Final Project: Texture Transferring with Multiple Sources

CS445: Computational Photography - Spring 2022

Jack Chen (zihengc2), Ruisi Liu (ruisil3), Xiaosen Wang (xiaosen2)

In [1]:
```
# jupyter extension that allows reloading functions from imports without clearing kernel :D
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

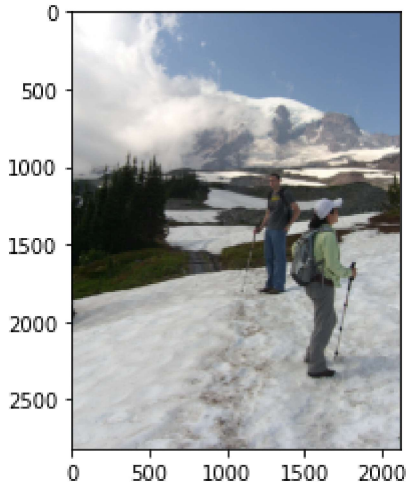
In [2]:
```
# System imports
from os import path
import math

# Third-Party Imports
import cv2
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import griddata
import random
import glob

# Helper Imports
from utils import *
```

In [3]:
```
# modify to where you store your project data including utils
# datadir = "/content/drive/My Drive/proj4/"
datadir = ''

samplesfn = datadir + "samples"
# !cp -r "$samplesfn" .
```

Segmentation using K-means

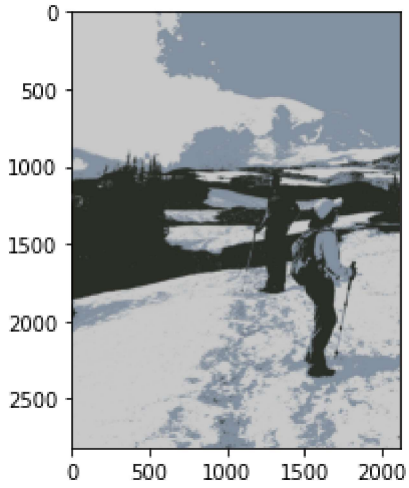
In [4]:
```
img = cv2.imread('samples/im1.JPG', cv2.COLOR_BGR2RGB)
width = img.shape[0]
height = img.shape[1]
Z = img.reshape((-1,3))
Z = np.float32(Z)  # convert to np.float32
plt.imshow(img[:, :, [2, 1, 0]])
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x20c992cf580>
```



In [5]:
```
# define criteria, number of clusters(K) and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 3
ret, label, center = cv2.kmeans(Z,K,None,criteria,10,cv2.KMEANS_RANDOM_CENTERS)
label = label.flatten()
# Now convert back into uint8, and make original image
center = np.uint8(center)
res = center[label.flatten()]
res2 = res.reshape((img.shape))
label.resize(width, height)

