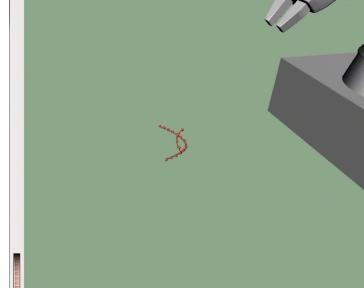


# 3D Printing for Mobile Robots

Andrew Spielberg\*

Vicki Crosson†



**Figure 1:** Our 3Doodler robot in the real-world and in simulation.

## Abstract

We present a pipeline for translating user-specified 3D spline-based drawings to physical, 3D printed objects. Our contributions are five-fold. First, we present a graphical user interface in which users can easily draw B-Splines which can be fabricated. Second, we provide an algorithm for generating a fabrication order of these splines which are consistent with support-based constraints, as well as a heuristic for ordering the splines and downsampling them to fabricable polylines. Third, we provide inexpensive hardware specifications for providing a Kuka YouBot with a 3Doodler end-effector tool which interfaces with ROS. Fourth, we provide a planning algorithm which enable the YouBots to plan collision-free paths with a working structure. Finally, we provide an attempt at smooth velocity control which works for a variety of starting configurations and target trajectories, to mixed results. We demonstrate our current results through simulation (with which we can perform motions within around 1 mm of the segment endpoints) and by fabricating a few simple primitives in the real world (with which we can perform motions within around 3 mm of the segment endpoints).

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

## 1 Introduction

3D printing has for some time been touted as a potential tool for mobile robots both for applications in manufacturing and environmental manipulation. Despite this, there have been few papers in which mobile 3D printing is executed in a way that is fast, precise, and solves some pressing challenge at hand for the robot. 3D printing of plastics is typically layered, and therefore slow. 3D printing of expansive foams [Hunt et al. 2014], [Napp and Nagpal 2014] is fast, but very low fidelity and potentially destructive to surrounding objects, and typically only useful for making coarse structures.

We are interested in the fabrication of high fidelity 3-D objects on a mobile platform with quick turnaround time.

Certain aspects of this problem has been explored by previous work, with several limitations. [Mueller et al. 2014] explored the rapid 3D printing of wireframes, but was limited to the fabrication of triangular wireframes and surfaces. Further, this work was limited to the print area of the printer, and only used a 3-DOF manipulator. [Keating and Oxman 2013] explored additive manufacturing by mobile manipulators in the context of building construction, but there were no end-to-end demonstrations provided and the focus was more on developing a sophisticated robotic platform rather than demonstrating a complete pipeline. [Hack et al. 2013] explored large-scale wireframe printing, but like [Mueller et al. 2014], was also limited to triangular meshes. [Mataerial ] has provided a 5-DOF arm for additive fabrication, but it is unclear from their website the full capabilities of their tool and how robust it is.

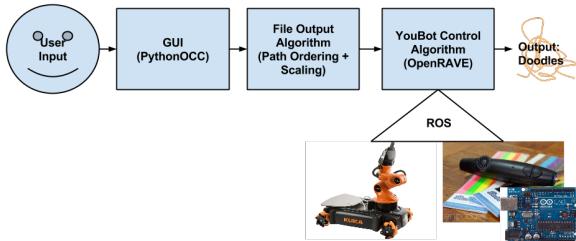
In this project, we seek to add a plastic extruder (namely, the 3Doodler [3Doodler ]) to the end of a 5-DOF mobile manipulator arm (which is 6-DOF when including mobility of the base) in order to enable the rapid prototyping of useful, minimalistic structures. We see potential applications in manufacturing, self-modification (the fabrication of grippers or tools), construction of temporary environmental structures such as bridges, and even potentially in fabricating linkage-based robots which can be deployed *in situ*. We demonstrate a pipeline for users which accepts GUI-based user design of spline-based structures, and outputs planning and control algorithms by which a robot can fabricate this drawing without collisions. We also provide the necessary hardware modifications to provide robots with autonomous 3Doodler control. We mostly demonstrate our results in simulation, but also spend perform a few simple real-world tests, with which we demonstrate fabricating a few simple primitives.

