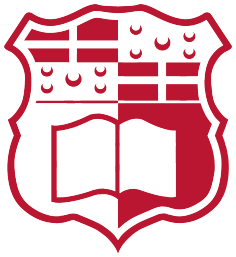

Sneaky Snakes

Implemenatation of a multiplayer variant of snakes over TCP/IP

Andre' Vella
32601L
andre.vella.19@um.edu.mt

University of Malta

An assignment submitted in fulfilment of the requirements for the unit of Operating Systems and Systems Programming II in 2022.

Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines” (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta)

I, the undersigned, declare that the Assigned Practical Task report submitted is my work, except where acknowledged and referenced.

Andre’ Vella

July 5, 2022

Contents

1 | Video Presentation

A video presentation discussing implementation and showcasing the game is given in the following link: video link.

2 | Compiling Server and Client

- The project was compiled on multiple debian-based systems (POP-OS,UBUNTU,LINUX-MINT) and makes use of the ncurses library (can be installed on a debian system by running the command `sudo apt-get install libncurses5-dev libncursesw5-dev`)
- A make file is included for easy compilation (compile using `make` command).
- 2 binaries will be produced: server & client.
- Socket port is set to 6969 (`#define PORT 6969`).
- To run the server run `./server`. To start a client run command `./client <ip address>`. Where the `<ip address>` is a place holder of any IP-address that the server resolves to from the `AF_INET` family of addresses.

3 | Design

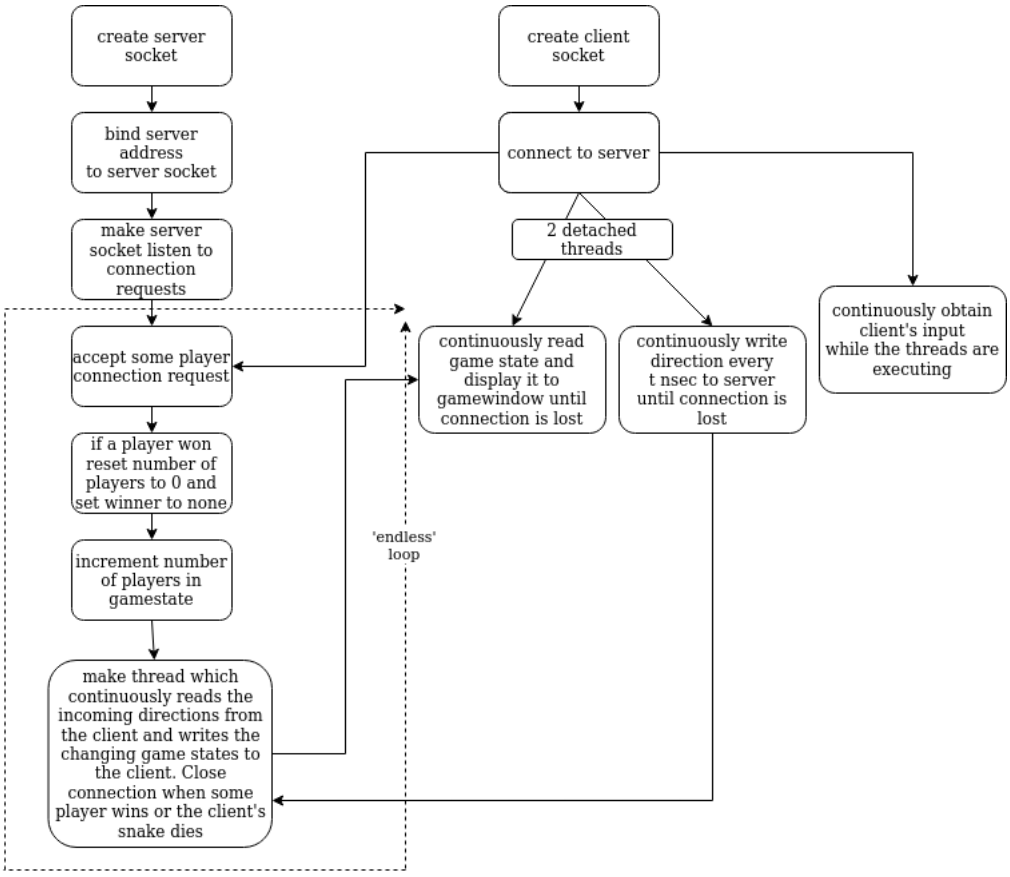

Figure 3.1: A diagrammatic overview of the project.

