Merge Down

Entrega 2 - Flutter

Desarrollo de aplicaciones para dispositivos móviles

4º Diseño y desarrollo de videojuegos - Quintana

Ana Ordóñez Gragera

Laura García Martín

Marta Raboso Gómez

Lucía Sánchez Abril

Francisco Rodríguez Martínez

Índice

1. Introducción

Este proyecto tiene como objetivo poner en práctica el desarrollo de aplicaciones móviles con **Flutter** mediante la creación de un juego casual.

Esta aplicación ha tomado como referencia juegos como "**2048**" o "**Tetris**" adaptando sus mecánicas principales para ofrecer una nueva y dinámica experiencia para todos los usuarios. El juego ofrecerá un tablero cuyas columnas deben ser rellenadas con los números generados, haciendo que el jugador tenga como objetivo emparejar aquellas fichas de números iguales, para así alcanzar el mayor número posible hasta ocupar el número máximo de casillas disponibles. Además, la interfaz de juego cuenta con un contador de puntuación para mantener al jugador informado del progreso de su partida, así como, un contador de movimientos para tener en conocimiento la cantidad de números que se han desplazado.

"Merge Down" permite que el jugador pueda personalizar sus partidas, permitiendo cambiar los colores mostrados por pantalla o la activación de los sonidos y música que acompañan al jugador en la partida.

2. Implementación del juego

En el siguiente apartado, se podrá observar como se ha desarrollado e implementado cada una de las funciones principales del juego.
El juego cuenta con dos clases principales, **GameLogic** y **GameScreen**. La primera definirá con sus funciones la lógica del juego, como serian los movimientos posibles o las casillas existentes en el tablero, mientras que la segunda se encargará de definir las propiedades de cada casilla según el dispositivo, desplazamientos, gravedad, combinaciones de números, es decir, aquello que solo surge durante la partida y que afecta directamente al dispositivo del jugador.

a. Tabla

Para inicializar la tabla compuesta por un número concreto de casillas, existe la función **initializeGrid()**, en GameLogic, que se encargará de inicializar la tabla desde cero.

```
// Método para inicializar la cuadrícula
void initializeGrid() {
    // Aseguramos que la cuadrícula sea de 5x5
    tileGrid = List.generate(
      gridSize,
      // Cada casilla comienza con valor 0
      (i) => List.generate(gridSize, (j) => Tile(x: i, y: j, value: 0)),
    );
    print("Grid initialized with dimensions: ${tileGrid.length}x${tileGrid[0].length}");
    printGrid();
    // Imprimimos el estado inicial de la cuadrícula para verificar
}
```

Mientras que **GameScreen** se encarga de inicializar las casillas de manera personalizada, dependiendo del tamaño del dispositivo del jugador.

```
@override
void didChangeDependencies() {
    super.didChangeDependencies();
    // Calculamos el tamaño de las casillas de manera dinámica según el ancho de la pantal
    double screenWidth = MediaQuery.of(context).size.width;
    double gridSizePx = screenWidth - 16.0 * 2;
    tileSize = (gridSizePx / 5) - 8;

    // Inicializamos las posiciones de los tiles (fuera de la cuadrícula)
    initializeTilePositions();
}

void initializeTilePositions() {
    // Inicializar posiciones de todos los tiles en sus posiciones originales (fuera de la
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            // Fuera de la cuadrícula
            tilePositions["$i-$j"] = -tileSize;
        }
    }
}
```

Finalmente con la función **Widget build()**, se construirá completamente el tablero, definiendo el tamaño de este, el espacio entre las casillas y su tamaño, los

diferentes colores de las fichas y tablero, e incluso el color de la fuente aplicada. Mediante un bucle for se crean las cuadrículas del tablero y que gracias a la lista de **Widget stackItems**, se pueden añadir las diferentes animaciones de movimiento de cada ficha con sus personalizaciones concretas, y así poder detectar además los arrastres horizontales que realice el jugador.

Además se creará un **Widget** para mostrar el siguiente número al jugador, el cual en su definición se puede encontrar la posición correcta que debe presentar y sus tamaño y color.

b. Interfaz

La personalización de esta se encuentra en la clase **GameScreen**, mostrando la estructura principal que presenta compuesta por su color de fondo con degradado, los spacers necesarios y los diferentes Widget a mostrar, como el botón para pausar el juego, que conducirá al menú de pausa, el tablero, las fichas, el próximo número y por último la fila de flechas a pulsar.

Fondo, puntuación y movimientos	Widget

`return Scaffold(`
`  // Fondo con degradado`
`  body: Container(`
`  decoration: gradientDecoration,`
`  child: Center(`
`  child: Column(`
`  mainAxisSize: MainAxisSize.min,`
`  children: [`
`  // Espaciador flexible`
`  const Spacer(flex: 5),`
`  Container(`
`  // Alineaciones y personalizació`
`  // Botón de menú`
`  options,`
`  const Spacer(flex: 10),`
`  // Puntuación`
`  showScore,`
`  const Spacer(flex: 1),`
`  // Número de movimientos`
`  showMoves,`
`  SizedBox(width: outerPadding),`
`  ],`
`  ),`
`  ),`
`  ),`
`  ),`
	`  // Widget que muestra el próximo número`
	`  nextNumberDisplay,`
	`  const Spacer(flex: 5),`
	`  // Tablero`
	`  Container(`
	`  // Fichas`
	`  child: Stack(children: stackItems),`
	`  ),`
	`  // Espacio entre elementos`
	`  const Spacer(flex: 4),`
	`  // Fila con flechas`
	`  Container(`
	`  return GestureDetector(`
	`  onTap: () => onColumnTap(index),`
	`  ),`
	`  ),`
	`);`

