112 TERM PROJECT PROPOSAL

Project Description:

My term project is Kami! This is a version of an existing mobile/computer game called Kami 2. In Kami 2, the user is given a board of tiles, with different regions of tiles colored differently. The user can select a color at the bottom of the screen, tap a region on the board, and that region will change into the color that the user has selected. The goal of the game is to get the entire board to become one single color in as few moves as possible. My version will also include an auto-solving feature and allow users to create their own game boards.

Competitive Analysis:

An obvious competitor would be the game Kami itself. There will definitely be a myriad of similarities between my game and the original Kami, including the general use of triangle-shaped tiles to form the board, the rules of how the game is played and what a winning state looks like, and also a"board-maker" space, where the user uses different colors to create their own "levels". However, some main differences between my project and the original game of Kami will be that my project will come with an auto solver program - after a user creates their own board, they can tell the auto solver to analyze their board, and the program will output the fewest number of moves needed to win. The user can then test their skills to see if they can win their own board in the correct number of moves! This currently does not exist in the Kami game.

Additionally, there will be a hint system, where if a user is stuck, they can request the program to give them a hint of what the best next move should be. This is something that many current players of Kami have included in their feedback and reviews in the App Store and appears to be something that they want to have. Another thing that users have stated they would like to see is the ability to undo multiple moves. This is currently something that you cannot do in Kami (you can only do your most recent move), which means you must remember the moves

you have applied each time you refresh the game to try a different move. Therefore, my term project will aim to allow users to undo multiple moves when they play the game.

Structural Plan:

Graphics/animation functions

	Main purpose(s)
appStarted	Initializes all the starting variables.
MousePressed	Detects when the user is pressing on the board. This occurs either when: - The user is painting tiles to create their own board - The user is playing the board and wishes to fill a region
MouseDragged	Detects when the user drags across the board. This occurs when the user wants to color in multiple tiles at once during the "create" mode.
redrawAll	Draws the board in its given stored state as well as displays information such as instructions, the number of moves, the options for colors, etc.
drawBoard	This draws the board in its given state by calling upon the draw triangle functions for each row, col.
drawLeftTriangle	This draws a left-pointing triangular tile at a specified location on the screen. This is what forms half of the tiles on the board.
drawRightTriangle	This draws a right-pointing triangular tile at a specified location on the screen. This is what forms the other half of the tiles on the board.

Objects:

	Attributes	Purpose
Graph Object	- A dictionary mapping regions on the board (i.e. the nodes) to their neighbors	This object will store the graph representation of the board (i.e. the regions of the board and which regions each region is connected to)
Region Object	- The tiles that make up that region - The color - Its neighboring regions	This object will store each "region" of the board and the relevant information. This will be used to model the game board as a graph of nodes and edges, so that the solving algorithm may be applied to solve each board.

Other major functions:

	Purpose	Params and Return
Flood	This function will be called upon when the user clicks on a region of the board with a selected color. The program will flood the region with said color, stopping at the edges of the shape so it doesn't spill over to the neighboring space.	Row, col, color, clickedColor Returns None
getRowCol	This function will be called upon when the user clicks on a tile. The program will then utilize the coordinates of that click to calculate which row and column that click was situated on (i.e. determine which tile was clicked on based on the position of the mouse on the screen). This is a View-To-Model function.	app, x, y Returns (row, col)
getCoordinates	This function will be called upon to get the three points necessary to draw each triangular tile. The program will compute the coordinate based on a row, col index, and do calculations based on the width/height of the screen to see where the row's coordinate should lie relative to the given margins. This is a Model-To-View function.	app, row, col Returns coordinate points (x, y)
BFS auto solving algorithm	This function will be used to "auto solve" the boards given to the program. It will do so by analyzing the board and trying every possible move from its given state, repeating layer by layer until it reaches the winning state. It should then produce the path and/or the number of moves needed to reach the "win" from the starting board.	initialState, visitedStates, graph Returns path/number of moves

Algorithmic Plan:

The auto solver program

This is the most complicated aspect of my project. This algorithm aims to take any board of tiles and analyze the setup so that it outputs the fewest number of moves needed to get the board to its winning state. This will allow the user to essentially test out their own boards and see if they can win in the fewest number of moves! In order to accomplish this feature, I intend to represent the game board as a graph of nodes and edges and utilize a breadth-first-search algorithm to go through each stage of solving the puzzle, in order to find the path that leads to the solution using the fewest number of moves. Breadth-first-search will be implemented as follows. The first layer of the search would be the board in its starting state. The second layer would then be comprised of all the states of the board after applying a move (i.e. flooding a region with a color). If there are 4 regions on the board, and 3 possible colors to pick from, then for each region, I can apply 3-1 possible moves (i.e. try every color except for the color that the region itself is filled with, which is why we have -1). The BFS algorithm will visit each layer before moving on to the next until it finds the winning state (i.e. all tiles are flooded with one color!). This should produce the pathway to win the board with the fewest number of steps.

