

GAMMS: Graph based Adversarial Multiagent Modeling Simulator

Rohan Patil*

UC San Diego

San Diego, United States

rpatil@ucsd.edu

Andre Dion†

University of Southern California

Los Angeles, United States

adion@usc.edu

Jai Malegaonkar*

UC San Diego

San Diego, United States

jmalegaonkar@ucsd.edu

Gaurav S. Sukhatme

University of Southern California

Los Angeles, United States

gaurav@usc.edu

Xiao Jiang†

University of Southern California

Los Angeles, United States

xjiang26@usc.edu

Henrik I. Christensen

UC San Diego

San Diego, United States

hchristensen@ucsd.edu

ABSTRACT

As intelligent systems and multi-agent coordination become increasingly central to real-world applications, there is a growing need for simulation tools that are both scalable and accessible. Existing high-fidelity simulators, while powerful, are often computationally expensive and ill-suited for rapid prototyping or large-scale agent deployments. We present GAMMS (Graph based Adversarial Multiagent Modeling Simulator), a lightweight yet extensible simulation framework designed to support fast development and evaluation of agent behavior in environments that can be represented as graphs. GAMMS emphasizes five core objectives: scalability, ease of use, integration-first architecture, fast visualization feedback, and real-world grounding. It enables efficient simulation of complex domains such as urban road networks and communication systems, supports integration with external tools (e.g., machine learning libraries, planning solvers), and provides built-in visualization with minimal configuration. GAMMS is agnostic to policy type, supporting heuristic, optimization-based, and learning-based agents, including those using large language models. By lowering the barrier to entry for researchers and enabling high-performance simulations on standard hardware, GAMMS facilitates experimentation and innovation in multi-agent systems, autonomous planning, and adversarial modeling. The framework is open-source and available at <https://github.com/GAMMSim/GAMMS/>

KEYWORDS

Robot Planning, Simulation, Multi-agent

1 INTRODUCTION

Robot automation has moved beyond simple automation in structured industrial settings [4] to serving coffee to humans [27], guide around an exhibit [11] as well as helping to clean homes [9]. A major part of these intelligent robot systems to navigate and understand the complexities of our world is thanks to the sophisticated planning algorithms that convert high level instructions into low level movement logic for the robot to execute. However, it becomes increasingly difficult to test the behavior of the planning algorithm in various situations in the real world due to physical constraints. In the prototype phase, usually the goals are to test a particular

part of the system. However, the planning system is dependent on the proper functioning of the control logic which further depends on proper functioning of the various sensors and mechanical components. Simulation provides an efficient way to test the basic behavior of the planning system. Obviously, simulation is not a perfect solution but it provides quick initial feedback to iterate during prototyping phase. This is especially helpful in the current approaches that utilize deep neural networks. Due to the black box nature of these approaches, there are multiple scenarios where it is necessary to test the behavior of the system and validate it. In addition, heuristics are an important engineering tool but need to be tested to get an idea of its efficacy.

In an ideal world, the simulation exactly simulates every part of the robot system as well as the environment. However, the compute cost increases based on the detail to which everything is simulated. There is a direct relation between the fidelity of the simulation and its scalability with compute. The scalability issue is further attenuated when dealing with systems that have multiple robots. The scalability problem highlights the limit of using high fidelity for testing multi-robot systems in prototyping phase as well as in scenarios where the motive is to observe a multi-robot system's overall high level plan or strategy.

In contrast to end-to-end systems, the recent development of agentic systems have shown that it is possible to develop components independently and finally put them together to create a highly intelligent systems. The upcoming field of embodied artificial intelligence is relying on large language models to do a high level planning for tasks while using a mix of classical control or reinforcement learning based control to create the low level instructions. In addition, it has been observed that LLMs have many limitations when generating low-level (control) instructions accurately [13]. This separation of control and high-level planning creates the opportunity to develop a simulator directed towards testing the behavior using abstract data that have some grounding with the real world but need not be exact [15, 22, 31].

In this paper, we present GAMMS (Graph based Adversarial Multiagent Modeling Simulator), a simulator targeted towards addressing the issue of testing and developing strategies for systems that can be abstracted as a graph structure. Graph-based abstraction is ideal for large real-world environments because it efficiently models complex relationships and dependencies between entities,

*Both authors contributed equally to the paper

†Both authors contributed equally to the paper

enabling scalable analysis and decision-making. It allows for flexible navigation, optimization, and problem-solving across dynamic and interconnected systems. Road networks are a prime example of real-world systems that can be represented as a graph. In the case of autonomous cars or robots in general, the road connectivity encodes the possible routes the agent can take to travel from one point to another. However, it should be noted that such graphs can be large (1000+ nodes). In the city of Manhattan itself, there are over 2800 traffic signals [25]. Another reason for developing GAMMS is to have a structure that helps make it possible to simulate at scale for such large graphs.



Figure 1: Central Park, NYC OSM on the left. GAMMS graph for Central Park on the right.

2 GAMMS OBJECTIVES

The design and development of GAMMS is influenced by five core objectives that address fundamental challenges/issues in multi-agent simulation research: 1. Scalability, 2. Ease of Use, 3. Integration First Arch, 4. Fast Visualization Feedback, 5. Real World Grounding. Through these objectives we intend to democratize access to large-scale multi-agent modeling while maintaining the flexibility needed for diverse research applications.

