Designing and Creating Buffers of Spherical Polygons

Michael Tan

April 29, 2015

Abstract

We design and code an algorithm that could buffer spherical polygons. A spherical polygon is a set of points on the unit sphere that represents an enclosed area. The function buffer(polygon P, angle A) returns a new polygon P' that approximately contains all the points within P, and any other points X on the surface of the unit sphere such that angle POX is at most A, where O is the origin. We designed a version of the algorithm that works in all cases, and programmed a version of it that works for most convex, concave and nested-loops (polygons with 'holes') polygons. Our algorithm is roughly as follows: Given a polygon P, trace along the edges of the polygon. For each edge, create the buffer of the edge that will occur in the new polygon by using coordinate transformations. Then link the edges together, adding in additional points to approximate the curved circular area that will occur between consecutive edges that meet at convex angles. When linking edges together we always trace along the edges in a counterclockwise direction, leading to some edges that may even occur inside the polygon, when handling concave angles. To remove such edges we apply an algorithm that detects self-intersections and removes the points. To handle nested loops polygons we apply a version of the buffer algorithm that shrinks instead of expands the polygon, and we combine it with the original polygon. The algorithm assumes that no polygon combined with its buffer can encompass the whole unit sphere, and that no inner-loop of a polygon is smaller than the buffer itself.

1 Introduction

We worked with Google’s open source Java Spherical Geometry library, located at code.google.com/p/s2-geometry-library-java/. The spherical geometry library is primarily used to do calculations on the earth’s surface, in coordination with GPS or latitude-longitude data. The library makes the assumption that the earth is the unit sphere, and contains a suite of functions for calculating and testing relations between different points, lines and shapes.

Of particular interest for the current project are:

- S2Points
- S2Loops
• S2Polygons

These are data representations in the S2 library of points, loops, and polygons. The ‘S2’ in the title of these data representations stands for ‘spherical geometry in two dimensions’.

1.1 S2Points

S2Points are the building blocks of all shapes in the S2 Geometry Library.

The documentation of the S2Point reads: "An S2Point represents a point on the unit sphere as a 3D vector. Usually points are normalized to be unit length, but some methods do not require this."

There are three feasible representations of an S2Point data structure: a latitude-longitude pair of coordinates, an x-y-z coordinate triple, or a (theta, alpha) angle pair representing polar angle, and angle from equator. The S2 geometry library chooses to represent points in an x-y-z coordinate system, as it is suitable for most calculations done in the library. However, calculating the buffer of a spherical polygon oftentimes requires the (theta, alpha) coordinate system, so we introduced conversions between the two systems in our code.

1.2 S2Loops

S2Loops are ordered lists of S2Points that represent shapes on the Earth’s surface.

The documentation reads: “An S2Loop represents a simple spherical polygon. It consists of a single chain of vertices where the first vertex is implicitly connected to the last. All loops are defined to have a CCW orientation, i.e. the interior of the polygon is on the left side of the edges. This implies that a clockwise loop enclosing a small area is interpreted to be a CCW loop enclosing a very large area.

Loops are not allowed to have any duplicate vertices (whether adjacent or not), and non-adjacent edges are not allowed to intersect. Loops must have at least 3 vertices...”

1.3 S2Polygons

An S2Polygon consists of at least one S2Loop, and can represent a more complex region on the earth’s surface than an S2Loop. Particularly, it can represent regions that have “holes” in them. A point is considered to be contained within an S2Polygon if it is contained by an odd number of loops. Polygons’ loops may not cross or share edges, although loops may share vertices.

Our primary task is to “enlarge” S2Polygons by a fixed amount, resulting in another S2Polygon.

2 The Task of Polygon Buffering

Given a set of points on the earth’s surface representing an S2Polygon, one may desire to create a larger version of the S2Polygon. The function Buffer(S2Polygon P, Angle A)
takes in an S2Polygon and returns an S2Polygon \( P' \) with the property that \( P' \) contains all the points of \( P \), as well as any points \( X \) on the unit sphere such that angle \( \angle QOX \) is at most \( A \), where \( Q \) is any point within \( P \), \( O \) is the origin (coordinates \((0, 0, 0)\)).