# check outputs
print(label.shape)
print(center.shape)
print(center)
plt.figure()
plt.imshow(res2[:, :, [2, 1, 0]])
plt.show()
```

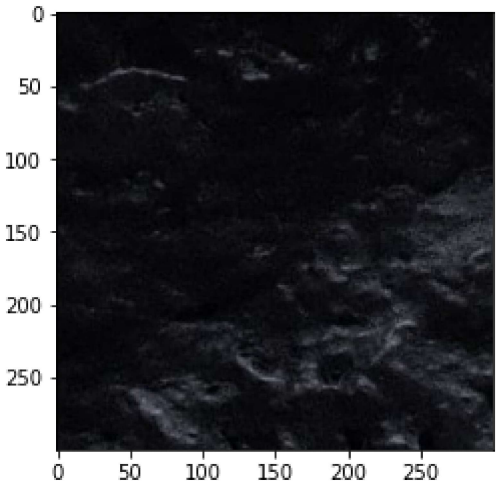
```
(2816, 2112)
(3, 3)
[[200 201 201]
 [ 38  45  42]
 [159 144 131]]
```

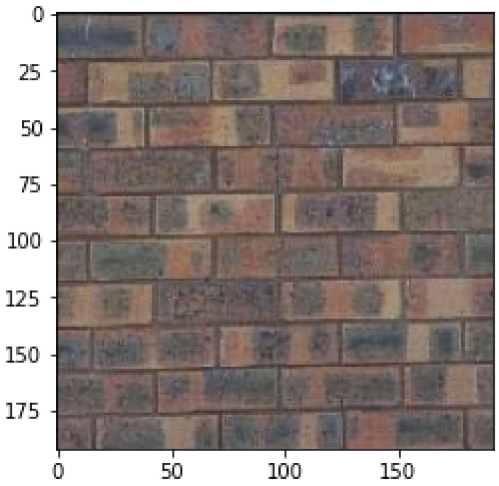


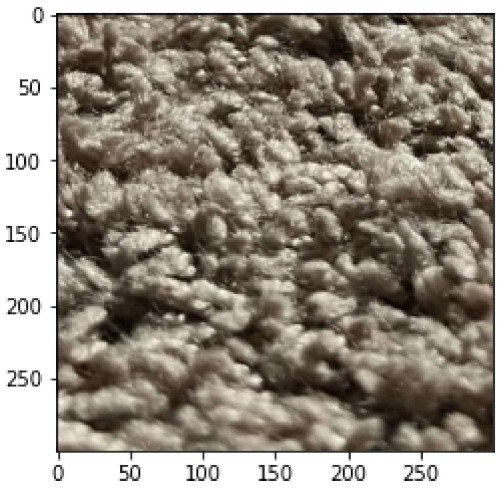
Selecting best-fitting textures

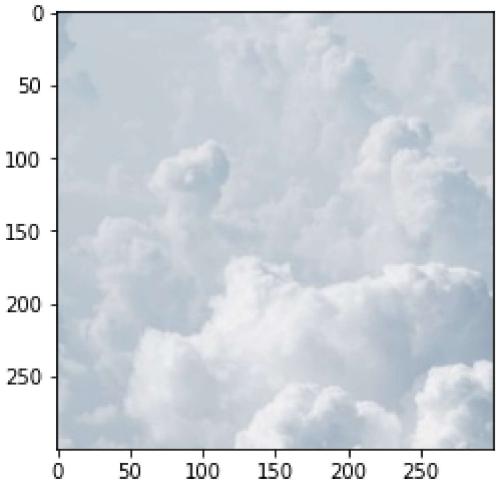
In [6]:
```
# Reading textures
textures = [cv2.imread(file) for file in glob.glob("textures/*.jpg")]
for texture in textures:
```

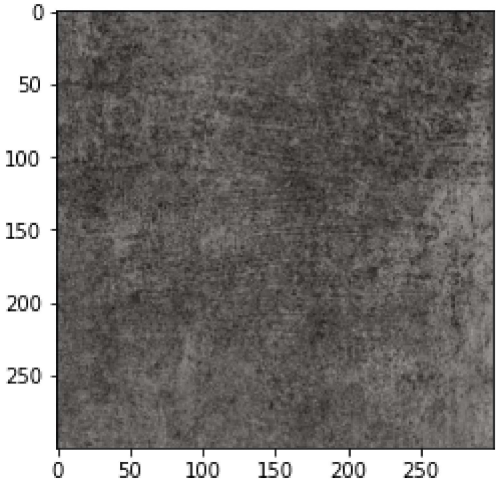
```
    plt.imshow(cv2.cvtColor(texture, cv2.COLOR_BGR2RGB))
    plt.show()
```

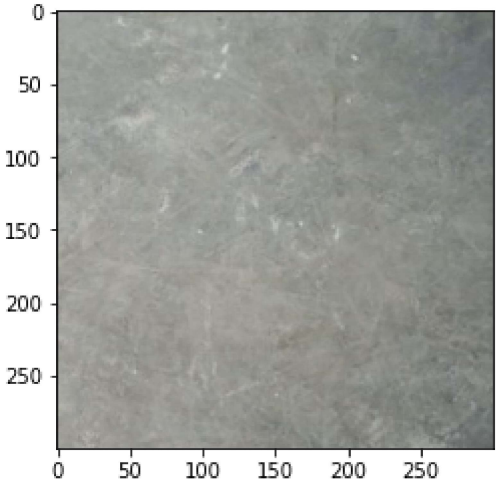


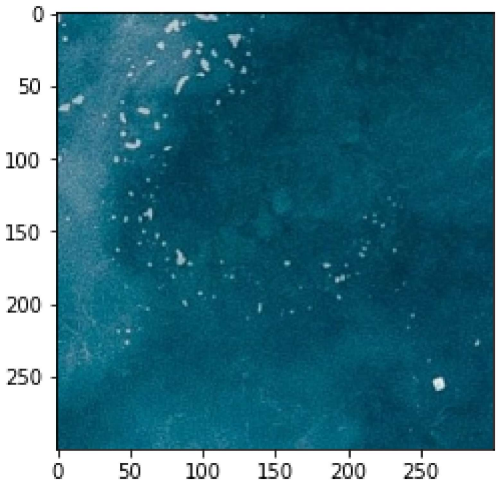


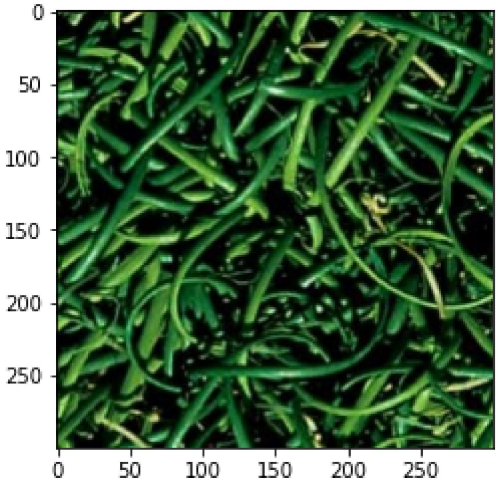


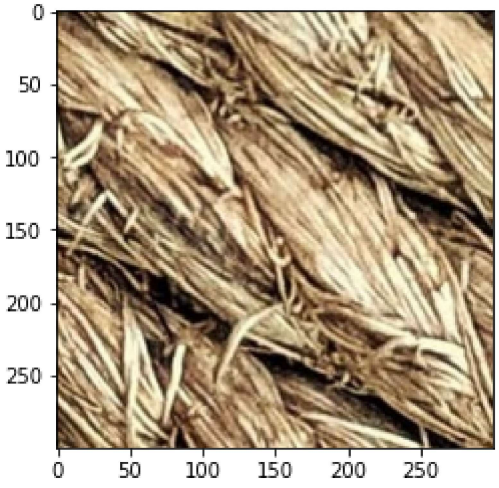

0
50
100
150
200
250
0
50
100
150
200
250

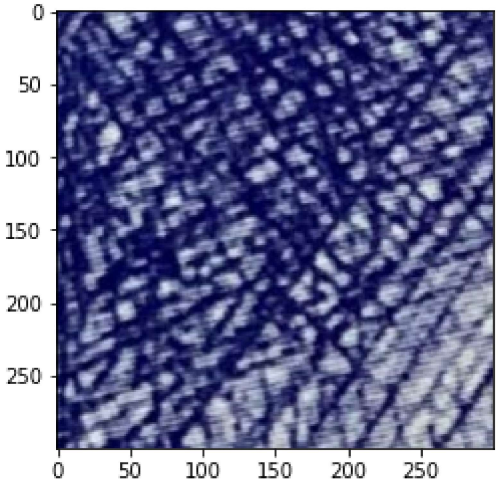

0
50
100
150
200
250
0
50
100
150
200
250

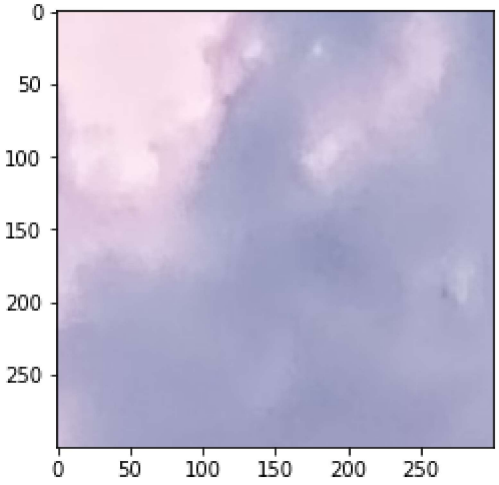

0
50
100
150
200
250
0
50
100
150
200
250

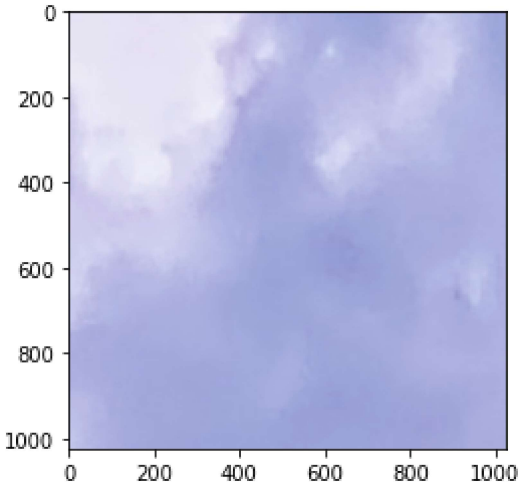

0
50
100
150
200
250
0
50
100
150
200
250

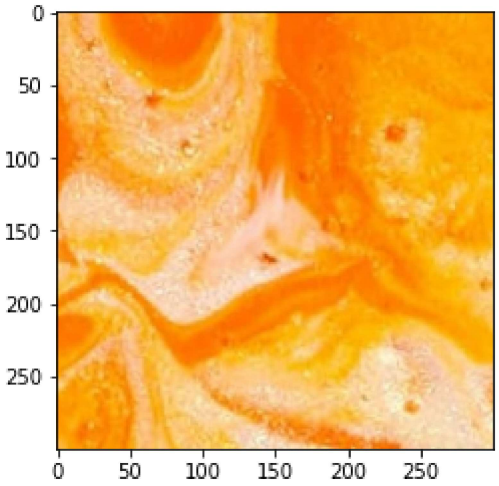

0
50
100
150
200
250
0
50
100
150
200
250

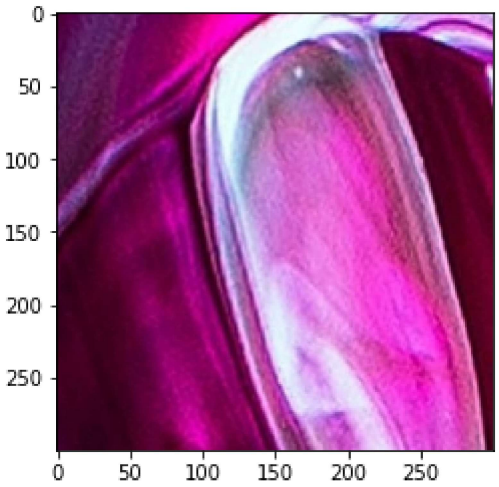

0
50
100
150
200
250
0
50
100
150
200
250

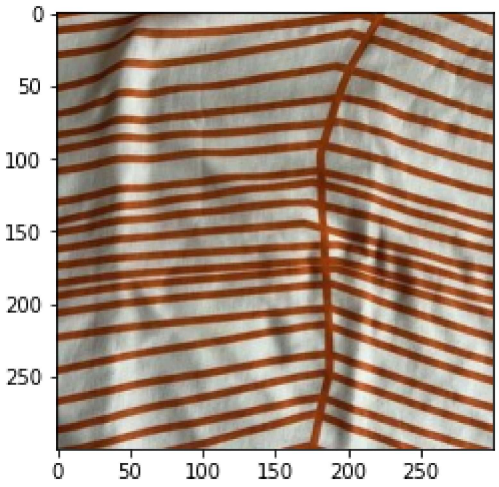

0
50
100
150
200
250
0
50
100
150
200
250

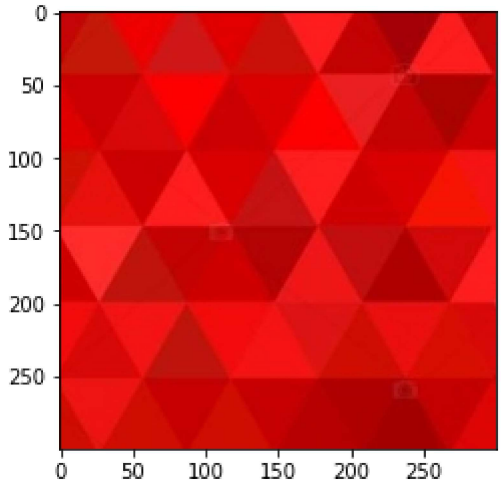

0
50
100
150
200
250
0
50
100
150
200
250

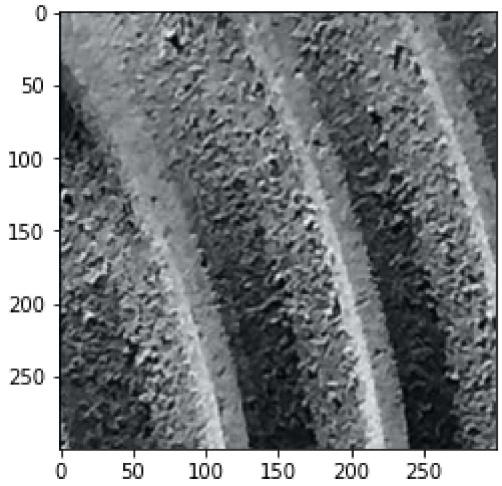

0
200
400
600
800
1000
0
200
400
600
800
1000

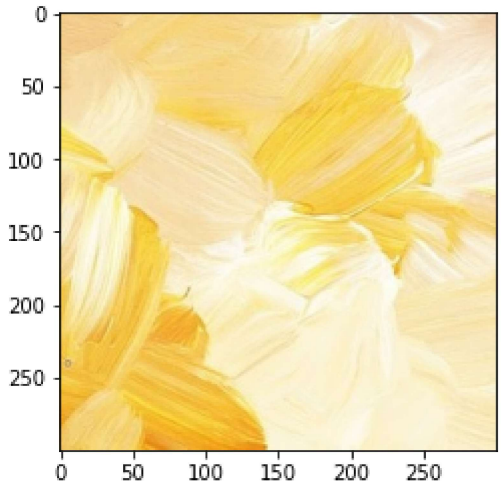

0
50
100
150
200
250
0
50
100
150
200
250

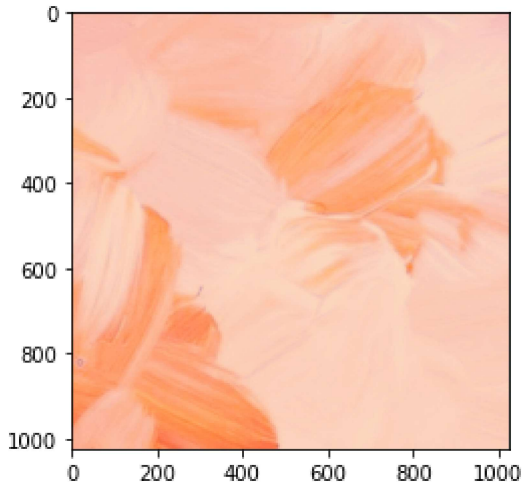

0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250

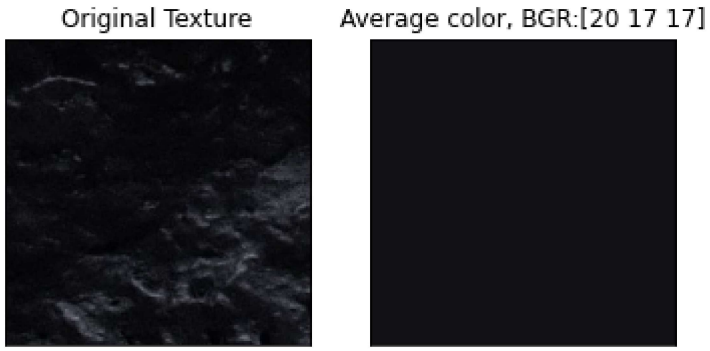

0
50
100
150
200
250
0
50
100
150
200
250

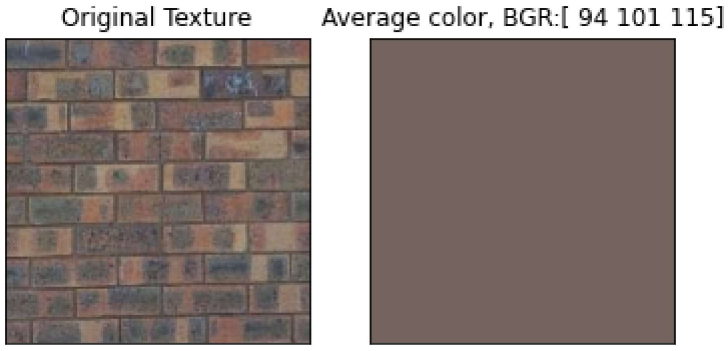

0
200
400
600
800
1000
0
200
400
600
800
1000

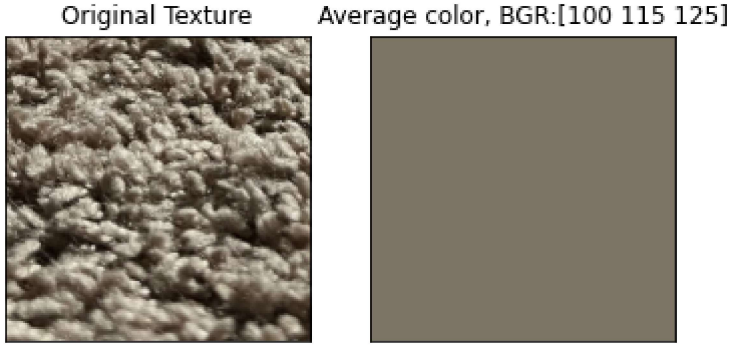
In [7]:
```
# Computes the average color of a texture
# https://www.delftstack.com/howto/python/opencv-average-color-of-image/
def compute_avg_color(textures):
    texture_colors_bgr = []
    for texture in textures:
        average_color_row = np.average(texture, axis=0)
        average_color = np.average(average_color_row, axis=0)
        texture_colors_bgr.append(average_color)
    return texture_colors_bgr
```

In [8]:
```
# Returns a texture filled with input color
def visualize_color(color_BGR):
    image = np.zeros((300 , 300, 3), np.uint8)
    image[:,:,:] = (color_BGR[0], color_BGR[1], color_BGR[2])
    return image
```

In [9]:
```
texture_colors_bgr = compute_avg_color(textures)

# Displaying textures and their computed average colors
for i in range(len(textures)):
    computed_color = visualize_color(texture_colors_bgr[i])

