Wireless Android Camera

By: Ryan Bentz, Andrew Capatina,

Ryan Bornhorst

ECE 544 Final Project

Spring 2019

Table of Contents

1.0 Project Description

The project attempted to create a wireless camera with accompanying Android application. The goal was to implement an embedded system capable of controlling an OV7670 CMOS camera where one could use the Android application to take a picture with the camera and view the picture on their smartphone. The original intent was to have the camera mounted on a motor that could rotate the camera, but unexpected difficulties with the camera control, image transmission, and image decoding in Android proved to be more than enough for this project. Figure 1 shows the final design of the project.

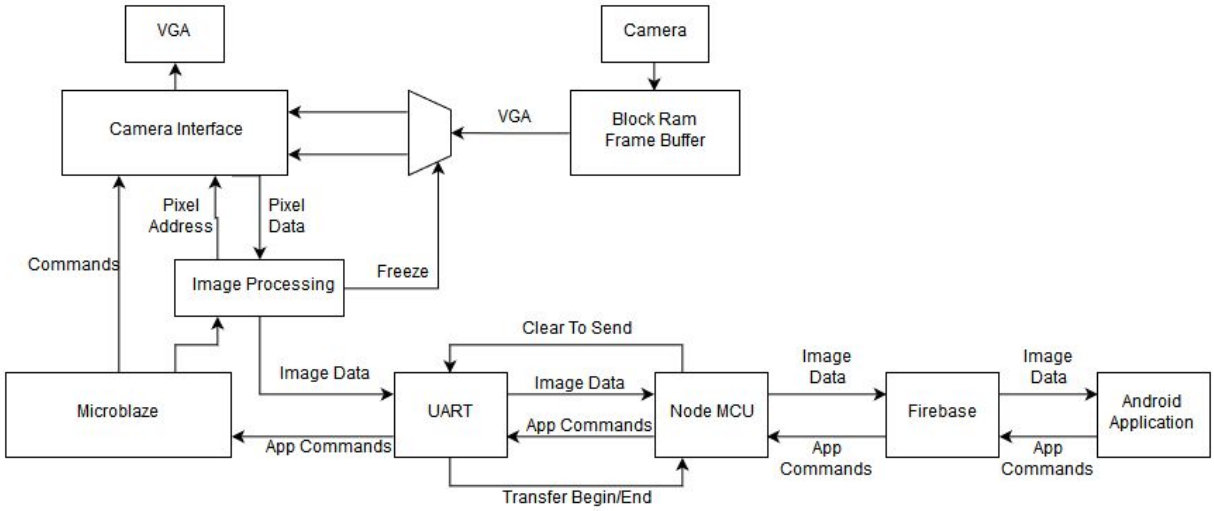


Fig 1: Diagram of the embedded system

The diagram remains similar to that discussed during the progress report. However, the motor is no longer being implemented in the project. The Android app can communicate control parameters to Firebase and receive images.

2.0 Hardware

OV7670 Camera

The hardware involves setting up the camera for display through the VGA ports. Although the VGA display was not a primary goal of this project, it was very useful for debugging the quality of the camera capture. Most of the hardware setup is done through HDL verilog code within Vivado. There is a camera controller module that is used for setting up the camera for I2C writes. Writes can be done using software for the I2C peripheral but the shell of the peripheral is setup in verilog. This peripheral takes a 16-bit

command value where the most significant 8 bits correspond to the address of the camera register and the least significant 8 bits are the value that is written to the register.

The camera capture module controls how frames are captured by the camera and how they will be displayed. As data comes in from the Serial Camera Control Bus (SCCB), this module waits for the right moment to capture a pixel and send out the address and the pixel data for writing to the BRAM module. Data is setup on the camera for RGB565 but since we need to see what the image looks like on the VGA, we need to convert this to the VGA compatible RGB444 format. The capture module uses a write enable signal to let the RAM know when it has valid RGB pixel data to write.

