ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Fixed-shape Regions in Moving Object Databases

Ligne du sous-titre du mémoire

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en Informatique à finalité spécialisée

Maxime Schoemans

Directeur
Professeur [Prénom Nom]

Co-Promoteur
Professeur [Prénom Nom]

Superviseur
[Prénom Nom]

Service
[Nom du service]

Année académique
2019 - 2020

Résumé

Résumé

Abstract

Abstract goes here

Acknowledgments

I want to thank...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Currently, location data is everywhere: which phone, car or laptop does not come with a GPS device included? All of these devices generate a large amount of spatio-temporal data, and they are not the only ones. Smart thermostats produce streams of continuously changing temperature measurements; satellite imaging helps keep track of the evolution of forest boundaries, forest fires, glacier extends and more. A lot of applications make use of this data, and it is thus crucial to be able to store and query this data efficiently. For this reason, Moving Object Databases are used to store these temporal or spatio-temporal data. These databases make use of the inherent structure of moving objects to save data more efficiently and implement specific operations on these data types.

[Talk about the heavy ongoing research in this domain, mostly for point data. Describe moving regions and their importance. Than say that this master thesis will try to extend the work done on moving regions, mainly for fixed-shape regions.]

1.2 Previous work

[Maybe more than the 2 subsection?]

1.2.1 Moving object databases and MobilityDB

[Talk about previous moving object databases and MobilityDB]

1.2.2 Research on moving regions

[Mostly the two research papers about moving regions]

1.3 Objective of the thesis

Temporal or spatio-temporal data exists in many types. It ranges from simple values evolving through time (ex. temperature) to positions or even regions moving through time. This master thesis expands on one type of moving object: a fixed-shape moving region. The first objective of this thesis is to describe the theoretical concepts

needed to represent and use these moving regions in a moving object database. A second part revolves around implementing these concepts in MobilityDB and describing the practical challenges that appeared during this implementation.

[Talk more about the long-term objective of creating a basis/structure for implementing moving regions in practice, starting with fixed-shape regions. First, adding moving region concepts to MobilityDB, then applying the algorithms from the paper or implementing new ones to apply the needed operations on these objects. Lastly test this implementation on a use case.]

1.4 Structure of the thesis

The structure of this master thesis is as follows. First, theoretical concepts are explained. The different types of moving objects are described and illustrated. The representation and operations of fixed-shape moving regions are then defined. Then, the practical implementation, and challenges thereof, are described. Lastly, a use case illustrates the usability of the resulting implementation. A summary, followed by possible future work concludes the thesis.

[Will be adapted when the structure becomes clearer]

Chapter 2

Theoretical Background

2.1 Terms and Concepts

- Moving object
- Moving region
- Fixed-shape moving region
- MOD
- MobilityDB

2.2 Moving objects

There exist a large variety of moving objects. These object can differ in two main ways. Either the type of their value differs, or they are using a different interpolation method.

2.2.1 Value types

In theory, for every non-temporal type, there could exist a moving object that extends this type with a temporal dimension. However, in this master thesis, we will only focus on the main types used in this field.

- Scalar types: boolean, string, integer, float
- Point types: Either 2d or 3d PostGIS Point types, or network-constrained points called npoints.
- Region types: Defined using the PostGIS Polygon type, these regions can either be deforming or fixed-shape.

2.2.2 Interpolation methods

Since the values of a moving object are not stored at each time step, intermediate values have to be recomputed using a specific interpolation method. Moving objects with the same value type, but different interpolation methods will thus behave

differently and can be considered as different objects. The two main interpolation methods used are:

- Stepwise interpolation (or piecewise constant interpolation): The value of the object does not change between two measurements, but jumps instantly at the time of the next measurement.

- Linear interpolation: The value of the object changes linearly between two saved values.

Discrete value types such as integer or boolean can only use the stepwise interpolation, but continuous types can use both. More complex interpolation methods, such as polynomial or spline interpolation could also be used, but this is not discussed in this work.

2.2.3 Examples

	Stepwise	**Linear**
Boolean	State of a lightbulb in a room.	N/A
String	Name of the world record holder for the 100m.	N/A
Integer	Number of people in a bus.	N/A
Float	World record time for the 100m.	Temperature in a room.
Point	Position of the heighest building in the world	Position of players on a court
NPoint	Position of temporary radars on the side of the road	Position of a car
Fixed-shape Region	???	Movement of a floating city
Deforming Region	Cadastre	Spread of a forest fire

2.3 Fixed-shape regions

In the previous section, the broad range of moving object has been presented. A lot can still be said about these different types of object, however, this thesis is mainly going to focus on diving deeper into the fixed-shape moving regions. These moving regions can generally be represented using simple polygons with zero or more holes. When we talk about a moving fixed-shape polygon, we talk about a certain polygon moving and rotating through time.

