ECE532 Final Design Report

Gesture Controlled Multimedia Playback System

Li Zeqi

Yin Yaoming

Yu Suyuan

Contents

1. Overview

1.1 Motivation

Currently voice controlling system is implemented in TV and computer operating systems/apps to help improve the inconveniences of remotes/mouse. However, we found that in the case of multimedia, voice control is one dimensional. While it can achieve basic functionalities like play, pause, and next, it fails to provide a smooth user experience for “two-dimensional” controls like tuning volumes and play speed. With voice control, user can only tune volume or play speed at a certain step. For example, if a user tells the multimedia app to turn the volume up, it will only increase volume by a certain amount, but if the user still wants to increase it he/she will have to speak the command again.

The multi-dimensional nature of multimedia control is what motivated our group to implement a gesture control system. With gestures, the basic functionalities can be achieved with static gestures. For more complicated ones like change volume and play speed, dynamic gestures can be used to both identify which command a user wants to execute and how much he/she wants to adjust preciously. This cannot be easily achieved with voice control as the dimensions of sound (signature, length, and magnitude) are not practical for tuning (e.g. user “yells” for a big increase in sound).

1.2 Goals

The goal of this project is to create a gesture recognition tool to control multimedia playback. The motivation behind the project is the need for a remote or mouse free method of controlling home media with the growing popularity of Internet of Things(IoT). The traditional way of controlling multimedia (e.g. playing, pausing, and going next) typically require some form of remote for TVs or a mouse if played on computer. However, there are several disadvantages of using TV remotes or mouse. For example, TV remotes need batteries to operate, cause difficulties when objects block the infrared light, and require pinpointing to a small button or a small block of pixel for the case of computer to control multimedia.

1.3 High-level Description

Our gesture recognition system consists of a black background (a constraint scoped for this project), a camera, a server on the computer, and a FPGA. The flow of our system goes as follows: the FPGA continuously samples pixels output from the camera, then it formats the pixels into one frame and pass it to two IP blocks that run area and aspect ratio algorithms (will be explained in detail in later sections). The IP blocks would then output area and aspect ratio results to the server through TCP-IP protocol. In idle state, server receives the data from FPGA but does nothing. When a user shows a gesture within a specified area in front of the black background, the server will then execute multimedia control command accordingly based on recognition result it identified from area and aspect ratio results sent from FPGA.

