Imperial College London

Department of Aeronautics

Group Design Project

Autonomous Drone Delivery

Project RAST

ADD02 - Computer Vision

Supervisors: Dr Elnaz Naghibi, Dr Nigel MacCarthy
Department: Department of Aeronautics
Course: H401 MEng Aeronautical Engineering
Student: Vlad Marascu
CID: 01338522

Date: June 15, 2020

Abstract

This paper aims to present an overview of the research and implementation of Computer Vision and Machine Learning algorithms used for Project RAST (Rapid Aerial Search Technology). The related mission profile for this subgroup was split into two main tasks: outer-wall entrance points image recognition and labeling of windows and holes, and human identification once inside the premises.

The former task was initially approached from a Deep Learning perspective, yielding significant results for the windows recognition. However, since the computational costs exceeded the expectations and due to the fact that there is no specific 'hole in the wall' training image dataset available online, this approach was replaced by a more straight-forward OpenCV and Python script that was programmed for contours and edges detection, object classification, as well as area and center point calculation. The script can approximate the entry point's area in meters with an average absolute error of 12.8%, whilst producing numerical outputs that are easy to integrate within ROS nodes and topics. With the final addition of a frontal sonic sensor, the drone would be able to further distinguish if the detected entry points are gaps or not.

The latter task was achieved by implementing a pre-trained model that uses the Single Shot Multibox Detector (SSD) object detection framework and 'MobileNets' as the base convolutional neural network (CNN). This combination, SSD-MobileNet, provides a fast and efficient deep-learning based human detection algorithm, that is most suitable for computationally-limited devices. This script is able to detect human models in Gazebo with an average confidence of 64%; however, the tests conducted on real people yielded much higher accuracy values, of 95-100%, revealing the detector's potential to be used outside of the simulated environment.

In terms of structure, this paper aims to give ample motivation behind each task's chosen method in the *Literature Review* section, followed by a detailed description of the implemented algorithms and features in the *Code Implementation* section. The *Results* section presents the test scenarios done in the Gazebo simulation environment, as well as certain improvements for the codes' functionality and applicability in real-world. Finally, the *Discussion* and *Conclusion* sections summarize the obtained results and overview the chosen Computer Vision methods.

Acknowledgements

Firstly, I would like to thank my parents, Renata and Augustin-George, for their continuous support and for providing me with the opportunity to study at Imperial College London. I would also like to thank and express my gratitude towards my colleagues, without whom achieving this task would have not been possible. They provided great support and helped me strengthen my teamwork and communication abilities while undergoing this amazing engineering challenge. More so, I would like to show my appreciation towards my Flight Controls subgroup colleagues, whose help was crucial in successfully completing our task.

Secondly, I would like to thank the board of supervisors for their constructive feedback, advice and support during this project. Their interesting perspectives, questions and points of view pushed me past my limits, to further develop myself as an engineer, as well as helped me experience a real-world working environment.

I would also like to show my utmost gratitude to Dr Mirko Kovac, who made this interesting project possible and guided me towards becoming a more complete engineer and person.

Contents

Acknowledgements **i**

1 Introduction **1**

1.1 Motivation 1

1.2 Project overview 1

1.3 Tasks summary 2

2 Literature review **2**

2.1 Computer vision software 2

2.2 Human recognition 3

2.3 Entry points recognition 5

3 Code Implementation **6**

3.1 Human recognition script 7

3.2 Entry points detection and classification scripts 8

3.2.1 Main script 9

3.2.2 Calibration script 10

4 Results **11**

4.1 Human recognition results 11

4.2 Entry points detection results 11

5 Discussion and improvements **12**

6 Conclusion **12**

References **13**

Appendix **17**

A Human detection script **17**

B Entrance points detection scripts **19**

B.1 Main script 19

B.2 Calibration script (pre-flight) 21

C Results **24**

C.1 Gazebo human recognition results 24

C.2 Real human recognition results 24

C.3 Entry points detection: area measurement accuracy 24

List of Figures

1 Computer Vision sub-team block diagram 2

2 HOG descriptors example; each pixel from the 8x8 'cells' (green) has a numerical norm and direction 4

3 Final output of the HOG descriptors 4

4 SSD framework architecture, using VGG-16 as the *base network*; can be replaced by any other CNN 5

5 Custom-trained YOLOv3 window detector, video 6

6 Custom-trained YOLOv3 window detector, image 6

7 Contour and edge detection in OpenCV (expected result) 6

8 Example of classification based on the number of points: windows and holes are detected separately 9

9 Filters applied to a frame (img) from the video-stream and expected outputs; done in Gazebo environment 10

10 Drone camera-object assembly: $A_{p/m}$=calibration object; $A_{1p/m}$=actual (required) object 10

11 Tested main script outputs (center coordinates[p], area[m^2]) 10

12 Human recognition on Gazebo results with confidence; *Proceed=1* if detection made . 11

13 Calibration image (Type 1,$DC = 1m$). Detection example: Type 3 (right) & 4 (left) windows, $D'C = 6m$ 12

14 Sonic sensor implementation diagram: needed to classify detected entry points as open (gaps) or closed 12

15 Real human recognition results with confidence; *Proceed=1* if detection made. Note that various postures (sitting down) are detected, as well as different profiles (side,back view). 24

16 Detection example: *Type 2* window, $D'C = 2m$ 25

List of Tables

1 Comparison of popular base networks (CNNs), depending on version 5

2 Gazebo human recognition confidence results: average confidence is 64.70% 24

3 Testing results obtained on a set of Gazebo window types; *Type 1* is the calibration window 25

1 Introduction

1.1 Motivation

In today's world, where natural disasters such as floods have become increasingly frequent, unpredictable and severe [1], with the potential of affecting large urban areas and millions of people [2], the need of computerized assistance has arisen in order to tackle this issue more efficiently and provide much needed relief to the rescue forces. As described in the Pitt report [3], which focused on the 2007 UK floods, lack of search and rescue teams proved to be a significant problem in combating this type of natural disaster. Furthermore, since the last decade has been crucial in the hardware and software development of the UAV and autonomous vehicles sector, with most developed countries already exploiting drone technology for numerous applications like security, defence, mapping and communications [4], such technologies could potentially be implemented in a civilian rescue scenario, such as the one proposed by Project RAST. While the hardware advances in this regard are well known and documented, culminating with the development of highly powerful micro-processors and sensors, the advances in the software field are equally important, especially the current cutting edge research conducted in the Machine Learning and Deep Learning areas. These AI frameworks, which are the main object of this report, have particularly shown immense development in the last 5 years, as most of the scripts implemented in the scope of this project use recently-developed algorithms and models. Last but not least, another sector of recent development is represented by the inter-drone communication field, which allows a multitude of UAVs to communicate with each other and solve any task with increasing efficiency, especially in situations where time is vital.

Therefore, keeping these considerations in mind, a solution to this ever-growing problem represented by floods is the implementation and programming of a 'swarm' of drones that can take off from a centralized raft, search affected households and identify residents, thus greatly helping the rescue teams in their effort of saving lives.

1.2 Project overview

The main goals of this project can be summarized by the successful implementation of a network of 4 drones that can autonomously complete the following mission profile:

1. The rescue crew and 4 drones will start from the ground-station raft, where the rescuers will use a local map of the street to select the houses that need searching.
2. The drones will take-off from the raft, fly to the outside of a house and detect available entrance points (open windows, holes). Once inside, the drones will map the building in real time and avoid all obstacles, while the cameras are used to detect any humans.
3. Upon detecting any occupants, the drone is then required to drop a live video feed and audio camera in their vicinity and then return to the ground-station raft with enough information for the rescue team.
4. The rescue team can therefore plan and provide appropriate support for each household, based on the details sent by the drones. The ground station map will be updated with the searched houses and number of people found.

With this overview in mind, the Flight Controls team was assigned the following 3 main objectives: Computer Vision, Pathfinding and Inter-drone Communication, with specific tasks split accordingly between each sub-team. This report will mainly focus on the Computer Vision objectives, image recognition and deep learning algorithms used in order to complete this mission profile.

1.3 Tasks summary

In the overview of Project RAST's mission, the Computer Vision team was particularly tasked with implementing successful algorithms that can efficiently scan and detect any entrance points on the building's outer walls, such as open windows or holes, as well as with designing a method of detecting the occupants inside the premises. Therefore, this report will tackle both these problems separately and offer suitable solutions for each one of them in a detailed manner.

As logged on the Microsoft Teams Group and discussed in the weekly presentations with the supervisors, the tasks of the Computer Vision sub-team were the following:

- Starting with the frontal camera feed input and the drone hovering perpendicularly to the outer wall, detect any contours of entrance points in the building and label them (i.e. contours of windows and holes in the wall).
- Using the frontal ultrasound sensor and the position of the entrance point center, check if there is a gap in the wall. In the case of a window, check if the window is open or closed.
- Output the center point of the entrance to the Pathfinding team, as well as the area of the gap. Decide if the morphing drone is able to pass through the opening and output this as a Boolean variable (true or false) down the chain.
- Implement a human detection algorithm that is accessed once the drone is inside the building.

The block diagram in Figure 1 below summarizes the used inputs and transmitted outputs of the Computer Vision scripts, as described in this section.

