Team 3044 Driverstation Image Processing

Joseph Lyon

https://www.github.com/jlyon1/StrongholdVision

Contents

Introduction

Foreword: The purpose of this document is to describe the problems the programming team on FRC Team 3044 determined were present in the development of our vision solution, as well as how we solved these problems. The document will be separated out into a problems section, where problems will be listed, and implementation section, where I will discuss some implementation, and a conclusion section where I will propose improvements and other items. The document will reference classes in the actual code, some of which will not be immediatley shown, but can be found by going to the github page where they are stored.

Applications in FIRST Stronghold: Stronghold presented a unique set of reqirements as a FIRST game, it's small goals relative to the size of the ball made some form of assisted aiming a necessity. This could come in the form of an actual sight, or through image processing. Teams have used tape or lines on the screen as a sight, however this comes with the drawback of not being usable in autonomous and makes the driver responsible for aligning the robot. Taking these factors into consideration, our team decided on using driverstation software to process images and send the results back to the robot.

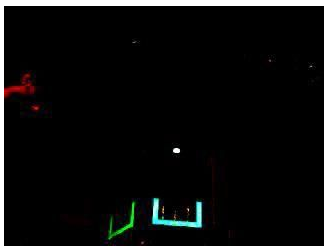
Challenges and Limitations:

- **7mbps Bandwidth limitation:** FMS imposes a 7mpbs bandwidth limitation on robots on the field, this becomes an issue for higher resolution images, and for faster image acquisition
- **Network Latency:** Latency on the network becomes an issue for fast processing and acting on processed information. Aligning the robot with only a PID Loop with the camera as an input requires fast network transmission.
- **Speed of the processing loop:** In order to obtain the most recent information for use on the robot, the processing time must be fast. Our driverstation laptop does not have a graphics processor, only a CPU with integrated graphics processing. The result of this is that the algorithm must be efficient, and require as little processing as possible in order to ensure that it is functional on the robot.

- **Illuminating the target:** In order to track the target, a light source must be used to illuminate the target, which is made of retro reflective tape.
- **External Light sources:** Many regionals have external light sources that can interfere with aformentioned tracking, including large lights, as well as windows and other sources of interference.

Problems

Image Acquisition



- Our team very quickly settled on an axis m1011 camera for this purpose, as it does not have the adjustable focus of the axis m1013, and is easily configurable through its web interface and api. The Camera allows you to modify image settings and other information to achieve dark images with just the target illuminated, that are of a very low resolution, such as the one pictured above. The camera is wired into a 5v power supply on the voltage regulator module, and it's ethernet port is wired into a separate dlink bridge on our robot which is wired into the main router, however, the same task could be accomplished with any usb camera and some code on the roboRIO or a coprocessor. We decided against using either of these methods as the Axis camera required the least setup and would likley be more reliable.

Target Illumination

- To Illuminate The target, we used two bright green LED Rings from SuperBrightLEDs.com, one smaller ring, and one larger ring, the same types of rings have been provided by andymark and through FIRST choice in the past. These LED rings may be wired to any 12v 500mAh power source.

Noise Reduction

- We modified camera settings in order to achieve the most easily processible image from the camera, the settings can be modified through the axis web interface (shown right) and are as follows
- **Resolution:** 320x240
- **Max Frame Rate:** Unlimited
- **Color Level:** 100 – Max
- **Sharpness:** 100 – Max
- **Brightness:** 0 – Min

- **Exposure:** 0 – Min
- **White Balance:** In order to get an accurate white balance for each venue, we followed the following procedure:
    - Aim camera at target
    - Set white balance to “Auto” for about 15 seconds
    - Set white balance back to hold current

Processing

- We quickly determined that the best way to process images would be through using opencv 3.0.0, as some members of the team had previous experience with opencv, It is a very efficient library, and it has a significant amount of community support, which meant that we would be able to utilize the community for help with troubleshooing if need be.
- We also settled on a Driverstation Side processing algorithm, which generated a few other issues that would not have been present if a coprocessor were used, such as network latency and the transmitting of information to and from the robot.

