Hand-Gantry Project Progress Report, Spring 2022

Siddharth Diwan
Brown University
#2144, 69 Brown St., Providence, RI 02912
siddharth_diwan@brown.edu

Abstract

I present a summary of my research contributions to the Hand-Gantry project in the Interactive 3D Vision and Learning Lab at Brown University. These include conducting a thorough analysis of the SQL line of spy cameras, making slight modifications to Kara-Recorder - an application that displays simultaneous views from connected USB cameras - and attempting different approaches to developing Kara-Droid - a mobile application that can display simultaneously from USB cameras connected to an android device.

1. Introduction

Over the past few years, datasets such as the Kitchen Dataset [3] have allowed individuals to, from limited camera views, categorize not only hand movements but also hand interactions. But in all these datasets, the camera has always been egocentric, i.e., attached to the head of the user. There is a lot to learn from performing similar analyses using oblique views of the hand from the forearm or wrist.

Data must be collected sufficiently for such oblique view analysis to be possible. Creating a wearable setup or Hand-Gantry that can record oblique views will help with this. To that end, I have been leading efforts to develop the appropriate software; additionally, I have contributed to some significant hardware choices. I detail all my contributions this semester in this report.

2. USB Camera Selection

The most crucial hardware component of the Hand-Gantry setup is the cameras to use when recording the oblique views of the hands. To help determine which USB camera may be optimal for our task, I researched three camera models in the Quelima SQL Line.

I omit camera details such as battery life, charging time, storage, and loop recording for this report since, in the Hand-Gantry, all USB cameras are connected to a phone via a USB hub (3 cameras per phone, see Fig. 1). Therefore, all cameras will be charging, and we will use the phone's local storage.

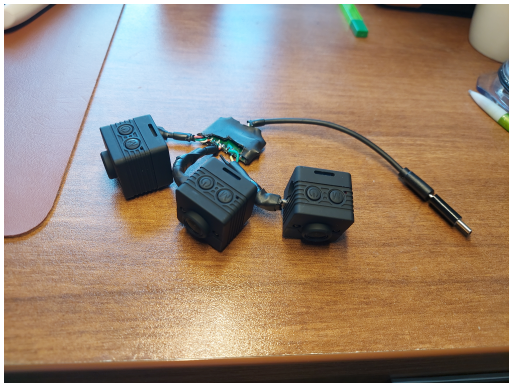

Figure 1. Three SQ12 cameras connected to a USB Hub.

2.1. SQ12

The SQ12 is a spy camera that, under a wired connection, can record and upload recordings to a storage space [22]. It can record videos in 720 or 1080p, with a 155-degree wide-angle lens at 30 FPS. It is a cube with an inch of side length. It is also very light, weighing just under 20 grams [2].

The SQ12's size and weight make it a good candidate for our purposes, but a setback is that the cameras do not record audio. In post-processing, aligning camera audios can sync the videos, so not having audio may pose an issue in future semesters. Ideally, we will not face any hurdles if we sync all cameras by creating threads in succession, preempting impartially, and closing threads simultaneously.

2.2. SQ13 and SQ23

The SQ13 is a similar model to the SQ12. It shares most camera features, such as the 155-degree wide-angle lens, and has the same FPS cap and video resolutions.

Unlike the SQ12, though, the SQ13 supports WiFi connection via the SportsDV app [23]. Unfortunately, SportsDV only allows for connection to one SQ13 camera at

a time, which makes it unideal for the Hand-Gantry project. Furthermore, the SQ13 is slightly larger than the SQ13 - it is a cube with a 1.1-inch side length. It is also significantly heavier at 140 grams [31].

Despite being able to record audio, the SQ13 is likely unsuitable: it may make the gantry bulky and was designed for WiFi connection, whereas the Hand-Gantry uses wired connections.

A sister model of the SQ13 is the SQ23. Regrettably, the only major improvements it sees are in WiFi range (extended to 10 meters), which is irrelevant to the Hand-Gantry's wired setup [24]. Thus, for similar reasons as the SQ13, this model is unlikely to be suitable.

