Contents

Chapter 1

Introduction

1.1 The BeachBot Project

The BeachBot project is a focus project at ETH Zurich. During two semesters of the bachelor studies, the team had the opportunity to develop a mobile and autonomous robot prototype for creating sand drawings on beaches (the final robot prototype is shown in Figure 1.1). In total 7 mechanical engineering students, one electrical engineering student and two industrial design students (from the Zürcher Hochschule der Künste) were working on the project.
The result of the project is a three wheeled mobile robot that can drive autonomously. The key features are:

Localization The robot is able to reliably localize itself on the beach, using a laser range finder and 3 or more reflective poles. The accuracy of the localization is about 3 centimetres.

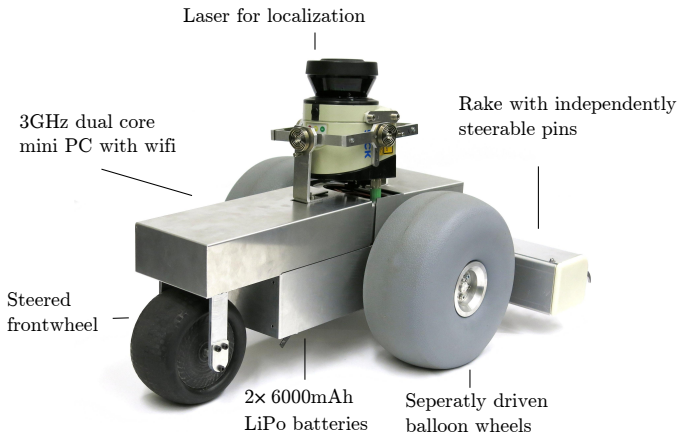
Driving speed and turning radius The top speed of the robot is about 0.4 metres per second and it can turn on the spot. Both back wheels are independently steerable. The front wheel is also actuated. This is done to reduce the risk of getting stuck in sand.

Rake The main drawing tool of the robot is a rake. The rake consists of seven pin-pairs which are individually liftable.

Controller The controller uses the output from the localization to steer the robot so that it follows a pre-defined trajectory.

Automatic Path Generation An application was created to generate the trajectory for an arbitrary input drawing that can then be drawn on the beach.

The robot has been successfully tested at the beach.

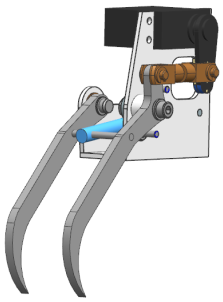


Specifications

Max. battery time
5h
Topspeed (theoretical)
1.3m/s
Power
2× 50W
Size (W × L × H)
40cm × 60cm × 40cm
Weight
12kg
Max. canvas size
28m × 28m

(a) The outer shell of the BeachBot

(b) The rake in action



(c) One rake unit with servo motor and pin pair

Figure 1.1: Various images of the BeachBot

1.2 Requirements and Inspiration

The BeachBot project itself was inspired by the images of sand artists like Peter Donnelly and Andres Amador who manually create large scale sand art at beaches using a rake. Some of the imagery that was found online can be seen in Figure 1.2. Since the goal of the project is to create drawings similar to those of the artists, the requirements were derived from these sample images.
First of all, the BeachBot should be able to draw lines. But more important is the support of filled areas. While it is relatively straightforward how to drive over lines or curves to draw them, there is no easy solution to derive the path for the filled areas. Not only should the area be covered to the highest extent, but the drawing process should also look interesting and artistic to spectators as well.
Derived from Figure 1.2 is also the requirement that the BeachBot only has to be able to draw in two colors. Gradients or differently colored areas are unnecessary.
Crossing lines or filled areas should be avoided as good as possible since the balloon wheels and the front wheel leave visible marks on the raked sand (a humanoid has a huge advantage here, since it can jump over the drawn areas). The effect of driving over the drawing depends on the scale. However, if the drawing is very big, it does not matter much if part of the image is crossed out.
The input of the path generator should not only be computer readable, but also human editable. Typing in endless lists of coordinates would be tedious in the long run, and having a difficult format to work with would make it hard to collaborate with artists, for example. In the project vector graphics, which can be created and edited by tools like *Inkscape*[1], are used as the input to the path generator software. During the testing phase it was found out, that some of the generated paths needed some manual adjusting. To be able to easily edit the output of the generator program a graphical user interface should be created to work with the output of the generator program.
In the following chapters it will be shown how and to which extent these requirements are fulfilled.

Path Converter Another bachelor thesis that was written as part of the BeachBot project deals with the rake offset of the robot. The rake is not at the same spot as the turning point of the robot, which leads to errors when the image is drawn in the sand. Therefore, the path converter uses an optimization technique to adjust the path generated by the software developed in this work in a way that the rake closely follows the generated path by offsetting the robot trajectory. The Path Converter is introduced here because, as it will be discussed later, leads to certain limitations that had to be regarded for the design of the path generation algorithms. Indeed, the path that is generated in this thesis is the rake path and not the robot path.

[1] `http://inkscape.org`

(a) Sand drawing by Peter Donnelly. Source: `http://becky-garrett.blogspot.ch/2009/03/sand-dancer-peter-donnelly.html`

(b) Sand drawing by Andres Amador. Source: `http://sftimes.co/?id=25`

(c) Sand drawing by Andres Amador. Source: `http://sftimes.co/?id=25`

Figure 1.2: Various beach drawings by artists

Chapter 2

Path Planning Algorithms

2.1 Overview

The output of the path generator should be a single trajectory that completely connects and covers all elements of the drawing.
Derived from the requirements, three different elements were identified as part of drawing (illustrated in Figure 2.1):

Polyline A line consisting of 2 to n vertices.

Polygon A closed line, consisting of 2 to n vertices, where the last segment is a closing one. Vertex v_{n+1} coincides with v_0.

Filled Polygon Defined in the same way as the polygon, except that the inner space should be filled by a generated trajectory. Another difference is that the filled polygon can also contain holes, which should not be covered and are excluded from the fill trajectory.

The path generation happens in three steps: First, the polygons that have to be filled are selected and the fill algorithm is seperately executed for each of the polygons with polygon specific settings. In a second step all of the drawing elements are connected by an open tour and a Traveling Salesman heuristic minimizes the total driving distance. As last step, the connections are shaped with curves that conform to a curvature limit.
Each of those steps will be discussed in detail in this section.

2.2 Polygon Filling

2.2.1 Related Work

Over time several complete coverage algorithms have been developed, and some distinctions can be made.

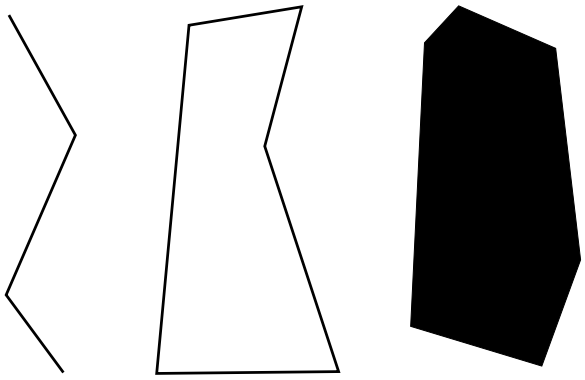


Figure 2.1: Polyline, polygon and filled polygon

One category of solutions for the surface coverage was developed for robots that do not know the surroundings beforehand and use a variant of *simultaneous localization and mapping* (SLAM) to gather knowledge of the environment.
Coverage algorithms exist for household applications like autonomous vacuum cleaners or autonomous lawn mowers but also search and rescue robots usually and they do not care if they visit the same spot twice. The target for those algorithms is rather to achieve complete coverage of an previously unknown terrain in sensible time. Usually, the complete surface coverage algorithms in are also connected with online map generation techniques *SLAM*, whereas the path generation for the BeachBot should happen offline.
However, in agricultural applications some interesting algorithms have been found, which served as inspiration for the explorations presented in this thesis. Especially [1], who hinted at exploiting the straight skeleton algorithm to generate the inset polygons. An optimization strategy is employed to find the shortest trajectory through the field by repeatedly offsetting the remainding shape and traversing all possible ways off filling the shape. The algorithm is relatively computationally intensive, what might be justified when using large agricultural machines but what was not necessary for the goals of this thesis as the benefit of this optimization would be relatively small.
Another field where trajectories have to be generated is in *Computer Aided Machining* (CAM). The process of removing layers of material from a block of metal is quite similar (though inverse, usually) to what is achieved in this thesis. Many publications deal with the problem of multi-axis milling machines which are far more complex and are also able to move over the machined surface without problem because the machining head can be lifted – something that is not possible with an autonomous ground vehicle.
One interesting publication in the CAM field is [2] that presents a method to reduce gaps that are present when simply offsetting a polygon with spirals. The presented method could be a possible future improvement to the spiral fill algorithm of **??**.
The OpenCAM library has also been inspected...

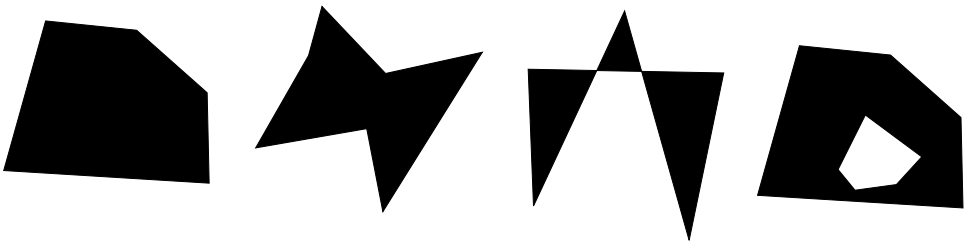


Figure 2.2: Four different polygon types, from left to right: (a) simple and convex, (b) simple and concave, (c) self-intersecting, (d) polygon with a hole

2.2.2 Mathematical Discussion of Polygons

The following is a mathematical definition of a polygon (taken from [3]):
"A polygon may be defined as consisting of a number of points (called vertices) and an equal number of line segments (called sides), namely a cyclically ordered set of points in a plane, with no three successive points collinear, together with the line segments joining consecutive pairs of the points. In other words, a polygon is closed broken line lying in a plane".
Polygons can have several properties that simplify or complicate their treatment:

Simple A simple polygon is not self intersecting and has a well-defined interior and exterior.[4]

Convex In the convex case any line can intersect the polygon only at two points. The interior all have to be smaller than 180°.[5]

Concave A concave polygon can be intersected more than twice by a given line. Therefore, at least one interior angle has to be larger than 180°.[6]

Self-intersecting A self intersecting polygon has at least one side which intersects with another side.

Polygon with Holes A hole of a polygon is an enclosed or partially enclosed area that is not part of the filled surface of the outer polygon.

2.2.3 Spiral Filling

The first method to cover the area of an arbitrary polygon is the spiral fill method using inset polygons that are created using the straight skeleton of the polygon.

