3D Simulation of Unmanned Aerial Vehicles

Massimiliano De Benedetti, Fabio D'Urso, Fabrizio Messina, Giuseppe Pappalardo, Corrado Santoro
University of Catania – Dept. of Mathematics and Computer Science
Viale Andrea Doria, 6 — 95125 - Catania, ITALY
{m.debenedetti, durso, messina, pappalardo, santoro}@dmi.unict.it

***Abstract*—This paper describes the software architecture of a multi-agent simulator specifically designed to simulate Unmanned Aerial Vehicles (UAVs). The simulator has the aim of helping a developer to design a multi-agent application in which a team or flock of UAVs are employed for a certain mission in an environment. The basic feature of the described tool is the ability to simulate the real physics and dynamics of the entities involved (UAVs), not only by emulating real reactions to forces and torques but also showing the UAVs in a 3D view. The simulator provides the basic classes handling stabilization and control of a quadrotor UAV and let the developer to design her/his application by concentrating only on behavioral aspects of the entities. As usual in multi-agent applications, in order to handle interactions among UAVs, a set of classes is included to simulate a wireless communication system with a certain latency and packet loss probability. The simulator can be also executed without the visualization engine in order to run simulations (that could require a long time to complete) in headless servers. The paper also includes a case-study of a flocking application.**

I. Introduction

In the recent years, *aerial monitoring and inspection* of geographic areas is often performed by means of *Unmanned Aerial Vehicles (UAVs)* [8] that are able to perform flight missions in a complete autonomy. Furthermore, the recent appearance in the market of small *multirotors* (the so-called "drones") along with the availability of high performance and low-cost sensor and control boards, and high capacity batteries, make these unmanned aircrafts a very interesting solution for such kind of applications.

UAVs present other problems that, in the current stage, do not permit them to be a complete replacement of manned aircrafts. The aspect of *application testing* is one of the most important: when designing the algorithms to drive one or more UAV in performing a mission, a software bug could provoke a crash, thus causing the loss of the UAV and the data gathered; in the same way, tuning the parameters of a e.g. a flocking algorithm [14], [9], [3], [7], [6] is quite hazardous when done "in-flight", since a wrong parameter could have dangerous consequences.

For these reasons, the use of *software simulators* becomes mandatory, and helps to verify the correctness, effectiveness and performances of a navigation or collective behavior algorithm before make it run on-board. However, the key requirements of a simulation approach or tool for UAV applications are not only related to the sole behavior aspect: many "real-world" issues, like the physics and dynamics of such kind of objects (which are indeed characterized by a high inertia), can affect quite a lot the performances of algorithms, which, in turn, must consider the physical constraints of the UAVs and their ability to perform certain kind of maneuvers. A good simulator must also be able to emulate the real physics of involved entities, handle collisions and use real-world measurement units, in order to let the developer design her/his algorithms as close as possible to the reality.

The work presented in this paper belongs to the outlined context: the paper describes a software simulator for multi-agent/UAV applications developed by authors and able to take into account the physical model and constraints of the simulated entities, thus aiming at proving a scenario as realistic as possible. The tool is able to simulate the UAVs according to the behavior implemented by the developer; it not only considers navigation and movement but includes also interaction and (wireless) communication, which are treated emulating latencies and limited-range broadcasting. In order to evaluate the effectiveness of the developed algorithms, some performance indexes can be computed during simulation and exported to proper result files.

The simulator is developed in C++ and presents a modular structure able to offer an adequate flexibility to the developers; the *core* provides the basic classes defining the simulation world and the abstract agents; here dynamics and collisions are handled by a real-time physics simulation library, which is *Bullet* [5]; the *uav* module provides the real UAVs, implemented as quadrotor VTOL vehicles, with all the needed PID controllers to perform in-flight stabilization and navigation; *2D/3D visualization* is also possible, and handled by exploiting OpenGL [16], it is an optional feature than can be enabled of the basis of developer desires: indeed graphic display can be disabled in order to run long-term batch simulations.

The paper is structured as follows. Section II outlines the background and discusses the related literature in the field. Section III present the main aspects of the simulator, while Section IV describes a case-study. Section V reports our conclusions.