Widget de opciones

```
// Widget de opciones
Widget options = ElevatedButton(
  onPressed: () => {
    // Navega al menú de pausa
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => PauseGame()),
    );
  },
  child: Icon(Icons.menu_rounded),
);
```

c. Generación de números

GameLogic se encarga de la generación aleatoria de números gracias a la función **generateNextNumber**(), la cual cuenta con unos números ya definidos entre los que puede generar uno aleatorio, para así poder controlar el valor de estos.

```
// Método para generar un número aleatorio
void generateNextNumber() {
    // Números posibles para el juego
    List<int> possibleNumbers = [2, 4, 8];
    nextNumber = possibleNumbers[Random().nextInt(possibleNumbers.length)];
    print("Generated next number: $nextNumber");
}
```

d. Movimientos

GameLogic tiene la función **placeNumberInColumn()**, cuyo objetivo será detectar si es posible la colocación del número en alguna casilla del tablero. Comprueba si hay un número generado, calcula si hay alguna posición vacía en la columna recorriendo todas sus casillas, en caso de haber hueco, generará otro número, sin

embargo, si detecta que la tabla está completa, gracias a la función **isGridFull(),** imprimirá un mensaje por pantalla y devolverá el valor “false”.

Método placeNumberInColumn()

```
bool placeNumberInColumn(int column) {
    if (nextNumber == null)
        // Si no hay un número generado, no se puede colocar
        return false;

    // Itera desde la fila inferior hacia arriba buscando una posición vacía
    for (int row = gridSize - 1; row >= 0; row--) {
        if (tileGrid[row][column].value == 0) {
            // Coloca el número en la posición vacía
            tileGrid[row][column].value = nextNumber!;
            console.log(`Placed ${nextNumber} in row ${row}, column ${column}`);
            // Imprimimos el estado de la cuadrícula después de cada colocación
            printGrid();
            // Genera un nuevo número para la próxima colocación
            generateNextNumber();
            return true;
        }
    }

    // Verifica si la cuadrícula está llena después del intento
    isGridFull();
    // Si no hay posiciones vacías en la columna, muestra un mensaje y retorna falso
    console.log(`Column ${column} is full. Could not place ${nextNumber}`);
    return false;
}
```

Método isGridFull()

```
// Método para verificar si la cuadrícula está llena
bool isGridFull() {
    for (int row = 0; row < gridSize; row++) {
        for (int col = 0; col < gridSize; col++) {
            if (tileGrid[row][col].value == 0) {
                // Si encuentra una casilla vacía, la cuadrícula no está llena
                return false;
            }
        }
    }
```

```
    // Verifica si hay movimientos posibles
    gameOver = possibleMoves();
    // Retorna true si la cuadrícula está llena
    return true;
}
```

También, existe una función que calculará si existen movimientos posibles, recorriendo todas las casillas para encontrar valores iguales,y en este caso poder seguir con la partida, mediante el uso de un bucle for que explorará todas las casillas posibles y las comparaciones mediante el uso de if para devolver el resultado correcto.

Método possibleMoves()

```
bool possibleMoves() {
    for (int row = 0; row < gridSize; row++) {
        for (int col = 0; col < gridSize; col++) {
            int value = tileGrid[row][col].value;

            // Verifica si hay un vecino a la derecha con el mismo valor
            if (col < gridSize - 1 && tileGrid[row][col + 1].value == value) {
                // Hay un movimiento posible
                return false;
            }

            // Verifica si hay un vecino abajo con el mismo valor
            if (row < gridSize - 1 && tileGrid[row + 1][col].value == value) {
                // Hay un movimiento posible
                return false;
            }

            // Verifica si hay un vecino a la izquierda con el mismo valor
            if (col > 0 && tileGrid[row][col - 1].value == value) {
                // Hay un movimiento posible
                return false;
            }

            // Verifica si hay un vecino arriba con el mismo valor
            if (row > 0 && tileGrid[row - 1][col].value == value) {
                // Hay un movimiento posible
                return false;
```

Mientras que **GameScreen** contiene la función **onColumTap()**, que se encargará de calcular toda la lógica tras tocar alguna columna para colocar un número, en caso de ser posible, se aumentará el contador de movimientos y reproducirá un sonido determinado, además de aplicar la gravedad a la casilla para que llegue al primer hueco vacío de la columna, si esta columna está completa, se imprimirá un mensaje por pantalla sugiriendo que se utilice otra, al terminar el juego, se reproducirá una transición que conducirá al jugador a la pantalla de derrota, acabando así con la partida actual.

