pyimagesearch
be awesome at building image search engines

# Hobbits and Histograms – A How-To Guide to Building Your First Image Search Engine in Python

by **Adrian Rosebrock** on January 27, 2014 in **Examples of Image Search Engines**, **Image Descriptors**, **Tutorials**

> *One Ring to rule them all, One ring to find them; One ring to bring them all and in the darkness bind them.*

The image search engine we are about to build is going to be so awesome, it could have destroyed The One Ring itself, without the help of the fires of Mt. Doom.

Okay, I've obviously been watching a lot of *The Hobbit* and the *Lord of the Rings* over this past week.

And I thought to myself, you know what would be awesome?

Building a simple image search engine using screenshots from the movies. And that's exactly what I did.

Here's a quick overview:

- **What we're going to do:** Build an image search engine, from start to finish, using *The Hobbit* and *Lord of the Rings* screenshots.
- **What you'll learn:** The 4 steps required to build an image search engine, with code examples included. From these examples, you'll be able to build image search engines of your own.
- **What you need:** Python, NumPy, and OpenCV. A little knowledge of basic image concepts, such as pixels and histograms, would help, but *is absolutely not required*. This blog post is meant to be a How-To, hands on guide to building an image search engine.

**Looking for the source code to this post?**
**Jump right to the downloads section.**

## Hobbits and Histograms – A How-To Guide to Building Your First Image Search Engine in Python

I've never seen a "How-To" guide on building a simple image search engine before. But that's exactly what this post is. We're going to use (arguably) one of the most basic image descriptors to quantify and describe these screenshots — the color histogram.

I discussed the color histogram in my previous post, a guide to utilizing color histograms for computer vision and image search engines. If you haven't read it, no worries, but I would suggest that you go back and read it after checking out this blog post to further understand color histograms.

But before I dive into the details of building an image search engine, let's check out our dataset of *The Hobbit* and *Lord of the Rings* screenshots:

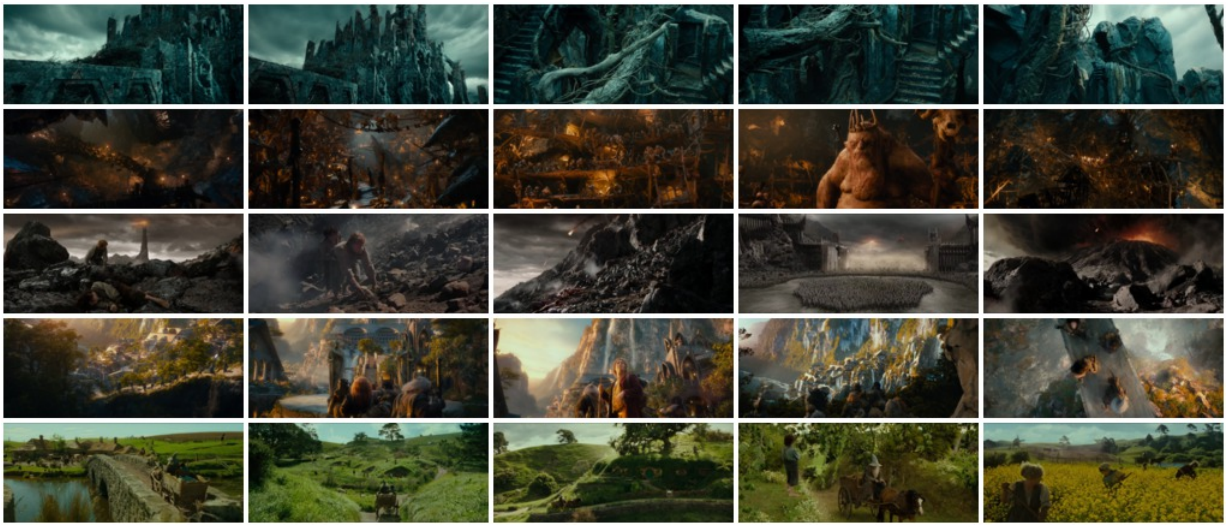

**Figure 1:** Our dataset of *The Hobbit* and *Lord of the Rings* screenshots. We have 25 total images of 5 different categories including Dol Guldur, the Goblin Town, Mordor/The Black Gate, Rivendell, and The Shire.

So as you can see, we have a total of 25 different images in our dataset, five per category. Our categories include:

- **Dol Guldur:** "The dungeons of the Necromancer", Sauron's stronghold in Mirkwood.
- **Goblin Town:** An Orc town in the Misty Mountains, home of The Goblin King.
- **Mordor/The Black Gate:** Sauron's fortress, surrounded by mountain ranges and volcanic plains.
- **Rivendell:** The Elven outpost in Middle-earth.
- **The Shire:** Homeland of the Hobbits.

The images from Dol Guldur, the Goblin Town, and Rivendell are from *The Hobbit: An Unexpected Journey*. Our Shire images are from *The Lord of the Rings: The Fellowship of the Ring*. And finally, our Mordor/Black Gate screenshots from *The Lord of the Rings: The Return of the King*.

## The Goal:

The first thing we are going to do is *index* the 25 images in our dataset. Indexing is the process of quantifying our dataset by using an *image descriptor* to extract *features* from each image and storing the resulting features for later use, such as performing a search.

An *image descriptor* defines how we are quantifying an image, hence extracting features from an image is called *describing an image*. The output of an image descriptor is a *feature vector*, an abstraction of the image itself. Simply put, it is a list of numbers used to represent an image.