II. Related Work

The literature reports a large number of agent simulation environments but only few of them considers physical environments, physical constraints and provides 2D/3D display capabilities.

NetLogo [17] is one of the most popular toolkits for agent simulation; it is Java-based and is able to perform simulation of a large number of agents. Agents live in a virtual environment which is bi-dimensional and discretised into small square cells: each cell can be occupied by a (static) environment element or by a (mobile) agent. Ad an-hoc language is provided to model the behavior of agents that can live, move and interact in the 2D environment. Beside these simple environmental

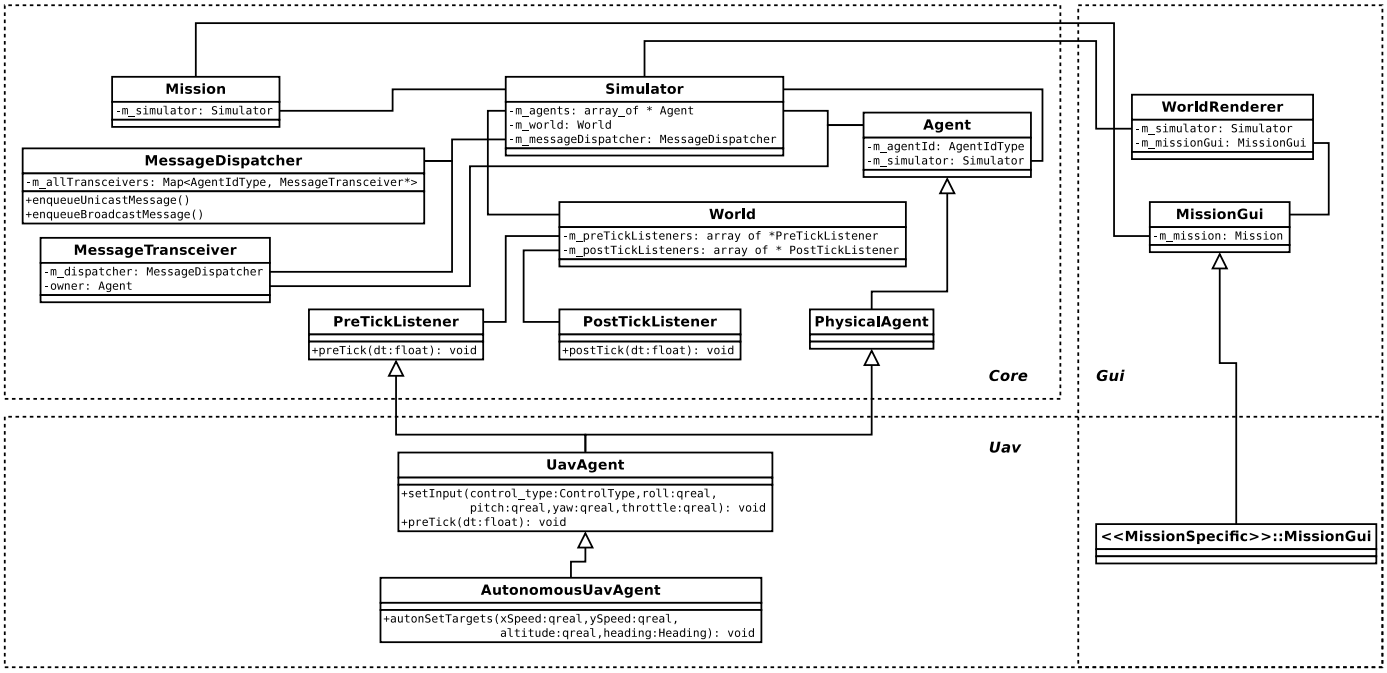

Fig. 1. Simplified Class Diagram of the Simulator

aspects, NetLogo is not able to simulate real-world physics and dynamics of entities nor to provide a 3D display.

Tridimensional capabilities are instead provided by *Breve* [10], a 3D environment for the simulation of decentralized systems and artificial life. It allows the designer to simulate continuous time and 3D space by including an interpreted object-oriented language, an OpenGL display engine, as well as the support for body physical simulation and collision resolution. Agent behavior is implemented in Python [1] or by another easy-to-use language named "steve". Breve is no longer maintained since 2009, but the simulation environment is still used, thus the author of Brave has partially restored the website[1].

