Augmented Reality Map

Team Members

Ryan Buck

John Cornish

Adam Frick

Roy Procell

Miguel Ruiz

June 19th, 2018

I. Introduction

Client Description

LGS Innovations is an American technology company specializing in Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR); cyberspace operations; and network assurance. LGS provides supporting services to U.S. defense and intelligence agencies and has won numerous awards, including eight Nobel Prizes.

Product Vision

LGS tasked us with creating an “augmented reality” map system. The ultimate goal was to create a visualization tool in which the user could see a video on a map, uploaded or live-streamed, and also be able to view the video’s telemetry data, such as the GPS location, associated with it. This would be represented on the map as a marker moving along a highlighted route such that the GPS location is synchronized with the video feed at all times. LGS wanted this product to be built on a locally hosted website using JavaScript and Mapbox. The original product vision was to use a GoPro Hero 5 Black in order to achieve the video and GPS capabilities required for the project (this was later changed to Android). LGS also requested auxiliary features if time permitted, such as an overlaid heat map displaying WiFi or cell signal strength, 3D buildings, traffic, and the ability to filter data on the map with bounding boxes.

II. Requirements

Functional Requirements

- Create a webpage with a Mapbox rendering on it (similar to Google Maps)
- Stream live video and GPS to website from Android device
- Tie each second of video to a GPS location on map and display corresponding location (moving marker on map with corresponding video in sidebar)
- Highlight route corresponding to current video playing
- Bonus features (if time permits)
    - WiFi heatmap overlay (color-coded signal strength)
    - Transport stream for live video that allows you to scrub through the video and update marker accordingly
    - Render buildings in 3D
    - Filter data with bounding boxes

Non-Functional Requirements

- Intuitive UI
- Scalable
- Project must be entirely offline/local
- Create unique file format for tying video and location data together
- Be able to save and playback live streams
- Buttons to toggle map layers (heatmap, traffic, etc.)

