ROS-X-Habitat: bridging the ROS ecosystem with embodied AI

by

Haoyu Yang

B.Sc., Computer Science, The University of British Columbia, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2022

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

ROS-X-Habitat: bridging the ROS ecosystem with embodied AI

submitted by **Haoyu Yang** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Ian M. Mitchell, Professor, Department of Computer Science, UBC
Supervisor

Dinesh K. Pai, Professor, Department of Computer Science, UBC
Supervisory Committee Member

Abstract

We introduce ROS-X-Habitat, a software interface that bridges the AI Habitat platform for embodied reinforcement learning agents with other robotics resources via ROS. This interface not only offers standardized communication protocols between embodied agents and simulators, but also enables physics-based simulation. With this interface, roboticists are able to train their own Habitat RL agents in another simulation environment or to develop their own robotic algorithms inside Habitat Sim. Through in silico experiments, we demonstrate that ROS-X-Habitat has minimal impact on the navigation performance and simulation speed of Habitat agents; that a standard set of ROS mapping, planning and navigation tools can run in the Habitat simulator, and that a Habitat agent can run in the standard ROS simulator Gazebo. Furthermore, to show how ROS-X-Habitat can be used in data collection and RL training, we present the training and evaluation of an agent we train to perform a multiple point goal navigation task we define.

Lay Summary

Simulation software plays an essential part in modern artificial intelligence and robot development and testing. In the past decade, powerful simulation software and tools were built to offers high speed and realism to its users. However, not many of these simulators have the compatibility to be used along with other software platforms or middleware. Habitat AI for example, consists of a high-speed and photo-realistic simulation environment (Habitat-sim) for reinforcement learning research and a modular high-level library to train embodied AI agents (Habitat-lab). In this thesis, we present `ros_x_habitat`, a software interface that bridges the ROS ecosystem including ROS-compatible simulators with Habitat AI to allow AI researchers and roboticists to take advantage of both platforms at the same time. We demonstrate the interface's features and conduct in-depth evaluations on the performance impact of using it.

Preface

All of the work presented henceforth was conducted in the Collaborative Robotics Laboratory at the University of British Columbia under the supervision of Dr. Ian M. Mitchell. I collaborated closely with undergraduate research assistant Guanxiong Chen as well as my supervisor Dr. Ian M. Mitchell in the developement and presentation of this work. Working closely with myself, Guanxiong contributed to the implementation and the execution of the research. Specifically, Guanxiong executed the experiments in Chapter 4 and implemented the visualization of experiment results.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I want to express my sincere gratitude to my supervisor, Dr. Ian M. Mitchell for his continuous guidance and support during my time at UBC. Next, I would give special thanks to my family for supporting my decision to pursue my interests in doing research and to Gareth Brown, the CEO of Clir Renewables, for keeping me employed part-time so that I have extra income during my studies.

I would also like to thank my second reader, Dr. Dinesh K. Pai, for taking the time to read my work and provide insightful comments. Finally, I would like to thank my friends for their constant encouragement and for making this time much more pleasant. To my friends from the program: Bicheng, Devon, Guanxiong, Jocelyn, and Yuhao, thank you for the wonderful time we spent together, I will always keep our friendship in my heart. To my friends outside of the program: Dylan, Frank, Ivan, Wen, and Yiming, thank you for bringing me so much joy into my life, I would not make it if you were not there for me during my tough times.

Acknowledgments

In loving memory of my grandmother Chenglan Fu, I lost you during the pandemic and I could not make it back for your funeral. Deep in my heart you will always stay, loved and remembered every day.

Chapter 1

Introduction

Since the earliest days of robotics, researchers have sought to build embodied agents to perform a variety of jobs, such as assistive tasks in factories [29] or wildfire surveillance [16]. Following tremendous advancements in deep learning and convolutional neural networks in the past decade, researchers have been able to develop reinforcement learning (RL)-based embodied agents that interact with the real world on the basis of sensory observations. Software platforms such as OpenAI Gym [7], Unity ML-Agents Toolkit [17], and AI Habitat [31] have emerged to address the community's need for training and evaluating RL-based embodied agents end-to-end. Our research group was particularly intrigued by the AI Habitat platform, which offers a high-performance, photorealistic simulator, access to a sizeable library of visually-rich scanned 3D environments, and a modular software design.

However, even though these platforms allow roboticists to reuse existing RL algorithms and train agents in simulators with ease, there is a critical step to using them for embodied agents which is only partially addressed: Connecting the trained agent with a real robot. Ideally, after training an RL agent in simulation one would like to take advantage of the extensive set of tools and knowledge from the robotics community to make it easy to embody that agent.

1.1 The ROS Ecosystem

One particularly popular tool from the robotics community is ROS, a robotics-focused middleware platform with extensive support for classical robotic mapping, planning and control algorithms ([2, 3]) as well as drivers for a wide variety of compute, sensing and actuation hardware. But ROS support for directly training an RL agent is limited, and Gazebo—the standard simulation environment used for ROS systems—cannot match the level of photorealism or simulation speed of tools specifically designed to train large-scale RL agents [23].

Meanwhile, robotics researchers are still developing classical embodied agents that follow the "sense-plan-act" approach [6]. Compared with learning-based agents, classical agents tend to be less data and compute intensive, and many developers build and test classical agents for indoor tasks within the ROS ecosystem. For navigation tasks, packages such as `hector_mapping` [20] and `move_base` [3] allow users to easily map an environment and set up a planner. Simulation tools such as Gazebo that work under ROS offer simulation dynamics and collision detection that have a moderate degree of physical realism. The large number of hardware drivers from ROS packages allow users to easily test an agent in the real world. But ROS also has its own shortcomings: 1) little support for building learning-based agents, and 2) limited choices of simulation tools.

1.2 AI Habitat

Amongst many Embodied AI research platforms, Facebook's AI Habitat stands out for its high-performance, photorealism, and compatibility with visually-rich scanned 3D environments. It consists of two main components: Habitat-sim and Habitat-lab. Habitat-sim is a high-performance 3D simulator with support for physics, high-fidelity 3D datasets, configurable sensors. Habitat-lab is a modular high-level library for end-to-end development in embodied AI, it allows researchers to define embodied AI tasks, to configure embodied agents and sensors, and to train embodies agents.

However, AI Habitat framework does not provide any support or interface to ROS. Moreover, AI Habitat does not provide support for development of classical embodied agents or robot control algorithms, limiting its potential to the robotics

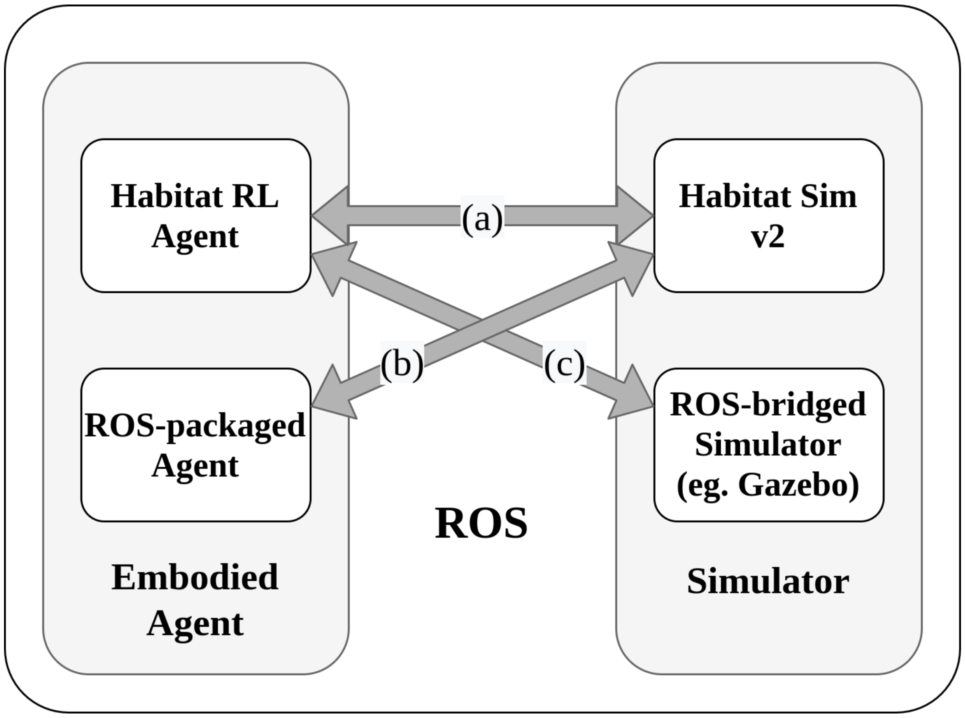


Figure 1.1: High-level overview of `ros_x_habitat`'s architecture. The arrows represent communications between an agent and a simulator. Operating mode (a), (b) and (c) are detailed in Section 3.3

community.

1.3 Contribution

In order to take advantage of the strengths and overcome the weaknesses of these two independent sets of tools, we therefore present ROS-X-Habitat (**ros_x_habitat**), an interface that bridges the AI Habitat training platform with the ROS ecosystem. Figure 1.1 shows a simplified view of the interface's architecture. The interface makes the following contributions to the robotics community:

- It allows AI Habitat's RL agents to be evaluated in ROS, so RL agent developers can take advantage of ROS' rich set of tools and community support, as well as Gazebo's ability to let users author 3D assets and customary testing scenes. Although not demonstrated here, this bridge to ROS dramatically shortens the path to embodying the Habitat agents in a physical robot.

- It allows classical embodied agents implemented in ROS packages (such as [2, 3]) to access the state-of-the-art simulator from AI Habitat (now with physics!). Thus researchers can evaluate the classical agents in more photorealistic environments (many of them scanned from the real world) with higher simulation speed.
- It allows researchers to easily collect, process, and annotate data from the supported datasets. With augmented data via this interface, further development of robotic agents, including training, testing, and evaluation, can proceed.

Although not use cases for the interface, we also demonstrate that:

- Connecting a Habitat agent to Habitat Sim through ROS does not change navigation performance and has modest effect on execution speed when compared to the standard direct connection in the AI Habitat codebase.
- For an Habitat agent trained in a simulation environment without physics, the performance difference in a test environment between simulations with and without physics is slight.

Chapter 2

Related Work

To help guide the objectives of our work, we review previous work in four relevant areas of research in this chapter: 1. embodied agents, 2. robotic middleware, 3. simulation software for robotics, and 4. datasets for either training and evaluation of learning-based embodied agents or for evaluating classical embodied agents in general. By carefully reviewing the existing work, we show the robotics community has need to

1. bridge learning-based embodied agents with Gazebo or other ROS-bridged simulators;
2. bridge classical ROS-based planners with state-of-the-art robotic simulators that offer a greater level of photorealism as well as speed;
3. evaluate embodied agents' performance in a continuous action space to leverage simulators' physics simulation capability.

2.1 Embodied Agents

In this work, we consider two categories of embodied agents commonly used to complete PointGoal navigation tasks: Classical robotics approaches and learning-based.

Most commonly deployed classical embodied agents do navigation in two phases: Construct a map of the environment using, for example, a SLAM-based algorithm (such as [5, 20, 22]), then use the map to plan out a path to the goal position (such as [2, 3]). While many packages following this approach are available in ROS, most recent photorealistic, physics-capable simulators (see Section 2.2) do not interface to ROS and thus evaluating classical agents in those simulators is difficult.

So-called "end-to-end" learning-based agents use a neural network to produce a sequence of actions directly from visual observations and/or localization data without relying on prior maps [12, 31, 36]. Another popular approach is to combine learning-based agents with classical mapping-then-planning [8, 10]. But none of these frameworks make direct connections with ROS.

Although not fundamental to their designs, a common distinction between classical and RL agents is that the former operate in a continuous action space while the latter are often trained to produce discrete actions. Continuous action spaces are more representative of how physical robots actuate [26] in the real world, but training for these spaces has higher computational cost [25]. As an example, Habitat's default PointGoal navigation agents have an action space consisting of four actions [31]: `move_forward`, `turn_left`, `turn_right`, and `stop`. To simulate these actions, Habitat Sim teleports the robot from one state/position to another without taking account of interactions between the agent and other objects at intermediate states. It is certainly possible to map from discrete to continuous actions, but it is not clear a priori that a composition of an agent trained to produce discrete actions with transformation of the actions from discrete to continuous will produce desirable outcomes.

2.2 Simulators

Simulators are powerful tools for early stage development, training and testing of embodied agents before deploying them in the real world. In the past, simulators were judged on the dimension of their environment (2D or 3D), realism of sensing and actuation, usability, and OS support [33], but the exploding interest in RL agents, and particularly in vision-based RL agents, has put a premium on simulation speed and photorealism.

