Bullet World

Contents

This Notebook will show you the basics of working with the PyCRAM BulletWorld.

First we need to import and create a BulletWorld.

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

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

This new window is the BulletWorld, PyCRAMs internal physics simulation. You can use the mouse to move the camera around:

- Press the left mouse button to rotate the camera
- Press the right mouse button to move the camera
- Press the middle mouse button (scroll wheel) and move the mouse up or down to zoom

At the moment the BulletWorld only contains a floor, this is spawned by default when creating the BulletWorld. Furthermore, the gravity is set to 9.8 m^2, which is the same gravitation as the one on earth.

To spawn new things in the BulletWorld we need to import the Object class and create and instance of it.

```
from pycram.world_concepts.world_object import Object

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

latest

As you can see this spawns a milk floating in the air. What we did here was create a new Object which has the name “ milk” as well as the type `MILK`, is spawned from the file “milk.stl” and is at the position [0, 0, 1].

The type of an Object can either be from the enum ObjectType or a string. However, it is recommended to use the enum since this would make for a more consistent naming of types which makes it easier to work with types. But since the types of the enum might not fit your case you can also use strings.

The first three of these parameters are required while the position is optional. As you can see it was sufficient to only specify the filename for PyCRAM to spawn the milk mesh. When only providing a filename, PyCRAM will search in its resource directory for a matching file and use it.

For a complete list of all parameters that can be used to crate an Object please check the documentation.

Since the Object is spawned, we can now interact with it. First we want to move it around and change its orientation

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

```
milk.set_orientation(Pose(orientation=[1, 0, 0, 1]))
```

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

In the same sense as setting the position or orientation, you can also get the position and orientation.

```
print(f"Position: \n{milk.get_position()}")

print(f"Orientation: \n{milk.get_orientation()}")

print(f"Pose: \n{milk.get_pose()}")
```

latest

Attachments

You can attach Objects to each other simply by calling the attach method on one of them and providing the other as parameter. Since attachments are bi-directional it doesn't matter on which Object you call the method.

First we need another Object

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

```
milk.attach(cereal)
```

Now since they are attached to each other, if we move one of them the other will move in conjunction.

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

In the same way the Object can also be detached, just call the detach method on one of the two attached Objects.

```
cereal.detach(milk)
```

Links and Joints

Objects spawned from mesh files do not have links or joints, but if you spawn things from a URDF like a robot they will have a lot of links and joints. Every Object has two dictionaries as attributes, namely `links` and `joints` which contain every link or joint as key and a unique id, used by PyBullet, as value.

We will see this at the example of the PR2:

```
pr2 = Object("pr2", ObjectType.ROBOT, "pr2.urdf")
print(pr2.links)
```

latest

For links there are similar methods available as for the pose. However, you can only **get** the position and orientation of a link.

```
print(f"Position: \n{pr2.get_link_position('torso_lift_link')}")

print(f"Orientation: \n{pr2.get_link_orientation('torso_lift_link')}")

print(f"Pose: \n{pr2.get_link_pose('torso_lift_link')}")
```

Methods available for joints are:

- `get_joint_position()`
- `set_joint_position()`
- `get_joint_limits()`

We will see how these methods work at the example of the torso_lift_joint:

```
print(f"Joint limits: {pr2.get_joint_limits('torso_lift_joint')}")

print(f"Current Joint state: {pr2.get_joint_position('torso_lift_joint')}")

pr2.set_joint_position("torso_lift_joint", 0.2)

print(f"New Joint state: {pr2.get_joint_position('torso_lift_joint')}")
```

Misc Methods

There are a few methods that don't fit any category but could be helpful anyway. The first two are `get_color()` and `set_color()`, as the name implies they can be used to get or set the color for specific links or the whole Object.

```
print(f"Pr2 forearm color: {pr2.get_link_color('r_forearm_link')}")
```

```
pr2.set_link_color("r_forearm_link", [1, 0, 0])
```

Lastly, there is `get_axis_aligned_bounding_box()`, AABB stands for *Axis Ali* *Box*. This method returns two points in world coordinates which span a rectangl the AABB.

latest

```
pr2.get_axis_aligned_bounding_box()
```

To close the BulletWorld again please use the `exit()` method since it will also terminate threads running in the background

```
world.exit()
```

latest