“Litter Audit System Utilizing Machine Vision in UAV”

RAY Project 2024

Min Khant Tun & Garnet Maxwell

York University, Lassonde School of Engineering
Department of Mechanical Engineering

August 17th, 2024

Table of Contents

1.0 Introduction

The objective of this project is to improve upon the current methods of litter audits by making them faster, simpler, and cost effective through utilizing drone technology paired with advanced object detection models. Currently litter audits in the City of Toronto are conducted manually which is time consuming and labor-intensive. Through deploying UAVs equipped with machine vision integration capable of real-time video streaming capabilities we aim to automate the process. The proposed system will not only be able to detect but also categorize litter and record the essential information. Upon detection the UAVs will be able to no only record the size and type of litter but also its precise GPS location. This paper will outline current progress and explain how the system operates to improve upon efficiency and accuracy of litter management in urban environments.

1.1 Programing Language and Libraries

The system is a combination of hardware and software. The hardware that was used for this project is a drone (Parrot Anafi), and a laptop with a linux os. The software is the code, and different libraries that it uses. Python was used as the coding language for its compatibility with the Olympe library, which is used to communicate with the drone. It's recommended to create a virtual environment, since it would reduce the chance of conflicts that may happen when installing different libraries. This can be created with pycharm (Source), that was used for development, and testing.

1.2 Olympe

. The olympe library is what is used to communicate with the drone. It only works on a linux system (Ubuntu 22.4 or higher), and it's built on python. The library doesn't work on a virtual machine, which limits the other libraries that could have been used for the project.The installation of Olympe requires pip version 20.3 or higher [6].

It can be installed with this statement, by typing it into the terminal [6]: **pip3 install parrot-olympe**

The computer connects to the drone, though it's wifi connection. When the drone is turned on it starts to broadcast its wifi connection, but it has a limit of one device per connection. This means that the code and controller can't connect to it at the same time, unless the code is changed. The wifi code has a password that can be found on the container, along with its name. The IP address of the drone is also needed for the connection, the current drone has an IP address of ("192.168.42.1").

The following method can be used to connect to the drone [6]:
- First list the IP address: **DRONE_IP = os.environ.get("DRONE_IP", "192.168.42.1")**
- Then create a variable with the information: **drone = olympe.Drone(DRONE_IP)**
- Finally connect to the drone: **drone.connect()**

1.3 Inputs

This library is responsible for reading the gamepad inputs.This library has been chosen to control the drone, while it's connected to the computer. It was also chosen for its ease of use, and implementation since it doesn't require other systems to function [7]. The current code uses an xbox controller, but other types of controllers could be used. If they are used the inputs would change, and the code would have to be adjusted.
To install this library you can use [7]: **pip3 install inputs**

Formation:
This library uses events to track the inputs of the gamepad. A basic code would be this;

```
from __future__ import print_function
# Turns the print statement into a function, so it can print the input from the controller.
from inputs import get_gamepad
# Import for the library.

def main():
    while 1:
        events = get_gamepad()
            # Creates sets the gamepad as an event
        for event in events:
            print(event.ev_type, event.code, event.state)
             # Prints the different properties of the event.
             # This can be used when you want to find what each button is, and their range.


if __name__ == "__main__":
    main()
```

Each input is categorised by their type, code, and state [7]. The type would either be 'Absolute' or 'key'. Key types are usually buttons, while absolute would be the sticks or triggers. The code is the specific code of each button. This may change based on the type of controller, but it is usually something like 'ABS_RY', which is the right trigger. The last property is the state, and it returns an integer based on how hard the button was pressed. If it's a key type it would only be 1 or 0, while an absolute would return a number from 3500 to -3500 [7].

1.4 Pandas

Pandas is a library that allows for the creation of data structures and tables. In the code it is used to create a data frame [5]. A data frame is a data table, which is able to store information based on the different categories defined in the data frame. As the code runs, it adds information to the data frame, which is saved to an excel file before the code ends.

To install this library you can use [5]: **pip3 install pandas**

1.5 csv

CSV is a library that is included in the installation of python. It is used to read and write tabular data in the csv format [4]. In the current version of the code, it is used to write information about the streaming performance.

1.6 cv2

CV2 or open cv is a library that was made for machine learning, and computer vision [3]. This library is used with Yolo to perform the different machine vision functions built in the code. It is

currently used to convert the frame data of the drone to BGR format so it can be used by the object detection model.

To install this library you can use [3]: **pip3 install opencv-python**

1.7 ultralytics

This library holds the information for the object detection model. The models are called Yolo, and it is used to apply object detection to the video frames captured by the drone [2].

To install this library you can use [2]: **pip3 install ultralytics**

1.8 pynput

This library allows for keyboard controls to be read, and used in the code [1]. In the code it is used to create variables, and statements that are able to check use certain keys to control parts of the code.

To install this library, you can use [1]: **pip3 install pynput**

1.9 Hardware Requirements

1. ***Drone:*** Parrot Anafi (discontinued as of 2024)

    - **Camera**: Equipped with a 4K HDR camera, providing high resolution video capture with a bitrate of 100 Mbs
    - **Controller**: The drone is operated using the SkyController which is able to transmit to a maximum distance of 4 km.
    - **Weight**: The drone weighs 320g, and exceeds the 280g limit to operate a drone without license, hence the operation is required to obtain the basic drone license from the Transport Canada website.
    - **Altitude**: Capable of reaching a maximum altitude of 4500m above sea level.
    - **GPS System**: Parrot Anafi have a integrated GPS system for precise navigation and location tracking.
    - **Battery**: The drone contains a replaceable battery slot, with each batter capable of operating a maximum of 25 minutes on a single charge.

As for the computer and GPU requirements, the selected model of ROG laptops was able to fulfill the requirements for the model training due to integrated components within the device, no longer requiring an external GPU to accelerate the model training at current scale.

2. Computer & GPU:

    - **Model**: ROG Strix Scar 17 G733ZM
    - **GPU**: Integrated NVIDIA GeForce RTX 3060 GPU
    - **CPU**: 12th Gen Intel(R) Core(™) i7-12700H, offering high performance for running complex computations and processes.
    - **RAM**: 16GB, although sufficient for the training model an improved RAM would result in smoother video display and reduced overloading errors during operations.

