ISBN: 978-981-18-5852-9
WCSE 2022 Spring Event: 2022 9th International Conference on Industrial Engineering and Applications
doi: 10.18178/wcse.2022.04.056

Efficient Crawler Robot Modelling in Gazebo: A Tutorial

Alexandra Dobrokvashina [1], Roman Lavrenov [1], Yang Bai [2], Mikhail Svinin [2] and Evgeni Magid [1]

[1] Intelligent Robotics Department Kazan Federal University
Kazan, Russia

[2] Information Science and Engineering Department Ritsumeikan University, Kyoto, Japan

Abstract. Simulation and virtual experiments are important stages of robotics research. Robotics simulators allow testing new hypothesis and algorithms without taking risks to destroy valuable hardware. Therefore, most of popular robots are sup-plied with simulation models from their makers or associated research groups. This tutorial paper describes a process of creating of a robot model for the Gazebo simulator starting from the model construction and physics setup, and ending up with sensors and Robot Operating System based control integration. The process is illustrated with an example of a step-by-step modelling of a crawler-type robot Servosila Engineer supplied with an open-source code via a public Gitlab repository.

Keywords: modelling, simulation, Gazebo, ROS, tutorial

1. Introduction

Nowadays a number of robotic applications, their hardware and software complexity increase every single day, which allows to automate dangerous and repetitive processes [1]. Robots plays important roles in manufacturing [2], entertainment [3] and service [4], search and rescue [5], transportation [6], human-robot interaction [7], medicine [8] and healthcare [9].

Prior to integration of new approaches and algorithms into control systems of real robots, typically, they are tested in simulators. Virtual experiments in simulators became a fundamental part of research activities since easy created and reproducible complicated physical setups for testing reduce valuable time and resource spendings. The usefulness of simulation became the reason for companies to create simulation models alongside with real robots. Unfortunately, not all manufacturers provide a proper simulation model for their robots, if any, which forces researchers to create such models on their own.

This paper presents a tutorial on creating a simulation model of a real (existing) robot. It contains information about modelling parts of the robot, shows different ways to cope with collision meshes problem, explains a procedure of controllers and sensors integration. The model was constructed in the Gazebo simulation environment and employs Robot Operating System (ROS, [10]) for control purposes. All modelling steps are illustrated using our project of Servosila Engineer modelling [11], [12], which is a crawler mobile robot (Fig. 1) produced by Russian company Servosila [13].

2. Instruments

This article presents a step-by-step simulation process using ROS/Gazebo environment. ROS is a broad set of tools and libraries packed as a framework that is used for robot application, both for real robots and simulations, which are widely used by leading robotics companies, e.g., such as PAL robotics [14] or Robotis [15]. Gazebo is a robot simulator, integrated with ROS [16]. A vast majority of robot simulation models and plugins with ROS were created for the Gazebo simulator. RViz is used together with the Gazebo as a ROS visualizer for data that a robot receives from its sensors [17].

To work with a 3D model, we recommend using Blender software [18]. It is a free 3D computer graphics software, which is used for modelling, animation, and computer games. It appears to be quite popular and useful for scientific research, visualization, and modelling [19].

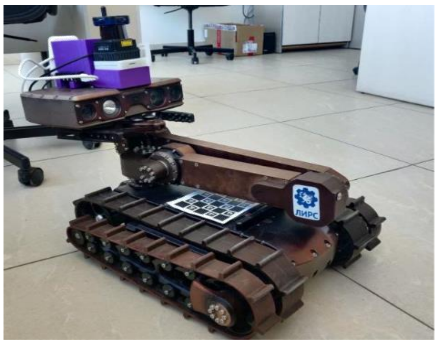
Fig. 1: Servosila Engineer robot at Laboratory of Intelligent Robotic Systems, Intelligent Robotics Department, Institute of Information Technology and Intelligent Systems, Kazan Federal University.

