Collaborative Project

Deliverable D4.2
Simulator

Project acronym:	AIRobots
	Innovative Aerial Service Robots for Remote Inspections by Contact
Grant agreement no:	FP7 248669
Project web site:	www.airobots.eu

Due date: February 29, 2012	**Submission date: March 16, 2012**
Start date of project: February 1, 2010	**Duration: 36 months**
Lead beneficiary: UNIBO	**Revision:**

Nature: R	**Dissemination level: PU**
R = Report P = Prototype D = Demonstrator O = Other	PU = Public PP = Restricted to other programme participants (including the Commission Services) RE = Restricted to a group specified by the consortium (including the Commission Services) CO = Confidential, only for members of the consortium (including the Commission Services)

Executive Summary

This document describes the AIRobots simulator development and provides an user manual and installation guide for future reference. Although the deliverable will contain some remarks to the physics and mathematical modeling of the simulated rigid bodies, it is not intended to be an exhaustive reference for the robotic components that are being used. For this purpose, please refer to the deliverable D4.1. This report will cover mostly the end-user aspects and will provide an architectural description to better understand the simulation logic from a software point of view.

The first section describes how the simulation environment is represented with respect to the real world data and scenarios, and provides details of the Aerodynamics simulation and manipulator integration for the use with the Blender Game Engine. A brief reference of the programming libraries and interfaces is then given in order to help with the understanding of the infrastructure that is being used and to ease future integrations or updates. Finally, a guide to the installation process describes the steps needed to accomplish a fully functional and working environment on new available computers, and the user manual guide the operator to the several features of the simulator. Due to the evolving nature of the software provided, this document will be extended and integrated periodically to reflect the further improvements in the simulation environment.

Acronyms
UAV: Unmanned Aerial Vehicle
MAV: Micro Aerial Vehicle
IMU: Inertial Measurement Units
EKF: Extended Kalman Filter
UKF: Unscented Kalman Filter
GPS: Global Positioning System
NED: North-East-Down
LPF: Low Pass Filter
SURF: Speeded Up Robust Features
BRIEF: Binary Robust Independent Elementary Features
PCA: Principal Components Analysis
RTWT: Real-time Windows Target (Matlab Toolbox)

TABLE OF CONTENTS

1 Introduction 3

1.1 Software Requirements for the Development of the Simulation Environment 3

2 Simulation Environment 5

2.1 Aerial Vehicle 6

2.2 Manipulator 7

2.3 Sensors 8

2.4 Environment modelling 9

3 Software Integration 10

3.1 Integration with low-level control 11

3.2 Integration with the supervisory control 11

3.3 Integration with ROS 11

4 Programming reference 11

4.1 Simulink components 12

4.2 MORSE architecture components 15

4.3 In-Blender programming 18

4.4 Protocols 20

5 Installation guide 20

5.1 Installation overview 20

5.2 Install the Linux Box 21

5.3 Install the Windows Box 23

6 User manual 23

6.1 Starting the environment 23

6.2 Parameter configuration 24

6.3 Running the simulation 25

6.4 Releasing waypoints 26

1 Introduction

The development of the aerial service robot prototypes includes situations in which using directly the real vehicle would be risky, unpractical or simply time-consuming. In these scenarios, which include the training of the operator and the validation of new low-level and high-level functionalities, an advanced simulation environment has to be taken into account [7]. To be effective in both the design and validation of the control algorithms, the simulator environment should appear transparent both to the rest of the control architecture and to the human operator: switching from simulations to real flight operations should be achieved as seamlessly as possible. This fact implies that, on one side, the simulation environment should have the same software interface as the real prototype and, on the other, that the dynamical model of the aerial system and of the environment should match as far as possible the real ones.

The simulation environment has been designed to address a number of relevant scenarios ranging from simple free-flight operations to the physical interaction with a realistic 3D reconstruction of the end-user environment.

The most simple kind of experiments that can be carried out with the simulator are given by free-flight operations with the two prototypes of aerial service robots developed in the project. In this scenario the simulator validates both the dynamical model of the system and the performances of the low-level control law. Human Interface Devices (HIDs), such as a simple joystick, can be employed to generate the reference signals for the low-level free-flight controllers in order to pilot the vehicle in a virtual 3D environment in the same way as they can be employed for real flight tests.

The second and more advanced kind of experiments that have to be achieved are given by the validation of the high-level control and sensor fusion algorithms. In particular, by means of an accurate 3D model representation of the operational scenario and the correct implementation of the sensors mounted onboard, the effectiveness of high-level vision based algorithms can be validated by considering some of the characteristics of the real environment (e.g. textures, illuminations, etc). The high-level sensor fusion, together with the high level supervisor, can be employed to test mission planning, obstacle avoidance and re-planning in the selected virtual world. Finally, telemanipulation algorithms can be validated by integrating also haptic devices in the simulation.

Through an accurate modeling of the environment in which the robot has to operate, the simulator can also be used to validate low-level control algorithms that are able to handle physical interaction with the environment. In particular the simulator is capable to model the manipulator device installed on the vehicle and allows to perform several tests on position and force control algorithms for the overall multi-body system, as it has been shown in Deliverable D4.1. All these features can be achieved by implementing specific force and contact sensors in the simulation environment.

Finally, the simulator can be employed to train the operator in the selected end-user environment. This is achieved though the integration of all the advanced functionalities discussed above, with particular attention to the accuracy and refinement of the environment in which the aerial service robot is required to flight and though.

In order to be really effective in validating both the low-level and high-level control algorithms, the simulator has been designed according to a modular structure [5] in which both the dynamical models of the aerial robots and the control algorithms can be easily integrated as separated modules. This fact allows easily to obtain also software-in-the-loop and hardware-in-the-loop simulations by replacing the different control modules with the real controllers installed on the prototypes. Observe that this characteristic is of paramount importance in order to actually keep updated the simulation environment with the latest version of the controllers developed in the project at no additional development cost. And also, on the other hand, it permits to keep separate the business logic from the visual and graphics components for future improvements or upgrades.

1.1 Software Requirements for the Development of the Simulation Environment

Aerial service robotics represents a new field of research for which a complete off-the-shelf tool able to accomplish advanced simulations is not yet available. However, a number of open-source and commercial software tools addresses the issue of advanced simulations for different robotic platforms, including aerial robots - [18] - mobile robots - [12], [8], [6], [3], [14], [16], [13], [18], [17], [16] - and robotic arms - [15], [10]. The AIRobots simulator takes then into account only the open-source software and, for the existing projects, at first, to take advantage from the functionalities already available in some open-source architectures, and then, to join the international open-source robotic community. Accordingly, in this part we show the main guidelines that have motivated the design of the simulator environment and the choices in term of the existing open-source tools to be integrated in our project.

Mandatory requirements

A number of requirements can be detailed in order to implement the basic AIRobots functionalities. With respect to a standard simulation environment for aerial systems, where essentially only free-flight is considered, AIRobots main goals require to take directly into account the physical interaction between the vehicle and the environment. Apart this important issue, the simulator should also be able to integrate the advanced control architecture of the real system, including all the components required for advanced human-machine interaction and high-level vision based control algorithms. All this features should be linked together in an effective software environment.

Physics As stated above, the physics simulation is a crucial part of the simulator, because one of the main task that must be accomplished is to help in studying the dynamic behavior of the UAV in multiple operational conditions, like when it is freely flying in the air or when physical interaction, such as collisions, with the environment is performed. With this respect, the simulator must be able to calculate three-dimensional collisions and to apply the resulting reaction forces to the robot. Moreover, in order to correctly model physical interaction, realistic friction models and advanced contact models should also be available. Regarding the manipulator, in order to model the device attached to the vehicle, multi-body systems should also be implemented by using different types of joints. Accurate kinematical and dynamical models of the manipulator are also required.

To implement and validate the different feedback control strategies, both low-level sensors, such as IMU, and high-level sensors, such as vision, should be available within the simulation environment.

Haptic feedback During inspection-by-contact operations, the vehicle should approach the surface to be inspected, make contact with it and then slide along it, acting as a sort of virtual hand for the operator. To get a realistic feeling, the operator will drive the vehicle using a haptic device able to provide force feedback. On the physical setup, force sensors are installed in specific points of the vehicle's body, providing information about the force and torque applied in that point. This same kind of information must be computed inside the simulator and sent to the haptic device in order in order to provide the operator with the same feeling of real operation scenarios.

These feature requires to implement in the simulator environment both a realistic compliant contact model of the environment and the force and contact sensors able to detect the contact forces exchanged during the interaction. All the haptic devices used by the operator should also be integrated within the simulation environment.

Video feedback Stereoscopic cameras are installed on the real UAV, whose output is sent both to the operator and to high-level vision algorithms able to estimate the speed and the pose of the vehicle with respect to the environment. In a simulation environment, speed and distance information can be obtained from the simulator itself, but video feedback to the user is required, therefore at least two cameras or a single stereoscopic camera must be installed. Information from the virtual cameras must be available in a format which can be easily extracted and sent over the network to the operator's console. Furthermore, the information retrieved from the simulated virtual cameras can be used to speed up the development and testing of the vision algorithms. To validate the vision algorithms, parameters such as focal length, aperture and resolution of the cameras must be available and it should be possible to reflect the ones of the real setup.

Frame rate In order to simulate the dynamical model of the real vehicle, the simulator must be able to read inputs and send outputs at a frequency of about 100Hz. The above sample time is compatible with the dynamical properties of the closed -loop attitude and position control subsystems implemented to stabilize the real physical prototype.

Video output can have a lower sample-rate, in fact the onboard physical cameras are able to stream data at approximately 10-20Hz.

It must then be possible to have at least two different communication channels, one for the high-speed small packets which carry control information and position / attitude information for feedback, and one for the image data.

Non-mandatory requirements

These requirement relate mainly to the modelling of the UAV and the virtual environment.

Graphical modelling tool and Open source software The simulator may use three-dimensional models of the UAV that are imported from the available CAD design. To model the end-user environment, textures obtained from the real set-up should be integrated in the simulations. This requirement is mandatory if high-level vision-based sensor fusion has to be tested with the simulator. Open source software can be customized to implement missing features, and given the peculiar kind of application, this is an highly desirable feature.

