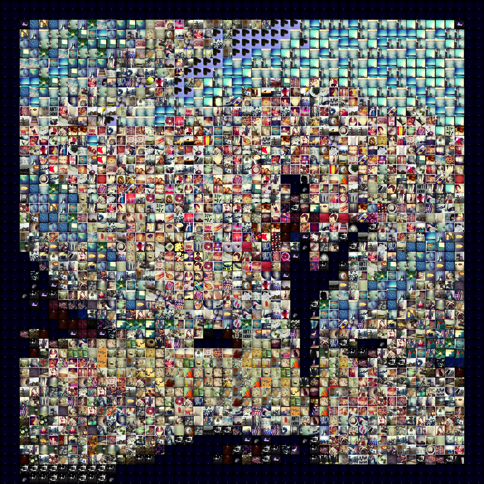
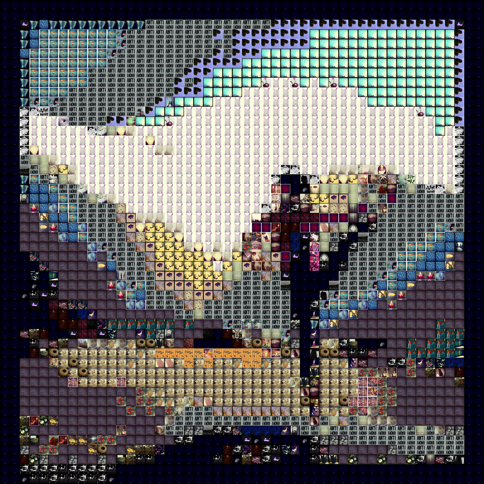
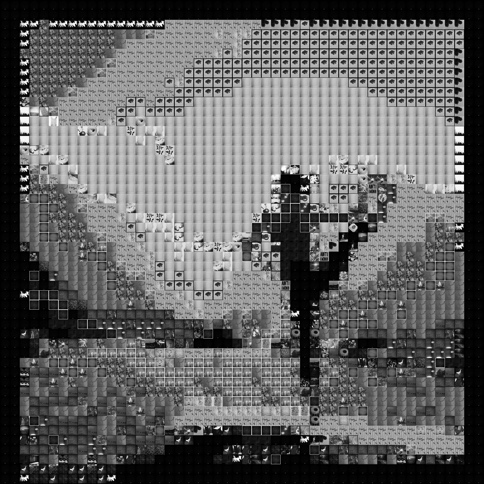
# A Preliminary Exploration of Generating A Mosaic Maker System

Final Project

jointly presented on:

5/13/2015 @ 11:30am

|  |  |  |  |
| --- | --- | --- | --- |
| by | Nina Baculinao  uni: nb2406 | & | Melanie Hsu  uni: mlh2197 |

for COMS W4735: Visual Interfaces to Computers

1. Introduction

The goal of the Mosaic Maker system is to generate a mosaic that resembles a base image when viewed from far away, in which each tile comes from a directory of tile images. It is a natural, practical, and (we hope) somewhat beautiful, extension of our individual Visual Information Retrieval systems, which sought to explore different ways of deciding the degree of similarities between images. The computational collage system in this case primarily looks at color similarity, using distance defined by the L1 norm as its matching heuristic, though we experimented with different measures of similarity to produce results with ranging aesthetic and performance characteristics. Finally, the system is evaluated through user studies in which human subjects are asked if they can guess what the mosaic represents and what features it holds. Their response reveal not only whether our mosaic system can indeed capture the big picture, and what details may have been lost from the original, but also are telling about human perception, the act of visual recognition, and the colorings of each person’s unique background contexts.

Images analyzed and used in the Tile library had the following properties:

* PNG format
* 150 x 150 pixels (square)

The base image typically didn’t have any restrictions on format or aspect ratio, although we sought higher resolution images than the tiles for producing mosaics in the user studies.

Hardware and library specifications of this project are as follows:

* MacBook Air 11 inch running on OSX Yosemite 10.10 (Nina)
* MacBook Pro 13 inch running on OSX Yosemite 10.10 (Melanie)
* Python Standard Library 2.7.9
* OpenCV with a Python binding, v2.4.10.1
* Python Imaging Library 1.1.7
* NumPy 1.9.1
* Matplotlib 1.4.2

# 2. Domain Engineering

# 3. Image Processing and Tile Matching

## Overview

A photomosaic is created by replacing each quadrant of pixels in some base image with a corresponding tile from an image bank. Each tile is selected to have properties similar to section of the base image it replaces, such that when the assembled mosaic is displayed at a coarse resolution (low-level zoom) it resembles the base image. Zooming in to finer resolutions reveals the individual tile images.

Our algorithm to match quadrants from the base image to the tile images in the database is described directly below, and in more detail following this general description.

1. For each image in the Tile database:
   1. Initialize as Tile object with image path and title as parameters
   2. Analyze the image
      1. Load as a Numpy array with OpenCV2 for analysis purposes
      2. Shrink to suitable tile width (30) and crop squarely
      3. Save new height and width
      4. Calculate its color histogram and grayscale histogram
   3. Prepare the image for display
      1. Load with PIL format for easily pasting into mosaic later
      2. Resize to desired display tile width (150) and crop squarely
   4. *Optional:*
      1. Visualize the histogram as a bar chart and save to disk
      2. Find list of most dominant colors in the tile image and later in main save a dictionary where key is color and value is list of tiles with that dominant color
   5. Add new Tile object to a dictionary in main where its key is its title
2. For the Base image:
   1. Initialize as Base object by passing in image path and title as parameters
   2. Analyze the image
      1. Load as Numpy array with OpenCV2
      2. Resize to desired width (tile width of 30, multiplied by the desired number of columns in mosaic)
      3. Save new height and width
      4. Calculate color histogram and grayscale for every tile-sized (30 \* 30) quadrant and append to “row list”, then append each “row list” to histograms list
      5. *Optional*: Find list of most dominant colors in the tile image
      6. Save number of rows and columns (where their product is the number of tiles needed to compose the image)
   3. Return Base object to main
3. For each quadrant histogram in the Base image:
   1. *Optional*: Dominant operation: Find best match by dominant color
   2. Expensive operation: Find “best match” by comparing quadrant histogram to *every single histogram* in the tile library, and using the tile with the least distance by L1 norm
   3. History operation: If this quadrant histogram has the same histogram as a previous quadrant, reuse the same “best match” tile

Running the Program

Our program accepted 3 command-line arguments after the name of the program:

1. Base image (path)
2. Tile directory (path)
3. Tile image format

A valid example of running our program could be:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| python | main.py | Low.jpg | \_db/justinablakeney | .png |

We chose to allow such specification in the command line, in order to test the results for different base images and determine a flexible tile directory for different kinds of images.

Initializing the Tile Objects

a. Data Reduction

A key component of visual processing is the the art of throwing away. Much of this happens in the domain engineering step, where we ensured that we only used square images from public Instagram accounts. However, we decided to take a few more data reductive steps in the processing of the tile database in order to ensure our system ran smoothly.

**Limiting Number of Images in the Tile Library**

First, if the user-specified directory containing tile images contained more than 500 images, we only added the first 500 images to our dictionary of tile objects for mosaic generation.

