Computer Vision Based Route Mapping

by Ryan Kehlenbeck and Zack Cody
Department of Computer Science and Software Engineering
California Polytechnic State University, San Luis Obispo

June 2017

Abstract

The problem our project solves is the integration of edge detection techniques with mapping libraries to display routes based on images. To do this, we used the OpenCV library within an Android application. This application lets a user import an image from their device, and uses edge detection to pull out a path from the image. The application can find the user's location and uses it alongside the path data from the image to create a route using the physical roads near the location. The shape of the route matches the edges from the given image and the user can then follow along in real time. We chose this project to further explore the use of technologies like computer vision and web APIs, while also creating a useful app.

1 Introduction

This project is relevant to us because it allows us to further our current knowledge we have gained while at Cal Poly. Both of us have had experience within the past year working with web API's but only ones that provide little processing of information we send to the cloud. This project gave us the opportunity to work with more well defined web API's provided by Google which are much more verbose. In addition to this we also learned about computer vision techniques, specifically edge detection. Neither of us have taken a class where we were formally taught about computer vision and so this was a chance for us to learn new technologies we could apply to future projects.

Not only were we able to learn more about these technologies, but also got to solve a problem for a group of users. People who are long distance runners or bikers constantly are going out in the world to either practice for future competitions or because they just enjoy the activity. Our application gives an added user experience to these users because they can now have a mapping software that takes the hassle away of figuring out long routes they can take where they live. Not only does our application give them the option to define a distance of the route that is generated, but they can also have more fun by creating routes out of images which adds a fun element. This can help eliminate the mundane task of planning out a route on a map themselves which in the long run can save them time and create a more fun experience overall.

Since we were attempting to solve an actual problem that users face this adds to the relevance of our project. As software engineers we spent the last four years learning how to solve real world problems that affect people. By going through the process of identifying a problem and developing a software solution to the problem we have effectively used many of the techniques and skills acquired throughout our college experience making this project extremely relevant to our success.

Some previous works that are similar to our project are route generator applications. Most of these have the user select various locations that they want to stop at along the way. There are a couple applications that allow users to draw on the map and then have that generate a route that resembles what they drew. This is the closest we found to actual applications that are similar to ours. However the one addition that makes ours stand out is the image to route feature. There have been many other projects that use edge detection in applications such as facial recognition or image overlay features. But from market research it appears that our application is the only one that integrates these two technologies into a single platform.

2 Implementation

We designed and implemented this project as an application for the Android operating system. After exploring different relevant technologies, we decided to use the OpenCV library for the edge detection, the Google Maps Android library for our map, and Google Maps Web APIs for fitting the route to the map. We began by planning a feature set and timeline for the project.

In order to follow some of the principles of Agile software development in our implementation, we broke our feature set down into two week sprints spanning two quarters. Our first sprint was spent planning, and building a user interface prototype. This includes the time we spent drawing user interface sketches and refining them out in Photoshop. We were influenced by Material Design and tried to make the app look and feel like any other Android app. The rest of the quarter was spent implementing the core features of our app. These features were edge detection for the path, conversion of the path to geographic coordinates, and converting our path to a route on a map.

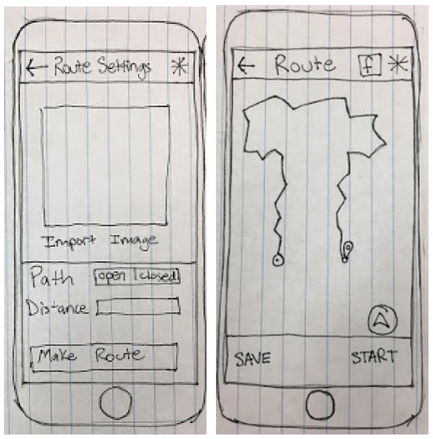


Figure 1: User interface sketches

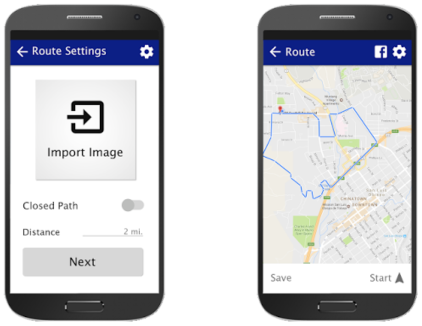
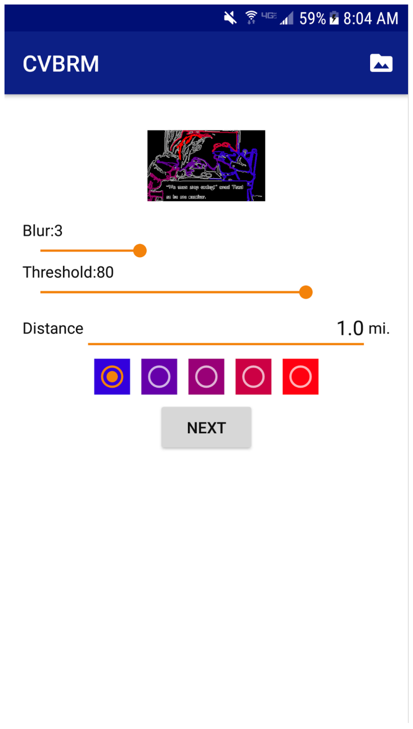


Figure 2: User interface prototype

The second quarter was spent improving our existing features and adding a few new ones to the app. We wanted to improve the paths that were generated by the edge detection algorithm because they had lots of imperfections. Other features we added were the ability to select a start location for the route, the ability to save and load routes, a canvas to draw paths instead of detecting them, and location tracking.

