Report

Basic function (Grayscale Photomosaic)

The process of Grayscale Photomosaic including:

1. Read the source image and the photo tiles.
2. Bilinear interpolation to resample (resize) the input image.
3. RGB to Grayscale.
4. Query for photo tiles.
5. Compose the photo tiles into source image.

Read the source image and the photo tiles:

The program will first read the image by `cv2.imread(canvas_path).astype(float)`, then it will generate a 2D-array[Height][Width], and store the RGB value. (Given)

Bilinear interpolation to resample (resize) the input image:

- The basic idea is we using bilinear interpolation to find the color of the new pixel (x,y in the picture below:), using the **closest four pixel**, finding their weighted average, then we can calculate the what is the color of the new pixel. Simply, it is summarizing the colour of the for nearest point.

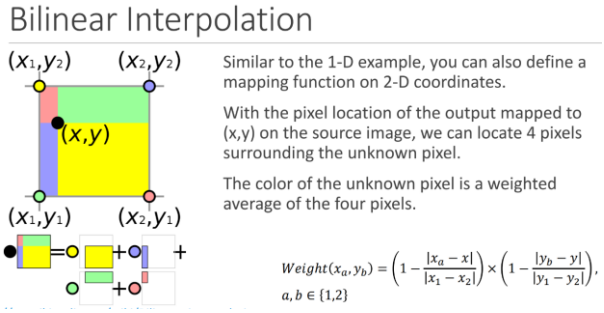


Figure 1: Bilinear Interpolation from Tutorial1

- The photomosaic is going to use a **similar photo to replace the new pixel**, the new pixel is coming from the resized image. To determined the resized pixel, we are going to find the new pixel from the deserved width and hight. By finding the ratio, we can know what is the cloested for pixel around the resized pixel.

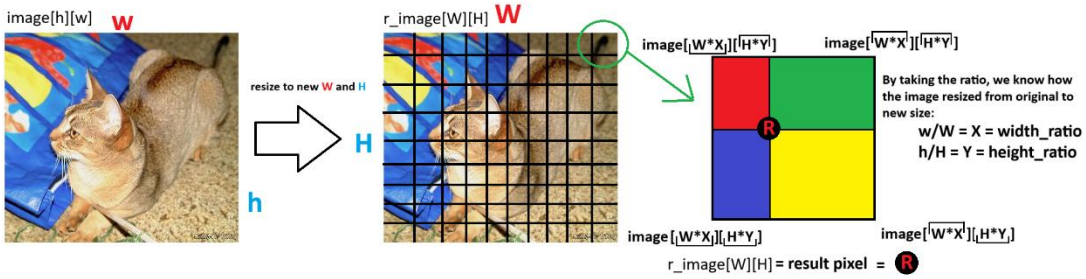


Figure 2: Process of the image resample.

- Special Case (the result pixel is landing on the edge)

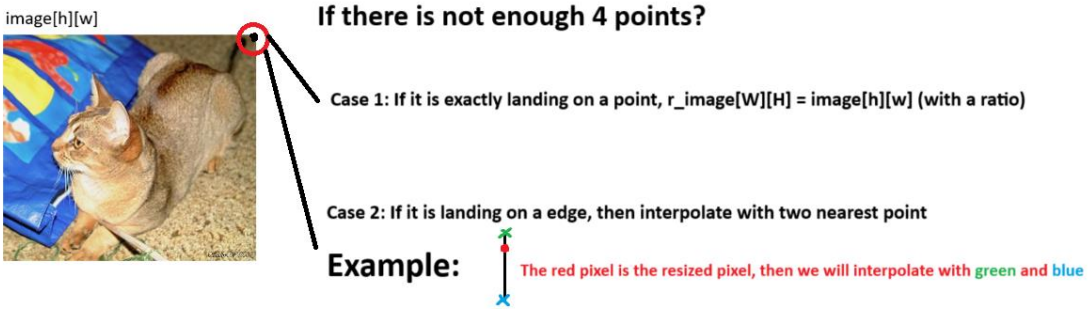


Figure 3: Special case handling.

When resizing the image using the bilinear interpolation, there can be cases where are not enough surrounding pixels to perform the interpolation at the boarders of the image, which may cause the total area to become zero at some calculation. In this case, we will going to use Nearest-neighbor interpolation to use the near point around the pixel.

- Key code

```
# find the floor and ceil value of the x and y
# if the value is out of the range (x < 0 or x > w), set it to the max/min value
# here will be use nearest neighbor interpolation
x_lower = max(min(floor(j * width_ratio), original_w - 1), 0)
x_upper = max(min(ceil(j * width_ratio), original_w - 1), 0)
y_lower = max(min(floor(i * hight_ratio), original_h - 1), 0)
y_upper = max(min(ceil(i * hight_ratio), original_h - 1), 0)

# get the value of the four points
p0 = image[y_lower][x_lower]
p1 = image[y_lower][x_upper]
p2 = image[y_upper][x_lower]
p3 = image[y_upper][x_upper]

# calculate the total area for weighted mean later on
total_area = (x_upper - x_lower) * (y_upper - y_lower)
```

