Comp Photography
Final Project

Namit Gupta

Summer 2016

ngupta40@gatech.edu

Photomosaic

A photomosaic is a picture that has been divided into tiled sections, each of which is replaced with another photograph that matches the target photo. When viewed at low magnifications, the individual pixels appear as the primary image, while close examination reveals that the image is in fact made up of many hundreds or thousands of smaller images.

(https://en.wikipedia.org/wiki/Photographic_mosaic)

The Goal of Your Project

The goal of this project was to architect and develop a software system which can be fed with thousands of sample images (i_1, i_2….i_n) and those images be used later to develop a photomosaic out of a given input image (IMG_{in}). The output image (IMG_{out}) is made up by dividing the input image into square shaped tiles of equal dimensions and for each tile, the average color (RGB values) of the tile was matched with the average color of each sample images and the tile which matched closest to the sample image in terms of root square mean error would be replaced by the sample images.

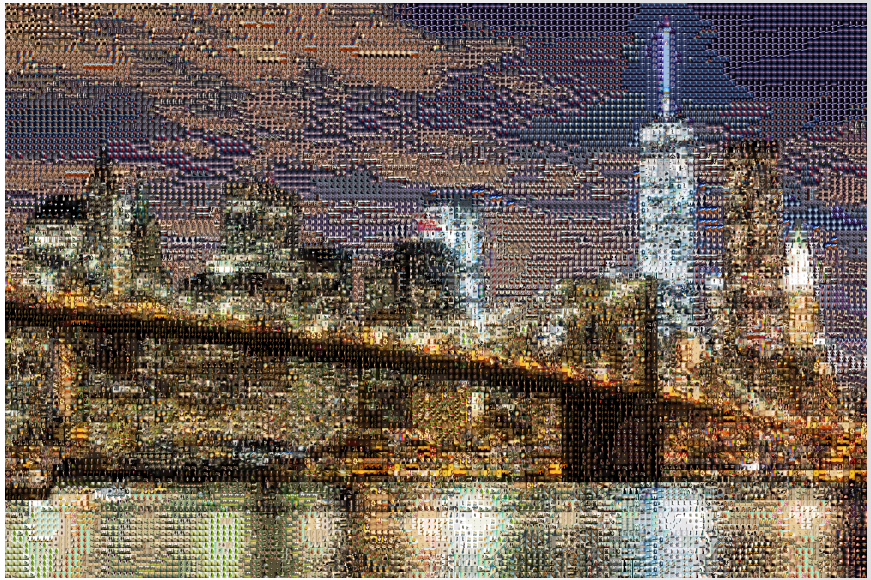
Scope Changes

The original scope as envisioned by myself included developing a system which could fetch sample images from the internet (google images or picasa or imgur) automatically at startup based on the image category specified by the user. This way, a user could simply specify the category (eg – plants, animals, techno etc.) of images that should be used as tiles to create the photomosaic and the software would handle fetching the required images from the web.

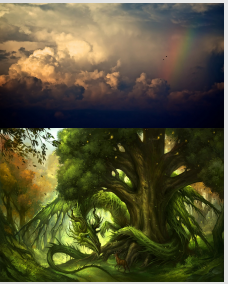
Instead, When I tried implementing this in code, I realized that the set of sample images downloaded from the web was not high quality. Due to time constraints, I decided to download the sample images from google images and use them for this project.

Input Image

Output Photomosaic Image



Sample Tile images



Zoomed version of output image

Your Pipeline

Download 1000 images from google images

Load the downloaded images in the code on startup and compute the average RGB values of each image and store it in a dictionary

The input image for which we would like to create a photomosaic is now taken and resized to a square shape with dimensions closest to the original dimensions

Split the input image into n tiles of square shape and dimensions which were entered by the user

Iterate through each of the tiles in the input image and compute the average RGB value for each tile. Once the average is computed, compare this value to the average RGB values of sample images (computed in step 2 above) and a winner is declared based on the closest match (using root mean square computation)

Compress the winner image to the size of a tile and replace the tile in input image with that of the winner image.