2.0 Model Training

2.1 YOLO Model Overview

The project primarily used the YOLOv8n model for this project as it provides a consistent performance and more publicly available datasets for our use case. In comparison to the YOLOv5n and YOLOv10n models which lacked consistent performance within our use case.

The YOLO (You Only Look Once) model by Ultralytics is a real-time object detection model with a large library of resources publicly available. The key features of the YOLO such as its ability to accurately detect objects with unparalleled speed and simplicity fitted well with the scope of the Litter Audit System project. The YOLO model also have a variant called the YOLOv8n-seg model which contains information that can be extracted to perform live object segmentation that can be utilized in future works of the project. The YOLO models ability to perform segmentation for the users to extract and utilize is a massive advantage for the project's future scope of aiming to perform weight estimation. Hence the YOLO model, specifically the YOLOv8n-seg, is the optimal model for this project.

2.2 GPU & Nvidia Library Optimization

The RTX 3060 GPU was utilized to have a hardware acceleration power. In order to fully utilize GPUs in training for machine learning models, the Nvidia CUDA library available in Python was necessary. The publicly available Nvidia CUDA Deep Neural Network Library (cuDNN) which is required to be integrated into the environment variable of the operating device allows for a significant expansion in capabilities of GPU acceleration. Utilizing GPU to train such models allows for each cycle of training to be significantly faster compared to a CPU-intensive training process.

The RTX 3060 GPU increased the computation power substantially, enabling efficient processing of large amounts of data. The use CUDA allowed us to optimize model training through:

- **Parallel Processing:** Enable us to use parallel processing abilities of GPUs to handle multiple operations simultaneously

- **Accelerated Computations:** Substantially sped up the training period of the model

- **Memory Management:** Allows for the memory of the operating system to be used efficiently during the training process, significantly reducing the chances of crashing and optimizing training time

The utilization of the built-in RTX 3060 alongside the Nvidia library was crucial for achieving the desired performance and approximately allowed us to produce multiple versions of a trained model within a day, prior to this utilization we were restricted to one fully trained version of the YOLO model.

2.3 Training Parameter & Initialization

The pretrained weights for the YOLOv8n model is initialized before training begins in order to build a custom model around the pretrained model. The training parameters are later adjusted to ensure the best accuracy. The following parameters were utilized:

- Optimizer: *Adam* optimizer is a very popular choice for training deep learning model and is highly recommended by the Ultralytics library, the optimizer computes and enables the best learning rate based on different parameters ensuring a reliable training.

- Batch Size: 16; batch size is the amount of training images used in one epoch (cycle), a batch size of 16 was the optimal amount after multiple trial and errors. Choosing a smaller batch size lowers the memory requirements but also reduces accuracy at times. Larger batch sizes are more accurate but can cause biases in training data if other parameters are not adjusted.

- Epochs: 50; An epoch is a complete cycle through the entire dataset. Training a model for 50 epoch means that the training model will see the entire dataset exactly 50 times during training. Increasing the number of epochs means higher accuracy, however it also increases the chances of the model ‘overfitting’ where it will start to learn the noise in the images and develop biases

The next set of training parameters are the data augmentation parameters which allows for randomness in training data to ensure that the model is able to operate and detect unforeseen objects in different scenarios. These parameters allows us to recreate the perspective of the drone’s camera in operation. The augmentation parameters are:

- Hue: 0.01; this parameter allows the model to perform slightly change colors of training images, which is beneficial when the color of the litter varies with exposure to the environment over a long period of time.

- Saturation: 0.6; helps the model to operate effectively even if there is dramatic changes in lighting and color intensity

- Rotation: 180; rotate images within the 0 to 180 degree range, this ensures that the model can recognize objects despite the orientation of the of the object.

- Shear:20; this shifts one part of the image in a particular direction while keeping the opposite side the same, this is implemented with the intention to simulate the effect of the camera angle and help the model detect from different perspectives.
- Copy-Paste: 0.5; this parameter copies an object from one image of a training dataset and paste it to another image. This allows for the model to be able to detect multiple categories of the image increasing variability within the training dataset.

3.0 Drone Integration:

3.1 Overview of Parrot Anafi drone

The Parrot Anafi was chosen for its specifications, and SDK. For this project, the drone needed to do multiple things, such as connect to a computer, share data, and allow for other libraries to interact with it. Parrot Anafi with the use of its SDK allows for a large variety of functions. The current code uses the olympe library for communicating with drones, and sending commands.

3.2 Specific functions of the drone and code

The drones function for this code can be split into three different sections. The first is its ability to record, and stream video from its camera. The second is its ability to move based on the controller input. The third is to produce GPS data based on the drones location. Each function will be explained by going through the code, and explaining what is a part of it. The blue parts are the relevant parts of the code.

3.2.1 Video Feed

The video portion of the code makes up for most of it. The video starts at the beginning of the code, when it's initiated;

```
# Gets the drone IP address, and the port so the code can connect to it.
DRONE_IP = os.environ.get("DRONE_IP", "192.168.42.1")
DRONE_RTSP_PORT = os.environ.get("DRONE_RTSP_PORT")

# Finds the model which is saved as a .pt file, and saves it to a variable.
Model_Path = r"/home/labpc/Downloads/best(5).pt"
model = YOLO(Model_Path)

# Define debounce delay in seconds (adjust as needed)
DEBOUNCE_DELAY = 0.3

# Define a global DataFrame to store GPS data
gps_data_df = pd.DataFrame(columns=['Timestamp', 'Latitude (Degrees)', 'Longitude (Degrees)', 'Altitude (ft)'])


class StreamingExample:
```

```
def __init__(self):
    # Create the olympe.Drone object from its IP address.
    self.drone = olympe.Drone(DRONE_IP)
    # Creates a temporary for storing output files.
    self.tempd = tempfile.mkdtemp(prefix="olympe_streaming_test_")
    # Prints the path for the temporary directory.
    print(f"Olympe streaming example output dir: {self.tempd}")
    # Opens an empty list to store h264 statistics.
    self.h264_frame_stats = []
    # Opens a temporary directory for writing video stream statistics.
    self.h264_stats_file = open(os.path.join(self.tempd, "h264_stats.csv"), "w+")
    self.h264_stats_writer = csv.DictWriter(
        self.h264_stats_file, ["fps", "bitrate"]
    )
    self.h264_stats_writer.writeheader()
    self.frame_queue = multiprocessing.Queue()
    self.processed_frame_queue = multiprocessing.Queue()
    self.processes = []
    self.running = multiprocessing.Event()
    self.running.set()
```

- The DRONE_IP is used to connect to the drone over its internet connection, while the DRONE_PORT is used to retrieve the RTSP (Real-Time Streaming Protocol) from the environment variables. The purpose of the RTSP is to allow for the code to change the drone streaming server address if it's available.

