Computer Vision in FRC

Edward Wawrzynek
FRC 2036 Black Knights

Vision in FRC

- Vision software is typically used to identify field/game elements
  - Used for robot navigation
    - During Autonomous
    - During Teleop (driver assist)
- FRC fields usually have retroreflective tape in key locations
- Vision can be done on game pieces (which often have a distinctive color)

Running Vision

We need a camera and a processing unit:



Processing on RoboRio:
- Easy to set up
- Slow processing capabilities
- Low(ish) latency

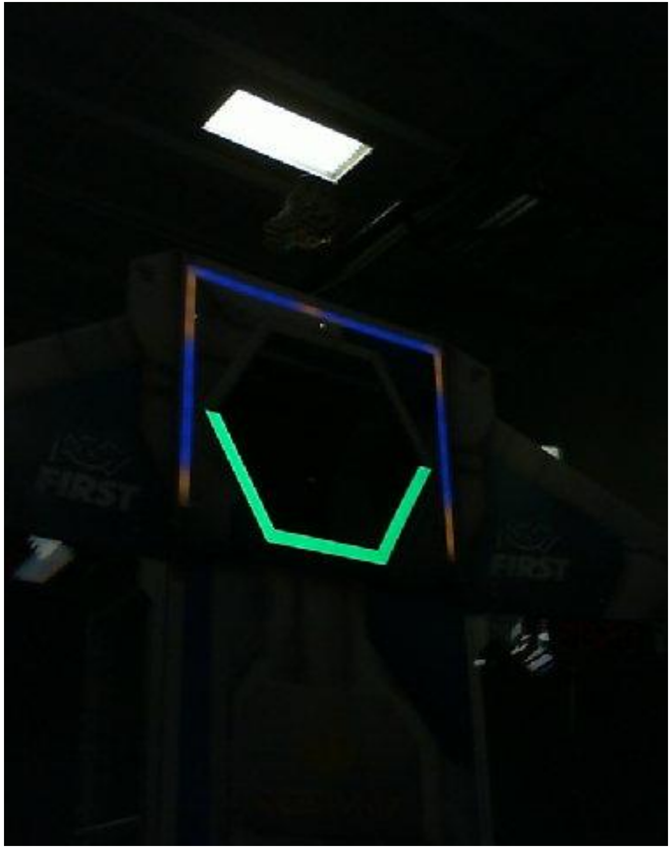
Processing on Driver station:
- Moderate/difficult setup
- Fast processing capabilities
- High latency (wireless transfer)

Dedicated hardware on robot:
- Difficult setup
- Fast(ish) processing capabilities
- Low latency

Retroreflectivity

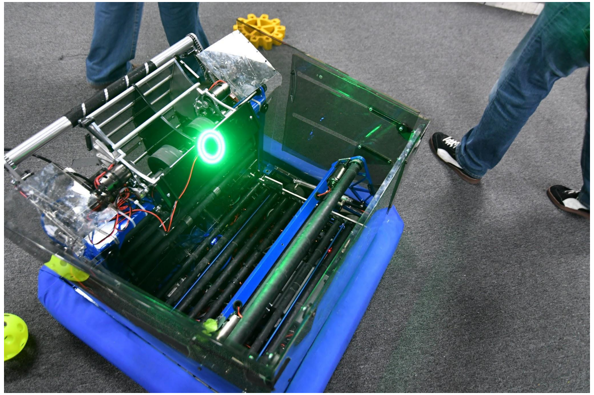
- Retroreflective materials reflect light back at their sources from a wide range of angles
    - If there is a light close to our camera, it can help the robot identify retroreflective targets (targets appear lit up in image)

