



EMGU CV

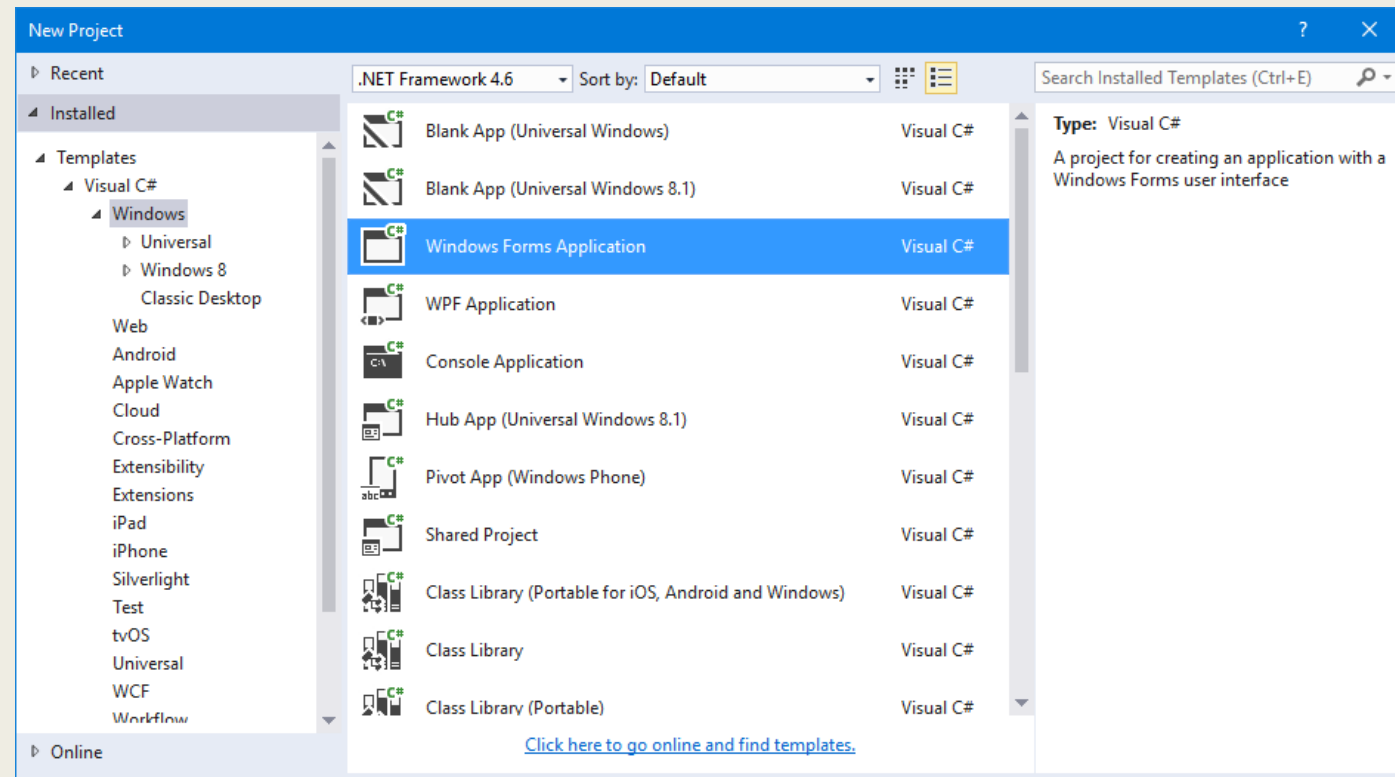
Prof. Gordon Stein
Spring 2018

Lawrence Technological University
Computer Science
Robofest



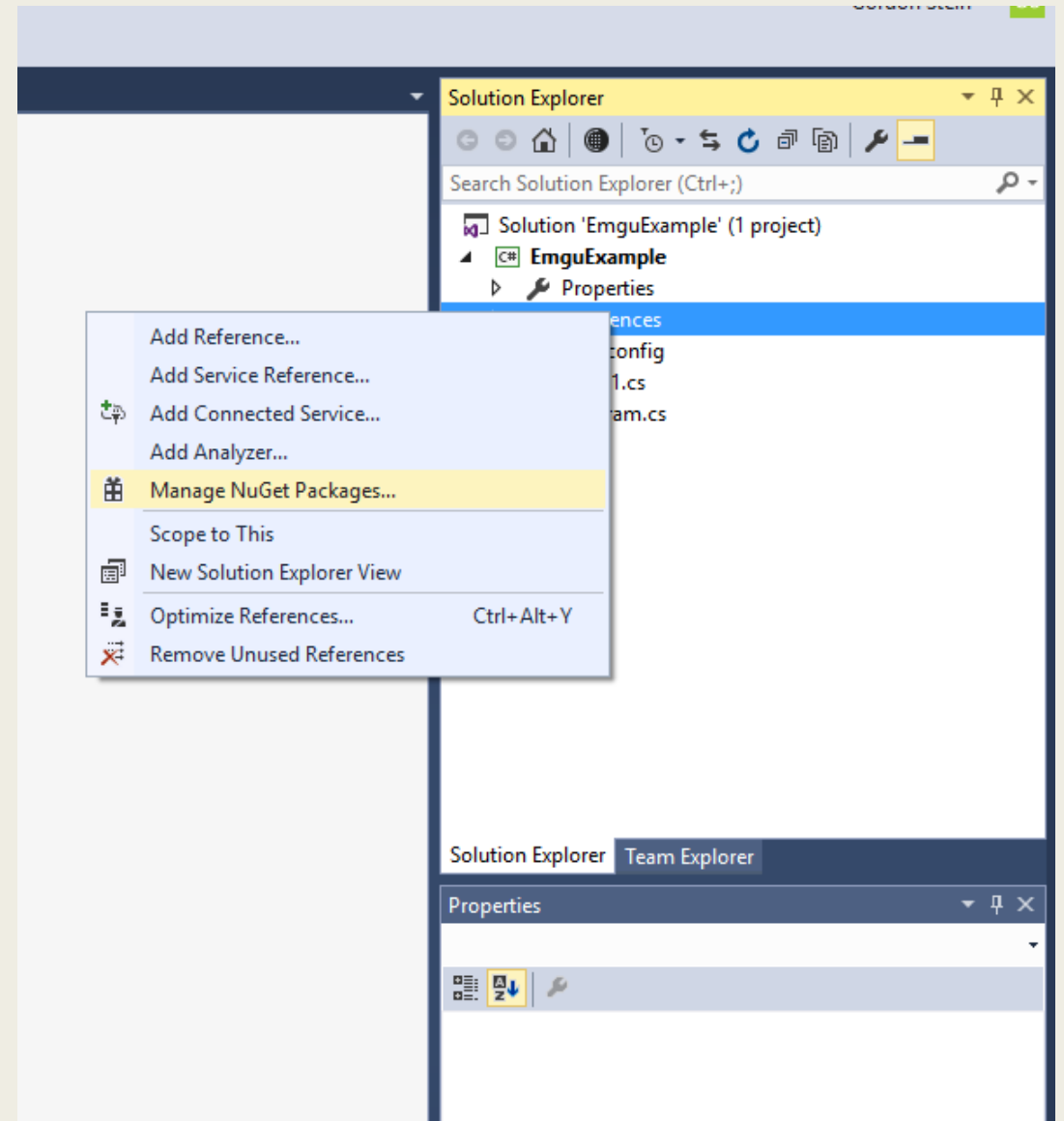
Creating the Project

- In Visual Studio, create a new Windows Forms Application
