COMPARATIVE ANALYSIS OF ROS-UNITY3D AND ROS-GAZEBO FOR MOBILE GROUND ROBOT SIMULATION

by

JONATHAN THOMAS PLATT

KENNETH G. RICKS, COMMITTEE CHAIR
CHRIS S. CRAWFORD
RYAN A. TAYLOR

A THESIS

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department
of Electrical and Computer Engineering
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2022

ABSTRACT

Several highly anticipated technologies including self-driving cars, delivery robots, and interplanetary rovers are predicated on continued advances in autonomous navigation research. One of the most significant challenges being faced in the development of autonomous navigation systems is that training and testing them is time consuming, costly, and potentially dangerous. A common way to address these concerns without significantly impeding development is to use a computer simulated environment. Currently, several dedicated simulation suites exist, but they are limited in terms of environment size, visual quality, and feature sets. As a result, many researchers have begun to consider repurposing game engines as simulators to take advantage of their greater flexibility, scalability, and graphical fidelity.

This thesis investigates a robotics simulation suite based on the Unity3D game engine and ROS robotics middleware, collectively referred to as ROS-Unity3D, and compares it to the popular ROS-Gazebo robotics simulation suite. They are compared in terms of their architecture, environment creation process, resource usage, and accuracy while simulating an autonomous ground robot in real-time. Overall, ROS-Unity3D is found to be a viable alternative to ROS-Gazebo with support for many of the same file types and a powerful scripting interface for creating custom functionality like sensors. Test results indicate that ROS-Unity3D scales better to larger environments, has higher shadow quality, and is more capable of real-time LiDAR simulation than ROS-Gazebo. As for its advantages over ROS-Unity3D, ROS-Gazebo has a more streamlined interface between ROS and Gazebo, has more existing sensor plugins, and is more computer resource efficient for simulating small environments.

LIST OF ABBREVIATIONS AND SYMBOLS

2D	Two-dimensional
3D	Three-dimensional
ADAS	Advanced Drier Assistance System
API	Application Programming Interface
BSON	Binary JSON
CPU	Central Processing Unit
DARTS	Dynamics Algorithms for Real-Time Simulation
DLL	Dynamic Link Library
DoF	Degrees of Freedom
Dshell	DARTS Shell
ERP	Error Reduction Parameter
FLU	Forward, Left, Up (Coordinate System)
FPS	Frames per Second
GLSL	OpenGL Shader Language
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HLSL	High-Level Shader Language
IMU	Inertial Measurement Unit
JSON	JavaScript Object Notation
LiDAR	Light Detection And Ranging

NASA	National Aeronautics and Space Administration
ODE	Open Dynamics Engine
OGRE	Object-Oriented Graphics Rendering Engine
PID	Process ID
PSUtil	Process and System Utilities
RGB-D	Red Green Blue – Depth (Image)
ROAMS	Rover Analysis, Modeling, and Simulation
ROS	Robot Operating System
RTAB-Map	Real-Time Appearance-Based Mapping
RUF	Right, Up, Forward (Coordinate System)
SDF	Simulation Description Format
SDK	Software Development Kit
SI	International System of Units
SLAM	Simultaneous Localization and Mapping
SMI	System Management Interface
UA SEQ	University of Alabama Science and Engineering Quad
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
URDF	Unified Robot Description Format
V-REP	Virtual Robot Experimentation Platform
YAML	YAML Ain’t Markup Language
XML	Extensible Markup Language

ACKNOWLEDGMENTS

I would like to acknowledge the friends, colleagues, and faculty who helped make this thesis a reality. Thank you to Ethan Brookes, Max Eastepp, Lauren Faris, and Leonardo Yang for helping get me started with robotics research and giving me advice along the way. I am also grateful for the advice given by my committee members Dr. Drew Taylor and Dr. Chris Crawford towards the completion of this thesis. I would also like to thank Dr. Kenneth Ricks for his leadership and guidance on this research and throughout my time at the University of Alabama.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1:
INTRODUCTION

Stories centered on mobile robotic automata have fascinated humanity for millennia [1]. Undoubtably inspired by these stories, modern autonomous mobile robotics research began with the development of the mobile robot Shakey at the Stanford Research Institute. Shakey, along with many of the research projects to follow, demonstrated independent and intelligent navigation within a large, indoor, laboratory environment [2]. Outdoor environments, which lack much of the structure found indoors, pose additional challenges for autonomous robots due to potentially large sizes, uneven terrain, and 3D obstacles [3]. As these challenges have begun to be addressed, interest in autonomous robotic systems for outdoor applications including security, guidance, transportation, and exploration has grown rapidly [4], [5]. The state of the art in this category is the Mars 2020 Perseverance rover. Perseverance is equipped with autonomous navigation and obstacle avoidance capabilities which it uses while collecting samples of Martian soil [6]. These samples are left on the Martian surface to be identified and retrieved by the upcoming Mars Sample Return Mission. This new task requires the development of even more advanced autonomous capabilities than that of Perseverance.

A major challenge being faced for autonomous navigation is that the iterative development and testing of these algorithms is both expensive and time consuming [7]. This is especially true of the aforementioned Martian rovers since the surface of Mars is so different to that of Earth. Testbeds for rovers including the Mars Exploration Rover have been developed and used extensively in the past, but they are often constrained in size making it impossible to

evaluate autonomous navigation to distant goals [8]. Terrestrial environments such as Mauna Kea and the Atacama Desert have been used to test Martian rovers [9], [10], however these still necessitate the construction of an expensive standalone prototype rover [11]. Furthermore, neither artificial testbeds nor terrestrial environments are adequate for gathering the large amount and variety of labeled training data required to take advantage of modern machine learning techniques [7]. A promising way to address the expense, size limitations, and data collection issues associated with developing and testing autonomous rovers is to use a high-fidelity computer simulated environment. Currently, several robot simulation suites exist each with their own benefits in terms of physical realism, visual realism, simulation speed, and industry support. Using the popular Robot Operating System (ROS) as a robot control backbone, this thesis evaluates and compares the Gazebo simulator and the Unity3D game engine on their ability to simulate mobile, ground-based, wheeled rovers.

1.1. Rover Simulation

Robotics simulators could not exist until computer robot control and computer physics simulations were both possible. One of the first computer physics simulators developed for testing spacecraft software was DARTS, or Dynamics Algorithms for Real-Time Simulation. DARTS was created to be capable of modeling articulated multibody dynamic systems. The generalized modeling within DARTS gave it the flexibility to be used as the simulator for several projects [12]. As more projects began using DARTS, it became clear that a significant development effort was expended creating the same sensors on different projects and creating interfaces between the simulator and controller software. Thus, Dshell, or DARTS Shell was created to integrate the DARTS simulator with a collection of actuator and sensor models. As a result, Dshell was more easily configured with flight hardware and software across a variety of

missions including usage in the development of the Galileo and Cassini missions [13], [14]. ROAMS, or Rover Analysis, Modeling, and Simulation, was developed as a rover specific extension of Dshell. The simulator was used as a real-time testing environment for a rover's subsystems including mechanical, electrical, sensing, and control subsystems. A key feature of ROAMS is its modular architecture which allows vehicles, environments, and navigation strategies to be defined and reused across multiple simulations. One of the first applications of ROAMS was simulating a model of the Rocky 7 rover testbed traversing Mars-like terrain before aiding in the development of the Mars Pathfinder and Mars Science Laboratory rovers [15], [16].

More recently, growth around these tools has continued but at a slower pace. In [17] Lim and Jain propose Dshell++, a reimplementation and expansion of Dshell in C++ using object-oriented programming. The reimplementation was structured around increasing user productivity by facilitating reuse and by using a data-flow architecture managed by Python allowing a dynamic runtime interface. The use of Python interfaces simplifies simulation set up and control while the underlying C++ supports real-time simulation [17]. Separately, Verma and Leger developed SSim, a robotics simulator utilizing ROAMS for planning during a rover's post landing operations phase. As a result, SSim interfaces with finalized software code initialized with the current rover state, allowing faithful simulation of actions planned for the real rover. To do so, SSim has the capability of using several simulation modules alongside ROAMS such as the MSC Automated Dynamic Analysis of Mechanical Systems (ADAMS) solver or the open-source Gazebo 3D rigid body simulator for robots. According to the authors, SSim is anticipated to be used during the Mars 2020 mission to simulate all surface operations, including instrumentation, power, thermals, and communications [18].

Despite these developments, neither ROAMS, Dshell++, nor SSim have become widely used among the research community outside of NASA. Instead, autonomous rover researchers have coalesced around several open-source robotic simulation software suites. One early suite that quickly gained popularity was the Player Project. The Player Project began at the University of Southern California Robotics Research Lab in 1999 as a tool for interfacing with and simulating multirobot systems. In its original design, the Player Project consisted of two components: the robot control server referred to as Player and the two-dimensional multirobot simulator referred to as Stage. Since then, it has continued to be adopted and improved by researchers around the world [19]. One such improvement was Gazebo, which began development in early 2002 and was released in 2003. Gazebo was conceived as a high-fidelity three-dimensional (3D) outdoor simulator, complementary to the existing Stage simulator. In 2009 support for the Robot Operating System (ROS) was integrated and by 2012 the Open Source Robotics Foundation became the primary stewards of both ROS and Gazebo, giving rise to the ROS-Gazebo robotic control and simulation suite [20].

1.2. ROS-Gazebo

The Robot Operating System (ROS) is an open-source robotics development framework. It was designed in the late 2000s at Stanford University to overcome the challenges encountered when developing large-scale service robots. The ROS framework is a collection of tools used to create, connect, visualize, and run pieces of software for robotics in addition to a repository of reusable implementations of common robot behaviors [21]. Fundamentally, a robot architecture within ROS is a collection of modular behaviors connected as nodes in a graph. Each node is its own computing process that communicates with other nodes via messages. Nodes send messages by publishing them to a topic and nodes subscribe to a topic to receive new messages from the

topic as they are published [21]. This results in robot designs with a high degree of modularity such that any given node can be substituted for one that publishes and, if necessary, subscribes to the same topics. This also allows developers to update existing behaviors and create all new behaviors in a way that they can be easily shared and integrated into other projects. This continuous improvement and ease of integrating updates is a large part of what makes ROS so popular among the robotics community.

Gazebo, a high-fidelity three-dimensional robot simulator, also benefits tremendously from the modularity of ROS. In addition to providing its own tools for creating robots [22], Gazebo supports importing robots from ROS native unified robot description format (URDF) files [23]. As a result of the modular architecture of ROS robots, sensors simulated in Gazebo can publish to the same topics as the sensors on the real robot. To do so, Gazebo simulates real sensors and their interactions with the environment to generate data like what would be captured in the real-world. Similarly, simulated actuators within Gazebo can subscribe to the topics used by the actuators in a real robot. Thus, a robot within Gazebo can be controlled via ROS, and Gazebo can publish robot sensor messages to ROS, provided the appropriate packages and topics are used [23]. This allows the same ROS architecture to be reused between simulation and the real-world, speeding up development and making robotics more accessible.

Besides robots and sensors, Gazebo also supports the creation of virtual environments. Both indoor and outdoor environments can be constructed from a variety of sample objects included with Gazebo or by importing custom objects and scenes [24]. Physical interactions between the robot and environment are simulated using one of several performant physics engines, by default the Open Dynamics Engine (ODE). Both the environment and robot in Gazebo are rendered using the Object-Oriented Graphics Rendering Engine (OGRE) [20]. With

a foundation of open-source software and a powerful set of capabilities, the ROS-Gazebo robotics simulation suite has been used with success not only in terrestrial robotics, but also in recent extraterrestrial rover research.

Recently, ROS-Gazebo was used to develop a lunar rover driving simulator in a collaboration between the NASA Ames Research Center and the Open Source Robotics Foundation. One goal of the simulator was to produce synthetic images suitable for evaluating human drivers and computer vision algorithms, making high fidelity visuals a priority. The simulated environment was based on a large randomly selected region of one of the lunar poles. Despite the powerful capabilities of ROS-Gazebo, significant modifications to the Gazebo renderer were required. The authors implemented improvements to the Gazebo terrain rendering, shader, shadow mapping algorithm, and camera settings. These modifications addressed issues the authors faced including poor performance with large terrains and the presence of visual artifacts from the default shadow mapping algorithm used by Gazebo [25]. The final simulator was successfully used to test rover navigation algorithms, perception algorithms, operation concepts, and configurations showing the promise of simulation and ROS-Gazebo for extraterrestrial rovers. Despite this, the large number of significant modifications suggests that ROS-Gazebo lacks some of the features required for this application.

In another study, ROS-Gazebo was used to simulate a rover to test several perception and navigation approaches in a simulated Martian environment. The authors selected the Gazebo simulator because of its tight integration with ROS. Compared to other robot software architectures, ROS benefits from existing navigation and perception implementations as well as the ability to use a common platform between simulated and real robots. The real six-wheeled MORPEUS rover has stereo cameras and a 3D LiDAR sensor, but it is simulated using a

simplified model for performance reasons. Results indicated that both visual and LiDAR-based localization and mapping algorithms performed well, though LiDAR benefitted from more detailed maps in the form of point clouds with the tradeoff of greater localization error. The authors note that one goal moving forward is to enhance the photorealism of the simulation [26].

Clearly, despite the powerful capabilities of Gazebo, there is still desire within the research community for simulation suites with improved visual fidelity. One approach, pursued in [25] is to invest significant time and resources into improving the visual fidelity of Gazebo. Another approach being pursued by an increasing number of researchers is to use an alternative high visual fidelity renderer. Thus, many have begun to turn their attention towards the usage of game engines for robotic vehicle simulation due to their higher visual fidelity and developer friendly architectures [27]. One such popular game engine is Unity3D.

1.3. Unity3D

Commercially released in 2005, Unity3D was one of the first freely available game engines playing a major role in the democratization of game development [28]. Today, Unity3D uses a powerful physics engine and produces high fidelity visuals, even finding use by film studios [29]. As a game engine, Unity3D is not only software that models physics, simulates interactions, and renders graphics, it is also an integrated development environment for games and visualization applications. As a tool in its development environment, Unity3D supports a scripting system. Scripts can be attached to any object to modify its behavior, ideal for controlling the segments of a character or robot [27]. Scripts can also be attached to the editor itself enabling new functionality, for example allowing new object file types to be imported. Another of the resources in the Unity3D development ecosystem is the Unity Asset Store, which allows scripts, objects, and environments to be easily shared [30]. This allows detailed

environments and other assets to be reused from external projects, accelerating development [31]. Many of the features of Unity3D intended for the development of video games like its powerful physics, graphics, and scripting engines are also desired in robot simulators.

Mattingly et al. presented one of the first robot design and simulation systems using Unity3D. The authors note that Unity3D includes many of the functions required for robotic simulation including environment editing, collision detection, and rigid body dynamics while benefiting from high flexibility and ease of use. As a technical demonstration, a humanoid robot and an industrial robot arm are modeled in Unity3D using the proposed design system. The design system consists of a library of material, mesh, skeletal, animation, and script assets intended to simplify the creation of new robots in Unity3D [27]. Since then, others have continued to refine the practice of using Unity3D as a simulator.

A 2016 collaboration between researchers at Vanderbilt University and Toyota used a custom composite simulator, including Unity3D, to model a vehicle with an advanced driver assistance system (ADAS), a form of robotic autonomous control. Specifically, the composite simulator used Dymola to model vehicle physical components such as the engine, Simulink to model the ADAS, Unity3D to simulate the ground-wheel interaction and visualize the vehicle, and OpenMETA, a custom software bridge developed at Vanderbilt University, to integrate the components with one another. According to the authors, the Unity3D game engine was used for several reasons including its ease-of-use, 3D physics engine, scripting engine, ability to import 3D models, large asset library, and large and supportive user base. Within the Unity3D environment, assets from the Unity Asset Store were used to build the road and city models. The final simulator was validated by using it to constrain the parameter values of an automatic cruise

control system [31]. The success of the validation experiment demonstrated the feasibility of a Unity3D-based simulator in autonomous robotic system design.

Whereas the previously presented simulator focused on physical accuracy, Niemirepo, Toivonen, Viitanen, and Vanne presented a Unity3D-based simulator with the goal of photorealism for the purpose of developing vision based ADAS. To do so, a pre-made city environment from the Unity Asset Store was augmented with driving vehicles, pedestrians, weather effects, and lighting. Vehicles and pedestrians were scripted to follow predefined paths and avoid collisions. Weather effects and lighting were implemented by modifying the global illumination, changing the skybox, adding particle effects, and adding shaders to the camera view. To capture the images used in developing the vision based ADAS, cameras could be added inside the vehicles or elsewhere as needed. The presented Unity3D-based simulator was user-friendly, featured diverse environmental conditions, and could maintain high-quality graphics during real-time simulation [32]. This confirms that one of the benefits of a Unity3D-based simulator is high-fidelity visuals, making simulated evaluation of vision-based robotic autonomy systems feasible.

Despite the presented benefits of Unity3D as a robotics simulator, it lacks many of the tools used for developing robot software. While it is possible to develop robotic control systems entirely within the Unity3D scripting engine, this would likely lead to a difficult to maintain monolithic architecture and would not be easily transferable from simulation to a real robot. Thus, it is desirable to integrate Unity3D as a simulator with a robotics development framework. Due to the benefits discussed above, ROS is selected as the robotics development framework to integrate with Unity3D, yielding the ROS-Unity3D robotics simulation suite.

1.4. Contribution

This thesis provides an overview of the ROS-Unity3D robotics simulation suite and an analysis of its capabilities with respect to the simulation of wheeled ground rovers. ROS and Unity3D will be connected using the ROS-Unity Integration plugin published by Unity3D. The ROS-Unity3D simulation suite will also be compared to the ROS-Gazebo simulation suite. Both simulators are used to simulate the Clearpath Robotics Jackal unmanned ground vehicle (UGV) in outdoor unstructured environments performing LiDAR and visual-based autonomous navigation tasks. Specifically, the simulators will be compared on their theoretical architectures, computer resource usage, and accuracy of the results produced by the simulated autonomous robots.

1.5. Outline

Chapter 2 provides an overview of existing research in connecting ROS and Unity3D along with any comparisons made to the ROS-Gazebo simulation suite. Chapter 3 details the proposed ROS-Unity3D and ROS-Gazebo simulation architectures. Chapter 4 explains the process of creating a robot and simulation world in Unity3D and compares it to the equivalent processes in ROS-Gazebo. Chapter 5 describes the test scenarios used to evaluate ROS-Unity3D and ROS-Gazebo. Chapter 6 presents and explains the results from the test scenarios. Chapter 7 contains concluding remarks and possible areas for future work.

CHAPTER 2:
RELATED WORKS

2.1. ROS-Unity3D

One of the first published connections between ROS and Unity3D was presented in 2014. The connection and resulting ROS-Unity3D suite was developed as an immersive user interface for robotic teleoperation. Specifically, a ROS-based ground contact robot was driven by the user through a virtual reality teleoperation interface depicting the robot in its environment implemented in Unity3D. As in many other works, the de facto standard robotic middleware, ROS, was used because of its flexibility, modularity, compatibility, and popularity. As a mature game engine and virtual reality environment, Unity3D was used because it supports many operating systems, it has a powerful scripting engine, and it supports external virtual reality hardware such as headsets and controllers. The interface between ROS and Unity3D was a custom design using the rosbridge framework to send and received YAML-encoded messages through a network connection which interfaces with a custom Unity3D script. The authors note that while this method has a limited throughput and would not allow sending full point clouds, it was sufficient for simple commands. Unity3D received commands from the driver via a controller. The commands were then sent to ROS and used to control the physical robot. As the physical robot moved, new sensor data was collected and used to obtain a new pose estimate. The estimate was sent back to Unity3D to update the position of the virtual robot within the existing environment [33]. This work pioneered the connection between ROS and Unity3D and laid the foundation for subsequent research.

Just over a year later, Meng et al. published a ROS-Unity3D-based navigation and control simulator [34]. Though intended for unmanned aerial vehicles (UAVs), the simulator addressed several robot-agnostic requirements, including sensor modeling, which had not previously been done in Unity3D. The authors chose to create a new simulation suite over existing alternatives including Virtual Robot Experimentation Platform (V-REP) and Gazebo because they do not achieve a high enough level of realism. The Unity3D game engine is selected as the visualization tool primarily because it is beginner friendly. Again, the interface between ROS and Unity3D was implemented by the authors using a custom ROS node, a network connection, and a Unity3D script. The ROS node sent trajectory commands to Unity3D to control the movement of the simulated UAV while the Unity3D script sent UAV LiDAR sensor data and state transform data to ROS. The LiDAR is modeled in a Unity3D script using raycasts to determine the distance from the sensor to the nearest object at each angle within the scanning range. As a demonstration, a UAV flight through a difficult forest environment that could not be accurately modeled in other simulators was simulated [34]. This work was pivotal for introducing sensor modelling from within Unity3D and creating an early ROS-Unity3D robotic simulation suite.

Mizuchi and Inamura expanded upon these ideas with their ROS and Unity3D-based human-robot interaction simulator. The proposed simulator was designed around allowing users to experience a Unity3D virtual environment populated by simulated ROS-based robots and to study the interactions between users and robots. To allow the simulated robots to perceive the virtual environment, camera and depth sensors were modeled within Unity3D. After analyzing several previously proposed systems for bridging ROS and Unity3D, it was determined that no existing interface had a high enough bandwidth to transfer camera images in real-time.

Consequently, the rosbridge framework discussed in [33] was modified to have the ability to transmit JavaScript object notation (JSON) messages as well as binary JSON (BSON) messages resulting in less encoding time and smaller message sizes. It was found that 57.6 frames per second (FPS) of RGB-D images could be transmitted using BSON messages. This rate was sufficient for real-time sensor feedback and represented an over 100 times improvement compared to the unmodified implementation. The simulation suite was evaluated by comparing the paths taken by a real ground robot programmed to follow a real human and a simulated ground robot following a virtual human constructed from motion capture of the real human. The motion of the real and virtual robots only differed slightly, indicating that the simulated environment was highly accurate [35]. With respect to the development of ROS-Unity3D robot simulation suites, the major contributions of this work were the implementation of simulated camera sensors, the improvement of the interface for real-time performance, and a demonstration showing the high accuracy of a ROS-Unity3D-based simulator.

Taking advantage of simulated camera sensors within Unity3D, Zhang and Zhao created URCV, a virtual testing system for computer vision algorithms based on ROS and Unity3D. In URCV, a simulated camera sensor was moved along a trajectory through a Unity3D environment while images from the sensor were published to ROS. Unity3D was used over alternatives like Gazebo because of the variety of ready-made models in the Unity Asset Store and the powerful capabilities of the Unity3D editor for building high-complexity and detailed virtual environments. Within ROS, a monocular simultaneous localization and mapping (SLAM) algorithm produced an estimated trajectory from the sequence of images, which was compared to the ground truth trajectory to evaluate the accuracy of the algorithm. URCV used ROS# to interface ROS and Unity3D [36]. ROS# is a standardized open-source set of libraries and tools

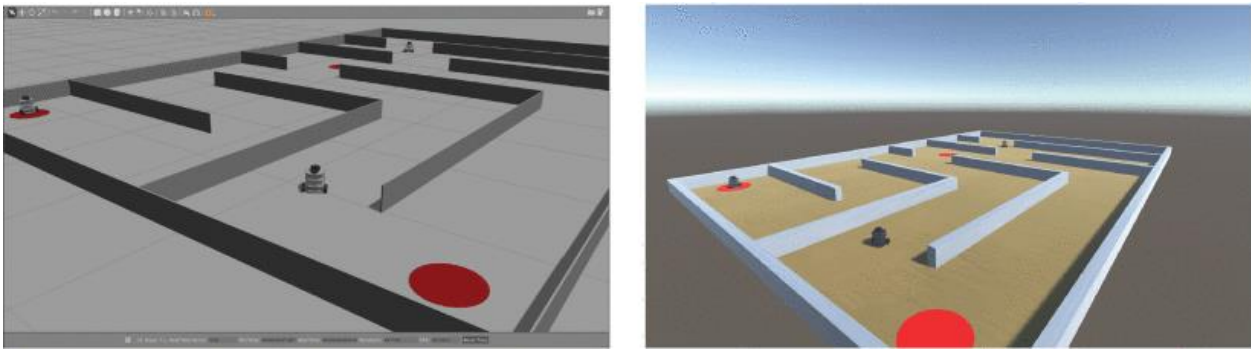
for interfacing ROS with Unity3D, among other .NET applications, maintained by Siemens since 2017 [37]. The ROS# interface is like the previously developed ROS-Unity3D interfaces such as rosbridge, however its wider base of users and contributors has enabled it to become more optimized necessitating fewer modifications by researchers. In URCV, images are sent from the Unity3D camera sensor through ROS# to ROS at 30 FPS. To demonstrate the feasibility of URCV, the ORB_SLAM2 algorithm was tested in indoor, urban, and suburban Unity3D environments and compared to the results from a standard real-world dataset. The experiments showed that the SLAM algorithm could run successfully using images from a ROS-Unity3D simulated environment and achieve accuracies equivalent to those seen using a real-world dataset [36]. This research further validated that camera sensors can be modeled in real-time in a ROS-Unity3D simulator and showed that the graphics are of suitable fidelity for accurately testing a vision-based SLAM algorithm.

2.2. Comparisons Between ROS-Gazebo and ROS-Unity3D

Evidently, several research groups have created and evaluated ROS-Unity3D-based robotics simulators as an alternative to the ROS-Gazebo simulation suite. Many have suggested that Unity3D offers several advantages over Gazebo including that it supports more operating systems, it supports more external hardware, it is more beginner friendly, and larger, more detailed environments can be constructed, but thus far these advantages have only been mentioned in passing with little, if any, concrete evidence for these claims. In response, there have been a few attempts at creating identical virtual environments in both ROS-Gazebo and ROS-Unity3D to better assess the capabilities and advantages of each simulation suite.

Hussein et al. developed a ROS-Unity3D-based simulator for intelligent vehicles and compared it to a Gazebo-based simulator in 2018. To do so, a test scenario consisting of several

wheeled ground robots performing SLAM using onboard lidar sensors in a 6 x 9 m environment was constructed in both Unity3D and Gazebo [38]. The Gazebo-based simulation was connected to ROS using the capabilities built into Gazebo while the Unity3D-based simulator was connected to ROS using rosbridge as in [33] and [35]. The environments used for the comparison are shown in Figure 1 below with the Gazebo-based simulation on the left and the Unity3D-based simulation on the right. Both environments are based on the same maze structure, contain the same robot, and have similar shadow positions, however the Unity3D-based simulator uses more highly detailed textures, for example the floor material. Aside from the apparent visual differences, no comparative analysis between the simulators is presented by the authors. The Unity3D-based simulator was also used to successfully simulate a vehicle driving 2.6 kilometers in Vienna to evaluate an ADAS with little deviation from the expected result, however this scenario was not evaluated in the Gazebo-based simulator [38]. In summary, this work showed that comparisons between ROS-Unity3D and ROS-Gazebo simulators using equivalent environments are possible, setting the groundwork for future comparisons.



(a) (b)

Figure 1. Robotic simulation environments constructed in (a) Gazebo and (b) Unity3D [38].

More recently, a dedicated analysis and comparison of several 3D robotics simulators, including Unity3D and Gazebo, was published. To begin, the popularity of the simulators within the robotics research community was determined by tallying the number of works referencing them on Google Scholar. Among the top ten robotics simulators, the most highly referenced was Gazebo followed closely by Unity3D then V-REP, while the remaining seven had significantly fewer references. Subsequently, Gazebo, Unity3D, and V-REP were analyzed according to their capabilities, as summarized in Table 1 below. As another point of comparison, the simulators were evaluated according to usability, as assessed by three expert researchers in robotic simulation, while installing each simulator and implementing a standardized 3D scene. The scene consisted of a keyboard controlled TurtleBot ground robot located inside a rectangular room. The authors concluded that even though Unity3D had the highest graphics quality and is typically used when simulation realism is desired, it was not recommended because it was the most difficult to use. Namely, while creating the Unity3D simulation suite the experts were unable to link it with ROS and the simulator seemed to produce unrealistic movement. Between Gazebo and V-REP, it was concluded that V-REP was easier to use because of its intuitive and powerful user interface and greater suite of native features [39]. Overall, this research expanded upon

TABLE 1. CAPABILITIES OF THE GAZEBO, UNITY3D, AND V-REP SIMULATORS

	Gazebo	**Unity3D**	**V-REP**
ROS Compatibility	Native support for ROS	Supported via plugins such as rosbridge and ROS#	Supported via V-REP API
Supported Physics Engines	Bullet, ODE, Vortex, Newton	PhysX	Bullet, ODE, Sim-body, DART
URDF Import	Supported	Supported	Supported
URDF Export	Unsupported	Supported	Unsupported
Supported Programming Languages	C++	C#	C++, Python, Java, MATLAB, Lua
Software License	Open-Source (Apache 2.0)	Free for entities with annual revenue under $100,000	Free for education applications

earlier comparisons between Gazebo and Unity3D in equivalent environments while concisely assessing capabilities and evaluating usability. The authors conclusions are valuable, but additional points of comparison between the simulators including physics accuracy, performance, maximum environment size, and usability with computer vision algorithms are important to consider for autonomous ground robot simulation.

Since then, Konrad published a detailed overview of mobile robot simulation in ROS-Unity3D along with a comparison to ROS-Gazebo [40]. The comparison primarily focused on the differences between the PhysX and ODE physics engines used by Unity3D and Gazebo, respectively. Several virtual and real experiments involving a TurtleBot2 robot driving up slopes, into walls, and along predefined paths were conducted to assess the collision, integration, and friction accuracy of the simulators. It was concluded that Unity3D can reproduce the basic physical behavior of the real robot with respect to friction and slopes while Gazebo tends to differ from the behavior of the real robot by modeling ideal conditions. Having confirmed the physical accuracy of Unity3D, a case study on the ability of ROS-Unity3D to simulate a mobile robot performing SLAM was conducted. The robot used simulated LiDAR, inertial measurement unit (IMU), and wheel encoder sensors during the SLAM experiment. After overcoming challenges related to mismatched simulation times, transform trees, and laser rotation, a satisfactory result was obtained as shown below in Figure 2. The case study was not conducted in an equivalent ROS-Gazebo simulation suite for comparison [40]. This research validated the accuracy of the ROS-Unity3D simulation suite, showed that the realism of ROS-Unity3D can exceed that of ROS-Gazebo under certain tests, and showcased a case study modeling an autonomous ground robot performing SLAM in ROS-Unity3D.

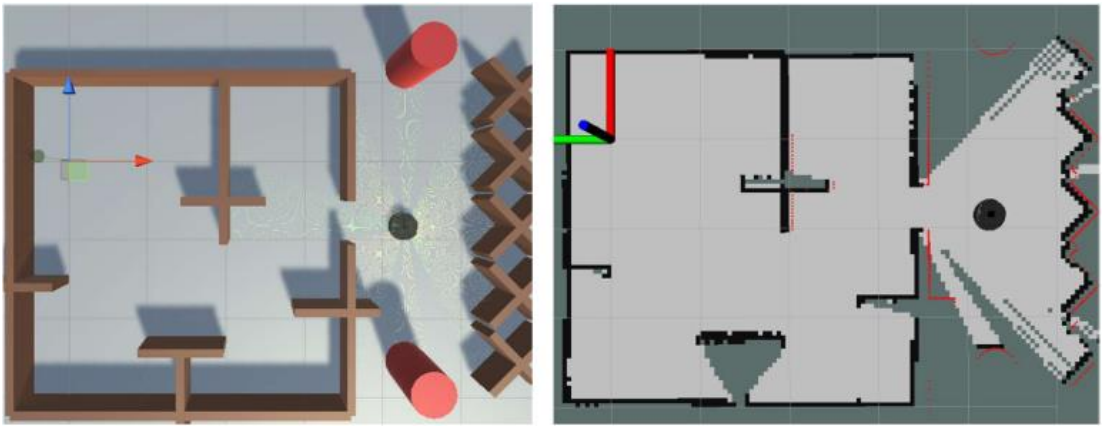


Figure 2. Laser-based SLAM simulation in Unity3D [40].

Existing comparisons between ROS-Unity3D and ROS-Gazebo simulators have thoroughly explored the similarities and differences between their basic visuals, features, ease-of-use, and physics engines. Researchers have also presented mobile robots with advanced capabilities, including LiDAR, computer vision, SLAM, and autonomous navigation, simulated in either ROS-Unity3D or ROS-Gazebo. Yet to the best of this author's knowledge, there are no existing comparisons between ROS-Unity3D and ROS-Gazebo considering the same robot platform in the same simulated environment performing advanced autonomous navigation capabilities. The prevalence of computer vision systems in modern autonomous robots also makes it important to analyze the accuracy of the visual environments as modeled by both Unity3D and Gazebo as well as their impact on autonomous navigation systems. Furthermore, the desire for large training sets also makes it important to consider the performance of the simulators, especially when it comes to large virtual environments. Finally, the release of a first party ROS-Unity3D integration published by Unity Technologies in 2021 [41] makes it important to determine if any capabilities have changed since the publications of earlier ROS-Unity3D simulators based on ROS# or rosbridge. Thus, this thesis continues in the following section with a description of the ROS-Unity3D architecture used in the comparison testing against the ROS-Gazebo simulator.

2.3. Combination Architectures

Several groups have investigated coupled ROS-Gazebo-Unity3D environments with some success [42]–[45]. Typically, the ROS-Gazebo-Unity3D environment is structured such that the physics simulations are performed in Gazebo while the visuals are captured in Unity3D with the position of the robot mediated through ROS. This approach allows most robots sensors to be modeled in Gazebo while camera images are captured in Unity3D. Using camera sensors modeled in Unity3D is needed because Gazebo visuals are not sufficiently realistic for computer vision applications, such as transfer learning in deep vision networks [46]. In this architecture, Unity3D is also often used to integrate virtual reality headsets which can provide a more powerful user interface. The issue with a ROS-Gazebo-Unity3D approach is that it increases computation requirements and introduces noticeable communication delays [45]. Additional coordination must also be performed to ensure the environment the robot interacts with in Gazebo is identical to that which it sees in Unity3D, which significantly increases development effort. Lastly, as discussed in [40] the Unity3D physics engine is satisfactory for robot simulation and in some cases is more realistic than the Gazebo physics engine, meaning it is not necessarily beneficial to use Gazebo. In summary, a ROS-Gazebo-Unity3D architecture is not investigated in this thesis because of the inherent performance penalties, the increased development effort, and the lack of clear benefit towards using the Gazebo physics engine when the Unity3D physics engine is also available. Though these are not the only simulation suites currently in use, this research is not extended to software outside of ROS, Gazebo, and Unity3D because of an intentional decision to limit the scope to the most popular simulators. Therefore, the goal of this thesis is to specifically compare the ROS-Unity3D and ROS-Gazebo robot simulation suites.

CHAPTER 3:
COMPARISON OF SIMULATION ARCHITECTURES

In this chapter, the architectures of the ROS-Unity3D and ROS-Gazebo simulators will be discussed. Primarily, the simulators are described and compared in terms of the underlying Unity3D and Gazebo components. This thesis considers Unity3D version 2022.1 and Gazebo version 11, both of which are the latest versions at the time of this research. Unity3D and Gazebo are analyzed in terms of simulation hierarchy, coordinate system, time keeping, physics, model compatibility, and user interface. These sections are not specific to any robotic platform or even the application of robot simulation itself, however they are relevant to the development of the ground robot simulation suite proposed in this thesis. Specific to robotics applications, the final section describes the connection to ROS used in the ROS-Unity3D and ROS-Gazebo simulation suites. Here, the ROS Noetic release is considered since it is the newest release of ROS 1. ROS 1 is used in this thesis over newer versions of ROS 2 because of its proven stability and reliability [36].

3.1. Simulation Hierarchy

The structure and organization of simulated objects within Unity3D and Gazebo are important for understanding how each simulation is constructed. Both simulators use a hierarchical structure where each simulated object has exactly one parent but may have zero or more child objects. Each object can also have several properties which can be inherited by child objects, depending on the property. The objects, properties, and levels of the hierarchy available differ between Unity3D and Gazebo, as do the terms used to refer to them.

3.1.1 Unity3D

The highest level of the hierarchy in Unity3D is the project. Multiple projects can exist on a computer, each located in its own directory independent of other projects. Since multiple versions of Unity3D can be installed simultaneously, projects can be created using a specific version of Unity3D, though it is typically not advised to change the version of Unity3D a project uses after it has been created. Aside from the version of Unity3D being used, there are also several properties that are configured at the project-level. These project-level properties include settings related to physics, the player, overall quality, and timing [47], many of which will be discussed in the following sections. Each project can contain one or more scenes, the next level down in the hierarchy.

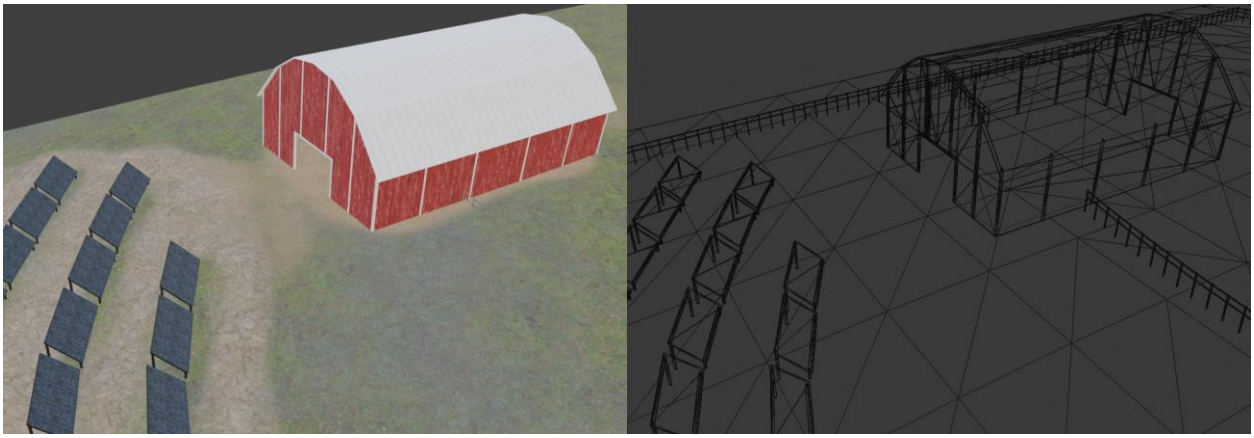
Each Unity3D scene is the root of an environment; in the context of a game engine this means scenes are typically used as game levels and menus [27]. Using Unity3D as a simulator, a scene is a single simulation scenario containing objects representing the terrain, obstacles, lighting, and any other simulated entities, including the ClearPath Robotics Jackal platform used in this thesis. Scenes themselves also have several properties independent of the properties of descendant objects and the project-wide properties, including global illumination settings, the texture of the sky, the look of reflections, and the presence and appearance of fog in the environment. The aforementioned objects are descendants of the scene and are specifically referred to as “GameObjects”. Scenes can contain any number of GameObject children.

GameObjects are considered the most important concept in Unity3D [48]. They are fundamental objects that represent nearly everything in the game or simulation. GameObjects can have any number of GameObject children. The parent of a GameObject is either another GameObject or a scene. GameObjects themselves have no appearance or impact on the

simulation, instead they act as containers for components which implement functionality. All GameObjects contain a single Transform component which specifies position, rotation, and scale of the GameObject relative to its parent. GameObjects can contain any number of components, including multiple versions of the same component, apart from the Transform component of which there is always exactly one.

Three common types of components are renderers, colliders, and bodies. Renderers give a GameObject a visual appearance. As in most graphics software, 3D objects in Unity3D are approximated by polyhedrons, referred to as meshes because of their mesh-like appearance when displaying only their edges and vertices as shown in Figure 3. As a result, mesh filter and mesh renderer components are added to a GameObject to display a 3D object in Unity3D. The mesh filter is used to specify the mesh to be displayed while the mesh renderer displays the mesh at the location of the GameObject. Mesh renderers can be configured to display a texture on the mesh using a material and to cast and receive shadows from other 3D objects in the scene. Colliders give Unity3D GameObjects the ability to detect and process interactions between touching or intersecting objects. For example, adding a collider component to the barn and the Jackal robot in the agriculture environment in Figure 3 prevents the robot from passing through the walls. Colliders have a configurable physics material for setting friction and bounciness properties. Simple objects can use basic colliders such as box colliders and sphere colliders to detect collisions using simple shapes while more complex objects use mesh colliders to detect collisions with a specified mesh with the tradeoff of increased computational load. Lastly, bodies are used to cause GameObjects to be affected by the physics simulation. The two types of bodies available are “Rigidbodies” and “Articulation Bodies”. Adding a Rigidbody component to a GameObject will cause it to be pulled downwards by gravity, affected by forces and torques, and

will cause it to react to collisions with incoming objects, provided that a collider component has also been added to the GameObject [49]. A Rigidbody component has a configurable mass, center of mass, drag, and angular drag to facilitate realistic movement within the simulated environment. Articulation Bodies are similar to Rigidbodies with the addition of the ability to constrain the relative movement of GameObjects [50]. For example, the simulated Jackal robot and its wheels are composed of GameObjects with Articulation Bodies to ensure that the rotation of the wheels is to the axis of the axle. Additional components beyond the renderers, colliders, and bodies included in Unity3D such as cameras and lights are typically used for more specific needs and will be discussed in more detail as needed.



(a) (b)

Figure 3. The agriculture environment shown as a (a) 3D object and (b) mesh.

Aside from the built-in components, custom components can also be created within Unity3D using scripts. Unity3D supports scripts written in C# directly as well as other .NET languages that can be compiled into a compatible dynamic link library (DLL) [51]. Like the built-in components, custom components must be attached to a GameObject to function. Scripts interface with Unity3D by inheriting from a built-in class called “MonoBehavior”. This inheritance allows properties of the component’s GameObject as well as the properties of other components attached to the GameObject to be accessed and modified. For example, a simple

custom script could update a GameObject's Transform component to teleport the GameObject. For more realistic movement, a custom script could apply a force to a GameObject and allow the physics simulation system to move the object based on the force applied. Scripts can only perform actions at certain times dictated by the callback functions inherited from the MonoBehavior class, most commonly using the Start, Update, and FixedUpdate functions. The Start function is called for each custom component by Unity3D once when the script is first initialized. The Update function is called for each custom component by Unity3D every simulation rendering step. The FixedUpdate function is called for each custom component by Unity3D at a fixed rate relative to the simulation time used by the physics system. This rate is configured as a project-level property [52]. The specific differences between the frequency of execution between the Update and FixedUpdate functions is discussed in the time section. An example Unity3D simulation hierarchy is shown in Figure 4.

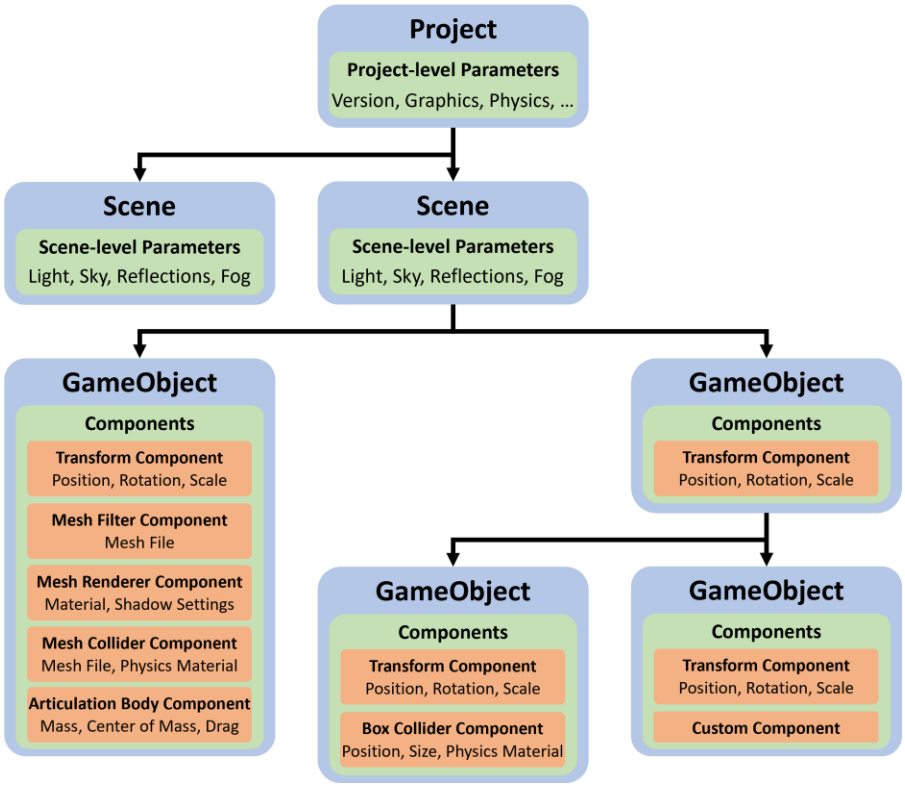


Figure 4. A sample Unity3D simulation hierarchy.

