In *Proc. of Intl. Conf. on Discrete Global Grids*, Santa Barbara, CA, March 2000.

Augmenting SAND with a Spherical Data Model*
(EXTENDED ABSTRACT)

Houman Alborzi and Hanan Samet
Department of Computer Science
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20740
{houman,hjs}@cs.umd.edu

November 30, 2014

Abstract

SAND is a spatial database and information browser system developed at University of Maryland which supports 2D and 3D data models. The experience in extending SAND to include a spherical data model thereby enabling SAND to perform spatial queries on spherical data and to browse their results in an incremental manner is described. The focus is on the issues that arise in adding a spherical data model to a database that has been built on the basis of the planar data model. Special emphasis is placed on how the data structure was chosen for the task. The various geometrical primitives that were needed for modeling spherical data as well as the algorithms that were required in the implementation are also described. As SAND is based on the quadtree representation, a natural method for adding spherical data to SAND was to find a spherical adaptation of the quadtree. The sphere was mapped into a quadtree representation by projecting the sphere onto a cube, and then constructing the quadtree on the faces of cube. Alternatively, it is also possible to map the sphere into a plane, and then simply use a planar quadtree. Some of the differences between these approaches are discussed. In addition, the display models and visual query mechanisms used in SAND with our new spherical data model to ease the task of navigating through the data are described. By adding a spherical sector primitive, it is possible for users to locate data on the sphere which are located in a spherical lune anchored at two antipodal spherical points.

*The support of the National Science Foundation under Grants EIA-99-00268 and IRI-97-12715 is gratefully acknowledged.

1 Introduction

SAND [ES96] is an interactive spatial database and browser developed at University of Maryland. SAND combines a graphical user interface with a spatial and non-spatial database engine. It supports queries on spatial and non-spatial data, such as spatial selections and spatial joins. SAND supports a number of different spatial data structures, including the PMR quadtree [NS86] and the R*-tree [BKSS90]. In this paper we focus on the PMR quadtree which is a variant of the region quadtree (e.g., [Sam90]) that can handle spatial objects of arbitrary dimensionality (i.e., including 2D and 3D). For example, in two dimensions, the PMR quadtree subdivides the underlying rectangular space r into four congruent rectangular areas whenever the number of objects that overlap r exceeds a predefined value s, termed the splitting threshold. Each of the resulting areas contains references via pointers to the spatial descriptions of the objects that overlap them. The PMR quadtree is different from other bucketing methods in that when the number of objects that overlap r exceeds the splitting threshold, then r is only subdivided once even though some of the resulting areas, say a, may still be overlapped by more than s objects. The key is that a will be subdivided the next time an object is inserted that overlaps it. In this way, regions are not subdivided many times when more than s objects are very close to each other.

Spatial selections in SAND are in the form of finding all data objects, where the value of one of their spatial attributes overlaps a particular region in space. Of particular interest are range queries where SAND enables a user to find data objects whose distance to another data object is within a given range. For example, this feature enables a user to find all warehouses that are between 100 and 200 miles of a particular retail store. Another query feature of SAND allows the user to find all the objects which have a certain orientation with respect to another data object. For example, a user can locate all warehouses which are north of a given location.

SAND also supports the join operation. There are many variants of this operation whose result is a set whose elements are the members of the Cartesian product of the tuples of two relations that satisfy the join condition. When the join condition is based on the values of the spatial attributes, the operation is known as a *spatial join*. The join condition often involves tuples that are co-located or within a given distance of each other. Another variant of the join is the *distance join* [HS98] in which case the resulting tuples are ordered according to the spatial proximity of the objects associated with the joined tuples. The distance semi-join [HS98] is a special case of the distance join in which case each item of one of the joined sets is paired up with the closest member (in terms of the spatial attribute values of the two sets) of the other of the joined sets. The resulting tuples are ordered according to the distance between their constituent spatial attributes. For example, consider

the case that one data set contains the locations of the warehouses of a merchant, and another dataset contains the locations of retail stores of the merchant. Using the distance semi-join, a user can find the closest warehouse for each retail store. SAND performs this kind of spatial join by utilizing an algorithm that finds the nearest neighbors of spatial objects in an incremental manner [HS99].

In this paper we report on an effort to design and implement a spherical data model for SAND. The goal was to give SAND the ability to perform correct queries for data on the surface of the Earth. The original SAND implementation was based on a planar model of space, and hence was not able to perform correct computations for distances of data objects on the surface of the Earth. In particular, the planar model only provided reasonably accurate responses to a small portion of the Earth. The main shortcoming of SAND was that the distance function did not take into account the curvature of the Earth.

