AdaSim: A Modular, Game Engine-Agnostic Framework for 3D Drone Path Planning and Simulation

Cole Kingery[§]
Purdue University
West Lafayette, Indiana
ckinger@purdue.edu

Anugunj Naman[§]
Purdue University
West Lafayette, USA
anaman@purdue.edu

Will Stonebridge[§]
Purdue University
West Lafayette, Indiana
jwstoneb@purdue.edu

***Abstract*—Motion planning is a fundamental yet computationally intensive aspect of robotics, especially within the domain of aerial robotics where robust testing frameworks are essential for validating algorithms. Existing simulators like AirSim and Gazebo offer high-fidelity environments but often suffer from platform dependencies, lack modularity, or prioritize real-time simulation over preplanning capabilities. In this paper, we introduce *AdaSim*, a modular and game engine-agnostic framework for 3D drone path planning and simulation. *AdaSim* decouples the computational backend from the visualization environment, enabling seamless integration with any game engine. At its core is a Python library that facilitates the development and testing of preplanning algorithms within customizable 3D environments. We demonstrate *AdaSim*'s capabilities in both simple and complex scenarios, validating its robustness in converting Unity scenes into navigable grids and reintegrating computed waypoints for simulation. With its lightweight, flexible, and extensible design, *AdaSim* bridges the gap between algorithm development and simulation, addressing the growing need for accessible and versatile tools in robotics research.**

***Index Terms*—simulation, motion planning, 3D environments, drones**

I. Introduction

Motion planning is a complex and computationally expensive task, requiring significant effort in algorithm development and real-world feasibility testing [1]. Real-world testing often necessitates costly equipment and extensive sensor suites, creating barriers for smaller teams with limited resources. To mitigate these challenges, many researchers rely on simulations to validate initial findings and reduce capital expenses. Simulators are particularly effective for ground-based robotics, providing a controlled environment for testing algorithms and motion planning strategies [2, 3].

However, in the domain of aerial robotics, the availability of accessible and flexible simulation tools is limited. The increasing use of drones in diverse applications, such as logistics, inspection, and public safety, underscores the urgent need for effective testing frameworks for new motion planning techniques [4]. Existing simulators like AirSim [7] and Gazebo [5] provide high-fidelity simulation environments, but they have notable limitations. AirSim, while powerful, is tightly coupled with Unreal Engine, making it less flexible for users preferring other visualization platforms. Similarly, Gazebo is primarily Linux-based, has steep learning curves, and can be computationally intensive when simulating large or complex environments. Both simulators are often more suited for real-time applications, with less focus on modular preplanning algorithm development and evaluation.

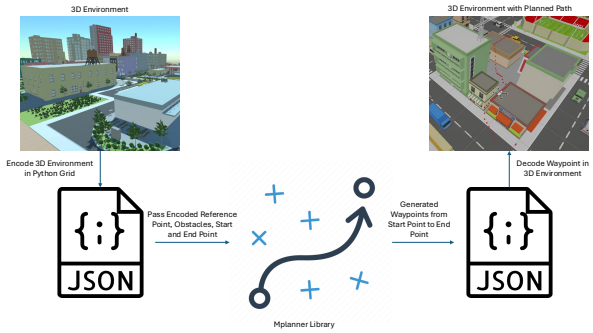

Fig. 1. Modular Workflow of AdaSim Framework .The figure depicts AdaSim's process: encoding 3D environments into grids, generating waypoints using Mplanner, and decoding them back into 3D visualizations for flexible and engine-agnostic motion planning.

Furthermore, there is a pressing need for simulation frameworks that offer flexibility in environmental design. For instance, drones may be deployed in highly variable settings, such as cluttered warehouses or open outdoor spaces. Each setting presents unique obstacles and conditions, requiring significant setup time for testing. In dynamic environments like warehouses, operational changes necessitate proactive testing of motion planning algorithms to maintain uninterrupted operations. Moreover, physical testing in such environments may be constrained by privacy laws and safety concerns for individuals in the vicinity [6]. These constraints call for a simulator that facilitates flexible, repeatable, and safe testing scenarios.

