GoPro PTZ Tower Camera

Michael Serignese, Andre Samaroo, Chisom Ikejiani

Dept. of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 32816-2450, USA

***Abstract* — This paper presents the design of a pan-tilt-zoom camera system that is tripod mounted in order to provide a greater effective field of view. The goal of the project includes realization using consumer-grade prefabricated hardware, and therefore focuses on software architecture and communications exchange.**

Index Terms — Cameras, 802.11n

I. Introduction

Commercial-grade pan-tilt-zoom (PTZ) tripod mounted camera systems are used to capture video from above ground level in order to maximize the quality of the shot when recording scenes that include sports action scenes. These cameras can be expensive and therefore inaccessible for purchase to general audiences. If we then examine the nature of the camera setup, logically break down its pieces, and rebuild the device using consumer-grade hardware, we implement the design in a more accessible manner.

II. Overview of Tripod-Mounted Cameras

Tripod-mounted pan-tilt-zoom camera systems have three key components. The key components are as follows:

The media capture device;
The motor system;
The control system.

The media capture device is responsible for physically sampling the frames of real-world video and audio in a digital form. The media capture device may also be responsible for storing the video, or it may transmit the digitally encoded signal upstream to a separate computer system for storage, due to limitations in storage capacity. The media capture device is also responsible for implementing the zoom feature of the setup, implemented either with fixed lenses or digitally.

The motor system is responsible for moving the media capture device to the appropriate angles, panning and tilting as necessary in order to capture the scene according to the operator's instructions. The motors must be rated with an appropriate wattage to provide the power necessary to move the weight of the media capture device at the desired speed. Additionally, the motor type must step in a manner resembling continuous motion in order to avoid vibrations that would interfere with capture quality.

The control system is responsible for supporting the full range of features of the device. The control system also takes responsibility for bringing the media capture device and the motor system under the command of itself. The control system connects to and sends commands to the media capture device and the motor system that grants user control. Furthermore, the control system is also responsible for receiving and displaying the video streamed from the media capture device while also encapsulating the motor software.

III. Transformation of Commercial to Casual Parts

The media capture system is implemented with the *GoPro Hero 7* action camera. The GoPro camera is a class of consumer audio-video recording devices that have 802.11n wireless communications built into the device with an onboard micro Secure Digital (micro-SD) card. The 802.11n radio transmitter is capable of using 5GHz or 2.4GHz frequencies, and the maximum supported transmission bitrate is 300M bits per second according to the specifications of IEEE 802.11n-2009 in high throughput mode.

The micro Secure Digital card installed in the GoPro is a microSDHC Class 10 card with a digital throughput of 10 MB per second.

The digital video format used by the device is H.264 (AVC1) in High Profile mode, using the yuvj420p color profile. The digital audio format is AAC at a bandwidth of 48000 Hz, and a bitrate of 179k bits per second. The media is encapsulated in an MP4 container (version 4.1). When the GoPro video stream is previewed over the wireless interface, the media is encapsulated in the Transport Stream (ts) format.

The motor system is implemented with two Hitec-422 servo motors, a Raspberry Pi zero W, a Sparkfun servo HAT, aluminium mounting hardware, and a 3D printed housing unit.

The 2 servo motors provide 2 degrees of freedom, panning and tilting. The servos have an operating voltage range between 4.8 and 6 volts. The small operating voltage makes them easy to use without a separate power source so we can reduce the number of electrical components under the device housing unit. They also have a current draw between 8 and 180 mA, which is consistent with the control boards delivering power to the servos [1].

In order to control the servos, we used a Raspberry Pi zero W for general communication and a Sparkfun servo HAT to easily control the servos. The Pi uses a Broadcom BCM 43438 for wireless communication [4]. The wireless chip is IEEE 802.11 compliant with bluetooth capabilities as well. The Raspberry Pi has a micro-USB port for power delivery and a second micro-USB port for data control. However, the data port can also be used for power. The Sparkfun servo HAT is an intermediary device that uses a PCA 9685 pulse width modulator to control the servos with better synchronization than the clock built into the Raspberry Pi [2]. The servo HAT has 16 channels for attaching servos and a USB-C port for power. The 2 devices are connected through the Pi's GPIO pins so that they can share power and data. This means that the power port on either device can power the whole system. The Sparkfun servo HAT also gives us access to the Sparkfun servo library which contains simple but useful functions for interacting with the servos.

