ACTIVITY

Final Project

CLASS

Programming Languages

STUDENT

Juan Arturo Cruz Cardona

ID

A01701804

PROFESSOR

Benjamín Valdés Aguirre

The student who carried out this work declares that 100% of the content of this activity was written by him.

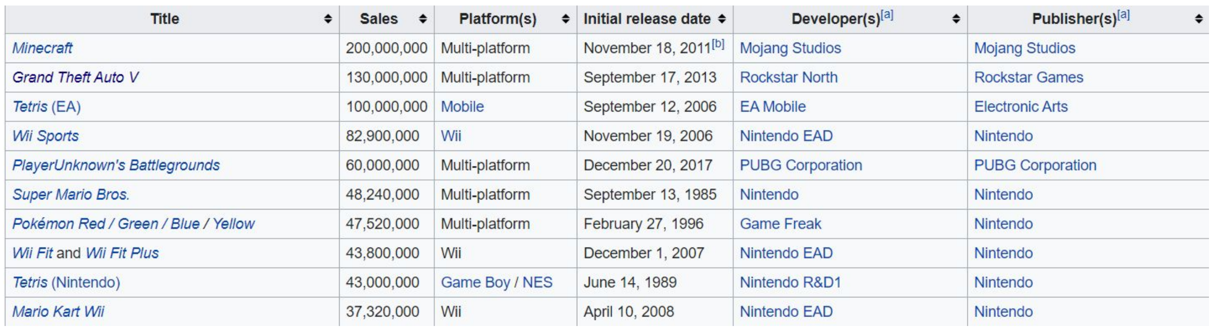
Campus Querétaro June 01, 2020

Contents

Abstract

The popularity of the video games has reach a point where if someone develops a good enough video game for an audience, it can easily get rich. An example is *Grand Theft Auto V*, which is the second best-selling video game of all time with over 120 million copies sold [1] and one of the most financially successful entertainment products of all time, with about $6 billion in worldwide revenue despite of debuting in 2013. And even now, 7 years later, the success of this game continues to give things to talk about in the gaming community.

The following table shows the top 10 best selling video games of all time:



Title	Sales	Platform(s)	Initial release date	Developer(s)[a]	Publisher(s)[a]
Minecraft	200,000,000	Multi-platform	November 18, 2011[b]	Mojang Studios	Mojang Studios
Grand Theft Auto V	130,000,000	Multi-platform	September 17, 2013	Rockstar North	Rockstar Games
Tetris (EA)	100,000,000	Mobile	September 12, 2006	EA Mobile	Electronic Arts
Wii Sports	82,900,000	Wii	November 19, 2006	Nintendo EAD	Nintendo
PlayerUnknown's Battlegrounds	60,000,000	Multi-platform	December 20, 2017	PUBG Corporation	PUBG Corporation
Super Mario Bros.	48,240,000	Multi-platform	September 13, 1985	Nintendo	Nintendo
Pokémon Red / Green / Blue / Yellow	47,520,000	Multi-platform	February 27, 1996	Game Freak	Nintendo
Wii Fit and *Wii Fit Plus*	43,800,000	Wii	December 1, 2007	Nintendo EAD	Nintendo
Tetris (Nintendo)	43,000,000	Game Boy / NES	June 14, 1989	Nintendo R&D1	Nintendo
Mario Kart Wii	37,320,000	Wii	April 10, 2008	Nintendo EAD	Nintendo

Image 1. Top 10 best selling video games, taken from wikipedia

Nowadays, making video games requires many different talents, it is not just an easy task as talking to a PC and asking for a great video game with the potential to be a great success. Things as the visual imagination of an artist, the storytelling skills of an author, the appropriate music skills to create an amazing soundtrack, and of course the technical computer programming skills necessary to make the vision of the game makers reality [2] (the skills this project is focusing on).

Typically, the more advanced the game in terms of performance, graphics and online services, the more advanced the code. That's why the skills of knowing how to read and write computer code is very helpful in making video games, and this project demonstrate the advantages of using multithreading in a basic project such as a 2D video game where multiple things needs to be controlled and executed concurrently.

