Sketchbot: Redefining Robotic Drawing of 3D Objects with Fine Details*

*Robotic Manipulation (6.4210, Fall 2023)

Zhishen Chen
Dept. of MDes
Harvard University
Cambridge, MA
zhishen_chen@gsd.harvard.edu

Maura Kelleher
Dept. of EECS
Massachusetts Institute of Technology
Cambridge, MA
maurakel@mit.edu

Shayda Moezzi
Dept. of EECS
Massachusetts Institute of Technology
Cambridge, MA
smoezzi@mit.edu

***Abstract*—We explore how a robot arm can capture and draw captivating artworks from seen objects. This equips machines with artistic expression which for long has been perceived as a human endeavor. However, drawing with precision and artistry using robot arms presents many challenges: from abstracting reliable drawing trajectories (e.g. a clean line drawing) from the observed objects, to handling the control of visual effects (e.g. having stroke capability). In this report, we present the SketchBot, a full-stack robot painter, empowering machines to bridge the gap between technology and artistry. Our proposed approach comprises three pivotal elements: image-to-sketch simplification, drawing trajectories from vectorized sketches, and mapping and fine-tuning SketchBot's motion to recreate rich stroke effects. These elements synergize to enable the robot to vividly capture intricate details and faithfully reproduce observed 3D models.**

***Index Terms*—robotic manipulation, drawing robot**

I. Introduction

Art has long been a conduit for human expression, a canvas on which emotions, ideas, and creativity find their voice. Yet, what if we could transcend the boundaries of human capability and imbue machines with artistic prowess? This is the journey we embark upon—to transform robots into artists, capable of translating intricate 3D models into captivating artworks using varying strokes. The motivation behind this endeavor stems from the pressing need to address limitations encountered in previous course projects: the inability to render finer details and dynamically change key parameters for stylized drawings.

However, precisely capturing the essence of a 3D model and translating it into intricate artwork presents many challenges. Firstly, advanced perception techniques are needed to abstract sketches from the 3D model, enabling the robot to grasp the essence of the subject. Secondly, the task involves simplifying and translating these sketches into precise trajectories, effectively mapping the artist's intent onto the canvas. Thirdly, the mapping of varying line widths into the nuanced movements of the robot painter is crucial, ensuring that the artwork faithfully reflects the desired strokes. Finally, fine-tuning the robot's capabilities is also critical, ensuring the execution of intricate drawings on the canvas with a high level of precision throughout the creative process.

In this report, we present the SketchBot Fig. 1, a full-stack simulated robot painter. Our overarching design goal is to empower the robot not only to faithfully depict the outline but also to breathe life into finer details, through an enhanced perception system and the ability to wield varying strokes.

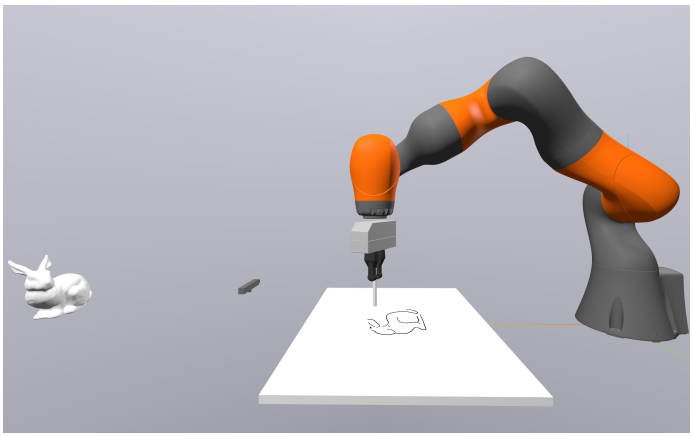


Fig. 1. SketchBot: The robot detects object placed in front of the camera and recreates a line drawing of it on a canvas. To faithfully reproduce fine details, images captured from the camera are process through two image translation models to get a clean line drawing for trajectory mapping. The variation of line widths results from the stroke capability and fine-tuned simualtion parameters that mimics drawing with a chalk on the canvas.

