**Instituto Politécnico Nacional**

**Escuela Superior De Computo**

**Desarrollo De Aplicaciones Móviles Nativas**

# Práctica 6

# Manejo de sensores del dispositivo móvil


Nombre Del Alumno:

García Quiroz Gustavo Ivan | 2022630278

Grupo: 7CV3

Nombre Del Profesor: Hurtado Avilés Gabriel


Fecha De Entrega: 12/05/2025

# Índice

# 1 Introducción

En esta práctica se desarrolló la implementación de sensores y comunicación Bluetooth en una aplicación de juego de puzzle previamente creada. El objetivo principal fue integrar tecnologías de hardware disponibles en dispositivos móviles para mejorar la experiencia de usuario y ofrecer nuevas funcionalidades.

Se han implementado dos características principales:

1. **Sensor de luz ambiental**: Para adaptar la interfaz del juego según las condiciones de luz del entorno.
2. **Comunicación Bluetooth**: Para permitir el modo multijugador, facilitando la interacción entre dos dispositivos.

La aplicación está desarrollada en Kotlin siguiendo los estándares de desarrollo modernos para Android y utiliza una arquitectura modular para facilitar el mantenimiento y las futuras extensiones.

# 2 Desarrollo

## 2.1 Implementación de Sensores en Aplicaciones Existentes

### 2.1.1 Implementación del Sensor de Luz Ambiental

El sensor de luz ambiental se ha integrado para adaptar la experiencia del juego a las condiciones lumínicas del entorno. Esta implementación incluye:

- **Detección automática del nivel de luz**:

Se implementó un LightSensorManager que monitorea continuamente el nivel de luz ambiental. En esta clase de java se establecieron umbrales para determinar condiciones de luz baja, media y alta.

```kotlin
package com.example.puzzle.util

import android.content.Context
import android.hardware.Sensor
import android.hardware.SensorEvent
import android.hardware.SensorEventListener
import android.hardware.SensorManager
import android.util.Log
import androidx.lifecycle.DefaultLifecycleObserver
import androidx.lifecycle.LifecycleOwner

/**
 * Manages the device's light sensor and provides brightness adaptation functionality
 */
class LightSensorManager(private val context: Context) : SensorEventListener, DefaultLifecycleObserver {

    companion object {
        private const val TAG = "LightSensorManager"

        // Threshold values for light levels in lux
        private const val DARK_LUX_THRESHOLD = 10f
        private const val DIM_LUX_THRESHOLD = 50f
        private const val NORMAL_LUX_THRESHOLD = 200f
        private const val BRIGHT_LUX_THRESHOLD = 1000f

        // Hysteresis value to prevent rapid toggling
        private const val LUX_HYSTERESIS = 5f
```

```kotlin
        // Singleton instance
        @Volatile
        private var INSTANCE: LightSensorManager? = null

        fun getInstance(context: Context): LightSensorManager {
            return INSTANCE ?: synchronized( lock: this) {
                INSTANCE ?: LightSensorManager(context.applicationContext).also { INSTANCE = it }
            }
        }
    }

    private var sensorManager: SensorManager? = null
    private var lightSensor: Sensor? = null
    private var isRegistered = false

    // Current light level category
    enum class LightLevel { DARK, DIM, NORMAL, BRIGHT, VERY_BRIGHT }

    // Callbacks
    private var onLightLevelChanged: ((LightLevel) -> Unit)? = null
    private var currentLightLevel: LightLevel = LightLevel.NORMAL
    private var lastLuxReading = -1f

    private var autoModeEnabled = false
```

```
init {
    // Get the system's sensor service
    sensorManager = context.getSystemService(Context.SENSOR_SERVICE) as SensorManager?
    // Get the light sensor
    lightSensor = sensorManager?.getDefaultSensor(Sensor.TYPE_LIGHT)

    if (lightSensor == null) {
        Log.w(TAG, msg: "Light sensor not available on this device")
    }
}

/**
 * Start monitoring light changes
 */
fun startMonitoring() {
    if (lightSensor != null && !isRegistered) {
        sensorManager?.registerListener(
            listener: this,
            lightSensor,
            SensorManager.SENSOR_DELAY_UI
        )
        isRegistered = true
        Log.d(TAG, msg: "Light sensor monitoring started")
    }
}
```

```kotlin
/**
 * Stop monitoring light changes
 */
fun stopMonitoring() {
    if (isRegistered) {
        sensorManager?.unregisterListener( listener: this)
        isRegistered = false
        Log.d(TAG,  msg: "Light sensor monitoring stopped")
    }
}

/**
 * Set a callback to receive light level change events
 */
fun setOnLightLevelChangedListener(listener: (LightLevel) -> Unit) {
    onLightLevelChanged = listener
}

/**
 * Enable or disable automatic mode switching based on ambient light
 */
fun setAutoModeEnabled(enabled: Boolean) {
    autoModeEnabled = enabled

    if (enabled) {
        startMonitoring()
    } else {
        stopMonitoring()
    }
}
```

```
    fun isAutoModeEnabled(): Boolean = autoModeEnabled

    /**
     * Get the current light level category
     */
    fun getCurrentLightLevel(): LightLevel = currentLightLevel

    /**
     * Get the last measured light value in lux
     */
    fun getLastLuxReading(): Float = lastLuxReading

    // SensorEventListener implementation
    override fun onSensorChanged(event: SensorEvent) {
        if (event.sensor.type == Sensor.TYPE_LIGHT) {
            val luxValue = event.values[0]
            lastLuxReading = luxValue

            // Determine the new light level
            val newLightLevel = when {
                luxValue < DARK_LUX_THRESHOLD -> LightLevel.DARK
                luxValue < DIM_LUX_THRESHOLD -> LightLevel.DIM
                luxValue < NORMAL_LUX_THRESHOLD -> LightLevel.NORMAL
                luxValue < BRIGHT_LUX_THRESHOLD -> LightLevel.BRIGHT
                else -> LightLevel.VERY_BRIGHT
            }
```

```
            // Only notify if the light level category has changed
            if (currentLightLevel != newLightLevel) {
                currentLightLevel = newLightLevel
                onLightLevelChanged?.invoke(newLightLevel)
                Log.d(TAG,  msg: "Light level changed to: $newLightLevel (${luxValue} lux)")
            }
        }
    }

    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
        // Not needed for this implementation, but required by the interface
    }

    // DefaultLifecycleObserver implementation - automatically manage sensor registration
    override fun onResume(owner: LifecycleOwner) {
        if (autoModeEnabled) {
            startMonitoring()
        }
    }

    override fun onPause(owner: LifecycleOwner) {
        stopMonitoring()
    }
}
```

*Figura 1 Clase de java LightSensorManager.java*

- **Ajuste dinámico de la interfaz**:

En condiciones de luz baja, la aplicación cambia automáticamente al tema oscuro. En condiciones de luz alta, se aumenta el contraste de los elementos visuales.

- **Configuración personalizada**:

El usuario puede ajustar manualmente los umbrales de luz para personalizar cuándo se realizan los cambios.

Se incluye la opción de desactivar esta función si el usuario lo prefiere.

