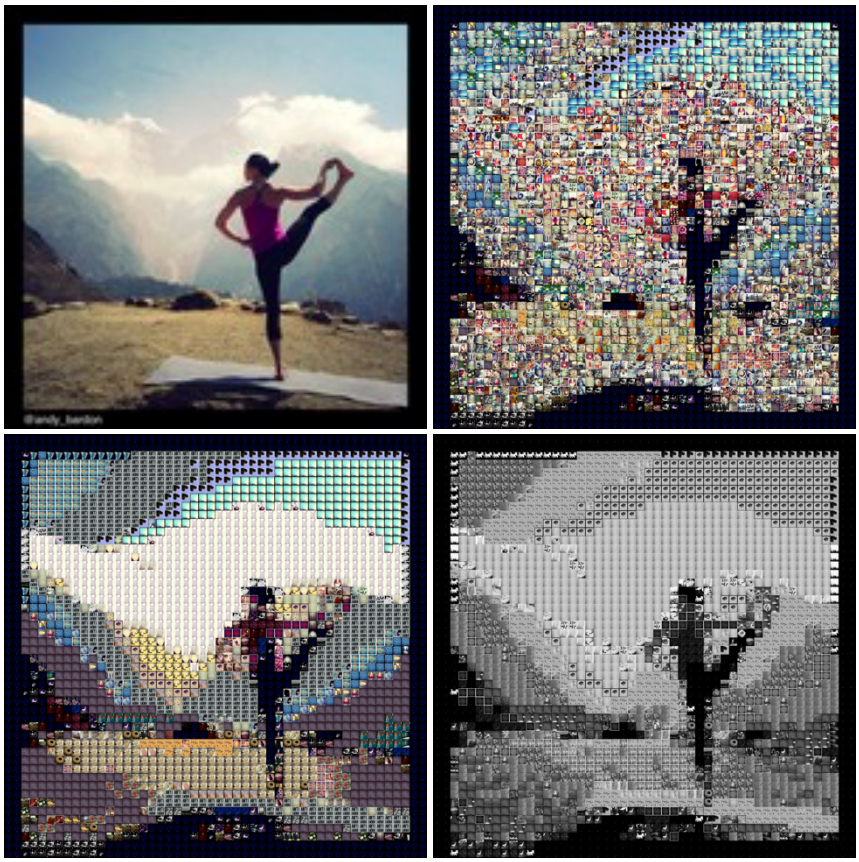
A Preliminary Exploration of Generating

A Mosaic Maker System



Final Project

jointly presented on:
5/13/2015 @ 11:30am

by Nina Baculinao & Melanie Hsu
uni: nb2406 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)

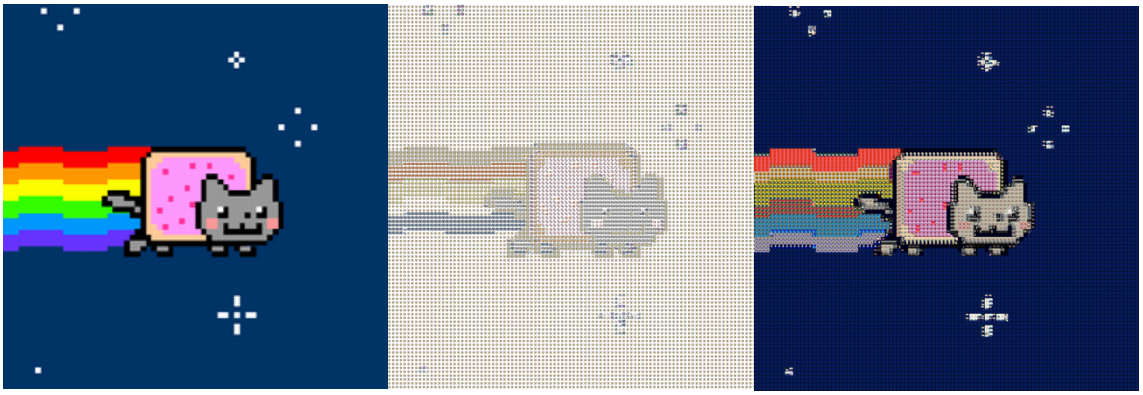
The base image typically didn't have any restrictions on format or aspect ratio, although we sought higher resolution images than the tiles for producing mosaics in the user studies.

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

Initially, we attempted to make mosaics using cartoons as both the base and tile images. However, the mosaics turned out to be very inaccurate. Below are a series of mosaics of nyan cat. The leftmost mosaic was made using pixel images of Pokemon as tiles; the background and rainbow colors are completely wrong. The mosaic on the right was made using photos from an Instagram account. While the nyan cat in the second mosaic is easy to recognize, we realized that the recognizability of cartoon characters is heavily dependent on whether the mosaic can accurately capture their basic colors scheme. This made the recognizability of cartoon mosaics heavily dependent on the availability of the characters' dominant colors in the tile database. Real people, places, and things are not as strongly affected by the quality of the database: for example, most people can probably recognize Lady Gaga even if the colors of her clothes and hair are slightly off. Because of these considerations, we decided instead to use photographs of real entities for both the base and tile images.



We specified in our proposal that we would not manually search for and select images for our database (for instance, add images that had the exact shade of orange that the base image has to our database), nor would we create our own photos for the database, as we believed that it would make the mosaic-making task artificially easy. We wanted to see whether we could make high-quality mosaics using existing databases and what limitations they may have; thus, we ultimately settled on downloading images from the accounts of some Instagram users.

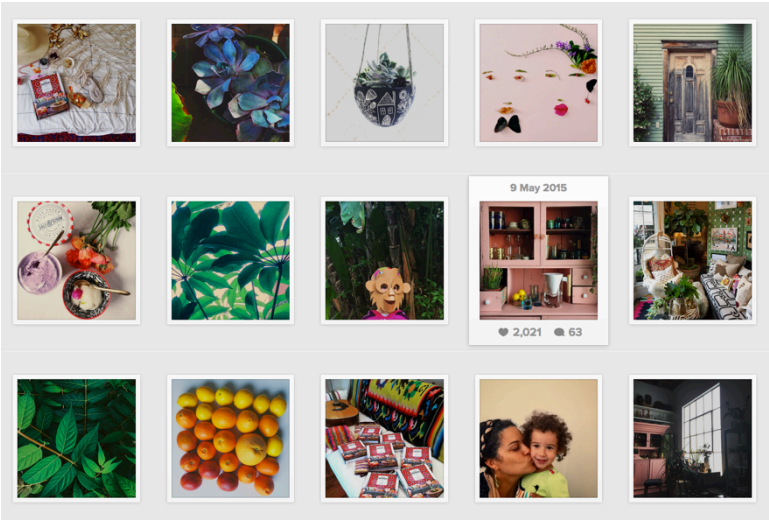
Another reason for our design choice is efficiency and practicality: during our experiments in the user study section, we discovered that using a database containing 1000 images resulted in a runtime of almost 20 minutes, which is unreasonable for the demonstration; if this project were an online app, users would run out of patience while waiting for the mosaic to complete. Since online mosaic-making programs tend to ask the user to upload their own database (ex. http://mosaic-creator.en.softonic.com/, www.easymoza.com/, http://www.aolej.com/mosaic), we decided to use existing databases and work with their limitations instead of creating one that would contain all possible shades of colors.

ig_zipper.py creates a directory of images, given the name of a user's Instagram gallery and a developer token that allows you to access the Instagram API (for example, you can generate one by visiting: http://instagram.pixelunion.net/):

python ig_zipper.py justinablakeney <token>

The above command allows you to download Justina Blakeney's Instagram gallery and have the image thumbnails (150x150) stored in a zipped folder. The full code for ig_zipper.py is available in the Appendix.

Below are some of her photos.



3. Image Processing

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 the 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:
    a. Initialize as Tile object with image path and title as parameters
    b. Analyze the image
        i. Load as a Numpy array with OpenCV2 for analysis purposes
        ii. Shrink to suitable tile width (30) and crop squarely
        iii. Save new height and width
        iv. Calculate its color histogram and grayscale histogram
    c. Prepare the image for display
        i. Load with PIL format for easily pasting into mosaic later
        ii. Resize to desired display tile width (150) and crop squarely
    d. *Optional:*
        i. Visualize the histogram as a bar chart and save to disk
        ii. 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
    e. Add new Tile object to a dictionary in main where its key is its title
2. For the Base image:
    a. Initialize as Base object by passing in image path and title as parameters
    b. Analyze the image
        i. Load as Numpy array with OpenCV2
        ii. Resize to desired width (tile width of 30, multiplied by the desired number of columns in mosaic)
        iii. Save new height and width
        iv. 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
        v. *Optional*: Find list of most dominant colors in the tile image
        vi. Save number of rows and columns (where their product is the number of tiles needed to compose the image)
    c. Return Base object to main
3. For each quadrant histogram in the Base image:
    a. *Optional*: Dominant operation: Find best match by dominant color
    b. 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
    c. 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 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.

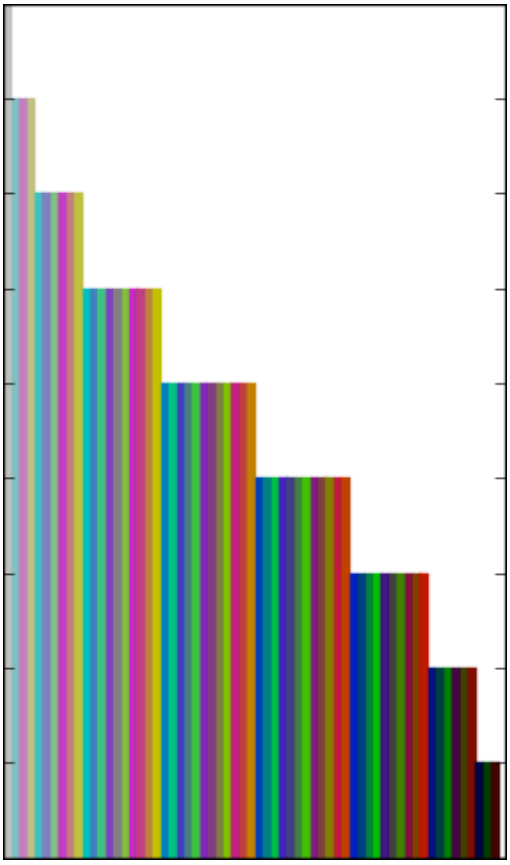
In the same spirit of data reduction, we chose the following bin size for calculating color histograms in similarity.py:

```
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 tile 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 a decent runtime.

As can be expected, we were inclined towards 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 regardless of 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 fewer 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]

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 on the left, representing the actual colors of our 64 bins, was generated such that 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
```

[1] Source: http://www.thealmightyguru.com/Games/Hacking/Wiki/index.php?title=NES_Palette

```
                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 them 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 for other formats. Because we changed transparent pixels to white, we had to return the (possibly) corrected image as well.

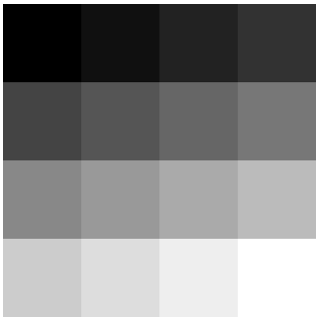
In addition to the histogram and the image, we also returned a list of color tuples for any colors with counts above 0 in the image. This critical line sorts the color tuples:

```
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 was 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 using 4 * 4 = 16 shades of gray bins. This essentially translated to a 4 bit grayscale palette, such as the shades on the right.

