School of Computing
Advanced Undergraduate Project

Visualization of Attraction Trajectories in the Beacon Model

Brandon Bloch

supervised by

Dr. David Rappaport and Dr. Bahram Kouhestani

April 2017

Abstract

The beacon model introduced by Biro et al. [1] is an idealization of geographical greedy routing in communication networks. Given a point object and beacon in a polygonal network, the beacon attraction trajectory describes the movement of the object directly towards the beacon, subject to collisions with the polygon's boundary.

In this paper, an algorithm is presented to determine the attraction trajectory from a point object to a beacon in a simple polygon. To implement such an algorithm, a number of relevant sub-problems are first introduced and dealt with. Next, an application is presented to visualize and manipulate instances of the beacon model in the context of a graphical user interface. The algorithm is then used by the application to animate the point object's attraction towards the beacon. The application animates the motion of the object towards the beacon over time as calculated by this algorithm. Enforcement of the beacon model's constraints during user interactions with the GUI and manipulation of an instance of the model is discussed. Finally, a number of examples and noteworthy edge cases are provided to demonstrate the success of the application and underlying algorithm.

Acknowledgements

If I have seen further, it is by standing on the shoulders of giants.
Isaac Newton

I would like to first sincerely thank my project supervisor, Dr. David Rappaport. Thank you for your guidance along the path to completion of this project; for never failing to be excited by a new idea and leave my gears turning after a meeting; and for ensuring that I would strive to make this project something I am truly proud of.

I would also like to give deep thanks to Dr. Bahram Kouhestani. Your prior work on the subject at hand provided a strong foundation upon which this project could be built. Your committed mentorship and your enthusiasm for what you do are truly inspiring. I am profoundly grateful to have had the opportunity to work with you and learn from you.

Thank you to Melissa Mangos, Robert Wood, Bahram Kouhestani, and David Rappaport for your insightful help and thoughtful editing.

Thank you to my teachers in the School of Computing at Queen's University, as well as my many classmates over the years. You were instrumental in the journey of discovery and improvement that was my undergraduate experience.

Thank you to my friends, who keep me both focused and distracted.

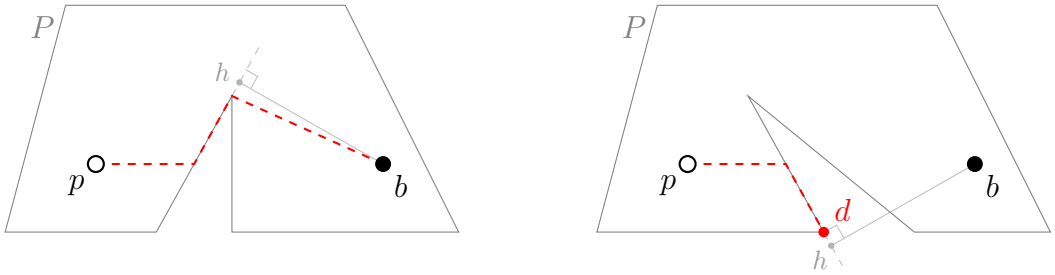
Finally, thank you to my family—you have always been there for me. Without your constant love, support, and encouragement, I would never have made it here.

Brandon

Background

The beacon model was introduced by Biro et al. [1] as a continuous analog to the problem of geographical greedy routing in sensor networks. Each node in such a network is assigned a coordinate, and greedy routing is performed by passing messages along to the neighbouring node with the smallest Euclidean (straight-line) distance from the destination. A message has arrived at its destination when this distance reaches zero. As the density of nodes in the network increases, the path produced by greedy routing increasingly follows a straight line to the destination. In cases where the message encounters the network boundary, the path continues by sliding along the boundary if the distance towards the destination decreases.

The network boundary is represented by a simple polygon P. A beacon $b \in P$ is a point that, when activated, produces an attractive pull towards itself. The beacon represents a message destination. When b is activated, it will attract a point object $p \in P$ until the distance between p and b cannot decrease further. The object represents a message being greedily routed to its destination. The path taken by the object, illustrated in Figure 1 as a dashed line, is called its *beacon attraction trajectory*.



(a) Successful routing from p to b

(b) Failure to route from p to b

Figure 1: Illustrations of point attraction in the beacon model

This greedy approach does not always succeed in routing a message to its destination. Failure occurs when the object reaches a point d on the boundary of the polygon where it can no longer decrease its distance from the beacon. Such a point d, called a *dead point* with respect to b, is a non-zero local minimum of the distance to b. Thus, a beacon attraction trajectory terminates when the object reaches either the beacon (as shown in Figure 1a) or a dead point (as shown in Figure 1b).

