INFO-H402: Computing Project

Visualizing MobilityDB Data using QGIS

Author:

Ludéric Van Calck

Supervisors:

Esteban Zimányi

Mahmoud Sakr

2020-2021

Contents

1 Introduction

MobilityDB is a database management system that can store and manage moving object geospatial trajectory data, i.e. all of the information that can be used to describe a moving object's (such as a car or a plane) trajectory. The goal of this project is to explore ways to visualize these moving objects automatically using QGIS, a desktop application that can be used to view, edit and analyze geospatial data. In order do so, several experiments were conducted to try and find the most efficient way to achieve this goal.

2 QGIS High-Level Description

Much like a graphical design software such as Photoshop or GIMP, QGIS' main component is its *canvas*, which takes up most of the application's visual space. Objects can be "painted" onto the canvas by drawing them on *layers*. Layers act like transparent sheets which can show a particular set of objects. Let's also note that a layer only contains objects of a single type (or *geometry*), so if one wants to show different types of objects, different layers have to be used.

For example, if one wished to show the position of busses on the STIB's network (Brussels' public transport company) at a given time, a layer would contain the network's roads (lines) while another would contain the busses themselves (points). They would then be superimposed and rendered onto the canvas to show the whole picture.

2.1 Layers

Layers are the components that allow objects to be rendered onto the canvas. This means that objects need to be added to a layer before they can be rendered. Adding an object to a layer basically means to store an object's information somewhere the layer can access it. All of the object's information is stored inside a *feature*. Basically, a feature is the concrete representation or encoding of the object, but since object and feature pretty much refer to the same thing, both terms will be used interchangeably.

Coming back to our STIB example, since we wish to show the busses' positions at a given time, each feature of the bus layer could contain a bus' id, driver's name and position. The position variable has a special role since it contains the bus' spatial information and is called the feature's *geometry*. A feature's geometry controls where and with what kind of shape that feature should be rendered inside the layer. The other attributes can be used to change its "cosmetics", such as its color, size, or maybe to show some text with an attribute's value (like the driver's name for the STIB example) next to the feature, but they can also be completely ignored. If one wanted to render more complex objects, such as polygons, lines or any other shape, QGIS provides support for many geometries. As a reminder, a layer can only contain features that all have the same geometry.

This covers *how* objects can be rendered by the layer, but not *where* the data describing them is stored. To access and manage the objects' data, the layer uses a *data provider*. This data provider can be the application's memory, a file (such as a shapefile or csv) or even a PostGIS database table. Even though there is a wide array of different data providers, they work in a similar way. You can simply imagine that the data provider is a table, where each line refers to a different *feature*, and the columns contain the different features' attributes (one column will contain each feature's geometry). For most data providers, QGIS can automatically recognize the features' attributes and geometry.

2.2 Temporal Controller

As mentioned, layers can very well detect and load features and their spatial geometry. However, mobility database systems such as MobilityDB, store objects that also contain temporal information on top of the spatial information. As one could imagine a static layer is ill-equipped to display moving objects. Fortunately, a somewhat recent QGIS update introduced the temporal controller tool. This tool makes it possible to mark a layer as *temporal*. When a layer is temporal, each of its features should have one or more temporal attributes, or timestamps. The temporal controller then allows to filter the features of that layer based on their timestamps. To do so, the temporal controller tool provides a simple GUI containing a scroll bar, which can be used to navigate between a start and end time.

The period between start and end times is divided into a number of frames. This number of frames depends on the *frame duration* (a parameter that can be set inside the temporal controller GUI), and the length of time between start and end points. At each frame, only the features with a timestamp intersecting with the frame's begin and end timestamps (which depend on the frame number and frame duration) are filtered and visible inside the temporal layer. The temporal controller's *frame rate* controls the speed at which the temporal controller goes from one frame to the next (i.e. the speed of the scroll bar).