II. Related Work

Our project draws inspiration from the work done by students in previous years of MIT 6.4210, particularly the "Da Vinci" [1] and later "Michelangelo" [2] project. For both the Da Vinci and Michelangelo project, the teams explore the task of manipulation and grasping by means of creating a robotic painter that paints the outline of objects in a scene. They follow a pipeline of image segmentation of the object in scene, path generation for drawing out the said object, object grasping for the drawing utensils, and finally the planned steps of execution. However, it's important to note that in these projects, the focus primarily remained on capturing the silhouette of objects

and executing drawings without variations, leaving room for further innovation and advancement in our SketchBot project.

The foundation of our project and what we hope to recreate is the work done in [3]. In this paper, the authors propose a semi-automatic framework that extracts expressive strokes from 3D models and draws them in oriental ink painting styles using a robotic arm. Their framework consists of a simulation stage where geometrical contours were automatically extracted from a certain viewpoint and a neural network was employed to create simplified contours. Then, in the robotic drawing stage, an optimization method was presented for drawing smooth lines with varying stroke widths. In addition, [4] also explores a robot painting system that is designed to translate a user's high-level intentions into real world paintings. Their approach explores the integration of computer vision and machine learning techniques to create artwork from a blend of input sources, including images, text, sounds, and sketches. Of particular interest to us was their parameterization of brush strokes using three key parameters—pressure, length, and bend—an idea that has significantly influenced the development of our own stroke mapping approach.

III. Overview of Methodology

Image to sketch simplification: Leveraging two consecutive image-to-image models, we first transform visual input from the simulated camera into intricate sketches, capturing all contours and shadows. The second model then refines these sketches into clean, simplified line drawings, ready for the subsequent trajectory generation. The result is an image that encapsulates the essential drawing lines, faithfully mirroring the details of the observed 3D model.

Drawing trajectories from vectorized sketches: We employ a process that begins with skeletonization through Medial Axis Transfer (MAT), followed by resampling and the separation of independent drawing lines. The identification of distinct drawing lines is paramount for their faithful rendition on the canvas. Subsequently, we map these trajectories onto the canvas within our simulated environment, laying the foundation for SketchBot's motion planning.

Mapping and fine-tuning SketchBot's motion in the z-axis: Mapping the drawing lines' curvature to motion enriches the visual effects with stroke capability, inspired by the *Noutan* results with writing brush in previous work [3]. We map variations in the z-axis motion during the drawing process to the curvature, simulating a brush-painting effect where thicker lines emerge at turning points. This simulation is achieved by mapping the contact force between the drawing tool and the canvas to the generated "lines" in the virtual environment.

The primary contribution of our work lies in enhancing the drawing capabilities of the robot, specifically by improving the perception system's ability to comprehend and abstract from observed models and by refining the motion planning process with carefully mapped drawing trajectories. We evaluate the efficacy of our approach through visual inspection of the robot's output, seeking smooth and faithful renditions of observed models. Notably, our SketchBot vividly captures and faithfully reproduces intricate details not only of the silhouette but also the body, as exemplified in Fig. 1. Qualitative results(videos) are available at https://youtu.be/3vUXPieEtBU?si=gXghcNDke4-a13Jo.

IV. Implementation and Results

A. Visual Input to Sketch Pipeline

*1) **Scene Setup** :* We create the manipulation environment with Drake using an iiwa robot arm and a WSG gripper, which is welded to a writing utensil (a chalk) as shown in Fig. 1. We included a camera that provides a direct view of a 3D model (or a scene of YCB objects) and placed all of these items accordingly. The SDF files for the chalk, canvas, and rabbit are created from scratch.

A key functionality of our simulation is to simulate drawing, which is implemented in a contact-based technique. Adapted from Prof. Russ Tedrake's MeshcatWriter leaf system, a system that determines whether a writing utensil and drawing surface (in our case, the chalk and white canvas) are in contact and creates a capsule at that point. Depending the magnitude of the spatial force at the contact, we vary the drawn line's width to simulate the effect of stroke.

*2) **Extracting Detailed Contours**:* After setting up the scene, we are able to capture an image of the 3D bunny model. Fig. 2.A shows the view from the camera in the scene. We determine that 0.5 meter was the optimal distance from the camera to the bunny in order to capture a detailed and high quality image. The next step in fulfilling our goal of creating a descriptive and pleasing drawing of the bunny is extracting the contours from the 3D model to generate a 2D sketch.

