Amazing Maze Design Documentation

Danielle Midulla Katy Sprout James Jia Pritika Vig

August 28, 2015

The Amazing Maze

You're on a Caribbean Island vacation with some of your friends. It's all sunshine and fruit drinks, when a mysterious fellow walks up to you on the beach. He offers each of you a week's stay at the finest resort on the island if you and your friends can solve the resort's garden maze challenge. After a brief consultation, you and your friends agree. There are some rules, the fellow says, but you all agree anyway and follow him to the resort.

He tells you that you will all be led into the maze while blindfolded. He goes on to say that some time later he will return, and if you and your friends are all together, you will win the challenge. Once you're blindfolded and led into the maze, he instructs you and your friends to listen for a bell to ring three times at which point each of you may remove the blindfold. Each of you are then blindfolded and led into the maze. You all assume you're together, but when the blindfolds are removed, you realize that while you are indeed in the maze, you are no longer with your friends - you can see no one.

The walls of the maze are too high to see over, and they are too thick with thorns to see through or to climb over. You decide that the best course of action is to go through the maze looking for your friends. You quickly realize that you can call out to them and they can hear you and respond with hints as to where they are. Remembering that you and your friends have a limited amount of time to get back together before the man returns, you dash off through the maze to find them.

The Amazing Maze program creates a client connection to the maze server set up by the Dartmouth Computer Science department, hosted at pierce.cs.dartmouth.edu. The server generates a maze at a specified port based on the difficulty level input by the user. The program then creates a user-specified N number of avatar threads to wander the maze. The avatars are created share information through a global variable maze structure. Based on the information in the structure, the avatars use the right hand rule to traverse the maze and find each other.

Design Documentation

0.1 AMStartup.c

Inputs:

- -n nAvatars
  (int) the number of Avatars in the maze
- -d Difficulty
  (int) the difficulty level, on the scale 0 (easy) to 9 (excruciatingly difficult)
- -h Hostname
  (char *) hostname of the server. Our server will be running on pierce.cs.dartmouth.edu

Outputs: Graphical representation of maze created by graphics.c

Data Structures:

- Avatar structure - passed into thread from avatar.c.
  - int avID;
  - int nAvatars;
  - int difficulty;
  - char ip[100];
  - int MazePort;
  - int MazeWidth;
  - int MazeHeight;
  - FILE* pLog;
- Message struture - defined by amazing.h provided.

Data Flow:

AMStartup validates arguments and then creates a socket using pierce.cs.dartmouth.edu. It then sends an initial message to the socket. If it does not receive a positive message back, it will communicate an error to the user and exit. If it receives

a positive message back, it will parse the maze port and maze parameters and opens a log file to begin logging data. Then it creates a struct for each avatar with the log file, number of avatars, maze parameter, ip address of the server, and mazeport. Next it calls a function from avatar.c to create a thread for each avatar struct. The threads are joined, and open exit, the program cleans allocated memory, frees the ports, and exits.

Pseudocode:

1. Validate arguments using getopt
2. Set up server and send the initial message to the server and wait for response.
3. Quit if AM_INIT_OK not received.
4. Parse out the maze port number, width and height from message.
5. Create N avatar structs
6. Create N threads and join them.
7. Upon completion of threads, clean messages sent, close port and log file and close.

0.2 avatar.c

Inputs:

- An avatar structure, as defined above.

Outputs: None - quits on maze solved, error, or max moves.

Data Structures: None.

Data Flow:

First the avatar() function in avatar.c opens a socket for the avatar passed in to talk to the maze port. Then the avatar() function sends an initial message. When all threads have sent a setup message, the server returns the location of all avatars. The avatar() function listens for server responses in a while loop that breaks on maze solved, max moves, or a server error. After receiving the turn message, each thread will check if the turn id indicates their turn. If so, the avatar picks a move using the maze algorithm in maze.c and then makes a move and writes to the log. Additionally, the avatar will look at the previous move and the current position, and add a wall to a global variable maze structure if the last move was invalid. This continues until we hit an exit condition. Upon exit, avatar() cleans message

data and breaks back into AMStartup.c

Pseudocode:

1. Set up avatar on maze port.
2. Send AM_AVATAR_READY signal to the server
3. Enter a while loop that breaks on maze solved, error message, or too many moves.
4. Wait for a message from the server; upon receiving, if it is a turn message and if avatar ID is equal to turn ID, enter the pick a move sequence.
5. First look at the last move, if the last avatar to have a turn is in the same spot as previously, add a wall to the global variable maze structure to indicate an invalid move was made.
6. Then check to see if a final destination has been picked yet (a place where two or more avatars are standing). If not, check to see if the avatar is in the same place as another avatar; if so, save the position as final destination.
7. If the avatar is at final destination, don't move. Else, pick a move using the right hand rule defined in maze.c.
8. Write the move and current board status to the log.
9. Render graphics with functions defined in graphics.c.
10. Upon an exit condition, free messages and the maze variable and exit back to AMStartup.c.

0.3 maze.c

Inputs:

- An avatar structure defined in amazing.h and explained above for which to pick a move.

Outputs: an integer signifying a turning direction.

Data Structures:

- MazeNode
  - int row

  - int column
  - int north_wall
  - int south_wall
  - int east_wall
  - int west_wall

- Maze
  - MazeNode** maze;
  - int num_row
  - int num_col

Data Flow:

Maze.c uses the right hand rule maze solving heuristic in order to pick a move for the avatar specified. First the rightHandRule() checks to see if the move is a known dead end. If it is not, it returns the move. If it is in, then the rightHandRule gets the direction to the right of the "facing direction" of the avatar and checks to see if it is a dead end. If it is not, that direction is returned. If it is a dead end, The avatar checks the direction to its left and whether it is a dead end. Finally, if all three are dead ends, the avatar turns around and backs out and adds a "fake" wall into our maze to indicate dead end.

Pseudocode:

1. Check to see if moving forward is a dead end, if not return it.
2. Check to see if the move to the right is a dead end, if not return it.
3. Check to see if the move to the left is a dead end, if not return it.
4. If all other directions have walls, turn avatar around, back out and create a fake wall at your previous position, to block off pathway and save time.

0.4 graphics.c

The graphics file uses Aske graphics to render pictures of our maze structure. It utilizes the global variable maze defined in maze.h and the the turn messages returned by the server to indicate the position of known walls and the avatars. The functions drawInside() and drawAvatar() are called by the avatar() threads to visualize each move. The functions print an ascii representation of the maze and avatars.

0.5 Error Conditions

The initial program validates arguments. However, the portions meant to be run by our main function (maze.c, avatar.c, etc) do not argument check and assume correct input by the function calling them.

0.6 Test Cases

Testing focused on possible messages returned from the server, and paths through our maze algorithm. To view our unit testing, please see the ./test folder located in our project directory.