Real-Time Movement Visualization of Public Transit Data

Hannah Bast
University of Freiburg
79110 Freiburg, Germany
bast@informatik.uni-freiburg.de

Patrick Brosi
geOps
Kaiser-Joseph-Str. 263
79098 Freiburg, Germany
patrick.brosi@geops.de

Sabine Storandt
University of Freiburg
79110 Freiburg, Germany
storandt@informatik.uni-freiburg.de

ABSTRACT

We introduce a framework to create a world-wide live map of public transit, i.e. the real-time movement of all buses, subways, trains and ferries. Our system is based on freely available General Transit Feed Specification (GTFS) timetable data and also features real-time delay information (where available). The main problem of such a live tracker is the enormous amount of data that has to be handled (millions of vehicle movements). We present a highly efficient back-end that accepts temporal and spatial boundaries and returns all relevant trajectories and vehicles in a format that allows for easy rendering by the client. The real-time movement visualization of complete transit networks allows to observe the current state of the system, to estimate the transit coverage of certain areas, to display delays in a neat manner, and to inform a mobile user about near-by vehicles. Our system can be accessed via `http://tracker.geops.ch/`. The current implementation features over 80 transit networks, including the complete Netherlands (with real-time delay data), and various metropolitan areas in the US, Europe, Australia and New Zealand. We continuously integrate new data. Especially for Europe and North America we expect to achieve almost full coverage soon.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.2.8 [**Database Management**]: Database Applications

General Terms

Algorithms, Data Structures, Visualization

Keywords

Public Transit Network, Spatio-Temporal Grid, Real-Time Visualization

SIGSPATIAL '14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA

http://dx.doi.org/10.1145/2666310.2666404

1. INTRODUCTION

A variety of live public transit maps were developed over the last decades, displaying the actual positions of certain vehicles. First approaches dealt with shuttle buses around universities as Stanford[1] or Utah[2]. Meanwhile, larger transportation systems have been visualized, e.g. the complete Swiss train network[3], subways in Munich[4], and the train networks of Germany[5], Austria[6] and the UK[7]. Several other transit agencies provide small live maps for their service range. Unfortunately, all existing live maps are either restricted to a very small area or to a certain type of vehicle (e.g. bus). Moreover most live maps do not feature smooth vehicle movements, but vehicles 'jump' in certain intervals. Also often proprietary data is used, which typically comes in some individually defined format. Therefore combined live maps of several agencies are hard to find.

But global live maps would be beneficial in many aspects. From an operator's point of view this would provide a useful tool to supervise the complete system, track single vehicles, and observe non-scheduled stops or delays immediately. Also the coverage of an area (at a certain time) can be estimated well if all vehicles passing through are displayed. The same holds for evaluating how well-synchronized vehicles from different agencies are. From a user perspective, the easy retrieval of all vehicles that are near-by can help to decide for a certain transportation mode or e.g. a particular bus to take. Also it can tell if you should hurry to catch your selected bus. In combination with a route planner, a live map can inform about delays of relevant vehicles neatly, and may give a hint about alternative transfers.

We present the first live map implementation which has the potential to efficiently visualize the public transit of the whole world. For that purpose, we developed a framework based on GTFS data. GTFS is a general format to describe static timetable data as well as real-time information. Many agencies already provide their data openly in this format, and expectedly their number is going to grow.

The main challenge in creating a world-wide live map is to handle the huge amount of data necessary for the application. For example, in the area of New York alone, over three

[1] `http://transportation.stanford.edu/marguerite/`
[2] `http://www.uofubus.com/`
[3] `swisstrains.ch`
[4] `http://s-bahn-muenchen.hafas.de`
[5] `http://bahn.de/zugradar`
[6] `http://zugradar.oebb.at`
[7] `http://traintimes.org.uk/map`

million times a day a vehicle departs from station. At 8am over 4,700 vehicle movements need to be visualized only in this area (see Figure 1 for an impression). Note, that other world-wide live maps, as e.g. provided for planes[8] or ships[9], hardly ever have to display more than 10,000 elements in total. So extrapolating the New York values to the whole world, it becomes clear that custom-tailored data structures have to be designed to store the transit data, and to answer requests efficiently.

1.1 Related Work

Most live map providers have not published the approach behind their application. A notable exception is e.g. the BusCatcher system [2]. Here GPS based bus positions are displayed on an interactive city map, along with the correct schedule and possibly actual delays. The evaluation of the system focuses on user trials, identifying the lack in system responsiveness as major drawback. For vessels a web architecture for live tracking and visualization was described by Bertrand et al. [3]. The vessel positions are again determined sensor-based (GPS or GSM) combined with inertial systems. These kind of input data is also used for traffic flow estimations and other systems which track vehicles in street networks. For example, the VAST (Visual Analytics for Smart Transportation) system [9] visualizes taxi positions in real-time and is built on GPS and road-sensors as well. The system is used to detect hot spots of taxi usage, to compute good taxi routes and to provide a user with the locations of near-by taxis.

In contrast to all those methods our system is not based on GPS data. In the next section, we will explain in detail why GPS is not a good data source for public transit vehicle tracking. Instead, we will use static timetable data combined with real-time updates. TransitGenie [4] is based on the same input, with the purpose to improve route planning by incorporating delays. But this and related systems do not feature live visualization.

Client/server interfaces for rendering and routing have been studied mostly for street networks. In [10] a scheme was introduced for sending very small data packages (precomputed by the server) to the client, which nonetheless allow for optimal route planning on client-side. In [5] a corridor graph between source and target is computed by the server and transmitted to the client in order to deal with traffic updates efficiently and to take care of periods of time when the user is offline. We are not aware of similar published approaches for public-transit network data.

2. OBTAINING VEHICLE POSITIONS

The very basis of all live map functionality is to obtain the information about all actual vehicle positions continuously. On top of that, we would like to have a comparison to the position the vehicle is *supposed* to be according to the schedule, in order to be able to inform about delays. We will discuss now several data sources with the conclusion that interpolated schedules augmented with real-time information are the method of choice.

[8] `http://www.flightradar24.com/`
[9] `http://www.marinetraffic.com/de/`

2.1 GPS and Sensor Data

The most common way to track vehicles of all kind is to place a GPS-device inside and transmit the position information to a server, where it can be accumulated and processed. A GPS based visualization has the advantage of always displaying the current vehicle position. But multiple reasons speak against GPS as the sole information source:

Low availability and coverage. We are not aware of any transportation agency that provides access to raw GPS positions of their vehicles. And it seems utopian that this will change in the near future. There were some recent attempts to use mobile phone data to determine the mode of transportation of a passenger and also the movement of the vehicle, see e.g. [11]. But again, there is little hope for good coverage world-wide. Additionally, GPS is not available everywhere with a precision high enough for our purpose, and signal blockages due to obstructing foliage or high buildings can not be ruled out.

