Motion Tracking with Raspberry Pi

Saarang Srinivasan
National Public School Koramangala, Bengaluru
Intern at Mistral Solutions Pvt Ltd, Bengaluru
saarangsrini35@gmail.com

Abstract

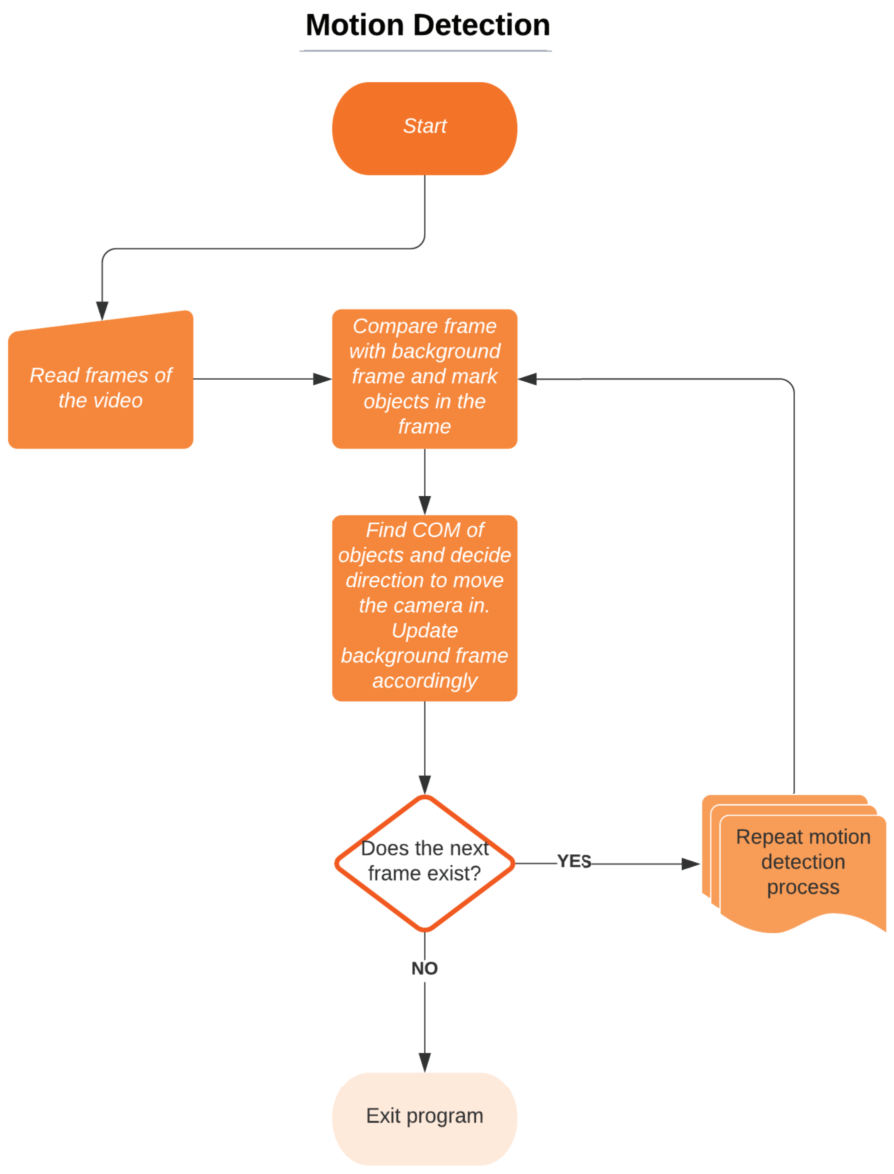
The aim of this project is to detect the motion in a video and accordingly follow the motion. This program uses python and contour detection to find the moving objects in the video and determine which direction we must move in order to follow the motion. We move the camera in the direction of the motion in order to follow it.

Table of Contents

1.1 Introduction

Objects move all the time. Tracking and following them is a task by itself. The project targets the tracking of an object's motion and following it.

