

# Problem Set 5

CMSC 426

Assigned Wednesday, March 11, Due Wednesday, March 25

## Texture Synthesis

(Note that this problem set is worth 75 points. I shortened it a little, due to the break)

The goal of this problem is to implement the texture synthesis method of Efros and Leung. This is described in the paper: "Texture Synthesis by Non-parametric Sampling", by Efros and Leung, in the International Conference on Computer Vision, 1999. There is a link to their web site on the class web site, which includes links to their paper and pseudocode for their algorithm. In this assignment, I have simplified their algorithm a little bit, to remove some details such as Gaussian weighting, which don't seem to be necessary to achieve good performance.

This algorithm takes a *sample* of some texture and generates a new *image* containing a similar texture. The strategy of the algorithm is to generate each new pixel in the image using a *neighborhood* of already generated pixels. One looks in the sample for similar neighborhoods, selects one of these similar neighborhoods at random, and copies the corresponding pixel into the new image. You are only required to implement this program for black and white images.

1. **SSD 20 points:**  $S$  will contain a sample of the texture we want to generate.  $T$  contains a small  $(2n+1) \times (2n+1)$  neighborhood of pixels. Not all the pixels in this neighborhood have been filled in with valid values however. So  $M$  (the *mask*) is a  $(2n+1) \times (2n+1)$  matrix that contains a 1 for each position in which  $T$  contains a valid pixel, and a 0 whenever the corresponding pixel in  $T$  should be ignored. Computing the SSD is like correlation in that we shift the template over every position in the sample, and compute a separate result for each position. Thus, the output  $D$  is the same size as  $S$ . To compute  $D(i,j)$  we shift  $T$  so that its center is right on top of  $S(i,j)$ . Then we take the difference between each valid pixel in  $T$  and the corresponding pixel in  $S$ , square the result, and then add all these together.

To begin, you should write a function called `SSD` which performs the central step of the algorithm. This will compute the sum of squared difference between a little portion of the new image you are making and every portion of the sample. It should have the form:

While you can compute SSD directly, it is interesting to note that it is closely related to correlation. To make this connection, we will ignore the mask for the moment. SSD is given by the expression:

$$D(x, y) = \sum_{i=-n}^n \sum_{j=-n}^n (S(x+i, y+j) - T(i, j))^2$$

To see how this is related to correlation, let's open up the expression:

$$\begin{aligned}
 D(x, y) &= \sum_{i=-n}^n \sum_{j=-n}^n (S(x+i, y+j) - T(i, j))^2 \\
 &= \sum_{i=-n}^n \sum_{j=-n}^n (S(x+i, y+j)^2 - 2T(i, j)S(x+i, y+j) + T(i, j)^2) \\
 &= \sum_{i=-n}^n \sum_{j=-n}^n S(x+i, y+j)^2 - \sum_{i=-n}^n \sum_{j=-n}^n 2T(i, j)S(x+i, y+j) + \sum_{i=-n}^n \sum_{j=-n}^n T(i, j)^2
 \end{aligned}$$

The bottom line shows that to compute SSD we can combine 3 separate summations. You should notice that the middle term is just -2 times the result of correlating the template,  $T$ , with the sample image,  $S$ . This gives us a sense of the relationship between SSD and correlation; in general, the higher the correlation between the template and the sample at some location, the lower the SSD will be. This is why correlation alone is sometimes used as a way of finding locations where the template and sample are similar.

Things get just a little more complicated when we have a mask. We can write this as:

$$D(x, y) = \sum_{i=-n}^n \sum_{j=-n}^n (S(x+i, y+j) - T(i, j))^2 M(i, j)$$

In this case, for every summation, we only want to use the pixels in the template that are part of the mask.

Test your function using the binary checkerboard image:

```

111000111
111000111
111000111
000111000
000111000
000111000
111000111
111000111
111000111

```

The template:

```

100
011
011

```

And the mask:

```
111
110
110
```

The output of your routine should be a 9x9 array of distances. If we call this  $D$ , then, for example,  $D(4,4)$  should be 0, while  $D(5,5)$  should be 4.

Turn in a (virtual) printout of the results, along with your code.

You may be worried about how to compute SSD when the template goes outside the boundary. Don't worry about this. You can do anything that produces reasonable results. For example, you can just treat pixels outside  $S$  as if they were 0. This will work fine, since these areas then will generally not match your template very well.

2. **15 points:** Using SSD and the pseudocode below, implement the function `FindMatches`. This should find all candidate pixels in the sample that have a neighborhood that is sufficiently similar to the template.

You might want to write `FindMatches` so that it takes three inputs: the sample image, the template, and the mask. Test your program on the same example that you used in problem 1. If you test it using the error threshold listed below, you will get two possible matches, at  $D(4,4)$  and  $D(7,7)$ . Hand in printouts of the results, and of your program.

3. **25 points:** Now complete your program. I am including below the pseudocode given by Efros and Leung. Test your program using a checkerboard pattern, and using the bricks image in `brickbw.jpg`. Hand in printouts of the results.
4. **10 points:** Use two more images of your own choosing as sample textures and generate new textures using these samples. Turn in printouts of the original images and of the textures you generate. Also, include these images in your electronic submission. For **10 points** extra credit, modify your program so that it will work with color images, and generate color textures.

**Appendix:** Below is pseudocode provided by Efros and Leung.

## Algorithm details

Let `SampleImage` contain the image we are sampling from and let `Image` be the mostly empty image that we want to fill in (if synthesizing from scratch, it should contain a 3-by-3 seed in the center randomly taken from `SampleImage`, for constrained synthesis it should contain all the known pixels). `WindowSize`, the size of the neighborhood window, is the only user-settable parameter. The main portion of the algorithm is presented below. I have removed the part of the pseudocode that deals with Gaussian filtering, for simplicity.

```
function GrowImage(SampleImage, Image, WindowSize)
  while Image not filled do
    progress = 0
    PixelList = GetUnfilledNeighbors(Image)
    foreach Pixel in PixelList do
      Template = GetNeighborhoodWindow(Pixel)
      BestMatches = FindMatches(Template, SampleImage)
      BestMatch = RandomPick(BestMatches)
      Pixel.value = BestMatch.value
    end
  return Image
end
```

Function `GetUnfilledNeighbors()` returns a list of all unfilled pixels that have filled pixels as their neighbors (the image is subtracted from its morphological dilation). The list is randomly permuted and then sorted by decreasing number of filled neighbor pixels.

`GetNeighborhoodWindow()` returns a window of size `WindowSize` around a given pixel.

`RandomPick()` picks an element randomly from the list. `FindMatches()` is as follows:

```
function FindMatches(Template, SampleImage)
  ValidMask = 1s where Template is filled, 0s otherwise
  TotWeight = sum i,j ValidMask(i,j)
  for i,j do
    for ii,jj do
      dist = (Template(ii,jj)-SampleImage(i-ii,j-jj))^2
      SSD(i,j) = SSD(i,j) + dist*ValidMask(ii,jj)
    end
    SSD(i,j) = SSD(i,j) / TotWeight
  end
  PixelList = all pixels (i,j) where SSD(i,j) <=
min(SSD)*(1+ErrThreshold)
  return PixelList
end
```

In our implementation the constant were set as follows: `ErrThreshold = 0.1`. Pixel values are in the range of 0 to 1.

---