Mountain Skyline Recognition

Matthew Beldyk
University of Colorado
CSCI 5722
Boulder, Colorado
beldyk@colorado.edu

ABSTRACT

In the past several years there have been leaps and bounds in the amount of processor power and sensors available to the average person when they are in a remote environment. Smart phones with cameras, GPS, and processor power that dwarfs what was available only a few years ago in a desktop have become commonplace. In addition to powerful handheld devices, many high quality geodetic datasets have become available to the public for free.

For my semester project, I studied one of these datasets and the constraints offered by the average smart phone to create a system that attempts to identify geological features from a photograph of a skyline and the location where that photograph was taken. I describe an algorithm for rendering a model skyline from SRTM data and an algorithm for finding the edge between the sky and the ground in a photograph of mountains. I also describe how a match can be found between these two skylines and what future application can be realized from this information.

1. INTRODUCTION

A hiker often finds themselves looking at a mountain and wondering what a peak or other interesting geological feature is called. With modern advances in technologies, specifically related to cellular phones, one can begin to assume that the average person will be carrying quite an array of sensors. [2] Thus a it is reasonable to assume a hiker is also carrying a smart phone equipped with a GPS unit, Internet, and a camera along with all the usual hiking gear. The intention of this project was to attempt to create a system that would help these users identify these features. Using publicly available datasets and simple image processing techniques, I have attempted to create a system that answers that question within the constraints of the sensors a hiker would likely have in their cellular phone.

2. BODY

2.1 The Current State of the Art

There are a number of systems that currently do similar things to what my project attempts to achieve. Marmota currently does everything my system attempts to do, but uses a much richer feature set than what I am attempting to utilize.[3, 4] http://www.heywhatsthat.com is a website that creates the skyline render from DEMs and allows the user to manually identify the skyline they are wondering about [10, 7]. This website only renders the skyline and annotates the model skyline, it does not annotate photographs. The most relevant research that I found is Behringer‘s work [1] where he uses the photograph and DEM skyline alignments as a way to augment other sensors in his system. Behringer created a system quite similar to what I attempted to create. I reused a significant amount of the math from Behringer‘s paper in my research.

2.2 Overview of the Algorithm

The basic flow of this algorithm is fairly simple, there are only a couple steps required to calculate interesting results. The first step is to calculate a number of parameters to set up the system. First, latitude and longitude where the photograph was taken must be calculated. The next step is to load the DEM that corresponds to this location and calculate the elevation at this location. Once the system has the coordinates for the location and the DEM is loaded into RAM, an optimal model 360 skyline must be rendered. After this, the skyline in the photograph must be calculated. The photograph is decimated and several filters are run on the image to calculate the edge where the ground ends and the sky begins. Once both of these skylines are rendered, I treat them as waveforms [9] and try to find where the photograph‘s window best fits the model skyline.

I used two datasets for this project. The first was the SRTM (Shuttle Radar Topography Mission) 30 Meter dataset. This is freely availabe from several NASA datacenters. It gives 30 Meter resolution grid points across the continental United States in 1 lat X 1 lon tiles. The second dataset I used was my own personal collection of photographs. Over the years I have taken thousands of photographs, and often these have been of landscape scenes. I have also been very regimented over this time and have tagged many of my photographs with the location where I took them. I did some initial poking around online for good landscape image datasets that include where the image was taken, and, most importantly, had licensing terms that were agreeable with use in a publication, but I did not have a great deal of luck finding such a dataset. So I created my own small 25 image dataset, hand selected from all my photographs to illustrate, what I hypothesised, would be difficult skylines to parse.

2.3 Rendering from Model

The first step in my algorithms is to create a model skyline from some geodetic dataset. I decided to use the SRTM 30M DEM dataset as the basis of my model of the skyline. The SRTM dataset provides 1 lat by 1 lon square tiles that each contain a 3601x3601 matrix of elevations to meter resolution.

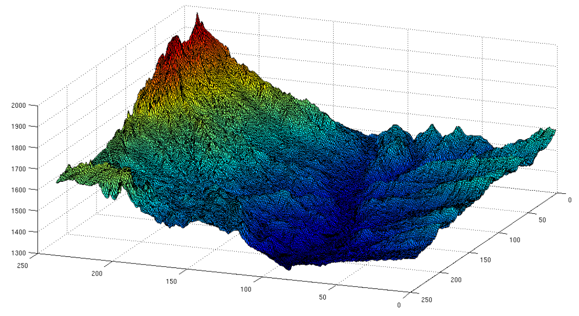


Figure 1: An example of the decimated SRTM DEM for the area nearby Boulder, Colorado (the N40W105 tile.)

