**Implementing the game of Battleship with an Arduino**

Aiden Van Dyke and Jacob Burritt

Salt Lake Community College

CSIS 2810: Computer Architecture

Professor Hau Moy Kwan

April 20, 2022

**Table of Contents**

|  |  |
| --- | --- |
| Abstract | 3 |
| Introduction | 4 |
| Issues | 4 |
| Solutions | 5 |
| * Getting Player Input | 5 |
| * Keeping the Player Informed | 6 |
| * Visualizing the Game Grid | 6 |
| * Creating a Challenging Computer Opponent | 8 |
| * Managing Memory | 10 |
| Conclusions | 11 |
| Appendix | 12 |
| References | 13 |

**List of Figures**

|  |  |
| --- | --- |
| Figure 1 - Analog joystick dead zone diagram | 5 |
| Figure 2 - Serpentine vs traditional matrix | 7 |
| Figure 3 - Probability grid at game start, search mode, and destroy mode | 9 |
| Figure 4 - Battleship wiring schematic | 12 |

**Abstract**

When implementing the classic game of battleship as a one player game on an electronic device such as an Arduino, there are a few hurdles that must be overcome. These problems include getting user input, displaying output to the user, visualizing the game grid, creating an effective opponent algorithm, and keeping memory usage within the low amounts that the Arduino has. Through the use of off the shelf input and LCD hardware, a custom LED matrix, probability, and careful use of strings and libraries, these problems can be overcome in order to create a functioning and fun Battleship game for one person**.**

**Introduction**

This paper will go over an implementation of the classic game of Battleship running on an Arduino. First, a quick overview of the game rules. Typically, the game is played on two 10 by 10 grids of squares. Each player places five ships on their own grid -- one that takes up five spaces (Carrier), one that uses four spaces (Battleship), two that use three spaces each (Submarine and Cruiser), and one that uses two spaces (Destroyer). Players take turns selecting a single space to shoot at, and then are informed if that shot hits their opponents ship or misses. If they hit all squares that a ship is in, the opponent tells them that they sunk their ship, and tells them the name (and thus the length) of the ship that they just sunk. The first player to sink all five of their opponents’ ships is the winner. So in order to facilitate these rules, the goal of this project is, besides implementing these rules in hardware and software, to replace one of the players above with a computer algorithm, allowing players to play against a challenging opponent any time they wish.

**Issues**

When implementing Battleship as a one player electronic game on an Arduino, there are several challenges that must be overcome. These five problems are, in order:

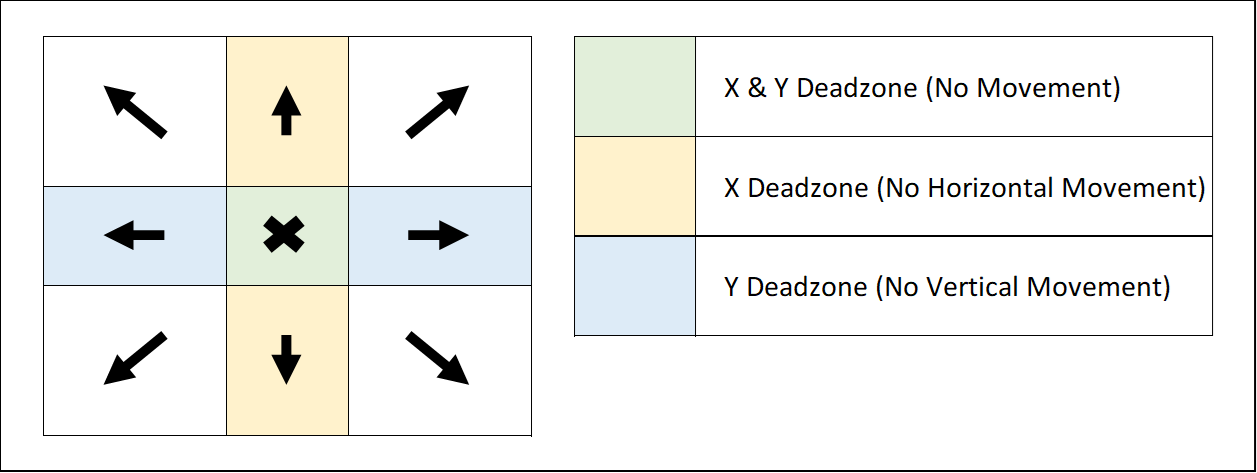
* How to get user input
* How to inform the user of changes in state
* How to display the game grid
* How to implement the algorithm that the computer opponent will follow,
* How to fit all this on the limited RAM of an Arduino

This paper will discuss these issues and possible solutions below.