The VGA module reads valid pixel data from the RAM in order to display the pixels through the VGA ports in an array. The camera was originally configured to display an image size of 640x480 but due to this limited memory in our embedded system and the slow transfer rate of our image processing we decided to use a display size of 160x120. VGA uses the vsync and hsync signals along with the valid address to direct the VGA when it has a pixel that it wants to display and which row and column that it belongs in.

Our primary goal for the camera in this project was to take a picture and not stream it through the VGA display. In order to do this while maintaining the integrity of the VGA, we used a freeze frame signal that gets detected by the capture module. When this signal goes high, the capture module freezes the address bus which basically makes the VGA display go dark while a frame is being captured. The control of the read address to the BRAM gets transferred to the AXI bus, allowing the software to select the address that it wants to read from memory. So while a frame is frozen, software can iterate through the entire frames address space and grab all the pixels it needs to create the frame size that it wants. Once the frame is captured, it can be sent through the UART port to be read in by the NodeMCU.

3.0 Software

3.1 Image Processing

A single frame is extracted from the camera frame buffer in 444RGB format as a 12-bit word. Each word is encoded into the Base-64 encoding format before being sent to the NodeMCU, and ultimately to Firebase. Encoding in the Base-64 format is necessary because there needs to be a means for easily distinguishing between pixel values. Long strings of numbers mean individual pixel values become

indistinguishable from each other. And since the source data is 12-bit words and UART transfers are 8-bits wide, each pixel will need to split into two consecutive transfers anyways. The Base-64 algorithm takes in a 12-bit pixel word is splits it into 2 6-bit values. Each 6-bit value is used as an index into an array containing only alphanumeric ASCII characters (plus 2 extra special characters). The character at the specified index value becomes the encoded value of the 6-bit number. In the Android app, the BaseDecoder class decodes a pair of ASCII values into the 12-bit RGB value which is used to populate pixels in the image reconstruction.. Each pair of characters is individually decoded to the corresponding decimal value and the values are bitwise manipulated to reconstruct the 4-bit RGB values.

3.2 Image Transfer

Communication between the NodeMCU and the FPGA is handled over UART. Using SPI would have been the preferred protocol but limitations of the Arduino/ESP8266 ecosystem prevented us from doing so. The asynchronous data transfers for images are implemented with a destination controlled protocol where the NodeMCU asserts a “Clear To Send” pin when it is ready to accept new data and de-asserts the pin while it is sending data to Firebase. The FPGA uses a “Request To Send” pin to communicate to the NodeMCU the “start” and “stops” of an image transfer. The NodeMCU uses these signals to update a dedicated member in Firebase that the Android app uses to know when changes to the image string location are for a new image and when to convert the string into a bitmap..

3.3 System Communication

Hooks were built for a more robust system communication scheme including getting notifications from the Android app. However, due to time constraints and not being able to incorporate the motor, these were not fully tested and the main program was scaled back to eliminate the unused functionality. The NodeMCU has a periodic timer interrupt that polls specific Firebase members to check if signals have been sent from the app to move the camera position or take a picture. The NodeMCU then sends the data to the FPGA over serial which is read by the embedded system on the FPGA.

4.0 Android Application

The Android Application hierarchy is shown at a high level in **Figure 2**. This application consists of the following: two activities for main and splash activity, the main activity has the fragment adapter subclassed. MainActivity also instantiates ImageDecoder and BaseDecoder classes for decoding base64 encoded strings. Below, each of the important files shall be discussed.