- (Emgu works with WPF and Universal as well, but this tutorial will use WinForms)



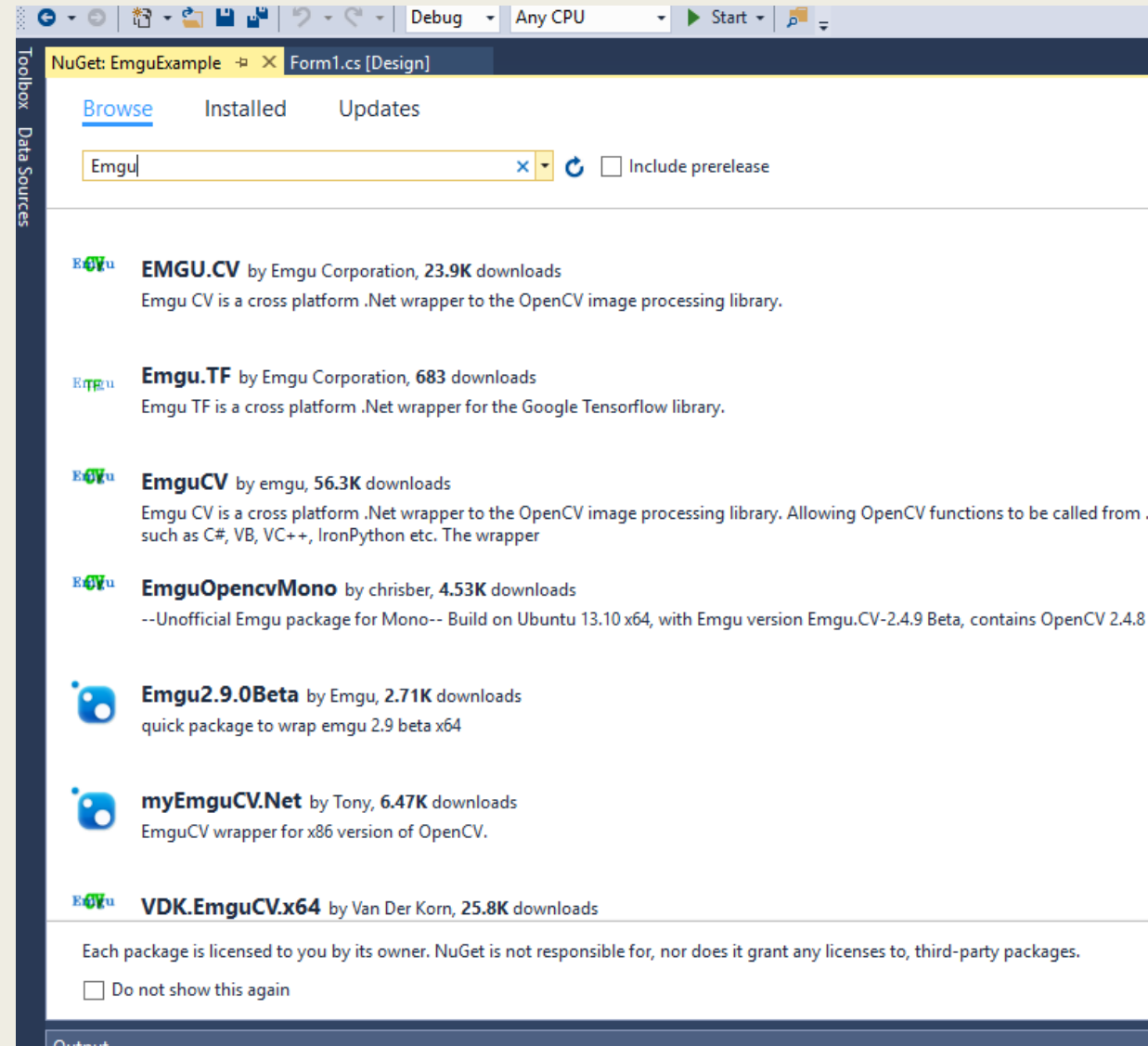
Adding Emgu

- In the Solution Explorer, right click on References and choose “Manage NuGet Packages...”
- If this option is not available, NuGet will need to be installed with the Visual Studio installer