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 prevalent colors in the tile images, either framing 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) as 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 below 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
```

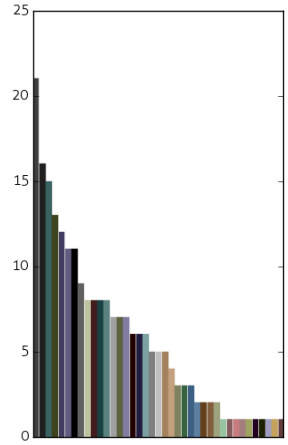
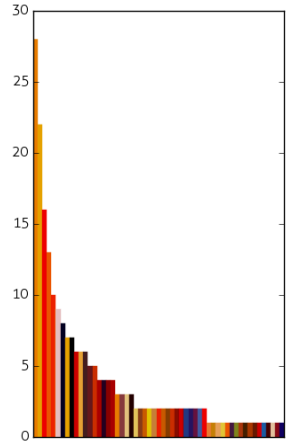
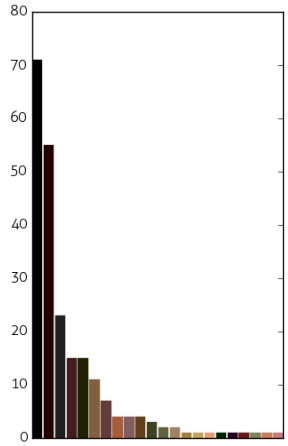
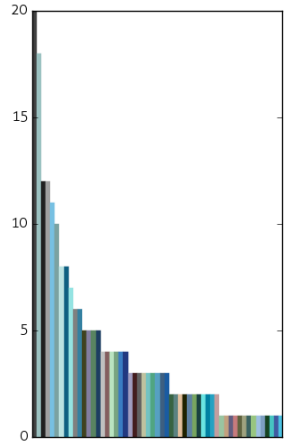
The first set of four images and color histograms were for a different image database, and used 8 bins. For these conditions, a threshold as low as 0.08 made sense, although we still had to discount black and white pixels because they often had fairly high ratios while it was hard to advocate for them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

Top 3 ratios
[0.093, 0.071, 0.067]

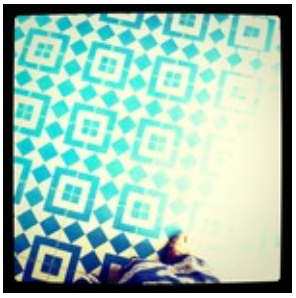
Top 3 ratios
[0.124, 0.098, 0.071]

Top 3 ratios
[0.316, 0.244, 0.102]

Top 3 ratios
[0.089, 0.08, 0.053]

In this set of images below (from our finalized database), we see justification for less bins. The colors cluster together, and a threshold of 0.3 seems reasonable for identifying a dominant color.



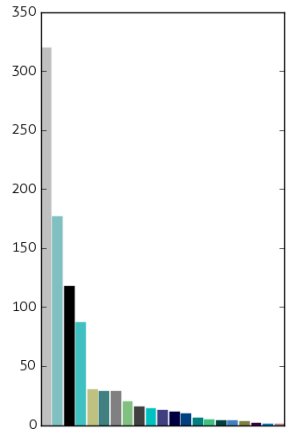
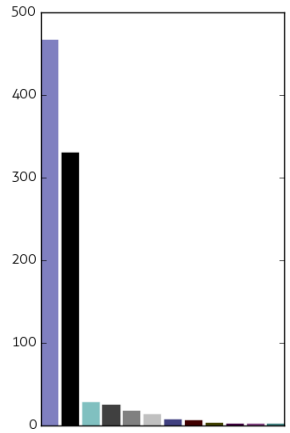
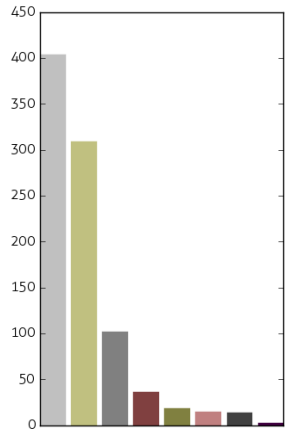
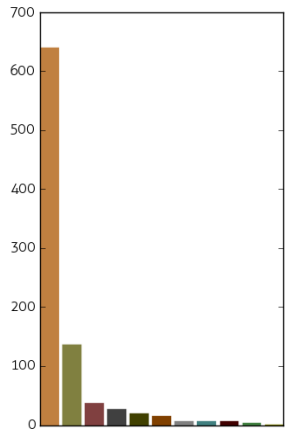


```
Top 3 for 1308516372
[ 0.456 0.302 0.131 ]
```

```
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]
```

Quick Note on K-Means Clustering

We tried color-based segmentation using K-Means clustering twice: once with a black box algorithm from the Scikit-Learning library (implemented in similarity.py) and in another raw attempt with named tuples by following an open source tutorial (implemented in dominance.py; tutorial link here: https://charlesleifer.com/blog/using-python-and-k-means-to-find-the-dominant-colors-in-images/). 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 using 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 use 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 color code is) as their dominant color, 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 seen the implementation of resize_by_w above in the reduction.py excerpt, so we won't go into that again.

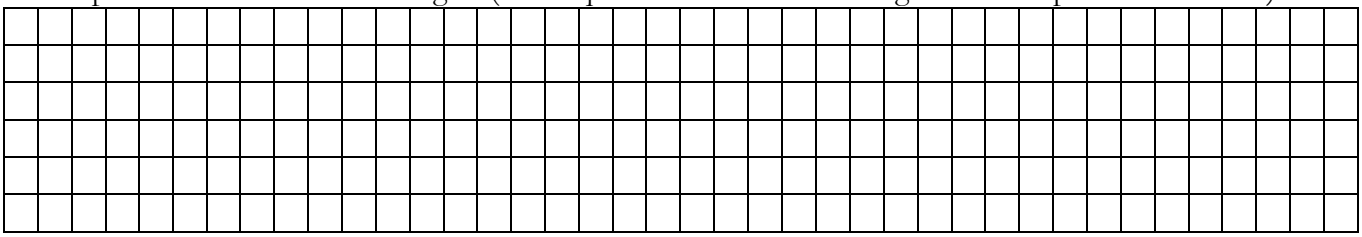
Instead, we will consider the value of DESIRED_COLS. The effect of increasing the number of DESIRED_COLS is an increase in accuracy and recognition, but also a large increase in time because we effectively have to calculate TILE_WIDTH * DESIRED_COLS of many histograms, and later many best matches. Take 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 * 6 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 cropped quadrant to calculate grayscale histograms, and use the 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 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 lengths of self.histograms and self.histograms[0] respectively.

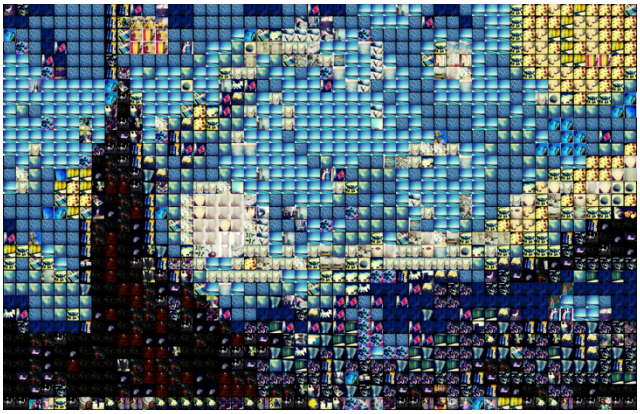
Note that the resize_by_w, crop color_histogram, grayscale_histogram, dominant_color histogram are all the exact same implementations as we used and described before.

Sneak Peek, and the Effects of Granularity and Changing `DESIRED_COLS` in base.py

Although we haven't discussed tile matching yet, below are some completed mosaics we have that showcase the importance of setting the number of desired columns variable in the base class.

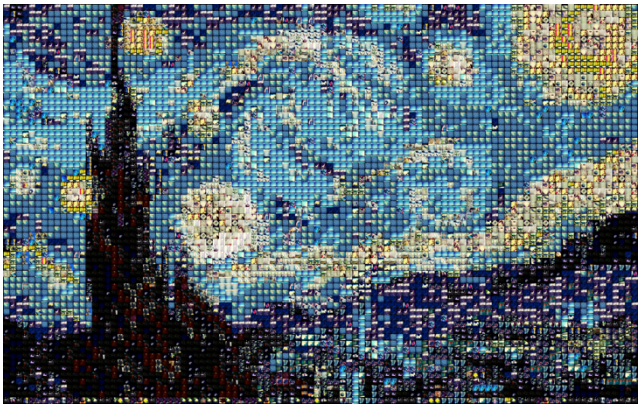
25 columns by 62 rows
1:35 minutes to generate

```
Percent of possible tiles used: 0.116, 58 out 500
images from tile library used

Expensive operations: 400 of 400 : 1.0
Dominant operations: 0 of 400 : 0.0
History operations: 0 of 400 : 0.0
```



50 columns by 32 rows
4:37 minutes to generate

```
Percent of possible tiles used: 0.210, 105 out 500
images from tile library used

Expensive operations: 1056 of 1600 : 0.66
Dominant operations: 543 of 1600 : 0.339375
History operations: 1 of 1600 : 0.000625
```



100 columns by 64 rows
8:03 minutes to generate

```
Percent of possible tiles used: 0.308, 154 out 500
images from tile library used

Expensive operations: 4057 of 6300 : 0.64396825396
Dominant operations: 2173 of 6300 : 0.344920634921
History operations: 70 of 6300 : 0.0111111111111
```

The last one is very close to van Gogh's *The Starry Night* (original depicted above)! Right?

This example hopefully illuminates the motivation behinds all this detailed set up, as well showcase what we meant in this section by the grid quadrants in the base image being replaced by image tiles.

4. Tile Matching and Mosaic Making

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. For this color similarity, we definitely needed to think of methods that didn't involve searching through all tile databases to reach each cell.

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 we 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's 100 * 60 * 500 = 3,000,000 comparisons being made in this step alone!

In addition to this performance problem, we had 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 the 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:

$$\text{L1\_norm} = \sum(\text{differences}) / \sum(\text{pixel count})$$

where

$$\text{L1\_norm} = \text{distance} = 1 - \text{similarity}$$

$$\text{similarity} = 1 - \text{distance} = 1 - \text{L1\_norm}$$

Since we found it more intuitive to work with distances than similarities, when 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

To advance the goal of performance improvement and temporarily ignoring the repeating image problem, we added the History Method. Namely, if a quadrant has same histogram as earlier one, then reuse the closest_tile variable for tile 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, we required 500 L1 norm 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 aesthetics 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

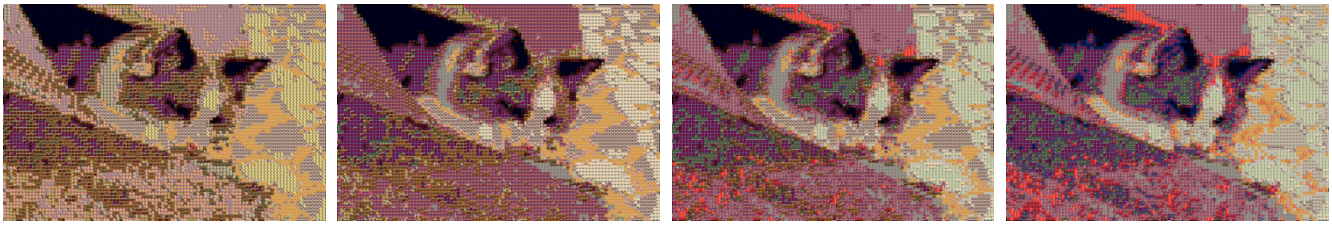
```
def l1_gray_norm(h1, h2):
    diff = 0
    total = 0
    for g in xrange(0, gBINS):
        diff += abs(h1[g]-h2[g])
        total += h1[g]+h2[g]
    l1_norm = diff/2.0/total
    return l1_norm
```

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 grayscale method was quite clear.

Can you tell what this is? Read on and validate your guess! It will show up in our User Studies later.

Below are examples of mosaics made using different levels of ALPHA. From left to right: ALPHA = 1.0, ALPHA = 0.75, ALPHA = 0.5, and ALPHA = 0.5. As ALPHA decreases, there is clearly deterioration in

the quality of the mosaic – some of the colors are noticeably off. This is due to the increasing weight on texture over color in calculating the L1 norm.



d. Dominant Color Method

All along, we were thinking that if we could find a way to translate each color into a single number in a way that is perceptually meaningful, we could use the that as the basis for a sorting technique to find the closest available tile. Had we more time, we would have tried to do this with the grayscale tiles, arranging them along a brightness spectrum, but since producing grayscale mosaics was not our main motivation, we turned to the dominant color as a possibility for sorting the tiles into dominant color bins and extracting random but close matches by color at near-constant time.

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 quadrant holds, see if there any tile images associated with that dominant color in the dictionary, and pick one! This is also nearly constant time lookup compared to the expensive method, since most images have 0-3 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 and debuggers to determine how often each method is actually being used over the life of the program.

```
# Find best tiles to recompose 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 "%d out of %d rows" %(len(the _chosen), base.rows)
```

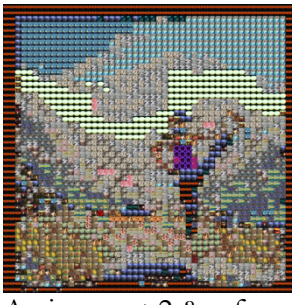
Fine-Tuning Database and Dominant Color Techniques

The grid below and evolution of mosaics somewhat describe our step process and challenges producing a dominant color method. We originally were using the images (though there are only 40 of them) from Assignment 2 plus a few other images we found online as our tile database. That database lacked a black image, so even though we chose to exclude black from the dominant method, the closest tile match from this database was still a photo of tomatoes with a black background. So we decided from there to zip an Instagram user's public directory.

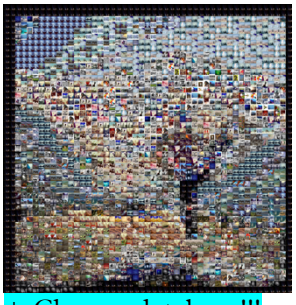
Original

All following mosaics are 50 columns wide

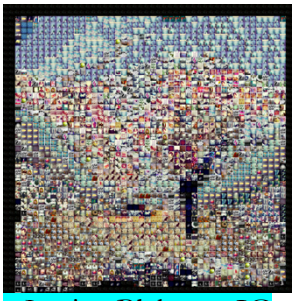
Assignment 2 images & a few additions
Threshold: 0.05



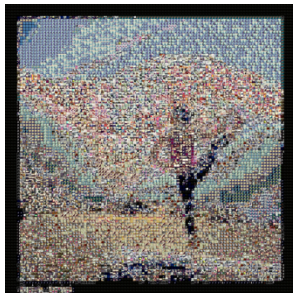
Assignment 2 & a few additions
Threshold: 0.1
+ Exclude black



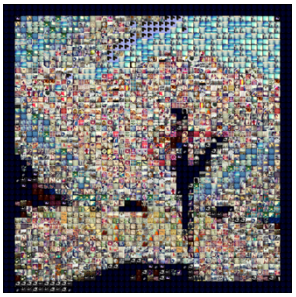
+ Change database!!!
NatGeo IG
Threshold: 0.1
Exclude black



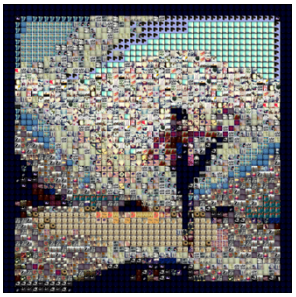
+Justina Blakeney IG
Threshold: 0.1
Exclude black
50 columns



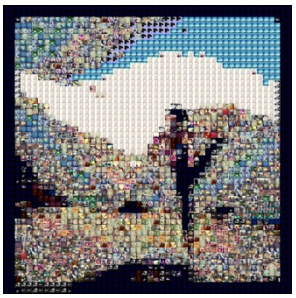
Justina Blakeney IG
Threshold: 0.1
Exclude black
100 columns



Justina Blakeney IG
Threshold: 0.1
Exclude black
50 columns
+ 4 bins instead of 8

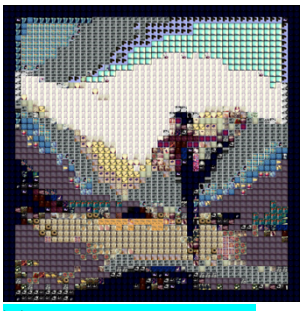


Justina Blakeney IG
+ Threshold: 0.3
Exclude black
50 columns
4 bins instead of 8



Justina Blakeney IG
Threshold: 0.3
Exclude black
+ Exclude white too
50 columns
4 bins instead of 8

