UNIVERSITÄT DUISBURG ESSEN

Open-Minded

UNIVERSITY OF DUISBURG-ESSEN

FACULTY OF ENGINEERING

INTELLIGENT SYSTEMS LABORATORY

Introduction to Isaac Sim

Introduction, Documentation, and Tutorials

Authors:
Anak Agung Krisna Ananda Kusuma

Tutor:
Fatih Oezgan, M.Sc.

Winter semester 2022/2023
April 12, 2023

Contents

1 Introduction

Based on [12], Metaverse is a network of 3D virtual worlds that is focused on connecting social environment. It can gives enormous possibilities for everyone who wants to create a digital version of the world in various scenarios such as simulation environment, digital twin, and game development. With Metaverse, only human imagination is the limit when it comes to creation. NVIDIA Omniverse is a platform based on Universal Scene Description (USD) that allows individuals and teams to create those digital version of the world by enabling users to build custom 3D pipelines (conceptual model that describes the steps that a graphic systems need to do to render a 3D image into a 2D screen [11])and simulate virtual worlds [6]. There are many application inside NVIDIA Omniverse platform, such as Omniverse Create XR for creating Virtual Reality and Augmented Reality, NVIDIA Drive Sim to accelerate autonomous vehicles in virtual world, and NVIDIA Isaac Sim for simulating robotics application and synthetic data generation to accurately model the virtual environments with real world scenario.
NVIDIA Isaac Sim can be used to build virtual robotic worlds and experiments which allows researchers and practitioners to have a robust, physically accurate simulations and synthetic datasets. It can also simulates sensor data such as RGB-D, Lidar, and IMU and works with ROS/ROS2 using ROS Bridge (extensions to connect Isaac Sim to ROS/ROS2. There are four development workflows that can be used to interact with Isaac sim, which are:

- Isaac Sim UI
- VSCode Python API
- ROS
- Jupyter Notebook Python API

NVIDIA Isaac Sim is built based on two key platform components, which are Omniverse Nucleus and Omniverse Kit. Omniverse Nucleus is the database and collaboration engine that allows groups of people to apply changes and modify the representation of the virtual worlds together that is created by them. It works by having an Omniverse client that can publish changes to virtual worlds and subscribe to the changes in real time using omnivers connectors. Omniverse Kit is a tool for developer to built their own microservices, apps, extensions, plugins, etc for their own ecosystems. Other key components platform including Omniverse Connect, Omniverse Simulation, and Omniverse RTX Renderer. Together, they make up the Omniverse Ecosystem [1].

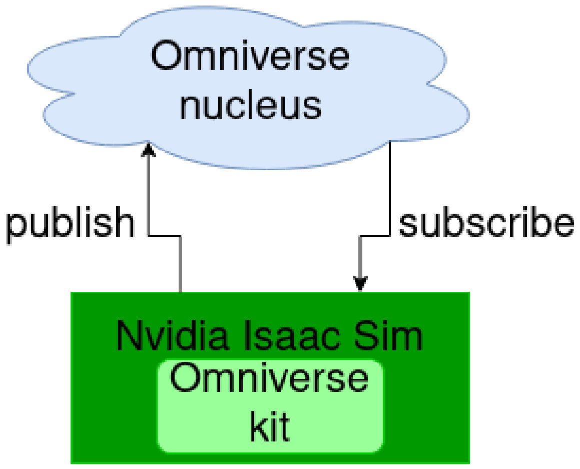

Figure 1: NVIDIA Isaac Sim building blocks

2 Universal Scene Description

Universal Scene Description (USD) is an open source, 3D extensible framework developed by Pixar Animation Studios. It contains a collection of fundamental tools for creating digital assets and virtual worlds and also the tools for creating this digital assets and virtual worlds possible. USD is the foundation of Omniverse. Several key benefits of USD including [9]:

- USD is not only a file format, but it also consist of APIs for composing, editing, rendering, querying, colaboration, and simulating 3D world.
- It provides non destructive workflow in which teams can iterate collaboratively.
- USD is file system agnostic, which means it can receive data in multiple formats or from multiple sources, and still process that data effectively [10]
- USD support custom renderer for visualizing its data

3 Getting Started

To get started with Isaac sim, it is first important to know where is Isaac sim is downloaded in the local pc directory. The directory in which isaac sim is located is called the **root** folder of isaac sim, and this is located in

```
~/.local/share/ov/pkg/isaac_sim-2022.1.1
```

This is where the user can search for the Isaac Sim API, sample code, python committed for isaac sim, and basically everything related to Isaac Sim. To run Isaac Sim, run the command below in the root folder of isaac sim.

```
isaac-sim.sh
```

It takes time until the Isaac sim loads completely. However it is also important to mention that Isaac Sim NEEDS internet to work properly. otherwise, Isaac Sim can not access its assets_root_path which is located on the cloud.
User still can use Isaac Sim offline, but it won't have all the functionality that it should have such as running python code outside Isaac Sim using `python.sh` located on the root folder, access examples from the top menu bar, as well as using USD API for a cleaner python code.
This documentation is all based on the official documentation of Isaac sim which can be found from the link below:

`https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim/overview.html`

4 Isaac Sim Interface

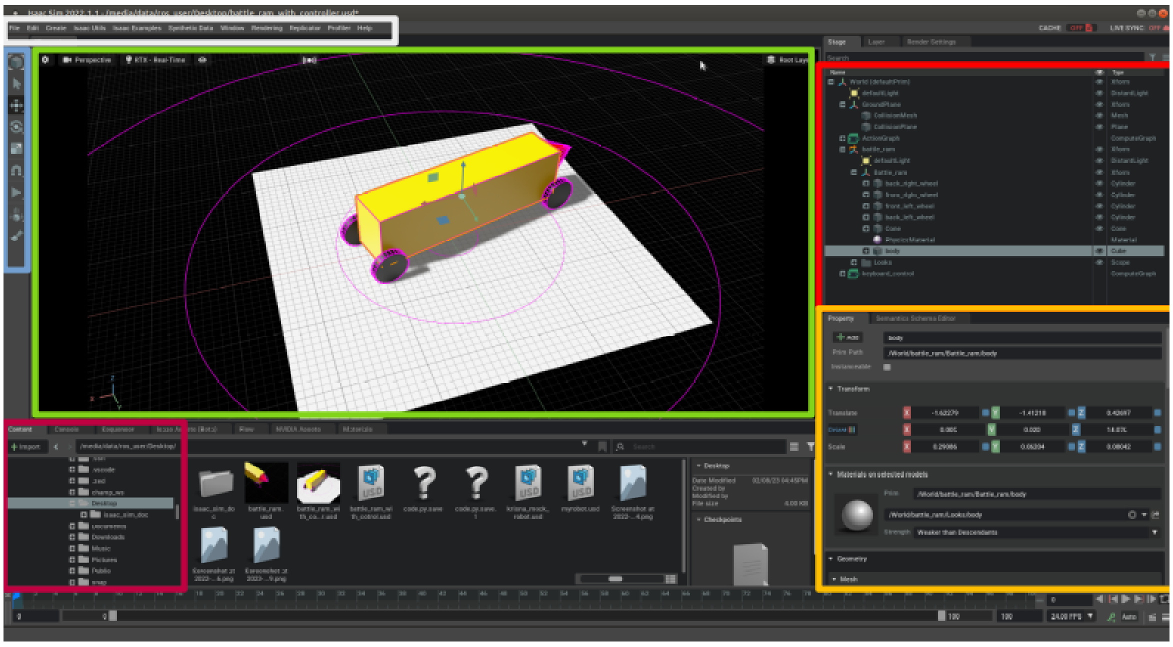

Figure 2: NVIDIA Isaac Sim GUI

Several area of the GUI is bounded with coloured rectangle. Those areas are named based on their functionaloty. Below are the names for each area of the GUI:

- White: **top menu bar**. It is used to create new, safe, and load USD files, insert **prim** to the **stage**, add physics properties, insert more/new type of **window**, load examples, record synthetic data, and much more.
- Blue: left menu bar. It is a quick toolbox for manipulating several **prims's property**. The start button for simulation is also located in this toolbox.
- Green: **viewport window**. This is where all **prims** is visualized in a virtual world with physics properties. Viewport window can also used to display relative view of the virtual world, for example for camera view. To add another viewport, go to the top menu bar > Window and select which type of window you want to display.
- Purple: **Content window**. This is the window to easily search for USD files in your local directory.
- Orange: **Stage window**. This is where all **prim's** parent and child transform tree are displayed. The prim with less tab is the parent and the prim with more tab is the parent's child. To make a certain prim became the child of some prim, use drag and drop functionality.
- Yellow: **Property** panel **window**. This window is to access the respective **prim's property**. Each prim has its own unique **property** to manipulate the **prim's** size, name, angle, etc.