[1] According to this article from Forbes
[2] More information in the next HowStuffWorks article

1 Context of the project

Multithreading is a lot more common now, to the point where the majority of games are using this technique pretty effectively. Even though, some games have a tendency to lean heavily on one thread with other threads not doing so much.

To understand why that's the case we need to understand how game processing works at a higher level but with a basic logic. Basically there is a *game loop* which does:

1. Receives an input
2. Updates the game (execute game logic, calculate physics, load new objects in game world, etc)
3. Draw output

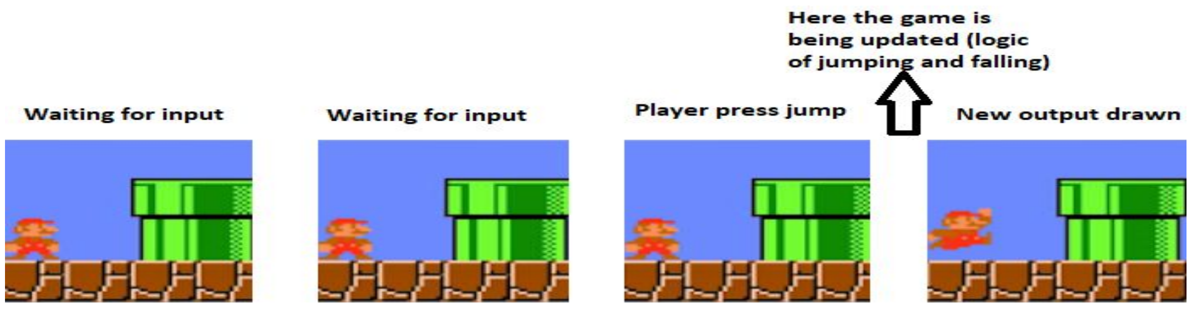


Image 2. Picture made by me to explain the loop I am talking about

And multithreading works best when threads are independent of each other and don't have to battle for a shared resource or be dependent on each other's outputs [3].

Modern games use a job queue system to maximize usage of threads. The most important point of it is that there's a pool of available threads. You have one thread acting as a dispatcher that is receiving jobs in a queue and assigning them to any available thread [4].

Also, it's no coincidence that most recent C++ releases (from 11 to now) has enhanced concurrency (that being the name for doing computations at the same time) covering multithreading [5], and that is well known that C++ is one of the main languages for game development [6].

[3] A concrete example of this is the Dining Philosopher Problem

[4] More information in these forums:

- https://www.reddit.com/r/pcgaming/comments/8t5tzp/the_rise_of_multithreaded_gaming/
- https://stackoverflow.com/questions/5932712/threadpool-multi-queue-job-dispatch-algorithm

[5] According to C++ releases and documentation

[6] According to:
https://www.freelancer.es/articles/programming/what-is-the-best-programming-language-for-games

2 Overview of the Solution

The idea for my project is simple, I took two of the mechanics of the *Monster Hunter World: Iceborne* video game as a reference. These mechanics consist of:

- Allowing multiple players join the same game session but playing in different groups
- Attempt to kill a common enemy between all the players in the game session [7]

With this in mind I began investigating how to implement this idea in Java, which led me to find the use of socket programming.

But that's not all, as commented at the beginning, a video game requires more things than just code, since it must be a complete entertainment experience for the final user, so in an attempt to demonstrate these, I should comment that the project also uses a lot more of different java libraries and design patterns, in purpose to get the maximum performance possible and a good enough demonstration of the object oriented programming, multithreading and concurrency, trying to execute and control the music, image rendering, logic and other things a video game should have.

2.1 Socket Programming

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket listens a particular port at an IP, while other socket reaches out to the other to form a connection [8].

If we read this definition, we can think about the client/server architecture, where the server can be the listener socket in a particular port while clients are the other sockets that reaches out to the server.

This architecture basically is a distributed application structure that divide tasks or workloads between the providers of a resource or service (those are called servers) and service requesters (those are called clients) [9].