III. System Architecture

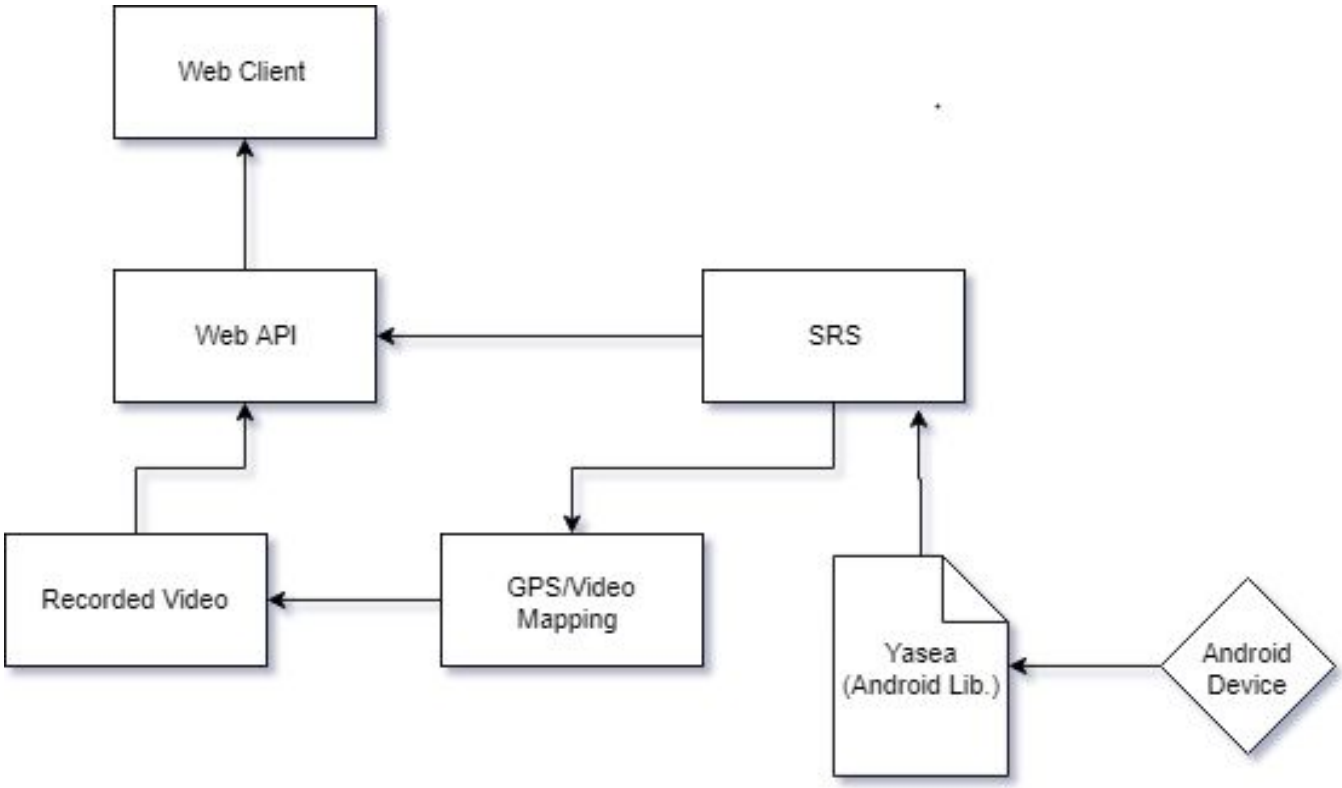


Figure 1: Components of the Augmented Reality Map system

Web Client

The web client is a React.js application running on Node.js. The React application utilizes Mapbox GL JS to display the map and its various features such as the cell data heatmap, the live traffic overlay, 3D buildings, and the markers and paths for pre-recorded and live videos to the user. The React application also displays both the pre-recorded videos and live streams to the user. The web client runs on the localhost port 3000.

Web API

The web API uses the “fast, unopinionated, minimalist framework for Node.js,” aptly called Express. It uses the Model-View-Controller paradigm for application development, and we were originally running our web application on it before we decided to instead use it as an API. It runs on localhost on port 3001. It also uses Server Sent Events to update the web client with coordinates as it retrieves and parses them from the SRS Live Convert node app.

SRS

SRS is an open-source C++ library for providing an RTMP server. The Android application streams video and audio data to the server via RTMP, and SRS provides a transfer protocol to HLS via Nginx, which the web client reads from. It only runs on a Linux operating system, which limits the portability of the project.

GPS/Video Mapping

A Javascript program listens on the recorded video directory for a temporary video file. Its existence indicates that there is a video stream in progress, and starts populating an array of GPS coordinates until the stream stops. When finished, it writes the coordinates in GeoJSON format with the same filename as the video, but with a different extension.

Recorded Video

Video is recorded while the Android device is livestreaming. The videos are given a unique timestamp and are read alongside corresponding GPS data once recorded.

Yasea (Android Library)

Yasea is an open source Android streaming client hosted on GitHub. It is able to capture video and audio data from the data, encode it in a convenient transport stream, and expose it on a network port accessible via ADB. This is aided by the RTMP/HLS server, SRS, which the web client uses to view the livestream.

Android Device

The Android device itself utilizes the Yasea Library to stream video to the client, and we wrote our own code to stream live GPS updates from the device over a different port than the video. No video or GPS data is saved to the Android device; it is all stored on the client machine. The data is streamed over USB via ADB to optimize latency and bandwidth.

IV. Technical Design

Livestreaming with Android

It was technically challenging and nuanced to allow Android to livestream video and GPS coordinates concurrently. A substantial amount of boilerplate is performed by the Yasea library, which the Android application uses as an interface to MainActivity.

The Android app layout is straightforward. It always runs in landscape mode, has a full-screen view of a camera, and has buttons to toggle streaming and cameras. The camera switch button allows either the front or back camera to be used. The stream button begins both video streaming and GPS coordinate streaming, and stops it once pressed again.

There are a few helper functions for MainActivity. `getPermissions()` requests all the hardware that the Android device requires explicit user consent for, such as camera and microphone. `screenSetup()` sets the attributes for the Android screen layout, like setting the screen orientation to landscape. `streamSetup()` configures an instance of SrsPublisher.

`Connection` is a singular instance which keeps a socket open to write bytes to. It is assisted by helper functions located in `GPSLocation` to encapsulate the GPS setup. The socket is opened at `localhost:59900` , and after having pressed Connect on the app, a listener watches for changes in location. The new location is stringified and sent through the output stream for the web client to receive. The GPS/video format expects a coordinate pair every second, so the web client interpolates the received pairs such that each second of video has a respective pair.