Although the concept is similar to that of gravitational or electromagnetic attraction, the beacon model is not a full-fledged model of the motion of physical bodies. Specifically, effects of mass such as acceleration do not occur; the massless, sizeless point object moves at a constant speed in its attraction towards the similar point beacon. This reflects the behaviour of the communication networks being idealized rather than a physical system.

The beacon model has been further characterized through work by Kouhestani et al. [2]. They have shown that when a point object p is attracted to a beacon b inside a simple polygon, the length of its trajectory is at most $\sqrt{2}$ times the length of the shortest path between p and b that is contained in P. Assuming the beacon is reached, this ensures that the length of the path taken by greedy routing will be no more than a constant multiple of the shortest possible path length, making the use of greedy routing feasible in real-world situations where faster calculation is important. However, Kouhestani et al. additionally show that in polygons with holes, this ratio can be unbounded.

Each segment of the attraction trajectory path can be classified as one of two types. When the object is pulled directly towards the beacon through the interior of the polygon, a *pull edge* results. If the object instead slides along the polygon boundary, pulled indirectly towards the beacon, a *slide edge* results. In both examples given in Figure 1, the first edge of the attraction trajectory is a pull edge and the second is a slide edge. Behaviour along the boundary is dictated by the orthogonal projection of the beacon onto the current boundary segment—the point on the line containing the current segment that is closest to the beacon, illustrated as point h in Figure 1. If the segment contains this point, a slide edge and the attraction trajectory end there, having reached a dead point. If the orthogonal projection lies beyond the edge, the slide edge will move to the end of the segment closer to the beacon, continuing until the object reaches a dead point or the beacon.

The geometric concepts of the beacon model are basic, but it becomes more difficult to attain a good intuition of the model's behaviour given increasingly complicated polygons. A system for automating the calculation of beacon attraction trajectories given combinations of polygons, beacons, and objects may be useful in real-world situations that can be reasonably modelled as continuous or near-continuous beacon networks. Furthermore, a tool to aid in the visualization of attraction trajectories for such combinations may be helpful when studying the characteristics of a network or its configurations. For instance, the tool could aid in the optimization and improvement of a network based on locations of dead points with respect to certain key beacons of interest. Such a calculation and visualization tool would likely be of particular interest to those studying the beacon model in general and looking to improve their intuition.

Approach

The application presented in this paper visualizes and enables the manipulation of simple polygons (i.e. those with no self-intersections) that do not contain holes. A beacon and point object are positioned within the polygon and the attraction trajectory is calculated in real-time as GUI interactions occur. The trajectory can be drawn instantaneously, or the position of the object can be animated over time through playback functionality.

Encoding Polygons, Point Objects, and Beacons

A polygon is encoded by its (x, y) vertex coordinates listed in counterclockwise order. The choice of starting vertex is arbitrary, but the polygon is always considered closed (an edge joins the last vertex to the first). Therefore, a polygon with n vertices also has n edges.

Polygons are two-dimensional. With a zero-dimensional polygon, only one point exists in the polygon (the single vertex). Since the object can only be at the beacon, the problem is uninterestingly trivial. With one-dimensional polygons (line segments), assuming the object is not at the beacon, the attraction trajectory always consists of a single pull edge straight to the beacon. This path will always be optimal, and greedy routing in such shapes is similarly uninteresting. Two-dimensional space is the lowest dimension that introduces non-trivial cases, and thus two-dimensional shapes are the focus of the application.

Edges are encoded as ordered pairs containing their start and end vertices. Given the vertex encoding of a polygon, the list of edges can be found in $O(n)$ time (where n is the number of vertices) by Algorithm 1. Because the vertices are stored in counterclockwise order, the edges are similarly given in counterclockwise order.

$edges \leftarrow$ empty list
for i **in** $0..|vertices|$ **do**
  $edges_i \leftarrow (vertices_i, vertices_{(i+1) \bmod |vertices|})$
end
return *edges*

Algorithm 1: Get the edges of a polygon from its vertices

General-Purpose Polygon Functions

A number of underlying functions are needed to enforce the beacon model's constraints in the context of GUI manipulation, as well as to calculate the beacon attraction trajectory.

Test for Self-Intersection

To allow manipulation of polygon vertices, the application must ensure that the user may not cross one edge over another. Algorithm 2 determines self-intersection in $O(n^2)$ time, which is sufficient for the relatively low values of n that the application encounters.

```
for i in 0..|edges| do
    for j in (i + 1)..|edges| do
        if edges_i neighbour of edges_j then continue
        if edges_i intersects edges_j then return True
    end
end
return False
```

Algorithm 2: Determine if a polygon intersects itself

Test for Containment of a Point

The ability to reposition both the beacon and point object arbitrarily dictates the need for a function that determines whether the polygon contains a given (x, y) point.