1.4 Block Diagram

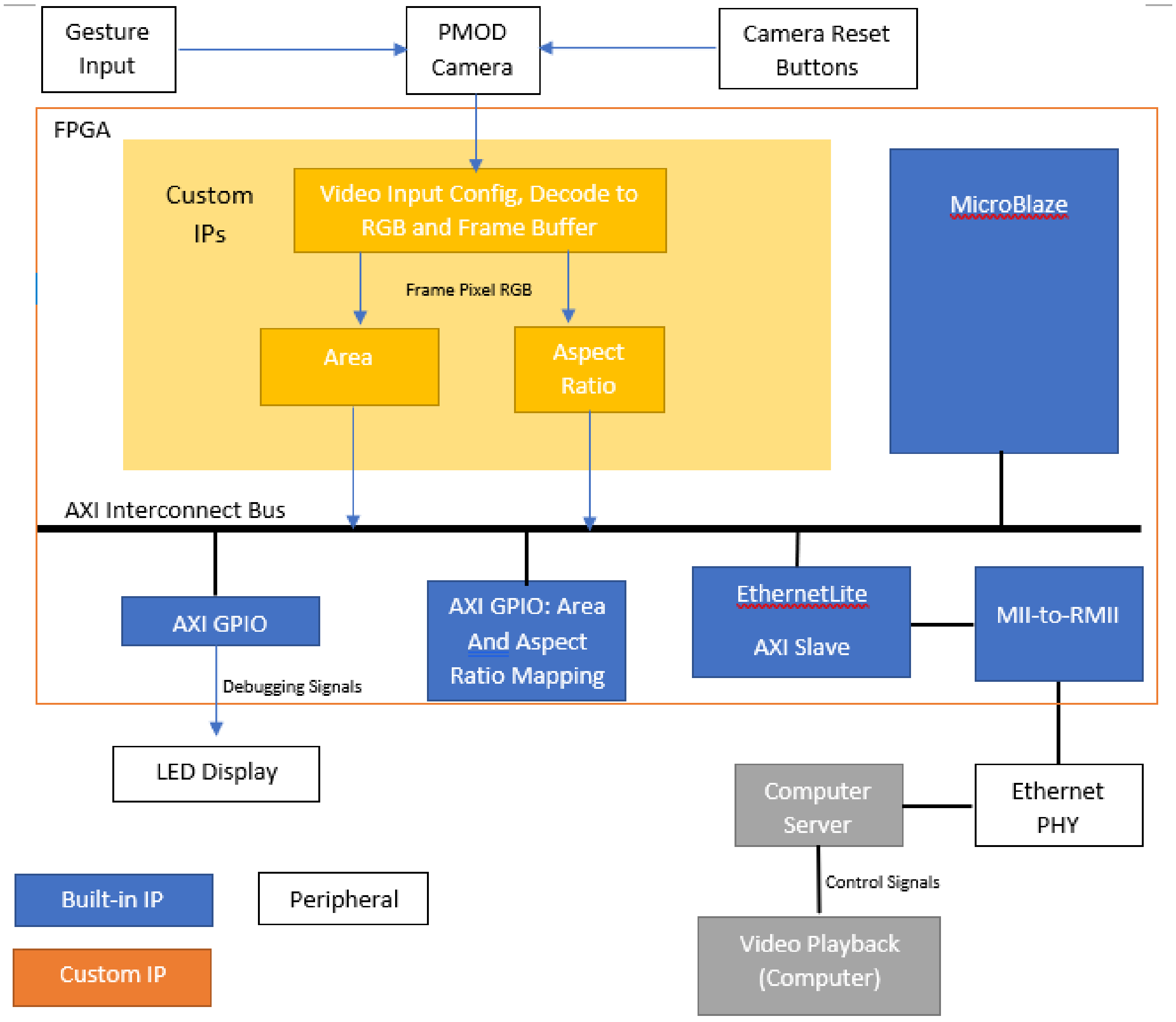


Fig. 1 Block diagram

1.5 Brief Description of IPs

- Reference Design: OV7670 PMOD Camera Decode and Configuration

The OV7670 PMOD reference design, provided in Piazza Resources, contains basic PMOD capture (ov7670_capture), I2C configurations (I2C_OV7670_RGB444_Config), video decoder as well as VGA video output setup (vga444) in order to directly show what camera captures on VGA display. In this reference design block, it converts real-time raw data that camera captures to pixels in RGB444 values and parse these values to VGA input, pixels are also stored into BRAM for buffering.

Our design on real-time Gesture Recognition is based on this reference design for acquiring RGB values for each pixel in each video frame for Gesture Recognition algorithms.

- Area in a Frame

The area block mainly calculates the number of bright pixels in one video frame. The pixel RGB values that are provided from video decoder are used to compare with a pre-set RGB threshold. The block outputs the number of total bright pixels each frame in real time after evaluating the last pixel in each frame.

- Aspect Ratio in a Frame

The aspect ratio block compares each pixel's RGB with a pre-set threshold to count the range of bright pixels in vertical and horizontal directions of one frame. The block will output the aspect ratio value by dividing vertical bright pixel range with horizontal range for each frame.

- AXI GPIO

Utilizing AXI functionalities on Microblaze with GPIO to store the results output from two gesture recognition custom IP blocks for SDK access. Besides, we also output the results to LEDs via GPIO for hardware debugging.

- EthernetLite, MII-RMII

The EthernetLite provides a data tunnel from FPGA to the multimedia playback server. At the time when LAN is established, the outputs of the two gesture algorithms are sent through Ethernet to server for gesture classification.

2. Outcome Results

2.1 Design Changes

- Communication:

Instead of bidirectional communication specified in the proposal where the FPGA sends <encoded result> and server responds with <success/fail>, our final communication protocol is unidirectional. The FPGA continuously sends the data in the format of <ratio measurement:area measurement>. The server would then process this data with its logic without replying to FPGA.

- Memory:

Instead of DDR memory proposed in proposal, we used GPIO as a FIFO to store the ratio and area measurement as arrays of 8-bit integer and 7-bit integer respectively. The width of those two arrays are determined by our design decision on the number of significant digits used for the measurements (3 digits for ratio and 2 for area to be specific).

- OLED:

OLED is no longer used to display the result of the recognition. For debug purposes we used LEDs to display the area and aspect ratio measurement in binary format.

