Assignment 1 - Photomosaic

Unknown macro: {center}

Assignment 1 - Photomosaic

by Bogo Giertler

Introduction

The objective of the Photomosaic assignment was to acquaint oneself with the elements that will be used throughout the course - Python programming language, Numpy math library and OpenCV computer vision library. While Dr. Eastwood recommends using Eclipse as editor for the Python code, I found myself much more efficient working in TextMate. A matter of personal preference, I guess.
In this assignment an application that can take in a graphics file, a list of tiles and create a contact sheet of tiles and a mosaic composed of the tiles was created.

Approach

My approach to the code and program has been modified many times during the process of writing it. I started out from a simple application that was able to read in images, convert them to grayscale and spit them out, and I finished with almost-half-a-thousand-lines-of-code monster, that can run from command line and do many nifty tricks according to users wishes.

The ability to create a list of thumbnails was expanded into more customizable lists, possibly with color. Now one can easily read his or her vacation photos and with the right setting, get a nice contact sheet of their photos.

The ability to create a mosaic was heavily worked on. From a simple select'n'go, the app transitioned into using a number of bins to store images of appropriate intensity. To fight against holes in the histogram of the picture (i.e. not all of the pictures will be suitable to replace any pixel), the app also creates a mapping of colors into the nearest colors (i.e. shades of gray). Together, when run it allows to quickly compare the value of intensity to a bin and select the item from the bin or intensity list (for larger difference) quickly. These all features put together allow one to create a grayscale, 5000x5000 mosaic out of 576 tiles in just 40 seconds on a not-so-fast MacBook Pro 2.6 GHz. I would expect that the complexity in most places will be linear and just few places will be approaching n^2 in the worst case.

I considered similar approach for color images (i.e. having separate bins for red, green and blue channels), from later the lists of closest channel values would be created and intersected (think of the RGB circle representation and white being the place where we would find fitting images), however implementing unions and intersections was simply too much work for an effect that would probably be roughly similar to what the system can do so far with the color images. Creating a color image takes around 60-70 seconds on my MacBook Pro and most probably is of n^2 complexity, as all pixels have to be checked against all tiles.

App allows user to customize most of its setting - from selecting the tile dictionary, modifying the dimensions of tiles and changing the number of tiles in contact sheets, to defining whether color or black and white are going to be used.

Upon starting up the app with a --help switch, the following output will appear:

```
dhcp-10-7-Assignment 1 Bogo$ /usr/bin/python tiles.py --help
Usage: tiles.py [options] filename
filename serves as output name for contact and input/output for mosaic

Options:
  -h, --help            show this help message and exit
  -c COLOR, --color=COLOR
                        a yes/no switch for use of color; no by default
  -t TYPE, --type=TYPE  a contact/mosaic switch; contact by default
  -s SIDE, --side=SIDE  tile's side length; 50 by default
  -l LINE, --line=LINE  images per line in the contact sheet; 24 by default
  -d DIR, --dir=DIR     directory containing tiles; /tiles by defaut
```

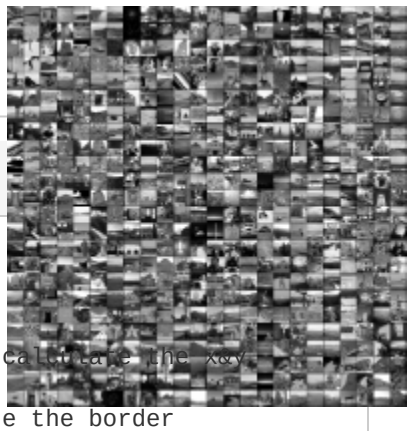
Application also counts the percentage of calculations done, providing live feedback to the user, to reassure him or her, that the app did not yet hang. What I think makes the application easy to use and fairly self explanatory 🙂

Code and results

Contact Sheets

Original contact sheet

This contact sheet is characterized by the images being put based on the order in which they were read. The code is fairly simple and easy to understand. This version also did not put the border on top and left of the contact sheet.

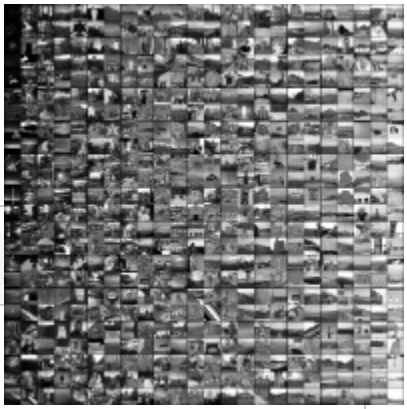


Original contact sheet creation

```
for index, image in enumerate(imageBank):
 insertImage(contactSheetNum,      # our contact sheet
   image[1],       # we directly pick at CV array
   (index*(squareSide+1))%(photosPerLine*(squareSide+1)), # we calculate the x&y
origins
   (index/photosPerLine)*(squareSide+1))   # +1 serves to create the border
```

Improved contact sheet

As the general imageBank code progressed towards using intensity bins and maps for data storage, a new method of creating contact sheets was necessary. The new method simply iterates over bins and picks up all of their contents to feed them to the inserting function. It is a little bit more complicated (and requires an additional iterator to keep count of all the loops that we went through - if there is a better solution for it, please let me know), but the useful side effect is that all images are sorted by their intensities in the contact sheet, which gives a nice feel of gradient.



