Students identification
- Rodrigo Coimbra 100078
- Tomás Martins 100102

1 Introduction

The objective of this project was to create a multiplayer space invaders game that holds up to 8 players. For this purpose, we can take advantage of threads and communication messages to create a system.

2 Implemented functionalities

2.1 Part A

Table 1: Implemented functionalities (PART A)

	Not implemented	With faults	Totally Correct
game-server.c			
Suitable sockets			yes
Assignment of Astronaut letters			yes
Storage of clients Astronauts and Aliens			yes
Management of astronauts			yes
astronaut-client.c			
Connect			yes
Movement			yes
Zap			yes
Disconnect			yes
Display score			yes
outer-space-display.c + display on game-server.c			
Correct update of screen			yes
Zaps			yes
Destruction of aliens			yes
Rules			
Movement of Aliens			yes
Astronaut zapping			yes
Astronaut zapping (fire rate)			yes
Astronaut zapping (0.5 second ray)		Blocks the program for 0.5 seconds	
Aliens are destruction (points)			yes
Astronauts stunning			yes

Misc			
Messages validation		If a message doesn't follow any of the actions, we choose to skip that action and respond to the user with the same data structure received.	
Optimization of communication			yes
Code organization			yes
Functions return values validation			yes

2.2 Part B

Table 2: Implemented functionalities (PART B)

	Not implemented	**With faults**	**Totally Correct**
game-server.c			
Suitable sockets			yes
Disconnect of clients			yes
Communication thread			yes
Aliens thread			yes
Synchronization			yes
astronaut-display-client.c			
Connect			yes
Movement			yes
Zap			yes
Disconnect			yes
Update of display			yes
Display score			yes
Threads			yes
space-high-scores			
Client on other language			yes
Protocol buffer messages			yes
Rules			

Movement of Aliens			yes
Astronaut zapping			yes
Astronaut zapping (fire rate)			yes
Astronaut zapping (0.5 second ray)			yes
Aliens destruction and points			yes
Astronauts stunning			yes
Aliens population recovery			yes
Misc			
Messages validation		If a message doesn't follow any of the actions, we choose to skip that action and respond to the user with the same data structure received.	
Optimization of communication			yes
Code organization			yes
Functions return values validation			yes

2.3 Description of faulty functionalities

In part A, we choose to use a sleep function to display the zap for 0.5 seconds, as these would be easily solved using threads allowed in part B.

3 PART A Description of code

3.1 Data types

The ships are saved in an array of structs. This struct holds information about the character that the ship represents, its current position with a struct that has positions x and y, its points that are the number of aliens killed, the style of movement (if it moves vertically or horizontally), the timeouts for both zapping delay and stun delay (we save the time that such action occurred), an identifier if it has zapped, to know if the display needs to draw or not and its encryption to handle hackers.

The aliens are also an array of structs. This struct holds information about the alien position and its current state, if its alive or not.

There is a main struct that holds pointers for both the alien array struct and ships array struct.

3.2 Functions List

In this section students should present a list with every implemented function, divided by the various components implemented:

- game-server.c
  run_players: handles all astronaut and aliens related logic. From messages received to updating positions of both astronaut and aliens. It also publishes to the outer display.
  run_aliens: Done in another process, saves the aliens that are alive and sends a random direction for them to move to the function run_players via TCP. Also sends a message when all aliens are killed.
  Astronaut_connect: Handles all new connections to astronauts. Chooses a predefined letter in order of connection from A to H. Puts them in a predefined place in the board from top, bottom, left, right, then in second road top, bottom, left, right.
  Astronaut_movement: Handles the logic behind updating the movement of astronauts.
  Astronaut_zap: Checks recharge and stunned timers, sees if the zap has to be vertical or horizontal and handles its logic.
  Alien_movement: Saves the new positions of the aliens from the received message of the other process.
- astronaut-client.c
  main: executes the main logic behind the client, simply sends messages to the server with the player input, and receives its pontuation.
- outer-space-display.c.
  There are specific functions to draw the data received from the server and delete the old data.