    fig, axes = plt.subplots(1, 2)
    axes[0].imshow(cv2.cvtColor(textures[i], cv2.COLOR_BGR2RGB))
    axes[0].set_title('Original Texture'), axes[0].set_xticks([]), axes[0].set_yticks([])
    axes[1].imshow(cv2.cvtColor(computed_color, cv2.COLOR_BGR2RGB))
    axes[1].set_title('Average color, BGR:'+ str(computed_color[0][0])), axes[1].set_xticks([]),
```

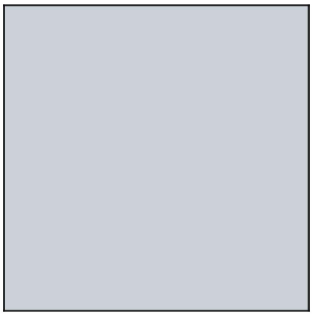






Original Texture

Average color, BGR:[216 208 201]

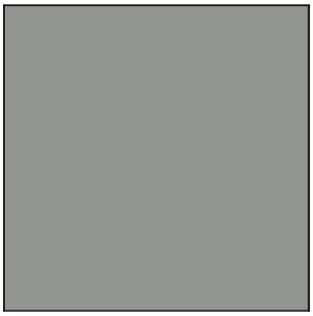


Original Texture

Average color, BGR:[86 88 89]

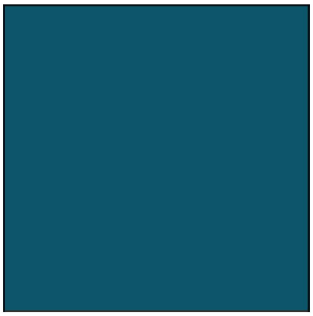
Original Texture

Average color, BGR:[143 149 147]



Original Texture

Average color, BGR:[107 86 11]

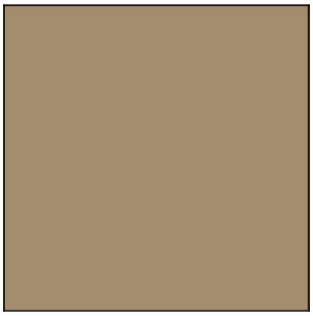


Original Texture

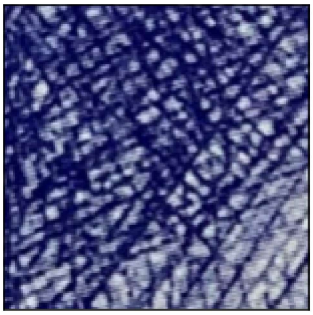
Average color, BGR:[32 69 33]

Original Texture

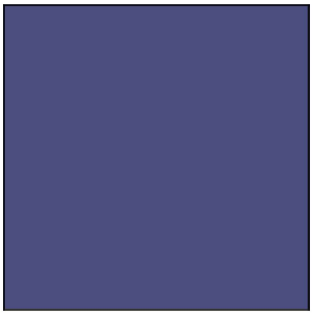
Average color, BGR:[109 141 163]



Original Texture

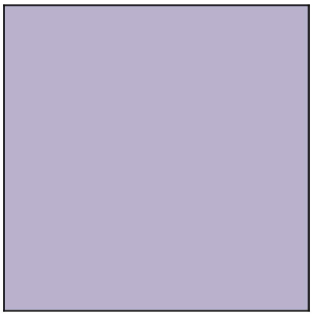


Average color, BGR:[127 78 74]



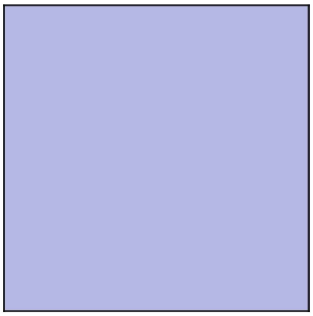
Original Texture

Average color, BGR:[207 179 185]



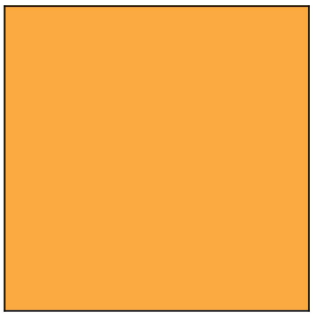
Original Texture

Average color, BGR:[226 184 180]

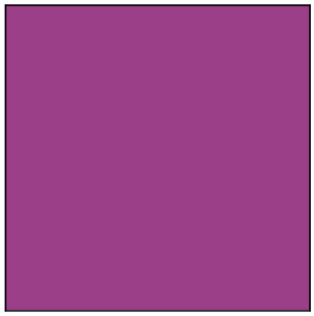


Original Texture

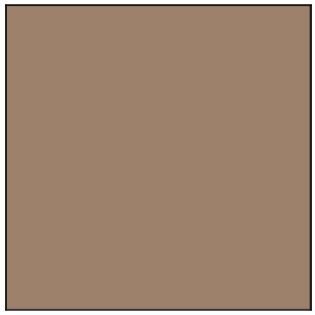
Average color, BGR:[62 170 250]



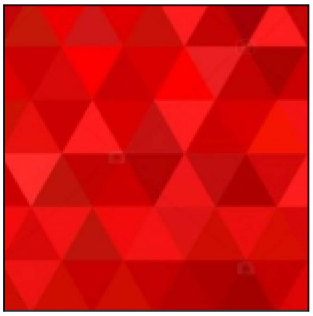
Original Texture Average color, BGR:[135 62 156]

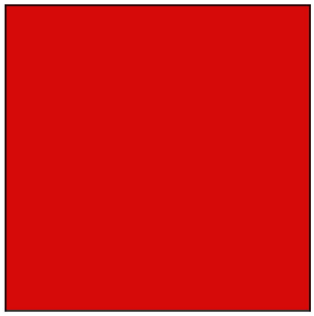


Original Texture Average color, BGR:[109 129 157]

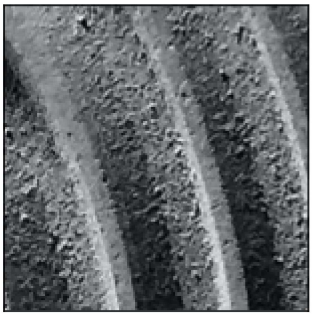


Original Texture Average color, BGR:[9 11 214]

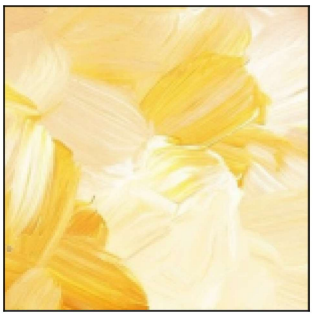


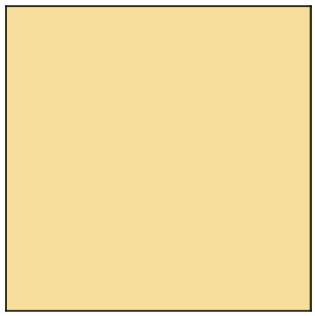


Original Texture Average color, BGR:[106 103 99]



Original Texture Average color, BGR:[157 224 247]





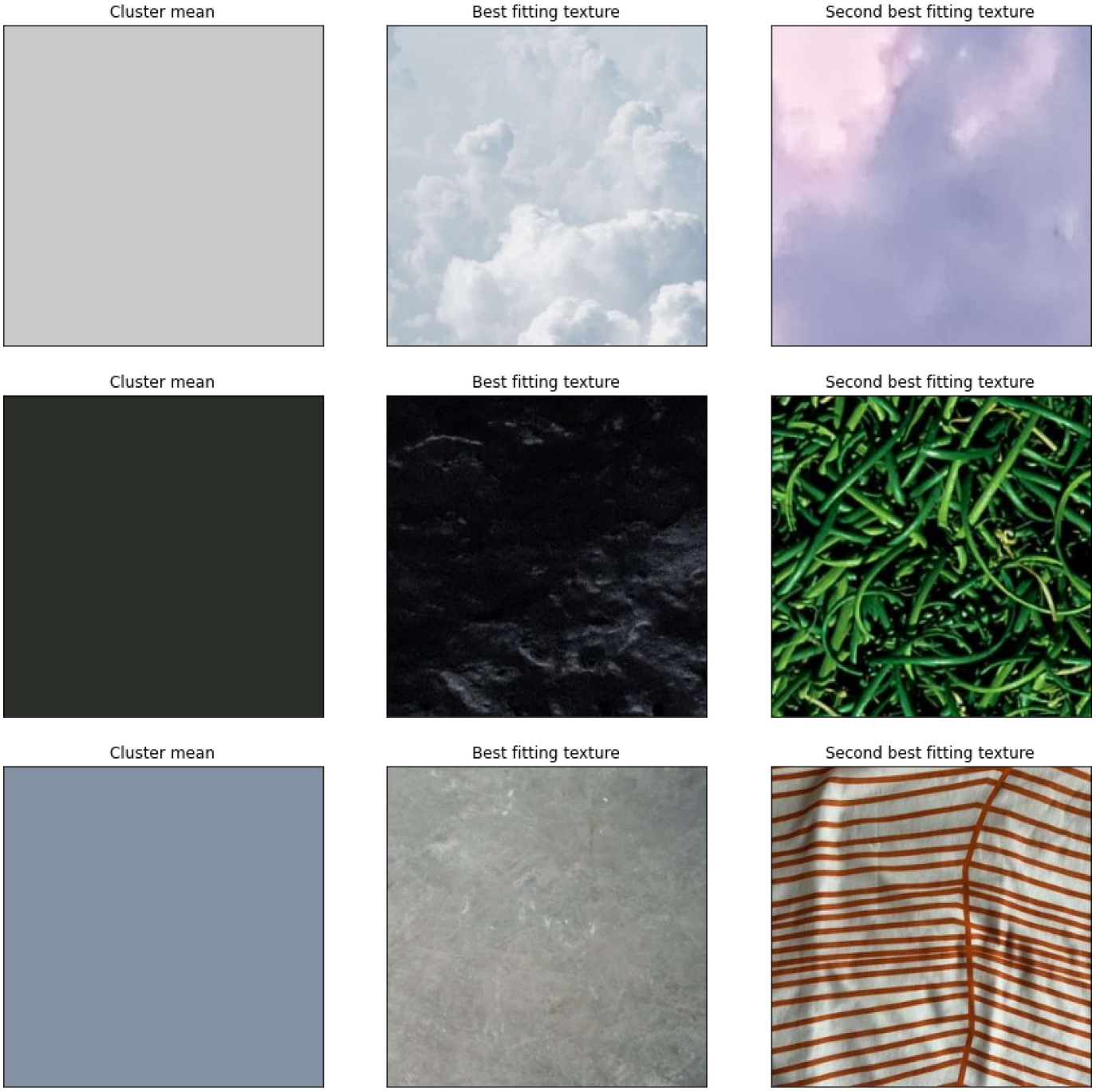
In [10]:
```
# Given a color, find the index of the texture in the texture array with closest color
# Finding the sum of the absolute difference between the red, green and blue numbers and choose
# https://stackoverflow.com/questions/54242194/python-find-the-closest-color-to-a-color-from-giv
def color_to_texture_index(color, texture_colors):
    target_r = color[2]
    target_g = color[1]
    target_b = color[0]
    color_diffs = []
    for i in range(len(texture_colors)):
        r = texture_colors[i][2]
        g = texture_colors[i][1]
        b = texture_colors[i][0]
        color_diff = math.sqrt((r - target_r)**2 + (g - target_g)**2 + (b - target_b)**2)
        color_diffs.append((color_diff, i))
    s = set(color_diffs)
    return min(color_diffs)[1], sorted(s)[1][1]
```

In [11]:
```
# PROBLEM: Sometimes chooses a wrong color since the only determining factor is euclidean distan
# Try to convert to other color spaces for more accurate comparison
def color_to_texture_index_LAB(color, texture_colors):
    color_block = visualize_color(color)
    color_lab = cv2.cvtColor(color_block, cv2.COLOR_BGR2LAB)
    target_l = color_lab[0][0][0]
    target_a = color_lab[0][0][1]
    target_b = color_lab[0][0][2]
    print("color_to_texture_index_LAB: l", target_l)
    print("color_to_texture_index_LAB: a", target_a)
    print("color_to_texture_index_LAB: b", target_b)
    color_diffs = []
    for i in range(len(texture_colors)):
        curr_color_block = visualize_color(texture_colors[i])
        curr_color_lab = cv2.cvtColor(curr_color_block, cv2.COLOR_BGR2LAB)
        l = int(curr_color_lab[0][0][0])
        a = int(curr_color_lab[0][0][1])
        b = int(curr_color_lab[0][0][2])
        color_diff = math.sqrt(int(l - target_l)**2 + 2*int(a - target_a)**2 + int(b - target_b)
        color_diffs.append((color_diff, i))
    s = set(color_diffs)
    return min(color_diffs)[1], sorted(s)[1][1]
```

In [12]:
```
# Displaying target color, best choice, second choice
for i in range(3):
    first_color = center[i]
    print(first_color)
    color_block = visualize_color(first_color)
    texture_index, texture_index_2 = color_to_texture_index_LAB(first_color, texture_colors_bgr)
    print(texture_index, texture_index_2)

    fig, axes = plt.subplots(1, 3, figsize=(15,15))
    axes[0].imshow(cv2.cvtColor(color_block, cv2.COLOR_BGR2RGB))
    axes[0].set_title('Cluster mean'), axes[0].set_xticks([]), axes[0].set_yticks([])
```

```
    axes[1].imshow(cv2.cvtColor(textures[texture_index], cv2.COLOR_BGR2RGB))
    axes[1].set_title('Best fitting texture'), axes[1].set_xticks([]), axes[1].set_yticks([])
    axes[2].imshow(cv2.cvtColor(textures[texture_index_2], cv2.COLOR_BGR2RGB))
    axes[2].set_title('Second best fitting texture'), axes[2].set_xticks([]), axes[2].set_yticks
```

```
[200 201 201]
color_to_texture_index_LAB: l 207
color_to_texture_index_LAB: a 128
color_to_texture_index_LAB: b 129
3 10
[38 45 42]
color_to_texture_index_LAB: l 46
color_to_texture_index_LAB: a 125
color_to_texture_index_LAB: b 132
0 7
[159 144 131]
color_to_texture_index_LAB: l 151
color_to_texture_index_LAB: a 127
color_to_texture_index_LAB: b 118
5 14
```



Generate background images with textures

Using seam-finding approach

In [13]:
```
def ssd_patch(M, T, I, patch_size):
    """
    helper functions to quilt_sample
    computing the cost of sampling each patch based on the sum of squared differences (SSD) of t
    of the existing and sampled patch
    :param M: numpy.ndarray         The mask
    :param T: numpy.ndarray         The template
    :param I: numpy.ndarray         The sample image
    :return ssd_cost: numpy.ndarray
    """

