Mazewar

1 Mazewar Overview and Objectives

Mazewar is a distributed, multiplayer game that allows each player to control a rat in a maze.\(^1\) A player receives points for tagging other rats with a projectile (“Killing other rats with a missile”, for the more bloodthirsty among you), and loses points for being tagged. Our version of Mazewar is based on the X window system game of Mazewar, which is in turn based on the “classic” game of Mazewar that ran on Xerox’ Alto’s in the 1970s. (Some claim that an even earlier version of this maze game ran on the IMLAC displays at MIT, before that.) Our project is also based on Amaze, a multi-player game with a distributed implementation on top of the V kernel. Amaze is referred to in the paper on “dead reckoning” in the course reading list and there is a detailed paper about Amaze by Berglund and Cheriton, if you’re really interested).

The maze is presented graphically in an X window, as illustrated in Figure 1. There are three parts to the window:

1. Rat’s eye view: The upper section is a perspective view of a rat’s view forward.
2. Bird’s eye view: The toplevel view of the maze showing the position and orientation of a rat from above,
3. Nerd’s eye view: The bottom area is reserved for text to be used to report the scores and status of other players.

If you’re just interested in the numbers, and not the pretty pictures . . .

1.1 The Rules

The rules of the game are as follows.

- Each instantiation of the program is a separate player.
- Each player controls a single rat in the maze.
- The maze consists of a 32 \(\times\) 16 array of cells. Each cell is either occupied by a wall, a rat, or is empty.
- A player sees the toplevel view of the maze, but only their own rat’s position and orientation in the bird’s eye view (see figure).
- A player sees any “visible” (not blocked by intervening walls) opposing rats in the perspective view (see figure).
- There are commands which allow a rat to peek left and right around corners without exposing itself. (Perspective view only). If this is not clear, we will demonstrate it. You should be able to figure this out, though, by experimenting with the sample program provided.
- There are commands to rotate your rat 90, 180, and 270 degrees.
- There is a command to move your rat forward one unit.
- There is a command to move your rat backward one unit, without changing direction.
- There is a command to lob a projectile in the direction your rat is facing. (“Fire a missile”).
- Projectiles have a finite speed (you can choose it, say 1 cell every 200 milliseconds).
- A rat needs some amount of time to recuperate (your choice of how long) after lobbing a projectile. It can do anything except for launch another projectile. It is an allowable design decision to limit the number of outstanding “missiles” a rat can own to 1, and as soon as a missile hits a rat or a wall, the rat can launch another one. Alternatively, you can say that a rat can recover after 2 (for example) seconds. We recommend that you keep this as simple as possible.

\(^1\)Mazewar has been used in classes at Stanford University and the University of Pennsylvania. Dr. David Cheriton (Stanford) and Dr. Michael Greenwald (Penn) have generously made the project and related support code available for our use.
Figure 1  Sample view of the maze
• A rat touched by an opponent’s projectile is tagged instantly; projectiles pass through their own rat transparently, without causing any harm. Projectiles/missiles are the full width of the corridor, so that if a projectile passes through any cell, it certainly tags the rat in that cell.

• A rat that is tagged loses 5 points. A rat that tags another rat gains 11 points. A projectile that is lobbed costs 1 point.

• When a rat is tagged, it reappears at a random position (in a clear cell only – not one occupied by a wall or another rat) with a random orientation (though not facing a wall) in the maze.

Although there are two rats in the Maze, in the figure, the arrow in the bird’s eye view represents only your rat, “Frobber”. No arrow representing “Demo”, the other rat, appears. If such arrows were visible, the arrow representing “Demo” would be pointing to the right in the space immediately to the left of “Frobber’s” arrow.

In the perspective-view, however, “Demo” does appear. It is represented by an eyeball (all rats appear the same), but there is no representation of “Frobber”, because the convention is that we are looking at the scene from “Frobber’s” eye.

In the text area, “Demo” is highlighted because it is visible in the perspective view.

