Name: Geva Kipper **I.D:** 307876219

3D Printing Project

2D Part Orienting

Abstract

The project deals with the problem of orienting a given polygon P without the use of sensors. We focus on a method called *oblivious push-plans*, which means we push P using a straight arm from several different directions, and construct this series of directions such that P will always end up in the same, predetermined orientation, regardless of the orientation in which it is given to us.

The project includes the implementation of several algorithms that construct valid and optimal push-plans for a given polygon P, as well as a simulation program which can be used to demonstrate how the push-plan we obtain operates on different orientations of P.

Theoretical background

The centroid of the polygon P, which takes a major role in the analysis of the push operation, is the center of mass of a uniform-density object in the shape of P. We compute it by the formula:

$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})\,(x_i y_{i+1} - x_{i+1}y_i) \qquad C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})\,(x_i y_{i+1} - x_{i+1}y_i)$$

Where x_i, y_i are the x and y coordinates of the i-th vertex of P, $x_n := x_0$, $y_n := y_0$, and A is the signed area of P, given by:

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1}y_i)$$

The radius function $\rho: [0,2\pi) \to \mathbb{R}$ maps a direction d to the distance from the centroid of P to the ground when P is rotated by d. We can compute the convex hull of P first, and then find the minimum and maximum points of $\rho(d)$ by simultaneously sweeping two sphere arrangements:

1. The arrangement of normals of the edges of P (the 'Gaussian map' of P).
2. The arrangement of directions from the centroid to the vertices of P (the 'central map').

It can be shown that a local minimum of the radius function corresponds to an edge of the convex hull, $CH(P)$, and that when pushed from some direction d, P eventually stops rotating, and one of the edges of $CH(P)$ is aligned to the pushing arm. We say that P rests on this edge of $CH(P)$.

The push function $p(d)$: $[0,2\pi) \to [0,2\pi)$ maps a direction d to the normal of the edge of $CH(P)$ that P rests on when pushed from direction d (in the rotated coordinate system of P).
The function $p(d)$ is constant between consecutive local maxima of the radius function, and its value is the normal of the edge corresponding to the local minimum between them.

Algorithms implemented

The project includes the implementation of the following algorithms:

1) An algorithm by Chen and Ierardi [2], which finds a push-plan of length at most n, and runs in time $O(n \log n)$ for general polygons and $O(n)$ for convex polygons.
2) An algorithm suggested by Goldberg, described in [3], which finds an *optimal* push-plan (one with minimal length) in $O(n^4)$ time.
3) An improved version of the above algorithm, whose running time is $O(n^2 \log n)$.

We provide brief reviews of the implemented algorithms. For the full details and analysis of the algorithms see the bibliography section below.

1) Chen-Ierardi algorithm

The algorithm of Chen and Ierardi finds a push-plan whose length is at most n.[1] It works by constructing the push function of P, and finding the largest and second-largest half-angles in the push function of P, marked by α and β in the figure below. We then choose an angle $x \in [\beta, \alpha]$. As Chen and Ierardi show, if we push the polygon from angles $0, x, 2x, \ldots, (n-1)x$, it must end up on the stable orientation corresponding to the angle α. Finding α and β takes $O(n \log n)$ time for general polygons and $O(n)$ for convex polygons.

2) Goldberg's algorithm

Goldberg's algorithm [3] constructs an *optimal* push-plan. That is, one with minimal length. It works by constructing a graph $G = (V, E)$, where the vertices V are the n^2 intervals between pairs of stable orientations of P. We connect two intervals $I, J \in V$ by an edge if, knowing that the polygon lies on one of the stable orientations in I, there is some direction from which we can push P that will guarantee it ends up in some stable orientation in J. This can be determined in $O(1)$ time.

In this graph G we then find the shortest path, of length k, from the 'full' interval, containing all stable orientations, to a singleton interval. By the construction we can choose a series of push directions of length k which guarantees that P always ends up in the same orientation.

3) Improved version

The construction of the interval graph requires going over all pairs of intervals I, J, which takes $O(n^4)$ time, but in fact we only need to consider the edge from I to the shortest interval J that can be reached from I by a single push. This allows us to make the algorithm more efficient, and Goldberg [1] described a version of the algorithm that runs in $O(n^2)$ time. This project also includes a **nearly-optimal implementation** with a time complexity of $O(n^2 \log n)$.

Bibliography

[1] Goldberg, K. Y. (1993). Orienting polygonal parts without sensors. *Algorithmica*, *10*(2), 201-225.

[2] Chen, Y. B., & Ierardi, D. J. (1995). The complexity of oblivious plans for orienting and distinguishing polygonal parts. *Algorithmica*, *14*(5), 367-397.

[3] van der Stappen, A. F., Berretty, R. P., Goldberg, K., & Overmars, M. H. (2002). Geometry and part feeding. In *Sensor Based Intelligent Robots* (pp. 259-281). Springer, Berlin, Heidelberg.

[1] Longer plans are required to support polygons for which the largest half-angle in the push function is non-unique, but their length is still O(n), and this was not implemented in the project.