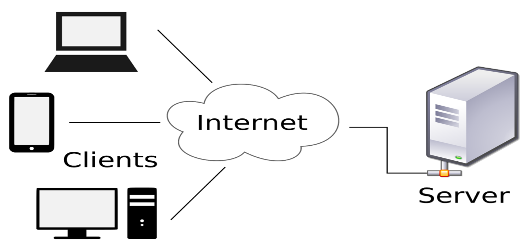


Image 3. Clients communicating with a server via the Internet

[7] Capcom Safi'jiiva Siege
[8] Definition taken from GeeksForGeeks
[9] Definition of client/server architecture taken from wikipedia

Now, with that in mind and remembering about the mechanics of the *Monster Hunter World: Iceborne* video game, I implemented 5 different classes [10] to control the socket management between the clients and the server.

2.1.1 Server Implementation

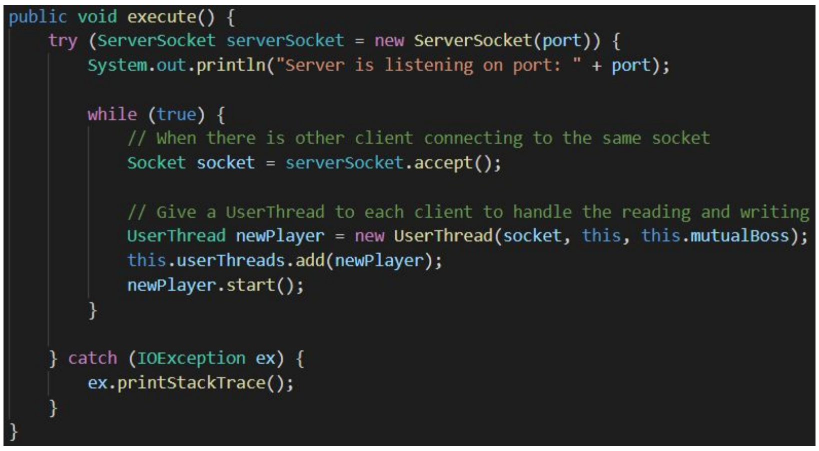
The server is implemented by the next two classes:

Server Class

Is responsible of starting the server and listening on a specific port. If a new client gets connected, it creates an instance of UserThread to serve that client and since each connection is processed in a separate thread, the server is able to handle multiple clients at the same time.

If we compare it to the video game this would be like the game session where the different players join.

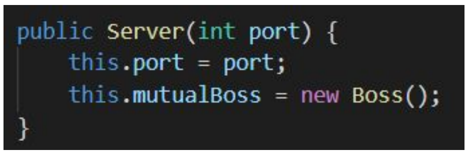
Now, the following code is the method to run and start listening in an specific port, if a new client get connected then proceeds as indicated above.



```java
public void execute() {
    try (ServerSocket serverSocket = new ServerSocket(port)) {
        System.out.println("Server is listening on port: " + port);

        while (true) {
            // When there is other client connecting to the same socket
            Socket socket = serverSocket.accept();

            // Give a UserThread to each client to handle the reading and writing
            UserThread newPlayer = new UserThread(socket, this, this.mutualBoss);
            this.userThreads.add(newPlayer);
            newPlayer.start();
        }

    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Image 4. Method used to accept new client connections and give them a thread to handle communication

In addition, here is created the instance of the global enemy that all players face, which unlike the other that is rendered in each panel, this is the object to which life changes are applied with each shot of the players.



```java
public Server(int port) {
    this.port = port;
    this.mutualBoss = new Boss();
}
```

Image 5. Constructor of the Server class

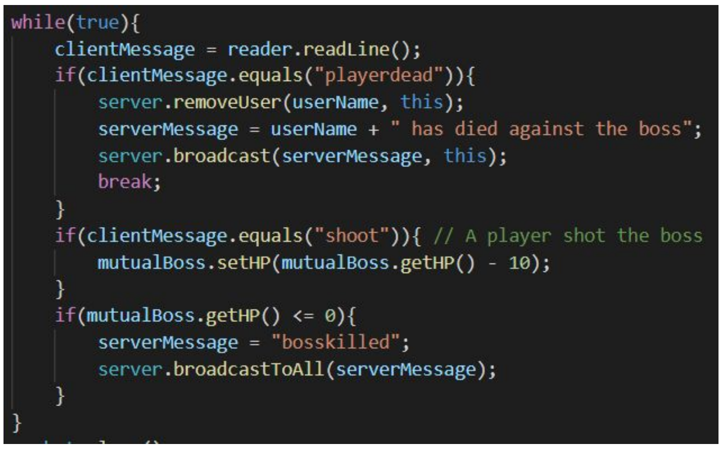
[10] Every time we are talking about classes we're referring to the files with .java at the end

UserThread Class

Is responsible for reading messages sent from the client (instances of the game) and broadcasting messages to all other clients.

The class is constantly checking for these possible status:

- If the client writes a message saying *bye*, is because the player decided to left the game manually or the enemy killed him.
- If the client writes a message saying *shot*, is because the player pressed the button to shoot a bullet and this bullet hit the enemy.
- If there is a moment when the hp of the boss is less than zero, then communicates all the players in the game session they won the game.