No fall-back. Relying on raw GPS positions means that if a tracking device fails, the visualized vehicle will stop or disappear. There is no fall-back.

No extrapolation. Raw GPS positions are completely semantic. The data does not reveal the underlying structure of the transit network at all. A typical server output simply says that a certain vehicle currently is at position (x, y). In general, the route of the vehicle, the remaining time until its next arrival, the information if the vehicle is on time, etc., can not be extrapolated from the GPS data alone. So the GPS data has to be combined with static timetable information for those purposes.

Huge amount of data. Receiving GPS positions from all currently moving vehicles world-wide results in heavy traffic and an enormous amount of data to manage, especially for the client/server-interface. We will discuss this in more detail in the next section.

2.2 Interpolated Schedules

Static schedules are much easier to manage than GPS data. In fact, static schedules exhibit none of the drawbacks of GPS described above. With the development of GTFS, there exists a common format to represent public transit schedules and related geodata. A typical GTFS feed exhibits the following components:

- *agency.txt* Holds information about one or multiple service agencies of this feed. This file also holds the timezone. This is important because times are always provided in a HH:MM:SS format, never as absolute timestamps.
- *stops.txt* A list of all stations along with their respective IDs, human-readable names and geographical positions.
- *trips.txt* The headers of all vehicle movements in this feed. Each trip has a service ID which specifies the days it operates on.
- *stop_times.txt* The exact station sequence for each trip. Each station has an arrival and a departure time.
- *calendar.txt* Holds weekly service times referenced by trips.txt.
- *calendar_dates.txt* Holds explicit date-wise service times referenced by trips.txt. These services can extend services specified in calendar.txt with single exceptions, but they can also stand alone.

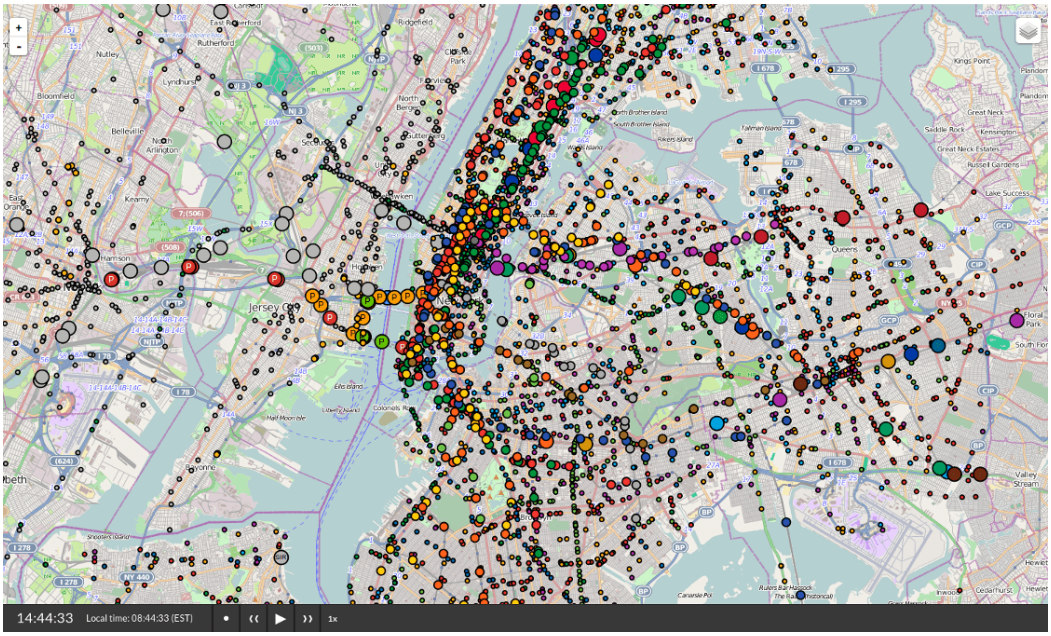

Figure 1: Map zoomed in on New York early in the morning. The high density of public transit vehicles (buses, light rail, subways, ferries) – displayed by our system as coloured circles – indicates the complexity of real-time visualization of large transit networks.

- *shapes.txt* Representation of geographical polylines that describe the exact route a vehicle takes.
- *routes.txt* Groups trips into single services that are presented to users.

Many agencies world-wide already provide their data as a GTFS feed. So the coverage is already high and likely to grow in the next years. One drawback of using static schedules is that they typically only feature station locations and travel times between stations, but no vehicle positions in-between (the *shape.txt* file is optional). So these intermediate positions have to be interpolated somehow to realize smooth vehicle movements. More severely, static schedules lack real-time information. Delays, cancellation and route changes at short notice are not taken into account. This is discussed in the next section.

2.3 Schedules Plus Real-Time Information

We would like to have the best of both worlds: coverage, fall-back, complete schedule information, compact data and real-time information. We achieve this by using a combined approach. The static timetable data provides all of the basic information, like station locations, envisioned arrival and departure times of vehicles at stations, sequences of stations and so on. By interpolating this data we can continuously compute the positions for all relevant vehicles. If live delay information is available, we will incorporate it by updating the respective vehicle positions in a suitable way. This approach adds minimal additional traffic (only a delay value for a time point) and falls back seamlessly to the static schedule if no real-time information is available. There exists an extension to GTFS, called GTFS-realtime, which was developed to specify delay informations in a neat manner. Unfortunately, the coverage of GTFS-realtime is still far behind the general GTFS coverage, but for some countries, like the Netherlands, such information is already available nationwide.
Even with this combined approach the amount of data that has to be managed by the server is huge. Therefore an efficient client/server infrastructure is crucial to enable real-time movement visualization on client side.

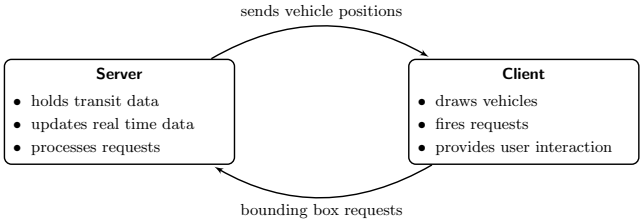

Figure 2: Basic architecture.

3. CLIENT/SERVER INFRASTRUCTURE

We will focus now on a client/server architecture similar to the one sketched in Figure 2. The server manages transit data, receives requests and outputs information that enables the client to display vehicles on the screen. A client can either be a web application, a desktop application, a smartphone app, or something completely different. In this section, we present the two most common interface designs currently used in transit visualization maps.

