We are FRC Team 4004, the MARS Rovers. This season, we worked with a JeVois camera for machine vision. We were attempting to track strips of retro-reflective tape using Python modules run on the JeVois camera. The core processor of the camera is run on C++, however, you are able to write Python scripts that are then implemented by the camera.

Mockup of the vision targets on the field.

The way we went about making this retro-reflective tape stand out to our camera (which enables us to reliably locate the tape) was to shine infrared light at the tape, which light would then be reflected back into our camera, making the tape appear very bright. We also talked with team 4003, Trisonics, and they gave us some pointers on how to filter out abstract noise when using machine vision. They showed us that floppy disks actually filter out most all light spectrums other than infrared light. So, after doing some tests, we decided to put a piece of a floppy disk over our camera lens, effectively filtering out everything but our illuminated vision target.

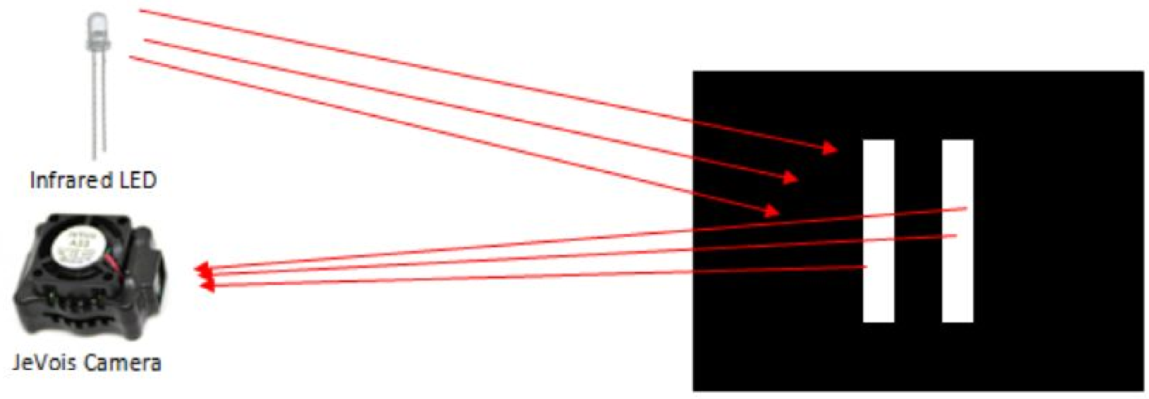
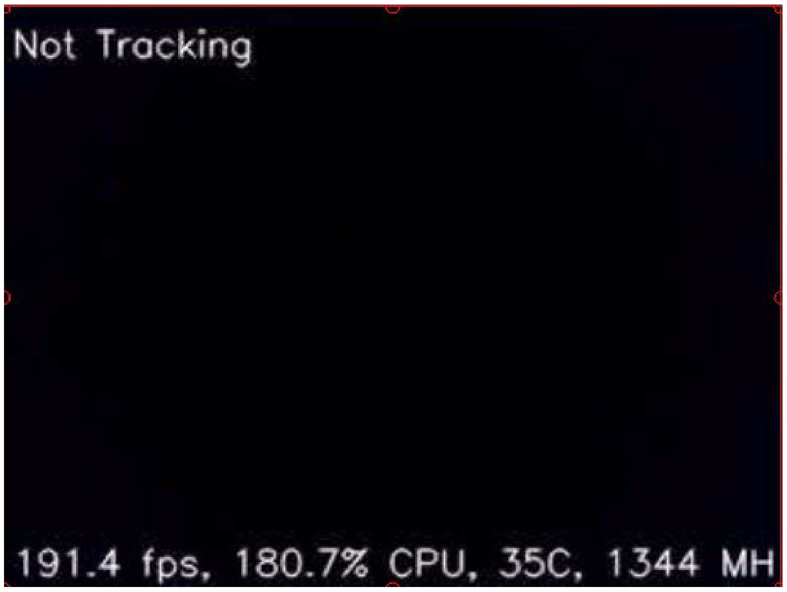


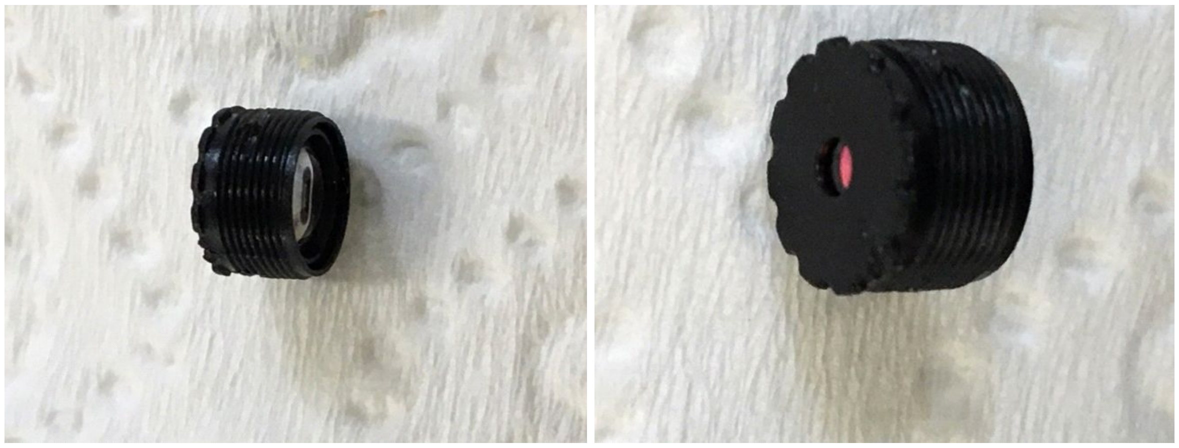
Diagram depicting our thought process for tracking the vision target.

However, after initial testing, we discovered that the JeVois camera has an internal filter that blocks infrared light. This prevented the camera from seeing infrared light, which kept us from using the camera to track the reflective tape. In order to properly test the effectiveness of our piece of floppy disk as a filtering device, we had to first remove the infrared filter from our camera.

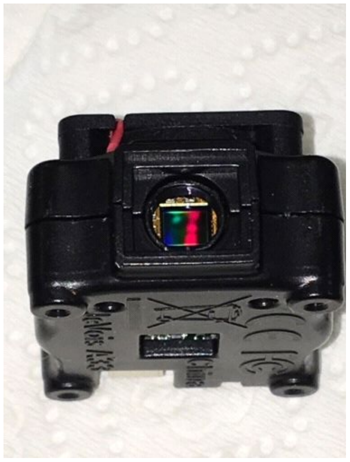


Screenshot from camera pre-infrared filter removal.

We contacted the people at JeVois Incorporated, and told them our situation. They confirmed that there is an infrared filter on their cameras, and they gave us some advice on how to remove it. First, we unscrewed the lens from the camera.



Camera lens pre-infrared filter removal. The shiny red reflection is the IR filter.

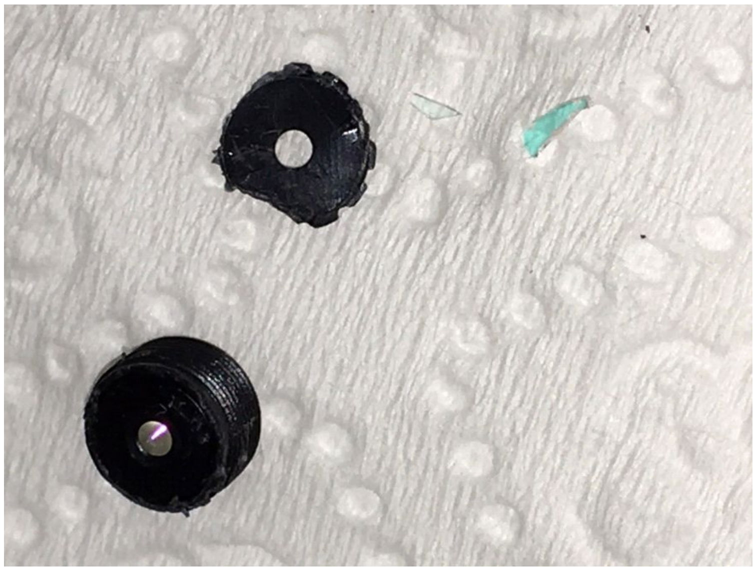


JeVois camera with the lens removed.

Once we removed the lens, we weren't sure if the infrared filter was on the CCD chip of the camera, or on the lens itself. So we talked some more with the people at JeVois Inc., and they gave us some more information about the guts of the camera. We eventually found a website that shows you the steps to remove the infrared filter from a generic digital camera. (The link is below)

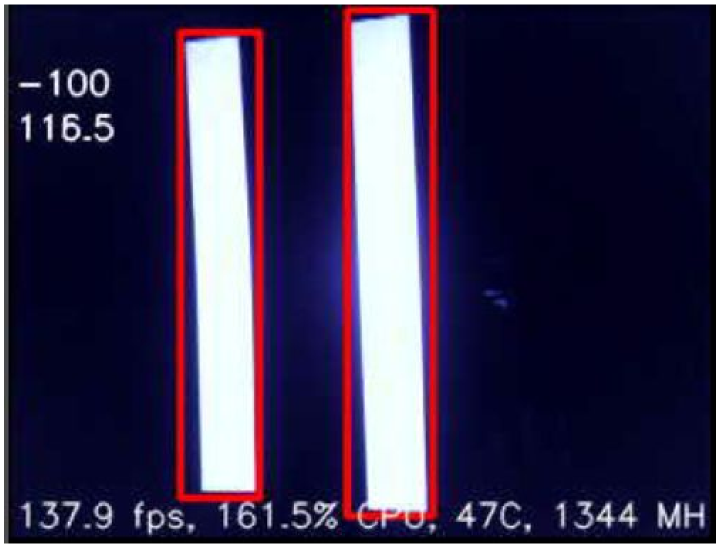
http://www.instructables.com/id/infrared-digital-camera---the-real-way/

We then applied those steps to our JeVois camera, determined that the infrared filter was on the outside of the lens, and carefully cut the filter off of the lens with a razor knife.



The camera's lens post-infrared filter removal. The small blue fragments used to be part of the infrared filter.

After cutting off the infrared filter, we reinserted the lens into the camera, and powered the it back up.

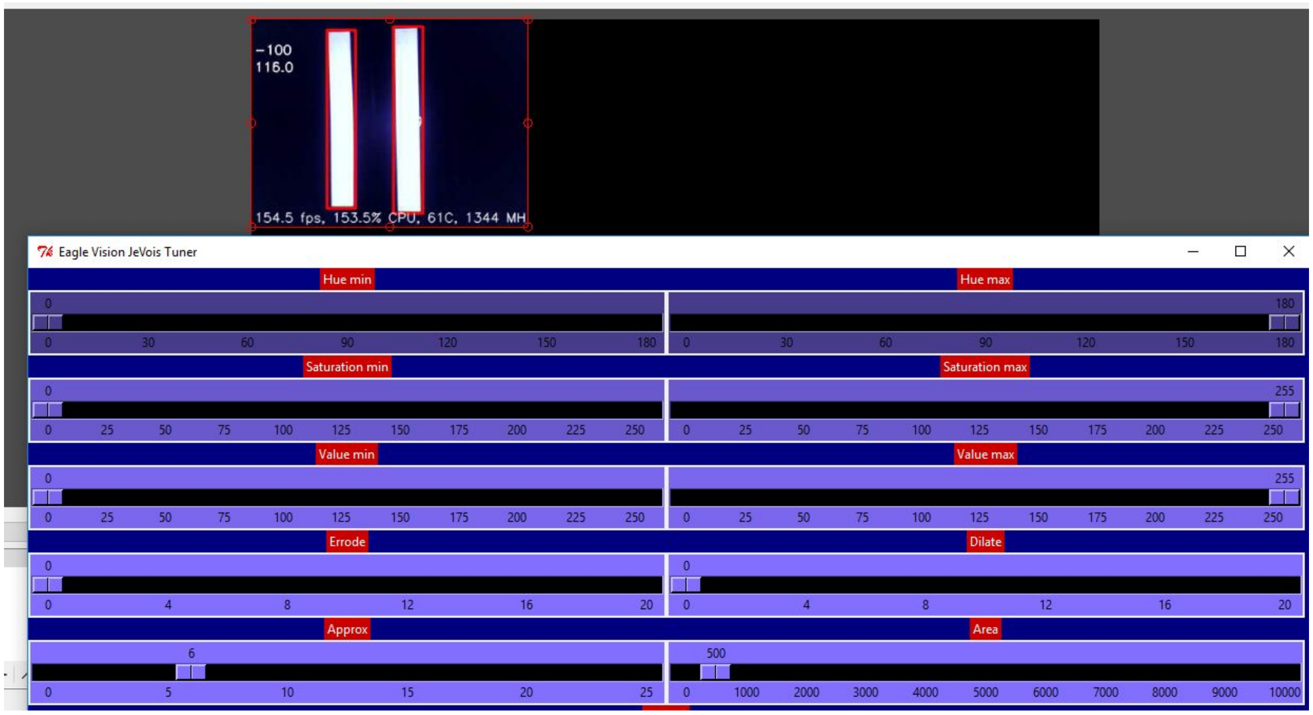


Screenshot from camera

As you can see in the picture above, once the infrared filter was removed from the camera, the retro-reflective tape appeared very bright to the camera. Having successfully filtered everything out of the image except for the retro-reflective tape, we moved on to tracking the tape in code.

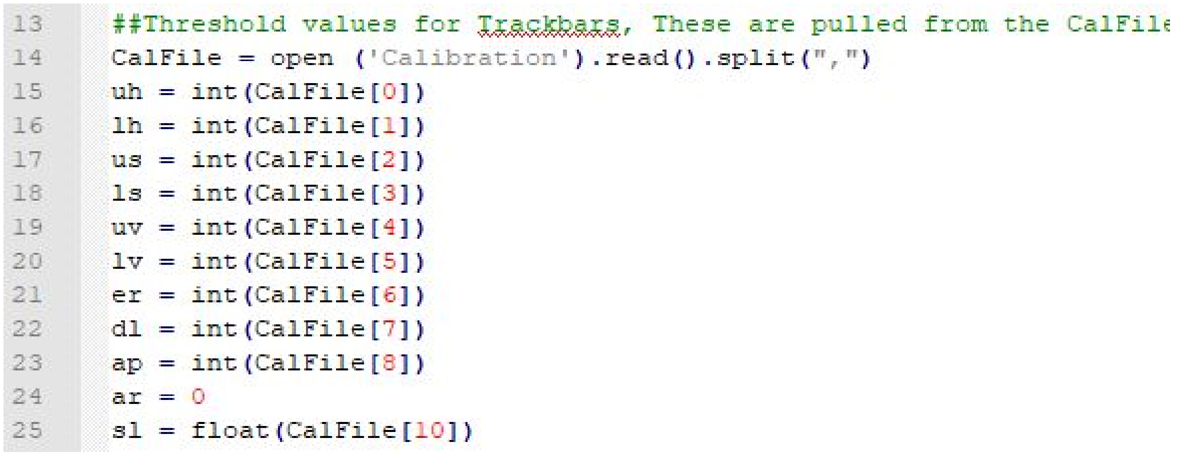
We started by getting familiar with how to operate the JeVois camera, how to write simple python modules, and how to run those modules on the camera. Then we found a White Paper created by team 2073, EagleForce, which paper helps you get started using the JeVois camera. After we looked over the White Paper, as well as doing extensive research on the core documentation for the JeVois camera, we built off of some of 2073's provided sample code, and created our own python modules to track the vision targets in this year's game.

We have two main python modules that we run, our calibration module, and our tracking module. We run our calibration module to tune the HSV values that the camera is looking for, as well as tune the size of the shape that the camera is looking for (see the picture on the next page).



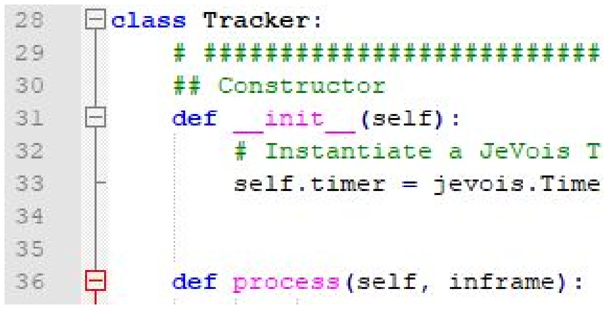
Python calibration module

Then, once we are satisfied with the values that the camera is tracking, we store these values in a calibration file which is kept in the onboard memory of the camera. Now, when we run the python module that actually does the tracking, we read the values from the calibration file we created earlier, and apply them in the acquisition of our target shape (as seen in the snippet of code below).



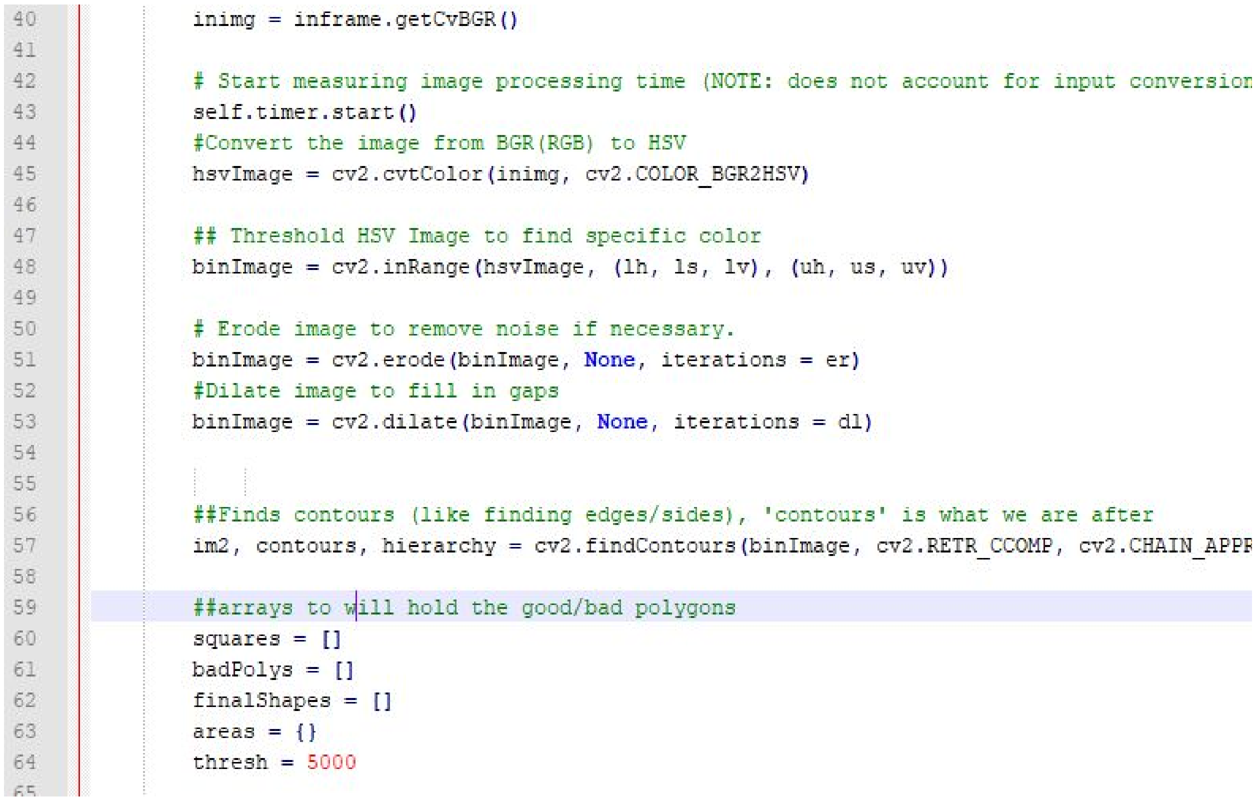
```
##Threshold values for Trackbars, These are pulled from the CalFile
CalFile = open ('Calibration').read().split(",")
uh = int(CalFile[0])
lh = int(CalFile[1])
us = int(CalFile[2])
ls = int(CalFile[3])
uv = int(CalFile[4])
lv = int(CalFile[5])
er = int(CalFile[6])
dl = int(CalFile[7])
ap = int(CalFile[8])
ar = 0
sl = float(CalFile[10])
```

In order for the core C++ framework of the camera to easily access our python module, we define a class called *Tracker*, and a method called *process*. The *process* method takes two parameters, *self* (which is a parameter in almost every python method), and *inframe*. *Inframe* is the current image that the camera is seeing; is what you extrapolate, and process, to locate the vision target. The *process* method is called by the core C++ framework of the camera once per frame; so it runs continuously, and is where the bulk of our code resides.



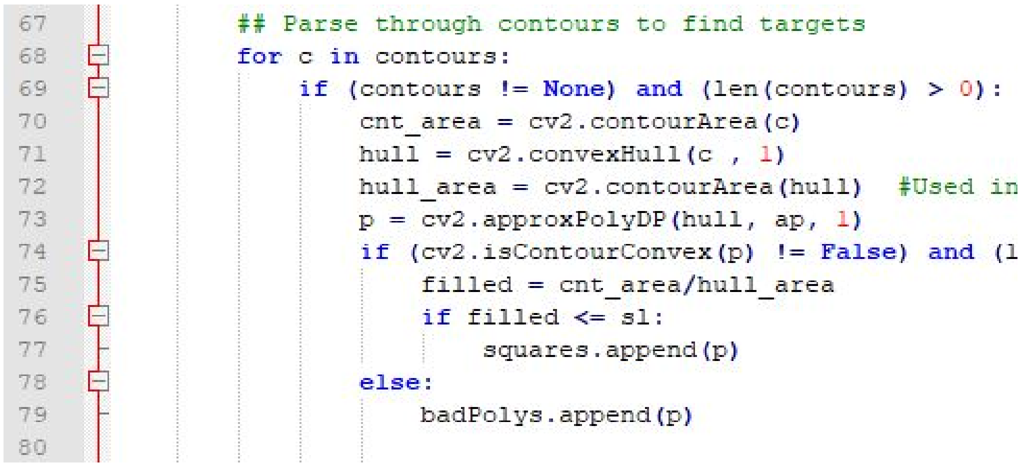
```
class Tracker:
    # ##########################
    ## Constructor
    def __init__(self):
        # Instantiate a JeVois T
        self.timer = jevois.Time


    def process(self, inframe):
```

Next, we get the current image the camera is seeing in “BGR” (**B**lue **G**reen **R**ed) form using *inframe.getCvBGR()*, and put it into a variable called *inimg*. We then convert the image to HSV values, dilate the image as needed, erode the image as needed, and define some arrays and dictionaries that will be used later in the code (the values that we dilate and erode the image are set in our calibration module).



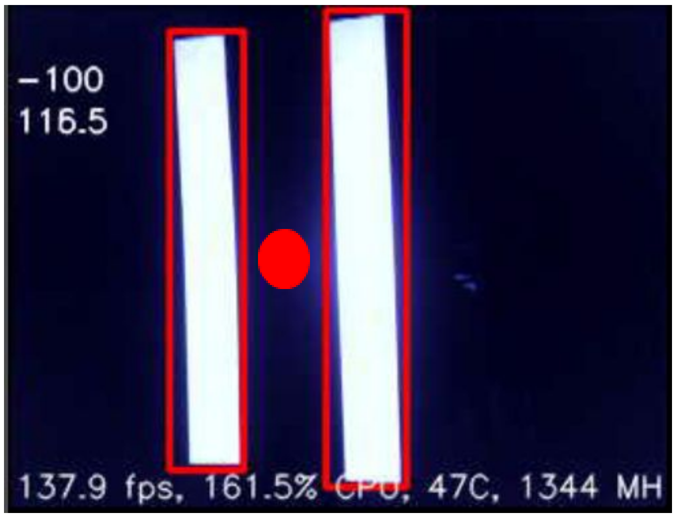
```
        inimg = inframe.getCvBGR()

        # Start measuring image processing time (NOTE: does not account for input conversion
        self.timer.start()
        #Convert the image from BGR(RGB) to HSV
        hsvImage = cv2.cvtColor(inimg, cv2.COLOR_BGR2HSV)

        ## Threshold HSV Image to find specific color
        binImage = cv2.inRange(hsvImage, (lh, ls, lv), (uh, us, uv))

        # Erode image to remove noise if necessary.
        binImage = cv2.erode(binImage, None, iterations = er)
        #Dilate image to fill in gaps
        binImage = cv2.dilate(binImage, None, iterations = dl)


        ##Finds contours (like finding edges/sides), 'contours' is what we are after
        im2, contours, hierarchy = cv2.findContours(binImage, cv2.RETR_CCOMP, cv2.CHAIN_APPR

        ##arrays to will hold the good/bad polygons
        squares = []
        badPolys = []
        finalShapes = []
        areas = {}
        thresh = 5000
```

After that, we cycle through all of the contours found in the current image, and weed out all of the contours that aren't four sided (we only want four sided polygons because we are tracking two rectangles).



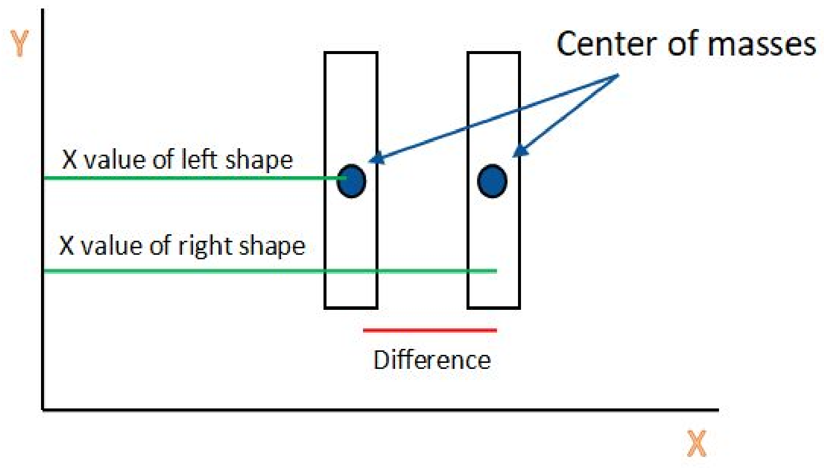
```
        ## Parse through contours to find targets
        for c in contours:
            if (contours != None) and (len(contours) > 0):
                cnt_area = cv2.contourArea(c)
                hull = cv2.convexHull(c , 1)
                hull_area = cv2.contourArea(hull)  #Used in
                p = cv2.approxPolyDP(hull, ap, 1)
                if (cv2.isContourConvex(p) != False) and (l
                    filled = cnt_area/hull_area
                    if filled <= sl:
                        squares.append(p)
                else:
                    badPolys.append(p)
```

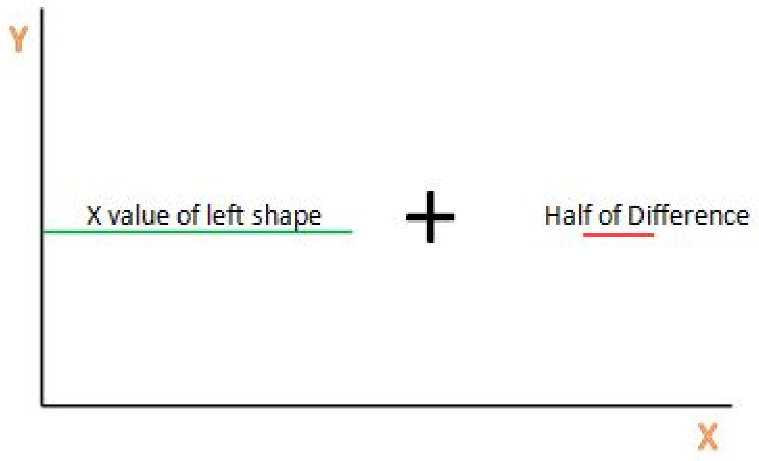
Then, once we have located the two rectangles, we calculate the center of mass of each of the two rectangles, subtract the leftmost rectangle's *x* value from the rightmost rectangle's *x* value, divide the difference by 2, then add that number to the leftmost rectangle's *x* value. This gives us the center point between the two rectangles, which value we then output to the roboRio as serial data via the RS-232 port on the roboRIo. (We are only concerned with the *x* value of the vision target, because the robot will only be moving laterally when aligning itself with the target)

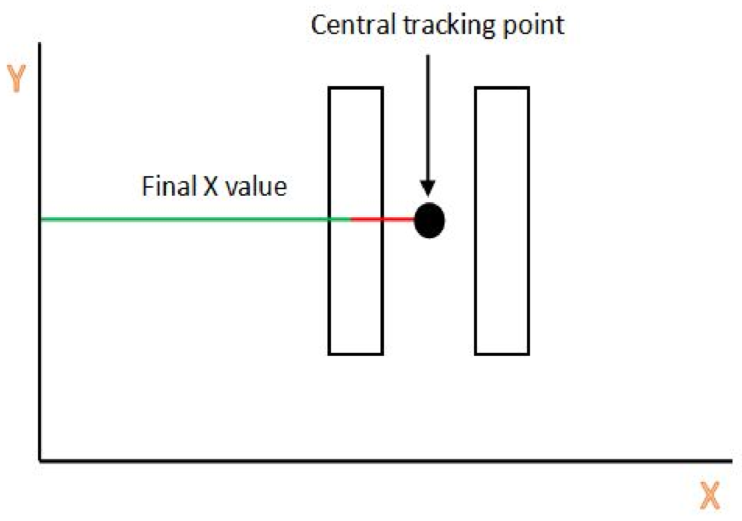


The red dot is the point in space we are tracking.

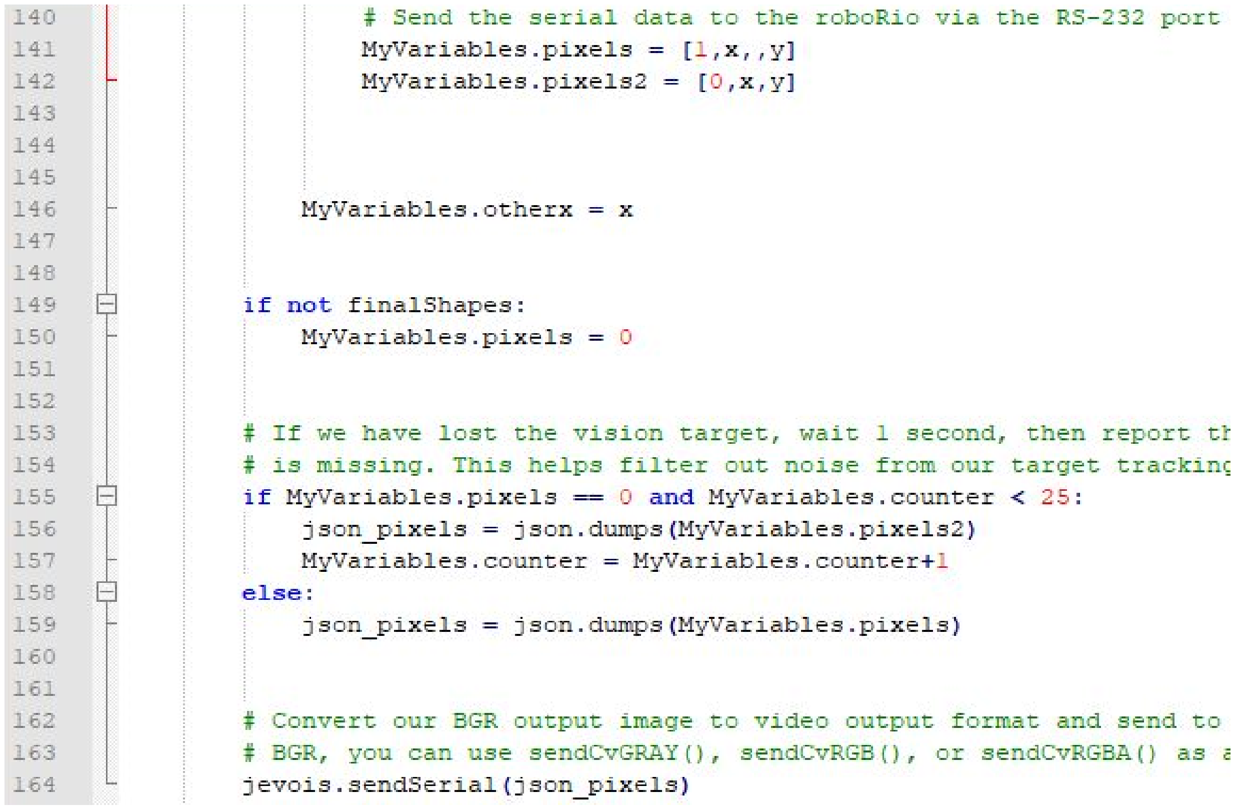
Below are some diagrams that visually demonstrate our method of locating the center point between the two rectangles.





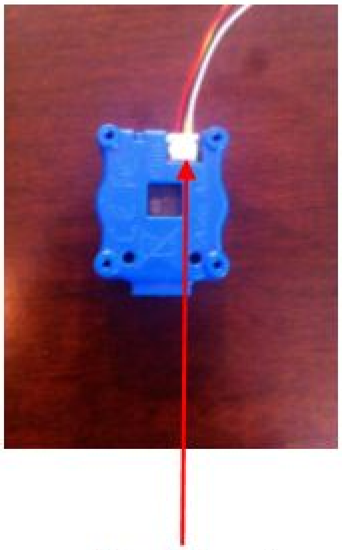


Once we have calculated the central tracking point, we take all of the data that we want to send to the roboRio, and store it in an array called *pixels*. If you look at the code below, it is actually called *MyVariables.pixels*, but that is just because *MyVariables* is the class that *pixels* is instantiated in. We use the class *MyVariables* for all of our global variables. Once we have the information we need, we then check to see if we have lost the target for a set amount of time. If we have lost the target for that amount of time, then we just send 0s to the roboRio, otherwise, we send the *x*, and *y* data of the target to the roboRio. We push the data out of the camera using a function called *jevois.sendSerial(WhatEverDataYouWant)*, which function outputs the desired serial data to either the micro-JST port on the camera, or the serial-over-USB port on the camera (see code below). The port through which the serial data flows is defined in the boot script of the JeVois camera itself. For our purposes, we send the serial data out through the micro-JST port.

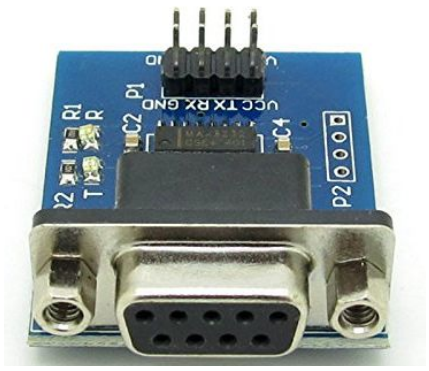


```
        # Send the serial data to the roboRio via the RS-232 port
        MyVariables.pixels = [1,x,,y]
        MyVariables.pixels2 = [0,x,y]



    MyVariables.otherx = x


if not finalShapes:
    MyVariables.pixels = 0


# If we have lost the vision target, wait 1 second, then report th
# is missing. This helps filter out noise from our target trackin
if MyVariables.pixels == 0 and MyVariables.counter < 25:
    json_pixels = json.dumps(MyVariables.pixels2)
    MyVariables.counter = MyVariables.counter+1
else:
    json_pixels = json.dumps(MyVariables.pixels)


# Convert our BGR output image to video output format and send to
# BGR, you can use sendCvGRAY(), sendCvRGB(), or sendCvRGBA() as a
jevois.sendSerial(json_pixels)
```

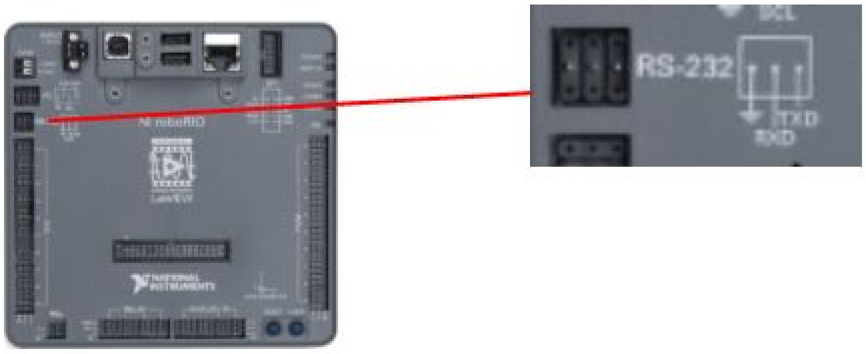
As I mentioned before, we send our tracking coordinates from the micro-JST port on the JeVois camera, to the RS-232 port on the roboRio. But in order to do this, we first have to convert the serial data that the camera outputs from TTL logic levels, to RS-232 logic levels, which is done with a converter board (see images below.)



Micro-JST port



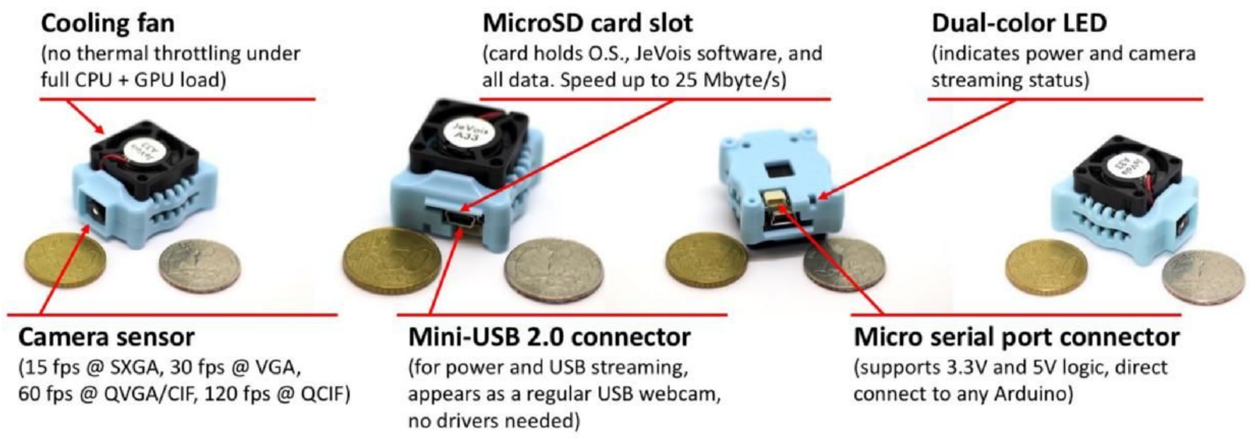
TTL to RS-232 logic level converter.



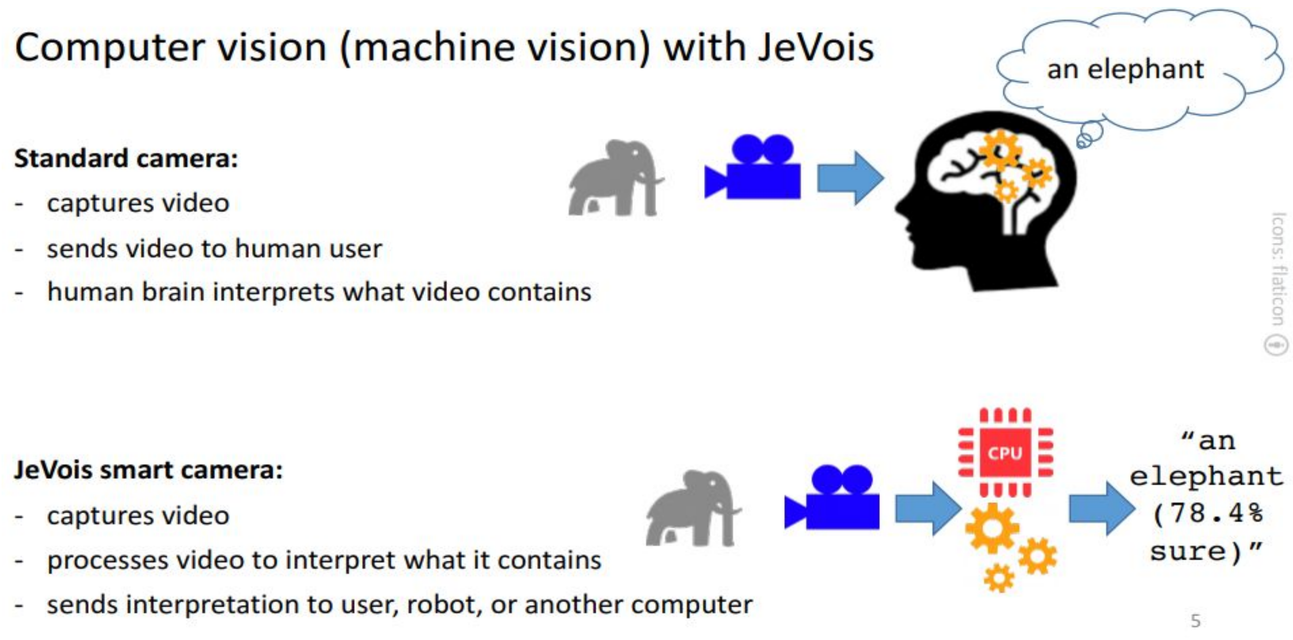
RS-232 port on the roboRio.

We then read the serial data from the camera using LabView code on the roboRio, and send drive commands to the robot accordingly.

This was our first year successfully implementing vision processing on our robot, and we are very grateful for all of the teams who helped us in achieving this goal. Namely, we would like to thank teams 4003, Trisonics, and 2073, EagleForce, as well as Laurent from JeVois Inc., for all of their assistance. Provided are some more visual aids depicting both the main components and subsystems of the JeVois camera, as well as our finished custom camera case.


Cooling fan
(no thermal throttling under full CPU + GPU load)
MicroSD card slot
(card holds O.S., JeVois software, and all data. Speed up to 25 Mbyte/s)
Dual-color LED
(indicates power and camera streaming status)
Camera sensor
(15 fps @ SXGA, 30 fps @ VGA, 60 fps @ QVGA/CIF, 120 fps @ QCIF)
Mini-USB 2.0 connector
(for power and USB streaming, appears as a regular USB webcam, no drivers needed)
Micro serial port connector
(supports 3.3V and 5V logic, direct connect to any Arduino)

A brief description of the key components of the JeVois camera.