Scalable Graph-Based Simulation. GAMMS prioritizes computational efficiency to enable large scale multi-agent scenarios that can run on standard consumer hardware rather than requiring specialized computing resources. The graph based environmental representation provides a natural abstraction for many real-world domains where agents operate on network structures such as urban road systems, communication networks, and social interaction graphs. The simulator implements a simple interface which does not enforce any particular semantics, providing flexibility for researchers working with different paradigms. The ability to scale simulation to large graphs (1000+ nodes) allows researchers to simulate multi-agent systems on a scale similar to the real world while maintaining reasonable computational performance. This addresses a key barrier that often limits multi-agent research to small-scale toy problems.

Ease of Use. A critical objective of GAMMS is to reduce the barrier to entry for researchers who want to experiment with multi-agent scenarios but lack extensive software engineering experience. The simulator is designed to require minimal training and setup time, allowing domain experts to quickly translate their conceptual models into running simulations. We recognize that many

valuable insights in multi-agent research come from researchers whose primary expertise lies in application domains (economics, planning, game theory, etc.) rather than computer science, and that complex simulation frameworks often prevent these researchers from effectively testing their hypotheses.

Integration-First Architecture. The framework is explicitly designed with an *integration-first* philosophy, making it easier to incorporate external tools, libraries, and existing code rather than requiring users to re-implement functionality within a closed ecosystem. This architectural decision acknowledges that effective multi-agent research often requires combining insights and tools from multiple domains. For example, deep/machine/reinforcement learning libraries for agent intelligence, optimization solvers for planning problems, statistical packages for analysis, or domain-specific simulators for particular environmental effects. By prioritizing integration capabilities, GAMMS enables researchers to leverage existing tools while focusing their efforts on the novel aspects of their research.

Fast Visualization Feedback. Building on top of the previous two objectives, GAMMS also emphasizes minimal code requirements for creating meaningful visualizations of simulation results. Rather than requiring users to implement complex rendering pipelines or learn specialized graphics libraries, the framework provides built-in visualization capabilities that can display agent behaviors, environmental states, and simulation dynamics with none to minimal additional code. This objective stems from the recognition that visualization is crucial for both debugging multi-agent systems and communicating results.

Real-World Grounding. GAMMS is designed to facilitate the modeling of scenarios that connect to real-world data and environments, rather than focusing exclusively on abstract or toy problems. The framework includes native support for processing real-world geographical data (such as OpenStreetMap [26]) and converting it into graph representations suitable for multi-agent simulation. This objective reflects the understanding that many of the most important questions in multi-agent systems research ultimately concern how agents behave in realistic environments with real constraints, and that simulators should facilitate this connection to reality rather than abstracting away from it.

These objectives collectively position GAMMS as a practical tool for advancing multi-agent systems research by lowering barriers to experimentation while maintaining the flexibility and realism needed for meaningful scientific investigation.

3 RELATED WORKS

Computer simulation has grown hand-in-hand with the rapid growth in computing power. In the context of multi-agent simulators, multiple simulators to test simple automations like in Conway’s Game of Life have been made. As such, there is a large body of work here to consider, and hence we limit our discussion primarily to simulators that have been used in the context of developing multi-robot systems. We divide the simulators into two groups: high-fidelity and low-fidelity simulators. High-fidelity simulators aim to closely replicate real-world conditions, incorporating detailed features such as realistic physics models, non-idealized sensor data, and real-time

feedback mechanisms. A key advantage of high-fidelity environments is that algorithms validated within these simulators tend to perform reliably on physical robots. In contrast, low-fidelity simulators offer simplified abstractions of reality, emphasizing lightweight codebases that facilitate rapid prototyping and enable extensive parallel experimentation. These environments are geared towards high-level algorithm development, allowing researchers to quickly iterate on new ideas before moving to more complex testing phases. This distinction is particularly relevant in the context of GAMMS and related frameworks, where balancing fidelity with development efficiency is crucial.

High Fidelity Simulators. High-fidelity simulation frameworks serve as sophisticated tools for constructing realistic simulations. They provide modular architectures that allow users to compose physics, sensors, environments, and execution pipelines flexibly. For example, Gazebo and Isaac Sim [18] integrate tightly with ROS2 [20] through a ROS Bridge, enabling simulation components to publish and subscribe to standard ROS topics such as joint states, camera images, and transforms in real time. They support a broad range of realistic sensors sourced from industry manufacturers (e.g., Intel, Orbbec, SICK) and allow users to implement custom sensors [7]. Isaac Sim leverages GPU-accelerated ray-traced rendering and physics simulation to deliver highly realistic environments and sensor data. Such frameworks support importing robot models via URDF or USD formats, scene composition with complex objects, and precise physical interactions. However, using these frameworks as tools requires significant setup effort, advanced hardware (e.g., RTX GPUs for Isaac Sim), and expertise to tune simulation parameters. Iteration can be slow due to high computational demands and complexity, especially when scaling to large numbers of agents or structured multi-team scenarios. Consequently, while high-fidelity tools excel at final validation, robot-in-the-loop testing, and sim-to-real transfer, their overhead can be prohibitive during early-stage algorithm development.