3.1 Periodic Updates

The basic method of displaying vehicle movements on a map is to fire periodic position requests for a certain area of the network and to draw the results on the map. If client performance is low, this can be the preferred way to visualize vehicles, as the client code can be extremely thin. Most of the live maps mentioned in the introduction are based on periodic updates. If the server uses raw GPS positions as a data source, this is the only practicable interface. Nevertheless, it is also possible to use this approach with interpolated schedule data.

There are several shortcomings in this method. Its biggest advantage, the absence of any client code to calculate vehicle positions, is also its biggest disadvantage. Without a new

position request, vehicles will not move. To achieve a smooth simulation, the requests have to be repeated frequently. This generates a lot of server load. If client and server are physically separated, it also means heavy network traffic. This is especially problematic on mobile devices, where network connections can be extremely slow or even break down completely. If the server connection is interrupted, the vehicle movements stop.

3.2 Spatio-temporal Bounding Boxes

A better interface design allowing for look-ahead requests are *spatio-temporal queries*. In this design, the client does not request vehicle positions, but partial vehicle trajectories within a certain rectangle for a specific timespan Δt (e.g. 2 minutes). We call this request a *spatio-temporal bounding box*. Note that we operate in two-dimensional space.

Definition 1. A spatio-temporal bounding box B_{st} is a 6-tuple $(x_1, y_1, x_2, y_2, t_b, t_e)$ where x_1, y_1 is the lower left (we say: south-west) corner of a rectangle and x_2, y_2 the upper right one (we say: north-east). B_{st} is additionally bounded by a begin time t_b and an end time t_e.

Our client works with spatio-temporal bounding boxes with a fixed[10] difference $\Delta t = t_e - t_b$. Then, as long as the client's viewport stays entirely within the rectangle (x_1, y_1, x_2, y_2), a new such bounding box is required every Δt time units. Note that we can choose the rectangle larger than the client's viewport. Then more data needs to be transferred per request, but with the advantage that the user can move the viewport without the need for a new request. Also note that vehicle delays are only transmitted on each new request. The client therefore tends to become inaccurate for large Δt.

It is easy to see that periodic update queries are a special case of spatio-temporal queries, namely with $t_b = t_e$. Then $\Delta t = 0$, and we theoretically require a new bounding box every 0 seconds. In practice, however, the vehicle positions are refreshed only every r time units anyway. A typical monitor has a screen refresh rate of 60 Hz, in which case r does not neet to be smaller than 16 ms. When the vehicles move fast relative to the viewport[11], we indeed need a refresh rate that small for a smooth visualization. But launching a periodic update query that frequently is unrealistic if considering communication via HTTP requests and a server that has to answer 60 requests per second from possibly hundreds of clients. This is the reason why it is generally not possible to get smooth vehicle movements with periodic updates. We hence prefer spatio-temporal queries.

4. MODEL AND DATA STRUCTURES

The main server task is to answer spatio-temporal queries as efficiently as possible, and to send the relevant information to the client in a format that allows for easy visualization. In this section, we first formally define a vehicle trajectory in space and time. Then we introduce a grid-based data structure, which allows to extract all relevant trajectories inside a spatio-temporal bounding box efficiently. Finally, we point out how live delay information can be incorporated into our model.

4.1 Vehicle Trajectories

To visualize all vehicle movements in a certain area, we first define the path taken by an individual vehicle through (two-dimensional) space and time. We call this the trajectory of the vehicle. A trajectory is described as a list of spatio-temporal way-points.

Definition 2. A spatio-temporal waypoint p is a 3-tuple (x, y, t) where x, y are coordinates on the two-dimensional spatial plane and t is a time stamp.

We call the set of all spatio-temporal waypoints of a trajectory $\mathcal{P}$. The sequence of coordinates x, y of the way-points describes a piecewise linear curve. The time stamps are a parametrization of this curve. Obviously, a vehicle defined by a trajectory can only appear once a day. To model service days of a vehicle in a neat manner, we assign an activity function $\alpha : \mathcal{D} \to \{0, 1\}$ to each trajectory, with $\mathcal{D}$ being the set of all possible dates (e.g. all dates in the validity period of the timetable, typically something like half a year or a complete year). We say that a trajectory with activity funtion α is active for a specific date $d \in \mathcal{D}$ if $\alpha(d) = 1$. Thus, from now on, we describe a trajectory as a tuple $(\mathcal{P}, \alpha)$.

When extracting the trajectory data from GTFS, we are confronted with the problem that spatio-temporal way-points are typically only defined for stations. At a station, the location as well as the arrival and departure times are known. However, vehicle coordinates and time stamps *between* stations are typically not available. Some agencies provide in their GTFS feed shape files, which describe the curve the vehicle moves on – but not at which time the vehicle will be at which position. So we would like to assign timestamps to those coordinates. Also, to allow for smooth vehicle movement later on and to be able to crop trajectories in our data structure, we need a tool for calculating timestamped waypoints between stations.

For that purpose, we assume that a vehicle drives with constant speed between stations. So consider two consecutive waypoints (x, y, t) and (x', y', t'), and let $t_{cur} \in [t, t']$ be a point in time after leaving the first and before arriving at the second waypoint. What are the belonging coordinates x_{cur}, y_{cur}? The relative progress at time t_{cur} on the way from x, y to x', y' can be expressed as:

$$\lambda = \frac{t_{cur} - t}{t' - t}$$

Accordingly, we get $x_{cur} = x + \lambda(x' - x)$ and $y_{cur} = y + \lambda(y' - y)$. The other way around, if x_{cur}, y_{cur} are known, we can compute the respective timestamp using the following value for λ:

$$\lambda = \frac{(x' - x)(y' - y)}{(x_{cur} - x)(y_{cur} - y)}$$

Therefore it yields $t_{cur} = t + \lambda(t' - t)$. These simple calculations can be performed fast enough for on-the-fly interpolation of vehicle trajectories.

4.2 Efficient Trajectory Retrieval

To answer spatio-temporal queries, we need to find the set of all trajectories which intersect the given spatio-temporal

[10] A variable difference could make sense to adapt to very low zoom levels, where the vehicles hardly move, or to adapt to rush hour periods with a high frequency of real-time updates. However, we found this to be an optimization with little gain (performance-wise and quality-wise).

