Leonardo Zadkiel Mosqueda Hernandez K12524422
Leonardo-Zadkiel.Mosqueda-Hernandez@eurecom.fr or mosqueda@eurecom.fr

VSnake

Project in Software Engineering 367.035 2025W 20/10/2025

Objective

The goal of this project was to create an app using the communication between 2 devices and to use the sensors of the smartphones in the App.
So the app I decided to develop is a 2 players version of the snake game but instead of controlling the snake through buttons on the screen we would control the snake by tilting the Smartphone.

Approach of the objective

To do so I'm planning to code the app on android studio in kotlin. So first I'm going to code a snka game that can be controlled by tilting the phone either with the gyroscope or the accelerometer. Once the game works perfectly I'll implement the communications between both devices, to implement the communication between both devices I was planning to either use the Bluetooth communication as I already coded this type of communication in python for Raspberries or to use one type of wifi communication. Then when the multiplayer game is working I'll implement the use of the others sensors, I'm planning to change the aspect of the snake depending on the continent where the player is. Depending on the hour of the day the fruit of the snake will vary. If I do not use the accelerometer for the control of the snake then by shaking the phone I could implement a speed boost on the snake for some seconds, but if it is used then depending on the temperature where the user is the speed of the snake will vary a bit. And the last sensor I'm planning to use is the light sensor of the phones, depending on how illuminated the place they are is, the back ground of the game will change, going from a white to a black background.
And if I finish everything in advance I could improve the visuals of the game.

Implemented technology

So for the control of the Snake I chose to use the accelerometer instead of the gyroscope as the gyroscope can only detect the tilting of the smartphone on 1 axe while the accelerometer can detect the mouvement on 2 axes.
For the communication between devices I chose to use the Google's Nearby Connections API as I had some constrains on the other types of communication because of the brand of both phones I used.
To detect the continent on which the user is I decided to use the Maptiler API to get my position.
As I already use accelerometer to modify the speed I choose to use the Open Meteo API and the maptiler API together.

Description of the project

As planned the first part of the game I implemented was a snake game that was working with the tilt of the smartphone, to help me with the debug of this fonctionality on every menu of the game I Added some bars on the sides of the screen showing on which side the phone was tilted. At this point the game had 4 menus, the main menu to lauch the game, the pause menu that appeared when the game was paused and the game over menu when the player lost, on both of these screens the score and the timer can been seen. And the last menu was the game menu where the snake game was visible, a timer and te score were visble in real time.
Then when I tested the game with some friends I fixed a bit the detection of the tilt of the phone.
And the I started to implement the communiction with between both phones, I started with Bluetooth as after checking a bit on internet I find out that it was probably the easiest communication to implement.
So I created a menu for the Bluetooth Discovery and Added the permissions on the androidManifest but

after muliple tries and some researches I found out that the restrictions of Bluetooth increased a lot since Android 12 and so I wouldn't be able to create a communication for the game as I wasn't able to detect the other device as both phones I used were on android 12 and 14. The only way to detect the other devices was to use BLE (Bluetooth Low Energy) but BLE doesn't support the normal exchange of data, just GATT exchanges.
Then after some researches I decided to make an UDP communication between both devices with one phone being the host of the game and the other the client. Once again I implemented everything and when I arrived to the test of the exchanges of messages and the discovery wasn't working on 1 of the phones, at first I though it was again because of the version of Android but this time the problem was the phone. The 2 phones I use for test are a Redmi note 9 pro Under android 12 and a Redmi note 12 pro Under android 14 and this time the problem was releated to the Redmi note 12 as since android 13 and 14 Xiaomi severely restrict Wi-Fi Direct so the group creation, discovery and the asymmetric host/client behaviors are limited or blocked, so one phone can only see the other and only one can respond to the other one.
So I with the help of deepseek I tried multiple types of communications (UDP Broadcast, UDP Active scanning, TCP socket, WIFI P2P) thanks to code provided quickly by deepseek and after some test we discover that Xiaomi was blocking all these communications and that the only one that would probably work is the one provided by google, Google's Nearby Connections API.
Once the communication between both phones was establish I decided how the game would work, first I though about both players having their snake o their device and reciving and printing the other snake on the screen, but chat gpt and deepseek both told me that the easiest way to make a multiplayer game like the one I wanted to do was to create all the game on just the host device and send the position of the fruit and the snakes position to the client and the client would just send his movements to the host and the host would applied them to the second snake. The main problem I encounter here was the multithreading crashes, as I studied the multithreading but never really put it in practice I forgot that if there was a desincronisation between the treads the app would crash.
When all the problems of related to multithreading were solved I added all the personalization related to the sensors, te only problem is that both phones would be both in the same continent and city as the communication I establich only works Under the same network, so to simulate different positions for the users when the app starts there is a list of cities and each user can choose their position. And I find out that I wasn't able to use the lights detectors, they always detect low luminosity.