With our STIB example, this would mean that we could now animate the busses, provided we had the data for their positions and times between 4PM and 6PM let's say. Now, each feature from the layer's data provider represents a bus *at a given time*, e.g. each feature could contain a bus's geometry (its location), its ID and the time at which the location was sampled. Playing the temporal controller scroll bar would successively filter the features between 4 and 6PM, thus creating an animation. If the frame duration is set at 1 minute, and the frame rate is set at 30 FPS (frames per second), the animation would last 4 "real life" seconds (120 minutes of busses data divided into 120 frames of 1 minute each, playing at 30 FPS). On the other hand, the layer containing the roads of the network, wouldn't be marked as temporal and would simply remain statically in the background, while the busses would "travel" on top.

3 Problem Description

Unfortunately for QGIS, MobilityDB doesn't represent a bus's trajectory as a succession of rows in a database with associated timestamps, but rather as a single row where the whole trajectory is stored as a temporal geometry point or *tgeompoint*. This means that you cannot simply use a MobilityDB database table as a data provider for a QGIS layer, since QGIS wouldn't recognize a *tgeompoint* as a valid geometry. However, to animate points onto the canvas, it would be possible to simply transform that row into a table with the tgeompoint's position at constant intervals (possibly equal to the temporal controller's frame duration), and to then use that resulting table as a data provider to create an animation using the temporal controller. This works, but it is pretty tedious to do, since it would require writing a custom SQL query, running it and storing the result, and finally importing/linking it to QGIS anytime an animation is to be created.

The goal of this project is to automate the procedure of transforming temporal geometry points (trajectories) to geometries recognized by QGIS, and to use the temporal controller to create animations in real time (i.e. not having to wait for an animation to generate). This could be achieved with the python QGIS API, which can be used to automate tasks and create plugins for QGIS. On top of this, the goal is also to try and find the most efficient way to do this, since interpolating the data of many, large trajectories can end up being costly.

4 Using the QGIS API

QGIS provides a python API which can be used to control and interact with pretty much all of its components, such as the canvas, layers and temporal controller to only name a few. Python scripts can be used from within QGIS itself, to run actions on these components, as well as to listen for signals of when an action is performed by the user or QGIS itself. For example, when the temporal controller scroll bar moves (i.e. at every new frame), it emits a signal that can be intercepted by a python script, that can then run some code.

For our purposes, there are a few actions that can be automated:

- Creating a temporal layer
- Getting access to a MobilityDB table from QGIS
- Interacting with the temporal controller to generate an animation

- Transforming the tgeompoints from the MobilityDB table to features that can be shown inside the layer

Creating a temporal layer and linking a MobilityDB database to QGIS is pretty straightforward and can be done with a few lines of python code.

Interacting with the temporal controller can be done using Qt signals and slots. Qt is the GUI framework used to build widgets that is used by QGIS, and it offers the possibility for these widgets to send signals when they are updated. In our case, we can know that a new frame of the temporal controller is to be played by connecting the specific signal the temporal controller emits for that action with a slot. A slot is basically just a python function that is called whenever the signal it is connected to is emitted. In our case, that function will be called *onNewFrame*, and will trigger whenever the temporal controller is being played (the function will be called 30 times a second for an animation at 30 FPS).

Transforming the tgeompoints into features that can be added to the layer is probably the most tricky part. What we want to do is to get the interpolation of a tgeompoint trajectory that is stored inside the MobilityDB database, for a specific time. That specific time can simply be the start time of a frame of the temporal controller. If we are able to get the positions of all the trajectories at the beginning of every frame from the temporal controller, we could then simply create features with these positions as their geometry and the corresponding frame start time as the temporal attribute. We could finally add them all to the layer's data provider, and moving the temporal controller scroll bar would simply filter the features according to the frame that needs to be shown. We would thus have a functioning animation, able to display the trajectories inside the MobilityDB database. There are different ways this can be done, which will be discussed in section 5.

