Who Stole My Pen?

Overview

We've all done it. "Borrowed" a pen and then lost it. Let's see if we can get PincherX 100 to hold on to it for us!

The goal is enable the PincherX 100 to grab a pen that is held in front of it.

Setup

Software

The software setup was completed as part of Computer Setup (./computer_setup.html).

Physical

The physical setup for this project requires the Trossen PincherX 100 and the Intel Realsense D435i. Here are some tips for how to locate the hardware.

1. The field of view of the RealSense should substantially overlap with part of the PincherX's workspace
2. Try to keep the PincherX away from walls and valuables; (you never know when it will go rogue!)
3. I've found the most compact setup is to offset the D435i 90 degrees from the front of the PincherX
4. Part of the challenge is to place the robot and sensor in appropriate locations. Depending on how you do this, the algorithm may have different parameters.
5. Remember to plug the power cable into the PincherX 100.
6. The RealSense has a tripod you can use to keep it in one location (also, the legs of the tripod extend, if needed).
    - An alternative is to place the RealSense on top of it's box.

Pen Recognition

Use the RealSense to measure the 3D location of your purple MSR pen. This guide provides steps that I consider to be the simplest that can minimally accomplish the accuracy needed to complete the challenge. However, feel free to use any other techniques that you see fit.

The approach is to use classical computer vision techniques on the RGB image to locate the pen in 2D space. We then align the Depth map (an image where each pixel is a depth measurement) to the RGB image and use the pen location as a mask to get the 3D information. Finally, we find the centroid (and possibly other geometric moments) of the pen. This information will be fed into a controller that will enable the robot to grab the pen.

Alignment

1. Alignment is the process of placing two images, taken from different perspectives into the same reference frame
2. In this case, we collect two images: a depth map and a Red, Green, Blue (RGB) image
    - The Depth Map is an `NxM` array, where each "pixel" is a 16-bit integer proportional to the distance from the camera to the nearest object.
    - The RGB image is an `NxMx3` array, where each `NxM` slice is a different R, G, or B color (be careful, OpenCV often uses BGR instead of RGB ordering for colors!)
3. This Alignment Example (https://github.com/IntelRealSense/librealsense/blob/master/wrappers/python/examples/align-depth2color.py) from Intel demonstrates how to do several tasks and is a good starting point for your project
    1. Stream depth and RGB images from the camera
    2. Convert the depth "pixel" value into meters
        - The example finds a scaling factor and manually converts. This scaling factor is useful if you want to, for example, convert the entire depth-image to meters.

- The depth image also has a get_distance (https://intelrealsense.github.io/librealsense/python_docs/_generated/pyrealsense2.depth_frame.html function that does the conversion for a single pixel.

3. Remove objects that are farther than a distance threshold away
4. Align both images
5. Process the RGB image with OpenCV
6. Display the results with OpenCV

Design Notes

1. What follows are some notes on how to think about designing and organizing your code:
2. A good strategy, after running and understanding the Alignment Example (https://github.com/IntelRealSense/librealsense/blob/master/wrappers/python/examples/align-depth2color.py) is to divide it into the components that you need and encapsulate them in a class.
3. For example, there are a few steps required to capture images:
   - Setup streaming (this can be done, for example, in the `__init__` method).
   - Capture a frame, get the depth image, etc. (these are class methods)
   - Stop the pipeline and cleanup.
   - By creating a class you can separate two concerns: the actual mechanics of obtaining a frame, and the continuous processing of the frame (as done by the `while` loop in the example)
4. In python the `with` statement is useful for using objects that must initialize a resource (e.g., a file) and then clean it up when finished.
   - With statements work with Context Managers (https://docs.python.org/3/reference/datamodel.html#context-managers) to enter and exit a *runtime context*
   - In this case, entering the context can be seen as setting up the streaming pipeline, and leaving the context shuts the pipeline down.
   - The general idea is that, even if code within the `with` block throws an exception, the cleanup code will run automatically (without requiring `try-finally`)
5. A good way to test this class is to use it to write a script that displays the aligned depth and RGB images to the user.
6. The OpenCV GUI will not update unless events are processed. Two functions are available to process events:
   - `cv2.waitKey` will wait for a key to be pressed for some specified number of milliseconds (the minimum is 1). Run `help("cv2.waitKey")` for more information.
   - `cv2.pollKey` updates the event loop and checks if a key was pressed since it's last invocation (so it does not wait). Run `help("cv2.pollKey")` for more information.
   - It is necessary to call one of these functions periodically in order to have a responsive GUI: the most straight-forward way is for the bulk of your program to be designed to operate in a loop where one of these functions are called once per iteration.