In main.py:

*# Parse command line args*

base\_path = sys.argv[1]

tile\_path = sys.argv[2]

format = sys.argv[3]

if os.path.exists(base\_path) and os.path.exists(tile\_path):

imfilelist = [os.path.join(tile\_path,f) for f in os.listdir(tile\_path) if f.endswith(format)]

num\_images = len(imfilelist)

tiles = {} *# init dictionary of tile objects*

if num\_images > 500:

num\_images = 500 *# only look up first 500 images*

for i in xrange(num\_images):

impath = imfilelist[i]

imtitle = entitle(impath, tile\_path, format)

tile = T.Tile(impath, imtitle)

tiles[imtitle] = tile

**Scaling Down Tile Images for Analysis**

Second, we chose to shrink all our tile images to square thumbnails that were 30 \* 30 pixels wide. We experimented with different values for tile width, and found this to be suitable for analytical purposes and notably faster than calculating histograms for the original tiles in our database, which were 150 \* 150 pixels wide.

However, when we zoomed in to view the tiles of our finished mosaics, we found it hard to discern the individual images in each 30 \* 30 tile, so we decided to differentiate between TILE\_WIDTH = 30 (for analytical purposes) and DISPLAY\_WIDTH = 50 (for display purposes). In short, while we grab histograms from tile images that are 30 \* 30 pixels wide, we later generate the mosaic using 50 \* 50 pixels wide images.

In tile.py:

import cv2

from PIL import Image

import reduction as R

import similarity as S

TILE\_WIDTH = 30

DISPLAY\_WIDTH = 50

**class** **Tile**():

**def** \_\_init\_\_(self, path, title):

*"""Open in Numpy array for histogram analysis"""*

self.path = path

self.title = title

size = (TILE\_WIDTH, TILE\_WIDTH)

self.image = cv2.imread(path, cv2.IMREAD\_UNCHANGED)

self.image = R.crop\_square(self.image, size)

self.height = len(self.image)

self.width = len(self.image[0])

self.histogram, self.image, self.colors = S.color\_histogram(self.image, self.title)

self.gray = S.grayscale\_histogram(self.image, self.title)

*"""Open with PIL format for display purposes"""*

self.display = Image.open(path)

self.display = R.resize\_square(self.display, (DISPLAY\_WIDTH, DISPLAY\_WIDTH) )

*"""Additional options (extra runtime)"""*

self.dominants = S.dominant\_colors(self.histogram, self.colors)

**Two Image Formats for Analysis and Display**

You may have noticed that in the Tile class we open the image path twice, once using OpenCV to read the image in as a Numpy array, and the second time as a PIL Image. That is because the array is very useful for analysis and calculating histograms. However, concatenating images in their array format is not as forgiving (they generally want to have to have the same dimensions, and throw numerous complaints and errors), so instead we chose to use the paste method of PIL, which only requires you to specify a 2-coordinate tuple for the top-right corner of a tile image to be “pasted” into a canvas, without care for overlaps and so on.

**Cropping Square Tiles for Uniformity**

Our Tile class imported the file “reduction.py”, which contained the image processing and manipulation methods that we wrote for this reductive processing step. Because we had images in both Numpy and PIL Image formats, we had to write two sets of methods for resizing the tiles to our desired TILE\_WIDTH and DISPLAY\_WIDTH.

It’s worth noting that while we only dealt with square and uniformly sized tiles in our test database, in order to make the system more robust, we first added square cropping to our resizing method, before actually resizing each tile to our desired width for analysis. That way, our tiles maintained their aspect ratio during resizing, rather than skewing and stretching non-square images. Moreover, square tiles rather than rectangular image tiles allowed us to later treat the base image as a uniform grid with square cells. With a Numpy array, cropping was especially easy, all we had to do was find the right values for image[start\_y:end\_y, start\_x:end\_x] and voila, we had a square.

In reduction.py:

**import** **cv2**

**import** **numpy** **as** **np**

**from** **PIL** **import** Image

*# ============================================================*

*# OpenCV methods*

*# ============================================================*

*# resize image to w pixels wide*

**def** resize\_by\_w(image, new\_w):

r = new\_w / float(image.shape[1]) *# calculate aspect ratio*

dim = (int(new\_w), int(image.shape[0] \* r))

*# print r, dim*

image = cv2.resize(image, dim, interpolation = cv2.INTER\_AREA)

**return** image

*# crop image*

**def** crop(image, start\_y, end\_y, start\_x, end\_x):

image = image[start\_y:end\_y, start\_x:end\_x]

**return** image

**def** crop\_square(image, size):

w, h = get\_dimensions(image)

**if** (w > h):

offset = (w - h) / 2

image = crop(image, 0, h, offset, w-offset)

**elif** (h > w):

offset = (h - w) / 2

image = crop(image, offset, h-offset, 0, w)

*# else it is already square*

**if** len(image) != size[0]:

image = resize\_by\_w(image, size[0])

**return** image

*# return width, height*

**def** get\_dimensions(image):

**return** len(image[0]), len(image)

*# ============================================================*

*# PIL methods*

*# ============================================================*

**def** flat( \*nums ):

**return** tuple( int(round(n)) **for** n **in** nums )

**def** resize\_square(img, size):

original = img.size

target = size

*# Calculate aspect ratios*

original\_aspect = original[0] / original[1]

target\_aspect = target[0] / target[1]

*# Image is too tall: take some off the top and bottom*

**if** target\_aspect > original\_aspect:

scale\_factor = target[0] / original[0]

crop\_size = (original[0], target[1] / scale\_factor)

top\_cut\_line = (original[1] - crop\_size[1]) / 2

img = img.crop( flat(0, top\_cut\_line, crop\_size[0], top\_cut\_line + crop\_size[1]) )

*# Image is too wide: take some off the sides*

**elif** target\_aspect < original\_aspect:

scale\_factor = target[1] / original[1]

crop\_size = (target[0]/scale\_factor, original[1])

side\_cut\_line = (original[0] - crop\_size[0]) / 2

img = img.crop( flat(side\_cut\_line, 0, side\_cut\_line + crop\_size[0], crop\_size[1]) )

**return** img.resize(size, Image.ANTIALIAS)

b. Color Histogram Computation

After the tile image has been properly resized and returned from reduction.py, the next step is to calculate the color histogram representation of the image. To do so, we called the color\_histogram method on the new resized image, which was a function imported from similarity.py.

In the same spirit of data reduction, we chose the following bin size for calculating color histograms in similarity.py:

COL\_RANGE = 256

BINS = 4

BIN\_SIZE = int(COL\_RANGE/BINS)

**Why 4 for Number of Color Bins**

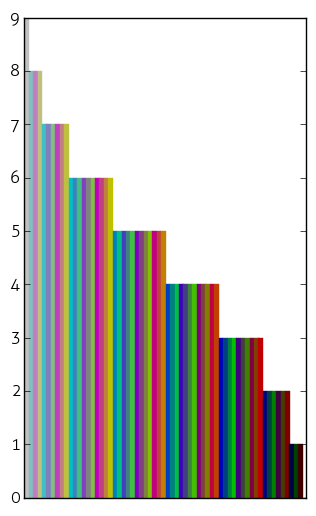
As we learned from developing our Visual Information Retrieval system for Assignment 2, because there are 256 RGB possible values in each color pixel, giving each color value a separate bin would make generating a histogram unfeasible (let alone for each of the 500 tiles images!) because that would be accounting for ~16 million bins! Not to mention, there would be a scarcity of data for each bin. Therefore, it was critical to determine a good size for clustering color values that would still capture the color distribution of an image histogram within decent runtime.