**Getting Player Input**

When using an analog joystick for directional input, only two pins are required, instead of the four digital pins that would be needed for a traditional d-pad composed of push buttons. These two pins each send an analog signal, representing the joystick x and y position from the bottom left, reading from 0 to 1023. So a centered stick would have a value of 512. This could be fed into the cursor code, which would check if the value was greater or lesser than 512 in order to determine if the cursor should be moved, but it would be very sensitive and therefore frustrating to players. Instead, the cursor was only updated if the joystick x or y value is outside of a dead zone value that is added to the center value, as shown below.

*Figure 1 - Analog joystick dead zone diagram*



Compared to the joystick, the buttons are relatively simple. However, while a normal button would require a pulldown resistor added to ground in addition to a 5 volt and input wire, by using an internal pullup resistor in the Arduino though pinMode(pinNumber, INPUT\_PULLUP) for each button, the pulldown resistor can be eliminated, thus simplifying project wiring. One only needs to keep in mind that using a pullup resistor requires a check for a LOW signal to determine if a button has been pressed, not a HIGH signal as with a pulldown resistor.

For this project, the directional input was used to move a cursor. After the cursor is incremented on either the x axis, y axis, or both, it should not be allowed to go outside the bounds of the game board, as this would likely result in an array out of bounds exception and/or cause unexpected behavior. In order to achieve the wrap-around behavior that is desired, one can go through for each axis, adding the axis width of the board to each axis value of the cursor, then dividing each resulting value by each axis length. When this is done, the resulting remainder will always be where expected, wrapping around to the other side of the board if the new positions would otherwise end up out of bounds.

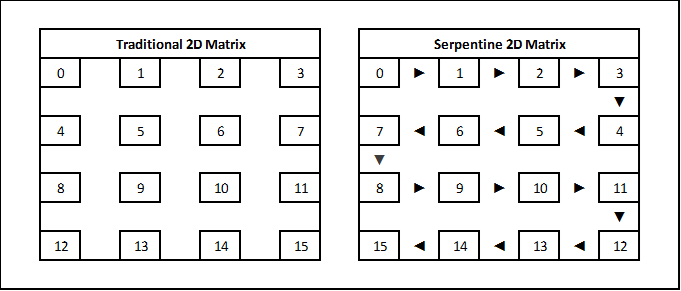
**Keeping the Player Informed**

Next, the issue of informing the user of the current state of the game and the results of their actions. This can be done with a cheap off the shelf HD 44780 LCD Module. Normally one of these modules would require at least 6 data pins connected to the Arduino. However, with a cheap ($1.50) I2C backpack adapter, one only needs to connect two data wires. While the reduction in wires was not ultimately necessary for this project, it didn’t hurt, and it can lead to an increase in redundancy and reliability, since there are lower chances of a wire being accidentally damaged.

**Visualizing the Game Grid**

Since an LCD is insufficient to show the state of the game grid, another form of input is also needed. An RGB matrix, composed of a strip of LEDs arranged in a two-dimensional array, can be used as an output grid for many tile-based two player games including Battleship. Normally, a matrix of 10 by 10 RGB LEDs would require at least 300 input wires (1 for each Red, Green, and Blue LED times 100 array locations) and 1 ground wire, which is far too many to be practical for most projects. However, by using off the shelf WS2812b LED hardware together with the Adafruit\_NeoPixel library, each individual LED can be controlled with just one data wire and one ground connection, vastly simplifying wiring and project code complexity. This is possible because each LED has a dedicated microchip that can read serial data from the data wire, and then pass on the next value to the next LED in the strip. Since LEDs can refresh much faster than the human eye can detect, they can also wait for all the data to come into each LED before updating the whole matrix. In any case, high speed refreshes, such as those for displaying video are not necessary, so the matrix speed is more than sufficient for our purposes as a virtual representation of a game board.

When using an LED strip wired in a matrix, it is often easier to solder them together in a serpentine pattern, with even rows running to the right and odd rows to the left, as shown below:

*Figure 2 - Traditional vs Serpentine indices in a two-dimensional matrix*

However, if unmitigated, this could lead to increased code complexity and confusion, as the index of each led in the matrix, represented in the LED library as a one-dimensional array of colors, does not match up to where one would expect on a traditional matrix of rows and columns. Fortunately, this was easily fixed by using a translation function to convert 2d coordinates to a 1d index that matches what one would expect, by mirroring every other row. Then the output of that translation function was used when writing to led array indexes as part of the led update code.

