The first of the seven elements of art that we'll deal with is *value*. This concerns how light or dark things are. Value can be an overall aspect of awork, or it can vary from place to place, but in all cases the scale we use is from the darkest value, which is black, to the lightest, which is white. We will talk about value first because it underlies line and color, and getting the values correct might be more important than getting the colors correct.

Works that consistently use dark values we call *low key*. Such work can seem brooding, or mysterious, or even frightening. Rembrandt was famous for low key paintings, and Yousuf Karsh used low key lighting for many of his famous photographs. Figure 2.1 shows some examples.

Chapter 2 Value

On the other hand, work that uses light, bright values is called *high key*. We tend to see this as cheerful and happy, but in the excess the images are washed out and little remains except color or whiteness. Figure 2.2 shows examples of high key imagery.

In this chapter we'll start by discussing the nature and representation of value in computer

Figure 2.1 – (Left) Low key photograph. (Right) A Rembrandt painting typical of his low key style.

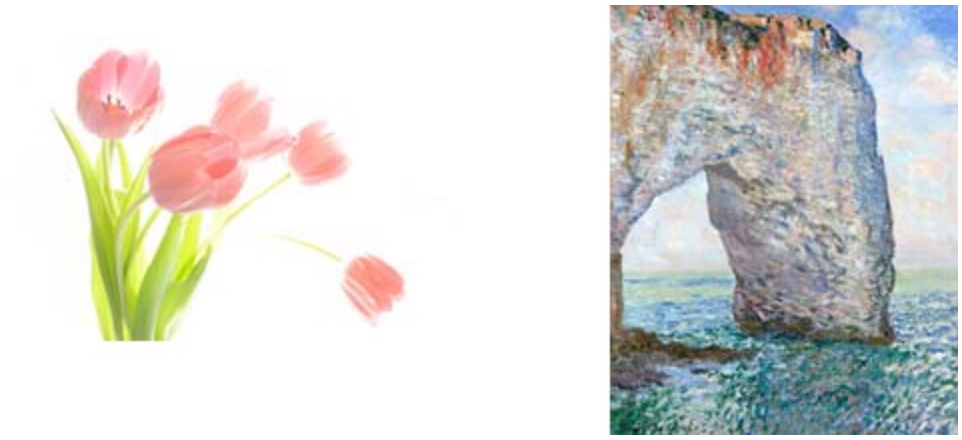


Figure 2.2 – (Left) High key photograph. (Right) A Monet painting in a high key style.

images. Then we'll talk about how to use it to modify and create visual artworks.

Pixel Values

We have seen that a *pixel*, the basic element of an image, is a color value that contains a red, green, and blue component. The value of a pixel is the brightness or intensity of the pixel, irrespective of the color it represents. Let' think of it as a grey level, and it will have a value typically between 0 and 255. Normally 0 represents black, and 255 represents white, but both the range and the value associated with any number can differ.

The number 255 represents the largest number that we can represent in a single byte, which is a standard unit of memory on a computer. One byte can store a single character. Some images use more than one byte to store a grey value, and the range of possible values is normally a multiple of the range possible for the pixel size: 2 bytes per pixel can store 65536 different levels. Without really being too restrictive, let's assume that 0-255 (256 level) is what we are dealing with. Extensions to larger ranges is pretty straightforward.

Using a programming language we can get the value of a pixel at any point (**x,y**) in an image using a built-in function like:

p = im.get(x, y)

where **p** is the pixel color, and **im** is the image we are looking at. As a pixel, p has a red, green, and blue component that we can get using functions like **red(p)**, **green(p)**, and **blue(p)**. Question: what is the value associated with **p**?

The most common way to compute this is by taking the average of the three color values; that is:

$$value = (red(p) + green(p) + blue(p))/3 \qquad \text{EQ 1}$$

This makes a lot of intuitive sense, but it presumes a basic equality of the sampled colors; that there was no bias when the pixels were created. This is not often true, although this average is very commonly used and is good enough. When does this not represent a true brightness value?