## 2 System Overview

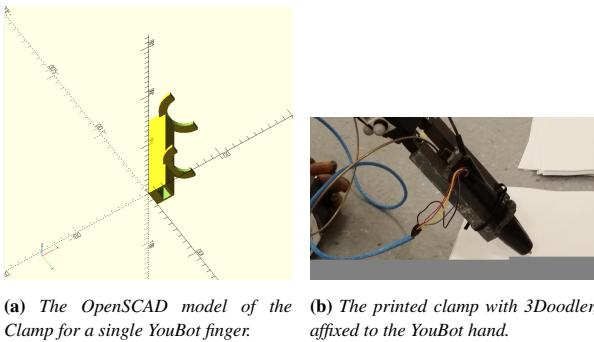
Our system pipeline consists of several interworking components, which we illustrate in figure 2. A general workflow is described as follows. Users interact with our GUI in order to draw a set of splines, which represent the structure that they want the robot to draw. When the user is satisfied with 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

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

†e-mail:viccro.mit@gmail.com



**Figure 2:** A diagram of our 3Doodler pipeline.



**Figure 3:** The 3Doodler Clamp.

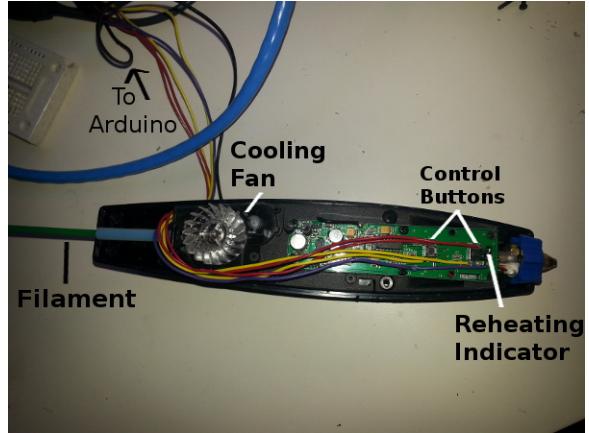
workspace for the robot, and approximated as a set of polylines, which make robot control easier. The robot, a Kuka YouBot [Kuka ] which is equipped with a 3Doodler on its end-effector, then reads in this file one polyline at a time. For each polyline, it plans to achieve an initial configuration which 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 [Pyt ], an open source API for developing Computer-Aided Design (CAD) software, which includes both visualization and geometric computing tools. Our robotics infrastructure is built upon the Robot Operating System (ROS) [Quigley et al. 2009]. OpenRAVE [Diankov and Kuffner 2008] is used as a platform for computing inverse kinematics of the robot arm, as well as performing robot planning.

Our code can be found at <https://github.com/aespielberg/3DoodlerController>.

### 3 Clamp Design

Inspired by the idea of interchangeable tools for the YouBot [Knepper et al. 2013], 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 (with some asymmetry added near the power port to prevent rotation). The clamp was 3D printed and coated in TangoBlack+ [Stratasys ] 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 Fig. 3a and the printed version, complete with 3Doodler, can be found in Fig. 3b.



**Figure 4:** Hardware modifications to the 3Doodler.

### 4 Arduino Interface

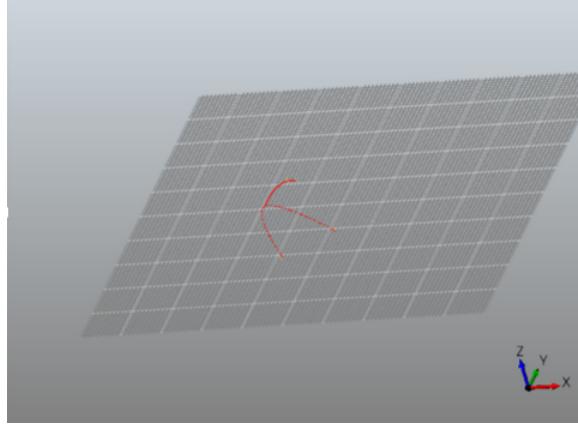
In order to remotely control the 3Doodler pen, a hardware interface was required. The 3Doodler was modified in order to interface with an Arduino Uno, a microcontroller board with USB interface, as can be seen in 4. Wires were run from the control buttons on the circuit board of the 3Doodler to external switching circuitry, which allows users to toggle the fast and slow extrusion settings on and off. The 3Doodler pauses intermittently to reheat internally, stopping all extrusion for some variable amount of time. Additional circuitry was needed to identify whether the heater was actively running, so that the robot does not attempt to draw during this reheating period. A 5V buck-boost converter was used to regulate this signal before reading it with the Arduino. The Arduino was programmed to directly publish and subscribe to ROS messages used by the robot controller, using the rosserial package and roslib Arduino library, in particular broadcasting when the 3Doodler is ready for printing and accepting signals for fast and slow extrusion. This allows for direct communication between hardware and software components.