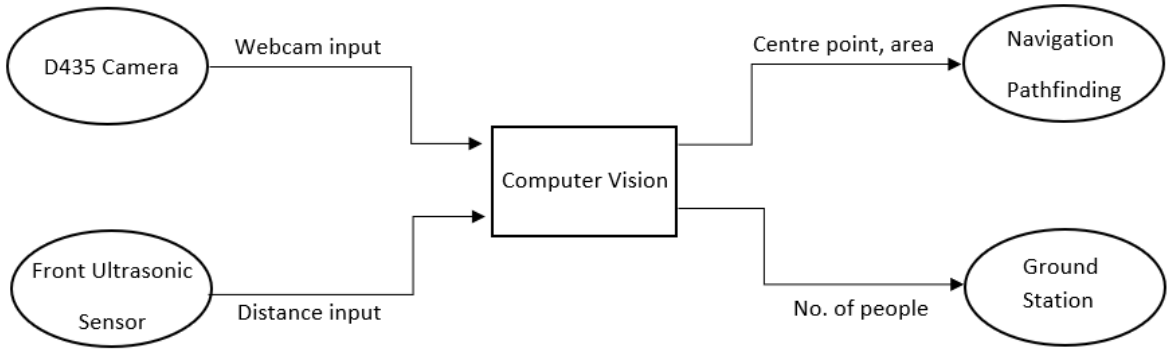

Figure 1: Computer Vision sub-team block diagram

2 Literature review

This section aims to present an overview of the research conducted while implementing the required algorithms and will therefore conclude both successful and failed attempts. Moreover, adequate reasoning and discussions will be attached at every choice made along the way and certain features of the mentioned approaches will be displayed for a better comparison.

2.1 Computer vision software

Starting off the discussion about computer vision, the first concept that comes to mind, regardless of the implemented object detection method, is the term of *image processing*, which can be done using

various software libraries. Considering the bulk code of Project RAST is done in Python and the elected simulation environment is Gazebo, a compatible such toolbox needs to be identified.

In order to fully understand the concept of image processing, some basic key definitions need to be introduced. Firstly, any image or frame from a video stream has to be stored as a Numpy array of pixels, *np.ndarray* [5][6], with coordinates starting from the top left corner of the image, similarly to a table. This is the standard representation of any multi-dimensional array in scientific Python and ensures further required processing can be done to the image, in the form of: colouring, filtering, segmentation (splitting an image into multiple regions) and edge detection [7]. Based on these image processing proprieties, two main toolboxes have been identified: Scikit-image and OpenCV.

Scikit-image is a Python-specific toolbox that can be used for image processing, as it imports images as pixelated Numpy arrays [7]. Since Numpy, an indispensable vectorization package [8], is also part of the greater Python data science ecosystem, there is an important interlink between the two libraries, making Scikit-image optimized and easy to use with Python. Moreover, this toolbox has wide documentation resources, a well commented source code [9] and works in line with Python-specific machine learning libraries such as Scikit-learn. It features numerous image processing algorithms (segmentation, edge detection, noise filtering and colour space editing), but it can only be operated in a Python scientific environment and has less performant real-time video detection tools, compared to its counterpart.

On the other side, OpenCV is a open-source computer vision library with numerous image processing algorithms that are mainly aimed for real-time applications and high efficiency [10]. While the functionality and provided algorithms are similar to Scikit-learn's, OpenCV can be integrated with almost all programming languages, including Python. Moreover, this library can take advantage of multi-core processing power, as well as overclocking the processor for higher performance [11], all while being compatible with the Gazebo simulation environment. In terms of efficiency and run-time, an extensive comparison between the 2 libraries can be accessed at [12], which uses the *timeit* Python tool for measuring computational time.

Finally, with the deciding factor being the lower computational time of optimised OpenCV, as well as the fact that it's better suited for real-time applications, the OpenCV library was chosen as the main image processing tool. Note that OpenCV is capable of implementing both a deep learning approach to object detection, using trained Neural Networks, as well as more linear approaches such as edge detection and object classification. The following sections will compare these 2 approaches to object detection for both tasks, adding appropriate discussion and comments.

2.2 Human recognition

Human recognition is one of the most difficult tasks related to object detection and has been a focal point of research in the last decade. For this task, two methods were deemed capable: the *Histograms of Oriented Gradients (HOG) + Support Vector Machine (SVM) Classifier* and the *Single Shot Multibox Detector (SSD) + MobileNet Neural Network* pre-trained human detector.

To start off with the former, HOG is an image feature extractor (a sequence of image processing techniques) that can be used to output specific object descriptors, which can then be fed to the Binary SVM Classifier, outputting the class of an object (i.e. human/not human). Introduced by Dalal and Triggs in 2005 [13], the HOG Descriptor is based on extracting the appearance and shape of an object, characterized by local intensity gradients and edge directions. In summary, the following sequence is used [14][15], starting from the idea that a greyscale image is a x-y pixel matrix of various intensities, represented by numbers from 0 (black) to 255 (white).

- Pre-processing: greyscale image is set to 64x128(1:2) resolution, suited for human detection[16].

- Image is divided into 8x8 pixel 'cells' and horizontal and vertical gradients (changes in x and y direction) are computed separately for each pixel, using the *Sobel Derivatives* filter in OpenCV.
- The magnitude and direction/angle of each pixel are computed and stored separately in 8x8 matrices. Then, the magnitude values are divided into 9 categories, based on angle (from 0 to 180 in increments of 20). Thus, a 9x1 frequency *histogram* is obtained for each 'cell'[14][17].
- Overall image is *normalized* in blocks of 16x16 pixels to reduce the effect of the local discrepancies in illumination and contrast, obtaining a 36x1 HOG descriptor for each block. Thus, the final HOG feature vector (3780x1) is obtained as the concatenation of all such descriptors[17].
- The HOG descriptor is then fed to the linear SVM Classifier, that outputs a binary result[15].

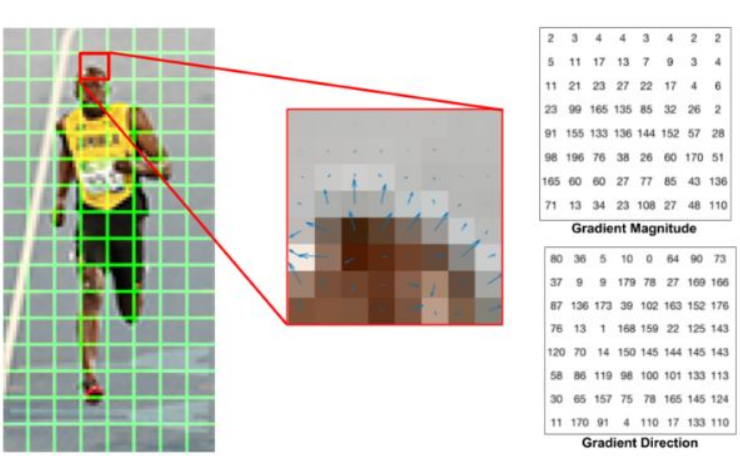

Figure 2: HOG descriptors example; each pixel from the 8x8 'cells' (green) has a numerical norm and direction

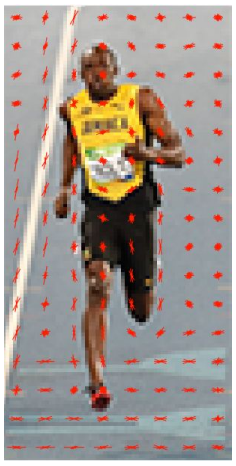

Figure 3: Final output of the HOG descriptors

In short, compared to simple edge detection functions, HOG can detect the gradient and orientation of the edges [14], as seen from Figures 2 and 3 [17]. Thus, the images passed to this feature extractor are required to have clear distinctive edges, and in the case of humans, the detection is based on the outline of a standing person, leading to errors when tested against various other postures such as sitting or leaning forward. Note that OpenCV has an included HOG descriptor function, as well as a pre-trained pedestrian SVM classifier at hand [18], making implementation easy and computationally inexpensive. However, upon testing, real-time detection with this method presented some lag, only reaching up to 5 FPS, depending on the CPU specifications and available memory.

On the other hand, more recent advances in the object detection field have culminated in replacing the HOG classifiers with the more accurate Convolutional Neural Network (CNN) based ones [19]. These object detection frameworks are of 2 types: single shot detectors (You Only Look Once- YOLO, Single Shot Multibox Detector- SSD) that prioritize speed, and two-stage detectors (Faster R-CNN) that focus on accuracy [19]. Moreover, these state-of-the-art frameworks use a *base network* (CNNs), or *feature extractor* [20], in the first layers of their architecture, as illustrated in Figure 4[21]. Thus, the choice of the base network, similar to the use of HOG, is crucial in the performance of the object detector. The proprieties of popular such CNNs, of varying size and number of layers, are thoroughly compared in [20] and [22] and summarized in Table 1. The consensus is that MobileNet [23] stands out, in particular for its lightweight architecture with very low memory usage, superior speed and comparable mean average precision (mAP) and classification accuracy, tested on the COCO[24] and ImageNet[25] data-sets respectively. Note that all mAP values trend to be twice as big for large objects (i.e. humans).

Since the real-time video processing requirement is crucial, and considering that the two-stage frameworks can only run at the order of 5-7 FPS, the single shot detectors were preferred for their speed.