Low Fidelity Simulators. Low-fidelity simulators prioritize simplicity to enable rapid iteration and better scalability during early development stages. Designed primarily for prototyping and early algorithm testing, they rely on idealized sensors, simplified physics, and minimal computational overhead, allowing developers to focus on high-level logic and strategy rather than intricate low-level details. However, this simplicity often results in behaviors that diverge significantly from real-world dynamics, limiting their direct applicability to production environments. Common agent-based modeling frameworks like Repast [23], Mason [19], Agents.jl [8], and GAMA [3] offer greater flexibility in spatial, network, and agent interactions and can handle larger populations, but they are not Python-native and typically it becomes harder to integrate with other existing software. NetLogo [32], which can be integrated with Python for parallel and headless execution, offers ease of use but is not optimized for handling massive agent populations or large environments. Mesa [28] is fully Python-based and supports both grid and continuous environments, with extensions for hierarchical agent groupings. It was created in response to the complexity of previous simulators to leverage the Python ecosystem similar to GAMMS. However, the primary focus of Mesa is on modeling artificial societies. In general, we do not find a big intersection between robotics and low fidelity simulators.

4 SIMULATOR ARCHITECTURE

In this section, we will cover the design principles that GAMMS adheres to in its development. We will also briefly discuss why the particular choice was made. Followed by that, we will describe the internal design of the simulator based on the design principles and cover briefly how it helps achieve the objectives mentioned in the previous section. Finally, we conclude the section with an overview of the overall API that a user gets to develop scenarios using GAMMS.

4.1 Design Principles and Choices

We will first discuss the design principles followed by the particular choices made during the actual implementation of the simulator.

Minimal Maintained Interfaces. “Everything is a file” is a well-known concept in Linux philosophy [29]. Having a fewer number of interfaces makes it easier for the tool user to implement as well as extend functionality. As there are only a few things the user needs to know to get started, it helps ensure achieving the “Ease of Use” objective. From a testing perspective, it becomes easier to write unit tests for correctness [5]. Furthermore, for integrating other tools, a developer at most might need to re-implement the interfaces. Hence, keeping a small number of interfaces helps in integration with external tools.

Stable Interface Contract. Only limiting the number of interfaces does not ensure ease in developing and extending the implementations for integrating other tools or pieces of code in general. While many rely on modularity and try to maintain loose coupling of different components, GAMMS accepts the fact that such implementations are time-consuming to develop and put many restrictions on development. Instead, GAMMS takes the “Design by Contract” [21] philosophy for each maintained interface. Since the contracts are formalized, it is easier to reason about the behavior of components when maintaining or extending the software. Furthermore, it becomes possible to maintain a minimum level of documentation that is *complete* for the user to use the tool.

Intermediate Representation when Unstructured. There are certain components that are internal to the development work and do not have a maintained interface. An example in GAMMS is the recording module that emits an optimized representation of the entire sequence of states that happens during a scenario run. What and how the recording happens depends on the implementation in question, hence it becomes impossible to maintain a strict contract without compromising on integration ease. The recorder instead uses maintained representations which provide flexibility to work in an implementation-independent format. This is inline with the concept of transparent runtime model [17]. Any internal optimization modules must have maintained representations so that it is possible to add new optimizations .

Single Point of Access. GAMMS centralizes all simulation state and control through a unified context object that serves as the single authoritative source for the current state of the simulation. This design is inspired from the very successful “Cuda Context” [24] which hid the complexity from the user but has allowed development of LLMs with billions of parameters. GAMMS uses a context

Intermediate representations for visualization optimizations is not implemented yet.

object that internally manages the implementations of the interfaces and allows the tool user to only think in terms of the small set of contracts defined by the maintained interfaces. Furthermore, a common access point prevents the problem of state inconsistency that arises when multiple components maintain their own copies of simulation data. By funneling all state access through a single point, the framework ensures that all components have a consistent view of the simulation state.

In contrast to the strict and structured format of the design principles, we select Python as the language of choice for development. The primary reason is the low learning requirement for Python for the users, as well as the vast amount of libraries that have been developed over the years.

Python is known to be a slow and heavy language, and many attempts have been made to improve the speed by creating separate implementations of the interpreter. Furthermore, Python is dynamically typed, meaning there is no enforced strict typing. While it may appear at first glance that the choice of Python is in complete contradiction, what we want to clarify is that the design principles themselves are completely orthogonal to the choice of any object-oriented programming language. The principles put restrictions solely on the developer and not on the user. As such, the developer can choose to use any language for implementation (even choose to write direct machine code) as long as the user can import the implementation as a normal Python package and get the behavior as described by the interface.

Starting initially with Python has benefits for the development process as well. The first benefit is that the developers can balance between the time investment on new features and optimizations with ease. The second benefit is that it is possible to monkey-patch [10] internally such that any call can be converted into a dispatch for a compiled implementation similar to PyTorch [2]. In general, there is always scope to internally optimize using just-in-time (JIT) compilation without the users having to make any changes (or minor changes) to their code.

4.2 Simulator Components

GAMMS splits sets of interfaces broadly into different parts such that each set is responsible for a specific task or tasks. This is primarily done so that it is easy to arrange the different interfaces in a hierarchical format. Figure 2 gives an overview of the internal structure of GAMMS.

Graph. The Graph module in GAMMS is the component that defines the environment in which agents operate. It represents entities as nodes and their relationships as edges, which can be customized for different environments, such as networks or spatial layouts. The module is designed to be flexible, allowing it to model a range of structures, from simple connectivity graphs to more complex urban or communication networks. The Graph module consists of the *Graph Engine* interface and the *Graph* interface itself. The engine is supposed to be a gateway to create and update graphs, and maintain any form of context that may arise for the graph. The graph itself is a simple interface that allows adding, modifying, and deleting nodes and edges.

Sensor. GAMMS treats a sensor as a module that is responsible for collecting data about the environment and the entire scenario.

