REPORT PROGETTO SISTEMI DIGITALI

Matteo Giri

INTRODUZIONE

L'applicazione realizzata nasce come un "prototipo" di strumento di supporto per persone non vedenti, in quanto il suo obiettivo principale è quello di analizzare l'ambiente catturato dalla fotocamera dello smartphone indicando all'utente la troppa vicinanza di un oggetto alla camera con un segnale audio che si attiva se l'oggetto identificato è vicino al telefono. Oltre a questo, l'utente può inviare comandi vocali all'applicazione tramite una pressione prolungata dello schermo, con i quali può attivare e disattivare il suono del segnale audio, aumentarne o diminuirne il volume e cambiare la distanza di "triggering" del segnale. Altre funzionalità dell'applicazione sono state aggiunte solo per scopi "accademici" (rimaste da una prima realizzazione dell'app che è stata poi cambiata), come la possibilità di effettuare una segmentazione semantica della scena ripresa dalla fotocamera e di costruire mappe di profondità dei frame della scena stessa.

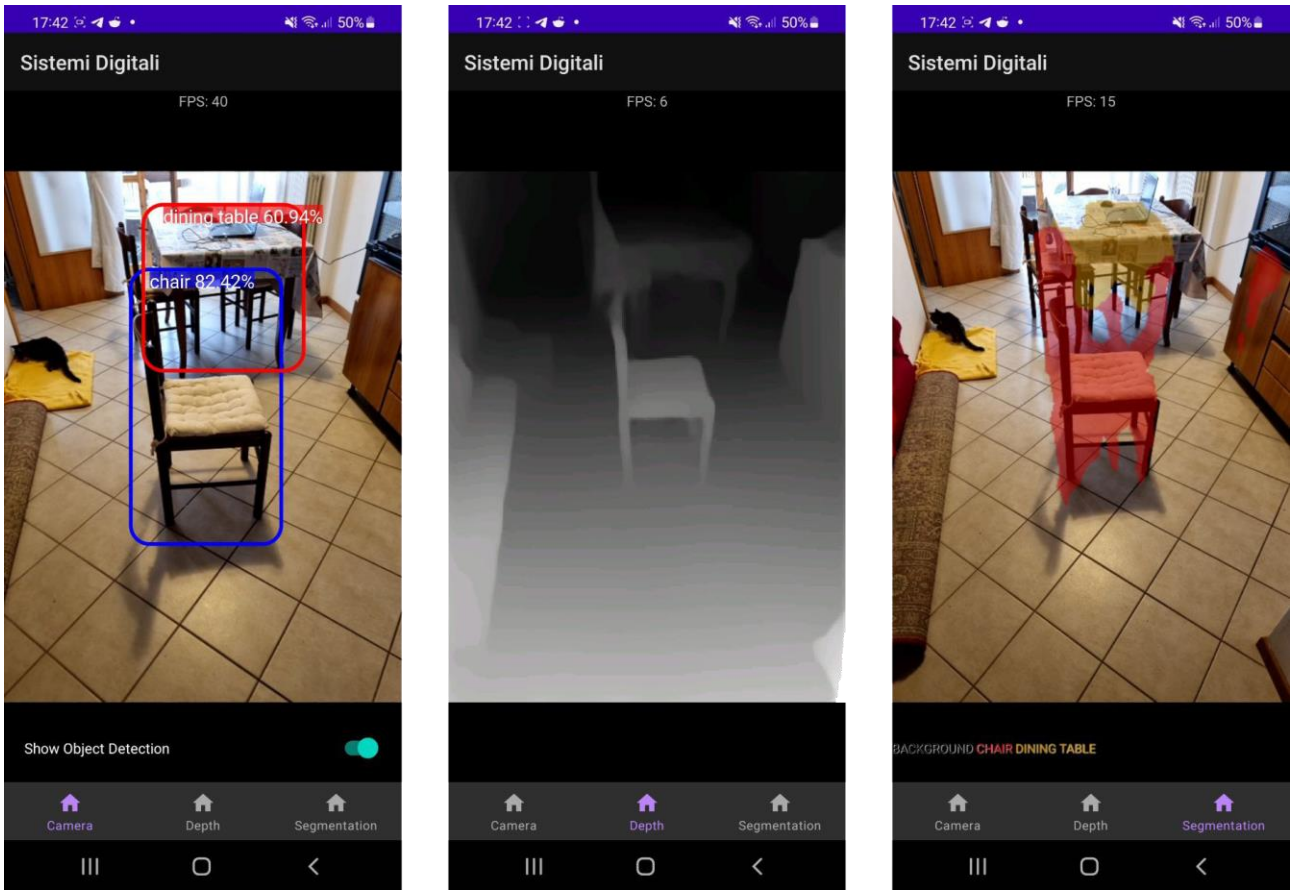

Figura 1: Da sinistra: schermata della fotocamera e object detection; schermata della depth estimation; schermata della scene segmentation

L'applicazione è composta da tre schermate, mostrate in Figura 1: Da sinistra: schermata della fotocamera e object detection; schermata della depth estimation; schermata della scene segmentation. La schermata principale è quella a sinistra, nella quale viene mostrato lo stream della fotocamera insieme alla possibilità

di mostrare i risultati del tracking della Object Detection, tecnica utilizzata per ottenere in maniera approssimativa la distanza di un oggetto dalla fotocamera. Nell'immagine centrale è contenuta la schermata relativa alla Depth Recognition, tecnica utilizzata per produrre mappe di profondità che vengono poi visualizzate a schermo come nella figura. Infine, nell'immagine a destra è mostrata la schermata relativa alla Scene Segmentation, nella quale oggetti di un tipo vengono evidenziati con un determinato colore, e la relativa etichetta viene visualizzata sotto.

La schermata relativa al riconoscimento vocale per impartire comandi all'applicazione è mostrata in Figura 2: Schermata di riconoscimento vocale.