[5] This is a simple format to read, but care must be taken to make sure that the corners of the tiles are aligned properly and that the tile is not being read backwards. If either of these cases happen, the error is not immediately apparent and can easily cascade later into the system.

When I create my render, I load the DEM for the location where the photograph was taken. A potential pitfal, is if the camera's location is too close to the edge of the tile and a hole will appear. If the camera location is too close to the edge of the DEM tile, I also need to also render the contents of neighbor tiles. Once I my DEM is loaded, I need to render a full 360 degree view from where the photo was shot.

To create this render, I use an algorithm similar to ray tracing. I create a blank image that represents this view. So that I do not have to deal with as many normalization issues later, I take the field of view of my photo (online, one can look up the field of view parameters for a given camera for its various settings) and the number of pixels it is wide and tall to decide what size the 360 degree view model should be. There are limitations to the amount of RAM in the computer for this render, so this 360 degree view should not be too large, and the image must be decimated. To calculate the size of the render image:

$$width_{renderImage} = ceil(\frac{width_{photo} * 2 * \pi}{fieldOfView_{horizontal}}) \tag{1}$$

$$height_{renderImage} = ceil(\frac{height_{photo} * \pi}{fieldOfView_{vertical}}) \tag{2}$$

For every point in the DEM tile(s), I project this point into the image space. I look at the point's elevation, distance, and location to calculate the angle where it belongs in my render. I use an external function [6] to calculate the distance to a DEM grid point for the perfect skyline, but at the latitudes of my testset, I can treat the DEM tile as having cartesian coordinates; this allows me to significantly speed up the processing of the skyline[1]. I precalculate the size of the DEM tile in meters using vdist()[6] and create the variables $dist_{\mathrm{perLat}}$ and $dist_{\mathrm{perLon}}$.

$$lat_{dist} = dist_{perLat} * (lat_{point} - lat_{camera}) \tag{3}$$

$$lon_{dist} = dist_{perLon} * (lon_{point} - lon_{camera}) \tag{4}$$

$$distance = \sqrt{lat_{dist}^2 + lon_{dist}^2} \tag{5}$$

$$azimuth = atan2(lat_{dist}, lon_{dist}) \tag{6}$$

I then need to calculate the angular elevation of the point. Using $elev_{point}$ and $elev_{camera}$ from the DEM

$$elevation_{angle} = atan(\frac{elev_{point} - elev_{camera}}{distance}) \tag{7}$$

After this, I need to project these angles into the actual render picture

$$pixel_{height} = floor(\frac{height_{renderedImage} * elevation_{angle}}{\pi}) \tag{8}$$

I store the latitude, longitude and distance from the camera for each pixel (for future extensions of this research.) During this projection, I felt it would be important to give precedence to closer items in each image pixel, so I do a check to see if the newest point is closer than the point currently projected into the same pixel. In some future version of this algorithm, it might make sense to look at other features that appear from one mountain in front of another, instead of simply the ground/sky edge. For example, with this mode of rendering, I might be able to also annotate mountains in my foreground or use canyons and gullies as features in a future version of this system.

There are a number of issues that arise with this approach. The first one noticed is that all of the points in the DEM are just that: points. The skyline will result in a jagged edge that is not particularly useful for this system; Figure 2 shows an example of this. I solved this approach by asserting that each point would represent a window halfway to its neighbors. The azimuth for each point becomes a window calculated by taking the maximum and minimum angles calculated by adding half the distance to the neighboring points to the point I am attempting to render. This approach creates a more realistic render that looks as if I am rendering a series of square tiles instead of a cloud of points; it also has the advantage that it makes closer items look larger and farther items look smaller. To do this render I must first determine what location is halfway to the next grid point. Assuming the grid is square and size $size_{grid} * size_{grid}$ and we are counting points $point_{ij}$ to calculate the actual point location

$$lat_{point} = \frac{i}{size_{grid}} + lat_{dem} \tag{9}$$

$$lon_{point} = \frac{j}{size_{grid}} + lon_{dem} \tag{10}$$

Then the relative azimuths must be calculated using lat_{point} and lon_{point} using equations 3 and 6 In reality, the minimum and maximum azimuths must be calculated from

$$lat_{points} = \frac{i \pm 0.5}{size_{grid}} + lat_{dem} \tag{11}$$

$$lon_{points} = \frac{j \pm 0.5}{size_{grid}} + lon_{dem} \tag{12}$$

I then fill in the values for this specific point between the minimum and maximum azimuths found. Figure 3 shows the improvements this enhancement shows.

