**Capítulo 4 – Yolo**

En este entorno virtual instalar:

pip install torch==2.5.1 torchvision==0.20.1 torchaudio==2.5.1 --index-url https://download.pytorch.org/whl/cu121

**convertir.py**

import torch

from ultralytics import YOLO

import shutil

# 1. Cargar el modelo pre-entrenado

model = YOLO("yolo11n.pt")

# 2. Exportar directamente al formato TorchScript

# La exportación predeterminada ya incluye algunas optimizaciones

model.export(format="torchscript", imgsz=640)

# 3. Dado que estamos teniendo problemas con el optimizador para móviles,

# utilizaremos directamente el modelo exportado que generalmente es

# compatible con PyTorch Mobile

shutil.copy("yolo11n.torchscript", "yolo11n\_mobile.pt")

print("Modelo exportado y listo para aplicaciones móviles")

# 4. Probar el modelo exportado

mobile\_model = torch.jit.load("yolo11n\_mobile.pt")

dummy\_input = torch.zeros(1, 3, 640, 640)

# Ejecutar prueba de inferencia

with torch.no\_grad():

try:

output = mobile\_model(dummy\_input)

print("El modelo está funcionando correctamente")

print(f"Forma de la salida: {output.shape}")

except Exception as e:

print(f"Error durante la inferencia: {e}")

# Verificar tamaño del modelo

import os

model\_size\_mb = os.path.getsize("yolo11n\_mobile.pt") / (1024 \* 1024)

print(f"Tamaño del modelo: {model\_size\_mb:.2f} MB")

# Probar con una entrada más realista (formato de imagen RGB)

test\_input = torch.rand(1, 3, 640, 640) # Valores aleatorios entre 0-1

with torch.no\_grad():

output = mobile\_model(test\_input)

print("El modelo procesó correctamente una imagen de prueba")

print(f"Forma de la salida de detección: {output.shape}")

*// CameraX  
implementation* ("androidx.camera:camera-camera2:1.4.2")  
*implementation* ("androidx.camera:camera-lifecycle:1.4.2")  
*implementation* ("androidx.camera:camera-view:1.4.2")

*// Permissions  
implementation* ("com.google.accompanist:accompanist-permissions:0.32.0")  
  
*// PyTorch Mobile  
implementation* ("org.pytorch:pytorch\_android:2.1.0")  
*implementation* ("org.pytorch:pytorch\_android\_torchvision:2.1.0")  
  
*// Coroutines  
implementation* ("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")

**build.gradle.kts**(Module: App):

plugins **{** alias(*libs*.*plugins*.*android*.*application*)  
 alias(*libs*.*plugins*.*kotlin*.*android*)  
 alias(*libs*.*plugins*.*kotlin*.*compose*)  
**}***android* **{** namespace = "com.example.capitulo4\_practica1"  
 compileSdk = 35  
  
 defaultConfig **{** applicationId = "com.example.capitulo4\_practica1"  
 minSdk = 34  
 targetSdk = 35  
 versionCode = 1  
 versionName = "1.0"  
  
 testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"  
 **}** buildTypes **{** *release* **{** isMinifyEnabled = false  
 proguardFiles(  
 getDefaultProguardFile("proguard-android-optimize.txt"),  
 "proguard-rules.pro"  
 )  
 **}  
 }** compileOptions **{** sourceCompatibility = JavaVersion.*VERSION\_11* targetCompatibility = JavaVersion.*VERSION\_11* **}** *kotlinOptions* **{** jvmTarget = "11"  
 **}** buildFeatures **{** compose = true  
 **}  
}***dependencies* **{** *implementation*(*libs*.*androidx*.*core*.*ktx*)  
 *implementation*(*libs*.*androidx*.*lifecycle*.*runtime*.*ktx*)  
 *implementation*(*libs*.*androidx*.*activity*.*compose*)  
 *implementation*(platform(*libs*.*androidx*.*compose*.*bom*))  
 *implementation*(*libs*.*androidx*.*ui*)  
 *implementation*(*libs*.*androidx*.*ui*.*graphics*)  
 *implementation*(*libs*.*androidx*.*ui*.*tooling*.*preview*)  
 *implementation*(*libs*.*androidx*.*material3*)  
 *testImplementation*(*libs*.*junit*)  
 *androidTestImplementation*(*libs*.*androidx*.*junit*)  
 *androidTestImplementation*(*libs*.*androidx*.*espresso*.*core*)  
 *androidTestImplementation*(platform(*libs*.*androidx*.*compose*.*bom*))  
 *androidTestImplementation*(*libs*.*androidx*.*ui*.*test*.*junit4*)  
 *debugImplementation*(*libs*.*androidx*.*ui*.*tooling*)  
 *debugImplementation*(*libs*.*androidx*.*ui*.*test*.*manifest*)  
  
 *// CameraX  
 implementation* ("androidx.camera:camera-camera2:1.4.2")  
 *implementation* ("androidx.camera:camera-lifecycle:1.4.2")  
 *implementation* ("androidx.camera:camera-view:1.4.2")  
  
 *// Permissions  
 implementation* ("com.google.accompanist:accompanist-permissions:0.32.0")  
  
 *// PyTorch Mobile  
 implementation* ("org.pytorch:pytorch\_android:2.1.0")  
 *implementation* ("org.pytorch:pytorch\_android\_torchvision:2.1.0")  
  
 *// Coroutines  
 implementation* ("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")  
**}**

Permisos en **AndroidManifest.xml**:

*<!-- Permisos necesarios -->*<uses-permission android:name="android.permission.CAMERA" />  
<uses-feature android:name="android.hardware.camera" />  
<uses-feature android:name="android.hardware.camera.autofocus" />

**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">  
  
 *<!-- Permisos necesarios -->* <uses-permission android:name="android.permission.CAMERA" />  
 <uses-feature android:name="android.hardware.camera" />  
 <uses-feature android:name="android.hardware.camera.autofocus" />  
  
 <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.Capitulo4\_practica1"  
 tools:targetApi="31">  
 <activity  
 android:name=".MainActivity"  
 android:exported="true"  
 android:label="@string/app\_name"  
 android:theme="@style/Theme.Capitulo4\_practica1">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN" />  
  
 <category android:name="android.intent.category.LAUNCHER" />  
 </intent-filter>  
 </activity>  
 </application>  
  
</manifest>

**Paso 6: Codificación de la App**

**Fase 1:** Formato en la App  
**Color.kt** en ui > theme

Código **Color.kt**:

package com.example.capitulo4\_practica1.ui.theme  
  
import androidx.compose.ui.graphics.Color  
  
val *Purple80* = *Color*(0xFFD0BCFF)  
val *PurpleGrey80* = *Color*(0xFFCCC2DC)  
val *Pink80* = *Color*(0xFFEFB8C8)  
  
val *Purple40* = *Color*(0xFF6650a4)  
val *PurpleGrey40* = *Color*(0xFF625b71)  
val *Pink40* = *Color*(0xFF7D5260)  
  
*// Colores para las detecciones*val *DetectionBox* = *Color*(0xFF00C853)  
val *DetectionText* = *Color*(0xFFFFFFFF)  
val *DetectionBackground* = *Color*(0x88000000)

**Type.kt** en ui > theme

Código **Type.kt**:

package com.example.capitulo4\_practica1.ui.theme  
  
import androidx.compose.material3.Typography  
import androidx.compose.ui.text.TextStyle  
import androidx.compose.ui.text.font.FontFamily  
import androidx.compose.ui.text.font.FontWeight  
import androidx.compose.ui.unit.sp  
  
val *Typography* = Typography(  
 bodyLarge = TextStyle(  
 fontFamily = FontFamily.Default,  
 fontWeight = FontWeight.Normal,  
 fontSize = 16.*sp*,  
 lineHeight = 24.*sp*,  
 letterSpacing = 0.5.*sp* ),  
 titleLarge = TextStyle(  
 fontFamily = FontFamily.Default,  
 fontWeight = FontWeight.Bold,  
 fontSize = 22.*sp*,  
 lineHeight = 28.*sp*,  
 letterSpacing = 0.*sp* ),  
 labelSmall = TextStyle(  
 fontFamily = FontFamily.Default,  
 fontWeight = FontWeight.Medium,  
 fontSize = 11.*sp*,  
 lineHeight = 16.*sp*,  
 letterSpacing = 0.5.*sp* )  
)

Código **Theme.kt**:

package com.example.capitulo4\_practica1.ui.theme  
import android.app.Activity  
import android.os.Build  
import androidx.compose.foundation.isSystemInDarkTheme  
import androidx.compose.material3.MaterialTheme  
import androidx.compose.material3.darkColorScheme  
import androidx.compose.material3.dynamicDarkColorScheme  
import androidx.compose.material3.dynamicLightColorScheme  
import androidx.compose.material3.lightColorScheme  
import androidx.compose.runtime.Composable  
import androidx.compose.runtime.SideEffect  
import androidx.compose.ui.graphics.toArgb  
import androidx.compose.ui.platform.*LocalContext*import androidx.compose.ui.platform.*LocalView*import androidx.core.view.WindowCompat  
  
private val *DarkColorScheme* = *darkColorScheme*(  
 primary = *Purple80*,  
 secondary = *PurpleGrey80*,  
 tertiary = *Pink80*)  
  
private val *LightColorScheme* = *lightColorScheme*(  
 primary = *Purple40*,  
 secondary = *PurpleGrey40*,  
 tertiary = *Pink40*)  
  
@Composable  
fun Capitulo4\_practica1Theme(  
 darkTheme: Boolean = isSystemInDarkTheme(),  
 dynamicColor: Boolean = true,  
 content: @Composable () -> Unit  
) {  
 val colorScheme = when {  
 dynamicColor && Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*S* -> {  
 val context = *LocalContext*.current  
 if (darkTheme) *dynamicDarkColorScheme*(context) else *dynamicLightColorScheme*(context)  
 }  
 darkTheme -> *DarkColorScheme* else -> *LightColorScheme* }  
 val view = *LocalView*.current  
 if (!view.*isInEditMode*) {  
 SideEffect **{** val window = (view.*context* as Activity).*window* val insetsController = WindowCompat.getInsetsController(window, view)  
  
 *// Configurar la apariencia de la barra de estado según el tema* insetsController.*isAppearanceLightStatusBars* = !darkTheme  
 **}** }  
  
 MaterialTheme(  
 colorScheme = colorScheme,  
 typography = *Typography*,  
 content = content  
 )  
}

Código **ImageUtils.kt**:

package com.example.capitulo4\_practica1.util  
  
import android.content.Context  
import android.graphics.Bitmap  
import android.graphics.BitmapFactory  
import android.graphics.ImageFormat  
import android.graphics.Matrix  
import android.graphics.Rect  
import android.graphics.RectF  
import android.graphics.YuvImage  
import android.media.Image  
import android.util.Log  
import androidx.camera.core.ExperimentalGetImage  
import androidx.camera.core.ImageProxy  
import java.io.ByteArrayOutputStream  
import java.nio.FloatBuffer  
  
object ImageUtils {  
 private const val TAG = "ImageUtils"  
  