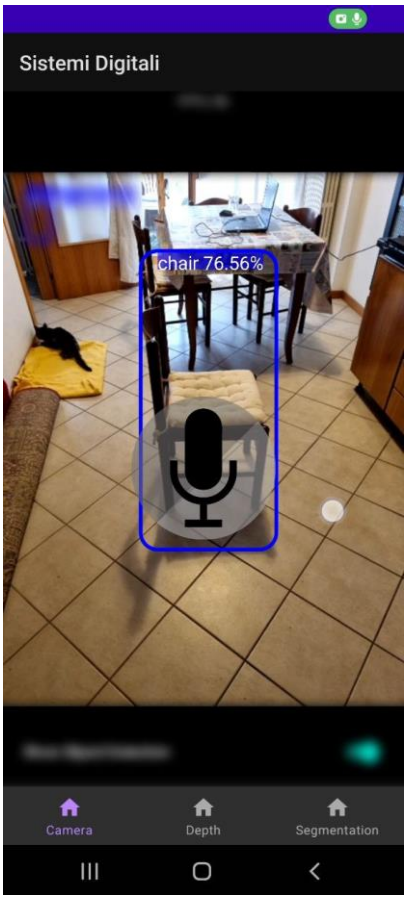

Figura 2: Schermata di riconoscimento vocale

Può essere attivata premendo in maniera prolungata sullo schermo, e l'attivazione e disattivazione del riconoscimento vocale sono anticipati da un suono specifico.

TECNOLOGIE UTILIZZATE

L'applicazione è stata realizzata su Android Studio utilizzando il linguaggio Kotlin. Tutta la parte di Machine Learning è stata gestita tramite il framework TensorFlow Lite per Android. In particolare, per l'**Object Detection** è stato utilizzato un modello pre-addestrato chiamato Single-Shot-Detector (SSD_MobileNet_v1.1), che prende in input un'immagine 300x300x3 e fornisce in output dei vettori che indicano posizione, classe, punteggio e numero di oggetti riconosciuti.

```
val model = DetectionModel.newInstance(context)

// Creates inputs for reference.
val image = TensorBuffer.createFixedSize(intArrayOf(1, 300, 300, 3), DataType.UINT8)
image.loadBuffer(byteBuffer)

// Runs model inference and gets result.
val outputs = model.process(image)
val location = outputs.locationAsTensorBuffer
val category = outputs.categoryAsTensorBuffer
val score = outputs.scoreAsTensorBuffer
val numberOfDetections = outputs.numberOfDetectionsAsTensorBuffer
```

Figura 3: Codice di esempio relativo al modello SSD_MobileNet_v1.1 per la Object Detection

Per la **Depth Estimation**, invece, è stato utilizzato il modello MiDas, che prende in input un'immagine 256x256x3 e fornisce in output la relative inverse depth map di dimensioni 256x256x1 dell'immagine passata in input.

```
val model = MidasDepth.newInstance(context)

// Creates inputs for reference.
val inputFeature0 = TensorBuffer.createFixedSize(intArrayOf(1, 256, 256, 3), DataType.FLOAT32)
inputFeature0.loadBuffer(byteBuffer)

// Runs model inference and gets result.
val outputs = model.process(inputFeature0)
val outputFeature0 = outputs.outputFeature0AsTensorBuffer
```

Figura 4: Codice di esempio relativo al modello MiDas per la Depth Estimation

Infine, per la **Semantic Scene Segmentation,** è stato utilizzato il modello DeepLab_v3.1, che prende in input un'immagine 257x257x3 e fornisce un output 257x257x21 relativo alle maschere create in base alle classi associate a ogni pixel dell'immagine fornita in input.

Tensors

Inputs

Name	Type	Description	Shape	Min / Max
image	Image <float32>	Input image to be segmented. The expected image is 257 x 257, with three channels (red, blue, and green) per pixel. Each element in the tensor is a value between -1 and 1.	[1, 257, 257, 3]	[-1] / [1]

Outputs

Name	Type	Description	Shape	Min / Max
segmentation masks	Feature <float32>	Masks over the target objects with high accuracy.	[1, 257, 257, 21]	[] / []

Sample Code

Kotlin | Java

```
val model = SegmentationModel.newInstance(context)

// Creates inputs for reference.
val image = TensorImage.fromBitmap(bitmap)

// Runs model inference and gets result.
val outputs = model.process(image)
val segmentationMasks = outputs.segmentationMasksAsCategoryList
```

Figura 5: Codice di esempio relativo al modello DeepLab_v3.1 per la Scene Segmentation

Tutti e tre i modelli descritti sono pre-addestrati e già quantizzati, in quanto sono stati pensati per essere utilizzati su dispositivi mobili e convertiti in modelli di TensorFlow Lite.

Per la realizzazione dei comandi vocali è stata utilizzata una tecnica di Speech Recognition fornita dalla API **SpeechRecognizer** di Android Studio.

Infine, per lo stream dei frame della fotocamera è stata utilizzata la API **Camera2** insieme a classi predefinite utilizzate in diverse applicazioni.

ARCHITETTURA DELL'APPLICAZIONE

La struttura dell'applicazione nelle sue directory è mostrata in Figura 6: Struttura generale delle directory dell'applicazione.

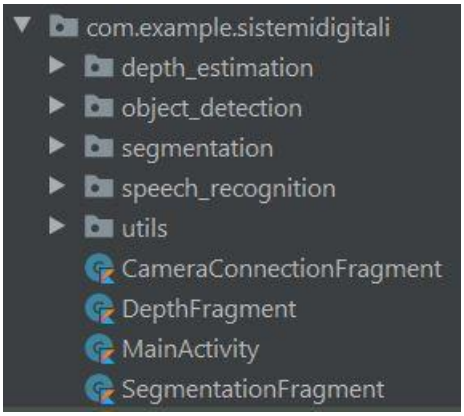

Figura 6: Struttura generale delle directory dell'applicazione

L'app è costituita da una singola activity (*MainActivity*) attraverso la quale vengono gestite e controllate tutte le funzionalità. Ad essa sono legati 3 fragments: quello relativo allo stream della camera e alla Object Detection (*CameraConnectionFragment*), quello relativo alla visualizzazione delle Depth Maps (*DepthFragment*), e quello relativo alla Scene Segmentation (*SegmentationFragment*). I tre fragments sono collegati e raggiungibili attraverso un Bottom Navigation Menu posto nella activity, come mostrato in Figura 7: Disposizione dei framgents