5 Environment setup

5.1 Setting the stage properties

Before start working on adding prims and physics, the stage must be setup to met our requirements. These requirements include the XYZ Axis, stage units, rotation order, etc. For this, go to the top menu bar > Edit > Preferences. For the default settings: The axis that determines up is the **Z axis**, Stage units is in **meters**, and the rotation order is **X-Y-Z**.

5.2 Creating the Physics Scene

Physics Scene is the one that will provide our virtual world with the general physics simulation properties such as gravities and time steps. For this, go to the top menu bar > Create > Physics > Physics Scene. This will bring the Physics Scene prim to the stage. To examine its properties, click the prim and see on the property panel window. Gravity is set to -Z direction with the magnitude of 9.8.

5.3 Adding a Ground Plane

Ground plane will prevent any physics-enabled prims to fall indefinitely. To add Ground Plane, go to the top menu bar > Create > Physics > Ground Plane. To enable grid, on the upper part of the viewport window, there is an ”eye” button, click it, and click grid.

5.4 Add lighting

Lighting will allow us to see what is in the viewport window. If there is no lighting, any prim could not be seen. By default, when creating new stage, the stage is pre-populated with defaultLight prim. To add another light source, go to the top menu bar > Create > Light > Sphere Light. To change its property, click the prim and change it in the property panel window.

6 Robot Creation and Control

6.1 Creation

For building a robot in Isaac Sim, it is worth mentioning the three important terms, which are Prims, Joints, and Articulation. These three terms will be mentioned often when creating a robot in Isaac Sim.

1. Robot is made up of many prims. These prims can be in the form of shape (Square, Cylinder. etc), XForm, Mesh, Camera, Material, Physics, etc (more prims at top menu bar > Create). Each prims must be set manually about its position and orientation relative to other prims and must also be set to have physics properties of a rigid body with collider. To set the position and orientation of the prim, click the respective prim, go to the property panel window, and search for *Transform* tab. This will move the corresponding prim in the world. To make the prim have the physics properties that we need (have mass and affected by gravity and collision-enabled), in the top section of the property panel of the respective prim, click add > physics and then select *Rigid Body With Colliders Preset*. Do this for each of the prims. By then, the objects for crating a robot is ready. However, for assigning a certain friction parameters for the prims, one needs to create a new *PhysicsMaterial* prim from the top menu bar > Create > Physics > Physics Material > Rigid Body Material. The parameter for the friction coefficient and restitution can then be tuned on the property of the *PhysicsMaterial* prim. To assign the robot's prim with this *PhysicsMaterial* prim, select the respective *PhysicsMaterial* prim on the *Materials on selected models* tab of the robot's prim property. For assigning colours to certain objects (prims) of the robot, use the *OmniPBRTwice* prims from the top menu bar > Create > Materials > OmniPBRTwice.

2. Joints is the thing that connects the prims of the robot. To add joint between two prims, select the first prim as the parent prim, and select the second prim while holding ctrl + shift for the child prim. Do this in the stage window. With the two prims highlighted, then right click > Create > Physics > Joints > Revolute Joint. There are other types of joints to be selected. Please note that when applying joints, the two prim's coordinate must be alligned, otherwise, it will resulted in the wrong behavior of the child prim. The added joints can now be seen on the stage window.

   To drive those joints, joint drive is used. adding joints to the robot's prim only gives the mechanical connection. To add joint drive, select the joint from the stage window, then in the upper part of the property panel window of the prim, click add > Physics > Angular Drive.

   There are two types of angular drives, which are position control and velocity control. To set it to become velocity control, set a high damping and zero stiffness in the property of the joint. To set it to become position control, set a high stiffness and relatively low or zero damping.

3. Articulation is a series of connected rigid bodies and joints. Articulation will be used in publishing all the joint states in ROS. To create an articulation, an **articulation root** must be define to anchor the articulation tree. To do this, select the body (prim) of the robot which has the connection to all the joints, then on the property panel window of the body, click add > Physics > Articulation root. Once the articulation root has being defined, it can automatically detects all the joints and becomes articulation.

6.2 Control

There is two ways to create a controller in Isaac sim, and the two will be discussed in the subsection below.

6.2.1 Omnigraph [7]

Omnigraph is Omniverse's visual programming framework. It works by having a block called nodes that connects to other blocks (much like blocks programming). It works mainly for early stage development, where users dont require much programming but instead want to test only the functionality of certain nodes.
To use Omnigraph, go to the top menu bar > Window > Visual Scripting > Action Graph. The Graph editor will appear in the same pane as the Content window. To start a new graph, click the new action graph option. On the left side of the window, there is search bar in which user can search for many different nodes, including controllers, ROS publisher, ROS subscriber, keyboard inputs, etc. These nodes needs to be configured based on user desirable parameter.
To create a controller for wheeled robots, the available type of controller when making this documentation is differential controller and holonomic controller. user can search for this using the keyword "controller" and drag and drop the nodes from the search bar to the graph editor. Differential and holonomic controllers have the functionality to compute drive commands given the some target linear and angular velocities. The parameters that needs to be set for this type of controllers is the wheelDistance, wheelRadius, maxAngularSpeed, and maxLinearSpeed. The input for this nodes is the form of double data type and can be given by other nodes or directly given in the property window of the node. The output of this node is then given to the articulaton controller node.
Articulation controller is the controller that applies the drive commands to the specific joints of any prim with **articulation root**. To insert articulation controller to the graph editor, search for it in the search bar and drag and drop it to the graph editor. To set up the articulaton controller, the parameter that needs to be defined in the property window of the controller is the
input:targetPrim in the and the jointNames int the *inputs* tabs. fill the input:targetPrim with the **articulation root** path of the robot. For joint names, it expects an array of constant token. For this, there is a node called Constant Token and Make Array that can be used to create the input of the articulation controller jointNames. For the Constant Token node, set the value in the *input* tab to joint names of the robot. The number of joints in the robot determines how much Constant Token node needed.

For all the Constant Token created, connect the output of the node to the input of the Make Array node. The Make Array node takes tokens and array size as inputs. Set the arraySize of the Make Array node in the *input* tab to the number of Constant Token connected to the node. Connect the output of the Make Array node to the Joint Names input of the articulation controller.
After having all the required nodes for controlling the robot, one more node is needed to run the event. This node is called On Playback Tick. This node will emit an execution event to all the nodes. Connect the Tick output of the node to the Exec in input of the Articulation controller node and the Differential Controller node. The end result of the action graph would look like the image below.