A common source for images is television in some form. Video recordings use specific encoding schemes, and they are not unbiased. In North America the NTSC standard is used. The formula for finding the value from an NTSC pixel is:

$$value = 0.299*red(p) + 0.587*green(p) + 0.114*blue(p) \qquad \text{EQ. 2}$$

We call this a *weighted mean*. Note that the multipliers sum to 1.0. The implication of this formula is that NTSC will favor blue values at the expense of green ones, and so to get a reasonable grey from a pixel we have to weight the green part higher. In Europe and other places they use a difference system called PAL. The formular for a PAL pixel would be:

$$value = 0.222*red(p) + 0.707*green(p) + 0.071*blue(p) \qquad \text{EQ. 3}$$

Any specific device may vary in its response to different colors. For most purposes the method of EQ. 1 is probably good enough.

The Grey Level Histogram

A histogram in general is a kind of graph, and is used to show how a set of numerical values are distributed. A grey level histogram shows which grey values are most common, which are least common, and what ranges of values are more populated. Since there are 256 possible values, then building a grey level histogram begins with creating 256 distinct *bins*, on for each possible value, and initially set them to 0. Then we look at every pixel, and increment the bin associated with it: finding a value of 20, for example, causes us to add 1 to bin 20.

(Code: ch2/aatoGrey)

a b c d

Figure 2.3 – Finding the value from a color image. (a) Original. (b) Mean of the colors (c) NTSC (d) PAL

When we have examined all pixels, the result is a collection of bins where bin i contains the number of pixels in the image that have the value **i**. (Code: abhistogram) We only know how many there are, not where they are. That alone can tell us a lot about the image. Figure 2.4 shows the histogram for the image of Figure 2.3b, the values of the windsurfer image. From this we can see that there is a large number of pixels around the value 149 and lesser peaks around 80, 100, and 200. The mean is shown as a red line, and we can see that there are more pixels greater than the mean (i.e. lighter) than there are smaller, making the image a bit on the bright side. A darker image, the right image of Figure 2.4, has a peak nearer to the 0 value.

An image of text would have two peaks. One near the background value (usually around 255) and another near the text value, which will be near 0. Depending on how the image was created, the two levels will not be the only ones. *Noise* and illumination effects creep into real images causing values that are unwanted. (Code: ch2/acrange)

We can find out if certain values correspond to certain physical areas by selecting only those pixels to draw and setting all other pixels to white. In the case of the windsurfer image (Figure 2.3a) there is a sharp peak in the histogram between 190 and 200. What does that represent? When we draw only those pixels we find that they represent specific parts of the sky, not by position but by what kind of object it represents, in this case sky. In the case of portrait (Figure 2.4) the second major peak between 32 and 60 represents the person's hair and jacket, along with some shadow under the chin. We can use this kind of information when trying to compose works based on images.

Example: Contouring

The idea of selecting value ranges for display leads to an idea: why not break up the image by value into a collection of range values, and select on value to represent the entire range? Pixels with a specific value are likely to be close to other pixels with similar values. Not always, of course, but this similarity would naturally define areas that belongs to the same kind of