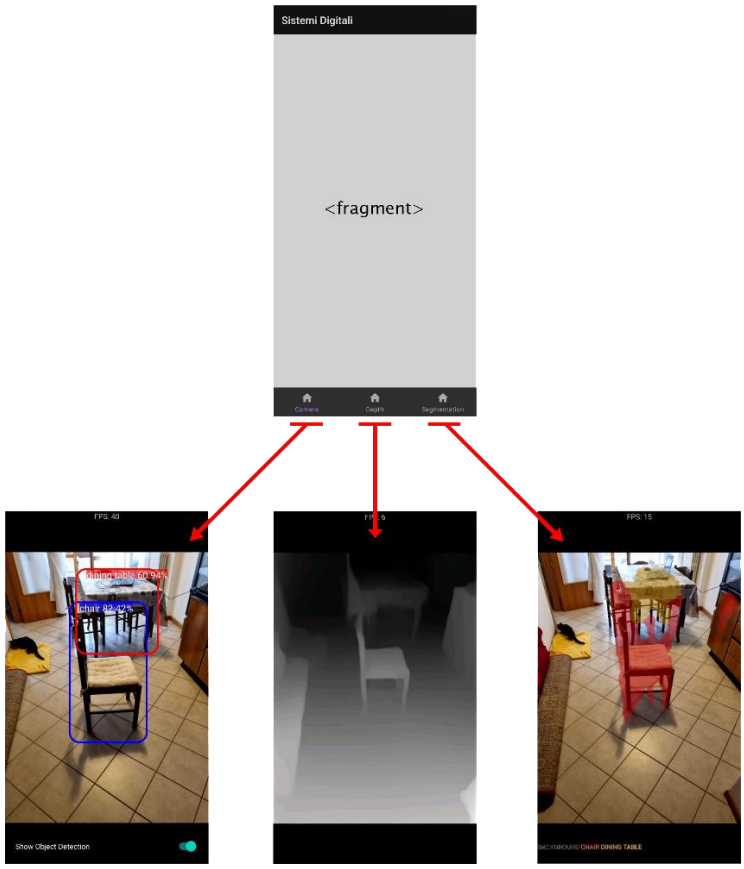

Figura 7: Disposizione dei framgents

Oltre all'activity principale e ai vari fragments sono presenti diverse cartelle relative alle singole funzionalità dell'applicazione.

La cartella "depth_estimation" contiene la helper class *MiDasModel*, nella quale sono definite tutte le funzioni adibite alla creazione del modello di Depth Estimation e al suo utilizzo per la generazione delle mappe di profondità.

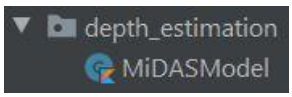

Figura 8: Contenuto della directory "depth_estimation"

La cartella "object_detection" contiene tutte le classi che riguardano la Object Detection. La classe *TFLiteObjectDetectionAPIModel* implementa l'interfaccia *Detector* e contiene le varie funzioni per la creazione del modello di Object Detection e al suo utilizzo. La classe *MultiBoxTracker* si occupa di mappare ogni risultato ottenuto dal modello di rilevamento oggetti a un box colorato che viene poi disegnato sulla vista *OverlayView* posta sopra allo stream di frame della fotocamera nel fragment *CameraConnectionFragment*. La classe *BorderedText* contiene il codice che permette di renderizzare correttamente il testo dei vari box nel canvas.

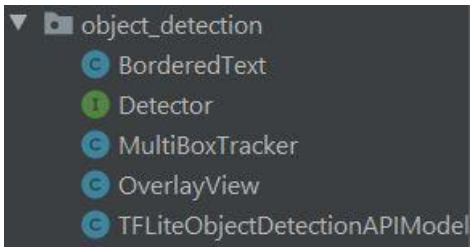

Figura 9: Contenuto della directory "object_detection"

La cartella “segmentation” contiene le classi relative alla Scene segmentation. La classe *ImageSegmentationModelExecutor* è la helper class che contiene le varie funzioni per creare il modello di Scene Segmentation e per il suo utilizzo. La classe *ModelExecutionResult* è una Data Class che modella il risultato ottenuto dalla segmentazione, e ha come attributi la bitmap dell’immagine originale passata in input, la bitmap della maschera di segmentazione ottenuta come risultato, la bitmap della sovrapposizione della maschera sull’immagine originale e l’insieme delle labels delle classi date ai vari pixels dell’immagine.

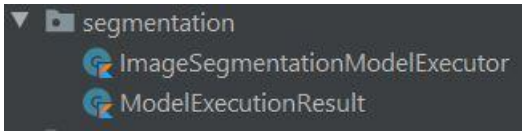

Figura 10: Contenuto della directory "segmentation"

La cartella “speech_recognition” contiene la classe *SpeechRecognitionListener* che sarebbe il listener che viene agganciato allo SpeechRecognizer e che permette di gestire le varie fasi della recognition.

Figura 11: Contenuto della directory "speech_recognition"

La cartella “utils” contiene alcune classi di utilità per l’applicazione. La classe *AutoFitTextureView* contiene una texture view custom che si adatta automaticamente alle dimensioni dello schermo. La classe *ImageUtils* contiene infine varie funzioni di utilità per fare operazioni sulle immagini.

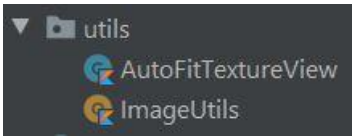

Figura 12: Contenuto della directory "utils"

DETTAGLI REALIZZATIVI

Aquisizione Immagini

Per quanto riguarda lo stream del flusso di frame della fotocamera, il compito è delegato al *CameraConnectionFragment*, che utilizza l'API Camera2 e l'ImageReader per creare delle sessioni per lo stream delle immagini. La classe è stata presa da un esempio di tensorflow lite ed è stata lasciata pressoché invariata.