To do this, we leverage widely available open source models from Hugging Face. We first experimented with using an image to line drawing model [5] that makes a sketch based on an input image. However, after implementing and executing later stages of the perception pipeline we found that this model made too detailed of a sketch. We determined that it would be infeasible to construct a trajectory for the robot based on these extracted contours (Fig. 3).

Our next approach was to leverage the OpenCV python library and utilize the built-in contour extraction and Canny edge detection. The Canny edge detection on its own worked well to construct an outline of the bunny.

Past teams in 6.4210 have utilized this method to extract the outline of a desired object and contstruct the trajectory based on this outline [1] [2]. However, we found that the the extracted contours from OpenCV did not capture the surface well and would not produce a drawing with our desired level of detail (Fig. 4).

Fig. 4. Contours extracted with OpenCV

Pivoting back to Hugging Face, we found a new image-to-sketch model that was able to produce a drawing of the bunny

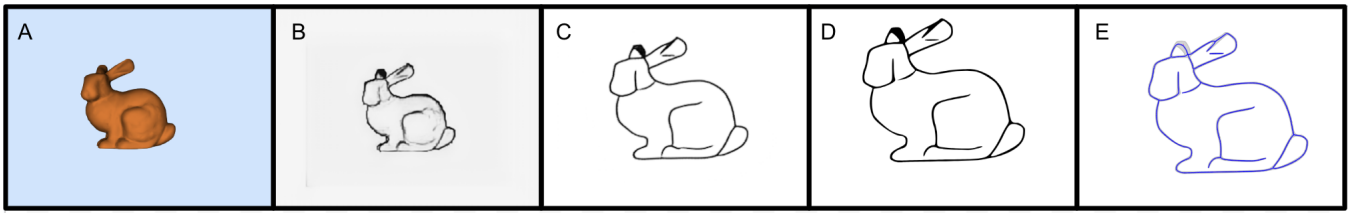


Fig. 2. Overview of each step in the processing pipeline to take the image from the camera to vectorized lines suitable for trajectory planning for drawing. (A) is the image captured from the camera in the scene as show in Fig. 1. (B) is the output from the image-to-sketch model from Hugging Face. (C) is the result of applying the sketch to contour simplification using methods from [6]. (D) is the result of converting raster image (C) into the SVG format. (E) is skeletonized contours resulting from the Medial Axis Transform

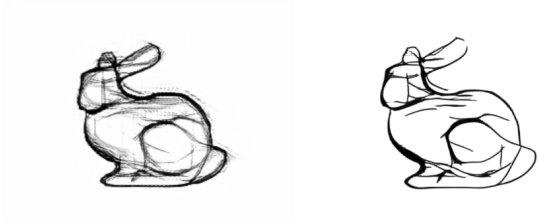


Fig. 3. First attempt of extracting the contours from the camera image. The bunny on the left is the output of the image-to-sketch model from hugging face. The bunny on the right is the output after simplifying the contours. The sketch creates a lot of noise that leads to busy contours. These are not suitable for clear vectorization and trajectory planning

with our desired level of detail (Fig 2. B) [6]. We determined this image was ideal because the contours were clear and there is not too much noise that could potentially over complicate the trajectory.

After creating the initial sketch, it is clear that we need to simply it into distinct contours in order to implement a trajectory for the drawing. So, we implored techniques to simplify these contours similar to [7]. This method utilizes a fully convolutional neural network to produce clean and clear contours from rough sketch images. This model is unique because it is able to process images of any dimensions and aspect ratio into an output image of the same size and it does not require the input image to be a vector image. The model from the original paper utilized depricated pytorch modules, so we had to use a newer version of the model and update some of the surrounding code to migrate it to newer versions of pytorch. Once implemented, we passed our extracted bunny sketch into the model to obtain the simplified contours from the sketch (Fig 2. C). These contours form the basis for our trajectory. In order to translate the contours into a trajectory, we must first vectorize the raster image.

*3) **Storing Detailed Image Data via SVG**:* Scalable Vector Graphics (SVG) is a vector image format that is based on mathematical equations to define shapes and paths. This means that SVG images are independent and can be scaled to any size without loss of quality. This image is defined by points, lines, curves, and shapes, which makes it inherently more precise—this is essential in the case of generating a trajectory for the arm to precisely draw an object in the scene. Raster images, on the other hand, store information about each individual pixel in the image. Thus when you resize a raster image, the software has to interpolate and estimate colors for the new pixels, potentially resulting in a loss of quality or precision.