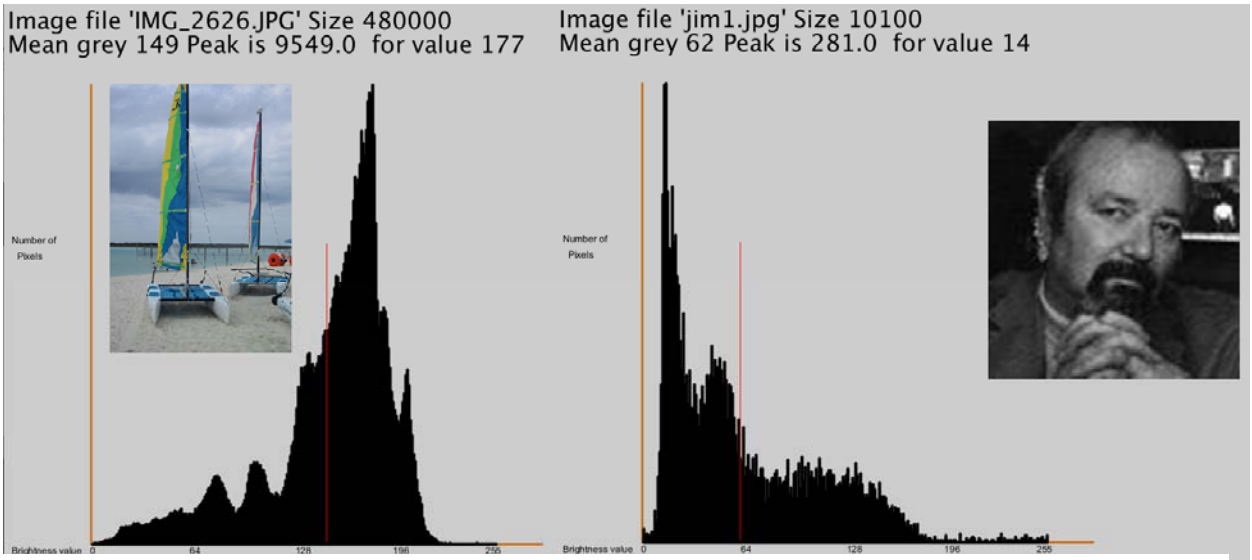


Figure 2.4 – Histograms (Left) Histogram of the values of Figure 2.3b (Right) Histogram of a grey level portrait.

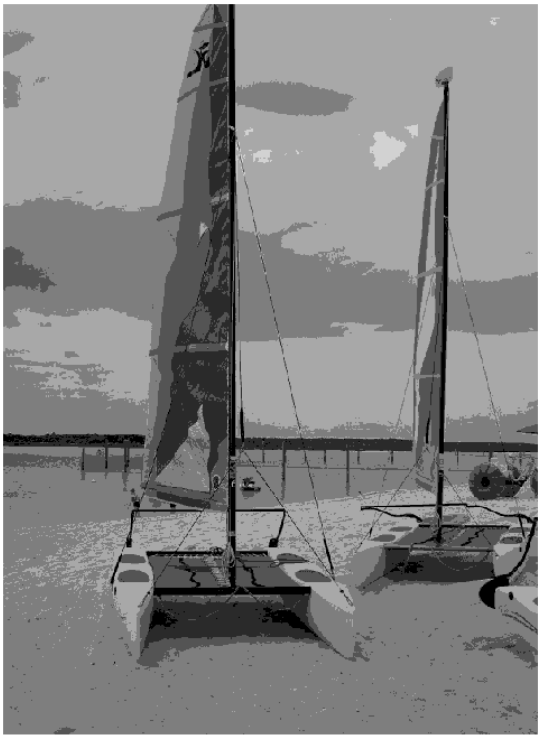
Figure 2.5 -

objects in the scene. The result should be a set of regions or layers that flowed from one level to another but with greater differences between regions than in the original. What we would do is:

1. Select a number of distinct value ranges N that we wish to break the image into.
2. Examine each pixel in the image and
3. Compute the value V for that pixel
4. Compute the number of values in each range as R = 256/N
5. Let W = (V/R*R) using integer arithmetic
6. Replace the pixel with the value W

Integer arithmetic is the trick for a simple solution here. When using integers there are no decimal points or fractions, so while 4/2 = 2, 4/3 = 1. The result of I/J is the number of times J goes into I with no fraction or remainder. (Code: ch2/adrange)

As an example, let's suggest a value of N = 10. This means that the final image will contain just 10 distinct values. In this example, R = 256/N = 256/10 = 25, so each value in the final image represents 25 values in the original. When transforming the values, we see that a value of 128 in the original becomes (128/25)*25 = 5*25 = 125. In fact, values of 127, 126, 130, and 134 all have a value of 125 in the final image. Figure 2.6 shows the result for the two images (windsurfer and portrait) that we've been using. The contours are very clear in these images.