### 2.1.2 Implementación de Comunicación Bluetooth

Se desarrolló un modo multijugador basado en Bluetooth que permite a dos jugadores competir en tiempo real:

- **Arquitectura de comunicación**:

Se implementó una capa de abstracción BluetoothManager para gestionar la conexión entre dispositivos.

```kotlin
package com.example.puzzle.bluetooth

import android.Manifest
import android.bluetooth.BluetoothAdapter
import android.bluetooth.BluetoothDevice
import android.bluetooth.BluetoothManager
import android.bluetooth.BluetoothServerSocket
import android.bluetooth.BluetoothSocket
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.content.IntentFilter
import android.content.pm.PackageManager
import android.os.Build
import android.util.Log
import androidx.core.app.ActivityCompat
import java.io.IOException
import java.io.InputStream
import java.io.OutputStream
import java.util.UUID

/**
 * Esta clase gestiona las comunicaciones Bluetooth para el modo multijugador.
 */
class BluetoothManager(private val context: Context) {

    companion object {
        private const val TAG = "BluetoothManager"
        private const val APP_NAME = "PuzzleGame"
        private val MY_UUID = UUID.fromString( name: "8989063a-c9af-463a-b3f1-f21d9b2b827b")
```

```
    // Constantes para los estados de conexión
    const val STATE_NONE = 0
    const val STATE_LISTEN = 1
    const val STATE_CONNECTING = 2
    const val STATE_CONNECTED = 3

    // Constantes para los tipos de mensajes
    const val MESSAGE_STATE_CHANGE = 1
    const val MESSAGE_READ = 2
    const val MESSAGE_WRITE = 3
    const val MESSAGE_DEVICE_NAME = 4
    const val MESSAGE_TOAST = 5

    // Constantes para los tipos de juego
    const val GAME_MODE_COOPERATIVE = 0
    const val GAME_MODE_COMPETITIVE = 1
}

// BluetoothAdapter representa el adaptador Bluetooth del dispositivo
private var bluetoothAdapter: BluetoothAdapter? = null

// Estado actual de la conexión
private var state = STATE_NONE
```

```
// Hilos para las diferentes partes de la conexión
private var serverThread: ServerThread? = null
private var clientThread: ClientThread? = null
private var connectedThread: ConnectedThread? = null

// Listener para eventos de Bluetooth
private var listener: BluetoothEventListener? = null

// Modo de juego actual
private var gameMode = GAME_MODE_COOPERATIVE

init {
    val bluetoothManager = context.getSystemService(Context.BLUETOOTH_SERVICE) as BluetoothManager
    bluetoothAdapter = bluetoothManager.adapter
}

/**
 * Establece el listener para eventos de Bluetooth
 */
fun setListener(listener: BluetoothEventListener) {
    this.listener = listener
}

/**
 * Configura el modo de juego (cooperativo o competitivo)
 */
fun setGameMode(mode: Int) {
    gameMode = mode
}
```

```
fun isBluetoothSupported(): Boolean {
    return bluetoothAdapter != null
}
  /**
 * Verifica si el Bluetooth está habilitado
 */
fun isBluetoothEnabled(): Boolean {
    try {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
            if (ActivityCompat.checkSelfPermission(context, Manifest.permission.BLUETOOTH_CONNECT) != PackageManager.PERMISSION_GRANTED) {
                listener?.onBluetoothPermissionRequired()
                return false
            }
        }
        return bluetoothAdapter?.isEnabled == true
    } catch (e: SecurityException) {
        Log.e(TAG, msg: "Error de permisos al verificar el estado de Bluetooth", e)
        listener?.onBluetoothPermissionRequired()
        return false
    }
}
```

```
fun startServer() {
    // Verificar permisos de Bluetooth antes de proceder
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S &&
        ActivityCompat.checkSelfPermission(context, Manifest.permission.BLUETOOTH_CONNECT) != PackageManager.PERMISSION_GRANTED) {
        listener?.onBluetoothPermissionRequired()
        return
    }

    // Cancelar cualquier hilo intentando establecer una conexión
    if (clientThread != null) {
        clientThread?.cancel()
        clientThread = null
    }

    // Cancelar cualquier hilo actualmente conectado
    if (connectedThread != null) {
        connectedThread?.cancel()
        connectedThread = null
    }

    // Iniciar el hilo de servidor
    if (serverThread == null) {
        serverThread = ServerThread()
        serverThread?.start()
    }

    setState(STATE_LISTEN)
}
```

```
    /**
     * Interfaz para escuchar eventos de Bluetooth
     */
    interface BluetoothEventListener {
        fun onConnectionStateChanged(state: Int)
        fun onDeviceConnected(deviceName: String)
        fun onConnectionFailed()
        fun onConnectionLost()
        fun onMessageReceived(buffer: ByteArray)
        fun onMessageSent(buffer: ByteArray)
        fun onBluetoothPermissionRequired()
    }
}
```

*Figura 2 Clase de java BluetoothManager.java*

Se creó un GameSyncManager para sincronizar el estado del juego entre los dispositivos.

```
// Estados de la partida
private var isRemotePlayerReady = false
private var isLocalPlayerReady = false
private var isGameStarted = false

// Handler para procesar mensajes en el hilo principal
private val handler = object : Handler(Looper.getMainLooper()) {
    override fun handleMessage(msg: Message) {
        when (msg.what) {
            BluetoothManager.MESSAGE_STATE_CHANGE -> {
                when (msg.arg1) {
                    BluetoothManager.STATE_CONNECTED -> {
                        resetGameState()
                        onConnectionEventListener?.onConnected(msg.obj as String)
                    }
                    BluetoothManager.STATE_CONNECTING -> {
                        onConnectionEventListener?.onConnecting()
                    }
                    BluetoothManager.STATE_LISTEN, BluetoothManager.STATE_NONE -> {
                        onConnectionEventListener?.onDisconnected()
                    }
                }
            }
            BluetoothManager.MESSAGE_DEVICE_NAME -> {
                onConnectionEventListener?.onConnected(msg.obj as String)
            }
            BluetoothManager.MESSAGE_TOAST -> {
                onConnectionEventListener?.onError(msg.obj as String)
            }
        }
    }
```

```
class GameSyncManager(
    private val context: Context,
    private var gameMode: Int = BluetoothManager.GAME_MODE_COOPERATIVE
) : BluetoothManager.BluetoothEventListener {
    companion object {
        private const val TAG = "GameSyncManager"

        // Tipos de mensajes que se pueden enviar
        private const val MSG_TYPE_MOVE = "move"
        private const val MSG_TYPE_GAME_STATE = "gameState"
        private const val MSG_TYPE_CHAT = "chat"
        private const val MSG_TYPE_READY = "ready"
        private const val MSG_TYPE_START = "start"
        private const val MSG_TYPE_GAME_OVER = "gameOver"
    }

    // Instancia de BluetoothManager
    private val bluetoothManager = BluetoothManager(context)
    private val gson = Gson()

    // Listeners para eventos del juego
    private var onGameEventListener: OnGameEventListener? = null
    private var onConnectionEventListener: OnConnectionEventListener? = null

    // Estado del juego remoto (para modo competitivo)
    private var remoteGameState: GameState? = null
```

```
package com.example.puzzle.bluetooth

import android.content.Context
import android.os.Handler
import android.os.Looper
import android.os.Message
import android.util.Log
import com.example.puzzle.models.GameBoard
import com.example.puzzle.models.GameState
import com.example.puzzle.models.TileState
import com.google.gson.Gson
import org.json.JSONObject
```

```
    /**
     * Interfaz para eventos de juego multijugador
     */
    interface OnGameEventListener {
        fun onRemoteMove(row: Int, col: Int, result: Boolean)
        fun onRemoteStateUpdated(gameState: GameState)
        fun onChatMessageReceived(text: String)
        fun onRemotePlayerReady(ready: Boolean)
        fun onBothPlayersReady()
        fun onGameStarted(initialState: GameState)
        fun onRemoteGameOver(win: Boolean, score: Int)
    }

    /**
     * Interfaz para eventos de conexión
     */
    interface OnConnectionEventListener {
        fun onConnecting()
        fun onConnected(deviceName: String)
        fun onDisconnected()
        fun onError(message: String)
        fun onBluetoothPermissionRequired()
    }
}
```