```java
while(true){
    clientMessage = reader.readLine();
    if(clientMessage.equals("playerdead")){
        server.removeUser(userName, this);
        serverMessage = userName + " has died against the boss";
        server.broadcast(serverMessage, this);
        break;
    }
    if(clientMessage.equals("shoot")){ // A player shot the boss
        mutualBoss.setHP(mutualBoss.getHP() - 10);
    }
    if(mutualBoss.getHP() <= 0){
        serverMessage = "bosskilled";
        server.broadcastToAll(serverMessage);
    }
}
```

Image 6. Loop constantly checking for messages

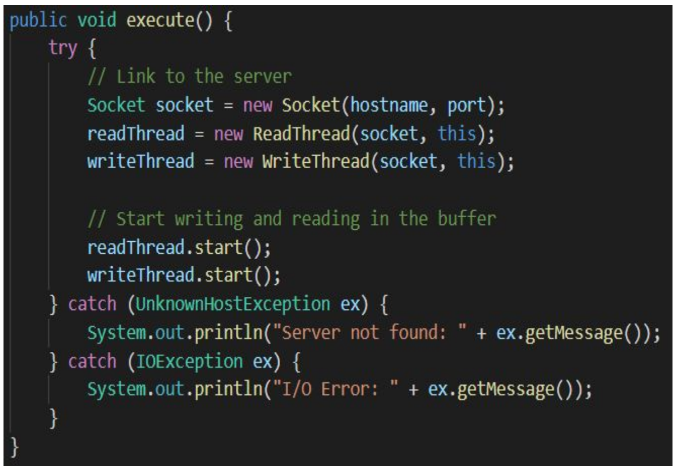
2.1.2 Client Implementation

The client is implemented by the next three classes:

GameManager Class

Starts the client program, connects to a server specified by hostname address and port number. Once the connection is made, it creates and starts ReadThread and WriteThread.

In addition to that, as its name implies, is also in charge of managing the game itself, it controls the interaction that the server and the player have, to then make certain modifications to the gameplay.

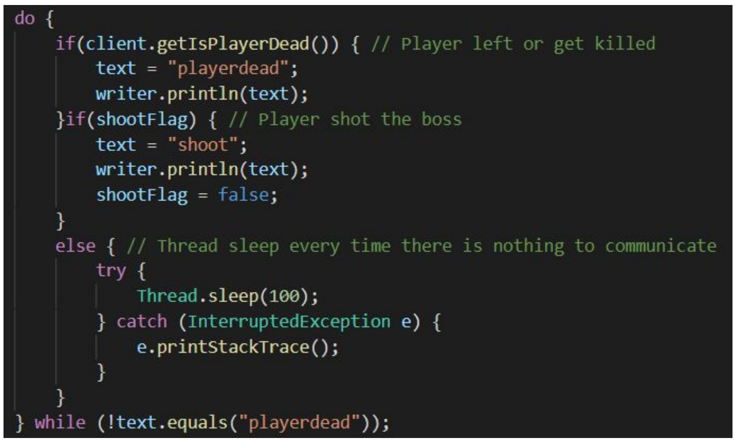


```java
public void execute() {
    try {
        // Link to the server
        Socket socket = new Socket(hostname, port);
        readThread = new ReadThread(socket, this);
        writeThread = new WriteThread(socket, this);