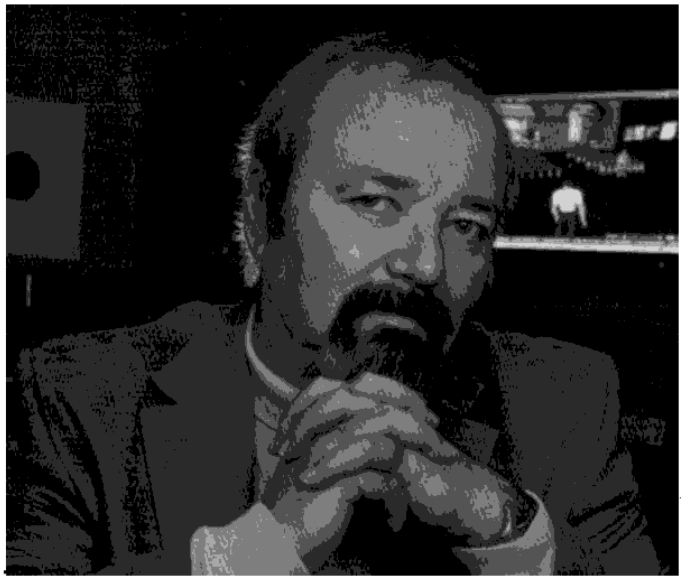


Figure 2.6 -

In a technical sense what we've done is *resampled* the pixels so that there are fewer distinct levels. This can be of advantage in art creation. We can use this method to make cartoon-like images, creating pointillist works, and for creating an illusion of distance just to suggest three applications.

Contrast

Contrast is the range of grey levels that are occupied (I.E. have pixels with that level) compared with the total number of levels possible. The idea is that there may be grey values at the bottom or top of the possible range that have no pixels belonging to them, and the fraction of the range that *does* have pixels is a measure of how much contrast is in the image. An image with low contract can be either bright or dark, but the range of pixels is compressed into a smaller range than it needs to be. There are no pixels at the low and/or the high ends of the range. We could stretch the histogram by moving the lowest actual pixel values to the lowest possible ones, and doing that iterative up the brightness levels.

This is an automatic level adjustment that we call *histogram equalization*, which is an attempt to make each brightness level have nearly the same number of elements. A completely flat histogram for a 256x256 image having 256 brightness levels would have 256 bins, each containing a value of 256. It is unlikely that any real image will have a histogram like this, but the histogram equalization process attempts to make one by deciding how many pixels should be in each bin, and then changing their values to make that happen.

In general, if there are N possible levels then each bin in the histogram would contain 1/N of the pixels in the image. For example, if an image has 16 rows and 16 columns and 8 possible levels, then each bin should have b=(16*16)/8, or 32 pixels in it. In addition, we'll enforce the rule that the total number of pixels a value of **k** or less will not be less than the value **k*b**. The number of pixels with value less than k is the *cumulative sum,* which we find from the histogram by:

$$cum_i = \sum_i h_i \qquad \textbf{EQ. 4}$$

In English this expression is the sum of the histogram values from bin 0 to bin i.For all possible values of **i**. Imagine an example having the following histogram:

Grey level	0	1	2	3	4	5	6	7
Actual number of pixels	6	28	34	94	40	48	2	4
Actual cumulative sum	6	34	68	162	202	250	252	256
Ideal number of pixels	32	32	32	32	32	32	32	32
Ideal cumulative sum	32	64	96	128	160	192	224	256

The first step in equalizing the number of pixels in each bin is to move pixels from bin 1 to bin 0, since the number of pixels in bin 0 is less than the idea number of 32 found using the cumulative sum. Note that we cannot split up a bin arbitrarily: either all of the pixels are moved or none are. In this case the new bin 0 will have 6+28, or 34, pixels, and the new bin 1 will have 0 pixels. Now bin 1 has too few pixels, and so we move (all) pixels from bin 2 into bin 1, making bin 2 empty. Continuing, bin 2 has too few pixels and we move pixels from bin 3 into bin 2. There are far too many, but we have to move them all, making bin 2 far too large and bin 3 empty. Should we move pixels from bin 4 into bin 3? No. The current situation is:

Grey level	0	1	2	3	4	5	6	7
Actual number of pixels	34	34	94	0	40	48	2	4
Actual cumulative sum	34	68	162	162	202	250	252	256
Ideal number of pixels	32	32	32	32	32	32	32	32
Ideal cumulative sum	32	64	96	128	160	192	224	256

We are looking at bin 3 – we don't move pixels from bin 4 because the cumulative sum for bin 3 is 162 and the ideal is 128, which is less than 162. This says that there are already enough pixels having a value 3 or less, so we leave bin 3 empty and move to bin 4. The ideal number for bin 4 is 160, and is still smaller than that actual cumulative sum of 162, so we leave bin 4 empty also. Finally, bin 5 will take the pixels from bin 4, bin 6 will take those from bin 7, and 7 will take all of the pixels from bins 6 and 7. The final histogram is:

Actual number of pixels	34	34	94	0	0	40	48	6

Figure 2.7– Histogram Equalization. (Top) A low contrast image and its histogram. (Bottom) After histogram equalization the histogram is far from perfectly distributed, but the image does show improved contrast.