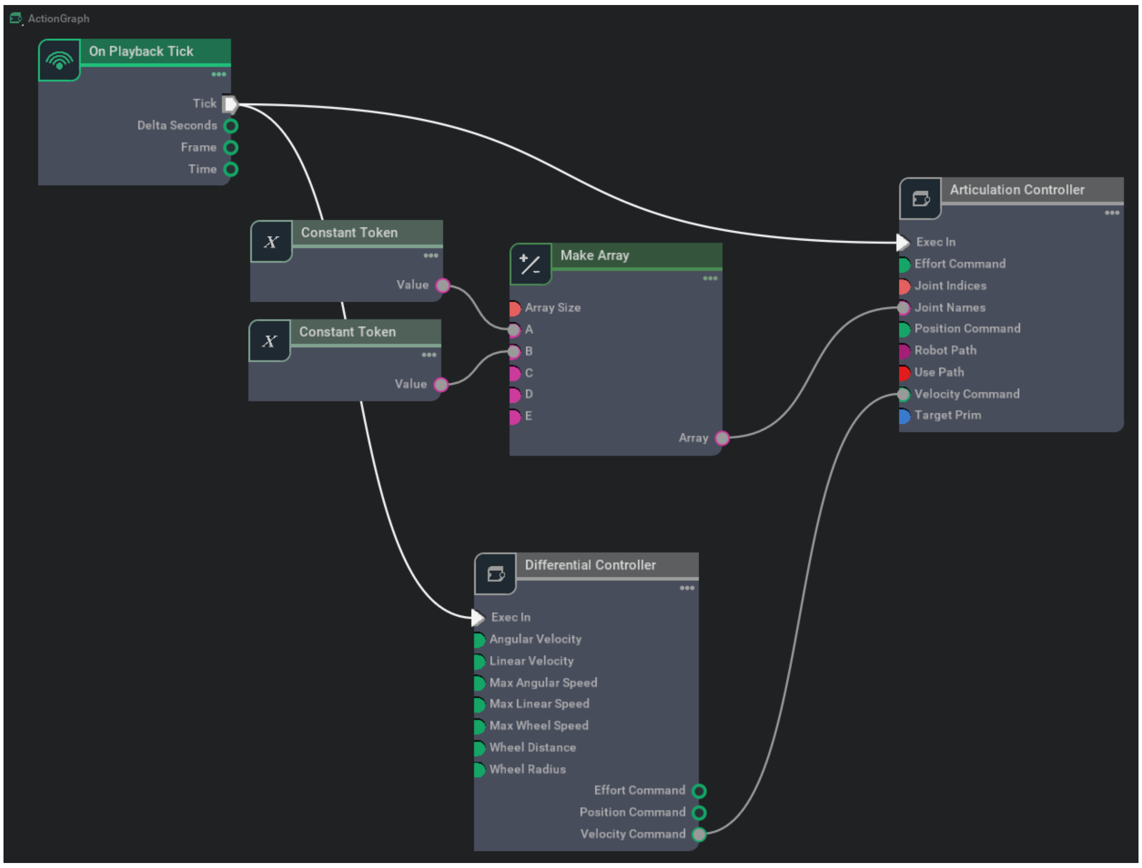

Figure 3: Omnigraph minimal controller

6.2.2 Python Scripting [8]

The difference between using Omnigraph and Python is that, Python uses scripting to create all of the nodes with its parameters as well as the connections. To create one, there one library that needs to be imported to create a controller is only

```
import omni.graph.core as og
```

Then, there are 3 parts of the code, which are

```
og.controller.keys.CREATE_NODES
og.controller.keys.CONNECT
og.controller.keys.SET_VALUES
```

`CREATE_NODES` is used to create the nodes that is available from the search bar. `CONNECT` is used to create the connection from one node output to one node input. `SET_VALUES` is used to set the property values of the nodes. Important to mention that the property values that this function has access to is in the *Raw USD Properties* tab of the property panel window.
For the nodes created using omnigraph in figure 3, the code would look like so:

```python
import omni.graph.core as og

og.Controller.edit(
    #graph_path -> the path and name for your graph
    {"graph_path": "/World/simple_controller", "evaluator_name": "
   execution"},
    {
        og.Controller.Keys.CREATE_NODES: [
            # (name, node:type)
            # name -> any name for your node
            # node:type -> search it in the raw USD properties of the
            # prim property
            ("OnPlaybackTick", "omni.graph.action.OnPlaybackTick"),

            ("ArticulationController",
            "omni.isaac.core_nodes.IsaacArticulationController"),
            ("DifferentialController",
            "omni.isaac.wheeled_robots.DifferentialController"),

            ("MakeArray", "omni.graph.nodes.MakeArray"),
            ("ConstantToken_left","omni.graph.nodes.ConstantToken"),
            ("ConstantToken_right","omni.graph.nodes.ConstantToken"),
        ],
        og.Controller.Keys.CONNECT: [
            # (name.tab:key, name.tab:key)
            # tab -> elements of the specific prim property
            # key -> elements of the specific tab
            ("OnPlaybackTick.outputs:tick",
            "ArticulationController.inputs:execIn"),
            ("OnPlaybackTick.outputs:tick",
            "DifferentialController.inputs:execIn"),

            ("DifferentialController.outputs:velocityCommand",
            "ArticulationController.inputs:velocityCommand"),
            ("MakeArray.outputs:array",
            "ArticulationController.inputs:jointNames"),

            ("ConstantToken_left.inputs:value",
            "MakeArray.inputs:a"),
            ("ConstantToken_right.inputs:value",
            "MakeArray.inputs:b"),
        ],
        og.Controller.Keys.SET_VALUES: [
            # (name.tab:key, value)
            # value -> elements of the specific prim property
            ("ArticulationController.inputs:usePath", True),
            ("ArticulationController.inputs:robotPath",
            "/World/battle_ram/Battle_ram/body"),
            ("DifferentialController.inputs:wheelDistance", 75.0),
            ("DifferentialController.inputs:wheelRadius", 50.0),
            ("MakeArray.inputs:arraySize", 2),
            ("ConstantToken_left.inputs:value",
            "front_left_wheel_joint"),
            ("ConstantToken_right.inputs:value",
            "front_right_wheel_joint"),

            ("DifferentialController.inputs:linearVelocity", -500.0),
            ("DifferentialController.inputs:angularVelocity", 100.0),
        ],
    },
)

```

Apart from the node creation and connection, there are some node values which needs to be configured correctly in order to make the controller work as intended for the specific robot (in this case a differential wheeled robot), and this is done in the SET_VALUES section of the code. Those are:

- Articulation root path
- Robot wheel distance and radius
- Joint names to control

The articulation root path must match with the one in the stage. This is set in the key robotPath in the *inputs* tab of the ArticulationController property window. Robot wheel distance and radius must also match with the wheel on the robot. Otherwise, it will not corresponds to the correct linear and angular velocity. Set this in the wheelDistance and wheelRadius in the *inputs* tab of the DifferentialController property window respectively. Joint names must also corresponds to the correct name of the joints (note that it is not the joints path in the stage). Set this in the value key in the *inputs* tab of the created ConstantToken node property.
To test with the controller, this documentation provide the .usd file as well as the python script called simple_controller.py that you can get from the repository

https://github.com/anakagungkrisna/isaac-sim-documentation.git

First, run Isaac sim by using the command ./isaac-sim.sh from the Isaac sim root directory. Once it is loaded, add a ground plane to the world. Then, open battle_ram.usd in isaac sim from the content window, and then open the python editor from the top menu bar > Window > script editor. Copy and paste it to the bottom part of the script editor, then run it by using ctrl+enter. The battle ram front wheel should turn. If the case it does not, please make sure the

`"ArticulationController.inputs:robotPath"` key match the articulation root of the robot.

7 Isaac Sim with Python [5]

Everything happened in the Isaac Sim Interface has its corresponding python API, which makes all of the function that is done in the interface can be done with a python code. In Isaac sim, there are four ways that user can use python with Isaac sim, and each of them will be described in the subsection below. Important to note that the python for Isaac Sim is different with the python in the local pc. For this, in this documentation, the name ”Isaac sim Python” refers to the python for Isaac sim, and ”local pc python” refers to the python that is built on the local pc.
Isaac sim python has its own list of modules and its own shell script to run. This shell script is located in the root folder of Isaac sim and it is called `python.sh`. This shell script is used to open python console, run python script intended for Isaac sim, as well as installing python modules. The command for installing modules is

```
./python.sh -m pip install name_of_the_package
```

the command for running python script intended for Isaac sim is

```
./python.sh /path/to/your/script.py
```

and the command for opening python shell is

```
python.sh
```

7.1 Script Editor

Script editor is available from the top menu bar > script Editor. This is used to quickly debug with the opened .usd file on Isaac sim. This editor will have access to all of the omni.isaac API on the *standalone_examples/api* directory in the root folder of isaac sim. However, this script editor does not have all of the downloaded python modules on the local pc because the script editor uses the python for Isaac Sim.

7.2 Isaac Python REPL extension

REPL stands for “Read–Evaluate–Print loop”. It is actually a python shell for Isaac sim python. To use the REPL extension, on the top menu bar, click Windows > Extension and search for ”Isaac Sim REPL”. Enable it if it’s not yet enabled. After the extension is enabled, open a new terminal, then run the command `telnet localhost 8223`. A new python shell should start, and with it the user can now interact with everything on the stage. To exit the python shell, press ctrl+d

7.3 USD API

USD API can be used in user-created python script. However, it needs to be run using `python.sh` shell script instead of the normal python on the local pc. This API is simpler than the Isaac Sim Core API and it is recommended for beginners to use this API for working with Isaac sim. The API can be seen in the directory ... of the root folder.

7.4 Isaac Sim Core API

Isaac Sim Core API are detailed and many of the functionality of the Isaac Sim is created using this API. However its quite complex to work with for beginners. The way it is used is also the same as USD API, where user can have a user-created python script. To see the available Isaac Sim Core APIs, the directory is on /exts/omni.isaac.core/omni/isaac/core of the root folder. Using this API produces much more lightweight and readable code.

