UNIVERSITY OF MICHIGAN

ThirdEye Project Information

Global Challenges

Version 1.0

Table of Contents

Revision History

Version	**Changes**
1.0	Initial Version

1. Introduction

ThirdEye is an Android app that allows the user to interact with the flask-video-streaming server. Its main goals are to provide low latency streaming of the video, as well as providing a convenient interface for using the server's API. This document describes how the app works on a device as well as how it is implemented. Should additional features need to be added, there are examples at the end that outline what would need to be done.

2. App Overview

When ThirdEye is launched, it opens with the the Device Control screen open (Figure 1.). This gives the user control over connecting to the wifi network the server is on as well as rebooting or powering off the server. For both of the device control options, a dialog box will ask for confirmation of the action before sending the command to the server.

To change the currently displayed screen, the user needs to use the Navigation Drawer, which can be opened by pressing the hamburger menu in the top action bar or by sliding their finger across the screen from the left edge to the right. The currently active screen is highlighted in yellow (Figure 2.), and once the user picks something from the list, the Navigation Drawer closes and the requested screen will appear.

Selecting the Stream item in the Navigation Drawer will display the live stream from the server (Figure 3.). Depending on what video settings the user has selected, there may also be various buttons and sliders for controlling different camera options as well as taking pictures and recording video. The buttons should be self explanatory, with some like Snapshot, needing just a single press to perform the action, others like Autofocus being a toggle between two settings, and Record which tells the server to start and stop recording. The slider on the left side of the screen controls the focal depth of the camera (if that is supported) and the slider on the right controls the brightness of the light on the server (if one is connected). For all of

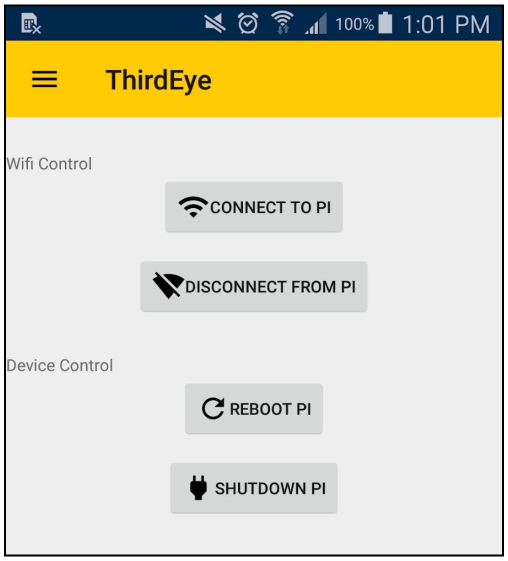


Figure 1

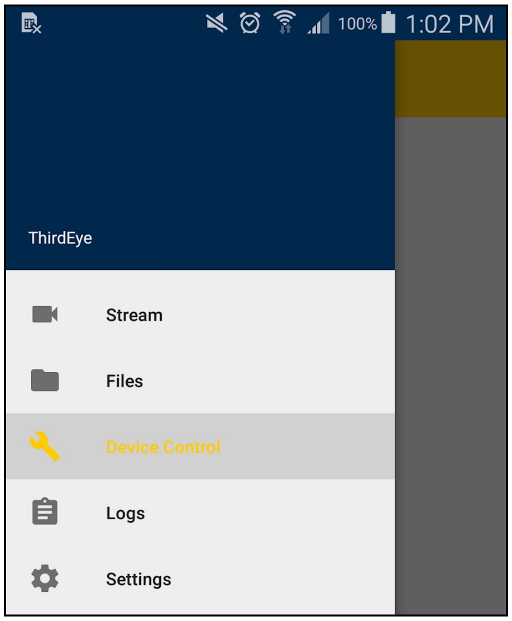


Figure 2

these buttons and sliders, messages will be given to the user at the bottom of the screen giving the result of the desired action. On the left side of the action bar there are other buttons, these may all appear or may be put into a menu accessible by pressing the three dots symbol. The five buttons are; Edit Filename (the pencil), Refresh (the refresh symbol), Zoom Out (magnifying glass with minus symbol), Color Picker (painters pallet), and Extra Settings (gear). Edit Filename allows the user to change the default filename for saving pictures and video without having to go to the Settings screen. Refresh reloads the video stream. Zoom Out will zoom out the video stream all the way, should the stream be zoomed in or panned at all. Color Picker allows the user to change the color of the light attached to the server. Extra Settings allows the user to select which sets of controls they would like on the screen, these can also be controlled by changing the options on the Settings screen.