Implementation details

The code of the app can be divided in 3 parts, first the classes that constitute the logic of the game, which are:
Direction.kt
Snake.kt
NearbyGameSocket.kt
NetworkManager.kt
GameNetworkController.kt
GameNetwork.kt
SensorController.kt
SensorHolder.kt
City.kt
TemperatureManager.kt
LocationStyleManager.kt
AccelerometerDebugOverlay.kt

Then the menus visible which are:
GameView.kt
GameOverActivity.kt
PauseGameActivity.kt
WifiDiscoveryActivity.kt
CitySelectionActivity.kt
BluetoothDiscoveryActivity.kt
MainMenuActivity.kt

And finally the mainActivity.kt of the game which coordinates everything.

Now I'm going to present you some parts of the 4 main codes of the app.

First the NearbyGameSocket.kt, this class is the primary communication channel for the game. It handles hosting (the host starts advertising its presence):

```
fun startHosting() {
    Log.d("DEBUG_NEARBYGAMESOCKET_STARTHOSTING", "HOST: startAdvertising()")
    connectionsClient.startAdvertising(
        "Host",
        SERVICE_ID,
        connectionLifecycleCallback,
        AdvertisingOptions.Builder()
            .setStrategy(Strategy.P2P_POINT_TO_POINT)
            .build()
    )
}
```

, device discovery (the client looks for hosts):

```
fun startDiscovery(onFound: (String, String) -> Unit) {
    Log.d("DEBUG_NEARBYGAMESOCKET_STARTDISCOVERY_1", "CLIENT: startDiscovery()")
    connectionsClient.startDiscovery(
        SERVICE_ID,
        object : EndpointDiscoveryCallback() {
            override fun onEndpointFound(
                endpointId: String,
                info: DiscoveredEndpointInfo
            ) {
                Log.d("DEBUG_NEARBYGAMESOCKET_STARTDISCOVERY_2", "FOUND endpoint=${info.endpointName}")
                onEndpointFoundCallback(endpointId, info.endpointName)
            }

            override fun onEndpointLost(endpointId: String) {
                Log.d("DEBUG_NEARBYGAMESOCKET_STARTDISCOVERY_3", "LOST endpoint=$endpointId")
            }
        },
        DiscoveryOptions.Builder()
            .setStrategy(Strategy.P2P_POINT_TO_POINT)
            .build()
    )
}
```

, connection establishment (the client initiates a connection to a discovered hos):

```
fun connect(endpointId: String) {
    connectionsClient.requestConnection(
        "Client",
        endpointId,
        connectionLifecycleCallback
    )
}
```

, and message sending/receiving over Wi-Fi/Bluetooth:

```
fun send(message: String) {
    endpointId?.let {
        Log.d("DEBUG_NEARBYGAMESOCKET_SEND", "TX: $message")
        connectionsClient.sendPayload(
            it,
            Payload.fromBytes(message.toByteArray())
        )
    }
}
```

Secondly WifiDiscoveryActivity.kt, the page that creates the connection between both players, set ups which player will be the host and the client and launches the game:

The "HOST" button creates a NearbyGameSocket in host mode and starts advertising. It waits for a client to connect.

```
btnHost.setOnClickListener {
    isHost = true

    NetworkManager.socket = NearbyGameSocket(
        context = this,
        isHost = true,
        onEndpointFoundCallback = { _, _ -> },
        onConnected = {
            runOnUiThread {
                Toast.makeText(this, "Client connected", Toast.LENGTH_SHORT).show()
                btnPlay.isEnabled = true
            }
        },
        onDisconnected = {
            Log.d("DEBUG_WIFIMENU_ONDISCONNECTED", "Disconnected")
        },
        onMessageReceived = { msg ->
            Log.d("DEBUG_WIFIMENU_ONMESSAGERECEIVED", "Response=$msg")
            if (msg == "START") {
                runOnUiThread {
                    Log.d("DEBUG_WIFIMENU_ONCMESSAGERECEIVED_START", "going to launch the game")
                    launchGame(isHost = false)
                }
            }
        }
    )

    NetworkManager.socket.startHosting()

    items.clear()
    items.add("Waiting for player…")
    adapter.notifyDataSetChanged()
}
```

