Toward Usable Programming Systems for Geospatial Analysis and Visualization

Parker Ziegler

Toward Usable Programming Systems for Geospatial Analysis and Visualization

by

Parker Ziegler

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Assistant Professor Sarah E. Chasins, Chair
Professor Marti A. Hearst
Associate Professor Aditya Parameswaran

Fall 2023

The thesis of Parker Ziegler, titled Toward Usable Programming Systems for Geospatial Analysis and Visualization, is approved:

Chair	Sarah E. Chasins	Date	12/6/23
	[illegible]	Date	12/6/2023
	[illegible]	Date	11/27/23

University of California, Berkeley

Toward Usable Programming Systems for Geospatial Analysis and Visualization

Abstract

Toward Usable Programming Systems for Geospatial Analysis and Visualization

by

Parker Ziegler

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Assistant Professor Sarah E. Chasins, Chair

Geospatial data is playing an increasingly critical role in the work of Earth and climate scientists, social scientists, and data journalists exploring spatiotemporal change in our environment and societies. However, existing software and programming tools for geospatial analysis and visualization are challenging to learn and difficult to use. The aim of this thesis is to identify—and begin addressing—the unmet computing needs of the diverse and expanding community of geospatial data users. Toward this goal, this thesis makes four contributions. First, I conducted a contextual inquiry study ($n = 25$) with domain experts using geospatial data in their current work. Second, I performed a thematic analysis of the contextual interviews, finding that participants struggled to (1) find and transform geospatial data to satisfy spatiotemporal constraints, (2) understand the behavior of geospatial operators, (3) track geospatial data provenance, and (4) explore the cartographic design space. Third, I used these findings to synthesize a set of design opportunities for developers and designers of geospatial analysis and visualization systems. Fourth, I put these design opportunities into practice in `cartokit`, a new direct manipulation programming environment for interactive cartography on the web. Cumulatively, this work presents a novel vision for what useful, usable programming systems for geospatial analysis and visualization could look like.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I want to thank my advisor, Professor Sarah E. Chasins, for her expertise, mentorship, and relentless, unwavering support of my research. Her contributions to this work cannot be overstated. I also want to thank my anonymous participants for sharing their working practices with geospatial data—this research would not have been possible without them. At Berkeley, the folks I am indebted to are almost too numerous to thank. Rachel Leven at the Center for Computing, Data Science, and Society was instrumental in connecting me with data journalists for this research. The members and friends of PLAIT Lab, including David Cao, Serena Caraco, Hellina Hailu Nigatu, Eunice Jun, Slim Lim, Justin Lubin, Gabriel Matute, Mae Milano, Eric Rawn, Lisa Rennels, J.D. Zamfirescu-Pereira, and many others, have been an everlasting source of amazing ideas, incisive feedback, and support during the hardest days. And of course, the broader community of the EPIC Data Lab, the Programming Systems group, and friends spread across the Department of Electrical Engineering and Computer Sciences, who have always made me feel valued and welcomed. Beyond Berkeley, the folks who have believed in me, supported me, and guided me are unquestionably too numerous to mention. But within that, I want to thank my parents Christine and John, who have fostered my love of learning always; my brother, Quinn, and sister-in-law, Carey, who have been some of my most trusted mentors and confidants; my parents-in-law, Lauren and Bob, whose advice, perspective, and presence have shaped me profoundly; my sister-in-law Ella and brother-in-law Cesar, whose ambition and drive are sources of inspiration; and most of all my partner, Bess, whose love, support, intelligence, and compassion have been my guiding light.

Chapter 1

Introduction

Geospatial data—data encoding the location and attributes of phenomena on the Earth's surface [72]—is growing in scale and accessibility at a tremendous rate [74]. Researchers estimate that Earth observation satellites generate 80TB of new imagery daily [103]. Closer to the surface, cheap, power-efficient sensors create massive volumes of geolocated data measuring real-time environmental change [52]. Additionally, crowdsourcing efforts like OpenStreetMap have fostered an explosion in publicly available volunteered geographic information [96, 62]. Geospatial data has long played a fundamental role in the research of geographers and cartographers. As this data becomes more available, experts across a widening array of domains are turning to geospatial analysis and visualization to address challenges in climate change [21], public health [42], school segregation [102], hazard modeling [121], and other areas.

Despite this expansion in the community of geospatial data users, research has yet to explore the specific challenges domain experts face in gathering, analyzing, and visualizing geographic information. Many domain experts are self-taught in the theory of geospatial data and the specialized Geographic Information System software used to manipulate it. Human-Computer Interaction (HCI) researchers have found that non-geographers struggle to use these systems because they require familiarity with concepts and terminology from geography [55]. Some of these users have turned to programming as an alternative. While geospatial libraries are increasingly common in Python, R, and JavaScript, domain experts must develop proficiency in at least one of these general-purpose languages to benefit from these abstractions.

My research aims to investigate—and begin addressing—the computing needs of the growing community of geospatial data users. Answering calls from HCI researchers for increased collaboration with geography [59, 60], I conducted a contextual inquiry study with 25 geospatial data users from academia, industry, newsrooms, and the public sector. Thematic analysis of observations and semi-structured interviews revealed common challenges across five phases of participants' work with geospatial data: data discovery, data transformation, analysis, analysis representation, and visualization. I observed that participants had difficulty (1) finding and transforming geospatial data to satisfy complex sets of spatiotemporal constraints, (2) understanding the behavior of geospatial operators, (3) tracking geospatial data provenance, and (4) efficiently exploring the cartographic design space, among other challenges. My findings deepen our understanding of requirements for supporting domain experts in their work with geospatial data and suggest design opportunities for geospatial analysis and visualization systems.

Building off of these design opportunities, I next developed `cartokit`, a direct manipulation programming environment for interactive cartography on the web. `cartokit` aims to simplify the process of programming interactive maps by transforming sequences of interactions with geospatial data in a graphical user interface (GUI) into JavaScript programs. Toward this goal, `cartokit` makes three key technical contributions. First, it exposes direct manipulation interactions for transforming and styling geospatial data while preserving the underlying data representation, supporting dynamic movement between analysis and visualization work in a single system. Second, it enables single-click transitions between map types to support rapid exploration of the cartographic design space. Third, it compiles user-created maps to JavaScript programs, providing direct access to a reproducible program artifact that can be copied, modified, and deployed in other code-based environments. In essence, `cartokit` reimagines the core interaction paradigm used to design maps today—direct manipulation—as a form of programming.

In summary, this thesis makes the following contributions:

- A contextual inquiry study of 25 geospatial data users to understand their computing needs
- A thematic analysis of challenges participants faced across distinct phases of their work with geospatial data
- A set of design opportunities for geospatial analysis and visualization systems
- A novel direct manipulation programming environment for interactive cartography on the web

Chapter 2

Background

2.1 Geospatial Data

Geospatial data describes the location and attributes of phenomena on the Earth's surface [111]. It differs from tabular data in that it links geometric representations of real-world geographies—referred to as the *geometry* of the data—with attributes of those geographies [72]. In this way, geospatial data connects information to place.

There are two models of geospatial data, distinguished by their geometric representations (Figure 2.1):

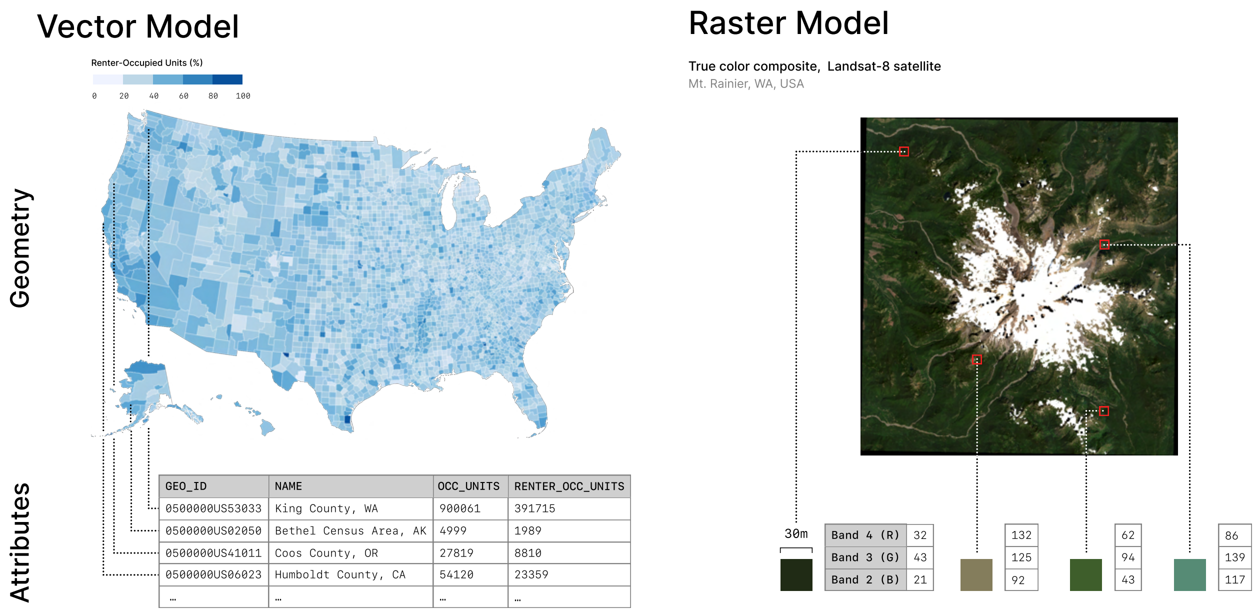
1. The **vector model** represents geographic features as points, lines, and polygons, connecting tabular data to features via an attribute table. For example, the U.S. Census Bureau's American Community Survey connects demographic estimates to geographic areas (e.g., counties) modeled as polygons [13].

2. The **raster model** partitions geographic space into a pixel grid. Each pixel corresponds to a portion of the Earth's surface depending on the spatial resolution of the raster. For example, the Landsat-8 satellite collects data at 30m spatial resolution, meaning each pixel in the raster represents a 30 x 30m area [104]. The value associated with a raster pixel corresponds to the data attribute at that location.

2.2 Geographic Information Systems vs. Programming Environments

Geographic Information Systems

A Geographic Information System (GIS) is a software system for "capturing, storing, querying, analyzing, and displaying geospatial data" [17]. GISs represent geospatial datasets as layers, which can be edited, combined, and analyzed to generate new layers using built-in



GEO_ID	NAME	OCC_UNITS	RENTER_OCC_UNITS
0500000US53033	King County, WA	900061	391715
0500000US02050	Bethel Census Area, AK	4999	1989
0500000US41011	Coos County, OR	27819	8810
0500000US06023	Humboldt County, CA	54120	23359
…	…	…	…

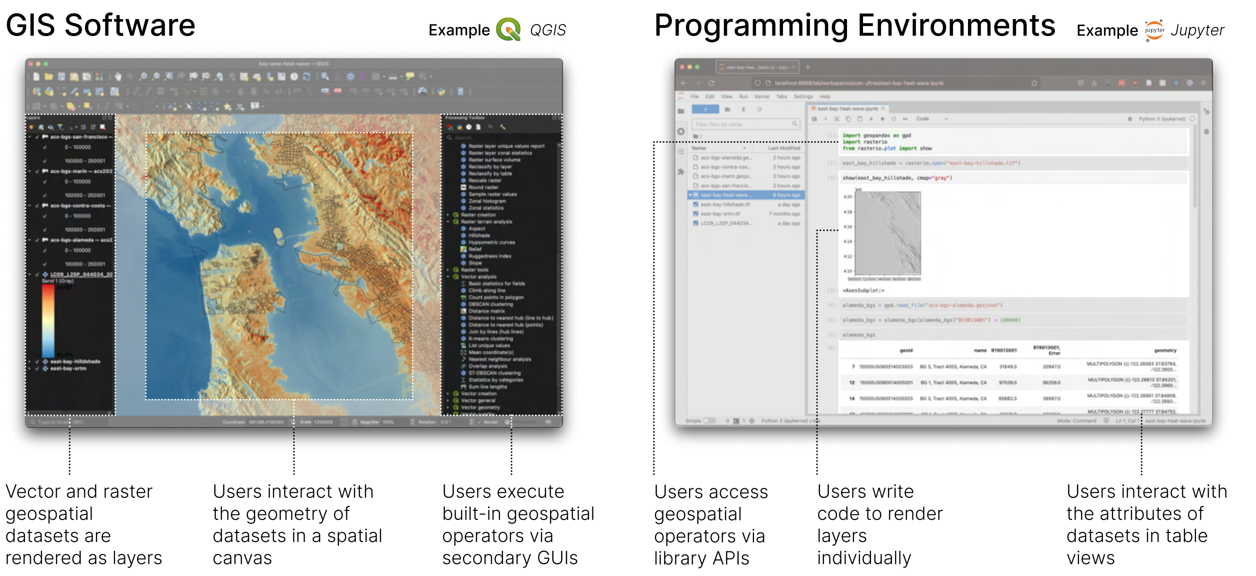
Band 4 (R)	32	132	62	86
Band 3 (G)	43	125	94	139
Band 2 (B)	21	92	43	117

Figure 2.1: **Geospatial data models.** The vector model represents geographies as points, lines, and polygons. Geographies are attached to tabular data via an attribute table. For example, in the choropleth map (left), U.S. counties encoded as polygons are associated with housing data from the U.S. Census Bureau's 2020 American Community Survey. The raster model partitions space into a pixel grid. Each pixel has an attached value corresponding to the data attribute at that location. For example, in the Landsat-8 satellite image of Mt. Rainier (right), each pixel is associated with an RGB value measuring light reflected off the Earth's surface.

geospatial operators accessed via GUIs. Users visualize and interact with layers in a spatial canvas that allows them to zoom, pan, style, and select geographic features directly. In this way, GISs center interaction with the geometry of geospatial datasets. Interaction with attributes happens in secondary table views where users write SQL to query and manipulate their data. Many GISs exist; my participants used ArcGIS [39] and QGIS [4].

Programming Environments

In contrast to GIS software, programming environments used to work with geospatial data center interaction with the attributes of the data rendered as tables or dataframes. This is especially true of computational notebooks like Jupyter notebooks [97], R Markdown [105], and Observable [92], which have been adopted by geospatial data users but are not purpose-built for geospatial data. In these environments, users write code to visualize and interact with the geometry of their data. Rather than executing geospatial operators via GUIs, they rely on APIs from geospatial analysis and visualization libraries. Newer programming


GIS Software
Example QGIS
Programming Environments
Example Jupyter
Vector and raster geospatial datasets are rendered as layers
Users interact with the geometry of datasets in a spatial canvas
Users execute built-in geospatial operators via secondary GUIs
Users access geospatial operators via library APIs
Users write code to render layers individually
Users interact with the attributes of datasets in table views

```
[1]: import geopandas as gpd
     import rasterio
     from rasterio.plot import show
[2]: east_bay_hillshade = rasterio.open("east-bay-hillshade.tif")
[3]: show(east_bay_hillshade, cmap="gray")
[3]: <AxesSubplot:>
[4]: alameda_bgs = gpd.read_file("acs-bgs-alameda.geojson")
[5]: alameda_bgs = alameda_bgs[alameda_bgs["B19013001"] < 100000]
[6]: alameda_bgs
```

	geoid	name	B19013001	B19013001, Error	geometry
7	15000US060014003003	BG 3, Tract 4003, Alameda, CA	31649.0	20947.0	MULTIPOLYGON (((-122.26563 37.83764, -122.2655...
12	15000US060014005001	BG 1, Tract 4005, Alameda, CA	97039.0	36259.0	MULTIPOLYGON (((-122.26613 37.85201, -122.2660...
14	15000US060014005003	BG 3, Tract 4005, Alameda, CA	85662.0	39567.0	MULTIPOLYGON (((-122.26951 37.84858, -122.2693...

Figure 2.2: **Examples of GIS software and programming environments for working with geospatial data.** The QGIS project (left) and Jupyter notebook (right) contain the same geospatial data, but users interact with this data differently in each tool.

environments like Google Earth Engine [53] and Microsoft Planetary Computer [85] mix features from both GISs and computational notebooks but are designed for particular forms of geospatial analysis (e.g., remote sensing).

Chapter 3

Related Work

This section surveys findings from observational studies of geospatial data users, empirical evaluations of GIS usability, and studies exploring the needs of data scientists more generally.

3.1 Observational Studies of Geospatial Data Users

Prior observational studies of geospatial data users have focused on identifying GIS usability issues [29, 31, 116, 115]. My work is most similar to a workplace study of 21 GIS practitioners, which used video recordings, semi-structured interviews, and usability checklists to uncover recurrent participant challenges [31]. The insights centered around error states, finding that GISs failed to prevent common user errors, surfaced errors in difficult-to-understand language, and provided insufficient guidance for correcting errors. Additionally, they observed that non-expert GIS users relied on a "local expert" to perform their analysis, also reported in [49, 36]. My study differs in two ways. First, I investigate how users interact with geospatial data across tools other than GISs, including computational notebooks, design software, and geospatial analysis and visualization libraries. In fact, most participants $(13/25, 52\%)$ did not use GIS software. Second, while [31] observed data transformation and analysis, I identified additional challenges related to data discovery, analysis representation, and visualization.

Another closely related study observed non-expert GIS users (social science faculty and computer science graduate students) and identified data provenance tracking as a common struggle [115]. Participants' GISs maintained no record of how outputs were generated, making it difficult to reproduce past analyses. Additionally, modifying or retargeting existing maps at new data entailed reverse engineering the original analysis through trial and error. My study extends our understanding of provenance needs by (1) identifying frustrations with provenance features in modern GISs and (2) describing participants' informal methods for tracking provenance and reproducing past analyses.

3.2 Evaluating GIS Usability

Several studies have evaluated GIS usability using non-observational qualitative methods, including expert task analysis [116], user surveys [30], interviews [37], and screenshot analysis [56]. A task analysis of seven GISs concluded that GIS software is challenging to use because it (1) requires users to understand concepts from multiple disciplines, including geography, cartography, statistics, and databases, and (2) relies on domain-specific vocabulary and concepts (e.g., "overlay," "thematic layer") that reflect the system architecture rather than a user's view of their work [116]. A survey of 159 GIS users found respondents had difficulty understanding and fixing errors, customizing the interface via provided macro languages, and finding sufficient documentation to use GISs [30].

Other studies have employed quantitative methods such as interaction logging [43, 118], controlled experiments [77, 98], and eye-tracking [78, 82] to evaluate particular GIS interfaces. A controlled experiment compared five interaction techniques for cross-layer comparison and correlation [77]. Fechner and colleagues logged interface interactions in a web-based GIS to understand how users collaboratively create and edit geospatial datasets [43]. Unrau and Kray provide a comprehensive survey of studies assessing the usability of different GISs [117]. Rather than evaluating specific GIS interfaces, my study focuses on challenges across various tools.

3.3 Needs of Data Scientists

Research on the needs of data scientists has identified struggles with wrangling and aligning data from multiple sources [89, 34], iterating on and maintaining analysis versions [67, 68], and editing data collaboratively [70]. Data transformation and preparation have consistently emerged as the most challenging phases of data scientists' work [65, 54]; practitioners must develop domain-specific knowledge to identify patterns and anomalies in their datasets, handle missing values, and combine data from differing sources and temporal paradigms [89]. For geospatial data, ensuring that datasets cover the target area and time range of analysis is essential [70]. Beyond data transformation, interviews, surveys, and formative studies have revealed data scientists struggle to track iterations of their analyses, often relying on informal versioning techniques [67, 68]. I examine both challenges—data transformation and version management—in the special case of geospatial data, highlighting areas of overlap and divergence with prior work on data science more broadly.

Chapter 4

Method

To understand the challenges facing geospatial data users, I conducted a contextual inquiry [64] study with 25 participants from academia, industry, newsrooms, and the public sector using geospatial data in their current work.

4.1 Participants and Recruitment

I recruited participants via social media (Twitter, Meetup, Reddit, Slack), direct outreach to academic departments, and the authors' networks. I used a screening survey to select participants from multiple domains—including Earth and climate science, the social sciences, and data journalism—with varying years of prior experience working with geospatial data (Figure 4.1). Our aim with this design was to observe a wide range of user challenges and identify those that recurred across a varied group, revealing needs that transcend domain and expertise boundaries. However, this study design favors breadth at the cost of depth; by prioritizing participant diversity, I may have missed details of challenges that arise only for experts, non-experts, or users in a particular domain. Additionally, recruiting from social media and personal networks runs the risk of creating a more homogeneous participant pool that may not represent the broader community of geospatial data users. Table 4.1 provides information about our participants.

4.2 Consent and Compensation

Before participating in the study, participants signed a consent form in accordance with our institutional review board. Participants received compensation in the form of a \$40 gift card or a \$40 donation to a 501(c)(3) organization of their choice.

ID	Exp.	Domain	Languages	Tools
PE1	1-3	Earth and Climate Science	JavaScript	Google Earth Engine
PE2	3-5	Earth and Climate Science	R, Python	Google Earth Engine
PE3	<1	Earth and Climate Science	—	QGIS
PE4	1-3	Earth and Climate Science	Python	Google Earth Engine, Jupyter, `geemap`
PE5	<1	Earth and Climate Science	Python	Jupyter, Google My Maps, Leaflet
PE6	5-10	Earth and Climate Science	—	ArcGIS
PE7	>10	Earth and Climate Science	—	QGIS
PE8	3-5	Earth and Climate Science	Matlab	—
PE9	5-10	Earth and Climate Science	Python	Jupyter, `geopandas`
PE10	1-3	Earth and Climate Science	Python	Jupyter, `geopandas`
PS1	1-3	Social Science	—	QGIS, Adobe Illustrator
PS2	>10	Social Science	—	QGIS, Adobe Illustrator
PS3	<1	Social Science	Python	QGIS, Jupyter, `geopandas`
PS4	1-3	Social Science	R	R Markdown, `sf`
PS5	5-10	Social Science	—	ArcGIS
PJ1	1-3	Data Journalism	R	R Markdown, Leaflet
PJ2	<1	Data Journalism	—	QGIS, VisiData
PJ3	5-10	Data Journalism	JavaScript	Observable, D3
PJ4	>10	Data Journalism	Python	Jupyter, `geopandas`
PJ5	5-10	Data Journalism	JavaScript, CSS	QGIS, Adobe Illustrator, Adobe Photoshop, D3
PJ6	1-3	Data Journalism	JavaScript	QGIS, Mapbox
PJ7	3-5	Data Journalism	Python	Jupyter, Microsoft Excel, Tableau
PJ8	1-3	Data Journalism	Python	QGIS, Jupyter, `geopandas`
PO1	3-5	Other (Finance)	—	ArcGIS
PO2	5-10	Other (Computer Science)	Python	IPython, `geopandas`

Table 4.1: **Participant characteristics.** Throughout the rest of the paper, I use the participant IDs in the ID column to refer to individual participants. Exp. refers to participants' prior experience working with geospatial data, in years.

