Nick Schwartz, Caleb Kim, Will McConnell
“Amazing Maze” Documentation
AMStartup and Avatar Design Specifications

AMStartup Design Spec

Introduction

AMStartup.c is a C program that initializes the connection with the server. It sends the initial information to the server, gets a confirmatory response message from the server, and then starts up the avatars.

1) Input

AMStartup.c contains the **main** function of the Amazing Maze program. The **Makefile** compiles this **main** function into an executable called **maze_runner**. This executable is run from the command line (from the **project** directory) in the following manner:

./maze_runner [NUMBER OF AVATARS] [DIFFICULTY] [SERVER NAME]

./maze_runner 4 3 pierce

Failure to provide exactly three inputs to **maze_runner** will result in an error.

[NUMBER OF AVATARS] – 4
This parameter specifies how many avatars the user wants the server to put into the maze.
Requirement: This parameter must be an integer between 2 and 10, inclusive.
Usage: The Amazing Maze program needs to inform the user if the number of avatars is not valid.

[DIFFICULTY] – 3
This parameter specifies how challenging the user wants the server to make the maze. Levels 0 – 4 are predetermined, while levels 5 – 9 are randomly generated.
Requirement: This parameter must be an integer between 0 and 9, inclusive.
Usage: The Amazing Maze program needs to inform the user if the difficulty level is not valid.

[SERVER NAME] – pierce
This parameter specifies the name of the server on which the server side of Amazing Maze is being run. Note that this will be pierce.cs.dartmouth.edu.
Requirement: This parameter must be a valid string representing the server name of the machine hosting the server side of Amazing Maze is running. This will always be pierce.cs.dartmouth.edu
Usage: The Amazing Maze program needs to inform the user if the server cannot be connected to.

2) Output

The Amazing Maze program will output a log file with information about the run, and **AMStartup.c** starts this log. The name of the log file is as follows:

Amazing_$USER_N_D.log

Where $USER is the current user's username, N is the number of avatars, and D is the difficulty. While the avatars write most of the information to the log, **AMStartup.c** starts this log by writing the first line in the following format:

$USER, 10829, `date`

Once again, $USER is the username, 10829 is the MazePort, and `date` is the date on which the log was created. **AMStartup.c** also outputs some information about the generate maze and the Maze data structure to the log.

AMStartup.c can also have output in the form of error statements.

3) Data Flow

AMStartup.c initializes the connection with the server. It first sends the AM_INIT message to the server, which contains information about the difficulty and the number of avatars. It then gets a confirmatory response message from the server, which is an AM_INIT_OK message. This message contains the MazePort, the MazeWidth, and the MazeHeight. With this information from the AM_INIT_OK message, **AMStartup.c** builds a Maze data structure, to be used by the avatars to solve the maze. **AMStartup.c** then starts up the avatar threads. It passes the following parameters to each thread:

uint32_t ID = ID of the newly created avatar
int Difficulty = Difficultly of the maze for this run
char *address = IP address of the server
uint32_t MazePort = Port number for the server connection
char *file = Name of the Log file
int nAvatars = Number of Avatars
int width = Width of the maze
int height = Height of the maze
BlockNode*** maze = Pointer to the Maze data structure
XYPos** graphics_spots = Array of avatar locations for the graphics

Finally, **AMStartup.c** waits until all the threads have finished running to exit the program.

4) Data Structures

AMStartup.c builds the Maze data structure, and sends a pointer to it to each

avatar. This data structure is a two dimensional array that represents the entire maze. For each location in the maze, and therefore each location in the array, an avatar has four possible directions in which they can move. Each spot in the array has information about the possible directions of motion from that spot, i.e. North, South, East, and West. This Maze structure will be used by the avatars to keep track of where the walls are, and the details of this are explained further in the Avatar Design Spec.

5) Pseudocode

```
// Check the user's parameters
        // Is the number of avatars a valid number between 2 and 10?
        // Is the difficulty a valid number between 0 and 9?
        // Can the host server be connected to?

// Create a socket connection to the host
        // Exit if this failed

// Send the AM_INIT_MESSAGE

// Receive the AM_INIT_OK message
        // Exit if there is a problem with this messages

// Extract the maze information from the AM_INIT_OK message

// Create the maze data structure

// Check if a valid “results” directory exists
        // If it does, create the log file
        // If not, exit

// Start the avatars

// As long as the avatars are still running
        // Sleep

// Exit
```

Avatar Design Spec

Introduction

The avatars of the Amazing Maze are run by a function in the file **maze.c**. They are started by **AMStartup.c**, and they run until the maze is solved, there is an error, or they have taken too many turns. Each avatar in the maze represents a different thread of the program.

1) Input

avatar is the name of the function in **maze.c** that runs the avatars. It technically only takes one parameter, a pointer. This pointer, however, points to a struct that acts as a whole bundle of parameters. All the variables that the avatars need are packaged into this struct.