3. Kara-Recorder

Srinath created Kara-Recorder in 2019 [29] to display simultaneous streams from all webcams connected to a laptop, desktop, or PC. It is coded in C++ and uses OpenCV 3.2.0.

My goal was to get Kara-Recorder working on my laptop and then use the code in Kara-Droid. The updated Kara-Recorder is here: `https://github.com/sidwan02/Project-Kara.git`.

3.1. Running Native Code on Windows

Unlike Python, C/C++ programs cannot be directly run. They are compiled into a library file, such as a static library file (.a) or a shared library file (.so), and then built into an executable that the user can run.

CMake. To generate the library file, I used CMake. CMake works by specifying paths to all library dependencies (which in this case is only OpenCV) and all source files (which in this case is the Kara-Recorder and other `hpp` and `cpp` files).

CMake GUI and CMakeLists. Specifically, I recommend the use of CMake GUI [1] since the interface makes adding dependency paths intuitive, without needing to edit the `CMakeLists.txt` file. The `CMakeLists.txt` file records all source files, libraries, and dependencies and is run by the compiler to compile the native source files into library files.

Visual Studio. After creating the libraries, it is necessary to build an executable (.exe on Windows) from the libraries. For this, I used Visual Studio 2015 [19]. By specifying Visual Studio 2015 as the project builder in the CMake GUI, I could directly open up the Visual Studio application and build the code in the VS UI. If Release mode is selected, the Release folder will contain the executable (the Release folder is within the build directory specified in CMake GUI's build directory destination).

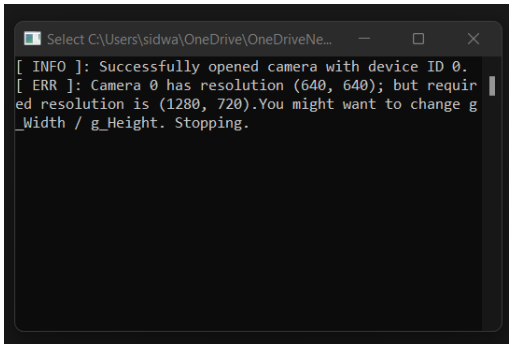

Figure 2. Kara-Recorder detecting inconsistent resolutions.

3.2. Camera Resolution Inconsistency

When attempting to run the Kara-Recorder executable, I would consistently get an error with a matrix multiplication operation. This matrix multiplication was performed on the streamed output to produce the frames to display to the user.

I found out that varying image frame dimensions were causing the error. The camera device with id 0 (which is always the laptop's webcam if enabled or present) had a resolution of (640 x 640), whereas the SQ12 cameras were (1280 x 720). Armed with the knowledge that all camera resolutions must be the same, I decided to skip Camera ID 0. However, this fix would not scale, especially if the desired camera resolution were to change. Thus, I added a script to detect any frame inconsistency, i.e., camera resolution variance among all connected cameras, and print out a reasonable error message (See Fig. 2).

3.3. Main Thread Termination on Error

When I opened the Kara-Recorder release executable directly (via double-clicking), if the program encountered an error, it would close before I could see the error message. The reason for this was that any `g_CamThreads` (camera thread) encountering an error would kill the process's thread.

To fix this, I modified the `init` function (which starts all of the cameras) to await an error code from any camera thread. Then, if a camera thread returns with an error status, the program will hang until the user hits Ctrl+C or closes the application. Thus, the error message remains displayed after the program has ended.

An alternate approach is to run the executable from within a terminal such as GitBash. Then, even if the process thread terminates, the error message has been printed.

3.4. Upgrade to OpenCV 4.5.5

I did this only after realizing that integrating OpenCV Android 3.2.0 as a library dependency within Android Studio would not be possible (See Sec. 4.5 for more details).

First, I made changes to the imports and enum declarations. For instance, I replaced all declarations of the method `CV_FOURCC` with `cv::VideoWriter::fourcc` to reflect changes in the library hierarchy between OpenCV 3 and 4.