4.3 Session Structure

Each study session consisted of a 50-70 minute observation followed by a 15-20 minute semi-structured interview. I conducted sessions remotely over Zoom and recorded them for subsequent analysis. One participant opted out of recording; I analyzed their session via

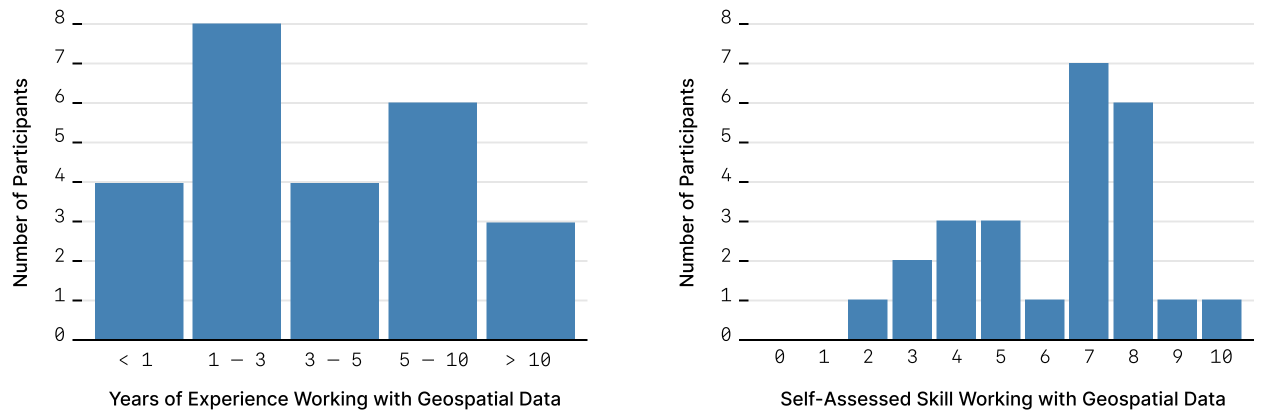


Figure 4.1: **Participant experience and skill.** Participants reported their years of prior experience working with geospatial data (left) and their self-assessed skill level working with geospatial data on a scale of 1-10 (right).

written notes. During observation, I asked participants to share their screen and narrate their thought processes as they worked on a task of their choice related to gathering, analyzing, or visualizing geospatial data. I intentionally left the choice of task open for two reasons:

1. **Faithfulness to participants' work.** I aimed to study the challenges participants face in their everyday work with geospatial data. Researcher-designed tasks may not elicit the challenges they typically encounter.

2. **Experience, domain, and tool diversity.** Participants varied widely in their prior experience working with geospatial data, their domain of expertise, and the software and programming environments they use to work with geospatial data. Researcher-designed tasks might lead us to identify erroneous needs that are artifacts of task design.

While the tasks I observed were more representative of participants' actual work than researcher-designed tasks, our study design does not give us insight into how representative they are of the broader community of geospatial data users. Assessing the prevalence of our participants' challenges will require further study.

During semi-structured interviews, I discussed specific observations from the session to confirm or refine our interpretations of participant actions.

4.4 Data Analysis

I conducted an inductive thematic analysis [10] of video recordings of the observations and semi-structured interviews using MaxQDA [109]. I started with an open coding phase in which I associated short, descriptive sentences of participant behaviors with segments of the video recordings. I then grouped these open codes into a hierarchy of axial codes and, eventually, top-level themes. I met weekly with collaborators to refine the code hierarchy, splitting and merging open and axial codes based on discussion. I analyzed 29 hours of footage from 24 sessions and written notes from one unrecorded session.

Chapter 5

Findings

I organize my findings into five sections corresponding to distinct phases of participants' work with geospatial data: data discovery, data transformation, analysis, analysis representation, and visualization.

5.1 Finding Geospatial Data

Participants struggled to find geospatial data satisfying a complex set of spatial and temporal constraints derived from their analysis goals (PE1, PE2, PE3, PE4, PS4, PJ1, PJ2, PJ6, PO2). The most common constraint required that a dataset cover a specific geographic area (PE1, PE3, PE4, PS4, PJ1, PJ6, PJ7). However, it was rare for participants to find existing datasets tailored to their analysis regions. More often, they reduced datasets collected for larger geographic extents by clipping them to their study areas (PE3, PJ7) or filtering out features based on attribute values (PE4, PE8, PE10, PS4). For example, PE3 derived their dataset by clipping a global soil region dataset to their study area and filtering the remaining features by a soil type attribute. In other cases, data for an analysis region was spread across multiple sources and had to be combined manually (PJ1, PJ6). PJ6 traversed 45 pages of the California Air Resource Board's website to obtain the air monitoring boundaries for 15 communities in their analysis region, which they then composed into a single layer. These findings are consistent with prior observations that geographic coverage affects dataset selection [70] and that analysts combine datasets from disparate sources to meet analysis requirements [89, 44].

Some constraints were related to geographic accuracy, which occasionally varied across the analysis region. Accuracy inconsistencies were especially pronounced in crowdsourced geospatial datasets like OpenStreetMap (PE3, PS2, PO2). PO2 explained that in poorly-surveyed areas, "you'll get weird things where building footprints don't fall within block boundaries, or you'll have weird self-intersections ... that don't semantically or geographically make sense." These issues were difficult to detect before analysis began due to the size and detail of participants' datasets. Some manually inspected their data to identify and correct topological errors preemptively (PE6, PO1), while others compared their data to satellite imagery (PJ7) or Google Street View images (PJ1) to corroborate its accuracy.

For Earth and climate scientists, constraints on spatial resolution, temporal resolution, and occlusion characteristics of satellite imagery were critical—though difficult—to satisfy (PE1, PE2, PE4, PE8). For example, PE1's analysis of drought patterns in Chile required them to filter Sentinel-2 [35] satellite images of their study area to those captured during the dry season over a six-year period. Occlusions like clouds, mist, and shadows skewed the analysis, prompting them to implement additional image manipulation algorithms to mask the affected pixels.

Participants had additional constraints related to:

- *Cost* (PE2, PE3) – Participants could only use freely-available data.
- *File Format* (PE3, PJ7) – Participants needed data in formats readable by their analysis tool.
- *Programmatic Access* (PE4, PS4) – Participants wanted to query and access data via APIs.

5.2 Transforming Geospatial Data

Transforming geospatial data was especially challenging for participants, with a plurality (12/25, 48%) reporting this phase most difficult (Figure 5.1). As PE6 noted, "The data doesn't come all nice, neat, and packaged ... The analysis process [can be] pretty thin and bare compared to the preprocessing."

Aligning Geospatial Datasets

Participants needed to align datasets of differing spatial extents, spatial resolutions, temporal resolutions, and areal units to a shared spatial and temporal reference (PE2, PE10, PJ3, PJ6, PJ8). This often required multiple transformations, including resampling, clipping, and spatial and temporal aggregations. For example, PE2's groundwater prediction model used a combination of MOD16 global evapotranspiration data [106] (8-day, 500m), PRISM climate data [27] (monthly, 4km), and USDA-NASS land cover data [119] (yearly, 30m). To align these datasets to a shared spatial and temporal reference, they implemented (1) a resampling algorithm to transform rasters at finer spatial resolutions (30m, 500m) to the coarsest resolution (4km) and (2) an algorithm to accumulate data collected at finer temporal resolutions (8-day, monthly) to the coarsest resolution (yearly). Similarly, PE10 aligned observations from NASA's GRACE satellite to predictions from a land surface model. They created two "masks" in the form of `geopandas` [50] `GeoDataFrames` to filter model predictions to the geographic locations and timestamps for which they had ground truth GRACE observations. In some cases, participants could not align datasets without making approximations.

Figure 5.1: **Participant responses to the question, “What part of your work or research with geospatial data feels most difficult?”** A plurality of participants (12/25, 48%) selected data transformation, while 28% (7/25) selected analysis.

Topological Errors

Participants spent significant time correcting the topology of their datasets (PE6, PE7, PS2, PO1, PO2). Topological errors refer to violations of geometric invariants such as unclosed polygons, overlapping adjacent polygons, or gaps between adjacent polygons. Most participants identified topological errors through time-consuming visual inspection in GISs or `matplotlib` figures (PE7, PS2, PO1, PO2). PE6 used automated tools in ArcGIS to find topological errors but explained that fixing these errors required manual intervention.

Reducing Resolution to Improve Performance

Participants faced a trade-off between using data with high geographic precision and spatial resolution and being able to analyze data efficiently (PE3, PJ3, PJ4, PJ5). Greater precision and resolution require more space to encode and, in turn, more compute to process. PE3 balanced this trade-off by using a coarser resolution version of their dataset ($\approx$340km) while iterating on an analysis, even though a higher resolution version ($\approx$1km) was available. This allowed them to experiment with multiple analysis approaches without incurring the performance penalty of processing higher resolution data: “I’m doing it at the lowest definition to just run through the workflow first so I know what I’m doing. I’m probably going to pick a higher definition later.” For journalists developing maps for the web, reducing geographic precision minimized the amount of data loaded over users’ network connections (PJ3, PJ4, PJ5). PJ3 and PJ4 used MapShaper [9] to simplify the geometry of their vector datasets;

in PJ3's case, simplification yielded a 98% decrease in the size of their GeoJSON file.

Data Subsetting and Caching

Participants' datasets were often so large that even analysis and visualization tools purpose-built for this data lagged. "Just waiting for all this to ... [render]" (PO2) was a common refrain among participants using both GISs and computational notebooks. PE9, who used `geopandas` to analyze a 3-million point dataset in a Jupyter notebook, waited 50 seconds for a `within` operation to run. PJ2 ran an Overlap Analysis in QGIS between Census block groups and a collection of 2-mile buffers that took six minutes to complete; a previous run using 10-mile buffers "took like two hours." Prior studies have observed GIS users' frustrations with system performance [30, 31], but I found participants using programming environments shared these frustrations.

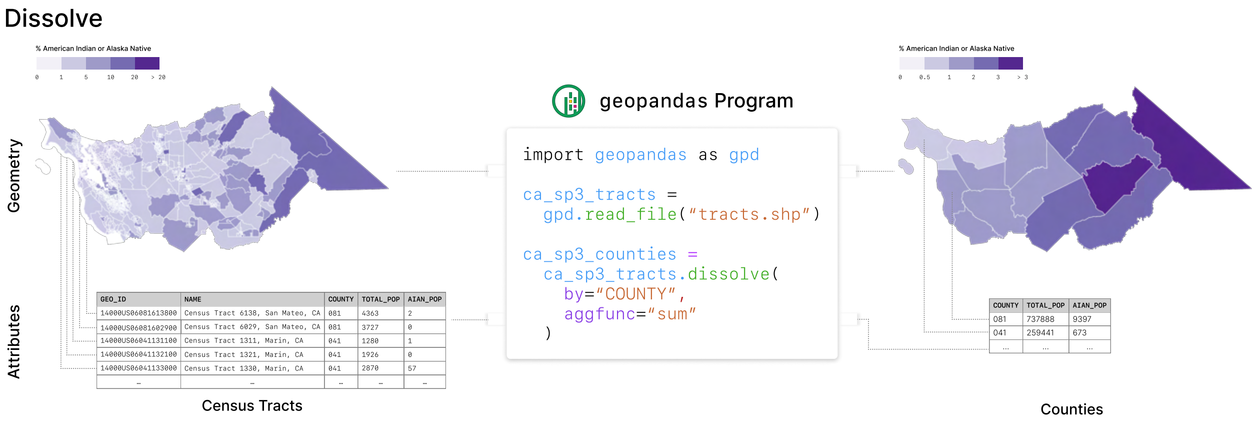
Some participants accelerated the analysis feedback loop by subsetting data by spatial extent (PO1, PO2). For example, while investigating a bug, PO2 filtered their dataset to features within a subarea of their analysis region. This reduced `matplotlib`'s rendering time from five minutes to one second, allowing them to iterate quickly on a fix. However, they cautioned that this strategy could silence errors when applying the modified code to the full dataset: "We'll subset the entire data universe we're trying to work with and start developing what we think is a generalized tool. And then once we run it on the large universe, we'll find weird inconsistencies and bugs."

Participants also used past outputs as "waypoints" from which they could rerun individual transformations without restarting their entire pipeline (PE7, PJ2, PJ8). For example, PE7 organized the outputs of each preprocessing stage in separate folders ("Level 1 – USGS Product", "Level 2 – Stacked", "Level 3 – MESMA", "Level 4 – Shade Normalized"), explaining:

> I think Levels 3 and 4 [are] where a lot of stuff is going to change, where I might decide to change the parameters or do it a little differently. And so what I can do is just quickly [delete] this entire folder [Level 3] ... and it'll clean the slate. And then I'll go back to Level 2, and I'll just rerun everything again from Level 2 to get me to Level 3. ... It kind of speeds up the process.

5.3 Analyzing Geospatial Data

For participants, developing geospatial analyses involved constructing pipelines that applied many geospatial operators (in a particular order) to input layers to produce target outputs. Geospatial operators transform both the geometries and attributes of geospatial data, making it difficult to reason about their behavior. For example, the Dissolve operator merges the boundaries of geographic features possessing a shared attribute value and combines attributes of merged features using an aggregation function (e.g., `sum`) (Figure 5.2).



```python
import geopandas as gpd

ca_sp3_tracts =
  gpd.read_file("tracts.shp")

ca_sp3_counties =
  ca_sp3_tracts.dissolve(
    by="COUNTY",
    aggfunc="sum"
  )
```

GEO_ID	NAME	COUNTY	TOTAL_POP	AIAN_POP
14000US06081613800	Census Tract 6138, San Mateo, CA	081	4363	2
14000US06081602900	Census Tract 6029, San Mateo, CA	081	3727	0
14000US06041131100	Census Tract 1311, Marin, CA	041	1280	1
14000US06041132100	Census Tract 1321, Marin, CA	041	1926	0
14000US06041133000	Census Tract 1330, Marin, CA	041	2870	57
…	…	…	…	…

COUNTY	TOTAL_POP	AIAN_POP
081	737888	9397
041	259441	673
...	...	...

Figure 5.2: **An example geospatial operator.** The DISSOLVE operator is used to group Census tracts in northern California by a shared attribute (`COUNTY`). In addition to merging the geographic boundaries of Census tracts, DISSOLVE aggregates values in the attribute table using an aggregation function (e.g., `sum`, `first`, etc.).

Constructing analysis pipelines required participants to have deep knowledge of operators and their semantics as well as the ability to inspect and debug generated outputs.

Identifying Geospatial Operators

Participants struggled to identify the correct operators to transform input layers into target outputs (PE3, PE7, PE9, PS4, PJ4, PO2). Even an expert, PE7, noted that distinguishing the behaviors of different geospatial operators is challenging: "I can never remember the vector operations. There's like UNION and MERGE. COMBINE! I can never remember exactly what they do. I know exactly what the output should look like in the end; I'm just trying to figure out the tool that gets me that output." PE9 spent 16 minutes searching for a `geopandas` operator to filter a point layer to locations falling within a specific polygon in a separate layer. They experimented with programs using `intersection` and `sjoin` before identifying a solution using `within`, reflecting: "I feel like I spend a lot of time getting stuck on, like, very simple GIS. It's things like MERGE vs. JOIN, getting confused with which one you want. Or SPATIAL JOIN vs. a regular JOIN. Sometimes just the terminology can be confusing, and sometimes it's not consistent between QGIS and Arc[GIS] and `geopandas`." The number of operators in GIS software and geospatial analysis libraries exacerbates this challenge. For example, ArcGIS has over 200 operators in its Spatial Analyst toolbox, ranging from bitwise operators to kriging algorithms [38]. This is only one of its 41 toolboxes.

Alternative Expressions of Intent

Although participants struggled to construct analysis pipelines, many could describe their intent in other ways (PE7, PE9, PS4, PJ3, PJ4, PJ7, PJ8). Some used natural language descriptions, either spoken aloud (PE7, PE9, PS4, PJ3, PJ4, PJ7) or written as comments (PE8, PS4, PJ1, PJ4). For example, PJ4 phrased their intent as a question: "How many homicides did each neighborhood have this year, and how did that compare to, like, last year or the last five years, or something like that, right? ... So now I'm doing the puzzle in my head, like, how am I gonna get there?" They proceeded to write individual subgoals for each analysis step in comments in their Jupyter notebook (e.g., "Spatially join homicides to [neighbor]hoods"). Some participants interacted directly with features in a map view to express their intent (PS4, PJ7). PJ7 used their mouse to demonstrate in QGIS how they would compute buffers around each line feature in their stream dataset, then compute the area of overlap between these buffers and a raster deforestation dataset. This would yield the total area of illegal logging in their analysis region.

Code Foraging

When participants could not identify the correct operator for an analysis context, they resorted to foraging for similar analysis examples on Google (PE3, PE7, PE9, PJ2, PJ4), StackOverflow (PE9, PE10), in documentation (PE7, PE9, PS4, PJ4), in online tutorials (PE3, PE7, PE9, PS3, PS4, PJ3, PJ8), in colleagues' computational notebooks and source code (PE1, PE4, PE5, PS3, PJ3), or in their own notebooks and source code (PE1, PE9, PE10, PS3, PJ3, PJ4, PJ8). PE9 demonstrated nearly all of these behaviors, visiting six online tutorials, six StackOverflow pages, and two pages of the `geopandas` documentation to determine the first two operators to use in their pipeline (Figure 5.3).

Understanding Geospatial Operator Semantics

Even when participants could identify candidate operators, they struggled to understand operator behaviors (PE3, PE7, PE8, PE9, PJ4, PJ8, PO2). As PE7 and PE9 noted in Section 5.3, this is partly due to the ambiguous naming of geospatial operators. Moreover, operator semantics differ subtly across GISs and geospatial analysis libraries, meaning "you do need some sort of specificity for doing the actual [analysis]" (PS2) in a particular environment. For example, ArcGIS's MERGE combines vector layers of any geometric type—point, line, or polygon—into a single layer [40], while its QGIS-equivalent, MERGE VECTOR LAYERS, can only merge vector layers of the *same* geometric type [3]. `geopandas merge` inherits from `pandas`, ignoring geometry altogether and performing a join on shared attributes [51]. As this example illustrates, knowledge of geospatial operator behavior in one tool rarely transfers to another.

Participants used diverse strategies to understand operator semantics. I highlight two common techniques.

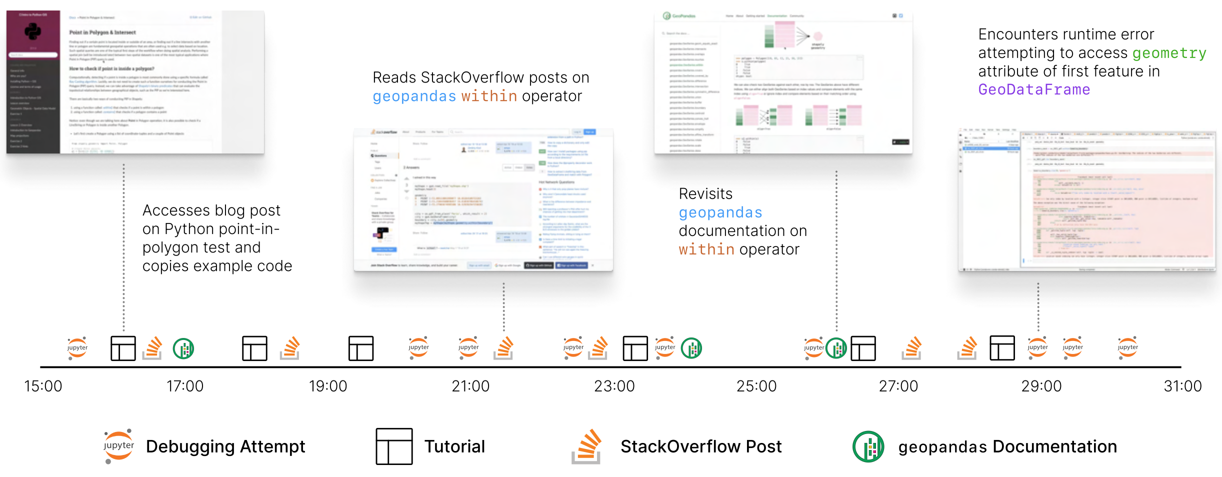

Accesses blog post on Python point-in-polygon test and copies example code
Reads StackOverflow posts on geopandas within operator
Revisits geopandas documentation on within operator
Encounters runtime error attempting to access geometry attribute of first feature in GeoDataFrame
15:00
17:00
19:00
21:00
23:00
25:00
27:00
29:00
31:00
Debugging Attempt
Tutorial
StackOverflow Post
geopandas Documentation

Figure 5.3: **Timeline of PE9's attempts to identify geospatial operators.** PE9 moved between tutorials, StackOverflow, and library documentation to identify the correct `geopandas` operator and syntax to filter a `GeoDataFrame` of cell phone location records to those falling within a polygon from a separate `GeoDataFrame`. They intermixed foraging for example code with testing candidate operators for 16 minutes before arriving at a working solution.

Output-Centered Hypothesis Testing

To test hypotheses about candidate operators' behaviors, participants ran operators, then manually inspected generated outputs (PE1, PE3, PE7, PE9, PS1, PJ4, PO2). For example, PE3 attempted to combine two single-band rasters into one multi-band raster in QGIS, hypothesizing that the Merge operator might be appropriate for the task. After running Merge, they inspected the output raster and found that it was still composed of a single band. They next examined pixel values of this raster, noticing they were identical to pixel values of *just one* of the input rasters. From this inspection, they inferred that Merge stitches together input rasters of differing geographic extents rather than combining raster bands.

