RoGaTa Engine Documentation

Release 1.0

Jan Baumgärtner

Jan 11, 2022

Contents


ROGATA

CHAPTER 1

What is the RoGaTa Engine?

The Robotic Games Tabletop engine is a set of ROS based software tools that bridge the gap between mobile robotics and game development. It uses a camera to identify real objects and areas and iinitialize them as game objects. This allows the engine to offer the same functionality a normal game engine provides.

Example functionalities include:

- calculate line of sight between robots without on-board cameras.
- check if a robot has entered a specified area
- simulate laser scanner readings with virtual walls

Using these functionalities it is possible to develop video games which use real robots as actors.

1.1 Who this engine is for

Since most games require multiple robots this engine is primarily aimed at educators. The idea being that games can be set up with students developing the 'ai' of the npcs thereby familiarzing themself with the development of mobile robots.

However the engine can also be used for experiments with single robots as the virtual game objects allow the simulation of sensor data such as a laser scanner. Therefore the engine might also be usefull to researchers who need to quickly set up multiple arenas for their mobile robots, or might want to use the additional information the enigne provides for a fitness function of a machine learning approach

CHAPTER 2

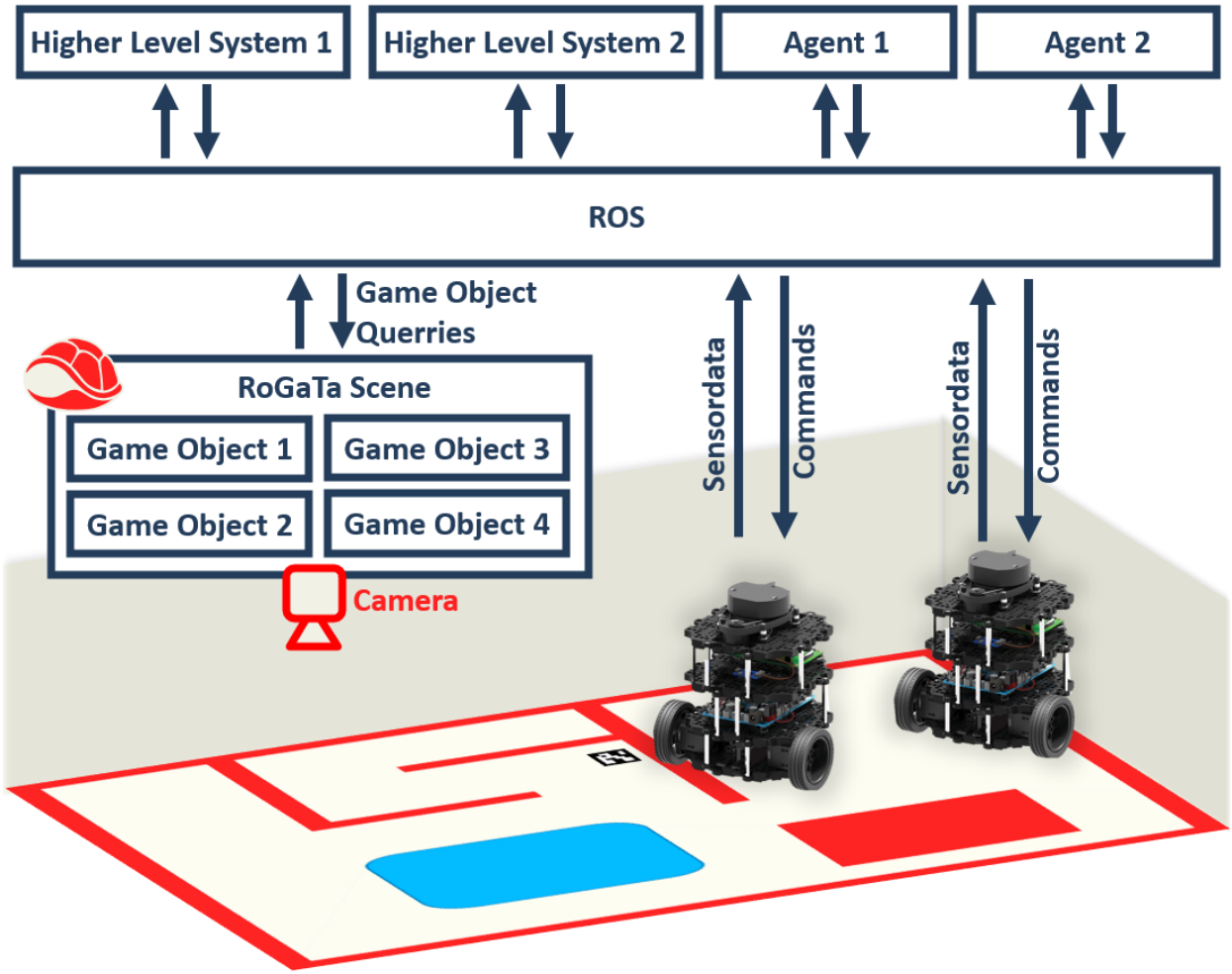
How it Works

The RoGaTa engine defines game objects by detecting and tracking their position using a camera affixed over the game area. To make it quick and easy to setup a game area this is achieved using a mixture of marker based tracking and color based detection.

The game objects are are managed by scenes. Scenes are ROS Nodes that provide service interfaces which allow interaction with the game objects. The services of a scene can be called by other ROS nodes in order to either build higher level systems systems or provide an agent controlling a robot with additional information. For example querrying the position of a game object called 'cheese' can be performed using the following codesnippet:

```
import rogata_library as rgt
rogata = rgt.rogata_helper()
cheese_pos = rogata.get_pos("cheese")
```

This code uses the rogata_helper class which abstracts the ROS communication. The section about Scenes explains how to directly call the different services the RoGaTa engine offers. A schematic view of the communication can be seen here:



In this example there are two ROS enabled robots in the game area. These also share their sensor information and accept commands via ROS interfaces. The rest of this page will go into more detail about how to build game objects and scenes. It will also explain how the RoGaTa engine can be used to interact with these objects to build higher level systems for more complex games or control robots.

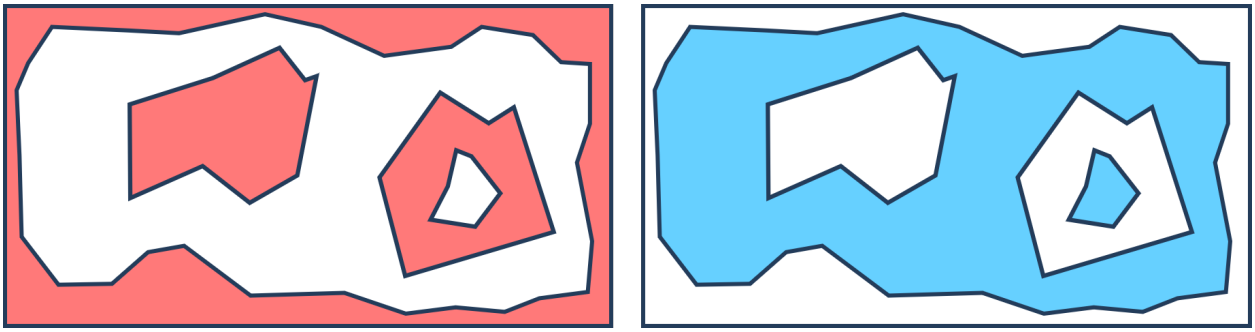
2.1 Game Objects

Game Objects are the basic building blocks of the engine. Examples of game objects include:

- robots
- movable obstacles
- Areas
- Buttons
- Walls

In general, such an object is defined by a name as well as the space it occupies within the game area. The simplest way to define an area is using a mask that specifies for each position whether the point is inside or outside the area. In robotics, such a concept is sometimes also called an occupancenter_ygrid. However, since storing and manipulating such grids is resource-intensive. For this reason, only the borders of an object are used to define its area.