OUTPUT

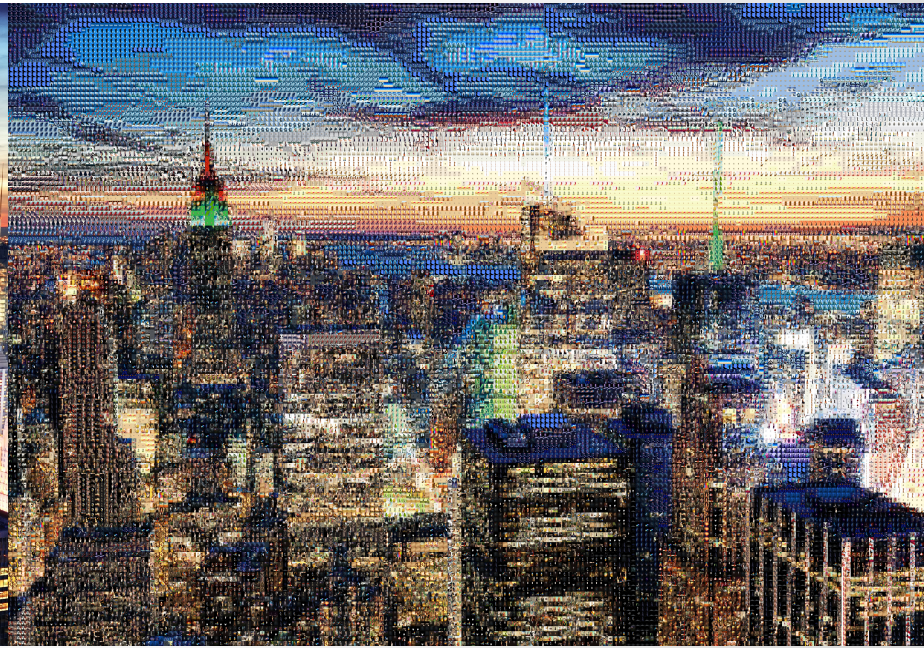
Once the iteration through each tile is complete and the winner images are replaced with each tile, the final image is the resultant photomosaic

Demonstration: Show complete Input/Output results

The following two pages show three different photo mosaics that my software developed. The source code and sample images used to develop these photo mosaics could be checked using the following github link: https://github.com/nmtgpta/CompPhoto/tree/master/Results

Input Image

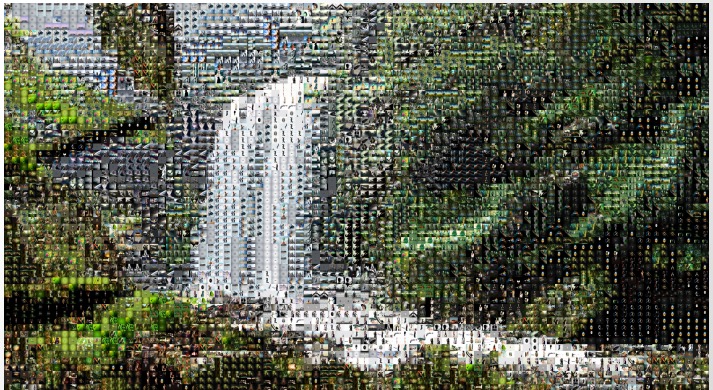
Output Photo mosaic



Demonstration: Show complete Input/Output results

Input Image

Output Photo mosaic



Computation: Project Development

The following command line expression is used to execute the software:
python mosaic.py Input.jpg <tile_size> Output.jpg ./Library

The above command calls the main function in mosaic.py with a few parameters as described below:

1) Input.jpg → The input image for which we want to create a photomosaic.
2) tile_size → An integer which specifies the dimensions of a square tile that will be used to replace tiles in Input.jpg with sample images.
3) Output.jpg → The name of the final photomosaic after the computation is finished.
4) Library → The name of the folder which contains all the sample images which will be used as tiles in the photomosaic.

For example, To create a photomosaic with each tile dimension of 35 x 35 pixels, the following command will work:
python mosaic.py Input.jpg 35 Output.jpg ./Library

Computation: Project Development

Let us assume that we would like to create a photomosaic with tile dimensions of 35 x 35 and we pass this argument as we initiate the mosaic.py. It is assumed that we have already downloaded hundreds (or thousands) of pictures from the web in the **Library** folder.

