CPSC 425 Assignment 2 （due October 29, 2018）

Quan Zhang
48486154

3 “Texture Synthesis by Non-paramatric Sampling” is to synthesize texture in order to cover a region, with searching the input image for all such neighborhood to produce a histogram and pick the best match with SSD error. In Holefill.py, we run it by answer “Yes” to “Are you happy with this choice of fillRegion and textureIm?”

4 computeSSD computes the sum squared difference between an image patch and texture image. ShowPyramid(pyramid) Joins the images of image pyramid into a single horizontal image. It must ignore empty pixels that have a value of 1 in the given mask image.

For each columns in patch_Cols and each rows in patch_rows where TODOMask is not 0, we find patch array and texture array values. Since there are 3 color values, we separate the RGB SSD into 0, 1 and 2 color channels.

```
for patchRow in range(patch_rows):
        for patchCol in range(patch_cols):
                if TODOMask[patchRow][patchCol] == 0:
                        patchArray = TODOPatch[patchRow][patchCol]
                        textureArray = textureIm[r + patchRow][c + patchCol]
                        SSD[r][c] += ((patchArray[0] - textureArray[0] * 1.0) ** 2)
                        SSD[r][c] += ((patchArray[1] - textureArray[1] * 1.0) ** 2)
                        SSD[r][c] += ((patchArray[2] - textureArray[2] * 1.0) ** 2)
```

(implemented in Holefill.py as ComputeSSD)

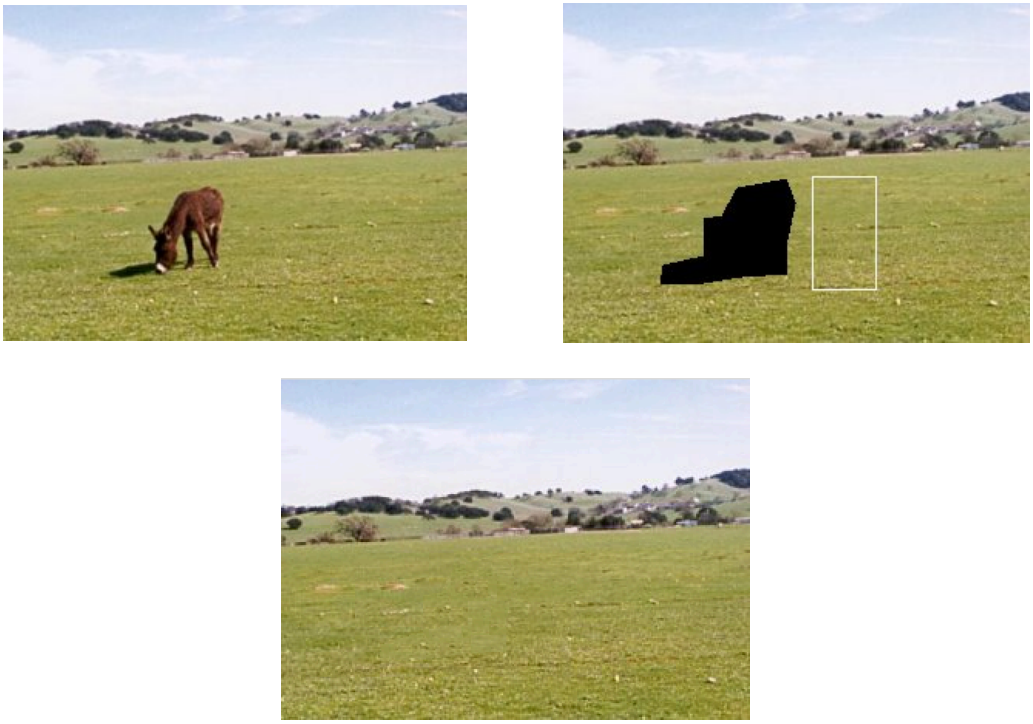
5 CopyPatch copies this selected patch into the final image.

Since there are 3 color values, we separate the RGB into 0, 1 and 2 color channels. (Note that this technique of copying a whole patch is much faster than copying just the center pixel as suggested in the original Efros and Leung paper. However, the results are not quite as good. We are also ignoring the use of a Gaussian weighted window as described in their paper.)

```
if TODOMask[i][j] == 1:
        holeLocation = imHole[iPatchCenter - patchL + i][jPatchCenter - patchL + j];
        textureLocation = textureIm[iMatchCenter - patchL + i][jMatchCenter - patchL + j];
        holeLocation[0] = textureLocation[0];
        holeLocation[1] = textureLocation[1];
        holeLocation[2] = textureLocation[2];
```

(implemented in Holefill.py as CopyPatch)

6 After implementing previous 2 method, we should get the original result, with donkey in the fill region in black and texture region in white box.



figure(6) Original Result

If we choose another texture region with polyselect.py, we could replace the donkey with another grass area.