La *MainActivity* implementa l'interfaccia di callback *ImageReader.OnImageAvailableListener*, che permette di chiamare il metodo "onImageAvailable" ogni volta che un nuovo frame della fotocamera viene acquisito dall'ImageReader. L'implementazione di onImageAvailable è mostrata in Figura 13: implementazione di onImageAvailable

```
override fun onImageAvailable(reader: ImageReader) {
    try {
        val image = reader.acquireLatestImage() ?: return
        if (isProcessingFrame) { //se un'altra immagine sta già venendo processata l'ultima immagine viene scartata
            image.close()
            return
        }
        isProcessingFrame = true
        val planes = image.planes
        ImageUtils.fillBytes(planes, yuvBytes)
        yRowStride = planes[0].rowStride
        val uvRowStride = planes[1].rowStride
        val uvPixelStride = planes[1].pixelStride
        imageConverter = Runnable {
            ImageUtils.convertYUV420ToARGB8888( yuvBytes[0]!!, yuvBytes[1]!!,yuvBytes[2]!!,previewWidth,previewHeight,
                    yRowStride, uvRowStride,uvPixelStride,rgbBytes!!)
        }
        postInferenceCallback = Runnable {
            image.close()
            isProcessingFrame = false
        }
        processImage()
        processForFragment()
    } catch (e: Exception) {
        Log.e( tag: "processing error", e.stackTrace.toString())
        return
    }
    checkSoundTrigger()
}
```

Figura 13: implementazione di onImageAvailable

L'immagine viene prima acquisita e poi viene controllato che non ci siano altri frame che stanno venendo processati nello stesso momento tramite il flag "isProcessingFrame". Se non ci sono altre immagini in processamento, viene costruito sull'immagine un imageConverter per trasformarla in una bitmap, che potrà essere utilizzata poi come input per i vari modelli di machine learning. L'effettiva conversione dell'immagine acquisita in bitmap viene effettuata nella funzione "processImage", mostrata in Figura 14: implementazione di processImage Nella funzione l'immagine viene prima convertita utilizzando l'imageConverter costruito in precedenza, viene poi ruotata (perché l'immagine viene acquisita con una rotazione sbagliata in alcuni telefoni) e salvata nella variabile "latest_Image_bitmap". Infine, viene

richiamato il callback che chiude l'immagine ormai processata e cambia il valore del flag "isProcessingFrame" per permettere a nuovi frame di essere processati.

```
private fun processImage() {
    imageConverter!!.run()
    latest_image_bitmap = Bitmap.createBitmap(
            previewWidth,
            previewHeight,
            Bitmap.Config.ARGB_8888
    )
    latest_image_bitmap.setPixels(rgbBytes,  offset: 0, previewWidth,  x: 0,  y: 0, previewWidth, previewHeight)
    latest_image_bitmap = rotateBitmap(latest_image_bitmap)
    postInferenceCallback!!.run()
}
```

Figura 14: implementazione di processImage

Infine viene chiamata la funzione "processForFragment" che in base al fragment in cui ci troviamo processa l'immagine secondo i vari modelli di machine learning.

La funzione "checkSoundTrigger" serve per attivare la coroutine del suono di vicinanza, e verrà mostrata nella sezione corrispondente.

Object Detection

Se ci troviamo nel fragment della Object Detection, ogni immagine trasformata in bitmap viene poi processata per effettuare il rilevamento degli oggetti. Il codice corrispondente nella funzione "processForFragment" è mostrato nella Figura 15: Codice corrispondente alla Object Detection nella funzione processForFragment

```
private fun processForFragment(){
    if (currentFragment == camera_fragment) {
        if (isDetectionProcessing == false) {
            CoroutineScope(Dispatchers.Main).launch {  this: CoroutineScope
                val startTime = SystemClock.elapsedRealtime()
                objectDetection(latest_image_bitmap)
                val endTime = SystemClock.elapsedRealtime()
                val time = endTime-startTime
                val fps = (1000.0 / time).roundToInt()
                frameText.setText("FPS: " + fps.toString())
            }
        }
    }
}
```

Figura 15: Codice corrispondente alla Object Detection nella funzione processForFragment

Tramite il flag "isDetectionProcessing" si controlla se non ci siano altri frame già in processamento per la Object Detection. In caso positivo viene creata una nuova coroutine Kotlin che si occuperà dell'effettivo processamento dell'immagine.

La funzione “objectDetection” è mostrata in Figura 16: Implementazione della funzione objectDetection

```
suspend fun objectDetection(bitmap: Bitmap){
    isDetectionProcessing = true
    trackingOverlay.postInvalidate()
    withContext(Dispatchers.Main) {  this: CoroutineScope
        val results: List<Detector.Recognition> = objectDetectorModel.recognizeImage(bitmap)
        var minimumConfidence: Float = MINIMUM_CONFIDENCE_TF_OD_API
        val mappedRecognitions: MutableList<Detector.Recognition> = ArrayList<Detector.Recognition>()
        val distances = ArrayList<Float>()
        for (result in results) {
            if (result.getConfidence() >= minimumConfidence) {
                mappedRecognitions.add(result)
                val distance = calculateDistance(result.Location)
                distances.add(distance)
            }
        }

        val minDistance = distances.minOrNull()
        soundTrigger = !(minDistance == null || minDistance > MINIMUM_DISTANCE_TRIGGER)

        //dà un valore alla frequenza del suono in base alla distanza minore
        if (minDistance != null && minDistance <= MINIMUM_DISTANCE_TRIGGER/3)
            frequence = Frequence.HIGH
        else if (minDistance != null && minDistance <= (MINIMUM_DISTANCE_TRIGGER*2)/3)
            frequence = Frequence.MEDIUM
        else
            frequence = Frequence.LOW

        //se lo switch della object detection è attivo il tracker processa i risultati e li mostra a schermo
        if (odSwitch.isChecked) {
            tracker.processResults(mappedRecognitions)
            trackingOverlay.postInvalidate()
        }
        else{
            val dummyList: MutableList<Detector.Recognition> = ArrayList<Detector.Recognition>()
            tracker.processResults(dummyList)
            trackingOverlay.postInvalidate()
        }
        isDetectionProcessing = false
    }
}
```