1.2 Block Diagram

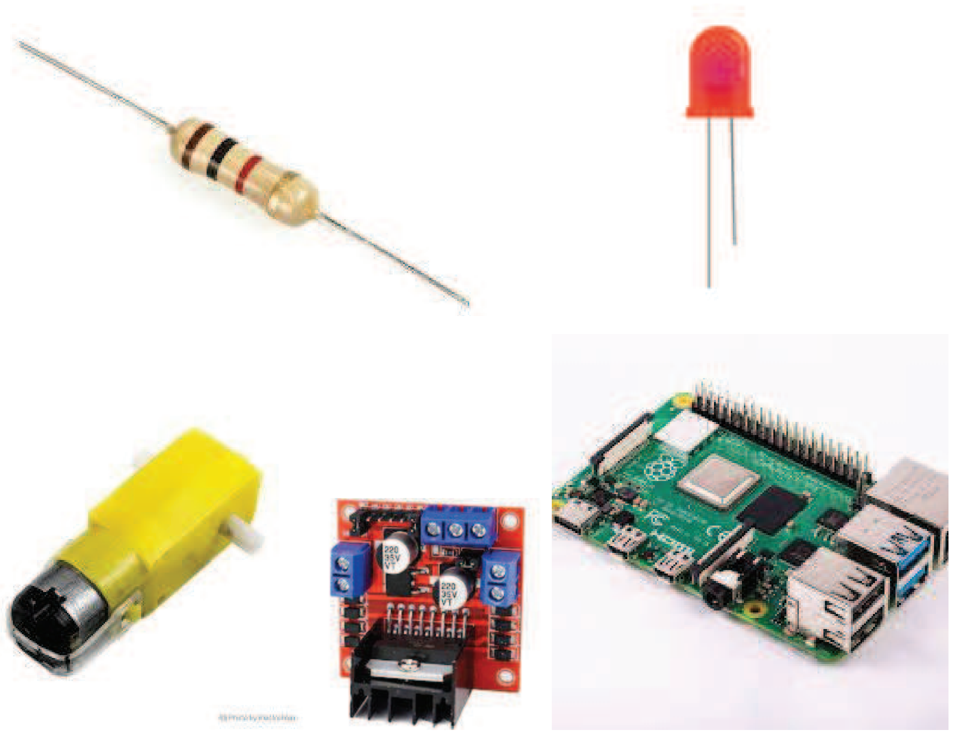

Motion Detection
Start
Read frames of the video
Compare frame with background frame and mark objects in the frame
Find COM of objects and decide direction to move the camera in. Update background frame accordingly
Does the next frame exist?
YES
Repeat motion detection process
NO
Exit program

1.3 Features and Applications

Motion tracking has a variety of applications in the military and in security. Security cameras, drones, military weapons usually have object tracking features to aid in object identification and counting as well as detection of dangerous substances and/or people.

2. Component's Build Equipment

1. Raspberry Pi - The Raspberry Pi is a low cost, credit-card sized computer which can be used to run python scripts and other programs.
2. Camera - A camera is used to take the video input for the motion tracking program
3. Servo/Motor - A servo/motor was used for debugging and to point out the direction in which the camera should move to follow the motion.
4. Motor Controller - If a motor is used, a motor controller is needed to connect the motor to the raspberry pi and a battery.
5. Python - On the software side, python was used to write the program.


Pictures of the build equipment (servo not shown)

3.1 Algorithm

3.1.1 Basic Algorithm

The most basic algorithm to detect motion in an image is to compare the frame differences between the first frame and the current frame and extract the contours. Contours can be explained simply as a curve joining all the continuous points (along the boundary), having the same color or intensity. Finding the contours can, therefore, help you find the object boundaries in a frame. Contour detection works more accurately on binary images so the frames are binarized based on a certain threshold (pixels above the threshold are 1 and below are 0).

3.1.2 Flaw with Basic Algorithm

The flaw with this algorithm is that it makes the assumption that the camera is stationary and the background for all frames remains the same. In a real life scenario, say a webcam on a drone, the camera may be moving as well and the background changes. To fix this issue, we cannot continue to compare the first frame and the current frame. Would comparing the current and the previous frame to find contours work? This heuristic looks like it'll work but in a video there are thousands of frames and the differences between two adjacent frames is usually minute and therefore the motion will not be detected over the entire video.

3.1.3 Background Frame

The major idea used in the motion detection algorithm is to compare two frames with something which works as a background frame. In the initial algorithm explained, the first frame works as the background frame (this works for stationary backgrounds). Therefore to make a working algorithm for mobile cameras, we will need to construct a background frame as such to detect motion.

How do we create this background frame? There are two possible approaches to make this background frame. The more format, complicated approach which possibly works much better is to create the background frame using the previous frames. Our approach is to "move" the background frame to the current frame on the specified amount. We shift this background frame slightly in the direction of the current frame. In the second approach of comparing adjacent frames, we assume the previous frame works as the background frame. In the current heuristic, we shift the background frame only slightly towards the current frame i.e achieving a more accurate background frame. This approach is quite difficult as we will have to morph the frames in specific percentages and find the numbers for which the motion is detected the most accurately. Morphological operations such as erosion and dilation were used on the frames to make the object boundaries more clear before the contour detection.

The other approach is a basic addition to the initial approaches. Instead of comparing the first and current frame, or adjacent two frames, we can compare the current frame and the $(current\ -\ X)^{th}$frame where X is an independent variable chosen. Why does this work more accurately than the other basic approaches? This is because we will choose X such that it is not too near to the current frame (in which case the difference between the frames will be too minor) nor is it too far from the current frame (in which case the difference between the frames will be inaccurate for mobile cameras). Here the $(current\ -\ X)^{th}$ frame is assumed as the background frame and it works fairly accurately upon testing as well.

3.1.4 Direction of Motion

Using the above approaches, we can detect the differences in the background frame and the current frame. We then find the contours to mark out the images and find the object boundaries. The next step is to find which direction the object is moving in and how to follow it. Note that there may be multiple objects detected in the video, some due to inaccuracies in the contour detection process.

To find this "side", there are multiple approaches. The trivial idea is to just brute force over all rectangles the camera frame can now view after moving it in that direction. For each rectangle, we can count the number of objects and/or number of pixels with objects in it and accordingly choose the rectangle i.e camera movement which is the most appropriate. Since this approach is brute force, it works accurately but it has other flaws. For a camera frame of dimensions $n\ *\ m$, the total number of rectangles possible is of the order $O((nm)^2)$. As the size of the rectangle is fixed (camera frame size), this number reduces to $O(nm)$. We need to check the number of objects/pixels in this rectangle, therefore the total complexity will be $O((nm)^2)$ which can further be optimized to $O(nm)$ by precomputing $2D$ prefix sums. Since the number of frames in a video is usually in thousands and camera frame dimensions are usually quite large too, this algorithm, though accurate, is usually too inefficient to be used for real life.