Selecting the Files item in the Navigation Drawer will display all the pictures and videos saved on the server (Figure 4.). The user can select any of the files listed and the selected file will be downloaded and displayed on the user's device. If the user presses and holds, a dialog box will open prompting the user if they wish to delete the file on the server. Should the user wish to do so, the file will be deleted on the server, but it WILL NOT be deleted on their device. In the action bar there is a button that allows for the user to refresh the file listing.

Selecting the Logs item in the Navigation Bar will display both the server's internal logs and the server's boot logs (Figure 5.). There are two buttons in the action bar, which may be in the extra menu (triple dots) if there isn't enough space. The first lets the user refresh the logs, and the second allows the user to email a copy of the logs to someone. Note this screen should only be used for development and testing purposes, and should be removed before public release.

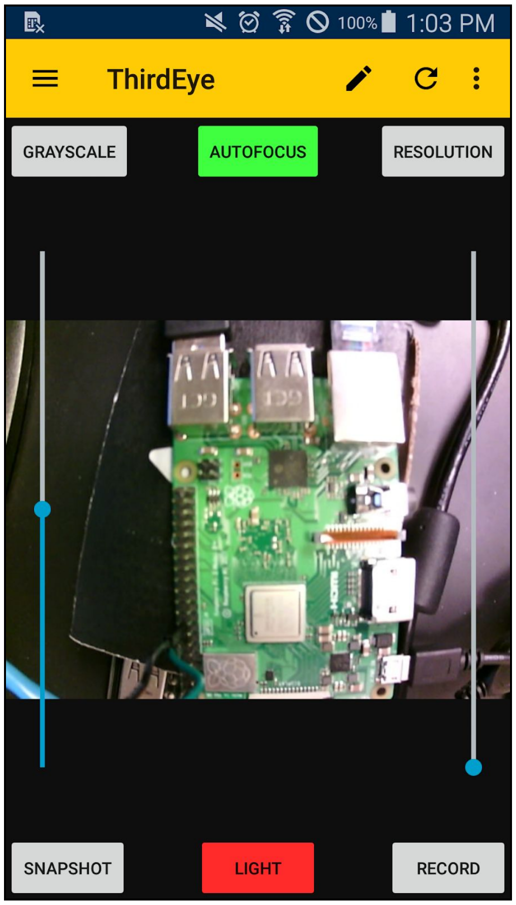
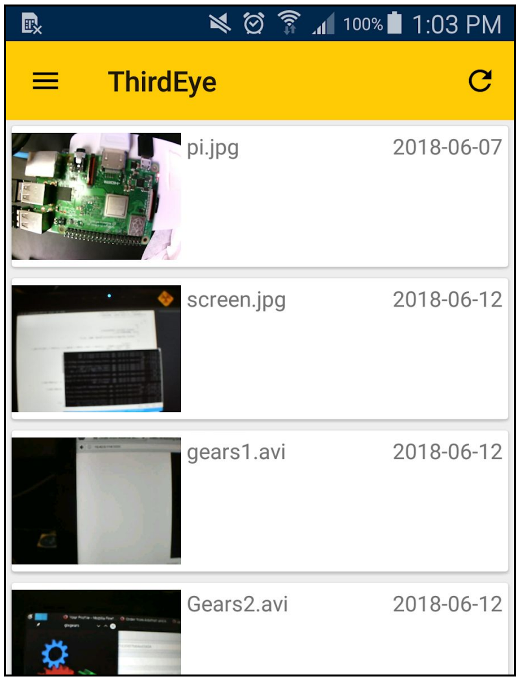