    I = I.copy() / 255
    ssd_cost1 = ((M*T[:,:,0])**2).sum() - 2 * cv2.filter2D(I[:,:,0], ddepth=-1, kernel = M*T[:,:
    ssd_cost2 = ((M*T[:,:,1])**2).sum() - 2 * cv2.filter2D(I[:,:,1], ddepth=-1, kernel = M*T[:,:
    ssd_cost3 = ((M*T[:,:,2])**2).sum() - 2 * cv2.filter2D(I[:,:,2], ddepth=-1, kernel = M*T[:,:

    cost_image = ssd_cost1 + ssd_cost2 + ssd_cost3
    return cost_image
```

In [14]:
```
def choose_sample(cost_image, patch_size, tol):
    """
    :param cost_image: numpy.ndarry
    :param tol: float
    """
    radius = patch_size // 2
    candidates = []

    min = np.argsort(cost_image, None)

    for i in min:
        a, b = np.unravel_index(i, cost_image.shape)
        if cost_image.shape[0] - radius > a > radius and cost_image.shape[1] - radius > b > radi
            candidates.append((a, b))
        if len(candidates) >= tol:
            break

        r = random.randint(0, tol - 1)
    return candidates[r]
```

In [15]:
```
def error(simple_overlap, sample_overlap):
    err = np.square(simple_overlap - sample_overlap)
    err = np.sum(err, axis=2)
    return err
```

In [16]:
```
def create_3D_mask(mask):
    x, y = mask.shape
    new_mask = np.zeros((x, y, 3), dtype=np.float64)
    for i in range(x):
        for j in range(y):
            if mask[i, j] == 1:
                new_mask[i, j, :] = 1
            else:
                new_mask[i, j, :] = 0
    return new_mask
```

In [17]:
```
def mask_vertical(output, sample, x, y, u, v):
    simple_overlap = output[x:x+patch_size, y:y+overlap]
    sample_overlap = sample[u:u+patch_size, v:v+overlap]
```

```
    err_patch = error(simple_overlap, sample_overlap)
    mask = cut(err_patch.T).T
    mask = create_3D_mask(mask)
    seam = sample_overlap * mask + simple_overlap * (1 - mask)
    output[x:x+patch_size, y:y+patch_size] = sample[u:u+patch_size, v:v+patch_size]
    output[x:x+patch_size, y:y+overlap] = seam

def mask_horizontal(output, sample, x, y, u, v):
    simple_overlap = output[x:x+overlap, y:y+patch_size]
    sample_overlap = sample[u:u+overlap, v:v+patch_size]
    err_patch = error(simple_overlap, sample_overlap)
    mask = cut(err_patch)
    mask = create_3D_mask(mask)
    seam = sample_overlap * mask + simple_overlap * (1 - mask)
    output[x:x+patch_size, y:y+patch_size] = sample[u:u+patch_size, v:v+patch_size]
    output[x:x+overlap, y:y+patch_size] = seam

def mask_else(output, sample, x, y, u, v):
    simple_overlap = output[x:x+patch_size, y:y+overlap]
    sample_overlap = sample[u:u+patch_size, v:v+overlap]
    err_patch = error(simple_overlap, sample_overlap)
    mask1 = cut(err_patch.T).T
    mask1_sqr = np.ones((patch_size,patch_size), dtype = "int")
    h, w = mask1.shape
    for c in range(h):
        for d in range(w):
            if mask1[c,d] == 1:
                mask1_sqr[c,d] = 1
            else:
                mask1_sqr[c,d] = 0
    simple_overlap = output[x:x+overlap, y:y+patch_size]
    sample_overlap = sample[u:u+overlap, v:v+patch_size]
    err_patch = error(simple_overlap, sample_overlap)
    mask2 = cut(err_patch)
    mask2_sqr = np.ones((patch_size,patch_size), dtype = "int")
    h, w = mask2.shape
    for c in range(h):
        for d in range(w):
            if mask2[c,d] == 1:
                mask2_sqr[c,d] = 1
            else:
                mask2_sqr[c,d] = 0
    mask = np.logical_and(mask1_sqr,mask2_sqr)
    mask = create_3D_mask(mask)
    temp = output[x:x+patch_size,y:y+patch_size]
    seam = mask * sample[u:u+patch_size, v:v+patch_size] + (1 - mask) * temp
    output[x:x+patch_size, y:y+patch_size] = seam
```

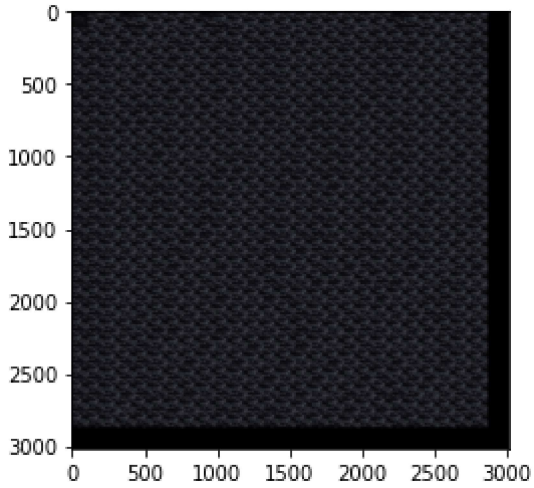
In [18]:

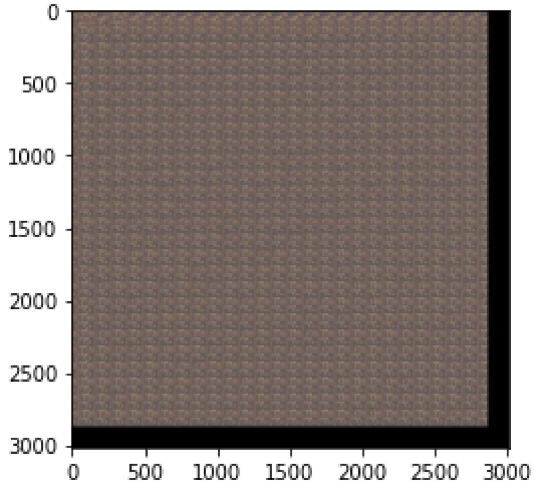
```
def quilt_cut(sample, out_size, patch_size, overlap, tol):
    """
    Samples square patches of size patchsize from sample using seam finding in order t
    Feel free to add function parameters
    :param sample: numpy.ndarray
    :param out_size: int
    :param patch_size: int
    :param overlap: int
    :param tol: float
    :return: numpy.ndarray
    """
    output = np.zeros((out_size, out_size, 3), dtype=np.float64)
    fill_size = patch_size - overlap
    num_sqr = (out_size - patch_size) // fill_size
    radius = patch_size // 2
    # randomly pick the first patch
```

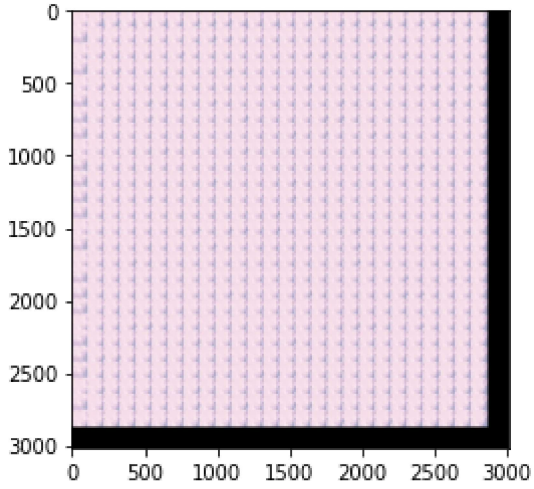
```
    num_pick = sample.shape[0] - patch_size     # randomly pick from the sample within
    # r = (int)(random.random()*num_pick)         # randomly choose the first patch
    r = 0
    patch = output[0:patch_size, 0:patch_size]  # waiting to be iterated
    output[0:patch_size, 0:patch_size] = sample[r:r+patch_size, r:r+patch_size]
    for i in range(num_sqr):
        for j in range(num_sqr):
            mask = np.zeros((patch_size, patch_size, 3), dtype=np.float64)
            # compute mask
            if (i == 0) and (j == 0):
                continue
            if i == 0:
                mask[:, :overlap, :] = 1
            elif j == 0:
                mask[:overlap, :, :] = 1
            else:
                mask[:, :overlap, :] = 1
                mask[:overlap, :, :] = 1
            x = i * fill_size
            y = j * fill_size
            mask = mask[:,:,0]
            template = output[x:x+patch_size, y:y+patch_size]
            cost_image = ssd_patch(mask, template, sample, patch_size)
            a, b = choose_sample(cost_image, patch_size, tol)
            u = a - radius
            v = b - radius
            if i == 0 and j != 0:
                mask_vertical(output, sample, x, y, u, v)
            elif j == 0 and i != 0:
                mask_horizontal(output, sample, x, y, u, v)
            elif i != 0 and j != 0:
                mask_else(output, sample, x, y, u, v)
    output /= 255
    return output
```

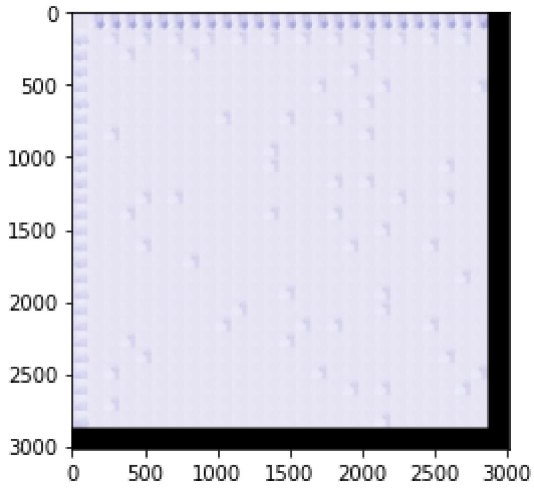
In [19]:

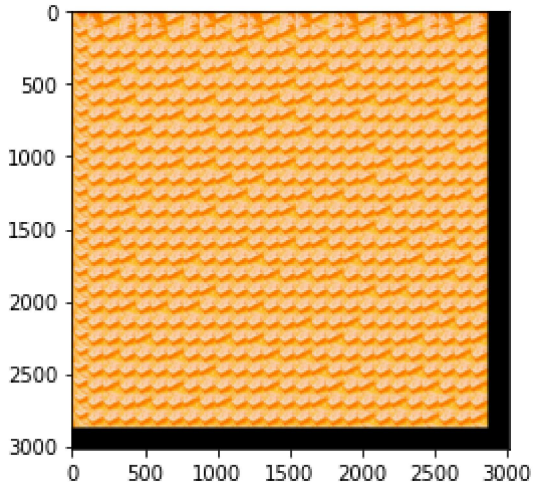
```
out_size = max(width, height)+200
patch_size = 120
overlap = 10
tol = 10

backgrounds_seam = []
temp = [cv2.imread(file) for file in glob.glob("generated_seam/*.png")]

if len(temp) < len(textures):
    for i in range(len(textures)):
        curr_texture = cv2.cvtColor(textures[i], cv2.COLOR_BGR2RGB)
        res = quilt_cut(curr_texture, out_size, patch_size, overlap, tol)

