Delos Chang
Gedas Bertasius
F. Cameron Setian

Design Specification for AMStartup.c

Input: -n [Number of Avatars (nAvatars)] -d [Maze difficulty (D)] -h[Hostname]

1. Arguments must come in this order
2. Maze difficulty between 0 (easy) and 9 (excruciatingly difficult).
3. The only acceptable hostname is stratton.cs.dartmouth.edu (or kancamagus.cs.dartmouth.edu for the backup server)

Output: Void

1. The program doesn't return anything formally
2. Starts *nAvatar* copies of amazing_client with the following parameters: avatar id, # of avatars (*nAvatar)*, maze difficulty *(D)*, server IP address, Maze Port, name of the log file.
    a. Example: ./amazing_client 0 3 2 129.170.212.235 10829 Amazing_3_2.log
3. Creates a new log file called Amazing_$USER_N_D.log where N-number of avatars, D-maze difficulty
    a. First line starts with: $USER, MazePort and date and time.
    b. Example shell script cmd: echo $USER, 10829, `date` > $filename where filename contains the log file

Data Flow:

1. User inputs *nAvatars* for number of Avatars. *D* for Maze difficulty and *Hostname* for hostname to connect to.
2. AM_INITIALIZE message to the server at the AM_SERVER_PORT with *nAvatars* and *D* for difficulty.
3. Expected response with AM_INITIALIZE_OK. If so, parse the *maze width* and *maze height* and *unique TCP/IP port number* to communicate to.
4. *nAvatar* copies of the main client software to simulate “friends” in the maze. Data parameters will be: AvatarID, *nAvatar*, *Difficulty (D)*, server IP address (parsed above), Maze Port (parsed above), name of log file

Data Structures:

- *Shell script commands.*
- *No formal data structures.*

Pseudo Code:

1. Validates the arguments
2. Retrieves all of the necessary information to send the message to the server (such as server IP address, Avatars, difficulty)

3. Sends AM_INITIALIZE message to the server with nAvatars and Difficulty as parameters
4. Check if the response was AM_INITIALIZE_OK or AM_INITIALIZE_FAILED. For example, server could response AM_INITIALIZE_FAILED if nAvatars is greater than AM_MAX_AVATAR.
5. If the response was AM_INITIALIZE_FAILED, stop processing, print error message, clean up and exit.
6. If server responds AM_INITIALIZE_OK, retrieves information such as MazePort and Maze dimension integers (width and height) from the server message. The MazePort is unique and is a TCP/IP port number that should be used to communicate with the server about the new maze.
7. Creates a new log file in format Amazing_$USER_N_D.log where $USER is current userId, N is the value of nAvatars, and D is the difficulty of the maze.
    a. First line of this file will contain the date and time. Example: echo $USER, 10829, `date` > $filename
8. Starts amazing_client program *nAvatar*-times with the required parameters. These will be the Avatar copies.
9. Terminates

Design Specification for Avatar Program

Input: The following arguments in this order (sent from AMStartup):

1. Avatar ID Number
2. Total # of avatars (*nAvatars* from the AMStartup script)
3. Difficulty (difficulty of the maze)
4. IP address of the server
5. Maze Port (as returned in the AM_INITIALIZE message from AMStartup)
6. Filename of the log that the Avatar should open for writing in *append* mode

Example: "./amazing_client 0 3 2 129.170.212.235 10829 Amazing_3_2.log"

Output:

1. *Appends* (to the log filename specified in the parameter argument) with errors, common outputs and the first line specifying the date, time, port and ID.