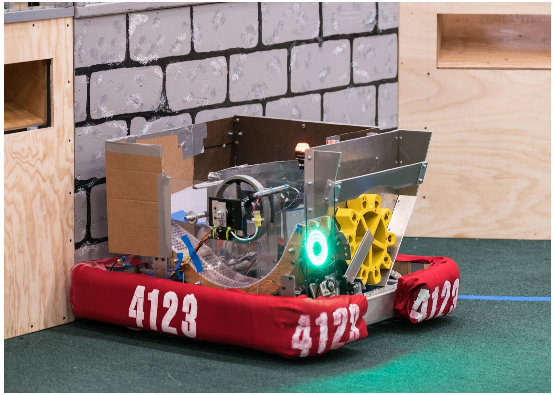
Retroreflectivity



Retroreflectivity

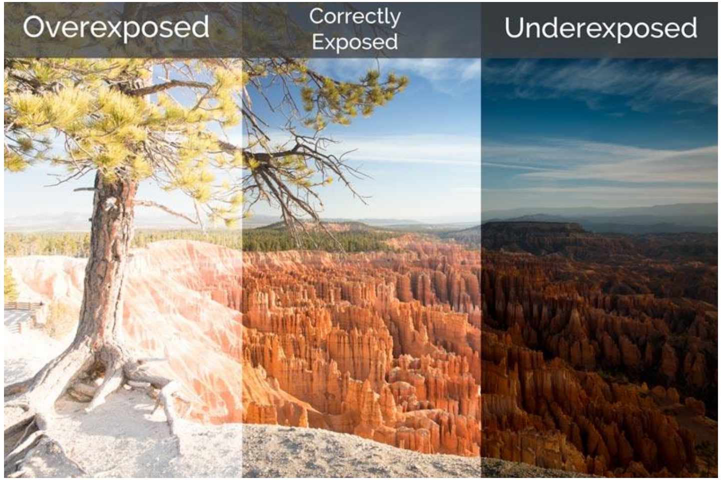
- Typically, bright green leds are placed close to the camera used for vision
    - FRC doesn’t use green lights anywhere on the field on purpose





Camera Exposure

- Exposure is the amount of light that enters the camera during a single frame
- The auto exposure feature in most usb cameras **will not work** for getting usable images for vision
- The exposure must be manually set low
    - We only want to see the target, nothing else



OpenCV

- OpenCV (Open Source Computer Vision) is a library that provides a range of computer vision functions
- C++, with Java bindings (usable in Kotlin)
- 2036 uses OpenCV for vision processing

A Sample Vision Pipeline

Identify the target (high goal) in this image:

1. Reading the Image

- OpenCV represents images as `Mat` objects
- For FRC, we typically want to read images from a webcam
- Images should be resized (for faster processing speed). We use 320x240 here

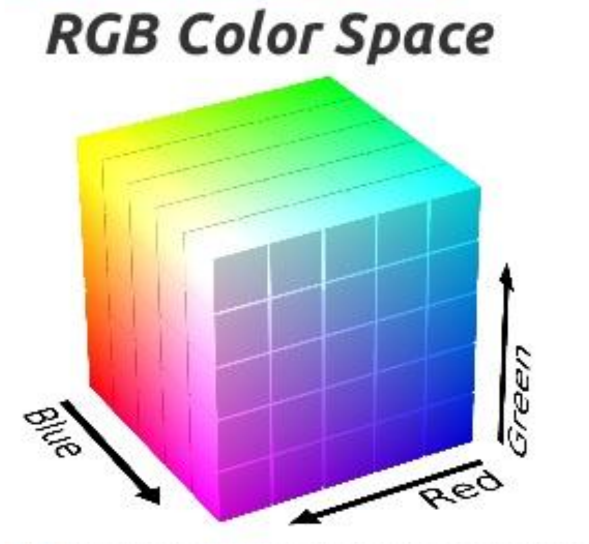
2. Gaussian Blur

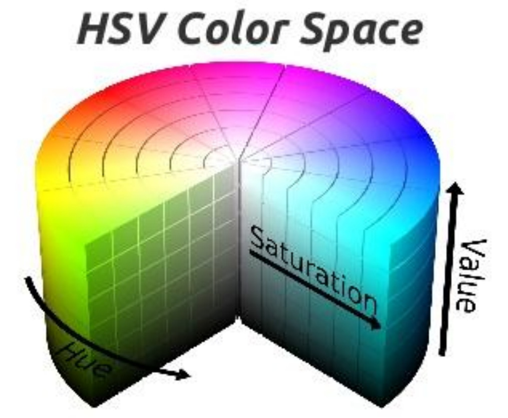
- Blurring an image helps remove noise
- Typical kernel size ~3/5 px
- Tradeoff between lost detail + noise

```
/* blur */
Mat imageBlur = new Mat();
Imgproc.blur(
      imageRs,
      imageBlur,
      new Size(blurRadius, blurRadius)
);
```

Color Spaces: RGB vs HSV

- Images are typically represented in RGB (Red, Green, and Blue components)
    - This isn't the best if we want to find certain colors (like green). (Why?)
- HSV represents colors as a Hue, Saturation (grayness) and Value (brightness)
    - Easier to find certain colors (Why?)

