**Capítulo 2 - OpenCV**

Para averiguar las versiones de las bibliotecas necesarias:

<https://developer.android.com/jetpack/androidx/releases/camera?hl=es-419>

<https://google.github.io/accompanist/permissions/>

<https://github.com/google/accompanist>

*// Accompanist*

*implementation* ("com.google.accompanist:accompanist-permissions:0.32.0")

*implementation* ("com.google.accompanist:accompanist-systemuicontroller:0.32.0")

*// CameraX*

*implementation* ("androidx.camera:camera-camera2:1.4.1")

*implementation* ("androidx.camera:camera-lifecycle:1.4.1")

*implementation* ("androidx.camera:camera-view:1.4.1")

*// OpenCV*

*implementation* ("org.opencv:opencv:4.10.0")

Pulsar **Sync now** para sincronizarlo.

**AndroidManifest.xml:**

*<?*xml version="1.0" encoding="utf-8"*?>*<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools">  
  
 <uses-permission android:name="android.permission.CAMERA" />  
 <uses-feature android:name="android.hardware.camera" android:required="true" />  
  
 <application  
 android:allowBackup="true"  
 android:dataExtractionRules="@xml/data\_extraction\_rules"  
 android:fullBackupContent="@xml/backup\_rules"  
 android:icon="@mipmap/ic\_launcher"  
 android:label="@string/app\_name"  
 android:roundIcon="@mipmap/ic\_launcher\_round"  
 android:supportsRtl="true"  
 android:theme="@style/Theme.Capitulo2\_practica1"  
 tools:targetApi="31">  
 <activity  
 android:name=".MainActivity"  
 android:exported="true"  
 android:label="@string/app\_name"  
 android:theme="@style/Theme.Capitulo2\_practica1">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN" />  
  
 <category android:name="android.intent.category.LAUNCHER" />  
 </intent-filter>  
 </activity>  
 </application>  
  
</manifest>

**YuvToRgbConverter.kt**

package com.example.capitulo2\_practica1  
  
import android.content.Context  
import android.graphics.Bitmap  
import androidx.camera.core.ImageProxy  
import android.graphics.YuvImage  
import android.graphics.Rect  
import java.io.ByteArrayOutputStream  
import android.graphics.BitmapFactory  
import android.graphics.Canvas  
import android.graphics.ImageFormat  
  
  
class YuvToRgbConverter(private val context: Context) {  
 @Synchronized  
 fun yuvToRgb(image: ImageProxy, output: Bitmap) {  
 try {  
 *// Obtener los planos de la imagen* val planes = image.*planes* val yBuffer = planes[0].*buffer* val uBuffer = planes[1].*buffer* val vBuffer = planes[2].*buffer* val ySize = yBuffer.remaining()  
 val uSize = uBuffer.remaining()  
 val vSize = vBuffer.remaining()  
  
 *// Crear nv21 con los datos YUV* val nv21Buffer = ByteArray(ySize + uSize + vSize)  
 yBuffer.get(nv21Buffer, 0, ySize)  
 vBuffer.get(nv21Buffer, ySize, vSize)  
 uBuffer.get(nv21Buffer, ySize + vSize, uSize)  
  
 *// Crear YUV Image y hacer la conversión* val yuvImage = YuvImage(  
 nv21Buffer,  
 ImageFormat.*NV21*,  
 image.*width*,  
 image.*height*,  
 null  
 )  
  
 *// Comprimir a JPEG temporalmente para mantener el color* val out = ByteArrayOutputStream()  
 yuvImage.compressToJpeg(  
 Rect(0, 0, image.*width*, image.*height*),  
 100,  
 out  
 )  
  
 *// Convertir JPEG de vuelta a bitmap* val jpegBytes = out.toByteArray()  
 val tempBitmap = BitmapFactory.decodeByteArray(  
 jpegBytes,  
 0,  
 jpegBytes.size,  
 BitmapFactory.Options().*apply* **{** inPreferredConfig = Bitmap.Config.*ARGB\_8888* **}** )  
  
 *// Dibujar en el bitmap de salida* val canvas = Canvas(output)  
 canvas.drawBitmap(tempBitmap, 0f, 0f, null)  
  
 *// Limpiar* tempBitmap.recycle()  
 out.close()  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 }  
}

