CPSC 425 Assignment 3

Jakob Khalil 86176674

March 6, 2024

Question 1 -

a. The assignment

I. (8 points) First, you need to complete the implementation of the ComputeSSD function that computes the sum squared difference (SSD) between an image patch and the texture image, for each possible location of the patch within the texture image. It must ignore empty pixels that have a value of 1 in the given mask image. Skeleton code for this function is provided in Holefill.py.

answer

Note: I'm only providing snippets of relevant functions in this PDF. Full code can be found in *.py and .ipynb files.

```
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))

    # Turn the uint8 images into floats to perform precise calculations
    # (synth patch is the patch we will synthesize AKA TODOpatch)
    synth_patch = np.copy(TODOPatch).astype('float64')
    sample_img = np.copy(textureIm).astype('float64')

    # Turn the image into greyscale
    synth_patch = np.mean(synth_patch, axis=2)
    sample_img = np.mean(sample_img, axis=2)

    # NOTE: I commented out the gaussian part because Q2 says we are ignoring
    # the Gaussian weighted window....
    # Create a gaussian distribution the same size as the patch for spatial
    # weighting later. (I'll use it for element-wise operation when computing SSD)
    sigma = patchL / 3
    gauss_1d = np.arange(-patchL, patchL + 1, dtype='float64')
    gauss_1d = np.vectorize(lambda x: np.exp(- (x ** 2) / (2 * sigma ** 2)))(gauss_1d)
    gauss_1d = gauss_1d[:, None] # I want this as a column vector
    gauss_2d = gauss_1d @ gauss_1d.T

    # Normalize so the sum is 1.
    gauss_normalized_2d = gauss_2d / np.sum(gauss_2d)

    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

            # Get column index of the patch borders in the sample image.
            left_col_sample_patch = c
            right_col_sample_patch = c + 2 * patchL

            # Get row index of the patch borders in the sample image.
            top_row_sample_patch = r
            bottom_row_sample_patch = r + 2 * patchL

            # Get all pixels within the patch centered at the SSD location (r, c)
            sample_patch = sample_img[top_row_sample_patch : bottom_row_sample_patch + 1,
                                      left_col_sample_patch : right_col_sample_patch + 1]

            # Remove unknown pixels from sample patch
```

```
        # (using inverted TODOmask because 1 represents missing pixels)
        sample_patch = sample_patch * (1.0 - TODOMask)

        # 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.
        # SSD[r, c] = np.sum(((sample_patch - synth_patch) ** 2) * gauss_normalized_2d)

        # Compute SSD.
        SSD[r, c] = np.sum((sample_patch - synth_patch) ** 2)

return SSD
```

II. (6 points) The next section of Holefill.py takes the SSD image created above and chooses randomly amongst the best matching patches to decide which patch to paste into the texture image at that point. Next you need to complete the implementation of the CopyPatch fucntion which copies this selected patch into the final image. Remember again that you should only copy pixel values into the hole section of the image, and the existing pixel values should not be overwritten. Skeleton code for CopyPatch is provided in Holefill.py, along with comments explaining the arguments.

Hand in a printed copy of the donkey image after texture synthesis has been used to remove the donkey. Note that the texture synthesis might take some time to fill in all the empty spaces.

answer

```
def CopyPatch(imHole,TODOMask,textureIm,iPatchCenter,jPatchCenter,iMatchCenter,jMatchCenter,patchL):
    # it looks like i is representing the row, and j represents the column in this instance...
    # Get row index of the patch borders for the synthetic patch
    top_row_synth_patch = iPatchCenter - patchL
    bottom_row_synth_patch = iPatchCenter + patchL

    # Get col index of the patch borders for the synthetic patch
    left_col_synth_patch = jPatchCenter - patchL
    right_col_synth_patch = jPatchCenter + patchL

    # get the pixels inside the synthetic patch
    synth_patch = imHole[top_row_synth_patch : bottom_row_synth_patch + 1,
                         left_col_synth_patch : right_col_synth_patch + 1]

    # Get row index of the patch borders for the selected patch
    top_row_selected_patch = iMatchCenter - patchL
    bottom_row_selected_patch = iMatchCenter + patchL

    # Get col index of the patch borders for the selected patch
    left_col_selected_patch = jMatchCenter - patchL
    right_col_selected_patch = jMatchCenter + patchL

    # get the pixels inside the selected patch
    selected_patch = textureIm[top_row_selected_patch : bottom_row_selected_patch + 1,
                               left_col_selected_patch : right_col_selected_patch + 1]

    # Merge the known pixels from synth_patch with the unkown pixels
    # taken from selected_patch
    synth_patch = synth_patch + selected_patch * TODOMask[:,:,None]

    # Copy the selected patch selectPatch into the image containing
    # the hole imHole for each pixel where TODOMask = 1.
    # fill in imHole with the merged patch
    imHole[top_row_synth_patch : bottom_row_synth_patch + 1,
           left_col_synth_patch : right_col_synth_patch + 1] = synth_patch
    return imHole
```

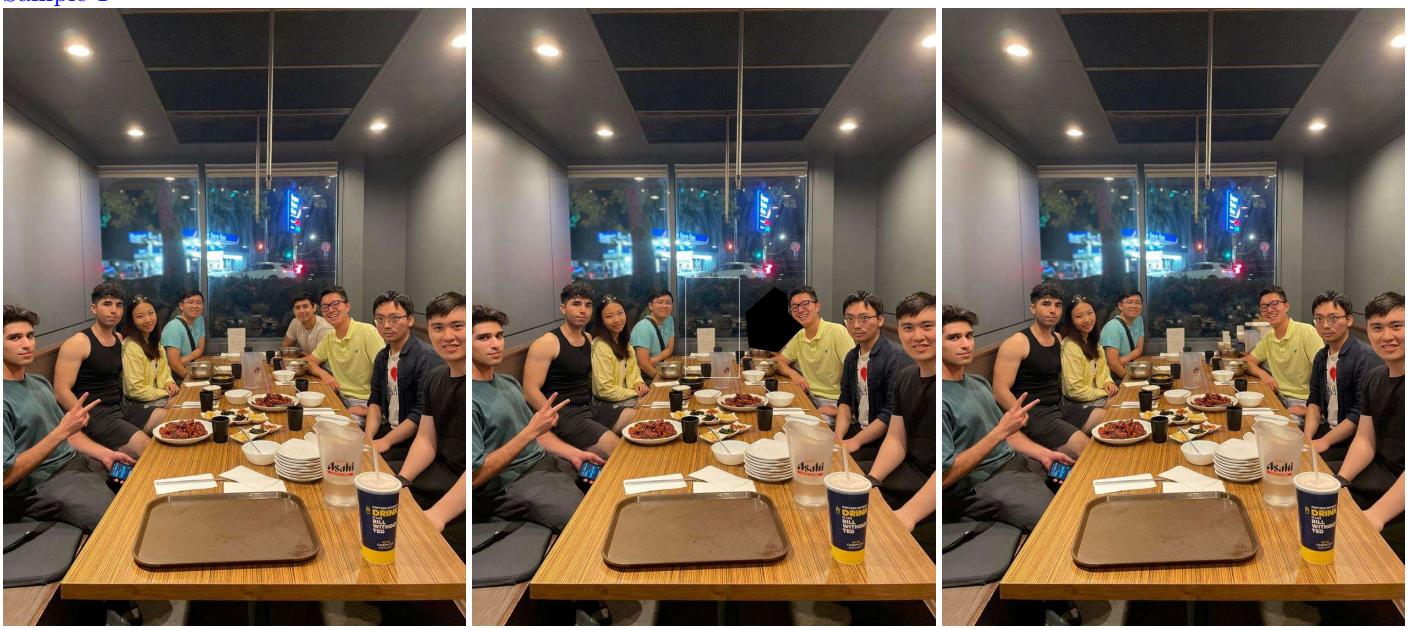
III. (5 points) Try running this texture synthesis method on some new images of your choosing. You will need to indicate the area of the removed region in the code. You can load your own image by altering the line that reads donkey.jpg in Holefill.py. You will need to select small regions to avoid long run times.

Hand in texture synthesis results for 2 new images, where one shows the algorithm performing well and the other shows it performing poorly. For each example, show both the original image and the modified one. You do not need to print in colour. Briefly describe why the method failed in the case in which it performed poorly.

answer

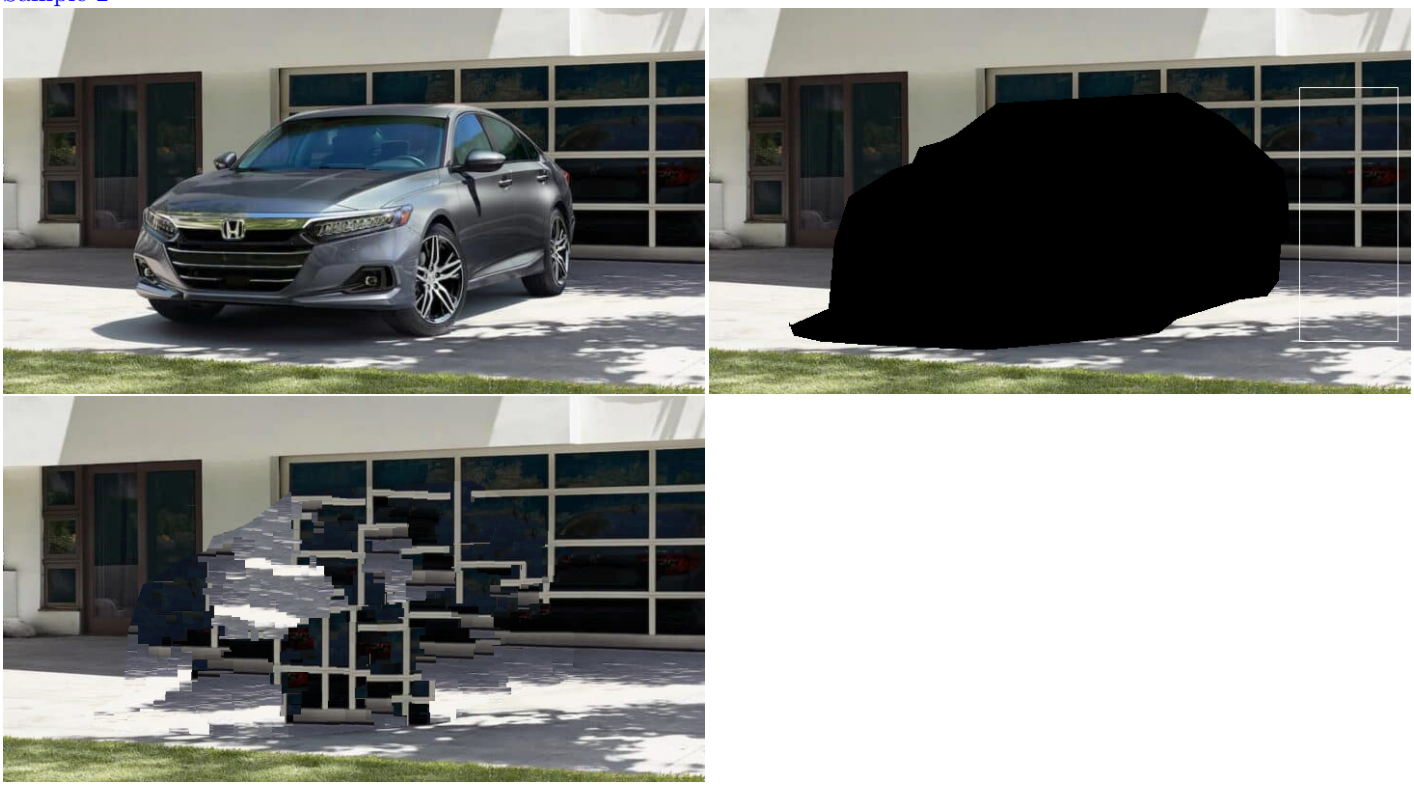
Sample 2 failed because the background has edges and cannot be expressed as a single uniform texture. Our method struggle to stitch textures at the edges properly. Furthermore, the fact that we are filling chunks of pixels rather than individual pixels to save runtime ends up resulting in poor transitions between patches within the filled area. This effect is magnified by the fact that we are using a relatively small window-size.

Sample 1



Sample 2's results are on the next page.

Sample 2



IV. (6 points) Provide an explanation for the effects of the randomPatchSD and patchL parameters. What results can be expected if these values are too small or too large, and why do these results happen?

answer

Variable 'randomPatchSD' controls the standard deviation of our Gaussian RNG function used to select a patch stochastically based on their SSD scores. When the standard deviation is too small, our RNG function will select the patch with the best (or occasionally second best) SSD score nearly every time. As we increase the standard deviation to be very large, our RNG function approaches a uniform distribution, thus randomly selecting a patch with roughly equal probability.

Variable 'patchL' controls the window size of pixels considered when observing the border points of the TODO regions (and consequently by the way the assignment was implements, it is also heuristic for the size of each synthetic regions that gets copied and stitched from our template patch to the image hole). If 'patchL' is too large, but there is a good matching patch for the window-size, we will get a more seamless result that captures more complex patterns since we take a larger neighbourhood into consideration when computing SSD. HOWEVER, a larger window-size also reduces the odds of finding a match, making it less flexible, which and may result in poor synthesized results if a good match is not found. If it is too small, we end up with many matches that don't take a potential larger more complex underlying pattern into consideration which may lead to an inconsistent synthetic pattern accompanied by poor transitions.