Figura 16: Implementazione della funzione objectDetection

Come prima cosa l’immagine viene passata all’oggetto “objectDetectorModel” (istanza della helper class della Object Detection), che la elabora tramite il modello di Object Detection e ritorna i risultati come lista di oggetti “Recognition” (data class definita nell’interfaccia *Detection*), che contengono titolo dell’etichetta dell’oggetto riconosciuto, confidenza con la quale l’oggetto è stato riconosciuto e posizione nell’immagine. Successivamente viene iterata la lista con un for per filtrare gli oggetti riconosciuti sopra una certa soglia di confidenza (sotto quella soglia l’oggetto viene scartato) e di questi oggetti ne calcola la distanza approssimata tramite la funzione “calculateDistance”, che verrà mostrata nella sezione corrispondente al calcolo della distanza. Infine, dopo alcuni passaggi relativi all’attivazione del suono di cui si parlerà in seguito, si richiama il tracker (oggetto della classe *MultiBoxTracker*), che tramite la funzione “processResults” crea i vari box colorati relativi a ogni oggetto riconosciuto, e li disegna nel trackingOverlay (OverlayView). In particolare, la funzione “processResults” del tracker prende la lista dei vari oggetti

riconosciuti, e li mappa a degli oggetti di tipo TrackedRecognition (data class definita all'interno della classe del tracker), che hanno come attributi la confidenza di detection, il rettangolo che ne indica la posizione, il titolo e il colore assegnato all'oggetto. È poi compito della funzione "draw" del tracker prendere la lista di oggetti TrackedRecognition e disegnarli concretamente sul canvas in base agli attributi dell'oggetto. Il "postInvalidate" viene eseguito per fare in modo che venga rieseguita la funzione "draw", in quanto il trackingOverlay ha impostato come callback la funzione del tracker (mostrato in Figura 17: Callback aggiunto al trackingOverlay per permettere al tracker di disegnare i box sul canvas).

```
trackingOverlay.addCallback(
        object : DrawCallback {
            override fun drawCallback(canvas: Canvas?) {
                tracker.draw(canvas)
            }
        })
```

Figura 17: Callback aggiunto al trackingOverlay per permettere al tracker di disegnare i box sul canvas

La funzione "recognizeImage" dell'objectDetectorModel è mostrata in Figura 18: Implementazione della funzione recognizeImage della classe TFLiteObjectDetectionAPIModel

```
@Override
public List<Recognition> recognizeImage(final Bitmap bitmap) {
  List<Detection> results = objectDetector.detect(TensorImage.fromBitmap(bitmap));

  //converte i risultati in oggetti della classe Recognition che vengono poi messi in un'ArrayList recognitions
  final ArrayList<Recognition> recognitions = new ArrayList<>();
  int cnt = 0;
  for (Detection detection : results) {
    recognitions.add(
        new Recognition(
            id: "" + cnt++,
            detection.getCategories().get(0).getLabel(),
            detection.getCategories().get(0).getScore(),
            detection.getBoundingBox()));
  }
  Trace.endSection(); // "recognizeImage"
  return recognitions;
}
```

Figura 18: Implementazione della funzione recognizeImage della classe TFLiteObjectDetectionAPIModel

Nella funzione la bitmap dell'immagine viene trasformata in un input tensor che viene poi passato al modello di Object Detection per effettuare il rilevamento. Dopo questo i vari risultati vengono mappati in una lista di oggetti Recognition che viene poi ritornata.

Depth Estimation

Se ci troviamo nel fragment della Depth Estimation, ogni immagine trasformata in bitmap viene poi processata per la creazione della mappa di profondità. Il codice corrispondente nella funzione "processForFragment" è mostrato nella Figura 19: Codice corrispondente alla Depth Estimation nella funzione processForFragment

Tramite il flag “isDepthProcessing” si controlla che non ci siano altri frame già in processamento per la Depth Estimation. In caso positivo viene creata una nuova coroutine Kotlin che si occuperà dell’effettivo processamento dell’immagine.

La mappa di profondità viene costruita tramite la funzione “depthEstimation” e mostrata a schermo tramite la funzione “changeImg” definita nel fragment della Depth Estimation, che semplicemente cambia la vecchia mappa in una image view con quella nuova appena costruita.

```
private fun processForFragment(){
    if (currentFragment == depth_fragment) {
        if (isDepthProcessing == false) {
            CoroutineScope(Dispatchers.Main).launch {  this: CoroutineScope
                val startTime = SystemClock.elapsedRealtime()
                val depthFrame = depthEstimation(latest_image_bitmap)
                depth_fragment.changeImg(depthFrame)
                val endTime = SystemClock.elapsedRealtime()
                val time = endTime-startTime
                val fps = (1000.0 / time).roundToInt()
                frameText.setText("FPS: " + fps.toString())
            }
        }
    }
}
```

Figura 19: Codice corrispondente alla Depth Estimation nella funzione processForFragment

La funzione “depthEstimation” è mostrata in Figura 20: Implementazione della funzione "depthEstimation"

```
suspend fun depthEstimation(bitmap: Bitmap):Bitmap{
    lateinit var scaled_depthMap : Bitmap
    withContext(Dispatchers.Main) {  this: CoroutineScope
        isDepthProcessing = true
        val depthMap = depthEstimationModel.getDepthMap(bitmap)
        scaled_depthMap = Bitmap.createScaledBitmap(depthMap,  dstWidth: 480,  dstHeight: 640,  filter: false)
        isDepthProcessing = false
    }
    return scaled_depthMap
}
```

Figura 20: Implementazione della funzione "depthEstimation"

In questa funzione la bitmap relativa all’immagine viene passata al metodo “getDepthMap” della helper class della Depth Estimation, che passa l’immagine al modello MiDas e ritorna una mappa di profondità, che viene poi scalata per renderla conforme alle dimensioni originali dell’immagine.

La funzione “getDepthMap” è implementata come mostrato in Figura 21: Implementazione della funzione "getDepthMap" della classe MiDasModel

Nella funzione viene creato un input tensor dalla bitmap di ingresso, che viene poi processato da un “inputTensorProcessor” che ridimensiona l’input e lo normalizza. Viene poi creato il tensore di output di dimensioni 256x256x1 sul quale viene scritta la mappa di profondità ottenuta dell’esecuzione del modello MiDas sul tensore di ingresso. Successivamente l’output tensor viene processato dall’ “outputTensorProcessor” che applica la trasformazione “min-max scaling” all’output e lo porta nel range 0-