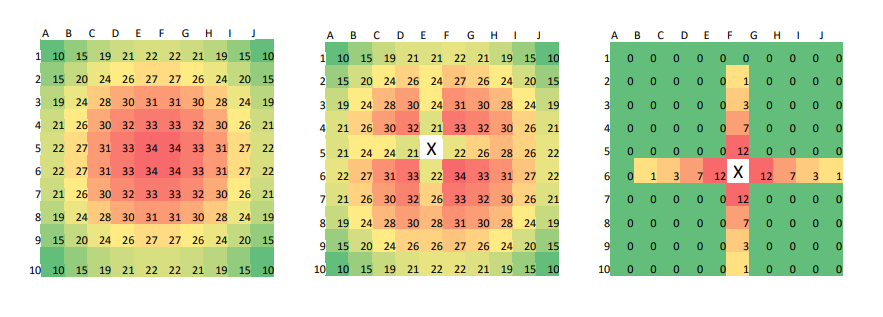
**Creating a Challenging Computer Opponent**

Perhaps one of the most important issues when creating a single player game is making sure that the computer opponent is challenging enough to provide a fun challenge. There were a few algorithms considered for guessing squares, such as just purely guessing randomly, but eventually one was chosen that is near optimal, to the point of being very difficult for an average human player to win against, even with ship arrangements that were likely to make the algorithm take longer, and with the computer player simply placing its own ships randomly.

Most people choose (mostly) random spots during what could be called search mode, and then switch to a destroy mode if they score a hit, checking in each direction from the original hit for additional possible hits. However, when making an algorithm for a computer opponent, an even better approach is to check the probability of each unguessed square containing a part of a ship, and to then only guess a square S such that the probability of S containing a ship is maximal, i.e., not exceeded by the ship probability of any other square. If multiple squares exist that have this same highest probability, one of those squares will be selected randomly. Like human players, the computer will also be in search mode if there are no hits corresponding to an un-sunk ship and switch to destroy mode if any such hits are found. During search mode, the probabilities of all the unguessed squares on the board are considered, whereas in destroy mode, the only squares considered are those on which an un-sunk ship lying on any of the hit squares could hit.

In order to generate this probability grid, which happens at the start of each turn, the algorithm first checks that all ships left in play have not been hit. If this is the case, the algorithm will be in search mode, trying to find a ship. It will check for each ship that has not been sunk, in both vertical and horizontal orientations, for each origin square on the board, if a ship can be placed from that origin point going right or down. If it can, then every square that that hypothetical ship would cover has its probability value incremented by one. Thus squares that can fit more ships in different positions have a higher value, such as the middle of the board at the start of the game.

If a ship has been hit, but has not yet been sunk, then the algorithm will switch to destroy mode. In this case, the probability grid is generated in a similar manner as in search mode, except that it only considers ships that are still in play that could pass through hits that are not part of sunken ships, and the only squares that can be guessed are the unguessed squares on which these ships could lie. This lets the algorithm focus on choosing ships that are most likely to be part of a ship that has not been sunk. The algorithm will stay in this mode until all ships that have been hit have been sunk, and then it will return to search mode if there are any ships left to sink. The figure below shows some examples of how the probability grid could look at different stages of the game.

*Figure 3 - Probability grid at game start, search mode, and destroy mode.*

*Images from Boyd & Boyd.*

Thanks to this probabilistic approach, a challenging computer opponent can be created without too much hassle or memory usage. There are quite a few iterations that are computed on each turn, but since the grid is only ten by ten and contains only five ships, the computational complexity is manageable, even for an embed device such as an Arduino. Additionally, these calculations could likely be dramatically reduced in an improved algorithm by only considering the squares that could have changed based on where the hit or miss was. In any case, this algorithm is surprisingly challenging to play against, and was not too difficult to develop.

