7MR10070: Software and Robotic Integration
Semester 2

Final Project: ROS-Slicer integration and robotic control

Introduction

Before describing why integrating image processing and robotic simulation is useful in medical applications, we need to describe if they are useful on their own.

Image Processing

Image processing (such as segmentation, registration, image guided systems) has proven to be a vital tool in the medical field. Some of its uses include:

- Improving/Restoring the quality of images in order to help physicians pick out important details/diseases in them
- Through machine learning and neural network techniques, extracting features from the image and learning how to detect said details/diseases automatically or separating different tissues from each other again in order to assist physicians
- In combination with machine learning techniques, assisting untrained physicians in figuring out how pick out certain details in unseen scenarios by learning how to generate new examples[1]
- Assisting surgeons in surgeries by providing visual guidance for procedures

Robot Simulation

There is a huge need of robotic assistance in the medical field. Whether it is in surgery or in assistive robotics, robots will start playing a bigger role in medicine. There is a multitude of reasons for this and some of them are:

- Robots provide high precision movement and in cases where automation is not desired, they can augment the abilities of physicians (i.e. using virtual constraints)
- Assuming a model/procedure has been figured out, creating a new robot takes far less time than training a new physician/surgeon
- Fills in certain roles that people lack interest in doing so themselves (such as nursing positions)
- Reduces the bus factor[1] the field currently suffers from by heavily relying on individual people

Given the sensitive subject and that the intended use of robots can have long lasting or even fatal consequences if something does not go as planned, simulations help figure out certain problems before using a robot on an actual person. More specifically, robotic simulations can be used to:

[1] The risk resulting from information and capabilities not being shared among/by enough team members either because of lack of planning or long training times

- Determine if the specifications provided satisfy the task at hand
- Identify any design flaws early on
- Provide the ability to test multiple configurations to decide which configuration is better for the task
- By isolating software and hardware components, help figure out which part of the robot might cause/is causing an issue

Integrating the two

Integrating image processing and robotic simulation allows us to optimise our system, improving performance both in terms of accuracy and execution time. For example, using image processing techniques we can plan a path or choose the best starting point in a surgery simulation and then have the robot perform the task based on that optimisation. Furthermore, using proper image processing techniques, we can recognise important/vulnerable areas that the robot can be programmed to take better care around

An example of integration of the two can be found in microsurgery, where using image processing techniques, we can 1) create virtual constraints that restrict wrong movement and involuntary movement (perhaps caused by tremor) and 2) as mentioned above, pre-plan the operation and thus reduce the time required for the patient to be on the operating table.

Developing an end-to-end pipeline

Developing an end-to-end pipeline is probably one of the most important parts of robotic and imaging integration. By developing an end-to-end pipeline, we can figure out certain problems that will arise to individual parts of the pipeline right from the beginning. For example, if we know that a specific part of our image processing algorithm consistently underperforms (and can't fix it), we can configure our robot to take that into account. This would not have been possible if the two parts of the system were designed in isolation.

Moreover, designing the architecture and how different modules and components interact with each other, we allow for better scaling of the system when new features need to be added. Furthermore, proper design and implementation can help make certain components re-usable, speeding up the process of development as well as allow other developers/organisations to take advantage of said components in order to solve greater problems faster.

After designing the pipeline, we can then use automated (and manual) validation techniques in order to assess the accuracy and robustness of our whole system. Using unit testing we can see how each individual part performs and once we are sure every part works well on its own, we can start using integration testing. Here all (or multiple) components are tested together, thus ensuring that our framework can work well from one end to the other. Of course, we must still be cautious as if we have not designed our tests or model correctly, we can miss important issues in the system and under the false assumption of passing tests go into manufacturing which might be a waste of time and money. In even worse cases, we can cause serious harm to a person in a real-life scenario.

Methodology

3D slicer path planning

The path planning aspect of the pipeline is separated into 6 different tasks, performed in the following order, starting from the least expensive (in terms of time taken) algorithm, working its way to the most expensive:

1. First, create the OBB trees required for each task
2. Then, filter for targets that are within the hippocampus
3. Then we further filter for trajectories of the specified distance
4. After that, filter for entry/target trajectories that do not pass through the blood vessels and blood vessels dilate
5. We then filter so that only trajectories of a certain angle (degrees) are accepted
6. Finally, we choose the best trajectory based on its distance from the blood vessels

By using this order, we rule out most of the trajectories before we reach the more expensive checks in our overall algorithm.

Note: Where pseudocode is provided, it is written in a generic form and does not contain any language/library specific syntax. Moreover, the provided .vtk files were converted to labelMaps using the ***Model to Label Map*** module, using refence volumes of size 256x256x220 (The label maps from the previous coursework were used)

Placement of the tool into a target structure

We transform the input node to an IKJ matrix and then loop over each target image. To decide if a target point is within our target area, we iterate over all the target points and retrieve the pixel value for each one. If its value is greater than zero (or 255), it is a valid target. The function used to do this is called ***getFilteredTargets(targets, area)*** and should accept any area we want to filter for.

Time complexity: $O(N_{tp} * R_{pv})$

$where\ N_{tp}\ is\ the\ total\ number\ of\ target\ points, R_{pv}\ is\ the\ lookup\ time\ for\ the\ pixel\ value$

```
Pseudocode:
For each point in target points:
     Get the coordinates (x, y, z) of the point
     Retrieve the pixel value of (x, y, z) in the image
     If pixel value != 0 ? add the point to the list : nothing
```

Trajectory is below a certain length

We iterate over each available entry-target pair and calculate the distance between the two. If it is below the threshold, we add the pair to the list of valid pairs. The function responsible for this is called ***getEntryTargetPairsWithinLengthThreshold(entriesAndTargets, lengthThreshold)***.

```
Pseudocode:
For each entry point in entry points:
     For each target point in target points:
          Get the distance between the two points
          Get the points of the line
          If dinstance <= distanceThreshold:
               add (entry, target) point to valid points list
```

Avoidance of a critical structure

We first create an oriented bounding box tree (OBBTree) of the blood vessels and blood vessels dilate. We then iterate over each entry and target pair and check if the pair intersects any of the bounding boxes defined by the OBBTree. If there is an intersection, we reject the path. The function used to do this is called ***getTrajectoriesAvoidingArea(entriesAndTargets, area)*** and should accept any area we want to avoid. It is important to note that this function uses ***isPassThroughArea(tree, entry, target)*** which is where the actual intersection check is made. I chose to separate the check so that it can be used in combination with the other constraints, without having to loop through each entry target pair for each one every time.

Time complexity: $O\left(2 * N_{ep} * N_{tp} * log(N_{vp}) + N_{vp} log(N_{vp})\right)$

$where\ N_{ep}\ is\ the\ number\ of\ entry\ points, NP_{tp}\ is\ the\ number\ of\ target\ points\ and\ N_{vp}\ is\ the\ number\ of\ blood\ vessel\ points$

```
Pseudocode:
For each entry point in entry points:
     For each target point in target points:
          Get the line between the two points
          Get the points of the line
          validLine = true
          For each point on the line:
               If point passes through structure:
                    validLine = false
                    break
          if validLine:
               add (entry, target) point to valid points list
```

Filter for incisions below a certain angle

We first create an oriented bounding box tree (OBBTree) of the cortex. We then iterate over each entry and target pair and check if the pair intersects any of the bounding boxes defined by the OBBTree. If there is an intersection, we create a line perpendicular to the intersection. We then create two vectors, one for our entry/target pair and one for the intersecting points and calculate the angle between the two. If the angle is below the specified limit (55), we accept the path. The function used to do this is called

getTrajectoriesWithSpecifiedAngle (entriesAndTargets, area, specifiedAngle) and should accept any area we want check the angles for. It is important to note that this function uses ***isValidAngle()*** which is where the actual check for the angle is made. I chose to separate the check so that it can be used in combination with the other constraints, without having to loop through each entry target pair for each one every time.