3.1 | Game State

- Every element in the game is naturally a point in $\mathbb{R}^2$ filling a table T of r rows and c columns. For this implementation we define $r = 20$ and $columns = 40$ (`#define ROWS 20` and `#define COLUMNS 40` respectively).
- The coordinate (x, y) defines the entry $(x + 1, y + 1)$ in our table T.

```
typedef struct coordinate_struct{
    unsigned short  x,y;//a coordinate contains an x & y point
} coordinate_t;
```

Listing 3.1: defining a coordinate type

- Each **snake/player** is made up of a point which represent the snake's **head** and a set of points which represents the snake's **tail** where the size of the tail is predefined to be `MAX_SNAKE_LEN-1`. Since a snake of winning length is of size 15, we set `MAX_SNAKE_LEN` to be 20 for some overhead. A snake also has a state `isDead` which is used to check if a snake hit the boundary or some other snake and another variable holding the length of the snake which is useful when updating the snakes coordinates with each move, if a snake eats some fruit, displaying the snake etc.

```
typedef struct snake_struct{
    short int length;
    bool isDead;//indicate if a snake is dead (collided with
   some propery of the map)
    coordinate_t head;//a head is a cordinate
    coordinate_t tail[MAX_SNAKE_LEN-1];//a tail is at max 14
   cordinates long
} snake_t;
```

Listing 3.2: defining a snake type using the coordinate type defined in listing 3.1

- Each game state has an array of snakes of size `MAX_PLAYERS` which is defined to be 100 (I think this is more than enough given the table T

size). It has an array of coordinates of size MAX_FRUITS which are used to represents the points in our table where the fruit will appear. MAX_FRUITS is defined to be 6 that is at most only 6 fruit can exist.

```
typedef struct gamestate_struct{
    int numberOfPlayers;
    int playerWinner;
    coordinate_t fruits[MAX_FRUITS];//MAX_FRUIT -> max number
   of fruits that can appear on a gameboard
    snake_t snakes[MAX_PLAYERS]; //MAX_PLAYERS-> max number of
   snakes that can appear on a gameboard
} gamestate_t;
```

Listing 3.3: a game state

3.1.1 | A snake's movement

The following algorithms describes all the functions that were implemented in order to handle a snake's movement based on a given direction `direction`.

Algorithm 1 `void moveSnake(pthread_mutex_t,gamestate_t* ,...)`

```
define coordinate_t newHead
if snakeDirecton==DIR_UP then
    newHead.y=snake.head.y-1;
    newHead.x=snake.head.x;
end if
if snakeDirecton==DIR_DOWN then
    newHead.y=snake.head.y+1;
    newHead.x=snake.head.x;
end if
if snakeDirecton==DIR_LEFT then
    newHead.y=snake.head.y;
    newHead.x=snake.head.x-1;
end if
if snakeDirecton==DIR_RIGHT then
    newHead.y=snake.head.y;
    newHead.x=snake.head.x+1;
end if
declare bool hitBoundary and assign it to false
if newHead.x<=0||newHead.y<=0||newHead.x>=COLUMNS-1||newHead.y>=ROWS-1 then   ▷ if snake hit boundary
    assign hitBoundary to true
end if
if hitBoundary then
    mark snake as dead (snake.isDead=true)
else
    call nextMove(gameStateLock,gameState,snakeIndex,snake,newHead) where we check for properties such as: if new
head is colliding with fruit where we remove old fruit and add a random new fruit and append the length of the snake,
check if snake collides with another snake where we mark the snake as dead, or check if snake collides with nothing!
    mutex lock gameStateLock
    copy updated snake to corresponding snake in gameState
    mutex unlock gameStateLock
end if
```

- Thread Safe: MUTEX LOCK/UNLOCK are used to lock/unlock the shared state of the game between threads to ensure a single atomic update to avoid data races when different threads are manipulating the game state (with snakes movements, updating fruit positions etc).

3.1.2 | Displaying the game states

- A window is created to display an initial starting screen and continuously display the incoming game states (snakes and fruits) from the server.

