AR Demos

[Github: github.com/jhygate/AR-demos-android](https://github.com/jhygate/AR-demos-android) Contact: jhygate@gmail.com

This app provides 3 examples of augmented reality on android mobile. This documentation intends to explain the code and help provide a template to extend it further.

A screenshot of a computer

Description automatically generated with medium confidence A screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated with medium confidenceA picture containing floor, indoor, furniture, cluttered

Description automatically generated

The app contains 3 demos:

* OpenCV2D
  + A demonstration app to apply filters to the camera
* OpenCV3D
  + This demo contains two activities, a calibration activity and an activity that displays 3d pose information on the camera
* ARCore
  + A small demo showing the smallest scale ARCore app that can be made

# OpenCV

OpenCV is an opensource library for computer vision. Three activities in the app primarily use OpenCV for augmenting the camera feed.

## ArUco

ArUco is an opensource library for camera pose estimation using square markers. This is the basis for the 3D OpenCV demo, and how the orientation of the camera is calculated.

## Setting up OpenCV for Android

The following steps document how to move from a new Android Studio App to an app that can run Java OpenCV on mobile with Aruco.

(https://medium.com/android-news/a-beginners-guide-to-setting-up-opencv-android-library-on-android-studio-19794e220f3c)

Step 1: Download OpenCV Android Library

Go to https://,.com/chaoyangnz/opencv3-android-sdk-with-contrib page and download the latest OpenCV Android library

When the download completes, you should extract the contents of the zip file into a folder.

Graphical user interface, text, application, email

Description automatically generated

Step 2: Setup project

Create a new Android project using Android Studio only if you have not created one already for your computer vision project.

Note: Skip this step if you already have an Android project you want to use the OpenCV library in.

Graphical user interface, text, application

Description automatically generated

Step 3: Import OpenCV Module

After successfully creating an Android project, it is time to import the OpenCV module into your Android project. Click on File -> New -> Import Module…

Text

Description automatically generated

It should bring up a popup like the image below where you can select the path to the module you want to import.

Graphical user interface, text, application

Description automatically generated

Browse to the folder where you extracted the OpenCV Android library zip file contents. Select the java folder inside of the sdk folder.

A screenshot of a computer

Description automatically generated

After selecting the correct path and clicking OK, you should get a screen like the image below.

Graphical user interface, text

Description automatically generated

Click on Next to go to the next screen. On the next screen (the image below) you should leave the default options checked and click on Finish to complete the module import.

Text

Description automatically generated

Step 4: Fixing Gradle Sync Errors

You should get a Gradle build error after you finish importing the OpenCV library. This happens because the library is using an old Android SDK that you probably don’t have installed yet.

Text

Description automatically generated

To quickly fix this error, switch from the Android pane to the Project pane on the left side of Android Studio.

Text

Description automatically generated

Browse to OpenCV library module and open its build.gradle file.

Text

Description automatically generated

To fix the error, you just have to change the compileSdkVersion and targetSdkVersion to the latest Android SDK version or the one you have installed on your PC. After changing the version you should click on the sync button so that Gradle can sync the project.

Quick tip: buildToolsVersion can be ignored

Text

Description automatically generated

Step 5: Add the OpenCV Dependency

To work with the OpenCV Android library, you have to add it to your app module as a dependency. To easily do this on Android Studio, click on File -> Project Structure.

Text

Description automatically generated

When the project structure dialog opens, click on the app module or any other module that you want to use OpenCV library in.

After navigating to the module, click on the Dependencies tab. You should see a green plus button on the far right of the dialog, click on it and select Module dependency.

Text

Description automatically generated

When the choose modules dialog opens, select the OpenCV library module and click on OK.

A screenshot of a computer

Description automatically generated

When you return to the dependencies page, confirm that the module was actually added as a dependency then click on the OK button to continue.

Text

Description automatically generated

Step 6: Add Native Libraries

On your file explorer, navigate to the folder where you extracted the content of the OpenCV Android library zip file. Open the sdk folder and then the native folder (Use the image below as a guide).

Graphical user interface, text, application, email

Description automatically generated

Copy the libs folder in the native folder over to your project app module main folder (Usually ProjectName/app/src/main).

