TESTING DOCUMENT

GymVision

GymVision is a unique application that utilizes computer vision to provide you with live feedback and analysis while you perform a gym exercise in front of either an android camera or a webcam connected to a windows device. The application will provide feedback in a way that will help you improve your technique while also ensuring your safety to avoid injury during exercise.

Fawaz Alsafadi – 15380871

Ayman El Gendy – 15395461

Supervisor – Alistair Sutherland

May 14th, 2020

CONTENTS

1. UNIT TESTING

1.1 FRAMEWORK

For unit testing the android application we used a tool called JUnit. JUnit is a unit testing framework for Java that is easily integrated with Android Studio IDE and allowed us to quickly begin writing tests with very little Unit testing experience for Java.

1.2 MOCKING

While unit testing for Java we ran into an issue where our application used the graphics library that android studio provides for android development. However, when using JUnit we ran into a problem where we realized that for unit testing the libraries built into android are not supported and only the base java environment was available to us when testing. After some research, we found the solution was to mock classes that were not provided by the base Java environment. One such mock is shown below in figure 1 where we had to mock the 'PointF' class which returns x and y coordinates of locations in an image.

Figure 1

```
public class FakePointF extends PointF {

    public FakePointF(float x, float y) {
        this.x = x;
        this.y = y;
    }
```

1.3 UNIT TESTS

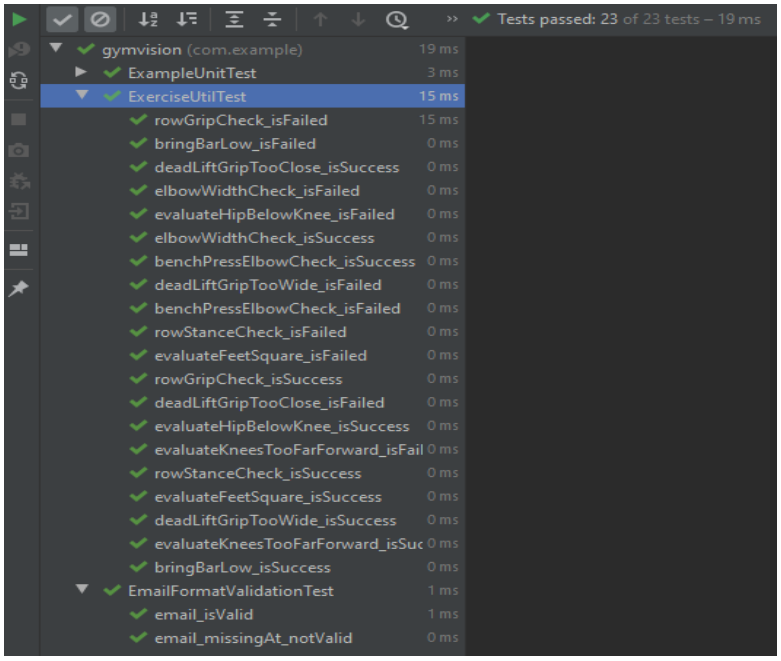
While we did not achieve a high coverage percentage of the code we focused on the most crucial logic and code in our application and the most likely to produce errors. This for us was the technique analysis logic, due to the constant tweaking and modification of our technique analysis heuristics values we wanted to ensure that we never made a change that began to produce incorrect results and give false feedback to the user. As such we created unit tests for every single exercise and individual checks, these unit tests consisted of fake user data being passed in to simulate users limb positions and for each check we created a unit test that we knew should fail and should pass. Due to the continuous integration pipeline we had set up this ensured that we never pushed a build that was giving users false feedback which could result in injury.

Figure 2 shows an example of 1 set of unit tests for the 'Full range of motion' check which determines if the user has bought the bar low enough to their chest in a bench press.