**MainScreen.kt**:

package com.example.capitulo2\_practica1  
  
import android.graphics.Bitmap  
import android.graphics.BitmapFactory  
import androidx.camera.core.CameraSelector  
import androidx.camera.core.ImageAnalysis  
import androidx.camera.core.Preview  
import androidx.camera.lifecycle.ProcessCameraProvider  
import androidx.camera.view.PreviewView  
import androidx.compose.foundation.layout.fillMaxSize  
import androidx.compose.material3.AlertDialog  
import androidx.compose.material3.Text  
import androidx.compose.material3.TextButton  
import androidx.compose.runtime.Composable  
import androidx.compose.runtime.DisposableEffect  
import androidx.compose.runtime.LaunchedEffect  
import androidx.compose.runtime.getValue  
import androidx.compose.runtime.mutableStateOf  
import androidx.compose.runtime.remember  
import androidx.compose.runtime.setValue  
import androidx.compose.ui.platform.*LocalContext*import androidx.compose.ui.platform.*LocalLifecycleOwner*import androidx.compose.ui.viewinterop.AndroidView  
import androidx.core.content.ContextCompat  
import com.google.accompanist.permissions.ExperimentalPermissionsApi  
import com.google.accompanist.permissions.PermissionStatus  
import com.google.accompanist.permissions.rememberPermissionState  
import org.opencv.android.Utils  
import org.opencv.core.Mat  
import org.opencv.core.MatOfRect  
import org.opencv.core.Point  
import org.opencv.core.Scalar  
import org.opencv.imgproc.Imgproc  
import org.opencv.objdetect.CascadeClassifier  
  
import androidx.compose.ui.Modifier  
import java.io.File  
import androidx.compose.foundation.layout.Box  
import android.graphics.Matrix  
import android.graphics.ImageFormat  
import android.graphics.Rect  
import android.graphics.YuvImage  
import java.io.ByteArrayOutputStream  
  
  
*// Indica que se está utilizando una API experimental de permisos*@OptIn(ExperimentalPermissionsApi::class)  
*// Define un componente composable llamado MainScreen*@Composable  
fun MainScreen() {  
 *// Obtiene el contexto actual de la aplicación* val context = *LocalContext*.current  
  
 *// Crea un estado para gestionar el permiso de la cámara  
 // rememberPermissionState mantiene el estado entre recomposiciones* val cameraPermissionState = rememberPermissionState(  
 android.Manifest.permission.*CAMERA* )  
  
 *// Evalúa el estado actual del permiso de la cámara* when (cameraPermissionState.status) {  
 *// Si el permiso está concedido, muestra la vista previa de la cámara* is PermissionStatus.Granted -> {  
 CameraPreview()  
 }  
  
 *// Si el permiso está denegado, maneja dos escenarios posibles* is PermissionStatus.Denied -> {  
 *// Verifica si se debe mostrar una explicación al usuario sobre por qué  
 // se necesita el permiso (esto ocurre cuando el usuario ha denegado  
 // el permiso anteriormente)* if ((cameraPermissionState.status as PermissionStatus.Denied).shouldShowRationale) {  
 *// Muestra un diálogo explicativo con la opción de solicitar el permiso* RationaleDialog(  
 onConfirm = **{** cameraPermissionState.launchPermissionRequest() **}** )  
 } else {  
 *// Si es la primera vez que se solicita el permiso,  
 // lanza directamente la solicitud* LaunchedEffect(Unit) **{** cameraPermissionState.launchPermissionRequest()  
 **}** }  
 }  
 }  
}  
  
@Composable  
fun CameraPreview() {  
 *// Obtiene referencias esenciales para la cámara y el ciclo de vida* val context = *LocalContext*.current  
 val lifecycleOwner = *LocalLifecycleOwner*.current  
  
 *// Inicializa el proveedor de la cámara  
 // remember preserva el valor durante las recomposiciones* val cameraProviderFuture = remember **{** ProcessCameraProvider.getInstance(context) **}** *// Declara variables de estado mutables para los componentes principales  
 // Estas variables pueden actualizarse y provocar recomposiciones* var frameProcessor by remember **{** *mutableStateOf*<ImageAnalysis.Analyzer?>(null) **}** var previewView by remember **{** *mutableStateOf*<PreviewView?>(null) **}** var imageView by remember **{** *mutableStateOf*<android.widget.ImageView?>(null) **}** *// DisposableEffect gestiona la configuración y limpieza del clasificador de ojos  
 // Unit como clave significa que este efecto solo se ejecuta una vez* DisposableEffect(Unit) **{** *// Inicializa el clasificador en cascada para la detección de ojos* val eyesCascade = CascadeClassifier().*apply* **{** try {  
 *// Crea un archivo temporal para el clasificador XML* val cascadeFile = File(context.*cacheDir*, "haarcascade\_eye.xml")  
  