Why would be care about the contrast enhancement process? Because good art often has vibrant colors and separation between distinct areas and objects. A dingy and dark artwork might sometimes be desired, but would generally not be an attraction. Images that are too bright seem *washed out* and, again, need some adjustment to make them interesting. In any case, we can now adjust the levels so as to make the contrast more even.

Histogram Specification

Thresholding

The first idea when processing the orange tree image was to select a range of hues that would represent the oranges. So, orange could be hues between 16 and 40. Setting those pixels to orange and all other pixels to white is a kind of thresholding, where pixel values above or below a certain specified value are collected into one class and assigned a single color. The common example of this in image analysis and vision is to threshold brightness values to create an image that is bi-level, meaning black and white only.

The brightness portion of the HSB coordinates of a pixel represent a level of grey, and we can use that value to threshold an image into black and white. We would commonly do this for images that were primarily bi-level in principle, like printed text or line drawings. Scanners and photos create images that have multipole grey values or color pixels. A first step in the processing of a text image is to *threshold* it so as to easily find the characters (black pixels) and the background (white pixels).

The problem in thresholding is not to set the pixel values, because that is very straightforward. The problem is to find the value of the threshold to be used that best allocates pixels into the classes that are wanted. In a typical image the brightness value ranges from 0 (black) to 255 (white), so a threshold value for an entire image would be a value between 0 and 255 inclusive. But *which* value?

Let's look at two images as examples for thresholding; these are shown in Figure 2.13. Both are excellent candidates because they are essentially black and white to begin with. If we look at the pixels, though, we'll see a variation in color values caused by variations in illumination, noise, slight differences in ink color, imperfections and stims in the paper, and so on. In addition, images in JPEG format frequently have artifacts caused by the compression algorithm used in that file format, and these artifacts tend to be more prominent around edges and boundaries.

Thresholding: Select a Value

As a first try we could use the middle pixel value, which because the range here is from 0 to 255 would be 128. In this case all pixels with a brightness value less than 128 would be set to black, and those greater than or equal to 128 would become white. The principle here is that the middle value will separate the pixels into two equal groups, but that is only true in some cases. Also, it's not always true that we want half of the pixels to be black. Still, it works better than we would expect.

Using the Mean

The mean value of the brightness values is more likely to be in the middle of the range of grey levels than is any guess we might make. The median is actually the middle value, but it's

time consuming to calculate. Using the mean means first finding the mean level of all pixels, then thresholding the pixel values as before.

Code:

// Thresholding using 128 as the threshold
`PImage img;`
`color white = color (255,255,255);`
`color black = color (  0,  0,  0);`
`color c;`
	Image to be thresholded
	White pixels are > threshold
	Others are black.

`float sum = 0.0, mean = 0.0, threshold=0;`
 	
`for (int i=0; i<img.width; i++)`
`  for (int j=0; j<img.height; j++)`
`    sum = sum + brightness( img.get(i,j));`
`mean = sum/( img.width * img.height);`
	Compute the mean by looking at all pixels in the image **img** and summing the brightness values. Then divide by the number of pixels.

`threshold = mean;` `// threshold = 128;`	Setting the variable **threshold** to 128 uses 128 as the threshold (previous discussion)

`for (int i=0; i<img.width; i++)`
`  for (int j=0; j<img.height; j++)`
`{`
`  c = img.get(i,j);`
`  if (brightness(c) < threshold)`
`  imgt.set(i,j,black);`
`  else`
`  imgt.set (i,j,white);`
`}`
	Now look at all of the pixels again.
	
	Get the brightness value.
	
	If the brightness is less than the mean, set the pixel to black, otherwise to white.

Using the Grey Level Histogram

A grey level histogram is a graph of the number of pixels of each possible brightness level. There is a histogram for each of the test images given in Figure 2.13. There are two major peaks in each, because there are two levels that form most of the image: black and white. These peaks are near 0 and near 255. Threshold selection will choose a numerical value for brightness that is in between two peaks in the histogram, if two peaks exist. This is because two peaks in the histogram are likely to represent the two pixel classes that we wish to identify. If there is one peak or more than two peaks, then that idea fails, but we nonetheless need to select a threshold that distinguishes two pixel classes.