Figure 2

```
public class ExerciseUtilTest {

    @Test
    public void bringBarLow_isSuccess() {
        PointF leftWristPosition = new FakePointF( x: 160.0F,  y: 0.0F);
        PointF leftShoulderPosition = new FakePointF( x: 140.0F,  y: 0.0F);
        PointF rightWristPosition = new FakePointF( x: 100.0F,  y: 0.0F);
        PointF rightShoulderPosition = new FakePointF( x: 140.0F,  y: 0.0F);

        Keypoint leftWrist = new Keypoint( id: 1,  name: "left_wrist", leftWristPosition,  score: 100.00F);
        Keypoint leftShoulder = new Keypoint( id: 2,  name: "left_shoulder", leftShoulderPosition,  score: 100.00F);
        Keypoint rightWrist = new Keypoint( id: 1,  name: "right_wrist", rightWristPosition,  score: 100.00F);
        Keypoint rightShoulder = new Keypoint( id: 2,  name: "right_shoulder", rightShoulderPosition,  score: 100.00F);

        int score = ExerciseUtils.benchPressBringBarLow(rightWrist, leftWrist, leftShoulder, rightShoulder);

        Assert.assertEquals( expected: 1, score);
    }

    @Test
    public void bringBarLow_isFailed() {
        PointF leftWristPosition = new FakePointF( x: 30.0F,  y: 0.0F);
        PointF leftShoulderPosition = new FakePointF( x: 60.0F,  y: 0.0F);
        PointF rightWristPosition = new FakePointF( x: 120.0F,  y: 0.0F);
        PointF rightShoulderPosition = new FakePointF( x: 60.0F,  y: 0.0F);

        Keypoint leftWrist = new Keypoint( id: 1,  name: "left_wrist", leftWristPosition,  score: 100.00F);
        Keypoint leftShoulder = new Keypoint( id: 2,  name: "left_shoulder", leftShoulderPosition,  score: 100.00F);
        Keypoint rightWrist = new Keypoint( id: 3,  name: "right_wrist", rightWristPosition,  score: 100.00F);
        Keypoint rightShoulder = new Keypoint( id: 4,  name: "right_shoulder", rightShoulderPosition,  score: 100.00F);

        int score = ExerciseUtils.benchPressBringBarLow(rightWrist, leftWrist, leftShoulder, rightShoulder);

        Assert.assertEquals( expected: 0, score);
    }
```

Figure 3



We also included unit tests to verify minor things such to ensure that a user was creating an account with a valid email that included at least an '@' symbol.

Figure 4

```
@Test
public void email_isValid() {
    String email = "test@valid.email.com";

    boolean isValid = EmailUtils.validateEmail(email);

    assertTrue(isValid);
}
```

2. INSTRUMENTATION TESTING

2.1 FRAMEWORK

The framework used to carry out the instrumentation tests on android is espresso. Espresso includes a large number of UI testing functionality and is easy to integrate and get started using as such it was our choice of frameworks for carrying out the UI testing portion of the application.

2.2 INSTRUMENTATION TESTS

As mentioned previously unit testing does not have access to a running android application and only to the java code, as such in JUnit testing mocking some classes are necessary to test them. However, this is unfeasible when attempting to test User interface components. For this instrumentation testing is essential as these tests have access to instrumentation information such as the context and the UI of the application allowing them to for example click buttons and navigate the app in an emulator that is spun up for each run of testing. While these tests run a lot slower (Around 2 mins) than unit tests they are essential to ensure that all components of the user interface have been tested. In total we created 19 instrumentation tests that targeted every aspect of every screen in the android application.

Figure 5

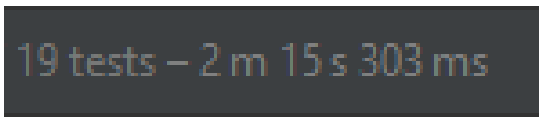


Figure 6 shows an example of an instrumentation test that inputs a test email and password into the relevant fields in the login and screen and checks that the UI has successfully changed to the home screen and that the user is logged in.