Two feature vectors can be compared using a *distance metric*. A distance metric is used to determine how "similar" two images are by examining the distance between the two feature vectors. In the case of an image search engine, we give our script a *query image* and ask it to rank the images in our index based on how relevant they are to our query.

Think about it this way. When you go to Google and type "Lord of the Rings" into the search box, you expect Google to return pages to you that are relevant to Tolkien's books and the movie franchise. Similarly, if we present an image search engine with a query image, we expect it to return images that are relevant to the content of image — hence, we sometimes call image search engines by what they are more commonly known in academic circles as *Content Based Image Retrieval (CBIR)* systems.

So what's the overall goal of our Lord of the Rings image search engine?

**The goal, given a query image from one of our five different categories, is to return the category's corresponding images in the top 10 results.**

That was a mouthful. Let's use an example to make it more clear.

If I submitted a query image of The Shire to our system, I would expect it to give me all 5 Shire images in our dataset back in the first 10 results. And again, if I submitted a query image of Rivendell, I would expect our system to give me all 5 Rivendell images in the first 10 results.

Make sense? Good. Let's talk about the four steps to building our image search engine.

## The 4 Steps to Building an Image Search Engine

On the most basic level, there are four steps to building an image search engine:

1. **Define your descriptor:** What type of descriptor are you going to use? Are you describing color? Texture? Shape?
2. **Index your dataset:** Apply your descriptor to each image in your dataset, extracting a set of features.
3. **Define your similarity metric:** How are you going to define how "similar" two images are? You'll likely be using some sort of distance metric. Common choices include Euclidean, Cityblock (Manhattan), Cosine, and chi-squared to name a few.
4. **Searching:** To perform a search, apply your descriptor to your query image, and then ask your distance metric to rank how similar your images are in your index to your query images. Sort your results via similarity and then examine them.

## Step #1: The Descriptor – A 3D RGB Color Histogram

Our image descriptor is a 3D color histogram in the RGB color space with 8 bins per red, green, and blue channel.

The best way to explain a 3D histogram is to use the conjunctive **AND**. This image descriptor will ask a given image how many pixels have a Red value that falls into bin #1 **AND** a Green value that falls into bin #2 **AND** how many Blue pixels falls into bin #1. This process will be repeated for each combination of bins; however, it will be done in a computationally efficient manner.

When computing a 3D histogram with 8 bins, OpenCV will store the feature vector as an `(8, 8, 8)` array. We'll simply flatten it and reshape it to `(512,)`. Once it's flattened, we can easily compare feature vectors together for similarity.

Ready to see some code? Okay, here we go:

3D RGB Histogram in OpenCV and Python — Python

```
# import the necessary packages
import imutils
import cv2

class RGBHistogram:
    def __init__(self, bins):
        # store the number of bins the histogram will use
        self.bins = bins

    def describe(self, image):
        # compute a 3D histogram in the RGB colorspace,
        # then normalize the histogram so that images
        # with the same content, but either scaled larger
        # or smaller will have (roughly) the same histogram
        hist = cv2.calcHist([image], [0, 1, 2],
            None, self.bins, [0, 256, 0, 256, 0, 256])

        # normalize with OpenCV 2.4
        if imutils.is_cv2():
            hist = cv2.normalize(hist)

        # otherwise normalize with OpenCV 3+
        else:
            hist = cv2.normalize(hist,hist)

        # return out 3D histogram as a flattened array
        return hist.flatten()
```

As you can see, I have defined a `RGBHistogram` class. I tend to like to define my image descriptors as *classes* rather than *functions*. The reason for this is because you rarely ever extract features from a single image alone. You instead extract features from an entire dataset of images. Furthermore, you expect that the features extracted from all images utilize the same parameters — in this case, the number of bins for the histogram. It wouldn't make much sense to extract a histogram using 32 bins from one image and then 128 bins for another image if you intend on comparing them for similarity.

Let's take the code apart and understand what's going on:

- **Lines 6-8:** Here I am defining the constructor for the `RGBHistogram`. The only parameter we need is the number of bins for each channel in the histogram. Again, this is why I prefer using classes

instead of functions for image descriptors — by putting the relevant parameters in the constructor, you ensure that the same parameters are utilized for each image.

- **Line 10:** You guessed it. The describe method is used to "describe" the image and return a feature vector.
- **Lines 15 and 16:** Here we extract the actual 3D RGB Histogram (or actually, BGR since OpenCV stores the image as a NumPy array, but with the channels in reverse order). We assume `self.bins` is a list of three integers, designating the number of bins for each channel.
- **Lines 19-24:** It's important that we normalize the histogram in terms of pixel counts. If we used the raw (integer) pixel counts of an image, then shrunk it by 50% and described it again, we would have two different feature vectors for identical images. In most cases, you want to avoid this scenario. We obtain *scale invariance* by converting the raw integer pixel counts into real-valued percentages. For example, instead of saying bin #1 has 120 pixels in it, we would say bin #1 has 20% of all pixels in it. Again, by using the percentages of pixel counts rather than raw, integer pixel counts, we can assure that two identical images, differing only in size, will have (roughly) identical feature vectors.
- **Line 27:** When computing a 3D histogram, the histogram will be represented as a NumPy array with `(N, N, N)` bins. In order to more easily compute the distance between histograms, we simply flatten this histogram to have a shape of `(N ** 3,)`. **Example:** When we instantiate our RGBHistogram, we will use 8 bins per channel. Without flattening our histogram, the shape would be `(8, 8, 8)`. But by flattening it, the shape becomes `(512,)`.

Now that we have defined our image descriptor, we can move on to the process of indexing our dataset.