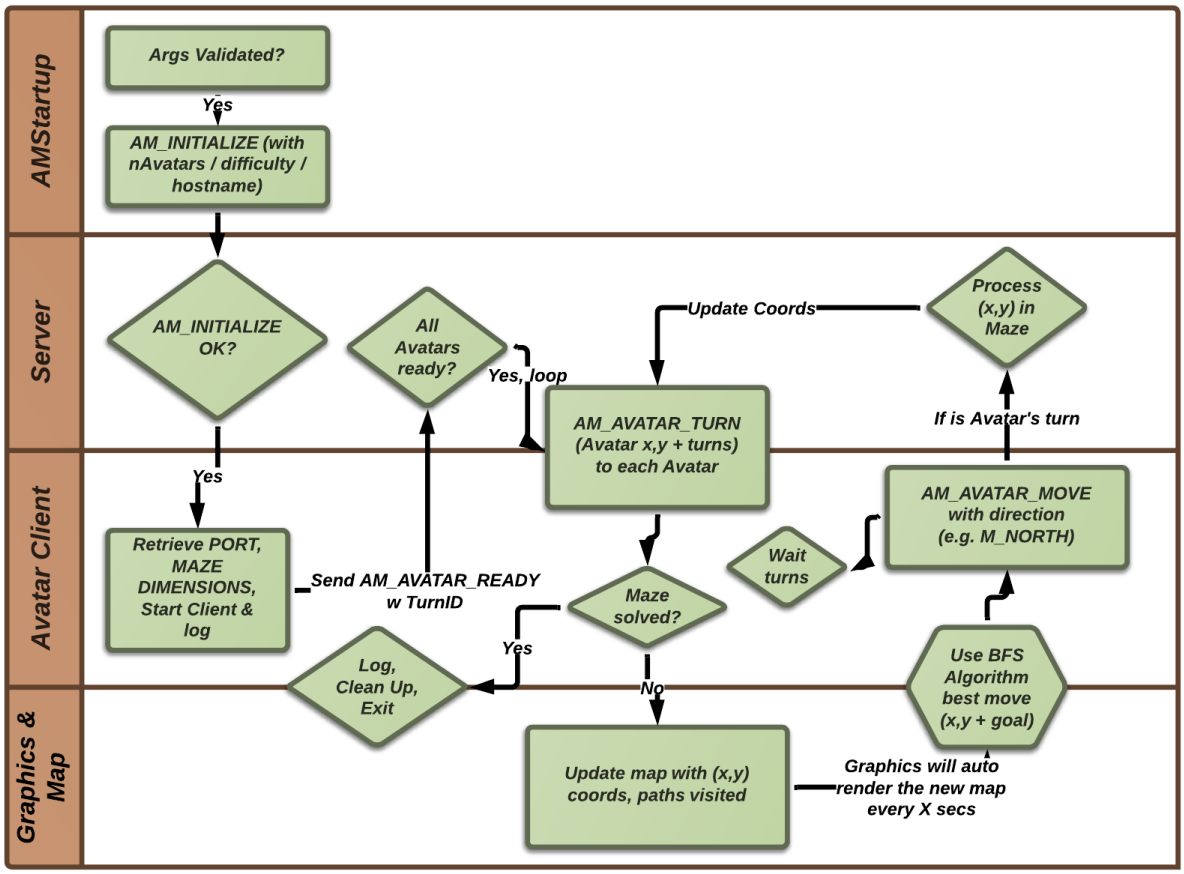
Data Flow:

Files:

- Bfs.c : contains the modified BFS algorithm
    - Input: a matrix struct (representing the map), (x,y) coordinate of the Avatar (XYPos), goal coordinate (XYPos), empty int array (to be filled with direction).
    - Output: Fills the array with a sequence of moves for the Avatar to make.
- Graphics.c : contains functions that will draw the map visually from the 2D array data structure. It will redraw a global 2D char array every UPDATE_INTERVAL (default 1 second)
    - Input: No formal parameters. Takes in global 2D char array for redrawing

    - Output: GTK window that is rerendered every UPDATE_INTERVAL
- Amazing_client.c. : updates the global shared map and calls updates to graphics, processes the turn handling and move making of the Avatars.