Additionally, all the functions have their own function descriptions inside the code
For each function students should describe the objective of each function.

3.3 Implementation details

Description of the implementation of the astronaut zapping fire rate.

Each astronaut has a struct. In it, there is a variable that holds the last time that a zap has been fired. Every time that the user requests to fire, it checks if enough time has passed for it to fire again or if not skip the action.

Description of the implementation of the astronaut zapping 0.5 second ray display

We update the display with the zap, sleep for 0.5 seconds and then erase the zap from the display.

Description of the implementation of the astronauts stunning (immobility)

The same has the zapping fire rate. There is a variable that holds the last time that it has been hit. Every time that the user requests to move, it checks if enough time has passed for it to move again or if not skip the action.

4 PART B Description of code

4.1 Data types

The same as part A, but now the `ships` struct has been modified to include the zap position, allowing it to track where the zap was fired, as the ship can now move while its zap is being displayed.

4.2 Functions List

In this section students should present a list with new of modified functions from Part A to Part B:

- game-server.c
  run_aliens: before it was a different process, now it's a thread.
  keyboard_handler: handles the input of "q" in case the server wants to quit. Is a thread.
  Thread_refresh: thread that updates the display every 50 ms.
- astronaut-display-client.c
  The functions that were separated into outer-space-display and astronaut-client, are now comprised into one program, where the astronaut-client part is now a thread in the function joystick.
- space-high-scores.c

  Implemented in python, receives messages of type ScoreBoard, which points to another message ScoreUpdate which holds information about the astronaut letter and its points. The program receives this information from every ship and displays it.

For each function students should describe the objective of each function.

4.3 Implementation details

Description of the implementation of the astronaut zapping fire rate.

Same as part A.

Description of the implementation of the astronaut zapping 0.5 second ray display.

There is a thread that handles this. When it registers the fire, the thread writes on the Window the ray and sends a msg through a publisher for the outer displays update their display too. Then it sleeps for 0.5 seconds and erases the rays and send a msg again, which erases the zap from the outer display.

Description of the implementation of the astronauts stunning (immobility).

Same as part A.

Description of the implementation of the aliens destruction and points

Depending on the zap orientation (vertical/horizontal), it checks all the elements in its column/row and if it is an alien alive it erase it and award a point to the player that destroyed it.

Description of the implementation of the aliens population recovery

Every second, when the aliens' positions are updated, if the number of aliens alive remains the same for 10 seconds, 10% of that number is calculated and revived at random positions. The time of the last alien kill is stored in a variable and compared to the current time to determine if 10 seconds have passed. In this implementation, the number of aliens alive at any given time is limited to the initial number of aliens spawned at the beginning of the program.

4.4 Threads

In the section students should present a list of every thread implemented, divided by components:

- astronaut-display-client.c
  One thread handles all the commands from the keyboard and sends messages to the server. This is essential for the user to play and see the display at the same time.
- game-server.c

  One thread does all the aliens movement and population recovery and publishes their position to the displays.

  Another thread waits for the server to press the letter “q”, which sends a messages to disconnect all clients and close the server.

  Another thread simply refreshes all windows every 50 ms. We do to update the display information because the getch() from the keyboard thread sometimes conflicts with the screen being displayed.

Students should describe its overall functioning and objectives of each thread.

4.5 Shared variables

In this section students should present all shared variables that are accessed by multiple threads, divided by components.
Students should describe the objective of each variable and what threads access it.

The structure that saves the ships and the aliens is a shared variable, this is important as the run_players and run_aliens threads both need to access the alien information. The run_players thread needs this to check collisions with aliens.

The windows of the game are also shared variables, these are accessed in run_players thread to update the scores and players movement, in the alien thread to update the alien movement. The refresh thread is responsible to update the windows.

A variable for informs the end of the game also is shared. Either by the run_players when the server presses “q”, or when all aliens are killed in run_aliens.

The last shared variable is the socket for the publisher, which is accessed in the run_players thread to publish the ship data to all displays and in run_aliens to display all the aliens.