Figure 4: Find the four nearby pixels.

```
# checking the total area
# if the total area is 0, there must be a point on the edge/it is a single point
# here will be use nearest neighbor interpolation or interpolate a line
if (total_area == 0):
    # checking the x-axis
    if (x_lower - x_upper == 0 and y_lower - y_upper != 0):
        # set the weight to 0 for p1 and p3
        w0 = (y_upper - i * hight_ratio) / (y_upper - y_lower)
        w1 = 0
        w2 = (i * hight_ratio - y_lower) / (y_upper - y_lower)
        w3 = 0
    # checking the y-axis
    elif (x_lower - x_upper != 0 and y_lower - y_upper == 0):
        # set the weight to 0 for p2 and p3
        w0 = (x_upper - j * width_ratio) / (x_upper - x_lower)
        w1 = (j * width_ratio - x_lower) / (x_upper - x_lower)
        w2 = 0
        w3 = 0
    else: # it is a single point, so it means p0 = p1 = p2 = p3
        # set the weight to 1 for p0 as we only take it
        w0 = 1
        w1 = 0
        w2 = 0
        w3 = 0
else: # normal case
    w0 = (x_upper - j * width_ratio) * (y_upper - i * hight_ratio) / total_area
    w1 = (j * width_ratio - x_lower) * (y_upper - i * hight_ratio) / total_area
    w2 = (x_upper - j * width_ratio) * (i * hight_ratio - y_lower) / total_area
    w3 = (j * width_ratio - x_lower) * (i * hight_ratio - y_lower) / total_area

# calculate the new pixel value
new_pixel = w0 * p0 + w1 * p1 + w2 * p2 + w3 * p3
```

Figure 5: Interpolation and special case handling.

- Result and performance

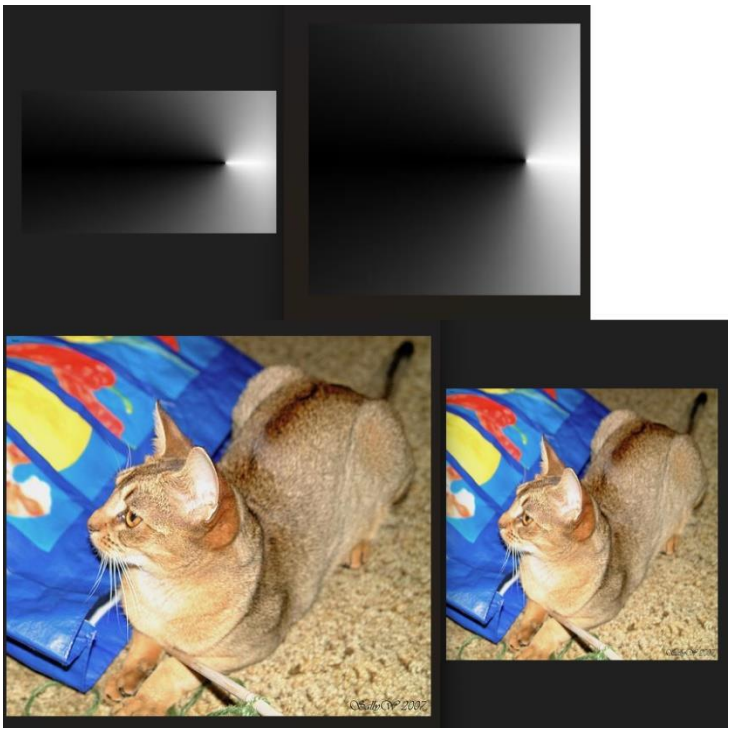


Figure 6: Result (Left is original image; Right is resized)

The time complexity: $O(W \times H), where\ W = width\ of\ image, H = height\ of\ image$

The execution time of resize a `cat.bmp(500x448)` to 320x320 is **4.886s**.

RGB to Grayscale and Image brightness:

To make the process of photomosaic easier, we will first convert the RGB image into grayscale as a grayscale display can carry out an amount of light in the pixel, for example,

10% grey = 90% brightness. [1] As a result, we can use the grayscale information to query the nearest brightness values.

- Convert RGB to Grayscale

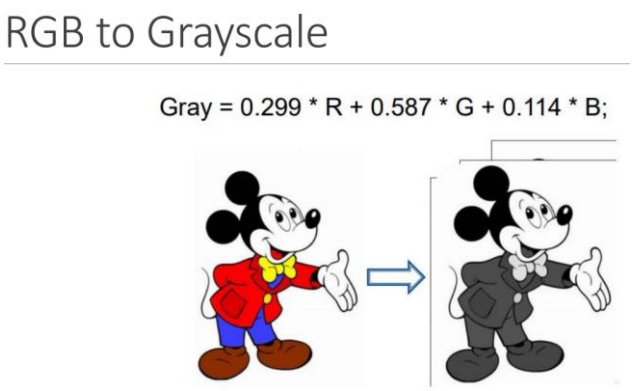


Figure 7: RGB to Grayscale equation.

Base on the tutorial slides, we simply convert it to temp.append(0.299 * pixel[i][0] + 0.587 * pixel[i][1] + 0.114 * pixel[i][2]), take the index 0-2 (the RGB information) and time the value, then we convert it to Grayscale.

- Summarization the brightness

To summarize the brightness of an image, we will create an array s[] to store the brightness of the image. The size of the array is equal to how many tiles that the original images can be divided into, which we have s[size] = size of image / size of tile (we assume it is always divisible). Then we map the original image pixel to the index of array s[] then divided by size of tile, we can get the brightness. The average brightness of tile = sum of pixel / size of tile.