The "CONNECT" button creates a NearbyGameSocket in client mode and starts discovering hosts. Discovered hosts appear in a ListView. When the user selects one, the client attempts to connect.

```
btnConnect.setOnClickListener {
    isHost = false

    NetworkManager.socket = NearbyGameSocket(
        context = this,
        isHost = false,
        onEndpointFoundCallback = { endpointId, name ->
            runOnUiThread {
                endpoints[name] = endpointId
                items.add(name)
                adapter.notifyDataSetChanged()
            }
        },
        onConnected = {
            Log.d("DEBUG_WIFIMENU_MANAGER_ONCONNECTED", "Connected to host")
        },
        onDisconnected = {
            Log.d("DEBUG_WIFIMENU_MANAGER_ONDISCONNECTED", "Disconnected")
        },
        onMessageReceived = { }
    )

    if (!hasNearbyPermissions()) {
        ActivityCompat.requestPermissions(this, REQUIRED_PERMISSIONS, REQ_NEARBY)
        return@setOnClickListener
    }

    NetworkManager.socket.startDiscovery { _, _ -> }
}
```

Then there is the Gameview.kt, the heart of the game. It's a custom SurfaceView that handles: the game loop, drawing, collision detection, and network state synchronization:

Game Loop: A continuous loop that runs on a separate thread. In each iteration:

- If the device is the host, it calls update() to advance the game logic.
- It calls draw() to render everything on the screen.
- It sleeps for a duration determined by the snake's speed multiplier.

```
override fun run() {
    // thread loop, monitored by isPlaying
    Log.d("SNAKE_GAMEVIEW_RUN", "Game loop started")
    while (isPlaying.get()) {
        if (isHost) {
            update()
        }
        draw()

        try {
            val baseDelay = 200L
            val delay = (baseDelay / hostSpeedMultiplier).toLong()
            Thread.sleep(delay)
        } catch (_: InterruptedException) {
            // The thread was interrupted during a pause() → we exit cleanly
            break
        }
    }
}
```

Game Logic: Called only by the host. It moves both snakes, checks for food consumption (increasing score and spawning new food), and most importantly, checks for game-over conditions:

- Collision with the walls.
- Collision with the snake's own body.
- Collision with the other snake (checkSnakeVsSnake()):

```
private fun update() {
    if (!gameInitialized) {
        Log.d("SNAKE_GAMEVIEW_UPDATE_1", "Update OK | host=${hostSnake.getHead()} client=${clientSnake.getHead()}")
        return
    }

    hostSnake.move()
    clientSnake.move()

    ticks++

    if (ticks < 2) return

    if (hostSnake.getHead() == food) {
        hostSnake.grow()
        spawnFood()
        hostScore++
    }

    if (clientSnake.getHead() == food) {
        clientSnake.grow()
        spawnFood()
        clientScore++
    }
    if (isHost) {
        val vsResult = checkSnakeVsSnake()
        if (vsResult != null) {
            val elapsed = (System.currentTimeMillis() - startTime) / 1000

            onGameOver?.invoke(
                hostScore,
                clientScore,
                elapsed,
                vsResult
            )

            networkController?.onGameOver(
                vsResult,
                hostScore,
                clientScore,
                elapsed
            )

            stopGameThread()
            return
        }

    }

    if (checkCollision(hostSnake) || checkCollision(clientSnake)) {
        Log.d("SNAKE_GAMEVIEW_UPDATE_2", "Collision detected → game over")

        val elapsed = (System.currentTimeMillis() - startTime) / 1000
        val loser =
            if (checkCollision(hostSnake)) GameOverResult.HOST_LOST
            else GameOverResult.CLIENT_LOST
            onGameOver?.invoke(
                hostScore,
                clientScore,
                elapsed,
                loser
            )

            networkController?.onGameOver(
                loser,
                hostScore,
                clientScore,
                elapsed
            )


            stopGameThread()
            sensorController.unregister()
            return
        }
    }
```

