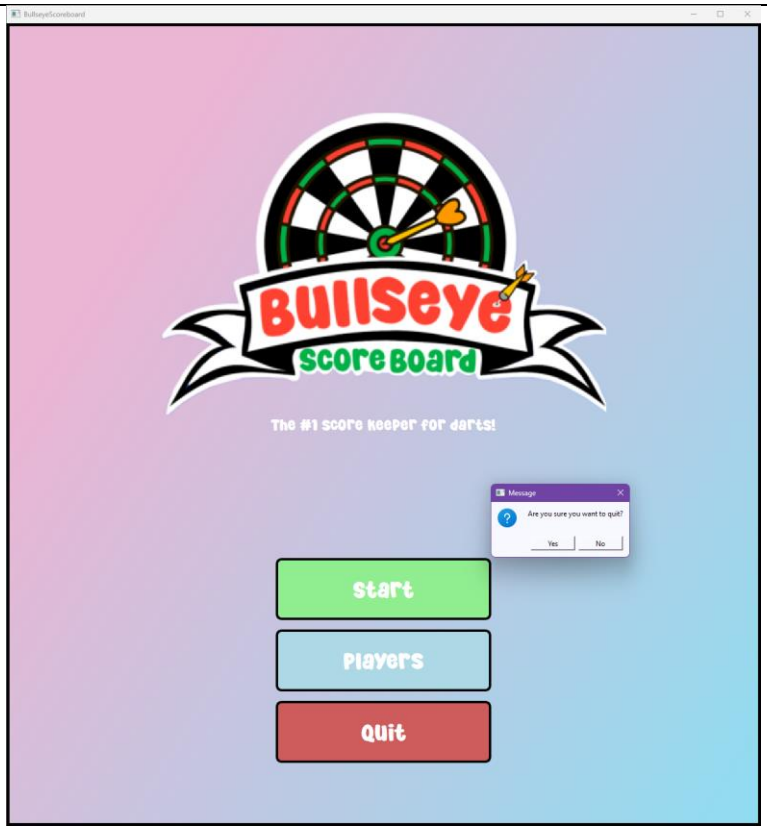

Figure 9: Resized Welcome Screen (Show Center Labels and Center Buttons)

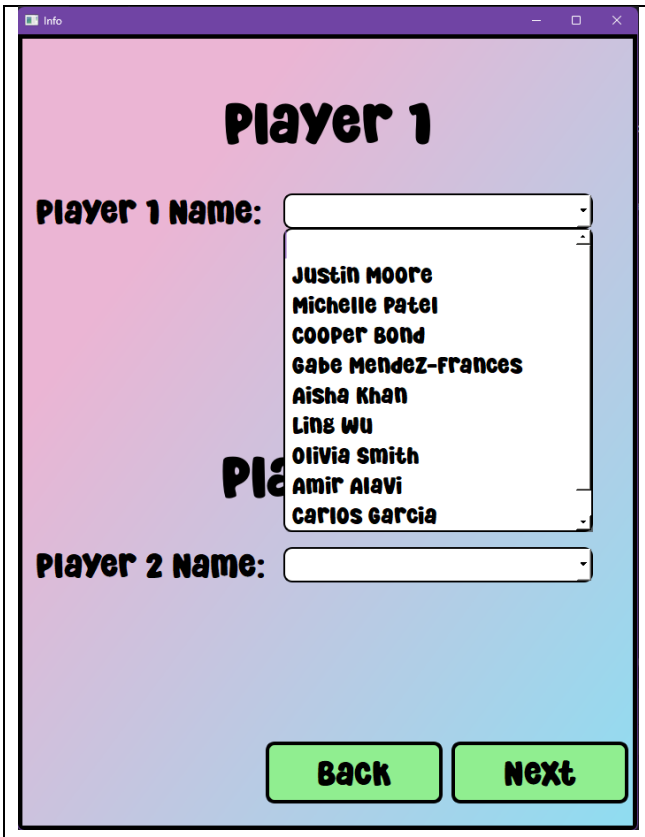

Figure 6: Fixed Player Select Screen (Visually identical to Figure 3) (Default Size)

