3D Printing for Mobile Robots

Andrew Spielberg* Vicki Crosson†

Abstract

TODO: awesome abstract!

Citations can be done this way [**?**] or this more concise way [**?**], depending upon the application.

Keywords: 3D Printing, Robotics, 3Doodler, Trajectory Controller

1 Introduction

TODO: talk about previous work, what we hope to do different and why we're the best, and goals.

2 System Overview

TODO: annotate this with the figure steps and sections. Our system pipeline consists of several interworking components, which we illustrate in figure (TODO). A general workflow id described as follows. Users interact with our GUI in order to draw a set of splines, which represent the structure that they want the robot to draw. When the user is satisfied with their drawing, she may choose to output it to a file readable to the robot fabrication. In this output process, paths are ordered in a way that guarantees supports are generated before the splines which rest upon them while heuristically attempting to reduce potential for collisions. In this process, the splines are also resized to a scale that fits a reasonably sized workspace for the robot, and approximated as a set of polylines, which make robot control easier. The robot, a Kuka YouBot which is equipped with a 3Doodler on its end-effector, then reads in this file one polyline at a time. For each polyline, it plans to achieve an initial configuration which is collision-free with the currently fabricated structure (which we call the working structure), and iteratively fabricates the polyline segment-wise using our robot control algorithm. 3Doodler interfacing is achieved through an Arduino control module, and we equip the robot with a spool of 2.85 mm PLA plastic so that the 3Doodler does not have to be re-filled.

Our UI is built on top of PythonOCC (TODO: cite), an open source API for developing Computer-Aided Deisgn (CAD) software, which includes both visualization and geometric computing tools. Our robotics infrastructure is built upon the Robot Operating System (ROS) (TODO: cite). OpenRAVE (TODO: Cite) is used as a platform for computing inverse kinematics of the robot arm, as well as performing robot planning.

*e-mail:aespielberg@csail.mit.edu

†e-mail:viccro.mit@gmail.com

3 Clamp Design

Inspired by the idea of interchangeable tools for the YouBot, (TODO: cite IkeaBot) we developed a 3Doodler attachment which fits on the fingers of the YouBot as a glove. The rings of the 3Doodler were tightly fitted to the form factor of the 3Doodler and away from its center where the 3Doodler is widest, so as to lock it firmly in place (TODO: it can still rotate though. Address this). The clamp was 3D printed and coated in TangoBlack+ (TODO: cite Objet) to increase friction with the fingers and 3Doodler and decrease translational and rotational slip. The design of a clamp half (a single finger) is shown in figure (TODO) and the printed version, complete with 3Doodler, can be found in figure (TODO).

TODO: talk about total load of YouBot and mass of clamp + doodler?

4 Arduino Interface

TODO: Vicki - describe interfacing, slow extrusion, fast extrusion, ROS interface, on/off sensing (if we have it), isReady detection, and a circuit diagram, if possible.

5 UI Design

TODO: Vicki Note that we use cubic splines.

6 Path Generation Algorithm

In this section, we detail how our system converts our collection of B-Splines to ordered polylines for the YouBot to fabricate. While the need to convert B-Splines into polylines is necessary for our algorithm, it may be tempting to simply order them for fabrication in the same order that they were drawn in the GUI. This, however, is nave for several reasons. First, some splines may be unfabricable - splines which users draw suspended in mid-air, unattached to the ground or any other spline obviously cannot exist in the real world. Further, consider Figure (TODO: make a good drawing). Obviously, splines (TODO) cannot be fabricated before splines (TODO) which support them. Further, for those splines which do not require support (e.g. those that lie on the ground), fabricating them outwardly will require no extra planning as the 3Doodler tool will never intersect with any of its pre-fabricated splines at any step. However, fabricating them inwardly would require maneuvering the 3Doodler tool around the pre-fabricated splines to avoid collision.

In this section, we first explain how we transform a graph representing the splines and their connections into a tree in which any traversal, starting from the root, will never lead to fabricating a spline before its support. Next, we describe a heuristic for traversing that tree in a way that hopes to simplify the collision-free motion planning problem. Finally, we describe how we sample the splines and scale them to fit to a reasonable print volume.

6.1 Constraint Generation

Consider a graph $G = (V, E)$ where V corresponds to the set of splines and there exists an edge E between two nodes in V when their corresponding splines are connected (for each connection). With each edge $e_i \in E$ associate the minimum connection height

of all points in which those two splines connect, e_i^z, and with each node $v_j \in V$, associate the height of the minimum e_i^z among all edges e_i connected to v_j as e_{min_z}. Let $R \subseteq V$ be the root nodes of this graph, which correspond to splines which intersect the plane $z = 0$.

TODO: clean up all this terminology to match algorithm.

Given this setup, we present an algorithm in Figure (TODO) for transforming this graph into a forest such that the fabrication of splines in any path from any root to a connected leaf guarantees that any spline's support will be built before it. Note that as a limitation, this only considers geometric feasibility, and does not perform any sort of physical simulation to verify the stability of such a fabrication order.