Computer vision (machine vision) with JeVois

Standard camera:

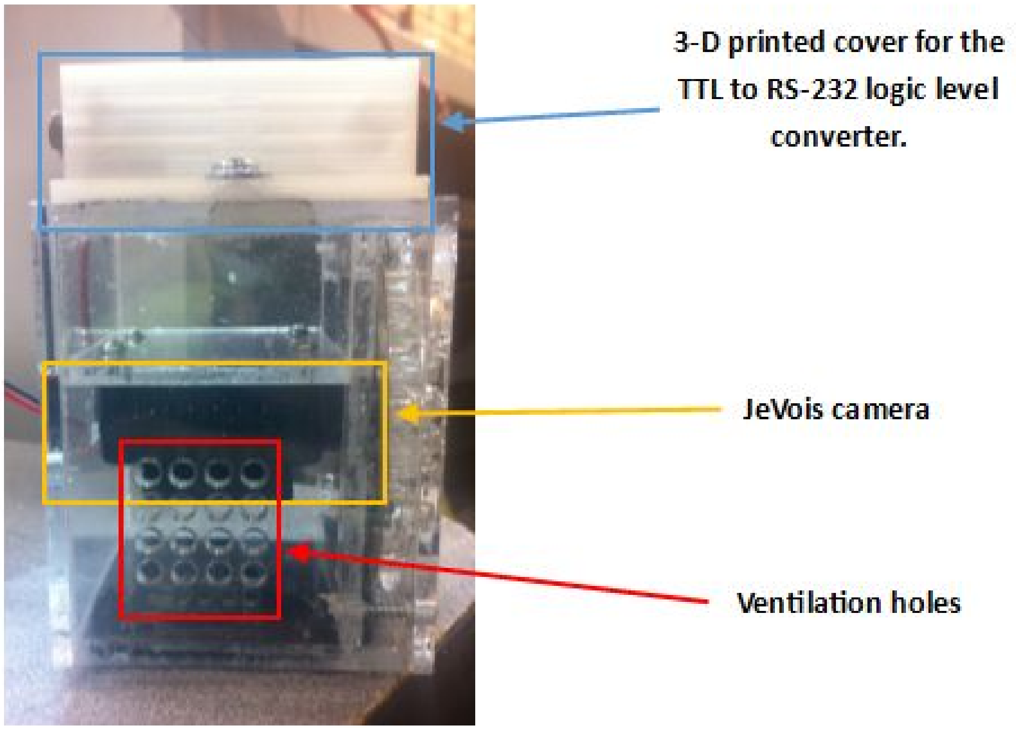
- captures video
- sends video to human user
- human brain interprets what video contains

JeVois smart camera:

- captures video
- processes video to interpret what it contains
- sends interpretation to user, robot, or another computer

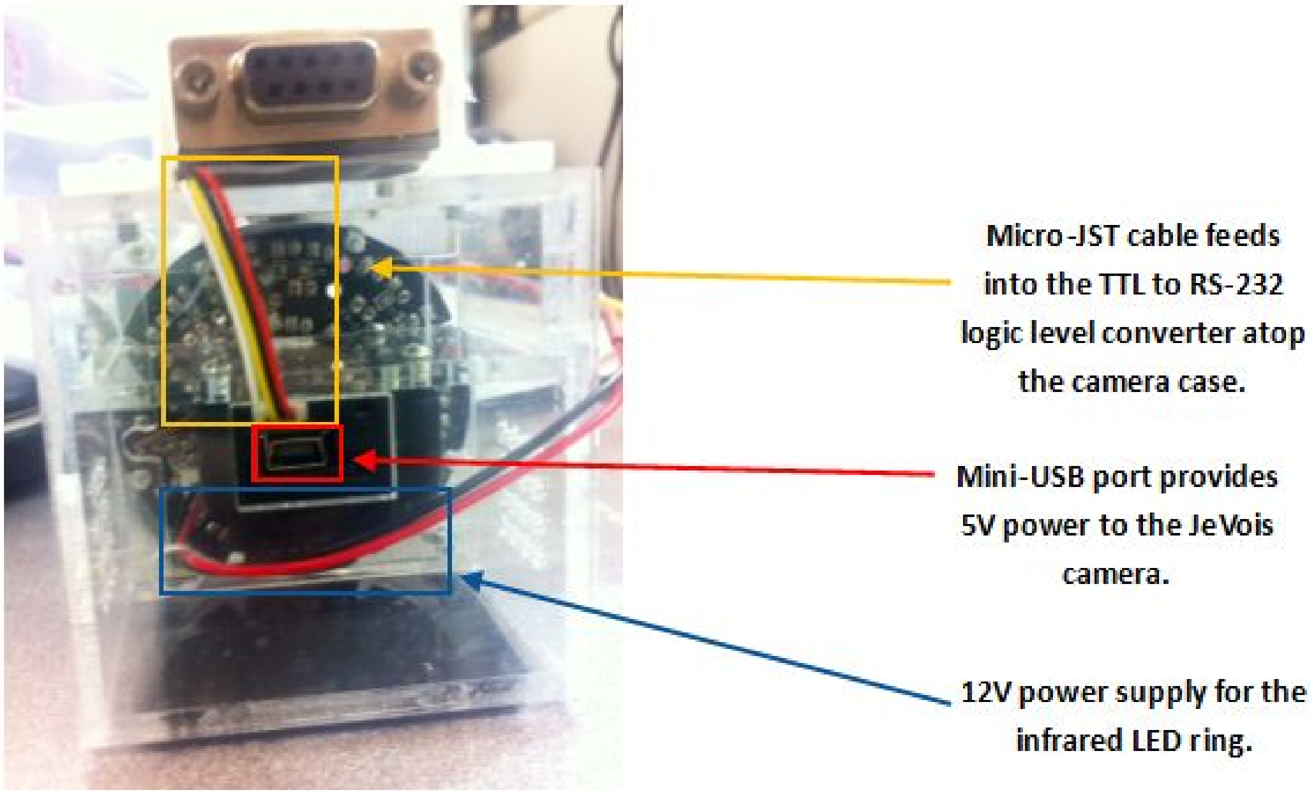
an elephant
CPU
“an elephant (78.4% sure)”
Icons: flaticon
5

Infrared LED ring used for illuminating the vision target.


3-D printed cover for the TTL to RS-232 logic level converter.
JeVois camera
Ventilation holes

Side view of the finished camera case. The front of the case is facing to the right in this image.

Front view of the finished camera case. The camera lens looks through the hole in the center of the LED ring.


Micro-JST cable feeds into the TTL to RS-232 logic level converter atop the camera case.
Mini-USB port provides 5V power to the JeVois camera.
12V power supply for the infrared LED ring.