Another definition of neighbor allows for neighboring pixels to be in any of eight directions from the one being considered (the *target*, we call it). Not only are the pixels to the left, right, top, and bottom considered as neighbors but so are the ones at the corners or the target. These are said to be 8-neightbors. Figure 2.5 shows all neighboring pixels of a general target at (x,y) and what their indices are.

The distance between the target (x,y) and any of the diagonal pixels is $\sqrt{2}$, or 1.41 approximately, so 8-connected pixels are those that are less than a distance of 1.5 from the target; or those that are at locations (x±1, y±1).

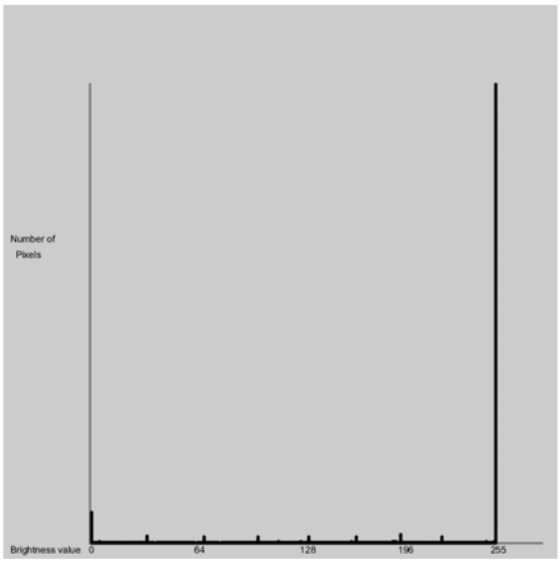
Two pixels are *4-connected* (*8-connected*) is there is a set of

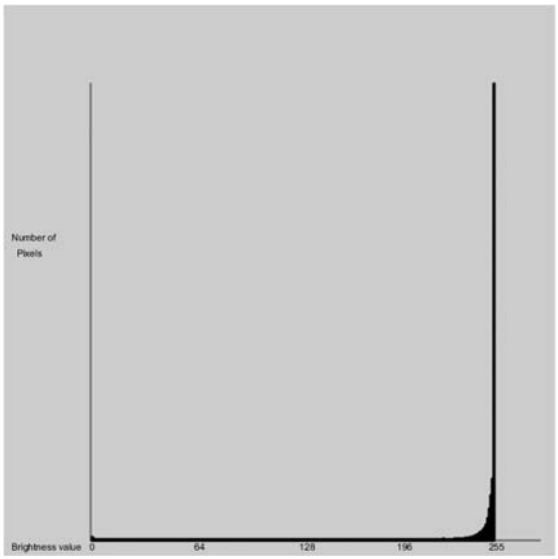
x-1,y-1	**x, y-1**	x+1,y-1
x-1, y	**x, y**	**x+1,y**
x-1,y+1	**x, y+1**	x+1,y+1

Figure 2.5 4-neighbors (black) and 8-neighbors (grey).

Figure 2.13 – Images for testing the thresholding methods. The image above is text, and is part of a page taken from this book. The image of the right is a map, an example of a line drawing.

Below: the histograms for the text image and the map image, left to right.





There are multiple threshold selection methods that use the histogram, and one that is easy to understand is the *Iterative Selection* algorithm. Using this method we select a threshold at the outset. It is common to use **T = 128**. We now group the pixels into black and white using this threshold. Now calculate the average value of the black pixels (call this **Tb**) and then do the same for the white ones (call this **Tw**). Find the average of these two values: **(Tb+Tw)/2**. Use this as the threshold, and recalculate the black and white pixels using this value as **T**. Keep repeating this process until T does not change between two iterations, and the final value of **T** is the threshold.

This method allows the pixel values to select their own threshold. We start with a known one, but if the resulting black pixels have a mean that is very low then it means we might have over-estimated the threshold, and in the next iteration it will use a smaller value, for one example.