I determined the stale library import paths by repeatedly building Kara-Recorder for release in Visual Studio and tracing build errors to the conflicted variables and functions. Once I identified the invalid imports, I used the official OpenCV documentation [21] to determine the version 4 library import paths.

However, I still could not run the executable due to a missing DLL. DLLs or Dynamic Link Libraries are similar to library dependencies that are explicitly saved in `CMakeLists.txt` but have already been compiled and are exclusive to Windows [14]. Specifically, I was missing `opencv_world455.dll`. I was able to resolve the issue by copying the DLL from within `...\OpenCV-4.5.5\build\x64\vc15\bin` to the `Release` folder of the Kara-Recorder executable (x64 represents the laptop's architecture, and vc15 reflects the version of the Visual Studio C/C++ Tools installed). I may need to have edited an additional variable in the CMake GUI dependency variables to avoid this, but I was not able to determine which one.

4. Kara-Droid

The purpose of Kara-Droid is to serve as an android application replicating the function of Kara-Recorder.

4.1. Integrating Native Code in an Android Project

A mobile application consists of a frontend and a backend. I wanted the backend of the application to be native so I could directly integrate Kara-Recorder. But, as discussed earlier in Sec. 3.1, calling a C/C++ function from a function in a different programming language (here, from the frontend code) does not work. An added layer of difficulty is that most mobile applications have a Java backend, and so the only way to interface C++ code with the frontend is to make it interface via the Java backend.

JNI. JNI, or Java Native Interface, is a programming interface that allows the bytecode that Android generates from compiling Java to interface with native code [8, 13].

JNI also provides special JNI bridging functions. To call a native function, I needed to define a Java function with the `native` keyword in the caller's Java class. Then, I included the package structure of the calling Java class in the native function's name. Now, the Java function with the `native` keyword can call the native function.

JNI Environment. When retrieving arguments and returning values within the native JNI function, I used the JNI Environment `JNIEnv` to cast the data types either to C++ data structures or to Java data types. Furthermore, the files that JNI functions exist in must be detected as source files by the JNI by specifying as such in the `CMakeLists.txt` of the application (similar to listing native source files in CMake GUI as discussed in Sec. 3.1).

build.gradle, NDK and SDK. Just like I needed to give CMake GUI the path to the `CMakeLists.txt`, I made JNI aware of the path to the application's `CMakeLists.txt` by modifying the `build.gradle` file of the application [26].

The `build.gradle` file serves three purposes. First, it helps the JNI detect the right NDK (Native Development Kit) to compile the native code. Second, it helps the compiler identify the correct SDK (Software Development Kit) to convert compiled native code to Java. Third, the `build.gradle` records the dependencies of the application or library. These could be libraries specified in the `CMakeLists.txt`, or even modules created along with the given application in the project workspace.

JNILibs. Finally, any native android project must have a JNILibs folder. It contains different ABI architecture types, such as `arm64-v8a`, `armabi-v7a`, and `x86`. Each of these Application Binary Interfaces is responsible for interfacing between the application's Java binary and native binary code, and one of them is chosen by the compiler depending on the device hardware.

4.2. Expo Framework

My first attempt at developing the android application was through Expo [12]. Expo is a framework that abstracts away components of an application's build files and allows users to focus on frontend app development (primarily through React).

My decision to choose Expo was twofold. First, I was comfortable developing the frontend with React since I am familiar with React's components and features like hooks and use states. Second, Expo gives users the functionality to test the app on their device rather than on an emulator by scanning a QR code on the Expo Go application [10]. Using the Tunnel option creates a Tunnel process from a localhost URL on the laptop to the web application [11] (I would realize later that updates to the Android Studio IDE would make its interface more suitable for testing, which I cover in Sec. 4.4).

To create a basic Expo application, I installed Node. Node is an event-driven JavaScript runtime and is responsible for compiling the JavaScript in the react frontend [20]. The Expo-CLI can be used to run Node commands implicitly [12].

Figure 3. Native integration in Expo without library dependencies.