However, borders alone are not enough to fully define an object, since for more complex objects it is not clear what is defined as inside and outside. This can be illustrated in the following image:



To make the area definition unambiguous, a hierarchy can be introduced. The outermost border of an object has a hierarchy of 1, if it has a corresponding inner border it has a hierarchy of -1. Since more complex objects might have another outside border inside an inner border, these are denoted by a hierarchy of 2. The general definition is as follows:

1. Outer borders are denoted by positive numbers a
2. The inner border corresponding to an outer border has a hierarchy of -a
3. The hierarchy a is equal to 1+b where b is the hierarchy of the smallest border surrounding the considered border

An example can be seen in the following image:

Using the border, which is specified as an opencv contour object as well as a name and a hierarchy a game object can be initialized. Its documentation can be seen in *rogata_library.GameObject*.

2.1.1 Interacting with a Game Object

There are multiple ways to interact with Game Objects. Since contrary to conventional game engines, graphic rendering and physics simulation is not needed, these focus mostly on detecting collisions and ray casting.

A full overview of the functionality can be seen in *rogata_library.GameObject*.

2.1.2 Dynamic Game Objects

Dynamic Game Objects are a subclass of Game objects. They differ slightly in their initialization since they also keep track of a marker ID which the engine uses to update their position. Their contour is also built automatically using the specifications of a hitbox.

Currently, only rectangular hitboxes are supported.

2.2 Scenes

Scenes are the equivalent of video game levels. A scene defines which game objects are currently loaded and offers the functionality to interact with them. As such it is initialized using a list of *rogata_library.GameObject*.

Using the objects, a scene offers several ROS communication schemes that allow other nodes to interact with the game objects. These include the following ROS services:

This service allows any ROS node to change the position of a GameObject by providing the desired object's name `NAME` and a new position `POS`. In python the service can be set up and called using:

```
# Set up
from rogata_engine.srv import *

set_position = rospy.ServiceProxy('set_position',SetPos)

# Calling the service
req          = SetosRequest(NAME,POS[0],POS[1])
resp         = set_position(req)
```

Its returned response is a ROS service message containing a boolean value which can be called using:

```
resp.sucess
```

This service allows any ROS node to calculate the intersection of a line with starting point `START`, direction `THETA` and length `LENGTH` and a desired object with name `NAME`. In python the service can be set up and called using:

```
# Set up
from rogata_engine.srv import *
from geometry_msgs.msg import Pose2D

intersect = rospy.ServiceProxy('intersect_line',RequestInter)

# Calling the service
line      = Pose2D(START[0],START[1],THETA)
req       = RequestInterRequest(GameObject,line,length)
resp      = intersect(req)
```

Its returned response is a ROS service message containing the position of the intersection. This intersection can be extracted using:

```
import numpy as np
INTERSECTION_POINT=np.array([resp.x,resp.y])
```

This service allows any ROS node to get the shortest distance between a point `POINT` and the border of an object with name `NAME` .. warning:

```
Note that this means the distance is positive even if the point is inside the object!
```

In python the service can be set up and called using:

```
# Set up
from rogata_engine.srv import *
get_distance = rospy.ServiceProxy('get_distance',RequestDist)

# Calling the service
req          = RequestDistRequest(NAME,POINT[0],POINT[1])
resp         = get_distance(req)
```

It returns a ROS service message containing the distance. This distance can be extracted using:

```
resp.distance
```

This service allows any ROS node to check whether a given point `POINT` is inside a object with name `NAME` In python the service can be set up and called using:

```
# Set up
from rogata_engine.srv import *
check_inside    = rospy.ServiceProxy('check_inside',CheckInside)

# Calling the Service
req             = CheckInsideRequest(NAME,POINT[0],POINT[1])
resp            = check_inside(req)
```

It returns a ROS service message containing a boolean value which is 1 if the value is inside. It can be extracted using:

```
resp.inside
```

Note: The ROS communication interface is very versatile and allows the engine to interface not only with Python scripts but also C++ programs. However, it is also a bit cumbersome to use. For this reason, the *rogata_library.rogata_helper* class can be initialized at the start of any python script. It directly implements the service setup and abstracts it using simple class functions

Warning: Be carefull using the same instance of *rogata_library.rogata_helper* in multiple callbacks or other parallel constructs, this can lead to the programm getting stuck. This happens because the same service is called from the same instance twice and thus only one will receive an answer. Since rospy has no timeout for services this will result in the service being stuck forever.

CHAPTER 3

Getting Started

The RoGaTa engine builds upon **'Ros Noetic'_** to interface with the robots, and OpenCV-Python to calibrate the arena and track all dynamic objects. The first step is thus to install ROS on all robots as well as the host PC.

The host PC refers in this case to the PC which runs the engine which means that additionally OpenCV has to be installed there.

3.1 Setting up the Environment

Once ROS and OpenCV are installed the environment of the engine has to be set up. First, a catkin workspace has to be created. For this catkin tutorial can be followed.

The RoGaTa Engine itself is a ROS package containing nodes for camera-based sensing and a library that provides utilities for game development. The package has to be placed inside the `catkin_ws/src` directory:

```
cd ~/catkin_ws/src
git clone https://github.com/liquidcronos/RoGaTa-Engine
```

Afterward, the `catkin_make` command has to be called inside the `catkin_ws` directory:

```
cd ..
catkin_make
```

If everything is correctly installed, the following command should change the current directory to the one containing the ROS package:

```
roscd rogata_engine
```

The package installation should also install the rogata_library, a python package which allows for writing of of own game objects and extensions to the engine.It is imported by calling

```
import rogata_library as rgt
```

Warning: The rogata_library depends on other packages. While catkin_make should automatically resolve them, it does not check for a minimum version. On old systems this might lead to problems. For this reason the `/src` directory contains a requirements.txt file that can be used to update all the needed python packages. This is done by calling `sudo pip install -r /path/to/requirements.txt`.

3.2 Setting up the Game Area

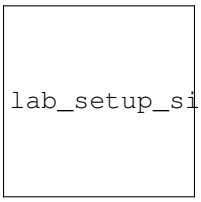
The first step in actually using the engine is to set up the game area. Since the engine needs a camera to calibrate itself and track at least one *rogata_library.dynamic_object*, the first step is setting up a camera.