Despite which method that is used for interacting NVIDIA Isaac Sim with Python, it is important to note that Isaac Sim uses internet to run properly. Because of this, some API can not be accessed for Isaac Sim. In order to use Python for offline Isaac Sim, it is suggested to use Script Editor instead of the shell command `python.sh`. For the complete online documentation about the API, the link is as follows:

```
https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim/
reference_python_api.html#isaac-sim-python-manual
```

8 Isaac Sim with ROS [4]

The extensions that allows Isaac Sim to interact with ROS is called ROS and ROS2 Bridge. The way it works is it provides a common set of components to the data that is published or received between Isaac Sim and ROS. This allows user to publish and subscribe to rostopics and rosservices that works with Isaac Sim. Important to note that ROS and ROS2 Bridge can not be enabled both at the same time. By default, ROS bridge is enabled and ROS2 bridge is disabled. To set this, from the top menu bar > Window > Extensions. Search for ”ros”, and there should be ROS bridge and ROS2 bridge. choose which one to enabled by using the radio button.
Similar to creating a controller for the robot, there are two ways to create a publisher subscriber node in Isaac Sim, which are using Omnigraph and Python scripting. The two methods will be explained below.

8.1 Omnigraph

To use Omnigraph, go to the top menu bar > Window > Visual Scripting > Action Graph. Click the New Action Graph button, and add the nodes as well as the connection as the image below.

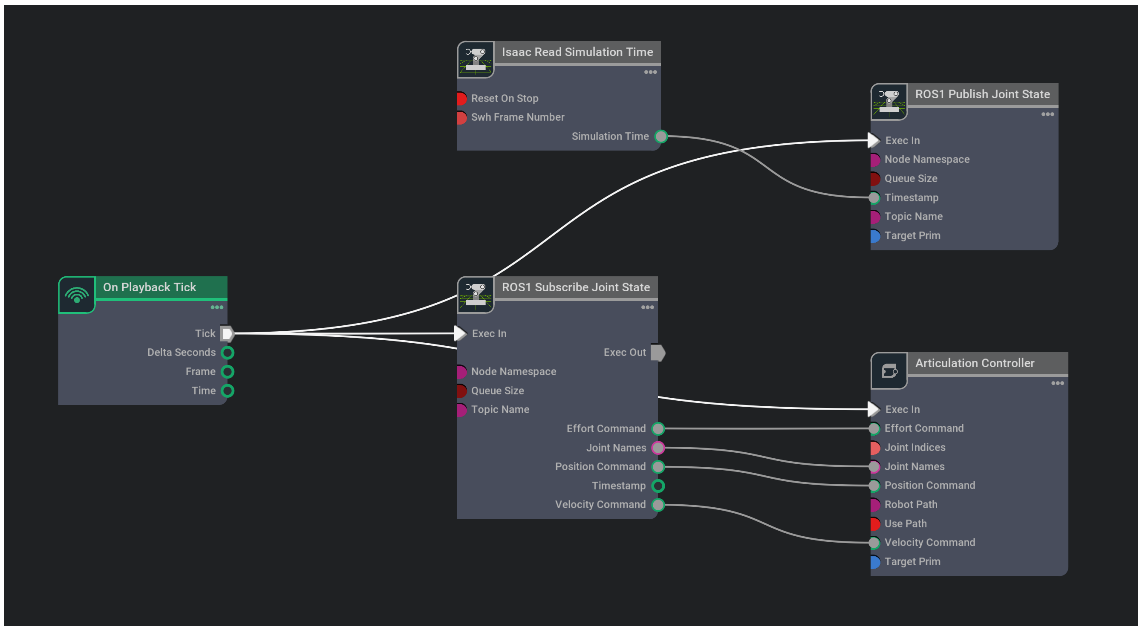

Figure 4: Joint State Publisher and Subscriber

The image shows a very simple method on creating a ROS publisher and subscriber node that publish the joint states of the robot and subscribe from a specific topic to move the joints of the robot through articulation root. Important to know that the subscribed or published topic must have the same message type as the node in Isaac Sim. Otherwise, it will not work. There are many nodes in Isaac Sim which handles various ROS message type, and it can be search from the search bar in Action Graph window by typing the keyword ”ros 1”

8.2 Python script

Using python script to add the nodes as well as the connections is done similarly as in the case for robot control creation. There are two library that is used for this purpose, which are called

```
import omni.graph.core as og
from omni.isaac.core_nodes.scripts.utils import set_target_prims
```

og will act as the main node creation and connection library and `set_target_prims` is used for setting a key in isaac sim ROS publisher node to the robot articulation root. The complete python script to create the ROS nodes that publish robot joint state as well as subscribing to a joint command (as in figure 4) is as follows:

```python

import omni.graph.core as og
from omni.isaac.core_nodes.scripts.utils import set_target_prims

og.Controller.edit(
    {"graph_path": "/World/BackWheelPubSub", "evaluator_name": "
    execution"},
    {
        og.Controller.Keys.CREATE_NODES: [
            ("OnPlaybackTick",
            "omni.graph.action.OnPlaybackTick"),
            ("PublishJointState",
            "omni.isaac.ros_bridge.ROS1PublishJointState"),
            ("SubscribeJointState",
            "omni.isaac.ros_bridge.ROS1SubscribeJointState"),
            ("ReadSimTime",
            "omni.isaac.core_nodes.IsaacReadSimulationTime"),
            ("ArticulationController",
            "omni.isaac.core_nodes.IsaacArticulationController"),
        ],
        og.Controller.Keys.CONNECT: [
            ("OnPlaybackTick.outputs:tick",
            "PublishJointState.inputs:execIn"),
            ("OnPlaybackTick.outputs:tick",
            "SubscribeJointState.inputs:execIn"),
            ("OnPlaybackTick.outputs:tick",
            "ArticulationController.inputs:execIn"),

            ("ReadSimTime.outputs:simulationTime",
            "PublishJointState.inputs:timeStamp"),

            ("SubscribeJointState.outputs:jointNames",
            "ArticulationController.inputs:jointNames"),
            ("SubscribeJointState.outputs:positionCommand",
            "ArticulationController.inputs:positionCommand"),
            ("SubscribeJointState.outputs:velocityCommand",
            "ArticulationController.inputs:velocityCommand"),
            ("SubscribeJointState.outputs:effortCommand",
            "ArticulationController.inputs:effortCommand"),
        ],
        og.Controller.Keys.SET_VALUES: [
            ("ArticulationController.inputs:usePath", True),
            ("ArticulationController.inputs:robotPath",
                "/World/battle_ram/Battle_ram/body"),
            ("PublishJointState.inputs:topicName", "
    joint_states_battleram"),
        ],
    },
)

#setting the inputs:targetprim key to the robot articulation root
#located on the Raw USD Properties of the ROS 1 Publish Joint State
    property panel
set_target_prims(primPath="/World/BackWheelPubSub/PublishJointState",
targetPrimPaths=["/World/battle_ram/Battle_ram/body"])
```

Based on the script, there are two nodes used for running purposes, which are OnPlaybackTick and ReadSimTime. These two nodes does not need any configuration, just add them and connect its output to the corresponding input nodes. The nodes for publishing joint states is called ROS 1 Publish Joint State node and for subscribing to a joint command is called ROS 1 Subscribe Joint State node. These two nodes needs a short configuration, which are setting the articulation root for the ROS 1 Publish Joint State node and the topic name for ROS 1 Subscribe Joint State node. To set up the ROS 1 Publish Joint State node, just set the inputs:targetPrim key of the node property to the articulation root of the robot. To set up the ROS 1 Subscribe Joint State node, just set the inputs:topicName key to the topic name that wants to be subsribed. The two ROS nodes in Isaac Sim is only to handle the messages that is published and subscribed. However, to drive the robot, the actual node that does this is the articulation controller node. To set this node, just set the inputs:usePath key to True and inputs:robotPath key to the robot articulation root.
To test with the publisher and subscriber node, this documentation also provides the python script as well as the .usd file that can be downloaded from the gitlab repository

`https://github.com/anakagungkrisna/isaac-sim-documentation.git`

The script for creating the isaac sim ROS nodes is called `battle_ram_ros_node.py` and the script for publishing the joint command to the isaac sim subscriber node is called `battleram_joint_command.py`. Important to know that for `battle_ram_ros_node.py` must be executed by using the script editor (see section 7.1) while `battleram_joint_command.py` can be run using the local pc python. So for this tutorial, the steps are:

1. Launch Isaac sim, then create a ground plane (Create > Physics > Ground plane), then insert by drag and drop the `battle_ram.usd` to Isaac Sim from the content window.