Some popular examples of simulators include Gazebo [19], Unity [17], and Nvidia's Isaac Sim [27]. Gazebo has a large user community and lots of community-shared pre-built assets, but lacks photorealism and high simulation speed. Unity is a powerful game engine, and the Unity ML-Agents Toolkit available to researchers provides simulation environments suitable for embodied agents [17]. But the toolkit does not provide photorealistic simulation spaces by default and lacks compatibility with off-the-shelf 3D datasets such as Replica [34] or Matterport3D [9]. Isaac Sim shows great promise in terms of configurability and photorealism, but is not currently open-source or free of charge. The recently released Sapien [39] platform offers a photorealistic and physics-rich simulated environment, but currently provides limited support for tasks other than motion planning.

In this paper we explore the use of Habitat Sim v2 from the AI Habitat platform [35] for several reasons:

1. Faster simulation speed. This feature is particularly useful for RL agents, since agent performance may continue to improve even after many millions of training steps.

2. Photorealistic rendering of scanned spaces. Habitat Sim can render photorealistic scenes (including depth maps) from datasets such as Replica [34] and Matterport3D [9]. Training and testing embodied agents in photorealistic scenes can help to reduce the *sim2real* gap.

3. Simulation of many different tasks. We focus here on PointGoal navigation [31], but object picking has also been demonstrated [35].

2.3 Robotics Middleware

Robotics middleware is an abstraction layer that resides between the robotics software and the operating system (which is itself abstracting the underlying hardware). Middleware provides standardized APIs to sensors, actuators, and communication; design modularity; and portability [13].

There have been a variety of robotics middleware systems over the years (for example, see [21]) each with its own strengths and limitations. For example, while Microsoft RDS supports multiple programming languages, it runs only on the Windows OS[14]. OROCOS offers its own optimized run-time environment for real-time applications, but it does not have a graphical environment for drag-and-drop development nor a simulation environment [14].

ROS (Robot Operating System) is a popular robotics middleware introduced in 2007 [30]. Among its features:

1. it is free and open-source,
2. it promotes modular and robust designs by breaking implementations into communicating nodes and services that run independently, and
3. the huge user community has generated thousands of ready-to-use packages, including drivers for all common robotics hardware [15, 24, 30].

Given the popularity of ROS, it should come as no surprise that others have sought to build an interface between an embodied AI platform and the ROS ecosystem. Zamora et al. [40] used ROS as a bridge between OpenAI Gym and Gazebo, but did not consider other simulation environments which could provide more photorealism. PyRobot is a lightweight, high-level interface on top of ROS that provides a consistent set of hardware-independent mid-level APIs to control different robots [28], and the Habitat-PyRobot Bridge (HaPy) [18] further provides integration between Habitat agents and PyRobot. However, the HaPy-PyRobot combination suffers from four limitations:

1. tight coupling with the `LoCoBot` [31] asset;
2. tight coupling with Habitat Sim;
3. unable to support physics-based simulation from Habitat Sim v2;
4. no support for a direct connection between classical agents and Habitat Sim.

2.4 Datasets

Unlike Gazebo—in which users have to author their own 3D assets and scenes—Habitat Sim can ingest 3D scenes from off-the-shelf datasets. The Habitat framework's modular design [31, 35] offers easy access to many photorealistic 3D scene datasets, including Matterport3D [9], Gibson [38], Replica [34] and ReplicaCAD

Figure 2.1: Images extracted from two of the 18 scenes from the Matterport3D test set. (a): Scene with ID `2t7WUuJeko7`; (b): Scene with ID `RPmz2sHmrrY`.

[35]. In this work, we use the Matterport3D dataset. It is a diverse and visually realistic 3D indoor space dataset, and it also offers a large amount of training space for a variety of supervised and self-supervised computer vision tasks. We chose it for two main reasons:

1. The original Habitat RL agents were trained in its training set [31].

2. The test set is publicly available. The test set contains 18 scenes and a total of 1008 PointGoal navigation episodes [31].

We show some example images from this data set in Figure 2.1.

Ingesting the datasets mentioned above into Gazebo is not practical, so we used the `turtlebot3_house` scene from the `turtlebot3_gazebo` package [4] when testing a Habitat agent in Gazebo.

Chapter 3

ROS Cross Habitat Interface

In this chapter, we first discuss the four most important design requirements that guided the design of the ROS Cross Habitat Interface. In the second section, we elaborate on its core concepts and components. We illustrate its architecture as well as three operational modes in the third section.

3.1 Design Requirements

As pointed out in Chapter 2, there are a number of properties which are desirable or even required for modern robotics software. Drawing on knowledge gleaned from working with other packages, we established four requirements which guided our design of the `ros_x_habitat` interface. With these requirements, we seek to produce an interface which will allow users to exploit the strengths of both AI Habitat and ROS to build better embodied agents.

3.1.1 Serve as a Lightweight Communication Bridge

The interface should allow agents to communicate through ROS with sensors and actuators with minimal impact on performance and execution time. To explore this requirement, we connect a Habitat agent with Habitat Sim, and compare performance and execution times with and without ROS serving as the communication medium in Section 4.1.

3.1.2 Support Different Agents

The interface should allow users to deploy a variety of planning agents; in other words, it should be robust and modular. To explore this requirement we demonstrate that classical ROS planners can navigate within Habitat Sim using the interface in Section 4.2.

3.1.3 Support Different Environments

In further support of modularity and robustness, the interface should allow users to evaluate Habitat agents in other simulation environments with ROS bridges. To explore this requirement we demonstrate that a Habitat agent can navigate in Gazebo using the interface in Section 4.3. Although we do not demonstrate control of a physical robot, the test against Gazebo lends confidence that the communications plumbing to do so is operational.

3.1.4 Support Intermediate Data Processing

One of the useful features of the ROS architecture of communicating nodes is the ease with which data transforming filters can be inserted between producers and consumers with just a little rewiring. In the case of Habitat trained agents, we can take advantage of this type of transformation to test agents trained in the high-throughput, physics-free, discrete-time mode of Habitat Sim against higher-fidelity, physics-based, continuous-time simulators: The discrete actions are converted into continuous control signals. Sections 4.1 and 4.2 demonstrate success on this requirement. In Chapter 5, we also demonstrate use of this capability in a novel navigation task.

3.2 Core Concepts

To formally define the `ros_x_habitat` interface we use four main concepts:

1. The *agent* is an embodied entity which consumes observations and produces actions.

2. The *environment* is either a real space or a simulated space in which the agent's embodiment and other objects exist.

3. The *actions* are decisions the agent makes, typically with an objective in mind, that affect its embodiment and/or the environment.
4. The *observations* are information about the environment and/or its embodiment to which the agent has access.

The `ros_x_habitat` interface is intended to bridge different combinations of agents and environments—with an emphasis on Habitat-trained agents and the Habitat Sim environment—through the ROS middleware capabilities, and support taking advantage of other capabilities available within the ROS ecosystem, such as classical planning agents and the Gazebo simulation environment.

To create the interface, we encapsulate each of the four concepts mentioned above into ROS-based components so that they can take advantage of the ROS communication stack including messages, topics, and services. The agent and the environment will each be wrapped inside a ROS node where they subscribe to the topics of interest and connect to the services they require. For example, an embodied agent requires sensor observations from the environment in order to produce actions, and the environment will have new information about its states after the agent has taken an action. In this case, we put the agent inside a ROS node so it can publish its actions to a ROS topic and the environment (also encapsulated in a node) can extract the actions from that topic. Observations are also exchanged in such a manner. Since there is no direct communication required between the agent and the environment, each of them can run independently using their original technical setup and all of the information exchange happens through the interface. Additionally, other nodes can be added to help process the data shared via the topics to make the interface more flexible. One can easily introduce an intermediate node to map data that is not in the right format for the consumer (observations going to the agent, or actions going to the environment). In this way, neither the agent or the environment needs to be modified to handle new data types, maximizing their re-usability.

3.3 Architecture

In this section, we elaborate how the `ros_x_habitat` interface bridges the four concepts introduced in Section 3.2. We present three operating modes, each a representative use case of our interface.

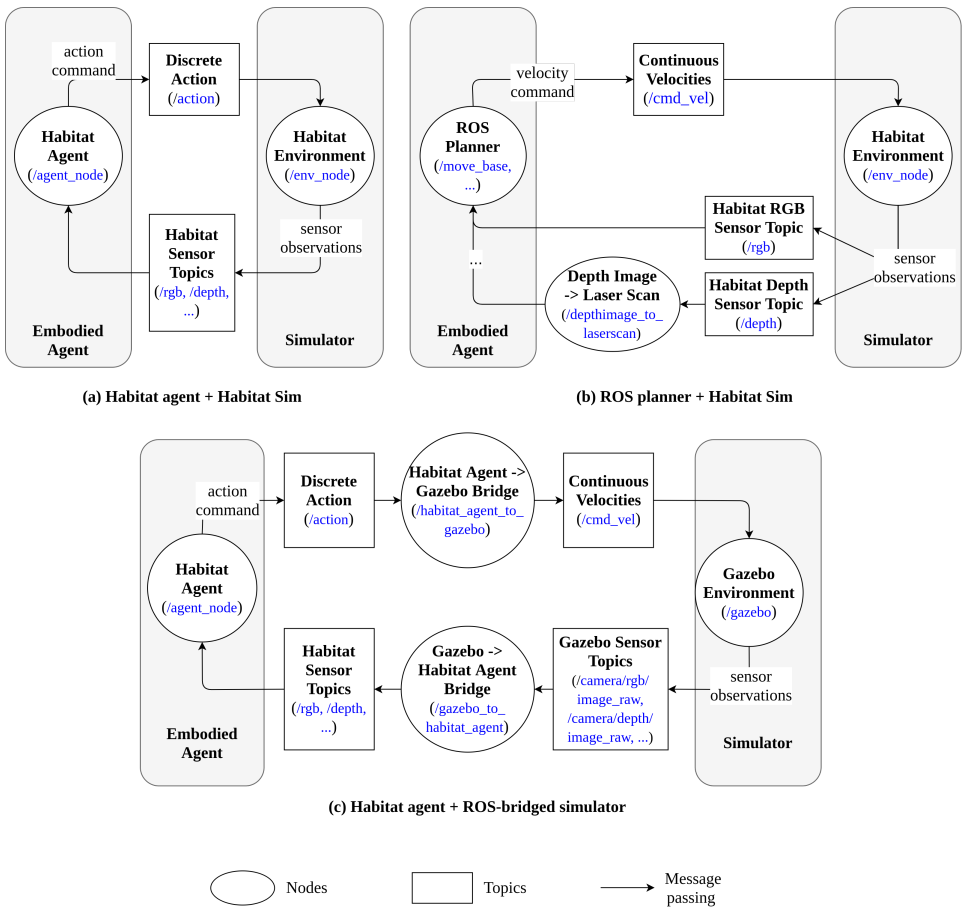

action command
Discrete Action (/action)
Habitat Agent (/agent_node)
Habitat Environment (/env_node)
sensor observations
Habitat Sensor Topics (/rgb, /depth, ...)
Embodied Agent
Simulator
(a) Habitat agent + Habitat Sim
velocity command
Continuous Velocities (/cmd_vel)
ROS Planner (/move_base, ...)
Habitat Environment (/env_node)
Habitat RGB Sensor Topic (/rgb)
sensor observations
...
Depth Image -> Laser Scan (/depthimage_to_laserscan)
Habitat Depth Sensor Topic (/depth)
Embodied Agent
Simulator
(b) ROS planner + Habitat Sim
action command
Discrete Action (/action)
Habitat Agent -> Gazebo Bridge (/habitat_agent_to_gazebo)
Continuous Velocities (/cmd_vel)
Habitat Agent (/agent_node)
Gazebo Environment (/gazebo)
sensor observations
Habitat Sensor Topics (/rgb, /depth, ...)
Gazebo -> Habitat Agent Bridge (/gazebo_to_habitat_agent)
Gazebo Sensor Topics (/camera/rgb/image_raw, /camera/depth/image_raw, ...)
Embodied Agent
Simulator
(c) Habitat agent + ROS-bridged simulator
Nodes
Topics
Message passing

Figure 3.1: The architecture of `ros_x_habitat` under each of the three operating modes. For simplicity, we omitted auxiliary nodes, topics, and services; for example, topics and nodes performing mapping actions under operating mode (b).

3.3.1 Bridging Communications Only

In this mode, our interface only serves as a communication middleware. Specifically, the agent and the environment are each encapsulated inside a ROS node designed for the interface. The actions and the observations are published to their corresponding interface topics. In this way, no direct communications occur between the agent and the environment. Figure 3.1(a) shows a Habitat agent navigating in a Habitat Sim-rendered scene, with or without physics, through ROS. It should be noted that this mode is not an expected use-case of the interface for external users, as it would be more efficient to connect Habitat agents directly with Habitat Sim through the existing AI Habitat interfaces. Instead, this mode is intended for testing the effect of the interface on performance and execution time.

