CPSC425: Assignment 1 (Python v2.7.15)

Alec Xu
38108130 (p7g9)

March 3, 2020

Question (5) Donkey Image

Figure 1: Donkey removal using random patch $\sigma = 1$, patch size = 21

Question (6) Other Images

Bad Image

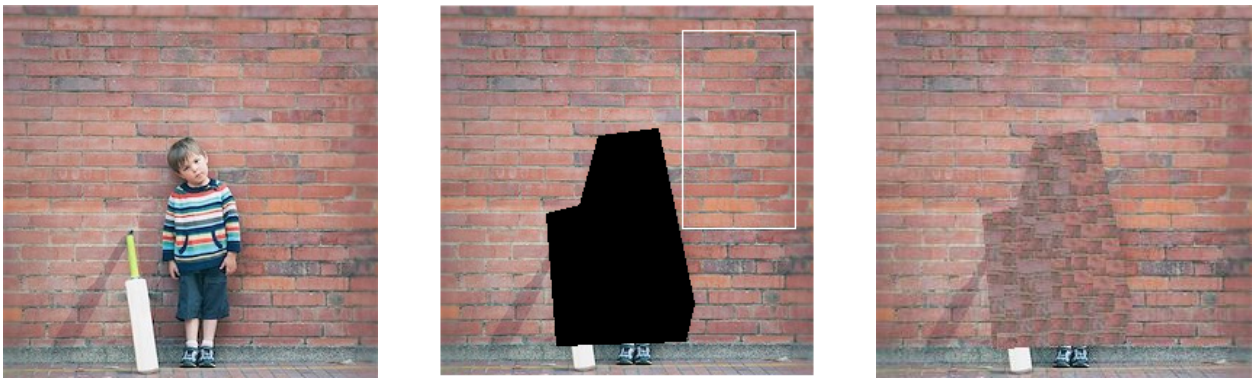

Figure 2: Kid removal using random patch $\sigma = 0$, patch size = 21

This method may have failed due to the patch size being too small for the size of the bricks. I also note that $\sigma = 0$, which makes the patch look unnatural as it lacks the

randomness of the real brick. Finally, the child cast a shadow, which may not have been fully patched out. This biased the algorithm to look for darker coloured brick as a result.

Good Image

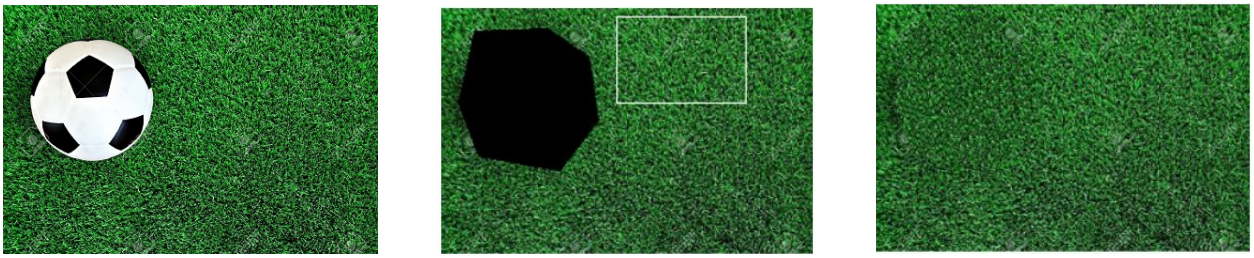

Figure 3: Soccer ball removal using random patch $\sigma = 5$, patch size = 11

Question (7) Explanations

For the soccer ball image, I reduced the patch size and increased σ. This removed repeat clumps of grass I saw at lower sigmas and bigger patches.

The σ value influences how random our patch selection is compared to strictly using the SSD ranking. For more naturally messy textures such as grass and sand, we should have a relatively high σ. For more ordered patterns such as brick and checkerboards, we should use a lower σ.

The patch size determines how big our patch is. I found that I achieved the best results when the patch size was at least the size of a texton.