- First the initializer creates an object of the drone, based on its IP address. This was done so different functions in the class could interact with the drone. The next two lines create a temporary directory where the streaming information is stored.

- The initializer does a similar process for the h.264 statistics, which is information about the video that the drone is streaming. The drone streams its video in the h.264 format, and the code monitors its properties as the drone streams it.

- It does a similar process for the h.264 statistics, which is information about the video that the drone is streaming. The drone streams its video in the h.264 format, and the code monitors its properties as the drone streams it.

- self.running.set(), is used there to set it as true. This is a condition that multiple while loops use, so they run continuously while the code is running.

The first function that interacts with the video portion, is the start function. Similarly to the start of the class. This function is used to set up, and start different processes.

def start(self):

```
    # Connect to drone
    assert self.drone.connect(retry=3)

    if DRONE_RTSP_PORT is not None:
```

```
            self.drone.streaming.server_addr = f"{DRONE_IP}:{DRONE_RTSP_PORT}"

        # You can record the video stream from the drone if you plan to do some
        # post processing.
        self.drone.streaming.set_output_files(
            video=os.path.join(self.tempd, "streaming.mp4"),
            metadata=os.path.join(self.tempd, "streaming_metadata.json"),
        )

        # Setup your callback functions to do some live video processing
        self.drone.streaming.set_callbacks(
            raw_cb=self.yuv_frame_cb,
            h264_cb=self.h264_frame_cb,
            start_cb=self.start_cb,
            end_cb=self.end_cb,
            flush_raw_cb=self.flush_cb,
        )
        # Start video streaming
        self.drone.streaming.start()

        # Start multiple processing threads
        num_processes = multiprocessing.cpu_count()
                        self.processes    =    [multiprocessing.Process(target=self.yuv_frame_processing,
args=(self.running,))
                    for _ in range(num_processes)]
        for p in self.processes:
            p.start()

        # Start the video output processing thread
        self.output_thread = threading.Thread(target=self.process_video_output)
        self.output_thread.start()
```

- The code first attempts to connect to the drone with assert self.drone.connect(retry=3), this attempts to connect to the drone three times before closing the code.

- The second line checks to see if there is a RTSP port it can connect to, and if it can it would connect to the server.

- The third section of the code saves the metadata and the streamed connection into the temporary directory that was defined in the earlier section.

- The fourth section is to set up functions that are responsible for specific aspects of streaming the video. Each one will have its own section.

- The last line starts streaming the video from the drone.

The next sections of code will explain the callback functions that are used for video processing. The first is yuv_frame_cb.

```
def yuv_frame_cb(self, yuv_frame):
    """
    This function will be called by Olympe for each decoded YUV frame.

        :type yuv_frame: olympe.VideoFrame
    """
    yuv_frame.ref()
    info = yuv_frame.info()
    height, width = info["raw"]["frame"]["info"]["height"], info["raw"]["frame"]["info"]["width"]
    yuv_data = yuv_frame.as_ndarray()
    self.frame_queue.put((yuv_data, height, width))
    yuv_frame.unref()
```

- The objective of this call back function is to get the frame's dimensions, and convert it into a numpy array for further processing.

- yuv_frame.ref() calls the frame data from the video, and tells the code it will be used.

- The next two lines create a variable called info, which stores the information about the frame. The other line creates a dictionary containing information about the frame.

- The code uses as_ndarray() to turn the frame data into a numpy array, and saves it to yuv_data.

- self.frame_queue.put() is used to queue the different information into a separate thread, so it can be used without having to interfere with the main thread.

- The final line yu_frame.unref(), is used to close the reference frame, so the process can loop again.

The next function uses the information stored in the queue from the previous function to feed into the machine learning model, and display it.

```
def yuv_frame_processing(self, running):
    while running.is_set():
        try:
            yuv_data, height, width = self.frame_queue.get(timeout=0.1)
        except queue.Empty:
            continue
        # Convert YUV data to OpenCV format
        yuv_data = yuv_data.reshape((height * 3 // 2, width))
        bgr_frame = cv2.cvtColor(yuv_data, cv2.COLOR_YUV2BGR_I420)
        results = model(bgr_frame)
        # Plot boundary & image masking from obtained results in f2f format
        annotated_frame = results[0].plot()
        self.processed_frame_queue.put(annotated_frame)
```

- This function uses a while loop that runs as long as the code is running. It uses a try block to first try to retrieve the frame data from the queue, if the queue isn't empty it would continue

the rest of the code. If the queue is empty then it would try again, and would not continue the rest of the code.

- If it gets the frame information it first reshapes it with yuv_data.reshape(), after reshaping the frame it converts it with the cvtColor function, and saves it into a variable called bgr_frame.

- The frame data is then passed through the object detection model with model(bgr_frame), and saved to the results variable.

- The results variable is used with the plot() function, which creates bounding boxes around the detected objects, and labels the images with predicted values.

- Finally the annotated frame is stored into another queue called self.processed_frame_queue, where it can be used by another function.