As the code executes for the first time, it creates a dictionary of all the images in Library folder by loading them iteratively and computing their average RGB values and storing this information as a (key, value) pair in the dictionary. The code which accomplishes this task is as follows:

```
for img in images:
        Libimage = cv2.imread(img)
        Libimage = cv2.resize(Libimage, (int(Tile),int(Tile)),
interpolation=cv2.INTER_AREA)
        Red, Green, Blue = FindAverage(Libimage)
        Library[img, Red, Green, Blue] = Libimage
```

Computation: Code Explanation

Definition of FindAverage():

```
def FindAverage(image):
        red_hist = cv2.calcHist([image],[2], None, [256],[0,256])
        grn_hist = cv2.calcHist([image],[1], None, [256],[0,256])
        blu_hist = cv2.calcHist([image],[0], None, [256],[0,256])

        sum = 0
        for i in range(0,len(red_hist)):
                sum = sum + (int(red_hist[i])*i)
        Red = sum / (image.shape[0]*image.shape[1])

        sum = 0
        for i in range(0,len(grn_hist)):
                sum = sum + (int(grn_hist[i])*i)
        Green = sum / (image.shape[0]*image.shape[1])

        sum = 0
        for i in range(0,len(blu_hist)):
                sum = sum + (int(blu_hist[i])*i)
        Blue = sum / (image.shape[0]*image.shape[1])

        return Red, Green, Blue
```

The above function calls the cv2.calcHist() to fetch the histogram of a given sample image and later figures out the average value of red, green and blue channels by iterating in a for loop and returning these average values to the calling function. It is the centerpiece of this algorithm.

Computation: Code Explanation

Once the average RGB values of all sample images is stored in a dictionary, we will start playing with the actual input image for which the photomosaic needs to be constructed.

First and foremost, the input image needs to be cropped based on the dimensions of the tile so that the number of tiles used to fill up the input image is a whole number. We do so by calling the following function:

```
InputImg = InputImg[0:InputImg.shape[0] - InputImg.shape[0]%int(Tile), 0:InputImg.shape[1] - InputImg.shape[1]%int(Tile)]
```

For example, if the input dimensions of an image was 702 x 355 pixels, the above transformation (for tile size 35 x 35 pixels) will crop the input image to dimensions of 700 x 350 pixels so that a total of 200 tiles could be used to make up the final image.
Once we know for a fact that 200 tiles are needed to create a photomosaic of the input image, we iterate through the entire region of input image 200 times and for each tile location, an image is selected to replace those pixels. For each tile location, FindAverages() is called to fetch the average RGB values of the pixels in this region. Once this is done, the average RGB values are compared against the average RGB values computed earlier from sample images using mean square algorithm and the sample image which happens to offer the smallest magnitude of difference in RGB pixels with the tile area under consideration wins and the area is replaced with that tile. Finally, Output.jpg is written to disk. The code on the next page accomplishes this task.

Computation: Code Explanation

Definition of ComputeDifferences(): (Explanation of this function on previous page)

```
def ComputeDifferences(InputImg, Library, Tile, OutputImageFile):

        rows = InputImg.shape[0]
        cols = InputImg.shape[1]

        for i in range(0, rows/Tile):
                for j in range(0, cols/Tile):
                        SingleTileImg = InputImg[i * Tile:(i + 1) * Tile, j * Tile:(j + 1) * Tile]
                        Red_Tile, Green_Tile, Blue_Tile = FindAverage(SingleTileImg)

                        heap = []

                        for key in Library.keys():
                            Red_lib = key[1]
                            Green_lib = key[2]
                            Blue_lib = key[3]

                           MeanSquare = pow((Red_lib - Red_Tile), 2) + pow((Green_lib - Green_Tile), 2) + pow((Blue_lib - Blue_Tile), 2)
                                    heapq.heappush(heap, (MeanSquare, key))

                        InputImg[i * Tile:(i + 1) * Tile, j * Tile:(j + 1) * Tile] = Library[heap[0][1]]


        cv2.imwrite(OutputImageFile, InputImg)
```

Details: What worked?

The software worked very well for input images of any dimension as long as the size of the tile was smaller than the dimension of the input image itself. I made an executive decision to crop the input image to comply with the dimensions of the tiles. There are other techniques which could have been applied such as resize() or border fill algorithm.

The output images were of very high quality when viewed under low magnification and as the magnification of the output image was increased, once was successfully able to view the individual tiles that made up the output image.

Details: What did not work? Why?

As I was playing with input images of various dimensions, I realized that if the input image was of low dimensions (pixel length x pixel width), the resultant image was of low resolution as well. As a result, I decided to use images of very high megapixel values (>15 MP) for this project and come up with acceptable results. The reason why high dimensional pictures performed better than low dimensional pictures is because high dimensional pictures contain more information to denote a given point in the image (by virtue of having more pixels portray that point). As a result, a tile of 35 x 35 pixels, which is clearly visible to a human eye will be a very small portion of a high dimensional image when viewed under low magnification. On the other hand, low dimensional images offer less freedom to experiment with the tile size.

Any additional details?

