Ultimate Tic-Tac-Toe

Flynn Michael-Legg
Noah D’Souza
Software Design Spring 2018

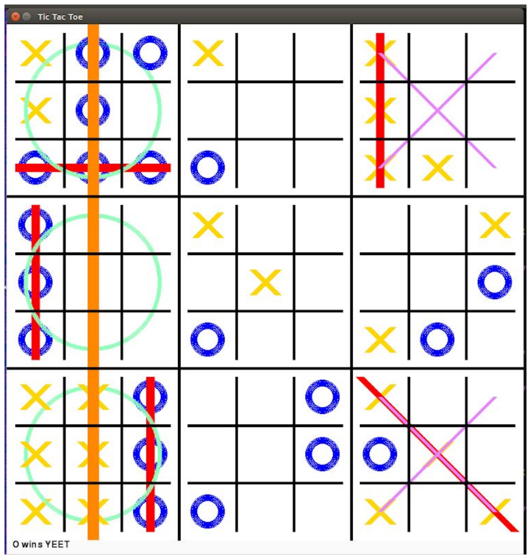
Project Overview

This project implements “Ultimate Tic-Tac-Toe”, a version of regular Tic-Tac-Toe in which each square contains a smaller game of Tic-Tac-Toe within that must be won before the larger square can be claimed by X or O. An added twist is that when a player puts their mark on one small square, the other player’s move should be in the small grid within the corresponding large square (for example, if Player 1 puts an X in the top-left of a small square, then Player 2 must put an O somewhere in the grid located in the top-left large square).

Results

We have developed a functional version of Ultimate Tic-Tac-Toe for two players. Both players take turns on the same machine clicking on their desired small squares in accordance with the rules of the game. When a player clicks on a square within a smaller grid, their corresponding symbol (x or o) is drawn in gold or blue, respectively, on that square.

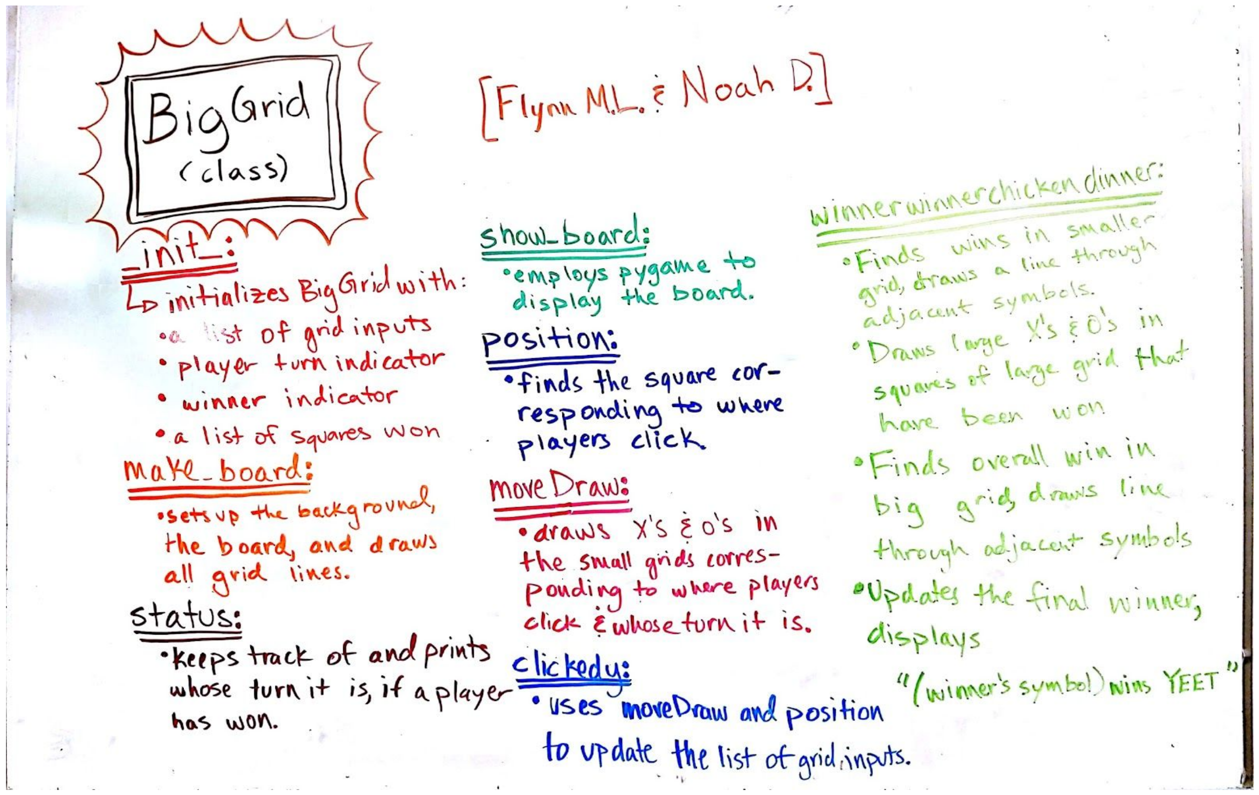
To win a square on the large grid, a player must obtain three consecutive symbols (either horizontally, vertically, or diagonally) the smaller tic tac toe grid within. If a player wins a smaller grid, a red line is drawn through their three marks. Then a larger symbol of a new color is drawn over the square of the larger grid. To win the whole game, a player must achieve three squares in a row in the larger grid, at which point an orange line connects their adjacent symbols and the screen reads “(player’s symbol) wins YEET.”