Since the polygon contains no self-intersections and no holes, it has exactly one "inside" region and one "outside". A ray drawn from (x, y) in any straight direction will intersect the polygon a finite number of times. Each time it crosses an intersection point, the ray alternates from being inside the polygon to being outside, and vice versa. If the ray does not intersect the polygon at all, the point is outside the polygon. Thus, if an even number of intersections is counted, the point is outside; if an odd number is counted, the point is inside. This approach also considers points on the polygon boundary as inside, rather than outside.

Algorithm 3 produces a solution in $O(n)$ time based on the notion of alternating between inside and outside. It uses the fact that the application's canvas begins at $(0, 0)$ as a guarantee that $(-100, -100)$ is far enough to safely end the ray; this can be handled in general simply by determining the bounding box of the polygon (also in $O(n)$ time) and choosing a point just outside it. This same approach can be extended to polygons with holes by treating a hole as a nested polygon—if a point is inside a hole, then it is outside the polygon.

$extreme \leftarrow Point(-100, -100)$
$ray \leftarrow Edge(p, extreme)$
$inside \leftarrow False$
foreach $edge \in edges$ **do**
  if ray intersects $edge$ **then** $inside \leftarrow \neg inside$
end
return $inside$

Algorithm 3: Determine if a polygon contains a point p

GUI Manipulation

In the application, polygon vertices can be repositioned by left-clicking and dragging, or removed by right-clicking. Edges can be left-clicked to split them into two edges, connected by a new vertex created under the cursor. Vertices cannot be moved outside the canvas, or so that the object self-intersects. They also cannot be moved sufficiently close to an edge to create a "near-closed" polygon, where the polygon is segregated into two distinct areas. Finally, the object and beacon can be repositioned so long as they remain inside the polygon.

The object and beacon are registered for a set of event handlers that determine when they are clicked and dragged. When moving either point, the new cursor position is tested for containment within the polygon—if the point is contained, the new position is stored and a redraw of the canvas occurs. If not, the new position is ignored. Because of this, invalid movements produce no change in the scene until the polygon is re-entered.

In addition to the outline of the polygon, transparent *vertex handles* are created. These larger, clickable objects enable interaction with the vertices, either to remove them by right-clicking or to reposition them by dragging. The latter mechanism matches that of the object

and beacon, except in the constraint violation checks performed before committing the new vertex position (as discussed previously). Each edge is overlayed with a transparent *edge handle*, much thicker than the edge and slightly shortened at each end. These regions can be clicked to produce a new vertex, and the altered shape helps users interact with edge handles while minimizing interference with vertex handles.

With the few operations available, the application can represent any simple polygon through combinations of clicking, dragging, and right-clicking. It can also represent any combination of positions for a single object and beacon within such polygons.

Calculating the Beacon Attraction Trajectory

The attraction trajectory is calculated in a recursive manner given a polygon, beacon, and point object. A greedy algorithm is given in Figure 2 that either produces a pull or slide edge or returns a final result based on the current position. This algorithm is then used by the application.

In Figure 2, the current position is denoted by p_{now}, the beacon position by b, and a dead point by d. $\rightarrow_{pull}$ and $\rightarrow_{slide}$ represent a pull or slide edge to the specified position.

In the base case, where the current position is at the beacon, the algorithm simply returns with success. If anywhere else, the algorithm checks whether the current position is on the boundary or interior to the polygon.

If interior to the polygon, a pull edge will always be produced. A function is used to determine the intersection points of a line segment with the polygon, sorted in ascending order by their distance from the segment's start. A line segment is drawn from the current position to the beacon and the aforementioned function is called. If no intersections occur, there is a direct path from the current position to the beacon; the algorithm pulls to the beacon and recurses. Otherwise, the algorithm pulls to the intersection point that is nearest to the current position and recurses.