The Camera needs to be affixed above the game area perpendicular to the ground. In General the higher up the camera the better since it leads to a larger game area. However, it also decreases the precision of the object tracking. This also means that bigger markers have to be used. This can of course be circumvented by using a higher resolution camera. Since higher resolution pictures take longer to process this could however slow down the engine depending on the computer it's running on.

Note: Once the camera is set up, a picture of the game area can be taken. Such a picture can be used as a reference for the size of the game area. This prevents setting up game objects which end up partially outside of the camera view.

After the camera is installed the area itself has to be prepared. Given a desired layout of static objects, color has to be applied to the ground to mark the borders of the object. The easiest way to do this is using colored tape.

The colors of the tape should be chosen to strongly contrast the grounds hue. An example of a prepared game area can be seen below.



3.3 Setting up a Scene

Scenes can be thought of as real-life video game levels. A Scene is made up of multiple game objects whose position is determined by a camera above the physical level. Setting up a scene thus simply means getting the engine to recognize physical objects as game objects.

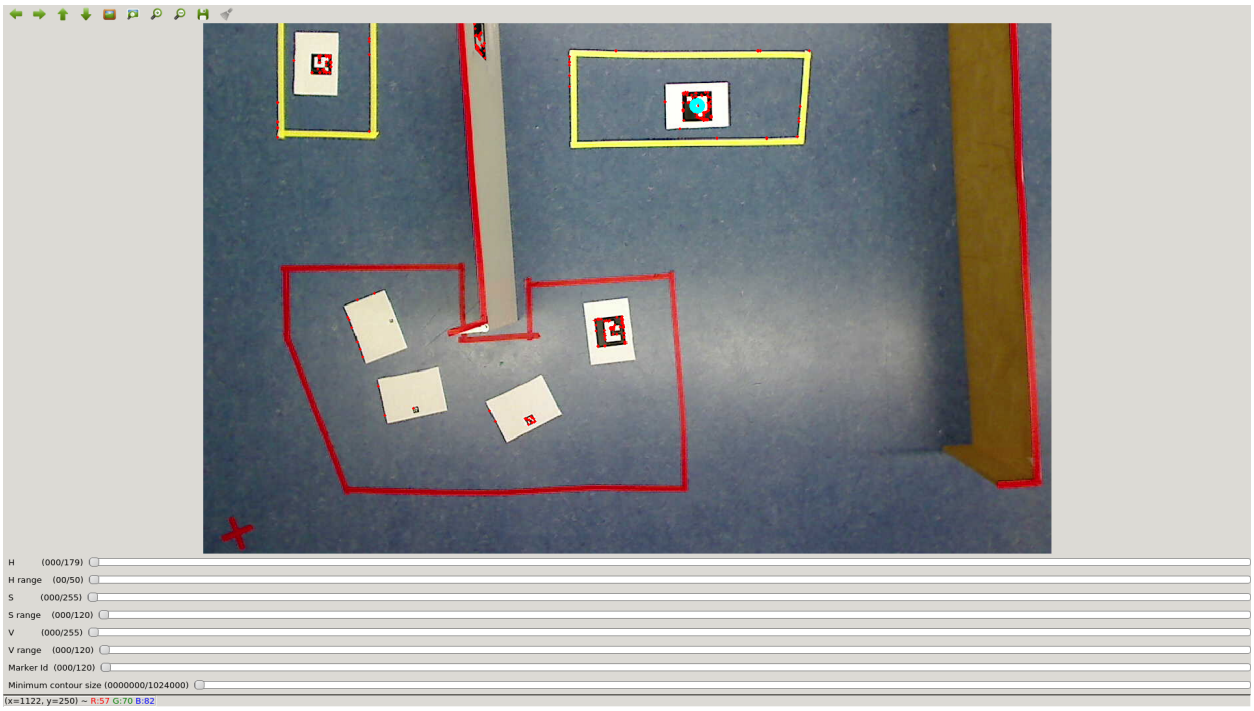
The RoGaTa engine uses a simple recognition based on the color of an object. The full theory behind the detection of game objects can be seen in **'How it Works'_**.

3.3.1 Calibrating the Arena

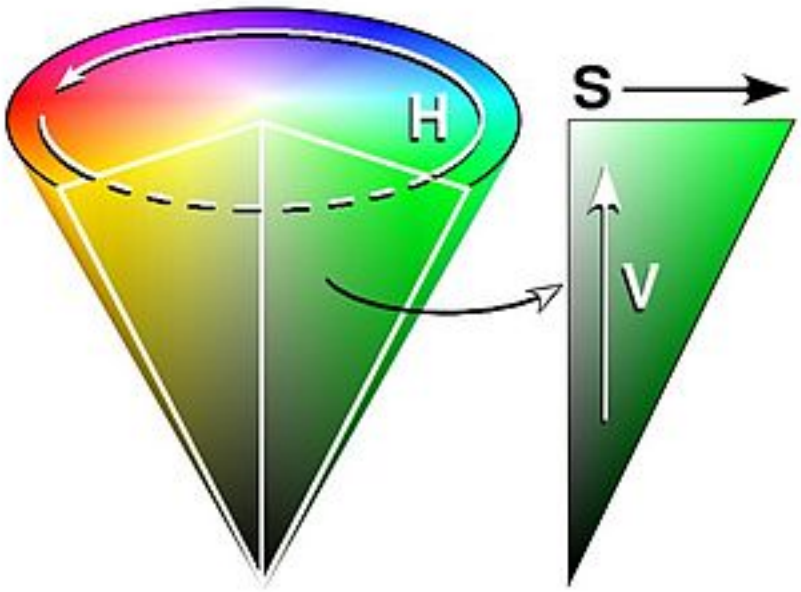
To calibrate the arena the calibrate_scene.py script can be used. It is called using:

```
python calibrate_scene.py PATH_TO_IMAGE
```

Where `PATH_TO_IMAGE` is the path to an image of the scene. This image has to be captured with the same **stationary** camera used to later track dynamic objects. The script will open a window where one can see the image of the scene as well as several sliders:



The first six sliders are used to select the color of the desired object. This color is specified in the HSV colorspace. HSV refers to Hue, Saturation, and Value. A good visualization of the HSV values can be seen in this image from its wikipedia page:



For OpenCV, The Hue Range goes from 0-179 (which is half that of the common Hue definition which goes from 0 to 359). since the Hue value is cyclical, both these values represent a type of red. While the borders between colors are not clearly defined the following table gives an idea which hue values to pick for each object color