Limitation of the simulator engine

Another important specification that had to be defined before examining the available solutions was about what has to be calculated inside and what can be delegated outside. It has previously been highlighted that control logic, given its development process, is already available in a MATLAB/Simulink model, hence it is best kept outside of the simulator. The output of the control logic is an array of six values, namely forces and torques along the UAV's axes. This is then the input to the simulator. Since aerodynamics can be modelled in MATLAB, the values sent to the simulator will actually correspond to resultant forces and torques generated by the fan and the aerodynamic control surfaces.

In order to close the loop, information about the position and pose of the UAV must be fed to the control logic, and this must be the output of the simulator. Gravity, interactions and collisions, everything pertaining to dynamics and motions must be calculated within the simulator.

Thus, the simulation engine must be able to communicate with the different components using a networked computer environment or serial communication in order to exchange data in a fast and reliable way. Since the whole main simulation loop has to be performed in less than 10ms to attain a frequency of 100Hz, the communication protocol must be kept simple enough so that it doesn't require time-consuming parsing and elaboration. It must nevertheless be flexible enough to allow the input of different data, for driving the UAV, controlling appliances, starting and stopping the simulation and so on.

All calculations which are potentially time-consuming must be kept outside of the simulator, so image data must be acquired by the virtual cameras but need not be manipulated and can be output as raw data.

Choice of the tools

With all the requirements in mind, the search of some open-source base tools to start developing the simulation environment ended on the choice of the Blender 3d content creation suite [11], [4], [2]. Blender provides a 3d rendering framework and a game engine that allows to interact with the 3d scene that is presented to the user. Moreover, the Blender architecture can be expanded using the *python* programming language and in this context the MORSE (Modular OpenRobots Simulation Engine) library [14] has come in aid for our ``robotics'' purposes.

MORSE defines a neat architecture of components, tailored specifically for robotics, providing abstractions for robots, sensors and actuators. It also defines a middleware layer that can be implemented according to the requirements of the deployment. With this architecture, the same simulation can interact with ROS nodes or use proprietary communication protocols without changing the simulation itself, but just by switching middlewares. Different middlewares can be used simultaneously to communicate with different clients.

2 Simulation Environment

The simulation environment is intended to substitute and emulate the physical parts of the real UAV in the most realistic possible way, while using all the same control algorithms and integrating with external systems such as the Trajectory planner. In particular, the simulation environment must take care of all physics simulation aspects, like rigid body dynamics, collisions, aerodynamics and sensors output. The simulator's components have been implemented part in Simulink and part in Blender, using the Python language. Simulink and Blender communicate through UDP packets, so communication has to be kept as lean as possible, developing all the integration with the low-level control in Simulink and using Blender only when 3D physics simulation is involved.

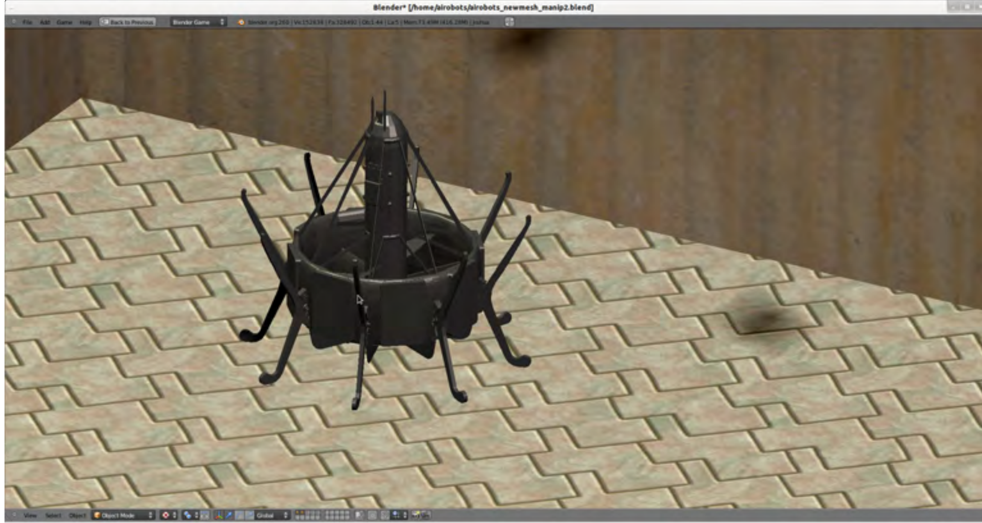


Figure 1: The 3D model of the UAV in a running simulation.

2.1 Aerial Vehicle

Rigid Body Dynamics

The Game Engine embedded in Blender is used to compute the rigid-body dynamics for the virtual UAV and dynamical model of the environment. The engine uses the Bullet Physics Library that provides

- Discrete collision detection for rigid-body simulation.
- Support for in-game activation of dynamic constraints.
- Full support for vehicle dynamics, including spring reactions, stiffness, damping, etc.

Thus, the simulated UAV can fly and interact with the virtual environment in a realistic way, and the data retrieved can be exported to the Simulink controller.

A mathematical model for the rigid-body dynamics can be given by means of the so called Newton-Euler equations:

$$\begin{aligned} M\ddot{p} &= Rf^b \\ J\dot{\omega} &= -\omega \times J\omega + \tau^b \end{aligned} \tag{1}$$

where f^b and τ^b represent respectively the vector of forces and torques applied to the vehicle expressed in the body frame, M the vehicle total mass, J the diagonal inertia matrix, $p = \text{col}(x, y, z)$ the position of the center of mass, ω the angular velocity expressed in the body frame and R the rotation matrix relating the body frame with the inertial frame.

Aerodynamics

Since Blender cannot natively handle the aerodynamic forces that applies on the control vanes of the UAV, and to expose an abstract and generic torque/force layer to be used with the different prototypes, the aerodynamic laws that apply to the individual prototypes are implemented using a Simulink block.

With reference to Fig. 3 and to Deliverable D4.1, the vector of forces f^b in the body frame can be computed as

$$f^b(\underline{\alpha}) := \begin{bmatrix} 0 \\ 0 \\ -T \end{bmatrix} + \begin{bmatrix} \sum_{i=1}^{8} \mathbf{L}_i(\alpha_i)^T i^b + \sum_{i=1}^{8} \mathbf{D}_i(\alpha_i)^T i^b \\ \sum_{i=1}^{8} \mathbf{L}_i(\alpha_i)^T j^b + \sum_{i=1}^{8} \mathbf{D}_i(\alpha_i)^T j^b \\ \sum_{i=1}^{8} \mathbf{L}_i(\alpha_i)^T k^b + \sum_{i=1}^{8} \mathbf{D}_i(\alpha_i)^T k^b \end{bmatrix} \tag{2}$$

while the resultant torque vector is given by

$$\tau^b(\underline{\alpha}) := \sum_{i=1}^{8} \mathbf{r}_i \times \mathbf{L}_i(\alpha_i) + \sum_{i=1}^{8} \mathbf{r}_i \times \mathbf{D}_i(\alpha_i) \tag{3}$$

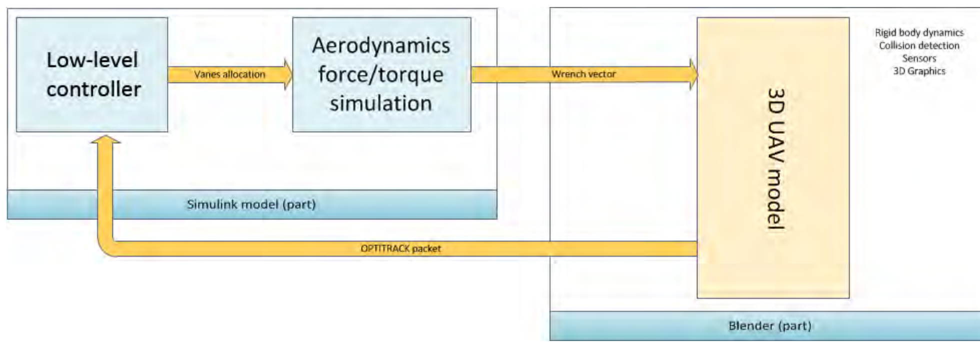
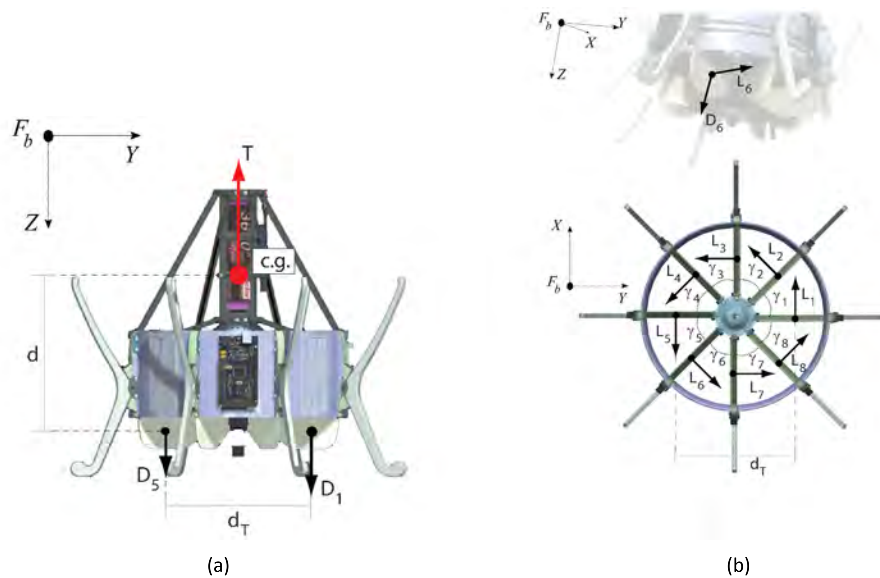


Figure 2: Splitting the aerial vehicle dynamical model between Simulink and Blender: rigid-body dynamics is implemented in Blender while force/ torque generation mechanisms are implemented in Simulink taking into account for the aerodynamical model.



(a) (b)

Figure 3: Force generation scheme in the Ducted-Fan MAV: (a) drag forces, (b) lift forces.

where, for each $i \in \{1, 2, ...8\}$,

$$\mathbf{r}_i := \begin{bmatrix} \frac{1}{2} d_T \sin(\gamma_i - \pi/4) \\ \frac{1}{2} d_T \cos(\gamma_i - \pi/4) \\ d \end{bmatrix}$$

denotes the point of application of each aerodynamic pair of lift and drag forces with respect to the center of gravity of the system which coincides precisely with the origin of the body fixed reference frame.