*Figura 3 Clase de java GameSyncManager.java*

- **Flujo de juego multijugador**:
    - Un dispositivo actúa como anfitrión y el otro como invitado.
    - El estado del juego (posición de las piezas, tiempo, puntuación) se sincroniza en tiempo real.
    - Se implementó un protocolo de comunicación eficiente para minimizar la latencia.
- **Interfaz de usuario para multijugador**:

Se diseñó una pantalla dedicada para la búsqueda y conexión de dispositivos cercanos. Se incluyeron indicadores visuales del estado de la conexión y del progreso del oponente.

### 2.1.3 Implementación de Temas Personalizados

Se desarrollaron dos temas principales conforme a los requisitos:

1. **Tema Guinda (IPN)**: Utiliza el color guinda como color principal. Incorpora elementos visuales inspirados en la identidad del IPN.
2. **Tema Azul (ESCOM)**: Utiliza el color azul como color principal. Incorpora elementos visuales inspirados en la identidad de la ESCOM.
3. **Adaptación automática**: La aplicación detecta y se adapta al modo del sistema (claro/oscuro). Se implementaron variantes nocturnas de ambos temas.

# 3 Diagramas

## 3.1 Diagrama de Clases

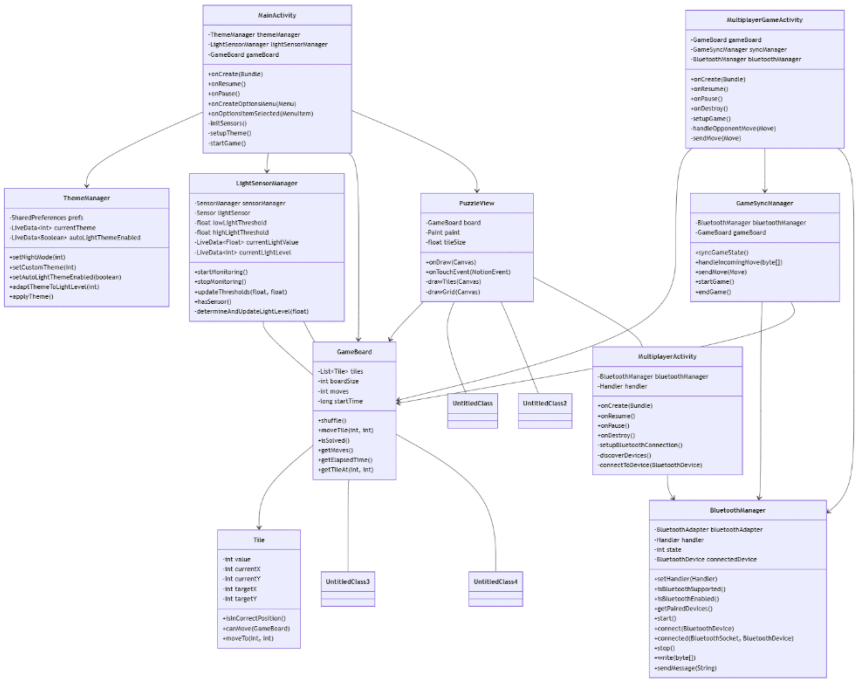



*Figura 4 Diagrama de Clases*

## 3.2 Diagramas de Secuencia

- Diagramas de Secuencia - Comunicación Bluetooth

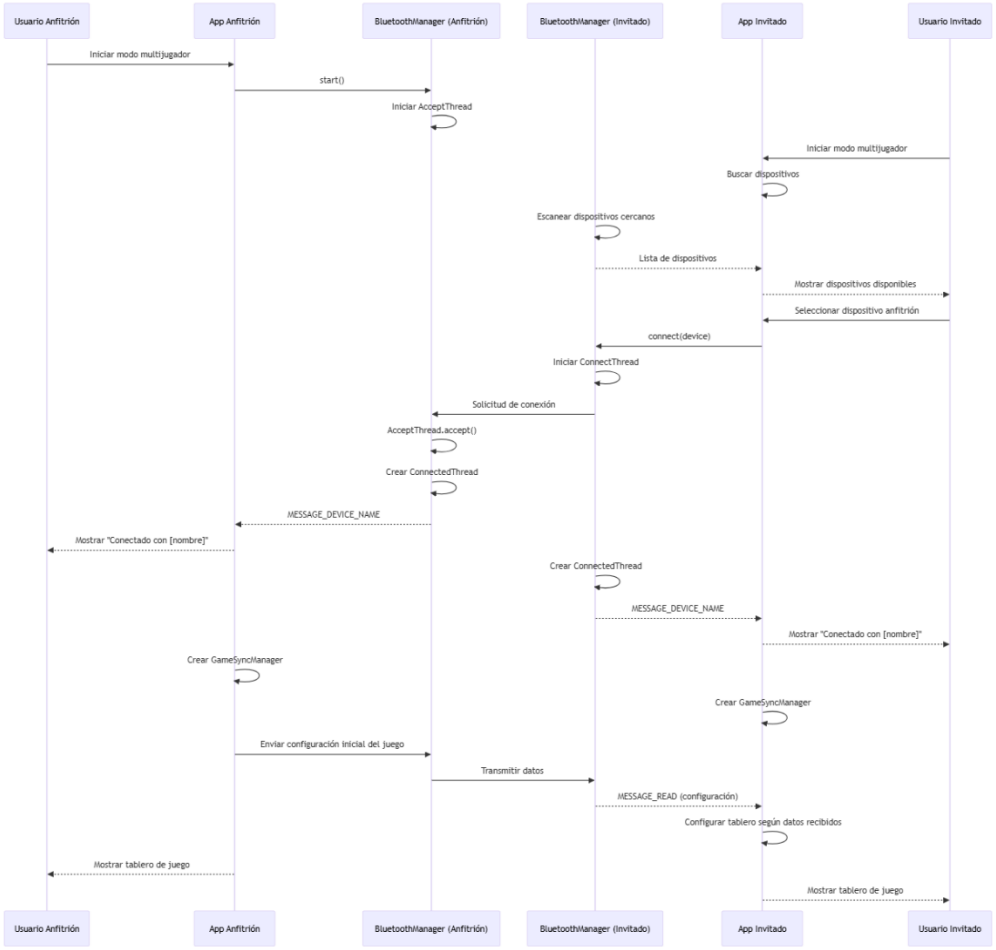



*Figura 5 Comunicación Bluetooth*

- Diagrama de Secuencia - Juego en progreso

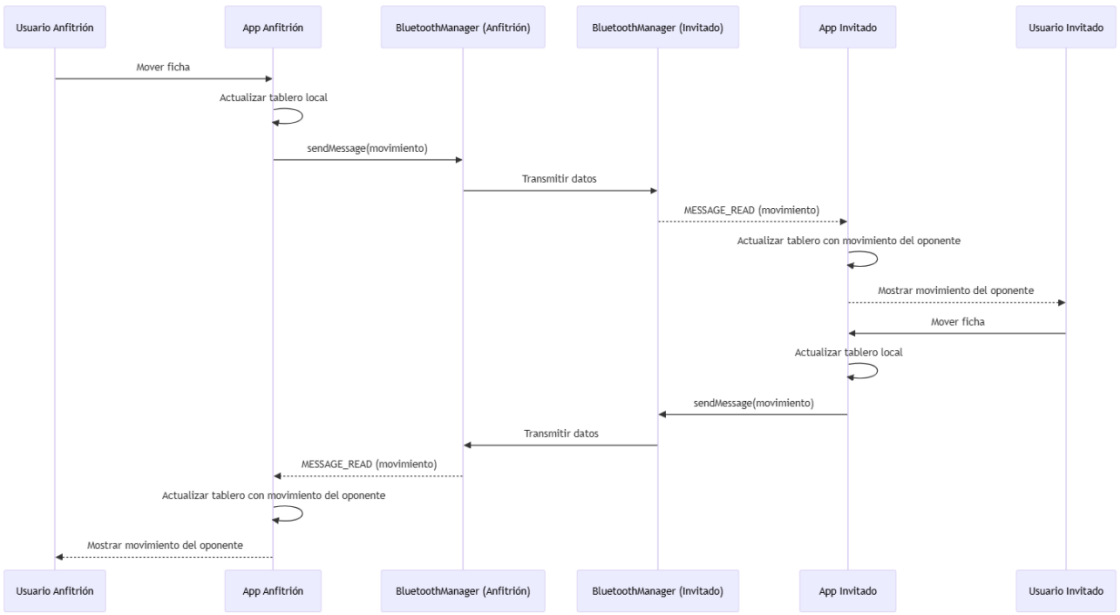



*Figura 6 Juego en progreso*

- Diagrama de Secuencia - Fin del juego

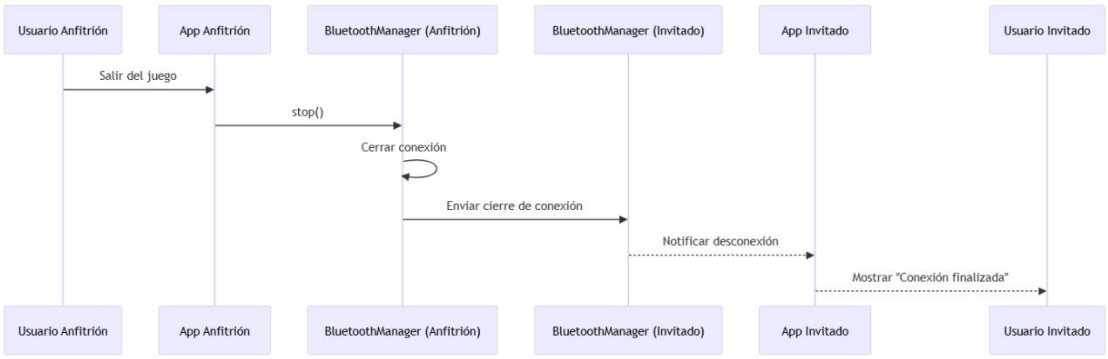
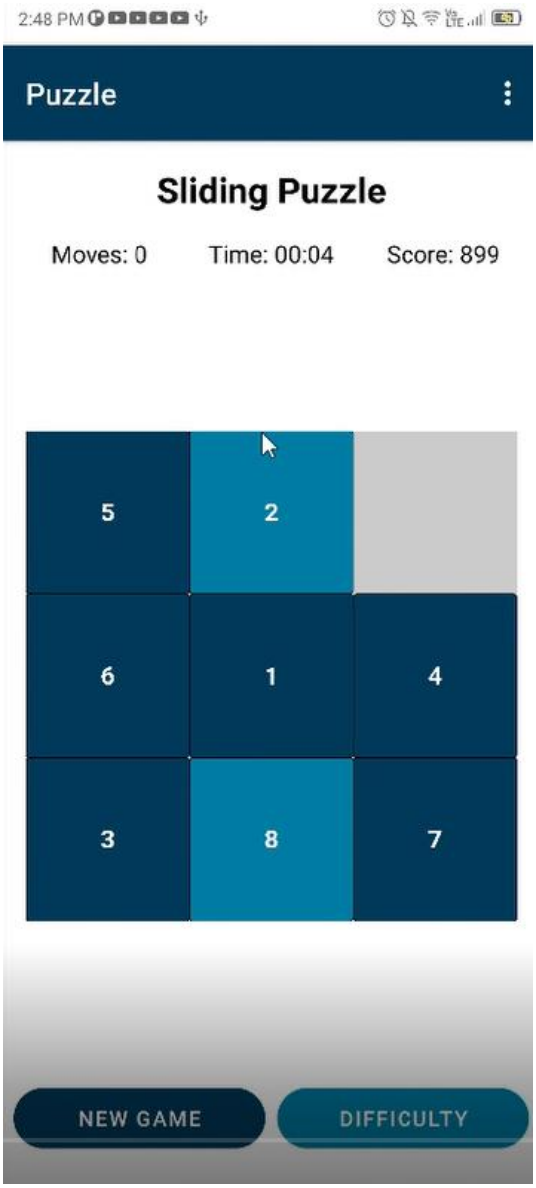
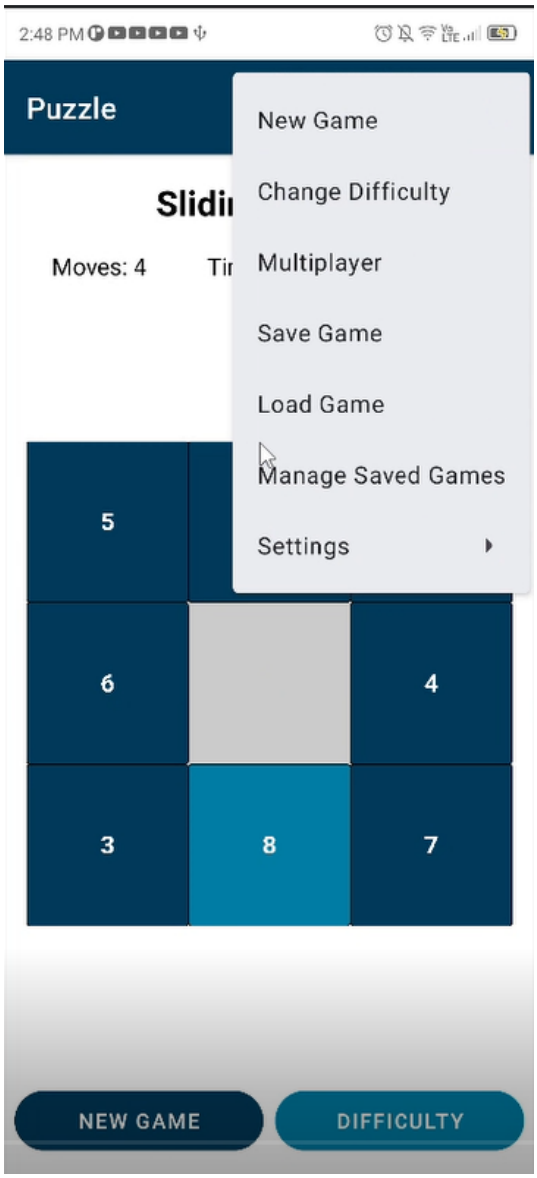