### 5 UI Design

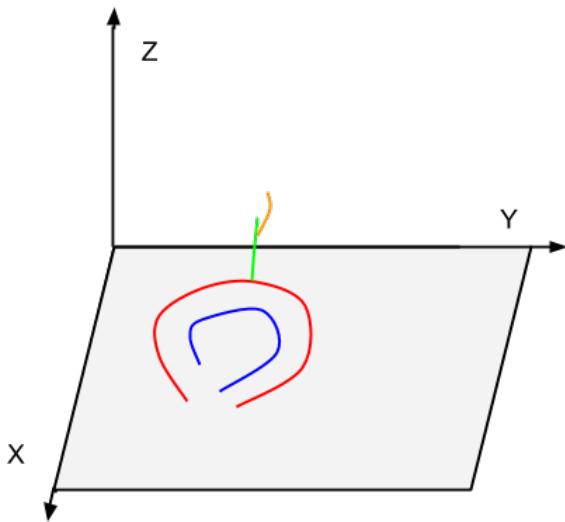
Our user interface (Fig. 5) was designed in order to allow users to easily draw arbitrary splines. The GUI allows for B-spline creation via cube interpolation over a series of knot points, which are created on click. Individual splines can be deleted by selecting them and hitting the delete key. Finally, since doodled structures must be supported, we added two features. First, when adding a knot point, if a spline is currently selected, the added knot will snap to that spline by taking the projection to the curve. Secondly, the first spline is always completely in the  $z = 0$  plane, giving future splines a reliable base to build upon. In order to make adding knot points easier when not snapping them to existing splines (since a point on a 2D screen can correspond to an infinite number of locations along a ray in 3D), we add knot points beyond the first point of a spline to the plane uniquely defined by the first point and the camera ray as the planes normal.

### 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,



**Figure 5:** The graphical user interface with two splines drawn.



**Figure 6:** A cartoon of a working drawing.

is naïve 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 Fig. 6. Obviously, a spline such as the one in orange cannot be fabricated before splines which support them (the one in green), except for splines which lie on the ground (red and blue), which do not need to be supported. Further, fabricating splines 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 Fig. 6 it is much harder to fabricate the blue spline after the red spline as the 3Doodler tip may intersect the red spline while moving into position to start that spline's fabrication sequence.

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

First, a definition: We say a spline  $a$  is supported by a spline  $b$  if  $b$ 's lowest connection point is on  $a$ . Splines which intersect  $z = 0$  do not have nor need support.

Now, 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  $e_{ij} \in E$  connecting nodes  $v_i$  and  $v_j$  associate the minimum connection height of all points in which those two splines connect,  $e_{ij}^z$ . Let  $R \subseteq V$  be the root nodes of this graph, which correspond to splines which intersect the plane  $z = 0$ .

Given this setup, we present an algorithm in Alg. 1 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. Also note by the very nature of the algorithm, it never reaches splines unconnected to a root (intersecting the ground), and therefore these are excluded from the final path output, as desired.

---

### Algorithm 1 Graph To Forest

---

```

1: procedure GRAPHTOFOREST
2: Input:  $G$ 
3:  $pq \leftarrow PriorityQueue()$ 
4:  $F \leftarrow Graph()$   $\triangleright$  We will build the output graph as  $F$ .
5: For each  $r_i \in R$ :
6:    $F \leftarrow F \cup \{r_i\}$ 
7:   For each  $e_{ij}$  connected to  $r_i$ 
8:      $pq.put((r_i, e_{ij}), e_{ij}^z)$   $\triangleright$  Put  $(c_i, e_{ij})$  into the priority
    queue with priority  $e_{ij}^z$ .
9: While  $pq \neq \emptyset$ :  $\triangleright$  While the priority queue still has nodes
10:    $(v_i, e_{ij}^z) = pq.pop()$ 
11:   if  $v_j \notin F$ :
12:      $F \leftarrow F \cup \{v_j, e_{ij}\}$ 
13:     For each  $e_{jk}$  connected to  $v_j$ 
14:        $pq.put((v_j, e_{jk}), e_{jk}^z)$ .
15: Output:  $F$ 

```

---

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.* 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 connection 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 to  $v'$  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, 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 axis-aligned bounding box (AABB) of our spline set, centered at the set's median. Observe that the only way we could move our 3Doodler nozzle monotonically outward (necessary condition) is if we order our splines by increasing distance to the median of their innermost point (although, also note that such an ordering does not guarantee that we can move monotonically outward as it is not a sufficient condition, since two splines can each be more "inward" than each other at different radial angles about the median; the intuition however is that such a phenomenon may roughly occur several times as we move upward in  $z$ ).