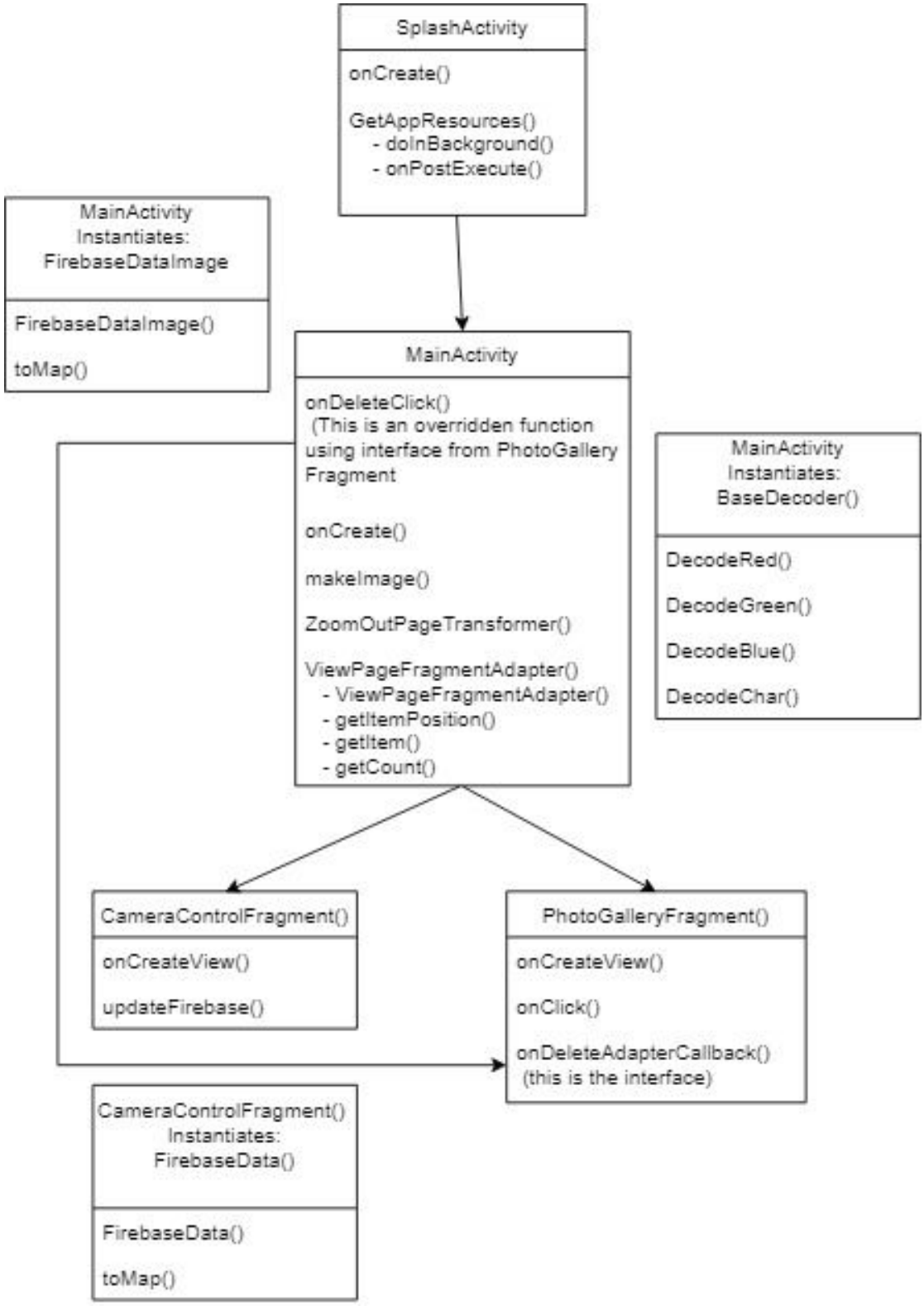


Figure 2: This is the layout of the Android application. Take note where the fragments are managed and the class instantiations. Additionally, the MainActivity implements an interface.

4.1 SplashActivity

This activity is responsible for grabbing web URIs for images. Upon execution of Async task, doInBackground method grabs web URIs for images. At the moment, fifteen image paths are being searched. Once the list is gathered an intent is created with list of URIs supplied. Main activity receives this data once started.