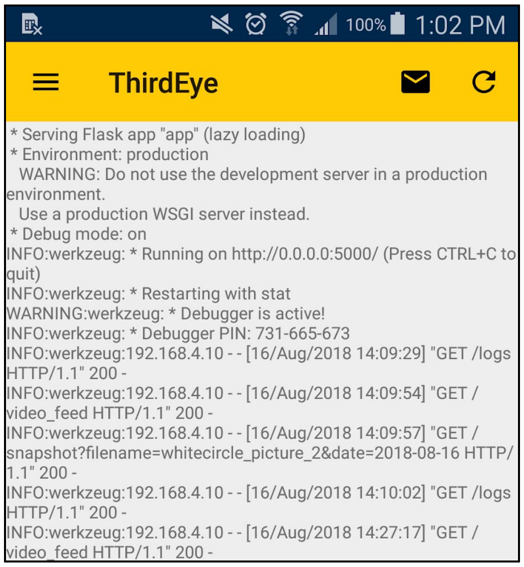
Figure 3



pi.jpg	2018-06-07
screen.jpg	2018-06-12
gears1.avi	2018-06-12
Gears2.avi	2018-06-12

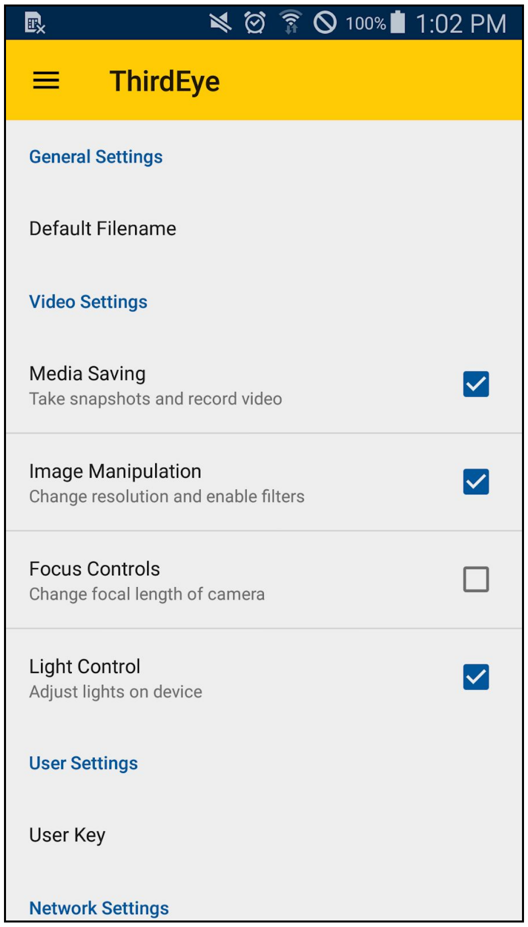
Figure 4

The final item in the Navigation Drawer is the Settings screen (Figure 6.). This allows the user to change various settings, such as what the default filename will be and what controls will be displayed when viewing the video stream. There are also a field for changing the user's userkey, which is what the app sends to the server for various things that should be done only by approved people. The last two fields allow for changing the name and password to the network the server is on.



```
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production
environment.
   Use a production WSGI server instead.
 * Debug mode: on
INFO:werkzeug: * Running on http://0.0.0.0:5000/ (Press CTRL+C to
quit)
INFO:werkzeug: * Restarting with stat
WARNING:werkzeug: * Debugger is active!
INFO:werkzeug: * Debugger PIN: 731-665-673
INFO:werkzeug:192.168.4.10 - - [16/Aug/2018 14:09:29] "GET /logs
HTTP/1.1" 200 -
INFO:werkzeug:192.168.4.10 - - [16/Aug/2018 14:09:54] "GET /
video_feed HTTP/1.1" 200 -
INFO:werkzeug:192.168.4.10 - - [16/Aug/2018 14:09:57] "GET /
snapshot?filename=whitecircle_picture_2&date=2018-08-16 HTTP/
1.1" 200 -
INFO:werkzeug:192.168.4.10 - - [16/Aug/2018 14:10:02] "GET /logs
HTTP/1.1" 200 -
INFO:werkzeug:192.168.4.10 - - [16/Aug/2018 14:27:17] "GET /
video_feed HTTP/1.1" 200 -
```