Appendix A: Holefill.py

```
from PIL import Image, ImageDraw
import numpy as np
import random
import os.path
import pickle

##########################################################################
#                     Functions for you to complete                      #
##########################################################################
import random as rand
rand.seed(10)
def ComputeSSD(TODOPatch, TODOMask, textureIm, patchL):
   patch_rows, patch_cols, patch_bands = np.shape(TODOPatch)
   tex_rows, tex_cols, tex_bands = np.shape(textureIm)
   ssd_rows = tex_rows - 2 * patchL
   ssd_cols = tex_cols - 2 * patchL
   SSD = np.zeros((ssd_rows,ssd_cols))

   # Modifies TODOPatch to apply TODOMask, making all TODOMask=1 elements 0.
   # In this way, only Elements in TODOPatch with corresponding TODOMask=0 have value !=0
   # Also converts this modified TODOPatch to be 3 layers repeated to handle RGB
   TODOPatchMask = 1.0*np.multiply(TODOPatch, np.transpose(np.array([1-TODOMask,]*3)))

   for r in range(ssd_rows):
      for c in range(ssd_cols):

         # Compute sum square difference between textureIm and TODOPatch
         # for all pixels where TODOMask = 0, and store the result in SSD

         # Slice textureIm array according to patch size, and square difference elementwise with TODOPatchMask
         # Apply sum through the whole array. Assign to SSD
         # Multiply by 1.0 to conver uint8 -> float
         SSD[r,c] =
              np.sum(np.square(np.subtract(1.0*textureIm[r:r+2*patchL+1,c:c+2*patchL+1],1.0*TODOPatchMask)))
         pass
      pass
   return SSD

def CopyPatch(imHole,TODOMask,textureIm,iPatchCenter,jPatchCenter,iMatchCenter,jMatchCenter,patchL):
   patchSize = 2 * patchL + 1
   for i in range(patchSize):
      for j in range(patchSize):

         # Copy the selected patch selectPatch into the image containing
         # the hole imHole for each pixel where TODOMask = 1.
         # The patch is centred on iPatchCenter, jPatchCenter in the image imHole
         if TODOMask[i,j] == 1:
```

```
            imHole[iPatchCenter-patchL+i,jPatchCenter-patchL+j] =
                textureIm[iMatchCenter-patchL+i,jMatchCenter-patchL+j]
        pass
      pass
   return imHole

################################################################################
#                          Some helper functions                               #
################################################################################

def DrawBox(im,x1,y1,x2,y2):
   draw = ImageDraw.Draw(im)
   draw.line((x1,y1,x1,y2),fill="white",width=1)
   draw.line((x1,y1,x2,y1),fill="white",width=1)
   draw.line((x2,y2,x1,y2),fill="white",width=1)
   draw.line((x2,y2,x2,y1),fill="white",width=1)
   del draw
   return im

def Find_Edge(hole_mask):
   [cols, rows] = np.shape(hole_mask)
   edge_mask = np.zeros(np.shape(hole_mask))
   for y in range(rows):
      for x in range(cols):
         if (hole_mask[x,y] == 1):
            if (hole_mask[x-1,y] == 0 or
                  hole_mask[x+1,y] == 0 or
                  hole_mask[x,y-1] == 0 or
                  hole_mask[x,y+1] == 0):
               edge_mask[x,y] = 1
   return edge_mask

################################################################################
#                         Main script starts here                              #
################################################################################

#
# Constants
#

# Change patchL to change the patch size used (patch size is 2 *patchL + 1)
patchL = 10
patchSize = 2*patchL+1

# Standard deviation for random patch selection
randomPatchSD = 5

# Display results interactively
showResults = True
```

```
#
# Read input image
#

im = Image.open('soccer.jpg').convert('RGB')
im_array = np.asarray(im, dtype=np.uint8)
imRows, imCols, imBands = np.shape(im_array)

#
# Define hole and texture regions. This will use files fill_region.pkl and
#   texture_region.pkl, if both exist, otherwise user has to select the regions.
if os.path.isfile('fill_region.pkl') and os.path.isfile('texture_region.pkl'):
   fill_region_file = open('fill_region.pkl', 'rb')
   fillRegion = pickle.load( fill_region_file )
   fill_region_file.close()

   texture_region_file = open('texture_region.pkl', 'rb')