```
Percent of possible
tiles used: 0.250, 125
out 500 images from tile
library used

Expensive operations:
215 of 2500 : 0.086
Dominant operations:
1009 of 2500 : 0.4036
History operations: 1276
of 2500 : 0.5104
```



Turn off dominance method because our system's goal is recognition and this is clearer (randomization was artful though)

```
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
```

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']]
```

Given 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:

```
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"
```

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

```
    f = open('mosaic_keys.txt', 'w')
    f.write(str(the_chosen))
```

Finally, we also calculated the percentage of the tile library that was actually utilized, and the relative percentages of expensive, dominant and history operations.

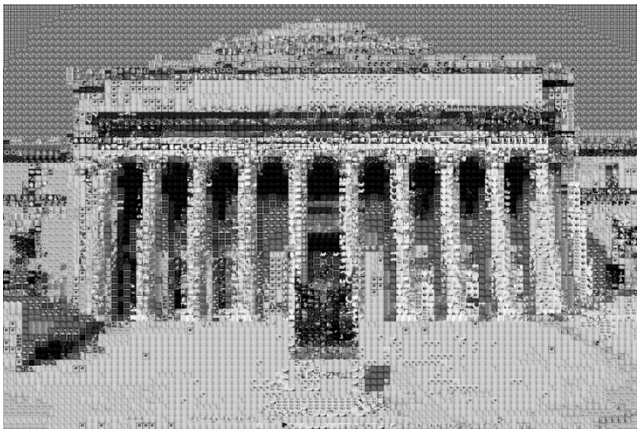
```
    n = len(set([img for sublist in the_chosen for img in sublist]))
    print "Percent of possible tiles used: %.3f, %d out %d images from tile library used"
%(round((float(n)/len(tiles)), 3), n, len(tiles))
    print ""
    print "Expensive operations:", expensive_count, "of", count, ":", expensive_count/count
    print "Dominant operations:", dom_count, "of", count, ":", dom_count/count
    print "History operations:", history_count, "of", count, ":", history_count/count
```

Comparison of Different Matching Methods

As one can appreciate by now, different techniques definitely generate different aesthetic effects and different performance characteristics (i.e. it took a different amount of time to create the photomontage). Also, different "kinds" of base images and tile images tend to do better with our method compared to others. We will explore some of the successes and failures of our system in the next section. For instance, if we were dealing purely with a grayscale mosaic with ALPHA = 0, then a high contrast black and white base image would probably work well.

Original image

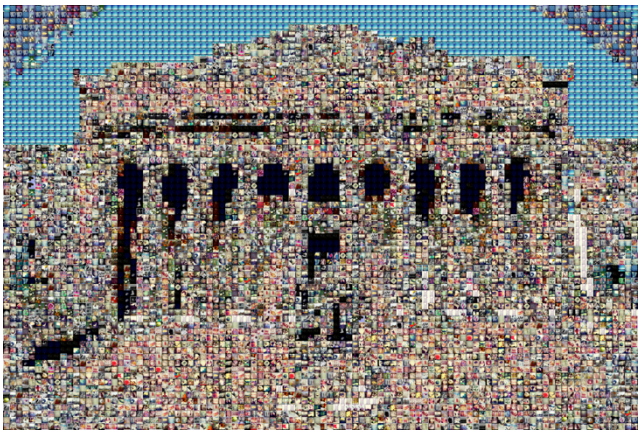
All the following images are 100 columns by 67 rows.



(ALPHA = 1, DOM_ON = 0)

Percent of possible tiles used: 0.792, 396 out 500 images from tile library used

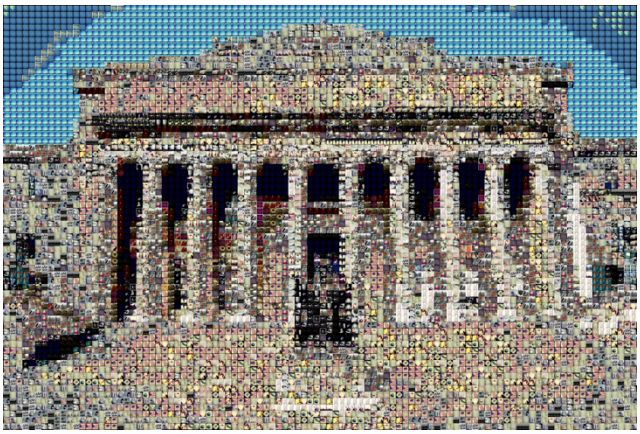
Expensive operations: 5749 of 6700 : 0.85805970149
Dominant operations: 0 of 6700 : 0.0
History operations: 951 of 6700 : 0.141940298507



(ALPHA = 1, DOM_ON = 1, DOM_COL_THRESH = 0.1)

Percent of possible tiles used: 0.798, 399 out 500 images from tile library used

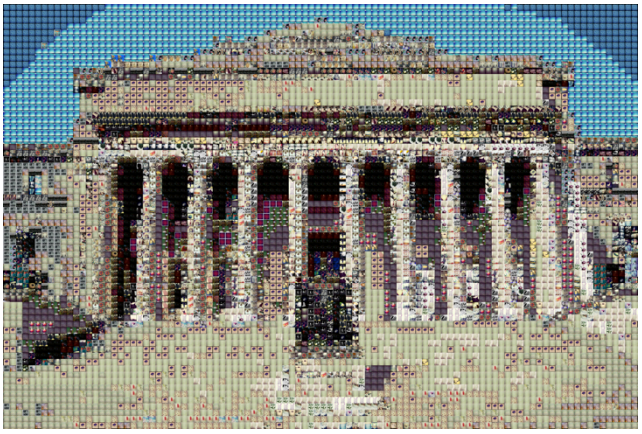
Expensive operations: 280 of 6700 : 0.041791044776
Dominant operations: 6194 of 6700 : 0.92447761194
History operations: 226 of 6700 : 0.0337313432836



(ALPHA = 1, DOM_0N = 1, DOM_COL_THRESH = 0.3)

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

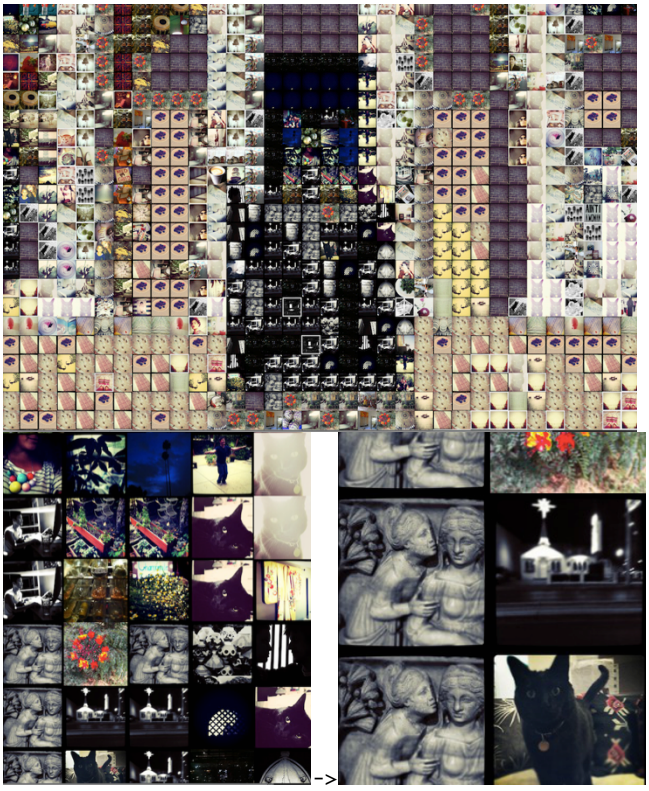


(ALPHA = 1.0, DOM_ON = 0)

Percent of possible tiles used: 0.466, 233 out 500 images from tile library used

Expensive operations: 5749 of 6700 : 0.85805970149
Dominant operations: 0 of 6700 : 0.0
History operations: 951 of 6700 : 0.141940298507

ZOOMING INTO THE ALMA MATER ---------->



Hopefully you recognized Low Library and the Alma Mater. All in all, the results, with the exception of the second mosaic, are quite good. The grayscale mosaic uses no dominant operations, but its expensive operations were quite fast because it only had to calculate the L1 norm along one axis, and it uses 80% of the image database. Had grayscale images been our focus, we could have used a better sorted list of tiles by brightness to achieve even faster performance.

For all four of these mosaics, there were very few history operations, meaning that the base image was quite textured and its quadrants didn't have many identical histograms. However, thanks to the pickling, generating these four images didn't take more than 15 minutes despite the lack of history operation lookups, as we got to skip straight to the image matching section from the second collage onwards.

For the second image, the dominant color threshold is too low, but raising it to 0.3 in the third picture produces fuzzy yet aesthetically pleasing results that use almost half of the database images to achieve its colorful yet still defined aesthetic. Some might even say that those results are more artful and impressionistic, but since our system sought precision and recognition, we settled on the settings of the last image, which produced the clearest result.

Finally, the last collage, computed purely by RGB color similarity along the L1 norm, shows the final configuration we used for the mosaics in our User Tests and System Evaluation (coming up next!). It uses no methods based on dominant colors, and uses almost half of our database, so it has a good variety of images, as you can see when you zoom in and espy the cuddling statues and pitch-black cat.

To comment further, setting the desired number of columns for the base image was one of the key decisions we had to make toward increasing precision and recognition for our mosaics. Increased number of desired columns, corresponding with increased granularity of the image, resulted in much more clearly depicted imagery. As can be expected, with this improved aesthetic performance, however, came the cost of diminishing runtime performance. Times tended to double with the doubling the number of columns compared to a previous mosaic generation, while the number of operations that had to be made later in the tile matching phase quadrupled. Consequently, it became critical for us to find different ways to reduce the runtime for tile matching, as we will discuss in the next section. For the first image, all the 400 operations 25 columns wide is barely recognizable, for 50 columns people can hazard a good guess, but 100 columns is nearly unmistakable – so that is the value we chose to go with during the image processing of the base image object class.

Pickling the Tiles and Base Objects

To improve the performance of our system and reduce runtime, we pickled the tiles dictionary and base image object, saving them in .p files based on the names of the tiles and base path. In main.py, the system initially checks if the pickle exists, and if so, retrieves all the data from the binary stream. If it doesn't exist, then all the processing steps as described above take place, and the objects gets dumped into pickle files.

As a quick example of how the Base Image object is set after we added the pickling:

```
# Check if pickle file exists first
  if os.path.exists(base_ppath):
      base_pickle = open(base_ppath, "rb")
      base = pickle.load( base_pickle )
      base_pickle.close()
      print "Reloaded pickled file."
  elif os.path.exists(base_path):
      base = B.Base(base_path)
      pickle.dump( base, open( base_ppath, "wb" ) )
  else:
          sys.exit(base_path + " does not exist")
```

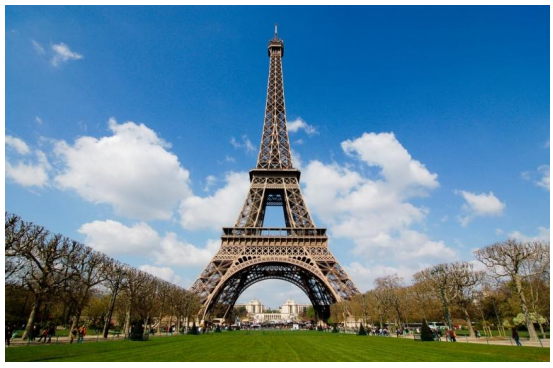
5. User Evaluation

Earlier, we stated that the goal of a Mosaic Maker is to generate a mosaic that resembles a base image when viewed from far away. Hence, the main feature we tested in this section was whether users could recognize and describe a series of base images.

