PyCRAM Introduction

Contents

```
import pycram
```

The BulletWorld is the internal simulation of PyCRAM. You can simulate different actions and reason about the outcome of different actions.

It is possible to spawn objects and robots into the BulletWorld, these objects can come from URDF, OBJ or STL files.

A BulletWorld can be created by simply creating an object of the BulletWorld class.

```
from pycram.worlds.bullet_world import BulletWorld
from pycram.world_concepts.world_object import Object
from pycram.datastructures.enums import ObjectType, WorldMode
from pycram.datastructures.pose import Pose

world = BulletWorld(mode=WorldMode.GUI)
```

The BulletWorld allows to render images from arbitrary positions. In the following example we render images with the camera at the position [0.3, 0, 1] and pointing towards [latest are looking upwards along the x-axis.

The renderer returns 3 different kinds of images which are also shown on the left side of the BulletWorld window. (If these winodws are missing, click the BulletWorld window to focus it, and press “g”) These images are:

- An RGB image which shows everything like it is rendered in the BulletWorld window, just from another perspective.
- A depth image which consists of distance values from the camera towards the objects in the field of view.
- A segmentation mask image which segments the image into the different objects displayed. The segmentation is done by assigning every pixel the unique id of the object that is displayed there.

```
world.get_images_for_target(Pose([1, 0, 1], [0, 0, 0, 1]), Pose([0.3, 0, 1], [
```

Objects

Everything that is located inside the BulletWorld is an Object. Objects can be created from URDF, OBJ or STL files. Since everything is of type Object a robot might share the same methods as a milk (with some limitations).

Signature: Object:

- Name
- Type
- Filename or Filepath

Optional:

- Position
- Orientation
- World
- Color
- Ignore Cached Files

latest

If there is only a filename and no path, PyCRAM will check in the resource directory if there is a matching file.

```
milk = Object("Milk", ObjectType.MILK, "milk.stl")
```

Objects provide methods to change the position and rotation, change the color, attach other objects, set the state of joints if the objects has any or get the position and orientation of a link.

These methods are the same for every Object, however since some Objects may not have joints or more than one link methods related to these will not work.

```
milk.set_position(Pose([1, 0, 0]))
```

To remove an Object from the BulletWorld just call the ‘remove’ method on the Object.

```
milk.remove()
```

Since everything inside the BulletWorld is an Object, even a complex environment Object like the kitchen can be spawned in the same way as the milk.

```
kitchen = Object("kitchen", ObjectType.ENVIRONMENT, "kitchen.urdf")
```

Costmaps

Costmaps are a way to get positions with respect to certain criterias. The currently available costmaps are:

- `OccupancyCostmap`
- `VisibilityCostmap`
- `SemanticCostmap`
- `GaussianCostmap`

It is also possible to merge multiple costmaps to combine different criteria.

latest

Visibility Costmaps

Visibility costmaps determine every position, around a target position, from which the target is visible. Visibility Costmaps are able to work with cameras that are movable in height for example, if the robot has a movable torso.

```
import pycram.costmaps as cm

v = cm.VisibilityCostmap(1.27, 1.60, size=300, resolution=0.02, origin=Pose([0,
```

```
v.visualize()
```

```
v.close_visualization()
```

Occupancy Costmap

Is valid for every position where the robot can be placed without colliding with an object.

```
o = cm.OccupancyCostmap(0.2, from_ros=False, size=300, resolution=0.02, origin=
```

```
s = cm.SemanticCostmap(kitchen, "kitchen_island_surface", size=100, resolution=

g = cm.GaussianCostmap(200, 15, resolution=0.02)
```

You can visualize the costmap in the BulletWorld to get an impression what information is actually contained in the costmap. With this you could also check if the costmap was created correctly. Visualization can be done via the 'visualize' method of each costmap.

```
o.visualize()
```

```
o.close_visualization()
```

latest

It is also possible to combine two costmap, this will result in a new costmap with the same size which contains the information of both previous costmaps. Combination is done by checking for each position in the two costmaps if they are zero, in this case to same position in the new costmap will also be zero in any other case the new position will be the normalized product of the two combined costmaps.

```
ov = o + v
```

```
ov.visualize()
```

```
ov.close_visualization()
```

Bullet World Reasoning

Allows for geometric reasoning in the BulletWorld. At the moment the following types of reasoning are supported:

- `stable()`
- `contact()`
- `visible()`
- `occluding()`
- `reachable()`
- `blocking()`
- `supporting()`

To show the geometric reasoning we first spawn a robot as well as the milk Object again.

```
import pycram.world_reasoning as btr

milk = Object("Milk", ObjectType.MILK, "milk.stl", pose=Pose([1, 0, 1]))
pr2 = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
```

We start with testing for visibility

latest

```
milk.set_position(Pose([1, 0, 1]))
visible = btr.visible(milk, pr2.get_link_pose("wide_stereo_optical_frame"))
print(f"Milk visible: {visible}")
```

```
milk.set_position(Pose([1, 0, 0.05]))

plane = BulletWorld.current_bullet_world.objects[0]
contact = btr.contact(milk, plane)
print(f"Milk is in contact with the floor: {contact}")
```

```
milk.set_position(Pose([0.6, -0.5, 0.7]))

reachable = btr.reachable(milk, pr2, "r_gripper_tool_frame")
print(f"Milk is reachable for the PR2: {reachable}")
```

Designators are symbolic descriptions of Actions, Motions, Objects or Locations. In PyCRAM the different types of designators are represented by a class which takes a description, the description then tells the designator what to do.

For example, let’s look at a Motion Designator to move the robot to a specific location.