*4) **Converting from PNG to SVG**:* In order to vectorize the raster image of the simplified contours into the svg format, we used a tool called potrace [8]. Potrace enables us to convert a bitmap into a smooth, scalable image - the SVG. The potrace tool traces the bitmap to get a series of curves that together, construct a path representing all of the lines in the input image. Then, iterating through each of these curves allows us to create a properly formatted SVG image. SVG files are XML documents that define graphical elements using tags. We found that not many tools were available in Python for properly extracting curve commands and dimensional information from SVG files, thus we created our own functions to extract this information from the file. We parse the XML file for the "path" tag, where we can see a string of commands (e.g. M for MoveTo, L for LineTo command, Q for quadratic Bézier curve, C for cubic Bézier curve), each followed by coordinates associated with the curve or move command. In our case, we want to extract all the quadratic and cubic bezier curves and their associated 2 or 3-coordinate points, respectively. Upon successful extraction of this data, we now have all the coordinate and curve data we need to be able to generate a detailed potential trajectory for the robot to follow.

*5) **Skeletonizing Image via Medial Axis Transform**:* One challenge we observed while examining our SVG image was the excessive detail in our contour drawing, resulting in multiple lines (SVG paths) for specific features that incorporated shading and depth information. We can see this in box D of Figure 2. This intricate representation could be useful when determining stroke width in specific areas but posed an issue for trajectory planning, as it led to redundant arm movements. Consequently, we need to simplify our contoured SVG file, condensing it into a single "skeleton" curve.

To achieve this simplification, we leveraged the Medial Axis Transform (MAT), a mathematical technique frequently employed in image processing to extract an object's central

axis or skeleton. Skeletonization involves reducing the representation of an object to its essential structure or core, often represented as a set of interconnected lines or curves. Essentially, the MAT determines the closest boundary points for each point in an object. An inner point belongs to the "skeleton" if it has at least two closest boundary points. While most existing methods in Python primarily cater to raster images, we found the Flo-Mat Javascript Medial and Scale Axis Transform(SAT) Library specifically for transforming SVG images. SAT is based on the work done by [9].

Using the Flo-Mat library, we modified an existing script for SVG transformation. The function extracts the Medial Axis Transforms of all shapes created by the array of input beziér loops (that is stored in the SVG path tag as mentioned previously). It then takes an additional parameter indicating the number of points on each bezier for which a MAT vertex 'point' should be calculated. A value of 3 is the default and is a reasonable compromise between speed and accuracy. A value of 15 would give highly accurate results.

Subsequently, we retained this simplified, skeletonized MAT curve, preserving it as an SVG file to employ in our trajectory planning. As illustrated in Figure 2, a clear distinction can be observed between the pre-skeletonization (box D) and post-skeletonization (box E) stages. In box E, the blue curve signifies the extracted line, while the lighter shade of grey represents the original drawing's simplification in terms of contour representation.

B. Generating Trajectory from Vectorized Sketch

1) ***Down-sampling and Separation the Drawing Lines****:* The initial skeleton output of our sketch comprises over 5000 curve segments, , consisting of various kinds of bezier and cubic curves. To streamline this complex structure, we first filter and down-sample the key points, which correspond to the "MoveTo" points in the SVG file, representing key-frame origins along the drawing path. Through extensive testing with varying sampling densities, we identify the optimal balance between preserving smoothness and detail while minimizing the number of key points (Fig. 5).

Then, the next step is to seperate the points into distinct drawing lines to ensure alignment with the sketch's intent. To accomplish this, we adopt an intuitive approach: we analyze the rate of change in distances between consecutive points. When this rate exceeds a predefined threshold, it signifies a transition to the start of a new drawing line. By iteratively processing the down-sampled key points, we generate a sequence of lists, each representing the trajectory for a specific drawing line.