2.1 Buffers of Convex Polygons

It is not difficult to imagine what the buffer \( P' \) of a convex polygon \( P \) and angle \( \theta \) would look like. Given the edge set of \( P = e_1, e_2, \ldots, e_n \), we can see that the buffer will contain edges \( f_1, f_2, \ldots, f_n \), where each \( f_i \) is an angle \( \theta \) away from each \( e_i \). However, although the \( e_i \) are consecutive in \( P \), it does not follow that the \( f_i \) are consecutive. In fact, the \( f_i \) are never consecutive for convex polygons. Between consecutive \( f_i \), there will be arcs along the unit sphere that represent points that are within \( \theta \) of individual vertices of the original polygon. (Analogously, consider a square in two dimensions. Its “buffer” would contain four line segments as well as four semicircular arcs. A similar thing is happening with spherical polygons.)

2.2 Buffers of Concave Polygons

Buffers of concave polygons do not preserve the simple property that one can find edges \( f_i \) of the buffer given edges \( e_i \) of the original polygon. This is most easily illustrated using a two-dimensional example. Consider a shape with \((x,y)\) coordinates \((0,0)\), \((2,2)\), \((4,0)\), \((2,4)\). It sort of looks like an arrow’s tip. If we want to buffer this shape with a size of 0.3, we find that the buffers of the two lower edges in fact overlap, and we have to take their point of intersection and remove parts of the edges that are obtained. The buffers of the upper edges act similarly to the convex polygon case, in that they are easily calculated from the original edges and are separated by circular arcs. Any algorithm in the 3-dimensional spherical geometry case must have a way of handling concave edges.

2.3 Buffers of Nested Polygons

Polygons containing holes in them require special casing in that the algorithm applied to the outer loop cannot simply be repeated in the inner loops. Consider a doughnut shape in two dimensions containing two circles. The buffer of such a polygon contains the enlarged outer loop, as well as a “shrunk” inner loop, since the buffer grows inward. The algorithms for shrinking polygons are similar in nature to expanding them, except that certain constants must be changed.

3 An Algorithm for Buffering Spherical Polygons

This is the main result of the paper. My algorithm for buffering spherical polygons contains four major steps:

1. retrieving buffers of individual edges
2. chaining buffers of edges by adding circular arcs between them
3. removing interior intersections formed in the edge chain (happens only with concave polygons)

4. adding in shrunk version of inner loops of the polygon, in case the polygon contains holes

The algorithm assumes that: no spherical polygon combined with its buffer contains the entire unit sphere; that no inner loop of a nested polygon is smaller than the buffer region; and that there are all inner loops of a nested region are disjoint. These will be true of most data point calculations on the earth’s surface. An exception to this would be data point sets containing landmasses with islands inside lakes. If the boundaries of land regions are designated with individual S2Loops, one could easily have an island nested within a lake nested within a larger land mass, resulting in a triply-nested S2Loop. We believe most data point calculations that need to invoke the nested characteristic of S2Polygons will use at most one nesting level of depth. Consider, for example, a data set containing a forest. Although the forest may have “holes” in them (one level of depth), it seems unlikely that the forest would contain a clearing that itself had a mini forest within it (two levels of depth).

3.1 ALG1 - Retrieving Buffers of Individual Edges

Given an edge AB and a buffer angle \( \theta \), suppose its buffer is edge CD. If again we denote O to be the origin, we must have that angle COA and DOB are equal to \( \theta \). Furthermore, let \( t \) be any real number such that \( 0 \leq t \leq 1 \). If R is \( t \) of the way from A to B, and S is \( t \) of the way from C to D, then angle \( ROS \) must also be equal to \( \theta \). We compute the points C and D in eight steps. Roughly speaking, we use a change of coordinate matrix to from the XYZ system to a new X’Y’Z’ system, in order to calculate where C and D will be in the new system. Then we apply the inverse of the change of coordinates to find the coordinates of C and D in the normal XYZ system.

1. Generate a transformation matrix U and its inverse, such that A is the new unit vector in the X direction. It is a well known result that if \( S = v_1, v_2, ..., v_n \) for a vector space V and \( v \) is a vector in V, that the coordinates of \( v \) using the basis vectors in \( S \) is equal to the matrix \( U^{-1}v \), where \( U \) is the matrix whose column vectors are \( v_1, v_2, ..., v_n \).

2. Map point B to point B’ using the transformation matrix U. B’ represents what the coordinates of B would be in the new coordinate system with A as the unit vector in the X’ direction. This simply involves multiplying the matrix \( U^{-1} \) by the 3x1 matrix representing the point B.

3. Find the counterclockwise angle of B from the Y’ axis, and call it \( \alpha \). This can be computed as the arctangent of the quotient \( B_z'/B_y' \).

