Android Camera Imaging Test Suite / CTS Verifier

Document version: 1.14
Document date: 16th April, 2021

1. Introduction

The ITS is a framework for running tests on the images produced by an Android camera. The general goal of each test is to configure the camera in a desired manner and capture one or more shots, and then examine the shots to see if they contain the expected image data. Many of the tests will require that the camera is pointed at a specific target chart or be illuminated at a specific intensity.

The ITS framework is located in the CTS Verifier test harness (under the `cts/` area in the source) in Android 12. It is required that devices pass a specific set of ITS tests as a part of Android CTS (for the camera frameworks).

Users who only want to run the ITS tests to satisfy CTS Verifier requirements need only read Sections 2 and 3. Section 4 provides additional detail about the infrastructure for users who may want to write their own tests, or in general understand the components of this framework in greater detail.

2. Setup

There are three things that must be set up correctly to run the ITS tests: the device itself, a host machine (e.g. a Linux desktop or laptop), and the physical scene that the camera will be photographing.

2.1. Device setup

Connect the device over USB to a host machine, and grant permissions for the host to access the device over `adb`.

Install the CtsVerifier.apk onto the device, following the instructions for using the CTS Verifier application. As a quick reference, when building from source directly on the attached host machine:

```
cd cts/apps/CtsVerifier
mma -j32
adb install -r -g <OUTPUT_PATH>/CtsVerifier.apk
```

If you are installing this from a CTS Verifier bundle (`android-cts-verifier.zip`) rather than from a full Android source tree, both `CtsVerifier.apk` and the `CameraITS/` directory can be found at the top level of the zipped `android-cts-verifier` folder.

Any apps on the device that use the camera should be killed prior to running any ITS tests. The device's camera app can be used to line up the camera and the scene, but it should be killed once this is done.

2.2. Host machine setup

The ITS tests require a host machine to be connected to the device under test via USB, and make use of `adb` for device control and communication.

2.2.1. Software installation

2.2.1.1. Android SDK Platform Tool

The Android SDK platform tools must be installed, and `adb` must be in the executable path of the shell/terminal that is running on the host machine. Public released version of Android SDK platform tool can be found at: https://developer.android.com/studio/releases/platform-tools

2.2.1.2. Python 3.7.9 and numpy stack (Ubuntu)

Python 3.7.9 must be installed on the machine, along with the numpy/scipy/matplotlib/opencv stack and the Python Imaging Library. On an Ubuntu machine running Python 3.7.9, the following commands should suffice:

```
sudo apt-get install python-numpy
sudo apt-get install python-scipy
sudo apt-get install python-matplotlib
sudo apt-get install python-opencv
sudo apt-get install python-pillow
```

If these instructions don't work properly, and script errors (e.g. "import" errors) are observed, then it is recommended to instead use a bundled Python 3.7.9 distribution, as is described in the next section.

2.2.1.3. Python 3.7.9 and numpy stack (Linux, Windows or Mac OS X)

Setting up a full numpy/scipy/matplotlib/opencv installation by installing the modules separately is possible, however it can be complicated and time-consuming, especially on a Windows machine. For this reason, it is highly recommended to install a bundled distribution of Python 3.7.9 that comes with these modules. Some different bundles are listed here: http://www.scipy.org/install.html.

Of these, Anaconda has been verified to work with the ITS infrastructure, and it is available on Mac OS X, Linux, and Windows from here: http://continuum.io/downloads. After downloading and running the Anaconda installer, configure it with the following commands (in this specific order):

```
conda install opencv numpy=1.19
conda install anaconda numpy=1.19
```

After running these commands, the Anaconda installation should have versions of numpy, scipy, opencv, and matplotlib that work together, however the `conda install` mechanism may itself become broken. If other packages are needed, then reinstall a fresh Anaconda distribution, use `conda install` to get the needed packages, and then run the above commands.

Note that the Anaconda Python executable's directory must be at the front of your `PATH` environment variable; the Anaconda installer may set this up for you automatically. Once this is installed (and the above commands have been run), you'll see something like the following:

```
> python --version
Python 3.7.9
```

2.2.1.4 adb

Android debug bridge (adb) needs to be installed on the host machine. adb can be dowloaded from https://developer.android.com/studio/releases/platform-tools.

2.2.2. Environment setup

On Linux or Mac OS X, run the following command (in a terminal) from the `CameraITS/` directory, from a bash shell:

```
source build/envsetup.sh
```

This will do some basic sanity checks on your Python installation, and set up the `PYTHONPATH` environment variable, and run unit tests on the `utils/*.py` modules. If no errors are printed to the terminal then the environment is ready to run the tests.