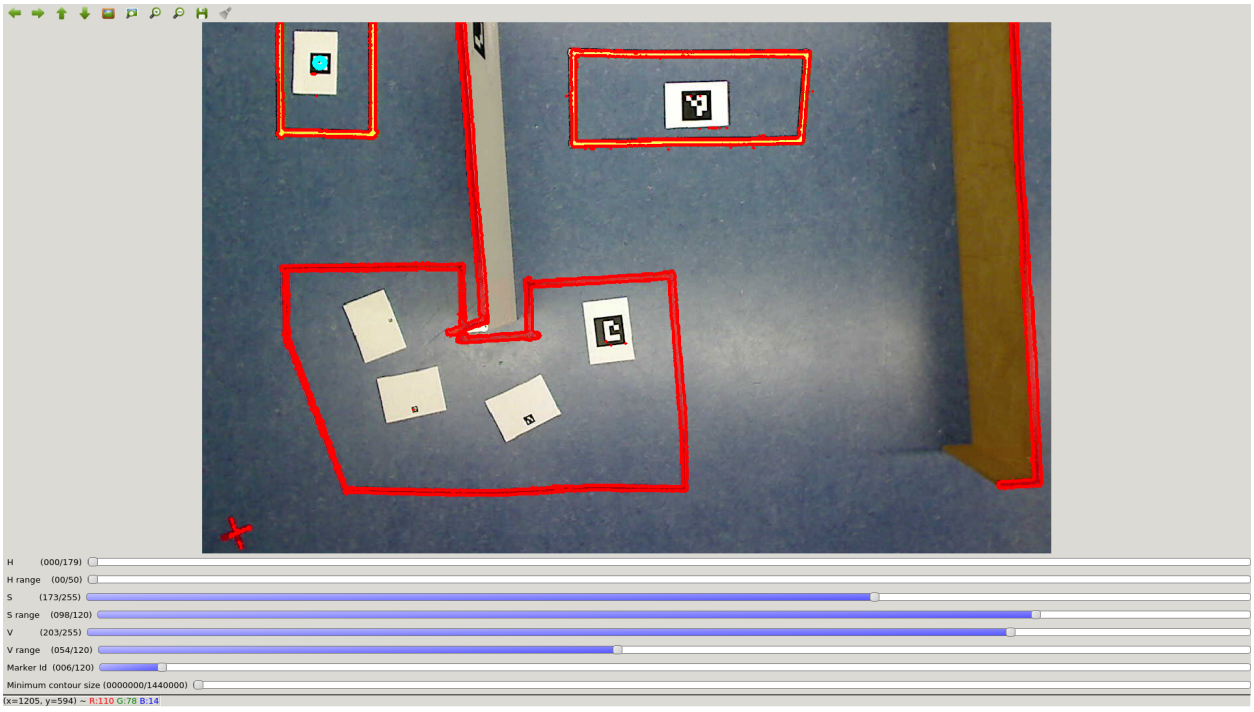
Color	Hue Value Range
Red	0-30
Yellow	31-60
Green	61-90
Cyan	91-120
Blue	121-150
Mageta	151-179

Saturation and Value are defined from 0-255. The less saturated an image, the less colorful it is. Additionally the lower its value, the darker it is.

Due to different lighting conditions across the scene, these last two values will vary for an object. For this reason, each object also has a range slider. Each range *d* specifies an acceptable color interval of *[VALUE-0.5d, VALUE+0.5d]* Where *VALUE* is the value of the main slider.

Note: For fast tuning, it is advisable to first select the desired hue value while setting the saturation and value ranges to the maximum. From there it is easy to dial in the values until the desired object is selected.

Example values for the sliders can be seen in the following image:

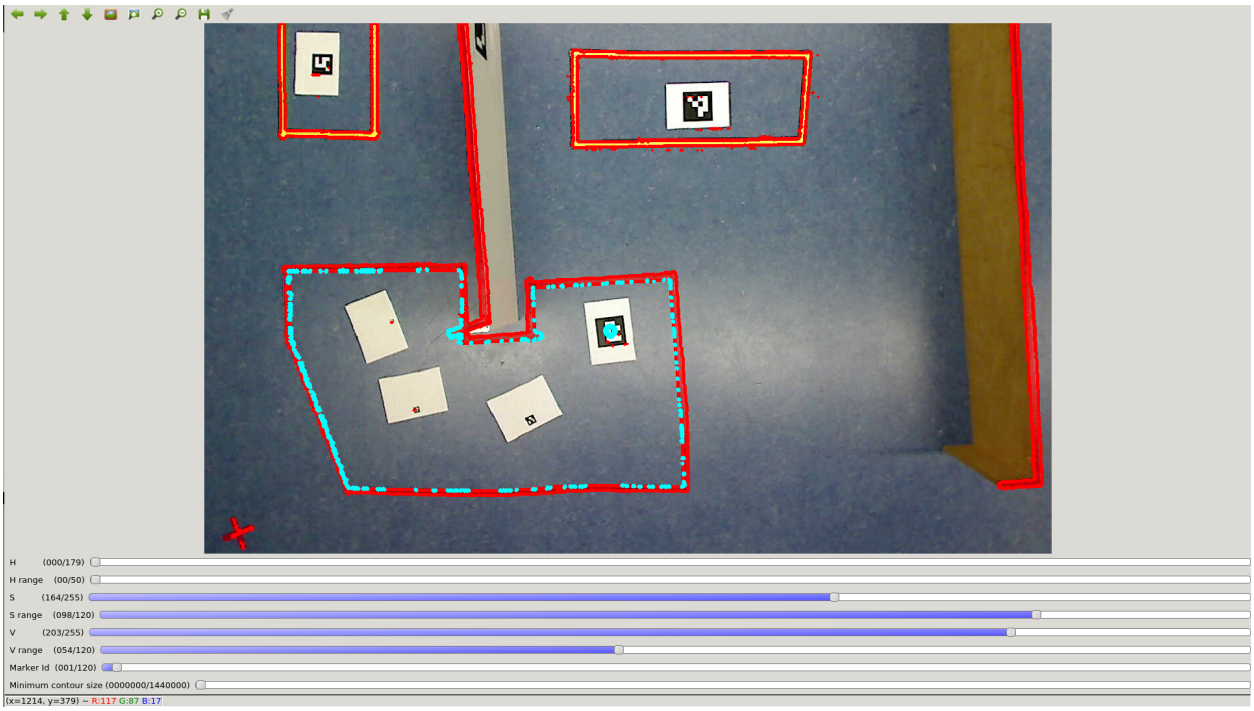


As one can see there are multiple objects outlined in red. To pick the desired object to track, aruco markers are used. If placed inside the desired object, the contour can be selected by specifying the corresponding marker ID using the Marker ID slider.