3.1.2 Gazebo

Gazebo simulation worlds, including robots, sensors, lights, and other objects are fully defined using Simulation Description Format (SDF) files [53]. Since SDF files define the Gazebo simulation world, they also dictate the general structure of the object hierarchy. Outside of the simulation hierarchy, multiple SDF files can be easily created and modified. One benefit of the usage of SDF files is their ability to be dynamically inserted within other SDF files using an include element, facilitating a modular design architecture. Within the hierarchy, the highest level and thus root element of an SDF file is the SDF element. At this level, the only configurable parameter is the version of the SDF specification used in the file. The SDF element can contain multiple of several different kinds of child elements; however it is most common for the SDF element to contain a single world element child [54].

A world element is an environment in which models can be created and simulated, somewhat analogous to a scene in Unity3D. Many of the simulation properties are configured at the world level via child elements including the graphical user interface (GUI), scene, physics, atmosphere, and wind. The GUI element contains properties related to the positioning, tracking, and draw distance of the simulated camera used for displaying the simulation to the user. The scene element contains properties related to the global illumination, the texture and color of the sky, and the presence and appearance of fog in the environment. The physics element contains properties related to gravity and the physics engine in use, described in more detail in the physics section. The atmosphere and wind elements define the temperature, pressure, and density of the atmosphere and environmental windspeed, respectively. World elements can only contain one of each of the prior elements to avoid conflicting parameters. Beyond simulation properties, the world element can also contain multiple light elements and model elements [55].

Light elements each define a singular light source. Light sources have several properties configurable via child elements including intensity, diffuse light color, specular light color, attenuation, and position. Like GameObjects in Unity3D, Gazebo model elements represent nearly everything in the simulation. Model elements have several important child elements, one of which is other model elements, enabling nested models. Other child elements include those related to parameters such as model position and if the model should be considered as static [55]. Interestingly, Gazebo defaults to assuming that all models are dynamic such that the terrain must be specifically notated as static while Unity3D defaults to assuming that all GameObjects are static unless specifically created with a Rigidbody or Articulation Body component.

The final two significant children of model elements are link elements and joint elements. A link element represents a physically simulated part of a model with inertia, collision, and visual properties. As a result, a link element is similar to a Unity3D GameObject with renderer, collider, and body components. Inertial properties are specified in an inertial element in terms of the mass, center of mass, and rotational inertia. Collisions are described in the collision element with sub-elements for specifying the relative pose of the collider, the collision geometry, and the material properties of the collision interface. Likewise, visual properties are described in a visual element containing sub-elements for specifying the pose, material, and geometry. Both the collider geometry and visual geometry can be specified as simple shapes such as boxes and spheres or as meshes. Within a model element, joint elements are used to connect link elements with kinematic and dynamic properties, like how Articulation Bodies can be used in Unity3D. Joint elements require that the type of joint be specified and contain sub-elements for defining the two links to be joined and the properties of the joint movement axes [55].

Link and joint elements may also contain sensor elements which are used to measure and publish information measured from within the simulated world. Sensor elements can be configured to simulate several different types of sensors including cameras, torque sensors, LiDARs, and IMUs. Configurable sub-elements of the sensor element include a pose element, an update rate element, and a topic element which is used to configure where the sensor data is published [55]. As an element specifically required in robot simulation, Gazebo sensor elements have no analogous object or component built in to Unity3D. It should be noted that it is possible to replicate the functionality of sensor elements using custom scripts in Unity3D.

To implement additional functionality outside of that built into Gazebo, the SDF specification defines plugin elements which can be the child of world, model, sensor, visual, and GUI elements. Gazebo plugin elements refer to a custom shared library that is inserted into the simulation in the context of its parent element. Gazebo plugins are typically written in C++ and must be compiled as a shared library against the development libraries for the version of Gazebo in use. Plugins interface with Gazebo though the C++ "gazebo" namespace, by inheriting from the plugin class corresponding to the type of the parent element, and by registering the plugin with the simulator. These steps allow plugins to control almost any aspect of Gazebo from controlling the world properties like the physics engine, to controlling joints and model state, to interacting with sensors [56]. Unlike Unity3D scripts, Gazebo plugins are called once when initialized, loaded, and reset, but are not regularly called during the simulation. Therefore, plugin authors are responsible for manually configuring the plugin to listen to world update events [57]. A sample Gazebo simulation hierarchy is shown in Figure 5.

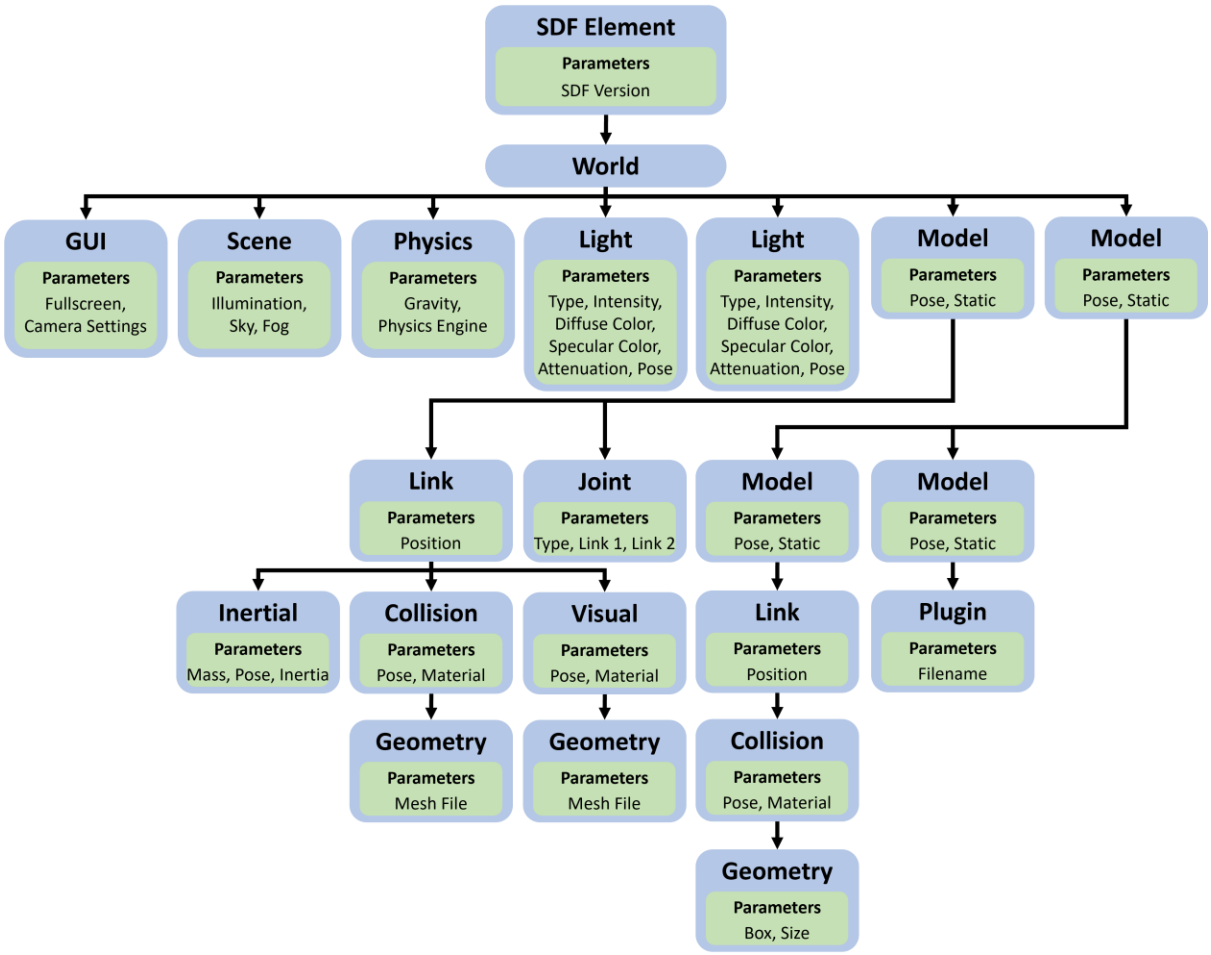


Figure 5. A sample Gazebo simulation hierarchy.

3.1.3 Comparison

When visually inspecting the hierarchical structures used by Unity3D and Gazebo, it is difficult to find much in common. One of the most significant differences between the two structures is that a single Unity3D project can, and often does, contain multiple scenes. This allows multiple simulated worlds to use the same settings assets without resorting to duplicating them for use in multiple projects. In Gazebo, while a single SDF file can store multiple simulated worlds, it is rare. Even when multiple world elements exist under the SDF element, simulation settings must be specified for all of them since they cannot be stored at the project level. Instead, the Gazebo hierarchy facilities reuse with the include element, which allows the contents of another SDF file to be included in the hierarchy at run time. In terms of implementing custom

functionality Gazebo plugins and Unity3D custom components offer similar functionality, but Unity3D scripts tend to be simpler, are programmed in a higher-level language, and feature a much more user-friendly compilation process as discussed in detail in the user interface section.

3.2. Coordinate Systems and Units

The coordinate systems and units used by Unity3D and Gazebo determine how coordinates and rotation angles correspond to the simulated locations and orientations of objects within the environment. Neither the coordinate system nor the preferred units of the simulators will impact simulation quality, however they must be accounted for to ensure equivalent environments have been created with regard to the placement of the robot and obstacles.

3.2.1 Unity3D

Unity3D uses a coordinate system where the X-axis points right, the Y-axis points up, and the Z-axis points forward. In ROS terminology, this coordinate system is referred to as right, up, forward, or RUF. This is as opposed to the ROS coordinate system wherein the X-axis points forward, the Y-axis points left, and the Z-axis points up, referred to as forward, left, up or FLU [58]. As an example, the RUF position vector $(1, 2, 3)$ is equivalent to the FLU position vector $(3, -1, 2)$. An image showing the Jackal robot on the axes in the Unity3D coordinate system is shown in Figure 6a. Beyond translation, the coordinate system also affects rotations. Since Unity3D uses an RUF coordinate system, rotation around the X-axis is pitch, rotation around the Y-axis is yaw, and rotation around the Z-axis is roll. Given a three-element rotation vector, Unity3D applies the rotations around the Z-axis first, around the X-axis second, and around the Y-axis last, following a roll, pitch, yaw order [59]. This application order is significant since rotations are non-communitive. Also, note that the RUF coordinate system is left-handed

compared to the right-handed FLU coordinate system [60]. Consequently, in Unity3D a positive angle results in a clockwise rotation about an axis, when viewed from above.

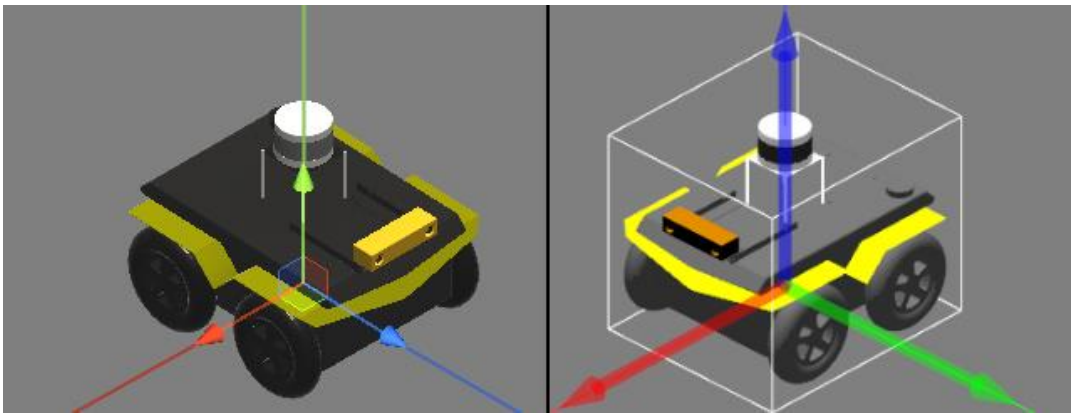
Regarding units, the unit length in Unity3D is equivalent to 1 meter by default [61]. It is possible to adjust the Unity3D unit scale, however the physics calculations assume that 1 unit is equivalent to 1 meter, so scale adjustments can lead to unrealistic behavior [49]. Likewise, masses, forces, and durations in Unity3D default to the International System of Units (SI) standards of kilograms, newtons, and seconds, respectively [62]. The Unity3D interface uses units of degrees for angles and rotations. Internally, Unity3D represents rotations using unitless quaternions to avoid gimbal lock, but these values are not accessible from outside of the programming interface [63].

3.2.2 Gazebo

Like ROS, Gazebo uses an FLU coordinate system where the X-axis points forward, the Y-axis points left, and the Z-axis points up. An image showing the Jackal robot on the axes in the Gazebo coordinate system is shown in Figure 6b. In the FLU coordinate system, rotation around the X-axis is roll, rotation around the Y-axis is pitch, and rotation around the Z-axis is yaw. Given a three-element rotation vector, Gazebo applies the rotations around the Z-axis first, around the Y-axis second, and around the X-axis last, following a yaw, pitch, roll order. As noted in the previous section, the FLU coordinate system follows a right-handed convention, so positive angles result in counterclockwise rotation about an axis when viewed from above [64].

Unless configured otherwise, Gazebo uses the SI standard units of meters, kilograms, newtons, and seconds for measures of length, mass, force, and duration [64]. According to the SDF specification, rotations can be expressed as Euler angles (roll, pitch, and yaw) in radians, as Euler angles in degrees, or as Quaternions, depending on the values of the rotation format and

degrees attributes of the pose element [55]. Regardless of the way the rotation is defined, the Gazebo interface displays the rotation as Euler angles in radians.



(a) (b)

Figure 6. The Jackal robot in its default orientation in both coordinate frames (a) Unity3D and (b) Gazebo with the X-axis in red, the Y-axis in green, and the Z-axis in blue.

3.2.3 Comparison

When considered as individual simulators, the differences between the coordinate systems and units used by Unity3D and Gazebo have little effect. The display of angles in degrees by Unity3D is easier to intuitively understand and modify than the radians used by Gazebo, but this is also subject to user preference. The differences become more significant when considering interactions with other pieces of software, namely ROS. To effectively communicate object positions with ROS, Unity3D must convert between its RUF coordinate representation and the FLU coordinate representation used by ROS and Gazebo, as will be discussed in the connection to ROS section. While this conversion is simple, care must be taken to ensure it is properly applied in both communication directions.

3.3. Time

Differences in the time step and time management strategies used by Unity3D and Gazebo are a major factor in the accuracy and performance differences between them. The time step of a simulator is the smallest difference in simulated time between discrete simulated states,

also called steps. Smaller time steps generally increase accuracy, since the steps will better approximate the continuous nature of reality but will also decrease performance, since a larger number of steps for each time interval need to be simulated. For example, a time step of 0.1 seconds means that every second to be simulated requires 10 intermediate steps to be simulated while a time step of 0.01 seconds requires 100 intermediate steps per simulated second. A time management strategy dictates how time is allocated for simulating different parts of the environment and how the simulation is affected when every simulated second takes longer than one second to compute. When a simulated second takes longer than one second to compute, the simulation is considered slower than real-time, whereas a simulator that simulates exactly one second for every second of elapsed time in the real-world is considered real-time.

3.3.1 Unity3D

As a game engine first and foremost, "[Unity3D] tries to keep the physics simulation up-to-date with the amount of time that has elapsed…" [65]. In other words, Unity3D prioritizes real-time simulation. It does so by using two systems to keep track of time, one with a variable amount of time between steps and the other with a fixed amount of time between steps. The variable time step system is based on the repeated process of drawing the simulation state to the screen. The rate at which the simulation state on the screen is updated, the inverse of the length of a variable time step, is referred to as the frame rate. The fixed time step system uses a predefined time step length, independent of the frame rate. The Unity3D physics system operates exclusively based on the fixed time step system [65].

Outside of the physics system, the majority of custom components use the variable time step system via the Update callback function [66]. The Update function is called by Unity3D every frame, just before the simulation state is rendered to the screen. Since Unity3D renders the

simulation state at the fastest frame rate possible, the variable time step system always uses the smallest time step feasible while maintaining real-time simulation. This represents the optimal tradeoff between real-time simulation and accuracy, since a more accurate simulation would require a shorter time step, which would necessarily result in a slower than real-time simulation. The length of the variable time step, also called the delta time, changes because the time required to render the simulation state is dependent on the complexity of the environment seen by the camera and the computing time allocated to other programs on the same computer, including ROS [65]. For example, in the agriculture environment when the Jackal robot equipped with the Bumblebee2 stereo camera system turns from its initial orientation facing the empty field towards the array of solar panels, the average frame rate decreases from 48 FPS to 45 FPS, representing a 6.6% increase in time step duration.

Despite its benefits, the Unity3D physics system does not use the variable time step system because it can lead to inconsistent behavior amongst simulation runs [65]. Consider the case of a ball rolling towards a ledge 0.75 m away at 1 m / s. If the simulator used a variable time step with a sequence of time steps of 0.4 s, 0.3 s, and 0.5 s, the sequence of ball locations at each step would be 0.4 m, 0.7 m, and 1.2 m, so the simulator would not realize the ball was unsupported until 1.2 s at which point the ball would begin to fall. If instead the time step sequence was 0.4 s, 0.5 s, and 0.3 s, the resultant ball locations at each step would be 0.4 m, 0.9 m, and 1.2 m, wherein the simulator would realize the ball was unsupported at 0.9 s and would begin its descent then. Using a variable time step system leads to inconsistent behavior across runs, even if the steps have the same lengths just in a different order. If instead the simulator used a fixed time step of 0.4 s, the sequence of ball locations would be 0.4 m, 0.8 m, and 1.2 m, and the ball would begin to fall at 0.8 s, every time. In reality, the ball would begin to fall after it

fell off the edge, a moment after 0.75 s. None of the simulators accurately represent this behavior, although in this case the fixed time step physics simulator is closest with the ball beginning to fall at 0.8 s. A summary of these three simulation scenarios is shown in Figure 7. Although it is not necessarily more accurate than variable time step physics simulation, the fixed time step physics simulator is preferred because of its greater consistency and reliability.

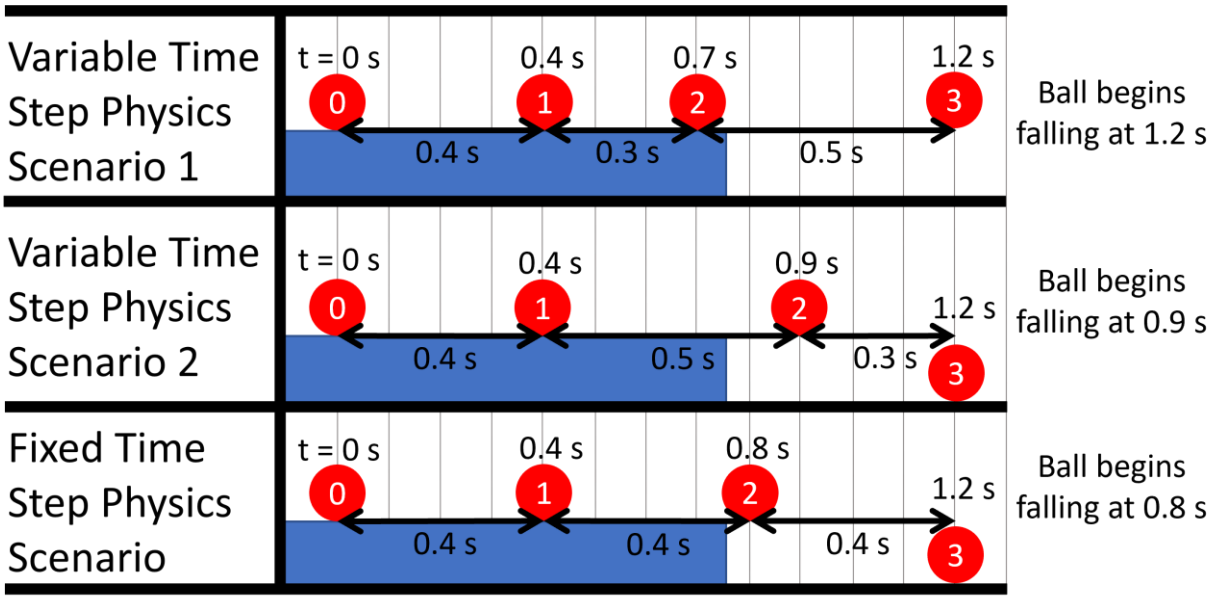


Figure 7. Differences between variable time step and fixed time step physics simulation.

The Unity3D fixed time step works by performing as many fixed time step updates as are needed to catch up with the current time before every variable time step update. So, if for a particular simulation the fixed time step length is longer than the variable time step length, Unity3D will perform zero or one fixed time step updates per variable time step so that the fixed time step system does not get ahead. An example considering fixed time step lengths of 0.02 s and variable time step length of 0.015 s is shown in Figure 8. If instead the fixed time step length is shorter than the variable time step length, Unity3D will perform one or more fixed time step updates per variable time step update, so that the fixed time step system does not fall behind. An example considering fixed time step lengths of 0.02 s and variable time step length of 0.025 s is

shown in Figure 9. In general, the fixed time step length is referred to as the fixed delta time which defaults to 0.02 s. In addition to performing physics calculations, Unity3D also calls the FixedUpdate function of all components for each fixed time step update. A common misconception is that fixed time step updates occur at a fixed frequency set by the fixed time step length. In reality, Unity3D events operate in a single loop that can only loop as fast as the frame rate. If multiple fixed time step updates need to occur per variable time step length, as in Figure 9, Unity3D will simply call the fixed time step update multiple times sequentially in the same event loop iteration [65]. A simplified event loop diagram is shown in Figure 10.

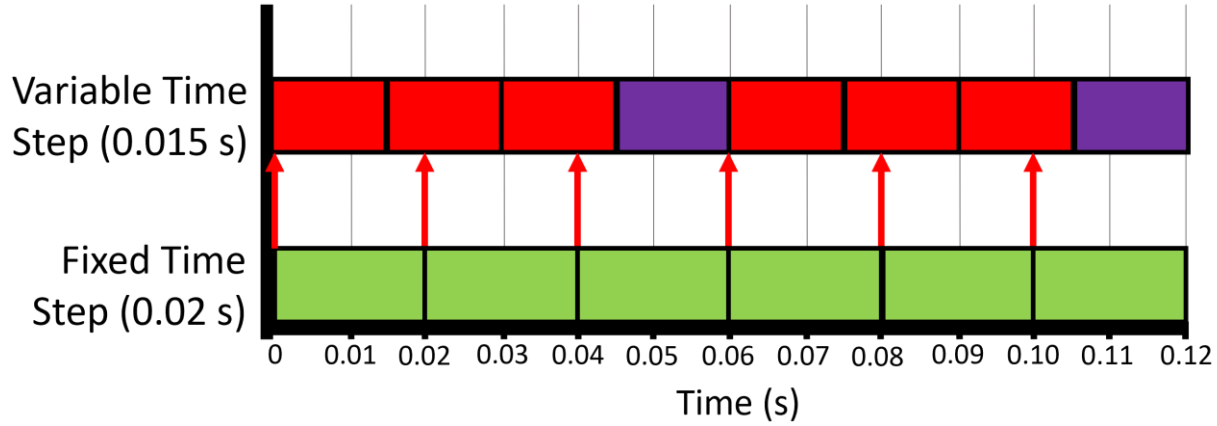


Figure 8. Example with a longer fixed time step length than variable time step length. One fixed time step occurs for every variable time step marked in red while zero fixed time steps occur for variable time steps marked in purple.

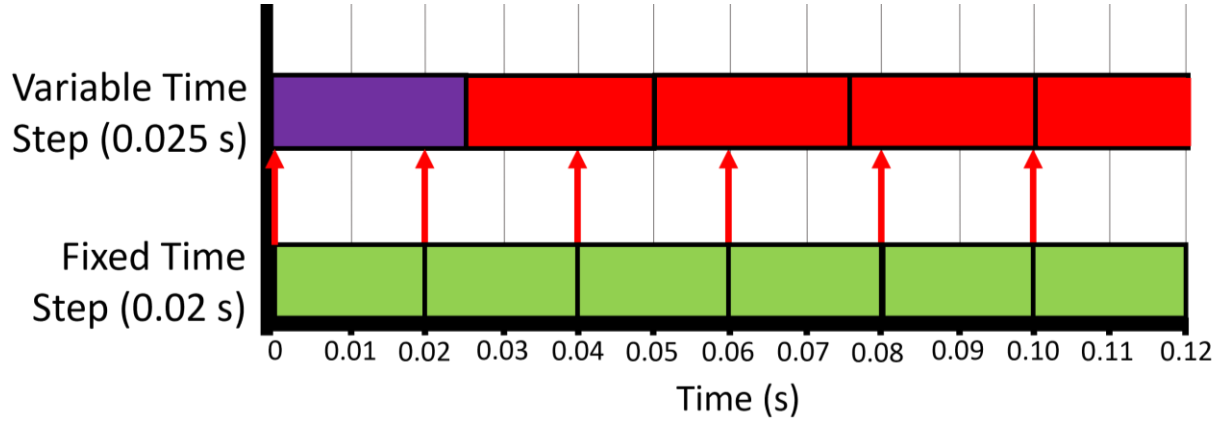


Figure 9. Example with a shorter fixed time step length than variable time step length. One fixed time step occurs for every variable time step marked in red while two fixed time steps occur for variable time steps marked in purple.

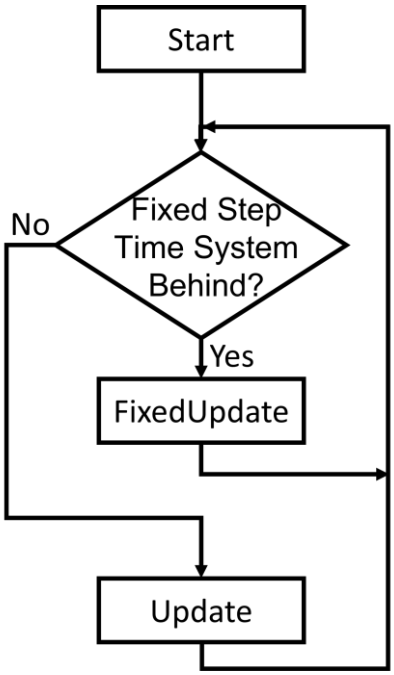


Figure 10. A simplified version of the Unity3D event loop.

Even with the two systems to keep track of time, it is still possible for Unity3D to encounter situations in which it is momentarily unable to maintain real-time performance. In these situations, the image on the screen does not update and the variable time step length can become significantly extended. As the variable time step length increases, the number of fixed time step updates to perform on the subsequent event loop iteration grows, which can further slow the event loop, increasing the variable time step length. To prevent too many fixed time step updates from building up, Unity3D enforces a maximum delta time. The maximum delta time is the length of the maximum permissible variable time step, as perceived by the simulated environment. For example, if a long running calculation temporarily slowed down rendering as to lead to a 1 s long variable time step, this would ordinarily require 50 fixed time step updates, assuming a 0.02 s fixed time step length, before the next variable time step update. Instead enforcing the default 0.333 s maximum delta time, the simulation would perceive the maximum 0.333 s as having passed, despite 1 s of real time having passed, requiring only 16 fixed time step updates before the next variable time step update. As a consequence, the Unity3D internal clock

will be 0.667 s behind the true time, until the simulation is restarted. Despite the delay, Unity3D will still attempt to simulate 1 s for every 1 s of time that passes in the real-world. The enforcement of a maximum delta time means that Unity3D simulation is not guaranteed to be real-time, but these situations are typically rare and are an acceptable trade-off for maintaining real-time performance most of the time [65]. If it is not acceptable that the Unity3D simulation fall behind real-time, the maximum delta time can be increased within the limits of a 32-bit floating point number.

If instead it is more desirable to increase accuracy by intentionally performing slower than real-time simulation, Unity3D supports a configurable time scale parameter. By default, the time scale is set to 1.0 so that simulated time matches real time. Decreasing the time scale to 0.5 would slow down the simulation such that for every 1 s of real time, only 0.5 s of simulated time would elapse. The variable time step system still uses the smallest time step possible, but the halved time scale means that the simulation perceives the variable time step updates as occurring twice as often. By contrast, the fixed time step system uses the slowed internal clock to determine its rate so that fixed time step updates occur at the same rate relative to the simulated time, but half as often relative to real-world time. For example, consider a simulation with a variable time step length of 0.05 s and a fixed time step length of 0.02 s. At a time scale of 1.0, 20 variable time step updates and 50 fixed time step updates occur every real second as well as every simulated second, since they are the same at a time scale of 1.0. At a time scale of 0.5, 20 variable time step updates and 25 fixed time step updates occur every real second, but 40 variable time step updates and 50 fixed time step updates occur every simulated second, since 2 real seconds pass for every simulated second at a time scale of 0.5. The increased number of variable time step updates per simulated second can increase sensor update rates or simulation

accuracy in general while the consistent number of fixed time step updates per simulated second ensures consistency in the physics system. The time scale can also be increased above 1.0 for faster than real-time simulation, with the caveat that fewer variable time step updates will occur per simulated second [65].

With regard to robot simulation, sensors that depend primarily on the visual state of the environment, such as cameras and LiDAR, should be simulated using the variable time step system in the Update function of a custom component. This is because Unity3D can only produce data for these sensors at the rate at which the environment is rendered, which is the rate at which the variable time step update is executed. Sensors that are not dependent on the visual state of the environment, such as IMUs and wheel encoders, can be simulated using either the variable time step system, using an Update function, or the fixed time step system, using a FixedUpdate function. Since these sensors typically only depend on the state of a robot in the physics system, they can be simulated at the same rate as the physics system in the fixed time step update. Nevertheless, these sensors can also be simulated in the variable time step update since the visual state of the environment itself depends on the state of the robot in the physics system. Any type of sensor that does not need to update every time step can be implemented by only simulating the operation of the sensor when the current simulation time plus the time step minus the time the sensor was last used exceeds the length of the update period of the sensor.

3.3.2 Gazebo

While Unity3D tends to prioritize real-time simulation, Gazebo neither prioritizes real-time simulation nor simulation accuracy above the other, instead providing a wide array of configuration options that can be used to achieve the desired level of performance. The time step length, real-time factor, simulation constraints, and physics engine itself are all configurable to

trade-off between simulation speed and accuracy [67]. In all cases, Gazebo uses a single time system and calculates the state of the simulated world once per iteration [68].

The length of the simulated time between discrete simulation states, the time step length, used by Gazebo is configured using a parameter referred to as the max step size. It is called the max step size because some physics engines like Bullet can simulate using a smaller step size to achieve greater simulation accuracy when enough computing resources are available. By default, the minimum allowed step size and the maximum allowed step size are set to the same value. This ensures consistent physical behavior between simulation runs, like with the fixed time step system in Unity3D. The time step length in Gazebo defaults to 0.001 s [55].

Simulation time steps are executed at a rate called the real-time update rate. The default real-time update rate in Gazebo is 1000 Hz, which multiplied by the time step length of 0.001 s yields a real-time factor of 1. Thus, both Unity3D and Gazebo are configured for real-time simulation by default. Unlike Unity3D, if Gazebo is not able to update at the desired rate the only option is for it to update as fast as it can, based on the computing power available. This means that Gazebo will continuously simulate at a slower than real-time rate when overloaded. A figure graphically demonstrating real-time, slower than, and faster than real-time simulation is shown below in Figure 11. Like with Unity3D, slower than real-time simulation can be intentionally achieved by decreasing the real-time update rate, however this will not increase accuracy since the step size will remain constant. To simulate slower than real-time while improving accuracy, the step size must be decreased while the real-time update rate is kept constant. For example, halving the time step length to 0.0005 s and keeping the real-time update rate of 1000 Hz yields a real-time factor of 0.5, indicating that 0.5 s of simulated time pass for

every 1 s of real time. Likewise, real-time factors above 1.0 for faster than real-time simulation can be achieved by increasing either the step size or the real-time update rate [67].

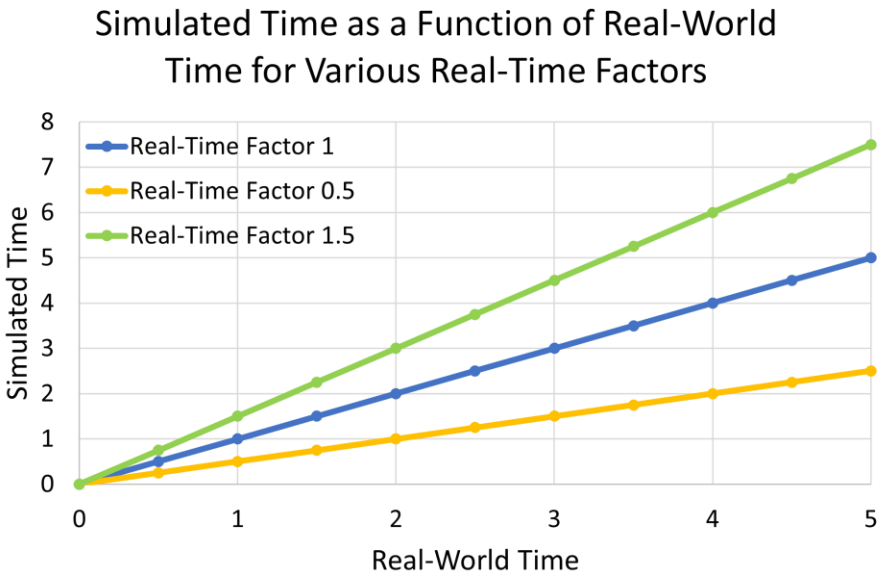


Figure 11. Simulated time as a function of real-world time for various real-time factors.

With only a single time system, all sensors in Gazebo are simulated according to that system. One consequence of this is that to accurately simulate a sensor, its update period length should be an integer multiple of the step size length. For example, a sensor with an update rate of 500 Hz has an update period of 0.002 s, which is an integer multiple of the default 0.001 s step size. A sensor with a slower update rate of 400 Hz has an update period of 0.0025 s, which is not an integer multiple of the default 0.001 s step size. As a result, the sensor updates more slowly than the desired 400 Hz. This may be acceptable in some cases, but if it is not, it can be rectified by decreasing the step size, here to 0.0005 s [69].

3.3.3 Comparison

In general, the single fixed time system used by Gazebo is easier to understand and interface with sensors, but the combination of a variable time step and fixed time step system used by Unity3D provides the best trade-off between real-time performance, accurate simulation,

and reliable physics. With that being said, it also must be considered that the default fixed time steps used by the simulators vary drastically with 0.02 s being used in Unity3D and 0.001 s being used in Gazebo. The 20 times smaller time step used by Gazebo implies that it should simulate much more accurately than Unity3D, but this is also highly dependent on the accuracy of the simulators' physics engines.

3.4. Physics

In the real-world, the way in which objects move in response to interactions with other objects is described by physics. Similarly, the way in which simulated objects move in response to interactions with objects and external forces is determined by the simulated physics. The portion of a simulator responsible for simulating physics is referred to as the physics engine. Unity3D and Gazebo use different physics engines, both of which will be described and investigated in the following subsections.

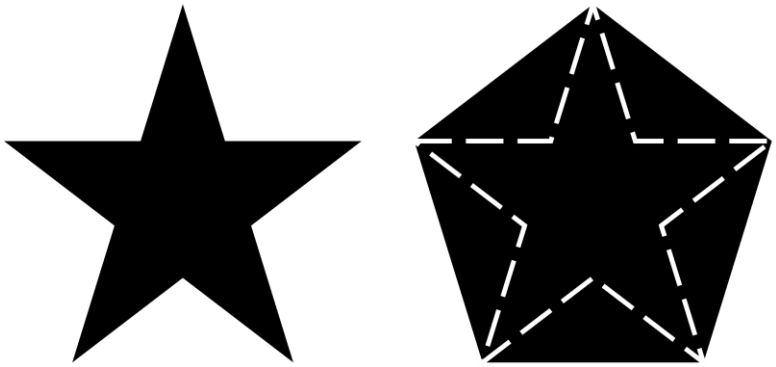
3.4.1 Unity3D

Since 2019, the latest versions of Unity3D use the PhsyX 4 physics engine [70]. PhysX is an open-source physics software development kit (SDK) written in C++ published by Nvidia. Two design priorities of PhysX are scalability and portability meaning that PhysX runs well for both simple and complex simulation environments across a wide range of devices. PhysX 4 is primarily intended for simulating rigid body dynamics and supports connecting rigid bodies using joints with varying degrees of freedom (DoF). Currently, the next version of PhysX, PhysX 5, is in development and is expected to support soft body dynamics and large scale fluid simulations once completed [71]. Since PhysX 5 is neither complete nor implemented into Unity3D, the remainder of this section will only consider PhysX 4.

One of the most important aspects of rigid body dynamics is how the forces applied to a simulated object are translated into motion. In the PhysX engine, this is a multistep process for each rigid body being simulated. Every fixed time step update, the forces acting on a body are obtained. Some forces, like gravity, are constant and can be immediately found, while others must be calculated based on the state of the environment. Next, all the forces and torques acting on the body are accumulated. The forces and torques are applied as accelerations, subject to the mass and rotational inertia of the body. After the forces and torques have been applied, they are reset to zero in preparation for the next frame. As expected, the velocities of objects are updated according to the product of their accelerations and the time step. The position and orientation of objects are updated similarly based on the velocity [72].

One force that must be found from the environment is the normal force. Normal forces are generated via contact between two objects, for example a wheel and the ground. In PhysX, normal forces are generated for a rigid body with a collider when it detects that a collision with another collider has occurred. Rather than start when the colliders visually touch, collision detection occurs whenever the distance between colliders is less than a contact offset value. The normal force generated by a collision smoothly increases as the distance between the colliders decreases. When the distance between the colliders reaches a value called the rest distance, the force is set to be just high enough to prevent further penetrating motion. Should the distance decrease below the rest distance, meaning the colliders overlap, the normal force increases further to force the objects apart [73]. In Unity3D, the global default contact offset is set at the project level as 0.01 units (1 cm) while the rest distance is hardcoded to 0 cm [74]. Collisions are detected using a two-step process. First, pairs of colliders are classified using a simple broad-phase algorithm to determine whether it is possible that the colliders are in contact with one

another [75]. Next, a more rigorous contact detection algorithm is used on all pairs of colliders that might be in contact with each other to determine definitively if and where specifically on the objects contact is occurring [73]. Both steps support multiple possible algorithms with Unity3D defaulting to the “sweep and prune” broad-phase algorithm and the “persistent contacts manifold” contact detection algorithm [74]. One caveat of the PhysX collision detection system is that it can only detect collisions when the shape of at least one of the colliders is convex, meaning the shape can have no interior faces [76]. Fortunately, this tends not to pose an issue in practice, especially since Unity3D can approximate colliders based on the convex hull of meshes when necessary. Figure 12 shows an example of a non-convex polygon and its convex hull.



(a) (b)

Figure 12. Example of a (a) non-convex polygon and (b) its convex hull.

Another type of environmental force that is intimately related to the normal force is the friction force. Friction forces are forces that resist lateral motion between two contacting surfaces. PhysX simulates friction using the Coulomb friction model which is based on a static friction coefficient and a dynamic friction coefficient. When the two contacting surfaces are not moving laterally relative to one another, the friction force resists the action of a net force in the lateral direction, as long as the magnitude of the net force is less than the product of the static friction coefficient and the normal force generated by the surface contact. When the two contacting surfaces are moving laterally relative to one another, the friction force is equal to the

product of the dynamic friction coefficient and the normal force generated by the surface contact. In the dynamic case, the friction force is applied in the opposite direction of the lateral movement velocity [72]. In Unity3D, colliders are assigned a material that determines the static and dynamic friction coefficients of the surface. As noted in the Unity3D documentation, the PhysX friction model is tuned for performance and stability and is not necessarily an accurate approximation of real-world physics [77].

Forces can also be generated by physically simulated bodies that have been joined to one another. PhysX supports joining simulated bodies using either joints or articulations. Only articulations, specifically reduced coordinate articulations, will be discussed in this section since they replicate the features of joints while being designed for accurate simulation in robotic applications [78], which is the primary concern of this thesis. In general, an articulation is a set of links, each of which behaves like a rigid body, that is organized in a hierarchical structure. Each link, except for the root, is connected to its parent by one of four types of joints that set the movement DoF relative to the parent. The 4 joint types are: fixed with 0 DoF, prismatic (linear) with 1 DoF, revolute with 1 DoF, and spherical with up to 3 DoF. Reduced coordinate articulations cannot be broken because the articulation joint positions are internally tracked using only one value for each DoF. These positions are then used to determine the poses of the links in the articulation relative to the root. Articulations produce forces when they are given a target velocity command. PhysX automatically calculates the force required to accelerate the link to the target velocity and applies it, subject to a configurable maximum force limit. The rate at which the joint velocity can be changed can also be limited by a velocity damping parameter to prevent jerky motion [78]. In the Unity3D simulation, the Jackal robot is an articulation with revolute joints connecting the chassis to the wheels. When the Jackal robot is commanded to move

forwards, a target angular velocity for each wheel is calculated, and the articulation applies a torque to the revolute joints to reach and maintain the desired angular velocity. The torque on the revolute articulation joints is applied to the chassis and wheel links, ultimately leading to wheel rotation and forward rotation.

3.4.2 Gazebo

Gazebo has supported the ODE, Bullet, Simbody, and DART physics engines since Gazebo version 3.0. However, support for the Bullet, Simbody, and DART physics engines is not included in the default Gazebo binary. Since just the ODE physics engine is supported by default [20], it will be the only physics engine discussed in the remainder of this section. ODE or Open Dynamics Engine is an open-source, high performance, physics SDK written in C++ for simulating rigid body dynamics. It supports many of the same features as PhysX 4, including linking rigid bodies with joints of varying DoF [79].

As another example of the similarities between the two physics engines, forces are applied to simulated objects in ODE identically to how they are applied in PhysX. Each Gazebo time step, the forces that have been applied to a body are accumulated, then applied to the body. During the integration step, forces and torques are divided by the mass and rotational inertia, respectively, to obtain the accelerations which are integrated to update the velocities. The velocities themselves are also integrated to update the position and orientation of the bodies. Forces and torques are set to zero after each time step, also just as in PhysX [80].

In ODE, normal forces are calculated when two bodies collide to determine friction forces, but the normal forces themselves are not applied to the contacting bodies. Instead, all colliding bodies are treated as a special type of joint called a contact joint. To prevent colliding bodies from penetrating one another, the velocities of the colliding bodies are altered to eliminate

inward velocity components and only allow sliding velocities and outward velocities in the direction of the contact normal. This prevents the colliding bodies from passing through one another without requiring the explicit application of a normal force. A contact surface layer depth parameter can be configured to allow the contacting bodies to partially sink into one another before the inward velocity components are enabled [80]. A small default value of 0.001 (1 mm) is used for this parameter in Gazebo because it can increase the stability of contacts and prevent jittery behavior [55]. Should collision depth of the bodies exceed this threshold, a contact correcting velocity is applied to the bodies. The contact correcting velocity is calculated as the length of the overlap divided by the time step to ensure that the bodies are separated immediately in the following step [80]. In Gazebo, the maximum contact correcting velocity is limited to 100 m/s by default since it can prevent deeply embedded objects from appearing to teleport [55]. Collisions between objects in ODE are detected using a two-step process, similar to that used by PhysX, with a broad-phase step that determines which pairs of objects are potentially colliding, followed by a narrow-phase step that finds the specific collision points for each pair of potentially colliding objects. The broad-phase step chooses from several algorithms to determine potentially contacting pairs including via a multi-resolution hash table search, a quadtree search, and a sweep-and-prune algorithm. The narrow-phase step does not use any single algorithm but rather uses specialized code based on the types of the colliding objects [81].