3.3.2 One-way Data Processing Support

To the features of the communications only mode we add observational data conversion nodes so that a different agent can use the converted data to produce actions. In the specific case shown in Figure 3.1(b), the ROS planner being used expects laser scan readings as input, but Habitat Sim supports only a depth camera sensor. We use a standard ROS package [1] to convert the depth image data to a pseudo laser scan. However, the data conversion is one-directional in this mode since only the observations are converted with the support of our interface.

3.3.3 Two-way Data Processing Support

Of course, transformations can be applied to communications in both directions. As an example, Figure 3.1(c) shows a discrete action Habitat agent navigating in a Gazebo-rendered scene. Intermediate nodes are added to convert Gazebo's sensory data into formats that the Habitat agent expects, and to convert the Habitat agent's action outputs into the continuous velocity commands that the Gazebo embodiment asset expects.

Chapter 4

Experiments on Operating Modes

In this chapter, we conduct experiments in each of the operating modes introduced in Section 3.3. Section 4.1 will focus on studying the performance impact brought by our interface. Sections 4.2 and 4.3 will showcase how the interface can support different combinations of Agents and Simulators.

4.1 Navigation of Habitat Agents in Habitat Sim

A ROS bridge for the Habitat platform would be ineffective if it significantly degrades the performance of the Habitat components or it cannot use the new physics simulation capabilities of Habitat v2. Here we ask two research questions, aiming to explore the performance impact brought by the `ros_x_habitat` interface. To answer the research questions, we must first evaluate the Habitat v2 RGBD agent [35] on the PointGoal navigation task (in which the agent navigates from an initial position to a goal position [31]) over 1,008 episodes (an instance of the task) from the Matterport3D test set [9]. An episode ends after the agent issues the `STOP` command or the agent has taken 500 steps. Finally, we reflect on and analyze the results to give answers to the research questions.

4.1.1 Research Questions

1. **(RQ1) Given a Habitat navigation agent that was trained in Habitat Sim without enabling physics, how will this agent perform when physics is**

turned on? Having a Habitat agent's discrete actions (e.g. TURN_LEFT, TURN_RIGHT, ...) converted to a sequence of velocity commands allows us to leverage Habitat Sim's physics engine (i.e. Bullet Physics [11]) to simulate the actuation process in a more realistic fashion. We would like to measure how much of an impact physics has on navigation performance and execution speed.

2. **(RQ2) Does the ROS middleware impair navigation performance or introduce excessive run-time overheads?** First, we would like to verify that adding the ROS interface as a middleware does not negatively impact a Habitat agent's navigation performance within Habitat Sim. Second, we would like to measure how much ROS overhead impacts the execution speed of the task (including both agent and simulation components).

Note that these research questions and the experiments performed in this section are not an expected use case for `ros_x_habitat`; after all, running a Habitat agent in Habitat Sim is precisely what AI Habitat is already designed to do. The expected use cases of `ros_x_habitat` for robotics researchers are demonstrated in Sections 4.2 and 4.3.

4.1.2 Methodology

Evaluation metrics. We employ the Success Weighted by Path Length (`spl`) metric to evaluate the RGBD agent's navigation performance in each episode [31, 35]. This metric lies in the range $[0, 1]$ and measures the length of the traversed path relative to the shortest path from the source to the destination. An `spl` closer to 1 implies a shorter path and thus better performance. Failure to reach the goal is defined as an `spl` of 0.

To capture our interface's impact on execution speed we implement a set of timing measurements so that we capture the impact on individual components as well as the overall execution:

1. Per-step agent time (`agent_time`): Given sensor data as input, the CPU time used by a Habitat agent to produce an action.

2. Per-step simulation time (`sim_time`): Given an action from the agent as input, the CPU time used by Habitat Sim to update the world.

3. Total running time (`running_time`): The total wall clock time of evaluating an agent's navigation over all 1,008 episodes. This metric not only accounts for the agent's action time and the simulator's step time but also all of the computational overheads, such as initialization and inter-process communication.

Experiment configurations. We conducted our experiments on four configurations in order to independently observe the impact of introducing physics-based simulation and adding ROS.

1. `-Physics & -ROS`. We have the Habitat RGBD agent actuate in its default, discrete action space without using the ROS middleware, and we run Habitat Sim without physics turned on. This setting is the configuration in which the agent was trained, and serves as a baseline for our experiment.

2. `+Physics & -ROS`. We enable physics-based simulation, but do not use ROS. We explain below what the physics-based simulation entails.

3. `-Physics & +ROS`. The Habitat RGBD agent communicates with Habitat Sim through the ROS interface, as shown in Figure 3.1 (a). The agent still navigates using its discrete action space, and the simulation is run without physics.

4. `+Physics & +ROS`. The combination of the previous two settings.

Physics-based simulation and mapping from discrete to continuous action space. What do we mean by physics-based simulation in the context of Habitat Sim? Table 4.1 summarizes how Habitat Sim operates with and without physics, where:

- The 3D asset we attached to our agent is the `LoCoBot` model provided in Habitat Sim's code base [31].

- An *action step* is defined as one update of the world's state due to the agent completing an action in a discrete time simulation without physics.

Table 4.1: Habitat Sim with and without Physics

	Agent's Geometry	**Agent's Physical Properties**	**Simulation of Motion**
Simulation without physics	A cylinder 0.1m in radius, 1.5m in height	Friction coefficient is undefined; mass defined but not used for simulation	World state advances by one *action step* for each discrete action simulated; for each action the agent is translated or rotated instantaneously; no forces simulated.
Physics-based simulation	Defined by the 3D asset attached to agent's scene node	Friction coefficient is defined; mass is defined and used to compute dynamics	World state advances by one *continuous step* for each velocity command; gravitational and frictional forces fully simulated.

- A *continuous step* is defined as the advancement of the world's state over a predefined time period ($\frac{1}{60}$ second in our experiments) in a continuous time simulation with physics.

Since the simulated robot embodiment in the physics-based simulation expects velocity inputs, we convert the RGBD agent's discrete actions into a sequence of velocity commands using Algorithm 1, where

- $control\_period$ is a user-supplied parameter and defines the time in seconds it takes an agent to complete a discrete action (set to 1 in our experiments);
- $steps\_per\_sec$ defines the number of continuous steps Habitat Sim advances for each second of simulated time (set to 60 in our experiments);
- the angular velocities are measured in degrees / second; and
- the linear velocities are measured in meters / second.

The problem of reproducibility. Given sensor observations as input, Habitat's reinforcement learning agents use a neural network to generate an action; however, the action is chosen non-deterministically by default: The output action is sampled from the entire action space, with each action's probability of being sampled

Algorithm 1 Convert a discrete action to a sequence of velocities, then simulate the action with physics.

```
Require: a discrete action as one of the following: move_forward, turn_left, or turn_right
 1: initialize linear_velocity, angular_velocity as zero vectors
 2: $num\_steps = control\_period \cdot steps\_per\_sec$
 3: if $action ==$ move_forward then
 4:     $linear\_velocity = [0.25/control\_period, 0, 0]$
 5: else if $action ==$ turn_left then
 6:     $angular\_velocity = [0, 0, 10.0/control\_period]$
 7: else if $action ==$ turn_right then
 8:     $angular\_velocity = [0, 0, -10.0/control\_period]$
 9: end if
10: for $count\_steps = 1, 2, \ldots, num\_steps$ do
11:     env.step_physics($(linear\_velocity, angular\_velocity)$)
12: end for
```

equal to its confidence score from the last layer. Sampling is implemented with a standard pseudo-random number generator, so we can impose determinism and achieve reproducible results by fixing the random number generator's initial seed.

In order to build confidence that the choice of initial seed does not impact experimental outcomes too significantly, we ran the `-Physics & -ROS` configuration (the fastest, since it does not include physics or ROS) using ten randomly chosen seeds. The variability in navigation and timing metrics between seeds was dwarfed by the variability between scenarios, so for the remaining (slower) configurations we ran only a single seed.

4.1.3 Results and Analysis

We return to the questions posed in Section 4.1.1.

(RQ 1) Given a Habitat navigation agent that was trained in Habitat Sim without physics, how effective is it when physics is turned on? Comparing subplots (a) and (c) in Figure 4.1 we can see the effect that enabling physics has on the distribution of `spl` over the 1,008 episodes. We see that the number of failed episodes (those with `spl` $= 0.0$) increased (by 129 or roughly 30%) after we mapped actions to velocities and introduced physics-based simulation, but the

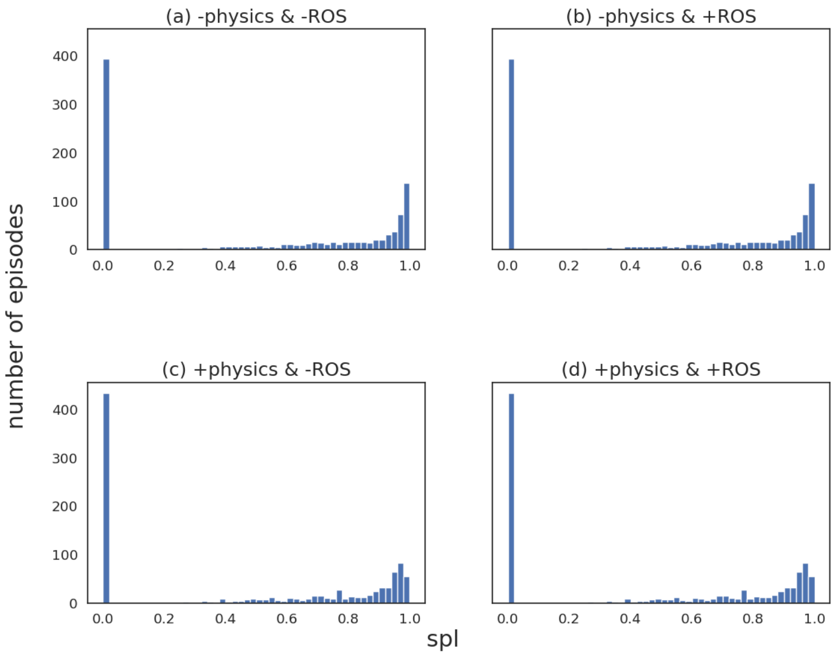


Figure 4.1: Distribution of `spl` under the four experimental configurations. Failed episodes have `spl` at 0.

average `spl` only dropped slightly (from 0.495 to 0.480). We then visually examined the 129 episodes in which the agent navigated successfully in the discrete action space but failed in the continuous action space. Four causes of failure came to our attention:

- Mesh cavity. This term refers to empty regions in a Matterport3D scene in which mesh vertices and surfaces are undefined. Because the agent visits many more states when physics is turned on—it must traverse intermediate states during any motion, rather than just jumping to the final state—it is more likely that an agent will blunder into one of these cavities. We found 15/129 failed cases in which the agent walked into a mesh cavity and was not able to get out and resume its original course.

- The robot is stuck. By “stuck” we mean the agent is not able to move forward but remains within the scene; for example, trapped against the corner of a table. The majority of the episodes (80/129) failed for this reason. While some of these failures were to be expected when an agent trained without friction or inertia suddenly has to deal with them, in some episodes the agent was stuck because the agent's movement is constrained to two dimensions by our implementation of Algorithm 1: It was not able to climb stairs when `move_forward` is converted into a sequence of planar velocities. This problem does not appear to manifest in the discrete-time / non-physics mode of Habitat Sim: the agent is automatically teleported to the correct height for whatever planar position it should occupy.

- The agent issued the `STOP` action at a distance close to the goal but slightly larger than 0.2m and was thus considered to have failed. This category involved 9/129 cases.

- The agent chose a poor path and was unable to achieve the goal within 500 steps, This category involved 25/129 cases.

Figures 4.2, 4.3 and 4.4 show timing results. Our summary is as follows:

- The per-step simulation time increased significantly with physics enabled (see first and third box plots in Figure 4.2). The average increased by roughly a factor of five from 0.063 to 0.337 per (action) step. The increase is due to the added computational cost incurred by Habitat Sim when a single discrete action is divided into *num_steps* (60 in this experiment) continuous steps when physics is enabled (see Algorithm 1).

- The per-step agent time has nearly identical distributions without or with physics-based simulation (see first and third box plots in Figure 4.3). The averages for these two configurations are 0.053 seconds and 0.054 seconds respectively.