Straight Skeleton

For this thesis, the most interesting property of the straight skeleton is the ability to easily obtain inset polygons with an arbitrary offset. The straight skeleton is defined as the topolgical skeleton of a polygon that is created by moving the edges

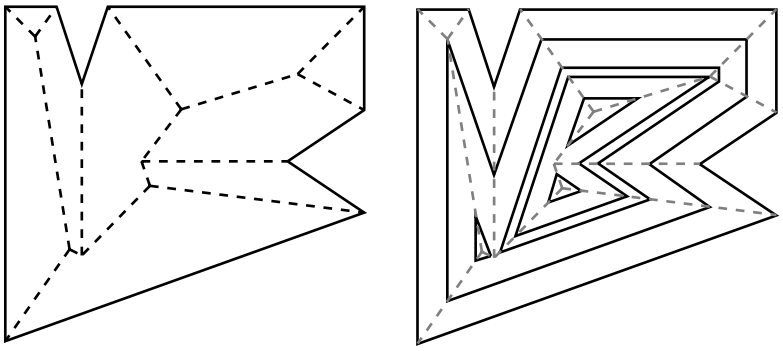


Figure 2.3: Straight skeleton and generated inset polygons (right). Split and edge events are well visible. Figure modified from [7].

parallel to themselves inwards at a constant speed and observing the intersections of the vertices (first described by Aichholzer et al. (1995) [7]). It is similar to the median axis of a polygon, but unlike the median axis it is not defined to have a constant distance to the polygon edges. Another commonly used offset mechanism for polygons in motion planning is the Minkowski sum. However, there is no defined Minkowski subtraction, which makes the creation of inset polygons more complicated[1]. An efficient implementation to compute the straight skeleton is discussed by Huber [8].
To create a spiral trajectory, two events that occur during the creation of the straight skeleton have to be kept in mind (see Figure 2.2):

Edge Event Depending on the length difference between the edges, one edge might be reduced to zero sooner than the others, resulting in an edge event. The neighboring edges are adjacent after the event. At this point, the inset polygon will be reduced by one vertex.

Split Event A concave polygon will, at some point during the shrinking process, intersect itself. At the intersection point, a split event happens. After the split event, the number of inset polygons is increased by one.

The inset polygons are used to create a spiral fill for the polygons. The procedure is as follows:

1. The straight skeleton is generated.
2. The first inset polygon is created. The inset distance is the constant width of the rake divided by the number of vertices of the polygon. Through dividing the inset length by the number of vertices in the polygon, one revolution of the spiral will travel one rake distance inwards.

[1] It is possible to create an enclosing polygon, subtract the original polygon and offset the enclosing polygon with hole by taking the Minkowski sum (which also offsets the "hole").

3. If a split event has happened, then it is decided which polygon should be used to extend the current spiral (that is the one with the closer point to the current position). All other newly created polygons are recursively filled by the same algorithm.

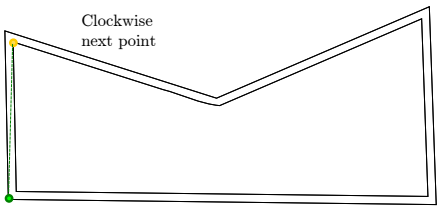
4. The closest point to the current point on the inset polygon is searched (v_n). The next vertex of the inset polygon in clockwise direction (v_{n+1}) is appended to the current spiral

5. The next inset polygon is generated with inset length divided by the number of vertices of the previous inset polygon, and the process continues at (4).

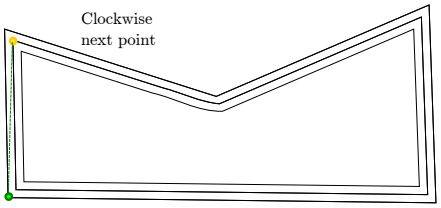
Algorithm 1 Spiral Filling

```
function createSpiralFill(Polygon p, Point startPoint)
    RakeWidth ← (const. rake width for BeachBot)
    result ← empty list of points
    ss ← getStraightSkeleton(p)
    lOffset = RakeWidth/p.numVertices()
    if startPoint then currPoint ← startPoint
    else currPoint ← p.firstVertex
    end if
    insetPolys ← getOffsetPolygons(ss, lOffset)‘
    while insetPolys.numPolys() ≥ 1 do
        if insetPolys.numPolys() > 1 then
            closestPoly = searchClosestPolygon(insetPolys, currPoint)
            for all {poly ∈ insetPolys|poly ∉ closestPoly} do
                createSpiralFill(poly, currPoint)
            end for
            ss ← getStraightSkeleton(closestPoly)
            lOffset = RakeWidth/closestPoly.size()
            insetPolys ← getOffsetPolygons(ss, lOffset)
        end if
        index ← findClosestIndex(currPoint, insetPolys.firstPoly)
        currPoint ← insetPolys.firstPoly.pointFromIndex(index + 1)
        result.appendToList(currPoint)
        insetPolys ← getOffsetPolygons(ss, lOffset)
    end while
end function
```

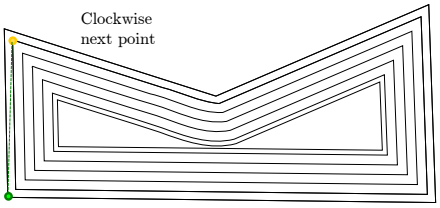
The algorithm is both displayed in Algorithm 1 and graphically in Figure 2.4. The result of the algorithm is presented in Figure 2.5.



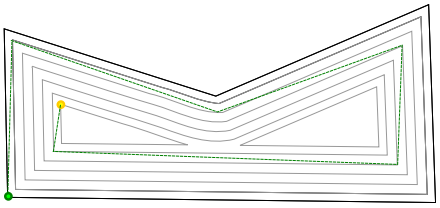
(a) The first inset polygon and the first part of the spiral line



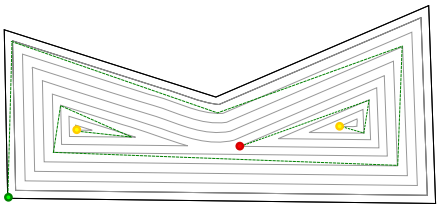
(b) Next inset polygon generated



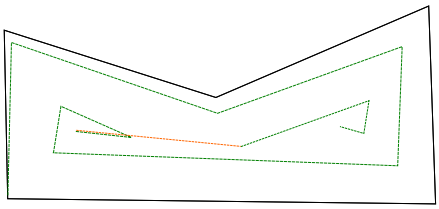
(c) The polygon right before the *split* happens



(d) After split, next polygon was found for continuing the current spiral



(e) Continuing spiral 1 and starting the second spiral. Note how the start point of the second spiral is the closest point to the endpoint of the first.



(f) The completed spiral with all inset polygons removed.

Figure 2.4: Generating the fill spiral. Note that this is only an example for illustration purposes. The density of inset polygons for a real application is much higher.

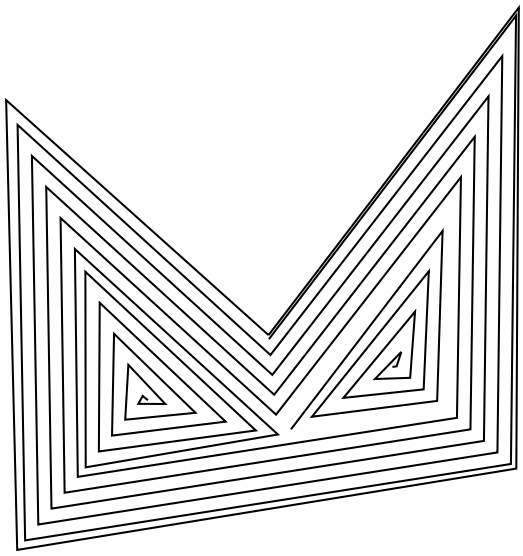


Figure 2.5: Generated spiral

2.2.4 Zig Zag Filling

This fill method is quite straightforward: the polygon is filled by a number of lines that are parallel to each other and clipped at the polygon edges. The direction of the lines can be freely chosen.
To place the lines, the starting point is determined. If the chosen direction for the fill lines is inclined by more than 45°, the leftmost vertex is used as starting point, otherwise if the inclination is smaller, the bottommost vertex is selected.
Beginning from the starting point, a line in the specified direction is placed and the polygon edges are tested for intersection. If two intersections on the polygon edge for a given line are found, a line segment with those two intersection points is added to the resulting trajectory. The starting point of the line is translated by the specified rake width constant in the direction normal to the chosen direction and the process is repeated. In the case that no two intersections are found any more, the algorithm stops.
This method works very well for convex polygons, but is not applicable for non-convex polygons, since a non convex polygon can have more than two intersections for any given line. Therefore it requires the polygon to be decomposed into convex parts. It was decided to use the optimal convex partitioning with the algorithm by Greene [9]. The decomposed parts have vertices that coincide with the original vertices. Since it is an optimal convex partitioning algorithm, the number of created convex polygons is minimal (therefore the size of each decomposed part is maximized). Having larger decomposed parts is positive for the trajectory, because fewer turns in general reduce the drawing time and make for a more consistent look. Additionally, the created segments (or the convex polygon) are offseted by a certain margin inwards to make way for the turns and reduce the amount of drawing that extends to the outside. As last step, each segment is driven with the rake lowered and the complete polygon is also traversed.
Because the Zig Zag filling can be thought of as one continuous line, all generated lines are connected with an *enforced connection* (which is explained later) to reduce the amount of free connection points that the Traveling Salesman algorithm has to examine.

2.3 Path Generation

As already mentioned, to finally create a drawing all elements have to be connected by support trajectories that are not part of the drawing (where the robot drives with the rake lifted). The supporting paths should have two properties: the total distance of all support paths should be as small as possible and the curvature of the connections should be limited.
As previously defined, a drawing consists of the three different element types: lines, polygons and filled polygons. Lines and polygons differ in the way they can be connected, as illustrated in Figure 2.8. A line has two free connection points and is traversed from one end to the other. A polygon however can be entered at any vertex, but can only be exited at the same vertex so that it is closed. Filled polygons