Figure2.14 shows the result of using this method on the two sample images. They seem pretty much like the other threshold images, but we chose the examples to be nearly bi-level. Figure 2.14 also shows iterative selection and the mean value used to threshold a color image of a wind surfer, as a more apt comparison between the methods on a

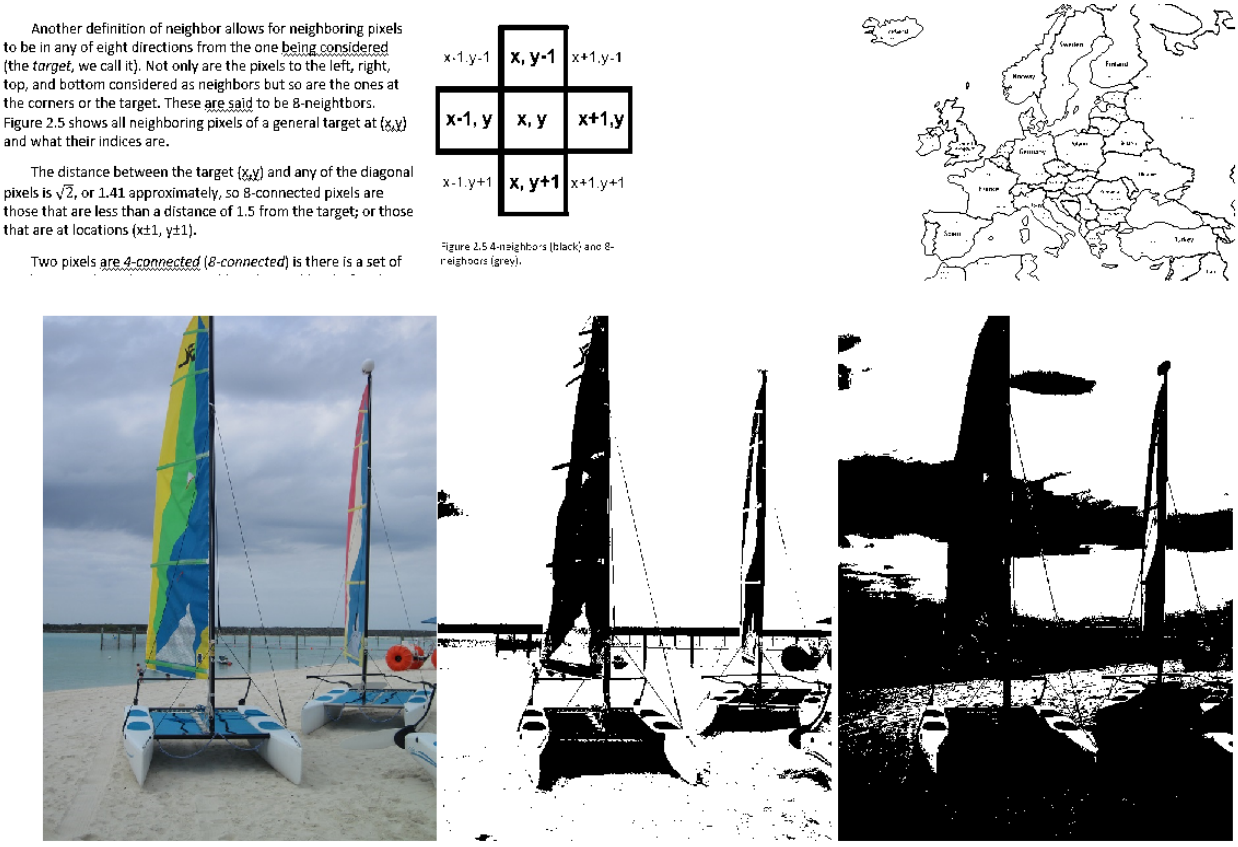


Figure 2.14 (Top) Iterative selection applied to the text and line image. (Bottom) Image of a wind surfer thresholded using iterative selection (center) and the mean value (right).

Bad Illumination: Multiple Thresholds

These examples are still pretty good images, in terms of illumination, noise, and distortion. In many cases the illumination changes from one part of the image to another, and using the same threshold applied to all of the pixels can result in a result that is too dark in some places and too light in others. Consider the test images in Figure 2.16, which are modified versions of the text and map images used previously. There is a change in brightness as we move horizontally across the image. When a single threshold is used on these images, a part of the resulting image will be far too dark, and another part will be too light.

