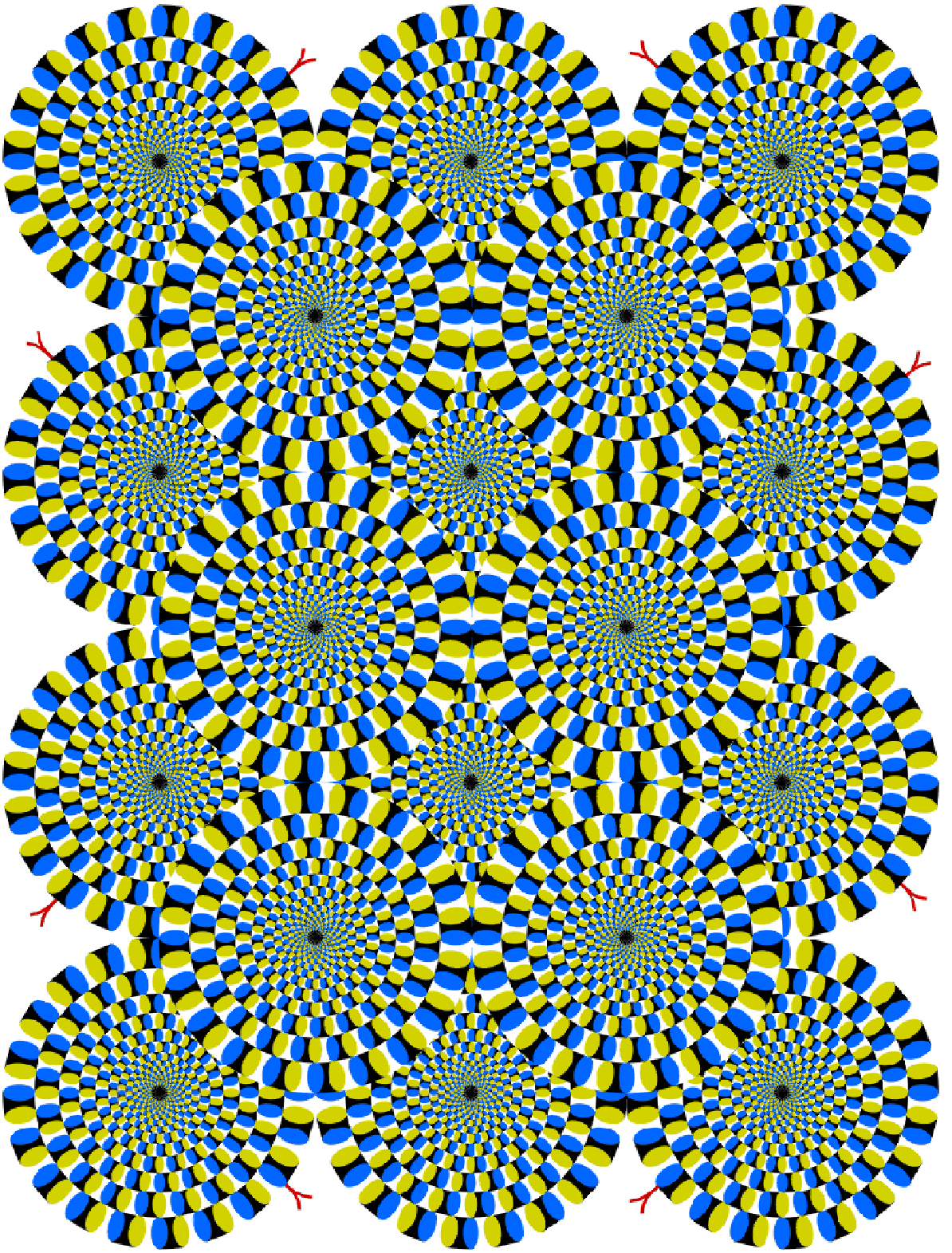


9

Image Processing & Perception

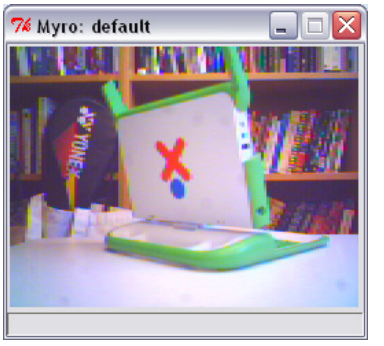
Seeing is believing.
Proverb

Opposite page: Rotating Snakes (A Visual Illusion)
Created by Akiyoshi Kitaoka (www.ritsumei.ac.jp/~akitaoka/index-e.html)

Your robot has a small digital camera that can be used to take pictures. A picture taken by a digital camera is represented as an *image*. As you have seen in the previous chapter images can be drawn and moved about in a graphics window just as if it were any other graphics object (like a line, circle, etc.). We also saw in Chapter 5 how an image taken from the Scribbler's camera can be viewed on your monitor in a separate window. In this chapter we will learn how to do computing on images. We will learn how images are represented and how we can create them via computation and also process them in many different ways. The representation of images we will use is same as those used by most digital cameras and cell phones and the same representation can be used to display them in a web page. We will also learn how an image taken by the robot's camera can be used to serve as the camera's eye into its world using some image understanding techniques. Image understanding is the first step in visual perception. A robot equipped with even the most rudimentary visual perception capabilities can be designed to carry out more interesting behaviors.