        // Start writing and reading in the buffer
        readThread.start();
        writeThread.start();
    } catch (UnknownHostException ex) {
        System.out.println("Server not found: " + ex.getMessage());
    } catch (IOException ex) {
        System.out.println("I/O Error: " + ex.getMessage());
    }
}
```

Image 7. Start threads for reading and writing

WriteThread Class

Is responsible for reading input from the user and sending it to the server, continuously until the user dies or left the game. Is running inside an infinite loop until the player get killed or left the game, these 2 states are equal to the server for simplicity.

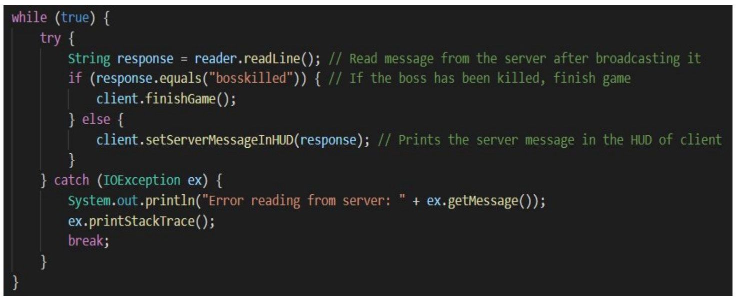


```
do {
    if(client.getIsPlayerDead()) { // Player left or get killed
        text = "playerdead";
        writer.println(text);
    }if(shootFlag) { // Player shot the boss
        text = "shoot";
        writer.println(text);
        shootFlag = false;
    }
    else { // Thread sleep every time there is nothing to communicate
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
} while (!text.equals("playerdead"));
```

Image 8. Loop constantly checking different important actions

ReadThread Class

Is responsible for reading input from the server and telling the GameManager to execute some actions depending on the message, the one is basically telling if some player died or the boss has been defeated.