Time complexity: $O\left(2 * N_{ep} * N_{tp} * log(N_{cp}) * T_a + N_{cp} log(N_{cp})\right)$

where N_{ep} *is the number of entry points,* NP_{tp} *is the number of target points and* N_{cp} *is the number of points cortex points and* T_a *is the time needed to calculate the angle between two vectors*

```
Pseudocode:
For each entry point in entry points:
     For each target point in target points:
          Get the line, lineET, between the two points
          For each point on line:
               If point connects with the cortex:
                    Get perpendicular line where lineET passes
through the cortex
                    Calculate the angle between the two lines
                    If angle < 55:
                         Add (entry, target) point to valid
points list
```

Maximizing distance from critical structures

We first create an oriented bounding box tree (OBBTree) of the critical structure using the CellLocator of VTK. This was chosen as it provides a utility method that helps as find the closest point from an input point to a point in the critical structure. We then iterate over each entry-target pair, creating points within their trajectory. For each point those points, we calculate the distance from the structure and pick the minimum one. We then add them up and store them in a dictionary. Then in order to get the best one, we simply sort the dictionary and get the one with the most distance from the critical structure. The function used for this is ***getBestAndWorstTrajectory(entriesAndTargets, area1, area2, precision)***

```
Pseudocode:
For each entry point in entry points:
     For each target point in target points:
          Get the points between entry-target pair
          For each point on the line:
               Calculate distance from mesh
               If distance <= minDistance:
                    minDistance = distance
          add distance, pair to valid points list
sort valid points in descending order
the first entry should now be the best trajectory
```

OpenIGTLink

In order to send messages between the two we use the OpenIGTLink protocol. OpenIGTLink defines the message format that is used to transfer data between Slicer and ROS. The message consists of a header, an extended header, the content and some meta data. Furthermore, we need to define an importer in order to interpret the above message format. In order to achieve two-way communication, we define an exporter that sends the current location of the end effector back to Slicer. Finally, we define a Calibrator in order to correctly transform the points between Slicer and ROS.

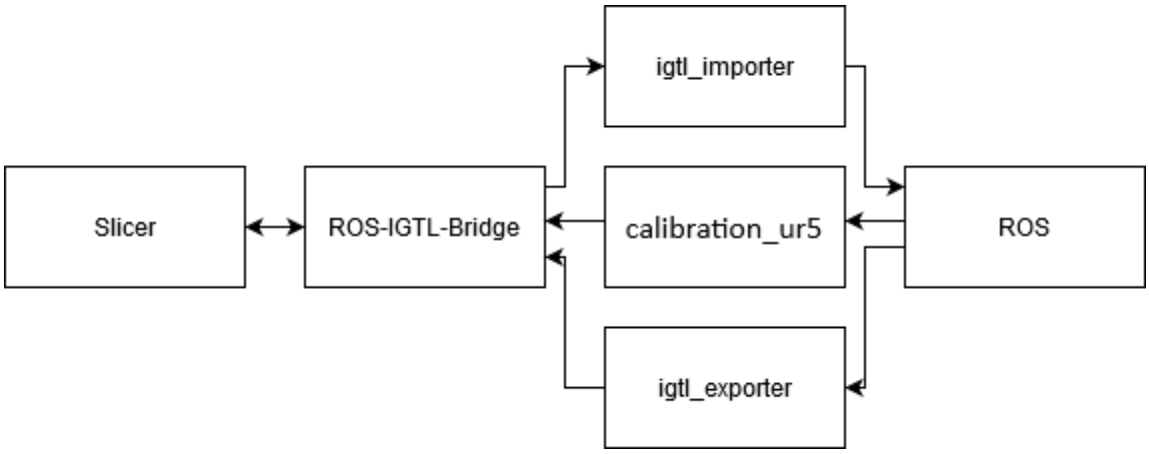

Figure 1: Connection between Slicer and ROS

Data transfer from 3D Slicer to ROS: igtl_importer.py

The data transfer from 3D Slicer to ROS works using the ***igtl_importer.py*** file. It takes an entry point, 'Entry', and a target point, 'Target', loads them as ***geometry_msgs.msg Pose*** objects and operates as follows:

- Once both points are received, they are converted from millimetres to meters and the kinematics are solved using the ***move_it*** framework
- The robot is then instructed to move the "Entry" point
- Once the end-effector is at the entry point, the importer prompts the user to press "enter" to move to the "Target" point. The robot is moved to the target point by creating a straight line/trajectory between the two. This is done by breaking said line to smaller waypoints and forming a cartesian path using ***moveit_commander***'s ***compute_cartesian_path(waypoints)***. The reason it is implemented this way (the straight line) is because we are moving the robot's end effector (i.e. a needle) in a straight line through the brain
- In order to move, the importer also restricts the orientation of the robot as we don't want to be moving frantically within the patient's brain. This is done by creating a vector, **ET**, between the entry and target. Then using a reference vector **O**, it calculates the cross-product, **X**, of **ET** and **O** and then further calculates another cross-product, **Y**, of **ET** and **X**. Using **X, Y** and ET we create an orthogonal matrix which we pass to the mat2quat function in order to calculate the quaternion.

Furthermore, the importer places the Cortex model and the entry-target points in RVIZ for visualisation. The cortex is converted to an .stl file by using Slicer's Model Maker module after converting it to a label map. Its orientation is hard coded within the importer and perhaps it could have been saved within the model itself using a third an appropriate application.

Data transfer from ROS to 3D Slicer: igtl_exporter.py

This is a rather simpler implementation. After initialising the node, we simply use the ***moveit_commander*** package to create a ***move_group***. Using the ***move_group*** we retrieve the robot's position and after converting it to millimetres (as required by Slicer), we send the end-effector's coordinates through the ***ros_igtl_bridge*** package as an ***igtlpoint***. The main benefit of the exporter is to assist us in validation/sanity checks.

ROS (to move the robot)

Initially, a simple robot was designed using the XACRO format as seen in *Figure 13: Basic robot* and *Figure 14: Simple robot xacro*. The model described a simple two-revolute joint robot, where one joint would rotate around the x-axis and the other around the z-axis. Given the complexity of our task and the restricted degrees of freedom of the robot, the ur5 manipulator was chosen instead *(Figure 6: Robot at default position (RVIZ))*. This is an open-source robot that was used as part of a workshop at the International Symposium on Medical Robotics by the Georgia Institute of Technology [4]. Furthermore, it has been modified to include a needle as the end effector, in order to completely satisfy our task.

It is available with the ***ismr19_moveit*** package under the ***demo.launch*** file. The package provides:

- A RobotModel that is built for ROS and visualised in RVIZ
- A kinematics model for the robot as well as various kinematic solvers to automate the calculation of kinematics based on the used RobotModel

The core classes of this are RobotModel and RobotState.

The RobotModel contains relations between all links and joints including their limits (collision, safety limits, etc), as defined by the URDF and SDRF files.

The RobotState contains information about the robot at any point in time. It is used to obtain kinematic information about the robot depending on its current state.