Figure 5



General Settings

Default Filename

Video Settings

Media Saving ✓
Take snapshots and record video

Image Manipulation ✓
Change resolution and enable filters

Focus Controls ☐
Change focal length of camera

Light Control ✓
Adjust lights on device

User Settings

User Key

Network Settings

Figure 6

3. App Breakdown

From a programming standpoint, there is only one activity that is launched with the app. This is aptly named *MainActivity*, and it holds all of the fragments[1] that will provide the other screens needed for the user action. *MainActivity* also adds the NavigationDrawer to the app, and listens for any events that happen to it to swap out the active fragment for the one selected.

Files: MainActivity.java[2], activity_main.xml

Following are the descriptions of the fragments that *MainActivity* may load depending on what the user needs.

a. Device Control

Currently *DeviceControl* is the first fragment to get loaded when the app starts. It has a couple of buttons that control the wifi network the device is connected to, as well as some buttons to power-off and reboot the server. Since this is a fragment, any buttons defined in the corresponding layout xml file will not be able to call any functions in *DeviceControl*. When *DeviceControl*'s view is created, it gives all the buttons a reference to itself in initializeListeners() so that it can capture all the onClick() events the buttons make. For the shutdown and reboot server actions, *DeviceControl* first prompts the user with a dialog, so when it makes the appropriate dialog, it gives them a reference to itself so it can get the result of the dialog. The dialogs will call onActivityResult(), so *DeviceControl* has to check to see what dialog called it before doing any actions.

Files: DeviceControlFragment.java, device_control_fragment.xml

b. View Stream

When the user wishes to view the video stream, the *DisplayStreamFragment* is needed. This uses a WebView to display the video stream, and provides different buttons and sliders for sending commands to the server. This fragment adds things to the options menu, so when those are selected onOptionsItemSelected() is called where the id of the button pressed is checked. Based on the id of the menu item selected, the fragment will either perform an action on the WebView or open a dialog for more input. The dialogs (except for the color picker) will call

[1] More information about Fragments can be found in the Appendix
[2] For more information about all the classes used in this project, see the the JavaDocs in the documentation folder of the project

onActivityResult(), so *DisplayStreamFragment* has to check to see what dialog called it before doing any actions. There are also buttons and sliders that may be displayed on the screen, based on what the user set in the settings, so any time the settings may have changed updateButtons() is called to show or hide them as appropriate.
Since this is a fragment, any buttons defined in the corresponding layout xml file will not be able to call any functions in *DisplayStreamFragment*. When *DisplayStreamFragment*'s view is created, it gives all the buttons a reference to itself in initalizeListeners() so that it can capture all the onClick() events the buttons make as well as any changes to the sliders. Based on the id of the button, onClick() will call the appropriate function for that button, such as sending a message to take a snapshot or toggle autofocus. The sliders have special listeners that define what will happen when the user starts and stops moving the slider. For both the buttons and sliders, a snackbar message is shown to the user based on the result of the desired action.