## Step #2: Indexing our Dataset

Okay, so we've decided that our image descriptor is a 3D RGB histogram. The next step is to apply our image descriptor to each image in the dataset.

This simply means that we are going to loop over our 25 image dataset, extract a 3D RGB histogram from each image, store the features in a dictionary, and write the dictionary to file.

Yep, that's it.

In reality, you can make indexing as simple or complex as you want. Indexing is a task that is easily made parallel. If we had a four core machine, we could divide the work up between the four cores and speedup the indexing process. But since we only have 25 images, that's pretty silly, especially given how fast it is to compute a histogram.

Let's dive into some code:

Indexing an Image Dataset using Python — Python

```
1  # import the necessary packages
2  from pyimagesearch.rgbhistogram import RGBHistogram
3  from imutils.paths import list_images
4  import argparse
5  import pickle
6  import cv2
7
8  # construct the argument parser and parse the arguments
9  ap = argparse.ArgumentParser()
```

```
ap.add_argument("-d", "--dataset", required = True,
    help = "Path to the directory that contains the images to be indexed")
ap.add_argument("-i", "--index", required = True,
    help = "Path to where the computed index will be stored")
args = vars(ap.parse_args())

# initialize the index dictionary to store our our quantifed
# images, with the 'key' of the dictionary being the image
# filename and the 'value' our computed features
index = {}
```

Alright, the first thing we are going to do is import the packages we need. I've decided to store the `RGBHistogram` class in a module called `pyimagesearch`. I mean, it only makes sense, right? We'll use `cPickle` to dump our index to disk. And we'll use `glob` to get the paths of the images we are going to index.

The `--dataset` argument is the path to where our images are stored on disk and the `--index` option is the path to where we will store our index once it has been computed.

Finally, we'll initialize our `index` — a builtin Python dictionary type. The key for the dictionary will be the image filename. We've made the assumption that all filenames are unique, and in fact, for this dataset, they are. The value for the dictionary will be the computed histogram for the image.

Using a dictionary for this example makes the most sense, especially for explanation purposes. Given a key, the dictionary points to some other object. When we use an image filename as a key and the histogram as the value, we are implying that a given histogram *H* is used to quantify and represent the image with filename *K*.

Again, you can make this process as simple or as complicated as you want. More complex image descriptors make use of term frequency-inverse document frequency weighting (tf-idf) and an inverted index, but we are going to stay clear of that for now. Don't worry though, I'll have *plenty* of blog posts discussing how we can leverage more complicated techniques, but for the time being, let's keep it simple.

Indexing an Image Dataset using Python — Python

```
# initialize our image descriptor -- a 3D RGB histogram with
# 8 bins per channel
desc = RGBHistogram([8, 8, 8])
```

Here we instantiate our `RGBHistogram`. Again, we will be using 8 bins for each, red, green, and blue, channel, respectively.

Indexing an Image Dataset using Python — Python

```
# use list_images to grab the image paths and loop over them
for imagePath in list_images(args["dataset"]):
    # extract our unique image ID (i.e. the filename)
    k = imagePath[imagePath.rfind("/") + 1:]

    # load the image, describe it using our RGB histogram
    # descriptor, and update the index
    image = cv2.imread(imagePath)
    features = desc.describe(image)
    index[k] = features
```

Here is where the actual indexing takes place. Let's break it down:

- **Line 26:** We use `list_images` to grab the image paths and start to loop over our dataset.
- **Line 28:** We extract the "key" for our dictionary. All filenames are unique in this sample dataset, so the filename itself will be enough to serve as the key.
- **Line 32-34:** The image is loaded off disk and we then use our `RGBHistogram` to extract a histogram from the image. The histogram is then stored in the index.

Indexing an Image Dataset using Python — Python

```
36 # we are now done indexing our image -- now we can write our
37 # index to disk
38 f = open(args["index"], "wb")
39 f.write(pickle.dumps(index))
40 f.close()
41
42 # show how many images we indexed
43 print("[INFO] done...indexed {} images".format(len(index)))
```

Now that our index has been computed, we write it to disk so we can use it for searching later on.

To index your image search engine, just enter the following in your terminal (taking note of the command line arguments):

Building an Image Search Engine in Python and OpenCV — Shell

```
1 $ python index.py --dataset images --index index.cpickle
2 [INFO] done...indexed 25 images
```

## Step #3: The Search

We now have our index sitting on disk, ready to be searched.

The problem is, we need some code to perform the actual search. How are we going to compare two feature vectors and how are we going to determine how similar they are?

This question is better addressed first with some code, then I'll break it down.

Building an Image Search Engine in Python and OpenCV — Python

```
 1 # import the necessary packages
 2 import numpy as np
 3
 4 class Searcher:
 5     def __init__(self, index):
 6         # store our index of images
 7         self.index = index
 8
 9     def search(self, queryFeatures):
10         # initialize our dictionary of results
11         results = {}
12
13         # loop over the index
14         for (k, features) in self.index.items():
15             # compute the chi-squared distance between the features
16             # in our index and our query features -- using the
17             # chi-squared distance which is normally used in the
18             # computer vision field to compare histograms
19             d = self.chi2_distance(features, queryFeatures)
```

```
20
21              # now that we have the distance between the two feature
22              # vectors, we can udpate the results dictionary -- the
23              # key is the current image ID in the index and the
24              # value is the distance we just computed, representing
25              # how 'similar' the image in the index is to our query
26              results[k] = d
27
28          # sort our results, so that the smaller distances (i.e. the
29          # more relevant images are at the front of the list)
30          results = sorted([(v, k) for (k, v) in results.items()])
31
32          # return our results
33          return results
34
35      def chi2_distance(self, histA, histB, eps = 1e-10):
36          # compute the chi-squared distance
37          d = 0.5 * np.sum([((a - b) ** 2) / (a + b + eps)
38              for (a, b) in zip(histA, histB)])
39
40          # return the chi-squared distance
41          return d
```

