Design Study for a Geospatial-Video Data Analysis Query Language

Chanwut Kittivorawong
UC Berkeley, USA
chanwutk@berkeley.edu

Shadaj Laddad
UC Berkeley, USA
shadaj@berkeley.edu

Andrew Lenz
UC Berkeley, USA
andrew.lenz@berkeley.edu

Amy Lu
UC Berkeley, USA
amyxlu@berkeley.edu

ABSTRACT

Geospatial-video databases help data scientists store their video data efficiently, but many of them do not provide intuitive query languages for the data scientists to retrieve the data back. Databases with intuitive query languages provide data abstractions that hide their internal data representations. These data abstractions limit the number of operations that data scientists can perform on their data. In this paper, we address the failure for existing languages to easily describe queries over a combination of video frames and detected objects, as identified through need-finding interviews with users of geospatial-video query languages. We focus on developing a novel data abstraction for video data and domain-specific language for describing video queries with a functional interface. Our new query language provides operations for users to reason about video frames in their data while also considering the behavior of specific objects detected within these frames. Finally, we explore a design that would enable non-programmers to interact with our query system through a graphical interface.

Author Keywords

User Studies; Geospatial-Video Data; Data Model; Query Language

CCS Concepts

•**Human-centered computing** → **User studies;** •**Information systems** → **Query languages;** *Geographic information systems;*

INTRODUCTION

Capturing moments with video cameras is easy, and storage costs for storing video data is cheap. Video cameras have thus become a widely used tool to document moments through visual information. However, visual information of video data is unstructured, making it difficult to store and look-up. Processing and analyzing is often labor- and time-intensive for the data scientist. A common approach is to extract spatial and temporal information from the video data, using computer vision techniques [13, 18]. The extracted information makes the data more structured and easier to organize. We can then map the spatial information to the real-world location, thus transforming it into geospatial information. Geospatial information helps data scientists make sense of the video data as they correlate with the real world. However, we still lack easy-to-use and expressive languages to retrieve and explore the data.

Our goal for this project is to design a language that makes it easier for data scientists to search and explore massive geospatial-video datasets. We explore two geospatial-video data analysis use cases, and use them to shape the design of our tool:

- **Autonomous Driving**: Scenic [4] is a language for generating visual scenes, specifically for autonomous vehicle training data. A Ph.D. student in Computer Science is working on validating visual scenes generated [8], by comparing the scenes with a real-world autonomous vehicle video data [3].

- **Evident of Misconduct:** Data journalists often have large banks of video data [9], but not nearly enough time to look through these data. This hinders their ability to retrieve evidence involved in situations like police or military misconducts or false court statements.

In order to uncover the essential user necessities of a geospatial-video analysis tool, introduce new features to expediate workflows, and decrease barrier-to-entry for working on these important problems, we conducted user interviews related to the two use cases above.

We have explored prior attempts on geospatial-video data analysis tools. VisualWorldDB [7] presents a new data model to help users make sense of their video data after they are stored in the database. However, the work focuses heavily on the data storage system and optimizations. The proposed SQL-like query language is difficult to use as it exposes its internal data tables from which users can query. Apperception [6] is built on top of the VisualWorldDB idea and aims to improve

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

its usability. Apperception has a much simpler query language compared to VisualWorldDB. In the attempt, Apperception hides all the the internal data table and present a new data abstraction to users. As a result, the query language cannot express some data exploration questions. CLIP [12] is a large language model capable of generating video frames from a natural language description as the prompt. However, the results are not deterministic, and it is difficult to query geospatial or temporal information.

In this project we made following contributions:

1. We interview a data journalist and a computer science undergraduate, who are working on geospatial-video data analysis tasks.
2. We design a Python [15] programming tool based on prior works and participants' need from the interviews. Our tool presents an easy-to-understand data abstraction that will allow programmers of any experience level to easily explore large, annotated geospatial-video databases. Our tool also provides an expressive domain-specific language for users to explore their geospatial-video datasets through the data abstraction we provide.
3. We explore how users will be able to extend our language with custom functions that describe complex shared logic for queries in specific domains. These custom functions help users to easily express queries that would otherwise require complex compositions of the primitives in our query language.
4. We propose an unimplemented design for a Graphical User Interface (GUI) that will allow a non-programmer to apply the same principles that our Python tool provides but without programming experience.

RELATED WORK

VisualWorldDb

In this paper [7], the author introduces the concept of Visual World Applications (VWAs). VWAs are applications that rely on spatial and temporal information from visual data. The authors present VisualWorldDB: a video database management system optimized for VWAs. With VisualWorldDB, users can create a *world* that represents the real world. Then, users can ingest multiple geospatial-videos into the *world*. VisualWorldDB then places all the *visual objects* present in the videos into the *world*. Each *visual object* has properties that include its location in the real world. After the users have all the *visual objects* added to the *world*, they can query for these *visual objects* instead of their videos. In the end VisualWorldDB retrieves the videos associated with the *visual objects* queried.