Figure 6

```
@Test
public void testingLaunchOfAppAfterLogin(){

    Instrumentation.ActivityMonitor monitor = getInstrumentation().addMonitor(HomeActivity.class.getName(),  result: null,  block: false);


    try {
        // Type email and password
        loginEmail = activity.findViewById(R.id.resetPassEmail);
        loginPass = activity.findViewById(R.id.login_password);
        assertNotNull(loginEmail);
        assertNotNull(loginPass);

        Espresso.onView(withId(R.id.resetPassEmail)).perform(typeText( stringToBeTyped: "123@gmail.com"));//email
        Espresso.onView(withId(R.id.login_password)).perform(typeText( stringToBeTyped: "123456"));//password
        Espresso.closeSoftKeyboard();
        onView(withId(R.id.loginBtn)).perform(click());

        Activity homeActivity = getInstrumentation().waitForMonitorWithTimeout(monitor,  timeOut: 5000);

        assertNotNull(homeActivity);
        homeActivity.finish();
    } catch (NoMatchingViewException e) {
        //NoMatchingViewExcpetion
    }
```

3. USER TESTING

Once ethical approval was obtained, we began user acceptance testing. Initially we had planned for two user testing phases both questionnaires will be linked below. The first phase was to have interview style meetings with consenting users to allow us to observe their use of the application and based on their feedback and our observation tweak the app accordingly, however due to the unforeseen circumstances we were unable to complete this phase of our user testing plan. We felt this interview style testing was necessary because our application, whether it is on the desktop platform or the android platform, requires modern powerful hardware due to the large processing power required to run the pose estimation model and algorithm. The second phase however we were able to complete successfully and this was sending the application to our peers and family and the relevant consent forms and plain language statements and asking them to use the application and fill out a questionnaire.

3.1 USER TESTING RESULTS

The first initial feedback we received was generally very positive with specific mention of the design of the application user interface. Users were happy with the design of the application, they found it easy and intuitive to navigate through and understand. They also appreciated the vibrant colour scheme that we chose.

Figure 7

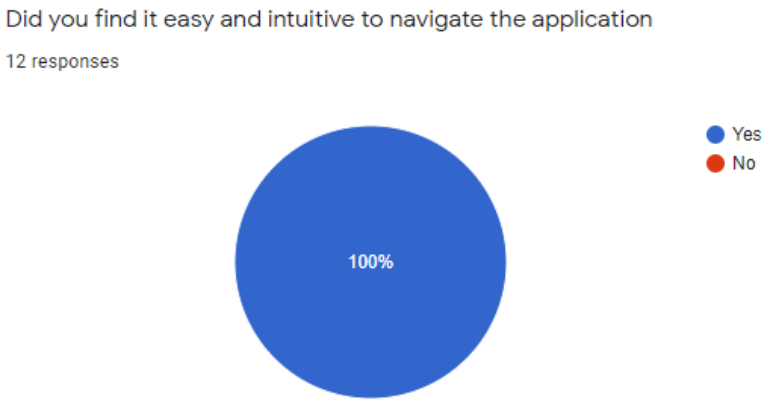


Figure 8

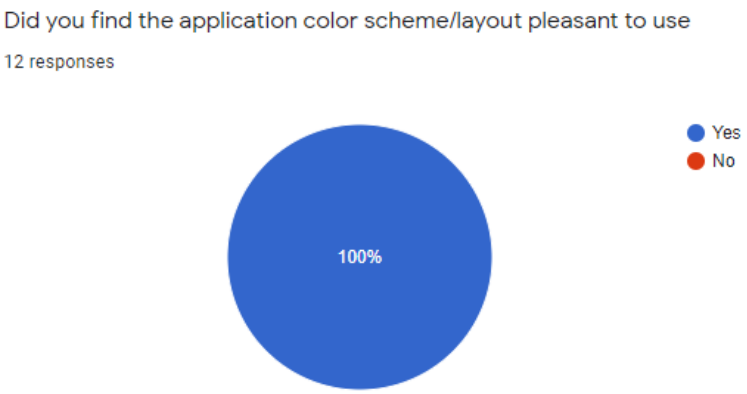


Figure 9

Any additional comments in relation to the user interface of the application

10 responses

It is really pleasant and intuitive to use
Easy to register, log in and navigate the app.
Clean design
No
Very pleasant colour scheme and design
Fast registration system
No, I was satisfied with the overall experience on the app
Using front facing camera would be nice
The text of the feedback was falling off the screen

The next portion of the questionnaire focused on the technique analysis component of the application. We first asked the users the device they were using and the size of the room they were in as we wanted to see how the application performed in different environments. We also asked the users what exercise they chose to perform.