In other words, define the heuristic function of a spline  $v$  as  $h(v) = d(c, \min_t(v(t)))$ , where  $c$  is the median 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 smallest heuristic function as it's available. This generates our path ordering.

## 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  $\forall t_{max} > t > t' > 0, |t' - t| < T \Rightarrow \|v(t') - v(t)\|_2 < V$  (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. In practice, we keep  $V$  relatively small, at 1 cm, although this can be a parameter depending on the robot arm being used.

## 7 YouBot Control

In this section we detail the algorithm used for controlling the algorithm and extruding material. We note that *arm control* and *base control* are used exclusively; the arm and base are never moving simultaneously. This simplifies our problem by avoiding the difficulty of having to synchronize arm and base movement. Further, arm motions are used exclusively for drawing line segments of the polylines, while base movements are used to move the YouBot in order to prepare for the next segment.

Call the forward direction of the robot's frame  $r^x$  direction, the vertical direction  $r^z$ , and the remaining orthogonal direction  $r^y$ . Because preliminary experiments showed that printing by moving the arm straight forward is problematic (the print head tends to interfere with the newly extruded material), and that movement in  $r^y$  is difficult, we keep the arm on the  $r^x$ - $r^z$  plane and only move it within the second quadrant (considering the end effector's starting position as the origin). Thus, we are bringing the arm backward (and possibly up) for each stroke.

Because experiments also showed that material extruded best when the 3Doodler was pointed away from the direction in which it moved (as in the teaser), we point the 3Doodler away from the direction of motion and solve for arm joint angles which keep the 3Doodler angle *fixed* through motion.

Note that we are making a very key assumption here, that material always sweeps out the same curve as the end effector when extruding. In this regard, our control problem is open loop. We do not address how to fix problems caused by inconsistent or irregular extrusion here.

### 7.1 Arm Control

Given a starting position  $S \in \mathbb{R}^3$  and a goal position  $G$ , we seek to generate smooth, constant velocity trajectories between the two points. Because we are only using 3 joints of our arm for the in-plane arm movements, inverse kinematics can be solved in an almost trivial amount of time. Call the arm configuration at pose  $p$   $A_p$ , that is, the three-dimensional array corresponding to the current joint angles of joints 2, 3, and 4 on the YouBot.

We loop until our end-effector reaches its 3D goal within a desired tolerance (1 mm). Let  $T_i$  refer to the position for the end-effector of the robot at step  $i$ . At each step, we calculate a subgoal position,  $g_i = T_i + \Delta(G - T_i)$ , where  $\Delta$  is a small step size. In practice, we set this value to be 1 cm. We then solve for  $A_{g_i}$  using inverse kinematics, keeping our desired rotation fixed and equal to the starting rotation of the 3Doodler. Because  $A_{g_i}$  could potentially yield two solutions, we choose the one which is closest to our robot arm's current configuration in  $L_\infty$  distance, which prevents large jumps. Because we are only moving over short distances in practice, hitting the robot joint limits is not a problem as long as we do not start too high (this can be guaranteed when we set our print workspace in the file output step). We calculate  $A_{\Delta_i} = A_{g_i} - A_{T_i}$  and set our arm velocities proportional to this difference by normalizing this vector and multiplying it by some velocity factor. In practice, the total velocity of the arm in 3D is virtually static despite us only fixing the rotational velocities' norm.

We perform a number of refinements to this basic algorithm to attempt to improve results, which we detail below.

Because sometimes our arm may not reach its desired goal within the desired 1 mm threshold, we set a second, sort of "good enough" threshold, at 4 mm. At each step  $i$  we keep track of the best distance  $d_{best_i} = \|G - A_{T_i}\|_2$  seen so far. If we are then farther than this distance for some number of consecutive steps (in practice 10 after

step  $i$ , we assume we've done the best we can and will only make backwards progress at this point, at which point we break our loop.