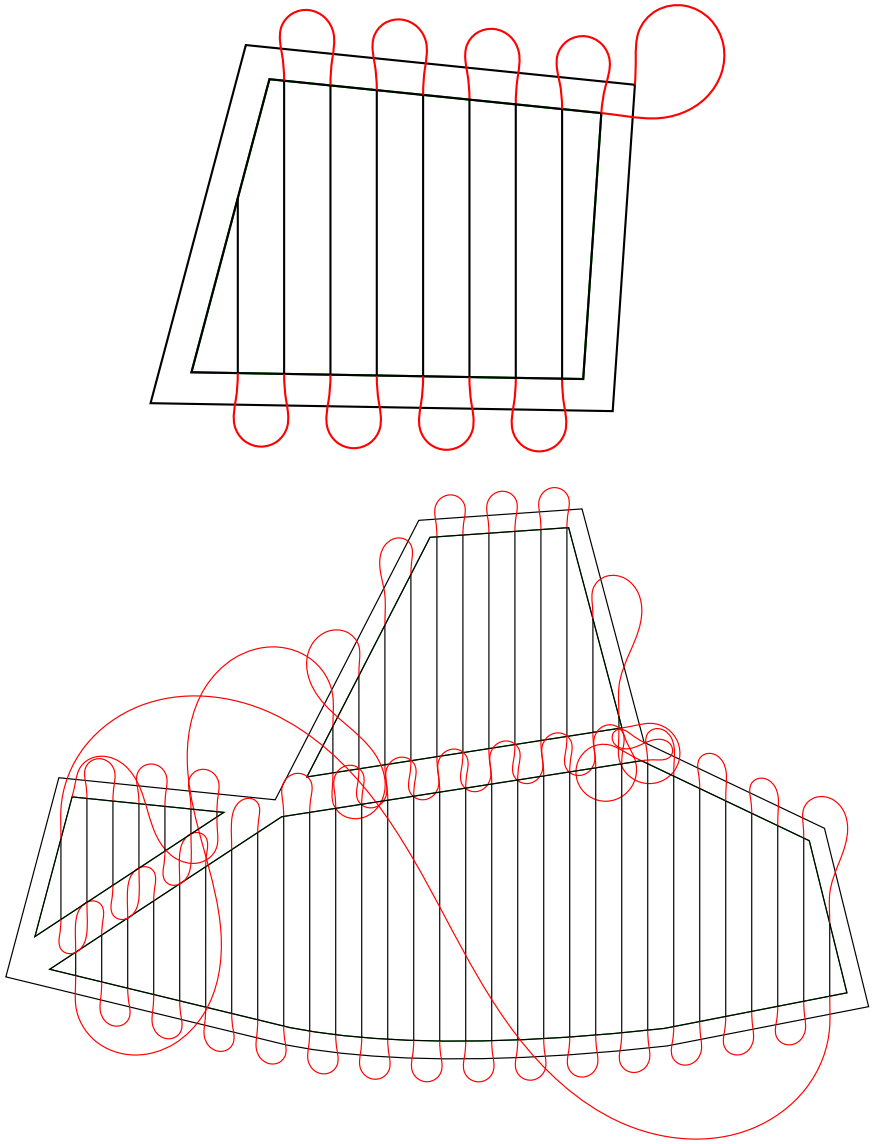


Figure 2.6: Zig Zag filling of two shapes. The first shape is convex, the second concave. The second shape is partitioned into convex elements using the optimal convex partitioning, which are separately filled with the Zig Zag fill.

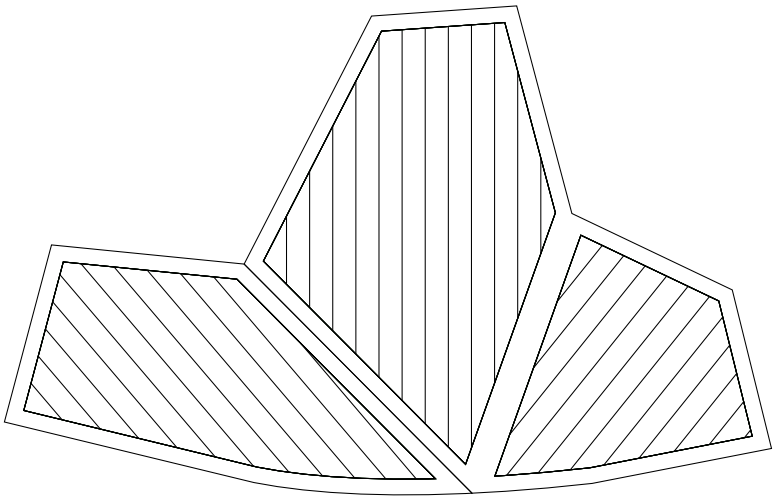


Figure 2.7: Users can choose an arbitrary segmentation and Zig Zag direction in the user interface

work the same as the regular polygon in that sense, since the spiral and the zig zag filling create a polyline which is inserted into the polygon.

2.3.1 Traveling Salesman Problem

The task of connecting the drawing elements is similar to that of a Travelling Salesman Problem. The *Travelling Salesman Problem* (TSP) is the problem of finding the minimum-weight Hamiltonian Circuit in a weighted graph. A weighted graph is a graph where every edge between two vertices has a certain weight associated. The Hamiltonian Circuit is defined as a tour through the graph that travels to every vertex exactly once forms a cycle. It is called travelling salesman problem because in the historical context the vertices were cities, the weights Euclidean distances and the salesman, seeking to minimize the distance travelled to do a round-trip through all the cities in a given tour, would seek to obtain a solution for this problem. However, one of the properties of the TSP that make this problem difficult to solve is that it is *Non-deterministic Polynomial-time hard* (NP-hard) which means that, although the non-existence has not yet been proven, there is as of now no polynomial-time solution. If no heuristic is used, the amount of solutions that have to be searched in order to test all available possibilities is "$n!$".
The weights of the problem are usually defined in a distance matrix $D^{n \times n}$ matrix, where n is the number of vertices in the graph and the elements in the matrix are given by d_{ij}.
The following properties for a general Travelling Salesman Problem are defined:

Symmetric A TSP is symmetric if all distances between every node pair are symmetric (equal) in both directions ($d_{ij} = d_{ji} \; \forall \; i, j \; , \; D^{\intercal} = D$).

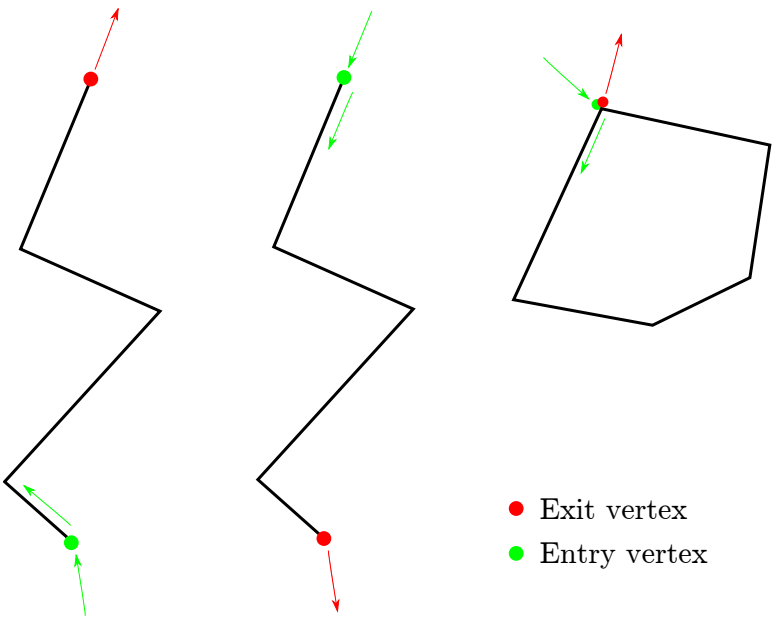


Figure 2.8: Possible entry- and exit vertices and direction of traversal for polylines and polygons. Note that the polygon could also be traversed in clockwise direction.

Metric In a metric TSP, all distances conform to the triangle inequality $d_{xy} \leq d_{xz} + d_{zy}$ which is, for example, true if the distances are calculated using the Euclidean distance formula between two points: $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Or for the so called Manhattan distance which is defined as $d_{ij}^M = |x_i - x_j| + |y_i - y_j|$.

Asymmetric If $d_{ij} \neq d_{ji}$ for at least one distance pair, then the TSP is asymmetric. The triangle inequality is not guaranteed to be fulfilled any more.

The easiest heuristic to the TSP is the nearest neighbour search. From a given starting point the nearest and unvisited vertex is searched and connected. This is done until no more unvisited elements exist. Positive about the nearest neighbour approach is that the complexity is only growing linear to the search size. However, the obtained solutions are often of unsatisfying quality.
Another standard heuristic is the branch and bound method. It works by splitting the set of possible solutions into two sets and calculating an upper and lower bound for both sets. If the upper bound of the first set is lower than the lower bound of the second, the second is discarded (and vice versa). During the course of this thesis a variant of the depth-first branch and bound search was implemented. First, the graph is traversed until a tour is completed. Afterwards, another tour is recursively started. As soon as the travelled distance of the entire tour in the new branch gets higher than the current minimum tour, the branch is discarded. While this already

reduces the search space, the effect becomes relatively small on larger problems (e.g. the first 6 out of 15 nodes might only seldom span a tour where the distance is higher than the current minimum, and it gets only worse). Only the basic version of the heuristic was implemented.

While the branch-and-bound implementation could have been optimized by a good margin, it became clear that for a drawing of reasonable size a state-of-the-art implementation had to be found. Currently there are, among others, two well-known state of the art implementations for the TSP[2]: Concorde[3] and the Improved Lin-Kernighan Heuristic of Helsgaun (LKH)[10][11]. The Concorde solver is one of the best currently available solvers which produce an exact solution[12]. However, it only works with symmetric distance matrices. While it is possible to transform a general non-Euclidean asymmetric problem into a symmetric one, the resulting matrix is bigger (in the case of [13] the transformed distance matrix is of size $2n \times 2n$), which will increase the computation time. Furthermore, an exact solution is not needed. Contrary to the Concorde solver, the Improved LKH of Helsgaun is able to directly work with asymmetric distance matrices, which makes it an ideal candidate for the purpose of this thesis. Furthermore, it is using a heuristic which is not guaranteed to obtain optimal solutions, but optimal solutions are arguably not needed for the case of a sand drawing.

The Lin-Kernighan heuristic is a local search optimization algorithm that works by using 2–opt and 3–opt moves. For a 2–opt move, 2 connections between 4 vertices are cut and the only other possible connection is checked (there is only one other possible connection because otherwise two closed tours would result). If the resulting tour is shorter, it is picked as new basis tour. Repeating this step generates a more optimal solution each time.

An extension of the 2–opt move is the 3–opt move, where connections between 6 vertices (3 connections) are removed and all new connection possibilities are evaluated. The Lin-Kernighan algorithm extends this even further and works with λ–opt moves. This generally degrades the runtime to $\mathcal{O}(n^{\lambda})$. Therefore the Lin-Kernhighan algorithm uses variable λ-opt moves: starting at $\lambda = 2$ at each step is evaluated if λ should be increased by one or not through applying a series of tests. At a stopping condition this process ends.

The author, Keld Helsgaun, also demonstrates how the specific LKH implementation is suited to solve the equality generalized[14] as well as the clustered TSP[15]. A combination of both cases is used in this thesis to find a high quality tour through the drawing of.

The LKH solver uses the *TSPLIB*[4] file format as interface to read and write the problems. The TSPLIB is a benchmark collection of various traveling salesman problems, where some of them have known optimal solutions.