For the index mapping, the index of s[] will represent from the left to right for each row, and top to down for each column, and the image pixel to tile will use the formula:

```
tileNumber = int(y / hight) * numOfTileInRow + int(x / width)
```

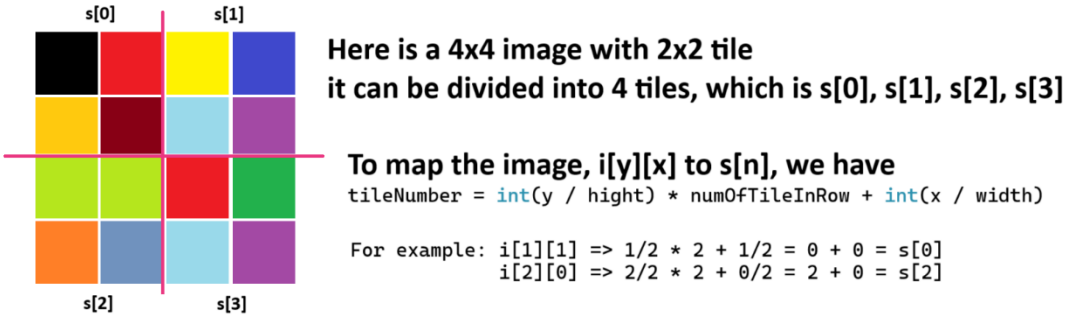


Figure 8: How to map to s[n].

Query for photo tiles with nearest brightness value:

- Sorting the tiles set according to the brightness value:

```
# Sort the tile_brightness array based on the brightness value
tile_brightness.sort(key=lambda x: x.brightness)
```

We use the Python build-in sort function to sort the array list for later we will use binary

search for easier to find the nearest brightness value from the tiles pool.

- Use Binary search for query the nearest brightness value:

The binary search algorithm is below:

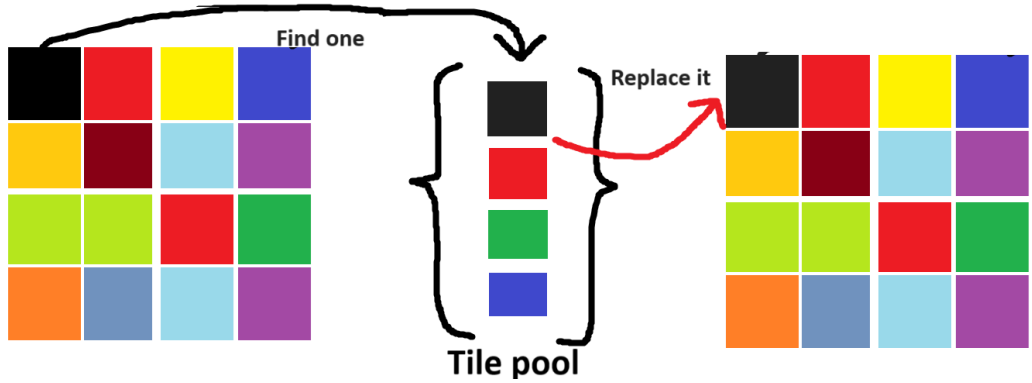
```
    while left < right:
        mid = int((left + right) / 2)
        if tilePool[mid].brightness < brightness:
            left = mid + 1
        elif tilePool[mid].brightness > brightness:
            right = mid - 1
        elif tilePool[mid].brightness == brightness:
            return tilePool[left].image
```

In binary search, we will get the nearest left pointer (index) which are close to the brightness value (the key). For example, we have the following brightness value in the below:

1	3	4	5

If our key is 3.5, it will give us 2 in that case. However, if the key is 3.1, it also returns 2, the correct nearest brightness value should be 1. So, before returning, it will have a checking to verify the result: `if tilePool[left].brightness - brightness < brightness - tilePool[left - 1].brightness:`

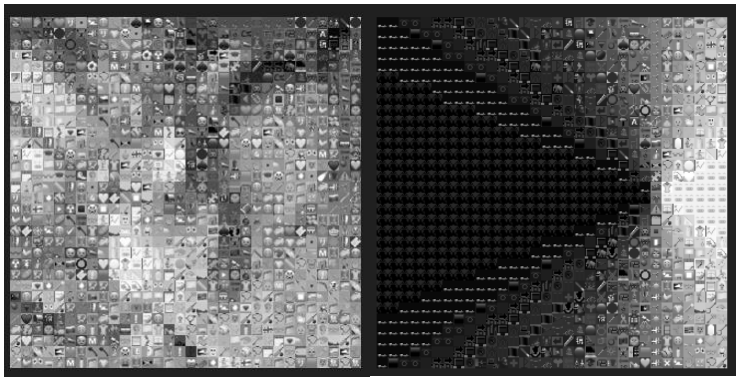
- Compose the output image with photo tiles:




```
def ComposeTiles(canvas:GrayImage, tiles, width, hight):
    # get the number of tiles in a row and a column
    numberOfTileInRow = int(len(canvas.image[0]) / width)
    numberOfTileInCol = int(len(canvas.image) / hight)

    # for every tile, find the nearest brightness tile
    for i in range(len(canvas.brightness)):

        # find the nearest brightness tile
        nearest_tile = FindNearestTiles(canvas.brightness[i], tiles)

        # compose the tile to the canvas
        for y in range(hight):
            for x in range(width):
                # map the tile index to the canvas
                iy = int(i / numberOfTileInCol) * hight + y
                ix = (i % numberOfTileInRow) * width + x

                # compose the tile to the canvas
                canvas.image[iy][ix] = nearest_tile[y][x]
```

```
    return canvas
```

The idea of compose tiles is to copy every tile pixel into the image, we simply map the tile index to the canvas. And we can get the result below:



Code implementation and performance:

The whole process of photomosaic is below:

```
def photomosaic(canvas, tiles, W, H, w, h):

    # resize the canvas to the target width and height
    canvas = bilinear_interpolation(canvas, W, H)

    # resize the tiles to the target width and height
    for i in range(len(tiles)):
        tiles[i] = bilinear_interpolation(tiles[i], w, h)

    # convert the image to gray scale
    gray_canvas = image2brightness(canvas, w, h)

    # create a empty tile to store gray scale tiles
    tile_brightness = []

    # convert the tiles to gray scale
    for i in range(len(tiles)):
        tile_brightness.append(tile2brightness(tiles[i]))

    # Sort the tile_brightness array based on the brightness value
    tile_brightness.sort(key=lambda x: x.brightness)

    # compose the result image
    gray_canvas = ComposeTiles(gray_canvas, tile_brightness, w, h)

    return gray_canvas.image
```

- For easier processing the image data, a new data type `GrayImage` is created for later operation.

```
class GrayImage:
    def __init__(self, image, brightness):
        self.image = image
        self.brightness = brightness
```

- In the bilinear resize function, `def bilinear_interpolation(image, width, hight)` will return a resized image (width X hight). Since it will process every pixel
  The time complexity will be: $O(W \times H), W = width\ of\ image, H = height\ of\ image$

- In the RGB to Gary function, `def image2brightness(canvas, width, hight) -> GrayImage:` and `def tile2brightness(tiles) -> GrayImage:`, they will convert every pixel into Gary and then calculate the average brightness
  The time complexity will be: $O(W \times H), W = width\ of\ image, H = height\ of\ image$
- In query the nearest brightness tile function, `def FindNearestTiles(brightness, tilePool):`, it simply using the binary search to find. Also the tile is pre-sorted using Python build-in sort function, where the time complexity of the function is $O(nlog(n))$.
  The overall time complexity will be: $O\big(\log(n) + nlog(n)\big), n = number\ of\ tiles$
- At last, `ComposeTiles(canvas:GrayImage, tiles, width, hight)`, is copying the tile pixel into original image.
  The time complexity will be: $O(W \times H), W = width\ of\ image, H = height\ of\ image$
- Program runtime: 

(photomosaic on the cat.bmp with W = H = 320, w = h = 10)

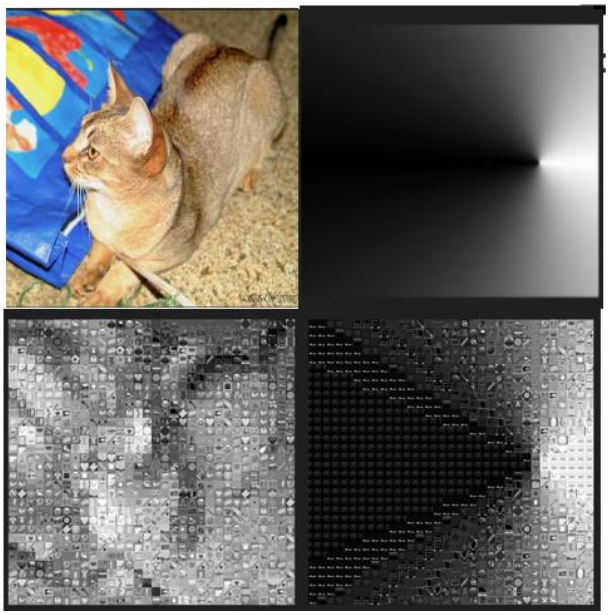


Figure 9: Result.

Enhancements

Image padding for edge: [2]

In Bilinear interpolation, we simply use nearest neighbourhood for the edge case. It is simple and work for bilinear interpolation as it at most only consider 4 pixels. However, in bicubic interpolation, we are going to use 16 pixels to resample, just simply use nearest neighbourhood is not a good method. Therefore, before doing Bicubic interpolation, we will be going to pad the image in the edge by using replicate padding method.

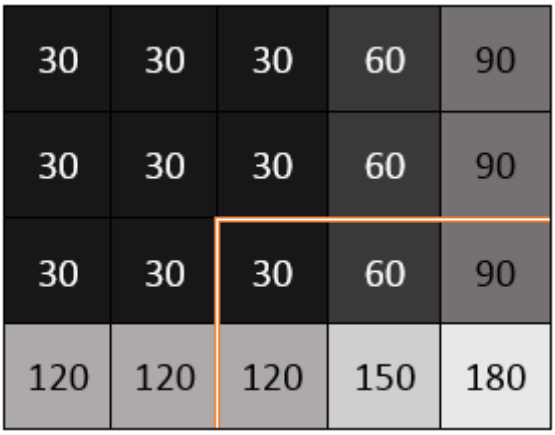


Figure 10:Replicate padding.

- Code implementation and result:

```python
# replicate padding for the image
def replicate_padding(image):
    # get the original width and height
    h = len(image)
    w = len(image[0])

    # create a new image with the target width and height
    padded_image = []

    # replicate padding
    for i in range(h + 4):
        temp = []
        for j in range(w + 4):
            # replicate padding
            if i < 2: # top
                y = 0
            elif i >= h + 2: # bottom
                y = h - 1
            else:
                y = i - 2 # middle (copying the original img)

            if j < 2: # left
                x = 0
            elif j >= w + 2: # right
                x = w - 1
            else:
                x = j - 2 # middle (copying the original img)

            temp.append(image[y][x]) # copy the pixel value
        padded_image.append(temp) # append the new row to the result

    return padded_image
```

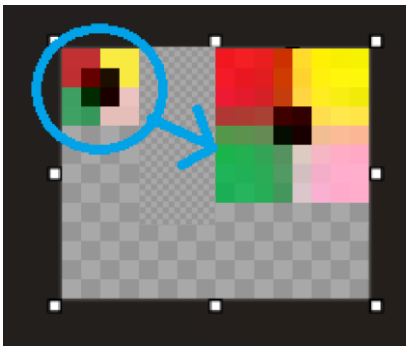


Figure 11: Code and result (left is original image, right is padding version)

Bicubic interpolation to resample (resize) the input image: [3] [4] [5]

Similar on using bilinear interpolation, we will be going to use bicubic interpolation to resample the input image.