Transmitting information to robot:

- The FIRST wpilib provided network tables library seemed like the most obvious choice for transmitting information to and from the robot. The network tables library is implemented for both the roborio and for most desktop purposes, the benefit being

Network transmission speed and Processing efficiency

- There is a signifigant amount of latency over the network, to the point where the slowest part of the processing loop is image acquisition rather than processing. This relationship between processing speed and acquisition speed lends itself to a separate acquisition thread rather than one single loop where an image is aquired, and then processed. We determined that a triple buffer would be adaquate for this purpose. This means that the software always has two full images in it’s buffer while it is aquiring the new image. This prevents the processing loop from having to stop to aquire images, which greatly increases the speed of the processing loop.

Implementation

Algorithm

1. Connect to the RoboRIO and Connect to the Camera, This is done using the two functions shown below, they are both fairly self explanatory, but comments are included where needed.

```
public boolean openCamera(ConsoleWindow console) {
    console.println("Initializing VideoCapture Camera: " + cameraAddr);
    this.console = console;
    //cameraAddr is the string ip address of the camera in this case:
    //"http://10.30.44.20/axis-cgi/mjpg/video.cgi?test.mjpeg" with a dummy
    //param ?test.mjpg so opencv recognizes the stream as an mjpeg stream
    camera.open(cameraAddr);

    try {
        Thread.sleep(300);
    } catch (InterruptedException e) {

        e.printStackTrace();
    }

    if (camera.isOpened()) {
        return true;
    } else {
        return false;
    }

}
```

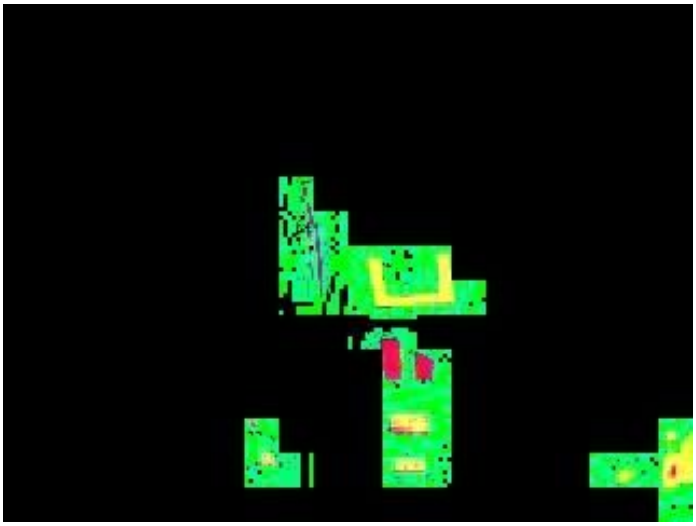
```
public void connectToRobot(String host) {
    //Most networking is handled by the wpilib
    //NetworkTable class and the static Robot class
    Robot.setHost(host);
    Robot.setUseMDNS(true);
    Robot.setTeam(3044);
    visionTable = NetworkTable.getTable("SmartDashboard");
}
```

2. Create and begin the image acquisition thread, this is done through the Axis Grabber class, and begins aquiring images at as fast of a rate as the camera will allow.

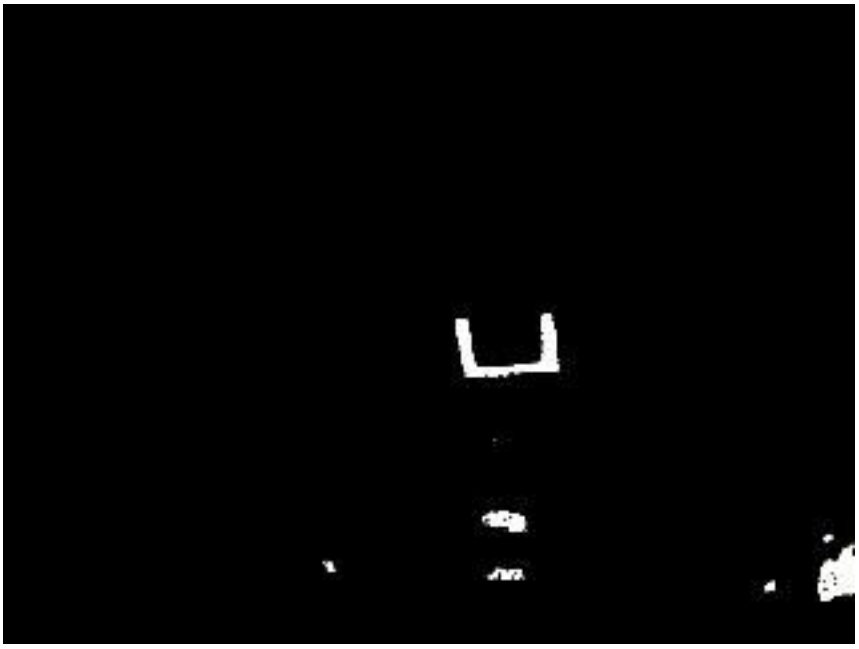
```
public void run() {
    running = true;

    while (running) {
        //Grab a new image and store it in the buffer
        Mat temp = new Mat();
        cap.read(temp);
        buffer[i] = temp;
        //Grab the time the image was read and store it in another
        //array of Times
        timeTags[i] = System.currentTimeMillis();

        j++;
        i++;
        //The buffer is 3 items long, reset the iterator if it is too high
        if (j > 2) {
            j = 0;
        }
        if (i > 2) {
            i = 0;
        }

    }

}
```