Another part of the motor system is the mount which includes the frame for the pan-tilt rig and the housing unit for all the motor components. The aluminium frame is part of a kit made specifically for the Hitec HS-422, so the frame fits exactly and provides the full pan and tilt function. The aluminum frame is coated with a rust resistant paint that will provide protection against light rain, which this project is expected to experience [3]. The housing unit is 3D printed from ABS plastic in the texas instruments innovation lab. The plastic body means that the housing unit is lightweight but sturdy. The 3D printing aspect allows us to make a custom case that includes mounting posts for the panning servo and a tight fit that will prevent rainwater from getting underneath the cover.

The control systems' software is implemented with NodeJS, Electron, and Python. The physical control is implemented with a wireless Xbox core controller.

NodeJS provides asynchronous event handlers, interprocess communication, and high quality documented libraries available to use in the javascript language outside of the web browser. At the core of the control system NodeJS is used to interconnect the software of the media capture system and the motor system asynchronously.

To create the graphical user interface of the control system, Electron, a software framework was used to design cross-platform graphical user interfaces using only Javascript, HTML, and CSS.

The physical controller used to relay user input to the control system was the Xbox core controller. The core controller can be connected with either USB-C or the bluetooth protocol interchangeably. After pairing the controller can be probed for input using the Python wrapped xinput library, a Windows API that allows applications to receive the output of input devices.

IV. Implementation of the Media Capture Device

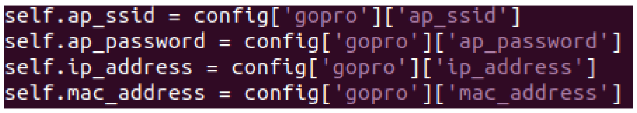
The GoPro Hero 7 publicly exposes a wireless access point with its 802.11n-capable radio transmitter that can be associated to by an arbitrary peer device in ad-hoc mode. The transmission is protected by WPA2 cryptographic standards in order to avoid malicious interference by third parties where the device may be used in a public environment. The wireless access point SSID and password key are generated automatically by the GoPro device, and can be accessed digitally using its touchscreen interface. When a peer associates with this wireless access point, the GoPro awaits HTTP GET requests with a crafted URL in order to expose its configuration settings and controls over the wireless interface. The IP address of the GoPro is preset to 10.5.5.9. The embedded HTTP server software responds on port 80 using the UDP protocol to encapsulate the data payload. In order to maintain the connection to the device, the GoPro expects a periodic "keep-alive" datagram to arrive on UDP port 8554 every 3 seconds.

The HTTP URL structure that GET requests are applied to is http://10.5.5.9/gp/gpControl/, followed by unique directory parameters that determine the type of command that is issued, and the associated arguments. We use the Python Requests library and a Python thread responsible for transmission in order to open the request and transmit the GET payload. The GoPro's status can be read by sending a GET to http://10.5.5.9/gp/status. This indicates whether a command executed successfully. We define the commands using a dictionary mapping to the unique URL associated with the command on the GoPro. This command dictionary program acts as a driver between the Control System and the Media Capture device. The API is text-based pipes, so the Control System gets a file handle for the standard input of the GoPro driver process. By writing the names of the commands contained by the command dictionary on the standard input of the process, the Control System can manage the GoPro with a large degree of abstraction away from the details of the GoPro device.

The Python driver program identified as GPC, for "GoPro Control", is organized as follows.

A. The GoPro Class

The GoPro is represented as a class GoPro. The class does not provide methods but instead logically represents the state and configuration settings of the device. The fields include the Access Point (AP) SSID 'ap_ssid", the AP password 'ap_password', the IP address 'ip_address', the link-level address "mac_address", and the keep-alive period 'keepalive_period'. These configurations settings are read from the file "gpc.conf" in the working directory of the gpc.py application using Python's configParser library.