What is an Image?

In Myro you can issue a command for the robot to take a picture and display it on the computer's monitor using the commands:




```
pic = takePicture()
show(pic)
```

The picture on the next page shows an example image taken from the Scribbler's camera. An image is made up of several tiny picture elements or *pixels*. In a color image, each pixel contains color information which is made up of the amount of red, green, and blue (also called, RGB) values. Each of these values can be in the range [0..255] and hence it takes 3 bytes or 24 bits to store the information contained in a single pixel. A pixel that is colored pure red will have the RGB values (255, 0, 0). A grayscale image, on the other hand, only contains the level of gray in a pixel which can be represented in a single byte (8 bits) as a number ranging from 0..255 where 0 is black and 255 is white. The entire image is just a 2-dimensional array of pixels. For example, the images obtained from the Scribbler have 256x192 (WxH) or a total of 49,152 pixels. Since each pixel requires 3 bytes of data, this image has a size of 147,456 bytes.

All digital cameras are sold by specifying the number of megapixels. For example, the camera shown below is rated at 6.3 megapixels. This refers to the

size of the largest image that it can take. The more pixels in an image the better the image resolution can be when it is printed. With a 6.3 megapixel image you will be able to create good quality prints as large as 13x12 inches (or even larger). By comparison, a conventional 35mm photographic film has roughly 4000x3000 or 12 million pixels. Professional digital cameras easily surpass this resolution and that is why, in the last decade or so, we have seen a rapid decline in film-based photography. For displaying sharp images on a computer you need less than half the resolution offered by the camera shown here. The Scribbler camera, with an image size of 147,456 bytes is only about 0.14 megapixels. While this is low resolution it is sufficient for doing visual perception for the robot.

To make electronic storage and transfer of images (web, e-mail, etc.) fast and convenient the data in an image can be compressed. Several formats are available to electronically store images: JPEG, GIF, PNG, etc. JPEG is the most common format used by digital cameras, including the Scribbler's. JPEG enables excellent compression coupled with a wider range of colors compared with the GIF format making it useful for most image applications. Myro supports both JPEG and GIF image formats. When we intend to process an image, we will always use the JPEG format. We will primarily use the GIF format for creating animated images.

Myro Image Basics

After you take a picture with the Scribbler as above you can get some information about the size of the picture with the `getWidth()` and `getHeight()` functions:

```
picWidth = getWidth(pic)
picHeight = getHeight(pic)
print "Image WxH is", picWidth, "x", picHeight, "pixels."
```

If you wish to save the image for later use, you can use the Myro command:

```
savePicture(pic, "OfficeScene.jpg")
```

The file `OfficeScene.jpg` will be saved in the current folder. The .jpg extension signals the command to save the image as a JPEG image. If you wanted to save it as a GIF image, you can use the .gif extension as shown below:

```
savePicture(pic, "OfficeScene.gif")
```

Later, you can load the picture from disk with the `makePicture()` function:

```
mySavedPicture = makePicture("OfficeScene.jpg")
show(mySavedPicture)
```

A nice command combination that allows you to navigate and then select the image to load is:

```
mySavedPicture = makePicture(pickAFile())
show(mySavedPicture)
```

The `pickAFile` command gives you a navigational dialog box, just like the one you see when you open and select files in many other computer applications. You can navigate to any folder and select a file to open as an image. In fact, you can use the `makePicture` command to load any JPEG picture file regardless of whether it was created from your own digital camera or one you downloaded from the web. Below, we show you how to load a picture and display it:

```
lotusTemple = makePicture(pickAFile())
show(lotusTemple, "Lotus Temple")
```

If you move your mouse and click anywhere in the picture, you can also get the x- and y- coordinates of the pixel you selected along with its RGB values. This is shown in the picture on the right. The mouse was clicked on the pixel at location (65, 196) and its RGB values were (168,174,104). Can you guess where that is in the picture? By the way, the show command takes an optional second parameter which is a string that becomes the title of the image window. One advantage of being able to load and view any image is that we can also learn to process or

manipulate such images computationally. We will return to image processing later in this chapter.

A Robot Explorer

If you do not need a full color picture, you can tell Myro to capture a gray-scale image by giving the `takePicture()` function the `"gray"` parameter.

```
grayPic = takePicture("gray")
show(grayPic)
```

You will notice that taking the gray-scale picture took less time than taking the color picture. As we explained earlier, a gray scale picture only uses one byte per pixel, instead of three. Because gray-scale images can be transferred from the robot to your computer faster than full color images, they can be useful if you want the images to update quickly. For example, you can use the `joyStick()` function combined with a loop that takes and displays pictures to turn your robot into a remotely piloted explorer, similar to the mars rovers.

```
joyStick()
for i in range(25):
  pic = takePicture("gray")
  show(pic)
```

The above code will open a joy stick window so that you can control your robot and then capture and show 25 pictures, one after the other. While the pictures are being captured and displayed like a movie, you can use the joystick to drive your robot around, using the pictures to guide it. Of course, if you removed the `"gray"` parameter from the `takePicture()` function call, you would get color pictures instead of grayscale pictures, but they would take much longer to transfer from the robot to your computer, and make it more difficult to control the robot.