2.3.1 Representation

The first important question that can be asked about fixed-shape moving regions is: how can they be efficiently represented? In the following we will suppose that the polygon is defined at multiple moments in time and we assume that the position of the polygon at intermediate time steps can be found via some kind of interpolation. This is analogous to a moving point (ex: a car), where the position is measured and saved at multiple moments in time, and is supposed to be linearly interpolated. Seeing as the polygon is of fixed shape, storing the position of the polygon at each time step involves storing some redundancy and is thus not really efficient. The first idea is thus to to store the polygon only for the first time step, and then compute and store transformation vectors for the subsequent time steps. This idea has already been explored in [3] and their proposal is to store what they call a transformation unit. This transformation unit is defined between a start instant t_s and an end instant t_e, and stores the following information.

$$T = (C, v_0, v, \theta_0, \theta, t_s, t_e) \tag{2.1}$$

Where v is a translation vector and θ is an angle. This representation assumes that both the start and the end transformation of the region is stored in the transformation. This is necessary if we suppose that the transformation center can change or if the subsequent transformations are not contiguous to each other. If however we suppose that the transformation center is always the same and that these transformations are contiguous, then there is a lot of redundant information in these transformation units. Since in the standards [Maybe more info about the standards?], there is no mention of the rotation center, and the moving regions are all defined using polygons and not transformations, the rotation center can be chose arbitrarely. Our following supposition is thus that the rotation center of all moving regions is the centroid of the polygon representing the region. Below is a new proposition for the representation of a fixed-shape moving region:

$$fmregion = (P_{ref}, [(T_0, t_0), (T_1, t_1), ..., (T_n, t_n)]) \tag{2.2}$$

, where P_{ref} is a simple polygon with zero or more holes, and T_i is the transformation vector defining the position of the polygon at time t_i. The instants t_i have to be sorted in increasing order for all i. Since we suppose that the center of rotation is always the centroïd of the polygon defining the region, the information stored in a transformation vector only contains the translation an rotation fo the region.

$$T_i = \left(t_x^i, t_y^i, \theta^i\right) \tag{2.3}$$

In most cases, the position of the polygon at time t_0 is used as reference polygon, and we have thus $T_0 = (0, 0, 0)$.

To compute these transformation vectors starting from two regions at two different time steps, multiple techniques are possible. [3] proposes a way to compute these transformation units that allow for some slight differences between the start and the end region. In the following however, we will suppose that the start and end regions are identical and that the corresponding points on each polygon are known.

With this assumption made, it is sufficient to take the center of rotation and one other point of each polygon to compute the transformation. We define the start and end centroids, additional points and transformation vectors as follows:

$$\begin{aligned} C^i &= (c_x^i, c_y^i) \\ P^i &= (p_x^i, p_y^i) \qquad i \in \{start, end\} \\ T^i &= \left(t_x^i, t_y^i, \theta^i\right) \end{aligned} \tag{2.4}$$

Since every transformation is defined relative to a reference polygon, we also need to define this reference polygon. In this example case we will take the start region as being the reference polygon. This means that the start transformation vector will be zero: $T_{start} = (0, 0, 0)$. The transformation vector for the end region can then be computed in the following way. First the translation component is computed by doing the difference between the positions of the start and end centroids.

$$\begin{aligned} t_x^{end} &= c_x^{end} - c_x^{start} \\ t_y^{end} &= c_y^{end} - c_y^{start} \end{aligned} \tag{2.5}$$

Then, the end polygon (only the needed point) is moved back to its starting position by cancelling the translation.

$$\begin{aligned} P^{end\prime} &= (p_x^{i\,\prime}, p_y^{i\,\prime}) \\ p_x^{end\prime} &= p_x^{end} - t_x^{end} \\ p_y^{end\prime} &= p_y^{end} - t_y^{end} \end{aligned} \tag{2.6}$$

Finally, the angle between the start and end polygon is computed.

$$\theta^{end} = \angle P^{start} C^{start} P^{end\prime}, \quad \theta^{end} \in \,]-\pi, \pi] \tag{2.7}$$

2.3.2 Operations on fixed-shape regions

Intro on operations

List of operations with input to output types

- atinstant/getValue
- interpolate
- traversed area
- point inside
- distance?
- union or diff ?

AtInstant/getValue

When defining a moving region over a certain time period, it is important to be able to know what the position of the region is at any instant in time during that time period. This is done using the [getValue / atInstant?] function. This function takes a moving region and an instant as input and returns the position of the region at that moment in time. To compute this position, we first have to compute the transformation of the region at that moment. When the transformation with respect to the reference region is know, we can just apply this transformation and we obtain the needed region. Applying this transformation to the reference region means applying it to every point of the region. With $P = (p_x, p_y)$ being the start point and $T = (t_x, t_y, \theta)$ being the transformation, we compute the end point $P' = (p_x', p_y')$ as follows:

$$\begin{aligned} p_x' &= cos(\theta) * p_x - sin(\theta) * p_y + t_x \\ p_y' &= sin(\theta) * p_x + cos(\theta) * p_y + t_y \end{aligned} \tag{2.8}$$

The main issue now is to compute the correct transformation for the given instant. Suppose that the position of the region is defined at time steps $[t_0, t_1, t_2, ..., t_n]$ and that the region is well-defined between these time steps.

$$fmregion = (P_{ref}, [(T_0, t_0), (T_1, t_1), ..., (T_n, t_n)]) \tag{2.9}$$

If the given instant is one of the t_i's, then the corresponding transformation will be T_i and the work is done. If the instant is before t_0 or after t_n, then the region will return an empty region [empty or error or null?], since the region is not defined at that moment. Lastly if the instant is between t_i and t_{i+1}, then the transformation has to be computed by interpolating between the transformations T_i and T_{i+1}. This interpolation is explained in the following subsection.