PALAIS [15] is another 3D simulation tool for prototyping, testing, visualization and evaluation of AI algorithms for games; the designer can define the game by executing arbitrary three-dimensional scenes and behaviors. A scripting environment and a simple programming interface are also provided for simulation control and data visualization. This scripting interface is minimal and can be also accessed via JavaScript. One of the interesting feature of PALAIS is the ability of sharing a project in order to, e.g., collaborate with peers and build up showcases for algorithms and behaviors.

ARGoS [12] is an open source, modular, multi-robot simulator to simulate real-time large heterogeneous swarms of robots. User can easily add custom features and allocate further computational resources where needed, moreover multiple physics engines can be used and assigned to different parts of the environment.

Gazebo [11] is an open-source 3D robotic simulator, able to simulate the physics and dynamics of any mechanical structure made of joints. It offers the ability to drive joints with forces or torques and integrates the definition of sensors to let the robot perceive the environment. Behavior programming can be done by means of external software interacting with Gazebo by means of its API. A ROS [13] interface is also available.

A tool similar to Gazebo is *VREP* [4], a commercial simulator also available in a free educational (and limited) license. From a certain point of view, it offers more features than Gazebo: indeed VREP has a quite friendly user interface that can be used to both define robot structure—by means of plug-and-play action for robot components—and program the behavior. Programming can be performed by means of the built-in Lua interpreter (which is however a little bit weird) or by using a C/Python API.

With respect to the solution reported in this paper, the cited simulators are more general purpose, while our work focuses mainly on multi-rotor UAVs. Therefore, while, in our simulator, UAV models and all the UAV stabilization and navigation algorithms are ready to use, in the other tools not only the UAV model must be designed, but also the control algorithms must be specifically written and tuned. Therefore, in our solution the designed can only concentrate on "high-level" behavior and interaction aspects rather than control issues.

III. Architecture of the Simulator

The software simulator described in this paper is written in C++; its simplified class diagram is reported in Figure 1. it is composed of three main modules: **Core**, **Uav** and **Gui**.

A. The Core

The **Core** consists of the classes that represents the basic entities of the simulator. `Mission` is the main class that has the responsibility of instantiating the simulator and starting the

[1] www.spiderland.org

overall system; indeed it reads a configuration file (in ".INI" format) that specifies all the parameters of the simulation, including the type of the agents employed, the presence of the GUI, the size of the environment and other mission-specific items. This class creates, at start-up, an instance of class `Simulator`, which is a container that refers all the `Agents` instantiated in the simulation and the `World`, which is the class handling the physics and dynamics of agents; thus class `World` relies on the Bullet library. Simulation execution is performed by `World` by using a discrete-time approach[2]; for each time interval, this class triggers the invocation of callback methods of the simulated entities as well as the updating of data on their dynamics (via the Bullet library); indeed, an event delegation model is adopted by the definition of two *listeners*, `PreTickListener` and `PostTickListener` that are triggered respectively *before* and *after* the update of the physics of the agents.

The **Core** includes also the abstract classes `Agent` and `PhysicalAgent` that must be extended in order to implement the specific user-defined agent behavior. These two classes are designed to make a distinction between *logical* and *physical* agents: the latter category includes agents located at a precise point in the simulated space/environment and with the ability to move, while the former category refers to other not location-aware entities that however live in the environment and needs to be simulated. Each agent is uniquely identified by an *id* (class `AgentIdType`) that is indeed an integer.

Interaction among agents is also handled in the **Core**. Since the aim is the simulation of physical entities, the communication is modeled by simulating a wireless transmission channel with a configurable latency and loss probability. Agents have the capabilities to communicate with their peers by means a short distance communication, and with a possible *base station* through a long-range system; both kind of channels support unicast and broadcast communication. These kind of communication are handled by classes `MessageTransceiver` and `MessageDispatcher`. The former is a class associated to each agent and basically implements the message reception queue. The latter is a singleton that has the objective of collecting sent messages thus delivering them to receiving transceivers. A message is characterized by the *id*s of sender and receiving agents (or only sender, is the message is sent in broadcast) and the *payload* which can be any C++ type[3].