As can be expected, we were inclined toward testing values to the power of 2. While it is true that the color histograms generated with 8 and 16 bins (as we had respectively chosen to do in assignment 2) had a wider and more diverse array of color bars compared to the histogram visualizations, we found out that we got very comparable results for best matching images and recognizable mosaics, whether we chose 4, 8, or 16 bins for color. Therefore, we decided 4 bins was good enough, and fast enough, as it would result in less calculations for image matching as we were using the L1 norm for each bin. There were still 4\*4\*4 = 64 possible color bins for each pixel to be counted into.

Fun fact: the NES color palatte had only 64 pre-set colors (though only 56 of which are unique), generated not with RGB settings, but with the YpbPr algorithm.[[1]](#footnote-1)



The palette available in our bins would be similar to this but actually have 64 unique values. Since NES can represent a fairly wide range of graphics with such a limited palatte, we hope that our chosen colors bins also suffice in creating some recognizable mosaics. The figure below, in somewhat arbitrary order, represents our 64 color bins, though this representation is missing black and white.

 I believe this graph was made s.t. bar\_count = 2b + 2g + 2r

**Color Histogram Implementation**

Similar to assignment 2, we chose to compute the histograms without the aid of any black box algorithms. While OpenCV does have the cv2.compareHist function and SciPy also comes with its own distance metrics, implementing a 3D color histogram is actually quite simple, as can be seen in the code fragment below:

**def** color\_histogram(image, title):

*'''*

*Calculate the 3D color histogram of an image by counting the number*

*of RGB values in a set number of bins*

*image -- pre-loaded image using cv2.imread function*

*title -- image title*

*'''*

colors = []

h = len(image)

w = len(image[0])

*# Create a 3D array - if BINS is 8, there are 8^3 = 512 total bins*

hist = np.zeros(shape=(BINS, BINS, BINS))

*# Traverse each pixel in the image matrix and increment the appropriate*

*# hist[r\_bin][g\_bin][b\_bin] - we know which one by floor dividing the*

*# original RGB values / BIN\_SIZE*

**for** i **in** xrange(h):

**for** j **in** xrange(w):

pixel = image[i][j]

*# Handling different image formats*

**try**: *# If transparent (alpha channel = 0), change to white pixel*

**if** pixel[3] == 0:

pixel[0] = 255

pixel[1] = 255

pixel[2] = 255

**except** (**IndexError**):

**pass** *# do nothing if alpha channel is missing*

*# Note: pixel[i] is descending since OpenCV loads BGR*

r\_bin = pixel[2] / BIN\_SIZE

g\_bin = pixel[1] / BIN\_SIZE

b\_bin = pixel[0] / BIN\_SIZE

hist[r\_bin][g\_bin][b\_bin] += 1

*# Generate list of color keys for visualization*

**if** (r\_bin,g\_bin,b\_bin) **not** **in** colors:

colors.append( (r\_bin,g\_bin,b\_bin) )

*# Sort colors from highest count to lowest counts*

colors = sorted(colors, key=**lambda** c: -hist[(c[0])][(c[1])][(c[2])])

*# Return image in case transparent values were changed*

**return** hist, image, colors

In the try block, we added an additional check to handle different image formats, some of which may contain transparent pixels, in which case we chose to replace with white pixel values. Although our image database didn’t have any transparent pixels, we were trying to develop a system that could be extended later ofr other formats. Because we changed transparent pixels to white, we had to return the (possibly) corrected image as well.

In addition to the histogram and the image, we also returned a list of color tuples for any colors with counts above 0 in the image. This critical line sorts the color tuples:

colors = sorted(colors, key=**lambda** c: -hist[(c[0])][(c[1])][(c[2])])

colors is the list of RGB color tuples from the color\_histogram method, and this lambda function re-sorts it from the highest count for that tuple in the histogram to the lowest count. This sorted list of color tuples, which were useful later on for our additional options of visualizing the bar charts and calculating dominant colors, because we could just retrieve the histogram count with the color keys, rather than iterating through every single bin.

c. Grayscale Histogram Computation

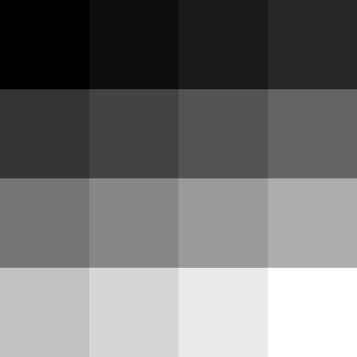
We chose to compute both the color histograms and grayscale (brightness) histograms of each tile, so we had the option to generate both colorful and black-and-white mosaics.

gBINS = BINS\*BINS *# need more bins for grayscale since there's only one axis*

gBIN\_SIZE = int(COL\_RANGE/gBINS)

**Number of Gray Bins: 16**

Because there is only one axis for grayscale, we needed more than 4 bins. We settled on 4 \* 4 = 16 shades of gray bins. This essentially translated to a 4 bit grayscale palaette.



To calculate the counts for the grayscale histogram, we took an average of the RGB values, then determined the bin the pixel should fall in by dividing it by the gray bin size. This was much like the color\_histogram method, except instead of a 3D array we only had a 1D array.

**def** grayscale\_histogram(image, title):

*'''*

*Calculate the grayscale / luminescence histogram of an image*

*by counting the number of grayscale values in a set number of*

*bins*

*'''*

grayscale = []

h = len(image)

w = len(image[0])

hist = np.zeros(shape=(gBINS))

**for** i **in** xrange(h):

**for** j **in** xrange(w):

pixel = image[i, j]

gray = operator.add(int(pixel[0]), int(pixel[1]))

gray = operator.add(gray, int(pixel[2]))

gray = gray/3

g\_bin = gray/gBIN\_SIZE

hist[g\_bin] += 1

**return** hist

d. (Optional) Color Histogram Visualization

## The visualize\_chist function invokes matplotlib to plot a bar graph for the color histogram, so it’s a lot of boilerplate, but basically it iterates through our sorted list of colors and generates a count bar for each color bin.

**for** idx, c **in** enumerate(colors):

r, g, b = c

plt.subplot(1,2,1).bar(idx, hist[r][g][b], color=hexencode(c, BIN\_SIZE), edgecolor=hexencode(c, BIN\_SIZE))

## In order to get the right hex string for the colored bars in the chart, it uses the following helper function:

**def** hexencode(rgb, factor):

*"""Convert RGB tuple to hexadecimal color code."""*

r = rgb[0]\*factor

g = rgb[1]\*factor

b = rgb[2]\*factor

**return** '#**%02x%02x%02x**' % (r,g,b)

to convert our reduced color bins to a legitimate hexadecimal code. Because writing new images to file for every single tile was costly for runtime, this step was optional and actually only used for the report analysis. An example of a color histogram visualization is shown in the discussion of dominant colors.

e. (Optional) Dominant Colors Listing

In order to find the dominant colors in an image, we use the list of colors returned by the color\_histogram method, and calculate the percentage of pixels that go into that bin. This is accomplished by summing all the counts in the histograms, and then taking every tuple in the sorted list of colors, finding the count, and dividing it by the number of pixels in the histogram counts to get the percentage.