Graphical user interface, text, email

Description automatically generated

Rename the libs folder you just copied into your project to jniLibs.

Graphical user interface, text, email

Description automatically generated

After following these steps, you should be able to import use OpenCV and Aruco within your code.

## Creating a template OpenCV app

A default OpenCV app will take the input camera frame, manipulate it in some way and then return a new camera frame to be displayed on screen. The following steps will explain how to create a default OpenCV app that can do that.

To successfully use the phone camera, your app should have the camera permission added to its AndroidManifest.xml file.

|  |  |
| --- | --- |
|  | <uses-permission android:name="android.permission.CAMERA"/> |
|  |  |
|  | <uses-feature android:name="android.hardware.camera" android:required="false"/> |
|  | <uses-feature android:name="android.hardware.camera.autofocus" android:required="false"/> |
|  | <uses-feature android:name="android.hardware.camera.front" android:required="false"/> |
|  | <uses-feature android:name="android.hardware.camera.front.autofocus" android:required="false"/> |
|  | The app needs to display the camera frames on screen, so a JavaCameraView needs to be added into the layout for the camera to display in. Edit the layout file to match the following:  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"    android:layout\_width="match\_parent"  android:layout\_height="match\_parent">  <org.opencv.android.JavaCameraView  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content" />  </RelativeLayout> |

The Java activity code can now be written.

In Activity, define the following and fix the imports accordingly (Alt + Enter, click import):

//A Tag to filter the log messages

private static final String TAG = "Example::HelloVisionWorld::Activity";

//A class used to implement the interaction between OpenCV and the //device camera.

private CameraBridgeViewBase mOpenCvCameraView;

//This is the callback object used when we initialize the OpenCV //library asynchronously

private BaseLoaderCallback mLoaderCallback = new BaseLoaderCallback(this) {

@Override

//This is the callback method called once the OpenCV //manager is connected

public void onManagerConnected(int status) {

switch (status) {

//Once the OpenCV manager is successfully connected we can enable the camera interaction with the defined OpenCV camera view

case LoaderCallbackInterface.SUCCESS:

{

Log.i(TAG, "OpenCV loaded successfully");

mOpenCvCameraView.enableView();

} break;

default:

{

super.onManagerConnected(status);

} break;

}

}

};

 Update the onResume activity callback method to load the OpenCV library and fix the imports accordingly:

@Override  
public void onResume(){  
  
 super.onResume();  
  
 //Intialise OpenCV manager on mobile  
 OpenCVLoader.*initDebug*();  
 mLoaderCallback.onManagerConnected(LoaderCallbackInterface.*SUCCESS*);  
}

 Your activity needs to implement CvCameraViewListener2 to be able to receive camera frames from the OpenCV camera view:

public class HelloVisionActivity extends Activity implements CvCameraViewListener2

 Fix the imports error accordingly and also insert the unimplemented methods in your activity.

 In the onCreate activity callback method, we need to set the OpenCV camera view as visible and register your activity as the callback object that will handle the camera frames:

@Override

protected void onCreate(Bundle savedInstanceState) {

Log.i(TAG, "called onCreate");

super.onCreate(savedInstanceState);

getWindow().addFlags(WindowManager.LayoutParams.FLAG\_KEEP\_SCREEN\_ON);

setContentView(R.layout.activity\_hello\_vision);

mOpenCvCameraView = (CameraBridgeViewBase) findViewById(R.id.HelloVisionView);

//Set the view as visible

mOpenCvCameraView.setVisibility(SurfaceView.VISIBLE);

//Register your activity as the callback object to handle //camera frames

mOpenCvCameraView.setCvCameraViewListener(this);

}

 The last step is to receive the camera frames. In order to do so, change the implementation of the onCameraFrame callback method:

public Mat onCameraFrame(CvCameraViewFrame inputFrame) {

//We're returning the colored frame as is to be rendered on //thescreen.

return inputFrame.rgba();

}

* For the app to use the devices camera you must request permission from the user. This can be achieved by updating the OnCameraViewStarted function:
* public void onCameraViewStarted(int a, int b){  
   requestPermissions(new String[]{Manifest.permission.*CAMERA*},100); }

At this point this project should run on a phone and simply display the camera on screen.