Interpolation

Suppose that we know the position of a moving region at times t_i and t_{i+1} and that we want to know what the position of that region is in between these instants. Since we previously supposed that the respective points of different regions/polygons are known, a first possible interpolation method would be to linearly interpolate each respective vertex of the start and end polygon. An example of this can be seen below.

[Add an example]

As we see on above example, this seems to work well when the region is only translating. However, when the moving region is also rotating, this interpolation method fails to maintain the fixed-shape aspect of the region, and produces regions that are deforming along the translation. This comes from the fact that this interpolation method assumes that the vertices of the region move freely and independently of each other. For fixed-shape moving region this is unfortunately not the case, so this interpolation method cannot be used when rotations are involved. [Add an example]

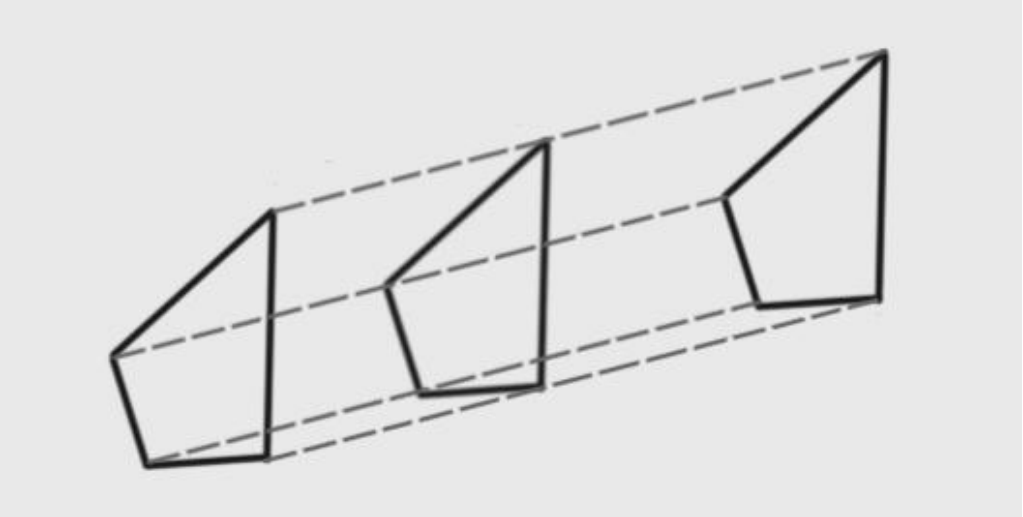

Figure 2.1: Example of a correct linear interpolation of vertices between two regions

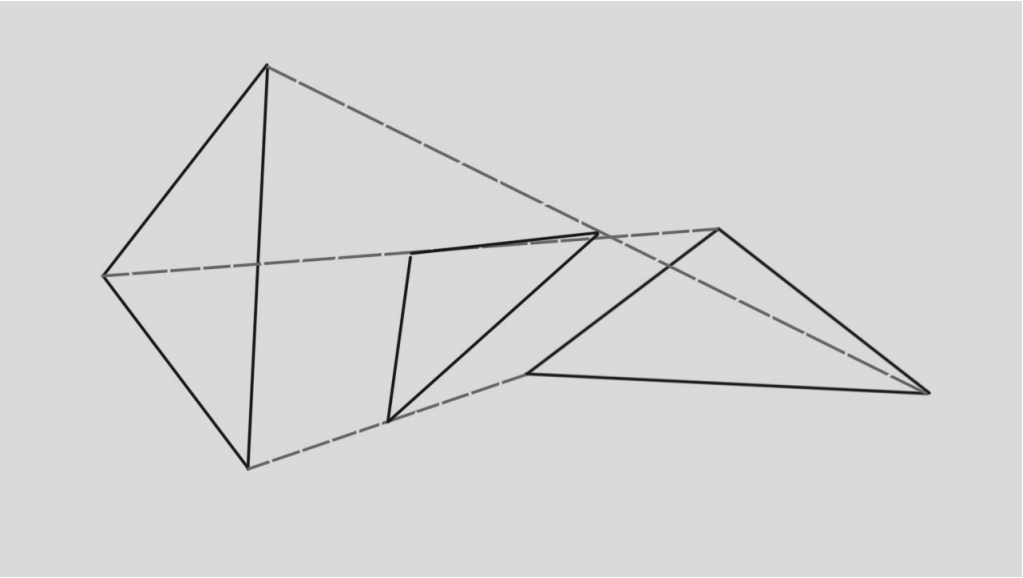

Figure 2.2: Example of a wrong linear interpolation of vertices between two regions

The solution for this is to interpolate the transformation instead of the vertices of the region. When applying a transformation (translation + rotation) to a region, the shape of the region will be maintained, which is what we want. To interpolate two transformations, two steps have to be done. First the translation vectors have to be linearly interpolated. Then the rotation has to be interpolated using a spherical interpolation method. For example: suppose that we want the transformation at time $t_0 < t_i < t_1$, with the transformations $T_0 = (t_x^0, t_y^0, \theta^0)$ and $T_1 = (t_x^1, t_y^1, \theta^1)$ at times t_0 and t_1 known. The transformation $T_i = (t_x^i, t_y^i, \theta^i)$ can then be computed in the following way.

$$
\begin{aligned}
r &= \frac{t_1 - t_i}{t_1 - t_0} \\
t_x^i &= t_x^0 * r + t_x^1 * (1 - r) \\
t_y^i &= t_y^0 * r + t_y^1 * (1 - r) \\
\theta^i &= slerp(\theta^0, \theta^1, r)
\end{aligned}
\tag{2.10}
$$

To compute θ^i, we have to take into account the fact that we are interpolating an angle and that we want $\theta^i \in]-\pi, \pi]$. Suppose without loss of generalization that $\theta^0 <= \theta^1$ (Otherwise just call *slerp* with the parameters $slerp(\theta^1, \theta^0, 1 - r)$). Two different cases have to be considered: the case where $\theta^1 - \theta^0 <= \pi$ and the case where $\theta^1 - \theta^0 > \pi$. Both cases are illustrated below.

