Mothership: Python for Home Security

Prepared For:

Dr. Darcy Benoit

COMP 4983

Acadia University

Prepared By:

Brent Rushton

COMP 4983

Acadia University

Table of Contents

Introduction

The Capstone Project that is a part of the Computer Science degree at Acadia University is a great opportunity for students to be able to develop a piece of technology that is related to their chosen degree path. The project affords students the ability to control the entirety of the process, from what technologies/methods are used in the development all the way down to what color the GUI (Graphical User Interface) is. For some, the Capstone Project may lead into future employment once leaving Acadia, hopefully with their undergraduate degree in hand. For others, myself included, the Capstone Project gave the freedom to learn an entirely new technology stack in order to build a project that was in their mind for some time. It was out of that freedom that Mothership was born.

Mothership, in its current state, is a security camera implementation aimed to provide its users with features that are standard for most consumer security camera interfaces. The inspiration for this system came about when my in-laws were saying they were going to invest in a camera system for the cottage we are currently staying in while attending Acadia, something that can be both pricey, and can lock you into an ecosystem depending on the type of cameras and who manufactures them. It was then that I remembered seeing an open source IP camera app for Android phones and suggested that I try creating my own camera system to use which would be a lot cheaper, and support a wider array of cameras, with the aim of supporting wireless video transmission via Android phones.

As a result, Mothership is purpose built for filling the needs of my particular situation, knowing that I may not have another opportunity to spend months learning a tech stack for a personal project. However, despite Mothership being built for my own use, I believe it may be of some utility to those looking for a free way to monitor their property with hardware that they may already have. Additionally, with the use of virtual hardware cameras, a user could monitor any video source inside of Mothership.

Proposed Implementation

My proposition for Mothership was to have the following features implemented by the end of the term:

- Real-Time Monitoring
    - Mothership will provide a real-time video stream from your camera, giving you an instant view of your camera location.
- Motion Detection
    - Analyze video frames in real-time to detect movement with the camera's field of view. Mothership will trigger various events such as snapping pictures, recording video, or sending alerts to a user.

- Event Notifications
    - Receive email alerts, SMS messages, or even push notifications when designated triggers are activated. An example of such a trigger is when motion is detected in the camera view.
- Video Recording
    - Captures videos when motion is detected. These clips are automatically saved to a storage solution. Ideally, Video recording would be triggered by motion, time, and/or schedule.
- Multi-Camera Support
    - Will be able to handle a wide array of cameras simultaneously. Exact numbers on the number of supported cameras will be determined once further development is completed. For a bare minimum, Mothership aims to provide 4 camera feeds simultaneously.
- Secure Access Control

  - If Mothership development progresses into an app that is remotely accessible, access to accounts will follow modern authentication practices.

- Compatibility

  - Mothership will aim to be compatible with major operating systems eventually, but will begin as a python file. A vast number of cameras are planned to be supported, ip cameras and USB will be the focus although remote access may only be available to cameras that are connected via an internet facing IP/Domain.

- A Cohesive User Interface

  - This feature is one that will develop with time as other features get implemented. The goal is that the user will be able to implement and manage a security camera array with relative ease. Documentation will be provided in some form to achieve this.

I had also outlined which features I would be implementing when, outlined as a minimal viable product, a baseline build, and a stretch build. They are outlined in the table below:

Build Version	Features Included
Minimal Viable Product	Real Time Monitoring, Video Recording, Motion Detection. Multiple Camera Support, Barebones UI. Notes: Features work, but may be buggy or not integrated well. Storage systems may not be automatic, for example.
Baseline Build	Everything in the Minimal Viable Product as well as: Event Notifications, Compatibility, Passable UI, Facial Recognition Notes: Features are well designed and have thought-out implementations. This is starting to show utility as a legitimate surveillance system. Compiles and is tested with at least one desktop OS.
Stretch Build	Everything in Baseline build as well as: Expansive UI, Secure Access to cloud storage, controlling other IoT devices, camera zoom/tilt/two way audio controls if supported, user authentication Notes: This could be a production level app. Works natively on all platforms and provides other home utilities than cameras.

The proposition was to have the baseline build completed and to be working on the stretch build by the end of the term. This was to ensure that a cohesive piece of software would be submitted and would resemble software that people would actually use. The builds were separated into what I thought would be good benchmarks for the different iterations of Mothership. The minimal viable product was designed to be a demonstration that the features do in fact work, with a minimal GUI and would be a “worst-case scenario” build to submit in order to at least get some sort of grade. The baseline build was my goal for the end of the term, with an emphasis on fully implementing the features outlined in the minimal viable product as well as a few more features introduced like event notifications and facial detection. The stretch build acted as an ideal scenario where I could flesh out the baseline build and add things like remote access and cloud storage. The idea was that both the minimal viable product and the baseline build would both be locally managed, instead of having a cloud storage implementation and logins like the stretch build.

The technology stack that I initially proposed was the following:

- Python and common libraries such as Numpy
- OpenCV: Open source computer vision and machine learning software library.
- Kivy: For packaging python projects into native executables for a variety of operating systems. Kivy can make packages that compile with any dependencies for Windows, MacOS, Linux, Ios, and Android.

- Ngrok: Used to expose local development environments to the internet using the ngrok.com domain. Basically a tunneling service that may be useful for port forwarding the ip cameras for remote access.