[2] according to `http://www.adaptivebox.net/CILib/code/tspcodes_link.html`

[3] `http://www.math.uwaterloo.ca/tsp/concorde.html`

[4] `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`

2.3.2 Adaptation of Traveling Salesman Problem for the Algorithm

To find a suited tour with the generic LKH solver, the correct weights for the distance matrix have to be specified. The term "outgoing" is used for all weights from one specific vertex to all other vertices and "ingoing" for the weights from all vertices of the graph to one vertex. This happens in the following steps:

- On polylines, all vertices that are not the first or last node are removed, since they can never be accessed without entering at either the first or last node
- Polygons can be entered at any node. However, once entered, the polygon cannot be exited at any location but must be traversed completely. To enforce this, the weights on the polygon edges are set to be zero for the next vertex in clockwise direction and infinity for all other nodes of the polygon.
- Every node is only visited once. This leads to problems in polygons, where the tour should be closed. There are two possible solutions: double cities or weight shifting. With the double cities approach (illustrated in Figure 2.9), each vertex on the polygon is added two times to the distance matrix: one with Euclidean distances as ingoing weights and all outgoing weights set to ∞ and the second with Euclidean distances as outgoing weights and ingoing weights set to ∞. The weights between all cities on the polygon are set to zero in one direction, i.e., clockwise (also between the double cities). Therefore the exit city has to be positioned *behind* the entry city, so that it is the last visited city by the algorithm which is forced to close the circuit around the polygon. Since the double cities approach is doubling the number of cities it is clearly the less optimal approach. Because it is known that entering at node n will result in the visitor ending up at node $n-1$ if all edges are zero, using the weight shifting method, the outgoing weights of the polygon at node n are shifted to node $n-1$. Thereby, the element-wise symmetry is restored and a optimal solution can be found with a generic TSP solver. In a post processing step, the polygon is closed again.
- All other edge weights are the Euclidean distances between each vertex pair.
- A regular TSP tour is closed. This implies that a regular tour has a tendency to go back to the starting point after diverging into another direction at the start of the tour. The solution that was found is to enforce a defined starting point with special properties to be inserted. The outgoing weights of the start point are either euclidean distances or defined to be set to the same value to every node in the graph. If they are euclidean, it is very likely that the first visited line is the nearest neighbour. If the outgoing weights are all defined to be a constant value the LKH can find the best suited first tour point on its own because the starting point will not induce any preference for any of the nodes. With the same reasoning, the weights pointing back to the starting

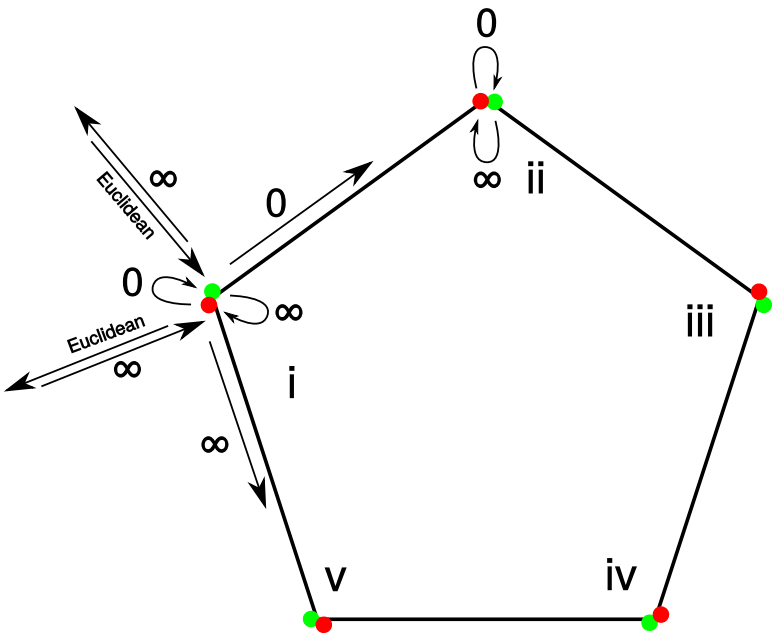


Figure 2.9: Double cities approach to solve the TSP on a polygon.

point are chosen to be equal for all nodes. Thus, the TSP will never optimize towards travelling back to the starting point.

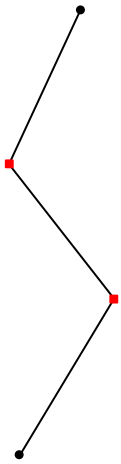
How the weights are aquired is also illustrated in Figure 2.10. The distance matrix of a simple drawing is shown in Figure 2.11.

Reprojecting the Tour The output of the LKH algorithm is a text file with ordered *tour indices* (shown in Figure 2.11c). The text file is read back in the main program and evaluated. Since the LKH algorihtm is not aware of the structure of the problem at hand, the indices have to be "projected" to the drawing elments. The tour indices correspond to the matrix indices and the ordering is that of the shortest found tour through the elements, vertex by vertex. To connect all elements in the way the tour was found, first the tour starting point is searched among the tour indices (the tour starting point index is always "1").
All other indices have to be saved beforehand and need to be reapplied now: The tour index corresponds to a vertex on a given element. The elements are now connected in the order they appear in the found tour. For the connection only the first tour index on each element is relevant, because the exit point of the element is trivially known for each type of element when the entry point is given.

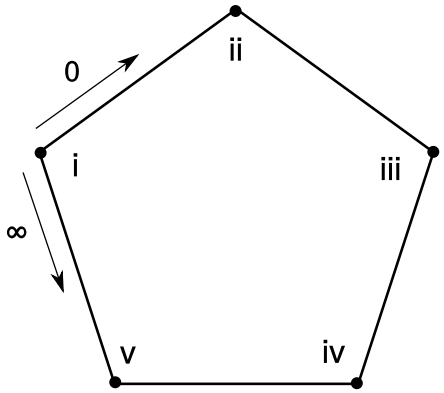
Enforcing Connections If the found solution is not sufficient or satisfying, the user might want to alter the tour. This works through the enforcement of con-

nections between two points on the elements. If two elements have an enforced connection, they can be represented by a single line with entry and exit point and an unknown traversal direction. This is due to the fact that if one point on an element is connected, there is a trivially known other point that is the last available connection point. For example, if a polygon has an enforced connection at vertex v_i, it is known that v_i will also be either start- or endpoint for the connected elements because the polygon loop has to be closed. Similarly, if the first or last point of a polyline has an enforced connection the other is the remaining free connection point that can be used to connect to other elements. Thus, instead of the single elements, the start and endpoint of said "imaginary" polyline are added to the distance matrix and the tour is completed during the post processing. An element with two enforced connections is completely ignored in the distance matrix, since it has no free connection points.

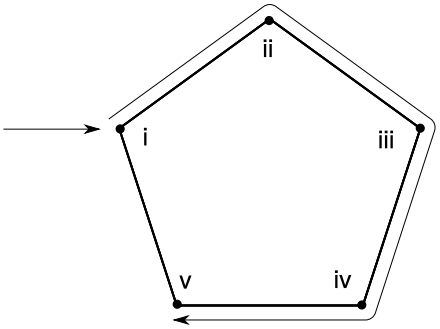
Containment Properties A polygon can enclose other elements, such as a polygon or a set of lines. Generally, to avoid crossing too many lines, it is undesired to enter this enclosing polygon twice. Therefore the weights of the polygon vertices towards the inner elements are set to infinity. Thus the tour will always start on the inside and continue to travel further outside.



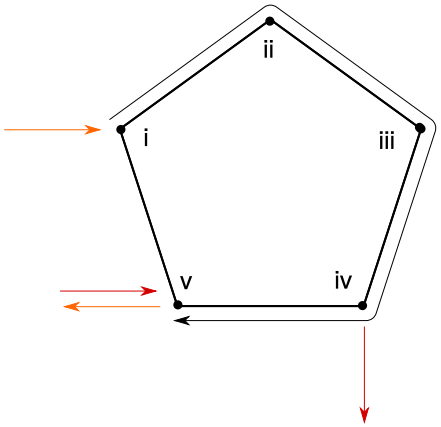
(a) A polyline. The red squares (inner vertices) will be removed in the distance matrix as they cannot be reached from the "outside".



(b) A polygon with the traversal direction on the polygon itself: Only the clockwise direction is allowed (arbitrarily chosen).

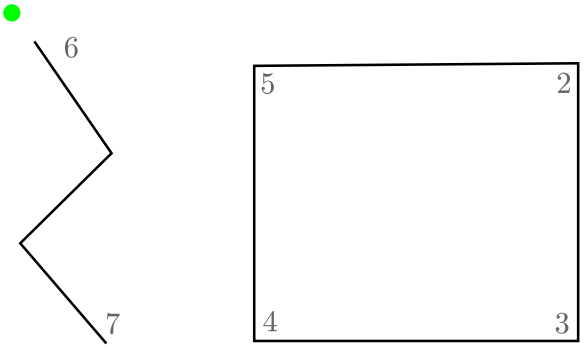


(c) Entering the polygon at node (i) it is known that the exit vertex must be (v), because no vertex is visited twice and all intermediate edges have a weight of zero.



(d) Because the exit node is known, the outgoing weights from node (i) are shifted to node (v). Since the polygon can be entered from any vertex, the same happens to all vertices in the polygon.

Figure 2.10: Illustrating how weights for the distance matrix are determined



(a) Simple TSP figure: tour startpoint in green, and two elements. Note that the "inner" vertices of the polyline are removed in the matrix. Gray numbers indicate corresponding matrix indices.

$$D = \begin{pmatrix} - & 1812 & 2097 & 1328 & 806 & 156 & 1127 \\ 1 & - & 0 & \infty & \infty & 1959 & 1489 \\ 1 & \infty & - & 0 & \infty & 1173 & 467 \\ 1 & \infty & \infty & - & 0 & 698 & 993 \\ 1 & 0 & \infty & \infty & - & 1717 & 1732 \\ 1 & 1717 & 1959 & 1173 & 698 & - & 0 \\ 1 & 1732 & 1489 & 467 & 993 & 0 & - \end{pmatrix}$$

(b) The corresponding matrix: First row and column are values of the tour start point. Thereafter, the distances of the polygon follow (note how the zeros are shifting and the shifted symmetry of the rightmost columns and bottom rows). The last two entries correspond with the polyline (where the two inner vertices are removed).

$$\{1, 6, 7, 4, 5, 2, 3, -1\}$$

(c) The indices of the shortest tour through the drawing, as returned by the LKH application.

- Index 1: tour starting point
- Indices 6 & 7: points on the polyline
- Indices 4, 5 2, 3: points on the polygon
- Index -1: indicates return to start (end of tour).