B. Unmanned Aerial Vehicles

The objective of the simulator is to study the behavior of UAV-based applications, therefore the **uav** module provides all the necessary classes to simulate the behavior and dynamics of UAVs. Two kind of UAVs are simulated, which differ on the basis of the commands that can handle. Class `UavAgent` implements a UAV that can be externally controlled through the imposition of certain *Euler angles* to the airframe—i.e. *roll*, *pitch* and *yaw*—and a certain amount of power for propulsion (*thrust*). On the other hand, class `AutonomousUavAgent` can be controlled by directly imposing target *horizontal speeds* v_x and v_y of the UAV (relative to the body frame), *heading* and *altitude*. Both kind of agents are characterized by a certain location in the physical space that resembles the geographical coordinates of a real environment; this location (as well as all the other physical parameters) are updated by the Bullet library on the basis of the movements of the UAV body.

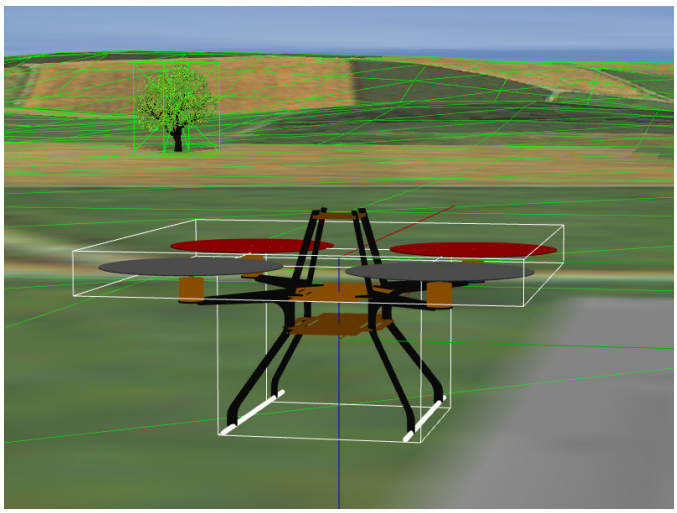

Fig. 2. The graphical model of the UAV implemented in the simulator

The UAV modeled is a classical X-shaped quad-rotor whose graphical model is depicted in Figure 2. To simulate it, the classes of the **uav** module implement all the control algorithms needed to perform in-flight stabilization and navigation. Motors are modeled by applying a certain *drag force* to the points of the body where motors themselves are placed; such drag forces are determined on the basis of the outputs of the control algorithms and using a simplified mathematical model for motors and propellers.

Different *control modes* are implemented. In **manual** mode (Figure 3), the input throttle, pitch, roll and yaw values are directly sent to the mixer that calculates the power to apply to the motors. In **attitude** mode (Figure 4), *pitch* and *roll* are controlled by means of two PID[4] controllers that compare target (desired) angle values with the real ones returned by the Bullet physical engine; these PIDs are thus used to perform in-flight stabilization.

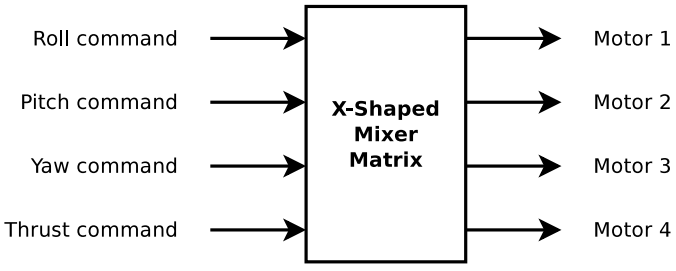

Fig. 3. Manual mode

When **GroundSpeedAndVertSpeed** mode is selected (Figure 5), targets are given in terms of speeds relative to the body frame v_x, v_y and v_z; such speeds are compared with the actual

[2]In the implementation, time tick is fixed to 2.5 ms, which is the sampling time usually employed in control loops of flight algorithms for multi-rotors.

[3]Indeed, since the simulator uses the Qt library, the payload is implemented using a `QVariant` type.