Calibration (to translate points between ROS and Slicer)

The calibration (and therefore the transformation) occurs after we have calculated the entry and target points using the Path Planner module. This is done using the following steps:

1. Create 8 markups/points around the provided models (i.e. the critical structures and the cortex, forming a bounding box) in Slicer.
2. Create 8 points that form a bounding box in ROS, using the same measurements as the bounding box we created in Slicer (note: These are currently hard coded in calibration_ur5.py in the ***robot_control*** package and might have to be changed for completely different configurations).
3. Then we run the ***calibration_ur5.py*** script, which prompts the user to save each calibration point in Slicer, one by one.
4. We then go to Slicer, where we can build a linear transformation matrix using the Fiducial Registration Wizard which is found in the IGT extension/module. The Fiducial Registration Wizard maps the 8 fiducials created in Slicer to the 8 points sent by ROS through the calibrator.
5. Then, we invert the transformation matrix and we transform the critical structures, the cortex, the entry points and target points. In order to send the transformed items, we need to use the 'harden' functionality provided by Slicer.

Validation

3D slicer path planning

For each part of the Path Planner module, following overall process was used:

1. Pick a small subset of the data
2. Visually look for a trajectory / point that is obviously valid for a task and for one that is obviously invalid for a task. For example, when filtering for targets within the hippocampus, the invalid point could be one outside the image.
3. Run the algorithm
4. Inspect the output in slicer
5. Increase the subset of data gradually

Furthermore, we use unit testing to provide automated tests as well. These can be found in PathPlanner.py and are the following:

testLoadAllData(path): check that data has been loaded successfully
testGetFilteredHippocampusValidTargets(): check that targets are correctly filtered down to hippocampus targets

testGetFilteredHippocampusInvalidTargets(): check that targets are correctly rejects invalid targets

testAvoidBloodVesselsDilateValidPath(): check that the algorithm accepts a path that doesn't pass through the blood vessels dilate
testAvoidBloodVesselsDilateInvalidPath(): check that the algorithm rejects a path that passes through the blood vessels dilate

testAvoidBloodVesselsValidPath(): check that the algorithm accepts a path that doesn't pass through the blood vessels
testAvoidBloodVesselsInvalidPath(): check that the algorithm rejects a path that passes through the blood vessels

testAngleValidPath(): check that the algorithm accepts a path that hits the cortex at the correct angle

testAngleInvalidPath(): check that the algorithm rejects a path that hits the cortex at an incorrect angle
testCountRejectedTrajectories(True): To count rejected trajectories and time each part. A powerful feature of this test is that it can help us figure out the correct order for each part of the path planning algorithm. This is a slow test
testAllTogether(): Just to see if everything can run together (pseudo test). This is a slow test

ROS (to move the robot and confirm connection)

Setting up the connection

Slicer Part

1. Launch Slicer 4.8.1
2. Go to your favourite IGTLink extension (SlicerIGT in this case)
3. Create a connection with Slicer as the server on port 18944
4. Check that the current status is set to "WAIT"

ROS Part

1. Start the VM (if using one. I run this on Ubuntu, so it was not required) with the appropriate configuration for two-way communication
2. Launch the bridge file of the ***ros_igtl_bridge***
3. Choose to run as client
4. Set IP to the appropriate IP found by writing ***ifconfig*** in your terminal
5. Set port to 18944
6. If you check Slicer, the status should have changed to "***ON***"

A connection should be now established.

Sending data from Slicer to ROS

First initialize the connection as mentioned in the methods section. Then:

Step 1:

1. In Slicer 4.8.1
2. Go to Markups
3. Create two new MarkupFiducials called “Entry” and “Target”
4. Place a point in the workspace (I did this the other way around. I manually moved the robot in RVIZ and those coordinates instead)

Step 2:

1. Launch a new terminal
2. Go to your workspace and source it
3. Enter ***roslaunch ismr19_moveit demo.launch***

Step 3:

4. Launch a new terminal
5. Go to your workspace and source it
6. Enter ***rosrun robot_control igtl_importer.py***

Step 4:

1. Launch a new terminal
2. Go to your workspace and source it
3. Enter ***rosrun robot_control igtl_exporter.py***

Step 5:

1. Go back to slicer
2. Go to IGT -> IGTLinkIF

3. Scroll down to I/O Configuration
4. Click send for both “Entry” and “Target”
5. The robot should now move to the entry point and you should be prompted to click "enter" on the terminal to proceed to the target point.

Note: You need to send both points for the robot to begin the operation

Input Validation

Valid Input (moves to a point)

Figure 9: Robot's end effector at a random point (RVIZ)

This figure shows our robot moved to a random position from the default one. This is to validate that the kinematics work as intended

Invalid Input (can’t get to the point)

Figure 10: Invalid position sent to ROS

This figure shows the error message the robot publishes if an invalid configuration is requested. If the kinematics can’t be resolved, the robot simply passes message stating so.

Slicer Scene

The images concerning the Slicer scene contain many elements.

- Green dots with blue background: The blue represents the points set on Slicer for the bounding box and the green ones within them are the ones sent by ROS. This clearly shows that the calibration was successful.
- Red dots: All the possible entry points
- Cyan dots: All the possible target point
- Yellow background surrounding a dot: The current position of the end-effector (as sent by ROS)
- Green dot: Best entry point
- Blue dot: Best target point

Position Validation

Default Position

Figure 11: Slicer with models loaded

Figure 12: Best points chosen by PathPlanner

These two figures show the Slicer scene with all the models and the best entry-target pair loaded, with and without the yellow background marker for the end-effector

At Entry position

Figure 2: End effector at entry point (view 1)

Figure 3: End effector at entry point (view 2)

Here we have both a top and bottom view of the brain. As mentioned in the instructions above, we can see the yellow background around the entry point (green dot) which represents the end-effectors position. The yellow background is set only when we hit the "Send" button through the IGT extension.

At Target Position

Figure 4: End effector at target point (view 1)

Figure 5: End effector at target point (view 2)

Here we have both a top and bottom view of the brain. As mentioned in the instructions above, we can see the yellow background around the target point (blue dot) which represents the end-effectors position. The yellow background is set only when we hit the "Send" button through the IGT extension.

Ros Scene

At default position

Figure 6: Robot at default position (RVIZ)

Here we can see our robot in its default position. The red part represents the needle that will be used to enter the brain

At Entry position

Figure 7: Robot's end effector at entry point (RVIZ)

Here the cyan dot represents the entry point where the end-effector is currently position. The robot moves here once we send the entry point to the robot.

At Target Position

Figure 8: Robot's end effector at target point (RVIZ)

Finally, we once we send the target point (the blue point within the brain), we can see that the robot correctly moves in a straight like and enters the brain at the desired point as requested.

Whole system

The whole pipeline is a multistep process. Most parts require some manual observations, and this serves to provide extra safety/protection, assuming this would be translated to a real-life scenario. The steps required to run it are as follows:

1. We launch Slicer and load all our volumes (critical structures, cortex and entry-target points)
2. We convert the critical structures from ".vtk" format to labelMaps/markupFiducials using Slicer's ***Model to Label Map*** module using as reference volumes the labelMaps given from the previous coursework.
3. We run our PathPlanner algorithm in order to calculate and save the best entry-target pair. Our scene should now resemble figures 2-4 (without the yellow background on the entry-target points).
4. We then initialise the connection and launch our robot in RVIZ as described in the *ROS (to move the robot and confirm connection)* section.
5. We then start the calibration process as mentioned in the *Calibration (to translate points between ROS and Slicer)* section.
6. Then using the resulting transformation matrix, we transform the critical structures, cortex and entry-target points of Slicer to match the workspace of ROS.
7. We use the 'harden' functionality on Slicer to have a correct representation of the structures/points and send the correct entry-target points to ROS.
8. We load the cortex (converted to .stl format using Slicer) as a marker node to RVIZ through RVIZ's interface, using a scale factor of 0.001 to match Slicer's and ROS' measurements.
9. The cortex's origin point in RVIZ is set according to the bounding box created during the calibration step. Once we ensure that the robot is placed correctly around the structure, we can start sending our entry and target point.
10. We go again to IGT in Slicer and click "Send" on the "Entry" and "Target" points. We should now be able to see the robot's end effector moving to the entry point in RVIZ (Figure 7: Robot's end effector at entry point (RVIZ)). The "Entry" point is loaded as a cyan marker. The robot will not move unless both points are sent.
11. At the same time, we can check the Slicer scene to confirm that the end effector is at the correct entry point. This is shown by the yellow background around the optimal entry position (Figure 2: End effector at entry point (view 1)).
12. Now there should be a prompt in the terminal to press "enter" to proceed to the target point. After doing this, we should be able to see the robot's end effector moving to the target point from the entry point in a straight line (Figure 8: Robot's end effector at target point (RVIZ)). The "Target" point is loaded as a blue marker.
13. At the same time, we can check the Slicer scene to confirm that the end effector is at the correct target point. This is shown by the yellow background around the optimal target position (Figure 4: End effector at target point (view 1)).