2) ***Generating Target Key Frame Pose Trajectory****:* Building upon the separated key points, we augment each point with both a pre-draw point (elevated from the starting point) and a post-draw point (elevated from the ending point) to mimic the natural behavior of drawing a line. Subsequently, we construct piece-wise key frame poses using these augmented

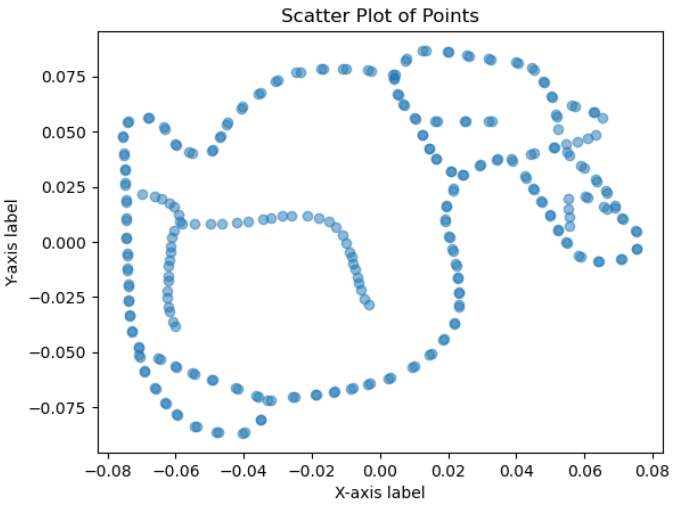


Fig. 5. Feature points are down-sampled from 5636 to 282 and re-positioned at the bunny's center. The plotted points above only possess the 2D coordinates mapped form the image.

points, associating them with timestamps aligned to the total drawing length. It's important to note that, at this stage, the translated points possess uniform z-coordinates, aligning with the 2D sketch's characteristics.

C. Varying Stroke and Fine-tuning Mortion

In our efforts to artistically enhance Sketchbot's drawing abilities, we embedded the ability to dynamically change the stroke width of the rendered drawing. The original inspiration for varying stroke width came from [3]. The paper creates a framework for users to create personalized expressive digital strokes in the simulation stage. This then is followed by stroke optimization, mapping and motion planning methods, allowing the framework to draw stylized ink paintings in real execution stage on a robotic arm.

1) ***Mapping Hydroelastic Contact Force to Width****:* To implement the drawing functionality, we examine the hydroelastic contact information between the chalk and the canvas. By understanding the hydroelastic contact points, we gain insights into when we should initiate drawing on the canvas and precisely where the lines should be placed. To achieve the dynamic stroke width, we use the spatial force applied on the canvas by the chalk at the centroid point. We then calculate the magnitude of the scaling force and apply a constant scaling factor of 0.0003. This value is then mapped to the width of the line at this specific contact point. In this way, we are able to dynamically vary the stroke width based on the hyrdroelastic contact between the canvas and the chalk.

2) ***Calculating Drawing Regions of Higher Stroke Width****:* Initially, our approach to determine where varying stroke widths should be applied in the sketch involved mapping visually thicker regions from the original contour image to areas of increased stroke width. This would have been achieved

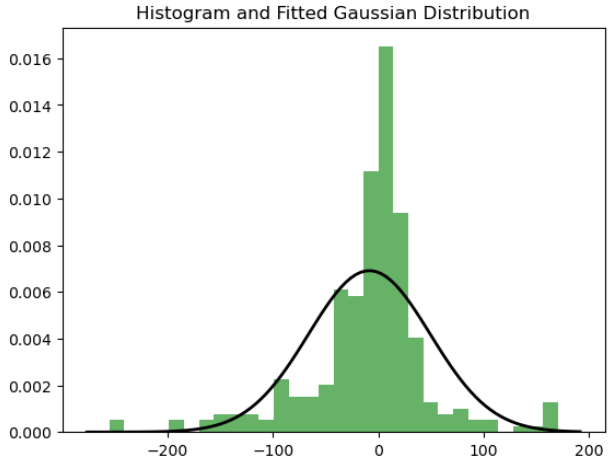


Fig. 6. The distribution of Menger Curvatures of our bunny sketch used in determining areas of thicker stroke width.

