Yahoo! Developer Network Blog

« Previous | Main | Next »

SEPTEMBER 17, 2008

Creating photo mosaics with Yahoo! BOSS image search

The past few days of Open Hack Day 2008 were awesome, one of the most fun developer events I've been to in a long time. In the course of going to Sunnyvale, I was playing with BOSS (Build your Own Search Service) and it tickled me that using BOSS to search for images was actually easier than using the Flickr APIs themselves. Of course, there are more things you can do with Flickr APIs but if you want to just dig out lots of photos, nothing beats BOSS image search. I was mulling on a good hack to write (Yahoos are not eligible to participate in the competition but I was interested in just the hacking) when it occurred to me that photo mosaics require a large number of photos, and this need is perfectly served by the BOSS image search.

Photo mosaics are pretty and fascinating. For the uninitiated, a photo mosaic is a picture montage made with a large number of photos. An original photo is divided into a number of smaller rectangular parts, each of which is replaced by a photo (called a photo tile) that is about the same average color. When viewed from afar, the original photo stands out but when magnified appropriately, each photo tile shows up as individual photos. Photo mosaics have been popular for a while.

In this article I will describe how we can use Ruby to create a simple photo mosaic building program that stitches together a mosaic from any number of photos. While this article is primarily for developers, anyone can just use the program to create fun photo mosaics.

Sau Sheong Chang
Engineering Director for South-East Asia

Normally, one of the issues faced when creating a photo mosaic is getting the photo tiles. While a good sized photo album would contain thousands or tens of thousands of photos, if you want to create one that is based on a particular theme, you might be hard-pressed to find a source for a large number of photos. In comes **Yahoo!'s BOSS image search APIs (http://developer.yahoo.com/search/boss)**. BOSS (Build your Own Search Service) is Yahoo!'s open search web services platform that allows any developer to tap on Yahoo!'s search engine data through REST-like API calls. BOSS provides 3 types of searches -- web, image, and news -- but we will only use the image search in our application. To get started with BOSS you need to go to the developer page, and register for an application id.

Sometimes you end up with a smaller pool of photo tiles. It could be that you chose a rather unique search term for your theme. You might also want to chose a smaller set of photo tiles since processing a large number of photos could be time consuming. Whatever the reason, if you don't have enough photos you might not be able to finish the master image, so you might want to use a tile more than once. However if the photo tile pool is not big enough, the final product might look a bit off-color. In this article, for simplicity I will use photo tiles more than once but you can easily change the mosaic to have unique photo tiles.

I will also use **RMagick**, the Ruby library that interfaces with **ImageMagick**, the very useful and comprehensive image manipulation command-line program. ImageMagick is cross-platform and has a GPL-like licence that allows its to be freely distributable.

Let's get cracking. The algorithm for this program is quite straightforward:
1. Take a photo, called the master photo, and reduce it into a small and manageable size.
2. Next we use BOSS to search for a particular theme, which returns URLs of all photos it can find on that theme. These photos are our photo tiles
3. Then for each photo tile, we get the average color of the tile
4. For each pixel in the reduced master photo, we compare its color with any of the photo tiles' average color and look for the closest match
5. We pick the closest match and use that photo tile in place of the pixel in a new image and repeat for each pixel in the master photo

Before we start the main flow of the algorithm, let me explain 2 functions that we will need to do. First we need to find the average color of an image. This is quite simple. The RGB color model is an additive color model in which red, green, and blue light (hence RGB) are added together in various ways to produce

SUBSCRIBE

YDN Blog: Get Yahoo! Developer Network Blog on your personalized My Yahoo! home page.

MY YAHOO!

YDN Link Blog: Get Yahoo! Developer Network Linkblog on your personalized My Yahoo! home page.

MY YAHOO!