 *// Método mejorado para convertir ImageProxy a Bitmap a color* @ExperimentalGetImage  
 fun imageProxyToBitmap(imageProxy: ImageProxy): Bitmap {  
 val image = imageProxy.*image* ?: throw IllegalArgumentException("Image proxy no contiene imagen")  
  
 *// Convertir YUV\_420\_888 a NV21* val nv21 = yuv420888ToNv21(image)  
  
 *// Convertir NV21 a Bitmap* val yuvImage = YuvImage(nv21, ImageFormat.*NV21*, image.*width*, image.*height*, null)  
 val out = ByteArrayOutputStream()  
  
 *// Comprimir a JPEG con alta calidad para mantener colores* yuvImage.compressToJpeg(Rect(0, 0, image.*width*, image.*height*), 100, out)  
  
 val imageBytes = out.toByteArray()  
 val bitmap = BitmapFactory.decodeByteArray(imageBytes, 0, imageBytes.size)  
  
 *// Verificar que la conversión produjo un bitmap válido con colores* if (bitmap != null) {  
 Log.d(TAG, "Conversión exitosa a bitmap ${bitmap.*width*}x${bitmap.*height*}, config: ${bitmap.*config*}")  
  
 *// Verificar que el bitmap tiene una configuración válida para colores* if (bitmap.*config* != Bitmap.Config.*ARGB\_8888*) {  
 Log.d(TAG, "Convirtiendo bitmap de ${bitmap.*config*} a ARGB\_8888")  
 return bitmap.copy(Bitmap.Config.*ARGB\_8888*, true)  
 }  
 } else {  
 Log.e(TAG, "Conversión de bitmap falló")  
 throw IllegalStateException("No se pudo crear el bitmap a color")  
 }  
  
 return bitmap  
 }  
  
 *// Convertir imagen YUV\_420\_888 a formato NV21 (YUV420SP)  
 // Este es un paso crucial para mantener el color* private fun yuv420888ToNv21(image: Image): ByteArray {  
 val width = image.*width* val height = image.*height* val ySize = width \* height  
 val uvSize = width \* height / 2  
  
 val nv21 = ByteArray(ySize + uvSize)  
  
 *// Get the YUV planes* val yBuffer = image.*planes*[0].*buffer* val uBuffer = image.*planes*[1].*buffer* val vBuffer = image.*planes*[2].*buffer* val yRowStride = image.*planes*[0].*rowStride* val uvRowStride = image.*planes*[1].*rowStride* val uvPixelStride = image.*planes*[1].*pixelStride* var pos = 0  
  
 *// Copy Y channel* if (yRowStride == width) { *// The stride matches the width* yBuffer.get(nv21, 0, ySize)  
 pos = ySize  
 } else { *// Copy each row separately* for (row in 0 *until* height) {  
 yBuffer.position(row \* yRowStride)  
 yBuffer.get(nv21, row \* width, width)  
 }  
 pos = ySize  
 }  
  
 *// Copy VU data* val vPos = vBuffer.position()  
 val uPos = uBuffer.position()  
 vBuffer.position(vPos)  
 uBuffer.position(uPos)  
  
 *// Interleave the V and U channels into the last half of the array  
 // V first, then U* for (row in 0 *until* height / 2) {  
 for (col in 0 *until* width / 2) {  
 nv21[pos++] = vBuffer.get(row \* uvRowStride + col \* uvPixelStride)  
 nv21[pos++] = uBuffer.get(row \* uvRowStride + col \* uvPixelStride)  
 }  
 }  
  
 return nv21  
 }  
  
 *// Rotar bitmap según la rotación del dispositivo* fun rotateBitmap(bitmap: Bitmap, rotationDegrees: Int): Bitmap {  
 if (rotationDegrees == 0) return bitmap  
  
 val matrix = Matrix()  
 matrix.postRotate(rotationDegrees.toFloat())  
  
 val rotatedBitmap = Bitmap.createBitmap(  
 bitmap, 0, 0, bitmap.*width*, bitmap.*height*, matrix, true  
 )  
  
 *// Liberar el bitmap original si hemos creado uno nuevo* if (rotatedBitmap != bitmap) {  
 bitmap.recycle()  
 }  
  
 return rotatedBitmap  
 }  
  
 *// Preparar imagen para la entrada del modelo YOLOv11* fun prepareImageForModel(bitmap: Bitmap, modelInputSize: Int): Bitmap {  
 Log.d(TAG, "Preparando imagen: ${bitmap.*width*}x${bitmap.*height*} para modelo: ${modelInputSize}x${modelInputSize}")  
  
 *// Verificar que el bitmap tiene formato de color adecuado* val processedBitmap = if (bitmap.*config* != Bitmap.Config.*ARGB\_8888*) {  
 Log.d(TAG, "Convirtiendo bitmap de ${bitmap.*config*} a ARGB\_8888")  
 bitmap.copy(Bitmap.Config.*ARGB\_8888*, true)  
 } else {  
 bitmap  
 }  
  
 return Bitmap.createScaledBitmap(processedBitmap, modelInputSize, modelInputSize, true)  
 }  
  
 *// Convertir bitmap a FloatBuffer para entrada del modelo* fun bitmapToFloatBuffer(bitmap: Bitmap): FloatBuffer {  
 val inputWidth = bitmap.*width* val inputHeight = bitmap.*height  
  
 // Crear un FloatBuffer del tamaño adecuado (3 canales: RGB)* val floatBuffer = FloatBuffer.allocate(inputWidth \* inputHeight \* 3)  
 floatBuffer.rewind()  
  
 *// Extraer valores de píxeles* val intValues = IntArray(inputWidth \* inputHeight)  
 bitmap.getPixels(intValues, 0, inputWidth, 0, 0, inputWidth, inputHeight)  
  
 *// Normalización para YOLOv11 (valores entre 0-1)* for (i in intValues.*indices*) {  
 val pixel = intValues[i]  
  
 *// Normalizar cada canal (RGB) dividiendo por 255* floatBuffer.put(((pixel shr 16) and 0xFF) / 255.0f) *// R* floatBuffer.put(((pixel shr 8) and 0xFF) / 255.0f) *// G* floatBuffer.put((pixel and 0xFF) / 255.0f) *// B* }  
  
 floatBuffer.rewind()  
 return floatBuffer  
 }  
  
 *// Mapear coordenadas de cajas delimitadoras al tamaño original* fun mapBoxToOriginalImage(  
 box: RectF,  
 originalWidth: Int,  
 originalHeight: Int,  
 modelInputSize: Int  
 ): RectF {  
 val scaleX = originalWidth.toFloat() / modelInputSize  
 val scaleY = originalHeight.toFloat() / modelInputSize  
  
 return RectF(  
 box.left \* scaleX,  
 box.top \* scaleY,  
 box.right \* scaleX,  
 box.bottom \* scaleY  
 )  
 }  
  
 *// Cargar etiquetas de clases COCO para YOLOv11* fun loadClassLabels(context: Context): List<String> {  
 try {  
 Log.d(TAG, "Cargando etiquetas de clase desde coco.names")  
 val inputStream = context.*assets*.open("coco.names")  
 val text = inputStream.*bufferedReader*().*use* **{ it**.*readText*() **}** val labels = text.*split*("\n").*filter* **{ it**.*isNotBlank*() **}** Log.d(TAG, "Cargadas ${labels.size} etiquetas de clase")  
 return labels  
 } catch (e: Exception) {  
 Log.e(TAG, "Error al cargar etiquetas de clase", e)  
 return *emptyList*()  
 }  
 }  
}

**PermissionUtils.kt** en **util**:

@OptIn(ExperimentalPermissionsApi::class)

@Composable

fun CameraPermissionScreen(

permissionState: PermissionState,

onPermissionGranted: @Composable () -> Unit

)

**Lógica Condicional Principal**

if (permissionState.status.isGranted) {

onPermissionGranted()

} else {

// UI para solicitar permiso

}

**Interfaz de Usuario para Solicitud de Permiso**

Column(

modifier = Modifier

.fillMaxSize()

.padding(16.dp),

horizontalAlignment = Alignment.CenterHorizontally,

verticalArrangement = Arrangement.Center

) {

// Contenido de la columna

}

**Texto Explicativo Adaptativo**

val textToShow = if (permissionState.status.shouldShowRationale) {

// Si el sistema muestra el diálogo de justificación, primero explicamos por qué necesitamos el permiso

stringResource(R.string.camera\_permission\_required)

} else {

// De lo contrario, se trata de la primera vez que se solicita o después de que se ha seleccionado "No volver a preguntar"

stringResource(R.string.camera\_permission\_required)

}

**Componentes de UI para la Solicitud**

Text(

text = textToShow,

textAlign = TextAlign.Center

)

Spacer(modifier = Modifier.height(16.dp))

Button(

onClick = { permissionState.launchPermissionRequest() }

) {

Text(stringResource(R.string.grant\_permission))

}

Código **PermissionUtils.kt**:

package com.example.capitulo4\_practica1.util  
  
import androidx.compose.foundation.layout.Arrangement  
import androidx.compose.foundation.layout.Column  
import androidx.compose.foundation.layout.Spacer  
import androidx.compose.foundation.layout.fillMaxSize  
import androidx.compose.foundation.layout.height  
import androidx.compose.foundation.layout.padding  
import androidx.compose.material3.Button  
import androidx.compose.material3.Text  
import androidx.compose.runtime.Composable  
import androidx.compose.ui.Alignment  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.res.stringResource  
import androidx.compose.ui.text.style.TextAlign  
import androidx.compose.ui.unit.dp  
import com.example.capitulo4\_practica1.R  
import com.google.accompanist.permissions.ExperimentalPermissionsApi  
import com.google.accompanist.permissions.PermissionState  
import com.google.accompanist.permissions.*isGranted*import com.google.accompanist.permissions.*shouldShowRationale*@OptIn(ExperimentalPermissionsApi::class)  
@Composable  
fun CameraPermissionScreen(  
 permissionState: PermissionState,  
 onPermissionGranted: @Composable () -> Unit  
) {  
 if (permissionState.status.*isGranted*) {  
 onPermissionGranted()  
 } else {  
 Column(  
 modifier = Modifier  
 .*fillMaxSize*()  
 .*padding*(16.*dp*),  
 horizontalAlignment = Alignment.CenterHorizontally,  
 verticalArrangement = Arrangement.Center  
 ) **{** val textToShow = if (permissionState.status.*shouldShowRationale*) {  
 *// Si el sistema muestra el diálogo de justificación, primero explicamos por qué necesitamos el permiso* stringResource(R.string.*camera\_permission\_required*)  
 } else {  
 *// De lo contrario, se trata de la primera vez que se solicita o después de que se ha seleccionado "No volver a preguntar"* stringResource(R.string.*camera\_permission\_required*)  
 }  
  