Hence, the model (2)-(3) is implemented in a Simulink block as input for the rigid-body dynamics (1) in Blender for the case of DFMAV.

2.2 Manipulator

The modelling of the Delta Robot, its joints and other parameters such as friction and mass is too complex for the Blender Game Environment to produce realistic results. In particular, the small mass of the manipulator's parts is too little to be reproduced in Blender, where each object can't have a mass smaller than 0.010 Kg.

Reduction to end-effector and task space to joints space transformations

These limitation have lead to modelling the manipulator only considering the end effector's task space, instead of modelling the delta robot's joints space. The end-effector can only translate in the task space, and rotate around its local z-axis, a constraint that is easily implemented in Blender, and a single control force vector applies to the end-effector to make it translate. Figure 4 shows a schematic model of the delta robot with indication of the torques and the corresponding end-effector force.

The control law for the manipulator computes the control action required to govern the joints in order to obtain a desired position / force to the end-effector. On the other side, the model implemented in the simulator only considers the task space of the robot. To this end all the control forces have to be translated from the joint space to the task space in order to derive the control inputs for the simulator model of the manipulator. Accordingly a Simulink block implements the inverse kinematics to estimate for the joint coordinates given an end-effector position and then the differential kinematics in order to translate forces from joint to task space and viceversa. Figure 5 shows a high-level overview of the components used.

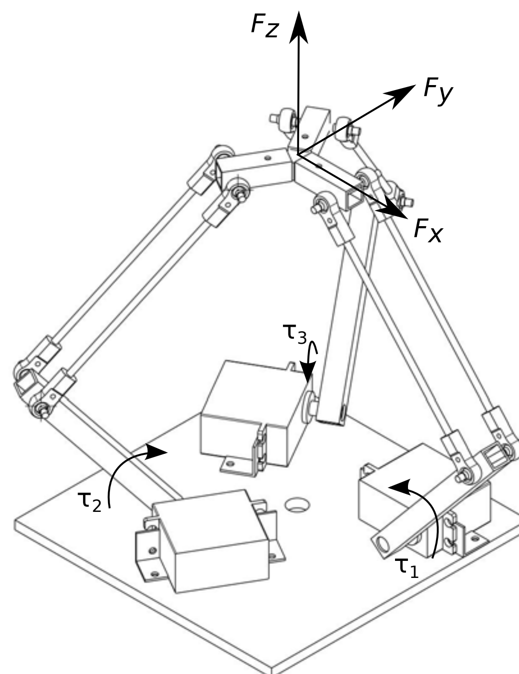


Figure 4: Forces at the base joints and forces at the end effector

Graphical modelling To have a more realistic look, the CAD-model of the manipulator can be shown between the UAV model and the end-effector. This manipulator is then represented as an image rather than a real kinematical model. From a graphical view point the applied image of the manipulator follows precisely the end effector and base position to appear similar to what the real manipulator would be.

2.3 Sensors

The simulation environment substitutes the UAV's sensors in order to give feedback both to the operator and to the low-level and high-level control laws. Force information computed during the interaction are also transmitted to the haptic devices used by the operator. Force sensors have to be implemented by defining a suitable compliant multi-body system as detailed in the following.

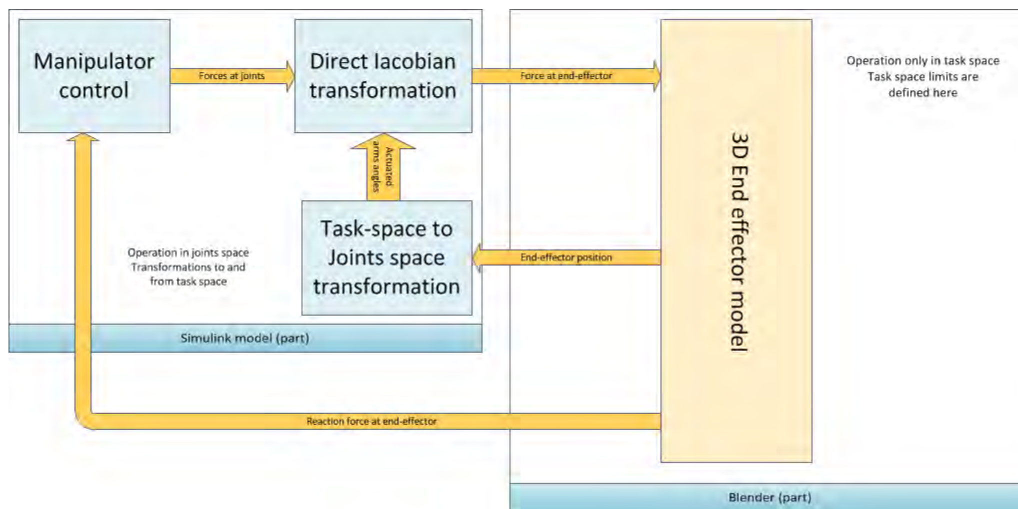


Figure 5: the Manipulator components in Simulink and Blender and integration with control

Inertial sensor

Information regarding the position of objects, can be easily retrieved in Blender. The MORSE framework provides out-of the box sensors that will retrieve attitude, location, speed and acceleration information, thus emulating both the IMU and Optitrack information. Only attitude and location are sent back to the control law in the Optitrack packet so as to provide the same amount of information used in free-flight indoor experiments. Additional speed information can be added to simulate IMU gyroscopic information, validating also the case in which an IMU is available on the vehicle.

Streaming camera output

Two camera sensors, provided by MORSE, are installed on the 3D model of the vehicle. Each sensor provide a continuous image stream, at a frequency which can be obtained as a fraction of the one used to send telemetry information such as Optitrack and IMU. Namely, attitude and location data are sent at 100Hz, while camera output is streamed at 10Hz or 20Hz. Cameras can be controlled in depth of field and focal length. Other non-streaming cameras can be used in the simulation environment providing different perspectives for piloting and monitoring purposes.

Force sensor

Force in Blender must be measured using virtual spring-dampers, since there is no way to retrieve the information directly from the physics engine. Where force information is needed, a virtual force sensor must be applied to the model, just as a real force sensor would have to be applied to the real UAV. Force sensors have been developed since they were not provided by the MORSE framework. The idea is to measure the deformation of a spring with known stiffness so that the force applied to it can be easily computed. For multi-dimensional impacts a suitable multi-body structure can be defined according to the above idea.

2.4 Environment modelling

Blender provides a very complete GUI that allows modelling of complex 3D scenarios. Meshes (3D objects shapes) can be imported from different standard CAD formats, and realistic textures can be applied to the surfaces. Complex objects, like the UAV with all the sensors and actuators, can be easily shared among different scenes.

Most of the 3D modelling is done using the 3D GUI editor, but it can also be fully programmed in Python, allowing run-time modifications of the environment or other parameters during the simulation.

Figure 6: our simulated flight arena

3 Software Integration

The simulation environment is designed to substitute parts of the real vehicle and its sensors, while maintaining other software and hardware components (like the low-level control and Human-Machine-Interface), that are exactly the same that are being used in the real flights. As such, the simulation environment must communicate with these components in the same way that the real vehicle does. What we describe here are the specifications needed in order to achieve this goal.

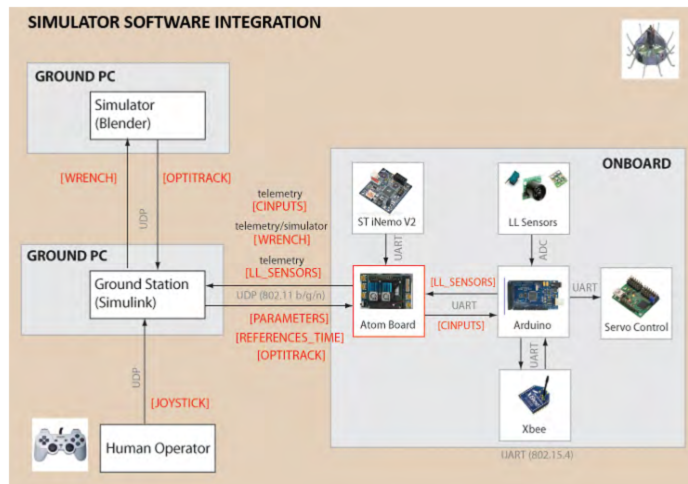


Figure 7: Components used in the simulation

The main integration points are with the low-level control, with the supervisory control and with other possible ROS nodes. Each component must communicate to the others using preferably UDP datagrams over the network. By the nature of this schema, it is possible to distribute the computation and the control laws over multiple computational units, up to achieving a true remote-controlled application over the internet.

3.1 Integration with low-level control

As the low-level control algorithms relay on Simulink, the same data packets commonly used for the laboratory Optitrack installation are used to export attitude and position values of the virtual UAV. This lead to a 1 to 1 replication of the physical testing environment that speeds up the development phase. Torques and forces from the environment and the control laws are applied to the rigid body and then new values for attitude and position are returned from the simulation in order to close the control loop.

3.2 Integration with the supervisory control

The simulator allows the operator to set a number of way-points in order for the high-level control law to compute a path for the vehicle. The environmental model in this situation should be similar to the one of the end-user application so that the operator is able also to employ the simulator to plan the trajectory for the real inspection. Simulator can also be used as a benchmark to have a 3D view of real-flight trajectories, so that the operator can evaluate the actual mission with respect to the one planned before.

3.3 Integration with ROS

Additional high-level functionalities can be integrated using an off-the-shelf middleware, and in particular ROS, able to provide a publish-subscribe communication paradigm. The integration with ROS happens by defining suitable ROS nodes having on one side a public ROS interface and, on the other, the ability to communicate over the UDP network with the simulator interface.

4 Programming reference

According to the architecture of the simulation environment, which comprises both components developed in Blender and others in Simulink, programming of the simulator must be carried out partially in Blender and partially in Simulink. As a general guideline, everything which has to do with rigid body dynamics, force measurement and other sensing is done in Blender, while integration with the low-level control, aerodynamic forces calculations, transformations from joints space to task space and viceversa is done in Simulink.

The Blender part can be further distinguished in MORSE-compliant software components, Blender-native Python code and Blender-GUI modelling.

The communication between Blender and Simulink is carried out with UDP packets, according to the protocols described in paragraph 4.4

Figure 8 shows the general layout of the different components between Simulink, the MORSE framework and Blender.