If on the boundary, the same question is asked: excluding the current position as an intersection point, is there a straight path to the beacon that does not intersect the polygon? If so, the algorithm will similarly pull to the beacon and recurse. Otherwise, the next question is whether a pull edge between the current position and the nearest intersection point would leave and re-enter the polygon, or remain interior. If it remains interior to the polygon, any point along it will be contained in the polygon—its midpoint is selected

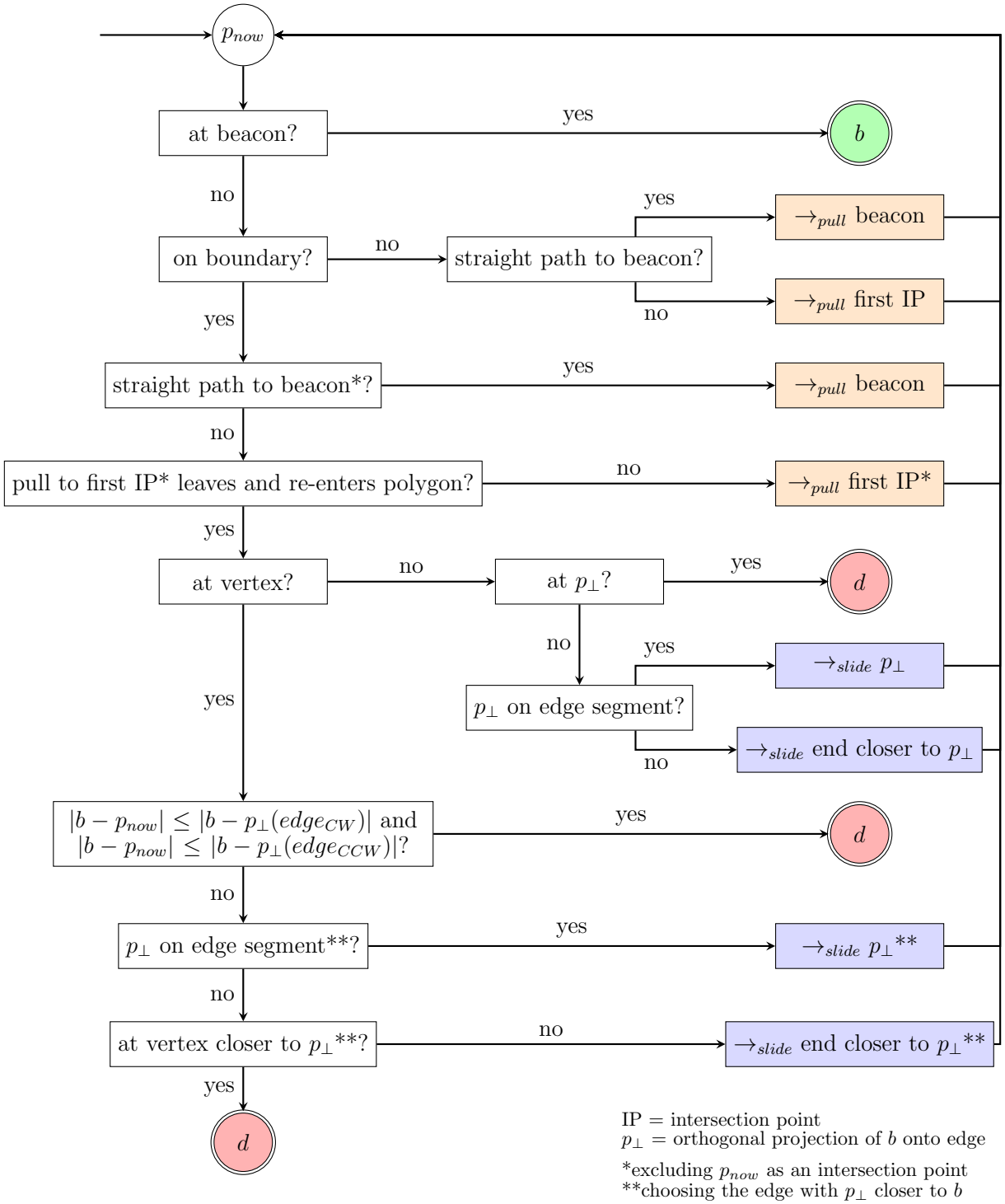


Figure 2: Recursive structure of the beacon attraction trajectory algorithm

for simplicity of calculation. If the midpoint is interior to the polygon, the pull edge is produced, with the algorithm recursing from the intersection point at its other end; if not, the pull edge is invalid, and the algorithm proceeds to determine if either a slide edge can be produced or the current position is a dead point.

Checking for a slide edge is handled differently at vertices than at other points along the boundary. If not at a vertex, the number of possible actions is more limited: the object can only slide along the current boundary segment, towards the orthogonal projection of the beacon onto the segment, until the object reaches either this point or the end of the segment. If already at the orthogonal projection, a dead point has been reached and the algorithm returns with failure. If not at the orthogonal projection and the orthogonal projection lies on the segment, the algorithm slides to it and recurses. Otherwise, the algorithm slides towards the projection point, but stops at the end of the segment and then recurses.