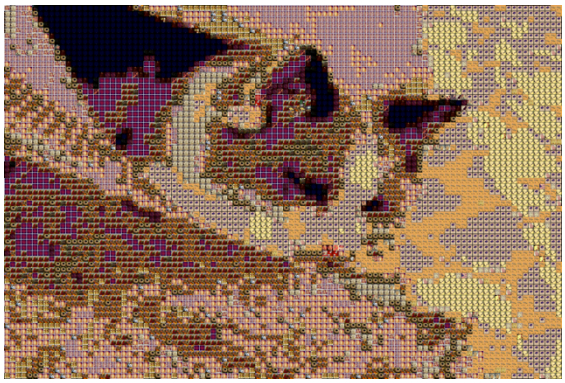
Base Image Set:
Twenty base images were used for user testing. We divided the images into sets based on the subject(s) of the image, as we believed that users would have differing levels of success in recognizing the mosaics of these images based on their content.

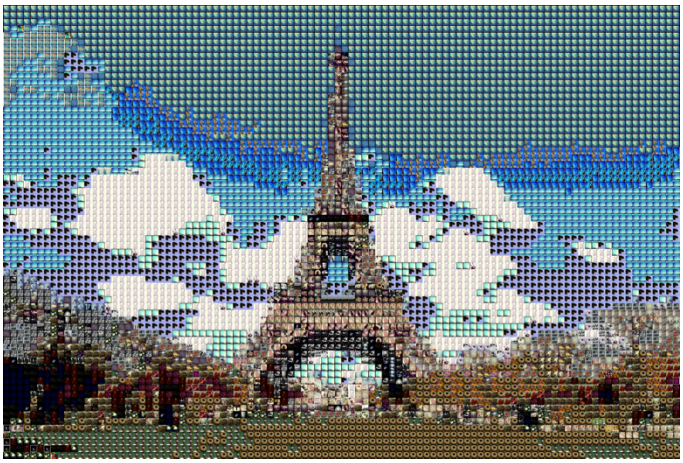
Set A: Highly Recognizable Entities, Single Subject in Photo, Simple Background
This set includes photos of highly recognizable entities, such as famous people (ex. Barack Obama), objects (ex. the Eiffel Tower), and organisms (ex. a cat). Each image only contained one subject, and the background was relatively simple and did little to distract from the main subject of the photo. We predicted that all users would be able to easily recognize these entities from their mosaics.

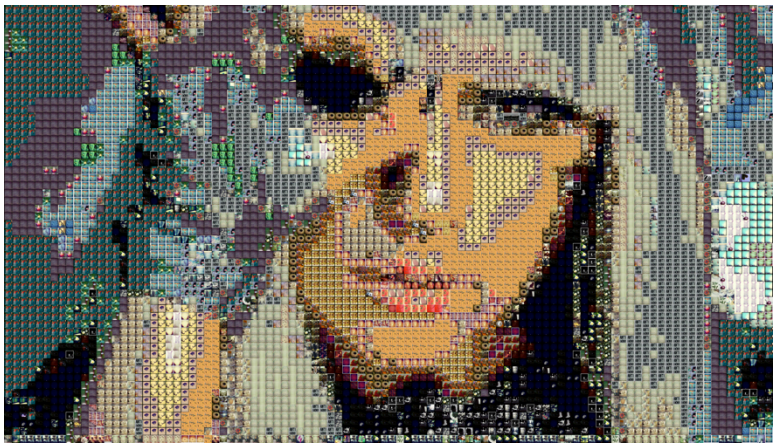




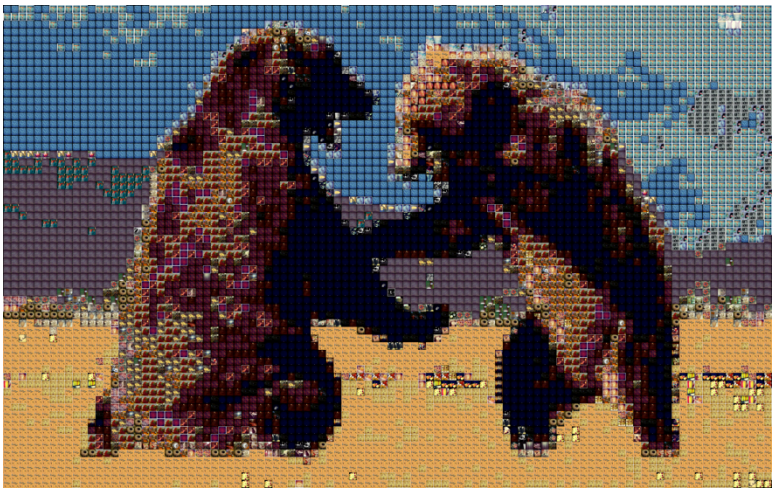

The mosaics for these images are shown below:

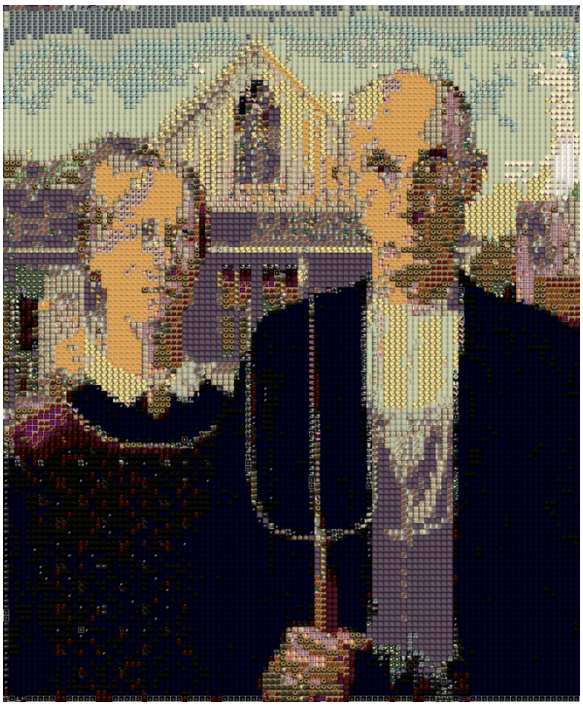


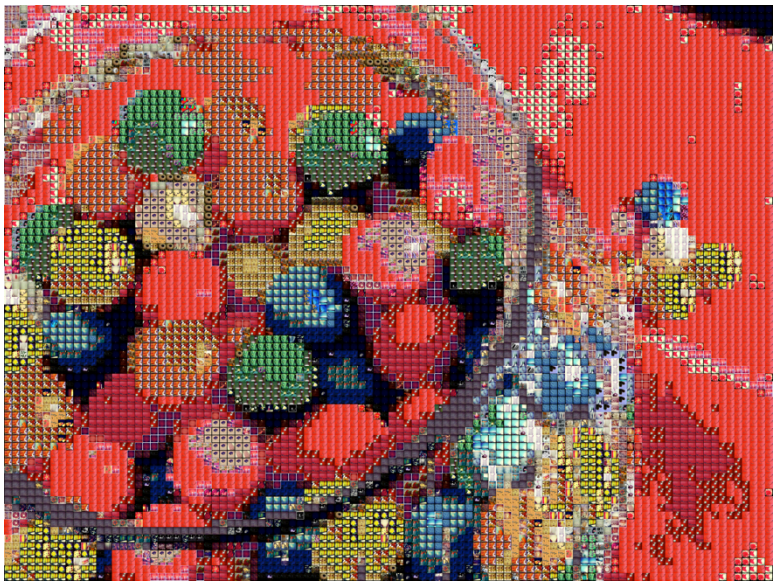





Set B: Highly Recognizable Entities, Multiple Subjects in Photo, Simple Background
To examine whether our system could handle more complicated photographs, this set includes highly recognizable entities, with multiple prominent subjects or objects in the base photo. The background was again simple and did not overly distract from the main subjects in the image. We also predicted that users would have a relatively easy time recognizing theses subjects from their mosaics, or at least provide a reasonable description (ex. "two people getting married").

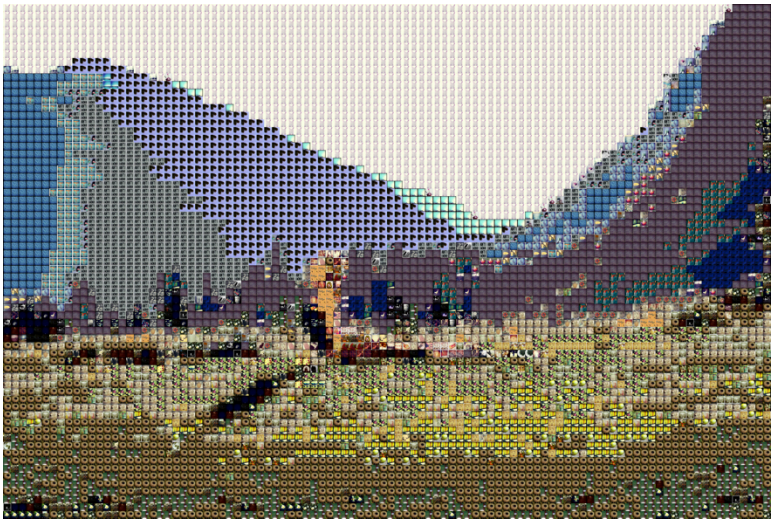


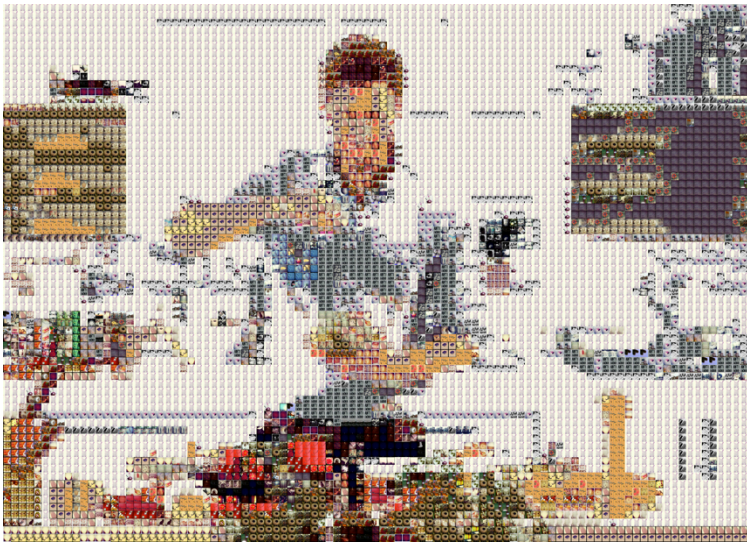


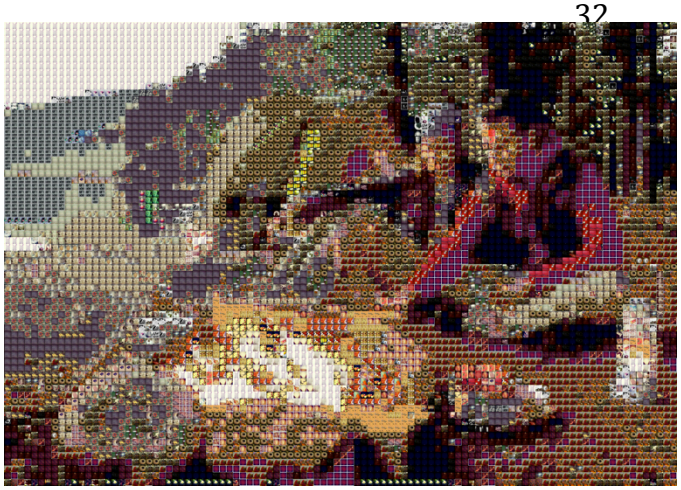


Set C: Less Recognizable Entities, More Prominent and Complex Backgrounds
For this section, we decided to push the limits of our system and use more complex photos. The subject(s) featured in this photo set are not well-known (for example, if the subject is a person, they might be a stock photo model), and the background may have a more prominent presence in the picture and possibly distract the viewer away from the main subject(s). We predicted that users would have a moderate to difficult time identifying the content of the mosaics.

