Mathew Partin

CMPEN 351

Final Project

Writeup: Part 2

Visualization of the Depth First Algorithm for Solving Mazes

In order to complete this project, I broke it up into several phases. The first major phase was reading in the mazes and getting them to display on the bitmap properly. I did this using syscall 14. I stored the result of this syscall into a buffer. Once I had accomplished that, I created a function to calculate the address of the current location within the maze based on an x and y coordinate. This allowed me to get the value in the buffer at the x and y coordinate that I was looking for. At that point, I through the x and y coordinates and checked the value at each pair. If the character stored there was a character used to represent a wall (either a +, -, or a |), then I would draw a black dot. If the value at the coordinate was a blank space, I would draw a white dot. If the value was an S (which I used to represent the start of the maze), then I drew a teal dot, and also stored those coordinates for use later. If the value was an E (used to represent the end of the maze) then I drew a purple dot. At this point the maze was drawn on the bitmap. In order to test this functionality, I took several steps. First, when I read in the file I printed that as a string using syscall 4 to ensure that my file I/O was working correctly. Then, once I was assured that that was functioning correctly, and the display was working, I tested about 10 separate mazes to make sure there were no unexpected bugs when throwing new mazes at it.

The next major phase was to develop the actual visualization of (and functionality) of the depth first algorithm. This algorithm works in a very systematic way. To begin, I pass in the coordinates of the start of the maze. For the entirety of the algorithm I have a priority order of right, bottom, left, top. Once I have the coordinates of the start, I check the adjacent nodes to the start in the order of priority. I stop checking nodes as soon as a blank space is found. When a blank space is found, I call the function again on the node that was just discovered as blank. The algorithm will run again on this new node. As it moves along, each node that it hits will be marked with a C in memory, and displayed as yellow on the display. This will loop until one of two things happens. In the situation that the check finds and E, then it has found the end of the maze, and therefore, it will stop this portion of the program. The other situation the algorithm enters a different state is if it finds a dead end. This means that no blank space has been found adjacent to the current node. In this case, the current node cannot be part of the solution, so it is marked as an F in memory, and displayed as red on the bitmap. Then, the algorithm will search in reverse priority order for a C in memory along the adjacent nodes. When it finds that node, it will call the entire function again on that node. This serves as a manner of backtracking out of dead ends. With all of that put together, the algorithm should be able to find the end of the maze from the beginning of the maze. This portion of the program was tested extensively. There were several logic errors, and once I solved them, I threw all 40 mazes at the algorithm to ensure that there were no errors. One method that I used to simplify the testing for this was to ensure that the algorithm worked in all four directions. There were times when, for example, I could only back track down and left. So once I solved this issue, I ran the algorithm on all 40 mazes to ensure that it was 100% correct.

The next major portion of the project was to implement the final path tracing. This was implemented in a very similar manner to the backtracking portion of the previous function. I passed the coordinates of the start of the maze, and began searching in the same priority order for a C in memory. When I found one I mark the current node as P in memory, draw a green dot, and then call the function on the node I just found. This was tested in the same way as above.

One of the things that I had discussed was maybe doing this project with A\* instead of depth first. I decided to tackle this using depth first simply because A\* would have been far outside the scope of this project. I did a lot of research on A\* before I began development, and I think it would have easily quadrupled my development time. Also, one thing that changed from my homework submission, is that I did significantly less memory manipulation than I originally expected. This is simply because I ended up doing the file I/O slightly differently than I originally thought I was going to, which saved me several steps along the way.