The rest of this paper is organized as follows. Section 2 discusses the different considerations that were taken into account in choosing the spatial data structure to support spherical data. Section 3 presents the spatial objects that are needed in our spherical data model. Section 4 describes the algorithms needed to deal with spherical geometry in our implementation. Section 5 indicates the limited number of changes that needed to be made to the SAND Browser to enable viewing spherical data. Section 6 contains concluding remarks and directions for future research.

2 Spatial Data Structure to Support Spherical Data

In this section, we describe the different approaches that we attempted in order to extend SAND to support spherical data. Before our extension, SAND supported polygons, lines, and points on a plane. We enhanced SAND to support their counterparts on the surface of a sphere — that is, spherical polygons, spherical lines, and points on a sphere. Given that SAND already contains a large volume of software to support spatial data structures for 2D space, we focussed on the different ways in which a sphere could be mapped to a 2D space (plane). In the following discussion, we use the term *data space* to describe the space in which the data resides, and *grid space* for the space in which the data structure manipulations take place. For example, if the sphere is mapped onto a plane p, and a quadtree decomposition is subsequently developed on the plane, then p is the 'grid space'. Obviously, there should exist a mapping between the data space and the grid space. In particular, an appropriate mapping permits efficient operations on the data.

There are two ways to implement the mapping. One approach is to map the data objects directly onto the grid space whenever they are modified or inserted

into the database. An alternative approach is to map the grid space into the data space whenever a decision should be made about where an object should reside in the partition of the data space induced by the grid. In this case, we are mapping the partition lines of the grid space into the data space. In the first approach, all of the data must be mapped from the data space onto the grid space, whereas in the second approach only the grid partitions must be mapped from the grid space onto the data space. Assuming that there is more data than partition lines, it appears that mapping the grid space onto the data space is cheaper from a computational complexity standpoint than mapping the data space onto the grid space.

Mapping the grid space onto the data space is made more efficient by storing the result of the mappings of the grid space onto the data space in the actual data structure. For example, in the case of a quadtree-like subdivision in the grid space, we can maintain the result of mapping the partition points from the grid space onto the data space in the data structure. In addition, we should bear in mind that even for a database with a few insertions or modifications, mapping data objects onto the grid space may not be computationally feasible. For example, in the case of mapping a sphere onto a plane, a spherical line may not necessarily be mapped into a line on the plane. Hence, performing computations on the result of the mappings is not a straightforward task. Similar problems can be encountered when designing a mapping from the grid space onto the data space. For example, suppose again that the grid space is the plane which is subdivided into triangle blocks, and we use the Lambert equal-area cylindrical projection [Sny87] as described in Section 2. In this case, the mapping of non-vertical non-horizontal edges of grid triangles, are not simple arcs on the sphere. The mappings of partitions of the grid should make use of simple geometrical primitives where the required geometrical algorithms — distance and intersection — are easy to implement.

In the case of the spherical data model, we used the second approach where we do not map the data objects onto the grid space. Instead, we map the data structure grids onto the data space, and perform the geometrical algorithms in the data space. This idea is frequently used in applications involving spatial data structures. For example, in the case of the PM quadtree representation of vector objects (i.e., non-raster objects) [SW85], although the underlying space is decomposed into blocks, the objects are never decomposed into the subobjects that pass through each block. In particular, in each block b that overlaps object o, we store a pointer to o instead of clipping o to b [NS86]. In this way we need not have to worry about whether or not, for example, two connected line segments in adjacent blocks are part of the same line. Moreover, we can safely remove and add parts of the same line without concerns about roundoff errors resulting from the clipping process [FvDFH90]. This solution is also used in overcoming the null object detection problem in solid modeling applications (e.g., [Til84]).

We investigated three different mappings between the data space and the grid space. The first two were based on embedding a cube in the sphere and projecting locations on the cube to the sphere, or vice versa as they are equivalent since the mappings are 1-1 and onto. The third was based on an equal area cylindrical projection of the plane onto the sphere (also known as Lambert's cylindrical equal area projection [Sny87]). The first two mappings were based on the ideas proposed by Scott [Sco96].

In the first mapping that we tried, once we mapped the sphere onto the cube, we modeled each face of the resulting cube with a quadtree data structure thereby resulting in 6 quadtrees. This mapping has the property that each line on the cube is a mapping of a spherical line on the sphere (i.e., a great circle arc). Therefore, we only needed to implement the geometrical algorithms dealing with spherical lines. However, if one wants to model parallels to the Equator, then the resulting projection onto the cube will not be a line anymore. It should noted that while Scott's idea of projecting onto the cube is novel, his method [Sco96] of calculating mapping between the subdivisions of cubical faces and their spherical counterparts is wrong. In particular, by using a parallel projection, some parts of the sphere are not covered on the cube. The problem can be solved by projecting through the center of the sphere.