 Text(  
 text = textToShow,  
 textAlign = TextAlign.Center  
 )  
  
 Spacer(modifier = Modifier.*height*(16.*dp*))  
  
 Button(  
 onClick = **{** permissionState.launchPermissionRequest() **}** ) **{** Text(stringResource(R.string.*grant\_permission*))  
 **}  
 }** }  
}

**Configuración Visual**

private val boxPaint = Paint().apply {

color = Color.GREEN

style = Paint.Style.STROKE

strokeWidth = 8f *// Línea más gruesa para mayor visibilidad*

}

private val textBackgroundPaint = Paint().apply {

color = Color.BLACK

style = Paint.Style.FILL

alpha = 200 *// Más opaco para mejor legibilidad*

}

private val textPaint = Paint().apply {

color = Color.WHITE

textSize = 48f *// Texto más grande*

isFakeBoldText = true

}

**Método Principal: drawDetectionResults**

fun drawDetectionResults(

bitmap: Bitmap,

detectionResults: ObjectDetector.DetectionResult,

objectDetector: ObjectDetector

): Bitmap

**Procesamiento Inteligente de Detecciones**

for (i in boxes.indices) {

try {

if (scores[i] <= 0.5f) {

continue

}

highConfidenceCount++

*// Obtener información del objeto*

val label = objectDetector.getClassLabel(classIds[i])

val isPhone = label.contains("phone") || label.contains("cell")

*// Procesar el objeto según su tipo*

if (isPhone) {

*// Lógica para teléfonos*

} else {

*// Lógica para otros objetos*

}

} catch (e: Exception) {

Log.e(TAG, "Error procesando detección #$i", e)

}

}

**Posicionamiento Adaptativo**

*// Para otros objetos: usar áreas predefinidas según la posición vertical*

val centerY = (box.top + box.bottom) / 2

*// Dividir la pantalla en tres zonas verticales según centerY*

val safeBox: RectF

when {

centerY < 0.5f -> {

*// Objetos en la parte superior (como tazas)*

safeBox = RectF(

bitmapWidth \* 0.1f,

bitmapHeight \* 0.1f,

bitmapWidth \* 0.9f,

bitmapHeight \* 0.3f

)

}

*// ... otras zonas*

}

**Código Defensivo**

try {

*// Validar que el recuadro tiene dimensiones válidas*

if (box.width() <= 0 || box.height() <= 0) {

Log.w(TAG, "Recuadro con dimensiones inválidas: $box")

return

}

*// ... resto del código*

} catch (e: Exception) {

Log.e(TAG, "Error dibujando recuadro: ${e.message}")

}

**Detalles Visuales Mejorados**

*// Dibujar esquinas para mayor visibilidad*

val cornerLength = minOf(box.width(), box.height()) \* 0.2f

*// Esquinas*

canvas.drawLine(box.left, box.top, box.left + cornerLength, box.top, boxPaint)

*// ... más líneas para las esquinas*

**Función de Depuración**

kotlin

fun addDebugInfo(

bitmap: Bitmap,

debugInfo: String

): Bitmap

Código **PrePostProcessor.kt**:

package com.example.capitulo4\_practica1.ml  
  
import android.graphics.Bitmap  
import android.graphics.Canvas  
import android.graphics.Color  
import android.graphics.Paint  
import android.graphics.RectF  
import android.util.Log  
import java.text.DecimalFormat  
  
class PrePostProcessor {  
 companion object {  
 private const val TAG = "PrePostProcessor"  
 private val DECIMAL\_FORMAT = DecimalFormat("#.##")  
  
 *// Configuración mejorada para dibujar los cuadros de detección* private val boxPaint = Paint().*apply* **{** *color* = Color.*GREEN  
 style* = Paint.Style.*STROKE  
 strokeWidth* = 8f *// Línea más gruesa para mayor visibilidad* **}** private val textBackgroundPaint = Paint().*apply* **{** *color* = Color.*BLACK  
 style* = Paint.Style.*FILL  
 alpha* = 200 *// Más opaco para mejor legibilidad* **}** private val textPaint = Paint().*apply* **{** *color* = Color.*WHITE  
 textSize* = 48f *// Texto más grande  
 isFakeBoldText* = true  
 **}** *// Dibujar los resultados de detección en el bitmap* fun drawDetectionResults(  
 bitmap: Bitmap,  
 detectionResults: ObjectDetector.DetectionResult,  
 objectDetector: ObjectDetector  
 ): Bitmap {  
 try {  
 Log.d(TAG, "Dibujando ${detectionResults.boxes.size} detecciones en bitmap ${bitmap.*width*}x${bitmap.*height*}")  
  
 *// Crear un nuevo bitmap mutable para dibujar* val outputBitmap = bitmap.copy(Bitmap.Config.*ARGB\_8888*, true)  
 val canvas = Canvas(outputBitmap)  
  
 val bitmapWidth = bitmap.*width*.toFloat()  
 val bitmapHeight = bitmap.*height*.toFloat()  
  
 *// Configuración visual mejorada* boxPaint.*color* = Color.*RED* boxPaint.*strokeWidth* = 10f  
  
 val boxes = detectionResults.boxes  
 val scores = detectionResults.scores  
 val classIds = detectionResults.classIds  
  
 *// Filtrar por alta confianza* var highConfidenceCount = 0  
  
 for (i in boxes.*indices*) {  
 try {  
 if (scores[i] <= 0.5f) {  
 continue  
 }  
  
 highConfidenceCount++  
  
 *// Obtener información del objeto* val label = objectDetector.getClassLabel(classIds[i])  
 val isPhone = label.*contains*("phone") || label.*contains*("cell")  
  
 *// Procesar el objeto según su tipo* if (isPhone) {  
 *// Para teléfonos: usar posiciones fijas en pantalla que funcionan bien* val phoneBox = RectF(  
 bitmapWidth \* 0.2f,  
 bitmapHeight \* 0.4f,  
 bitmapWidth \* 0.8f,  
 bitmapHeight \* 0.6f  
 )  
  
 *// Dibujar recuadro con color especial para teléfonos* drawSafeBox(canvas, phoneBox, label, scores[i], Color.*BLUE*)  
 } else {  
 *// Para otros objetos: usar áreas predefinidas según la posición vertical  
 // en el espacio de coordenadas del modelo* val box = boxes[i]  
 val centerY = (box.top + box.bottom) / 2  
  
 *// Dividir la pantalla en tres zonas verticales según centerY* val safeBox: RectF  
  
 when {  
 centerY < 0.5f -> {  
 *// Objetos en la parte superior (como tazas)* safeBox = RectF(  
 bitmapWidth \* 0.1f,  
 bitmapHeight \* 0.1f,  
 bitmapWidth \* 0.9f,  
 bitmapHeight \* 0.3f  
 )  
 }  
 centerY > 2.0f -> {  
 *// Objetos en la parte inferior (como coches)* safeBox = RectF(  
 bitmapWidth \* 0.1f,  
 bitmapHeight \* 0.7f,  
 bitmapWidth \* 0.9f,  
 bitmapHeight \* 0.9f  
 )  
 }  
 else -> {  
 *// Objetos en la parte central* safeBox = RectF(  
 bitmapWidth \* 0.1f,  
 bitmapHeight \* 0.4f,  
 bitmapWidth \* 0.9f,  
 bitmapHeight \* 0.6f  
 )  
 }  
 }  
  
 *// Determinar color según tipo de objeto* val color = when {  
 label.*contains*("bottle") -> Color.*GREEN* label.*contains*("car") -> Color.*YELLOW* label.*contains*("person") -> Color.*MAGENTA* else -> Color.*RED* }  
  
 *// Dibujar recuadro de forma segura* drawSafeBox(canvas, safeBox, label, scores[i], color)  
 }  
 } catch (e: Exception) {  
 Log.e(TAG, "Error procesando detección #$i", e)  
 }  
 }  
  
 Log.d(TAG, "Dibujadas $highConfidenceCount detecciones con confianza >50% de un total de ${boxes.size}")  
 return outputBitmap  
  
 } catch (e: Exception) {  
 Log.e(TAG, "Error dibujando resultados", e)  
 return bitmap *// Devolver bitmap original si hay error* }  
 }  
  
 *// Método seguro para dibujar recuadros que evita errores de rango* private fun drawSafeBox(  
 canvas: Canvas,  
 box: RectF,  
 label: String,  
 confidence: Float,  
 color: Int  
 ) {  
 try {  
 *// Validar que el recuadro tiene dimensiones válidas* if (box.width() <= 0 || box.height() <= 0) {  
 Log.w(TAG, "Recuadro con dimensiones inválidas: $box")  
 return  
 }  
  
 *// Configurar estilo de dibujo* boxPaint.*color* = color  
  
 *// Dibujar sombra para mejor visibilidad* val shadowPaint = Paint(boxPaint)  
 shadowPaint.*color* = Color.*BLACK* shadowPaint.*strokeWidth* = boxPaint.*strokeWidth* + 4  
 shadowPaint.*alpha* = 150  
 canvas.drawRect(box, shadowPaint)  
  
 *// Dibujar recuadro principal* canvas.drawRect(box, boxPaint)  
  
 *// Dibujar esquinas para mayor visibilidad* val cornerLength = *minOf*(box.width(), box.height()) \* 0.2f  
  
 *// Esquinas* canvas.drawLine(box.left, box.top, box.left + cornerLength, box.top, boxPaint)  
 canvas.drawLine(box.left, box.top, box.left, box.top + cornerLength, boxPaint)  
  
 canvas.drawLine(box.right, box.top, box.right - cornerLength, box.top, boxPaint)  
 canvas.drawLine(box.right, box.top, box.right, box.top + cornerLength, boxPaint)  
  
 canvas.drawLine(box.left, box.bottom, box.left + cornerLength, box.bottom, boxPaint)  
 canvas.drawLine(box.left, box.bottom, box.left, box.bottom - cornerLength, boxPaint)  
  
 canvas.drawLine(box.right, box.bottom, box.right - cornerLength, box.bottom, boxPaint)  
 canvas.drawLine(box.right, box.bottom, box.right, box.bottom - cornerLength, boxPaint)  
  
 *// Texto con confianza* val confidenceText = String.*format*("%.1f%%", confidence \* 100)  
 val displayText = "$label $confidenceText"  
  
 *// Calcular posición segura para el texto (evitar errores de coerceIn)  
 // En lugar de usar coerceIn, calcular directamente una posición válida* val textWidth = textPaint.measureText(displayText)  
 val textHeight = 50f  
  