[4]Proportional-Integral-Derivative

ones (determined by Bullet) and the differences are sent to speed PIDs which, in turn, drive the PIDs on Euler angles.

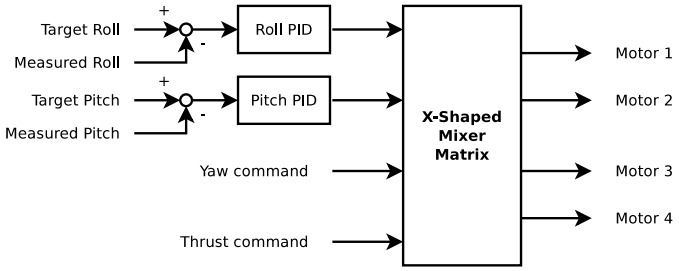

Fig. 4. Attitude mode

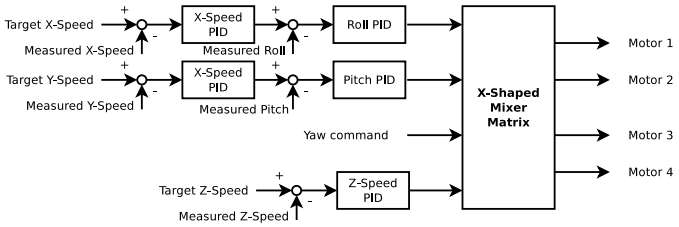

Fig. 5. GroundSpeedAndVertSpeed mode

In **GroundSpeedAndAltitude** mode, an altitude PID is included to the previous mode to control the target height of the UAV (Figure 6).

Finally, with **GroundSpeedAltitudeAndHeading**, control on target heading is included with respect to the previous mode (Figure 7).

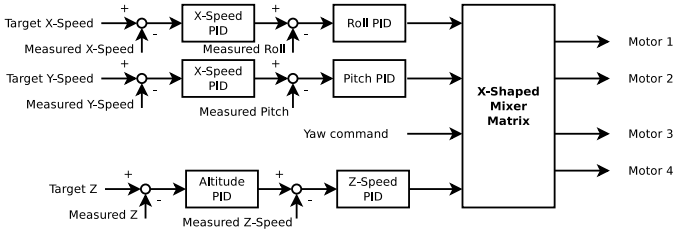

Fig. 6. GroundSpeedAndAltitude mode

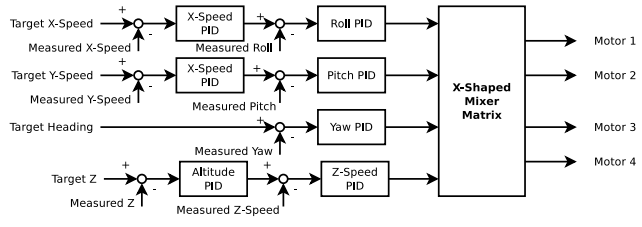

Fig. 7. GroundSpeedAltitudeAndHeading mode

During the development of a certain specific behavior algorithm, it has been helpful to take a manual control of agents. For this reason, the simulator includes also a graphical *virtual joystick* as well as the interface to a real USB joystick.

Classes of the **uav** module also implement *fault detection*. Faults can occur due to UAV-to-UAV and UAV-to-terrain collisions or be configured to occur in a UAV with a certain probability; the latter feature is used to simulate problems in hardware (e.g. propeller breaking, battery drain, etc.) or in software (e.g. bugs in flight control stack). In both cases, UAVs involved in the fault are killed and removed from the simulation. Collisions are handled at each simulation step by retrieving, from Bullet engine, information about all intersections among rigid bodies and then killing the relevant agents. In order to perform tests, the GUI also allows the user to kill an agent manually.