Our work *AdaSim* addresses these gaps by proposing a novel

[§]Equal contribution. Author names are listed alphabetically.

simulation framework for aerial robotics. Central to our framework is a Python-based library that decouples path planning from the simulation environment, enabling users to integrate it with game engines like Unity or Unreal. This approach allows users to create and test motion planning algorithms within customizable 3D environments. Unlike traditional simulators, our framework focuses on preplanning algorithms such as A* and Dijkstra, while maintaining a modular architecture that supports the addition of new algorithms. The framework supports vertical navigation and collision detection in all directions, extending the traditional 2D coordinate plane into a fully realized 3D space. Users are empowered to design scenes tailored to specific testing requirements, such as warehouses or outdoor environments, while visualizing the simulation output for effective evaluation.

Our simulator bridges the gap between scene creation, path planning, and simulation output, providing an all-in-one solution for aerial robotics. By enabling iterative design processes, the framework reduces the barrier to entry for smaller teams and fosters innovation in motion planning algorithms. With over 396,746 commercial drones registered in the United States as of October 2024 [4], the demand for accessible and cohesive drone simulation tools is more evident than ever. Our work contributes to meeting this demand by delivering a lightweight, flexible, and modular simulation solution tailored for aerial robotics.

The key contributions of this work are summarized as follows:

1) **Decoupled, Game Engine-Agnostic Architecture:** A modular Python-based library that integrates seamlessly with any game engine, such as Unity, allowing flexible scene creation and algorithm testing.
2) **Support for Preplanning Algorithms:** Focused support for offline path planning algorithms, including A* and Dijkstra, with a modular design enabling the easy addition of new algorithms.
3) **3D Path Planning Capability:** Extends the traditional 2D coordinate plane into 3D, supporting vertical navigation and full collision detection for aerial robotics.
4) **Rapid and Iterative Design Process:** Provides a streamlined workflow for scene creation, algorithm integration, and visual simulation output, fostering accessibility for smaller teams and researchers.

II. Related Work

Simulators have long been a cornerstone of robotics research, providing controlled environments for testing algorithms and models. For aerial robotics, two of the most prominent simulators are *AirSim* [7] and *Gazebo* [5]. *AirSim*, developed by Microsoft, offers a high-fidelity physics-based simulation environment leveraging Unreal Engine. It supports various sensors, such as RGBD cameras, LiDAR, and GPS, making it well-suited for testing perception, planning, and control algorithms. While its visually rich and physics-accurate simulations have driven significant advancements in autonomous motion research, *AirSim* tightly couples its architecture with Unreal Engine. This restricts flexibility for teams using other game engines and imposes high computational demands. Furthermore, its focus on real-time simulation and integration with machine learning frameworks leaves gaps in support for iterative preplanning and pathfinding.

Similarly, *Gazebo*, a widely adopted open-source robotics simulator, excels in providing physics-based 3D environments for validating robotic systems. Its use of the Open Dynamics Engine ([8]) enables accurate simulation of rigid body dynamics, supporting the integration of various sensors for realistic results. While *Gazebo* is flexible and extensible, it presents challenges for aerial robotics due to its steep learning curve, reliance on Linux-based systems, and performance limitations in complex or dynamic environments. Both *AirSim* and *Gazebo* focus on full-system robotics simulation, often overlooking the modularity and standalone usability of motion planning tools for benchmarking and testing.

Our work, *AdaSim*, introduced in Section I, presents a modular and lightweight drone simulation framework designed to decouple path planning from the simulation environment. By facilitating seamless integration with various 3D game engines, such as Unity, *AdaSim* offers a versatile platform for evaluating preplanning algorithms within highly customizable 3D environments. The details of our simulation framework are elaborated upon in the following section with a general overview of workflow shown in Figure-1.