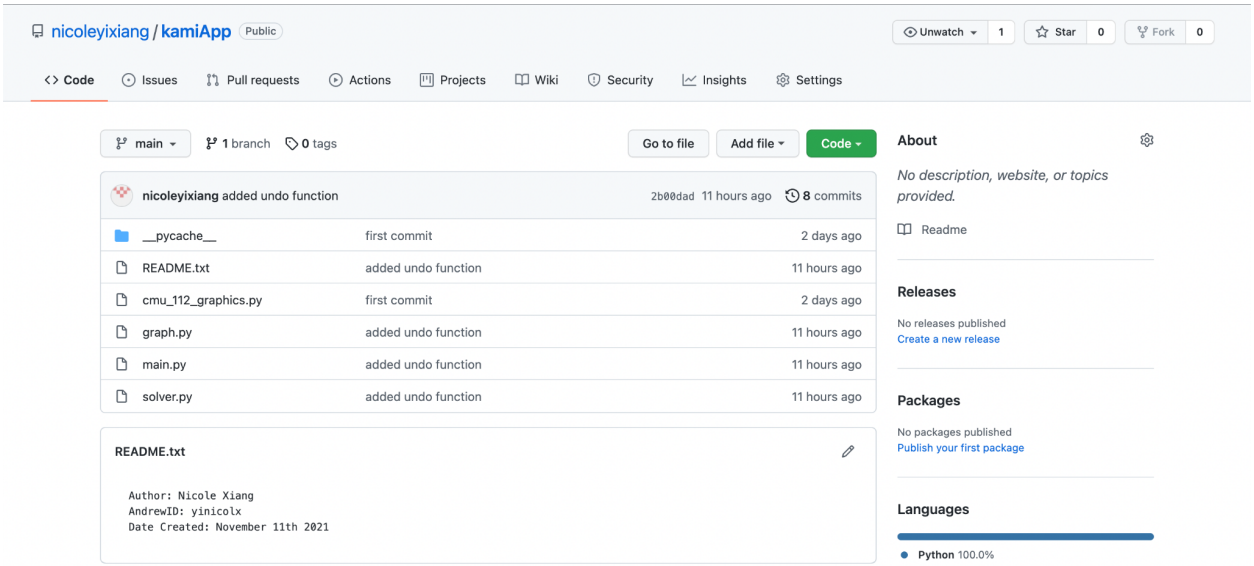
The flood filling algorithm

Flood filling is needed in order to allow the user to play the game: regions of the board need to change to the corresponding selected color when the user clicks on the screen. To accomplish the flood filling, I intend to represent my board as a graph of nodes (each node being a single triangular unit tile) and edges (edges will show how the triangular tiles are adjacent, i.e. connected to, another triangular tile). When the user clicks on a tile on the board, the program will implement a depth-first-search algorithm to recursively check neighboring tiles, filling said tiles with the selected color when appropriate. Essentially, this would be a recursive DFS algorithm: the program will flood the clicked triangle tile with the selected color, then call the flood filling function again on all of the triangle tile's neighbors, flooding those ones as well as long as they are not out of bounds of the shape.

Timeline Plan:

	Task	Expected completion date	Expected hours
1	Create a basic structure for representing the board internally and displaying the internal representation in a graphical manner to the user	November 13th	3
2	Create the drawing feature of the program that allows the user to fill in specific tiles with a selected color	November 14th	3
3	Create the flooding algorithm that will fill in tiles when a region is clicked on by the user	November 15th	3
4	Create the algorithm that detects the regions of a given board of tiles and (i.e. how many 'chunks' there are on the screen)	November 16th	2
5	Create the algorithm that finds the neighboring "chunks" of each "chunk" on the board and keeps track of these relationships	November 17th	4
6	Build the user interface and user experience of the project (i.e. add different modes and screens, make buttons that the user can click on, etc.)	November 18th	4
7	Create the algorithm that transforms a given board into a graph illustrating the nodes and edges between the regions of the board	November 19th	2
8	Complete the first version of the auto solver program	November 21st	8
9	Create multiple game levels and screens displaying each mode of the game	November 23rd	6
10	Beautify the game (i.e. improve UX) and run through user testing, fixing problems where necessary	November 26th	8
11	Fix lingering bugs and complete extensions beyond the original idea (if possible!)	November 29th	6

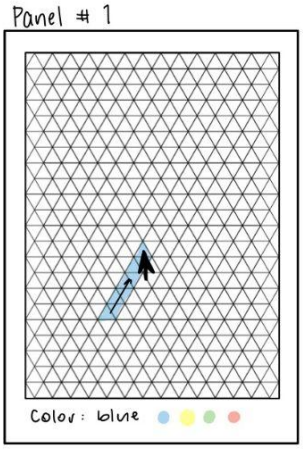
Version Control Plan:
In order to back up my code regularly, I plan on using Git/GitHub. I have created a repository (see image below) and have been able to successfully add, commit and push my code multiple times thus far. I will be pushing my code every time I make any major changes and/or after each work period to ensure that I have multiple versions to fall back on should I lose any data or need to revert to an older state of my project.