Animated GIF Movies

The `savePicture()` function also allows you to make an animated GIF, which is a special type of picture that in a web browser will show several pictures one after another in an animation. To save an animated GIF, you must give the `savePicture()` function a list of pictures (instead of a single picture) and a filename that ends in `".gif"`. Here is an example:

```
pic1 = takePicture()
turnLeft(0.5,0.25)
pic2 = takePicture()
turnLeft(0.5,0.25)
pic3 = takePicture()
turnLeft(0.5,0.25)
pic4 = takePicture()

listOfPictures = [pic1, pic2, pic3, pic4]

savePicture(listOfPictures, "turningMovie.gif")
```

The best way to view an animated GIF file is to use a web browser. In your favorite browser use the `FILE->Open File` menu, and then pick the `turningMovie.gif` file. The web browser will show all frames in the movie, but then stop on the last frame. To see the movie again, press the `"Reload"` button. You can also use a loop to make a longer movie with more images:

```
pictureList = []  #Start with an empty list.
for i in range(15):
  pic = takePicture()
  pictureList = pictureList + [pic]   #Append the new picture
  turnLeft(0.5,0.1)

savePicture(pictureList,"rotatingMovie.gif")
```

The above commands create an animated GIF movie from 15 pictures taken while the robot was rotating in place. This is a nice way to capture a complete scene around the robot

Making Pictures

Since an image is just an array of pixels it is also possible to create or compose your own images by coloring in each individual pixel. Myro provides simple commands to fill in or examine a color or grayscale value in individual pixels. You can use computations to determine what to fill in each pixel. To start with, you can create a blank image as follows:

```
W = H = 100
newPic = makePicture(W, H, black)
show(newPic)
```

The 100x100 pixel image starts out with all its pixels colored pure black (i.e. RGB = (0,0,0)). If you'd rather like all pixels to be a different color, you can specify its RGB values:

```
newPic = makePicture(W, H, makeColor(R,G,B))
```

Alternately, if you ever need to, you can also set all the pixels to any color, say white, using the loop:

```
for x in range(W)
   for y in range(H):
      pixel = getPixel(newPic, x, y)
      setColor(pixel, white)
repaint(newPic)
```

The `getPixel` command returns the pixel at specified `x`- and `y`- locations in the picture. `setColor` sets the given pixel's color to any specified color. Above we're using the predefined color white. You can create a new color by specifying its RGB values in the command:

```
myRed = makeColor(255, 0, 0)
```

To visually select a color and its corresponding RGB values, you can use the command:

```
myColor = pickAColor()
```

A color palette will be displayed from which you can select a color of your choosing. The palette also shows you the chosen color's RGB values. After you select a color and press OK, the value of `myColor` will be the selected color.

The `repaint` command refreshes the displayed image so you can view the changes you made. In the example loop above, we are using it after all the pixels are modified. If you want to see the changes as they are being made, you can include the repaint command inside the loop:

```
for x in range(W)
   for y in range(H):
      pixel = getPixel(newPic, x, y)
      setColor(pixel, white)
      repaint(newPic)
```

You will be able to view each pixel being changed. However, you will also notice that repainting this way takes a considerable amount of time even on the small image we are creating. Thus, it is a good idea to refresh once all the pixels are modified.

In addition to the `setColor` command, Myro also has the `setRGB` command that can be used to set a pixel's color. This command uses RGB values themselves, instead of a color.

```
setRGB(pixel, (255, 0, 0))
```

There is also a corresponding command to get the RGB values of a given pixel:

```
r, g, b = getRGB(pixel)
```

The `getRGB` command returns the triple (R,G,B) that can be assigned to individual variables as shown above. Additionally, given a pixel, you get the individual RGB values using the commands `getRed`, `getGreen`, and `getBlue`. These are described in more detail at the end of this chapter and are illustrated in examples below.

Many image creation and processing situations require the processing of every pixel in the image. In the loops above, we are using the x- and y- variables to reach every pixel in the image. An alternate way of accomplishing the same result is to use the following form of the loop:

```
for pixel in getPixels(newPic):
   setColor(pixel, gray)
repaint(newPic)
```

Like the `timeRemaining` function used in earlier chapters, `getPixels` returns the next pixel to be processed each time around the loop thus guaranteeing that all pixels of the image are processed. Just to watch how the loop above works you may want to try and put repaint inside the loop. The difference between the two methods is that the first loop gets a pixel at the given `x-` and `y-` coordinates, while the latter gives you a pixel at a time without worrying about its `x-` and `y-` values.