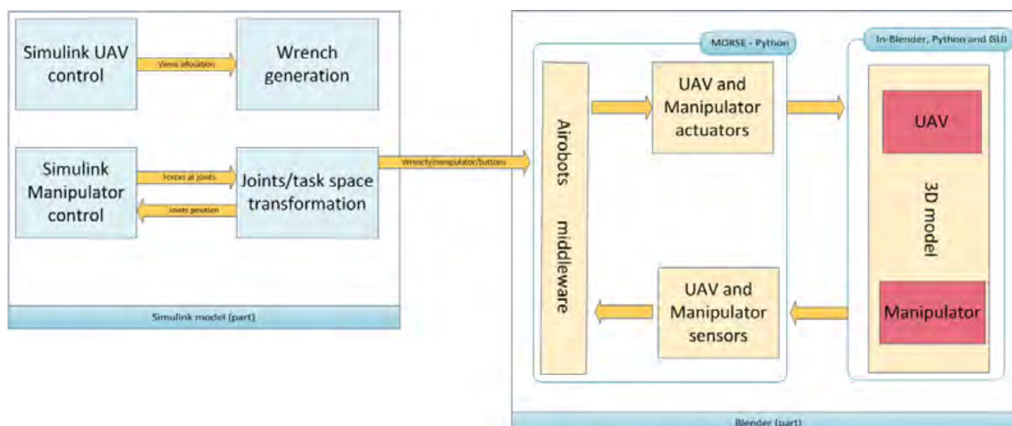


Figure 8: Layout of the components in Simulink, MORSE and Blender

The simulation loop of the Aerial vehicle and that of the manipulator are implemented in different components at each level, and communicate on separate UDP channels and protocols. The Airobots Middleware on the MORSE side is

a single component but dispatches the information independently to the appropriate manipulators and actuators.

Aerial Vehicle The aerodynamics calculation performed in Simulink is sento to the Airobots Middleware on the MORSE-Blender environment that parses the information and delivers it to Force/ Torque actuator. This is meant to apply forces and torques to the 3D object, so that the underlying physics engine can simulate the motion.

Feedback is obtained through the Attitude and Location sensor, then sent through the middleware back to the control law.

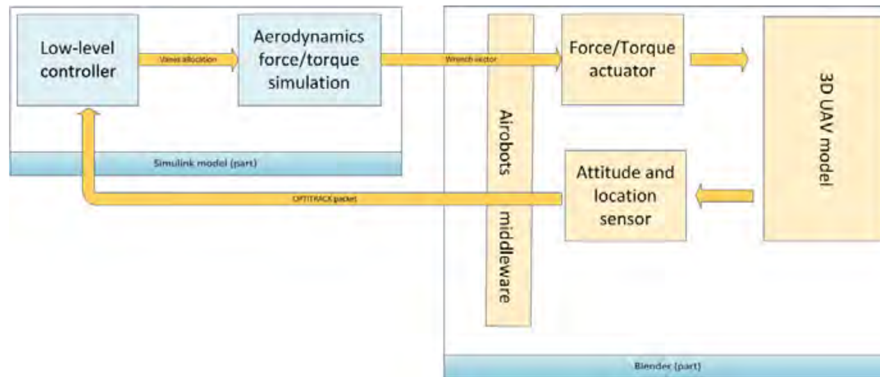


Figure 9: The UAV components in detail

Figure 9 shows the components involved in controlling the UAV at each level - Simulink, MORSE and Blender - and the flow of data.

Manipulator In a similar fashion, the control law of the manipulator is integrated in the Simulink model with blocks that reduce every interaction to simple rigid body dynamics and transform forces and position in the joints space to forces and positions in the task space. At the MORSE-Blender side, the same middleware applies forces to the 3D model of the end-effector by means of a force actuator, and reads information on the resulting force exerted at the end-effector and its position through two different sensors.

Figure 9 shows the components involved in controlling the Manipulator at each level - Simulink, MORSE and Blender - and the data flow.

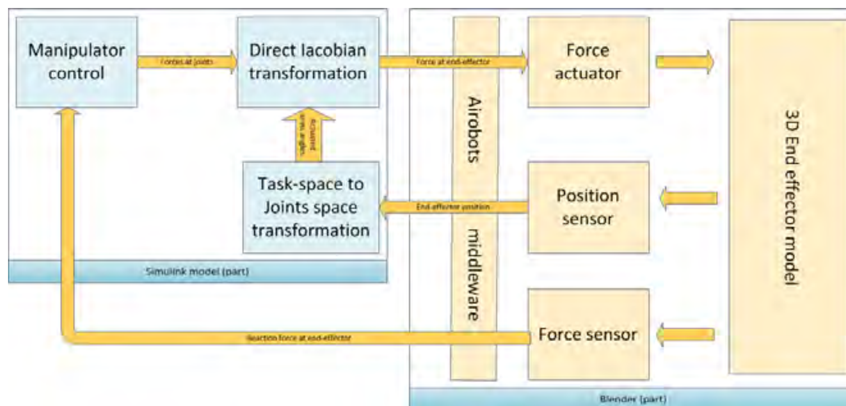


Figure 10: Manipulator components in detail

4.1 Simulink components

Simulink blocks specific to the simulator are used to set the gain parameters, to integrate the Simulator with the control law and to calculate the transformation from joints space to task space and viceversa. For mathematical details regarding

modeling and control of the manipulator and the aerial vehicle the reader is referred to Deliverable D4.1.

Simulation parameters

In the main GroundStation Simulink model, the parameters needed for the low-level controller can be set using a special configuration packet that contains several settings. The gain and w_c parameters can be used to tune the attitude and position controller.

Aerial vehicle Simulink integration

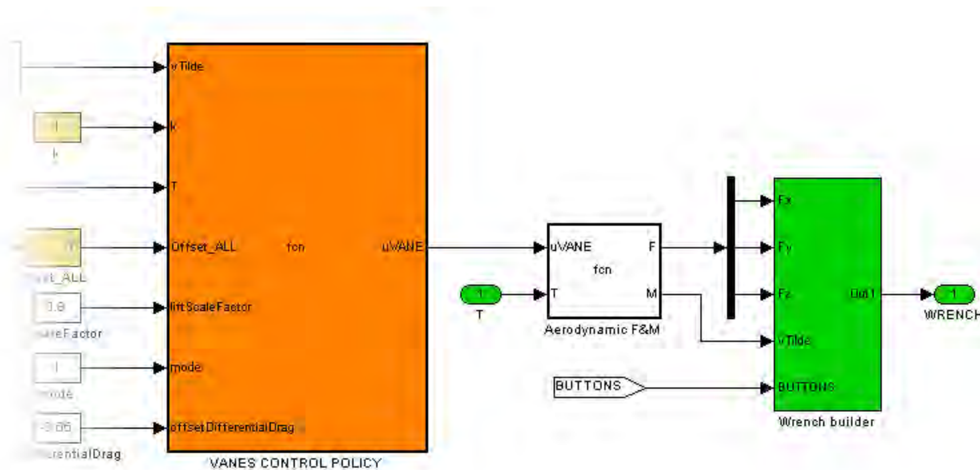


Figure 11: Portion of the Simulink model with the blocks for wrench calculation.

Wrench calculation The simulator is meant to replace the UAV at the lowest possible software level, therefore the calculation of the wrench vector - the forces and torques applied to the 3D model - starts from the UVANES allocation - see among others [1] and Deliverable D4.1 -, as explained in paragraph 2.1.

This calculation can be found in the Aerodynamic F&M block shown in figure 11.

Wrench builder This block builds the packet to send the wrench vector to the Blender simulation. Forces and torques are multiplied by 10^6 and then truncated as integers to respect protocol specifications. The Buttons field holds the map of the buttons of the joypad, used to control some aspects of the simulation, like way-point release or point-of-view switching.

See paragraph 4.4 for details on the protocol.

The built message is then sent to a RTWT output block that sends it to Blender via UDP.

Manipulator Simulink integration

The manipulator control is integrated in the same way. Since the control law operates in the joints space and the simulator operates in the task space, some blocks perform the necessary transformations of positions, velocities and forces.

Given the current position of the end-effector in the task space, the configuration of the joints is calculated using inverse kinematics [9]. The Jacobian operator is calculated and used to transform the linear velocity of the end effector into the rotational velocities of the joints. Configuration and velocities are feed into the control law, then the same Jacobian operator is used to transform the torques at the joints into the force at the end-effector. Figure 13 shows the portion of the Simulink model that implements this integration.

Observe that in Blender basically the end-effector is modeled as a rigid-body with constraints on the admissible motions deriving from the kinematics of the specific manipulator installed on the vehicle. This means that all forces and torques exchanged between the aerial vehicle and the manipulator have to be computed in Simulink. In particular, to

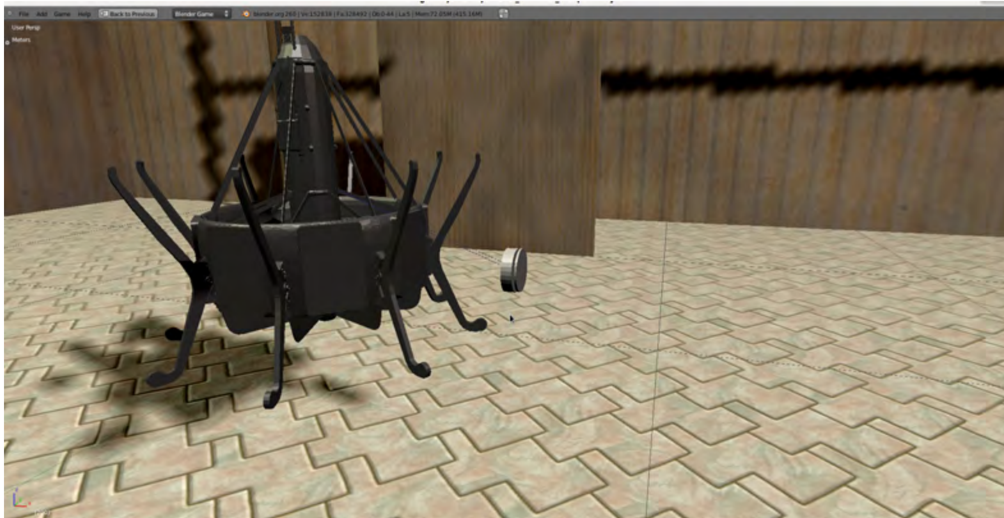


Figure 12: The simulated manipulator attached to the UAV

handle interaction with the environment in a proper way, it is of paramount importance to model the reaction forces applied to the basis of the manipulator, namely to the aerial vehicle (see also Deliverable 4.1 where a complete model of the system is proposed). This is done by taking into account for the current configuration of the manipulator and the torque control inputs applied at joints. With these information at hand the wrench vector (additional wrench in Figure 13) is computed and applied to the aerial system as a disturbance. This transformation as well as the Jacobian operator strictly depend on the physical model and parameters of the manipulator to be installed onboard. Thus by separating their computation from the visualization and rigid-body dynamics implemented Blender allows to have a simulator which can be easily expanded to describe different kinds of aerial service robots.