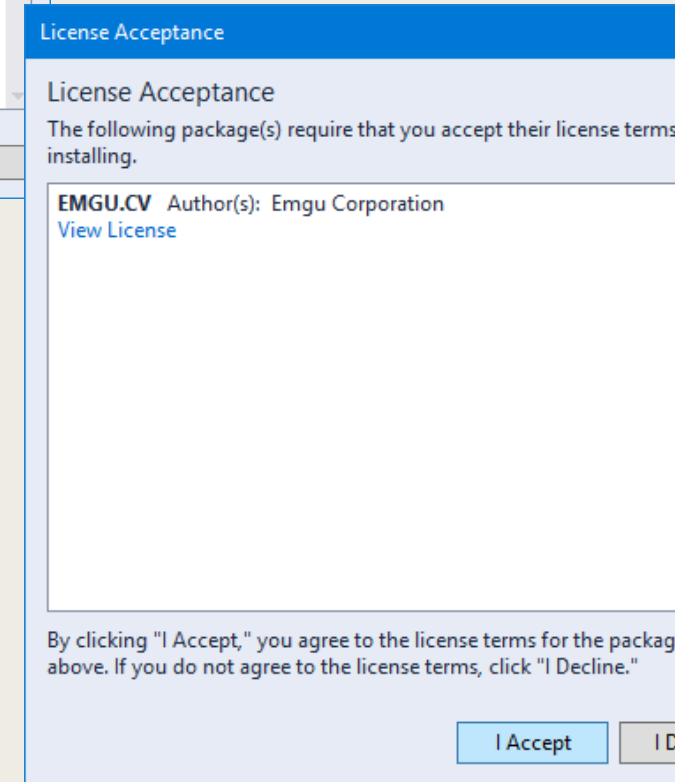
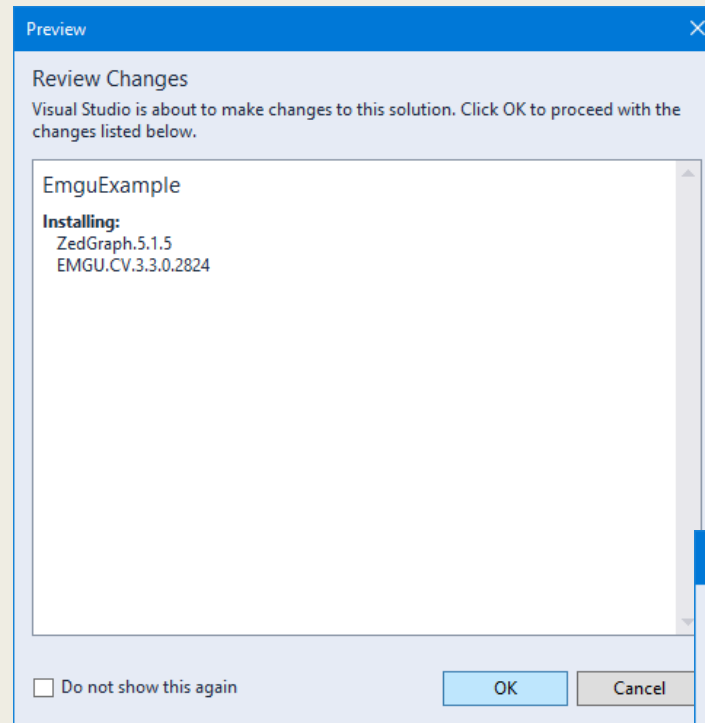
Adding Emgu

- Click “Browse” and type “Emgu” into the search bar
- Click on “EMGU.CV” and then Click the “Install” button



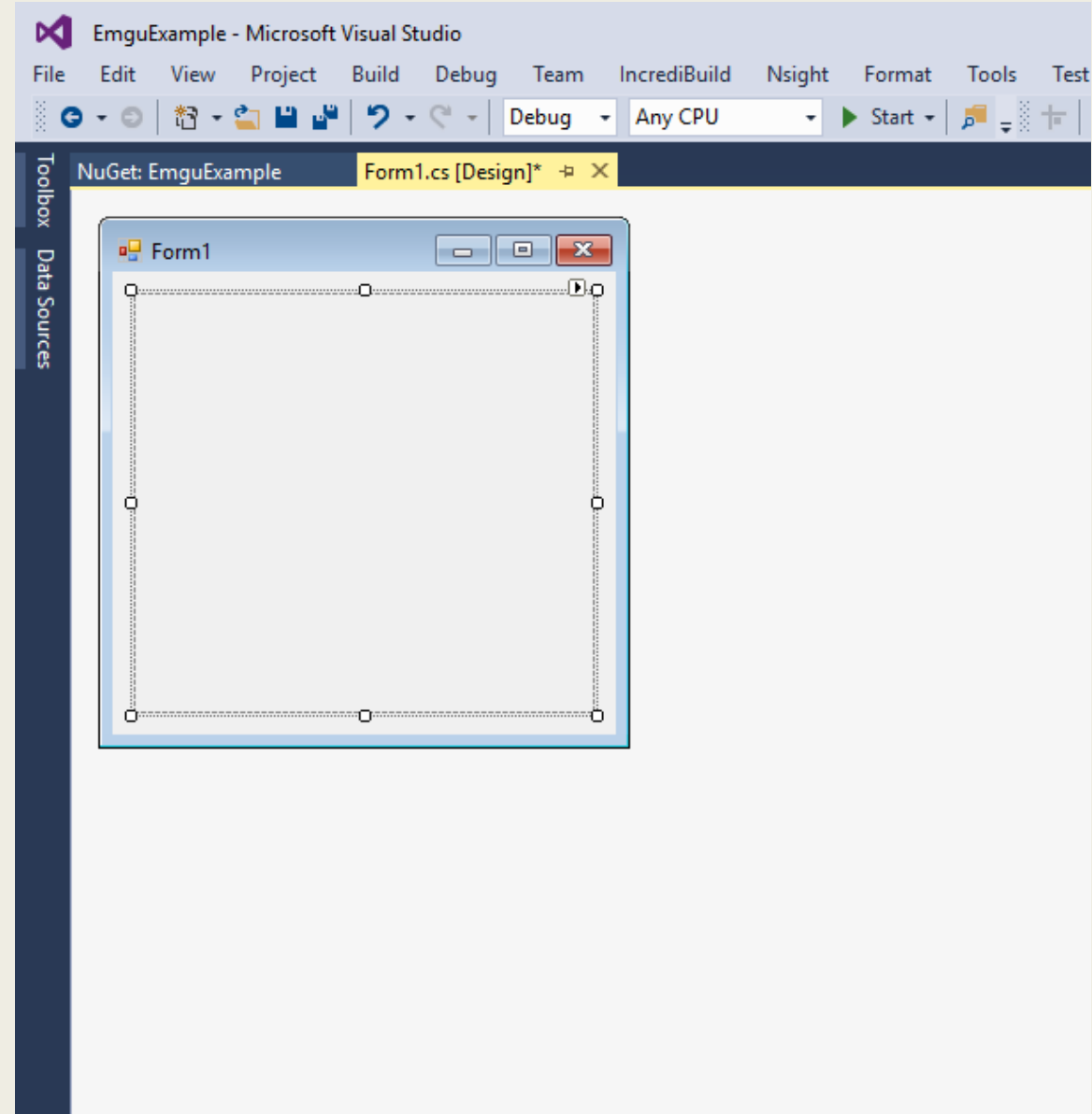
Adding Emgu

- Two windows should pop up after choosing to install it
- Click OK in the “Review Changes” window and “I Accept” in the “License Acceptance” window
- Emgu is now installed!