Finally, we could combine the last two elements from the list. Indeed, instead of generating all of the frames for all of the temporal controller's period, storing them and then navigating through them, we could use the signal we get at every frame to generate the features that have to be displayed for that frame, and then add them to the layer on the fly. This could potentially prevent the layer from having to load an enormous number of features at once, if we have a database with many objects whose trajectories span over a long time. Of course, loading features frame by frame can lead to other problems, so it might be interesting to have a middle ground approach and to generate a buffer of N frames every N frames, at the same time the animation is playing.

5 Experiments

The goal of these experiments is to measure the most efficient way to produce an animation able to visualize the data from a MobilityDB database. Achieving high performance is necessary if we want to be able to visualize the data in real time (i.e. not needing to pre-generate frames). There are 3 main time sinks:

- Querying the data from the database
- Interpolating a trajectory with a given timestamp (either database-side or QGIS-side using the MobilityDB python driver)
- Adding features to a layer so they can be displayed

The experiments from this section attempt to compare different possible solutions.

5.1 Data Set

The data set used for the experiments consists of a MobilityDB table containing the trips of 100 taxis. The columns contain IDs, as well as the tgeompoint column representing their whole trajectory. This table is probably much smaller than a real life dataset but still serves as a good basis to be able to evaluate performance.

5.2 Generating Frames

There are two main ways to generate the frames without generating all of the features ahead of time (we want to be able to generate the features as they are needed).

5.2.1 On the Fly Interpolation

We could do the interpolation every time the *onNewFrame* function is called. Again, the animation will only be smooth if $execution\_time < 1/FPS$. Here, features would not even need to store an attribute with the time of the interpolation since only features from the current frame are part of the layer (either the features are deleted and created at every frame, or the geometry of a fixed 100 features is changed every tick), which means that we could say that the temporal layers shows all of the features, and only use its GUI to get signals for every frame, and not use the filtering capabilities. However, we could also add features with their timestamp, and use the filtering, and not need to delete them for the next frame. This method is pretty much equivalent to using a buffer of N=1 frame

5.2.2 Buffering Frames

We can do the interpolation for a fixed amount of frames, N, and only need to do it on once in every N *onNewFrame* call. If we want the animation to be completely smooth, the execution of interpolation of the points for these N frames needs to be less than the time it takes for the N frames to be rendered ($execution\_time < N/FPS$). For this solution, even though N frames are buffered, only 1 frame of the animation needs to be rendered at any given time. This means that the features generated need to contain an attribute with the time of the interpolation. The temporal controller can then filter on this attribute to only show the features corresponding to the current frame.

5.3 Experiment 1

In this experiment we try to do as much as possible with python. There are two main steps:

- A one-time query to the database to retrieve the whole trajectories and store them in memory. This takes some time but we don't take it into account since it is a one-time operation.
- The interpolation and addition of the interpolated values to add to the layer. This is done every Nth frame, where N is the number of frames that are buffered (N=1 means

on the fly). Here, there are two main time sinks, the time for the driver to do the interpolation and the time to add the features to the layer.