Figure 10

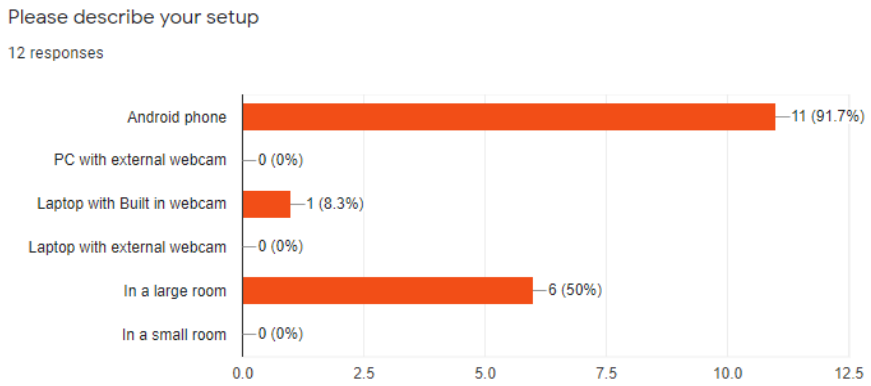
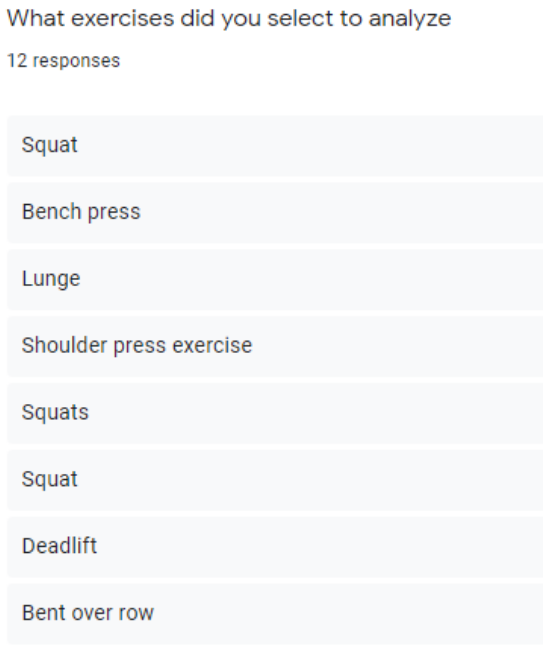


Figure 11



From all the responses that we received, 75% of users felt that the application successfully provided feedback on their form and that the application detected the users body and limbs correctly 91% of the time.

Figure 12

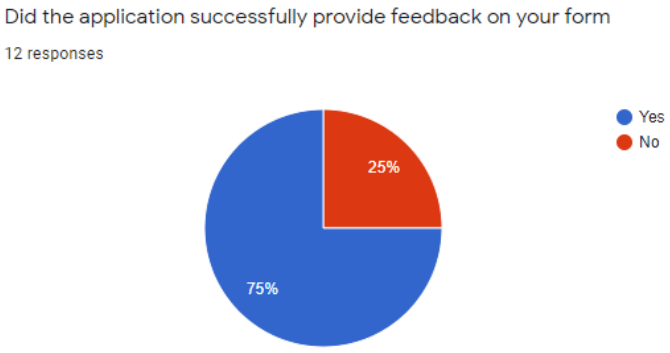
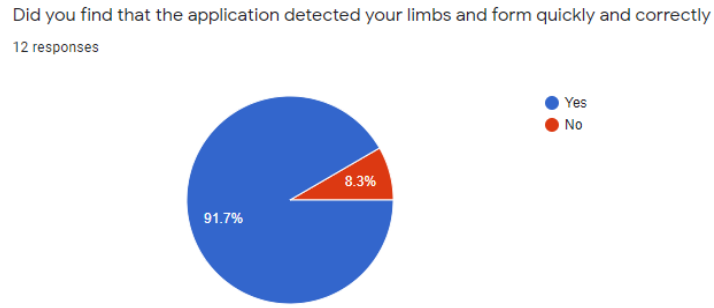