Getting the edges from the images was the first problem we faced. We researched how to perform edge detection using OpenCV and decided to use the Canny algorithm to locate the edges and extract them from the image. These paths were not perfect so we gave the user the parameters to adjust the results. We blur the image and set a threshold for the Canny algorithm, so these are the parameters the user can adjust.



Blur:3
Threshold:80
Distance 1.0 mi.

Figure 3: The edge detection configuration screen

However, the path still needed to be manipulated because of other errors that came up as a result of the edge detection. The path contained zig-zags that messed up our conversion to geographic coordinates because these repeated segments would use up significant portions of the user specified route distance. If the user wants to run two miles, but half of the Cartesian path is a single segment repeated over and over again, the result would be the user running back and forth for the first mile and the rest of the shape would cover the remaining mile. Another restriction was the API call we make to Google, which limits us to paths of 100 points or less, so the thousands we were getting from the edge detection results needed to be cut down.

We remove straight segments and zig-zags by checking the angle between each pair of points and removing them if they are within a certain threshold, usually just a few degrees. If after performing the previous operation points still need to be removed, we find the best points to remove by determining how much the line will be changed by removing that point. We find a point on the straight line between each point's neighboring points that has a ratio equivalent to the ratio of the distance from the original point to its neighbors. The distance between the original point and this new point is our error metric and we eliminate points beginning with those with the smallest error, until we reach our goal of 100 points.

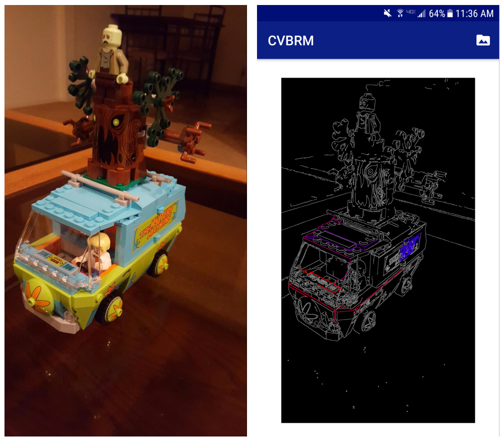


Figure 4: When using real images, the lines are often short and broken up. The colored lines above are the longest segments found from the image on the left.

Once we had the points for our path, we needed to convert them to geographic coordinates so they can be displayed on a map. To do this we needed a starting geographic location for each path and the total distance the path should cover, so we prompt the user for this information. We used an algorithm based on the Haversine formula for calculating distances on a sphere, using the approximate radius of the earth.

In order to get data about maps, locations, and roads, we used Google Maps APIs. This allowed us to display a map in the app, get the coordinates of addresses as starting points, get the user's location, and fit the path to roads. Since we converted all of our points from cartesian space to geographic coordinates, we simply write them to a String in pairs and send them to the Google Maps Snap to Roads API, which returns either a point per road along your path, or a route that follows the roads and does not cut through any buildings. For the purpose of our app, we use the latter feature and we display the returned route on the map in the application. If the user has given the app permission to access their location, they can use the app to track their position on the route as they run or cycle.

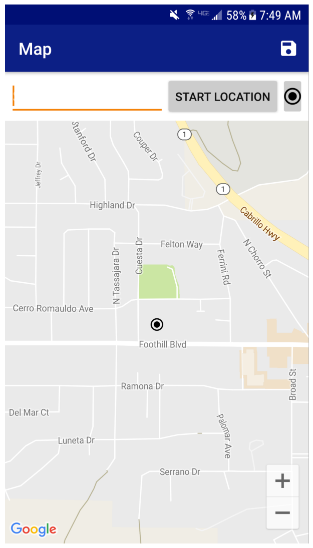
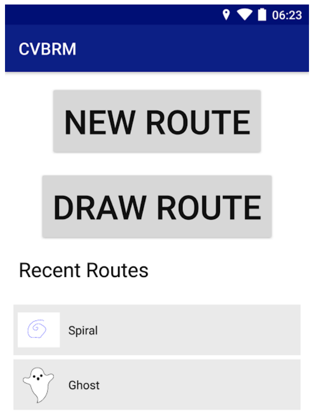


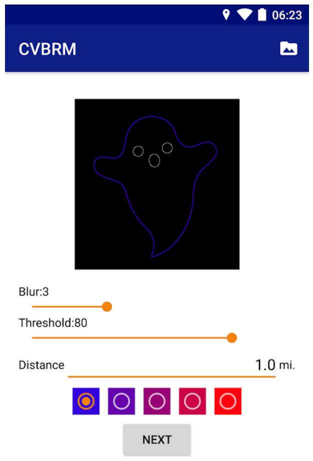
Figure 5: The map screen of our application marking the user's current location.

