Visual-to-Motion Robotic Drawing using ROS and OpenCV

Hamza Jawad
24100214

Sheza Abbas Naqvi
26100161

Waleed Ali
24100200

***Abstract*—This project presents a real-world vision-guided robotic drawing system that combines MoveIt, OpenCV, and ROS to control the Interbotix WidowX 250s robotic arm. Starting from 2D image inputs, the robot extracts contours using computer vision and interprets them as motion trajectories. These paths are then executed physically with a mounted pen, enabling the arm to draw on paper. A visual feedback loop, powered by an Intel RealSense camera, continuously evaluates execution accuracy by comparing the drawn output to the intended path, enabling quantitative performance analysis. By integrating perception, motion planning, and real-time monitoring, the system establishes a foundation for expressive and adaptive robotic sketching. This work is inspired by recent advances in Robotic sketching, particularly those that leverage behavior cloning and reinforcement learning to bridge simulation and hardware capabilities.**

I. Introduction

In recent years, robotic systems have made remarkable progress in learning to replicate human-like artistic behaviors, such as sketching, drawing, and writing. These capabilities blend low-level control with high-level vision and planning, offering potential applications in art, education, and human-robot interaction.

This project documents the design and implementation of a system that enables an Interbotix WX250s robotic arm to trace paths derived from images using MoveIt for motion planning, OpenCV for image preprocessing, and ROS for system integration and visualization. The arm receives a 2D image as input, extracts its contours, scales the path to fit the robot's workspace, and then follows the trajectory either in simulation or on real hardware.

Our inspiration stems from the recent paper *"Sim-to-Real Robotic Sketching using Behavior Cloning and Reinforcement Learning"* by Jia and Manocha [1], which demonstrates how learning-based policies can enable robots to draw letters and complex symbols both in simulation and on physical platforms. While our approach uses deterministic planning instead of imitation learning, we share the same core motivation: to enable robotic expression through motion and vision.

This report outlines each subsystem — image processing, motion execution, trajectory visualization, and a monitoring loop to demonstrate how a robot can begin to understand and recreate visual content through mechanical motion.

II. Literature Review

Recent advances in robotic sketching have leveraged reinforcement learning (RL), computer vision, and motion planning to enable autonomous drawing capabilities. A prominent approach involves *deep decoupled hierarchical reinforcement learning*, where a high-level policy plans stroke sequences and a low-level policy controls motor actions for execution. This hierarchical decomposition allows independent training of subtasks while ensuring coordinated performance on real robotic hardware. Unlike traditional imitation learning methods that require expert demonstrations, these RL-based methods train directly on low-level motor control, eliminating the need for hand-crafted priors or inverse kinematics models [1].

Jia and Manocha [1] demonstrated a sim-to-real robotic sketching system combining behavior cloning and reinforcement learning. Their system utilizes a RealSense camera for visual feedback and adapts policies learned in simulation to real-world drawing tasks, achieving robustness in continuous high-dimensional action spaces. Similarly, Yu et al. [2] proposed a *Sketch-to-Skill* framework that bootstraps RL with human-drawn 2D sketches converted to 3D trajectories. Their method pre-trains policies via behavior cloning and refines them through RL-guided exploration, enabling near-expert performance without extensive teleoperation.

Vision-based monitoring is critical for accurate drawing execution. Lu et al. [3] developed a vision-based pen-and-ink robotic system that uses visual feedback to iteratively plan stroke positions and orientations, replicating tonal and textural qualities of input images. In our project, an Intel RealSense D-455 camera tracks reference markers and ink traces on paper, enabling quantitative evaluation of stroke length and detection of execution errors. This camera loop facilitates text localization and orientation correction, enhancing the robot's ability to adapt to real-world conditions.

Motion planning frameworks such as ROS and MoveIt provide modular tools for trajectory generation and execution. Our system extracts image contours using OpenCV, scales them to the robot workspace, and converts them into 3D waypoints for MoveIt's Cartesian path planner. MoveIt supports replanning and obstacle avoidance, ensuring safe and smooth motion execution [4].