Friction forces are simulated by ODE in Gazebo using a simplified Coulomb friction model. By default, Gazebo uses a pyramidal model that models the friction force as a combination of friction forces along primary and secondary directions along the X- and Y- axis of the world [67]. The maximum limits of the friction forces along each direction are equal to the product of the normal force and the coefficient of friction [80]. Different coefficients of friction

can be specified for each axis in Gazebo simulation materials. No distinction is made between static and dynamic friction, so the same coefficients of friction are used when surfaces are still or move relative to one another. One caveat of the pyramidal friction model is that an object which slides along a surface with an initial velocity not aligned to a world frame axis still experiences the same friction forces along each axis. As a result, the velocity component in one axis dissipates more quickly than in the other, causing the object to follow a curved trajectory [67].

ODE joints are used to constrain the motion of objects relative to one another. Joints may be used for the purpose of implementing wheels, hinges, and pistons, as well as in the already discussed context of creating contact joints for colliding objects. One example of a joint is a hinge, which constrains all relative linear movement while allowing relative rotation along a single axis. At each simulation step, joints apply forces to the bodies they link in such a way as to preserve the constraint that the joints enforce. Even with the application of constraint forces, joint constraints can occasionally be violated as a result of the accumulation of errors throughout a simulation. When this occurs, ODE will apply an additional force to the bodies to bring them into a state where the joint constrain is met. The additional force is controlled by the error reduction parameter (ERP) which takes a value between 0 and 1. An ERP of 0 means ODE will not apply a force to correct any joint errors. A value of 1 means ODE will attempt to fully fix all joint errors in the following time step, however this can lead to instability. As a compromise between correcting action and stability, the default ERP in ODE and Gazebo is 0.2. In other words, ODE joint forces permit a small amount of error in the relative positions of joined bodies. Beyond the forces required to enforce joint constraints, ODE joints can also be driven using motors. Motors take in a desired velocity and maximum force value and apply the force that

needs to be applied to bring the affected body up to speed in a single step, provided that the force required is not greater than the specified maximum [80].

3.4.3 Comparison

The PhysX and ODE physics engines, used by Unity3D and Gazebo, respectively, are largely similar with only a few differences. The largest difference is in the modeling of friction forces. The PhysX engine more realistically simulates separate static and dynamic friction in addition to friction forces that directly oppose movement. Another major difference is that PhysX articulations have separate force limits and damping properties to limit the amount the velocity can increase by in a single step while ODE joint motors do not. This can lead to more realistic movement in PhysX as articulations accelerate over time whereas ODE joints instantly accelerate. The final major difference between the simulators is that ODE has greater flexibility in that it can detect collision between arbitrary 3D meshes while PhysX requires that at least one mesh be convex.

Smaller differences between the engines include the use of normal forces and the collision detection algorithms used. Neither the PhysX step of applying normal forces to separated bodies nor the ODE technique of forgoing normal forces and simply canceling inward velocities exactly model real-world behavior, but it is possible that the simpler ODE approach may have a slight performance advantage. As for the collision detection algorithms, determining which algorithm has the best performance would likely require analysis of the implementations and is out of the scope of this thesis. In aggregate, the small differences between the physics simulators tend to suggest that PhysX and thus Unity3D should yield more realistic results.

3.5. Model Compatibility

Model compatibility refers to the types of files that can be imported and used by each simulation suite. Models of 3D objects are typically made up of mesh files, which represent the 3D structure of the object, and texture files which describe the appearance of the surface of the object. The following section will describe the compatibility with several popular 3D object file formats and texture formats.

3.5.1 Unity3D

A wide variety of model and texture file types are supported by Unity3D. The following standard 3D model file formats are supported, listed as "name (file extension)": Autodesk Filmbox (.fbx), Collada (.dae), drawing exchange format (.dxf), and OBJ (.obj) [82]. ROS URDF model files can also be imported into a simulation using the URDF Importer Unity3D package published by Unity Technologies [83]. The following image file formats can be imported as textures: bitmap image (.bmp), OpenEXR (.exr), graphics interchange format (.gif), high-dynamic-range imaging (.hdr), interchange file format (.iff), JPEG (.jpg), PICT (.pict), portable network graphics (.png), Photoshop document (.psd), Truevision TGA (.tga), and tag image file format (.tiff). Unity3D supports a maximum image size of 16,384 x 16,384 pixels [84].

3.5.2 Gazebo

Gazebo also supports several model and texture file formats. The following 3D model file types are supported: stereolithography (.stl), Collada (.dae), and OBJ (.obj), with Collada and OBJ being the preferred formats [85]. URDF files can be imported by Gazebo, however it is required that all link elements in the URDF be updated to include a properly configured inertial child element [86]. Gazebo supports all of the textures supported by the underlying OGRE rendering engine, which are as follows: bitmap image (.bmp), graphics interchange format (.gif),

JPEG (.jpg), Camera RAW (.raw), portable network graphics (.png), Truevision TGA (.tga), and DirectDraw Surface (.dds) [87].

3.5.3 Comparison

Both Unity3D and Gazebo support importing from model and texture formats that the other does not. Support for URDF files in both simulators allows both to be easily used in robotics applications. In terms of the raw number of supported formats, Unity3D supports more with four model formats, URDF files, and 11 texture formats while Gazebo supports three model formats, URDF files, and seven image formats. However, support for a larger number of file types is not necessary since programs to convert between different 3D model files and image files exist. An advantage of Gazebo is that there are no image import size limits, meaning that it is possible to use more photorealistic, higher resolution textures.

3.6. User Interface

The user interface is the mean or means by which the user interacts with the simulator. An interaction encompasses a wide variety of actions including configuration simulation world settings, moving objects within the simulated world, starting a simulation, and stopping a simulation. Interactions can be mediated via several methods including GUIs, command line-based user interfaces, and the editing of text files.

3.6.1 Unity3D

When working with Unity3D, many configuration and visualization actions are done through a GUI referred to as the Unity Editor. An image of the Unity Editor with the agriculture test environment is shown in Figure 13 below. The content in the section outlined in red and numbered as 1 is called the scene view. The scene view allows for visual navigation and editing of the simulated scene. The scene is navigated using the keyboard and mouse. GameObjects in

the scene can be selected by clicking on them which brings up various tools for visually editing the position, orientation, and scale. Currently, the game object representing the left camera of the BumbleBee2 stereo camera system is selected and the translation tool is displayed in its position as the multicolored axis. The content in the section outlined in yellow and number 2 is the hierarchy window. The hierarchy window displays a hierarchical text representation of every GameObject in the scene. Currently, the GameObject labeled as “left” is highlighted, since this is the left camera of the BumbleBee2 stereo camera that was selected in the scene view. The section outlined in green and numbered 3 is the inspector window. The inspector window displays all of the components in the currently selected GameObject and their properties. Custom script components also have editable properties that are shown in the inspector window. In the current view, camera sensor settings including the ROS topic to publish to and the publish period are shown and can be edited. The section outlined in blue and numbered 4 is called the project window. The project window displays all the assets imported into the project, even if they have not yet been added to the scene. New models can be imported by dragging them into the project window or right clicking and selecting the import option. Model assets can be dragged from the project window into the scene or hierarchy view to initialize them as GameObjects in the current scene. Script assets can be dragged from the project window into the inspector window to initialize them as components on the currently selected GameObject. In the figure, the project window displays the contents of the scripts folder of the project. The small purple section numbered 5 contains the play, pause, and step controls. When the play button is pressed, the Unity Editor switches from “edit mode” into “play mode”. In play mode, the physics engine is enabled and the simulation starts. In play mode, the pause and step controls can be used to temporarily halt the simulation and advance one simulation step, respectively. In play mode, the

play button turns into a stop button which can be used to end the simulation. When the simulation is ended, all objects return to their original locations [88]. The section outlined in gray, numbered 6, displays the console window which is used to show errors, warnings, and other messages. The information displayed in this window is primarily intended to help identify and resolve issues in the Unity3D project [89].

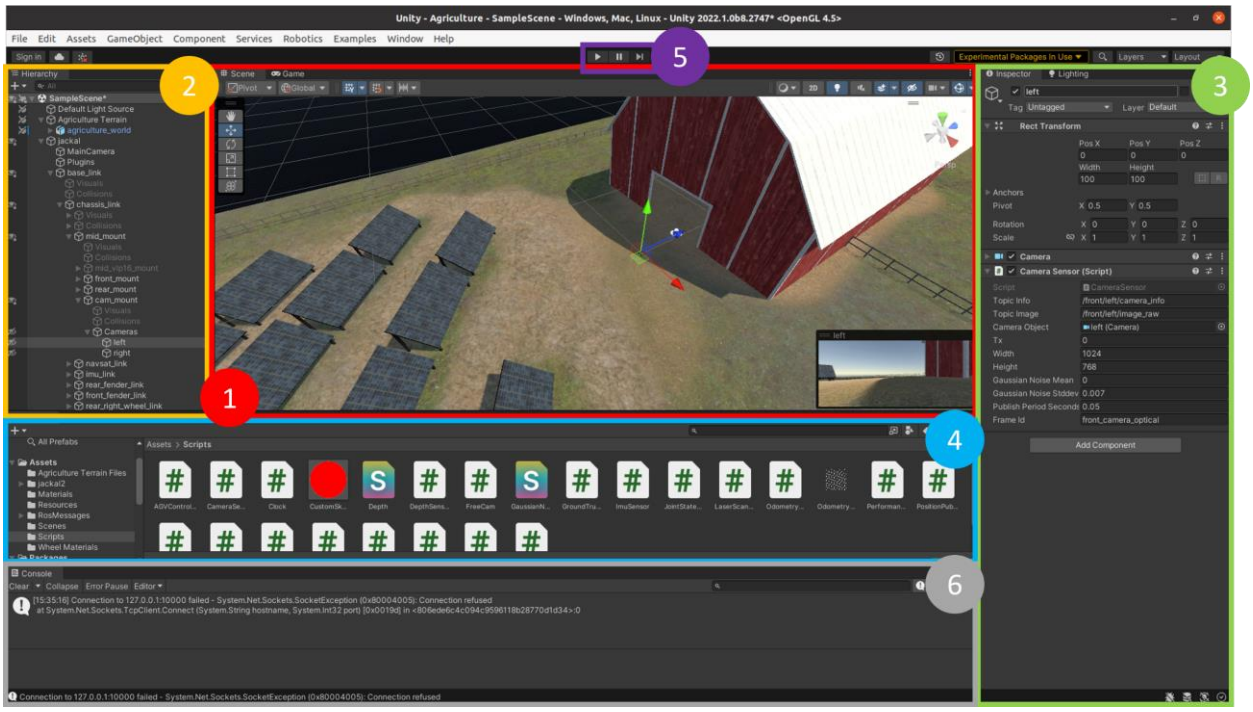


Figure 13. The Unity Editor showing the agriculture test world.

A few interactions with Unity3D occur outside the Unity Editor. Project-level settings are not shown in the main Unity Editor since they are rarely edited after a project is initially created. These settings can still be configured via the project settings window which is accessible under the edit tab in the Unity Editor menu bar. The other main action done outside the Unity Editor is editing script files. Double clicking on a script file in the project window will open it in the computer's default text editor. When a script file is saved, Unity3D automatically attempts to recompile it and prints any errors to the console. For more advanced editing, Unity3D has simple guides for integrating code editors like Visual Studio Code with debugging capabilities that

allow stepping through scripts a line at a time and accessing script variables in play mode while the simulation is running [90].

3.6.2 Gazebo

When working with Gazebo, some configuration and many visualization actions are done through a GUI referred to as the Gazebo GUI. An image of the Gazebo GUI with the agriculture test environment is shown in Figure 14 below. The content in the section outlined in red and numbered as 1 is called the scene view. The scene view shows the simulated objects and allows interacting with the simulated environment. The scene is navigated using the mouse. Objects in the scene can be selected by clicking on them. Objects' positions can be visually altered by selecting the translate button in the toolbar and dragging the object with the mouse. Currently, the Jackal robot is selected and is surrounded by a white wireframe box. The content in the section outlined in yellow and number 2 is the hierarchy portion of the world tab. The world tab window displays a hierarchical representation of the models in the scene along with their properties. Currently, the "base_link" link in the Jackal robot model is selected. The section outlined in green and numbered 3 shows the properties of the element selected in the hierarchy portion of the world tab. Currently, the pose of the "base_link", along with all of its inertial, collision, visual elements, and sensors are shown. Although the camera sensor is shown in this panel, it has no options that can be configured in the Gazebo GUI. The section outlined in blue and numbered 4 is the right panel. The right panel displays the properties of the joints in the currently selected model. Since the "base_link" element that is currently selected is not a model, no joints are shown. The purple section numbered 5 is called the bottom toolbar. The bottom toolbar displays information about the simulation like the time passed in the simulation, number of simulation steps performed, and the real-time rate. Pause and step buttons are also present to

temporarily halt the simulation and advance one time step. The simulated environment can be reset by accessing the edit tab in the Gazebo GUI menu bar and selecting reset world [68].

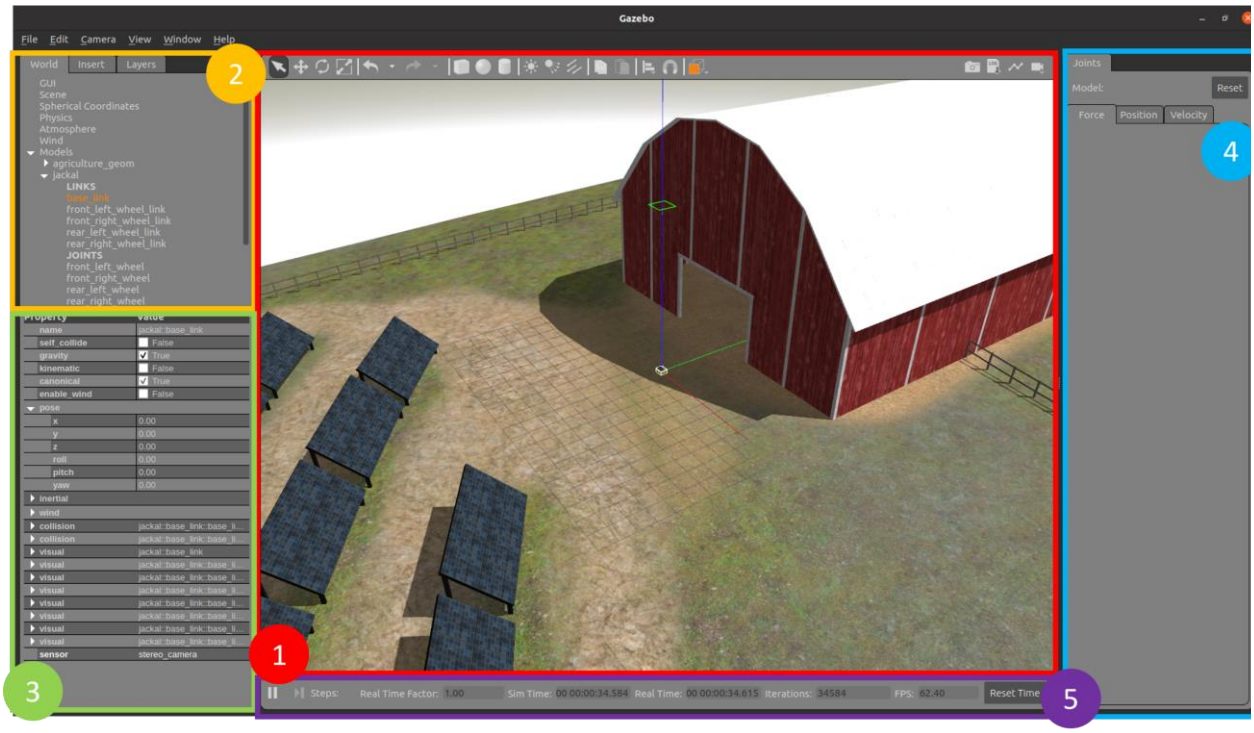


Figure 14. The Gazebo graphical user interface showing the agriculture test world.

Several interactions with Gazebo simulation environments occur outside of the main Gazebo GUI. One such interaction is inserting 3D model files. While the Gazebo GUI itself supports inserting models, this is only possible with models contained in SDF files. SDF model files can either be created manually using a text editor or by opening the Gazebo Model Editor and adding a custom mesh of one of the supported types [91]. The configuration of robot sensors and Gazebo plugins must also be performed by manually editing the relevant simulation text files since this is not possible in the Gazebo GUI or Model Editor. Like with scripts in Unity3D, custom Gazebo plugins must also be created outside of the main Gazebo GUI. Unlike Unity3D, Gazebo does not automatically compile custom code, so it must be manually compiled as a shared library. If the custom code is recompiled, Gazebo must be restarted to reload the updated

shared library [56]. Debugging plugins during simulation is possible using external software like the GNU debugger, however this requires that Gazebo itself be compiled in debug mode [92].

3.6.3 Comparison

Both Unity3D and Gazebo have feature rich GUIs and situations in which the manual editing of text files outside of the primary interface is required. Comparing the two interfaces reveals that the Unity Editor supports a greater number of features compared to the Gazebo GUI. The Unity Editor supports directly importing 3D objects into the scene, configuring sensors, and compiling scripts while the Gazebo GUI cannot perform these tasks. Furthermore, Unity3D has superior out-of-the-box support for debugging than Gazebo. On the other hand, Gazebo worlds can be fully created and configured using only text files which can be more efficient when a graphical interface is not required. Unity3D scenes can only be created within the Unity Editor.

3.7. Connection to ROS

Though not required for generic simulation needs, a connection to ROS is arguably one of the most important aspects of any robot simulation suite. While fully autonomous robotic systems could be implemented entirely within Unity3D scripts or Gazebo plugins, connecting to ROS allows existing ROS libraries for sensor data processing and navigation to be used. ROS also provides a common platform that can be used for both simulated and real robots. The following sections discuss the methods used by Unity3D and Gazebo to integrate with ROS including how ROS messages are received and sent and how the ROS clock is published.

3.7.1 Unity3D

As discussed in the second chapter, several frameworks for bridging Unity3D and ROS have been developed including rosbridge, ROS#, and ROS-Unity. Of the three, ROS-Unity was published the most recently and is the only one maintained by Unity Technologies, the creators

of Unity3D. Therefore, this section will discuss the structure and operation of the parts that make up the ROS-Unity framework required to fully bridge Unity3D and ROS.

The message passing interface between ROS and Unity3D is created using a combination of the ROS TCP Connector Unity3D package and the ROS TCP Endpoint ROS node. On the Unity3D side, custom components created via scripts can use functions within the ROS TCP Connector package to send and receive ROS messages. Scripts can connect to ROS by importing the package and creating a ROSConnection object. Publishing and subscribing to ROS topics is accomplished through the ROSConnection object. Within the ROS TCP Connector package, C# classes have been created for all standard ROS message and service types to ensure that ROS can properly interpret the information sent by Unity3D. C# classes for non-standard message and service types can be automatically generated within the Unity Editor by the MessageGeneration plugin within the ROS TCP Connector package [93]. As the names implies, the ROSConnection object connects to the ROS TCP Endpoint ROS node using a transmission control protocol (TCP) network connection. ROS messages are sent directly as binary data over the TCP connection in a dedicated thread. Sending the messages as binary data in a dedicated thread allows large messages like images to be sent quickly with little impact on the simulation. As a ROS node itself, the ROS TCP Endpoint simply relays messages between the rest of the ROS nodes and Unity3D via the ROS TCP Connector.

The ROS TCP Connector passes messages to and from scripts without modifying their content to account for the differences between Unity3D and ROS. Primarily, this means that scripts dealing with data relative to the coordinate system must convert the data to and from the FLU coordinate system used by ROS. For this purpose, the ROS TCP Connector package includes an importable ROSGeometry library. This library extends the Unity3D vector and

quaternion classes with methods for converting between the RUF and FLU coordinate systems. Thus, scripts can convert vectors and quaternions based in the RUF Unity3D coordinate system to the FLU coordinate system before sending them in a message to ROS. Similarly, scripts can convert vectors and quaternions from ROS to use the RUF coordinate system immediately after receiving them [94].

Since in robotics it is often desirable to make decisions based on time, for example integrating accelerometer measurements over time to estimate velocity, it is important that ROS be aware of the passage of time. ROS provides the "/clock" topic specifically for receiving updates about the passage of simulated time since it often differs from real-world time. In the ROS-Unity3D environment, the ROSClockPublisher script publishes the simulated time from Unity3D to ROS every variable time step. This way, if the simulation is paused or intentionally run at a reduced time scale ROS correctly tracks the time passed in the simulation. The ROSClockPublisher script is not included in the main ROS-Unity framework, but instead is provided with the ROS Navigation 2 SLAM example ROS-Unity3D project published by Unity Technologies [95]. A summary of the connection between Unity3D and ROS is shown in Figure 15.

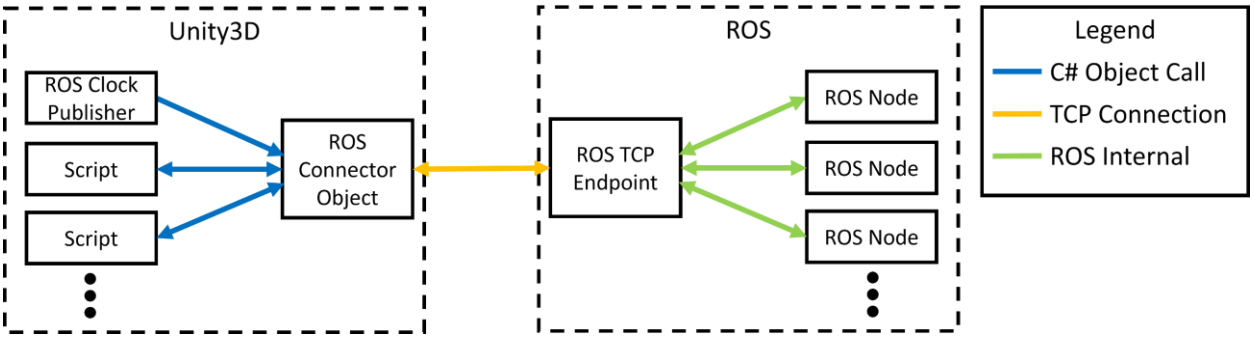


Figure 15. Communication between Unity3D and ROS.

3.7.2 Gazebo

Having been first bridged over a decade ago and developed in tandem for several years, Gazebo and ROS have a very mature and performant integration framework [20]. The framework consists of a set of packages collectively referred to as “gazebo_ros_pkgs”. This collection includes a Gazebo plugin for controlling Gazebo from ROS and several Gazebo plugins interacting with various kinds of sensors and joints from ROS [23]. The following section will discuss the packages within the collection and how message passing between Gazebo and ROS is accomplished in more detail.

The most important single piece of software for establishing communication between ROS and Gazebo is the “gazebo_ros_api_plugin”. This Gazebo plugin initializes a ROS node called “gazebo” that is used as the intermediary for all messages between Gazebo and ROS, analogous to the ROS TCP Endpoint in the ROS-Unity3D integration. Unlike the ROS TCP Endpoint, the “gazebo” node does not communicate with Gazebo via a TCP connection, but instead is directly manipulated as a C++ object. The “gazebo_ros_api_plugin” also interfaces with the Gazebo internal scheduler to publish the simulation time, the state of all links in the simulation, and the state of all models in the simulation through the “gazebo” node every time step. Beyond publishing, the plugin implements numerous ROS services through the “gazebo” node allowing full control of the simulation including updating element properties, applying forces, pausing the simulation, and resetting the world [96]. Custom plugins can also publish and subscribe to ROS topics through the “gazebo” node. Several plugins implementing sensors, such as cameras, LiDARs, and IMUs, that publish to ROS topics exist and are included in the “gazebo_plugins” package within “gazebo_ros_pkgs”. The same package also contains plugins

implementing actuators and drivetrains that subscribe to ROS topics [23]. A summary of the connection between Gazebo and ROS is shown in Figure 16.

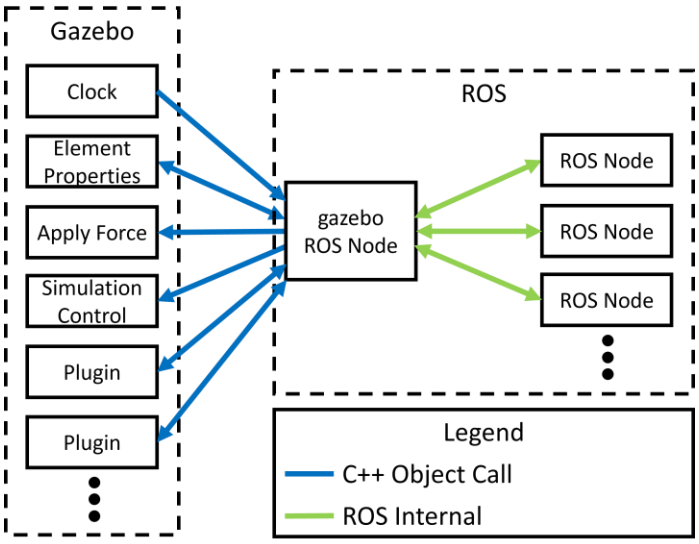


Figure 16. Communication between Gazebo and ROS.

3.7.3 Comparison

Overall, both Unity3D and Gazebo have integrations that enable full control of simulated robots from ROS. Compared to Unity3D, Gazebo has more mature ROS integration with features like the ability to pause and reset the simulated world with ROS services that are not possible to implement within the current ROS-Unity3D integration framework. Furthermore, the ROS-Gazebo integration functions via direct object calls whereas Unity3D requires a TCP connection. One advantage of the approach taken by Unity3D is that it is possible to run ROS and the Unity3D simulation on separate computers sharing a network connection, but in general, the TCP connection will consume a larger amount of computer resources and introduce additional latency. The wide variety of pre-existing Gazebo plugins simulating ROS compatible sensors and drivetrains is a benefit in Gazebo's favor; however, this is partially offset by the ease with which Unity3D custom scripts can be created.

CHAPTER 4:
SIMULATION ENVIRONMENTS

In the previous chapter, Unity3D and Gazebo were compared at an architectural level, both as general simulators and as ROS-based robotic simulators. While these comparisons are valuable for understanding how simulations created in Unity3D and Gazebo may differ from one another on a theoretical level, they cannot demonstrate how these differences will manifest in practical applications. To begin to understand how differences between the simulators affects simulations of autonomous ground robots, several simulated environments, and an autonomous mobile robot capable of navigating through them, were constructed identically in both ROS-Unity3D and ROS-Gazebo. This chapter discusses the construction of the ground robot and its sensors, the ROS stack used to implement autonomous navigation, and the environments.

4.1. Jackal UGV

The ground robot used in the simulated environments is the Jackal UGV, a small four-wheeled, all-terrain, unmanned vehicle manufactured by Clearpath Robotics. The simple and flexible design of the Jackal UGV makes it a good representative of this category of ground robots. Beyond that benefit, the Jackal UGV is also used because it fully integrates with ROS and has a URDF model provided by Clearpath Robotics [97]. The URDF model of the Jackal UGV is highly detailed, configurable with a wide variety of sensors, and supported in Gazebo [98]. Thus, the Jackal UGV comes fully integrated and ready to simulate in ROS-Gazebo with no additional development required. This is not the case with ROS-Unity3D, since Clearpath Robotics does not provide resources for simulating the Jackal UGV in other simulators.

Consequently, the Jackal UGV must be manually integrated into Unity3D-ROS. The first step in this process is importing the Jackal UGV URDF file using the URDF Importer Unity3D package. The sensor configuration of the imported Jackal UGV URDF file includes the Bumblebee2 stereo camera system, the Velodyne VLP-16 LiDAR sensor, and a generic IMU sensor. The URDF Importer package imports the Jackal UGV as a hierarchical collection of GameObjects based on the elements within the URDF file. The chassis and wheels of the Jackal UGV are automatically imported as GameObjects with renderer, articulation body, and collision components based on the configurations in the URDF file. The friction properties of the wheel surfaces were not automatically imported, so they were manually assigned a material with static and dynamic friction coefficients of 0.5 to match the definition within the URDF file. The Jackal UGV as imported into Unity3D is shown in Figure 17 with each of the major components labeled. The joints between the chassis and wheels are created as articulations within Unity3D. The stereo camera system, LiDAR, and IMU sensors are also imported as GameObject children of the chassis, but since Unity3D has no built-in components to simulate sensors they are non-functional. Consequently, custom sensor implementations were required to emulate the functions normally provided by Gazebo plugins. The general goal of the custom sensor implementations is to use the same topics to communicate with ROS so that the same ROS architecture can be used between ROS-Unity3D and ROS-Gazebo.

4.1.1 Drivetrain

One of the sensor types implicitly included in the Jackal UGV is the wheel encoder. Wheel encoders are used to give feedback to ROS on the position and velocity of the wheels. They are generally used to estimate the distance traveled using wheel odometry. Wheel encoders make up a portion of the drivetrain which itself is responsible for the interface between the

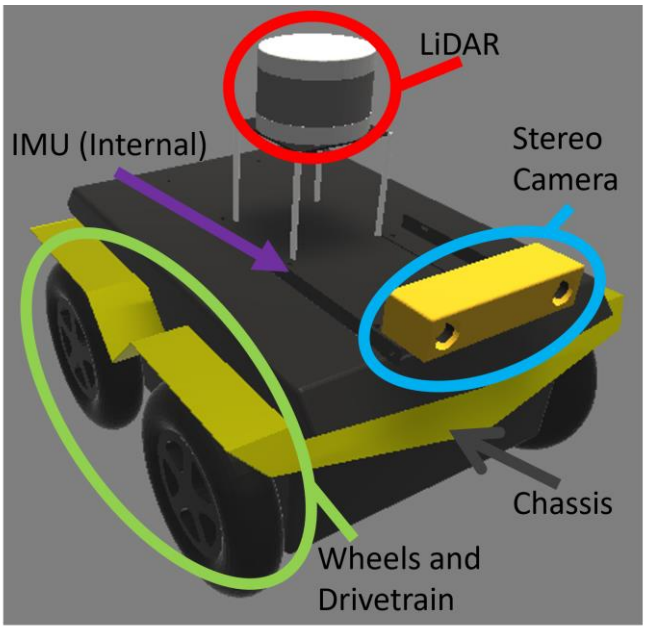


Figure 17. The Jackal UGV in Unity3D with labeled components.

wheels and ROS. In ROS-Gazebo, the entire drivetrain is modeled by a single plugin, the differential drive controller plugin. This plugin subscribes to linear and angular velocity commands from ROS on the "/cmd_vel" topic and converts them into wheel velocities using standard differential drive kinematics equations based on the Jackal UGV drivetrain parameters listed in Table 2. The wheel velocities are then used to apply forces to the wheel joints using ODE motors as described in the physics section in the previous chapter. The plugin simulates wheel encoders by directly publishing a joint state message containing the simulated position and velocity of the joints linking the chassis and the wheels to the ROS "/joint_states" topic.

TABLE 2. JACKAL UGV DRIVETRAIN PARAMETERS

Wheel Radius	0.098 m
Wheel Width	0.040 m
Wheelbase Width	0.262 m
Wheelbase Length	0.376 m
Maximum Velocity	2.0 m/s
Drive Power	500 W

In Unity3D, the functionality of the differential drive controller was implemented in separate components for controlling the wheels and providing wheel encoder feedback to increase modularity. The component responsible for controlling the wheels is called the UGV controller, which is a modified version of the AGV controller script from [95] intended for use with 4-wheeled ground robots. The UGV controller component is placed on the root Jackal UGV GameObject and is configured with references to the four wheel articulations via the inspector window in the Unity Editor. Once the simulation starts, the UGV controller subscribes to the "/cmd_vel" ROS topic and sets up a callback function. Whenever an incoming linear and angular velocity command is received, the function stores it. Then, every fixed update step the stored command velocities are converted into wheel velocities using the same differential drive kinematics equations used by the differential drive controller. The calculated wheel velocities are applied as the target velocities of the four wheel articulations, which realistically apply forces to accelerate the wheels based on the Jackal's drive power.

The publishing of wheel encoder feedback is performed by the joint state publisher component. This component is loosely based on the SourceDestinationPublisher script used for publishing the articulation states of a robotic arm in the Unity Robotics Hub pick and place tutorial [99]. This component is placed on the root Jackal UGV GameObject. From there it can access the properties of all four wheel articulation components. During simulation variable time step updates, the joint state publisher accesses the current position and velocity of each wheel articulation and stores all four positions in an array and all four velocities in an array. After that, a ROS joint state message is created consisting of a numbered and time-stamped header, an array containing the names of the articulations, and the position and velocity arrays. As soon as the

joint state message is created, it is published to the ROS "/joint_states" topic through the ROSConnection object.

4.1.2 Stereo Camera

The simulated Jackal UGV is equipped with a front mounted Teledyne FLIR Bumblebee2 08S2 to stand in as a generic stereo camera system. The Bumblebee2 08S2 uses two high resolution camera sensors each with a maximum resolution of 1032 x 776 pixels and a maximum frame rate of 20 FPS [100]. In ROS-Gazebo, the stereo camera is simulated using the multicamera library configured via the point grey camera description package [101]. This configures the cameras with a resolution of 1024 x 768 pixels, a framerate of 20 FPS, a 66° horizontal field of view, and Gaussian sensor noise with a mean of 0 and a standard deviation of 0.007. Images from the left and right cameras are published to the "/front/left/image_raw" and "/front/right/image_raw" topics, while information about the camera distortion parameters are published on the respective "camera_info" topics.

To model this in ROS-Unity3D, two new GameObjects were created as children of the Jackal UGV chassis link. This ensures that the GameObjects will move with the chassis as it moves. The GameObjects were given identical camera components, then translated to the locations of the two camera sensors within the Bumblebee2 model. The camera components were created with the same 66° horizontal field of view as the real stereo camera sensors. While Unity3D camera components do capture images of the environment, they are not capable of transmitting these images through the ROSConnector object on their own. To handle the transmission of rendered images through the ROSConnector to ROS, the camera sensor script was created. One camera sensor script component is placed on each of the previously placed GameObjects. When the script is first started, it initializes its copy of the ROSConnector object

and uses it to register publishers for the “image_raw” and “camera_info” topics. Next, it initializes a Unity3D RenderTexture with a size of 1024 x 768 pixels that will be used to store the camera image before it is transmitted to ROS. During variable time step updates, the script first checks to see if more than 0.05 seconds have elapsed since the last camera image was transmitted, since this is the period between images at 20 FPS. If so, the camera is used to render an image to the RenderTexture. Ultimately, the image data is transmitted as a raw byte array within a ROS image message through the ROSConnector object. Immediately after the image data is sent, a camera info message for the Bumblebee2 is sent containing the same camera distortion parameters as sent by Gazebo, since the cameras are simulated with the same settings. To keep the implementation of the camera sensor script general, the image resolution, frame rate, image raw topic name, camera info topic name, and Gaussian noise are all set as parameters that can be altered from within the inspector window in the Unity Editor.

One interesting challenge faced when implementing the camera sensor script is that there is no built-in support for applying Gaussian noise to a rendered image. One avenue for applying the noise would be to loop through the image one pixel at a time and apply the noise within the existing camera sensor script, but this would be extremely slow, especially for the pair of high-resolution images produced by the Bumblebee stereo camera system. Another option would be to not model camera sensor noise, however this would be unrealistic, since all real-world cameras experience some random noise, and would invalidate the comparison with Gazebo since the multicamera plugin does model noise. The final option was to develop a shader to add noise to the captured images. Since shader programs run on a GPU and are often used for modifying images quickly, this approach would mitigate the slowdown seen in the first approach. In fact, this is the approach used by Gazebo to apply camera sensor noise. The challenge is that shader

programs are written in high-level shader language (HLSL) for Unity3D and OpenGL shader language (GLSL) for Gazebo, both of which are more restrictive and difficult to learn than general programming languages. Fortunately, HLSL and GLSL are structurally similar with primarily syntactical differences, so it was possible to reimplement the Gazebo camera sensor noise GLSL shader as an HLSL shader for Unity3D. This not only enables camera sensors in Unity3D to support sensor noise, but also means that camera sensor noise is modeled identically between Unity3D and Gazebo. Within the camera sensor script, the Gaussian noise shader is set to run every time a new camera image is rendered, adding random Gaussian noise to every pixel in the image according to the configurable mean and standard deviation parameters.

4.1.3 LiDAR

The LiDAR scanner on the Jackal UGV is the Velodyne Puck VLP-16. The VLP-16 is a compact 3D LiDAR scanner equipped with 16 scan lines covering a 360° horizontal field of view and a 30° vertical field of view [102]. In ROS-Gazebo, the VLP-16 is simulated using the Velodyne simulator Gazebo plugin maintained by Dataspeed Inc. The plugin is configured to simulate the full 16 scan line field of view with 900 samples per line at a 10 Hz update rate. The simulated sensor has a minimum range of 0.9 m, a maximum range of 130 m, and Gaussian sensor noise with a standard deviation of 0.008 m. LiDAR scans are published as point cloud 2 messages to the “/mid/points” ROS topic.

The Velodyne LiDAR sensor script was created to model the VLP-16 3D LiDAR scanner in ROS-Unity3D. The sensor script was added as a custom component to the Velodyne GameObject on the Jackal UGV model. When the script first starts at the beginning of the simulation, a ROSConnector object is initialized and used to register a point cloud 2 message publisher for the “/mid/points” ROS topic. During variable time step updates, the script checks to

see if more than 0.1 s have passed since the last point cloud was sent to ROS, since this corresponds to an update rate of 10 Hz. If so, the script begins the process of collecting the next point cloud. All the laser time of flight measurements performed by the lasers in the LiDAR scanner are simulated using Unity3D raycasts. In general, a raycast takes a point and a vector and returns the distance between the point and the next closest object in the direction of the vector. The raycast command schedule batch function is used to efficiently perform the thousands of raycasts required for each sensor update without slowing down the main simulation loop. This function collects all of the raycast commands for a single LiDAR scan and executes them asynchronously in a separate thread. Then, on subsequent variable time step updates, the script queries the thread containing the batched raycast commands to determine if it has finished. When all the commands have completed, the returned distance values are stored in an array. Next, random Gaussian noise is added. Each distance must then be converted from its current spherical coordinate representation with a known distance and vector direction into a point as an FLU cartesian coordinate. After this has been completed, a point cloud 2 message is constructed using the points for each laser measurement and the message is sent to the "/mid/points" ROS topic. One limitation of the current approach is it does not model the returned light intensity so each point in the point cloud message is assigned an intensity of 0. As with the other scripts, the implementation is kept general by using user configurable parameters for the publish period, noise standard deviation, horizontal field of view, vertical field of view, number of lasers, number of samples per laser, and ROS topic name.

4.1.4 IMU

The IMU used on the real Jackal UGV is the 3DM-GX3-25 produced by LORD MicroStrain Sensing Systems [97]. This IMU is an attitude heading reference system (AHRS)

because it not only produces triaxial acceleration and gyroscope measurements, but it is also capable of internally fusing them to produce orientation estimates [103]. In ROS-Gazebo, the IMU is simulated using the GazeboRosImu plugin maintained by Team Hector. Like the real IMU, this plugin publishes an estimated orientation by mimicking a simple AHRS, in addition to publishing acceleration and angular velocity measurements [104]. The plugin is configured to publish IMU measurements at 50 Hz with additive Gaussian noise applied to all axes with a mean of 0 and a standard deviation of 0.005. The IMU is also modeled as slowly drifting over time with a standard deviation of 0.005 and a 1 hr time constant. IMU measurements are published as ROS IMU messages to the "/imu/data" topic.

To model the IMU in ROS-Unity3D, the IMU Sensor script was created. This script was added as a custom component to the internal imu_link GameObject on the Jackal UGV model. At the beginning of the simulation, the script initializes a copy of the ROSConnector object and uses it to register a IMU message publisher on the "/imu/data" ROS topic. At this point, the script also initializes the internal state of a simple AHRS error model based on the open-source AHRS implementation produced by Team Hector. During variable time step updates, the script checks to see if more than 0.02 seconds have passed since the last time an IMU message was published, since this corresponds to an update rate of 50 Hz. If so, the script begins collecting a new IMU measurement. First, the current linear acceleration of the imu_link GameObject to which the script is attached is determined. This is done by dividing the change in velocity by the change in time between two time steps, then factoring in the acceleration due to gravity. The angular velocity is then directly obtained via the angular velocity property of the imu_link GameObject. Subsequently, each of the axes of the linear acceleration and angular velocity are independently corrupted by Gaussian noise and the current IMU sensor drift offset value. The

orientation estimate is then found by corrupting the true imu_link GameObject orientation with the drift from the accelerometer according to the AHRS model. A ROS IMU message is constructed from the corrupted orientation, linear acceleration, and angular velocity values. Lastly, the message is sent to the "/imu/data" ROS topic. The implementation has been kept general by enabling the ROS topic name, publish period, Gaussian noise standard deviation, and drift noise standard deviation parameters to be configured directly within the Unity Editor inspector window.

4.2. ROS Autonomy Stack

While Unity3D and Gazebo allow the Jackal UGV to perceive and act within the simulated environments with its sensors and drivetrain, there is no intelligent connection between the two without ROS. At a high level, the ROS autonomy stack subscribes to sensor information and uses it to perform calculations to determine what specific command to publish to the drivetrain, facilitating the connection between perception and action. The two main tasks of the autonomy stack are SLAM, to determine the location of the robot and to map out the locations of potential obstacles, and navigation, to determine how to reach a destination while avoiding obstacles. The ROS autonomy stack is composed of several interconnected programs called nodes. Aside from the ROS TCP Endpoint node and the Gazebo node used to communicate with the simulators, the autonomy stack is identical between the two simulators, and could even be reused with a real Jackal UGV. All the nodes come from pre-existing ROS packages, specifically the stereo image processing package, the RTAB-Map ROS package, the move base package, and the follow waypoints package. In the following section, the nodes used in the ROS autonomy stack from the aforementioned packages along with the connections between them will be briefly described. Block diagrams illustrating the communication between

the ROS autonomy stack configured for visual SLAM and the Unity3D and Gazebo simulators are shown in Figure 18 and Figure 19, respectively.

4.2.1 Stereo Image Processing

The only node used in the autonomy stack from the stereo image processing package is the stereo image processing node. This node subscribes to the images from the left and right stereo cameras on the "/front/left/image_raw" and "/front/right/image_raw" topics as well as the camera_info messages from each of the "/front/left/camera_info" and "/front/right/camera_info" topics. The information within the "camera_info" messages is used to correct distortion from the particular lens and camera sensor used. Rectified versions of the images with the distortion removed are published to the "/front/left/image_rect_color" and "/front/right/image_rect_color" topics. After rectifying the images, the stereo image processing node interprets the disparities between the two images to calculate depths, resulting in a point cloud. The point cloud is published to the "/points2" topic. A stereo image point cloud from the agriculture environment is shown in Figure 20 below.

