Chapter 1

Introduction

1.1 The BeachBot Project

The BeachBot project was a "focus project" at ETH Zurich. During the last two semesters of the bachelor studies the team had the opportunity to develop a mobile and autonomous robot for creating sand drawings on beaches. In total 7 mechanical engineering students, one electrical engineering student and two industrial design students (from the Zürcher Hochschule der Künste) where working on the project. The result of the project is a 3 wheeled mobile robot that can drive autonomously. The key features are:

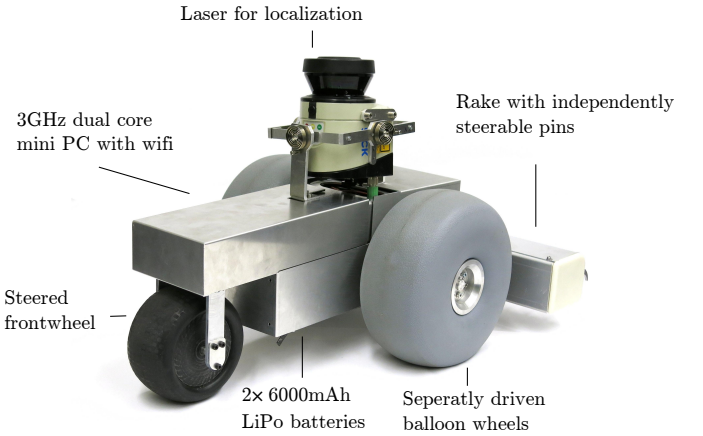
Localization The robot is able to reliably localize itself on the beach, using a laser range finder and 3 or more reflective poles. An localization accuracy of about 3 centimetres was achieved.

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 in a way that it follows a pre-defined trajectory. The trajectory generally is a text file with coordinates.

The robot was tested successfully 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

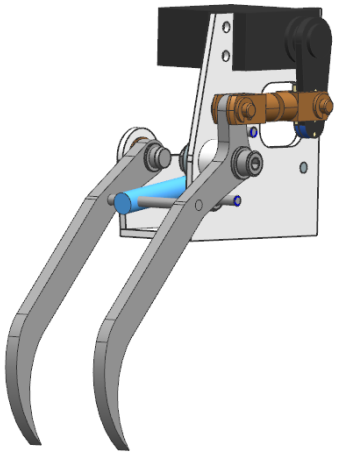


Figure 1.1: Various images of the BeachBot

Chapter 2

Requirements and Inspiration

The BeachBot project itself was inspired by the images of sand artists like Peter Donnelly and Andres Amador, who create large scale sand art at beaches using a rake. Some of the imagery that we found online can be seen in Figure 2.1.
Since our goal was to create drawings similar to those of the artists, we derived our requirements from these sample images.
First of all, and probably the easiest, the BeachBot should be able to draw lines. But even more important is the support of filled areas. While driving lines or curves is straight forward, there is no easy solution how to derive the path for filled areas. Not only should the area be covered to the highest extent, the drawing process should look interesting and artistic to spectators as well (unlike, for example, a printer).
Derived from the pictures is also the requirement that the BeachBot should only be able to draw in two colors. Gradients or differently colored areas are not needed.
Crossing lines or filled areas should be prohibited as good as possible since the balloon wheels and also the front wheels 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. 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. Therefore the requirement to be able to use a modern graphics editor tool to work with was set. To conform to this requirement we soon decided to use *Inkscape*[1], a popular open source vector graphics editor, as our input creation tool of choice. Further discussion about vector graphics as input and the steps to integrate a standardized vector graphics format into the path generator are

[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 2.1: Various beach drawings by artists

explained in **??** and **??** respectively.
During the testing phase it was found out, that some of the generated paths needed some extra care by humans. 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 sections it will be shown how these requirements were fullfilled and to which extent.

Chapter 3

Path Planning Algorithms

3.1 Algorithm Overview

The output of the path generator should be a single trajectory that completely connects and covers all elements of the drawing.
This 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 tries to minimize the traveled distance by applying a traveling salesman heuristic. As last step, connections with limited curvature are generated to complete the trajectory.
Each of those steps will be discussed in detail in this section.