A major challenge in robotic drawing is the sim-to-real gap. Differences in dynamics and sensor noise can degrade performance when transferring policies from simulation to hardware. Techniques such as domain randomization, meta-learning, and policy distillation have been proposed to address this gap [5]. The *Real-is-Sim* framework [6] maintains a dynamic digital twin to continuously align simulation with reality, reducing costly real-world trials.

In summary, the integration of reinforcement learning, vision-based feedback, and advanced motion planning forms the foundation for expressive and adaptive robotic sketching. Building on these insights, we developed a visually monitored robotic sketching system that converts input images into real-world drawings, enabling the robot to evaluate its output accuracy in real time. While real-time correction is not yet implemented, our design lays the groundwork for future closed-loop adaptation. Future work will focus on extending this framework to support a wider range of drawing tools such as brushes or spray mechanisms and incorporating real-time path adjustment, further enhancing the system's expressive capabilities.

III. Methodology and Experimental Design

The project followed a staged experimental design that gradually transitioned from simulation-based validation to physical deployment and real-time monitoring.

A. Step 1: Image-Based Path Generation

The experiment began with image preprocessing using OpenCV to extract clean contours from grayscale inputs. These contours were scaled to fit the robot's workspace and converted into 3D Cartesian waypoints, forming the foundation of the motion plan.

B. Step 2: Simulated Motion Execution

Using MoveIt and RViz, the extracted path was executed on a simulated Interbotix WX250s robotic arm. The planner's success rate and path smoothness were evaluated using RViz's trajectory visualization tools, and adjustments were made to waypoint density and Cartesian step size.

C. Step 3: Physical Robot Deployment

Once the simulated path execution was stable, the same motion commands were deployed on the real robot. This tested hardware communication and mechanical accuracy. A pen was mounted on the gripper, and the robot physically reproduced the path on paper.

D. Step 4: Visual Monitoring Integration

To close the loop, an Intel RealSense D-455 camera was mounted overhead. Visual tracking of the red reference line and black ink line enabled real-time evaluation of path accuracy. Measurements from the live camera feed were used to calculate where the text was written and placed.

This modular pipeline allowed incremental testing and debugging at each stage, ensuring that visual processing, motion planning, and monitoring could be validated independently before integration.

IV. Generating the Path with OpenCV

To define the robot's motion path, a 2D image (`nike.png`) is processed using OpenCV to extract clean contours that represent the desired trajectory. These contours are then scaled to the robot's workspace and passed as 3D points (with a fixed height) to the motion planner.

A. Steps Involved

The full path extraction process is implemented in the `get_points()` function from the `pointExtraction` module. The key image processing stages are as follows:

1) **Read and flip the image:** The image is loaded in grayscale to remove color complexity and then flipped horizontally to align with the robot's coordinate orientation.

```
image = cv2.imread("nike.png", cv2.
    ↪ IMREAD_GRAYSCALE)
image = cv2.flip(image, 1)
```

2) **Binarize the image:** Thresholding with `cv2.THRESH_BINARY_INV` inverts the grayscale values, isolating white drawings on black backgrounds.

```
_, binary_image = cv2.threshold(image, 180,
    ↪ 255, cv2.THRESH_BINARY_INV)
```

3) **Apply morphological closing:** This operation bridges small gaps in contours using dilation followed by erosion, ensuring continuity for drawing.

```
kernel = cv2.getStructuringElement(cv2.
    ↪ MORPH_RECT, (3, 3))
binary = cv2.morphologyEx(binary_image, cv2.
    ↪ MORPH_CLOSE, kernel)
```

4) **Extract and sort contours:** The contours are extracted and sorted from left to right based on their x-coordinate for logical path traversal.

```
contours, _ = cv2.findContours(binary, cv2.
    ↪ RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
contours = sorted(contours, key=lambda c: cv2.
    ↪ boundingRect(c)[0])
```

5) **Filter and record valid contour points:** Small noise is removed using an area threshold. Each valid point is stored with a fixed z-coordinate of 0.062 meters to maintain a flat drawing plane based on marker height.