When testing, images could not be displayed when the web URIs were passed from the Activity to the Fragment. To work around this issue, PhotoGalleryFragments receive a position argument rather than the web URI; therefore, the fragment queries the database for the image. The position argument indicates the fragment position in viewpager, and is used to reference path to image. The purpose of passing the URI list is to have a handle to the current images on database as well as being able to properly allocate the correct number of ViewPager slots. **Figure 3** demonstrates how the URIs are being passed to MainActivity.

Lastly, a delay was added after each call to the getDownloadUrl method. Without it, splash activity would start the methods but not wait for the result.

```
@Override
protected void onPostExecute(Void aVoid) {
    super.onPostExecute(aVoid);

    Intent intentObj = new Intent( packageContext: SplashActivity.this,MainActivity.class);
    ArrayList<String> mTemp = new ArrayList<>();
    for(int i = 0; i<mUris.size(); ++i) {
        mTemp.add(mUris.get(i).toString());
    }
    intentObj.putStringArrayListExtra(MainActivity.EXTRA_URI_LIST,mTemp);

    startActivity(intentObj);   // Start MainActivity.
    finish();
```

Figure 3: Intent created with list of URIs.

4.2 Camera Control Fragment

This fragment manages the camera controls and updates firebase accordingly. Only one of these classes are instantiated during program run time. Options available to the user is adjustment of camera position, capture rate, and a manual mode for capturing images. Figure X shows the layout of the app. Note the small purple button on the right hand side. The purpose of that button is to update firebase when user

selects their options. This was necessary because the custom circular seekbar used doesn't have a method for detecting the progress bar stopping. Every change initiates a method detecting the change in progress; therefore when we were updating firebase within this callback, the applications was lagging. The FirebaseData class is instantiated within this fragment. Purpose of this class is to create a key value pair for retrieving and storing data members from Real Time Database. **Figure 4** demonstrates the main purpose of this class; send commands to camera.

```
private void updateFirebase() {

    data.mIsManual = mSwitchModeStatus;     // Set members of the structure
    data.mCapRate = mIntervalSeekProgress;
    data.mCamPos = mCamSeekProgress;
    // Get current time: https://stackoverflow.com/questions/36301543/get-t
    data.TIMESTAMP = DateFormat.getDateTimeInstance().format(new Date());
    Map<String, Object> dbaseValues = data.toMap();  // Create map object c
    // Update Firebase.
    mDatabase.updateChildren(dbaseValues);  // Overwrite database.
}
```

Figure 4: Method to update firebase with user selections. Invoked on button press.

4.3 Photo Gallery Fragment

The purpose of this class is to inflate an image view using the URI of the image. This class receives an intent with the position of fragment respect to Viewpager. This value is used for creating the image path. Once downloaded, image is inflated using the Glide API as shown in **Figure 5**. In addition to the ImageView, the layout contains a button for deleting images as well. This class has an interface which the MainActivity implements and overrides. The interface is for the onClickListener method of the delete button. **Figure 6** shows the interface declaration as well as the interface methods.

```
mStorageReference.child("img" + Integer.toString(mPosition)+ ".png").getDownloadUrl().addOnSuccessListener((

        // https://github.com/codepath/android_guides/wiki/Displaying-Images-with-the-Glide-Library
        // Was having issues getting the URI to show, using the Glide library worked.
        Glide.with(getContext()) RequestManager
                .load(uri) DrawableTypeRequest<Uri>
                .override( width: 1024, height: 900) DrawableRequestBuilder<Uri>
                .centerCrop() DrawableRequestBuilder<Uri>
                .fitCenter() DrawableRequestBuilder<Uri>
                .into(mImageView);

});
```

Figure 5: Getting download URI of image path and setting imageView with valid URI.