4.6 Synchronization

In this section students should present all synchronization variables used: mutexes.

Students should describe what type of guarding of synchronization they implement, what variables are related and on what functions they are accessed.

We used mutexes to guard critical regions. The mutexes used were:

- Client
  - Variable lock -> Is used to guard the disconnect value, which when 1 means that every thread must end and the program should finish. Used in the following functions:
    - Joystick
    - display
  - Variable lock_space -> Is used to guard the windows, when we have to refresh them or update them. Used in the following functions:
    - draw_vertical
    - draw_horizontal
- Server
  - Variable lock_space -> Is used to guard the windows, when we have to refresh them or update them. Used in the following functions:
    - thread_refresh
    - alien_movement
    - astronaut_connect
    - astronaut_movement
    - astronaut_disconnect
    - display_zap
    - draw_vertical
    - draw_horizontal
  - Variable lock_aliens -> To guard the alien structure, when they need to change their alive status or position. Used in the following functions:
    - run_aliens
    - revive_alien
    - alien_recovery
    - zap
  - Variable lock_game_end -> Used to guard the game end variable in each thread, which is responsible for checking if the game has ended (run out of aliens or someone pressed the key “Q” in the server). When the game has ended, all threads are expected to finish and the program end. Used in the following functions:
    - run_players
    - run_aliens
    - thread_refresh
  - Variable lock_publish -> Use to ensure that only one msg is being published at that time. Used in the following functions:
    - run_players

- run_aliens
- display_zap

5 PART A Communication

5.1 Sockets

In this section student should describe the sockets used in Part A.

We used a req-rep for handling the communication with the clients and a pub-sub for handling the display data.

5.2 Transferred data

In this section students should present the exchanges data structures/messages and relate them to the sockets described earlier.
The main message send with the req-rep socket holds information about which ship is sending the message, their current points, the action which its about to perform, the direction it can move (vertically or horizontally) and its encryption key. For the display we send the information about the aliens and the ships in the struct that has both their information.
Students can present the typedef used and relate them to the messages presented in the assignments.
The types of messages are: Astronaut_connect, Astronaut_movement, Astronaut_zap, Astronaut_disconnect, Alien_movement, Alien_end (For when all aliens are killed and the game ends).

5.3 Error treatment

In this section students should present what type of error treatments were implemented related to communication: validation of read data, verification of read/write return errors, ...

The treatment of errors is implemented with an exit(1). We should have allowed the client to disconnect from the server before exiting the program.

6 PART B Communication

6.1 Sockets

In this section student should describe any new socket in Part B.

We send a new topic in the publisher socket to send the scores to the program in python with protobuf.

We also have a new topic for when the game ends to inform all clients to disconnect. In the clients perspective, the game freezes as it shows the message of who won the game, what happens is that the server closes, so the game freezes, but it sent a message saying that the game ended updating a variable. This variable will only be processed the next time the player presses a button, which will close the client. This way, the client can see the final result of the game.

6.2 Transferred data

In this section students should present the modified messages from part A to Part B.

We now send information with protobuffers. We have a message ScoreBoard which holds multiple ScoreUpdate messages, the same as the number of ships in the game. The ScoreUpdate message has information about the astronaut letter and its score. Additionally, the struct that is published and received through the subscriber in the outer-space-display is different, requiring an adaptation to the code developed in Part A.

6.3 Error treatment

In this section students should present what type of error treatments were implemented related to communication: validation of read data, verification of read/write return errors, …

We should have allowed the server to disconnect all clients before closing itself.

7 Conclusion

Final remarks about the project, what was learned, accomplishments, major difficulties and adversities.

The main objective of the project of creating a system that can communicate via messages and run synchronously with the help of thread and mutexes was achieved. This allowed us to understand better the utility of them, greatly improving the game experience when compared to Part A, where such were not allowed.

One of the members had never worked with threads before, and now sees the benefits of using them. Learning system programming in an environment like this (creating a game), motivates students (at least it did for us).