Resources

1. Realsense Python API Documentation (https://intelrealsense.github.io/librealsense/python_docs/)
2. PyRealSense Examples (https://github.com/IntelRealSense/librealsense/tree/master/wrappers/python/examples)

Recording

- Being able to record data and play it back is useful when testing an image pipeline because it ensures that you are using the same data each time.
- To record data, use config.enable_record_to_file(filename) (https://intelrealsense.github.io/librealsense/python_docs/_generated/pyrealsense2.config.html#pyrealsense2
- To play back the data, use the config.enable_device_from_file(filename) (https://intelrealsense.github.io/librealsense/python_docs/_generated/pyrealsense2.config.html#pyrealsense2
- You may wish to provide command-line arguments to easily switch between recording and playback: the argparse (https://docs.python.org/3/library/argparse.html) library may be helpful.

Design Notes

1. By encapsulating the camera streaming in a class, it is easier to write a program that lets a user choose between playing from a recording or streaming live data.

2. The recording and playback functionalities are useful to implement before moving forward with the image pipeline.

OpenCV (index.html) | HOME (./index.html)

Image Pipeline

1. We will use OpenCV to process the images from the Realsense.
2. A complete list of python tutorials is here (https://docs.opencv.org/4.6.0/d6/d00/tutorial_py_root.html).
3. It is useful during debug to visualize each step in an image processing pipeline
   - Create a new window using `cv2.imshow("name", image)` or `cv2.newWindow()`
   - First, get something reasonable, than worry about refining your object detection.
4. You can use any image processing methods that you want.
   - I've listed some techniques to get you started
   - The end goal of this pipeline is to output the centroid of the pen
5. The Depth map is calibrated by Intel and you can directly convert to real units (as the alignment example does)
   - You can use camera calibration and the known positions of the robot and RealSense to solve for the other dimensions.
   - The Realsense's built-in calibration is not great so for most projects you should do a calibration yourself: however, it is good enough for this project.
   - More information on calibration is under Useful Techniques
6. The output of the image pipeline should be the (x, y, z) location, in meters, relative to the camera frame.
   - Be sure to display or print this information, as it will be needed later.