\

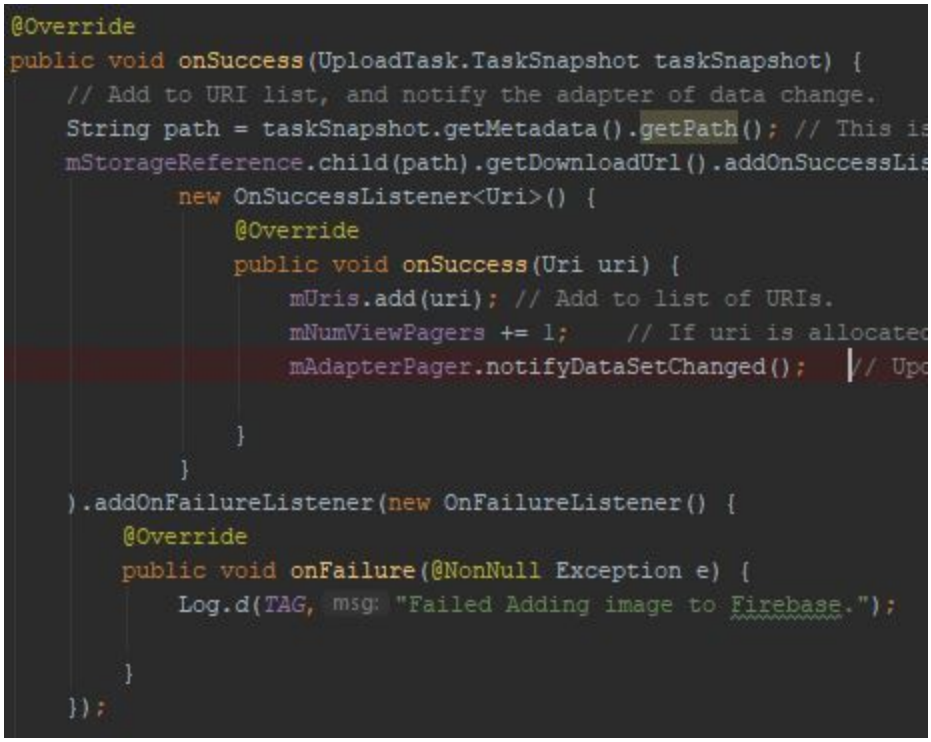
```
/**
 *  Interface implemented by MainActivity.
 *
 */
public interface onDeleteAdapterCallback{
    // Has integer paramater for fragment position.
    public void onDeleteClick(Integer position);   // Destructor method to be overridden by main activity.
}
```

Figure 6: Interface used by MainActivity. The function to be overridden is onDeleteClick() which receives the current position of fragment.

4.4 Main Activity

MainActivity does the work of managing the Viewpager and fragments. The first position of the viewpager will always have the camera controls. After that, each position is inflated with PhotoGalleryFragment type. To add images, the database listener will concatenate strings until the command signal is given to stop transactions. A bitmap will be generated and the resulting image will be uploaded to a distinct path on Firebase as shown in **Figure 7.** Once the image is uploaded, a listener reports success and the number of viewpagers are updated. The adapter is notified and adjusts the positions of fragments accordingly. Once the adapter is notified, the method in **Figure 8** is called and a new object is created.

MainActivity implements the interface created in PhotoGalleryFragment. The onDeleteClick() method is used to delete the current visible fragment. Once a fragment is deleted, the view pager count is updated and the adapter is notified of the changes. At the moment, additional logic needs to be implemented to manage deletion of images. Firebase does not provide support in the API for determining what is in storage. Therefore a list of URIs needs to be managed during run time. The application at this point will delete the viewpager, but not the image stored on Firebase.