255. Essendo l'output stato definito come un float32, viene infine convertito in bitmap attraverso la funzione di utilità "byteBufferToBitmap" contenuta nella classe *ImageUtils* e ritornata.

```
fun getDepthMap( inputImage : Bitmap ) : Bitmap {
    var inputTensor = TensorImage.fromBitmap( inputImage )

    val t1 = System.currentTimeMillis()
    inputTensor = inputTensorProcessor.process( inputTensor )

    // Output tensor of shape ( 256 , 256 , 1 ) and data type float32
    var outputTensor = TensorBufferFloat.createFixedSize(
            intArrayOf( inputImageDim , inputImageDim , 1 ) , DataType.FLOAT32 )

    // Perform inference on the MiDAS model
    interpreter.run( inputTensor.buffer, outputTensor.buffer )

    // Perform operations on the output tensor as described by `outputTensorProcessor`.
    outputTensor = outputTensorProcessor.process( outputTensor )

    Log.i( tag: "MIDAS",  msg: "MiDaS inference speed: ${System.currentTimeMillis() - t1}")

    // Create a Bitmap from the depth map which will be displayed on the screen.
    return ImageUtils.byteBufferToBitmap( outputTensor.floatArray , inputImageDim )
}
```

Figura 21: Implementazione della funzione "getDepthMap" della classe MiDasModel

Scene Segmentation

Se ci troviamo nel fragment della Depth Estimation, ogni immagine trasformata in bitmap viene poi processata per la creazione della mappa di profondità. Il codice corrispondente nella funzione "processForFragment" è mostrato nella Figura 19: Codice corrispondente alla Depth Estimation nella funzione processForFragment

```
private fun processForFragment(){
    if (currentFragment == segmentation_fragment) {
        if (isSegmentProcessing == false) {
            CoroutineScope(Dispatchers.Main).launch {  this: CoroutineScope
                val startTime = SystemClock.elapsedRealtime()
                val segmentedFrame = semanticSegmentation(latest_image_bitmap)
                segmentation_fragment.changeImg(segmentedFrame.first, segmentedFrame.second)
                val endTime = SystemClock.elapsedRealtime()
                val time = endTime-startTime
                val fps = (1000.0 / time).roundToInt()
                frameText.setText("FPS: " + fps.toString())
            }
        }
    }
}
```

Figura 22: Codice corrispondente alla Scene Segmentation nella funzione processForFragment

Tramite il flag "isSegmentProcessing" si controlla che non ci siano altri frame già in processamento per la Scene Segmentation. In caso positivo viene creata una nuova coroutine Kotlin che si occuperà dell'effettivo

processamento dell'immagine. La segmentazione viene effettuata tramite la funzione "semanticSegmentation", che ritorna un "Pair" contenente la bitmap dell'immagine processata con sovrapposta la maschera di segmentazione e l'insieme delle labels degli oggetti identificati. Questi due parametri vengono passati alla funzione "changeImg" definita nel fragment della Scene Segmentation, mostrata in Figura 23: Implementazione di "changeImg" e "setChipsToLogView" del fragment SegmentationFragment

```
fun changeImg(img: Bitmap, itemsFound: Map<String, Int>){
    imageview.setImageBitmap(img)
    if (!itemsFound.equals(chipsLabels)) {
        chipsLabels = itemsFound
        setChipsToLogView(itemsFound)
    }
}

private fun setChipsToLogView(itemsFound: Map<String, Int>) {
    var text = ""
    for ((label, color) in itemsFound) {
        text +=
            "<font color=" + color+ ">" + "   " +  label + "   " +  "</font>"
    }
    textView.setText(Html.fromHtml(text, flags: 0))
}
```

Figura 23: Implementazione di "changeImg" e "setChipsToLogView" del fragment SegmentationFragment

Nella funzione vediamo che la nuova immagine viene settata nella imageview e successivamente si controlla se le labels trovate sono le stesse di quelle trovate all'iterazione precedente (salvate in "chipsLabels"). Se non sono le stesse allora si richiama il metodo "setChipsToLogView", che per ogni label trovata crea un testo colorato con il colore specificato nella mappa "itemsFound" e infine le setta in una textView.

La funzione "semanticSegmentation" è implementata come in Figura 24: Implementazione della funzione "semanticSegmentation"

```
suspend fun semanticSegmentation(bitmap: Bitmap):Pair<Bitmap, Map<String, Int>>{
    lateinit var result: ModelExecutionResult
    withContext(Dispatchers.Main) {  this: CoroutineScope
        isSegmentProcessing = true
        result = imageSegmentationModel.execute(bitmap)
        isSegmentProcessing = false
    }
    return Pair(result.bitmapResult, result.itemsFound)
}
```

Figura 24: Implementazione della funzione "semanticSegmentation"

In questa funzione la bitmap relativa all'immagine vinee passata al metodo "execute" della helper class della Scene Segmentation, la quale ritorna un oggetto di tipo *ModelExecutionResults* (descritto nella sezione dell'architettura), del quale vengono poi ritornati la bitmap relativa all'immagine con sovrapposta la maschera di segmentazione e l'insieme delle labels degli oggetti trovati (con i rispettivi colori).

Il metodo “execute” dell’*ImageSegmentationModelExecutor* è mostrato nella Figura 25: Implementazione della funzione "execute" dell'ImageSegmentationModelExecutor

```
fun execute(inputImage: Bitmap): ModelExecutionResult {
  val tensorImage = TensorImage.fromBitmap(inputImage)
  val results = imageSegmenter.segment(tensorImage)

  val (maskBitmap, itemsFound) =
    createMaskBitmapAndLabels(results[0], inputImage.width, inputImage.height)

  return ModelExecutionResult(
    /*bitmapResult=*/ stackTwoBitmaps(maskBitmap, inputImage),
    /*bitmapOriginal=*/ inputImage,
    /*bitmapMaskOnly=*/ maskBitmap,
    formatExecutionLog(inputImage.width, inputImage.height),
    itemsFound
  )
}
```

Figura 25: Implementazione della funzione "execute" dell'ImageSegmentationModelExecutor

La bitmap dell’immagine da processare viene prima trasformata in un input tensor e poi processata dal modello di scene segmentation utilizzato. Il risultato del processamento viene passato alla funzione “createMaskBitmapAndLabels”, che si occupa di costruire la bitmap relativa alla machera di segmentazione e le varie labels. Infine viene creato un oggetto di tipo ModelExecutionResult che viene poi ritornato. Il metodo “stackTwoBitmaps” serve per creare la sovrapposizione della maschera di segmentazione all’immagine, mentre “formatExecutionLog” serve per il logging dei risultati ottenuti (tempo di esecuzione, dimensioni dell’immagine in input…).