Useful Techniques

1. Trackbars (https://docs.opencv.org/4.6.0/da/d6a/tutorial_trackbar.html)
   - Trackbars let you set parameters in a GUI in real time. Useful when trying to tweak parameters when processing an image.
   - For each step of the pipeline, you can create trackbars to control the basic parameters and experiment until you find ones that work
2. Convert between color spaces (https://docs.opencv.org/4.6.0/df/d9d/tutorial_py_colorspaces.html)
   - When distinguishing objects based on colors, it is often useful to convert to the Hue, Saturation, Value (HSV) colorspace.
     - In this space, the part of the light that contributes to color is relatively isolated from other factors such as brightness
   - In OpenCV, The Value (V) is between 0 and 255, Saturation (S) is between 0 and 255, and the Hue is between 0 and 180 (details (https://docs.opencv.org/4.6.0/de/d25/imgproc_color_conversions.html#color_convert_rgb_hsv))
3. Thresholding (https://docs.opencv.org/4.6.0/da/d97/tutorial_threshold_inRange.html)
   - Creates a mask image, with either black or white pixels depending on if the pixel in the original image met some criteria
   - More about Thresholding (https://docs.opencv.org/4.6.0/d7/d4d/tutorial_py_thresholding.html)
   - Use thresholding to mask off the pixels that are part of the pen.
4. Arithmetic Operations (https://docs.opencv.org/4.6.0/d0/d86/tutorial_py_image_arithmetics.html)
   - Images can be combined using arithmetic operations
   - Bitwise operations are especially useful for combining image masks, for example to see the RGB image only in the unmasked areas
5. How to find contours (https://docs.opencv.org/4.6.0/d4/d73/tutorial_py_contours_begin.html)
   - A contour is a closed geometric shape in an image
   - Getting the contour from the masked image of a pen gives you geometric information about the pen
6. How to get properties of contours (https://docs.opencv.org/4.6.0/dd/d49/tutorial_py_contour_features.html)
   - Useful for getting properties of contours, like area or the centroid
7. Filtering an image (https://docs.opencv.org/4.6.0/d4/d13/tutorial_py_filtering.html)
   - Computer vision sensors are noisy, these filters can help
   - Additional image filters (https://docs.opencv.org/4.6.0/db/df6/tutorial_erosion_dilatation.html)
   - More Filters (https://docs.opencv.org/4.6.0/d3/dbe/tutorial_opening_closing_hats.html)
8. Intel Documentation (https://github.com/IntelRealSense/librealsense/wiki/Projection-in-RealSense-SDK-2.0) describes various camera parameters and how to convert between pixel and real-world coordinates.
   - The approach I suggest uses a 2D color image and a depth map, but you may also be able to complete this project using point-clouds (a collection of 3-dimensional points).

- rs2_deproject_pixel_to_point (https://intelrealsense.github.io/librealsense/python_docs/_generated/pyrealsense2.html) can be used to to convert the depth map and pixel coordinates into real coordinates

```
rs.rs2_deproject_pixel_to_point(intrinsics, [px, py], depth)
# intrinsics - the intrinsic parameters
# (px, py) - the pixel coordinates
# depth - the depth in meters
# returns the x,y, and z coordinates in meters as a list
```

- The realsense knows its own intrinsic (https://intelrealsense.github.io/librealsense/python_docs/_generated/pyrealsense2.intrinsics.html?highlight=intrinsics) parameters. You can get them with the python api

```
cfg = pipeline.start(config)
profile = cfg.get_stream(rs.stream.color)
intr = profile.as_video_stream_profile().get_intrinsics()
```

- Given the location of the pincherX relative to the camera frame, you can convert camera (x,y,z) coordinates into cylindrical coordinates centered at the base frame of the pincherX
- For some theoretical knowledge about camera calibration
  - Camera Calibration (https://docs.opencv.org/4.6.0/d4/d94/tutorial_camera_calibration.html) explains how to convert pixel coordinates to real coordinates
  - Calibration (https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html) shows how to calibrate using OpenCV.

9. The depth map can be used to remove parts of the image that are clearly not in the workspace.
   - This technique is only available because we have depth information: it can reduce the burden of tuning the detector
   - The Alignment example code does this step.
10. For debugging it is sometimes useful to be able to draw (https://docs.opencv.org/4.6.0/dc/da5/tutorial_py_drawing_functions.html)

Design Notes

1. The ultimate goal of the pipeline is to get the 3D pen location from the RGB image and depth map.
2. Do not worry about precision and accuracy at this stage: just get a rough reasonable version working.
   - You do not know how accurate and precise the tracking needs to be to grab the pen until you work on the control
   - You can always come back and refine the tracking pipeline once it appears as a problem while attempting the control.
3. It may be helpful to give yourself the option to easily toggle debugging information on or off without modifying the code:
   - It is useful to be able to visualize each step in the pipeline
   - It is useful to be able to visualize important features on top of the original RGB image (such as the contour location).
4. It may be useful to write a class to manage trackbar settings

Robot Control

We will use the `interbotix_xs_toolbox` to control the robot. Although the arm can be fully controlled with ROS 2, we will use only minimally use it during the hackathon, instead relying on the simpler `python` API (which uses ROS 2 under the hood).