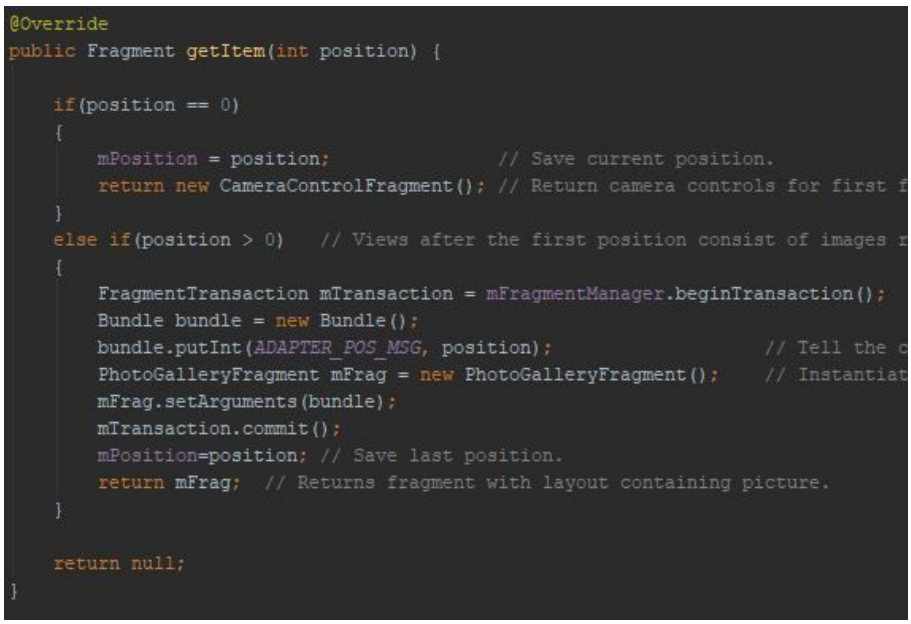


```
@Override
public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
    // Add to URI list, and notify the adapter of data change.
    String path = taskSnapshot.getMetadata().getPath(); // This is
    mStorageReference.child(path).getDownloadUrl().addOnSuccessLis
            new OnSuccessListener<Uri>() {
                @Override
                public void onSuccess(Uri uri) {
                    mUris.add(uri); // Add to list of URIs.
                    mNumViewPagers += 1;    // If uri is allocated
                    mAdapterPager.notifyDataSetChanged();   // Upd

                }
            }
    ).addOnFailureListener(new OnFailureListener() {
        @Override
        public void onFailure(@NonNull Exception e) {
            Log.d(TAG, msg: "Failed Adding image to Firebase.");

        }
    });
```

Figure 7: This is the listener for uploading a new image to the database. On success, the number of viewpagers is updated and the adapter uses the new URI to get the image.



```
@Override
public Fragment getItem(int position) {

    if(position == 0)
    {
        mPosition = position;              // Save current position.
        return new CameraControlFragment(); // Return camera controls for first f
    }
    else if(position > 0)   // Views after the first position consist of images r
    {
        FragmentTransaction mTransaction = mFragmentManager.beginTransaction();
        Bundle bundle = new Bundle();
        bundle.putInt(ADAPTER_POS_MSG, position);                // Tell the c
        PhotoGalleryFragment mFrag = new PhotoGalleryFragment();    // Instantiat
        mFrag.setArguments(bundle);
        mTransaction.commit();
        mPosition=position; // Save last position.
        return mFrag;  // Returns fragment with layout containing picture.
    }

    return null;
}
```

Figure 8: Method which instantiates all fragments.

4.5 Image Decoder Class

The image decoder class provides an encapsulation around the process of converting the received string of characters into the correct bitmap image. The input string is decomposed to an array of char-types. Then, the height and width of the bitmap image is determined dynamically based on the length of the string. Safety checks are performed to make sure the height and widths aren't set outside the bounds of the character string once it gets converted. An empty bitmap is created and then each pixel is manually set based on the decoded RGB values from pairs of characters.

5.0 Challenges Faced

- The first mistake made was the super class used for the Fragment adapter. Initially FragmentPagerAdapter was used; when fragments were attempting to be dynamically added and removed, the app was behaving oddly. For example, the widget didn't have the photo anymore but the viewpager was still allocated for the fragment. Later it was learned that FragmentPagerAdapter can't delete fragments. To fix this issue, super class FragmentStatePagerAdapter was used. The method notifyDataSetChange() of the class will update the positions of the fragments once fragment data is altered.