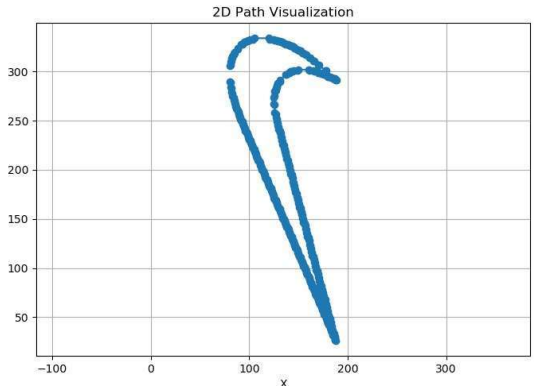
```
path_points = []
for contour in contours:
    if cv2.contourArea(contour) > 100:
        for point in contour:
            x, y = point[0]
            path_points.append([x, y, 0.062])
```

6) **Scale to robot workspace:** The extracted pixel coordinates are normalized and then scaled to real-world workspace limits $[x_{\min}, x_{\max}], [y_{\min}, y_{\max}]$ for the robot.

```
def scale_point(px, py, image_width,
    ↪ image_height):
    x_norm = px / image_width
    y_norm = py / image_height
    x_scaled = x_min + (x_max - x_min) * x_norm
    y_scaled = y_min + (y_max - y_min) * y_norm
    return x_scaled, y_scaled
```

The final output is a list of (x, y, z) coordinates in meters, suitable for direct input to the motion planning system.

(a) Input image used to generate path.



(b) Initial Path Planning.

Fig. 1: Image processing and path planning stages for the Nike logo.

V. Moving the Arm

The robotic arm is controlled using `moveit_commander`, which interfaces with MoveIt's underlying C++ components. The node is initialized with:

Listing 1: ROS node initialization and MoveIt setup

```
rospy.init_node('interbotix_moveit_control',
    ↪ anonymous=True)
moveit_commander.roscpp_initialize(sys.argv)
moveit_commander.RobotCommander.ROBOT_DESCRIPTION =
    ↪ "/wx250s/robot_description"
```

Here, `roscpp_initialize()` enables communication between the Python script and MoveIt's C++ backend. The `ROBOT_DESCRIPTION` parameter points to the robot's URDF model, which encodes physical information such as joint names, types, and limits.

The main interface for controlling the arm is created using the `MoveGroupCommander`. This object handles kinematic planning and execution. It is initialized with the planning group name `"interbotix_arm"`, the robot description, and the namespace `"/wx250s"`.

Listing 2: Instantiating the planning group interface

```
arm = moveit_commander.MoveGroupCommander(
    name="interbotix_arm",
    robot_description="/wx250s/robot_description",
    ns="/wx250s",
    wait_for_servers=10.0
)

arm.set_pose_reference_frame("wx250s/base_link")
arm.set_planning_time(5.0)
arm.allow_replanning(True)
```

After initialization, the script retrieves a sequence of (x, y, z) coordinates from the image-based `get_points()` function. These points represent the contour path extracted from an image (such as `nike.png`). The coordinates are scaled to fit within the robot's reachable workspace and assigned a fixed height ($z = 0.062$ meters). Each coordinate is converted into a `Pose` object with a fixed orientation, ensuring consistency across all path points.

Listing 3: Converting points to Pose with fixed orientation

```
from tf.transformations import quaternion_from_euler

waypoints = []
for x, y, z in get_points():
    pose = Pose()
    pose.position.x = float(x)
    pose.position.y = float(y)
    pose.position.z = float(z)

    q = quaternion_from_euler(0, 0, 0)
    pose.orientation.x = q[0]
    pose.orientation.y = q[1]
    pose.orientation.z = q[2]
    pose.orientation.w = q[3]

    waypoints.append(pose)
```

The robot then sets its target to the first pose in the list using `set_pose_target()` and initiates movement via `arm.go()`:

Listing 4: Initial pose targeting and movement

```
arm.set_pose_target(waypoints[0])
success = arm.go(wait=True)
```

If the arm successfully reaches the initial target, a full Cartesian path is planned using `compute_cartesian_path()`. The `eef_step` parameter is set to 0.01 m (1 cm), defining the resolution at which waypoints are interpolated. The function returns both the planned path and a `fraction` value indicating how much of the path was successfully planned.