1.2 Keyboard and Mouse Commands

Commands are as follows:

<table>
<thead>
<tr>
<th>Input Gesture</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>About face (180 deg turn)</td>
</tr>
<tr>
<td>S</td>
<td>Left turn (270 deg turn)</td>
</tr>
<tr>
<td>D</td>
<td>Forward</td>
</tr>
<tr>
<td>F</td>
<td>Right turn (90 deg turn)</td>
</tr>
<tr>
<td>Space</td>
<td>Back one space</td>
</tr>
<tr>
<td>Mouse Left</td>
<td>Peek left</td>
</tr>
<tr>
<td>Mouse Right</td>
<td>Peek right</td>
</tr>
<tr>
<td>Mouse Middle</td>
<td>Launch projectile</td>
</tr>
</tbody>
</table>

2 Mazewar Programming Framework

The programming assignments that require you to use code that we provide are all to be done on the CADE Linux machines. If you do not have a CADE account, you will need to acquire one ASAP. You use the

2.1 Initial Implementation

The directory /home/cs5961/projects/mazewar/ on contains the support code for the Mazewar projects. The version of sample-toplevel.c provided for you places a single rat in a random location in the maze for your amusement. It is a single (local) player mazewar, without shooting or projectiles, and thus is only mildly amusing.

Use the provided single-player version of mazewar as the starting point for your multi-player mazewar game. You can build the single-player mazewar by copying Makefile, mazewar.h, and sample-toplevel.c to your own directory, then typing make all. The result should be an executable file called: mazewar. Make sure the first non-commented line of your Makefile applies to the type of machine you are on: gexx’s are Intel boxes running Linux. You should be able to run it on any machine that runs the X window server.

Please note that mazewar optionally takes two arguments: -robot and -time_interval N. If mazewar is called with -robot, then the program will randomly choose keyboard input to control the rat. It will choose a new action every -time_interval milliseconds. -time_interval defaults to 500 milliseconds. The robot mode will only be used for testing and timing your implementation.
2.2 Programming Assignment

You will modify `mazewar.h` and `sample-toplevel.c` during this assignment. If you need to create any other files, you must modify your copy of `Makefile`. If you need help with the `Makefile`, please ask me, other members of the class, or Josef (the TA). Do not spend or waste time trying to solve problems with the `Makefile` - the point of this exercise is to gain a small amount of experience with distributed systems, not to learn about the Unix(tm) make facility.

2.2.1 Utility and Initialization Routines

```c
/* The following routine sets x, y, and dir to be a random * point in the maze, guaranteed to not be inside a wall, * and dir is guaranteed to not be facing a wall. * You will need to modify this in order to guarantee that * a randomly placed rat is not placed on top of another * rat. It is located in sample-toplevel.c */
NewPosition(Loc *x, Loc *y, Direction dir);
```

```c
/* Called to initialize the maze, and the window system and * stuff. * Pass in argc and argv as passed to the toplevel program */
MazeInit(int argc, char **argv);
```

2.2.2 Display Routines

The following routines are used to control the display manager. They are not meant to control the state of the game.

You inform the display manager of the position of each rat you know about by calling `SetRatPosition`. When a rat leaves the game you call `ClearRatPosition`. If you want the perspective view to be updated, call `ShowView`. If you want to update the bird’s eye view, call `ShowPosition`.

```c
/* * Sets Rat # ratId to be at position xpos, ypos, and facing in * direction dir */
SetRatPosition(ratId, xpos, ypos, dir)
RatId ratId;
Loc xpos, ypos;
Direction dir;

/* Rat # ratId is no longer to be displayed */
ClearRatPosition(ratId)
RatId ratId;

/* * Show the perspective view from position x,y in direction dir. * Other rats are displayed as set by previous calls to * RatPosition. */
ShowView(x, y, dir)
int x, y;
Direction dir;

/* * Show my rat’s position as an arrow in the top-down view. */
ShowPosition(xloc, yloc, tdir)
int xloc, yloc;
Direction tdir;
```
### 2.2.3 Input Routines

The following routines are used to interact with the user. They all take an argument, `prompt`, which is a string used to prompt a user, and they all return their values in the locations pointed to by the other arguments, and all of the returned values are allocated out of the heap by `malloc`. You can manage the memory accordingly.

```c
/* returns a user name, if none is given the username (login ID) is determined from the environment. */
getName(char *prompt, char **ratName);

/* prompts for an arbitrary string. CR returns NULL. The string is returned in string. */
getString(char *prompt, char **string);

/* similar interface to getName. Provide a prompt, and then addresses in which the routine will store the name of a host, and its Sockaddr. The hostname will be checked for validity, and won’t give up until the user returns a valid host name, or a CR which will return as NULL. */
getHostName(char *prompt, char **hostName, Sockaddr *hostAddr);

/* Returns next X event, or an arrival of a network packet on socket. If your implementation requires you to listen on more than one socket at a time, you must modify the function NextEvent, which is found in the file winsys.c. */
NextEvent(MWEvent *event, SockAddr socket)
```

