Informe técnico - Prevención de Sueño en conductores con una aplicación Flutter implementando Google ML Kit en tiempo real

Estudiante: Miguel Alejandro Carrasco Céspedes

Materia: SIS-204 Programación de Dispositivos Móviles

Introducción

El presente informe técnico tiene como objetivo presentar el desarrollo de una aplicación móvil que previene el sueño en conductores, implementando Google ML Kit en tiempo real. La aplicación fue desarrollada en Flutter, un framework de código abierto creado por Google que permite el desarrollo de aplicaciones móviles para Android y iOS.

Desarrollo

Arquitectura de la aplicación

La aplicación final consiste en una página de bienvenda que explica el propósito de la aplicación y una breve revisión científica de cómo se usa el estadístico PERCLOS para detectar el sueño en una persona. Esta vista tiene un drawer que permite navegar a las tres vistas principales de la aplicación: la vista de la galería, la vista de la cámara y la vista de video en tiempo real. Cualquiera sea la opción escogida la ventana debería tener un funcionamiento similar. Dada la imagen cargada, capturada, o grabada, la aplicación debería ser capaz de detectar si la persona en la imagen está dormida o no.

El funcionamiento de fondo debe cargar los recursos necesarios: Las cámaras disponibles, el detector de rostros, y los recursos gráficos (imágenes y audio). La aplicación utilizará como proveedor de estados el patrón BLoC, que permite gestionar los estados mediante eventos y flujos de datos.

Gestión de Estados con BLoC

Se tienen definidos los siguientes BLoCs:

1. CameraBloc:

Se encarga de gestionar el estado de la cámara, permitiendo abrir y cerrar la cámara, y capturar una imagen o un stream de video.

Estos son los estados que maneja:

- `CameraLoadingState`: Indica que la cámara está cargando.
- `CameraReadyState`: Indica que la cámara está lista para ser usada. Provee un objeto `CameraController` que es el objeto clave para interactuar con la cámara.
- `CameraPictureTakenState`: Indica que se ha tomado una foto. Provee un string con la ruta de la imagen.

- `CameraStreamingState`: Indica que se está transmitiendo video. No provee información adicional, solo es un indicador.
- `CameraErrorState`: Indica que ha ocurrido un error. Provee un string con el mensaje de error.

Estos estados son manejados por el `CameraBloc` que utiliza 3 eventos para manejar la cámara:

- `_onInitializeCamera`: Emite un estado `CameraLoadingState` mientras inicializa la cámara elegida (frontal o trasera), se le debe indicar configuraciones de la cámara como la resolución y la orientación, al finalizar emite un estado `CameraReadyState` con el `CameraController` inicializado.

```
// Initialize the camera
_controller = CameraController(
        event.camera,
        ResolutionPreset.medium,
        imageFormatGroup: Platform.isAndroid
            ? ImageFormatGroup.nv21 // formato recomendado por ML Kit para Android
            : ImageFormatGroup.bgra8888, // formato recomendado por ML Kit para
iOS
        );
```

- `_onTakePicture`: Captura una única imagen y regresa la ruta de la imagen capturada.

```
final image = await _controller!.takePicture(); // Take a picture
emit(CameraPictureTakenState(image.path)); // Emit the path of the image
emit(CameraReadyState(_controller!)); // Emit the camera controller if it is still
open
```

- `_onStartImageStreamEvent`: Siempre que se tenga cargado el controlador, inicia un stream de video y emite un estado `CameraStreamingState`. La función que se ejecuta en el stream corresponde a la orquestación entre BLoCs:

```
await _controller!.startImageStream((CameraImage image) async {
    // This function is called every time a new frame is available
    // ...
    // Lots of code we'll see later
    // ...
    emit(CameraStreamingState(_controller!));
});
```

2. DetectorBloc:

Se encarga de gestionar el estado del detector de rostros, permitiendo cargar el detector, detectar rostros en una imagen y liberar el detector.