4. Compute the counterclockwise angle \( \beta \) of C’ from axis Y’, where C’ is the coordinates of the point C in the new coordinate system. (and C is the buffer of the point A.) This is computed as \( \beta = \alpha - \pi/2 \).
5. Compute the X’ coordinate of C’. This is equal to the cosine of the buffer angle \( \theta \). To see why this is the case, note that in the new coordinate system, A is the new unit vector in the X’ direction. Any point that is an angle \( \theta \) from A will have the same X’ coordinate, since the locus of points \( \theta \) from A on the unit sphere is a circular region on the sphere centered at A, with constant X’ coordinate.

6. Compute the Y’ and Z’ coordinates of C’. \( C'_y \) is first computed as the cosine of \( \beta \), and \( C'_z \) is first computed as the sine of \( \beta \). However, these are scaled versions of what they should be, since we want to maintain the invariant that \( C'_x^2 + C'_y^2 + C'_z^2 = 1 \). So we must multiply our \( C'_y \) and \( C'_z \) by the sine of \( \theta \).

7. Apply \( U \) to to C’ to get the coordinates of C in the XYZ system.

8. Repeat steps 1-7 with B as the new unit vector in the X’ direction, in order to obtain the point D that is the buffer of B. However, in steps 4 we must use the formula \( \beta = \alpha + \pi/2 \) instead of \( \beta = \alpha - \pi/2 \), since the latter formula would place D on the wrong side of edge AB.

3.2 ALG2 - Chaining Buffers of Edges

This algorithm describes the process of taking the individual buffers of edges and concatenating them to form a list of edges that will be simplified into the final polygon.

Suppose that we have edges MP and QR being consecutive edge buffers of the original polygon. Then we must generate points \( p_1, p_2, ..., p_k \) between P and Q that approximate a circular arc around P and Q. Suppose that P and Q are both buffers of the vertex Z of the original polygon. The circular arc will have Z as its center.

1. Generate a transformation matrix \( U \) and its inverse, such that ‘pivot’ is the unit vector in the new X’ direction. This is similar to step 1 of the previous algorithm.

2. Map P and Q to P’ and Q’ using the transformation matrix \( U^{-1} \). Note that P’ and Q’ will have identical X’ coordinates, since they are both created at the angle \( \theta \) from Z.

3. Compute \( \alpha_p \) and \( \alpha_q \), the CCW angles of P’ and Q’ from axis Y’. Similar to step 3 from previous algorithm.

4. At every 10 degree interval from P’ to Q’ we create a vertex R’, map it to XYZ coordinate values, and add it in between P and Q in the edge chain. This is similar to the creation of C’ in the previous algorithm. Specifically, we place points at angles \( \alpha_p + 10, \alpha_p + 20, ... \) until we reach an interval that exceeds the angle \( \alpha_q \).

In step 4, the constant of 10 was chosen to maintain a faithful approximation of the original polygon without creating too many points that would bloat the size of polygons in memory. A version of the code could easily take in an angle \( \gamma \) as a constant to use instead of 10. Alternatively, we could have the user specify a distance \( d \) such that any two consecutive points (except the last pair) on the circular arc are \( d \) apart.
3.3 ALG 3 - Removing Interior Intersections in the Edge Chain

The edge chain formed by the above algorithm is exactly the result needed for all convex polygons whose buffer will not intersect itself (which happens if the polygon and buffer combination is large enough to stretch around the whole earth, and touch itself there. This seems unlikely to be a necessary calculation when dealing with data sets.) However, the algorithm produces a result for concave polygons that must be corrected.

Consider the arrow example mentioned previously, in 2D with the coordinates (0,0), (2, 2), (4, 0), and (2, 4). If we applied the 2D analogous of ALG1 and ALG2 to this arrow with a buffer size of 0.5, we would find that the buffers of the (0,0) - (2,2) edge and (2,2) and (4,0) edges intersect, and that when chaining the buffers of these edges, we add points along a circular arc that travels inside the original polygon. Clearly such edges must be deleted. In order to correct for this, we detect intersections that occur along the raw edge chain and eliminate interior edges resulting from them. The algorithm assumes that vertex 0 of the original polygon has a convex angle; this is the responsibility of the caller of the function.

1. Let $i < j$ and edges consisting of vertex pairs $(i, i+1)$ and $(j, j+1)$ intersect in the edge chain, where $i$ is the minimal index of any edge that intersects other edges, and $j$ is minimal given $i$.

2. Get the point of intersection between edges $(i, i+1)$ and $(j, j+1)$. This is handled by an already written function in the S2 Java geometry library.

3. Remove vertices $(i+1, i+2, ..., j)$ from the list of vertices in the edge chain.

4. Add the intersection point in between vertex $i$ and the new vertex $i+1$.

5. If the edge chain $C$ is altered to produce $C'$, re-run the algorithm with edge chain $C'$ so that further intersections can be discovered and deleted.