Jacobian calculation This block calculates the Jacobian operator, based on the position of the base joints and the knowledge of the kinematical model of the robot.

Direct Jacobian transformation This block computes the control forces at the end effector by means of the knowledge of the control forces at joints and using the Jacobian operator:

$$[Fx, Fy, Fz] := (J^T)^{-1} * \tau \tag{4}$$

Inverse Kinematics transformation This block transforms the relative position between the end-effector and the point where it is attached on the vehicle into joint coordinates for the robotic manipulator. The former derive from the Blender environment, the latter are used to compute all transformations in Simulink.

Inverse differential Kinematics transformation This block transforms the velocity of the end-effector, computed with respect to the point where the manipulator is actually attached, into velocities of the joints:

$$[\dot{q1}, \dot{q2}, \dot{q3}] := J^{-1} * [\dot{x}, \dot{y}, \dot{z}] \tag{5}$$

Additional Wrench This block computes the forces and torques that the motion of the manipulator applies to the basis, namely to the aerial robot, during its free motion and also during possible interaction with the environment.

Manipulator packet output This block builds the packet to send the force at the end effector to the Blender environment. Forces are multiplied by 10^6 and then truncated as integers.

See paragraph 4.4 for details on the protocol.

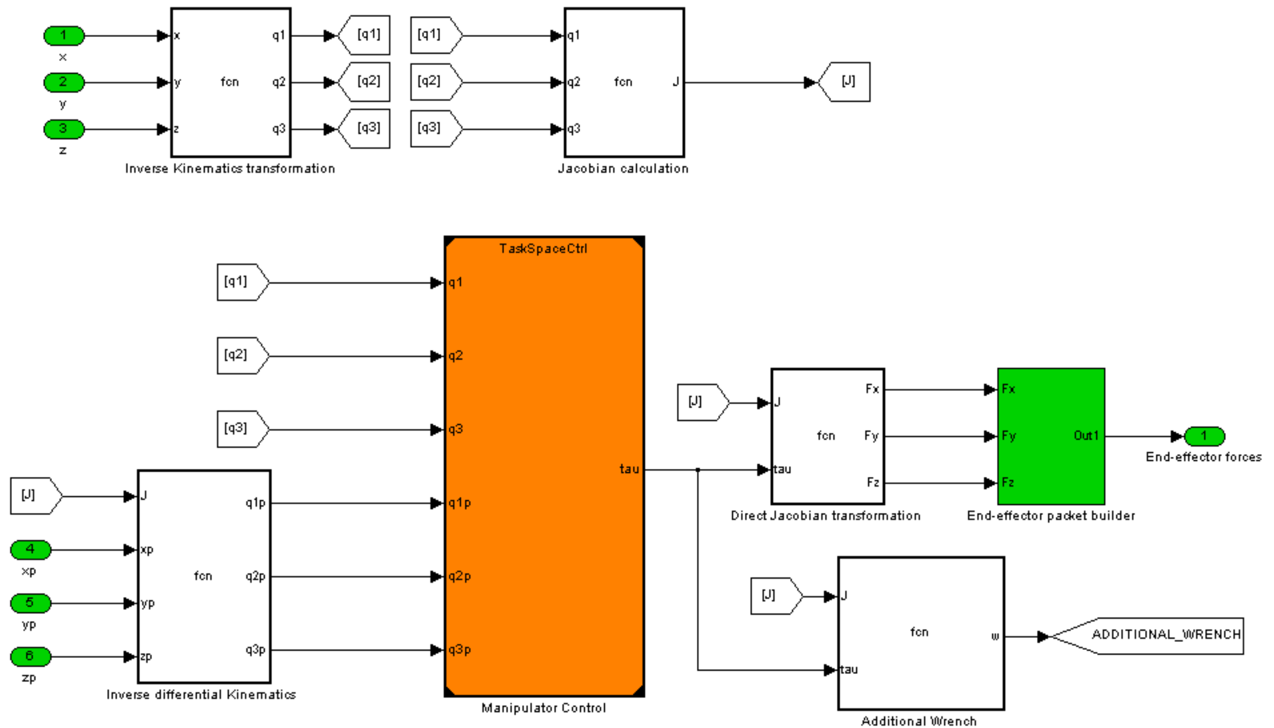


Figure 13: Portion of the Simulink model with the blocks for transformation from joints space to task space and viceversa.

The built message is then sent to a RTWT block that sends it to Blender via UDP. Note that this is a different packet on a different communication channel than the one used for the wrench packet.

4.2 MORSE architecture components

Each MORSE component is made of two files: a .blend file in which the Blender object is contained, and a Python module that implements the behavior of the component. The .blend file usually contains the mesh, even though some components do not feature a mesh. In those cases, a Blender empty object is used. Particular values, called ``Game properties'', that can be attached to a Blender object in the GUI, must be set in order to make the MORSE library interact correctly with the simulation engine. A robot base would for example have a "Robot_tag" property to tell MORSE that that object is to be considered as a robot. This connection is realized at runtime by MORSE init scripts. MORSE components are divided in categories according to their function

- Robots: a robot base must be the parent object of all attached sensors and actuators. Many robots can be deployed in the same scene. Usually a robot base has a mesh that models the physical robot.
- Sensors: provide data about the parent robot or the environment. These include pose sensors, cameras, accelerometers and so forth. The Blender object can be a mesh or an empty object.
- Actuators: affect the simulation by changing speed, applying forces, changing point of view, setting properties. Actuators usually have an empty Blender object.
- Modifiers: affect the output of a sensor. A modifier can be used to transform the output of a pose sensor from Euler angles to Quaternion, or to apply a noise to the measure.
- Middleware: define functions used to communicate with other nodes. Middleware implementations for connecting Blender with YARP or ROS are included in the standard MORSE distribution.

Components available out-of-the-box

Morse comes with many components, some of which have been used for this version of the simulator. In particular:

- Video camera sensor: the sensor emulates a single video camera by generating a series of RGBA images encoded as binary char arrays, with 4 bytes per pixel.
- Stereo Camera unit: the combination of two cameras mounted on an joint bar with constrained motion. This is used to simulate the two cameras mounted on the real UAV for stereo-vision. This component can provide the stereo information generated from the two camera images.
- Pose sensor: this reads the rotation of the UAV relative to the World Axes (fixed). The output is in yaw, pitch and roll values. Also, it gives the position of the robot relative to the World Axes origin.

Some new components have been developed for AIRobots. The Python code is external to the main Blender file, not embedded into it. It's mandatory that the source modules are in the same directory as the MORSE files, because the MORSE framework will look there to find them.

Figure 14 shows an overview of the Python classes, where details about methods and properties are omitted. Grayed classes are from the MORSE framework.

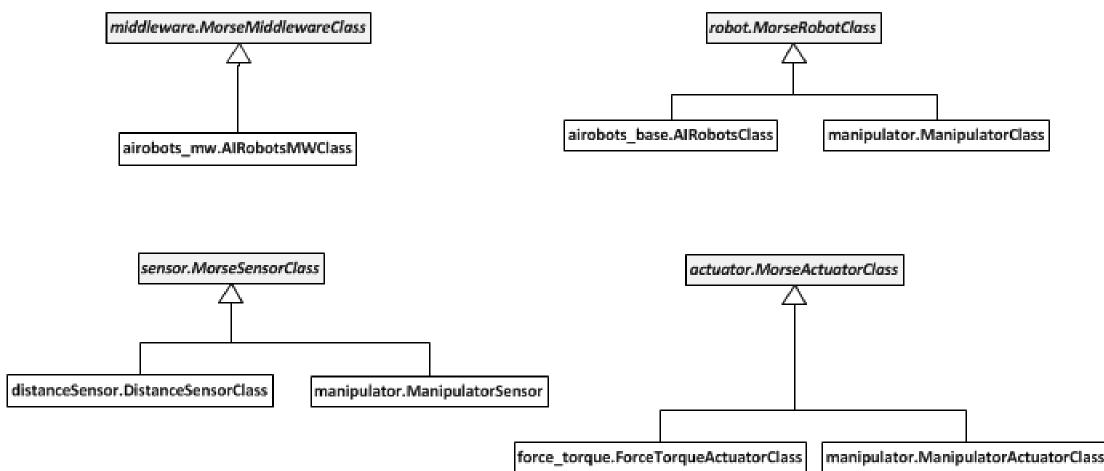


Figure 14: General class diagram.

Middleware

One of the main principles followed in designing this simulator, also accordingly with the MORSE framework, is that components such as actuators and sensors should be unaware of the ways their input is received and their output is sent. The simulation scenarios, robots and different components can then be reused in several deployment situations. Ideally, if the communication infrastructure were modified to use YARP, only the middleware should be substituted.

In our case, the middleware is slightly dependent on the application, because the messages received from MATLAB can specify both forces and torques applied to the robot by means of the ForceTorque Actuator, and other simulation-related information, as changing the point of view or resetting the simulation, so that some logic is required to correctly parse the protocol message.

The middleware is connected to specific sensors and actuators by means of a component_config.py file that is internal to Blender. This file specifies which middleware function must be invoked at each simulation loop for each sensor or actuator.

A `rot_matrix` is declared in the `AIRobotsMiddlewareClass` to represent the rotation of the Blender's world frame with respect to the frame expected in the control law. The main difference is that the Z axis points upwards in Blender and downwards in the control law.