The next two approaches use the concept of center of mass. Assume each object is in the form of a rectangle, therefore its center is the center of the rectangle. Say we have N objects each with the weight W_i and their respective centers (X_i, Y_i). Therefore the coordinates of the center of mass of this distribution will be given by the formulae:

$$COM_x = \frac{\sum_{i=1}^{i=n} W_i * X_i}{\sum_{i=1}^{i=n} W_i}$$

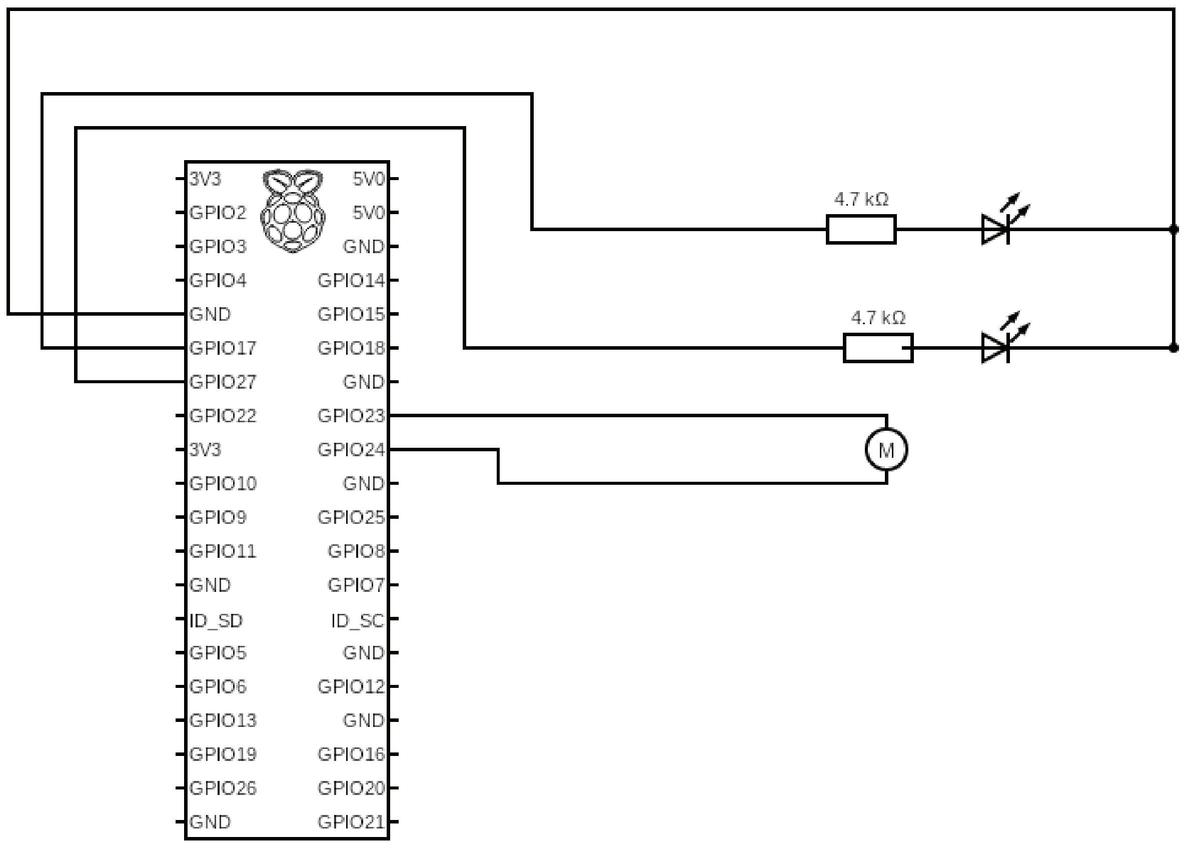
$$COM_y = \frac{\sum_{i=1}^{i=n} W_i * Y_i}{\sum_{i=1}^{i=n} W_i}$$

The center of mass of this object distribution would give the "side" where the majority of the objects are. If the center of mass is on the left of the center of the camera frame, we need to move the camera towards the left. Using the x-coordinate and y-coordinate of the center of mass, we can similarly determine the directions in which the camera should be moved in order to follow the majority of the objects and their motion. Note that finding the center of mass would be more efficient than the $O(nm)$ algorithm as we are only traversing through the detected objects whose area is greater than a certain threshold (set by default to 500). These objects are likely to be disjoint components of the image further limiting the total number of objects possible.

In the first approach, we consider each object has a weight of 1 and find the center of mass using this assumption. Though this heuristic would work, its accuracy is questionable as it will be easily influenced by outliers (objects which are small and negligible/irrelevant) as each object is given equal weightage and priority in this algorithm. The second approach builds on this by giving higher preference to an object of larger size than to an object of smaller size. For each object of size$r * c$ pixels, we give it a weight of $r * c$. Putting these weights into the center of mass formula would give a more accurate position which would further improve the overall motion following accuracy.