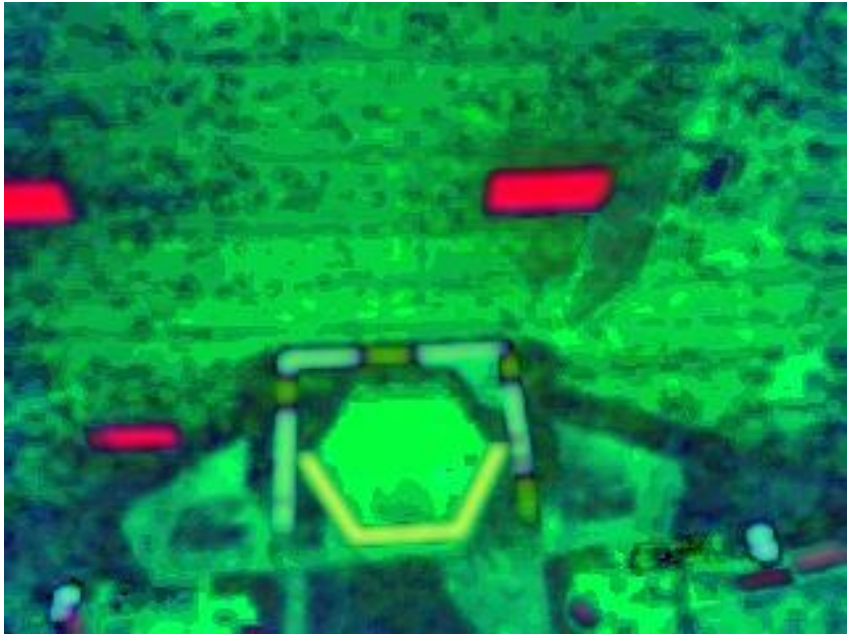




3. RGB -> HSV

```
/* convert to hsv */
Mat imageHsv = new Mat();
Imgproc.cvtColor(
      imageBlur,
      imageHsv,
      Imgproc.COLOR_BGR2HSV
);
```

In the image below, the blue channel is hue, the green saturation, and the red value

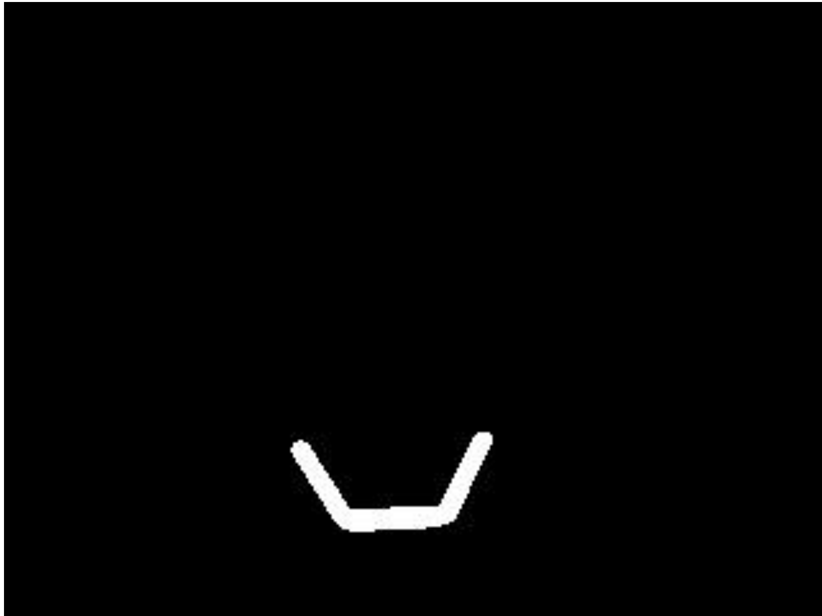


4. HSV Thresholding

- A threshold range is applied to the hsv image, producing a binary image (1/white = in range, 0/black = not in range)
- HSV ranges were:
    - Hue : 53-100 (green)
    - Saturation 213-255 (high saturation only)
    - Value: 100-255 (at least moderately bright)

```
/* mask hsv to specified range */
Mat imageMask = new Mat();
Core.inRange(imageHsv, new Scalar(53, 213, 100), new Scalar(53,
213, 100), imageMask);

/* blur and re-threshold image (further noise elimination) */
Imgproc.blur(imageMask, imageBlur, new Size(reblurRadius,
reblurRadius));
Imgproc.threshold(imageBlur, imageMask, 50, 255,
Imgproc.THRESH_BINARY);
```



5. Contour Detection

- Contours are curves joining all contiguous points on a boundary in an image
    - Contours are most effective on binary images
- Filter contours:
    - Area, width to height ratio, etc

```
/* find contours */
List<MatOfPoint> contours = new ArrayList<>();
Mat contourHierarchy = new Mat();

Imgproc.findContours(imageMask, contours, contourHierarchy,
Imgproc.RETR_LIST, Imgproc.CHAIN_APPROX_TC89_KCOS);

/* sort by size (largest -> smallest) */
Collections.sort(contours, (MatOfPoint cnt1, MatOfPoint cnt2) ->
-Double.compare(Imgproc.contourArea(cnt1),
Imgproc.contourArea(cnt2)));
```