Files: DisplayStreamFragment.java, display_stream_fragment.xml

c. View Files

For viewing the files on the server *FileViewerFragment* does a lot of different things. For starters, the layout file defines a couple of different layouts inside the main FrameLayout. This is so that *FileViewerFragment* is able to change what message is displayed depending on what is going on. When files are being fetched from the server, the LinearLayout is displayed with a ProgressBar; when no files are on the server, the FrameLayout is displayed with a message; finally when files are on the server, the RecyclerView is displayed and populated with information about the files. This fragment adds things to the options menu, so when those are selected onOptionsItemSelected() is called where the id of the button pressed is checked, in this case it is just a refresh button to fetch a new listing of files from the server.
When there are files that need to be listed, *FileViewerFragment* has to give the RecycleView an ArrayList of FileItems. Each FileItem has a thumbnail, filename and date the file was saved. FileItem also defines the look in its own layout xml file.
FileViewerFragment has two functions for dealing with click events; onItemClick(), which downloads and opens the file from the server with the appropriate app, and onItemLongClick(), which will create a dialog and pass it a reference to *FileViewerFragment*. The dialog will call onActivityResult(), which will delete the file from the server if it is the right dialog.

Files: FileViewerFragment.java, file_viewer_fragment.xml

d. View Logs

THIS FRAGMENT SHOULD ONLY BE USED FOR DEBUGGING/DEVELOPMENT AND SHOULD BE REMOVED BEFORE PUBLIC RELEASE.

LogViewerFragment fetches information from the server and populates the appropriate TextView. This fragment adds things to the options menu, so when those are selected onOptionsItemSelected() is called where the id of the button pressed is checked. When the email option is selected, a dialog is created *LogViewerFragment* gives a reference to itself so it can get the result of the dialog. The dialogs will call onActivityResult() which then checks to see what dialog called it before doing any actions, which in this case is launching a email application.

Files: LogViewerFragment.java, log_viewer_fragment.xml

e. Settings

SettingsFragment simply creates a PreferenceResource from the settings xml file to be displayed.

Files: SettingsFragment.java, perf_general.xml

4. Example Additions

a. Adding a Button

Since ThirdEye makes use of fragments for all of its screens, adding a button is more involved than it would be if it didn't use fragments. There are two ways of adding a button to a fragment, you can either add it in the layout xml file for that fragment or you can add it programatically in either the fragment's `onCreateView()` method or any time after `onCreateView()` has been called. In this example we are going to add a button to the layout xml file for a fragment, since this allows us to use the design view to see where the button will be without having to build and run the application to check where it will end up. In all of the fragments, the steps will be the same as outlined below.

Here we are going to add a button to the *DeviceControlFragment* that will show us a message saying we pressed it. Open the device_control_fragment.xml file in the layout folder and switch to the Text view if it isn't already. We are going to want to place our button in the LinearLayout section so that it appears under all the other buttons and labels on the screen. Create a few lines above the `</LinearLayout>` tag and add in the following.

```
<Button
        android:id="@+id/press_me"
        android:layout_gravity="center_horizontal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Press Me" />
```

Let's quickly go over what all is happening here. First we create a Button tag, then we give it an id of press_me. You can give it any name you want, just change the part after `@+id/` since that signals we want this to be seen as an id. Then we specify the layout_gravity to center_horizontal, this makes the button center justified. There are a bunch of other options that you can use or combine to put things how you want, such as start or end which puts things on the left or right side. Then we have the layout_width and layout_height which are set to wrap_content, this makes the button just big enough to fit the text and/or icons we set. You can set this to a specific number of pixels such as 10dp or set it to match_parent, which will make it as tall or wide as the parent layout (in this case it is the LinearLayout, which is as wide as the screen). Setting the text should be self explanatory, but if you wanted to add an icon you would put
`android:drawableStart="@drawable/name_of_icon_file"`
before the `/>` closing tag. You can switch back to the Design view to make sure that the button you just added looks like what you expect.

Now that we have a button in the layout file, it is time to add things to the DeviceControlFragment.java file to make it do anything. There are four places we need to add things so the button does anything when the user presses it. The first place is at the top of everything in the class definition, we need to make a new private variable with all the other buttons and things.

```
private Button press_me;
```

Second is initializing the button in `onCreateView()`. This gets called after `onCreate()`, but before the fragment gets displayed to the user. Anywhere before `initializeListeners(view)` and `return view` add:

```
press_me = (Button) view.findViewById(R.id.press_me);
```