3. Creating a robot model

Few steps should be done before creating a robot model. A physically realistic model construction requires reliable data about the robot, including dimensions of robot links and their weights. Another important element for a robot description are default hardware and software limits, e.g., joint limits provide information about a workspace of a manipulator in real life.

3.1. Visual meshes

The first step is creating a reliable visual 3D model of a robot. Often, a CAD model of the robot could be obtained from a manufacturer as these models are used at robot design and production stages. Such model could be used as *visual meshes*. Otherwise, it should be created manually, which is a time-consuming procedure that requires experience and execution of a full-stack modelling process. Physical accuracy is ensured solely by a good CAD model and documentation (in the first case) or thorough measurements (in the second). For multiple reasons, it could be discovered that a manufacturers' CAD model does not precisely correspond to a real robot, and it is a responsibility of a modelling designer to verify measurements and update the CAD model accordingly.

Modelling could be done using any popular 3D engine, such as Maya [20], Blender [21], 3DsMax [22] etc. With some efforts, using an existing software, a model could be transferred from one file format (associated with a particular file extension) to another.

3.2. Collision meshes

Calculating physics of a visual model using only its geometry could be quite efficient. For this reason, simulators require additional meshes for every link of a robot, called a *collision mesh*. It is a mesh that is maximally simplified relatively to a visual mesh. There are two methods to create the collision mesh: *generating* models from visual meshes with automatic tools or *creating* models manually.

Automatic generation of collision meshes is an easy and fast solution. It suits for research teams that do not have a qualified 3D modelling specialist or are severely limited in time. There exist a large variety of graphical applications that provide users an ability of an automatic polygon decimation. In our case, Blender open-source solution was employed. *Decimate* function is released in Blender as one of available modifiers. First, the model is imported using *File-Import-(type of file with your model)* tab. Then, if the model is complicated and contains multiple parts, it is recommended to decide which parts could be deleted (for example, small-size pins, insignificant elements of a decor or inner elements). Each remaining part of the model should be supplied with a corresponding *Decimate* modifier. Modifiers appear in a right menu shown in the Fig. 2. Using parameters of this modifier the model could be significantly simplified. Other instruments that could be helpful for this task, are *ProOptimizer* modifier in 3DsMax or *Mesh-Reduce* option in Maya.

The second approach for collision meshes construction is a **manual creation** of required models. This option requires some expertise in 3D modelling. Creating collision meshes in most cases means covering a visual model with a new mesh while excluding small details and keeping only main geometry of objects. The expertise and experience are important in order to decide which details of the original model could be omitted.

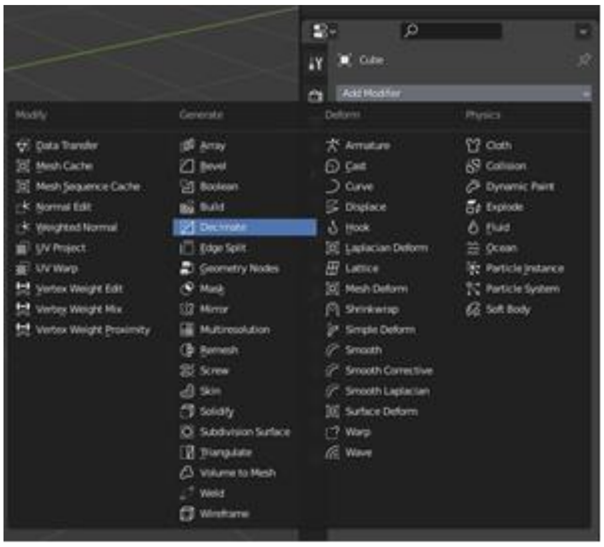

Fig. 2: List of modifier options in Blender.