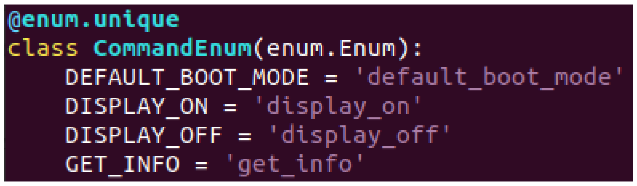


```
self.ap_ssid = config['gopro']['ap_ssid']
self.ap_password = config['gopro']['ap_password']
self.ip_address = config['gopro']['ip_address']
self.mac_address = config['gopro']['mac_address']
```

Figure 1:GoPro class instance member initialization

B. The Command Classes

The commands are logically enumerated and associated with a command string in the class CommandEnum as unique elements.



```
@enum.unique
class CommandEnum(enum.Enum):
    DEFAULT_BOOT_MODE = 'default_boot_mode'
    DISPLAY_ON = 'display_on'
    DISPLAY_OFF = 'display_off'
    GET_INFO = 'get_info'
```

Figure 2: Class definition and example elements

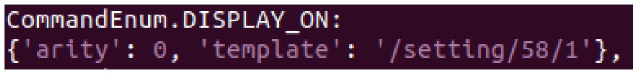
The commands are then defined in terms of their arity, the HTTP GET URL template that manages the functionality of the command, special command mappings, and a field that indicates whether we want a result returned from the GoPro.

The arity of the command, under the dictionary key 'arity', describes the number of arguments required by the command; if the command issued does not match the arity in the definition, the command is rejected and does not execute in order to avoid unintended side effects on the GoPro.

The template of the command, under the dictionary key 'template', is the tail of the control URL http://10.5.5.9/gp/gpControl, and when appended to this control URL, with the appropriate arguments formatted into position, and a GET request applied in the described fashion, the GoPro command will be carried out on the device.

The mapping of the command, under the key 'template', provides special transformations of the command string input from a user-friendly format to the raw format expected by the GoPro. An example of this is the DEFAULT_BOOT_MODE command. The GoPro provides media boot modes of video, photo, and multishot, but expects '0' to represent video, '1' for photo mode, and '2' for multishot. We create a mapping with an embedded dictionary so that a command can be written "default_boot_mode video" that automatically transforms this into an equivalent "default_boot_mode 1".

Finally, we determined whether the driver returns a result using the 'want_result' key in the dictionary. Many of the GoPro commands emit results that are unnecessary for controlling the GoPro, and instead the GET_STATUS command is used to determine whether the GoPro device is in an error state, or ready to begin filming.



```
CommandEnum.DISPLAY_ON:
{'arity': 0, 'template': '/setting/58/1'},
```

Figure 3:

C. The Message Class

Now that a Command has been defined, we wrap the Command in a Message class instance that holds the command and knows how to transmit itself to a GoPro. The Message class uses the template of the command in its private _build_url function to construct the final URL that constitutes the literal GET command that the GoPro understands directly. The transmission is performed using the Python Requests library in Message's send_to(gopro) method. We take an instance of the GoPro class and apply this method on it. The Message.send_to method is capable of implementing synonym commands that are not directly available on the GoPro, such as GET_BATTERY_LEVEL that is implemented in terms of an invocation of the GET_STATUS command, and Message.send_to is also an ideal region of code to implement pseudocommands, such as sending a wake-on-lan datagram that does not directly utilize the _build_url template paradigm. After the GET is transmitted to the GoPro, the return result can either be saved if the "want_result" field in the Command definition is set to true, or otherwise discarded.

The Gopro, Command, and Message classes are tied together in the main function. The GoPro is instantialized, reading the file from disk and then parsing it with Python's configParser. A daemon thread is constructed that is responsible for sending keep-alive packets to the GoPro's IP address until the termination of the program. Since the thread is operating in Python's daemon mode, when the main thread terminates, the daemon thread will not persist, and will terminate immediately. When this preceding initialization is complete, the GPC driver program enters an eternal loop reading from standard input. Text that is written to its standard input is stripped of leading and trailing whitespace, and the class method Message.from_text() is invoked in order to transform the input from raw text into a Command and arguments. If the text presented to the program is formatted correctly, the text will parse into a Message, and the Message is then sent to the GoPro. Desired replies are captured and printed to standard out for processing by an external program.

