Final Project Report: Evolution in Flappy Bird ML

By Ethan Polley

Saturday, December 14, 2019

CSCI0150

<u>Handin:</u> On-Time
<u>Containment Diagram:</u> Evolution.pdf

<u>Project Design Summary / Choices</u>

My project had three primary game modes, each using different subclasses:

<u>Single Player, Non-AI Game:</u>

- Standard version of Flappy Bird Game
- Game starts by pressing the space bar
- Space bar makes bird jump

<u>Single AI Game</u>

- Typical AI Game
- *AIMenuOne* and *AIMenuTwo* allows user to specify the Neural Network configuration, the fitness settings, and the AIInputNodes
- Has BirdViews that represents AIBird's
- Left Info panel displays info regarding the current generation, number of birds remaining, average fitness of the last generation, and the maximum fitness from the last generation

<u>MultiScreen AIGame</u>

- Displays 9 different instances of AIGames
- Shows the info regarding each game in the GameView's on the left side.
- Allows user to add and delete games from the screen, as well as change the game settings on the left side of the screen.

<u>Optimization and Training</u>

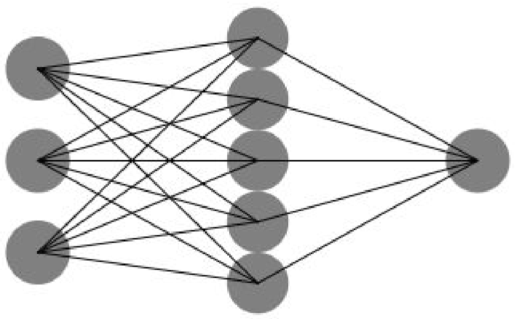
<u>Optimizing the neural network configuration:</u>

The first step in optimizing the neural net configuration was finding the optimal inputs. Though I had many choices (see *AIMenuOne*), I deemed that the optimal set of inputs would only include the inputs necessary for the AI to make its decision: any other inputs could hinder its decision-making process. Thus, I settled on using the birds Y-dist to top of pipe gap, y-dist to bottom of pipe gap, and the x-dist to the pip as inputs. The user can change these inputs and experiment themselves using *AIMenuOne*.

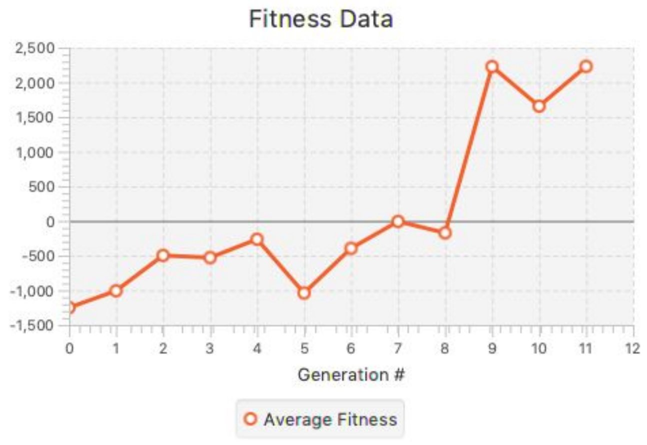
Because I made my neural networks configurable, I needed to find the optimal neural net layout. Using the three inputs that I initially determined as necessary for the bird to make its decisions, I simulated different neural networks, varying the number of layers and the size of each layer. These were the configurations I simulated, as well as the results of these simulations:

Test 1:

Neural Net Config:

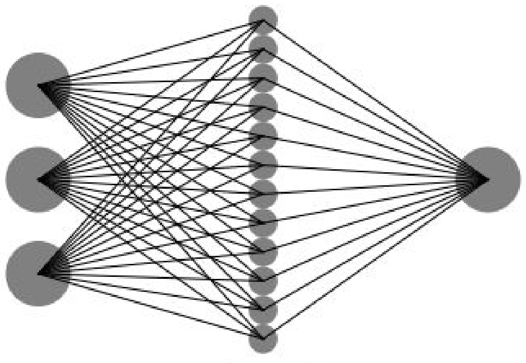


Average Fitness Graph:

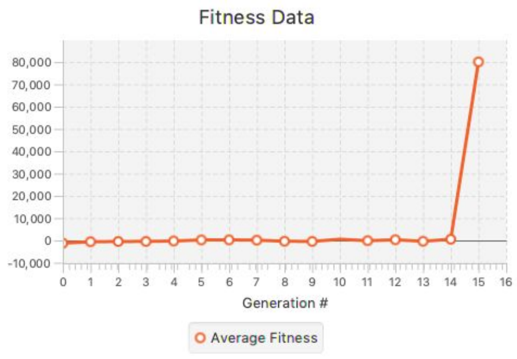


Test 2:

Neural Net Config:

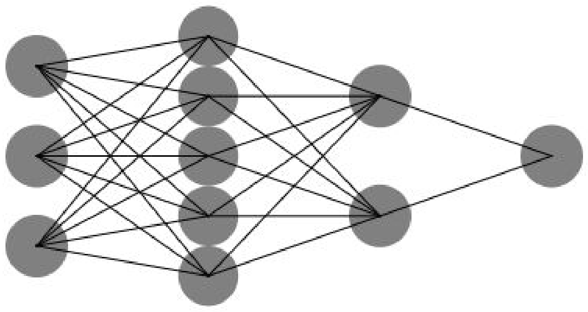


Average Fitness Graph:

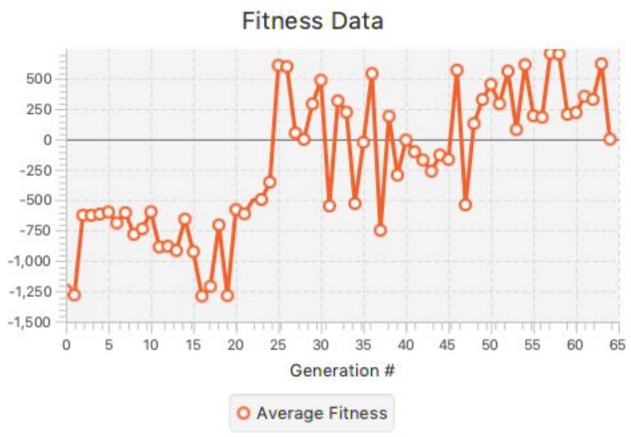


Test 3:

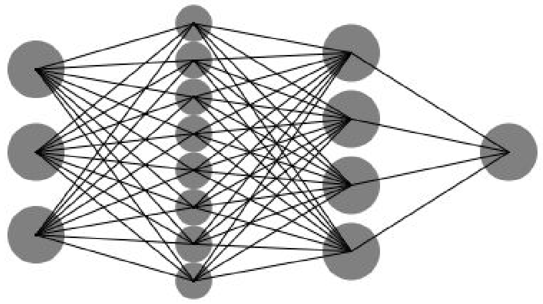
Neural Net Config:



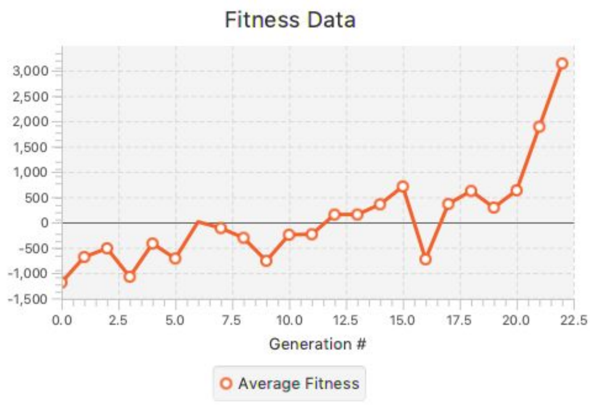
Average Fitness Graph:



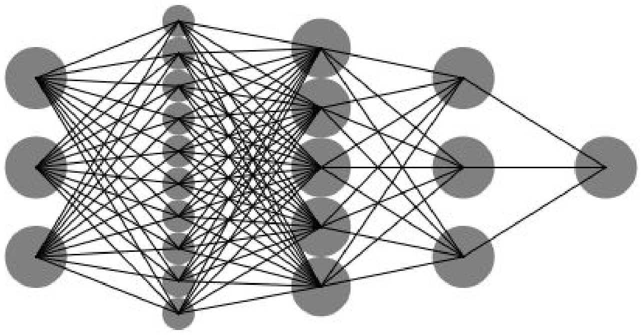
Test 4:
Neural Net Config:



