**Introduction**

The goal of this project was to create a fully functional connect 4 game. The game has several features. It can play either one player vs a CPU or one player vs another player. It can also save the game at any point and then load that game save back. Our main reason for making a connect 4 game was because we both enjoyed connect 4 as kids and wanted to bring it to a computer game.

**Proposed method / System description / Implementation**

We started the design by looking at the game and identifying 3 main objects that could be symbolized with a class, the game pieces, the columns and the game board. We created a main game class called Connect4 that we used to manage the whole game. In total, we have four classes: gamePiece, gameColumn, gameBoard, Connect4. As you can see in the class diagram, gameColumn includes gamePiece and gameboard include gameColumn and Connect4 includes gameBoard.

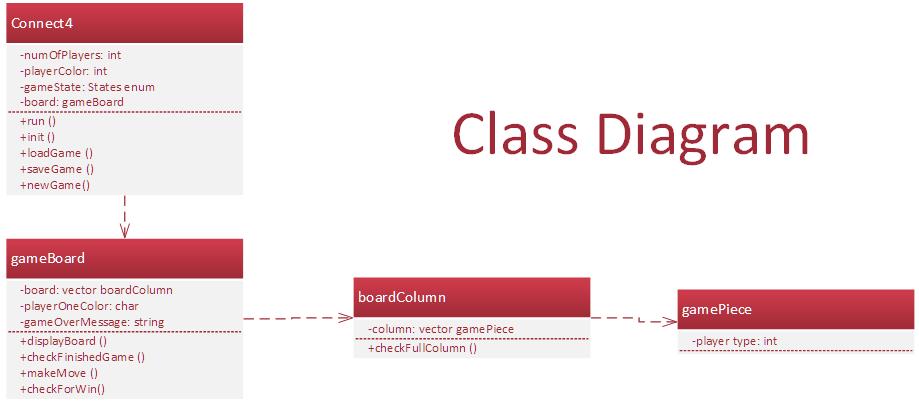
We designed the classes to stack on top of the ones before it. The gameColumn class creates a vector of gamePiece objects. The gameBoard class creates a vector of gameColumn objects. The Connect4 class manages all the logic and displays the UI. The gameBoard class handles checking if there are four piece in a row on the board. It also has a method to display the board on the screen. It also has a method to make a move. This method has a lot of logic to check if the column chosen is full or if the entire board is full. It also calls other methods to check if a connect four has occurred.

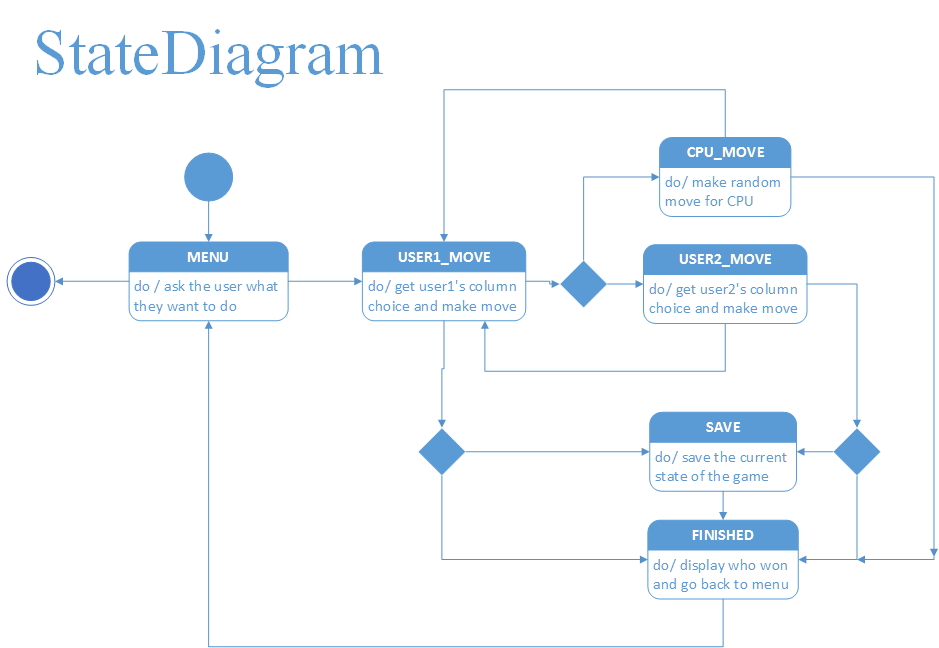
In the Connect4 class is the main game loop in a method called Connect4::run(). We designed the looped on what is called a state machine. The loop consists of a single infinite while loop with a single switch statement inside of it. The cases of the switch statement are defined by and enum containing all the states of our main loop. The states are as follows: INIT, USER1\_TURN, CPU\_TURN, USER2\_TURN, SAVE, LOAD, FINISHED. Inside each state in the switch statement we perform the required action for the given state. For example, in the USER1\_TURN state the program asks the user1 for the column they’d like to play their game piece in and then does the proper error checking and makes the move. At the end of each state there is code to decide which state should be executed next. It then changes the state variable accordingly and the while loop loops again. See the state diagram attached to this report for further information.

Most of the input and output is done with the standard cin and cout. The problem we ran into was while the user is making a move we couldn’t use cin and at the same time get input from the Arduino because cin is a blocking function. We settled on a function from the Windows API that gets the current state of any key on the keyboard. This allows us to check both the keyboard and the Arduino buttons at the same time.

Our program is very reliable. Together, we’ve spent tens of hours trying to break the system. Any bugs we did find we patched. We are certain there are no remaining bugs in our program. One thing we had to make sure wouldn’t break the game was unplugging the Arduino in the middle of the game. We’ve got it designed so that if it does get unplugged then the program will just forget there ever was an Arduino and only get input from the keyboard.

We went through several iterations. The first was the most unpolished. We through together some code to see if our ideas would work. This first iteration only had player vs CPU, it didn’t have any save features or even a main menu. During the second iteration, we refactored quite a bit of code to make it easier to read. We added a main menu with options to start a new game, load a saved game, and display the instructions. The main game loop was also rewritten. In iteration one the main loop was very convoluted and hard to follow. In iteration two we redesigned the main loop using a state machine model. This made the code easier to read and the flow of the program easier to follow.





**Discussion and Conclusion**

Our final product ended up just as we had planned it. One problem we encountered was trying to get the keyboard input and the Arduino input at the same time. This was a problem because the standard cin is a blocking function that doesn’t let any other code run until the user has entered something. This wouldn’t work for us because the Arduino could be used to select the input instead of the keyboard. We ended up using a function from the windows api to get the states of the keyboard. Then we wrote some code that said when the state of the key changes from pushed down to released then return said key. Similar code was written for the Arduino buttons.

One weird problem with our program is that during the game since we are getting input with the win api function the input buffer is still filling up with key presses so then at the end of the game when we ask the user to hit enter to go to the main menu and use cin to get that input all the character that were typed throughout the game are then displayed on the screen. The user then must backspace through these characters before they can enter anything useful. We couldn’t find any way of clearing this. There were several suggestions on the internet, but none worked.

As far as the previously expressed problem the program works flawlessly as expected. In the future, all the input could be taken from the Arduino with more buttons. Overall, we had few problems and could implement the code the way we had originally planned. This was a fun project that was at first challenging until we made the class diagram to help aid in the design process.