First off, most of this code is just comments. Don't be scared that it's 41 lines. If you haven't already guessed, I like well-commented code. Let's investigate what's going on:

- **Lines 4-7:** The first thing I do is define a `Searcher` class and a constructor with a single parameter — the `index`. This `index` is assumed to be the index dictionary that we wrote to file during the indexing step.
- **Line 11:** We define a dictionary to store our `results`. The key is the image filename (from the index) and the value is how similar the given image is to the query image.
- **Lines 14-26:** Here is the part where the actual searching takes place. We loop over the image filenames and corresponding features in our index. We then use the chi-squared distance to compare our color histograms. The computed distance is then stored in the `results` dictionary, indicating how similar the two images are to each other.
- **Lines 30-33:** The results are sorted in terms of relevancy (the smaller the chi-squared distance, the relevant/similar) and returned.
- **Lines 35-41:** Here we define the chi-squared distance function used to compare the two histograms. In general, the difference between large bins vs. small bins is less important and should be weighted as such. This is exactly what the chi-squared distance does. We provide an `epsilon` dummy value to avoid those pesky "divide by zero" errors. Images will be considered identical if their feature vectors have a chi-squared distance of zero. The larger the distance gets, the less similar they are.

So there you have it, a Python class that can take an index and perform a search. Now it's time to put this searcher to work.

**Note:** For those who are more academically inclined, you might want to check out *The Quadratic-Chi Histogram Distance Family* from the ECCV '10 conference if you are interested in histogram distance metrics.

## Step #4: Performing a Search

Finally. We are closing in on a functioning image search engine.

But we're not quite there yet. We need a little extra code to handle loading the images off disk and performing the search:

Building an Image Search Engine in Python and OpenCV — Python

```python
# import the necessary packages
from pyimagesearch.searcher import Searcher
import numpy as np
import argparse
import os
import pickle
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required = True,
    help = "Path to the directory that contains the images we just indexed")
ap.add_argument("-i", "--index", required = True,
    help = "Path to where we stored our index")
args = vars(ap.parse_args())

# load the index and initialize our searcher
index = pickle.loads(open(args["index"], "rb").read())
searcher = Searcher(index)
```

First things first. Import the packages that we will need. As you can see, I've stored our `Searcher` class in the `pyimagesearch` module. We then define our arguments in the same manner that we did during the indexing step. Finally, we use `cPickle` to load our index off disk and initialize our `Searcher`.

Building an Image Search Engine in Python and OpenCV — Python

```python
# loop over images in the index -- we will use each one as
# a query image
for (query, queryFeatures) in index.items():
    # perform the search using the current query
    results = searcher.search(queryFeatures)

    # load the query image and display it
    path = os.path.join(args["dataset"], query)
    queryImage = cv2.imread(path)
    cv2.imshow("Query", queryImage)
    print("query: {}".format(query))

    # initialize the two montages to display our results --
    # we have a total of 25 images in the index, but let's only
    # display the top 10 results; 5 images per montage, with
    # images that are 400x166 pixels
    montageA = np.zeros((166 * 5, 400, 3), dtype = "uint8")
    montageB = np.zeros((166 * 5, 400, 3), dtype = "uint8")

    # loop over the top ten results
    for j in range(0, 10):
        # grab the result (we are using row-major order) and
        # load the result image
        (score, imageName) = results[j]
        path = os.path.join(args["dataset"], imageName)
        result = cv2.imread(path)
        print("\t{}. {} : {:.3f}".format(j + 1, imageName, score))

        # check to see if the first montage should be used
        if j < 5:
```

```
51              montageA[j * 166:(j + 1) * 166, :] = result
52
53          # otherwise, the second montage should be used
54          else:
55              montageB[(j - 5) * 166:((j - 5) + 1) * 166, :] = result
56
57      # show the results
58      cv2.imshow("Results 1-5", montageA)
59      cv2.imshow("Results 6-10", montageB)
60      cv2.waitKey(0)
```

Most of this code handles displaying the results. The actual “search” is done in a single line (#31). Regardless, let’s examine what’s going on:

- **Line 23:** We are going to treat each image in our index as a query and see what results we get back. Normally, queries are *external* and not part of the dataset, but before we get to that, let’s just perform some example searches.
- **Line 25:** Here is where the actual search takes place. We treat the current image as our query and perform the search.
- **Lines 28-31:** Load and display our query image.
- **Lines 37-55:** In order to display the top 10 results, I have decided to use two montage images. The first montage shows results 1-5 and the second montage results 6-10. The name of the image and distance is provided on Line 27.
- **Lines 58-60:** Finally, we display our search results to the user.

So there you have it. An entire image search engine in Python.