Figure 2.11: Drawing, TSP distance matrix and final tour for a simple drawing

2.4 Smooth Line Connections

While not required by the robot kinematics of the BeachBot (which can turn on the spot), generating smooth connections makes the developed algorithms generalizable for four-wheeled vehicles. It is also necessary to constrain the curvature for the path converter that translates the generated rake path into a robot path that is drivable by the bot. If the curvature at a given point of the connection is too high, it is not possible for the path converter to find a converging solution which makes tedious manual work needed to adjust the curvature of the connections until the converter is able to find a solution. The rounded connections would also create a nice visual effect for spectators.
Ideally a way would have to be found that limits the curvature of the connection path.
Curvature is defined as the inverse of the radius of the circle that the tangent produces at any given point in the curve. For a two dimensional curve, the curvature κ is defined as

$$\kappa = \frac{|x'y'' - y'x''|}{(x'^2 + y'^2)^{3/2}}$$

Beziér Splines

A first approach to generate more curved connections was to use Beziér splines. The start- and endpoint of the beziér spline is trivially found as the start- and endpoint of the connection. As first solution, we used beziér splines of 3rd order. Beziér splines of 3rd order only yield 2 control points, which can, by moving them along the desired tangent, easily be used to create a connection that is smooth in the start and endpoint. However, the curvature for the complete curve can not be set because the two control points offer to few degrees of freedom. A beziér curve of 5th degree (quintic) has enough DOF to constrain the connection in terms of curvature.
The quintic Beziér curve consists of 6 control points $P_0 ... P_5$ that define the shape of the curve. P_0 and P_5 are trivially chosen because they coincide with the start- respectively the endpoint of the curves that should be connected. P_1 and P_4 are also easy to choose as they have to lie on the tangents of the curves that should be connected. P_2 and P_3 on the other hand are more difficult to obtain. Even after an extensive search through available literature it remained unclear if an analytical solution to this problem can be found or not, since the equation for curvature is getting quite complicated if expanded. In [16] three approaches to generate beziér splines with monotone curvature are discussed. [17] uses piecewise 3rd degree bezier curves to create a curvature and corridor constrained trajectory. [to be expanded]

Spiro Splines

Spiro splines, introduced by R. Levien [18] are a different approach to designing fair curves, initially for the purpose of designing fonts. They have several properties which make them well-suited for the task at hand:

They offer 4 different control points to control the shape of the curve. In contrast to beziér curves, spiro spline control points are always passed by the generated curve, whereas the bezier curve only goes through the first and last control point. The 4 different control points, denoted by a single character, of spiro splines are:

- `v` Corner point
- `c` A G^2 continous constraint. The curvature on the left side is the same as on the right side ($\kappa_l = \kappa_r$).
- `o` Similiar to `c`, a G^4 continous constraint. Not only $\kappa_l = \kappa_r$ is true, but also the first and second derivative of the curvature is constrained ($\kappa_l' = \kappa_r'$, $\kappa_l'' = \kappa_r''$).
- `[` A straight-to-curved control point that acts like a tangent constraint in this case.
- `]` A curved-to-straight control point that also acts like a tangent constraint here.

The author, Raph Levien, has licensed his implementation [5] of spiro splines under the GPLv2, which makes it possible to be used by this thesis. The implementation is capable of being used for realtime editing. That guarantees that the curve generation would not be the bottleneck of the application in terms of time consumption. The output of the library is a set of Beziér curves which are easier to handle in regular graphics programs.

Heuristic for Setting the Control Points To find suitable locations for the spiro spline control points, a heuristic was used.
All connection constraints can be expressed by 5 variables: Connection length (d), tangents with angles α and β and the curvature limit κ, as shown in **??**. For the calculation of the control points, the difference between the two angles was defined as $\delta = \alpha - \beta$. It equals the angle between the two tangent vectors.
Finding the first 4 points of the curve is straightforward: The first control point is a corner point at the endpoint of the first element. The second point is a straight-to-curve point, that is positioned an infinitesimally small distance in the direction of the tangent from the first point. This constrains the "outgoing" tangent. Vice versa, the same control points are set at the end of the curve, where the element that is connected with begins. Thus both tangent constrains are fullfilled. Because the spiros do not offer a direct way of setting the maximum and minimum curvatures, zero to two more control points have to be added. The curvature can be limited by positioning a G^2 control point at distance $2 * R$ from the start- and endpoint. This limits the curvature to the left and to the right to be the same. Since the highest curvature between the startpoint and the G^2 point would be a circular arc with radius R this limits the curvature to $\kappa = 1/R$, as long as both G^2 points are keeping a distance of $2 * R$ as well.
The heuristic differentiates 3 cases:

[5] libspiro: `http://www.levien.com/spiro/`

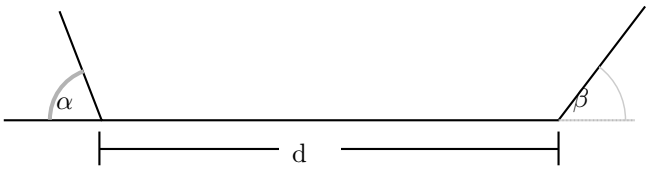


Figure 2.12: Variables for the connection between two elements

If the both tangent vectors α and β have a similar direction and the circle that is described by them and d has a radius that is in the range of $R < r < 2 * R$ then no intermediate G^2 points are set because the connection will already be smooth and optimal.
Otherwise, the G^2 points are set by elongating the tangent vector to $2 * R$ and rotating it by $0.5 * \alpha$. This is done to reduce the size of the resulting curve, which is demonstrated in .. Fig ...
As discussed, the two G^2 points, called c_1 and c_2 are closer than $2 * R$ to each other, the positions have to be recalculated. This happens by rotating the c_1 and c_2 points gradually in different direction until positions are found that conform to the distance constraint. By rotating both points about the same angle, a symmetry is kept.
If the distance d is smaller than

Chapter 3

Implementation

3.1 Input

For the implementation of the presented algorithms an input format had to be specified. As already mentioned in the requirements, it was desired that the input would be easy to edit for artists. That implies that a proper editing tool should be available. To create drawings basically two types of representation are existing: raster images and vector graphics. Raster images work on a *pixel* basis. Each pixel is saved as respective color value depending on the color space (grayscale, indexed or RGB are common). Lossy compression for raster images is available. Scaling raster graphics is limited by the resolution the image was saved in.

In contrast, vector graphics have infinite resolution because they define the image in terms of mathematical functions. Vector graphics contain information about seperate elements where raster graphics do only contain color information and don't have any concept of "element". That makes it easy to retrieve and store metadata and, for example, parse the color of a line or area.

Due to the mentioned points it was easy to decide that vector graphics will be used as input for the path generator program. Several excellent editors are existing, such as Inkscape, which was already mentioned in the beginning, or Adobe® Illustrator®, which is a graphics industry standard.

Since vector graphics is only a concept, a file format has to be specified as well. A number of proprietary formats, like the Adobe® Illustrator® format (`.ai`) are available, but because of their proprietary nature they have been ruled out. The most widely supported format for vector graphics is the *Scalable Vector Graphics* (SVG) specification, which is an open standard format, defined by the SVG Working Group[19]. Rendering SVG is supported in all major web browsers and most modern user interface toolkits such as QT or GTK+. SVG is based on the *Extended Markup Language* (XML), for which a number of proprietary and open source parsers are available, thus making it easy to build upon.

3.2 SVG Parser

The main part of the SVG specification that we needed to implement is the path parsing. A path, in SVG, is represented by a sequence of commands. A command is a lower- or uppercase letter followed by a list of coordinates [1]. Coordinates following uppercase letters indicate *absolute* coordinates and lowercase letter are followed by *relative* coordinates. It is stateful in the way that for correct parsing the previous coordinate has to be known, except for the `M` (Move To) command.
A not complete list of possible commands is listed below:

`[M/m] (x,y)+` Move To command. Starts a new (sub-)path at x,y.

`[L/l] (x,y)+` Line To command. Draws a straight line from the previous coordinate to the current coordinate.

`[C/c] (x1, y1, x2, y2, x3, y3)+` Curve To command. Defines a cubic Beziér curve with 4 control points.

`Z` Closes the element.

The SVG parser is implemented in Python. After an initial attempt to use an implementation by the author, it reached a certain limit of complexity that was needed to deal with output produced by Inkscape, and a rewrite would have been imminent. The implementation did not regard element-wise transforms nor to inheritance of group transforms, where a parsing tree structure would be needed. Luckily, a good implementation of all necessary features for the path generation was found online, called "svg" [2].
It was enhanced by an attribute parser and some smaller changes to the inner workings were made to produce the necessary output. The SVG parser also has functions to reduce Beziér splines to a polyline by evaluating the polynomial at certain, linearly distributed intervals of $t \in [0, 1]$. The resulting points are not equally spaced, which is a property of Beziér curves. The points are, however, more densly spaced in regions where the curvature is higher and that is actually positive as the region with the most change gets the most "attention".
A wrapper script was created that exports two functions to the main path generator and passes the parsed objects.

3.3 Element Polymorphism

An element container class exists from which three different classes are derived (corresponding to the three defined image elements). The classes are `PolyLineElementPtr`, `PolygonElementPtr` and `FilledPolygonElementPtr`.
Some common properties are:

[1] `http://www.w3.org/TR/SVG/paths.html`
[2] `https://github.com/cjlano/svg`

`int getSize()` Returns the size of the container holding the vertices.

`Point_2 getFromIndex(int idx)` Returns the point at index. This feature is implemented as circulator by taking the modulo: `return element[idx mod element.size()];`. It is useful for accessing the polyon elements as well as the end of a polyline, which can be accessed by calling the function with `idx = -1;`.

`ElementPtr * to, from` see below.

`Point_2 entryPoint, exitPoint` Those two items keep track of the tour through the drawing. The pointers `from` and `to` keep track of the previous and next element. A complete tour is thus saved as a doubly linked list that spans all elements. This structure makes it easy to iterate through the tour. The `exitPoint` and `entryPoint` are the points on the element where the tour is entering and leaving.

`TourConnector enforcedConnection[2]` An element can have enforced connections to at most two other elements. The tour connector class stores the exit- or entry node (it is not known in which direction a polyline is traversed) as well as the target element.

`int enforcedStartIndex` Stores an optional enforced start index if the traversal of the element should be started at that this index.

`int getType();` returns element type (can be one of `EL_POLYLINE`, `EL_POLYGON`, `EL_FILLED_POLYGON`, which are defined in a enumeration).

3.4 Tree Container

All elements are sorted into a tree structure. The tree structure was chosen to keep track of the containment information.

3.5 Preprocessing

In the preprocessing phase of the program, the bounding box of all elements is calculated to scale and translate them to the actual canvas size which can be arbitrarily set. An optional margin can be added to the canvas to mitigate the risk of connection lines going outside the canvas and thereby loosing localization or driving into the spectators.

3.6 Implementation of the Algorithms

All core geometric algorithms have been provided by the *Computer Geometric Algorithms Library* (CGAL)[20]. CGAL operates with kernels, which implement the

common types, such as Point_2, Line_2, Segment_2, Vector_2, Polygon_2. Since exact constructions are not necessary for our goals, the "exact predicates inexact constructions kernel" is used throughout the application, which stores all coordinates as double-precision floating-point values (and is thereby limiting the exactness of constructions). The functions that are used the most in this implementation are `CGAL::intersect`, `CGAL::transform` and `CGAL::squared_distance`. Vector algebra is also simple thanks to operator overloading.
CGAL contains algorithms for the straight skeleton methods[21] as well as for the convex decomposition[22].