3.4 ALG 4 - Creating Buffers of Hole Regions

The algorithm for creating the buffer that shrinks an S2Loop rather than expands it is analogous to ALGS 1-3, although some constants must be changed. We also assume that vertex 0 of any inner loop is a convex angle; this is the responsibility of the caller of the function.

Specifically, for any loop L nested within the outer loop, we repeat ALGS 1 through 3 with using $-\pi/2$ to create vertex C in ALG1, and $\pi/2$ to create D in ALG1, as the angles that we must add to $\alpha$. In ALG2 we still add points in a CCW fashion at vertices 10 degrees apart for each other. (ALG2 is in fact unchanged when creating shrunk versions of holes.) In ALG3, instead of deleting vertices $(i + 1, i + 2, ..., j)$, we delete vertices $(1, 2, ..., i)$ and $(j + 1, j + 2, ..., n - 1)$, where $n$ is the number of vertices in the inner loop. When we rerun the algorithm, we re-run it with the original ALG3 which deletes vertices between $(i + 1)$ and $j$, inclusive.
4 Runtime Analysis

ALG1 runs in time $O(n)$, where $n$ is the number of vertices in the loop. This is because each step of the loop runs in $O(1)$ time. We can save time by saving the transformation matrices, since the $U$ for creating the buffer of edge AB will be used again when creating the buffer of edge BC, but this is a premature optimization. If any step needs to be optimized it is probably ALG3, for reasons discussed later.

ALG2 also runs in time $O(n)$, where $n$ is the number of vertices in the loop. Its runtime complexity is also dependent on what the constant angle between different vertices on the circular arc is. However, the steps to compute each point on the circular arc are relatively fast.

ALG3 runs in time at least $O(n^2)$, since we have to test whether each edge intersects another edge. However, it is not clear what the true runtime complexity of ALG3 is, because we don’t know how many interior intersections the algorithm will find. If there are at most $g(n)$ interior intersections formed by the raw edge chain from ALGs 1 and 2 for a polygon of $n$ vertices, then the runtime complexity is bounded above by $O(g(n)n^2)$. This is why the constant chosen in ALG2 should not be too small (we chose 10 degrees). If the constant were say, 1 degree, it would slow down ALG3 by a lot. Circular arcs consisting of 8 points (a 90 degree angle in the original polygon) would now contain 89 points, and the $O(n^2)$ nature of the algorithm would increase runtime significantly.

ALG4 is a concatenation of ALGs 1 through 3, with changing some constants in a way that will not affect runtime complexity. So ALG4 is also $O(g(n)n^2)$.

5 Sample Test Case from S2PolygonBufferTest.java

5.1 A concave polygon

The vertices of the original polygon are created from the Latitude-Longitude coordinates 4:2, 0:0, 2:2, 0:4. The S2Point representations of these points are:

1. (0.99696, 0.03481, 0.069756)
2. (1.0, 0.0, 0.0)
3. (0.9987820, 0.034878, 0.034899)
4. (0.997564, 0.069756, 0.0)

The final points obtained (using a constant of 30 degrees instead of 10 in ALG2, to shorten the final list) are:

1. (0.9969184001502955, 0.027005054637494214, 0.07365073296174032)
2. (0.9999619230641713, -0.007808103016236176, 0.003896915330389402)
3. (0.9999619230641713, -0.00871047323262026, -5.292238356037835E-4)
The first column contains x coordinates and can largely be ignored; what is important is that they are all close to one. Observe the y and z coordinates, starting at points 8-10. Point 8 is at the bottom left of the arrow. At point 9, the y coordinate increases sharply and the z coordinate increases sharply. At point 10, the y coordinate increases sharply again but the coordinate drops back to what it was at in point 8. This reflects how the buffer of the convex angle in this polygon consists of the intersection of the buffer’s edges.
6 Running the Code

I have attached a zip file containing the S2 Java Geometry library, which includes my files S2PolygonBuffer.java and S2PolygonBufferTest.java. These files are located in src/com/google/common/geometry/ . Sample tests of creating edge chains, as well as buffers for simple, convex, and nested polygons can be found in that file.

To compile the code, make sure that /lib/ and /src/com/google/common/geometry/ are in the classpath. Then go to src/com/google/common and run

javac geometry/*.java

To run the test file, go to the /src directory and run

java com/google/common/geometry/S2PolygonBufferTest

One can also add print statements to see the points produced by the algorithm.