Despite the simplified data model, VisualWorldDB focuses on its storage systems and optimization. The paper does not go into much discussion about VisualWorldDB's query language. The paper presents an example program in VisualWorldDB's language to ingest videos into a *world*. The language has a SQL-like syntax that expose internal data tables for users to manually input rows of data themselves. However, this approach is not intuitive to users. First, the programs that users write are prone to errors. All the internal data tables should have an invariant that keep the *world* representation valid. However, users have access directly to the internal data table, so they can unintentionally break some of the invariant. Second, the program that the user writes is verbose. To add a video camera into a created *world*, users need to:

1. Add the camera as an object to the world into an Objects table.
2. Add the position of the camera as a point in to the world into a Point table.
3. Add the orientation of the camera as another point into the world into the same Point table.
4. Add a video file associated with the camera into the created world.

Adding a camera to the *world* could be done as one operation – instead, we need four operations to perform this task. In this project, we take inspiration from this *visual world* concept.

Apperception

Apperception [6] has a similar *world* concept as VisualWorldDB but with a more usable Application Programming Interface (API). Apperception stores all the geospatial data in MobilityDB [19]. However, it provides a Python API for users to interact with its geospatial-video data storage. Apperception also provides a Python *World* object that represent a *world*. There are multiple operations that users can do with a *world*.

- `add_camera` to add a camera and video to the *world*. Compared to VisualWorldDB, users only need one operation to add a camera to a *world*.
- `recognize` to extract objects from an existing video and place them into the *world*.
- `filter` to filter only objects that the user is interested in.

After all the operations done, users will have a *World* that contains only the objects that they are interested in. Then, they can use a `get_*` method to observer the state of the world.

- `get_trajectory` gives users the trajectory of each of the objects in the world. The users can then use an `overlay_trajectory` method to overlay the trajectories into the original video.
- `get_videos` gives the videos of all the objects in the world.

Apperception's interface is easy to use and not verbose. It wraps around the internal geospatial-video data storage, so the users do not have direct access to the data table. As a result, it is less likely for them to accidentally violate the data store invariant.

A *World* is immutable, which means that it cannot be modified. Any operation done to a *World* create a new *World* that represents the original one with the operation applied to. This approach for making a *World* immutable helps users to easily reason about the program they have written. In addition,

having a immutable DataFrame-style syntax is common in tools [10, 11, 17] for data exploration tasks. Users who are already familiar with data exploration tools could learn how to use Apperception quickly.

However, Apperception's language is designed for users to find objects in a *World*. It is difficult for users to describe a query that can find a specific moment of an object using only Apperception language. For example, if a user has an hour-long video that capture the same car for the whole video. The car crashes at 45th minute of the video. The user can use Apperception language to query for a car that crashes. Apperception will return a *World* that only contains the car. But, Apperception cannot provide the information about the when the incident occurs.

Our project will build on top of Apperception's language: we aim to address the problems with Apperception language, specifically, that it is not expressive enough to describe complex data exploration questions.

Rekall

Rekall [5] is another video data analysis tool that we looked into. Rekall presents an abstraction for video data for its video frames. Rekall does not organize videos as a *world* – instead it simply organize them as collections of video frames. Rekall also provide a `filter` method like in Apperception, but instead of filtering by objects, Rekall's `filter` filters only the frames that the user is interested in.

This abstraction and operator is helpful to our project because it fills in the main disadvantage of Apperception. Our project will be integrating the by-frame abstraction to our data model and query language. However, instead of being limited to just reasoning about raw frames over time, our system includes abstractions and APIs that make it possible to process objects that are recognized or annotated in the video data.

Fructure for Racket

When considering the design for our user interface, we were inspired by Andrew Blinn's Fructure for Racket [2], a projectional editor that allows users to either type code or select code via mouse to fill in holes in a program. Projectional editors are efficient for basic code-editing activities [1] and useful for educational uses, particularly as they prevent syntax errors [14, 16], which is important for our users who are less experiences with coding. On the other hand, our more experienced programmers may want to type code the traditional way: Fructure's flexibility allows users to type in the holes rather than just scrolling through the list, and we would implement this as well.

INTERVIEWS

In order to identify short-comings for current tools and guide the design of our work, we conducted contextual inquiry and observed users performing tasks involving geospatial video analysis using existing tools. An area of the pipeline which we were especially interested in is how researchers specify queries that identify situations with multiple objects, potentially with some specific behavior over time. The following section discusses the findings from our two interviews in greater detail.