3.2 Image Structure

Derived from the requirements, three different elements were identified as part of drawing (also shown in Figure 3.1:

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. So that vertex v_{n+1} equals v_0.

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

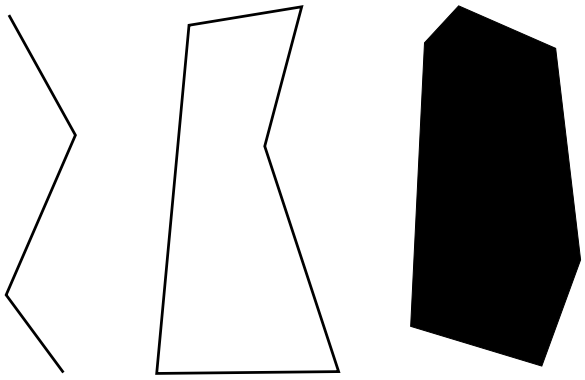


Figure 3.1: Polyline, polygon and filled polygon

3.3 Polygon Filling

3.3.1 Related Work

Over time several surface coverage algorithms have been developed, and some distinctions can be made.
Coverage algorithms exist for applications like autonomous vaccuum cleaners or autonomous lawn mowers, but also search and rescue robotic applications usually and they usually 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 computing intensive, what might be justified when using huge 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*. 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 go over the machined surface without problem because they can lift the machnining head – something that is not possible with the autonomous ground robot.
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...

3.3.2 Spiral Filling

Straight Skeleton

To generate a spiral fill for the trajectory, we use the properties of the so called straight skeleton. The straight skeleton is defined as the topolgical skeleton of a polygon that is created by moving the edges inwards at a constant speed and observing the intersections of the vertices. It was first described by [3].
During the creation of the straight skeleton, two events can happen:

- Edge event: An edge shrinks to zero, making its neighboring edges adjacent now.
- Split event: An edge is split, i.e., a reflex vertex runs into this edge, thus splitting the whole polygon. New adjacencies occur between the split edge and each of the two edges incident to the reflex vertex.

(cited from [3]).
The straight skeleton 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.
For the case of this thesis, the most interesting property of the straight skeleton is the ability to easily obtain inset polygons. The inset polygons are used to create a spiral fill for the polygons. The procedure is as follows:

(0) The straight skeleton is generated

(1) The inset polygon is created

(2) 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 our current position). All the others are recursively filled in the same way. Additionally, a starting point to the spiral is passed, so that the beginning of the next spiral will be close to the current one. (The procedure restarts with the split polygon as input at (0)).

(3) The $n + 1$ vertex of the inset polygon is appended to the current spiral

(4) An inset polygon is generated with inset length divided by number of vertices. Through dividing the inset length (which is predefined, usually as one rake width) by the number of vertices in the polygon, we make sure that one revolution of the spiral will travel only one rake distance.

Algorithm 1 Spiral Filling

```
function createSpiralFill(Polygon p, Point startPoint)
    ss ← getStraightSkeleton(p)
    lOffset = RakeWidth/p.size()
    if startPoint then currPoint ← startPoint
    else currPoint ← p[0]
    end if
    while insetPolys.size() ≥ 1 do
        if insetPolys.size() > 1 then
            closestPoly = searchClosestPolygon(insetPolys, currPoint)
            for all {poly ∈ insetPolys|poly ∉ closestPoly} do
                createSpiralFill(poly)
            end for
            ss ← getStraightSkeleton(closestPoly)
            lOffset = RakeWidth/closestPoly.size()
            insetPolys ← getOffsetPolygons(ss, lOffset)
        end if
        index ← findClosestIndex(currPoint, insetPolys[0])
        currPoint ← insetPolys[0].fromIndex(index + 1)
        result.append(currPoint)
        insetPolys ← getOffsetPolygons(ss, lOffset)
    end while
end function
```

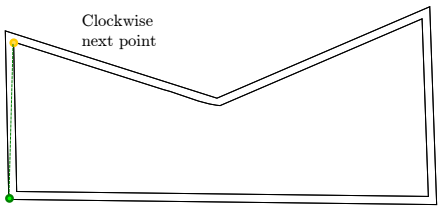
The algorithm is both displayed in pseudo code and graphically.