I also worked on using other algorithms from the cv2.compareHist() function to compute image similarity instead of the plain vanilla average RGB computation and its comparison. The different metrics that I played with are as follows:

CV_COMP_CORREL : Correlation
CV_COMP_CHISQR : Chi-Square
CV_COMP_INTERSECT : Intersection
CV_COMP_BHATTACHARYYA :Bhattacharyya distance

The code for this functionality can be found in mosaic_histogram_diff.py
Per my experiments, Chi-square algorithm seemed to work very well in order to compute the similarity between a given tile and a sample image. Although, the results were neither better or worse compared to average RGB algorithm.

Details: What would you do differently?

If I were to add more features into this system, I would have added a way to automatically download sample images from the web based on user preference for a category of images to be used to build a photomosaic. I did try to play with this but my function calls to fetch the images returned images of very low resolution. Ultimately, I decided to download high resolution images manually instead. It would have been nice to finish that part of the project to download high quality images from the web.

Resources

1) Google Images for all Input and sample images
2) http://docs.opencv.org/2.4/modules/imgproc/doc/histograms.html?highlight=comparehist
3) https://en.wikipedia.org/wiki/Photographic_mosaic
4) https://github.com/nmtgpta/CompPhoto
5) http://www.cs.virginia.edu/cs150/ps/ps1/

All the results are available to be downloaded from this page:
https://github.com/nmtgpta/CompPhoto/tree/master/Results

Your Code

The code is available on the following github location: https://github.com/nmtgpta/CompPhoto/blob/master/mosaic.py

The source is as follows:

```
import os
import cv2
import numpy as np
import sys
import scipy as sp
import glob
import heapq

def FindAverage(image):

        red_hist = cv2.calcHist([image],[2], None, [256],[0,256])
        grn_hist = cv2.calcHist([image],[1], None, [256],[0,256])
        blu_hist = cv2.calcHist([image],[0], None, [256],[0,256])

        sum = 0
        for i in range(0,len(red_hist)):
                sum = sum + (int(red_hist[i])*i)
        Red = sum / (image.shape[0]*image.shape[1])

        sum = 0
        for i in range(0,len(grn_hist)):
                sum = sum + (int(grn_hist[i])*i)
        Green = sum / (image.shape[0]*image.shape[1])
```

```
sum = 0
        for i in range(0,len(blu_hist)):
                sum = sum + (int(blu_hist[i])*i)
        Blue = sum / (image.shape[0]*image.shape[1])

        return Red, Green, Blue

def ComputeDifferences(InputImg, Library, Tile, OutputImageFile):

        rows = InputImg.shape[0]
        cols = InputImg.shape[1]

        for i in range(0, rows/Tile):
                for j in range(0, cols/Tile):
                        SingleTileImg = InputImg[i * Tile:(i + 1) * Tile, j * Tile:(j + 1) * Tile]
                        Red_Tile, Green_Tile, Blue_Tile = FindAverage(SingleTileImg)

                        heap = []

                        for key in Library.keys():
                                Red_lib = key[1]
                                Green_lib = key[2]
                                Blue_lib = key[3]
```

```
                                        MeanSquare = pow((Red_lib - Red_Tile), 2) + pow((Green_lib -
Green_Tile), 2) + pow((Blue_lib - Blue_Tile), 2)
                                        heapq.heappush(heap, (MeanSquare, key))

                                InputImg[i * Tile:(i + 1) * Tile, j * Tile:(j + 1) * Tile] = Library[heap[0][1]]


        cv2.imwrite(OutputImageFile, InputImg)


def main():
        if len(sys.argv) !=5:
                print "Invalid Arguments. Usage: python mosaic.py <InputImage.jpg> <TilePixels>
<OutputImage.jpg> <Path_To_Image_Library>"
                return

        InputImg = cv2.imread(sys.argv[1])
        Tile = sys.argv[2]
        OutputImageFile = sys.argv[3]
        LibraryPath = sys.argv[4]

        Library = {}

        images = glob.glob(LibraryPath + "/" + "*.jpg")
```

```
        for img in images:
                Libimage = cv2.imread(img)
                Libimage = cv2.resize(Libimage, (int(Tile),int(Tile)), interpolation=cv2.INTER_AREA)
                Red, Green, Blue = FindAverage(Libimage)
                Library[img, Red, Green, Blue] = Libimage

        InputImg = InputImg[0:InputImg.shape[0] - InputImg.shape[0]%int(Tile), 0:InputImg.shape[1] -
InputImg.shape[1]%int(Tile)]

        ComputeDifferences(InputImg, Library, int(Tile), OutputImageFile)

if __name__ == "__main__":
    main()
```