Método onColumnTap()

```
void onColumnTap(int column) {
    final musicState = Provider.of<MusicState>(context, listen: false);
    setState(() {
        bool placed = gameLogic.placeNumberInColumn(column);
        if (placed) {
            // Incrementamos el contador de movimientos
            moves++;
            // Reproducimos sonido de toque
            musicState.tapSound();
            // Aplica gravedad para ajustar las fichas
            applyGravity();
        } else {
            // Si la columna está llena, mostramos un mensaje al jugador
            ScaffoldMessenger.of(context).showSnackBar(
                SnackBar(content: Text("Full column. Choose another one.")),
            );
        }
        // Si el juego ha terminado, navegamos a la pantalla de derrota
        if (gameLogic.gameOver) {
            Navigator.of(context).push(
                PageRouteBuilder(
                    pageBuilder: (context, animation, secondaryAnimation) =>
                        DefeatScreen(
                        score: score,
                        moves: moves,
                    ),
                    transitionsBuilder:
                        (context, animation, secondaryAnimation, child) {
                        return FadeThroughTransition(
                            animation: animation,
                            secondaryAnimation: secondaryAnimation,
                            child: child,
```

En ***"Merge Down"***, el juego permite que las casillas se deslicen horizontalmente mediante la función **onSwipe(),** que se encarga de verificar si las casillas pueden combinarse con las contiguas a través de la función **moveAndCombine**(). Si las casillas tienen el mismo valor, se suman, y si no, se desplazan hasta el primer espacio vacío. Cada movimiento incrementa el contador de movimientos. Después de cada deslizamiento, se aplica la función **applyGravity**() para asegurar que no queden espacios vacíos debajo de las casillas.

La función **applyGravity()** recorre cada columna de la cuadrícula y mueve las casillas hacia abajo para llenar cualquier espacio vacío, repitiendo el proceso hasta que no haya más combinaciones posibles en la columna.

Además, para mostrar la cantidad de movimientos realizados, se utiliza un widget **showMoves**, que ajusta su tamaño y contenido según los movimientos realizados, mostrando el contador en la pantalla.

Este proceso de diseño permite que el juego sea dinámico y reactivo, manteniendo una experiencia de juego fluida y desafiante para los jugadores.

e. Puntuación

Para mostrar la puntuación en la pantalla de juego, se puede encontrar la creación del **Widget showScore** en la clase **GameScreen**, donde se definen las personalizaciones deseadas para su impresión por pantalla, así como el texto que lo acompaña y el ajuste del tamaño al contenido deseado a mostrar.

```
// Widget para mostrar la puntuación
Widget showScore = Container(
  // Personalizaciones
  ...
),
...
  Text(
    'SCORE',
    // Personalización del texto
    ...
  ),
```

3. Clases y navegación

a. MyApp

Main es la función principal que ejecuta la app. Una vez se asegura de que todos los widgets están inicializados, bloquea la orientación a modo retrato. Inicializa el estado de la música y comienza a reproducirla si está habilitada. **MyApp** es un consumidor que contiene dos proveedores para gestionar varios estados: el tema y

el estado de la música y efectos de sonido. Esto significa que se le notifica cuando hay cambios y actualiza su estado. Define **HomeScreen** como la pantalla inicial del juego.

b. Menú principal

El menú principal está compuesto por una columna centrada con los elementos separados mediante espaciados dinámicos o Spacers.

Tiene el logo de la aplicación y tres botones:

- **'Start'**: comienza la partida.
- **'Settings'**: muestra los ajustes de la aplicación.
- **'How To Play'**: enseña una breve descripción de cómo jugar.