Testing Emgu

- In the Form (GUI) that was automatically created with the project, add a PictureBox
- For a real project, it is recommended to give your GUI components names, but to simplify the example we will leave the name the default “pictureBox1”



Testing Emgu

- Double click on the Form's background to create a Load event for the Form
- In this example, we will be connecting to the webcam and displaying its image on the PictureBox

Testing Emgu

- In the class itself (not inside any methods) we need to add two variables
- The first is the VideoCapture to get the stream from the webcam
- The second is a Thread we'll use to update the image
- We need to use a thread so we can process images in the background to avoid freezing up the GUI

```
private VideoCapture _capture;  
private Thread _captureThread;
```


Testing Emgu

- In the Form1_Load method, add the following code
- The first line creates the capture with the default camera
- The VideoCapture constructor can take a parameter to change the camera if it is not using the correct one
- The other two lines set up and start a thread running a function we will create next

```
_capture = new VideoCapture();  
_captureThread = new Thread(DisplayWebcam);  
_captureThread.Start();
```

Testing Emgu

- Finally, we must create the DisplayWebcam method:

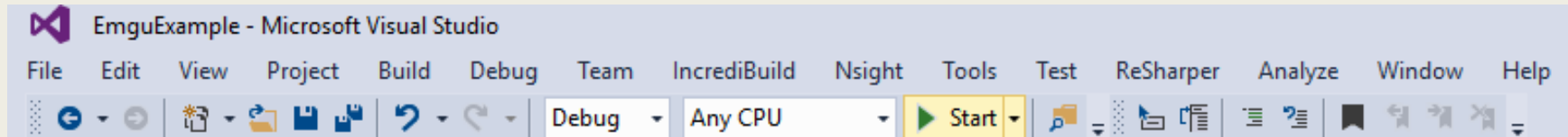
```
private void DisplayWebcam()
{
    while (_capture.IsOpened)
    {
        Mat frame = _capture.QueryFrame();
        CvInvoke.Resize(frame, frame, pictureBox1.Size);
        pictureBox1.Image = frame.Bitmap;
    }
}
```

Testing Emgu

- This code will run in a loop as long as the webcam is being captured
- Each iteration it requests a frame from the camera, resizes it to the Size of the PictureBox (storing the resized version in the same variable), and gives the image to the PictureBox to display

Testing Emgu

- If you run this code, it should turn on your webcam and display its image in the PictureBox



- Congratulations, you are now ready to start working with computer vision!

One More Thing

- If you close the window for this program, the webcam doesn't turn off automatically! (Stopping it through Visual Studio or Task Manager will shut it off)
- This is because the Thread is still running in the background
- To stop it when we close the window, we need to add an event handler to the Form's FormClosing/FormClosed event (either one) with the following code to end the Thread:

```
_captureThread.Abort();
```

Storing an Image

- Images can be stored in two different types:
 - *The Mat type*
 - *The Image type*
- While it seems a little counterintuitive, the Mat type is the newer one
 - *Converting between the two is simple though*
 - *We'll end up using the Image type anyways*

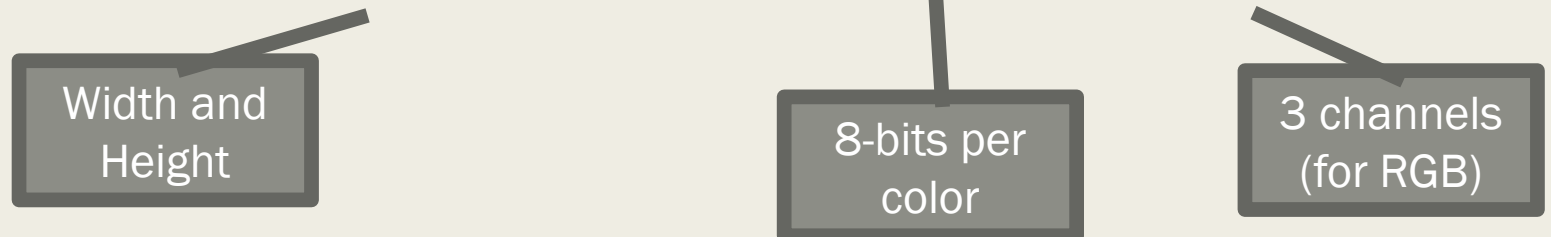
Creating a Mat

- A Mat that will store the output from something else:

```
Mat mat = new Mat();
```

- A blank Mat with a known size and color type:

```
Mat mat = new Mat(200, 400, DepthType.Cv8U, 3);
```



- From a file:

```
Mat mat = CvInvoke.Imread("img.jpg", ImreadModes.AnyColor);
```

Converting Between Mat and Image

- Mat to Image:

```
Image<Bgr, Byte> img = mat.ToImage<Bgr, Byte>();
```

- Image to Mat:

```
Mat mat2 = img.Mat;
```


Color Spaces

- You may know about RGB
 - *Red, Green, Blue*
- That Image example used BGR
- This just changes the order the colors are stored in

Color Spaces

- Emgu also supports:
 - *Grayscale (no color)*
 - *BGRA (BGR with alpha for transparency)*
 - *HLS (Hue, Lightness, Saturation)*
 - *HSV (Hue, Saturation, Value)*
 - *Lab, Luv, XYZ (Perceptual color spaces)*
 - *YCbCr (Designed to be more efficient)*

Color Spaces

- The number of bits per color is not as important
- 8-bit is enough for each color to get “true color”
 - *Our displays and cameras probably won't show any more than that*

Converting to Grayscale

- With an Image:

```
Image<Gray, Byte> img2 = img.Convert<Gray, Byte>();
```

- Going from Mat to Image:

```
Image<Gray, Byte> img = mat.ToImage<Gray, Byte>();
```

Accessing Pixels

- With an Image, we can use it like an array to get the color at a specific location
- Works for read and write:

```
Gray pixel = img[1, 5];
```

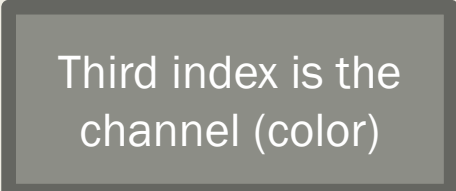
```
img[1, 5] = new Gray(255);
```

Accessing Pixels

- That method can be slow
- Directly accessing the data of the image is faster:

```
byte pixel = img.Data[1, 5, 0];
```

```
img.Data[1, 5, 0] = 255;
```



Third index is the
channel (color)

A related trick

- You can get one channel by using an image with multiple channels (BGR, HSV, etc) like it's a 1-D array
- You get a grayscale image out of that:

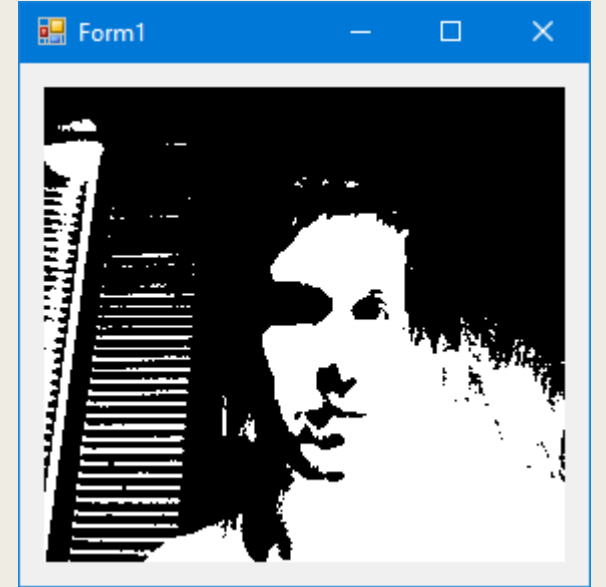
```
Image<Gray, Byte> blues = imgBgr[0];
```

Thresholding

- A binary threshold operation will take all pixels above a certain value and make them the same value, and everything else set to 0 (black)

```
img = img.ThresholdBinary(new Gray(100), new Gray(255));
```

- There are other types as well



Counting White Pixels

- Two ways to do it
- The first way would be to loop through the image:

```
int whiteCount = 0;
for (int x = 0; x < img.Width; x++)
{
    for (int y = 0; y < img.Height; y++)
    {
        if (img.Data[y,x,0] == 255)
            whiteCount++;
    }
}
```

Counting White Pixels

- There's an easier way (although you may find that you need to loop through the image for some tasks)
- Assuming a thresholded image:

```
img.CountNonzero()[0]
```

Modifying Labels

- If you just try to change a Label's text (and some other properties) from a Thread, it'll throw an exception!
- To fix this, you need to use the Invoke method to interact with some parts of the GUI

```
Invoke(new Action(() => label1.Text = "Hello"));
```



Your code here

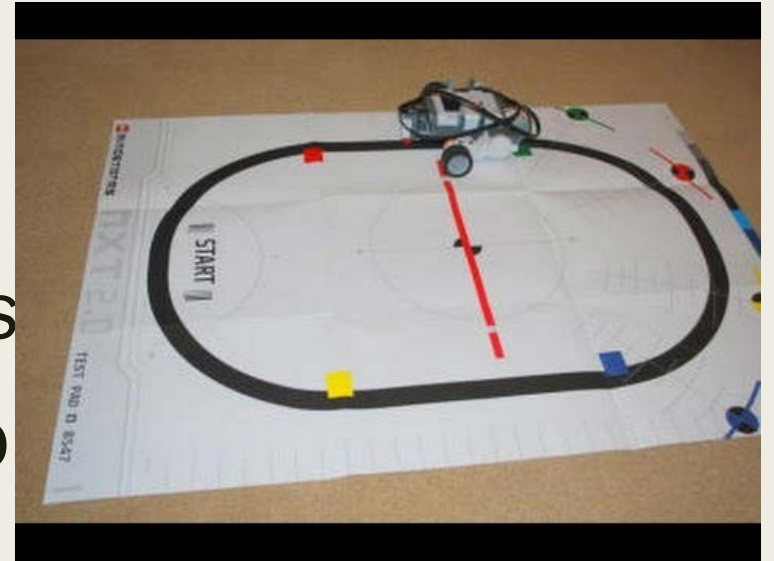
Modifying Labels

- Or for multiple lines at once:

```
Invoke(new Action(() =>
{
    label1.Text = "Hello";
    label2.Text = "World";
}));
```

What is Line Following?

- Path on ground
 - *Path may have gaps or branches*
- Robot follows it to the end/in a loop



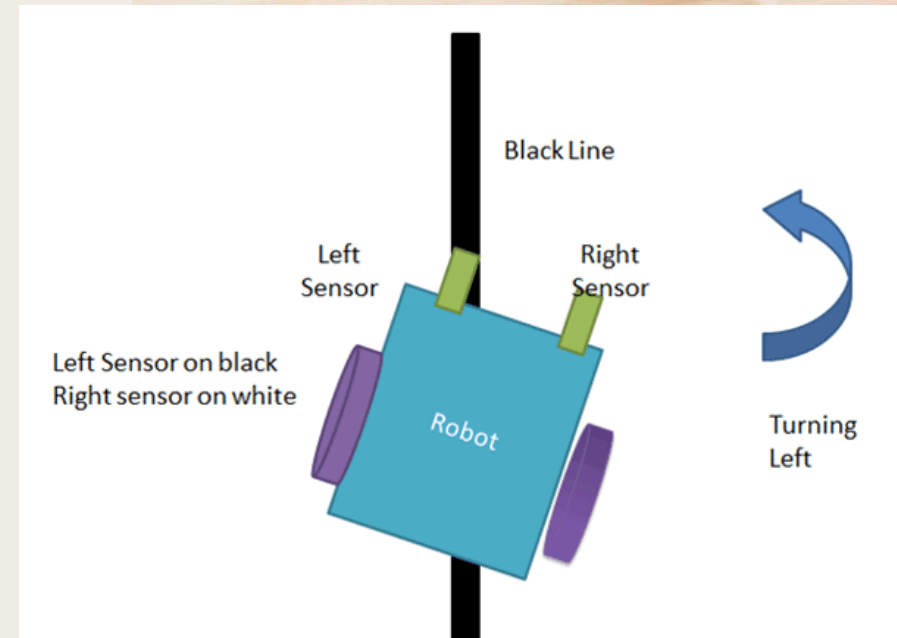
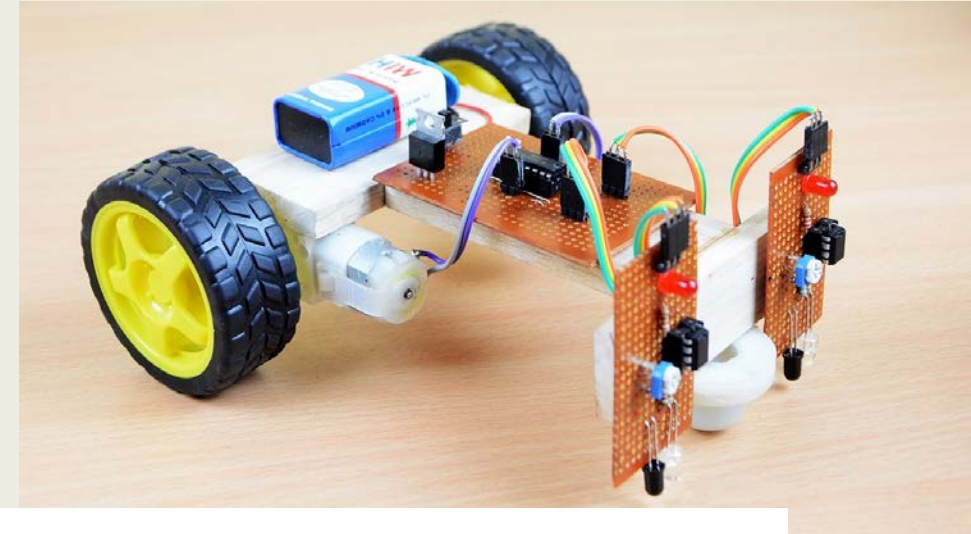
Line Following Without Vision