```
while (true) {
    try {
        String response = reader.readLine(); // Read message from the server after broadcasting it
        if (response.equals("bosskilled")) { // If the boss has been killed, finish game
            client.finishGame();
        } else {
            client.setServerMessageInHUD(response); // Prints the server message in the HUD of client
        }
    } catch (IOException ex) {
        System.out.println("Error reading from server: " + ex.getMessage());
        ex.printStackTrace();
        break;
    }
}
```

Image 9. Loop constantly checking for messages

With these 5 classes, it is how basically the project controls the calls between the instances of the game and server.

2.2 Gameplay

On the other hand, a video game also consists of giving the user a gameplay, with which they can enjoy a complete experience. As mentioned in the beginning, it's all about combining the music, the layouts, and the narrative alongside the code to make a great video game.

For this reason, I decided to implement multiple threads to handle different things as boss shooting, rendering animation, etc. But also analyzing how I would implement my code is where the different design patterns comes into consideration. But… what are those? They are well-proven solutions for solving a specific problem or task [11].

[11] Definition taken from javaTpoint

2.2.1 Design Patterns

The ones I used in my project are catalogued as follow:

- Creational Design Pattern
  - Factory
    - Handle most of the object creation in one class
  - Singleton
    - Create one instance of an object to be called in any class of the project, for example:
      - The **HUD** is called when the server sent a message but also when it needs to be draw in panel.
      - Every time is pressed a movement key the **player** needs to move in the panel, also when there is collisions between the player and thunders and other example is when the player image needs to be draw in panel.
      - The **ImageLoader** is used in every state and object in game entity to load a different image.
- Behavioral Design Pattern
  - Observer
    - Control key interaction with the user.
  - State
    - Define a finite state machine to control behavior in any possible state of the game like: pause, menu, game over, playing, etc.

2.3 Licenses of content

For all audio and image resources used in this project, the following statement was taken into consideration:

Copyright Disclaimer under section 107 of the Copyright Act of 1976, allowance is made for “fair use” for purposes such as criticism, comment, news reporting, teaching, scholarship, education and research. Fair use is a use permitted by copyright statute that might otherwise be infringing.

3 Results

Due to the contingency for COVID-19, I was unable to run usage tests on multiple users with different computers in a same location in order to test the connection on the same local network. So what I did was asking friends to run 4 instances of the game in their PC, but taking a screenshot of the resources consumption.

3.1 Test PC Features

These are the specifications of the 3 computers:

PC1 (Laptop)

- Gigabyte Nvidia GeForce GTX 1050 ti
- Intel(R) Core(™) i7-8750H 2.2 GHz
- 16 GB RAM
- 5 Mbps Internet Speed Connection

PC2 (Desktop)

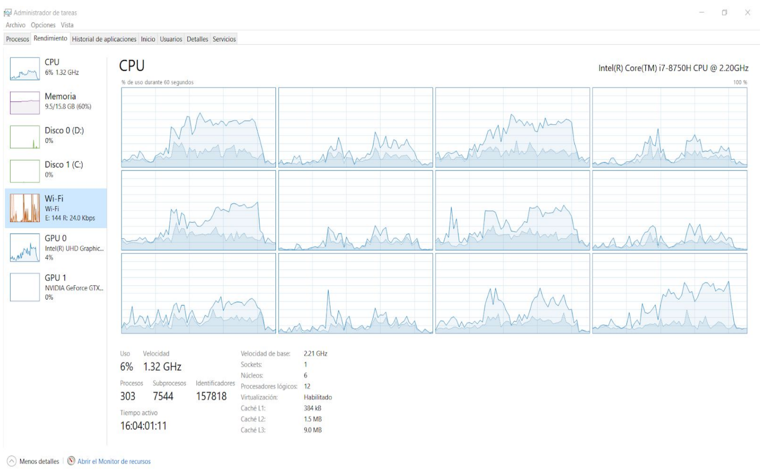
- Gigabyte Nvidia GeForce GTX 1060 ti
- AMD Ryzen 5 3600x 3.8GHz
- 8 GB RAM
- 30 Mbps Internet Speed Connection

PC3 (Laptop)

- Gigabyte Nvidia GeForce GTX 1050
- Intel(R) Core(™) i5-7300HQ 2.5GHz
- 8 GB RAM
- 3 Mbps Internet Speed Connection

3.2 Resources Consumption

Execution on PC1

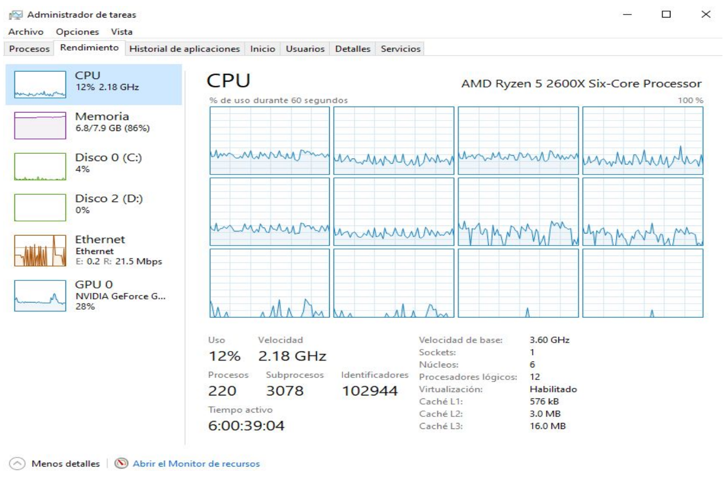

Administrador de tareas
Archivo Opciones Vista
Procesos Rendimiento Historial de aplicaciones Inicio Usuarios Detalles Servicios
CPU 6% 1.32 GHz
Memoria 9.5/15.8 GB (60%)
Disco 0 (D:) 0%
Disco 1 (C:) 0%
Wi-Fi Wi-Fi E: 144 R: 24.0 Kbps
GPU 0 Intel(R) UHD Graphic... 4%
GPU 1 NVIDIA GeForce GTX... 0%
% de uso durante 60 segundos
100 %
Menos detalles
Abrir el Monitor de recursos

CPU Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

Uso	Velocidad	
6%	1.32 GHz	
Procesos	Subprocesos	Identificadores
303	7544	157818
Tiempo activo		
16:04:01:11		

Velocidad de base:	2.21 GHz
Sockets:	1
Núcleos:	6
Procesadores lógicos:	12
Virtualización:	Habilitado
Caché L1:	384 kB
Caché L2:	1.5 MB
Caché L3:	9.0 MB

Execution on PC2

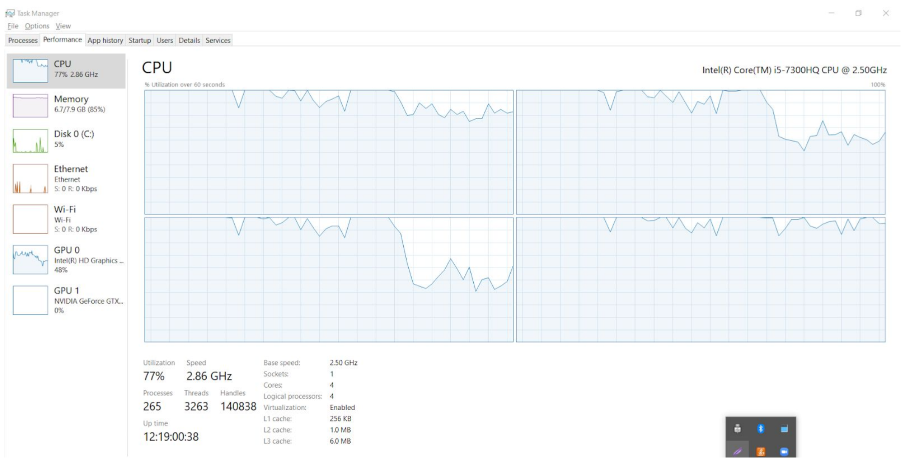

Administrador de tareas
Archivo Opciones Vista
Procesos Rendimiento Historial de aplicaciones Inicio Usuarios Detalles Servicios
CPU 12% 2.18 GHz
Memoria 6.8/7.9 GB (86%)
Disco 0 (C:) 4%
Disco 2 (D:) 0%
Ethernet Ethernet E: 0.2 R: 21.5 Mbps
GPU 0 NVIDIA GeForce G... 28%
% de uso durante 60 segundos
100 %
Menos detalles
Abrir el Monitor de recursos

CPU AMD Ryzen 5 2600X Six-Core Processor

Uso	Velocidad	
12%	2.18 GHz	
Procesos	Subprocesos	Identificadores
220	3078	102944
Tiempo activo		
6:00:39:04		

Velocidad de base:	3.60 GHz
Sockets:	1
Núcleos:	6
Procesadores lógicos:	12
Virtualización:	Habilitado
Caché L1:	576 kB
Caché L2:	3.0 MB
Caché L3:	16.0 MB

Execution on PC3


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU 77% 2.86 GHz
Memory 6.7/7.9 GB (85%)
Disk 0 (C:) 5%
Ethernet Ethernet S: 0 R: 0 Kbps
Wi-Fi Wi-Fi S: 0 R: 0 Kbps
GPU 0 Intel(R) HD Graphics ... 48%
GPU 1 NVIDIA GeForce GTX... 0%
% Utilization over 60 seconds
100%

CPU Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz

Utilization	Speed	
77%	2.86 GHz	
Processes	Threads	Handles
265	3263	140838
Up time		
12:19:00:38		

Base speed:	2.50 GHz
Sockets:	1
Cores:	4
Logical processors:	4
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	6.0 MB

Image 10, 11, 12.Comparison between the resources consumed in different PC’s

3.3 Video game being executed

To make the distinction between how my project looks like and what the *Monster Hunter World: Iceborne* video game looks like, we are going to use only 2 clients playing simultaneously.

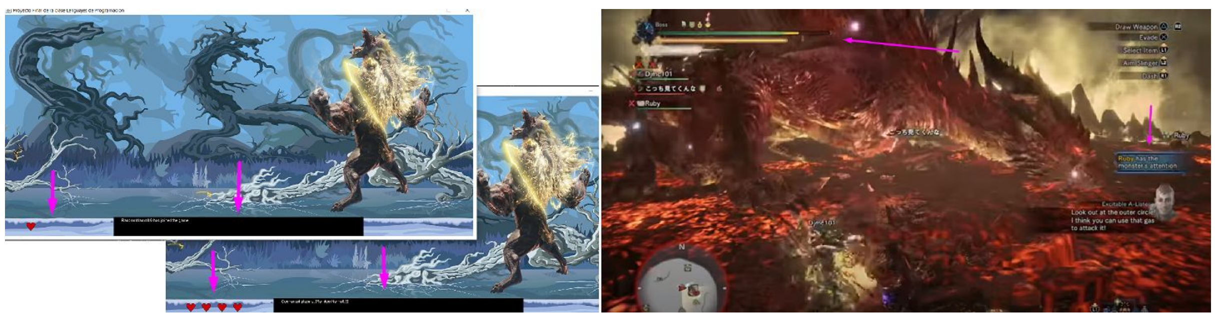


Image 13. HUD Comparison between the real video game and my project

Leaving aside that my project looks a thousand times worse graphically than the game (Always wanted to make a "joke" like this in a report), I endeavored to replicate in-game message behavior about different players in the raid (black rectangle in my game) and player-relevant usage information (remaining hp with hearts) as much as possible.

4 Conclusions

There is an important need in using multiple threads when wanting to achieve multiple tasks at the same time, as shown in the execution for my video game, where there is one thread given in the server to handle the request, and 4 threads in the game to handle the boss shooting, the writing and reading from the server and one more thread to run the animation.

The problem comes when there are many threads running parallelly, because this may cause two possible scenarios: Low performance and big resources consumption.

As shown in the previous images, even if there are 3 different PC's, the result is the same, there is a big resource consumption running 4 instances of the videogame in a single computer.

But I think finding a balance is the correct path to follow, because it is also important to apply concurrency to our algorithms to avoid losing data, especially when communicating between multiple threads.

Also, I have learn especially about the socket programming, due to the other things I used were taught in my video game class 2 years ago.

5 Set Up Instructions

5.1 Source Tree

The project has the following source structure

