The Amazing Project

Daniel Y Lee, John Ling, Pratap Luitel

COSC 50, Dartmouth College

6/1/2015

Design Documentation

Contents

1) Introduction

The goal of the project is to design, implement and test a client side application in which multiple characters(avatars) traverse a maze and try to converge to a common location. The maze to be traversed is randomly generated by a server. The avatars are initialized to random locations within the maze and the maze is solved if all the traversing avatars end up in the same location. Since the maze has no inaccessible sections, no circular paths and no open areas, it is always possible to solve the maze.

The input and output of the program are as follows.

Input:

./AMStartup -n *nAvatars* -d *difficulty* -h *server_location*

where,

nAvatars valid range: [2 - 10]

difficulty valid range: [0 - 9]

server_location: stowe.cs.dartmouth.edu or tahoe.cs.dartmouth.edu

Output:

A GUI simulation of the traversal of the maze by the avatars

2) General System Description

The system consists of a server which generates a rectangular maze with a difficulty level given by the user. Although the server generates the maze at once, the exact interiors of the maze are unveiled one at a time as a response to avatar's move request. For a session, the server uses the same port for TCP/IP communication with the avatars. The server's functionality to generate the maze and communicate with the client was already implemented and provided prior to the development of the client side of the communication for this project. The scripts listed below model the client side application and was implemented by our team.

AMStartup.c : A client side startup script which initiates and maintains communication with the server.

Avatar.c: The main client script which models the avatars.

3) Data Structure

AvatarStruct

Each avatar is represented by an AvatarStruct which has the following data fields.

Data type	Variable name	Purpose
int	id	id of the avatar
int	total_num_avatars	total avatars in the maze
int	difficulty	difficulty level of the maze
struct	in_addr	socket address
int	maze_port	port number on the server for TCP/IP communication
char*	logfile_name	name of the log file
FILE*	file	log file to record progress
int	moved	to track the very first move
int	dir	direction currently facing

MazeTile

We represent the maze by creating a 2d array of MazeTile struct. Each tile in the maze is then represented by a single MazeTile struct. Each struct has the following data fields.

Data type	Variable name	Purpose
int	northBlocked	true if immediate north is an invalid location to move to
int	eastBlocked	true if immediate east is an invalid location to move to
int	southBlocked	true if immediate south is an invalid location to move to
int	westBlocked	true if immediate west is an invalid location to move to
int [nAvatar]	hasVisited	hasVisited [k] = 1 if the tile has been visited by the kth avatar
int	mazeWidth	width of the maze
int	mazeHeight	height of the maze
int	occupied	if the maze is occupied by any avatar
int	gNorth	drawing information for GUI
int	gEast	drawing information for GUI
int	gSouth	drawing information for GUI
int	gWest	drawing information for GUI

4) Data Flow Overview

A. The client startup script, AMStartup, is invoked with appropriate arguments: *nAvatar*, *difficulty*, *serverLocation*

B. The server in *serverLocation* generates a random maze of difficulty level *difficulty* and initializes the position for each of *nAvatar* avatars. The server also returns a message specifying the *MazePort* (the TCP/IP port number), *MazeWidth* and *MazeHeight* (width and height of the maze respectively).

C. AMStartup upon receiving the message from B does 3 things specifically.
   a. initialize a MazeStruct *learn_maze [*MazeWidth*] [*MazeHeight*].
   b. create *nAvatar* threads and initialize *nAvatar* AvatarStruct, and link each AvatarStruct to a thread.
   c. create a thread for graphics and link it to the appropriate method for drawing (void* mazeDrawer( )).

D. Each avatar communicates with the server on *MazePort.* The server sends back (x,y) position for each avatar and a *TurnId* that dictates which avatar moves first.

E. Each avatar then sends the direction it wishes to move to.

F. The server determines the feasibility of requested direction to move for each avatar. If it is a valid move, the server updates the avatar's position (x,y) to the newly moved place. If not, it will send the old position to the avatar.

G. The process continues unless,
   a. All the avatars are at the same location (x,y) as determined by the server. The server sends a AM_MAZE_SOLVED message to all the avatars.
   b. A socket connection is broken between an avatar and the server
   c. A maximum number of moves has been made
   d. Server waits for more than the specified threshold of AM_WAIT_TIME

H. When the process terminates because of one of the above mentioned reasons, the server frees up related data structures and closes the *MazePort.*

5) Design Choices

There are few key design choices undertaken for the project. They are described in reference to the data flow from section 4.

4)C)

We chose to implement *nAvatar* threads instead of *nAvatar* separate processes. Use of threads allowed for easy sharing of the global maze information (*learn_maze [MazeWidth] [MazeHeight]) amongst the avatars.

For c), we used Gtk+ and the cairo library to implement the GUI. Since drawing the maze and the avatars is a repetitive process, we use a 'timer' function g_timeout_add( ) from the Gtk+ libary to draw the maze and avatars every 50ms.

4)F)

We have implemented a left-hand-rule as avatars preference of direction. An avatar will request moving to the left of where it is currently facing. If the feasibility of the direction is not established yet, the avatar will still make the request to move. The interior of the maze is learned through this process of avatar asking the server and the server replying.

When an avatar with id *avatarID* visits a maze tile, the tile keeps track of the visit by setting hasVisited[*avatarID*] = 1.To avoid the problem of dynamic memory allocation, we pre-allocate hasVisited to have size 10 * sizeof(int) where 10 is the maximum number of allowed avatars. If an avatar has already visited a maze tile, it implies that the path will not lead to the convergent location. So the avatar returns back to the tile where it came from. In doing so, it also marks the direction leading to the visited tile 'blocked'. This helps other avatars avoid the same path. Such a design allows avatars to 'learn' information about the maze through each other.

4)G)a)

We have arbitrarily chosen (5, 5) as the convergent location for the avatars. If an avatar gets to (5, 5) it will stop moving and wait for other avatars to get to the location. When all avatars get to (5, 5) the maze is solved.

6) Test Cases

As part of defensive programming, the following different input cases have been tested.

- Invalid Argument Input cases,

  Lesser argument, argc = 3 (valid argc = 4)

  More argument, argc = 5

  Invalid nAvatar, n = -20 (valid nAvatar range [2-10])

  Invalid nAvatar, n = 20

  Invalid nAvatar, n = ab

  Invalid difficulty, d = -50 (valid difficulty range [1-9])

  Invalid difficulty, d = 50

  Invalid difficulty, d= abc

- Valid Argument Input cases,

  nAvatar = 2, difficulty = 0, server = stowe.cs.dartmouth.edu (minimum possible values)

  nAvatar = 3, difficulty = 3, server = stowe.cs.dartmouth.edu

  nAvatar = 6, difficulty = 6, server = stowe.cs.dartmouth.edu

  nAvatar = 10, difficulty = 9, server = stowe.cs.dartmouth.edu (maximum possible values)

The result of running MazeBats.sh is saved in a log file named *Testlog.datestamp* where datestamp is the actual date and time stamp of running the MazeBats.sh file.

7) Error Cases

There are no known error cases and memory leaks.