On Windows, the bash script won't run (unless you have cygwin (which has not been tested)), but all you need to do is set your `PYTHONPATH` environment variable in your shell to point to the `CameraITS/pymodules` directory, giving an absolute path. Without this, you'll get "import" errors when running the test scripts. Note that Windows is not officially supported by CTS. So, we will do our best to keep things running, but it might become more difficult with future releases.

2.3. Physical setup

We highly recommend the automated ITS-in-a-box setup describe in the AOSP site here: https://source.android.com/compatibility/cts/camera-its-box. While ITS-in-a-box is not mandatory, the Android team generally only debugs automated test setups. The ITS-in-a-box test rigs will provide all the of the lighting, centering, and chart changing requirements.

For manual testing, the high-level requirements for running the tests are:

- Device is on a tripod, pointed at various scene setup. ITS test script will prompt to ask user to change scene setup before starting test in a new scene. See Section 2.3.1 for scene setup details.
- A steady (non-fluctuating) light source is illuminating the scene (though illumination uniformity isn't important). Don't use a fluorescent light as this introduces flicker.
- The device is connected over USB to a host machine, and isn't moved or jostled during the test run.

2.3.1. Test scenes

The tests are grouped into scene folders: `tests/scene0/`, `tests/scene1_1/`, etc. Each assumes some specific details about the scene where the camera is pointing at while running those tests.

- **scene 0**: The camera can be pointing at anything (including being face down on the desk, or using the same setup as scene 1). As of API level 30, a new test requires the device to not be put on damping materials such as clothes or handheld as the test detects device vibration based on accelerometer data.
- **scene 1_***: The camera is on a tripod pointing at a static scene containing a grey card and white background, under a constant (stable) relatively bright illumination source. This is the scene that is described above for the CTS Verifier physical setup. This scene is split in two as the number of tests is long and the CTS Verifier requires all tests in a scene to pass.
- **scene 2_***: These scenes are for testing face detection. The camera is on a tripod pointing at a static picture containing 3 human faces, under a constant (stable) relatively bright illumination source. The 3 pictures are provided in tests/scene2_*/scene2_*.pdf
- **scene 3**: This is the scene for testing image sharpness. The camera is on a tripod pointing at a static picture containing some edges, such as a printed ISO 12233 chart. The scene should be under a constant (stable) relatively bright illumination source.
- **scene 4**: This is the scene for testing aspect ratio. The camera is on a tripod pointing at a static test page containing a black circle and a square box. The scene should be under a constant (stable) relatively bright illumination source.
- **scene 5**: This is the scene for testing lens shading and color uniformity. A diffuser is placed in front of the camera. The camera is on a tripod pointing at a constant (stable) relatively bright illumination source.
- **scene 6**: This scene is a pattern of smaller circles to test the zoom function of the camera.

The `tests/inprog` directory contains a mix of unfinished, in-progress, and incomplete tests. These may or may not be useful in testing a camera HAL implementation, and if and when these tests are completed they will be moved into the scene folders.

The remainder of this section details a particular example physical setup that works well, however any setup that satisfies the above requirements will suffice.

2.3.2. Example equipment list for manual testing

The following equipment list is used in the example setup described below for manual testing.

- Light box: http://www.amazon.com/dp/B000PC4A0O
- LED desk lamp: http://www.amazon.com/dp/B00B0A18MI
- Tripod: http://www.amazon.com/dp/B0002J2TLC
- Tripod head: http://www.amazon.com/dp/B008VI7ORA
- Grey card: http://www.amazon.com/dp/B00290Y9K6
- Diffuser: http://www.edmundoptics.com/optics/windows-diffusers/optical-diffusers/opal-diffusing-glass/46168/
- Tall tripod: http://www.amazon.com/Manfrotto-MKCOMPACTADV-BK-Compact-Advanced-Tripod/dp/B00L6F1J9Y