Their mosaics:



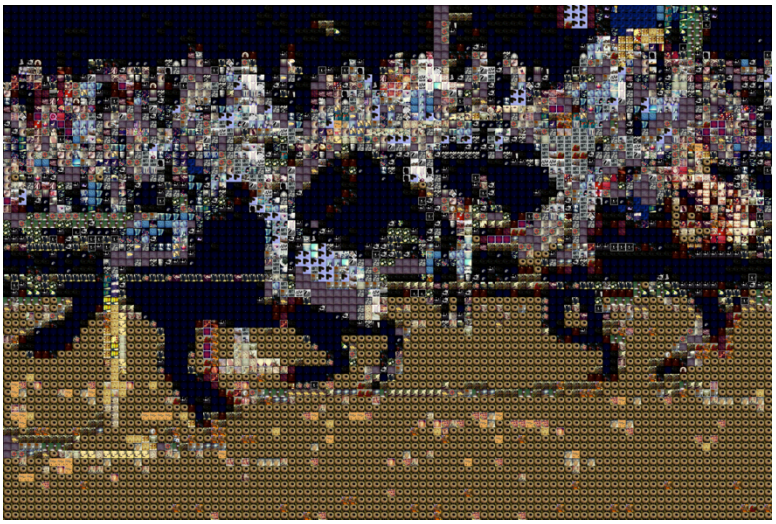


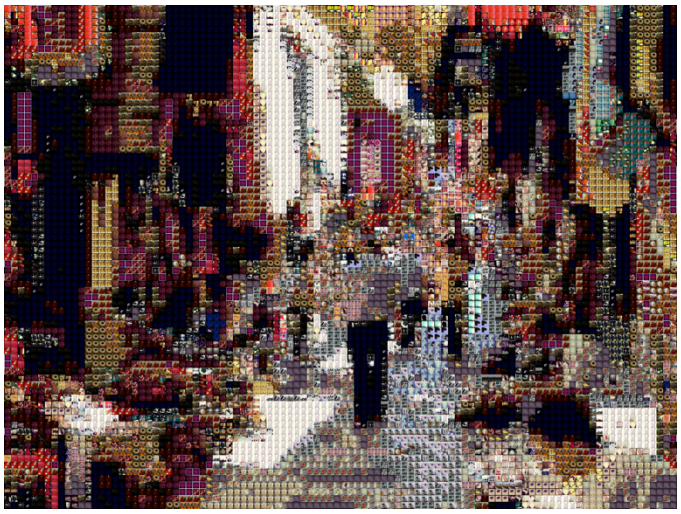


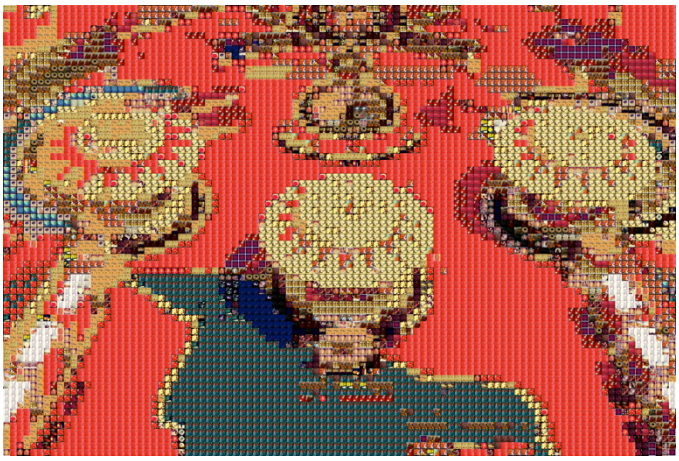
Set D: Complex Scenes

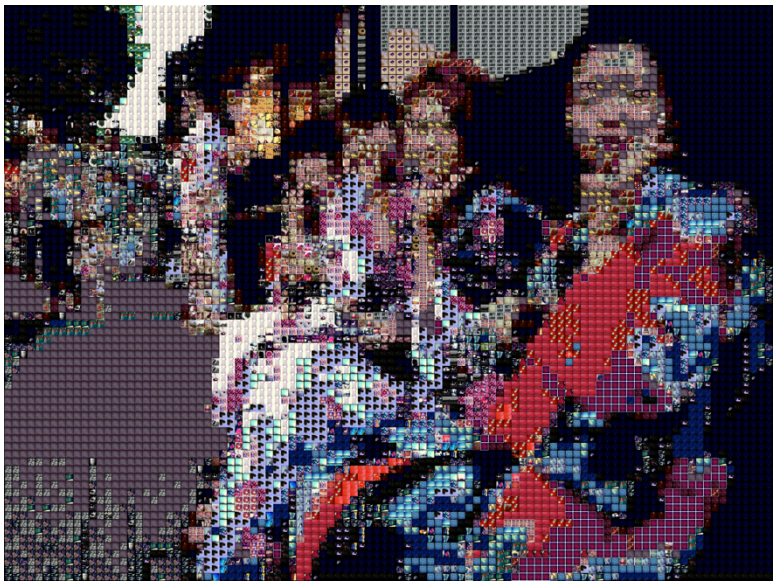
This set is a notch above Set C, and includes more complex scenes such as scenes capturing movement (ex. an active sport), elaborate backgrounds, and/or photos with many subjects in them. We predicted that users would have a lot of trouble identifying and describing the content of these mosaics.

Their mosaics:



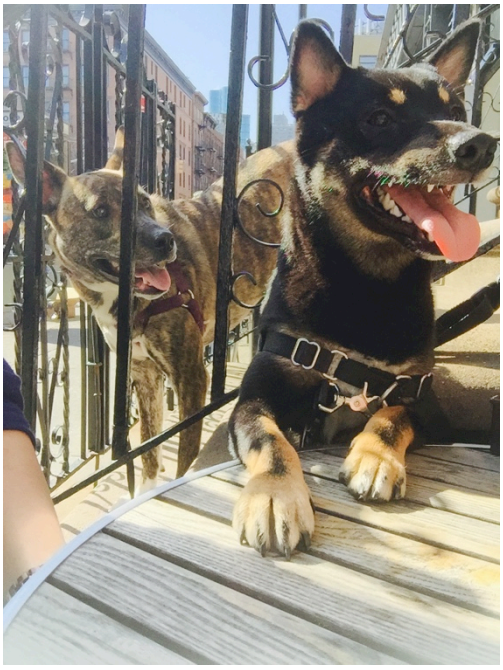




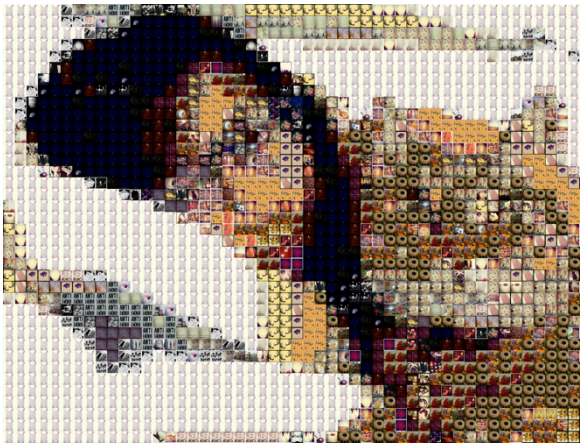


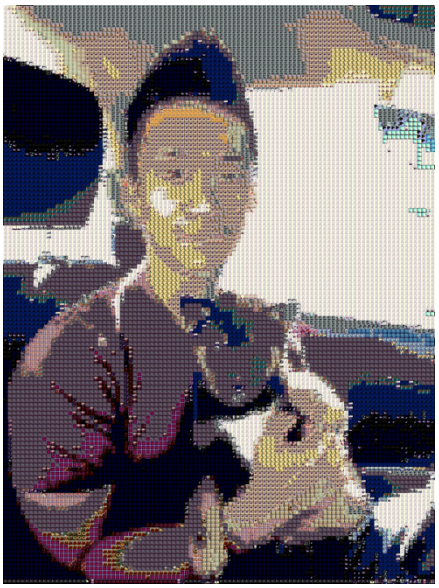
Set E: Personal

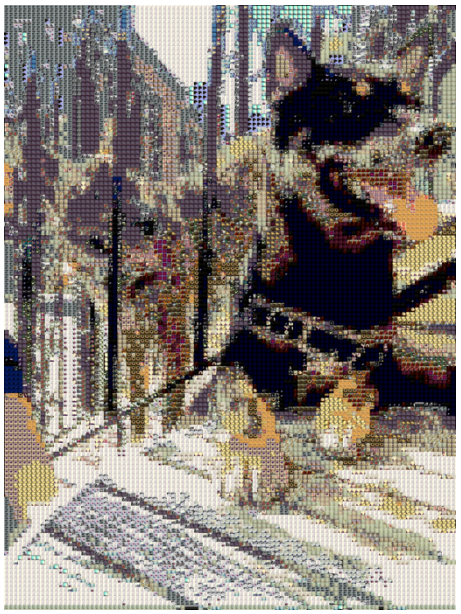
Since mosaics often use familiar subjects as the base image, this set includes mosaics of people whom the user knows personally; the specific subjects presented depended on the user being tested. The Set E that was used for Users 3 and 4 is shown below. We predicted that all users would have little to no difficulty identifying the subjects and contents of these mosaics.



Their mosaics:







The correct answers for the photos are as follows:
A1 A cat/Cero
A2 Obama
A3 Eiffel Tower
A4 Lady Gaga
B1 Kate and William/the Royal Couple
B2 Two Bears Wrestling/Fighting/Interacting
B3 Gumballs/Candy
B4 American Gothic
C1 A person doing yoga in the mountains
C2 A man cooking in the kitchen
C3 Four turtles on a log in water
C4 Camping outdoors
D1 Two knights jousting
D2 An outdoor marketplace
D3 A pinball machine
D4 Japanese women in kimonos posing for a photo
E1 Nina at her graduation
E2 Nina with Molly
E3 Robert and Happy
E4 Happy and Mila
E1 (second set) Margaret
E2 (second set) William
E3 (second set) Emily
E4 (second set) Melanie

Four users were tested, and their results are shown below:

Photo	User #1	User #2	User #3	User #4
A1	Cero	Cero	Cero	Cat
A2	Obama	Obama	Obama	Obama
A3	Eiffel Tower	Eiffel Tower	Eiffel Tower	Eiffel Tower
A4	Gaga	Iggy Azalea	Sia	Lady Gaga
B1	Kate and the Prince	Royal Couple	Prince and Princess at a Wedding	Prince of England, Marriage
B2	Two Bears Talking	Two Bears Fighting	Bears Fighting	Two Grizzly Bears
B3	Candy	M&Ms	Marbles	Gumball machine
B4	American Gothic	Famous painting of the farmers	That Famous Painting, Holding a Pitchfork, Old Man and Woman	The painting with the pitchfork
C1	Mountains with a house in the foreground	Mountains, with a bird in the foreground	Mountain, something with a shadow	Yoga, outside in the mountains

C2	A guy cooking in the kitchen	Man in the kitchen blending	Man cooking something with boats or harbor in the background	Some white dude cooking
C3	Three ducks on water	Rocket in space	Spaceship, aircraft	Starship
C4	Guy and girl sitting in front of a campfire outdoors	Couple in front of a bonfire in the forest, next to a lake	Couple camping	Dragon fire
D1	Two dudes on horses	Two knights jousting	Horse derby	Horse racing
D2	An outdoor clothing market	Traditional market	Person in a narrow alley with storefronts	Person in bottom right, storefronts
D3	An amusement park	Dining tables with big umbrellas over them	Huts	Clocks, doorway
D4	Japanese women sitting outside, wearing kimonos, posing using peace signs	Japanese girls in kimonos smoking	Geishas	Family doing kissy faces
E1	Margaret	Margaret	Picture of Nina Holding Flowers and her Parents Taken at Columbia During Her Graduation	Graduation
E2	William	William	Nina and Molly the Cat on the Couch	Nina and Molly sleeping
E3	Emily	Emily	Robert and Happy	Robert and Happy
E4	Melanie	Melanie	Happy (1 shiba Inu, black) - Missed the other dog in the picture, Mila, a brindle mix, because she looked like part of the background	Happy and Mila
Accuracy	17/20 = 85%	17/20 = 85%	15/20 = 75%	17/20 = 85%
Precision	19/20 = 95%	18/20 = 90%	16/20 = 80%	15/20 = 75%
% Correct	16/20 = 80%	15/20 = 75%	9/20 = 45%	8/20 = 40%

The red cells indicate cases where the user gave an incorrect description of the main subject in the photo. Each cell represents a 5% deduction in accuracy (20 photos * 5% = 100%). The orange cells indicate cases where the user gave a correct description of the main subject, but their description was either insufficient or included incorrect details (for example, User 2 said the geishas were smoking, when in fact they were using peace hand signs). Each of these orange cells represents a 5% deduction in precision. If the user was

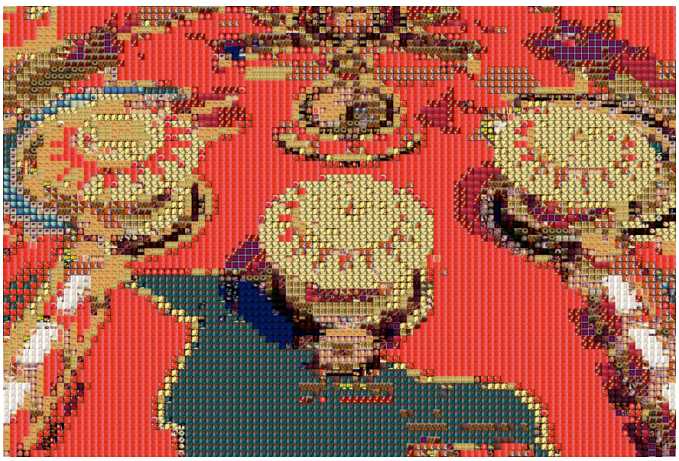
both wrong and gave too short of a description (when evaluating their description against the correct answer), accuracy and precision were deducted by 5% each.

If the % correct is based on both accuracy and precision, then the results are extremely poor – Users 1 and 2 did fairly well for both accuracy and precision, but Users 3 and 4 did pretty badly, getting over half of the images incorrect. In general, the users did very well on categories A, B, and E, and stumbled the most on sets C and D. Most notably, everyone got images C3 and D3 wrong, and only one user got C1 correct.