[11] This happens when either the zoom level is large, or when we use the fast-forward button to accelerate time.

box B_{st}. A naive way to do this would be to parse through the list of all trajectories and check, for each trajectory, its relevance for B_{st}. A simple way to assure that all necessary trajectories are identified correctly is to compute the spatio-temporal bounding box $(x_{min}, y_{min}, x_{max}, y_{max}, t_{min}, t_{max})$ for each trajectory and intersect this box with B_{st}. But performing these steps for all available trajectories is impractical. In the Netherlands alone, there are over 100,000 trips per day. Considering the whole world, we have to deal with millions of vehicle trajectories. But of course it makes little sense to check trajectories in Australia when being zoomed in on the Netherlands. Therefore we need efficient data structures to group trajectories, in order to allow for more output-sensitive query times.
A common index structure for geo-coordinates is an R-tree [8]. R-trees group nearby objects and represent them with their minimum bounding rectangle. Multiple bounding rectangles can again be grouped by their respective bounding rectangles on the next level. R-trees allow for fast nearest-neighbor and bounding rectangle requests, and they are commonly used in geographic information systems. However, we do not only aim for finding all trajectories in a certain box, but we want to create a hierarchy among trajectories. City buses, for example, should only appear in the live transit map if we are above a certain zoom level. Subways should not be visible if we are looking at an entire country. Trains should appear on the highest zoom level along with buses or ferries. The following tables shows, which vehicles are shown on which zoom level in our implementation. Zoom level 1 is maximally zoomed out, zoom level 20 is maximally zoomed in.

14 - 20	:	bus
13 - 20	:	streetcar, cable car, funicular
11 - 20	:	subway, ferry
5 - 20	:	rail

We could use multiple R-trees for the different zoom levels. But then we would have to traverse multiple R-tree in order to display vehicles from multiple levels. We therefore propose another data structure, which allows for spatial and temporal indexing, and an easy hierarchical representation.

Multi-Layer Spatio-Temporal Grids. A classical grid divides the spatial plane in grid cells of equal size. It can simply be stored as a two-dimensional array. A trajectory is then assigned to all grid cells it traverses. To model the hierarchy of transportation modes, we use a multi-layer grid. Like an R-tree, it groups multiple bounding rectangles into a bigger rectangle on the next level. Figure 3 gives an example of such a multi-layer grid. Each layer is visible on a certain zoom level. The side lengths of a cell on level i are two times the lengths of a cell on level $i+1$. This resembles the way tiles are generated for web map services like OpenStreetMap or Google Maps, which will turn out to be beneficial for the visualization by the client. More complicated grid-based data structures as described in [1] produce different sized cells on the same hierarchical level and are therefore not as easy applicable here. In Figure 3, a rectangle request for zoom level 15 is highlighted along with the grid cells that have to be checked for trajectories traversing the rectangle. The obvious disadvantage of this approach is that some trajectories have to be indexed for multiple cells. However, this index blow-up can be controlled by choosing appropriate side lengths; see Figure 6 in our experimental evaluation.

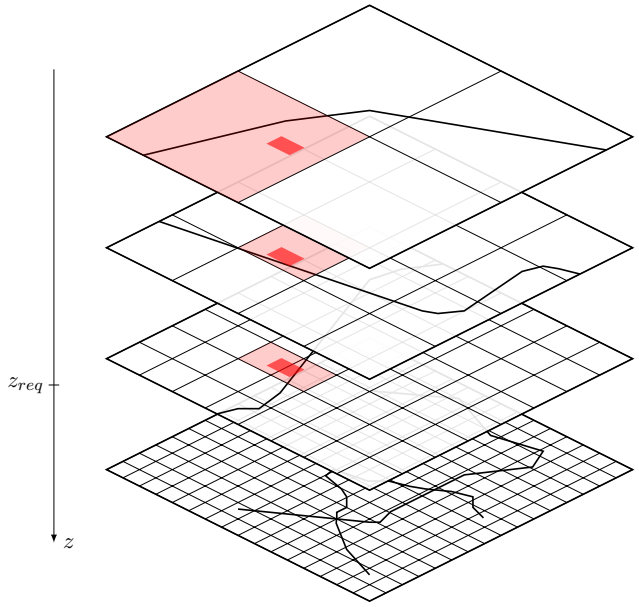

Figure 3: Example of a multi-layer grid. The current view-box and the grid cells that have to be checked are highlighted.

To index the temporal component of a trajectory, we chose a discrete approach that sorts trajectories into multiple date bins, based on their activity function. In particular, we use 9 bins per grid cell. Bins 1-7 model weekdays. For example, a trajectory is indexed in Bin 2 if it is active on Tuesdays. Services that are active on all workdays are stored in Bin 8. Bin 9 is reserved for irregular trajectories that are only active on specific dates. Each bin is stored using a simple array, with items sorted by date (for Bin 9) and time. We now discuss the algorithms for spatial and temporal indexing of trajectories in more detail.

Spatial Indexing. Spatial indexing is not as trivial as it may seem at first glance. Figure 4 illustrates the difficulties of two naive approaches to this problem. In the first approach, a trajectory is spatially indexed into a grid cell c if its minimum spatial bounding box $B_{min}(\mathcal{T}_1)$ intersects c. As shown in the figure, this can lead to unnecessary index items: $\mathcal{T}_1$ never crosses c. In a different approach, $\mathcal{T}_2$ is indexed into c if one or more waypoints of $\mathcal{T}_2$ lie within c. Following this approach, $\mathcal{T}_2$ would not be sorted into c despite the fact that $\mathcal{T}_2$ crosses c three times. Instead of those naive approaches, we use a variant of the Cohen-Sutherland clipping algorithm [7] to determine whether a trajectory really crosses a grid cell c. The algorithm surrounds the bounding box (the grid cell) with 8 rectangular regions. To efficiently calculate clipping points, flags are computed for each endpoint of a straight line that specify the region the endpoint lies in. If, for example, one endpoint lies in the upper right area and the other endpoint lies in the bottom right area, we can safely assume that the straight line does not cross the bounding box. If a non-trivial situation occurs, the algorithm clips the straight line at one endpoint based on the

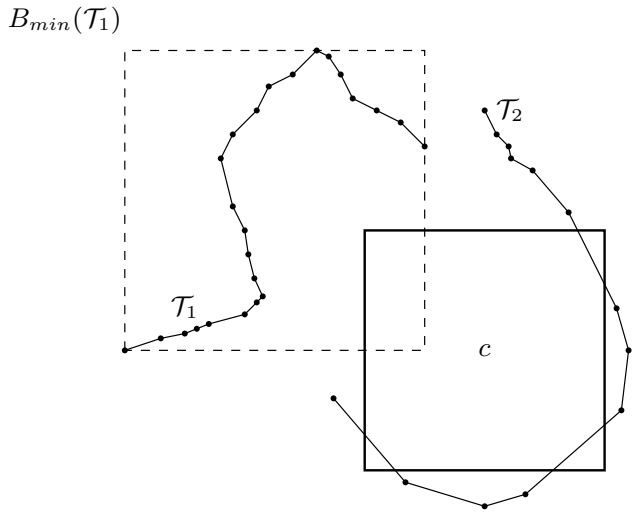

Figure 4: Sorting trajectories into a grid cell c.

area the endpoint is in. In the worst case, this linear interpolation has to be done for both endpoints, which means the algorithm terminates in constant time.