These should be all the steps required to perform the whole pipeline. Scenes of these steps and saved models can be found within the repository under the "models and scenes" folders.

Conclusions

What was achieved

Through this project, we managed to correctly calculate the best trajectory that avoids critical structures based on the length, angle and distance from critical structures. This is achieved by employing various image processing techniques that optimise the traversal between points within large data structures using packages such as VTK, numpy and Slicer. Furthermore, we describe and set up a connection between Slicer 3D and ROS in order to perform the operation. Using the ***move_it*** and ***igtl_bridge*** packages, we successfully command a robot and transfer points and information between Slicer and ROS. We've also learned how to create a custom RobotModel even though we use the one provided by ***ismr19_moveit*** for simplicity. Finally, we learned how to calibrate models between different workspaces (Slicer and ROS) and how to set up markers in RVIZ to demonstrate the entry and target points and load cortex as an object in the scene.

Future improvements

While the overall system performs the desired task, there is still a lot of room for improvement in various parts of the pipeline.

Path planner module

The path planner module seems to work well. Still, we could have further optimised our code in order to speed up the process. Another point worth mentioned is the way we calculate the best trajectory. Here we weigh all critical structures as equal. Perhaps here we could have used a weighting system to better calculate the distance from multiple structures instead. Furthermore, after calculating the optimal entry-target, we could have automatically sent it to ROS but perhaps this might be risky if this was a real-case scenario.

RobotModel

The chosen robot was appropriate as it includes a needle and is specialised for our task at hand. The robot is part of an open source project found on Github for a workshop meant to demonstrate a connection between Slicer and ROS [4]

Calibration

An obvious improvement here is an automated bounding box calculator as we are currently choosing the points manually.

Importer & Exporter

These are mostly fine. Here we could have split the code in more files/classes in order to make some parts reusable and separate some of the robot's repositioning logic and IGTL logic.

Overall

As mentioned above, while everything works and the pipeline is complete, we could have automated some of the manual parts (such as the calibration and sending points to ROS). Although, perhaps it is better this way as if it was a real case scenario, we would have to oversee each step individually in order to make sure everything is done safely.

Code / Git Repository

https://github.com/Meldanen/kcl/tree/master/robotics/final

References

[1] A. B. L. Larsen, S. K. Sønderby, H. Larochelle, and O. Winther, "Autoencoding beyond pixels using a learned similarity metric," *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 4, pp. 2341–2349, 2016.

[2] A. Mohammed, L. Wang, and R. X. Gao, "Integrated image processing and path planning for robotic sketching," *Procedia CIRP*, vol. 12, pp. 199–204, 2013, doi: 10.1016/j.procir.2013.09.035.

[3] A. Norouzi *et al.*, "Medical image segmentation methods, algorithms, and applications," *IETE Tech. Rev. (Institution Electron. Telecommun. Eng. India)*, vol. 31, no. 3, pp. 199–213, 2014, doi: 10.1080/02564602.2014.906861.

[4] ISMR19 workshop. Available at: https://github.com/rosmed/rosmed.github.io/wiki/ISMR2019

[5] Ros-Melodic. Available at: http://wiki.ros.org/melodic

[6] ROS-IGTL-Bridge. Available at: https://github.com/openigtlink/ROS-IGTL-Bridge

[7] Slicer 3D. Available at: https://www.slicer.org/

[8] VTK. Available at: https://vtk.org/doc/nightly/html/index.html

Appendix

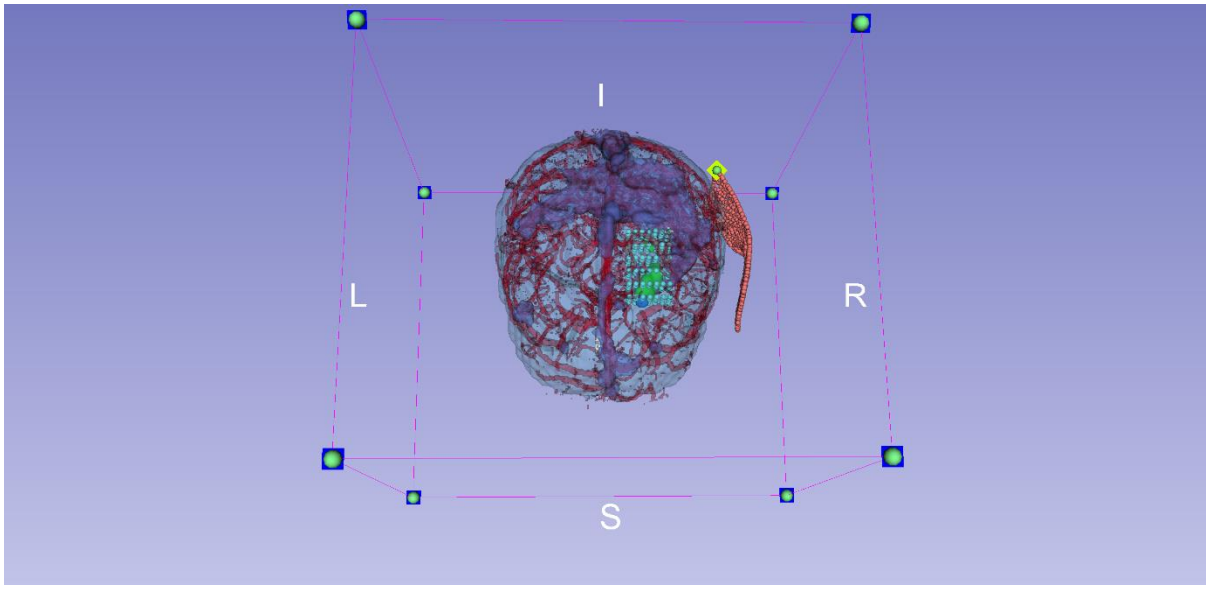

Figure 2: End effector at entry point (view 1)