A method is implemented for each communication channel

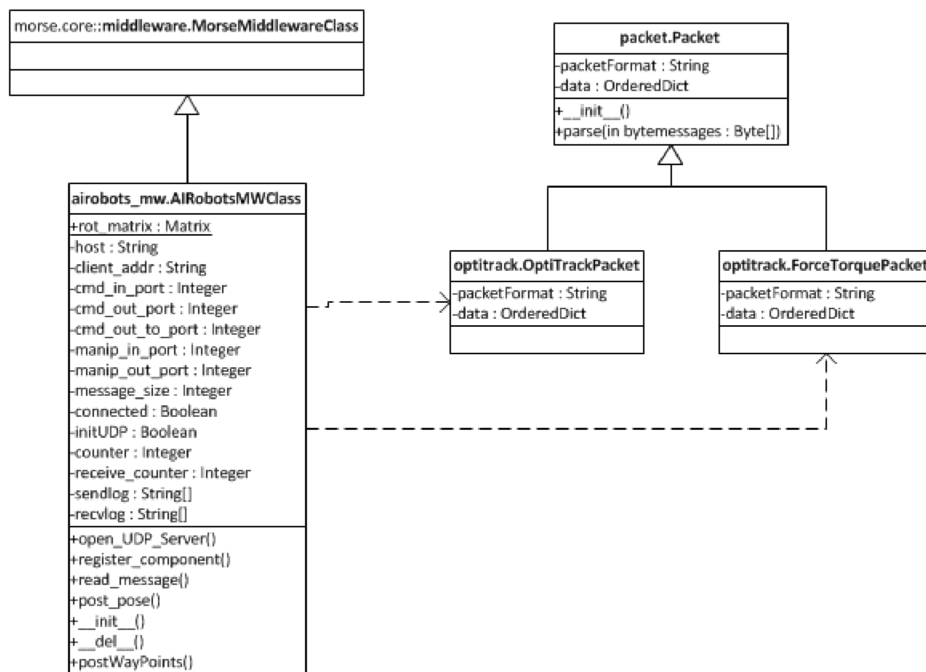


Figure 15: Middleware package classes.

- `+ read_message(component)`: reads the ForceTorquePacket and applies it to the corresponding actuator. Also manages buttons.
- `+ post_pose(component)`: posts the UAV location and attitude in an Optitrack packet
- `+ post_waypoints()`: publishes the list of waypoints once every 2 seconds
- `+ read_manipulator()`: reads the force at the end effector and applies it to the actuators
- `+ post_manipulator()`: reads the end-effector position and the force measured in the environment and posts them

Robots

The simulation environment specifies two robots: the Aerial Vehicle and the Manipulator. They act independently, with different sensors and actuators and are linked in the 3D interface with in-Blender constraints.

As MORSE components they are very simple classes, implementing only the default methods required by the framework, as shown in 16.

The AIRobotsClass implements a single additional method `releaseWaypoint()` that is used to release a waypoint in the 3D space.

Actuators

Actuators are the means through which the middleware sets properties in the 3D environment, such as applying a motion or setting a location. The actuator for the UAV applies forces and torques to the 3D object. The forces vector is rotated by `rot_matrix` before applying it.

The actuator for the manipulator applies only forces to the end-effector, since it can only translate and not rotate relative to its base.

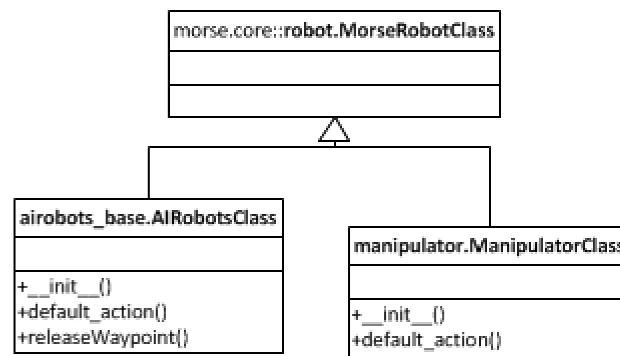


Figure 16: Class diagram of the robots

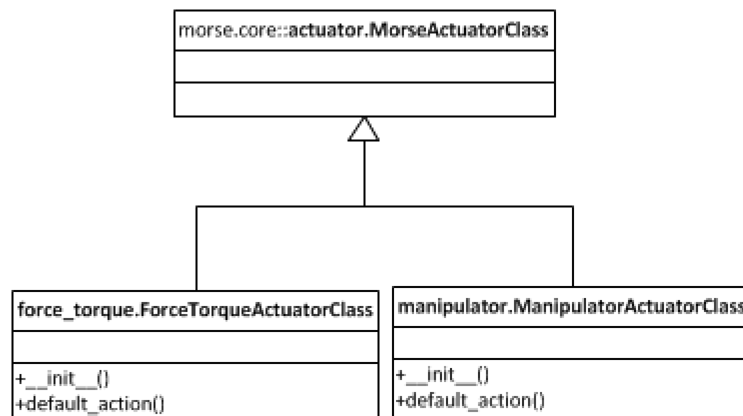


Figure 17: Class diagram of the actuators

Sensors

Sensors are the components through which the middleware retrieves information form the 3D model. For the UAV, a default MORSE pose sensor is used to retrieve the location and attitude of the UAV. For the manipulator a sensor that retrieves the end-effector position and the force applied has been built.

Force sensors Bullet, the physics engine that lies at the heart of Blender, does not provide a way to measure the force applied to an object directly, but it must be measured indirectly by means of a contact proxy. The end effector is modelled as two parallel discs, with a very rigid and spring-damper, that with a deformation that is irrelevant for the simulation purposes allows to measure the applied force. The same force is then transmitted to the UAV as a reaction force. When equilibrium is reached, the reaction force will equal the manipulator's control force, thus determining the end-effector position.

4.3 In-Blender programming

The main components inside the Blender 3D environments are meshes and constraints. Meshes are 3D objects, with properties such as shape, material - color, texture - weight, collision characteristics. Meshes have an origin and a relative reference frame, and can act as rigid bodies.

The 3D model of the Aerial vehicle is a complex hierarchy of meshes and other object, so that it can be physically realistic as a rigid body, have a simple collision model and be visually detailed. These three purposes are achieved using three different meshes:

- the main UAV mesh. It's the root of the hierarchy, a simple invisible cube whose origin corresponds to the center of mass of the UAV and that can be stretched to obtain realistic inertia simulation. Figure 19 shows the panel to edit such properties. The mass of the UAV is defined here;

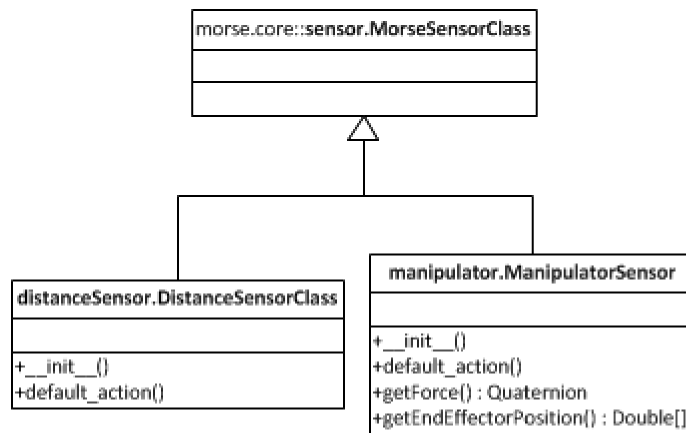


Figure 18: Class diagram of the sensors

- two invisibile octagonal meshes, one aligned with the top of the UAV legs and another one aligned with the bottom of the UAV legs. These octagons are used as collision boundaries during the simulation. They are very simple in shape and thus allow the realtime simulation to compute collisions easily.
- a visible mesh, created importing the CAD model of the real UAV.

Figure 19: Physical properties panel for the root object

The 3D model of the manipulator is reduced to the end-effector, modelled as a small disc and constrained to the UAV's mesh.

Binding components to MORSE

Binding components to the 3D scene is done in the component_config.py file that is embedded as a text block in Blender. This file binds sensors and actuators to specific methods of middlewares, one-to-one:

- the 3D object GPS sensor is bound to the middleware's post_pose method

- the 3D empty object ForceTorqueActuator is bound to the middleware's read_message method
- the 3D empty object EndEffectorActuator is bound to the middleware's read_manipulator method
- the 3D empty object ManipulatorSensor is bound to the middleware's post_manipulator method

4.4 Protocols

The different components can communicate using the well-known AIRobots protocols also used during the Integration Weeks.

[JOYSTICK] packet – 16 bit fields

Length	Type (1)	Axis A [0-1024]	Axis B [0-1024]	Axis C [0-1024]	Axis D [0-1024]	Buttons [0-16132]	Timestamp [1E-4 sec]	CRC

[WRENCH] packet – 32 bit fields

Length	Type (565)	FX [1E-6 N]	FY [1E-6 N]	FZ [1E-6 N]	TauX [1E-6 Nm]	TauY [1E-6 Nm]	TauZ [1E-6 Nm]	Buttons [0 16132]

Timestamp [1E-4 sec]	CRC

[CINPUTS] packet – 16 bit fields

Length	Type (3)	Servo1 [0-4096]	Servo2 [0-4096]	Servo3 [0-4096]	Servo4 [0-4096]	Servo5 [0-4096]	Servo6 [0-4096]	Servo7 [0-4096]	Servo8 [0-4096]

Servo9 [0-4096]	Timestamp [1E-4 sec]	CRC

[OMEGA] packet – 16 bit fields

Length	Type (45)	Vx [0-1024]	Vy [0-1024]	Vz [0-1024]	Timestamp [1E-4 sec]	CRC

[MINPUTS] packet – 16 bit fields

Length	Type (71)	Joint1 [uint16]	Joint2 [uint16]	Joint3 [uint16]	Joint4 [uint16]	Timestam p [1E-4 sec]	CRC

Figure 20: The messages implemented for the simulator.

Here it is introduced the new ``Wrench'' data packet, specifically designed to control the simulated UAV.

5 Installation guide

This chapter will guide you through the configuration and deployment of the simulation environment.

5.1 Installation overview

In order to use the simulation environment, several components must be installed and deployed on at least two different machines:

- Blender Host - A Linux machine to run the 3D simulation engine. On this machine you will have to install Blender, MORSE and the AIRobots python and .blend files.

  Note that, due to the requirements of MORSE, it is currently not possible to use the 3D simulator under Windows.

  Also note that OpenGL 3D acceleration is required for the 3D simulator. Tests conducted so far have shown that the software can't run smoothly in a Virtual Machine. WUBI install will work fine. The reference platform is Ubuntu 64-bit, but other distributions should be fine. In this Installation Guide, this machine will be called Linux Box.