4.3. Expo Ejection

Once I had a basic expo application working, I wanted to test native code integration. Unfortunately, when Expo creates a new project, it lays a workflow by default (unless otherwise specified during creation) which abstracts out native development components [9]. To uncover the native development components, I ejected the app.

But even after ejection, Expo does not give users access to the `CMakeLists.txt` or any default native sources folder (which is a `cpp` folder for our purposes). I knew I had to include libraries such as OpenCV into the `CMakeLists.txt` file, so needing to create that file from scratch, not to mention linking the libraries manually, would be very arduous. Thus, despite successfully running native code (albeit without library dependencies; see Fig. 3), I abandoned Expo.

4.4. Android Studio IDE

The biggest reason for switching to Android Studio is because it gives users the option to create a native project. With this, I had access to the `build.gradle` file of the application, a `settings.gradle` file to set the NDK and SDK paths for all applications and libraries, a `cpp` folder which contained the native sources, a pre-linked template `CMakeLists.txt` file, and a MainActivity Java class (akin to the Main Java class from which the app runs).

With a native project template and access to the new WiFi debugging feature (see Appendix A), this was already looking more promising than Expo. My next step was to integrate the OpenCV library.

Figure 4. An example of blob detection on my hand.

4.5. OpenCV Android

After downloading OpenCV Android 3.2.0 (OpenCV Android contains a JNILibs, whereas OpenCV does not), I successfully included it within the `CMakeLists.txt` and was able to see OpenCV's headers populate within the included libraries folder as expected. However, I had trouble adding OpenCV as a module dependency to my application to be used for imports in the Java classes (to clarify the distinction between library imports in Java versus library imports in native code, see Appendix B): Android Studio was unable to understand the library's `build.gradle` file. I then tried the same setup with OpenCV Android 4.5.5, and the issue did not occur (I am unsure which dependencies were missing in 3.2.0).

In fact, with OpenCV Android 4.5.5, I was able to get one of OpenCV's demo projects, blob-color-detection, to work on my device (See Fig. 4).

So, after successfully upgrading Kara-Recorder to use OpenCV 4.5.5 (see Sec. 3.4), I was able to include Kara-Recorder's native files as additional source files within the `CMakeLists.txt`. I then shifted my attention to developing the JNI bridge functions.

4.6. JNI Bridging and Android API

I first converted Kara-Recorder's main function into a JNI function. Whereas the code was compiling correctly, when attempting to run the application, `VideoCapture` was unable to find any cameras.

To verify whether this was a hardware issue or not, I wrote some Java code using the Android API (a library similar to OpenCV giving access to camera hardware such as sensors and cameras) [4] to detect connected cameras, which worked.

Further investigation led me to realize that OpenCV Android does not support `VideoCapture` [17], likely be-

cause OpenCV does not directly interface with the cameras. Instead, it uses different camera capture APIs depending on the device and operating system. For instance, to interface with USB devices on Windows, OpenCV uses the WinUSB library, which requires physically installed drivers on the device [18]. The requirement of installed drivers (such as DirectShow on Windows) could mean that OpenCV Android analogously needs specific installed drivers for `VideoCapture` to work, which may not be present on all android devices.

At this point, I could continue working with the Android API, but I wanted to be able to reuse as much of the Kara-Recorder's native code as possible (especially the thread creation and frame collage functions). Thus, I began exploring the Android NDK API [6]. This API provides the same functionality as Android API but for native code. By using this API, I hoped to circumvent using OpenCV's `VideoCapture` but continue using OpenCV's other arguments and functions as were already laid out in Kara-Recorder.

4.7. Android NDK API

To use Android NDK API with C++, I first modified the `CMakeLists.txt` file to include an additional library `camera-lib` (similar to how I had included OpenCV Android). Then, I was able to include header files from the camera NDK API and translated the Java code I wrote earlier with the Android API to C++.