Data Journalism

User L is a data journalist at a public radio station who must look at video evidence on the order of thousands to collect and report on evidence, specifically surrounding police misconduct. With permission, we recorded L performing the video queries required for this task. In L's workflow, L goes through several videos and makes quick notes on how they relate to each other and which ones might be most helpful, and clicks through frames to roughly gauge what is happening in the video. She logs information about each video and key frames, consisting of information such as the subjects and objects present in the video and what is happening.

This user study illustrated two key needs. One is the need for users to be able to have as much metadata as possible about videos and frames in order to save time; much of the pipeline involves scoping down parts of the video until the relevant segments and frames are found. Second, we observe the level of detail within frames that is needed (e.g. "This car is moving towards the officer, that's very important") for key frames, as well as the level of context needed before an inference can be made on what has happened. This indicates that simplifying how users access and query the video annotations and metadata alleviates bottle-necks in the workflow. In our language, the data model are designed to address this (see Data Model: Section 5).

Programmer

User Y is a student researcher who translates between Scenic to Apperception queries to aid in autonomous vehicle research. We give Y a task to describe what the queries in our language aim to do by looking at the initial version of our syntax alone, after a short overview of the key elements of our language. The tasks are to: (1) identify all frames where more than two cars are present; (2) identify all objects which has another object moving towards them; (3) identify objects of people where they are moving away from a car ; and (4) identify all cars to the left of the camera.

These reflect calls which may be necessary for a researcher building planning or machine learning models from such data for autonomous driving. The tasks are designed to involve reasoning about the behavior of specific objects over time, since a key element of our language is to enable users to filter by frame, and then switch to filtering by objects. We find that Y was able to correctly understand and explain the purpose of the syntax without too much time or guesswork.

In the end, Y agree that the abstraction and language that we present provides the functionalities that Apperception is missing. He also gives suggestions to an improvements of the language design. Instead of exploring the video data through either *Frame* or *Object*, how do we design a language that allows users to construct a query that involve both *Frame* and *Object* at the same time?

DATASET

For our prototype, we use the nuScenes dataset [3], a collection of camera, lidar, and radar images to facilitate autonomous driving research. Data is collected from 15 hours of driving data in Boston and Singapore, and key frames are sampled at

a rate of 2Hz for labeling by human annotators, resulting in a rich collection of information suitable for various downstream needs in autonomous driving research. The structure of the decomposition of video frames into data and annotation structure was a good representative of geospatial settings for our language design.

DATA MODEL

To interface with the data, the user uses the *Annotation*, *Instance*, *Frame*, and *Video* abstractions.

- ***Video*** represents a video file.
- ***Frame*** represents a video frame in a video file.
- ***Instance*** represents an object that presents in a video. An *Instance* can be present on a video spanning over multiple frames as shown in Figure 5.0.1.
- ***Annotation*** represents properties of an *Instance* for a specific *Frame*.

Figure 5.0.1 illustrates this data model in greater detail. Each video is denoted by an unique scene ID, and frames are indexed by the order in which they appear. All instances have an unique instance token which can be used to query specific properties for that instance specifically. By passing the scene ID, the user can load the video with corresponding annotations and properties needed for downstream tasks. The user may also directly instantiate any Frame and Instance directly from the nuScenes database, if provided with the frame order and instance token in addition to the scene ID. All metadata are stored in Annotations, with the exception of “Category" (e.g. “car", “person"), which is an attribute in *Instance*.

Our data model is an improvement to Apperception’s data model. In Apperception, users mostly interact with *World*, *Video*, and *Instance*, and do not have access to individual frames of the video. The output from Apperception can also only represent *World* and *Instance*. In the new data model, users can interact with all of the data types shown in Figure 5.0.2. This abstraction allows users to construct queries to answer more complex data exploration questions, while not exposing the internal geospatial data store.

DOMAIN-SPECIFIC LANGUAGE

To address the pain points we noticed in our user interviews, we chose to design a new language for describing queries over geospatial video data as a domain-specific language embedded in Python. Instead of building a new language from scratch, which would require designing a new syntax and editor integrations that users would have to learn about, we are able to inherit the large ecosystem of existing tooling around Python. Furthermore, some potential users of our system (such as User Y) are already familiar with Python, which makes it easier to adopt our language.

Design Guidelines

There are three key factors constraining the design of our language: incremental query creation, support for using the language through a graphical user interface, and supporting queries that combine predicates over time and space. Before we dive into how we design the language around these constraints, let us walk through how we derived the factors based on our user-studies and their consequences on the space of language designs.