Table 1: Comparison of popular base networks (CNNs), depending on version

Network	Layers	Size [MB]	Classification Accuracy, ImageNet[%]	mAP, SSD/COCO [%]
MobileNet	28	14-16	71	20
VGG-16	16	528	71.5	21
Inception	48	92-215	78-80	22
ResNet	18-101	98-232	75-78	24

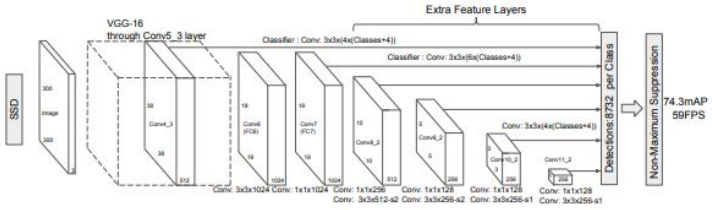

Figure 4: SSD framework architecture, using VGG-16 as the *base network*; can be replaced by any other CNN

These frameworks consider detection as a simple regression problem and use a single feed-forward CNN in order to predict classes; given an input image, they simultaneously learn the bounding box coordinates and classification probabilities of the classes [26]. They are described by a trade-off between speed, accuracy and computational limitations: YOLO [27] is faster and SSD [21] is more accurate, especially for larger objects (even overtaking Faster-RCNN [20]), while also requiring less processing power to run. One main difference is that YOLO is constrained to use the *Darknet* base network, combination not well-suited for computationally-limited devices, while SSD provides freedom of choice for the feature extractor. The latest YOLO iteration, v3, has reached a slightly higher mAP compared to SSD, due to its improved Darknet53 base network (77% ImageNet Accuracy [27]); however, its size and computational requirements make it unsuitable for our processor. As demonstrated by [28] and [29], the combination of a SSD framework and MobileNet base network provides the desired balance between speed (up to 25FPS) and accuracy (71%; mAP of 19.3% [23]) for computationally-constrained devices such as a drone.

Finally, considering both approaches and their limitations, it was opted to use the SSD detection framework and the MobileNet base network, mainly due to HOG's lack of accuracy for various human postures and YOLO's exceeding specifications. The chosen combination provides the best precision from the fast detectors category [20], and is able to run in real time. Note that for the MobileNet+SSD detection network, open source pre-trained weights using Caffe (72.7% mAP) are available and can detect 20 classes, including humans, making the implementation in OpenCV effortless [30].

2.3 Entry points recognition

Similarly to the previous section, 2 approaches were considered for the entrance points detection:

1. Contours and edges detection, using OpenCV integrated functions and filters: can determine the exact bounding box pixel coordinates and area, can classify based on shapes.

2. Custom-trained object detector (i.e. on the YOLOv3 framework): can classify based on images.

To start with the latter approach: whereas for the human detection open-source pre-trained weights are available for both SSD (able to detect 20 classes) and YOLOv3 (trained on COCO dataset, able to detect 80 classes) frameworks, the same cannot be said about entrance points such as windows and holes in the wall, as they are not included in each of the mentioned classes. It has to be

mentioned that YOLO9000 [31], a version of YOLOv2, is able to detect 9000 classes (trained on the WorldTree hierarchical database, combining both COCO and ImageNet datasets), but is highly inaccurate (16% mAP) and not compatible with OpenCV. Therefore, as seen from Figures 5 and 6, custom deep-learning training was successfully implemented using the YOLOv3 framework and the *Google Open Images V6+* dataset for windows [32], closely following the indications of [33] and [34] (for building the labeled windows image dataset used in training). Although this custom-detection network worked well for both images and webcam applications using OpenCV (see Figure 5), the training was stopped prematurely at 22/100 epochs (stages) due to GPU limitations, sacrificing accuracy. However, no such online dataset could be found and trained for holes, as they technically do not represent objects. Moreover, this tested approach was proven to be computationally expensive to run in the Python/OpenCV environment, needing a GPU, and the resulting bounding boxes were positioned inaccurately (see Figure 6), leading to errors in the center point and area computations.

Figure 5: Custom-trained YOLOv3 window detector, video

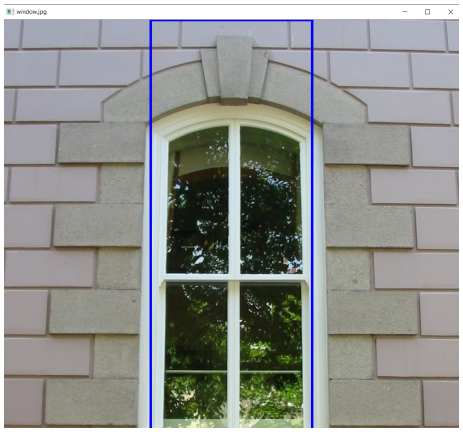

Figure 6: Custom-trained YOLOv3 window detector, image

Figure 7: Contour and edge detection in OpenCV (expected result)

On the other hand, even though the former approach seems more linear in theory, its features perfectly match the required outputs. Working with existing OpenCV functions and filters in order to extract contours and classify various objects presents many benefits, as illustrated by Figure 7: the bounding boxes are perfectly centered, center point and area values are exact (determined in pixels) and classification can be obtained by considering the shape of the object (number of points in the contour). Moreover, this approach requires little processing power and is perfectly fit to be implemented on a drone. It has to be mentioned that several assumptions are made in regards to this method: the entry points should have distinctive frames (colour gradients) compared to the wall, to ensure edge detection is possible, objects should have distinctive shapes in order to be classified and the drone should fly relatively close and normal to the wall, to avoid parallax errors.

To conclude, based on the enumerated considerations, the chosen method for entrance points recognition is the *Contours and Edges Detection*, predominantly for its low CPU running requirements, ability to detect both windows and holes, and accurate area and center point determinations, that represent the Computer Vision team's outputs (see Figure 1).

3 Code Implementation

This section aims to explain and demonstrate how the 2 algorithms used for each task were implemented, as discussed and chosen in *Literature Review*: the final choices, in terms of implementation, for each task are: the *SSD framework + MobileNet CNN* Deep Learning based approach for human recognition, followed by an OpenCV *Contours and Edges Detection* method for entrance point identification and classification. Moreover, certain features present in lines of code will be highlighted

properly for review. All pieces of code are thoroughly commented for good-practice and can be accessed from `https://github.com/mirkokovac/GDP2020/tree/master/Computer_Vision`.

3.1 Human recognition script

As reminded above and mentioned in Section 2.2, the chosen method for human detection is a combination of the Single Shot Multibox Detector (SSD) framework and MobileNet base network. This real-time detector achieves good FPS (around 15-25 in testing), accuracy and precision, while being perfectly suited for its intended application on a drone, as presented in [29]. The overall layer configuration is illustrated in Figure 4, with the mention that the first network (feature extractor layers), represented there by the VGG-16 base network, is replaced by the chosen MobileNet CNN.

For this model, open source pre-trained weights are available and can detect 20 classes, including humans, making the implementation in OpenCV effortless. In more detail, the model that was employed is a Caffe implementation of the original TensorFlow method seen in the original MobileNets paper [23], and the model's weights were trained by Github user *chuanqi305* [30] (access at: `github.com/chuanqi305/MobileNet-SSD`): fine-tuned on the VOC Pascal database after initially being trained on COCO [24], the model reached up to a 72.7% mAP, as presented in [30]. The implementation of the model's pre-trained weights in the Python and OpenCV environment, following the ideas of [35], is attached in Appendix A and can be accessed at: `https://github.com/mirkokovac/GDP2020/blob/master/Computer_Vision/human_detection_custom.py`.

Detailed descriptions of the most important features of the 'human_detection_custom.py' Python script are presented below; however the code is thoroughly commented, as seen in Appendix A.

• Model classes are predefined, as taken from [30], and an ignored classes set (except 'person') is created for further use. Model is loaded using the OpenCV *dnn* function as *network*.

```
CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
           "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
           "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
           "sofa", "train", "tvmonitor"]
# Ignored classes set, everything except PEOPLE
IGNORE = set(["background", "aeroplane", "bicycle", "bird", "boat",
              "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
              "dog", "horse", "motorbike", "pottedplant", "sheep",
              "sofa", "train", "tvmonitor"])

# Load pre-trained model from the same folder using the OpenCV dnn function (Ref: chuanqi305)
print("[INFO] loading model...")
network = cv2.dnn.readNetFromCaffe(
    "MobileNetSSD_deploy.prototxt.txt", "MobileNetSSD_deploy.caffemodel")
```

• Video stream is initialized using the *VideoCapture* OpenCV function (0=webcam) and a loop that analyses each frame separately is created. Frames are grabbed and their shape stored, then each frame is transformed into a 4D blob that is used as a feed-forward input in the dnn-imported SSD+MobileNet network (*network*), thus obtaining the detected *targets* (multiple for each frame).

```
# Initialize video stream using OpenCV
print("[INFO] starting video stream...")
capture = cv2.VideoCapture(0)

# Loop over each frame from the video stream
while True:
    grabbed, frame = capture.read()  # Grab each frame
    # Determine the frames shape, extract width and height for later
    (h, w) = frame.shape[:2]
    # Transform to 4D blob (resize, crop from center, substract mean values, scale values by scalefactor)
    blob_4d = cv2.dnn.blobFromImage(cv2.resize(frame, (315, 315)),
                                    0.007843, (315, 315), 115.15)

    # Feed-forward the blob through the SSD+Mobilenet network and obtain the detections, our "targets"
    network.setInput(blob_4d)
    targets = network.forward()
```

• Loop over each detection: determine where and what objects are and, depending on the confidence (probability) value (default 20%), decide if label + rectangular box are drawn for highlighting. If confidence $\leq 20\%$ threshold, object is ignored, filtering any uncertainties. Lastly, classes defined in the *IGNORE* set are skipped, leaving the only detectable class as 'person'.

```
    for i in np.arange(0, targets.shape[2]):
        # Extract the probability (confidence) associated with the detection (targets=detections)
        probability = targets[0, 0, i, 2]
        # Ensure probability is greater than the threshold (0.2), can be varied
        if probability > 0.2:
            # Index of the class label from 'targets'
            index = int(targets[0, 0, i, 1])
            # If the class label is in the set of ignored classes, skip detection
            if CLASSES[index] in IGNORE:
                continue
```