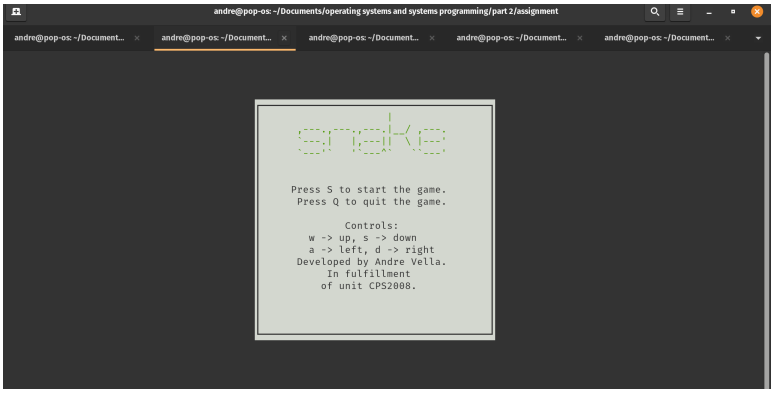

Figure 3.2: Starting screen

Figure 3.3: Game screen

  - A function `centerText(win,row,string)` is written to help us center text on any window.

- We use the size of table T to set the size of the window and make reference to the size of the terminal screen to center it

```
1 gameWindow = newwin(ROWS,COLUMNS,stdscr->_maxy/2-ROWS/2,stdr->
    _maxx-COLUMNS/2);
```

Listing 3.4: Initialising Game Window

- Colour pairs (i.e. foreground and background) are defined forthe window, fruits and snakes. E.g.

```
1     if(init_pair(FRUIT_COLOUR,COLOR_WHITE,COLOR_RED)==ERR){
2         delwin(gameWindow);
3         echo();
4         curs_set(1);
5         endwin();
6         fprintf(stderr,"ERROR : colour definition of white
      foreground and red background failed");
7         exit(1);
8     }
```

Listing 3.5: Specifying the colour of the fruit

- Each time a client reads a game state from the server it display the snakes by looping through `i=0,1,..,gameState.numberOfPlayers` snakes and displaying each snake's head and tail using the following code snippets respectively.

```
1 mvwprintw(gameWindow,ntohs(gameState.snakes[i].head.y),ntohs(
      gameState.snakes[i].head.x),":)");
```

Listing 3.6: display the i'th snake head

```
1 mvwprintw(gameWindow,ntohs(gameState.snakes[i].head.y),ntohs(
      gameState.snakes[i].head.x),":)");
```

Listing 3.7: display the i'th snake tail

Similarly we display the `MAX_FRUIT` fruits.

- Error handling mechanisms:
  - It was noticed that if we try to print at an area which is out of terminal size a segmentation fault is thrown. This was handled by checking that the game window fits the terminal size.

```
1 if(stdscr->_maxx<COLUMNS||stdscr->_maxy<ROWS){
2     endwin();
3     fprintf(stderr,"Terminal size not supported (too small
      )\n");
4     exit(EXIT_FAILURE);
5 }
```

Listing 3.8: checking that everything fits in the terminal screen

- Checking if the terminal supports colour and checking that the colour definitions do not fail as highlighted in listing 3.5.

```
1 if (has_colors() == FALSE) {
2     endwin();
3     printf("Terminal does not support color\n");
4     exit(EXIT_FAILURE);
5 }
```

Listing 3.9: checking if terminal supports colour

3.2 | Reading and writing

3.2.1 | Functions used to read/write n bytes to descriptor

To ensure that the correct number of bytes are read/written from both the client's and the server's side 2 functions from book Unix Network Programming, Volume 1: The Sockets Networking API, 3rd Edition were used:

```
1  ssize_t writen(int fd, const void *vptr, size_t n)
2  {
3      size_t nleft;
4      ssize_t nwritten;
5      const char *ptr;
6      ptr = vptr;
7      nleft = n;
8      while (nleft > 0) {
9      if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
10         if (nwritten < 0 && errno == EINTR)
11             nwritten = 0;   /* and call write() again */
12         else
13             return (-1);    /* error */
14     }
15     nleft -= nwritten;
16     ptr += nwritten;
17     }
18     return (n);
19 }
```

Listing 3.10: Writing n bytes to descriptor

```
ssize_t readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    size_t nread;
    char   *ptr;
    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR) nread = 0; /* and call read() again
   */
            else return (-1);
        } else if (nread == 0)  break; /* EOF */
        nleft -= nread;
        ptr += nread;
}
```

Listing 3.11: Read n bytes from descriptor

3.2.2 | Reading and writing from the server's side

- First we need to accept a connection with a new player return the file descriptor and run a detached thread associated with that file descriptor.