- The total running time of `+Physics & -ROS` is about eight times that of `-Physics & -ROS` (see first and third bars of Figure 4.4). Aside from the increased per-step simulation time, extra time was required for simula-

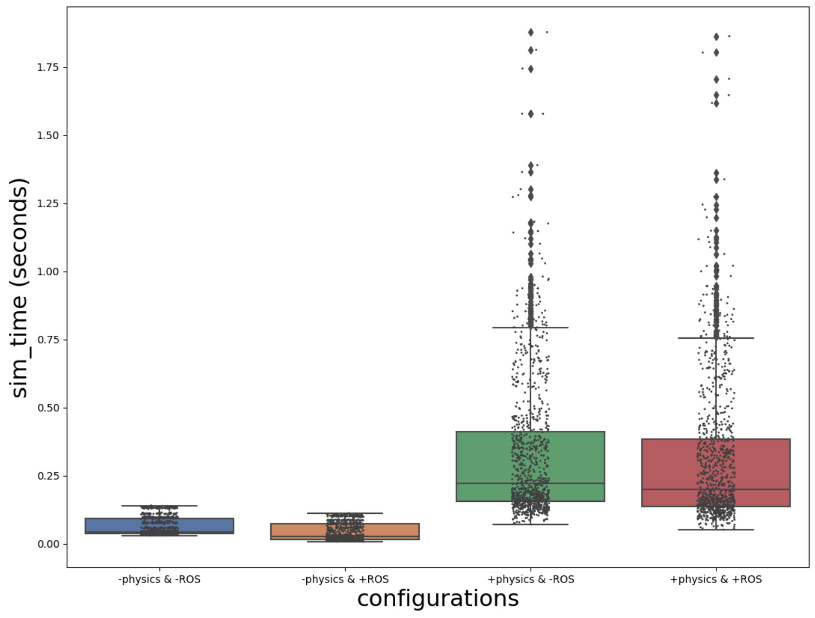


Figure 4.2: Distribution of per-step simulation time under the four experimental configurations. Each dot represents the result for a single episode. The box plot shows the boundaries between the quartiles.

tor reset (to load and delete physics-based object assets and reconfigure the simulator).

Given the minor gap in navigation performance and the significantly longer run-times for the physics-based simulator, this experiment further bolsters the claim that training RL agents in a discrete action space can be more cost effective [25], and that physics need not be turned on until validation or even final testing.

(RQ 2) Does the ROS middleware impair navigation performance or introduce excessive run-time overheads? In terms of navigation performance, once we removed non-determinism from the system by fixing the random seed value we were able to show that the introduction of ROS did not affect navigation performance at all, regardless of whether physics was enabled: Once ROS was intro-

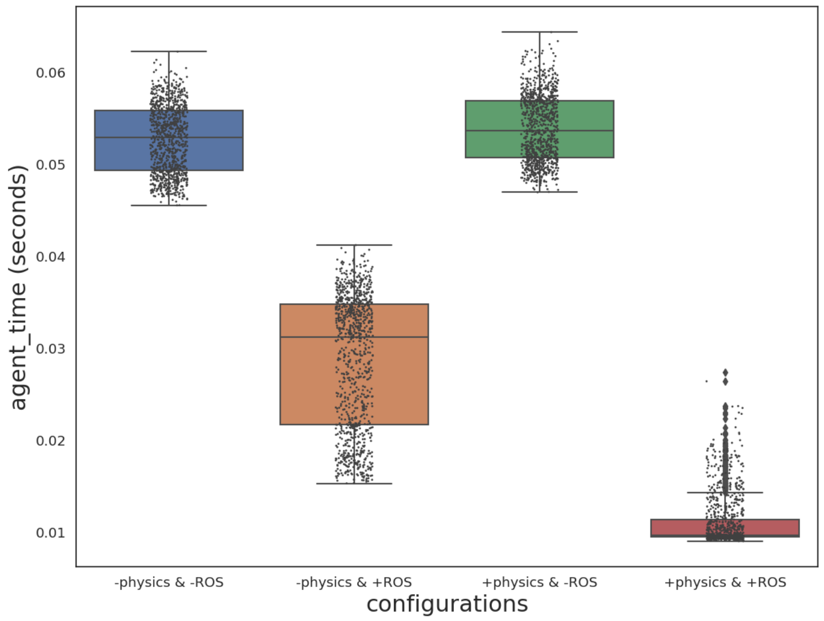


Figure 4.3: Distribution of per-step agent time under the four experimental configurations.

duced, each episode had the exact same number of steps and the SPL discrepancies were at the level of floating point round-off. Comparing the distributions in the left and right columns of figure 4.1 shows the latter effect qualitatively.

We now analyze the execution time:

- Both per-step simulation time and per-step agent time *dropped*(!) after we introduced ROS (see Figures 4.2 and 4.3), regardless of whether physics simulation was enabled or not. We are quite surprised by this outcome, as we did not anticipate that encapsulating either the agent or the simulator inside a ROS node would affect its execution time. Since encapsulation in ROS nodes changes the set of processes on the computer, we hypothesize that the difference may be due to changes in how the operating system is scheduling execution, although we have confirmed that all processes involved are executing at the same OS priority level. These measurements deserve further

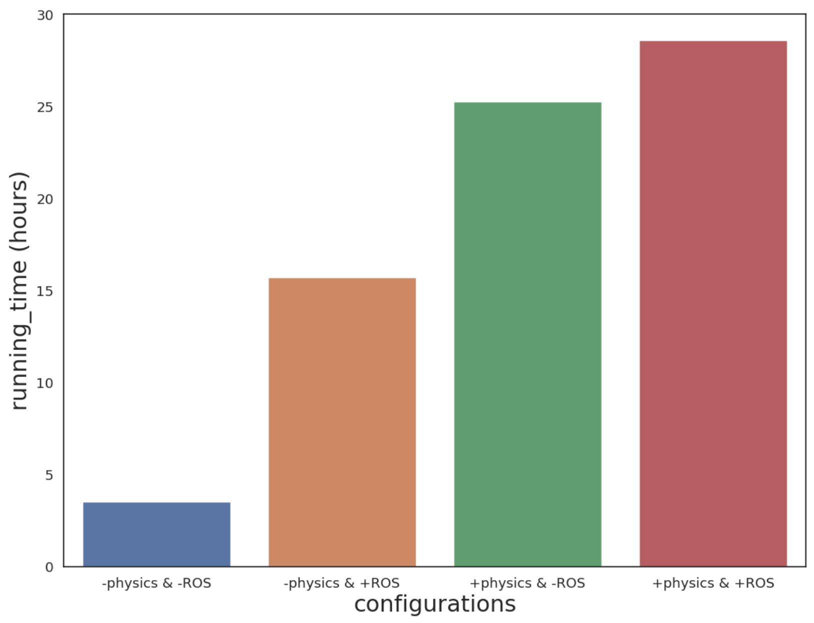


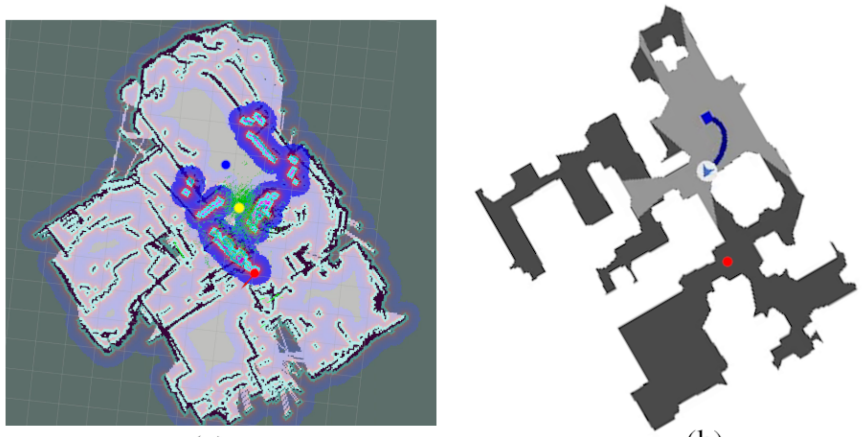
Figure 4.4: Total running time under the four experimental configurations for all 1008 episodes.

investigation.

- Figure 4.4 shows that the total running time increases by roughly five times from `-Physics & -ROS` to `-Physics & +ROS`, but only moderately from `+Physics & -ROS` to `+Physics & +ROS`. The increases in total running time are due to the overheads from inter-process communication between ROS nodes, as well as intra-process communication between threads running various service handlers and subscriber callbacks within each node.

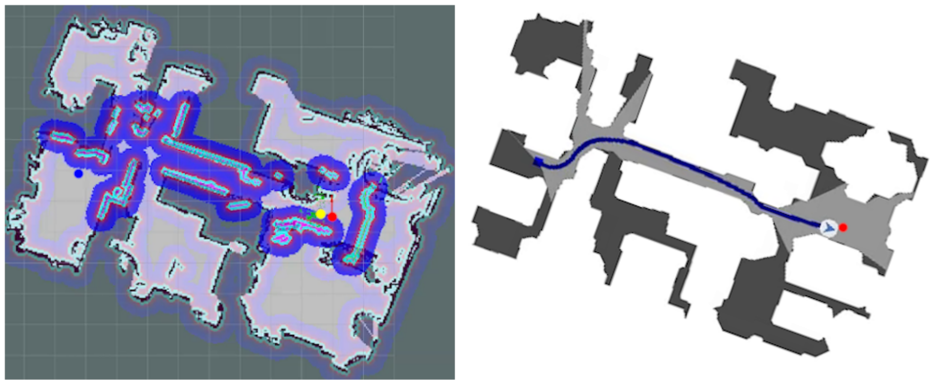
4.2 Navigation of ROS-based Planners in Habitat Sim

For robotics researchers interested in designing and testing navigation algorithms—be they classical or RL-based—we believe that this configuration of `ros_x_habitat` will be the most useful, as Habitat Sim provides a novel, high speed, photorealis-



(a) (b)

Figure 4.5: ROS-based planner `move_base` navigating in Matterport3D scene `2t7WUuJeko7` simulated by Habitat Sim. (a) The agent's final position overlayed on top of the map built by `rtab_map_ros`. (b) Top-down map of the scene from Habitat Sim. The blue curve indicates the agent's trajectory. The agent's starting position is marked in blue; final position is marked in yellow; and goal position is marked in red.



(a) (b)

Figure 4.6: ROS-based planner `move_base` navigating in Matterport3D scene `RPmz2sHmrrY` simulated by Habitat Sim. (a) The agent's final position and laser-scanned map. (b) Top-down map from Habitat Sim.

tic simulation environment in which to test such algorithms. In this section we do not intend to show the merits of a particular mapping, planning and/or navigation system, but simply that some standard packages from ROS can be easily and successfully connected to Habitat Sim v2.

First, we mapped two scenes from the Matterport3D dataset with `rtab_map_ros` [22]. Second, we attached the `LoCoBot` [31] asset to the agent in Habitat Sim, as we did to Habitat agents navigating in Habitat Sim with physics enabled. Finally, we manually set a goal position for the `move_base` [3] planner. Figures 4.5 and 4.6 show the generated map (left) and ground-truth map (right) overlaid with the final agent position and sensor readings (left) and path (right) for the two scenes. The planner failed to reach the goal in Figure 4.5 but succeeded in Figure 4.6.

We observed during the simulations that the agent often had a hard time localizing itself, especially in cluttered regions. We expect that with some tuning of the mapping, planning and navigation algorithms' parameters we could achieve better performance.

4.3 Navigation of Habitat Agents in Gazebo

For RL researchers interested in testing their agents on real robots, this configuration demonstrates the benefits of `ros_x_habitat`. Researchers familiar with ROS know that despite the significant sim2real gap in the Gazebo simulator, it can be an effective first target during design and testing because it will expose whole classes of common design bugs, such as incorrectly typed or connected data flows, coordinate transform errors, and gross timing issues.

In that vein, although we do not demonstrate a Habitat agent connected to a physical robot, by demonstrating a Habitat agent connected to Gazebo we show that `ros_x_habitat` can allow connection of Habitat agents to other simulation environments with ROS bridges, and that connection of a Habitat agent to a physical robot can be performed without the kinds of common design bugs mentioned above. Note that we do not expect the Habitat agent to perform particularly well in this Gazebo environment since it was trained in Habitat Sim environments providing much richer visual and geometric cues.