Note: It is possible to directly specify values by clicking on the slider values on the left

---

Selecting Marker 1 results in the following image:

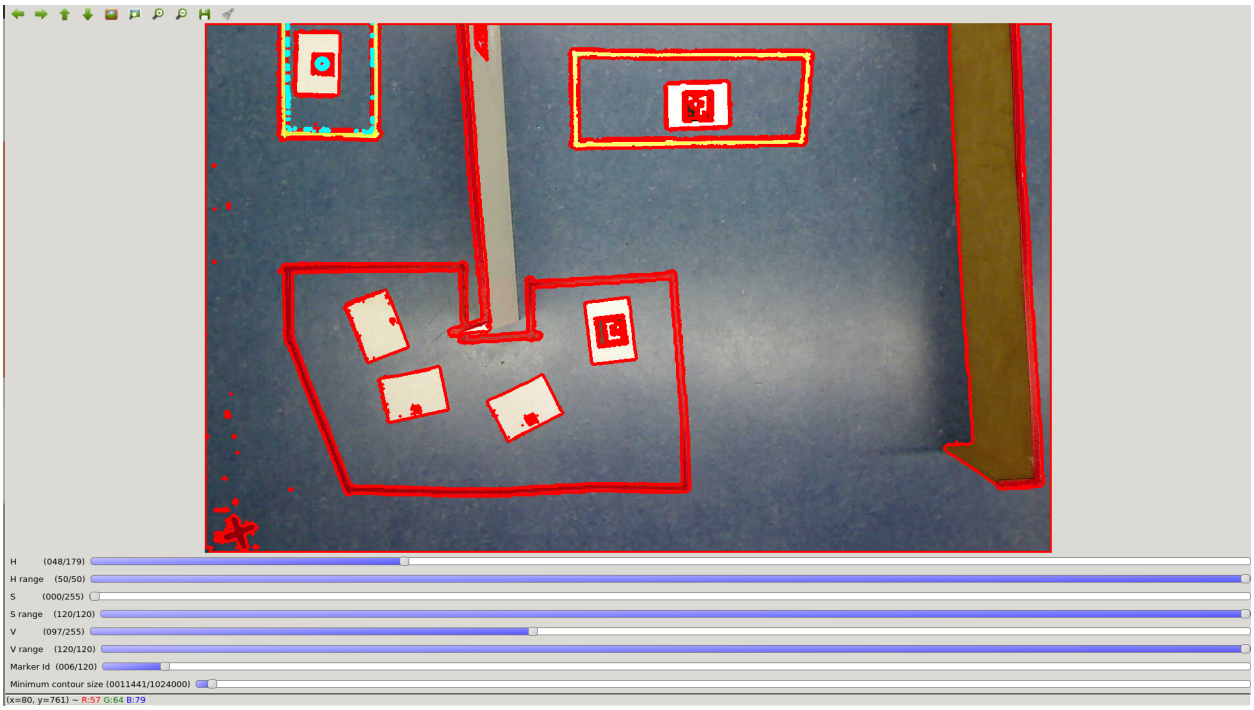


the currently selected contour is now highlighted in torques. It can now be saved by pressing *s* on the keyboard. The Terminal in which the *calibrate_scene.py* script was called will now ask for a file name. If provided it saves the contour as a *.npy* object.

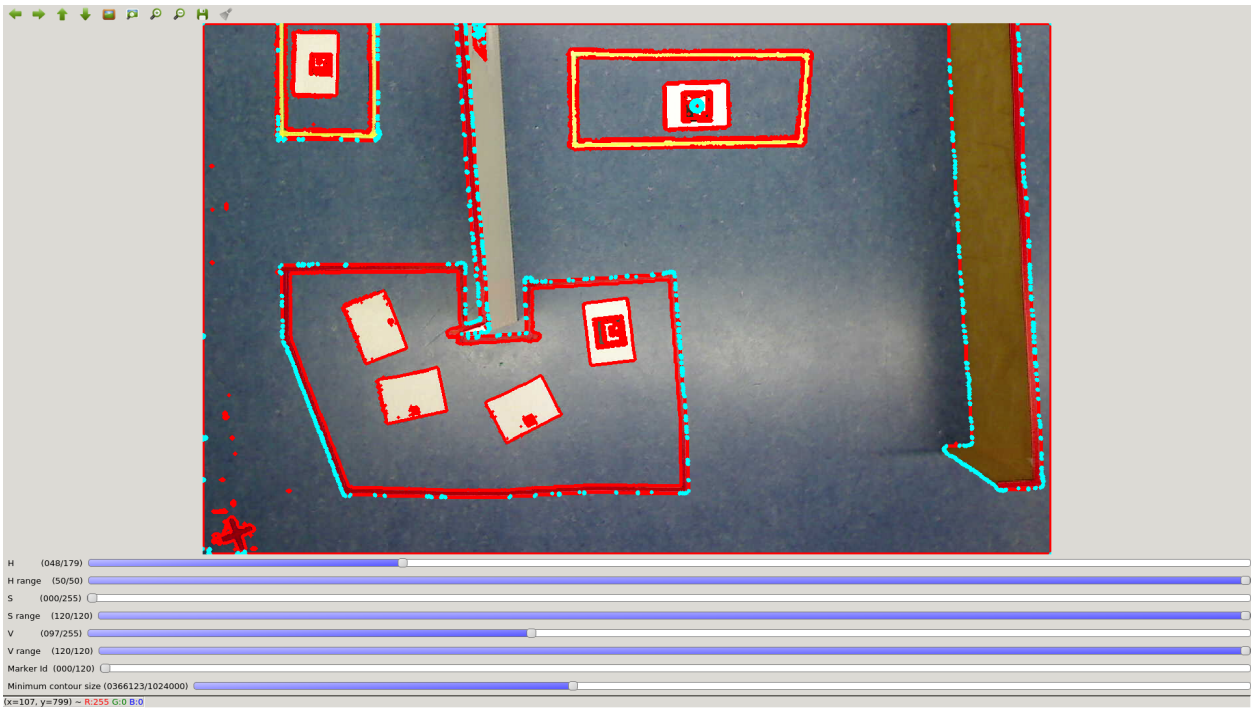
This procedure can now be repeated for each contour.

If there are very small objects inside the scene such as walls or open contours such as the yellow one in the top left a trick can be employed. Instead of using the color of the object, the color of the Ground can be used. This specifies a contour around the desired object which can then be selected.

However, in this case, the border of the marker itself might count as a contour. To circumvent this the Minimum Contour Size slider can be used to specify the minimum size of the chosen contour. This way it is possible to select such open object:



However, this trick might also select other game objects such as seen here when specifying the contour of the wall:



This can be circumvented by first setting up the walls and using an image of the arena without the other objects.

3.3.2 Building Game Objects

Using the contours calibrated in the last section it is possible to set up game objects. The architecture of the game objects is described in the how it works section. In the above example, the game area has two objects, the yellow

`goals` and the red `line of sight breakers`

Note: One can of course also have multiple objects of the same color. This is however only advisable if they behave similarly as it might otherwise be confusing when playing or observing a game.

the first step in setting up the objects is loading the contour objects saved by `calibrate_scene`:

```
import rogata_library as rgt
import numpy as np

# Setting up the goal object
left_goal  = np.load('left_yellow_rectangle.npy')
right_goal = np.load('right_yellow_rectangle.npy')
goals      = rgt.GameObject('goals',[left_goal,right_goal],np.array([1,1]))


# Setting up the line of sight breakers
large_polygon = np.load('red_polygon.npy')
floor_trick   = np.load('floor.npy')
los_breakers  = rgt.GameObject('line_of_sight_breakers',
                               [large_polygon,floor_trick],
                               np.array([1,-1]))
```

Setting up the first object is rather straight forward, the contours are loaded and are used to initialize the game object. The object hierachy is [1,1], since both objects are simple objects sitting side by side.