III. AdaSim

In this section, we detail the design and workflow of *AdaSim*, beginning with the Unity-based simulation environment and its integration with our motion planning pipeline, followed by a discussion of the modular Python library that powers its core functionalities.

A. Unity Integration for Scene Creation and Visualization

Unity serves as the primary simulation environment for *AdaSim*, leveraging its robust physics engine and visualization capabilities to create customizable 3D scenarios. By packaging *AdaSim* as a Unity asset, we enable seamless integration into any Unity project, allowing users to utilize pre-existing or custom-designed scenes. This design provides flexibility in simulating diverse environments, from cluttered warehouses to open outdoor spaces.

The workflow begins with the definition of a *Scene* in Unity, comprising various *GameObjects* such as obstacles, floors, and drones. Each *GameObject* has specific attributes, including:

- **Colliders**: Used to detect obstacles and simulate interactions within the environment.
- **Rigid Bodies**: Provide physical properties for dynamic simulation (e.g., gravity or collision response).
- **Renderers**: Define the visual appearance of objects within the scene.

Figure 2 illustrates a demo scene created in Unity to test the *AdaSim* asset. This scene consists of various GameObjects such as rectangles, spheres, and a floor, which represent

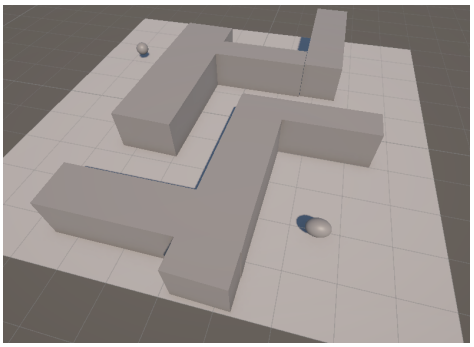

Fig. 2. A demo scene created in Unity to test the *AdaSim* asset. The rectangles, floor, and spheres represent GameObjects with attributes such as colliders, rigid bodies, and renderers.

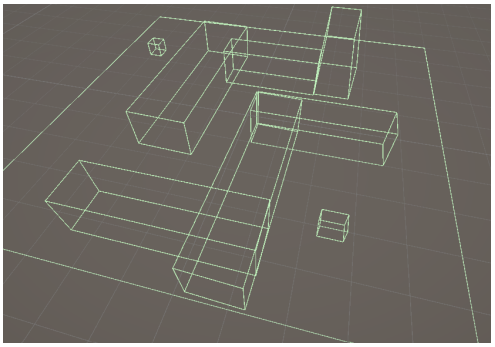

Fig. 3. The collider view of the demo scene shown in Figure 2. Colliders are used to define physical boundaries and simulate interactions within the environment.

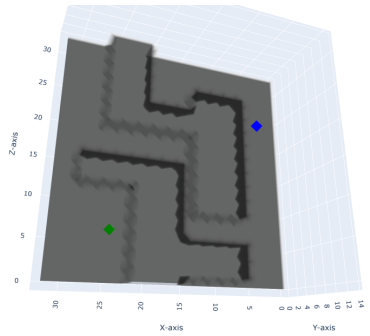

Fig. 4. The demo scene from Figure 2 converted into a discretized Python grid representation.

environmental elements. The colliders associated with these objects, shown in Figure 3, enable the system to define obstacles and interactions accurately.

Once the scene is created, *AdaSim* processes the environment by identifying all *GameObjects* with colliders, treating them as obstacles. The spatial dimensions and positions of these obstacles are exported into a JSON file, referred to as *Scene Data*. This JSON format provides a structured representation of the environment, serving as input for the motion planning algorithms in the Python library.

B. Python Library for Modular Motion Planning

The Python library at the core of *AdaSim* is a robust, modular framework designed to facilitate the development, testing, and visualization of motion planning algorithms in three-dimensional environments. Its architecture emphasizes extensibility, allowing users to implement and compare a variety of pathfinding techniques with minimal overhead. The library operates independently of the simulation engine, providing a flexible computational backbone that seamlessly integrates with Unity or any other visualization tool.