 *// Asegurar que el texto quede dentro del bitmap* val maxTextX = canvas.*width* - textWidth - 10  
 val textX = if (box.left < maxTextX) box.left else maxTextX  
 val textX1 = if (textX < 0) 0f else textX  
  
 val minTextY = textHeight + 10  
 val textY = if (box.top > minTextY) box.top - 10 else minTextY  
  
 *// Dibujar fondo para el texto* val textBgPaint = Paint(textBackgroundPaint)  
 canvas.drawRect(  
 textX1,  
 textY - textHeight,  
 textX1 + textWidth + 10,  
 textY,  
 textBgPaint  
 )  
  
 *// Dibujar texto* canvas.drawText(  
 displayText,  
 textX1 + 5,  
 textY - 10,  
 textPaint  
 )  
 } catch (e: Exception) {  
 Log.e(TAG, "Error dibujando recuadro: ${e.message}")  
 }  
 }  
  
 *// Función para añadir información de depuración al bitmap* fun addDebugInfo(  
 bitmap: Bitmap,  
 debugInfo: String  
 ): Bitmap {  
 try {  
 val outputBitmap = bitmap.copy(Bitmap.Config.*ARGB\_8888*, true)  
 val canvas = Canvas(outputBitmap)  
  
 val debugPaint = Paint().*apply* **{** *color* = Color.*WHITE  
 textSize* = 36f  
 *isFakeBoldText* = true  
 **}** val bgPaint = Paint().*apply* **{** *color* = Color.*BLACK  
 style* = Paint.Style.*FILL  
 alpha* = 180  
 **}** *// Obtener las líneas de depuración* val lines = debugInfo.*split*("|")  
  
 *// Área de fondo para el texto de depuración* val margin = 10f  
 val lineHeight = 40f  
 val textY = outputBitmap.*height* - margin - (lines.size \* lineHeight)  
  
 canvas.drawRect(  
 margin,  
 textY - lineHeight,  
 outputBitmap.*width* - margin,  
 outputBitmap.*height* - margin,  
 bgPaint  
 )  
  
 *// Dibujar cada línea de información* lines.*forEachIndexed* **{** index, line **->** canvas.drawText(  
 line.*trim*(),  
 margin + 10,  
 textY + (index \* lineHeight),  
 debugPaint  
 )  
 **}** return outputBitmap  
 } catch (e: Exception) {  
 Log.e(TAG, "Error añadiendo info de depuración", e)  
 return bitmap  
 }  
 }  
 }  
}

**Configuración y Parámetros**

companion object {

const val MODEL\_INPUT\_SIZE = 640

const val NUM\_CLASSES = 80 *// COCO dataset*

const val CONFIDENCE\_THRESHOLD = 0.30f *// Umbral para filtrar ruido*

const val IOU\_THRESHOLD = 0.45f

}

**Inicialización del Detector**

init {

try {

Log.d(TAG, "Inicializando detector de objetos con YOLOv11")

*// Comprobar y cargar el modelo*

val modelPath = assetFilePath(context, "models/yolo11n\_mobile.pt")

Log.d(TAG, "Ruta del modelo: $modelPath")

module = Module.load(modelPath)

Log.d(TAG, "Modelo cargado: ${module != null}")

*// Cargar etiquetas de clase*

classLabels = ImageUtils.loadClassLabels(context)

Log.d(TAG, "Etiquetas cargadas: ${classLabels.size}")

} catch (e: Exception) {

Log.e(TAG, "Error al inicializar el detector", e)

debugInfo = "Error: ${e.message}"

e.printStackTrace()

}

}

**Método Principal: detect**

fun detect(bitmap: Bitmap): DetectionResult {

if (module == null) {

Log.e(TAG, "Modelo no inicializado")

return DetectionResult(emptyList(), emptyList(), emptyList(), 0, "Error: Modelo no inicializado")

}

try {

*// Preparación de imagen para el modelo*

val resizedBitmap = ImageUtils.prepareImageForModel(bitmap, MODEL\_INPUT\_SIZE)

*// Normalización específica para YOLOv11*

val inputTensor = TensorImageUtils.bitmapToFloat32Tensor(

resizedBitmap,

floatArrayOf(0.0f, 0.0f, 0.0f), *// Sin media*

floatArrayOf(1/255.0f, 1/255.0f, 1/255.0f) *// Escalar a [0,1]*

)

*// Medir tiempo de inferencia*

val startTime = System.currentTimeMillis()

*// Ejecutar inferencia*

val outputTuple = module!!.forward(IValue.from(inputTensor))

*// Obtener tensor de salida*

val outputTensor = outputTuple.toTensor()

val outputs = outputTensor.dataAsFloatArray

*// Registrar tiempo de inferencia*

inferenceTime = System.currentTimeMillis() - startTime

*// Procesamiento según formato detectado...*

}

}

**Procesamiento Adaptativo de la Salida**

*// Analizar el formato de salida del modelo*

val detectionInfo = if (outputShape != null && outputShape.size > 1) {

*// Formato común para YOLOv11: [1, 84, 8400] o [1, 8400, 84]*

if (outputShape[1] > outputShape[0] && outputShape.size > 2) {

*// Formato [1, 84, 8400] - necesita transposición*

processDetectionsWithShapeAnalysis(outputs, outputShape, bitmap.width, bitmap.height)

} else {

*// Formato [1, 8400, 84] - formato estándar*

processDetections(outputs, bitmap.width, bitmap.height)

}

} else {

*// Fallback a procesamiento genérico*

processDetections(outputs, bitmap.width, bitmap.height)

}

**Detección de Formato y Procesamiento de Resultados**

enum class OutputFormat {

YOLOV5\_STYLE, *// [x, y, w, h, objectness, class probabilities]*

YOLOV8\_STYLE, *// [x, y, w, h, class probabilities]*

UNKNOWN

}

private fun detectOutputFormat(outputs: FloatArray): OutputFormat {

*// Intentamos encontrar el mejor formato basado en el tamaño*

return when {

outputs.size % (NUM\_CLASSES + 5) == 0 -> OutputFormat.YOLOV5\_STYLE

outputs.size % (NUM\_CLASSES + 4) == 0 -> OutputFormat.YOLOV8\_STYLE

else -> OutputFormat.UNKNOWN

}

}

**Procesamiento de Resultados**

for (i in 0 until numDetections) {

val baseOffset = i \* stride

*// Buscar la clase con mayor probabilidad*

var maxClassScore = 0f

var detectedClass = -1

for (c in 0 until NUM\_CLASSES) {

val score = outputs[baseOffset + 4 + c]

if (score > maxClassScore) {

maxClassScore = score

detectedClass = c

}

}

*// Filtrar por umbral de confianza*

if (maxClassScore < CONFIDENCE\_THRESHOLD) continue

*// Extraer coordenadas normalizadas (0-1)*

val x = outputs[baseOffset]

val y = outputs[baseOffset + 1]

val w = outputs[baseOffset + 2]

val h = outputs[baseOffset + 3]

*// Convertir a coordenadas absolutas en el espacio del modelo*

val left = (x - w / 2) \* MODEL\_INPUT\_SIZE

val top = (y - h / 2) \* MODEL\_INPUT\_SIZE

val right = (x + w / 2) \* MODEL\_INPUT\_SIZE

val bottom = (y + h / 2) \* MODEL\_INPUT\_SIZE

}

**Non-Maximum Suppression (NMS)**

*// Aplicar Non-Maximum Suppression para eliminar detecciones duplicadas*

val selectedIndices = nms(boxes, scores, IOU\_THRESHOLD)

private fun nms(boxes: List<RectF>, scores: List<Float>, threshold: Float): List<Int> {

val indices = scores.indices.sortedByDescending { scores[it] }.toMutableList()

val selected = mutableListOf<Int>()

while (indices.isNotEmpty()) {

val idx = indices.removeAt(0)

selected.add(idx)

val box1 = boxes[idx]

indices.removeAll { j ->

val box2 = boxes[j]

calculateIoU(box1, box2) > threshold

}

}

return selected

}

Código **ObjectDetector.kt**:

package com.example.capitulo4\_practica1.ml  
  
import android.content.Context  
import android.graphics.Bitmap  
import android.graphics.RectF  
import android.util.Log  
import com.example.capitulo4\_practica1.util.ImageUtils  
import org.pytorch.IValue  
import org.pytorch.Module  
import org.pytorch.Tensor  
import org.pytorch.torchvision.TensorImageUtils  
import java.io.File  
import java.io.FileOutputStream  
import java.io.IOException  
import kotlin.math.exp  
import kotlin.math.max  
import kotlin.math.min  
  
class ObjectDetector(private val context: Context) {  
 private val TAG = "ObjectDetector"  
  
 *// Parámetros del modelo YOLOv11* companion object {  
 const val MODEL\_INPUT\_SIZE = 640  
 const val NUM\_CLASSES = 80 *// COCO dataset* const val CONFIDENCE\_THRESHOLD = 0.30f *// Umbral para filtrar ruido* const val IOU\_THRESHOLD = 0.45f  
 }  
  
 private var module: Module? = null  
 private var classLabels: List<String> = *emptyList*()  
 private var inferenceTime: Long = 0  
 private var debugInfo: String = ""  
  
 *// Clase para almacenar resultados de detección* data class DetectionResult(  
 val boxes: List<RectF>,  
 val scores: List<Float>,  
 val classIds: List<Int>,  
 val inferenceTime: Long,  
 val debugInfo: String  
 )  
  
 init {  
 try {  
 Log.d(TAG, "Inicializando detector de objetos con YOLOv11")  
  
 *// Comprobar y cargar el modelo* val modelPath = assetFilePath(context, "models/yolo11n\_mobile.pt")  
 Log.d(TAG, "Ruta del modelo: $modelPath")  
  
 module = Module.load(modelPath)  
 Log.d(TAG, "Modelo cargado: ${module != null}")  
  
 *// Cargar etiquetas de clase* classLabels = ImageUtils.loadClassLabels(context)  
 Log.d(TAG, "Etiquetas cargadas: ${classLabels.size}")  
  
 debugInfo = "Modelo cargado con éxito. Etiquetas: ${classLabels.size}"  
 } catch (e: Exception) {  
 Log.e(TAG, "Error al inicializar el detector", e)  
 debugInfo = "Error: ${e.message}"  
 e.printStackTrace()  
 }  
 }  
  
 fun detect(bitmap: Bitmap): DetectionResult {  
 if (module == null) {  
 Log.e(TAG, "Modelo no inicializado")  
 return DetectionResult(*emptyList*(), *emptyList*(), *emptyList*(), 0, "Error: Modelo no inicializado")  
 }  
  
 try {  
 *// Preparación de imagen para el modelo* val resizedBitmap = ImageUtils.prepareImageForModel(bitmap, MODEL\_INPUT\_SIZE)  
  