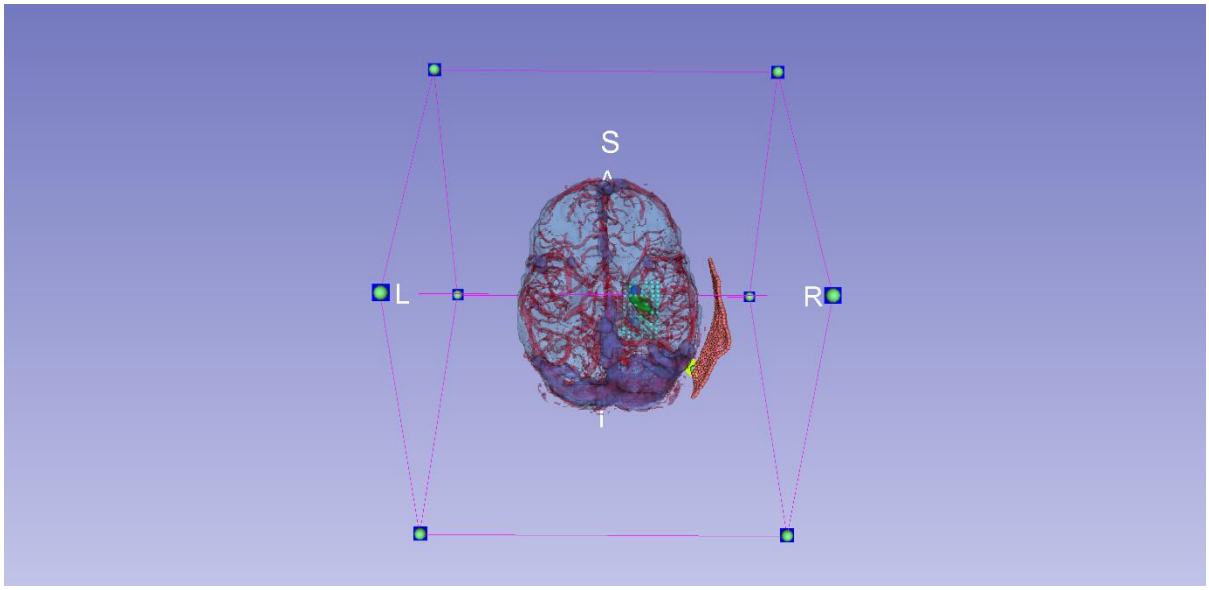

Figure 3: End effector at entry point (view 2)

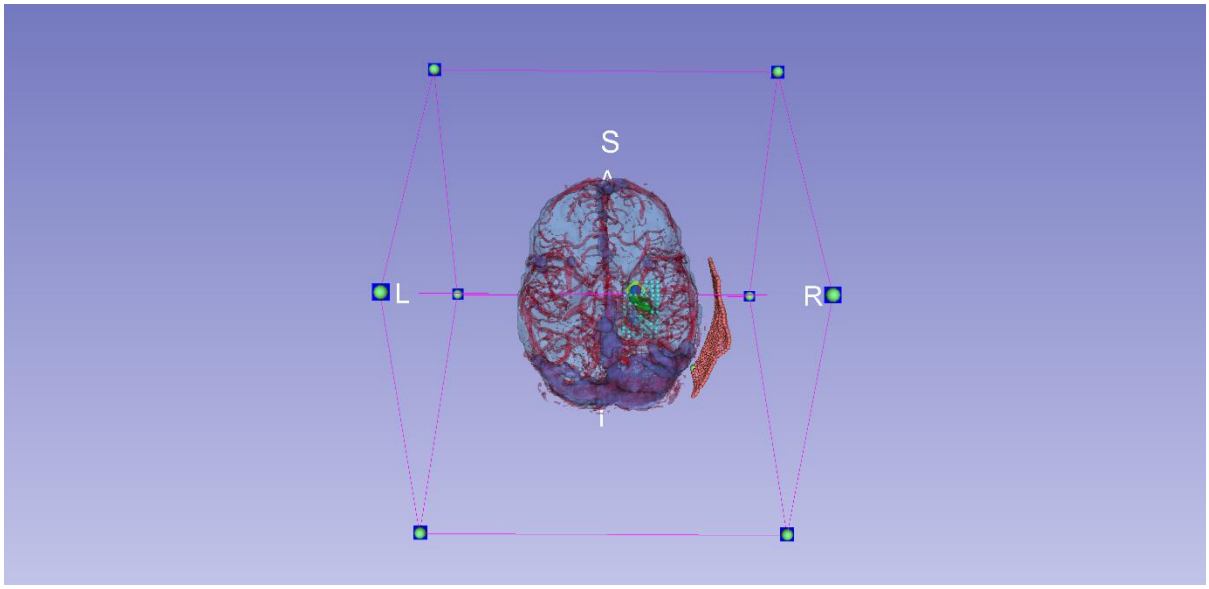

Figure 4: End effector at target point (view 1)

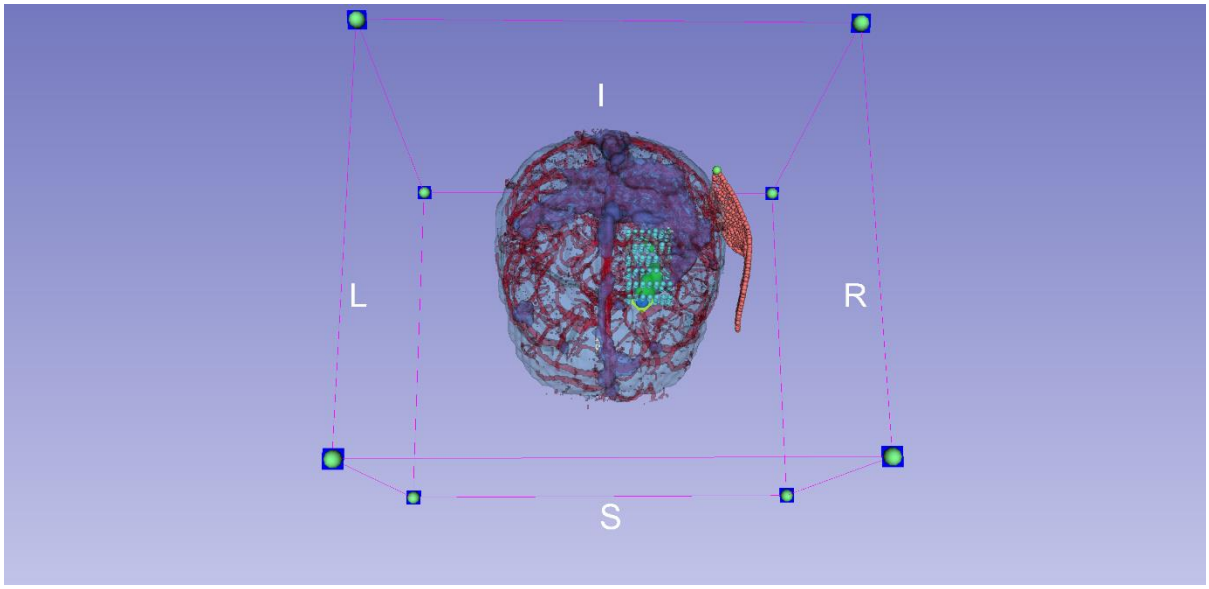

Figure 5: End effector at target point (view 2)