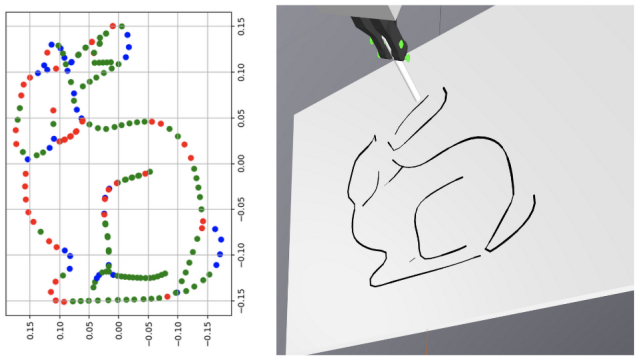


Fig. 7. On the left, we can see the plot of our bunny where red dots represent all curvatures half a standard deviation above the mean curvature, and blue dots represent curvatures half a standard deviation below the mean curvature, and all other dots are green. On the right, we see an example sketch that modifies stroke width in areas of high curvature.

by analyzing the density of line segments in specific areas, extracted from parsing the Bézier curves in the SVG file. We opted instead for a more creative solution inspired by the dynamic nuances of brush strokes on a canvas. When painting on a canvas, variations in the angle and pressure of the brush result in nuanced strokes, enhancing the aesthetic quality of the artwork.

To mirror these expressive strokes, our solution calculates the curvature of line segments generated for the drawing, utilizing the Menger curvature calculation. Menger curvature in n-dimensional Euclidean space is defined as the reciprocal of the radius of the circle passing through three points. In simple terms, curvature measures the extent to which a geometric object deviates from being a straight line.

Our method takes in the array of coordinates representing our drawing, computing the Menger curvature for each point in the input list (excluding the first and last points). For each point i the neighboring $i+2$ and $i-2$ points are used in deriving the curvature. Once curvature values are derived for all points, a statistical analysis is conducted to extract the mean and standard deviation from the resulting Gaussian distribution (see Figure 6). We then use them to derive regions of our drawing that are higher in curvature compared to the rest of the drawing. As shown in figure 7, areas of "higher curvature" will be mapped to varying width in the sketch as indicated by the blue and red dots.

*3) **Finding Optimal Z-values**:* To map force to stroke width through varying z-values, we generate a list of delta-z values adding to the z coordinates of the key frame poses of the drawing trajectory. The smaller the z-value, the greater the force being applied, thus the thicker the stroke width. Using the aforementioned extracted areas of higher curvature, our initial attempt was to change the z-value for those coordinates and their 2 nearest neighbors to the left and right, to be a smaller value. Specifically, the default z-value for the pose accounts for the offset from the canvas thickness and the chalk length. We reduced it by approximately 0.004m to exert greater force for three consecutive poses, thereby creating a thicker width in the drawing. From the rendered output, (Fig. 8) the drawing was choppy and segmented and failed to mirror the smooth fluctuation in width seen in natural brush strokes that we desired.

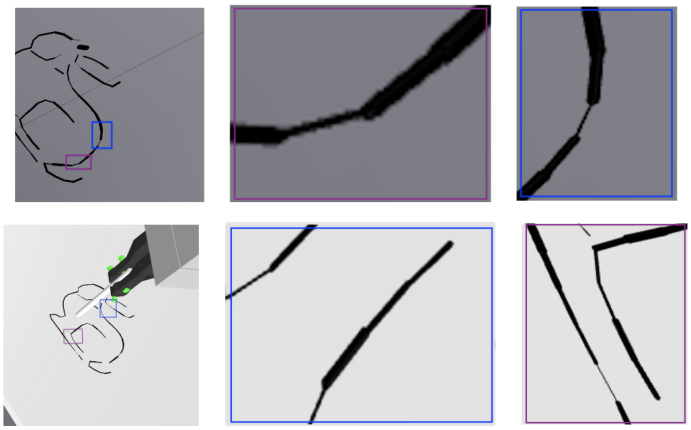


Fig. 8. Top: The rendered drawing using our initial method of changing z-values to generate thicker lines only in the 3 point neighboring region of areas of higher curvature. The middle and right images are zoomed-in portions of our bunny sketch with varying widths. Bottom: The rendered drawing using a method of logarithmically increasing and decreasing z-values to generate varying widths.