        if res is not None:
            plt.figure(figsize=(10,10))
            plt.imshow(res)
            backgrounds_seam.append(res)
            write_image(backgrounds_seam[i], 'generated_seam/'+str(i) +'.png')
else:
    backgrounds_seam = [cv2.imread(file) for file in glob.glob("generated_seam/*.png")]
    for background in backgrounds_seam:
        plt.imshow(cv2.cvtColor(background, cv2.COLOR_BGR2RGB))
        plt.show()
```

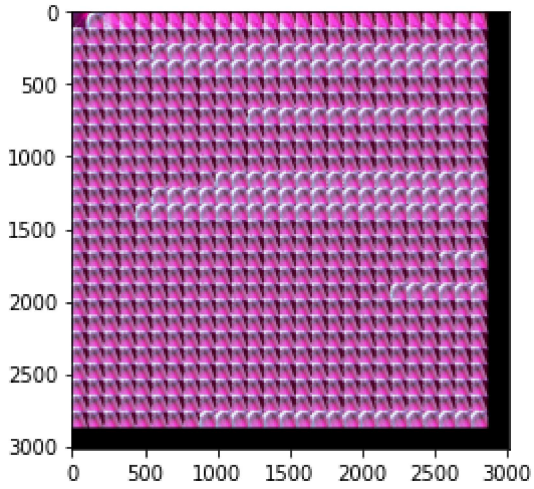

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

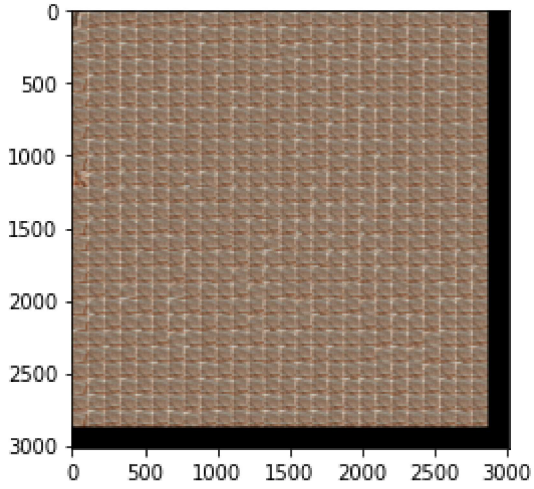

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

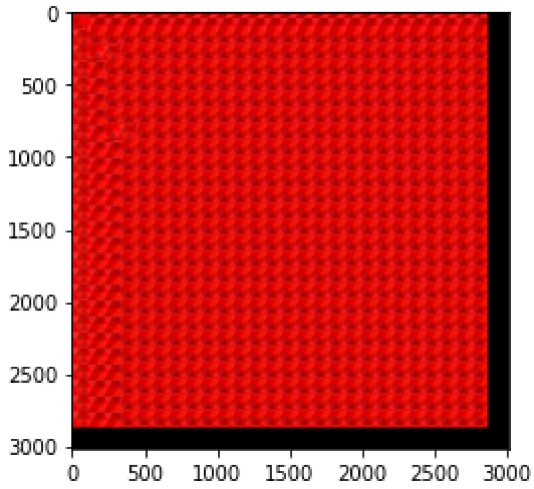

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

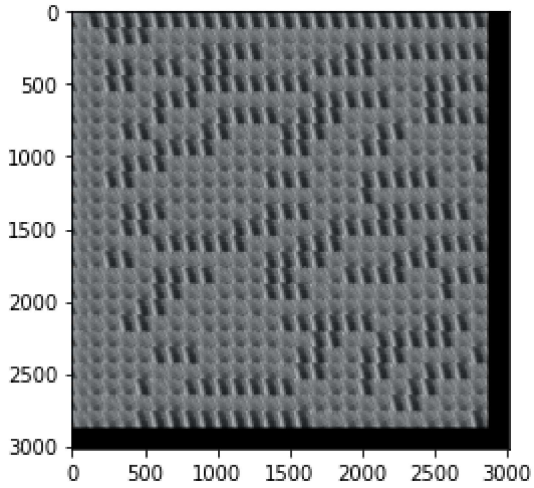

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

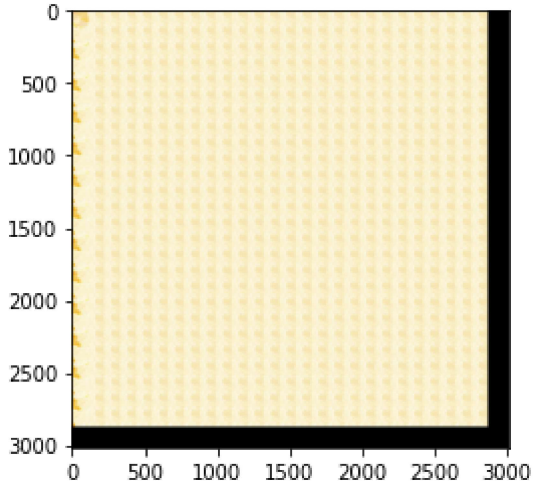

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

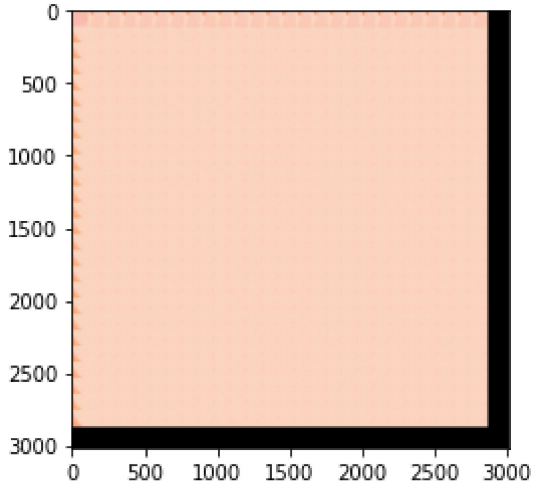

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

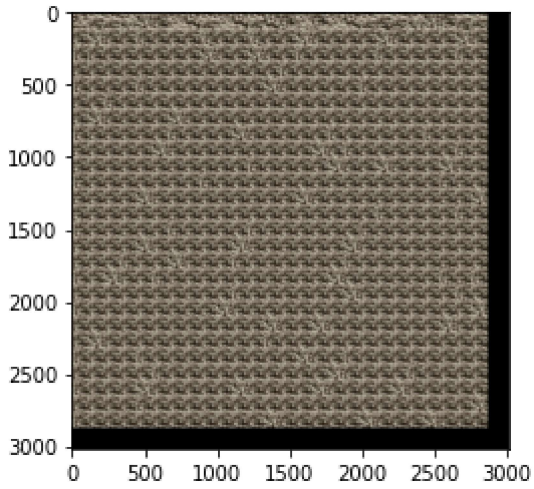

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

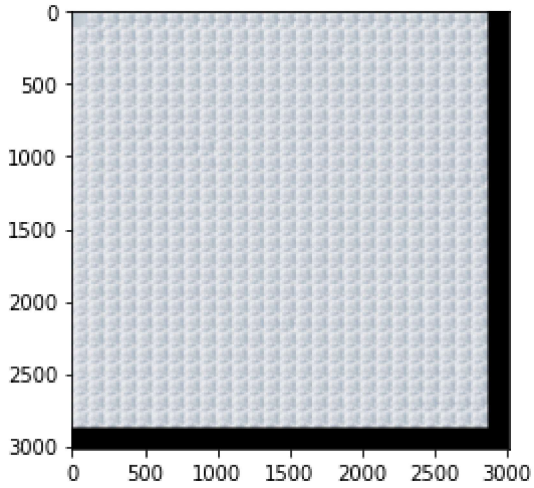

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

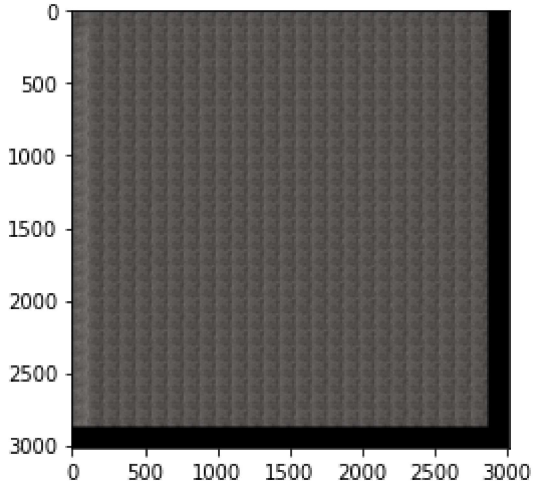

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

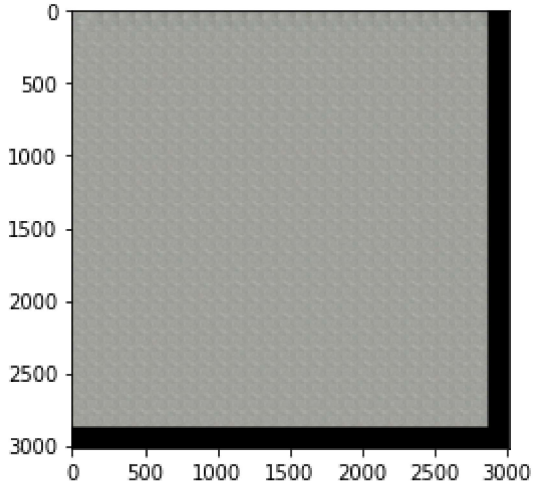

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

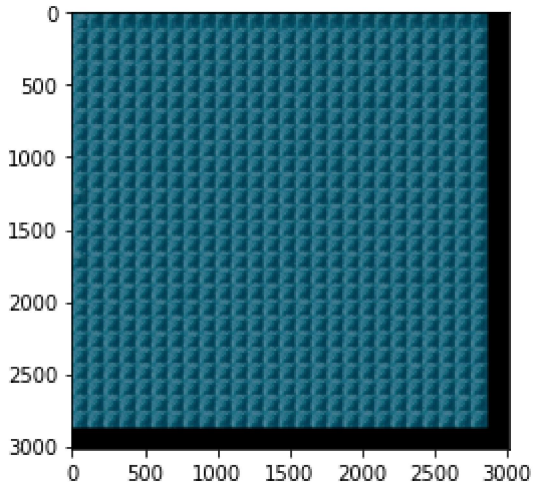

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

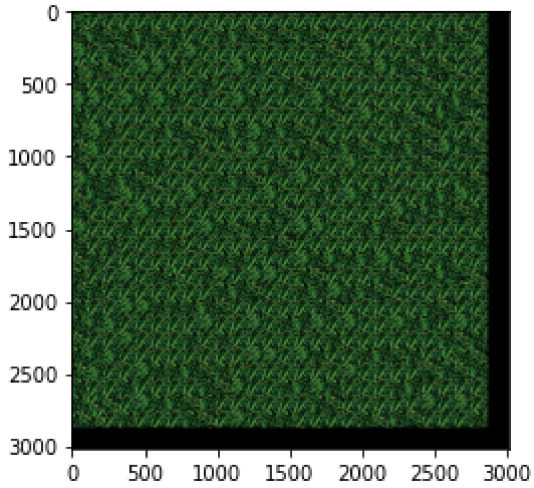

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

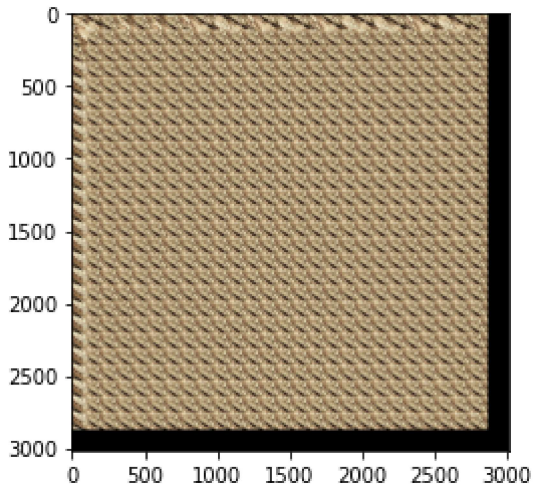

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

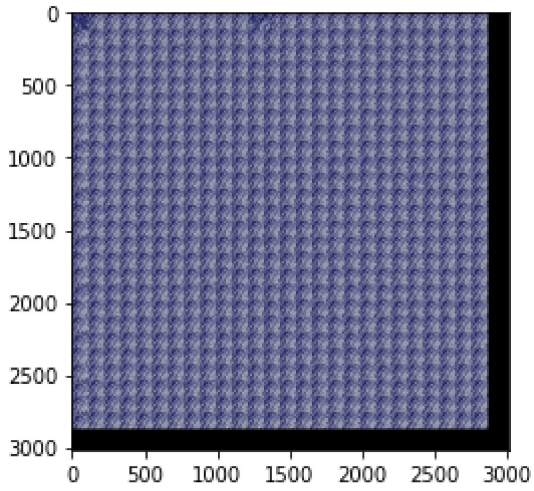

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000


0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000


0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000







Using simple approach

In [20]:
```
def patch_start(sample_size, patch_size):
        """
        Returns the position of a random pixel as patch in sample image
        :param sample_size: int         The width of the sample image
        :param patch_size: int          The width of the square sample patch
        :return: numpy.ndarray
        """
        patch_num = random.randint(0, (int(sample_size/patch_size)**2)-1)
        x_num = int(patch_num%int(sample_size/patch_size))
        y_num = int(patch_num/int(sample_size/patch_size))
        return x_num*patch_size, y_num*patch_size
```

In [21]:
```
def ssd_patch_simple(target, mask, sample, patch_size):
```

```
    sample_R, sample_G ,sample_B = cv2.split(sample/225)
    template_R, template_G, template_B = cv2.split(target/225)

    result_R = np.zeros(sample.shape[0:2]);
    result_G = np.copy(result_R);
    result_B = np.copy(result_R);