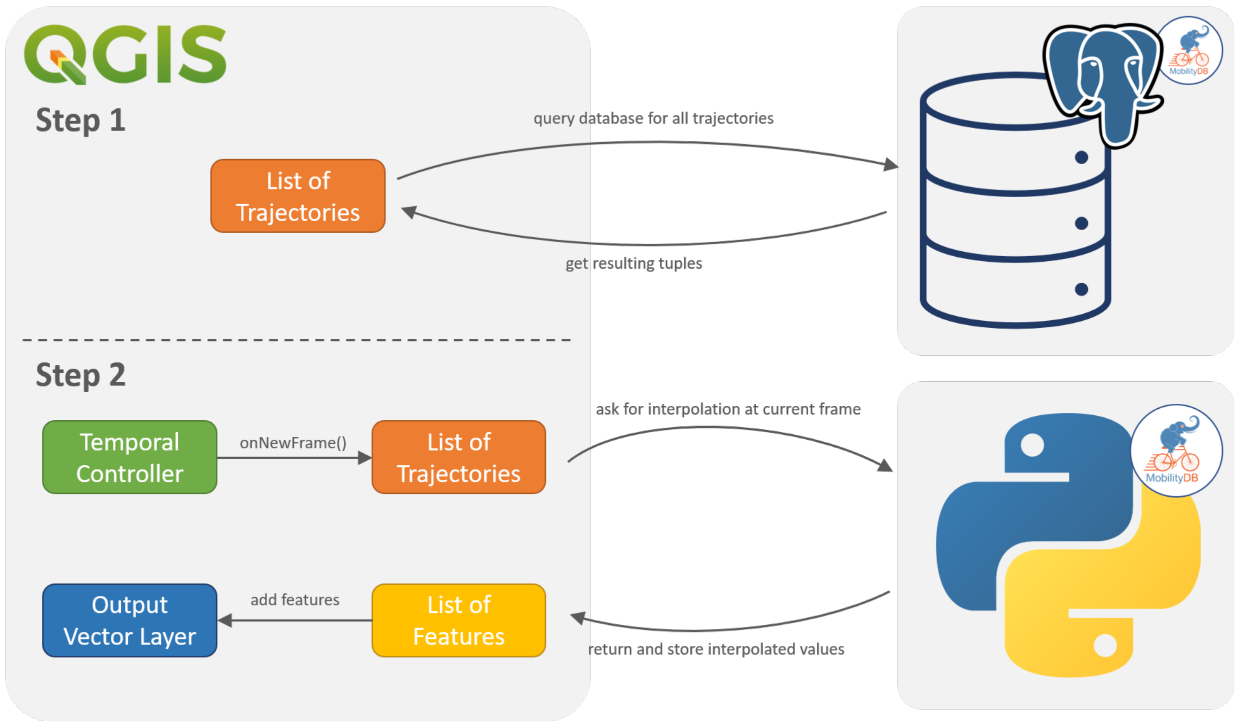

Figure 1: Experiment 1

The experiment was run using a buffer of 50 frames, as well as without a buffer. The results were pretty consistent and depended on the number of frames that were generated. Here are the results for two runs without a buffer.

Nb Features Generated	Interpolation Time	Time to Add Features	Total Time
7	0.052	0.023	0.076
24	0.061	0.045	0.11

Table 1: Performance (Times in Seconds)

We can see that the interpolation time doesn't change much with the number of features generated. This is expected since interpolation is done on 100 features in any case, even if only a quarter (or tenth) actually yield a non-null result. On the other hand, we can see the editing time (i.e. time to add features to the map) also logically increases with the number of features to be added to the layer. By extrapolating these results, we can conclude that if this script was run at each frame (on-the-fly interpolation), the maximum framerate that could be achieved would be around 10 FPS.

Trying the same experiment with a buffer of 50 frames gives the following:

Nb Features Generated	Interpolation Time	Time to Add Features	Total Time
369	2.04	0.48	2.55
1151	2.53	1.45	4.05

Table 2: Performance (Times in Seconds)

Please note that the total time isn't exactly the sum of both times since other negligible time sinks were discarded. We can see that the interpolation time doesn't change much even though the number of features generated is very different. This is expected since the valueAtTimestamp() function (function that does the interpolation) from the driver is called 5000 times in both cases regardless of its return value. The editing time seems to be proportional to the number of features that are effectively added to the map. All in all, if we consider running this script takes between 3 and 4 seconds to generate 50 frames, the framerate's theoretical cap would be around 15 FPS, which isn't much better than on the fly interpolation.

5.4 Experiment 2

In this experiment, we try to query the interpolation of the trajectory directly from the database (i.e. without using the mobilitydb python driver). We can do so using the *postgisexecuteandloadsql* algorithm, which is provided by QGIS and allows us to obtain a layer with features directly.

The two main time sinks are now:

- The processing algorithm time which runs the query, which includes the interpolation time since it is done database side.