[graph with interpolation of angles]

The [red?] portion of the circle corresponds to the values that the interpolated angle can take depending on the value of r. In the first case, a linear interpolation of the angles gives the correct result. In the second case, however, we have to take into account the fact that both angles are between $-\pi$ and π. The interpolated angle can then be computed by first adding 2π to the smallest angle, interpolating linearly the resulting angles and then substracting 2π from the resulting angle if it is larger than π.

$$
\begin{aligned}
&\text{case } 1 : \theta^i = \theta^0 * r + \theta^1 * (1 - r) \\
&\text{case } 2 : \theta = (\theta^0 + 2\pi) * r + \theta^1 * (1 - r) \\
&\theta^i = \begin{cases} \theta - 2\pi, & \text{if } \theta^i > \pi \\ \theta, & \text{otherwise} \end{cases}
\end{aligned}
\tag{2.11}
$$

This method will give the correct interpolation of regions for transformations in 2d. However, in 3d, Euler angles are hard to interpolate with the same method. There is thus another technique that allows for the interpolation of rotations in both 2d and 3d in a similar manner. This technique uses *quaternions*, and is described in section [link to section].

TraversedArea

[Todo]

PointInside

[Todo]

Distance

[Todo]

Union / Diff?

[Todo]

2.3.3 Quaternions for interpolation of rotations

[say that no formal definitions or proofs are given and that a more complete definition/explanation of quaternions can be found somewhere else + refs (ex. see paper from mit in favorites)]

Section [link to section] describes a basic way to compute the interpolation of a rotation in 2d defined by a single angle. In 3d, however, the angles needed to describe a rotation are the Euler angles, and this method is not well-suited for these angles. [note + ref to gimbal lock] Another method for describing rotations in 2d and 3d are rotation matrices. [Compared to rotation matrices they are more compact, more numerically stable, and more efficient. - Wikipedia: Quaternions and spatial rotation → find source] Lastly, rotations in 3d (and 2d, 4d) can also be described using quaternions, and this technique allows for an easy interpolation method, which will be described later. Althought this master thesis mainly focusses on the moving regions in 2d (since most operations on moving objects are either really easy to generalise to 3d (ex: atInstant) or extremely hard (ex: TraversedArea), so only the 2d version is described [Maybe add this note earlier (intro or start of theory)]), since this technique using quaternions works for both 2d and 3d, we are adding this section to allow for an easy generalisation to 3d if needed. A small section is added in the end showing how complex numbers can be used in a similar way to interpolate 2d rotations by using less parameters than quaternions.

Representing 3d rotations using Quaternions

Quaternions were first described by William Rowan Hamilton in 1843 [citation?] and are an extention to complex numbers. Quaternions are represented in the following way.

$$q = a + bi + cj + dk, \text{ with } a, b, c, d \in \mathbb{R} \tag{2.12}$$

The symbols i, j and k can be interpreted as unit vectors in a three dimensional space, and are analoguous to the symbol i in complex numbers. Multiplication of complex numbers uses the fact that $i^2 = -1$. In a similar way, quaternion multiplication is defined the multiplication rules of i, j and k with each other.

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.13}$$

These equation above define the complete behaviour of the three symbols, and the equations below can be found using them.

$$\begin{aligned} ij &= k & ji &= -k \\ jk &= i & kj &= -i \\ ki &= j & ik &= -j \end{aligned}$$

Quaternion multiplication can then be done similarly to the multiplication of complex numbers by taking into account the multiplication rules for these three symbols. An important thing to note is that quaternion multiplication is not commutative, as can be seen when multiplying quaternions $q_1 = i$ and $q_2 = j$: $q_1 * q_2 = ij = k$

and $q_2 * q_1 = ji = -k = -q_1 * q_2$. A unit quaternion is a quaternion of unit length, meaning its norm is equal to 1.

$$\|q\| = \sqrt{a^2 + b^2 + c^2 + d^2} \tag{2.14}$$

Unit quaternions can be used to efficiently represent orientations or rotations in three dimensions. When a unit quaternion is used to represent a rotation, it is called a rotation quaternion. For example, a rotation of an angle θ around the axis $\vec{v} = (x, y, z)$ can be represented by the rotation quaternion:

$$q = \cos(\frac{\theta}{2}) + \frac{(xi + yj + zk)}{\sqrt{x^2 + y^2 + z^2}} * \sin(\frac{\theta}{2}) \tag{2.15}$$

Rotating a point/vector $p = (p_x, p_y, pz_)) = p_x i + p_y j + p_z k$ by θ around $\vec{v} = (x, y, z)$ involves evaluating the conjugation of p by q

$$p' = q * p * q^{-1} \tag{2.16}$$

, where q^{-1} is the reciprocal of q, which for unit quaternions is the same as the conjugate of q since q is already normalized.