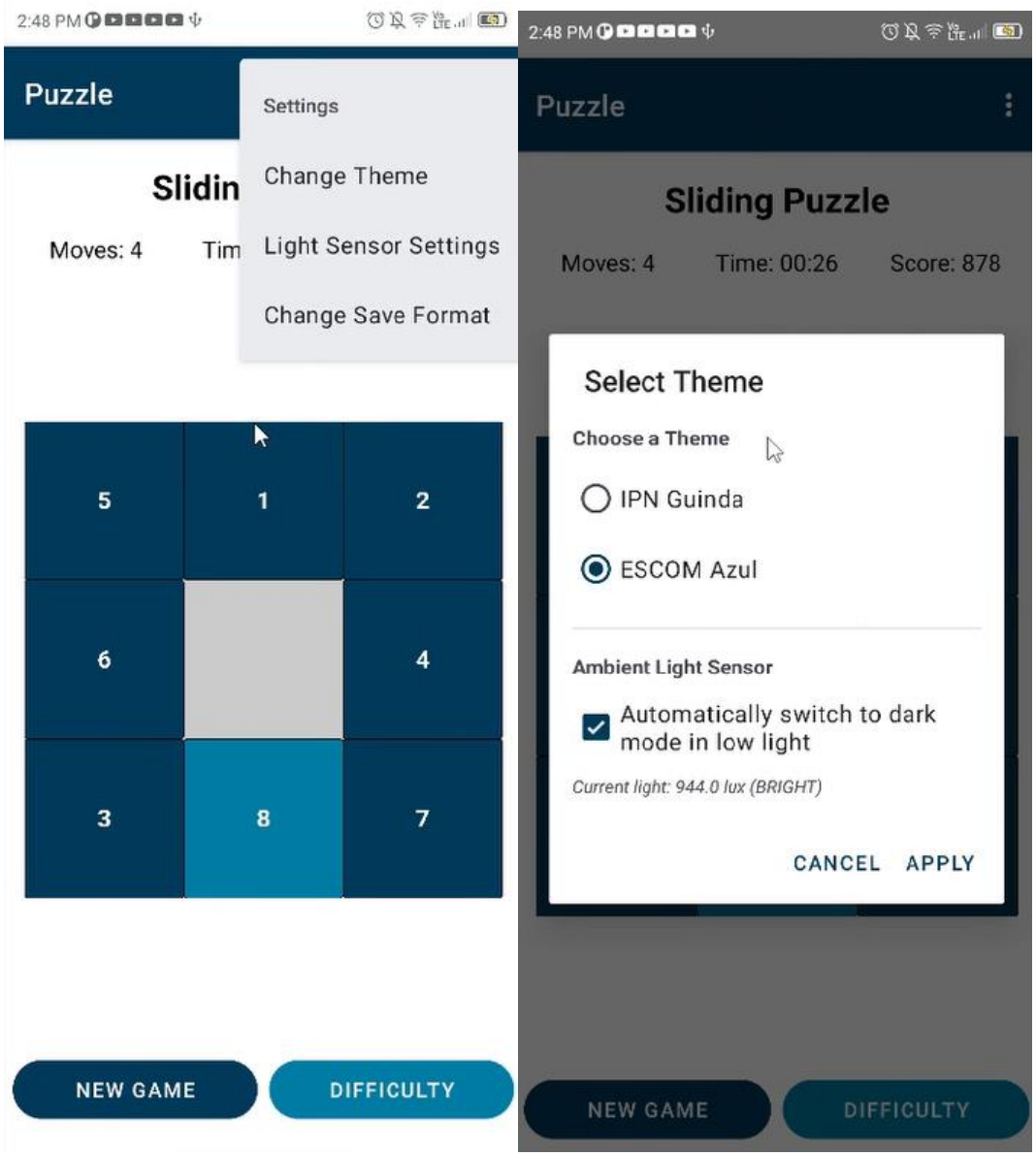

*Figura 7 Fin del juego*

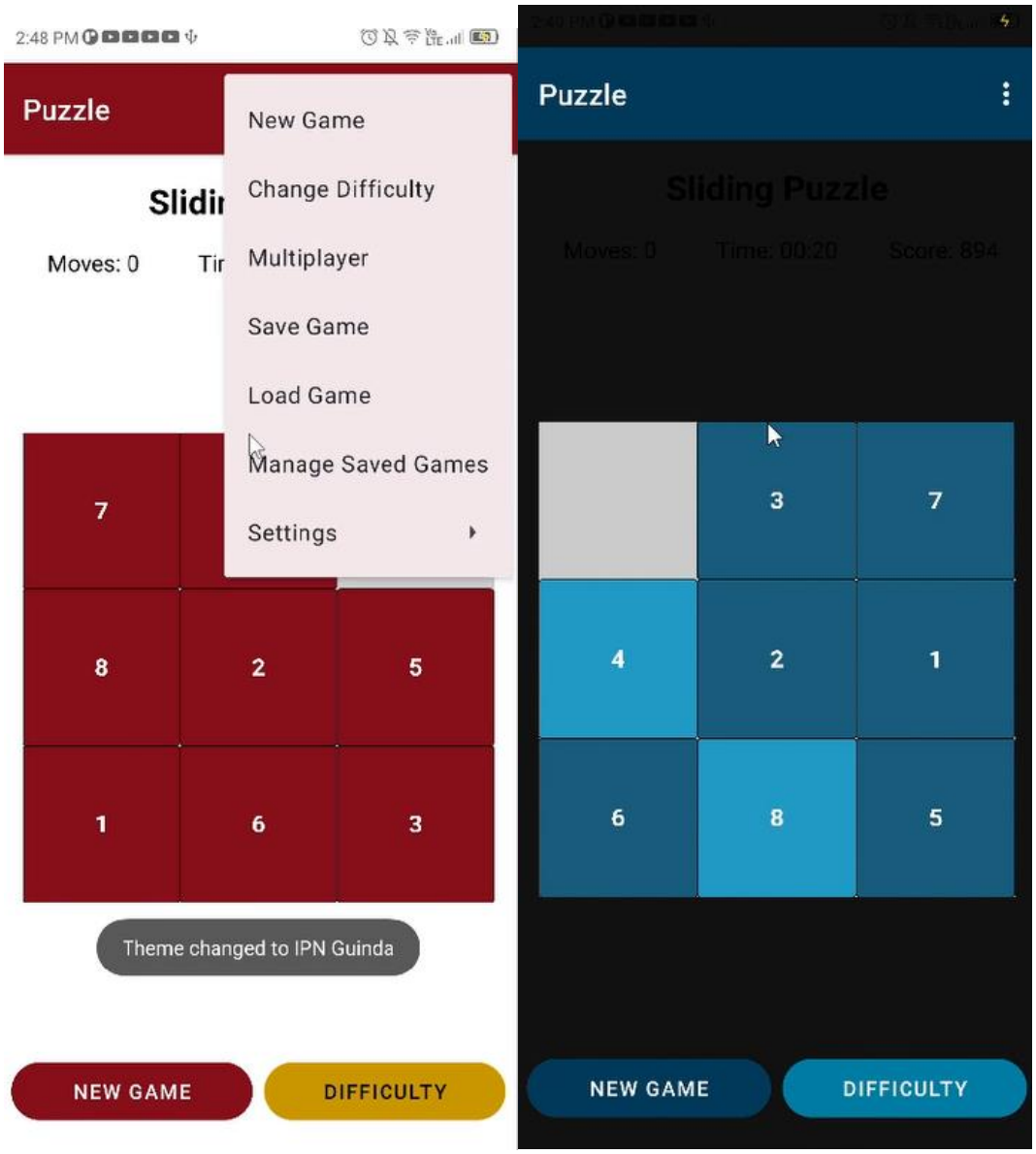
# 4 Capturas de Pantalla

- Implementación de sensores




2:48 PM
Puzzle
Settings
Change Theme
Light Sensor Settings
Change Save Format
Slidin
Moves: 4
Tim
5
1
2
6
4
3
8
7
NEW GAME
DIFFICULTY
2:48 PM
Puzzle
Sliding Puzzle
Moves: 4
Time: 00:26
Score: 878
Select Theme
Choose a Theme
IPN Guinda
ESCOM Azul
Ambient Light Sensor
Automatically switch to dark mode in low light
Current light: 944.0 lux (BRIGHT)
CANCEL
APPLY
NEW GAME
DIFFICULTY


