**Compile instructions**

1. Download NDK **r17c**.
2. Download **OpenCV 3.4.9** in <https://opencv.org/releases/>

## Change your NDK and SDK path in *local.properties.*

1. Change your OpenCV path in *app/build.gradle.* (the current setting is below)

DOpenCV\_DIR=C:/work/Japan/OpenCV-android-sdk/sdk/native/jni/

1. Change your header and library paths in [*app*](https://github.com/wzhy2000/ARwall/tree/master/app)*/[src](https://github.com/wzhy2000/ARwall/tree/master/app/src)/*[*main*](https://github.com/wzhy2000/ARwall/tree/master/app/src/main)*/[cpp](https://github.com/wzhy2000/ARwall/tree/master/app/src/main/cpp)/CMakeLists.txt* (the current settings are below)

set(pathToOpenCv C:/work/Japan/OpenCV-android-sdk/sdk)

set(pathJniLibs C:/work/Japan/ARwall/app/src/main/jniLibs)

set(pathCommonm C:/work/Japan/ARwall/common)

**Code overview**

*<project\_path>/app/src/main*:

* *MainActivity.java*: only one activity.
* *CameraClass.java*: It contains CameraClass that manages the device’s camera. Its JNI calls are implemented in *cameraClass.cpp*.
* [*MyGLSurfaceView.java*](https://github.com/wzhy2000/ARwall/blob/master/app/src/main/java/com/example/arwall/MyGLSurfaceView.java)*: It* overrides GLSurfaceView and creates GLES context to control the display on the camera preview.
* [*MyGLRenderer.java*](https://github.com/wzhy2000/ARwall/blob/master/app/src/main/java/com/example/arwall/MyGLRenderer.java)*:* It creates a custom renderer for drawing on the customized MyGLSurfaceView.
* *assets/*: It contains text pictures, marker pictures, and shader files for GLES models.
* *assets/shaders/back.vsh* and *back.fsh*: These are shaders used for rendering by BackTexture.
* *assets/shaders/picture.vsh* and *picture.fsh*: These are shaders used for rendering by TextPicture.
* *cpp/nativeCode/simpleARClass/simpleARClass.cpp*: It has methods of SimpleARClass that implements all the AR-related algorithms to detect key points and paste the text pictures.
* *cpp/nativeCode/common/backTexture.cpp*: It contains implementation of the class BackTexture that uses GLES to render an OpenCV Mat containing the camera image.
* *cpp/nativeCode/common/textPicture.cpp*: It contains implementation of the class TextPicture that uses GLES to render a text picture using transformation.
* *cpp/nativeCode/common/myGLCamera.cpp*: We have added a couple of functions to MyGLCamera that help in pose estimation.
* *cpp/nativeCode/common/misc.cpp*: We have added functions that draw the reference marker’s location and compute the camera intrinsic matrix.

**Main workflow**

1: Starting from the main activity (MainActivity), which includes connecting the camera (CameraClass) and overriding preview area (MyGLSurfaceView) with GLES. Here we use Camera1 API interface for simplicity.

2: Creating AR-related handler object gSimpleARObject (simpleARClass) in the main activity. This object will handle two requests from the Java codes:

* Call gSimpleARObject->ProcessCameraImage to process new camera image invoked by SendCamImageToNative in *CameraClass.Java*;
* Call gSimpleARObject->Render to paint the preview area invoked by DrawFrameNative in *MyGLRenderer.java*

3: Detecting key points and finding matched positions in the camera image are handled in simpleARClass::ProcessCameraImage.

* Only one marker file is loaded and its key points are saved in the simpleARClass::PerformGLInits now.
* We use BackTexture to draw the camera image at real time, and key points in red circles, and matched key points in blue big circle, and red frame for matched area for debug.
* We use TextPicture to paste the text picture once the marker is detected in the camera image.

4: The rendering of camera image is invoked by DrawFrameNative.