4.2.2 SLAM

In the current ROS autonomy stack, SLAM is implemented using several nodes from the RTAB-Map ROS package. RTAB-Map, an abbreviation for real-time appearance-based mapping, is an RGB-D, stereo, and LiDAR SLAM approach for 3D environments based on appearance-based loop closure that has been integrated into ROS through the RTAB-Map ROS package. Previous research has compared RTAB-Map to other SLAM approaches in both simulated and real-world environments and found that it compares favorably, especially in outdoor environments [26], [105], [106]. This thesis considers two primary RTAB-Map configurations: visual-based SLAM and LiDAR-based SLAM. Both approaches use the same

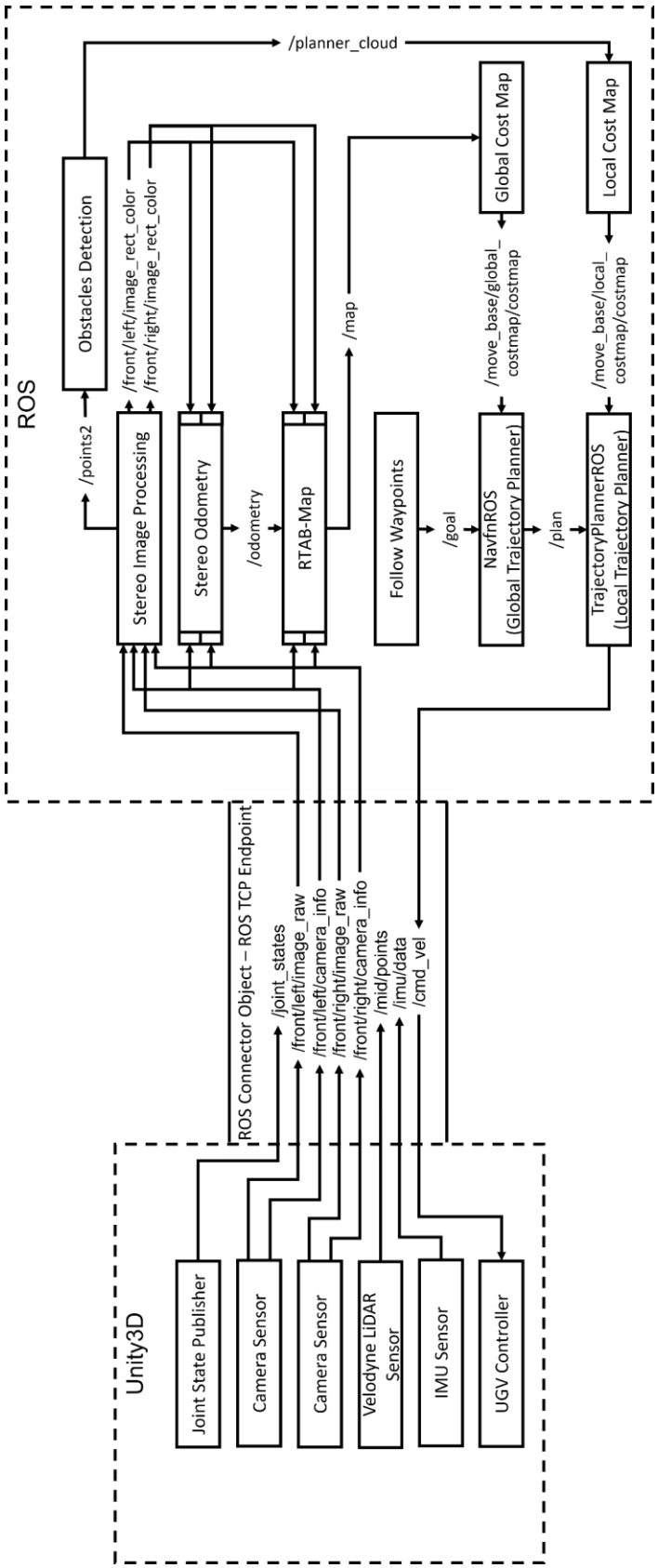


Figure 18. Communication with the ROS autonomy stack in ROS-Unity3D.

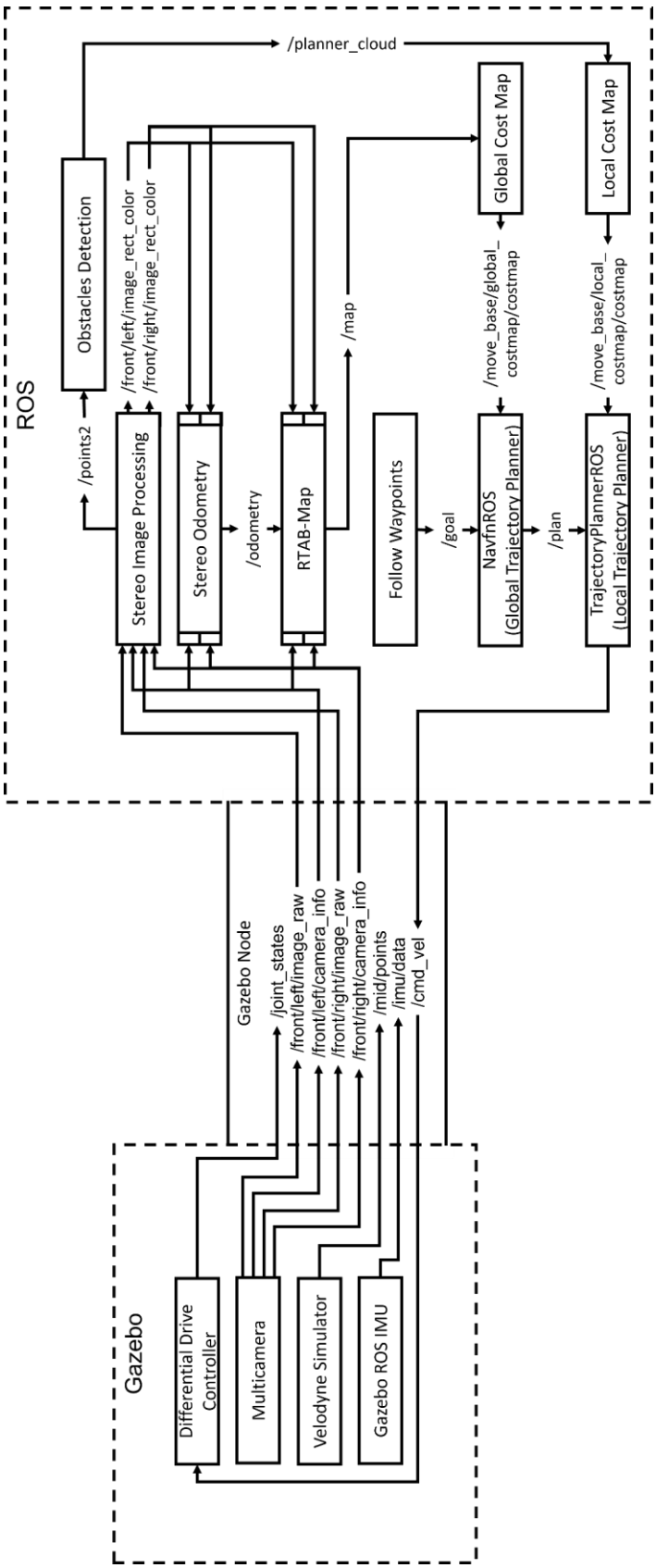
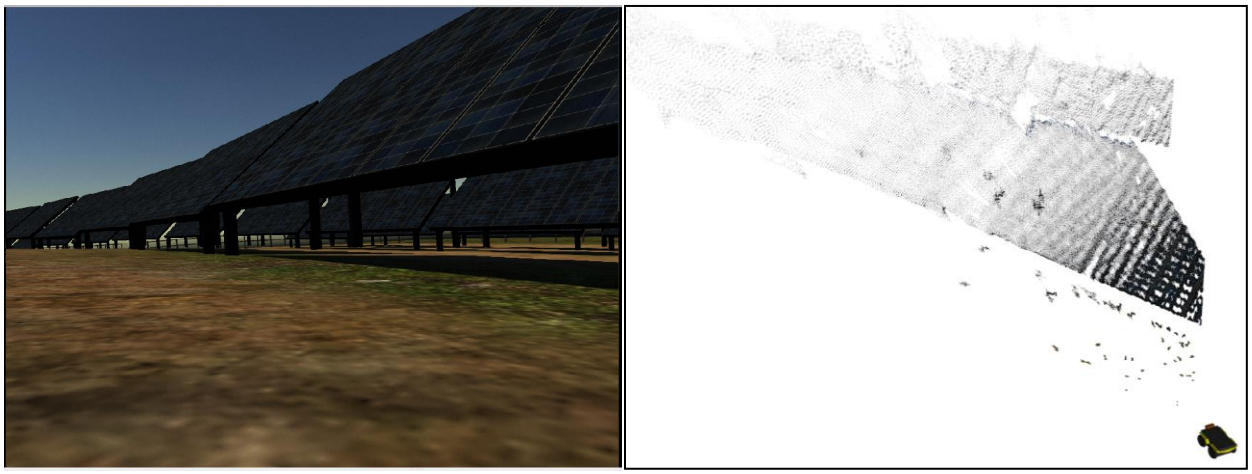


Figure 19. Communication with the ROS autonomy stack in ROS-Gazebo.



(a) (b)

Figure 20. Stereo image processing (a) input from left camera and (b) point cloud output in the agriculture environment simulated in Unity3D.

general method of obstacle detection[26], [105], [106], but differ in how odometry, global localization, and global mapping are performed.

Within the RTAB-Map ROS package, obstacle detection is performed by the obstacles detection node. This node subscribes to a topic containing point cloud messages, "/points2" in visual SLAM and "/mid/points" in LiDAR SLAM, and separates the points representing obstacles from the points representing navigable ground. The classification is performed by clustering points and estimating the normal vector of the cluster. If the normal angle deviates from vertical by more than a set value, the points in the cluster are considered obstacles, otherwise they are considered navigable ground. Detected obstacles are published to the "/planner_cloud" topic for the navigation local planner. This node is used in its default configuration with a minimum cluster size of 20 and a maximum deviation from vertical of 45°.

For visual-based SLAM, odometry is performed using the "stereo_odometry" node. This node subscribes to the left and right rectified image topics and their respective camera info topics. Visual features are extracted from each stereo image pair and are used to calculate the distance to the features in the left image. For each new image pair received, the features in the

left image are compared to those in the previous image using a random sample consensus approach to estimate the relative position of the camera compared to when the last image was received. This estimated relative position is used to publish the current estimated position as a ROS odometry message to the "/odometry" topic. This node also provides the transform between the "odom" coordinate frame and the "base_link" frame of the Jackal UGV.

Global localization and mapping are both performed by the main "rtabmap" node. This node always subscribes to the "/odometry" topic published by whichever odometry system is in use. In visual-based SLAM, this node also subscribes to the left and right rectified image and camera info topics. As the robot traverses and images are collected, the "rtabmap" node incrementally builds a full 3D map of the environment. It does so by positioning each stereo image point cloud it receives according to the estimated Jackal UGV position from the odometry topic. To reduce error, by default only points within 5 m of the sensor are saved to the map. As in the obstacles detection node, the points making up the map of the environment are segmented as either an obstacle or as ground. While the obstacles detection node only publishes obstacles that are currently in view of the sensors for local navigation, the "rtabmap" node publishes a 2D projection of the full environmental map as an occupancy grid message to the "/map" topic for the global navigation system. The occupancy grid is filled such that any vertical column of the environmental map containing an obstacle point is projected as an occupied cell in the 2D occupancy grid, vertical columns containing only ground points are projected as free cells, and columns containing neither are projected as unknown cells. Global localization is performed by comparing the current image from the camera with images collected earlier and stored in the RTAB-Map database to decide if they represent the same location. If the discrepancy, or error, between the images is below a certain threshold, a loop closure is performed which updates the

current position estimate as well as the estimate of the previous trajectory. To provide global localization within the environmental map, the “rtabmap” node publishes the transform between the “map” coordinate frame and the “odom” frame, which combined with the “odom” to “base_link” transform from the odometry node fully localizes the Jackal UGV within the map.

For LiDAR-based SLAM, odometry is performed using the “icp_odometry” node. ICP, or iterative closest point, is an algorithm to minimize the difference between point clouds by iteratively transforming the position of one relative to another. In the context of LiDAR-based odometry, the transform of the previous LiDAR point cloud that leads to the best match with the current LiDAR point cloud is the same as the transform of the robot between the scans, assuming a static environment. Thus, the “icp_odometry” node subscribes to LiDAR point clouds on the “/mid/points” topic and publishes position estimates as ROS odometry messages on the “/odometry” topic. Like the stereo odometry node, it also provides the transform between the “odom” coordinate frame and the “base_link” frame of the Jackal UGV. With respect to global localization and mapping, the main change when performing LiDAR-based SLAM is that the “rtabmap” node subscribes to the “/mid/point” topic for LiDAR point clouds instead of the stereo image topics. This means that the map of the environment is created using the LiDAR data. Additionally, the strategy parameter is changed from the default value of 0 for visual SLAM to 1 for ICP-based SLAM. The consequence of this parameter change is that RTAB-Map cannot perform global loop closure using LiDAR point clouds alone.

Note that the ROS autonomy stack is configured to calculate odometry using only visual odometry methods or point cloud ICP odometry. Despite both Unity3D and Gazebo modeling wheel encoder feedback and IMU measurements, neither of these are used for odometry for two main reasons. First, neither wheel encoder feedback nor IMU measurements are natively

supported by nodes within the RTAB-Map ROS package, so this would have required additional integration with outside packages. Second, limiting the autonomy stack to visual and LiDAR-based SLAM methods more specifically assesses the impact of the visual differences between Unity3D and Gazebo, which is a comparison that was not possible to make based on their architectures alone.

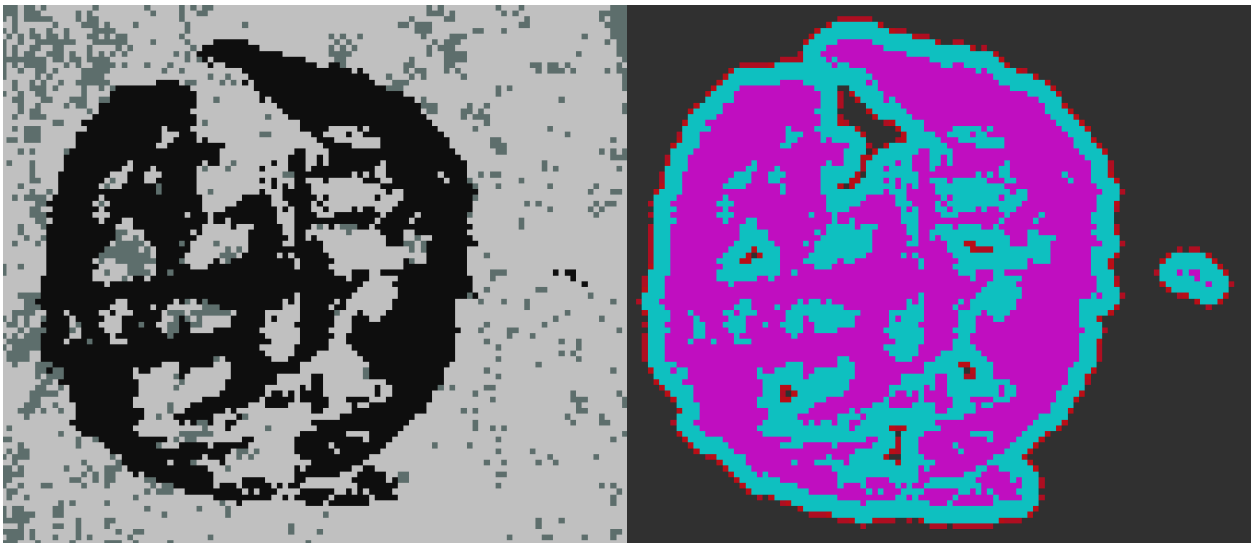
4.2.3 Navigation

The navigation portion of the ROS autonomy stack is implemented using several nodes from the move base ROS package. At a high level, the purpose of the navigation subsystem is to find the best sequence of velocity commands to publish to reach a desired goal location. The situation is complicated by obstacles within the environment which restrict the number of possible traversable paths. Thus, the navigation subsystem contains cost maps to catalog the extent to which different portions of the terrain are traversable. Trajectory planners are then used to find a path through the cost map that minimizes the overall cost. As is typical for navigation systems using move base, this implementation uses a global cost map, a local cost map, a global trajectory planner, and a local trajectory planner.

In general, the cost maps keep track of occupied and clear space within a 2D occupancy grid populated with information from sensors or other existing maps. The cost of traversing through different parts of the occupancy grid is calculated by inflating the size of the obstacles by a circumscribed radius of the Jackal UGV, plus a user defined constant. As the inflation distance from the obstacle increases, the cost of traversal decreases until it reaches 0 outside the inflation radius. Both cost maps are implemented using ROS cost map 2D nodes which simplifies navigation compared to a full 3D representation of the environment. Of the two cost maps used, the global cost map is simpler because it only subscribes to and inflates the SLAM

occupancy grid published by the “rtabmap” node on the “/map” topic. The inflated cost map is published as an occupancy grid message to the “/move_base/global_costmap/costmap” topic. An image showing the SLAM occupancy grid around an obstacle and its corresponding inflated cost map is shown in Figure 21. The local cost map subscribes to the “/planner_cloud” topic from the obstacles detection node and uses it to populate a 3 x 3 m occupancy grid centered on the the Jackal UGV. The local cost map is published to the “/move_base/local_costmap/costmap” topic. A separate local cost map is used because its smaller area allows it to use smaller grid cell sizes than the global cost map. This allows more granular decision making at the local level. For this research, the global cost map uses a resolution of 0.05 m for consistency with RTAB-Map while the local cost map uses a resolution of 0.01 m. Both cost maps use a user defined inflation parameter of 0.2 m.

The trajectory planners each subscribe to their respective global or local cost map. The global trajectory planner, implemented as a NavfnROS node, also subscribes to the high-level navigation goal on the “/goal” topic. This is used to create a simplified 2D navigation plan from the current location to the goal. The NavfnROS node minimizes the cost of the navigation plan using Dijkstra’s algorithm. For the purposes of navigation, unknown cells in the occupancy grid are treated the same as known clear cells. The navigation plan is published as a ROS path message to the “/move_base/NavfnROS/plan” topic. The local planner, a TrajectoryPlannerROS that should be executed by the drivetrain to follow the path to the goal. The velocity commands are published as linear and angular velocities to the “/cmd_vel” topic. The local planner has several configurable parameters, such as acceleration limits, to account for differences between node, subscribes to the plan published by the global planner to determine the velocity commands the locomotive capabilities of different drivetrains. For the Jackal UGV, the parameters used in



(a) (b)

Figure 21. Obstacle shown as in (a) SLAM occupancy grid and (b) navigation cost map. In the occupancy grid, black represents occupied, light gray is clear, and blue-grey is unknown. In the cost map, pink is an obstacle, blue is Jackal UGV inflation radius, red is the user inflation radius.

the Jackal Navigation package provided by Clearpath Robotics are reused except for the maximum linear and angular velocity parameters. The maximum linear velocity has been halved from 0.5 m/s to 0.25 m/s and the maximum angular velocity has also been halved from 1.57 rad/s to 0.79 rad/s because RTAB-Map can lose odometry if the Jackal UGV moves too quickly.

4.2.4 Waypoint Following

The final package used in the ROS autonomy stack is the follow waypoints package which implements the “follow_waypoints” node. This node is needed because the navigation subsystem can only accept a single navigation goal at a time, so attempting to provide a second goal will replace the first goal, even if it has not been completed yet. In general, the “follow_waypoints” node buffers a sequence of navigation goals, only publishing the next goal when the previous one has been completed. For the ROS autonomy stack, the “follow_waypoints” node loads a sequence of navigation goals from a file and publishes them one at a time in sequence to the “/goal” topic. Combined with the other elements of the autonomy stack, this allows the Jackal UGV to autonomously navigate between multiple points.

Since the goals are read from a file, repeatable experiments can be designed to assess the ability of the Jackal UGV to accurately map and navigate through each environment in both Unity3D and Gazebo.

4.3. Environments

The environment consists of everything that the Jackal UGV interacts with in simulation. To obtain a diverse set of interactions and thoroughly evaluate the practical differences between each simulator, the autonomous Jackal UAV is simulated in five different outdoor unstructured environments each representing potential use cases for autonomous ground robots. Three of the environments are based on Gazebo worlds provided by Clearpath Robotics for simulating their robots, including the Jackal UGV. In these cases, any modifications to the original Gazebo world file will be mentioned. For Unity3D, these environments are constructed to match the Gazebo world as closely as possible, if not identically, by importing and arranging the 3D mesh files referenced in the world file. The remaining two environments are constructed in an external 3D modeling program and are saved in the Collada format which is compatible with both Unity3D and Gazebo. Beyond presenting additional unique use cases for autonomous ground robots, these environments also can be considered to present a more neutral comparison since they were not specifically constructed for either simulation suite, whereas the previous environments were originally intended for use with Gazebo exclusively. The remainder of this section presents the five simulation worlds, the steps for using them in the simulators, and the processes for creating the worlds, if applicable.

4.3.1 HRATC World

The first environment is the HRATC world included in the jackal_gazebo package provided by Clearpath Robotics [98]. This world was originally provided for use in the 2017

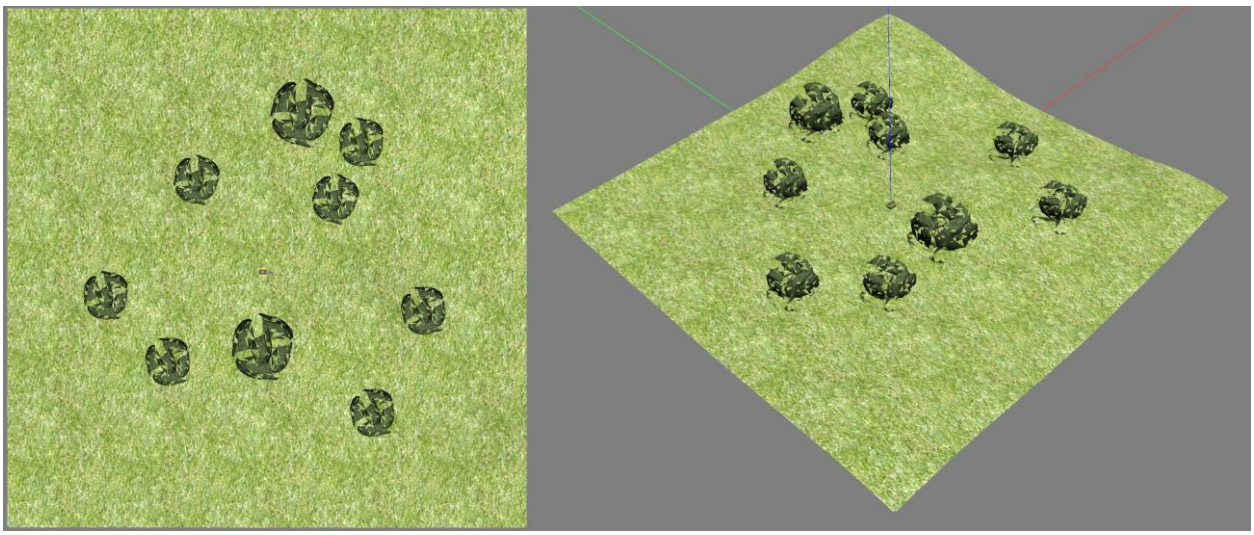
IEEE Robotics & Automation Society Humanitarian Robotics & Automation Technology Challenge (HRATC), the goal of which was to develop new strategies for autonomous landmine detection [107]. The environment itself is rather simple consisting of a small undulating landscape with a noisy green texture resembling grass with a few sparsely spaced trees. The landscape is 30.48 x 30.48 m in size. Between the landscape and the trees, the model file contains 18,771 mesh triangles and with textures requires 1.23 MB of disk space. The exact same model file and textures are used in both Unity3D and Gazebo

Since this the provided Gazebo world file is dim by default, the ambient and background lighting in the scene was manually increased from 50% to 100%. All the Gazebo environments, including the HRATC world, use the time and physics parameters provided by Clearpath Robotics as listed in Table 3, unless otherwise noted. At the start of simulation, the Jackal UGV is placed at the center of the terrain with a height of 1. The Gazebo HRATC world is pictured in Figure 22 with the Jackal UGV placed in its starting position.

TABLE 3. GAZEBO ENVIRONMENT TIME & PHYSICS PARAMETERS

Parameter Name	**Value**
Max Step Size	0.001 s
Real-Time Factor	1.0
Real-Time Update Rate	1000 Hz
Physics Engine	ODE
Gravity	-9.81 m/s^2
Error Reduction Parameter	0.2
Contact Max Correcting Velocity	100 m/s
Contact Surface Layer	0.001 m

The first step towards recreating the HRATC world in Unity3D was importing the 3D meshes and textures of the landscape and trees into a new project. All the meshes were Collada files while the textures were a mix of PNG and JPEG image files. To simplify this process, the entire directory structure containing the meshes and textures was imported into Unity3D by



(a) (b)

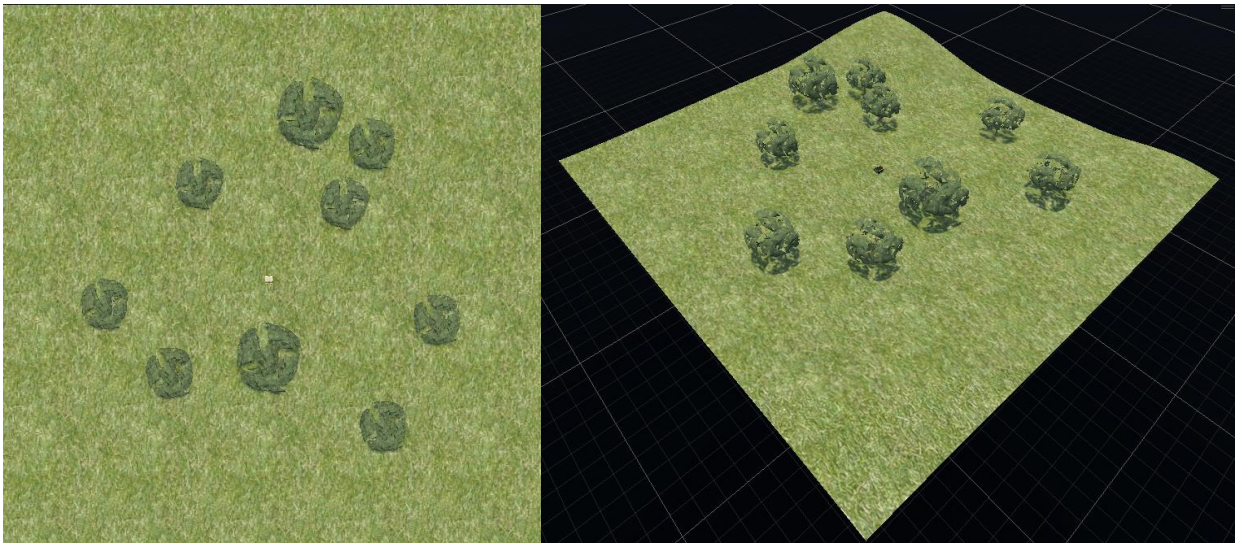
Figure 22. The HRATC world simulation environment in Gazebo (a) top-down orthogonal view and (b) angled perspective view.

dragging the main folder into the project window. Subsequently, the imported landscape and trees were dragged from the project window into the scene window to place them in the simulated world. Since the mapping between the pixels of the texture files and the faces of the 3D object mesh is stored in Collada files, the textures for the landscape and trees were automatically applied when placed in the simulated world. The relative positions of the 3D objects in the HRATC environment are stored in the Gazebo world file, so the landscape and trees had to be manually positioned in Unity3D. Fortunately, since the world file follows the SDF format, the exact position and rotation of the landscape and trees could be directly accessed and applied to their Unity3D counterparts via their transform components in the inspector window. However, it was necessary to manually account for the differences between the FLU and RUF coordinate systems when positioning the trees. Next, to ensure that the Jackal UGV would not fall through the terrain or pass through the trees, each imported mesh required a mesh collider component. Since it would be tedious to manually apply mesh collider components to the landscape and nine trees through the Unity Editor, a short script called AddColliders was written to automatically loop through the meshes making up the environment and apply mesh

colliders when the simulation starts. A Unity3D GameObject with a directional light component was added to the scene, configured with a light intensity of 100% and oriented identically to the Gazebo directional light component as defined in the world file. All Unity3D environments, including the HRATC world, use the default Unity3D time and physics values listed in Table 4, unless otherwise noted. The Jackal UGV was positioned at the origin of the horizontal plane with a height of 1 m to match Gazebo. The Unity3D HRATC world is pictured in Figure 23 with the Jackal UGV placed in its starting position at the center of the landscape.

TABLE 4. UNITY3D ENVIRONMENT TIME & PHYSICS PARAMETERS

Parameter Name	**Value**
Fixed Time Step Length	0.01 s
Time Scale	1
Maximum Delta Time	0.333 s
Gravity	-9.81 m/s^2
Global Contact Offset	0.01 m



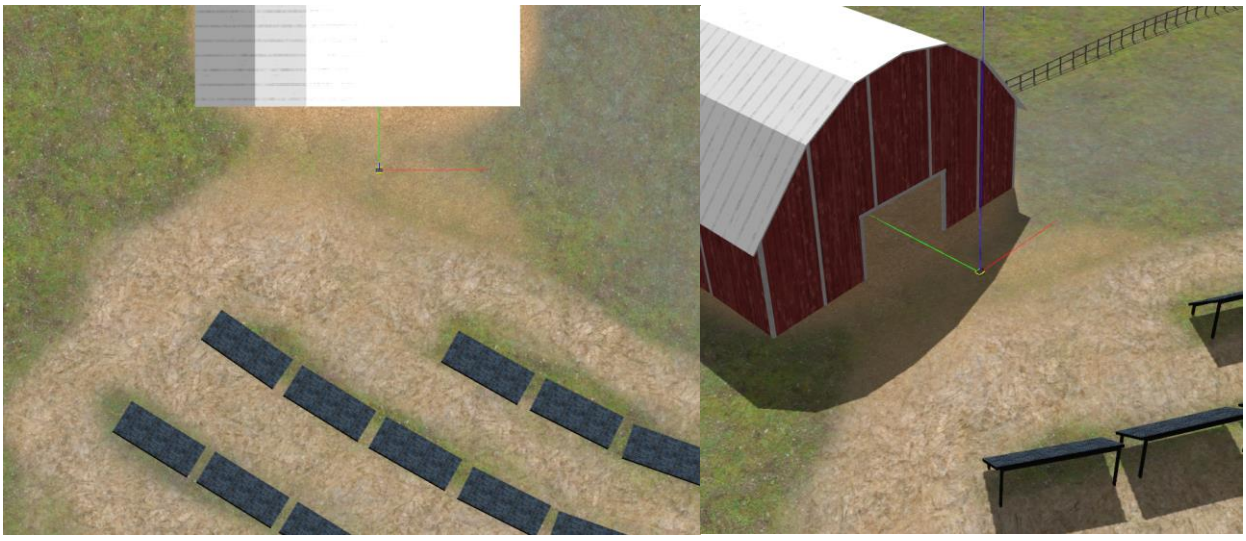
(a) (b)

Figure 23. The HRATC world simulation environment in Unity3D (a) top-down orthogonal view and (b) angled perspective view.

4.3.2 Agriculture World

The second environment is the agriculture world from the Clearpath Robotics "cpr_gazebo" package [108]. The environment consists of a medium-sized, mostly flat landscape with a barn, fences, and a solar array. The solar array consists of 43 identical solar panels spaced in gently curving rows. As indicated by the name, the world is intended to be used for simulating ground robots in agricultural applications, as well as in more specialized tasks like solar panel inspection. The landscape is roughly 160 x 120 m. In total, the environment is made up of 17,820 mesh triangles and requires roughly 42.4 MB of disk space with 39.7 MB attributed to the six high-resolution image files for texturing the mesh.

In the Gazebo world file, lighting is applied from a singular directional light source nearly directly overhead with an intensity of 100%. At the start of simulation, the Jackal UGV is placed on the terrain in front of the open barn door facing such that the barn is on its left and the panels are on the right. The Gazebo agriculture world is pictured in Figure 24 with the Jackal UGV placed in its starting position.



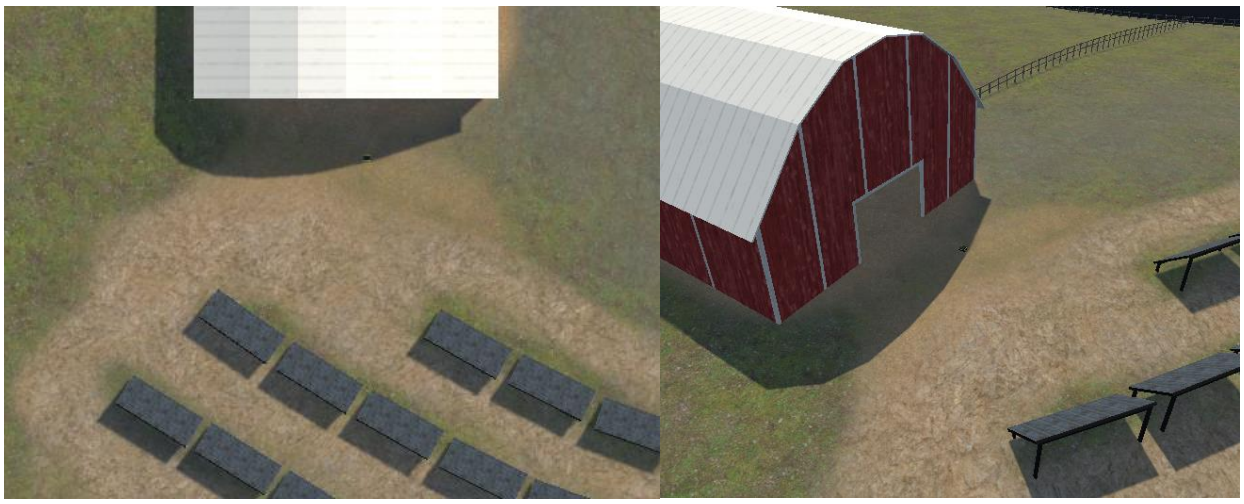
(a) (b)

Figure 24. The agriculture world simulation environment in Gazebo (a) top-down orthogonal view and (b) angled perspective view.

As with the HRTAC world, the first step towards recreating the agriculture world in Unity3D was importing the 3D mesh and textures referenced in the Gazebo world file into a new project. Fortunately, the agriculture world only consists of a single Collada mesh file, itself containing several nested meshes, and six JPEG texture files which are fully compatible with Unity3D. As before, the entire folder containing the meshes and textures was imported at once. Dragging the imported mesh into the Unity3D scene placed the full textured environment, including the landscape, barn, fences, and solar panels, into the simulated world simultaneously. Unity3D defaults to importing textures with a maximum resolution of 4096 x 4096 pixels which meant that the 8192 x 8192 landscape texture was down sampled and appeared blurry when applied to the terrain. After the maximum resolution was manually increased to 8192 x 8192 pixels and the texture was reimported, the landscape received the full resolution texture. A comparison between the textured landscape at the two resolution levels and in Gazebo is shown in Figure 25. Since the full environment with all objects already positioned relative to one another was contained in a single mesh file, no objects had to be manually adjusted. Colliders were added to the landscape, barn, fences, and solar panels using the same AddColliders script developed for use with the HRATC world. A Unity3D GameObject with a directional light component with an intensity of 100% was added to the scene and oriented identically to the Gazebo directional light component as defined in the world file. Similarly, the Jackal UGV was positioned to start in the same position and orientation defined in the Gazebo world file. The Unity3D agriculture world is pictured in Figure 26 with the Jackal UGV placed in its starting position.

(a) (b) (c)

Figure 25. Close-up comparison between textured landscapes (a) Unity3D at default import resolution, (b) Unity3D at full import resolution, and (c) Gazebo reference.



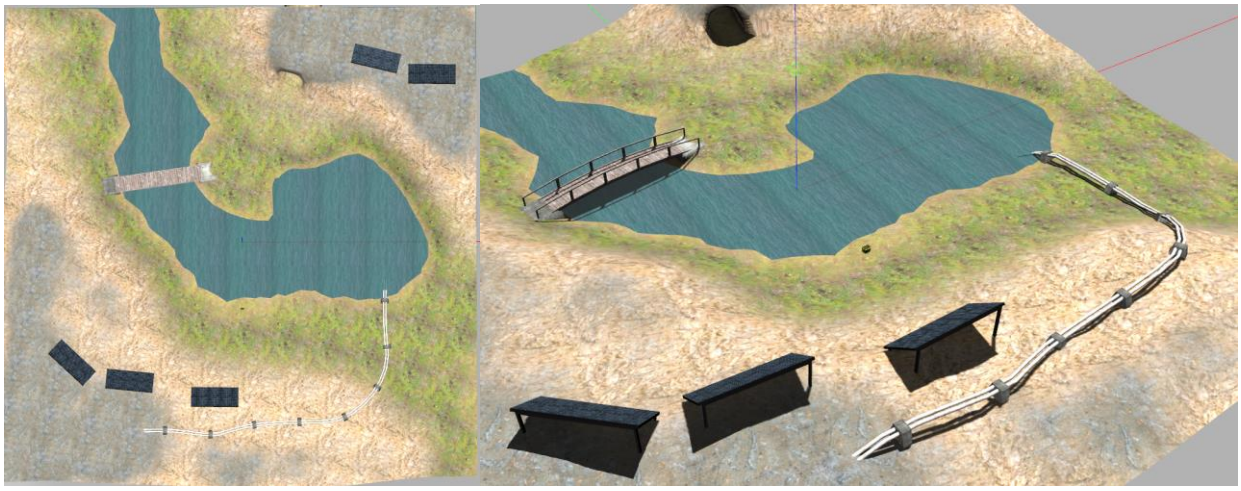
(a) (b)

Figure 26. The agriculture world simulation environment in Unity3D (a) top-down orthogonal view and (b) angled perspective view.

4.3.3 Inspection World

The inspection world is another environment from the Clearpath Robotics "cpr_gazebo" package [108]. The environment consists of a medium-sized hilly landscape with a small pond, a bridge, five solar panels, and two parallel water pipes. The inspection world is intended to simulate ground robots used for inspecting environmental conditions and remote equipment. The landscape is roughly 60 x 60 m and features slopes as steep as 25° from horizontal. In total, the environment is made up of 5,767 mesh triangles and requires 47.4 MB of disk space, with 46.6 MB attributed to nine high resolution texture files.

In Gazebo, lighting is applied from two directional light sources. The first light is placed above the terrain to direct lighting to the environment while second is placed below the environment to provide additional indirect background lighting to the environment. Both are configured with an intensity of 100%. The surface of the pond is simulated as a flat plane with a water texture provided by the Gazebo unmanned underwater vehicle simulator plugin [109]. As a ground vehicle, the Jackal UGV passes through the plane of the water and is not affected by it in any way. By default, the position of the Jackal UGV at the start of the simulation is on a slope beside the pond, 5 m in the air above the terrain. Since this is unrealistic and often causes the Jackal UGV to fall into the pond, the starting location was adjusted to just on the surface on a flat area on the bank of the pond. The Gazebo inspection world is pictured in Figure 27 with the Jackal UGV placed in its updated starting position.

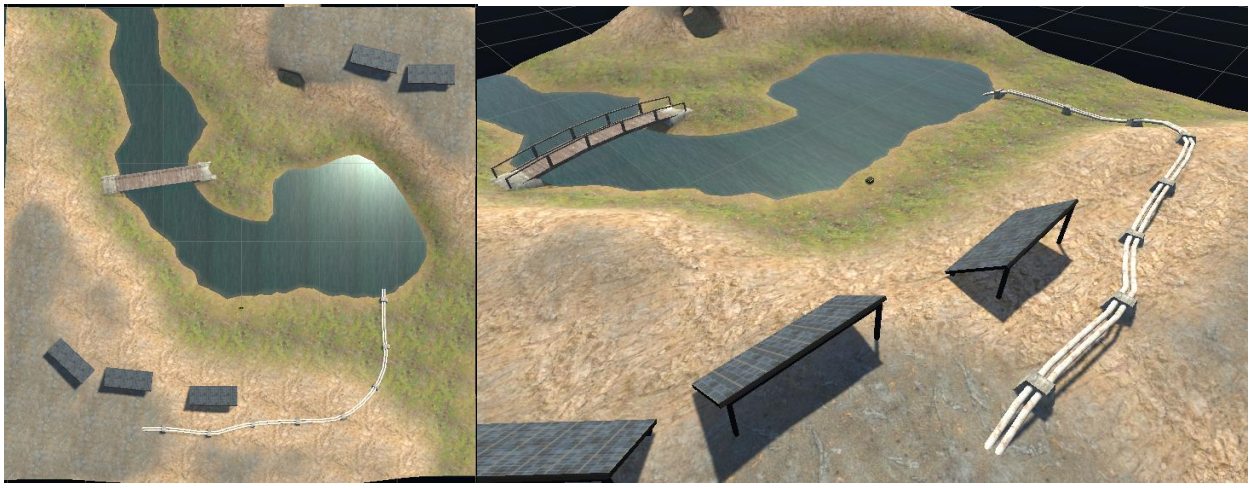


(a) (b)

Figure 27. The inspection world simulation environment in Gazebo (a) top-down orthogonal view and (b) angled perspective view.

As with the prior two environments, the folder containing the meshes and textures for the inspection world was imported into a new Unity3D project by dragging it into the Unity Editor project window. The folder contained a single Collada mesh file and nine JPEG texture files. Beyond the files directly included with the inspection world, an additional texture

“water_water_0076_03_s.jpg” was imported from the uuv_gazebo_worlds package since this is the water texture used by the unmanned underwater vehicle simulator plugin in the Gazebo inspection world [109]. Of the ten total texture files, one had a resolution above 4096 x 4096 requiring that its import resolution be increased to 8192 x 8192 pixels. Colliders were added to the landscape, bridge, solar panels, and pipes using the AddColliders script previously mentioned. To model the surface of the pond, a Unity3D GameObject with a plane mesh filter component and mesh renderer component were added to the scene and sized to cover the extent of the pond at a height of 0, exactly as in Gazebo. The water texture image was used to create a new Unity3D material with an opacity of 80% which was subsequently applied to the plane GameObject, making it match the appearance of the water surface in Gazebo. To match the lighting, two directional light components, each with an intensity of 100%, were created and positioned above and below the terrain as in the Gazebo world. Similarly, the Jackal UGV was positioned to start in the same adjusted position and orientation on the surface beside the pond. The Unity3D agriculture world is pictured in Figure 28 with the Jackal UGV placed in its starting position.



(a) (b)

Figure 28. The inspection world simulation environment in Unity3D (a) top-down orthogonal view and (b) angled perspective view.

4.3.4 University of Alabama Science and Engineering Quad

The next environment is a full-scale model of the University of Alabama Science and Engineering Quad (UA SEQ). The environment consists of a large, flat grassy quadrangle broken up by sidewalks, trees, bushes, and a fountain all surrounded by several tall buildings. This world could be used for simulating ground robots used for deliveries, transportation, and landscaping. This environment is also unique in that it represents a real location that a real Jackal UGV could navigate through for comparison with the results from both simulators. The model was created externally in Blender, an open-source 3D creation suite. The data used for creating the model, including the textures and mesh, was imported from the publicly available model of the University of Alabama campus on Google Maps using the Maps Model Importer Blender add-on [110]. After the model was imported from Google Maps, it was simplified by removing portions of the terrain that would not be visible from within the quadrangle, combining all objects into a single mesh object, and joining overlapping vertices. Next, simplified lamp post models were added to increase the realism of the environment, since they were not included in the imported model. The simplified lamp posts were constructed in Blender from two black cylinders representing the post and lamp components. After the edits, the UA SEQ model was exported as a Collada file and numerous image files. The environment as exported is roughly 575 x 340 m, is made up of 330,374 mesh triangles, and requires 76.1 MB of disk space, including 220 PNG texture files totaling 21.9 MB.