Shades of Gray

Using the basic image pixel accessing and modifying commands one can write computations to create interesting and creative images. To introduce you to basic image creation and processing techniques, we will create some images entirely in the grayscale spectrum. In a JPEG image, all shades of gray have equal RGB values. The darkest shade of gray is (0,0,0) or black and the brightest is (255,255,255) or white. Rather than worrying about the triple of RGB values, we can think of just a single value in the range 0..255 to represent the grayscale

spectrum. We can write a simple function that will return the RGB shade of gray as follows:

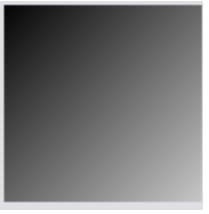
```
def gray(v):
    # returns an rgb gray color for v
    return makeColor(v, v, v)
```

Let's create an image that displays the entire range of shades of gray using the following rule:

pixel at x, y is colored using the grayscale x+y

That is, a pixel in the image at 0, 0 will get the shade 0+0=0 or black, which the pixel at 50,50 will get the shade 50+50=100 which will be a midrange gray. We can accomplish this using the following loop:

```
def main():
    MAX = 255
    W = 100
    H = 100

    # Create the blank image
    myPic = makePicture(W, H)
    show(myPic, "Shades of Gray")

    # Fill in each pixel with x+y shade of gray
    for x in range(W):
        for y in range(H):
            setPixel(myPic, x, y, gray((x+y)%(MAX+1))
```



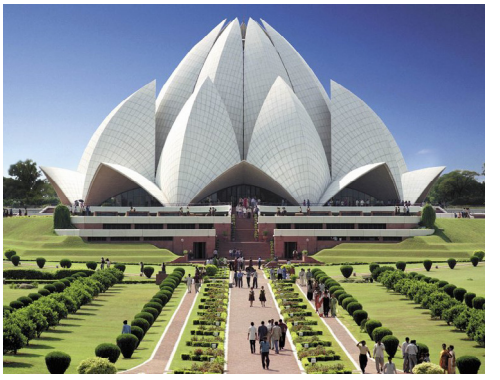
We have used the variable `MAX` above to represent the maximum value of the grayscale range. In the last line we take the remainder `(x+y)%(MAX+1)` to ensure that the gray value will be in the range 0..255. The image generated by the above commands is shown on the right. It may seem strange at first that we have used the x- and y- locations of a pixel to compute its grayscale or brightness value. Since `x-` and `y-` are numbers and since the grayscale value itself is a number it is ok to do this. Recognizing that `x-` and `y-` values (and their sum) might be larger than the range of the grayscale spectrum, we take the remainder to wrap the values around.

Do This: Write a complete program and try this out. Once you have the program, try it out on an image of size 123x123 (why?). Then again, try it out on a much larger image, say 500x500.

The grayscale range basically represents levels of brightness. We can have some more fun by changing the range, say to 0..50. Then we can modify the gray function as shown below to proportionately map any value in the range 0..50 to the range 0..255:

```
MAX = 50
def gray(v):
    # returns an rgb gray color for v
    brightness = v*255/MAX
    return makeColor(brightness, brightness, brightness)
```

Do This: Rewrite the program to use the above definition of `gray` and create a 500x500 image. Modify the value of MAX to 25 and create a 100x100 image (shown above). Note that if you set MAX to 255, it will revert to the earlier version of the function.



Do This: Next use the rule:

*pixel at x, y is colored using the grayscale x*y*

Remember to take remainder as above. Change `MAX` to 250 and create a 500x500 image. Try other sizes and values of `MAX`.

You can experiment with other rules, geometric transformations, and trigonometric functions. See exercises at the end of this chapter for more suggestions.

Image Processing

One doesn't always have to start with a blank image to create a new one. You can take existing images and transform them in interesting ways. The basic principles remain the same: you access an individual pixel and its color value

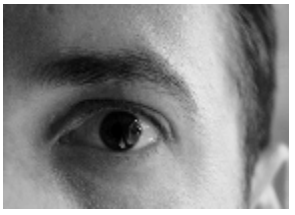
and transform it in any way you like. In this section we will do some simple image transformations to illustrate this. All the image transformations we will learn will require the same transformation on all pixel values of a given image. You can of course choose to select specific regions if you like.

We will use the sample images shown here to learn image transformations. You can feel free to select one or more images from your own digital camera. In most of the examples we have used transformations on grayscale images. You should feel free to use color images and experiment with them. You will also notice that the bigger the size of the image, the longer it will take to transform it. This is a direct consequence of the amount of computation you have to do. For example, to transform a 500x500 image you will be operating on 250,000 pixels. If each pixel transformation involves 10 operations, you will be specifying 25 million operations! How long this will take depends on the computer you are using. We will return to this in more detail later in the chapter. For now, try and restrict yourself to small size images, say no bigger than 300x300 pixels.

As we mentioned earlier, even the most rudimentary digital cameras these days give you several megapixels of image resolution. So the first thing we should learn to do is how to shrink a given image into a smaller size.