La funzione “createMaskBitmapAndLabels” è mostrata in Figura 26: Implementazione della funzione "createMaskBitmapAndLabels" dell'ImageSegmentationModelExecutor

```
private fun createMaskBitmapAndLabels(result: Segmentation,inputWidth: Int,inputHeight: Int): Pair<Bitmap, Map<String, Int>> {
  val coloredLabels = result.coloredLabels
  val colors = IntArray(coloredLabels.size)
  var cnt = 0
  for (coloredLabel in coloredLabels) {
    val rgb = coloredLabel.argb
    colors[cnt++] = Color.argb(ALPHA_VALUE, Color.red(rgb), Color.green(rgb), Color.blue(rgb))
  }
  // Use completely transparent for the background color.
  colors[0] = Color.TRANSPARENT

  // Create the mask bitmap with colors and the set of detected labels.
  val maskTensor = result.masks[0]
  val maskArray = maskTensor.buffer.array()
  val pixels = IntArray(maskArray.size)
  val itemsFound = HashMap<String, Int>()
  for (i in maskArray.indices) {
    val color = colors[maskArray[i].toInt()]
    pixels[i] = color
    itemsFound.put(coloredLabels[maskArray[i].toInt()].getlabel(), color)
  }
  val maskBitmap = Bitmap.createBitmap(pixels,maskTensor.width,maskTensor.height,Bitmap.Config.ARGB_8888)
  // Scale the maskBitmap to the same size as the input image.
  return Pair(Bitmap.createScaledBitmap(maskBitmap, inputWidth, inputHeight,  filter: true), itemsFound)
}
```

Figura 26: Implementazione della funzione "createMaskBitmapAndLabels" dell'ImageSegmentationModelExecutor

In questa funzione per prima cosa vengono assegnate tutte le labels degli oggetti individuati alla variabile “coloredLabels” e impostati i vari colori nell’array “colors”. Il primo oggetto trovato è sempre l’oggetto “background” (ovvero lo sfondo), e a questo oggetto viene assegnato il colore “trasparente” per evitare che dia fastidio nella visualizzazione. Successivamente si passa alla creazione della maschera, assegnando ad ogni pixel della maschera (array “pixels”), il colore corrispondente all’oggetto trovato. Nella mappa “itemsFound” vengono inseriti tutte le labels con i rispettivi colori. Viene poi creata la bitmap della maschera con l’array di pixels creato e viene poi scalata e ritornata insieme alle etichette.

Approssimazione Distanza e Suono

Come visto nella sezione Object Detection e in particolare in Figura 16: Implementazione della funzione objectDetection, per ogni oggetto riconosciuto dal modello viene calcolata un’indicazione della distanza dell’oggetto dalla fotocamera tramite la funzione “calculateDistance”, mostrata in Figura 27: Implementazione della funzione "calculateDistance"

```
private fun calculateDistance(location: RectF): Float{
    val width = location.width()
    val height = location.height()
    val maxArea = 640 * 480
    val area = width * height
    val ratio: Float = if (area/maxArea <= 1) area/maxArea else 1F
    val distance = 1F - ratio
    return distance
}
```

Figura 27: Implementazione della funzione "calculateDistance"

La funzione prende come parametro il rettangolo che rappresenta la posizione dell’oggetto riconosciuto e ne calcola l’area. Successivamente si fa il rapporto tra l’area calcolata e l’area dell’intero spazio ripreso dalla fotocamera (640 x480). La distanza è calcolata come 1 – questo rapporto. Ovviamente il calcolo è molto approssimativo e la distanza ritornata non è una vera e proprio distanza, ma semplicemente un’indicazione

che la approssima andando a confrontare l'area del rettangolo con l'area totale. Più il rettangolo è grande e più l'oggetto verrà considerato vicino alla fotocamera.

Ottenute le varie distanze degli oggetti, si va a cercare la distanza minima rilevata e se quella distanza è più piccola di una certa soglia "MINIMUM_DISTANCE_TRIGGER", allora il valore del flag "soundTrigger" viene messo a True, altrimenti a False. Infine, si da un valore alla variabile "frequence", che rappresenta la frequenza con la quale il suono di distanza viene emesso. Più l'oggetto è considerato vicino e più il valore della frequenza risulterà elevato.

Ogni volta che è disponibile un'immagine, nella funzione "onImageAvailable" (Figura 13: implementazione di onImageAvailable) viene anche eseguita la funzione di controllo "checkSoundTrigger". Questa funzione (Figura 28: Implementazione della funzione "checkSoundTrigger") controlla che il flag "soundTrigger" sia True (ovvero che c'è un oggetto abbastanza vicino da far attivare il suono) e che il flag "isSoundPlaying" sia False (ovvero che non ci sia già il suono attivo). In caso positivo si fa partire la coroutine "playSound".

```
private fun checkSoundTrigger(){
    if (soundTrigger && !isSoundPlaying){
        soundJob = CoroutineScope(newSingleThreadContext( name: "SoundThread")).launch{ playSound()}
    }
}
```

Figura 28: Implementazione della funzione "checkSoundTrigger"

Il codice relativo alla coroutine è mostrato in Figura 29: Implementazione della funzione "playSound"

```
private suspend fun playSound(){
    isSoundPlaying = true
    while(soundTrigger){
        if (soundVolume) {
            when (frequence) {
                Frequence.HIGH -> {
                    alarmSound.playbackParams = alarmSound.playbackParams.setSpeed(2.0f)
                    alarmSound.start()
                    delay( timeMillis: 50)
                }
                Frequence.MEDIUM -> {
                    alarmSound.playbackParams = alarmSound.playbackParams.setSpeed(1.5f)
                    alarmSound.start()
                    delay( timeMillis: 500)
                }
                Frequence.LOW -> {
                    alarmSound.playbackParams = alarmSound.playbackParams.setSpeed(1.0f)
                    alarmSound.start()
                    delay( timeMillis: 1000)
                }
            }
        }
    }
    isSoundPlaying = false
    Log.i( tag: "speech recognizer",  msg: "Sound Coroutine canceled")
}
```

Figura 29: Implementazione della funzione "playSound"

Finché "sound Trigger" è a True e se l'audio non è mutato (soundVolume == True) si fa partire il suono che indica la distanza, con una frequenza diversa in base al valore della variabile "frequence".