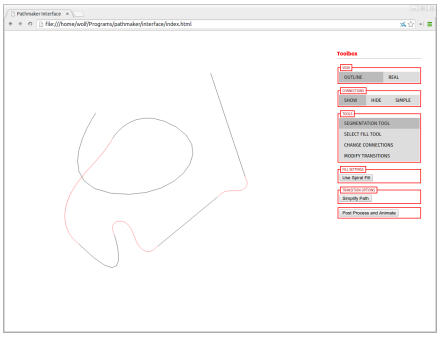
3.7 Post Processing

After the complete trajectory for the drawing process was evaluated and, eventually modified, in the graphical user interface, the post processing step does two enhancements.
First, the edges of the trajectory are rounded. The fill elements are receiving an inner rounding, whereas the outer elements (polygon boundaries) are rounded by analysing a threshold angle. If the angle is too sharp, the trajectory will create an outer loop so that the edges will stay sharp (illustrated in **??**. Otherwise, the rounding is also on the inner side. In the next step, all connecting curves are discretized to polylines and the complete trajectory is equally spaced with points and corresponding rake states, so that it can easily be handled by the *Path Converter* or directly be passed to the point following controller of the BeachBot.

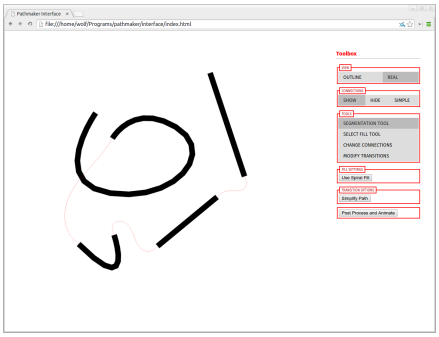
3.8 User Interface

The graphical user interface (GUI), that runs in a regular web browser, was implemented in *JavaScript*, or, more specifically in *CoffeeScript*, which compiles to regular JavaScript. A screenshot of the user interface is shown in Figure 3.1. The *paper.js*[3] library was used for drawing and provides most of the abstractions used throughout the program. A *Python* server is used as slim host of the main application. The application provides means to parse and send JSON serializations of the complete vector element tree, which are parsed in the JavaScript user interface. All data is sent and retrieved using asynchronous `GET` and `POST` requests (commonly referred to as *AJAX*). This implementation was chosen to allow for great flexibility in terms of devices accessing the server. A separate bachelor thesis was working on a Android tablet based touch interface to control the BeachBot. Since Android tablets are perfectly capable of including a *WebView* component which could easily serve the *HTML* and JavaScript files developed in this bachelor thesis, integration of those two tools would be relatively easy. The main program also compiles on the internal computer of the BeachBot, where the server could possibly run, allowing for a fully integrated system. As mentioned in section 3.3, all elements have a

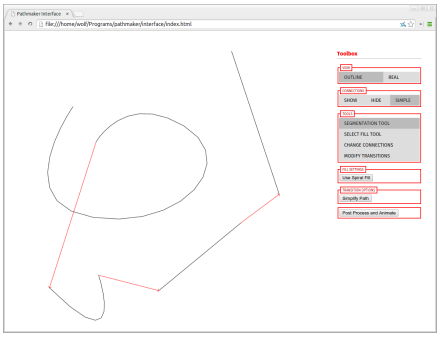
[3] `http://paperjs.org/`



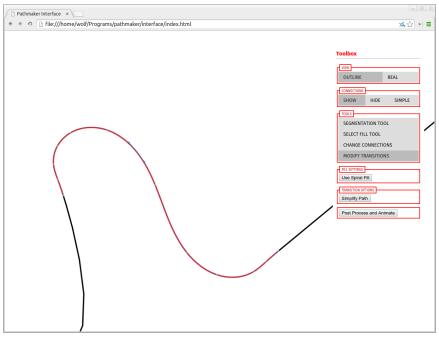
(a) The user interface with Outline View activated



(b) Realistic View activated



(c) Showing connections as arrows



(d) Editing the connection curve

Figure 3.1: Screenshots of the graphical user interface.

unique identifier which makes it easy to select them in the element container and what is used by several of the user interface methods.
Three different view modes and four tools have been implemented which have proven to be helpful in the design of useful trajectories for the BeachBot.

3.8.1 Views

The *Outline* view (Figure 3.1a) is the standard view, where all paths of the image are displayed with a stroke width of 2 pixels. The *Realistic* view (Figure 3.1b) tries to approximate the real rake size of the generated trajectory by using a stroke width that corresponds to the real world dimensions. Empirical data shows that the conversion does work reasonably well. The connections can also be displayed differently. If *Show Connections* is selected, all connecting trajectories are rendered in red color on the screen. Selecting *Simple Connection* (Figure 3.1c), the connections are rendered as directed arrows, which is useful for changing and analyzing the TSP solution. *No Connections* hides all the connecting trajectories. All view

code is implemented client-side (in JavaScript). The user can zoom in and out of the drawing by scrolling the mouse-wheel up and down, as well as pan the view by pressing and holding the right mouse button and dragging the cursor across the screen.

3.8.2 Tools

Paper.js offers a useful tool abstraction layer, which makes the creation of various tools very flexible. Four different tools have been implemented:

Change Connection Tool As mentioned, the `ElementPtr` can store enforced connections between elements. Selecting the first or last node of a polyline or any node of a polygon element, and subsequently the same on another element adds an enforced connection between those elements at the desired nodes, and also deletes any previous enforced connection between the node and any other element.

Shape Transitions Tool The transitions, as they are a set of cubic Beziér curves, sometimes have to be modified because they cross an area or show otherwise undesired behaviour. While for example the crossing of already drawn areas could also be mitigated in the code, it was not possible to do so in the scope of this bachelor thesis. The shape transitions tool offers a convenient way to so: just like in vector graphics programs the node can be selected, which then also displays its two handles. The handles or the node itself can be moved. However, the rotation of the handles (which influences the continuity of the curve), is kept so that both handles form a straight line (the length of the handle vector is not influenced by an operation on the other one).

Select Fill Tool As discussed, two different fill mechanisms are available: The spiral fill and the back and forth fill. Selecting a convex segment of a partitioned polygon, a vector can be drawn that indictates the desired direction of back-and-forth fill for the selected segment. In the same way, the selected segment can also be chosen to be filled by a spiral by clicking the *Use Spiral Fill* button.

Segmentation Tool If the segmentation of a polygon is not as desired, the segmentation tool can be used to resegment the polygon. Drawing a line from any point to another sends a segmentation request to the server. If any of the filled polygons is intersecting with the drawn line segment, the polygon will be cutted along it, yielding two or more polygons. Afterwards, the optimal convex partitioning is executed on each of the new segments.

Chapter 4

Results

In this chapter, the results of the previously described algorithms will be discussed. Generally, all black lines are drawn with the rake down, and the red lines are the connection paths where the BeachBot is driving with the rake in upper position.

4.1 Verifying LKH Solutions

To verify that the LK heuristic produces viable output and that the tour postprocessing works, two simple test images are presented:

4.2 Line Drawings

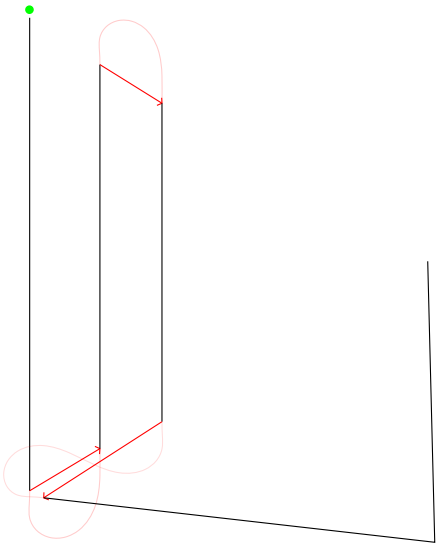
Line drawings consist of no filled areas. These types of drawings are easily created manually and therefore we have used them extensively during testing. The solutions of the manual drawings (with manual connections) are now compared to the solutions generated by the algorithm.

4.2.1 The Lion Drawing

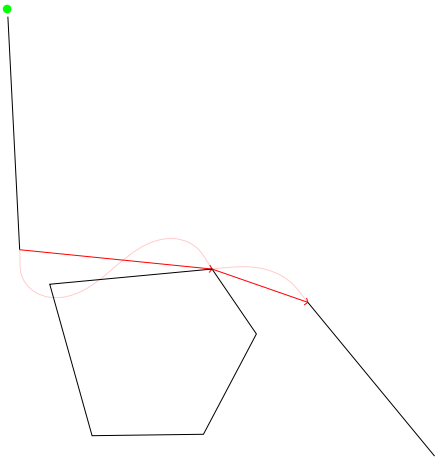
The lion was one of the earliest drawings that the BeachBot was able to reproduce on the beach canvas. It is also available as part of the distributed source of the application (`assets/lion_example.svg`).

4.2.2 The Shark Drawing

Another line drawing, with a closed polygon, is the “Danger, Sharks” drawing.



(a) The red arrows show the tour: The nearest neighbour is not chosen and a more global optimum is obtained.



(b) Again, not the nearest vertex on the polygon is chosen but rather a vertex where the global tour distance is minimized.

4.2.3 Fonts

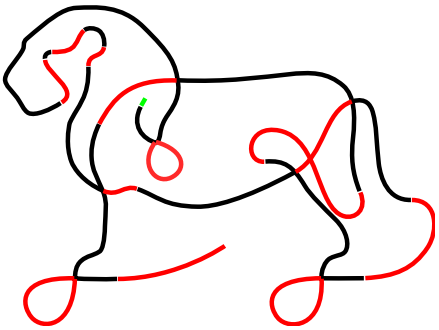
Inkscape has an extension called Hershey Text[1], which offers a variety of engraving fonts that have been designed by Hershey in 1967 [23]. While designed for the purpose of being engraved to metal or stone, they are equally well suited to be drawn on sand. There are variants of each font, from single line to three lines per letter. As of now, only the single line variant has been tested with the path generator. For the single line variant two different styles are available: script and sans-serif.