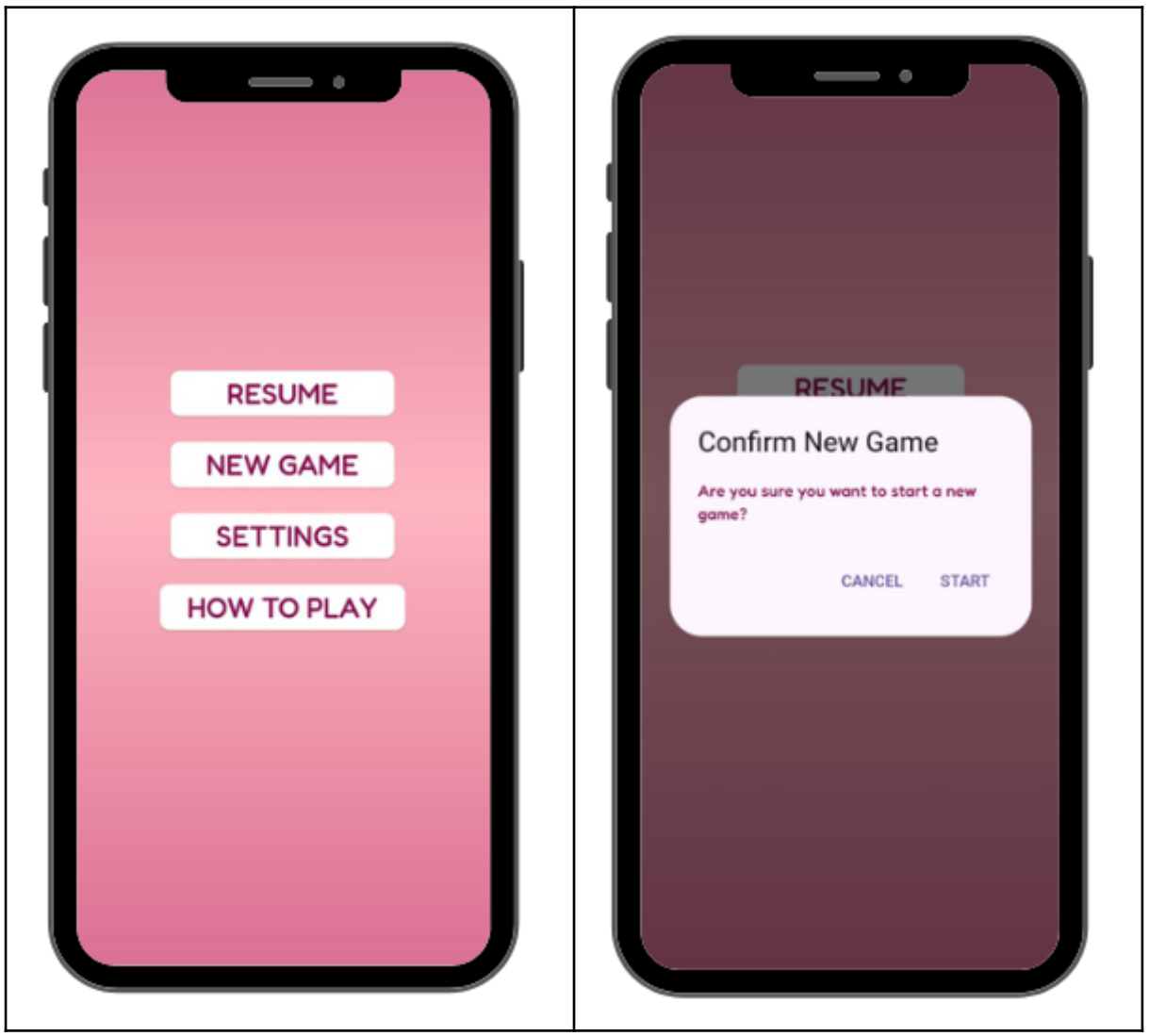
c. Pantalla de juego

La pantalla con el juego está ordenada con una columna centrada y elementos espaciados dinámicamente. En primer lugar, tiene una fila con un botón para navegar al menú de pausa, y a la derecha los contadores de puntuación y movimientos de la partida. A continuación, tiene un Widget que muestra el siguiente número que va a aparecer en el tablero al pulsar una de las columnas. Debajo, está el tablero con sus fichas y debajo de este, una fila con iconos de flechas hacia abajo que indican que al tocarlas caerá en esa columna el número siguiente.	

d. Menú de pausa

La siguiente pantalla tiene varios botones centrados vertical y horizontalmente con varias opciones:

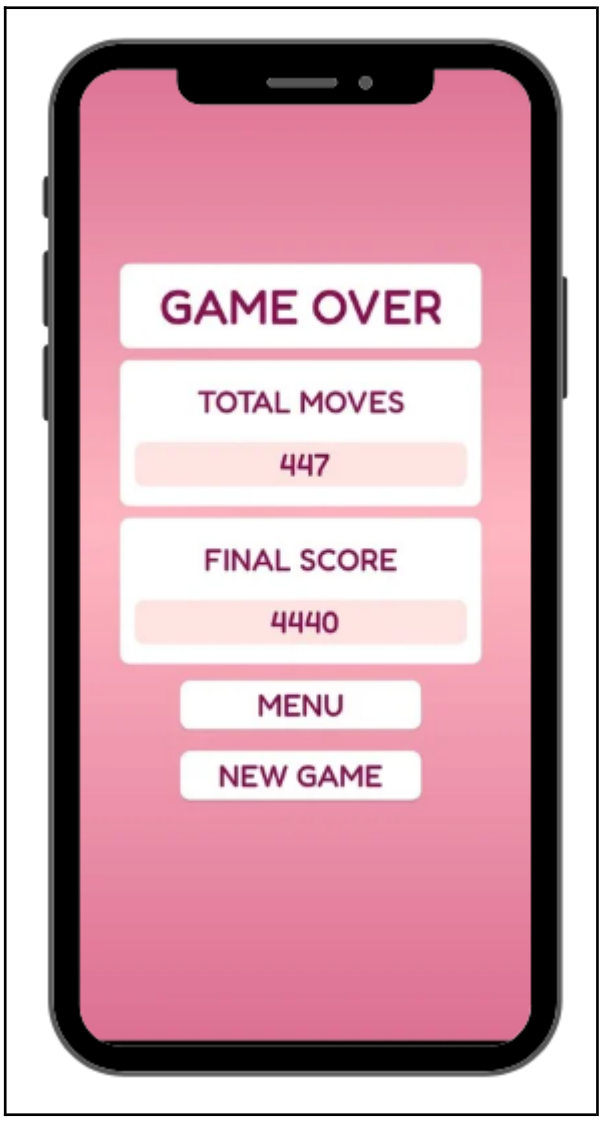
- **Resume**: cierra esa ventana y regresa a la partida
- **New Game:** muestra un cuadro de diálogo para confirmar el inicio de una nueva partida. Ese cuadro a su vez tiene dos botones, uno para cerrarlo y otro para iniciar otro juego.
- **Settings**: navega a la pantalla de ajustes.
- **How To Play:** muestra la pantalla de tutorial.



e. Pantalla de derrota

Cuando el tablero está lleno y no hay más movimientos posibles, se muestra la pantalla de derrota. Esta cuenta con cuatro elementos principales centrados y espaciados de manera dinámica.