Incremental Queries

Across the interviews, a key pattern that stuck out was the process by which the participants would identify portions of video that were of interest—either by manual scrubbing through video files or the creation of queries in existing systems. Rather than define the end-to-end query, the participants would start with a coarse grained search, looking for portions of the video that are likely to have the information they are looking for. Across these sections, the participants would then refine their search/query by playing back the video in real-time or querying individual frames in more detail.

Our insight from this observation was that users of video query languages will want to develop the query incrementally, where they start with a query that identifies sections of the video with general patterns related to their specific goal–such as the presence of multiple cars in the same frame–and iteratively refine it to identify the specific scenario–such as cars moving towards each other–while receiving feedback from the query engine.

Supporting incremental query creation places significant constraints on the design of the language since the language must offer modular APIs to support users adjusting their query pipelines over time and must also have a performant runtime that can execute updated queries efficiently. Furthermore, it requires us to ascribe meaning to intermediate steps of a query, including complex concepts such as windows over time or groups of objects, so that users can see the results of each component of their pipeline.

Graphical User Interface

Based on our interview with User L, who works on data journalism but primarily uses manual scrubbing methods to query videos rather than writing code, we identified that a key feature to make our language useful is offering a graphical interface for constructing programs for users who are not comfortable with text languages. At the same time, however, we want to support a lower-level Python API for constructing queries that expert users such as User Y would be more comfortable using.

To support this graphical interface, we decided to implement a layer on top of the Python API with mappings from each concept in Python to features in the interface. This architecture means that the low-level API should be strongly typed, since types in the API are important to constrain the programs in the graphical interface to those that are executable by construction. Furthermore, it requires us to include several built-in APIs for common scenarios to reduce the overhead of describing various queries.

Queries over Time and Space

Finally, we observed in both interviews that queries over video data often require repeatedly moving between the axis of time and detected objects, something that existing query systems do not support. For a concrete example, consider an query over autonomous vehicle data that is looking for collisions. We

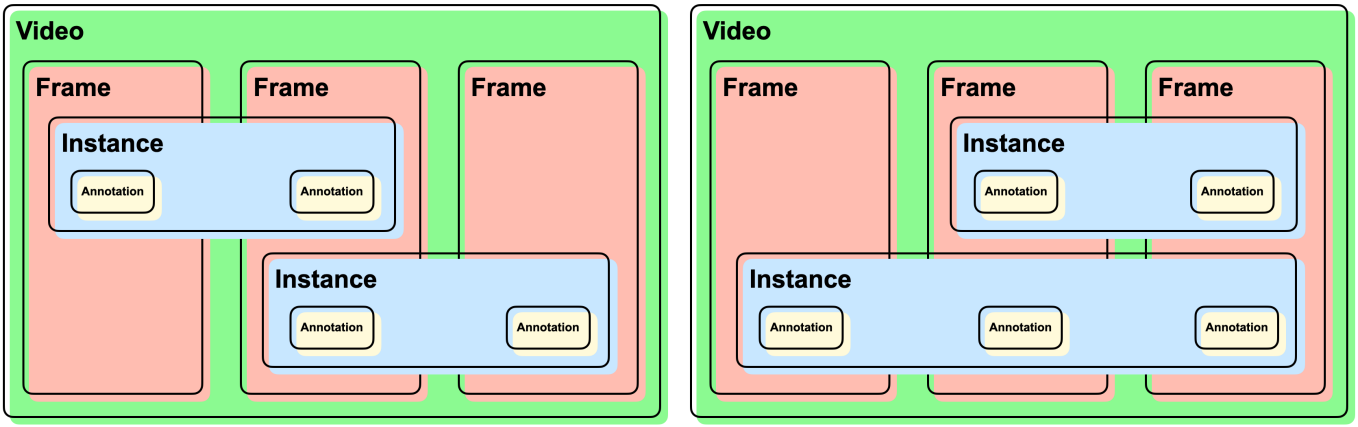

Figure 5.0.1: The wealth of image, annotation, spatial, and temporal information in geospatial data is simplified into in our language into Video, Frame, Instance, and Annotation. A video consists of several consecutive frame, which point to a specific image, and instances (e.g., cars, pedestrians) may persist through many frames. Each instance contains a category attribute, and an annotations attribute for accessing more detailed metadata of interest.