Caution

- See Specifications (http://www.support.interbotix.com/html/specifications/px100.html) for a list of joint ids and their limits
- If a joint limit is exceeded, it will hit a hard-stop and the motor will turn off and flash red.
- If a motor is flashing red, you should end the `ros2 launch` and power-cycle the robot
- The default `sleep` position
- See Troubleshooting Notes (https://www.trossenrobotics.com/docs/interbotix_xsarms/troubleshooting/index.html)
- Using the robot requires the `interbotix` workspace setup to be run in each terminal window you are using.

First Moves

UP (../index.html) | HOME (./index.html)

1. To start the arm, open a new terminal window and run the following commands (see Interbotix Setup Instructions (computer_setup.html#interbotix_setup) for details).

```
source ws/interbotix/install/setup.bash
ros2 launch interbotix_xsarm_control xsarm_control.launch.py robot_model:=px100
```

2. An `rviz` window showing the location of the arm should open.
3. To Stop running the arm, press `C-c` in the terminal window.
   - When you stop the robot the motors loose power and the arm may crash to the ground.
   - To prevent damage, always return the robot to its sleep position prior to stopping it or hold the robot and guide it down gently
4. Keep this terminal open until you are ready to stop using the robot. It must be open and running for you to run the robot
5. I have provided (very minimal!) sample code for using the interbotix python sdk.
   - See the comments in interbotix_xs_modules/arm.py (https://github.com/Interbotix/interbotix_ros_toolboxes/blob/humble/interbotix_xs_toolbox/interbotix_xs_m and the official documentation (https://docs.trossenrobotics.com/interbotix_xsarms_docs/python_ros_interface.html)
   - This code only works if the previous `ros2 launch` command is still running in the background
   - Especially when moving the robot for the first time, make sure it is clear of any potential obstacles
   - The code below lets you toggle between the sleep and home position
   - You must `source ws/interbotix/install/setup.bash` from any terminal where you want to run Interbotix code

```
from interbotix_xs_modules.xs_robot.arm import InterbotixManipulatorXS
from interbotix_common_modules.common_robot.robot import robot_shutdown, robot_startup
# The robot object is what you use to control the robot
robot = InterbotixManipulatorXS("px100", "arm", "gripper")

robot_startup()
mode = 'h'
# Let the user select the position
while mode != 'q':
    mode=input("[h]ome, [s]leep, [q]uit ")
    if mode == "h":
        robot.arm.go_to_home_pose()
    elif mode == "s":
        robot.arm.go_to_sleep_pose()

robot_shutdown()
```

   - To see what other commands you can run, look at the documentation linked above.
   - Keep in mind, the robot has only 4 degrees of freedom (DOF). This means that not all end-effector poses are possible.
6. Dive-weight positioning
   - The dive weight helps stabilize the robot
   - However, it is a bit large and can interfere with the sleep position
   - Try to position the weight so that the arm can reach the sleep position and go back to the home position
   - Run the example code to move the arm between sleep and home several times.
     - If any of the motors flash red when going to the sleep position, this means your weight is interfering. you will need to power cycle the robot and try again
   - As the robot moves, you can watch it move in the RViz window as well

Next Moves

Here are some recommended steps to get the robot capturing the pen.

1. Start from my example code and add some testing modes to perform the following actions:
   1. Open and close the grippers.
   2. Move forward or backward by small distance (relative or absolute?)
   3. Move up and down by a small distance (relative or absolute?)
   4. Rotate arm about the base by a small distance (relative or absolute?)
2. Use these modes to get a feel for the workspace limitations of the robot.
   - There are many limitations to the reachable areas of the robot's workspace.

- Singularities: where the robot geometry causes it to lose a degree of freedom, preventing it from moving in certain directions
- Joint Limits: mechanical stops and limitations that prevent a joint from rotating more than a certain amount
- Torque Limits: at some locations the motors may not have enough torque to move the weight of the robot.
  - At these locations, it is possible to command a move that causes a motor to over-torque and shutdown
  - The robot will then fall.
  - Industrial robots typically have more than enough power to lift themselves so this would not be an issue with them.
- If the planner fails to compute the trajectory, it means that the algorithm could not find a feasible path.
  - A path might be infeasible due to geometry but it also might be infeasible due to time
  - Most movement functions have various parameters related to the timing of the action (for example, if commanding the robot to a position how long should it take for the robot to arrive).
  - Timing is important: large displacements require more time or the acceleration and velocity limits are exceeded. Small displacements can only be done so slowly due to friction and other losses in the system.