We use the `House` scene from the `turtlebot3_gazebo` package [4] (Fig-

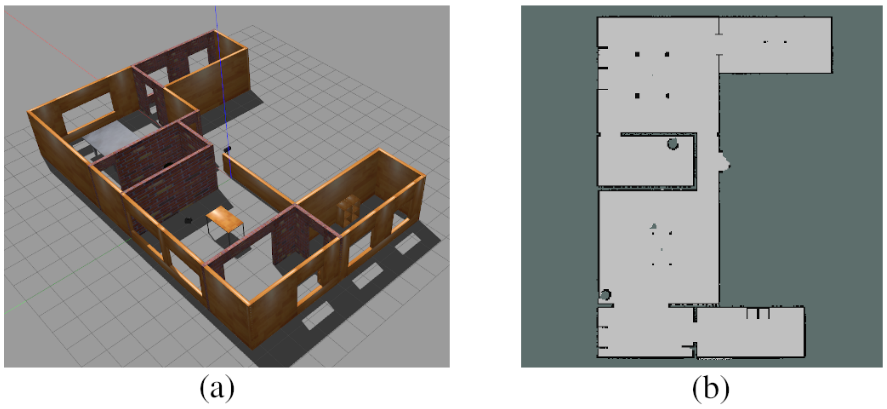
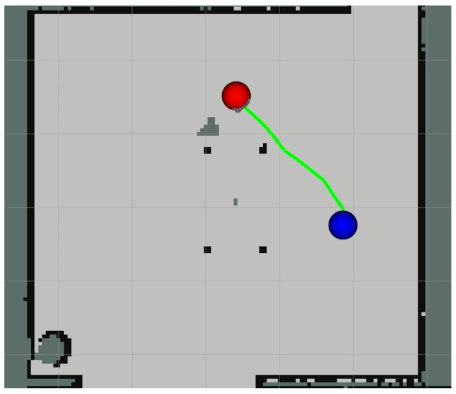
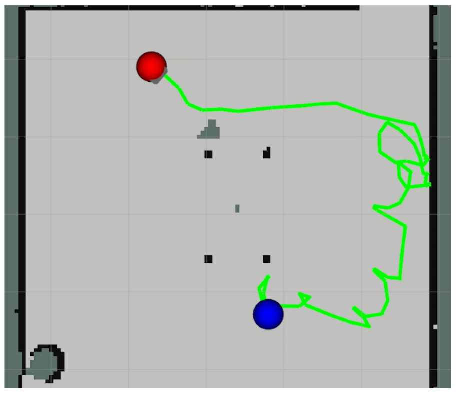


Figure 4.7: The `House` scene from ROS package `turtlebot3_gazebo`. a) a view of the scene in 3D; b) a map scanned from the scene using `hector_mapping` [20].

ure 4.7) to build three navigation episodes of increasing path length as shown in Figure 4.8. Then we instantiate a Habitat v2 RGBD agent inside each episode, and let it navigate until it reaches the goal or has generated 500 actions. Since we only intend to show that this configuration works, we do not report quantitative metrics such as `spl` or `sim_time` for these episodes. We see that the agent succeeded in the two shorter of the three episodes, although in the middle episode the agent took a path much longer than needed.


(a)


(b)

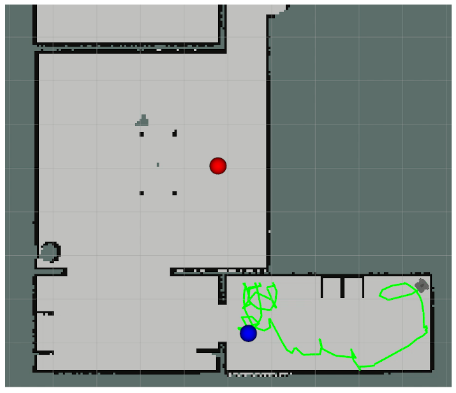

(c)

Figure 4.8: Maps showing the Habitat v2 RGBD agent navigating in three episodes. The blue spheres indicate the starting positions of each episode; the red spheres indicate the goal positions of each episode. The green curves represent paths traversed by the agent.

Chapter 5

Training of a Habitat Agent using the interface

In this chapter, we explore an example of what we expect would be a more common use case of the `ros_x_habitat` interface: Training and evaluating an agent for a novel navigation task in the Habitat environment. Specifically, we generalize the PointGoal navigation task discussed previously to a multiple or "one-of" point goal navigation task where the agent must navigate to any one of a collection of goal locations. In this case the goals are potential safe docking locations around tables, and are generated by a separate, recently developed computer vision pipeline [37]. We train a PPO (Proximal Policy Optimization) [32] RL agent to accomplish this task using Habitat's training infrastructure and evaluate the agent's performance using the `ros_x_habitat` interface.

5.1 Multiple PointGoal Navigation

For the PointGoal task defined in [31], an agent is initialized at a random starting position and orientation in an environment, and must navigate to within a proximity threshold of a target location. The coordinates of this target relative to the agent's position are provided before each step. In this section, we generalize this "single" PointGoal navigation task to a navigation task with multiple possible goals. Multiple goals can represent many meaningful navigation tasks in real life; for example, a delivery robot could choose the next delivery location from among the packages that it carries based on a variety of criteria.

For our purposes, we define the multiple PointGoal navigation task with the following components:

1. `Goals`: a set of targets defined as 3-dimensional positions (without heading).

2. `Start Position`: the position where the agent is initialized to be at the start of each simulation.

3. `Success`: a success is achieved whenever the agent navigates to within a proximity threshold of any one of the `Goals`.

The motivation for this task is an agent for autonomous navigation of a smart wheelchair. A common activity of daily living for users of powered wheelchairs is to approach, or "dock", with a table or desk so that the horizontal surface can be used for manual activities such as eating or writing. To solve this problem we need to generate PointGoals next to tables and desks in the Habitat Sim environment so that we can annotate training data with multiple goals and train the agent to perform the multiple PointGoal navigation task. In the next section we discuss how we use a computer vision pipeline along with a data collection node via the interface to generate docking locations for each simulation scene in which there is a suitable table or desk.

In addition to the set of target locations, training an agent for the multiple PointGoal navigation task requires the definition of a new sensory metric by which the agent gains knowledge about this set. Although not necessarily the most effective or realistic metric, we choose the simplest extension to the metric used by the single PointGoal task: Before each step the agent is told the relative coordinates of the *nearest* target in the set. This data is delivered in the form of two values: **distance_to_goal** and **goal_position**. To compute these two values, we implement a **MultiPointGoalSensor** with Algorithm 2. We use Habitat Sim's built-in **is_navigable** function and **path_finder** class to determine whether a position is navigable and whether there exists a path between two positions. We also use Habitat Sim's **compute_point_goal** function to compute the goal's position in the agent's coordinate frame. Finally, we calculate the **distance_to_goal** from the closest **goal_position**. As Habitat Sim cannot handle goals that are unreachable at runtime, we ignore any such docking locations during an episode, and implement a simple but effective fallback mechanism in lines 8–10 to avoid crashes if all docking locations are discarded: If none of the docking goals we generated are reachable, we add the episode's original single PointGoal to the current training episode. This goal is guaranteed to be reachable as the authors in [31] manually generated the goals and made sure they are reachable from all valid starting locations. Although this fallback choice will not directly improve the agent's ability to navigate to a docking location, it will favour actions which cause the agent to navigate to a specified location (and will avoid the need for a costly training run restart).

Algorithm 2 Given a list of pre-defined goals from each episode, compute the nearest goal location

```
Require: a list of PointGoal in the form of 3-dimensional positions [x, y, z]
 1: initialize valid_goals as an empty list
 2: initialize agent_position and agent_rotation with the current agent states
 3: for each G in input list of PointGoal do
 4:     if G is at a navigable location and agent_position has a path to G then
 5:         add G to valid_goals
 6:     end if
 7: end for
 8: if valid_goals is empty then
 9:     add original_goal to valid_goals
10: end if
11: initialize minimum_distance_to_goal to MAX FLOAT
12: initialize closest_goal to an empty 3D vector
13: for each goal VG in valid_goals do
14:     magnitude_of_path = ShortestPathLength(agent_position, VG)
15:     if magnitude_of_path < minimum_distance_to_goal then
16:         closest_goal = VG
17:         minimum_distance_to_goal = magnitude_of_path
18:     end if
19: end for
20: return compute_point_goal(closest_goal, agent_position, agent_rotation)
```

5.2 Data Collection and Goal Generation Method

To quickly annotate training data with docking locations around tables and desks, we must reduce the manual work involved. Thukral [37] trained a computer-vision based network to automatically detect tables / desks from point cloud data and then generate suitable docking locations around them. The ROS architecture makes it easy to add data collection nodes through `ros_x_habitat` and thereby collect depth image data from Habitat Sim and convert this data into point clouds. After we have the point cloud data, it is straightforward to automate the goal generation process either at runtime or as a preprocessing step offline. Figure 5.1 shows how the **point_cloud_saver** node is added into a design which allows users to drive freely through the Habitat Sim environment.

With the above framework, we can easily drive through the training scene datasets used in the original Habitat work [31] looking for tables or desks, and then automatically generate parking positions for any we find. Figure 5.2 shows an example depth image that we collected through this interface, which is then converted to point cloud format and saved. Finally, we use the network trained in [37] to generate the 3D parking locations shown in Figure 5.3. Additional examples of tables we found in the Matterport3D scenes dataset can be found in Appendix A.

In order to reduce training time, we choose to generate the docking locations as a preprocessing step and modify the episode metadata from Habitat Sim v1 to include the new docking locations. We reuse the `start_positions` for the agent, and filter out all scenes which do not contain a table or desk, as well as all episodes where the start position and docking locations are not on the same floor.

5.3 Training of a Habitat Agent

As we did in Chapter 4, we use the Matterport3D scene datasets for the training and evaluation of a PPO RL agent. We train this PPO agent using Habitat-lab, which as previously mentioned, is a modular high-level library for end-to-end development in embodied AI. For this task, scenes are only relevant if there is a table or desk; therefore, after carefully reviewing the 61 scenes in Habitat's Matterport3D training datasets we select a total of 12 scenes that have tables and desks in them. There are a few scenes containing tables or desks that were not included because of

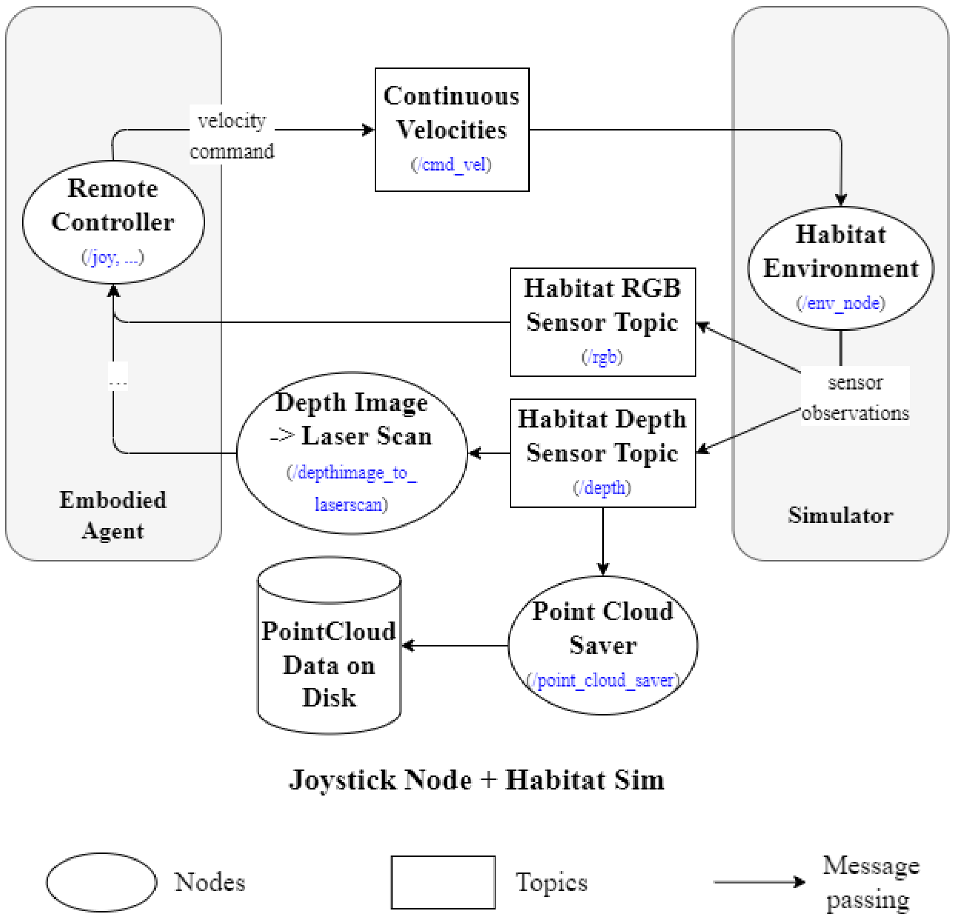


Figure 5.1: Architecture of `Point Cloud Data Saving Process`. We had already developed a design which allowed users to navigate freely through any Habitat Sim environment using a joystick for testing and debugging purposes. A single new node (bottom right) was added to listen to the existing depth sensor topic and save point clouds to disk.