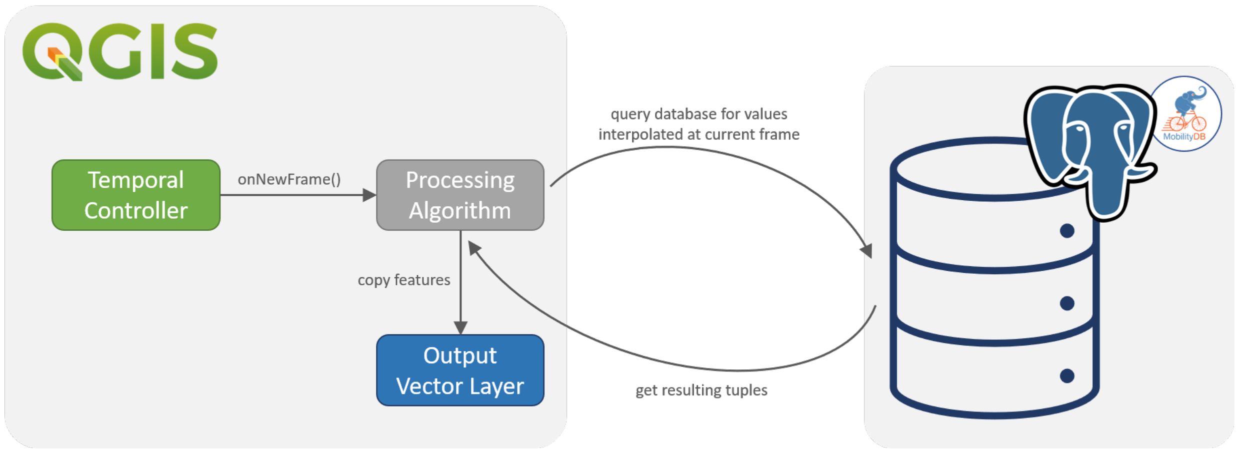

Figure 2: Experiment 2

- The time to copy features from the processing algorithm result to the output layer.

The experiment was run for different buffer sizes, the results for three different runs are presented in table 3. We can see performance here also seems to be capped at around 15 FPS.

Frames Generated	Processing Time	Time to Add Features	Total Time
1 (= on the fly)	0.073	0.046	0.12
50	2.37	0.94	3.31
200	8.73	3.50	12.23

Table 3: Performance (Times in Seconds)

Unfortunately, due to an unknown bug, features that are generated using the algorithm don't actually show on the map unless the temporal controller is turned off, which makes it impossible to use unless the bug can be fixed.

5.5 Experiment 3

Let's revisit experiment 1, but this time, instead of storing the whole tgeompoint column from the database into memory, let's only store the part needed to do the interpolation for the next 50 frames. Since the interpolation will be done on a smaller segment of the trajectory, we can expect it to be much faster. However, the time to add the features to the layer, which is proportional to the number of features, probably won't change much.