By using data structures called `metadataObj`, I was able to successfully identify not only cameras on my device but also their properties. For instance, I could print the state of each camera's lens direction. However, I could not find any analogs for the frame width and height camera attributes within the NDK API documentation. I was hoping to use the frame width and height to be able to determine the resolution of each camera output as was done in the `init` function of Kara-Recorder.

Unfortunately, the lack of documentation or example projects using the Android NDK API prompted me to find another library. At this stage, I realized that I needed to port over Kara-Recorder to Java. Whereas initially convinced that I would need to use the Android API, I decided to look at one final library, LibUVC.

4.8. LibUVC

LibUVC is a promising library candidate since it is agnostic to the application's hardware. It interfaces with another library, LibUSB, which is an open-source library that allows users to communicate with USB devices from any user space [16, 30]. LibUVC code should be able to work unaltered on any device, where the only change in android devices is the requirement of an NDK, JNILibs, and `build.gradle`. Being agnostic to the hardware comes with many benefits. One of note is that, unlike OpenCV, LibUSB (and thus LibUVC) does not require drivers to be installed on the device to be able to interface with the cameras [18].

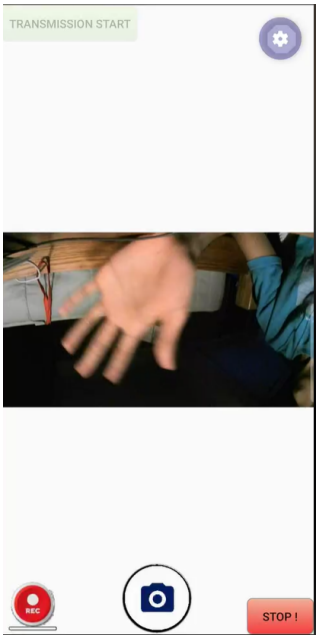

Figure 5. A snapshot of the Peter-St app stream.

I decided to recreate my Android Studio environment but include LibUVC dependencies instead of OpenCV ones. I did this by exploring existing LibUVC projects, hoping to build upon one of them.

4.9. Existing LibUVC Projects

Peter-St This project [25] integrates LibUVC and LibUSB quite cleverly by including both libraries as source files rather than libraries (within the includes folder in `cpp`). Then, the author refactors the imports within LibUVC to refer to paths within the sources `cpp` folder rather than the include folder.

Getting this project to build was largely straightforward: I edited the `settings.gradle` file to have the correct SDK and NDK paths and then ensured all imports to the NDK API reflected my local NDK path.

I was able to get the app to run on my phone and stream from a single USB camera (See Fig. 5). Unfortunately, the app is difficult to use, crashes frequently, and has many unintuitive parameters. I had the most difficulty adjusting the Packets per Request and Maximal Packet Size; the former is the number of packets sent from the connected camera to the phone in one cycle of transfer, and the latter is the maximum number of bytes each packet can contain.

I attempted to use the recommended parameter values (1 and 3072) but could only get frame rates of about

0.1 FPS. Moreover, I am unable to download video recordings. Further investigation into this repository is ongoing, and it may be worthwhile attempting to determine what the parameters mean in the context of the SQ12. Still, I have abandoned this repository since a large portion of the code focuses on logic for interfacing with one camera (the code makes no notion of camera threads either).

saki4510t This [28] is a more promising repository that handles LibUSB and LibUVC imports similarly to Peter-St. However, these imports exist within a separate repository, `libcommon` [27], which contains other useful Java functions that the author created to interface with the native thread functions of LibUVC.

The repository itself contains nine applications of LibUVC, all of which depend on the packages in `libcommon`. `usbCameraTest7`, is most important to get working since it deals with simultaneous streaming and displaying from 2 USB cameras.

Closer inspection of `libcommon` led me to see that the author had been maintaining compiled library versions of the `libcommon` packages as Android Archive (.arr) files [7]. These files contain all necessary source code, dependencies, `build.gradle` and other library files precompiled.