V. Implementation of the Motor System

The main function of the motor system is to move the camera so that the operator can follow the action. This requires the system to take control data as input but no output data to the controller. The Raspberry Pi takes the control data from the laptop over wifi and sends it to the Sparkfun servo HAT which signals the servo to move into the necessary position. This system is the most hardware oriented of the project but it is very simple. The main consideration for the system components are the power consumption and response time. The power consumption is important because we want the devices to operate on the same power level. If we use a 12 volt brushless motor on a board that uses 3.3 volts, the two devices would need separate power sources. If that were the case, we would fail to achieve our goal of reducing the number of cables running to the device. The response time is important because we are live streaming the video, so a slow response in the motor system compounds down the pipeline to affect the stream quality.

A. Power

The two control boards, the servo HAT and the Pi, are connected through the GPIO pins . This gives access to both the 5 volt line for powering the servos and the 3.3 volt line for powering the board. It also means that we can run power to either device and power the whole motor system. This allows more flexibility when it comes to the type of power cable a customer can use. Since the board uses such a low voltage, we cannot use a standard 12 volt AC power supply, otherwise we would burn out the control boards as we found out by tragic error. We instead used a 5 volt supply that was recommended by the official Raspberry Pi website. We also have servos that operate on a similar voltage, so we can use the same power supply and avoid using two different power cables. Also, the power supply comes with either a barrel end or a micro-USB end. While the micro-USB version would allow us to easily plug in the cable without a barrel-to-USB adapter, the barrel end would be better because we can use barrel cable extenders to meet the 25 foot height of the tripod. However, for the purposes of our demonstration, we are using a standard tripod, so we went with the micro-USB power adapter instead. The final result is one power cable for the motor system.

B. WiFi

The wireless communication is what helps us reduce the number of wires running to the project. To get the controller and servos talking we use a wifi connection from the Raspberry Pi. The Pi uses the BCM 43438 to connect to a network so that it may talk to other devices. In this instance, we are using the wifi network from the GoPro Hero 7 since it also connects to our laptop. The GoPro network also gives us a stable wifi network so that we don't have to connect to a new source everytime we move locations. Usually, we would write a wpa_supplicant.conf file with the name of the network and the password, but since we use the same network we only need to do it once and the Pi will use the same credentials stored in the wpa_supplicant file. This also provides a stable network address so that we don't have to worry about finding the Pi IP address on excessively large networks such as the UCF campus wifi.

This aspect of the motor system affects response time the most since the speed of the wifi network dictates the data transmission. We are using a smaller network with only our 3 systems so the wifi speed is relatively unaffected, but the unusual setup of the camera as the source of the wifi makes an initial connection slow. For the motor system, there is no significant reduction in speed, so response time remains unaffected.

C. Servo Control

The servo 3 wire connection connects directly to the Sparkfun servo HAT through one of the 16 servo channels. Since we only used 2 servos, pan and tilt, we only needed 2 channels. The servo HAT is connected to the Pi through the GPIO pins to share power and data easily. The Pi runs the python code that controls the servos and the servo HAT uses the pulse width modulator to regulate the control data. Even though the servos can be controlled from the GPIO pins on the Pi, the clock can be unreliable and will affect the performance. So we use an external pulse width modulator to regulate the signals going to the servos. Thus we use the servo HAT.

Servos worked best for this project because of their precision. Servos have high precision because of the signal wire which tells the motor what position to turn the rotor. Even if you try to manually move the rotor, the servo will try to correct the position until the force against the motor is greater than its stall torque. Precision is key when controlling the speed and position of the camera from a distance in order to follow the action being recorded.

In code, the sparkfun servo python library has all the functions needed to control the servos. The functions are wrapped in the qwiic_pca9685 module for this servo HAT. It includes a move_servo_position() function that takes the channel on the board and the position of the rotor in degrees as input and moves the rotor on the corresponding channel accordingly and a get_servo_position() function for storing a position for later use. While the library is great on the Pi, it really needs to communicate with the controller.