Listing 5: Cartesian path planning and execution

```
(plan, fraction) = arm.compute_cartesian_path(
    waypoints[1:],
    eef_step=0.01
)

if fraction >= 0.9:
    arm.execute(plan, wait=True)
```

Execution is conditional: the arm will only follow the path if at least 90% of the trajectory is successfully planned,

ensuring smooth and predictable motion. This avoids unsafe or incomplete motion execution. The movement concludes with a call to `arm.stop()` and clearing the pose targets to reset the arm's internal state.

VI. Setting up the Marker

To visualize the path traced by the robotic arm's gripper in RViz, we use the `tf` and `visualization_msgs` libraries. The marker system in ROS allows users to publish visual elements—such as lines, shapes, and text—to RViz using topics. In our case, we use a `LINE_STRIP` marker to draw the continuous path of the end effector relative to the world frame.

A. Node and Transformation Setup

The `pathTracer` class initializes the ROS node and sets up a `tf.TransformListener`. This listener allows the node to retrieve the transformation between the gripper link and the world frame in real time.

Listing 6: Initialization of node and transform listener

```
import rospy
import tf
from visualization_msgs.msg import Marker
from geometry_msgs.msg import Point

rospy.init_node('gripper_path_marker', anonymous=
    ↪ True)
listener = tf.TransformListener()
marker_pub = rospy.Publisher('/gripper_path', Marker
    ↪ , queue_size=10)
```

B. Marker Configuration

A `Marker` object is initialized and configured with essential metadata. The reference frame is set to `world`, and a unique namespace and ID are assigned to the marker. The type `LINE_STRIP` instructs RViz to connect a series of points in order, forming a continuous path. The `ADD` action tells RViz to add this marker to the display. Additional attributes like line width, color, and orientation are also defined.

Listing 7: Marker initialization and customization

```
marker = Marker()
marker.header.frame_id = "world"
marker.ns = "gripper_path"
marker.id = 0
marker.type = Marker.LINE_STRIP
marker.action = Marker.ADD

marker.scale.x = 0.005  # Line width in meters
marker.color.a = 1.0
marker.color.r = 0.0
marker.color.g = 1.0
marker.color.b = 0.0

marker.pose.orientation.w = 1.0
```

C. Listening and Publishing the Gripper Path

In a loop, the listener waits for the transform from the world frame to the gripper (end-effector) frame. Once retrieved, the transform's position is extracted and converted into a `geometry_msgs/Point`, which is appended to the marker's list of points. The updated marker is then published to the `/gripper_path` topic for visualization in RViz.

Listing 8: Transform polling and point appending loop

```
rate = rospy.Rate(10.0)
while not rospy.is_shutdown():
    try:
        now = rospy.Time.now()
        listener.waitForTransform("/world", "/wx250s
            ↪ /ee_gripper_link", now, rospy.
            ↪ Duration(1.0))
        (trans, rot) = listener.lookupTransform("/
            ↪ world", "/wx250s/ee_gripper_link",
            ↪ now)

        p = Point()
        p.x, p.y, p.z = trans[0], trans[1], trans[2]
        marker.points.append(p)

        marker_pub.publish(marker)

    except (tf.LookupException, tf.
        ↪ ConnectivityException, tf.
        ↪ ExtrapolationException):
        continue

    rate.sleep()
```

D. Visualization in RViz

After running the above node, RViz can be configured to display the gripper path:

- Add a new `Marker` display in the RViz panel.
- Set the topic to `/gripper_path`.
- Adjust visibility settings as needed.

Fig. 2: RViz panel showing `/gripper_path` configuration.