• Find x,y coordinates of the bounding box for each detected target (start=top left, end=bottom right in pixels), and create label name 'person' for display. Boolean variable is initialized for output: if a person is detected, 1 is returned and the drone will drop the 2-way camera in their vicinity.

```
            bounding_box = targets[0, 0, i, 3:7] * np.array([w, h, w, h])
            (X_start, Y_start, X_end, Y_end) = bounding_box.astype("int")

            # Create label name for display
            label = "person"

            # Create bool variable that is 1 if humans are detected, 0 if no humans are derected (REQUIRED OUTPUT)

            varbool = 0
            if label == "person":
                varbool = 1
            else:
                varbool = 0
```

• Draw rectangle bounding box over the detected object in each frame and display the label of the class ('person'), as well as the boolean variable ('Proceed: 1' for human detected, camera is to be dropped). Output each frame from the video-stream as a real-time detection method. The boolean (1/0) is also printed in the terminal and the script can be stopped by pressing *'q'*. Average FPS values obtained in testing (expecting to be less for the drone) are around 15-25 FPS.

```
            cv2.rectangle(frame, (X_start, Y_start), (X_end, Y_end),
                          (255, 0, 0), 2)

            # Display the label of the class (person) if detected
            cv2.putText(frame, label + str(probability*100), (X_start, Y_start - 25),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
            # Display bool variable (1 for proceed/ drop camera in vicinity)
            cv2.putText(frame, "Proceed:"+str(varbool), (X_start, Y_start - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 1)

    # Display the output frame as a videostream
    cv2.imshow("Frame", frame)
```

The script was then tested using a webcam as camera input, and the pictures in Appendix C.2, Figure 15 illustrate the expected results of the simulations. To be noticed that various postures are easily detected, and the confidence values range from 95% to 100%, revealing this script's accuracy and potential to be used both inside and outside of the Gazebo simulation environment. Since the model weights were trained using real-life pictures of humans, it is expected that the script will perform better in a real-world scenario; these results will be discussed and compared in Section 4.

3.2 Entry points detection and classification scripts

As reminded above and mentioned in Section 2.3, the chosen method for entry points (windows, holes) detection is a Python and OpenCV implementation that can detect contours and edges of objects, classify them and calculate their areas accordingly. This method only uses the Python environment and OpenCV pre-defined functions and filters, resulting in low-memory and processing power requirements, whilst being able to run in real-time.

This implementation contains 2 scripts, whose features are detailed in the following subsections.

1. **Main entrance points detection script** (*'Shapes from webcam.py'*): uses the *calibration.py* script and returns the detected object types (windows/holes), their area in pixels and the computed area in meters, deciding if the morphing drone can enter through the detected point.
2. **Calibration script**(*'Calibration.py'*): uses an input picture taken by the same camera/resolution as the one of the drone; the input picture must contain an object of known area (in meters: A_m[m]) and known distance from camera to the respective object (DC[m]).

3.2.1 Main script

The *getContours* function takes an input image/frame from video-stream (img) and returns an image/frame (imgContour) that contains: object class, bounding rectangle showing the position, centre point coordinates and the area in pixels/meters, outputs used by the Pathfinding subgroup.

• The OpenCV functions used below extract the external contours from the image, as well as each contour's area in pixels (iterating). Each contour area is compared to a set threshold value, *areamin*, that filters out the small detected objects, thus reducing noise. Only the relevant contours are drawn.

```
def getContours(img,imgContour):
  # Extract EXTERNAL contours from the input image, img=Dilated_img, when the function will be called
  contours, unused_var = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)# simple-less no of points
  # Draw external contours over the output image, imgContour will be a copy of img initially
  cv2.drawContours(imgContour,contours,-1,(255,0,255),3)
  # Iterate over all contours (cnt)
  for cnt in contours:
    # Calculate area of each contour (in pixels)
    area = cv2.contourArea(cnt)
    # Set an arbitraty minimum area, from Area trackbar in 'Variables' tab
    areamin=cv2.getTrackbarPos("Area","Variables")
    # Any contour with less than areamin will be ignored (reduces unwanted noise)
    if area > areamin:
      # If area is large enough, draw the contours over the output image, imgContour
      cv2.drawContours(imgContour,cnt,-1,(255,0,255),3)
```

• The following pre-defined functions extract the length of the relevant contours and approximate their shapes by outputting their number of points. The coordinates of the minimum-area bounding rectangle are calculated, and a bounding box is drawn to encapsulate the object: w and h are the width and height of the object, and (x,y) are the start coordinates of the bounding box in pixels, as illustrated in Figure 10. Thus, for each relevant object, the area A_{1p}(pixels) and center point (pixels) are displayed on the outputted image (imgContour).

```
      # Compute the contour perimeter / length (True=only closed contours considered)
      perimeter=cv2.arcLength(cnt,True)
      # Approximate the shape of object, contours (cnt) as input, resolution=0.01*length, True=only closed contours
      shape_approximation = cv2.approxPolyDP(cnt, 0.01*perimeter, True)# contains a number of points, used in
          determining shapes: length(shape_approximation)
      # Calculates minimum upright bounding rectangle for the specific point set (object)- outputs start coordinates
          (x,y) (top left) and rectangle dimensions (w,h)
      x,y,w,h=cv2.boundingRect(shape_approximation)
      # Draw bounding box over object: input the start coords (x,y) and end coords (x+w,y+h)
      cv2.rectangle(imgContour,(x,y),(x+w,y+h),(0,255,0),3)
```

• Using the *approxPolyDP* function, the number of points of each object is extracted (*len(shape)*) and a classification based on shape is made: if the object has between 4 or 5 points (approximately rectangular), its class is 'window', otherwise, if the object has 6 or more points, it is a 'hole' (circular, elliptic, irregularly shaped), see Figure 8. Labels and center points are printed for display.

```
      if len(shape_approximation)>=4 and len(shape_approximation)<=5:
        cv2.putText(imgContour, "Window", (x, y + 60), cv2.FONT_HERSHEY_COMPLEX, .5,
                    (0, 0, 255), 1)
        # Draw center point of window/rectangle in red
        cv2.circle(imgContour, (x+int(w/2),y+int(h/2)), radius=0, color=(0, 0, 255), thickness=5)
```

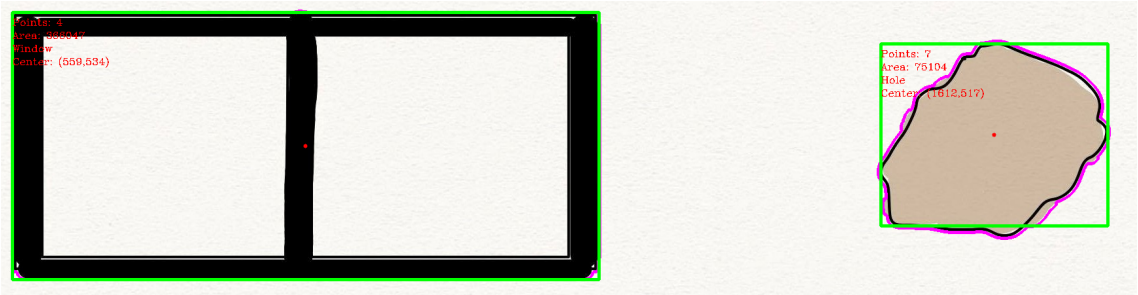

Figure 8: Example of classification based on the number of points: windows and holes are detected separately

Each frame from the video-stream is grabbed (img=frame) and read separately, and imgContour is initialized as a copy of the video-stream frames. The suite of functions used on the extracted frames (img) are filters, as better illustrated in Figure 9. Using the final dilated image (Dilated_img) as input, the *getContours* function is used to output each object's type, bounding box, area A_{1p} (pixels) and center point location (pixels), all displayed over the original video-stream frames (imgContour).

```
while True:
    grabbed, img = capture.read() # grab the frames and store them as img
    imgContour = img.copy() # initialize the output imgContour as a copy of img
    # Apply Gaussian Blur Filter, (9,9) kernel to smooth the image and reduce noise
    Blur_img = cv2.GaussianBlur(img, (9, 9), 1)
    # Convert to Greyscale
    Gray_img = cv2.cvtColor(Blur_img, cv2.COLOR_BGR2GRAY)
    # Set treshold values for Canny Edge Detector, using trackbars in the 'Variables' window
    thresh_1 = cv2.getTrackbarPos("Thresh_1", "Variables")
    thresh_2 = cv2.getTrackbarPos("Thresh_2", "Variables")
    # Apply Canny Edge Detection filter with variable thresholds, tested defaults are 172,140, can be varied
    Canny_img = cv2.Canny(Gray_img,thresh_1,thresh_2)
    # Creating convolution kernel used for Dilation
    kernel = np.ones((5, 5))
    # Dilating the Canny Edges in order to accentuate the shape, iterations=1 for most accurate object area
    Dilated_img = cv2.dilate(Canny_img, kernel, iterations=1)
    # Use defined function to draw the boxel, labels and areas using the Dilated_img as input, for accurate
        detection, and the copy of img (imgContour) as the output image. Thus, the boxes, labels and areas will
        appear on the original un-filtered videostream live.
    getContours(Dilated_img, imgContour)
    # Display original videostream frames with contours, boxes, labels and areas
    cv2.imshow("Contours", imgContour)
```