2. Copy and paste the script of `battle_ram_ros_node.py` to the script editor of Isaac Sim and press ctrl + enter

3. Start the Isaac Sim simulation, then open a new terminal, run `roscore`.

4. Open a new terminal, then run the `battle_ram_joint_command.py` using the local pc python

By then, you should see the battle ram moves its wheels and cone. Use the command `rostopic list` to see the available topics and `rostopic echo "/joint_states_battleram"` to see the message being published and subscribed. The topics that should be available are:

```
/joint_command_battleram
/joint_states_battleram
/rosout
/rosout_agg
```

If there are topics that is missing, please double check the `ArticulationController.inputs:robotP` key and the `targetPrimPaths` param in `set_target_prims` function matches the articulation root of the robot. Sometimes the Isaac sim itself does not functions well. In this case, restarting Isaac sim might work.
The python script above can also be extended for multiple robots as well as a standalone project in Isaac sim (in which user can just run everything, including Isaac sim, through a python script). Below is the python script which can be executed using the Isaac sim Python located in the root folder of Isaac sim that starts Isaac sim and create ROS utilities for Isaac sim. You can get the the code below from the github repository called `standalone_multi_battleram_ros.py` and for the ROS publisher python code called `battleram_joint_command.py`

```python
# launch Isaac Sim before any other imports
# must include these default first two lines in any standalone
# application
from omni.isaac.kit import SimulationApp
simulation_app = SimulationApp({"headless": False})

import omni
from pxr import Gf, Sdf, UsdPhysics
from pxr import PhysxSchema
from pxr import PhysicsSchemaTools
import numpy as np

# libraries used for deploying multiple agents with ros nodes and
# creating the world
from omni.isaac.core.utils.stage import add_reference_to_stage
from omni.isaac.core.world import World
from omni.isaac.core.utils.extensions import enable_extension
import omni.graph.core as og
from omni.isaac.core_nodes.scripts.utils import set_target_prims
from omni.isaac.core.utils.prims import define_prim, get_prim_at_path
from omni.isaac.core.utils.nucleus import get_assets_root_path


# enable ROS bridge extension
enable_extension("omni.isaac.ros_bridge")
simulation_app.update()

```

```
# check if rosmaster node is running
# this is to prevent this sample from waiting indefinetly if roscore
# is not running
# can be removed in regular usage
import rosgraph
if not rosgraph.is_master_online():
    carb.log_error("Please run roscore before executing this script")
    simulation_app.close()
    exit()

if World.instance():
    print("World exist")
    World.instance().clear_instance()
world = World(stage_units_in_meters=1.0)

assets_root_path = get_assets_root_path()
if assets_root_path is None:
    carb.log_error("Could not find Isaac Sim assets folder")

#using warehouse environments
prim = get_prim_at_path("/World/Warehouse")
if not prim.IsValid():
    prim = define_prim("/World/Warehouse", "Xform")
    asset_path = assets_root_path +
        "/Isaac/Environments/Simple_Warehouse/warehouse.usd"
    prim.GetReferences().AddReference(asset_path)

#add battle ram articulation
usd_path = "/media/storage/Isaac/soankusu/battle_ram.usd"
num_of_agents = 3
for i in range(num_of_agents):
    add_reference_to_stage(usd_path=usd_path,
        prim_path="/World/battleram_" + str(i+1))

world.reset()
print("finished spawning robots")

for i in range(num_of_agents):
    og.Controller.edit(
        {
            "graph_path": "/World/battleramnode_" + str(i+1),
            "evaluator_name": "execution",
        },

        {
            og.Controller.Keys.CREATE_NODES: [
                # (name, node:type) -> you can search for node:type
                # in the raw USD properties of the prim
                # ("OnPlaybackTick",
                # "omni.graph.action.OnPlaybackTick"),
                ("OnTick", "omni.graph.action.OnTick"),
                ("ReadSimTime",
                "omni.isaac.core_nodes.IsaacReadSimulationTime"),
                ("publishClock",
                "omni.isaac.ros_bridge.ROS1PublishClock"),

                ("ArticulationController",
                "omni.isaac.core_nodes.IsaacArticulationController"),
                ("PublishJointState",
                "omni.isaac.ros_bridge.ROS1PublishJointState"),
                ("SubscribeJointState",
                "omni.isaac.ros_bridge.ROS1SubscribeJointState"),
            ],
            og.Controller.Keys.CONNECT: [
                ("OnTick.outputs:tick",
                "PublishJointState.inputs:execIn"),
                ("OnTick.outputs:tick",
                "SubscribeJointState.inputs:execIn"),
                ("OnTick.outputs:tick",
                "ArticulationController.inputs:execIn"),

                ("OnTick.outputs:tick","publishClock.inputs:execIn"),
                ("ReadSimTime.outputs:simulationTime",
                "PublishJointState.inputs:timeStamp"),
                ("ReadSimTime.outputs:simulationTime",
                "publishClock.inputs:timeStamp"),

                ("SubscribeJointState.outputs:jointNames",
                "ArticulationController.inputs:jointNames"),
                ("SubscribeJointState.outputs:positionCommand",
                "ArticulationController.inputs:positionCommand"),
                ("SubscribeJointState.outputs:velocityCommand",
                "ArticulationController.inputs:velocityCommand"),
                ("SubscribeJointState.outputs:effortCommand",
                "ArticulationController.inputs:effortCommand"),
            ],
            og.Controller.Keys.SET_VALUES: [
                ("ArticulationController.inputs:usePath", True),
                ("ArticulationController.inputs:robotPath",
                "/World/battleram_" + str(i+1) + "/Battle_ram/body"),
                ("PublishJointState.inputs:topicName",
                "joint_states_battleram_" + str(i+1)),
                ("SubscribeJointState.inputs:topicName",
                "joint_command_battleram"),
            ],
        },
    )

    # setting the robot target prim to publish JointState node
    # cant you set this directly to the node??? no...
    # this method is used so that we can use the
    # "Add Target(s) " button in the property panel
    set_target_prims(primPath=
    "/World/battleramnode_"+ str(i+1)+"/PublishJointState",
    targetPrimPaths=
    ["/World/battleram_"+str(i+1)+"/Battle_ram/body"])

    print("created node")

print("finished creating ros utilities")
```

```python



if __name__ == "__main__":

    while simulation_app.is_running():
        world.step(render=True)

    rospy.signal_shutdown("completed the program")
    simulation_app.close()
```

Isaac sim also provides a standalone application using the a1 quadruped with ROS. One can execute the script using the command below inside the Isaac sim root directory:

```
./isaac-sim.sh standalone_examples/api/omni.isaac.quadruped
/a1_direct_ros1_standalone.py
```

Remember to first run the command `roscore` to start the ROS master. To move the joints of the robot, run the python script called `a1_joint_command_publisher.py` from the github repository using the local pc python.

9 Parallel Manipulation of Multiple Robots in Isaac Sim [3]

This chapter will explain how to create a world with multiple robots that can be controlled individually and can sends various sensor data. This example uses the `battle_ram.usd` for the robot and can be found from the gitlab repository

```
https://github.com/anakagungkrisna/isaac-sim-documentation.git
```

To begin with, first we need to create a python script. This python script will be executed with the python.sh shell command located on the root folder of isaac sim. For all standalone application of Isaac sim, the first most important two line is

```
from omni.isaac.kit import SimulationApp
simulation_app = SimulationApp({"headless": False})
```

`SimulationApp` is used to open Isaac sim from python script. One argument that needs to be included is ”headless”, which means disabling visualization. If we want to see the agents learning in Isaac sim on real-time, then we need to set `headless = False`. If the case we need more computational speed, we can also make `headless = True` if we are sure that the algorithm used for learning is executed correctly. It is also important to note that `SimulationApp` import must be the first one to be imported before any other imports. Otherwise the standalone application would not work.
Parallel manipulation of robots is done using the Core extension available from the Main Extensions of Isaac sim [2]. This extension provide so called view classes, which are `ArticulationView`, `RigidPrimView`, and `XFormPrimView`. They provide the high-level functionalities to manipulate in parallel sets of articulation, rigid prims, and Xforms respectively. On this example, since on the previous chapter we have deal with articulation, `ArticulationView` class will be used to manipulate articulations in parallel. For this, one must have a robot model with articulation root (see chapter 6 in Robot Creation for more details).
To use `ArticulationView`, we must first create a world for the robots to spawn, and add the robots to the world. World creation is done by using the `World` class from `omni.isaac.core.world` module. Spawning robots is done by using `add_reference_to_stage` function from `omni.isaac.core.utils.stage` module. It is important to note that the name of each robots must unique. The code snippets below shows on how to do the two things mentioned earlier. The `usd_path` and `prim_path` parameter of the `add_reference_to_stage` function points to the .usd file located on local pc and the prim path that the user desired in the stage respectively.

```
from omni.isaac.core.utils.stage import add_reference_to_stage
from omni.isaac.core.articulations import ArticulationView
from omni.isaac.core.world import World

world = World()
world.scene.add_default_ground_plane()
```

```
print("world created!")

#add battle ram articulation (use complete path)
usd_path = "/media/data/ros_user/Desktop/battle_ram.usd"
num_of_agents = 5
for i in range(num_of_agents):
    add_reference_to_stage(usd_path=usd_path,
        prim_path="/World/battleram_" + str(i+1))
print("Agents created!")
```