Module list:
For my project, I will be using cmu_112_graphics! No other external modules will be needed.

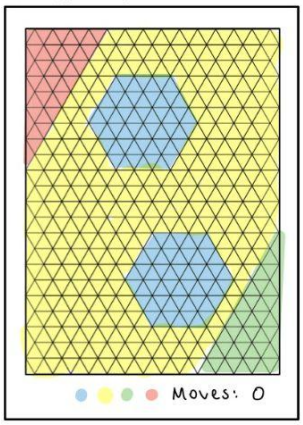
Storyboard:

Panel # 1



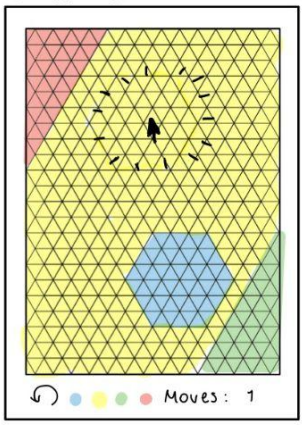
The user can draw/create their own Kami boards by coloring in tiles with their mouse (dragging).

Panel # 2



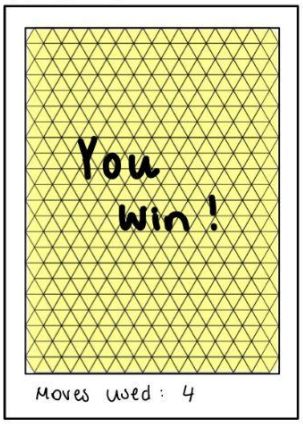
An example of what a starting board might look like.

Panel # 3



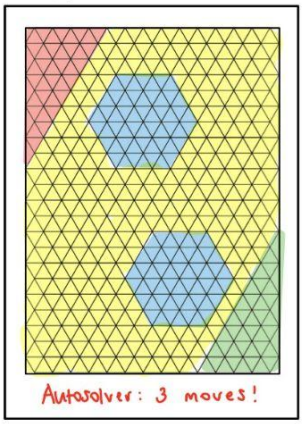
Selecting a color and clicking on a region will flood that space with that color. Undo button will bring the board back to panel # 2.

Panel # 4



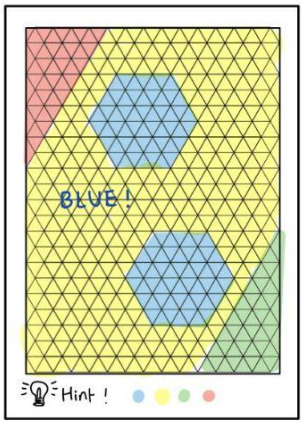
Board is "won" when entire screen becomes one color

Panel # 5



Autosolver program will display and calculate the fewest number of moves needed to solve the given board.

Panel # 6



Hint system will display the next best move when the user wants some help.

TP 2 Updates

In general, my design has not changed much from the last TP checkpoint! The one difference is related to the hint system I mentioned for the first proposal. Right now, the program will give out hints, but the hints aren't based on the auto-solver program. As in, it doesn't necessarily tell the user what moves they need to apply in order to solve the board with the fewest number of moves. It computes the suggested move based on analyzing the connections on the board and the amount of coverage each color has (i.e. the number of tiles each color covers). This works, but I do think it would be cooler if I can store the actual moves needed to solve the board in the fewest number of moves (i.e. store the pathway being taken to get to the solution from the auto-solver BFS algorithm)! I will attempt to accomplish this moving forward, and hopefully it will work out.

There are also a couple of things that I received as feedback during the User-A-Thon that I may or may not implement moving forward. I have listed these ideas below, just for reference!

Feedback from User-A-Thon:

1. Provide preset shapes in the draw mode that users can just "drag" into place (i.e. minimize the amount of drawing that they need to do manually)
2. A point/reward system for if the user manages to solve in the correct number of moves (complete it using only the amount of moves matching computed "fewest moves possible" from the auto-solver)
3. When drawing in the 'draw' mode, the application could "erase" what was drawn if you drag over a region of the same color that has already been drawn in (i.e. toggling - if a region is blue and I draw over it with blue selected, it will erase instead)
4. Allowing the users to choose their own colors before the game starts, instead of only having the four default options.
5. Being able to refresh the board (clear the board in draw mode)
6. Have regions that are essentially "empty" and cannot be drawn on. i.e. instead of only having rectangular shaped boards, the shape can be more interesting and dynamic.

Some other ideas that I may implement (based on the TP2 meeting):

1. Creating some "paper unfolding" graphics when the user clicks on a region (i.e. make the flood filling more "pretty")
2. In the draw mode, instead of drawing the border of each triangular tile (i.e. drawing the default "grid"), make the grid invisible but create outlines for shapes/regions once the user applies colors to the blank tiles.
3. Possibly upgrading the algorithms for solving the boards via Dijkstra's or A*.