 *// Copia el archivo del clasificador desde los recursos raw  
 // al almacenamiento temporal de la aplicación  
 // .use asegura que los recursos se cierren correctamente* context.*resources*.openRawResource(R.raw.*haarcascade\_eye*).*use* **{** input **->** cascadeFile.*outputStream*().*use* **{** output **->** input.*copyTo*(output)  
 **}  
 }** *// Carga el clasificador desde el archivo temporal* load(cascadeFile.*absolutePath*)  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 **}** frameProcessor = ImageAnalysis.Analyzer **{** imageProxy **->** try {  
 *// 1. Crear un bitmap con el tamaño correcto* val bitmap = Bitmap.createBitmap(  
 imageProxy.*width*,  
 imageProxy.*height*,  
 Bitmap.Config.*ARGB\_8888* )  
  
 *// 2. Extraer los planos YUV* val yBuffer = imageProxy.*planes*[0].*buffer* val uBuffer = imageProxy.*planes*[1].*buffer* val vBuffer = imageProxy.*planes*[2].*buffer  
  
 // 3. Calcular tamaños y strides* val ySize = imageProxy.*width* \* imageProxy.*height* val uvSize = (imageProxy.*width* \* imageProxy.*height*) / 4  
  
 *// 4. Crear buffer NV21* val nv21Buffer = ByteArray(ySize + uvSize \* 2)  
  
 *// Copiar Y* yBuffer.get(nv21Buffer, 0, ySize)  
  
 *// Copiar UV* var uIndex = 0  
 var vIndex = 0  
 val uvPixelStride = imageProxy.*planes*[1].*pixelStride* val uvRowStride = imageProxy.*planes*[1].*rowStride* val width = imageProxy.*width* / 2  
 val height = imageProxy.*height* / 2  
 val uvPos = ySize  
  
 for (row in 0 *until* height) {  
 for (col in 0 *until* width) {  
 nv21Buffer[uvPos + (row \* width + col) \* 2] = vBuffer.get(vIndex) *// V antes que U* nv21Buffer[uvPos + (row \* width + col) \* 2 + 1] = uBuffer.get(uIndex) *// U después de V* uIndex += uvPixelStride  
 vIndex += uvPixelStride  
 }  
 uIndex += uvRowStride - width \* uvPixelStride  
 vIndex += uvRowStride - width \* uvPixelStride  
 }  
  
 *// 5. Convertir a YuvImage y luego a JPEG* val yuvImage = YuvImage(  
 nv21Buffer,  
 ImageFormat.*NV21*,  
 imageProxy.*width*,  
 imageProxy.*height*,  
 null  
 )  
  
 val out = ByteArrayOutputStream()  
 yuvImage.compressToJpeg(  
 Rect(0, 0, imageProxy.*width*, imageProxy.*height*),  
 100,  
 out  
 )  
  
 *// 6. Convertir JPEG a Bitmap* val jpegBytes = out.toByteArray()  
 val colorBitmap = BitmapFactory.decodeByteArray(  
 jpegBytes,  
 0,  
 jpegBytes.size,  
 BitmapFactory.Options().*apply* **{** inPreferredConfig = Bitmap.Config.*ARGB\_8888* **}** )  
  
 *// 7. Rotar el bitmap* val matrix = Matrix()  
 matrix.postRotate(90f)  
 val rotatedBitmap = Bitmap.createBitmap(  
 colorBitmap,  
 0,  
 0,  
 colorBitmap.*width*,  
 colorBitmap.*height*,  
 matrix,  
 true  
 )  
  
 *// 8. Convertir a Mat manteniendo el color* val colorMat = Mat()  
 Utils.bitmapToMat(rotatedBitmap, colorMat)  
  
 *// Convertir de BGR a RGB* Imgproc.cvtColor(colorMat, colorMat, Imgproc.*COLOR\_BGR2RGB*)  
  
 *// 9. Crear una copia en escala de grises para la detección* val grayMat = Mat()  
 Imgproc.cvtColor(colorMat, grayMat, Imgproc.*COLOR\_RGB2GRAY*)  
  
 *// 10. Detectar ojos en escala de grises* val eyes = MatOfRect()  
 eyesCascade.detectMultiScale(  
 grayMat,  
 eyes,  
 1.1,  
 4,  
 0,  
 org.opencv.core.Size(20.0, 20.0),  
 org.opencv.core.Size()  
 )  
  
 *// 11. Dibujar rectángulos en la imagen a color* val eyeArray = eyes.toArray()  
 eyeArray.*forEach* **{** eye **->** Imgproc.rectangle(  
 colorMat,  
 Point(eye.x.toDouble(), eye.y.toDouble()),  
 Point(  
 (eye.x + eye.width).toDouble(),  
 (eye.y + eye.height).toDouble()  
 ),  
 Scalar(255.0, 0.0, 0.0), *// RGB: Rojo* 4  
 )  
 **}** *// Convertir de nuevo a BGR antes de convertir a bitmap* Imgproc.cvtColor(colorMat, colorMat, Imgproc.*COLOR\_RGB2BGR*)  
  