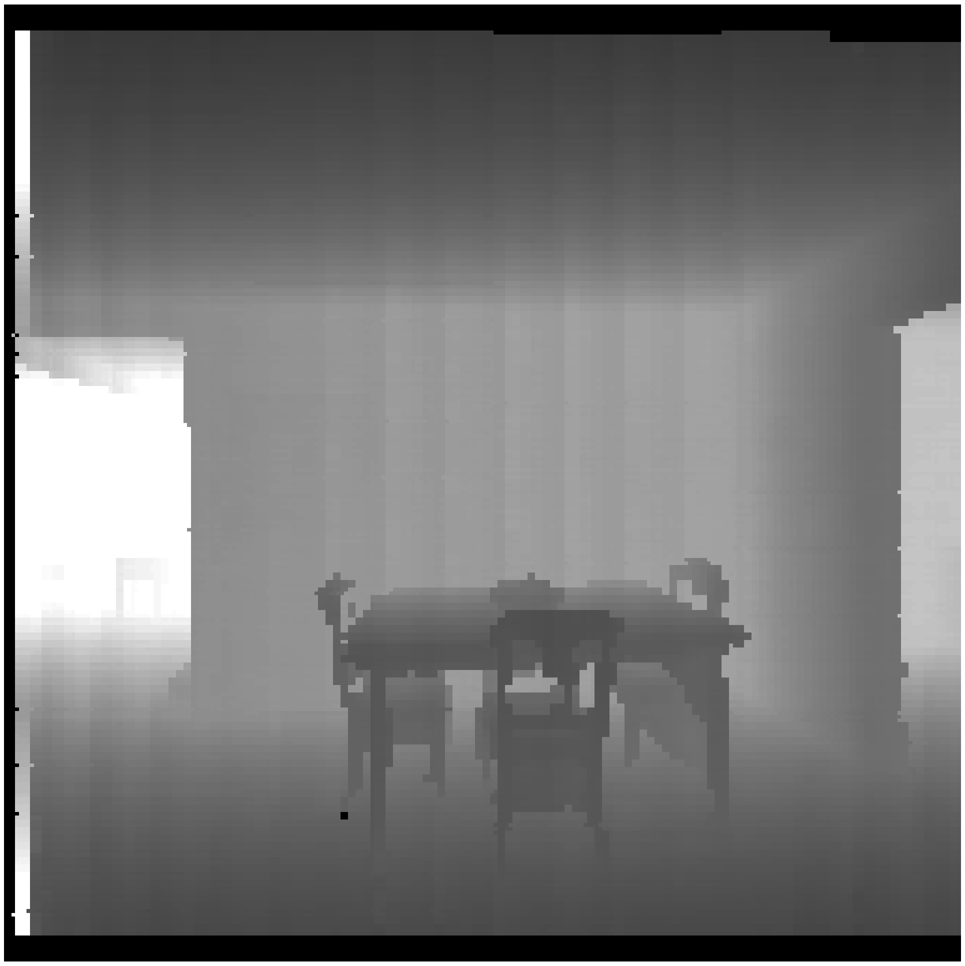


Figure 5.2: Example of a depth image collected in front of a table from the Matterport3D scene dataset.

- bad mesh quality in the vicinity of the table or desk which works poorly with the Goal Generation Method we built in Section 5.2 or
- the table or desk is in a position which is visible but to which the agent cannot navigate due to other obstacles.

Knowing that our computational resources are rather limited to compared to those used in [31] and [35], we pick docking locations around only a single table or desk in each scene.

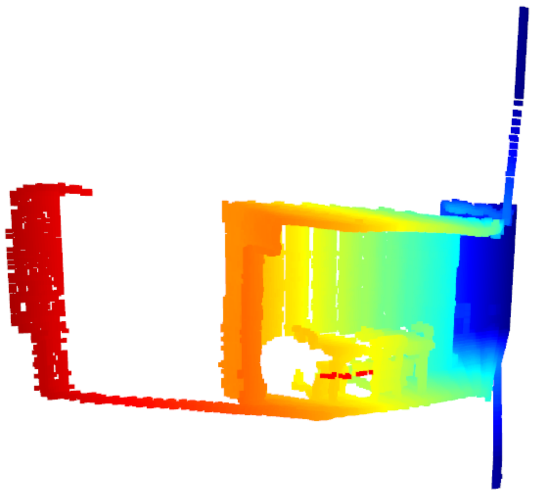


Figure 5.3: Example of docking locations surrounding a table from the Matterport3D scene datasets. The small red dots indicate docking locations that appear to be free of collision.

5.3.1 Training Configuration

We configure Habitat-lab's PPO trainer to use the following parameters:

```
RL:
  PPO:
    # ppo params
    clip_param: 0.2
    ppo_epoch: 4
    value_loss_coef: 0.5
    entropy_coef: 0.01
    num_mini_batch: 2
    lr: 2.5e-4
    eps: 1e-5
    max_grad_norm: 0.5
    num_steps: 128
    hidden_size: 512
    use_gae: True
    gamma: 0.99
    tau: 0.95
    use_linear_clip_decay: True
    use_linear_lr_decay: True
```

```
reward_window_size: 50
use_double_buffered_sampler: False
```

The agent itself is configured to have both a **RGB_SENSOR** and a **DEPTH_SENSOR** of dimension 256 * 256. We limit each training episode to a maximum of 500 simulation steps and use a success proximity threshold of 0.2 meters, as was used for the single PointGoal agent training.

5.3.2 Training Process

Training is performed on a desktop PC running Ubuntu 20.04 with an Intel® Core™ i7-10700K CPU @ 3.80 GHz with 16 cores, 64 GBs of RAM, and a Zotac RTX 3070 GPU with 8 GB memory. The total number of simulation steps we allocate for the training process is 75 million, and these are taken in a total of 474,837 episodes from the 12 scenes we selected.

With 4 Habitat simulation environments running in parallel the training process took roughly 60 GPU hours. We collected the SPL and success rate of the agent while it was being trained. Figure 5.4 shows the evolution of these metrics with respect to the number of training steps. Performance improves steadily through the first 20 million steps but plateaus after that; in fact, performance occasionally suffers temporary declines beyond 40 million steps.

5.4 Evaluation of the Trained Agent

To evaluate the agent we apply the Data Collection and Goal Generation Method we designed in Section 5.2 to the test set. Our objective is to demonstrate that we can train an agent to complete a novel task using the `ros_x_habitat` interface (in this case it was used to collect training data). Given our limited computational resources and hence limited number of training steps, we do not expect the agent's performance to be particularly impressive.

Because we are not seeking to accurately measure the performance, we use just 4 scenes from the Matterport3D test datasets for our evaluation. After goal generation and annotation, we have a total of 152 episodes for testing. We again choose a fixed seed for random number generation (in this case the value `7`) to ensure reproducibility.

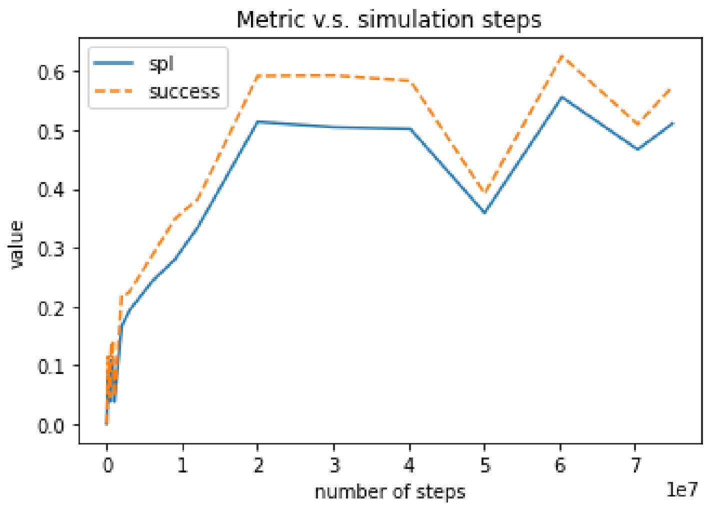


Figure 5.4: SPL and Success rate metrics v.s. the number of simulation steps in training

We collect the same metrics from the experiments as in Chapter 4. Specifically, we run the test cases with and without ROS serving as the communication channel between agent and simulation. As Figures 5.5 and 5.6 show, we observe no difference in the performance values with or without ROS.

Additionally, we include two top-down maps of test episodes as examples of how the trained agent navigates in the Multiple PointGoal Navigation task. Figure 5.7 shows a successful test navigation and Figure 5.8 shows an unsuccessful example.

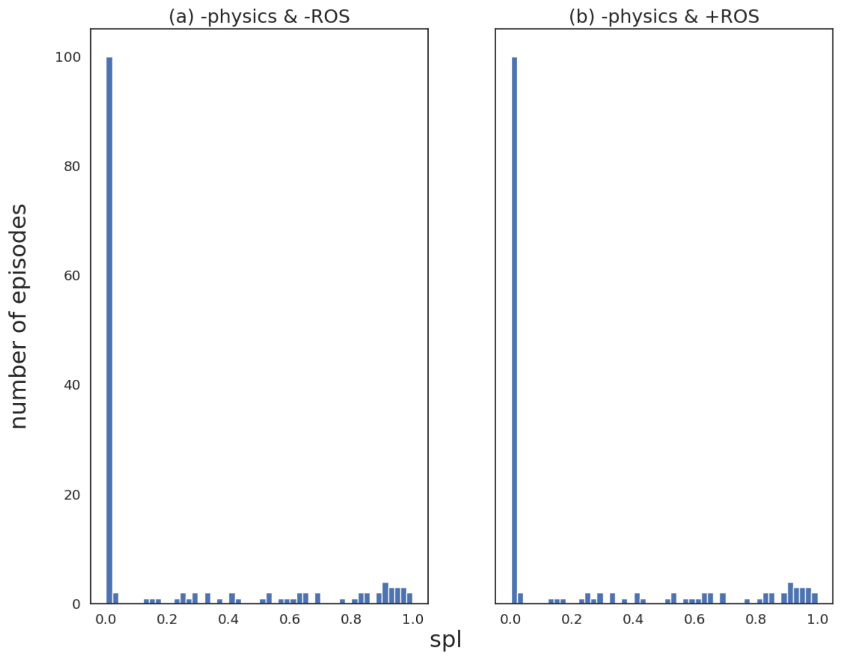


Figure 5.5: SPL of the trained Multiple PointGoal PPO agent in two test-time configurations. Note that agent training was completed without ROS to maximize training throughput.

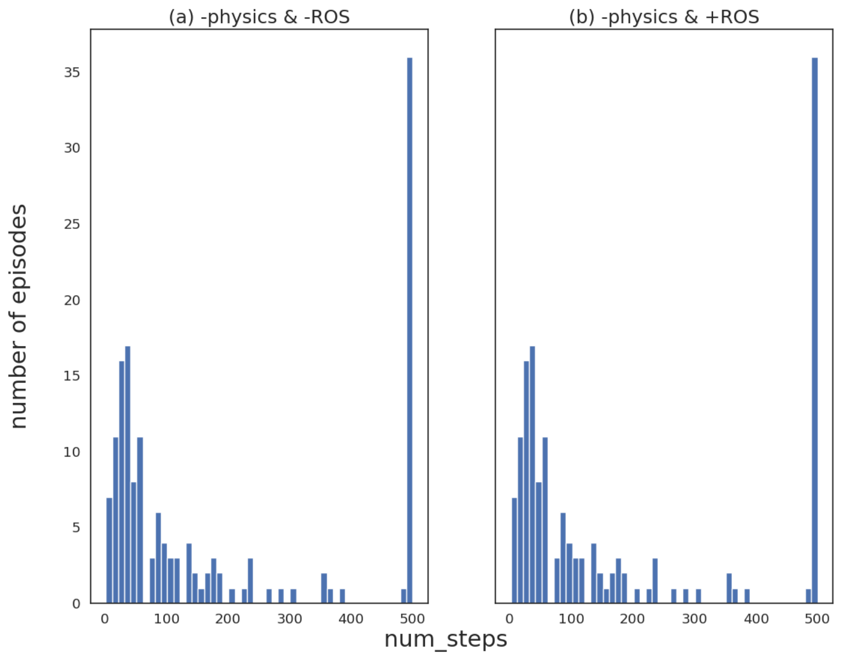


Figure 5.6: Number of simulation steps the trained Multiple PointGoal PPO agent took in the two test-time configurations.