In order to prevent large instantaneous jumps in velocity at any of the arms which can lead to large errors, we slowly accelerate from any rest configuration.

In order to combat drift in the arm, which can occur over some larger distances, at each step we also project  $T_i$  onto the ray created by  $S - G$  before calculating  $g_i$ .

Finally, we employ a PID controller, where  $A_{\Delta_i}$  represents the  $P$  term at each step, and the  $I$  term at step  $i$  is calculated as  $A_{f_i} = \sum_{k=1}^i (A_{T_k} - A_G)(t_k - t_{k-1})$ . In practice, we then command our velocities as proportional to a normalized  $A_{PID_i} = PA_{\Delta_i} + IA_{f_i}$ .  $A_{f_i}$  is reset to 0 before each arm motion.

## 7.2 Base Control

Although base control was not implemented for this project (it was implemented by Spielberg at a previous date), it is worth describing for completeness. Base control is handled by a simple proportional controller in  $x$ ,  $y$ , and  $yaw$ . The YouBot base is outfitted with Mecanum wheels, providing it omnidirectional motion. Our  $x$  and  $y$  thresholds in the base position are set to 1 mm and our  $yaw$  threshold is set to 0.05 rad. Velocity in this process proportional to the remaining distance in  $x$  and  $y$  as well as in  $yaw$ .

## 8 YouBot Planning and Collision Avoidance

---

### Algorithm 2 SamplesToFab

---

```

1: procedure SAMPLESTOFAB
2: Input:  $\mathcal{S} = [S_1, S_2, \dots, S_n]$   $\triangleright$  Each  $S_i$  is a list of waypoints
   on a polyline
3: For each  $S_i \in \mathcal{S}$ :
4:   For each waypoint  $w_j \in S_i, j > 1$ :
5:     Move YouBot to  $w_{j-1}$  aligned along  $w_{j-1} - w_j$  vector
       in  $x$ - $y$  plane.
6:
7:     Calculate line segment  $(m_{j-1}, m_j) = w_{j-1} - w_j$  rep-
       presentation in  $r$  frame's  $r^x$ - $r^z$  plane.
8:     Move YouBot arm smoothly from  $m_{j-1}$  to  $m_j$ .
9:     Move YouBot base backwards by  $(m_{j-1} - m_j)_x$   $\triangleright$ 
       This happens before arm re-positioning to avoid collisions with
       freshly extruded material
10:    Move end-effector to  $w_j$ , maintaining 3Doodler angle.

```

---

Our algorithm for fabricating our polylines can be seen in 2. A key requirement for this project is collision avoidance. After material is extruded, we must be careful not to touch it with the robot so as not to ruin our fabricated structure.

In order to do this, we augment our model of our environment whenever we extrude material. Since we assume segments are extruded straight, we add rigid, static kinematic thin rectangular prism bodies (to simulate the thin material strands) into our OpenRAVE environment which extend from the starting point to the end point of each extruding arm movement.

We run a rapidly exploring random tree (RRT) [Lavalle 1998] algorithm (specifically, the bidirectional RRT [Jr. and Lavalle 2000]) whenever we need to move the base and whenever we need to move the arm into a position while not extruding. We set the size of each step of kinematic motion in the RRT algorithm as 0.5 mm. Since our extruded material is  $\geq 1$  mm in diameter, this makes it unlikely

that RRT, which is evaluated would ever find a solution which the arm or base "tunnels" through extruded material.

Note that there are four drawbacks of this approach which make our search incomplete given our problem. First, because our fabrication order is fixed before this step, it is possible that our system may fail to find a collision-free path, but a collision-free path may exist if the splines were fabricated in a different order. Second, we do not check for collisions while the arm is extruding, and assume that these small motions will necessarily be collision free. In the future, detecting whether or not this motion will be collision free, and potentially changing our drawing angle if it is not collision free remains a desirable goal. Third, we treat the base and arm as separate entities moving only one at a time, so there may be solutions which avoid collisions which require moving the base and arm simultaneously. Fourth, because we keep YouBot joint 1 fixed at an angle of 0, we may not be able to find a solution to our planning problem although one in fact exists with rotating joint 1. Again, these last two items are something we hope to solve in the future by searching over the combined kinematic object and eventually moving the first YouBot joint.