Implementation considerations

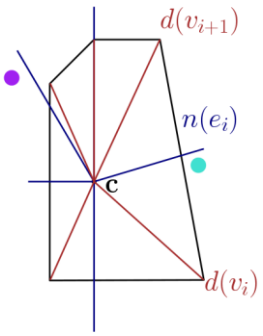
The algorithms in this project were implemented in C++, making use of CGAL, and they only use exact computation. In this section we describe some of the code elements that were implemented for the algorithms. The algorithms support convex as well as non-convex polygons.
Additional information can be found in the documentation within the code.

SphereArrangement_1: This class was written to represent a general arrangement on the one-dimensional sphere S^1. The underlying Vertex type is templated by the Direction_2 type, describing the geometry of the vertices. It also supports extending the vertices with data of a template type. An overlay function was implemented, allowing us to overlay two sphere arrangements in linear time.

Angle_2: This class was written to represent an exact angle between two directions. All angle operations and comparisons are performed in an exact manner using CGAL. It has convenient functions like comparison operators and a method to select an arbitrary direction within the given angle. Exact comparisons, for example, are required by the Chen-Ierardi algorithm, where we need to find the largest and second largest half-angles of the push function.

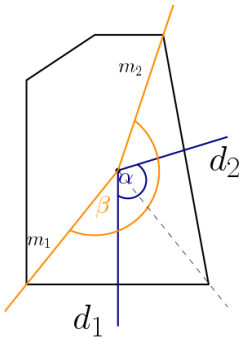
Sphere functions: Functions on S^1 are represented as arrangements on the sphere. Every vertex v is assigned a data value indicating the value of the function on the edge that begins at v and ends at the next vertex u. The Gaussian map, direction map, radius function and push function are all maintained using this structure.

Constructing the push function: We first overlay the **Gaussian map** and the **central map** of P.



Not every edge-normal constitutes a local minimum in the radius function. For example, the normal marked in **purple** does not – if the polygon is pushed on the corresponding edge, it will not be stable and will fall over. The push function steps, such as the normal marked in **cyan**, correspond to edges e_i such that the normal $n(e_i)$ is located in the overlay immediately between the direction vectors from the centroid of P to the vertices of e_i. In this case, $n(e_i)$ is immediately between $d(v_i)$ and $d(v_{i+1})$ and thus $n(e_i)$ is a minimum point for the radius function and constitutes a step for the push function of P.

Identifiying edges of the interval graph: To implement Golberg's algorithm, we need to know which pairs of intervals to connect. We define two properties of an interval of stable orientations:



Consider the interval between two stable orientations, $[d_1, d_2]$.
The **known-angle** of the interval, α, is the angle between the first and last stable orientations. The **fit-angle** of the interval, β, is the angle between the local-maximum of the push function m_1 preceding d_1, and m_2, the one following d_2.

The interval I_1 can then be 'collapsed' into the interval I_2 by a single push iff the size of the known-angle of I_1 is at most the size of the fit-angle of I_2, and any push direction between $Angle(m_1, d_1)$ and $Angle(m_2, d_2)$ will do.

Graph algorithms: BGL, also known as the boost::graph library, was used for graph algorithms, specifically BFS that is used within in Goldberg's algorithm.

Simulation

The simulation is written in Python, and it simulates the physics of the push-plan operating on the polygon. The same push-plan is demonstrated on multiple copies of the polygon simultaneously to show that it works regardless of the starting orientation. Additional information can be found in the documentation within the code.

Simulating pushed polygons: The simulation uses the pymunk 2D physics simulation library. We simulate objects 'lying on a table', which is not the typical case for such physical simulation, so configuring the physical environment to describe our situation was not immediate.

Open-Source contribution to pymunk: During the development of the simulation I found and reported a bug in the pymunk library. While debugging it I found several possible solutions and suggested them to the maintainer of the package, who implemented one of the said solutions. The full details can be found on the GitHub page: https://github.com/viblo/pymunk/issues/118

Numerical inaccuracies: Unlike the C++ code, the simulation is only approximate and the polygon sometimes ends up in the wrong orientation due to these numerical inaccuracies. With the time limitation of the project I decided to leave it this way, as it is not a critical feature.

Non-convex polygons: The pymunk library does not support collisions with non-convex polygons. Therefore, even though the algorithms can generate push-plans for non-convex polygons, we will not be able to simulate these plans.

Usage

Dependencies

The python simulation requires installing pymunk and pygame, which can be done by:

```
pip install pymunk
pip install pygame
```

The C++ code requires having CGAL configured, and it also requires CMake for compilation.

Running the algorithms

The C++ program, called part_orient, is compiled using CMake. Several randomly generated examples are provided and they can be used to test the program. All examples provided are convex due to the simulation limitations described above. After compilation, the code can be executed by:

./part_orient <polygon_path> For instance:

./part_orient ./examples/pres.cin

It will then write to standard output:

1. The push function of the input polygon, as described earlier.
2. A naive $O(n)$ push-plan that orients the input polygon ("naïve").
3. An optimal push-plan that orients the input polygon ("optimal").

The push-plans will also be saved to files, which will in this example be named:

./examples/pres.cin.naive.plan ./examples/pres.cin.optimal.plan

Running the simulation

After generating a push-plan using the part_orient program, we can visualize it by running:

python simulation/simulation.py <polygon_path> <plan_path> For instance:

python simulation/simulation.py ./examples/pres.cin ./examples/pres.cin.optimal.plan

This should display a window with 4 copies of the input polygon being oriented simultaneously by the provided push-plan.