Algorithm 2 accept connection with new clients

```
while true do
    declare playerSocketFd
    declare playerAddressFd
    if playerSocketFd=accept(serverSocketFd,playerAddress,clilen)<0 then
        error("ERROR: connection with some player")          ▷ print error and exit with an error status
    end if
    declare playerSocketFd
    declare playerAddressFd
    if winner exists in gameState then                                          ▷ refresh state of game
        in gameState reset winner to none
        in gameState reset number of players to 0
    end if
    increment number of players in gameState                                    ▷ player joined the game
    if pthread_create(thread,NULL,updateGameState,playerSocketFd)<0 then
        close(serverSocketFd)
        error("ERROR : creation of a new thread failed")     ▷ print error and exit with an error status
    end if
    if pthread_detach(thread)<0 then
        close(serverSocketFd)
        error("ERROR : thread detaching failed failed")      ▷ print error and exit with an error status
    end if
end while
close(serverSocketFd)
```

- **THREAD SAFETY** : since we do not have any other threads joining the executed threads we make the threads detached to automatically release all resources once execution has finished.
- The routine `updateGameState` invoked by each thread as shown in algorithm 2 is one that handles all the reads and writes to the corresponding client's file descriptor.

Algorithm 3 `void* updateGameState(void* arg)`

Obtain playerSocketFd from argument.
declare bool `endPlayer` and assign it to false
declare `gamestate_t gameStateNetworkByteOrdering;`
`*snake=makeSnake(&gameStateLock,gameState,snakeIndex)` ▷ init a random snake that does not intersect with any point
while !endPlayer **do**
  if player's snakes length is greater or equal to 15 **then** ▷ If player wins
    lock mutex gameStateLock
    mark gameState winner to some predefined int WINNER_CODE
    unlock mutex gameStateLock
    set global var playerWinnerNumber to currentPlayerNumber
    set endPlayer to true
  else if some other player won in gameState **then**
    lock mutex gameStateLock
    set playerWinnerNumber in gameState
      to global var currentPlayerNumber
    unlock mutex gameStateLock
    set endPlayer to true
  end if
  memset gameStateInNetworkByteOrdering to 0 and copy contents of gameState using the appropriate `htonl()` and `htons()` functions
  if `writen(playerSocketFd,(char*)&gameStateNetworkByteOrdering, sizeof(gamestate_t))<=0` **then** break; ▷ connection with stopped/failed
  end if
  if `readn(playerSocketFd,(&gameStateNetworkByteOrdering, sizeof(gamestate_t))<=0` **then** break; ▷ connection with stopped/failed
  end if
  `moveSnake(&gameStateLock,gameState,snake,snakeIndex,dirBuffer)`
  if snake status is dead **then** ▷ check if snake hit some boundary/other snake
    set endPlayer to true
  end if
end while
close(playerSocketFd)
lock mutex gameStateLock
memset correspoding snake in gameState to 0
unlock mutex gameStateLock

3.2.3 | Reading and writing from the client's side

- Once a connection has been established between client and server we run 2 detached threads one that continuously writes the current direction (based on the key pressed) to the descriptor and one that reads the current direction received from the server.

Algorithm 4

```
if connect(playerFD,server_add,size of server address)<0 then
    close(playerFD);                                          ▷ close file descriptor
    error("ERROR : connecting");                              ▷ print error and exit
end if
if pthread_create(&writeThread,NULL,updateServer,&sockfd)<0 then
    close(playerFD);                                          ▷ close file descriptor
    error("ERROR : creation of a new thread failed");         ▷ print error and exit
end if
if pthread_detach(writeThread)<0) then
    close(playerFD);                                          ▷ close file descriptor
    error("Error : detach thread failed");                    ▷ print error and exit
end if
if pthread_create(&readThread,NULL,readServer,&sockfd)<0 then
    close(playerFD);                                          ▷ close file descriptor
    error("ERROR : creation of a new thread failed");         ▷ print error and exit
end if
if pthread_detach(readThread)<0 then
    close(playerFD);                                          ▷ close file descriptor
    error("Error : detach thread failed");                    ▷ print error and exit
end if
```

- The routines `readServer` and `updateServer` invoked by each thread as shown in algorithm 4 are described in the following pseudo codes:

Algorithm 5 void* updateServer(void* arg)

```
declare time to be 100000000 nanoseconds/0.1 seconds
while true do
    if writen(sockfd,&currentDirection,sizeof(char))<=0 then
        gameOver=true                          ▷ a global bool used to stop a loop from reading user's input
        break                                  ▷ The connection dies (or error occurs) and hence we stop writing to server
    end if
    nanosleep time                             ▷ write to server every time seconds
end while
```

Algorithm 6 void* readServer(void* arg)

```
Obtain socket file descriptor sockfd from arg
Initialise gameStateBuffer and allocate sizeof(gamestate_t) bytes to it.
Set bytes of gameStateBuffer to 0 value.
declare gamestate_t gameState
if connect(playerFD,server_add,size of server address)<0 then
    close(playerFD);                                          ▷ close file descriptor
    error("ERROR : connecting");                              ▷ print error and exit
end if
while true do
    if readn(sockfd,gameStateBuffer,sizeof(gameState))<=0 then   ▷ This block will execute if connection is not alive
anymore.
        gameOver=true
        winner=ntohl(gameState.playerWinner)                  ▷ get player winner
        break
    else
        copy contents of gameStateBuffer to gameState
        obtain current player number by obtaining the value from the first
          read of ntohl(gameState.NumberOfPlayers)
        display player numbers
        display snakes
        display fruits
    end if
    pthread_exit(NULL)                                        ▷ terminate thread and return nothing
end while
```

4 | Testing

For testing I made any snake stop its movement when a player tries to move backwards so it gives me a handle to see that the desired results are achieved. A sample test case:

- Connect 6 players to the server.