Another interesting feature of this mapping is that spherical polygons are mapped into polygons on the cube, and by keeping the cubic projection of each data object we can use the faster planar geometric algorithms instead of the spherical ones. However, a drawback of this mapping is that it is not an equal area projection. This means that the projections on the grid space of uniformly distributed data in the data space are not uniformly distributed. Although implementing this approach seemed straightforward initially, we encountered considerable difficulties when we tried to modify many parts of SAND to incorporate it. For example, in the case of a general spatial join, we would have to perform 36 pairwise intersections — one for every possible pair of faces of the two cubes that correspond to the two joined sets. However, in many of the standard functions in SAND there is an implicit assumption that the data is stored in a single quadtree. Finding and debugging all the related code seemed impractical for this task.

Observing the infeasibility of using six quadtrees for any dataset, we employed an alternative approach where we flattened the cubic faces on a plane. In other words, the grid space was considered to be a single rectangle which contains all resultant six faces of result of projecting the sphere onto the cube (see Figure 1). This approach allowed us to reuse many of the SAND routines with no extra effort. However, the main drawback of this approach was that some of the regions in the grid space did not have a counterpart (i.e., were undefined) in the data space. Thus some of the algorithms in SAND failed to work properly without further

modification. In particular, not every connected region in the grid space had a corresponding region on the sphere. This was a problem because some of the operations in SAND examined every block spanned by the region in the grid space and some of these blocks were not well-defined on the sphere, and hence difficult to deal with. The dotted rectangle in Figure 1 shows such a block.

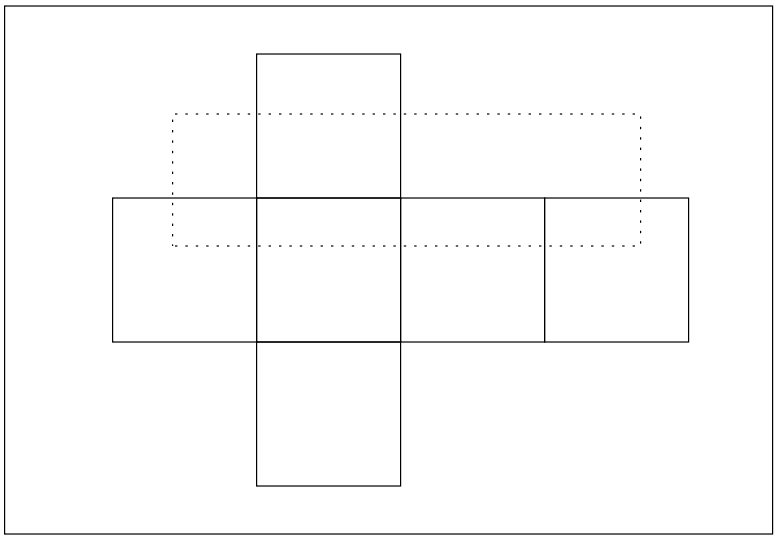

Figure 1: Flattening a cube on the plane.

Based on experiments with the first two approaches, we concluded that an appropriate mapping for SAND should have the following properties:

1. Map a sphere into a single rectangle,

2. Any rectangle on the plane should map to a simple shape on sphere.

We used Lambert's cylindrical equal area projection as the mapping. This mapping is also an equal area projection and hence preserves the uniformity of data points. However, it has singularities at the poles, where the poles will be mapped into lines in grid space. A side effect is that data primitives around the poles will be elongated in the projection.

In the sections that follow, we describe more details about spherical data primitives and geometrical algorithms needed for the implementation.

3 Spatial Objects in the Spherical Data Model

This section introduces the spatial objects in our spherical data model. The spherical objects include spherical points, spherical lines, spherical polygons, and Lambert rectangles. In the next section, we describe the geometrical algorithms on these objects.

3.1 Preliminaries and Notations

All objects in our spherical data model reside on the surface of a *sphere* of radius 1 (i.e., the unit sphere). The center of the sphere serves as the center of all of the coordinate systems that we use. Let O denote the center of the sphere and let S^2 denote the surface of the sphere. While all the objects reside in S^2, it is convenient to also use three-dimensional Euclidean space $\mathbb{R}^3$ whenever needed. We use either a Cartesian coordinate system, or a spherical coordinate system, or both to specify the coordinates of objects. The triple (x, y, z) and the pair (λ, ϕ) denote a point in the two coordinate systems, respectively. λ is also known as the *longitude* of the point, and ϕ is known at its *latitude*.

For any point P on S^2 there is exactly one corresponding point and vector in $\mathbb{R}^3$ both of which are denoted by P. $A \times^n B$ denotes the normalized cross product of two vectors A and B.

The distance between any two points A and B can be either defined in $\mathbb{R}^3$, which is the length of the shortest line connecting them in three-dimensional space, or in S^2, in which case it is the length of the shortest arc on the sphere that connects them. The former is denoted by $d_3(A, B)$, while the latter is denoted by $d_S(A, B)$.