If currently at a vertex, the path will either terminate (having reached a minimum-distance corner) or continue with a slide edge along one of the two boundary segments joined by the vertex. If the current position is closer to the beacon than the orthogonal projection of either of the neighbouring segments, a corner has been reached and the algorithm returns with failure. Otherwise, only the segment whose projection is closer to the beacon is considered—if the projection point is on the edge segment, the algorithm moves to it and recurses. If not, however, the algorithm must ensure that it is not already at the end of the segment that is closer to the projection point. If this is the case, no movement can occur because the best projection point is further from the beacon than the current position, so the algorithm returns with failure. Otherwise, the algorithm slides across this segment to the far corner and recurses.

It is important to note that the behaviour at vertices where the object can be pulled along either neighbouring segment is not universally agreed-upon. Some authors dictate that only the counterclockwise edge is considered; others choose only the clockwise edge; a different mechanism of tiebreaking altogether may be used. This algorithm moves along the "steeper" edge in such cases, choosing the edge whose orthogonal projection is closer to the beacon, an assumption made to reflect the greediness of the overall model. If this approach results in a tie, where both edges are equally "steep" and a best-first choice cannot be made, the clockwise edge is chosen as other authors have done.

This procedure can be easily augmented to return the attraction trajectory path in addition to the final success result. At each level of recursion, the algorithm also returns the list of edges required to move from the current position to the beacon either or a dead point. Base cases where no movement occurs return an empty list; cases that produce a pull

or slide edge are handled by prepending the new edge to the return value of the recursive call. In languages that do not facilitate multiple return values, the algorithm need only return the path list, as it can be used on its own to determine success: if there is at least one edge in the path, success is attained if the end vertex of the last edge equals the beacon's position. In all other cases, the algorithm has produced a path ending at a dead point. If the path is empty, the same question can instead be asked of the starting position.

Encoding and Drawing Paths

With an algorithm to produce the attraction trajectory path, the application can calculate and render the full path, as well as animate the object's attraction towards the beacon over time. Paths are encoded as an ordered list of edges, generated as the algorithm recurses along the beacon trajectory path. The path begins at the object's initial position, and for each edge in the path, moves along the edge to its end vertex. Successful paths terminate at the beacon, while unsuccessful paths terminate at a dead point.

Visualizing and Animating Paths

The full path is drawn in the application as a polyline consisting of the edges produced by the attraction trajectory algorithm. If the final position is not at the beacon, a visual dead point is added to the end of the path, indicating the algorithm's failure to route from the object to the beacon.

Animation of the attraction trajectory entails moving the object along the path over time, at a constant speed. This is done by linearly interpolating along the polyline, translating a progress value in the range [0..1] into a point along the polyline. Playback begins at 0, and an animation loop controlled by the GUI increments this progress value and updates the scene. The object is animated by repeatedly rendering the progression of the object along its trajectory until the progress reaches 1.

The application includes the option of hiding the full attraction trajectory path, as well as drawing a trail of only past movement. This is done by using the animation facilities to draw only the portion of the path up to and including the current position.

Results

The application, shown in Figure 3, consists of a large variable-sized canvas on which the scene is rendered, as well as a toolbar providing playback controls and settings. Though the interactions can produce any combination of simple polygon, point object, and beacon, a handful of preset configurations are provided to make initial experimentation with the application faster, as well as to highlight some interesting aspects and corner cases of the beacon model. A "shuffle" button in the toolbar quickly repositions the object and beacon to random locations within the polygon.

The application is written in JavaScript, allowing it to be embedded within a web page or used via a web frame in a native application. The scene is rendered using PIXI.js, a JavaScript-based 2D graphics library used for creating interactive web applications and animations. Polygons and paths are drawn as polylines, with vertices and beacons as circles. All math is handled as vectors, using Sylvester.js for basic linear algebra operations.

The attraction trajectory is calculated whenever the polygon, point, or beacon are modified, after appropriate debouncing to ensure that the application remains responsive during short bursts of many interaction events.