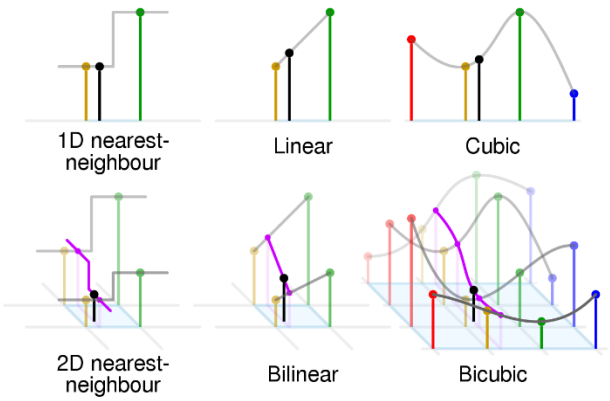


Figure 12: Comparison of different interpolation. [4]

From the above comparison, in the Bilinear interpolation, we are using 3 linear interpolations to get the result, **which mean we used 4 pixels to resample one pixel**. On the other hand, similarly, in bicubic interpolation, we will use 16 pixels to resample one pixel, where the value of pixel p can be written as $p(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j$.

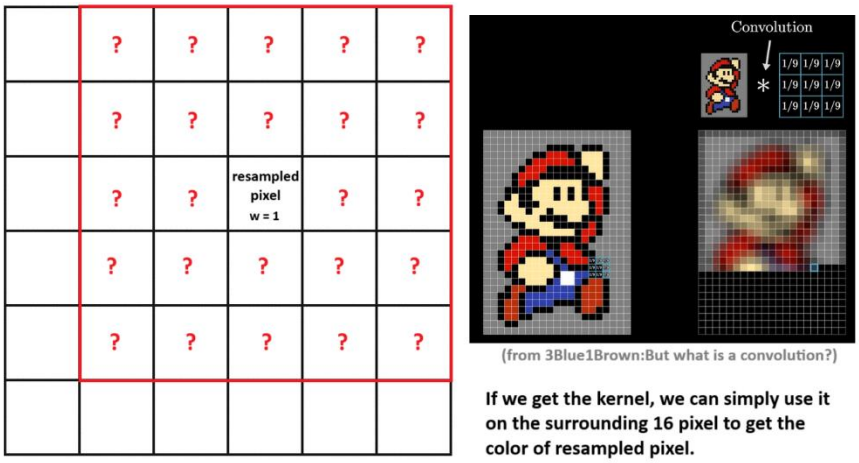


Figure 13: How the kernel to find the resample pixel. [6]

Since we only consider the 16 closest pixels around the pixel p and each pixel distance is one unit far, finding the value of pixel p is actually taking a weighted average w_{ij} for the 16 closest pixels. For example, if the resampled pixel located in the exactly the same position in the original image, the weight of the position is 1. At the same time, it also means that the distance between the resampled pixel and source pixel will determine the weight, therefore, the implementation of bicubic interpolation can be formulated as a convolution. We can find a predefined kernel and we can know the weight of each the 16 closest pixels around the pixel p.

$$u(s) = \begin{cases} (a+2)|s|^3 - (a+3)|s|^2 + 1 & 0 < |s| < 1 \\ a|s|^3 - 5a|s|^2 + 8a|s| - 4a & 1 < |s| < 2 \\ 0 & 2 < |s|. \end{cases}$$

Figure 14: The interpolation kernel. [5]

Based on "Cubic Convolution Interpolation for Digital Image Processing" finding, we know that the kernel is the u(s), where s is the distance between resampled pixel and selected source pixel, a is a parameter controls the sharpness of the bicubic kernel, we will take a = -0.75 in here. And the implementation code is below:

```
# bicubic kernel equation
def cubic_kernel(distance, a:float):
    distance = abs(distance)
    if distance <= 1:
        return (a + 2) * (distance ** 3) - (a + 3) * (distance ** 2) + 1
    elif distance < 2:
        return a * (distance ** 3) - 5 * a * (distance ** 2) + 8 * a * distance - 4 * a
    else:
        return 0
```

As a result, the key steps of implementation of cubic interpolation are below:

1. Padding the image
2. Map the output coordinates to the mapped image (similar to Bilinear interpolation)
3. Get the x and y coordinate of the 16 points → Find dx and dy

```
# bicubic interpolation part
#     dx
#   |---------|
#   |--||-------------------------|
# dy|  ||p0    p1     p2     p3   | y0
#   |  ||p4    p5     p6     p7   | y1
#   |--||         R(x, y)         |
#      |p8    p9     p10    p11  | y2
#      |p12   p13    p14    p15  | y3
#      |------------------------ |
#        x0       x1     x2     x3
for h in range(hight):
    temp = []
    for w in range(width):
        # map the output coordinates to the mapped image
        x = (w + 0.5) * width_ratio + 1.5
        y = (h + 0.5) * hight_ratio + 1.5

        # get the x and y coordinate of the 16 points
        dx = []
        dx.append(int(x - 1))
        dx.append(int(x))
        dx.append(int(x + 1))
        dx.append(int(x + 2))

        dy = []
        dy.append(int(y - 1))
        dy.append(int(y))
        dy.append(int(y + 1))
        dy.append(int(y + 2))
```

4. Find the weight of the 16 points by interpolation kernel.

```
# find the weight of the 16 points
w = []
for i in range(4):
    for j in range(4):
        w.append(cubic_kernel(x - dx[i], -0.75) * cubic_kernel(y - dy[j], -0.75))
```