Primero, tenemos un contenedor con el título de la pantalla, ‘Game Over’. Después tenemos dos contenedores que indican la puntuación final y la cantidad total de movimientos del juego. A su vez tienen un contenedor como hijo para enseñar el número de manera más llamativa. Por último, tenemos un botón para comenzar una nueva partida.



f. Pantalla de ajustes

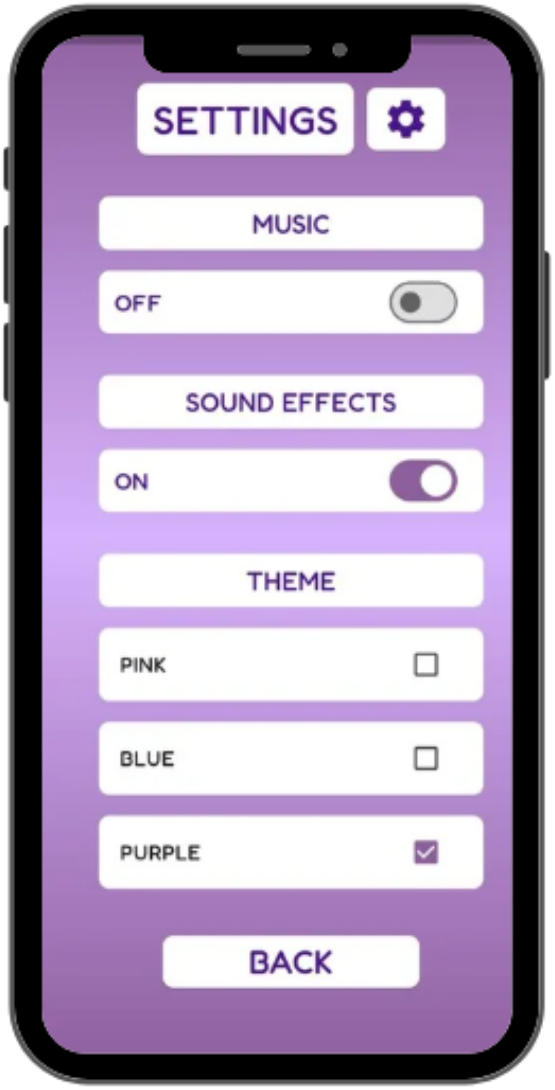
La pantalla de ajustes utiliza la clase **appTheme**, ya que es la encargada de gestionar el cambio de temas. También está organizada a través de una columna y varios contenedores.

Primero muestra dos contenedores con el título **'Settings'** y un icono de un engranaje. Debajo, se encuentran las opciones de sonido. Estos son dos Switches que controlan el estado de la música y los efectos de sonido. Se muestra **'On'** u **'Off'** dependiendo de este estado. Cuando cambian su valor, llaman a las funciones correspondientes de la clase **musicState**.

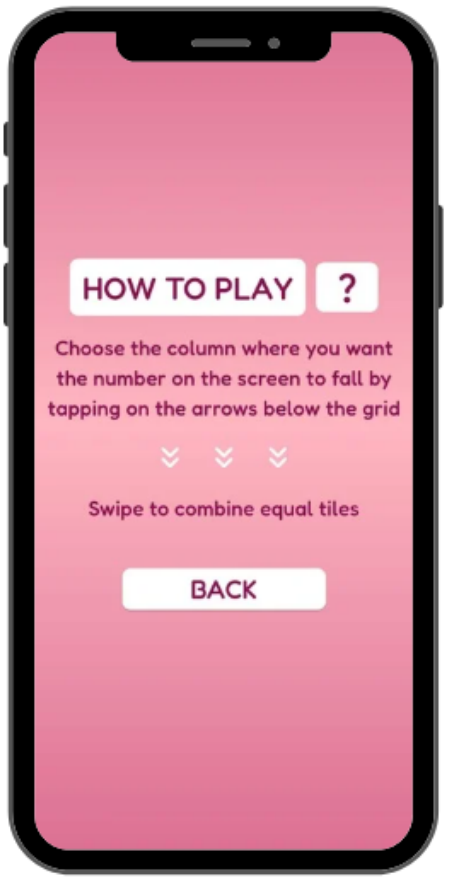
El interruptor de **'Music'** llama a **musicState.toggleMusic(),** la cual se encarga de pausar o continuar la canción. El interruptor de **'Sound Effects'** llama a **musicState.toggleSounds(),** que cambia el valor del estado de los efectos del juego.

A continuación, se encuentran los ajustes del tema de la aplicación. Hay tres checkboxes encargados de gestionar los colores del juego. Si el valor del tick es verdadero, cambia el tema. Guardan en value si el tema aplicado es el correspondiente a cada uno accediendo al índice del tema.