- IP/USB cameras

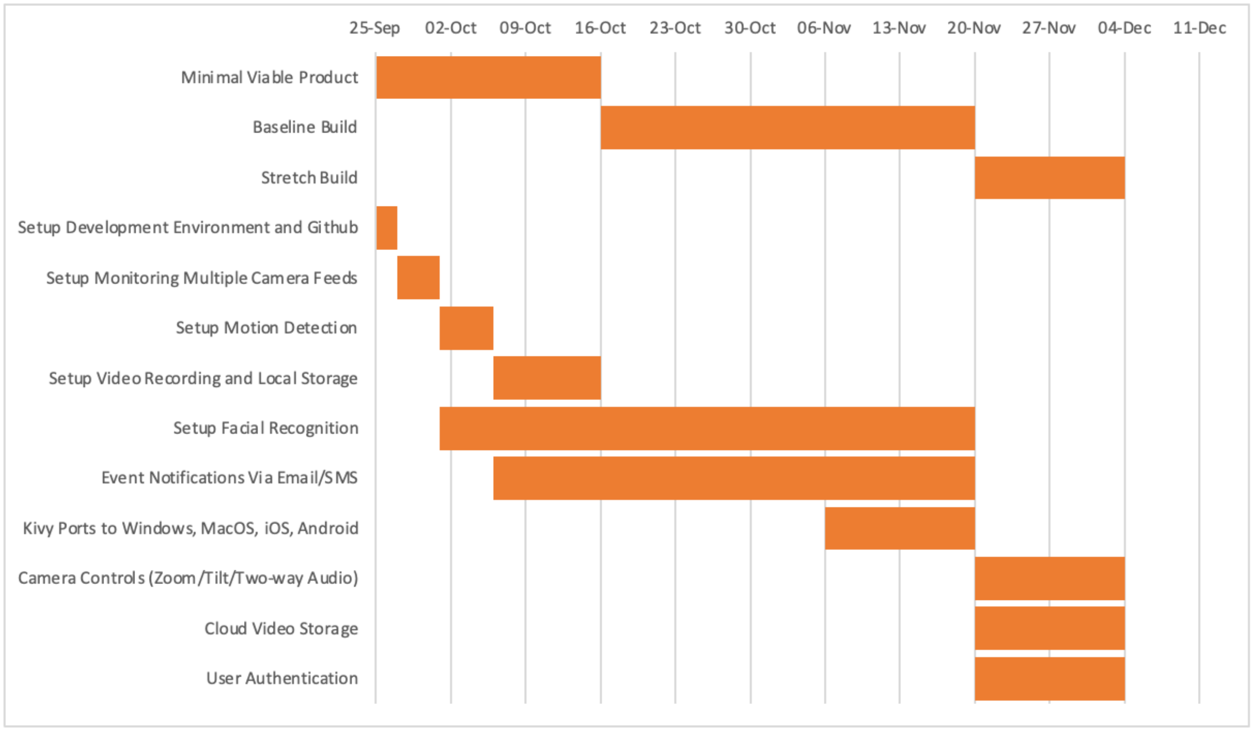
- Visual Studio Code: My IDE of choice

Additionally, there were some other technologies that I outlined in my initial proposal:

- Some sort of DBMS might be needed, but I might design the application around local instances connecting to the cameras; meaning that a local instance must be running on a machine somewhere if you want 24/7 security. For this I usually lean towards something that stores data in JSON like Firebase or MongoDB, but may have to structure data in SQL depending on how complex the schema gets.

- Cloud Computing platforms

Furthermore, my initial proposal included a Gantt chart that outlined the target dates for the completion of the various tasks associated with the development of Mothership. Below is the Gantt chart:



This Gantt chart was referenced heavily and followed pretty accurately until the first week of november, but more on that in the next section.

Implementation

Mothership, in its current state, leverages the following technologies in order to achieve all of the features outlined in the minimal viable product as well as the baseline build (aside from motion detection as of the writing of this report):

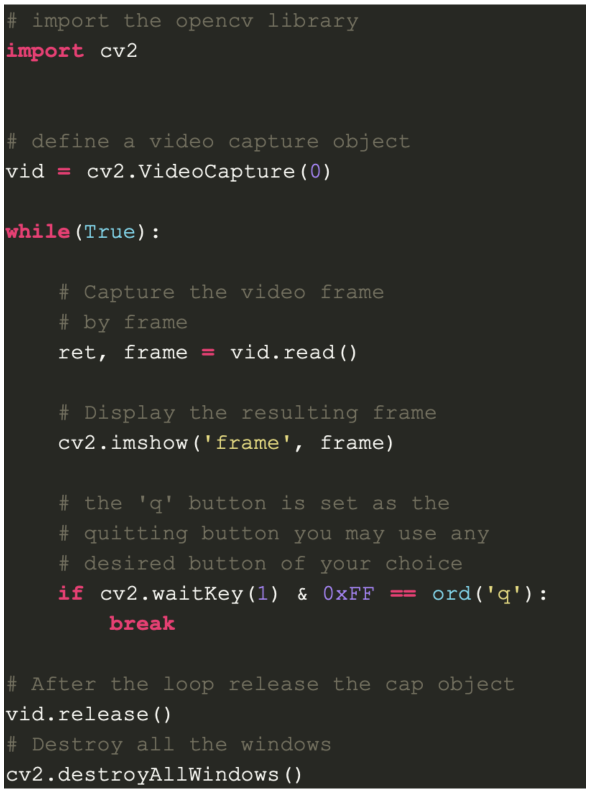
- Python
- OpenCV

- IP Webcam by Pavel Khlebovich (https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en_CA&gl=US&pli=1)
- USB Webcams
- The following Python Libraries
    - Pillow
    - Tkinter
    - Threading
    - Time
    - Smtplib
    - Queue

Looking back on my proposed implementation, it was clear almost immediately that the stretch goals were not a realistic thing to achieve given the timeframe of the project, and my limited knowledge of cloud computing practices. I proposed them initially without doing much research into the technologies that I would be using for this project. I knew that python could be a viable language for producing webapps but I didn't know to what extent the complexity of the python script could be before implementing Flask or Django (python web frameworks). It turns out to be pretty near impossible to do in an eloquent manner once my GUI was built in Tkinter and especially tricky once threading was implemented.