**Ignore Black and White**

Because black and white were such prevalant colors in the tile images, either faming the image, or forming the background/foreground, we decided to ignore black and white pixels in the dominant colors identification, because too many images easily had over 30% of their pixels fall into the black (0,0,0) or white (BINS-1,BINS-1,BINS-1) bin.

**Justifying Threshold Value of 0.3 and Color Bin Size of 4**

While we began with a threshold of 0.1 (10% of the pixels in that bin) to be the dominant color threshold, after several trials we decided to settle on a threshold of 0.3. Results will follow in a discussion of different aesthetic techniques for the mosaic. Because the colors were already sorted by histogram count, once a color bin fell underneath the threshold, we could simply break the loop and return the list of dominant colors.

DOM\_COL\_THRESH = 0.3

**def** dominant\_colors(hist, colors):

*"""Helper method to determine percentages of color pixels in a picture"""*

num\_pixels = 0

dominant\_colors = []

**for** (r,g,b) **in** colors:

num\_pixels += hist[r][g][b]

**for** (r,g,b) **in** colors:

*# Ignore black and white pixels*

**if** (r,g,b) != (0,0,0) **and** (r,g,b) != (BINS-1,BINS-1,BINS-1):

p = round( (float(hist[r][g][b]) / num\_pixels), 3)

**if** p > DOM\_COL\_THRESH:

dominant\_colors.append( (r,g,b) )

**else**:

**return** dominant\_colors *# don't care about the rest*

**return** dominant\_colors *# in case*

The first set of four images and color histograms were for a different image database, and used 8 bins. For these conditions, a threshold as low as 0.08 made sense, although we still had to discount black and white pixels because they often had fairly high ratios while it was hard to advocate for them being the domina color in a picture.

|  |  |  |  |
| --- | --- | --- | --- |
| https://lh3.googleusercontent.com/Qt3uXeZeiWBEKBeMFj3mRCk0a1Ut1bdeS9nD03M9zex6PuYvII7HtQ4nRayWCSIHONVavwSyA4VowqUC5be3vUCWD54ieGyggT0YkfU1QTnJnogJKgPsJ8WYBARXeGnhCyjHgEg | https://lh6.googleusercontent.com/U7ZInisGPns_84WXU-I3D6dYvjOw3_q3VDk-9BxMwML2oiWGwYQaFI4iuWYNmZ515YvJBY_GwOyPB8yIcDxkeega6UMnCkX7TvxcJOIQ8pkLtBE-SEO8-wg_wX96IFPdFnujoQM | https://lh6.googleusercontent.com/kS4igZBMQakLGi03M6YqvCU6gfMGDxHY0K5d3Ys6xgH9PDmYEdL0TZjiQQd6EGedS0Rw0myuU01BDC7sjDW_seMAlJ8LrCytwvZ9-s3LLu--oDmbC8TGc2p_1ciVVK8j3mgXNFQ | https://lh3.googleusercontent.com/WGHpjViaCJq_v5v4_Z3EHA4Boj6zRJM_QebxU8Zl9b3B2szaFyyxBAuPrThqBUnTikpKuqR_qMSN-rKWkw3w4Yzw0CqRTCLkIP_udDOM8YQ5d8dOln6hOIBRsCza9PpvUEPPvRs |
| https://lh5.googleusercontent.com/mgqkAF4JP_lzutSK1-t-aAzKFVuKrUkhT-iBIpgkHE088mXvWe_z5SZnIHpcJ_Ozkc789ATxsyDmMY9gSf5IIm9hSZukPbMDN8faA8gFA1yvRM3WtxugB-Pfvn4x7i1_vOEu9Rc  Top 3 ratios  [0.093, 0.071, 0.067] | https://lh3.googleusercontent.com/_ePG6l-FysqJtxtaHN0WfCsuLpKwAHkkmQrwZMT7tuShb0YuINKlDHDgDxNOTcfembVxokYkOrbv0ApgO0Izacp2ORtztzMp93kfUDitwPtvm8dfE_GXNMbf3-cdiHzWvP1i-SE  Top 3 ratios  [0.124, 0.098, 0.071] | https://lh4.googleusercontent.com/s4a9B1UJLqB40htkwSMXb2beAysktUQ0ADDhILD_ET1hnlxUEct3fWfMs-FO1aIJGW6F0OZ-wIPLJMyVMRdhgsNvgOrX5H92UnMDpT4HCxeVbMOoiOftKJap7eXIJWaScDDRfZ4  Top 3 ratios  [0.316, 0.244, 0.102] | https://lh4.googleusercontent.com/kUVqMCBkW9D9r_pnldet7Guko8J2UiQ8vwxNKBqH9jGhaegVip6S5r0GJXFNBjw87OsYX0ze3-Gb8i4I5J7-dbt41I1SqWxF56k58cAsTL7LOvfdF4Rp7tj7BT3jqPVE2IapT4w  Top 3 ratios  [0.089, 0.08, 0.053] |
| Note for last image: though clearly blue dominates from a human perspective, it is not a dominant color according to the histogram or the ratios, likely because it is spread out across too many bins. | | | |
|  |  |  |  |
| Top 3 for 1308516372  [ 0.456 0.302 0.131 ] | Top 3 for 1307917054  [ 0.522 0.382 0.008 ] | Top 3 for 1320444556  [ 0.449 0.343 0.113 ] | Top 3 for 1320778302  [ 0.711 0.151 0.041] |
|  |  |  |  |

In this set of images, we see justification for less bins. The colors cluster together fairly nicely, and a threshold of 0.3 seems reasonable for identifying a dominant color.

**Quick Note on K-Means Clustering**

We tried color-based segmentation using K-Means clustering twice: once with a blackbox algorithm from the Scikit-Learning library (implemented in similarity.py) and in another raw attempt with namedtuples by following an open source tutorial (implemented in dominance.py). In both cases, pixels were represented in a 3D vector space and random sampling and K-means were used to find cluster centers and locate the three largest clusters. However, the runtime for calculating dominant color for every single tile (and later, every single quadrant in the base image) was much slower than our previous implementation, and the results were similar enough to our method of 4 reduced bins and simply calculating ratios of color counts, that we ended up sticking with our simplistic method.

**Organizing Tiles into Dominant Color Bins**

If we chose to this optional step of finding dominant colors, then we had an extra self.dominants field in the Tile object. In that case, we had to add an additional step in main.py after initializing every tile object in the tile directory.

*# Optional: use dominant colors method*

**if** (DOM\_ON):

**for** color **in** tile.dominants:

**if** color **in** dominants:

dominants[color].append(tile)

**else**:

dominants[color] = [tile]

Here, we iterated through the list of colors in tile.dominants, and created a dictionary where the key was a color tuple, and the value was a list of tiles with that tuple in their list of dominant colors. That way, we could later attempt to match mosaic tiles with the base image regions via dominant colors; this had the double advantage of being faster than a brute force matching, and also resulted in a diversity of results because for example, a patch of cerulean blue sky from the base image could pick randomly from a pool of multiple tiles with cerulean blue (whatever its colod code is) as their dominant color, and and therefore not be so uniform.

Initializing the Base Object

**Resizing Base Image**

We chose to resize the base images to the width TILE\_WIDTH \* DESIRED\_COLS. Because unlike the tile images, the base image is not restricted to square images, we didn’t have to call the crop method, but instead just resized the base image to ensure that we have enough tile columns. We’ve see the implementation of resize\_by\_w above in the reduction.py excerpt, so we won’t go into that again.