Every sensor needs to implement the *sense* method, which is responsible for data collection. As the context object provides a single access point for the entire state of the scenario, it is easy to define custom sensors alongside a wide variety of sensors that are provided prepackaged. Furthermore, as the data flow does not only have to be from the context object, external information can also be injected into the agent through custom sensors. This provides the users flexibility to pass conditioned information that needs to be computed based on various factors. Alongside sensors, the module also consists of *Sensor Engine* which is responsible for maintaining contexts and providing methods to ease the use of prepackaged sensors.

```
ctx.sensor.create_sensor(
    'camera', type=gamms.sensor.SensorType.ARC, sensor_range=50, fov=2.5
) # Creates sensor named 'camera' that senses nodes and edges in range
ctx.visual.set_sensor_visual(
    'camera', node_color=(0, 255, 0), edge_color=(0, 255, 0)
) # Highlights the nodes and edges sensed by the sensor
```

Agent. The Agent module provides an API to register sensors, create the state by collecting data from registered sensors, and process the actions supplied. It is the container object responsible for maintaining the data and state associated with a particular agent. The separate implementation of agents provides a structure that allows it to be agnostic to the type of policy being used – classical, deep learning method, or even human-controlled. Similar to previous modules, there is an *Agent Engine* which maintains contexts and eases the interactions like adding or removing agents. The separation between agent logic and environment interaction also ensures that different agent behaviors can be easily swapped or modified without altering the simulation’s structure.

```
# Single line to create an agent
ctx.agent.create_agent("agent_0", start_node_id=0)
# Visualize agent_0 in red color on the graph
ctx.visual.set_agent_visual(name='agent_0', color=(255, 0, 0), size=10)
```

Recorder. GAMMS provides a lightweight recorder which is integrated with other interface implementations that allow recording a scenario for replay. The recording process can be extremely unstructured, and hence it uses an *Intermediate Representation* (IR) for recording the changes in the scenario. The recorder stores only agent interactions alongside custom user-defined data that needs to be recorded. With the assumption that the user ensures recording external data required for reproducing the entire scenario, it allows the recorder to create highly compressed recordings. Inspired by how StableHLO [1] provides compatibility between IRs across different versions, the recorder IR ensures that replaying an older version with a newer version of the recorder translates the older version recording to the newer version. Furthermore, there is an automatic check on correctness, similar to compilers [14], as recording while replaying a recording from the same version should produce the exact same bytes.

Logger. While setting up the context, it accepts arguments to get different levels of logs for debugging. This module is similar to

A general check for compiler correctness is to recompile the source code of the compiler with itself to check if it produces the same binary file.

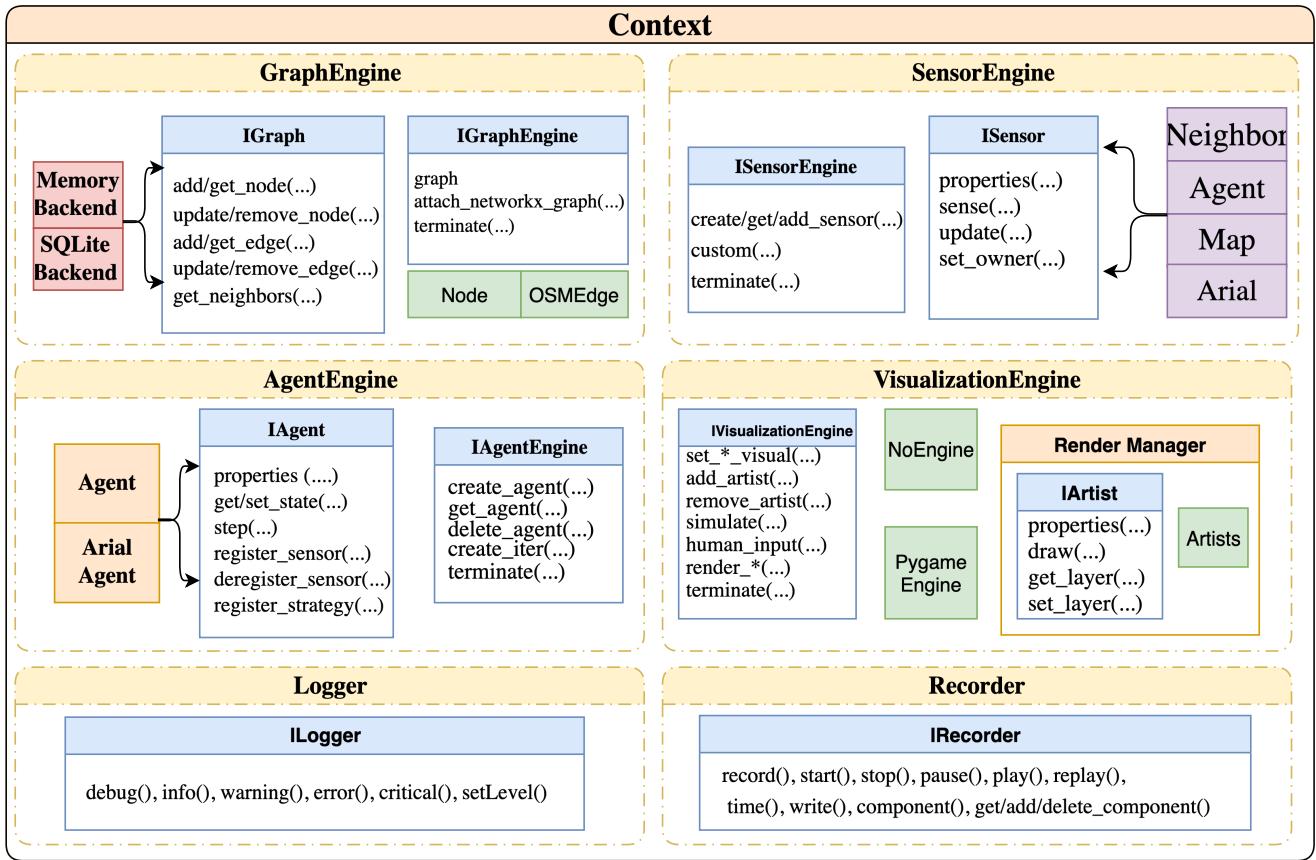


Figure 2: The Figure shows the current overall architecture of GAMMS.

any vanilla logger , but a strict interface is defined so that users can change it as required.