3. After each movement, the program should print out the end-effector pose and joint values.

Useful Functions

1. `robot.arm.set_ee_cartesian_trajectory`
   - Move the end-effector along a straight-line path
2. `robot.arm.get_joint_commands`
   - Get the commanded joint positions of the robot
   - The `robot.core.robot_get_joint_states` function gets the actual state of the joints.
     - This is better information for our purposes, but it is also lower-level and not needed (`get_joint_commands` is sufficient)
     - It is listed here for reference.
3. `robot.arm.set_joint_positions`
   - Set joint positions of the robot
4. `set_single_joint_position`
   - Set the position of a single joint
   - The joint names (from base to end-effector) are `waist` , `shoulder` , `elbow` , `wrist_angle`
5. `robot.arm.get_ee_pose()` Get's the pose of the end effector
   - This is a transformation matrix of the end effector relative to the base of the robot
   - The last column contains the x, y, and z positions in rows 0-2 respectively

6. To compute the end-effector given the joint states, you can use the modern_robotics (https://github.com/NxRLab/ModernRobotics) library directly:

```
import modern_robotics as mr
joints = [waist, shoulder, elbow, waist_angle] # pseudocode, use your own values
T = mr.FKinSpace(robot.arm.robot_des.M, robot.arm.robot_des.Slist, joints)
[R, p] = mr.TransToRp(T) # get the rotation matrix and the displacement
```

7. Some of the above commands have an option to make them *blocking* or *non-blocking*
   - When blocking, the function will wait until the action is complete before returning (this is the default)
   - When *non-blocking*, the function returns while the robot is still moving. Movement can then be updated by issuing another command

Workarounds

- The following are workarounds either to undocumented or incorrect behavior in the `interbotix` library
- Narrowing the problem down to a minimal example and reporting it on github can likely lead to a successful pull-request.
- `robot.arm.set_trajectory_time(2, 0.3)` resets the trajectory timing to the defaults
  - The current behavior of the library sometimes causes moves like `robot.arm.go_to_home_pose()` to use the settings from the last call to `set_ee_cartesian_trajectory` , which can lead to some unexpected motions

- `robot_startup()` starts a ROS 2 node in a separate thread to process updates to the joint positions. As far as I can tell, the need to use this function is not documented anywhere, but without it the robot's position is only updated when calling some (but not all) movement commands.

Extrinsic Calibration

To control the robot using data from the Realsense, it is necessary to know where the Realsense is relative to the robot. To be done properly this type of *hand-eye calibration* must use measurements from the camera, because it is extremely difficult to measure the physical location of the camera's center (which is inside the camera itself and cannot be measured externally).

The goal of the calibration is to find the transformation (i.e., the translation and rotation) between the camera's coordinate frame $\{c\}$ and robot's coordinate frame $\{r\}$.

There are two approaches:

1. Hand Tuning: take a measurement, and adjust until the system works.
   - This method is hack, (but this is a *hackathon*).
   - Measurements are only approximations so tuning is required.
   - There are technically 6 Degrees of freedom that describe the transformation between camera and robot (i.e., six inter-related parameters to tune)
   - Making some simplifying assumptions can reduce the number of parameters (e.g., assume the camera is flat relative to the ground)
   - Inaccurate, not repeatable, but can get the job done in a pinch (if the goal is a one-time demo).
2. Calibration procedure: collect a bunch of data points where the end-effector is at a known location relative to the robot base and the camera.
   - Requires more math and coding than hand-tuning
   - Once implemented and working, calibration can be automated
   - Little or no hand-tuning required
   - Most accurate method (properly calibrated cameras can be highly precise)
   - Usually requires some infrastructure (such as a calibration fixture). We will use the pen!

Calibration Setup

Regardless of the method you choose, understanding the assumptions you are making and having a good picture of the coordinate frames in question is essential for debugging.