The unrecognizability of C3 probably reflected a limitation in the base image color distribution. As our database had a relatively fewer green and blue images, and pretty much everything in C3 was green, the turtles in the picture were drowned out by the background. This most likely hampered users' ability to recognize the subjects of this image. For photos that are not as monochrome, recognizing the main subject(s) is not an issue; however, for C3, the background would've been a cue to the users that the subject(s) of the photo were aquatic-related.

The low contrast in image C1 also made it very difficult for users to identify the subject of the image. Since our algorithm tended to pick dark-colored matches for the green and blue areas in the image, the images that contained prominent amounts of green and blue tended to suffer from a further loss of brightness and contrast. This likely contributed to the difficulty of identifying the person in C1. In the original, bright and high-contrast image, she could be identified very quickly, but in the darkened, lower-contrast mosaic, she was partially swallowed up by the background. Like with C3, using a database with a more diverse selection of green and blue images, and more importantly picking a brighter and high-contrast base image would probably have mediated this problem.

For D3, the main issue is a loss of precision; if the numbers on the pinball machine hadn't been lost in the photo to mosaic conversion, most of the users would probably have gotten D3 correct. However, loss of precision does not necessarily reflect an issue with the mosaic-making algorithm. When constructing a photomosaic, some details of the original image are necessarily sacrificed: it is very difficult to find tiles that will allow every little detail of the original photograph to be preserved, such as the words on a sign. Indeed, even if we used 500 tiles to represent D3 instead of 100, the numbers on the machine are still missing from the mosaic:

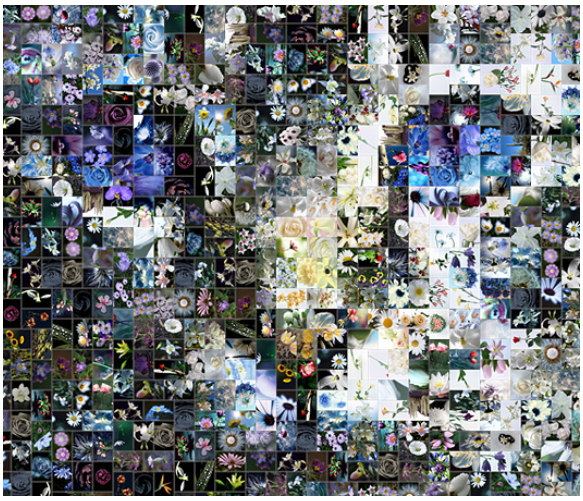
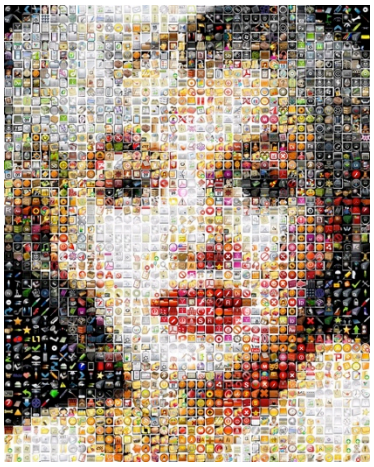


Moreover, since people tend to make photo-mosaics using either highly recognizable entities or photos with a single, prominent subject, it may have been unrealistic to expect users to describe some of the scenes in sets C and D with a high degree of precision. A Google Image search of photo-mosaics supports

our belief that people tend to make mosaics of highly recognizable entities that can be described in 1-2 words, as opposed to complex scenes with many subjects and elements. We believe that some of the test images we used (particularly sets C and D) do not reflect the type of base images that people tend to use for mosaics. Thus, the low precision and combined % correct measures we obtained are not necessarily indicative of the performance of our system.

Based on these considerations, it seems reasonable to evaluate our system based on accuracy alone. Accuracy-wise, then results are fairly good: the four users have an average accuracy of 82.5%. Users for the most part were able to figure out the general subject of the mosaics, and when it came to recognizing famous entities and people whom they were personally familiar with, all of the users did very well.

Below: Google Image search for mosaics

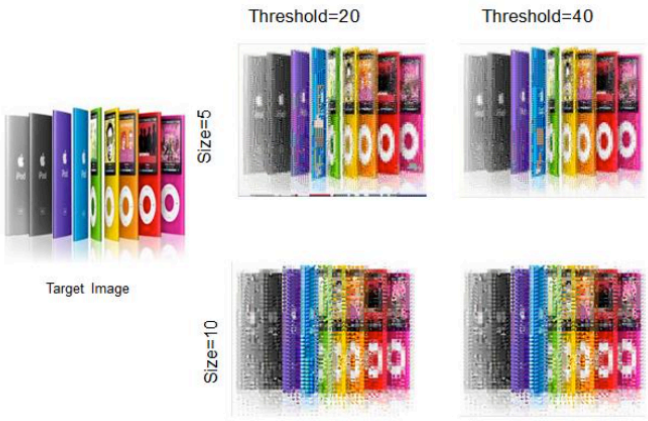

Another evaluation metric of our system is tile variety. A wide variety of different tiles were clearly used to construct the three above mosaics. To compute the database utilization ratio, we check how many different tiles appear in a mosaic, and take that as a ratio of the total number of images in the database:

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

Below are the database utilization measures of the four Set E images we included above:

E1: Percent of possible tiles used: 0.518, 259 out 500 images from tile library used
E2: Percent of possible tiles used: 0.344, 172 out 500 images from tile library used
E3: Percent of possible tiles used: 0.342, 171 out 500 images from tile library used
E4: Percent of possible tiles used: 0.462, 231 out 500 images from tile library used

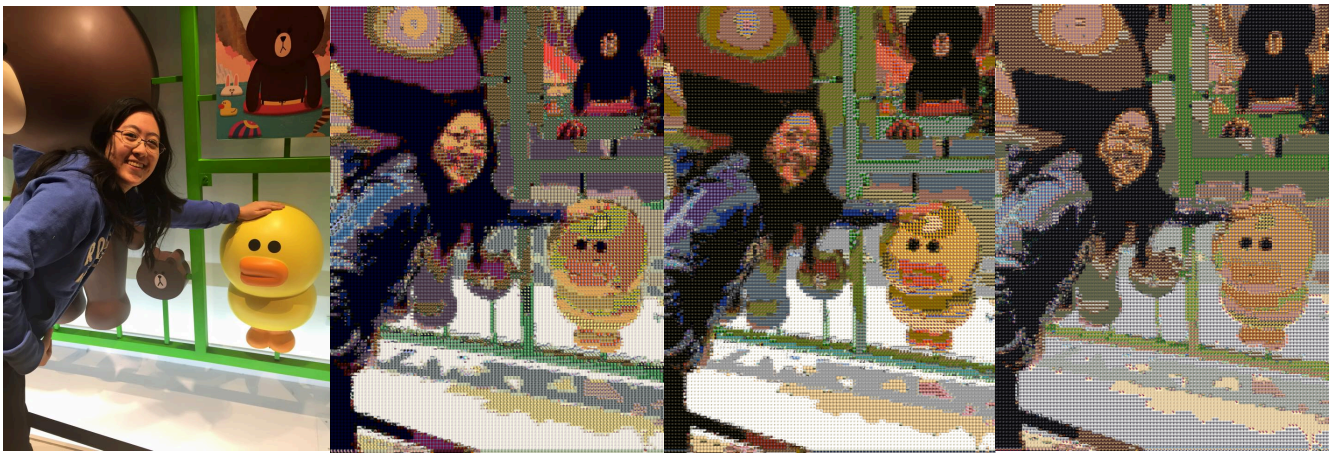
These four images use between 34.2 to 51.8% of the Justina Blakeney database, which contains 500 images. Given a tile width of 100, each mosaic contains somewhere on the order of magnitude of 100^2 or 10,000 tiles; that means, on average, each tile is repeated around 40-60 times. Our average repeat rates are comparable to those used by Shah, Gala, Parmar, Shah, and Kambli (2014), who found that using a maximum of 40 repeats per tile produced a smooth mosaic. Depending on the source image, however, it's possible that a handful of tiles could be repeated hundreds of times in our mosaics: in E3, the same white tile was used for the background hundreds of times, causing the background to look very flat.



Given that the background of the original image was a glaring, bright white, there were probably very few other tiles that could've been a good match for this kind of background. However, allowing high potentially high levels of tile repetition have likely caused our mosaics to have less aesthetic appeal compared to the three mosaics shown above.

We did not explicitly set a maximum threshold for the number of times a tile can be used in an image, because we discovered through earlier experiments that the second to fourth best tile matches for an image region tended to be far worse matches compared to the "best match." With a larger database containing more diverse images, it may be possible to set a maximum of 40-50 uses without compromising the quality of our mosaics.

Lastly, the quality of mosaics is heavily dependent on both the color distribution of the image and the type of database used to generate the mosaic. Below is the colorful E4 image used for Users 1 and 2 (single subject - person, relatively simple background) and four different mosaics (generated using the Justina Blakeney, National Geographic, and muradosmann databases respectively).



While Users 1 and 2 were able to quickly recognize the subject of the Justina Blakeney mosaic during user testing, this database clearly had trouble finding matching brown colors, as the large bear in the left-hand side of the image turned purple in the mosaic. The National Geographic database produced a mosaic that was smoother and closer in color to the original, with more precise brown, blue, and green color matches. Note, however, the increased amount of noise on the yellow chick; not everything in the mosaic was improved as a result of switching to this database. The smaller muradosmann database generated a mosaic

that was significantly dimmer and less contrasted than the original image.

The database utilization ratios were pretty high, due to the image having a lot of color and gradations:
JB: 0.438, 219 out 500 images
NG: 0.480, 240 out 500 images
M: 0.568, 154 out 271 images

Contrast this against the E1 image used for Users 1 and 2, whose main colors are localized on the yellow-orange parts of the color wheel:

For E1, the National Geographic mosaic is the most similar to the original image; this is most apparent when the three mosaics are viewed from afar. The muradosmann database is again the worst, as a some of the major details have disappeared from the mosaic, such as the pink of the person's lips.

As expected from the lower variety of colors, E1 used far fewer database images compared to E4.
JB: 0.246, 123 out 500 images
NG: 0.242, 121 out 500 images
M: 0.280, 76 out 271 images

Since different databases are optimal for different base images, one possible system extension could involve generating multiple mosaics for each base image, then picking the best mosaic based on its color similarity to the original base image. Or, perhaps we could use a bigger database.

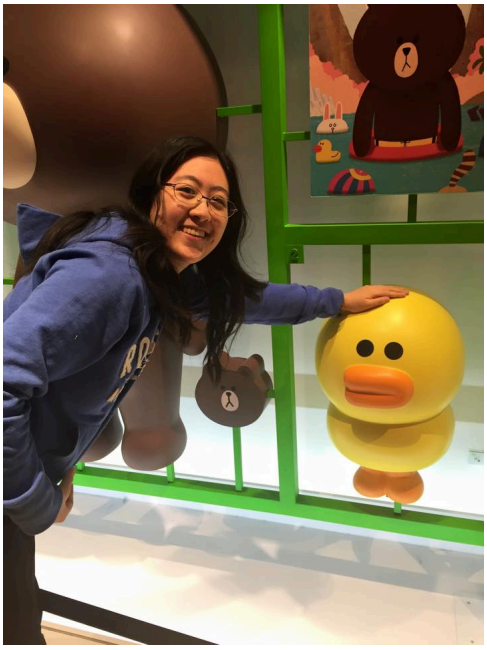
To test the effects of using a larger image database, we changed the code in main.py to look up the first 1000 images in the database instead of the first 500 images, with the following results for E4:

The quality of the mosaic has increased slightly; for example, the shadows on the ground look smoother and more similar to the shadows in the original photo. Doubling the effective image database size however caused the runtime to increase from just under ten minutes to nearly twenty minutes, and only slightly increased the number of different tiles used. Database utilization was now at 0.290, 290 out of 1000 (up from 219 out of 500). Such a small increase in diversity is not worth taking such a large decrease in runtime of the program, especially since the users didn't have trouble recognizing its subject when only the first 500 images of the database was used to make the mosaic. Of course, the extent of quality improvement likely depends on the picture and database used.

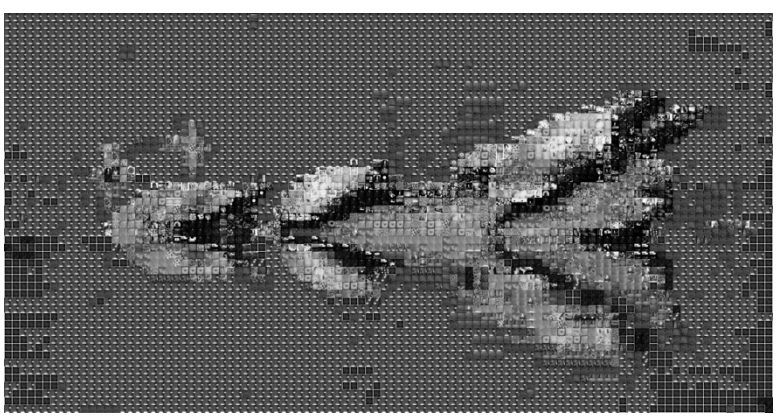
As for tile size, we experimented on using a larger tile size for the mosaic. Here are the results of using a tile size of 50:

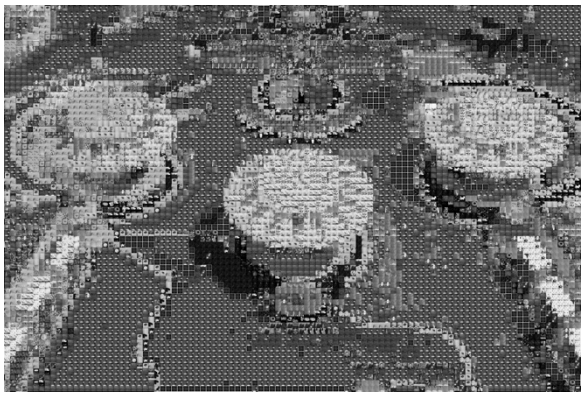
There was no decrease in quality; however, using a tile size of 50 doubled the runtime, so for the sake of efficiency we stuck with a tile size of 30.