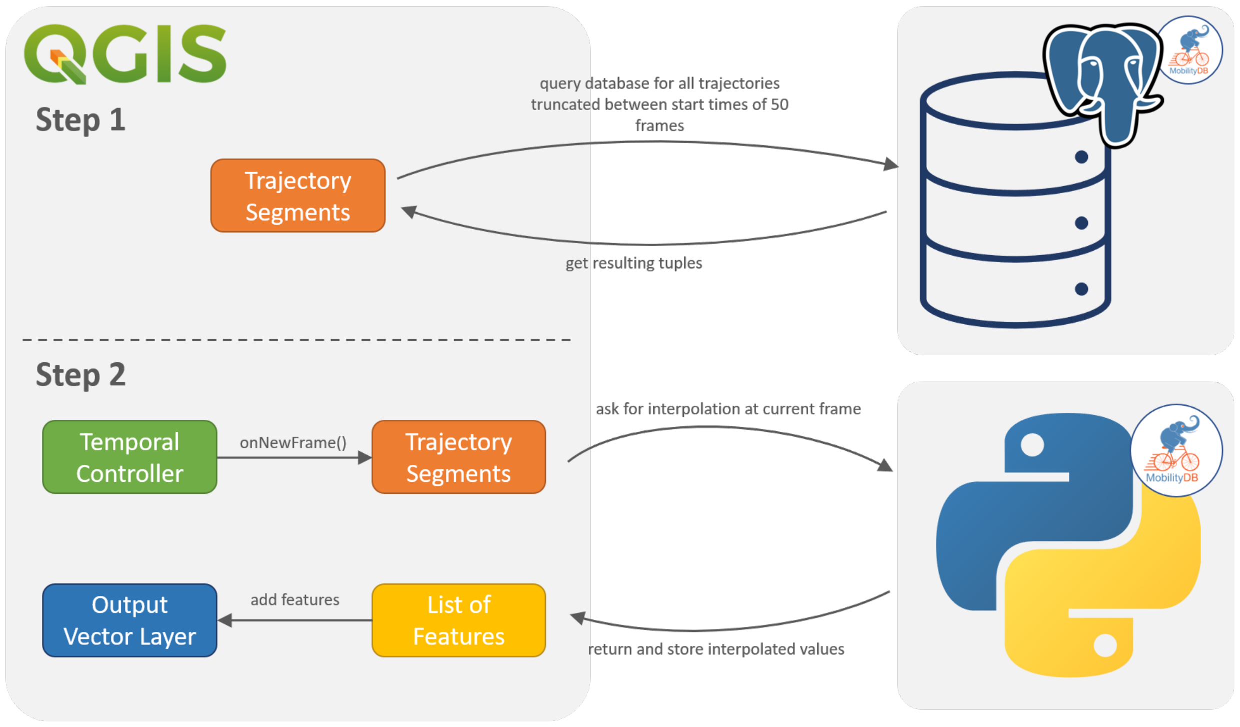

Figure 3: Experiment 3

Here are the results for running the experiment for a single frame: Since we only want the

Nb. Features Generated	Interpolation Time	Time to Add Features	Total Time
24	0.00069	0.064	1.17

Table 4: Performance - On the Fly (Times in Seconds)

features for 1 frame, we are ignoring the time for the query takes to execute (which is included in the total time), since it would only need to be run once every 50 frames. We can see that the interpolation time is now very small. The limiting factor is now the time it takes for features to be added to the layer, or about 0.064 seconds which would result in a framerate around 15 FPS.

Let's now generate the features for all 50 frames of the period

Since the query is done for 50 frames, we need to take its time into account, which is about 1.1 seconds. We can see that the interpolation time is still very low, almost negligible. Since 1200

Features Generated	Query Time	Interpolation	Add Features	Total
1204	1.10	0.16	1.69	3.07

Table 5: Performance - 50 Frames Buffer (Times in Seconds)

features need to be added, the time to do so is relatively large. The main time consumption is due to the addition of the features to the layer and the time to run the query. This brings us to a time of around 2.8 seconds to generate 50 frames (the total time is different since it includes time to connect to the database), or a bit above 15 FPS.

5.6 Summary

From the 3 experiments, we can conclude that the third one seems to give the best results (slightly better than 15 FPS depending on the buffering implementation). Here are the major takeaways from the three experiments:

- In experiment 1, all of the trajectories need to be stored in memory before being able to produce an animation. Also, the interpolation takes quite a lot of time since it is done on a whole trajectory. The theoretical FPS cap is between 10 and 15 FPS depending on the buffering implementation.

- For experiment 2, we make use of a built in QGIS processing algorithm to generate features using a direct query to the database. However, it doesn't perform much better than the first experiment FPS-wise (also capped around 15 FPS). Due to an unknown bug, features aren't rendered by the layer, which is also why this approach couldn't be implemented in practice (even though it looked promising).

- In experiment 3, we use the MobilityDB python driver to query the database and do the interpolation just like in experiment 1, but this time, the query only retrieves a part of the whole trajectory. The interpolation is thus much faster. This experiment gives the best results (a bit above 15 FPS).

Of course, these experiments and their results aren't very scientific but are able to give a quick idea of what would work and what wouldn't.

6 Practical Implementation

The end goal is to be able to connect QGIS to the MobilityDB database and to be able to use the temporal controller to navigate the temporal dimension of the data, which would then be animated inside a temporal layer. This means that we want to generate the frames as they are needed.