Figure 13



Some users reported issues which helped us to discover errors and problems in the application such as;

- Crashing when the user pressed too many buttons
    - To resolve this, we revisited our instrumentation tests to try and identify the cause of this error,
- Disappearing timer button from the exercise screen.
    - We addressed this issue by testing the application on multiple devices and screen sizes.
- Looping/overlapping audio feedback.
    - We resolved this issue by performing Ad-hoc tests to try and attempt to recreate the issue, once found we fixed the bug and made a note to always check the audio feedback.
- Feedback was not very useful and didn't help them improve their technique.
    - This led us to refine our feedback system and provide more detailed response on each check that passes or fails

We found user testing to be the most helpful in finding user interface bugs and identifying missing features from our application. This helped improve the overall quality of the app and improved the user experience.

Link to the questionnaire and interview forms that were sent to the users.

- https://forms.gle/vxeR79GnwFC1aKo36
- https://forms.gle/95L1D2Wzz2EBeNvC8

4. CONTINUOUS INTEGRATION

We implemented GitLab's Continuous integration (CI) platform and utilized it through the project's life cycle. We implemented a CI pipeline that included 3 stages.

- Pre - The 'pre' stage setup and initialize the necessary tools and environment required to run the rest of the pipeline
- Build - This stage was responsible for booting and running the android gradle [4] to ensure that it was compiling and passing successfully.
- Test - This was the main stage and ran all the unit tests mentioned previously.

The CI pipeline was triggered every time a commit was pushed to the master branch. This was essential for our application as the constant tweaking of the technique analysis heuristic values could mean that the application's core functionality becomes defective and does not function as intended. In the event of a failed pipeline run we could easily identify the issue due to the separate stages and then investigate the logs to find the specific test that failed or any gradle build issues. The pipeline configuration is located in the root directory of the repo in a file called '.gitlab-ci'. An example pipeline build can be seen in figure 14 and statistics for the lifetime of the pipeline can be seen in figure 15.

Figure 14

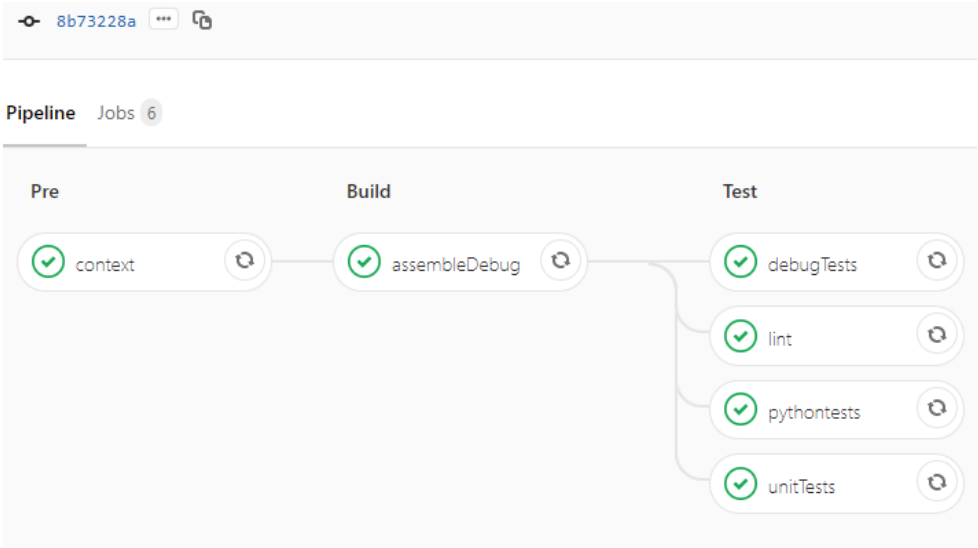
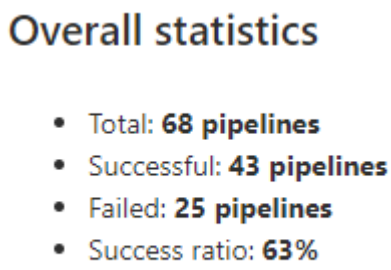