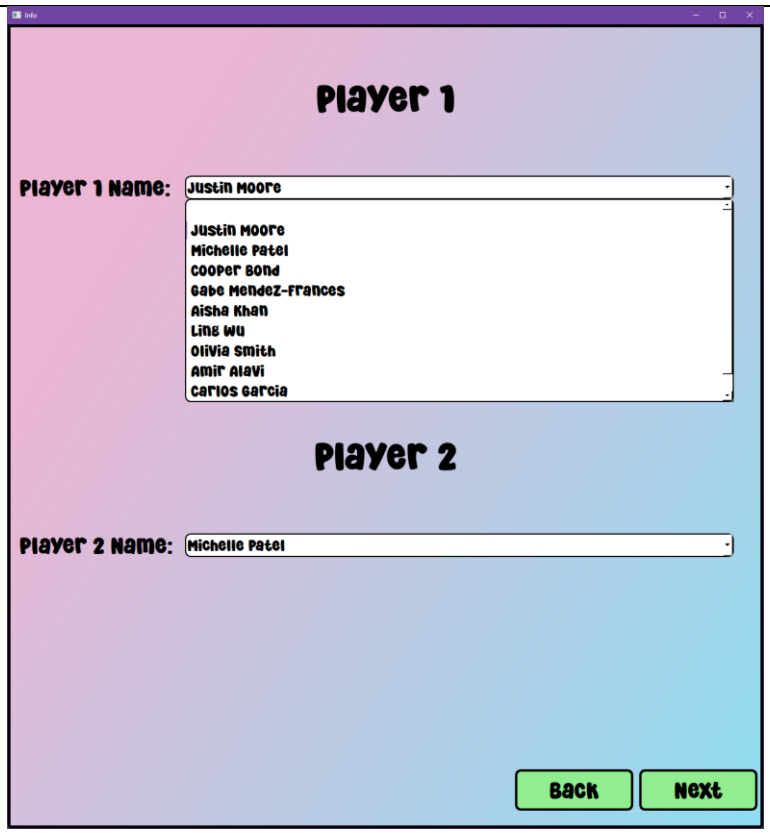

Figure 10: Resized Player Select Screen (Shows Center Labels, Left Labels, and Left Label's Menus)

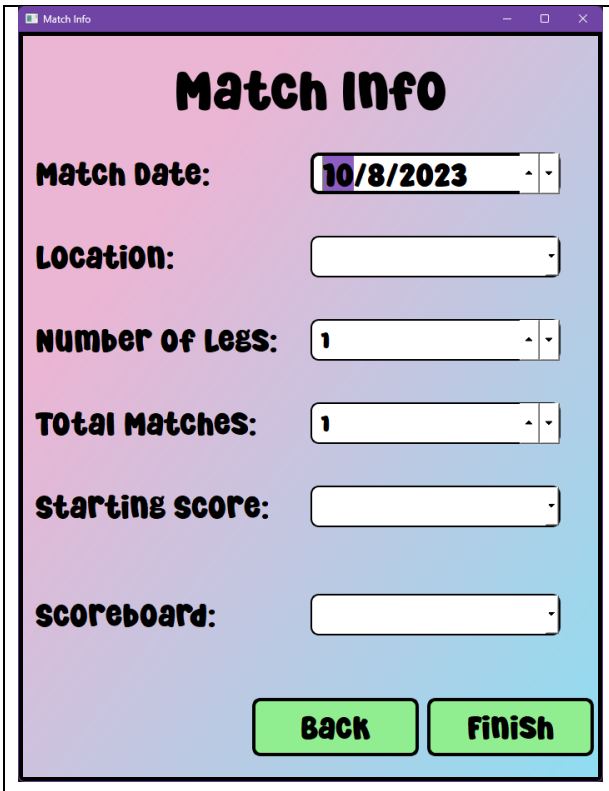

Figure 11: Match Info Screen (Default Size)