To access the `libcommon` package, I edited the saki4510t repository's project `build.gradle` file (this is separate from each of the applications' `build.gradles`, which deal with each of the tests) to include the path to one of the `libcommon` version Android Archive files. I did this by including a special repository dependency and fed it in the path as a raw GitHub link. This resolved all dependencies and allowed me to successfully run `usbCameraTest` and `usbCameraTest0` on my phone (these deal with simple camera recognition, see Fig. 6).

I am currently working on displaying actual USB camera footage, which is test `usbCameraTest2`, after which I plan to work on `usbCameraTest7`.

The fully integrated project is here: https://github.com/sidwan02/UVC-Camera.

5. Future Trajectory

5.1. USB Camera Selection

The SQ12 seems to be the most promising candidate amongst the Quelima SQ products. However, analysis of Pinhole Cameras may result in switches to the hardware design. Nevertheless, the choice of camera should not immediately impact the software that connects, streams and displays simultaneously from multiple cameras.

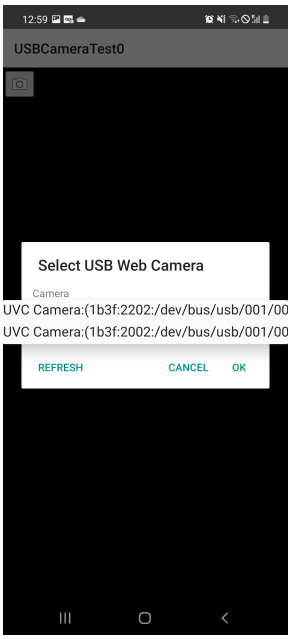

Figure 6. A snapshot of usbCameraTest0.

5.2. Kara-Droid

I am working with Jacob to get the saki4510t repository's tests to function. Currently, I am resolving a dependency issue with displaying a single connected USB camera, where the project seems to be unable to detect the `jpeg-turbo1500_static` module for JPEG compression and decompression in videos [15].

Next, I plan to modify `usbCameraTest7` to support an arbitrary number of displays depending on the number of connected cameras.

Acknowledgements I would like the thank the following individuals for helping me make important decisions during the semester: Srinath Sridhar, Jacob Frausto, Jing Qian and Aurthur Chen.

References

[1] Ruslan Baratov. CMake Installation. https://cgold.readthedocs.io/en/latest/first-step/installation.html. Last Commit: 2021-03-03. 2

[2] Anthony Barker. Quelima SQ12 Sports 1080p Mini DV Camera Review . https://www.carplaylife.com/review/quelima-sq12-sports-1080p-mini-dv-camera-review/. Published: 2018-06-15. 1

[3] Dima Damen, Hazel Doughty, Giovanni Maria Farinella, Sanja Fidler, Antonino Furnari, Evangelos Kazakos, Davide Moltisanti, Jonathan Munro, Toby Perrett, Will Price, and Michael Wray. Scaling egocentric vision: The epic-

kitchens dataset. In *European Conference on Computer Vision (ECCV)*, 2018. 1

[4] Android Developers. Android API Reference. https://developer.android.com/reference. Last Updated: 2022-02-17. 4

[5] Android Developers. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb. Last Updated: 2022-03-30. 7

[6] Android Developers. Android NDK API Reference. https://developer.android.com/ndk/reference. Last Updated: 2022-03-17. 5

[7] Android Developers. Create an Android library. https://developer.android.com/studio/projects/android-library. Last Updated: 2022-04-05. 6

[8] Android Developers. JNI tips. https://developer.android.com/training/articles/perf-jni. Last Updated: 2021-10-27. 3

[9] Expo. Ejecting to ExpoKit. https://docs.expo.dev/expokit/eject/. 4

[10] Expo. Expo Go. https://expo.dev/client. 3

[11] Expo. How Expo Works. https://docs.expo.dev/guides/how-expo-works/. 3

[12] Expo. Installation. https://docs.expo.dev/get-started/installation/. 3

[13] IBM. The Java Native Interface (JNI). https://www.ibm.com/docs/en/sdk-java-technology/7?topic=components-java-native-interface-jni. Last Updated: 2021-02-28. 3