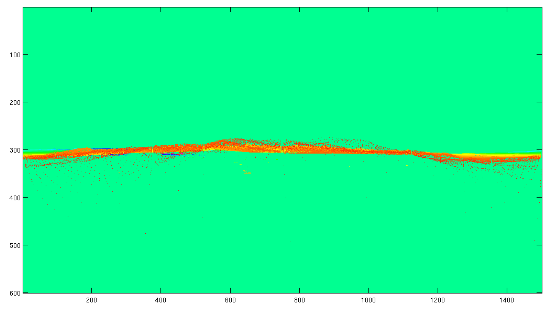


Figure 2: An example of a render using only the grid points and doing no interpolation.

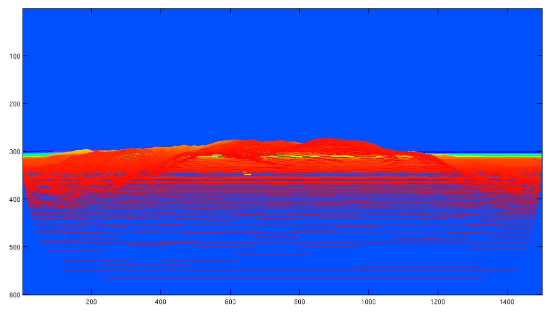


Figure 3: An example of a render with interoplated grid points showing a more realistic skyline.

Once I have rendered the entire skyline, I need to calculate the actual skyline waveform. The algorithm to do this is much simpler than in the photo, I simply find the highest defined pixels in the image and store these values and their angles in the rendered image. These will be use at a later stage.

2.4 Finding Skylines in Photos

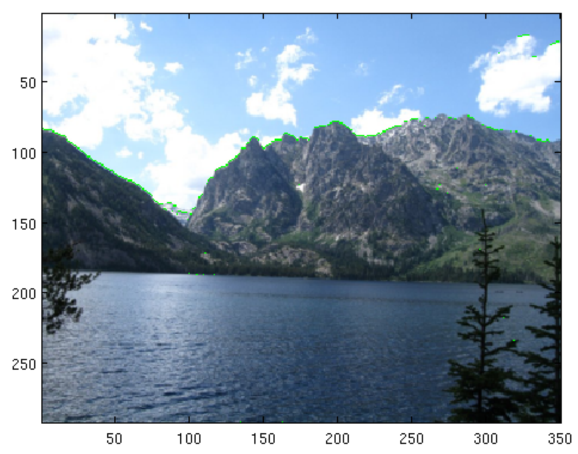
There are numerous ways to find the edge where the ground meets the sky in a photograph. Before I finally found a method that suited my purposes, I tried several convolution based methods; eventualy I settled on a horizontal Sobel edge detector with some heuristic logic. My initial testing involved using Matlab‘s built in edge detectors [8] because it was trivial to try a large number of different filters quickly, but for my final version I wrote my own. Initially, I tried a Canny edge detector, but it had too many false positives in the sky. I next attempted to use several simple convolution [9] based filters of several different size, but most had too many false positives or simply did not find the skyline edge. Finally, I settled on a 3x3 horizontal Sobel filter. [1]

As a preprocessing step, I decimate the image into one much smaller so that I have enough RAM in my machine to store the full 360 degree model render. For my system, I decimate my photographs to 150 pixels wide; I found this to be a good compromise as large enough to still see features, but small enough that I would not use too much RAM for my model render. After I decimated, I would find the results of a convolution with a horizontal Sobel filter. Once I had the results from the convolution, I would apply a secondary filter to only keep filter responses above a certain threshold; figure 4b shows an example of these filter responses calculated from figure 4a. After this filtration, I would search through each column of the image looking for the highest edge found; the relative elevation of these points in my image would be the skyline; figures 5a, 5b and 5c shows images annotated by this algorithm.