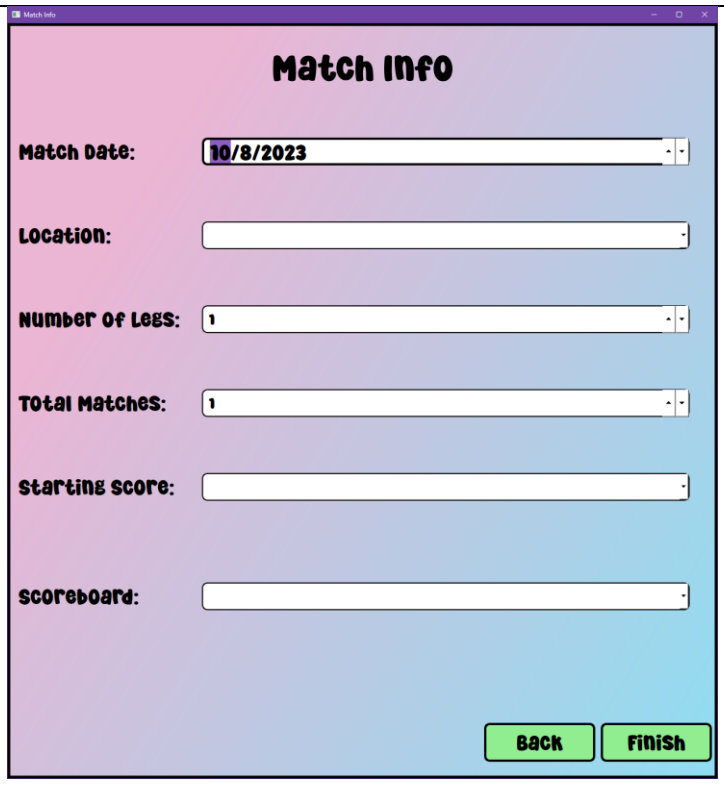

Figure 12: Resized Match Info Screen (Shows Center Labels, Left Labels, and Left Label's Menus)

All Three Stadium Scoreboard Are to Be Part of the Same Window

The three stadium scoreboards are the views at which the audience see, and each have data regarding the match. The three are as follows: "Competitor Info View" which will display information about the current players and their statistics from previous matches in the league, "Match Info View" which will display information regarding the current players and their statistics from the current match, and "Main Scoreboard" which will display an alternative version of "Match Info View" that includes information of the current leg of the match rather than statistics encompassing the entire match so far (which may consist of multiple legs) including if a player is on track for a perfect leg. The intent behind the new requirement, "All

three stadium scoreboards will be part of the same window, rather than three separate windows that give the illusion of being one window" was to remove the visual issues that occur when swapping between scoreboards, the window would flash and change size and position on the screen. This would take away from the viewing experience of the audience.

The planned solution to this would have been through implementation of Qt's StackedWidget, which enables stacking multiple window widgets, and then selecting which one is visible. Ultimately, this was determined to be far too impractical to implement given the current structure of the code. The way the individual scoreboards are defined, along with positioned in the code, made it that setting them up as components of the stacked widget would require a massive amount of refactoring. Only MainScoreboard is defined as a window, MatchScoreboard and CompetitorScoreboard are both defined as "objects." Thus, leading to two variations on creating a window that requires a dramatically different set up for the code to build their respective windows, and changes in the structure for built in Qt functions. Attempting to standardize them and then combine the modified windows would likely introduce an unnecessarily large number of errors in the code that attempting to fix would cascade into breaking other parts of the code. This can be seen how when originally implemented, it broke all fonts on the windows (unrelated to the dark mode font issue mentioned earlier).

The second attempt was to make it that the MainScoreboard had different functions that populated it with different components depending on the Scoreboard selected, when a scoreboard was "changed" instead of hiding the current scoreboard's window and showing the other scoreboard's window, it would delete the contents of the window and then repopulate it with the contents of the selected scoreboard. While this enabled swapping between the 3 scoreboards with the only issue being the dimensions visually change, solving the issues where the position on

screen and the “flashing” that occurred on scoreboard swap, only MainScoreboard would properly render, the other two would become gray boxes.

The third attempt, which succeeded, was adding lines to copy over the current scoreboard’s Geometry (x position on screen, y position on screen, width, height) to the selected scoreboard, and loading all the scoreboards once before choosing which one stays visible on the screen. This does not meet the literal criteria of the requirement, “All three stadium scoreboards will be part of the same window, rather than three separate windows that give the illusion of being one window,” but does meet the intention of the requirement, improving the viewing experience of the audience by removing the flashing, jumping, and resizing of the windows when scoreboards are changed. Copying over the geometry removes the jumping and resizing of the windows, and loading all the scoreboards once prevents flashing the first time a scoreboard is loaded manually (as now they’ve been loaded ahead of time)

Loading the scoreboards preemptively:

```
        # HON code, Loads scoreboards before continuing to the original code
        self.set_screen_selection_to_competitor_scoreboard()
        self.set_screen_selection_to_main_scoreboard()
        self.set_screen_selection_to_match_scoreboard()
        # Original code, load selected scoreboard
        if self.scoreboard_selection == 'Competitor Scoreboard':
            self.set_screen_selection_to_competitor_scoreboard()
        elif self.scoreboard_selection == 'Main Scoreboard':
            self.set_screen_selection_to_main_scoreboard()
        elif self.scoreboard_selection == 'Match Scoreboard':
            self.set_screen_selection_to_match_scoreboard()
        else:
            return
```