Since the UA SEQ model was created in external 3D modeling software, it required manual integration with both Gazebo and Unity3D. For Gazebo, a simple world file was created using the same lighting and physics settings as in the HRATC world file. Next a model element with a nested link element was created in the world file. The link element itself contained a

visual element for displaying the UA SEQ Collada mesh file and a collision element for preventing the Jackal UGV from passing through the ground or the buildings. The visual element was configured to cast shadows and the model element was configured as static to prevent the terrain, buildings, and other objects making up the environment from being affected by gravity. At the start of the simulation, the Jackal UGV is placed at the origin directly in front of one of the entrances to the South Engineering Research Center facing towards the doors. The UA SEQ environment is shown below in Figure 29 with the starting location of the Jackal UGV circled in red. Despite the lamp posts being colored black in the Blender modeling software, in Gazebo they appear gray colored, indicating that Gazebo is not fully compatible with Collada files produced by Blender. Furthermore, the collision mesh of the UA SEQ does not appear to perfectly match the visual mesh. Figure 30 shows three boxes in the UA SEQ world where the boxes on the left and right sit on the surface while the center box floats on an invisible collider. As discussed later in the summary section, the presence of the invisible collider negatively affects simulation accuracy in several ways.

(a) (b)

Figure 29. The University of Alabama Science and Engineering Quad in Gazebo (a) top-down orthogonal view and (b) angled perspective view.

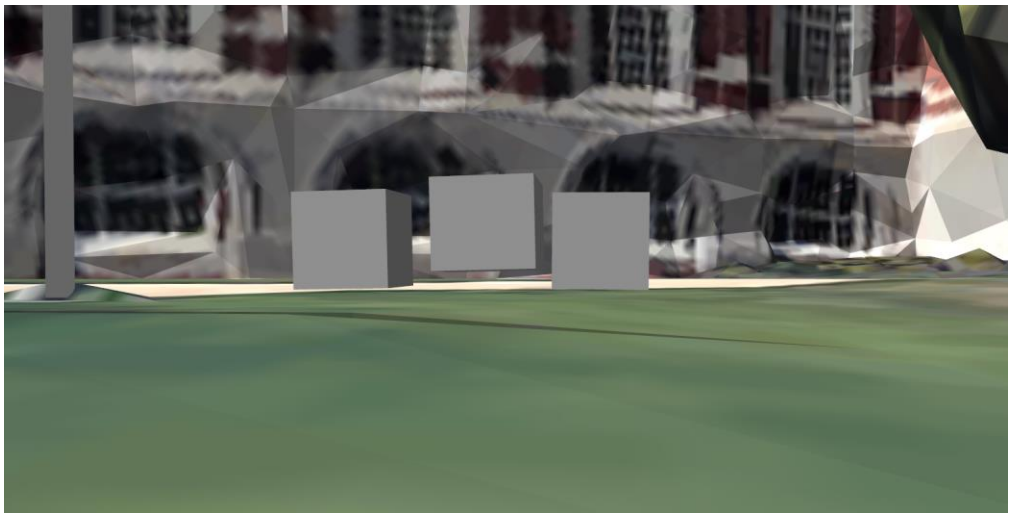


Figure 30. The invisible collider in the Gazebo UA SEQ environment shown by the apparent levitation of the center box.

For Unity3D, the Collada 3D mesh and 220 PNG texture files were imported into a new project by dragging their parent folder into the project window. The largest of the texture files has a resolution of 512 x 512 pixels, so no adjustments to the import resolution were required. The imported UA SEQ mesh was placed by dragging it from the project window into the scene. Since all objects were compressed into a single mesh within Blender, collision was added to the full environment using a single mesh collider component on the UA SEQ GameObject. A Unity3D GameObject with a directional light component was also added to the scene, configured with a light intensity of 100% and oriented directly downwards to match the light used in Gazebo. Lastly, the Jackal UGV was positioned in the same starting location and orientation. The Unity3D UA SEQ world is pictured in Figure 31 with the starting position of the Jackal UGV circled in red. Unity3D had no apparent issues with the Collada file produced in Blender with the lamp posts appearing black as modeled and the collision mesh exactly matching the visual mesh with no invisible colliders.

(a) (b)

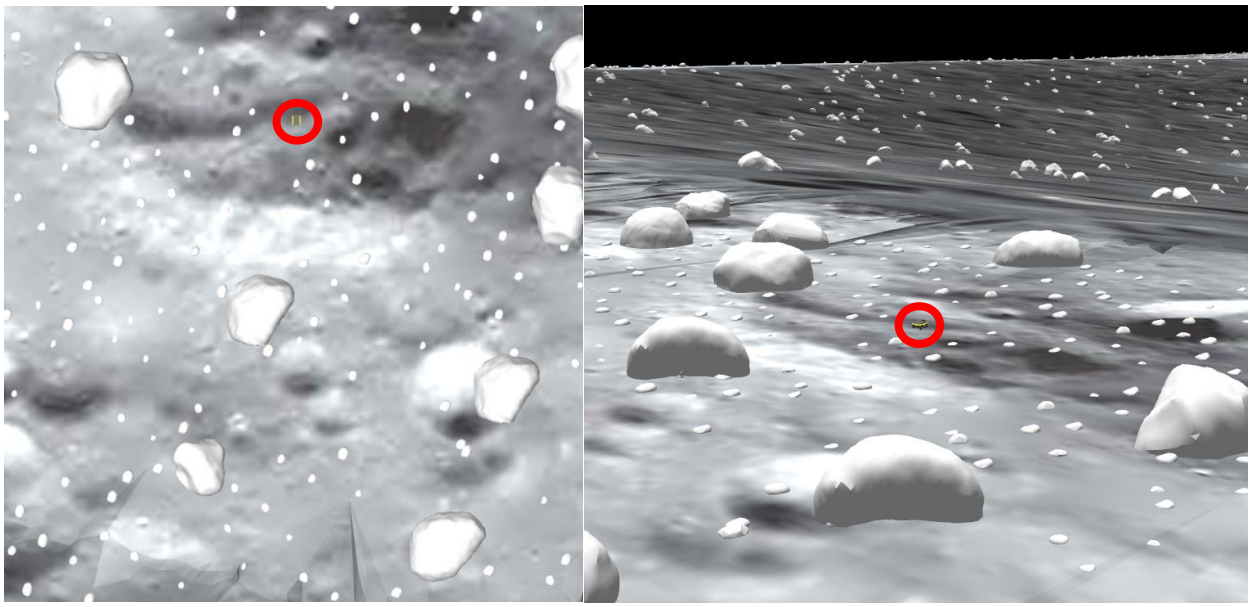
Figure 31. The University of Alabama Science and Engineering Quad in Unity3D (a) top-down orthogonal view and (b) angled perspective view.

4.3.5 Lunar Surface

The final environment is based on a portion of the lunar surface. It consists of a large, gray-colored, cratered landscape interspersed with rocks of varying sizes. Consistent with the view from the surface of the Moon, the sky color is set to black. This world is primarily intended to simulate lunar rovers; however, it could also serve as a template for creating other interplanetary environments. The model of the lunar surface was created using a combination of several software programs and data sources. The digital elevation map used for creating the terrain was downloaded directly from the NASA Moon Trek Web Map Tile Service [111]. Specifically, the 10 m resolution relief of the south pole based on data captured by the Lunar Orbiter Laser Altimeter instrument [112] was used. Subsequently, the digital elevation map was imported into QGIS, an open-source program for visualizing and editing geospatial information [113]. QGIS was used to crop the data to the area around a single large crater. The cropped digital elevation map was then saved and imported into Blender using the BlenderGIS add-on [114]. From there, BlenderGIS was used to create a 1-to-1 scale mesh from the digital elevation map data. To artificially increase the resolution of the mesh and accentuate the changes in

elevation, the mesh file was resized to 1/5 scale in the horizontal plane, but left at the original scale along the height axis. To add visual contrast to the otherwise blank mesh, a publicly available lunar surface texture was applied, and the Blender Rock Generation add-on was used to randomly disperse rock obstacles in the environment. The finalized lunar surface environment was exported as a Collada file. As scaled, the environment is roughly 1600 x 2500 m. Including the terrain mesh and the rocks, the environment is made up of 2,742,273 mesh triangles, and requires 649.5 MB of disk space, including the 4.9 MB PNG texture file.

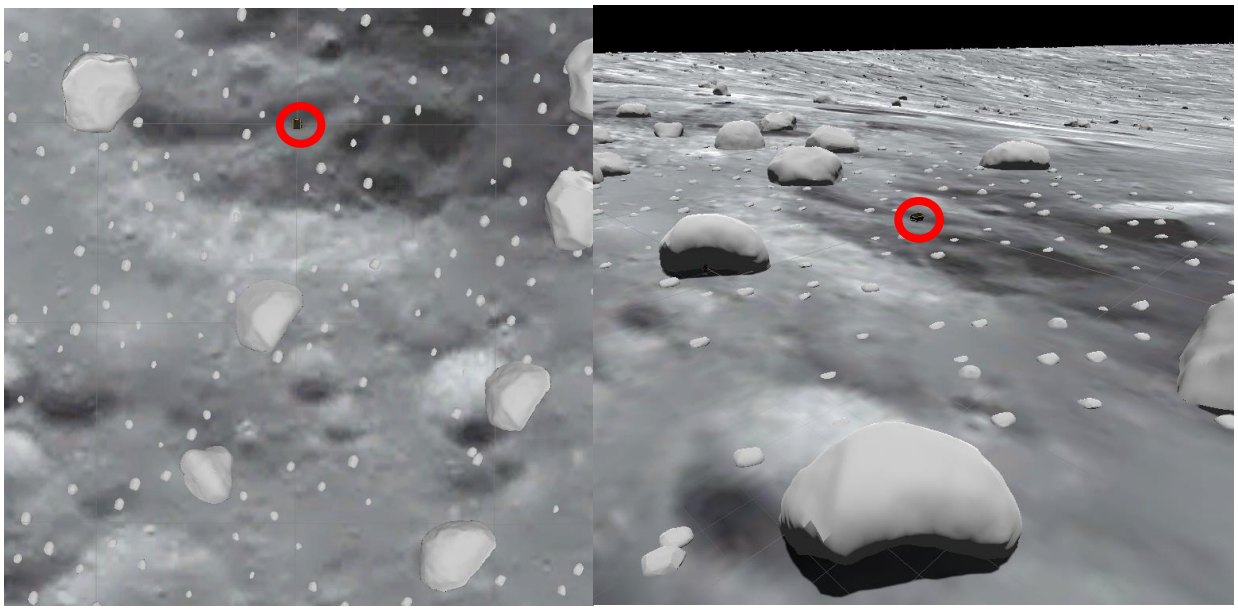
Like the UA SEQ model, the lunar surface model required manual integration with Gazebo and Unity3D. For Gazebo, a simple world file was created with a directional light set at 100% intensity pointing directly downwards. The scene element was configured to produce a solid black sky color by setting the background color vector to all zeros. The physics element was configured to use the same settings as used by the other environments. Although it could be changed, gravitation acceleration was kept at -9.81 m/s^2 since lower values allowed the rover to occasionally leave the surface in Unity3D. Next, a model element with a nested link element was created in the world file. The link element itself contained visual and collision elements which were used to display and add collision to the lunar surface environment. The visual element was configured to cast shadows and the model element was configured as static to prevent it from being affected by gravity. At the start of the simulation, the Jackal UGV was placed at the on the edge of the large crater facing towards the opposite rim. The lunar surface environment is shown below in Figure 32 with the starting position of the Jackal UGV circled in red.



(a) (b)

Figure 32. The lunar surface simulation environment in Gazebo (a) top-down orthogonal view and (b) angled perspective view.

For Unity3D, the Collada 3D mesh file and PNG textures were imported into a new project by dragging the parent folder into the project window. The single texture file has a resolution of 2048 x 2048 pixels, so no adjustments to the import resolution were required. The imported lunar surface mesh was placed by dragging it from the project window into the scene. Since the landscape and rocks were compressed into a single mesh file within Blender, nothing had to be manually positioned within Unity3D. Collision was added to the landscape and rocks using the same AddColliders script used for the first three environments. A GameObject with a directional light component was also added to the scene, configured with a light intensity of 100% and oriented directly downwards to match the light used in Gazebo. The scene skybox was configured with an atmospheric thickness of 0 to model the environment without atmospheric refraction, resulting in crisp shadows and a black sky color. In the project physics settings, the gravity was not changed from the -9.81 m/s^2 for the reasons discussed above. Lastly, the Jackal UGV was positioned to start at the edge of the large crater facing the opposite rim. The Unity3D lunar world is pictured in Figure 33 with the starting position of the Jackal UGV circled in red.



(a) (b)

Figure 33. The lunar surface simulation environment in Unity3D (a) top-down orthogonal view and (b) angled perspective view.

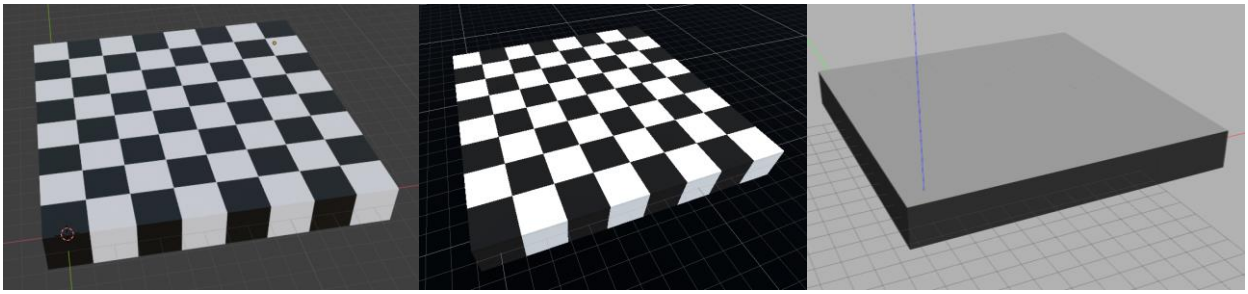
4.3.6 Summary

In summary, five simulation environments were constructed to be identical in Unity3D and Gazebo. Each is visually distinct and represents a different possible use case for autonomous ground robots. The environments have different file sizes, levels of detail, and types of features present. The properties and characteristics of the five environments are summarized in Table 5. The first three environments were provided by ClearPath robotics as Gazebo worlds allowing them to be easily modified and used in Gazebo. Despite being designed exclusively for use with Gazebo, the underlying meshes could still be easily imported into Unity3D and positioned manually to create identical environments.

TABLE 5. SUMMARY OF ENVIRONMENT PROPERTIES

Environment	**Dimensions (m)**	**Area (m^2)**	**Number of Triangles**	**Texture Size (MB)**	**Total File Size (MB)**
HRATC	30.48 x 30.48	929.0	18,771	0.01	1.23
Agriculture	160 x 120	19,200	17,820	39.7	42.4
Inspection	60 x 60	3,600	5,767	46.6	47.4
UA SEQ	575 x 340	195,500	330,374	21.9	76.1
Lunar Surface	1600 x 2500	4,000,000	2,742,273	4.9	649.5

The last two environments were developed in a simulator neutral way in Blender. Both were easily imported into both Unity3D and Gazebo, however Gazebo specifically exhibited some notable inconsistencies from the intended design. One inconsistency was that the lamp posts in the UA SEQ environment were colored gray instead of black. While this was not a large detriment in this specific case, it is easy to imagine a scenario where inaccurate colors could play a larger role. Consider the image in Figure 34 of a black and white tiled floor. In Unity3D the floor appears exactly as intended meanwhile in Gazebo the floor is monotone gray. Clearly a visual-based SLAM algorithm would perform much better in this environment in Unity3D than in Gazebo. The other inconsistency was the presence of an invisible collider in the Gazebo world, not present in Blender or Unity3D. The primary issue is that this creates an invisible obstacle that the Jackal UGV would drive into and become stuck yet continue trying to drive forward. Overcoming this requires extremely robust recovery behaviors or manual intervention. Either way, the presence of the invisible collider in Gazebo inhibits the realism and autonomy of the simulated Jackal UGV. Overall, thanks to Unity3D and Gazebo sharing many of the same compatible file types, little effort was required to get the worlds working in both simulators, however the observed differences do suggest that Unity3D has better support for environments developed in third-party modeling software such as Blender.



(a) (b) (c)

Figure 34. Checkered tile floor example model in (a) Blender, (b) Unity3D, and (c) Gazebo.

CHAPTER 5:
TEST SCENARIOS

To compare the practical differences between ROS-Unity3D and ROS-Gazebo when simulating the autonomous Jackal UGV, several test scenarios were developed using the environments discussed in the previous section. In total, seven test scenarios were developed to assess the performance of the simulators and the autonomous vehicle in different conditions. Each test scenario consists of a unique combination of active sensors, SLAM strategy, environment, and set of navigation waypoints. The first section describes three test scenarios that are conducted in the HRATC world to assess the differences between ROS-Unity3D and ROS-Gazebo while simulating different suites of sensors. The next section describes four test scenarios used to assess the differences between the simulators when performing stereo camera-based SLAM in different environments. The final section discusses the methods by which performance metrics are acquired and processed for each test scenario run.

5.1. Differing Sensor Scenarios

Different combinations of sensors are simulated and used to perform SLAM in the first three test scenarios for ROS-Unity3D and ROS-Gazebo. To ensure that any differences that are observed between the simulation scenarios are attributable primarily to the simulated sensor implementations, all three scenarios consider the same environment and set of navigation waypoints. The scenarios consist of autonomous navigation between six waypoints in the HRATC environment. The waypoints alternate between the initial pose of the Jackal UGV and other places in the environment. The path has been structured such that there is always a tree in

view of the stereo camera. An overhead view of the waypoints and a simple path connecting them overlaid on the environment map is shown in Figure 35. The aggregate length of the direct paths between the waypoints is about 34.5 m long with mild elevation change to and from the second waypoint. All nine of the trees are at least partially visible from the path and they are expected to be mapped by the autonomy stack. As for sensing the environment, the first scenario uses only the stereo camera, the second uses only the LiDAR sensor, and the third uses both.

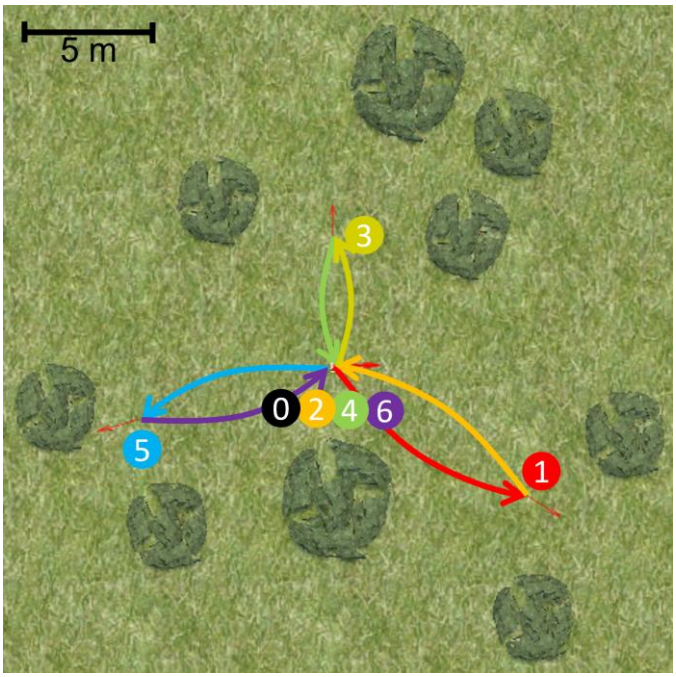


Figure 35. Overhead view of the waypoints and path for the first three test scenarios.

5.1.1 Scenario 1: Stereo Camera Sensing

Scenario 1 simulates the Jackal UGV autonomously navigating through waypoints in the HRATC environment performing SLAM using only the Bumblebee2 stereo camera system. In ROS-Unity3D, this was accomplished by disabling the Velodyne GameObject part of the Jackal UGV. For ROS-Gazebo, the simulator was started using the "front_bumblebee2" configuration option which specifies that the simulated Jackal UGV should only have the Bumblebee2 stereo camera in addition to its default sensor suite. In both simulators, the wheel encoder sensors and

IMU are still simulated, however they are not used by the ROS autonomy stack. Instead, RTAB-Map is configured to perform odometry and mapping using the stereo camera system.

5.1.2 Scenario 2: LiDAR Sensing

Scenario 2 simulates the Jackal UGV in the HRATC environment performing SLAM using only the VLP-16 LiDAR sensor. In ROS-Unity3D, this was done by disabling the "front_mount" GameObject which disables the Bumblebee2 stereo camera along with its mounting hardware. For ROS-Gazebo, the simulator was started using the "vlp16" configuration option which specifies that the simulated Jackal UGV should only have the VLP-16 LiDAR sensor beyond its default sensor suite. For both ROS-Unity3D and ROS-Gazebo simulation RTAB-Map is configured to perform odometry and mapping using exclusively LiDAR.

5.1.3 Scenario 3: Stereo Camera and LiDAR Sensing

Scenario 3 simulates the Jackal UGV autonomously navigating through the HRATC environment performing odometry using the Bumblebee2 stereo camera system and mapping using the VLP-16 LiDAR sensor. In ROS-Unity3D, this was accomplished by enabling both the "front_mount" and Velodyne GameObjects, ensuring that both sensors were fully simulated. For ROS-Gazebo, the simulator was started using the "bumblebee_laser" configuration option which specifies that the simulated Jackal UGV should have both the Bumblebee2 stereo camera and VLP-16 LiDAR sensor, in addition to its default sensor suite. As in the previous two scenarios, both simulators include the wheel encoder sensors and IMU, but they are not used by the ROS autonomy stack. Instead, RTAB-Map is configured to use the "stereo_odometry" node for odometry while the "rtabmap" node itself subscribes to point clouds on the "/mid/point" topic from the LiDAR scanner for mapping.

5.2. Differing Environment Scenarios

The remaining four test scenarios simulate the Jackal UGV autonomously navigating through differing environments using exclusively stereo camera-based odometry and SLAM. As in the first test scenario, stereo camera-based SLAM was accomplished in ROS-Unity3D by disabling the Velodyne GameObject and in ROS-Gazebo by starting the simulator with the "front_bumblebee2" configuration option. Similarly, the wheel encoders and IMU are still simulated, but RTAB-Map is configured to only use visual SLAM from the Bumblebee2 stereo camera system. The simulation scenarios are conducted in the four remaining environments: the agriculture world, the inspection world, the UA SEQ, and the lunar surface, each with a unique set of waypoints. In this way, each scenario represents a different possible use case for an autonomous ground vehicle.

5.2.1 Scenario 4: Agriculture World

Scenario 4 simulates the Jackal UGV navigating between two rows of solar panels in the agriculture world environment. This scenario represents the solar panel inspection use case of autonomous ground robots. The specific path followed by the Jackal UGV consists of 11 waypoints with an aggregate direct path distance between the waypoints of approximately 61.4 m. From the starting location beside the barn, the waypoints command the Jackal UGV to navigate between the first and second rows of solar panels, then back to its initial position. An overhead view of the waypoints and a simple path connecting them overlaid on the environment map is shown in Figure 36. There is a mild downward slope and decrease in elevation moving from the starting location towards the solar panels which is accounted for by the RTAB-Map localization system. The barn and several solar panels from the first and second rows are visible

from the path and are expected to be perceived by the mapping system. Specifically, their supports and edges should be considered as obstacles.

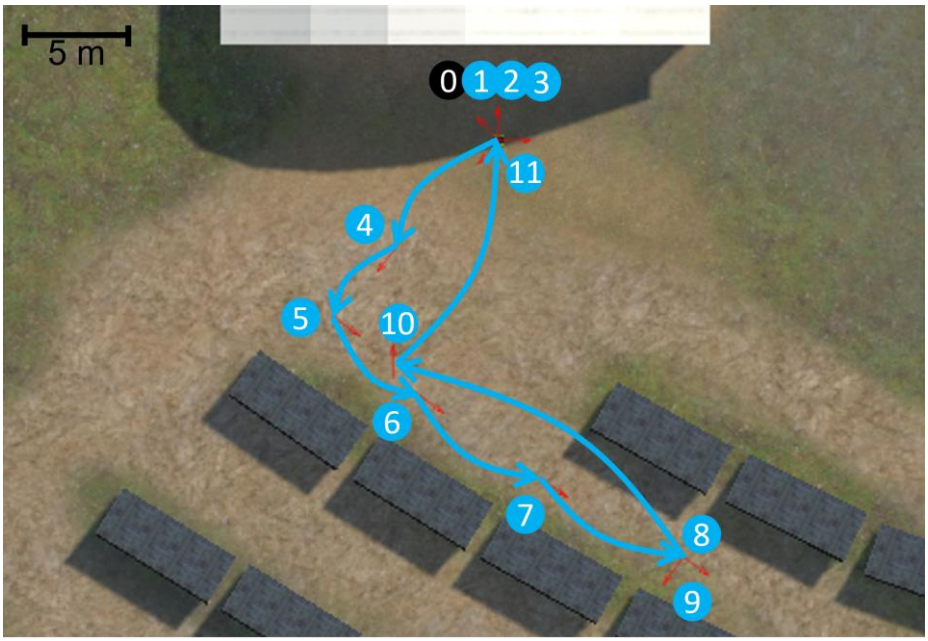


Figure 36. Overhead view of the waypoints and path for the scenario 4.

5.2.2 Scenario 5: Inspection World

Scenario 5 puts the Jackal UGV in the inspection world environment. The navigation path takes the UGV from its starting location beside the pond to the water pipes and back again. This scenario represents the pipeline inspection use case of autonomous ground robots. The specific path followed by the Jackal UGV consists of six waypoints with an aggregate direct path distance between the waypoints of approximately 36 m. Beginning beside the pond, the Jackal UGV up a small hill where it can get a close-up view of the water pipes. Afterwards, the rover turns around and navigates back towards its starting position. The final waypoint is 2 m further from the pond than the starting location to prevent accumulated navigation errors from causing the Jackal UGV to inadvertently fall into the pond. An overhead view of the waypoints and a simple path connecting them overlaid on the environment map is shown in Figure 37. In terms of obstacles, the environment is relatively sparse with the main obstacle being the pipeline.

Figure 37. Overhead view of the waypoints and path for the scenario 5.

5.2.3 Scenario 6: University of Alabama Science and Engineering Quad

Scenario 6 has the Jackal UGV navigating between two entrances to the South Engineering Research Center in the UA SEQ environment. This scenario represents a portion of a route that an on-campus, autonomous delivery vehicle would be expected to follow. The specific path followed by the Jackal UGV consists of 20 waypoints with an aggregate direct path distance between the waypoints of approximately 47.9 m. From the starting location facing the doors appearing on the left side of the building, the waypoints command the Jackal UGV to turn clockwise and navigate along the sidewalk to the other entrance on the right side of the building. Many intermediate waypoints are used to ensure that the simulated Jackal UGV follows the curved path of the sidewalk, since it currently has no way of differentiating between the sidewalk and the grass. An overhead view of the waypoints and the overall desired path connecting them overlaid on the environment map is shown in Figure 38. The path is mostly flat with a 15 cm decrease in elevation along the 47.9 m long path. Along the path, the main obstacles within sight of the Jackal UGV are two sets of bushes, two trees, and three lamp posts. The building likely will not be mapped as an obstacle since it will fall outside the 5 m mapping radius.

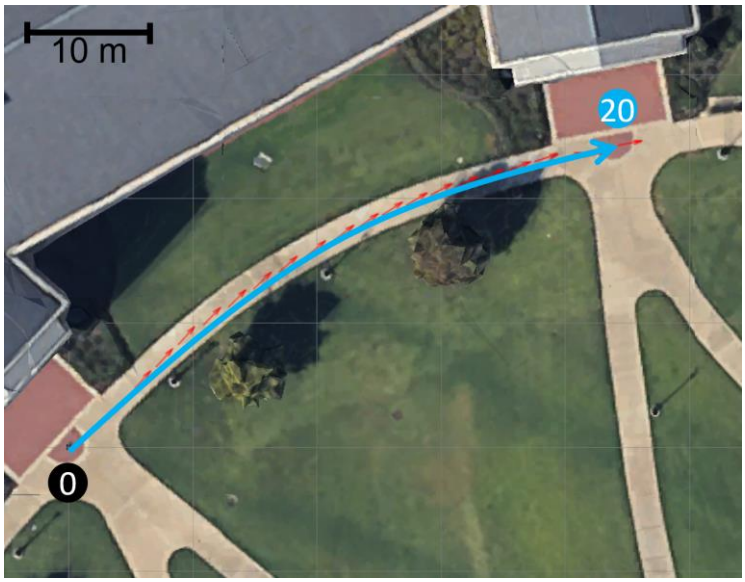


Figure 38. Overhead view of the waypoints and path for the scenario 6.

5.2.4 Scenario 7: Lunar Surface

The final scenario, Scenario 7, simulates the Jackal UGV navigating around a large boulder in the lunar surface environment. This scenario represents the extraterrestrial rover use case of autonomous ground robots. The specific path followed by the Jackal UGV consists of 17 waypoints with an aggregate direct path distance between the waypoints of approximately 56.9 m. From the starting location facing towards the crater, the waypoints command the Jackal UGV to drive in a rough circle around a moon boulder while avoiding smaller rocks. After completing a circuit around the boulder, the rover comes to a stop back at its initial position. About halfway through the path at the tenth waypoint, the Jackal UGV is commanded to turn inwards towards to rock allowing it to be partially mapped. An overhead view of the waypoints and the circular path overlaid on the lunar surface map is shown in Figure 39. There is no elevation change along the circular path around the rock. There are six boulders and numerous smaller rocks that are expected to be detected and mapped as obstacles in the occupancy grid.

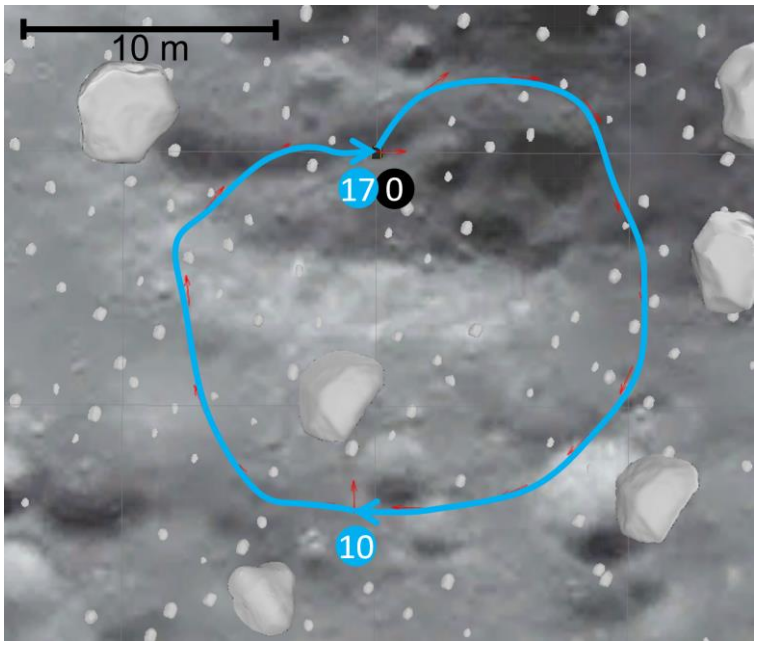


Figure 39. Overhead view of the waypoints and path for the scenario 7.

5.3. Measuring Simulated Performance

Qualitative comparison between the ROS-Unity3D and ROS-Gazebo simulators in the seven test scenarios required that several numerical performance metrics be collected during each test scenario simulation run. Broadly speaking, two types of metrics are collected: those related to the efficiency of the simulation software and those related to the accuracy of the simulated autonomous Jackal UGV. The efficiency of simulation is further subdivided into computer resource utilization metrics and simulation speed metrics. The simulated Jackal UGV accuracy is also subdivided into two categories: localization error and mapping accuracy. ROS-Unity3D and ROS-Gazebo both use the same ROS autonomy stack, so any performance differences in this area are attributed to differences in the way the Jackal UGV senses and moves within the simulated environment. Both localization error and mapping accuracy are relative to the ground truth data. The following sections describe the specific performance metrics that are considered and how they are measured in ROS-Unity3D and ROS-Gazebo.

5.3.1 Computer Resource Utilization

For each test scenario simulation run, the computer resources utilized by the simulators are assessed using several metrics. Being that ROS-Unity3D and ROS-Gazebo use the same autonomy stack, the resources used by ROS are not measured, instead focusing exclusively on the resources used by the main Unity3D and Gazebo processes. Performance metrics are captured using a custom ROS node written in Python. This node takes a process ID (PID) and logs the computer resources used by that process to an external file at 1 Hz. The custom node uses the process and system utilities (PSUtil) python library [115] to measure the following resources used by the specified process: central processing unit (CPU) threads, CPU percent usage relative to a single core, and memory. Since the CPU percent usage is relative to a single core, usages above 100% are possible and imply that multiple CPU cores are used. The custom node also calls the Nvidia System Management Interface (SMI) command line utility [116] and parses the output to obtain the GPU frame buffer memory usage and GPU streaming multiprocessor usage associated with the provided PID. Following a test scenario run, the metrics for each second are averaged across the length of the run.

5.3.2 Simulation Speed

Unity3D and Gazebo simulate the passage of time differently meaning that each uses different metrics to track simulation speed. In Unity3D simulation speed is measured using the number of variable time step updates per second, the number of fixed time step updates per second, and the reported real-time factor. Unity3D was configured with a fixed time step length of 0.01 s and a time scale of 1, so it is expected that there will always be 100 fixed time step updates per second and a real-time factor of 1. Therefore, simulation speed will be primarily assessed based on whether the variable time step update rate is high enough to accurately

simulate the sensors on the Jackal UGV: 20 FPS for the Bumblebee2 stereo camera and 10 FPS for the VLP-16 LiDAR sensor. The three Unity3D simulation speed metrics are captured directly from Unity3D using a custom component. The metrics are then sent as a ROS message to the custom ROS node which logs them in an external file at 1 Hz. At the conclusion of a run, the entries across every second are averaged yielding a single value for each metric.

The only simulation speed metric reported by Gazebo is the real-time factor. Since any real-time factor below 1 represents slower than real-time simulation, the simulation speed of Gazebo will be assessed based on the extent to which it can maintain a real-time factor of 1 over the course of a test scenario run. The Gazebo real-time factor is published on the "/gazebo/performance_metrics" topic. A custom ROS node subscribes to the performance metrics topic and stores the average simulation real-time factor over the course of a simulation run in an external file.

5.3.3 Localization Error

The localization error of the Jackal UGV is defined as the difference between its actual position in the simulator, referred to as the ground truth, and its position as estimated by the ROS autonomy stack. Fortunately, RTAB-Map can measure localization error on its own, when provided with ground truth position and rotation measurements. In Unity3D, ground truth odometry messages are published to the "/ground_truth/state" topic by the custom ground truth odometry publisher component. In Gazebo, this is done using the P3D base controller plugin. Within its internal database, RTAB-Map stores the positional and rotational error at each localization step and computes various positional and rotation error statics including maximum error, mean error, and median error. After each test scenario simulation run, these positional and

rotational error statistics are copied from the RTAM-Map database viewer and are used to compare the simulated localization accuracy in ROS-Unity3D and ROS-Gazebo.

5.3.4 Mapping Accuracy

Unlike with localization error, RTAB-Map has no way of assessing mapping accuracy on its own. Instead, at the end of every test scenario simulation run, the full RTAB-Map 2D occupancy grid is saved as an image file using the ROS "map_saver" tool [117]. The pixels in the occupancy grid image can have one of three grayscale values: 0 representing occupied, 205 representing unknown, and 254 representing clear. The occupancy grid is lined up and compared with a ground truth occupancy grid image using a custom MATLAB script. The ground truth occupancy grid is created manually for each environment by converting a top-down view of the environment into a grayscale image, scaling it so that each pixel represents a 5 cm by 5 cm section of the simulated world as in the RTAB-Map occupancy grid, then coloring cells only containing ground with a value of 254 and cells containing an obstacle with a value of 0. The comparison script iterates through every pixel in the ground truth image and finds its equivalent in the RTAB-Map occupancy grid if it exists. Pixels in the ground truth are tallied as either truly occupied but classified as unknown, truly occupied but classified as clear, truly occupied and classified as occupied, truly clear but classified as unknown, truly clear and classified as clear, or truly clear but classified as occupied. The six possible states are summarized in the confusion matrix shown in Table 6. Thus, the occupancy grid maps produced by ROS-Unity3D and ROS-Gazebo can be compared visually and numerically, based on the number of correct, incorrect, and unknown classifications.

TABLE 6. MAPPING OCCUPANCY GRID CONFUSION MATRIX

		RTAB-Map Classification		
		Occupied	**Clear**	**Unknown**
Ground Truth Classification	**Occupied**	Truly occupied and classified as occupied *(True Positive)*	Truly occupied but classified as clear *(False Negative)*	Truly occupied but classified as unknown *(Unknown Occupied)*
	Clear	Truly clear but classified as occupied *(False Positive)*	Truly clear and classified as clear *(True Negative)*	Truly clear but classified as unknown *(Unknown Clear)*

CHAPTER 6:
RESULTS AND DISCUSSION

This chapter lists and discusses the results of the seven test scenarios as simulated in ROS-Unity3D and ROS-Gazebo. All simulations were conducted on a desktop computer equipped with an Intel Core i9-9960X CPU, 64 GB of RAM, and two Nvidia Titan RTX GPUs with a combined total of 48 GB of VRAM running Ubuntu 20.04 LTS. As discussed earlier, the simulators are based on Unity3D 2022.1, Gazebo 11, and ROS Noetic. Each test scenario was simulated five times in each simulator, with each individual simulation being referred to as a "run". Between every run, the RTAB-Map internal database was cleared ensuring that no mapping data was preserved between subsequent simulations. Unless otherwise noted, all results discussed in this chapter represent the average across all five runs for the given combination of simulator and test scenario. The results from the individual test runs are included in the Appendix. For each test scenario, ROS-Unity3D and ROS-Gazebo are compared based on their computer resource usage, simulation speed, and localization and mapping accuracy of the Jackal UGV relative to ground truth, collected as discussed in Section 5.3: Measuring Simulated Performance. The final section of this chapter contains general performance notes that are not associated with any individual test scenario.

6.1. Scenario 1: Stereo Camera Sensing

The average computer resource utilization metrics for the five runs of Scenario 1 in ROS-Unity3D and ROS-Gazebo are shown in Table 7. Across all metrics, ROS-Unity3D used more computer resources than ROS-Gazebo. This is surprising, since ROS-Gazebo is configured to

simulate using a significantly shorter time step than ROS-Unity3D. This indicates that ROS-Unity3D may not be as well optimized as ROS-Gazebo for this simple simulation scenario. By far the largest discrepancy in terms of resource usage is that ROS-Unity3D uses over 16 times as much GPU framebuffer memory as ROS-Gazebo suggesting that ROS-Unity3D takes greater advantage of GPU resources.

TABLE 7. SCENARIO 1 AVERAGE COMPUTER RESOURCE UTILIZATION

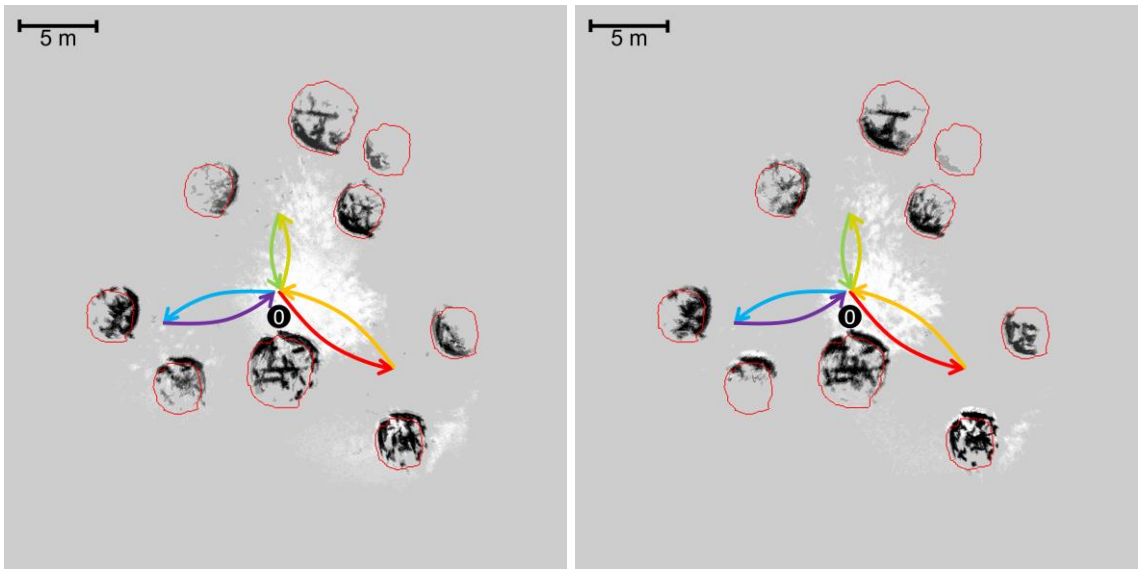
	Unit	**ROS-Unity3D**	**ROS-Gazebo**
# CPU Threads	1	171	122
CPU %	%	136.89	111.41
Memory	MB	714	233
GPU Framebuffer Memory	MB	2642.24	160
GPU Streaming Multiprocessor	%	14.50	5.45
Real-Time Factor	1	1.00 (35.3 FPS)	1.00

In terms of simulation speed, ROS-Unity3D ran at a variable time step update rate of 35.3 FPS on average. This exceeds the requirement of 20 FPS to maintain accurate simulation of the Bumblebee2 camera sensor. As expected, the reported time scale and fixed time step update rate were exactly 1 and 100, respectively, across all simulation runs. This indicates that the ROS-Unity3D physics simulation maintained real-time performance and was unaffected by the performance demands of simulating the sensors on the Jackal UGV. The ROS-Gazebo simulator reported an average time scale of 1.00, indicating that it also maintained real-time simulation.

The scenario 1 localization performance of the Jackal UGV simulated in ROS-Unity3D and ROS-Gazebo is shown in Table 8. Averaged across the five simulation runs, ROS-Gazebo had a higher degree of rotational inaccuracy whereas ROS-Unity3D had a higher degree of positional inaccuracy. For the most part, the differences in error between the simulators are small, implying that ROS-Unity3D and ROS-Gazebo simulate the stereo camera-based localization in scenario 1 equally well.

TABLE 8. SCENARIO 1 AVERAGE LOCALIZATION ERRORS						
	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	1.481	0.446	0.302	0.244	0.081	0.060
ROS-Gazebo	2.281	0.580	0.452	0.153	0.035	0.031

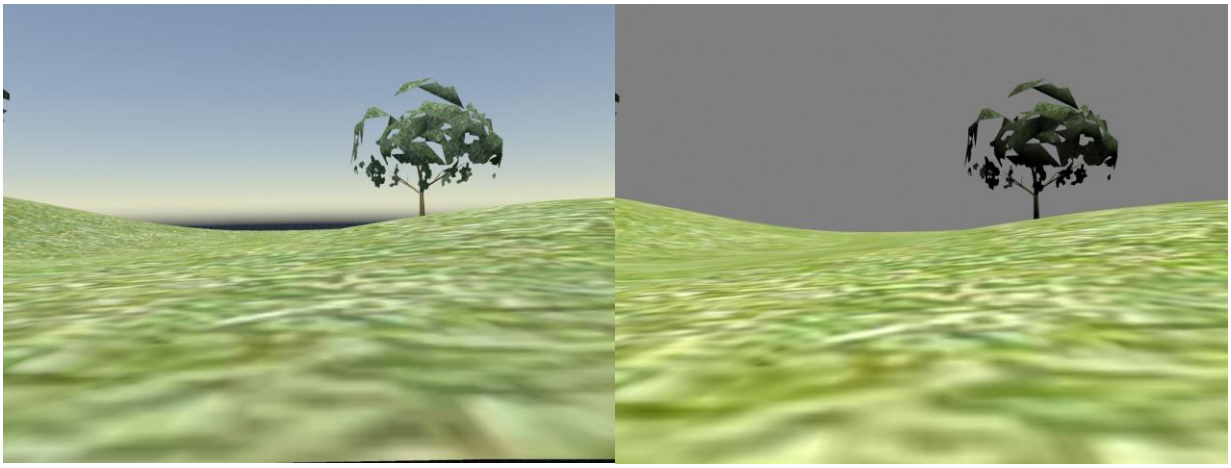
The occupancy grids produced from scenario 1 simulated in ROS-Unity3D and ROS-Gazebo are shown in Figure 40. This figure shows all the occupancy grids from all five runs in each simulator aligned and stacked along with the expected path from the set waypoints. Completely black, white, and gray pixels correspond to cells that are consistently classified as occupied, clear, and unknown, respectively. Cells that are not consistently classified will appear as a darker or lighter gray based on whether they were more frequently classified as occupied or clear. Red lines are placed for reference and represent the ground truth boundary between clear space and obstacles. The average statistics of the occupancy grids across the five runs are tabulated in Table 9. The table shows that ROS-Gazebo slightly outperformed ROS-Unity3D when it came to detecting obstacles, leading to just under 1% more occupancy grid cells being correctly classified as occupied. This is attributed to the slightly higher positional accuracy of the ROS-Gazebo localization system. On the other hand, ROS-Unity3D correctly classified over 48% more clear cells than ROS-Gazebo. This is attributed to slightly greater contrast in the ground texture rendered in ROS-Unity3D as compared to ROS-Gazebo, even though both have been configured with the same textures and lighting settings. Images from the left camera in the HRATC world as seen in ROS are shown in Figure 41. The increased contrast makes it easier for the stereo image processing system to identify points on the otherwise featureless ground. The effect is not as significant for obstacles because of their greater number of geometrical features.