Visualization. GAMMS employs a modular rendering pipeline built around a central Visualization Interface that abstracts the rendering process and delegates execution to specific backend implementations, such as a graphical PyGame engine or a headless NO_VIS mode. When NO_VIS is enabled, the rendering subsystem is bypassed entirely to eliminate graphical overhead, whereas enabling PYGAME activates full visual output. Visual elements in the simulation are represented as artists, each composed of primitive drawing operations like *render_rectangle* and *render_circle*, defined within the interface. During each visualization update, these abstract rendering calls are translated into backend-specific implementations, with optimizations such as view culling applied to skip rendering objects outside the visible window. Simultaneously, user input is processed within the same frame to ensure real-time responsiveness and maintain interactive performance. This round-about way of handling visualization makes it easy to integrate different backends. It opens up the way to easily integrate something like Plotly [12], potentially allowing support for web-based visualization.

Current implementation uses the default python logger.

4.3 Overall Flow and Construction of the Game Loop

The game loop orchestrates the temporal progression of GAMMS simulations through a structured multi-phase approach. Each iteration consists of: environmental state synchronization, agent observation collection, strategy execution and action selection, custom rule enforcement, and visualization rendering. Figure 3 shows the overall flow of a typical scenario from a user’s perspective.

OSM Loading and Graph Construction. GAMMS converts OpenStreetMap data into simulation-ready graph representations through a specialized processing pipeline that addresses key challenges in geographic data utilization for multi-agent simulation. Traditional approaches place nodes exclusively at intersections, creating irregular spacing that complicates agent movement and sensor modeling. GAMMS addresses this through uniform spatial resampling, subdividing road segments into nodes spaced at consistent intervals. The processing pipeline uses OSMnx [6] to do edge resampling to ensure uniform node spacing, intersection consolidation to merge nearby nodes representing the same physical intersection, and bidirectional edge handling that respects transportation mode constraints while providing experimental flexibility.

```
# Location-based loading with automatic processing
G = gamms.osm.create_osm_graph(
    "La Jolla, San Diego, CA, USA", resolution=10)
# Or .. XML-based loading for custom scenarios
G = gamms.osm.graph_from_xml("manhattan_map.osm", resolution=5)
ctx.graph.attach_networkx_graph(G)
```

User Strategies and Policy Architecture. GAMMS employs a compositional strategy pattern where agent behaviors are implemented as callable functions that accept and modify agent state. This functional approach enables strategy composition and runtime behavior modification without simulation restart.

```
# red_strategy.py - Modular strategy implementation
def strategy(state):
    sensor_data = state['sensor']
    for sensor_type, data in sensor_data.values():
        if sensor_type == sensor.SensorType.NEIGHBOR:
            choice = random.choice(range(len(data)))
            state['action'] = data[choice]
            break
def map_strategy(agent_config):
    strategies = {}
    for name in agent_config.keys():
        strategies[name] = strategy
    return strategies
```

Strategy registration supports both individual agent customization and team-based behavior patterns, enabling heterogeneous agent populations within single simulations.

```
# Team-based strategy assignment
red_agents = {name: config for name, config in agent_config.items()
              if config['meta'][team] == 1}
strategies = red_strategy.map_strategy(red_agents)
for agent in ctx.agent.create_iter():
    agent.strategy = strategies.get(agent.name, None)
    agent.register_strategy(agent.strategy)
```

As the entire strategy implementation is only linked via a single function call, the strategy is completely isolated from the simulator. Having such a strategy agnostic structure gives the user the flexibility to work with any classical rule-based strategies, machine learning policies, and human-controlled agents within the same simulation environment. Human players interact through the visualization interface, enabling human-in-the-loop experiments and direct comparison of human versus algorithmic decision-making under identical conditions.

Game Rules Execution and Simulation Loop. The execution framework implements a multi-phase game loop that maintains state consistency across large agent populations while resolving action conflicts deterministically.

The core simulation loop processes all agents simultaneously within each phase, preventing timing dependencies and ensuring fair competition between different strategies.

```
turn_count = 0
while not ctx.is_terminated():
    turn_count += 1
    # Agent observation and decision phase
    for agent in ctx.agent.create_iter():
```

```
state = agent.get_state() # Collect sensor data Dict[str, Any]
agent.strategy(state) # Updates state with 'action' entry
agent.set_state() # Processes the updated 'action' entry
rule1(ctx, turn_count) # Call the rule which is just a function
ctx.visual.simulate() # Visualization and recording
```

The action conflict resolution system handles situations where multiple agents attempt conflicting actions through configurable resolution mechanisms including priority-based systems, random selection, and custom logic tailored to specific experimental requirements.