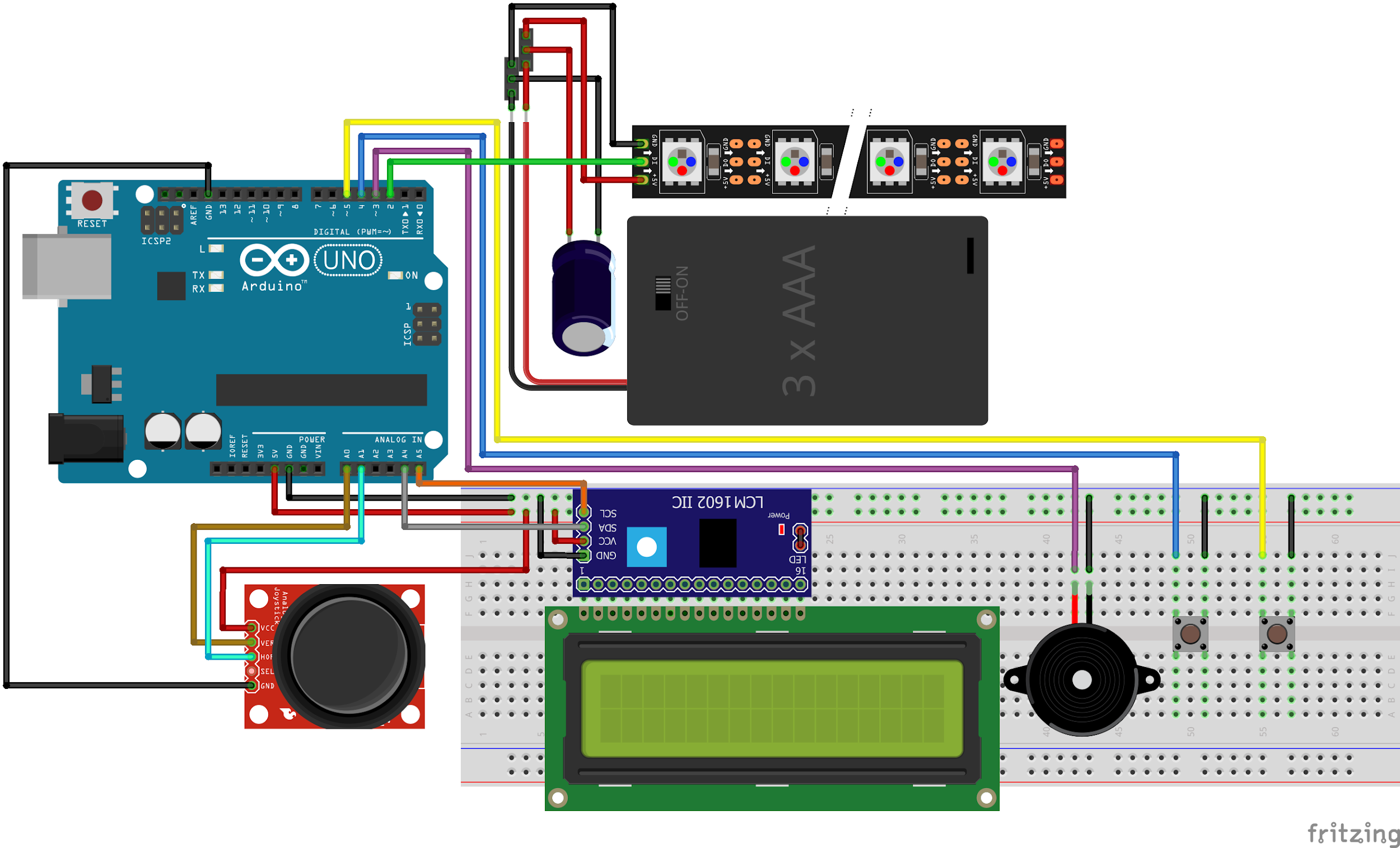
**Managing Memory**

Lastly, one must make sure that there is enough RAM for all the above. In a Java program running a simulation of Battleship, there might be many gigabytes of RAM available, but on the Arduino UNO, there is only 2 kilobytes of dynamic RAM. This means that memory usage must be carefully managed. In order to fit within these memory constraints, the implementation ended up doing two or three things differently than it might have otherwise. First, it uses the Adafruit\_NeoPixel library instead of the FastLED library, which results in a significant savings of RAM with no loss of features for this use case (Jake, 2019). Next, instead of using String objects, which can cause heap fragmentation, C style null terminated char arrays are used instead. Finally, since most strings are game responses to predictable results and thus don’t need to change, they were stored as constants. This means that they could also be stored in program memory instead of dynamic memory, and then pulled into dynamic memory only when needed. This resulted in a large RAM savings of about 20%. Without these optimizations, it’s possible that the array requirements of the computer opponent would have run into hard to troubleshoot memory issues or may have been just too big to fit into RAM.

**Conclusions**

In summary, creating an electronic battleship game can be a challenging hurdle, but one that is achievable with research and planning. By using dead zones for analog input, internal pull up resistors, and an I2C backpack for a LCD display, inputs and outputs were made easy to use and wire. Also, with a custom wired RGB Matrix, a bright and easy to understand representation of the game grid was created. Next, a computer opponent was made challenging by computing probabilities. Finally, memory was kept in mind during software development in order to fit all this into 2 kilobytes of dynamic RAM. Though these challenges applied to the game of battleship that was implemented here, they can also apply to other games and projects, such as LED signage or analog input methods. By keeping these solutions in mind, one can use the lessons learned here to improve other such projects, possibly for much cheaper than off the shelf components.

**Appendix**

*Figure 4 - A wiring schematic for the completed solution*

**References**

Berry, N. (2011, December 3). Battleship. *DataGenetics*. <http://www.datagenetics.com/blog/december32011>

Boyd, J. [Jacob], & Boyd, J. [Johnathan]. (2016). Battleship Playing Program Utilizing Probability Density Functions. *Kent State University*. <https://www.csuohio.edu/sites/default/files/82B-2016.pdf>

Jake. (2019, June 2). ESP32: FastLED vs. NeoPixelBus vs. NeoPixel Library*. Jake’s Blog*. <https://blog.ja-ke.tech/2019/06/02/neopixel-performance.html>