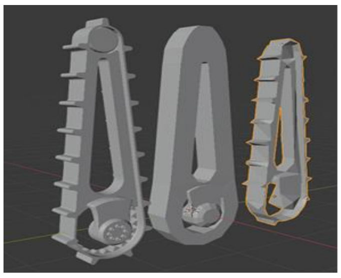

Fig. 3: Mesh examples of the Servosila Engineer robot's front sub-crawler: an original mesh (left), a simplified manually created mesh (center), an automatically generated mesh (right).

Figure 3 demonstrates three different models. The one on the left is a visual model of a front sub-crawler (flipper) of the crawler-type robot Servosila Engineer. The others are collision meshes, which were created using different options. The one in the center was created manually [11] and the one on right was generated automatically [23]. Table I presents a comparison of the two approaches. When possible, we strongly recommend a manual modelling.

Table 1: Creating collision meshes methods comparison

Comparison criteria	**Mode**	
	Manual	**Automatic**
Time consumption	Low	High
Quality	High	Low (most cases)
Optimization	High	Low (most cases)
Modelling skills requirements	High	Low

4. Building a robot

4.1. Writing description file

Unified Robot Description Format (URDF) is a main instrument for a robot description in ROS/Gazebo environment. One of the URDF realizations is Xacro (XML Macros) that became popular among users because of several improvements such as parametrization and macros. It makes a description more readable and easier to construct. The Xacro description could be valuable for a large and complicated project. Moreover, it significantly decreases a size of a file.

Each element of the robot should be properly described. Information about links' length and joint limits allows reaching a good level of similarity between a simulation model and its real-world counterpart. A part of a XACRO file code that describes the Servosila Engineer robot is presented in Fig.4; it contains a description of two links of the manipulator – a waist link (lines 1-14) and a shoulder (lines 23-40) link – and a shoulder joint (lines 41-45) between them.

A description of every link of the robot contains paths to its visual and collision meshes and an inertial unit, which includes information of weight and inertia. Joints have information about links they connect, a rotational axis and an origin, friction data and limits of position and velocity.

4.2. Setting up inertia

Inertias are one of the most significant parts of a robot link description. Incorrectly tuned inertia can make a model unrealistic and even destroy it. An example of improperly tuning of the Servosila Engineer robot inertias that caused incorrect model behavior after its spawning in a Gazebo world is shown in Fig. 5. A proper visual model of the robot initially appeared at a predefined height of a Gazebo world 3D space and under gravitation force glided down; upon its contact with a ground plane the parts of the model felt apart.

```
01: <!--     waist        -->
02: <link name="waist link">
03: <visual>
04: <geometry>
05: <mesh filename="package://(path-to-the-directory-with-visual-
meshes)/cronstain.dae"/>
06: </geometry>
07: </visual>
08: <collision>
09: <geometry>
10: <mesh filename="package://(path-to-the-directory-with-collision-
meshes)/Cronstain.dae"/>
11: </geometry>
12: </collision>
13: <xacro:cuboid inertia mass="${waist mass}" length="0.08" width="0.08"
height="0.08">
14: <origin xyz="0 0 0" rpy="0 0 0"/> </xacro:cuboid inertia> </link>
15: <gazebo reference="waist link">
16: <selfCollide>false</selfCollide>
17: <kp>${kp}</kp>
18: <kd>${kd}</kd>
19: <mu1>100</mu1>
20: <mu2>50</mu2>
21: </gazebo>
22: <!--     shoulder    -->
23: <link name="shoulder link">
24: <visual>
25: <geometry>
26: <mesh filename="package://(path-to-the-directory-with-visual-
meshes)/shoulder.dae"/>
27: </geometry>
28: </visual>
29: <collision>
30: <geometry>
31: <mesh filename="package://(path-to-the-directory-with-collision-
models)/Shoulder.dae"/>
32: </geometry>
33: </collision>
34: <inertial>
35: <mass value="${shoulder mass}"/>
36: <origin xyz="-0.0303 -0.0001 0.1511" rpy="0 0 0"/>
37: <inertia ixx="0.0295383" ixy="-0.0000001" ixz="-0.0004068"
38: iyy="0.0292352" iyz="0.0000004" izz="0.0011211"/>
39: </inertial>
40: </link>
41: <joint name="shoulder" type="revolute"> <parent link="waist link"/>
<child link="shoulder link"/>
42: <axis xyz="1 0 0"/>
43: <dynamics friction="${friction}" damping="${damping}"/> <origin
xyz="0.036 0.051 -0.07" rpy="${pi} 0 0"/>
44: <limit lower="${shoulder llimit}" upper="${shoulder ulimit}"
effort="${shoulder mass * 50}" velocity="${joints vlimit}"/> </joint>
45: <gazebo reference="shoulder link">
46: <selfCollide>false</selfCollide>
47: <kp>${kp}</kp>
48: <kd>${kd}</kd>
49: <mu1>100</mu1>
50: <mu2>50</mu2>
51: </gazebo>
```