[14] Han Liang, Anna-Li, Daniel Rugerio, Lu Chen, and Simonx Xu. What is a DLL. https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library. Published: 2022-04-12. 3

[15] libjpeg turbo. libjpeg-turbo. https://libjpeg-turbo.org/. Last Updated: 2022-05-03. 6

[16] libusb. libusb-1.0 API Reference. https://libusb.info/. 5

[17] Olivier Matrot. C++ cv::VideoCapture.open(0) always return false on Android. https://github.com/opencv/opencv/issues/11952. Issue ID: 11952. 4

[18] Olivier Matrot. Difference between LibUSB and WinUSB? https://www.microchip.com/forums/m437429.aspx. Published: 2009-07-28. 5

[19] Microsoft. Visual Studio Community 2015 with Update 3. https://my.visualstudio.com/Downloads?q=visual%20studio%202015&wt.mc_id=o~msft~vscom~older-downloads. Release Date: 2016-06-27. 2

[20] Node.js. Downloads. https://nodejs.org/en/download/. Version: 16.15.0 LTS. 3

[21] OpenCV. OpenCV modules. https://developer.android.com/training/articles/perf-jni. Version: 4.5.5-dev. 3

[22] Org-Info.Mobi. SQ12 Mini DV. https://org-info.mobi/manual/sq12-en.htm. 1

[23] Org-Info.Mobi. SQ13 Mini DV. https://org-info.mobi/manual/sq13.htm. 1

[24] Org-Info.Mobi. SQ23 Mini DV. https://org-info.mobi/manual/sq23.html. 2

[25] Peter-St. Android-UVC-Camera. https://github.com/Peter-St/Android-UVC-Camera. Last Commit: 2022-04-18. 5

[26] Marius Reimer. React Native and C++. https://reime005.medium.com/react-native-and-c-87a2311dc3d. Published: 2019-08-11. 3

[27] Saki. libcommon. https://github.com/saki4510t/libcommon. Last Commit: 2022-04-05. 6

[28] Saki. UVCCamera. https://github.com/saki4510t/UVCCamera. Last Commit: 2018-10-02. 6

[29] Srinath Sridhar. ProjectKara. https://github.com/brown-ivl/ProjectKara/tree/master/tools/KaraTools. Last Commit: 2022-04-19. 2

[30] Ken Tossell and Robert Xiao. libuvc. https://github.com/LibUVC/LibUVC. Last Commit: 2022-03-07. 5

[31] UNIKAMI. Mini Camera WiFi SQ13 Camera. https://www.amazon.com/Camera%EF%BC%8CMini-Detection-Upgrade-Camcorder-Recorde/dp/B07G6DRV5C. 2

A. WiFi Debugging

By updating to Android Studio version Bumblebee, I could debug over WiFi [5]. WiFi debugging works by creating an ADB or Android Debugging Bridge that uses a Tunneling process (similar to Expo) to communicate with an android application while logging errors and states in the Android Studio logcat UI. WiFi debugging also downloads the APK of the app tested on the phone, which lets the user run applications whenever they want (not just when the tunnel is open).

WiFi debugging is necessary for me (wired debugging, also supported by Android Studio, will not work) because I need to test the app while the phone uses its USB port to connect to the USB hub.

I faced some issues with WiFi debugging. Most notably, none of Brown, Brown Guest, or eduroam allows P2P communication. I got around this, however, by connecting my laptop and phone to a different phone's hotspot.

B. Java and Native Library Imports

When I added a module as a dependency in an application's `build.gradle`, this allowed only the Java code to access the module's packages. For instance, including OpenCV as a dependency of the main application only let me import OpenCV within the Java classes such as the MainActivity class.

However, to use the libraries of a module in the native code, I directly modified the `CMakeLists.txt` in the `cpp` folder. So, to access OpenCV's header files, I included the path to OpenCV as a shared library dependency. Then, by syncing the project's Gradle files, all folders included in the `CMakeLists.txt` file appeared in a special

`includes` folder within the `cpp` sources folder, giving full access to the contents of the library's headers.