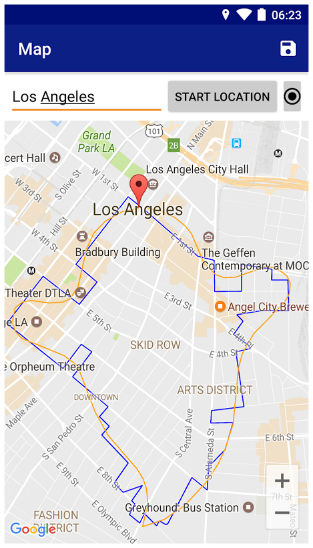
Given the nature of our app, it was very difficult to test. Aside from the fact that Android user interface components are hard to test, the other features of our app are potentially even harder to test. We verified the performance of our app with manual testing. The reasoning behind this is that we did not have any way to verify the results of edge detection without looking at images. Even at the end of the project, our lines aren't perfect, and there was no way for us to guarantee a best contour from an image. The other features of our app work with web APIs, which we used free payment tiers for, so we had a limited number of calls and felt it was best to assume we got proper data from each API call. Ultimately, our app doesn't have any major flaws and very rarely crashes, so we are satisfied with the app's performance.

3 Results

Here is an example flow of our application.







The system image is simple to understand as all of the application can operate in 3 screens. The first screen displays options for generating new routes or referencing old ones. In this example I chose to create a new route based on an image. Then in the next screen I can see the edge that is generated from the image and play with the computer vision options to get a better edge. Finally on the last screen my image is projected onto the streets of Los Angeles where I can then follow along in real time.

Overall from a holistic view, we accomplished everything that we initially set out to do. There were a couple instances where we had to change exactly how we were planning on implementing certain features, but in the end we found solutions to address those issues. Specifically looking at integrating into Google Maps, we had hoped to export our users from our app into theirs allowing users to use a navigation app they may already be comfortable with. But after our research and attempted solutions we could not find an elegant solution to fully integrate into Google Maps from our application. To work around this we added in our own location services. This allows our users to see where they are and follow along the route that we generated for them. This still gives the illusion of navigation although it is not as engaging as Google Maps is.

The current timeline of our project has come to an end, but there is still a lot of future work that our application can gain from. The current state of the application is a working state that provides users with the functionality we promised. But we believe this needs to be expanded upon in order to truly make our application a success. Specifically we would like to address three main aspects we could improve in. Improving the edge detection within the application would greatly increase the accuracy of the route that we produce. In addition to this it would also allow more complex images to be processed through our application. This is a challenge for us currently because we are strictly relying on the OpenCV library to do the bulk of the work for edge detection. At a certain point if we want to expand this functionality we might need to implement our own edge detection algorithms. Finally to add to the user experience we see potential for us to integrate more with social media such as Twitter and Facebook. This would allow users to publish routes they made online and create more of a community around our app.

References

[1] "Contour / polyline simplification." [Online]. Available: https://forum.openframeworks.cc/t/contour-polyline-simplification/4461

[2] G. Android, "Making your app location-aware." [Online]. Available: https://developer.android.com/training/location/index.html

[3] ——, "Saving data." [Online]. Available: https://developer.android.com/training/basics/data-storage/index.html

[4] Google, "Google directions api documentation." [Online]. Available: https://developers.google.com/maps/documentation/directions/intro

[5] ——, "Google maps services documentation from github." [Online]. Available: https://github.com/googlemaps/google-maps-services-java

[6] ——, "Google roads api documentation." [Online]. Available: https://developers.google.com/maps/documentation/roads/nearest

[7] P. S. H. Michael Garland, "Surface simplification using quadric error metrics." [Online]. Available: https://people.eecs.berkeley.edu/ jrs/meshpapers/GarlandHeckbert2.pdf

[8] O. Ogbo, "How to use a web api from your android app." [Online]. Available: http://www.androidauthority.com/use-remote-web-api-within-android-app-617869/

[9] S. Onyszko, "Opencv in advanced android development: Edge detection." [Online]. Available: https://blog.zaven.co/opencv-advanced-android-development-edge-detection/

[10] A. Poliakon, "Importing custom map into google nav." [Online]. Available: https://productforums.google.com/forum/#!topic/maps/DwPoT0pklas

[11] S. Schmitz, "Simplifying a contour to a fixed length, to smooth over several frames." [Online]. Available: http://stackoverflow.com/questions/23525856/simplifying-a-contour-to-a-fixed-length-to-smooth-over-several-frames

[12] M. T. Scripts, "Calculate distance, bearing and more between latitude/longitude points." [Online]. Available: http://www.movable-type.co.uk/scripts/latlong.html

[13] D. D. Waele, "Using google apis on your map : Directions and places." [Online]. Available: https://ddewaele.github.io/GoogleMapsV2WithActionBarSherlock/part5