Fig. 4: Code listing of the file engineer *arm.xacro*.

An inertial block describes a mass of a link, its center of mass (6 coordinates) and a matrix of inertia tensors. The inertial block code example appears in Fig.4, lines 34-39.

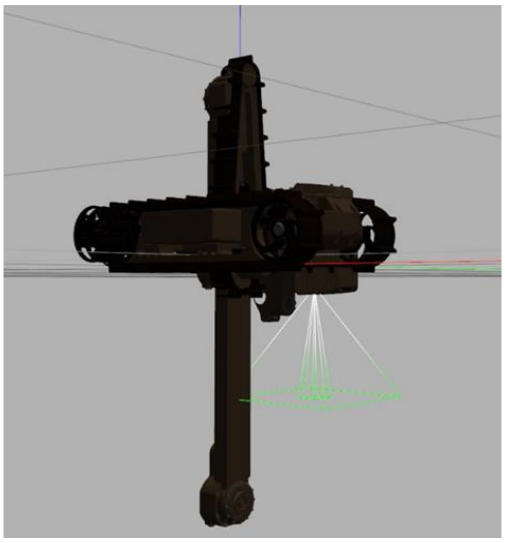

Fig. 5: The robot model felt into parts due to wrong inertia settings.

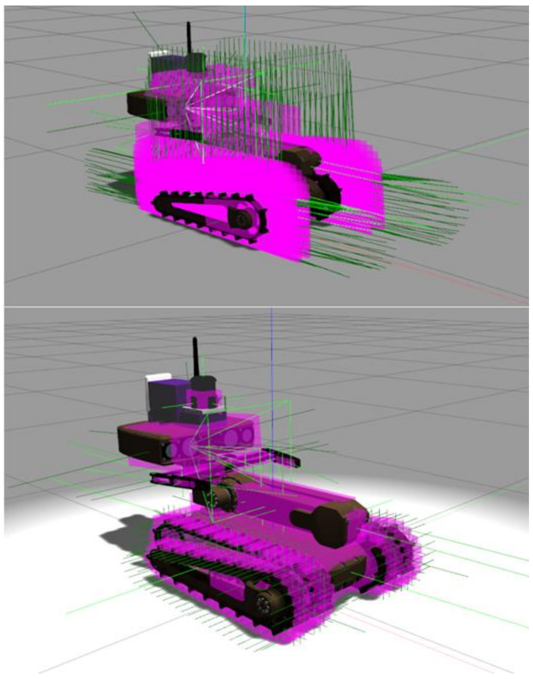

Fig. 6: Inertial blocks for Servosila Engineer model are shown with magenta color.

Figure 6 presents two attempts of setting the inertial block of the Servosila Engineer robot model in Gazebo. Properly tuned inertia should have a shape, which is maximally close to an object it is attached to. In Fig. 6 (top picture) one can note that inertia of the robot head (magenta color) nearly three times exceeds the original parts of the robot (brown and black colors). Figure 6 (bottom picture) presents corrected inertial data. In addition, while setting up inertia, it is important to remember about weights (mass) that do not have visual parts; the weights should be set according to the real robot technical information.