Therefore, we can both detect and follow object motion using a combination of background detection, contour finding and using the center of mass concept. The actual following of the motion has been implemented using a Raspberry Pi. A motor and an LED are connected and the motor moves in the direction of motion. The program is also tested on a camera which can rotate in the direction of motion.

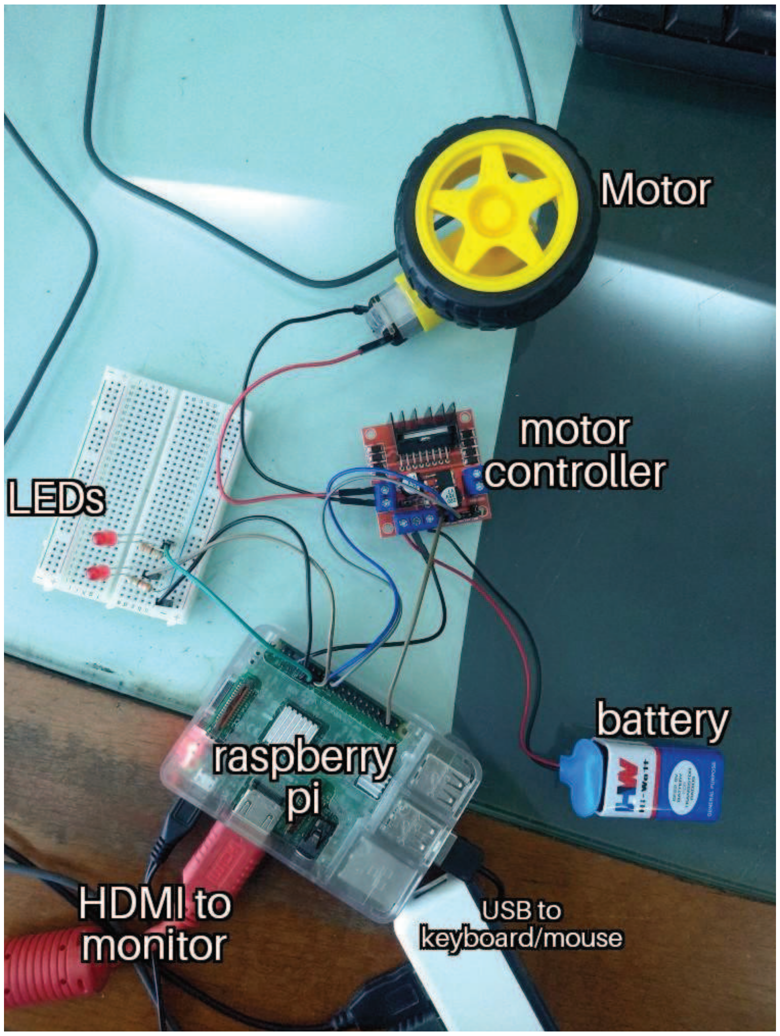
3.2 Circuit Diagram

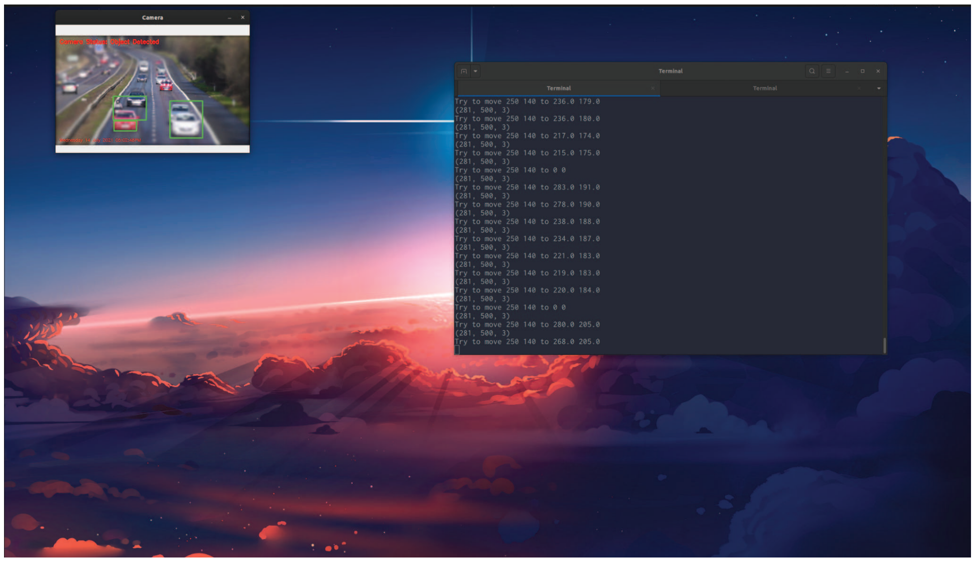


3.3 Device Construction

Connect 2 LEDs and resistors to the GPIO pins as mentioned in the circuit diagram. Use a motor/servo and connect it to the Raspberry Pi. A battery may be required as well depending on the requirements of the motor/servo.

Ensure you have Python 3 installed on your Raspberry Pi along with necessary modules such as `RPi`, `open-cv`, `imutils`, `argparse`, `logging`, `time` and `datetime`. You may need a camera to run the program as well.