```
|
|  Server |
|.........|Server.java
|.........|ReadThread.java
|.........|WriteThread.java
|.........|UserThread.java
|Videogame|
|.........|  Factory  |
|.........|...........|Factory.java
|.........|  Figures  |
|.........|...........|MyGraphics.java
|.........|...........|Player.java
|.........|...........|Boss.java
|.........|...........|Thunder.java
|.........|...........|Bullet.java
|.........|    HUD    |
|.........|...........|HUD.java
|.........|ImageLoader|
|.........|...........|ImageLoader.java
|.........|    Main   |
|.........|...........|GamePanel.java
|.........|...........|RunGame.java
|.........| Management |
|.........|...........|GameManager.java
|.........|  Observer |
|.........|...........|GameKeys.java
|.........|...........|Observer.java
|.........|...........|Subject.java
|.........| Resources |
|.........|...........|Backgrounds
|.........|...........|Gifs
|.........|...........|Images
|.........|...........|Music
|.........|   Sound   |
|.........|...........|MusicPlayer.java
|.........|   State   |
|.........|...........|GameContext.java
|.........|...........|State.java
|.........|...........|GameOverState.java
|.........|...........|PauseState.java
|.........|...........|PlayState.java
|.........|...........|MenuState.java
|  README.md
|
```

5.2 Controls

The game controls are quite intuitive, since they are based on most video games. The definition of these is inside the Obsever package in the GameKeys class.