When I began work on Mothership, I knew I would have to use some sort of library to open a camera source at the very least. To do this, I did some research on ways to go about retrieving camera feeds from a system and by and large the most popular suggestion was OpenCV. OpenCV is an open source computer vision and machine learning software library that employs the use of over 2500 optimized algorithms. This algorithm library includes both legacy and state-of-the-art vision and machine learning algorithms that can be used to detect and recognize faces, identify objects, track camera movements, extract 3D models, produce 3D point clouds from stereo cameras and more. Companies that employ the use of OpenCV include Google, Microsoft, Intel, Honda, Toyota, and more. OpenCV has interfaces for C++, Python, Java, and MATLAB and supports all major desktop operating systems as well as Android (OpenCV).

Immediately I began experimenting with OpenCV and performing simple tasks such as opening a camera feed. OpenCV is quite versatile in how it opens video feeds, it can open hardware cameras simply by entering the camera index into a VideoCapture object and performing some logic in a while loop for capturing consecutive frames from the VideoCapture object. The following code snippet from geeksforgeeks.com shows a simple method for capturing a video source from a hardware camera at index 0:

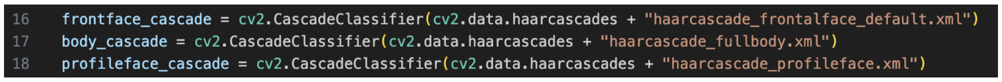


```python
# import the opencv library
import cv2


# define a video capture object
vid = cv2.VideoCapture(0)

while(True):

    # Capture the video frame
    # by frame
    ret, frame = vid.read()

    # Display the resulting frame
    cv2.imshow('frame', frame)

    # the 'q' button is set as the
    # quitting button you may use any
    # desired button of your choice
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# After the loop release the cap object
vid.release()
# Destroy all the windows
cv2.destroyAllWindows()
```

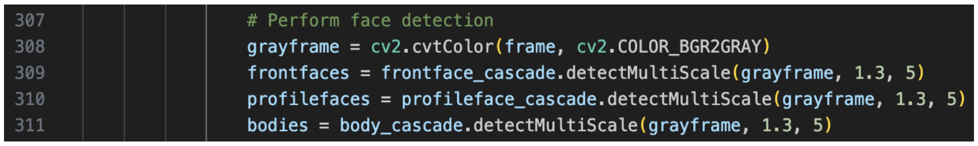
(geeksforgeeks)

As one can see, the VideoCapture function is called to create the “vid” object and a while loop is made to then use the built in imshow function to show each frame in succession in a window. This served as a great starting point for me and then eventually led into the discovery of haar cascade classifiers that are built into the OpenCV libraries.

Haar cascade classifiers are a built-in method for object detection in OpenCV. Haar cascade classifiers are a machine learning based approach to object detection where a cascade function is trained on both positive and negative images. It is trivial to set up a single camera with the haar cascade classifiers for frontal face recognition, side profile face recognition, and body recognition, but the complexity came with implementing it for multiple cameras simultaneously inside of the GUI that I had built. The following snippets of code from Mothership's source code are the importing of the Haar cascade classifier data sets, and the initialization of the cascade classifier itself inside the thread that runs a given camera:

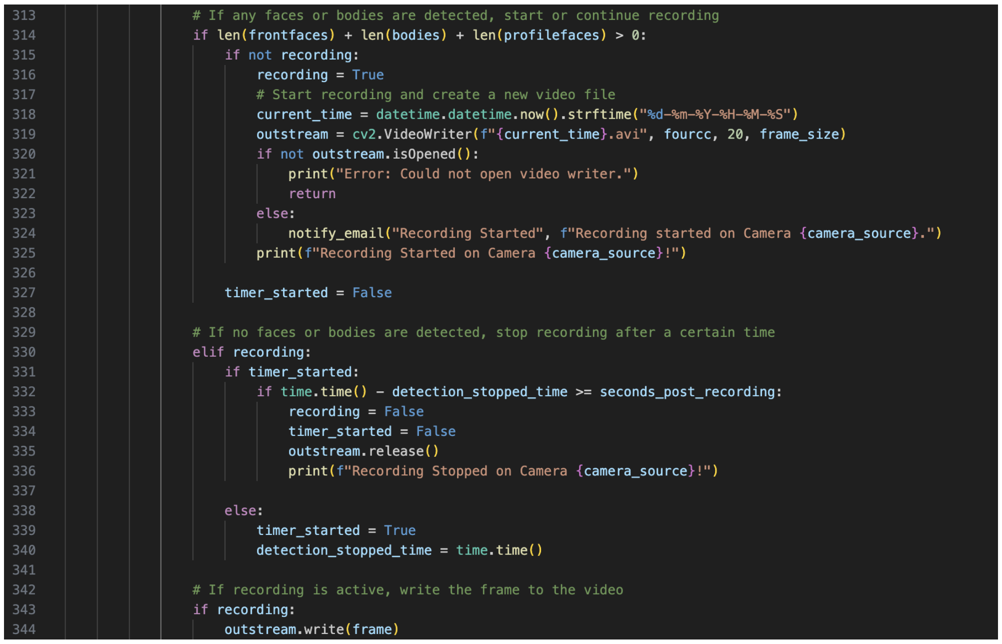


```
16  frontface_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")
17  body_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_fullbody.xml")
18  profileface_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_profileface.xml")
```



```
307                 # Perform face detection
308                 grayframe = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
309                 frontfaces = frontface_cascade.detectMultiScale(grayframe, 1.3, 5)
310                 profilefaces = profileface_cascade.detectMultiScale(grayframe, 1.3, 5)
311                 bodies = body_cascade.detectMultiScale(grayframe, 1.3, 5)
```