For the `line of sight breakers` however the floor contour trick was used. Since this mapped the inverse of the red object, so long as one is within the floor contour one is not inside a `line of sight breaker`. For this reason, the object gets a contour hierarchy of -1.

Warning: This only works because the Floor polygon does not include the large red polygon. If the floor contour was saved before the polygon was drawn, inside the polygon would also register as being outside of the object!

In addition to the static objects on the floor, robots themself also need to be set up as a `rogata_library.dynamic object`. Assuming the Robot is equipped with a marker with ID 6, it can be initialized using:

```
# Setting up the robot object
name      = "pioneer_robot"
id        = 6
hit_box   = {"type":"rectangle","height":30,"width":30}
robot_obj = rgt.dynamic_object(name,hit_box,id)
```

the hitbox specifies the space the robot takes up in the arena. For now, this has to be set manually, however, a future script similar to `calbibrate_scene.py` is planned.

3.3.3 Building a Scene

Using these two objects a scene can now be built witch enables the basic functionalities of the engine. It can be set up using:

```
example_scene =rgt.scene([goals,los_breakers,robot_obj])
```

This scene can now be packacked inside a ROS Node:

```
#!/usr/bin/env python
import numpy as np
import rogata_library as rgt
import rospy


# Setting up the goal object
left_goal  = np.load('left_yellow_rectangle.npy')
right_goal = np.load('right_yellow_rectangle.npy')
goals      = rgt.GameObject('goals',[left_goal,right_goal],np.array([1,1]))


# Setting up the line of sight breakers
large_polygon = np.load('red_polygon.npy')
floor_trick   = np.load('floor.npy')
los_breakers  = rgt.GameObject('line_of_sight_breakers',
                               [large_polygon,floor_trick],
                               np.array([1,-1])


# Setting up the robot object
name      = "pioneer_robot"
id        = 6
hit_box   = {"type":"rectangle","height":30,"width":30}
robot_obj = rgt.dynamic_object(name,hit_box,id)


if __name__ == "__main__":
    try:
        rospy.init_node('rogata_engine')
        example_scene =rgt.scene([goals,los_breakers,robot_obj])
    except rospy.ROSInterruptException:
        pass
```

The node can be started with:

```
rosrun PACKAGE_NAME SCRIPT_NAME.py
```

But in order to be recognized the node first has to be made executable using the following command:

```
chmod +x SCRIPT_NAME.py
```

More information about ROS and what a `Package` is can be found in the ROS tutorials. It is strongly encouraged that one familiarizes himself with ROS before trying to use the RoGaTa Engine.

If the scene is initialized correctly it should publish the position of the robot and provide a multitude of services. The existence of the publisher can be checked using

```
rostopic echo /pioneer_robot/odom
```

Which should return odometry values. The services offered by the engine can be checked using

```
rosservice list
```

These should include:

- set_position
- get_distance

- intersect_line
- check_inside

3.4 Tracking Dynamic Objects

The Tracking of dynamic objects, while a game is running, was consciously decoupled from the scene node, because there are multiple approaches suited for different use cases.

If the objects should be tracked with a camera, the **:py:function:'rogata_library.track_dynamic_objects'** function can be used. Using a grayscale image it can track a list of dynamic objects. To use it with ROS one can use cv_bridge to convert published Images of a camera node into usable images:

```
#!/usr/bin/env python
import sys
import cv2
import rospy
import rogata_library as rgt
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import numpy as np



def img_call(ros_img,object_name_list):
    cv_img = bridge.imgmsg_to_cv2(ros_img,'mono8')
        rgt.track_dynamic_objects(cv_img,object_name_list)



if __name__ == '__main__':
    try:
        rospy.init_node("Dynamic Object Tracker")
                bridge       = CvBridge()
                object_numb  = len(sys.argv)
                object_list  = [sys.argv[i] for i in range(2,object_numb)]
        cam_sub      = rospy.Subscriber(sys.argv[1],Image,img_call,object_list)
        rospy.spin()
    except rospy.ROSInterruptException:
        rospy.loginfo("Dynamic Object Tracker  Node not working")
```

However, it is also possible to use the track_image function on a prerecorded video. In some cases, however, tracking with a camera is not beneficial. This is why in general own functions can be written to track such objects. They can then update the Position of the objects using the `set_position` service.

3.5 Using the Engine in Gazebo

One possible for such a scenario would be when using the engine inside a simulation such as Gazebo. While a simulation already provides much of the tools the rogata engine provides it might be beneficial to be able to simulate a given game before implementing it in the real arena.

Scenes can be set up in Gazebo the same way that normal scenes are set up, however instead of a camera the position of dynamic objects is directly provided by the simulation.

Note: For robots this position is usually provided by a Odometry Topic.

However the image used for the setup is in this case not captured by the camera above the scene. Instead an arbitrary image can be used that is then set as the ground texture in Gazebo. A tutorial of how to do this can be found here.

All that remains is to convert the odometry positions of the simulation into pixel position in the game area. Given a specified 2 dimensional scale X_{sim} in meters in the simulation, and image dimensions X_{game} in pixels the conversion for a image point P_{sim} into a point P_{game} is calculated using:

$P_{game} = [P_{simx}, -1 * P_{simy}] * ||X_{game}||/||X_{sim}|| - X_{game}/2$

CHAPTER 4

Tutorials

4.1 Simple Scoreboards

Scoreboards a staple of Video games, defining what constitutes good play. In terms of machine learning, they also have uses for calculating an agent's fitness.

This simple tutorial will give an example of how to implement such a scoreboard. It will assume that the game scene is already set up and can be accessed. For this example, the scene will include several coins that the player needs to collect. Since each coin is independent of the rest, each coin is its own game objects and will follow the naming convention `coin/i` where `i` is an integer smaller or equal to 12.

The complete Scoreboard code now looks like this:

```
#!/usr/bin/env python
import numpy as np
import rogata_library as rgt
from rogata_engine.srv import *
import rospy

class coin_scoreboard():

    def __init__(self,player_list,coin_name,coin_number):
        self.player_list = player_list
        self.coin_name   = coin_name
        self.coin_number = coin_number
        self.collection  = {}

        self.check_inside   = rospy.ServiceProxy('check_inside',CheckInside)

        for players in player_list:
            self.collection[players] = np.zeros(self.coin_number)
            rospy.Subscriber(players+"/odom",self.manage_score,players)

        rospy.spin()
```

(continues on next page)

(continued from previous page)

```
    def manage_score(self,data,player):
        point = np.array([data.pose.pose.position.x,data.pose.pose.position.x])
        already_collected = self.collection[player]
        for i in range(self.coin_number):
            req  = CheckInsideRequest("coint/"+str(i),point[0],point[1])
            resp = self.check_inside(req)

            already_collected[i] = already_collected[i] or resp.inside

        self.collection[player] = already_collected
        rospy.set_param(player+"/score",np.sum(already_collected))


if __name__ = "__main__":
    rospy.init_node("score_board")
    score = coin_scoreboard(["player_1","player_2"],"coin",12)
```

This code might seem complex, the following sections will therefore break it down line by line. The first question one might ask beforehand is why a class is needed to implement this scoreboard. The reason for this is that it is not trivial to store information from a subscriber outside its callback function. Class variables are simply a convenient way to circumvent this problem.