2.3.3. Desk-side test station setup

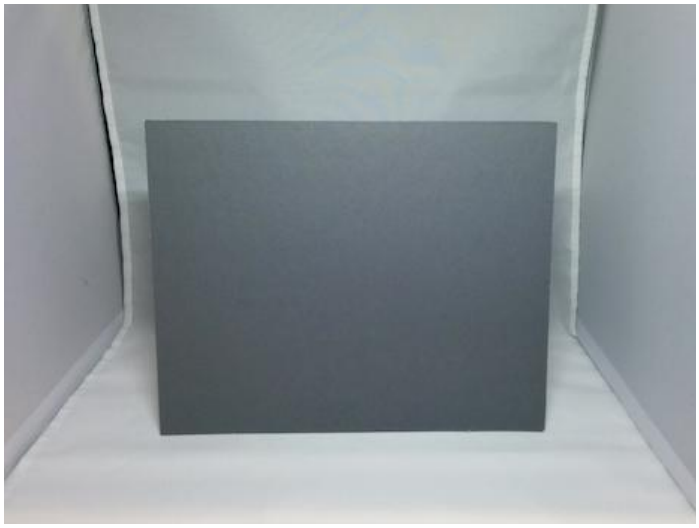
The equipment should be set up as is depicted in the following photo (using a Nexus 5 device under scene1_1 as an example):

The desk lamp should be reasonably bright: bright enough that it wouldn't be considered a "low light" environment, but not so bright that the device's flash won't make any difference to the

scene illumination when turned on. Using the specific desk lamp in this example, setting the brightness to the middle of its three adjustable brightness levels works well.

2.3.4. Scene as viewed by camera

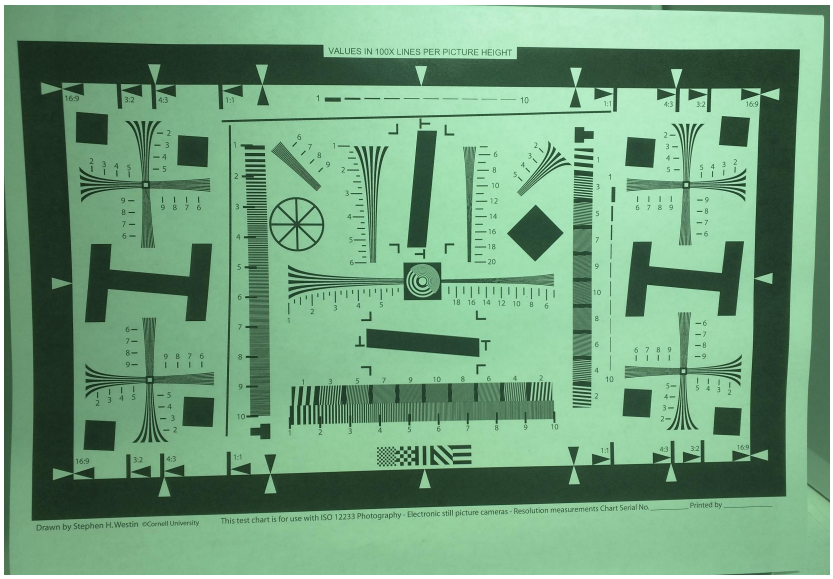
scene1_*: The camera (in the tripod) and the chart should be lined up so that a photograph of the scene from the camera looks approximately like the following:



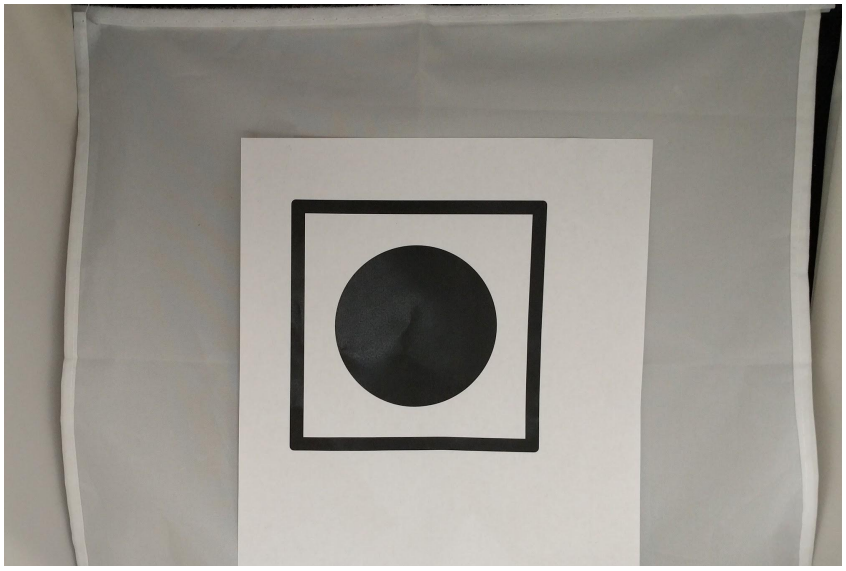
In particular, the grey card must be roughly centered in the field of view, taking up around ½ x ½ of the scene, with the white background visible on all sides around it. As is evident from the image above, a high degree of precision is not required; the most important factors are that the scene is completely static (including no light source flicker) and that the very center of the image is within the grey card area.