 *// Normalización específica para YOLOv11* val inputTensor = TensorImageUtils.bitmapToFloat32Tensor(  
 resizedBitmap,  
 *floatArrayOf*(0.0f, 0.0f, 0.0f), *// Sin media  
 floatArrayOf*(1/255.0f, 1/255.0f, 1/255.0f) *// Escalar a [0,1]* )  
  
 *// Medir tiempo de inferencia* val startTime = System.currentTimeMillis()  
  
 *// Ejecutar inferencia* val outputTuple = module!!.forward(IValue.from(inputTensor))  
  
 *// Obtener tensor de salida* val outputTensor = outputTuple.toTensor()  
 val outputs = outputTensor.*dataAsFloatArray  
  
 // Registrar tiempo de inferencia* inferenceTime = System.currentTimeMillis() - startTime  
  
 *// Información de depuración sobre la salida* val outputShape = outputTensor.shape()  
 Log.d(TAG, "Forma del tensor de salida: ${outputShape?.*contentToString*()}")  
  
 *// Analizar el formato de salida del modelo* val detectionInfo = if (outputShape != null && outputShape.size > 1) {  
 *// Formato común para YOLOv11: [1, 84, 8400] o [1, 8400, 84]* if (outputShape[1] > outputShape[0] && outputShape.size > 2) {  
 *// Formato [1, 84, 8400] - necesita transposición* processDetectionsWithShapeAnalysis(outputs, outputShape, bitmap.*width*, bitmap.*height*)  
 } else {  
 *// Formato [1, 8400, 84] - formato estándar* processDetections(outputs, bitmap.*width*, bitmap.*height*)  
 }  
 } else {  
 *// Fallback a procesamiento genérico* processDetections(outputs, bitmap.*width*, bitmap.*height*)  
 }  
  
 val boxes = detectionInfo.first  
 val scores = detectionInfo.second  
 val classIds = detectionInfo.third  
  
 Log.d(TAG, "Detecciones encontradas: ${boxes.size}")  
  
 debugInfo = "Inferencia: $inferenceTime ms | Detecciones: ${boxes.size} | " +  
 "Tensor: ${outputShape?.*contentToString*() ?: "desconocido"} | " +  
 "Primera detectado: ${if (classIds.*isNotEmpty*()) getClassLabel(classIds[0]) else "ninguno"}"  
  
 return DetectionResult(boxes, scores, classIds, inferenceTime, debugInfo)  
  
 } catch (e: Exception) {  
 Log.e(TAG, "Error durante la detección", e)  
 debugInfo = "Error de detección: ${e.message}"  
 return DetectionResult(*emptyList*(), *emptyList*(), *emptyList*(), 0, debugInfo)  
 }  
 }  
  
 *// Procesamiento de detecciones según el formato detectado* private fun processDetections(  
 outputs: FloatArray,  
 originalWidth: Int,  
 originalHeight: Int  
 ): Triple<List<RectF>, List<Float>, List<Int>> {  
 val boxes = *mutableListOf*<RectF>()  
 val scores = *mutableListOf*<Float>()  
 val classIds = *mutableListOf*<Int>()  
  
 try {  
 *// Para YOLOv11, la estructura de salida varía según la versión específica  
 // Intentamos detectar automáticamente el formato* val format = detectOutputFormat(outputs)  
 Log.d(TAG, "Formato detectado: $format")  
  
 when (format) {  
 OutputFormat.*YOLOV8\_STYLE* -> {  
 *// Formato YOLOv8: [num\_dets, 84] donde 84 = [x, y, w, h, 80 clases]* val numDetections = outputs.size / (NUM\_CLASSES + 4)  
 val stride = NUM\_CLASSES + 4  
  
 for (i in 0 *until* numDetections) {  
 val baseOffset = i \* stride  
  
 *// Buscar la clase con mayor probabilidad* var maxClassScore = 0f  
 var detectedClass = -1  
  
 for (c in 0 *until* NUM\_CLASSES) {  
 val score = outputs[baseOffset + 4 + c]  
 if (score > maxClassScore) {  
 maxClassScore = score  
 detectedClass = c  
 }  
 }  
  
 *// Filtrar por umbral de confianza* if (maxClassScore < CONFIDENCE\_THRESHOLD) continue  
  
 *// Extraer coordenadas normalizadas (0-1)* val x = outputs[baseOffset]  
 val y = outputs[baseOffset + 1]  
 val w = outputs[baseOffset + 2]  
 val h = outputs[baseOffset + 3]  
  
 *// Convertir a coordenadas absolutas en el espacio del modelo* val left = (x - w / 2) \* MODEL\_INPUT\_SIZE  
 val top = (y - h / 2) \* MODEL\_INPUT\_SIZE  
 val right = (x + w / 2) \* MODEL\_INPUT\_SIZE  
 val bottom = (y + h / 2) \* MODEL\_INPUT\_SIZE  
  
 *// Log para depuración* Log.d(TAG, "Detección: clase=${getClassLabel(detectedClass)}, " +  
 "confianza=${maxClassScore}, coordenadas=[$left,$top,$right,$bottom]")  
  
 *// Verificación de coordenadas válidas* if (left >= 0 && top >= 0 && right <= MODEL\_INPUT\_SIZE && bottom <= MODEL\_INPUT\_SIZE &&  
 right > left && bottom > top) {  
 boxes.add(RectF(left, top, right, bottom))  
 scores.add(maxClassScore)  
 classIds.add(detectedClass)  
 } else {  
 Log.w(TAG, "Coordenadas inválidas ignoradas: [$left,$top,$right,$bottom]")  
 }  
 }  
 }  
  
 OutputFormat.*YOLOV5\_STYLE* -> {  
 *// Formato YOLOv5: [num\_dets, 85] donde 85 = [x, y, w, h, conf, 80 clases]* val numDetections = outputs.size / (NUM\_CLASSES + 5)  
 val stride = NUM\_CLASSES + 5  
  
 for (i in 0 *until* numDetections) {  
 val baseOffset = i \* stride  
  
 *// Obtener confianza de objectness* val confidence = outputs[baseOffset + 4]  
 if (confidence < CONFIDENCE\_THRESHOLD) continue  
  
 *// Buscar la clase con mayor probabilidad* var maxClassScore = 0f  
 var detectedClass = -1  
  
 for (c in 0 *until* NUM\_CLASSES) {  
 val score = outputs[baseOffset + 5 + c]  
 if (score > maxClassScore) {  
 maxClassScore = score  
 detectedClass = c  
 }  
 }  
  
 *// Calcular puntuación final* val finalScore = maxClassScore \* confidence  
  
 *// Filtrar por umbral de confianza* if (finalScore < CONFIDENCE\_THRESHOLD) continue  
  
 *// Extraer coordenadas* val x = outputs[baseOffset]  
 val y = outputs[baseOffset + 1]  
 val w = outputs[baseOffset + 2]  
 val h = outputs[baseOffset + 3]  
  
 *// Convertir a coordenadas absolutas* val left = (x - w / 2) \* MODEL\_INPUT\_SIZE  
 val top = (y - h / 2) \* MODEL\_INPUT\_SIZE  
 val right = (x + w / 2) \* MODEL\_INPUT\_SIZE  
 val bottom = (y + h / 2) \* MODEL\_INPUT\_SIZE  
  
 *// Log para depuración* Log.d(TAG, "Detección: clase=${getClassLabel(detectedClass)}, " +  
 "confianza=${finalScore}, coordenadas=[$left,$top,$right,$bottom]")  
  
 *// Verificación de coordenadas válidas* if (left >= 0 && top >= 0 && right <= MODEL\_INPUT\_SIZE && bottom <= MODEL\_INPUT\_SIZE &&  
 right > left && bottom > top) {  
 boxes.add(RectF(left, top, right, bottom))  
 scores.add(finalScore)  
 classIds.add(detectedClass)  
 } else {  
 Log.w(TAG, "Coordenadas inválidas ignoradas: [$left,$top,$right,$bottom]")  
 }  
 }  
 }  
  
 OutputFormat.*UNKNOWN* -> {  
 *// Intentamos un enfoque genérico para caso desconocido* Log.w(TAG, "Usando formato de detección genérico para salida de tamaño ${outputs.size}")  
  
 *// Probamos divisibilidad por varios tamaños comunes* val possibleStrides = *listOf*(NUM\_CLASSES + 4, NUM\_CLASSES + 5, NUM\_CLASSES + 6)  
 val stride = possibleStrides.*firstOrNull* **{** outputs.size % **it** == 0 **}** ?: (NUM\_CLASSES + 5)  
 val numDetections = outputs.size / stride  
  
 for (i in 0 *until* numDetections) {  
 val baseOffset = i \* stride  
  
 *// Suponemos que las primeras 4 posiciones son coordenadas* val x = outputs[baseOffset]  
 val y = outputs[baseOffset + 1]  
 val w = outputs[baseOffset + 2]  
 val h = outputs[baseOffset + 3]  
  
 *// Intentamos ubicar la confianza* val confidence = if (stride > 4) outputs[baseOffset + 4] else 1.0f  
  
 *// Buscamos el índice donde empiezan las clases* val classOffset = *min*(baseOffset + 5, baseOffset + stride - NUM\_CLASSES)  
  
 *// Encontramos la clase con mayor probabilidad* var maxClassScore = 0f  
 var detectedClass = -1  
  
 val classLimit = *min*(NUM\_CLASSES, stride - (classOffset - baseOffset))  
 for (c in 0 *until* classLimit) {  
 val score = outputs[classOffset + c]  
 if (score > maxClassScore) {  
 maxClassScore = score  
 detectedClass = c  
 }  
 }  
  
 *// Calculamos score final* val finalScore = maxClassScore \* confidence  
  
 *// Filtramos por umbral* if (finalScore < CONFIDENCE\_THRESHOLD) continue  
  
 *// Convertimos a coordenadas absolutas* val left = (x - w / 2) \* MODEL\_INPUT\_SIZE  
 val top = (y - h / 2) \* MODEL\_INPUT\_SIZE  
 val right = (x + w / 2) \* MODEL\_INPUT\_SIZE  
 val bottom = (y + h / 2) \* MODEL\_INPUT\_SIZE  
  
 *// Log para depuración* Log.d(TAG, "Detección genérica: clase=${getClassLabel(detectedClass)}, " +  
 "confianza=${finalScore}, coordenadas=[$left,$top,$right,$bottom]")  
  
 *// Verificación de coordenadas válidas* if (left >= 0 && top >= 0 && right <= MODEL\_INPUT\_SIZE && bottom <= MODEL\_INPUT\_SIZE &&  
 right > left && bottom > top) {  
 boxes.add(RectF(left, top, right, bottom))  
 scores.add(finalScore)  
 classIds.add(detectedClass)  
 } else {  
 Log.w(TAG, "Coordenadas inválidas ignoradas: [$left,$top,$right,$bottom]")  
 }  
 }  
 }  
 }  
  
