Preface

To date, we’ve gone through several iterations of the target acquisition code/automation code for this project. Major concepts have stayed the same, but the code has been written in C++ and Python, for both Linux and Windows platforms. These changes were made for a variety of reasons but the main driving force behind our final change to the code base was a client request. Dr. Enderle requested that we use a Windows environment and to write the code with maintainability in mind.

To facilitate maintainability, I rewrote the code in Python due to its design philosophy which emphasizes code readability. The Python bindings for OpenCV are well documented at [http://opencv.willowgarage.com/documentation/python/index.html](http://opencv.willowgarage.com/documentation/python/index.html), and easy to follow. In addition to these benefits, both Python and OpenCV are well supported in Windows. This will help anyone who wishes to update the code base in the future. I hope that these benefits, in conjunction with the user guide to follow, will make updates/maintaining easy.

Pyserial & Arduino

Pyserial is a Python library which provides support for serial connections. With this library, we are able to establish a serial connection to our Arduino UNO which in turn, allows us to control our MigaOne motors.

To accomplish this, the Arduino code was written to have the serial port listening for incoming information from the Pyserial connection. In this case, the target acquisition code is sending position to the Arduino, so that it may control the movement. This can be broken down into 2 phases; write and read.

In figure 1, I’ve shown a quick python implementation of the Pyserial library. `writeSerial()` is a python function which allows you to do serial out, or send coordinate values to the Arduino listening on the other end.
When data from the Python code is sent over the serial port, it is stored in variable `pt`. Variable `pt` may then be used to do further calculations as needed.

Key Functions of Targeting

Next I wanted to outline the major functions that allow us to target objects in the code. Documenting this will help clear up anything that may be hard to understand from the comments within the code itself.

`cvCreateCameraCapture()`

Parameter(s):

- **Index (int)**

  This function initializes capturing a video stream from a camera. It takes one parameter, and integer, which specifies the camera you wish to use. If only one camera exists on the computer you’re running the code from, this parameter may be passed as -1. If multiple cameras exist, the index of the camera you wish to use must be specified. From experience, 0, references a laptops integrated webcam. When I had an additional camera connected via USB, I passed the value of 1, or the second camera connected.

  *Note: Maximum capture frame size is 480px by 640px*

`cvCvtColor()`

Parameter(s):

- **Src (CvArr)** – Source Image(8 bit, 16 bit, or single-precision image)
- **Dst (CvArr)** – Destination image, will be of the same data type as src. The number of channels may be different.
- **Code (int)** Color conversion operation, specified by constants [CV_BGR2HSV]
cvInRangeS()

**Parameters(s):**

- **Src (CvArr)** – The first source array (in our case our source image)
- **Lower (CvScalar)** – Lower threshold boundary
- **Upper (CvScalar)** – Upper threshold boundary
- **Dst (CvArr)** – The destination array (our threshold image)

The benefit of this transform may not be abundantly clear, so I’ve included figure 3. By transforming out color space to a “Hexcone” model all of our colors (Hues) lie on one face. This means that the color we want to filter on is simplified from 3 (RGB) values to 1 (The H of the HSV).

This can be explained more thoroughly by looking at the Lower and Upper parameters of the cvInRangeS() function. cvScalar() programmatically speaking, is just a 4-tuple, however when referenced in this context, each value within the 4-tuple corresponds to a value in the color space.

\[
\text{RGB} \rightarrow \text{cvScalar}(0-255,0-255,0-255,0)
\]

\[
\text{HSV} \rightarrow \text{cvScalar}(0-255,0-255,0-255,0)
\]

In the RGB cvScalar model above, the first value corresponds to how much blue is present, the second is the amount of green, and the third is how much red is represented (*Note: the forth value corresponds to the alpha channel, or transparency*).
In the HSV model, the first value (between 0 and 255) represents the **Hue**, or the actual color you want. This saves you the trouble of trying to create the color you want to use by mixing red, green, and blue values together. The second value is the **saturation**, or pale the image is. The third, **value**, is how dark the color is.

The result of the `cvInRangeS()` function is a binary (black and white) image depicting objects whose color falls between our scalar thresholds.

![Camera Feed, HSV Transform, Binary Image](image)

*Figure 4: Filter Walk-Through*

---

**cvMoments()**

**Parameters(s):**

- **Arr (CvArr)** – Image (Binary image, in our case. See `GetMat()` function)
- **Binary (int)** – For images only, if the flag is non-zero, all black(zero) pixels are treated as zeros, all others are treated as 1’s

The `cvMoments()` function absolutely critical to this project, all steps taken so far were to effectively implement this function. `cvMoments` takes a raster image and computes the central moment of the white pixel mass in your binary image.
Hours Worked

<table>
<thead>
<tr>
<th>Hours Worked</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 hours</td>
<td>Researching and writing pyserial code</td>
</tr>
<tr>
<td>6 hours</td>
<td>Reworking code for Windows &amp; Python</td>
</tr>
<tr>
<td>3 hours</td>
<td>Writing Arduino code</td>
</tr>
<tr>
<td>2 hours</td>
<td>Configuring Lab Computer</td>
</tr>
</tbody>
</table>

References

**OpenCV Documentation**
http://opencv.willowgarage.com/documentation/python/index.html

**RGB-HSV Image**