Finally, using grayscale instead of color mosaics yielded the following mosaic:



Database utilization was fairly high (220/500 images – nearly 50%, better than the color mosaic), and the mosaic looks more natural, especially since the turtles are no longer swallowed by the background. It's notable that using a grayscale mosaic resulted in similar levels of database utilization as doubling the effective size of the database; perhaps some of our test mosaics can be improved by making them grayscale instead of color. However, this technique could come with a loss of precision: for this image, the mosaic is noticeably darker than the original photo, and the entities in the poster in the upper-right hand corner are a lot more difficult to identify. Below are some more grayscale versions of our test images:





Some of the most difficult images in our user tests database seem to benefit from the grayscale treatment. For instance, the awkward color of the water is no longer visible in C3's mosaic, making it easier to see the log and turtles in the picture. However, D1 was negatively affected by the grayscale treatment; the knights in the foreground are nearly indistinguishable from the audience in the background. To test whether the images are really easier to identify from a user standpoint, I asked three additional users to describe the four images shown above.

	User 5	User 6	User 7
C1	A house on the prairie	Deer in the woods	Yoga on a field with shadow
C3	A series of pacmen	Sideways tree	Spaceship
D1	A horse race	Horses running at each other	Horse race
D3	Amusement park	Tables	Pinball machine

One user guessed D3 and C1 correctly; User 5 also guessed "pinball," but they ultimately settled on

amusement park as their final answer. It seems that using grayscale mosaics did not significantly increase the recognizability of these complex images. However, depending on the picture, a grayscale mosaic may yield better results than a color one; this would require high contrast source images, where the foreground and background objects can be easily distinguished from each other.

Overall, we believe that our system did well given its database limitations, since users were able to identify the subjects of the base image when prominent subjects were used, and mosaic makers tend to use prominent and easily identifiable subjects in their base images.

References:

Shah, J., Gala, J., Parmar, K., Shah, M., & Kambli, M. (2014). Range Based Search For Photomosaic Generation. International Journal of Advanced Research in Computer and Communication Engineering, 3(2).

6. Appendix – Code Listing

base.py

```
import cv2
import reduction as R
import similarity as S
from dominance import colorz

TILE_WIDTH = 10
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 = []
	  # Equivalent to for(int j=0; j<self.height-TILE_WIDTH; j+=TILE_WIDTH)
	  # Advantage of this way is if self.width%TILE_WIDTH != 0
	  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])
```

- -

dominance.py

```
"""
Source: https://charlesleifer.com/blog/using-python-and-k-means-to-find-the-dominant-colors-in-images/
"""

from collections import namedtuple
from math import sqrt
```

```
import random
try:
    import Image
except ImportError:
    from PIL import Image

Point = namedtuple('Point', ('coords', 'n', 'ct'))
Cluster = namedtuple('Cluster', ('points', 'center', 'n'))

def get_points(img):
    points = []
    w, h = img.size
    for count, color in img.getcolors(w * h):
        points.append(Point(color, 3, count))
    return points

rtoh = lambda rgb: '#%s' % ''.join(('%02x' % p for p in rgb))

# Slightly modified to convert numpy array to PIL image
def colorz(arr, n=3):
    # img = Image.open(filename)
    # img.thumbnail((30, 30))
    img = Image.fromarray(arr)
    w, h = img.size

    points = get_points(img)
    try:
        clusters = kmeans(points, n, 1)
    except (ValueError, ZeroDivisionError) as err:
        n = 1
        clusters = kmeans(points, n, 1)
    rgbs = [map(int, c.center.coords) for c in clusters]
    return map(rtoh, rgbs)

def euclidean(p1, p2):
    return sqrt(sum([
        (p1.coords[i] - p2.coords[i]) ** 2 for i in range(p1.n)
    ]))

def calculate_center(points, n):
    vals = [0.0 for i in range(n)]
    plen = 0
    for p in points:
        plen += p.ct
        for i in range(n):
            vals[i] += (p.coords[i] * p.ct)
    return Point([(v / plen) for v in vals], n, 1)

def kmeans(points, k, min_diff):
    clusters = [Cluster([p], p, p.n) for p in random.sample(points, k)]

    while 1:
        plists = [[] for i in range(k)]

        for p in points:
            smallest_distance = float('Inf')
            for i in range(k):
                distance = euclidean(p, clusters[i].center)
                if distance < smallest_distance:
                    smallest_distance = distance
                    idx = i
            plists[idx].append(p)
```

```
        diff = 0
        for i in range(k):
            old = clusters[i]
            center = calculate_center(plists[i], old.n)
            new = Cluster(plists[i], center, old.n)
            clusters[i] = new
            diff = max(diff, euclidean(old.center, new.center))

        if diff < min_diff:
            break

    return clusters
```

- -

ig_zipper.py

```
import requests
import json
import sys
import urllib2
import os
import zipfile


'''
Makes Requests to Instagram API
Takes 2 command line arguments

argv1 = gallery_name
argv2 = access_token

stores thumbnail sized images in handle_response()
'''
gallery_name = sys.argv[1]
access_token = sys.argv[2]

def fetch_user_id():
    user_url = 'https://api.instagram.com/v1/users/search?q=%s&access_token=%s' %(gallery_name,
access_token)

    make_request(user_url, None)

def start_instagram_requests(parsed_request):
    BASE_URL = 'https://api.instagram.com/v1/users/'
    user_id = parsed_request['data'][0]['id']

    initial_request = '%s%s/media/recent?access_token=%s' %(BASE_URL, user_id, access_token)
    make_request(initial_request, gallery_name)

'''
Python requests
'''
def make_request(request_url, gallery_name):
    raw_request = requests.get(request_url)
    parsed_request = json.loads(raw_request.text)

    if gallery_name == None:
        start_instagram_requests(parsed_request)
    else:
        handle_response(parsed_request, gallery_name)

def handle_response(parsed_request, gallery_name):
    for item in range(len(parsed_request['data'])):
```

```
        content = parsed_request['data'][item]
        image_key = content['created_time']

        imagePath = content['images']['thumbnail']['url']
        print image_key

        if imagePath:
            store_images_locally(gallery_name, image_key, imagePath)

    if parsed_request['pagination'] and parsed_request['pagination']['next_url']:
        request_url = parsed_request['pagination']['next_url']
        make_request(request_url, gallery_name) # get next page of content
    else:
        zip_folder(gallery_name) # generate zip folder

def store_images_locally(directory, filename, imagePath):
    if not os.path.exists(directory):
        os.makedirs(directory)

    f = open(directory + '/' + filename + '.png', 'wb')
    f.write(urllib2.urlopen(imagePath).read())
    f.close()

'''
Functions to create zip folder locally
Folder/Zip name are created by gallery_name
'''
def zipdir(path, zip):
    for root, dirs, files in os.walk(path):
        for file in files:
            zip.write(os.path.join(root, file))

def zip_folder(directory):
    zipf = zipfile.ZipFile(directory + '.zip', 'w')
    zipdir(directory, zipf)
    zipf.close()

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

- -

main.py

```
from __future__ import division
try:
   import cPickle as pickle
except:
   import pickle
import sys
import os
import random
from PIL import Image
import cv2
import numpy as np
import tile as T
import base as B
import similarity as S

# Toggle between 1.0 and 0.0 for linear sum ratio of best match
# 1.0 is all color, 0.0 is pure grayscale
ALPHA = 1
DOM_ON = 0
```

```
def entitle(impath, path, format):
    start = len(path)+1
    end = len(format)*-1
    title = impath[start:end]
    return title

def main():
    # Check if user has provided a base image and tile library
    if len(sys.argv) < 2:
        sys.exit("Usage: python main.py base-image-path tile-directory-path tile-format\n \
                  Note: base-image-path should not end in / \n \
                  Example: python main.py Low.jpg _db/justinablakeney .png")

    # Parse command line args
    base_path = sys.argv[1]
    tile_path = sys.argv[2]
    format = sys.argv[3]

    # Pickle path names
    base_ppath = base_path[:-4]+".p"
    tile_ppath = tile_path+".p"
    dom_ppath = tile_path+"_dom.p"
    # Warning: if experimenting with changing constants other than ALPHA, better
    # to delete these pickle files and try again
    # Saving history doesn't make sense as alpha values often shift
    # history_ppath = tile_path+"-"+os.path.basename(base_path)[:-4]+".p"

    # Read tile library images first
    print "Analyzing tile library images..."

    # Check if pickle file exists
    if os.path.exists(tile_ppath) and os.path.exists(dom_ppath):
        base_pickle = open(tile_ppath, "rb")
        tiles = pickle.load( base_pickle )
        base_pickle.close()
        dom_pickle = open(dom_ppath, "rb")
        dominants = pickle.load( dom_pickle)
        dom_pickle.close()
        print "Reloaded pickled file."
    else:
        if os.path.exists(tile_path):
            imfilelist=[os.path.join(tile_path,f) for f in os.listdir(tile_path) if f.endswith(format)]
            if len(imfilelist) < 1: # number of tile images
                sys.exit ("Need to specify a path containing " + format + " files")
            tiles = {} # init dictionary of tile objects
            dominants = {} # init dictionary of list of tiles by dominant color
            num_images = len(imfilelist)
            if num_images > 500:
                num_images = 500 # only look up first 500 images
            print "IMAGES"
            for i in xrange(num_images):
                impath = imfilelist[i]
                # print(impath)
                if ( (i+1)%50 == 0 or i == num_images-1 ):
                    print i+1, "out of", num_images, "images"
                imtitle = entitle(impath, tile_path, format)
                tile = T.Tile(impath, imtitle)
                tiles[imtitle] = tile
                # 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]
            # Save tiles in pickle file for future use
            pickle.dump( tiles, open( tile_ppath, "wb" ) )
            pickle.dump( dominants, open( dom_ppath, "wb" ) )
        else:
            sys.exit(tile_path + " does not exist")

    # Next, analyze base image
    print ""
    print "Analyzing base image..."
    # Check if pickle file exists first
    if os.path.exists(base_ppath):
        base_pickle = open(base_ppath, "rb")
        base = pickle.load( base_pickle )
        base_pickle.close()
        print "Reloaded pickled file."
    elif os.path.exists(base_path):
        base = B.Base(base_path)
        pickle.dump( base, open( base_ppath, "wb" ) )
    else:
            sys.exit(base_path + " does not exist")

    # Find best tiles to compose base image
    print ""
    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
                # print closest_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)

    # 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 COLOR MOSAIC will be done soon."
        mosaic = Image.new('RGBA', (base.cols*size[0], base.rows*size[1]))
    rowcount = 0
    # print "row: " + str(rowcount)
    for row in xrange(base.rows):
        colcount = 0
    # print "column: " + str(colcount)
        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()
```

\- -

reduction.py

```
from __future__ import division
import os, sys, time
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

# resize image by percentage
def resize_by_p(image, percent):
      w, h = get_dimensions(image)
      desired_w = round( (percent/100) * w )
      image = resize_by_w(image, desired_w)
      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

# 270: rotate image 90 degrees counterclockwise
def rotate(image, degrees):
      (h, w) = image.shape[:2]
      center = (w / 2, h / 2) # find center
      M = cv2.getRotationMatrix2D(center, degrees, 1.0)
      image = cv2.warpAffine(image, M, (w, h))
      return image

# convert color image to grayscale
def grayscale(image):
      image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
      return image

# convert grayscale image to color (to permit color drawing)
def colorize(image):
      image = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)
      return image

# invert image colors
def invert(image):
      image = (255-image)
      return image
```

```
# find otsu's threshold value with median blurring to make image black and white
def binarize(image):
      blur = cv2.medianBlur(image, 5)
      # better for spotty noise than cv2.GaussianBlur(image,(5,5),0)
      ret,thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)
      image = thresh
      return image

# apply morphological closing to close holes - removed from main as it closes gaps
def close(image):
      kernel = np.ones((5,5), np.uint8)
      image = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
      return image

# canny edge detection: performs gaussian filter, intensity gradient, non-max
# suppression, hysteresis thresholding all at once
def edge_finder(image):
      image = cv2.Canny(image,100,200) # params: min, max vals
      return image

# Replace all pixels with empty tiles
def blank(image):
      w,h = get_dimensions(image)
      for i in xrange(h):
        for j in xrange(w):
                image[i][j] = [255, 255, 255, 0]
      save(image, './0.png')

def save(image, name):
      cv2.imwrite(name, image)