 *// Aplicar Non-Maximum Suppression para eliminar detecciones duplicadas* val selectedIndices = nms(boxes, scores, IOU\_THRESHOLD)  
  
 Log.d(TAG, "Después de NMS: ${selectedIndices.size} detecciones")  
  
 return Triple(  
 selectedIndices.*map* **{** boxes[**it**] **}**,  
 selectedIndices.*map* **{** scores[**it**] **}**,  
 selectedIndices.*map* **{** classIds[**it**] **}** )  
  
 } catch (e: Exception) {  
 Log.e(TAG, "Error al procesar detecciones", e)  
 return Triple(*emptyList*(), *emptyList*(), *emptyList*())  
 }  
 }  
  
 *// Procesamiento para formato de salida transpuesto* private fun processDetectionsWithShapeAnalysis(  
 outputs: FloatArray,  
 shape: LongArray,  
 originalWidth: Int,  
 originalHeight: Int  
 ): Triple<List<RectF>, List<Float>, List<Int>> {  
 val boxes = *mutableListOf*<RectF>()  
 val scores = *mutableListOf*<Float>()  
 val classIds = *mutableListOf*<Int>()  
  
 try {  
 *// Asumimos formato [1, c+4, anchors] o similar* if (shape.size < 3) {  
 Log.w(TAG, "Forma de tensor inesperada: ${shape.*contentToString*()}")  
 return processDetections(outputs, originalWidth, originalHeight)  
 }  
  
 *// Número de clases y anchors* val numClasses = (shape[1] - 4).toInt()  
 val numAnchors = shape[2].toInt()  
  
 Log.d(TAG, "Procesando formato transpuesto: clases=$numClasses, anchors=$numAnchors")  
  
 *// Para cada anchor* for (i in 0 *until* numAnchors) {  
 *// CORRECCIÓN: Las coordenadas están en valores absolutos muy grandes  
 // Las normalizamos dividiéndolas por un factor grande y multiplicándolas por MODEL\_INPUT\_SIZE  
 // Esto es una estimación - puede necesitar ajustes según el modelo específico* val normalizationFactor = 100000.0f *// Ajusta este valor según sea necesario  
  
 // Extraer coordenadas (primeros 4 elementos)* val x = (outputs[i] / normalizationFactor)  
 val y = (outputs[numAnchors + i] / normalizationFactor)  
 val w = (outputs[2 \* numAnchors + i] / normalizationFactor)  
 val h = (outputs[3 \* numAnchors + i] / normalizationFactor)  
  
 *// Convertir a coordenadas absolutas normalizadas al tamaño del modelo* val left = (x - w / 2) \* MODEL\_INPUT\_SIZE  
 val top = (y - h / 2) \* MODEL\_INPUT\_SIZE  
 val right = (x + w / 2) \* MODEL\_INPUT\_SIZE  
 val bottom = (y + h / 2) \* MODEL\_INPUT\_SIZE  
  
 *// Encontrar la clase con mayor probabilidad* var maxClassScore = 0f  
 var detectedClass = -1  
  
 for (c in 0 *until min*(numClasses, NUM\_CLASSES)) {  
 val score = outputs[(4 + c) \* numAnchors + i]  
 if (score > maxClassScore) {  
 maxClassScore = score  
 detectedClass = c  
 }  
 }  
  
 *// Filtrar por umbral de confianza* if (maxClassScore < CONFIDENCE\_THRESHOLD) continue  
  
 *// Log para depuración* Log.d(TAG, "Detección normalizada: clase=${getClassLabel(detectedClass)}, " +  
 "confianza=${maxClassScore}, coordenadas=[$left,$top,$right,$bottom]")  
  
 *// Verificación de coordenadas válidas* if (left >= 0 && top >= 0 && right <= MODEL\_INPUT\_SIZE && bottom <= MODEL\_INPUT\_SIZE &&  
 right > left && bottom > top) {  
 boxes.add(RectF(left, top, right, bottom))  
 scores.add(maxClassScore)  
 classIds.add(detectedClass)  
 Log.d(TAG, "Detección VÁLIDA: ${getClassLabel(detectedClass)}")  
 } else {  
 Log.w(TAG, "Coordenadas inválidas ignoradas: [$left,$top,$right,$bottom]")  
 }  
 }  
  
 *// Aplicar NMS* val selectedIndices = nms(boxes, scores, IOU\_THRESHOLD)  
  
 return Triple(  
 selectedIndices.*map* **{** boxes[**it**] **}**,  
 selectedIndices.*map* **{** scores[**it**] **}**,  
 selectedIndices.*map* **{** classIds[**it**] **}** )  
  
 } catch (e: Exception) {  
 Log.e(TAG, "Error procesando detecciones con análisis de forma", e)  
 return Triple(*emptyList*(), *emptyList*(), *emptyList*())  
 }  
 }  
  
 *// Enumeración para diferentes formatos de salida YOLO* enum class OutputFormat {  
 *YOLOV5\_STYLE*, *// [x, y, w, h, objectness, class probabilities]  
 YOLOV8\_STYLE*, *// [x, y, w, h, class probabilities]  
 UNKNOWN* }  
  
 *// Método para detectar el formato de salida* private fun detectOutputFormat(outputs: FloatArray): OutputFormat {  
 *// Intentamos encontrar el mejor formato basado en el tamaño* return when {  
 outputs.size % (NUM\_CLASSES + 5) == 0 -> OutputFormat.*YOLOV5\_STYLE* outputs.size % (NUM\_CLASSES + 4) == 0 -> OutputFormat.*YOLOV8\_STYLE* else -> OutputFormat.*UNKNOWN* }  
 }  
  
 *// Implementación de Non-Maximum Suppression* private fun nms(boxes: List<RectF>, scores: List<Float>, threshold: Float): List<Int> {  
 val indices = scores.*indices*.*sortedByDescending* **{** scores[**it**] **}**.*toMutableList*()  
 val selected = *mutableListOf*<Int>()  
  
 while (indices.*isNotEmpty*()) {  
 val idx = indices.removeAt(0)  
 selected.add(idx)  
  
 val box1 = boxes[idx]  
  
 indices.*removeAll* **{** j **->** val box2 = boxes[j]  
 calculateIoU(box1, box2) > threshold  
 **}** }  
  
 return selected  
 }  
  
 *// Calcular Intersection over Union entre dos cajas* private fun calculateIoU(box1: RectF, box2: RectF): Float {  
 val areaBox1 = (box1.right - box1.left) \* (box1.bottom - box1.top)  
 val areaBox2 = (box2.right - box2.left) \* (box2.bottom - box2.top)  
  
 if (areaBox1 <= 0 || areaBox2 <= 0) return 0f  
  
 val intersectLeft = *max*(box1.left, box2.left)  
 val intersectTop = *max*(box1.top, box2.top)  
 val intersectRight = *min*(box1.right, box2.right)  
 val intersectBottom = *min*(box1.bottom, box2.bottom)  
  
 val intersectWidth = *max*(0f, intersectRight - intersectLeft)  
 val intersectHeight = *max*(0f, intersectBottom - intersectTop)  
 val intersectArea = intersectWidth \* intersectHeight  
  
 return intersectArea / (areaBox1 + areaBox2 - intersectArea)  
 }  
  
 *// Función auxiliar para copiar archivos desde assets* private fun assetFilePath(context: Context, assetName: String): String {  
 val file = File(context.*filesDir*, assetName.*split*("/").*last*())  
  
 if (file.exists() && file.length() > 0) {  
 return file.*absolutePath* }  
  
 *// Crear directorios si no existen* file.*parentFile*?.mkdirs()  
  
 try {  
 *// Copiar archivo desde assets* context.*assets*.open(assetName).*use* **{** inputStream **->** FileOutputStream(file).*use* **{** outputStream **->** val buffer = ByteArray(4 \* 1024)  
 var read: Int  
 while (inputStream.read(buffer).*also* **{** read = **it }** != -1) {  
 outputStream.write(buffer, 0, read)  
 }  
 outputStream.flush()  
 **}  
 }** return file.*absolutePath* } catch (e: IOException) {  
 Log.e(TAG, "Error copiando archivo de assets: $assetName", e)  
 throw IOException("No se pudo copiar el archivo de assets: $assetName", e)  
 }  
 }  
  
 *// Obtener etiqueta de clase por ID* fun getClassLabel(classId: Int): String {  
 return if (classId >= 0 && classId < classLabels.size) {  
 classLabels[classId]  
 } else {  
 "Desconocido"  
 }  
 }  
  
 *// Obtener tiempo de inferencia en ms* fun getInferenceTime(): Long {  
 return inferenceTime  
 }  
  
 *// Obtener información de depuración* fun getDebugInfo(): String {  
 return debugInfo  
 }  
}

**Estructura Principal**

@ExperimentalGetImage

@OptIn(ExperimentalPermissionsApi::class)

@Composable

fun CameraScreen(context: Context) {

val permissionState = rememberPermissionState(android.Manifest.permission.CAMERA)

CameraPermissionScreen(

permissionState = permissionState

) {

CameraContent(context)

}

}

**Gestión de Estado con Compose**

*// Estados para la detección*

var objectDetector by remember { mutableStateOf<ObjectDetector?>(null) }

var processingImage by remember { mutableStateOf(false) }

var detectionBitmap by remember { mutableStateOf<Bitmap?>(null) }

var showDebugInfo by remember { mutableStateOf(true) }

*// Métricas*

var fps by remember { mutableFloatStateOf(0f) }

var detectionCount by remember { mutableIntStateOf(0) }

var inferenceTime by remember { mutableLongStateOf(0L) }

var debugInfo by remember { mutableStateOf("") }

*// Para cálculo de FPS*

var frameCount by remember { mutableIntStateOf(0) }

var lastFpsTimestamp by remember { mutableLongStateOf(System.currentTimeMillis()) }

var skipFrameCount by remember { mutableIntStateOf(0) }

**Ciclo de Vida y Recursos**

*// Inicializar el detector de objetos*

LaunchedEffect(key1 = Unit) {

try {

Log.d(TAG, "Inicializando detector de objetos")

objectDetector = ObjectDetector(context)

} catch (e: Exception) {

Log.e(TAG, "Error inicializando detector", e)

}

}

*// Executor para procesamiento de imágenes - OPTIMIZADO*

val cameraExecutor = remember {

Executors.newFixedThreadPool(2) *// Usar más threads para mejores resultados*

}

*// Liberación de recursos cuando se destruye*

DisposableEffect(Unit) {

onDispose {

cameraExecutor.shutdown()

