Image Analysis for Visual Victims on Low Powered Machines

Daniel Xue and Ethan Wu

Storming Robots, Branchburg NJ, USA

Abstract. Computer Vision is a field applicable to many applications in modern life. But how does one balance the demands of high speed performance while deploying to hardware which potentially needs to be very small in size and cheap in construction? In the process of solving the tasks presented by the RoboCupJunior Rescue Maze Competition, we combined the use of a costume designed vision apparatus to minimize the space needed along with a specialized original algorithm capable of running on low performance machines, allowing for the deployment of letter detection algorithms on a very small and compact robot.

1 Introduction

The field of Computer Vision has, in these few years, permeated the forefront of modern research. Such has also been reflected in the increasing importance of Computer Vision in Robotics, and as a result many high school level competitions have began incorporating the usage of the field in their tasks. This presentation is about the usage of Computer Vision in the RoboCupJunior Rescue Maze Competiton.

1.1 The Task

The task presented by the RoboCupJunior competition involved the detection and interpretation of letters. Specifically, a robot must distinguish the letters, with unfixed font, ”H”, ”S”, and ”U” autonomously. Furthermore, since the competition involved rather small autonomous vehicles, the processing power is rather limited. Thus, the challenge is not only to devise an algorithm to interpret these letters, but also to make an algorithm extremely efficient in its usage of processing power and very robust and adaptable in its ability to distinguish characters that may not be the same as what one may expect.

2 Hardware Design

Due to the limited space available on these vehicles, it is desirable to utilize as few cameras as possible. When using common single board computers (SBCs) like the Raspberry Pi Zero, one SBC can typically only support one camera, and there is very limited space available for these SBCs. However, the robot needs to be able to detect letters on both sides of itself while passing walls, or else the robot would have to make multiple passes to scan both sides. To save space and time, a split mirror is used to allow one camera to view both sides of the robot at the same time. A configuration consisting of a camera pointing up to two mirrors, each occupying half of the camera's field of view and reflecting in opposite directions, is used (see Fig. 1). It is constructed out of a 3D printed frame for accuracy and design flexibility, and utilizes plastic reflective elements for ease of construction, reduced weight, and safety.

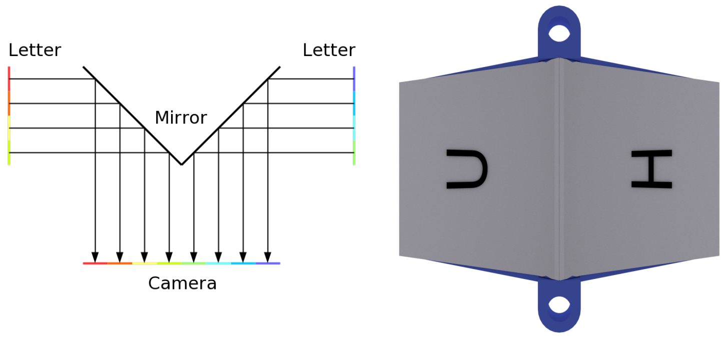


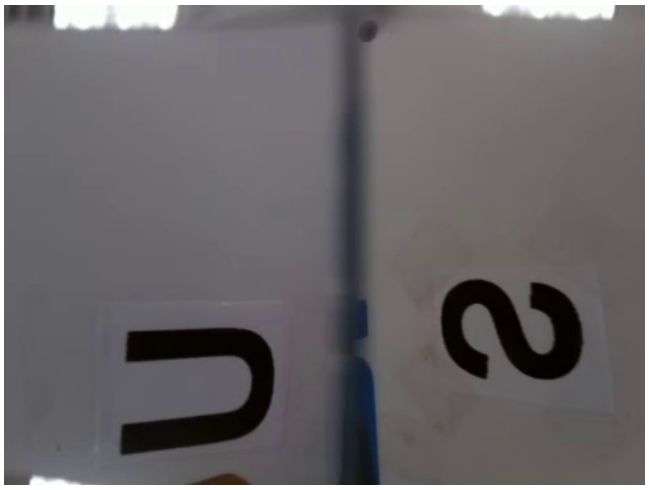
Fig. 1: Mirror design for viewing both sides of robot with one camera

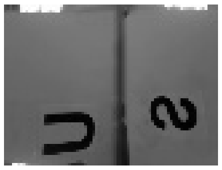
3 Image Processing Algorithm

The interpretation of a live image feed utilized the power of OpenCV, an open source computer vision library. This library contains a myriad of functions to analyze and transform the image, which when used in conjunction can yield algorithms to do tasks such as the one presented. This presentation focuses on the usage of basic functions in the OpenCV library in efficiently determining the letter seen by the camera.