Instead, we will consider the value of DESIRED\_COLS. The effect of increasing the number of DESIRED\_COLS is an increase in accuracy and recognition, but also a large increase in time because we effectively have to calculate TILE\_WIDTH \* DESIRED\_COLS many histograms, and later best matches. Take the example below of a TILE\_WIDTH = 30 and DESIRED\_COLS = 50, that’s 150 iterations just to calculate histograms for every quadrant in one base image. Now imagine if we had 100 columns… that would be even more. For most testing purposes, we found DESIRED\_COLS = 50 to be a fair amount. However, for our user studies (as we’ll describe later), we defined DESIRED\_COLS = 100 to give more accurate results, so that was the value we used there.

**import** **cv2**

**import** **reduction** **as** **R**

**import** **similarity** **as** **S**

**from** **dominance** **import** colorz

TILE\_WIDTH = 30

DESIRED\_COLS = 100

**class** **Base**():

**def** \_\_init\_\_(self, path):

self.path = path

self.image = cv2.imread(path, cv2.IMREAD\_UNCHANGED)

self.image = R.resize\_by\_w(self.image, TILE\_WIDTH\*DESIRED\_COLS)

self.height = len(self.image)

self.width = len(self.image[0])

self.histograms = []

self.dominants = [] *#*

self.grayscales = []

**for** j **in** xrange(0, self.height, TILE\_WIDTH):

hist\_row = []

dom\_row = [] *#*

gray\_row = []

**for** i **in** xrange(0, self.width, TILE\_WIDTH):

start\_y = j

end\_y = j + TILE\_WIDTH

start\_x = i

end\_x = i + TILE\_WIDTH

quadrant = R.crop(self.image, start\_y, end\_y, start\_x, end\_x)

title = "base" + str(end\_x) + "-" + str(end\_y)

histogram, quadrant, colors = S.color\_histogram(quadrant, title)

grayscale = S.grayscale\_histogram(quadrant, title)

*# Optional, save histogram as bar graph; or record dominant colors*

*# plot\_path = S.plot\_histogram(histogram, title, colors)*

dominants = S.dominant\_colors(histogram, colors) *#*

*# dominants = S.kmeans\_dominance(self.image)*

*# dominants = colorz(quadrant)*

hist\_row.append(histogram)

dom\_row.append(dominants) *#*

gray\_row.append(grayscale)

self.histograms.append(hist\_row)

self.dominants.append(dom\_row) *#*

self.grayscales.append(gray\_row)

**print** "**%d** out of **%d** rows" %((j/TILE\_WIDTH)+1, (self.height/TILE\_WIDTH))

self.rows = len(self.histograms)

self.cols = len(self.histograms[0])

**Computing Histograms and Dominant Colors for Each Quadrant**

Example of a 40 column \* 10 row grid (each “quadrant” in the base image would map to a 30 \* 30 tile):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

From there, we iterate across the image 30 pixels to the right at a time and then 30 pixels down when we move to the next row, in order to calculate color histograms for every single equivalent tile in the image. We use the simple crop method of truncating a Numpy array to our desired indices and assigning it a new variable quadrant, then using the color\_histogram method on the quadrant as if it were its own image, and obtaining the return value. Similarly, we take the same cropped quadrant to calculate grayscale histograms, and use dominant\_colors method on the histogram matrix and colors list that was returned by the color\_histogram() method. Each time we iterate across the number of desired columns in our picture, our row list of histograms, row list of grayscale histograms, and and row list of dominant colors is added to self.histograms, self.grayscales and self.dominants respectively. Finally, we calculate the number of rows and columns in our new image by looking at the length of self.histograms and length of self.histograms[0] respectively.

Note that the resize\_by\_w, crop color\_histogram, grayscale\_histogram, dominant\_color histogram are all the exact same implementations as we used and described before.

Matching Tile Images

This was the heart of our algorithm, and much of the visual processing that took place in the initialization of the tiles and base image objects were rationalized for enabling and improving the performance of this step.

We improved it in a series of steps, by implementing in this order:

* Expensive method
* History method
* Alpha method
* Dominant color method

and evaluating the different performances and aesthetics produced by these methods.

a. Expensive Method

We began our tile matching with what I would later call the expensive method, a naive brute force method that compared every single quadrant in the base image to every single tile in the database using the L1 norm. The problem is if we only pick the best tile each time, it’s as expensive as O(N\*M) where N is the number of tiles in the database and M is the number of quadrants in the source image. For instance, if we have a 100 column by 60 row base image, and 500 tile images, that 100 \* 60 \* 500 = 3,000,000 comparisons being made in this step alone!

In addition to this performance problem, we ended somewhat of an aesthetic problem. Since we’re only using the best match, we often only end up using what seemed less than 10% of the database, and had many repeating images for single color swathes in the base image.

To implement this method in main, we just looped through every row and column in the base image, and compared the histogram of each quadrant to every single tile in the tile database, seeking the “best tile” with the minimum distance according to the L1 norm between their two histograms. We had a list called the\_chosen, which was a list of lists containing the titles of the tile images, which we could later use to retrieve any tile from the tile dictionary.

the\_chosen = []

**for** i **in** xrange(base.rows):

hist\_row = base.histograms[i]

the\_row = []

**for** j **in** xrange(base.cols):

histogram = hist\_row[j]

closest = 100

**for** key **in** tiles:

tile = tiles[key]

distance = S.l1\_color\_norm(histogram, tile.histogram)

**if** (distance < closest):

closest = distance

closest\_tile = tile

the\_row.append(closest\_tile.title)

the\_chosen.append(the\_row)

**Comparing Two Images By Calculating L1 Color Norm**

To calculate the relative similarity or distances between a quadrant and a tile, we used the L1 norm. We considered the norm (or distance) for two images to be:

L1\_norm = ∑ (differences) / ∑ (pixel count)

where

L1\_norm = distance = 1 – similarity

or you could say

similarity = 1 – distance = 1 – L1\_norm

Since we found it more intuitive to work with distances than similarities, looking for images with “shorter distances” and closer to 0, we referred to distance values. Therefore, for best tile match, we used the minimum distance between their color histograms.

Our simple implementation is below where h1 and h2 are the histograms of the base image quadrant and tile image respectively. This code is listed in similarity.py:

**def** l1\_color\_norm(h1, h2):

diff = 0

total = 0

**for** r **in** xrange(0, BINS):

**for** g **in** xrange(0, BINS):

**for** b **in** range(0, BINS):

diff += abs(h1[r][g][b] - h2[r][g][b])

total += h1[r][g][b] + h2[r][g][b]

l1\_norm = diff / 2.0 / total

similarity = 1 - l1\_norm

**return** l1\_norm

**Improving the Matching Algorithm**

To improve the matching algorithm, we tried to think both in terms of performance and aesthetics. Performance was the most pressing factor, as it was taking almost 10 minutes just to generate a 100 column image, which was replete with repeated tiles. Our secondary concern was to diversify our tile matches – not to always use the same red tile repeatedly. However, our primary concern was still how to improve performance.

b. History Method

Toward the goal of performance improvement and temporarily ignoring the repeating image problem, we added the History Method. Namely, if quadrant has same histogram as earlier one, then reuse that closest\_tile variable from earlier.