Voltage Range	4.8V - 6.0V
No-Load Speed (4.8V)	0.21sec/60°
No-Load Speed (6.0V)	0.16sec/60°
Stall Torque (4.8V)	45.82 oz-in. (3.3kg.cm)
Stall Torque (6.0V)	56.93 oz-in. (4.1kg.cm)
Pulse Amplitude	3-5V
Current Drain - idle (4.8V)	8mA
Current Drain - idle (6.0V)	8.8mA
Current Drain - no-load (4.8V)	150mA
Current Drain - no-load (6V)	180mA
Max Rotation	195°
Travel per μs	0.0975°/μsec
Max PWM Signal Range	500-2500μsec

Table 1: Spec sheet for Hitec HS-422 servo [1]

The response time of the servos is very quick. For a continuous sweep, the rest time between a change in degrees is 0.01 seconds which is consistent with the specifications on the data sheet. However, when testing moving the servo from 0 degrees to 180 degrees, there were some issues. The servo jittered while turning to 180 degrees and didn’t complete the turn with the same rest time. This was because the servo did not have enough time to make the turn, rest, and then make the next turn. This is consistent with the specification in table 1 that it takes 0.2 seconds to turn 60 degrees, which means that it needs a minimum rest time of 0.6 seconds. In a test, I looped the 0-to-180 code while reducing the rest time each loop. I found that the servo would start to jitter at the 0.6 second rest time. However, this response time doesn’t affect the system that much because the controller will move the servo in small increments of 2 or 3 degrees. Using the 0.2sec/60 degree specification, that would mean it takes 0.003 seconds to move a degree, so as long as the rest time is larger than that, the servo shouldn’t jitter. This also means that we can have a sampling rate of 3 milliseconds, however that would be faster than we need. The slew rate can be improved by using a 6 volt power source, but the reduction in response time is not sufficient enough to warrant that.

D. Housing unit

The physical components of the system include the housing unit. We used a 3D printed custom unit based on designs from similar products and projects.

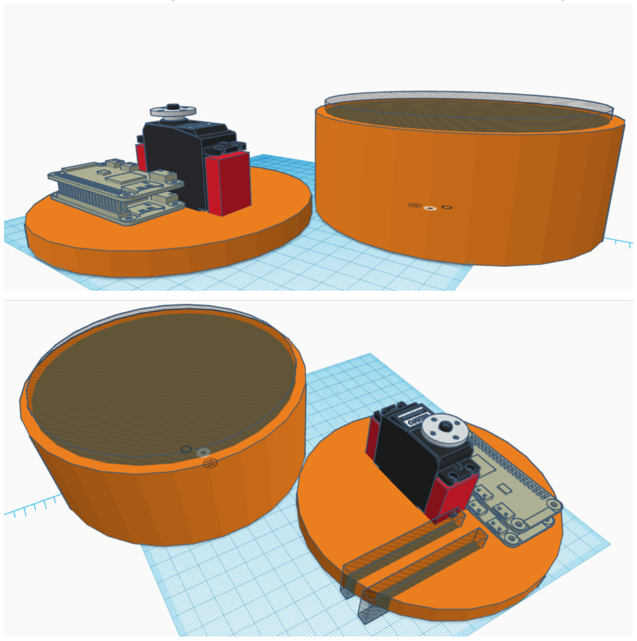


Figure 4: 3D print files for housing unit on TinkerCad

The design needed to accommodate a tripod mount, have a servo at the center, and be large enough to house the control boards while also being able to spin. The last consideration is another reason why we chose to use the Pi Zero. It's the smallest of all the communication board options. The alternative would be an excessively large base for the boards and servo to sit side-by-side or an excessively tall base so that the servo can sit on top of the control boards. The design also considers power cables. We chose to make 2 slots instead of 2 holes because of the tripod mount. Since the mount is screwed in under the base of the housing unit, it covers part of the base. This mount can also be a different size depending on the tripod. We found that we would have to make the base larger so that we could have access through the bottom. So the slots can accommodate a tripod mount of any size smaller than the 13 cm base. We considered printing the mount shape, a square platform, directly onto the base, but that would cause the 3D printer to make an understructure for the empty space under the base for support that would ruin the design. The design also has mounting posts for the panning servo so that it can remain centered and secure.