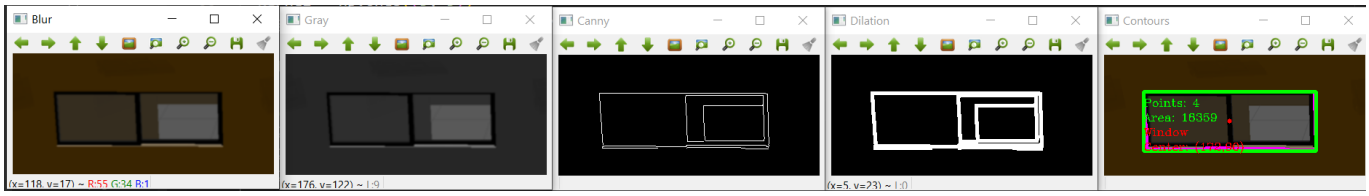

Figure 9: Filters applied to a frame (img) from the video-stream and expected outputs; done in Gazebo environment

3.2.2 Calibration script

This script is used in addition to the main code: it takes the input of a picture with known calibration object dimensions in meters (A_m, DC) and it is accessed by the main code automatically. Its main use is in transforming the obtained object areas in pixels A_{1p} (from the *cv2.contourArea* function in the main script) to areas in meters A_{1m}, as required in order to check if the morphing drone fits through the entry point. The *'calibration.py'* script has an identical structure to the main script, but its only needed output is the calibration object's area in pixels (A_p). The following steps are used:

- Knowing the calibration object area in pixels (A_p), as returned by this script, as well as its actual area in meters (A_m), we can compute the PPM (pixel per meter) ratio at a certain known distance from drone (camera) to wall (object), denoted DC, as $PPM = A_p/A_m$. This is implemented in *'calibration.py'* between Lines 65-76 (see Appendix B.2).
- Observing that if the camera is moved forwards (DC decreased to D'C), the area in pixels (A_p) increases proportionally, followed by a subsequent increase in PPM (to PPM'), these quantities are inversely proportional: $PPM' = PPM * DC/D'C$, where PPM' is the ratio at a distance D'C, for the required object, whose unknown area in meters is denoted by A_{1m}.
- Assuming that D'C, the distance from the drone to the unknown object (A_1) is outputted by the frontal sonic sensor, PPM' can be computed, and the required object area in meters is calculated: $A_{1m} = A_{1p}/PPM'$, where A_{1p} is obtained in the main script. This is implemented in the main script between Lines 64-74 (see Appendix B.1) and displayed accordingly.

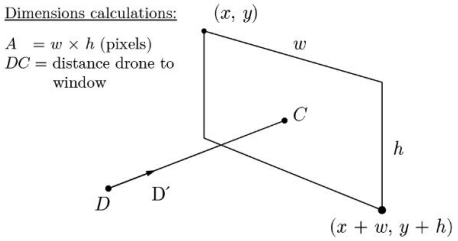

Figure 10: Drone camera-object assembly: $A_{p/m}$=calibration object; $A_{1p/m}$=actual (required) object

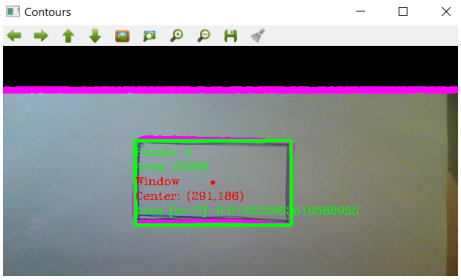

Figure 11: Tested main script outputs (center coordinates[p], area[m^2])

4 Results

4.1 Human recognition results

The human recognition script was tested both on the provided Gazebo simulation environment, with a simulated person, and in real-life, using a webcam camera input. The confidence values, seen next to the label of each prediction, were tabulated in Appendix C.1 for comparison. Using Gazebo, 10 captures were extracted, as well as their confidence values (see Appendix C.1, Table 2). Note that the simulated person was rotated and various zoom levels were used to test the script. Based on Table 2, the average confidence of human detection in the Gazebo environment is 64.70%, while the average percent error is 17.46%, highlighting wide variations in the obtained results, with some detections reaching 84% accuracy. This can be attributed to the different zoom levels used (better performance upon zooming in) and human positioning (better performance for frontal view compared to profile view). The results are summarized in Figure 12 below, outlining these variations in performance.

Figure 12: Human recognition on Gazebo results with confidence; *Proceed=1* if detection made

In comparison to the Gazebo environment, the tests conducted on real humans showed huge improvements in confidence, in the range of 95%-100%, and this is likely due to the fact that the SSD+MobileNet model was trained using pictures of real people. The results of these tests are attached in Appendix C.2, Figure 15: to be noted that various postures (sitting down) and profiles are easily detected, making this script ideal for the drone's real-world task of detecting survivors.

4.2 Entry points detection results

The entry points detection script was mainly tested against a set of Gazebo-environment images, containing 4 types of windows with different areas. To test the script's accuracy in calculating the window areas in meters, a picture of a *Type 1* window was selected as the calibration image (Figure 13), with a known window area $A_m = 0.22m^2$ and drone-wall distance $DC = 1m$, as highlighted in Appendix C.3 Table 3. Several other images of the *Type 2-4* windows, from various D'C distances, were inputted in the main script, and their areas were computed in meters, as seen in Table 3 and Figure 13 example. Note that all simulated window types (2-hinged, sliding and awning) were easily detected by the script, as illustrated in Figure 13 and Appendix C.3- Figure 16.

Comparing the obtained areas with the actual values provided by the Simulations subgroup, as seen in Table 3, the percentage error for the conducted tests averages at 12.8%, with some sources of errors being parallax (slightly angled pictures) and different picture resolutions. The most important source of errors is the fact that, due to hardware limitations, the drone could not be fully integrated and, therefore, the drone-wall distances (D'C) were mere approximations. However, in the case of a fully integrated drone, with a working frontal sonic sensor that outputs the exact drone-window distances (DC, D'C), it is expected that these percentage errors drop significantly.

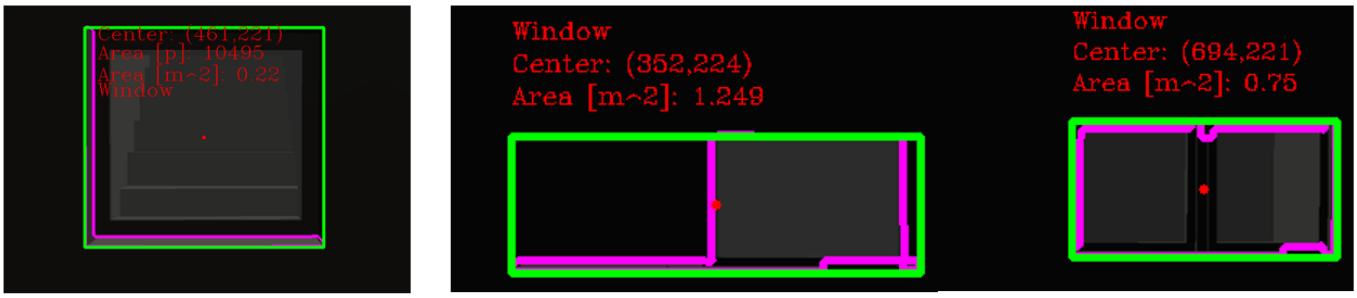

Figure 13: Calibration image (Type 1,$DC = 1m$). Detection example: Type 3 (right) & 4 (left) windows, $D'C = 6m$

5 Discussion and improvements

Although both the human and entry points detection scripts have been tested on the Gazebo environment and work well for their designed tasks, with errors mainly coming as a cause of the simulation environment and the lack of full integration, one important feature has to be discussed: the *frontal sonic sensor*. The idea behind this addition is to allow the drone to further classify if a detected entry point (window) is open or closed, using a comparison of distances outputted by this sensor. Although the ultrasonic sensors emit acoustic beams in a cone [36][37] of varying angle, it is now approximated that the ultrasonic pulse is a straight line to the target (i.e. drone is very close to the wall). Therefore, 2 distances can be measured by this sensor: one from the drone to the center of the window (center coordinates known from detection script, drone can be aligned to the window center by equating these coordinates to the global center of the image), and one from the drone to the wall, by shifting the drone an arbitrary distance, as illustrated in Figure 14. The distances are then compared: if the first one is larger (drone-window center), it means that the object (i.e. window) is open (gap) and the drone can therefore pass through the targeted item.

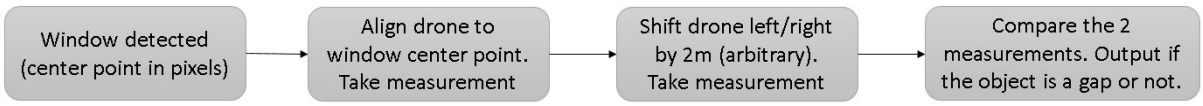

Figure 14: Sonic sensor implementation diagram: needed to classify detected entry points as open (gaps) or closed

An important discussion topic is the ROS nodes integration of these scripts, combined with the use of the sonic sensor. As highlighted in Figure 14, the drone shifts in order to take both measurements and thus, full integration by ROS topics is required to simultaneously access the drone's dynamics and object detection script. On the other hand, as outlined in Figure 1, the human detection script's output is a Boolean variable and the outputs of the object detection code (area, center) are stored as numeric variables, resulting in an effortless integration between all the drone's scripts within ROS.