### 2.2.4 Required Functions

You need to provide the following functions, with these exact names, and these interfaces:

```c
/* returns the score for rat ratId */
Score GetRatScore(RatId ratId);

/* returns the name of rat ratId */
char *GetRatName(RatId ratId);

/* convert net representation of packet to local representation. If you use the standard IP representation of numbers, then you can use the macros ntohl, ntohs, etc to do your conversions. */
ConvertIncoming(MW480Packet *p);

/* convert local representation of packet to net representation. If you use the standard IP representation of numbers, then you can use the macros htonl, htons, etc to do your conversions. */
ConvertOutgoing(MW480Packet *p);
```

/* dispatch on type of packet, and perform whatever operations are necessary, based on the content. */
processPacket (MW480Packet *p);

netInit must initialize the network for use, and enter us into a game. It will use the information acquired by getName.

It must either join an existing game (if specified by the user), or start a new one.

netInit()

2.2.5 Sample Network Code

/* Here is some code to show how to use the socket and bind calls */
{
    /* Pick a MAZEPORT that does not conflict with any other service */
    /* or other teams’ mazewar implementation. */
    M.mazePort = htons(MAZEPORT);
    gethostname(buf, sizeof(buf));
    if ( ((thisHost = resolveHost(buf)) == (Sockaddr *) NULL) )
        MWError("who am I?");
    bcopy((caddr_t) thisHost, (caddr_t) &M.myAddr, sizeof(Sockaddr));
    nullAddr = M.myAddr;
    bzero((char *)&nullAddr.sin_addr, sizeof(nullAddr.sin_addr));

    /* Figure out which port I can use. Start at MAZEPORT and go up till there’s a free one. We use nullAddr so that we can receive any incoming packet, in case there’s some multi-homed nonsense. However, not binding may cause * troubles if a specified host is on a net that causes * the source of the packets to be something other than what * M.myAddr is... There also seems to be a problem with * broadcasts not being accepted on sockets that have an * address bound to them. */

    M.theSocket = socket(PF_INET, SOCK_DGRAM, 0);
    if ( M.theSocket < 0 )
        MWError("can’t get socket");
    for ( port = ntohs(M.mazePort); ; port++ ) {
        nullAddr.sin_port = htons(port);
        if ( bind(M.theSocket, &nullAddr, sizeof(nullAddr))<0 )
            if ( errno != EADDRINUSE )
                MWError("netInit binding");
            else
                continue;
        else
            break;
    }
    M.myAddr.sin_port = nullAddr.sin_port;
}

/* Sample code to send a packet to a specific destination */

/* Notice the call to ConvertOutgoing. You must call * ConvertOutgoing before any call to sendto. */

sendPacketToPlayer(ratId)
RatId ratId;
{
    MW480Packet pack;
    DataStructureX *packX;

    pack.type = PACKET_TYPE_X;
    packX = (DataStructureX *) &pack.body;
    packX->foo = d1;
    packX->bar = d2;
    ....

    ConvertOutgoing(pack);
    if (sendto((int)mySocket, &pack, sizeof(pack), 0,
                (Sockaddr) destSocket, sizeof(Sockaddr)) < 0)
        { MWError("Sample error") ; }
}

/*
Sample code to send a broadcast packet, if you'd need to. It is
only for the local net, though. It also has a bit of the
receiving code, too...

Notice that BROAD_ADDR is gotten from the Makefile.
*/
Sockaddr *doBroadCast()
{
    #ifndef NO_BROADCAST
    register int i;
    int cnt = 0;
    int maxAnswers = 10;
    static Sockaddr rmtAddr;
    struct timeval timeout;
    MW480Packet *p = (MW480Packet *) malloc(sizeof(MW480Packet));
    struct answer *answers;
    int fds, fd = M.theSocket;
    int ret, oc;
    Boolean succeeded(), freeSlot();
    struct in_addr inet_makeaddr();

    printf("Broadcasting and checking..."); fflush(stdout);