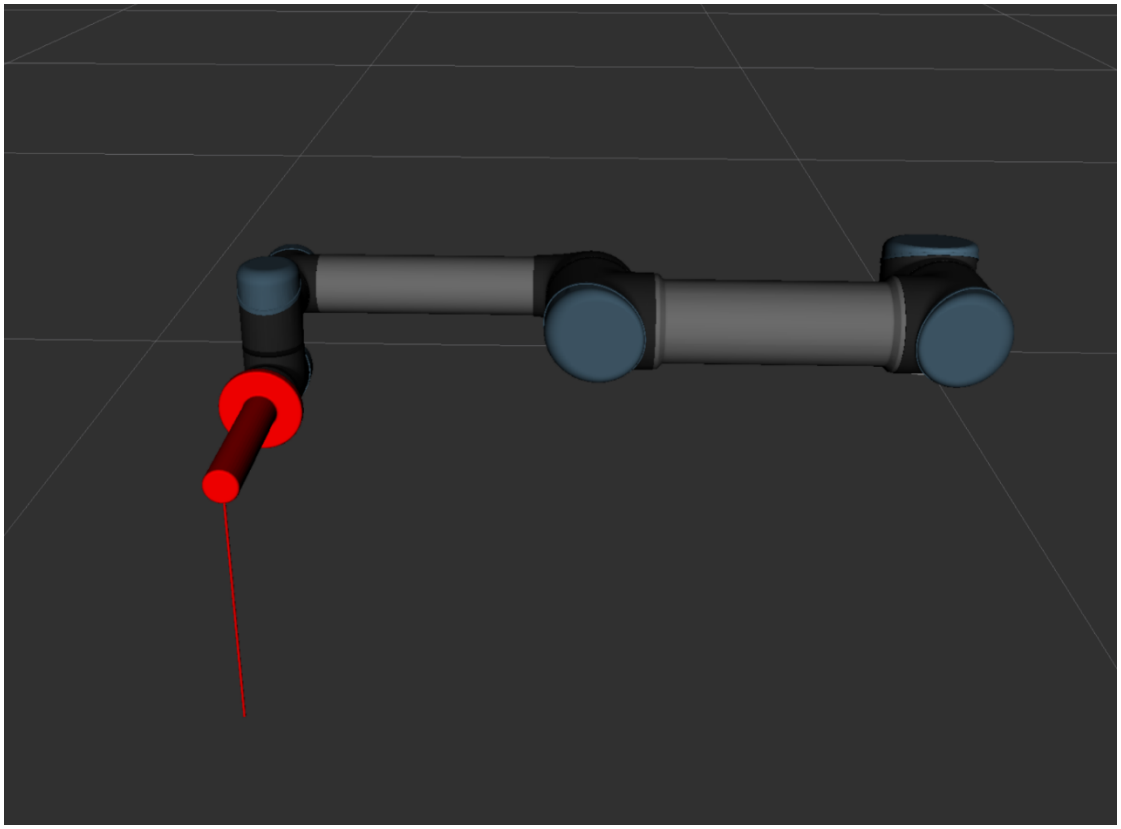

Figure 6: Robot at default position (RVIZ)

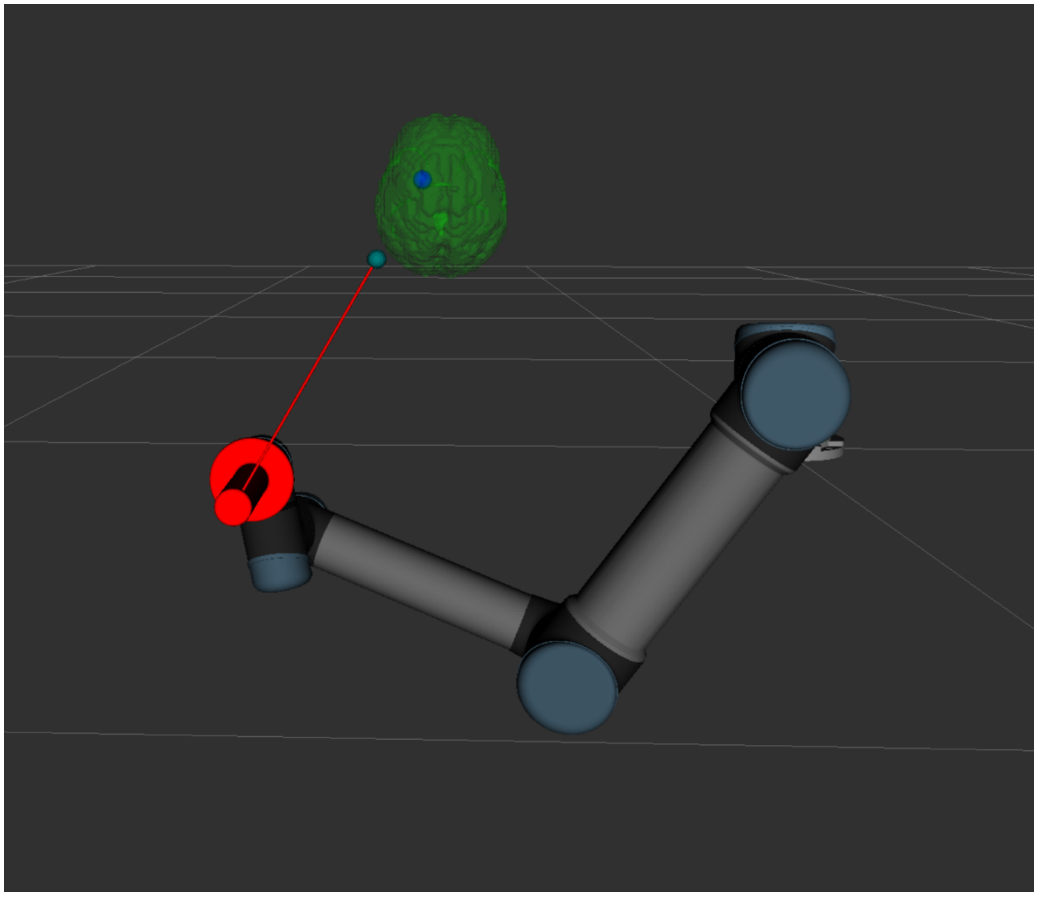

Figure 7: Robot's end effector at entry point (RVIZ)

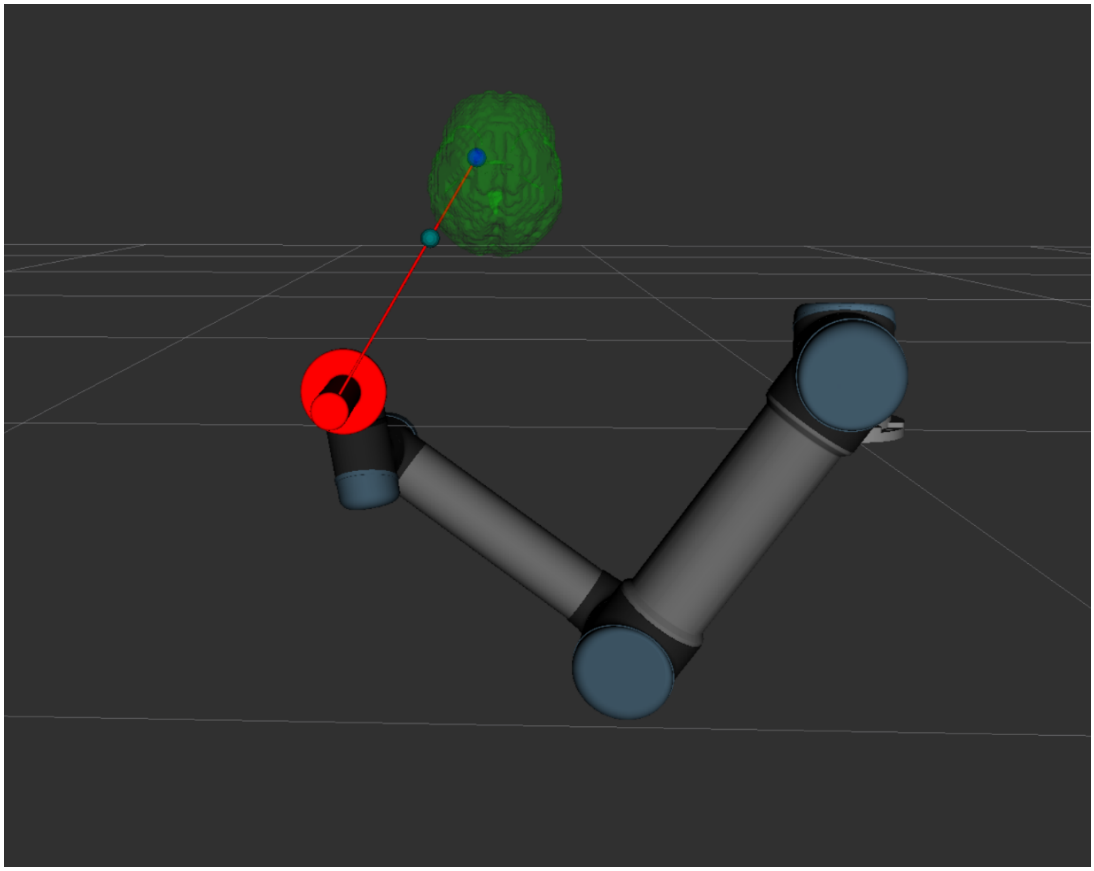

Figure 8: Robot's end effector at target point (RVIZ)