Estos son los estados que maneja:

- `DetectorLoadingState`: Indica que el detector está cargando.
- `DetectorModelLoadedState`: Indica que el modelo del detector está cargado.
- `DetectorResultState`: Indica que se ha detectado un rostro. Provee un objeto `Face` con la información del rostro detectado.
- `DetectorErrorState`: Indica que ha ocurrido un error. Provee un string con el mensaje de error.

Estos estados son manejados por el `DetectorBloc` que utiliza 3 eventos para manejar el detector:

- `_onInitializeModel`: Emite un estado `DetectorLoadingState` mientras carga el modelo del detector. Al finalizar emite un estado `DetectorModelLoadedState` con el modelo cargado. Los detalles de la configuración del modelo se discuten más abajo.

- `_onRunModel`: Siempre que se tenga cargado el modelo, detecta rostros en una imagen y emite un estado `DetectorResultState` con la información del rostro detectado.

```
// Run the model
List<Face> faces = await _faceDetector!.processImage(_inputImage!);
// Emit the results
if (!faces.isEmpty) emit(DetectorResultState(faces.first));
```

- `_onPickImageEvent`: Siempre que se tenga cargado el modelo, detecta rostros en una imagen cargada desde una ruta y emite un estado `DetectorResultState` con la información del rostro detectado. Su estructura es similar a `_onRunModel`.

Más adelante, se abordará la orquestación entre BLoCs. Para el retorno de resultados de la cámara al detector, y el display en pantalla.

3. ImageBloc:

Se encarga de gestionar el estado de la imagen estática, permitiendo cargar una imagen desde galería.

Estos son los estados que maneja:

- `ImageInitialState`: Indica que no se ha cargado ninguna imagen.
- `ImagePickedState`: Indica que se ha cargado una imagen. Provee un string con la ruta de la imagen.
- `ImageProcessedState`: Indica que se ha procesado la imagen. Provee un objeto `File` con la imagen procesada.
- `ImageErrorState`: Indica que ha ocurrido un error. Provee un string con el mensaje de error.

Estos estados son manejados por el `ImageBloc` que utiliza 4 eventos para manejar la imagen:

- `_onPickImage`: Abre la galería y selecciona una imagen. Al finalizar emite un estado `ImagePickedState` con la ruta de la imagen.

```
final pickedFile = await _picker.pickImage(source: ImageSource.gallery);// Pick an
image
    if (pickedFile != null) {
      final image = File(pickedFile.path);
      BlocProvider.of<DetectorBloc>
```

```
(event.context).add(PickImageDetectEvent(event.context, image)); // Detect the
image
      emit(ImagePickedState(image));
    } else {
        // manage error
    }
```

- `_onImageProcessed`: Procesa la imagen cargada y emite un estado `ImageProcessedState` con la imagen procesada.

```
final bytes = await event.image.readAsBytes();
final codec = await instantiateImageCodec(bytes);
final frame = await codec.getNextFrame();
final image = frame.image;
final width = image.width;
final height = image.height;
emit(ImageProcessedState(event.image, width, height));
```

- `_onImageError`: Emite un estado `ImageErrorState` con el mensaje de error.
- `_onImageReset`: Emite un estado `ImageInitialState` para resetear la imagen.

Orquestación

La orquestación comprende la corrdinación de eventos disparados entre los 3 BLoCs. Estos eventos deben originarse o bien dentro de la vista o desde otro BLoC. A continuación se describe el flujo de eventos para la vista de la stream de video.