After the world has been created and the robots has been spawned, the next thing is to create an `ArticulationView` for all robots. This `ArticulationView` will have access to all the actuators of all robots, in which one can just define a matrix consist of goal positions, goal velocities or goal effort (torque) for each of the robot to follow. The code snippets to define an articulation view for multiple robots is as follows. The `prim_paths_expr` parameter of the `ArticulationView` class points to the articulation root of the robot.

```
from omni.isaac.core.articulations import ArticulationView

articulation_view = ArticulationView(prim_paths_expr=
    "/World/battleram_.*"+ "/Battle_ram/body",
    name="battleram_view")
print("articulationview for the agents defined!")

world.scene.add(articulation_view)
```

If there is an error when defining an ArticulationView, stating ”a prim matching the expression needs to created before wrapping it as view”, then it means it cannot find the prim path that is defined in the `prim_paths_expr` argument. Make sure that the `usd_path` argument in the `add_reference_to_stage` functions points to the correct .usd file. After the `ArticulationView` object is created, by then we can start to control the robots. For the ease of doing that, there is a special class called `ArticulationActions` from `omni.isaac.core.utils.types` module which provide the three inputs that the robots can take, which are goal positions, velocities, or efforts. Once the actions is defined, we can use the function `ArticulationView.apply_action(actions)` to send the commands to the robots. There is also a function to get the actions being applied, the state of the joints, and the world pose of the robots. For more functions that `ArticulationView` has to offer, visit the link below.

```
https://docs.omniverse.nvidia.com/py/isaacsim/source/extensions/
omni.isaac.core/docs/index.html#module-omni.isaac.core.articulations
```

Once everything has been created and added to the world, we need to first reset the world using the function `world.reset()`. This is important because View classes are internally initialized when they are added to the scene and the world is reset. Then to run our simulation continuously, use the while loop in python (or it can be also have a certain parameter that could stop the simulation once the parameter has reached a certain value of increments) and use the function `world.step(render=True)` to execute one physics step and one rendering step of the simulation. In the end, do not forget to close the Isaac sim simulation using the function `simulation_app.close()` to finish the program cleanly. Below is the complete python code that implements all the methods above. One can get it from the github link of this documentation called `multi_agent_training_online.py`. To run the python code, simply go to the Isaac sim root folder, and use the command `./python.sh` followed with the complete path to `multi_agent_training_online.py`.

```python
# This standalone application is based on:
# https://docs.omniverse.nvidia.com/py/isaacsim/
# source/extensions/omni.isaac.core/docs/
# https://docs.omniverse.nvidia.com/app_isaacsim/
# app_isaacsim/tutorial_core_hello_world.html
# https://docs.omniverse.nvidia.com/app_isaacsim/
# app_isaacsim/tutorial_gym_new_rl_example.html
# https://docs.omniverse.nvidia.com/app_isaacsim/
# app_isaacsim/ext_omni_isaac_core.html#isaac-core

# launch Isaac Sim before any other imports
# must include these default first two lines in any
# standalone application
from omni.isaac.kit import SimulationApp
simulation_app = SimulationApp({"headless": False})

import omni
from pxr import Gf, Sdf, UsdPhysics
from pxr import PhysxSchema
from pxr import PhysicsSchemaTools
import numpy as np
import time

# libraries used for deplying multiple agents
from omni.isaac.core.utils.stage import add_reference_to_stage
from omni.isaac.core.articulations import ArticulationView
from omni.isaac.core.world import World
import asyncio
# For one agent
from omni.isaac.core.utils.types import ArticulationAction
# For multi agent
from omni.isaac.core.utils.types import ArticulationActions

# multi tasking as in arduino millis
from time import time
previousTime = int(time() * 1000) # in miliseconds
intervalTime1 = 2500 # 5 seconds
state = False
print("Time in milliseconds since epoch",previousTime)

class Agents:
    # for more ArticulationView functions to define, visit
    # https://docs.omniverse.nvidia.com/py/isaacsim/
    # source/extensions/omni.isaac.core/docs/index.html
    def __init__(self, agents_num):
        #add battle ram articulation
        self.usd_path = "/media/data/ros_user/Desktop/battle_ram.usd"
        num_of_agents = agents_num
        for i in range(num_of_agents):
            add_reference_to_stage(usd_path=self.usd_path,
                prim_path="/World/battleram_" + str(i+1))
        print("Agents created!")

        #batch process articulation via ArticulationView
        self.articulation_view = ArticulationView(prim_paths_expr=
            "/World/battleram_.*"+ "/Battle_ram/body",
            name="battleram_view")
        print("articulationview for the agents defined!")

    def set_world_poses(self, positions):
        self.articulation_view.set_world_poses(positions)
        #print("world poses applied")

    def send_actions(self, actions):
        self.articulation_view.apply_action(actions)

    def get_actions(self):
        applied_actions=self.articulation_view.get_applied_actions()
        print("applied actions are: ",
        applied_actions.joint_velocities)

    def get_jacobians(self):
        jacobian_matrices = self.articulation_view.get_jacobians()
        print(jacobian_matrices)

    def get_joints_state(self):
        # current joint positions and velocities.
        joints_state = self.articulation_view.get_joints_state()
        positions = joints_state.positions
        velocities = joints_state.velocities
        efforts = joints_state.efforts
        print("positions: ", positions)
        print("velocities: ", velocities)
        print("efforts: ", efforts)

    def get_velocities(self):
        # linear and angular velocities of the prims
        # in the view concatenated.
        # shape is (M, 6).
        velocities = self.articulation_view.get_velocities()
        print("linear & angular velocities: ", velocities)

    def get_local_poses(self):
        # Gets prim poses in the view
        # with respect to the local frame (the p r i m s parent frame)
.
        local_poses = self.articulation_view.get_local_poses()
        print("local poses: ", local_poses)

    def get_world_poses(self):
        # Gets the poses of the prims in the view
        # with respect to the w o r l d s frame.
        world_poses = self.articulation_view.get_world_poses()
        print("world poses: ", world_poses)

world = World()
world.scene.add_default_ground_plane()

print("creating agents")
agents = Agents(2)
world.scene.add(agents.articulation_view)

world.reset()

#set root body pose of the agents based on world frame
new_positions = np.array([[-50.0, 50.0, 0.0],
    [50.0, 10.0, 0.0]]) # for 2 agents
agents.set_world_poses(new_positions)

#the main loop for training
current_episode = 0
total_episodes = 2000
while current_episode < total_episodes:
    currentTime = int(time() * 1000)
    if(currentTime - previousTime > intervalTime1):
        previousTime = currentTime
        print("passing 2.5 seconds")
        if(state == True):
            # create articulation actionS to send
            # command to all articulation root
            actions= ArticulationActions()
            actions.joint_velocities = [[50, 50, 50, 50, 50],
                [50, 50, 50, 50, 50]]
            agents.send_actions(actions)
            state = False
        else:
            actions= ArticulationActions()
            actions.joint_velocities = [[-50, -50, -50, -50, -50],
                [-50, -50, -50, -50, -50]]
            agents.send_actions(actions)
            state = True
        print(state)
        print("for all agents: ")
        agents.get_actions()
        agents.get_joints_state()
        agents.get_velocities()
        agents.get_local_poses()
        agents.get_world_poses()

    #world.pause()

    # we have control over stepping physics and rendering
    # in this workflow
    # things run in sync
    # execute one physics step and one rendering step
    world.step(render=True)
    current_episode += 1

simulation_app.close() # close Isaac Sim
```