Example of copying over geometry when swapping to match scoreboard from main scoreboard or competitor scoreboard:

```
    def set_screen_selection_to_match_scoreboard(self):
        if self.scoreboard_selection == 'Main Scoreboard':
            # Setting the geometry of match scoreboard to main scoreboard, the previous scoreboard
            self.match_window.setGeometry(self.main_scoreboard.geometry())
            self.main_scoreboard.hide()
        elif self.scoreboard_selection == 'Competitor Scoreboard':
            # Setting the geometry of match scoreboard to competitor scoreboard, the previous scoreboard
            self.match_window.setGeometry(self.competitor_window.geometry())
            self.competitor_window.hide()
        self.match_window.show()
        self.scoreboard_selection = 'Match Scoreboard'
        self.update_all()
```

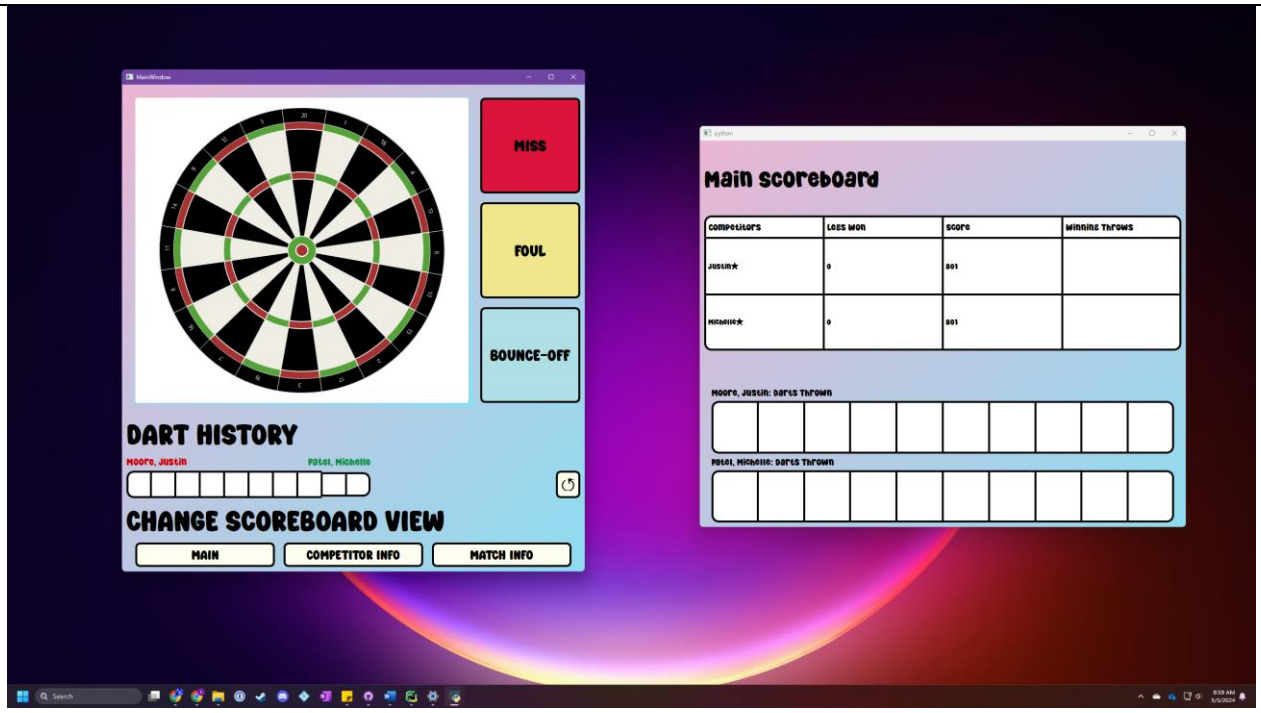

Figure 13: Resized Scorer's Interface & Resized Main Scoreboard View

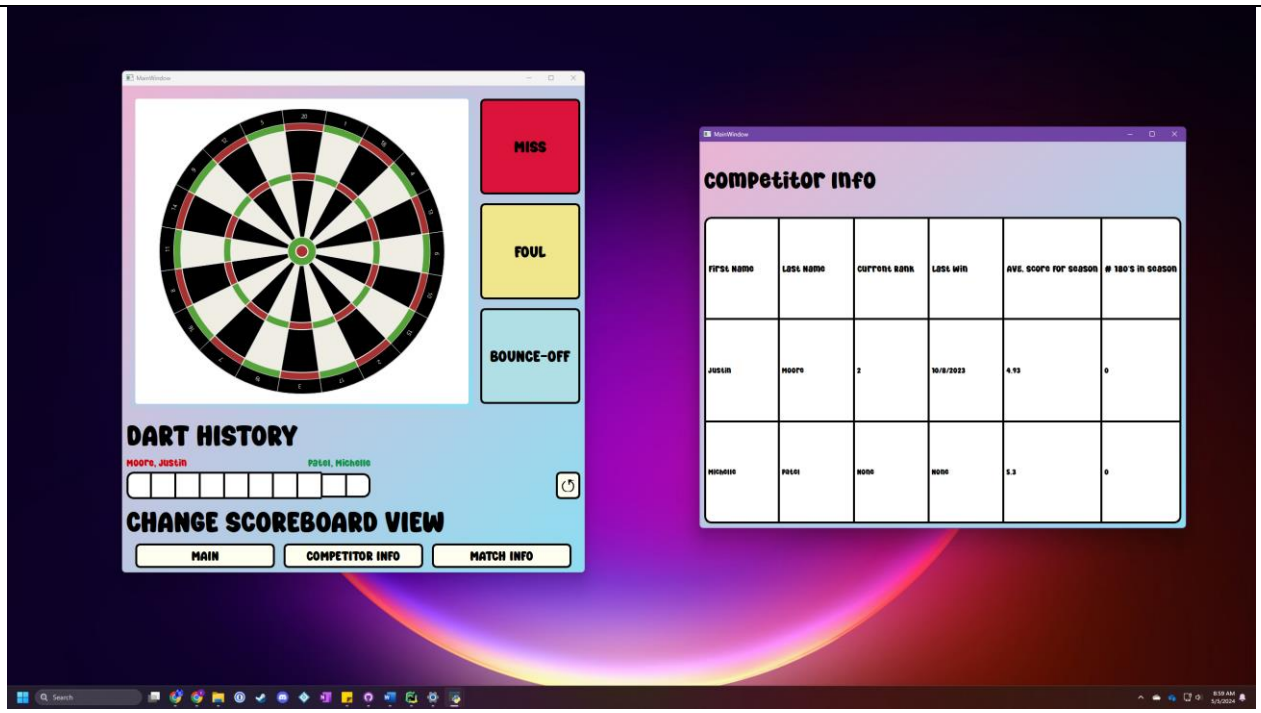

Figure 14: Resized Scorer's Interface & Resized Competitor Info (Competitor Info changed from Main Scoreboard in Figure 13 kept the geometry of Main Scoreboard, being visually indistinguishable from being the same window)

Other Changes and Notes:

- Calculating and displaying if a player was on track for a perfect leg was broken because a variable it used checking for the number of darts thrown by a player was not being updated properly for player 2. Instead, player 1's darts thrown was being incremented during both their turn and player 2's turn, causing the code to think player 1 was no longer on track for a perfect leg once player 1 threw a dart after player 2 threw a dart, and player 2 to supposedly always be on track for a perfect leg. This was discovered and fixed.

