AMStartup (in C)

1. Input
    a. Number of avatars
    b. Difficulty
    c. Hostname
2. Output
    a. Log file
    b. N processes of the Avatar Program
3. Data Flow
    - Sent:
        a. AM_INIT: is a message sent to the server telling it to set up the maze
    - Received:
        a. AM_INIT_OK: initialization succeed
        b. AM_INIT_FAILED: init failed
    - Received Errors:
        a. AM_INIT_TOO_MANY_AVATARS: too many avatars
        b. AM_INIT_BAD_DIFFICULTY: invalid difficulty
        c. AM_UNKNOWN_MSG_TYPE: unknown message
        d. AM_UNEXPECTED_MSG_TYPE: out of order message
        e. AM_SERVER_TIMEOUT: exceeded time between messages
        f. AM_SERVER_DISK_QUOTA: server exceeded disk quota
        g. AM_SERVER_OUT_OF_MEM: server failed to allocate memory
    - To current directory:
        a. The log file is appended to contain all the necessary data that (num avatars, difficulty, time, etc.)

4. Data Structures
    a. AM_Message: a data structure that is a union of the many possible structures so it can hold the data for any of the possible AM messages.

AMStartup Pseudocode:

1. Reads # avatars, difficulty, hostname from command line
2. Validate the input
    a. Ensure correct number of arguments
    b. # avatars and difficulty are whole positive integers
    c. 2 <= Number of avatars <= AM_MAX_AVATAR (in amazing.h)
    d. Difficulty <= AM_MAX_DIFFICULTY (in amazing.h)
3. Check if a directory for logs exists
4. Create the AM_INIT message (specify # avatar, difficulty)
    a. Make new AM_Message
    b. Give it type AM_init

    c. Access the init struct
        i. message.init.nAvatars = # avatars
        ii. message.init.Difficulty = difficulty
5. Client sends the AM_INIT message to AM_SERVER_PORT identified by amazing.h
    a. Create socket - socket()
    b. Connect to address of server - connect()
    c. Send init message - send()
6. recv() the server response and use ntohl() to change message to host byte order
    a. If AM_INIT_OK
        i. Store MazeWidth, MazeHeight, MazePort
        ii. continue
    b. AM_INIT_ERROR() - print useful statement specifying the error
        i. If AM_INIT_FAILED
            1. Read the Errnum
        ii. AM_SERVER_DISK_QUOTA
        iii. AM_SERVER_OUT_OF_MEM
        iv. AM_UNKNOWN_MSG_TYPE
        v. AM_SERVER_TIMEOUT
        vi. AM_UNEXPECTED_MSG_TYPE
        vii. AM_INIT_ERROR_MASK in amazing.h
            1. AM_INIT_TOO_MANY_AVATARS
            2. AM_INIT_BAD_DIFFICULTY
    c. close() socket connection and exit program if errored out
7. Close socket
8. Create logfile name “Amazing_$USER_numavatar_difficulty.log” and open for writing (“w”)
    a. $USER = USER (getenv("USER"))
    b. N = number of avatars
    c. D = difficulty
    d. First line should have $USER, Mazeport, date and time
        i. time() and localtime() in time.h
8. Loop thru all given avatar ID’s and start individual process for each - fork() and execlp()
    a. Give execlp() the name of client program “./AmazingClient” and all arguments client program needs (avatar ID, number of total avatars, IP address, Mazeport, name of logfile, maze width, maze height) - no need to pass in difficulty level

Avatar Program

1. Input (got permission from CCP to include Mazewidth and Mazeheight)
    a. AvatarID
    b. Number of Avatars
    c. IP Address of Server
    d. MazePort (from AM_INIT_OK)