3.1 Preprocessing

The first part of interpreting the image is preprocessing. The goal of this stage is to reduce the possible sources of error and noise. The first step is resizing the image to a much lower resolution. A letter is clearly identifiable even at a very low resolution. Downscaling is able not only to reduce any noise, such as unwanted background imagery, but more importantly also allows further processing to be much quicker. Second, the image is thresholded—converted so each pixel is either black or white, and then inverted. Since the letter is meant to be black on a white background, thresholding makes the image clearer and easier to work with while not changing anything meaningful about the image.


(a) Unprocessed camera feed (scaled by .5x)


(b) Downscaled camera feed

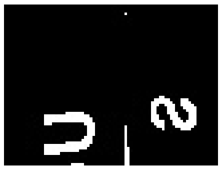
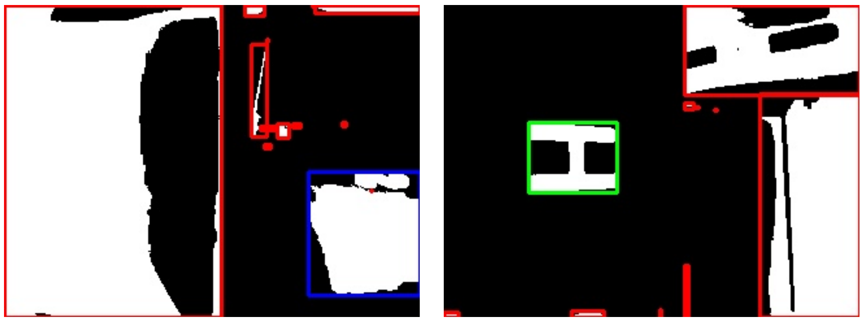

(c) Thresholded and inverted camera feed

Fig. 2: Unprocessed and processed camera feed comparison

3.2 Extracting Regions of Interest

The first step in identifying the letter is to reliably extract the specific region of interest, i.e. a bounding rectangle around the letter. The first step is to find the contours in the image (through OpenCV's `findContour` function), and from there find the bounding rectangles of the contours. However, contours detected are not always letters. In fact, they may be imagery that the camera sees over the walls or past the mirror (lights and such, see top of Fig. 1). Furthermore, one might find a letter, but it may happen that this letter is actually in an adjacent tile. Thus, the extraction of regions of interest must be robust enough to reject these sources of error.

Error Rejection There were a number of criteria that was required for a bounding rectangle to pass error rejection. The first was the distance of the bounding rectangle from the edge. A letter that is touching the edge of the image is likely also to be partially cut off , making it incomplete and thus not fit to be analyzed. The second is the size of the bounding rectangle. The bounding rectangle must be large enough that one can confidentially say it is on the tile right besides the robot. Third, the contour must be of a specific aspect ratio. This is probably the most rigorous of the three criteria, as most common contours would be shaped as long, thin stretches, as opposed to letters which are more close to a square (depending on the font). These three steps are able to yield a confident prediction of whether or not the region of interest is a letter.



(a) Ex. 1–no misidentified regions (b) Ex. 2–successfully identified letter

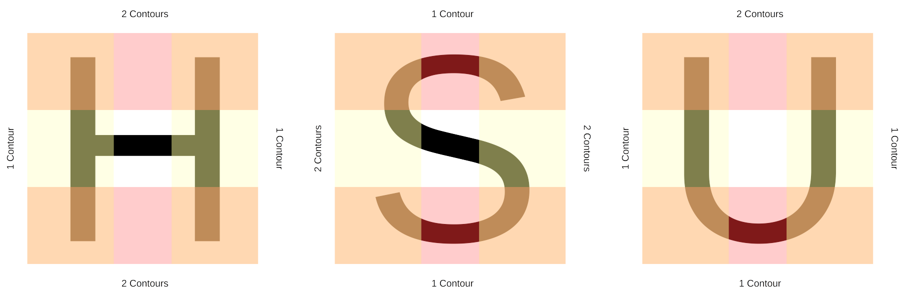
Fig. 3: Regions of Interests Extracted from Image. Red: Rejected by first error rejection (§ 3.2). Blue: Identified as a possible letter but rejected due to contours not matching (§ 3.3). Green: Identified letter (§ 3.3).

3.3 Interpretation

'Finally, after extracting the regions of interest, it is finally time to determine exactly what letter we are looking at. To do this, there are two different approaches that were tried.