- Gestures:

In proposal we identified three static gestures. In our final projects, we have five gestures: three static gestures that control play, pause, and next functionalities as described in proposal, and two dynamic gestures to control volume up and down.

2.2 Results

Our gesture recognition functionality was all successfully achieved (refer to Table 1). However, the downside is that the system requires retuning under environments with different ambient light settings. Also, there are some learning required for users. (the functionalities worked fine with three of us) Our specified rules for user interaction are:

1. Users should put their gestures into the frame in a relatively fast manner (0.2 seconds to be exact). This is mainly due to our detections algorithm (explained in detail in later sections)
2. After the gesture is within frame, wait for the response of the server and then remove hand out of scene.
3. For the next gesture repeat step 1 and 2

These rules could be a little bit hard to learn at the beginning which was what happened during the demo, but after playing it for a while, most users can adapt it without problem. As far as we concerned, these rules are acceptable because they provide robustness and accuracy of the system, which will offer the best user experience.

Gesture Type	**Gesture**	**Result**
Static	Play	Completed with very high accuracy (~1 in 50 failed)
Static	Pause	Completed with very high accuracy (~1 in 50 failed)
Static	Next	Completed with very high accuracy (~1 in 50 failed)
Dynamic	Volume Up	Completed with high accuracy (~1 in 30 failed)

Dynamic	Volume Down	Completed with high accuracy (~1 in 30 failed)

Table 1: result of all functional gestures

2.3 Future Work and Potential Improvement

Although we completed our project well beyond our original expectation, we still identified many areas could be improved in the future

- Noise-proof version of ratio algorithm:

One of the most difficult problem we have encountered was the unstable behavior of our ratio algorithm in very bright rooms. Currently, our aspect ratio algorithm is subjected to a single pixel failure meaning if the recognition logic misreads one black pixel as a white one, the ratio measurement could become extremely inaccurate. In fact, one strange behavior we observed was the ratio measurement of the black background sometimes fluctuate to 0.75, which is an all-white background under assumption of a noise-free environment. However, it is more likely caused by misclassified pixels on the border of the frame.

At the time of development, we have thought about more robust algorithm for calculating aspect ratio like we can record the longest streak of white pixels to determine its horizontal and vertical range. This will clearly give us more valid range because it will skip all isolated white pixels. However, this method proved to be very memory intensive and we seldom observed the “problem of 0.75” in our testing environment. Therefore, we decided to proceed with our original implementation.

As a natural extension, one of the future work will be to complete this noise-proof implementation of our aspect ratio algorithm.

- More gestures

Currently our project only includes implementation of 5 gestures, but there are still more controls can be done in multimedia playbacks. For example, fast forward and backward was one thing we considered to add but designing the gestures that distinguish from the rest we already defined requires extra time and effort. For the time being we resorted to 5 gestures instead of 7 if added fast forward and backward. If we have more time to develop this project, we would realize those two gestures in our implementation.

3. Project Schedule

Original Proposed Milestone Schedule

Milestone 1	Add GPIO IP cores into block design, get familiar with building custom IP blocks and utilizing AXI interfaces. Write and test Ethernet Server blocks further based on the result of warmup project
Milestone 2	Add Camera PMOD into design, figure out how to read output from camera, start writing gesture detection and recognition algorithm
Milestone 3	Continue on gesture recognition algorithm, start writing testbench for custom IP, setup TFT controller and OLED module for debug and visualization
Milestone 4 (Mid-project Presentations in lecture)	Finalize first stage of gesture recognition algorithm, simulate it on testbench, demo the functionality of first stage
Milestone 5 (Mid-project Demo in lab)	Finalize second stage of gesture recognition algorithm, simulate it on testbench, demo the functionality of second stage
Milestone 6	Integrate all hardware IP blocks, realize software capability on server to control multimedia playback based on received gesture type
Milestone 7 (Final Demo)	Debug and polish our system, prepare for final demonstration

Table 2: Original milestone accomplishment

Actual Weekly Milestone Accomplishments

Milestone 1	Experiment with custom IPs, get familiar with outputting result via GPIOs. Verify TCP-IP blocks for data transmitting and receiving
Milestone 2	Add Camera reading IP. Visualize change of pixel value on LED while moving the camera. Attempt to use VDMA module to read consecutive frames into MicroBlaze
Milestone 3	Introduce Reference Design to make the PMOD camera working with VGA display. Implement 2 gesture recognition algorithms on software and verify correctness of the algorithms by capturing example pictures using Arduino
Milestone 4 (Mid-project	Implement the first (area-based) gesture recognition algorithm on hardware and encapsulated it into one custom IP. Tune the brightness threshold for differentiating between hand pixel and background pixel.