By default the OpenCV display will be rotated by 90 degrees when drawn onto the screen. The solution used in this project is to update the OpenCV library (https://mikeheavers.com/tutorials/opencv\_cam/). This article explains the following code

To fix this add the following code to CameraBridgeViewBase found in the OpenCV library under android, replacing the existing updateMatrix function :



The camera view will not be scaled to fill the phone screen, this can be fixed by editing JavaCamera2View and JavaCameraView. Both need to be edited as different files will be used depending on the age of the device. These files are found in the OpenCV library under android.

Search for these lines in both files:

if ((getLayoutParams().width == LayoutParams.MATCH\_PARENT) && (getLayoutParams().height == LayoutParams.MATCH\_PARENT)) mScale = Math.min(((float)height)/mFrameHeight, ((float)width)/mFrameWidth);else mScale = 0;

Rather than the conditional, we can just set it to:

mScale = Math.max(((float)height)/mFrameHeight, ((float)width)/mFrameWidth);

This will scale the display to fill the screen completely.

The next issue that needs to be resolved is displaying the camera in a different resolution. By default the app will use the lowest resolution available which is not ideal. This can be fixed by creating a helper class to change camera settings. In the given project this is called CameraHelper.

Create a new class ‘CameraHelper’ in the same location as your MainActivity file containing the following:

import android.content.Context;  
import android.hardware.Camera;  
import android.hardware.Camera.PictureCallback;  
import android.hardware.Camera.Size;  
import android.util.AttributeSet;  
import android.util.Log;  
  
  
import org.opencv.android.JavaCameraView;  
  
import java.io.FileOutputStream;  
import java.util.List;  
  
  
//Helper class taken from https://docs.nvidia.com/gameworks/content/technologies/mobile/opencv\_tutorial\_camera\_control.htm  
//Allows for camera control (Used for Setting Resolution in this demo)  
public class CameraHelper extends JavaCameraView implements PictureCallback {  
  
 private static final String *TAG* = "Sample::Tutorial3View";  
 private String mPictureFileName;  
  
 public CameraHelper(Context context, AttributeSet attrs) {  
 super(context, attrs);  
 }  
  
 public List<String> getEffectList() {  
 return mCamera.getParameters().getSupportedColorEffects();  
 }  
  
 public boolean isEffectSupported() {  
 return (mCamera.getParameters().getColorEffect() != null);  
 }  
  
 public String getEffect() {  
 return mCamera.getParameters().getColorEffect();  
 }  
  
 public void setEffect(String effect) {  
 Camera.Parameters params = mCamera.getParameters();  
 params.setColorEffect(effect);  
 mCamera.setParameters(params);  
 }  
  
 public List<Size> getResolutionList() {  
 return mCamera.getParameters().getSupportedPreviewSizes();  
 }  
  
 public void setResolution(Size resolution) {  
 disconnectCamera();  
 mMaxHeight = resolution.height;  
 mMaxWidth = resolution.width;  
 connectCamera(getWidth(), getHeight());  
 }  
  
 public Size getResolution() {  
 return mCamera.getParameters().getPreviewSize();  
 }  
  
 public void takePicture(final String fileName) {  
 Log.*i*(*TAG*, "Taking picture");  
 this.mPictureFileName = fileName;  
 // Postview and jpeg are sent in the same buffers if the queue is not empty when performing a capture.  
 // Clear up buffers to avoid mCamera.takePicture to be stuck because of a memory issue  
 mCamera.setPreviewCallback(null);  
  
 // PictureCallback is implemented by the current class  
 mCamera.takePicture(null, null, this);  
 }  
  
 @Override  
 public void onPictureTaken(byte[] data, Camera camera) {  
 Log.*i*(*TAG*, "Saving a bitmap to file");  
 // The camera preview was automatically stopped. Start it again.  
 mCamera.startPreview();  
 mCamera.setPreviewCallback(this);  
  
 // Write the image in a file (in jpeg format)  
 try {  
 FileOutputStream fos = new FileOutputStream(mPictureFileName);  
  
 fos.write(data);  
 fos.close();  
  