$$\begin{aligned} q^{-1} &= \frac{q^*}{\|q\|^2} \\ &= \frac{\cos(\frac{\theta}{2}) - \frac{(xi+yj+zk)}{\sqrt{x^2+y^2+z^2}} * \sin(\frac{\theta}{2})}{\|q\|^2} \\ &= \cos(\frac{\theta}{2}) - \frac{(xi + yj + zk)}{\sqrt{x^2 + y^2 + z^2}} * \sin(\frac{\theta}{2}) \end{aligned} \tag{2.17}$$

When rotating a point or polygon around an axis that does not go through the origin, we first have to translate the point/polygon to make the rotation axis go through the origin. For example if the axis goes through the centroid of a polygon, we first have to translate this polygon so that its centroid and the origin are coincident. Then, we apply the rotation to each vertex of the polygon. Finally, the polygon is translated back to return its centroid at the inital position.

Multiple rotations can be combined together into a single one by combining the corresponding quaternions. For example, the combined rotation of first applying q_1 and then q_2 can be computed like this:

$$\begin{aligned} q &= q_2 * q_1 \\ p' &= q * p * q^{-1} \\ &= (q_2 * q_1) * p * (q_2 * q_1)^{-1} \\ &= (q_2 * q_1) * p * (q_1^{-1} * q_2^{-1}) \\ &= q_2 * (q_1 * p * q_1^{-1}) * q_2^{-1} \end{aligned} \tag{2.18}$$

, which indeed correspond to first applying q_1 to p and then applying q_2 to the previous result.

The efficiency of using quaternions instead of rotation matrices or axis-angle representations can be debated, but the main advantage of using these rotation

quaternions is that we can easily apply a spherical linear interpolation (*slerp*) to them. This is necessary to correctly interpolate between two 3d rotations, which is needed for the 3d version of the *interpolate* function described in section [link to section]. To use the resulting quaternion to effectively rotate a polygon can be done by either first transforming all polygon vertices to a quaternion representation and then doing quaternion multiplication, or by representing the resulting rotation quaternion in another way, for example using a rotation matrix, and then applying this rotation matrix to the polygon. Transformations from rotation quaternion to other rotation representations has been presented in [1], and the equations have also been added in Annex [TODO: write annex and add ref to annex].

Interpolating 2d and 3d rotations using Quaternions

Now that rotation quaternions have been presented as a method used to represent 3d rotations, let's look at how we can compute an interpolation between two rotations. An example of the interpolation of two 3d rotations can be seen below. [add example] The computations can be done relatively easily using quaternions, and corresponds to a spherical linear interpolation (*slerp*) in 3d.

The interpolation process can be represented mathematically as follows. To interpolate between two rotations represented by quaternions q_0 and q_1, we first compute an angle θ between them, and then apply the *slerp* process to find the interpolated rotation.

$$\begin{aligned} \cos(\theta) &= q_0[dot]q_1 \\ q(r) &= Slerp(q_0, q_1, r) \\ &= \frac{q_0 * \sin((1-r) * \theta) + q_1 * \sin(r * \theta)}{\sin(\theta)} \end{aligned} \tag{2.19}$$

Interpolating 2d rotations using complex numbers

We have shown how quaternions can be used to represent 3d rotations and to compute interpolation between 3d rotations. These techniques can thus also be used for 2d rotations, since they are a special type of 3d rotations and the interpolation between two 2d rotations also results in a 2d rotation. Nevertheless, using a quaternion rotation for 2d rotations might be overkill if 3d rotations do not need to be handled. A method has previously been described that handles only 2d rotations, but as said previously, it has to handle edge cases, because of the fact that rotation angles have to be between $-\pi$ and π. Another method uses complex numbers to, similarly to the previously mentioned quaternions, compute interpolation of 2d rotations in an efficient way. This is done using the exact same technique as with quaternions, but using less parameters. The equations are shown below, and the similarity with the previously defined equations for quaternions can clearly be seen.

A rotation of θ around the origin can be represented using the complex number $q = \cos(\frac{\theta}{2}) + i * \sin(\frac{\theta}{2})$ and a 2d point $p = (x, y)$ can be represented using the complex number $p = x + i * y$. Computing the interpolation between two rotations q_0 and q_1 can be done in a similar manner as quaternion interpolation.

$$\begin{aligned}
\cos(\frac{\theta}{2}) &= q_0[dot]q_1 \\
&= \cos(\frac{\theta_0}{2}) * \cos(\frac{\theta_1}{2}) + \sin(\frac{\theta_0}{2}) * \sin(\frac{\theta_1}{2}) \\
&= \cos(\frac{\theta_1 - \theta_0}{2}) \\
q(r) &= Slerp(q_0, q_1, r) \\
&= \frac{q_0 * \sin((1-r) * (\frac{\theta}{2})) + q_1 * \sin(r * (\frac{\theta}{2}))}{\sin((\frac{\theta}{2}))} \\
&= \cos((1-r) * \frac{\theta_0}{2} + r * \frac{\theta_1}{2}) + i * \sin((1-r) * \frac{\theta_0}{2} + r * \frac{\theta_1}{2})
\end{aligned} \tag{2.20}$$

and the same can be said for applying the rotation to an arbitrary point.