For any point P with coordinate values (x, y, z), we define its *antipodal* point $\overline{P}$ as the point with coordinate values of $(-x, -y, -z)$. A point, its antipodal, and O are collinear. Furthermore, the distance from a point P to O equals the distance from P's antipodal point $\overline{P}$ to O, or formally $d_3(O, P) = d_3(O, \overline{P})$.

The intersection of a sphere with a plane forms a circle. If the plane passes through the center of the sphere, then the intersection is called a *great circle*, and it has the same radius as the sphere. Any other circle is termed a *small circle*.

Any three non-collinear points in $\mathbb{R}^3$ specify one and only one plane passing through them. Hence, the center of the sphere and any two non-antipodal points on the sphere specify exactly one plane and exactly one great circle of the sphere.

Any two points A and B and the origin specify a plane passing through them such that the plane's normal is $A \times^n B$. The angle between two normal vectors A and B is $\arccos(A \cdot B)$.

Spherical data is any kind of spatial data on the surface of a sphere. The basic unit of data is a *spherical point*. A *spherical line* is the collection of all points that

lie on the sphere on the shortest path between two spherical points that are termed its two endpoints. Notice that if the two endpoints of a spherical line are antipodals, then there are many spherical lines defined by them.

A *spherical polygon* is a list of spherical points where the edges of the polygon are the spherical lines between adjacent elements (i.e., spherical points) of the list.

3.2 Spherical Point

A spherical point is a single point on the surface of the unit sphere centered at the origin. Given a spherical point P with Cartesian coordinate values (x, y, z), and spherical coordinate values (λ, ϕ), the following relationships hold:

$$\lambda = \arctan(y, x) \tag{1}$$

$$\phi = \arcsin(z) \tag{2}$$

$$x = \cos(\lambda)\cos(\phi) \tag{3}$$

$$y = \sin(\lambda)\cos(\phi) \tag{4}$$

$$z = \sin(\phi) \tag{5}$$

$$1 = x^2 + y^2 + z^2 \tag{6}$$

Recall that λ is also known as the longitude of the spherical point, and ϕ is known at its latitude.

3.3 Spherical Line

Any two non-antipodal spherical points specify a unique spherical line which is defined as the shortest path on the surface of the sphere connecting the two spherical points. The great circle formed by two non-antipodal spherical points is divided into two arcs where the spherical line is the shorter arc. The length of this arc is defined to be the length of the spherical line. Figure 2 shows two spherical points A and B and the depicted circle is the great circle passing through them and O. The length of the arc shown with a bold line is $2\arcsin(d/2)$, or more formally:

$$d_S(A, B) = 2\arcsin\left(\frac{d_3(A, B)}{2}\right). \tag{7}$$

of

3.4 Spherical Polygon

A spherical polygon is a closed region on the sphere bounded by non-intersecting spherical lines. We represent a spherical polygon by a list of spherical points ordered in such a way that two adjacent spherical points in the list specify a spherical

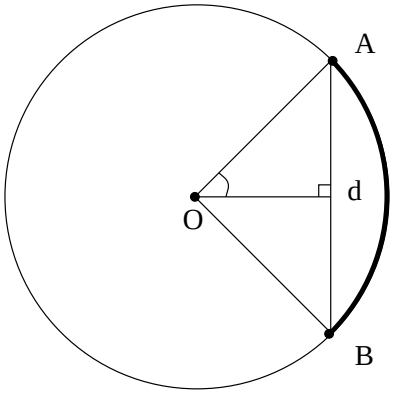

Figure 2: Length of a spherical line between spherical points A and B

line (edge) bounding the spherical polygon. Figure 3 shows an example spherical triangle.

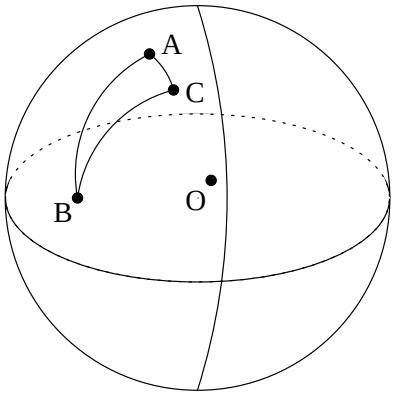

Figure 3: Example spherical triangle

The angle between two intersecting spherical lines is defined as the angle between the tangents of the great circles of the spherical lines that pass through the intersection point. If the intersection point of two adjacent spherical line segments of a spherical polygon is denoted by B and the other endpoints of the spherical line segments are denoted by A and C, then the angle at vertex B of the spherical polygon is equal to the angle between the planes containing the great circles of the spherical lines, which is:

$$\pi - \arccos((A \times^n B) \cdot (B \times^n C)). \tag{8}$$

The area of a spherical polygon is related to the sum of its angles as follows. Assume the spherical polygon has n vertices $v_1 \ldots v_n$, and the angle at vertex v_i is denoted by α_i, then

$$Area = \sum_{i=1}^{n} \alpha_i - (n-2)\pi. \tag{9}$$

A spherical polygon divides the sphere into two areas. We assume that the smaller of these two areas is inside the polygon. Considering that the area of the unit sphere is 4π, the area of a spherical polygon is always less than 2π.