- At the time of the demo presentation, image transfers were inconsistent in the sense that we frequently experienced data loss when doing the transfers. The UART buffer has a max FIFO of 16 bytes and the buffer was filled for each transfer. There is logic to indicate if a transfer does not complete successfully, but no warnings were given. Analysis of console printouts of characters "sent" by the FPGA shows the correct amount sent. Investigation with a logic analyzer showed that data "appeared" to be transferred but the sheer size of the data and random nature of the characters being sent meant true analysis was impossible. In hindsight, if we had more time, a more robust communication scheme would have to be developed where error checking could automatically be done in software. Preferably, migrating to the Espressif tools where we can use the proper SPI peripheral would be best. After our demo presentation, we tried slowing down the baudrate from 115200 to 9600 and now we are now getting consistent and accurate results. Current testing shows that we are "only" experiencing a 14% average data loss and have a standard deviation of 260 bytes transmitted but the results are consistent. The 14% data loss is apparent when viewing reconstructed images where parts of the image are shifted and offset across rows due to the missing characters.

- Image reconstruction was harder than anticipated. At face value, base-64 encoding seemed easy to implement as there are online encoder/decoders and even Java libraries. However, there are over a dozen different variations of the encoding scheme and few algorithms. The algorithm we chose did not match the built in libraries and finding a solution was one of the reasons we were unable to complete the project on time. However, after the demo presentation, we did come up with a solution which was to build a ImageCreator class that takes in the string of received characters and manually assembles a bitmap pixel-by-pixel. See software section for more info on ImageCreator class. We still had problems, however. Determining how to take a 444RGB value and scale it to values usable by Android was difficult. The Android Color classes use 32-bit values for their colors, but the 4-bit value can only scale so far before it becomes saturated. A simple linear scaling proved to at least give us something but more advanced techniques would be required to get better images or further work will be needed to extract higher resolution RGB values from the camera and transmit them even if there is a time penalty for the higher quality data.

- We were able to capture at least one decent image with the camera. Figure 9 shows a reconstruction of a picture with the camera looking out the window. The upper left picture is taken with a smartphone. The upper right is what the picture from the OV7670 looks like. And, in the bottom left, the reconstructed image from the Android application. The bottom right image shows the traces of the window frame and helps give some perspective of the decoded picture. **Figure 9** shows a comparison of the best photo we were able to decode after letting the system run on auto-pilot for a few hours just taking pictures and uploading them to Firebase.

- Even though we eventually got everything working with the camera setup, the initial configuration of the camera was a lot more difficult than we anticipated. It took a lot of toying with camera configuration settings to get a decent image on the VGA monitor. Eventually we ended up modifying around 76 camera register values to get a decent image. Even after getting an image on the monitor we had to figure out a way to capture a single image. While we did eventually get all of this working, the process consumed a lot of valuable time, and shortened the amount of time we could work on the image processing. This was mostly due to the fact that we weren't sure exactly what kind of data we would be processing until we captured it.