Rendering: Locks the canvas and draws all game elements:

- The background color, which changes based on ambient light (getBackgroundColorByLight()).
- The gray walls around the play area.
- The two snakes, each with their own color.
- The food, whose color changes based on the time of day (getFruitColorByTime()).
- The HUD displaying scores and time.

```
private fun draw() {
    synchronized(stateLock) {
        if (!holder.surface.isValid) return
        val canvas = holder.lockCanvas()
        canvas.drawColor(getBackgroundColorByLight())
        // walls
        paint.color = Color.DKGRAY
        for (x in 0 until gridCountX) {
            for (y in 0 until gridCountY) {
                if (x == 0 || x == gridCountX - 1 || y == 0 || y == gridCountY - 1) {
                    canvas.drawRect(
                        offsetX + x * gridSize.toFloat(),
                        offsetY + y * gridSize.toFloat(),
                        offsetX + (x + 1) * gridSize.toFloat(),
                        offsetY + (y + 1) * gridSize.toFloat(),
                        paint
                    )
                }
            }
        }
        // snake
        // host snake
        paint.color = hostColor
        for (p in hostSnake.getBody()) {
            canvas.drawRect(
                offsetX + p.x * gridSize,
                offsetY + p.y * gridSize,
                offsetX + (p.x + 1) * gridSize,
                offsetY + (p.y + 1) * gridSize,
                paint
            )
        }
        // client snake
        paint.color = clientColor
        for (p in clientSnake.getBody()) {
            canvas.drawRect(
                offsetX + p.x * gridSize,
                offsetY + p.y * gridSize,
                offsetX + (p.x + 1) * gridSize,
                offsetY + (p.y + 1) * gridSize,
                paint
            )
        }

        // food
        paint.color = getFruitColorByTime()
        canvas.drawRect(
            offsetX + food.x * gridSize.toFloat(),
            offsetY + food.y * gridSize.toFloat(),
            offsetX + (food.x + 1) * gridSize.toFloat(),
            offsetY + (food.y + 1) * gridSize.toFloat(),
            paint
        )
        val elapsed = (System.currentTimeMillis() - startTime) / 1000
        val minutes = elapsed / 60
        val seconds = elapsed % 60
        val timeText = String.format("%02d:%02d", minutes, seconds)
        canvas.drawText("Host: $hostScore", 20f, 60f, textPaint)
        canvas.drawText("Client: $clientScore", 20f, 120f, textPaint)
        canvas.drawText("Time: $timeText", 20f, 180f, textPaint)
        canvas.drawText("Host Speed x$hostSpeedMultiplier",20f,240f,textPaint)
        holder.unlockCanvasAndPost(canvas)
    }
}
```

Networking Role:

- It acts as the authoritative state holder on the host.
- setHostMode(): Starts a background thread that periodically calls buildGameState() and sends the full game state to the client.

```
fun setHostMode(sendState: (String) -> Unit) {
    isHost = true
    networkRunning.set(true)

    networkThread = Thread {
        try {
            while (networkRunning.get()) {
                Thread.sleep(50)
                sendState(buildGameState())
            }
        } catch (e: InterruptedException) {
            Log.d("SNAKE_GAMEVIEW_SETHOSTMODE", "Network thread stopped cleanly")
        }
    }

    networkThread?.start()
}
```

- applyRemoteState(): Parses a state string received from the host and updates the local game (snake positions, food, colors, speeds). This is how the client stays synchronized.

```
fun applyRemoteState(state: String) {
    synchronized(stateLock) {
        val map = state.split(";")
            .drop(1)
            .associate {
                val (k, v) = it.split("=")
                k to v
            }

        hostSpeedMultiplier = map["hostSpeed"]?.toFloat() ?: hostSpeedMultiplier
        clientSpeedMultiplier = map["clientSpeed"]?.toFloat() ?: clientSpeedMultiplier

        hostColor = map["hostColor"]?.toInt() ?: hostColor
        clientColor = map["clientColor"]?.toInt() ?: clientColor


        map["host"]?.let {
            hostSnake.setBody(parseSnake(it))
        }

        map["client"]?.let {
            val body = parseSnake(it).distinct()
            if (body.size >= 3) {
                clientSnake.setBody(body)
            }
        }

        map["food"]?.let {
            val (x, y) = it.split(",")
            food = Point(x.toInt(), y.toInt())
        }
    }
}
```

