Group H Planning Doc Essay

Throughout the semester we were given six coding assignments that all applied to a large program that we made. It was to be a maze game that involved tiles with the end goal to make the maze like the solution. This was divided up into six different assignments, and to help with the development process we planned out each one.

We started by exchanging contact information and created a group chat (text messages). This group chat is still in effect and used today. We exchanged schedules and decided to meet on Tues. and Thursday. Initially, we were meeting at 4 on Tuesdays, and 5 on Thursdays (this has been changed to 6 on both days since this became an online class). We met in the library for our meetings and checked out a room to meet in. In this beginning program, we worked on getting everyone setup with eclipse, and java. This included downloading, installing, and refreshing our knowledge on the basics of the language. This did not include setting the width and tabs vs spaces as that came at a later time.

To start our programming project we decided that we would each take a crack at implementing the code and would then meet to discuss our implementations. This included reviewing code, explaining code, proposing different ways of doing things, etc. Before the assignment was due, we, as a group, would pick the "best" implementation and merge it to master. The first coding review meeting we had only one person with functional code. As a group we tweaked it and made it much more presentable and readable instead of just being a functional unoptimized program. By utilizing creating classes and loops within these classes we made the code easier to maintain. We also started learning git at this time and while there were a few mistaken pushes to the master branch we worked it out in the end.

In program 2 we needed to give the quit button functionality (actually have the program quit), tiles had to be movable, and the user had to know when a tile was selected. With the exit button we knew we just had to write in a system.exit tied with the button. Easy enough as our framework was already in place. Next getting those tiles to move was going to be tricky. We had a couple of ideas at first like repainting the tiles to transferring all of the data to the new tile. Finally, we decided on a tile wrapper to set the tiles in then just have them move via wrapper. Then this would coincide with a swap function. Essentially it would take the tile selected and the empty tile and switch the two’s location on screen. For the last part of program 2 we had to make it known to the player that they have a tile selected. To do this we decided on having the tile selected have a border appear around the tile. This was done with a function that was called setAsSelected that would draw a border around a tile if it was the selected tile.

In program 3 we had to decode a file that was composed of raw data, the first hurdle was actually figuring out how to decode the file in the first place as it is the first time we have encountered a problem like this. After much trial and error, a few of our group members managed to find a method to get information from the file in a readable format and shared it with the rest of us. From there it became a free for all with each group member coming up with their own way to extract and store the data within the program. We had two main ideas that we went back and forth on while we worked. One was if we should pull all the information and store it all at once in a separate class to be able to be used later.The other idea was if we should pull the information as we are creating tiles and then store them immediately into the tiles. After much deliberation we decided to go with the former. Because our tile class was buried so deep into our program it seemed better just to pull and store all the information at once. This way if we ever needed any information, or needed to change information in the tiles, we could easily pull it from the class or change it, and then run a function to store all the information again. This would help because we wouldn't need to constantly keep re-reading the file for the tile data.

The next problem we had to work on in this program was the reset button. However after the previous endeavor this seemed like nothing and we quickly figured out that we could use our existing moveTile() function to just store the original position of our tiles. We could then use it on every tile to move them back to their original positions.

Compared to the last program, program 4 was a walk in the park. Considering we already set up the framework for everything that needed to be done, all we needed to do was to figure out how to randomly generate angles for the tiles. We quickly figured out that we could just generate a random number 0-3 and then times that number by 90 to get our random rotation angle and then just store it directly in the tile.

Our next task was to figure out how to get the actual tiles to rotate, even though there were already functions in place that rotated the tiles themselves. One of our group members preferred that we just rotate the tile ourselves because the functionality to easily do it already existed. This was instead of relying on some code we had no idea what it was doing in the background or if the precision was off. So we plugged some math equations whose purpose was to calculate the position of the images we were drawing on the tiles relative to the angle the tile was supposed to be currently at. After that we needed to figure out how to rotate the tile whenever it was right clicked. This was the simplest part of this program as all we had to do was to mimic our current function that detects left clicks and have it detect right clicks as well. Then the function would increment the rotation on the selected tile by 90 degrees whenever a right click was detected. The issues with this method was that eventually the variable storing the rotation would overflow if we did nothing about it so rather than use a bunch of if statements to control the limit of the variable, which would be very not elegant, we used a mod function to change our degree value by 360. This meant our variable would never go over 360 degrees, but always return a value 0-270 for a degree. This is what we wanted, thus solving our problem.

Now the major hurdle for this program was to place the tiles in random locations. This was solved by assigning each tile a random tile number, 1-16, and then feeding the tile the information. That way the tiles would effectively look randomized but we wouldn't have to put in much effort on our side to get the work done.

Finally, part of the program was to make an indication appear whenever the player made a move that was invalid. This was kind of an afterthought, since the rest of the program was so daunting that it simply slipped most of our minds until one of our group members pointed out that we had not gotten this part of the program finished. We ended up going through our code as a group to solve this problem and ended up just adjusting the if statements in our click swapper to change the border red whenever the player did anything other than “legal” moves.

The first thing to do in Program 5 was to look into the system we were going to use for having the user enter a file path. While initially the idea was to use a simple text box, it was found that there were already libraries created by java for selecting files. Once the file selection system was chosen, and we messed around with it a bit to get comfortable with it, we then moved onto implementing a save and load function. This ended up being more complicated and larger than initially expected. In order to have the load function first required that we check that the user did not want to save first. There ended up being multiple functions, some for the dialog system, and others for the actual saving and loading. Once that was complete, we needed to find a way to store the modified flag. At first we stored the flag in the “clickSwapper”, the class that controlled the mouse events. This was kinda inelegant since it required the game window to check the flag inside the swapper, which meant the swapper was now performing multiple tasks. This was changed in the last version so that the game window was instead passed to the clickswapper as a reference.

In the final stretch of the program, program 6, we were tasked with adding the last bit of functionality to the program. The first addition is a timer which tracks how long the user takes to complete the maze, which is displayed above our file, save, and quit buttons. The time would also be saved to the each maze game file in the form of a long integer. Implementing this feature was difficult at times as the timer has to be updated every second independent of the clicking update the tiles would respond to. We solved this by creating a “Increment Timer” class which implements the runnable interface which allows us to always have the timer update function running in the background. The timer itself displays in the (hour:minute:second) format and is updated every second and is reset to its initial loaded value if the maze was rest. The time is also saved when the maze is; this is stored as a long integer of seconds and is the second value in the file. That functionality was added in the both the save and loading functions in the game window class.

A game isn’t complete if the user has no way of winning, so our last feature was a function to test if the maze tiles are in the correct position of the game board and the rotation of each is correct. This was simple to implement because the initial layout in each .mze file is that mazes solution and the correct rotation of each file is always set to 0 degrees. To check if the maze is a solution we iterate over the .mze file just like a 2D array by checking the byte value of the tiles position. If the position matches, the program then checks the next four bytes for the number of lines on the tile. If the number of lines match, then the next sixteen bytes are checked which correspond to the float values of the endpoint of each line. Then the check moves to the next tile, and if at any point a false value is returned then the whole loop exits and the maze is not solved. This checking function is run every time a tile is clicked as a convenient way to constantly be checking if the maze has been solved without always running in the background. After this implementation the maze is complete and all we had left was to finalize the code by flushing out some comments, finishing up documentation, and pushing everything up to github.

This program changed vastly over the semester and couldn't have been complete without full effort from everyone. It took all of our knowledge to put this together and from program 1 all the way to 6 we had to plan and then achieve every step of the way. From starting off with just the base code all the way to having it show when the puzzle was solved as well as adding the timer, our group worked hard together.