Drawing and Assumptions

1. Make sure you understand the coordinate systems used by the camera and the robot.
   - You can discover this through experimentation or by reading the documentation (https://dev.intelrealsense.com/docs/projection-in-intel-realsense-sdk-20)
   - You should develop a theoretical understanding from the documentation that you then confirm with experiments
2. Make a drawing of the coordinate systems involved. It should include
   1. A top-down view, showing the camera and robot axes.
   2. A side view showing the camera and robot's vertical axis and the d_z offset between them.
3. Once you have a calibration use `matplotlib` to plot the top-down view of the coordinate frames and check if the result is reasonable.
   - This quick verification does not require the robot or camera, just the calibration and data
   - Doing a quick drawing in `matplotlib` can greatly speed up debugging
   - Check that the generated drawing matches with your sketch and that the distances are reasonable

Calibration Math

Here I provide practical guide for the calibration math. This section is based on the notes here: [1])

1. Assume we have a set of n points:
   - $P_i \in \mathbb{R}^3, i = 1$ to n are the points in the camera frame
   - $Q_i \in \mathbb{R}^3$ i = 1 \text{ to } n$ are the same points in the robot frame
2. Find the centroid of the points:
   - $\bar{P} = \frac{1}{n} \sum_i^n P_i$
   - $\bar{Q} = \frac{1}{n} \sum_i^n Q_i$

3. Subtract the centroid from each point:
   - $\bar{P}_i = P_i - \bar{P}$
   - $\bar{Q}_i = Q_i - \bar{Q}$
   - The centroid of both point sets are at the same location in space.
4. If the points in one set undergo a pure rotation about their centroid, they should line up with the points in the other set exactly (except for noise).
   - The immediate is to find a rotation matrix $R \in \mathbb{R}^{3\times3}$ such that $\bar{Q}_i = R\bar{P}_i$ for $i = 1$ to n.
   - An algorithm to find R is the Kabsch Algorithm (https://en.wikipedia.org/wiki/Kabsch_algorithm), which is implemented in scipy as scipy.spatial.transform.Rotation.align_vectors (https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.align_vectors.html)
   - The algorithm takes two sets of points and finds the rotation between them (assuming they only differ by a rotation and some noise).
5. After finding R the goal is to find the translation t between the point-sets:
   - Converting from a point P in the camera frame to a point Q in the robot frame is done with $Q = RP + t$.
   - Once R is known then t can be found using the conversion formula and the centroids (or any two corresponding points)

Calibration Procedure

Write a script that performs the calibration procedure:

1. The script should be run while the robot is holding a pen
2. The robot should move to a few pre-set positions to gather the necessary data
3. The script then computes and outputs the calibration parameters
4. The robot moves to a few test locations and the camera coordinates are converted to the robot coordinates and compared

Design Notes

1. There is an offset between where the pen is detected and the center of the robot's end-effector frame: this offset can be manually accounted for.
2. It makes sense to have a class that can handle the calibration math in a way that is agnostic to how the data was actually collected.
   - It's responsibilities could include:
   - Saving calibration parameters to a file
   - Loading calibration parameters
   - Performing a calibration, given data points
   - Performing the conversion between coordinate systems
3. It is up to you how the waypoints for the calibration procedure are specified and whether they are in end-effector or joint space.
4. It makes sense to save the data you collected. This way you can test your analysis code without needing to re-collect data each time.
5. You can also use your existing code to help specify the waypoints.
6. Calibration can be a bit finicky: you may get different results if you run it multiple times due to the inaccuracy of the detector and how you place and hold the pen.

Robot Control

A basic robot control loop consists of the following steps:

1. Move Arm to its starting position
2. Open the grippers
3. Measure the Pen Location
4. Turn at the *waist* until the end-effector is facing the pen
5. (Optional) Adjust the height of the arm so the grippers are level with the pen
   - To start out, just hold the pen at the correct height!
6. Move forward until the pen is inside the grippers
7. Close the grippers

There are two possible modes of control: you can use either or a combination of both (e.g., closed loop for the angle, open loop for moving forward). Below are descriptions of both the closed loop and open loop approach to pointing the robot at the pen.