scene2_*: These scenes for face detection looks approximately like the following (scene2_a is shown):

scene3: This scene for sharpness related tests looks approximately like the following:

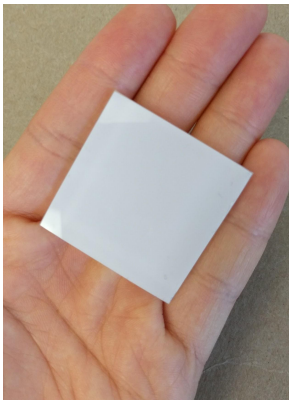


scene4: The scene for the aspect ratio test looks approximately like the following:

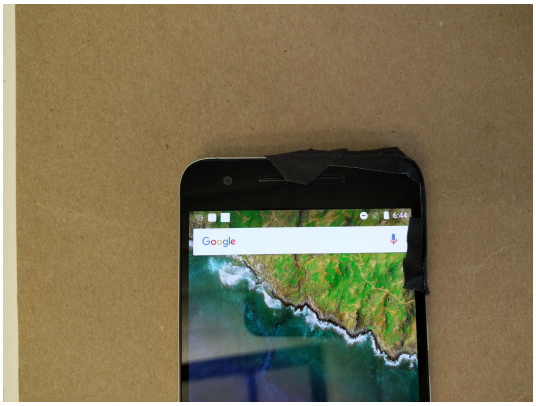


Make sure the square box is completely included in the picture, and no other complicated background is exposed in the scene except the original white one. Place the camera right in front of the test page, so that each of the four edges of the square is roughly parallel to its corresponding border of the picture. Do not point the lighting source directly at the test chart, or allow the black circle to be over-exposed.

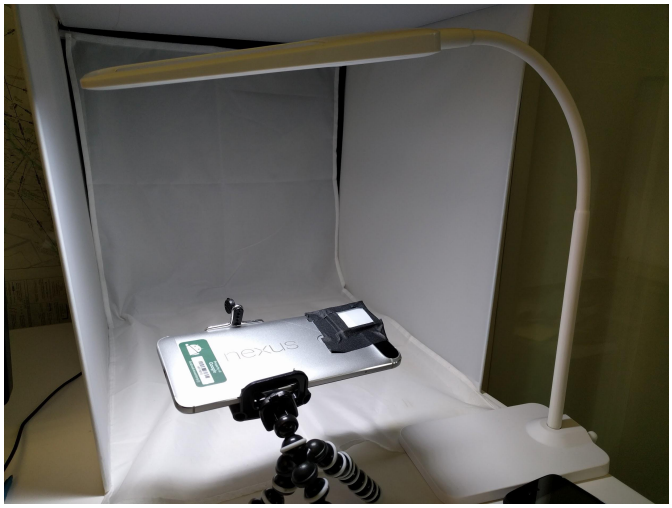
scene5: Prepare a diffuser like the following:



Carefully attach the diffuser in front of the camera, and use black tape to tape around edges to prevent light leaking in from the side. Make sure that the camera is not blocked by the black tape. The camera now looks approximately like the following:

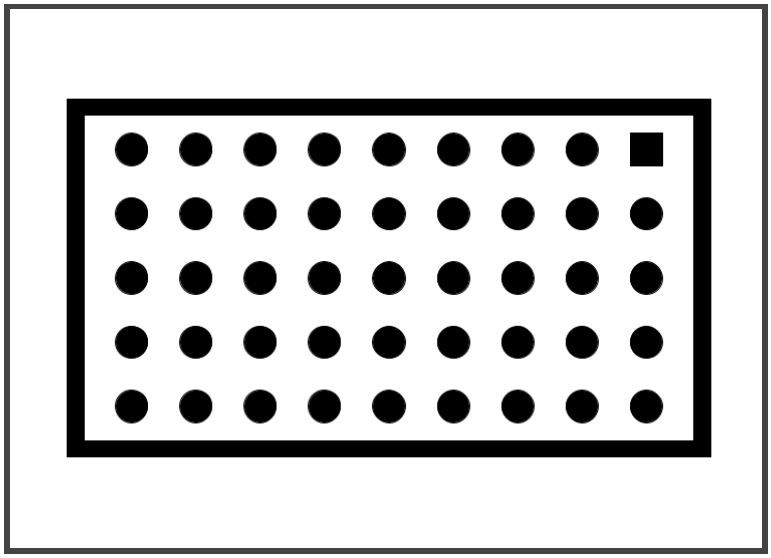


Point the camera to a gentle lighting source around 2000 lux preferably, like the following:



Images captured in this test should look approximately like the following:

scene6: The scene for test_zoom looks like the following:



The test will find the circle closest to the center, so exact alignment is not required. As the test zooms, some of the circles will be cropped out of the image. Provided a full circle can be found in the image, the test can run at the zoom level selected.

2.3.6. config.yml files

With the Mobly framework, we can set up a device under test (DUT) and chart tablet in its_base_test class. A config.yml (YAML) file is used to create a Mobly testbed. Multiple testbeds can be configured within this config file. Within each testbed's Controller section we specify device_ids to identify the appropriate Android devices to the test runner. In addition to

the device IDs, other parameters such as tablet brightness, chart distance, debug mode, camera_id, and scene_id are passed in the test class. Common test parameter values are:

```
brightness: 192  (all tablets except Pixel C)
chart_distance: 31.0  (rev1/rev1a box for FoV < 90° cameras)
chart_distance: 22.0 (rev2 test rig for FoV > 90° cameras)
```

Tablet based testing

A sample config.yml file for tablet based scenes is shown below. Note for tablet-based testing, the keyword TABLET must be present in the testbed name. During initialization, the Mobly test runner initializes these parameters and passes them to the individual tests.

```
TestBeds:
  - Name: TEST_BED_TABLET_SCENES
    # Test configuration for scenes[0:4, 6, _change]
    Controllers:
        AndroidDevice:
          - serial: 8A9X0NS5Z
            label: dut
          - serial: 5B16001229
            label: tablet

    TestParams:
      brightness: 192
      chart_distance: 22.0
      debug_mode: "False"
      chart_loc_arg: ""
      camera: 0
      scene: <scene-name>            # if <scene-name> runs all scenes
```

Sample config.yml file for tablet-based runs.

The test bed can be invoked using tools/run_all_tests.py. If no command line values are present, the test will be run with the config.yml file values. Additionally, it is possible to override the camera and scene config file values at the command line using commands similar to previous versions of Android.
For example:

```
python tools/run_all_tests.py
python tools/run_all_tests.py camera=1
python tools/run_all_tests.py scenes=2,1,0
python tools/run_all_tests.py camera=1 scenes=2,1,0
```

sensor_fusion testing

A sample config.yml file for sensor_fusion tests is shown below. For sensor_fusion testing, the testbed name should include the key word SENSOR_FUSION. Both a tablet based testbed and a sensor fusion testbed can be included in the same config.yml file: the correct testbed will be

determined by the scenes tested. Android 12 will still support both Arduino and Canakit controllers for sensor fusion.

```
Testbeds
  - Name: TEST_BED_SENSOR_FUSION
    # Test configuration for sensor_fusion/test_sensor_fusion.py
    Controllers:
        AndroidDevice:
          - serial: 8A9X0NS5Z
            label: dut

    TestParams:
      fps: 30
      img_size: 640,480
      test_length: 7
      debug_mode: "False"
      chart_distance: 25
      rotator_cntl: arduino         # cntl can be arduino or canakit
      rotator_ch: 1
      camera: 0
```

Sample config.yml file for sensor_fusion runs.

To run sensor_fusion tests with the sensor_fusion test rig simply type:

```
python tools/run_all_tests.py scenes=sensor_fusion
python tools/run_all_tests.py scenes=sensor_fusion camera=0
```

Multiple testbeds

Multiple testbeds can be included in the config file. The most common combination is to have both a tablet testbed and sensor_fusion testbed.

```
Testbeds
  - Name: TEST_BED_TABLET_SCENES
    # Test configuration for scenes[0:4, 6, _change]
    Controllers:
        AndroidDevice:
          - serial: 8A9X0NS5Z
            label: dut
          - serial: 5B16001229
            label: tablet

    TestParams:
      brightness: 192
      chart_distance: 22.0
      debug_mode: "False"
      chart_loc_arg: ""
      camera: 0
      scene: <scene-name>           # if <scene-name> runs all scenes

  - Name: TEST_BED_SENSOR_FUSION
    # Test configuration for sensor_fusion/test_sensor_fusion.py
```

```
    Controllers:
        AndroidDevice:
          - serial: 8A9X0NS5Z
            label: dut

    TestParams:
      fps: 30
      img_size: 640,480
      test_length: 7
      debug_mode: "False"
      chart_distance: 25
      rotator_cntl: arduino         # cntl can be arduino or canakit
      rotator_ch: 1
      camera: 0
```

Sample config.yml file with both tablet and sensor_fusion testbeds.