1) Design and Architecture: The library is structured around a 3D grid representation, which discretizes the environment into cells for efficient pathfinding. Each cell in the grid is represented by a `GridNode`, a class that encapsulates spatial coordinates, walkability, weight, and connections to neighboring nodes. This abstraction allows the grid to model real-world scenarios with varying obstacle layouts, terrain complexities, and movement constraints. Figure 4 demonstrates how the demo scene from Unity is discretized into a grid representation within the library.

The `Grid` class manages these nodes and offers methods to calculate neighbors, validate walkability, and handle boundary conditions. For multi-environment simulations, a `World` class can coordinate multiple interconnected grids, enabling transitions between different spatial regions. These abstractions form the foundation upon which algorithms operate, ensuring that the core components of the library are decoupled from algorithm-specific implementations.

At the heart of the pathfinding process is the `Planner` base class, which serves as a template for implementing custom algorithms. The class provides essential utilities such as neighbor evaluation, cost calculations, and path reconstruction. Users can extend this class to define specific behaviors for node expansion, cost estimation, and search strategies. For example, `Dijkstra3D`, a subclass of `Planner`, implements Dijkstra's algorithm by overriding the node evaluation methods to prioritize exhaustive cost minimization without heuristics.

2) Features and Capabilities: The library offers a range of features that make it a versatile tool for motion planning. Its grid-based architecture supports not only static environments but also dynamic scenarios where obstacles can change position or size. Obstacles and weights are defined programmatically or imported from JSON files, which encode the spatial layout of the environment. The JSON format ensures compatibility with external tools like game engines, enabling seamless integration into diverse workflows. Once a path is computed, the library outputs the results as a series of waypoints, also in JSON format. This output can then be re-integrated into the simulation environment, allowing for visualization or further processing, such as guiding a simulated drone or robot along the calculated path. This bidirectional JSON interface ensures efficient interaction between the simulation and computational layers.

The library's modular design enables seamless integration of new algorithms. Developers can extend the functionality by subclassing the `Planner` class and overriding specific methods to define the algorithm's node processing logic. This architecture abstracts away low-level complexities such as neighbor retrieval, path reconstruction, and grid management, allowing users to concentrate exclusively on implementing the core logic of their algorithms.

Another key feature is the library's efficient handling of computational overhead. It employs caching mechanisms for distance calculations and uses priority queues to manage the open list of nodes during the search process. These optimizations enable the library to scale effectively for larger environments and more complex scenarios.

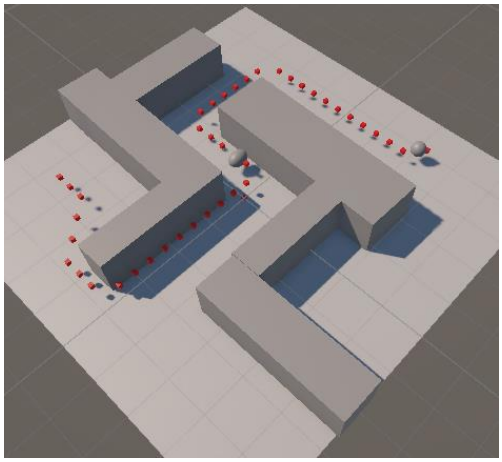

Fig. 5. Path generated by Dijkstra's algorithm in the simple maze scene. The algorithm calculates a valid path around rectangular obstacles.

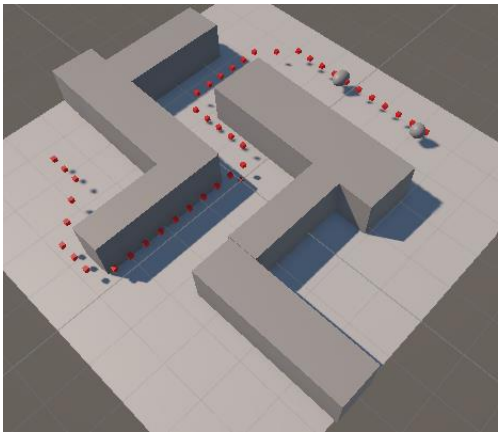

Fig. 6. Path generated by A* algorithm in the simple maze scene. A* computes a path similar to Dijkstra's while optimizing for faster computation.