5. Calculate the final pixel.

```
# calculate the pixel value
final_pixel = 0.0
for i in range(4):
    for j in range(4):
        final_pixel += w[i * 4 + j] * image[dy[j]][dx[i]]
```

Testing and result:

For the testing code, I modify the main.py to generate 3 png files, two images are generated by cv2 resize function, one is generated by my implementation.

```
# For enhancement features
binilnear = cv2.resize(canvas, (320, 320), interpolation =
cv2.INTER_LINEAR)
answer = cv2.resize(canvas, (320, 320), interpolation = cv2.INTER_CUBIC)
result = enhancements.photomosaic_Cubic(canvas, tiles, W, H, w, h)

cv2.imwrite('synthesisBilinear.png', np.array(binilnear).astype(np.uint8))
cv2.imwrite('synthesis.png', np.array(result).astype(np.uint8))
cv2.imwrite('synthesisCV2.png', np.array(answer).astype(np.uint8))
```

Comparing to my implementation and cv2.resize with `Cubic`, the photos are generated mostly the same.

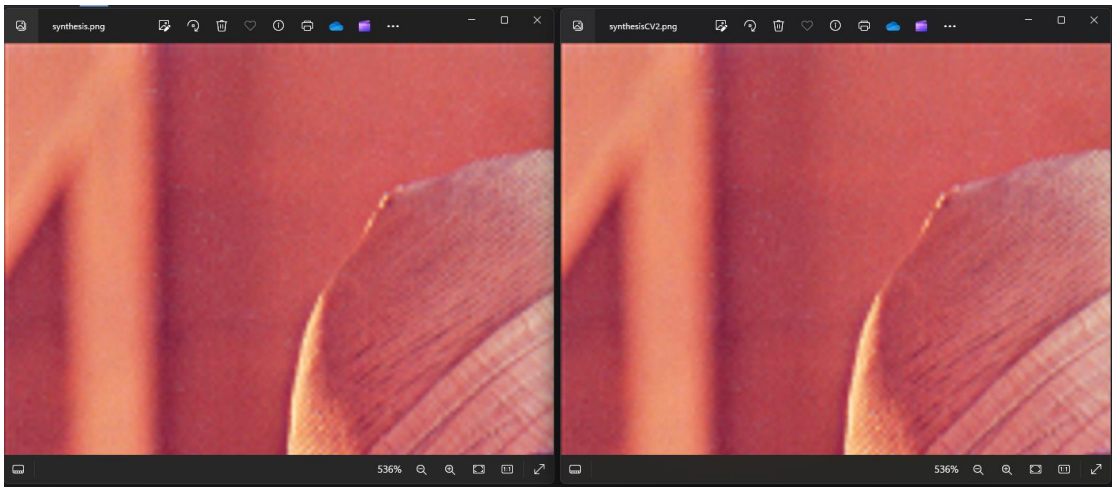


Figure 15: Result compare between self-implementation and cv2 version (lena.bmp)

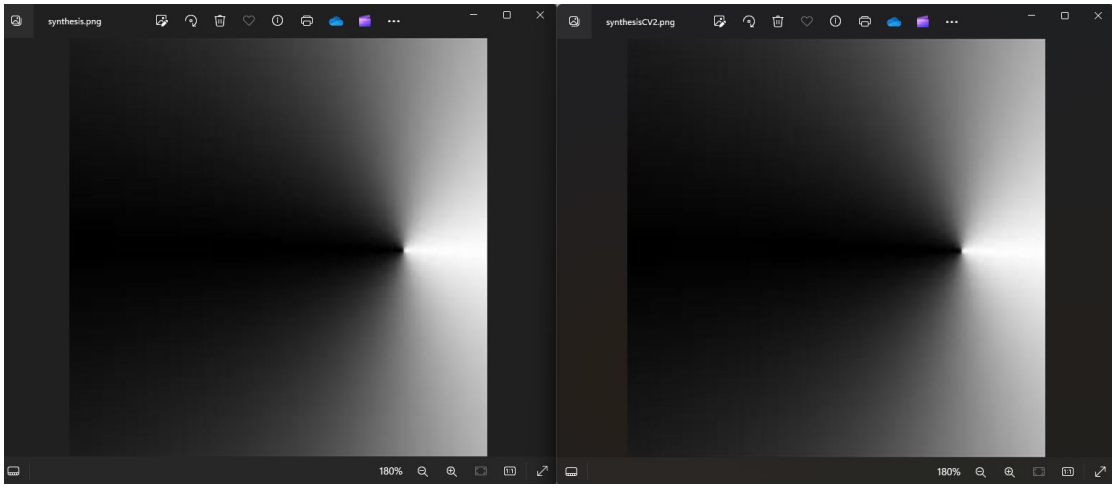


Figure 16: Result compare between self-implementation and cv2 version (bandingTest.bmp)

Their pixel array almost sharing the same value, but there are some floating point errors, it may related to my floating calculation in my implementation.