The next function serves as a method for calling the processed video frames, and displaying them.

```
def process_video_output(self):
    while self.running.is_set():
        try:
            frame = self.processed_frame_queue.get(timeout=0.1)
        except queue.Empty:
            continue

        cv2.imshow('Machine Vision', frame)
        if cv2.waitKey(1) & 0xFF == ord('r'):
            self.running.clear()
            break

    cv2.destroyAllWindows()
```

- This function uses the try method to load the frame data stored in the queue, which was annotated on the previous function and saved to the queue.

- After loading the queued frame, it would be displayed with cv2.imshow(), which is a function from OpenCV.

- The function ends with an if statement that checks if the r button was clicked. If it is clicked it would stop the loop, and stop the video display.

The flush_cb function is a function that is responsible for managing video frames during streaming. It would be inspecting the video output from the previous function (process_video_output(self)), and getting rid of any extra frames stored in the queue.

```
def flush_cb(self, stream):
    if stream["vdef_format"] != olympe.VDEF_I420:
        return True
    while not self.frame_queue.empty():
        self.frame_queue.get_nowait()
```

```
return True
```

- This callback function checks the format of the stream with the stream() operator. This retrieves the video format of the stream, and checks if it matches a certain format. If it doesn't match the function returns true, which means the stream won't have to go through the operation below it.
- The second operation checks if frame_queue is empty, if it isn't they are removed until it's emptied.

- This process checks if the frame is the required format, and is empty.

The purpose of this callback function uses the Olympe library to process, collect information, and write the drones H.264 frames to a CSV file.

```
def h264_frame_cb(self, h264_frame):
    """
    This function will be called by Olympe for each new h264 frame.

      :type yuv_frame: olympe.VideoFrame
    """

    # Get a ctypes pointer and size for this h264 frame
    frame_pointer, frame_size = h264_frame.as_ctypes_pointer()

    # For this example we will just compute some basic video stream stats
    # (bitrate and FPS) but we could choose to resend it over another
    # interface or to decode it with our preferred hardware decoder.

    # Compute some stats and dump them in a csv file
    info = h264_frame.info()
    frame_ts = info["ntp_raw_timestamp"]
    if not bool(info["is_sync"]):
        while len(self.h264_frame_stats) > 0:
            start_ts, _ = self.h264_frame_stats[0]
            if (start_ts + 1e6) < frame_ts:
                self.h264_frame_stats.pop(0)
            else:
                break
        self.h264_frame_stats.append((frame_ts, frame_size))
        h264_fps = len(self.h264_frame_stats)
        h264_bitrate = 8 * sum(map(lambda t: t[1], self.h264_frame_stats))
        self.h264_stats_writer.writerow({"fps": h264_fps, "bitrate": h264_bitrate})
```

- This callback function is for the processing of the frames that would directly come from the drone. Firstly the as_ctypes_pointer(), is used to find the memory location that stores information about the frame data.

- The function then creates a variable called info, which stores the information from the frame such as timestamps, size, etc.. The line under it saves the network time protocol (NTP) of the frame in the frame_ts variable.

- For the processing aspect it first checks if the frames between the code and drone are in sync, with if not bool(info["is_sync"]).
- If the frames are not in sync it would enter the while loop that checks the current time of each frame, if they are later than the current timestamp they would be removed from the list. The goal of this is to make sure the list only contains frames from the last second, so it remains consistent.

- After running the while loop, the frames timestamp and size are appended to the frame_ts, and frame_size list for future reference.

- The bitrate of all the frames is calculated, and stored in h264_bitrate.

- Finally the bitrate, and fps is written to a CSV file with self.h264_stats_writer, which serves as a log for the process.

3.2.2 GPS Logic

The GPS logic is controlled by the keyboard key, if 'a' is pressed then the location of the drone would be saved. This is completed in a function called start_keyboard_listener(). The function uses the Olympe library, to get the altitude, longitude and latitude. To save the GPS data, a data frame is created, which is a data structure capable of storing information based on different categories.

```
olympe.log.update_config({"loggers": {"olympe": {"level": "WARNING"}}})

# Gets the drone IP address, and the port so the code can connect to it.
DRONE_IP = os.environ.get("DRONE_IP", "192.168.42.1")
DRONE_RTSP_PORT = os.environ.get("DRONE_RTSP_PORT")

# Finds the model which is saved as a .pt file, and saves it to a variable.
Model_Path = r"/home/labpc/Downloads/yolov5nu.pt"
model = YOLO(Model_Path)

# Define debounce delay in seconds (adjust as needed)
DEBOUNCE_DELAY = 0.3

# Define a global DataFrame to store GPS data
gps_data_df = pd.DataFrame(columns=['Timestamp', 'Latitude (Degrees)', 'Longitude (Degrees)', 'Altitude (ft)', 'Objects', 'Confidence'])

# Define the key to check
key_to_check = 'a'  # Example: 'a' key

# Define the key to check
key_to_check = 'g'  # Example: 'a' key

# Dictionary to map class IDs to object names
class_names = {
    1: 'Water Bottle',      # Example class IDs and names
    0: 'Metal Can',
```

```
    # Add other class IDs and names as needed
}
```

- At the beginning of the code the data frame is created with pd.DataFrame(), using the pandas library. Inside the operation, columns are defined based on the type of data that will be recorded.

- Another important statement is the creation of the key_to_check variable which stores which key will be used to control the function.