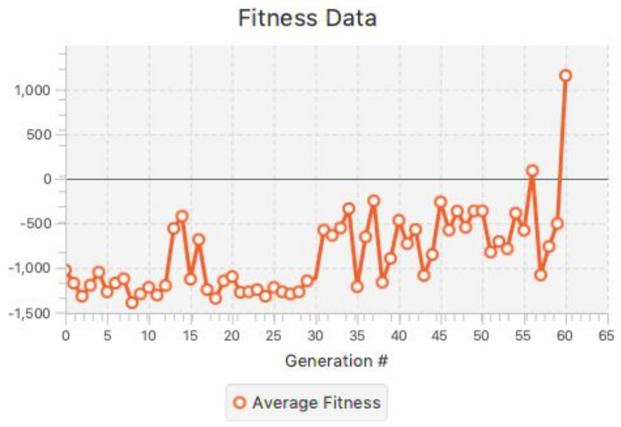
Average Fitness Graph:



Test 5:
Neural Net Config:

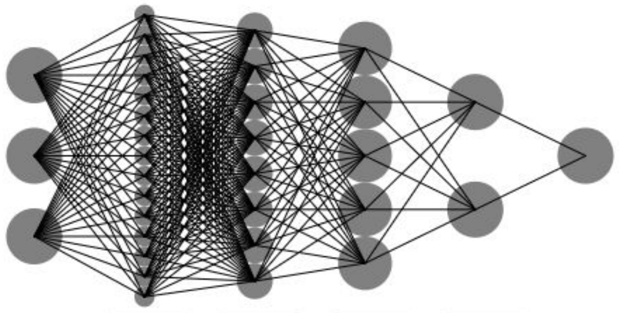


Average Fitness Graph:

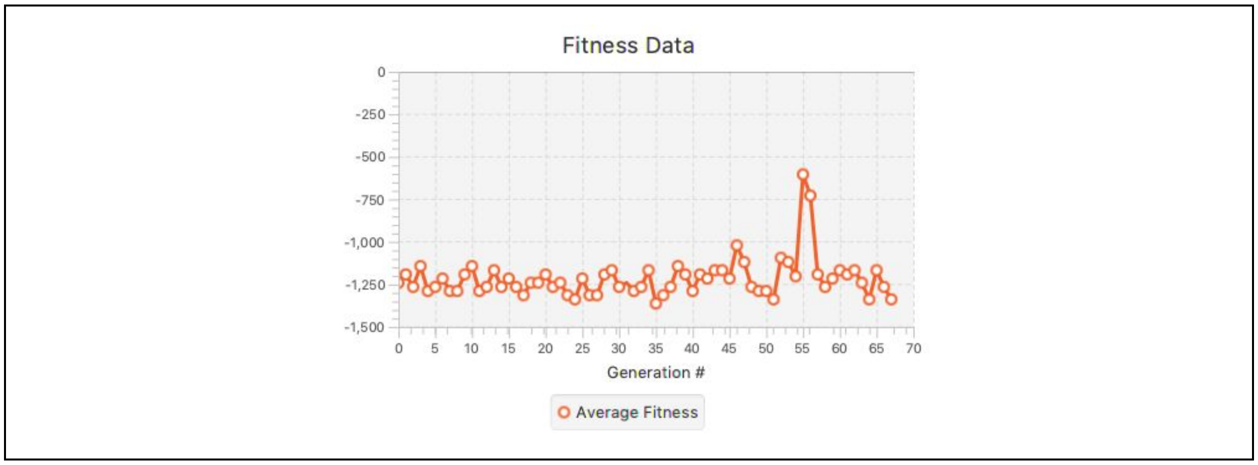


Test 6:

Neural Net Config:



Average Fitness Graph:



The synthesis of the results of these tests is quite interesting. I noticed that with only 3 inputs, the number of layers and nodes on each layer needed to be limited. As shown in **Test 6**, adding too many layers and nodes does not allow the birds to make substantial progress. I believe that this is largely because adding more layers and nodes makes the decision-making process more abstract for the Neural Network. Instead, the number of layers and nodes should be limited. Thus, I chose to use **Test 4** as the optimal and default configuration for the NeuralNetwork. However, the Neural Network configuration for a game can be altered in *AIMenuTwo*.