Camera
Camera Status: Object Detected
Wednesday 14 July 2021 05:02:48PM
Terminal
Terminal
Terminal

```
Try to move 250 140 to 236.0 179.0
(281, 500, 3)
Try to move 250 140 to 236.0 180.0
(281, 500, 3)
Try to move 250 140 to 217.0 174.0
(281, 500, 3)
Try to move 250 140 to 215.0 175.0
(281, 500, 3)
Try to move 250 140 to 0 0
(281, 500, 3)
Try to move 250 140 to 283.0 191.0
(281, 500, 3)
Try to move 250 140 to 278.0 190.0
(281, 500, 3)
Try to move 250 140 to 238.0 188.0
(281, 500, 3)
Try to move 250 140 to 234.0 187.0
(281, 500, 3)
Try to move 250 140 to 221.0 183.0
(281, 500, 3)
Try to move 250 140 to 219.0 183.0
(281, 500, 3)
Try to move 250 140 to 220.0 184.0
(281, 500, 3)
Try to move 250 140 to 0 0
(281, 500, 3)
Try to move 250 140 to 280.0 205.0
(281, 500, 3)
Try to move 250 140 to 268.0 205.0
```

Example of the construction and the monitor output

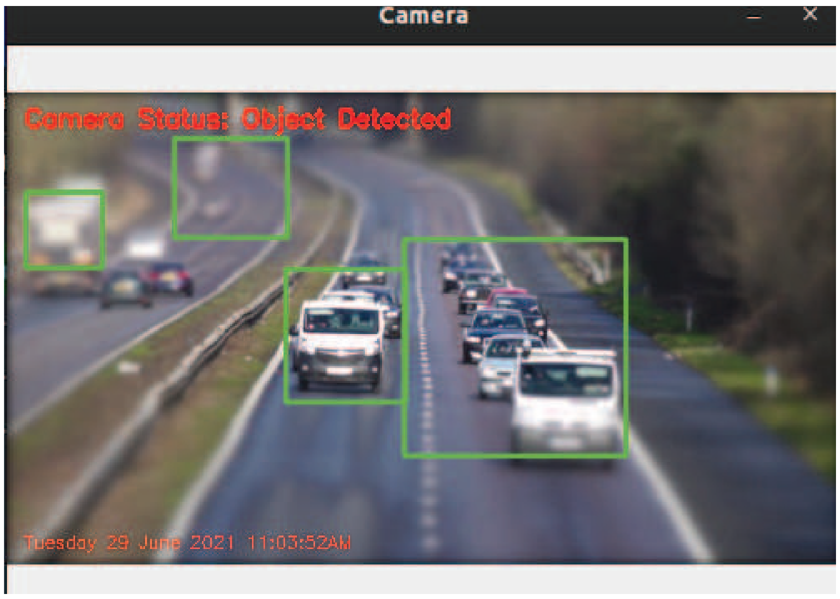
4.1 Advantages and Limitations

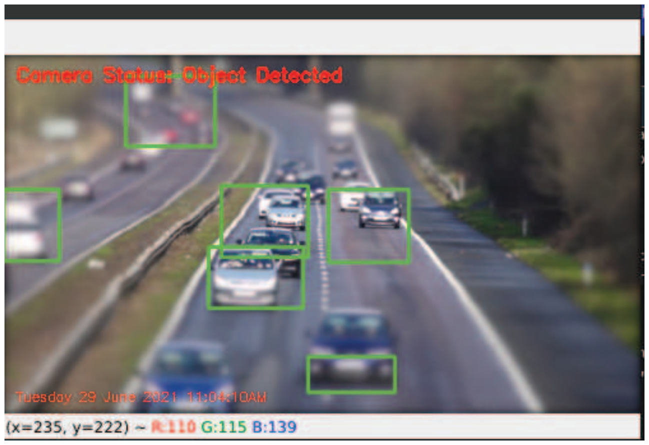
The major part of the algorithm is based on the tracking of contours or the object boundaries. This method works well on both rigid and non-rigid bodies and a change in illumination of the frame does not affect accuracy much. However, this algorithm is unable to handle entry/exit of objects as well as a continuously shifting background well and it leads to a drop in accuracy. This is due to the movement in the background being treated as a moving object and the program tends to misidentify objects and backgrounds in these situations. Using the morphing to make a continuous background frame, however, should be able to improve accuracy even for moving backgrounds.

4.2 Results and Analysis

4.2.1 Basic Algorithm

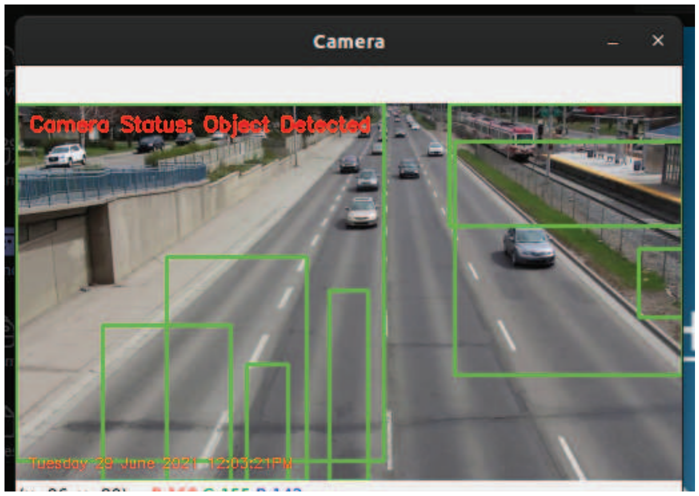
The basic algorithm for motion detection discussed above works amazingly accurately for stationary cameras with stationary backgrounds as the background frame in this case is the same as the actual background and we do not have to build it explicitly. Some example outputs using this basic algorithm on stationary backgrounds are below.