   textureRegion = pickle.load( texture_region_file )
   texture_region_file.close()
else:
   # ask the user to define the regions
   print "Specify the fill and texture regions using polyselect.py"
   exit()

#
# Get coordinates for hole and texture regions
#

fill_indices = fillRegion.nonzero()
nFill = len(fill_indices[0])            # number of pixels to be filled
iFillMax = max(fill_indices[0])
iFillMin = min(fill_indices[0])
jFillMax = max(fill_indices[1])
jFillMin = min(fill_indices[1])
assert((iFillMin >= patchL) and
      (iFillMax < imRows - patchL) and
      (jFillMin >= patchL) and
      (jFillMax < imCols - patchL)) , "Hole is too close to edge of image for this patch size"

texture_indices = textureRegion.nonzero()
iTextureMax = max(texture_indices[0])
iTextureMin = min(texture_indices[0])
jTextureMax = max(texture_indices[1])
jTextureMin = min(texture_indices[1])
textureIm  = im_array[iTextureMin:iTextureMax+1, jTextureMin:jTextureMax+1, :]
texImRows, texImCols, texImBands = np.shape(textureIm)
assert((texImRows > patchSize) and
      (texImCols > patchSize)) , "Texture image is smaller than patch size"

#
```

```
# Initialize imHole for texture synthesis (i.e., set fill pixels to 0)
#

imHole = im_array.copy()
imHole[fill_indices] = 0

#
# Is the user happy with fillRegion and textureIm?
#
if showResults == True:
   # original
   im.show()
   # convert to a PIL image, show fillRegion and draw a box around textureIm
   im1 = Image.fromarray(imHole).convert('RGB')
   im1 = DrawBox(im1,jTextureMin,iTextureMin,jTextureMax,iTextureMax)
   im1.show()
   print "Are you happy with this choice of fillRegion and textureIm?"
   Yes_or_No = False

   while not Yes_or_No:
      answer = raw_input("Yes or No: ")
      if answer == "Yes" or answer == "No":
         Yes_or_No = True
   assert answer == "Yes", "You must be happy. Please try again."

#
# Perform the hole filling
#

while (nFill > 0):
   print "Number of pixels remaining = " , nFill

   # Set TODORegion to pixels on the boundary of the current fillRegion
   TODORegion = Find_Edge(fillRegion)
   edge_pixels = TODORegion.nonzero()
   nTODO = len(edge_pixels[0])

   while(nTODO > 0):

      # Pick a random pixel from the TODORegion
      index = np.random.randint(0,nTODO)
      iPatchCenter = edge_pixels[0][index]
      jPatchCenter = edge_pixels[1][index]

      # Define the coordinates for the TODOPatch
      TODOPatch = imHole[iPatchCenter-patchL:iPatchCenter+patchL+1,jPatchCenter-patchL:jPatchCenter+patchL+1,:]
      TODOMask =
           fillRegion[iPatchCenter-patchL:iPatchCenter+patchL+1,jPatchCenter-patchL:jPatchCenter+patchL+1]

      #
```

```
        # Compute masked SSD of TODOPatch and textureIm
        #
        ssdIm = ComputeSSD(TODOPatch, TODOMask, textureIm, patchL)

        # Randomized selection of one of the best texture patches
        ssdIm1 = np.sort(np.copy(ssdIm),axis=None)
        ssdValue = ssdIm1[min(round(abs(random.gauss(0,randomPatchSD))),np.size(ssdIm1)-1)]
        ssdIndex = np.nonzero(ssdIm==ssdValue)
        iSelectCenter = ssdIndex[0][0]
        jSelectCenter = ssdIndex[1][0]

        # adjust i, j coordinates relative to textureIm
        iSelectCenter = iSelectCenter + patchL
        jSelectCenter = jSelectCenter + patchL
        selectPatch =
            textureIm[iSelectCenter-patchL:iSelectCenter+patchL+1,jSelectCenter-patchL:jSelectCenter+patchL+1,:]

        #
        # Copy patch into hole
        #
        imHole =
            CopyPatch(imHole,TODOMask,textureIm,iPatchCenter,jPatchCenter,iSelectCenter,jSelectCenter,patchL)

        # Update TODORegion and fillRegion by removing locations that overlapped the patch
        TODORegion[iPatchCenter-patchL:iPatchCenter+patchL+1,jPatchCenter-patchL:jPatchCenter+patchL+1] = 0
        fillRegion[iPatchCenter-patchL:iPatchCenter+patchL+1,jPatchCenter-patchL:jPatchCenter+patchL+1] = 0

        edge_pixels = TODORegion.nonzero()
        nTODO = len(edge_pixels[0])

    fill_indices = fillRegion.nonzero()
    nFill = len(fill_indices[0])

#
# Output results
#
if showResults == True:
    Image.fromarray(imHole).convert('RGB').show()
Image.fromarray(imHole).convert('RGB').save('results.jpg')
```