- The last names in Competitor Info were not using the proper font. This was due to setting the font on the first names twice, once after setting the text for first name and once after setting the text for last names.
- When a Scoreboard is snapped to any display ratio except for full screen via Window's snap assist (i.e. dragging a window to the left side of the screen makes it fill up the left half of the screen) it will not properly transfer geometries snapping a scoreboard, then swapping from and back to it (I.e. Snap window A to left half of screen, Swap to Window B where geometries properly transferred, Swap back to window A where geometries fail to transfer, resetting position and dimensions). It is unknown how to fix this but is not a cause for concern as in regular use cases it will take up a full screen of a different display, thus won't be impacted by this bug that only triggers when using window's snap assist for part of the screen.

Conclusion/Assessment

Overall, I am satisfied with the result of these additions to CS 499-02 FA23 Group 3's senior project. While it is true that the literal meaning of the requirement "making the three scoreboards part of the same window" was not met, the intention behind it, improving the viewing experience of the audience by removing the flashing and jumping of the windows when scoreboards are changed, was. The specific implementation of the "all windows are to be resizable" requirement makes it that performance would not be lost on weirdly sized windows due to poor rescaling making UI elements more difficult to interact with, excluding extreme cases. Through these changes, the project now is better suited for practical real-world uses, and due to its improved quality of life would be more appealing to prospective buyers.