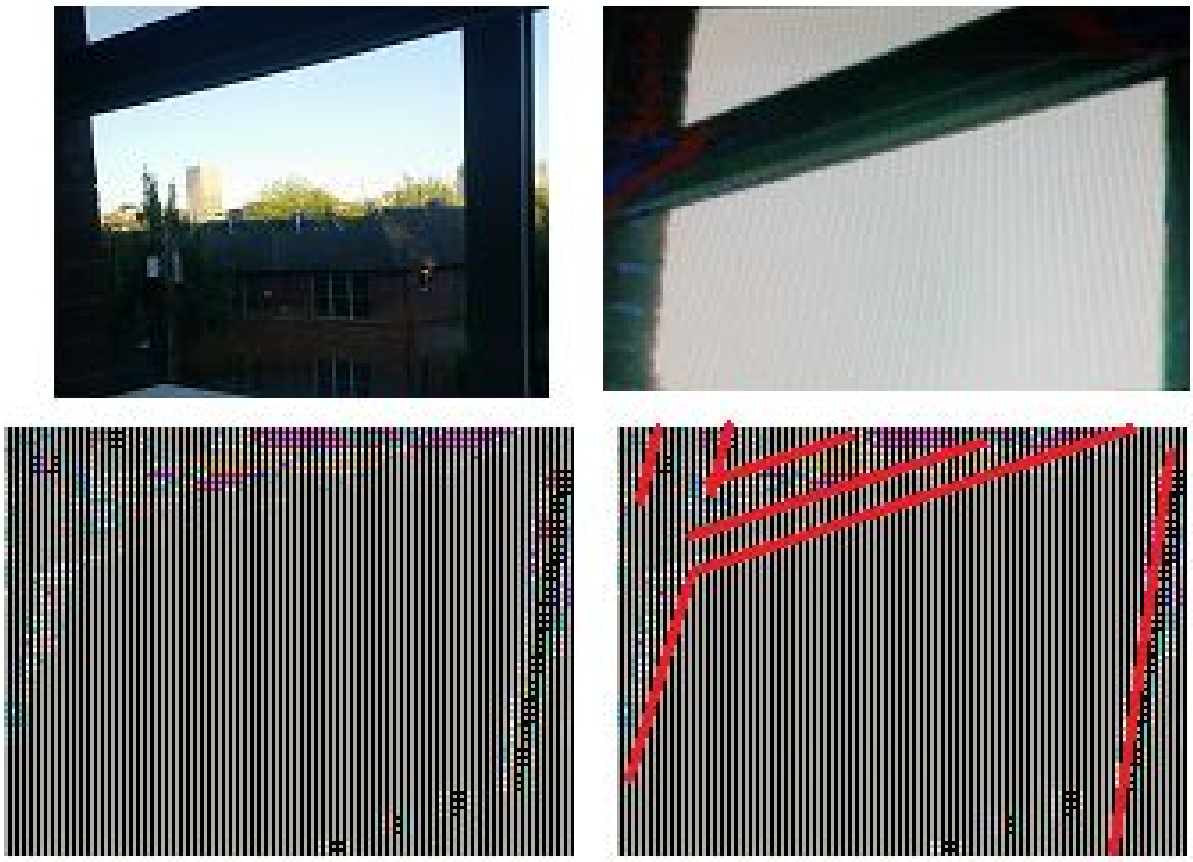


Fig 9: The best photo we could get from the camera shows a window frame against an otherwise sparse image.

6.0 Work Distribution

Andrew Capatina

- Android application
- Image Processing

Ryan Bentz

- Image Processing and Transferring
- Wifi Connectivity
- Embedded System Software
- Embedded System Hardware

Ryan Bornhorst

- Camera Custom IP and Interface
- Image Capturing
- Image Processing
- Embedded System Hardware

7.0 Conclusion

As of this writing, the project remains in progress. The issues with the data loss and problems with the color conversions have been too difficult to overcome before the deadline. It is likely that the design as-is would need a complete overhaul to attain the desired specifications. If we were going to do the project again, the image transferring and processing needs a complete overhaul and the data transfer between the FPGA and NodeMCU needs to be migrated to a more robust protocol with error checking capability. Further research is needed to find a way to move beyond Arduino on the NodeMCU and further research is necessary on how to best recreate images from raw RGB data in Android.

For the Android App, an unexpected issue was management of Firebase images. The API does not offer an easy solution for managing what's on storage. To maintain a list of images, the app must be able to query the database for a range of images. Managing this increases complexity and it shouldn't be done by the phone because its a strain on performance and data usage. Firebase offers functions for delivering data such as number of items to the app, however there is not enough time to explore this option.