The housing unit is weather proofed against light rain. The cover extends over the base so that no water gets on the electronics and the hole in top for the tilting servo connection is patched with water resistant putty to create a seal. However, the cover does not completely go over the base because the print tolerance adds an extra millimeter or two. So the cover sits closely above the base, but still covers the electronics. We would fix this error by making the cover opening larger, but 3D printing in the texas instruments lab takes too long and we couldn't remedy the issue in time.

VI. Implementation of the Control System

The Xbox core controller is connected to the workstation computer that hosts the control software. The Control System deploys a graphical user interface that is used in tangent with the controller for passing user input into the system. On activation the control system spawns both the media camera system and the motor system using interprocess communication. It then sets event listeners that deploy functions as messages from the GUI or controller are received. The xinput library is used to parse the incoming controller input into a dictionary that contains the status and values of all buttons and analog triggers. The button triggers are mapped to either the media capture device or the motor's servos. Before passing along the commands the magnitude of the analog trigger is scaled down to allow precise control over the corresponding system. The messages received are then run through the corresponding algorithms to handle camera and motor controlling.

A. The Graphical User-Interface

The graphical user interface is quickly made with the usage of the electron framework. A simple create window function is passed that creates a window instance with the dimensions and design preferences set as arguments. Event listeners for communicating with the front end are also set for in app functionality, from here commands to the other systems can also be set using inter-processing communication directly from the user interface.

B. Spawning other processes

After the GUI is set up other processes are ready to be spawned and managed by the control system. In order to spawn another process the NodeJS module ‘child_process’ is imported and the spawn() used for each correlated script. The spawn function takes three inputs: a command to run, a list of string arguments, and an object of options. As the scripts for all the systems are in python the initial command for spawn used is ‘python3’ with corresponding argument list passing the location of the file. The exception being the spawn of the motor system that is hosted on the Raspberry Pi, an ssh into the pi is used before passing the same commands in the arguments list.

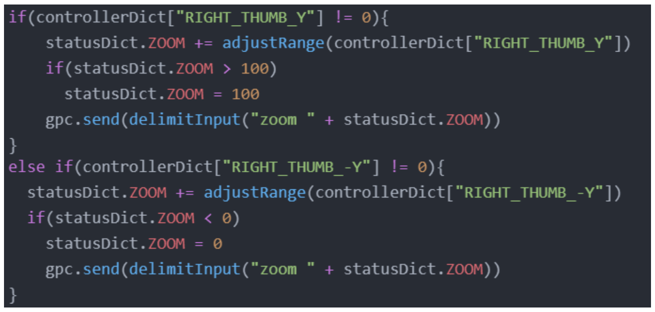
C. Asynchronous Events

The spawned subprocesses all run asynchronously while pulling in and pushing out data, to handle this state of events an event handling system is implemented. On one side there is an emitter that when reached sends out a signal that a listener can capture before executing the desired commands. In NodeJs the child_process module attaches listeners to the subprocesses so the stdout,stderr, and exit codes of the corresponding spawns are efficiently monitored.

The xbox controller’s event listener receives input from the subprocess that contains a dictionary containing the changed state of any pressed buttons and the value of any activated analog trigger. After the dictionary is received, parsing it into a JSON object is attempted and if successful a button ‘input’ event is emitted. He ‘input’ event is captured by a custom button event handler that parses the JSON object and outputs the data over to each individual subprocess as pre-mapped user controls.