    # Equations provided in doc
    result_R = ((mask*template_R)**2).sum() - 2 * cv2.filter2D(sample_R, ddepth=-1, kernel = mas
    result_G = ((mask*template_G)**2).sum() - 2 * cv2.filter2D(sample_G, ddepth=-1, kernel = mas
    result_B = ((mask*template_B)**2).sum() - 2 * cv2.filter2D(sample_B, ddepth=-1, kernel = mas
    return result_R + result_G + result_B
```

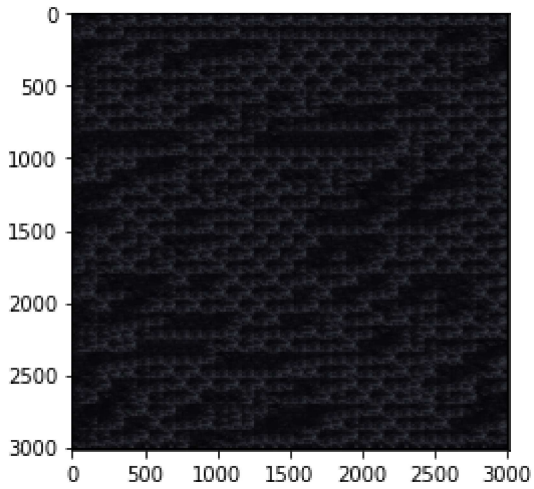
In [22]:
```
def choose_sample_simple(ssd, tol, patch_size):
    min_table = np.zeros((tol, 3))
    min_table[:, 0] = float('inf')
    min_table[:, 1] = 0
    min_table[:, 2] = 0
    center = patch_size // 2

    for i in range(center, ssd.shape[0]-center):
        for j in range(center, ssd.shape[1]-center):
            min_value = np.amax(min_table[:, 0])
            min_index = np.argmax(min_table[:, 0])
            if ssd[i, j] == 0:
                continue;
            if ssd[i, j] < min_value:
                min_table[min_index, 0] = ssd[i, j]
                min_table[min_index, 1] = i
                min_table[min_index, 2] = j
    ind = random.randint(0, tol-1)

    return int(min_table[ind, 1] - center), int(min_table[ind, 2] - center)
```

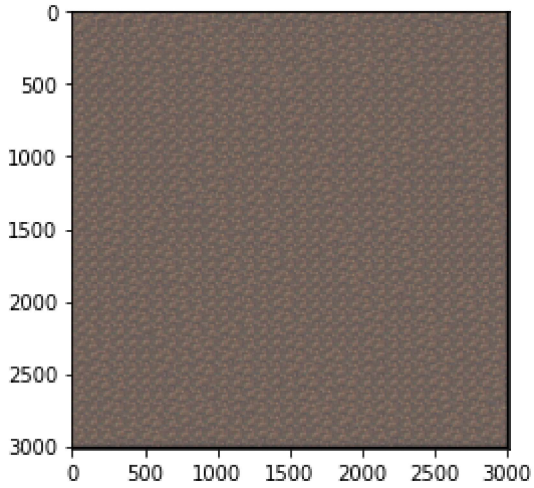
In [23]:
```
def quilt_simple(sample, out_size, patch_size, overlap, tol):
    """
    Randomly samples square patches of size patchsize from sample in order to create an output i
    Feel free to add function parameters
    :param sample: numpy.ndarray
    :param out_size: int
    :param patch_size: int
    :param overlap: int
    :param tol: int
    :return: numpy.ndarray
    """
    image = np.zeros((out_size,out_size,3))

    x_coord, y_coord = patch_start(sample.shape[0], patch_size)
    image[0:patch_size,0:patch_size :] = sample[x_coord:x_coord+patch_size, y_coord:y_coord+patc

    mask_top = np.zeros((patch_size, patch_size))
    mask_left = np.copy(mask_top)
    mask_top_left = np.copy(mask_top)
    mask_top[0:overlap, :] = 1
    mask_left[:, 0:overlap] = 1
    mask_top_left[:, 0:overlap] = 1
    mask_top_left[0:overlap, :] = 1

    i = 0
    j = patch_size - overlap
    diff = j
    while i + patch_size <= out_size:
        while j + patch_size <= out_size:
            print(i, j)
```

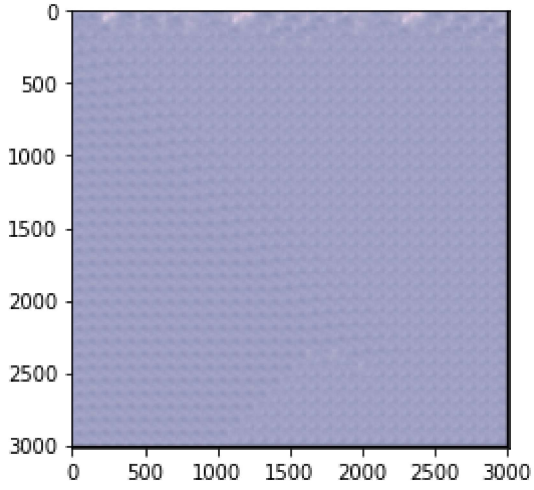
```
            mask = None
            if i >= diff and j >= diff:
                mask = mask_top_left
            elif j >= diff:
                mask = mask_left
            elif i >= diff:
                mask = mask_top

            template = image[i:i+patch_size,j:j+patch_size]
            ssd = ssd_patch_simple(template, mask, sample, patch_size)
            x, y = choose_sample_simple(ssd, tol, patch_size)
            image[i:i+patch_size, j:j+patch_size, :] = sample[x:x+patch_size, y:y+patch_size, :]
            j += diff
        j = 0
        i += diff

    return image/255
```

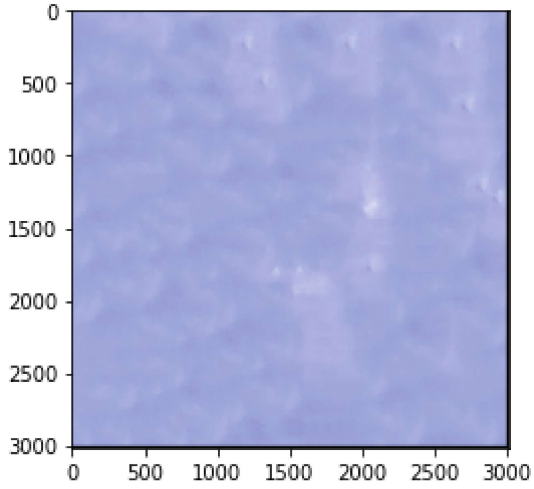
In [24]:

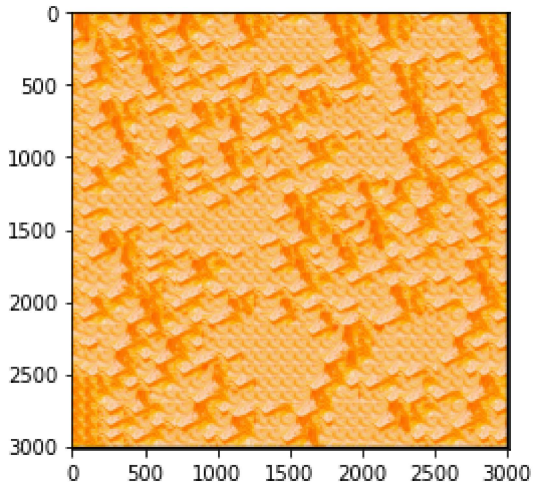
```
out_size = max(width, height) + 200
patch_size = 120
overlap = 30
tol = 4

backgrounds_simple = []
temp = [cv2.imread(file) for file in glob.glob("generated_simple/*.png")]

if len(temp) < len(textures)-1:
    for i in range(len(textures)):
        curr_texture = cv2.cvtColor(textures[i], cv2.COLOR_BGR2RGB)
        res = quilt_simple(curr_texture, out_size, patch_size, overlap, tol)

        if res is not None:
            plt.figure(figsize=(10,10))
            plt.imshow(res)
            backgrounds_simple.append(res)
            write_image(backgrounds_simple[i], 'generated_simple/'+str(i) +'.png')