Speech Recognition

Il riconoscimento di comandi vocali si basa, come introdotto nelle sezioni precedenti, sull'API Speech Recognizer di Android Studio. Nella Figura 30: inizializzazione variabili per la speech recognition sono mostrate le fasi di inizializzazione degli oggetti dell'API.

```
speechRecognizer = SpeechRecognizer.createSpeechRecognizer( context: this)
val speechRecognizerIntent = Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH)
speechRecognizerIntent.putExtra(
        RecognizerIntent.EXTRA_LANGUAGE_MODEL,
        RecognizerIntent.LANGUAGE_MODEL_FREE_FORM
)
speechRecognizerIntent.putExtra(RecognizerIntent.EXTRA_LANGUAGE,  value: "it-IT")

speechRecognizer.setRecognitionListener(SpeechRecognitionListener( mainA: this))
```

Figura 30: inizializzazione variabili per la speech recognition

Per prima cosa si crea uno speech recognizer usando la funzione "createSpeechRecognizer" dell'API. Successivamente si crea un intent per l'operazione da svolgere, al quale viene aggiunto il linguaggio italiano come parametro extra. Dopo di questo si deve settare un listener allo speech recognizer, in particolare un'istanza della classe "SpeechRecognitionListener", mostrata in Figura 31: Implementazione della classe SpeechRecognitionListener

```
class SpeechRecognitionListener constructor(mainA: MainActivity) : RecognitionListener{

    private val main = mainA
    override fun onReadyForSpeech(bundle: Bundle) {}
    override fun onBeginningOfSpeech() {
        Log.i( tag: "speech recogonizer",  msg: "beginning to record")
    }

    override fun onRmsChanged(v: Float) {}
    override fun onBufferReceived(bytes: ByteArray) {}
    override fun onEndOfSpeech() {}
    override fun onError(i: Int) {}
    override fun onResults(bundle: Bundle) {
        val data = bundle.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION)
        if (data != null) {
            Log.i( tag: "speech recogonizer", data.toString())
        }
        data?.get(0)?.let { main.executeCommand(it.toString()) }
    }

    override fun onPartialResults(bundle: Bundle) {}
    override fun onEvent(i: Int, bundle: Bundle) {}
}
```

Figura 31: Implementazione della classe SpeechRecognitionListener

In questa classe vengono definite le operazioni che devono essere svolte nelle varie fasi della Speech Recognition. Nel caso di questa applicazione è di interesse solo il callback "onResults" che viene richiamato quando lo speech recognizer ha riconosciuto una parola o frase dettata. Alla funzione viene passato un bundle, dal quale possiamo estratte la frase riconosciuta ed eseguire poi il comando associato definito nella funzione "executeCommand" della MainActivity (Figura 32: Implementazione della funzione "executeCommand").

```
fun executeCommand(data: String){
    if (data.contains( other: "disattiva allarme", ignoreCase = true)){
        soundVolume = false
    }
    else if (data.contains( other: "attiva allarme", ignoreCase = true)){
        soundVolume = true
    }
    else if (data.contains( other: "aumenta volume", ignoreCase = true)){
        volume += 0.1f
        alarmSound.setVolume(volume, volume)
    }
    else if (data.contains( other: "diminuisci volume", ignoreCase = true)){
        volume -= 0.1f
        alarmSound.setVolume(volume, volume)
    }
    else if (data.contains( other: "aumenta distanza", ignoreCase = true)){
        MINIMUM_DISTANCE_TRIGGER += 0.05f
    }
    else if (data.contains( other: "diminuisci distanza", ignoreCase = true)){
        MINIMUM_DISTANCE_TRIGGER -= 0.05f
    }
}
```

Figura 32: Implementazione della funzione "executeCommand"

Per attivare il riconoscimento dei comandi vocali si deve effettuare una pressione prolungata sullo schermo. Il codice che mostra i passaggi relativi a questa operazione è definito in Figura 33: Codice relativo all'attivazione della Speech Recognition

```
val handler = Handler(Looper.getMainLooper()) //handler per realizzare la gesture del long press
var longPress = false
val runnable = Runnable { // runnable eseguito quando si fa un long press sullo schermo
    longPress = true
    Log.i( tag: "speech recogonizer",  msg: "I'M PRESSING")
    speechRecognizer.startListening(speechRecognizerIntent)
    soundIn.start()
    val anim = AlphaAnimation(0.0f, 1.0f)
    anim.duration = 1000
    hiddenView.setVisibility(View.VISIBLE)
    hiddenView.setAlpha(1F)
    hiddenView.startAnimation(anim)
}
pressableView.setOnTouchListener(object : View.OnTouchListener {
    override fun onTouch(view: View?, motionEvent: MotionEvent): Boolean {
        if (motionEvent.action == MotionEvent.ACTION_UP) {
            handler.removeCallbacks(runnable)
            if (longPress) {
                longPress = false
                Log.i( tag: "speech recogonizer",  msg: "I RELEASED")
                speechRecognizer.stopListening()
                soundOut.start()
                val anim = AlphaAnimation(hiddenView.alpha, 0.0f)
                anim.duration = 300
                hiddenView.startAnimation(anim)
                hiddenView.setVisibility(View.GONE)
            }
            return false
        } else if (motionEvent.action == MotionEvent.ACTION_DOWN) {
            //esegue il runnable solo dopo un secondo, in questo modo si può realizzare un long touch
            handler.postDelayed(runnable,  delayMillis: 1000);
            return true
        }
        return false
    }
})
```

Figura 33: Codice relativo all'attivazione della Speech Recognition

Per realizzare la gesture del “Long press” si utilizza un thread handler, che permette di eseguire un certo codice dopo che sia passato un determinato periodo di tempo. In particolare, come si vede in figura, viene creato il thread handler e viene costruito del codice Runnable che quando eseguito mette in ascolto lo speech recognizer sull’intent creato. Nel layout della MainActivity è presente una “pressableView”, che ricopre l’intero schermo e alla quale è associato un OnTouchListener. Questo listener definisce le operazioni da svolgere quando si appoggia il dito sulla view (MotionEvent.ACTION_DOWN) e quando si toglie il dito dalla view (MotionEvent.ACTION_UP). Se l’evento è “ACTION_DOWN” allora si esegue il metodo “postDelayed” dell’handler che mette in esecuzione il codice scritto nel Runnable dopo un certo periodo di tempo. Se invece l’evento è “ACTION_UP” allora lo speech recognizer smette di ascoltare.