(a) (b)

Figure 40. Aggregated occupancy grids from scenario 1 with ground truth reference simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

TABLE 9. SCENARIO 1 AVERAGE MAPPING ACCURACY STATISTICS						
Ground Truth	**Occupied**			**Clear**		
Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	6,575.4	982.4	18,002.2	14,318.6	1,804.0	330,417.4
ROS-Gazebo	6,639.2	855.0	18,065.8	9,656.8	1,759.4	335,123.8



(a) (b)

Figure 41. Rendered camera images of the HRATC world as received by ROS from (a) ROS-Unity3D and (b) ROS-Gazebo.

6.2. Scenario 2: LiDAR Sensing

The average computer resource utilization metrics for the five runs of scenario 2 in ROS-Unity3D and ROS-Gazebo are shown in Table 10. It is noteworthy that while ROS-Unity3D previously used more computer resources than ROS-Gazebo across all metrics, in scenario 2 it uses memory more efficiently, requiring less than half of what ROS-Gazebo uses. In both simulators, GPU usage is low which is expected since they no longer need to render images for a camera sensor. The GPU streaming multiprocessor usage indicates that ROS-Unity3D performs calculations on the GPU outside of camera sensor simulation while ROS-Gazebo does not.

TABLE 10. SCENARIO 2 AVERAGE COMPUTER RESOURCE UTILIZATION

	Unit	**ROS-Unity3D**	**ROS-Gazebo**
# CPU Threads	1	173	121
CPU %	%	196.88	84.44
Memory	MB	733.78	1,510.3
GPU Framebuffer Memory	MB	157.15	5
GPU Streaming Multiprocessor	%	14.322	0
Real-Time Factor	1	1.00 (220.1 FPS)	0.738

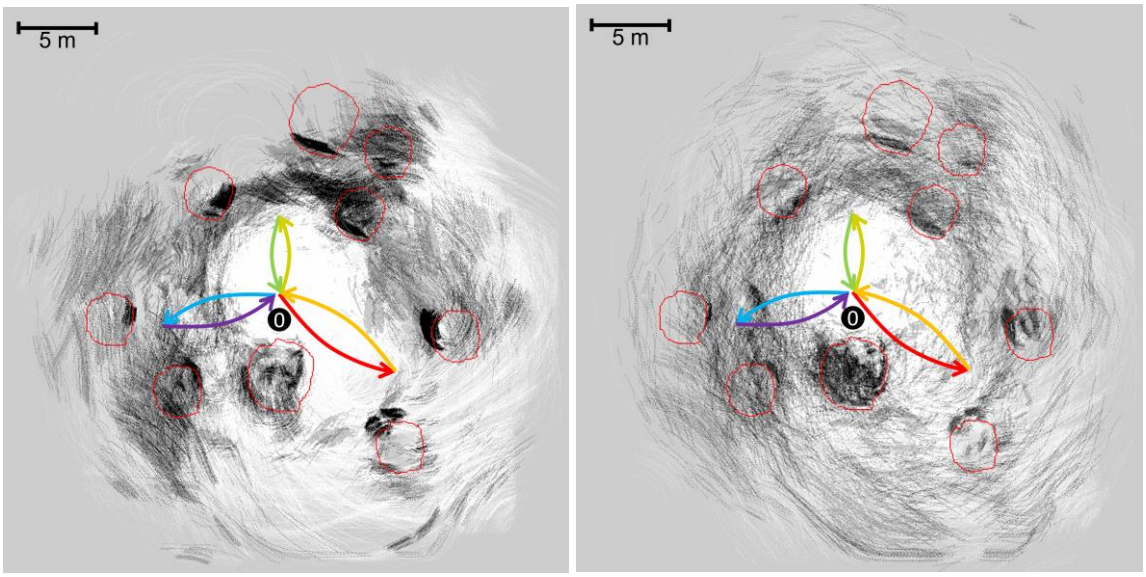
In terms of simulation speed, ROS-Unity3D ran at a variable time step update rate of 220.1 FPS on average. This far exceeds the 10 FPS requirement to simulate the VLP-16 LiDAR scanner as configured. As expected, the reported time scale and fixed time step update rate were exactly 1 and 100, respectively, in all simulation runs. This indicates that the ROS-Unity3D physics simulation maintained real-time performance and was unaffected by simulation of the LiDAR sensor. The ROS-Gazebo simulator reported an average time scale of 0.738, indicating that it was not able to maintain real-time simulation. It can be concluded that simulation of the LiDAR sensor is a large computational burden for ROS-Gazebo. Even though more CPU resources were available for ROS-Gazebo, its simpler LiDAR sensor implementation was unable to use them. This is opposed to the multithreaded ROS-Unity3D LiDAR sensor implementation which allows measurements to be made without slowing down other parts of the simulation.

The Jackal UGV LiDAR-based localization performance was poor for both simulators, as shown in Table 11. Averaged across the five simulation runs, ROS-Unity3D had lower rotational and positional errors than ROS-Gazebo by around a factor of 3. This is attributed to the jerkier movement of the Jackal UGV in ROS-Gazebo than in ROS-Unity3D. As discussed in the physics section, when given a velocity command, the ODE motors used by ROS-Gazebo instantly update the wheel velocity causing an instantaneous change and jerky movement as the Jackal UGV navigates towards the goal. The PhysX articulations used by ROS-Unity3D apply a force proportional to the desired change in velocity instead causing a more gradual acceleration and smoother movement. This finding agrees with earlier differences between Unity3D and Gazebo wheel actuations observed in [40]. The jerky movement of the Jackal UGV was observed in all ROS-Gazebo simulations, but it has an especially significant impact when the slower updating LiDAR sensors are used for odometry.

TABLE 11. SCENARIO 2 AVERAGE LOCALIZATION ERRORS

	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	36.449	14.381	11.274	1.502	0.490	0.267
ROS-Gazebo	149.716	35.377	18.793	4.491	1.327	0.979

In general, accurate mapping is dependent on accurate localization. Consequently, the high localization errors achieved in scenario 2 result in low quality occupancy grids in both ROS-Unity3D and ROS-Gazebo. The large range and 360° field of view of the LiDAR sensor allowed a significant portion of the environment to be mapped, but the high rotational and positional errors led to the position of obstacles becoming smeared across the map, as shown in Figure 42. As before, occupancy grid statistics are also tabulated in Table 12. Due to its previously discussed superior localization performance, ROS-Unity3D correctly classified a greater number of clear and occupied cells, while also having fewer misclassifications.



(a) (b)

Figure 42. Aggregated occupancy grids from scenario 2 with ground truth reference simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

TABLE 12. SCENARIO 2 AVERAGE MAPPING ACCURACY STATISTICS						
Ground Truth	**Occupied**			**Clear**		
Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	9258.4	6,501.2	9,800.4	97,116.4	27,264.6	222,159
ROS-Gazebo	7795.2	7,887.4	9,877.4	85,237.6	27,089.4	234,213

6.3. Scenario 3: Stereo Camera and LiDAR Sensing

Computer resource utilization metrics for the scenario 3 are shown in Table 13. ROS-Unity3D continues to use more CPU and GPU resources and significantly fewer memory resources than ROS-Gazebo. Simulation of the LiDAR sensor dominates the memory usage of ROS-Gazebo with a modest increase compared to scenario 2 from the addition of the stereo camera sensor.

TABLE 13. SCENARIO 3 AVERAGE COMPUTER RESOURCE UTILIZATION			
	Unit	**ROS-Unity3D**	**ROS-Gazebo**
# CPU Threads	1	172	123
CPU %	%	164.80	122.13
Memory	MB	785.23	1597.67
GPU Framebuffer Memory	MB	2,587.16	160
GPU Streaming Multiprocessor	%	9.97	3.42
Real-Time Factor	1	1.00 (24.8 FPS)	0.62

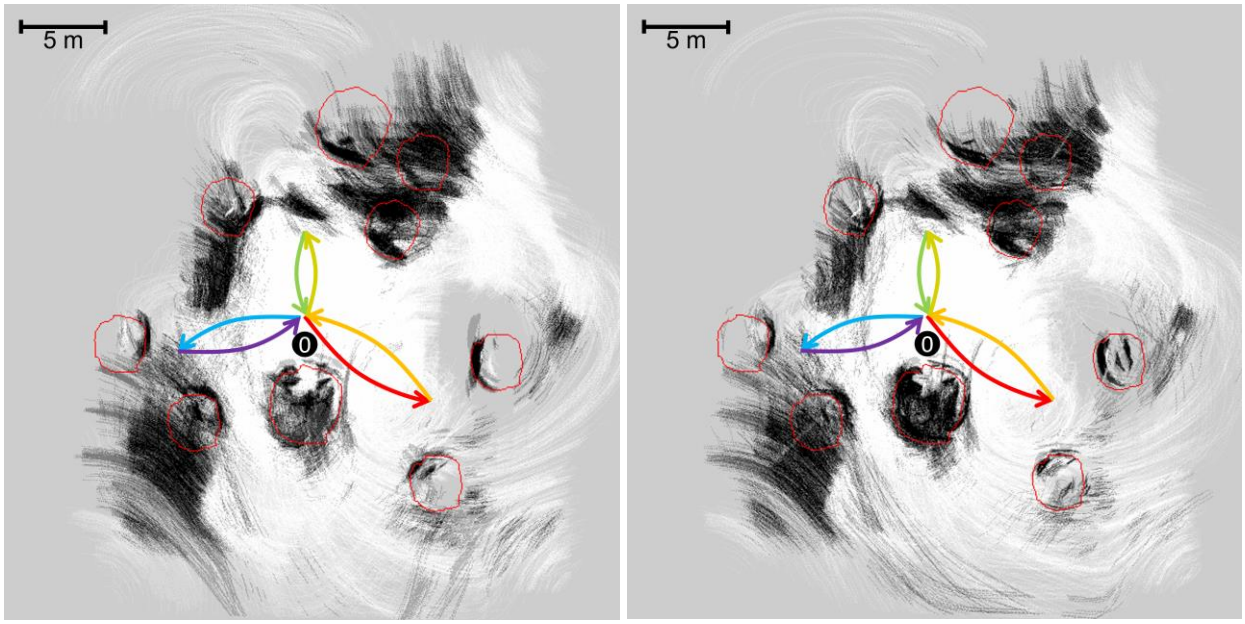
Simulating both the stereo camera system and the LiDAR sensor caused slowdowns in both simulators. Averaged across five scenario 3 simulation runs, ROS-Unity3D ran at a variable time step update rate of 24.8 FPS, just exceeding the 20 FPS required to simulate the Bumblebee2 stereo camera. As before, real-time performance was maintained throughout with 100 fixed time step updates per second and a time scale of 1. The ROS-Gazebo simulator reported an average time scale of 0.62, indicating slower than real-time simulation. It can be concluded that ROS-Gazebo is unable to perform real-time simulation of the VLP-16 LiDAR sensor while ROS-Unity3D can simultaneously simulate the VLP-16 and Bumblebee2 in real-time.

The scenario 3 localization performance is shown in Table 14. Both rotational and positional error are lower in ROS-Gazebo than ROS-Unity3D, however it is important to keep in mind that ROS-Unity3D is simulating in real-time while ROS-Gazebo is not. To better compare accuracy, an additional set of five runs was conducted with the ROS-Unity3D real-time rate intentionally reduced to 0.62 to match that of ROS-Gazebo. This resulted in an average mean rotational error of 0.64° and a mean positional error of 0.08 m, matching the performance from ROS-Gazebo. The increased accuracy at lower real-time factors occurs because slower simulation gives the RTAB-Map stereo odometry system more time to extract features from images before it must make a localization estimate. The full localization results from the additional five runs are shown in the Appendix.

TABLE 14. SCENARIO 3 AVERAGE LOCALIZATION ERRORS

	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	4.687	1.313	1.180	0.375	0.118	0.097
ROS-Gazebo	1.986	0.837	0.927	0.272	0.053	0.043

The scenario 3 occupancy grids are shown in Figure 43. Like in scenario 2, many of the ground truth obstacles appear to have been smeared across the occupancy grid. Since the localization error is much lower than in scenario 2, the most likely explanation for the smearing is that the lower resolution of the LiDAR sensor makes obstacles that are near to one another appear as though they are a single large obstacle. Both simulators showcase this through the three trees in the upper right corner of the environment which are perceived as one large obstacle, even partially connecting to a lone fourth tree to the left. Looking closer at the occupancy grid statistical data in Table 15, ROS-Unity3D left fewer cells classified as unknown, resulting in more correct classifications of clear cells despite running in real-time at nearly twice the speed of ROS-Gazebo. The Jackal UGV as simulated in ROS-Gazebo correctly classified slightly more cells as occupied, which can be attributed to the more accurate localization associated with its slower than real-time simulation rate.



(a) (b)

Figure 43. Aggregated occupancy grids from scenario 3 with ground truth reference simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

TABLE 15. SCENARIO 3 AVERAGE MAPPING ACCURACY STATISTICS						
Ground Truth	**Occupied**			**Clear**		
Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	11,342.2	4,714.2	9,503.6	104,006.6	27,068	215,465.4
ROS-Gazebo	11,360	3,340.4	10,859.6	93,880.6	24,447.4	228,212

6.4. Scenario 4: Agriculture World

The average computer resource utilization metrics for the five runs of scenario 4 in ROS-Unity3D and ROS-Gazebo are shown in Table 16. ROS-Unity3D uses slightly more CPU and memory resources than ROS-Gazebo. Despite increased GPU usage in ROS-Gazebo compared to previous scenarios, ROS-Unity3D still has three times higher framebuffer and streaming multiprocessor usage, indicating better overall usage of GPU resources.

TABLE 16. SCENARIO 4 AVERAGE COMPUTER RESOURCE UTILIZATION			
	Unit	**ROS-Unity3D**	**ROS-Gazebo**
# CPU Threads	1	172.56	122
CPU %	%	133.70	110.43
Memory	MB	1,012.17	802.84
GPU Framebuffer Memory	MB	2,779.94	772
GPU Streaming Multiprocessor	%	13.60	4.40
Real-Time Factor	1	1.00 (35.1 FPS)	0.99

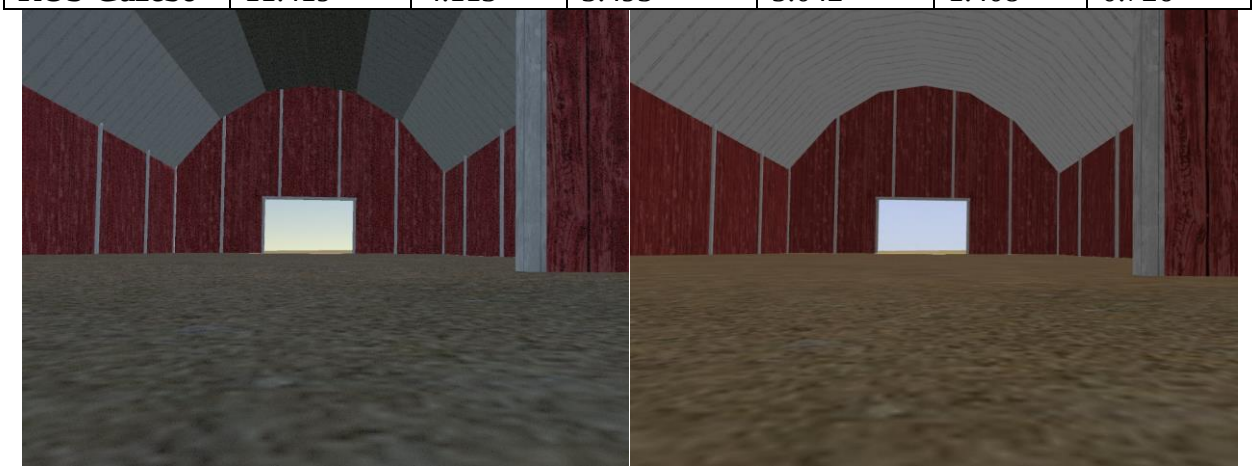
As for simulation speed, ROS-Unity3D had a variable time step update rate of 35.1 FPS, a fixed time step update rate of 100, and a time scale of 1. This exceeds the requirement of 20 FPS to maintain accurate simulation of the Bumblebee2 camera sensor and indicates that real-time simulation is achieved. The ROS-Gazebo simulator reported an average time scale of 0.99, which is slightly below real-time, but is very close to a real-time factor of 1.

The scenario 4 localization performance of the Jackal UGV simulated in ROS-Unity3D and ROS-Gazebo is shown in Table 17. Across all metrics, ROS-Gazebo experiences much higher localization error than ROS-Unity3D. This large error is the result of localization jumps where the odometry system interprets the Jackal UGV as suddenly moving from its current position to an erroneous one anywhere from a few tenths to several meters away. Typically,

these localization jumps are infrequent and can correct themselves by jumping back to the correct position, however in ROS-Gazebo the jumps frequently occur multiple times in a single run without correction. Currently, it is believed these localization jumps occur more often in ROS-Gazebo because it does not maintain as much detail in shadows as ROS-Unity3D. Consider the images in Figure 44: in ROS-Unity3D, the ground inside the barn has the same grainy texture as outside the barn whereas in ROS-Gazebo the ground inside the barn is smoothed out, appearing much more homogenous. The difference in shadow rendering is also pronounced in Figure 41 where the leaves in the shadow of the upper branches are visible in ROS-Unity3D but nearly completely black in ROS-Gazebo. Shadows have a particularly large impact in the agriculture world environment because the Jackal UGV starts in the shadow of the barn and navigates through the shadows cast by solar panels. Regardless of the specific reason, the results of the scenario 4 test environment reveal a significant difference in localization performance between the Jackal UGV simulated in ROS-Unity3D and ROS-Gazebo.

TABLE 17. SCENARIO 4 AVERAGE LOCALIZATION ERRORS

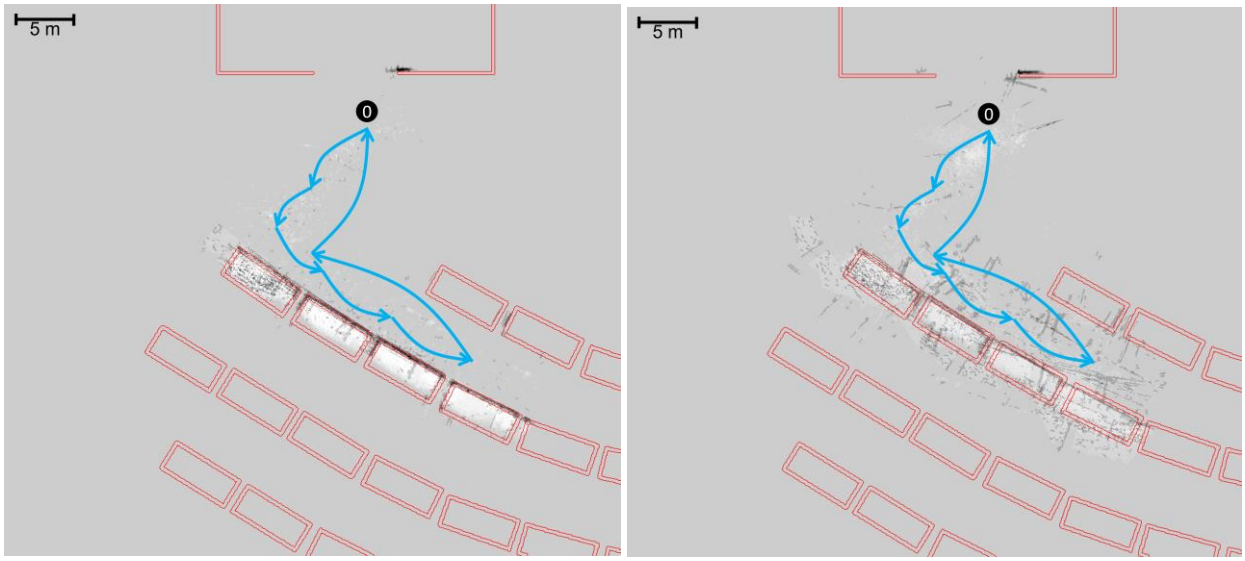
	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	2.204	0.618	0.381	0.556	0.240	0.153
ROS-Gazebo	11.419	4.113	3.493	5.042	1.408	0.726



(a) (b)

Figure 44. Rendered camera images of the agriculture world in shadow from (a) ROS-Unity3D and (b) ROS-Gazebo.

The occupancy grids produced from scenario 4 simulated in ROS-Unity3D and ROS-Gazebo are shown in Figure 45. The high localization error in ROS-Gazebo results in occupancy grids that neither line up with each other nor the ground truth. In contrast, the occupancy grids from the ROS-Unity3D simulator agree with one another and have clearly mapped features corresponding to the ground truth locations of the solar panels. The statistics displayed in Table 18 show that compared to ROS-Gazebo, ROS-Unity3D correctly classifies 87% more occupied cells with only a 37% increase in misclassified occupied cells. In terms of clear cells, ROS-Unity3D has 9% fewer correct classifications and 44% fewer misclassifications. The images corroborate these statistics, clearly showing that ROS-Unity3D produced more accurate occupancy grids in test scenario 4.



(a) (b)

Figure 45. Aggregated occupancy grids from scenario 4 with ground truth reference simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

TABLE 18. SCENARIO 4 AVERAGE MAPPING ACCURACY STATISTICS						
Ground Truth	**Occupied**			**Clear**		
Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	1,424	2,034.4	27,120.6	15,077	2,519.6	846,824.4
ROS-Gazebo	762	1,483.6	28,333.4	16,551.2	4,485.4	843,384.4

6.5. Scenario 5: Inspection World

The average computer resource utilization metrics for simulation scenario 5 are shown in Table 19. The results are very similar to those observed in the scenario 4, which is reasonable since the inspection world and agriculture world are similar in terms of their size on disk, with the inspection world being slightly larger but containing fewer mesh triangles.

TABLE 19. SCENARIO 5 AVERAGE COMPUTER RESOURCE UTILIZATION

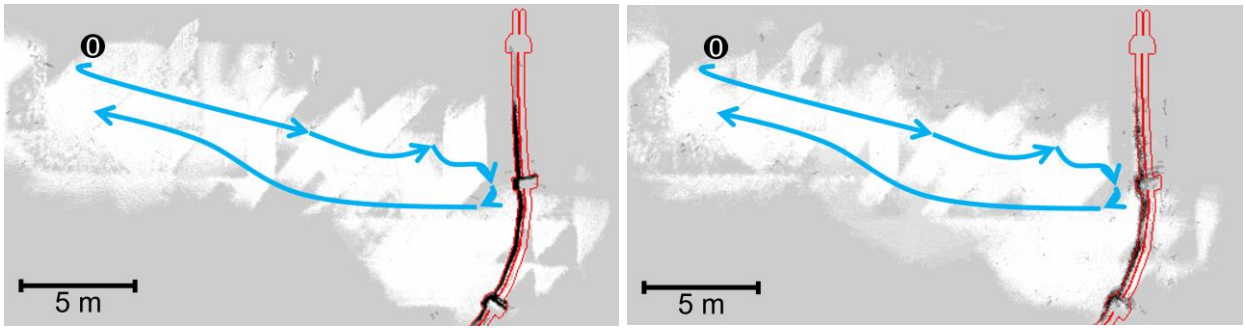
	Unit	**ROS-Unity3D**	**ROS-Gazebo**
# CPU Threads	1	171.1	122
CPU %	%	134.15	108.68
Memory	MB	983.47	750.75
GPU Framebuffer Memory	MB	2,774,22	682
GPU Streaming Multiprocessor	%	15.72	3.73
Real-Time Factor	1	1.00 (33.6 FPS)	0.99

In terms of simulation speed, ROS-Unity3D ran at a variable time step update rate of 33.6 FPS at a time scale of 1 with a fixed time step update rate of 100. This meets the requirement for real-time simulation but is slightly slower than the average framerate achieved in scenario 4. This implies that the 17% increase in texture size from the agriculture world to the inspection world had a larger impact on the frame rate than the 67.6% decrease in triangle count. ROS-Gazebo sees little change in its simulation speed between scenarios 4 and 5 maintaining a slightly under real-time average real-time factor of 0.99.

The scenario 5 localization performance metrics of the Jackal UGV simulated in ROS-Unity3D and ROS-Gazebo are shown in Table 20. On average, ROS-Unity3D achieves slightly lower rotational error and ROS-Gazebo achieves slightly lower positional error. This indicates that ROS-Unity3D and ROS-Gazebo simulate localization equally well in environments with sufficient visual detail, like that of scenario 5 and the earlier scenario 1.

TABLE 20. SCENARIO 4 AVERAGE LOCALIZATION ERRORS						
	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	0.909	0.248	0.267	0.172	0.070	0.048
ROS-Gazebo	1.111	0.283	0.496	0.136	0.063	0.038

The occupancy grids produced during the simulation of scenario 5 in ROS-Unity3D and ROS-Gazebo are shown in Figure 46. Overall, the occupancy grids appear very similar, with much crisper classification of the pipe obstacle in ROS-Unity3D. The quantitative occupancy grid data in Table 21 backs up this claim, showing that ROS-Unity3D has more correct classifications, fewer incorrect classifications, and fewer unknown cell classifications.



(a) (b)

Figure 46. Aggregated occupancy grids from scenario 5 with ground truth reference simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

TABLE 21. SCENARIO 5 AVERAGE MAPPING ACCURACY STATISTICS						
Ground Truth	**Occupied**			**Clear**		
Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	824.4	424.6	1,494	48,046.8	177.2	93,133
ROS-Gazebo	479.8	543.6	1,719.6	44,778.6	492.8	96,085.6

6.6. Scenario 6: University of Alabama Science and Engineering Quad

The average computer resource utilization metrics for the five runs of Scenario 6 are shown in Table 22. Resource utilization in ROS-Unity3D remains higher than in ROS-Gazebo and is consistent with scenarios 4 and 5. ROS-Unity3D ran at a variable time step update rate of 28.25 FPS at a time scale of 1 with a fixed time step update rate of 100. Thus, ROS-Unity3D

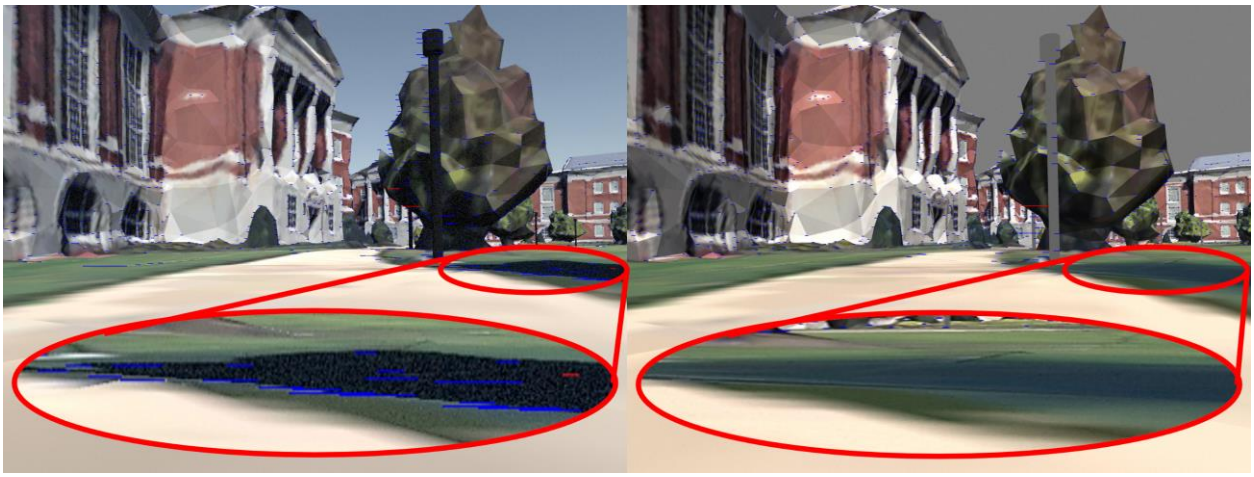
meets the requirement for real-time simulation. ROS-Gazebo again sees little change in its simulation speed maintaining an average real-time factor of 0.99.

TABLE 22. SCENARIO 6 AVERAGE COMPUTER RESOURCE UTILIZATION			
	Unit	**ROS-Unity3D**	**ROS-Gazebo**
# CPU Threads	1	170.33	122
CPU %	%	129.29	112.14
Memory	MB	876.82	596.59
GPU Framebuffer Memory	MB	2658.16	276
GPU Streaming Multiprocessor	%	16.68	3.25
Real-Time Factor	1	1.00 (28.25 FPS)	0.99

Localization performance is shown in Table 23. Comparing the localization of the simulators side by side, error in ROS-Gazebo is about twice as large as the error in ROS-Unity3D. The higher error in ROS-Gazebo is attributed to its poorer quality shadows. The impact is especially large since most of the stereo features that have been identified for localization are far away, leading to increased projection error. As shown in Figure 47, ROS-Unity3D detects nearby features on the edge of the tree shadow in the circled region, whereas the only features detected in ROS-Gazebo are on more distant objects and not in the shadows.

TABLE 23. SCENARIO 6 AVERAGE LOCALIZATION ERRORS						
	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	1.945	0.493	0.329	1.248	0.134	0.090
ROS-Gazebo	3.419	0.901	0.729	2.153	0.253	0.197

The occupancy grids produced from scenario 6 are shown in Figure 48. As expected, the buildings fall outside of the 5 m obstacle detection range and are classified as unknown in the occupancy grid in both ROS-Unity3D and ROS-Gazebo. Additionally, in both simulators the lamp posts beside the sidewalk are consistently detected as obstacles and the trees are partially detected. The primary difference between the two is that in ROS-Unity3D the border between the sidewalk and grass is frequently detected and classified as clear, navigable ground terrain



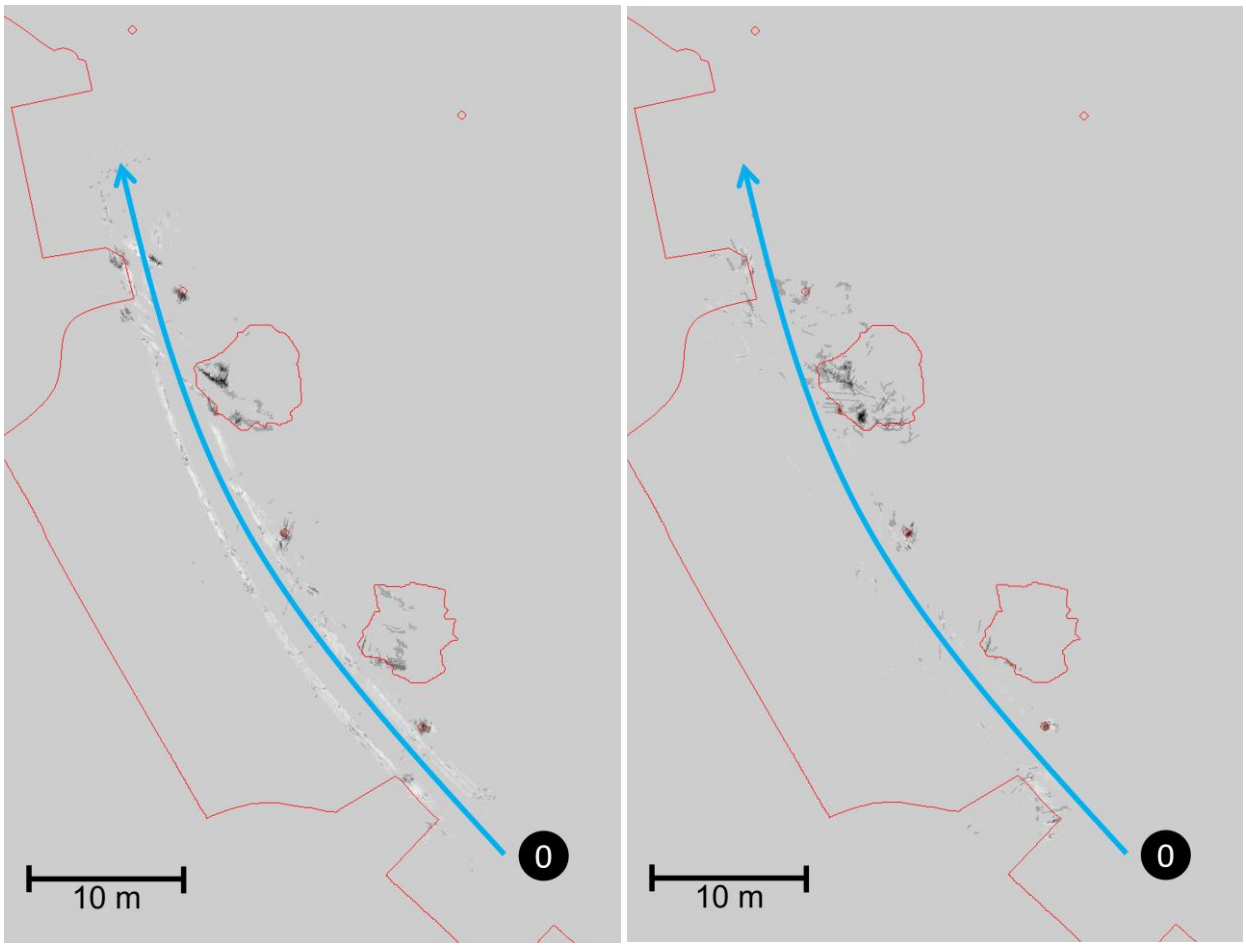
(a) (b)

Figure 47. Scenario 6 RTAB-Map stereo image feature detection simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

while this border is rarely detected in ROS-Gazebo. Outside of the border between the sidewalk and the grass, neither simulator consistently classifies the ground as clear in the occupancy grid. This is because the sidewalk texture is uniform without obvious features. Table 24 states definitively that in ROS-Unity3D more occupancy grid cells were correctly classified as clear while in ROS-Gazebo more cells were correctly classified as occupied. Overall, both simulators produced relatively sparse occupancy grids indicating that well populated occupancy grids produced by stereo camera-based SLAM systems require higher resolution textures than are available from Google Maps via the Maps Model Importer.

6.7. Scenario 7: Lunar Surface

The average scenario 7 computer resource utilization metrics are shown in Table 25. In contrast to the other simulation scenarios, here ROS-Gazebo has higher CPU and significantly higher memory usage compared to ROS-Unity3D. This is indicative of the fact that ROS-Gazebo does not scale to large world sizes as well as ROS-Unity3D. Despite this, ROS-Gazebo still under utilizes the available GPU resources compared to ROS-Unity3D.



(a) (b)

Figure 48. Aggregated occupancy grids from scenario 6 with ground truth reference simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

TABLE 24. SCENARIO 6 AVERAGE MAPPING ACCURACY STATISTICS						
Ground Truth	**Occupied**			**Clear**		
Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	743	239.8	202,365.2	4,435.4	908.4	719,136.2
ROS-Gazebo	841.8	241.4	202,264.8	656.8	671.6	723,151.6

TABLE 25. SCENARIO 7 AVERAGE COMPUTER RESOURCE UTILIZATION			
	Unit	**ROS-Unity3D**	**ROS-Gazebo**
# CPU Threads	1	171.6	122
CPU %	%	140.46	149.41
Memory	MB	1,344.1	5,267.2
GPU Framebuffer Memory	MB	2,798.6	590
GPU Streaming Multiprocessor	%	19.71	14.96
Real-Time Factor	1	1.00 (26.93 FPS)	0.99

In terms of simulation speed, ROS-Unity3D ran at a variable time step update rate of 26.93 FPS. As in every other scenario, ROS-Unity3D ran at a time scale of 1 with a fixed time step update rate of 100. ROS-Gazebo achieves an average real-time factor of 0.99. Thus, both simulators operate in real-time.

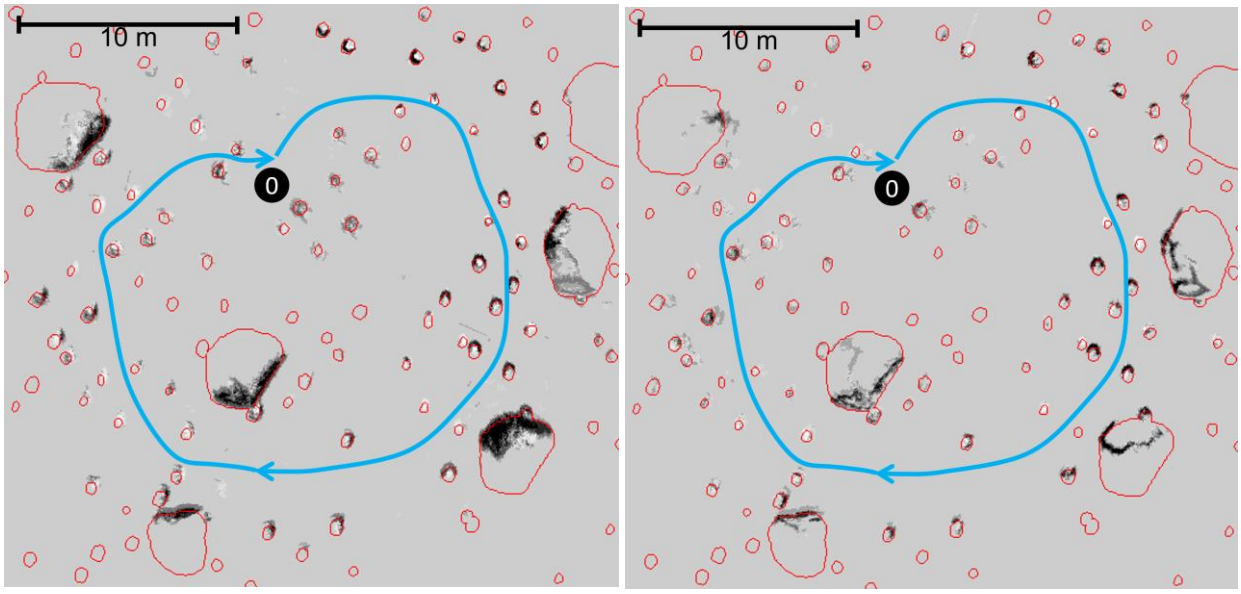
The scenario 7 localization performance of the Jackal UGV simulated in ROS-Unity3D and ROS-Gazebo is shown in Table 26. Both simulators have relatively high localization accuracy owing to the large number of features within the environment. On average, ROS-Gazebo has a slight positional accuracy advantage with a larger advantage in terms of rotational accuracy. In this case, the localization accuracy differences between ROS-Unity3D and ROS-Gazebo are small and are not indicative of a disparity in simulation capabilities.

TABLE 26. SCENARIO 7 AVERAGE LOCALIZATION ERRORS

	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	1.442	0.661	0.615	0.273	0.110	0.104
ROS-Gazebo	0.868	0.377	0.212	0.194	0.105	0.080

The occupancy grids produced from scenario 7 simulated in ROS-Unity3D and ROS-Gazebo are shown in Figure 49 while the numerical results are tabulated in Table 27. The results indicate that ROS-Unity3D tended to classify fewer cells as unknown with the trade-off of more misclassifications overall. Considering obstacles, ROS-Unity3D has 88% more correct classifications with a corresponding 46% increase in misclassifications. For clear cells, the comparison is less favorable, with a 30% increase and 72% increase for correct and incorrect classifications, respectively. Visually, the ROS-Unity3D occupancy grid shows much better mapping of the large boulders and slightly better mapping of the small rocks compared to ROS-Gazebo. The difference is attributed to the increased shadow detail in ROS-Unity3D providing additional visual features for mapping. Figure 50 shows RTAB-Map stereo feature detections

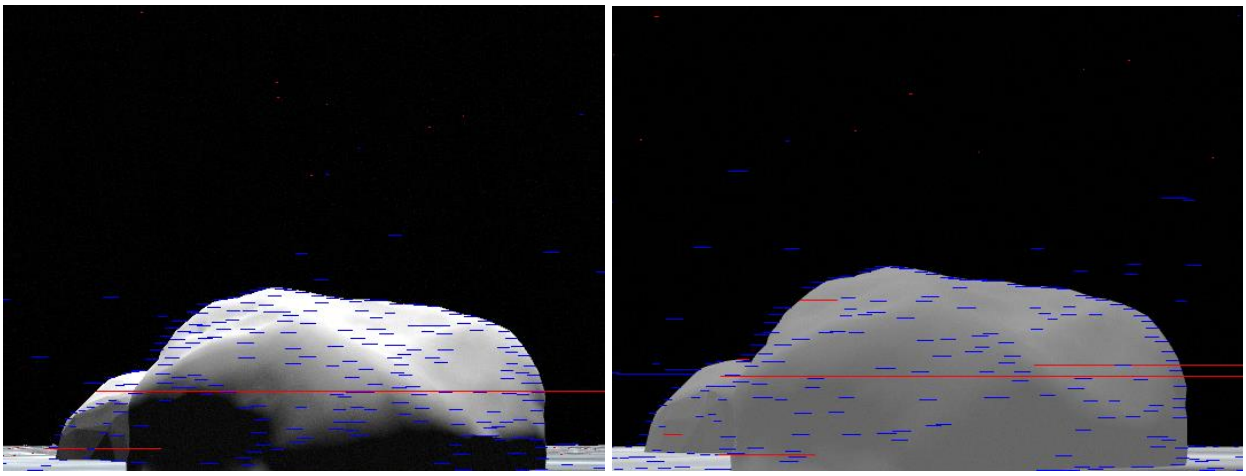
during a scenario 7 run, clearly showing much deeper shadow shading in the ROS-Unity3D image on the left. The ROS-Unity3D image shows that 179 features have been identified on the rock in the center of the frame, each represented as a short blue line showing the distance to the same feature in the other image of the stereo pair. Many features are identified along the edge of a shadow since this is a consistent visual fiducial that can be found in both stereo images. In ROS-Gazebo, only 136 features are identified. Ultimately, the lower level of shadow detail in the ROS-Gazebo simulation environment leads the mapping system to leave more cells classified as unknown relative to ROS-Unity3D. It is expected that in an environment with less visual detail, the lack of shadow detail would have a large impact on the localization system in ROS-Gazebo.