Figure 15



5. VALIDATION TESTING

We also carried out extensive validation testing following the advice from our supervisor. In our case validation testing consisted of myself and my partner performing what we deemed to be correct technique and incorrect technique and validating that the application produced the expected result. For each of the exercises in our application the technique analysis checks are based on joint positions in the x,y coordinate plane. We calculated technique correctness based on distances from one joint to another. Ensuring we used the correct values for the distances and distance boundaries required a lot of manual validation testing. The validation step process was as follows:

1. We performed an exercise with the values that we initially estimated would be correct.
2. We then performed exercises that we considered to be correct technique and tested these exercises against the applications defined heuristcs.
3. If the model was detecting the form to be incorrect, we would assume the values are incorrect and we would adjust the values accordingly.
4. We would also complete this step for videos labelled “bad” technique and if the model was determining them as completed with correct technique, we would assume the model is incorrect and we would adjust the values accordingly.

Below is an example for one round of validation testing performed for one technique check which attempts to determine if the user's grip is too wide while performing a deadlift.

Figure 16

```
def armsGripTooWide(self,human,image,pos):
    self.leftBoundary1 = pos.xLeftShoulder + 50
    self.rightBounadry1 = pos.xRightShoulder -50
    if pos.xLeftWrist > self.leftBoundary1 and pos.xRightWrist < self.rightBounadry1:
        return True
    return False
```

Figure 17

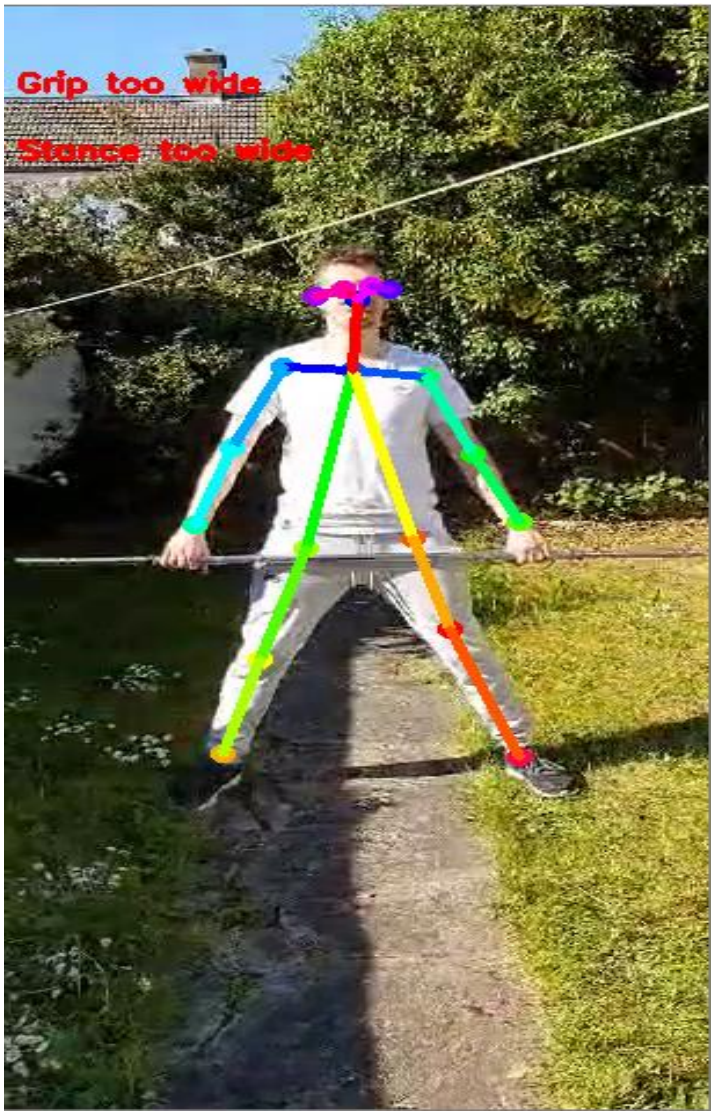
Here we see that although the grip is clearly too wide for the deadlift, the values were causing the model to produce a “Grip good” response. This meant that we needed to tune the values repeatedly until we found a distance would fail for the grip wide check here.