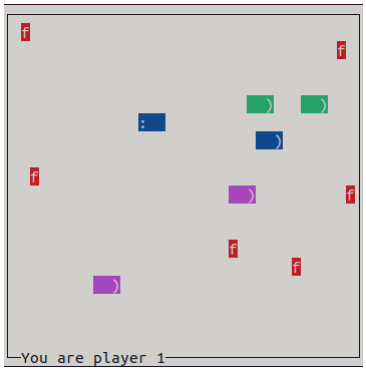

Figure 4.1: Player 1

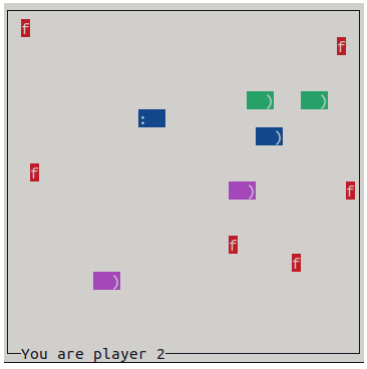

Figure 4.2: Player 2

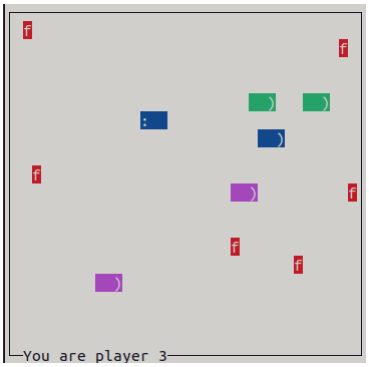

Figure 4.3: Player 3

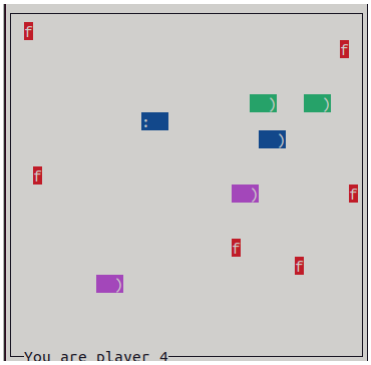

Figure 4.4: Player 4

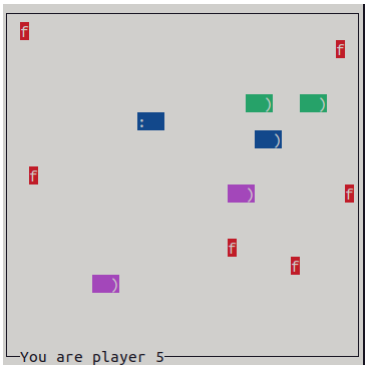

Figure 4.5: Player 5

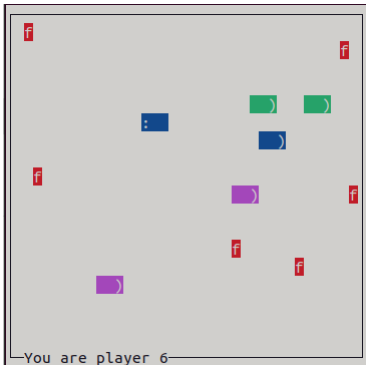

Figure 4.6: Player 6

- Ctrl-C player 5

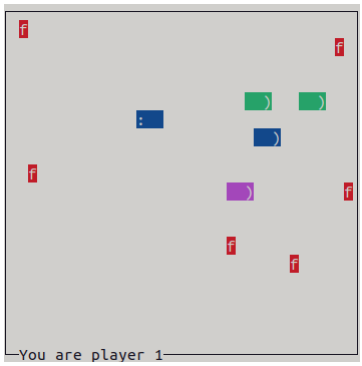

Figure 4.7: Player 1

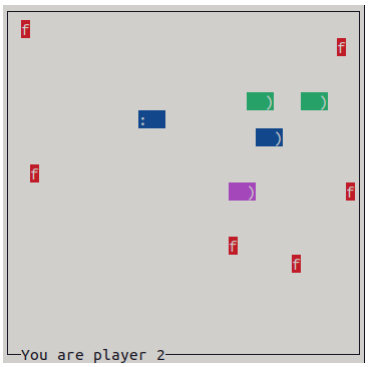

Figure 4.8: Player 2

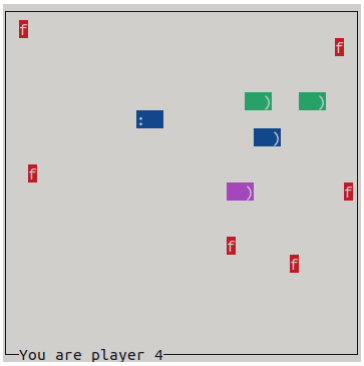

Figure 4.10: Player 4

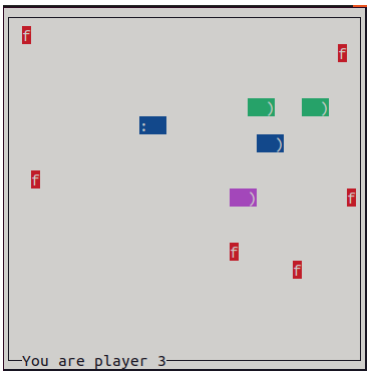

Figure 4.9: Player 3

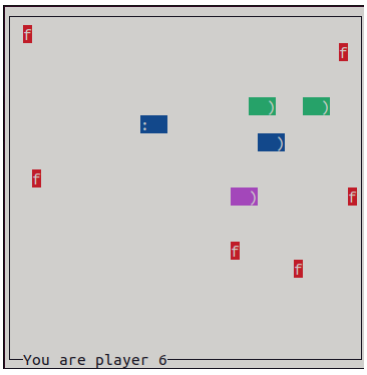

Figure 4.11: Player 6

- Close player 6 terminal

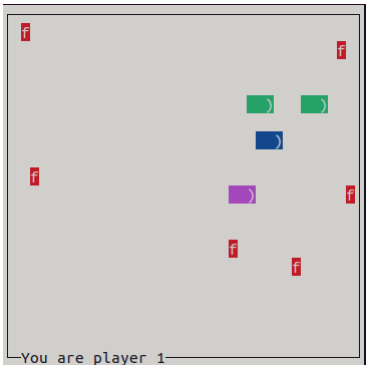

Figure 4.12: Player 1

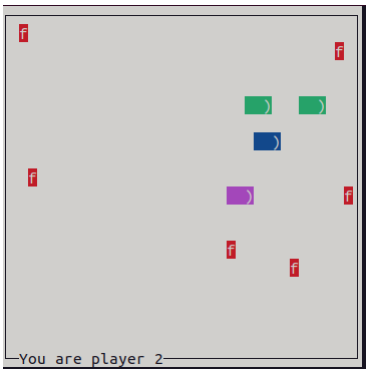

Figure 4.13: Player 2

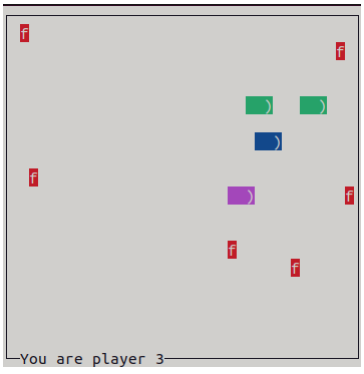

Figure 4.14: Player 3

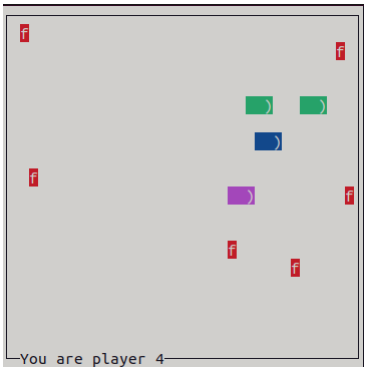

Figure 4.15: Player 4

- Make player 3 crash

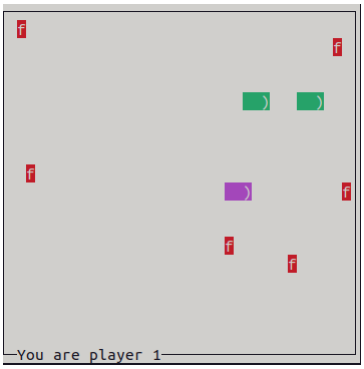