6 Conclusion

To conclude this report, 2 different Computer Vision algorithms were employed separately for human and object classification. While the human detector reaches up to 84% confidence in Gazebo, it is expected that this value jumps to over 95% in a real-world scenario. For the entry points detection script, accurate object areas in meters and center points are obtained; however, due to missing hardware resources that resulted in the lack of full integration within ROS, the sonic sensor's role was only added as a future improvement. Finally, it is expected that, upon full integration, these scripts work and produce outputs in tandem with the *'down the chain'* requirements. Overall, it can be stated that the research and implementation of the described Computer Vision algorithms was a success, more so in the ample overview and operationality of Project RAST.

References

[1] R. Pidcock.
How much flooding is in the UK's future? A look at the IPCC report.
Last accessed on: 10/06/2020.
2014.
URL: https://www.carbonbrief.org/how-much-flooding-is-in-the-uks-future-a-look-at-the-ipcc-report.

[2] Energy and Climate Intelligence Unit.
Flood risk and the UK.
Last accessed on: 10/06/2020.
URL: https://eciu.net/analysis/briefings/climate-impacts/flood-risk-and-the-uk.

[3] Fire Brigades Union.
Inundated: The lessons of recent flooding for the fire and rescue service.
Last accessed on: 10/06/2020.
2015.
URL: https://www.fbu.org.uk/publication/inundated-lessons-recent-flooding-fire-and-rescue-service.

[4] Ph.D. A. R. Jha.
Theory, Design, and Applications of Unmanned Aerial Vehicles.
Last accessed on: 10/06/2020.
2016.
URL: https://books.google.ro/books?id=guGVDQAAQBAJ.

[5] Antony Paulson Chazhoor.
Image processing using Scikit-Image.
Last accessed on: 10/06/2020.
URL: https://towardsdatascience.com/image-processing-using-scikit-image-cb57ce4321ed.

[6] Scipy Lectures.
Scipy lecture notes.
Last accessed on: 10/06/2020.
URL: http://scipy-lectures.org/index.html.

[7] Stefan van der Walt et al.
Scikit-image: Image processing in Python.
Last accessed on: 10/06/2020.
2014.
URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4081273/.

[8] NumPy Documentation.
Last accessed on: 10/06/2020.
URL: http://numpy.org.

[9] Scikit-image vs OpenCV.
Last accessed on: 10/06/2020.
URL: https://medium.com/@hashinclude/scikit-image-vs-opencv-a2ce6e9b23d1.

[10] OpenCV documentation.
Last accessed on: 10/06/2020.
URL: https://opencv.org/.

[11] Stackshare.

OpenCV vs Scikit-image.
Last accessed on: 10/06/2020.
URL: https://stackshare.io/stackups/opencv-vs-scikit-image.

[12] Modesto Mas.
Python image processing libraries performance: OpenCV vs Scipy vs Scikit-Image.
Last accessed on: 10/06/2020.
2015.
URL: https://mmas.github.io/python-image-processing-libraries-performance-opencv-scipy-scikit-image.

[13] Navneet Dalal and Bill Triggs.
Histograms of Oriented Gradients for Human Detection.
In IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005.
2005.
URL: https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf.

[14] Aishwarya Singh.
Feature Engineering for Images: A Valuable Introduction to the HOG Feature Descriptor.
Last accessed on: 10/06/2020.
2019.
URL: https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/.

[15] M.Kachouane et al.
HOG Based fast Human Detection.
In the 24th International Conference on Microelectronics.
2012.
URL: https://arxiv.org/ftp/arxiv/papers/1501/1501.02058.pdf.

[16] Carlo Tomasi.
Histograms of Oriented Gradients.
Last accessed on: 10/06/2020.
URL: https://www2.cs.duke.edu/courses/fall15/compsci527/notes/hog.pdf (visited on 12/02/2017).

[17] Satya Mallick.
Histogram of Oriented Gradients.
Last accessed on: 10/06/2020.
2016.
URL: https://www.learnopencv.com/histogram-of-oriented-gradients/.

[18] OpenCV Org.
cv::HOGDescriptor Struct Reference.
Last accessed on: 10/06/2020.
URL: https://docs.opencv.org/master/d5/d33/structcv_1_1HOGDescriptor.html.

[19] Ankit Sachan.
Guide to Object Detection using Deep Learning: Faster R-CNN,YOLO,SSD.
Last accessed on: 10/06/2020.
URL: https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/.

[20] Jonathan Huang et al.
Speed/accuracy trade-offs for modern convolutional object detectors. Google Research.
2017.
URL: https://arxiv.org/pdf/1611.10012.pdf.

[21] Wei Liu et al.
SSD: Single Shot MultiBox Detector.

In European Conference on Computer Vision.
2016.
URL: https://arxiv.org/pdf/1512.02325.pdf.

[22] Keras Applications.
Last accessed on: 10/06/2020.
URL: https://keras.io/api/applications/.

[23] Andrew G. Howard et al.
MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.
2017.
URL: https://arxiv.org/pdf/1704.04861.pdf.

[24] Common Objects in Context (COCO) Database.
Last accessed on: 10/06/2020.
URL: http://cocodataset.org/#home.

[25] ImageNet Database.
Last accessed on: 10/06/2020.
URL: http://www.image-net.org/.

[26] Adrian Rosebrock.
YOLO object detection with OpenCV.
Last accessed on: 10/06/2020.
2018.
URL: https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/.

[27] Ali Farhadi Joseph Redmon.
YOLOv3: An Incremental Improvement.
2019.
URL: https://pjreddie.com/media/files/papers/YOLOv3.pdf.

[28] Asifullah Khan et al.
A Survey of the Recent Architectures of Deep Convolutional Neural Networks.
In Artificial Intelligence Review.
2020.
URL: https://arxiv.org/ftp/arxiv/papers/1901/1901.06032.pdf.

[29] Muhammad Faique Shakeel et al.
Detecting Driver Drowsiness in Real Time through Deep Learning based Object Detection.
URL: https://www.researchgate.net/publication/333627508_Detecting_Driver_Drowsiness_in_Real_Time_Through_Deep_Learning_Based_Object_Detection.

[30] Github user *chuanqi305*.
Caffe implementation of Google MobileNet-SSD detection network, with pretrained weights.
URL: https://github.com/chuanqi305/MobileNet-SSD.

[31] Ali Farhadi Joseph Redmon.
YOLO9000: Better, Faster, Stronger.
2016.
URL: https://arxiv.org/pdf/1612.08242.pdf.

[32] Google Open Images Dataset V6.
Last accessed on: 10/06/2020.
URL: https://storage.googleapis.com/openimages/web/visualizer/index.html?set=train&type=detection&c=%2Fm%2F0d4v4.

[33] Rokas Balsys.
Training a custom YOLOv3 object detector.

2019.
URL: https://pylessons.com/YOLOv3-custom-training/.

[34] Vittorio Mazzia.
OIDv4 ToolKit: Download and visualize single or multiple classes from the Open Images V6 Dataset.
2019.
URL: https://github.com/EscVM/OIDv4_ToolKit.

[35] Adrian Rosebrock.
Real-time object detection with deep learning and OpenCV.
2017.
URL: https://www.pyimagesearch.com/2017/09/18/real-time-object-detection-with-deep-learning-and-opencv/.

[36] Tracy Molnar.
Understanding the Ultrasonic Sound Cone.
Last accessed on: 10/06/2020.
2014.
URL: https://blog.pepperl-fuchs.us/blog/bid/335473/Understanding-the-Ultrasonic-Sound-Cone.

[37] EGE Elektronik.
Ultrasonic sensors- Technology and application.
Last accessed on: 10/06/2020.
URL: https://ege-elektronik.com/en/products/temperature-pressure-ultrasonic/application_ultrasonic.html?m=1378769948&.