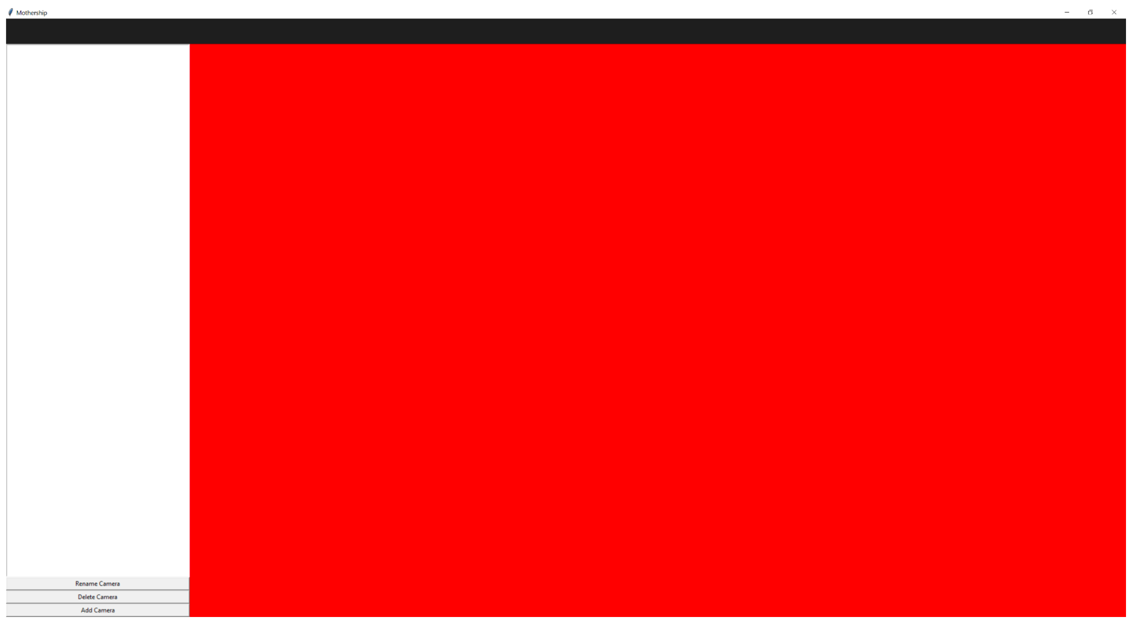
The lines from 16-18 import the cascade classifier data sets from the OpenCV library globally to be called on in lines 307-311 where the current frame is converted to grayscale before the detectMultiScale function performs the detections for front faces, profile faces, and bodies on the frame. What is the point of doing these calculations in Mothership? Well, Mothership uses these detections to see if there is anyone appearing in the view of a camera so it knows when to start recording a video for the user to watch back. Fortunately, OpenCV also includes a VideoWriter function that supports multiple video encoders that are determined by a four character code that discerns which encoder will be used. To name the files, I used a pretty standard method using the time library in order to name the file in the following format: Day-Month-Year-Hour-Minute-Second.avi. The following is my implementation and logic for recording a video:



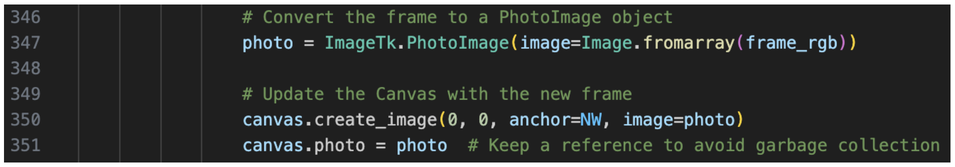
```python
                # If any faces or bodies are detected, start or continue recording
                if len(frontfaces) + len(bodies) + len(profilefaces) > 0:
                    if not recording:
                        recording = True
                        # Start recording and create a new video file
                        current_time = datetime.datetime.now().strftime("%d-%m-%Y-%H-%M-%S")
                        outstream = cv2.VideoWriter(f"{current_time}.avi", fourcc, 20, frame_size)
                        if not outstream.isOpened():
                            print("Error: Could not open video writer.")
                            return
                        else:
                            notify_email("Recording Started", f"Recording started on Camera {camera_source}.")
                        print(f"Recording Started on Camera {camera_source}!")

                    timer_started = False

                # If no faces or bodies are detected, stop recording after a certain time
                elif recording:
                    if timer_started:
                        if time.time() - detection_stopped_time >= seconds_post_recording:
                            recording = False
                            timer_started = False
                            outstream.release()
                            print(f"Recording Stopped on Camera {camera_source}!")

                    else:
                        timer_started = True
                        detection_stopped_time = time.time()

                # If recording is active, write the frame to the video
                if recording:
                    outstream.write(frame)
```

As soon as a face or a body is detected in a frame, recording is started and once all faces and bodies are out of frame, a timer for 10 seconds is started and if a face or body reappears in the frame, the timer is restarted and won't start the count again until all faces/bodies are gone from a frame again.