Presentations in lecture)	Record pixel count (area) for all 3 static gestures by outputting the first 7 bits of area result to LEDs
Milestone 5 (Mid-project Demo in lab)	Integrate the area-based recognition IP with existing systems. Realize multimedia playback control on server. Test interaction between FPGA and server. Realize multimedia control logic on server Start to implement the second (aspect ratio based) recognition algorithm Design 2 dynamic gestures: Volume up and Volume down and try to incorporate them into existing gesture list.
Milestone 6	Encapsulate aspect ratio-based algorithm into a custom IP. Modify communication encoding to accommodate aspect ratio result. Tune all parameters for 5 different gestures. Improve robustness in both hardware implementation and software handling.
Milestone 7 (Final Demo)	More improvement on system robustness and reliability. Fainalize all tunable parameters. Construct a camera mount for stabilizing video shooting. Prepare for final demonstration

Table 3: Actual milestone accomplishment

By comparing the actual accomplishments with our original plan, we have found out that:

- In the first 2 milestones, we spend longer time than we expected to figure out how to setup PMOD camera on FPGA and read/store video frames
- We used LEDs to output recognition results for debugging purpose and visualization is done on VGA display and LEDs instead of originally proposed OLED
- A solid PMOD camera mount has been built for stabilizing video shooting and ease of use

4. Description of the Blocks

4.1 OV7670 PMOD Camera Decode and Configuration block

The block contains essential I2C configuration and video decoding for FPGA to read video frames in RGB from OV7670 PMOD camera. It also contains setup and configuration files for VGA display showing real-time camera video captured.

4.2 Area IP Block

This IP block is for the first gesture recognition algorithm, area/threshold. In a given unicolor (we used black in this project) background frame, a hand in front of background will produce bright pixels with RGB values that are significantly higher than the RGB values of background. One is able to setup a threshold RGB value to differentiate bright pixels which represents hand gestures and dark pixels representing background (Fig.2). Thus, the area of hand gestures can be calculated by counting the total number of bright pixels in one video frame.

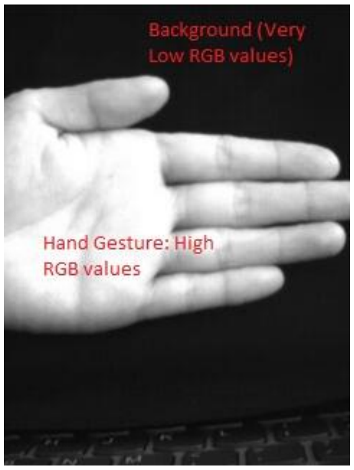

Background (Very Low RGB values)
Hand Gesture: High RGB values

Fig.2 Demo image

Hand gestures can be differentiated by comparing its area with certain pre-set area ranges because the sum of bright pixels (usually referred as area in our group) is quite different for different hand gestures. For example, the area of gesture “pause” is much smaller than the area of gesture “play next” (Fig.3). This way, 2 gestures can be accurately classified.

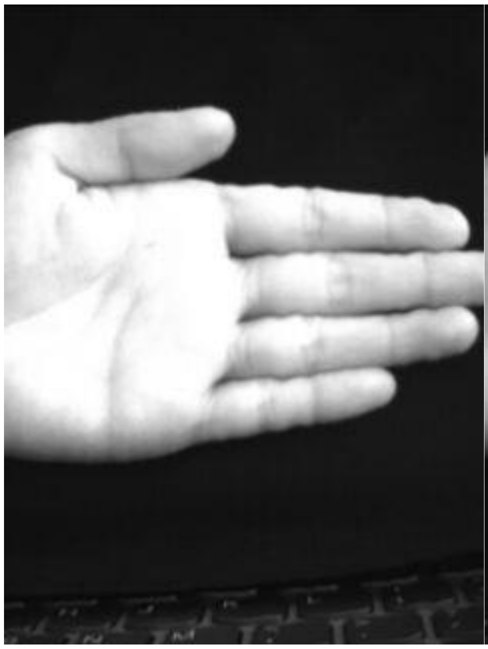


Fig.3.1 Gesture “Pause”