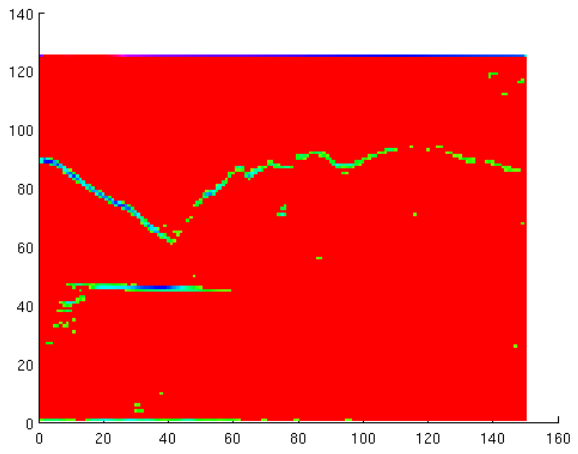
There are a number of assumptions in my code. First, I make the assumption that the camera is upright, and the photograph was not taken at a strange angle. I also make the assumption that there are no occlusions in the image, because they will appear as part of the skyline, and eventually throw off the entire eventual match. This assumption is rather problematic, as it is rare to find a photograph with nothing nearby obscuring the full skyline, therefore cropping of images as a preprocessing step may be required. Figure 6 shows an example of both occlusions and clouds. Another issue with my algorithm is that cumulonimbus and other fluffy white clouds create false positives for my algorithm. Another problematic situation is a wide open plane with mountains in the far distance; my algorithm will miss the light blue mountains and find the edge between the plane and the mountains (see Figure 7.)

2.5 Matching the Photograph Skyline with the Model Skyline

Once the two skylines have been calculated, the subwindow that the photograph viewport represents must be matched with the corresponding view in the model skyline. The literature suggests using local maximums and minimums (small



(a) Original Image with Annotated Skyline



(b) Threasholded Sobel Responses

Figure 4: An example Sobel filter response

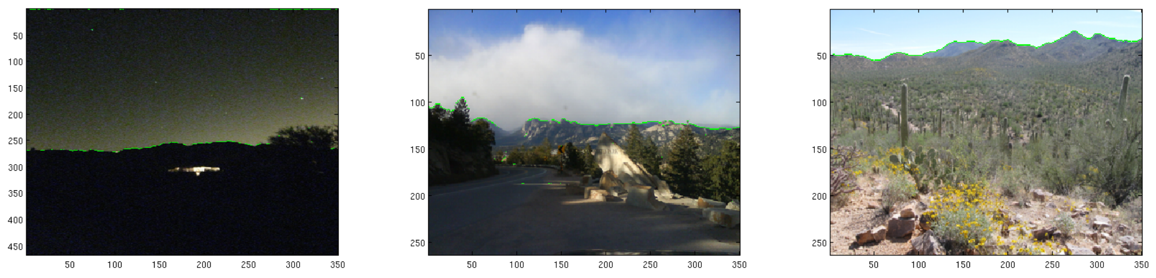


Figure 5: Several examples of good skyline edge finding

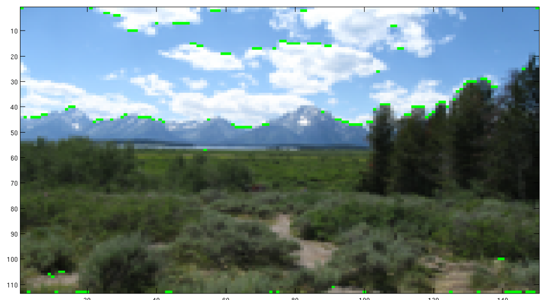


Figure 6: An example of failure due to both clouds and occlusions.

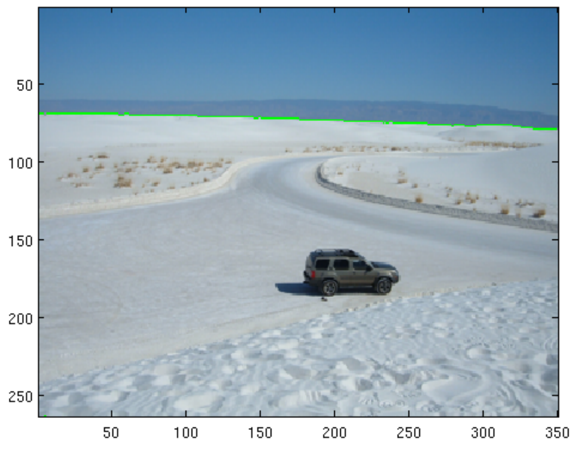


Figure 7: An example of failure due a weak response of the actual skyline.

peaks and valleys in the skyline) as features to match. My approach was to treat the entire skyline waveform from the photo as a feature vector with a number of missing data points. At this point I have a waveform that goes through the photograph at some elevation in the field of view. I need to normalize this waveform‘s absolute angles to some relative angle. I choose the median angle of all the skyline points that are actually defined, and then I subtract this median angle from the absolute angle in the skyline to create a new skyline with a median of zero. I would then iterate over the entire model skyline looking at every possible window in a 360 view around the model skyline. Each hypothesis window is normalized in the same way that I normalize the photo skyline, thus I am comparing two similar feature vectors. My algorithm then looks for the closest match using a modified Euclidean distance, but ignoring the pixels in the feature vectors where they are undefined in the photograph skyline. Once I find the best match in the model skyline, I then need map the pixel coordinates back to the actual angles of the field of view.