2:48 PM
Puzzle
New Game
Change Difficulty
Multiplayer
Save Game
Load Game
Manage Saved Games
Settings
Moves: 0
7
8
2
5
1
6
3
Theme changed to IPN Guinda
NEW GAME
DIFFICULTY
Puzzle
Sliding Puzzle
Moves: 0
Time: 00:20
Score: 894
3
7
4
2
1
6
8
5
NEW GAME
DIFFICULTY

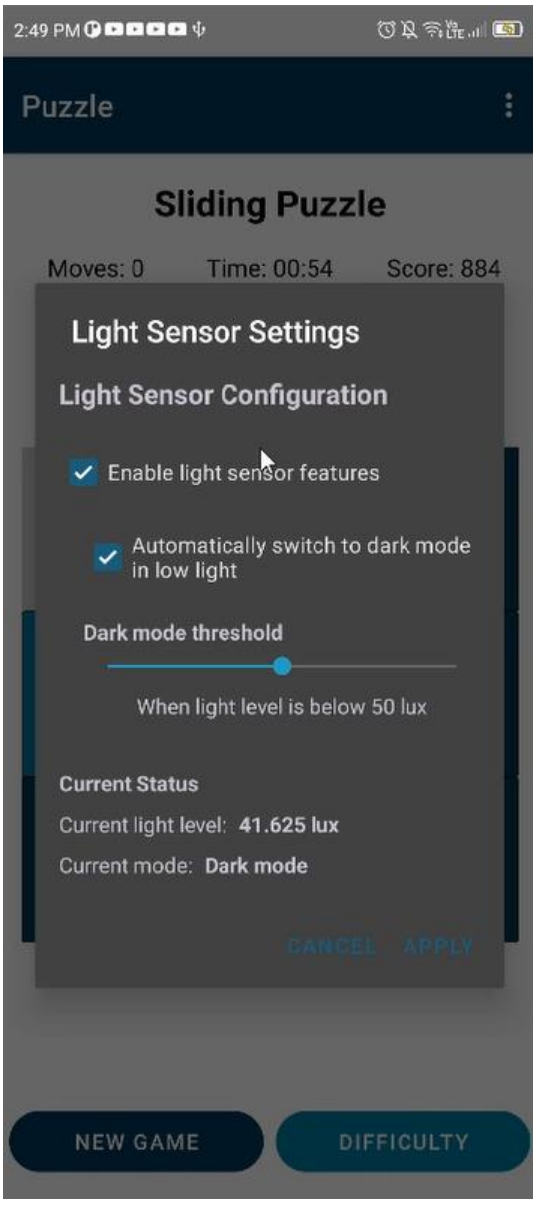



*Figura 8 Implementación de sensores*

- Conexión Bluetooth

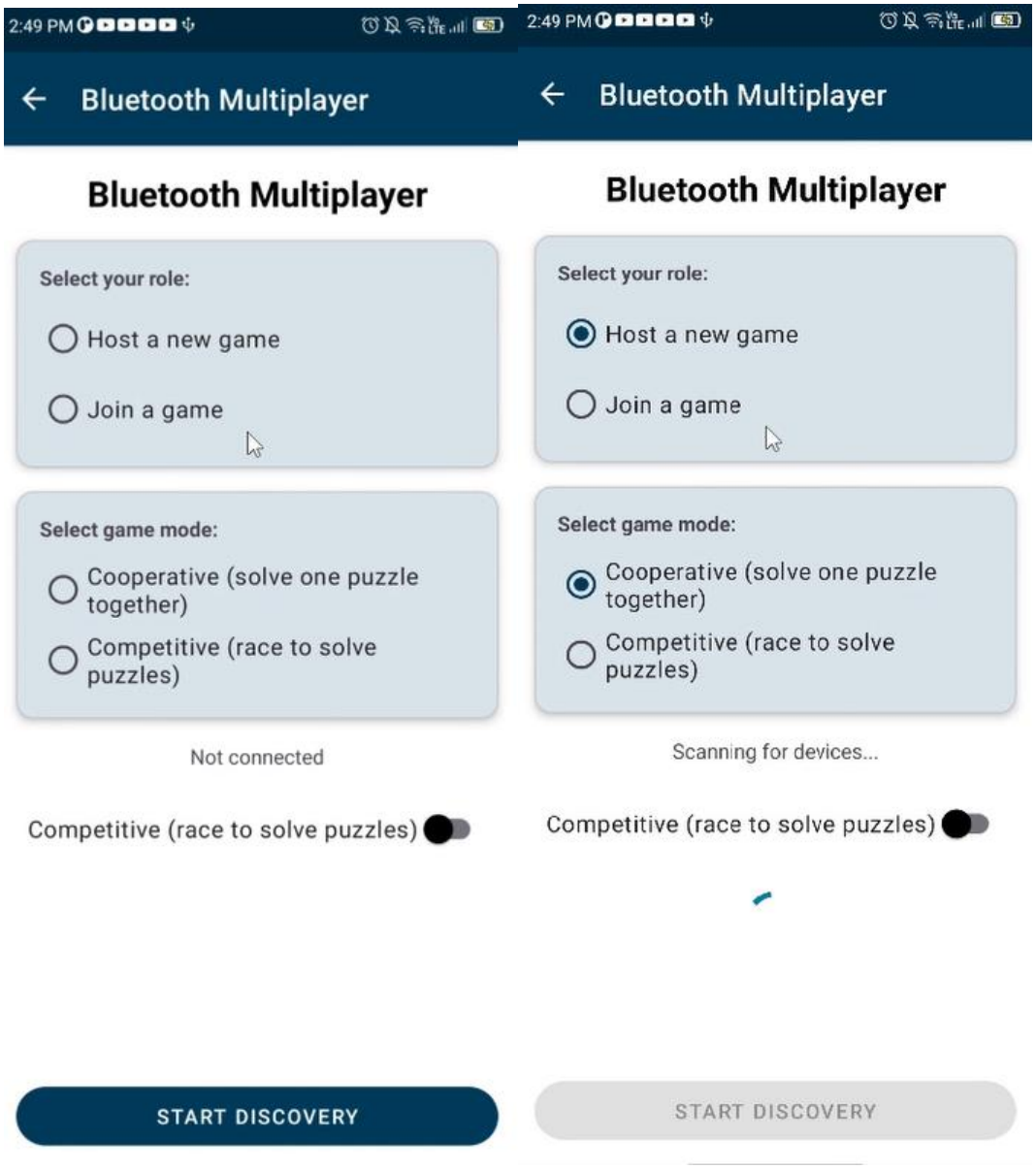



*Figura 9 Funcionalidades de Búsqueda, Favoritos y Recomendaciones*

# 5 Pruebas Realizadas

Se realizaron pruebas exhaustivas de las funcionalidades implementadas en diferentes dispositivos:

**Pruebas del sensor de luz**:

- Se verificó la correcta detección de cambios en las condiciones de luz.
- Se probó la adaptación automática de la interfaz en diferentes condiciones lumínicas.
- Se validó el funcionamiento del ajuste manual de los umbrales de luz.