 } catch (java.io.IOException e) {  
 Log.*e*("PictureDemo", "Exception in photoCallback", e);  
 }  
  
 }  
}

**This class provides extra functionality to interface with the camera.**

**MainActivity must now be updated to use this new class. This can be done by replacing every instance of JavaCameraView with CameraHelper.**

**Replace JavaCameraView with CameraHelper in the layout file as well.**

Text

Description automatically generated

**MainActivity can now be updated to get a list of accepted camera resolutions and display the highest one (Or any other).**

**Add the following variable definitition at the beginning of the class:**

private List<Camera.Size> mResolutionList;

**Then update the OnCameraViewStarted function to set the resolution**

//Function run when the camera is drawn to the screen  
public void onCameraViewStarted(int a, int b){  
 requestPermissions(new String[]{Manifest.permission.*CAMERA*},100); //Camera permission requested if needed  
 mResolutionList = mOpenCvCameraView.getResolutionList();  
 mOpenCvCameraView.setResolution(mResolutionList.get(0)); //Set resolution to max size  
  
}

**At this point in the project the app is very easy to extend. The only function that needs to be edited is the OnCameraFrame function. This function defines what will be drawn to the screen given an input frame, the Matrix representation of an image that is returned will be displayed.**

A template of an OpenCVApp is given in the project files called, OpenCVTemplateActivity

# OpenCV2D

This demo demonstrates 2d image manipulation based on the OpenCV app template. It applies a selection of OpenCV filters onto the image and displays it on screen.

To add an extra button:

Add a button to res/layout/activity\_main.xml:

<Button  
 android:id="@+id/buttonFill"  
 android:layout\_width="100dp"  
 android:layout\_height="40dp"  
 android:onClick="FillSwitch"  
 android:text="Fill" />

Change the ID, OnClick and text

Define the button function e.g:

public void FillSwitch(View view) {  
 filterType = 3;  
}

Define a function that manipulates the inputframe e.g:

private Mat detectEdges(Mat rgba) {  
  
 Mat edges = new Mat(rgba.size(), CvType.*CV\_8UC1*);  
 Imgproc.*cvtColor*(rgba, edges, Imgproc.*COLOR\_RGB2GRAY*, 4);  
 Imgproc.*Canny*(edges, edges, 80, 100);  
  
 return edges;  
  
}

Update onCameraFrame to change with the filter type:

public Mat onCameraFrame(CameraBridgeViewBase.CvCameraViewFrame inputFrame) {  
  
 switch (filterType){ //Different filters applied based on choice  
 case (0):  
 return inputFrame.rgba();  
 case (1):  
 return inputFrame.gray();  
 case (2):  
 return (detectEdges(inputFrame.rgba()));  
 case (3):  
 return fillShapes(inputFrame.rgba());  
  
 }  
 return inputFrame.rgba();  
}

# OpenCV3D

This demo uses ArUco and camera pose estimation to display a number of 3D effects on markers. It allows the user to toggle between detecting a singular ArUco marker and a ArUco gridboard and display the 3D axis of the marker and display a cube over it that is manipulated with the markers position and rotation.

The three markers used for this demo are shown below (calibration, single marker and gridboard):

Background pattern

Description automatically generated ![Qr code

Description automatically generated]() A picture containing text, crossword puzzle, scoreboard

Description automatically generated

## CalibrationActivity

The calibration activity asks the user to take at least 20 photos of a chessboard image from a variety of angles and orientations. OpenCv finds the locations of each square in the chessboard image and calculates camera calibration parameters from the data collected from the photos.

For accurate pose recognition camera calibration is necessary. This is because any camera will be distorted in some way (e.g., a slight fish eye lens). The relevant calibration information is stored in two variables a matrix of distortion coefficients and a camera matrix that includes information like focal length and optical centers. Further information on these values can be found in the OpenCv documentation (<https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html>)

The code for calibrating is based roughly on this OpenCV tutorial

<https://opencv-java-tutorials.readthedocs.io/en/latest/09-camera-calibration.html>

The CalibrationActivty activity is based on the OpenCVTemplateActivity, and calibration data is found in the OnCameraFrame() function.