Data Flow



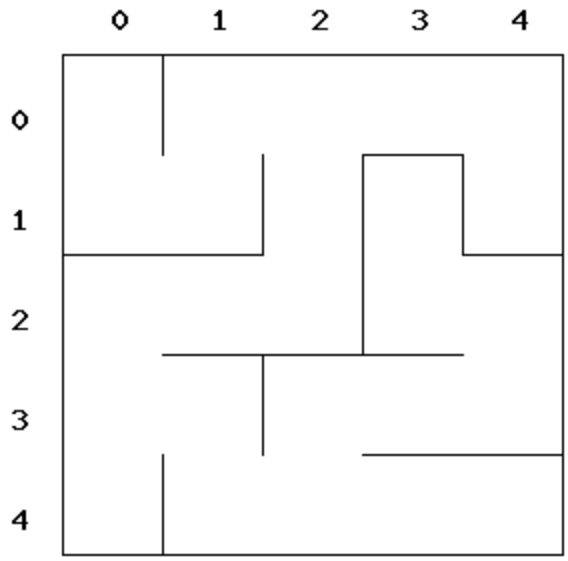
Data flow for Avatar positions:

1. Receive AM_AVATAR_TURN with (x,y) coords for each Avatar and turn numbers.
2. Update map with (x,y) coordinates
3. Use (x,y) coordinates to calculate the shortest distance between two Avatars for the modified BFS algorithm.
4. With the map, the start and goal coordinates, use bfs to calculate an avatar's sequence of move. Send the move message in AM_AVATAR_MOVE.
5. Update the shared map that all the avatars use. Call update_graphics() to convert this shared map to the 2D char array for graphics processing.

Data Structures:

- Shared map: the map of the maze that is used by every Avatar and is stored in shared memory.
- Semaphore: used to regulate access to the shared resources.
- Queue: structures that contains a pointer to a cell that is at the front of the queue.
- Head: the pointer to the first cell in a queue.
- Cell: Data structure that holds its location on the map, the a pointer to the parent node, and pointers so that it can be used in a doubly linked list, like a queue.
- XYPOS: Data structure that contains a pointer to an x position and a pointer to y position on the map.
- Matrix: Data structure that holds a 2-D array (that represents the maze) and the dimensions of the array.

Sample Maze (Level 0) Representation (adapted from Level 0 Maze):



E 1 E 0 E 0 E 0 E
0 Z 0 Z 0 Z _ Z 0
E 0 E 1 E 1 E 1 E
_ Z _ Z 0 Z 0 Z _
E 0 E 0 E 1 E 0 E
0 Z _ Z _ Z _ Z 0
E 0 E 1 E 0 E 0 E
0 Z 0 Z 0 Z _ Z _

E 1 **E 0 E 0 E 0 E**

Open Paths marked with Red

Legend:

- E - empty cell that avatar can move through
- 0 - empty cell that avatar can move through
- Z - null space that does not represent anything
- 1 - vertical wall
- _ - horizontal wall

- In this representation, every second row and column depicts maze entries while the rows and columns in between represent the wall structures.
- For example, coordinate (1,3) in the maze would be represented as E at (2,6) in our 2D array representation (conversion formula from original maze to our representation is (2n,2n)).
- Entries Z in our representation, depict wildcards (entries that don't matter) since they will never be reached.
- Movements in the maze are only possible from one E letter to another E letter. If there's a 1 between two 'adjacent' E letter that means there's a wall between them and the movement is not possible.
- 0 between two 'adjacent' E letters means that there's no wall and the avatar is allowed to move.

Pseudo Code:

The following pseudo-code is used by each Avatar copy generated by AMStartup script:

1. Send an AM_AVATAR_READY message to server with its avatar ID as a parameter.
    a. If the avatar ID is valid, expect an identical server response: AM_AVATAR_TURN message *after all the Avatars are ready*. This message will contain the Avatar's (x,y) position, other Avatars' (x,y) positions and its turn number.
    b. If the response is AM_NO_SUCH_AvatarID, then print an error message and stop processing for this Avatar. But continue processing for other Avatars.
2. Parse the AM_AVATAR_TURN message to retrieve the current Avatar's (x,y) coordinates and other Avatars' (x,y) coordinates.
3. Through functions in Amazing_client.c, update the shared map with the returned (x,y) Avatar coordinates from the AM_AVATAR_TURN message.
4. Initialize graphics the graphics that display the map by calling render_maze.
5. While the Avatars do not receive an AM_MAZE_SOLVED message *OR* do not receive AM_SERVER_TIMEOUT *OR* socket connection is not broken *OR* do not receive AM_TOO_MANY_MOVES, continue:
    a. Calculate the shortest distance between any 2 Avatars. This will serve as the end

goal coordinate for the modified BFS.