RECENT BLOG ARTICLES view all

Back off man, I'm a scientist: Science Hackday in London
Mon, 21 Jun 2010

InfoChimps Query API in public beta
Sun, 20 Jun 2010

Yahoo! Labs: Modeling networks of social behavior
Fri, 18 Jun 2010

Semantic Search: An Interview with Peter Mika, Yahoo! Research
Fri, 18 Jun 2010

Announcing Yahoo!'s X Award for the PayPal X Developer Challenge 2010
Thu, 17 Jun 2010

RECENT LINKS

Everything you need to know about the internet | Technology | The Observer
Mon, 21 Jun 2010

Tech Thursday - Vuvuzuela, Video editing, fighting code bloat and know your hex colours (Yahoo! Developer Network Blog)
Thu, 17 Jun 2010

The Yahoo! Merchant Summit: Back to the basics of online marketing (Yahoo! Developer Network Blog)
Wed, 16 Jun 2010

How a Silly Phone for Teens Reveals Microsoft's Plan for Us All
Sun, 06 Jun 2010

What is data science? - O'Reilly Radar
Sat, 05 Jun 2010

ARCHIVES

2010

- June (27)
- May (32)
- April (42)
- March (38)
- February (29)
- January (19)

different colors. In our program, color is described in the RGB model. What we need to determine is the RGB representation of the average color. To find that average color, we examine the color of each pixel in that image, add up all the reds and divide it by the total number of pixels to find the red of that average color and do the same for green and blue.

```
def average_color(image)
  red, green, blue = 0, 0, 0
  image.each_pixel { |pixel, c, r|
    red += pixel.red
    green += pixel.green
    blue += pixel.blue
  }
  num_pixels = image.bounding_box.width * image.bounding_box.height
  return {:red => red/num_pixels, :green => green/num_pixels, :blue =>
end
```

Then we need to find the difference between one color and another. Without going into the black hole of color difference (also called delta E), we determine the difference or distance between two colors by calculating the (Euclidean) distance between two points. In the case of a color, we treat it as a 3-dimensional coordinate but instead of x, y, an z we use R, G, and B.

Translating this to code:

```
def color_difference(rgb1, rgb2)
  red, green, blue = rgb1[:red] - rgb2[:red], rgb1[:green] - rgb2[:gree
  Math.sqrt((red * red) + (green * green) + (blue * blue))
end
```

This is a grossly crude way of determining the difference between one color and another and the colors that are matched eventually are a bit off. A closer approximation can be acquired by applying different values of constants to the red, green, and blue variables.

Now that we have both these essential functions, we create a third convenience function to compare a pixel with the average colors of a bunch of photos:

```
def match_photo(pixel,average_colors)
  ave = average_colors.collect { |color|  color_difference(pixel, color
  ave.index(ave.min)
end
```

This code takes in a pixel and an array of average colors, then finds the array index of the smallest average number in the array. Returning this index allows us to find the correct photo.

Suitably armed with the 3 functions, let's move on to the main flow of the program:

```
width = 100
master = ImageList.new('master_photo.jpg')
master.resize_to_fit!(width,(master.bounding_box.height.to_f/master.bou
```

First, load the master photo into an ImageList. Then resize the master photo to a scale factor. We want to reduce the master photo this way to reduce the number of pixels to process. Remember, each pixel in the master photo will be replaced by a photo tile, so you don't want a large master photo. For example, if you have a 640 x 480 photo, you will need to process and replace 307,200 pixels with photos! A fair number is 100 x 75 (which is the same ratio) but only has 7,500 pixels.

Next, create a pixel array of your master photo:

```
pixel_array = []
master.each_pixel {|pixel, c, r|
  pixel_array << {:red => pixel.red, :green => pixel.green, :blue => pi
}
```

We will iterate through this array later on. Next, we start to get the photo tiles through Yahoo! image search:

```
photo_tiles = ImageList.new
population = 1000
step_count = 10
search_query = 'Leonardo Da Vinci'
boss_app_id = YOUR_BOSS_APP_ID
```

```
num = 1
(0..population).step(step_count) { |count|
  url = "http://boss.yahooapis.com/ysearch/images/v1/#{search_query}?ap
  (res = Net::HTTP.get_response(URI.parse(URI.escape(url)))) rescue put
  data = XmlSimple.xml_in(res.body, { 'ForceArray' => false })['results
  data.each {|rec|
    photo_tiles.read(rec['url']) rescue puts 'Cannot read image'
    num = num.next
  }
} rescue 'no more images found'
```

Population is the number of photo tiles to take from the image search. Step count is the number of images requested from BOSS per query. You will also need a **BOSS App ID**.

The code simply sends a GET request to BOSS, get the response and parses it through XML Simple, which in turn converts it into an array of arrays. After that, we iterate through the response array and read in the image into a new ImageList.