- Control Host - A Windows XP 32-bit machine or virtual machine. On this machine you will install Matlab and Simulink, and the AIRobots control algorithms.

  Here you will also install other devices, such as the joypad or the haptic interface, and the software that produces the command stream. Since all the communication between components is carried out through UDP, these tools can be deployed on a different Windows machine, or if you develop your own device parser, you can use the platform you prefer. For simplicity, this installation guide will use a single Windows XP (Virtual) Machine. In this Installation Guide, this machine will be called Windows Box.

Install locations

Several locations referring to directories or hosts are needed to properly configure the software. Hosts will vary depending on your installations, while it is recommended that for ports and directories you use the values specified here.

Hosts

- `$BLENDER_HOST`: the IP address of the Linux Box, on which Blender is running.
- `$GROUND_STATION_HOST`: the IP address of the Windows Box, on which the Simulink Model is running.

Linux Box directories

- `$AIROBOTS_HOME`: the home directory of the airobots user. Use `/home/airobots`.
- `$BLENDER_HOME`: the directory where Blender is installed. Use `/usr/local/blender`.
- `$MORSE_BLENDER`: where MORSE can find the Blender executable. Set to `/usr/local/blender/blender`.
- `$MORSE_ROOT`: the directory where MORSE is installed. Use `/usr/local`.
- `$OPT_DIR`: the directory where the real Blender and MORSE packages are installed, for the purpouse of easily updating the software. Use `/opt`.
- `MORSE_INSTALL_LIB`: the directory where the MORSE libraries will be installed. Due to a problem in the installation with MORSE 0.5 and Python 3.2 it is necessary to set it as `/usr/local/lib/python3/dist-packages`. Note that it is python3, and **not** python3.2.

Windows Box directories

- `%AIROBOTS_HOME%`: the directory where the AIRobots components are installed. Set to `C:\AIRobots`.
- `%GS_HOME%`: the directory where the Simulink model and related Matlab files are located. Set to `C:\AIRobots\GS`

5.2 Install the Linux Box

Install Blender 2.61

The simulator has been tested with version 2.61 on Linux 64bit. Note that, at the time of writing, the current version of MORSE does not support newer versions of Blender.

Blender 2.61 Linux 64 bit can be downloaded from here:

`http://download.blender.org/release/Blender2.61/blender-2.61-linux-glibc27-x86_64.tar.bz2`

For other versions check the Blender website.

To unpack, open a shell in the directory where you saved the downloaded file and execute:

```
sudo tar -xf blender-2.61-linux-glibc27-x86_64.tar.bz2 -C /opt
```

Unpacking will create a subdirectory named `blender-2.61-linux-glibc27-x86_64`
Link this installation directory to the `$BLENDER_HOME` directory:

```
sudo ln -sf /opt/blender-2.61-linux-glibc27-x86_64/usr/local/blender
```

Install MORSE 0.5

NOTE If you check the MORSE website, there is an automated install procedure that uses robopkg. However, we have found that the most reliable and controllable way to install is by following the manual installation.

Set installation directory Using your favorite editor, edit $AIROBOTS_HOME/.bashrc and add an environment variable pointing to $MORSE_ROOT:

```
export MORSE_ROOT=/usr/local
export MORSE_INSTALL_LIB=/usr/local/lib/python3/dist-packages
export PYTHONPATH=$MORSE_INSTALL_LIB
export MORSE_BLENDER=/usr/local/blender/blender
```

When you have edited this file, open a new command shell or source .bashrc to make the setting effective.

Installation prerequisites Using the Ubuntu software center, make sure that you have all the following packages installed:

- g++
- python 3.2 (it's not necessary to remove python2.x)
- python3.2-dev
- cmake
- ccmake
- python-sphinx
- git

Then go to `http://www.openrobots.org/morse/doc/0.5/user/installation.html` and follow the instructions for manual installation.

Note that a typical installation problem occurs with python3.2: MORSE will install in `/usr/local/lib/python3` instead of `/usr/local/lib/python3.2`, but if you have correctly setup the variables in your -bashrc it should be fixed.

Test At this point you should be able to test that the MORSE installation works by typing
`morse check`
If this works, start MORSE by typing
`morse`
It will show a default scene. Maximize the window, move your cursor over the scene and type `P`. Try to move around the scene using the command described on the left of the screen. If this works smoothly, your HW setup should be ok for the simulator environment.

Install the AIRobots software and simulation file

Download the airobots-py.tgz from the project's website and unpack it in `$MORSE_INSTALL_LIB`:

```
sudo tar -xzf airobots-py.tgz -C
$MORSE_INSTALL_LIB
```

Now you must edit a couple of configuration parameters in the airobots_mw.py file.
Open `$MORSE_INSTALL_LIB/airobots/middleware/airobots_mw.py`, look for `_self.host` and `_self.client_addr` and modify them accordingly. `_self.host` must be set to `$BLENDER_HOST` and `_self.client_addr` must be set to `$GS_HOST`.
Download the airobots.blend file from the projects website and open it with MORSE:
`morseexec airobots.blend`
Your Linux box should be setup now.

5.3 Install the Windows Box

Prepare to run Real-Time Windows Target code

This software is installed on the Windows Box.

In order to run the Ground Station control code you must have Matlab installed with Simulink and the ``Real-Time Windows Target toolbox''. More information on this toolbox can be found here: `http://www.mathworks.it/products/rtwt/`

When you've succesfully installed Matlab with the Toolbox, you must install the RTWIN kernel by executing this command in the Matlab command window:

```
rtwintgt -install
```

and then press Y at the question:

```
You are going to install the Real-Time Windows Target kernel.
Do you want to proceed? [y] :
```

If you had a previous version of the real-time kernel, this message should appear:

```
There is a different version of the Real-Time Windows Target kernel installed.
Do you want to update to the current version? [y] :
```

You can then verify the correct installation of the kernel by typing:

```
rtwho
```

that will print the current version of the kernel.

Be advised that uninstalling Matlab will not uninstall the real-time kernel. This must be done by typing:

```
rtwintgt -uninstall
```

in the system command line or in the Matlab command window.

For further reference, more installation instruction are available at this page:
`http://www.mathworks.it/help/toolbox/rtwin/ug/f19807.html`.

Install the Ground Station

This software is installed on the Windows Box.

Create the `$GS_HOME` directory where you'll keep all the Ground Station files and unpack the contents of the airobots-gs.zip files. You will have some .mdl files and some .m files.

The model will have some I/O Blocks that may need configuration for the computer they're running on. If you have trouble building the model (ctrl-b) then you must open each I/O block, configuring and creating the specific ``board'' that is used.

Remember that to start the Simulink model (after a successful build), you'll have to first connect Simulink to the ``Windows'' target by using the appropriate button in the toolbar and then actually start the model by pressing the ``play'' button on the toolbar.

Install the JoyPad software

Unpack the contents of airobots-joypad.zip in `%AIROBOTS_HOME%`.

Start `%AIROBOTS_HOME%\joypad\bin\debug\joypad.exe`. In the drop-down list you should see your joypad device listed. If not, you might need to install some 3rd-party driver to make it run.

Configure the Joypad controls

Refer to paragraph 6.3 to check your joypad buttons configuration. Also check that the left and right sticks are configured as analog and not digital.

6 User manual

6.1 Starting the environment

Follow these steps to start the simulation environment. The names ``Linux Box'' and ``Windows Box'' and all directory names refer to the machines where you have installed the software, as explained in the Installation Guide. For troubleshooting, refer to the Installation Guide.

Start the 3D Environment

On your Linux Box, at a command prompt type

`morse airobots.blend`

This should start your scene and all middleware sockets automatically.

Start the Joypad parser

On your Windows Box, connect your USB joypad device and start `%AIROBOTS_HOME%\joypad\bin\debug\joypad.exe`. In the drop-down list select your joypad device. If necessary, select the ``Settings'' menu and assign axes as appropriate.

Type your `$GROUND_STATION_HOST` address in the ``Send UDP packets to...'' field (even 127.0.0.1 will be fine unless you move the joypadParser to a different Windows machine) and check that ``Port'' is set to 9010. Then click ``Connect''. The application should now begin to transmit data and show a real-time packet values log in the bottom-left textbox.

Try to move the joypad's sticks and click some buttons on it, to check the correct operation of the joypad and the application by monitoring the variations of the log values. If you don't see changes while moving the sticks or clicking the buttons, check your joypad configuration in the Game Devices tool from the Windows Control Panel. Also make sure that the sticks are set as ``Analog'' and not ``Digital'': you should see the numbers in the scrolling window change continuously from 512 down to 0 and up to 1024 according to how far you move the left or right stick.

Please note that for correct operation, you will need a device with at least 4 axis and 10/12 buttons.

Start the Simulink Environment

On your Windows Box, browse to `%AIROBOTS_HOME%\GS` and open GroundStation.mdl.

Once it is open, go to the main Matlab window and run `initialization.m` from the current working directory. This will configure all the necessary parameters needed to compile the model.

Switch to the Simulink window. If this is the first time you run the simulation, you may need to re-configure the I/O blocks in the simulink model to fit your configuration (and your Matlab/Simulink installation). Refer to the next section (6.2) to learn how to configure the address of your Blender Host.

Before starting the simulation and executing the real-time model, a full build is necessary by using the keyboard shortcut ``Ctrl-B'' (or the ``Tools/Real-Time Workshop/Build Model'' command) to generate the code needed by the Real-Time simulation to run on your machine. When compilation is done, locate the ``Connect to Target'' button and click it. Then click the adjacent ``Play'' button to start the simulation.

6.2 Parameter configuration

When starting the model for the first time, additional I/O configuration must be performed to install the correct Simulink UDP boards into the current simulation environment. Locate the I/O blocks named ``Stream Input'' or ``Stream Output'' and ``Packet Input'' or ``Packet Output''. Double clicking on them should bring up a messagebox stating:

```
This block references board ``Standard devices [...]''
which is not on the installed boards list for this computer.
Do you want to add it to the list?
```

Confirm by pressing ``YES''. This should install the current board on your computer to enable communication through the selected port.

Even if you've used our standard UDP port numbers, it is necessary to enter the ``board setup'' panel by clicking on the homonymous button. It is important to set the correct remote (IP) address with respect to your current virtual machine and network configuration. The port numbers should be fine if you've used the default values. If not, it's necessary to change them accordingly.