Shrinking & Enlarging

Original Image (300x213)



After Shrinking
(150x106, F=2)

Lets us write a program that will take an input image and shrink it by a given factor, say F. For example, if the original image is 3000x3000 pixels and we shrink it by a factor of 10, we would end up with an image of size 300x300 pixels. This transformation boils down to selecting the value of each of the pixels

in the new image based on the old image. We can use the following rule as our transformation:

*New pixel at x, y is a copy of the old pixel at x*F, y*F*

The program below prompts the user to pick an image, display it, and then ask the user to specify the shrinking factor (which has to be larger than 1).

```
def main():

    # read an image and display it
    oldPic = makePicture(pickAFile())
    show(myPic, "Before")

    X = getWidth(oldPic)
    Y = getHeight(oldPic)

    # Input the shrink factor and computer size of new image
    F = int(ask("Enter the shrink factor."))
    newx = X/F
    newy = Y/F

    # create the new image
    newPic = makePicture(newx, newy)

    for x in range(newx):
        for y in range(newy):
            setPixel(newPic, x, y, getPixel(myPic, x*F, y*F))

    show(newPic, "After")
```

Do This: Select one or more images and make sure they are in JPEG format and are high resolution images (for our purposes, 400x400 pixels or larger). Run the program on each image and observe the output for different values of F. Modify the program to use the `savePicture` command to save the images into a file for use in examples below.

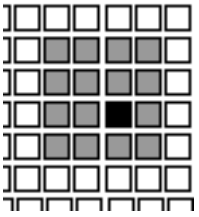
Enlarging is done in a complementary manner using the rule:

New pixel at x, y is a copy of the old pixel at x/F, y/F

The size of the new image will be F times the size of the original image.

When you enlarge an image from a smaller image you are really generating data where there wasn't any. Therefore, the resulting image is likely to be more

blurred than the original. This is similar to so called *upconverting* DVD players that claim to upconvert a movie to high definition from a standard signal. If you look closely, you will be able to see flaws in the unconverted images.



15-pixel neighborhood

Blurred Image

Blurring & Sharpening

You can blur an image by taking the average color values of neighboring pixels. The size of the neighborhood determines the amount of blurring. Below, we show how to blur each pixel using a neighborhood of 4 (see picture on next page):

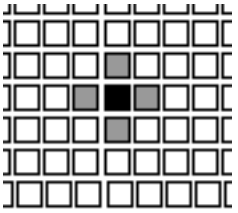
```
n1 = getPixel(oldPic, x, y)
n2 = getPixel(oldPic, x, y-1)
n3 = getPixel(oldPic, x-1, y)
n4 = getPixel(oldPic, x+1, y)
n5 = getPixel(oldPic, x, y+1)

v = (getRed(n1)+…+getRed(n5))/5
setPixel(newPic, x, y, gray(v))
```

Notice in the above that `n1` is the original pixel. You can use different sizes and shapes of neighborhoods to create more elaborate blurring effects. The picture above shows the effect of using a larger, still square, neighborhood.

Instead of averaging the pixel values, if you subtracted the pixel values you end up sharpening the image. That is,

```
v = 5*getRed(n1) -
      (getRed(n2)+…+getRed(n5))
setPixel(newPic, x, y, gray(v))
```

Do This: Write a complete program that uses the above neighborhood averaging formula to blur and sharpen images. Depending on the size and shape of the neighborhood you choose, the loop index ranges will vary. For example, the loop for the 4-pixel neighborhood will appear as follows:



4-pixel neighborhood

```
for x in range(1,X-1):
    for y in range(1,Y-1):
        # Average pixel values…
```

As the size of the neighborhood grows, you will have to adjust the start and stop values of the `x` and `y` variables in the loop above.

Blurred Image

Negative & Embossing

Creating a negative of an image is obtained by inverting the pixel values. For grayscale values this amounts to subtracting from 255. For color pixels you have to subtract from 255 for each of the RGB values. The rule for grayscale is:

New pixel at x,y is 255-value of old pixel at x,y

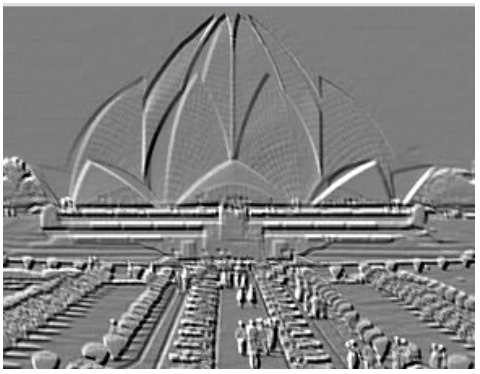
Sharpened Image

You can specify the entire transformation using the loop:

```
for x in range(X):
    for y in range(Y):
        pixel = getPixel(oldPic, x, y)
        v = MAX - getRed(pixel)
        setPixel(newPic, x, y, gray(v))
```

Similarly, you can create an embossing or relief effect by subtracting from the current pixels value, the value (or a fraction thereof) of a pixel two pixels away:

Negative



Emboss

```
pixel1 = getPixel(oldPic, x, y)
pixel2 = getPixel(oldPic, x+2, y+2)
v = getRed(pixel1) – getRed(pixel2)
setPixel(newPic, x, y, gray(v))
```

or

```
v = getRed(pixel1) + (MAX/2 – getRed(pixel2))
```

Do This: Implement the two rules shown above. Try creating a negative of a color image, as well as a grayscale image. Try different fractions for embossing to see their effect.

Most likely, as you were reading and experimenting with the above techniques, you were wondering about the kind of image processing done by some of the popular photo-editing software programs. You may already be familiar with or might have heard of Adobe Photoshop, Fireworks, GIMP, etc. Most of these programs use techniques similar to the ones we have described above and provide many more features and options for image manipulation.