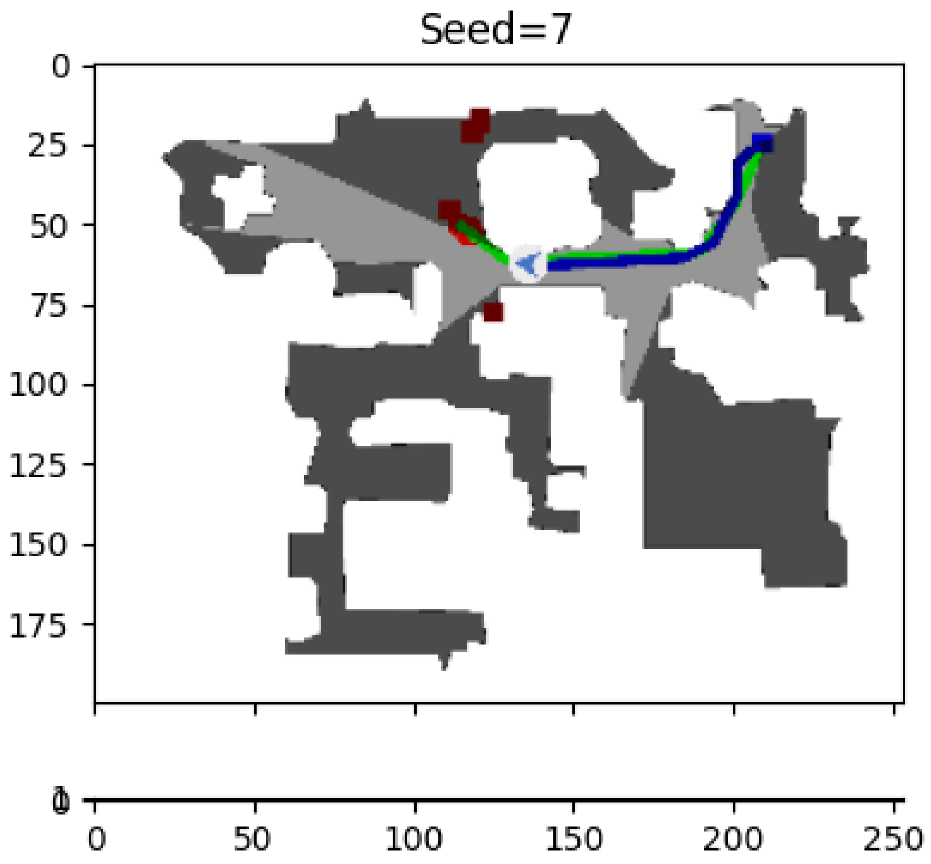


Figure 5.7: Successful Multiple PointGoal Navigation in scene `2t7WUuJeko7`. Potential docking locations are red boxes. Agent path starts from blue box and follows blue line. "Optimal" path for SPL calculation is shown as green line.

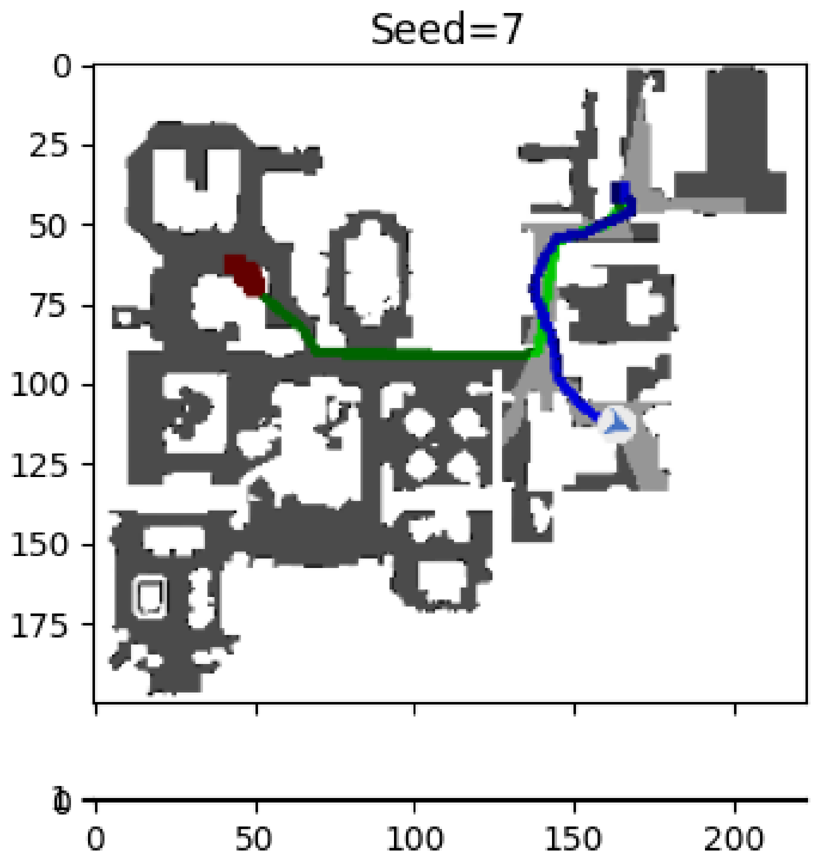


Figure 5.8: Failed Multiple PointGoal Navigation in scene `jtcxE69GiFV`. Potential docking locations are red boxes. Agent path starts from blue box and follows blue line. “Optimal” path for SPL calculation is shown as green line.

Chapter 6

Conclusion

We presented a software interface that bridges the AI Habitat platform with ROS middleware. Through this interface, researchers can benefit from easy-to-use and lightweight inter-process communication so that they can train/develop/test their embodied agents in both established simulators such as Gazebo or the state-of-the-art Habitat Sim. Although we demonstrate only two simulator options in this work, our design allows easy integration of additional platforms including physical robots. Last but not least, we explore an example of the interface's utility by showing how researchers can easily add a customized ROS node to collect and process data.

Using established metrics, we show that our interface introduces no impact on the navigation performance of embodied agents in the discrete action space and physics-free environment in which they were trained, and has a moderate execution time overhead. Furthermore, we show that if the same agents are evaluated in continuous time with physics enabled, the performance degradation is surprisingly modest, although the simulation speed is significantly longer.

To show the flexibility of our interface, we also demonstrated how the Habitat trained agent can be used in a Gazebo simulation environment, and how a classical ROS navigation stack can be used in the Habitat Sim environment.

6.1 Future Work

The original plan for this work was to demonstrate a Habitat trained agent embodied in a physical robot through ROS. Unfortunately, pandemic safety concerns did not allow access to suitable hardware until after the experimental component of the work was completed. Nevertheless, such a demonstration remains at the top of our future goals.

One of the key benefits of a machine learning approach to robot control is the ability to train agents for a variety of tasks. There is no reason that `ros_x_habitat` could not be used for tasks more general than basic navigation; for example, the agents trained for the Pick Task from [35].

Of course, agent training needs training data. Habitat Sim allows access to a huge collection of suitable environments, but it is not the only possibility. We would like to take advantage of `ros_x_habitat` to bridge to other ROS connected simulators and thereby access a wider variety of 3D assets and scenes. Simulators such as Isaac Sim [27] can also offer state-of-the-art photorealistic and physically realistic environments. We would be interested in seeing Habitat agents being trained in such environments for better navigation performance.

Bibliography

[1] depthimage_to_laserscan - ros wiki. URL http://wiki.ros.org/depthimage_to_laserscan. → page 14

[2] dwa_local_planner - ros wiki. URL http://wiki.ros.org/dwa_local_planner. → pages 2, 4, 6

[3] move_base - ros wiki. URL http://wiki.ros.org/move_base. → pages 2, 4, 6, 26

[4] turtlebot3_gazebo - ros wiki. URL http://wiki.ros.org/turtlebot3_gazebo. → pages 9, 26

[5] Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017. doi:10.1109/TRO.2017.2705103. → page 6

[6] J. M. Beer, A. D. Fisk, and W. A. Rogers. Toward a framework for levels of robot autonomy in human-robot interaction. *Journal of human-robot interaction*, 3(2):74, 2014. → page 2

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. → page 1

[8] G. Brunner, O. Richter, Y. Wang, and R. Wattenhofer. Teaching a machine to read maps with deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. URL https://ojs.aaai.org/index.php/AAAI/article/view/11645. → page 6

[9] A. Chang, A. Dai, T. Funkhouser, M. Halber, M. Niessner, M. Savva, S. Song, A. Zeng, and Y. Zhang. Matterport3d: Learning from rgb-d data in indoor environments. *arXiv preprint arXiv:1709.06158*, 2017. → pages 7, 8, 15

[10] D. S. Chaplot, D. Gandhi, S. Gupta, A. Gupta, and R. Salakhutdinov. Learning to explore using active neural SLAM. *CoRR*, abs/2004.05155, 2020. URL https://arxiv.org/abs/2004.05155. → page 6

[11] E. Coumans and Y. Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021. → page 16

[12] S. Datta, O. Maksymets, J. Hoffman, S. Lee, D. Batra, and D. Parikh. Integrating egocentric localization for more realistic point-goal navigation agents. *arXiv preprint arXiv:2009.03231*, 2020. → page 6

[13] A. Elkady and T. Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 05 2012. doi:10.1155/2012/959013. → page 7

[14] A. Hentout, A. Maoudj, and B. Bouzouia. A survey of development frameworks for robotics. In *2016 8th International Conference on Modelling, Identification and Control (ICMIC)*, pages 67–72, 2016. doi:10.1109/ICMIC.2016.7804217. → pages 7, 8

[15] A. Hentout, A. Maoudj, and B. Bouzouia. A survey of development frameworks for robotics. In *2016 8th International Conference on Modelling, Identification and Control (ICMIC)*, pages 67–72, 2016. doi:10.1109/ICMIC.2016.7804217. → page 8

[16] K. D. Julian and M. J. Kochenderfer. Distributed wildfire surveillance with autonomous aircraft using deep reinforcement learning. *Journal of Guidance, Control, and Dynamics*, 42(8):1768–1778, 2019. → page 1

[17] A. Juliani, V. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange. Unity: A general platform for intelligent agents. *CoRR*, abs/1809.02627, 2018. URL http://arxiv.org/abs/1809.02627. → pages 1, 6, 7

[18] A. Kadian, J. Truong, A. Gokaslan, A. Clegg, E. Wijmans, S. Lee, M. Savva, S. Chernova, and D. Batra. Sim2real predictivity: Does evaluation in simulation predict real-world performance. *IEEE Robotics and Automation Letters*, PP:1–1, 08 2020. doi:10.1109/LRA.2020.3013848. → page 8

[19] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004. doi:10.1109/IROS.2004.1389727. → page 6

[20] S. Kohlbrecher, J. Meyer, T. Graber, K. Petersen, U. Klingauf, and O. von Stryk. Hector open source modules for autonomous mapping and navigation with rescue robots. In S. Behnke, M. Veloso, A. Visser, and R. Xiong, editors, *RoboCup 2013: Robot World Cup XVII*, pages 624–631, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44468-9. → pages x, 2, 6, 27

[21] J. Kramer and M. Scheutz. Development environments for autonomous mobile robots: A survey. *Auton. Robots*, 22:101–132, 02 2007. doi:10.1007/s10514-006-9013-8. → page 7

[22] M. Labbé and F. Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446, 2019. doi:https://doi.org/10.1002/rob.21831. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21831. → pages 6, 26

[23] Z. Liang, Z. Cai, M. Li, and W. Yang. Parallel gym gazebo: a scalable parallel robot deep reinforcement learning platform. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 206–213, 2019. doi:10.1109/ICTAI.2019.00037. → page 2

[24] G. Magyar, P. Sincak, and Z. Krizsán. Comparison study of robotic middleware for robotic applications. *Advances in Intelligent Systems and Computing*, 316:121–128, 01 2015. doi:10.1007/978-3-319-10783-7_13. → page 8

[25] E. Marchesini and A. Farinelli. Discrete deep reinforcement learning for mapless navigation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10688–10694, 2020. doi:10.1109/ICRA40945.2020.9196739. → pages 6, 22

[26] W. Masson, P. Ranchod, and G. Konidaris. Reinforcement learning with parameterized actions. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016. → page 6

[27] F. Monteiro, A. L. Vieira e Silva, J. M. Teixeira, and V. Teichrieb. Simulating real robots in virtual environments using nvidia's isaac sdk. pages 47–48, 10 2019. doi:10.5753/svr_estendido.2019.8471. → pages 6, 43

[28] A. Murali, T. Chen, K. V. Alwala, D. Gandhi, L. Pinto, S. Gupta, and A. Gupta. Pyrobot: An open-source robotics framework for research and

benchmarking. *CoRR*, abs/1906.08236, 2019. URL http://arxiv.org/abs/1906.08236. → page 8

[29] H. Oliff, Y. Liu, M. Kumar, M. Williams, and M. Ryan. Reinforcement learning for facilitating human-robot-interaction in manufacturing. *Journal of Manufacturing Systems*, 56:326–340, 2020. → page 1

[30] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. volume 3, 01 2009. → page 8

[31] M. Savva, A. Kadian, O. Maksymets, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik, et al. Habitat: A platform for embodied ai research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9339–9347, 2019. → pages 1, 6, 7, 8, 9, 15, 16, 17, 26, 29, 31, 32, 34

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017. → page 29

[33] A. Staranowicz and G. L. Mariottini. A survey and comparison of commercial and open-source robotic simulator software. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307727. doi:10.1145/2141622.2141689. URL https://doi.org/10.1145/2141622.2141689. → page 6

[34] J. Straub, T. Whelan, L. Ma, Y. Chen, E. Wijmans, S. Green, J. J. Engel, R. Mur-Artal, C. Ren, S. Verma, et al. The replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797*, 2019. → pages 7, 8

[35] A. Szot, A. Clegg, E. Undersander, E. Wijmans, Y. Zhao, J. Turner, N. Maestre, M. Mukadam, D. Chaplot, O. Maksymets, et al. Habitat 2.0: Training home assistants to rearrange their habitat. *arXiv preprint arXiv:2106.14405*, 2021. → pages 7, 8, 9, 15, 16, 34, 43

[36] L. Tai, G. Paolo, and M. Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 31–36. IEEE, 2017. → page 6