Basically, we kept the same skeleton code as before (new code additions indicated by the cyan highlight), but added a dictionary called history in addition to the\_chosen list, and before doing the numerous comparisons with the tiles, we would check if the history contained a past histogram with the same values as our current histogram. If so, we just used the closest tile that was calculated earlier and stored as the value in the history dictionary. If this histogram was not in our history, then we would perform the expensive operation, find the best tile match, and store the histogram as the key and the best tile as the value.

the\_chosen = []

history = {} *# store histogram-best tile matches*

**for** i **in** xrange(base.rows):

hist\_row = base.histograms[i]

the\_row = []

**for** j **in** xrange(base.cols):

histogram = hist\_row[j]

closest = 100

**if** str(histogram) **in** history:

closest\_tile = history[str(histogram)]

*# This constant-time lookup saves a lot of calculations*

**else**:

**for** key **in** tiles:

tile = tiles[key]

distance = S.l1\_color\_norm(histogram, tile.histogram)

**if** (distance < closest):

closest = distance

closest\_tile = tile

history[str(histogram)] = closest\_tile

the\_row.append(closest\_tile.title)

the\_chosen.append(the\_row)

We predicted that this method would be particularly effective for cartoonified images, which contained many quadrant histograms that held just a single color block. For every quadrant that used the expensive operation to find a best tile match, that required 500 comparisons; compared to using the history method, which found the closest match in constant O(1) access time by just looking it up in the dictionary.

c. Alpha Method

With the aim of improving the aesthetic and recognizability of our images, we introduced the ALPHA constant at the beginning of main.py, which was a value between 0 and 1 that determined the ratio in the linear sum of color and grayscale similarity.

Again, new code is highlighted in cyan.

the\_chosen = []

history = {} *# store histogram-best tile matches*

**for** i **in** xrange(base.rows):

hist\_row = base.histograms[i]

grayscales = base.grayscales[i]

the\_row = []

**for** j **in** xrange(base.cols):

histogram = hist\_row[j]

graygram = grayscales[j]

closest = 100

**if** str(histogram) **in** history:

closest\_tile = history[str(histogram)]

*# This constant-time lookup saves a lot of calculations*

**else**:

**for** key **in** tiles:

tile = tiles[key]

**if** ALPHA == 1: *# All color*

distance = S.l1\_color\_norm(histogram, tile.histogram)

**elif** ALPHA == 0: *# All grayscale*

distance = S.l1\_gray\_norm(graygram, tile.gray)

**else**: *# Linear sum of ratio between the two*

dcolor = S.l1\_color\_norm(histogram, tile.histogram)

dgray = S.l1\_gray\_norm(graygram, tile.gray)

distance = ALPHA\*dcolor + (1-ALPHA)\*dgray

**if** (distance < closest):

closest = distance

closest\_tile = tile

history[str(histogram)] = closest\_tile

the\_row.append(closest\_tile.title)

the\_chosen.append(the\_row)

**Calculating L1 Grayscale Norm**

We calculated the l1 norm in the same way as the color norm, the only difference being that we had to loop through all three axes of the color histograms, whereas the grayscale histograms only had 1 axis we had to loop through. In similarity.py

**def** l1\_gray\_norm(h1, h2):

diff = 0

total = 0

**for** g **in** xrange(0, gBINS):

diff += abs(h1[g]-h2[g])

total += h1[g]+h2[g]

l1\_norm = diff/2.0/total

**return** l1\_norm

We considered using HSV values to measure intensity instead of RGB, but we were also very interested in seeing whether an all-gray mosaic was still recognizable – and it was! ALPHA = 0 meant the matching was solely determined by grayscale histograms, while ALPHA = 1 meant that matching was solely determined by color histograms. Anything in between was a linear sum of ratios between the two. Sadly, none of the linear sums produced very good results, but the all grayscale methods were very clear.

d. Dominant Color Method

Finally, we also tried the dominant color method. Just to recap my previous discussions of dominant colors: we find the dominant colors of each tile (listing them if it’s not black or white), organize them in a dictionary where the key is a color code, and the value is a list of tiles which have that as color as a dominant color. Then we look up what dominant colors a quadrants holds, and see if there any tile images associated with that dominant color in the dictionary, and pick one! This is also nearly cnstant time lookup compared to the expensive method, since most images have 3 to no dominant colors.

This is the full algorithm for matching image tiles, including counting methods and print statements to determine progress and figure out what percentage of operations are expensive, history, or dominant.

As before the cyan is the new code related to the new operation; green code are helpful counters to determine how often each method is actually being used over the life of the program.

*# Find best tiles to compose base image*

**print** "Generating mosaic..."

the\_chosen = []

history = {} *# store histogram-best tile matches*

count = base.rows \* base.cols

dom\_count = 0

history\_count = 0

expensive\_count = 0

**for** i **in** xrange(base.rows):

hist\_row = base.histograms[i]

grayscales = base.grayscales[i]

**if** (DOM\_ON):

dom\_row = base.dominants[i]

the\_row = []

**for** j **in** xrange(base.cols):

skip = False

histogram = hist\_row[j]

graygram = grayscales[j]

*# Optional: use dominant colors method*

**if** (DOM\_ON **and** ALPHA == 1):

**for** dom **in** dom\_row[j]:

**if** dom **in** dominants:

closest\_tile = random.choice(dominants[dom])

skip = True

dom\_count += 1

**break**

**if** (skip == False):

closest = 100

**if** str(histogram) **in** history:

closest\_tile = history[str(histogram)]

*# This constant-time lookup saves a lot of calculations*

history\_count += 1

**else**:

**for** key **in** tiles:

tile = tiles[key]

**if** ALPHA == 1: *# All color*

distance = S.l1\_color\_norm(histogram, tile.histogram)

**elif** ALPHA == 0: *# All grayscale*

distance = S.l1\_gray\_norm(graygram, tile.gray)

**else**: *# Linear sum of ratio between the two*

dcolor = S.l1\_color\_norm(histogram, tile.histogram)

dgray = S.l1\_gray\_norm(graygram, tile.gray)

distance = ALPHA\*dcolor + (1-ALPHA)\*dgray

**if** (distance < closest):

closest = distance

closest\_tile = tile

history[str(histogram)] = closest\_tile

expensive\_count += 1

the\_row.append(closest\_tile.title)

the\_chosen.append(the\_row)

*# print the\_row*

**print** "**%d** out of **%d** rows" %(len(the \_chosen), base.rows)

Generating the Mosaic

At this point, after we have assembled a double array called the\_chosen that looks somewhat like this:

[['1307907058', '1317768895', '1321142449', '1306283809', '1317772859', ... '1307907058', '1313439496', '1314153453', '1313439496', '1317768895'],

...