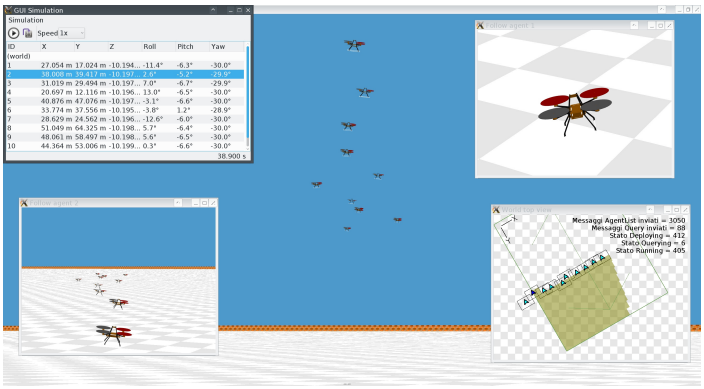

Fig. 8. A screenshot of the Simulator

C. The Graphical User Interface

The **Gui** module provides the Graphical User Interface with capabilities to visualize the physical environment, the agents, and to inspect their state. A screenshot is shown in Figure 8. In order to understand the real behavior of UAVs by looking at its flight, several windows can be opened with different point of views of the physical scenario: *(i)* the overall scene in 3D; *(ii)* a 2D display from the top; *(iii)* a 3D subjective view with respect to a specific agent. In all the windows, the view camera can be moved by using the mouse. Physical and graphical models are loaded from geometry definition files (OBJ) that can be generated by 3D graphic modelers, such as Blender [2].

From the software point of view, the **Gui** is composed of two basic classes. `WorldRenderer` has the task of performing the display of the UAVs and the background and exploits the OpenGL library. `MissionGui`, on the other hand, provides the necessary code to display the list of agents and interact with the simulation (starting/stopping the simulation, inspecting the agents, opening new views, killing agents); it is based on the QT5 library.

As the architecture of Figure 1 shows, the graphical interface is separated from the simulation engine: this is needed to perform batch simulation on headless high-performances servers and then gather results.

IV. Case Study: Simulation of a UAV Flocking Application

In this section we describe the case-study of a flocking application where a set of UAVs are employed to perform aerial inspection of a specific zone of terrain. The aim is to monitor an area using a set of UAVs each equipped with a *camera sensor* able to take aerial pictures. The idea is to have

a collaborating application so that, should an UAV fail, the other ones can identify the fault and try to recover data lost by performing a re-scouting of the areas relevant to the faulty agent. To this aim, UAVs organise themselves in a flock that scouts the area on the basis of a leader-followers approach. The algorithm used is fully described and evaluated in [7], [6], so we concentrate here on simulation aspects rather than the flocking approach itself.

Implementing the simulation requires to write some classes that extend the basic classes of the simulator, i.e. `AutonomousUavAgent`, `Mission` and `MissionGui`. In our case, the `FullAgent` class extends the first one of the previous three and implements the behaviour of the flocking and area coverage algorithms. The `Mission` class is instead extended in order to provide mission-specific code: the extended class will have the task of creating the agents (by reading the configuration file described below) and start the simulation. Finally, the mission-specific `MissionGui` needs to be written in order to include the basic code for agent rendering and other GUI aspects, if needed. All of these classes (together with the other code needed for the simulation) are then compiled into a *shared-object* in order to take advantage of the *plug-in* feature of the simulator: the shared-object is loaded by the simulator at run-time on the basis of a certain field present in the configuration file; in this way it is easy to guarantee the coexistence of different simulative algorithms, along with the ability to select at runtime the desired mission approach.

Simulation parameters are specified by means of text files (INI format). Listing 1 shows a part of the configuration used in the flocking application. The INI file is composed of three sections: *(i)* **global parameters** (section `[Global]`); *(ii)* **list of agents** to instantiate at the beginning of the simulation (section `[Uavagents]`); and *(iii)* **list of areas** to explore, along with related coordinates (section `[AreasToExplore]`).

In the first section, parameter `missionType` represents the model to adopt for the UAV and actually represents the name of the dynamic library that contains the customized code of a specific mission to run; parameter `autoStart` instead affects non-batch simulations, on which the GUI is activated. The second section (`[Uavagents]`) specifies the number of agents and the initial position and heading, as well as the identifier, of each of them. Finally, the third section specifies the areas to explore, their location, and the parameter *rectHeading*, which is the direction agents will follow while exploring.