Manual testing

Manual testing continues to be supported. However, the testbed must identify testing as such with the keyword in the testbed name MANUAL. Additionally, the testbed shouldn't include a tablet id. A sample config.yml file for manual testing is shown below.

```
TestBeds:
  - Name: TEST_BED_MANUAL
    Controllers:
        AndroidDevice:
          - serial: 8A9X0NS5Z
            label: dut

    TestParams:
      debug_mode: "False"
      camera: 0
      scene: 1
```

Sample config.yml file for manual testing.

2.3.7. Rear vs. front camera testing

The example shown here is for the rear camera. The ITS tests need to also be run on the front camera (and any other cameras on the device which are exposed to the app layer), and the camera and chart will need to be positioned appropriately to be able to run those sets of tests too.

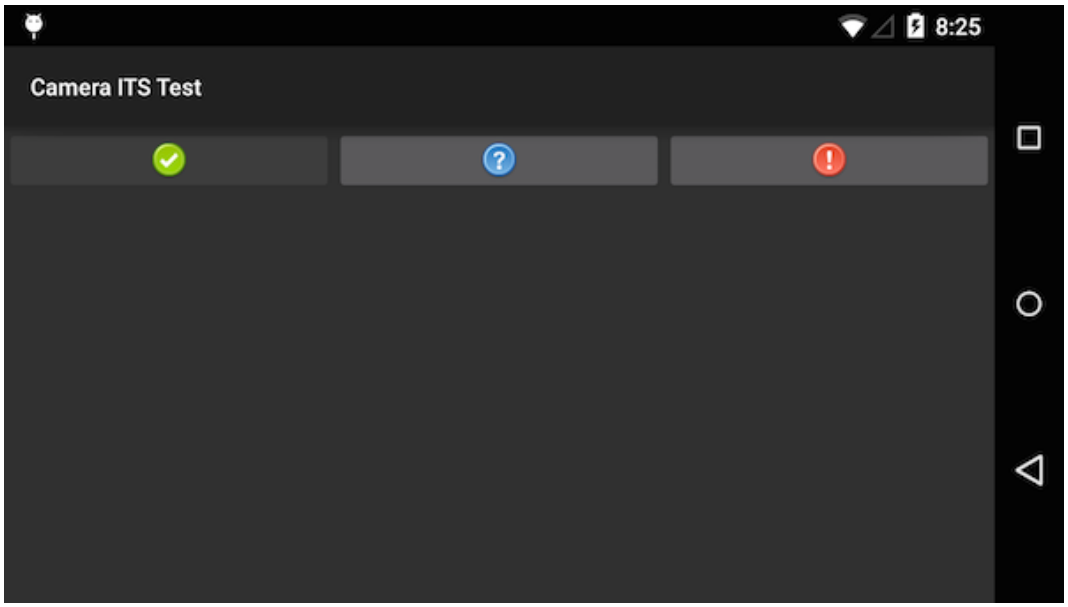
Note that the tests are run separately for each camera, so it's OK to have a single setup which serves to test each camera one at a time, with the tripod and camera repositioned in between each round of tests.

3. Running ITS tests under CtsVerifier

3.1. Invoking the tests

After the device, host machine (including environment), and physical scene have been set up as described above, the ITS tests can be run using the following process. Note the native camera app can't be active while running ITS tests as CtsVerifier app must be able to acquire camera control.

First, open the CtsVerifier app and in the menu of tests, scroll to "Camera ITS Test" and select it. There is a splash-screen dialog with some brief instructions (though note that the instructions in this doc are more detailed). Once the "OK" button is pressed, the following screen can be seen, in which the green tick ("pass") button is greyed out and not selectable:



Second, from the host PC, run the ITS tests from the top-level `CameraITS/` directory. For example, for a device with front and rear cameras, the command would be:

```
python tools/run_all_tests.py
```

The script will iterate through cameras and test scenes based on the config.yml file. For debugging setups, it is often best to run scene2_c as it is a single test and therefore runs the fastest.