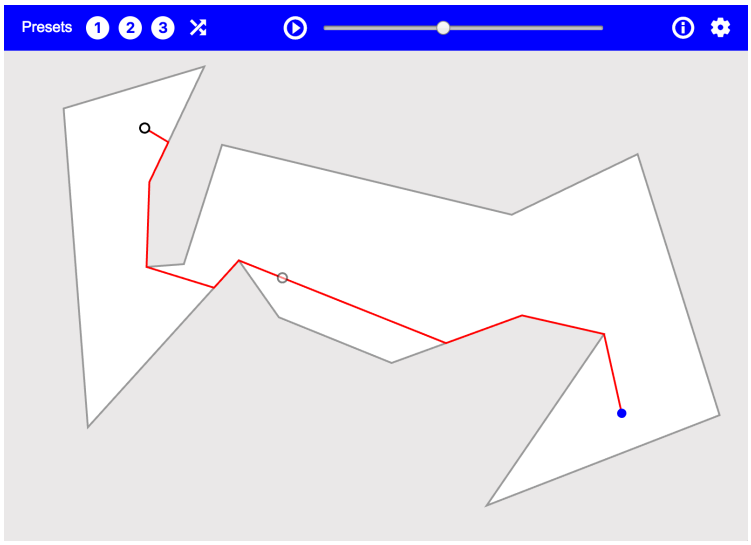
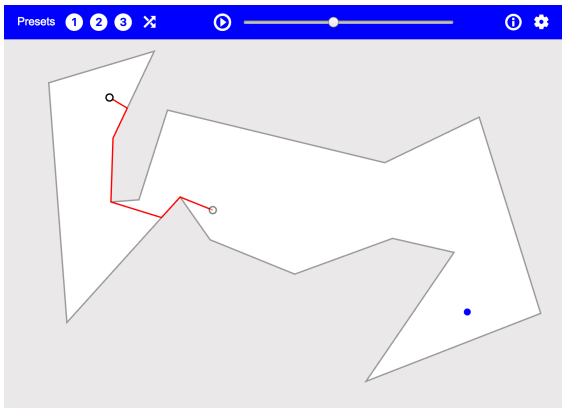


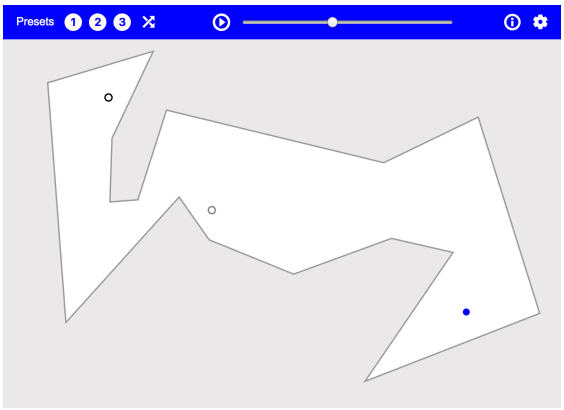
Figure 3: The application interface, consisting of a graphical canvas and toolbar

Playback controls enable animation of the object's position along the attraction trajectory over time. The animation can be started and stopped with a play/pause button in the toolbar. A progress slider indicates the beacon's current position in the attraction trajectory path, and enables scrubbing of the playback position. Playback can be looped repeatedly, and the playback speed can be adjusted from 0.1× to 5×.

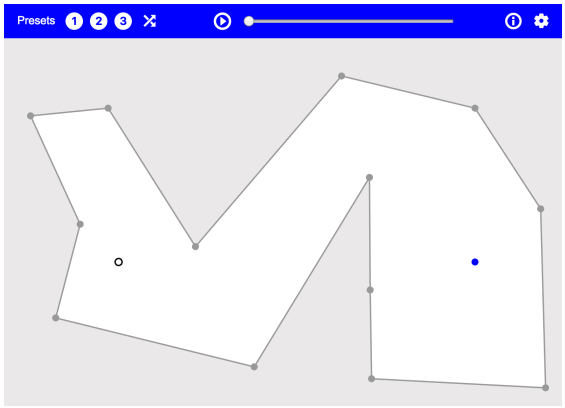
By default, the application displays a trail to represent the path taken by the object so far, as seen in Figure 4a. Options are provided to display the full attraction trajectory, as seen in Figure 3, or no path at all, as seen in Figure 4b.