## 9 Results

We have been able to successfully demonstrate the entirety of the pipeline from start to finish. A video of our complete pipeline working in simulation can be seen at [https://youtu.be/f8muS50Pp\\_A](https://youtu.be/f8muS50Pp_A). In simulation, precision is to 1 mm, and connection points are adequately aligned almost all of the time.

Physically, however, producing the splines that are drawn is not possible. Precision in the real world is to 3 mm at best, and we've found in practice that we've had to typically tailor the controller specifically to certain directions of motion to get them to work consistently. When not tailored, the arm motion is not smooth enough to create rigid line segments. We were able to demonstrate vertical (Fig. TODO) and horizontal lines (Fig. TODO), and in one case where we decided to activate YouBot joint 1, a square completely in plane with only the YouBot arm (Fig. TODO). Video of some simple 3Doodler primitives made via the YouBot (horizontal and vertical lines) can be seen at <http://youtu.be/gLuPqD9eVNw>.

## 10 Conclusion

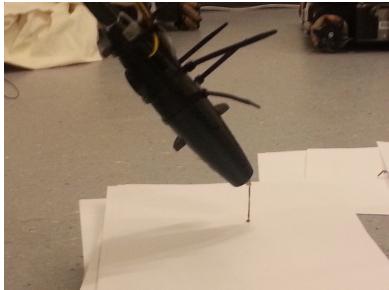
### 10.1 Future Work

### Acknowledgements

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

### References

- 3DOODLER. 3doodler: The world's first 3d printing pen. <http://www.the3doodler.com/>.
- DIANKOV, R., AND KUFFNER, J. 2008. Openrave: A planning architecture for autonomous robotics. Tech. Rep. CMU-RI-TR-08-34, Robotics Institute, Pittsburgh, PA, July.
- HACK, N., LAUER, W., LANGENBERG, S., GRAMAZIO, F., AND KOHLER, M. 2013. Overcoming repetition: Robotic fabrication processes at a large scale. *International Journal of Architectural Computing* 11, 3, 285–300.



(a) A vertical line.



(b) A horizontal line. The thinning sections are due to imperfections of extrusion with the thick 3Doodler nozzle with which we were experimenting.



(c) A square. Curvature is caused due to the slightly circular motion of rotating joint 1 on the YouBot.

HUNT, G., MITZALIS, F., ALHINAI, T., HOOPER, P., AND KOVAC, M. 2014. 3d printing with flying robots. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, 4493–4499.

JR., J. J. K., AND LAVALLE, S. M. 2000. Rrt-connect: An efficient approach to single-query path planning. In *Proc. IEEE Intl Conf. on Robotics and Automation*, 995–1001.

KEATING, S., AND OXMAN, N. 2013. Compound fabrication: A multi-functional robotic platform for digital design and fabrication. *Robot. Comput.-Integr. Manuf.* 29, 6 (Dec.), 439–448.

KNEPPER, R. A., LAYTON, T., ROMANISHIN, J., AND RUS, D. 2013. Ikeabot: An autonomous multi-robot coordinated furniture assembly system. In *ICRA*, IEEE, 855–862.

KUKA. Youbot detailed specifications. [http://www.youbot-store.com/wiki/index.php?title=YouBot\\_Detailed\\_Specifications](http://www.youbot-store.com/wiki/index.php?title=YouBot_Detailed_Specifications).

LAVALLE, S. M. 1998. Rapidly-exploring random trees: A new tool for path planning. Tech. rep.

MATAERIAL. Mataerial: A radically new 3d printing method. <http://www.mataerial.com/>.

MUELLER, S., IM, S., GUREVICH, S., TEIBRICH, A., PFISTERER, L., GUIMBRETIÈRE, F., AND BAUDISCH, P. 2014. Wireprint: 3d printed previews for fast prototyping. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ACM, New York, NY, USA, UIST ’14, 273–280.

NAPP, N., AND NAGPAL, R. 2014. Distributed amorphous ramp construction in unstructured environments. *Robotica* 32, 2, 279–290.

Pythonocc. <http://www.pythonocc.org/>.

QUIGLEY, M., CONLEY, K., GERKEY, B. P., FAUST, J., FOOTE, T., LEIBS, J., WHEELER, R., AND NG, A. Y. 2009. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.

STRATASYS. Tango+/tangoblack+. [http://www.growit3d.com/wp-content/uploads/2011/08/polyjet\\_tango+\\_final.pdf](http://www.growit3d.com/wp-content/uploads/2011/08/polyjet_tango+_final.pdf).