- applyClientInput(): Allows the host to apply a direction change received from the client to the clientSnake.

```
fun applyClientInput(input: String) {
    val dir = try {
        Direction.valueOf(input)
    } catch (e: Exception) {
        return
    }

    clientSnake.setDirection(dir)
}
```

To ensure that when one thread is reading/writing the game state, another thread cannot access it simultaneously there are synchronized blocks, it is visible at the start of Draw(), applyRemoteState() and buildGameState(), the message sent by the host to the client to share the data of the game (visible on setHostMode().

```
fun buildGameState(): String {
    synchronized(stateLock) {
        return "STATE;" +
                "host=${hostSnake.serialize()};" +
                "client=${clientSnake.serialize()};" +
                "food=${food.x},${food.y};" +
                "hostScore=$hostScore;"+
                "clientScore=$clientScore;" +
                "hostSpeed=$hostSpeedMultiplier;" +
                "clientSpeed=$clientSpeedMultiplier;" +
                "hostColor=$hostColor;" +
                "clientColor=$clientColor"
    }
}
```

And finally the main code of the game, the mainActivity.kt, it acts as the central coordinator, bringing together the game view, the network socket, and the location-based features. At initialization:

- It receives the selected City from the intent and sent it the client city to the host and determines the player's continent and their snake color.

```
val cityName = intent.getStringExtra("selectedCity")
if (cityName == null) {
    Log.e("MAIN_ACTIVITY", "selectedCity is NULL")
    finish()
    return
}
val selectedCity = City.valueOf(cityName)
```

```
if (!isHost) {
    Log.d("DEBUG_MAINACTIVITY_CITY", "CITY send: ${selectedCity.name}")
    socket.send("CITY;name=${selectedCity.name}")
}
```

```
private fun computeSpeedMultiplier(temp: Double): Float {
    return when {
        temp <= -20 -> 0.75f
        temp in -20.0..20.0 -> {
            0.75f + ((temp + 20) / 40.0 * 0.25).toFloat()
        }
        temp in 21.0..60.0 -> {
            1.0f + ((temp - 20) / 40.0 * 0.25).toFloat()
        }
        else -> 1.25f
    }
}
```

```
private fun handleClientCity(city: City) {
    clientCity = city
    maybeComputeGameParams()
}

private fun maybeComputeGameParams() {
    if (hostCity == null || clientCity == null) return

    computeCityParams(hostCity!!, true)
    computeCityParams(clientCity!!, false)
}
```

```
private fun continentToColor(continent: String): Int {
    val color = when (continent) {
        "EUROPE" -> Color.BLUE
        "AFRICA" -> Color.YELLOW
        "ASIA" -> Color.RED
        "NORTH_AMERICA" -> Color.GREEN
        "SOUTH_AMERICA" -> Color.MAGENTA
        "OCEANIA" -> Color.CYAN
        else -> Color.WHITE
    }

    Log.d("SNAKE_COLOR", "Snake color set for continent=$continent")
    return color
}
```

```
private fun computeCityParams(city: City, isHostCity: Boolean) {
    val tempManager = TemperatureManager()
    val locationStyleManager = LocationStyleManager()

    tempManager.fetchTemperature(city.lat, city.lon) { temp ->
        temp ?: return@fetchTemperature

        val speed = computeSpeedMultiplier(temp)

        runOnUiThread {
            if (isHostCity) {
                gameView.setHostSpeedMultiplier(speed)
            } else {
                gameView.setClientSpeedMultiplier(speed)
            }
        }
    }

    locationStyleManager.fetchContinent(city.lat, city.lon) { continent ->
        continent ?: return@fetchContinent

        val color = continentToColor(continent)
        Log.d("SNAKE_MAINACTIVITY_SNAKECONTINENT", "continent=$continent")
        runOnUiThread {
            if (isHostCity) {
                gameView.setHostColor(color)
            } else {
                gameView.setClientColor(color)
            }
        }
    }
}
```

- It sets up the GameView and its GameNetworkController to handle network events.

```
gameView.networkController = object : GameNetworkController {

    override fun onRemoteInput(input: String) {
        socket.send("INPUT;$input")
    }

    override fun onGameState(state: String) {}

    override fun onGameOver(
        loser: GameView.GameOverResult,
        hostScore: Int,
        clientScore: Int,
        time: Long
    ) {
        socket.send(
            "GAMEOVER;" +
                    "loser=${loser.name};" +
                    "hostScore=$hostScore;" +
                    "clientScore=$clientScore;" +
                    "time=$time"
        )
        handleGameOverLocal(hostScore, clientScore, time, loser.name)
    }
}
```