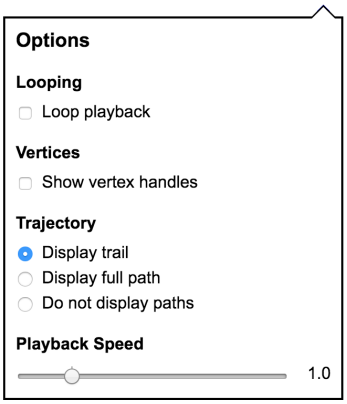
(a) Path trail only



(b) No path displayed



(c) Vertex handles shown



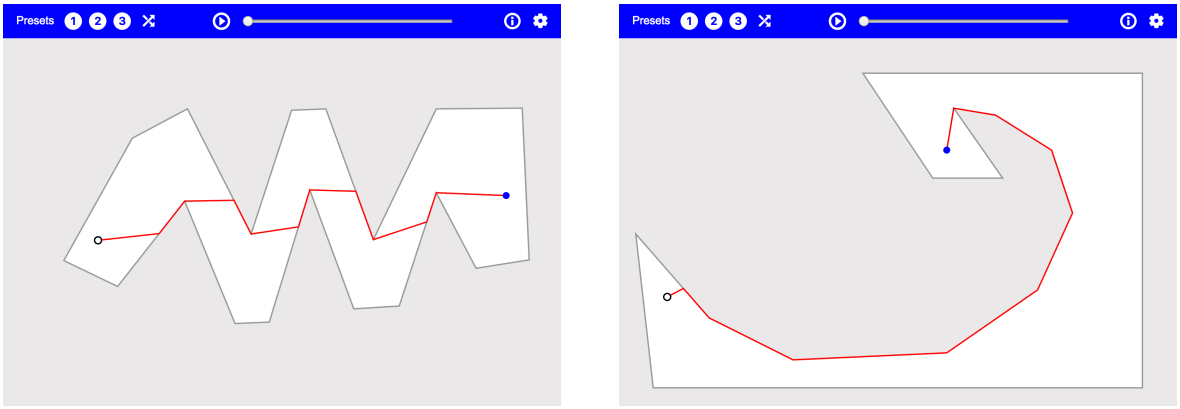
(d) Default application options

Figure 4: Visualization and playback options

Vertex handles and edge handles can be manipulated, subject to the various constraints outlined previously. An option is also provided to draw vertex handles non-transparently,

allowing for easier manipulation of the polygon. An example of the application with this option enabled can be seen in Figure 4c. The full set of options is shown in Figure 4d.

The application and underlying attraction trajectory algorithm handle a number of well-known difficult cases, as seen in Figures 5 and 6. Zigzags such as the one in Figure 5a present a number of situations in which the algorithm could terminate prematurely or accidentally cross the boundary of the polygon. Polygons that spiral around the beacon, where the orthogonal projection of the beacon onto each edge in the spiral is past the edge, should cause the object to be pulled inward along the spiral rather than stopping prematurely. The beacon in Figure 5b successfully attracts all points in the polygon.



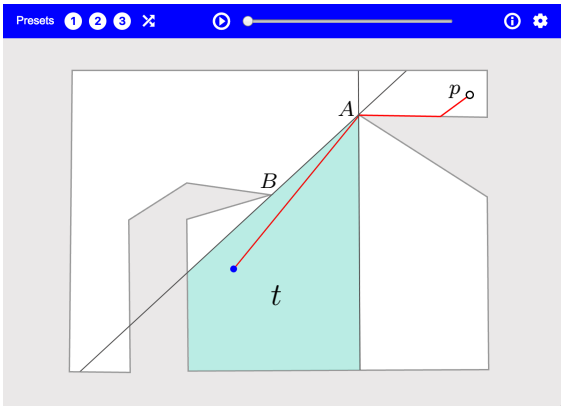
(a) Polygons with zigzags

(b) Polygons with spirals

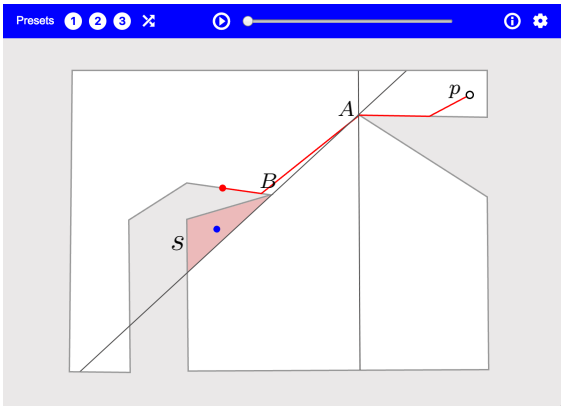
Figure 5: Commonly problematic polygons as handled by the application

In particular, the polygon in Figure 6 is an example that was used by Kouhestani [3] to illustrate some of the more complicated characteristics of beacon attraction. The diagonal $\overline{AB}$ introduces four distinct regions of attraction within the polygon as a result of the line $\overleftrightarrow{AB}$. Regions s and s' are on one side of $\overleftrightarrow{AB}$, with t and t' on the opposite side. Beacons in regions t and s' attract a point object at p, but beacons in regions s and t' do not. This instance highlights some of the intriciacies of the beacon model, as well as the non-triviality of a working attraction trajectory algorithm. As seen in Figure 6, the application correctly handles all cases.