3.5 Lambert Rectangle

When indexing 2D data objects in SAND, the underlying space is subdivided into rectangles. Rectangular subdivisions have two desirable properties. First, it is relatively easy to test for inclusion of a point in them. Second, it is easy to subdivide them into smaller rectangles. Re-examining equation 9 in Section 3.4, we see that a four-sided spherical polygon with four right angles has an area of 0. In other words, a spherical rectangle with four sides covers just a single point of sphere. Therefore, a right-angled quadrilateral cannot be defined nontrivially on a sphere which means that we needed another form of a rectangle for our spherical data model. We define such an entity below which we term a *Lambert rectangle*.

A *Lambert rectangle* is a collection of spherical points with their longitude and latitude values in a given range $((\lambda_1, \lambda_2), (\phi_1, \phi_2))$. The area of such a rectangle is $(\lambda_2 - \lambda_1)(\sin\phi_2 - \sin\phi_1)$ [Wei98]. Remembering that for any spherical point, its z-coordinate value is equal to the sin of its latitude ($z = \sin\phi$), and also the fact that $\sin(.)$ is a monotonically increasing function from $-\pi/2$ to $\pi/2$, we can specify the range with $((\lambda_1, \lambda_2), (z_1, z_2))$ instead. Hence, The area of such a Lambert rectangle is $(\lambda_2 - \lambda_1)(z_2 - z_1)$. Figure 4 is an example of a Lambert rectangle.

One of the benefits of using Lambert rectangles is that we can specify the whole sphere with a single Lambert rectangle with longitudinal range of $(-\pi, \pi)$ and latitudinal range of $(-\pi/2, \pi/2)$. A Lambert rectangle is also easily divisible into smaller Lambert rectangles. Another advantageous property is that Lambert rectangles are the natural choices for a spherical quadtree as subdividing a Lambert rectangle into four equal area smaller rectangles can be easily done by using the center of the rectangle $((\lambda_1 + \lambda_2)/2, (z_1 + z_2)/2)$. Assuming that the objects on a sphere are uniformly distributed, the Lambert rectangles provide the same performance for quadtree-based data structures as in the planar case (i.e., in 2D).

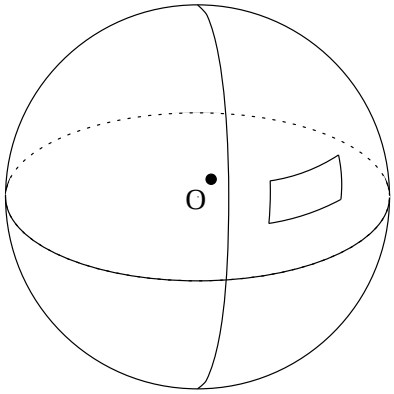

Figure 4: Example Lambert rectangle.

In particular, we can see that if the latitude values range between 0 and 80 degrees, then a subdivision at $((\lambda_1 + \lambda_2)/2, (\phi_1 + \phi_2)/2)$ results in four Lambert rectangles which do not have equal area (see Figure 5 where the subdivision is at $\phi = 40$ degrees), while a subdivision at $((\lambda_1 + \lambda_2)/2, (z_1 + z_2)/2)$ does result in four Lambert rectangles of equal area (see Figure 5 where the subdivision is at $z = sin(\phi) = sin(29.50) = .4924$).

4 Spherical Geometry Algorithms

In order to implement the spherical data model in SAND, we had add algorithms for determining if two spatial objects intersect and for calculating the distance between two spatial objects. In particular, for any pair of object types in our spherical data model, we had to implement the distance and intersection functions between them. In this section, we describe some of the algorithms used in the implementation of the spherical data model in SAND.

4.1 Distance between Two Spherical Points

The distance between two spherical points is the length of a spherical line having them as is endpoints.

4.2 Intersection of Two Spherical Points

Two spherical points intersect if and only if they have the same coordinate values.