That logic is all well and good when dealing with single camera sources, but how would one deal with multiple cameras at once? This is the main premise that Mothership operates on. The way that I set out to do this was by creating a GUI that would house an interface for adding, deleting, and renaming cameras, as well as a way for users to input their email address for notifications as well as the 3x3 camera array itself. I chose a 3x3 array for the cameras simply because I thought a 2x2 was too small and I wanted an even square. The following is an early version of Mothership's GUI that would serve to be the bones of what it would become:



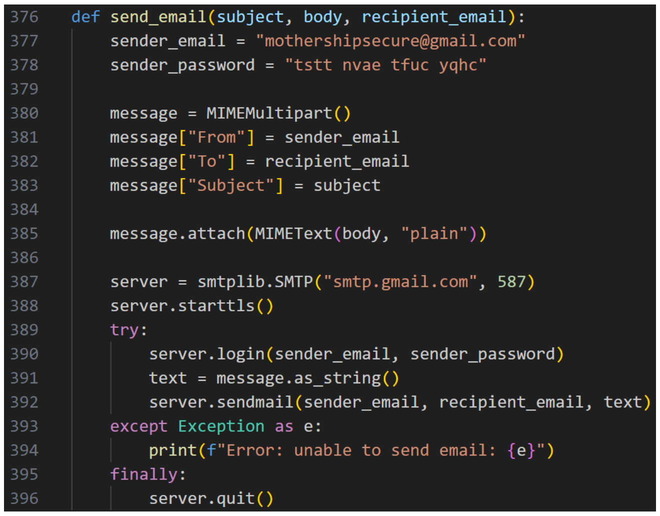
The white section on the left is a Tkinter ListBox component that houses the list of cameras in the array which is represented by the red Canvas component on the right. The ListBox also has 3 buttons for adding, deleting, and renaming cameras in the box, and as a result in the canvas as well. The red Canvas component is also divided into 9 "slots" in a 3x3 array that is using a python array to keep track of which slots are in use. The top bar is also where miscellaneous things such as the main Mothership logo and a user logo that will open an email address field when clicked. Most importantly for me however, was the use of the Pillow library's PhotoImage function to create an object holding a video frame which will then display it over the red canvas. The following code snippet shows that process:



```
                # Convert the frame to a PhotoImage object
                photo = ImageTk.PhotoImage(image=Image.fromarray(frame_rgb))

                # Update the Canvas with the new frame
                canvas.create_image(0, 0, anchor=NW, image=photo)
                canvas.photo = photo  # Keep a reference to avoid garbage collection
```

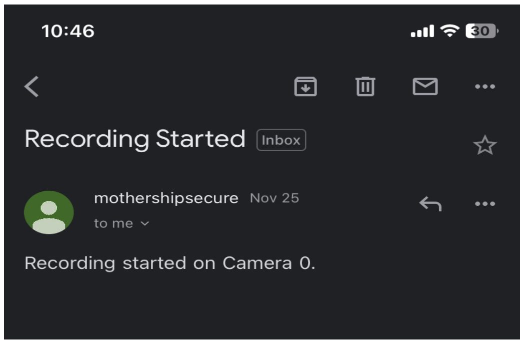
I felt it was important to do it in this manner as the PhotoImage object can resize the frame dynamically to fit any size that the canvas might be. This means that resizing the window will also resize all the cameras currently open in the array dynamically without a huge performance impact. Additionally, the GUI is updated on a separate thread than the detection/recording logic. This is so the GUI doesn’t get hung up when videos begin recording and stop recording. I ran into issues originally with the threading implementation in the sense that I had to restructure my code a bit so that the haar cascade classifier objects were separated into each camera’s respective thread as it would try to amalgamate the frame data of all the concurrent cameras that are being displayed on the camera grid.

Another feature that I knew I wanted to implement right away was a method of alerting the user. To achieve this, I used the smtplib python library which defines a SMTP (Simple Mail Transfer Protocol) client session that can send email to any listener that supports SMTP or ESMTP (Enhanced Simple Mail Transfer Protocol) which most major email clients support. The caveat for this method is that a sender email and a password must be provided in order to send the email successfully. For this, I created a gmail account for Mothership (mothershipsecure@gmail.com) and also generated an app password which is google’s method for obfuscating passwords when using them in codebases. Sending emails are relatively trivial using this method once the account was set up for it. I made a function in the script called “send_email” that receives arguments for subject, body, and recipient email which formats the email correctly and performs the logic for sending emails. To call this function, I also started a new thread in a “notify_email” function that will open when an email is to be sent, calls the send email function and closes the thread. I did it this way to avoid hitching in the GUI as well as avoiding any interference with the detection/recording logic. Below is the “send_email” function that shows the logic behind sending emails in Mothership:



```python
def send_email(subject, body, recipient_email):
    sender_email = "mothershipsecure@gmail.com"
    sender_password = "tstt nvae tfuc yqhc"

    message = MIMEMultipart()
    message["From"] = sender_email
    message["To"] = recipient_email
    message["Subject"] = subject

    message.attach(MIMEText(body, "plain"))

    server = smtplib.SMTP("smtp.gmail.com", 587)
    server.starttls()
    try:
        server.login(sender_email, sender_password)
        text = message.as_string()
        server.sendmail(sender_email, recipient_email, text)
    except Exception as e:
        print(f"Error: unable to send email: {e}")
    finally:
        server.quit()
```

And the resulting email:



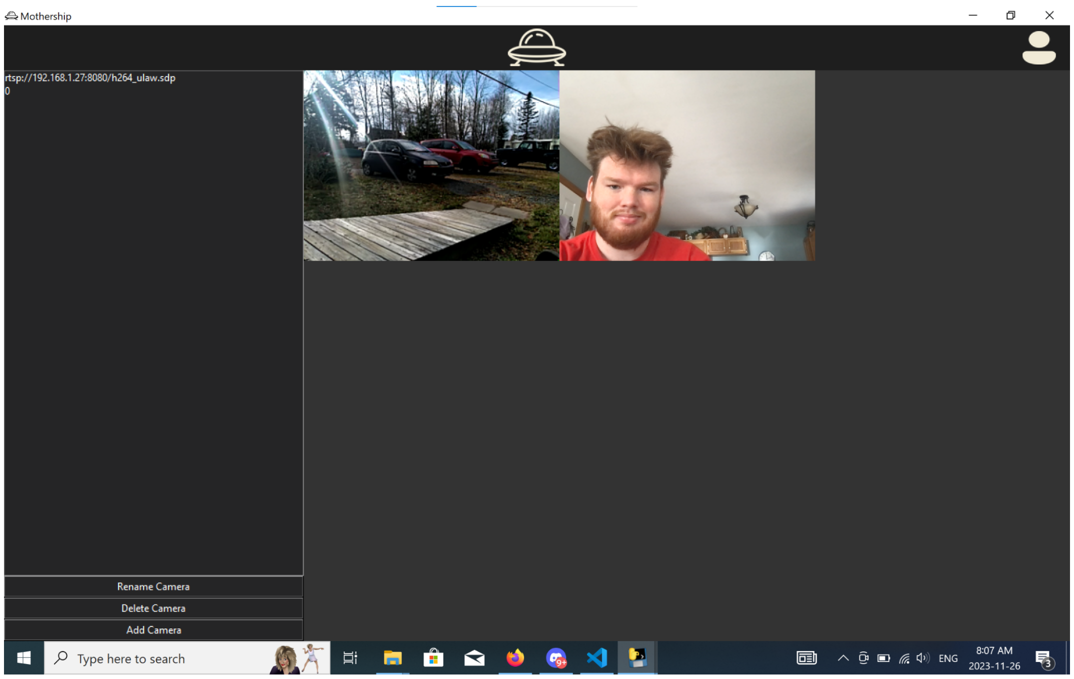
Recording Started

mothershipsecure Nov 25

to me

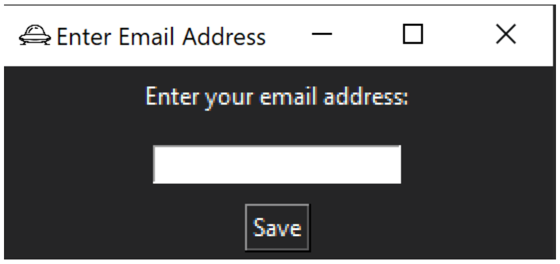
Recording started on Camera 0.

The last feature that I would like to make note of is the packaging of the script into a native windows executable file. To do this, I used a package called pyinstaller that bundles all the dependencies and a python compiler into a single executable that can be launched on a fresh install of windows. The reason for doing this in part was for ease of marking, there are a lot of things that would need to be installed otherwise and making an extensive read me file with instructions on what needs to be installed did not seem to be the most elegant solution when a marker may only just want to see the code running. Another reason for packing Mothership's code into an executable file was for ease of use. The alternative to just clicking on the executable is to compile the code manually, either through an IDE or via command line, which is simply too inconvenient to myself and other potential users of Mothership. I had to make some modifications to the code in order for the packaged executable to be able to find the path to the haar cascade datasets, and also made some modifications to the spec file (essentially a make file for pyinstaller) in order for the executable to find the images that are used by the code. Another benefit of an executable file is that shortcuts can be made and placed in any location on your machine, regardless of where the source files are located. This makes it easy for me to maintain a clean look of my desktop, where I want the executable to be located, but not have any of the other files also appearing on my desktop. The following screenshots show the current version of Mothership running as an executable, with some cameras open.

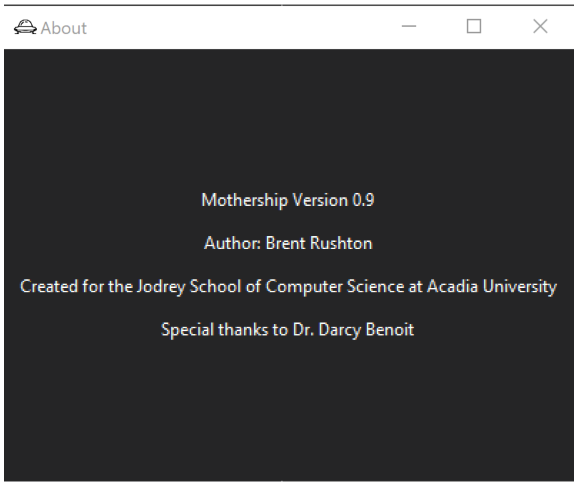


```
rtsp://192.168.1.27:8080/h264_ulaw.sdp
0
```

This screenshot shows the IP camera running outside showing the driveway and a hardware camera on my laptop at index 0 with an extremely flattering image of myself being displayed. Additionally, this screenshot was taken on Tina Turner’s birthday, hence the nice little image in the search bar at the bottom of the screen. Upon clicking the image in the top right, a window will open with a field for the user to input their email address, which will be used by the SMTP client to send an email to the inputted address:



A window will also appear when the main logo is clicked (positioned in the middle of the top bar) that provides a bit of information about the version, the author, and other general notes:



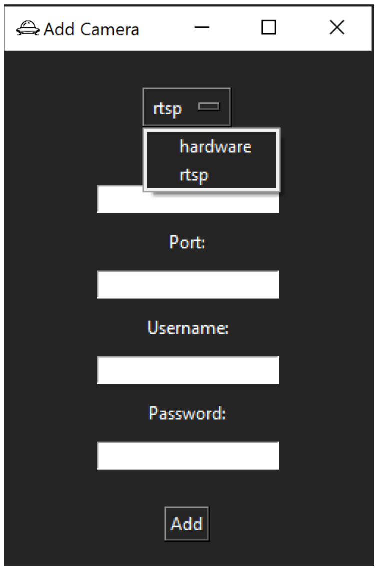
Mothership Version 0.9

Author: Brent Rushton

Created for the Jodrey School of Computer Science at Acadia University

Special thanks to Dr. Darcy Benoit

The last window that appears in Mothership that isn't an error alert is the adding camera window that includes a dropdown for either hardware or RTSP sources. Please note that RTSP sources have only been tested with the IP Webcam app by Pavel Khlebovich, and aren't guaranteed to work with any IP camera off the shelf. The following is the add camera window:



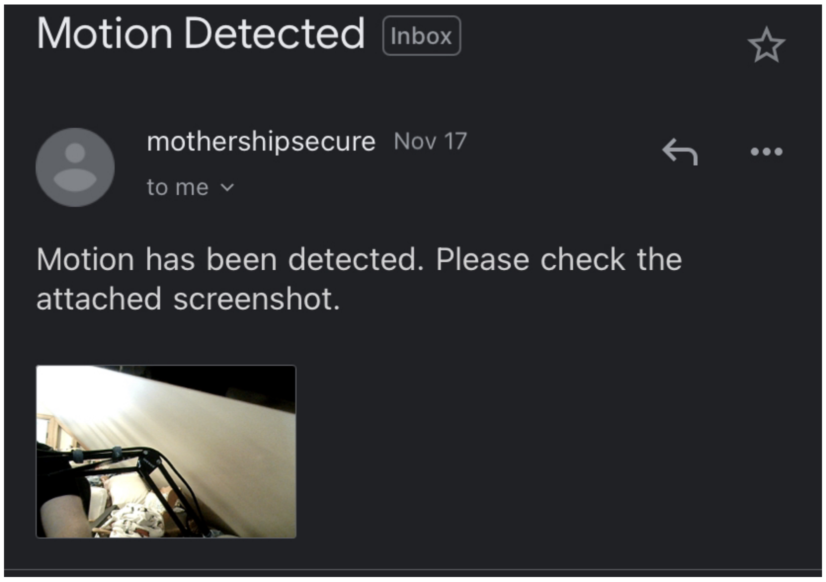
Roadblocks

There were some roadblocks that came about when developing Mothership. A lot of my early grievances came from how restricted my home network is in terms of port-forwarding. Because I am very rural, I rely on Elon Musk's Starlink LEO (Low Earth Orbit) satellite solution for an internet connection which comes with a proprietary router. This router supplies diagnostic data and other common router info via an app which is wonderful, but because of how the "cell" system works, I share a common IP with others in the same cell, with a dynamic address also assigned to each individual which changes frequently. This raises issues when I considered remote access to cameras as I realized you cannot port forward with the default starlink router, there are ways to do it with different routers but you lose out on the features of the default router, including controlling the behavior of the dish itself. So, I decided to not port forward anything and focus on the IP Camera app for android which accesses your local network for transmission. This decision kind of nullified my plans for remote access of cameras or hosting a web version on my machine at home. Moving forward with that design plan would surely have cost money somewhere along the line, whether it be hosting or for a new router, something that I wanted to avoid with this project.

The biggest roadblock that I faced was once I tried to implement motion detection, which is why it is missing from the implementation section and also why I describe Mothership as being in version 0.9 instead of 1.0. The way that I tried to implement it was that it would work separately from the face/body detection logic as I wanted it to take a screenshot when motion was detected, and email it to the user with a message and the captured screenshot as an attachment. I started to implement motion detection right after the study break, as I had implemented face/body detection the week before as I knew that recording video was my end goal vs taking screenshots which would be tied to the motion detection. My implementation worked correctly with a single camera, making the foreground from the background using OpenCV and comparing frames using the “absdiff” function that is also a part of OpenCV. However, while this implementation worked with a single camera, I ran into thread asynchronous issues when having more than a single camera open. It took a few weeks to resolve this threading issue with thread locks all while ensuring that deadlocks were impossible. After the asynchronous issue was fixed, I ran into more problems with what I assume is both the face/body detection thread and the motion detection thread trying to access the same data at the same time because I get the following error:

```
An exception occurred: invalid vector<T> subscript
An exception occurred: OpenCV(4.8.1)
d:\a\opencv-python\opencv-python\opencv\modules\objdetect\src\cascadedetect.hpp:46
: error: (-215:Assertion failed) 0 <= scaleIdx && scaleIdx < (int)scaleData->size() in
function 'cv::FeatureEvaluator::getScaleData'
```

I got this “Invalid vector <T> subscript” error before when realizing each camera had to have separate Haar cascade classifiers when it was trying to access all the cameras at once. I believe that this invalid vector error is what is causing the second error that relates to the facial/body detection as it breaks only that feature when more than one camera is added. Despite working at this issue for a minimum of 3 hours a day since study break, I am unsure what is causing the current issues but am working at more error handling to try to identify exactly where the error is happening. I am going to really try to have it resolved by the project due date of December 5th, and if I am successful, a new executable file will be built in place of Mothership version 0.9. Below is a screenshot of the email being sent when motion is detected, this shows the feature is working, just not working in the correct manner for the needs of Mothership:


Motion Detected

mothershipsecure Nov 17
to me
Motion has been detected. Please check the attached screenshot.