4.3 Drawings with Filled Areas

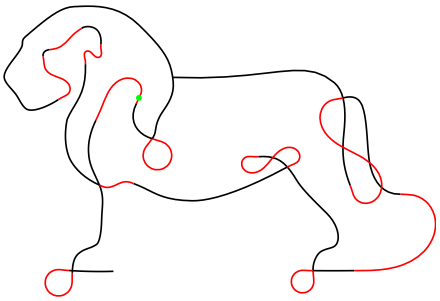
4.3.1 ASL Logo

4.3.2 A *Maleficient* Character

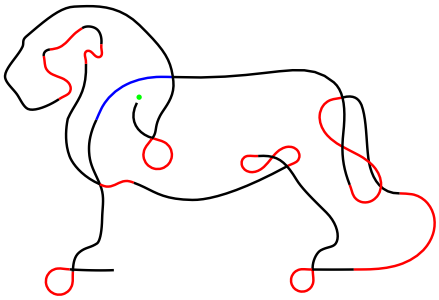
[1] `http://www.evilmadscientist.com/2011/hershey-text-an-inkscape-extension-for-engraving-fonts/`



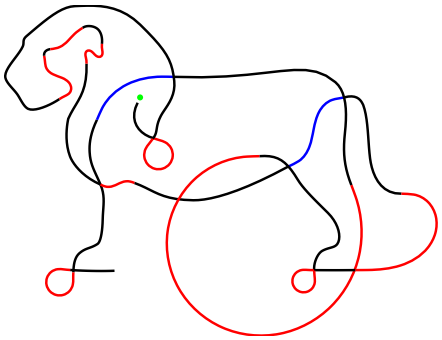
(a) Reference lion drawing created manually



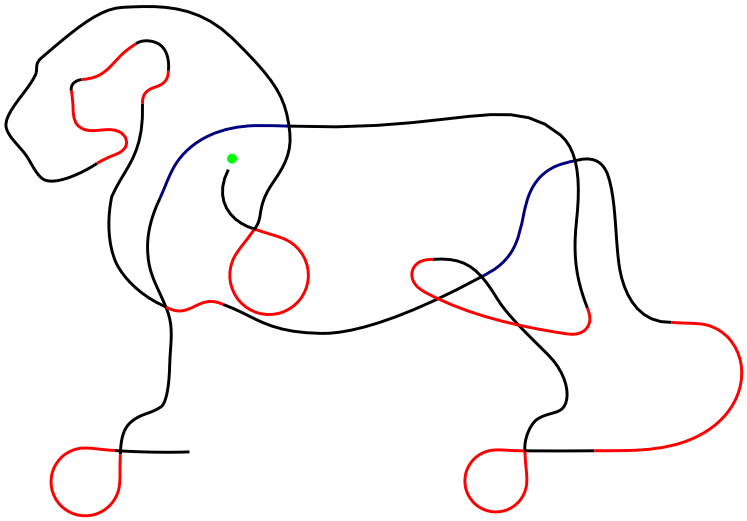
(b) Generated by the generator algorithm



(c) Generated, with enforcement of one connection (blue).

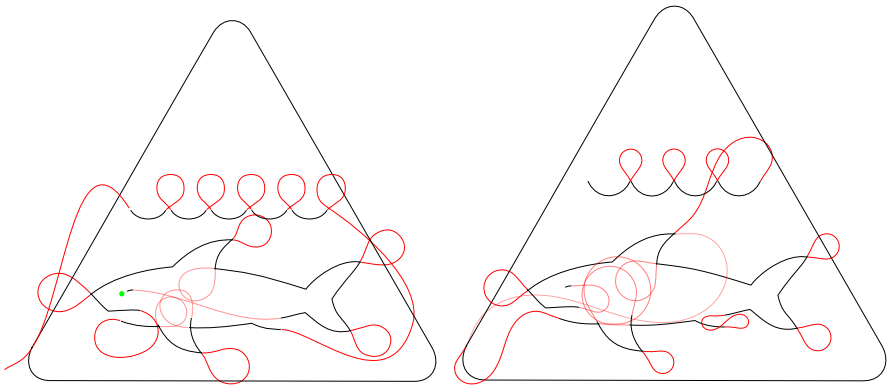


(d) Generated, with two enforced connections. With the unimproved spiro control point heuristic, there is one larger circular connection visible.



(e) With the improved spiro control point heuristic, the huge circular connection is not present anymore ("curvature" limit is also slightly higher). The connection between neck and ear is also smoother, now.

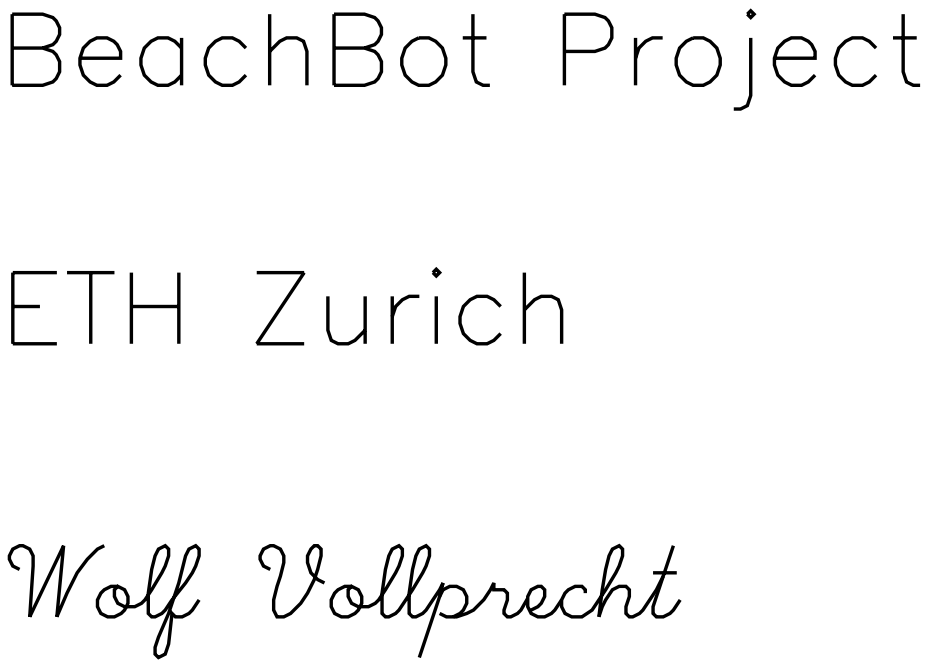
Figure 4.2: Comparison of manual and automatically generated lion trajectory



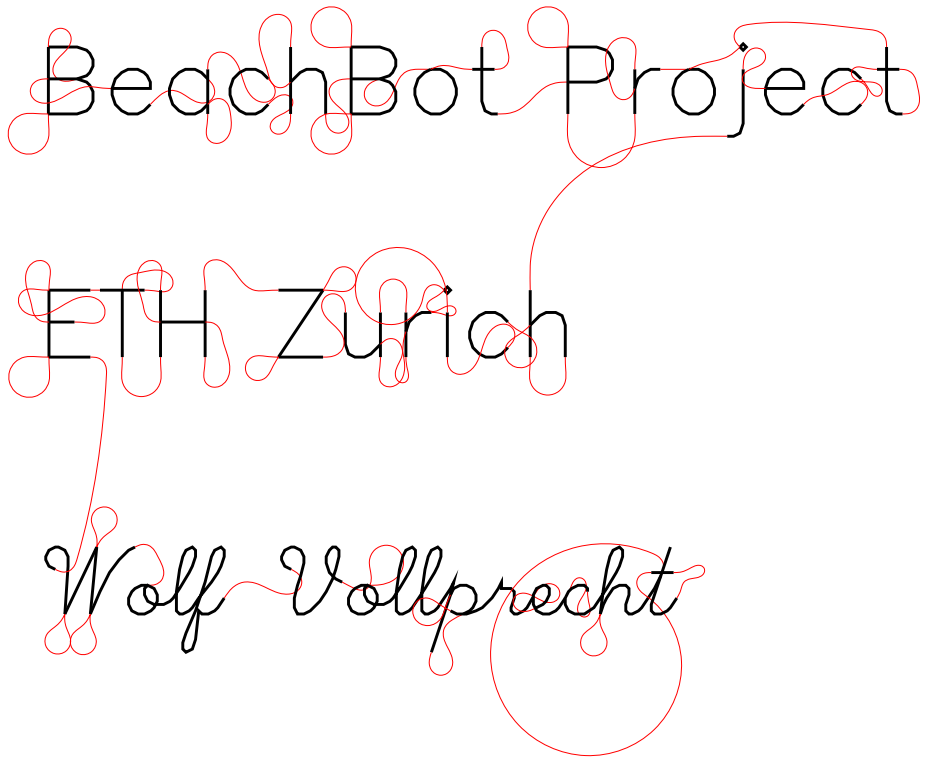
(a) The manually created reference image.

(b) Generated by the path generator: Note how the "outer" polygon is visited previous to finishing the inner drawing.

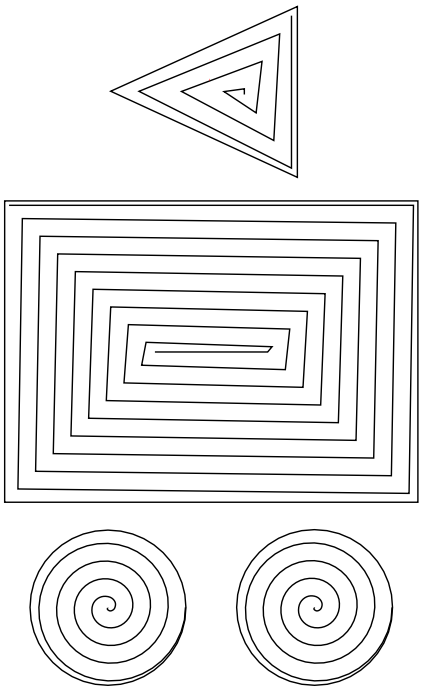
Figure 4.3: Comparison of manual and automatically generated *Danger, Shark* trajectory



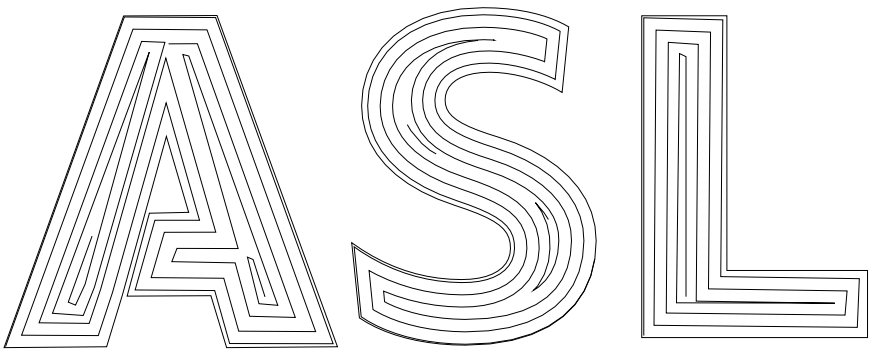
(a) Example text set with sans-serif and script font.



(b) The connected drawing.