1. **Validar la cámara**: Ni bien arranca la aplicación esta solicita permisos para acceder a la cámara. Si el permiso es concedido, se identifican las cámaras disponibles y se inicializa la cámara frontal, de ser posible.

```
// Ensure that plugin services are initialized so that `availableCameras()`
// can be called before `runApp()`
WidgetsFlutterBinding.ensureInitialized();

// Obtain a list of the available cameras on the device.
final cameras = await availableCameras();

// Get a specific camera from the list of available cameras.
final firstCamera = cameras[1];
```

2. **Iniciar el proveedor de BLoCs**: Se inician los BLoCs dentro de un `MultiBlocProvider` para que puedan ser accedidos desde cualquier parte de la aplicación.

```
runApp(MultiBlocProvider(
    providers: [
```

```
      BlocProvider(create: (context) => CameraBloc()),
      BlocProvider(create: (context) => DetectorBloc()),
      BlocProvider(create: (context) => ImageBloc()),
    ],
    child: MaterialApp(
      debugShowCheckedModeBanner: false,
      theme: ThemeData.dark(),
      home: MyHomePage(camera: firstCamera),
    ),
  ));
```

3. **Iniciar vista de stream de video**: Al iniciar la vista de stream de video, se disparan dos eventos: `_onInitializeCamera` y `_onInitializeModel` que inicializan la cámara y el detector de rostros, respectivamente.

```
context.read<CameraBloc>().add(InitializeCameraEvent(camera));
context.read<DetectorBloc>().add(InitializeModelEvent());
```

4. **Gestión de eventos durante el stream**: El evento `InitializeCameraEvent` debería resultar en un estado `CameraReadyState` que provee un `CameraController` que se utiliza para iniciar el stream de video. Una vez alcanzado este punto, un componente dinámico en la vista muestra distinta información dependiendo del estado de la cámara.

```
if (state is CameraLoadingState) {
    return const Center(child: CircularProgressIndicator()); //Loading
} else if (state is CameraReadyState) {
    // What to do when the camera is ready
} else if (state is CameraStreamingState) {
    // What to do when the camera is streaming
} else if (state is CameraErrorState) {
    // What to do when the camera has an error
} else {
    // Camera is not initialized
}
```

5. **Detección de rostros dentro del stream**: Una vez que la cámara está transmitiendo video, se detectan rostros en cada frame del video. Es importante destacar que cada frame debe ser su propio evento de detección. Por ello, se añade un evento `pickImageDetectEvent` en cada frame dentro del BloC de la cámara y no dentro del componente de la vista porque este estado depende de la velocidad de la cámara.

```
final detectorBloc = BlocProvider.of<DetectorBloc>(event.context);
bool isCapturing = false; // flag to avoid multiple captures on same event
await _controller!.startImageStream((CameraImage image) async {
    // This function is called every time a new frame is available
    if (isCapturing) return;
```

```
    isCapturing = true;
    try {
        var image = await _controller!.takePicture();
        detectorBloc.add(PickImageDetectEvent(event.context, File(image.path)));
    } catch (e) {
        // manage error
    } finally {
        isCapturing = false;
    }
});
```

Es importante destacar la asincronicidad de la función que se ejecuta en el stream. Pues si no se ha liberado el recurso de la cámara o el detector puede que se genere un error. Por ello, se utiliza un flag `isCapturing` para evitar que se capturen múltiples imágenes en un mismo frame. Así se mantiene el proceso fluido aún si no se puede correr el detector en todos los fotogramas.

6. **Procesamiento de resultados en el Detector**: El detector trabaja en base a la librería de Google ML Kit. Una vez que se detecta un rostro, evalúa si el rostro está dormido o no. Si el rostro está dormido, se emite un estado `DetectorErrorState` con un mensaje de advertencia. Caso contrario regresa un estado `DetectorResultState` con la información del rostro detectado.

```
_inputImage = InputImage.fromFile(event.image);

// Run the model
List<Face> faces = await _faceDetector!.processImage(_inputImage!);

// Add a ProcessedImageEvent to the ImageBloc
BlocProvider.of<ImageBloc>(event.context).add(ImageProcessedEvent(event.context,
event.image));

// Emit the results
if (faces.isEmpty) {
    // No faces detected
} else {
    if (!faces.isEmpty) {
        // Face(s) detected
        // Run subprocess to determine if the face is asleep
        // ...
        // Lots of code we'll see later
    }
}
```

Como se observa dentro de este procesamiento se debe llamar al `ImageBloc` para que procese la imagen y la muestre en pantalla. Además, se debe evaluar si el rostro está dormido o no. Para ello, se debe correr un subproceso que determine si el rostro está dormido, los detalles se encuentran en la próxima sección.