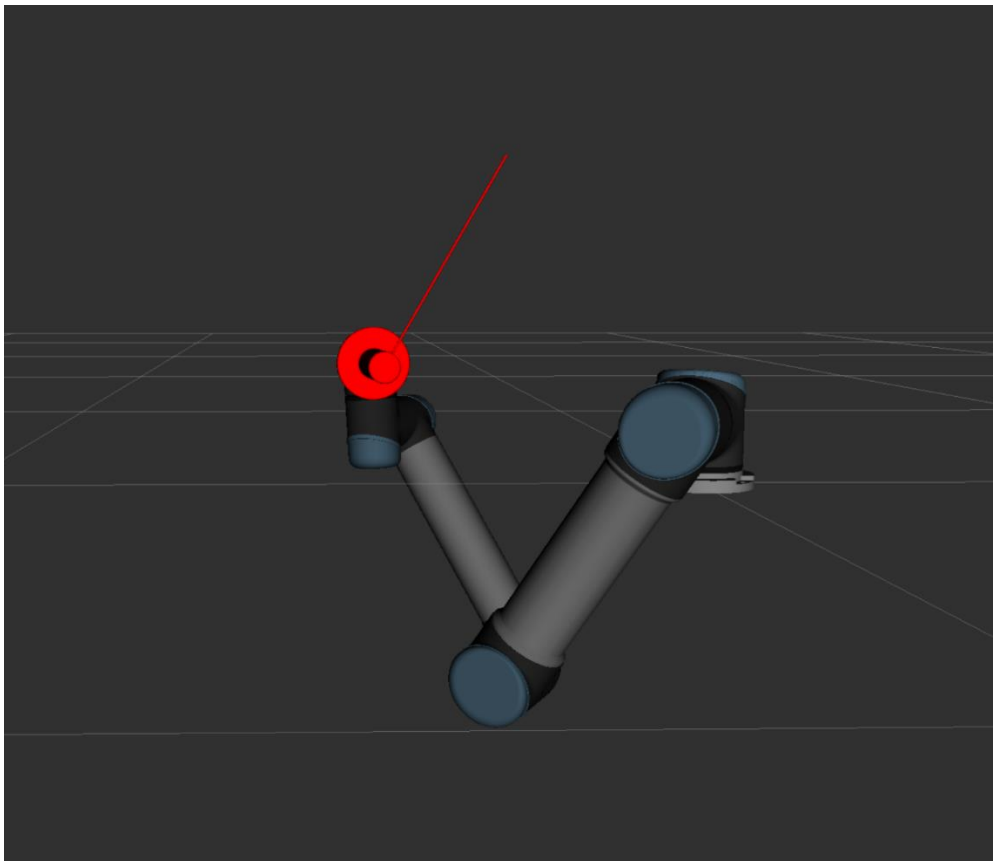

Figure 9: Robot's end effector at a random point (RVIZ)

```
[ INFO] [1590891139.158220620]: Ready to take commands for planning group ur5.
[ INFO] [1590891145.356496420]: ABORTED: No motion plan found. No execution atte
mpted.
[INFO] [1590891145.439868]: position:
  x: 1.5
  y: 0.13778
  z: 0.20744
orientation:
  x: 0.020134196836
  y: -0.259224018741
  z: 0.00539335069024
  w: 0.965592271093
Calibration Point 1 saved in slicer?
```