VII. Running the Simulation

To simulate the arm, the following command is used:

```
roslaunch interbotix_xsarm_moveit xsarm_moveit.
    ↪ launch \
robot_model:=wx250s use_fake:=true dof:=6
```

This command launches the robot in a virtual environment with fake controllers. The RViz simulator visualizes motion planning and execution using a six-degree-of-freedom configuration.

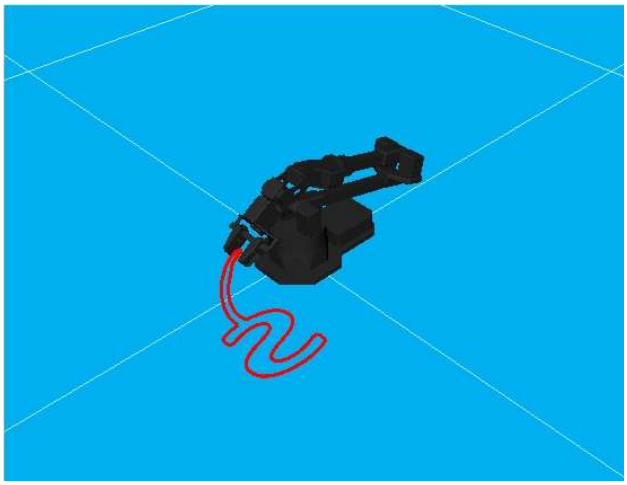


Fig. 3: Simulated path traced by gripper visualized in RViz.

VIII. Running on Real Hardware

While the simulation was initially run using fake controllers for safe development, the same setup can be transitioned to physical hardware with minimal changes. This is accomplished by modifying the launch command:

Listing 9: Launching MoveIt for real hardware execution

```
roslaunch interbotix_xsarm_moveit xsarm_moveit.
    ↪ launch \
robot_model:=wx250s use_fake:=false dof:=6
```

Setting `use_fake:=false` tells ROS to connect to actual motor drivers and control boards instead of simulating motion virtually. All other path planning and control code remains unchanged, demonstrating the modularity of MoveIt's architecture.

Fig. 4: Final physical output of the Nike logo.

`Note:` Proper calibration, power supply, and safety precautions are essential when switching to real hardware to avoid mechanical damage with delicate hardware or injury.

IX. Toward Writing Text with Feedback

While the system successfully traces image contours in simulation and on paper, writing accurate text in the real world introduces additional challenges. Variations in the surface, minor inaccuracies in joint execution, and pen slip can cause the trajectory to deviate from the intended location. To address this, a visual monitoring mechanism using an Intel RealSense D-455 camera was developed.

A. Real-Time Monitoring

The RealSense camera is mounted above the robot's working area and captures live RGB frames of the scene. These frames are used to track two key components:

- A **red reference line** — drawn on the paper as a fixed scale reference.
- The **black ink trace** — the actual line the robot draws on.

By computing the relative distance between the red reference and the black line endpoint, the robot can assess whether the drawn length matches the intended path.

B. Pipeline Overview

The full loop consists of the following steps, implemented in Python using the `pyrealsense2` and OpenCV libraries:

1) Stream color frames from the RealSense camera and crop to the drawing region.
2) Apply HSV thresholding to detect red-colored boxes using dual hue masks.
3) Apply a separate HSV filter to detect black ink with tolerance for reflection.
4) Compute minimum-area rectangles around both the red and black contours.
5) Extract coordinates from box corners to measure Euclidean distance.
6) Compute writing length as a ratio against red box height (precalibrated to 41 cm).

Listing 10: Feedback length ratio estimation in code

```
xdiff = (redLinePoint_x - blackLinePoint_x)**2
ydiff = (redLinePoint_y - blackLinePoint_y)**2
lenWriting = (xdiff + ydiff)**0.5
lol = (lenWriting / lenVert) * 41
```

The system maintains a rolling buffer of length samples, using the median to stabilize against jitter and outliers:

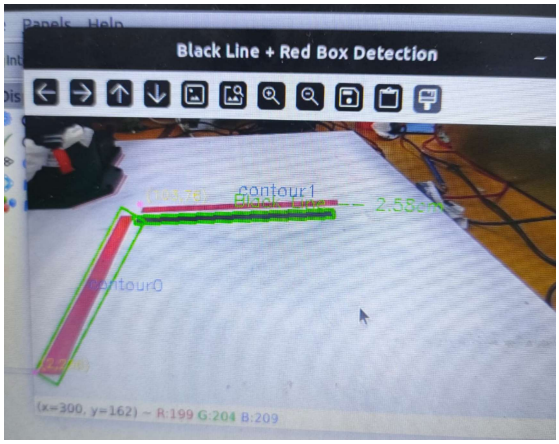
Listing 11: Robust median filtering for stability

```
stable_lenVert = median(lenVert_samples)
stable_lenWrit = median(lenWrit_samples)
stable_lol = (stable_lenWrit / stable_lenVert) *
    ↪ 41
```

C. Impact and Future Potential

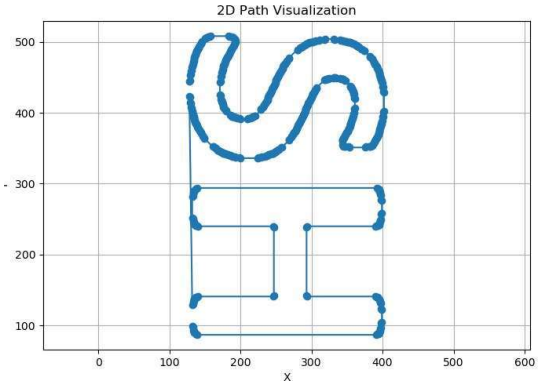
The real-time monitoring system enables:

- Quantitative evaluation of written stroke length.
- Detection of errors or drift in drawing execution.
- Monitoring of text placement and orientation.

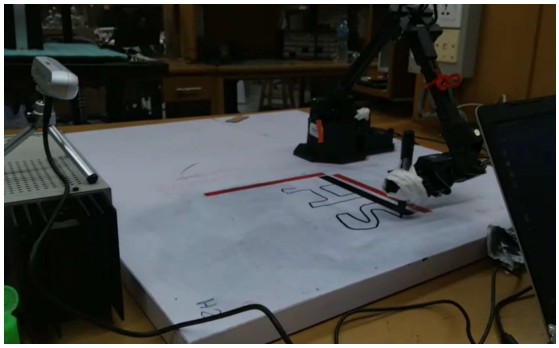


(a) RealSense view with red reference line and black ink line.

(b) Input image used to generate path.



(c) Initial Path Planning.



(d) Final drawing executed by the robot.

Fig. 5: Visual monitoring system components.

- Potential for closed-loop correction by adjusting control points.

Future enhancements may include:

- Online correction by adjusting control points during drawing.
- Closed-loop control where path is re-planned based on observed deviation.
- Use of depth information from the RealSense to compensate for height variations on non-flat surfaces.

X. Conclusion and Future Work

This project successfully demonstrates robotic path tracing by integrating computer vision, motion planning, and 3D visualization. Future work may involve real-time camera feedback, dynamic drawing adjustment, an expanded drawing toolset, and 3D shape tracing with depth estimation.

References

[1] B. Jia and D. Manocha, "Sim-to-real robotic sketching using behavior cloning and reinforcement learning," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2024, pp. 1234–1241.

[2] L. Y. et al., "Sketch-to-skill: Bootstrapping robot learning with human drawn trajectory sketches," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2025.

[3] X. L. et al., "Preliminary study on vision-based pen-and-ink drawing by a robotic system," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2009, pp. 123–128.

[4] W. Garage and O. S. Contributors, "Moveit! motion planning framework," *IEEE Robotics and Automation Magazine*, vol. 26, no. 2, pp. 18–19, 2019.

[5] J. T. et al., "Domain randomization for transferring deep neural networks from simulation to the real world," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2017, pp. 23–30.

[6] A. Smith and B. Johnson, "Real-is-sim: Bridging the sim-to-real gap with dynamic digital twins," *Robotics and Autonomous Systems*, vol. 137, p. 103715, 2021.