Let’s see how this thing performs:

```
Building an Image Search Engine in Python and OpenCV                    Shell
1 $ python search.py --dataset images --index index.cpickle
```

Eventually, after pressing a key with a window active, you’ll be presented with this Mordor query and results:

```
Building an Image Search Engine in Python and OpenCV                    Shell
167 query: Mordor-002.png
168     1. Mordor-002.png : 0.000
169     2. Mordor-004.png : 0.296
170     3. Mordor-001.png : 0.532
171     4. Mordor-003.png : 0.564
172     5. Mordor-005.png : 0.711
173     6. Goblin-002.png : 0.825
174     7. Rivendell-002.png : 0.838
175     8. Rivendell-004.png : 0.980
176     9. Goblin-001.png : 0.994
177     10. Rivendell-005.png : 0.996
```

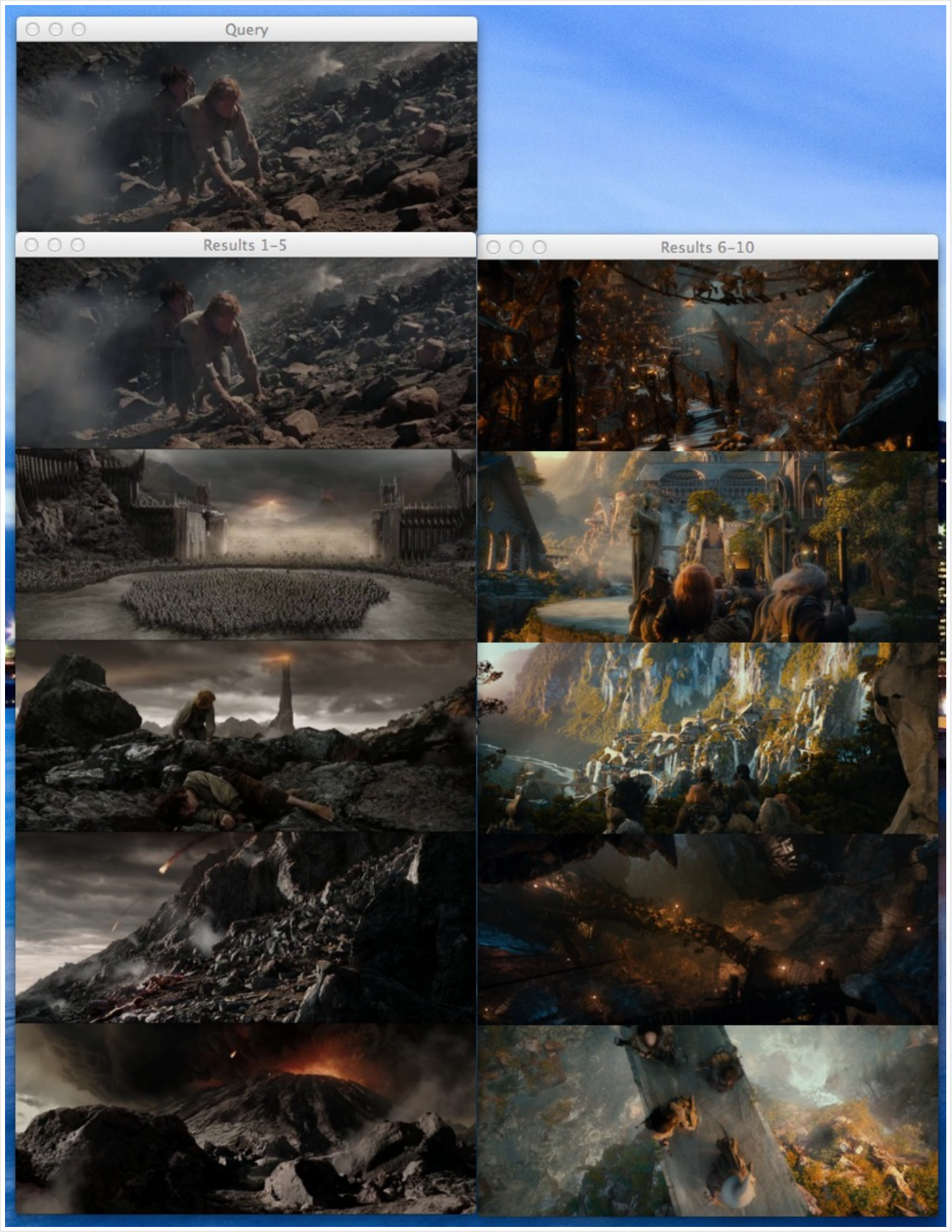


**Figure 2:** Search Results using *Mordor-002.png* as a query. Our image search engine is able to return images from Mordor and the Black Gate.Let's start at the ending of *The Return of the King* using Frodo and Sam's ascent into the volcano as our query image. As you can see, our top 5 results are from the "Mordor" category.

Perhaps you are wondering why the query image of Frodo and Sam is also the image in the #1 result position? Well, let's think back to our chi-squared distance. We said that an image would be considered "identical" if the distance between the two feature vectors is zero. Since we are using images we have already indexed as queries, they are in fact identical and will have a distance of zero. Since a value of zero indicates perfect similarity, the query image appears in the #1 result position.

Now, let's try another image, this time using The Goblin King in Goblin Town. When you're ready, just press a key while an OpenCV window is active until you see this Goblin King query:

```
Building an Image Search Engine in Python and OpenCV                    Shell
query: Goblin-004.png
    1. Goblin-004.png : 0.000
    2. Goblin-003.png : 0.103
    3. Golbin-005.png : 0.188
    4. Goblin-001.png : 0.335
    5. Goblin-002.png : 0.363
    6. Mordor-005.png : 0.594
    7. Rivendell-001.png : 0.677
    8. Mordor-003.png : 0.858
    9. Rivendell-002.png : 0.998
    10. Mordor-001.png : 0.999
```