['1314153453', '1321142449', '1305147738', '1306637389', '1306593722', ... '1313439496', '1317600561', '1307907058', '1317768895', '1314153453']]

a list of row lists that contain tile titles, it is a simple matter to paste the tiles together using PIL in order to generate the final mosaic for our viewing pleasure. It so happens that this row list correlates exactly with base.histograms coordinates. Depending on the ALPHA value, the image format will be in grayscale or RGBA, but otherwise it is just a matter of creating a mosaic Image canvas whose width is the number of base.cols \* the display tile’s width, and whose height is the number of the base.rows \* display tile’s width. From there, we just iterate through the “columns” in each row list and paste on the canvas by the tile’s display width, moving on to a new row when the sublist ends. The method in main.py is as follows:

*# Generate mosaic*

**print** ""

size = tile.display.size *# any tile will have the same size*

**if** ALPHA == 0: *#grayscale mosaic*

**print** "Your GRAYSCALE MOSAIC will be done soon."

mosaic = Image.new('L', (base.cols\*size[0], base.rows\*size[1]))

**else**:

**print** "Your COLORED MOSAIC will be done soon."

mosaic = Image.new('RGBA', (base.cols\*size[0], base.rows\*size[1]))

rowcount = 0

**for** row **in** xrange(base.rows):

colcount = 0

**for** col **in** xrange(base.cols):

idx = the\_chosen[row][col]

tile = tiles[idx]

img = tile.display

mosaic.paste(img, (colcount\*size[0], rowcount\*size[1]))

colcount += 1

rowcount += 1

mosaic.save(base\_path[:-4]+"-Mosaic"+str(ALPHA)+".png")

**print** "Successfully saved to "+base\_path[:-4]+"-Mosaic"+str(ALPHA)+".png"

f = open('mosaic\_keys.txt', 'w')

f.write(str(the\_chosen))

*# Calculate percentage of database used and print stats*

**print** ""

n = len(set([img **for** sublist **in** the\_chosen **for** img **in** sublist]))

**print** "Percent of possible tiles used: **%.3f**, **%d** out **%d** images from tile library used" %(round((float(n)/len(tiles)), 3), n, len(tiles))

**print** ""

**print** "Expensive operations:", expensive\_count, "of", count, ":", expensive\_count/count

**print** "Dominant operations:", dom\_count, "of", count, ":", dom\_count/count

**print** "History operations:", history\_count, "of", count, ":", history\_count/count

**if** \_\_name\_\_ == "\_\_main\_\_": main()

We also chose to write the\_chosen to a text file, because with by commenting out the tile matching code and copyiong the list from the text file, we could almost instantly recreate any mosaic using the method described above.

|  |  |  |  |
| --- | --- | --- | --- |
| https://lh3.googleusercontent.com/Ldu15uMZBIEsCJ7SDdnTOvT1cIh0nHnX_srpXnhf9jzs9_SJ7boe_BPiFpdPhlAkeMOINLHRMzY5t0khVfAnF6YQ5B_j1Ymv1Y6i9KHKEAf_EGpV215h9FrPM3tddBc6-KMgT1w | https://lh3.googleusercontent.com/j20ylutxX-VMHN6GxDKvQOmn5zmVXvUANCIL0Lyjpp1hWHPaxeKq7JFWQCqN4AKpsFXknUk0Ml6-iXFmy1k7wgkEUAOfKd8Hm3AhY7Qkxu4OAaVSb6oY7EhxRR3If3ZlVsMVPgo | https://lh5.googleusercontent.com/8MuP0c2CVhVZDe7ODU6mjrKRXWIs9e0svXkgmrXqt0x1zYQF49jVO2mtx1GLXwdFn8snCz8Vtt2FGZ3lWHw_czZUBrwjx9IgQ6xdVQE6Jmed2xpcUy7XZg0ebuONBCyWVuD9KRM | https://lh6.googleusercontent.com/dm3idmvCGCY0QAmIwK-k6oZRnCi2Je2kJ31jXRUFzTT-0RjSm7xCfdljCmG4ynXsecywGGFaDePpuNFPBBy6QCJe_fQXI2XnV3VqSqFAL7R4PIa13YVcAKlTIgN1luqo6K5599o |
| Original | Fruity database  Threshold: 0.05 | Fruity database  Threshold: 0.1 | NatGeo database  Threshold: 0.1 |
| https://lh4.googleusercontent.com/q5ouMAb2VRDCqUx5ynV5JvMgznkzIgWGgfLfa_GCEkKB2edSjWTAYjuXqhb28aFmbcm0RSmvOgWoz5oOog5R-HqYkG2Ml8wciwwUorwi4ZewaeLBCw-XN3adQjnEXjXphbXrmQM |  |  |  |
| Excludes black | 100 columns | 4 bins instead of 8 | 0.3 threshold |
|  |  | Taking out measures of color dominance  Expensive operations: 588 of 2500 : 0.2352  Dominant operations: 0 of 2500 : 0.0  History operations: 1912 of 2500 : 0.7648 |  |
| Excludes black and white | Does not use dominance | Expensive operations: 587 of 2500 : 0.2348  Dominant operations: 35 of 2500 : 0.014  History operations: 1878 of 2500 : 0.7512 | These look the same…  Percent of possible tiles used: 0.236, 118 out 500 images from tile library used |

Expensive operations: 588 of 2500 : 0.2352

Dominant operations: 0 of 2500 : 0.0

History operations: 1912 of 2500 : 0.7648

*# Calculate percentage of database used and print stats*

**print** ""

n = len(set([img **for** sublist **in** the\_chosen **for** img **in** sublist]))

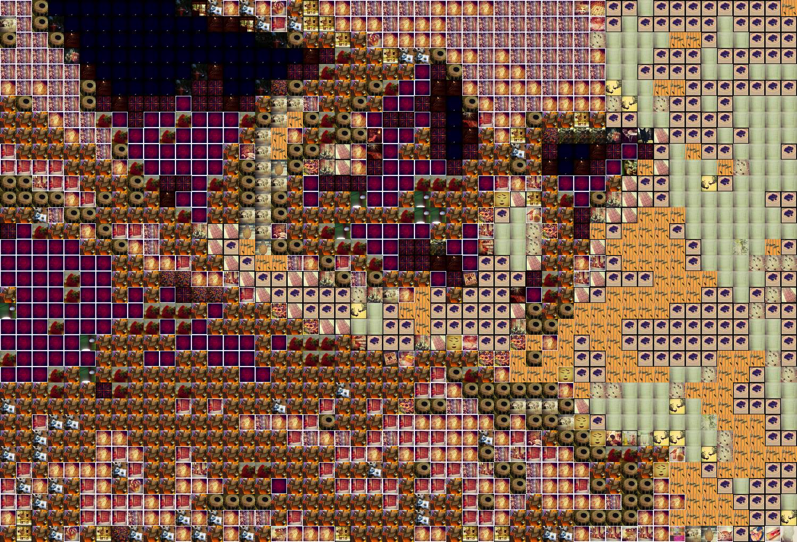
**print** "Percent of possible tiles used: **%.3f**, **%d** out **%d** images from tile library used" %(round((float(n)/len(tiles)), 3), n, len(tiles))

**print** ""

**print** "Expensive operations:", expensive\_count, "of", count, ":", expensive\_count/count

**print** "Dominant operations:", dom\_count, "of", count, ":", dom\_count/count

**print** "History operations:", history\_count, "of", count, ":", history\_count/count

** **

Notes for Evaluation

* User studies
* Check how many tiles we actually use and try to figure out what images work well with what kinda datbases: I wrote a method, and on average we’re using around 100 of a 500 image database, ranging from 10-25% - could we better utilize them in the future?
* Check how many different tiles appear in a collage, and take that as a ratio of total number from database, to see how many images we actually use:

**print** "Percent of possible tiles used: **%.3f**, **%d** out **%d** images from tile library used" %(round((float(n)/len(tiles)), 3), n, len(tiles))