To achieve a more natural-looking variation in stroke width akin to artistic paintings, we adopted a final method. This method generated a list of linearly decreasing and then increasing z-values, creating a gradual increase and decrease in force and, consequently, a smoother width change. While we also explored an array of logarithmically increasing and decreasing

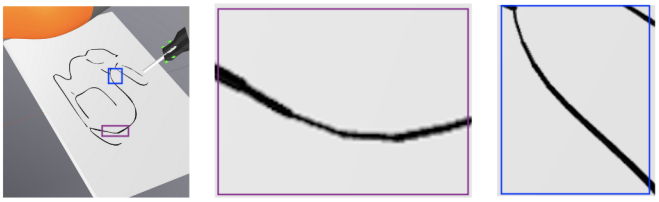


Fig. 9. The rendered drawing using our final method of linearly increasing and decreasing z-values to generate more natural-looking varying widths.

z-values, hoping to improve the force-to-width mapping, we observed that this approach resulted in choppy segments similar to our initial proposal (Fig. 8). After multiple iterations, we identified the minimum delta z-value that allowed the chalk to penetrate the paper without violating the simulation's physics (such as dragging or passing through the paper), which was decreasing by 0.005m. We applied this list of decreasing and increasing z-values to all the previously identified areas of higher curvature, while the remaining values retained the default.

D. Results Demonstration

In this section, we present the journey that led us to achieve our final best drawing with the SketchBot(Fig. 10). Throughout our project, we continually refined our techniques, from image-to-sketch simplification to dynamic stroke width mapping, and z-value optimization to enhance the drawing capabilities of our robot painter.

V. Discussion and Future Work

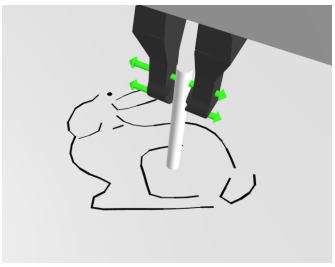
This paper provides a framework for the robotic drawing of 3D models with fine details, utilizing varying stroke width to create pleasing drawings. Our approach has 3 key steps:

1) Image to sketch simplification utilizing open source models to create contour images suitable for drawing
2) Vectorizing contours and generating optimal trajectories for drawing
3) Mapping and fine-tuning the robot's motion in the z-axis to create varying stroke widths throughout the drawing

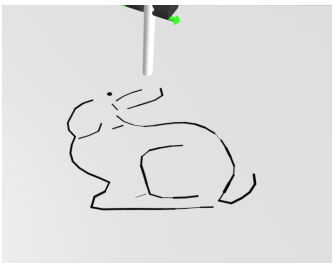
While this paper serves as a proof of concept, certain limitations exist within our current approach, paving the way for future improvements and extensions.

Currently, our approach is tailored to the drawing of the bunny model featured in this paper. Certain sections of the code have been hardcoded to accommodate the bunny implementation, resulting in a lack of seamless integration in the image processing-to-drawing pipeline. A specific limitation lies in our dependence on Typescript libraries for performing skeletonization via the Medial Axis Transform. The current setup requires manual intervention to run Typescript code separately from the main Python codebase. To enhance the pipeline's usability and universality, future efforts should involve implementing the skeletonization step in Python, eliminating the need for external dependencies. Future work should focus on generalizing our approach to accommodate any 3D object or scenes with multiple 3D objects.

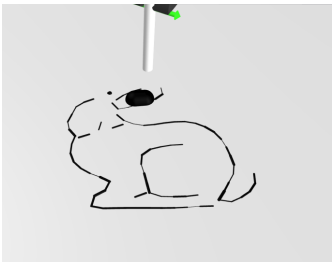
In addition, further exploration into stroke capabilities is warranted. Techniques that involve combining line widths from the SVG output with curvature information could yield more sophisticated stroke mapping, resulting in drawings with enhanced visual aesthetics. This exploration could lead to the



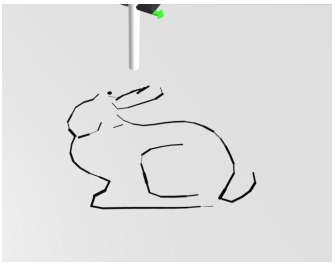
(a) Linearly changing z-values with 4 neighboring points used for each inflection point.