When testing candidate operators, participants focused on small subsets of pixels or features and compared their values in input layers to their corresponding values in outputs. Sometimes, selection of pixels or features was random (PE7, PS2, PO2). More often, they selected parts of the output where unexpected behavior would produce obviously incorrect values (PE1, PE3, PE8, PS1, PJ2, PJ3). For example, PE1 computed a Normalized Difference Water Index raster and checked the pixel values of a lake in the generated output; if the algorithm succeeded, these values would be close to the maximum value of one.

Observing Feature Count Changes

Many geospatial operations, such as those that filter, intersect, or aggregate geographies, produce output layers containing a different number of features than their inputs. Participants used changes in feature counts to assess operator behavior, with the magnitude and direction of change serving as proxies for correctness (PE9, PE10, PS1, PS4, PJ2, PJ3, PJ4, PJ8). For example, PS4 checked the feature count of the dataframe produced by an `st_join` operation: "This should only be 372 observations because each [Census] tract is unique, but instead `test2` [the output dataframe] is 2790, which is implying that there is something wrong."

Visibility of Geometry in Programming Environments

Participants relied on examining the geometry of their data to understand operator behavior and validate operator output. GISs center the geometry of geospatial data via a map view, a canvas that allows users to pan, zoom, and inspect features and pixels directly. Conversely, participants using programming environments had to write additional code to perform these interactions (PE8, PE10, PJ7, PJ8, PO2). For example, PO2 wrote code to pan and zoom static `matplotlib` figures to particular parcels in their OpenStreetMap dataset. This involved a repetitive process of guessing the coordinates of bounding boxes containing the parcels, updating a Python dictionary encoding these coordinates, re-executing their code in IPython, and re-rendering the `matplotlib` figures until they achieved the desired view.

Programming environments made rendering and interacting with geospatial data challenging enough that, even when participants used them for analysis, they often moved their data to GISs to "see" and "layer" (PJ6) it interactively (PE9, PJ2, PJ4, PJ6). PJ2 explained the immediate visibility of their data's geometry in GISs outweighed the performance benefits of code: "I'm working in QGIS. I know that it's slower than it would be to do it in PostGIS or maybe even `geopandas`, and so I've considered switching to that. But I'm still . . . new enough that I need to kind of 'see' to make sure my projections are right and stuff like that." PJ4 performed their analysis using `geopandas` in Jupyter but explained they would visualize the results in QGIS: "Now I could try to visualize it here with `matplotlib` and `geopandas`, but I know those things are . . . not interactive and so I'm like, 'I gotta take this to QGIS.'" These findings extend prior work highlighting visual exploration and cross-layer correlation as integral exploratory analysis techniques for geospatial data users [77, 37]. Participants wanted visibility into their data's geometry not only to identify spatial patterns but also to validate the correctness of their analyses visually.

5.4 Representing Geospatial Analyses

Participants sought to represent their analyses in reproducible, shareable forms. While some GISs maintain a record of users' geoprocessing operations, this record is not grouped by project or user session, omits changes to layer symbologies, and is encoded in formats like

XML [41] or command-line expressions [1] that participants did not otherwise use. As a result, participants using GISs created informal program representations outside their GIS to preserve information about analyses.

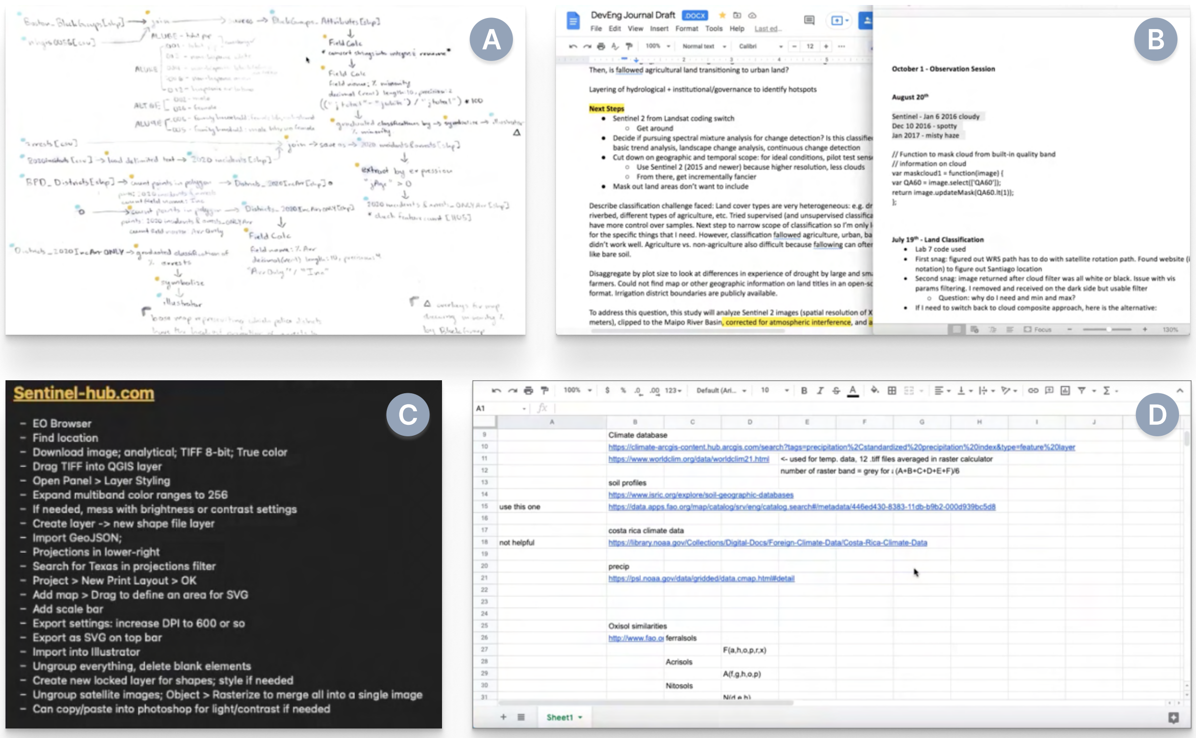
Reproducing Geospatial Analyses

Participants using GISs had difficulty reproducing their geospatial analyses, either because they struggled to remember the current analysis state (PJ7) or lacked documentation on how they performed the analysis previously (PJ5, PJ6). For example, PJ7 revisited a QGIS project to expand the geographic extent of their analysis region but could not recall if they had already clipped their layers to the new extent. They noted they "come across that problem a lot of remembering where I was and what I've done already." Some participants tried to reverse engineer their workflows from generated artifacts (PJ5, PJ6): "I'm just looking through . . . some of my previous [exported SVGs] to remember what I did from here" (PJ5). Prior studies of geospatial data users have similarly identified tracking data provenance as a recurrent challenge [115].

Participants using GISs frequently relied on built-in history interfaces to "backtrace" (PS1) operations they ran previously (PE3, PS1, PJ2). For example, PE3 and PS1 used the Recent menu in the QGIS Expression Editor to recover syntax for SQL queries they recently executed, using these as templates to repeat processing steps with modifications. Likewise, PJ2 used the Recently Used menu in the QGIS Processing Toolbox to rerun Buffer and Overlap Analysis operations with new arguments. However, participants explained these history interfaces are limited by how quickly they become overloaded with stale information. PS1: "[I]f you do so many analyses in Q[GIS] in a week, it all gets buried at the end of the day. There's, like, no way you can actually export that history, which is why I think fundamentally it's only good temporally for a week at most."

Participants using programming environments cited difficulty tracking provenance as a core reason they avoid GISs (PE7, PE9, PS4, PJ8). PE9 explained: "I don't do any of my processing in [QGIS], and mainly because I like that you can track what you did, the traceability of doing it in Python. Versus, there's like none of that if you do it in QGIS. It's like you use a plugin or a function, but there's no track record of it." These participants could also more easily recover information on the current analysis state. While converting a Google Earth Engine pipeline to use imagery from a different satellite, PE1 could not recall how much of this conversion they had completed. To determine where to resume refactoring, they simply ran the program: "I'm just gonna try running this and see what happens because I can't really remember where the part is that I left off." Additionally, participants using programming environments perceived their programs as inherently replicable, shareable artifacts (PE1, PE2, PE7, PS4, PJ1, PJ3, PJ4). PE7: "If someone wants to go back and look at my code—'Oh, he got this shapefile from here and this shapefile from here, and he's pushing them together.' Whereas if you do that in Arc[GIS], you can't really replicate that workflow in the same way."

Creating Informal Program Representations



DevEng Journal Draft .DOCX

File Edit View Insert Format Tools Help Last ed...

Then, is fallowed agricultural land transitioning to urban land?

Layering of hydrological + institutional/governance to identify hotspots

Next Steps

- Sentinel 2 from Landsat coding switch
  - Get around
- Decide if pursuing spectral mixture analysis for change detection? Is this classifie
  basic trend analysis, landscape change analysis, continuous change detection
- Cut down on geographic and temporal scope: for ideal conditions, pilot test sens
  - Use Sentinel 2 (2015 and newer) because higher resolution, less clouds
  - From there, get incrementally fancier
- Mask out land areas don't want to include

Describe classification challenge faced: Land cover types are very heterogeneous: e.g. dr riverbed, different types of agriculture, etc. Tried supervised (and unsupervised classifica have more control over samples. Next step to narrow scope of classification so I'm only l for the specific things that I need. However, classification fallowed agriculture, urban, ba didn't work well. Agriculture vs. non-agriculture also difficult because fallowing can ofte like bare soil.

Disaggregate by plot size to look at differences in experience of drought by large and sm farmers. Could not find map or other geographic information on land titles in an open-sc format. Irrigation district boundaries are publicly available.

To address this question, this study will analyze Sentinel 2 images (spatial resolution of X meters), clipped to the Maipo River Basin, corrected for atmospheric interference, and a

October 1 - Observation Session

August 20th

Sentinel - Jan 6 2016 cloudy
Dec 10 2016 - spotty
Jan 2017 - misty haze

```
// Function to mask cloud from built-in quality band
// information on cloud
var maskcloud1 = function(image) {
var QA60 = image.select(['QA60']);
return image.updateMask(QA60.lt(1));
};
```

July 19th - Land Classification

- Lab 7 code used
- First snag: figured out WRS path has to do with satellite rotation path. Found website (i notation) to figure out Santiago location
- Second snag: image returned after cloud filter was all white or black. Issue with vis params filtering. I removed and received on the dark side but usable filter
  - Question: why do I need and min and max?
- If I need to switch back to cloud composite approach, here is the alternative:

Sentinel-hub.com

- EO Browser
- Find location
- Download image; analytical; TIFF 8-bit; True color
- Drag TIFF into QGIS layer
- Open Panel > Layer Styling
- Expand multiband color ranges to 256
- If needed, mess with brightness or contrast settings
- Create layer -> new shape file layer
- Import GeoJSON;
- Projections in lower-right
- Search for Texas in projections filter
- Project > New Print Layout > OK
- Add map > Drag to define an area for SVG
- Add scale bar
- Export settings: increase DPI to 600 or so
- Export as SVG on top bar
- Import into Illustrator
- Ungroup everything, delete blank elements
- Create new locked layer for shapes; style if needed
- Ungroup satellite images; Object > Rasterize to merge all into a single image
- Can copy/paste into photoshop for light/contrast if needed

	A	B	C	D	E	F	G
9		Climate database					
10		https://climate-arcgis-content.hub.arcgis.com/search?tags=precipitation%2Cstandardized%20precipitation%20index&type=feature%20layer					
11		https://www.worldclim.org/data/worldclim21.html			<- used for temp. data, 12 .tiff files averaged in raster calculator		
12					number of raster band = grey for :		(A+B+C+D+E+F)/6
13		soil profiles					
14		https://www.isric.org/explore/soil-geographic-databases					
15	use this one	https://data.apps.fao.org/map/catalog/srv/eng/catalog.search#/metadata/446ed430-8383-11db-b9b2-000d939bc5d8					
16							
17		costa rica climate data					
18	not helpful	https://library.noaa.gov/Collections/Digital-Docs/Foreign-Climate-Data/Costa-Rica-Climate-Data					
19							
20		precip					
21		https://psl.noaa.gov/data/gridded/data.cmap.html#detail					
22							
23							
24							
25		Oxisol similarities					
26		http://www.fao.oı	ferralsols				
27				F(a,h,o,p,r,x)			
28			Acrisols				
29				A(f,g,h,o,p)			
30			Nitosols				
31				N(d,e,h)			

Sheet1

Figure 5.4: **Informal program representations of geospatial analyses.** (A) PS1 created a hand-drawn diagram using a custom notational system to record operations, layer names, and debugging steps in their analysis pipeline. (B) PE1 recorded high-level analysis steps, issues encountered during analysis, and code snippets in a Google Doc and Microsoft Word document. (C) PJ5 recorded step-by-step instructions for overlaying a GeoJSON file with a Sentinel-2 image across Sentinel Hub, QGIS, Photoshop, and Illustrator. (D) PE3 used a Google Sheet to record information on data sources, their use in the analysis pipeline, and arguments to pass to operators in QGIS.

Participants using GISs created informal program-like representations of their analyses *outside* the software (PE3, PS1, PS2, PJ2, PJ5). Representations ranged from spreadsheets (PE3) to semi-structured text documents (PE1, PS2) (Figure 5.4). PS1 created a hand-drawn diagram using a custom notation based on the QGIS Graphical Modeler [2]. Their diagram specified: the ordering of geospatial operators; the arguments, input layers, and output layers of each operator; attribute table modifications and validation steps to perform at specific pipeline stages; and layers to symbolize in QGIS before export to Illustrator. They also used color as a visual variable, distinguishing layers from operators using blue and gold dots.

Participants also used informal program representations to record data acquisition, cleaning, analysis, and visualization steps spread across multiple tools (PJ2, PJ5). For example, PJ5 used macOS Notes to document a workflow for overlaying a GeoJSON atop a Sentinel-2 image, which involved moving data across Sentinel Hub, QGIS, Illustrator, and Photoshop. They recorded steps ranging from querying a specific Sentinel-2 image in Sentinel Hub to recomposing raster tiles in Illustrator. This degree of tool-hopping was common among participants (PS1, PS2, PJ2, PJ5, PJ6, PJ7, PJ8), but all lacked automated tooling to track cross-system provenance. PJ7 explained: "Y'know, when I'm jumping between QGIS and Python and, well, we were just in Excel, *and* Tableau *and* Adobe Illustrator ... y'know, commenting my code in Python doesn't help me remember where I was in Illustrator."

5.5 Visualizing Geospatial Data

Participants wanted to explore an expansive design space of cartographic representations to visualize their analyses' outputs. However, existing tools made this exploration difficult. Visualizing the same data using different cartographic representations involved starting the cartographic design process from scratch for each map variant. Because the tools participants used for map design could not always natively handle geospatial data, spatial information was lost when cartographic work began.

Sketching Cartographic Variants

Participants wanted to visualize data using multiple cartographic representations to explore the design space of possible maps and provide tangible artifacts for collaborators to evaluate (PS2, PJ5, PJ6). PJ5 had over 20 "concept drafts" of maps for one story, ranging from a gridded heat map to a layout combining choropleth and dot density symbologies (Figure 5.5). One draft included a sequence of mockups showing how the map would respond to a user changing the visualized variables via dropdown menus. These drafts allowed PJ5 to explore cartographic choices with editors: "I have several different versions where someone's like, 'What if this was a fullscreen map and the controls were in the corner, or if this were a side-by-side map?' Yeah, it's predominantly thinking through what is the user experience and what kind of information do we want the reader to be focused on." PJ6 drafted multiple choropleth maps for a story in QGIS and Mapbox Studio, took screenshots of the maps in each tool, designed webpage layouts around the screenshots in Figma, and copied the layouts into a Google Doc for editors to provide comments. They noted that prototyping in a combination of GIS and design software allowed them to compare cartographic choices quickly and get feedback "before I code anything." While these drafts helped PJ5 and PJ6 explore the design space, they were not publication-ready. As a result, both authored code for the chosen maps after the fact.

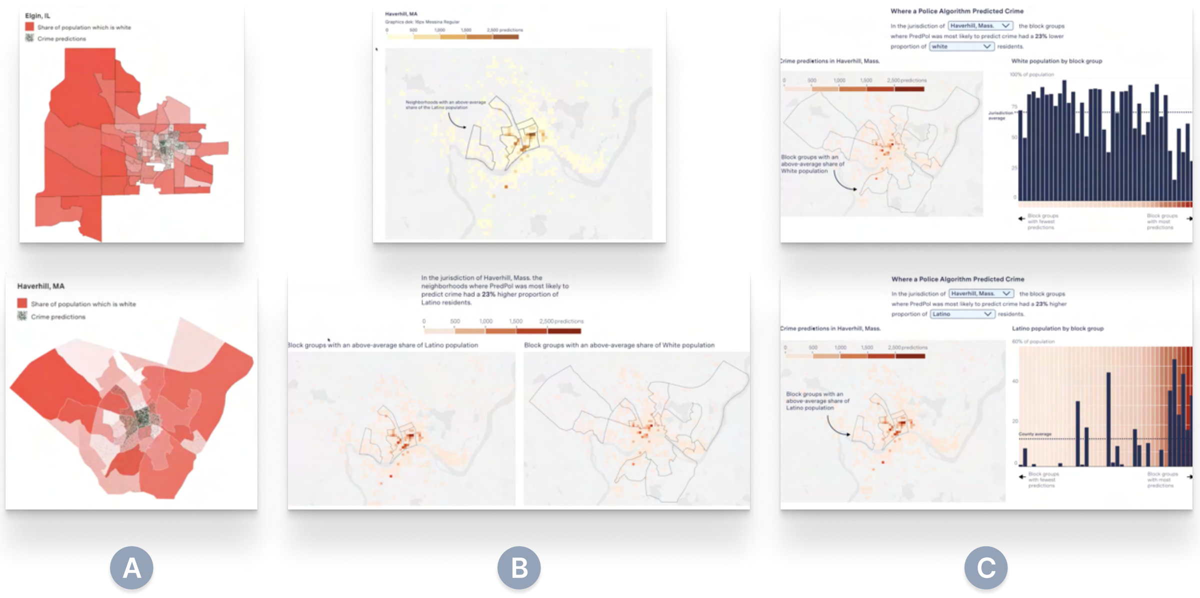

Elgin, IL
Share of population which is white
Crime predictions
Haverhill, MA
Share of population which is white
Crime predictions
Haverhill, MA
0 500 1,000 1,500 2,500 predictions
In the jurisdiction of Haverhill, Mass. the neighborhoods where PredPol was most likely to predict crime had a 23% higher proportion of Latino residents.
0 500 1,000 1,500 2,500 predictions
Block groups with an above-average share of Latino population
Block groups with an above-average share of White population
Where a Police Algorithm Predicted Crime
In the jurisdiction of Haverhill, Mass. the block groups where PredPol was most likely to predict crime had a 23% lower proportion of white residents.
Crime predictions in Haverhill, Mass.
0 500 1,000 1,500 2,500 predictions
Block groups with an above-average share of White population
White population by block group
100% of population
75
50
25
0
Jurisdiction average
Block groups with fewest predictions
Block groups with most predictions
Where a Police Algorithm Predicted Crime
In the jurisdiction of Haverhill, Mass. the block groups where PredPol was most likely to predict crime had a 23% higher proportion of Latino residents.
Crime predictions in Haverhill, Mass.
0 500 1,000 1,500 2,500 predictions
Block groups with an above-average share of Latino population
Latino population by block group
60% of population
40
20
County average
Block groups with fewest predictions
Block groups with most predictions
A
B
C

Figure 5.5: **A selection of PJ5's draft maps.** (A) PJ5's initial drafts combined choropleth and dot density symbologies in a single map. (B) PJ5 created a gridded heat map (top) and used this style with a modified color scheme in a small multiples layout (bottom). (C) PJ5 tried an alternate layout combining a heat map with a bar chart. These mockups also included a dropdown allowing users to change the variable displayed on the map. The top and bottom mockups show two different UI states in response to user interaction.

To create multiple map versions, participants went through their entire visualization pipeline—spread across code, GISs, and design software—for each variant (PJ5, PJ6). Several participants used PJ6's strategy of screenshotting draft maps in GISs (PS2, PE5) or the browser (PJ5) to capture variants at intermediate stages. This allowed them to record many versions distinguished by *minor* cartographic differences (e.g., color scales, basemaps), even if changing map styles was too time-consuming. PJ5 avoided repeating their analysis and visualization process by creating synthetic layers in some mockups: "If you look at those mocks, they're not fully accurate because I wasn't able to do any of the data analysis I wanted to do. So it was more of my crude approximation, like, 'Well, y'know, if we allowed the user to mess with these filters, here's kind of what it would look like.'"

Geospatial Information in Design Software

Participants using GISs for analysis rarely conducted cartographic work there; instead, they moved their data into design software such as Illustrator or Photoshop (PS1, PS2, PJ5, PJ7). This process involved transforming data encoded in geospatial formats (e.g., GeoJSON, GeoTIFF) into non-spatial formats (e.g., SVG, PNG). This transformation jettisons the spatial information of the data, making it challenging to alter analyses after beginning visualization work. Participants described moving from GIS to design software as “crossing a rubicon” (PS2) and “mapping without a net” (PJ5) because the transition broke the link between features and their real-world geographies. For example, PJ5 wanted to alter the brightness of a Sentinel-2 image exported from QGIS to Photoshop while keeping it geographically aligned to a GeoJSON exported separately to Illustrator: “If I crop this by so much as a pixel, right, then it’ll no longer be accurate to that geography . . . But as long as this raster image and my Illustrator SVG remain the same dimensions, then they will be accurate to one another.”

Participants used a combination of strategies to avoid spatial information loss when moving to design software. The most common was to avoid resizing a map after export from GIS (PS1, PS2, PJ5), which guaranteed the preservation of spatial accuracy during cartographic design. Participants also exported more data than they planned to use to avoid re-exporting. PS1 maintained a layer group in QGIS named “primary” to house all layers they believed could be important for visualization because “I never know what I want to end up exporting as an SVG into Illustrator.”

Chapter 6

Design Opportunities

Our findings suggest new research directions and design opportunities for geospatial analysis and visualization systems.