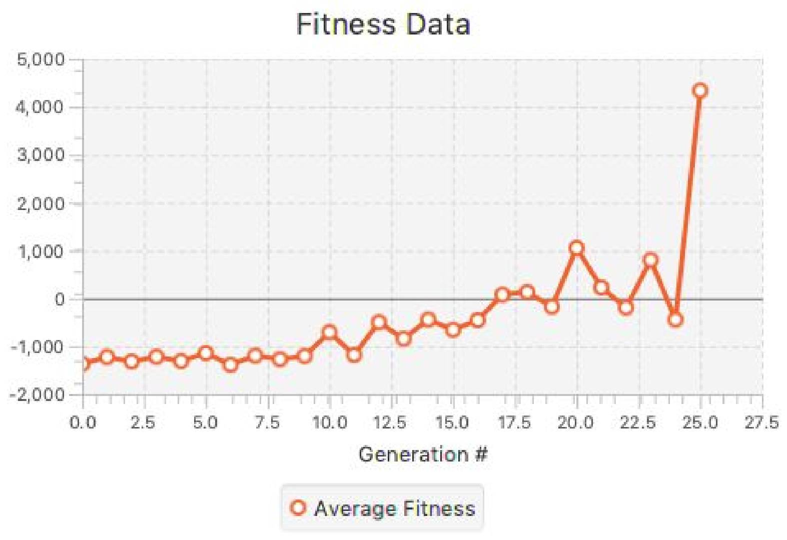
Optimizing the training and selection settings:

Once I found the optimal configuration for my neural network, I used this configuration to find the optimal selection settings for my network. My selection settings primarily involved four values:

- **Keep Rate:** this value specified the rate at which birds would be saved. All of the birds or mutations/crossovers of birds that appear in the next generation will be made out of birds that are in the top percentile of the previous generation. This percentile is the keep rate.
- **Direct Select Rate:** the direct select rate specified what percentage of birds in the new population would be a direct copy of a bird from the previous generation
- **Direct Mutate Rate:** the direct mutate rate specified what percentage of birds in the new population would be a mutated copy of a bird from the previous generation
- **Crossover rate:** the crossover rate specified what percentage of birds in the new population would be the product of two birds from the previous generation having their weights split at a specific point and then mixed together. This point is determined randomly using *Math.random()*.

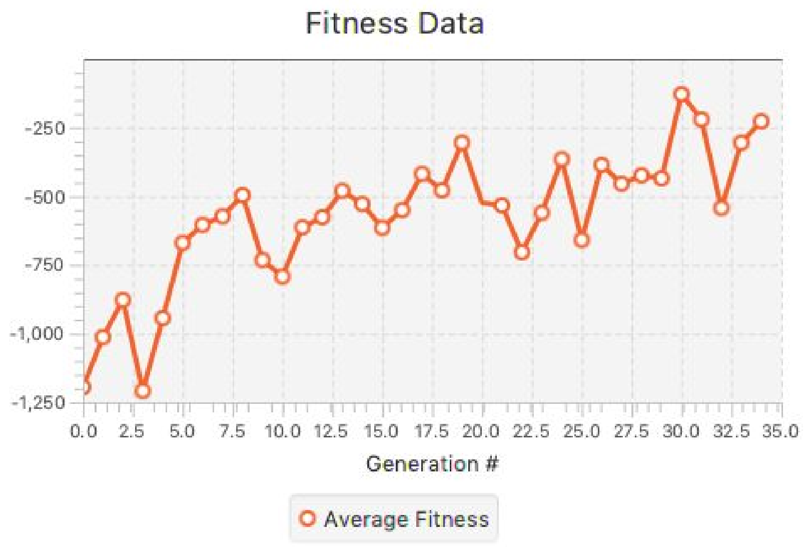
Testing the effectiveness and impact of these different selection methods helped me determine my optimal selection settings. Some charts from the tests I ran are included below, along with the settings that produced these charts.