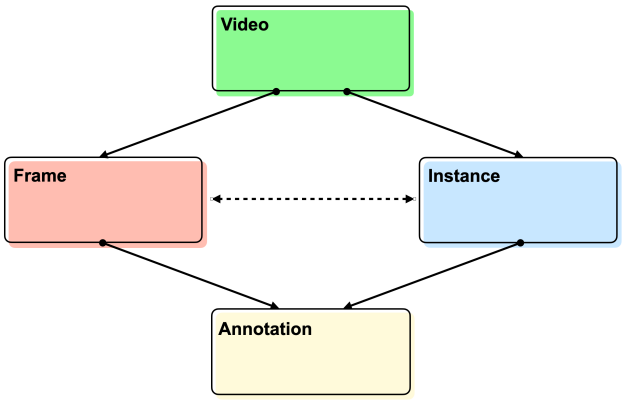

Figure 5.0.2: The hierarchy of our data type. Users can start exploring they geospatial-video data from *Videos*. Then, they can choose to explore the data through either *Frames* or *Instances*. And from both *Frames* and *Instances*, they can explore further to *Annotations*. In addition, the users have the flexibility in switching between exploring *Frames* and *Instances*.

want to find chunks of the video where two cars are moving to each other, but this requires predicates that both reason about multiple frames together and multiple objects together. After a user writes a query to identify cars, they will need to identify *frames* with multiple cares, and then process windows of frames to check if the detected objects are moving towards each other.

As we noticed in our interview with User Y, queries like this are difficult or impossible to describe with existing query languages, which either focus exclusively on time or objects as the primary axis being queried over. We aim to address this in our design by providing APIs that mix these two concepts together, enabling queries such as filtering over the properties of objects in windows of time. This requires us to extend the data model, as we discussed in the previous section, and also provide APIs for switching between the frame and instance views of video data.

Language Design

With these design goals, we decided to develop a DSL based on declarative and functional programming principles. Rather than writing out queries as imperative statements that mutate data, have unclear dependencies, and use hard-to-optimize control constructs, this approach allows us to optimize and execute the queries users write from a global perspective. By modeling queries as dataflows, such as the example in Figure 6.0.1, our language can also provide high-quality intermediate feedback since every intermediate node can be mapped to output samples. Finally, the declarative approach makes it simpler to build a graphical interface, since the dataflow model corresponds to a natural visual model of connecting primitive operations with dependencies.

Our DSL consists of four core types corresponding to the elements of our data model: videos, frames, instances, and annotations. To develop dataflows over these structures, we then provide collection types wrapping these elements, which represent nodes that will process incoming streams of video data. In the time domain, we also include a collection type that captures groups of frames rather than just individual elements, which is useful when performing tasks like tracking the motion of an object over a window of time.

To create complex queries, users can develop a dataflow pipeline by repeatedly transforming these collection types using functional operations. In addition to the classic transformations such as map, which transforms individual elements independent of each other, and filter, which applies a predicate to each element, we also include several operations specific to the domain of video data.

First, we include the `join` operator, which allows a user to combine multiple collections by matching elements with a specific property. This emits a collection containing groups of the elements in the original collections, and can be thought of as performing a crossproduct with a predicate. Rather than allowing an arbitrary condition, which can be difficult to evaluate efficiently, we constrain users to joining elements on properties which have already been computed. For example,

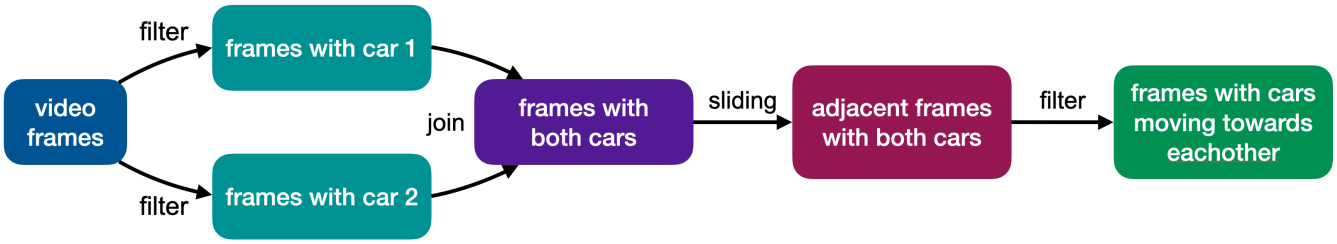

Figure 6.0.1: An example of a dataflow users can construct for a query that searches for adjacent frames where two specific cars are moving towards each other. The pipeline is composed of the primitive operations in our language, but by combining the frame and instance concepts users can create complex queries.

users may join on frames, which will gather pairs of elements from both inputs that are present in the same frame.

Next, we add the `sliding(n)` transformation, which makes it possible to develop queries that reason about windows of time in the video rather than just individual points in time. When applied to a collection of frames, this operator produces a new collection containing groups of `n` adjacent frames. When applied to a collection of instances, however, this operation produces a collection of groups of *annotations* corresponding to the states of the objects in adjacent frames.