$$\begin{aligned}
p' &= q * p * q \\
&= q^2 * p \\
&= (\cos(\frac{\theta}{2})^2 + \sin(\frac{\theta}{2})^2 + i * 2\cos(\frac{\theta}{2})\sin(\frac{\theta}{2})) * p \\
&= (\cos(\theta) + i * \sin(\theta)) * (x + i * y) \\
&= (x * \cos(\theta) - y * \sin(\theta)) + i * (x * \sin(\theta) + y * \cos(\theta))
\end{aligned} \tag{2.21}$$

Chapter 3

Implementation

3.1 MobilityDB

[Intro on mobilityDB]

3.1.1 Representation of moving objects

[Explain the physical implementation, see MobilityDBDemoSSTD19 paper]

3.1.2 Use of PostGIS functions and types

Most (geometric) functions are not implemented again

3.2 Fixed-shape moving regions

The implementation of fixed-shape moving regions uses the same sequence representation of temporal types that already exists in MobilityDB. The newly defined type is called *tgeometry*, short for *temporal geometry*, but this might change in case other types of regions are also added (ex. deforming regions). As said previously, the PostGIS type *polygon* is used to represent a given region, to be able to use PostGIS as much as possible for geometric functions.

A tgeometry is thus defined by a sequence of input values, where each input value is a pair of both a time instant and a PostGIS polygon object. However, as said in the theory section, storing this sequence like that is not efficient at all, since we do not use that fact these regions have the same shape. To solve this, the representation using a sequence of transformations, described in the theory section, is used to store this sequence of temporal regions efficiently. The practical challenges of implementing moving regions are described in the following sections.

An important assumption on which most of the implemented algorithms rely is that the rotatin center of all moving regions is assumed to be their centroid. This means that when we want to store a transformation with respect to a region, we only need three values (in 2d) corresponding to the translation vector and the rotation angle. The centroid can be easily recomputed from the stored region, and we thus don't need to store it in each transformation vector. Another assumption that is made is the fact that the regions are made out of a large set of vertices, so that the storage space of a region is much larger than that of a transformation. If this

assumtion was not verified, for example if the regions handled were simple triangles, it might be possible to create more efficient algorithms for some operations, but this is not discussed in this master thesis.

3.2.1 Input and computation of transformation vector

The input of a moving region resembles the inputs of other moving objects in MobilityDB and depends on the duration of their temporal value. Four possible durations exist and the input for each of the is described below.

Instant

The input of a tgeometry of instant duration is just a pair of both a time instant and a region value. The moving region is then stored as such, since there is no redundancy when storing a single instant. No checks have to be made either, since the only requirement is that the region is of fixed shape, wich is obvious when we have a single instant. An object of this type is denoted *tgeometryinst*.

Instant Set

The input is an array of tgeometryinst and the resulting type is denoted *tgeometryi*. Here, the first thing to check is the fact that every instant must have a region of the same shape. For each instant after the first one, we compare the region value with the initial region stored in the first instant. To compare two regions, the transformation from the first to the second one is computed using the technique described in the theory section [ref to section]. Then the initial region is transformed using the previously computed transformation. Lastly every vertex of the polygon representing the region is compared to the corresponding vertex on the second polygon. If the position of two corresponding vertices differ too much, then the regions are considered of different shape and the input is rejected. If all corresponding vertices are sufficiently similar, then the regions are considered fixed-shape and the next instant is checked.

During these checks, the transformations of the regions of each instant with respect to the inital instant are computed. If all the checks pass, these transformations are kept and used to store the moving region in a more efficient manner. The first instant stores the initial region, while subsequent only store the needed transformations. This allows to store an instant set using much less space, while still being efficient to process. If we want to output the regions initially recieved, all that is needed is to use copy the inital region and apply the stored transformations to it.

[visualization of the difference]

Sequence

A moving region of sequence duration is essentially the same as an instant set, except that we assume that the position of the region can be computed at intermediate time steps using interpolation. This type of object is called a *tgeometryseq*. The input, checks and storage of a tgeometryseq is the same as for a tgeometryi. The only difference occurs during the normalization of the input and the computation of the bounding boxes. This comes from the fact that we assume that the region still exists

between two input values and all interpolated values need to be correct and inside the precomputed bounding box. These two particular cases are explained in further sections in more detail.

[visualization of the difference]

Sequence Set

Just like an instant set is an array of instants, a sequence set is an array of sequences. Since we assume that the input sequences have already passes the fixed-shape region checks, we only need to make sure that subsequent sequences also correspond to the same fixed-shape region. If this is the case for all sequences, then the input is accepted, else the input is rejected. For sequence sets, two possible storage ideas are possible, both being more efficient than the naive idea of storing all region values. The difference between both ideas is described, illustrated and discussed below. The arrows point to the region to which the transformation has to be applied to result in the correct output.

[visualization of both ideas]

The first idea is to store every sequence in the set as it is recieved, which means that every sequence will store an 'inital region' in their first instant, and transformations relative to this first region in their subsequent instants. This allows for an easy extraction of a sequence from the set, but requires more storage, since the same fixed-shape region is still stored multiple times.