3.3.3 Back and Forth Filling

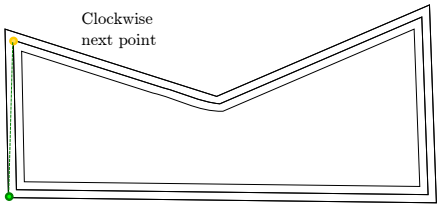
This fill method is quite straight forward: The polygon is filled by a number of lines that are parallel to each other and clipped at the polygon edges. The procedure takes an direction as input. First, a starting point is searched. If the inclination of the direction is bigger than 45°, then the leftmost vertex, whereas if the inclination is smaller, the bottommost vertex is selected.

Starting from the start point, lines are positioned with the specified direction and the polygon is tested for intersection. If two intersections for a given line are found, a line segment with those two points is added to the resulting trajectory.

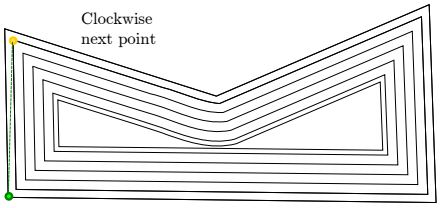
This method works very well for convex polygons, but has drawbacks for non-convex polygons, since a non convex polygon can have more than two intersections for any given line. Therefore it requires the polygon to be decomposed into convex parts. This happens through optimal convex partitioning with the dynamic programming algorithm by Greene. The produced result is a set of convex polygons of maximum size, which is good for the purpose of filling the shape.



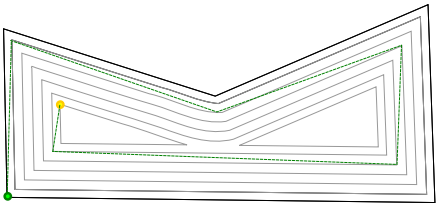
(a) The first inset polygon and the first part of the spiral line



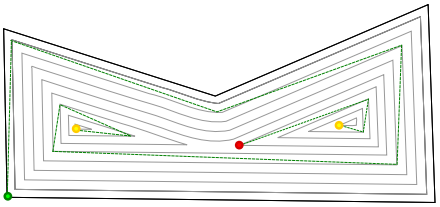
(b) Next inset polygon generated



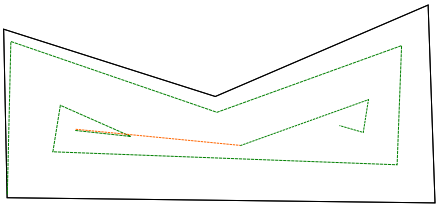
(c) The polygon right before the *split* happens



(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 3.2: Generating the fill spiral. Note that this is only an example for illustration purposes. Of course, the density of inset polygons for a real application is much higher.

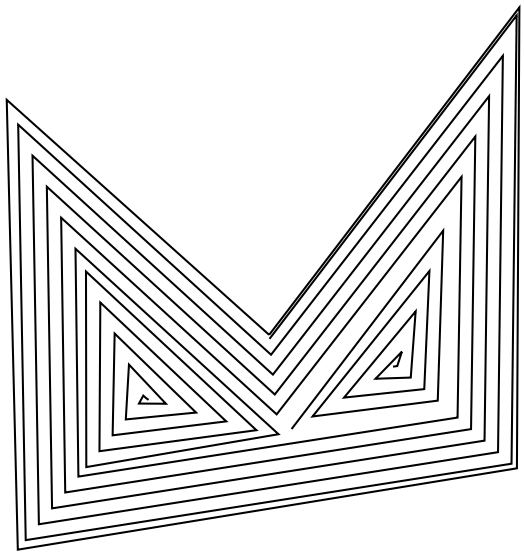


Figure 3.3: Generated spiral

3.4 Path Generation

As already mentioned, to finally create a drawing all elements have to be connected by support trajectories where the rake is lifted. The supporting paths should have two properties: The total sum 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 types: Lines, polygons and filled polygons. Lines and polygons differ in the way they can be connected, as illustrated in **??**. A line can be entered or exited on both ends and is traversed in a previously unknown direction. A polygon however can be entered at any vertex, but can only be exited at the same so that the polygon is closed. Filled polygons work the same as the regular polygon in that sense, since the spiral and the back and forth filling create a polyline that is inserted into the filled polygon.

