**\subsection{Configuration 4, Pi 4}**

**V2.**

In this section, we will present Configuration 4, which stands out as the sole configuration that does not incorporate hardware acceleration, unlike the other configurations discussed in this report.

**\subsubsection{Blob detection software}**

For image capture, we use the Pi Camera V3, interfaced via the Picamera2 library. And for image processing, we implement OpenCV, a widely-used library known for its powerful computer vision and image processing capabilities. It is particularly suitable for blob detection because of its versatility, efficiency, and ease of use.

**\subsubsection{Blob detection models and techniques}**

We opted for blob detection or blob analysis, an image processing technique that identifies and analyzes distinct areas of interest within an image. In the context of our project, blob detection was tasked with identifying balls of different colors. It does this by focusing on their HSV color values and contour characteristics.

Blob detection was chosen for its adaptability and precision. It can be finely tuned to detect specific objects, has a high degree of flexibility to meet different computational needs, and is particularly suited to the Raspberry Pi 4's limited processing power. These attributes made blob detection a compelling choice, as our project's aim was to balance detection accuracy with computational efficiency.\ref{appendix:config4}

**\subsubsection{Software overview}**

Camera interface: Picamera2 Python library\\

Object detector type: Blob detection algorithm\\

Image processing library: OpenCV and cvzone\\

Model input: Images from Pi Camera V3\\

Model output: Coordinates of HSV color values and contour properties (circle)\\

OpenCV and our blob detection algorithm process these frames to identify distinct blobs (regions of interest) within the image. The algorithm outputs the coordinates of detected blobs, along with their HSV color values and contour properties.

**\subsubsection{Config4 journey}**

To provide a clear and comprehensive understanding of our journey through this configuration, we developed a model accompanied by an explanatory guide. The model is divided into three phases, which illustrate the different stages of our work, the problems we encountered, and the solutions we implemented. This approach offers a clear view of how our project developed from start to finish.

In the first phase of our task, we decided to use Python with OpenCV for image processing. Python is a flexible and user-friendly language, making it an ideal choice for quick development. OpenCV is a powerful tool for image processing, offering a wide range of optimized algorithms. Using Python and OpenCV together enabled us to build and test our image-processing algorithms efficiently and effectively.

After deciding to use Python and OpenCV for image processing, we needed to choose a suitable algorithm for our task. We selected blob detection, considering its efficiency and simplicity. This algorithm was an easy choice due to the lack of computing resources in our configuration. It’s important to note that efficiency and effectiveness can still depend on the specifics of a task.

Moving to the second phase, we knew one of our client's wishes was for us to develop a program capable of detecting three tennis-sized balls, each of a different color. The program was also required to calculate and display their x, and y coordinates, the distance of each detected ball, and FPS on the frame. Initially, we worked in such a way that we developed and tested the program on a high-end laptop with a webcam. At this point, our blob detection algorithm could only detect the ball at approximately 1m, when our goal was to detect it up to 3-4m.

Having realized that the initial version (v1) of our program did not fulfill the client's requirement, it became evident that we had to improve our blob detection. As we went to the next step to expand the program's capabilities from detecting one ball to three balls, the computational demands increased, which lead to the challenge of balancing efficiency and performance within our resource constraints.

With the improvements in place, our v2 version of the program had advanced to the point where it could successfully detect three differently colored balls, simultaneously displaying their depth along with their x and y coordinates within the frame. However, despite better efficiency and accuracy, we were yet to reach our objective of detecting the balls from a distance of 3-4 meters. Furthermore, the frame rate was still a matter of concern, as we were only able to achieve an average of 4-8 frames per second, varying based on the resolution scale.

Upon testing the program on a Raspberry Pi, we found that the computational demands were too high for the hardware to handle effectively since we built and tested the program on a high-end laptop using the webcam. As a result, we needed to revisit our approach and develop a more simplified version that could effectively run on the Raspberry Pi.

In the third phase of our progress, in response to the computational constraints, we developed a simpler version of our blob detection algorithm for the v3 iteration. By simplifying and removing some of the more computationally intense functionalities from the program, we were able to finally achieve the detection of the balls from a distance of 3-4 meters on our PC. While the precision of this version was somewhat compromised in comparison to v2, it was a necessary trade-off given our resource constraints. This version also offered a significant improvement in frame rate performance as we were now achieving between 25-30 frames per second, also resolving the previous frame rate issues we had encountered.

Upon testing our code on the Raspberry Pi, we encountered consistent results, primarily due to the frame rate on the desktop PC being limited to 30 fps. Due to the change of cameras being used from webcam to pi camera 3, the HSV values needed to be updated. The code can be modified to be more accurate with further iterations since we see that we can press the raspberry pi even further.