A solution to that problem is to calculate one threshold per pixel, where we use a small region surrounding each pixel to calculate that threshold. Any method of finding a threshold will work; simply visit every pixel in the image, use a set of pixels centered at each one to compute a threshold using a thresholding method (mean or iterative selection or other method) and apply that threshold to the pixel. (Figure 2.17) We must use a second image to hold the thresholded pixels, otherwise the previously thresholded pixels will affect the threshold calculation in later regions.

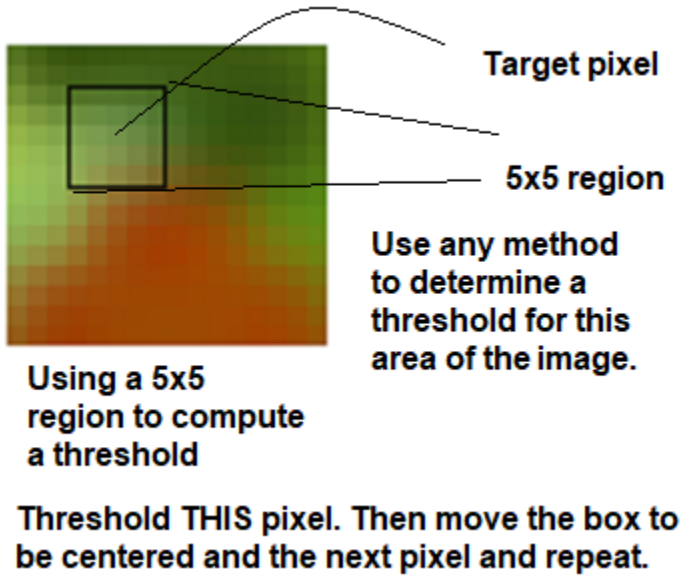


Figure 2.16 – Using a local theshold for each pixel, in this case using a 5x5 region.

Figure 2.16 also shows the example images as thresholded using the mean and iterative selection, showing that the methods fail to produce a good image in these instances, and shows the result of using a local threshold on these images. The results are not perfect, but are obviously much better. Note that as the region size increases the method becomes less local and looks more like the global image thresholding methods. If the size is too small then tiny artifacts and noise can affect the threshold and result in many small regions being classed as black.

References

Edmund Optics (2021). **Contrast**

https://www.edmundoptics.com/knowledge-center/application-notes/imaging/contrast/

Ikonomatakis N; Plataniotis, K.; Zervakis,, M. and Venetsanopoulos, A.N. (1977) **Region growing and region merging image segmentation.** 13th International Conference on Digital Signal Processing Proceedings, 1997. DSP 97., 1997 Volume: 1

Jarvaois, Chase (2012) **Photography Knowledge 101: What the Hell is SEPIA?** https://www.chasejarvis.com/blog/sepia-what-the-hell-is-it/

Kuehni, Rolf G. (2003). **Color Space and Its Divisions: Color Order from Antiquity to the present**. New York: Wiley. ISBN 978-0-471-32670-0

Kumar, S. (2019) **A straightforward introduction to Image Thresholding using python.** https://medium.com/spinor/a-straightforward-introduction-to-image-thresholding-using-python-f1c085f02d5e

Levkowitz, Haim; Herman, Gabor T. (1993). **GLHS: A Generalized Lightness, Hue and Saturation Color Model.** CVGIP: Graphical Models and Image Processing. 55 (4): 271–285.

Manahattan Distance: https://iq.opengenus.org/manhattan-distance.

Parker, J.R. (1993) **Practical Computer Vision Using C**, Wiley, New York.

Parker, J.R. (2011) **Algorithms for Image Processing and Computer Vision**, Wiley, New York.

Smith, Alvy Ray (August 1978). **Color gamut transform pairs**. Computer Graphics. 12 (3): 12–19.

Trussel, Henry. (1979) **Picture Thresholding Using an Iterative Selection Method.** IEEE Transactions on Systems Man and Cybernetics 9(5):311 - 311

Photoresistors: https://www.physics-and-radio-electronics.com/electronic-devices-and-circuits/passive-components/resistors/photoresistor.html