else:
    backgrounds_simple = [cv2.imread(file) for file in glob.glob("generated_simple/*.png")]
    for background in backgrounds_simple:
        plt.imshow(cv2.cvtColor(background, cv2.COLOR_BGR2RGB))
        plt.show()
```

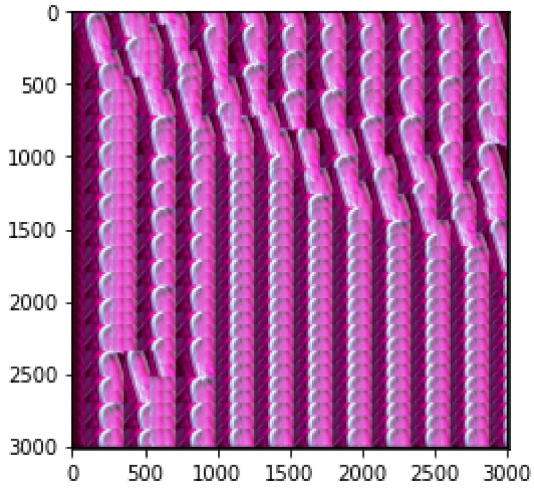


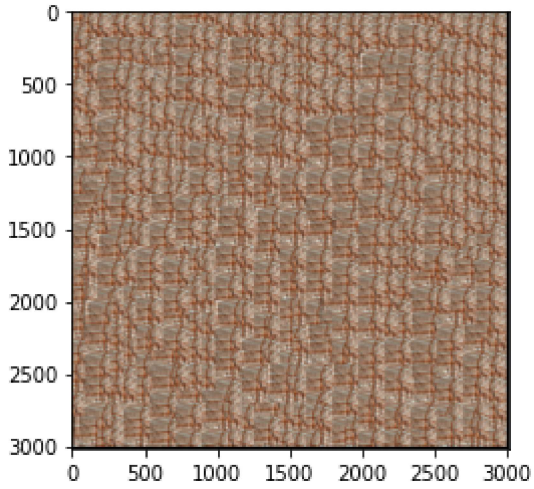

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

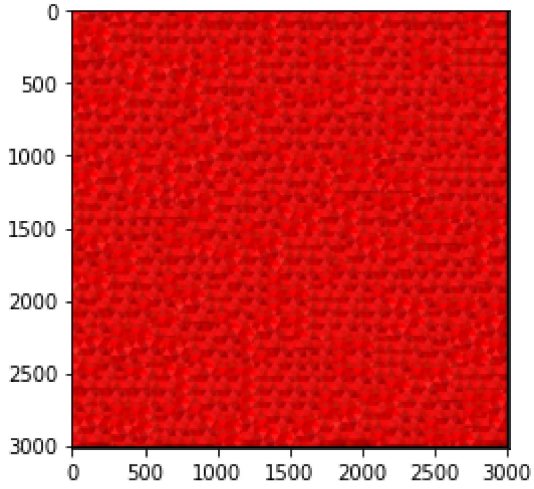

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

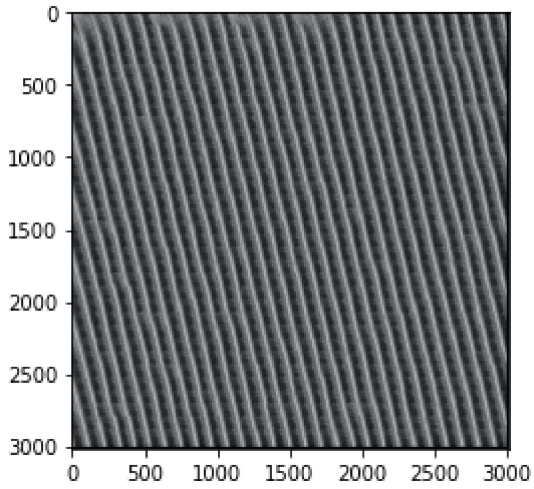

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

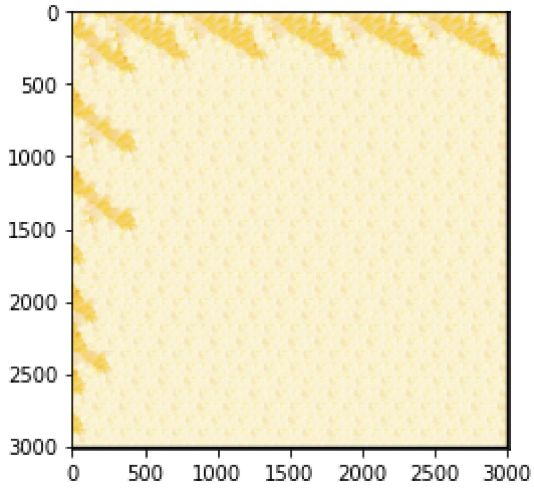

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

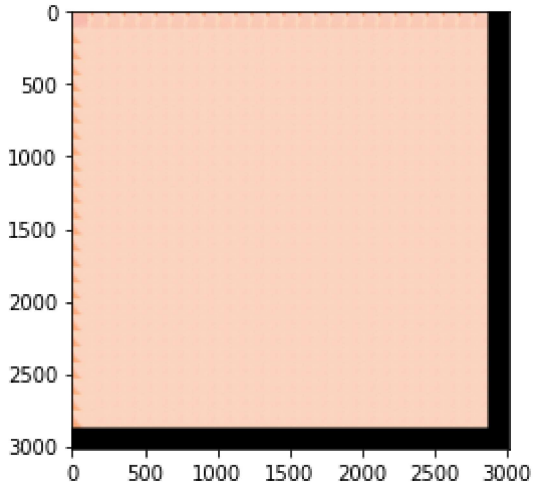

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

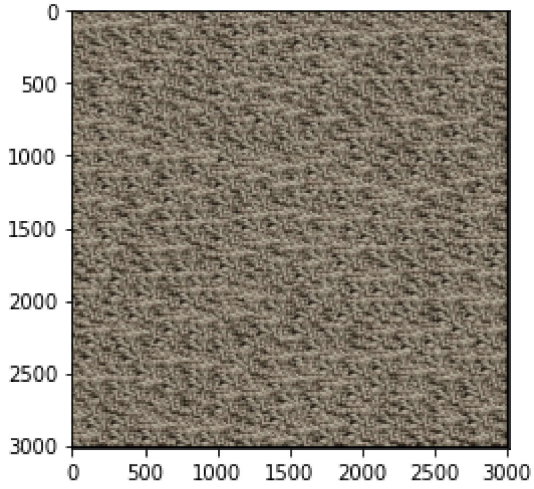

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

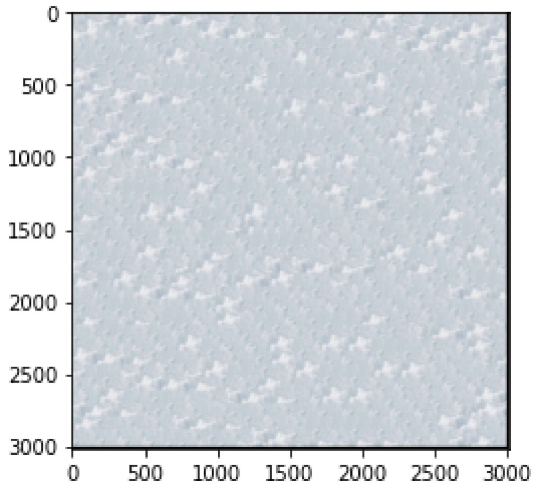

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

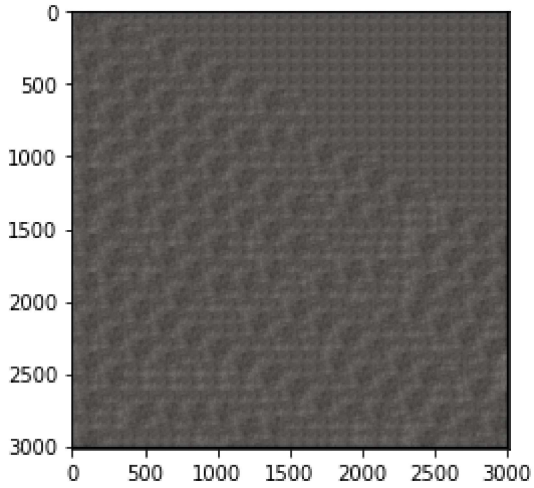

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

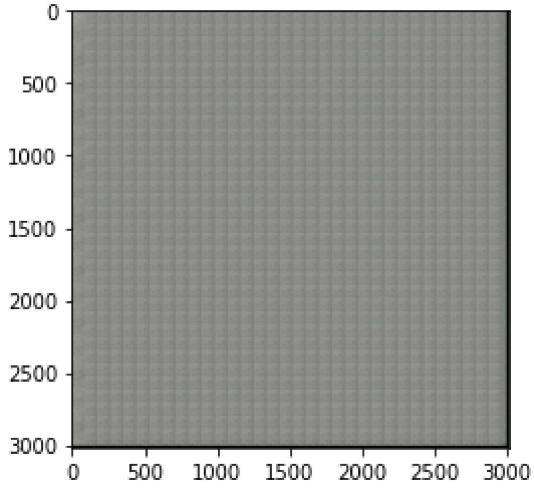

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

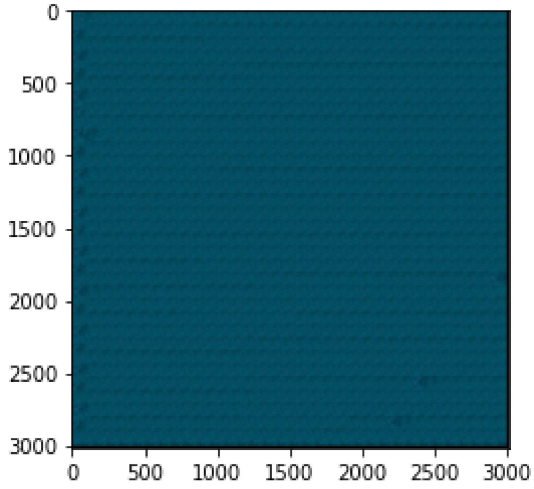

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

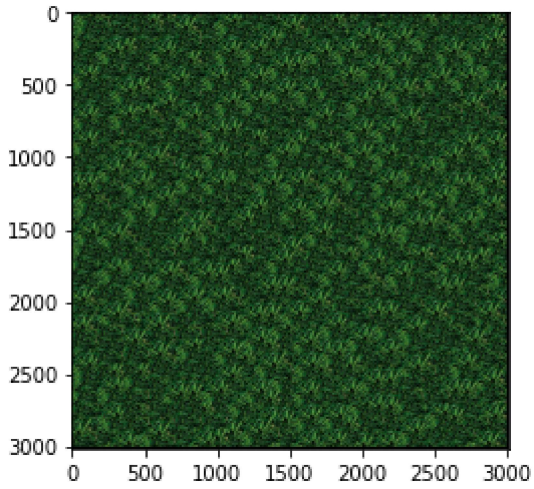

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

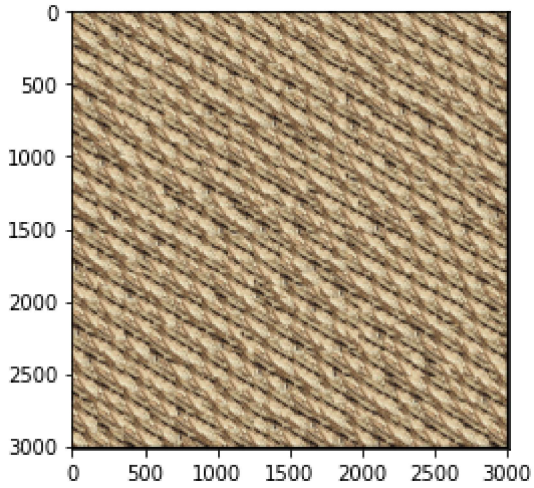

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000

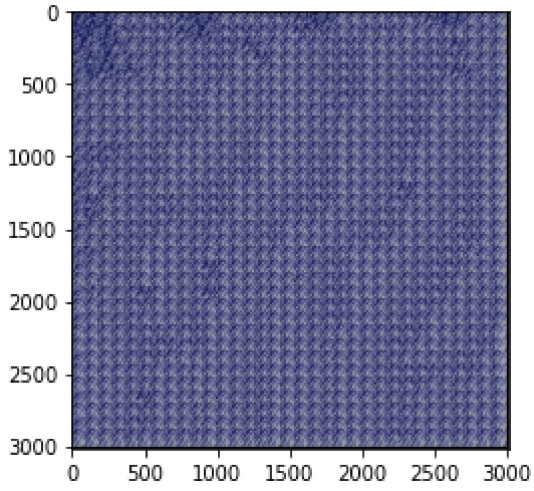

0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000


0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000


0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000


0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000





Apply texture based on K-means

In [25]:
```
def getSegement(labels, labelIndex):
    indices = np.where(labels==labelIndex, labels, -1)
    return indices
```

In [26]:
```
def getMask(input):
    mask = input.copy()
    mask[mask != -1] = 1
    mask[mask == -1] = 0
    mask3d = np.zeros((mask.shape[0], mask.shape[1], 3))
    for i in range(3):
        mask3d[:, :, i] = mask
    return mask3d
```

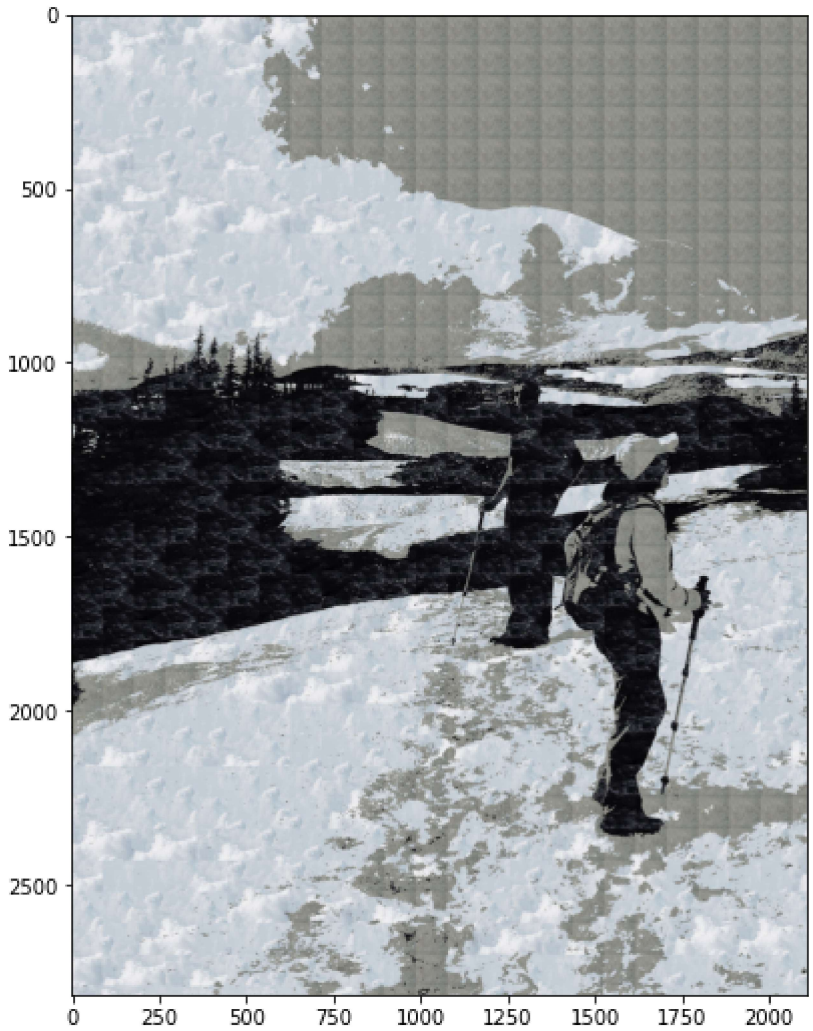
In [27]:
```
def blend(texture, input_im, mask):
    im = input_im.copy()
    output = mask * texture + (1-mask) * im
    return output
```

In [28]:
```
def blendAll(textures, label):
    input_im = np.zeros((width, height, 3))
    masks = []
    for i in range(3):
        layer = getSegement(label, i)
        masks.append(getMask(layer))
        output_im = blend(textures[i], input_im, masks[i])
        input_im = output_im
```

In [29]:
```
selected_textures = np.zeros((K, width, height, 3))
background_1 = cv2.imread('generated_simple/3.png', cv2.COLOR_BGR2RGB)
selected_textures[0] = background_1[0:width, 0:height, :]
background_2 = cv2.imread('generated_simple/0.png', cv2.COLOR_BGR2RGB)
selected_textures[1] = background_2[0:width, 0:height, :]
background_3 = cv2.imread('generated_simple/5.png', cv2.COLOR_BGR2RGB)
selected_textures[2] = background_3[0:width, 0:height, :]