3.4.1 Traveling Salesman Problem

The Traveling 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. The Hamiltonian Circuit is defined as a tour through the graph, that travels to every vertex exactly once. It is called salesman problem because in the historical context the vertices were cities and the salesman, seeking to minimize the distance traveled to visit all the cities in a given tour, would have liked to get a solution for this problem. However, one of

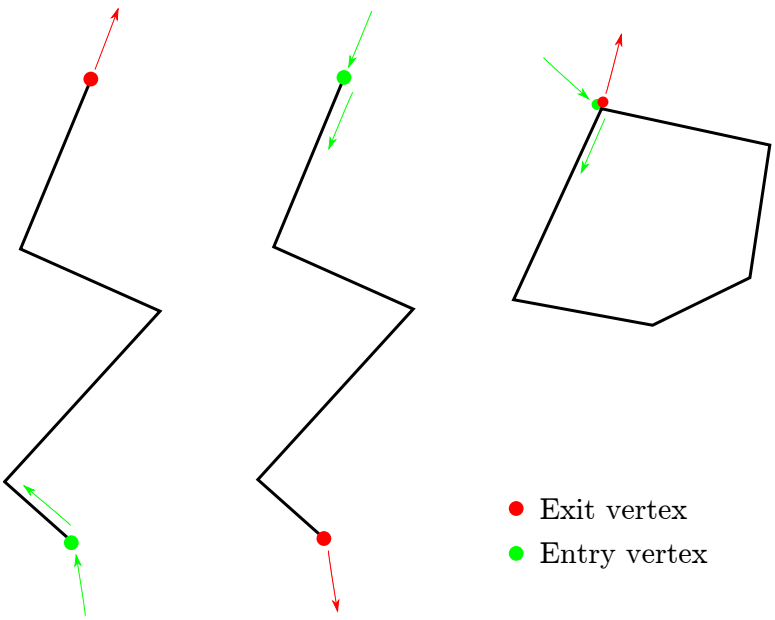


Figure 3.4: Possible entry- and exit vertices and direction of traversal for polylines and polygons. Note that the polygon could also be traversed in clockwise direction.

the properties that make this problem hard is that it is NP-hard. If no heuristic is used, the amount of solutions that have to be searched in order to test all available possibilities is $n!$.
The TSP can be subcategorized in three categories:

Symmetric A TSP is symmetric if all distances between every node pair are symmetric (equal) in both directions from $A \leftrightarrows B$.

Metric In a metric traveling salesman problem, all distances conform to the triangle inequality $c \leq a + b$ which is, for example, true if all distances are calculated using the euclidean distance formula between two points: $\overline{AB} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ but also for the so called Manhattan distance which is defined as $\overline{AB}_M = |x_1 - x_2| + |y_1 - y_2|$.

Asymmetric If the weights are different from $A \to B$ than from $B \to A$ then the TSP is assymetric. Of course, the triangle inequality cannot be fulfilled anymore.

The simplest heuristic to the TSP is the nearest neighbour search. From a given startpoint the nearest, 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 popular heuristic is the branch and bound method. It works by splitting the set of possible solution into two halves and calculating a upper and lower bound for both. If the upper bound of the first is lower than the lower bound of the second, the second is discarded. During the course of this thesis a variant of the depth-first branch and bound search was implemented. First, the graph was traversed until a tour was completed. Afterwards, another tour (in order) was recursively started. As soon as the traveled distance of the entire tour in that branch got higher than the current minimum tour, the branch is discared. 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 seldomly span a tour which distance is higher than the current minimum, and it gets only worse). It must be said, that the implementation was not very mature, and it actually produces an optimal tour in the current implementation, which is not desired in a heuristic. With more level-wise bounds it could have been a lot more efficient.
The final solution to the minimum connection problem was to use a state of the art TSP solver[4], which employs the Lin-Kernighan-Heuristic[5] (LKH) to approximately solve the Traveling Salesman Problem. It is, unlike many other TSP solver implementations that require the triangle inequality to hold, also well suited to solve assymetric problems.