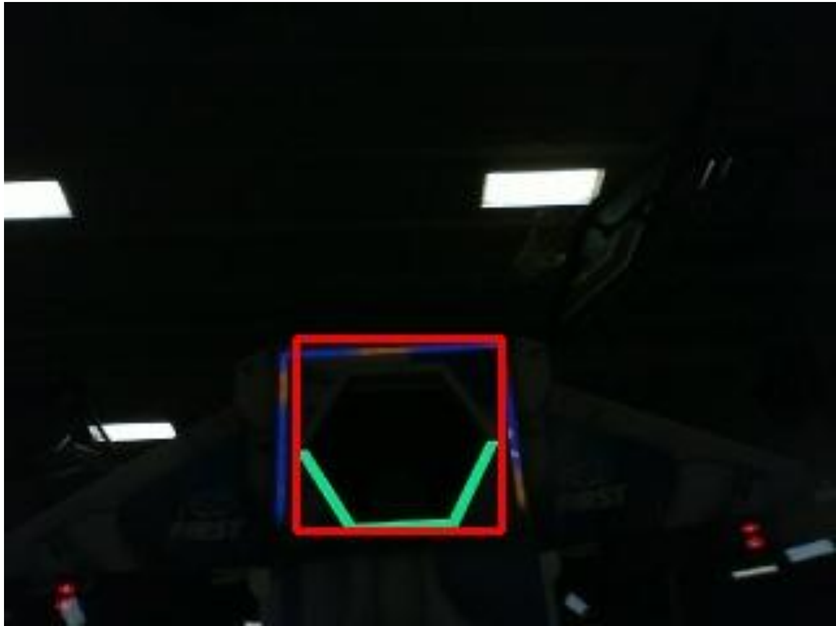
6. Target Identification

- OpenCV provides utilities to find centers of contours, bounding rectangles, ellipsoids, etc
- A bounding rectangle is used here

```
/* find bounding rect on target */
Rect boundRect = Imgproc.boundingRect(countour);
/* because vision target is just on the lower half of the
real target, adjust box to include top half */
boundRect.y -= boundRect.height;
boundRect.height *= 1.9;

/* find center of target */
long centerX = boundRect.x + boundRect.width / 2;
long centerY = boundRect.y + boundRect.height / 2;

/* calculate yaw + pitch offset from center (in radians) */
double yaw = calcAngle(centerX, (long)resize.width/2,
hFocalLen);
double pitch = calcAngle(centerY, (long)resize.height/2,
vFocalLen);
```



What Now?

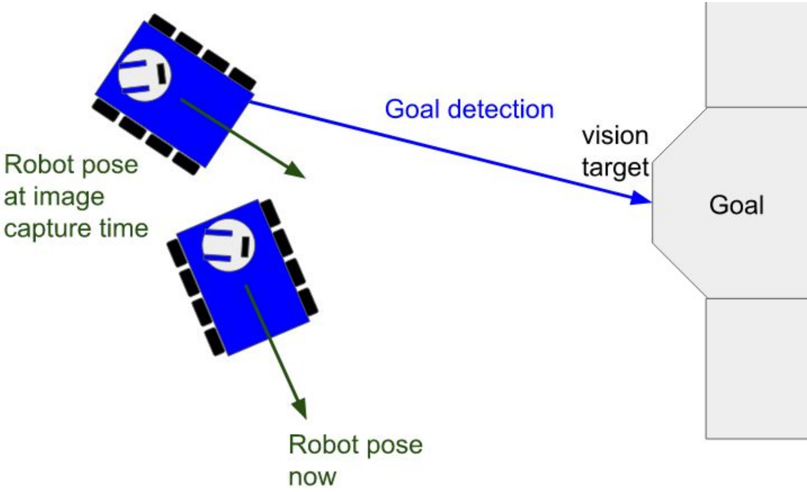
- Conversion to angle:

$$angle = \tan^{-1}\left(\frac{pos - center}{focalLen}\right)$$

- Doing something with the target:
    - Turning towards it
    - Driving at it
    - Etc
    - How?

Compensating for Latency

- Image processing takes time
    - By the time we know where the goal is, we have probably moved
- Don’t relying solely on vision to actually drive the robot
    - Use vision to figure out where the target is, and other control to actually get there
    - Eg: get an angle from vision, run a feedback loop with a gyro to turn to it



Other Vision Targets

- Retroreflective targets are the easiest, but vision can pick out other features
- Lighting conditions:
    - Easier to deal with on retro reflective targets (Why?)
    - Much harder for others: target might be in shadow, reflecting light, etc
- OpenCV can help — finding shapes regardless of lighting, edge detection, etc
- Combined computer vision + machine learning approaches