The Yasea library is responsible as an RTMP client for sending video and audio data to SRS. The setup for it is minimal, as the library just requires an instance of `SrsPublisher` which can begin running the stream with a function call. The MainActivity class implements several functions from events that the library can invoke, like the stream being stopped or interrupted.

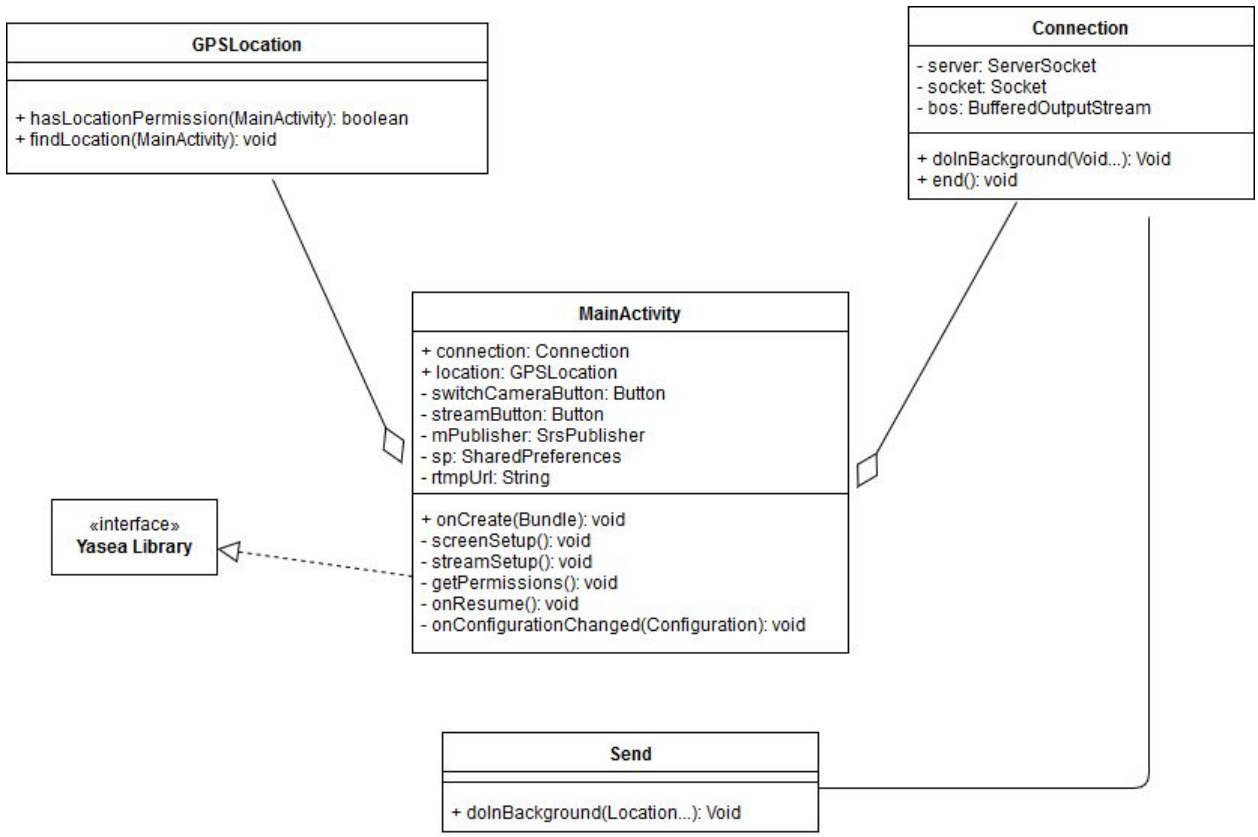


Figure 2: UML of Android streaming app code

Heatmap

The heatmap proved to be much more of a challenge than anticipated. The tool we used to render the heatmap was the Mapbox GL native heatmap tool. The Mapbox GL heatmap tool creates a density-based layer instead of a true gradient as seen with traditional cloreoplath-based heatmaps. This is too say that the weight of the heatmap's color and size for a given area is based upon the amount of cell towers in the area, not a gradient with signal strength.