Listing 1. Sample configuration

```
[Global]
missionType=uav/FullAgentWithBaseStation
autoStart=false ; start simulation automatically

[UavAgents]
size=10 ; Total number of UAVs
1\agentId=1;
1\initialPos=@Point(0 0)
1\initialHeading=0
2\agentId=2
2\initialPos=@Point(2 2)
2\initialHeading=0
; ...

[AreasToExplore]
size=1 ; Rectangle count
1\areaId=1
1\rectCenter=@Point(50 50)
1\rectSize=@Size(80 100)
1\rectHeading=60
```

V. Conclusions and Future Work

This paper has described the architecture of multi-agent simulator specifically designed for unmanned aerial vehicles. The tool described is able to simulate the physics and dynamics of several quadrotor UAVs engaged in a cooperative mission. The simulator is written in C++ and the designer can use it and implement the desired behavior by simply subclassing the `AutonomousUavAgent`. Since all aspects related to navigation, stabilization and control of the UAV are already provided by the basic classes of the simulator, the designer can concentrate only on the "high-level" behavior of the agents and their mutual interaction. A 3D display engine is also provided, in order to visualize the UAVs flying in the environment and thus make it possible to verify, at sight, the real agent's behavior. Indeed, the display engine can be disabled in order to run the simulation in a headless server and then gather output results.

As future work, we plan to include a scripting language in order to let developers to implement their own simulation by means of a simple approach rather than being involved in writing and compiling C++ source code.

VI. Acknowledgements

This work is partially supported by projects PRISMA PON04a2 A/F and CLARA funded by the Italian Ministry of University, and FIR-2014 funded by the University of Catania.

References

[1] "The python language reference manual," 2016, https://docs.python.org/3/reference/index.html.

[2] Blender Foundation, "Blender," https://www.blender.org.

[3] N. Bouraqadi and A. Doniec, "Flocking-based multi-robot exploration," in *Proceedings of the 4^{th} National Conference on Control Architectures of Robots*, Toulouse, France, 2009.

[4] Coppelia Robotics, "V-rep - virtual robot experimentation platform," http://www.coppeliarobotics.com/.

[5] E. Coumans *et al.*, "Bullet physics library," *Open source: bullet-physics.org*, 2013.

[6] M. De Benedetti, F. D'Urso, F. Messina, G. Pappalardo, and C. Santoro, "Uav-based aerial monitoring: A performance evaluation of a self-organising flocking algorithm," in *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*. IEEE, 2015, pp. 248–255.

[7] M. De Benedetti, F. DUrso, F. Messina, G. Pappalardo, and C. Santoro, "Self-organising uavs for wide area fault-tolerant aerial monitoring," in *XVI Workshop "Dagli Oggetti agli Agenti"*. CEUR-WS, 2015.

[8] S. Gupte, P. I. T. Mohandas, and J. M. Conrad, "A survey of quadrotor unmanned aerial vehicles," in *Southeastcon, 2012 Proceedings of IEEE*. IEEE, 2012, pp. 1–6.

[9] Gbor Vsrhelyi, Csaba Virgh, Gerg Somorjai, Norbert Tarcai, Tams Szrnyi, Tams Nepusz and Tams Vicsek, "Outdoor flocking and formation flight with autonomous aerial robots," in *Submitted for publication to the IEEE/RSJ International Conference on Intelligent Robots and Systems*, ser. IROS '14, Chicago, IL, USA, 2014.

[10] J. Klein, "Breve: a 3d environment for the simulation of decentralized systems and artificial life," in *Proceedings of the eighth international conference on Artificial life*, 2003, pp. 329–334.

[11] Open Source Robotic Foundation, "Gazebo simulator," http://www.gazebosim.org.

[12] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle *et al.*, "Argos: a modular, multi-engine simulator for heterogeneous swarm robotics," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2011, pp. 5027–5034.

[13] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[14] C. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the* 14^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 25–34.

[15] P. Schwab and H. Hlavacs, "Palais: A 3d simulation environment for artificial intelligence in games," in *Proceedings of the AISB Convention*, 2015.

[16] D. Shreiner, G. Sellers, J. M. Kessenich, and B. Licea-Kane, *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[17] S. Tisue and U. Wilensky, "Netlogo: Design and implementation of a multi-agent modeling environment," in *Proceedings of agent*, vol. 2004, 2004, pp. 7–9.