6.1 Solving Geospatial Data Constraints

Opportunity 1. Participants struggled to find geospatial data satisfying complex spatial and temporal constraints (Section 5.1). While many could describe their constraints succinctly, satisfying them involved constructing bespoke workflows to combine, align, and simplify their raw datasets (Section 5.2). These challenges suggest an opportunity for tools that (1) offer alternative programming abstractions to express data constraints and (2) infer geospatial data queries and transformations from constraints.

Designers could take inspiration from constraint-based programming systems, which have addressed similar problems in visualization [88] and mathematical diagramming [122]. These systems allow users to describe target outputs (e.g., charts, diagrams) via constraints expressed in domain-specific languages (DSLs). Compilers then translate these programs into optimization problems for constraint solvers. In the geospatial setting, GeoSPARQL's topology vocabulary extension [22] provides an example of a constraint-based system for enforcing topological invariants. Our findings suggest that a constraint-based language for geospatial data could allow users to compose additional constraints related to spatial extent, geographic accuracy, spatial resolution, temporal resolution, and occlusion characteristics.

Many constraint-based languages are declarative—users describe *what* a program should generate without specifying *how*. Declarative DSLs have addressed domain experts' needs in fields including cloud infrastructure engineering [57], interactive graphics [108, 61], and programmable biochemistry [95]. A declarative DSL for geospatial data transformation could shift much of the burden of wrangling to automated tooling. For example, we could imagine PE2 replacing their resampling algorithm (Section 5.2) with an expression defining the required spatial resolution of all input rasters; the language implementation would be responsible for generating code to perform the resampling. Prior programming languages work targeted at geospatial data [5, 71] has focused on simplifying querying, but our findings indicate that users need better abstractions for transformation.

6.2 Assistive Tools for Constructing Geospatial Analysis Pipelines

Opportunity 2. Participants could describe the target outputs of their geospatial analyses but struggled to construct pipelines to produce them (Section 5.3). This suggests an opportunity for tools that (1) accept non-code specifications of analysis intent, (2) synthesize analysis programs that satisfy specifications, and (3) support users in editing programs.

Program synthesis approaches, such as programming-by-example (PBE) and programming-by-demonstration (PBD), excel in contexts where users can express outputs but struggle to author code. In prior work, C-SPRL used PBD to synthesize spatial data queries from recordings of user interactions in a GIS [114, 115]. However, we are not aware of synthesizers that aid programmers in selecting geospatial operators for their analysis pipelines. Given participants' use of many alternative specifications of intent—natural language descriptions, direct interaction with maps, constraints on outputs (Sections 5.3 and 5.3)—operator selection may be fertile ground for synthesis. Moreover, prior research in data science has shown that PBE can assist with operator selection in libraries with large API surfaces [7], suggesting that search over the vast numbers of operators in GISs and geospatial analysis libraries is tractable.

Based on participants' code-foraging behaviors (Section 5.3), synthesis may be useful even if synthesizers cannot reach all plausible programs. We observed that participants were comfortable tweaking existing programs to reach their target solution. Thus, tools that support goal (3) above may be helpful both as companions to synthesis and independently. This echoes design guidance from [115] arguing that synthesized programs should be editable to support users in adapting them to similar tasks.

Opportunity 3. Participants relied on running operators and manually inspecting outputs to understand operator semantics (Section 5.3). This was computationally expensive and time-consuming, suggesting an opportunity for tools that surface information on operator semantics without requiring execution across entire inputs.

Live programming offers users immediate visual feedback on program behavior using concrete inputs [75, 107]. We observed that participants already use small collections of geographic features or pixels as test cases to infer operator behavior, implying that this technique may fit existing debugging patterns (Section 5.3). Our observations of data subsetting practices (Section 5.2) reinforce this connection—participants already manually reduce dataset size to get faster feedback. A live programming system for geospatial analysis could automate these practices.

Reproducible, Shareable Geospatial Workflows

Opportunity 4. Participants using GISs struggled to create reproducible, shareable geospatial workflows (Section 5.4). Limitations in existing history interfaces made it difficult to recover information on the current analysis state or revisit past analysis decisions (Section 5.4). These struggles suggest opportunities for tools that (1) support efficient search through system history and (2) distill history into a portable and executable representation.

Tools like Verdant [69] and Variolite [67] offer glimpses of alternative ways of surfacing analysis histories. For example, Verdant offers views of computational notebook history by time, artifact, and structured search. It also produces a single-file history representation that users can share with collaborators. Producing usable history tools for geospatial data will first require identifying what history information is valuable. Participants' informal program representations suggest that tracking provenance information at the layer level and recording modifications to layer symbologies and attribute tables could augment existing systems' approach of logging geoprocessing operations (Section 5.4).

Beyond making history searchable, developers could borrow techniques from record and replay [14, 76, 100, 84] to make history executable. An observational study of GIS users found that participants' work was often highly repetitive [31], implying that record and replay could help automate tedious tasks in GISs. Although we observed few cases of repetitive work, our participants frequently used history interfaces to manually replay past operations with modified arguments (Section 5.4). This suggests an opportunity for tools that can automatically parameterize recordings of user interactions into generalized programs. Ringer [6, 18, 19] and BluePencil [87] are models in this space; for example, Ringer transforms user demonstrations in web browsers into scripts that can be modified, parameterized, and replayed.

Exploring the Cartographic Design Space

Opportunity 5. Participants wanted to visualize their geospatial data using multiple cartographic representations, but transitioning between representations required engineering each one from scratch (Section 5.5). This suggests an opportunity for cartographic design tools that reduce the viscosity [8] of switching between map types.

High-level DSLs offer a low-viscosity approach for design space exploration. In visualization, grammars of graphics [120] like Vega-Lite [108] pair declarative primitives for describing visualizations with a compiler for generating low-level rendering code. Encouragingly, these grammars already have some support for geospatial data. However, because they restrict the geospatial file formats, data models, and cartographic types that users can work with, these grammars cannot express the majority of maps our participants created. In rethinking a grammar of graphics for cartography, our findings indicate that supporting more cartographic representations and minimizing the number of program edits required to switch representations are critical design considerations. Libraries like Plot [93] and Bertin.js [73] are promising examples of tools that make map type a first-class primitive.

Opportunity 6. Many participants used direct manipulation design software to visualize geospatial data. These tools discard all geographic information, making it difficult to refactor an analysis once visualization work has begun (Section 5.5). This suggests an opportunity for tools that (1) bridge geospatial analysis and cartographic design and (2) maintain the underlying geospatial data representation of graphical elements while supporting direct manipulation.

Prior work indicates that pairing programmatic and direct manipulation paradigms is possible for tasks with visual outputs. Sketch-n-Sketch [63] successfully applies this technique to SVG editing. Users can manipulate the output SVG or edit the program representation to make changes; Sketch-n-Sketch propagates edits bidirectionally to maintain the correspondence between program and graphic. Such an approach for geospatial data could preserve the geographic information of graphical elements during visualization, allowing users to return to analysis without obviating in-progress design work. Moreover, this approach could address participants' core issue with using direct manipulation design software for cartography—that once a particular map design was chosen, they often had to reproduce the map in code (Section 5.5).

Chapter 7

cartokit

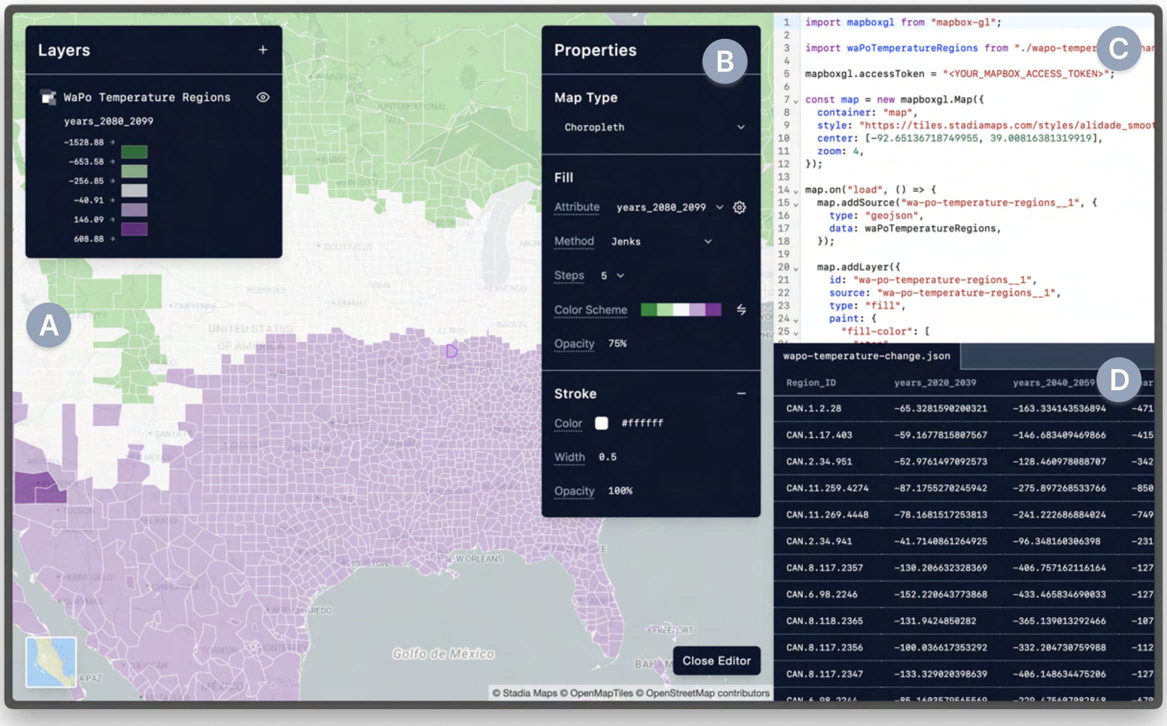
In this chapter, I introduce `cartokit`, a direct manipulation programming environment for interactive cartography on the web. `cartokit` aims to realize several of the design opportunities described in Chapter 6 in a unified system. In addition, it seeks to fill critical gaps in existing direct manipulation interfaces for cartographic design. Toward these goals, `cartokit` makes three novel contributions. First, it exposes direct manipulation interactions for styling geospatial data while preserving the underlying data representation, allowing users to move fluidly between analysis and visualization work. Second, it enables single-click transitions between map types, facilitating rapid exploration of the cartographic design space. Third, it compiles user-created maps—on each interaction—into JavaScript programs, providing direct access to a reproducible program artifact and supporting transfer to other code-based environments.

In the following sections, I describe `cartokit`'s design and implementation in detail. First, I provide a brief system overview. Next, I survey prior work on direct manipulation interfaces for cartographic design as well as direct manipulation interfaces supporting code generation from visual artifacts. From there, I highlight novel facets of `cartokit`'s design that extend beyond prior work, focusing on (1) support for cross-geometry map type transitions, (2) support for user-defined data transformations, and (3) code generation to a general-purpose language. Finally, I demonstrate these novel facets by way of a case study showing how `cartokit` users can create expressive thematic maps comparable to graphics published in national newsrooms.

7.1 System Overview

`cartokit` is a web application implemented in TypeScript [86] using Svelte [24], a functional, reactive programming framework for building user interfaces. The user interface consists of four primary elements (Figure 7.1):

(A) The **Map View**, which displays the map the user is actively manipulating and styling. `cartokit` uses MapLibre GL JS [23], a WebGL-based graphics library for interactive



Properties

Map Type: Choropleth

Fill
- Attribute: years_2080_2099
- Method: Jenks
- Steps: 5
- Color Scheme
- Opacity: 75%

Stroke
- Color: #ffffff
- Width: 0.5
- Opacity: 100%

```
import mapboxgl from "mapbox-gl";

import waPoTemperatureRegions from "./wapo-temper

mapboxgl.accessToken = "<YOUR_MAPBOX_ACCESS_TOKEN>";

const map = new mapboxgl.Map({
  container: "map",
  style: "https://tiles.stadiamaps.com/styles/alidade_smoot
  center: [-92.65136718749955, 39.00816381319919],
  zoom: 4,
});

map.on("load", () => {
  map.addSource("wa-po-temperature-regions__1", {
    type: "geojson",
    data: waPoTemperatureRegions,
  });

  map.addLayer({
    id: "wa-po-temperature-regions__1",
    source: "wa-po-temperature-regions__1",
    type: "fill",
    paint: {
      "fill-color": [
```

wapo-temperature-change.json

Region_ID	years_2020_2039	years_2040_2059	ear
CAN.1.2.28	-65.3281590200321	-163.334143536894	-471
CAN.1.17.403	-59.1677815807567	-146.683409469866	-415
CAN.2.34.951	-52.9761497092573	-128.460978088707	-342
CAN.11.259.4274	-87.1755270245942	-275.897268533766	-850
CAN.11.269.4448	-78.1681517253813	-241.222686884024	-749
CAN.2.34.941	-41.7140861264925	-96.348160306398	-231
CAN.8.117.2357	-130.206632328369	-406.757162116164	-127
CAN.6.98.2246	-152.220643773868	-433.465834690033	-127
CAN.8.118.2365	-131.9424850282	-365.139013292466	-107
CAN.8.117.2356	-100.036617353292	-332.204730759988	-112
CAN.8.117.2347	-133.329020398639	-406.148634475206	-127

Figure 7.1: **The cartokit user interface.** (A) Users interact directly with features rendered in the **Map View**. (B) Users style layers on the map via controls in the **Properties Panel**. (C) Users access the JavaScript program generated by `cartokit` in the **Code Viewer**. (D) Users can view and sort the tabular data associated with the selected layer in the **Data Table**.

maps, for rendering.

(B) The **Properties Panel**, which provides access to interface elements for editing visual and semantic properties of layers on the map.

(C) The **Code Viewer**, which displays the JavaScript program—generated by `cartokit`—corresponding to the user's map.

(D) The **Data Table**, which shows the tabular data associated with the currently selected layer.

Secondary interface elements supply functionality for uploading geospatial datasets, rendering layer information (e.g., display names, legends), authoring and previewing data transformations, and altering base layer selection.

Interaction Model

`cartokit` is an output-directed, direct manipulation programming environment for building web maps. Rather than writing a program by hand to visualize geospatial data on an interactive map, `cartokit` users style their geospatial data through direct manipulation while the system generates a program to reproduce the map one-for-one. In essence, users reach working programs by modifying the program's *output* (i.e., the map) rather than its source code.

In the following sections, I outline a typical user interaction with `cartokit` across three phases: uploading data, styling data, and compilation.

Uploading Data

`cartokit` users begin interacting with the system by uploading a geospatial dataset encoded as GeoJSON [15], a popular interchange format for vector geospatial data. I selected GeoJSON as `cartokit`'s primary data format due to its wide adoption among my participants and its compatibility with many different geospatial graphics engines (e.g., Mapbox GL JS, MapLibre GL JS, Leaflet, deck.gl, d3-geo). To upload data to `cartokit`, users start by clicking the `+` icon in the **Layers Panel**; `cartokit` then renders a modal prompting them for the GeoJSON and an associated layer name. Users have the option of uploading a GeoJSON file from disk or supplying an API endpoint from which `cartokit` can fetch the GeoJSON. Once supplied, `cartokit` parses the GeoJSON in a Web Worker and stores the result in memory. Moving data parsing to a Web Worker frees the JavaScript engine's main thread to continue processing user interactions while upload occurs. This is an important optimization for handling GeoJSON datasets, which can range in size from tens to hundreds of megabytes.

Styling Data

After `cartokit` ingests users' geospatial data, the system renders it using a default cartographic representation appropriate for the input geometry type. For example, `cartokit` uses the FILL map type to render a GeoJSON dataset consisting of `Polygon`s, applying a uniform fill and stroke to all features in the layer. Conversely, it uses the POINT map type for `Point` datasets, applying a uniform fill, stroke, and radius to all points in the layer. To initiate styling, the user selects an individual feature (e.g., a single `Point`, `Line`, or `Polygon`) on the map. Feature selection opens the **Properties Panel**, which includes interface controls for altering visual properties such as fill, stroke, opacity, size, and color scheme. The **Properties Panel** also contains interface controls for modifying higher-level properties of the map, including the map type, the statistical method used for setting breaks in continuous data, and mappings of data attributes to visual encodings. While user selections occur on *individual* features in a layer, interactions in the **Properties Panel** affect the appearance of *all* features in the layer. This behavior differs from the model used in direct manipulation design software (e.g., Adobe Illustrator, Sketch, Figma), where modifications of properties only affect selected features. However, this behavior closely mirrors GIS software, which

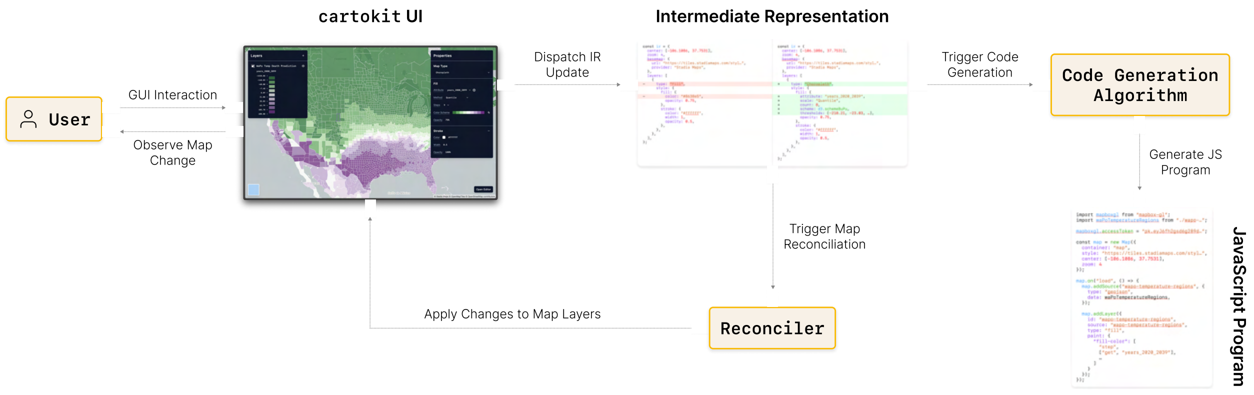


Figure 7.2: **`cartokit`'s system architecture.** User interactions in the **Properties Panel** dispatch updates to the `cartokit` IR. The **Reconciler** propagates these updates back to the map, while the **Code Generation Algorithm** generates a JavaScript program that, when executed, reproduces the map one-for-one.

groups all features of a dataset into a single layer; modification of a single feature is then interpreted as a change to the layer's overall symbology.

Compilation

On every modification of the map—including zoom events, pan events, and basemap changes, in addition to interactions in the **Properties Panel**—`cartokit` generates a JavaScript program that reproduces the map one-for-one. We can think of this component of `cartokit` as a compiler that transforms a sequence of user interactions with a map GUI into JavaScript programs producing equivalent output. The generated programs use Mapbox GL JS [79], a popular WebGL-based library for interactive maps, as the graphics engine. Notably, nothing about `cartokit`'s design is tied to this choice of "backend"; the compiler could just as easily target different geospatial graphics engines or different programming languages altogether.

Unlike a traditional compilation model—in which modifying a source program and compiling a source program are separate actions—styling and program generation are tightly coupled in `cartokit`. This coupling has several important benefits. First, the generated program is always up to date with the user's styling changes. Second, it is impossible for users to reach invalid programs; every available interaction in the interface predictably transforms the generated program. To achieve stable, near real-time compilation, `cartokit` maintains and repeatedly updates an intermediate representation of the map and all user-defined layers. Each interaction with the interface applies a transformation to the IR, which in turn triggers `cartokit`'s reconciler and code generation algorithm. The reconciler ensures that the map GUI reflects the user's changes while the code generation algorithm transforms the IR to the output JavaScript program. Figure 7.2 provides a high-level overview of the system architecture. `cartokit` displays the generated program in the **Code Viewer**, from which a user can inspect or copy it. This allows users to take their cartographic design work—codified in the program representation—to other tools when they wish to achieve something beyond the system's supported functionality.

7.2 Related Work

In this section, I discuss prior work on direct manipulation interfaces for cartographic design as well as direct manipulation interfaces supporting code generation from visual outputs.

Direct Manipulation Interfaces for Cartographic Design

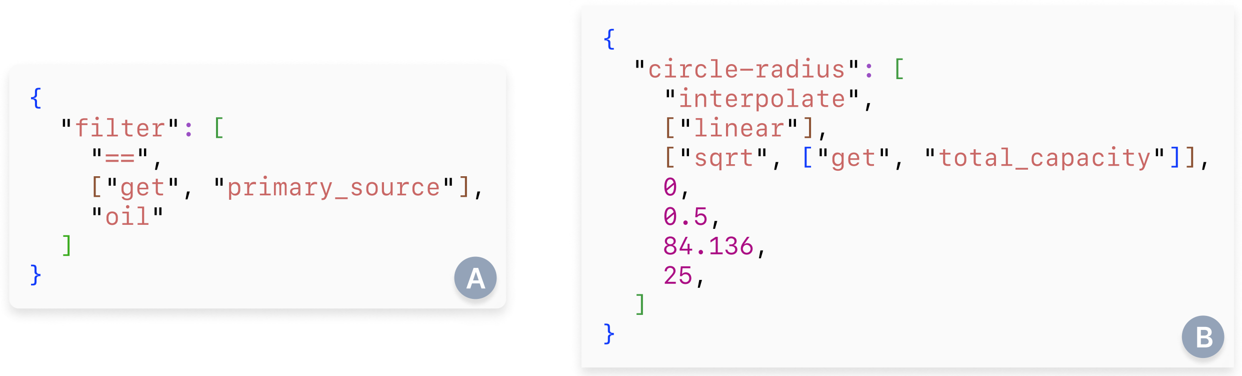
Existing direct manipulation interfaces for cartographic design align in their decisions to (1) limit in-system data transformation, (2) target custom specification formats rather than general-purpose languages, and (3) expose interface controls that map closely to the APIs of their respective graphics engines. We explore these themes in the two systems most closely related to `cartokit`—Mapbox Studio and Felt. Other direct manipulation systems for cartographic design exist (e.g., Tableau, Datawrapper). However, these systems differ substantively from the others in that they hide the underlying program from users. I briefly discuss these systems at the end of this section.

Mapbox Studio

Mapbox Studio [80] is a direct manipulation editor for geospatial data targeting the Mapbox Style Specification [81], a custom JSON specification format. Users style geospatial data using interface controls in a GUI; alternatively, they can textually edit individual key-value pairs of the generated JSON. Interestingly, Mapbox Studio never exposes the full JSON style specification to users within the interface. Instead, each direct manipulation control has an associated code editor to enable one-off textual edits at the property level.