 *// 12. Convertir de vuelta a bitmap* Utils.matToBitmap(colorMat, rotatedBitmap)  
  
 *// 13. Mostrar resultado* imageView?.post **{** imageView?.setImageBitmap(rotatedBitmap)  
 **}** *// Liberar recursos* imageProxy.close()  
 colorMat.release()  
 grayMat.release()  
 eyes.release()  
 bitmap.recycle()  
 colorBitmap.recycle()  
 out.close()  
  
 } catch (e: Exception) {  
 e.printStackTrace()  
 imageProxy.close()  
 }  
 **}** onDispose **{** frameProcessor = null  
 **}  
 }** Box(modifier = Modifier.*fillMaxSize*()) **{** *// Vista previa de la cámara (oculta)* AndroidView(  
 factory = **{** ctx **->** PreviewView(ctx).*apply* **{** *implementationMode* = PreviewView.ImplementationMode.*PERFORMANCE* previewView = this  
 *alpha* = 0f *// Ocultar la vista previa original* **}  
 }**,  
 modifier = Modifier.*fillMaxSize*()  
 )  
  
 *// Vista para mostrar la imagen procesada* AndroidView(  
 factory = **{** ctx **->** android.widget.ImageView(ctx).*apply* **{** imageView = this  
 *scaleType* = android.widget.ImageView.ScaleType.*FIT\_CENTER* **}  
 }**,  
 modifier = Modifier.*fillMaxSize*()  
 )  
 **}** *// LaunchedEffect observa los cambios en previewView y ejecuta la configuración de la cámara* LaunchedEffect(previewView) **{** *// Solo procede si previewView está inicializado* previewView?.*let* **{** preview **->** try {  
 *// Obtiene una instancia del proveedor de la cámara* val cameraProvider = cameraProviderFuture.get()  
  
 *// Configura el caso de uso de previsualización* val previewUseCase = Preview.Builder()  
 .build()  
 .*also* **{** *// Conecta el proveedor de superficie para mostrar la imagen* **it**.setSurfaceProvider(preview.*surfaceProvider*)  
 **}** *// Configura el análisis de imágenes* val imageAnalysis = ImageAnalysis.Builder()  
 *// Establece la estrategia para manejar nuevos frames  
 // STRATEGY\_KEEP\_ONLY\_LATEST mantiene solo el frame más reciente* .setBackpressureStrategy(ImageAnalysis.*STRATEGY\_KEEP\_ONLY\_LATEST*)  
 .build()  
 .*apply* **{** *// Asigna el procesador de frames en el hilo principal* setAnalyzer(  
 ContextCompat.getMainExecutor(context),  
 frameProcessor!!  
 )  
 **}** *// Configura el selector de cámara para usar la cámara frontal* val cameraSelector = CameraSelector.Builder()  
 .requireLensFacing(CameraSelector.*LENS\_FACING\_FRONT*)  
 .build()  
  
 *// Desvincula casos de uso previos* cameraProvider.unbindAll()  
  
 *// Vincula todos los casos de uso al ciclo de vida* cameraProvider.bindToLifecycle(  
 lifecycleOwner,  
 cameraSelector,  
 previewUseCase,  
 imageAnalysis  
 )  
 } catch (e: Exception) {  
 *// Maneja cualquier error durante la configuración* e.printStackTrace()  
 }  
 **}  
 }**}  
  
@Composable  
fun RationaleDialog(  
 onConfirm: () -> Unit  
) {  
 AlertDialog(  
 onDismissRequest = **{ }**,  
 title = **{** Text("Permiso de Cámara Requerido") **}**,  
 text = **{** Text("Esta aplicación necesita acceso a la cámara para detectar ojos.")  
 **}**,  
 confirmButton = **{** TextButton(onClick = onConfirm) **{** Text("Conceder Permiso")  
 **}  
 }** )  
}

**MainActivity.kt**:

package com.example.capitulo2\_practica1  
  
import android.os.Bundle  
import androidx.activity.ComponentActivity  
import androidx.activity.compose.setContent  
import androidx.activity.enableEdgeToEdge  
import com.example.capitulo2\_practica1.ui.theme.Capitulo2\_practica1Theme  
  
import org.opencv.android.OpenCVLoader  
  
class MainActivity : ComponentActivity() {  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 *enableEdgeToEdge*()  
  
 *// Inicializar OpenCV* OpenCVLoader.initLocal()  
  
 *setContent* **{** Capitulo2\_practica1Theme **{** MainScreen()  
 **}  
 }** }  
}