    e. Log file name
    f. Mazewidth
    g. Mazeheight
2. Output
    a. Edited Log file (includes all moves avatars requested, whether the maze was solved or not, and any errors)
    b. Stdout - each process prints out whether connected to server, client program arguments, each move the avatar attempts to make, and whether the maze was solved
3. Data Flow
    - Sent to Server:
        a. AM_AVATAR_MOVE: the move the avatar wishes to make
        b. AM_AVATAR_READY: tell server that this avatar is ready
    - Received Regularly from Server:
        a. AM_AVATAR_TURN: telling which avatar's turn it is to move
    - Fatal errors from Server:
        a. AM_NO_SUCH_AVATAR: invalid avatar
        b. AM_TOO_MANY_MOVES: exceeded max number of moves
        b. AM_MAZE_SOLVED: Maze is completed
        c. AM_SERVER_TIMEOUT: exceeded time between messages
        d. AM_SERVER_DISK_QUOTA: server exceeded disk quota
        e. AM_SERVER_OUT_OF_MEM: server failed to allocate memory
        f. AM_UNKNOWN_MSG_TYPE: unknown message
        g. AM_UNEXPECTED_MSG_TYPE: out of order message
        h. AM_AVATAR_OUT_OF_TURN: Avatar tried to move out of turn
    - To screen:
        a. If GRAPHICS not defined: the positions of any avatars and if any of them have reached the center.
        b. If GRAPHICS is defined: a picture of the maze is displayed to screen that is updated as the maze is solved.
    - To current directory:
        b. The log file is appended to contain 1) every move that every avatar attempts and 2) whether the run ended in maze_solved or in an error.
4. Data Structures
    a. XYPos: has x and y values storing the (x,y) position of the avatar (each avatar can remember an XYPos array of the last 4 moves they attempted)
        i. We allowed each avatar to remember the last four moves according to CCP's instructions, but each avatar only actually needs to remember and use the last two
    b. AM_Message: a data structure that is a union of the many possible data structures so it can hold the data for any of the possible AM messages.
    c. No need to use Avatar struct defined within amazing.h

d. Maze structure: contains the positions of where each avatar is (uses a bool @ and a 2D array of squares for the maze).

Pseudocode

1. Check if sufficient number of inputs from AMStartup and check if a directory for logs exists
2. Connect to mazeport, open socket
3. Send server AM_AVATAR_READY message which includes the AvatarId using value returned by htonl()
4. After the server has received has AVATAR_READY messages from all avatars, the server responds to each of the avatars with an identical AM_AVATAR_TURN message.
5. Create an array of compass directions, going clockwise, starting with M_WEST
6. While the avatar is still receiving messages:
    a. IS_AM_ERROR() - if any of the errors mentioned above (in section titled “Avatar Program”) are received, the program will terminate with an error message written to screen and logfile. Avatar 0 will write to the logfile and close the socket.
    b. If it doesn’t match then it does nothing
    c. Each Avatar will check if the AM_AVATAR_TURN corresponds to its AvatarId (check if it’s this avatar’s turn):
        i. If it is the avatar’s turn, the Avatar will prepare to send back an AM_AVATAR_MOVE message to the server that contain its AvatarId and the desired move direction.
        ii. If the avatar has reached the center, it will stay still
        iii. Else (the avatar has not reached the center):
            1. If avatar hasn’t changed position, it has hit a wall
                a. It must turn clockwise (increment index into compass array)
            2. Else, avatar has changed position:
                a. If avatar has just entered the maze, it will just move default left (West)
                b. Else, check to the avatar’s left (decrement the index into the compass array)
        iv. The avatar will print its ID, position, and direction of the requested move in logfile and in stdout, and send the AM_AVATAR_MOVE message with the appropriate direction
    d. If receive(AM_MAZE_SOLVED):
        i. Avatar 0 writes success to the log, close socket, and exit the program
        ii. Other avatars will just print a success statement to the stdout and exit the program

- NOTE1: When the game has been terminated, it may appear as if the program has not completely returned back to the command line, even though it has. Type in a command line command (ex. ls) to validate that the program has been terminated.
- NOTE2: If the server sends any error messages to the clients, the program will terminate with an appropriate error message, log error in logfile, close files, close the socket

Graphics Implementation

NOTE: requires #define GRAPHICS

1) Get the log file from the server (/var/tmp/MazePort)
    - REQUIRES that the user be on flume.cs.dartmouth.edu (
    - This means that our maze will show ALL walls, even ones that the avatars haven't discovered yet
2) Parsing the log file:
    - Make an array of characters (size = 2*maze.size + 1)
    - Initialize it to empty
    - MazeCell [][]: walls: WNSE borders:
    - Copy all of the walls into character array (-1 or +1) depending on whether they are N/S; E/W walls
3) Save all the positions of the avatars in a maze struct
    - Initially this structure was going to hold all of the walls that we hit and all the breadcrumbs we dropped.
    - This was changed because I realized we could just parse the log file and get all the walls at the beginning.
    - Breadcrumbs were never implemented because our algorithm didn't require them.
    - A smarter way would be to just pass the array of positions to the maze_print func. But I had already written the code for it, so I didn't want to change it.
4) Give the maze struct to the maze print func.
    - The maze print changes the values of its characters in those spaces from EMPTY (~) to AVATAR (@).
    - The maze print function prints the image to the screen.
6) If we receive maze_solve or maze_error then delete the log file we grabbed from the server.