Temporal Indexing. We introduced the concept of an activity function α for a trajectory, which indicates if a trajectory is providing service on a given day. Activity functions are related to and created from services in a GTFS feed. But they are rather an abstraction of the *calendar.txt* file in GTFS. In GTFS, it is possible for a regular service to have negative exceptions (in GTFS, they are defined in *calendar_dates.txt*). Many GTFS feeds only provide positive exceptions in *calendar_dates.txt*, which basically means that each active day of a trajectory is given explicitly. This does not allow for effective indexing and renders bins 1-8 useless. We are left with a single bin that contains each trajectory that crosses the bin's parent grid cell, indexed for every day it is active. In a feed that only provides *calendar_dates.txt*, we have to index each $\mathcal{T}$ per date because otherwise, for each requests, we would have to scan all trajectories inside a grid cell for those that are active on the time of the spatio-temporal request. This means that we either have to accept a bloated data structure or long query times, both of which are unacceptable to us. For example, consider a single trajectory that provides service from Monday till Saturday and whose service is given explicitly for each day in the GTFS feed. If the feed is valid for one year, we would have to create an index item for each of the about 310 days where that trajectory is active. In contrast, by using Bin 8 ("active on working days") we would require only a single index item for this trajectory. Therefore, before doing temporal indexing, we analyze and compress all activity functions.

4.3 Answering Spatio-Temporal Queries

After trajectories have been constructed and indexed in the multi-layer grid, the server enters request mode. The client fires a request in the form of a spatio-temporal bounding box B_{st} and expects the server to return all relevant trajectories. If a trajectory is only partly contained in B_{st}, the server should also clip the trajectory by adding new intermediate start and end points and transmitting only this partial trajectory. If a trajectory leaves B_{st} and re-enters it multiple times, all resulting partial trajectories are grouped by their parent trajectory. Also we decided to perform on-the-fly temporal interpolation of non-timestamped waypoints to save space. In the Netherlands feed alone, there are about 35 million shape vertices (specified in the *shape.txt* file). If timestamps are stored as 32-bit integers this alone would add about 140 MB to the total data set. If temporal interpolation is done at query time instead of in the pre-processing, the total memory consumption of our data is significantly smaller.

So the server first identifies the set of relevant trajectory IDs for B_{st} by invoking the multi-layer spatio-temporal grid, considering affected grid cells and parsing through the trajectories in the bins that match the current date. Then, for every trajectory $\mathcal{T}$ in the resulting set, the server has to perform three basic tasks:

- **A.** find the exact (interpolated) clipping points p_b, p_e which describe the begin/end of a partial trajectory of $\mathcal{T}$ inside B_{st}
- **B.** output all waypoints that lie between p_b and p_e
- **C.** perform on-the-fly temporal interpolation to transform shape vertices into full waypoints

Let $\mathcal{P} = p_1, \cdots, p_k$ be the waypoints of $\mathcal{T}$. To fulfil tasks **A.** and **B.**, every pair of consecutive waypoints p_i, p_{i+1} is checked for temporal crossings into B_{st} (which can be done in constant time) and for spatial crossing by giving the straight line induced by p_i, p_{i+1}, and B_{st} to the Cohen-Sutherland algorithm. This algorithm then computes the correct clipping points in $\mathcal{O}(1)$. Sweeping over all computed entry and exit waypoints of $\mathcal{T}$ and collecting the waypoints in between, the set of all partial trajectories of $\mathcal{T}$ inside B_{st} can be computed. On-the-fly interpolation is performed as described earlier, also only requiring constant time per waypoint. Therefore the total runtime to accomplish all three tasks is in $\mathcal{O}(k)$.

The waypoint coordinates of partial trajectories are subsequently transformed into coordinates relative to the current viewbox of the client and then output as JSON. Sending pixel coordinates instead of latitude and longitude values allows for intermediate rendering by the client without the necessity to perform time-intensive transformations.

Another important optimization is the following kind of trajectory simplification. For higher zoom levels, our server filters out projected waypoints (not time-points) with predecessor distances below a certain threshold (per zoom level). This greatly reduces the size of the JSON result as well the computational load of the client, which is essential for a smooth visualization. Without this step, the server would output complete trajectories with hundreds of waypoints on zoom levels where even distances of multiple kilometers are far below the width of a single projection pixel. The client would then have to do thousands of interpolations without any visible effect at all.

4.4 Modelling Real-Time Data

Real-time public transit data is usually modelled as a list of arrival and departure delays per trip and station. Since August 2011, the GTFS specification is extended by a real-time transit data feed. GTFS-realtime provides support for several kinds of information: Trip updates describe deviations from the official schedule like delays ('stop time update'), cancellations and route changes. This is our main

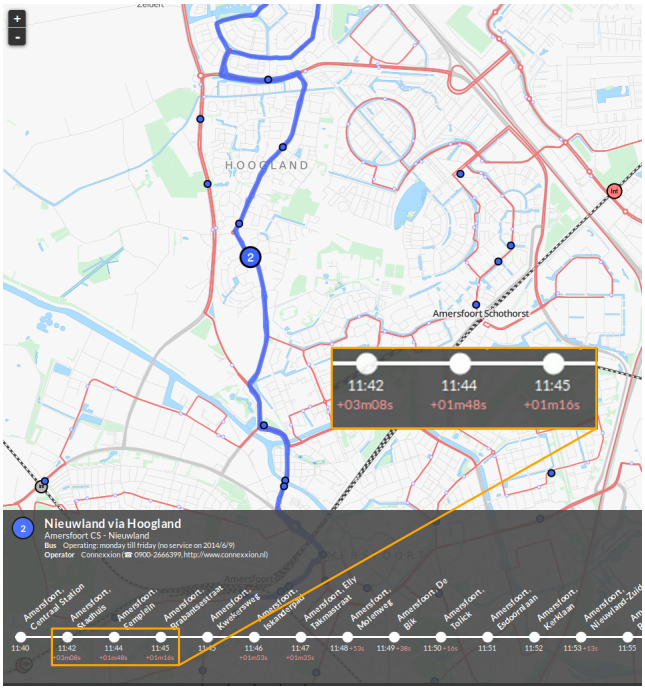

Figure 5: Displaying delay information.

station	0	1	2	3	4	5
δ_{arr}	0s	0s	60s	180s	240s	60s
δ_{dep}	0s	0s	120s	180s	120s	60s
t_{arr}	13:10	13:17	13:24	13:40	13:51	13:58
t_{dep}	13:11	13:19	13:25	13:45	13:52	13:59

Table 1: Example of erroneous delay information. The invalid delay is marked red.

source of real-time data. Updates are given with the IDs of the scheduled GTFS trip they relate to. There is also support for GPS vehicle positions, which can be used for an approach where actual vehicle coordinates are outputted explicitly, along with their current speed. However, that information is currently not provided in any of the GTFS-realtime feeds we know of.