The custom rule system enables domain-specific simulation logic while maintaining separation between the simulation engine and experimental requirements. Rules execute after agent actions but before visualization, providing the ability to modify simulation state, enforce constraints, implement scoring systems, and determine termination conditions. Rules can implement mechanics ranging from simple win/loss conditions to complex resource management, territory control, or communication protocols between agents.

The visualization and recording integration provides real-time observation capabilities for development and analysis, while comprehensive data capture supports offline analysis and reproducible research practices.

5 EXAMPLES

In this section, we will explore how GAMMS adapts to both simple scenarios, such as a grid world, and more complex ones, like a capture-the-flag setup, while also supporting both ground and aerial agents.

Grid World. Grid World is a discrete multi-agent environment where agents move on a lattice of nodes. It serves as a simple testbed for exploring coordination, path-finding, and basic strategies before scaling to real-world scenarios. A grid can be created manually by adding nodes and edges to a graph object. Suppose we want to have two teams, the blue team and the red team, where the objective of the teams is to reach the starting positions of the other team first. We can easily create some names like *red_1*, *blue_1* for the agents and store them. Figure 4 shows the visualization created by GAMMS.

```
def create_grid(graph, n):
    count = 0 # initialize the edge count to 0
    for i in range(n):
        for j in range(n):
            ... # Use add_node and add_edge
red_team = [...]; blue_team = [...]
def tag_rule(ctx):
    for red in red_team:
        for blue in blue_team:
            ragent = ctx.agent.get_agent(red)
            bagent = ctx.agent.get_agent(blue)
            if ragent.current_node_id == bagent.current_node_id:
                ... # Reset to starting positions
create_grid(ctx.graph.graph, 20) # Generate a 20x20 grid
graph_artist = ctx.visual.set_graph_visual(width=400, height=400)
# ... create agents and set color according to team
while not ctx.is_terminated(): # run loop until context is terminated
    # ... iterate over agents
    tag_rule(ctx) # Check for taggings
```

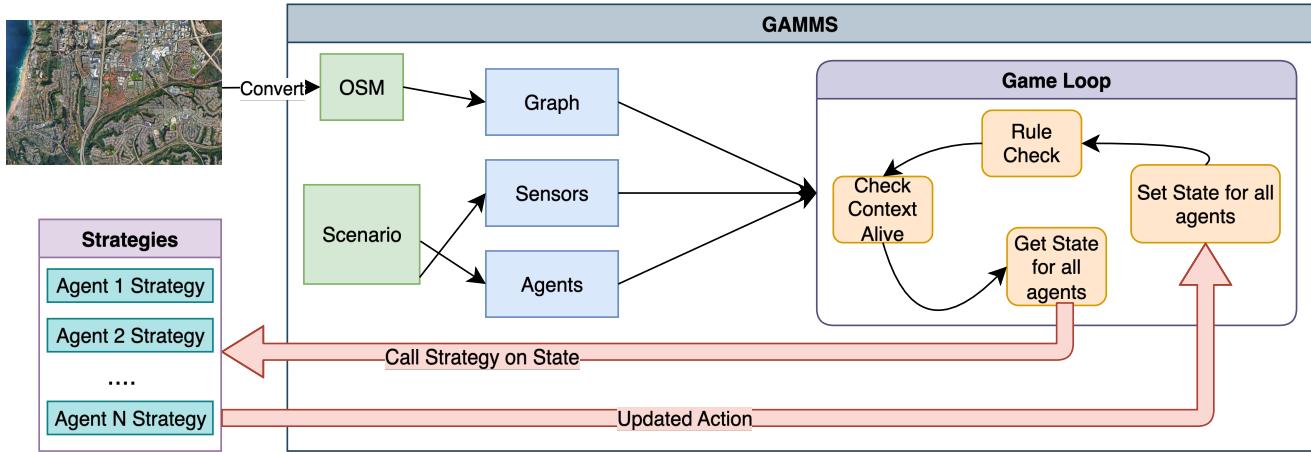


Figure 3: Overall flow and construction of a Game using GAMMS

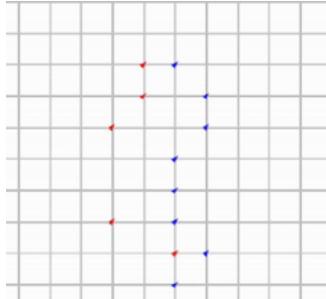


Figure 4: Part of the grid world created by GAMMS.

Capture the Flag. The base grid world can easily be made more realistic by first swapping the graph with a real-world map. Figure 5 shows a part of the La Jolla area in California. Here, the two teams have their own territories highlighted in the figure by their team color. The tagging mechanic is changed such that only the opponent agent with respect to the territory gets reset. Both teams want to capture the flag of the other team first. The entire figure was made purely using GAMMS via its custom drawing API. The *artist* API allows the user to create overlays directly in the world frame.

```
def draw_flags(ctx, data): # Example artist for drawing flags
    red_id = data.get('red_id') # Node id for red flag
    blue_id = data.get('blue_id') # Node id for blue flag
    blue = ctx.graph.graph.get_node(blue_id)
    red = ctx.graph.graph.get_node(red_id)
    ctx.visual.render_rectangle(blue.x, blue.y, 10, 10, color=(0,255,0))
    ctx.visual.render_rectangle(red.x, red.y, 10, 10, color=(255,0,0))
# Create artist with layer above the graph
capturable_artist = gamms.visual.Artist(ctx,drawer=draw_flags,layer=35)
capturable_artist.data['red_id'] = ...
capturable_artist.data['blue_id'] = ...
ctx.visual.add_artist('capturable_artist', capturable_artist)
```