3) Visualization and Metrics: To complement its computational capabilities, the library includes an integrated visualization module powered by Plotly. Users can generate interactive 3D visualizations of the environment, highlighting obstacles, weights, and computed paths. The visualization not only provides intuitive feedback on algorithm performance but also helps debug pathfinding issues by visually identifying problem areas in the environment.

In addition to visualization, the library provides quantitative metrics such as path length, computation time, and iteration counts. These metrics are critical for benchmarking algorithms and assessing their efficiency in different scenarios.

By adopting a decoupled architecture, *AdaSim* eliminates unnecessary complexities, offering a streamlined setup process that requires only a game engine and Python. The Unity simulation asset integrates seamlessly into existing projects, while the modular Python library allows users to develop and refine algorithms independently of the visualization environment. This design ensures flexibility and facilitates rapid experimentation, enabling researchers to test motion planning algorithms across diverse scenarios without being constrained by a specific platform or environment. The complete workflow of *AdaSim* is summarized in Algorithm 1.

IV. Illustrative Use Cases

To demonstrate the capabilities and effectiveness of *AdaSim*, we conducted experiments in two distinct environments, highlighting the system's adaptability and robust decoupled workflow. The first environment is a simple maze scene (Figure 2), designed to evaluate basic 3D path planning and navigation. The second environment is a complex, imported cityscape (Figure 7), featuring elevated structures and dense obstacles to test more intricate scenarios.

In both environments, the simulation pipeline was evaluated for its ability to accurately convert Unity scenes into structured JSON data, process the data using Python's modular motion planning library, and reintegrate the computed waypoints into

Algorithm 1 *AdaSim* Workflow

Require: Unity scene with *GameObjects*, Python motion planning library

Ensure: Computed path as waypoints visualized in the Unity environment

Step 1: Unity Scene Setup
Create or import a scene in Unity.
Add *GameObjects* representing obstacles, terrain, or drones.
Assign attributes like Colliders, Rigid Bodies and Renderers as described in Section III-A

Step 2: Environment Data Extraction
Identify all *GameObjects* with colliders in the Unity scene.
Export spatial data (positions, dimensions) into a JSON file (*Scene Data*).

Step 3: Path Computation in Python
Load *Scene Data* into the Python library.
Initialize a 3D grid based on scene dimensions and resolution.
Mark obstacles and set weights in the grid using extracted data.
Select or implement a motion planning algorithm:
    Use predefined algorithms like A* or Dijkstra.
    Or extend the `Planner` class for custom algorithms.
Compute the optimal path as a series of waypoints.
Export the computed waypoints to a JSON file.

Step 4: Visualization and Simulation in Unity
Import computed waypoints into Unity.
Visualize the path in the scene as markers or lines.
Simulate drone movement along the computed path.
Display performance metrics (path length, computation time).

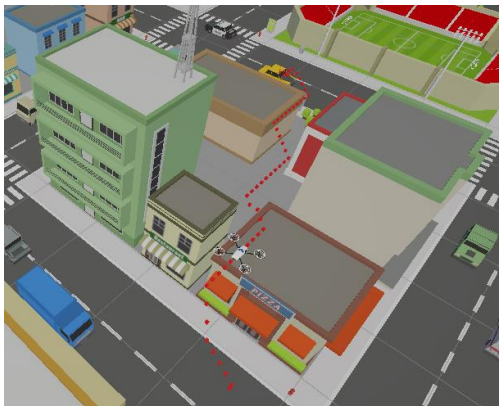

Fig. 7. Path generated by Dijkstra's algorithm in the city scene. The algorithm computes a 3D path that minimizes distance by navigating over buildings and obstacles.