The second idea is to keep only the first sequence as it is recieved, and to transform all subsequent sequences to only contain transformations relative to the 'initial region' (the region contained in the first instant of the first sequence). This alternative requires less storage space than the first, since every region except the very first one is replaced by a transformation. This storage idea also simplifies the normalization checks and the required modifications of the sequences during normalization. A downside of this approach is that it is computationally less efficient than the previous one when outoutting a sequence from the set or when splitting the set in multiple parts. An example of the required changes when splitting a sequence set in two is shown below. [visualize splitting of sequences]

While both alternatives are technically feasible, the first one was chosen because it was easier to implement and because it seemed to be the most computationally efficient solution. [practical test of the difference?]

3.2.2 Normalization of the representation of moving regions

Normalizing the representation of a moving object modifies the representation of the object to a unique representation. Every sequence or sequence set that represents the same moving object will thus have the same representation. This normalization is mostly used when creating a new moving object from an input or when joining multiple objects together (ex. when combining two sequences of the same region). During normalization, redundant instants that could be recomputed by interpolating between two other instants are removed. For instants and instant sets, there is no normalization process, since we do not assume any interpolation process between instants, and thus we cannot have redundant instants. A simple example with a moving point is shown below to visualize the concept of normalization.

[Example of normalization with a moving point]

Normalizing the representation of moving objects is important in multiple aspects. First, removing redundant instants in sequences and sequence sets makes the stored objects smaller. Secondly, having a unique representation for every moving objects also allows to compare them for equality much more easily than if multiple representations of the same moving object are allowed.

As said previously, normalizing happens only in sequences and sequence sets. Normalization in sequence set can be seen as joining two sequences when the last instant of the first sequence is identical to the first instant of the second one, and then normalizing the resulting sequence. In the following discussion, we will thus only focus on normalizing a single sequence.

To explain the normalization process of a moving region of sequence duration (tgeometryseq), we will first describe the normalization process of a moving point of sequence duration (tpointseq), since both processes are analogous. The interpolation technique used for moving points is a simple linear interpolation. In a sequence of points defined at increasing time instants (tpointseq), when three subsequent points are collinear and the ratio of their distance in the time dimension is the same as the ratio of their distance in 2d (or 3d) space, then the middle point is considered redundant, since it could easily be recomputed using interpolation. This situation can be seen in [ref to previous figure showing the normalization of a moving point].

Normalizing a moving region is done in a similar manner. The main difference here is that we do not check for collinearity of points, but we check for collinearity of transformation vectors. Three (2d) transformation vectors are collinear if their translation vectors are collinear in 2d space, and their rotation angles are also collinear with the same ratio as the translation vectors. To check collinearity of the rotation angles we have to handle edge cases, since the angles are always between $-\pi$ and π. An example of collinear regions can be seen below.

[Example of collinear regions]

When handling moving points, collinear points are considered redundant, since they can always be retrieved using interpolation. On the other hand, when handling moving regions, all collinear regions are not necessarely redundant. Indeed, since the interpolation method always interpolates two regions using the smallest angle between them, removing a collinear region could cause the interpolation to have a different direction of rotation than expected. This issue can be seen below.

[Figure showing errors when removing collinear regions with high rotations]

This issue only comes up when the sum of the angles between three subsequent transformation vectors is larger than π. If this sum of angles is smaller than π, the collinear region is redundant, and can be recomputed using interpolation. In this case the region can simply be removed. However, if this is not the case, then the collinear region is not completely redundant, since it gives an information about the direction of rotation that would not be present if the region was simply removed. Leaving the region present is also not a possiblity, since this could result in multiple moving regions being identical, while having a different representation.

[Show two possible representations with for the same moving objects]

To solve this issue multiple ideas are possible. A first idea is to add the direction of rotation in the transformation vector. Since the interpolation of regions is done between two adjacent instants, this extra value would be defined with respect to the previous instant in the sequence. This contradicts with the other values in the stored transformation vector, since previously the transformation vectors were define with

respect to the intial (first) region. This solution is thus feasible, but the values need to be handled with care.

Another solution is to add a *dummy* region that woud be the same for all objects representing the same moving region. This dummy region would serve as a kind of guide for the interpolation to preserve the correct direction of rotation. Altough this technique would not always reduce the storage space, it does take care of the unique representation of the moving regions, which is the main concern. It it however possible that in some cases a dummy region can take the place of two region instead of just one, and thus still reduce slightly the storage space. An example of this can be seen below.

[Example of dummy region replacing two 'redundant' regions]

This dummy region has to be well-defined, so that every possible input representing the same moving region could be transformed in a unique representation. The definition should also try to minimize the amount of dummy regions needed in this unique representation. We chose to define the dummy region/transformation [make sure the use of region vs transformation is either consistent, or well-explained at the start] in the following way.

When three collinear regions have a combined angle of larger than π, the middle region will be replaced by a region which has a rotation of $\pi - \epsilon$ with respect to the first region, where ϵ will be the same for all objects and has to be chosen so that the direction of rotation stay correct regardless of any rounding errors, while also being as small as possible. (probably take epsilon a bit larger, maybe 10x, than system error)

This (informal) definition produces a minimal amount of dummy regions, while still being able to recreate the initial movement of the region correctly. This method is also produces unique representations and is thus a good way to normalize moving regions.

Theoretically, since the transformation that is stored for this dummy region is partially redundant, we could replace it with a single value representing the direction of rotation, or the intermediate angle. This is in principle feasible to reduce storage space even more, but that would mean that we need to handle a third kind of object, which is maybe not the best idea when we want to implement clean code.