Finally, we add APIs on collections of videos to flatten all frames (`flatten_frames`) or instances (`flatten_instances`) into a single collection. Users often have multiple video sources corresponding to a single event, so it is helpful to process all frames or objects with a single query pipeline. Furthermore, we plan to support users specifying groups of instances that correspond across videos so that many sources can be combined into a global view of the scene, in a similar manner to the world view in Apperception.

Together, these constructs make it possible to express a wide variety of queries that process videos with both the frame-by-frame and per-instance views. By composing our video-specific APIs with the general functional APIs that support nested logic over frames or instances, users can easily build out the queries they need while ensuring that the pipeline is sufficiently modular to be adjusted in the future.

In our prototype implementation, we have implemented simple implementations of these APIs that iteratively process the input video streams in single-threaded Python logic. In future work, however, we plan to utilize the declarative nature of our DSL design to compile user-defined pipelines to efficient, parallel implementations.

EXTENSIONS

While we expect that the language will be sufficiently expressive to construct most types of queries, we also recognize that some of our users may not have the time, desire, or programming expertise to implement the functions that they may want to use to filter the videos. Therefore, we built out some of the possible helper functions that could be useful for filtering. We focused on exploring utility functions for the autonomous driving application, which already has a common set of query tasks we could take inspiration from.

These functions expect to be called within filter and thus all return boolean values. The ones provided below should take in the result of the sliding operation, as they intend to compare information about the instance across adjacent frames in order to determine something about the instance's movement. However, these utility functions do not cover all possible use cases – our language is designed such that a user can add their own utility functions as needed.

Utility functions for filtering:
(Note: `iX` and `fY` (where `X` and `Y` are whole numbers) will refer to Instances and Frames, respectively.)

- `move_away(i1, i2, f1, f2)`:
  This function calculates the distance between Instance `i1` and Instance `i2` in Frame `f1` and Frame `f2`, then compares these distances. If the distance between the instances in Frame `f2` is greater than the difference between the instances in Frame `f2`, then we consider the instances to be moving away from one another.

- `accelerating(i, f1, f2, f3)`:
  This function uses the location of Instance `i` to determine if it has accelerated between Frame `f1` and Frame `f2`, and Frame `f2` and Frame `f3`. To determine this, it compares the difference in location between both pairs of frames: if the difference between the locations in Frame `f2` and Frame `f3` is greater than the difference between the locations in Frame `f1` and Frame `f2`, we consider Instance `i` to have accelerated.

- `decelerating(i, f1, f2, f3)`:
  This function uses the location of Instance `i` to determine if it has decelerated between Frame `f1` and Frame `f2`, and Frame `f2` and Frame `f3`. To determine this, it compares the difference in location between both pairs of frames: if the difference between the locations in Frame `f2` and Frame `f3` is less than the difference between the locations in Frame `f1` and Frame `f2`, we consider Instance `i` to have decelerated.

- `stopped(i, f1, f2, tol: float)`:
  This function uses the location of Instance `i` to determine if it has stopped from Frame `f1` to Frame `f2`. To determine this, it compares the difference in location between the two frames to the tolerance `tol`: if the difference is greater than `tol`, we consider the instance to have moved; otherwise, it has stopped.

```
people = videos \
    .flatten_instances() \
    .filter(lambda i:  \
        i.property["type"] == "person")

cars = videos \
    .flatten_instances() \
    .filter(lambda i:  \
        i.property["type"] == "car")

result = people \
    .join(cars, on="frame") \
    .sliding(2) \
    .filter(lambda i1, i2, f1, f2:
        move_away(i1, i2, f1, f2)
```

Figure 8.1.1: Example Query in our Programming Language

USER INTERFACE

Although we did not implement a user interface for the language, we were able to prototype some designs for such an interface. Based on these explorations, we propose a block-based projectional editor interface to provide the most convenience and accessibility for programmers, particularly those with less experience who we expect to use this tool. To facilitate this, we would allow users to fill the holes in skeleton queries by selecting entries from a drop-down menu which would also have a search bar; this will provide suggestions to users who want to explore the dataset, but also allow users who know what want to see to search for what they are looking for.

As the output of a query in this language would be a series of clips or videos taken from the overall collection of videos, we envision the interface as having a region in which the output of the queries are displayed. However, this creates some problems when we have too many output videos. To amend this, we would split the list of videos into multiple pages to prevent the tool from loading all the videos at once. Then, we show the number of output videos. If the number of output videos is too large, we suggest that the users add more filters to the output to narrow down their search.

Mapping the Language to the Interface

First, the tool will store the collection of videos as some variable, which we will refer to as `videos`. A typical query in our language is shown in Figure 8.1.1.