    #ifdef SO_BROADCAST
    #ifdef BSD_43
    {
        int one = 1;

        if (setsockopt(M.theSocket, SOL_SOCKET, SO_BROADCAST,
                &one, sizeof one) < 0)
            MWError("can't get broadcast permission");
    }
    #else BSD_43
    if (setsockopt(M.theSocket, SOL_SOCKET, SO_BROADCAST, NULL, 0) < 0)
        MWError("can't get broadcast permission");
    #endif BSD_43
    #else SO_BROADCAST
    rmtAddr = M.myAddr;
    rmtAddr.sin_addr = inet_makeaddr(inet_netof(M.myAddr.sin_addr),
                      BROAD_ADDR);
    rmtAddr.sin_port = M.mazePort;
    answers = (struct answer *) malloc((unsigned) (maxAnswers * sizeof(*answers)));
    setupPacket(p);
    ConvertOutgoing(p);
}
if (sendto(fd, (char *) p, sizeof(*p), 0, &rmtAddr, sizeof(rmtAddr)) < 0)
    fprintf(stderr, "Can’t broadcast the survey\n");

while (1) {
    fds = (1<<fd);
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;
    ret = select(32, &fds, NULL, NULL, &timeout);
    if (ret < 0)
        MWError("broadcast select");
    if (ret == 0)
        break;
    answers[cnt].srclen = sizeof(Sockaddr);
    if (recvfrom(fd, &answers[cnt].packet, sizeof(RatPacket), 0,
                 &answers[cnt].source, &answers[cnt].srclen) < 0)
        MWError("broadcast recv");
    ConvertIncoming(&answers[cnt].packet);
    if (answers[cnt].packet.type != PACKET_TYPE_X)
        continue;
    if (!succeeded(&answers[cnt])) {
        RStatus rs;
        rs = (RStatus) &answers[cnt].packet.body;
        setupPacket(p);
        ConvertOutgoing(p);
        if (sendto(fd, p, sizeof(*p), 0,
                    &rs->rats[rs->x].addr,
                    sizeof(rs->rats[rs->x].addr)) < 0)
            fprintf(stderr, "survey resend can’t broadcast");
        continue;
    }
    if (freeSlot(&answers[cnt]))
        if (++cnt == maxAnswers)
            break;
    for (i = 0; i < cnt; i++)
        if (succeeded(&answers[i])) {
            rmtAddr = answers[i].source;
            break;
        }
    free((char *) answers);
    free((char *) p);
    if (i == cnt)
        return (Sockaddr *) NULL;
    else
        return &rmtAddr;
} #else NO_BROADCAST
        return (Sockaddr *) NULL;
    #endif NO_BROADCAST
}

You will be asked to measure the network traffic of your system as part of the assignment. Unfortunately, we can’t provide an unloaded network, or single user machines. A simple packet count and coarse clock measurements will have to suffice. If you are unsure of how to measure the number of packets generated by your program, please consult with a fellow classmate, or with us. For the purpose of this assignment, it will be sufficient to install a counter in your program that gets incremented every time you send a packet. Be sure to include a counter on the receive side to determine whether or not packets are getting lost.

You should be able to do this work, mostly, without modifying any file other than sample-toplevel.c and, possibly, creating some files of your own.
2.3 Unexpected Modifications

We don’t expect you to have to make many changes outside of sample-toplevel.c. However, there are a couple of implementation choices that you might consider that will require such changes. This section is an attempt to make those changes easier.

If you listen on more than one socket at a time, you will need to modify the code in NextEvent. This function is called by play() in sample-toplevel.c and is implemented in winsys.c.

It should be possible to create your own program without needing to modify the name of M, the representation of the maze itself, or the slots in M that refer to the location and orientation of the local rat, and the peek, and so on. If you feel you absolutely must change one or more of these things, then please check the following pieces of code.

If you modify init.c, display.c, or winsys.c, please create a copy in your own directory and modify Makefile to remove the ${MZDIR} prefix in SRCS.

First, several places in the code refer to the state of the maze and the local rat by using the data structure M. You are free to change this data structure and its name.

If you do, you must modify the references in the code that refer to the maze by M.maze, and to the location of the local rat by M.xloc, M.yloc and M.dir. Similarly, m.MyRatId, M.xPeek, M.yPeek, and M.dirPeek.

We assume that you will be making substantial changes to the file sample-toplevel.c, so we won’t enumerate all the references there.

There are no references in display.c.

In winsys.c:

RepaintWindow() uses M.xloc, M.yloc, and M.dir. If you change this datastructure (M), or even just change its name, modify RepaintWindow to include the location of the local Rat.

initMaze refers to M.maze. If you change the data structure or the name of M, update initMaze to point at the maze correctly.

RandomEvent refers to M.peeking, M.dir, M.peekdir, and M.myRatId.

In init.c

MazeInit and getMaze refer to M.maze and M.xloc, M.yloc, M.dir etc.