FUTURE EXTENSIONS TO SYSTEM

* A graphical user interface
  + Let users choose their tile database and base images
  + If we had more time, could we have written a method that can see how well an image database would perform with a given base image? i.e. see the range of colors represented in a database (for example, we’re missing green)
* Ghosting – placing a layer of original image on new image – but actually pretty proud we have such good results without ghosting!
* Use tile pictures that are conceptually related to the source image
  + Fix instagram zipper file not to grab images by user, but instead grab images by hash tags!
* Vary the size of the individual images in the grid
  + While the uniformity of the grid is nice for making the image as clear as possible, it would be compositionally more interesting and collage-like to have the sizes of the images vary more
  + Could try breaking up the rectangular shape of each of the source images and using masking
  + Using Laplacian or Canny edge dtection to detect edges of the picture
* Would have liked to play with Jigsaw method or explore Voronoi stippling. This is a technique for converting a grayscale image into a series of dogs of different weights to represent the darkness of each region in a natural way, much like stipped drawings created by hand
* Build a histogram of the number of tiles for each grayscale intensity. Rescale the intensity of images so that each grayscale level has several photos to choose from.
* Come up with a fast way of selecting image tiles that does not require comparing each pixel to each tile's mean intensity.

Deep Representations

* Could we tie in deep representations? Kender emphasized deep representation and neural networks quite a bit in the last class
  + If we had tile pictures conceptually related to source image
  + Better dominant color matching
  + Varied tile sizes so some of the tiles were larger and you could immediately see them
  + Would the human brain look and immediately make the various symbolic patterns and connections?

From Internet:

* Imagine you are trying to recognize someone's handwriting - whether they drew a '7' or a '9'. From years of seeing handwritten digits, you automatically notice the vertical line with a horizontal top section. If you see a closed loop in the top section of the digit, you think it is a '9'. If it is more like a horizontal line, you think of it as a '7'. Easy enough. What it took for you to correctly recognize the digit, however, is an impressive display of fitting smaller features together to make the whole - noticing contrasted edges to make lines, seeing a horizontal vs. vertical line, noticing the positioning of the vertical section underneath the horizontal section, noticing a loop in the horizontal section, etc.
* Ultimately, this is what deep learning or representation learning is meant to do: discover multiple levels of features that work together to define increasingly more abstract aspects of the data (in our case, initial image pixels to lines to full-blown numbers).
* Deep learning is about creating an abstract hierarchical representation of the input data to create useful features for traditional machine learning algorithms. Each layer in the hierarchy learns a more abstract and complex feature of the data, such as edges to eyes to faces.
* This representation gets its power of abstraction by stacking nonlinear functions, where the output of one layer becomes the input to the next.
* The two main schools of thought for analyzing deep architectures are probabilistic vs. direct encoding.
* The probabilistic interpretation means that each layer defines a distribution of hidden units given the observed input, P(h | x).
* The direct encoding interpretation learns two separate functions - the encoder and decoder - to transform the observed input to the feature space and then back to the observed space.

Key Ideas in VI that Kender emphasized in last class

* **VI -> decision what decision are we making?**
* **Throw away?** domain engineering, no background, less bins
* **Deep representation:** pattern recognition, dominant color recognition
* **Science, measurement, prediction:** user studies, system vs. human
* **Engineering: heuristics:** distance, L1 norm

|  |  |
| --- | --- |
|  |  |
|  | (100 columns by 67 rows)  Your COLORFUL MOSAIC will be done soon.  Percent of possible tiles used: 0.466, 233 out 500 images from tile library used  Expensive operations: 5749 of 6700 : 0.858059701493  Dominant operations: 0 of 6700 : 0.0  History operations: 951 of 6700 : 0.141940298507 |
|  |  |
| * 1. Threshold (still counting black white in num\_pixels)   Percent of possible tiles used: 0.798, 399 out 500 images from tile library used  Expensive operations: 280 of 6700 : 0.0417910447761  Dominant operations: 6194 of 6700 : 0.92447761194  History operations: 226 of 6700 : 0.0337313432836 |  |
|  |  |
| 0.3 Threshold  Your COLOR MOSAIC will be done soon.  Successfully saved to \_tests/Low-Mosaic1.png  Percent of possible tiles used: 0.436, 218 out 500 images from tile library used  Expensive operations: 2256 of 6700 : 0.33671641791  Dominant operations: 3613 of 6700 : 0.539253731343  History operations: 831 of 6700 : 0.124029850746 | Your GRAYSCALE MOSAIC will be done soon.  Percent of possible tiles used: 0.792, 396 out 500 images from tile library used  Expensive operations: 5749 of 6700 : 0.858059701493  Dominant operations: 0 of 6700 : 0.0  History operations: 951 of 6700 : 0.141940298507 |
|  |  |
|  |  |
|  |  |
| Expensive operations: 237 of 4100 : 0.0578048780488  Dominant operations: 908 of 4100 : 0.221463414634  History operations: 2955 of 4100 : 0.720731707317  Your COLOR MOSAIC will be done soon.  Successfully saved to nyan-Mosaic1.png  Percent of possible tiles used: 0.176, 88 out 500 images from tile library used |  |
|  |  |
| Your COLOR MOSAIC will be done soon.  Successfully saved to starry-Mosaic1.png  Percent of possible tiles used: 0.308, 154 out 500 images from tile library used  Expensive operations: 4057 of 6300 : 0.643968253968  Dominant operations: 2173 of 6300 : 0.344920634921  History operations: 70 of 6300 : 0.0111111111111 |  |

* Discuss different techniques: different aesthetic effects, different performance characteristics (i.e. took different amount of time to create the photomontage)
  + Also discuss what “kind” of images tend to do better with our method compared to others
* Limiting color bins / shades of gray (if we do grayscale)
* If we don’t actually use all the images, then we don’t need to load PIL display images for all of them. we could add a “cached” property to each tile… if it’s cached, that means it has a tile.display image and you can just use it, else you can load a new image with PIL and then cache it
* If we can increase efficiency and diversification, we can use a bigger database
  + Instead of loading only first 500 images for our database, we could load more images, or load RANDOM images from the database, not just 1-500!
* Try grayscale method and see effects
  + One way we can do a fast algorithm is to sort all the tile images in the database based on their brightness, then we can select the right image to match the cell in the source image based on itsposition, i.e. if we need a full black image, grab ones at the front of the sorted list, if needing full white, grab one at the end, and so forth for everything to begin
  + CAN WE DO SOMETHING SIMILAR WITH COLOR?
  + If we do blacmk and white, we’ll find that they work best with relatively HIGH CONTRAST source images
* For this color similarity, we definitely need to think of methods that doesn’t involve searching through all tile databases to meach each cell. So far tried to the history and dominant color method, but it would be great to extend this sorting technique of the grayscale method. If we could find a way to translate each color into a signle number in a way that is perceptually meaningful,w e could use the same sorting technique to find the closest available tile. For example, we could treat the tile images as RGB pixels and combine three of them in close proximity (and appropriate levels of color intensity) to produce the average color of any cell in the image.
* Try different ways to measure similarity of color between each cell and tile imager (see below)
  + for example we can find the mean intensity value of the image and store it along with the tile in the tile class

1. Source: http://www.thealmightyguru.com/Games/Hacking/Wiki/index.php?title=NES\_Palette [↑](#footnote-ref-1)