Experiment 3 gave the most promising approach on how to produce an animation. We now need to link the feature generation process with the *onNewFrame* function that is called by the temporal controller at every frame when the animation is to be played. The main choice we have is whether to make use of a buffer. The advantage of using a buffer is that it can be used to parallelize the animation rendering and execution with the feature generation. For example, using a buffer of 50 frames, if the features of the current buffer are already generated and added to the layer's data provider (meaning they can be rendered), while the animation is playing the frames of the current buffer, another thread could be generating the frames for the next buffer of the animation.

6.1 Parallelization

To parallelize the animation rendering and feature generation processes we can use a QGIS *task*. This object from the API provides a way to start some computer intensive work in a background thread, without the rest of the application freezing. This is exactly what we want to do when querying data from the database and generating features. However, there is a catch: anything interacting with GUI components (such as the canvas or layers) should only do so in the main thread, and cannot be done inside a task. This means that we can parallelize the query, the creation of the features themselves, but not the addition of the generated features to the layer.

The final implementation works in the following way, and we assume we are using a buffer of 50 frames, and that the current buffer the temporal controller is currently rendering has already been generated:

Every frame the *onNewFrame* function is called by the temporal controller. If the frame number is a multiple of 50 (let's say we are at frame number 0), the function launches a QGIS task that generate the features for the frames between frame 50 and frame 99, which are the frames from the next buffer (frames between 0 and 49 are assumed to already have been generated). Once the task has performed the query to the database to get the corresponding trajectory segments, it creates features with geometries that are the interpolation of between every segment and the timestamps of the 50 different frames. Once this is done, the task ends, and a function from the main thread can then add the features to the layer. If all of the features are added before the temporal controller reaches frame 50, the animation should be smooth.

6.2 Shortcoming

In practice the above example works as expected except for one (pretty major) problem. The task (meaning the interpolation and feature generation procedure) is done in about 20 frames. Which means that at frame 20 of the animation, the features corresponding to frames between 50 and 99 are added to the layer. Adding these features takes less than the 29 remaining frames before frame 50, but the temporal controller stops nonetheless. This means that the animation pauses at frame 20, before resuming at frame 21 once all of the features from the buffer have been added. Unfortunately, since adding the features to the layer must be done from the main thread, there is no way to solve this issue using this approach. Other approaches have been experimented with but weren't successful, which means that navigating through the MobilityDB table using the temporal controller in real time could not be achieved.

6.3 Alternatives and Possible Future Additions

Even though navigating through the data in real time doesn't end up working, it would still be possible to define two timestamps between which to generate an animation. This is not much different than the manual way of exporting data from the database and importing it to QGIS, but can be automated using the QGIS tools, thus saving time and providing a more enjoyable overall user experience.

All of the code used to do so is available inside a few python scripts, which could be run from within QGIS itself in 3 clicks of a button. These scripts could potentially also be compiled into a single plugin depending on the end goal. It would also be possible to extend this approach with other types of spatio-temporal geometries than a simple point, and other information could be retrieved with the query to potentially display more information than the points moving on a map.

For the STIB example, one could imagine changing the color of the point depending on the bus' ID, leaving a cosmetic trail behind them, or changing the size of the points depending on the bus' speed. In this regard, there is still much to explore but is outside the scope of the experiments conducted for this project.

7 Conclusion

The goal of this project was to use QGIS to display spatio-temporal data from a MobilityDB database. Different approaches were explored, from the manual transformation of the data and import into QGIS, to a completely automatic way to view the table's data by navigating through it using QGIS' temporal controller in real time. Experiments were conducted to compare the efficiency of automatically adding the interpolation of the trajectories from the MobilityDB table to a QGIS layer, and an approach where segments of the trajectories were queried and then interpolated using the MobilityDB python driver seemed to be the best choice. Finally, the automatic generation of an animation in real time was attempted using that approach, but failed due to the way QGIS visual components (such as layers) interact with threads.