To incorporate delays in our framework, each timestamped waypoint of a trajectory can hold two delays; an arrival delay δ_{arr} and a departure delay δ_{dep}. In the station sequence of a trajectory, a delay δ for station i affects all stations after i and has to be explicitly neutralized by another delay. After the real-time feed has been received and parsed, the server updates each affected trajectory. This is done by locking the trajectory (so that concurrent spatio-temporal request runs will not produce erroneous results) and updating the arrival and delay fields of each timestamped waypoint in $\mathcal{P}$. Both the interpolation algorithm, and the trajectory retrieval and cropping algorithm, described in the previous sections, respect delays by adding them to the timestamps of the affected stations. The server checks each received delay update for validity, because real-time feeds sometimes output invalid delay times. Especially for past stations, an erroneous delay information could break the clipping algorithm completely.

Table 1 gives an example of a corrupted delay feed. For Station 4, δ_{dep} sets the departure time to 13:54, while δ_{arr} sets the arrival time to 13:55, which is of course impossible and would lead to undefined outputs by our clipping algorithm.

After possible delays got filtered, δ_{arr} and δ_{dep} are used in the retrieval and interpolation process, in order to get the correct set of trajectories to be displayed by the client and to calculate the correct (delayed) vehicle position. Delay information is also output by the client, see Figure 5.

5. FURTHER APPLICATIONS

5.1 Combination with Route Planning

We already mentioned that a (mobile) user can benefit from a real-time public transit map in various ways. The user can easily get an overview of all near-by and approximating vehicles at any point in time. Delay information can be obtained by the vehicle position as well as by displayed infoboxes. Possible transfers between vehicles can be identified by comparing the vehicle tracks or zooming in on a particular station.

However, the full potential of the live map could be exploited in combination with a route planning engine. The user inputs a start time, a source and a target location. The route planner then computes a set of good journeys from source to target, considering e.g. travel time and number of transfers. On that basis, the client could now visualize only the movements of relevant vehicles. If the user decides for a specific journey, the client displays only those vehicles that are relevant for this journey. A traveler thus has continuous information about her current position, and the positions of connecting vehicles. When delays make planned transfers impossible, the client should display alternative connections.

5.2 Statistics and Replays

Our grid data structure allows for very efficient answering of spatio-temporal requests; see Section 6 below. This allowed us to implement a fancy "fast-forward" feature, where time can be accelerated by a factor of up to 60.

Moving backwards in time might also be of interest. If a user selects a bus at a certain time, she might be interested if this particular bus was on time yesterday, or to have statistics about how often the bus was delayed in the last weeks. Also if the user took a longer journey, he might welcome the opportunity to watch a replay of the vehicle movements of that journey. As the schedule data is stored by our system anyway, we just have to maintain a database to store delay values to feature statistical analysis and spatio-temporal requests reaching in the past.

5.3 Cleaner GTFS Feeds

In the process of our implementation work, it emerged that our server had become a powerful tool to validate and minimize GTFS feeds. The GTFS parser can handle corrupted feeds (also corrupted delays), is able to optimize the number of shape vertices, dramatically reduces the number of service dates by transforming explicit service dates into weekly services with exceptions and can even add missing timestamps to shape nodes. If the internal transformations would be used to create a new version of the GTFS feed, those would be cleaner and less space-consuming than the original one, in many cases to a large extent.

GTFS feed	$\#\lvert\mathcal{T}\rvert$	#stops	#arr/dep	box area
Vitoria-Gasteiz	6,041	338	122,184	66.84
Budapest	147,556	5,357	2,660,027	1,952
New York Area	300,417	34,948	11,665,443	98,965
Netherlands	548,007	73,293	12,221,953	$2.5{\cdot}10^7$
Combined feed (22 GTFS feeds)	2,507,566	298,535	76,287,281	$\sim 10^8$

Table 2: Datasets used for testing. Areas are given in km^2.

6. EXPERIMENTAL EVALUATION

Our server is written in C++ (compiled with gcc 4.4.6 and optimization flag -Ofast), it holds the transit data in the described multi-layered grid structure, requests real-time data from feeds and responds to spatio-temporal requests sent by the client via a HTTP-interface.

To evaluate the server performance, we ran tests on several GTFS feeds and measured computation times. All tests were executed on a machine with two Intel Xeon E5640 CPUs (8 cores in total) and 66 GB of RAM. To show that the computational cost of naive approaches quickly grows beyond any limit reasonable for live map requests, we started with smaller networks for single cities and gradually chose bigger networks of entire countries. A detailed overview of the dataset parameters is given in Table 2. At the time of testing, real-time feeds were available for New York, San Francisco and the Netherlands. However, delay information usually does not affect the request times at all.
We first give some insight in how to compute good grid cell sizes for our multi-layered data structure and then present server performance results for a variety of requests.

6.1 Grid-Layer Construction

To get optimal results with our multi-layer spatio-temporal grid data structure, the side-length l of the bottom grid has to be chosen wisely. If l is too small, the grid overhead gets too large and grid lookup times exceed even the actual interpolation times. If l is too big, trajectories of vehicles that are not currently moving through the spatial part of B_{st} are given to the interpolation algorithm, resulting in unnecessary interpolations. Figure 6 illustrates this problem. We ran several tests against the GTFS feed of the Netherlands projected onto a 268,435,456×268,435,456 map plane. Buses, streetcars and subways were loaded into a single grid of cell size l, on which a spatial request with a square of side lengths $l_r = 500{,}000$ ($\approx$ 45km) was executed. We measured the average index count per trajectory as well as the number of trajectories that were given to the interpolation algorithm. Note that only trajectories that were active on a normal Monday were output. With smaller l, the number of index items per trajectory explodes. With bigger l, more and more unnecessary potential trajectories $\mathcal{T}_{pot}$ are given to the interpolation algorithm. At $l = 2.5 \times 10^6$, a single cell almost spans the whole network and the number of $\mathcal{T}_{pot}$ reaches the total number of trajectories active on a Monday. Note that the number of index items per $\mathcal{T}$ never reaches 1 because there are some trajectories that have to be indexed temporally more than one time.