The function that calculates the camera parameters is shown:

private void calibrateCamera() {  
 List<Mat> rvecs = new ArrayList<>(); // Return containers for calibration values  
 List<Mat> tvecs = new ArrayList<>();  
 intrinsic.put(0, 0, 1);  
 intrinsic.put(1, 1, 1);  
  
 // calibrate!  
 Calib3d.*calibrateCamera*(calibrateObjectPoints, calibrateImagePoints, imageSize, intrinsic, distCoeffs, rvecs, tvecs);  
}

Intrinsic is the cameraMatrix parameter, and distCoeffs is the distortion cooeficent parameter, These values (along with unnececary pose estimation values for each image taken, rvecs and tvecs) are calculated by Calib3d.calibrateCamera. The purpose of this activity is to populate calibrateObjectPoints and calibrateImagePoints arrays so that the parameters can be calculated. This is achieved in onCameraFrame:

public Mat onCameraFrame(CameraBridgeViewBase.CvCameraViewFrame inputFrame) {  
 Mat outputImage = inputFrame.rgba();  
 imageSize = outputImage.size();  
  
 boolean found = Calib3d.*findChessboardCorners*(inputFrame.gray(), boardSize, imageCorners,  
 Calib3d.*CALIB\_CB\_ADAPTIVE\_THRESH* + Calib3d.*CALIB\_CB\_NORMALIZE\_IMAGE* + Calib3d.*CALIB\_CB\_FAST\_CHECK*); //Finds image corners in camera frame

Calib3d.findChessboardCorners populates imageCorners with the positions of each chessboard square if the chessboard is detected in the input frame.

if (found) {  
 if (calibrateSnapshotPressed) {  
 calibrateSnapshotPressed = false;  
  
 calibrateImagePoints.add(imageCorners);  
 imageCorners = new MatOfPoint2f();  
 calibrateObjectPoints.add(obj);  
 calibrateSuccesses++;

This will run if a chessboard has been detected and the user presses the button to take a calibration photo. It will add to the CalibrateImagePoints and CalibrateObjectPoints array with the details of the chessboard orientation in that photo. The number of successes is also incremented. This variable is used to ensure enough calibration photos are taken to get an accurate output of parameters. Try reducing the number of photos needed for calibration to see how it affects pose estimation in OpenCV3DActivity.

The rest of the code in this activity is setting up parameters for the activity (e.g., width/height of the chessboard marker) or effects UI changes (displaying the number of calibration photos taken and enabling/disabling button presses when necessary)

## OpenCV3DActivity

From the menu the user has two options to start OpenCV3DActivity, the user can do their own camera calibration, or use default values. The default values are the result of one instance of calibration that seemed accurate on a variety of android devices but is not guaranteed to be accurate on all.

The calibration values are assigned in onCameaViewStarted:

//Default camera calibration values  
cameraMatrix = Mat.*eye*(3, 3, CvType.*CV\_32FC1*);  
cameraMatrix.put(0, 0,  
 719.9602633448541, 0, 447,  
 0, 719.9602633448541, 335.5,  
 0, 0, 1);  
distCoeffs = new MatOfDouble();  
double[] distCoeffsArray = {-0.1007,  
 0.2118,  
 0,  
 0,  
 -0.6476};  
distCoeffs.fromArray(distCoeffsArray);  
  
//Sets the calibration values if the user calibrated  
String calibrated = intent.getStringExtra("calibrated");  
if (calibrated.equals("true")) {  
  
 long addr = intent.getLongExtra("camAddr", 0);  
 Mat tempImg = new Mat(addr);  
 cameraMatrix = tempImg.clone();  
  
 addr = intent.getLongExtra("distAddr", 0);  
 Mat testValue = new Mat(addr);  
 Log.*i*(*TAG*, testValue.dump() + " words");  
 distCoeffs = Mat\_to\_MatOfDouble(testValue);  
}

The default values are initially assigned and based on the intent received (a variable which contains data passed between activities) either the default values are kept the same or assigned with the values passed from the calibrationActivity via the intent.