Improved contact sheet loop

```
imageNumber = 0
for intensity in imageBank:
 print "Intensity level found"
 for image in intensity:
  insertImage(contactSheetNum,      # our contact sheet
   image,         # we directly pick at CV array
   (imageNumber*(squareSide+1))%(photosPerLine*(squareSide+1))+1, # we calculate the
x&y origins
   (imageNumber/photosPerLine)*(squareSide+1)+1)   # +1s serves to create the border
  iterator = iterator+1
  print "Image added, iterator at " + str(imageNumber)
```

Color contact sheet

As I progressed through assignment and completed the color mosaic section, I thought it would be a nice touch to add a color contact sheet to the options of the app - especially since I was streamlining the imageBank tuple, so use of a color property is the best possibility. The effect was simple, yet effective - just check it against the original contact sheet.

Mosaic

Basic black and white mosaic

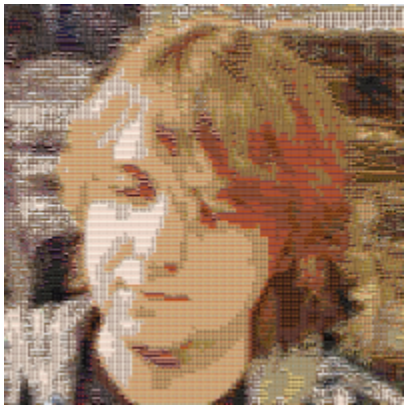
This most basic mosaic was created simply by pre calculating the intensities and storing them together with the image. Afterwards, the intensities were selected by a linear search. However, some intensities did not exist due to the lack of appropriate images. In these cases, the intensity searched was being successively reduced by one in each recursive call, until the closest one to the bottom was was found. This image has a significant problem of having really long streaks of ugly shades of gray shades and repetitive images in large zones of the picture.

Advanced black and white mosaic

The more advanced version of the mosaic was created using the aforementioned concept of intensity bins. During the creation of library, a list of 256 drawers/bins is created and during the analysis of imported pictures, each picture is assigned to one of the discrete bins. Later, the map is created containing the number of the closest bin at each of the discrete intensities to solve the problem of unmatched intensities. Based on these two inputs, closest ten bins in each direction is copied into the additional bin and later a random element is chosen. By that, the tiles appear more randomized, but yet still fitting the general picture. No more are long streaks of same tiles, except for the area of my cheek.

Color mosaic

Color mosaic is created based on the Euclidean distances between average color values of each channel in the picture's pixel and tile. While a perfect reassembling of the picture is nearly impossible, a fairly good simile is created, where the general shape and features are kept. Creating such mosaic is the most time consuming process in this app, since the mosaic is created by comparing each pixel's RGB values to mean RGB values of other pictures in the Euclidean distance, requiring basically n*m*r comparisons (for n, m dimensions and r pictures in the database) in the process - for the 100x100 image with 50x50 tiles per each pixel, creating the image may take up to 90 seconds (including databank analysis)



Apple and iPhone mosaics

As a part of the creative mosaic extension, two mosaics were created. As I am a huge Apple fan (not a fanboy though) - and icons are freely available out there in large quantities - I decided to portray myself in two ways - one, as a mosaic composed of grayscale icons of popular consumer IT products (mostly Apple, but also Canon, Sony, etc). The Chris Jordan style of description would be "We

are what we use". The icons were obtained from interfaceLift and the set used is CHUMS by Susumu Yoshida.
The second mosaic is composed out of icons of apps in my iPhone (not all of them though). Since I could not be bothered to copy all of them hand by hand, I just downloaded a set of Suave icons, which I used before on my jailbroken iPhone and really liked. It could be called a "Developer's self-portrait".
It was a lot of fun to do these two actually. Each of them is linked here to a 2500x2500 version of themselves in which you can actually see the icons fairly accurately.

Extensions

Through implementing intensity bins, I managed to implement the fast selection which was expanded into a very fast, efficient and beautiful code (4). Another banks of images were created (ex. 1) and interesting mosaics were created (ex. 2). By modifying a tile size and providing input image big enough, one can obtain JPG file suitable for printing (ex. 6)

Conclusions

The project was a lot of fun. While a lot of stuff was left unsaid by Dr. Esstwood, it just brought even more fun to the whole creation. The assignment was creative and the effect was a lot of fun too. I especially liked using the customized system to get virtually any sort of mosaic out of image/icon collections I had. In the end I learnt a lot of basic OpenCV functionality by heart and got a great grasp on Python again - programming robots for the whole term in C put me again in some bad mental concepts that I fortunately fought off with some nice Python philosophy. All in all? I liked it!