Closed Loop

When rotating (this assumes that both the arm and pen's locations are expressed as cylindrical coordinates relative to the robot base):

1. Measure the pen location
2. Measure the robot's location
3. Move based on the locations (e.g., proportionally to the error between the pen and robot angle)

Open Loop

1. Given the pen's location, compute the `waist` angle and move the arm to that angle
2. Then move forward based on the computed distance.

Design Notes

1. There may be a few modes of operation as you control the robot (e.g., moving to the pen, closing grippers, etc). It makes sense to track these modes by assigning a state to each of them. You can use a python enum (https://docs.python.org/3/library/enum.html) to label and track states
2. If the pen's location is presented as cylindrical coordinates in the robot frame, the computations can be significantly simplified:
   - It may be helpful to refer to your drawing of the coordinate system
3. It may be easier to command locations in joint coordinates rather than needing to go through the IK process
   - You can use other tools you have develo

Multi-Processing

- You may have noticed (depending on how you implemented your control loop) that when putting the control together, the image data from the camera slows down.
- There are a few contributing factors to this slowness
  1. Blocking functions: some `interbotix` functions do not have the option of being non-blocking: thus, while using these functions to move the robot, the OpenCV window is not updated and images are not read while the Realsense is moving.
     - It is technically possible to complete this task without using any functions that require blocking, but that would amount to essentially re-writing some of the `interbotix` library
  2. When `robot_startup` is called, the `interbotix` library runs a thread in the background to read the joint information from the robot
     - Due to the concurrency model of python, only one thread can execute at a time, thus while the joints are updated, your code does not run
     - It is possible to run code without `robot_startup` and manually trigger updates to the robot state, however, accomplishing this would require a deep dive into the `interbotix` library implementation.
- Overall, I believe it is possible to get smooth movement and image processing in a single thread, but not without essentially rewriting parts of the interbotix library and interacting with the robot on a lower-level than what is provided by the Python API
  - While not ideal, the target of this API is for beginner users: when using the robot via ROS 2 you would not have this problem.
- There is, however, a workaround:
  1. Use the python `multiprocessing` library to run the image updating and movement code in separate processes at the same time.
  2. While there are many ways of implementing this, one of the less error prone is to use a `multiprocessing.Queue` to pass information between the processes.

Relay Mode

If you've gotten this far, the challenge is complete. But there is always more to do!

1. The goal is to have your robot be able to hand off the pen to another robot.
2. Approach this problem with at least one partner.
3. Each robot does not necessarily need to be running the same code.
4. The robots do not necessarily need to communicate with each other.

5. How long of a hand-off chain can you create?

UP (../index.html) | HOME (./index.html)

Hints

1. I suggest tackling the project in stages
   - Be able to threshold the pen properly
   - Be able to measure the pen's location in cylindrical coordinates
   - Be able to move the robot to a given location in cylindrical coordinates
   - Put everything together with the control loop
2. Test as you go.
   - Use the python interpreter to try statements out and iterate quickly on designs
   - Draw images of each image processing step
   - Print out data when you need some more information
   - You can also use the debugger that is built into VSCode
3. It will help to draw diagrams of the workspace geometry
   - I gave you two pens: one to use for testing and one to write with!
4. You should expect to tune parameters.
   - I recommend saving good values when you find them
   - Parameters may change if the lighting conditions change
   - Get some good rough parameters, move on to the next step, and then do some find tuning at the very end.

Other References

- OpenCV Python Tutorials (https://docs.opencv.org/4.6.0/d6/d00/tutorial_py_root.html)
- Numpy Manual (https://numpy.org/doc/1.19/)
- Modern Robotics (https://github.com/NxRLab/ModernRobotics)

Bibliography

[1]N. Ho, “Finding optimal rotation and translation between corresponding 3d points.” Available: https://nghiaho.com/?page_id=671 (accessed Sep. 01, 2024) (https://nghiaho.com/?page_id=671 (accessed Sep. 01, 2024))

Author: Matthew Elwin.