This activity is also based on the OpenCVTemplateActivity. Markers are recognized and effects are drawn to the screen using the OnCameraFrame() function. The app contains the following buttons:

* Find Board/ Find Single Marker
  + Toggles between detecting the supplied single square Aruco marker and the Aruco gridboard.
  + The gridboard demonstrates pose estimation that allows partial occlusion of some of the board. This is useful as some of the marker can be obstructed and it will still function.
* Enable/Disable Draw Marker
  + Toggles drawing the marker border and marker ID
* Enable/Disable Draw Axis
  + Toggles drawing the XYZ axis of the marker, displaying the markers 3D rotation
* Enable/Disable Marker Image
  + Toggles drawing a 2d Image over the marker
* Enable/Disable Model
  + Toggles drawing a cube over the marker

## onCameraFrame()

The onCameraFrame function will be described here:

public Mat onCameraFrame(CameraBridgeViewBase.CvCameraViewFrame inputFrame) {  
 outputImage = inputFrame.rgba();  
  
 rvec = new Mat(); //Reset rotation and translations  
 tvec = new Mat();  
  
 try { //Returns the input frame if detection errors  
  
 Dictionary dictionary;  
 if (BoardSingleButton.isChecked()) { //Sets the correct marker type  
 dictionary = dictionarySingleMarker;  
 } else {  
 dictionary = dictionaryBoard;  
 }  
  
  
 Aruco.*detectMarkers*(inputFrame.gray(), dictionary, markerCorners, markerIds); //Detects marker location values

Output image is a matrix representing the inputframe, the inputframe is converted to a matrix with .rgba(). This value is drawn to and then returned to be displayed on screen.

Rvec and tvec are rotation and translation matrices that describe the pose of an object relative to the camera. These are reset to be calculated later in the function.

The dictionary is a value that describes the marker to be detected. The dictionary being used is different depending on the marker type chosen, so the dictionary is assigned here.

Aruco.detectMarkers takes a greyscale image (inputframe.rgba()) and a dictionary of the type of markers to be found. It then populates the markerCorners array with a list of the on screen 2d cooridinaes of each marker on screen. markerIds is also populated with the ID of each marker on screen.

if (MarkerImageButton.isChecked() & markerIds.rows() > 0) { //Draws marker image if selected  
 outputImage = drawMarkerCovers(outputImage);  
}

If the user selects the button to draw the marker image, the type of marker doesn’t matter, it will draw over every marker on screen with the image. This function checks if there is at least 1 marker detected (markerIds.rows > 0) and draws over them. The drawMarkerCovers function will be described in more detail later.

Imgproc.*cvtColor*(outputImage, outputImage, Imgproc.*COLOR\_RGBA2BGR*);  
  
if (BoardSingleButton.isChecked()) { //If a single marker is detected  
 if (markerIds.rows() > 0) {  
 Aruco.*estimatePoseSingleMarkers*(markerCorners, 0.01f, cameraMatrix, distCoeffs, rvec, tvec); //Calculate rotation and translation matrices  
  
  
 if (DrawAxisButton.isChecked()) { //If axis is being drawn  
 Aruco.*drawAxis*(outputImage, cameraMatrix, distCoeffs, rvec, tvec, 0.01f);  
 }  
 if (DrawModelButton.isChecked()) { //If cube is being drawn  
 float[] translation = {0, 0, 0};  
 int scale = 1;  
 outputImage = drawCube(outputImage, rvec, tvec, translation, scale);  
 }  
 if (DrawMarkerButton.isChecked()) { //Draw marker borders  
 Aruco.*drawDetectedMarkers*(outputImage, markerCorners, markerIds, new Scalar(0, 255, 0)); //BGR needed to draw markers... for some reason  
 }  
  
 }

The buttons that can be pressed all draw to the screen using Aruco functions. Aruco requires the image to be in a BGR format so the output image is temporarily converted to BGR to process these actions.

This section of code deals with enabled buttons for if a single marker is being detected as there are slight changes in how to deal with a markerboard vs a marker.

The function Aruco.estimatePoseSingleMarkers populates rvec and tvec matrices which describes the pose of the marker. This function calculates these values form the marker corners detected in Aruco.detectMarkers, 0.01f is the given size of the marker (in metres), and the camera calibration values are also passed.