3. Once the initialization algorithm completes, the processing loop begins (Some theoretical images will be used rather than actual ones in order to portray concepts). The loop begins by

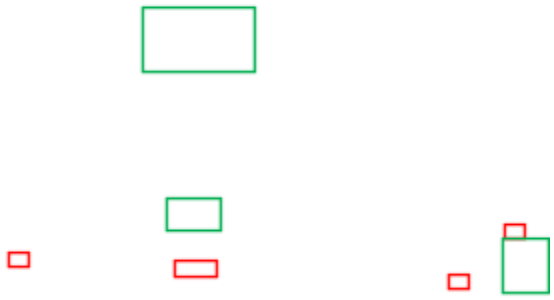
converting the most recent acquired image to HSV color space, creating an image like the one below.



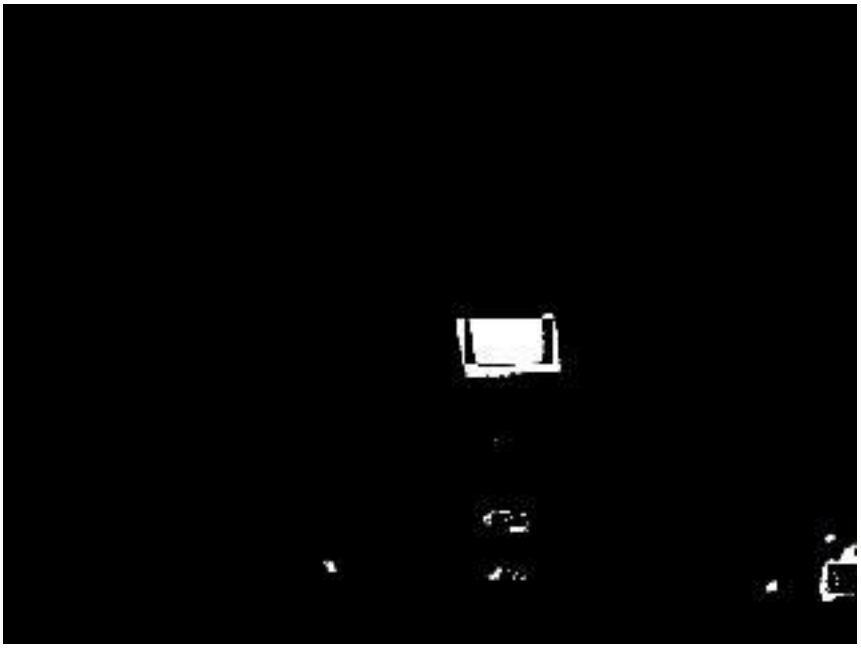
4. After the image is converted, it is binarized using a binary threshold algorithm. Every pixel within a specified HSV Range is set to white, and every pixel not in the range is set to black, we then erode and dilate the image to get rid of noise, producing an image similar to the one below.



5. After This, contours are drawn around the different “Blobs” and bounding boxes are created around those contours, we then throw out anything with an obviously too large or too small of an area, or with an aspect ratio that is not appropriate. As shown below, this removes most of the noise.



6. The middle seventy-five percent of each rectangle is then inverted, and a score calculated for each rectangle, the score is calculated by summing up the number of white pixels in the image and dividing by the area, the closer this value is to one, the more likely the image is the target. As seen below, the target is mostly white, this creates a higher density.



7. As a final measure, we look at the number of sides of the contour or shape, the target contour will ideally have eight sides, and we check as such. If it meets all of these criteria, it is added to an array of possible targets, and once all have been processed, the center of the largest target is sent to the robot for aiming.