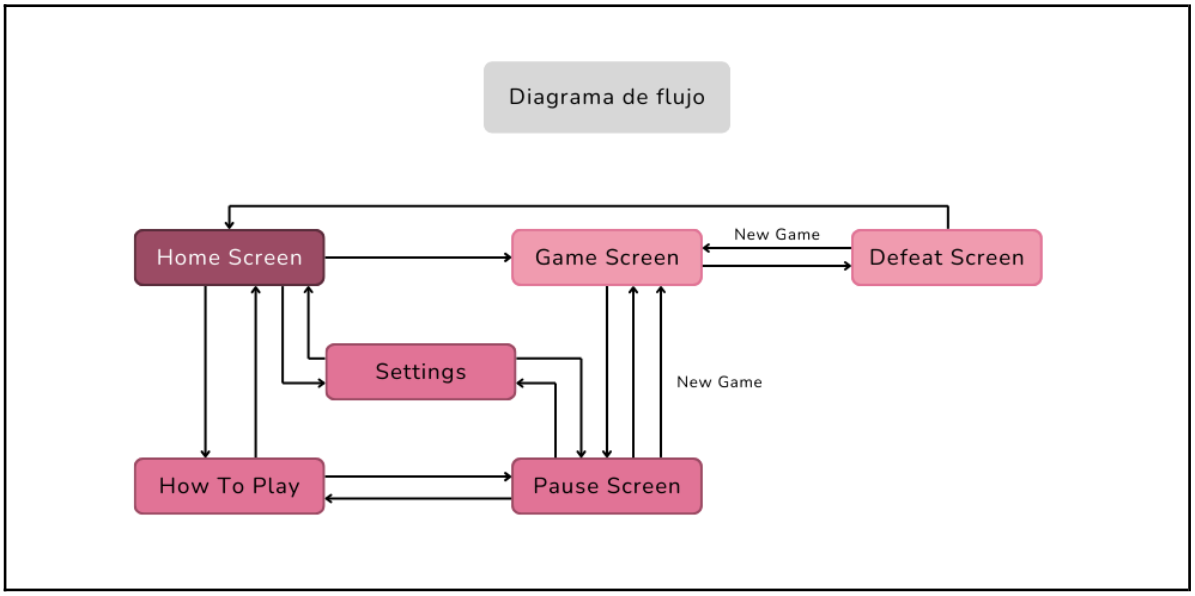
Para salir, se ha implementado un botón que cierra la ventana actual y vuelve a la pantalla anterior.



g. Pantalla de tutorial

La pantalla de tutorial tiene un título **'How To Play'** y un icono de interrogación envueltos en contenedores en la parte superior de la pantalla. Debajo, hay una breve explicación de cómo jugar y una fila con varios iconos con flechas que apuntan a abajo. Para salir también hay un botón **'Back'** que cierra esa ventana y vuelve a la pantalla previa.	

h. Diagrama de flujo



4. Persistencia

Para conseguir la persistencia y el guardado de ajustes entre sesiones, se han implementado las siguientes funcionalidades: tres temas y la opción de activar y desactivar tanto la música de fondo como los efectos de sonido.

a. Temas

La clase **AppTheme** es la encargada de gestionar los colores predeterminados de la app. En ella se han guardado varios colores predeterminados para cada tema, y un mapa para relacionar el valor de las fichas con un color. Desde cada tema se controlan los ajustes de varios elementos.

Se han modificado:

- **Textos**: color, peso y tamaño.
- **Botones elevados**: color del texto, color de fondo, fuente del texto y redondeado de los bordes.
- **Iconos**: color y tamaño.
- **Checkboxes**: colores cuando están seleccionados o no.
- **Switches**: colores de las pistas y botones.

```
// Tema Rosa
ThemeData get pinkTheme => ThemeData(
     // Color principal del tema
     primarySwatch: Colors.pink,
     // Color de fondo principal de la pantalla
     scaffoldBackgroundColor: lightPink,

     // Configuración de los estilos de texto
     textTheme: TextTheme(
       // Estilo para textos grandes
       bodyLarge: GoogleFonts.fredoka(
         textStyle: TextStyle(color: Colors.pink[900]),
         fontWeight: FontWeight.w500,
       ),
      // Estilo de los botones elevados
      elevatedButtonTheme: ElevatedButtonThemeData(
       style: ElevatedButton.styleFrom(
         // Color de fondo del botón
```

```
          backgroundColor: Colors.white,
          // Color del texto en el botón
          foregroundColor: Colors.pink[900],
          // Estilo del texto
          textStyle: GoogleFonts.fredoka(
            fontSize: 25,
            fontWeight: FontWeight.w500,
          ),
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(8.0),
          ),
        ),
      ),
      // Estilo de los iconos
      iconTheme: IconThemeData(
       color: Colors.pink[900],
       size: 36,
      ),
...
}
```

AppTheme hereda de **ChangeNotifier**. Esta es una clase que se utiliza principalmente en el patrón de diseño Provider para administrar el estado de una aplicación. **ChangeNotifier** facilita la notificación a los oyentes (listeners) cuando se producen cambios en el estado, lo que permite una actualización eficiente y reactiva de la interfaz de usuario. La clase **ChangeNotifier** permite que una instancia de esta clase notifique a sus oyentes cuando algo cambia, invocando el método **notifyListeners**.

```
      // Clase para gestionar el cambio de temas
      class AppTheme extends ChangeNotifier {
      ...
      }
```

En nuestra aplicación, tenemos dos listeners de AppTheme: **MyApp** y **Settings**. **Main** cargaría el tema rosa por defecto, pero toma los valores guardados en preferencias cuando inicia la app. **Settings** llama a los métodos que explicaremos a continuación de la clase **AppTheme** y cambia el tema mediante los checkboxes.

```
      // Main
```

```
      @override
      Widget build(BuildContext context) {
       // Permite gestionar múltiples estados de la app al mismo tiempo
       return MultiProvider(
         providers: [
           // Proveedor para gestionar el tema de la app
           ChangeNotifierProvider(create: (_) => AppTheme()),
           // Proveedor para gestionar el estado de la música
           ChangeNotifierProvider<MusicState>.value(value: musicState),
         ],
      ...
```

```
// Settings
Widget build(BuildContext context) {
 // Obtiene el fondo con gradiente desde el estado global del tema
 final gradientDecoration =
     Provider.of<AppTheme>(context).gradientBackground;
 // Obtiene el estado de la música desde el proveedor
 final musicState = Provider.of<MusicState>(context);

 // Interfaz de la pantalla
 return Scaffold(
   body: Consumer<AppTheme>(
     builder: (context, appTheme, child) {
...
```

Para gestionar la persistencia y guardado de los ajustes, se utiliza la clase **SharedPreferences**. Permite almacenar datos pequeños y persistentes en el dispositivo del usuario utilizando almacenamiento clave-valor. Este almacenamiento se encuentra en el disco local del dispositivo, por lo que los datos persisten incluso después de que la aplicación se cierre.