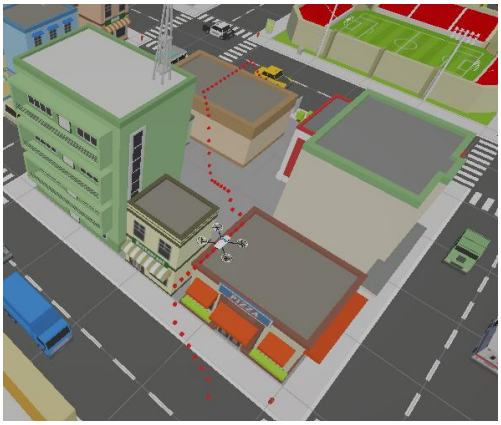

Fig. 8. Path generated by A* algorithm in the city scene. A* produces a valid 3D path with slight deviations, optimized for faster computation.

Unity for visualization and execution. We implemented two algorithms, Dijkstra and A*, as subclasses of the `Planner` class to validate the correctness and adaptability of the system. Key metrics such as path validity and computation time were assessed to demonstrate the system's performance.

A. Simple Scene: Validating Path Planning in a Controlled Environment

The first environment consists of a maze-like layout constructed using rectangular cuboids (Figure 2). This scene allows the drone to navigate through a controlled setup, avoiding obstacles while reaching a target point.

Using Dijkstra's algorithm (Figure 5), the system accurately computed a collision-free 3D path. The JSON export of obstacle positions was successfully processed into a Python grid world, which was then used to generate waypoints. These waypoints were seamlessly imported back into Unity, where the drone followed the computed path without any discrepancies. The simulation completed the planning process in 1970ms, demonstrating efficient handling of the scene.

Similarly, the A* algorithm (Figure 6) achieved identical results in terms of path validity, with a slightly faster computation time of 1902ms. This consistency across two algorithms confirms that the decoupled pipeline correctly integrates scene data, motion planning, and Unity visualization, enabling reliable testing of different pathfinding techniques.

B. City Scene: Handling Complex 3D Environments

The second environment is a cityscape imported from a Unity asset package [9], featuring tall buildings and densely packed obstacles. This scenario tested the system's ability to handle more complex environments and generate valid 3D paths over varied terrain.

Dijkstra's algorithm (Figure 7) computed a valid path, requiring the drone to fly over obstacles to reach its target. The decoupled pipeline effectively processed the large-scale scene, converting it into a navigable grid world and back into waypoints for Unity to consume. While Dijkstra's approach produced a path with minimal deviations, the computation time was higher at 24814ms due to the exhaustive nature of the algorithm.

The A* algorithm (Figure 8) also produced a valid path, with a slightly longer traveled distance but a significantly faster computation time of 13223ms. This disparity reflects the inherent trade-offs between the algorithms and highlights *AdaSim*'s ability to support comparative evaluations.

In both scenarios, the decoupled pipeline of *AdaSim* demonstrated its reliability and flexibility. Unity scenes were successfully converted into Python grid representations, allowing the motion planning library to generate accurate 3D paths. These paths were then seamlessly reintegrated into Unity for visualization and execution, validating the end-to-end workflow. The system handled varying levels of complexity without failure, showcasing its robustness across different scenarios.

By decoupling the simulation environment from the computational backend, *AdaSim* enables rapid iteration and testing of algorithms in diverse scenarios. Its ability to dynamically integrate changes in obstacles or layouts ensures that it remains scalable and adaptable for both research and real-world applications in robotics.

V. Discussion

AdaSim represents a significant step forward in modular, decoupled simulation for motion planning, yet several avenues for enhancement remain, each offering exciting potential for growth and expanded applicability.