The important part of this step is making sure that id we look up with `R.id.press_me` matches the id we set in the xml file. So if we gave it the id of `give_me_money`, we would find it with `R.id.give_me_money` otherwise we won't find anything and get an error. Next we add a listener to the button so that the button will call something when it gets pressed. In `initializeListeners()` we need to add the line:

```
press_me.setOnClickListener(this);
```

Now when the button is pressed, it will call `onClick()` inside of *DeviceControlFragment*. Inside of this function is a switch statement that figures out which button was pressed. Here is where we will add a couple of lines that will check for our new button being pressed and if it is we will display a snackbar message saying we were pressed. In the switch statement before
`default : break;`
line add the following:

```
case R.id.press_me:
        snack_message(view, “You pressed me!”);
        break;
```

Like before, we need the id to be the same as the button we added in the xml. Now you should be able to run the app, and when you press the button a snackbar message will pop up with the message “You pressed me!”

b. Making a HTTP Request to the Server

Anytime you need to get or send information to the server, you are going to need to make a HTTP request. Since this isn’t something that is built into the standard Android libraries, ThirdEye uses another library called Volley which handles all the required calls needed to make a request and deal with a response. Using Volley to make a request is as simple as knowing the URL you want to make a request to and filling in what you want to happen when there is a response and when there is an error, then just add the request to the queue of requests.

In this example, we are going to make a request to the server to set the date[3] which for this example is at `/set_date`. To start we need to build a string that will hold the URL we want to hit. The network the server runs on will direct all traffic pointed to stream.pi to the computer with the server, but you could put in an IP address here too, in either case we need the :5000 after either the IP or stream.pi to reach the server. The question mark signals the end of the API endpoint we want to hit and the start of what we are sending. Here we assume `date` has the date we want to set.

```
String url = “http://stream.pi:5000/set_date?date=” + date;
```

[3] The server currently doesn’t have a way to set the date, but it could be added. See Flask Video Streaming API for more information about what the server’s API is.

Once we have our string, we can build our request. The request has two parts, the response and the error. Response happens if we get a response code from 200 to 299, and error happens if we get any other code or nothing at all.

```
StringRequest stringRequest = new StringRequest(Request.Method.GET,url,
      new Response.Listener<String>() {
            @Override
            public void onResponse(String response) {}
      },
      new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {}
      }
      );
```

If you have anything you need to do on a response or an error, you would place the code inside the curly braces after `onResponse()` or `onErrorResponse()` as appropriate. Now all that is left is adding this request to the queue of requests. First we need to make a queue and then we can add to it. In most of the fragments in ThirdEye, there is already a variable that holds a queue for requests aptly named `queue`, but in this example we will assume there isn't a queue set up already.

```
RequestQueue queue = Volley.newRequestQueue(getContext());
queue.add(stringRequest);
```

With that our request will be sent when it gets to the top of the queue, but the app will keep doing things without having to wait for it. So if you need to update something on the screen, make that happen in either the `onResponse()` or `onErrorResponse()` of the request.

5. Building and Installing

With the project loaded up in Android Studio, building the app is as easy as clicking the run button[4] and selecting the device you wish to run it on[5]. It is recommended that you use a real device to run any builds on, as the app will need to connect to the Flask Video Streaming server on a specific network for many of its functions. If you need to make a Signed APK, check out Keystore Notes which has more details.

Once you have an APK that you want to install onto a device, simply copy the APK over to the device then find the file on it in a File Browser. If Unknown Sources is not enabled, then the device will ask if you wish to install the APK, which will temporarily enable

[4] See the Appendix for more in depth explanation of the build process.
[5] See the Appendix for more information about prepping a device for being used for a build target.

Unknown Sources to install the app. Otherwise the app will install without any prompts and you will be able to find the app in the app drawer.

6. Appendix

More information about Fragments and how to make them can be found here.

For more information about all the different java classes used in this project, see the JavaDocs in the Documentation folder of the project.

For more information about the server API, see Flask Video Streaming API.

More information about building and running an Android project can be found here.

More information about enabling a device to be used as a build target can be found here.