7. **Procesamiento de estados del Detector en la vista**: Una vez que se detecta (o no) un rostro, y el modelo ha determinado si el rostro está dormido o no, se actualiza la vista con la información correspondiente.

```
BlocBuilder<DetectorBloc, DetectorState>(
    builder: (context, state) {
        Widget detectorWidget = Container();
        if (state is DetectorResultState) {
            // What to do when a face is detected
        } else if (state is DetectorErrorState) {
            // What to do when...
            // ...an error occurs
            // ...no face is detected
            // ...the face is asleep
        } else {
            // Model is not initialized
        }
        return detectorWidget;
    },
)
```

El Widget `detectorWidget` es un componente dinámico que cambia dependiendo del estado del detector. Si se detecta un rostro, se muestra la información del rostro. Si no se detecta un rostro, se muestra un mensaje de error. Si el rostro está dormido, se muestra un mensaje de advertencia.

El resto de implmentaciones siguen una lógica similar y hasta más sencilla. Un componente dinámico en la vista responde a cambios en el estado y los BLoCs coordinan para que los eventos sigan un proceso coherente con la detección y exposición de resultados.

Subproceso de determinación de sueño con PERCLOS

La librería de Google ML Kit contiene un detector de rostros que provee información sobre la posición de los ojos y la probabilidad de que cada ojo esté abierto o cerrado. Para determinar si una persona está dormida o no, se utiliza el estadístico PERCLOS (Percentage of Eye Closure over Time). Este estadístico se calcula como el promedio de la probabilidad de que los ojos estén cerrados en un intervalo de tiempo. Si el valor de PERCLOS es mayor a un umbral, se considera que la persona está dormida.

En nuestro caso se está aproximando el estadístico con un contador sobre los frames del video. Cada frame que se detecta que los ojos están un 60% cerrados (entre ambos) se incrementa el contador. Si el contador supera un umbral, se considera que la persona está dormida. Igualmente, mientras los ojos estén por debajo del umbral, el contador se reduce.

```
if (faces.first.leftEyeOpenProbability == null ||
    faces.first.rightEyeOpenProbability == null) {
    // No eye probabilities error
    } else if ((faces.first.leftEyeOpenProbability! +
                faces.first.rightEyeOpenProbability!) <
                0.6) { // Eyes closed under threshold
        _alarmCount++;
        if (_alarmCount > 5) {
            // Face is asleep
            // Take an action
        }
```

```
    } else {
            _alarmCount--; // Eyes open
    }
```

Configuración inicial del detector

La librería ofrece diversas opciones para configurar el detector de rostros. En este caso, se ha configurado el detector con los siguientes parámetros. Aunque se pueden ajustar a necesidad:

```
FaceDetectorOptions options = FaceDetectorOptions(
    enableContours: true, // Detectará el contorno del rostro
    enableLandmarks: true, // Detectará puntos de referencia en el rostro
    enableClassification: true, // Clasificará el rostro (no necesario)
    enableTracking: true, // Seguirá el rostro en el video
);

_faceDetector = GoogleMlKit.vision.faceDetector(options);
```

Conclusiones

La aplicación desarrollada permite prevenir el sueño en conductores mediante la detección de rostros en tiempo real. La implementación de Google ML Kit en Flutter permite detectar rostros y determinar si una persona está dormida o no. La orquestación de eventos entre los BLoCs permite gestionar los estados de la cámara, el detector de rostros y la imagen de manera eficiente. El subproceso de determinación de sueño con PERCLOS permite calcular el estadístico en tiempo real y tomar acciones en consecuencia.