- The last set of statements is a dictionary which is used to identify the objects read from the model results. When extracting the object information from the model it returns integers based on the training data. The dictionary takes these integers and converts them into strings that someone would understand. This would have to be changed whenever the model is updated.

```
def start_keyboard_listener(self):
    # Declare gps_data_df as global
    global gps_data_df
    global gps_logging_active

    # Initialize debounce timers for each button
    debounce_tl = time.time()
    debounce_tr = time.time()
    debounce_tul = time.time()
    debounce_tur = time.time()
    debounce_au = time.time()
    debounce_ad = time.time()
    debounce_Gi = time.time()
    debounce_rs = time.time()

    # Define global variables
    gps_logging_active = False  # Initialize global variable

    # Start the keyboard listener in a separate thread
    keyboard_listener = self.start_keyboard_listener()

    # Variable to keep track of camera position
    V = 0
    H = 0

    try:
        while self.running.is_set():

            events = inputs.get_gamepad()
            for event in events:
                if event.ev_type == 'Absolute':
                    if event.code == 'ABS_RY':
                        y_value2 = event.state
```

```
                    if y_value2 > 30000 and (time.time() - debounce_ad) > DEBOUNCE_DELAY:  # Downward
                        self.drone(moveBy(0, 0, -0.25, 0)).wait()
                        print("Moving Downwards")
                        debounce_ad = time.time()
                    elif y_value2 < -30000 and (time.time() - debounce_au) > DEBOUNCE_DELAY:  # Upwards
                        self.drone(moveBy(0, 0, 0.25, 0)).wait()
                        print("Moving Upwards")
                        debounce_au = time.time()
                    elif abs(y_value2) <= 30000:  # Deadzone for y_value2
                        self.drone(moveBy(0, 0, 0, 0)).wait()  # Stop movement
                elif event.code == 'ABS_Z':
                    if event.state <= 1000 and (time.time() - debounce_tul) > DEBOUNCE_DELAY:
                        self.drone(moveBy(0, -1, 0, 0)).wait()
                        print("Moving Left")
                        debounce_tul = time.time()
                elif event.code == 'ABS_RZ':
                    if event.state <= 1000 and (time.time() - debounce_tur) > DEBOUNCE_DELAY:
                        self.drone(moveBy(0, 1, 0, 0)).wait()
                        print("Moving Right")
                        debounce_tur = time.time()
                elif event.code == 'ABS_HAT0Y':
                    value = event.state
                    if value == -1 and (time.time() - debounce_Gi) > DEBOUNCE_DELAY:
                        # Set the gimbal target orientation
                        # Uses upwards keypad on the controller
                        H = V + 10
                        gimbal_command = set_target(
                            gimbal_id=0,
                            control_mode="position",  # Use "velocity" for smooth movement
                            yaw_frame_of_reference="absolute",  # Can be "relative" or "absolute"
                            yaw=0.0,
                            pitch_frame_of_reference="absolute",  # Use "absolute" or "relative"
                            pitch=H,
                            roll_frame_of_reference="absolute",  # Use "absolute" or "relative"
                            roll=0.0,
                        )

                        # Send the command and wait for the completion
                        command_result = self.drone(gimbal_command).wait()

                        # Check if the command was successful
                        if command_result.success():
                            print("Gimbal target orientation set successfully.")
                        else:
                            print("Failed to set gimbal target orientation.")

                        V = H
```

```
            elif value == 1 and (time.time() - debounce_Gi) > DEBOUNCE_DELAY:
                # Set the gimbal target orientation
                # Uses downwards keypad on the controller
                H = V - 10
                gimbal_command = set_target(
                    gimbal_id=0,
                    control_mode="position",  # Use "velocity" for smooth movement
                    yaw_frame_of_reference="absolute",  # Can be "relative" or "absolute"
                    yaw=0.0,
                    pitch_frame_of_reference="absolute",  # Use "absolute" or "relative"
                    pitch=H,
                    roll_frame_of_reference="absolute",  # Use "absolute" or "relative"
                    roll=0.0,
                )

                # Send the command and wait for the completion
                command_result = self.drone(gimbal_command).wait()

                # Check if the command was successful
                if command_result.success():
                    print("Gimbal target orientation set successfully.")
                else:
                    print("Failed to set gimbal target orientation.")

                V = H
        elif event.ev_type == 'Key':
            if event.code == 'BTN_SOUTH' and event.state == 1:  # A button
                self.drone(TakeOff())
                print("TakeOff")
            elif event.code == 'BTN_EAST' and event.state == 1:  # B button
                self.drone(Landing())
                print("Landing")
            elif event.code == 'BTN_WEST' and event.state == 1:  # Y button
                self.drone(moveBy(0, 0, 0, 1)).wait()
                print("Turning clockwise")
            elif event.code == 'BTN_NORTH' and event.state == 1:  # X button
                self.drone(moveBy(0, 0, 0, -1)).wait()
                print("Turning counter clockwise")
            elif event.code == 'BTN_TL':
                if event.state == 1 and (time.time() - debounce_tl) > DEBOUNCE_DELAY:
                    self.drone(moveBy(-1, 0, 0, 0)).wait()
                    print("Moving Backwards")
                    debounce_tl = time.time()

            elif event.code == 'BTN_TR':
                if event.state == 1 and (time.time() - debounce_tr) > DEBOUNCE_DELAY:
                    self.drone(moveBy(1, 0, 0, 0)).wait()
                    print("Moving Forwards")
                    debounce_tr = time.time()
```

```
    except KeyboardInterrupt:
        print("Stopping control with Xbox controller...")
        print("GPS monitoring stopped.")
        # Save data to Excel file
        save_to_excel(gps_data_df)
        keyboard_listener.stop()  # Stop the keyboard listener

    print("Landing...")
    self.drone(Landing() >> FlyingStateChanged(state="landed", _timeout=5)).wait()
    print("Landed\n")
```

- Inside the function, there is another that was designed to respond to key requests. It uses a key as an input, which is from the library.

- The code calls the two variables from the beginning of the code. Since they are global variables (meaning any part of the code has access to them), they have to be called in the function in this way.

- A try statement is made, with an if statement inside it. A try statement was made in case any errors occur when pressing other keys. The if statement checks if the g key was pressed.

- If the key was pressed, it would first toggle the gps_logging_active to be true. This was added to make sure it doesn't remain active when it's not needed. It's defined in another function.

- Inside the if in another that handles the GPS potion. It gets the GPS location of the drone, and saves it to a variable as a dictionary. The code would record the GPS properties from the variable by calling the variables from the dictionary.

- Another set of information that is added to the dataframe is the object information, using a function called Frame_Check() it's able to use the results variable. With the result variable the function can extract the object, and confidence of the detected objects with the for statement.

- After saving the GPS data, and time they are added to the dataframe with pd.concat().