```
switch(keyPressed){
    case "D": state = "right"; break;
    case "A": state = "left"; break;
    case "W": state = "up"; break;
    case "S": state = "down"; break;
    case "P": state = "pause"; break;
    case "Intro": state = "enter"; break;
    case "Enter": state = "enter"; break;
    case "Escape": state = "escape"; break;
    case "Espacio": state = "shoot"; break;
    case "Space": state = "shoot"; break;
    default: state = "none"; break; }
```

5.3 How to run it

To run the project in visual studio code you have to follow the following steps:

1. Check the source tree code
2. Open the Server.java file inside the Server package, there is where the main method of the server is located
3. Run the class
4. Open the RunGame.java file inside the Videogame.Main package, there is where the main method to run the game is located
5. Run the class
6. Once the game window is open, press the enter key
7. Start playing

To run the project from jar files

1. Open 1 command prompt for server
2. Write java -jar Server.jar
3. Open 1 command prompt or as many as you want for the game
4. Write java -jar RunGame.jar
5. Start playing

5.3 Extra considerations

To avoid the game malfunction, avoid taking the following actions:

- Force shutdown of the server by pressing ctrl + c in command line
- Exit the game without closing the buffers and sockets by pressing the X to close in the game window or alt + f4.

6 Evidence

- GitHub Repository
- Running the project in video