b. If it is the avatar's turn, call maze-solving algorithm from bfs.c to get the avatar's move (the next direction for the Avatar to move). The algorithm will be the following:
    i. (See BFS algorithm pseudocode below)
    ii. Input: The global map, Start coordinate, Goal coordinate, an empty int array.
    iii. Calculate the shortest path between the start and goal coordinates. Unknown parts of the map are assumed to be free.
    iv. Fill the int array with the sequence of moves that move the Avatar to the goal..
c. Send an AM_AVATAR_MOVE message to the server with the direction from the path array filled by bfs, specifying the direction for the avatar to move.
d. If server responds with AM_AVATAR_OUT_OF_TURN message *or* AM_UNKNOWN_MSG_TYPE, wait and then send AM_AVATAR_MOVE message again.
e. Decrement the semaphore's value to 0 so the map cannot be accessed by other avatars.
f. Expect to receive an AM_AVATAR_MOVE message with updated (or not updated if move runs into wall) (x,y) coordinates.
g. Call functions to update the shared map.
h. After the move is calculated, the semaphore is incremented by 1.
i. The avatar program will wait until the semaphore is ready (equal to 1) to begin a turn.

6. If the avatar receives AM_MAZE_SOLVED, write the AM_MAZE_SOLVED message to the file *once*.
    a. Document that maze is solved.
    b. Clean up and exit
7. Else if the avatar receives AM_SERVER_TIMEOUT
    a. Document that the server has timed out
    b. Clean up and exit
8. Else if the avatar receives AM_TOO_MANY_MOVES
    a. Document that the avatar has made too many moves.
    b. Clean up and exit.
9. Else if the socket connection is broken
    a. Document that the socket connection is broken
    b. Clean up and exit.

Sample BFS Algorithm Code in bfs.c:

Input: Global Map, Start coordinates, Goal coordinates, Empty int array.
Output: Fills the int array will a sequence of moves for an Avatar to make.

Pseudo Code:

1. Initialize the container queue (variable container) to hold the already visited moves. The main queue (variable q) will hold the children to be checked.
2. While the queue head is not empty:
    a. Dequeue a cell from the queue
    b. If the dequeued cell has the same coordinates as the goal cell, BFS is complete:
        i. Trace the parents of the dequeued cell back to the start coordinates. This will construct a path from the start coordinates to the goal coordinates.
    c. If not, make an array with the each of possible directions the Avatars can move in. Randomize the order of this array each time.
        i. We have found that this, on average, leads to better results that solely relying on X direction every time.
    d. For each direction in the Array of possible directions
        i. Check if the adjacent cell is within the boundaries of the maze
            1. Check if the adjacent cell is not blocked by a wall *AND* the container queue doesn't already have the adjacent cell (as it would be then already visited)
                a. If these checks are passed, mark the current cell as the parent of the adjacent cell. Doing this enables the algorithm to be able to trace backwards from goal to start coordinates
                b. Enqueue the adjacent cell to the queue (as a child).
                c. Enqueue the current cell onto the container (already visited)
3. If the path hasn't been found
    a. Print error message and set the path accordingly.

Sample BFS Algorithm Code in graphics.c:
Input: No formal input (just shared global 2D char array)
Output: GTK window

Pseudo Code:

1. Once render_maze is called *once* from a separate thread in Amazing Client, create the GTK window.
2. Register the callback for cb_expose to draw the graphics.
3. Pull the 2D char array and render. Set up a timer to fire every X seconds (defined in UPDATE_INTERVAL).
4. Every time the timer fires, call cb_expose again (to rerender the graphics)