Getting back to the code, the line

```
#!/usr/bin/env python
```

Is needed for every Python ROS Node. This first line makes sure that the script is executed with Python.

```
import numpy as np
import rogata_library as rgt
from rogata_engine.srv import *
from nav_msgs.msg import Pose2D
import rospy
```

These lines declare the needed libraries and modules. Numpy offers convenient array math functions and rospy the ROS interface. `rogata_engine.srv` and `Pose2D` are needed to import the ROS service objects needed to call the RoGaTa engine services.

```
def __init__(self,player_list,coin_name,coin_number):
    self.player_list = player_list
    self.coin_name   = coin_name
    self.coin_number = coin_number
    self.collection  = {}

    self.check_inside   = rospy.ServiceProxy('check_inside',CheckInside)

    for players in player_list:
        self.collection[players] = np.zeros(self.coin_number)
        rospy.Subscriber(players+"/odom",Pose2D,self.manage_score,players)

    rospy.spin()
```

The `__init__` function initializes the class. In this case, this requires a list of player names, the name of the coin object, and the coin number. The last two arguments could also have been hardcoded, but are provided here as inputs to make the class more flexible. The function also initializes the dictionary `self.collection` which keeps track of which coins have already been collected by a given player. It is populated inside the foor loop. Here also a Subscriber

is set up for each player which reads in the position of each player. For this to work, the names of the players have to refer to dynamic objects initialized in a `rogata_library.scene` object.

Additionally a service `self.check_inside` is set up to request information from the engine whether a given point is inside an object. This will later be used to check whether a player has collected a `coin`.

Lastly `rospy.spin()` is called, which ensures that the system stays active and does not immediately terminate.

```
def manage_score(self,data,player):
    point = np.array([data.pose.pose.position.x,data.pose.pose.position.x])
    already_collected = self.collection[player]
    for i in range(self.coin_number):
        req  = CheckInsideRequest("coint/"+str(i),point[0],point[1])
        resp = self.check_inside(req)

        already_collected[i] = already_collected[i] or resp.inside

    self.collection[player] = already_collected
    rospy.set_param(player+"/score",np.sum(already_collected))
```

The `manage_score` function is the callback of the position subscriber. This means it gets called every time the position of the dynamic objects is updated. The returned argument `data` is of type `Pose2D` and first has to be converted to a point. Using the current position of a player, the list of collected coins can now be updated. For each coin, the foor loop calls the `self.check_inside` service to check if a new coin has been collected. The sum of all collected coins is lastly set as a ROS parameter This allows any other ROS node to check the score of a given player with the name `PLAYERNAME` by calling

```
rospy.get_param(PLAYERNAME/score)
```

4.2 Simple Line of Sight Calculation

Another Common Problem is line of sight calculation. While robots can do line of sight calculation with an onboard camera this requires object detection. The RoGaTa engine, allows circumventing these requirements by using line intersection. This allows the design of stealth-like games where a `thief` has to enter an area undetected by one or more `guards`. To calculate whether a `guard` can see a `thief` the following function can be used:

```
import numpy as np
import rogata_library as rgt

rogata = rgt.rogata_helper()

def visibility(guard,thief,wall_objects,max_seeing_distance):
    distance   = np.linalg.norm(thief-guard)
    direction  = (thief-guard)/distance
    direction  = np.arctan2(direction[1],direction[0])

    min_intersect = guard + max_seeing_distance * np.array([np.cos(direction),np.
↪sin(direction)])

    for walls in wall_objects:

        intersection = rogata.intersect(walls,guard,direction,max_seeing_distance)
        if np.linalg.norm(intersection-guard) <= np.linalg.norm(min_intersect-guard):
            min_intersect = intersection
```

(continues on next page)

(continued from previous page)

```
    if np.linalg.norm(min_intersect-guard) >= distance:
        return 1
    else:
        return 0
```

Here the `rogata_helper` class is used to abstract the `get_intersection` service of the engine. The Function defines a line between `guard` and `thief` and checks if this line is intersected by an object that is not see-through. Since multiple such `walls` could exist, the system accepts a list called `wall_objects`. If there is an intersection between the `guard` and the `thief` the line of sight is broken and the function returns `False`. Otherwise, the two see each other and the function returns `True`.

A visualization of the algorithm can be seen here:

4.3 Ray Casting

A laser scanner is a common tool for mobile robots that enables the use of SLAM algorithms and general navigation. However to use such algorithms one has to build up physical walls. Additionally, some robots may not have such sensors.

In both cases, it might be beneficial to simulate a laser scanner that interacts with game objects. This can be done using the ray casting functionality the engine provides. A simple example of such a code can be seen here:

```
def laser_scanner(object_list,test_point,angles):
    scan = np.zeros((len(angles),2))
    for i in angles:
        end_point = np.array([100000,100000])
        for k in range(len(objects)):
            line      = Pose2D(test_point[0],test_point[1],i)
            name=String()
            name.data = objects[k]
            req       = RequestInterRequest(str(objects[k]),line,length)
            response  = inters(req)
            new_point = np.array([response.x,response.y])

            if np.linalg.norm(new_point-test_point) <= np.linalg.norm(end_point-test_
↪point):
                end_point = new_point

        scan[i,:] = end_point
```

Where `object_list` is a list containing the names of the objects with which the laser scanner should intersect and `angles` a list of directions in which the laser scanner measures its distance. This direction is provided using an angle in radians convention.

Warning: Since a ray is cast out for each object to intersect with, the speed of the function scales with the number of objects. For this reason, all walls should be defined in as few objects as possible to preserve performance.

Lastly `test_point` is the origin of the laser scanner in the game area. The result of the function is visualized in the following function, where an angle range in 10-degree increments was chosen.

Note: The Stray blue line in the left top corner is an artifact of suboptimal marker placement with meant that the contour of the outside wall goes around the marker.

4.4 Changing the Robots Dynamics

The Purpose of a game engine is of course not only to observe the movement of robots, but also to influence it. The simplest use case is to change the dynamics of a robot depending on which area it is in. This might mean slowing down the robot when he goes off a *track*, or making him unable to turn while on *ice*.

In both cases, this is achieved by writing a dynamics node that offers a fake command velocity subscriber and which then publishes to the `cmd_vel` topic of the robot.

CHAPTER 5

Code Documentation

class rogata_library.**GameObject**(*name*, *area*, *holes*)

A class defining the most basic game objects of the engine

Parameters

- **name** (*string*) – The name of the object
- **area** (*numpy array*) – A array containin all borders of the object as a *contour <https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html>*
- **holes** (*numpy array*) – Array specifying which border constitutes a inner or outer border

get_position()

returns the position of the objects center

The center in this case refers to the mean position of the object. For a disjointed area this center can be outside of the object itself.