**Pruebas de comunicación Bluetooth**:

- Se verificó el descubrimiento correcto de dispositivos cercanos.
- Se probó el establecimiento de conexión entre dispositivos.
- Se evaluó la sincronización del estado de juego en tiempo real.
- Se midió la latencia de la comunicación en diferentes escenarios.

**Pruebas de temas**:

- Se validó la correcta aplicación de los temas personalizados.
- Se verificó la adaptación automática al modo del sistema.

**Dispositivos de prueba:**

- Samsung Galaxy S22 (Android 13)

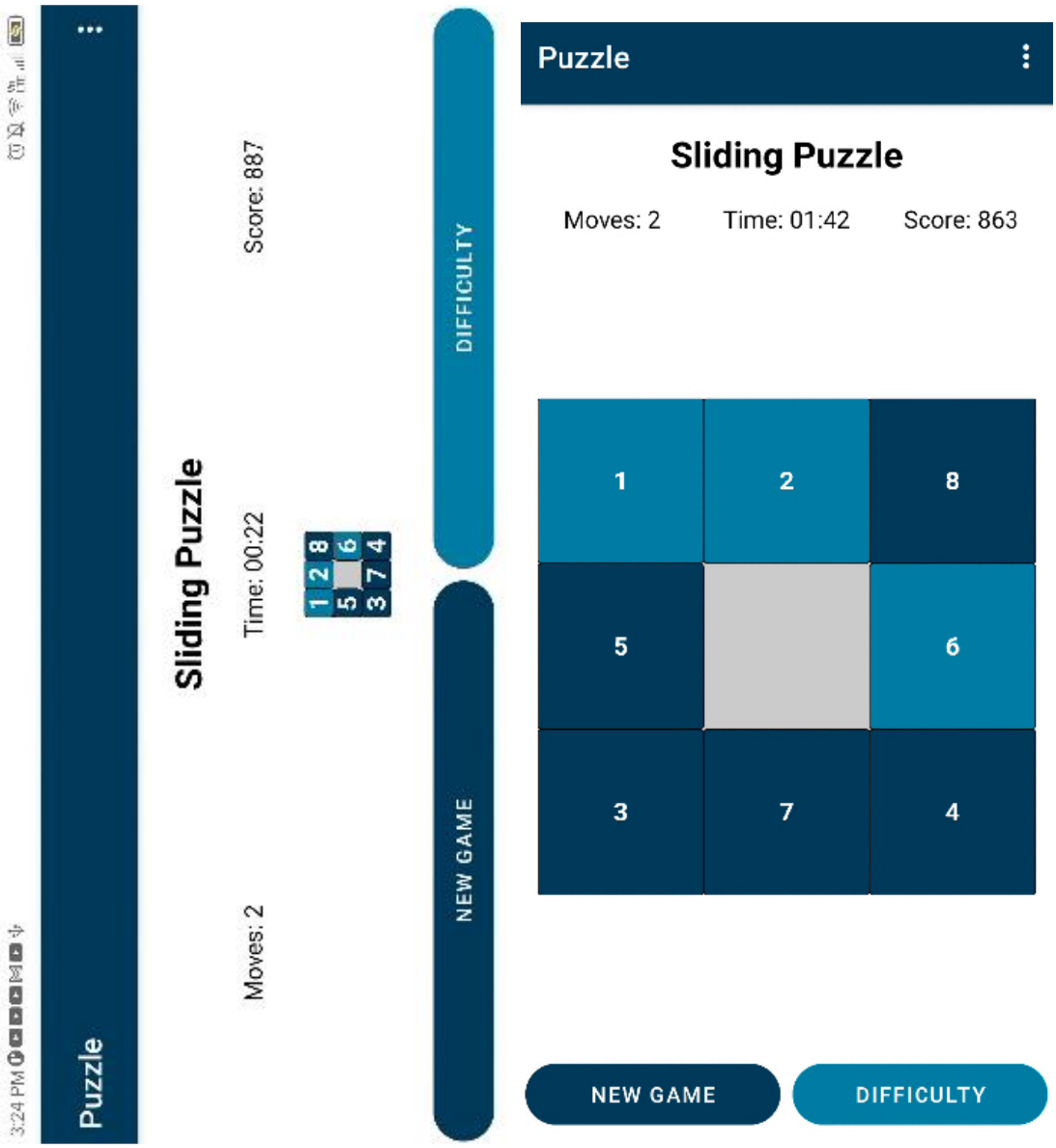



*Figura 10 Samsung Galaxy S22*

- Xiaomi Redmi Note 11 (Android 12)