- With one sensor, the robot can aim to keep the sensor on one side of the line
- If the sensor is not on the line, turn left/right, if it is on the line, turn the opposite way
 - *Zigzags along the line*
- A PID controller can improve it



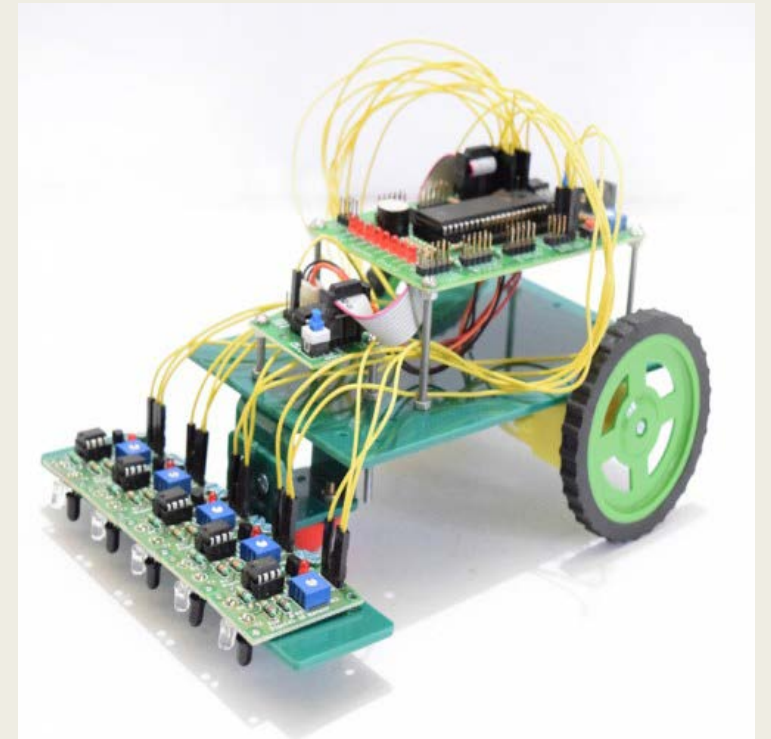
Line Following Without Vision

- IR sensors pick up the dark/light regions on the ground
- With two sensors, turn left if line is under left sensor, turn right if line is under right sensor



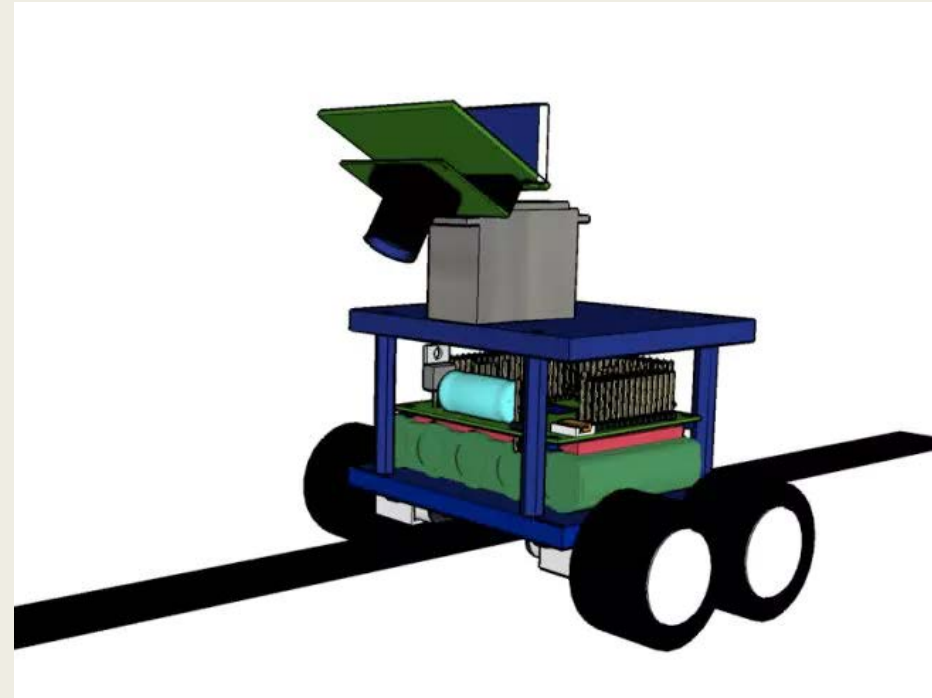
Line Following Without Vision

- With more sensors, the robot can better judge how far off from the line it is
- All of these approaches will have a few natural limitations:
 - *Limited resolution*
 - *Can't see beyond the front of the robot*
 - *Doesn't handle branches or gaps well*



Line Following With Vision

- A vision based approach has advantages:
 - *Can see ahead (past gaps and branches)*
 - *Better resolution (know exactly how far to turn)*
- Disadvantages:
 - *Requires more powerful processor*



Line Following With Vision

- Simple approach is very similar to the two sensor design:
 - *Apply threshold to find line*
 - *Look at thirds of image, determine if left/right one has the most white pixels*
 - *Turn in that direction*
- Good for robots like our L2Bots that can't steer precisely

Code Example (finding left third)

```
int leftWhiteCount = 0;
for (int x = 0; x < img.Width / 3; x++)
{
    for (int y = 0; y < img.Height; y++)
    {
        if (img.Data[y,x,0] == 255)
            leftWhiteCount++;
    }
}
```

Code Example (Alternative method)

```
img.ROI = new Rectangle(0, 0, img.Width / 3, img.Height);  
leftWhiteCount = img.CountNonzero()[0];  
img.ROI = Rectangle.Empty; // Reset ROI
```

```
img.ROI = new Rectangle(img.Width / 3, 0, img.Width / 3, img.Height);  
centerWhiteCount = img.CountNonzero()[0];  
img.ROI = Rectangle.Empty; // Reset ROI
```

```
img.ROI = new Rectangle(2 * img.Width / 3, 0, img.Width / 3, img.Height);  
rightWhiteCount = img.CountNonzero()[0];  
img.ROI = Rectangle.Empty; // Reset ROI
```

We're setting a "Region of Interest", which tells Emgu we only want it to consider a portion of the image.