To render a map created in Mapbox Studio in the browser, users rely on (1) embedding an `iframe` referencing a Mapbox-generated url in a web page, (2) fetching their style specification from a Mapbox-generated url in a JavaScript program using Mapbox GL JS, or (3) exporting their style specification from Mapbox Studio and inlining it in a JavaScript program using Mapbox GL JS. In the first case, a user has no ability to modify the underlying program outside of Mapbox Studio. In the latter two cases, the user has direct access to the style specification but must write the JavaScript program from scratch. These programs can quickly become complex; rendering a single dataset involves importing and parsing the GeoJSON data, instantiating the `Map` instance, wiring up `onload` event listeners, and creating layers and layer sources. In all three cases, Mapbox Studio users are dependent on graphics engines that can interpret the Mapbox Style Specification directly.

Data Transformation Mapbox Studio expects users' geospatial data to be pre-analyzed; once loaded, data cannot be transformed. However, Mapbox Studio does support "views" on data through attribute-based filtering and computed attributes. For users, constructing views involves writing expressions using Mapbox's expressions syntax, an array-based JSON DSL. These expressions are evaluated at run time by a custom interpreter in Mapbox GL JS. Figure 7.3 shows two example Mapbox expressions illustrating filtering and computed attributes.



```
{
  "filter": [
    "==",
    ["get", "primary_source"],
    "oil"
  ]
}
```

```
{
  "circle-radius": [
    "interpolate",
    ["linear"],
    ["sqrt", ["get", "total_capacity"]],
    0,
    0.5,
    84.136,
    25,
  ]
}
```

Figure 7.3: **Example Mapbox expressions.** The Mapbox expressions syntax is a JSON DSL loosely based off of Lisp S-expressions. (A) Users can specify predicates over attributes of a dataset via a `filter` expression. (B) Users can compute new data from existing attributes using a range of arithmetic operators.

The introduction of a custom syntax makes users dependent on the Mapbox GL JS interpreter supporting the language constructs they need. More complex data transformations, such as cross-geometry transitions or statistical clustering methods commonly used in geospatial visualization, are currently unsupported. In addition, because these expressions are evaluated internally by the interpreter and applied uniformly to all features, users have limited ability to debug expressions producing unexpected results for specific features (e.g., those containing `NaN` or `null` attributes).

Level of Abstraction Mapbox Studio's user interface exposes controls that map directly to the Mapbox Style Specification. Users work at a level of abstraction closer to the lower-level details of the graphics engine rather than the higher-level language of cartography. Instead of thinking in terms of cartographic representations (e.g., choropleth, proportional symbol, dot density, etc.), users manipulate graphical properties of features (e.g., `fill`, `stroke`, `circle-radius`). For example, in order to create a choropleth map, a user creates a `"fill"` layer, maps the `"fill-color"` attribute to a custom `"step"` expression referencing a data attribute, manually computes statistical breaks in that data attribute, and maps each break to a specific RGB value or hexadecimal code. One notable exception in the Mapbox Style Specification is the `"heatmap"` layer type, which allows users to specify abstract graphical properties like `"heatmap-intensity"` and `"heatmap-weight"` that parameterize an underlying bivariate kernal density estimation algorithm.

Felt

Felt [45] is a direct manipulation editor for geospatial data targeting the Felt Style Language (FSL) [46], a custom JSON specification format. Similar to Mapbox Studio, users can style their geospatial data either through direct manipulation or textual editing of the FSL specification. Unlike Mapbox Studio, Felt exposes the full FSL specification at all times in its Source Viewer.

To render a map created in Felt in the browser, users rely on (1) sharing Felt-generated urls or (2) embedding an `iframe` referencing a Felt-generated url in a web page. Felt has no accompanying JavaScript library to interpret FSL; users can only create and style maps that run on Felt's platform.

Data Transformation Similar to Mapbox Studio, Felt does not support persistent data transformation within the interface but does enable "views" through attribute-based filtering. Filters in FSL use infix operators; for example, the filter in Figure 7.3 (A) would be expressed in FSL as `"$primary_source == oil"`. Unlike Mapbox Studio, Felt currently has no support for computed attributes—users can only visualize fields that have been pre-computed in their datasets.

For certain cartographic representations, Felt includes algorithms for binning and clustering data automatically. For example, users creating choropleth or proportional symbol maps can select from a set of six algorithms to set breaks in continuous numerical data, including Jenks natural breaks, quantiles, quantization, standard deviations, geometric intervals, and manual breaks. Interestingly, these algorithms are only selectable in the direct manipulation interface. FSL provides no language abstractions for these algorithms; instead, users supply breaks manually as an array of numbers for the `"steps"` key in an FSL specification.

Level of Abstraction In addition to supporting modification of visual properties (e.g., the fill, stroke, size, and opacity of rendered marks), Felt's user interface exposes controls for editing higher-level properties of a map. Felt's map type selector is a key example, allowing users to switch cartographic representations with a single click. For example, users interacting with `Point` datasets can transition their data from a point map to a categorical point map, color scaled point map, proportional symbol map, or heat map in one interaction. Notably, Felt limits the set of reachable map types based on the geometry of the input dataset. For example, users cannot transition `Polygon` datasets to map types appropriate for `Point` or `LineString` geometries, despite such cross-geometry transitions being common in cartographic design.

Felt's higher-level interface controls do not always correspond to constructs in FSL. For example, the selected map type and breaks algorithm are represented in the interface but are absent in an FSL specification. In this way, users work at different levels of abstraction depending on whether they use the interface or edit the FSL specification directly.

Related Systems

Several other direct manipulation interfaces for cartographic design exist; however, these systems differ from those above in that they intentionally hide the underlying program representation from users. Tableau [112] is a no-code, direct manipulation editor for constructing data visualizations and data dashboards. While not specialized for geospatial data, Tableau allows users to visualize pre-analyzed vector geospatial data using one of six preset map types. Datawrapper [28] is a no-code, direct manipulation editor for creating annotated graphics, targeted at the data journalism community. Similar to Tableau, it allows users to visualize vector geospatial data using one of four preset map types. In this sense, both Tableau and Datawrapper operate at a similar level of abstraction to `cartokit`, orienting users around the central primitive of the map type. However, `cartokit` extends significantly beyond these systems by (1) supporting in situ, user-defined geospatial and tabular data transformations and (2) generating and exposing a corresponding JavaScript program to reproduce the maps users create.

Code Generation in Direct Manipulation Interfaces

Outside of the domain of cartographic design, several direct manipulation systems have experimented with generating code from sequences of user interactions with visual outputs. Sketch-n-Sketch [63, 83, 20] is one prominent example. As described in Section 6.2, Sketch-n-Sketch is a bidirectional, direct manipulation programming environment for SVG editing. Users can edit either the program source code or the output SVG while the system maintains the correspondence between the two. While `cartokit` is inspired by Sketch-n-Sketch, it differs in two key ways. First, Sketch-n-Sketch targets a custom-built functional language, meaning that generated programs can currently only be evaluated by the system's interpreter. Targeting a custom language comes with additional consequences; Sketch-n-Sketch users have no language tooling (e.g., libraries, debuggers, and performance profilers) to take advantage of outside of what the system provides. In contrast, `cartokit` intentionally targets a general-purpose language, JavaScript, with a sophisticated language tooling ecosystem. Users can take `cartokit`-generated programs and execute, debug, profile, or extend them in any environment supporting JavaScript. Second, Sketch-n-Sketch uses an evaluation model where changes to the SVG output yield program transformations that are applied to the source program. In the case of multiple candidate transformations, users pick which one to apply and Sketch-n-Sketch re-executes the program. In contrast, `cartokit` maintains an intermediate representation that models the map users interact with in the interface. Edits to the map using the **Properties Panel** update the IR, which in turn triggers (1) a reconciliation algorithm that ensures updates are propagated to the map, and (2) a code generation algorithm that produces the output JavaScript program.

Direct manipulation design software systems, such as Adobe Illustrator and Sketch, support modest code generation to aid designers in transitioning the visual appearance of static media to programs. Figma's Dev Mode [47] is a recent step forward in this space, supporting compilation of visual assets to CSS, SwiftUI, or Jetpack Compose code. In these contexts, code generation operates at the level of individual components, providing small, focused snippets for handling layout and appearance of one element at a time. In contrast, `cartokit` provides the *entire* program needed to produce the visual output (i.e., the map) the user interacts with. Beyond the layout and appearance of layers, `cartokit` handles data imports, data transformations, calls to external libraries, asynchronous loading and rendering of the map and layers, and more.

7.3 System Design

In this section, we explore `cartokit`'s design in greater depth, focusing on three novel contributions that distinguish it from existing direct manipulation interfaces for cartographic design: cross-geometry transitions, user-defined data transformations, and code generation to a general purpose language.

Cross-Geometry Transitions

Existing direct manipulation interfaces for cartographic design restrict the possible maps users can create based on the geometry of input datasets. To explore alternate cartographic representations, users must transform their data externally using GIS software or geospatial analysis libraries. This separation of transformation and visualization capabilities not only forces data transfer across tools, but also requires starting cartographic design work from scratch for each new map.

`cartokit` addresses this challenge through non-destructive, cross-geometry map type transitions via the **Map Type** `select` in the **Properties Panel**. The **Map Type** `select` is the core interface element in `cartokit`'s design, allowing users to switch cartographic representations with a single click. Unlike related systems, `cartokit` can also transition between cartographic representations of differing geometry types. For example, users uploading `Polygon` data can transition that data from `Polygon`-based map types (e.g., FILL, CHOROPLETH) to `Point`-based map types (e.g., PROPORTIONAL SYMBOL, DOT DENSITY) and back. Critically, `cartokit` preserves visual styles across transitions when those styles are not affected by the choice of map type. This means that lower-level, in progress design work is preserved even as the higher-level cartographic representation changes.

To support cross-geometry transitions, `cartokit` introduces novel, custom geospatial transformation algorithms. Internally, I implemented the transformation algorithms using a mixture of TypeScript and calls to library functions from Turf.js [25]. These algorithms take into account the source and target map type, as well as the initial and current geometry types of the user's dataset, to determine (1) whether a given map type transition is possible and (2) how to transform features of the input geometry type to the output geometry type. Listing 7.1 provides an example of one such algorithm, `transformDotDensity`, which transitions a `Polygon` or `MultiPolygon` dataset to a Dot Density map, a `Point`-based cartographic representation. Geospatial transformations applied to the original dataset are serialized and stored in a `transformations` array in the `cartokit` IR. The `transformations` array tracks provenance information for the currently rendered data; given the original dataset, applying the stored transformations in sequence reproduces the current map exactly. During code generation, these transformations are inserted as `function`s in the output JavaScript program. We also generate a `flow` call, which applies the transformations in sequence.

`cartokit`'s map type transitions are also non-destructive—users can recover their original maps even after initiating a transition. To accomplish this, `cartokit` maintains two representations of a layer's data in memory: the currently rendered version and the originally uploaded version. When computing the set of possible map type transitions, the system takes into account both data representations. For example, after transitioning a Fill layer to a Point layer, `cartokit` has two representations of the layer's data: (1) the original `Polygon`s and (2) the `Point`s corresponding to those `Polygon`s' centroids. If a user then transitions the Point layer to a Choropleth layer, `cartokit` will reuse the `Polygon`s of the original data. In practice, this means that map type transitions that are either impossible or destructive in existing systems—because they overwrite the source data's geometry—are possible in `cartokit`.

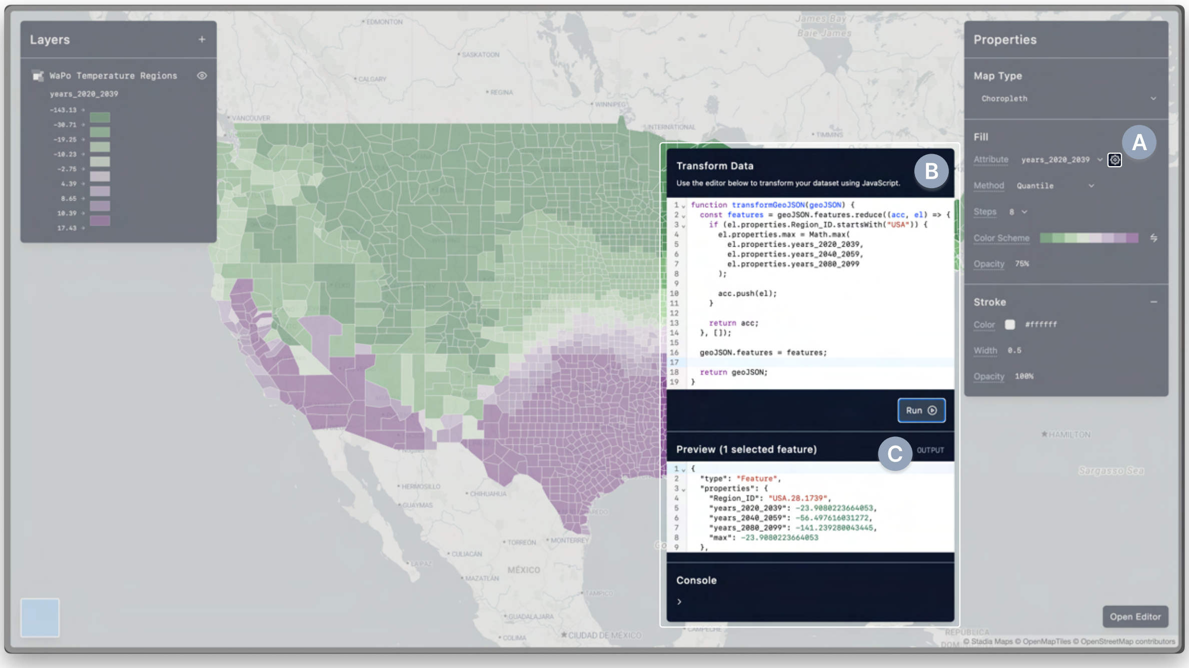
```
function transformDotDensity(layer) {
  // Obtain the layer's source and current geometry types.
  const sourceGeometryType = getLayerGeometryType(layer.data.sourceGeoJSON);
  const geometryType = getLayerGeometryType(layer.data.geoJSON);

  // If the source geometry type is not "Polygon" or "MultiPolygon", we cannot transition
  // to a Dot Density map - we don't have the requisite geometry.
  if (sourceGeometryTye !== "Polygon" && sourceGeometryType !== "MultiPolygon") {
    throwUnsupportedTransitionError(sourceGeometryType, "Polygon");
  }

  // If the current geometry type is not "Polygon" or "MultiPolygon", use the source Polygon
  // geometry from the original GeoJSON. Otherwise, use the current GeoJSON.
  const features = geometryType !== "Polygon" && geometryType !== "MultiPolygon"
    ? layer.data.sourceGeoJSON.features
    : layer.data.geoJSON.features;
  const attribute = getLayerAttribute(layer);
  const dotValue = deriveDotDensityStartingValue(features, attribute);

  const dots = features.flatMap((feature) => {
    // Compute the number of dots to generate for each Polygon feature based on the
    // attribute being visualized and the specified dot value.
    const numPoints = Math.floor(
      feature.properties?.[attribute] / dotValue
    );
    // Obtain the bounding box of the Polygon feature.
    const bbox = turf.bbox(feature);
    const selectedFeatures = [];

    // Generate random points within the bounding box of the Polygon feature, stopping
    // once we've generated the requisite number of points.
    while (selectedFeatures.length < numPoints) {
      // Generate a random point within the bounding box of the Polygon feature.
      const candidate = turf.randomPoint(1, { bbox }).features[0];

      // Verify the generated point is within the Polygon feature, not just its bbox.
      if (turf.booleanWithin(candidate, feature)) {
        selectedFeatures.push(candidate);
      }
    }

    // Return a FeatureCollection of the generated points, each with the same set of
    // attributes as the original Polygon feature.
    return selectedFeatures.flatMap((point) => turf.feature(point.geometry, feature.properties));
  });

  return turf.featureCollection(dots);
}
```

Listing 7.1: `cartokit`**'s algorithm for transitioning to a Dot Density map.** The algorithm first examines the source and current geometry types to determine if the transition is possible. From there, the algorithm generates a specific number of dots within the source `Polygon` based on the attribute being visualized.

User-Defined Data Transformations

`cartokit` users primarily transform the geometry of their geospatial datasets using the **Map Type** `select`. However, my observations with participants revealed that many geospatial analyses require additional computation over both a dataset's geometry and tabular attributes (Section 5.3). These computations vary widely in complexity, ranging from single-predicate filters to arithmetic expressions deriving new attributes to advanced aggregations and geostatistical analyses. To facilitate flexible, in situ data transformations, `cartokit` provides a **Data Transformation Panel** that allows users to author and apply arbitrary JavaScript functions to their GeoJSON datasets.

I implemented the **Data Transformation Panel** as a Codemirror [58] editor accessed via the **Properties Panel** (Figure 7.4). When a user opens the **Data Transformation Panel**, `cartokit` renders the editor pre-filled with a `function` (`transformGeoJSON`) tak-



Use the editor below to transform your dataset using JavaScript.

```
function transformGeoJSON(geoJSON) {
  const features = geoJSON.features.reduce((acc, el) => {
    if (el.properties.Region_ID.startsWith("USA")) {
      el.properties.max = Math.max(
        el.properties.years_2020_2039,
        el.properties.years_2040_2059,
        el.properties.years_2080_2099
      );

      acc.push(el);
    }

    return acc;
  }, []);

  geoJSON.features = features;

  return geoJSON;
}
```

Preview (1 selected feature)

```
{
  "type": "Feature",
  "properties": {
    "Region_ID": "USA.28.1739",
    "years_2020_2039": -23.9080223664053,
    "years_2040_2059": -56.497616031272,
    "years_2080_2099": -141.239280043445,
    "max": -23.9080223664053
  },
```

Map Type: Choropleth
Fill
Attribute: years_2020_2039
Method: Quantile
Steps: 8
Color Scheme
Opacity: 75%
Stroke
Color: #ffffff
Width: 0.5
Opacity: 100%

Figure 7.4: `cartokit`**'s Data Transformation Panel.** (A) Users click the gear icon next to the **Attribute** `select` to open the **Data Transformation Panel**. (B) Users can write arbitrary JavaScript functions to transform a layer's GeoJSON. In this example, a user employs `Array.prototype.reduce` to (1) filter features whose `Region_ID` starts with the string `''USA''` and (2) compute the maximum of three of its attributes, stored in a new property `max`. (C) The **Preview** shows the result of applying the function to the layer's GeoJSON on the currently selected feature. `cartokit` computes the result on every keystroke.

ing a single argument (`geoJSON`) corresponding to the layer's GeoJSON dataset. From here, the user can modify the body of the function to specify a transformation. For example, to combine filtering and attribute computation in a single function, a user can call `Array.prototype.reduce` on `geoJSON.features`. Within the reducer callback, they can remove features failing a given predicate and compute a new attribute that is added to each feature's `properties` `Object` (see the example in Figure 7.4). Users execute transformations by clicking the **Run** `button`, which sends the function body and GeoJSON to a Web Worker for execution. Similar to the benefits of parsing datasets in a worker, running transformations in a worker frees the main thread to continue processing user interaction. This is an important optimization for long-running computations or extremely large datasets, both of which are common in geospatial analysis contexts. If no errors occur while running user code, the data is transformed, serialized, and sent back to the main thread, where it replaces the layer's previous data in application state; `cartokit` also displays a success message. If the transformation results in a runtime error, the computation terminates and the error's message, line number, and column number in the source program are displayed to the user.

Beyond the code editor for defining data transformations, the **Data Transformation Panel** contains two additional interface elements to assist users in authoring custom transformations. First, the **Console** displays output printed by a user via `console.log` statements in their transformation code. This provides modest—though important—debugging capabilities, allowing users to inspect transformation behavior at the level of individual expressions or statements. Second, the **Preview** displays a live GeoJSON representation of running the transformation on the currently selected feature, giving users an "always on" view of the transformation's output using a concrete example. My observations of output-centered hypothesis testing (Section 5.3) and data subsetting to accelerate the transform-inspect-modify loop (Section 5.2) heavily influenced the **Preview** design. `cartokit` smooths and automates these interactions by (1) paring down the dataset to a single feature when executing the transformation, supporting fast generation of output, and (2) displaying the tabular data of the output in near real-time. Furthermore, the **Preview** provides a glimpse of what a live programming interaction for data transformation could look like in geospatial analysis environments (Section 6.2).

To our knowledge, `cartokit` is the first direct manipulation programming environment for cartographic design to support arbitrary, in situ, user-defined data transformations. Notably, the choice to support JavaScript as the transformation language opens up classes of transformations that cannot be expressed by other tools. Unlike Mapbox Studio and Felt, which use small DSLs with custom interpreters for a restricted set of data transformations, users can perform any computation that can be expressed in JavaScript. In addition, users have access to the full JavaScript ecosystem via dynamic `import` statements, which allows access to third-party libraries. This is particularly useful for bringing in packages to assist with more complex geospatial transformations or geostatistical analyses.

Compilation to a General-Purpose Language

A central goal of cartokit's design is to help users reach programs that (1) are difficult to write by hand, (2) use a programming language and graphics engine they are already familiar with, and (3) can be modified, executed, and deployed outside of the system. Toward this goal, cartokit's code generation algorithm targets JavaScript programs using Mapbox GL JS as the graphics engine. This pairing was widely adopted among my participants focusing on visualizing geospatial data, especially those in the data journalism community. In addition, using JavaScript as the compilation target improves the portability and extensibility of cartokit's output. Users can copy the generated program out of cartokit and modify it in a text editor, web-based IDE, or computational notebook to reach map types or interactions currently unsupported by the system. Users can also take advantage of the vast JavaScript ecosystem when modifying programs, bringing in additional libraries as needed. This is particularly important for users who need to integrate programs generated by cartokit with broader web site infrastructure, interface layouts, and deployment pipelines.