In Figure 8.1.1, we show an example query where a user is looking for all frames where a person and a car move away from one another. For clarity, this is split into three queries: the first two filter the videos for instances of people or cars, and the third joins these sub-queries to find sequences of frames in which the instances move away from one another; however, these can be combined into a single line of code.

In Figure 8.1.2, we show what the query would look like using our proposed interface. The highlighted sections represent the holes that the user has filled in.

```
Find frames with a person and a car
    moving away ...
```

Figure 8.1.2: Example Query in our Proposed Interface

Frames

When trying to find a certain frame, e.g., frames that occur between 7am and 8am, we will have a query in our interface that looks like Figure 8.1.3. This is equivalent to the code in Figure 8.1.4, which uses the `flatten_frames` method to allow users to filter along the frames.

```
Find frames between 7am and 8am
```

Figure 8.1.3: A frame-based query in the graphical programming interface

```
videos.flatten_frames() \
    .filter(lambda i:  \
        i.property["time"] > 7.00 and
        i.property["time"] < 8.00)
```

Figure 8.1.4: The same frame-based Query in our domain-specific language

Instances

When trying to find frames that contain a certain instance, e.g., a car, we will have a query in our interface that looks like Figure 8.1.5. This is equivalent to the code in Figure 8.1.6, which uses the `flatten_instance` method to allow users to filter along the instances.

```
Find frames with a car
```

Figure 8.1.5: An instance-based query in the graphical interface

```
videos.flatten_instances() \
    .filter(lambda i:  \
        i.property["type"] == "car")
```

Figure 8.1.6: The instance-based query in our domain-specific language

Multiple Instances in the Same Frame

When looking for frames that have multiple instances, e.g., a person and a car, we will have a query that looks like Figure 8.1.7. To translate this to our language, as seen in Figure 8.1.8, we'll first filter for the frames that contain a person, then filter for the frames that contain a car, and finally join these two lists of using the `join` method, returning only the frames containing both instances. To translate from the structure in the interface to this back-end, the interface should create a new list of frames for each new instance added, and join on matching frames for each pair of instances separated by the "and".

```
Find frames with a person and a car
```

Figure 8.1.7: Query for Multiple Instances in our Proposed Interface

```
people = videos \
    .flatten_instances() \
    .filter(lambda i:  \
        i.property["type"] == "person")

cars = videos \
    .flatten_instances() \
    .filter(lambda i:  \
        i.property["type"] == "car")

result = people \
    .join(cars, on="frame") \
```

Figure 8.1.8: Query for Multiple Instances in our Programming Language

Comparisons with Adjacent Frames

When comparing information between adjacent frames – for instance, when trying to find positional or motion-related information about an instance (in this case a person)– we will have a query that looks like Figure 8.1.9. To translate this to our language, as seen in Figure 8.1.10, we'll first filter for the frames that contain a person, then use the `sliding` operation to create pairs of frames, and finally filter these pairs to find those between which the person does not move.

```
Find frames with a person stopped
```

Figure 8.1.9: Frame Comparison Query in our Proposed Interface

```
people = videos \
    .flatten_instances() \
    .filter(lambda i:  \
        i.property["type"] == "person")

result = people \
    .sliding(2) \
    .filter(lambda i, f1, f2:  \
        stopped(i, f1, f2) \
```

Figure 8.1.10: Frame Comparison Query in our Programming Language

Notice that in this case, the `sliding` operation is placed before the second `filter` operation. When any utilify functions are used to compare between frames, we must apply a `sliding` operation first so we can access both frames.

Using the Interface

We will now demonstrate how to construct the query shown in Figure 8.1.2 using the proposed block-based interface. Note: `(def)` means that this is the default option.

First, we must select the type of query. In this case, since we want the frames where a car and person move away from one another, we will select the query skeleton corresponding to finding frames with a specified instance.

```
Find frames with Instance: all instances (def)
    ...
```

Figure 8.2.1: Query Skeleton

To specify that we want instances a car, we will replace the `Instance: all instances (def)` with `a car`.

```
Find frames with a car ...
```

Figure 8.2.2: Selecting the first Instance

Then, to add to the query, we will select the `...` and choose the option corresponding to a new instance.

```
Find frames with a car and
    Instance: all instances (def) ...
```

Figure 8.2.3: Adding the second (generic) Instance

To specify that this second instance should be a person, we will fill in the `all instances (default)` hole with `a person`.

```
Find frames with a car and a person ...
```

Figure 8.2.4: Selecting the second Instance

Then, to indicate that we want to filter on the frames the contain a car and a person, we will select the `...` and choose the option corresponding to a new filter.

```
Find frames with a car and a person
    Filter: no filter (def) ...
```

Figure 8.2.5: Adding the (generic) filter

Finally, as we want scenes where the car and person move away from one another, we will replace the `Filter: no filter (def)` with `moving away`. This will give us the desired query.

```
Find frames with a car and a person
    moving away ...
```

Figure 8.2.6: Selecting the filter

This iterative process of creating a query also ties into the incremental query execution goal of the language design. Because we have a declarative language that is optimized around providing feedback on queries as they are built up, we can provide feedback to users as they define queries in the GUI. For example, we can display samples of frames that include just a car before the user adds the additional constraint that a person should be present in the frame.