For manual testing, before starting to run the set of ITS tests on each scene, the script first takes a picture of the current scene, saves it as a JPEG, prints the path to the JPEG to the console, and asks the user to confirm if the scene is properly framed; see Section 2.3.1 for

detailed scene requirement description. This “capture and confirm” flow will loop until the user confirms that the scene looks suitable. These messages appear as below:

```
Preparing to run ITS on camera 0
Start running ITS on camera:  0
Press Enter after placing camera 0 to frame the test scene:
scene1_1
The scene setup should be: A grey card covering at least the
middle 30% of the scene
Running vendor 3A on device
Capture an image to check the test scene
Capturing 1 frame with 1 format [yuv]
Please check scene setup in /tmp/tmpwBOA7g/0/scene1_1.jpg
Is the image okay for ITS scene1_1? (Y/N)
```

Each run of the script prints out a log showing either `PASS`, `FAIL`, or `SKIP` for each ITS test, where `SKIP` indicates that the test was passed because the device didn’t advertise the underlying capability that was being tested. For example, if a device doesn’t advertise through the camera interfaces that it supports DNG, then any tests related to DNG file capture will be skipped and counted as passes.

Some tests may also report a result of `FAIL*`, where the asterisk means that the test did fail, but as that particular test is not yet mandatory it doesn’t count against the device. These tests are expected to become mandatory in some future Android release, however, so whatever HAL bugs are causing them to fail should be fixed if at all possible.

Finally, if all scenes on all cameras succeed then the green tick “pass” button will become available to be tapped in the CTS Verifier app that is running on the device. Tap it, and the “Camera ITS Test” entry in the CtsVerifier menu of tests will become green.

Starting in Android 12, the results of individual tests will be reported as part of the test report to enable data mining to determine status of tests that are `SKIP` or `FAIL*`. Reporting individual test status will inform decisions on test development in future Android releases.

3.2. Collecting test debug output

Each run of the ITS tests generates a large amount of intermediate data, including many of the captured images that were used in the analysis as well as plots that are used to visualize the results of test runs. These files are all saved to a new temporary directory, the path of which is printed when the `run_all_tests.py` script begins.

These output files will be invaluable for any developers who are debugging HAL bugs that are resulting in test failures.

3.3. Sensor fusion

The test for sensor fusion requires physical movement of the device in a constrained manner, in addition to a checkerboard target. The `tests/sensor_fusion` directory contains both the test as well as the PDF documentation for how to set up and run it.

A device may only claim to support sensor fusion if this test is passed. Support for sensor fusion is advertised by returning 1 for the `android.sensor.info.timestampSource` static metadata (which corresponds to the `REALTIME` capability level for this setting).

This test must be passed for each camera device that claims to support sensor fusion.

There is an automated sensor_fusion rig to make phone movement consistent and easy. See https://source.android.com/compatibility/cts/sensor-fusion-quick-start

3.4. Non-automated tests

Not all tests are currently fully automated. Automated tests are invoked by the `run_all_tests.py` script, however non-automated tests must be manually run, and manually verified to pass.

3.4.1. DNG noise model

Devices which advertise the ability to capture RAW/DNG must also provide a noise profile in the capture result metadata of each raw shot. The `tools` directory contains a script (`dng_noise_model.py`) to generate this noise model along with some PDF documentation explaining how to set up and run it. The primary output of this script is a C code snippet than can be cut-and-pasted into the camera HAL, to implement the noise model for the device. Note that this is done per-camera-model, not per-unit, meaning that each camera on a phone (e.g. Nexus 5 front-facing and rear-facing cameras) will have its own generated noise profile, and these noise profiles will be shared across all Nexus 5 units.

Put another way, a vendor or OEM who is shipping an Android product that claims RAW/DNG support is expected to run the `dng_noise_model.py` script to generate the noise model that is embedded into the camera HAL, for each camera on the product that claims support (e.g. front + back).

Separately, there is an automated ITS test that validates that the noise model provided by the camera HAL is correct; this is `tests/scene1_1/test_dng_noise_model.py`.

4. ITS framework details (for advanced users)

This section goes into more detail about the ITS framework, and users who are only interested in running the ITS tests as a part of CTS Verifier need not read any further.

The ITS infrastructure provides a very general framework for doing image-based testing of the device, by allowing arbitrary captures to be collected and analyzed using a powerful numerical framework (numpy). Users who are interested in learning how the tests work, or in writing their own tests, will find this section relevant.