Line Following With Vision

- Another approach:
 - *Apply Thresholding*
 - *Find average position of white pixels*
 - *Steer in that direction*
- Good for robots that can give a precise turning speed
- Used in CS IGVC robot demo at LTU

Line Following With Vision

- More complex approach:
 - *Apply Thresholding*
 - *Detect Lines/Curves in Image*
 - *Use angle of line from vertical to steer*

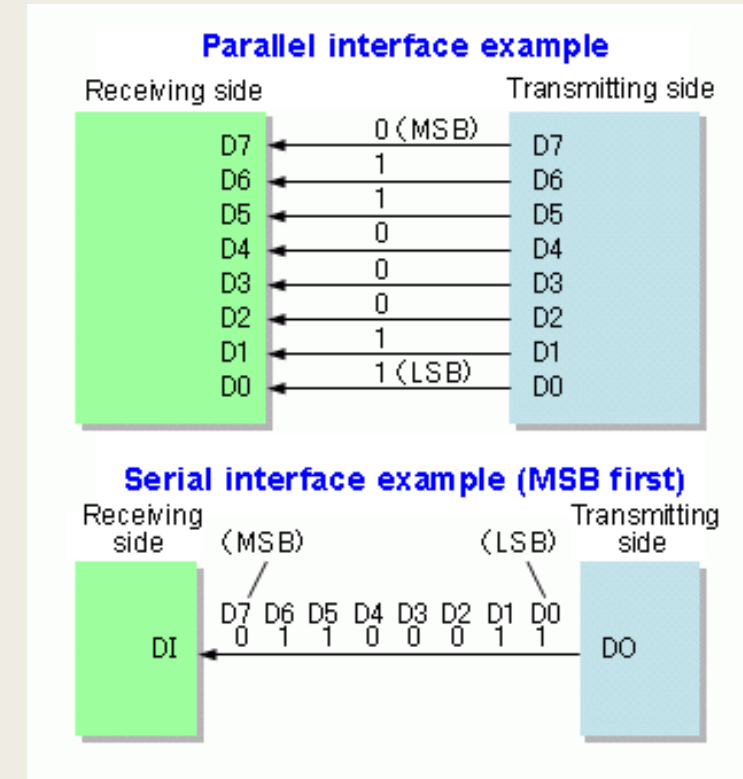
Threshold Values

- You've been using a value set by the user for the threshold
- If the lighting changes, you may need to change it
- There are other ways you can try
- For example, you know it will always be brighter than the average pixel, so that value could be your threshold

```
img.GetAverage()
```

Serial Communication

- Send bits one at a time over a wire
- Useful for communicating with many simple devices
- Easy to implement on a microcontroller



SerialPort Class

- In C#, we use the SerialPort Class to represent a port on the computer used for Serial communications, and Send/Receive using it

Setting up a SerialPort

```
SerialPort _serialPort = new SerialPort("COM4", 2400);
```

Serial port to connect to
(will likely be different)

Baud Rate (must be
compatible with other side)

```
_serialPort.DataBits = 8;  
_serialPort.Parity = Parity.None;  
_serialPort.StopBits = StopBits.Two;  
_serialPort.Open();
```

Settings for port (these
will be what we use)

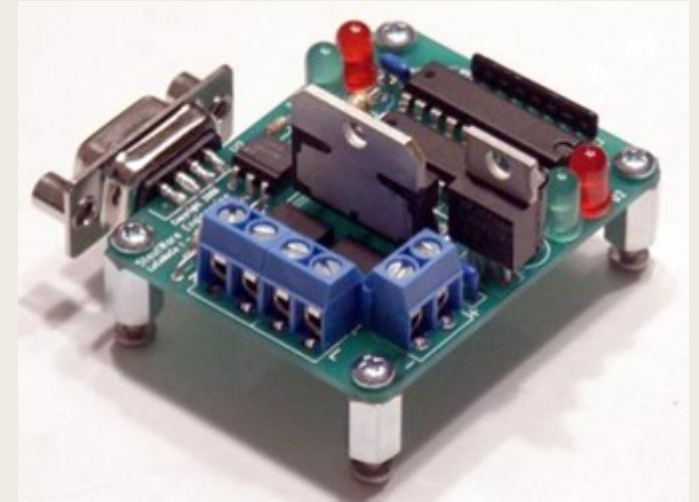
Get port ready for
communication

Communicating over a SerialPort

- The Write method sends an array of bytes over the port
- The Read method (we won't use it in this class) gets input and puts it into an array
- We also can Close the port, which we'll need to do like how we had to Abort the Thread if we want our program to end when we close the window

LoCoMoCo

- The motor controller we'll be using
- The LOW COst MOtor COntroller
- H-Bridge can make two motors go in two directions
- Communicates over serial



LoCoMoCo Commands

- There are four commands that can be sent to each motor:

```
const byte STOP = 0x7F;  
const byte FLOAT = 0x0F;  
const byte FORWARD = 0x6f;  
const byte BACKWARD = 0x5F;
```

LoCoMoCo Commands

- The commands for each motor are put into an array and sent to the controller

```
byte left = FORWARD;
```

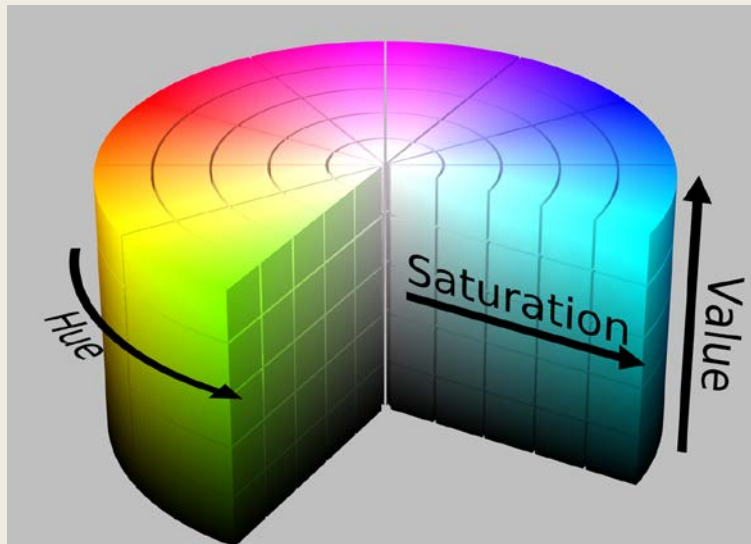
```
byte right = BACKWARD;
```

```
byte[] buffer = { 0x01, left, right };
```

```
_serialPort.Write(buffer, 0, 3);
```