3. RESULTS

In the end, I did not have enough time to create a completely working system. The final code for matching the two skylines that I created has a bug in it that is returning wrong results. I believe it to be something trivial, but there is no longer any time in the semester to complete debugging the codebase. Once that bug is fixed, the next step is to project interesting visible geological features into the photograph and annotating the image.

This said, I was successful in creating a system that was able to find the shape skyline edge in all but two of the 30 images in my test set. I also created a system that can create a skyline from a model in a fairly quick amount of time (a full undecimated run of the entire system takes under 20 minutes on a 2.2GHz AMD Turion.) Slightly decimated versions of the dataset run significantly faster; with a minimal amount of decimation, the loading of the 25MB SRTM datafile from the filesystem becomes the largest bottleneck.

4. FUTURE WORK

The code created in the project is a great start for an augmented reality mobile application that solves the problem I initially intended to solve. In a real version of the system, other ancillary sensors would be utilized to improve the accuracy of the identification. For example, many cell phones that contain GPS units also contain a digital compasses, tilt meters, and accelerometers which may be accurate enough to no longer need to do computer vision identification of the skyline. In fact, I intend to take much of what I learned with this project to create an augmented reality cellphone application that identifies mountains this winter break, but I do plan to utilize these other external sensors to cut down on processor time.

5. CONCLUSIONS

During the process of creating this system, I found many interesting and challenging avenues for future work. I also reached a number of dead ends trying to get various parts of my test set to work. The immediate conclusion is that even a simple task such as segmenting the sky from the ground is non-trivial. And to do feature recognition and identification, there needs to be a very rich feature set, otherwise one will not have anything to compare against. Overall, this project was a wonderful way to explore various tecniques for mountain recgonition. In the future I intend to take a lot of what I have learned to create a smart phone application. Although I will likely leave out much of the visual recognition features and instead rely more on the other sensors these phones have for better a shorter processing time (and a better user experience), this does not mean that the work I accomplished is not useful; it simply means I may need to wait another year or two before I can utilize the full dataset I would like without using too much processor time.

I also learned about how hard it is to get a perfect dataset that does not require any massaging to be useful. Even though we have 300+ sunny days in Colorado, it is exceedingly rare to have a full view of the skyline without any clouds obscuring a couple peaks that would be useful as features. These days do happen, but they only last for a couple hours, at best, and inevitably happen when I didn't have my camera for data collection.

6. REFERENCES

[1] R. Behringer. Registration for outdoor augmented reality applications using computer vision techniques and hybrid sensors. In *Virtual Reality, 1999. Proceedings., IEEE*, pages 244–251. IEEE, 2002.

[2] E. Burnette. *Hello, Android: Introducing Google's Mobile Development Platform.* Pragmatic Bookshelf, 2010.

[3] P. Chippendale, M. Zanin, and C. Andreatta. Environmental Content Creation and Visualisation in the 'Future Internet'. In *Future Internet-FIS 2008: First Future Internet Symposium Vienna, Austria, September 28-30, 2008 Revised Selected Papers*, page 82. Springer-Verlag New York Inc, 2009.

[4] P. Chippendale, M. Zanin, and C. Andreatta. Spatial and Temporal Attractiveness Analysis through Geo-Referenced Photo Alignment. In *Geoscience and Remote Sensing Symposium, 2008. IGARSS 2008. IEEE International*, volume 2. IEEE, 2009.

[5] T. Farr, P. Rosen, E. Caro, R. Crippen, R. Duren, S. Hensley, M. Kobrick, M. Paller, E. Rodriguez, L. Roth, et al. The shuttle radar topography mission. *Reviews of Geophysics*, 45(2), 2007.

[6] M. Kleder. Vectorized geodetic distance and azimuth on the wgs84 earth ellipsoid. http://www.mathworks.com/matlabcentral/fileexchange/8607-vectorized-geodetic-distance-and-azimuth-on-the-wgs84-earth-ellipsoid.

[7] M. Kosowsky. Hey, what's that?, Dec. 2010. http://www.heywhatsthat.com/.

[8] A. McAndrew. *An Introduction to Digital Image Processing with MATLAB.* Course Technology Press Boston, MA, United States, 2004.

[9] A. V. Oppenheim and R. W. Schafer. *Digital Signal Processing.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[10] J. Wong and J. Hong. What do we mashup when we make mashups? In *Proceedings of the 4th international workshop on End-user software engineering*, pages 35–39. ACM, 2008.