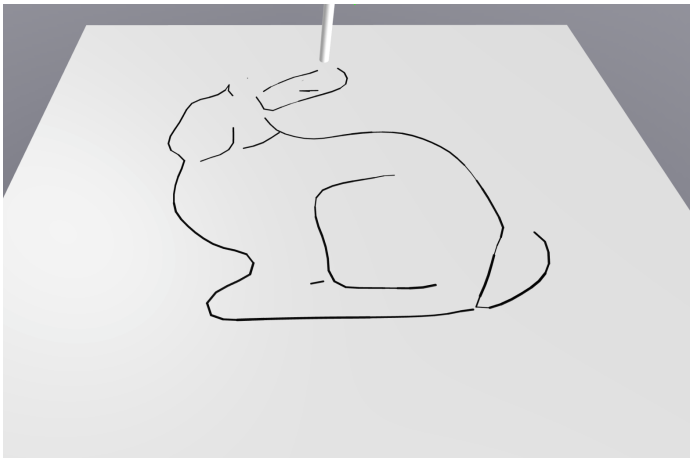
(b) Logarithmically changing z-values with 4 neighboring points used for each inflection point with a minimum z-value of 0.186m.



(c) Logarithmically changing z-values with 5 neighboring points used for each inflection point with a minimum z-value of .184m.



(d) Linearly changing z-values with 6 neighboring points used for each inflection point with a minimum z-value of 0.185m.



(e) Linearly changing z-values with 3 neighboring points on each side of the inflection point with a minimum z-value of 0.185m.

Fig. 10. The rendered drawings above display the various iterations of our method for generating varying widths. The parameters that were tuned include the number of neighboring points used for each point of high curvature, whether the z-values changed linearly or logarithmically, and the minimum z-value for which the chalk would reach (i.e. exert a higher force and thus generate a thicker width). The parameters are detailed in the captions of each image.

development of more advanced and pleasing stroke techniques for robotic drawing.

In conclusion, while our current framework demonstrates the feasibility of robotic drawing with varying stroke widths, addressing the outlined limitations and pursuing the suggested future directions will contribute to a more versatile, user-friendly, and visually compelling robotic drawing system.

ACKNOWLEDGMENTS

We would like to thank Professor Russ Tedrake for his help and guidance throughout this project. His responsiveness and willingness to help propelled us to success in our robotic drawing endeavors. We would also like to thank our communication instructor Nora Jackson for providing valuable feedback and support when creating the narrative for our project. Lastly, we would like to recognize the course staff for 6.4210 for their support throughout the course.

GROUP CONTRIBUTIONS

Zhishen Chen Setting up scene and local development, Generating key pose frame from vectorized sketch, Inverse kinematics control

Maura Kelleher Extracting detailed contours: camera image to sketch to simplified contours to SVG conversion, Mapping hydroelastic contact force to width

Shayda Moezzi SVG Processing, Skeletonization of SVG via MAT, Menger Curvature implementation, Fine-tuning z-values for Stroke Width

REFERENCES

[1] M. Lam and J. Muguira, “DaVinci Robot painter,” YouTube, 2021. Available: https://www.youtube.com/watch?v=szI8pAFz2Fo.

[2] A. Saravanan and V. Mukkamala, “Michelangelo: Drawing Silhouettes of Objects Using Image Perception, Point Cloud Transformation, and Motion Planning,” 6.4210/6.4212 Robotic Manipulation, Fall 2022.

[3] H. Jin, M. Lian, S. Qiu, X. Han, X. Zhao, L. Yang, Z. Zhang, “A Semi-automatic Oriental Ink Painting Framework for Robotic Drawing from 3D Models”, IEEE Robotics and Automation Letters, 2023.

[4] P. Schaldenbrand, J. McCann, J. Oh, “FRIDA: A Collaborative Robot Painter with a Differentiable, Real2Sim2Real Planning Environment” in arXiv preprint arXiv, arXiv:2210.00664, 2022.

[5] A. Wacker, ”Image-to-Line-Drawings,” Hugging Face, 2023. Available: https://huggingface.co/spaces/awacke1/Image-to-Line-Drawings/tree/main.

[6] H. Yu, ”Image-to-sketch,” Hugging Face, 2023. Available: https://huggingface.co/spaces/hossay/image-to-sketch/tree/main.

[7] E. Simo-Serra, S. Iizuka, K. Sasaki, and H. Ishikawa, “Learning to simplify,” ACM Transactions on Graphics, vol. 35, no. 4, pp. 1–11, 2016. doi:10.1145/2897824.2925972

[8] P. Selinger, Potrace, Available: https://potrace.sourceforge.net/

[9] H. Choi, S. Choi, ”New Algorithm for Medial Axis Transform of Plane Domain”, Journal of Graphical Models and Image Processing, vol. 59, no. 6, pp. 463-483, 1997.