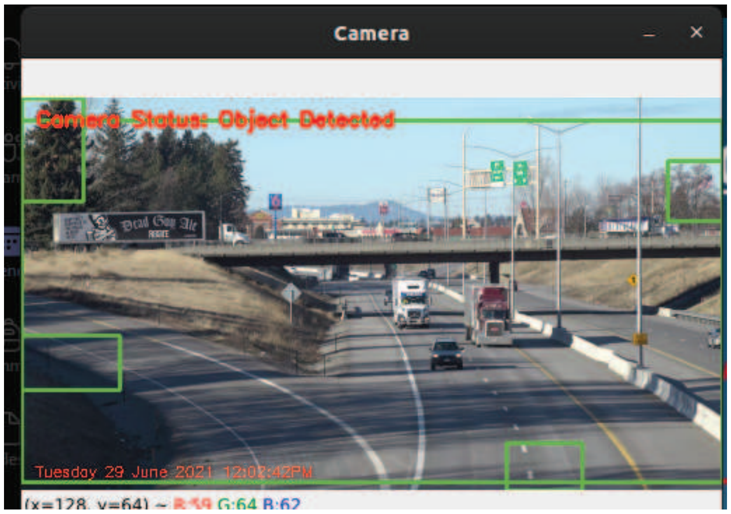

Camera
Camera Status: Object Detected
Tuesday 29 June 2021 11:03:52AM


Camera Status: Object Detected
Tuesday 29 June 2021 11:04:10AM
(x=235, y=222) ~ R:110 G:115 B:139

Basic algorithm works on stationary backgrounds

The basic algorithm however doesn't work accurately for mobile backgrounds at all. Examples of the algorithm on these backgrounds are below.


Camera
Camera Status: Object Detected
Tuesday 29 June 2021 12:03:21PM


Camera
Camera Status: Object Detected
Tuesday 29 June 2021 12:02:42PM
(x=128, y=64) ~ R:59 G:64 B:62

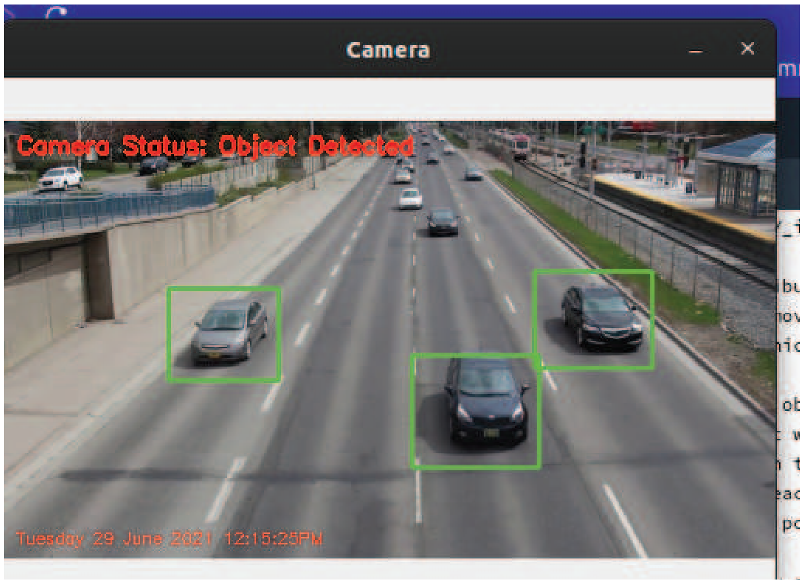
Basic algorithm fails on moving backgrounds

The difference in accuracy for stationary and mobile backgrounds is very noticeable. The drop in accuracy is because the movement in the background is counted as an object.

4.2.2 Adjacent Frames

This algorithm is the same as the basic one but instead of comparing the first and the current frame, we compare two adjacent frames. It works well for videos where objects are moving at high speeds so there are noticeable differences even between two adjacent frames.

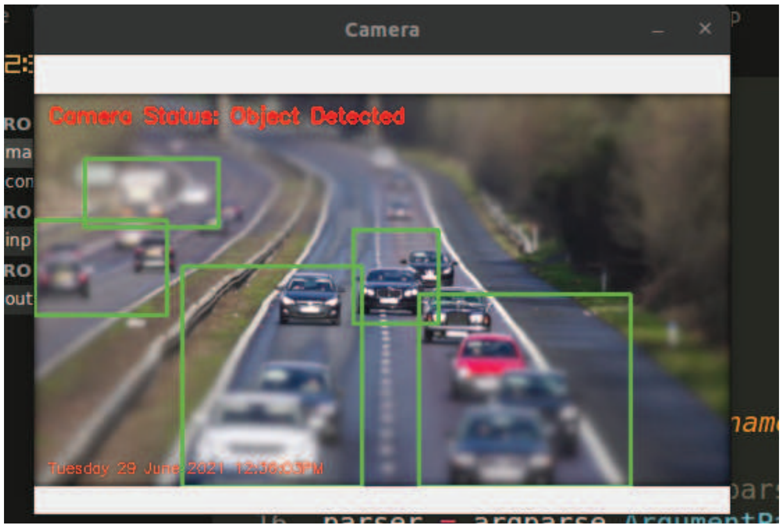
An example output of this on a fast moving video is below.