Contour Analysis This was the initial, and currently deployed approach. This method is hinged upon the fact that letters that must be distinguished are of a very limited scope. The 3 letters, "H", "S", and "U" are very clearly distinguishable. Thus, this solution was very specific in scope, which means it cannot quite be generalized to a whole alphabet, but it is very consistent for the given letters and is very processing efficient. The method works like such: the four sides of each letter had either one or two contiguous regions (e.g. "H" has 2 on the top and bottom and 1 on each side). The specific combination of how many contiguous regions exist on each side of the letter is distinct between the 3 letters. Thus by counting the number of contours on each side of the image, one can figure out whether it is a "H", "S", or "U".

This method is advantageous for two reasons. One, the general shape of a letter doesn't change even if one were to change the font or distort the image in some way. This increases the robustness of the algorithm. Second, the very specific combination of the number of contours on each side serves as a very rigorous gate for potentially misidentified regions of interest that may have made it through the initial error rejection process.



(a) H Contours (b) S Contours (c) U Contours

Fig. 4: Contours of each side of "H", "S", and "U"

K-Nearest Neighbors A second method that we investigated is the usage of the K-Nearest Neighbor (KNN) algorithm. As a brief overview, KNN operates through the comparison of a given data set and the trained data set. The trained data points with the closest Euclidean distance to the current data are the "neighbors", and thus the letter in which closest neighbors are is determined to be the letter of the given data set [3]. The Euclidean distance can then be used to judge how close of a fit the data set is to the trained set, being able to reject regions of interest that mistakenly got past error rejection. Our implementation of the KNN algorithm used each individual pixel from a resized version of the bounding rectangle found earlier.

Comparison The usage of KNN has its advantages compared to the first method, however we determined it to be ultimately worse. The primary advantage is that it can be generalized to many letters, as long as the training set was expanded. This advantage however, made very little difference in the scope of its usage in which only 3 letters needed to be determined. The drawbacks however, were much more influential on the competition. Our KNN algorithm was significantly less consistent than the contour analysis algorithm. Since we used a data set of all the pixels in a given region, this method was more sensitive to minor differences in the image such as distortion, lighting, or font type. This led it it being far less adaptable and thus accurate than the contour analysis method. On average, given a stable test environment and the same font as the training data, the two algorithms had a very similar accuracy rate of over 95%. However, anecdotally (since the data is not immediately available) with different fonts, (still sans serif, though some more unconventional) the contour method performed more consistently, and also performed consistently across redesigned iterations of the camera module and mirror, while the KNN algorithm likely would not weather such deviations from its dataset.

3.4 Performance

The ultimate goal was to have the software running on a very small SBC such as the Pi Zero (later we did upgrade the platform, but not due to issues in performance of the core algorithm). Such a computer has relatively limited processing capabilities (single core at 1GHz). With the aforementioned contour analysis algorithm, we were able analyze at 30fps in real-time, which maxed out the framerate of the camera. Thus this specialized algorithm, which through the usage of in general only basic OpenCV functionality, ran both consistently and on a system which was low performance, cheap, and compact.

4 Deployment

The vision system is designed to run on a Linux SBC, and often when deploying such projects, managing dependencies becomes an issue. To alleviate this issue, we utilized Docker [4] to build containers, which bundle all runtime dependencies (essentially the filesystem of a stripped-down Linux installation) and are entirely portable. A Docker-based workflow also allows running these containers—built for ARM—to run on x86 based machines with QEMU, enabling testing on x86 laptops or desktops exactly as it would run on the real robot hardware. Building these containers can also be done with x86 hardware, including continuous integration servers in the cloud such as Gitlab CI [1]. Thus long builds such as OpenCV need not be performed on the limited-performance SBC itself. These base images are publicly available on our GitLab page [2].

5 Conclusion

All in all, by combining the usage of hardware and software designed with the task in mind, one has the potential to create a product both efficient in space and sufficiently fast in performance. In the case of our experience in RoboCupJunior, our original solutions allowed us to create an extremely compact implementation of a normally fairly intensive process—letter recognition. Such can be applied to any project in that one must consider deeply what the individual nuance of one's project is and how a solution can be developed around it, instead of simply taking a more off-the-shelf solution.

References

1. Angelatos, P.: Building ARM containers on any x86 machine, even DockerHub (December 2015), `https://resin.io/blog/building-arm-containers-on-any-x86-machine-even-dockerhub/`
2. Embedded Linux components, `https://gitlab.com/beings-rcjb/elinux`
3. Understanding k-nearest neighbour, `https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/py_knn/py_knn_understanding/py_knn_understanding.html`
4. What is Docker?, `https://www.docker.com/what-docker`