To further test our configuration, we set out to evaluate the effectiveness of our blob detection. For this purpose, a script was developed to process labeled validation images, calculating precision, recall, and eventually yield the f1 score. This precision metric was chosen to compare our algorithm's performance with a trained model. However, given the algorithm's dependency on HSV values, its performance varied significantly under different lighting conditions. This was a result of predetermined HSV values becoming inconsistent under varying lighting conditions, leading to inaccurate detections and incomplete detection of objects. To test this, we used two distinct image sets; one was captured in the same room under identical lighting conditions, and the other involved varying lighting and distance. The results of these tests varied widely; the test under uniform lighting yielded highly accurate results, whereas the test under varied lighting conditions underperformed in comparison.

Our final examination focused on the detection range of our algorithm. To do this, we physically marked distances in half-meter increments up to a total of 3 meters. We then placed the ball, serving as the object, on each of these marks. The objective of this test was twofold: first, to ascertain the furthest distance at which we could reliably detect the object, and second, to evaluate the accuracy of our distance calculation algorithm. In this scenario, our blob detection algorithm demonstrated consistent results up to a distance of 2-2.5 meters. Beyond this range, however, the detection became unreliable. The ball was still detected occasionally, roughly once every 10 frames, but the inconsistency in detection did not meet the real-time requirements for a flying drone.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Blob** **detection.v1** | **Blob** **detection.v2** | **Blob** **detection.v3** |
| **Libraries** |  | Import additional libraries such as imutils, math cv2, cvzone.FPS, and numpy.  The first code imports more libraries, which increases memory usage and potentially affects the overall performance. | cv2, cvzone, cvzone.ColorModule, cvzone.FPS, and numpy.  In contrast, the second code uses the cvzone library, which is built on top of OpenCV and provides a simpler and more optimized interface for many common tasks in computer vision. |
| **Functionality** |  | The first code includes a more extensive set of features, including distance calculation, color detection, object detection, object tracking, and graphical overlays | The second code focuses on color detection, contour identification, circularity checks, and displaying object information. |
| **Distance Calculation** |  | The first code includes a distance calculation based on the object's radius and camera parameters.  The first code performs distance calculation based on camera parameters and trigonometric functions. These calculations involve more mathematical operations, which can be computationally expensive. |  |
| **Blob Detection** |  | The first code uses the Hough transform (cv.HoughCircles) for circle detection in each frame.  The Hough transform used in the first code is a more computationally intensive algorithm compared to contour identification and circularity checks used in the second code, especially for higher resolution images, as it involves a lot of mathematical computations for each pixel.  The Hough transform considers a wide range of possible circles, leading to more processing time and higher computational requirements.  Involves computationally intensive operations such as edge detection, gradient computation, and voting. | The second code uses contour identification (cvzone.findContours) and circularity checks to detect circles.  Which is generally faster and computationally lighter. |
| **Color detection: Both codes convert the frame to the HSV color space for color detection** |  | It applies a color mask, erosion, and dilation operations, followed by grayscale conversion and median blur.  Code performs this conversion for every frame, making it more computationally expensive. | Utilizes the ColorFinder module from cvzone  Code perform it only once, making it less computationally expensive. |
| **Extensive Features** |  | The first code includes additional functionalities such as graphical overlays, object tracking, and complex color detection.  These additional features require more processing power and memory utilization compared to the simpler functionality of the second code. |  |
| **Multiple Function Calls** |  | The first code calls the detect\_colored\_object and display\_object\_info functions for each color in every frame. These function calls involve overhead, and this overhead can add up when the functions are called multiple times. |  |
| **Iteration and Dictionary Operations** |  | Iterates over a dictionary of colors and performs operations for each color. It updates the dictionary with new values for each detected object. These dictionary operations and iterations can be computationally expensive, especially when dealing with a large number of colors or objects. | Handles the detected contours directly, which simplifies the process and reduces computational overhead. |
| **Frame Processing** |  | Resizes the frame using the imutils.resize function, which can be computationally expensive if the frame size is large. | The second code does not resize the frame, resulting in faster processing. |

Considering the limited computational resources of a Raspberry Pi 4 without a dedicated GPU or TPU, the second code is less computationally expensive and more suitable for running on such a system. Its simplified functionality and reduced computing make it more efficient in terms of resource usage, enabling smoother execution on resource-constrained devices like the Raspberry Pi 4.

In summary, the second code is likely to be less computationally expensive due to its simplified object detection approach, efficient color detection, direct handling of contours, and reduced usage of external libraries.