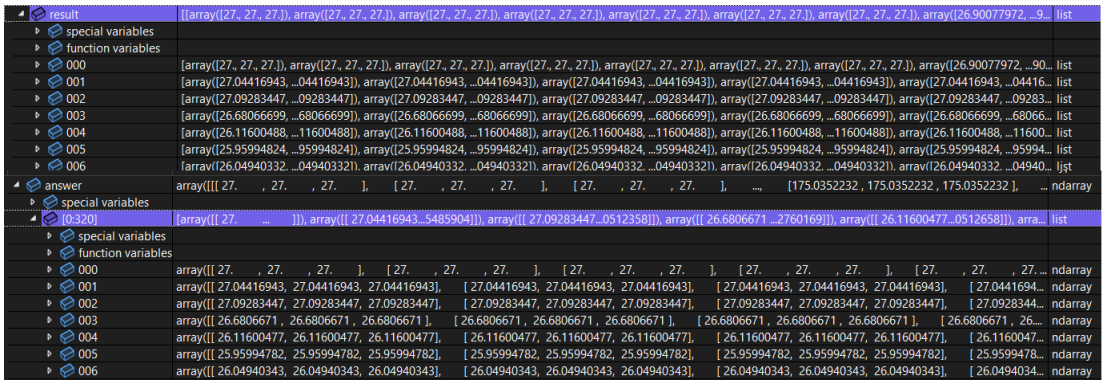


Figure 17: The detail pixel information between self-implementation and cv2 version (bandingTest.bmp)

After using the Cubic interpolation, the image is getting better then bilinear interpolation. The resampled image is showed clearer and more detailed than Bilinear.

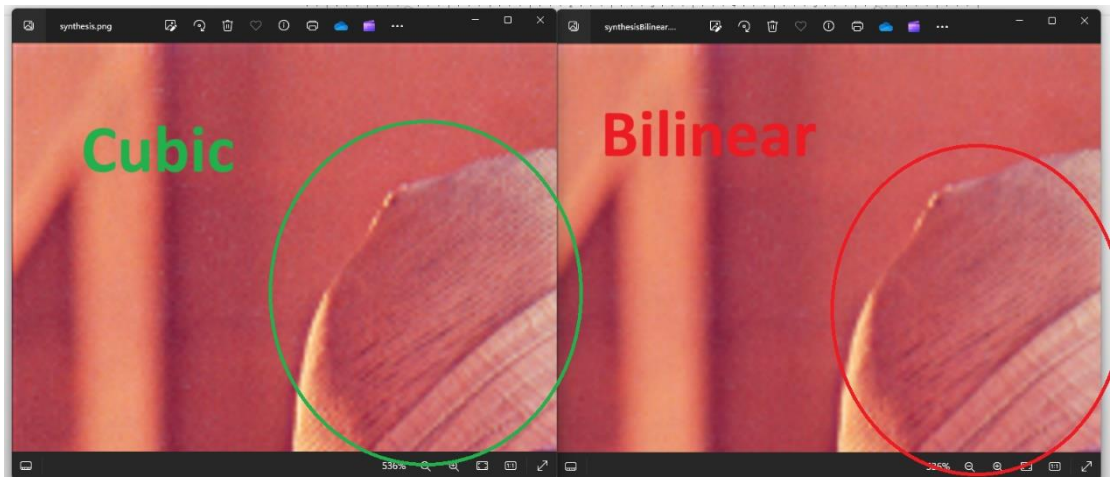
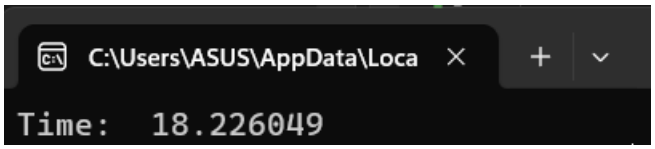


Figure 18: Comparing to Cubic and Bilinear interpolation.

However, the execution is much heavier than Bilinear interpolation. In pervious testing, the execution time of resize a `cat.bmp(500x448)` to 320x320 is **4.886s**. However, in cubic interpolation, the execution time of resize a `cat.bmp(500x448)` to 320x320 is **18.226s**. The reason why it spends more time because cubic is using more pixel than Bilinear (16 vs 4).



Ordered Dithering: [7]

The idea of ordered dithering is to display a continuous image on a display of smaller color depth, which can see clearly the information in the picture.

Figure 19:Dithering example. [7]

In my currently implementation, I am using the average color of the pixels and select the most similar tile to replace it. However, if we resample the image or the contrast of the image is not obvious, it may not get a suitable tile to replace it. (see the figure below)

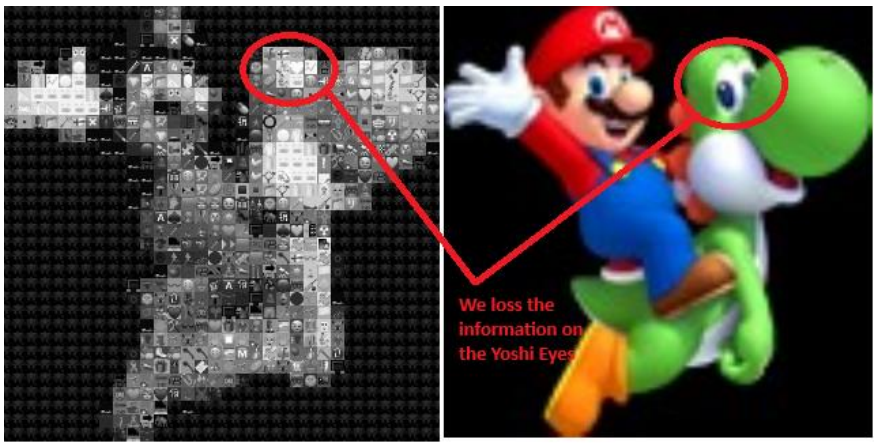


Figure 20: Current implementation, the photomosaic is loss the Yoshi's eyes information.