The calculated pose values can then simply be passed into Aruco.drawAxis (if the button has been enabled) and the 3D axis will be drawn to the output image.

drawCube is another function, that will be described in more depth later, that will draw a cube over the marker using rvec and tvec, the cube can be translated and scaled with the 2 other input parameters.

Finally, the markers border will be drawn with Aruco.drawDetectedMarkers if the button is chosen. As no 3D calculations are needed her, tvec and rvec are not passed into this function.

} else if (!BoardSingleButton.isChecked()) { //If the marker board is being detected  
 if (markerIds.rows() > 0) {  
 Aruco.*estimatePoseBoard*(markerCorners, markerIds, board, cameraMatrix, distCoeffs, rvec, tvec); //Calculate rotation and translation matrices  
  
 if (DrawAxisButton.isChecked()) {  
  
 Aruco.*drawAxis*(outputImage, cameraMatrix, distCoeffs, rvec, tvec, 0.02f);  
 }  
 if (DrawModelButton.isChecked()) {  
 float[] translation = {0.05f, -0.05f, 0};  
 int scale = 4;  
 outputImage = drawCube(outputImage, rvec, tvec, translation, scale);  
 }  
 if (DrawMarkerButton.isChecked()) { // check current state of a toggle button (true or false).  
 //BGR needed to draw markers... for some reason  
 Aruco.*drawDetectedMarkers*(outputImage, markerCorners, markerIds, new Scalar(0, 255, 0)); //BGR needed to draw markers... for some reason  
 }  
 }  
}  
 Imgproc.*cvtColor*(outputImage, outputImage, Imgproc.*COLOR\_BGR2RGBA*);  
  
}catch (Exception e){  
 Log.*e*(*TAG*, String.*valueOf*(e));  
}  
return outputImage;

The code to deal with an ArucoBoard is functionally identical, the only differences being Aruco.estimatePoseBoard over Aruco.estimateSingleMarker and the translation and scale values for drwanCube().

The output image is converted back to RGB and then returned, which displays the manipulated image on screen.

## drawMarkerCovers()

This function implements a simple form of AR, given marker details it overlays an image onto the marker.

public Mat drawMarkerCovers(Mat inputImage) {  
 //Draws the smile.png over each marker onscreen  
  
 for (int i = 0; i < markerIds.rows(); i++) {  
  
 MatOfPoint2f pts\_dst = Mat\_to\_MatOfPoint2f(markerCorners.get(i)); //Type conversion needed for findHomography  
 MatOfPoint pts\_dst1 = Mat\_to\_MatOfPoint(markerCorners.get(i));  
  
 Mat matTransform = Calib3d.*findHomography*(pts\_src, pts\_dst); //Calculates the transformation needed to apply the bitmap to the marker  
 Mat warpedImg = Mat.*zeros*(inputImage.rows(), outputImage.cols(), outputImage.type());  
  
 Imgproc.*warpPerspective*(smileImage, warpedImg, matTransform, outputImage.size());//Warps the smileImage using matTranform  
  
 Mat mask = Mat.*zeros*(inputImage.rows(), inputImage.cols(), CvType.*CV\_8UC1*);  
 Imgproc.*fillConvexPoly*(mask, pts\_dst1, new Scalar(255, 255, 255));//Creates a mask over the onscreen marker  
  
 warpedImg.copyTo(inputImage, mask);//copies the warped image only to the mask of the output  
 }  
 return inputImage;  
}

The code iterates through every marker detected and draws an image to it (in this demo a smiley face).

Due to the porting of OpenCV from C to Java for android some variable types must be converted, and functions do not exist to convert between every type of matrix. In this project functions have been written manually to convert between these types.

Calib3d.findHomography calculates the transformation needed to manipulate an image to be the same shape and size of another. It returns the matrix that describes the transformation from the source points (pts\_src, the coordinates of each corner of the image to be displayed) to the destination points (pts\_dst, the location of each corner of a marker).

A blank image is created, warpedImg, the transformation is applied in Imgproc.warpPerspective and saved to warpedImg.