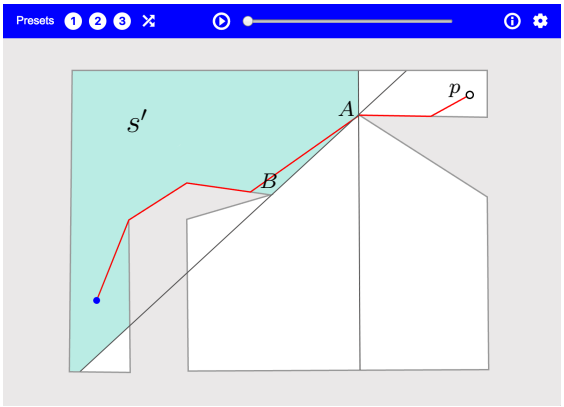
The examples provided constitute only a small number of the interesting cases and configurations that can be properly reproduced within the application.



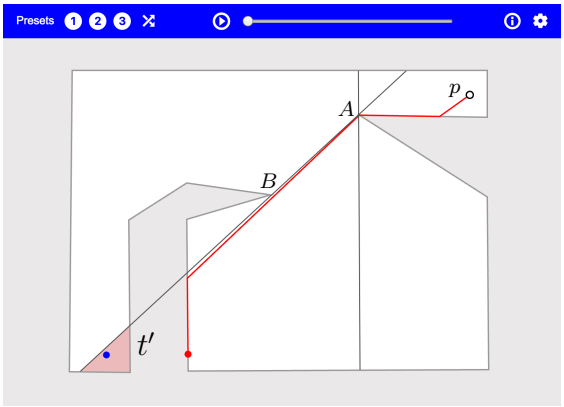
(a) Beacons in region t attract p



(b) Beacons in region s do not attract p



(c) Beacons in region s' attract p



(d) Beacons in region t' do not attract p

Figure 6: Correct representation of complex attraction trajectory behaviour

Open Problems

A number of open problems provide sources for future work. Some consist of extensions to this application in particular, while others are general open problems in the beacon model.

The attraction trajectory algorithm may benefit from more efficient implementations of some of the underlying functions, such as the detection of self-intersection in a polygon or containment of a point. This work would especially improve the calculation times of attraction trajectories for polygons with large numbers of vertices.

Placement of multiple point objects within the GUI is a trivial extension, entailing only minor changes to rendering and interactivity. The ability to place multiple beacons would require a broader extension to the beacon model and attraction trajectory algorithm. A multiple-beacon system may also allow for beacons with varying attractive strengths relative to one another, further complicating the problem.

Producing and manipulating holes within the polygon could be allowed through modifications to the GUI and appropriate changes to the functions underlying the attraction trajectory algorithm, while requiring no change to the algorithm itself. Handling polygons with self-intersections would require work on the beacon model, as neither the model nor the algorithm account for cases that are possible if self-intersection is allowed.

The *attraction region* of a beacon b in P is the set of points in P that b can successfully attract, while the *inverse attraction region* of a point p in P is the set of beacon positions that can attract p. As proven by Kouhestani et al. [4] the inverse attraction region can be determined in $O(n^3)$ time and $O(n^2)$ space. The *beacon kernel* of P is the set of beacon positons that can attract all points in P; the *inverse beacon kernel* is the set of points that can be attracted by all beacon positions in P. Biro et al. provided $O(n^2)$ algorithms for computing both kernels [5]. These existing approaches could be used to extend the application to visualize attraction regions, inverse attraction regions, beacon kernels, and inverse beacon kernels in real time.

Finally, extension of the algorithm to three-dimensional polyhedra remains an open problem in the beacon model [5, 6]. Obtaining a geometric intuition of the model in three dimensions is often far more challenging than in two dimensions, so a visualization tool would especially aid in the study of a 3D beacon model. However, this would require significant work on the beacon model as well as the application's GUI and rendering engine.

References

[1] M. Biro, J. Gao, J. Iwerks, I. Kostitsyna and J. S. B. Mitchell. *Beacon based routing and coverage*. Proceedings of the 21st Fall Workshop on Computational Geometry, 2011.

[2] B. Kouhestani, D. Rappaport and K. Salomaa. *The Length of the Beacon Attraction Trajectory*. Proceedings of the 28th Canadian Conference on Computational Geometry, 2016.

[3] B. Kouhestani. *Efficient algorithms for beacon routing in polygons*. Ph.D. Dissertation, Queen's University, 2016.

[4] B. Kouhestani, D. Rappaport and K. Salomaa. *On the Inverse Beacon Attraction Region of a Point*. Proceedings of the 27th Canadian Conference on Computational Geometry, 2015.

[5] M. Biro. *Beacon-Based Routing and Guarding*. Ph.D. Dissertation, Stony Brook University, 2013.

[6] B. Kouhestani, D. Rappaport and K. Salomaa. *Routing in a Polygonal Terrain with the Shortest Beacon Watchtower*. Proceedings of the 26th Canadian Conference on Computational Geometry, 2014.