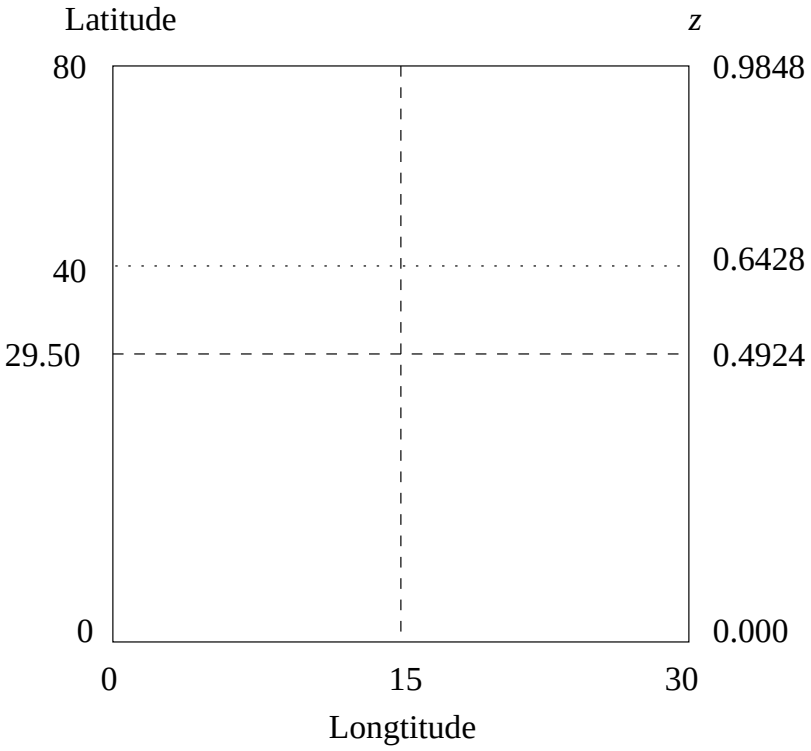

Figure 5: Example demonstrating the subdivision of a Lambert rectangle.

4.3 Distance between a Spherical Point and a Spherical Line

The distance from a spherical point p to a spherical line l is the distance from p to one of the endpoints (A, B) of l (see Figure 6a), or some point q on l. q has the property that it is colinear with the line joining the center O of the sphere and the projection C of p on the plane containing l. If m is the normal vector of the plane containing the spherical line l ($m = A \times^n B$), then $C = p - (p \cdot m)m$. We also need to check if q lies on the spherical line l It is easy to see that q lies on the spherical line l (i.e., the shorter arc between A and B) if and only if angle Aqb in triangle AqB is obtuse (see Figure 6b). Based on these considerations, in order to find the distance between a spherical point p and a spherical line l with endpoints A and B, function `dPointLine`, on the sphere (i.e., d_S) from p to either A, B, or q.

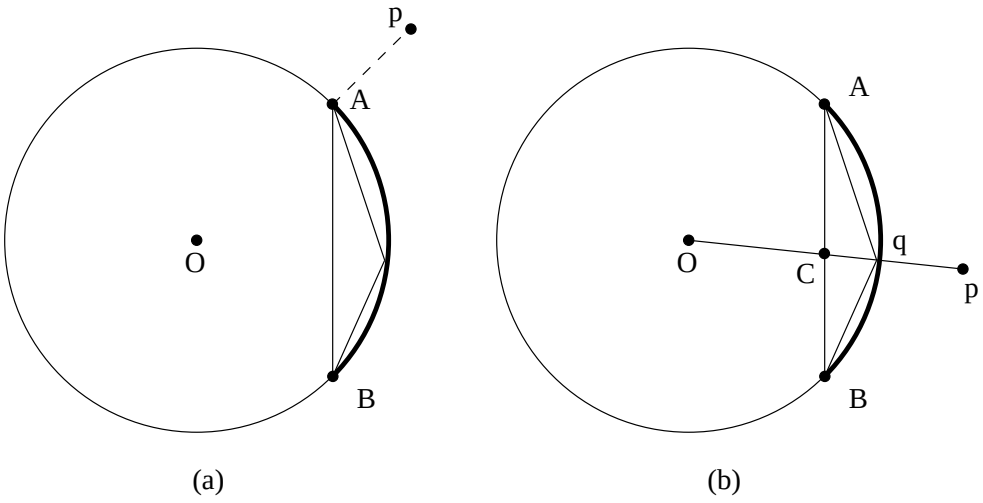

Figure 6: Example of the computation of the distance from a spherical point to a spherical line.

```
1 function dPointLine(p, l : line<p1, p2>)
2     n  =  p1 ×^n p2
3     c  =  p − (p · n)n
4     q  =  c/|c|
5     if (pointInArc(q, p1, p2))
6         return dS(p, q)
7     else
```

```
8       return  min(d_S(p, p_1), d_S(p, p_2))
```