"THE BEST - The All-WNY boys volleyball team surrounds the coach of the year, Frontier's Kevin Starr. Top row, from left, are Eden's Brian Heffernan, Lake Shore's Tom Kait, Sweet Home's Greg Gegenfurtner, Kenmore West's George Beiter. Bottom row: from left, are Eden's David Nagowski and Orchard Park's Rich Bland."

Image Understanding & Robot Vision

The picture shown on right was taken from a newspaper article in a local newspaper in western New York. Read the accompanying

caption and see if you can identify each individual in the picture. After reading the caption, you will no doubt confirm that the person in the front row center is Coach Kevin Starr. It would be really hard to do identify the people if the caption weren't present. Still you would be able to answer questions like, how many people are in the picture. This is an example of the *image understanding* problem: Given an image, how can you determine, computationally, that there are seven people in the image, that three of them are kneeling and four are standing, etc. You would also not have any trouble drawing a box on the face of each person in the image. In this section we introduce you some basic techniques in image understanding and use them for creating a visual perception system for the Scribbler robot. You are already familiar with the basic image processing commands. The same commands will be used to do image understanding.

For a robot to perceive its environment visually it has to start by taking a picture. Once the image is obtained perception of the contents of the image are done computationally. Instead of taking on the image understanding task shown above, we will start simple: how does a Scribbler recognize a ball? Once it recognizes it, can it follow the ball wherever it goes?

The second problem becomes easy once we know how to recognize a ball. We can write a simple control program like this:

```
while timeTemaining(T):
    # take picture and locate the ball
    …
    if <ball is on the left of visual field>:
        turnLeft(cruiseSpeed)
    elif <ball is in the middle of visual field>:
        forward(cruiseSpeed)
    else:
        turnRight(cruiseSpeed)
```

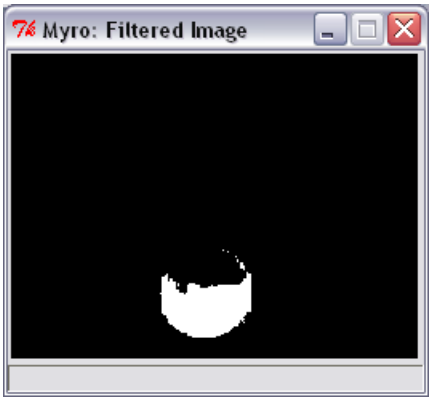
The main problem is locating the ball in the image. Given the pixel manipulation commands you have learned so far, think about how we might accomplish this?

The basic principles in image understanding rely on starting at the pixel level. Let us make things a little more concrete. Take a look at the image shown here on the right. The picture was taken by a Scribbler robot. For our purposes we have exaggerated the scene by choosing a bright colored (pink) ball. When you

click your mouse anywhere on the image it will give you the RGB values of that pixel. The picture shows the values of a pixel in the center of the ball. Notice that the red pixel is 253 and the green pixel's value is 66. By the way, true pink has RGB values (255,175,175). We can use this information as a way of identifying the ball. Here is a simple image transformation loop that turns all pixels in the image into white or black pixels. If a pixel has a high red value and a lower green value it will be turned white else it will be turned black.

```
for pixel in getPixels(p):
    r, g, b = getRGB(pixel)
    if r > 200 and g < 100:
        setRGB(pixel, (255, 255, 255))
    else:
        setRGB(pixel, (0, 0, 0))
```

The resulting image is shown here. As you can see, by just selecting the appropriate values of individual colors you can filter out the unwanted regions and focus on just the colors you are interested in. Even though the entire ball has not been filtered, it is sufficient for use in identifying its location. By tweaking the RGB threshold values you can further refine this filter. Further processing will be required to identify the ball's location in the visual field (which is the image).



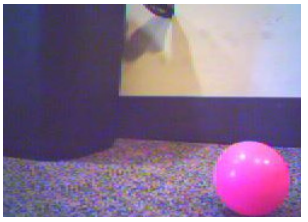
We are interested in locating the ball in the left, right, or center of the image. One way to accomplish this is to locate the `x-` coordinate of the center point of the ball. One way to do this is to take the average of the `x-` locations of all white pixels in the image:

```
def locateBall(picture):
    tot_x = 0
    count = 0
    for pixel in getPixels(p):
        r, g, b = getRGB(pixel)
        if r > 200 and g < 100:
            tot_x = tot_x + getX(pixel)
            count = count + 1
    return tot_x/count
```