(a) (b)

Figure 49. Aggregated occupancy grids from scenario 7 with ground truth reference simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

TABLE 27. SCENARIO 7 AVERAGE MAPPING ACCURACY STATISTICS						
Ground Truth	**Occupied**			**Clear**		
Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	3,874.4	1,976.6	24,576.0	1,535.8	2,265.8	267,511.4
ROS-Gazebo	2,064.6	1,355.2	27,007.2	1,182.8	1,318.8	268,811.4



(a) (b)

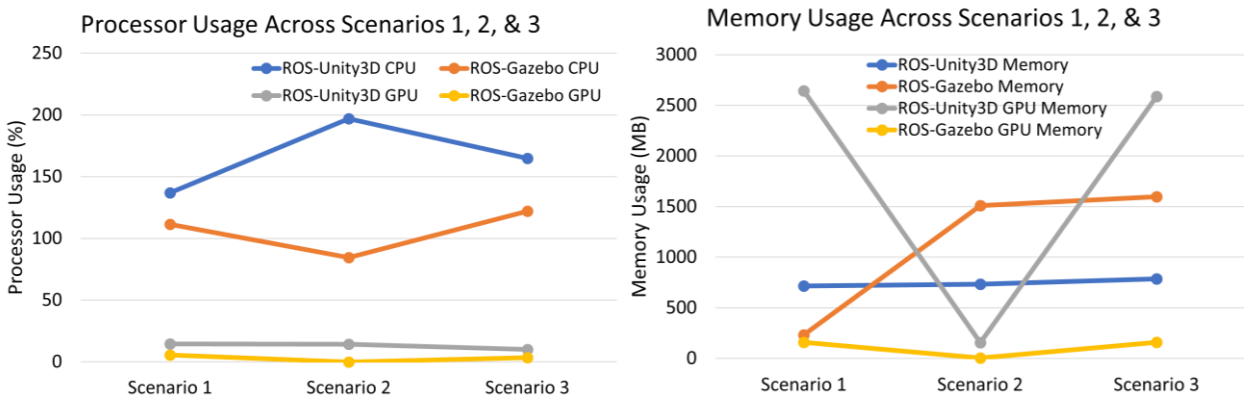
Figure 50. Scenario 7 RTAB-Map stereo image feature detection simulated in (a) ROS-Unity3D and (b) ROS-Gazebo.

6.8. Comparison

Beyond discussing the results of each test scenario individually, they can also be compared to one another to gain more insight into the similarities and differences between ROS-Unity3D and ROS-Gazebo. First, the results from scenarios 1, 2, and 3 are compared. These scenarios use the same HRATC environment and path but consider different sensor packages. Comparing these scenarios reveals differences specific to the sensors and their implementations. Next, the results from scenarios 1, 4, 5, 6, and 7 are compared because they all use the same stereo camera system in different environments. Therefore, comparing the results of these scenarios indicates how the simulators are affected by environment size and appearance.

6.8.1 Differing Sensor Scenarios

Comparing the computer resource results across the first three scenarios, as summarized in Figure 51, makes it clear that the simulators use widely differing sensor implementations. Going from scenario 1 to scenario 2, the CPU and memory usage of ROS-Unity3D increased modestly while ROS-Gazebo CPU usage decreased and memory usage increased six-fold. This suggests that the ROS-Gazebo LiDAR sensor implementation is highly memory intensive while



(a) (b)

Figure 51. Computer resource usage summary for scenarios 1, 2, and 3 divided between (a) processor usage and (b) memory usage.

the ROS-Unity3D LiDAR sensor implementation is moderately CPU intensive. The multithreading of the ROS-Unity3D LiDAR sensor implementation is evidenced by the two additional threads while in ROS-Gazebo the CPU thread count decreased by one. Without the need to render camera images in scenario 2, both simulators use fewer GPU resources, though ROS-Gazebo consistently uses fewer. Moving on to scenario 3, ROS-Gazebo uses more resources across almost all metrics compared to either scenario. This is expected since both the stereo camera and the LiDAR must be simulated. Interestingly, in ROS-Unity3D, CPU and GPU usage decrease relative to scenario 2. This indicates that ROS-Unity3D can manage its resources better than ROS-Gazebo as the number of simulated sensors increases.

That conclusion is further supported by comparing the simulation speeds across scenarios 1, 2, and 3. As summarized in Figure 52, ROS-Unity3D maintained real-time simulation in all cases while ROS-Gazebo only could in Scenario 1. In Scenario 2, ROS-Gazebo slowed to a real-time factor of 0.74 while ROS-Unity3D achieved its highest framerate of 220.13 FPS owing to its superior multithreaded LiDAR implementation. In both simulators, combining the stereo camera and LiDAR for scenario 3 reduces simulation speed more than using either sensor alone.

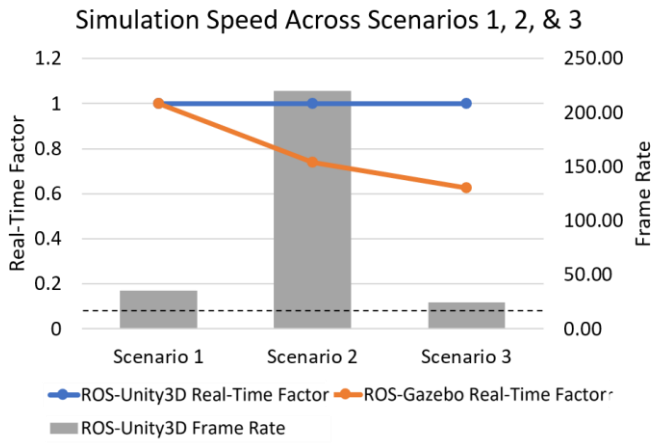


Figure 52. Simulation speed across scenarios 1, 2, and 3.

This is intuitive since more must be simulated, but suggests that, at least for ROS-Unity3D, simulation speed is not strongly correlated with computer resource usage.

In terms of localization accuracy, ROS-Unity3D and ROS-Gazebo are affected in the same ways by the changes across the first three scenarios. The mean rotational and positional localization accuracy results are summarized in Figure 53. In general, the localization error is lowest in scenario 1, followed by scenario 3, with by far the highest error obtained in scenario 2. The error in scenario 2 is the highest because the LiDAR point clouds used for localization are produced half as often and contain less information compared to the images from the stereo camera system. This agrees with the localization results presented in [26]. Despite using the

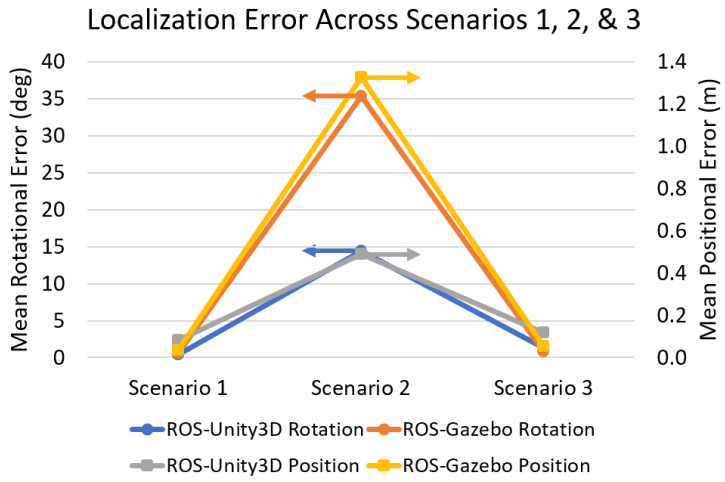
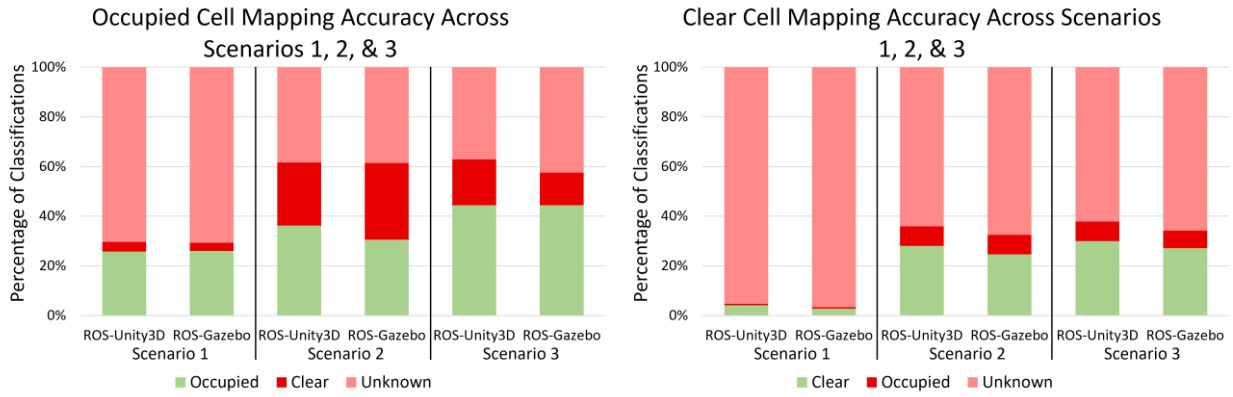


Figure 53. Localization error across scenarios 1, 2, and 3.

same visual odometry system as in scenario 1, localization error in scenario 3 is higher because the LiDAR-based mapping system used in scenario 3 detects more obstacles which must be navigated around, increasing the average distance traveled and thus the average error.

Both simulators also experienced similar trends for mapping accuracy across the first three scenarios, as summarized in Figure 54. The greatest area was classified as unknown in scenario 1 while scenarios 2 and 3 had significantly fewer cells classified as unknown. This is because the LiDAR sensors used for mapping in scenarios 2 and 3 have a much wider field of view and can generate points even where there is little visual detail. This yields more detailed maps compared to visual-based SLAM matching the results demonstrated in [26]. Even though there are fewer cells classified as unknown, there are also more incorrectly classified cells. This is because low resolution LiDAR scans are unable to differentiate between obstacles that are sufficiently close to one another. This causes the clear terrain between obstacles to be misclassified as occupied. In scenario 2, the even larger number of misclassifications can be attributed to its poor localization performance compared to scenario 3.



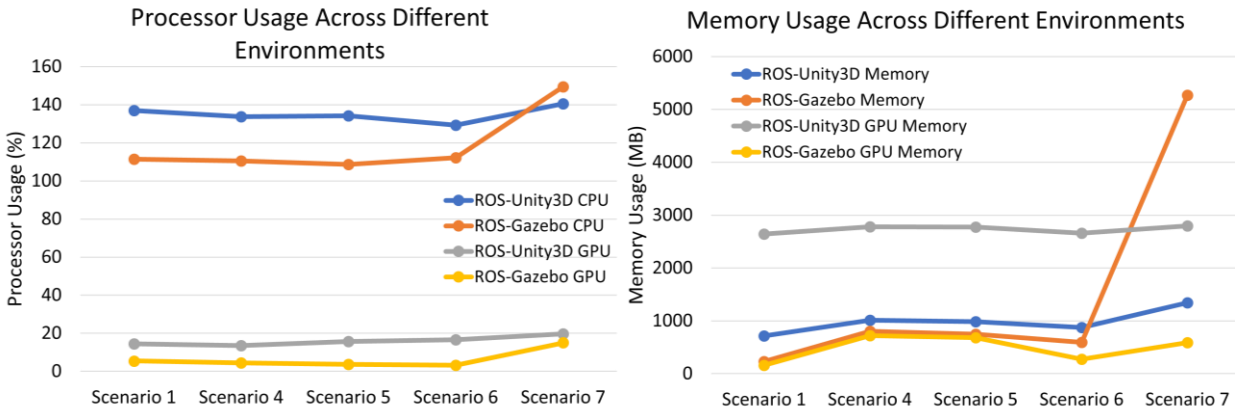
(a) (b)

Figure 54. Mapping accuracy statistics across scenarios 1, 2, and 3 divided between (a) occupied cell mapping accuracy and (b) clear cell mapping accuracy.

6.8.2 Differing Environment Scenarios

The computer resource results across scenarios 1, 4, 5, 6, and 7 are summarized in Figure 55. The computer resources used by ROS-Gazebo appear to grow more rapidly than those used by ROS-Unity3D as a function of the world size. Consider that from scenario 1 to scenario 4 as the world size increased from 1.23 MB to 42.4 MB the ROS-Gazebo memory usage increased by 245%, about 570 MB, while ROS-Unity3D memory usage only increased by 42%, about 300 MB. A similar trend exists with the GPU framebuffer memory. While ROS-Unity3D uses more computer resources for smaller world sizes, ROS-Gazebo uses more in the case of the 650 MB scenario 7 lunar environment. The 5 GB of memory used by ROS-Gazebo represents a 556% increase over the next highest memory usage in scenario 4, whereas ROS-Unity3D only experiences a 33% increase. This indicates that ROS-Gazebo does not scale as well to extremely large environments as ROS-Unity3D.

One interesting observation is that computer resource usage does not scale perfectly with overall world file size. For example, the overall world size of scenario 6 is nearly twice that of scenario 4, yet in both simulators scenario 6 requires less memory. Breaking the overall world file size into the mesh triangles and textures helps to remedy this apparent discrepancy. In the



(a) (b)

Figure 55. Computer resource usage summary for scenarios 1, 4, 5, 6, and 7 divided between (a) processor usage and (b) memory usage.

previous example, the scenario 4 agriculture world has 17,820 mesh triangles and a texture size of 39.7 MB. The scenario 6 UA SEQ world has 330,374 mesh triangles and a texture size of 21.9 MB. Therefore, the decreased memory usage in scenario 6 suggests that texture size (which decreased) has a larger impact on simulator memory usage than total number of triangles (which increased). This hypothesis is also supported by the fact that less memory is used when simulating scenario 1 compared to either scenarios 4 or 5, despite it having a greater number of mesh triangles. Still, the number of mesh triangles does have an impact on resource usage. The scenario 7 lunar surface environment has the second smallest texture size but requires the most memory and CPU resources because of its extremely large number of mesh triangles. The point at which the number of triangles outweighs the texture size will require future investigation.

As opposed to computer resource usage, simulation speed decreased or stayed the same as world sizes increased, as shown in Figure 56. In ROS-Unity3D, the variable time step update rate decreased slightly as the world sizes increased, always remaining above the 20 FPS required for real-time simulation. In ROS-Gazebo, the real-time factor fell to 0.99 in scenario 4 but did not significantly decrease further for larger world sizes.

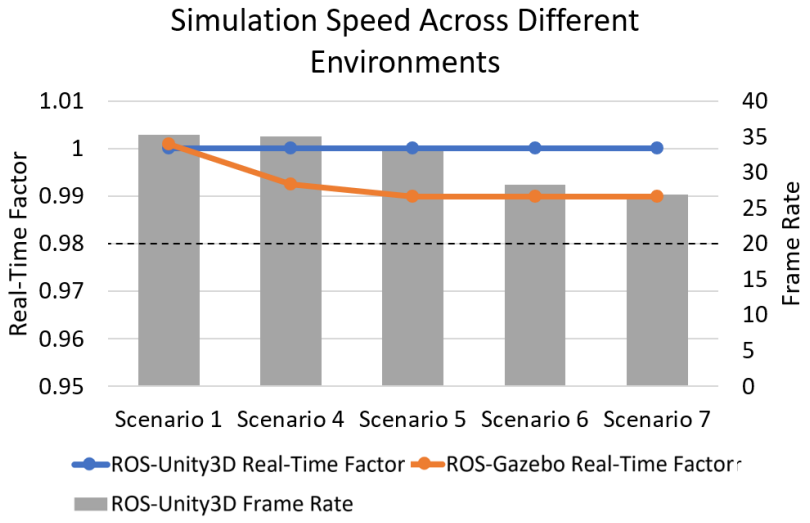


Figure 56. Simulation speed summary for scenarios 1, 4, 5, 6, and 7.

Localization accuracy in the two simulators is strongly affected by the path taken by the Jackal UGV and the contents of the environment. Localization accuracy results are summarized in Figure 57. Environments with short, simple paths and a greater amount of visual detail in textures, models, and shadows lead to greater localization accuracy. The detail in the ground texture and tree models of scenario 1 leads to similar localization performance in both simulators. Compared to scenario 1, scenario 4 uses several high-resolution textures but contains many shadows and has a path that is twice as long. Localization errors in ROS-Unity3D are around twice that of scenario 1, consistent with the longer path length, but errors are significantly higher in ROS-Gazebo because of the lack of detail in the shadows. Scenario 5 uses several high-resolution textures like scenario 4, but with very few shadows and a shorter path like scenario 1. Accordingly, the localization errors for scenario 5 are nearly the same as or lower than those attained in scenario 1. Performance in both simulators is similar because of the lack of shadows. Any decrease in error can be attributed to the higher resolution textures and the simpler single waypoint path compared to the branching path used in scenario 1. The UA SEQ environment used in scenario 6 uses a similar path length to scenario 4 with low resolution textures and

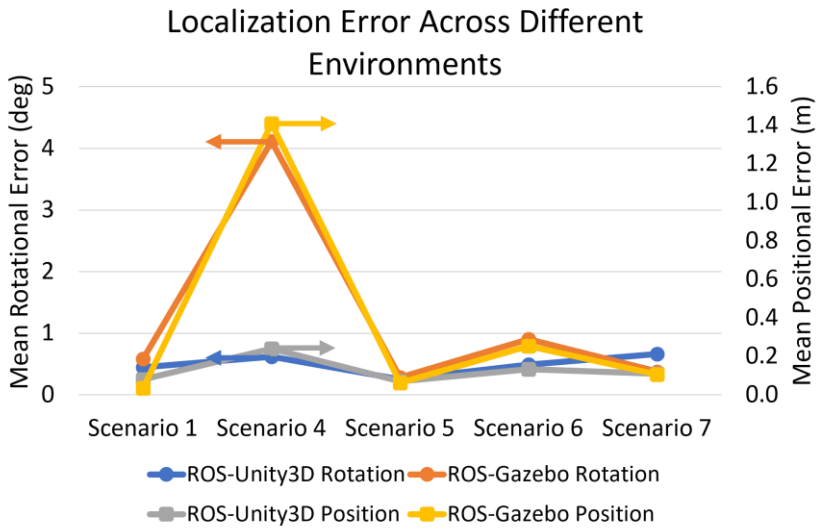


Figure 57. Mean localization error summary for scenarios 1, 4, 5, 6, and 7.

models supplemented with some shadow detail causing both simulators to perform worse than in scenarios 1 or 5. ROS-Unity3D performs better than ROS-Gazebo because of its better shadow rendering. Scenario 7 considers another long path through an environment with a low-resolution ground texture and many scattered rock obstacles. Considering the increased path length, both simulators have localization performance on par with scenario 5. This suggests that the detail provided by the rocks in the scenario 7 is equivalent to the detail provided by the high-resolution ground texture in scenario 5. Consistently across the five environments, those with shorter paths and more detail attained greater localization accuracy with the greatest difference between the simulators being that ROS-Unity3D provides more shadow detail.

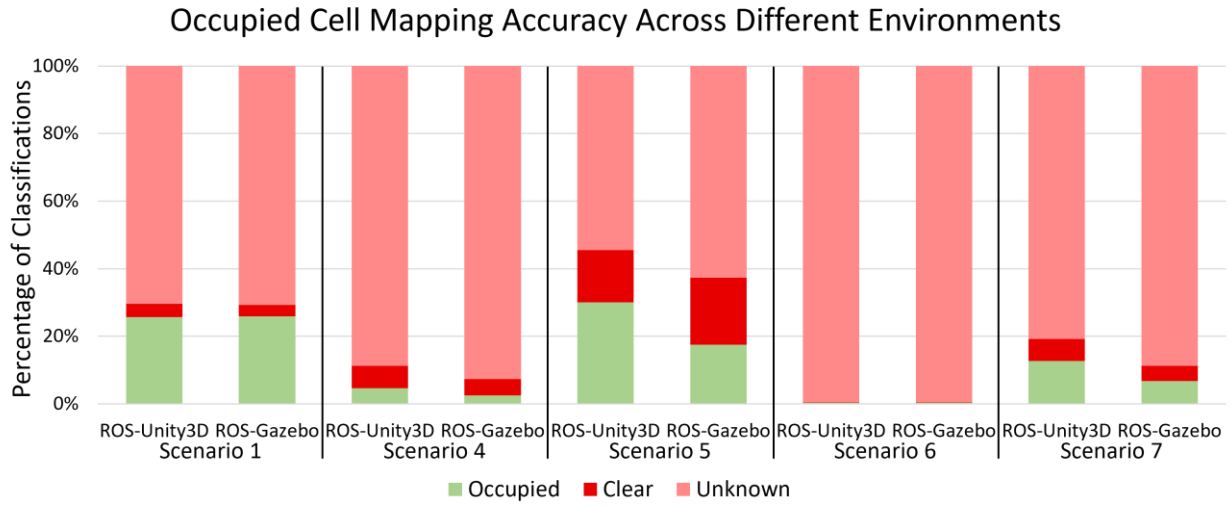
To better understand the impact of environmental detail on localization performance, a modification of scenario 7 in which the small rocks were removed was considered. Localization accuracy for this modified scenario is shown in Table 28 with the full results included in the Appendix. With the removal of the small rocks, the localization system has less detail and the shadows on the boulders have a larger impact. With less detail in general, positional and rotational error increase in both simulators, but ROS-Gazebo is more strongly affected because its shadows have less detail than those produced in ROS-Unity3D.

Across the five environments, mapping performance was most strongly impacted by localization accuracy and ground texture detail. Mapping performance across the five scenarios is summarized in Figure 58. Scenario 1 has good localization performance and thus adequate mapping performance. Some of the ground is properly mapped, but other regions are not. This indicates that the mapping system is limited by the level of detail in the ground texture. In ROS-

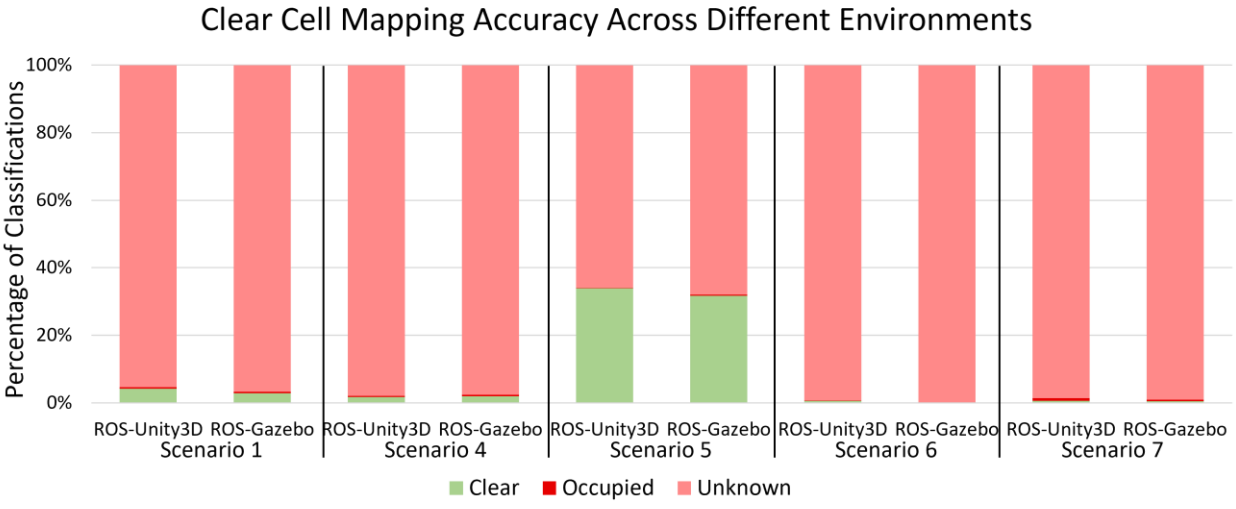
TABLE 28. MODIFIED SCENARIO 7 AVERAGE LOCALIZATION ERRORS

	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	2.497	0.766	0.659	0.810	0.431	0.472
ROS-Gazebo	5.692	3.331	3.894	1.673	0.749	0.567

Gazebo, the scenario 4 mapping accuracy is hindered by the low localization accuracy. Even in ROS-Unity3D, almost none of the ground is detected as clear space in scenario 4 because of the shadows along the path and the lack of apparent detail in the ground texture. Scenario 5 arguably demonstrates the best mapping performance with the pipe obstacle clearly visible and good coverage of the clear ground. This is consistent with the high level of detail in the texture of the ground and the lack of shadows along the path. The importance of detailed ground textures is shown in Figure 59. This figure compares RTAB-Map stereo image feature detection in ROS-



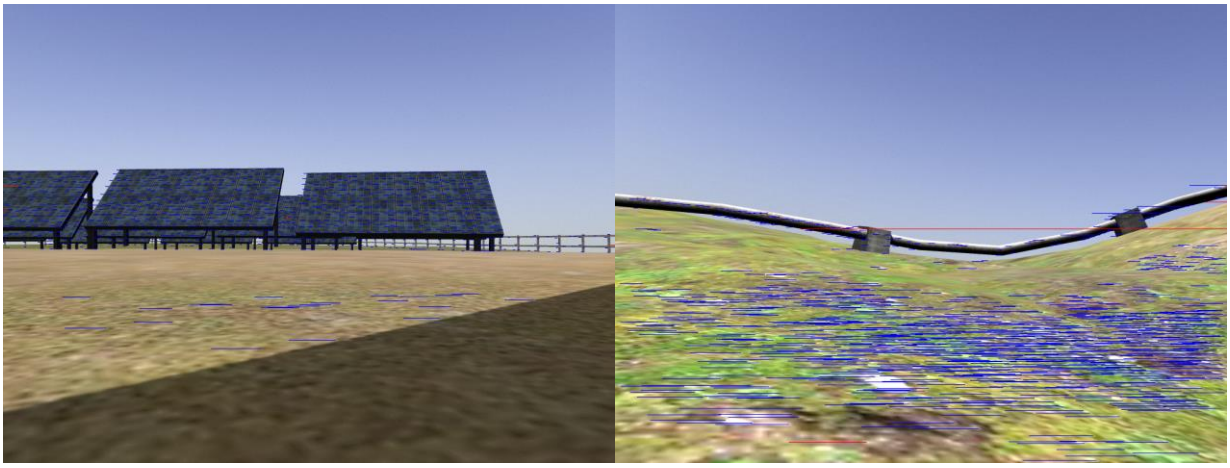
(a)



(b)

Figure 58. Mapping accuracy statistics across scenarios 1, 4, 5, 6, and 7 divided between (a) occupied cell mapping accuracy and (b) clear cell mapping accuracy.

Gazebo for scenarios 4 and 5. In scenario 4, the blue lines representing detected stereo features are sparse on the ground and completely absent from the region in the shadows causing poor localization accuracy and preventing the ground from being mapped. This is different to the image from scenario 5 where numerous stereo features are detected on the highly detailed ground texture leading to high localization accuracy and accurate mapping. The same is true for ROS-Unity3D even though it renders more detail in shadowed regions. Scenarios 6 and 7 both have small texture files which are applied over large areas, causing a lack of detail in the ground. In these scenarios, ROS-Unity3D seems to achieve slightly better mapping accuracy because of its increased visual detail in and on the edges of shadows. Occupancy grids from both simulators tended to be predominantly obstacles since they tend to have geometric features that can be identified even when texture detail is lacking. Ultimately, the results across these scenarios demonstrate that developing a simulation environment requires careful consideration of texture size to balance the tradeoffs between computer resource usage, simulation speed, localization accuracy, and mapping accuracy.



(a) (b)

Figure 59. ROS-Gazebo RTAB-Map stereo image feature detection in the (a) scenario 4 agriculture world and (b) scenario 5 inspection world.

6.9. Other Simulator Notes

This section discusses differences observed between ROS-Unity3D and ROS-Gazebo during testing that do not fall under any specific test scenario. One such difference is world load times. Both simulators support the ability to quickly reset simulations without fully restarting the simulator interface, which was used to conduct consecutive runs of the same test scenario. However, opening a new simulation environment requires that the simulators be started with a specific project for ROS-Unity3D or world file in the case of ROS-Gazebo, a process which takes more time than simply resetting a simulation. The times required to start each simulator in the five simulation environments are shown numerically in Table 29 and graphically in Figure 60. ROS-Unity3D simulator start times are less than ROS-Gazebo for all environments except the HRATC world. Although these times have no impact on the simulation itself, faster simulator start times are more convenient and facilitate more rapid iterative development. This is especially significant in ROS-Gazebo where the world file must be edited, and the simulator restarted to configure plugin settings.

Beyond its long load times, the lunar surface environment also negatively affects the performance of the Gazebo GUI. Specifically, navigating in the scene view is extremely delayed, with camera movements not occurring until several seconds after the mouse is moved. This has no impact on the performance of ongoing simulations in ROS-Gazebo, however it does increase the difficulty of viewing and making changes to the environment during development. The Unity

TABLE 29. SIMULATOR START TIMES BY ENVIRONMENT

	ROS-Unity3D Start Time (s)	**ROS-Gazebo Start Time (s)**
HRATC World	6.57	5.18
Agriculture World	6.14	8.64
Inspection World	7.16	7.31
UA SEQ	6.79	23.86
Lunar Surface	9.44	92.30

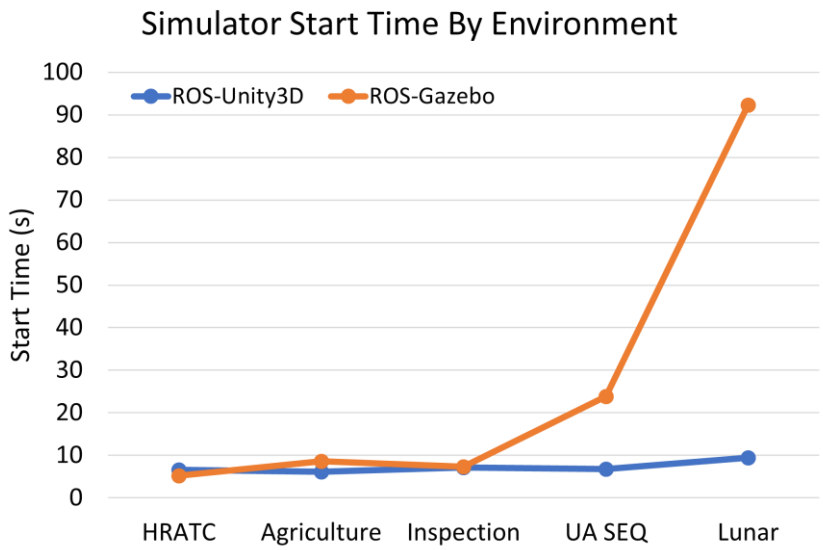
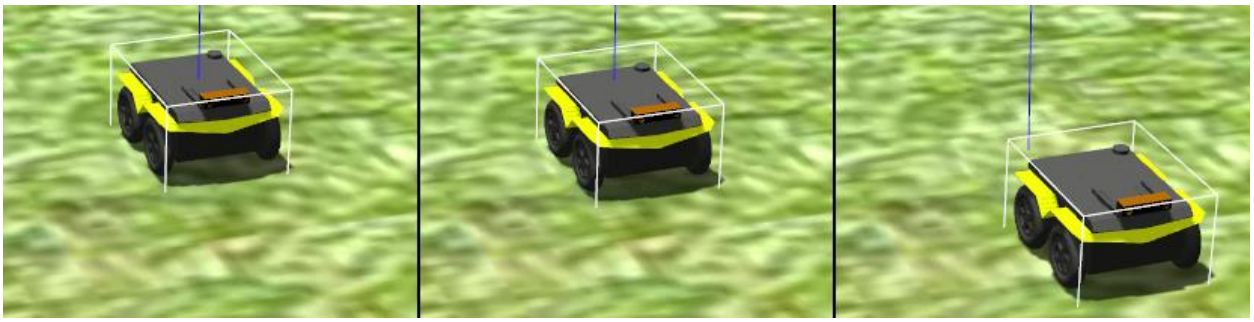


Figure 60. Simulator start time as a function of the environment.

Editor notably allows for smooth navigation through all the simulation environments, including the lunar surface environment. This is especially important in Unity3D since most edits must occur through the Unity Editor, whereas in Gazebo edits to the world are typically made as direct edits to the Gazebo world file in a text editor.

Another discrepancy observed between ROS-Unity3D and ROS-Gazebo is unintended movement of the Jackal UGV. Specifically, it was observed that in ROS-Gazebo the Jackal UGV will slowly slide down slopes when it should be stationary. A sequence of three images taken from the Gazebo GUI over three seconds are shown in Figure 61. Interestingly, the wheels on the Jackal UGV do not rotate. This behavior was not observed in ROS-Unity3D, despite both simulators being configured with the same friction coefficients. To determine if this was related to wheel slip during Jackal UGV navigation, a simple experiment was run in which the rover was commanded to drive directly forward for 10 s at 0.3 m / s in the HRATC world and again on flat ground. The average distance traveled was 2.86 m and 3.06 m in ROS-Unity3D and ROS-



(a) (b) (c)

Figure 61. Close-up comparison view of the Jackal sliding when it should be stationary after (a) 0 s, (b) 1.5 s, and (c) 3 s.

Gazebo, respectively, with no discernable difference between the HRATC world and flat ground. The fact that the Jackal UGV travel distance is so accurate in ROS-Gazebo suggests that this sliding behavior is not related to navigational wheel slip and therefore does not affect intentional movement. Further investigation is required to determine the origin of the sliding behavior.

A final small experiment was devised to determine the performance penalty associated with the TCP-based connection between Unity3D and ROS. The setup emulates a direct connection between Unity3D and ROS by skipping function calls to send sensor measurements over the TCP connection. As a result, the sensors are still simulated but the effect of the TCP connection is negated. The experiment was performed in the HRATC world considering just the stereo camera, just the LiDAR, and the combination of both for comparison with the results from scenarios 1, 2, and 3. The achieved variable time step update rates for each case were 39.2 FPS, 222.2 FPS, and 28.1 FPS, each representing a small increase over their counterpart. These results indicate that, at best, a more direct connection between ROS and Unity3D would increase the Unity3D variable time step update rate a few FPS over the current TCP-based connection. Note that all simulator comparisons in this study use the slower ROS-Unity3D TCP connection. This imply that without the TCP bottleneck, Unity3D could outperform Gazebo in simulation speed by an even greater amount than is shown in earlier results from the seven test scenarios.

CHAPTER 7:
CONCLUSIONS AND FUTURE WORK

This thesis describes the ROS-Unity3D robotics simulation suite and analyzes its capabilities with respect to the simulation of wheeled ground rovers in comparison to the current standard ROS-Gazebo simulation suite. Through analysis of the simulator architectures, of the process of creating equivalent test scenarios, and of the performance of the simulators while simulating the test scenarios, it is determined that ROS-Unity3D is a viable alternative to ROS-Gazebo. Both are largely compatible with the same file types, meaning the same robot URDF, environment meshes, and texture images can be easily used in both. One observed exception being that ROS-Gazebo is not fully compatible with Collada files from Blender, meaning that ROS-Unity3D has slightly better support for custom environments. Existing ROS-Gazebo sensor implementations cannot be used with ROS-Unity3D, however the ROS-Unity integration and Unity3D scripting engine allow custom sensors to be created. Still, the need to create custom sensor implementations and the slower network-based ROS-Unity interface are disadvantages of the ROS-Unity3D simulation suite in its current state.

As a consequence of their architectural differences, ROS-Unity3D requires more computer resources than ROS-Gazebo when simulating small and medium mesh triangle count environments. However, as world sizes grew, ROS-Gazebo began to consume significantly more computer resources and took longer to start and update the user interface whereas ROS-Unity3D attained consistent simulator performance across all world sizes. Therefore, ROS-Unity3D is

recommended over ROS-Gazebo when seeking to simulate environments with high mesh triangle counts with ROS-Gazebo being preferable for smaller environments.

While simulating the autonomous Jackal UGV performing stereo camera-based SLAM, both simulators achieve real-time performance. They achieve high localization and mapping accuracy in feature rich environments, with a decrease in performance in sparser environments where ROS-Unity performed better. In sparser environments, increased shadow detail in ROS-Unity3D increases the number of visual features available to the autonomy system, leading to improved localization and mapping performance. Consequently, ROS-Unity3D is the preferred simulator in environments where shadows will impact the navigation system.

In terms of sensor simulation, ROS-Gazebo is not able to maintain real-time simulation of the VLP-16 LiDAR sensor, even at a reduced resolution. This contrasts with ROS-Unity3D which can simulate the VLP-16 in real-time, owing to its multithreaded implementation. In fact, ROS-Unity3D can simultaneously simulate the VLP-16 and Bumblebee2 in real-time, while in this scenario ROS-Gazebo falls to a real-time factor of 0.62. Even during slower than real-time simulation, LiDAR-based SLAM in ROS-Gazebo is difficult because of the jerky movement associated with ODE joint motors as opposed to the smoother movement of PhysX articulation joints in ROS-Unity3D. Thus, ROS-Unity3D is suitable for real-time simulation of high-resolution 3D LiDAR scanners which can practically be used for SLAM, while ROS-Gazebo has several shortcomings when it comes to real-time simulation of LiDAR-based SLAM systems.

One potential area for future work is improved sensor implementations and integrations. A GPU accelerated implementation of a LiDAR sensor should be possible in ROS-Unity3D and ROS-Gazebo, possibly allowing for real-time performance at increased resolutions. Additionally, wheel encoder feedback and IMU data can be incorporated into a different SLAM approach

using sensor fusion leading to improved localization and mapping performance in both simulators. In terms of navigation, a more complex approach considering 3D navigation within a 3D occupancy grid could be pursued giving more valuable insights into the fidelity of the simulated environments in ROS-Unity3D and ROS-Gazebo. Simulation of other UGVs and UAVs beyond the Jackal UGV in even more varied environments should be conducted to better understand how different robotic platforms are affected by each physics engine. The performance of ROS-Unity3D and ROS-Gazebo in more constrained computing environments, including those without a discrete GPU, should be investigated.

REFERENCES

[1] C. Iavazzo, X.-E. D. Gkegke, P.-E. Iavazzo, and I. D. Gkegkes, "Evolution of robots throughout history from Hephaestus to Da Vinci Robot.," *Acta Med Hist Adriat*, vol. 12, no. 2, pp. 247–258, 2014.

[2] N. J. Nilsson, "Shakey the robot," no. 323. AI Center, SRI International, Menlo Park, CA, USA, 1984.

[3] S. Putz, T. Wiemann, M. K. Piening, and J. Hertzberg, "Continuous Shortest Path Vector Field Navigation on 3D Triangular Meshes for Mobile Robots," in *2021 IEEE International Conference on Robotics and Automation*, Oct. 2021, pp. 2256–2263. doi: 10.1109/icra48506.2021.9560981.

[4] J. H. Park, T. Y. Uhm, G. D. Bae, and Y. H. Choi, "Stability evaluation of outdoor unmanned security robot in terrain information," in *International Conference on Control, Automation and Systems*, 2018, vol. 2018-Octob, pp. 955–957.

[5] D. Lee, S. Son, K. Yang, J. Park, and H. Lee, "Sensor fusion localization system for outdoor mobile robot," in *ICCAS-SICE 2009 - ICROS-SICE International Joint Conference 2009, Proceedings*, 2009, pp. 1384–1387.

[6] "NASA's Self-Driving Perseverance Mars Rover 'Takes the Wheel' | NASA." https://www.nasa.gov/feature/jpl/nasa-s-self-driving-perseverance-mars-rover-takes-the-wheel (accessed Jan. 13, 2022).

[7] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," *Springer Proceedings in Advanced Robotics*, vol. 5, pp. 621–635, 2018, doi: 10.1007/978-3-319-67361-5_40.

[8] J. Carsten, A. Rankin, D. Ferguson, and A. Stentz, "Global path planning on board the mars exploration rovers," *IEEE Aerospace Conference Proceedings*, 2007, doi: 10.1109/AERO.2007.352683.

[9] "Mars Rover Tests Driving, Drilling and Detecting Life in the Desert | NASA." https://www.nasa.gov/feature/ames/mars-rover-tests-driving-drilling-and-detecting-life-in-chile-s-high-desert (accessed Jan. 13, 2022).

[10] "NASA - NASA, International Partners Test Rovers in Hawaii." https://www.nasa.gov/centers/kennedy/moonandmars/hawaii_testing.html (accessed Jan. 13, 2022).

[11] A. Jain *et al.*, "Recent developments in the ROAMS planetary rover simulation Environment," *IEEE Aerospace Conference Proceedings*, vol. 2, pp. 861–876, 2004, doi: 10.1109/AERO.2004.1367686.

[12] “DARTS Spacecraft Dynamics Simulator.” https://dartslab.jpl.nasa.gov/DARTS/index.php (accessed Jan. 13, 2022).

[13] “Dshell Spacecraft Dynamics Simulator.” https://dartslab.jpl.nasa.gov/DSHELL/index.php (accessed Jan. 13, 2022).

[14] J. J. Biesiadecki, D. A. Henriquez, and A. Jain, “Reusable, real-time spacecraft dynamics simulator,” *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, vol. 2, 1997, doi: 10.1109/DASC.1997.637259.

[15] “ROAMS Spacecraft Dynamics Simulator.” https://dartslab.jpl.nasa.gov/ROAMS/index.php (accessed Jan. 13, 2022).

[16] J. Yen, A. Jain, J. (Bob, and) Balaram, “ROAMS: Rover Analysis, Modeling and Simulation Software,” in *Fifth International Symposium on Artificial Intelligence and Automation in Space*, 1999, pp. 1–3. Accessed: Jan. 13, 2022. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/17412

[17] C. S. Lim and A. Jain, “Dshell++: A component based, reusable space system simulation framework,” *Proceedings - 2009 3rd IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT 2009*, pp. 229–236, 2009, doi: 10.1109/SMC-IT.2009.35.

[18] V. Verma and C. Leger, “SSim: NASA Mars Rover Robotics Flight Software Simulation,” *IEEE Aerospace Conference Proceedings*, vol. 2019-March, Mar. 2019, doi: 10.1109/AERO.2019.8741862.

[19] B. P. Gerkey, R. T. Vaughan, and A. Howard, “The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems,” pp. 317–323, Accessed: Jan. 13, 2022. [Online]. Available: http://playerstage.sf.net.

[20] “Gazebo.” http://www.gazebosim.org/ (accessed Jan. 13, 2022).

[21] M. Quigley *et al.*, “ROS: an open-source Robot Operating System”, Accessed: Jan. 13, 2022. [Online]. Available: http://stair.stanford.edu

[22] “Gazebo : Tutorial : Make a Mobile Robot.” http://www.gazebosim.org/tutorials?tut=build_robot&cat=build_robot (accessed Jan. 13, 2022).

[23] “Gazebo : Tutorial : ROS overview.” http://www.gazebosim.org/tutorials?tut=ros_overview&cat=connect_ros (accessed Jan. 13, 2022).

[24] “Gazebo : Tutorial : Building a world.” https://gazebosim.org/tutorials?tut=build_world&cat=build_world (accessed Jan. 13, 2022).

[25] M. Allan *et al.*, “Planetary Rover Simulation for Lunar Exploration Missions,” *IEEE Aerospace Conference Proceedings*, vol. 2019-March, Mar. 2019, doi: 10.1109/AERO.2019.8741780.

[26] R. Giubilato, A. Masili, S. Chiodini, M. Pertile, and S. Debei, “Simulation framework for mobile robots in planetary-like environments,” *2020 IEEE International Workshop on Metrology for AeroSpace, MetroAeroSpace 2020 - Proceedings*, pp. 594–599, Jun. 2020, doi: 10.1109/METROAEROSPACE48742.2020.9160154.

[27] W. A. Mattingly, D. J. Chang, R. Paris, N. Smith, J. Blevins, and M. Ouyang, “Robot design using unity for computer games and robotic simulations,” *Proceedings of CGAMES’2012 USA - 17th International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational and Serious Games*, pp. 56–59, 2012, doi: 10.1109/CGAMES.2012.6314552.

[28] “How Unity3D Became a Game-Development Beast.” https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/ (accessed Jan. 13, 2022).

[29] “Create 3D animation, CGI film, & cinematic trailers in real-time | Animation software | Unity.” https://unity.com/solutions/film-animation-cinematics (accessed Jan. 13, 2022).

[30] “Unity Asset Store - The Best Assets for Game Making.” https://assetstore.unity.com/ (accessed Jan. 13, 2022).

[31] M. Yamaura *et al.*, “ADAS Virtual Prototyping using Modelica and Unity Co-simulation via OpenMETA,” in *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*, May 2016, vol. 124, pp. 43–49. doi: 10.3384/ecp1612443.