AppTheme inicializa el tema y algunos ajustes adicionales por defecto. Después llama a **loadTheme()**, que aplica el tema guardado en **SharedPreferences**.

```
      class AppTheme extends ChangeNotifier {
       // Guarda el índice del tema actual
       int _currentThemeIndex = 0;

       // Getter que devuelve el tema del índice actual
       int get currentThemeIndex => _currentThemeIndex;
```

```
    // Variable ThemeData que guarda el tema actual
    ThemeData _currentTheme = pinkTheme;

    // Getter que devuelve el tema actual
    ThemeData get currentTheme => _currentTheme;
...
    // Constructor que carga el tema cuando se inicializa la clase
    ...
AppTheme() {
    _loadTheme();
  }
...
```

El tema se cambia desde el método **changeTheme()**, que toma un índice como parámetro. Este índice se guarda en SharedPreferences y se llama a los métodos **_applyTheme()** y **_saveTheme().**

```
// Método para cambiar el tema y actualizar SharedPreferences
void changeTheme(int themeIndex) async {
  // Guarda el índice del tema
  _currentThemeIndex = themeIndex;
  SharedPreferences prefs = await SharedPreferences.getInstance();
  prefs.setInt('currentThemeIndex', themeIndex);

  _applyTheme(_currentThemeIndex);
  _saveTheme(_currentThemeIndex);
}
```

_applyTheme() contiene un switch, que evalúa el índice que le es pasado como parámetro. Llama a los métodos que establecen el tema y colores de las fichas y otros elementos. Si por algún motivo no se le pasa un índice correcto se cambia a rosa. Después se llama a **notifyListeners**(), para que las clases suscritas actualicen su estado.

```
// Método para aplicar el tema según el índice dado
void _applyTheme(int themeIndex) {
  // Aplica el tema correspondiente dependiendo del índice
  switch (themeIndex) {
    case 0:
      setPinkTheme();
      break;
```

```
          case 1:
            setBlueTheme();
            break;
          case 2:
            setPurpleTheme();
            break;
          // Si el índice no es válido, se aplica el tema rosa
          default:
            setPinkTheme(); // Tema por defecto
        }
        // Notifica a los listeners (widgets que usan este tema) para que
se actualicen
        notifyListeners();
      }
```

_saveTheme() guarda el tema actual y su índice en preferencias. **setPinkTheme()**, **setPurpleTheme()** y **setBlueTheme()** establecen el tema elegido y otros ajustes como el color de las fichas, el color del texto de las fichas o el color del fondo del tablero.

```
// Método para guardar el tema en SharedPreferences
void _saveTheme(int themeIndex) async {
      final prefs = await SharedPreferences.getInstance();
      prefs.setInt('currentThemeIndex', themeIndex);
}
```

```
// Configura los colores y el estilo del tema rosa
void setPinkTheme() {
      _currentTheme = pinkTheme;
      tileFontColor = Colors.pink[900]!;
      numTileColor = pinkTileColors;
      gridBackGroundColor = softGreyPink;
      backgroundColor = lightPink;
}
```

b. Sonidos

Los efectos de sonido se gestionan desde la clase **MusicState**. También se han empleado las clases de SharedPreferences y **ChangeNotifier** para gestionar el estado de la aplicación.

Para reproducir los sonidos, se ha utilizado el paquete **Soundpool**. Este sirve para reproducir sonidos cortos de manera eficiente. Permite reproducir múltiples sonidos a la vez, y al cargar los sonidos en memoria se reproducen de manera rápida y reducen la latencia.

```
import 'package:soundpool/soundpool.dart';
```

Los efectos de sonido comienzan activados por defecto. Después, llama al método **_loadSettings(),** que carga el estado guardado en preferencias.

```
// Carga las configuraciones de música y efectos de sonido desde SharedPreferences
Future<void> _loadSettings() async {
 final prefs = await SharedPreferences.getInstance();
 // Carga el estado de la música, predeterminando en `true` si no está configurado
 _isPlayingMusic = prefs.getBool('isPlayingMusic') ?? true;
 // Carga el estado de los efectos de sonido, predeterminando en `true` si no está configurado
 _isSoundEffects = prefs.getBool('isSoundEffects') ?? true;
 // Si la música está habilitada, inicia la reproducción
 if (_isPlayingMusic) {
   await startMusic();
 }
 // Notifica a los widgets que el estado ha cambiado
 notifyListeners();
}
```

Para controlar el estado de los efectos de sonido, el switch de la pantalla de Settings llama al método **toggleSounds()** de **MusicState**. Este cambia el estado de los efectos, guarda el estado en SharedPreferences y notifica a los oyentes para que actualicen el estado.

```
child: Switch(
 // Estado del interruptor de efectos de sonido
 value: musicState.isSoundEffects,
 onChanged: (value) {
   // Cambia el estado de los efectos de sonido
```

```
    musicState.toggleSounds();
  },
),
```

```
// Alterna el estado de los efectos de sonido (activar/desactivar)
void toggleSounds() async {
 // Cambia el estado
 _isSoundEffects = !_isSoundEffects;
 // Guarda el nuevo estado en SharedPreferences
 await _saveSoundEffectsState();
 // Notifica a los widgets que el estado ha cambiado
 notifyListeners();
}
```