A Human detection script

```
# Created by Vlad Marascu for GDP ADD02 Group 2020
# 04/06/2020

# Script to be run upon entering the house, uses frontal camera as webcam input, returns bool variable 1 if PERSON is found

import numpy as np
import cv2

# List of class labels MobileNet+SSD framework was trained to detect
# Trained by chuanqi305: trained on COCO dataset, fine-tuned on PASCAL VOC, reaches a 72.7% mAP

CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
           "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
           "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
           "sofa", "train", "tvmonitor"]
# Ignored classes set, everything except PEOPLE
IGNORE = set(["background", "aeroplane", "bicycle", "bird", "boat",
              "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
              "dog", "horse", "motorbike", "pottedplant", "sheep",
              "sofa", "train", "tvmonitor"])

# Load pre-trained model from the same folder using the OpenCV dnn function (Ref: chuanqi305)
print("[INFO] loading model...")
network = cv2.dnn.readNetFromCaffe(
    "MobileNetSSD_deploy.prototxt.txt", "MobileNetSSD_deploy.caffemodel")

# Initialize video stream using OpenCV
print("[INFO] starting video stream...")
capture = cv2.VideoCapture(0)

# Loop over each frame from the video stream
while True:
    grabbed, frame = capture.read()  # Grab each frame
    # Determine the frames shape, extract width and height for later
    (h, w) = frame.shape[:2]
    # Transform to 4D blob (resize, crop from center, substract mean values, scale values by scalefactor)
    blob_4d = cv2.dnn.blobFromImage(cv2.resize(frame, (315, 315)),
                                    0.007843, (315, 315), 115.15)

    # Feed-forward the blob through the SSD+Mobilenet network and obtain the detections, our "targets"
    network.setInput(blob_4d)
    targets = network.forward()

    # Loop over each detection: determine where and what objects are and if label+box are drawn, depending on confidence value selected
    for i in np.arange(0, targets.shape[2]):
        # Extract the probability (confidence) associated with the detection (targets=detections)
```

```
47          probability = targets[0, 0, i, 2]
48          # Ensure probability is greater than the threshold (0.2), can be
                varied
49          if probability > 0.2:
50              # Index of the class label from 'targets'
51              index = int(targets[0, 0, i, 1])
52              # If the class label is in the set of ignored classes, skip
                    detection
53              if CLASSES[index] in IGNORE:
54                  continue
55              # Find x,y start/end coords of the bounding box for each object
                    as: X_start, Y_start, X_end, Y_end (start is top left, end is
                    bottom right in pixel coords)
56              bounding_box = targets[0, 0, i, 3:7] * np.array([w, h, w, h])
57              (X_start, Y_start, X_end, Y_end) = bounding_box.astype("int")
58
59              # Create label name for display
60              label = "person"
61
62              # Create bool variable that is 1 if humans are detected, 0 if no
                    humans are derected (REQUIRED OUTPUT)
63
64              varbool = 0
65              if label == "person":
66                  varbool = 1
67              else:
68                  varbool = 0
69              # Draw bounding rectangle using the start/end coords extracted
70              cv2.rectangle(frame, (X_start, Y_start), (X_end, Y_end),
71                            (255, 0, 0), 2)
72
73              # Display the label of the class (person) if detected
74              cv2.putText(frame, label + str(probability*100), (X_start,
                    Y_start - 25),
75                          cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
76              # Display bool variable (1 for proceed/ drop camera in vicinity)
77              cv2.putText(frame, "Proceed:"+str(varbool), (X_start, Y_start -
                    10),
78                          cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 1)
79
80      # Display the output frame as a videostream
81      cv2.imshow("Frame", frame)
82      # Print bool variable in terminal
83      # print(str(varbool))
84      # Break upon pressing 'Q'
85      if cv2.waitKey(1) & 0xFF == ord('q'):
86          break
87  # Release frames upon breaking/stop
88  cv2.destroyAllWindows()
89  capture.release()
```

B Entrance points detection scripts

B.1 Main script

```
# Created by Vlad Marascu for GDP ADD02 Group 2020
# 04/06/2020

# Main entrance points detection script: uses the calibration.py script and returns the detected object types (windows/holes), the area in pixels and the computed area in meters.

import cv2
import calibration # load calibration.py from the same folder first to obtain PPM at a set distance
import numpy as np

# Initialize videostream
#Frame width and height can be modified depending on need with cap.set function

# Each frame can be extracted separately from the capture variable (the videostream)
capture = cv2.VideoCapture(0)

# 'emptyFunction' function defined to be used by the 'Variables' Trackbar

def emptyFunction(a):
    pass
# Define Threshold parameters (Variables) and Trackbars for variation: area size filtering and 2 thresholds for Canny Edges
cv2.namedWindow("Variables")
cv2.resizeWindow("Variables",640,240)
cv2.createTrackbar("Thresh_1","Variables",172,255,emptyFunction) # Threshold parameter 1
cv2.createTrackbar("Thresh_2","Variables",30,255,emptyFunction) # Threshold parameter 2
cv2.createTrackbar("Area","Variables",7000,121500,emptyFunction) # Area filtering variable

# Function that takes input 'img' and returns output 'imgContour' (contours and labels drawn over original image)
def getContours(img,imgContour):
  # Extract EXTERNAL contours from the input image, img=Dilated_img, when the function will be called
  contours, unused_var = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)# simple-less no of points
  # Draw external contours over the output image, imgContour will be a copy of img initially
  cv2.drawContours(imgContour,contours,-1,(255,0,255),3)
  # Iterate over all contours (cnt)
  for cnt in contours:
    # Calculate area of each contour (in pixels)
    area = cv2.contourArea(cnt)
    # Set an arbitraty minimum area, from Area trackbar in 'Variables' tab
```

```
38      areamin=cv2.getTrackbarPos("Area","Variables")
39      # Any contour with less than areamin will be ignored (reduces unwanted
            noise)
40      if area > areamin:
41        # If area is large enough, draw the contours over the output image,
              imgContour
42        cv2.drawContours(imgContour,cnt,-1,(255,0,255),3)
43         # Compute the contour perimeter / length (True=only closed contours
               considered)
44        perimeter=cv2.arcLength(cnt,True)
45        # Approximate the shape of object, contours (cnt) as input, resolution
              =0.01*length, True=only closed contours
46        shape_approximation = cv2.approxPolyDP(cnt, 0.01*perimeter, True)#
              contains a number of points, used in determining shapes: length(
              shape_approximation)
47        # Calculates minimum upright bounding rectangle for the specific point
              set (object)- outputs start coordinates (x,y) (top left) and
              rectangle dimensions (w,h)
48        x,y,w,h=cv2.boundingRect(shape_approximation)
49        # Draw bounding box over object: input the start coords (x,y) and end
              coords (x+w,y+h)
50        cv2.rectangle(imgContour,(x,y),(x+w,y+h),(0,255,0),3)
51        # Display number of points of an object, area in pixels and CENTER
              POINT
52        cv2.putText(imgContour, "No. of Points: " + str(len(shape_approximation
              )), (x, y + 20), cv2.FONT_HERSHEY_COMPLEX, .5,
53                            (0, 0, 255), 1)
54        cv2.putText(imgContour, "Area [p]: " + str(int(area)), (x, y + 40), cv2
              .FONT_HERSHEY_COMPLEX, 0.5,
55                            (0, 0, 255), 1)
56        cv2.putText(imgContour, "Center [p]: ("+ str(x+int(w/2)) +","+ str(y+
              int(h/2))+ ")", (x, y + 80), cv2.FONT_HERSHEY_COMPLEX, 0.5, (0, 0,
              255), 1)
57        # If an object has 4 or 5 points, it is a window (rectangular window
              assumed)
58        if len(shape_approximation)>=4 and len(shape_approximation)<=5:
59          cv2.putText(imgContour, "Window", (x, y + 60), cv2.
                FONT_HERSHEY_COMPLEX, .5,
60                            (0, 0, 255), 1)
61          # Draw center point of window/rectangle in red
62          cv2.circle(imgContour, (x+int(w/2),y+int(h/2)), radius=0, color=(0,
                0, 255), thickness=5)
63
64          # AREA IN METERS CALCULATIONS, using the calibration.py script
65          DC1=0.12 # input variable distance measured by sensor [m] from drone
                to wall
66          PPM1=calibration.PPM*calibration.DC/DC1 # PPM' at that exact distance
                (DC1)
67          print("PPM at actual distance DC1: "+ str(PPM1))
68          A_p1=area # Area in pixels of actual object
69          print("Actual object area in pixels: "+ str(A_p1))
70          A_m1 = A_p1 / PPM1 # REQUIRED AREA of object [m^2]
71          print("Actual object area in meters: "+ str(A_m1)) # REQUIRED AREA OF
                ENTRY POINT DISPLAYED IN TERMINAL
72          # Display area [m^2] on image
```

```
73          cv2.putText(imgContour, "Area [m^2]: " + str(A_m1), (x, y + 100), cv2
                .FONT_HERSHEY_COMPLEX, 0.5,
74                      (0, 255, 0), 1)
75      # If an object has >5 points, it is a hole (circular,elliptic,irregular
            shaped holes assumed)
76      else:
77          cv2.putText(imgContour, "Hole", (x, y + 60), cv2.FONT_HERSHEY_COMPLEX
                , .5,
78                      (0, 0, 255), 1)
79          # Draw center point of hole in red
80          cv2.circle(imgContour, (x+int(w/2),y+int(h/2)), radius=0, color=(0,
                0, 255), thickness=5)
81
82 # Run for every frame in the videostream; img = each frame
83 while True:
84     grabbed, img = capture.read() # grab the frames and store them as img
85     imgContour = img.copy() # initialize the output imgContour as a copy of
           img
86     # Apply Gaussian Blur Filter, (9,9) kernel to smooth the image and reduce
           noise
87     Blur_img = cv2.GaussianBlur(img, (9, 9), 1)
88     # Convert to Greyscale
89     Gray_img = cv2.cvtColor(Blur_img, cv2.COLOR_BGR2GRAY)
90     # Set treshold values for Canny Edge Detector, using trackbars in the '
           Variables' window
91     thresh_1 = cv2.getTrackbarPos("Thresh_1", "Variables")
92     thresh_2 = cv2.getTrackbarPos("Thresh_2", "Variables")
93     # Apply Canny Edge Detection filter with variable thresholds, tested
           defaults are 172,140, can be varied
94     Canny_img = cv2.Canny(Gray_img,thresh_1,thresh_2)
95     # Creating convolution kernel used for Dilation
96     kernel = np.ones((5, 5))
97     # Dilating the Canny Edges in order to accentuate the shape, iterations=1
           for most accurate object area
98     Dilated_img = cv2.dilate(Canny_img, kernel, iterations=1)
99     # Use defined function to draw the boxel, labels and areas using the
           Dilated_img as input, for accurate detection, and the copy of img (
           imgContour) as the output image. Thus, the boxes, labels and areas
           will appear on the original un-filtered videostream live.
100    getContours(Dilated_img, imgContour)
101    # Display original videostream frames with contours, boxes, labels and
           areas
102    cv2.imshow("Contours", imgContour)
103    # Stop script upon pressing 'q'
104    if cv2.waitKey(1) & 0xFF == ord('q'):
105        break
```