* Although we calculate the 3D model (mvpMat, model-view-project matrix) in SimpleARClass::Render, we don't use it to draw the 3D model of text pictures (Transformation is used, not normal mapping).
* If a new camera image is passed to the app, the BackTexture object will load the new image copy (cameraImageForBack) and then show it.
* If a new matched result is available, the TextPicture object will render the text picture using matched positions calculated in ProcessCameraImage.
* In order to reduce the shaking of text picture, we can stand still until big movement is considered (whole image movement>8 pixels, each corner movement > 30 pixels)

**Note for Match the marker in a camera image**

In SimpleARClass::SimpleARClass, we create a detector that will use ORB as the algorithm to determine key points.

cornerDetector = cv::ORB::create(750);

matcher = cv::DescriptorMatcher::create("BruteForce-Hamming");

We use OpenCV's **[detectAndCompute](http://docs.opencv.org/3.1.0/d0/d13/classcv_1_1Feature2D.html" \l "a8be0d1c20b08eb867184b8d74c15a677" \t "_blank)** API to compute the key points as well as their descriptors. A feature descriptor is usually a vector that encodes and stores the local neighborhood of a key point. This helps us to uniquely identify and match the key points in a different image. Key point detectors like **ORB** (that is used in this project), **SIFT**, **SURF**, etc., are evaluated on the basis of their ability to successfully match descriptors of key points in the query image. Note that referenceKeypoints and referenceDescriptors are private variables and store the locations and descriptors for marker matching.

To match the marker, we compute the key points and their descriptors for a new camera image. We try to match descriptors of the new key points with the stored descriptors of the marker image. If we are able to find sufficient number of matches between the two sets, then we declare that we have a match between the two images (We need to still estimate the new pose for 3D rendering). The matching is done in SimpleARClass::MatchKeypointsInQueryImage. This function is called whenever a new camera preview image is available in ProcessCameraImage. We downscale the camera image in ProcessCameraImage to speedup processing and then call MatchKeypointsInQueryImage.

In MatchKeypointsInQueryImage, we calculate new key points and their descriptors firstly using the function **detectAndCompute** in openCV.

Next we match the above descriptors with those of the marker's using k-nearest neighbour algorithm (**knnMatch**) in openCV:

// knn-match with k = 2

matcher->knnMatch(referenceDescriptors, queryDescriptors, descriptorMatches, 2);

Since we choose k=2, we get the two key points whose descriptors are the closest match to each query key point.

Empirical studies have shown that simply choosing the closest neighbor of each key point leads to a poor performance in matching features. So we compare the distances of a key point to both its neighbors and only choose those key points in which distance to the closest neighbor is less than a threshold:

for (unsigned i = 0; i < descriptorMatches.size(); i++) {

if (descriptorMatches[i][0].distance < NN\_MATCH\_RATIO \* descriptorMatches[i][1].distance) {

…

}

Then we use the pairs of matching key points to compute the homography matrix that relates the reference and query images:

homography = cv::findHomography(Keypoint2Point(sourceMatches),

Keypoint2Point(queryMatches),

cv::RANSAC, RANSAC\_THRESH, inlierMask);

We do not extract the camera pose from the homography matrix and, strictly speaking, do not need to compute it, but we use it to remove outliers from the pairs of matching key points:

for (unsigned i = 0; i < sourceMatches.size(); i++) {

if (inlierMask.at<uchar>(i)) {

sourceInlierKeypoints.push\_back(sourceMatches[i]);

queryInlierKeypoints.push\_back(queryMatches[i]);

}

}

Then we use the homography matrix to draw the position of the marker in the new camera image and to calculate the positions (matchedVertices) of the text picture in SimpleARClass::Render.

if (debugIsOn)

DrawFrameAlongShiftedCorners(matchedVertices);

…

if (renderPicture)

myTextPic->Render(&mvpMat, matchedVertices, 1.0)

**Note for rendering in a camra image**

First we copy the camera image into a private variable cameraImageForBack. Since the origin of an image in OpenCV is different from the origin in OpenGL ES, we flip the matrix containing the image. Then we detect key points in the image using ORB in DetectAndHighlightCorners for debug. We add a small circle around each key point to highlight it in the image.

Back to ProcessCameraImage; we set a flag newCameraImage to indicate to  Render that a new image is available for display. Remember that ProcessCameraImage is called in the camera’s preview callback function through a JNI call. Preview callback happens on the CameraHandlerThread. Since rendering to the GLES display can be done only on the GLES thread, we cannot render to the display in ProcessCameraImage. Hence we share cameraImageForBack between SimpleARClass::ProcessCameraImage and SimpleARClass::Render and modify it under the mutex variable cameraMutex.

|  |
| --- |
|  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |