Khoa Do
ECE 558-01

November 05, 2021
Dr. Tianfu Wu

Project 2 – Report

Project 2 is to perform image blending from using Gaussian/Laplacian pyramid computation technique **gPyr, lPyr = ComputePyr(input_image, num_layers)** where **input_image** is an input image in grayscale or RGG and **num_layers** is the number of Gaussian/Laplacian pyramid layers to be computed. As a part of implementing the Gaussian/Laplacian pyramid, a smoothing function, a down sampling function, and a up sampling function need to be implemented as well although OpenCV built-in functions can be used as well. For the smoothing function, built-in functions can be used to generate the Gaussian kernel but the convolution/FFT function from project 1 must be used. A simple GUI is also required to create a black and white binary mask image. The GUI can open the foreground image to select a region of interest (ROI) in rectangular shape, elliptical shape, or polygon (free form) region. As a result, three blended images are generated from three image pairs of two, total of six images (pair A, pair B, and pair C).

The entire code for this project is broken up into three files: **main.py**, **P2_Packages.py**, and **P1_q1.py**. **Main.py** contains the main code to load input images and display several images including foreground image with marked mask, mask, background image, and background image. It also contains the GUI to draw and create the mask from the foreground image. **P2_Pakages.py** contains functions to align images, create mask pyramid, downscale/upscale images, and blend images. This file is used as a module that the main file imports the above functions. **P1_q1.py** contains the convolution and the padding function implemented in project 1. This file is used as a module that the P2_Packages file imports its functions.

```python
####################---MAIN CODE---####################

""" set up vars for drawing mask """
mouse_pressed = False                                   # true if mouse is pressed
rec_draw = True                                         # if True, draw rectangle, press 'e' to toggle to draw ellipse
init_x,init_y = -1,-1                                   # initial x-y position of mouse


""" read images """
f_img = cv2.resize(cv2.imread('lc1.jpg'), (940,620))              # foreground img as RGB, optional: grasycale
b_img = cv2.resize(cv2.imread('lc2.jpg'), (940,620))              # background img as RGB, optional: grayscale


""" images aligning """
if (b_img.shape[0] > f_img.shape[0]):                   # if b_img's no. rows > f_img's no. row
    f_img_align = align(f_img,b_img,50,35)              # row & col can be adjusted manually
    f_img = np.copy(f_img_align)
    f_imgcp = np.copy(f_img_align)                      # copy of aligned_img for mask
else:
    f_imgcp = np.copy(f_img)


""" GUI to draw mask and display mask """
cv2.namedWindow('Foreground Image', cv2.WINDOW_NORMAL)
cv2.setMouseCallback('Foreground Image', get_mask)      # call get_mask function to draw and define shape

while(1):
    cv2.imshow('Foreground Image',f_imgcp)
    key = cv2.waitKey(1) & 0xFF

    if key == ord('e'):                                 # press 'e' to toggle to ellipse
        rec_draw = not rec_draw                         # then draw rectangle set to False and draw ellipse
    elif key == 27:                                     # press 'ECS' to exit window after drawing
        cv2.destroyAllWindows()
        break


mask = create_mask_fimg(f_imgcp, init_x, init_y, new_x, new_y)      # create mask from aligned f_img
                                                                    # new_x, new_y are global in get_mask
```

Figure 1. Code Snip 1, Main.py.

Figure 1 shows some initial setups for the GUI's logics to draw the mask. The original x-y coordinate of the mouse is set to (-1, -1). Once the user starts drawing by pressing left button, the new starting x-y coordinate of the mouse on the image is assigned to the original x-y coordinate (line 59-75, figure 2). Two images are read as grayscale, RGB, or both. The images are resized to be less than 1000 pixels in either width or length, typically about 500x500 in size for pair A and pair B, for faster processing and for the convolution function to work properly with a padding width of two. Pair C is about 940x620. Most tested images are resized to be square images and are even in both width and length (both foreground and background images) before being blended, but the code should work for non-square images (for example, pair C). The foreground image is manually aligned if needed with the background image if its size is smaller than the background image's.

```
34  """ function to create mask from aligned f_img """
35  def create_mask_fimg(f_imgcp, init_x, init_y, new_x, new_y):
36  
37      new_mask = np.zeros(f_imgcp.shape).astype(np.float32)
38  
39      """ OpenCV function to draw rectangle and ellipse """
40      if rec_draw == False:
41          new_mask = cv2.ellipse(new_mask, ((init_x+(new_x-init_x)//2),(init_y+(new_y-init_y)//2)), (new_x-init_x,new_y-init_y)
42      else:
43          new_mask = cv2.rectangle(new_mask, (init_x,init_y),(new_x,new_y), (1,1,1), -1)
44  
45      return new_mask                           # return new_mask to main code
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  """ function to draw mask on displayed GUI """
57  def get_mask(event,x,y,f,param):                  # openCV defaut parameters
58  
59      global init_x, init_y, new_x, new_y, mouse_pressed  # these vars can be used outside of this function
60  
61      if event == cv2.EVENT_LBUTTONDOWN:            # left mouse button is pressed
62          mouse_pressed = True
63          init_x,init_y = x,y                       # current starting x-y mouse position assigned to initial position
64  
65      elif event == cv2.EVENT_LBUTTONUP:            # left mouse button is released
66          mouse_pressed = False
67  
68          if rec_draw == False:
69              cv2.ellipse(f_imgcp, ((init_x+(x-init_x)//2),(init_y+(y-init_y)//2)), (x-init_x,y-init_y), 0,0,360, (0,0,0), 1)
70          else:
71              cv2.rectangle(f_imgcp, (init_x,init_y), (x,y), (0,0,0), 1)      #show drawn shape
72  
73          # ending x-y mouse position
74          new_x = x;
75          new_y = y;
```

Figure 2. Code Snip 2, Main.py.

Get_mask and **create_mask_fimg** are called from the main code with in **main.py** to display the GUI for drawing the mask's shape (or ROI) and create the mask from ROI. In **get_mask**, after the user release the mouse [left] button, the ending x-y coordinate is assigned to new_x and new_y. Then, init_x (see figure 1), init_y (line 91, figure 1), new_x, and new_y are fed to **create_mask_fimg**. Using OpenCV's built-in functions, a mask is created in either rectangular shape or elliptical shape. The user has the option to press "e" to toggle to ellipse from the default rectangle.

```
126 """ display f_img, b_img, and mask """
127 cv2.namedWindow('Foreground Image', cv2.WINDOW_NORMAL)
128 cv2.imshow('Foreground Image',f_img)
129 cv2.namedWindow('Mask', cv2.WINDOW_NORMAL)
130 cv2.imshow('Mask', mask)
131 cv2.namedWindow('Background Image', cv2.WINDOW_NORMAL)
132 cv2.imshow('Background Image', b_img)
133 cv2.waitKey(0)                                        # after dislaying, press any key to close windows
134 cv2.destroyAllWindows()
135
136
137 """ compute Gaussian/Laplacian pyramid, create Gaussian mask, and blend images """
138 gPyr_fimg, lPyr_fimg = ComputePyr(f_img,10)
139 gPyr_bimg, lPyr_bimg = ComputePyr(b_img,10)
140 gPyr_mask = create_gPyr_mask(mask,len(gPyr_fimg))
141 blended_img = imgBlending(lPyr_fimg, lPyr_bimg, gPyr_mask, len(gPyr_fimg))
142
143
144 """ testing - print to see images """
145 #for i in range(len(gPyr_fimg)):
146     #cv2.imwrite('gPyr_fimg{i}.png'.format(i=i), gPyr_fimg[i])
147     #cv2.imwrite('lPyr_fimg{i}.png'.format(i=i), lPyr_fimg[i])
148     #cv2.imwrite('gPyr_bimg{i}.png'.format(i=i), gPyr_bimg[i])
149     #cv2.imwrite('lPyr_bimg{i}.png'.format(i=i), lPyr_bimg[i])
150     #cv2.imwrite('gPyr_mask{i}.png'.format(i=i), gPyr_mask[i])
151
152
153 """ display f_imgcp, b_img, mask, and blended_image"""
154 cv2.namedWindow('Foreground Image', cv2.WINDOW_NORMAL)
155 cv2.imshow('Foreground Image',f_imgcp)
156 cv2.namedWindow('Mask', cv2.WINDOW_NORMAL)
157 cv2.imshow('Mask',mask)
158 cv2.namedWindow('Background Image', cv2.WINDOW_NORMAL)
159 cv2.imshow('Background Image',b_img)
160 cv2.namedWindow('Blended Image', cv2.WINDOW_NORMAL)
161 cv2.imshow('Blended Image',blended_img)
162 cv2.waitKey(0)
163 cv2.destroyAllWindows()
164
165 """ write result images for report """
166 cv2.imwrite('f_imgcp.png', f_imgcp)
167 cv2.imwrite('b_img.png', b_img)
168 cv2.imwrite('bl_img.png',blended_img)
169
170 ####################---End of MAIN CODE ---####################
```

Figure 3. Code Snip 3, Main.py.

After being pre-processed, the two input images are sent to **P2_Packges.py** to compute their Gaussian/Laplacian pyramids by calling **ComputePyr**. The recently created mask is also sent to compute its pyramid. The number of layers (line 138 and 139, figure 3) are entered manually, in this case is 10 layers. However, the actual possible (maximum and minimum) layers computed can be less than the desired number of layers entered depending on the images' sizes and the kernel's size respectively (line 151-158, figure 4).

```
143 """ function compute Gaussian/Laplacian pyramid """
144 def ComputePyr(img,num_layers):
145
146     global pyr_layers, w                            # minimum layers should be 1 and w is Gaussian kernel
147
148     pyr_layers = 1
149     shape = img.shape[0]                            # since we're using square images, use either shape is alright
150
151     for i in range(num_layers-1):
152         if (shape//2 < 5):                          # floor division, cannot be less than kernel size 5x5
153             print("Maximum no. of Layers computed from the image is %d " %pyr_layers)
154             num_layers = pyr_layers                 # num_layers = last increment of pyr_layers in else statement
155             break
156         else:
157             shape = shape // 2                      # as long as shape is floored by two and > 5, increment no. layers in the
158             pyr_layers += 1
159
160     """ Gaussian kernel """
161     g_kernel = cv2.getGaussianKernel(5,2)           # 1-d Gaussian array, std deviation is 2
162     w = g_kernel*(g_kernel.T)                       # create 2-d gaussian filter
163
164     """ Gaussian pyramid """
165     g = img.copy().astype(np.float32)
166     g_pyr_img = [g]                                 # first layer of the Gaussian pyr = original img
167     for i in range(num_layers-1):
168         g = downSampler(conv2(g,w))                 # pyramid of smaller images - call downSampler, convolve then downscale
169         g_pyr_img.append(g)                         # add layers to list of Gaussian images
170
171     """ Laplacian pyramid """
172     l_pyr_img = [g_pyr_img[num_layers-1]]           # first layer of Laplacian pyr = last layer in Gaussian list of images
173     for i in range(num_layers-1,0,-1):
174         G = conv2(upSampler(g_pyr_img[i],g_pyr_img[i-1]),w)        # pyramid of bigger images - call upSampler, upscale then
175         l = np.subtract(g_pyr_img[i-1], G)
176         l_pyr_img.append(l)                         # add layers to list of Laplacian images
177
178     return g_pyr_img, l_pyr_img                     # return to main code
```

Figure 4. Code Snip 4, P2_Packages.py.

A Gaussian kernel is obtained by generating a 1-D Gaussian array multiplied by its own transpose using **cv2.getGaussianKernel**. Line 160-176 in figure 4 shows the logics in computing the Gaussian and Laplacian pyramid following the diagram shown in figure 5.

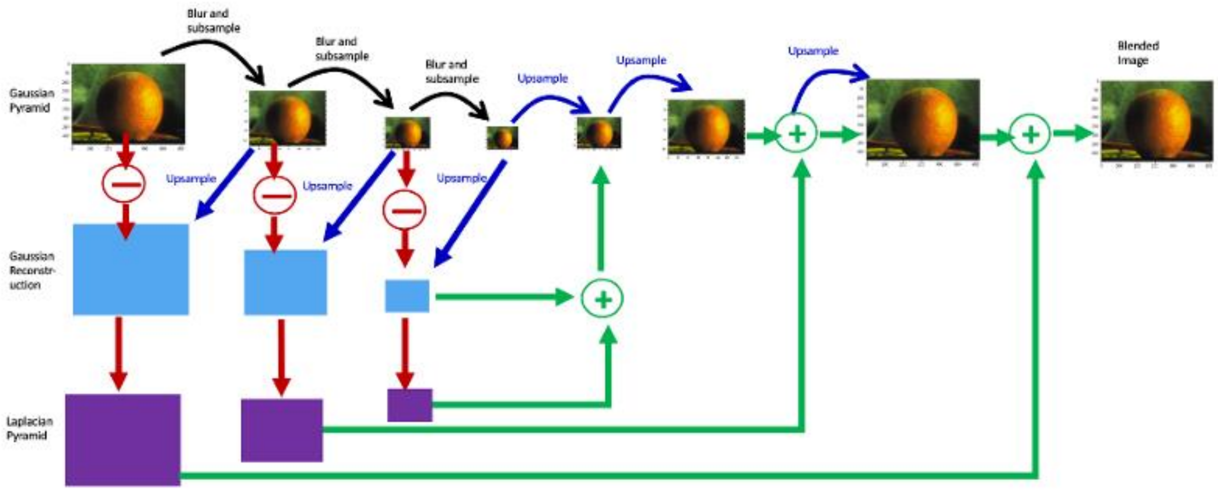

Figure 5. Gaussian/Laplacian Pyramid Logic [1].

The up-sampling and down-sampling function are implemented by referring to the built-in OpenCV functions. These functions are called upSampler and downSampler and are shown in figure 6.

```
""" function to upscale img using nearest neighbor interpolation """
def upSampler(img,img_m1):

    ogRow = img.shape[0]
    ogCol = img.shape[1]

    newRow = img_m1.shape[0]
    newCol = img_m1.shape[1]

    rRatio = newRow / ogRow
    cRatio = newCol / ogCol

    fRow = (np.floor((np.arange(0,newRow,1)) / rRatio)).astype(np.int16)
    fCol = (np.floor((np.arange(0,newCol,1)) / cRatio)).astype(np.int16)

    uS_img = img[fRow , :]
    uS_img = uS_img[: , fCol]

    return uS_img                                   # return uS_img to pyr_up










""" function to downscale img using nearest neighbor interpolation """
def downSampler(img):

    ogRow = img.shape[0]
    ogCol = img.shape[1]

    newRow = (np.floor(img.shape[0] / 2)).astype(np.int16)
    newCol = (np.floor(img.shape[1] / 2)).astype(np.int16)

    rRatio = newRow / ogRow
    cRatio = newCol / ogCol

    fRow = (np.floor((np.arange(0,newRow,1)) / rRatio)).astype(np.int16)
    fCol = (np.floor((np.arange(0,newCol,1)) / cRatio)).astype(np.int16)

    dS_img = img[fRow , :]
    dS_img = dS_img[: , fCol]

    return dS_img                                   # return dS_img to pyr_down
```

Figure 6. Code Snip 5, P2_Packages.py.

The 2-D convolution function used in this project has been implemented in project 1 since the computation is performed in the spatial domain. Though the computation in the spatial domain is not as fast as in the frequency domain, it works well as expected. “Reflect-across-edge” padding type is used for all tested images in this project. By the end of **ComputePyr**, two pyramids are obtained each for the foreground image and the background image: the Gaussian pyramid that contains a list of bigger images to smaller images with the first image being the original image and the Laplacian pyramid that contains a list of smaller images to bigger images with the first image is the last image in the Gaussian pyramid.

In the meantime, the created mask in line 123 of figure 1 is also sent to **create_gPyr_mask** to compute its Gaussian pyramid list of images (bigger to smaller). The list is then reversed to smaller images to bigger images, so that the list can be applied to the Laplacian list of images for blending images.

```
""" function to blend images """
def imgBlending(l_fimg, l_bimg, gMask, layers):

    """ blend images """
    LS = []
    for la,lb,mask in zip(l_fimg,l_bimg,gMask):
        ls = la * mask + lb * (1 - mask)
        LS.append(np.float32(ls))

    """ reconstruct final image """
    lap_bl = LS[0]
    for i in range(1,layers):
        lap_bl = conv2(upSampler(lap_bl,LS[i]),w)
        lap_bl = cv2.add(lap_bl,LS[i])

    final = np.clip(lap_bl,0,255).astype(np.uint8)

    return final                                # return final blended image to Main Code











""" function to create Gaussian pyramid for mask """
def create_gPyr_mask(mask,num_layers):

    gMask = np.copy(mask).astype(np.float32)
    g_pyr_mask = [gMask]                        # create list of masks

    for i in range(num_layers-1):
        gMask = downSampler(conv2(gMask,w))     # pyramid of smaller masks - call downSampler, convolve then downscale
        g_pyr_mask.append(gMask)                # add layers to list of Gaussian masks

    g_pyr_mask.reverse()                        # reverse Gaussian mask list from smaller to bigger to apply to Laplacia

    return g_pyr_mask
```

Figure 7. Code Snip 6, P2_Packages.py.

Upon obtaining the Laplacian pyramids of the foreground image, background image, and the reversed Gaussian pyramid of the mask, the three pyramids are sent to **imgBlending** to blend and reconstruct the final blended image, which is returned to the main code for displaying. Three pairs of two images are blended as required and shown below.

Figure 8. Blended Image from Pair A.

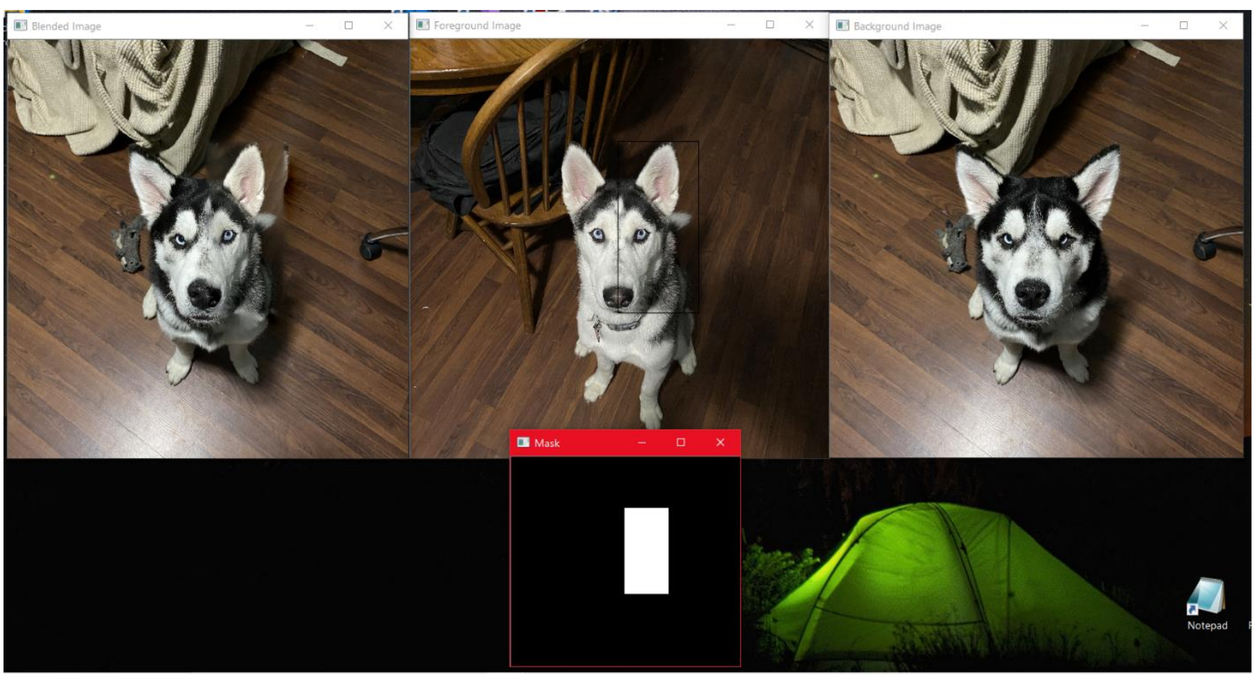

Figure 9. Pair A's Foreground and Background Image.

Figure 10. Blended Image from Pair B.

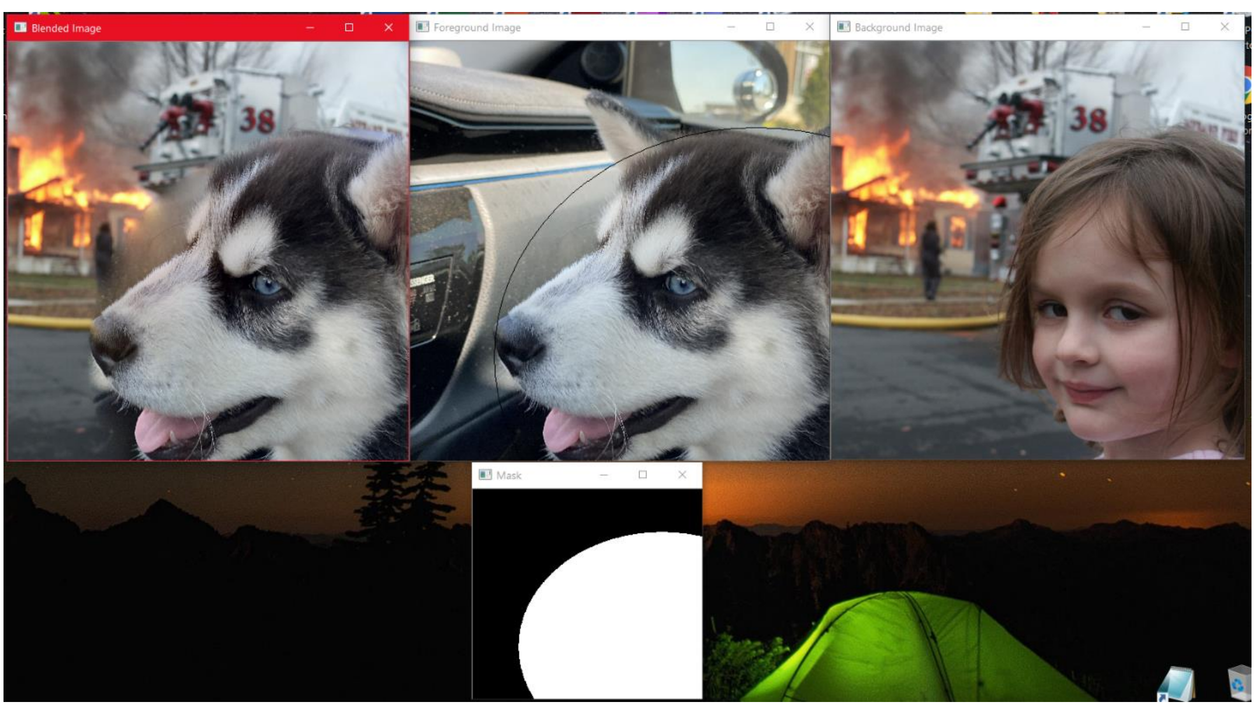

Figure 11. Pair B's Foreground and Background Image.

Figure 12. Blended Image from Pair C.