La reproducción de los efectos se controla desde **GameScreen**. Cuando se toca una columna, llama al método **tapSound**(). Esta función carga el archivo en memoria y lo reproduce, si los sonidos están activados. El efecto de **swipe** funciona exactamente igual. GameScreen llama a **swipeSound**(), y este hace que suene si está guardado para que suenen.

```
      void onColumnTap(int column) {
       ...
         if (placed) {
           // Incrementamos el contador de movimientos
           moves++;
           // Reproducimos sonido de toque
           musicState.tapSound();
      ...
      }
}
```

```
// Reproduce un sonido al tocar (si los efectos de sonido están activos)
Future<void> tapSound() async {
 // Crea un Soundpool para reproducir el efecto
 if (_isSoundEffects) {
   Soundpool pool = Soundpool(streamType: StreamType.notification);

   // Carga el archivo de sonido
   int soundId =
       await rootBundle.load("assets/click.mp3").then((ByteData soundData)
{
```

```
      // Carga el sonido en el Soundpool
      return pool.load(soundData);
    });
    // Reproduce el sonido
    int streamId = await pool.play(soundId);
  }
}
```

c. Música

El estado de la música también se guarda desde **MusicState**. Usa de nuevo **SharedPreferences** y **ChangeNotifier** para actualizar la aplicación. Sin embargo, para la música se ha escogido el uso de **AudioPlayers**, al no necesitar tanta rapidez de reproducción como con los efectos de sonido. Este paquete permite reproducir archivos de audio en aplicaciones móviles y web. Es una herramienta versátil que admite varios formatos de audio y diferentes fuentes, como archivos locales, URLs remotas, o incluso assets de la aplicación.

```
import 'package:audioplayers/audioplayers.dart';
```

La música comienza a sonar por defecto. En **_loadSettings()** carga las preferencias guardadas, y si la música está guardada como activada, ejecuta el método de **startMusic().** Esta función carga el archivo de la música, ajusta el volumen y la configura para que se reproduzca en bucle.

```
// Clase para manejar el estado de la música y efectos de sonido
class MusicState extends ChangeNotifier {
 // Reproductor de música principal
 final AudioPlayer _player = AudioPlayer();

 // Indica si la música está activa
 bool _isPlayingMusic = true;
...
MusicState() {
 // Cargar configuraciones desde SharedPreferences
 _loadSettings();
}
```

```
      // Carga las configuraciones de música y efectos de sonido desde
SharedPreferences
      Future<void> _loadSettings() async {
       final prefs = await SharedPreferences.getInstance();
       // Carga el estado de la música, predeterminando en `true` si no
está configurado
       _isPlayingMusic = prefs.getBool('isPlayingMusic') ?? true;
       // Carga el estado de los efectos de sonido, predeterminado en
`true` si no está configurado
       _isSoundEffects = prefs.getBool('isSoundEffects') ?? true;
       // Si la música está habilitada, inicia la reproducción
       if (_isPlayingMusic) {
         await startMusic();
       }

      // Verifica las configuraciones y comienza la música si está
habilitada
      Future<void> startMusicIfNeeded() async {
       // Asegura que las configuraciones estén cargadas
       await _loadSettings();
       // Inicia la música
       if (_isPlayingMusic) {
         await startMusic();
       }
      }
```

Desde el switch **'Music'** de Settings se controla el estado de la música. Si se presiona el interruptor, llama a **toggleMusic**() de **MusicState**. Este método pausa o reproduce la canción, cambia el estado y guarda la preferencia y notifica a las clases oyentes para que actualicen su estado.

```
      child: Switch(
       value: musicState.isPlayingMusic,
       onChanged: (value) {
         // Cambia el estado de la música
         musicState.toggleMusic();
       },
      ),
```

```
      // Alterna el estado de la música (activar/desactivar)
      void toggleMusic() async {
       // Pausa la música si está activa
       if (_isPlayingMusic) {
```

```
        await _player.pause();
      } // Reproduce la música si está desactivada
      else {
        await _player.play(AssetSource('background_music.mp3'));
      }
      // Cambia el estado
      _isPlayingMusic = !_isPlayingMusic;
      // Guarda el nuevo estado en SharedPreferences
      await _saveMusicState();
      // Notifica a los widgets que el estado ha cambiado
      notifyListeners();
    }
```

5. Conclusiones

Este juego inspirado en juegos como “2048” o ”Tetris”, desarrollado completamente en **Android Studio** con el uso de **Flutter**, presenta una experiencia completamente jugable, dinámica, inmersiva y personalizable. Gracias a su interfaz **intuitiva** y **responsiva** cada partida puede **personalizarse** profundamente cambiando el **color** de los elementos que están en juego o ajustando los niveles de **sonido** emitidos por la aplicación. La combinación de los **movimientos** posibles y el uso de la **gravedad** proporcionan una jugabilidad **estratégica** y constantemente cambiante, que con ayuda de los **elementos visuales** y **auditivos** sumergen al jugador en esta experiencia táctica, que retará al jugador a superarse constantemente para conseguir su mejor **puntuación** en el menor número de **movimientos** posibles. Todo esto completa un proyecto **sólido** y **atractivo** ante el mercado de los juegos casuales para dispositivos móviles.