Heterogeneous Robot Systems. A real-world multi-agent system usually has different types of agents that complement each other's abilities to achieve tasks. Drones or aerial agents are great

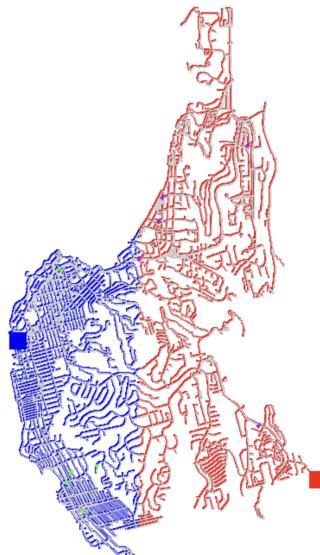


Figure 5: Part of La Jolla OSM converted into a GAMMS graph for the Capture the Flag scenario. Red and blue areas show the territories of the respective teams. The big squares show the flag positions.

for collecting environmental information, but have limitations with direct interaction. Furthermore, in scenarios like fires or areas with many humans, flying a drone closer to the ground can be a safety risk. Hence, we would ideally want a tag-team of aerial agents collecting information for ground agents. GAMMS supports aerial agents in addition to ground agents so that it is possible to test such complex team dynamics. Figure 6 shows an aerial agent along with some other ground agents.

```
ctx.agent.create_agent(
    'aerial', agent_type=gamms.agent.AgentType.AERIAL,
    start_node_id=0, speed=1.0
) # Creates an aerial agent that starts on the ground
```

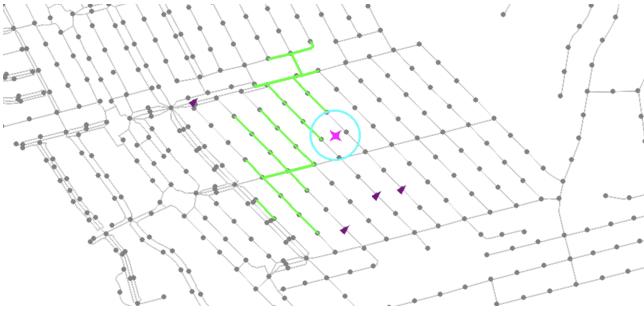


Figure 6: Figure shows a scenario with aerial agents and ground agents. The highlighted nodes and edges are the ones that are visible to the camera of the aerial agent.

6 DISCUSSION & FUTURE WORK

Simulator design is not just about efficient implementations but more about design choices that put the users first. Gymnasium [30], which is a popular choice for defining RL environments, saw its success because of good design choices and provided a structure for researchers and developers to work with. Similarly, the success of CUDA was because of its ease of use [16]. What sets GAMMS aside is that each design choice involved was based on the successes and failures of various frameworks, cherry-picking the choices that made the individual tools favorable to the users. At present, it is being used across six universities and one U.S. government research lab.

Usually when starting with simple examples, it becomes easier for users, particularly researchers, to have a clean single-file, high-quality implementation. From the various code examples we saw in previous sections, it is clear that a user can start working on small examples quickly by writing a small file containing a few hundred lines to get a fully functioning scenario with some basic visualization. The ability to get quick feedback and iterate allows quick development of complex scenarios. Furthermore, the entire package can be installed using any python package manager like *pip*. We intend to keep it as a hard constraint for any future development as the users should never have to wrestle with the toolchain just to get started.

Planned future work includes refining the current feature set and adding more integrated examples, especially those that show how to use external libraries for agent development. There is also ongoing work on transferring policies trained in GAMMS to real robots at WestPoint. GAMMS remains a work in progress, but its current design and usage indicate that it provides a practical starting point for building and experimenting with multi-agent systems.