- The last three lines of the function create a way for the function to be called in another function. This is used in the fly() function since multithreading is used so both can run at the same time.

```
def Frame_Check(self):
    try:
        # Get frame data from queue
        yuv_data, height, width = self.frame_queue.get()

        # Reshape and convert YUV to BGR
        yuv_data = yuv_data.reshape((height * 3 // 2, width))
        bgr_frame = cv2.cvtColor(yuv_data, cv2.COLOR_YUV2BGR_I420)
```

```
            # Run inference
            results = model(bgr_frame)

            return results

        except queue.Empty:
            # Handle empty queue case
            print("Queue was empty")
            return None
        except cv2.error as e:
            # Handle OpenCV errors
            print(f"OpenCV error: {e}")
            return None
        except Exception as e:
            # Handle other exceptions
            print(f"Unexpected error: {e}")
            return None
```

- This is the Frame_Check function that gets the frame data when the button is pressed. It uses a try loop to help with errors that may occur when trying to extract the data.

- Yuv_frame-processing wasn't used since it was responsible for streaming the video to the computer, and conflicted with the keyboard function.

3.2.3 Controller Input

The controller input uses the inputs, and Olympe library to control the actions of the drone. The operation is stored in a function that is called and looped through continuously, as the code is running. The function is also responsible for saving the GPS data by calling another function, and starting the keyboard listener.

```
def fly(self):
    # Declare gps_data_df as global
    global gps_data_df
    global gps_logging_active

    # Initialize debounce timers for each button
    debounce_tl = time.time()
    debounce_tr = time.time()
    debounce_tul = time.time()
    debounce_tur = time.time()
    debounce_au = time.time()
    debounce_ad = time.time()

    # Define global variables
    gps_logging_active = False  # Initialize global variable

    # Start the keyboard listener in a separate thread
    keyboard_listener = self.start_keyboard_listener()
```

```
        try:

            while self.running.is_set():

                events = inputs.get_gamepad()
                for event in events:
                    if event.ev_type == 'Absolute':
                        if event.code == 'ABS_RY':
                            y_value2 = event.state
                                if y_value2 > 30000 and (time.time() - debounce_ad) > DEBOUNCE_DELAY:  #
Downward
                                self.drone(moveBy(0, 0, -1, 0)).wait()
                                print("Moving Downwards")
                                debounce_ad = time.time()
                             elif y_value2 < -30000 and (time.time() - debounce_au) > DEBOUNCE_DELAY:  #
Upwards
                                self.drone(moveBy(0, 0, 1, 0)).wait()
                                print("Moving Upwards")
                                debounce_au = time.time()
                            elif abs(y_value2) <= 30000:  # Deadzone for y_value2
                                self.drone(moveBy(0, 0, 0, 0)).wait()  # Stop movement
                        elif event.code == 'ABS_Z':
                            if event.state <= 1000 and (time.time() - debounce_tul) > DEBOUNCE_DELAY:
                                self.drone(moveBy(0, -1, 0, 0)).wait()
                                print("Moving Left")
                                debounce_tul = time.time()

                        elif event.code == 'ABS_RZ':
                            if event.state <= 1000 and (time.time() - debounce_tur) > DEBOUNCE_DELAY:
                                self.drone(moveBy(0, 1, 0, 0)).wait()
                                print("Moving Right")
                                debounce_tur = time.time()

                    elif event.ev_type == 'Key':
                        if event.code == 'BTN_SOUTH' and event.state == 1:  # A button
                            self.drone(TakeOff())
                            print("TakeOff")
                        elif event.code == 'BTN_EAST' and event.state == 1:  # B button
                            self.drone(Landing())
                            print("Landing")
                        elif event.code == 'BTN_WEST' and event.state == 1:  # Y button
                            self.drone(moveBy(0, 0, 0, 1)).wait()
                            print("Turning clockwise")
                        elif event.code == 'BTN_NORTH' and event.state == 1:  # X button
                            self.drone(moveBy(0, 0, 0, -1)).wait()
                            print("Turning counter clockwise")
                        elif event.code == 'BTN_TL':
                            if event.state == 1 and (time.time() - debounce_tl) > DEBOUNCE_DELAY:
                                self.drone(moveBy(-1, 0, 0, 0)).wait()
                                print("Moving Backwards")
```

```
                    debounce_tl = time.time()

                elif event.code == 'BTN_TR':
                    if event.state == 1 and (time.time() - debounce_tr) > DEBOUNCE_DELAY:
                        self.drone(moveBy(1, 0, 0, 0)).wait()
                        print("Moving Forwards")
                        debounce_tr = time.time()

        except KeyboardInterrupt:
            print("Stopping control with Xbox controller...")
            print("GPS monitoring stopped.")
            # Save data to Excel file
            save_to_excel(gps_data_df)
            keyboard_listener.stop()  # Stop the keyboard listener

        print("Landing...")
        self.drone(Landing() >> FlyingStateChanged(state="landed", _timeout=5)).wait()
        print("Landed\n")
```

- This function has a few lines of setup. The first part calls the global variables, so they can be used in the function. The second part initialzes the timer variables. These are used for the controller. Since the controller is very sensitive, timers were created that limited the number of actions that can be performed. This stops multiple inputs from being registered when one is intended. The next one defines gps_logging_active to false so the keyboard function can be used. Finally the keyboard function is called as a separate thread that would run, as this code runs.

- After doing the setup, a try statement was created with a while loop inside it. The try statement was created to use the keyboardInterrupt statement.

- As explained at the start of the report, the input library has a gamepad to control the drone by reading the inputs. The Olympe library is used to send commands to the drone. For example, self.dronemoveby() is used to move the drone along the x,y and z axis. The numbers represent how far the drone should move in the direction.

- Another feature that the control controls is the position of the camera, which is called the gimbal.

- The lines under KeyboardInterrupt run when the code is going to be closed down. In these lines it stops the separate thread, uses a function to save the dataframe to an excel file, and lands the drone.

```
def save_to_excel(df):
    # Save DataFrame to Excel file
    filename = 'gps_data6.xlsx'
    df.to_excel(filename, index=False, engine='openpyxl')
    print(f"GPS data saved to {filename}")
```

- In this function df.to_excel() is used to save the dataframe to an excel file. A statement is printed at the end to confirm.