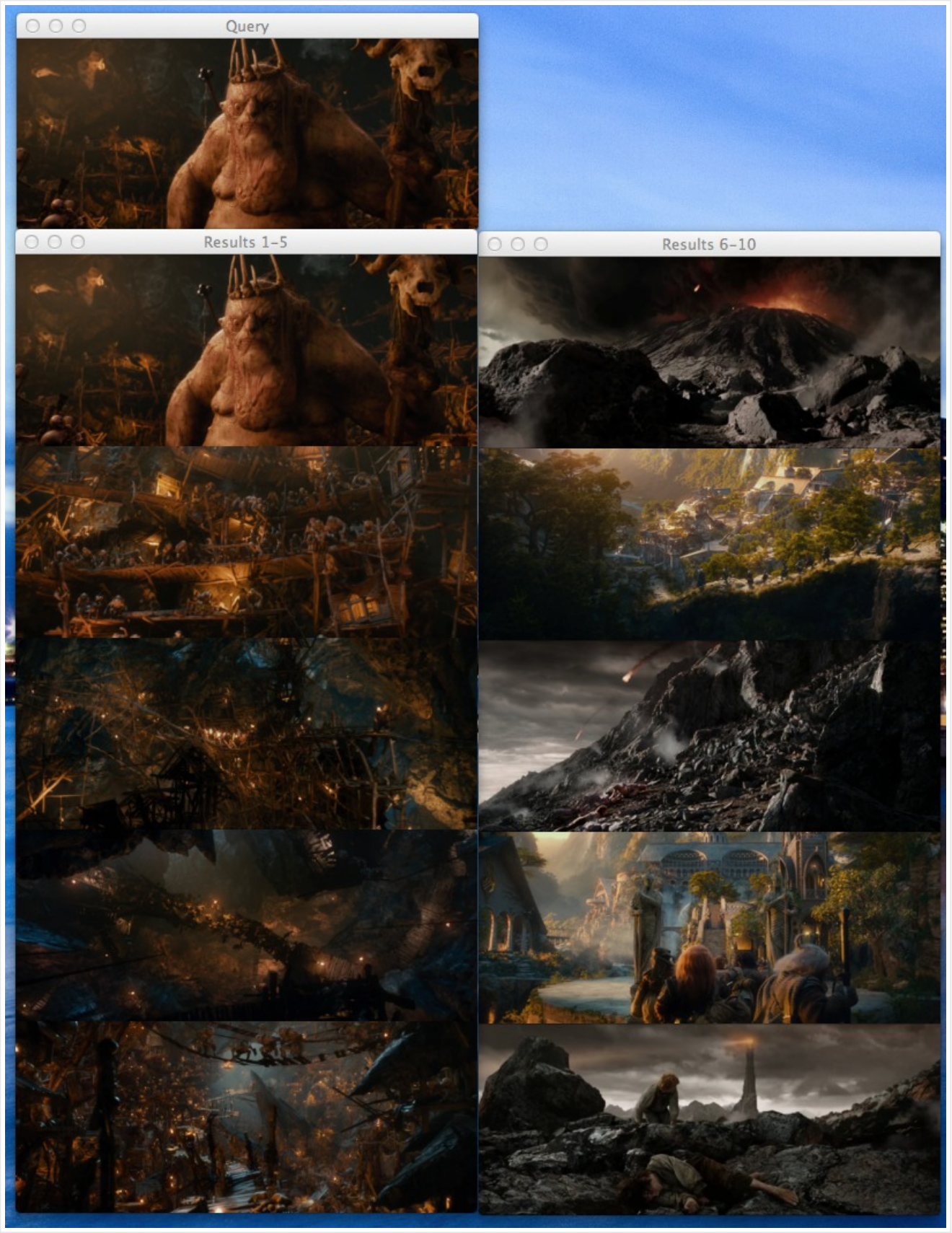


**Figure 3:** Search Results using *Goblin-004.png* as a query. The top 5 images returned are from Goblin Town.The Goblin King doesn't look very happy. But we sure are happy that all five images from Goblin Town are in the top 10 results.

Finally, here are three more example searches for Dol-Guldur, Rivendell, and The Shire. Again, we can clearly see that all five images from their respective categories are in the top 10 results.