By providing all the features of the low-level Python API while including features that simplify the process of defining common query elements, we believe that this interface design will make it easier for non-programmers like User L to use our system. In future work, we hope to develop this design into a full implementation and investigate how users interact with the language through this layer.

CONCLUSION

In this project, we explored prior attempts on geospatial-video data analysis tools. We investigated their strengths and weaknesses. We interviewed data journalists and programmers

working on geospatial-video data analysis tasks. Compiling the insights from both prior works and interviews, we designed a geospatial-video data analysis programming tool. The tool presents a data model and a query language that add missing functionalities from prior works and are necessary to the tasks of our interviewee. We also designed our tool to be extensible. Users can add their own custom functions that are necessary for their data exploration tasks. Finally, we designed a Graphical User Interface tool that allows users to do the same tasks as the programming tool without programming experience.

REFERENCES

[1] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 763–774. DOI:http://dx.doi.org/10.1145/2950290.2950315

[2] Andrew Blinn. 2020. Fructure: A structured interaction engine in Racket. (2020). https://github.com/disconcision/fructure

[3] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. 2019. nuScenes: A multimodal dataset for autonomous driving. (2019). DOI: http://dx.doi.org/10.48550/ARXIV.1903.11027

[4] Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2020. Scenic: A Language for Scenario Specification and Data Generation. (2020). DOI: http://dx.doi.org/10.48550/ARXIV.2010.06580

[5] Daniel Y. Fu, Will Crichton, James Hong, Xinwei Yao, Haotian Zhang, Anh Truong, Avanika Narayan, Maneesh Agrawala, Christopher Ré, and Kayvon Fatahalian. 2019. Rekall: Specifying Video Events using Compositions of Spatiotemporal Labels. (2019). DOI: http://dx.doi.org/10.48550/ARXIV.1910.02993

[6] Yongming Ge, Vanessa Lin, Maureen Daum, Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. 2021. Demonstration of Apperception: A Database Management System for Geospatial Video Data. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2767–2770. DOI: http://dx.doi.org/10.14778/3476311.3476340

[7] Brandon Haynes, Maureen Daum, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2020. VisualWorldDB: A DBMS for the Visual World.. In *CIDR*.

[8] Edward Kim, Jay Shenoy, Sebastian Junges, Daniel Fremont, Alberto Sangiovanni-Vincentelli, and Sanjit Seshia. 2021. Querying Labelled Data with Scenario Programs for Sim-to-Real Validation. (2021). DOI: http://dx.doi.org/10.48550/ARXIV.2112.00206

[9] Robin McDowell and Margie Mason. 2021. AP Investigation: Myanmar's junta using bodies to terrorize. (2021). https://apnews.com/article/myanmar-business-b2187c696e428139437778aeab0c43d4

[10] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56 – 61. DOI: http://dx.doi.org/10.25080/Majora-92bf1922-00a

[11] The pandas development team. 2020. pandas-dev/pandas: Pandas. (Feb. 2020). DOI: http://dx.doi.org/10.5281/zenodo.3509134

[12] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. (2021). DOI: http://dx.doi.org/10.48550/ARXIV.2103.00020

[13] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2015. You Only Look Once: Unified, Real-Time Object Detection. (2015). DOI: http://dx.doi.org/10.48550/ARXIV.1506.02640

[14] André L. Santos. 2020. *Javardise: A Structured Code Editor for Programming Pedagogy in Java*. Association for Computing Machinery, New York, NY, USA, 120–125. https://doi.org/10.1145/3397537.3397561

[15] Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

[16] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Trans. Comput. Educ.* 18, 1, Article 3 (oct 2017), 25 pages. DOI:http://dx.doi.org/10.1145/3089799

[17] Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. 2022. *dplyr: A Grammar of Data Manipulation*. https://dplyr.tidyverse.org, https://github.com/tidyverse/dplyr.

[18] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. 2017. Simple Online and Realtime Tracking with a Deep Association Metric. (2017). DOI: http://dx.doi.org/10.48550/ARXIV.1703.07402

[19] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. 2020. MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS. *ACM Trans. Database Syst.* 45, 4, Article 19 (dec 2020), 42 pages. DOI: http://dx.doi.org/10.1145/3406534