3.2.3 Rotating bounding boxes

Another implementation issue that arises when handling moving regions instead of points is the precomputation of the bounding boxes of the different moving region objects. In MobilityDB, bounding boxes for objects of instant set, sequence and sequence set durations are precomputed and stored with the object.

When computing the bounding box of a moving region of instant set duration, the same method as for the other moving objects can be applied, i.e.: the bounding box of each instant is computed (for regions, this is done using PostGIS geometric functions), and the resulting bounding box is just the smallest box containing all the bounding boxes of the instants. Since sequences of regions already have their bounding boxes precomputed, the same technique can be applied to moving regions of sequence set duration: the resulting bounding box is the smallest box containing all the bounding boxes of the sequences.

[Example of bounding box of instant set]

The last case that needs to be handled is the computation of the bounding box of a moving region of sequence duration. Applying the same method as for regions of instant set duration does not work anymore, since the vertices of the regions do not move linearly when the region is rotating. This means that some vertices will go outside the smallest bounding box containing both the start and end regions. A trivial example is when a square rotates 90 degrees without translating. The bounding box containing its start and end vertices will just be the square itself, but the corners of the square clearly exited this bounding box during the rotation.

A more advanced example can be seen below.

[Example of error when computing the naive bounding box]

To solve this issue, we imagined two possible alternatives. First of all, if the traversed area (see 2.3.2) of the moving is known, we can just take the bounding box of this area, which will be indeed the minimal bounding box containing the moving region completely. However, since computing this traversed area is a computationally heavy operation, we propose another method that computed a bounding box in a more efficient way. This solution produces a correct but non-optimal bounding box. The bounding box is correct, because the region will never exit this box during its movement, but non-optimal, because it will usually not be the smalles bounding box that can be found. These assumptions are still sufficient for the bounding box to be used for indexing and/or sorting purposes, so this solution has been implemented in MobilityDB.

To compute this bounding box for sequences, a similar procedure as for instance sets is used. The only difference is that when computing the bounding boxes of the instants, a *rotating bounding box* algorithm is used instead of the PostGIS bounding box functions. This rotating bounding box algorithm works as follows:

1. Find the vertex furthest from the rotation center. In our case the rotation center is the centroid of the polygon representing the region. This step can be done with a linear seach through the vertices.

2. Compute the distance d_{max} between the rotation center and this vertex.

3. Create a square of side $2 * d_{max}$, centered around the rotation center.

4. Return this square as the rotating bounding box.

5. (optional) Also return d_{max} to be used for the following instants, since this value does not change if the region and the rotation center of the region stay the same. If this is done, we can skip step 1 and 2 when computing the next rotating bounding boxes.

[Maybe visualization of the algorithm]

This algorithm will return a bounding box which completely contains the given region, independently of any rotation applied to it on its rotation center. After computing the bounding boxes of all the instants, the resulting bounding box will simply be the smallest box containing all the bounding boxes of the instants.

This solution was used to compute bounding boxes of moving regions of sequence duration before the traversed area algorithm was implemented.

3.3 Operations

Implementation of the operations

3.3.1 AtInstant

Get transformation at instant (compute it using interpolation if needed) Transform initial region using this transformation

3.3.2 Interpolation

2D

¡Tx, Ty¿, Theta Compute directly, while taking care of edge cases. Or using quaternions just like in 3d

3D

¡Tx, Ty, Tz¿, rotation matrix? Linear interpolation for the translation Quaternions for the rotation

3.3.3 Traversed Area

Traversed area

3.4 Discussion

- Measurements of moving regions? → maybe input as transformation could be useful

3.5 Stepwise interpolation of deforming moving regions

Explain the implementation requirements (stepwise interpolation of floats are also needed, implementation needs to be complete with respect to all operations) Practical use for cadastre (maybe in use case, idk where to put this section)

Chapter 4

Use case

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Chapter 5

Conclusion and future work

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Appendix A

Appendix Title

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Bibliography

[1] *Geographic information – Schema for moving features*. Standard. Geneva, CH: International Organization for Standardization, June 2008.

[2] Ralf Güting et al. "A Foundation for Representing and Querying Moving Objects". In: *ACM Transactions on Database Systems (TODS)* 25 (Mar. 2000), pp. 1–42. DOI: 10.1145/352958.352963.

[3] Florian Heinz and Ralf Hartmut Güting. "A data model for moving regions of fixed shape in databases". In: *International Journal of Geographical Information Science* 32.9 (2018), pp. 1737–1769. DOI: 10.1080/13658816.2018.1458103. URL: https://doi.org/10.1080/13658816.2018.1458103.

[4] Florian Heinz and Ralf Hartmut Güting. "A polyhedra-based model for moving regions in databases". In: *International Journal of Geographical Information Science* 34.1 (2020), pp. 41–73. DOI: 10.1080/13658816.2019.1616090. URL: https://doi.org/10.1080/13658816.2019.1616090.

[5] Esteban Zimányi et al. "MobilityDB: A Mainstream Moving Object Database System". In: *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*. SSTD '19. Vienna, Austria: Association for Computing Machinery, 2019, 206–209. ISBN: 9781450362801. DOI: 10.1145/3340964.3340991. URL: https://doi.org/10.1145/3340964.3340991.

[Check copyright for ISO reference!]