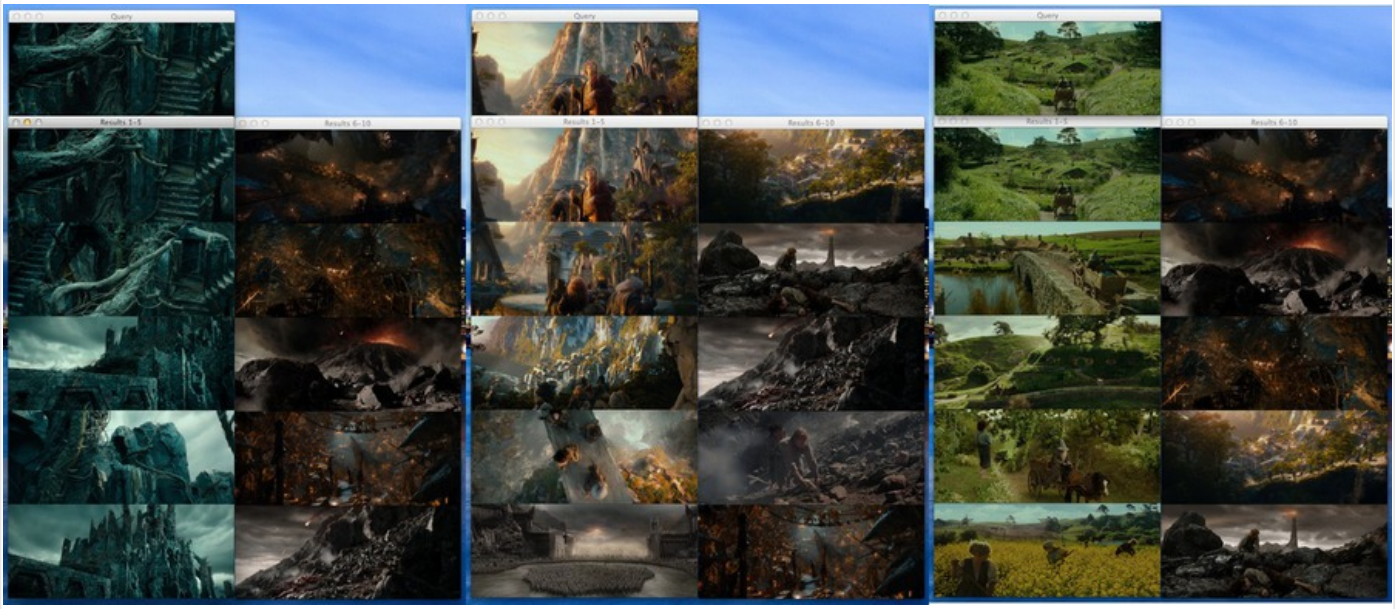

**Figure 4:** Using images from Dol-Guldur *(Dol-Guldur-004.png)*, Rivendell *(Rivendell-003.png)*, and The Shire (*Shire-002.png)* as queries.

## Bonus: External Queries

As of right now, I've only shown you how to perform a search using images *that are already in your index*. But clearly, this is not how all image search engines work. Google allows you to upload an image of your own. TinEye allows you to upload an image of your own. Why can't we? Let's see how we can perform a search using an image that we haven't already indexed:

Building an Image Search Engine using Python and OpenCV — Python

```python
# import the necessary packages
from pyimagesearch.rgbhistogram import RGBHistogram
from pyimagesearch.searcher import Searcher
import numpy as np
import argparse
import os
import pickle
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required = True,
    help = "Path to the directory that contains the images we just indexed")
ap.add_argument("-i", "--index", required = True,
    help = "Path to where we stored our index")
ap.add_argument("-q", "--query", required = True,
    help = "Path to query image")
args = vars(ap.parse_args())

# load the query image and show it
queryImage = cv2.imread(args["query"])
cv2.imshow("Query", queryImage)
print("query: {}".format(args["query"]))

# describe the query in the same way that we did in
```

```
26 # index.py -- a 3D RGB histogram with 8 bins per
27 # channel
28 desc = RGBHistogram([8, 8, 8])
29 queryFeatures = desc.describe(queryImage)
30
31 # load the index perform the search
32 index = pickle.loads(open(args["index"], "rb").read())
33 searcher = Searcher(index)
34 results = searcher.search(queryFeatures)
35
36 # initialize the two montages to display our results --
37 # we have a total of 25 images in the index, but let's only
38 # display the top 10 results; 5 images per montage, with
39 # images that are 400x166 pixels
40 montageA = np.zeros((166 * 5, 400, 3), dtype = "uint8")
41 montageB = np.zeros((166 * 5, 400, 3), dtype = "uint8")
42
43 # loop over the top ten results
44 for j in range(0, 10):
45     # grab the result (we are using row-major order) and
46     # load the result image
47     (score, imageName) = results[j]
48     path = os.path.join(args["dataset"], imageName)
49     result = cv2.imread(path)
50     print("\t{}. {} : {:.3f}".format(j + 1, imageName, score))
51
52     # check to see if the first montage should be used
53     if j < 5:
54         montageA[j * 166:(j + 1) * 166, :] = result
55
56     # otherwise, the second montage should be used
57     else:
58         montageB[(j - 5) * 166:((j - 5) + 1) * 166, :] = result
59
60 # show the results
61 cv2.imshow("Results 1-5", montageA)
62 cv2.imshow("Results 6-10", montageB)
63 cv2.waitKey(0)
```

- **Lines 2-18:** This should feel like pretty standard stuff by now. We are importing our packages and setting up our argument parser, although, you should note the new argument `--query`. This is the path to our query image.
- **Lines 21 and 22:** We're going to load your query image and show it to you, just in case you forgot what your query image is.
- **Lines 28 and 29:** Instantiate our `RGBHistogram` ***with the exact same number of bins as during our indexing step***. I put that in bold and italics just to drive home how important it is to use the same parameters. We then extract features from our query image.
- **Lines 32-34:** Load our index off disk using `cPickle` and perform the search.
- **Lines 40-63:** Just as in the code above to perform a search, this code just shows us our results.

Before writing this blog post, I went on Google and downloaded two images not present in our index. One of Rivendell and one of The Shire. These two images will be our queries.

To run this script just provide the correct command line arguments in the terminal:

```
Building an Image Search Engine in Python and OpenCV                    Shell
1 $ python search_external.py --dataset images --index index.cpickle \
2     --query queries/rivendell-query.png
```

```
query: queries/rivendell-query.png
    1. Rivendell-002.png : 0.195
    2. Rivendell-004.png : 0.449
    3. Rivendell-001.png : 0.643
    4. Rivendell-005.png : 0.757
    5. Rivendell-003.png : 0.769
    6. Mordor-001.png : 0.809
    7. Mordor-003.png : 0.858
    8. Goblin-002.png : 0.875
    9. Mordor-005.png : 0.894
    10. Mordor-004.png : 0.909
$ python search_external.py --dataset images --index index.cpickle \
    --query queries/shire-query.png
query: queries/shire-query.png
    1. Shire-004.png : 1.077
    2. Shire-003.png : 1.114
    3. Shire-001.png : 1.278
    4. Shire-002.png : 1.376
    5. Shire-005.png : 1.779
    6. Rivendell-001.png : 1.822
    7. Rivendell-004.png : 2.077
    8. Rivendell-002.png : 2.146
    9. Golbin-005.png : 2.170
    10. Goblin-001.png : 2.198
```