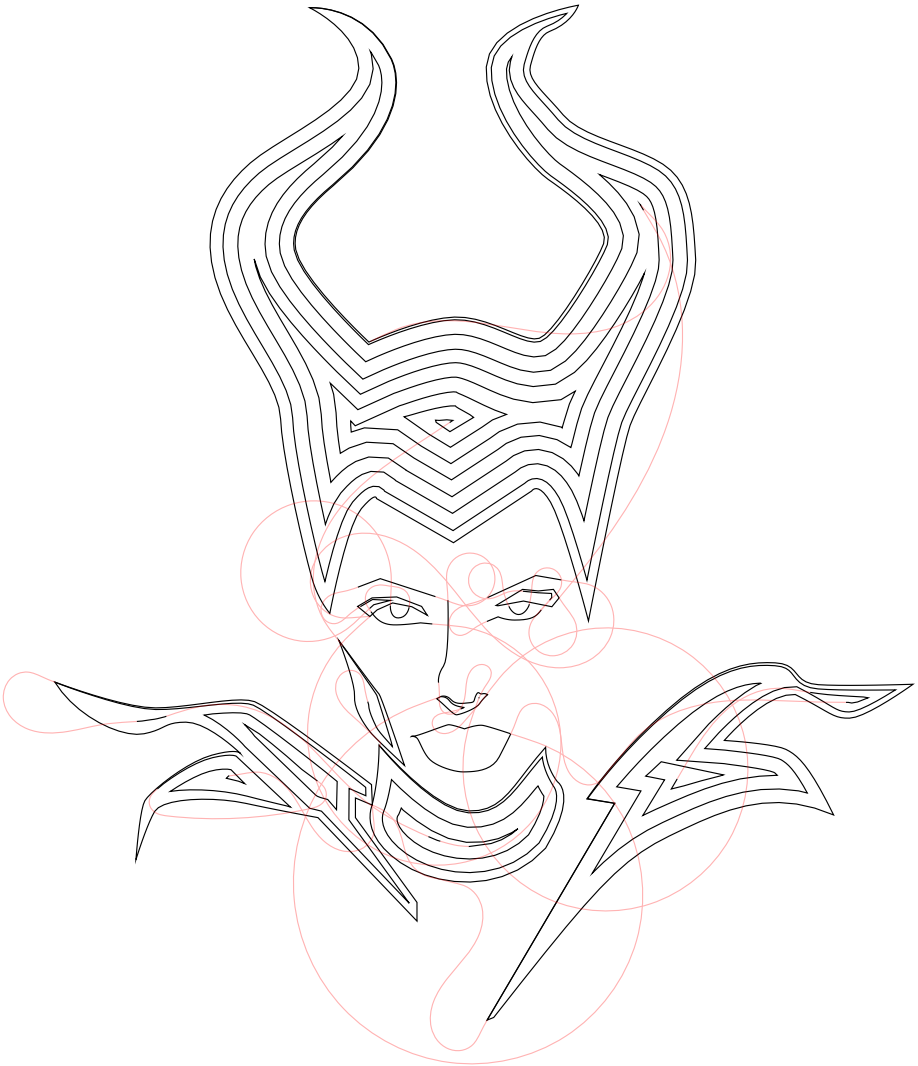
(a) ASL Logo areas filled by generated spirals.



(b) ASL written in a highly concave font.

Figure 4.5: Comparison of manual and automatically generated *Danger, Shark* trajectory

(a) Source bitmap image and manually traced vector graphics (right). All rights to the *Maleficent* character and the image reserved by ©Disney.



(b) Generated trajectory. Note that the mouth is not filled due to a small error in the traced vector graphics.

Figure 4.6: A drawing depicting the main character of the upcoming *Maleficent* (©Disney) movie. First, a vector graphic has to be manually created, using a bitmap as reference. Based on the vector graphic, a trajectory can be generated.

Chapter 5

Conclusion

A collection of algorithms was presented which make creating an drawing for a mobile autonomous drawing robot an easy task. While no real world tests have been performed with trajectories generated only by the presented algorithms, comparing the result of the manually connected Lion drawing to the generated one the quality of the path is very much comparable.

5.1 Outlook

Of course, the problem is not solved to perfection yet. Several things can be improved and ideas are presented in the following section:

Driving over filled areas The trackmarks are a serious problem for the drawing. While the connection paths can be manually adjusted to not cross through any already filled area, this is tedious and should be automatized.

- After an initial run of the TSP algorithm, the solution is examined. If a filled area is crossed over, the specific edge in the distance matrix gets a high weight assigned, whereby the connection should not be included when the TSP is solved again. This process is repeated until a solution is found which minimizes the crossings of areas.

- The intersections of the curved connection paths with already filled areas can easily be found by looking at the convex hulls of the Beziér curve control points that make up the connecting curve (a property of the Beziér curves is that they are always enclosed in the convex hull of the control points).

  If an intersection is evident, the polygon should get an offset which can be calculated by building the Minkowski sum of the polygon and a disk shape of some specified radius. The offset polygon would be cutted and the appropriate part of that polygon would be added to the curve trajectory (as shown in **??**).

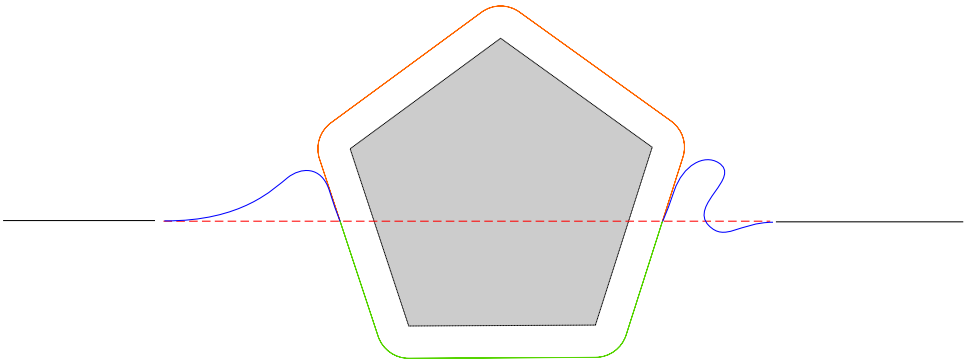


Figure 5.1: Using the Minkowski sum to offset the polygon and drive around the filled area (gray). The simple, straight connection is shown in red, the blue connection is the optimized, non-crossing connection.

- Another possibilty would be to add more G^2 spiro control points along the offset polygon which would have to maintain the same minimum distance to each other as defined in the section about smooth connection lines.

Appendix A

Appendix

A.1 Installation

The installation procedure for the installation of the path generator program under Ubuntu 14.04 is described below:

1. For running the application, the following dependencies have to be satisfied: `libcgal-dev, libboost1.54.0-all-dev, python2.7, libpython2.7-dev, build-essential, libeigen-dev, libjsoncpp-dev`. If the QT frontend should be installed, `libqt4-dev` has to be available. To be able to run the python server, the `flask` python library needs to be installed.
2. Cloning the source code from `https://github.com/asl-beachbot/pathmaker`: `git clone https://github.com/asl-beachbot/pathmaker`.
3. Running `cmake` with configuration options: `-DNOGUI=[ON/(OFF)]`, `-D32BIT=[ON/(OFF)]`. Round brackets indicate default value.
4. If `cmake` was run successfully, `make` can produce either the python module by using the target `python/beachbot_pathgen` or the standalone target, called `svg_parser` by calling e.g. `make svg_parser -j3`.
5. If python version was built, starting the python server by calling `python python/server.py`.
   If standalone version was built, `./svg_parser` will launch the standalone program.

Command Line Flags Command line flags can either be set as flags when the program is executed or can be set in the config file (the default file is `config.cfg` and `pythoncfg.cfg` for the server). The list of allowed options is: `-h [ --help ] produce help message`
`-f [ --filename ] arg SVG File for parsing`
`-r [ --round_radius ] arg set radius for corner rounding`
`-m [ --fill_method ] arg set fill method (1: wiggle or 2: spiral)`
`-s [ --scale_for_disp ] arg scale for display`
`--angle_step arg Interpolation stepsize for rounding`
`(e.g. 0.2 * PI)`
`-m [ --max_interpol_distance ] arg Max distance for points`
`-d [ --display ] Open up the QT Window for inspection`
`-t [ --threshold_round_angle ] arg Defines from which angle on it should be`
`rounded (or outer rounded)`
`-l [ --line_distance ] arg Line distance inside filled elements`
`--area_deletion_threshold arg Maximum area of filling elements that will get deleted`
`-c [ --config_file ] arg Use a different config file`
`--segmentation_on arg Turn on or off segmentation`
`--text_export_filename arg Filename for export to textfile`
`--svg_export_filename arg Filename for export to SVG File`
`--field_width arg Width of field`
`--field_height arg Height of field`
`--field_offset arg Offset (margin) of field`
`--segment_offset arg Offset of Segment (from partitioning)`
`--no_tree_ordering Disables ordering of the tree (Useful when manual image from Timo!)`
`--number_segments_bezier_connect arg Define the number of segments for bezier`
`interpolation)`
`--stop_go_outer Round (and outer round) outer contours or stop-turn-go cycle?`
`--round_connection_threshold Threshold for rounding connections (otherwise just place point) [squared length of point distance]`

Bibliography

[1] T. Oksanen and A. Visala, "Coverage path planning algorithms for agricultural field machines," *Journal of Field Robotics*, vol. 26, no. 8, pp. 651–668, 2009.

[2] J.-H. Kao and F. B. Prinz, "Optimal motion planning for deposition in layered manufacturing," in *Proceedings of DETC*, vol. 98, pp. 13–16, 1998.

[3] H. S. M. Coxeter and S. L. Greitzer, *Geometry revisited*. Mathematical Association of America Washington, DC, 1967.

[4] E. W. Weisstein, "Simple polygon. From MathWorld—A Wolfram Web Resource." Last visited on 24th June 2014.

[5] E. W. Weisstein, "Convex polygon. From MathWorld—A Wolfram Web Resource." Last visited on 24th June 2014.

[6] E. W. Weisstein, "Concave polygon. From MathWorld—A Wolfram Web Resource." Last visited on 24th June 2014.

[7] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner, "A novel type of skeleton for polygons," *Journal of Universal Computer Science*, vol. 1, pp. 752–761, dec 1995.

[8] S. Huber and M. Held, "Computing straight skeletons of planar straight-line graphs based on motorcycle graphs.," in *CCCG*, pp. 187–190, 2010.

[9] D. H. Greene, "The decomposition of polygons into convex parts," *Computational Geometry*, vol. 1, pp. 235–259, 1983.

[10] K. Helsgaun, "An effective implementation of the lin–kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[11] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[12] K. Hornik and B. Grün, "Tsp-infrastructure for the traveling salesperson problem," *Journal of Statistical Software*, vol. 23, no. 2, pp. 1–21, 2007.

[13] R. Kumar and H. Li, "On asymmetric tsp: Transformation to symmetric tsp and performance bound,"

[14] K. Helsgaun, "Solving the equality generalized traveling salesman problem using the lin-kernighan-helsgaun algorithm," 2013.

[15] K. Helsgaun, "Solving the clustered traveling salesman problem using the lin-kernighan-helsgaun algorithm," tech. rep., 2014.

[16] J. Roulier and T. Rando, *5. Measures of Fairness for Curves and Surfaces*, ch. 5, pp. 75–122.

[17] J.-W. Choi, R. Curry, and G. Elkaim, "Piecewise bezier curves path planning with continuous curvature constraint for autonomous driving," in *Machine Learning and Systems Engineering*, pp. 31–45, Springer, 2010.

[18] R. L. Levien, "From spiral to spline: Optimal techniques in interactive curve design," 2009.

[19] SVG Working Group, "Scalable vector graphics (svg) 1.1 (second edition)," 2011.

[20] The CGAL Project, *CGAL User and Reference Manual*. CGAL Editorial Board, 4.4 ed., 2000.

[21] F. Cacciola, "2D straight skeleton and polygon offsetting," in *CGAL User and Reference Manual*, CGAL Editorial Board, 4.4 ed., 2014.

[22] S. Hert, "2D polygon partitioning," in *CGAL User and Reference Manual*, CGAL Editorial Board, 4.4 ed., 2000.

[23] A. V. Hershey, "Calligraphy for computers.," tech. rep., DTIC Document, 1967.