input_im = res2

masks = []
for i in range(3):
    layer = getSegement(label, i)
    masks.append(getMask(layer))
    output_im = blend(selected_textures[i], input_im, masks[i])
    input_im = output_im
```

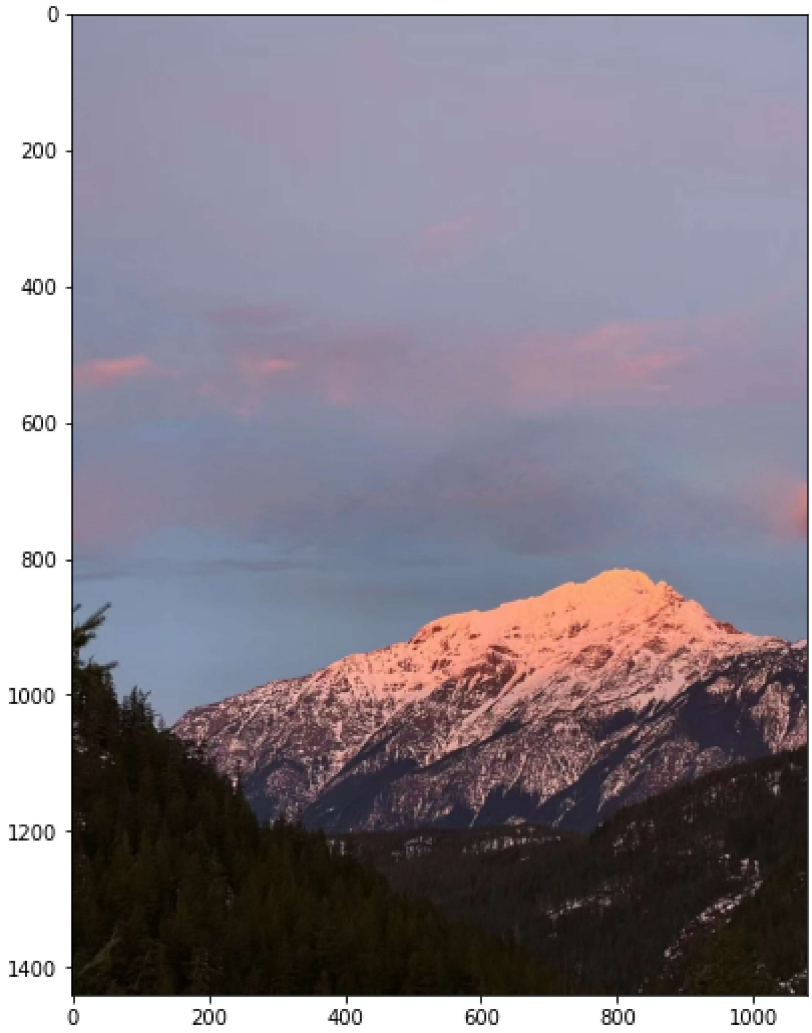
In [30]:
```
# change to unit8 to present original color
plt.figure(figsize = (16,9))
plt.imshow(np.uint8(input_im)[:, :, [2, 1, 0]])
```

Out[30]:
```
<matplotlib.image.AxesImage at 0x20ca3b82680>
```



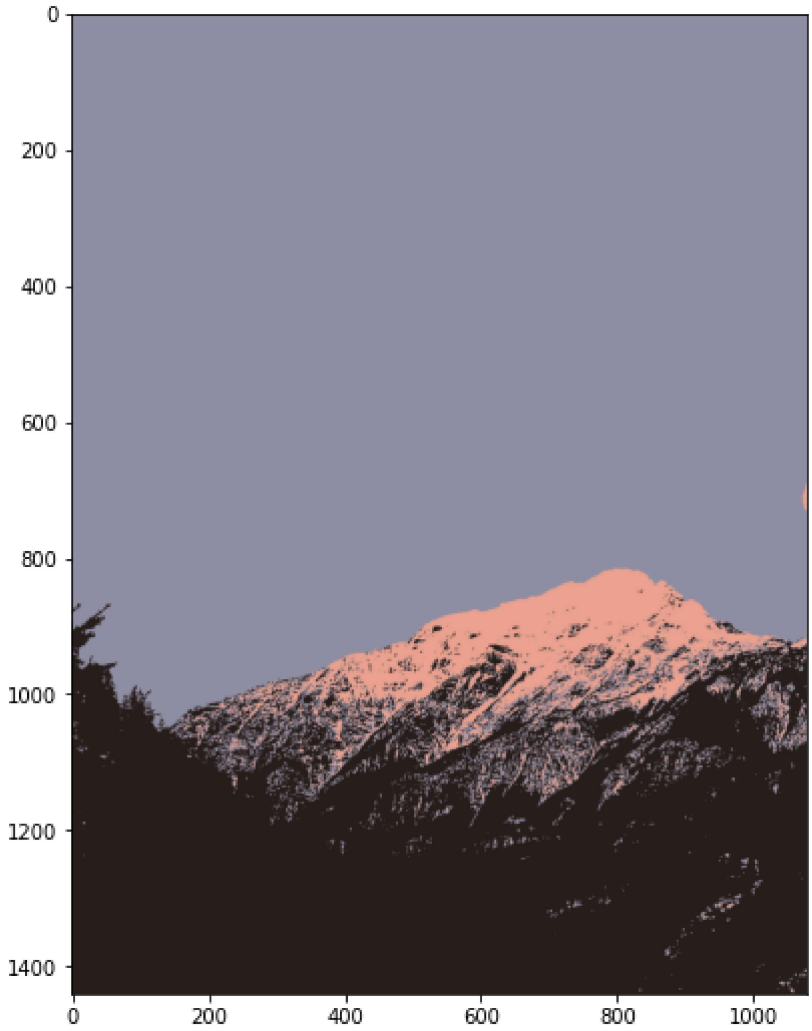
In [31]:
```
img2 = cv2.imread('samples/im2.JPG', cv2.COLOR_BGR2RGB)
width = img2.shape[0]
height = img2.shape[1]
Z = img2.reshape((-1,3))
Z = np.float32(Z)  # convert to np.float32
plt.figure(figsize = (16,9))
plt.imshow(img2[:, :, [2, 1, 0]])
```

Out[31]: <matplotlib.image.AxesImage at 0x20ca3cb5c90>



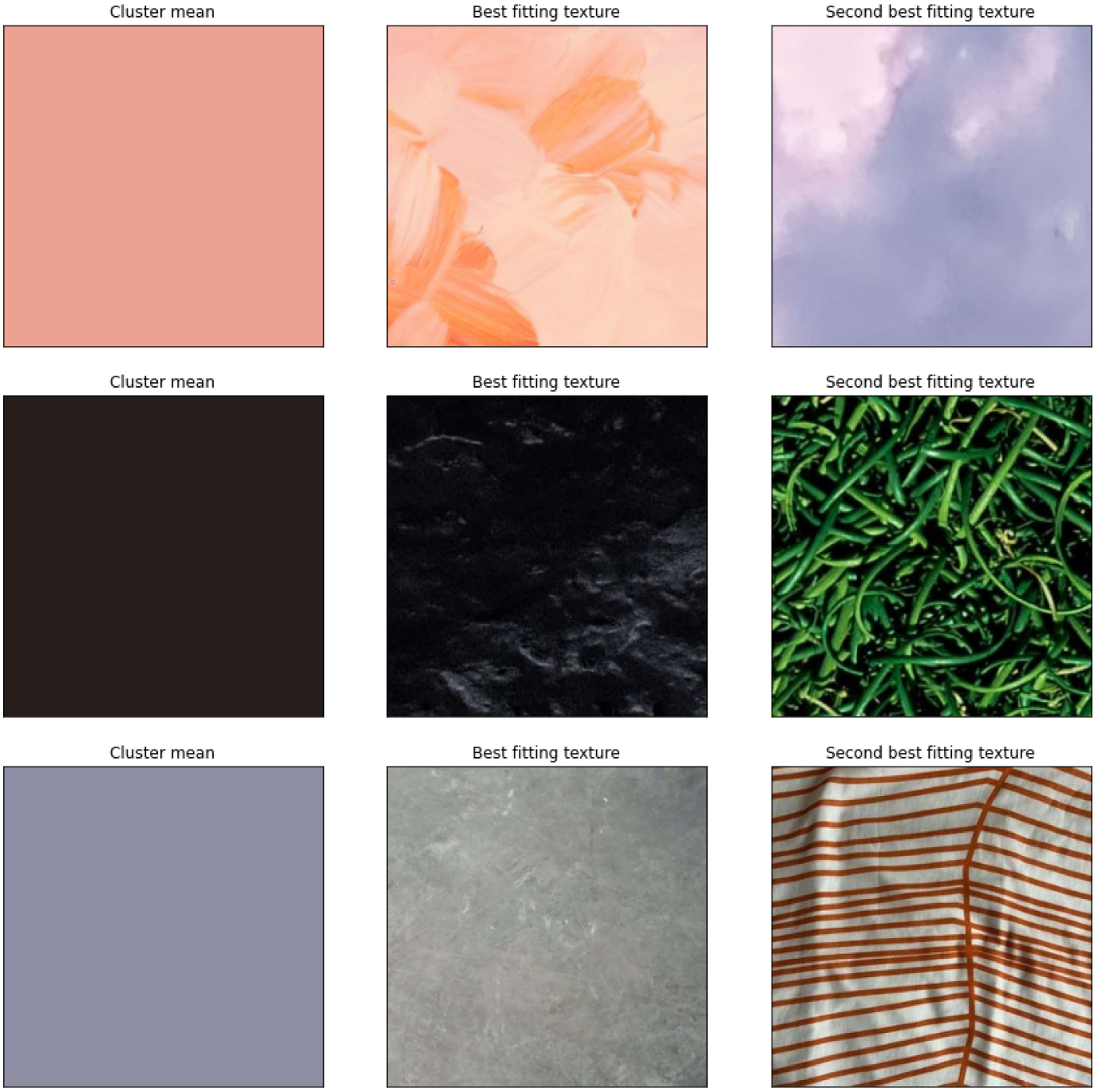
In [34]:
```
# define criteria, number of clusters(K) and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 3
ret, label, center = cv2.kmeans(Z,K,None,criteria,10,cv2.KMEANS_RANDOM_CENTERS)
label = label.flatten()
# Now convert back into uint8, and make original image
center = np.uint8(center)
res3 = center[label.flatten()]
res4 = res3.reshape((img2.shape))
label.resize(width, height)

# check outputs
print(label.shape)
print(center.shape)
print(center)
plt.figure(figsize = (16,9))
plt.imshow(res4[:, :, [2, 1, 0]])
plt.show()
```

```
(1440, 1080)
(3, 3)
[[143 161 234]
 [ 26  29  39]
 [163 142 143]]
```



In [35]:
```
# Displaying target color, best choice, second choice
for i in range(3):
    first_color = center[i]
    print(first_color)
    color_block = visualize_color(first_color)
    texture_index, texture_index_2 = color_to_texture_index_LAB(first_color, texture_colors_bgr)
    print(texture_index, texture_index_2)

    fig, axes = plt.subplots(1, 3, figsize=(15,15))

    axes[0].imshow(cv2.cvtColor(color_block, cv2.COLOR_BGR2RGB))
    axes[0].set_title('Cluster mean'), axes[0].set_xticks([]), axes[0].set_yticks([])
    axes[1].imshow(cv2.cvtColor(textures[texture_index], cv2.COLOR_BGR2RGB))
    axes[1].set_title('Best fitting texture'), axes[1].set_xticks([]), axes[1].set_yticks([])
    axes[2].imshow(cv2.cvtColor(textures[texture_index_2], cv2.COLOR_BGR2RGB))
    axes[2].set_title('Second best fitting texture'), axes[2].set_xticks([]), axes[2].set_yticks
```

```
[143 161 234]
color_to_texture_index_LAB: l 186
color_to_texture_index_LAB: a 153
color_to_texture_index_LAB: b 148
18 10
[26 29 39]
color_to_texture_index_LAB: l 30
color_to_texture_index_LAB: a 132
color_to_texture_index_LAB: b 132
0 7
[163 142 143]
color_to_texture_index_LAB: l 152
color_to_texture_index_LAB: a 133
color_to_texture_index_LAB: b 117
5 14
```



In [36]:
```
selected_textures = np.zeros((K, width, height, 3))
background_1 = cv2.imread('generated_simple/18.png', cv2.COLOR_BGR2RGB)
selected_textures[0] = background_1[0:width, 0:height, :]
background_2 = cv2.imread('generated_simple/0.png', cv2.COLOR_BGR2RGB)
selected_textures[1] = background_2[0:width, 0:height, :]
background_3 = cv2.imread('generated_seam/5.png', cv2.COLOR_BGR2RGB)
```

```
selected_textures[2] = background_3[0:width, 0:height, :]

input_im = res4

masks = []
```

In [37]:
```
for i in range(3):
    layer = getSegement(label, i)
    masks.append(getMask(layer))
    output_im = blend(selected_textures[i], input_im, masks[i])
    input_im = output_im
```

In [38]:
```
# change to unit8 to present original color
plt.figure(figsize = (16,9))
plt.imshow(np.uint8(input_im)[:, :, [2, 1, 0]])
```

Out[38]:
```
<matplotlib.image.AxesImage at 0x20c9d9b54b0>
```