Figure 10: Invalid position sent to ROS

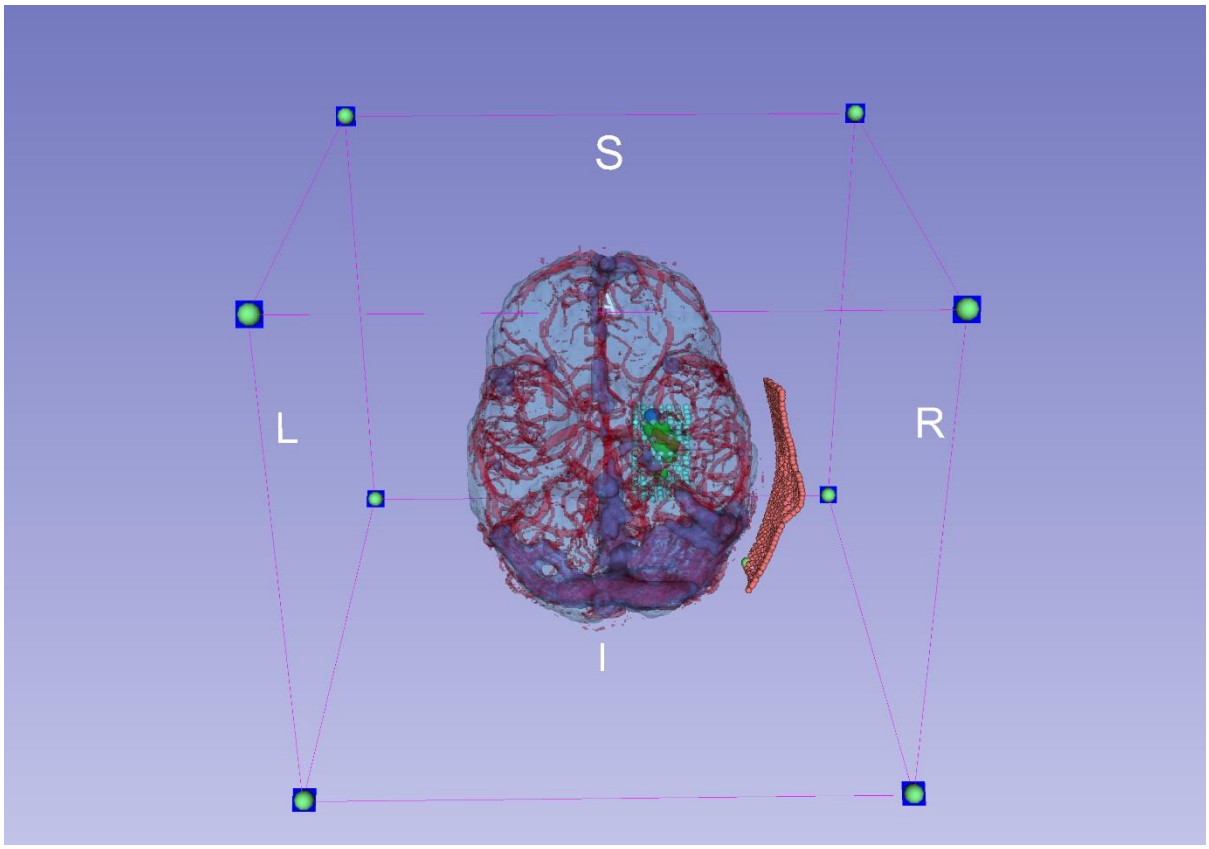

Figure 11: Slicer with models loaded

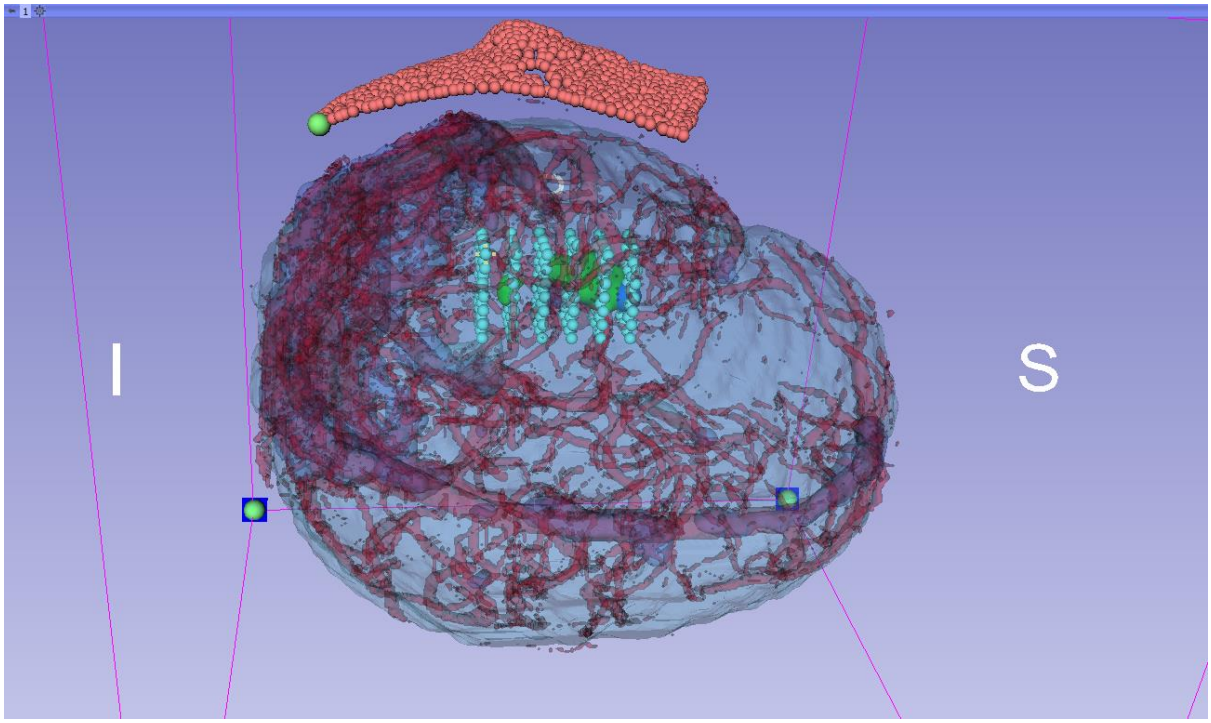

Figure 12: Best points chosen by PathPlanner

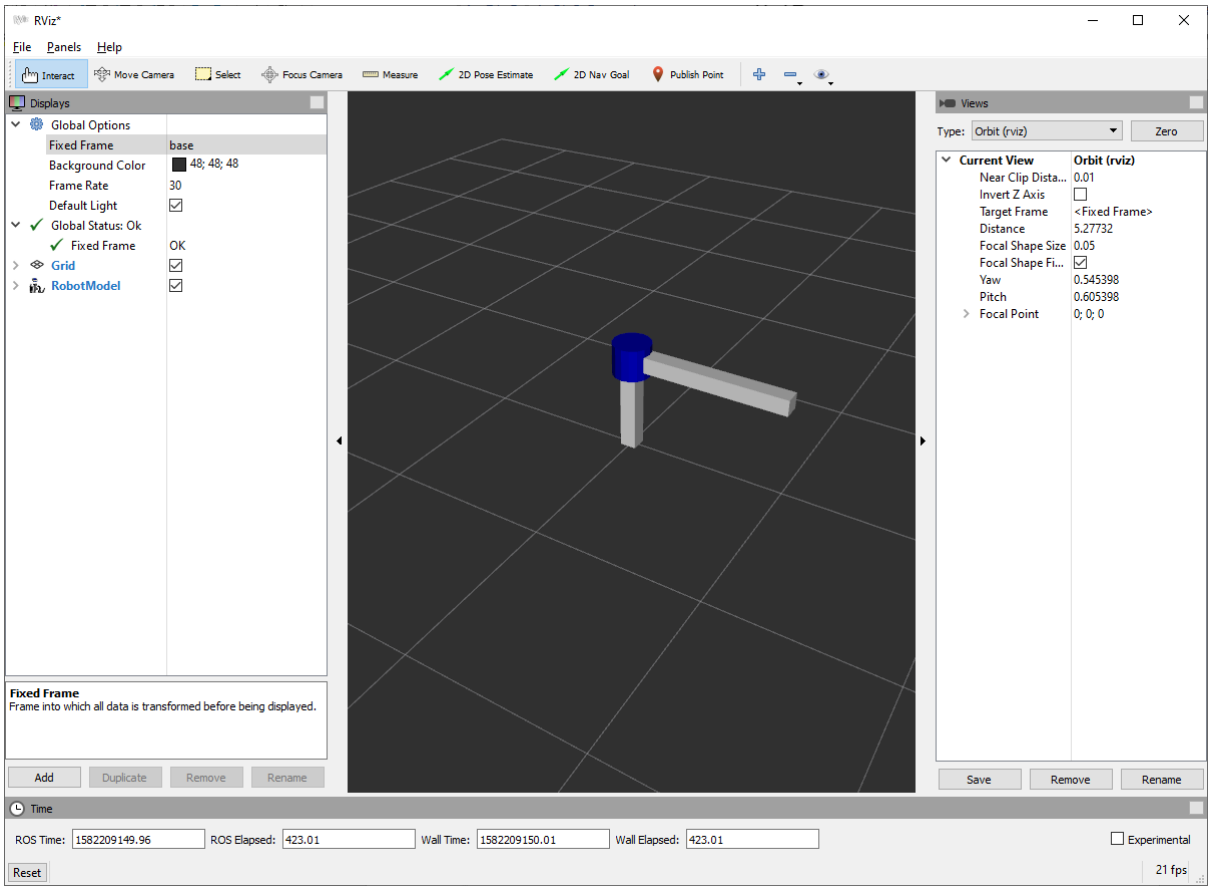

Figure 13: Basic robot

```xml
<?xml version="1.0" ?>

<robot name="r2d2" xmlns:xacro="http://www.ros.org/wiki/xacro">

    <material name="blue">
        <color rgba="0 0 0.8 1"/>
    </material>

    <material name="white">
        <color rgba="1 1 1 1"/>
    </material>

    <link name="base">
        <visual>
            <origin rpy="0 0 0" xyz="0 0 0.25"/>
            <geometry>
                <box size=".1 .1 .5"/>
            </geometry>
            <material name="white"/>
        </visual>
    </link>

    <joint name="lowerJoint" type="revolute">
        <axis xyz="0 0 1" />
        <limit effort="1000.0" lower="-3.14" upper="3.14" velocity="0.5" />
        <origin rpy="0 0 0" xyz="0 0 0.5"/>
        <parent link="base"/>
        <child link="lowerLink"/>
    </joint>
    <link name="lowerLink">
        <visual>
            <origin rpy="0 0 0" xyz="0 0 0.1"/>
            <geometry>
                <cylinder radius="0.12" length="0.2"/>
            </geometry>
            <material name="blue"/>
        </visual>
    </link>
    <joint name="upperJoint" type="revolute">
        <axis xyz="1 0 0" />
        <limit effort="1000.0" lower="-3.14" upper="3.14" velocity="0.5" />
        <origin rpy="0 0 0" xyz="0 0 0.1"/>
        <parent link="lowerLink"/>
        <child link="upperLink"/>
    </joint>
    <link name="upperLink">
        <visual>
            <origin rpy="0 0 0" xyz="0 0.5 0"/>
            <geometry>
                <box size=".1 1 .1"/>
            </geometry>
            <material name="white"/>
        </visual>
    </link>
</robot>
```

Figure 14: Simple robot xacro