Log.d(TAG, "Liberando recursos de cámara")

}

}

**Integración con CameraX**

AndroidView(

modifier = Modifier.fillMaxSize(),

factory = { ctx ->

val previewView = PreviewView(ctx).apply {

implementationMode = PreviewView.ImplementationMode.PERFORMANCE

scaleType = PreviewView.ScaleType.FILL\_START *// Llenar pantalla*

}

val cameraProviderFuture = ProcessCameraProvider.getInstance(ctx)

cameraProviderFuture.addListener({

val cameraProvider = cameraProviderFuture.get()

*// Configuración de resolución para pantalla completa*

val resolutionSelector = ResolutionSelector.Builder()

.setAspectRatioStrategy(AspectRatioStrategy.RATIO\_16\_9\_FALLBACK\_AUTO\_STRATEGY)

.setResolutionStrategy(ResolutionStrategy.HIGHEST\_AVAILABLE\_STRATEGY)

.build()

*// Configuración de Preview*

val preview = Preview.Builder()

.setResolutionSelector(resolutionSelector)

.build().also {

it.setSurfaceProvider(previewView.surfaceProvider)

}

*// Análisis de imagen*

val imageAnalysis = ImageAnalysis.Builder()

*// ...configuración...*

.build()

.also {

it.setAnalyzer(cameraExecutor) { imageProxy ->

*// Procesamiento de imágenes*

}

}

*// Selector de cámara (trasera por defecto)*

val cameraSelector = CameraSelector.DEFAULT\_BACK\_CAMERA

*// Bind los casos de uso a la cámara*

cameraProvider.bindToLifecycle(

lifecycleOwner,

cameraSelector,

preview,

imageAnalysis

)

}, ContextCompat.getMainExecutor(ctx))

previewView

}

)

**Estrategia de Procesamiento Eficiente**

it.setAnalyzer(cameraExecutor) { imageProxy ->

*// OPTIMIZACIÓN: Espaciar el procesamiento para reducir el retardo*

skipFrameCount++

if (skipFrameCount >= 2) { *// Procesar cada 2 frames para reducir carga*

skipFrameCount = 0

if (!processingImage) {

processingImage = true

processImage(

imageProxy = imageProxy,

objectDetector = objectDetector!!,

onResult = { resultBitmap, result ->

*// Procesamiento del resultado...*

processingImage = false

}

)

} else {

imageProxy.close()

}

} else {

imageProxy.close() *// Cerrar frames saltados*

}

}

**Procesamiento de Imágenes**

private fun processImage(

imageProxy: ImageProxy,

objectDetector: ObjectDetector,

onResult: (Bitmap, ObjectDetector.DetectionResult) -> Unit

) {

try {

*// Convertir ImageProxy a Bitmap (con corrección de color)*

val bitmap = ImageUtils.imageProxyToBitmap(imageProxy)

*// Rotar la imagen según la orientación*

val rotationDegrees = imageProxy.imageInfo.rotationDegrees

val rotatedBitmap = ImageUtils.rotateBitmap(bitmap, rotationDegrees)

*// Ejecutar detección*

val result = objectDetector.detect(rotatedBitmap)

*// Dibujar resultados en el bitmap*

val resultBitmap = PrePostProcessor.drawDetectionResults(rotatedBitmap, result, objectDetector)

*// Devolver el resultado*

onResult(resultBitmap, result)

} catch (e: Exception) {

Log.e("CameraScreen", "Error procesando imagen", e)

} finally {

*// Siempre cerrar el ImageProxy*

imageProxy.close()

}

}

**Interfaz de Usuario con Compose**

*// Mostrar el bitmap con las detecciones (encima de la vista de la cámara)*

detectionBitmap?.let { bitmap ->

Image(

bitmap = bitmap.asImageBitmap(),

contentDescription = "Detección",

modifier = Modifier.fillMaxSize()

)

}

*// Panel de información superior*

Box(

modifier = Modifier

.fillMaxWidth()

.align(Alignment.TopCenter)

.background(Color.Black.copy(alpha = 0.5f))

.padding(8.dp)

) {

Column {

Text(

text = stringResource(R.string.detection\_info),

color = Color.White,

fontWeight = FontWeight.Bold,

fontSize = 16.sp

)

*// Más información...*

}

}

*// Controles en la parte inferior*

Box(

modifier = Modifier

.fillMaxWidth()

.align(Alignment.BottomCenter)

.background(Color.Black.copy(alpha = 0.5f))

.padding(8.dp)

) {

Row(

verticalAlignment = Alignment.CenterVertically,

modifier = Modifier.fillMaxWidth()

) {

Text(

text = "Información de depuración",

color = Color.White,

modifier = Modifier.weight(1f)

)

Switch(

checked = showDebugInfo,

onCheckedChange = { showDebugInfo = it }

)

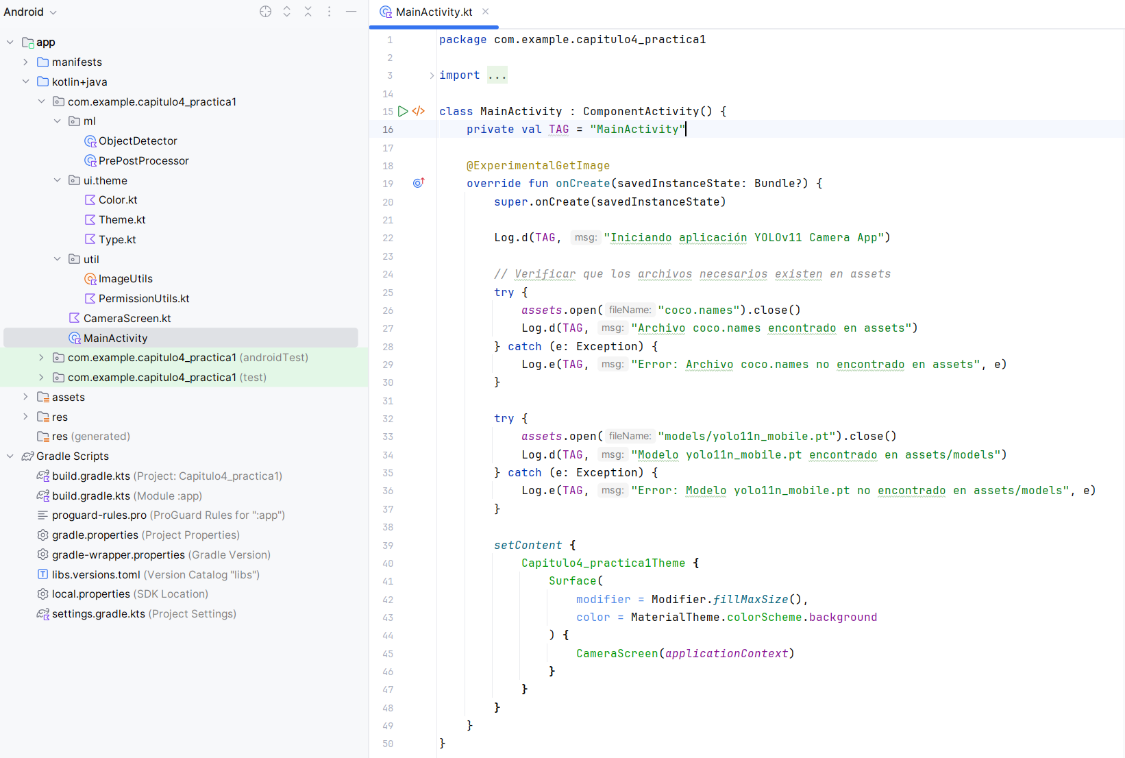
}

}

Código **CameraScreen.kt**:

package com.example.capitulo4\_practica1.ui  
  
import android.content.Context  
import android.graphics.Bitmap  
import android.util.Log  
import androidx.camera.core.CameraSelector  
import androidx.camera.core.ExperimentalGetImage  
import androidx.camera.core.ImageAnalysis  
import androidx.camera.core.ImageProxy  
import androidx.camera.core.Preview  
import androidx.camera.core.resolutionselector.AspectRatioStrategy  
import androidx.camera.core.resolutionselector.ResolutionSelector  
import androidx.camera.core.resolutionselector.ResolutionStrategy  
import androidx.camera.lifecycle.ProcessCameraProvider  
import androidx.camera.view.PreviewView  
import androidx.compose.foundation.Image  
import androidx.compose.foundation.background  
import androidx.compose.foundation.layout.Box  
import androidx.compose.foundation.layout.Column  
import androidx.compose.foundation.layout.Row  
import androidx.compose.foundation.layout.Spacer  
import androidx.compose.foundation.layout.fillMaxSize  
import androidx.compose.foundation.layout.fillMaxWidth  
import androidx.compose.foundation.layout.height  
import androidx.compose.foundation.layout.padding  
import androidx.compose.foundation.layout.width  
import androidx.compose.material3.CircularProgressIndicator  
import androidx.compose.material3.Switch  
import androidx.compose.material3.Text  
import androidx.compose.runtime.Composable  
import androidx.compose.runtime.DisposableEffect  
import androidx.compose.runtime.LaunchedEffect  
import androidx.compose.runtime.getValue  
import androidx.compose.runtime.mutableFloatStateOf  
import androidx.compose.runtime.mutableIntStateOf  
import androidx.compose.runtime.mutableLongStateOf  
import androidx.compose.runtime.mutableStateOf  
import androidx.compose.runtime.remember  
import androidx.compose.runtime.setValue  
import androidx.compose.ui.Alignment  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.graphics.Color  
import androidx.compose.ui.graphics.asImageBitmap  
import androidx.compose.ui.platform.*LocalContext*import androidx.compose.ui.res.stringResource  
import androidx.compose.ui.text.font.FontWeight  
import androidx.compose.ui.unit.dp  
import androidx.compose.ui.unit.sp  
import androidx.compose.ui.viewinterop.AndroidView  
import androidx.core.content.ContextCompat  
import com.example.capitulo4\_practica1.R  
import com.example.capitulo4\_practica1.ml.ObjectDetector  
import com.example.capitulo4\_practica1.ml.PrePostProcessor  
import com.example.capitulo4\_practica1.util.CameraPermissionScreen  
import com.example.capitulo4\_practica1.util.ImageUtils  
import com.google.accompanist.permissions.ExperimentalPermissionsApi  
import com.google.accompanist.permissions.rememberPermissionState  
import java.util.concurrent.Executors  
  
@ExperimentalGetImage  
@OptIn(ExperimentalPermissionsApi::class)  
@Composable  
fun CameraScreen(context: Context) {  
 val permissionState = rememberPermissionState(android.Manifest.permission.*CAMERA*)  
  
 CameraPermissionScreen(  
 permissionState = permissionState  
 ) **{** CameraContent(context)  
 **}**}  
  