Compiling to a general-purpose language like JavaScript distinguishes cartokit from other direct manipulation interfaces for cartographic design, which compile to custom JSON DSLs. Style specifications in these DSLs only describe the visual appearance of layers, providing a small portion of the full program needed to render the map. They do not handle tasks like importing data and libraries, instantiating and rendering the map instance, loading data sources and layers asynchronously, or wiring up event listeners. In contrast, cartokit provides a *complete* program to reproduce the map designed within the system, which can be run unmodified in any JavaScript environment. Achieving this functionality significantly complicates the code generation process. For example, cartokit needs to track and reference identifiers mapped to imported data, selectively import library functions for cross-geometry transitions, compose user-defined transformations, inject user-defined transformations within the appropriate lexical scope, handle asynchronous data fetching, and handle asynchronous layer rendering. In the following sections, we describe how cartokit approaches this problem through the design of its intermediate representation and code generation algorithm.

Intermediate Representation

The cartokit intermediate representation is a JavaScript Object wrapped by a Svelte store. Svelte stores are a framework abstraction for asynchronous state management, allowing any application component to both publish a new state and subscribe to state updates. In cartokit, the store abstraction allows any interface element to (1) modify the IR and (2) react to IR updates. In practice, every interface element defines its own function specifying how user interactions with the element should update the IR. Likewise, every interface element defines which portion of the IR it reads and how it should update the user interface when that portion changes.

At the root level, the IR stores information on the map's center coordinates, zoom level, and basemap. In addition, it maintains a dictionary of the map's layers, mapping layer identifiers to layer definitions. Each layer definition follows a standardized interface consisting of five properties: `id`, `displayName`, `type`, `data`, and `style`. I discuss each of these properties below.

`id` and `displayName` Each layer is given a unique `id`, generated by `cartokit`, alongside a user-provided `displayName`. `id`s are used in the generated program to uniquely identify a Mapbox `layer` and its corresponding `source`.

`type` The `type` property corresponds to the map type of the given layer. Currently, `cartokit` supports six map types: Choropleth, Dot Density, Fill, Line, Point, and Proportional Symbol. The IR models the `type` property using a TypeScript union of string literals.

`data` The `data` property is itself an `Object` storing all information about a layer's source data. The `url` property within the `Object` stores the endpoint from which the GeoJSON was fetched (if accessed remotely) while the `fileName` property stores the name of the GeoJSON file uploaded by the user (if accessed from disk). The `sourceGeoJSON` property stores the *original* version of the layer's GeoJSON at the time of layer creation. When cross-geometry transitions need to restore the original layer geometry, `cartokit` will use this version of the data. The `geoJSON` property stores the currently rendered version of the layer's GeoJSON. When transitioning data to a different cartographic representation or applying user-defined transformations, `cartokit` operates on this version of the data. Finally, the `transformations` property stores all cross-geometry and user-defined transformations in an array. Each transformation consists of a `name` property corresponding to its `function` name in the generated program and a `definition` property containing the body of the `function`. The code generation algorithm inspects the `transformations` array to determine the order in which to compose transformations in the generated program.

`style` The `style` property stores all information about a layer's visual appearance. The specific properties of the `style` property vary based on the map type of the layer. For example, a Fill layer's `style` property consists only of optional `fill` and `stroke` properties describing the color, opacity, and stroke width to apply to `Polygon`s in the layer. Conversely, a Choropleth layer's `style` property includes information about the attribute to visualize, the statistical method to use to set breaks in the data, the number of breaks, numeric thresholds for the breaks, and the color scheme, in addition to properties like opacity and stroke width.

As mentioned in the **Compilation** subsection of Section 7.1, the `cartokit` IR is agnostic to the choice of graphics engine and programming language used by the code generation algorithm. I achieved this agnosticism by skewing the IR design closer to higher-level cartographic concepts rather than details of any particular renderer. While this complicates code generation, it also opens up the possibility of targeting multiple language and engine "backends." I hope to explore this direction in future work to give users access to many different programs for a single map design.

Code Generation Algorithm

`cartokit`'s code generation algorithm consists of a hierarchy of functions that each operate on specific portions of the intermediate representation. Each function peeks at relevant information from the IR to generate its own program fragment—a portion of the final output program—and returns this fragment to its caller. Functions at a higher level in the hierarchy determine where program fragments generated by callees are inserted into the final generated program. Figure 7.5 provides a graph representation of the code generation algorithm's function hierarchy.

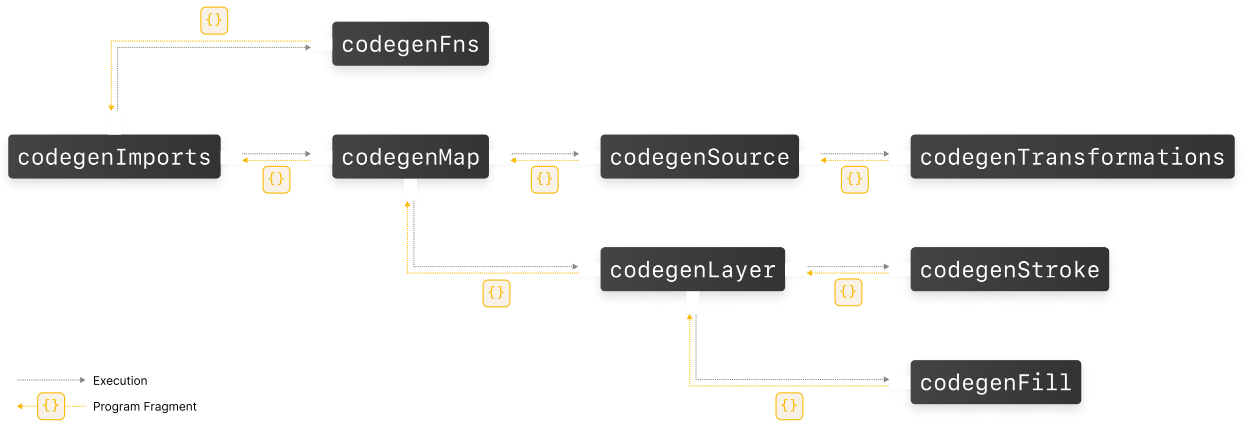


Figure 7.5: **The hierarchy of function calls in `cartokit`'s code generation algorithm.** Code generation begins with the top-level `codegenImports` function, which resolves data and library imports through analysis of the IR. Execution is passed to functions lower in the hierarchy (rightward in the graph), which generate their own program fragments and return them to callers. Callers determine where to insert these program fragments in the final output program.

During program fragment generation, some codegen functions analyze multiple parts of the IR to reason about the code to generate. For example, within `codegenMap`, `cartokit` must determine whether the callback passed to `map.on('load')` is synchronous or asynchronous; if asynchronous, `cartokit` will prepend the `async` keyword to the callback. To ascertain this information, `cartokit` iterates over the `layers` dictionary in the IR. On each layer, `cartokit` checks if (1) `layer.data.url` is defined and (2) if `layer.data.transformations.length > 0`. These two conditions together indicate that data was fetched from a remote API endpoint and transformed from its original representation either by a cross-geometry transition or a user-defined transformation. Therefore, the callback should be asynchronous to support data fetching using the browser's `fetch` API. If neither of these conditions are met by any layer, the callback can safely be marked as synchronous. Listing 7.2 shows the full algorithm, including additional details on how execution is passed from `codegenMap` to `codegenSource` and `codegenLayer`, and how `codegenMap` inserts program fragments from these callees into the program.

```
function codegenMap(map, ir) {
  // Generate program fragments for layer sources and layers.
  const layerSources = Object.values(ir.layers).reduce((p, layer) => {
    return p.concat('\n\n' + codegenSource(layer, uploadTable));
  }, '');
  const layerRenders = Object.values(ir.layers).reduce((p, layer) => {
    return p.concat('\n\n' + codegenLayer(layer));
  }, '');
  const { lng, lat } = map.getCenter();

  // Determine if the onload callback must be asynchronous by checking if
  // any layer fetches data from a remote API and transforms it locally.
  let isLoadAsync = false;
  for (const layer of Object.values(ir.layers)) {
    if (layer.data.url && layer.data.transformations.length > 0) {
      isLoadAsync = true;
    }
  }

  // Return the program fragment for instantiating the map, layer sources, and layers.
  const fragment = `
  const map = new mapboxgl.Map({
    container: 'map',
    style: '${ir.basemap.url.replace(
      PUBLIC_MAPTILER_API_KEY,
      '<YOUR_MAPTILER_API_KEY>'
    )}',
    center: [${lng}, ${lat}],
    zoom: ${map.getZoom()}
  });

  map.on('load', ${isLoadAsync ? 'async ' : ''}() => {
    ${layerSources}

    ${layerRenders}
  });
  `;

  return fragment;
}
```

Listing 7.2: **The `codegenMap` function within `cartokit`'s code generation algorithm, implemented in TypeScript.** To determine if the `map`'s `onload` callback should be asynchronous, `cartokit` checks multiple parts of the IR, including the urls and transformations of individual layers.

In addition to reasoning locally about the IR, codegen functions higher in the hierarchy perform additional analyses to derive relevant information for multiple callees. For example, `codegenImports`—the codegen function responsible for scaffolding top-level imports of libraries and data—creates a symbol table mapping layer IDs to the variable identifiers of their source data in the program. Later, in `codegenTransformations`, the compiler uses this symbol table to construct the function calls applying user-defined transformations to this identifier. Likewise, `codegenSource` references this identifier when specifying a value for a `source`'s `data` property.

7.4 Case Study: "Will global warming make temperature less deadly?"

To demonstrate the expressiveness of programming by direct manipulation in `cartokit`, I present a walkthrough showing how `cartokit` can reproduce and extend on a thematic map published by a national newsroom. Specifically, I will demonstrate how a `cartokit` user can create a modified version of the central map featured in Harry Stevens' piece, "Will global warming make temperature less deadly?" [110], published February 2023 in the Washington Post. This map is a global choropleth map showing the predicted change in average annual deaths from temperature exposure per 100,000 people between 2080 and 2099 (Figure 7.6).

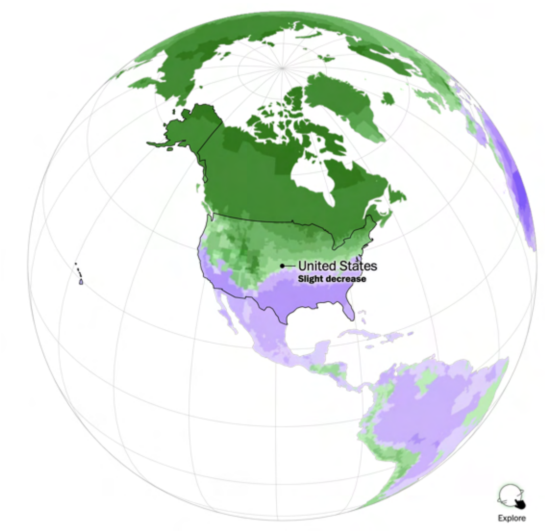
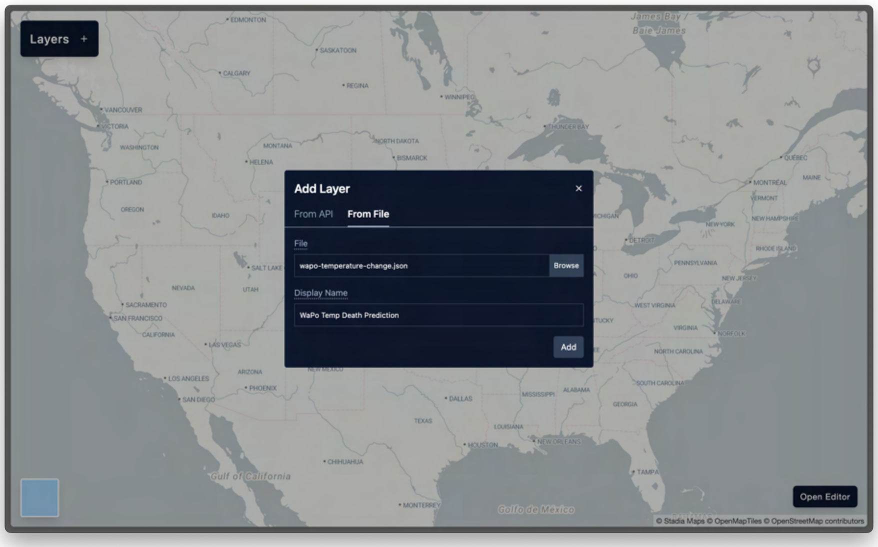


Figure 7.6: **The central map from "Will global warming make temperature less deadly?" by Harry Stevens, published in the Washington Post.** The map is a global choropleth map using a 12-break manual threshold scale and diverging green-to-purple color scheme.

The map visualizes results from Carleton et al. [16], which uses NASA's Earth Exchange climate projections [113] in an emissions stabilization scenario (Representative Concentration Pathways 4.5) to predict temperature change across 24,378 contiguous regions. The map uses a diverging green-to-purple color scheme to distinguish between regions with predicted increases versus predicted decreases in deaths. In addition, the map employs a manual threshold scale to introduce 12 discrete breaks in the data.

Uploading Data



File: wapo-temperature-change.json

Display Name: WaPo Temp Death Prediction

Figure 7.7: **Adding data to `cartokit` using the Add Layer Modal.** The user navigates to the **From File** tab to upload the GeoJSON dataset from disk. Users can also fetch data from remote API endpoints using `inputs` in the **From API** tab.

To begin reproducing the map, a `cartokit` user starts by uploading the GeoJSON dataset from disk. The dataset is 15MB in size, containing 24,378 `Polygons` each associated with four tabular attributes: the `Region_ID`, and predictions for change in deaths per 100,000 people for three separate temporal intervals: 2020–2039, 2040–2059, and 2080–2059 (represented by the attributes `years_2020_2039`, `years_2040_2059`, and `years_2080_2099`, respectively). To initiate the upload, the user clicks on the `+` button in the **Layers Panel**. This interaction opens the **Add Layer Modal**, which contains a tab-style interface with options for uploading data from an API or file on disk (Figure 7.7). The user clicks the **From File** tab, which navigates to a new tab showing two `inputs`: a **File Browser** `input` and a **Display Name** `input`. The user clicks the **File Browser** `input`, which opens a system dialog allowing them to navigate to and select the GeoJSON dataset from their local file system. Finally, they give the layer a display name of "WaPo Temp Death Prediction" using the **Display Name `input`** and click the **Add** button to add the layer to the map.

Transitioning Map Type

`cartokit` loads and renders the data on the map, with a new entry appearing in the **Layers Panel**. This entry contains the layer's display name and visibility toggle, in addition to a legend showing the layer's symbology, feature count, and geometry type. Noticing that `cartokit` rendered the data as a FILL map—applying a uniform fill and stroke to all `Polygons` in the layer—the user sets out to transition to a CHOROPLETH map. To do so, they click a `Polygon` feature on the map. By default, `cartokit` applies an outline to all features on hover to indicate they are selectable. Clicking on a `Polygon` opens the **Properties Panel**, which shows the user the current map type, fill color, fill opacity, stroke color, stroke width, and stroke opacity. From here, they use the **Map Type Select** to change the layer's map type to CHOROPLETH.

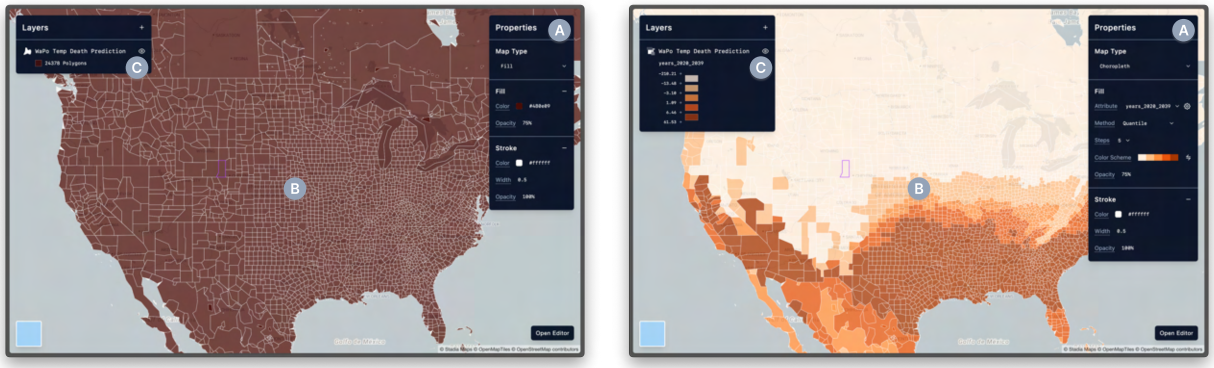


Figure 7.8: **Transitioning from a Fill map to a Choropleth map.** (A) The **Properties Panel** updates with new controls in the **Fill** section to parameterize the CHOROPLETH map. The **Stroke** section is unchanged. (B) The map updates to reflect the the map type transition, using the parameters specified in the **Properties Panel** to symbolize the data. (C) The **Layers Panel** updates the legend to show the visualized attribute, as well as the lower bound, upper bound, and color for each break in the data.

This interaction leads to several changes across the `cartokit` interface. In the **Properties Panel**, the **Fill** section renders new controls, including: (1) an **Attribute `select`**, (2) a **Method `select`**, (3) a **Steps `select`**, and (4) a **Color Scheme `select`**. Together, these controls parameterize the CHOROPLETH map. The **Attribute `select`** allows the user to select the tabular attribute to visualize. In this instance, `cartokit` infers the `years_2020_2039` attribute as the first numeric attribute in the dataset and selects it for visualization. The

Method `select` and **Steps** `select` give the user controls for computing statistical breaks in continuous numerical data. `cartokit` uses a `Quantile` binning method with `5` steps by default, which corresponds to the 5-quantiles of the `years_2020_2039` attribute across all 24,378 `Polygons`. Finally, the `Color Scheme select` allows the user to specify the output color range. `cartokit` applies a sequential white-to-orange color scheme with five colors, each of which corresponds to a given quantile. Notably, the **Stroke** section of the **Properties Panel** does not change. Because the transition from a Fill layer to a Choropleth layer does not modify the stroke in any way, `cartokit` preserves the previous default value.

On the map, each `Polygon` has been colored by (1) determining which quantile its `years_2020_2039` attribute falls into and (2) mapping this value to the appropriate color in the color scheme. In the **Layers Panel**, the legend has updated to show the upper bound, lower bound, and color of each break in the data. Figure 7.8 shows the state of the interface before and after the transition from a Fill map to a Choropleth map.

Iterating on Map Styles

With the core map type in place, the user begins iterating on the parameters of the Choropleth map. First, they use the **Attribute** `select` to pick the `years_2080_2099` attribute for visualization. Next, to match the color scheme of the original map, they use the **Color Scheme** `select` to choose a diverging purple-green color scheme. `cartokit` provides access to all D3 color schemes, including Color Brewer [11] palettes commonly used in cartographic design. Because the `cartokit` scheme ranges purple-to-green, but the original map's scheme ranges green-to-purple, the user clicks the ⇌ `button` to reverse the color scheme.

With the attribute and color scheme set, the user next modifies the steps and method used to set breaks in the data. First, they increase the number of breaks to the maximum of `9` using the **Steps** `select`. To mimic the custom threshold scale used in the original map, they use the **Method** `select` to switch from a `Quantile` scale to a `Manual` scale. This opens the **Set Stops Panel**, which allows the user to edit the intervals assigned to each color in the Choropleth layer. They modify the upper bound of each interval manually while the system modifies the lower bound of the adjacent interval to match. Finally, noticing that the original map applies no stroke to features, the user clicks the `-` button in the **Stroke** section to remove the stroke. With these tweaks, the user has reproduced a close approximation of the original map. Figure 7.9 depicts the full sequence of user styling interactions, showing the state of the interface after each change.

Transforming Data

Aiming to move beyond reproducing the original map, the user decides to visualize a new attribute that does not exist yet in the dataset—the variance of the `years_2020_2039`, `years_2040_2059`, and `years_2080_2099` attributes. Their goal is to uncover which regions of the dataset have the lowest variance, suggesting that predicted changes in deaths due to temperature exposure are similar across the three time intervals. To compute the variance,

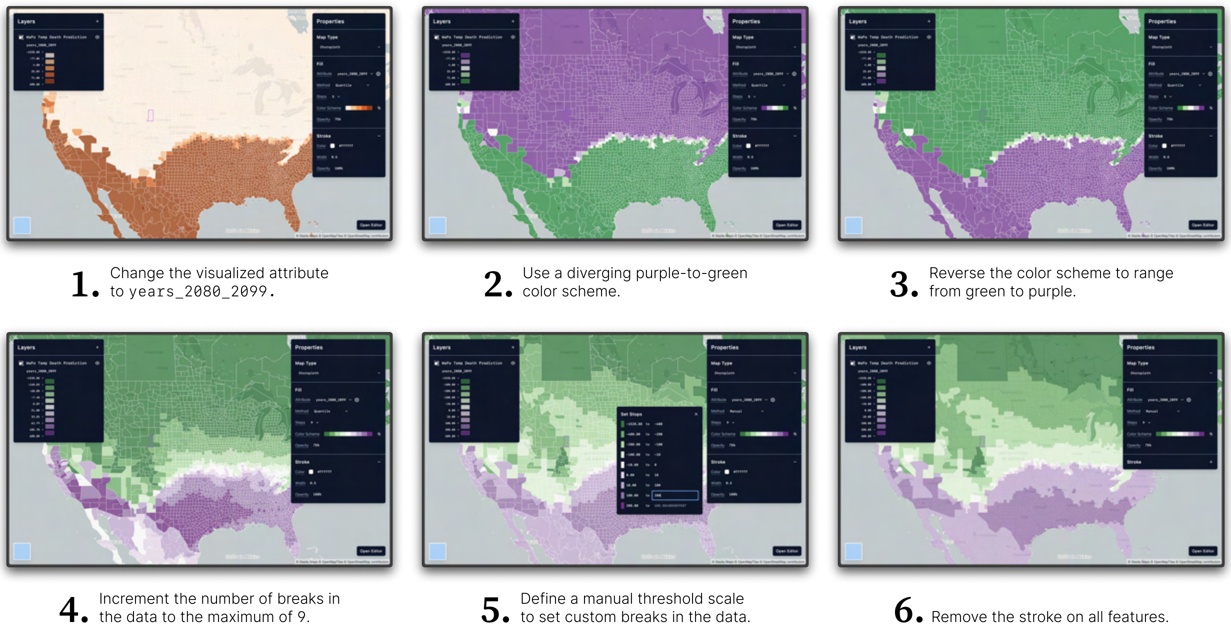

1. Change the visualized attribute to years_2080_2099.
2. Use a diverging purple-to-green color scheme.
3. Reverse the color scheme to range from green to purple.
4. Increment the number of breaks in the data to the maximum of 9.
5. Define a manual threshold scale to set custom breaks in the data.
6. Remove the stroke on all features.