6.2 Server Performance

We tested the server performance by running several spatio-temporal queries against the datasets described above. Each

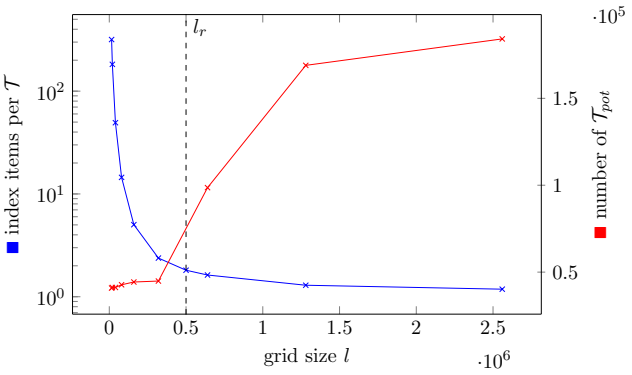

Figure 6: Effects of spatial grid size l. We measured the number of trajectory index items and the number of output potential trajectories $\mathcal{T}_{pot}$ inside a spatio-temporal bounding box with side length 500.000 (ca. 45 km). Data is taken from the complete Netherlands GTFS.

query requested the partial trajectories for the next 60 minutes and was run twice: once in the morning rush hour with t_b = 8:00:00 and once in the late evening with t_b = 23:00:00. In a first step, we did a total area scan that requested the partial trajectories of the entire dataset ($z = 20$). We then proceeded with a spatio-temporal bounding box request that resembles the requests fired by real clients. The request box with area $A \approx 10$ km^2 was hand-picked from the center of the dataset. The request zoom level was $z = 20$. A third request covered an area of about 60,000 km^2 at zoom level $z = 9$.

All of these queries were answered using two different approaches. First, we ran the test using a naive approach where trajectories were stored as an unsorted list. We then did the same request on a layered grid with base-cell side length $l = 500{,}000$ (approximately 40 km in central Europe.) Note that even in the naive approach, we implemented various simple optimizations. Consider, for example, a query that requests all partial trajectories between 8:00 and 9:00. If the naive approach finds a trajectory that leaves the first station at 9:20 and arrives at the last station at 10:50, the trajectory is skipped.

We measured the total query time, the number of output partial trajectories and the number of *affected* trajectories. We say a trajectory $\mathcal{T}$ is affected during a query if a non-trivial computation has to be executed on $\mathcal{T}$. Non-trivial computations include, for example, clipping and a call to the trajectory's activity function. Calls to getter functions are considered trivial. A good measurement for the scalability of an approach is the ratio of affected trajectories and partial trajectories in the output. If, for example, the server outputs 500 partial trajectories and only had to look at 700 trajectories at all, we consider this a good result.
For small networks like Vitoria-Gasteiz, the naive approach yields reasonable good results. However, even for this small dataset, request times are 5 to 20 times higher than for the grid layer approach. Despite the fact that Vitoria-Gasteiz does not have any vehicles that are displayed at zoom level 9, the naive approach still has to check 6,000 trajectories for the $z = 9$ box request. For bigger datasets like Budapest, box requests at ground level are still 20 times faster with

	Naive		Grid	
	8am	11pm	8am	11pm
Total area scan				
time in ms	500	480	100	50
#partial trajectories	3.3 k	1.6 k	3.3 k	1.6 k
#affected trajectories	147.6 k	147.6 k	3.8 k	1.8 k
request area in km^2	1.9 k	1.9 k	1.9 k	1.9 k
Box request				
time in ms	457	460	27	12
#partial trajectories	1 k	434	1 k	434
#affected trajectories	147.6 k	147.6 k	1.5 k	628
request area in km^2	9.9	9.9	9.9	9.9
Box request with $z = 9$				
time in ms	328	326	1.8	< 1
#partial trajectories	73	39	73	39
#affected trajectories	147.6 k	147.6 k	73	39
request area in km^2	60.2 k	60.2 k	60.2 k	60.2 k

Table 3: Testing results for Budapest.

	Naive		Grid	
	8am	11pm	8am	11pm
Total area scan				
time in ms	1.3 k	1.1 k	478	155
#partial trajectories	11.5 k	3.6 k	11.5 k	3.6 k
#affected trajectories	300 k	300 k	17.2 k	3.8 k
request area in km^2	98 k	98 k	98 k	98 k
Box request				
time in ms	1.1 k	1 k	87	26
#partial trajectories	1.1 k	353	1.1 k	353
#affected trajectories	300 k	300 k	3.4 k	783
request area in km^2	10.7	10.7	10.7	10.7
Box request with $z = 9$				
time in ms	676	670	13	5
#partial trajectories	485	161	485	161
#affected trajectories	300 k	300 k	559	227
request area in km^2	60.8 k	60.8 k	60.8 k	60.8 k

Table 4: Testing results for New York + New Jersey.

the grid layer than with the naive approach, see Table 3.

In Table 4, the $z = 9$ box request for the New York City area at 11pm yields 161 partial trajectories. To output this (small) number of $\mathcal{T}^{par}$, the naive approach has to look at 300,000 trajectories, while the grid only has to look at 227. Similar results can be seen in Table 5 for the Netherlands feed.

To the best of our knowledge, the Netherlands GTFS is (by far) the biggest transit feed available. It can be considered as 'complete', meaning that *every* public transit vehicle in the country is included with its full polyline. Still, despite the fact that the total number of trajectory vertices is 16 times as high as in the Budapest feed (the number of arrival/departure events is nearly 5 times as high), the time to output about 1,000 partial trajectories at ground level in the city center of Amsterdam is only slightly bigger than the time to output about 1,000 $\mathcal{T}^{par}$ in the city center of Budapest. The higher time for the box request in Amsterdam can be explained by the higher network density in the city and mainly because of the higher vertex density in the Netherlands feed. Note that the $z = 9$ box requests in Table 5, which nearly covers the whole country of the Netherlands, only takes 23 ms during the morning rush hour. At this zoom level, only trains are returned by the server.

To show that our back-end has the potential to handle the public transit network of the whole world, we ran the queries against a dataset consisting of 22 feeds from around the world. These feeds include three entire countries (Switzerland, Sweden and the Netherlands). The other feeds are: public transit in the cities of Albuquerque, Boston, Los Angeles, Miami, San Francisco, Portland, Chicago (all USA), Quebec, Montreal (both Canada), Manchester (UK), Budapest (Hungary), Rennes (France), Turin (Italy), Vitoria-Gasteiz (Spain), Auckland, Wellington (both New Zealand), Adelaide (Australia) and the public transit in the areas of Freiburg (Germany) and New York (USA). The parameters of this combined feed are listed in Table 2.