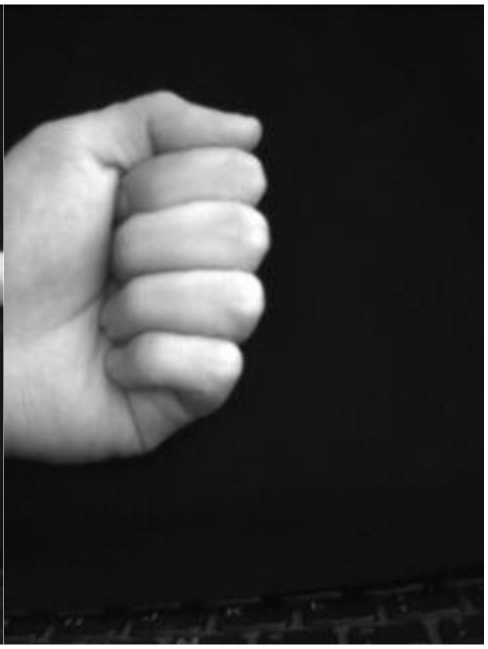


Fig.3.2 Gesture “Play Next”

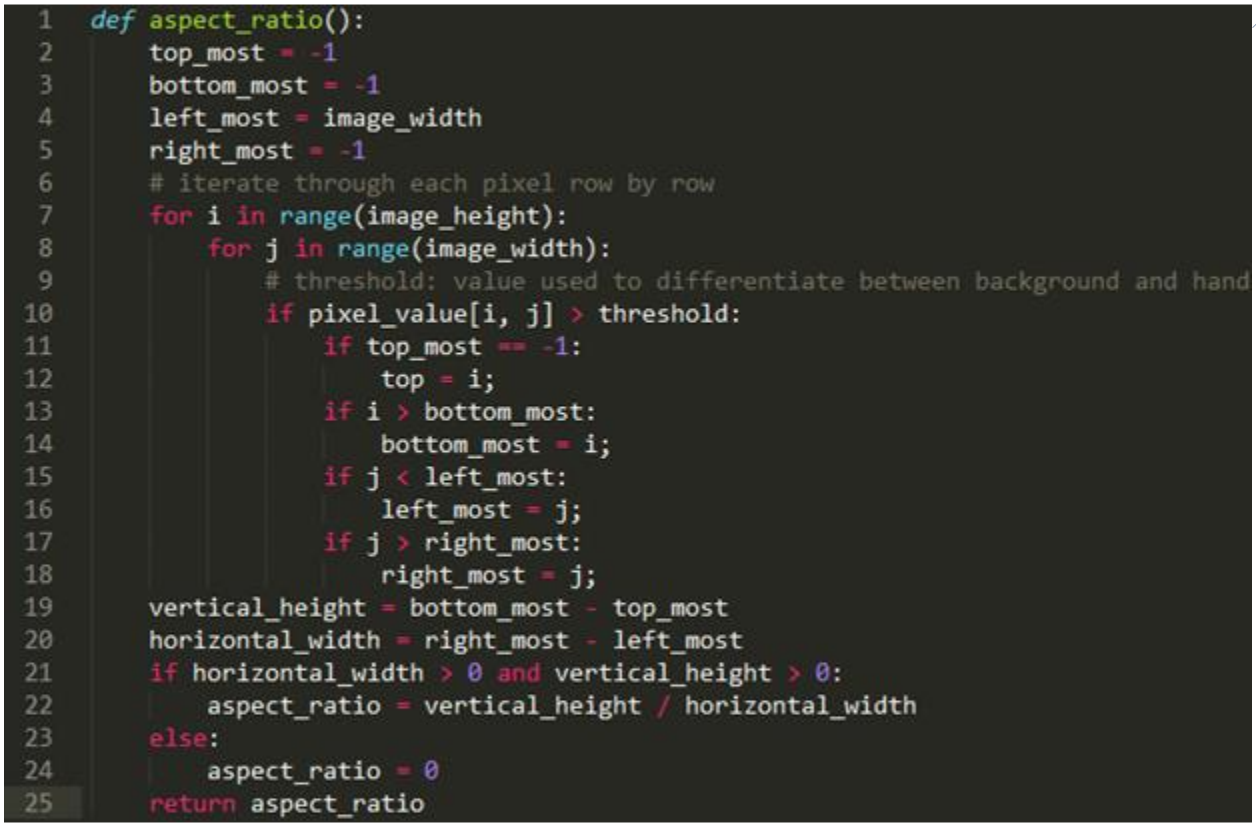
The RGB values are presented in RGB444 form, so pixel value is ranged from 0 to 15 (2^4 - 1). After experiments and calibrations, we set the threshold for differentiating between bright pixels (hand) and dark pixels (background) to 6. Since the area values between different gestures differ by a considerable large margin, we only need to output the first 7 most significant bits (2 significant digits in decimal which can be represented within the range of 128/2^7). The ranges of area for different gesture are listed below:

Gesture Command Name	Bright Pixel Area Range
Play	22.5 ~ 33
Pause	18 ~ 22.5
Play Next	>= 33
Volume Up/Down	5 ~ 18

Table 2: Area range between gestures

4.3 Aspect Ratio IP Block

This IP block is a fully customized block for our second gesture recognition algorithm. This block serves 2 major functions. Firstly, it is used to judge the orientation of the gesture by calculating its vertical height divided by horizontal width. This capability enables our system to differentiate the same gestures in different angles. Besides judging gesture orientation in the static images, it can also be used to detect moving gestures such as controlling volume dynamically. Basically, it monitors the changes of aspect ratio of the gesture from hand movement and translate to volume changes. Secondly, it is used to assist Area based algorithm to make more robust and reliable prediction of the gestures. The idea is very similar to sensor fusion. Most hand gestures we used in our system differ by more than 1 aspect. The ability to make the prediction based on multiple characteristics will certainly yield higher prediction accuracy. For example, the gestures we used in pause and play have relative close occupied area in some situations. Misclassification happens sometime. However, when aspect ratio characteristic is added into consideration. This type's misclassification is completely avoided as there is a big difference between the ratios of those two gestures. The robustness of the system is improved considerably. The pseudocode for calculating aspect ratio is presented below (Fig.4):



```
def aspect_ratio():
    top_most = -1
    bottom_most = -1
    left_most = image_width
    right_most = -1
    # iterate through each pixel row by row
    for i in range(image_height):
        for j in range(image_width):
            # threshold: value used to differentiate between background and hand
            if pixel_value[i, j] > threshold:
                if top_most == -1:
                    top = i;
                if i > bottom_most:
                    bottom_most = i;
                if j < left_most:
                    left_most = j;
                if j > right_most:
                    right_most = j;
    vertical_height = bottom_most - top_most
    horizontal_width = right_most - left_most
    if horizontal_width > 0 and vertical_height > 0:
        aspect_ratio = vertical_height / horizontal_width
    else:
        aspect_ratio = 0
    return aspect_ratio
```

Fig.4 Pseudocode for calculating aspect ratio

Its recognition result will be combined with area recognition result to make more robust and comprehensive prediction.

4.4 AXI GPIO

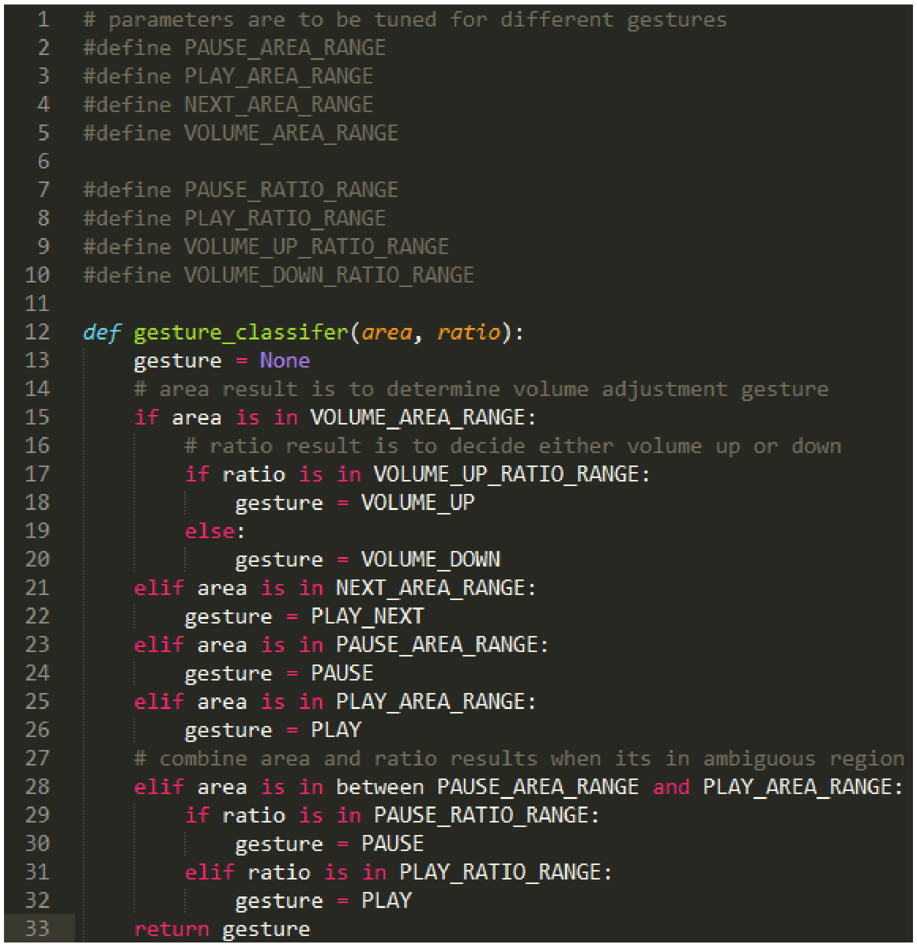
2 recognition IPs will output their results to 2 sets of on-board LEDs via AXI GPIO. On board LEDs are used for both storage and debugging purpose. Those values will be continuously read in MicroBlaze and 2 results will be encoded and sent to server via LAN.