We collected cell tower data from an open source database called OpenCelliD, which collects cell tower locations and other related data from around the world. We decided to only use data from the United States due to the large volume of data that OpenCelliD had. Additionally, the data collected from OpenCelliD was not in the GeoJSON format that can be read by Mapbox and was instead formatted as a CSV file. An example of the GeoJSON format can be found in Listing 1 below. GeoJSON is formatted as an object called `FeatureCollection`, which contains an array of

objects called `Feature`. Features have certain properties such as the weight called `dbh`, and geometry. The Feature's geometry consists of the coordinates and the shape of the Feature, such as `Point` or `Line`. We wrote a small program in C++ to convert the OpenCelliD data to the GeoJSON format, filtering out unnecessary data and only keeping data such as the latitude, longitude, and the range of the cell tower. The code for this program is listed in the Appendices.

```
{"type":"FeatureCollection","features":[{"type":"Feature","prope
rties":{"dbh":5},"geometry":{"type":"Point","coordinates":[-80.0
0118,40.45599]}},{"type":"Feature","properties":{"dbh":3},"geome
try":{"type":"Point","coordinates":[-79.95948,40.46403]}}]}
```

Listing 1: Example of the GeoJSON format

Due to the verbosity of the GeoJSON format and the more than four million points in the OpenCelliD data, the resulting GeoJSON file was too large for Mapbox to load and display in a reasonable amount of time. As a result, we decided to truncate the data to the point where we had a good compromise between reasonable loading time and appropriate amount of data displayed. Although we significantly truncated the data displayed, much of the four million points in the original data consisted of redundant data, so the effect of truncating the data was negligible.

After we collected and formatted our data to the GeoJSON format, we created our heatmap. We utilized Mapbox's add source tool to tell Mapbox where to look for our data and then create our heatmap layer. Since we couldn't get a true gradient based on cell signal strength across distance, we wanted to create a heatmap that would emulate that as closely as possible. We achieved this by lowering the effect that true density has on the heatmap. As you can see below in Figure 3, most of the map is covered in green here. Areas with a high number of cell towers will have green coloring, and red to no color have little to none cell towers compared to the other regions. If density was a major factor in the heatmap, we would have much more yellow and red. However, just because there is only one cell tower in that map's zoom location, doesn't mean that the signal is weak or shouldn't be represented. Thus, we have a more coverage-based map rather than a signal strength heatmap. However, it isn't a perfect fit. You can see below that Idaho and some states surrounding it appear to have no coverage, but this is because the data there is so sparse and will be represented on a closer zoom into those

areas. Through the use of the Mapbox's add layer functionality, we have created a fairly interactive and accurate cell coverage heatmap.