# 0 means forever
def show(image, millisec):
      cv2.waitKey(millisec)
      cv2.imshow('Image', 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)

def fill(img, title):
      """ Fill transparencies with dominant color """
      img = img.convert("RGBA")
      print "title: " + str(title)
      poll = img.getcolors() #most frequently used colors
      print "poll: " + str(poll)
      max = 0
      dom = None
      for i in range(len(poll)):
        colors = poll[i]
        print "compare: " + str(colors)
        if colors[1][3] != 255: #transparent
                continue
        if colors[1][0] == 49 and colors[1][1] == 49 and colors[1][2] == 49:
                continue
        if colors[0] > max:
                max = colors[0]
                dom = colors[1]
      color = dom #dominant color
      print "color: " + str(color)
      pixels = img.load()
      fill = Image.new('RGB', (15, 15))
      for y in range(img.size[1]):
        for x in range(img.size[0]):
                r, g, b, a = pixels[x, y]
        if a != 255: #transparent
                r = color[0]
                g = color[1]
                b = color[2]
                pixels[x,y] = (r,g,b,a)
                fill.putpixel((x, y), pixels[x,y])
      fill.save(str(title) + "-rgb.png") #save the image
      folder = os.path.dirname(os.path.realpath(__file__)) #now load it back in
      files = os.listdir(folder)
      for stuff in files:
        if not os.path.isdir(stuff):
                if stuff == str(title)+"-rgb.png":
                        path = str(title)+"-rgb.png"
                        img = Image.open(path)
                        return img

# ============================================================
# Main Method
# ============================================================

def main():
      if len(sys.argv) < 2:
        sys.exit("Need to specify a path from which to read images")

      imageformat=".png"
      path = "./" + sys.argv[1]

      make_blank = True

      # load image sequence
      if os.path.exists(path):
        imfilelist=[os.path.join(path,f) for f in os.listdir(path) if f.endswith(imageformat)]
```

```
        if len(imfilelist) < 1:
                sys.exit ("Need to specify a path containing .png files")
        for el in imfilelist:
                sys.stdout.write(el)
                image = cv2.imread(el, cv2.IMREAD_UNCHANGED) # load original
                # if make_blank is True:
                #     blank(image)
                #     make_blank = False
                # test square crop
                image = resize_by_w(image, 200)
                image = crop_square(image)
                show(image, 1000)
                save(image, el[:-4]+"_square"+".png")
      else:
        sys.exit("The path name does not exist")

      time.sleep(5)

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

- -

similarity.py

```
import os
import sys
import cv2
import numpy as np
from PIL import Image
from matplotlib import pyplot as plt
from matplotlib import gridspec as gridspec
from sklearn.cluster import KMeans
import operator

# ============================================================
# Constants
# ============================================================

COL_RANGE = 256
BINS = 4
BIN_SIZE = int(COL_RANGE/BINS)
gBINS = BINS*BINS # need more bins for grayscale since there's only one axis
gBIN_SIZE = int(COL_RANGE/gBINS)
DOM_COL_THRESH = 0.3

# ============================================================
# Analysis
# ============================================================

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

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

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
    # print 'diff, sum and distance:', diff, sum, distance
    return l1_norm

def l1_gray_norm(h1, h2):
    diff = 0
    total = 0
    #print h1
    #print h2
    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

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)
            # print p,
            if p > DOM_COL_THRESH:
                dominant_colors.append( (r,g,b) )
            else:
                # print 'Dominant colors:', dominant_colors
                return dominant_colors # don't care about the rest
    return dominant_colors # in case

def kmeans_dominance(image):
    # reshape the image to be a list of pixels
    image = image.reshape((image.shape[0] * image.shape[1], 3))
    # cluster the pixel intensities
    clt = KMeans(n_clusters = 3)
    clt.fit(image)
    # grab the number of different clusters and create a histogram
    # based on the number of pixels assigned to each cluster
    numLabels = np.arange(0, len(np.unique(clt.labels_)) + 1)
    (hist, _) = np.histogram(clt.labels_, bins = numLabels)

    # normalize the histogram, such that it sums to one
    hist = hist.astype("float")
    hist /= hist.sum()
    centroids = clt.cluster_centers_
    colors = []
    for (percent, color) in zip(hist, centroids):
        # print percent
        # print color.astype("uint8").tolist()
        color = color.astype("uint8").tolist()
        color = [c/BIN_SIZE for c in color]
        # print color
        colors.append(tuple(color))
    # print colors
    return colors

# ============================================================
# Visualization
# ============================================================

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)

def plot_histogram(hist, title, colors=None, image=None):
    """
    Visualize histograms as bar graphs where each bar is color-coded
```

```
    and sorted by greatest count to least count
    """
    # If information not given, deduce list of colors
    if (colors == None):
        colors = []
        for r in xrange(BINS):
            for g in xrange(BINS):
                for b in xrange(BINS):
                    colors.append( (r,g,b) )
        colors = sorted(colors, key=lambda c: -hist[(c[0])][(c[1])][(c[2])])
        # Remove bins from sorted list of colors if their count is 0 in the histogram
        for i in xrange(len(colors)):
            c = colors[i]
            if hist[(c[0])][(c[1])][(c[2])] == 0:
                colors = colors[:i]
                break
    # Generate bar graph
    plt.rcParams['font.family']='Aller Light'
    for idx, c in enumerate(colors):
        r, g, b = c
        # print c, ':', hist[r][g][b], ';',
        plt.subplot(1,2,1).bar(idx, hist[r][g][b], color=hexencode(c, BIN_SIZE), edgecolor=hexencode(c,
BIN_SIZE))
        plt.xticks([])
        # Optional, append image on the right
        if image != None:
            plt.subplot(1,2,2),plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
            plt.xticks([]),plt.yticks([])
    # Save plot
    dir_name = './color_hist/'
    if not os.path.exists(dir_name):
        os.makedirs(dir_name)
    path = dir_name+title+'.png'
    plt.savefig(path, bbox_inches='tight')
    plt.clf()
    plt.close('all')
    print path
    return path

# ============================================================
# Intra-set Analysis
# ============================================================

def calc_cdistance(chists):
    """
    Find distance between every image pair in the set by calculating
    L1 norm
    """
    chist_dis = {}
    for i in xrange(NUM_IM):
        for j in xrange(i, NUM_IM):
            if (i,j) not in chist_dis:
                d = l1_color_norm(chists[i], chists[j])
                chist_dis[(i,j)] = d
    # print chist_dis
    return chist_dis

def color_matches(k, chist_dis):
    """
    Find images most like and unlike an image based on color distribution.
    k -- the original image for comparison
    chists -- the list of color histograms for analysis
    """
```

```
    results = {}
    indices = []
    distances = []

    for i in xrange(0, NUM_IM):
        if k > i: # because tuples always begin with lower index
            results[i] = chist_dis[(i,k)]
        else:
            results[i] = chist_dis[(k,i)]
    # Ordered list of tuples (dist, idx) from most to least similar
    # -- first value will be the original image with diff of 0
    results = sorted([(v, k) for (k, v) in results.items()])
    # print 'results for image', k, results
    seven = results[:4]
    seven.extend(results[-3:])
    # print 'last seven for image', k, seven
    distances, indices = zip(*seven)
    # print 'distances:',distances
    # print 'indices:',indices
    return indices, distances

def find_four(chist_dis):
    results = {}
    # ensure that a<b, b<c and c<d as order does not matter
    for a in xrange(NUM_IM):
        for b in xrange(a+1,NUM_IM):
            for c in xrange(b+1,NUM_IM):
                for d in xrange(c+1,NUM_IM):
                    results[(a,b,c,d)] = \
                    chist_dis[(a,b)] + chist_dis[(a,c)] + \
                    chist_dis[(a,d)] + chist_dis[(b,c)] + \
                    chist_dis[(b,d)] + chist_dis[(c,d)]
    results = sorted([(v, k) for (k, v) in results.items()])
    best = results[0]
    worst = results[-1]
    indices = list(best[1])
    indices.extend(list(worst[1]))
    # print "results: ", len(results), #results
    # print "best, worst", best, worst
    return indices

# ============================================================
# Intra-Set Visualization
# ============================================================

def septuple_stitch_h(images, titles, dir_name, cresults, cdistances, cvt):
    plt.rcParams['font.family']='Aller Light'
    gs1 = gridspec.GridSpec(1,7)
    gs1.update(wspace=0.05, hspace=0.05) # set the spacing between axes.
    for k in xrange(0, NUM_IM*7, 7):
        for i in xrange(7):
            idx = cresults[k+i]
            ax = plt.subplot(gs1[i])
            plt.axis('on')
            if cvt is 0:
                plt.imshow(images[idx], cmap="Greys_r")
            elif cvt is -1:
                plt.imshow(images[idx], cmap="binary")
            else:
                plt.imshow(cv2.cvtColor(images[idx], cv2.COLOR_BGR2RGB)) # row, col
            plt.xticks([]),plt.yticks([])
            if cdistances:
                if i == 0:
```

```
                    plt.xlabel('similarity:')
                else:
                    sim = 1 - round(cdistances[k+i], 5)
                    plt.xlabel(sim)
                plt.title(titles[idx], size=12)
            ax.set_aspect('equal')

        title = titles[k/7]
        if not os.path.exists(dir_name):
            os.makedirs(dir_name)
        title = dir_name+title+'.png'
        plt.savefig(title, bbox_inches='tight')
        print title
        plt.clf()
        plt.close('all')

def four_stitch_h(images, titles, cresults, dir_name):
    plt.rcParams['font.family']='Aller Light'
    gs1 = gridspec.GridSpec(1,4)
    gs1.update(wspace=0.05, hspace=0.05) # set the spacing between axes.
    for k in xrange(0,8,4):
        for i in xrange(4):
            idx = cresults[i+k]
            ax = plt.subplot(gs1[i])
            plt.axis('on')
            plt.imshow(cv2.cvtColor(images[idx], cv2.COLOR_BGR2RGB)) # row, col
            plt.xticks([]),plt.yticks([])
            plt.title(titles[idx], size=12)
            ax.set_aspect('equal')
        if k is 0:
            title = 'best_match.png'
        else:
            title = 'worst_match.png'
        if not os.path.exists(dir_name):
            os.makedirs(dir_name)
        plt.savefig(dir_name+title, bbox_inches='tight')
        print title
        plt.clf()
        plt.close('all')

# ============================================================
# Testing
# ============================================================

def main():

    # Check if user has provided a directory argument
    if len(sys.argv) < 2:
        sys.exit("Need to specify a path from which to read images")

    format = ".png"
    if (sys.argv[1] == "./"):
        path = sys.argv[1]
    else:
        path = "./" + sys.argv[1]

    global NUM_IM
    images = []
    titles = []
    chists = []
    chist_images = []
    cresults = []
    cdistances = []
```

```
    # Process images from user-provided directory
    if os.path.exists(path):
        imfilelist=[os.path.join(path,f) for f in os.listdir(path) if f.endswith(format)]
        if len(imfilelist) < 1:
            sys.exit ("Need to specify a path containing .ppm files")
        NUM_IM = len(imfilelist) # default is 40
        print "Number of images:", NUM_IM
        for el in imfilelist:
            print(el)
            # Update images and titles list
            image = cv2.imread(el, cv2.IMREAD_UNCHANGED)
            start = len(path)+1
            end = len(format)*-1
            title = el[start:end]
            # Generate color histogram
            chist, image, plot = color_histogram(image, title)
            print chist
            chist_images.append(plot)
            # chist = color_histogram(image, title)
            chists.append(chist)
            images.append(image)
            titles.append(title)
    else:
        sys.exit("The path name does not exist")

    # Calculate lookup table for distances based on color histograms
    chist_dis = calc_cdistance(chists)

    # Determine 3 closest and 3 farthest matches for all images
    for k in xrange(NUM_IM):
        # By color
        results, distances = color_matches(k, chist_dis)
        cresults.extend(results)
        cdistances.extend(distances)

    # Visualize septuples of best and worst matches by image and histogram visualizations
    septuple_stitch_h(images, titles, './color_sim/', cresults, cdistances, 1)
    septuple_stitch_h(chist_images, titles, './color_hist_sim/', cresults, None, 1)

    # Find set of 4 most different and 4 most similar images
    # cfour = find_four(chist_dis)


    # Display four best and four worst, by color and by texture
    # four_stitch_h(images, titles, cfour, path)

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

- -

tile.py

```
import cv2
from PIL import Image
from dominance import colorz

import reduction as R
import similarity as S

TILE_WIDTH = 50
DISPLAY_WIDTH = 100

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) )
  # Crop image to square (simply set size for last param if you want 30x30 tiles)
  # (width, height) = self.display.size
  # if (width < height):
  #       self.display = R.resize_square(self.display, (width, width) )
  # elif (height < width):
  #       self.display = R.resize_square(self.display, (height, height) )
  # self.display = R.fill(self.display, self.title)

  """Additional options (extra runtime)"""
  # Optional: generate bar chart to visualize histogram
  # plot_path = S.plot_histogram(self.histogram, self.title, self.colors)

  # Optional: find dominant colors
  # print 'Title: ', title,
  self.dominants = S.dominant_colors(self.histogram, self.colors)
  # self.dominants = S.kmeans_dominance(self.image)
  # self.dominants = colorz(self.image)
  # print self.dominants
```