Function`dPointLine` makes use of the function `pointInArc` to determine if point q, which lies on the great circle of a spherical line l with endpoints of p_1 and p_2, is on l. As we pointed out above, this is only true if the angle p_1qp_2 is obtuse. This check is simple to make in the sense that angle p_1qp_2 is 90 degrees if the sum S of the squares of the lengths of the two edges that comprise it is equal to the square of the length of the edge p_1p_2 denoted by H. The angle is acute (obtuse) if C is less (greater) than H.

```
1 function pointInArc(q, p_1, p_2)
2    return  (d_3^2(q, p_1) + d_3^2(q, p_2) < d_3^2(p_1, p_2))
```

4.4 Distance between a Spherical Point and a Spherical Polygon

In order to to find the distance between a spherical point and a spherical polygon, we need to consider two cases: either the spherical point is on the polygon or not. In the first case, the distance is simply zero. In the second case, the point is not on the polygon and the distance is the minimum of all distances from the point to the edges of the polygon.

Function `dPointSphere`, given below, achieves this test.

```
1 function dPointSphere(p, g : polygon<l_1, l_2, ..., l_n>)
2    if intersects(p, g)
3       return  0
4    else
5       return  min_i d_S(p, l_i)
```

4.5 Intersection of two Spherical Lines

Two spherical lines l_1 and l_2 intersect if and only if one of the two intersection points of their corresponding great circles lies on both l_1 and l_2. Function `intersectLines`, given below, achieves this test.

section.

```
1 function intersectLines(l_1 : line<p_1, p_2>, l_2 : line<p_1, p_2>)
```

```
2 <x_1, x_2> = intersectionPoints(l_1.plane, l_2.plane)
3   for j = 1 to 2
4     if pointInArc(x_i, l_1.p_1, l_1.p_2, ) and pointInArc(x_i, l_2.p_1, l_2.p_2)
5       return True
6   end for
7   return False
```

Function `intersectLines` makes use of the function `intersectionPoints` to determine two points on the sphere which correspond to the endpoints of the spherical line formed by the intersection of two planes p and q of two great circles.

```
1 function intersectionPoints(p : plane<n>, q : plane<n>)
2   r = p.n ×^n q.n
3   return <r, -r>
```

5 Additions to the SAND Browser

The SAND Browser, which is the graphical user interface (GUI) of SAND, uses a two-dimensional display system for displaying the data. A user utilizes the GUI for performing queries. Incorporating the spherical data type into the SAND Browser was conceptually very simple. The main modification to the SAND Browser's GUI was the addition of the ability to render spherical lines. In the current implementation, a spherical line is approximated by many short line segments on the display. We used a heuristic to decide how many segments are needed for a good approximation of the spherical line. The heuristic that we devised uses the latitude of the two endpoints and the length of line. If the endpoints are far from poles or the line is long, then the heuristic uses more line segments.

An additional feature of the SAND Browser is the spatial selection operation which enables a user to select data items that are located in a sector. A sector is represented by a point and two rays emanating from that point. To support the sector on a sphere, we use a spherical lune [Wei98], which allows the user to select the spherical data that is located on a spherical lune. In order to specify the lune, the user selects one endpoint p of the lune, and two spherical lines having p and the antipodal of p as their endpoints. Notice that only p need be specified (i.e., the antipodal of p need not be specified by the user).

Since there is an infinite number of spherical lines between p and the antipodal of p, the user is specifying which of the spherical lines serve to demarcate the lune.

6 Concluding Remarks and Directions for Future Research

circle. The performance of the geometric primitives for spherical polygons was improved by making use of the notion of a minimum bounding box. When determining whether a point is inside a polygon, for example, we can quickly dismiss any points outside the minimum bounding box, without examining any of the vertices of the polygon. In addition, minimum bounding boxes enable the use of a filter-and-refine query processing strategy. In particular, the minimum bounding box serves to yield a set of candidate objects (the filter step) while the full description of the objects is used as a refinement step to produce the final answer to the query. In SAND, we store a minimum bounding rectangle with each 2D polygon. In the implementation of the spherical data model in SAND we store a minimum bounding Lambert rectangle with each spherical polygon. The same idea is also be applied to spherical lines. However, the minimum bounding Lambert rectangle is not so simple to compute as in the planar case because the range of latitude and longitude values that span a spherical line l cannot in general be obtained by simply examining the latitude and longitude values of l's spherical endpoints. Rather, one must consider the entire trajectory of l.

Future work involves the incorporation of additional primitives into the spherical model of SAND. Examples include great circles and any arc of them, small circles and any arc of them, and spherical polygons that cover more than half of the sphere. Also, the ability to perform spherical visualization is a desirable feature (i.e., visualizing on a spherical surface rather than on the projection of the sphere on the plane).