B.2 Calibration script (pre-flight)

```
1 # Created by Vlad Marascu for GDP ADD02 Group 2020
2 # 04/06/2020
3
4 # Calibration script- use same camera/resolution as drone and take a picture
    of a rectangular object (known area A[m]) from a known distance DC[m] to
    use in the main 'Shapes from webcam.py' script.
```

```
5
6  import cv2
7  import numpy as np
8
9  testimg=cv2.imread('images/calibration-type1,1m.png') # import picture used
       for calibration
10 # Calibration object has known area in meters, A_m, and DC (camera-object
       distance)
11 # 'emptyFunction' function defined to be used by the 'Variables' Trackbar
12 def emptyFunction(a):
13   pass
14
15 # Define Threshold parameters (variables) and Trackbars for variation: area
       size filtering and 2 thresholds for Canny Edges
16 cv2.namedWindow("Variables")
17 cv2.resizeWindow("Variables",640,240)
18 cv2.createTrackbar("Thresh_1","Variables",172,255,emptyFunction) # Threshold
       parameter 1
19 cv2.createTrackbar("Thresh_2","Variables",30,255,emptyFunction) # Threshold
       parameter 2
20 cv2.createTrackbar("Area","Variables",7000,121500,emptyFunction) # Area
       filtering variable
21
22 # Apply Gaussian Blur Filter, (9,9) kernel to smooth the image and reduce
       noise
23 testimgBlur=cv2.GaussianBlur(testimg,(9,9),1)
24 # Convert to Greyscale
25 testimgGray=cv2.cvtColor(testimgBlur,cv2.COLOR_BGR2GRAY)
26 # Set treshold values for Canny Edge Detector, using trackbars in the '
       Variables' window
27 thresh_1=cv2.getTrackbarPos("Thresh_1","Variables")
28 thresh_2=cv2.getTrackbarPos("Thresh_2","Variables")
29 # Apply Canny Edge Detection filter with variable thresholds, tested defaults
        are 172,140, can be varied
30 testimgCanny=cv2.Canny(testimgGray,thresh_1,thresh_2)
31 # Creating convolution kernel used for Dilation
32 kernel = np.ones((5, 5))
33 # Dilating the Canny Edges in order to accentuate the shape, iterations=1 for
        most accurate object area
34 testimgDil = cv2.dilate(testimgCanny, kernel, iterations=1)
35
36 # Copy test image to testimgContour
37 testimgContour=testimg.copy()
38 # Extract contours from the Dilated image, testimgDil
39 contours, unused_var = cv2.findContours(testimgDil, cv2.RETR_EXTERNAL, cv2.
       CHAIN_APPROX_SIMPLE)# simple-less no of points
40 # Draw extracted contours over the copied original image
41 cv2.drawContours(testimgContour,contours,-1,(255,0,255),3)
42
43 # Iterate over all contours(cnt)
44 for cnt in contours:
45   # Calculate area of each contour in pixels
46   area = cv2.contourArea(cnt)
47   # Set an arbitraty minimum area, from Area trackbar in 'Variables' tab
48   areamin=cv2.getTrackbarPos("Area","Variables")
```

```
  # Any contour with less than areamin will be ignored (reduces unwanted
      noise)
   if area > areamin:# and area<30000:
     # Compute the contour perimeter / length (True=only closed contours
        considered)
     perimeter=cv2.arcLength(cnt,True)
     # Approximate the type of shape of object, contour as input, resolution
        =0.01*length, True=only closed
     shape_approximation=cv2.approxPolyDP(cnt,0.01*perimeter,True) # contains
        a number of points, used in determining shapes: length(shape_approx),
        not needed for calibration script; rectangular object assumed
     # Calculates minimum upright bounding rectangle for the specific point
        set (object)- outputs start coordinates (top left) and x-y rectangle
        dimensions (w,h)
     x,y,w,h=cv2.boundingRect(shape_approximation)
     # Draw bounding box over object: input the start coords (x,y) and end
        coords (x+w,y+h)
     cv2.rectangle(testimgContour,(x,y),(x+w,y+h),(0,255,0),3)
     # Display area value in pixels
     cv2.putText(testimgContour, "Area: " + str(int(area)), (x, y + 40), cv2.
        FONT_HERSHEY_COMPLEX, 0.5,
                         (0, 255, 0), 1)
     # Display original image with contours, bounding box and area in pixels
     cv2.imshow("Contours",testimgContour)

#DISTANCE CALCULATIONS

A_p = area # Area in pixels, test object
print("Test object area in pixels: "+ str(A_p))

# Known calibration object parameters [m]
A_m = 0.007056 # known test object area in [m^2]
DC = 0.15 # known set distance [m]

# Pixels per Meters ratio at known calibration distance
PPM = A_p / A_m # PPM at set initial distance DC
print("PPM at set DC: "+ str(PPM))

cv2.waitKey(0)
#cv2.destroyAllWindows()
```

C Results

C.1 Gazebo human recognition results

Table 2: Gazebo human recognition confidence results: average confidence is 64.70%

Capture No.	1	2	3	4	5	6	7	8	9	10
Confidence [%]	47.31	49.63	59.82	56.96	59.31	68.82	58.70	79.39	82.93	84.14
Percent error [%]	26.87	23.29	7.54	11.96	8.33	6.36	9.27	22.70	28.17	30.04

C.2 Real human recognition results

Figure 15: Real human recognition results with confidence; *Proceed=1* if detection made. Note that various postures (sitting down) are detected, as well as different profiles (side,back view).

C.3 Entry points detection: area measurement accuracy

The *Type 1* window is considered as the calibration object, with known area ($A_m = 0.22m^2$) and drone-wall distance ($DC = 1m$), as highlighted in Table 3 below. The rest of the areas, for each type/dimension of window and D'C distance, are computed using the main script and compared to the actual areas provided by the Simulations subgroup. Table 3 summarizes the obtained results and includes the percentage error of each measurement. Note that all window types were easily detected (2-hinged, sliding and awning) by the script; however, due to some pictures being taken from a slight angle, parallax error was present through the calculations, altering the results. Moreover, differences

in the resolution of the pictures were also present, due to the fact that they were screen captures, as opposed to being taken with the drone's camera. The most important source of errors is the fact that, due to hardware limitations, the drone could not be fully integrated and, therefore, the drone-wall distances (D'C) were mere approximations made by the Simulations subgroup. However, in the case of a fully integrated drone, with a working frontal sonic sensor that outputs the exact drone-window distances (D'C), it is expected that the percentage errors drop significantly and the script is able to accurately predict the window areas in meters.

Table 3: Testing results obtained on a set of Gazebo window types; *Type 1* is the calibration window

Image No.	Type	Actual Area [m]	Distance [m] (D'C)	Calculated Area [m]	Error (%)
1 (Calibration)	Type1	0.22 (A_m)	1 (DC)	\	\
2	Type2	0.37	2	0.40	5.82
3	Type3	0.72	4	0.61	15.27
4	Type4	1.06	2	0.83	21.84
5	Type4	1.06	5	0.94	11.48
6	Type2	0.37	3	0.43	13.75
7	Type3	0.72	6	0.75	4.16
8	Type4	1.06	6	1.25	17.70

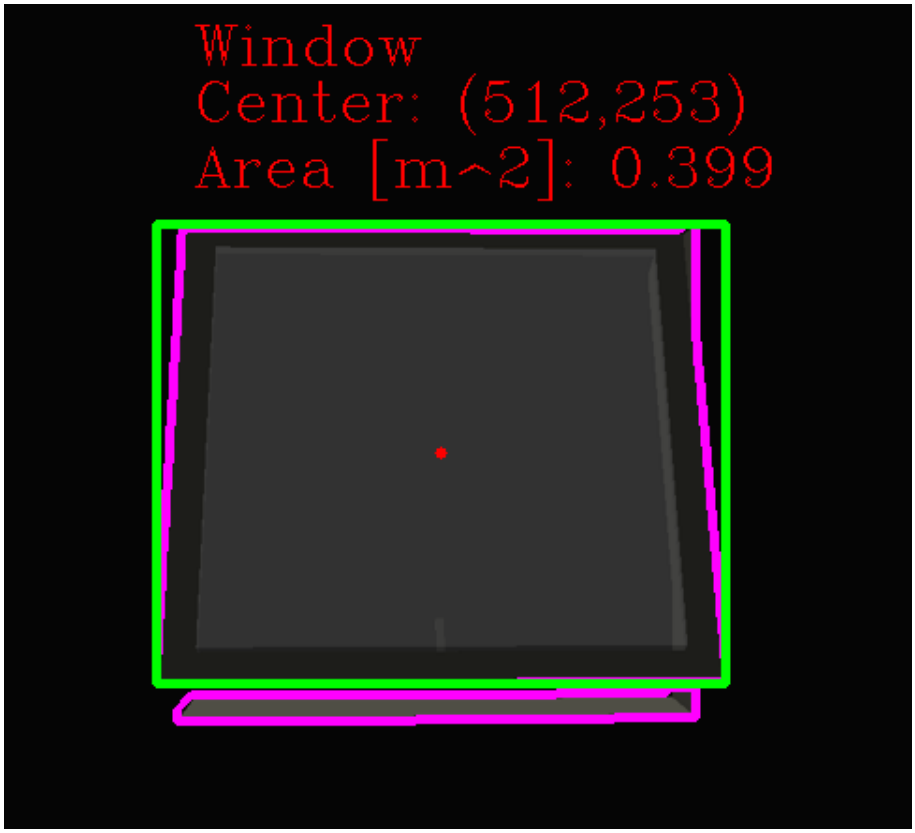

Figure 16: Detection example: *Type 2* window, $D'C = 2m$