What I Would do Differently

Overall, I am very pleased with the work that I have done on Mothership for the Capstone Project. That being said, I do believe that there are some things that I could have done differently, especially in terms of project planning and the SDLC (Software Development Life Cycle). I think I would have benefitted drawing out UML (Unified Modelling Language) diagrams which would have really helped me in the early stages of development. I kind of took a guerilla approach to development early on that saw September filled with plenty of experimentation and research into libraries and methods of achieving what I proposed for the project. In hindsight, I should have taken a more deliberate approach with what the user was *actually* going to do with the system, something that I struggled with in terms of having a clear vision.

Another thing that I would have done differently is wireframing a GUI rather than just attempting to create the vision that I had in my head. Although I am quite pleased with the final look of Mothership, designing a GUI with wireframes would have afforded the opportunity to see what other GUIs would have looked like, and might have resulted in a fresh looking design that would have been just as functional. It also would have been advantageous to use wireframes when considering the structure of the code. Admittedly, I don’t believe that my GUI is handled in the most elegant way, but it definitely gets the job done and I'm quite proud with how it turned out given the fact that this project was the first time I used python for anything more than coding puzzles.

A third major thing that I would have done differently is keeping a developer log. It would have been nice to have reference material to look back on in tracking my progress and reflecting back on the process of developing Mothership. I did start a developer log with the idea that I would create a blog that would update the progress of Mothership and act as a living document attesting to the work that I put into this project. I really should have stuck to the developer log as I think it would have helped me think through some of the code structure and would have shown my understanding of software engineering principles and a holistic approach to the process.

The last major thing that I would do differently would be using git for versioning and maintaining goals to meet version boundaries. For the development of Mothership, I decided against using a git or other SCM (Software Change Management) systems due to the fact that Mothership is contained solely in a single script, and I was the sole person working on the project. I had used github in past projects, so I know how useful it is for backing up work and for branching development paths in order to try new things. While I made sure to back up my code frequently both to the cloud and via USB storage as well as creating new python files when trying different methods for things, I wholeheartedly believe now looking back that using a SCM would have kept me more focused and on track. Additionally, if I had used github I would have been able to make notes of changes which would have been nice to have as a reference to the progress I had made.

Future Work

In the very near future I hope to have motion detection implemented properly which would mark the completion of the baseline build that was proposed at the beginning of the term. Following the implementation of motion detection, I will then turn my focus towards the support of HTTP links. Despite the RTSP sources being virtually identical to HTTP sources, HTTP is a more modern implementation of video streaming over the internet. Additionally, HTTP is the way that most IP cameras seem to implement video transmission by default, with RTSP being optional.

In the future I also plan to get a different router so I am able to port forward the cameras on my network so I could, in theory, remotely access them from anywhere using Mothership. Additionally, it would be advantageous to look into ngrok which is a service that enables developers to expose local environments to the internet. This would be used to be able to access Mothership itself remotely, instead of accessing the individual cameras. I did mention using ngrok in the stretch build section of my initial proposal of Mothership, but I did not get past the initial research stage with it.

Another feature I would like to look into is another stretch build feature that I outlined in the proposal; camera controls and two-way audio. I think to do this I might need to invest in dedicated IP cameras with those features, as I have not found a way to do it currently with the IP camera app on android phones. I really think two-way audio would be the most interesting feature to implement in the future as it could open the doors for a “ring doorbell” type situation where I could talk to visitors at the door using Mothership.

Conclusion

In conclusion, Mothership is a viable open source option for home security cameras. Mothership leverages OpenCV, an open source library for opening camera sources and performing manipulations on frames to achieve things like facial recognition and motion detection, to open and monitor/perform logic on both hardware and RTSP camera sources. Mothership also includes a SMTP client that will open and send emails alerting the user when faces or bodies are detected in the frame. Mothership’s GUI is also a noteworthy feature as it allows for dynamic resizing of the window that also resizes the open camera sources in an array using a mix of both Tkinter and Pillow python libraries.

While there was a lot of success with the development of Mothership, stretch goals and motion detection were not implemented to the degree that is needed to ship a build of Mothership that met my standards. Additionally, there was significant learning that occurred throughout the project, particularly the importance of thorough planning and the use of UML diagrams and wireframing. There were times that I struggled with the vision of Mothership and where I wanted to take the project and more concrete plans would have thwarted a lot of the issues I had. Furthermore, using a system like Github would have provided better project management and tracking opportunities, something that was sorely missed when reflecting back on the development process.

Future work on Motherships includes refining and fully implementing motion detection, supporting HTTP sources, exploring remote access capabilities, and adding more advanced features such as two-way audio. Mothership not only fulfilled the goal of being a functional security system, but also served as a valuable learning experience in the world of software development, from conception to implementation, and will be an experience that I will hold dearly moving forward into my career. I truly believe that the Capstone Project has been the most challenging, but also the most rewarding aspect of my time spent with the Jodrey School of Computer Science at Acadia University and I am so thankful that the Capstone Project is a degree requirement. It feels really good to have a piece of software that I can show off at the end of my time at Acadia.

Works Cited

"About - OpenCV (opencv.org)." OpenCV, OpenCV, opencv.org/about/. Accessed 27 Nov. 2023.

"Python OpenCV: Capture Video from Camera - GeeksforGeeks." GeeksforGeeks, www.geeksforgeeks.org/python-opencv-capture-video-from-camera/. Accessed 27 Nov. 2023.