3.3 CPU Multithreading & Multiprocessing

To fully utilize the processing power of the operating system to efficiently process the video transmission from the drone it is beneficial to implement CPU multithreading and multiprocessing. Multithreading allows us to utilize the multiple cores within a CPU equally and effectively. In order to achieve this the following steps are required:

1. **Importing Required Libraries:**

   In order to achieve both multithreading and multiprocessing, the following two basic basic libraries are required:

   ```
   Import threading
   Import multiprocessing
   ```

2. **Creating Queues for multiprocessing:**

   Multiprocessing queues have to be created initially to handle communication between different processes. The first queue is used to store the raw frames captured from the drone, and the second queue is used to store frames after the processing.

   ```
   self.frame_queue = multiprocessing.Queue()
   self.processed_frame_queue = multiprocessing.Queue()
   ```

3. **Create & Start Multiple Processes:**

   This section of the code will start the multiple processes for frame processing, firstly it will identify the number of processes to start which will be equal to the number of CPU cores available on the device. Then a list of 'multiprocessing.Process' objects is utilized alongside 'yuv_frame_processing' method. All the processes is initialized using the 'start' allowing for the 'yuv_frame_processing' method to be executed on all of the available CPU cores.

   ```
   num_processes = multiprocessing.cpu_count()
   self.processes = [multiprocessing.Process(target=self.yuv_frame_processing,
   args=(self.running,))
               for _ in range(num_processes)]
   for p in self.processes:
     p.start()
   ```

4. **Creating Thred for Output Video:**

   A separate thread is required to be made in order to handle video output processing. A similar approach is reused in order to achieve this; firstly the 'self.output_thread' is created as a 'threading.Thread' object to target the 'process_video_output' method. Then the thread is initialized using the 'start' method allowing it to run alongside the other processes.

```
self.output_thread = threading.Thread(target=self.process_video_output)
self.output_thread.start()
```

5. **Combine the multiple Processes and Threads:**

   To ensure all the process and threads complete before the program exits, we have to utilize the 'join' method which ensures that it blocks the running program to end before every process and thread has finished to ensure clean shutdown.

```
for p in self.processes:
    p.join()

self.output_thread.join()
```

The processes methods mentioned above are the frame processing section of the code and hence even if the method or approach were to change, it will have minimal impact on the foundational CPU multithreading and multiprocessing steps mentioned above.

Through the usage of multiprocessing and multithreading, the code can efficiently handle a very intensive task of real-time video processing from the drone. Neglecting these methods can lead to overloading errors which will crash the running program, or footage delays up to 2 minutes. Even if a stronger CPU were to be used, the process of multithreading and multiprocessing allows the project to be completed without having to invest in a higher end CPU allowing for the budget to be lower. This method is essential to ensuring that all hardware components within the project are utilized to its full potential through the utilization of software installations and programming. This setup allowed the project to ensure smooth and responsive video processing.

4.0 Results and Discussion

4.1 Key Findings

The project allowed for different interactions between multiple systems, and applications. The Olympe, and ultralytics libraries were used to create a logging system that used the GPS function of the drone. This method can be applied to other systems as well, since it doesn't need the drone to function. Another important aspect of the project was object detection. Yolo has proven to be a reliable method for doing object detection, with its large documentation, and support through other sources. For example, roboflow can be used to either make a dataset or download an existing one. This allows for the process to be simplified, and reduces the time needed to test different projects.

4.2 Performance

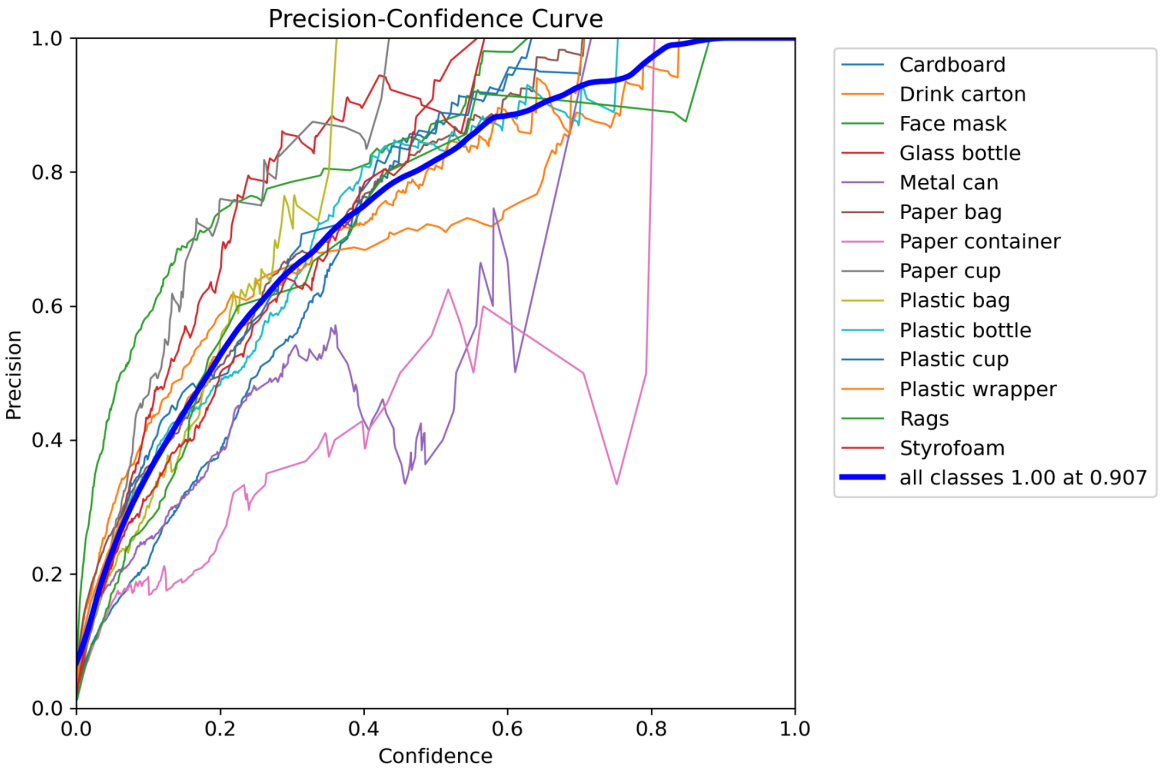
The performance of the project has been split into two sections. The first section focuses on the drone, and its effectiveness on getting data. The second section is how effective the object detection model is.