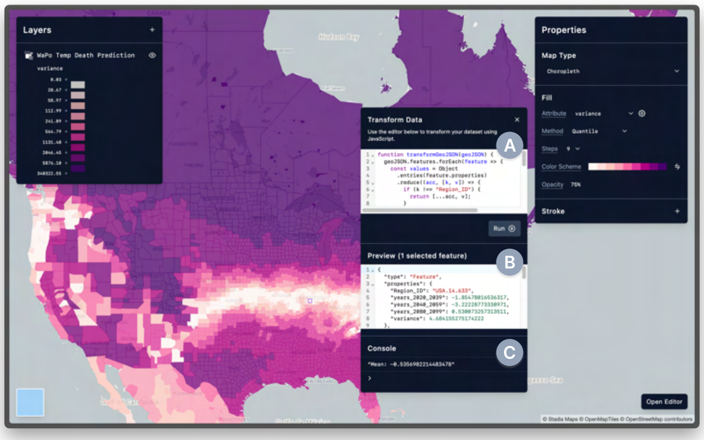
Figure 7.9: **Iterating on parameters of the Choropleth map.** The user explores six variants of a Choropleth map using controls in the **Properties Panel**. Each change modifies the map immediately, allowing the user to move efficiently toward their target map.

the user opens the **Data Transfromation Panel** by clicking the gear icon next to the **Attribute** `select`.

Within the **Data Transformation Panel**, the user starts modifying the body of the `transformGeoJSON` `function`. On each key stroke, `cartokit` applies the `function` to the single selected feature in the dataset and renders a live preview of the GeoJSON result. In this instance, the user implements an algorithm to compute variance by hand (Listing 7.3); however, they could have also used a dynamic `import` statement to bring in a third-party library to assist with the computation. While constructing the algorithm, the user periodically drops in `console.log` statements to verify intermediate values. The evaluated values of the logged expressions appear in the **Console** section of the **Data Transformation Panel**.

```
function transformGeoJSON(geoJSON) {
  geoJSON.features.forEach((feature) => {
    const values = Object
      .entries(feature.properties)
      .reduce((acc, [k, v]) => {
        if (k !== "Region_ID") {
          return [...acc, v];
        }

        return acc;
      }, []);

    const sum = values.reduce((acc, el) => acc + el, 0);
    const mean = sum / values.length;
    const sumSquares = values.reduce((acc, el) => acc + Math.pow(el - mean, 2), 0);

    feature.properties.variance = sumSquares / values.length;
  });

  return geoJSON;
}
```

Listing 7.3: **Computing variance.** The user implements a custom algorithm for computing the variance of the `years_2020_2039`, `years_2040_2059`, and `years_2080_2099` attributes in the body of `transformGeoJSON function`.



```
function transformGeoJSON(geoJSON) {
  geoJSON.features.forEach(feature => {
    const values = Object
      .entries(feature.properties)
      .reduce((acc, [k, v]) => {
        if (k !== "Region_ID") {
          return [...acc, v];
        }
```

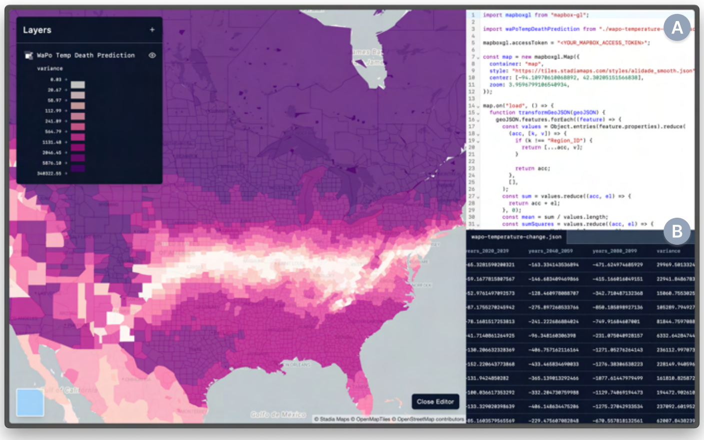
```
{
  "type": "Feature",
  "properties": {
    "Region_ID": "USA.14.633",
    "years_2020_2039": -1.85478016536317,
    "years_2040_2059": -3.22228773338971,
    "years_2080_2099": 0.530073257313511,
    "variance": 4.684155275174222
  },
```

```
"Mean: -0.5356982214483478"
>
```

Figure 7.10: **Visualizing variance on the Choropleth map.** (A) The user computes a new `variance` attribute using the JavaScript code editor in the **Data Transformation Panel**. (B) `cartokit` displays the output of running the user-defined variance algorithm on the currently selected feature in the **Preview**. (C) The **Console** displays the output of `console.log` statements inserted in the transformation.

Once the user finishes writing the algorithm, they inspect the value of the new `variance` property in the **Preview** to see if it matches their expectation. Satisifed with the result, they click the **Run** `button` to apply the function to all features in the dataset. The user then clicks on the **Attribute** `select` back in the **Properties Panel** and selects the newly computed `variance` attribute for visualization, with the map updating accordingly. Given that they are still using the `Manual` threshold scale from the previous map, they use the **Method** `select` to switch back to `Quantile`. Additionally, since variance values are always positive, they switch from a diverging color scale to a sequential white-to-purple scale. Figure 7.10 shows the final map, with the user-defined transformation, GeoJSON preview, and console output displayed in the **Data Transformation Panel**. As a final step, they move to copy the generated program out of `cartokit` into their development environment.

Accessing the Program



```
import mapboxgl from "mapbox-gl";

import waPoTempDeathPrediction from "./wapo-temperature-

mapboxgl.accessToken = "<YOUR_MAPBOX_ACCESS_TOKEN>";

const map = new mapboxgl.Map({
  container: "map",
  style: "https://tiles.stadiamaps.com/styles/alidade_smooth.json"
  center: [-94.10970610068892, 42.3020515156683B],
  zoom: 3.9596799106540934,
});

map.on("load", () => {
  function transformGeoJSON(geoJSON) {
    geoJSON.features.forEach((feature) => {
      const values = Object.entries(feature.properties).reduce(
        (acc, [k, v]) => {
          if (k !== "Region_ID") {
            return [...acc, v];
          }

          return acc;
        },
        [],
      );
      const sum = values.reduce((acc, el) => {
        return acc + el;
      }, 0);
      const mean = sum / values.length;
      const sumSquares = values.reduce((acc, el) => {
```

wapo-temperature-change.json

ears_2020_2039	years_2040_2059	years_2080_2099	variance
-65.3281590200321	-163.334143536094	-471.624974685929	29969.5013324
-59.1677815807567	-146.683409469866	-415.166016049151	22941.8486783
-52.9761497092573	-128.460978088707	-342.710487132368	15060.7553025
-87.1755270245942	-275.897268533766	-850.185898927136	105289.794927
-78.1681517253813	-241.222686884024	-749.91684687001	81844.7597088
-41.7140861264925	-96.348160306398	-231.075840928157	6332.64284744
-130.206632328369	-406.757162116164	-1271.05276264143	236112.997073
-152.220643773868	-433.465834690033	-1276.38386538223	228149.940596
-131.9424850282	-365.139013292466	-1077.61447979499	161810.825872
-100.036617353292	-332.284730759988	-1129.74869194473	194472.902610
-133.329020398639	-406.148634475206	-1275.27042933534	237092.601952
-85.1603579565569	-229.475607082848	-670.557818132561	62087.8438239

Figure 7.11: **Accessing the generated program.** (A) The user inspects and copies the program corresponding to their variance map. (B) The user can view and sort the tabular attributes of their geospatial data in the **Data Table**, including the newly computed `variance` attribute.

To access the program generated by `cartokit`, the user clicks the **Open Editor** `button`. This interaction opens the **Code Viewer** and **Data Table**. The user notices the **Data Table** contains a new column, `variance`, corresponding to the variance value computed by their transformation. In addition, they inspect the program in the **Code Viewer** and see that their transformation has been inserted into the program source code. They complete their interaction with `cartokit` by copying this program from the **Code Viewer** to their development environment. Figure 7.11 shows the **Code Viewer** and **Data Table** and Listing 7.4 shows the full program generated by `cartokit`.

```
import mapboxgl from "mapbox-gl";

import waPoTempDeathPrediction from "./wapo-temperature-change.json";

mapboxgl.accessToken = "<YOUR_MAPBOX_ACCESS_TOKEN>";

const map = new mapboxgl.Map({
  container: "map",
  style: "https://tiles.stadiamaps.com/styles/alidade_smooth.json",
  center: [-94.10970610068892, 42.30205151566838],
  zoom: 3.9596799106540934,
});

map.on("load", () => {
  function transformGeoJSON(geoJSON) {
    geoJSON.features.forEach((feature) => {
      const values = Object.entries(feature.properties).reduce(
        (acc, [k, v]) => {
          if (k !== "Region_ID") {
            return [...acc, v];
          }

          return acc;
        },
        [],
      );
      const sum = values.reduce((acc, el) => {
        return acc + el;
      }, 0);
      const mean = sum / values.length;
      const sumSquares = values.reduce((acc, el) => {
        return acc + Math.pow(el - mean, 2);
      }, 0);

      feature.properties.variance = sumSquares / values.length;
    });

    return geoJSON;
  }

  map.addSource("wa-po-temp-death-prediction__1", {
    type: "geojson",
    data: transformGeoJSON(waPoTempDeathPrediction),
  });

  map.addLayer({
    id: "wa-po-temp-death-prediction__1",
    source: "wa-po-temp-death-prediction__1",
    type: "fill",
    paint: {
      "fill-color": [
        "step",
        ["get", "variance"],
        "#fff7f3",
        20.669645900034187,
        "#fde0dd",
        58.96717040855061,
        "#fcc5c0",
        112.99252150319008,
        "#fa9fb5",
        241.8917213513634,
        "#f768a1",
        564.7926377701071,
        "#dd3497",
        1131.481855736475,
        "#ae017e",
        2046.4537964541362,
        "#7a0177",
        5876.09592446122,
        "#49006a",
      ],
      "fill-opacity": 0.75,
    },
  });
});
```

Listing 7.4: **The program generated by `cartokit`.** `cartokit` generates a complete JavaScript program to reproduce the variance Choropleth map the user created.

Chapter 8

Limitations and Future Work

In this chapter, I discuss the limitations of both my contextual inquiry study and the system design of `cartokit`. I also highlight opportunities for future research and engineering efforts to support domain experts in their work with geospatial data.

8.1 Contextual Inquiry Study

In my contextual inquiry study, I identified shared challenges and computing needs of geospatial data users across particular disciplinary and expertise boundaries, but future research should explore additional needs beyond my selected domains and experience levels. Specifically, I recruited from three domains: Earth and climate science, the social sciences, and data journalism. Therefore, my findings may not generalize to geospatial data users outside these areas, such as epidemiologists, digital humanities researchers, or statisticians. I believe there are also opportunities for additional research *within* my chosen domains and experience groups. The number of participants from any given subgroup of my participant pool is too small to reveal insights about the needs of each class independently.

In general, the sample size of my study ($n = 25$) limits my ability to make quantitative claims about the prevalence of my findings in a broader population [90]. Furthermore, qualitative research experts warn about the risks of quantifying qualitative data [33]. For these reasons, I intentionally avoided attempts to generalize my findings beyond my participant group. Instead, I hope they can serve as a basis for designing larger-scale studies to assess the prevalence of my identified challenges in the wider community of geospatial data users.

Using contextual inquiry with open task selection provided me with rich detail and context on participants' challenges but also came with drawbacks. First, asking participants to narrate their thought processes while performing cognitively challenging tasks can make these tasks more difficult [32]. Thus, tasks may appear harder in a lab setting than in a non-observational context. I attempted to mitigate this effect by permitting participants to pause narration until they reached a stopping point for discussion. Second, asking participants to work on their own tasks inhibits me from making comparative claims that a fixed-task design may support. For example, alternative studies could compare participant performance on fixed tasks across multiple tools to understand their relative strengths. Additionally, my study design does not assess whether participants' tasks are representative of the work of geospatial data users more generally. Validating task representativeness through additional studies would bolster my findings.

A critical next step for this research is to validate my design opportunities with domain experts. I followed the practice of other contextual inquiry studies that derive design opportunities directly from participant observation [66, 26, 12]. Indeed, a strength of contextual inquiry is that it exposes unforeseen participant challenges in situ, allowing me to identify needs that may not become visible in alternative methods relying on participants' memories of their work. However, this does not erase the threats of researcher confirmation bias [91] or causal error [94]. Future work can test my design opportunities through expert interviews, large-scale surveys, and formative studies of new systems.

8.2 `cartokit` System Design

Restricted Geospatial Data Models

`cartokit` currently can only process vector geospatial data in the GeoJSON interchange format, which inherently limits the class of maps users can create with the system. Raster geospatial data–including multispectral satellite imagery, Light Detection and Ranging (LIDAR) data, Synthetic Aperture Radar (SAR) data, and other forms—played a critical role in many participants' workflows, making it a clear target for support in `cartokit`. However, handling raster data would require significant alterations to the system's data management strategy. Currently, `cartokit` stores GeoJSON data in memory, which can be feasible for datasets in the <100MB range. Raster data, by contrast, can range anywhere from >100MB to multiple GBs in size depending on the spatial extent, spatial resolution, and number of bands. At this scale, storing and analyzing raster data entirely on the client is infeasible. For a web application like `cartokit`, shifting raster data storage to a spatial database (e.g., PostGIS [99], SpatialLite [48]) and raster analysis to a web server would be necessary.

A lower lift with the current architecture would involve supporting other vector geospatial formats. In particular, the Shapefile and Keyhole Markup Language (KML) formats are widely used vector interchange formats among GIS practitioners. Supporting these formats natively or enabling automatic conversion to GeoJSON within `cartokit` would reduce the amount of data preparation users must perform before interacting with the system.

Data Transformation

Currently, `cartokit` uses library functions from Turf.js in its geospatial transformation algorithms. Because Turf.js is implemented entirely in TypeScript, `cartokit` can perform all geospatial transformations client-side in the browser, either on the main thread or within a Web Worker. While obviating the need for a client-server architecture, performing data transformation on the client also increases memory pressure on users' devices. In addition, both my algorithms and Turf.js's library functions often iterate over all `Features` in a GeoJSON `FeatureCollection`, meaning that performance degrades with increasing dataset size.

A possible alternative would be to shift data transformation to an in-memory spatial database, such as DuckDB [101] paired with its Spatial Extension. Spatial databases are optimized to perform the kinds of computations `cartokit` currently offloads to Turf.js while offering better performance. In addition, employing an in-memory spatial database could address some of the challenges around file formats discussed above. DuckDB's Spatial Extension, PostGIS, and SpatialLite all natively support a diverse set of vector geospatial file formats and can interoperate between them.

Bidirectionality

Currently, `cartokit` does not allow users to edit the JavaScript program produced by its code generation algorithm. The only ways to modify the program within the system are through (1) direct manipulation interactions using the controls in the **Properties Panel** or (2) authoring transformations in the **Data Transformation Panel**. This restriction is in part a consequence of the system architecture. Rather than generating a program and subsequently executing it to render the output, `cartokit` instead maps user interactions to IR updates. From here, the reconciler and code generation algorithm transform the IR update into a map update and a program update, respectively, keeping the two in sync. To support arbitrary edits to the JavaScript program with the current architecture, `cartokit` would need to determine how to map users' program edits back to the IR.

While making the generated program read only is a drawback, it also comes with several advantages. First, it is impossible for users to create syntactically invalid programs. Even in the case where users write syntactically invalid data transformations using the **Data Transformation Panel**, `cartokit` prevents these transformations from making their way into the IR. Second, it allows us to instrument the map with additional interaction code without worrying about collisions with user code. For example, `cartokit` renders outline layers atop each user-defined layer that only become visible on feature hover and feature selection. These effects helps to bring familiar design software interactions to `cartokit`, but could easily be overwritten by user code defining custom `mouseover` or `click` event handlers.

Still, making the JavaScript program editable would have significant benefits for `cartokit` users. Modifying choices made by `cartokit`'s code generation algorithm cannot currently be done without moving the program out of the system and editing it manually. To address the challenge of bidirectional editing, future work on `cartokit` could start by shifting to an execution model in which the generated program is run directly within the system. Supporting this behavior would additionally require (1) analyzing the program to recover values to populate direct manipulation controls and (2) synthesizing program repairs when a direct manipulation update occurs. Inspiration could be taken from Mayer et al.'s [83] work on bidirectional evaluation in direct manipulation systems. They develop the notion of an evaluation update relation that, given a change to the program output v', rewrites the program e to e' to reconcile the change. Defining such a relation for `cartokit` could help address challenge (2) above, while existing program analysis techniques (e.g., program slicing) could help tackle (1).

Chapter 9

Conclusion

As geospatial data grows in scale and accessibility, domain experts require increasingly expressive tools to harness the insights hidden in this data. But what could such tools look like, and what challenges should they address? This thesis aims to answer this question by (1) deepening our understanding of the computing needs of domain expert geospatial data users and (2) presenting a novel direct manipulation programming system for geospatial analysis and visualization. Using contextual inquiry, I identified unreported challenges across five phases of participants' work: data discovery, data transformation, analysis, analysis representation, and visualization. For example, my work is the first to discuss how users (1) employ data subsetting and resolution reduction to speed up exploratory geospatial analysis, (2) create informal program representations to record geoprocessing workflows, and (3) observe changes to feature counts and geometry to infer geospatial operator behavior. Going beyond prior work on GIS usability, I also uncovered needs that extend to other tools used in modern geospatial workflows, including computational notebooks, design software, and geospatial analysis and visualization libraries. My observations revealed that four challenges—finding and transforming geospatial data to satisfy spatiotemporal constraints, understanding the behavior of geospatial operators, tracking geospatial data provenance, and exploring the cartographic design space—were especially difficult for participants. From these observations, I synthesized novel design opportunities for geospatial analysis and visualization systems. Finally, I used these design opportunities to guide the development of `cartokit`, an output-directed, direct manipulation programming environment for authoring interactive maps. Future work can build on these insights and system design to create useful and usable tooling that makes it easier to explore, analyze, and communicate patterns of spatiotemporal change in our environment and societies.

Bibliography

[1] QGIS Association. *24.4. The history manager — QGIS Documentation*. https://docs.qgis.org/3.22/en/docs/user_manual/processing/history.html. Accessed: 2022-06-24. 2022.

[2] QGIS Association. *24.5. The graphical modeler — QGIS Documentation*. https://docs.qgis.org/3.22/en/docs/user_manual/processing/modeler.html. Accessed: 2022-06-27. 2022.

[3] QGIS Association. *25.1.17. Vector general — QGIS Documentation*. https://docs.qgis.org/3.22/en/docs/user_manual/processing_algs/qgis/vectorgeneral.html. Accessed: 2022-06-21. 2022.

[4] QGIS Association. *QGIS Geographic Information System*. https://qgis.org/. Accessed: 2022-08-22. 2022.

[5] Marie-Aude Aufaure-Portier. "Definition of a Visual Language for GIS". In: *Cognitive Aspects of Human-Computer Interaction for Geographic Information Systems*. Ed. by Timothy L. Nyerges et al. Dordrecht, Netherlands: Springer Netherlands, 1995, pp. 163–178. DOI: 10.1007/978-94-011-0103-5_12.

[6] Shaon Barman et al. "Ringer: Web Automation by Demonstration". In: *ACM SIGPLAN Notices* 51.10 (Oct. 2016), pp. 748–764. DOI: 10.1145/3022671.2984020.

[7] Rohan Bavishi et al. "AutoPandas: Neural-Backed Generators for Program Synthesis". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), 168:1–168:27. DOI: 10.1145/3360594.

[8] Alan F. Blackwell and Thomas R.G. Green. "Notational Systems — the Cognitive Dimensions of Notations Framework". In: *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*. Ed. by John M. Carroll. Interactive Technologies. San Francisco, CA, USA: Morgan Kaufmann, 2003. Chap. 5, pp. 103–133. DOI: 10.1016/B978-155860808-5/50005-8.

[9] Matthew Bloch. *Mapshaper*. https://mapshaper.org/. Accessed: 2022-07-13. 2022.

[10] Virginia Braun and Victoria Clarke. "Using thematic analysis in psychology". In: *Qualitative Research in Psychology* 3.2 (Apr. 2006), pp. 77–101. DOI: 10.1191/1478088706qp063oa.

[11] Cynthia Brewer. *ColorBrewer*. https://colorbrewer2.org/. Accessed: 2023-11-13. 2023.

[12] Julia Brich et al. "Exploring End User Programming Needs in Home Automation". In: *ACM Transactions on Computer-Human Interaction* 24.2 (Apr. 2017), 11:1–11:35. DOI: 10.1145/3057858.

[13] U.S. Census Bureau. *American Community Survey 5-Year Estimates. S2502: Demographic Characteristics for Occupied Housing Units*. https://data.census.gov/. Retrieved from: https://data.census.gov/table?t=Owner/Renter+(Householder)+Characteristics&g=0100000US%240500000&tid=ACSST5Y2020.S2502. Type: dataset. 2020.

[14] Brian Burg et al. "Interactive Record/Replay for Web Application Debugging". In: *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*. UIST '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 473–484. DOI: 10.1145/2501988.2502050.

[15] H. Butler et al. *The GeoJSON Format*. RFC 7946. Aug. 2016. DOI: 10.17487/RFC7946. URL: https://www.rfc-editor.org/info/rfc7946.

[16] Tamma Carleton et al. "Valuing the Global Mortality Consequences of Climate Change Accounting for Adaptation Costs and Benefits". In: *The Quarterly Journal of Economics* 137.4 (Nov. 2022), pp. 2037–2105. DOI: 10.1093/qje/qjac020. (Visited on 11/11/2023).