Alternately, you could also average the x- locations of all pink pixels without using the filter. On the image shown, the average x- value returned was 123. Given that the image is 256 pixels wide, 123 is just about in the center. Below, we show the locations obtained for each of the images shown:

location = 41

location = 123



location = 215

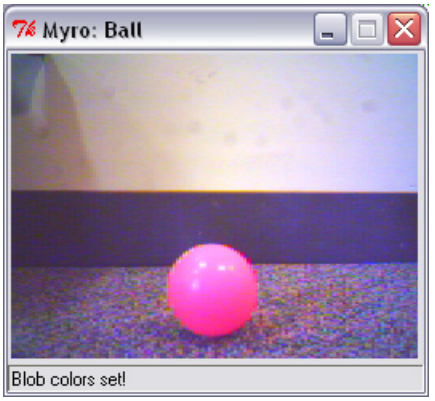
Given that the image's width is 256 pixels, you can divide the visual field into three regions: left (0..85), center (86..170), and right (171, 255). The robot control to follow the ball can then be defined as shown below:

```
while timeTemaining(T):
    # take picture and locate the ball
    pic = takePicture()
    ballLocation = locateBall(pic)

    if ballLocation <= 85:
        turnLeft(cruiseSpeed)
    elif ballLocation <= 170:
        forward(cruiseSpeed)
    else:
        turnRight(cruiseSpeed)
```

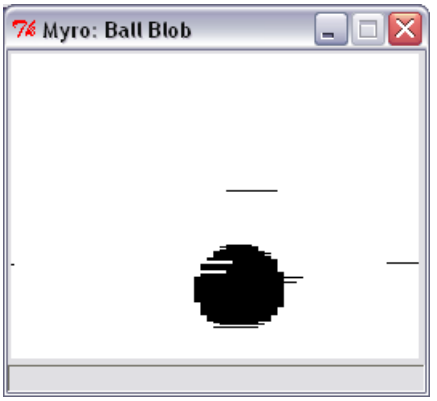
Do This: Implement the above program to recognize a given object in the picture. The object could be a box, a ball, a trash can, etc. Experiment with different colored objects. Can you use these techniques to recognize any object? Say an animal? A person? Etc.

Blobs

The kind of threshold filtering used above is commonly called *blob filtering*. All pixels satisfying the selected color properties are to be identified and the rest eliminated. This is such a common operation in image processing that many cameras and camera hardware integrate blob filtering as a primitive. The Scribbler also has this capability. Using this, you do not have to worry about identifying specific color values for the filter threshold. All you have to do is take a picture and then display it. Then using your mouse, define the rectangular



Setting a blob on the ball.



Result of `takePicture("blob")`

region that contains your desired blob. To do this, take a picture, display it, and then click your mouse on the top left corner of the ball in the image and then drag it to the bottom right corner before releasing it. You have to make sure that you are connected to the robot because the selected blob region is transferred back to the robot. Once that is defined, you will be able to make use of the predefined blob filter by taking pictures using the command:

```
pic = takePicture("blob")
```

As soon as the picture is taken, the camera's blob detector goes to work and returns the resulting image (shown above). Now you can use this image as before to control your robot. You will have to average the locations of all black pixels.

Alternately, you could use the `getBlob()` function as follows:

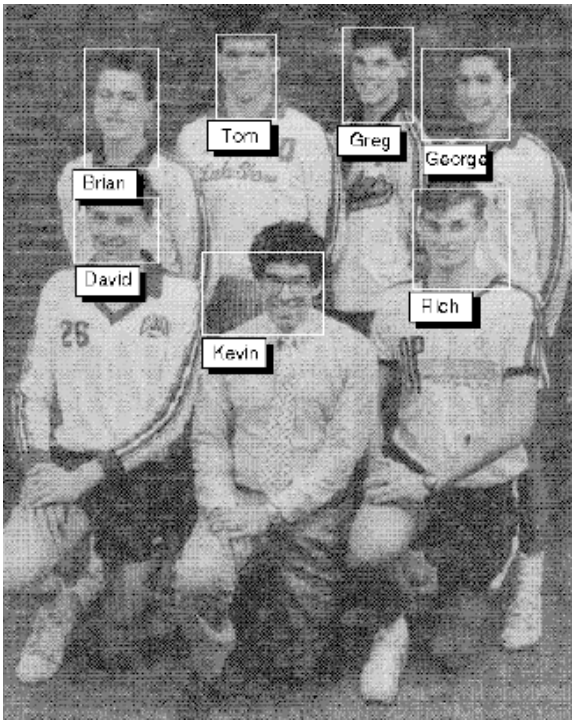
```
onPixels, avgX, avgY = getBlob()
```

The function returns three values: the number of 'on' pixels (i.e. the pixels in the image that were in the detected blob), and the average x- and y- location of the blob pixels. This is a better and much faster way of doing blob recognition and you will notice that your robot tracks the ball much better.

Do This: Redo the ball tracking example from above to use the blob features introduced here.

Further computations on an image can result in identifying object boundaries. This is called *edge detection*. These can be further used to do object recognition: Is there a car in the picture? A face? Etc.

Object recognition requires the ability to identify features that are typical of an object. The domain of image recognition and computational vision is rich with techniques and applications like these. On the right, we show the results of recognizing faces and identifying the individuals in the picture. As researchers advance these techniques and as computational capabilities **of** digital cameras increase, we are starting to see many of these incorporated into even the most basic digital cameras. Such techniques can be used to automatically focus on the main subject and also adjust exposure to get a perfect picture even for amateur photographers.