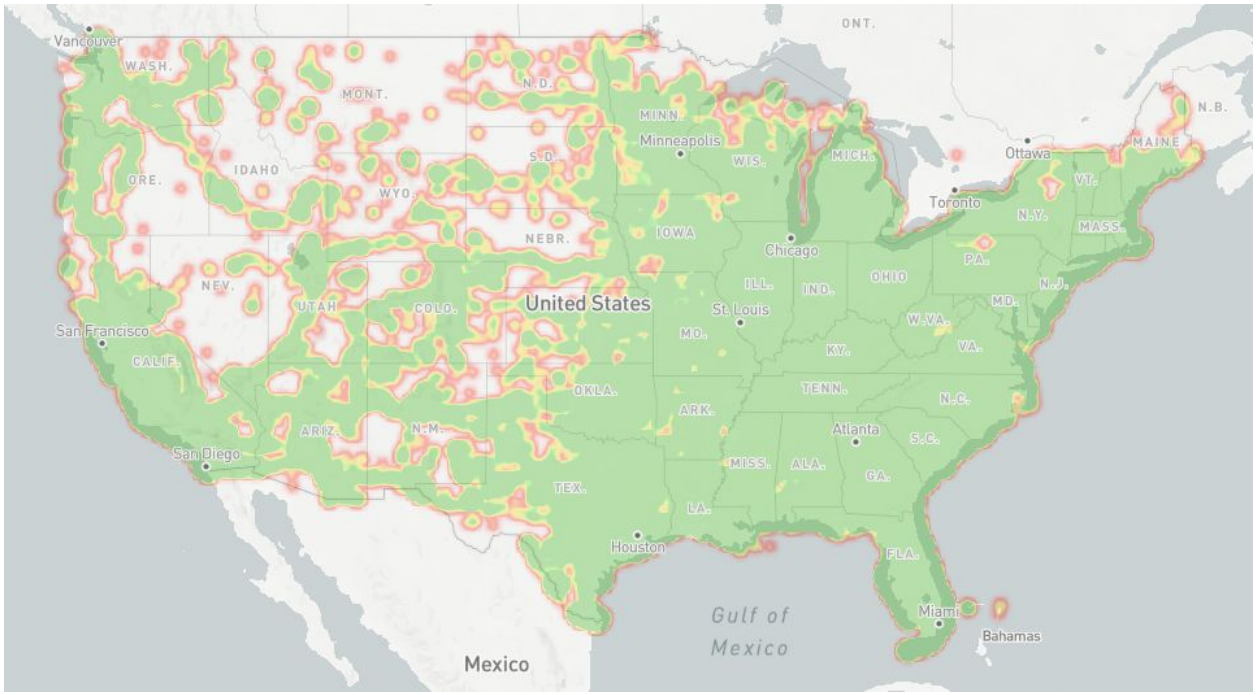


Figure 3: Heatmap showing cell tower locations in the United States

V. Decisions

Platform

We were initially tasked with doing this project with the GoPro as our source for both livestream and prerecorded video. However, after we had completed the pre-recorded video portion of the project, we found out that the GoPro is simply incapable of being used to complete the livestream portion. This was due to the fact that the GoPro could only pull telemetry data when recording and not live streaming. We then decided to switch to the Android platform, which we definitively knew could provide live location updates and shouldn't have any problems live streaming video.

Website structure

There are many ways to structure our website that we considered. Ultimately we decided that we wanted a single page application (SPA) like website where the user could interact in some way with the displayed data. We boiled the list down to Angular and React. Angular is a much more fleshed out tool that is a whole framework to create websites. React is more of a library and much more flexible and lightweight. We decided

to use React because it gave us all the functionality required while interfacing with Mapbox well and requiring a less intensive solution.

Video playback

We initially used Youtube for our video playback. However, in order to use it we needed to be connected to the internet and we were confined to the Youtube API. Ultimately, we decided on using the built-in HTML5 player because it gave us all the functionality of Youtube but with much more implementation flexibility.

Heatmap

Many tools were considered for the creation of the heatmap layer, like ggplot and the native Mapbox tool. We originally wanted to use GGplot to create an image to overlay on top of the map. This would allow us to save a lot of memory that the Mapbox heatmap tool uses, but ultimately wasn't feasible due to the fact that the resulting image could not reliably scale to the image that Mapbox creates. Therefore, we defaulted to the Mapbox tool.

Mapbox

There were two different versions of Mapbox that were considered. The first one was Mapbox.js and the other was Mapbox GL JS. We chose to use Mapbox GL JS because it allowed for easy graphics manipulation and had access to the Mapbox heatmap tool.

VI. Results

Our implementation meets all of the client's hard requirements and most of the bonus requirements, but we were not able to implement the bounding box bonus feature due to time constraints. The project was tested on Chrome and Firefox, but does not work on Internet Explorer. Due to technical limitations in the GoPro, we switched to Android for the livestreaming piece of the project, with the approval of the client. The Android portion was tested on Samsung Galaxy S5 and S7 devices with no issues--we would have liked to have tested on more devices but these were the only ones available to us. Use of a Linux OS is also required to run the project. Extensions to the project would consist of the bonus features we were unable to implement as well as finding and utilizing additional libraries to allow the project to run on other operating systems like Windows. This includes the ability to filter data by bounding boxes on the map, as well

as adding display markers as arrows facing the direction they are traveling. This project taught the team to investigate open-source code more thoroughly before trying to integrate it. On both the Android and Mapbox sides, we went through several GitHub projects/libraries trying to find one that would be useful for our purposes.