Figure 18

```
def armsGripTooWide(self,human,image,pos):
    self.leftBoundary1 = pos.xLeftShoulder + 20
    self.rightBounadry1 = pos.xRightShoulder -20
    if pos.xLeftWrist > self.leftBoundary1 and pos.xRightWrist < self.rightBounadry1:
        return True
    return False
```

.

Figure 19



Now the model is producing the correct result for that position, leaving us with a more accurate detection for wide deadlift grips. We repeated these steps for each check in each exercise.

6. ENVIRONMENT TESTING

We tested our application on multiple devices and screen sizes using the android emulator option. This is to ensure that our software can run well on a wide variety of android phone and that the interface looks and performs consistently across these devices. The devices we tested the application on are below, this is in addition to our personal android devices and the android devices of the users from the user testing that we carried out.

Figure 20

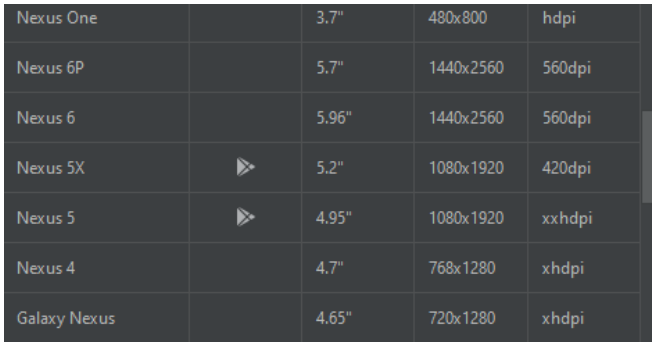
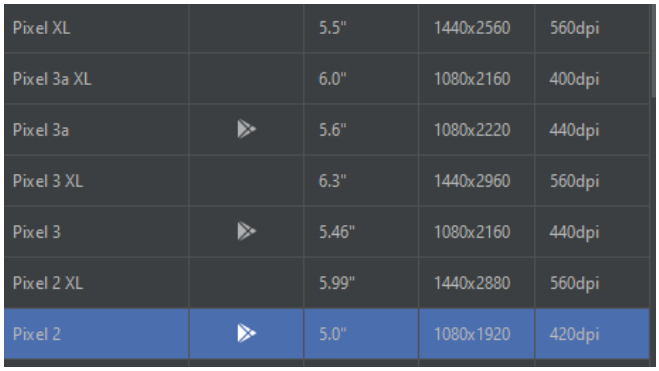


Figure 21



7. HEURISTIC ANALYSIS

In order to carry out the heuristic analysis we will be using Shneiderman's Eight Golden rules [1] heuristics as guidelines.

Strive for consistency

We tried to keep sequences of actions and the language used in feedback consistent for each individual exercise.

Offer informative feedback

The application offers informative feedback when the users login with incorrect details or create invalid accounts. It also provides exercise detailed and informative feedback on users exercise technique which is the purpose of our application.

Enable frequent users to use shortcuts

Since the sequence from start to finish in our apps use cases are relatively small, we did not implement shortcuts, so this rule was ignored.

Design dialog to yield closure

The design has a simple completion sequence form start to finish.

1. Go to exercise
2. Perform exercise
3. End at the feedback screen for sequence closure.

Offer simple error handling

The system is designed so that the user has limited power to break the application to ensure that a lack of understanding will not cause the application to crash. This was done through simple catch and accepts statements through the code.

Permit easy reversal of actions

The user can easily navigate backwards without any issues, they can cancel the exercise at any point.

Support internal locus of control

The users have full control over what they would like to do in the applications and what they want to allow the software to do. They have to manually enable their camera permissions which they can disable at any point.

Reduce short-term memory load

The design is kept simple, with a simple navigation drawer and small tutorial at the beginning to ensure users don't need to retain too much information when they go to the next page.

8. APPENDICES

1. https://www.interaction-design.org/literature/article/shneiderman-s-eight-golden-rules-will-help-you-design-better-interfaces
2. https://junit.org/junit4/
3. https://developer.android.com/training/testing/espresso
4. https://developer.android.com/studio/releases/gradle-plugin