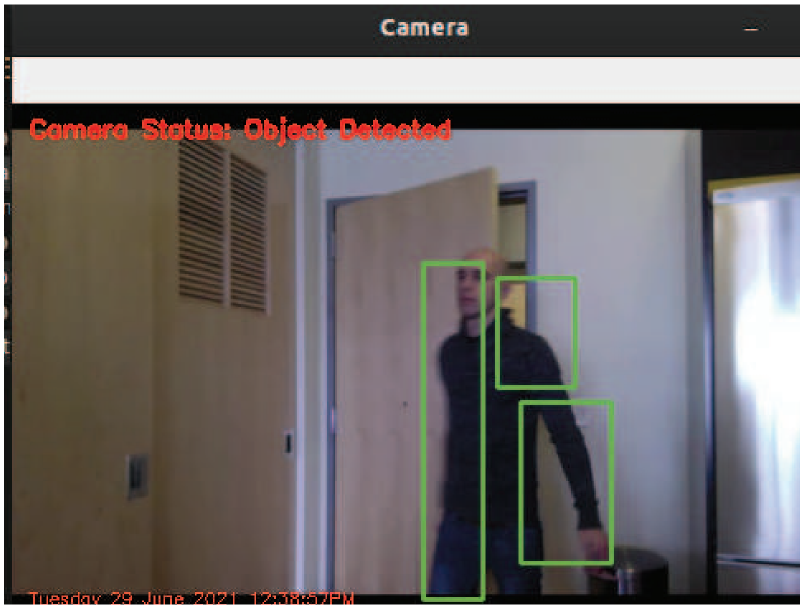

Camera
Camera Status: Object Detected
Tuesday 29 June 2021 12:15:25PM

Adjacent frames method on fast moving video

4.2.3 Background Frame Algorithm

The algorithm based on using the background frame to detect the motion leads to an overall increase in accuracy though its accuracy for stationary backgrounds is not as accurate as the basic algorithm. As mentioned before, the background frame and the current frame have to be merged. After researching a bit along with some testing, merging the frames with around $60 - 80\%$ of the original image and $20 - 40\%$ of the overlay image is usually appropriate. The other background construction method works most accurately when X is between $4 - 40$ depending on the camera specifications.