VII. Appendices

Yasea Android Library

https://github.com/begeekmyfriend/yasea

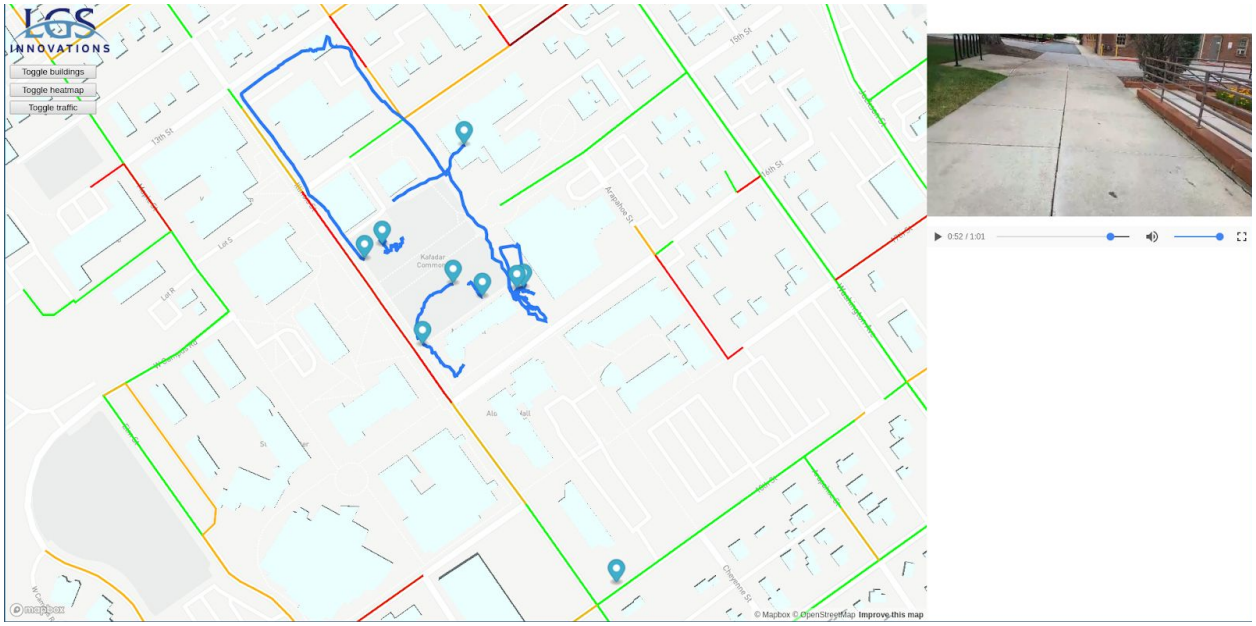
GeoJSON Converter

```
}/*
*		title: GeoJSONConverter
*		authors: Ryan Buck and Roy Procell
*		project: LGS field Session
*/
#include<fstream>
#include<iostream>
#include<cstdlib>
#include<string>
#include<vector>
#include<sstream>

using namespace std;

int main() {

		ifstream in("C:\\Scratch\\310_filtered.csv");
		if (in.fail()) {
			cerr << "Problem with input stream" << endl;
		}

		ofstream out("C:\\Scratch\\heatmap.geojson");

		string line;


		getline(in, line);
		out << "{\"type\":\"FeatureCollection\",\"features\":[" << endl;

		string newLine;
		string token;
		double areaTok;
		string dbh;
		string lon;
		string lat;

		while (!in.eof()) {
			newLine = string();
			token = string();

			getline(in, line);
```

```
            istringstream iss(line);

            //{"type":"Feature","properties":{"dbh":0},"geometry":{"type":"Point","coordinates":[-79.91746,40.44356]}

            newLine = "{\"type\":\"Feature\",\"properties\":{\"dbh\":";          //GeoJSON Feature header
            getline(iss, token, ',');               //RADIO
            getline(iss, lon, ',');                 //LON
            getline(iss, lat, ',');                 // LAT
            getline(iss, dbh, ',');                 // DBH
            newLine = newLine + dbh + "},\"geometry\":{\"type\":\"Point\",\"coordinates\":[" + lon + "," + lat + "]}},";   // Data

            //gets rid of comma in case this element is the last one
            if (in.eof()) {
                        newLine.pop_back();
            }

            out << newLine << endl; //outputs newLine

}

out << "]}"; //closes feature Collection
```

Screenshot of the Android Application

Screenshot of the Web Client


INNOVATIONS
Toggle buildings
Toggle heatmap
Toggle traffic
13th St
15th St
Arapahoe St
Cheyenne St
0:52 / 1:01
© Mapbox © OpenStreetMap Improve this map