ACKNOWLEDGMENTS

We gratefully acknowledge support from ARL DCIST CRA W911NF-17-2-0181

REFERENCES

- [1] [n.d.]. StableHLO | OpenXLA Project — openxla.org. <https://openxla.org/stablehlo>. [Accessed 22-09-2025].
- [2] 2020. Registering a Dispatched Operator in C++ #x2014; PyTorch Tutorials 2.8.0+cu128 documentation — docs.pytorch.org. <https://docs.pytorch.org/tutorials/advanced/dispatcher>. [Accessed 22-09-2025].
- [3] Edouard Amouroux, Thanh-Quang Chu, Alain Boucher, and Alexis Drogoul. 2009. *GAMA: An Environment for Implementing and Running Spatially Explicit Multi-agent Simulations*. Springer Berlin Heidelberg, 359–371. https://doi.org/10.1007/978-3-642-01639-4_32
- [4] Mohd Aiman Kamarul Bahrin, Mohd Fauzi Othman, Nor Hayati Nor Azli, and Muhamad Farihin Talib. 2016. Industry 4.0: A review on industrial automation and robotic. *Jurnal Teknologi (Sciences & Engineering)* 78, 6-13 (2016).
- [5] Dirk Beyer and Sudeep Kanav. 2020. An interface theory for program verification. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 168–186.
- [6] Geoff Boeing. 2025. Modeling and Analyzing Urban Networks and Amenities With OSMnx. *Geographical Analysis* (May 2025). <https://doi.org/10.1111/gean.70009>
- [7] Rishabh Chadha and Akhil Docca. 2024. A Beginner’s Guide to Simulating and Testing Robots with ROS 2 and NVIDIA Isaac Sim. <https://developer.nvidia.com/blog/a-beginners-guide-to-simulating-and-testing-robots-with-ros-2-and-nvidia-isaac-sim/>. [Accessed 26-09-2025].
- [8] George Datseris, Ali R. Vahdati, and Timothy C. DuBois. 2022. Agents.jl: a performant and feature-full agent-based modeling software of minimal code complexity. *SIMULATION* 0, 0 (Jan. 2022), 003754972110688. <https://doi.org/10.1177/00375497211068820>
- [9] Jodi Forlizzi and Carl DiSalvo. 2006. Service robots in the domestic environment: a study of the roomba vacuum in the home. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*. 258–265.
- [10] John Hunt. 2023. *Monkey Patching*. Springer International Publishing, Cham, 487–490. https://doi.org/10.1007/978-3-031-35122-8_43
- [11] Takamasa Iio, Satoru Satake, Takayuki Kanda, Kotaro Hayashi, Florent Ferreri, and Norihiro Hagita. 2020. Human-like guide robot that proactively explains exhibits. *International Journal of Social Robotics* 12, 2 (2020), 549–566.
- [12] Plotly Technologies Inc. 2015. *Collaborative data science*. Montreal, QC. <https://plotly.ly>
- [13] Yesung Kim, Dohyun Kim, Jeun Choi, Jisang Park, Nayoung Oh, and Daehyung Park. 2024. A survey on integration of large language models with intelligent robots. *Intelligent Service Robotics* 17, 5 (2024), 1091–1107.
- [14] Ramana Kumar. 2016. *Self-compilation and self-verification*. Technical Report. University of Cambridge, Computer Laboratory.
- [15] Adam Labiosa, Zhihan Wang, Siddhant Agarwal, William Cong, Geethika Hemkumar, Abhinav Narayan Harish, Benjamin Hong, Josh Kelle, Chen Li, Yuhao Li, et al. 2025. Reinforcement learning within the classical robotics stack: A case study in robot soccer. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 14999–15006.
- [16] Chris Lattner. 2025. Modular: How did CUDA succeed? (Democratizing AI Compute, Part 3). <https://www.modular.com/blog/democratizing-ai-compute-part-3-how-did-cuda-succeed#:~:text=The%20benefits%20of%20this%20approach%20were%20twofold.,created%2C%20making%20the%20platform%20even%20more%20valuable>.
- [17] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization*, 2004. CGO 2004. IEEE, 75–86.
- [18] Jacky Liang, Viktor Makovychuk, Ankur Handa, Nuttapong Chentanez, Miles Macklin, and Dieter Fox. 2018. Gpu-accelerated robotic simulation for distributed reinforcement learning. In *Conference on Robot Learning*. PMLR, 270–282.
- [19] Sean Luke, Claudio Cioffi-Reville, Liviu Panait, and Keith Sullivan. 2005. MASON: A Multiagent Simulation Environment. *SIMULATION* 81, 7 (2005), 517–527. <https://doi.org/10.1177/0037549705058073>
- [20] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics* 7, 66 (2022), eabm6074. <https://doi.org/10.1126/scirobotics.abm6074>
- [21] Bertrand Meyer. 2002. Applying ‘design by contract’. *Computer* 25, 10 (2002), 40–51.
- [22] Erfuna Noorani, Zachary Serlin, Ben Price, and Alvaro Velasquez. 2025. From abstraction to reality: DARPA’s vision for robust sim-to-real autonomy. *AI Magazine* 46, 2 (2025), e70015.
- [23] Michael J North, Nicholson T Collier, Jonathan Ozik, Eric R Tatara, Charles M Macal, Mark Bragen, and Pani Sydelko. 2013. Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling* 1, 1 (March 2013). <https://doi.org/10.1186/2194-3206-1-3>
- [24] NVIDIA. 2014. CUDA C++ Programming Guide x2014; CUDA C++ Programming Guide — docs.nvidia.com. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=context#initialization>. [Accessed 22-09-2025].
- [25] New York City Department of Transport. 2022. NYC DOT - Infrastructure - Traffic Signals – nyc.gov. <https://www.nyc.gov/html/dot/html/infrastructure/signals.shtml>. [Accessed 22-09-2025].
- [26] OpenStreetMap contributors. 2017. Planet dump retrieved from <https://planet.osm.org>. <https://www.openstreetmap.org>.
- [27] Hye Jin Sung and Hyeyon Mo Jeon. 2020. Untact: Customer’s acceptance intention toward robot barista in coffee shop. *Sustainability* 12, 20 (2020), 8598.

- [28] Ewout ter Hoeven, Jan Kwakkel, Vincent Hess, Thomas Pike, Boyu Wang, rht, and Jackie Kazil. 2025. Mesa 3: Agent-based modeling with Python in 2025. *Journal of Open Source Software* 10, 107 (March 2025), 7668. <https://doi.org/10.21105/joss.07668>
- [29] Linus Torvald. 2002. The everything-is-a-file principle (Linus Torvalds) – yarchive.net. https://yarchive.net/comp/linux/everything_is_file.html. [Accessed 22-09-2025].
- [30] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. 2024. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032* (2024).
- [31] Joanne Truong, Max Rudolph, Naoki Harrison Yokoyama, Sonia Chernova, Dhruv Batra, and Akshara Rai. 2023. Rethinking sim2real: Lower fidelity simulation leads to higher sim2real transfer in navigation. In *conference on Robot Learning*. PMLR, 859–870.
- [32] U. Wilensky. 1999. *NetLogo*. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. <http://ccl.northwestern.edu/netlogo/>