The current implementation of the spherical sector query in SAND is somewhat restricted in the sense that a sector can only span half of the sphere. This restriction is a result of the definition of a spherical sector to be a lune which is represented using a spherical polygon. Recall that spherical polygons can cover at most half of the sphere in the current implementation. Thus, permitting more general spherical polygons (i.e., with surface area greater than 2π) will immediately allow spherical sectors covering more than half of the sphere. In fact, complete generality can be obtained by permitting the user to select both endpoints of the spherical sector, instead of implicitly anchoring the sector at the other end by using the antipodal of the user-specified anchor point as the second endpoint of the spherical sector.

The definition of a spherical line segment is also restrictive in the sense that the extent of a spherical line segment is always less than 180 degrees. One possible way of allowing longer spherical line segments would be to specify the extent of the arc traversed by the spherical line. We can define an entity which we term a *spherical arc* as any arc of a great circle. A simple specification of a spherical arc can be achieved by giving its great circle and two endpoints. However, the two

endpoints divide a circle into two arcs and thus we need additional information to distinguish between them. We note that a great circle is specified by the normal vector of the plane that contains it. By using the direction of this normal vector, we can specify a spherical arc unambiguously. We say that the spherical arc is *counterclockwise* when we orient the great circle so that its normal vector is facing outwards.

Similarly, the definition of a spherical polygon is also overly restrictive in the sense that the area of the polygon cannot exceed one half of the area of the sphere. Moreover, the length of any edge of the spherical polygon is bounded, in the same way as the length of a spherical line segment. The polygon area restriction could be easily lifted by specifying a point that is located in the spherical polygon. Alternatively, we can use an anchor point p on the sphere (such as the North Pole), and associate a flag with each spherical polygon s that indicates whether or not s contains p. Longer polygon edges can be allowed by using the spherical arc definition given above.

It is also worthwhile to compare the performance of our representation in SAND of the spherical data with other representations. Recall that some of our decisions were heavily influenced by the current implementation of 2D data types in SAND. Some other techniques could involve making use of triangular spherical elements for the grid space as would be the case when an octahedron [Dut84, Dut90, GY92, OZ93] or an icosahedron [FD84, Fek90], as well as square spherical elements for the case that a cube [Sco96], are embedded in the sphere instead of the mapping of a plane onto the sphere that we used.

Acknowledgments

We want to thank Gisli R. Hjaltason for his invaluable assistance in helping to incorporate our techniques into the SAND system.

References

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[Dut84] G. Dutton. Geodesic modelling of planetary relief. *Cartographica*, 21(2&3):188–207, Summer & Autumn 1984.

[Dut90] G. Dutton. Locational properties of quaternary triangular meshes. In *Proceedings of the Fourth International Symposium on Spatial Data Handling*, pages 901–910, Zurich, Switzerland, July 1990.

[ES96] C. Esperanca and H. Samet. Spatial database programming using SAND. In M.J. Kraak and M. Molenaar, editors, *Proceedings of the Seventh International Symposium on Spatial Data Handling*, pages A29–A42, Delft, The Netherlands, August 1996.

[FD84] G. Fekete and L. S. Davis. Property spheres: A new representation for 3-d object recognition. In *Proceedings of the Workshop on Computer Vision: Representation and Control*, pages 192–201, Annapolis, MD, April 1984. (Also University of Maryland Computer Science TR-1355).

[Fek90] G. Fekete. Rendering and managing spherical data with sphere quadtrees. In *Proceedings of Visualization '90*, pages 176–186, San Francisco, CA, October 1990.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

[GY92] M. F. Goodchild and S. Yang. A hierarchical spatial data structure for global geographic information systems. *CVGIP: Graphical Models and Image Understanding*, 54(1):31–44, January 1992.

[HS98] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[HS99] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999.

[NS86] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, August 1986.

[OZ93] E. J. Otoo and H. Zhu. Indexing on spherical surfaces using semi-quadcodes. In D. Abel and B. C. Ooi, editors, *Advances in Spatial Databases — Third International Symposium, SSD'93*, pages 510–529, Singapore, June 1993. (Also Springer Verlag Lecture Notes in Computer Science 692).

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[Sco96] G. M. Scott. The cubic quadtree: a spatial data structure for spherical surfaces. scholarly paper CSC 1024, University of Maryland, College Park, MD, December 1996.

[Sny87] J. P. Snyder. *Map projections – a working manual*. U.S. geological survey professional paper 1395. United States Government Printing Office, Washington, DC, 1987.

[SW85] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, July 1985. (Also *Proceedings of Computer Vision and Pattern Recognition 83*, Washington, DC, June 1983, 127–132; and University of Maryland Computer Science TR–1372).

[Til84] R. B. Tilove. A null–object detection algorithm for constructive solid geometry. *Communications of the ACM*, 27(7):684–694, July 1984.

[Wei98] E. W. Weisstein. *The CRC Concise Encyclopedia of Mathematics*. CRC Press, Boca Raton, FL, 1998.