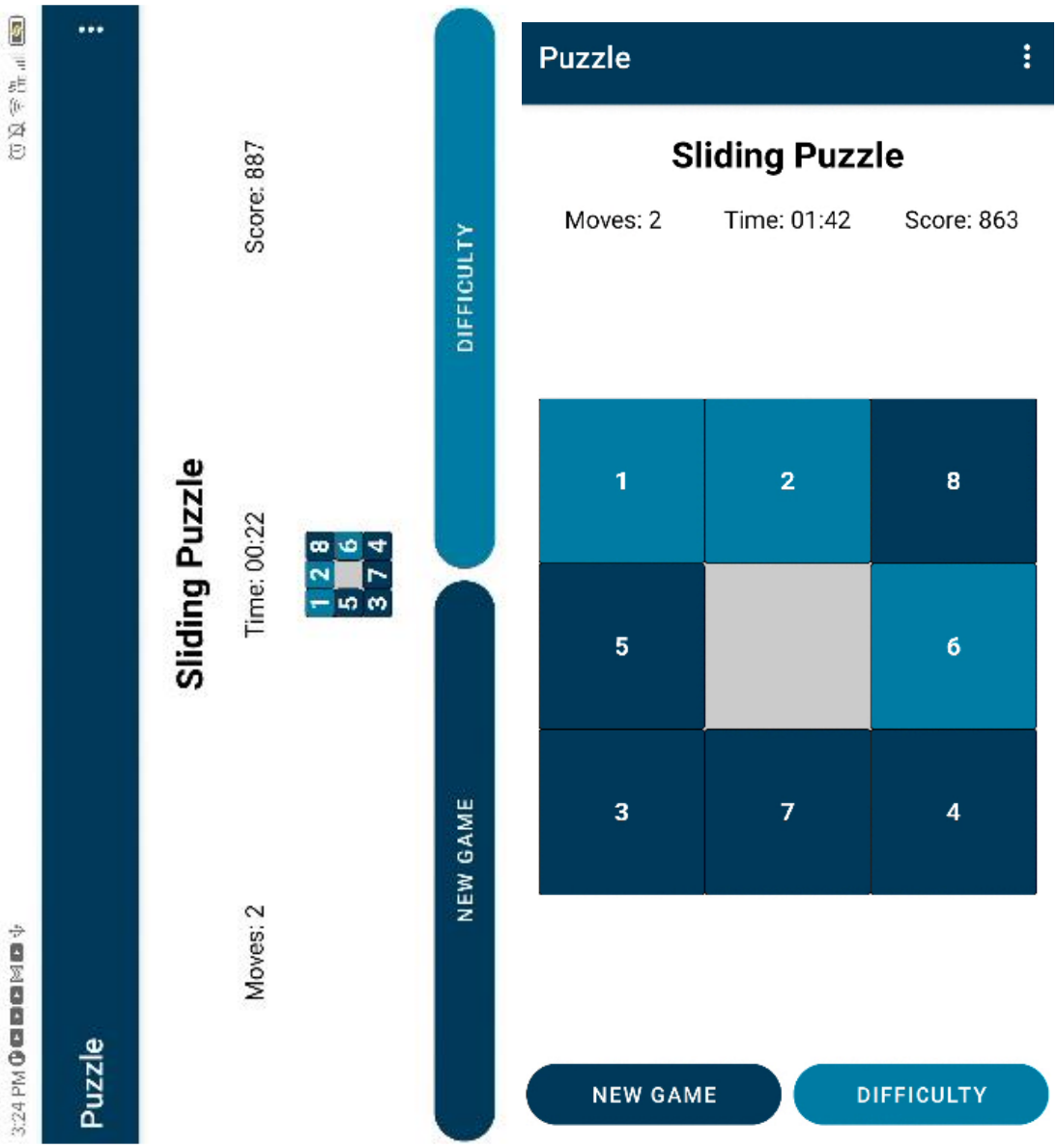



*Figura 11 Xiaomi Redmi Note 11*

- Google Pixel 6 (Android 14)

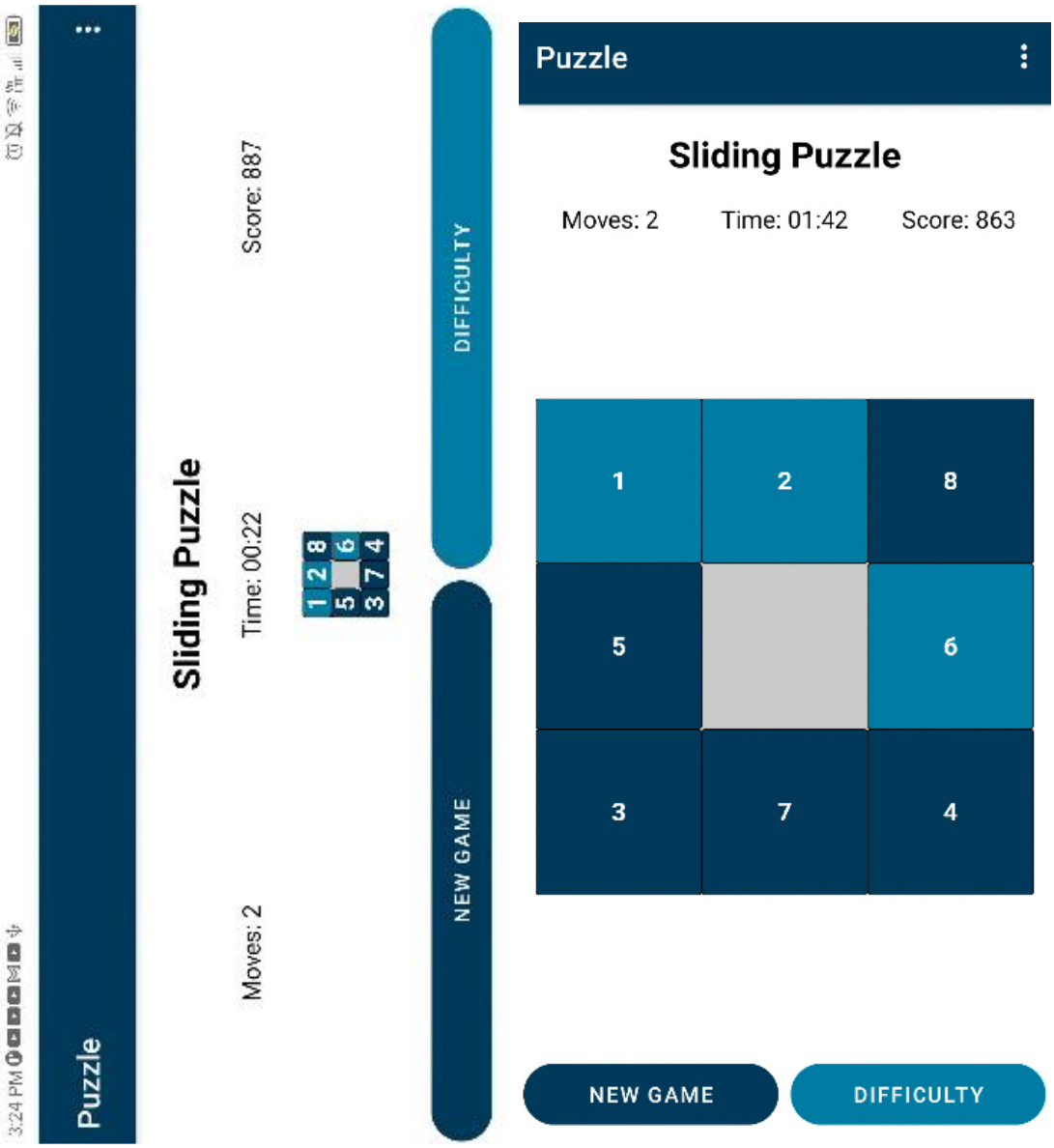



*Figura 12 Google Pixel 6*

# 6 Conclusiones

La implementación de sensores y comunicación Bluetooth en la aplicación de juego de Puzzle ha permitido enriquecer significativamente la experiencia de usuario. El sensor de luz ambiental proporciona una adaptación automática de la interfaz a las condiciones del entorno, mejorando la usabilidad en diferentes escenarios.

La funcionalidad de multijugador vía Bluetooth añade una dimensión social al juego, permitiendo a los usuarios competir en tiempo real. Esta característica aumenta el valor de la aplicación y su atractivo para los usuarios.

Los mayores desafíos enfrentados durante el desarrollo fueron:

1. La gestión eficiente de la energía al utilizar sensores continuamente.
2. La sincronización precisa del estado del juego entre dispositivos.
3. La adaptación de la interfaz para diferentes condiciones de luz sin comprometer la usabilidad.

Esta práctica ha permitido profundizar en el conocimiento de las capacidades de hardware de los dispositivos Android y cómo aprovecharlas para crear experiencias de usuario más ricas y contextuales.

# 7 Referencias


- Android Developers. (2025). Sensors Overview. https://developer.android.com/guide/topics/sensors/sensors_overview
- Android Developers. (2025). Bluetooth Overview. https://developer.android.com/guide/topics/connectivity/bluetooth
- Phillips, B., Stewart, C., & Marsicano, K. (2024). Android Programming: The Big Nerd Ranch Guide (6th ed.). Big Nerd Ranch.
- Griffiths, D., & Griffiths, D. (2023). Head First Kotlin: A Brain-Friendly Guide. O'Reilly Media.
- Meier, R. (2024). Professional Android (5th ed.). Wrox.