O wins YEET

Implementation

Our code employs only one class, and so a UML of our code would be a boring block. Since this diagram would not be useful, we will instead provide a diagram of the methods contained within our BigGrid class below:


BigGrid (class)

[Flynn M.L. & Noah D.]

init:
↳ initializes BigGrid with:
• a list of grid inputs
• player turn indicator
• winner indicator
• a list of squares won

make_board:
• sets up the background, the board, and draws all grid lines.

status:
• keeps track of and prints whose turn it is, if a player has won.

show_board:
• employs pygame to display the board.

position:
• finds the square corresponding to where players click

moveDraw:
• draws X's & O's in the small grids corresponding to where players click & whose turn it is.

clickedy:
• uses moveDraw and position to update the list of grid inputs.

winnerwinnerchickendinner:
• Finds wins in smaller grid, draws a line through adjacent symbols.
• Draws large X's & O's in squares of large grid that have been won
• Finds overall win in big grid, draws line through adjacent symbols
• Updates the final winner, displays "(winner's symbol) wins YEET"

We then created a BigGrid item with the desired specifics, and ran it, allowing it to update itself upon each click with nested `while` and `for` loops. We made a lot of important design decisions with the display, choosing line colors, lengths and sizes deliberately so that players can understand the layered visual information. For instance, we chose lighter colors for the larger X's and O's because that minimized visual obstruction of the squares beneath (which can still be played upon after being won). Additionally, within the winner method, in loops describing vertical and horizontal adjacencies, we used nested `for` loops of `range` 9 and then 3 to encapsulate all small squares. In contrast, our nested `for` loops for diagonal adjacencies of all `range` 3. This saved computation time and made our code look neater, and helped to make the winner function more cohesive.

Reflection

Our process of coding collaboratively went very well overall. We pair-programmed throughout to minimize errors and merge conflicts. We also made sure all or most ideation and actual work was done at least in proximity to each other so that we could ask each other questions or get help if needed. Our project was accurately scoped for both of our skill levels. We did not meet any of our stretch goals, but we were able to finish and create a working, viable program on time. Because of frequent, incremental testing, all of our testing was done by either running the program and checking for errors and by adding in `print` statements to inform us of the outputs of various statements. We learned that it is important to develop a more concrete plan for coding before beginning, and will use this lesson in the future by creating better diagrams and pseudo-code before starting a project. It would have been helpful to have had a better idea of what kind of objects we would need to use for our program, but that could be solved with the lessons we learned.

We divided the work evenly, taking turns pair-programming in and out of class. Our original plan was to just use whichever strategy would be most efficient, and we both agreed that pair-programming was best suited for our work styles. We would switch seats when one person got tired, and made sure that both of us were doing something when meeting. Next time we could try switching more often while pair-programming to avoid fatigue and maintain more active engagement.