Table 6 shows that even when run against the combined dataset, the grid layer approach is still able the handle requests very fast. For the box requests, computation times are nearly the same (± 1 ms) as for the Netherlands. This

	Naive		Grid	
	8am	11pm	8am	11pm
Total area scan				
time in ms	2.1 k	1.9 k	706	310
#partial trajectories	11.5 k	5.1 k	11.5 k	5.1 k
#affected trajectories	548 k	548 k	18.1 k	8.2 k
request area in km^2	25 M	25 M	25 M	25 M
Box request				
time in ms	1.82 k	1.82 k	51	33
#partial trajectories	911	556	911	556
#affected trajectories	548 k	548 k	2 k	1.2 k
request area in km^2	10.7	10.7	10.7	10.7
Box request with $z = 9$				
time in ms	1.28 k	1.28 k	23	14
#partial trajectories	707	451	707	451
#affected trajectories	548 k	548 k	986	533
request area in km^2	60 k	60 k	60 k	60 k

Table 5: Testing results for the Netherlands.

	Naive		Grid	
	8am	11pm	8am	11pm
Total area scan				
time in ms	8 k	9 k	1.3 k	1.7 k
#partial trajectories	40.5 k	40.3 k	40.5 k	40.3 k
#affected trajectories	2,5 M	2,5 M	67.7 k	67.1 k
request area in km^2		$\sim$100 M		
Box request				
time in ms	7.5 k	8.2 k	51	33
#partial trajectories	911	556	911	556
#affected trajectories	2,5 M	2,5 M	2 k	1.2 k
request area in km^2		10.7		
Box request $z = 9$				
time in ms	5.9	5.9	23	15
#partial trajectories	707	451	707	451
#affected trajectories	2,5 M	2,5 M	986	533
request area in km^2		60 k		

Table 6: Testing results for the combined feed. Box request areas are the same as in Table 5. Request times are CET.

proves that loading more feeds in our system does not affect the runtime of spatio-temporal queries covered by a single feed. With that observation, we can conclude that our implementation (given enough memory) can provide real-time vehicle movements all around the whole world efficiently. In combination with a client that visualizes this movements in an appealing manner, we now have a framework for a fully functional world-wide live public transit map.

We encourage the reader to visit our live demo at `http://tracker.geops.ch` . At the time of this writing, over 80 feeds have been incorporated, which is even more than in our experiments.

7. CONCLUSIONS AND FUTURE WORK

We presented a scalable approach for the live visualization of real-time public transit data. We described a strategy that handles vehicle trajectories as piecewise linear curves on the projected map plane and combines delay information with static schedule data to provide a visualization that is both close to reality and robust against missing real-time updates. We discussed the advantages of the approach compared to periodically updated GPS positions and developed a suitable client/server architecture. Evaluation showed that even in dense transportation networks, the average request times for the server are very low (usually between 1 and 80 ms). Further tests showed that the request times stay the same if the dataset grows in both the number of trajectories and covered area.

There are still several aspects in which our back-end could be improved. One important aspect is the space consumption of the whole system and the network traffic. Shape vertices use a lot of memory and slow down the clipping and interpolation algorithm. We mentioned that our server filters out waypoints whose distance is below a certain threshold. But there are still many shape vertices that are redundant, for example, sequences of waypoints on a straight line. Line simplification algorithms like Ramer-Douglas-Peucker [6] or the algorithm by Suri [12] could be applied to the dataset before storing it into the grid layer.

Another aspect is that shape information is often missing from GTFS feeds, especially for buses. Buses stick to street networks, and interpolating the way between two bus stops by a straight line does not do justice to this fact. Interpolation by computing shortest paths between consecutive bus stops in the street network (e.g. extracted from OpenStreetMap) could improve those shapes significantly.

There are some simplifications used in our model in order to keep processing times low and algorithms simple. For example, we assume constant travel speed or straight lines between waypoints. We think that user experience is not significantly impaired by these assumptions. Still, slowing down before a stop and initial acceleration after a stop could be added to our model. A similar embellishment would be to replace the simple linear interpolation between waypoints by interpolation with curves of higher order.

Even without these optimizations, we have already demonstrated that esthetically pleasing and scalable real-time public transit visualization is possible. With the increasing availability of static and real-time GTFS feeds, we hope that soon our live demo will indeed cover the entire world.

8. REFERENCES

[1] Walid G Aref and Hanan Samet. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 265–272. ACM, 1990.

[2] Michela Bertolotto, Ailish Brophy, Alan Martin, O Gregory, Robin Strahan, Eoin McLoughlin, et al. Bus catcher: A context sensitive prototype system for public transportation users. In *Web Information Systems Engineering Workshops, International Conference on*, page 64. IEEE Computer Society, 2002.

[3] Frédéric Bertrand, Alain Bouju, Christophe Claramunt, Thomas Devogele, and Cyril Ray. Web architecture for monitoring and visualizing mobile objects in maritime contexts. In *Web and Wireless Geographical Information Systems*, pages 94–105. Springer, 2007.

[4] James Biagioni, Adrian Agresta, Tomas Gerlich, and Jakob Eriksson. Transitgenie: a context-aware, real-time transit navigator. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 329–330. ACM, 2009.

[5] Daniel Delling, Moritz Kobitzsch, Dennis Luxen, and Renato Fonseca F Werneck. Robust mobile route planning with limited connectivity. In *ALENEX*, pages 150–159. SIAM, 2012.

[6] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[7] James D Foley, Andries Van Dam, Steven K Feiner, John F Hughes, and Richard L Phillips. *Introduction to computer graphics*, volume 55. Addison-Wesley Reading, 1994.

[8] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[9] Siyuan Liu, Ce Liu, Qiong Luo, Lionel M. Ni, and Huamin Qu. A visual analytics system for metropolitan transportation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 477–480, New York, NY, USA, 2011. ACM.

[10] Niklas Schnelle, Stefan Funke, and Sabine Storandt. Dorc: Distributed online route computation-higher throughput, more privacy. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, pages 344–347. IEEE, 2013.

[11] Leon Stenneth, Ouri Wolfson, Philip S Yu, and Bo Xu. Transportation mode detection using mobile phones and gis information. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 54–63. ACM, 2011.

[12] Subhash Suri. A linear time algorithm for minimum link paths inside a simple polygon. *Computer Vision, Graphics, and Image Processing*, 35(1):99–110, 1986.