A blank mask is created and filled only over the marker using Imgproc.fillConvexPoly. The warpedImg of the smiley face is then copied to the output image only over the mask (the marker). This results in the image covering the marker completely.

## drawCube()

This function draws a cube to a marker that follows the rotation and position of the marker. It takes the image, the rotation and translation matrices as well as translation and scale parameters for the cube.

public Mat drawCube(Mat img, Mat rvecsIn, Mat tvecsIn, float[] translation, int scale) {  
 //Draws a 3D cube over the marker that is affected by its pose  
  
 try {  
  
 //Generates a cube of points and puts it into a matrix axis  
 List<Point3> axisList = new ArrayList<>();  
 float[][] axisArray = {{-0.005f, -0.005f, 0}, {-0.005f, 0.005f, 0}, {0.005f, 0.005f, 0}, {0.005f, -0.005f, 0},  
 {-0.005f, -0.005f, 0.01f}, {-0.005f, 0.005f, 0.01f}, {0.005f, 0.005f, 0.01f}, {0.005f, -0.005f, 0.01f}};  
 int[][] relations = {{0, 1, 2, 3}, {0, 4, 7, 3}, {0, 1, 5, 4}, {6, 5, 1, 2}, {6, 2, 3, 7}, {4, 5, 6, 7}};  
  
  
 for (int point = 0; point < axisArray.length; point++) {  
 axisArray[point][0] = (axisArray[point][0] \* scale) + translation[0];  
 axisArray[point][1] = (axisArray[point][1] \* scale) + translation[1];  
 axisArray[point][2] = (axisArray[point][2] \* scale) + translation[2];  
 axisList.add(new Point3(axisArray[point][0], axisArray[point][1], axisArray[point][2]));  
 }  
 MatOfPoint3f axis = new MatOfPoint3f();  
 axis.fromList(axisList);

The function begins by defining the model to be drawn. This is done simply by an array of 3D points (axisArray) and an array of which points should be joined (relations). The axisArray is then translated and scaled and converted into MatOfPoint3f type which is needed for use in other functions.

MatOfPoint2f imagePoints = new MatOfPoint2f();  
Calib3d.*projectPoints*(axis, rvecsIn, tvecsIn, cameraMatrix, distCoeffs, imagePoints); //Projects the 3D values to 3D values translated and rotated to fit the pose (rvecs, tvecs)

Imagepoints is created, a matrix of 2d coordinates, this is then populated using Calib3d.ProjectPoints which used rvec and tvec to project the points from 3D to 2D onscreen coordinates.

//Using the relations array, draws contours around each face  
for (int faceVal = 0; faceVal < relations.length; faceVal++) {  
  
 MatOfPoint faceContour = new MatOfPoint();  
 List<Point> face = new ArrayList<>();  
 for (int pointVal = 0; pointVal < relations[faceVal].length; pointVal++) {  
 face.add(new Point(imagePoints.get(relations[faceVal][pointVal], 0)[0], imagePoints.get(relations[faceVal][pointVal], 0)[1]));  
 }  
  
 faceContour.fromList(face);  
 List<MatOfPoint> finalPoints = new ArrayList<MatOfPoint>();  
 finalPoints.add(faceContour);  
 Imgproc.*drawContours*(img, finalPoints, -1, new Scalar((60 \* (faceVal + 1)) % 255, (30 \* (faceVal + 1)) % 255, (115 \* (faceVal + 1)) % 255), 5);  
}  
  
return img;

The rest of the function simply iterates through the relations array and draws each face as a contour (each array contained inside of relations describes one face). The image is then returned.

## OpenGL

To render 3D models OpenGL should be used (<https://docs.opencv.org/4.5.2/d2/d3c/group__core__opengl.html>). The biggest issue in combing OpenCV and OpenGL is converting tvec and rvec values to values OpenGL can deal with (https://answers.opencv.org/question/23089/opencv-opengl-proper-camera-pose-using-solvepnp/)

# ARCore

ARCore is the recommended library for android developers to use AR in an app (https://developers.google.com/ar). It is included by default with Android so no libraries need to be downloaded to ensure this can be used. It is capable or marker-based AR and markerless (as demonstrated in this demo).