Algorithm 1 Graph To Forest

procedure GRAPHTOFOREST
Input: G
$pq \leftarrow$ *PriorityQueue()*
$F \leftarrow$ *Graph()* ▷ We will build the output graph as F.
For each $r_i \in R$:
    $F \leftarrow F \cup \{r_i\}$
    For each e_{ij} connected to r_i
        $pq.put((r_i, e_{ij}), e_{ij}^z)$.
While $pq \neq \emptyset$: ▷ While the priority queue still has nodes
    $(v_i, e_{ij}^z) = pq.pop()$
    if $v_j \notin F$:
        $F \leftarrow F \cup \{v_j\}$
        $F \leftarrow F \cup \{e_ij\}$
        For each e_{jk} connected to v_j
            $pq.put((v_j, e_{jk}), e_{jk}^z)$.
Output: F

We say a spline a is supported by a spline b if b's lowest connection point is on a.

As an additional constraint, we require that for any two connected splines, they do not support each other. This is something we eventually hope to be enforced by the GUI.

Theorem 6.1. *Any traversal from a root to a connected leaf in the forest output by Algorithm 1 will not lead to fabricating a spline before its support.*

Proof. TODO: Is this good? Proof by Induction. Base case: We will always add a root node first, and since they intersect at $z = 0$, they require no support. Inductive case: Assume that through step k, no node has been added before its support. Now, assume that at step $k + 1$, we want to add a new spline v_{k+1}, that's connected to one of the existing splines in F. We prove here that this must choose a spline that is supported by F. Assume for contradiction's sake that v_{k+1} is *not* supported by the spline it is being attached to. If that's the case, then it must be supported by a different spline v' with a lower connect point which is not added yet. But the only way that this spline could not be added yet is if it or some other spline on the path from the root too v' has a higher support point. If this is the case though, it will have to be that either v' and v_{k+1} support each other or two splines on the path support each other (TODO: be more explicit here), which is not allowed. Thus, we have a contradiction.

□

Note that the way in which we "break cycles" here may not be unique, which may lead to suboptimal orderings in the subsequent step.

6.2 Heuristic Ordering

We wish to avoid moving the 3Doodler nozzle "inward," which could increase the difficulty of finding collision-free motion paths. The resulting forest from the previous step provides the constraints on which order splines can be drawn (no spline on a tree can be drawn before its ancestor) but we are otherwise free to traverse the forest in any order that we wish, starting from the set of root nodes. Here, we provide a heuristic for doing so in a way that reduces the odds of moving significantly "inward" in our drawing. Define a point a and being more inner than b if a is closer than b to the centroid axis-aligned bounding box (AABB) of our spline set. Observe that the only way we could move our 3Doodler nozzle monotonically outward is if we order our splines by increasing distance to the centroid of their innermost point (although, also note that such an ordering does not *guarantee* that we can move monotonically outward as such an ordering typically does not exist; the intuition however is that such a phenomenon may roughly occur several times as we move upward in z).

In other words, define the heurstic function of a spline $h(v) = -d(c, \min_t(v(t))$, where c is the centroid of the drawing's AABB and $v(t)$ is the value of spline v at parameter point t. We perform a depth-first traversal of our forest, choosing the lowest heuristic function as it's available. This generates our path ordering.

TODO: Should I put an algorithm?

6.3 Polyline Conversion

Since fabricating continuous splines is difficult, we simplify the problem by approximating splines as polylines. This process is comprised of three steps.

The first step is, for each polyline in our ordering, we choose a threshold value V (which will correspond to the resolution of our polyline, i.e. the length of each segment) and calculate a threshold T which satisfies $\forall t_{max} > t > t' > 0, |t' - t| < T \leftarrow ||v(t') - v(t)||_2 < V$ (PythonOCC provides functionality for calculating this value). We then sample v at increments of T. to create a list of candidate points C for our polyline.

Next, we scale and center our points in C (analogous to a normalizing procedure, but uniformly among all dimensions) in order to fit the drawing within a predetermined print volume. In practice, we set the x and y ranges to 10 cm, the z range to 4 cm, and centered the output drawing at $(0, 0, 0)$.

For each point in C we then greedily add to an output list C' the next point when its distance from the previously added point is at least V (PythonOCC does not provide functionality for arclength, and rather than attempt to write code to approximate it, we assume that for splines with sufficiently small curvature and small V that using this distance should not destroy much information about the polyline). We also add to C' the closest point to each connection point, which should be helpful for making sure the material is at the connection point (we should add the connection point itself but it's not yet clear to us how to determine where in C' it should be inserted). In practice, we keep V relatively small, at 1 cm.

7 YouBot Control

7.1 Arm Control

7.2 Base Control

8 YouBot Planning and Collision Avoidance

9 Results

10 Conclusion

10.1 Future Work

Acknowledgements

Mehmet Dogar and Robert Katzschmann, for suggestions for fixing the YouBot arm control when it wasn't working, which still didn't work.

11 References

To Cite: Ikeabot for example for hand tools for the YouBot.