Camera
Camera Status: Object Detected
Tuesday 29 June 2021 12:38:35PM

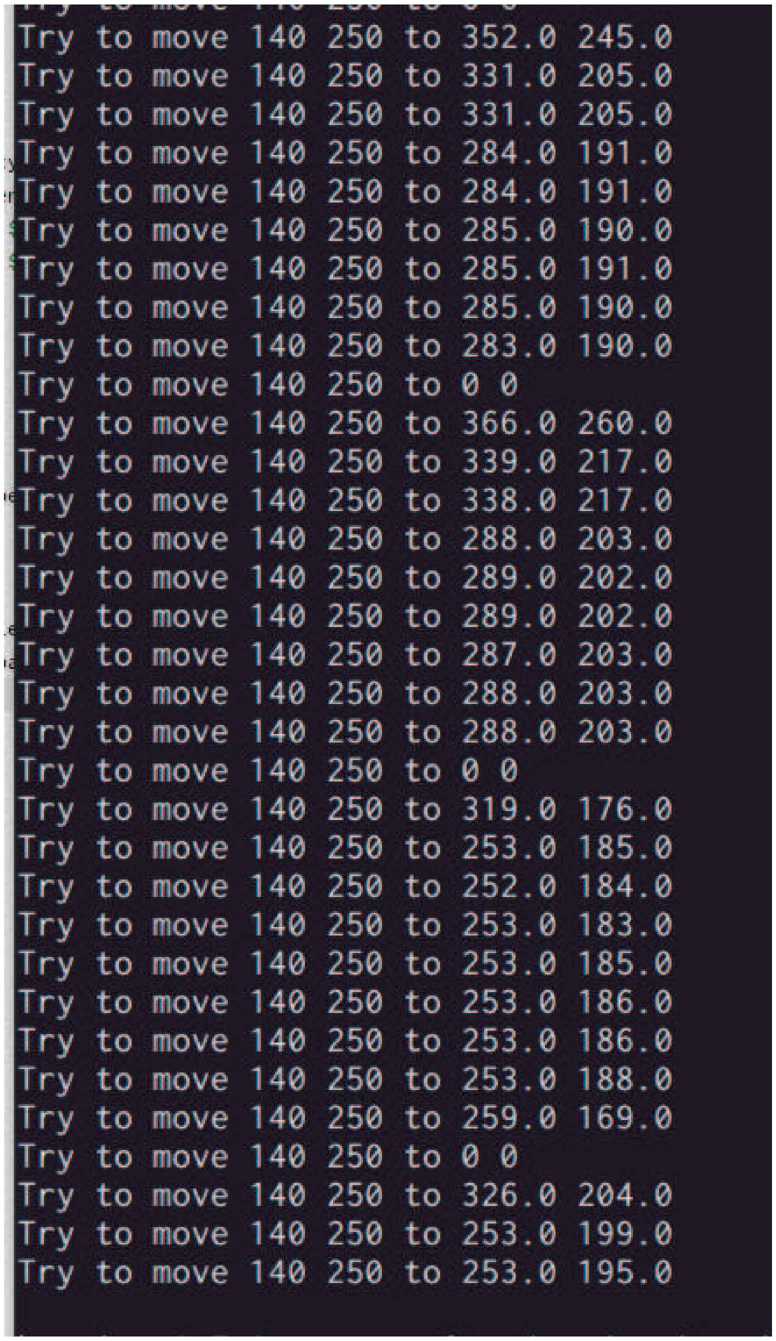

Camera
Camera Status: Object Detected
Tuesday 29 June 2021 12:38:57PM

Background construction method on example videos

The accuracy using the simple background frame algorithm is not extremely high overall but it does not have specific cases where it fails as compared to the initial algorithms.

4.2.4 Following the Motion

The accuracy of the center of mass coordinates obviously depends on the accuracy of the object detection. Implementation wise too, finding the center of mass is quite direct. To ensure we only consider actual objects, there is also a min-area argument passed and objects found are only considered if their area (number of pixels) is greater than a certain threshold. An example of the center of masses found for each frame and the camera frame center if given below.



```
Try to move 140 250 to 352.0 245.0
Try to move 140 250 to 331.0 205.0
Try to move 140 250 to 331.0 205.0
Try to move 140 250 to 284.0 191.0
Try to move 140 250 to 284.0 191.0
Try to move 140 250 to 285.0 190.0
Try to move 140 250 to 285.0 191.0
Try to move 140 250 to 285.0 190.0
Try to move 140 250 to 283.0 190.0
Try to move 140 250 to 0 0
Try to move 140 250 to 366.0 260.0
Try to move 140 250 to 339.0 217.0
Try to move 140 250 to 338.0 217.0
Try to move 140 250 to 288.0 203.0
Try to move 140 250 to 289.0 202.0
Try to move 140 250 to 289.0 202.0
Try to move 140 250 to 287.0 203.0
Try to move 140 250 to 288.0 203.0
Try to move 140 250 to 288.0 203.0
Try to move 140 250 to 0 0
Try to move 140 250 to 319.0 176.0
Try to move 140 250 to 253.0 185.0
Try to move 140 250 to 252.0 184.0
Try to move 140 250 to 253.0 183.0
Try to move 140 250 to 253.0 185.0
Try to move 140 250 to 253.0 186.0
Try to move 140 250 to 253.0 186.0
Try to move 140 250 to 253.0 188.0
Try to move 140 250 to 259.0 169.0
Try to move 140 250 to 0 0
Try to move 140 250 to 326.0 204.0
Try to move 140 250 to 253.0 199.0
Try to move 140 250 to 253.0 195.0
```

Center of Mass debug log from a video

The last 2 numbers are the center of mass x and y coordinates found and the first 2 numbers are the center of the camera. This is a screenshot of the debug log produced after running the program on a video. The x and y coordinates are $(0, 0)$ when there is no object detected in that frame. This case has to be handled separately to avoid division by 0 when finding the center of mass.

4.3 Conclusion

The algorithm worked fairly accurately on most environments. The constants, threshold, amount of dilation/erosion played an important role in the accuracy. The threshold values allowed us to neglect irrelevant objects moving such as moving leaves in a tree. The background frame computation process used is quite direct and simple here. The algorithm can be improved by using better background estimation and using cascade classifiers to break images into the required objects. Another optimization can be visualizing the frames as a graph where each connected component (pixels which are in the same colour) is potentially an object. Motion detection and tracking already has various uses in the current day scenario and will potentially have many more in the future.

5. Acknowledgements

I express my deepest thanks to Mr. Rohit Dey for taking part in the important decision making process and giving necessary advice and guidance and arranged all facilities to make life easier. I choose this moment to acknowledge his contribution gratefully. It is my radiant sentiment to place on record my best regards, deepest sense of gratitude to Mr. Rajeev Ramachandra, CTO and Founder of Mistral Solutions Pvt Ltd, for providing me this opportunity to work on this project under Mistral Solutions. I'd like to convey my regards to my computer science teacher Chandita Mukhopadhyay and my school principal Jyotsna Nair for supporting me throughout and allowing me to miss school to work on and complete this project. I perceive this opportunity as a big milestone in my career development. I will strive to use gained skills and knowledge in the best possible way, and I will continue to work on their improvement, in order to attain desired career objectives.

6. References

I. A robust and computationally efficient motion detection algorithm based on $\Sigma - \Delta$ background estimation by A. Manzanera and J. C. Richefeu https://perso.ensta-paris.fr/~manzaner/Publis/icvgip04.pdf

II. Wikipedia - https://en.wikipedia.org/wiki/Motion_detection

III. GPIO documentation and pinout.xyz for hardware with Raspberry Pi