Returns 2D position of the objects center

Return type numpy array

is_inside(*point*)

Checks wheter a point is inside the area of the object

A point directl on the border is also considered inside

Parameters **point** (*2D numpy array*) – A point which is to be checked

Returns a truthvalue indicating wheter or not the point is inside the game object

Return type bool

line_intersect(*start*, *direction*, *length*, *iterations=100*, *precision=0.001*)

calculates the intersection between a line and the border of the object

Iterations and precision are kept at standart values if non are provided

Parameters

- **start** (*numpy array*) – a 2D point which specifies the start of the line
- **direction** (*numpy array*) – a vector specifiying the direction of the line (it will be automatically normalized)
- **length** (*scalar*) – a scalar specifiying the maximum length of the line
- **iterations** (*scalar*) – the number of iterations for the ray marching algorithm used (Default value = 100)
- **precision** (*scalar*) – the precision with which the intersection is being calculated (Default value = 0.001)

Returns 2D position of the intersection

Return type numpy array

move_object(*new_pos*, *rotate=0*)
moves the object to a new position and orientation

Parameters

- **new_pos** (*numpy array*) – new 2D position of the object
- **rotate** (*scalar*) – angle of rotation in radians (Default value = 0)

shortest_distance(*point*)
calculates the shortest distance between the point and the border of the object

Also returns a positive distance when inside the object

Parameters **point** (*numpy array*) – A 2D point which is to be checked

Returns distance to border of the object

Return type scalar

class rogata_library.**Scene**(*game_object_list*)
A class implemennting scene objects comprised of multiple *GameObject* objects. It offers Ros Client interfaces which allow other nodes to request information about the game objects. the communication interfaces are described in the documentation

Parameters **game_object_list** – A list of containing :py:class‘GameObject‘ objects.

handle_get_distance(*request*)
Handles requests to the `get_distance` ROS service server

Parameters **request** (*RequestDistRequest*) –

handle_get_position(*request*)
Handles requests to `get_position` ROS service server

Parameters **request** (*GetPosRequest*) –

handle_inside_check(*request*)
Handles requests to the `check_inside` ROS service server

Parameters **request** (*CheckInsideRequest*) –

handle_line_intersect(*request*)
Handles requests to the `intersect_line` ROS service server

Parameters **request** (*RequestInterRequest*) –

handle_set_position(*request*)
Handles requests to the `set_position` ROS service server

Parameters **request** (*SetPosRequest*) –

rogata_library.**cart2pol**(*cart_x*, *cart_y*)
Converts a point (x,y) into polar coordinates (theta, rho)

rogata_library.**detect_area**(*hsv_img*, *lower_color*, *upper_color*, *marker_id*, *min_size*, *draw=False*)
Detects the contour of an object containing a marker based on color

It always returns the smallest contour which still contains the marker The contour is detected using an image with hsv color space to be robust under different lighting conditions. If draw=True the systems draws all found contours as well as the current smalles one containing the marker onto hsv_img

Parameters

- **hsv_image** (*numpy array*) – a Image in hsv color space in which the contours should be detected
- **lower_color** (*numpy array*) – a 3x1 array containing the lower boundary for the color detection
- **upper_color** (*numpy array*) – a 3x1 array containing the upper boundary for the color detection
- **marker_id** (*scalar*) – the ID of a 4x4 aruco marker which identifies the object
- **hsv_img** –
- **min_size** –
- **draw** – (Default value = False)

class rogata_library.**dynamic_object**(*name*, *hitbox*, *ID*, *initial_ori=0*)
A subclass of the basic *GameObject*.

Dynamic objects are able to change their position and can be tracked via aruco markers. Their current position is published by each :py:class‘Scene‘ containing them.

Instead of initializing the object using a contour a dictionary describing a hitbox needs to be provided. The dynamic object then builds the contour.

Currently only rectangular hitboxes are supported. The dictionary of such a hitbox can be set up as follows:

```
{'type':'rectangle','height':HEIGHT,'width':WIDTH}
```

Where HEIGHT and WIDTH are the objects height and width in the game area.

Parameters

- **name** (*string*) – The name of the object
- **hitbox** (*dictionary*) – A dictionary describing the shape of the objects contour
- **ID** – The ID of an aruco marker which can be used to track the object
- **initial_ori** – The inital orientation of the object in radians [0,2*pi]. Standart value is 0

;type number:

rogata_library.**pol2cart**(*theta*, *rho*)
Converts polar coordinates (theta, rho) into cartesian coordinates (x,y)

class rogata_library.**rogata_helper**
A class for people unfarmiliar with ROS.

It abstracts the ROS service communication with the *Scene* class into simply python functions.

dist(*game_object*, *point*)

Abstracts the `get_distance` ROS service communication to get the distance between a *GameObject* and a point

Parameters

- **game_object** (*string*) – The name of the game object whoose distance should be measured
- **point** (*2D numpy array*) – the point to which the distance should be measured

get_pos(*game_object*)

Abstracts the `get_position` ROS service communication to set the position of a *GameObject*

Parameters **game_object** (*string*) – The name of the game object

inside(*game_object*, *point*)

Abstracts the `check_inside` Ros Service communication to check wheter a given point is inside of a *GameObject*

Parameters

- **game_object** (*string*) – the name of the game object to check
- **point** (*2D numpy array*) – The point to check

intersect(*game_object*, *start_point*, *direction*, *length*)

Abstracts the `intersect_line` ROS service communication to get the intersection between a *GameObject* and a line

Parameters

- **game_object** (*string*) – The name of the game object to intersect with
- **start_point** (*2D numpy array*) – The start of the line
- **direction** (*scalar*) – The direction of the line as an angle following ROS convetion
- **length** (*scalar*) – The length of the line

set_pos(*game_object*, *position*)

Abstracts the `set_position` ROS service communication to set the position of a *GameObject*

Parameters

- **game_object** (*string*) – The name of the game object
- **position** (*2D numpy array*) – The new position of the object

rogata_library.**track_aruco_marker**(*gray_image*, *marker_id_list*)
Tracks a list of aruco markers

Returns None if the marker was not found in gray_image

Parameters

- **gray_image** – A grayscale image in which the marker is to be found
- **marker_id_list** (*list of numbers*) – A list of marker ids

Returns A dictionary of marker positions with the marker_ids as keys

rogata_library.**track_dynamic_objects**(*gray_image*, *object_name_list*)

Function which automatically tracks a list of *dynamic_object* that are part of a *Scene*

The functions returns no position and instead updates the internal state of each *dynamic_object*. This position can be accessed using the interfaces of the *Scene* containing the objects.

Parameters

- **gray_image** – A grayscale image in which the objects should be tracked
- **object_name_list** – A list of containing the names of the objects that should be tracked

calibrate_scene.**calibrate_colors**(*image*)

Utility to calibrate the colors for contour detection

Allows the visual calibration of contours which can be saved by pressing the s key. Colors are defined in HSV color space. For each Value H, S and V the median value as well as the acceptable range can be defined. Additionally the ID of a aruco marker used to specify the wanted contour can be specified. Since this can sometimes lead to the contour being the outline of the marker, the minimum hole size in pixels can be specified.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index