1. Amazing Maze Overview

Amazing Maze by Espadeiro is a program designed to solve complete mazes by directing avatars along a path that will bring them all together. Using the information gathered through turns, a path is determined leading the avatars to a solution. Do to the nature of a complete maze, this path is also guaranteed to be ideal. Amazing Maze also supports functionality allowing a graphical representation of the maze to be constructed as the information becomes available to the local machine.

Amazing Maze will consist of both a startup script, and an Avatar controller program. The design specifications for both are given below.

1. Amazing Maze Startup Design Specifications (AMStartup)

Input:

1. –n nAvatars: the numbers of Avatars in the maze (int).
2. –d Difficulty: (int) the difficulty level, on a scale from 0-9
3. –h Hostname: (char\*) the hostname of the server running the maze.

Output:

Amazing Maze Startup will generate Avatar Client threads for each of the avatars specified upon completion of the program.

Data Flow:

AMStartup will communicate with the server and receive information packets back. It will then process this information and use it to generate avatar client processes.

Data Structures:

Avatar—a struct to hold int and char\* information received by MazePort for use in Avatar Client

Pseudo Code:

* Begin by checking arguments, if acceptable continue, otherwise inform user of mistakes.
* Instruct network component to initiate contact with specified server. Check for success.
* Store the received information about MazePort and Avatars, and for each Avatar, initiate an Avatar Client.
* Exit when inferior process report an exit condition (see the Avatar Client)

1. Amazing Maze Avatar Client Design Specifications

Input:

1. AvatarId (Integer generated in Amazing Maze Startup starting at 0 and incrementing by 1 for each additional avatar).
2. The total number of avatars in the maze.
3. The difficulty of the maze.
4. The IP address of the server
5. The MazePort returned from AM\_INIT\_OK message.
6. Filename of the log the Avatar will write to (potentially incorporating avatar id and difficulty).

Output:

Amazing Client will create the log file specified for it’s avatar and update it with each move made. Amazing Client will also generate global structures after each turn acting as bread crumbs and informing the actions of the other avatars. As each avatar makes its move, it will save the current knowledge it has created about the total graph for future us by it and the other avatars.

Data Flow:

Information will be generated through the server-client interactions in the form of valid and invalid moves. This information will be stored within the shared memory Maze Struct so that avatars will know when they have crossed paths and will also be able to keep track of valid moves to avoid repeated experimentation. Additional information from the server will include the END type messages associated with success or some failure condition.

Data Structures:

Maze – Holds the MazeNode array as well as the target tuple. Has methods for traversal that allow the maze array to be used like a graph.

MazeNode\* maze[X Size][Y size] Holds information about each square in the maze. Stored in shared memory accessable to each Avatar Client and updated after each move. Designed such that the MazeNode also functions like a graph for traversal. Additionally, can be accessed by graphics to draw the maze as information is genereated.

MazeNode a struct holding the “bread crumb” information of a given move. This includes information about walls that has been discovered, preventing repeated experimentation.

Avatar a struct holding info for each of the avatars in the maze.

Stack a data structure following LIFO model

Queue a data structure following FIFO model

Pseudo Code:

* Process Parameters received from AMStartup
* Once ready, send AM\_AVATAR\_READY message via MazePort with AvatarId.
* Process AM\_AVATAR\_TURN message.
* Initialize data structures
  + Avatar’s graph starts of with one node (current location) and with no known edges.
  + Inform global 2 dimensional array of current location. (X,Y as keys).
* If TurnID indicates Avatar is ready to make move, use A\* style heuristic to make initial move.
  + The avatar will calculate the most central square in absolute position in the x,y plane, and make the move that would move it closer positionally. While it is likely this move is not graphically closer, the probability is slightly more favorable that this move will move the avatar closer to the central square than for the other three possible moves. The reason for this stems from the fact that the net displacement must be along this direction in order for any positional change to occur.
  + The idea here is that this will have an incremental benefit over simply using the left hand rule to inform decisions as the worst case should not be worse than left-hand rule in a randomly distributed maze.
* Once move has been decided, send AM\_AVATAR\_MOVE message.
* For each additional move:
  + Process position from last last move. This means updating the nodes of the avatar’s known location graph. This graph will store both legal and illegal moves, saving the avatar from having to hit the wall multiple times.
  + Check the avatar’s graph to determine if any of the other avatars are on this graph. (The nodes will all be initialized in a 2-d array such that when a node is added to an avatars graph, it will be the same structure in all graphs and thus the graphs of 2 avatars will overlap when they cross paths).
    - use hashtable instead to determine if the graphs overlap. Or 2-d array. Then if it overlaps, use DFS to find path by searching graphs. Use the first move of path as move.
    - If 2 or more nodes overlap, we will now need to re-compute the target location for them to arrive at. Just brute force as there will be a maximum of 3 paths leading into a node, so not hard to explore for ideal target square.
  + If the graph current holds more than one avatar, an ideal solution is known. (There is only one path between them, so it must be ideal.)
    - If a solution to bring 2 or nodes together exists, instruct both nodes to execute. They can then continue to look for other nodes together.
  + If no paths have crossed, continue to attempt to move toward the central position in grid until a path has been crossed.
  + If all three directions have been explored for current location, back trace until reaches the last node for which there is an unknown edge that must be checked.
  + Send move to server
  + Log move in log.
  + If an Avatar socket breaks, shutdown.
  + If solution code is sent, shutdown and write to log (once)
  + If max moves made, shutdown.

1. Amazing Maze Design Overview

├── Espadeiro

│   ├── amazing.h

│   ├── controller

│   │   └── placeholder

│   ├── graphics

│   │   └── placeholder

│   ├── library

│   │   ├── queue.c

│   │   ├── queue.h

│   │   ├── stack.c

│   │   ├── stack.h

│   │   └── testscripts

│   │   ├── queuetest.c

│   │   ├── queuetest.log

│   │   ├── stacktest.c

│   │   └── stacktest.log

│   ├── logic

│   ├── network

│   └── README.md

The overall program will be implemented in self-contained modules relying on shared data structures on and source code stored in the library. By disciplined use of consistent interfaces, this will allow for a complete overhaul of any one module (graphics, controller, logic, network) without requiring the restructuring of the entire program.