When each of these I/O blocks are correctly configured, the model should be ready to be built (Ctrl-B).

Configuring the AIRobots middleware

Hosts are defined in the airobots_mw.py that is located in

`$MORSE_INSTALL_LIB/airobots/middleware/airobots_mw.py`

Check the definitions at the top of the class and update the hosts' addresses, as explained in 5.2.

6.3 Running the simulation

With the low-level controller up and running as explained in 6.1 and the Joypad Parser transmitting data as explained in 6.1, move to your Linux Box. In the Blender window, which has appeared upon starting MORSE, **bring the mouse cursor over the 3D scene** then press `P` on your Linux Box keyboard. It is necessary that the Blender window has focus and that the mouse cursor is over the 3D scene to start the simulation.

Alternatively, you can start the 3D environment by typing `morseexec airobots.blend` but you might get a smaller screen resolution.

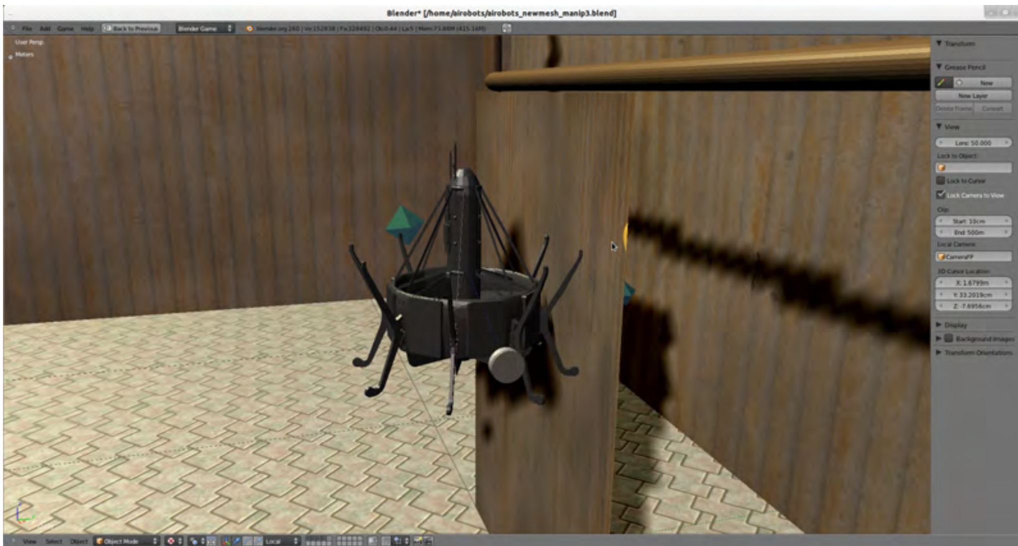


Figure 21: The simulated UAV in contact with a mockup wall

Joypad Buttons reference

Joypad buttons are used to control different functions of the UAV and some aspects of the simulation. The most common button number mappings are shown in figure 22.

Check the exact mapping of you joypad then refer to table 1.

Be advised that your button numbers may be different and the button schema shown in the table cannot be changed. Joypad buttons are expressed using a bitmap that is checked against these values by the low-level controller.

Flying the UAV

The joypad is used to send commands to the UAV, and to perform operations during the flight. At the start of the Blender Game Engine, the UAV should be on the floor and landed (with its virtual propeller switched off).

Takeoff To perform takeoff operation, it's advisable to switch on the propeller (using button 4) first, and then activate the takeoff command (using both buttons 6 and 8). The UAV should begin to leave the ground and gain altitude until the buttons are released. The controller now enters the free-flight mode and the UAV will hover maintaining the current position.

Flight During flight, the operator can move the position references of the controller by acting on the joypad's analog sticks. It's also possible to get in contact with the environment and perform manipulation using the haptic device.

Landing To terminate the flight, it's possible to use the landing command by keeping pressed the button 5. This will cut down the thrust of the UAV to make a soft landing. Upon releasing the button, the virtual propeller will stop and the UAV will enter a ``landed'' or ``off'' state.

Figure 22: Two common buttons mapping

Joypad btn combination	Button bitmap code	Action
1	1	SWITCH CONTROL TO AUTOMATIC MODE (keep pressed)
2	2	RESET REFERENCES
3	4	
4	8	START ENGINE
5	16	LANDING
6	32	GLOBAL RESET
5+6	48	LANDING RESET
7	64	USED AS SIMULATION MODIFIER
7+1	65	SWITCH TO FRONT CAMERA (simulator only)
7+2	66	DROP WAYPOINT (simulator only)
7+3	68	SWITCH TO AMBIENT CAMERA (simulator only)
5+7	80	STOP ENGINE
8	128	
6+8	160	TAKE OFF
9	256	IMU RESET
10	512	
11	1024	
12	2048	

Table 1: Buttons and corresponding simulation commands.

Changing the POV

Using the HAT-POV buttons of your joypad (usually the digital arrows on the left hand side of the joypad) you can change the ``Point Of View'' of the simulation, that is to say the camera through which you are looking while running the flight. If this doesn't work, check that your joyapd buttons are configured accordingly to 1.

6.4 Releasing waypoints

Fly the UAV to the desired location, press the ``Simulation modifier'' button (7) and button 2 simultaneously. A pop-up window will appear to allow the editing of some properties for the newly created way-point. The simulation will not stop in the meantime, therefore it's recommended that you release way-points only when hovering.

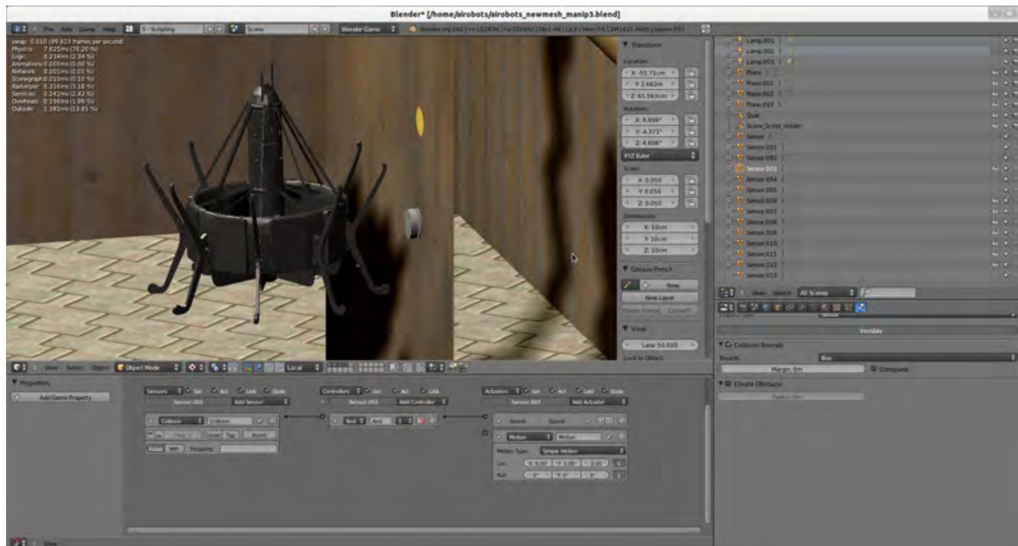


Figure 23: The manipulator perform actions on the mockup wall

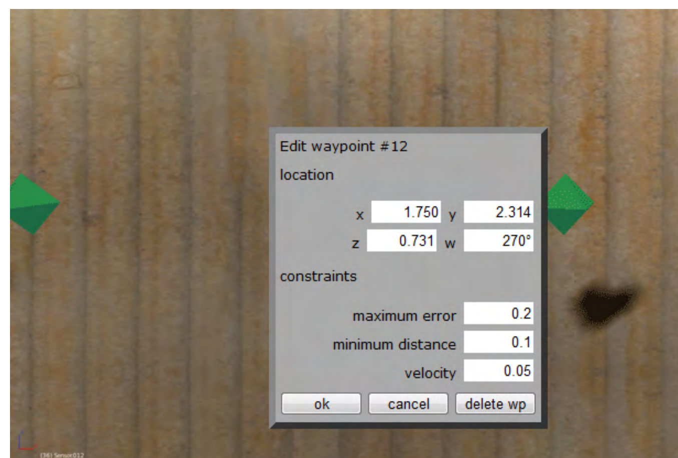


Figure 24: Waypoint editing window

References

[1] K.A. Bordignon. *Constrained Control Allocation for Systems with Redundant Control Effectors*. PhD. Thesis, Virginia Polytechnic Institute and State University, 1996.

[2] H. Bruyninckx. Blender for robotics and robotics for blender.

[3] H. Bruyninckx. Open robot control software: the orocos project. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Seoul, Korea, 2001.

[4] K. Buys, T. De Laet, R. Smits, and H. Bruyninckx. Blender for robotics: Integration into the leuven paradigm for robot task specification and human motion estimation. In Noriaki Ando, Stephen Balakirsky, Thomas Hemker, Monica Reggiani, and Oskar von Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 6472 of *Lecture Notes in Computer Science*, pages 15--25. Springer Berlin / Heidelberg, 2010.

[5] W.E. Dixon, D. Moses, I.D. Walker, and D.M. Dawson. A simulink-based robotic toolkit for simulation and control of the puma 560 robot manipulator. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Maui, HI , USA, 2001.

[6] G. Echeverria, N. Lassabe, A. Degroote, and S. Lamignan. Modular open robots simulation engine: Morse. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Shanghai, China, 2011.

[7] E.N. Johonson and S. Mishra. Flight simulation for the development of an experimental uav. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit*, Monterey, California, 2002.

[8] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004.

[9] R.M. Murray. *A mathematical introduction to robotic manipulation*. CRC Press, 1994.

[10] 3D Robot Arm website. http://sourceforge.net/projects/srom/.

[11] Blender website. http://www.blender.org/.

[12] Gazebo Simulator website. http://gazebosim.org/.

[13] Microsoft Robotics Development Studio website. http://www.microsoft.com/robotics/default.aspx.

[14] MORSE website. http://www.openrobots.org/wiki/morse/.

[15] Robosim website. http://robotics.ee.uwa.edu.au/robosim/.

[16] Simbad website. http://simbad.sourceforge.net/.

[17] Virtual Robot Simulator website. http://robotica.isa.upv.es/virtualrobot/.

[18] Webots website. http://www.cyberbotics.com/overview.