Motion Designators

When using a Motion Designator you need to specify which Process Module needs to be used, either the Process Module for the real or the simulated robot. A Process Module is the interface between a real or simulated robot, and PyCRAM designators. By exchanging the Process Module, one can quickly change the robot the plan is executed on, allowing PyCRAM plans to be re-used across multiple robot platforms. This can be done either with a decorator which can be added to a function and then every designator executed within this function, will be executed on the specified robot. The other possibility is a “with” scope which wraps a code piece.

These two ways can also be combined, you could write a function which should be executed on the real robot and the function contains a “with” scope which executes something on the simulated robot for reasoning purposes.

latest

```
from pycram.designators.motion_designator import *
from pycram.process_module import simulated_robot, with_simulated_robot

description = MoveMotion(target=Pose([1, 0, 0], [0, 0, 0, 1]))

with simulated_robot:
    description.perform()
```

```
from pycram.process_module import with_simulated_robot


@with_simulated_robot
def move():
    MoveMotion(target=Pose([0, 0, 0], [0, 0, 0, 1])).perform()


move()
```

Other implemented Motion Designator descriptions are:

- Accessing
- Move TCP
- Looking
- Move Gripper
- Detecting
- Move Arm Joint
- World State Detecting

Object Designators

An Object Designator represents objects. These objects could either be from the BulletWorld or the real world. Object Designators are used, for example, by the PickUpAction to know which object should be picked up.

```
from pycram.designators.object_designator import *

milk_desig = BelieveObject(names=["Milk"])
milk_desig.resolve()
```

latest

Location Designator

Location Designator can create a position in cartisian space from a symbolic desctiption

```
from pycram.designators.object_designator import *

milk_desig = BelieveObject(names=["Milk"])
milk_desig.resolve()
```

Location Designators

Location Designators can create a position in cartesian space from a symbolic description.

```
from pycram.designators.location_designator import *
from pycram.designators.object_designator import *

robot_desig = BelieveObject(types=[ObjectType.ROBOT]).resolve()
milk_desig = BelieveObject(names=["Milk"]).resolve()
location_desig = CostmapLocation(target=milk_desig, visible_for=robot_desig)

print(f"Resolved: {location_desig.resolve()}")
print()

for pose in location_desig:
    print(pose)
```

Action Designators are used to describe high-level actions. Action Designators are usually composed of other Designators to describe the high-level action in detail.

```
from pycram.designators.action_designator import *
from pycram.datastructures.enums import Arms

with simulated_robot:
    ParkArmsAction([Arms.BOTH]).resolve().perform()
```

To get familiar with the PyCRAM Framework we will write a simple pick and place plan. This plan will let the robot grasp a cereal box from the kitchen counter and place it on the kitchen island. This is a simple pick and place plan.

latest

```
from pycram.designators.object_designator import *

cereal = Object("cereal", ObjectType.BREAKFAST_CEREAL, "breakfast_cereal.stl",
```

```
from pycram.datastructures.enums import Grasp

cereal_desig = ObjectDesignatorDescription(names=["cereal"])
kitchen_desig = ObjectDesignatorDescription(names=["kitchen"])
robot_desig = ObjectDesignatorDescription(names=["pr2"]).resolve()
with simulated_robot:
    ParkArmsAction([Arms.BOTH]).resolve().perform()

    MoveTorsoAction([0.3]).resolve().perform()

    pickup_pose = CostmapLocation(target=cereal_desig.resolve(), reachable_for=
    pickup_arm = pickup_pose.reachable_arms[0]

    NavigateAction(target_locations=[pickup_pose.pose]).resolve().perform()

    PickUpAction(object_designator_description=cereal_desig, arms=[pickup_arm],

    ParkArmsAction([Arms.BOTH]).resolve().perform()

    place_island = SemanticCostmapLocation("kitchen_island_surface", kitchen_de
                                           cereal_desig.resolve()).resolve()

    place_stand = CostmapLocation(place_island.pose, reachable_for=robot_desig,

    NavigateAction(target_locations=[place_stand.pose]).resolve().perform()

    PlaceAction(cereal_desig, target_locations=[place_island.pose], arms=[picku

    ParkArmsAction([Arms.BOTH]).resolve().perform()
```

Task trees are a hierarchical representation of all Actions involved in a plan. The Task tree can later be used to inspect and restructure the execution order of Actions in the plan.

```
import pycram.tasktree
import anytree

tt = pycram.tasktree.task_tree
print(anytree.RenderTree(tt))
```

latest

```
from anytree.dotexport import RenderTreeGraph, DotExporter

RenderTreeGraph(tt).to_picture("tree.png")
```

```
import sqlalchemy.orm
import pycram.orm.base
import pycram.orm.action_designator

# set description of what we are doing
pycram.orm.base.ProcessMetaData().description = "Tutorial for getting familiar

engine = sqlalchemy.create_engine("sqlite+pysqlite:///:memory:", echo=False)
session = sqlalchemy.orm.Session(bind=engine)
pycram.orm.base.Base.metadata.create_all(engine)
session.commit()

tt.insert(session)
```

```
from sqlalchemy import select

navigations = session.scalars(select(pycram.orm.action_designator.NavigateActio
print(*navigations, sep="\n")
```

```
navigations = (session.scalars(
    select(pycram.orm.action_designator.NavigateAction, pycram.orm.base.Positio
    join(pycram.orm.action_designator.NavigateAction.pose).
    join(pycram.orm.base.Pose.position).
    join(pycram.orm.base.Pose.orientation)).all())
print(*navigations, sep="\n")
```

The world can also be closed with the ‘exit’ method

```
world.exit()
```

latest