[37] S. Thukral. *ApproachFinder: real-time perception of potential docking locations for smart wheelchairs*. PhD thesis, University of British Columbia, 2022. URL https://open.library.ubc.ca/collections/ubctheses/24/items/1.0406508. → pages 29, 32

[38] F. Xia, A. R. Zamir, Z. He, A. Sax, J. Malik, and S. Savarese. Gibson env: Real-world perception for embodied agents. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9068–9079, 2018. → page 8

[39] F. Xiang, Y. Qin, K. Mo, Y. Xia, H. Zhu, F. Liu, M. Liu, H. Jiang, Y. Yuan, H. Wang, et al. Sapien: A simulated part-based interactive environment. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11097–11107, 2020. → page 7

[40] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero. Extending the openai gym for robotics: a toolkit for reinforcement learning using ROS and gazebo. *CoRR*, abs/1608.05742, 2016. URL http://arxiv.org/abs/1608.05742. → page 8

Appendix

Appendix A

Images Collected From Training Datasets

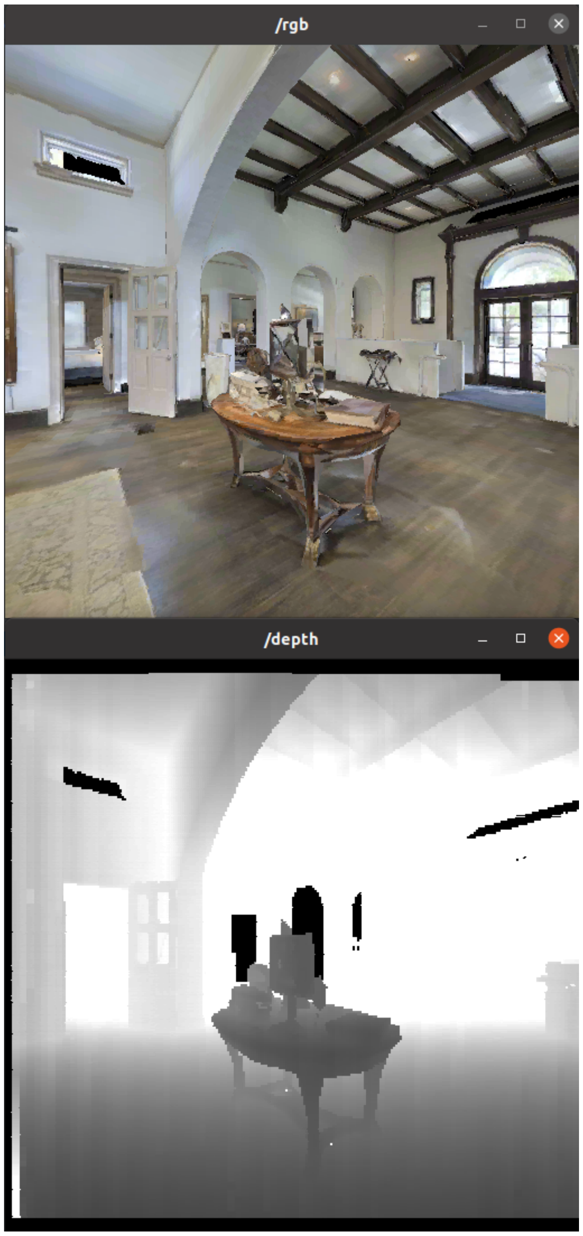


Figure A.1: A table from the training dataset used in Chapter 5

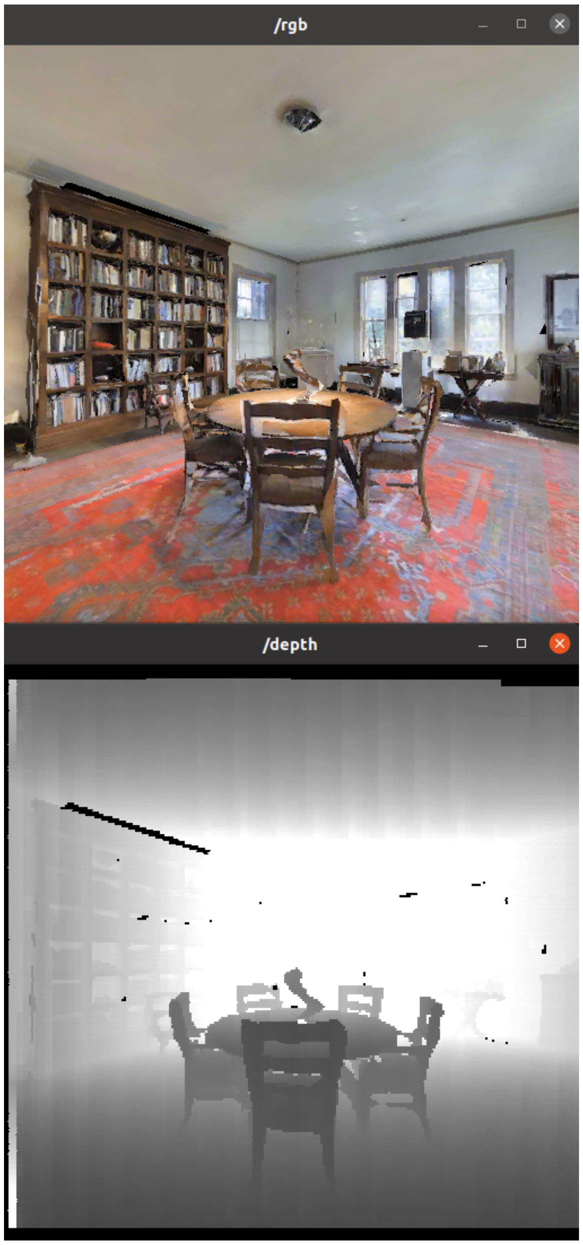


Figure A.2: A table from the training dataset used in Chapter 5

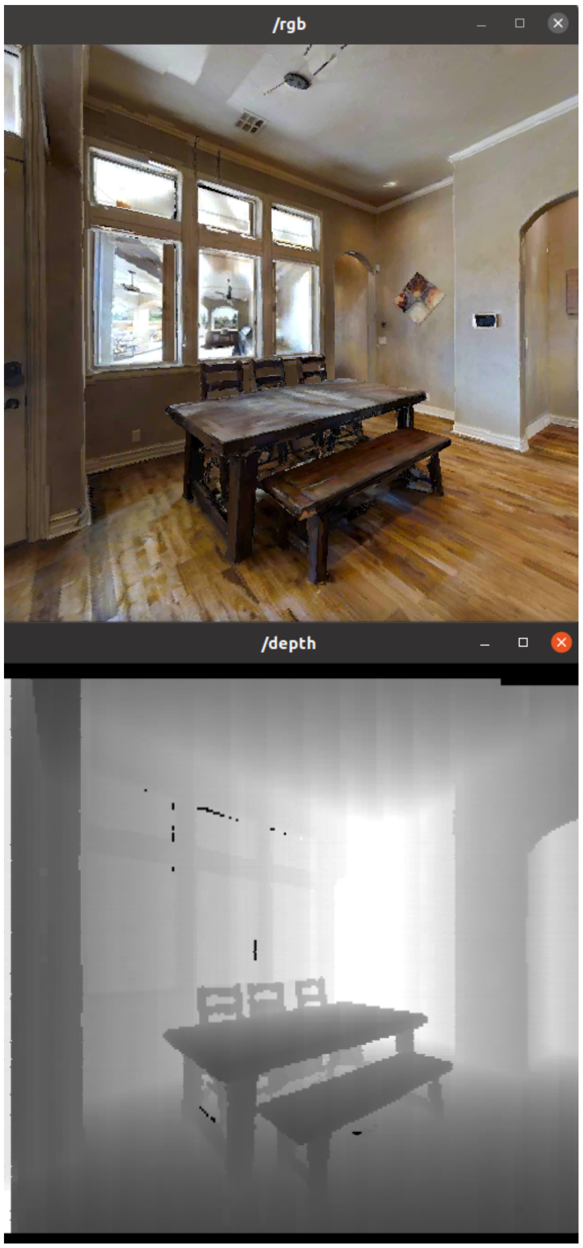


Figure A.3: A table from the training dataset used in Chapter 5

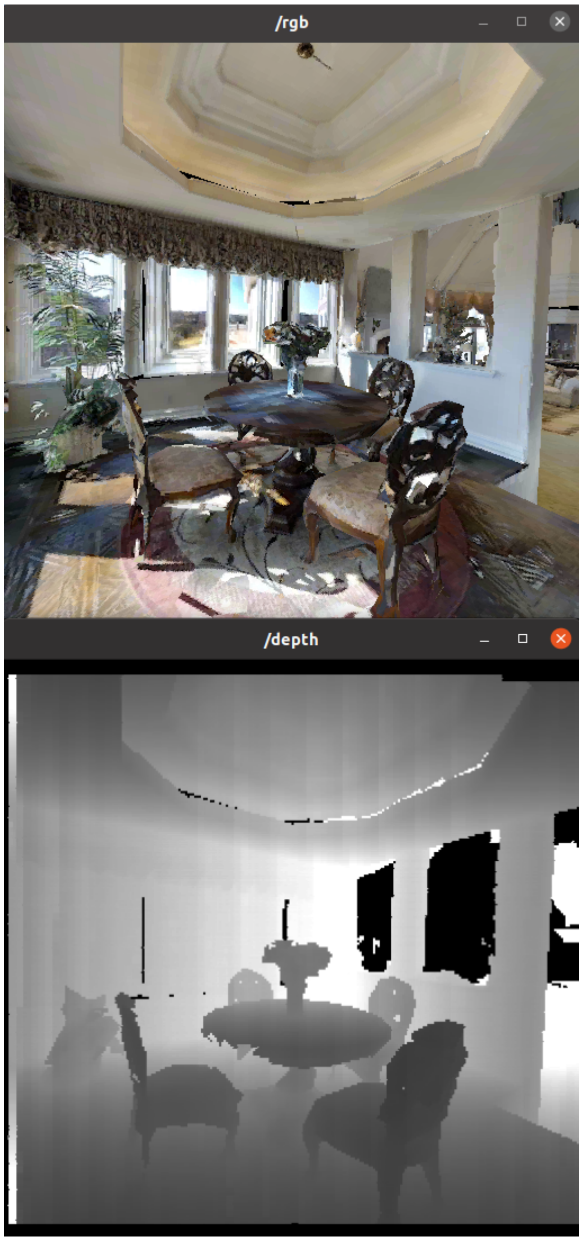


Figure A.4: A table from the training dataset used in Chapter 5

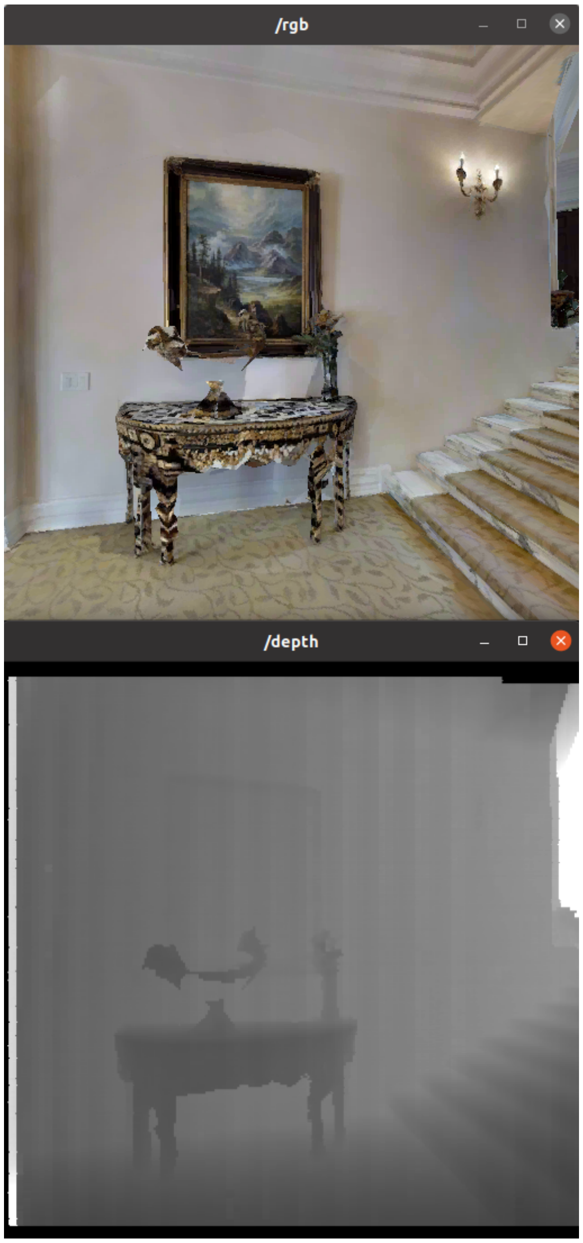


Figure A.5: A table from the training dataset used in Chapter 5