4.5 EthernetLite AXI Slave and MII-to-RMII

Those are build-in IPs used for LAN communication. It is configured as client in our system and will be responsible for sending area and aspect ratio results to server.

4.6 Multimedia Playback Control

After receiving area and aspect ratio results from MicroBlaze via LAN. Server will issue multimedia playback control signals accordingly. Control signals are issued based on the results of area and aspect ratio in 1 second window. In order to make the prediction more robust and resistant to noisy data, our sampling frequency for area and aspect ratio data is set to 10Hz (read data for every 0.1 second) and controller will fill up a 1.2 second queue consists of 12 data pairs. The first 2 data pair (0.2 second) will be discarded to regard as gesture stabilizing buffer (buffer time for gesture to be stabilized in front of camera). Then average area and aspect ratio will be calculated based on all non-zero entries. To further reduce fluctuation of received data, potential outlier data pairs are removed, which means 1 maximum value and 1 minimum value are excluded in calculating average area and aspect ratio. Gesture is then judged based on the average area and aspect ratio. Pseudocode of deciding different gesture type is presented below (Fig.5):



```
# parameters are to be tuned for different gestures
#define PAUSE_AREA_RANGE
#define PLAY_AREA_RANGE
#define NEXT_AREA_RANGE
#define VOLUME_AREA_RANGE

#define PAUSE_RATIO_RANGE
#define PLAY_RATIO_RANGE
#define VOLUME_UP_RATIO_RANGE
#define VOLUME_DOWN_RATIO_RANGE

def gesture_classifer(area, ratio):
    gesture = None
    # area result is to determine volume adjustment gesture
    if area is in VOLUME_AREA_RANGE:
        # ratio result is to decide either volume up or down
        if ratio is in VOLUME_UP_RATIO_RANGE:
            gesture = VOLUME_UP
        else:
            gesture = VOLUME_DOWN
    elif area is in NEXT_AREA_RANGE:
        gesture = PLAY_NEXT
    elif area is in PAUSE_AREA_RANGE:
        gesture = PAUSE
    elif area is in PLAY_AREA_RANGE:
        gesture = PLAY
    # combine area and ratio results when its in ambiguous region
    elif area is in between PAUSE_AREA_RANGE and PLAY_AREA_RANGE:
        if ratio is in PAUSE_RATIO_RANGE:
            gesture = PAUSE
        elif ratio is in PLAY_RATIO_RANGE:
            gesture = PLAY
    return gesture
```

Fig.5 Pseudocode for classifying gestures

Multimedia playback controller can be clearly represented in a finite state machine as below (Fig.6):