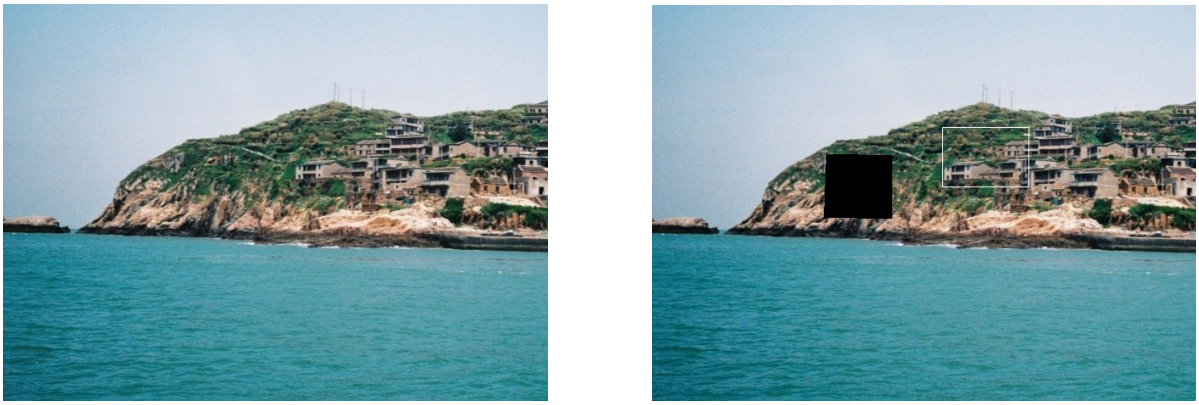
But if there are some different color in the texture too obvious, the “camouflage” for donkey is then not good.

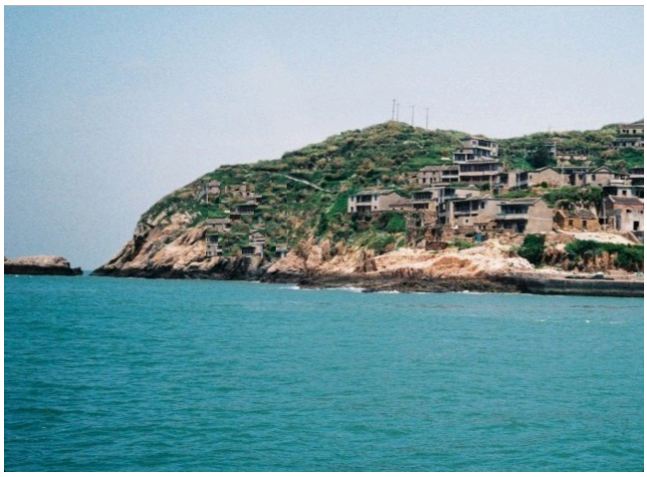
By changing randomPatchSD, we can change the standard deviation value, and thus pick different best matching patch from the patches. If this value is set to 0, then the optimal patch (minimum of the ssd image) is always chosen. If this value is large, then the patch choice will be more random. I experiment with different value from 0 to 1, to find images where it performs well and performs poor.

When randomPatchSD is close to 0 and the texture is continuous, like bricks, it performs well. Slightly better than randomPatchSD is close to 1.

figure(6) Perform Well

But when randomPatchSD is close to 1 and the texture is not continuous, for example like buildings, it performs poor. This is because the algorithm has difficulty recognize objects and meaning in the texture, like houses. But it can perform on simple textures like grass or bricks. If the houses get break down and gets thrown by a few bad patch selections, it becomes obviously strange in human eyes.





figure(6) Perform Poor

7 The randomPatchSD parameter affects the randomness in which the choice, from the set of the best texture patches.
The bigger the value of randomPatchSD, the more random it is at chooses, the less likely we are to choose the absolute best match. This randomness factor will help to add variation to our result when we encounter similar neighborhoods.
However, when randomPatchSD is set too low, the algorithm will always choose the best SSD match, resulting in a much greater chance that multiple instances of the same patch will appear in the result.

According to equation: patchSize = 2*patchL+1, patchL influences the size of the patch, which determines the size of the region filled at each step.
The bigger the patchSize is, the resulting patches standout more noticeably and the edges between patches may become more noticeable to human eyes.
However, when patchSize is too small, some patterns that don't fit in a patchSize window can be missed and the resulting image will fail to capture the overall texture in the texture region.