A. Improving Obstacle Representation

Currently, *AdaSim* simplifies obstacles by using bounding rectangles or cuboids, as shown in Figure 9. While this approach ensures computational efficiency, it does not fully leverage the detailed geometry provided by Unity's mesh colliders (Figure 10). Incorporating voxelization techniques to convert mesh colliders into the grid representation would improve the fidelity of obstacle modeling. This enhancement

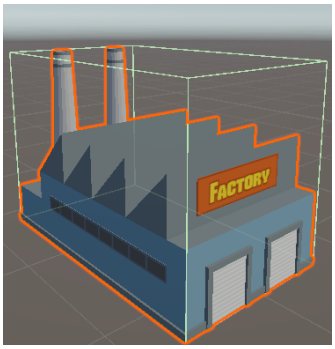

Fig. 9. A Unity Box Collider, simple but not descriptive

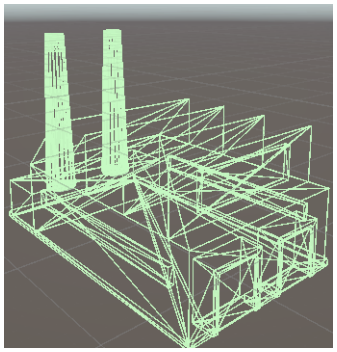

Fig. 10. A Unity Mesh Collider, complex but completely descriptive

would allow for more realistic simulations, particularly in highly detailed or complex scenes.

B. Expanding Applicability Beyond Aerial Robots

Though designed for aerial robotics, the framework's modularity opens pathways for adaptation to ground-based systems. By integrating terrain analysis and preprocessing to filter navigable surfaces, *AdaSim* could simulate ground robots navigating flat or sloped surfaces. This capability would broaden the framework's utility, making it applicable to scenarios such as autonomous vehicles, robotic arms, or warehouse robots.

C. Real-Time and Dynamic Scenarios

While the current preplanning approach demonstrates the effectiveness of *AdaSim* in static environments, extending it to real-time dynamic planning offers tremendous potential. The system's pipeline is well-suited for adaptation to continuous path planning, where real-time sensor inputs or vision data could dynamically update paths. This evolution would position *AdaSim* as a valuable tool for applications such as real-time navigation in changing environments or autonomous drones operating in unpredictable spaces.

VI. Conclusion

AdaSim introduces a decoupled, modular framework for motion planning and simulation, integrating Unity's 3D visualization capabilities with a versatile Python-based computational backend. By converting Unity scenes into navigable grids and reintegrating computed paths seamlessly, *AdaSim* provides a unified environment for designing, testing, and evaluating motion planning algorithms. Through its experiments in both simple and complex environments, the framework has demonstrated robustness and flexibility, supporting diverse pathfinding techniques with ease. With its extensible architecture and potential for enhancements in obstacle modeling, real-time planning, and scalability, *AdaSim* lays a solid foundation for advancing simulation and algorithmic research in robotics and autonomous systems.

References

[1] Z.M. Bi et al. "The state of the art of testing standards for integrated robotic systems". In: *Robotics and Computer-Integrated Manufacturing* 63 (2020), p. 101893. ISSN: 0736-5845.

[2] Jack Collins et al. "A Review of Physics Simulators for Robotic Applications". In: *IEEE Access* 9 (2021), pp. 51416–51431.

[3] Alexey Dosovitskiy et al. *CARLA: An Open Urban Driving Simulator*. 2017. arXiv: 1711.03938.

[4] FAA. *Drones by the Numbers*. URL: https://www.faa.gov/node/54496.

[5] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vol. 3. 2004, pp. 2149–2154.

[6] Bhawesh Sah, Rohit Gupta, and Dana Bani-Hani. "Analysis of barriers to implement drone logistics". In: *International Journal of Logistics Research and Applications* 24.6 (2021), pp. 531–550.

[7] Shital Shah et al. *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*. 2017. arXiv: 1705.05065.

[8] Russell Smith. *Open Dynamics Engine*. 2008. URL: http://www.ode.org/.

[9] VenCreations. *SimplePoly City - Low Poly Assets*. URL: https://assetstore.unity.com/packages/3d/environments/simplepoly-city-low-poly-assets-58899.