3.4.2 Adaptation of Traveling Salesman Problem for the Algorithm

To find a suited tour with the generic LKH solver, the correct weights for the graph have to be specified. This happens in logical 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. 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 $n-1$ (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 elementwise 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 the 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 beginning. The solution that was found is to enforce a defined start point to be inserted. The outgoing weights of the start point are either euclidean distances or defined to be the same to every node in the graph. If they are euclidean, chances are very high that the starting line is the nearest neighbour. If the outgoing weights are all defined to be 1 the LKH can find the best suited start point on it's own because the startpoint will not induce any preference for any of the nodes.
The ingoing weights for the startpoint are chosen to be equal for all nodes. Thus, the TSP will never optimize towards traveling back to the start point.

3.5 Smooth Line Connections

While not required by the robot kinematics of the BeachBot (which can turn on the spot), generating smooth connections is beneficial to 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 in 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 [6] three approaches to generate beziér splines with monotone curvature are discussed. [7] uses piecewise 3rd degree bezier curves to create a curvature and corridor constrained trajectory. [to be expanded]

Spiro Splines

Spiro splines, introduced by [8] 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 [1] 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

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

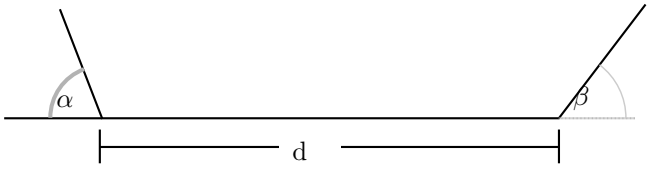


Figure 3.5: Variables for the connection between two elements

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:
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 4

Implementation

4.1 Input

For the implementation of the presented algorithms a 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 basically is a graphics industry standard.

Since vector graphics is only a concept, a file format had to be specified as well. A number of proprietary formats, like the Adobe® Illustrator® format (`.ai`) are available, but because of their proprietary nature 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[9]. 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.

4.2 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 the Python programming language. After an initial attempt to use an implementation by ourselves, it reached a certain limit of complexity that was needed to deal with output produced by Inkscape, and a rewrite would have been imminent. Our implementation did not pay respect to elementwise 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. 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.

4.3 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;`

`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, `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).

They derive a number of virtual and pure virtual functions from the base class

4.4 Tree Container

All elements are sorted into a tree structure. The tree strucure was initially chosen to keep track of all containment

4.5 Preprocessing

In the preprocessing phase of the program, the bounding box of all elements is calculated and they are scaled and translated to the field. An optional margin can be set to the field.

4.6 Implementation of the Algorithms

All core geometric algorithms have been provided by the *Computer Geometric Algorithms Library*[**?**] (CGAL).

4.7 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 different things.
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. x In the last 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* (another bachelor thesis) or directly be passed to the point following controller of the BeachBot, .

4.8 User Interface

The graphical user interface (GUI) was implemented in *Javascript*, or, more specifically in *CoffeeScript*, which is a strict superset of Javascript. The *paper.js* 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 Javascript user interface. All data is sent and retrieved using asynchronous `GET` and `POST` requests (commonly referred to as *AJAX*). This implemetation was chosen to allow for great flexibilty in terms of devices accessing the server. A seperate 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 should be 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 the section about the `ElementPtr`, all elements have a unique identifier number which makes it easy to select them programmatically and what is used by several of the GUIs 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.

4.8.1 Views

The *Outline* view is the standard view, where all paths of the image are displayed with a stroke width of 2 pixels. A *Realistic* view 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 quite well. The connections can also be displayed differently. If *Show Connections* is selected,

all connecting trajectories are rendered in blue color on the screen. Selecting *Simple Connection*, 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 was implemented clientside (in Javascript).

4.8.2 Tools

Paper.js offers a useful tool abstraction, which makes the creation of various tools very flexible. 4 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 number of 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 graphcis programs the node can be selected, which then also shows its handles. The node itself can be moved, or the handles 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 5

Conclusion

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] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] S. W. Group, "Scalable vector graphics (svg) 1.1 (second edition)," 2011.