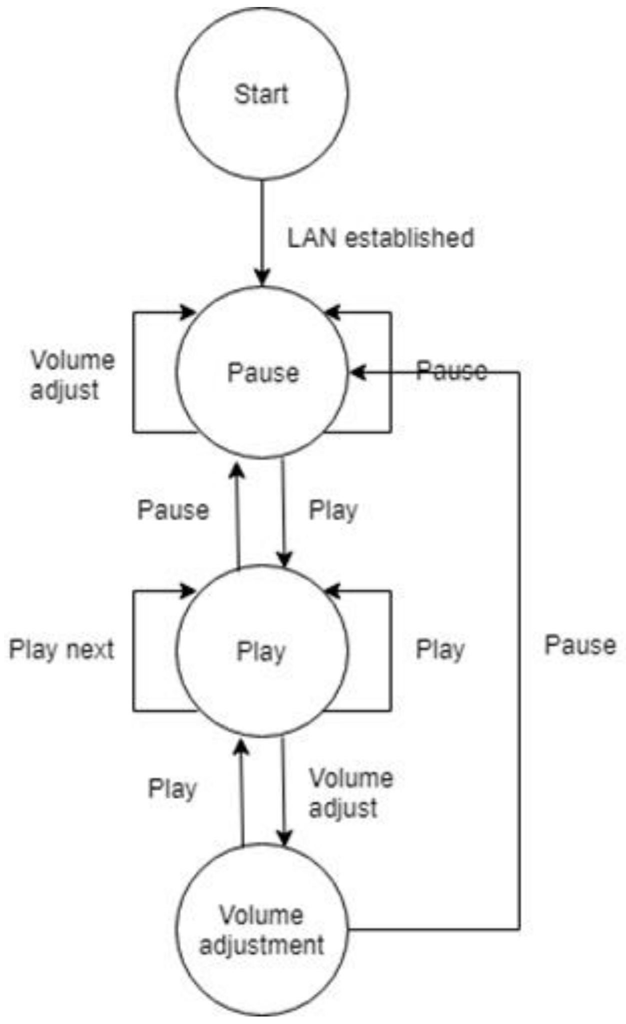


Fig.6 Multimedia control logic

In text, the finite state machine above means that when the server is started, multimedia is pause. During pause state, ONLY play command will be accepted. While in play state, commands other than play are accepted.

5. Description of The Design Tree

The all core design files are available on GitHub:
https://github.com/FusionLi/G1_GestureControlledMultimediaPlaybackSystem

Main Hardware Design: /src/hardware
Camera IP: /src/hardware/camera_ip

- OV7670_top.v: Top level of PMOD Camera configuration and video decoding
- I2C_AV_config.v, I2C_Controller.v, I2C_OV7670_RGB444_Config.v, debounce.v and ov7670_capture.v: Camera configuration, video decoding
- vga444.v: RGB444 value extractions and output frames to VGA display

Area IP: /src/hardware/area_ip

- area.v: Hardware implementation of area based gesture recognition algorithm

Aspect ratio IP: /src/hardware/ratio_ip

- aspect_ratio.v: Hardware implementation of aspect ratio based gesture recognition algorithm

Main Software Design: /src/hardware

Client: /src/hardware/client

- main.c: Ethernet setup and main send loop
- client.c: Connect callback, send callback and sent callback. Also read value from GPIO memory location and write it to send buffer.

Server: /src/hardware/server

- readserver.py: Receives real-time area and aspect ratio results from FPGA, apply range comparison and produce gesture commands to multimedia player.

Multimedia Playback track:

- 0.mp3 ~ 10.mp3: All songs in the multimedia playlist

6. Tips and Tricks

1. Use existing implementation if possible, do not reinvent wheel. For example, if you want to use camera OV7670, it's better to just use the camera module provided on Piazza instead of writing it yourself.
2. Before implement an algorithm on hardware, realize it on software first and think how to convert software implementation to hardware design.
3. Try to keep all of your tunable parameters in software logic instead of hardware IP blocks. It will make your design iteration much faster and less time-consuming.
4. Make things work first and then start to improve on it. Do not try to accomplish too much at once.
5. Make debug information available on both hardware and software if possible. This way you can easily identify whether the bug is caused by the hardware design or software implementation
6. Try to use modules and IP blocks you are more familiar with if possible. Do not try to be fancy and cool when you are time pressure.
7. Start integration as early as possible because it takes long time
8. Use version control tools such as git to keep all your working components in case you accidently break something.

7. Project Video

Viewable at https://drive.google.com/file/d/1KF0rXmzDoO-HfNa-SFUOMAXZRf6p2Ouu/view?usp=sharing