D. Command Transfer

The raw input received from the xbox controller is parsed and sent over to the sub processes as camera control commands such as zooming the camera, toggling the camera's display, the activation or deactivation of recording, and the starting of the live stream. Translation of the zoom command works by first detecting the controller’s input on the right thumb stick. The analog joystick has a maximum range from -32768 to 32767, which is adjusted to a range of -1 to 1. The adjustment of range allows the user to zoom the camera at a fine speed. In order to start the livestream the d-pad up button is pressed, this sends a start stream command over to the GoPro which starts udp transfer of video at the host/port address ’udp://10.5.5.100:8554’. The motor subprocess receives a command that signals the two servos to move either clockwise or counterclockwise respectively.



```
if(controllerDict["RIGHT_THUMB_Y"] != 0){
    statusDict.ZOOM += adjustRange(controllerDict["RIGHT_THUMB_Y"])
    if(statusDict.ZOOM > 100)
      statusDict.ZOOM = 100
    gpc.send(delimitInput("zoom " + statusDict.ZOOM))
}
else if(controllerDict["RIGHT_THUMB_-Y"] != 0){
  statusDict.ZOOM += adjustRange(controllerDict["RIGHT_THUMB_-Y"])
  if(statusDict.ZOOM < 0)
    statusDict.ZOOM = 0
    gpc.send(delimitInput("zoom " + statusDict.ZOOM))
}
```

Figure 5: Zoom command logic and send off

Button/Analog trigger	**Usage**
START	Start Recording
BACK	Stop Recording
LEFT THUMB	Display On
RIGHT THUMB	Zoom in/out
DPAD UP	Start Stream
LEFT JOYSTICK	Pan and tilt
RIGHT JOYSTICK	Zoom in/out

Table 2: A table mapping the controllers commands

VII. Project Results

The project results are broken down into its key constituent components.

A. Media Capture Device Results

We judged the result of the media capture device in terms of the bitrate of the transmission of the preview media at a distance of 30 feet. A command is issued for a high ceiling on the bitrate, STREAM_BITRATE invoked with a bitrate of 1M bit per second. Using the ffmpeg media library, the bitrate is measured and performed nominally as determined in the research phase of the project.

B. Motor System Results

We judged the result of the motor system in terms of the slew rate with the GoPro hardware attached. We intended for the motors to move at a speed sufficient to capture rapid motion from a wide angle. The pan-tilt speed of the motor was sufficient to capture action sports.

C. Control System Results

We judged the result of the control system in terms of the responsiveness to controller input. When a button is pressed on the controller, the delay in transmitting the controller command to the GoPro is consistently less than one second, and meets our criteria for responsiveness.

VIII. Conclusion

Since the Project Results produced a satisfactory mark in each of its key components, we believe the design of the PTZ Tower Camera is acceptable and ready for deployment

Acknowledgement

The authors would like to thank Dr. Samuel Ritchie for guiding us during the process of implementing this design.

The Engineers

Michael Serignese is a graduating Computer Engineering student who plans to begin his career by transforming his internship at Lockheed Martin into a full-time job. He plans to develop software whether that be high-level software architecture or embedded systems design.

Andre Samaroo is a graduating Computer Engineering student who plans to work with small engineering companies in different roles to discover the right field within computer engineering to continue studying in graduate school.

Chisom Ikejiani is a graduating Computer Engineering student who plans to pursue a career in the specialized field of embedded systems design or fintech, two fields he is very interested in.

References

[1] *Servocity*, "HS-422 Servo-Clockwise (stock)-Stock Rotation," (n.d.), https://www.servocity.com/hs-422-servo/ , Accessed 20 July, 2021

[2] *Sparkfun*, "SparkFun Servo pHAT for Raspberry Pi," (n.d.), https://www.sparkfun.com/products/15316 , Accessed 12 March 2021

[3] *RobotShop*, "Lynxmotion Pan and Tilt Kit / Aluminium," (n.d.), https://www.robotshop.com/en/lynxmotion-pan-and-tilt-kit-aluminium2.html , Accessed 12 March 2021

[4] *All About Circuits*. "The Raspberry Pi Zero W Adds Wireless Capabilities with Wi-Fi and Bluetooth," 28 February 2017, https://www.allaboutcircuits.com/news/the-raspberry-pi-zero-w-adds-wireless-capabilities-with-wi-fi-and-bluetooth/ , Accessed 16 April 2021.

[5] *GoPro Hero API*. "GoPro Documentation", 28 April, 2021. https://goprohero.readthedocs.io/en/latest/API/.