@ExperimentalGetImage  
@Composable  
fun CameraContent(context: Context) {  
 val TAG = "CameraContent"  
  
 *// Estados para la detección* var objectDetector by remember **{** *mutableStateOf*<ObjectDetector?>(null) **}** var processingImage by remember **{** *mutableStateOf*(false) **}** var detectionBitmap by remember **{** *mutableStateOf*<Bitmap?>(null) **}** var showDebugInfo by remember **{** *mutableStateOf*(true) **}** *// Métricas* var fps by remember **{** *mutableFloatStateOf*(0f) **}** var detectionCount by remember **{** *mutableIntStateOf*(0) **}** var inferenceTime by remember **{** *mutableLongStateOf*(0L) **}** var debugInfo by remember **{** *mutableStateOf*("") **}** *// Para cálculo de FPS* var frameCount by remember **{** *mutableIntStateOf*(0) **}** var lastFpsTimestamp by remember **{** *mutableLongStateOf*(System.currentTimeMillis()) **}** var skipFrameCount by remember **{** *mutableIntStateOf*(0) **}** *// Para controlar el procesamiento  
  
 //val lifecycleOwner = LocalLifecycleOwner.current* val lifecycleOwner = androidx.lifecycle.compose.*LocalLifecycleOwner*.current  
 val lifecycleContext = *LocalContext*.current  
  
 *// Inicializar el detector de objetos* LaunchedEffect(key1 = Unit) **{** try {  
 Log.d(TAG, "Inicializando detector de objetos")  
 objectDetector = ObjectDetector(context)  
 } catch (e: Exception) {  
 Log.e(TAG, "Error inicializando detector", e)  
 }  
 **}** *// Executor para procesamiento de imágenes - OPTIMIZADO* val cameraExecutor = remember **{** Executors.newFixedThreadPool(2) *// Usar más threads para mejores resultados* **}** *// Liberación de recursos cuando se destruye* DisposableEffect(Unit) **{** onDispose **{** cameraExecutor.shutdown()  
 Log.d(TAG, "Liberando recursos de cámara")  
 **}  
 }** Box(modifier = Modifier.*fillMaxSize*()) **{** if (objectDetector == null) {  
 *// Mostrar indicador de carga mientras se inicializa el modelo* Box(  
 modifier = Modifier.*fillMaxSize*(),  
 contentAlignment = Alignment.Center  
 ) **{** Column(horizontalAlignment = Alignment.CenterHorizontally) **{** CircularProgressIndicator()  
 Spacer(modifier = Modifier.*height*(16.*dp*))  
 Text(  
 text = stringResource(R.string.*loading\_model*),  
 fontSize = 18.*sp* )  
 **}  
 }** } else {  
 *// CORRECCIÓN: Configuración de CameraX optimizada para pantalla completa* AndroidView(  
 modifier = Modifier.*fillMaxSize*(),  
 factory = **{** ctx **->** val previewView = PreviewView(ctx).*apply* **{** *implementationMode* = PreviewView.ImplementationMode.*PERFORMANCE  
 scaleType* = PreviewView.ScaleType.*FILL\_START // Llenar pantalla* **}** val cameraProviderFuture = ProcessCameraProvider.getInstance(ctx)  
 cameraProviderFuture.addListener(**{** val cameraProvider = cameraProviderFuture.get()  
  
 *// CORRECCIÓN: Optimización de la resolución para pantalla completa* val resolutionSelector = ResolutionSelector.Builder()  
 .setAspectRatioStrategy(AspectRatioStrategy.*RATIO\_16\_9\_FALLBACK\_AUTO\_STRATEGY*)  
 .setResolutionStrategy(ResolutionStrategy.*HIGHEST\_AVAILABLE\_STRATEGY*)  
 .build()  
  
 *// Configuración de Preview actualizada* val preview = Preview.Builder()  
 .setResolutionSelector(resolutionSelector)  
 .build().*also* **{  
 it**.setSurfaceProvider(previewView.*surfaceProvider*)  
 **}** *// Análisis de imagen optimizado* val imageAnalysis = ImageAnalysis.Builder()  
 .setResolutionSelector(resolutionSelector)  
 .setBackpressureStrategy(ImageAnalysis.*STRATEGY\_KEEP\_ONLY\_LATEST*)  
 .setOutputImageFormat(ImageAnalysis.*OUTPUT\_IMAGE\_FORMAT\_YUV\_420\_888*)  
 .build()  
 .*also* **{  
 it**.setAnalyzer(cameraExecutor) **{** imageProxy **->** *// OPTIMIZACIÓN: Espaciar el procesamiento para reducir el retardo* skipFrameCount++  
 if (skipFrameCount >= 2) { *// Procesar cada 2 frames para reducir carga* skipFrameCount = 0  
 if (!processingImage) {  
 processingImage = true  
  
 *processImage*(  
 imageProxy = imageProxy,  
 objectDetector = objectDetector!!,  
 onResult = **{** resultBitmap, result **->** *// Añadir información de depuración si está habilitada* val finalBitmap = if (showDebugInfo) {  
 PrePostProcessor.addDebugInfo(resultBitmap, result.debugInfo)  
 } else {  
 resultBitmap  
 }  
  
 detectionBitmap = finalBitmap  
 detectionCount = result.boxes.size  
 inferenceTime = result.inferenceTime  
 debugInfo = result.debugInfo  
  
 *// Actualizar FPS* frameCount++  
 val now = System.currentTimeMillis()  
 val elapsedMs = now - lastFpsTimestamp  
 if (elapsedMs > 1000) {  
 fps = frameCount \* 1000f / elapsedMs  
 frameCount = 0  
 lastFpsTimestamp = now  
 }  
  
 processingImage = false  
 **}** )  
 } else {  
 imageProxy.close()  
 }  
 } else {  
 imageProxy.close() *// Cerrar frames saltados* }  
 **}  
 }** *// Selector de cámara (trasera por defecto)* val cameraSelector = CameraSelector.*DEFAULT\_BACK\_CAMERA* try {  
 *// Unbind cualquier caso de uso existente* cameraProvider.unbindAll()  
  
 *// Bind los casos de uso a la cámara* cameraProvider.bindToLifecycle(  
 lifecycleOwner,  
 cameraSelector,  
 preview,  
 imageAnalysis  
 )  
  
 Log.d(TAG, "CameraX configurado correctamente")  
 } catch (e: Exception) {  
 Log.e(TAG, "Error al configurar CameraX", e)  
 }  
 **}**, ContextCompat.getMainExecutor(ctx))  
  
 previewView  
 **}** )  
  
 *// Mostrar el bitmap con las detecciones (encima de la vista de la cámara)* detectionBitmap?.*let* **{** bitmap **->** Image(  
 bitmap = bitmap.*asImageBitmap*(),  
 contentDescription = "Detección",  
 modifier = Modifier.*fillMaxSize*()  
 )  
 **}** *// Panel de información superior* Box(  
 modifier = Modifier  
 .*fillMaxWidth*()  
 .*align*(Alignment.TopCenter)  
 .*background*(Color.Black.copy(alpha = 0.5f))  
 .*padding*(8.*dp*)  
 ) **{** Column **{** Text(  
 text = stringResource(R.string.*detection\_info*),  
 color = Color.White,  
 fontWeight = FontWeight.Bold,  
 fontSize = 16.*sp* )  
 Spacer(modifier = Modifier.*height*(4.*dp*))  
 Row **{** Text(  
 text = stringResource(R.string.*fps*, fps),  
 color = Color.White  
 )  
 Spacer(modifier = Modifier.*width*(12.*dp*))  
 Text(  
 text = stringResource(R.string.*inference\_time*, inferenceTime),  
 color = Color.White  
 )  
 **}** Text(  
 text = stringResource(R.string.*detection\_results*, detectionCount),  
 color = Color.White  
 )  
 **}  
 }** *// Controles en la parte inferior* Box(  
 modifier = Modifier  
 .*fillMaxWidth*()  
 .*align*(Alignment.BottomCenter)  
 .*background*(Color.Black.copy(alpha = 0.5f))  
 .*padding*(8.*dp*)  
 ) **{** Row(  
 verticalAlignment = Alignment.CenterVertically,  
 modifier = Modifier.*fillMaxWidth*()  
 ) **{** Text(  
 text = "Información de depuración",  
 color = Color.White,  
 modifier = Modifier.*weight*(1f)  
 )  
 Switch(  
 checked = showDebugInfo,  
 onCheckedChange = **{** showDebugInfo = **it }** )  
 **}  
 }** }  
 **}**}  
  
@ExperimentalGetImage  
private fun processImage(  
 imageProxy: ImageProxy,  
 objectDetector: ObjectDetector,  
 onResult: (Bitmap, ObjectDetector.DetectionResult) -> Unit  
) {  
 try {  
 *// Convertir ImageProxy a Bitmap (con corrección de color)* val bitmap = ImageUtils.imageProxyToBitmap(imageProxy)  
  
 *// Rotar la imagen según la orientación* val rotationDegrees = imageProxy.*imageInfo*.*rotationDegrees* val rotatedBitmap = ImageUtils.rotateBitmap(bitmap, rotationDegrees)  
  
 *// Ejecutar detección* val result = objectDetector.detect(rotatedBitmap)  
  
 *// Dibujar resultados en el bitmap* val resultBitmap = PrePostProcessor.drawDetectionResults(rotatedBitmap, result, objectDetector)  
  
 *// Devolver el resultado* onResult(resultBitmap, result)  
 } catch (e: Exception) {  
 Log.e("CameraScreen", "Error procesando imagen", e)  
 } finally {  
 *// Siempre cerrar el ImageProxy* imageProxy.close()  
 }  
}

**MainActivity.kt**:



**Declaración de la Clase y Atributos**

class MainActivity : ComponentActivity() {

private val TAG = "MainActivity"

**Anotaciones y Método onCreate**

@ExperimentalGetImage

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

Log.d(TAG, "Iniciando aplicación YOLOv11 Camera App")

**Verificación de Recursos**

*// Verificar que los archivos necesarios existen en assets*

try {

assets.open("coco.names").close()

Log.d(TAG, "Archivo coco.names encontrado en assets")

} catch (e: Exception) {

Log.e(TAG, "Error: Archivo coco.names no encontrado en assets", e)

}

try {

assets.open("models/yolo11n\_mobile.pt").close()

Log.d(TAG, "Modelo yolo11n\_mobile.pt encontrado en assets/models")

} catch (e: Exception) {

Log.e(TAG, "Error: Modelo yolo11n\_mobile.pt no encontrado en assets/models", e)

}

**Configuración de la Interfaz con Compose**

setContent {

Capitulo4\_practica1Theme {

Surface(

modifier = Modifier.fillMaxSize(),

color = MaterialTheme.colorScheme.background

) {

CameraScreen(applicationContext)

}

}

}