Figure 4.16: Player 1

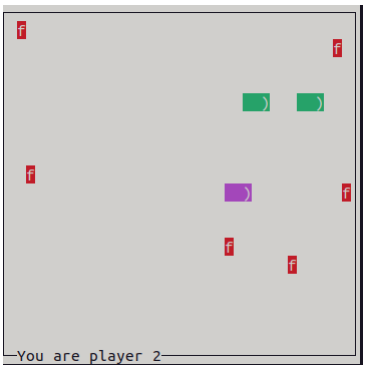

Figure 4.17: Player 2

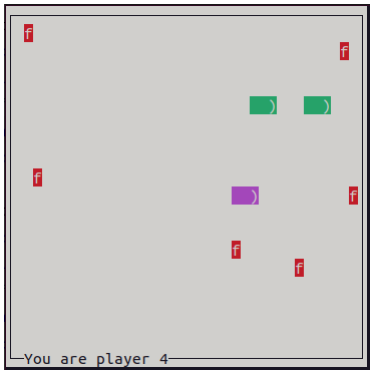

Figure 4.18: Player 4

- Make player 2 eat some fruits and crash

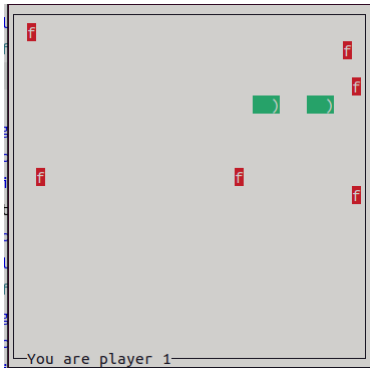

Figure 4.19: Player 1

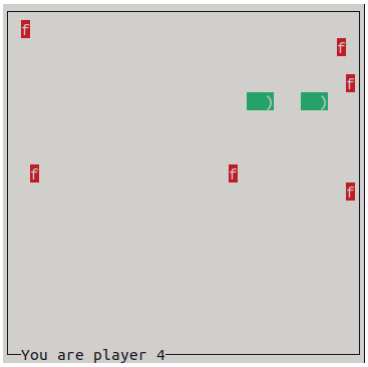

Figure 4.20: Player 4

- Make player 4 win

- Once a player has won the game brakes for all players and the following output screens are shown.

```
andre@andre-Satellite-Pro-R50-B:~/Documents/src$ ./client 192.168.1.218
RIP - you got owned !
Player 4 ate the whole game
andre@andre-Satellite-Pro-R50-B:~/Documents/src$
```

Figure 4.21: Player 1

```
You Won - GG
andre@andre-Satellite-Pro-R50-B:~/Documents/src$
```

Figure 4.22: Player 4

5 | Limitations

- **Some memory leaks:** when a snakes eats a fruit the snakes gets laggy (still playable).
- **A not so efficient game state representation:** when we communicate on different machines over the same network the game gets laggy (still playable), next time in some future implementation I will represent the game state using a 2d-array.