4.1. Python framework overview

The Python modules are under the `pymodules/` directory, in the `its` package.

- `utils/camera_properties_utils`: Utility functions to determine what functionality the camera supports.
- `utils/image_processing_utils`: Contains a collection of functions (built on numpy arrays) for processing captured images
- `utils/its_session_utils`: Utility functions to form an ItsSession and perform various camera actions.
- `utils/error_util`: The exception/error class used in this framework
- `utils/opencv_processing_utils`: Utility functions for image processing using opencv
- `utils/scene_change_utils`: Utility functions for scene_change test
- `utils/sensor_fusion_utils`: Utility functions for sensor_fusion test
- `utils/target_exposure_utils`: Utility functions to calculate targeted exposures based on camera properties.

Some of these modules have associated unit tests; to run the unit tests, execute the modules (rather than importing them).

4.2. Device control

The `its_session_utils.ItsSession` class encapsulates a session with a connected device under test (which is running the CTS Verifier APK). The session is over TCP, which is forwarded over ADB.

As an overview, the `its_session_utisl.ItsSession.do_capture()` function takes a Python dictionary object as an argument, converts that object to JSON, and sends it to the device over tcp which then deserializes from the JSON object representation to Camera2 Java objects (`CaptureRequest`) which are used to specify one or more captures. Once the captures are complete, the resultant images are copied back to the host machine (over TCP again), along with JSON representations of the `CaptureResult` and other objects that describe the shot that was actually taken.

The Python capture request object(s) can contain key/value entries corresponding to any of the Java `CaptureRequest` object fields.

The output surface's width, height, and format can also be specified. Currently supported formats are "jpg", "raw", "raw10", "dng", and "yuv", where "yuv" is YUV420 fully planar. The

default output surface is a full sensor YUV420 frame.

The metadata that is returned along with the captured images is also in JSON format, serialized from the `CaptureRequest` and `CaptureResult` objects that were passed to the capture listener, as well as the `CameraCharacteristics` object.

4.3. Image processing and analysis

The `image_processing_utils` module is a collection of Python functions, built on top of numpy arrays, for manipulating captured images.

Note that it's important to do heavy image processing using the efficient numpy ndarray operations, rather than writing complex loops in standard Python to process pixels. Refer to online docs and examples of numpy for information on this.

4.4. Tests

The `tests/` directory contains a number of self-contained test scripts. Most of the tests save various files in the current directory. To have all the output files put into a separate directory, run the script from that directory, for example:

```
mkdir out
cd out
python ../tests/scene1_1/test_linearity.py
```

Any test can be specified to reboot the camera prior to capturing any shots, by adding a `reboot` or `reboot=N` command line argument, where N is the number of seconds to wait after rebooting the device before sending any commands; the default is 30 seconds.

```
python tests/scene1_1/test_linearity.py reboot
python tests/scene1_1/test_linearity.py reboot=20
```

It's possible that a test could leave the camera in a bad state, in particular if there are any bugs in the HAL or the camera framework. Rebooting the device can be used to get it into a known clean state again.

By default, camera device id 0 is opened when the script connects to the unit, however this can be specified by adding a `camera=1` or similar argument to the script command line. On a typical device, camera id 0 is the main (rear) camera, and camera id 1 is the front-facing camera.

```
python tests/scene1_1/test_linearity.py camera=1
```

The `tools/run_all_tests.py` script should be executed from the top-level CameraITS directory, and it will run all of the tests in an automated fashion, saving the generated output files along with the test_log.DEBUG and test_log.INFO dumps to a temporary directory.

```
# This will run through all cameras in the config.yml file.
python tools/run_all_tests.py
```

This can be run with the `camera`, `reboot` and/or `device` arguments (see below), and in general any args provided to this command line will be passed to each script as it is executed.

```
# This will only run tests for camera 1.
python tools/run_all_tests.py camera=1
```

4.5. Docs

The `pydoc` tool can generate docs for the ITS Python modules, using the following commands (where the -w option indicates that the docs should be saved in a HTML file):

```
pydoc utils/image_processing_utils.py
pydoc -w utils/image_processing_utils.py
```

There is also a tutorial script in the `tests/` folder (named `tutorial.py`). It illustrates a number of the `image_processing_utils` and `capture_request_utils` primitives, and shows how to work with image data in general using this infrastructure. (The code is commented with explanatory remarks.)

```
python tests/tutorial.py
```