Check out the results for both query images below:

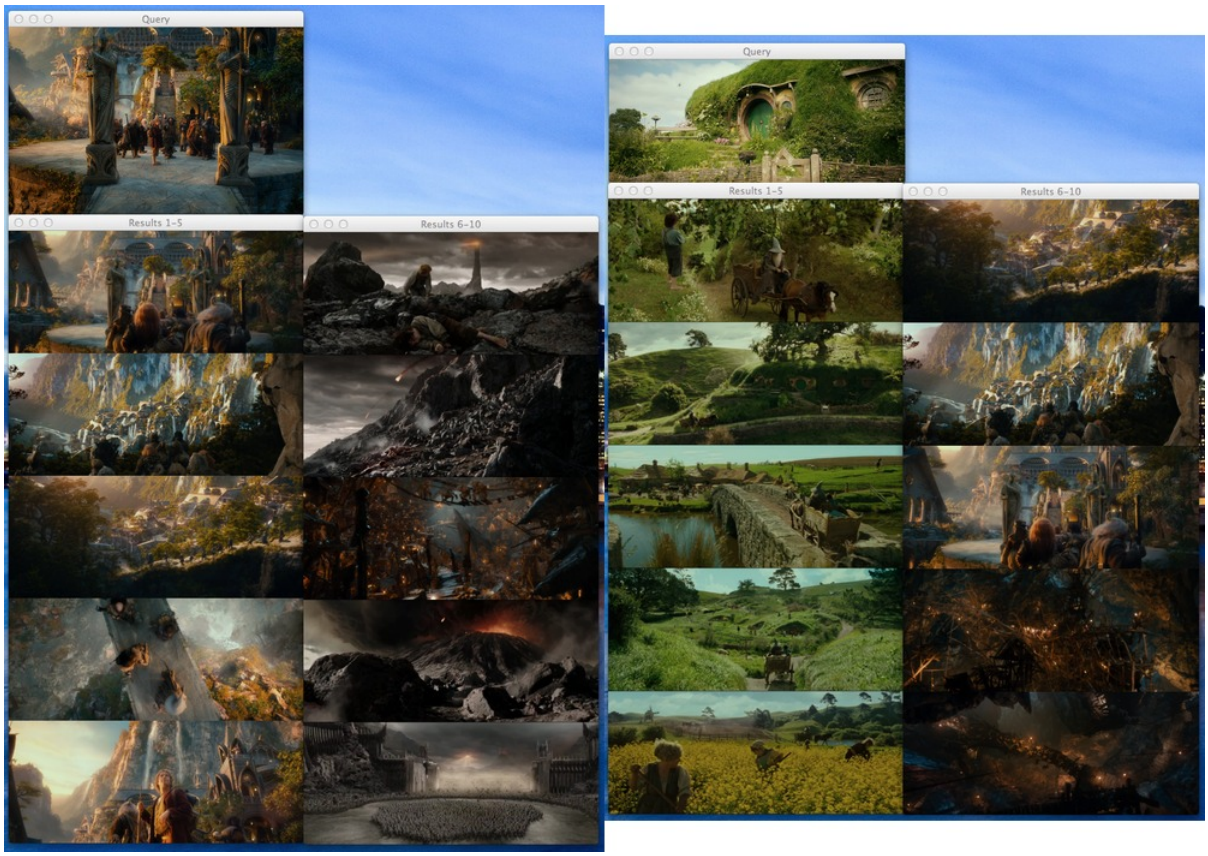


**Figure 5:** Using external Rivendell *(Left)* and the Shire *(Right)* query images. For both cases, we find the top 5 search results are from the same category.

In this case, we searched using two images that we haven't seen previously. The one on the left is of Rivendell. We can see from our results that the other 5 Rivendell images in our index were returned, demonstrating that our image search engine is working properly.

On the right, we have a query image from The Shire. Again, this image is not present in our index. But when we look at the search results, we can see that the other 5 Shire images were returned from the image search engine, once again demonstrating that our image search engine is returning semantically similar images.

## Summary

In this blog post, we've explored how to create an image search engine from start to finish. The first step was to choose an image descriptor — we used a 3D RGB histogram to characterize the color of our images. We then indexed each image in our dataset using our descriptor by extracting feature vectors (i.e. the histograms). From there, we used the chi-squared distance to define "similarity" between two images. Finally, we glued all the pieces together and created a *Lord of the Rings* image search engine.

So, what's the next step?

We're just getting started. Right now we're just scratching the surface of image search engines. The techniques in this blog post are quite elementary. There is a lot to build on. For example, we focused on describing color using just histograms. But how do we describe texture? Or shape? And what is this mystical SIFT descriptor?

All those questions and more, answered in the coming months.

### If you liked this blog post, please consider sharing it with others.

## Downloads:

If you would like to download the code and images used in this post, please enter your email address in the form below. Not only will you get a .zip of the code, I'll also send you a **FREE 17-page Resource Guide on Computer Vision, OpenCV, and Deep Learning.** Inside you'll find my hand-picked tutorials, books, courses, and libraries to help you master CV and DL! Sound good? If so, enter your email address and I'll send you the code immediately!

**Email address:**

Your email address

DOWNLOAD THE CODE!

### Resource Guide (it's totally free).

Enter your email address below to get my **free 17-page Computer Vision, OpenCV, and Deep Learning Resource Guide PDF**. Inside you'll find my hand-picked tutorials, books, courses, and Python libraries to