The function header for the **avatar** function is as follows:

void* avatar(void* input)

The pointer **input** is a pointer to the bundle of parameters. The bundle contains the following information (already briefly discussed in the **AMStartup** design spec):

uint32_t ID = ID of the newly created avatar
int Difficulty = Difficultly of the maze for this run
char *address = IP address of the server
uint32_t MazePort = Port number for the server connection
char *file = Name of the Log file
int nAvatars = Number of Avatars
int width = Width of the maze
int height = Height of the maze
BlockNode*** maze = Pointer to the Maze data structure
XYPos** graphics_spots = Array of avatar locations for the graphics

2) Output

Each of the avatars writes out a block of initial information to the log as soon as they have extracted their initial information from the input bundle. They each write out the parameters they have received.

The avatars also constantly write to the output log file with information about their current locations in the maze. At the beginning of each turn, each avatar writes a chunk of text to the log file in the following format:

[avatar #`my_ID`] Turn ID: `current turn ID`
[avatar #`my_ID`] Starting coordinates this turn: x=`my_x` y=`my_y`

Each avatar, at the beginning of each turn, writes out the current turn ID (whose turn it is) and their location at the start of this turn. The avatars will continue to write out this information to the log until the maze is over. Once the maze is over, the avatars will write to the log why the maze ended (i.e. victory, too many turns, some type of error).

Throughout the solving of the maze, there are ASCII graphics that the user can watch. This ASCII animation shows each avatar (represented simply by their ID number) running around a rectangular maze. If two or more avatars occupy the same spot in the maze, then the graphics will display that location in the maze as

an 'X'. The graphics also display how many turns have been taken by the avatars, and what phase of the solution the avatars are in (more explanation on this in section 3, Data Flow).

The avatars can also have output in the form of error statements.

3) Data Flow

The avatars receive their starting information from **AMStartup.c**. After extracting this information and writing initial message to the log, each avatar thread sends an AM_AVATAR_READY message to the server, which contains each avatar's ID. Each avatar then waits to receive an AM_AVATAR_TURN message. This message will give each avatar its initial location, and will tell each of the avatars whose turn it is (should always be avatar zero's turn to start initially). Once the avatars have their initial location, they enter a loop that does not get broken until the maze is over. This can happen for one of three reasons: the maze is solved, too many turns were taken, or there was an error of some sort. In each iteration of the maze, each avatar checks if it is their turn. If it is, the avatar makes a decision about how to move, and sends that decision to the server (specifically *how* this decision is made will be explained in a bit). All avatars then receive another AM_AVATAR_TURN message and get the new turn and position information from this message. The loop then repeats, with a new avatar sending off a move to the server.

The avatars moves are determined in two different ways, depending on what *phase* of the solution the avatars are in. The first phase is a searching / mapping phase, where the avatars systematically explore the maze and store all information about walls they hit into the Maze structure that all avatars have access to, so that other avatars don't unnecessarily bump the same wall later. This systematic exploration is done using a principle called the *right hand rule*. This rule consists of making movement decisions in the same way that a human would make decisions if his or her *right* hand was against a wall the whole time. The right hand rule leads to a systematic mapping of the maze.

Once there is a spot in the maze that every single avatar has stepped on (this information is kept in the Maze data structure), the avatars change strategy. When this condition is true, it means that the avatars now have *explored* paths that they can follow to meet up with each other; there is enough information in the Maze data structure, which holds information about where the walls are, to allow each avatar to map his way to a meeting place that everybody can reach. Each avatar proceeds to use a BFS algorithm to calculate how they can most quickly get to the meeting point (which is the location of one of the avatars at the time of strategy changing, chosen arbitrarily).

Each avatar then makes its way along the path to the meeting place. Once it has gotten there, it simply waits for the other avatars to arrive.

4) Data Structures

The Maze data structure is the structure used by the avatars to “map” the maze i.e. keep track of walls and free passageways. The Maze data structure also keeps track of which avatars have touch which locations. It is a two dimensional array of node structs, each of which contains a variable for North, South East and West. These variables are ints, and say whether or not an avatar can move in that direction (0 means there's a wall in that direction, 1 means it's open, -1 means it is currently unmapped).

Another data structure used is a linked list. This data structure is used to implement a queue for the BFS algorithm, which of course needs a queue.

5) Pseudocode

Each avatar does the following:

```
// Extract information from parameter bundle

// Print out initial information to the log

// Send AM_AVATAR_READY to server

// Receive the AM_AVATAR_TURN message
        // If any errors, exit

// Extract information about location and turn ID from the TURN message

// As long as the game hasn't ended…
        // Write out avatar's coordinates and the current turn to the log

        // Display ASCII graphics animation

        // If it's my turn
                // Make my decision based on what phase we are currently in
                // Send the move to the server

        // Receive the next AM_AVATAR_TURN message from the server

        // Is it a solved message?
                // Write appropriate message to the log
                // Exit
        // Is it a too many turns message?
                // Write appropriate message to the log
                // Exit
        // Is it some sort of error?
                // Write appropriate message to the log
                // Exit
```

```
        // Get new information from the TURN message

        // Update the Maze data structure depending on whether we hit a wall or
           moved freely

// Game has ended for one reason or another
// Kill the thread
```