Idea of the algorithm is reduce the number of colors by applying a threshold map M to the pixels displayed, causing some pixels to change color, depending on the distance of the original color from the available color entries in the reduced palette. [7]

The implementation of Ordered Dithering below:

1. Finding the Threshold map, simply follow the equation:

$$\mathbf{M}_{2n} = \frac{1}{(2n)^2} \times \begin{bmatrix} (2n)^2 \times \mathbf{M}_n & (2n)^2 \times \mathbf{M}_n + 2 \\ (2n)^2 \times \mathbf{M}_n + 3 & (2n)^2 \times \mathbf{M}_n + 1 \end{bmatrix}$$

```
# creating a Bayer matrix for dithering
def BayerMatrix(n):
    # define the base case 2x2 matrix
    base = [[0, 2],
            [3, 1]]

    while n > 2:
        # create a new matrix with 4 times the size of the original matrix
        new_base = []
        for i in range(2 * len(base)):
            temp = []
            for j in range(2 * len(base)):
                temp.append(0)
            new_base.append(temp)

        # copy the original matrix to the new matrix
        for i in range(len(base)):
            for j in range(len(base)):
                new_base[i][j] = 4 * base[i][j]
                new_base[i][j + len(base)] = 4 * base[i][j] + 2
                new_base[i + len(base)][j] = 4 * base[i][j] + 3
                new_base[i + len(base)][j + len(base)] = 4 * base[i][j] + 1

        # update the base matrix
        base = new_base
        n = n / 2

    return base
```

2. Compare the selected pixel and the corresponding location in the matrix, set it to 255 (white) if the color is larger than it, else set it to 0. Then replace with the new color.

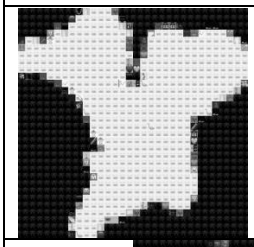
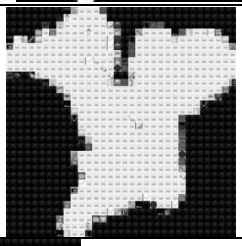
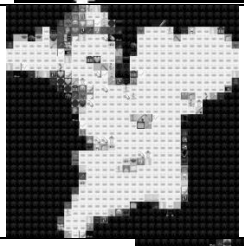
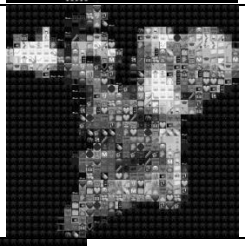
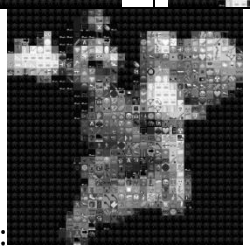
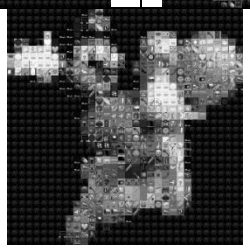
```
# dithering process
for y in range(h):
    temp = []
    for x in range(w):
        # get the pixel value
        pixel = canvas.image[y][x]

        # get the threshold value
        threshold = bayer[y % bayerSize][x % bayerSize]

        # compare the pixel value with the threshold value
        if pixel > threshold:
            temp.append(255)
        else:
            temp.append(0)
    result.append(temp)

return result
```

3. And get the output result, re-calculate the average brightness.

Result and performance:

To obtain different level of dithering, we can set the n in the function `def BayerMatrix(n):`

After trying different level of dithering, we get different performance, one important thing is we **<u>can see the eyes of Yoshi very clearly</u>**.

n = 2	n = 4	n = 8	n = 16
			
			
Original: 		Cubic without dithering: 	

For overall testing, we use the time.time() – now to get the running time, and here is the running time:

```
# implement the function `photomosaic`
start_time = time.time()
synthesis = photomosaic(canvas, tiles, W, H, w, h)
print("Time on basic part:%s seconds" % (time.time() - start_time))
start_time = time.time()
# For enhancement features
# for testing result
# binilnear = cv2.resize(canvas, (320, 320), interpolation = cv2.INTER_LINEAR)
# answer = cv2.resize(canvas, (320, 320), interpolation = cv2.INTER_CUBIC)
synthesisCubic = enhancements.photomosaic_Cubic(canvas, tiles, W, H, w, h)
print("Time on cubic part:%s seconds" % (time.time() - start_time))
start_time = time.time()
synthesisDithering = enhancements.photomosaic_dithering_bilinear(canvas, tiles, W, H, w, h, 16)
print("Time on dithering_bilinear part:%s seconds" % (time.time() - start_time))
ditheringOnly = enhancements.ditheringOnly(canvas, 16)
```

```
C:\Users\ASUS\AppData\Loca
Time on basic part:7.124475479125977 seconds
Time on cubic part:26.471630573272705 seconds
Time on dithering_bilinear part:7.235790729522705 seconds
Press any key to continue . . .
```

Reference

[1] T. Jeweet, "Tom Jewett's Color Tutorial - Grayscale," Department of Computer Engineering and Computer Science, Emeritus, 2017.

[2] A. Kamran, "Types of padding in images," educative, [Online]. Available: https://www.educative.io/answers/types-of-padding-in-images. [Accessed 14 2 2024].

[3] "Stack Overflow - imresize - trying to understand the bicubic interpolation," Stack Overflow, 12 2014. [Online]. Available: https://stackoverflow.com/questions/26823140/imresize-trying-to-understand-the-bicubic-interpolation. [Accessed 14 2 2024].

[4] "Bicubic interpolation," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Bicubic_interpolation. [Accessed 14 2 2024].

[5] R. G. KEYS, "Cubic Convolution Interpolation for Digital Image," *IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, VOL. ASSP-2,* 1981 .

[6] *But what is a convolution?.* [Film]. 3Blue1Brown, 2022.