- It gets the NearbyGameSocket instance from NetworkManager. And it manages all the messages exchanged between the host and the client.

```
socket = NetworkManager.socket

socket.setOnMessageReceived { msg ->
    runOnUiThread {
        when {
            msg.startsWith("STATE") -> {
                gameView.applyRemoteState(msg)
            }

            msg == "START" -> {
                gameView.startFromNetwork()
            }

            msg == "PAUSE" -> {
                handlePause()
            }

            msg == "RESUME" -> {
                if (!isFinishing) {
                    Log.d("DEBUG_MAINACTIVITY_RESUME", "received Resume from the other player")
                    gameView.resume()
                }
            }

            msg.startsWith("INPUT") -> {
                if (isHost) {
                    val dir = msg.split(";")[1]
                    gameView.applyClientInput(dir)
                }
            }
            msg.startsWith("INPUT") -> {
                if (isHost) {
                    val dir = msg.split(";")[1]
                    gameView.applyClientInput(dir)
                }
            }

            msg.startsWith("GAMEOVER") -> {
                Log.d("DEBUG_MAINACTIVITY_GAMEOVER", "Collision received → game over")
                val parts = msg.split(";")
                val map = parts.drop(1).associate {
                    val (k, v) = it.split("=")
                    k to v
                }

                val loser = map["loser"] ?: "UNKNOWN"
                val hostScore = map["hostScore"]?.toInt() ?: 0
                val clientScore = map["clientScore"]?.toInt() ?: 0
                val time = map["time"]?.toLong() ?: 0

                handleGameOverLocal(hostScore, clientScore, time, loser)
            }

            msg.startsWith("CITY") -> {
                if (isHost) {
                    val cityName = msg.split("=")[1]
                    clientCity = City.valueOf(cityName)
                    Log.d("DEBUG_MAINACTIVITY_CITY", "CITY recieved: ${clientCity}")
                    maybeComputeGameParams()
                }
            }
        }
    }
}
```

- It determines if the device is the Host or client.

```
if (isHost) {
    gameView.setHostMode { state ->
        socket.send(state)
    }
}
else {
    gameView.setClientMode()
}
gameView.startFromNetwork()
```

Example using the system

- When the app start you will see a table with multiple cities you have to choose your position by clicking on one,.
- Then the Main menu will open you will be see 2 buttons and the name of the game, you click on the WIFI button.
- This will open the Wifi Discovery Menu, on it there are 4 buttons, 2 to choose your roles, either as host or client, when you click on host you create a group and you wait that another player joins your group but if you click on client you start the Discovery and on the screen will be visible the different hosts. And the 2 other buttons are cancel and play, cancel can be used to cancel the creation of the group or the Discovery and play is a button locked until the host receive an invitation from a client. So when you arrive on this screen one of the players click on host and the other on client, the client will then see a host on his screen, he has to click on it and then the client will send a demand of game to the host (it can take some seconds), thus the play button of the host will activate and the host can choose to either click on cancel or to click on play, if he clicks on play a game will launch on both games at the same time but if he clicks on cancel his group will be erased.
- Once the game start the players can play freely. On the screen there is just 1 button, the pause button, if one player clicks on it, both players are sent to the pause menu where they can see the duration of the game and the score, from this menu they can leave the Game and go to the main screen or to restart the game where they were, the first layer that clicks on resume will be sent to the game menu but will not be able to play until the other player also clicks on the resume button.
- Then if one of the players go on a wall or that both players touch themselves the party ends and both players arrive on the game over menu, on this menu they can see if any of both players won or if it is a draw.
- Then from the game over menu there is a replay button not implemented and a main menu button, when you go on the main menu it cuts the communication between both players and so from this menu the players can restart another game going on the WIFI menu.

Conclusion

As a conclusion I would say that the goal has been reached as it is possible to play agains another player through a wifi connection and thanks to the movement of the smartphone, but it

the app to be complete would still need some improvements susch as a replay feature in the gameOver menu and aesthetically updates as because of the problems I met I wasn’t able to implement some of the ideas I had, specially to differentiate both snakes.