Summary

In this chapter, you have seen various Myro facilities that can be used for image creation, processing and understanding. Most image creation, processing, and understanding techniques involve pixel-level manipulations of an image. However, many interesting effects as well as applications can be developed using these simple ideas. Many of these features are increasingly found in everyday devices like digital cameras, biometric identification devices, and even automated mail sorting systems. Combined with robot behaviors these facilities can be used for modeling smarter behaviors.

Myro review

`getBlob()`
Returns a triple of `onPixels, avgX, avgY` (the number of pixels that matched the blob, the average x and y locations of blob pixels).

`getHeight(<picture>)`
`getWidth(<picture>)`
Returns the height and width of the `<picture>` object (in pixels).

`getPixel(<picture>, x, y)`
Returns the pixel object at `x,y` in the `<picture>`.

`getPixels(<picture>)`
When used in a loop, returns one pixel at a time from `<picture>`.

`getRGB(pixel)`
`getRed(<pixel>)`
`getGreen(<pixel>)`
`getBlue(<pixel>)`
Returns the RGB values of the `<pixel>`.

`makeColor(<red>, <green>, <blue>)`
Creates a color object with the given `<red>`, `<green>`, and `<blue>` values (all of which are in the range [0..255]).

`makePicture(<file>)`
`makePicture(<width>, <height>)`
`makePicture(<width>, <height>, <color>)`
Creates a picture object either by reading a picture from a `<file>`, or of the given `<width>` and `<height>`. If `<color>` is not specified, the picture created has a white background.

`pickAColor()`
Creates an interactive dialog window to select a color visually. Returns the color object corresponding to the selected color.

`pickAFile()`
Creates an interactive dialog window that allows user to navigate to a folder and select a file to open. Note: it cannot be used to create new files.

`repaint()`
`repaint(<picture>)`
Refreshes the displayed <picture>.

`savePicture(<picture>, <file>)`
`savePicture(<picture list>, <gif file>)`
Saves the `<picture>` in the specified file (a GIF or JPEG as determined by the extension of the `<file>`: `.gif` or `.jpg`). `<picture list>` is saved as an animated GIF file.

`setColor(<pixel>, <color>)`
`setRed(<pixel>, <value>)`

```
setGreen(<pixel), <value>)
setBlue(<Pixel>, <value>)
```
Sets the color of `<pixel>` to specified `<color>` or `<value>`.

```
show(<picture>)
show(<picture>, <name>)
```
Displays the `<picture>` on the screen in a window named `<name>` (string).

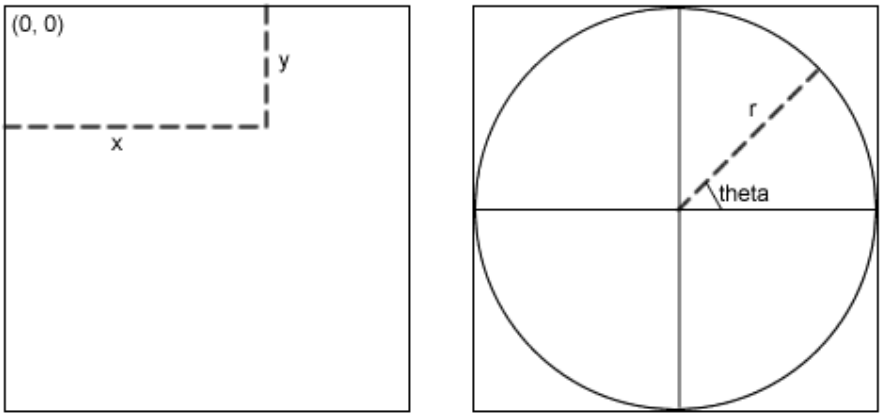
```
takePicture()
takePicture("gray")
takePicture("blob")
```
Takes a picture from the Scribbler camera. It is a color picture by default, or grayscale (`"gray"`), or a filtered image based on the defined blob (`"blob"`). See chapter text for examples.

Python Review

There were no new Python features introduced in this chapter.

Exercises

1. Instead of viewing each pixel in an image as x- and y- values in the Cartesian coordinate system, imagine that it is in a polar coordinate system with the origin



Cartesian Coordinates Polar Coordinates

in the center of the picture (See picture below).

Write a function called polar(x, y) that returns the polar coordinates (r, theta) of the given point. Using this function write a transformation as follows:

```
for x in range (1, W):
    for y in range(1, H):
        r, theta = polar(x, y)
        setPixel(myPic, x, y,
                  gray((((r+theta)%16)-8)*MAX/16+MAX/2))
```

Create a 400x400 picture using the above transformation. Start with a black picture.

2. Based on the above exercise, try transforming an image (like a portrait). For example use the rule: new x, y is a copy of original picture's $\sqrt{r * W/2}$, *theta*.

3. Write a transformation that flips an image clockwise (or counter clockwise) by 90 degrees. Try it on a square image of your choice.

4. Write an image transformation that averages two images (same size) into a single image.

5. Write a program that sews two or more images together to create a panorama.

6. Use blob detection techniques described in this chapter to recognize Coke and Pepsi cans. Write a program to hunt for a Coke or Pepsi can.

7. Can you expand on Exercise above to recognize small toy figurines of Disney cartoon characters like Mickey, Minnie, Daisy, Goofy, Pluto, and Donald?