4.2.1 Drone Performance

The drone was tested by flying it, and seeing if it was able to record and identify objects. With the current code the drone is able to fly by using the controller, record its location with the GPS when the keyboard is pressed, and detect objects. Another step the project could take would be to add what the drone detected to the logged data.

4.2.2 Model Performance

The model is tested to measure the effectiveness of the drone at detecting certain objects. Yolo has multiple options to help with measuring the effectiveness of the model. One of these options is the graphs which show information about the model as it's training.



This figure shows the different categories of the model, and two important statistics. The precision represents how effective the model was at creating bounding boxes around the objects when running through the validation process. The confidence represents how effective the model was at identifying objects found by the bounding boxes. For this table the 'all classes 1.00 at 0.907' means that the model has a 90.7% chance of being correct when identifying different objects in the class. With the use of this graph, the effectiveness of a model can be identified, and recorded.

4.3 Limitations and Challenges

This project is mostly limited by the compatibility of the hardware, and software. Since the Olympe library needs to run on a linux operating system, the code is limited to the different libraries that can be added to the project. One example would be the ability to use the GPU of the computer. Since it's on linux, a GPU can't be used to help speed up the processing of the computer when annotating the frames. Instead the CPU was maxed, which is not as effective as using the GPU. Another problem that we found was trying to communicate with the drone itself. The Olympe library does have documentation, but it misses many different things which are covered in the report. For example, the drone needed to connect to the app for a software update, before it would work with the computer. The documentation also doesn't fully explain some of their syntax that is used to control the drone. The GPS function is an example of this, since they don't have any examples, trial and error used to create the line of code.

5.0 Future Work

5.1 Improvements on Object Detection Accuracy

Optimizing the accuracy of the live object detection system through improvements in datasets and training parameters is one major factor to be considered. This improvement on accuracy can be enabled through experimentation with various object detection models including but not limited to the YOLO models. An examples of such different models include:

- YOLOv8n-seg
- YOLOv7x
- YOLOv10n
- Detectron2
- EfficientDet
- RetinaNet
- Cascade R-CNN

Evaluating the performance of a diverse set of models with improved work on training parameters will allow us to evaluate the advantages of different models in different scenarios of live object detections, and help us determine the model which is best for our project's objective.

5.2 Dataset Development

While publicly available datasets such as TACO are highly valuable for the testing phase of the project, they are insufficient for real-world applications. Most publicly available datasets for litter are created with the intention that the object detection model will be utilizing a webcam in a sufficiently bright environment; the scenarios are significantly different for our project.

The development of a unique dataset for the project's objective have to be developed with several key factors in mind:

1. The drone will often have a bird's-eye view of the litter it is detecting
2. The background of the training images will need to be diverse with sole focus of these images needing to be pavements
3. A larger amount of categories requires more training images per category to evenly distribute the confidence-precision for all the categories
4. Ensure that the number of images does not exceed a limit beyond which biases can occur within the trained model

With current works on the dataset, the accuracy of the model is not yet at a desired range. We aim to utilize the *'ROBOFLOW'* website to organize and develop these custom datasets.

5.3 Autonomous Navigation & Flight-path Planning

Creating an autonomous navigation system for the drone is an essential component to scalability of litter audits. Through having the options to pre-plan flight paths the drones will be able to autonomously conduct litter audits without the need for manual control; allowing the need for a large number of drone pilots. Current research into the Parrot ANAFI's Olympe library shows resources required to accomplish these, however it will require us to develop a system to convert GPS coordinates into moveable control units (such as the ones used by the controller code system).

6.0 Conclusion

The primary objective of the project is to develop an autonomous litter audit system through integrating machine vision into a UAV, this objective was achieved through utilizing the pretrained YOLO (You Only Look Once) object detection model paired with custom datasets which was implemented into the Parrot Anafi drone. The completion of this objective means that the project has accomplished real-time video processing and accurate litter detection which is logged into an excel file.

The project showed that automating litter audits by using machine vision integrated drones is a feasible and scalable project. Although there are challenges with compatibility issues requiring us to operate the live system in a linux environment as well as limitations in hardware; we were able to bypass it and optimize the system to its limit with the hardware components that were available to us. With improvements in hardware components for this system will only improve the operational efficiency of this project.

The successful deployment of the litter audit system have the advantages of being faster, more efficient, and cost-effective compared to the manual litter audits currently being done. This system can help cities like Toronto to improve their litter management strategies, leading to cleaner environments and better resource allocations.

Future works to be carried out on the project would be to improve object detection accuracy; developing custom datasets; and integrating autonomous navigation system. With successful work on these factors in the future will ensure that the litter audit system will prove to be feasible and scalable.

7.0 Source

[1] M. Palmér, “Pynput Package Documentation,” pynput Package Documentation - pynput 1.7.6 documentation, https://pynput.readthedocs.io/en/latest/ (accessed August 2024).

[2] “Home,” Ultralytics YOLO Docs, https://docs.ultralytics.com/ (accessed August 2024).

[3] “What is opencv library?,” GeeksforGeeks, https://www.geeksforgeeks.org/opencv-overview/ (accessed August 2024).

[4] “CSV - csv file reading and writing,” Python documentation, https://docs.python.org/3/library/csv.html (accessed August 2024).

[5] “Pandas documentation,” pandas documentation - pandas 2.2.2 documentation, https://pandas.pydata.org/docs/ (accessed August 2024).

[6] “Olympe Documentation,” 7.7, https://developer.parrot.com/docs/olympe/index.html (accessed August 2024).

[7] Zeth, “Introduction,” Introduction - Inputs 0.5 documentation, https://inputs.readthedocs.io/en/latest/user/intro.html (accessed August 2024).