Test 1:



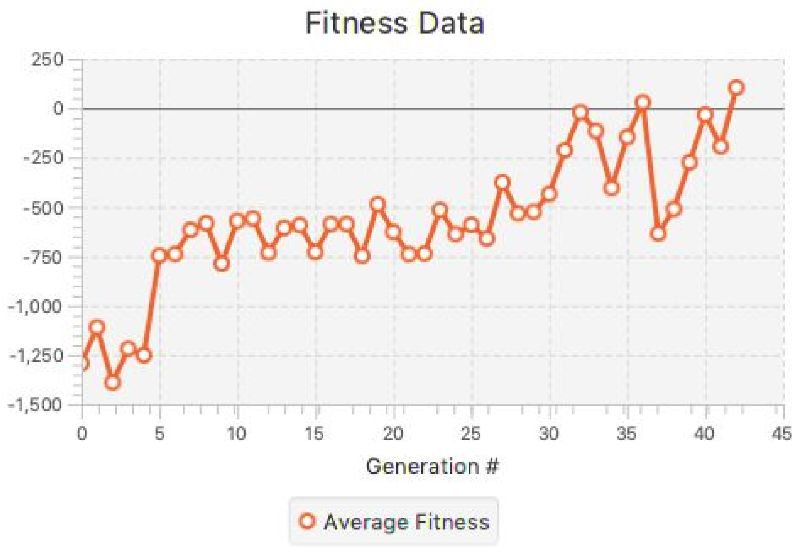
Keep Rate: .4
Direct Select Rate: .2
Direct Mutate Rate: .4
Crossover Rate: .4

Test 2:



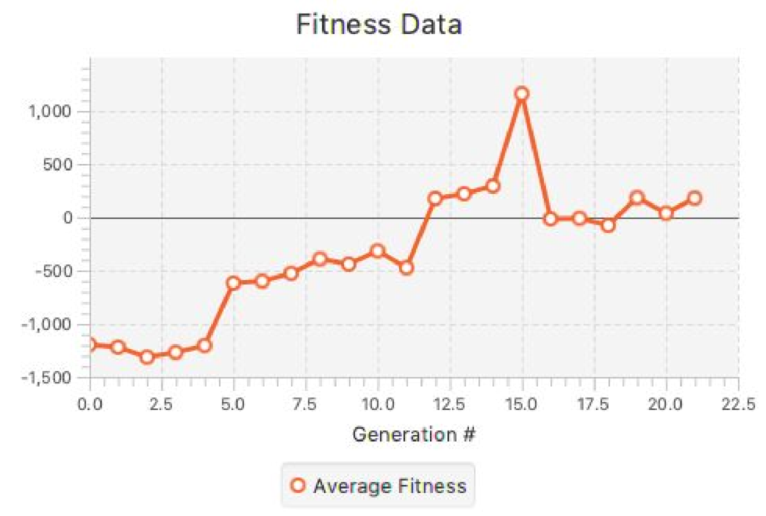
Keep Rate: .8
Direct Select Rate: .2
Direct Mutate Rate: .4
Crossover Rate: .4

Test 3:



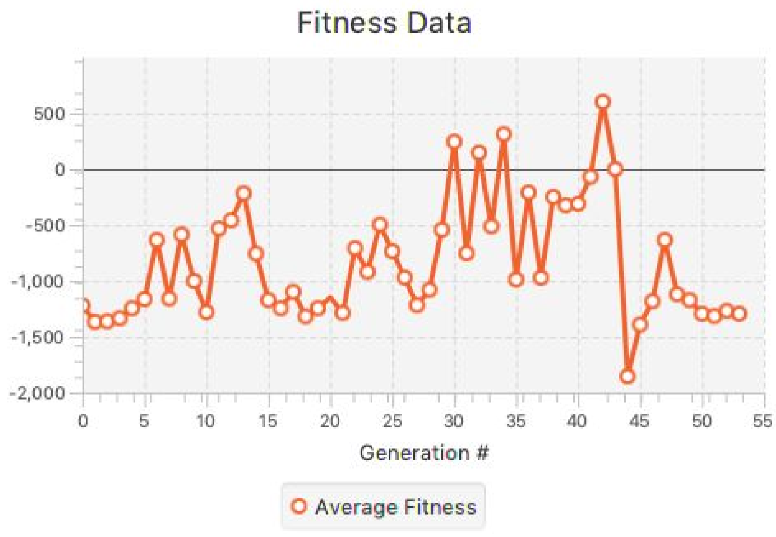
Keep Rate: .4
Direct Select Rate: .6
Direct Mutate Rate: .2
Crossover Rate: .2

Test 4:



Keep Rate: .4
Direct Select Rate: .2
Direct Mutate Rate: .2
Crossover Rate: .6

Test 5:



Keep Rate: .4
Direct Select Rate: .2
Direct Mutate Rate: .6
Crossover Rate: .2

Throughout the trials, I observed various different impacts of the different settings. First, the keep rate primarily controls the variation. If the keep rate is too high, bad birds from the previous population can move on. However, if it is too low, there will not be enough genetic variation in the new population. Secondly, the direct select rate allows for the best birds to stay alive. If there were no direct selection, we would risk losing the best birds to bad mutations or bad crossovers. Thirdly, the direct mutation rate allows for birds who are near the top, but not the best, to make minute changes in order to optimize their fitnesses. These birds still must be from the pool specified by the keep rate, so they will be in the top percentiles of their population, however, through slight mutations, they optimize their fitnesses. Finally, the crossover rate is more of a fun addition. This idea hails straight from genetics and allows two parents to make a "child" that shares genes (i.e. weights) from both "parents."

In **Test 1**, all of the values were somewhat equalized, allowing for a balance between genetic variation and allowing the best birds to move on. In **Test 2**, I maximized the keep rate. This meant that too many birds were in the selection pool and average fitness did not improve quickly enough, In **Test 3**, I maximized the direct select rate. This meant that although many of the top birds moved on, there was not enough variation for fitness to increase. In **Test 4**, I maximized the crossover rate. This made for too much genetic variation and thus fitness did not improve consistently enough. In **Test 5**, I maximized the direct mutate rate. This led to too much variation between generations.

Based on these trials, I believed that **Test 1** had the quickest and most consistent growth of average fitness. Thus, my optimized values were:

Keep Rate: .4
Direct Select Rate: .2
Direct Mutate Rate: .4
Crossover Rate: .4

<u>Other Optimization settings:</u>

In addition to testing the selection settings and the neural network configuration, there are other settings that can be modified to change the outcomes of these populations. The primary setting that I modified was the equation for each birds fitness. Initially, I believed each bird's fitness would simply be its X-distance covered, as the goal was to get the bird as far as possible in the X-direction. However, I quickly realized this would not work for one simple reason: some generations simple evolved so that each bird would get to the first pipe, however, they would die bumping into it. Because of this, I created the *FitnessCalculationFactors* and *FitnessCalcFactor* classes and added a way to modify these classes to *AIMenuOne*. This way, I could experiment with the way that each bird's fitness was calculated. Quickly, I realized that the optimal fitness algorithm not only had a positive correlation between the birds' x distance and the fitness, but there was also a negative correlation between the birds' distance x distance to the next pipe, as well as a negative correlation the birds y distance to the center of the next pipe. Thus, birds would be rewarded for getting as close the center of the next pipe's gap as possible. My final iteration of the optimal fitness calculation is as follows:

$$fitness_{(bird, pipe)} = [1 \cdot x_{bird}] + [-1 \cdot (x_{pipe} - x_{bird})] + [-3 \cdot |y_{pipe} - y_{bird}|]$$

<u>**Extra Credit/GUI Enhancements**</u>

- User can change game settings (i.e. Pipe gap, gravity, bounce vel) in real time.
- *AIMenuOne* and *AIMenuTwo* allow user to specify the Fitness Calculation Settings, the Input nodes, and the Neural Network Configuration
- Neural Networks with more than one layer
- Neural Network Editor class, allowing user to customize the configuration
- Regular Flappy Bird Option
- BirdView: allows user to see the top birds in an AISingleGame, view each birds inputs and output, see when they are going to jump. Plus, user can kill off each bird or view their NeuralNetwork as either a live representation of node values or a static representation of the weights associated with their nodes.
- Wrote my own network visualizer with TWO options:
    - View static weights, as represented by red and green lines with different thicknesses
    - View live values of each node on the Neural Network: the opacity of the circles represents the current node value.
- Used Sprites for pipes and birds
    - Each bird has different color, allowing user to identify different birds
- Wrote my own fitness visualizer. Allows user to see the best fitness, worst fitness, average fitness, as well as all of the settings over time.
- Displays multiple instances of evolution in one screen (*MultiGameScreen*).
    - Allows user to see the info regarding each game in *GameView*.
    - User can add and delete games from the screen, each having its own Fitness Settings, Input nodes, and Neural Network Configuration

Known Bugs

- On the Multi Screen Game, adding a game occasionally deletes the wrong button from the Gridpane.