The next step is to determine the average color of each photo in the photo tiles ImageList:

```
average_colors = []
num = 1
photo_tiles.each { |img|
  average_colors << average_color(img)
  num = num.next
}
```

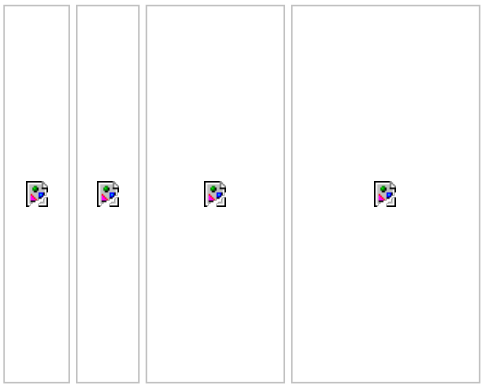
With all the pieces in place, we will calculate the where to put the photo tiles and create a new ImageList to contain our mosaic:

```
tile_size = 40
mosaic_images = ImageList.new
tile = Rectangle.new(tile_size,tile_size,0,0)
photo_tiles.scene = 0
num = 0
master.bounding_box.height.times do |row|
  master.bounding_box.width.times do |col|
    idx = match_photo(pixel_array[num], average_colors)
    mosaic_images << photo_tiles[idx].crop_resized(tile_size,tile_size)
    tile.x = col * mosaic_images.columns
    tile.y = row * mosaic_images.rows
    mosaic_images.page = tile
    (photo_tiles.scene += 1) rescue photo_tiles.scene = 0
    num = num.next
  end
end
```

We need to set a tile size now, and the larger the tile size is, the clearer and more vivid the photo tile is. However this also means the final mosaic file will be bigger and it will take longer to process, so we crop and resize it appropriately. We create a Rectangle with the tile size and this will be our tile. Then for each pixel in the master photo, we get the correct photo using the match_photo function we created earlier, and place it at the correct place in the ImageList.

Finally, we take the mosaic ImageList we have just created and make a mosaic out of it, and write it to a file:

```
mosaic = mosaic_images.mosaic
mosaic.write('mosaic.jpg')
```



This is a sample run on Mona Lisa, with the search term 'Leonardo Da Vinci'. Notice that photo tiles are re-used. To view more photos and mosaics of the photos, please go to this Flickr photostream - **http://www.flickr.com/photos/29714598@N03**

Try it out yourself!

Posted at September 17, 2008 5:53 PM | **Permalink**

Bookmark this on Delicious

COMMENTS

Any live site that implements this?

Posted by: invadereki at September 19, 2008 5:57 PM

Hello.

Where does the source code for this project exist? It sounds fun and I'd like to play with it.

Thanks,
Chris

Posted by: chris at November 24, 2009 8:41 PM

POST A COMMENT

Comment Policy: We encourage comments and look forward to hearing from you. Please note that Yahoo! may, in our sole discretion, remove comments if they are off topic, inappropriate, or otherwise violate our Terms of Service. **Fields marked with asterisk '*' are required.**

Name:*

Email Address:*

URL:

Remember Me? Yes No

Type "yahoo" here:*

Comments:*

Post

YDN LIBRARIES & BEST PRACTICES

ASTRA
ASTRA, the ActionScript Toolkit for Rich Applications, is a collection of Flash and Flex components, code libraries, toolkits and utilties developed by Yahoo! for ActionScript developers.

Design Pattern Library
Interaction solutions that describe an optimal solution to a common problem within a specific context.

Exceptional Performance
Best practices for improving web performance, including research and build tools that center around the rules for high performance web sites.

Yahoo! User Interface Library (YUI)
A set of utilities and controls, written in JavaScript, for building richly interactive web applications using techniques such as DOM scripting, DHTML and AJAX.

YAHOO! APIs & WEB SERVICES

Answers	BrowserPlus™	GeoPlanet™	Maps	My Yahoo!®	RSS Feeds	Traffic	YAP
Address Book	delicious®	Hadoop	Media Player	OAuth	Search	Upcoming	YOS
APT	Finance	HotJobs®	Mobile	OpenID	SearchMonkey®	Utilities	YQL
Authentication	Fire Eagle®	Local	Music	Placemaker™	Shopping	Weather	YWA
BOSS	Flickr®	Mail	MyBlogLog	Pipes	Social	Widgets	

LANGUAGE CENTERS

Flash Java JavaScript .NET PHP Python Ruby Silverlight