Land Lines

- Summary
- Line Detection
- Serverless
- Connected Line

Summary

Land Lines is an experiment that lets you explore Google Earth satellite images through gesture. Using a combination of machine learning, data optimization, and graphics card power, the experiment is able to run efficiently on your phone's web browser without a need for backend servers. This is a look into our development process and the various approaches we tried leading us to the final result.

https://g.co/LandLines

When the Data Arts team approached me about exploring a data set of earth images I was quite excited - the images were beautiful, revealing all different kinds of structures and textures, both human made and natural, and I was intrigued with how to connect this data set. I did a variety of initial experiments looking at image similarity and different ways of filtering and organizing them.

> {% Img src="image/T4FyVKpzu4WKF1kBNvXepbi08t52/Ouc4asBmTxj5qzJ9neV9.jpg", alt="t-sne similarity layout", width="800", height="392" %}
>
> t-sne similarity layout, high res 50 mb

As a group we kept coming back to the beautiful and dominant lines in the images. These lines were easy to spot - highways, rivers, edges of mountains and plots of land - and we designed a few projects to explore these. As an artist I was inspired by the beautiful things you can do with collections of lines - see for example Cassandra C Jones's work with lightning

- and I was excited to work with this data set.

Line Detection

One of the initial challenges was how to detect lines in the images. It's easy to take out a piece of tracing paper, throw it on top of a printout of one of these photos, and draw the lines that your eye sees but in general computer vision algorithms for finding lines tend to not work well across very diverse images.

I developed a previous version of the search by drawing algorithm on a project with Local Projects and for that we hand annotated the lines to search for. It was fun to draw on top of artworks but tedious as you move from dozens of images to thousands. I wanted to try to automate the process of finding lines.

With these aerial images I tried traditional line detection algorithms like openCv's canny edge detection algorithm but found they gave either very discontinuous line segments or if the threshold were too relaxed, tons of spurious lines. Also, the thresholds to get good results were different across different image sets and I wanted an algorithm for finding a consistent set of good lines without supervision.

I experimented with a variety of line detection algorithms including recent ones like gPb (PDF) which although producing amazing results, required minutes to run per image. In the end I settled with Structured Forest edge detection, an algorithm that ships with openCV.

Once I had a good "line image", I still had the problem of actually getting the lines and identifying individual lines from each other - i.e., how do I take this raster data and make it vector. Often times when I'm looking at computer vision problems, I investigate imageJ, an open source java based image processing environment used by scientists and researchers which has a healthy ecosystem of plugins. I found a plugin called ridge detection, which helps take an intensity image and turn that into a set of line segments. (As a side note, I also found this edge detection and labeling code from Matlab useful).

> {% Img
> src="image/T4FyVKpzu4WKF1kBNvXepbi08t52/VAUNajQgeKSnqGdKWSMy.png",
> alt="Image with detected line segments", width="600", height="210" %}
> Image with detected line segments

Serverless

I also wanted to see if it's possible to do a data visualization app that's essentially serverless, where the hard work of matching and connecting happens client side. I usually work in

openFrameworks, a c++ framework for creative coding and besides the occasional node project I haven't done a lot of server side coding. I was curious if it's possible to do all of the calculation client side and to only use the server just for serving json and image data.

For the draw application, the matching is a very heavy operation. When you draw a line, we need to find the closest match among over tens of thousands of line segments. To calculate the distance of one drawing to another we use a metric from dollar gesture recognizer which itself involves many distance calculations. In the past, I've used threading and other tricks but in order to make it work in real time on a client device (including mobile phones) I needed something better. I looked into metric trees for finding closest/nearest neighbors and I settled on vantage point trees (javascript implementation). The vantage point tree basically gets built off a set of data and a distance metric and when you put in a new piece of data it gives you quite quickly a list of the closest values. The first time I saw this work on a mobile phone instantly I was floored. One of the great benefits of this particular vantage point tree implementation is that you can save out the tree after it's computed and save on the costs of computing this tree.

> {% Img src="image/T4FyVKpzu4WKF1kBNvXepbi08t52/FkvF5ElUNo0CCCy3AWrS.png", alt="Closest results", width="600", height="346" %} {% Img src="image/T4FyVKpzu4WKF1kBNvXepbi08t52/4TxtCA5qFXb8p4WneqQh.png", alt="Drawn results", width="600", height="314" %}
> Examples of results from the vantage point tree, drawn input is on the right side and the closest results are on the left.

Another challenge of making it work without a server is getting the data loaded onto a mobile device - For draw, the tree and line segment data was over 12mb and the images are quite large, we wanted the experience to feel quick and responsive and the goal is to was try to keep the download small. Our solution was to progressively load data. In the draw app we split the vantage point tree data set into 5 pieces and when the app loads it only loads the first chunk and then every 10 seconds it loads another chunk of data in the background, so essentially the app gets better and better for the first minute of being used. In the drag app was also worked hard to cache images so that as you drag, new images are loaded in the background.

Finally, one thing I found harder than expected was making a pre-loader for both apps, so you the initial delay as data loads would be understandable. I used the progress callback on the ajax requests and on the pixi.js side, checked images that were loading asynchronously had actually loaded and use that to drive the preload message.

Connected Line

For drag, I wanted to create an endless line from the lines we found in the edge detection. The first step was to filter lines from the line detection algorithm and identify long lines that start on one edge and end on one of the three other edges.

{% Img src="image/T4FyVKpzu4WKF1kBNvXepbi08t52/SA7JLr9pPht7PagxYYA4.png", alt="Good lines for connecting marked in red", width="600", height="566" %} {% Img src="image/T4FyVKpzu4WKF1kBNvXepbi08t52/jmaLjxyXlruEhc3F8dqC.png", alt="Good lines for connecting marked in red", width="600", height="405" %}
Good lines for connecting marked in red

Once I had a set of long lines (or to use a more accurate term, polylines, a collection of connected points) in order to connect them I converted these lines into a set of angle changes. Usually when you think of a polyline you imagine it as a set of points: point a is connected to point b which is connected to point c. Instead, you can treat the line as a set of angle changes: Move forward and rotate some amount, move forward and rotate some amount. A good way to visualize this is to think about wire bending machines, which take a piece of wire and as it's being extruded perform rotations. The shape of the drawing comes from turning.

If you consider the line as angle changes and not points, it becomes easier to combine lines into one larger line with less discontinuities - rather than stitching points you are essentially adding relative angle changes. In order to add a line, you take the current angle of the main line aimage0nd add to it the relative changes of the line you want to add.

As a side note, I've used this technique of converting a line into a set of angle changes for artistic exploitation - you can make drawings "uncurl" similar to how wire can curl and uncurl. Some examples: one, two, three

This angle calculation is what allows us to steer the line as you drag - we calculate how off the main angle is from where we want to be and we look for a picture that will help the most getting the line going in the right direction. It's all a matter of thinking relatively.

Finally, I just want to say that this was a really fun project to be involved with. It's exciting as an artist to be asked to use a data set as lovely as these images and I'm honored the Data Arts team reached out. I hope you have fun experimenting with it!

□□□ #2
□ □□□ □□□ 2026-10-31 21:59:53 UTC
□ □□□ □□□ 2026-10-31 21:59:54 UTC