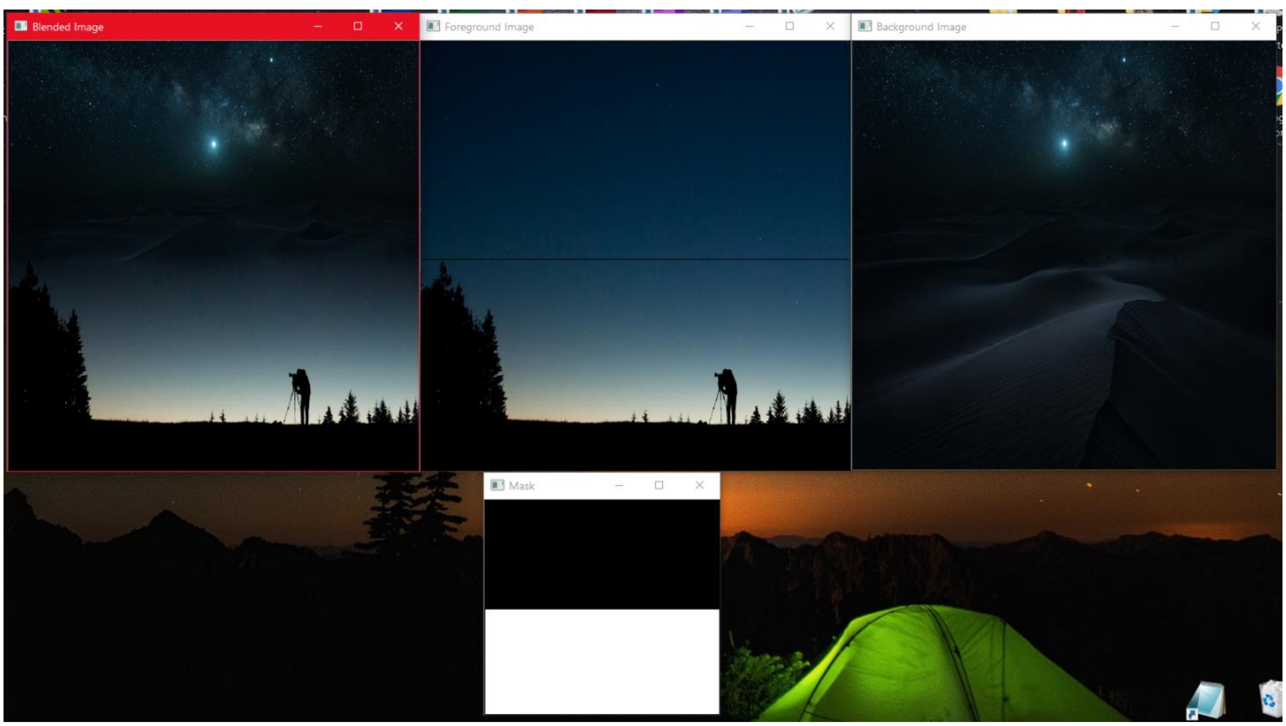

Figure 13. Pair C's Foreground and Background Image.

Attached to this report:

1) kddo_code

- main.py
- P1_q1.py
- P2_Packages.py

2) kddo_images

Pair A

- Original Images
  - bella,jpg
  - lucas.jpg
- Result Images
  - b_img.png
  - bl_img.png
  - f_imgcp.png
  - sum1.png

Pair B

- Original Images
  - dgirl,jpg
  - lucasface.jpg
- Result Images
  - b_img.png
  - bl_img.png
  - f_imgcp.png
  - sum2.png

Pair C

- Original Images
  - lc1,jpg
  - lc2.jpg
- Result Images
  - b_img.png
  - bl_img.png
  - f_imgcp.png
  - sum3.png

Reference

[1] M. Zhao, “Image Blending Using Laplacian Pyramids,” Becominghuman.ai, 14-May-2020. [Online]. Available: https://becominghuman.ai/image-blending-using-laplacian-pyramids-2f8e9982077f. [Accessed: 28-Oct-2021].
[2] F. Projcheski, “How to Blend Images Using Gaussian and Laplacian Pyramid,” Laconicml.com, 25-Jul-2020. [Online]. Available: https://laconicml.com/blend-images-laplacian-pyramid/. [Accessed: 28-Oct-2021].
[3] I. Nader, “Image Blending Using Pyramids in OpenCV Python,” Morioh.com, Jun-2021. [Online]. Available: https://morioh.com/p/1e6a4d2d950c. [Accessed: 28-Oct-2021].
[4] Kang and Atul, “Image Pyramids OpenCV Python,” Theailearner.com, 19-Aug-2019. [Online]. Available: https://theailearner.com/tag/image-pyramids-opencv-python/. [Accessed: 28-Oct-2021].
[5] https://github.com/PadovaY/pyramid_blend.
[6] https://github.com/jagracar/OpenCV-python-tests.