[17] Kang-Tsung Chang. "Geographic Information System". In: *International Encyclopedia of Geography: People, the Earth, Environment and Technology*. Hoboken, NJ, USA: John Wiley & Sons, Ltd, 2019, pp. 1–10. DOI: 10.1002/9781118786352.wbieg0152.pub2.

[18] Sarah Chasins et al. "Browser Record and Replay as a Building Block for End-User Web Automation Tools". In: *Proceedings of the 24th International Conference on World Wide Web*. WWW '15 Companion. New York, NY, USA: Association for Computing Machinery, May 2015, pp. 179–182. DOI: 10.1145/2740908.2742849.

[19] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. "Rousillon: Scraping Distributed Hierarchical Web Data". In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. UIST '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 963–975. DOI: 10.1145/3242587.3242661.

[20] Ravi Chugh et al. "Programmatic and direct manipulation, together at last". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 341–354. DOI: 10.1145/2908080.2908103.

[21] Luca Cinquini et al. "The Earth System Grid Federation: An open infrastructure for access to distributed geospatial data". In: *Future Generation Computer Systems*. Special Section: Intelligent Big Data Processing 36 (July 2014), pp. 400–417. DOI: 10.1016/j.future.2013.07.002.

[22] Open Geospatial Consortium. *GeoSPARQL*. https://opengeospatial.github.io/ogc-geosparql/geosparql11/spec.html#_topology_vocabulary_extension. Accessed: 2022-12-12. 2022.

[23] MapLibre GL JS Contributors. *MapLibre GL JS*. https://maplibre.org/maplibre-gl-js/docs/. Accessed: 2023-10-23. 2023.

[24] Svelte Contributors. *Svelte*. https://svelte.dev/. Accessed: 2023-10-22. 2023.

[25] Turf.js Contributors. *Turf.js*. https://turfjs.org/. Accessed: 2023-11-06. 2023.

[26] Bronwyn J. Cumbo, Tom Bartindale, and Dan Richardson. "Exploring the Opportunities for Online Learning Platforms to Support the Emergency Home School Context". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. New York, NY, USA: Association for Computing Machinery, May 2021, pp. 1–11. DOI: 10.1145/3411764.3445044.

[27] Christopher Daly, Ronald P. Neilson, and Donald L. Phillips. "A Statistical-Topographic Model for Mapping Climatological Precipitation over Mountainous Terrain". In: *Journal of Applied Meteorology and Climatology* 33.2 (Feb. 1994). Type: dataset, pp. 140–158. DOI: 10.1175/1520-0450(1994)033<0140:ASTMFM>2.0.CO;2.

[28] Datawrapper. *Datawrapper*. https://www.datawrapper.de/. Accessed: 2023-11-10. 2023.

[29] Clare Davies and David Medyckyj-Scott. "Feet on the Ground: Studying User-GIS Interaction in the Workplace". In: *Cognitive Aspects of Human-Computer Interaction for Geographic Information Systems*. Ed. by Timothy L. Nyerges et al. Dordrecht, Netherlands: Springer Netherlands, 1995, pp. 123–141. DOI: 10.1007/978-94-011-0103-5_10.

[30] Clare Davies and David Medyckyj-Scott. "GIS usability: recommendations based on the user's view". In: *International Journal of Geographical Information Science* 8.2 (1994), pp. 175–189. DOI: 10.1080/02693799408901993.

[31] Clare Davies and David Medyckyj-Scott. "GIS users observed". In: *International Journal of Geographical Information Systems* 10.4 (June 1996), pp. 363–384. DOI: 10.1080/02693799608902085.

[32] Simon P. Davies and Adrian M. Castell. "From Individuals to Groups Through Artifacts: The Changing Semantics of Design in Software Development". In: *User-Centred Requirements for Software Engineering Environments*. Ed. by David J. Gilmore, Russel L. Winder, and Françoise Détienne. New York, NY, USA: Springer-Verlag, 1994, pp. 11–23.

[33] Norma K. Denzin and Yvonna S. Lincoln, eds. *The SAGE Handbook of Qualitative Research*. 5th. Thousand Oaks, CA, USA: SAGE Publications, Inc., 2018.

[34] Ian Drosos et al. “Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–12. DOI: 10.1145/3313831.3376442.

[35] M. Drusch et al. “Sentinel-2: ESA’s Optical High-Resolution Mission for GMES Operational Services”. In: *Remote Sensing of Environment*. The Sentinel Missions - New Opportunities for Science 120 (May 2012), pp. 25–36. DOI: 10.1016/j.rse.2011.11.026.

[36] Max Egenhofer. “User Interfaces: Front Matter”. In: *Cognitive Aspects of Human-Computer Interaction for Geographic Information Systems*. Ed. by Timothy L. Nyerges et al. Dordrecht, Netherlands: Springer Netherlands, 1995, pp. 143–145.

[37] Miguel Elias et al. ““Do I Live in a Flood Basin?” Synthesizing Ten Thousand Maps”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’08. New York, NY, USA: Association for Computing Machinery, Apr. 2008, pp. 255–264. DOI: 10.1145/1357054.1357100.

[38] Esri. *A complete listing of the Spatial Analyst tools—ArcGIS Pro Documentation*. https://pro.arcgis.com/en/pro-app/2.8/tool-reference/spatial-analyst/complete-listing-of-spatial-analyst-tools.htm. Accessed: 2022-05-13. 2022.

[39] Esri. *ArcGIS*. https://www.esri.com/en-us/arcgis/about-arcgis/overview. Accessed: 2022-08-22. 2022.

[40] Esri. *Merge (Data Management)—ArcGIS Pro Documentation*. https://pro.arcgis.com/en/pro-app/2.8/tool-reference/data-management/merge.htm. Accessed: 2022-06-21. 2022.

[41] Esri. *Viewing tool execution history—ArcMap Documentation*. https://desktop.arcgis.com/en/arcmap/latest/analyze/executing-tools/history-log-files.htm. Accessed: 2022-06-24. 2022.

[42] Munazza Fatima et al. “Geospatial Analysis of COVID-19: A Scoping Review”. In: *International Journal of Environmental Research and Public Health* 18.5 (Jan. 2021), p. 2336. DOI: 10.3390/ijerph18052336.

[43] Thore Fechner, Dennis Wilhelm, and Christian Kray. “Ethermap — Real-time Collaborative Map Editing”. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI ’15. New York, NY, USA: Association for Computing Machinery, Apr. 2015, pp. 3583–3592. DOI: 10.1145/2702123.2702536.

[44] Melanie Feinberg. “A Design Perspective on Data”. In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI ’17. New York, NY, USA: Association for Computing Machinery, May 2017, pp. 2952–2963. DOI: 10.1145/3025453.3025837.

[45] Felt. *Felt*. https://felt.com. Accessed: 2023-10-16. 2023.

[46] Felt. *Felt Style Language*. https://felt.com/blog/felt-style-language. Accessed: 2023-10-16. 2023.

[47] Figma. *Figma Dev Mode*. https://www.figma.com/dev-mode/. Accessed: 2023-11-10. 2023.

[48] Alessandro Furieri. *SpatiaLite*. https://www.gaia-gis.it/fossil/libspatialite/index. Accessed: 2023-11-17. 2023.

[49] G. David Garson and Robert S. Biggs. *Analytic Mapping and Geographic Databases*. Quantitative Applications in the Social Sciences. Newbury Park, CA, USA: SAGE Publications, Inc., 1992. DOI: 10.4135/9781412983334.

[50] GeoPandas. *GeoPandas 0.12.2*. https://geopandas.org/en/stable/. Accessed: 2023-01-06. 2022.

[51] GeoPandas. *Merging Data — GeoPandas 0.12.2*. https://geopandas.org/en/stable/docs/user_guide/mergingdata.html#attribute-joins. Accessed: 2023-01-06. 2022.

[52] Jianya Gong, Jing Geng, and Zeqiang Chen. “Real-time GIS data model and sensor web service platform for environmental data management”. In: *International Journal of Health Geographics* 14.1 (Jan. 2015), p. 2. DOI: 10.1186/1476-072X-14-2.

[53] Noel Gorelick et al. “Google Earth Engine: Planetary-scale geospatial analysis for everyone”. In: *Remote Sensing of Environment* 202 (Dec. 2017), pp. 18–27. DOI: 10.1016/j.rse.2017.06.031.

[54] Philip J. Guo et al. “Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts”. In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST ’11. New York, NY, USA: Association for Computing Machinery, Oct. 2011, pp. 65–74. DOI: 10.1145/2047196.2047205.

[55] Mordechai (Muki) Haklay and Artemis Skarlatidou. “Human-computer interaction and geospatial technologies – context”. In: *Interacting with Geospatial Technologies*. West Sussex, UK: John Wiley & Sons, Ltd, 2010. Chap. 1, pp. 1–18. DOI: 10.1002/9780470689813.ch1.

[56] Mordechai (Muki) Haklay and Antigoni Zafiri. “Usability Engineering for GIS: Learning from a Screenshot”. In: *The Cartographic Journal* 45.2 (May 2008), pp. 87–97. DOI: 10.1179/174327708X305085.

[57] HashiCorp. *Terraform*. https://www.terraform.io/. Accessed: 01-07-2023. 2023.

[58] Marijn Haverbeke. *Codemirror*. https://codemirror.net/. Accessed: 2023-11-06. 2023.

[59] Brent Hecht et al. "Geographic Human-Computer Interaction". In: *CHI '11 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 447–450. DOI: 10.1145/1979742.1979532.

[60] Brent Hecht et al. "Geographic Human-Computer Interaction". In: *CHI '13 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '13. New York, NY, USA: Association for Computing Machinery, Apr. 2013, pp. 3163–3166. DOI: 10.1145/2468356.2479637.

[61] Jeffrey Heer and Michael Bostock. "Declarative Language Design for Interactive Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (Nov. 2010), pp. 1149–1156. DOI: 10.1109/TVCG.2010.144.

[62] Christian Heipke. "Crowdsourcing geospatial data". In: *ISPRS Journal of Photogrammetry and Remote Sensing*. ISPRS Centenary Celebration Issue 65.6 (Nov. 2010), pp. 550–557. DOI: 10.1016/j.isprsjprs.2010.06.005.

[63] Brian Hempel, Justin Lubin, and Ravi Chugh. "Sketch-n-Sketch: Output-Directed Programming for SVG". In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 281–292. DOI: 10.1145/3332165.3347925.

[64] Karen Holtzblatt and Hugh Beyer. "Principles of Contextual Inquiry". In: *Contextual Design: Design for Life*. Ed. by Karen Holtzblatt and Hugh Beyer. 2nd. Interactive Technologies. Boston, MA, USA: Morgan Kaufmann Publishers, Jan. 2017. Chap. 3, pp. 43–80. DOI: 10.1016/B978-0-12-800894-2.00003-X.

[65] Sean Kandel et al. "Wrangler: Interactive Visual Specification of Data Transformation Scripts". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 3363–3372. DOI: 10.1145/1978942.1979444.

[66] Anna Kawakami et al. "Improving Human-AI Partnerships in Child Welfare: Understanding Worker Practices, Challenges, and Desires for Algorithmic Decision Support". In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 1–18. DOI: 10.1145/3491102.3517439.

[67] Mary Beth Kery, Amber Horvath, and Brad Myers. "Variolite: Supporting Exploratory Programming by Data Scientists". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. New York, NY, USA: Association for Computing Machinery, May 2017, pp. 1265–1276. DOI: 10.1145/3025453.3025626.

[68] Mary Beth Kery et al. "The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–11. DOI: 10.1145/3173574.3173748.

[69] Mary Beth Kery et al. “Towards Effective Foraging by Data Scientists to Find Past Analysis Choices”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 1–13. DOI: 10.1145/3290605.3300322.

[70] Laura Koesten et al. “Collaborative Practices with Structured Data: Do Tools Support What Users Need?” In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 1–14. DOI: 10.1145/3290605.3300330.

[71] Manolis Koubarakis et al. “Data Models and Query Languages for Linked Geospatial Data”. In: *Reasoning Web: Semantic Technologies for Advanced Query Answering*. Ed. by Thomas Eiter and Thomas Krennwallner. Lecture Notes in Computer Science. Berlin and Heidelberg, Germany: Springer-Verlag, 2012, pp. 290–328. DOI: 10.1007/978-3-642-33158-9_8.

[72] M. J. Kraak. *Cartography: Visualization of Geospatial Data*. 4th. Boca Raton, FL, USA: Routledge, 2021. DOI: 10.1201/9780429464195.

[73] Nicolas Lambert. *Bertin.js*. https://github.com/neocarto/bertin. Accessed: 2022-12-14. 2022.

[74] Jae-Gil Lee and Minseo Kang. “Geospatial Big Data: Challenges and Opportunities”. In: *Big Data Research*. Visions on Big Data 2.2 (June 2015), pp. 74–81. DOI: 10.1016/j.bdr.2015.01.003.

[75] Sorin Lerner. “Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–7. DOI: 10.1145/3313831.3376494.

[76] Gilly Leshed et al. “CoScripter: Automating & Sharing How-To Knowledge in the Enterprise”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’08. New York, NY, USA: Association for Computing Machinery, Apr. 2008, pp. 1719–1728. DOI: 10.1145/1357054.1357323.

[77] María-Jesús Lobo, Emmanuel Pietriga, and Caroline Appert. “An Evaluation of Interactive Map Comparison Techniques”. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI ’15. New York, NY, USA: Association for Computing Machinery, Apr. 2015, pp. 3573–3582. DOI: 10.1145/2702123.2702130.

[78] Steven M. Manson et al. “Using Eye-tracking and Mouse Metrics to Test Usability of Web Mapping Navigation”. In: *Cartography and Geographic Information Science* 39.1 (2012), pp. 48–60.

[79] Mapbox. *Mapbox GL JS*. https://github.com/mapbox/mapbox-gl-js. Accessed: 2023-10-15. 2023.

[80] Mapbox. *Mapbox Studio*. https://www.mapbox.com/mapbox-studio. Accessed: 2023-10-15. 2023.

[81] Mapbox. *Mapbox Style Specification*. https://docs.mapbox.com/style-spec/guides/. Accessed: 2023-10-15. 2023.

[82] Jon May and Tim Gamble. "Collocating Interface Objects: Zooming into Maps". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 2085–2094. DOI: 10.1145/2556288.2557279.

[83] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. "Bidirectional Evaluation with Direct Manipulation". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018), 127:1–127:28. DOI: 10.1145/3276497. URL: https://doi.org/10.1145/3276497.

[84] James Mickens, Jeremy Elson, and Jon Howell. "Mugshot: Deterministic Capture and Replay for Javascript Applications". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI '10. USA: USENIX Association, Apr. 2010, p. 11.

[85] Microsoft. *Microsoft Planetary Computer*. https://planetarycomputer.microsoft.com/. Accessed: 2022-09-10. 2022.

[86] Microsoft. *TypeScript*. https://www.typescriptlang.org/. Accessed: 2023-10-22. 2023.

[87] Anders Miltner et al. "On the Fly Synthesis of Edit Suggestions". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), 143:1–143:29. DOI: 10.1145/3360569.

[88] Dominik Moritz et al. "Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco". In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (Jan. 2019), pp. 438–448. DOI: 10.1109/TVCG.2018.2865240.

[89] Michael Muller et al. "How Data Science Workers Work with Data: Discovery, Capture, Curation, Design, Creation". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 1–15. DOI: 10.1145/3290605.3300356.

[90] Brad A. Myers et al. "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools". In: *Computer* 49.7 (July 2016), pp. 44–52. DOI: 10.1109/MC.2016.200.

[91] Raymond S. Nickerson. "Confirmation Bias: A Ubiquitous Phenomenon in Many Guises". In: *Review of General Psychology* 2.2 (June 1998), pp. 175–220. DOI: 10.1037/1089-2680.2.2.175.

[92] Observable. *Observable*. https://observablehq.com/. Accessed: 2022-09-11. 2022.

[93] Observable. *Observable Plot*. https://observablehq.com/plot. Accessed: 2022-12-14. 2022.

[94] Anthony J. Onwuegbuzie and Nancy L. Leech. "Validity and Qualitative Research: An Oxymoron?" In: *Quality & Quantity* 41.2 (Apr. 2007), pp. 233–249. DOI: 10.1007/s11135-006-9000-3.

[95] Jason Ott et al. "BioScript: Programming Safe Chemistry on Laboratories-on-a-Chip". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018), 128:1–128:31. DOI: 10.1145/3276498.

[96] Leysia Palen et al. "Success & Scale in a Data-Producing Organization: The Socio-Technical Evolution of OpenStreetMap in Response to Humanitarian Events". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. New York, NY, USA: Association for Computing Machinery, Apr. 2015, pp. 4113–4122. DOI: 10.1145/2702123.2702294.

[97] Fernando Perez and Brian E. Granger. "IPython: A System for Interactive Scientific Computing". In: *Computing in Science & Engineering* 9.3 (May 2007), pp. 21–29. DOI: 10.1109/MCSE.2007.53.

[98] Alenka Poplin. "How user-friendly are online interactive maps? Survey based on experiments with heterogeneous users". In: *Cartography and Geographic Information Science* 42.4 (Aug. 2015), pp. 358–376. DOI: 10.1080/15230406.2014.991427.

[99] PostGIS. *PostGIS*. https://postgis.net/. Accessed: 2023-11-17. 2023.

[100] The Selenium Project. *Selenium*. https://www.selenium.dev/. Accessed: 2022-12-06. 2022.

[101] Mark Raasveldt and Hannes Mühleisen. "DuckDB: an Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 1981–1984. DOI: 10.1145/3299869.3320212.

[102] Meredith P. Richards. "The Gerrymandering of School Attendance Zones and the Segregation of Public Schools: A Geospatial Analysis". In: *American Educational Research Journal* 51.6 (Dec. 2014), pp. 1119–1157. DOI: 10.3102/0002831214553652.

[103] Esther Rolf et al. "A generalizable and accessible approach to machine learning with global satellite imagery". In: *Nature Communications* 12.1 (July 2021), p. 4392. DOI: 10.1038/s41467-021-24638-z.

[104] D. P. Roy et al. "Landsat-8: Science and product vision for terrestrial global change research". In: *Remote Sensing of Environment* 145 (Apr. 2014), pp. 154–172. DOI: 10.1016/j.rse.2014.02.001.

[105] RStudio. *rmarkdown: Dynamic Documents for R*. R package version 2.19.2. Accessed: 2023-01-07. 2022. URL: https://github.com/rstudio/rmarkdown.

[106] Steve Running, Qiaozhen Mu, and Maosheng Zhao. *MYD16A2 MODIS/Aqua Net Evapotranspiration 8-Day L4 Global 500m SIN Grid V006*. Type: dataset. 2017. DOI: 10.5067/MODIS/MYD16A2.006.

[107] Mark Santolucito, William T. Hallahan, and Ruzica Piskac. "Live Programming By Example". In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI EA '19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 1–4. DOI: 10.1145/3290607.3313266.

[108] Arvind Satyanarayan et al. "Vega-Lite: A Grammar of Interactive Graphics". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 341–350. DOI: 10.1109/TVCG.2016.2599030.

[109] VERBI Software. *MAXQDA*. https://maxqda.com/. Accessed: 2022-11-17. 2022.

[110] Harry Stevens. *Will global warming make temperature less deadly?* en. URL: https://www.washingtonpost.com/climate-environment/interactive/2023/hot-cold-extreme-temperature-deaths/ (visited on 11/11/2023).

[111] Kristin Stock and Hans Guesgen. "Geospatial Reasoning With Open Data". In: *Automating Open Source Intelligence*. Ed. by Robert Layton and Paul A. Watters. Waltham, MA, USA: Syngress, 2016. Chap. 10, pp. 171–204. DOI: 10.1016/B978-0-12-802916-9.00010-5.

[112] Tableau. *Tableau*. https://www.tableau.com/. Accessed: 2023-11-10. 2023.

[113] Bridget Thrasher et al. "Bias correcting climate model simulated daily temperature extremes with quantile mapping". In: *Hydrology and Earth System Sciences* 16.9 (2012), pp. 3309–3314.

[114] Carol Traynor and Marian G. Williams. "A Study of End-User Programming for Geographic Information Systems". In: *Papers presented at the seventh workshop on Empirical studies of programmers*. ESP '97. New York, NY, USA: Association for Computing Machinery, Oct. 1997, pp. 140–156. DOI: 10.1145/266399.266412.

[115] Carol Traynor and Marian G. Williams. "End Users and GIS: A Demonstration Is Worth a Thousand Words". In: *Your Wish is My Command: Programming by Example*. San Francisco, CA, USA: Morgan Kaufmann Publishers, Mar. 2001, pp. 115–134.

[116] Carol Traynor and Marian G. Williams. "Why Are Geographic Information Systems Hard to Use?" In: *Conference Companion on Human Factors in Computing Systems*. CHI '95. New York, NY, USA: Association for Computing Machinery, May 1995, pp. 288–289. DOI: 10.1145/223355.223678.

[117] René Unrau and Christian Kray. "Usability evaluation for geographic information systems: a systematic literature review". In: *International Journal of Geographical Information Science* 33.4 (Apr. 2019), pp. 645–665. DOI: 10.1080/13658816.2018.1554813.

[118] René Unrau, Morin Ostkamp, and Christian Kray. "An approach for harvesting, visualizing, and analyzing WebGIS sessions to identify usability issues". In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS '17. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 33–38. DOI: 10.1145/3102113.3102122.

[119] USDA. *USDA - National Agricultural Statistics Service - Research and Science - Cropland Data Layer Releases*. https://www.nass.usda.gov/Research_and_Science/Cropland/Release/index.php. Accessed: 2022-07-12. Type: dataset. 2022.

[120] Leland Wilkinson. *The Grammar of Graphics*. 2nd. New York, NY, USA: Springer Science+Business Media, Inc., 2005. DOI: 10.1007/0-387-28695-0.

[121] Chaowei Yang et al. "Utilizing Cloud Computing to address big geospatial data challenges". In: *Computers, Environment and Urban Systems*. Geospatial Cloud Computing and Big Data 61 (Jan. 2017), pp. 120–128. DOI: 10.1016/j.compenvurbsys.2016.10.010.

[122] Katherine Ye et al. "Penrose: From Mathematical Notation to Beautiful Diagrams". In: *ACM Transactions on Graphics* 39.4 (July 2020), 144:144:1–144:144:16. DOI: 10.1145/3386569.3392375.