Color Detection

- Easiest way to detect colors is an HSV image
- HSV:
 - *Hue: What color? (0 to 180)*
 - *Saturation: How intense is the color? (0 to 255)*
 - *Value: How bright is the color? (0 to 255)*



Converting to HSV

- It just needs the color type to be specified as Hsv:

```
Image<Hsv, byte> hsvImage = img.Convert<Hsv, byte>();
```


Counting Pixels of a Color

- With an HSV image, we can use the InRange method to look for pixels where each component is in a specific range
- To look for red pixels (with at least 150 in saturation and 100 in value):

```
int redpixels = hsvImage.InRange(new Hsv(0, 150, 100),  
                                new Hsv(25, 255, 255)).CountNonzero()[0];
```

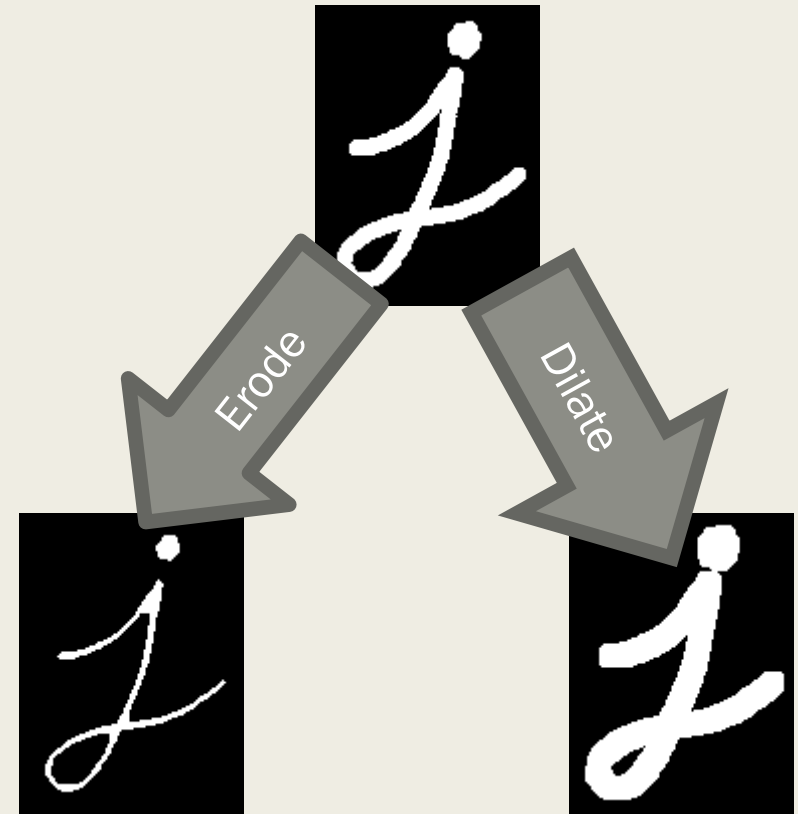
Note that finding red pixels may require searching values around 180 as well

Noise in images

- Sometimes a threshold alone does not work
- You'll divide the image up correctly, but something in the background is ruining the image

Dilation and Erosion

- Two actions that act on the pixels in the image using a “kernel” (a set of neighboring pixels and values for them)
- Each one expands white or black regions:
 - *Dilate: if one of the neighbors is a white pixel, this pixel is white*
 - *Erode: if one of the neighbors is a black pixel, the pixel is black*



```
img = img.Erode(3);    img = img.Dilate(3);
```

Open and Close

- There are two combinations of dilations and erosions
- Opening: an erosion and then a dilation, useful for removing background noise
- Closing: a dilation followed by an erosion, fills in holes in

```
img = img.Erode(1).Dilate(1);
```



```
img = img.Dilate(1).Erode(1);
```

Blurring images

- We have multiple types of blur/smoothing in EmguCV:
 - *Average (Mean)*
 - *Bilateral*
 - *Gaussian*
 - *Median*

Gaussian Blur

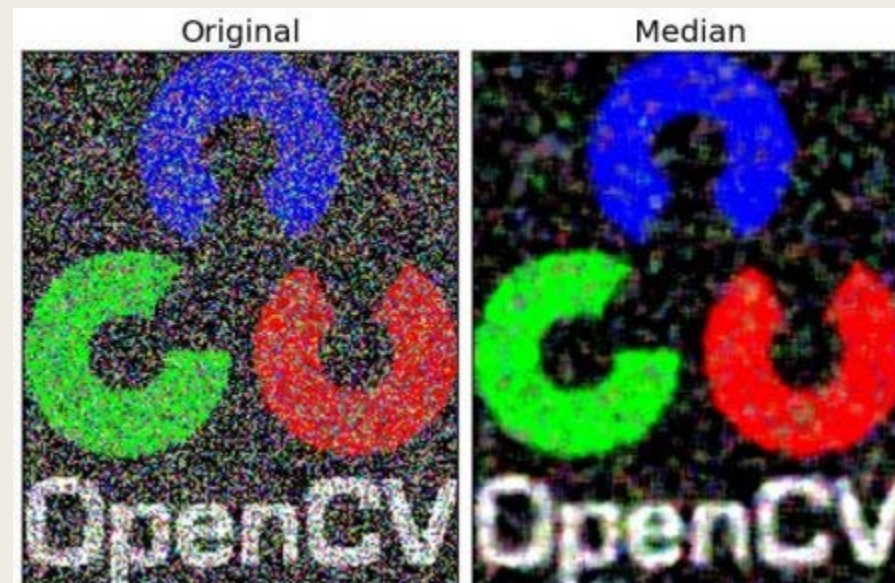
- Each pixel is a sum of fractions of each pixel in its neighborhood
- Very fast, but does not preserve sharp edges well

$1/16$	$1/8$	$1/16$
$1/8$	$1/4$	$1/8$
$1/16$	$1/8$	$1/16$

Median Blur



- Also a good way to remove noise
- Each pixel becomes the median of its surrounding pixels



```
img = img.SmoothMedian(3);
```