[32] T. Niemirepo, J. Toivonen, M. Viitanen, and J. Vanne, “Open-Source CiThruS Simulation Environment for Real-Time 360-Degree Traffic Imaging,” in *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, Nov. 2019, pp. 1–5. doi: 10.1109/ICCVE45908.2019.8965242.

[33] R. Codd-Downey, P. M. Forooshani, A. Speers, H. Wang, and M. Jenkin, “From ROS to unity: Leveraging robot and virtual environment middleware for immersive teleoperation,” *2014 IEEE International Conference on Information and Automation, ICIA 2014*, pp. 932–936, Oct. 2014, doi: 10.1109/ICINFA.2014.6932785.

[34] W. Meng, Y. Hu, J. Lin, F. Lin, and R. Teo, “ROS+unity: An efficient high-fidelity 3D multi-UAV navigation and control simulator in GPS-denied environments,” in *IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society*, Nov. 2015, pp. 002562–002567. doi: 10.1109/IECON.2015.7392488.

[35] Y. Mizuchi and T. Inamura, “Cloud-based multimodal human-robot interaction simulator utilizing ROS and unity frameworks,” *SII 2017 - 2017 IEEE/SICE International Symposium on System Integration*, vol. 2018-Janua, pp. 948–955, Feb. 2018, doi: 10.1109/SII.2017.8279345.

[36] Q. Zhang and Y. Zhao, “Implementation and Verification of a Virtual Testing System Based on ROS and Unity for Computer Vision Algorithms,” *Proceedings - 2019 12th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics, CISP-BMEI 2019*, Oct. 2019, doi: 10.1109/CISP-BMEI48845.2019.8965907.

[37] M. Bischoff, “GitHub - siemens/ros-sharp.” https://github.com/siemens/ros-sharp (accessed Jan. 18, 2022).

[38] A. Hussein, F. Garcia, and C. Olaverri-Monreal, “ROS and Unity Based Framework for Intelligent Vehicles Control and Simulation,” *2018 IEEE International Conference on Vehicular Electronics and Safety, ICVES 2018*, Oct. 2018, doi: 10.1109/ICVES.2018.8519522.

[39] M. Santos Pessoa de Melo, J. Gomes da Silva Neto, P. Jorge Lima da Silva, J. M. X. Natario Teixeira, and V. Teichrieb, “Analysis and Comparison of Robotics 3D Simulators,” in *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, Oct. 2019, pp. 242–251. doi: 10.1109/SVR.2019.00049.

[40] A. Konrad, “Simulation of Mobile Robots with Unity and ROS : A Case-Study and a Comparison with Gazebo,” 2019, Accessed: Jan. 19, 2022. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:hv:diva-14019

[41] “Teaching robots to see with Unity | Unity Blog.” https://blog.unity.com/technology/teaching-robots-to-see-with-unity (accessed Jan. 20, 2022).

[42] H. Anand *et al.*, “OpenUAV Cloud Testbed: A Collaborative Design Studio for Field Robotics,” *IEEE International Conference on Automation Science and Engineering*, vol. 2021-Augus, pp. 724–731, Aug. 2021, doi: 10.1109/CASE49439.2021.9551638.

[43] C. Piyavichayanon and M. Koga, “Validation of Robot Model with Mobile Augmented Reality,” *2021 6th Asia-Pacific Conference on Intelligent Robot Systems, ACIRS 2021*, pp. 1–5, Jul. 2021, doi: 10.1109/ACIRS52449.2021.9519362.

[44] J. J. Roldán *et al.*, “Multi-robot Systems, Virtual Reality and ROS: Developing a New Generation of Operator Interfaces,” *Studies in Computational Intelligence*, vol. 778, pp. 29–64, 2019, doi: 10.1007/978-3-319-91590-6_2.

[45] Z. Shi and C. L. R. McGhan, “Affordable Virtual Reality Setup for Educational Aerospace Robotics Simulation and Testing,” *Journal of Aerospace Information Systems*, vol. 17, no. 1, pp. 66–69, Dec. 2019, doi: 10.2514/1.I010723.

[46] C. Sevastopoulos and S. Konstantopoulos, “A Simulated Environment for Traversability Estimation Experiments in Field Robotics Applications,” *ACM International Conference Proceeding Series*, pp. 256–257, Jun. 2021, doi: 10.1145/3453892.3462214.

[47] “Key project-wide settings in Unity.” Arm Limited, 2020. Accessed: Mar. 15, 2022. [Online]. Available: http://www.arm.com/company/policies/trademarks.

[48] “Unity - Manual: GameObjects.” https://docs.unity3d.com/Manual/GameObjects.html (accessed Mar. 15, 2022).

[49] “Unity - Manual: Rigidbody.” https://docs.unity3d.com/Manual/class-Rigidbody.html (accessed Mar. 15, 2022).

[50] “Unity - Manual: Articulation Body.” https://docs.unity3d.com/Manual/class-ArticulationBody.html (accessed Mar. 15, 2022).

[51] “Unity - Manual: Creating and Using Scripts.” https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html (accessed Mar. 15, 2022).

[52] “Unity - Manual: Important Classes - MonoBehaviour.” https://docs.unity3d.com/Manual/class-MonoBehaviour.html (accessed Mar. 15, 2022).

[53] “Gazebo : Tutorial : Gazebo Components.” https://gazebosim.org/tutorials?tut=components&cat=get_started#ModelFiles (accessed Mar. 15, 2022).

[54] “Creating Worlds — Documentation.” http://sdformat.org/tutorials?tut=spec_world&cat=specification& (accessed Mar. 15, 2022).

[55] “SDFormat Specification.” http://sdformat.org/spec (accessed Mar. 15, 2022).

[56] “Gazebo : Tutorial : Plugins 101.” http://gazebosim.org/tutorials?tut=plugins_hello_world&cat=write_plugin (accessed Mar. 16, 2022).

[57] “Gazebo : Tutorial : Model plugins.” http://gazebosim.org/tutorials?tut=plugins_model&cat=write_plugin (accessed Mar. 16, 2022).

[58] “ROS-TCP-Connector/ROSGeometry.md at main · Unity-Technologies/ROS-TCP-Connector · GitHub.” https://github.com/Unity-Technologies/ROS-TCP-Connector/blob/main/ROSGeometry.md (accessed Mar. 19, 2022).

[59] “Unity - Scripting API: Quaternion.Euler.” https://docs.unity3d.com/ScriptReference/Quaternion.Euler.html (accessed Mar. 19, 2022).

[60] “Unity - Scripting API: Vector3.Cross.” https://docs.unity3d.com/ScriptReference/Vector3.Cross.html (accessed Mar. 19, 2022).

[61] “Unity - Manual: Glossary.” https://docs.unity3d.com/Manual/Glossary.html#Unityunit (accessed Mar. 19, 2022).

[62] “Unity - Scripting API: Rigidbody.AddForce.” https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html (accessed Mar. 19, 2022).

[63] “Unity - Manual: Rotation and Orientation in Unity.” https://docs.unity3d.com/Manual/QuaternionAndEulerRotationsInUnity.html (accessed Mar. 19, 2022).

[64] “Gazebo Gazebo: World File Syntax.” http://playerstage.sourceforge.net/doc/Gazebo-manual-svn-html/config_syntax.html (accessed Mar. 19, 2022).

[65] “Unity - Manual: Important Classes - Time.” https://docs.unity3d.com/Manual/TimeFrameManagement.html (accessed Mar. 19, 2022).

[66] “Unity - Scripting API: MonoBehaviour.Update().” https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html (accessed Mar. 19, 2022).

[67] “Gazebo : Tutorial : Physics Parameters.” http://gazebosim.org/tutorials?tut=physics_params&cat=physics (accessed Mar. 20, 2022).

[68] “Gazebo : Tutorial : Beginner: GUI.” http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b2 (accessed Mar. 21, 2022).

[69] “Gazebo : Tutorial : Performance metrics.” http://gazebosim.org/tutorials?tut=performance_metrics&cat=tools_utilities (accessed Mar. 20, 2022).

[70] “Physics updates in Unity 2019.3 | Unity Blog.” https://blog.unity.com/technology/physics-updates-in-unity-2019-3 (accessed Mar. 20, 2022).

[71] “PhysX SDK | NVIDIA Developer.” https://developer.nvidia.com/physx-sdk (accessed Mar. 20, 2022).

[72] “Rigid Body Dynamics — NVIDIA PhysX SDK 4.0 Documentation.” https://gameworksdocs.nvidia.com/PhysX/4.0/documentation/PhysXGuide/Manual/RigidBodyDynamics.html#applying-forces-and-torques (accessed Mar. 20, 2022).

[73] “Advanced Collision Detection — NVIDIA PhysX SDK 4.0 Documentation.” https://gameworksdocs.nvidia.com/PhysX/4.0/documentation/PhysXGuide/Manual/AdvancedCollisionDetection.html (accessed Mar. 20, 2022).

[74] “Unity - Manual: Physics.” https://docs.unity3d.com/Manual/class-PhysicsManager.html (accessed Mar. 20, 2022).

[75] “Rigid Body Collision — NVIDIA PhysX SDK 4.0 Documentation.” https://gameworksdocs.nvidia.com/PhysX/4.0/documentation/PhysXGuide/Manual/RigidBodyCollision.html#broad-phase-algorithms (accessed Mar. 20, 2022).

[76] “Unity - Manual: Colliders.” https://docs.unity3d.com/Manual/CollidersOverview.html (accessed Mar. 20, 2022).

[77] “Unity - Manual: Physic Material.” https://docs.unity3d.com/Manual/class-PhysicMaterial.html (accessed Mar. 20, 2022).

[78] “Articulations — NVIDIA PhysX SDK 4.0 Documentation.” https://gameworksdocs.nvidia.com/PhysX/4.0/documentation/PhysXGuide/Manual/Articulations.html (accessed Mar. 20, 2022).

[79] “Open Dynamics Engine.” https://www.ode.org/ (accessed Mar. 20, 2022).

[80] “Manual - ODE.” http://ode.org/wiki/index.php?title=Manual (accessed Mar. 20, 2022).

[81] “How Collision Detection Works - ODE.” http://ode.org/wiki/index.php?title=How_Collision_Detection_Works (accessed Mar. 21, 2022).

[82] “Unity - Manual: Model file formats.” https://docs.unity3d.com/Manual/3D-formats.html (accessed Mar. 21, 2022).

[83] “GitHub - Unity-Technologies/URDF-Importer: URDF importer.” https://github.com/Unity-Technologies/URDF-Importer (accessed Mar. 21, 2022).

[84] “Unity - Manual: Importing Textures.” https://docs.unity3d.com/Manual/ImportingTextures.html (accessed Mar. 21, 2022).

[85] “Gazebo : Tutorial : Make a model.” https://gazebosim.org/tutorials?tut=build_model (accessed Mar. 21, 2022).

[86] “Gazebo : Tutorial : URDF in Gazebo.” http://gazebosim.org/tutorials/?tut=ros_urdf (accessed Mar. 21, 2022).

[87] “A Brief Summary of Image File Formats | Ogre Wiki.” https://wiki.ogre3d.org/A+Brief+Summary+of+Image+File+Formats (accessed Mar. 21, 2022).

[88] “Unity - Manual: Unity’s interface.” https://docs.unity3d.com/2022.2/Documentation/Manual/UsingTheEditor.html (accessed Mar. 21, 2022).

[89] “Unity - Manual: Console Window.” https://docs.unity3d.com/2022.2/Documentation/Manual/Console.html (accessed Mar. 21, 2022).

[90] “Unity - Manual: Debug C# code in Unity.” https://docs.unity3d.com/Manual/ManagedCodeDebugging.html (accessed Mar. 21, 2022).

[91] “Gazebo : Tutorial : Model Editor.” http://gazebosim.org/tutorials?tut=model_editor (accessed Mar. 21, 2022).

[92] “Gazebo : Tutorial : From source (Ubuntu and Mac).” http://gazebosim.org/tutorials?tut=install_from_source (accessed Mar. 21, 2022).

[93] “ROS-Unity Integration.” https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/ros_unity_integration/README.md (accessed Mar. 22, 2022).

[94] “ROSGeometry component.” https://github.com/Unity-Technologies/ROS-TCP-Connector/blob/main/ROSGeometry.md (accessed Mar. 22, 2022).

[95] “Robotics Navigation 2 SLAM Example.” https://github.com/Unity-Technologies/Robotics-Nav2-SLAM-Example (accessed Mar. 24, 2022).

[96] “Gazebo : Tutorial : ROS communication.” http://gazebosim.org/tutorials/?tut=ros_comm (accessed Mar. 22, 2022).

[97] “Jackal UGV - Small Weatherproof Robot - Clearpath.” https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/ (accessed Mar. 23, 2022).

[98] “Simulating Jackal.” http://www.clearpathrobotics.com/assets/guides/noetic/jackal/simulation.html (accessed Mar. 23, 2022).

[99] “Unity Robotics Hub.” https://github.com/Unity-Technologies/Unity-Robotics-Hub (accessed Mar. 24, 2022).

[100] “What is the maximum frame rate of a Bumblebee? | Teledyne FLIR.” https://www.flir.com/support-center/iis/machine-vision/knowledge-base/what-is-the-maximum-frame-rate-of-a-bumblebee/ (accessed Mar. 24, 2022).

[101] “pointgrey_camera_driver.” https://github.com/ros-drivers/pointgrey_camera_driver (accessed Mar. 24, 2022).

[102] “Velodyne VLP-16.” https://www.mapix.com/lidar-scanner-sensors/velodyne/velodyne-puck-vlp16/ (accessed Mar. 24, 2022).

[103] “3DM-GX3® -25 | LORD Sensing Systems.” https://www.microstrain.com/inertial/3dm-gx3-25 (accessed Mar. 26, 2022).

[104] “hector_gazebo_plugins - ROS Wiki.” http://wiki.ros.org/hector_gazebo_plugins (accessed Mar. 26, 2022).

[105] K. J. de Jesus, H. J. Kobs, A. R. Cukla, M. A. de Souza Leite Cuadros, and D. F. T. Gamarra, "Comparison of Visual SLAM Algorithms ORB-SLAM2, RTAB-Map and SPTAM in Internal and External Environments with ROS," pp. 216–221, Nov. 2021, doi: 10.1109/LARS/SBR/WRE54079.2021.9605432.

[106] P. Sankalprajan, T. Sharma, H. D. Perur, and P. Sekhar Pagala, "Comparative analysis of ROS based 2D and 3D SLAM algorithms for autonomous ground vehicles," Jun. 2020. doi: 10.1109/INCET49848.2020.9154101.

[107] "IEEE RAS–SIGHT Humanitarian Robotics & Automation Technology Challenge." http://www.inf.ufrgs.br/hratc2017/HRATC2017/Welcome.html (accessed Mar. 28, 2022).

[108] "GitHub - clearpathrobotics/cpr_gazebo." https://github.com/clearpathrobotics/cpr_gazebo (accessed Mar. 29, 2022).

[109] M. M. M. Manhães, S. A. Scherer, M. Voss, L. R. Douat, and T. Rauschenbach, "UUV Simulator: A Gazebo-based package for underwater intervention and multi-robot simulation," *OCEANS 2016 MTS/IEEE Monterey, OCE 2016*, Nov. 2016, doi: 10.1109/OCEANS.2016.7761080.

[110] "GitHub - eliemichel/MapsModelsImporter." https://github.com/eliemichel/MapsModelsImporter (accessed Mar. 29, 2022).

[111] "Moon Trek API." https://trek.nasa.gov/tiles/apidoc/trekAPI.html?body=moon (accessed Mar. 29, 2022).

[112] LOLA Instrument Team, "LOLA DEM Southpole 10m, ColorHillshade." Goddard Space Flight Center, Greenbelt, Maryland, 2011. [Online]. Available: http://trek.nasa.gov

[113] "Welcome to the QGIS project!" https://www.qgis.org/en/site/ (accessed Mar. 29, 2022).

[114] "GitHub - domlysz/BlenderGIS." https://github.com/domlysz/BlenderGIS (accessed Mar. 29, 2022).

[115] "GitHub - giampaolo/psutil." https://github.com/giampaolo/psutil (accessed Apr. 05, 2022).

[116] "NVIDIA System Management Interface." https://developer.nvidia.com/nvidia-system-management-interface (accessed Apr. 05, 2022).

[117] "map_server - ROS Wiki." http://wiki.ros.org/map_server (accessed Apr. 05, 2022).

APPENDIX

TABLE A1. SCENARIO 1 COMPUTER RESOURCE UTILIZATION						
		# CPU Threads	**CPU %**	**Memory (MB)**	**GPU Framebuffer (MB)**	**GPU Streaming Multiprocessor %**
ROS-Unity3D	**Run 1**	169.82	134.93	688.48	2643.99	14.17
	Run 2	171.44	137.00	712.33	2681.86	15.05
	Run 3	172.43	137.90	716.15	2636.75	16.05
	Run 4	169.97	136.79	715.79	2619.14	13.36
	Run 5	171.17	137.85	741.05	2629.44	13.89
ROS-Gazebo	**Run 1**	122.00	112.38	232.13	160.00	4.81
	Run 2	122.00	111.60	232.17	160.00	5.39
	Run 3	122.00	111.24	234.58	160.00	6.31
	Run 4	122.00	111.05	234.62	160.00	5.16
	Run 5	122.00	110.77	234.69	160.00	5.57

TABLE A2: SCENARIO 1 SIMULATION SPEED				
		Variable Time Step Update Rate	**Fixed Time Step Update Rate**	**Real-Time Factor**
ROS-Unity3D	**Run 1**	35.46	100.00	1.000
	Run 2	35.31	100.00	1.000
	Run 3	35.40	100.00	1.000
	Run 4	35.57	100.00	1.000
	Run 5	34.84	100.00	1.000
ROS-Gazebo	**Run 1**	N/A	N/A	1.000
	Run 2	N/A	N/A	1.001
	Run 3	N/A	N/A	1.001
	Run 4	N/A	N/A	1.001
	Run 5	N/A	N/A	1.002

TABLE A3. SCENARIO 1 LOCALIZATION ERRORS							
		Rotational Error (degrees)			**Positional Error (meters)**		
		Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	**Run 1**	1.0686	0.2558	0.1235	0.1453	0.0741	0.0423
	Run 2	1.7234	0.2968	0.2281	0.1776	0.0396	0.0455
	Run 3	1.8776	0.9297	0.3511	0.4316	0.1438	0.0558
	Run 4	1.2541	0.5026	0.6117	0.2256	0.0844	0.0941
	Run 5	0.9529	0.2467	0.1978	0.2401	0.0634	0.0625
ROS-Gazebo	**Run 1**	2.2931	0.4831	0.6717	0.1736	0.0238	0.0493
	Run 2	3.7038	1.2989	0.9524	0.2363	0.0590	0.0361
	Run 3	0.7943	0.2842	0.1028	0.0771	0.0261	0.0352
	Run 4	2.9248	0.4648	0.3846	0.1583	0.0313	0.0217
	Run 5	1.6886	0.3697	0.1507	0.1198	0.0372	0.0148

TABLE A4. SCENARIO 1 MAPPING ACCURACY STATISTICS							
	Ground Truth	**Occupied**			**Clear**		
	Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	**Run 1**	6013	1067	18480	16972	1765	327803
	Run 2	6962	1220	17378	13594	1850	331096
	Run 3	6723	769	18068	13612	1559	331369
	Run 4	7654	1019	16887	13708	2190	330642
	Run 5	5525	837	19198	13707	1656	331177
ROS-Gazebo	**Run 1**	7256	686	17618	9610	1556	335374
	Run 2	7053	833	17674	9371	1605	335564
	Run 3	6056	744	18760	8938	1837	335765
	Run 4	6392	1272	17896	11062	1941	333537
	Run 5	6439	740	18381	9303	1858	335379

TABLE A5. SCENARIO 2 COMPUTER RESOURCE UTILIZATION						
		# CPU Threads	**CPU %**	**Memory (MB)**	**GPU Framebuffer (MB)**	**GPU Streaming Multiprocessor %**
ROS-Unity3D	**Run 1**	173.00	197.19	711.16	167.91	15.30
	Run 2	171.54	197.62	735.60	158.32	13.97
	Run 3	173.33	198.61	732.58	155.12	13.87
	Run 4	171.31	193.70	743.49	152.03	14.90
	Run 5	173.00	197.29	746.03	152.36	13.57
ROS-Gazebo	**Run 1**	121.00	82.96	1509.65	5.00	0.00
	Run 2	121.00	84.20	1509.77	5.00	0.00
	Run 3	121.00	84.91	1510.74	5.00	0.00
	Run 4	121.00	85.22	1510.81	5.00	0.00
	Run 5	121.00	84.88	1510.78	5.00	0.00

TABLE A6: SCENARIO 2 SIMULATION SPEED				
		Variable Time Step Update Rate	**Fixed Time Step Update Rate**	**Real-Time Factor**
ROS-Unity3D	**Run 1**	216.75	100.00	1.000
	Run 2	212.93	100.00	1.000
	Run 3	216.88	100.00	1.000
	Run 4	238.24	100.00	1.000
	Run 5	215.85	100.00	1.000
ROS-Gazebo	**Run 1**	N/A	N/A	0.747
	Run 2	N/A	N/A	0.737
	Run 3	N/A	N/A	0.766
	Run 4	N/A	N/A	0.740
	Run 5	N/A	N/A	0.700

TABLE A7. SCENARIO 2 LOCALIZATION ERRORS							
		Rotational Error (degrees)			**Positional Error (meters)**		
		Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	**Run 1**	60.6514	10.3362	3.8563	0.9874	0.3839	0.1153
	Run 2	55.6565	29.4993	27.7017	2.6950	0.8558	0.6133
	Run 3	7.3897	1.9196	0.4131	0.5446	0.2381	0.0419
	Run 4	50.8512	27.4681	21.5949	2.6008	0.7657	0.4298
	Run 5	7.6942	2.6800	2.8040	0.6828	0.2043	0.1367
ROS-Gazebo	**Run 1**	156.5920	16.8438	4.0790	0.9640	0.4028	0.1747
	Run 2	169.2580	57.4385	12.0655	12.1590	3.6967	3.0915
	Run 3	178.7080	49.2419	6.0967	1.4091	0.4813	0.3954
	Run 4	166.9230	39.2552	7.2438	7.0559	1.5633	0.9858
	Run 5	77.0994	14.1052	64.4798	0.8651	0.4921	0.2452

TABLE A8. SCENARIO 2 MAPPING ACCURACY STATISTICS							
	Ground Truth	**Occupied**			**Clear**		
	Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	**Run 1**	11531	5479	8550	97132	25951	223457
	Run 2	8126	7803	9631	97149	31684	217707
	Run 3	7549	5830	12181	93258	17405	235877
	Run 4	8614	9153	7793	102225	36075	208240
	Run 5	10472	4241	10847	95818	25208	225514
ROS-Gazebo	**Run 1**	9346	6328	9886	81086	35707	229747
	Run 2	7037	11279	7244	91249	24636	230655
	Run 3	8825	3990	12745	75876	26060	244604
	Run 4	3241	12562	9757	90210	22421	233909
	Run 5	10527	5278	9755	87767	26623	232150

TABLE A9. SCENARIO 3 COMPUTER RESOURCE UTILIZATION						
		# CPU Threads	**CPU %**	**Memory (MB)**	**GPU Framebuffer (MB)**	**GPU Streaming Multiprocessor %**
ROS-Unity3D	**Run 1**	174.87	164.80	743.19	2594.97	9.89
	Run 2	169.78	164.23	784.96	2635.83	10.19
	Run 3	172.12	165.08	785.95	2561.15	10.22
	Run 4	171.55	164.67	799.90	2580.73	10.10
	Run 5	171.51	165.24	812.13	2563.13	9.44
ROS-Gazebo	**Run 1**	123.00	122.66	1597.60	160.00	3.92
	Run 2	123.00	121.73	1597.61	160.00	2.88
	Run 3	123.00	121.70	1597.72	160.00	3.62
	Run 4	123.00	121.85	1597.70	160.00	2.71
	Run 5	123.00	122.74	1597.73	160.00	3.95

TABLE A10: SCENARIO 3 SIMULATION SPEED				
		Variable Time Step Update Rate	**Fixed Time Step Update Rate**	**Real-Time Factor**
ROS-Unity3D	**Run 1**	24.70	100.00	1.000
	Run 2	24.71	100.00	1.000
	Run 3	24.96	100.00	1.000
	Run 4	24.74	100.00	1.000
	Run 5	24.88	100.00	1.000
ROS-Gazebo	**Run 1**	N/A	N/A	0.584
	Run 2	N/A	N/A	0.676
	Run 3	N/A	N/A	0.665
	Run 4	N/A	N/A	0.608
	Run 5	N/A	N/A	0.597

TABLE A11. SCENARIO 3 LOCALIZATION ERRORS							
		Rotational Error (degrees)			**Positional Error (meters)**		
		Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	**Run 1**	4.0020	0.7897	1.6445	0.3910	0.0767	0.1576
	Run 2	4.4071	1.1298	1.2230	0.3097	0.0749	0.0356
	Run 3	6.7913	3.2123	1.1637	0.6409	0.3533	0.2529
	Run 4	2.5914	0.7368	1.3039	0.1455	0.0376	0.0164
	Run 5	5.6417	0.6973	0.5672	0.3889	0.0488	0.0220
ROS-Gazebo	**Run 1**	1.9675	1.0227	0.8518	0.1722	0.0526	0.0308
	Run 2	1.8482	0.8047	0.9526	0.3817	0.1244	0.1133
	Run 3	2.3809	1.2568	1.6431	0.1891	0.0350	0.0259
	Run 4	1.9816	0.6369	0.6369	0.3510	0.0215	0.0126
	Run 5	1.7496	0.4653	0.5521	0.2665	0.0312	0.0303

TABLE A12. SCENARIO 3 MAPPING ACCURACY STATISTICS							
	Ground Truth	**Occupied**			**Clear**		
	Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	**Run 1**	10638	6350	8572	119581	33352	193607
	Run 2	10635	3862	11063	93408	20744	232388
	Run 3	11677	4988	8895	99277	24764	222499
	Run 4	11941	4059	9560	104630	28505	213405
	Run 5	11820	4312	9428	103137	27975	215428
ROS-Gazebo	**Run 1**	12262	3841	9457	112631	26970	206939
	Run 2	12308	3750	9502	104720	26307	215513
	Run 3	9860	2601	13099	77110	21299	248131
	Run 4	11691	2928	10941	88103	24881	233556
	Run 5	10679	3582	11299	86839	22780	236921

TABLE A13. SCENARIO 4 COMPUTER RESOURCE UTILIZATION						
		# CPU Threads	**CPU %**	**Memory (MB)**	**GPU Framebuffer (MB)**	**GPU Streaming Multiprocessor %**
ROS-Unity3D	**Run 1**	171.58	132.95	998.75	2785.77	13.09
	Run 2	171.22	132.92	1008.37	2783.50	14.35
	Run 3	172.32	134.13	1011.19	2760.68	13.33
	Run 4	173.82	133.97	1020.09	2787.45	13.74
	Run 5	173.83	134.50	1022.46	2782.33	13.50
ROS-Gazebo	**Run 1**	122.00	109.99	802.81	722.00	4.85
	Run 2	122.00	110.52	802.81	722.00	4.13
	Run 3	122.00	110.67	802.84	722.00	4.16
	Run 4	122.00	110.70	802.82	722.00	5.75
	Run 5	122.00	110.29	802.90	722.00	3.09

TABLE A14: SCENARIO 4 SIMULATION SPEED				
		Variable Time Step Update Rate	**Fixed Time Step Update Rate**	**Real-Time Factor**
ROS-Unity3D	**Run 1**	35.69	100.00	1.000
	Run 2	34.99	100.00	1.000
	Run 3	35.10	100.00	1.000
	Run 4	34.84	100.00	1.000
	Run 5	34.72	100.00	1.000
ROS-Gazebo	**Run 1**	N/A	N/A	0.984
	Run 2	N/A	N/A	1.011
	Run 3	N/A	N/A	0.982
	Run 4	N/A	N/A	0.995
	Run 5	N/A	N/A	0.991

TABLE A15. SCENARIO 4 LOCALIZATION ERRORS							
		Rotational Error (degrees)			**Positional Error (meters)**		
		Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	**Run 1**	1.0468	0.4196	0.3052	0.2688	0.1096	0.0589
	Run 2	0.8584	0.3443	0.0442	0.2592	0.1096	0.0147
	Run 3	5.4458	1.0785	0.4234	1.5275	0.5953	0.4309
	Run 4	2.8841	0.8807	0.9348	0.4666	0.2299	0.1166
	Run 5	0.7861	0.3644	0.1968	0.2558	0.1538	0.1415
ROS-Gazebo	**Run 1**	3.3063	1.0055	1.2766	0.6286	0.2906	0.1823
	Run 2	18.2104	8.3329	10.9893	9.1675	2.0321	0.4341
	Run 3	19.0893	7.4443	3.1793	11.8169	3.3439	1.9933
	Run 4	14.1074	3.0942	1.7571	3.0259	1.1540	0.7198
	Run 5	2.3831	0.6881	0.2632	0.5692	0.2191	0.3028

TABLE A16. SCENARIO 4 MAPPING ACCURACY STATISTICS							
	Ground Truth	**Occupied**			**Clear**		
	Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	**Run 1**	1298	2490	26791	14358	2876	847187
	Run 2	1807	1210	27562	15062	1784	847575
	Run 3	713	2699	27167	17448	2987	843986
	Run 4	1593	1958	27028	13585	2847	847989
	Run 5	1709	1815	27055	14932	2104	847385
ROS-Gazebo	**Run 1**	652	1454	28473	18645	3463	842313
	Run 2	403	1014	29162	12440	3240	848741
	Run 3	1437	2387	26755	24310	8354	831757
	Run 4	190	786	29603	10687	3789	849945
	Run 5	1128	1777	27674	16674	3581	844166

TABLE A17. SCENARIO 5 COMPUTER RESOURCE UTILIZATION						
		# CPU Threads	**CPU %**	**Memory (MB)**	**GPU Framebuffer (MB)**	**GPU Streaming Multiprocessor %**
ROS-Unity3D	**Run 1**	170.64	132.94	933.37	2791.28	16.16
	Run 2	170.35	134.01	994.76	2718.40	17.00
	Run 3	170.46	134.11	996.28	2752.65	15.09
	Run 4	172.47	134.57	997.06	2813.96	15.89
	Run 5	171.61	135.14	995.85	2794.82	14.46
ROS-Gazebo	**Run 1**	122.00	107.93	750.66	682.00	3.17
	Run 2	122.00	108.64	750.97	682.00	3.66
	Run 3	122.00	108.79	750.77	682.00	3.84
	Run 4	122.00	109.31	750.65	682.00	4.05
	Run 5	122.00	108.74	750.72	682.00	3.95

TABLE A18: SCENARIO 5 SIMULATION SPEED				
		Variable Time Step Update Rate	**Fixed Time Step Update Rate**	**Real-Time Factor**
ROS-Unity3D	**Run 1**	35.29	100.00	1.000
	Run 2	32.93	100.00	1.000
	Run 3	33.71	100.00	1.000
	Run 4	33.02	100.00	1.000
	Run 5	32.96	100.00	1.000
ROS-Gazebo	**Run 1**	N/A	N/A	0.989
	Run 2	N/A	N/A	0.986
	Run 3	N/A	N/A	0.992
	Run 4	N/A	N/A	1.005
	Run 5	N/A	N/A	0.977

TABLE A19. SCENARIO 5 LOCALIZATION ERRORS							
		Rotational Error (degrees)			**Positional Error (meters)**		
		Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	**Run 1**	0.4178	0.1187	0.0442	0.0895	0.0401	0.0111
	Run 2	0.7287	0.2155	0.3122	0.2143	0.1002	0.0646
	Run 3	0.7541	0.2787	0.3026	0.1007	0.0448	0.0205
	Run 4	0.7646	0.2873	0.2948	0.1721	0.0668	0.0627
	Run 5	1.8801	0.3397	0.3795	0.2841	0.0994	0.0831
ROS-Gazebo	**Run 1**	1.6035	0.3863	0.7160	0.1885	0.0762	0.0535
	Run 2	0.9586	0.1424	0.2594	0.1900	0.0796	0.0552
	Run 3	0.8979	0.3517	0.3594	0.1157	0.0609	0.0388
	Run 4	1.5374	0.3725	0.9864	0.1530	0.0819	0.0318
	Run 5	0.5560	0.1624	0.1607	0.0335	0.0154	0.0126

TABLE A20. SCENARIO 5 MAPPING ACCURACY STATISTICS							
	Ground Truth	**Occupied**			**Clear**		
	Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	**Run 1**	780	421	1542	47742	176	93439
	Run 2	823	435	1485	48883	123	92351
	Run 3	724	466	1553	48370	257	92730
	Run 4	818	414	1511	48286	180	92891
	Run 5	977	387	1379	46953	150	94254
ROS-Gazebo	**Run 1**	626	515	1602	41667	498	99192
	Run 2	550	526	1667	47659	399	93299
	Run 3	507	533	1703	42852	485	98020
	Run 4	206	576	1961	43119	634	97604
	Run 5	510	568	1665	48596	448	92313

TABLE A21. SCENARIO 6 COMPUTER RESOURCE UTILIZATION						
		# CPU Threads	**CPU %**	**Memory (MB)**	**GPU Framebuffer (MB)**	**GPU Streaming Multiprocessor %**
ROS-Unity3D	**Run 1**	170.37	128.86	856.54	2678.47	15.64
	Run 2	171.23	129.35	872.18	2664.23	17.37
	Run 3	169.00	129.12	876.07	2654.10	15.67
	Run 4	170.79	129.56	879.36	2638.39	17.56
	Run 5	170.25	129.57	899.97	2655.63	17.16
ROS-Gazebo	**Run 1**	122.00	113.68	586.07	276.00	2.98
	Run 2	122.00	114.29	629.83	276.00	3.94
	Run 3	122.00	113.66	594.53	276.00	2.69
	Run 4	122.00	106.69	586.24	276.00	2.67
	Run 5	122.00	112.39	586.30	276.00	3.97

TABLE A22: SCENARIO 6 SIMULATION SPEED				
		Variable Time Step Update Rate	**Fixed Time Step Update Rate**	**Real-Time Factor**
ROS-Unity3D	**Run 1**	28.59	100.00	1.000
	Run 2	28.17	100.00	1.000
	Run 3	28.38	100.00	1.000
	Run 4	27.88	100.00	1.000
	Run 5	28.23	100.00	1.000
ROS-Gazebo	**Run 1**	N/A	N/A	0.993
	Run 2	N/A	N/A	0.996
	Run 3	N/A	N/A	0.994
	Run 4	N/A	N/A	0.992
	Run 5	N/A	N/A	0.997

TABLE A23. SCENARIO 6 LOCALIZATION ERRORS							
		Rotational Error (degrees)			**Positional Error (meters)**		
		Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	**Run 1**	1.7695	0.5096	0.4343	0.5390	0.1291	0.1073
	Run 2	1.1252	0.3083	0.2594	0.5653	0.1147	0.0288
	Run 3	3.6321	0.7392	0.2502	3.7326	0.1646	0.1415
	Run 4	2.3083	0.5612	0.3621	0.9905	0.1726	0.1318
	Run 5	0.8907	0.3476	0.3392	0.4116	0.0873	0.0423
ROS-Gazebo	**Run 1**	2.8682	0.7599	1.4326	3.3773	0.2972	0.3230
	Run 2	2.5653	0.9306	1.2509	0.9433	0.3088	0.4216
	Run 3	3.3844	0.6835	0.4088	1.3645	0.1541	0.1008
	Run 4	1.4594	0.1524	0.1203	0.2795	0.0699	0.0383
	Run 5	6.8167	1.9783	0.4348	4.7989	0.4335	0.0996

TABLE A24. SCENARIO 6 MAPPING ACCURACY STATISTICS							
	Ground Truth	**Occupied**			**Clear**		
	Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	**Run 1**	1002	389	201957	4566	934	718980
	Run 2	572	123	202653	4498	851	719131
	Run 3	362	140	202846	4289	954	719237
	Run 4	1210	322	201816	4111	1177	719192
	Run 5	569	225	202554	4713	626	719141
ROS-Gazebo	**Run 1**	1069	386	201893	711	678	723091
	Run 2	699	203	202446	537	450	723493
	Run 3	733	138	202477	516	515	723449
	Run 4	550	147	202651	464	231	723785
	Run 5	1158	333	201857	1056	1484	721940

TABLE A25. SCENARIO 7 COMPUTER RESOURCE UTILIZATION						
		# CPU Threads	**CPU %**	**Memory (MB)**	**GPU Framebuffer (MB)**	**GPU Streaming Multiprocessor %**
ROS-Unity3D	**Run 1**	170.73	139.74	1325.99	2824.33	20.78
	Run 2	170.99	140.43	1335.66	2787.83	19.30
	Run 3	174.00	141.08	1351.88	2792.92	20.22
	Run 4	170.00	139.69	1353.43	2781.87	19.18
	Run 5	172.39	141.35	1353.45	2806.07	19.08
ROS-Gazebo	**Run 1**	122.00	148.57	5266.50	590.00	15.32
	Run 2	122.00	149.91	5266.67	590.00	13.29
	Run 3	122.00	149.46	5267.59	590.00	16.92
	Run 4	122.00	149.50	5267.66	590.00	16.01
	Run 5	122.00	149.58	5267.57	590.00	13.24

TABLE A26: SCENARIO 7 SIMULATION SPEED				
		Variable Time Step Update Rate	**Fixed Time Step Update Rate**	**Real-Time Factor**
ROS-Unity3D	**Run 1**	27.03	100.00	1.000
	Run 2	27.43	100.00	1.000
	Run 3	26.84	100.00	1.000
	Run 4	26.63	100.00	1.000
	Run 5	26.73	100.00	1.000
ROS-Gazebo	**Run 1**	N/A	N/A	0.986
	Run 2	N/A	N/A	0.997
	Run 3	N/A	N/A	0.999
	Run 4	N/A	N/A	0.992
	Run 5	N/A	N/A	0.998

TABLE A27. SCENARIO 7 LOCALIZATION ERRORS							
		Rotational Error (degrees)			**Positional Error (meters)**		
		Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	**Run 1**	1.2430	0.6273	0.6401	0.1476	0.0894	0.0506
	Run 2	1.6359	0.8084	0.8614	0.2447	0.0895	0.2312
	Run 3	0.8693	0.5480	0.5747	0.1676	0.1038	0.0824
	Run 4	1.6363	0.6527	0.6600	0.5361	0.1347	0.0629
	Run 5	1.8264	0.6661	0.3392	0.2683	0.1322	0.0908
ROS-Gazebo	**Run 1**	2.1296	0.9822	0.2518	0.4847	0.2618	0.1897
	Run 2	0.6483	0.2930	0.3455	0.1000	0.0375	0.0441
	Run 3	0.4715	0.2774	0.2324	0.0884	0.0401	0.0610
	Run 4	0.7525	0.1634	0.0685	0.1087	0.0369	0.0233
	Run 5	0.3363	0.1675	0.1619	0.1886	0.1503	0.0830

TABLE A28. SCENARIO 7 MAPPING ACCURACY STATISTICS							
	Ground Truth	**Occupied**			**Clear**		
	Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	**Run 1**	3752	1887	24788	1244	1846	268223
	Run 2	4584	2253	23590	1645	2618	267050
	Run 3	3838	1726	24863	1476	2159	267678
	Run 4	3573	2287	24567	1306	2066	267941
	Run 5	3625	1730	25072	2008	2640	266665
ROS-Gazebo	**Run 1**	1607	1016	27804	1420	1451	268442
	Run 2	2679	1818	25930	1130	1222	268961
	Run 3	1987	1475	26965	911	1096	269306
	Run 4	2301	1541	26585	1048	1234	269031
	Run 5	1749	926	27752	1405	1591	268317

TABLE A29. SCENARIO 3 USING A REAL-TIME FACTOR OF 0.62 IN ROS-UNITY3D LOCALIZATION ERRORS						
	Rotational Error (degrees)			**Positional Error (meters)**		
	Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
Run 1	1.7220	0.5133	0.3800	0.9204	0.4660	0.0764
Run 2	2.5561	1.0510	0.4397	1.3260	0.9491	0.1410
Run 3	1.2821	0.4640	0.4640	0.2560	0.1040	0.0505
Run 4	0.6013	0.2958	0.2819	0.1249	0.0910	0.0437
Run 5	2.5944	0.8848	0.9394	0.5111	0.1706	0.0620
Average	1.7512	0.6418	0.5010	0.6277	0.3562	0.0747

TABLE A30. MODIFIED SCENARIO 7 COMPUTER RESOURCE UTILIZATION						
		# CPU Threads	**CPU %**	**Memory (MB)**	**GPU Framebuffer (MB)**	**GPU Streaming Multiprocessor %**
ROS-Unity3D	**Run 1**	172.00	142.09	1309.02	2859.79	25.00
	Run 2	171.93	141.60	1315.63	2828.75	24.22
	Run 3	172.00	140.00	1324.92	2823.86	27.04
	Run 4	172.00	141.26	1329.62	2812.18	23.78
	Run 5	170.89	142.72	1350.35	2803.32	24.58
ROS-Gazebo	**Run 1**	122.00	147.64	5180.68	576.00	11.55
	Run 2	122.00	147.77	5180.85	576.00	10.98
	Run 3	122.00	147.89	5180.73	576.00	13.62
	Run 4	122.00	147.95	5180.82	576.00	13.79
	Run 5	122.00	147.87	5180.81	576.00	13.08

TABLE A31: MODIFIED SCENARIO 7 SIMULATION SPEED				
		Variable Time Step Update Rate	**Fixed Time Step Update Rate**	**Real-Time Factor**
ROS-Unity3D	**Run 1**	31.48	100.00	1.000
	Run 2	31.07	100.00	1.000
	Run 3	30.79	100.00	1.000
	Run 4	30.10	100.00	1.000
	Run 5	30.81	100.00	1.000
ROS-Gazebo	**Run 1**	N/A	N/A	0.995
	Run 2	N/A	N/A	0.987
	Run 3	N/A	N/A	1.004
	Run 4	N/A	N/A	1.005
	Run 5	N/A	N/A	0.988

TABLE A32. MODIFIED SCENARIO 7 LOCALIZATION ERRORS							
		Rotational Error (degrees)			**Positional Error (meters)**		
		Maximum	**Mean**	**Median**	**Maximum**	**Mean**	**Median**
ROS-Unity3D	**Run 1**	1.5413	0.4905	0.5028	0.5885	0.2802	0.3358
	Run 2	1.4478	0.7082	0.9100	0.6030	0.1857	0.4989
	Run 3	2.0607	0.5954	0.4015	0.6683	0.3422	0.5809
	Run 4	1.4871	0.7662	0.9500	0.3580	0.1967	0.2857
	Run 5	5.9469	1.2714	0.5290	1.8312	1.1506	0.6562
ROS-Gazebo	**Run 1**	4.1952	1.6347	2.1428	1.4013	0.5025	0.6812
	Run 2	5.8949	3.3781	3.4587	0.9593	0.3868	0.3625
	Run 3	5.2378	3.5051	4.1799	2.5993	0.9817	0.8741
	Run 4	8.7606	5.1202	6.6833	2.1555	1.3032	0.5022
	Run 5	4.3714	3.0173	3.0035	1.2501	0.5725	0.4149

TABLE A33. MODIFIED SCENARIO 7 MAPPING ACCURACY STATISTICS							
	Ground Truth	**Occupied**			**Clear**		
	Classification	**Occupied**	**Clear**	**Unknown**	**Clear**	**Occupied**	**Unknown**
ROS-Unity3D	**Run 1**	629	23	24209	1	190	336148
	Run 2	501	87	24273	31	342	335966
	Run 3	720	156	23985	42	470	335827
	Run 4	572	65	24224	42	321	335976
	Run 5	714	79	24068	198	1108	335033
ROS-Gazebo	**Run 1**	236	42	24583	219	487	335633
	Run 2	236	38	24587	5	115	336219
	Run 3	329	21	24511	123	193	336023
	Run 4	295	79	24487	127	723	335489
	Run 5	268	44	24549	7	110	336222