The code above can also be extended for the available robot models in Isaac sim. The python code below is the extended version of the code above, in which it access the .usd file that contains the robot models in Isaac sim. One can get it from the github link of this documentation called `isaac_sim_cloner_example.py` and to run the script, use the Isaac sim Python located in the root directory of Isaac sim followed with the complete path to `isaac_sim_cloner_example.py`.

```
from omni.isaac.kit import SimulationApp
simulation_app = SimulationApp({"headless": False})

# import Cloner interface
from omni.isaac.cloner import Cloner
# import GridCloner interface
from omni.isaac.cloner import GridCloner
from omni.isaac.core.utils.stage import get_current_stage
from pxr import UsdGeom

from omni.isaac.core.world import World
from omni.isaac.core.utils.stage import add_reference_to_stage
from omni.isaac.core.utils.nucleus import get_assets_root_path
from omni.isaac.core.articulations import ArticulationView
# for single agent
from omni.isaac.core.utils.types import ArticulationAction
# for multi agent
from omni.isaac.core.utils.types import ArticulationActions
import numpy as np
import asyncio

# for unitree a1
from omni.isaac.quadruped.robots import Unitree

# multi tasking as in arduino millis
from time import time
previousTime = int(time() * 1000) # in miliseconds
previousTime2 = int(time() * 1000)
intervalTime1 = 500 # 5 seconds
state = False
print("Time in milliseconds since epoch",previousTime)

class Agents:
    # for more ArticulationView functions to define, visit
    # https://docs.omniverse.nvidia.com/py/isaacsim/
    # source/extensions/omni.isaac.core/docs/index.html
    def __init__(self, agents_num, robot_name):
        num_of_agents = agents_num
        assets_root_path = get_assets_root_path()
        asset_path = ""
        self.battle_ram = False
        # models which has controller defined does not work?
        if(robot_name == "a1"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Unitree/a1.usd"
        elif(robot_name == "go1"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Unitree/go1.usd"
        elif(robot_name == "anymal_instanceable"):
            asset_path = assets_root_path +
                "/Isaac/Robots/ANYbotics/anymal_instanceable.usd"
        elif(robot_name == "anymal"):
            asset_path = assets_root_path +
                "/Isaac/Robots/ANYbotics/anymal_c.usd"
        elif(robot_name == "carter_v1"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Carter/carter_v1.usd"
        elif(robot_name == "carter_v2"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Carter/carter_v2.usd"
        elif(robot_name == "aws_robomaker_jetbot"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Jetbot/aws_robomaker_jetbot.usd"
        elif(robot_name == "jetbot"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Jetbot/jetbot.usd"
        elif(robot_name == "transporter_sensors"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Transporter/transporter_sensors.usd"
        elif(robot_name == "kaya"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Kaya/kaya.usd"
        elif(robot_name == "o3dyn"):
            asset_path = assets_root_path +
                "/Isaac/Robots/O3dyn/o3dyn.usd"
        elif(robot_name == "cobotta_pro_900"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Denso/cobotta_pro_900.usd"
        elif(robot_name == "franka"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Franka/franka.usd"
        elif(robot_name == "franka_alt_fingers"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Franka/franka_alt_fingers.usd"
        elif(robot_name == "allegro_hand"):
            asset_path = assets_root_path +
                "/Isaac/Robots/AllegroHand/allegro_hand.usd"
        elif(robot_name == "shadow_hand"):
            asset_path = assets_root_path +
                "/Isaac/Robots/ShadowHand/shadow_hand.usd"
        elif(robot_name == "cf2x"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Crazyflie/cf2x.usd"
        elif(robot_name == "quadcopter"):
            asset_path = assets_root_path +
                "/Isaac/Robots/Quadcopter/quadcopter.usd"
        else:
            asset_path =
                "/media/storage/Isaac/soankusu/battle_ram.usd"
            self.battle_ram = True

        add_reference_to_stage(usd_path=asset_path,
            prim_path="/World/A1_0")
        print("One agents created!")

        # create a cloner object
        cloner = GridCloner(spacing=1)
        print("Cloner created!")

        # generate num_of_agents paths that begin with "/World/A1"
        # - path will be appended with _{index}
        target_paths = cloner.generate_paths("/World/A1", agents_num)
        # setting the base path for the cloner to clone
        cloner.clone(source_prim_path="/World/A1_0",
            prim_paths=target_paths)
        print("cloned agents created!")

        #batch process articulation via ArticulationView
        if(self.battle_ram == True):
            self.articulation_view =
                ArticulationView(prim_paths_expr=
                "/World/A1_.*"+ "/Battle_ram/body",
                name="battleram_view")
        else:
            self.articulation_view =
                ArticulationView(prim_paths_expr=
                "/World/A1_.*", name="A1_view")
                #"/World/A1_[0-" + str(num_of_agents-1)+ "]", name="A1_view")
        print("articulationview for the agents defined!")

        self.num_dof = self.articulation_view.num_dof
        self.prim_paths = self.articulation_view.prim_paths

    def set_world_poses(self, positions):
        print("setting world poses")
        self.articulation_view.set_world_poses(positions)
        #print("world poses applied")

    def send_actions(self, actions):
        self.articulation_view.apply_action(actions)

    def get_actions(self):
        applied_actions=self.articulation_view.get_applied_actions()
        print("applied actions are: ", applied_actions.joint_velocities)

    def get_jacobians(self):
        jacobian_matrices = self.articulation_view.get_jacobians()
        print(jacobian_matrices)

    def get_joints_state(self):
        # current joint positions and velocities.
        joints_state = self.articulation_view.get_joints_state()
        positions = joints_state.positions
        velocities = joints_state.velocities
        efforts = joints_state.efforts
        print("positions: ", positions)
        print("velocities: ", velocities)
        print("efforts: ", efforts)

    def get_velocities(self):
        # linear and angular velocities of the prims in the view concatenated.
        # shape is (M, 6).
        velocities = self.articulation_view.get_velocities()
        print("linear & angular velocities: ", velocities)

    def get_local_poses(self):
        # Gets prim poses in the view
        # with respect to the local frame (the p r i m s parent frame).
        local_poses = self.articulation_view.get_local_poses()
        print("local poses: ", local_poses)

    def get_world_poses(self):
        # Gets the poses of the prims in the view
        # with respect to the w o r l d s frame.
        world_poses = self.articulation_view.get_world_poses()
        print("world poses: ", world_poses)

world = World()
num_agents = 15
print("creating agents")
agents = Agents(num_agents, "a1")
world.scene.add_default_ground_plane()

# View classes are internally initialized when
# they are added to the scene and the world is reset
world.scene.add(agents.articulation_view)
print("resetting world")
world.reset()
print("finish resetting world")

current_episode = 0
total_episodes = 5000
#while current_episode < total_episodes:
while True:
    currentTime = int(time() * 1000)
    if(currentTime - previousTime > intervalTime1):
        previousTime = currentTime
        print("passing 2.5 seconds")

        if(state == True):
            #create articulation actionS to send command to all articulation root
            actions= ArticulationActions()
```

```
                actions.joint_efforts = np.random.randint(-50,50,
                    (num_agents,agents.articulation_view.num_dof))
                agents.send_actions(actions)
                state = False
            else:
                actions= ArticulationActions()
                actions.joint_efforts = np.random.randint(-50,50,
                    (num_agents,agents.articulation_view.num_dof))
                agents.send_actions(actions)
                state = True
            print(state)
            print("for all agents: ")
            agents.get_actions()
            agents.get_joints_state()
            agents.get_velocities()
            agents.get_local_poses()
            agents.get_world_poses()
            #print(agents.articulation_view.num_dof)
            #print(agents.articulation_view.prim_paths)

    # execute one physics step and one rendering step
    world.step(render=True)
    current_episode += 1

# Dont forget to properly close the simulation
simulation_app.close() # close Isaac Sim
```

The screenshots of the Isaac sim for the executed scripts above for the case of multi battleram robot and multi a1 robot can be seen on Figure 5 and Figure 6.
There are many robots available in Isaac sim to work on. These robots are all included inside Isaac sim and can be accessed from Isaac Examples menu of Isaac sim's top menu bar. For a complete list of available robots and environments, visit the link below.

    https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim/reference_assets.html

All of the robots can be used in Isaac sim and can be trained using reinforcement learning method. Isaac sim provide a proposed method on creating a task for reinforcement learning and there are also some examples on training and inferencing a reinforcement learning model. For more information on how to follow the steps provided by Isaac sim, go to the Isaac Gym Tutorials section on Isaac sim documentation website from the link below.

    https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim/tutorial_gym_isaac_gym.html

The github repository that provides all the python script for training and inferencing reinforcement learning model can be obtained below.

    https://github.com/NVIDIA-Omniverse/OmniIsaacGymEnvs

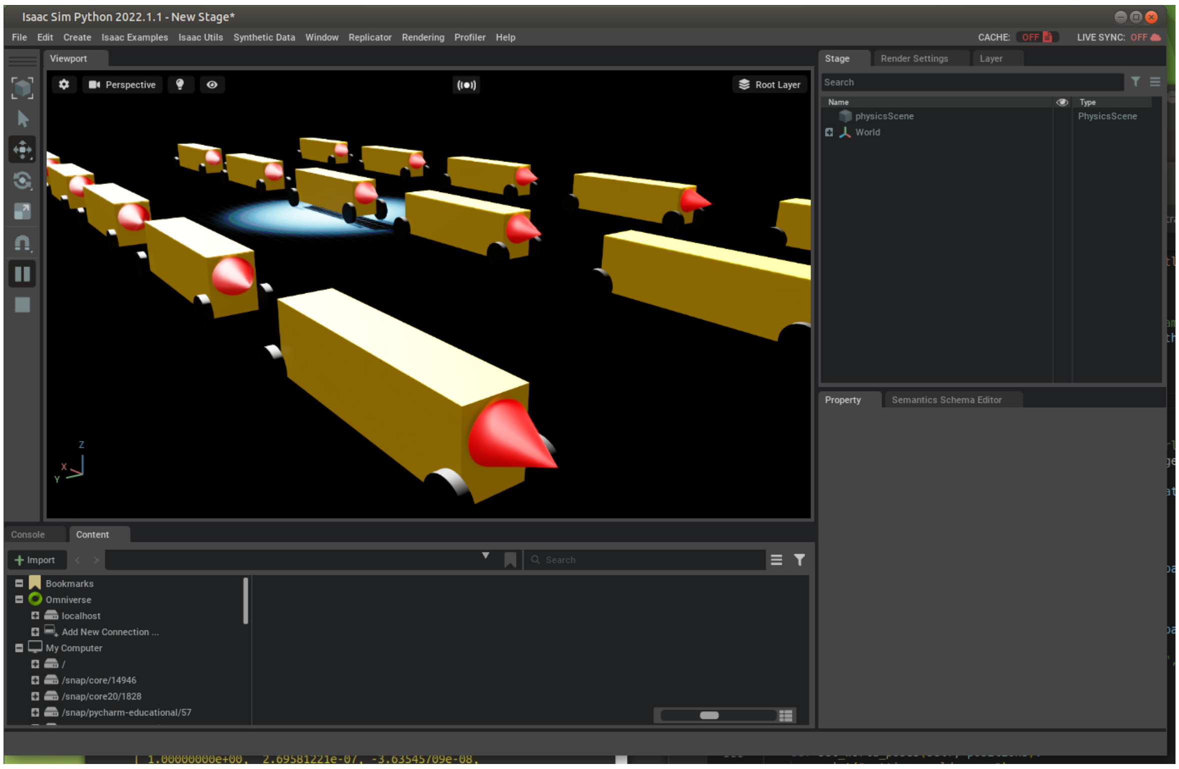

Figure 5: Multi battleram robot

Small note when cloning the repository, the example code will not work out of the box, so there must be some changes inside the code in order for the examples can be executed properly. Here are the steps on how to make it work:

1. Open a new terminal and go to the Isaac sim root folder.
2. Execute the command

   ```
   git clone https://github.com/NVIDIA-Omniverse/OmniIsaacGymEnvs.git
   ```

3. Go to the directory called `OmniIsaacGymEnvs`
4. run the command

   ```
   ../python.sh -m pip install -e .
   ```

5. After the build is complete, there are some python code that needs to be commented out. The location are in:

   ```
   /media/data/isaac/.local/share/ov/pkg/isaac_sim-2022.1.1/OmniIsaacGymEnvs
   /omniisaacgymenvs/scripts/rlgames_train.py
   ```

   ```
   /media/data/isaac/.local/share/ov/pkg/isaac_sim-2022.1.1/OmniIsaacGymEnvs
   /omniisaacgymenvs/tasks/base/rl_task.py
   ```

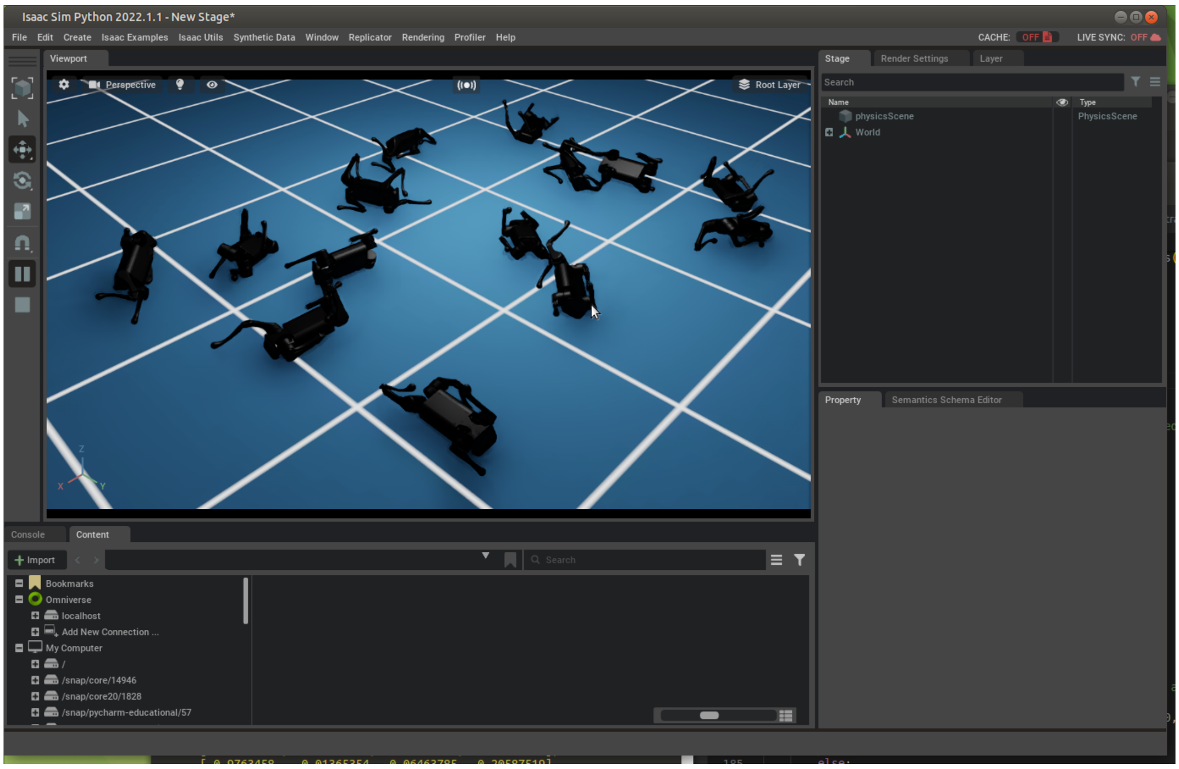

Figure 6: Multi a1 robot with random action

If we run the command

```
../python.sh omniisaacgymenvs/scripts/rlgames_train.py task=Cartpole
```

Then it will give an error as shown

```
TypeError: __init__() got an unexpected keyword argument 'enable_livestream'
```

Just comment out the argument 'enable_livestream' located on .py file mentioned . Same thing goes for the other arguments that will later mentioned in the error. Do this for all the error that might popped out.

By then, everything should work for the example code provided by Isaac sim.

References

[1] NVIDIA. Develop on nvidia omniverse. 3

[2] NVIDIA. Isaac sim extentions list. 23

[3] NVIDIA. Isaac sim main extensions core. 2, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34

[4] NVIDIA. Isaac sim ros tutorials. 2, 16, 17, 18, 19, 20, 21, 22

[5] NVIDIA. Isaan sim python interavtive scripting. 2, 14, 15

[6] NVIDIA. Nvidia omniverse. 3

[7] NVIDIA. Omnigraph. 2, 9

[8] NVIDIA. Omnigraph: Python scripting. 2, 10

[9] NVIDIA. Universal scene description (usd). 4

[10] Wikipedia. Agnostic (data). 4

[11] Wikipedia. Graphics pipeline. 3

[12] Wikipedia. Metaverse. 3