There are several ways to set up inertial blocks. First of all, inertias could be calculated using precise measurements and standard formulas. This option might be laborious, especially for complicated models. Another solution is to select parameters using visualization in Gazebo (Fig. 7). It could be faster than the first option but still it takes time and optimality of this method is quite questionable. The third option, which we recommend, is using a corresponding software, e.g., MeshLab [25]. It allows to easily calculate inertia values within three steps:

- Import a model using *File-Import Mesh* tab
- Open a console for a log output with *View-Show Layer Dialog*
- Calculate inertia value with option *Filters-Quality Measure and Computations-Compute Geometric Measures*

Table 2 compares the three options of setting up inertial blocks by their time consumption and resulting quality.

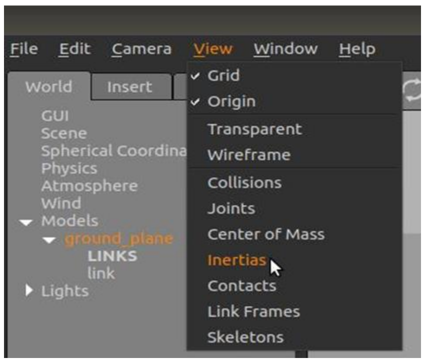

Fig. 7: *Inertias* option tab of *View* menu in Gazebo simulator.

Table 2: Inertia setting methods comparison

Comparison criteria	**Approach**		
	Inertia formulas	**Visual approximation**	**Software**
Time consumption	Low	Low (most cases)	High
Quality	High	Low (most cases)	High

4.3. Adding ROS controllers

Controllers are used for moving a robot within a simulation. They are connected to model joints and move them according to given commands. Adding controllers contains three steps:

1) Add transmissions to every moving joint. The trans-mission contains information about a type of a joint, an interface, an actuator and a name of a joint it is connected to. An example of code with a transmission description is listed in Fig.8.

```
<transmission name="waist_shoulder_trans">
<type>transmission_interface/SimpleTransmission</type>
<actuator name="waist_shoulder_motor">
<mechanicalReduction>1</mechanicalReduction>
</actuator>
<joint name="shoulder">
<hardwareInterface>
hardware_interface/EffortJointInterface
</hardwareInterface>
</joint>
</transmission>
```

Fig. 8: Code listing of the file *engineer_arm.xacro.*

2) Create a YAML file that contains parameters of the controllers. It contains a controller type, a joint name and PID parameters. An example of such description is listed in Fig.9.

```
shoulder_position_controller:
type: effort_controllers/JointPositionController
joint: shoulder
pid: {p: 100.0, i: 0.01, d: 10.0}
```

Fig. 9: Code listing of the file *engineer_control.yaml.*

3) Create a launch file for the controllers. It should contain a loader of a controllers' list created in the previous step and a launcher of the controllers. An example of a launch file is listed in Fig.10.

```
<rosparam file=
"$(find engineer_control)/config/engineer_control.yaml" command="load"/>
<node name="controller_spawner" pkg="controller_manager"
type="spawner" respawn="false" output="screen"
args="shoulder_position_controller"/>
```

Fig. 10: Code listing of the file *engineer_control.launch.*

Finally, after launching the model and its controllers, several ROS-topics are required to control the robot in a simulated environment. For example, to control *shoulder* joint there is a topic named *shoulder position controller* (refer the code in Fig.4). It works with a message of *std/msgsFloat64* type. To send a command to the controller the command in Fig.11 is used.

```
rostopic pub -1 \/shoulder\_position\_controller std\_msgs\/Float64 "data: 0.5"
```

Fig. 11: Console command for sending command to the robot.

4.4. Adding sensors

Robots are typically equipped with several types of onboard sensors, including cameras, laser range finders (LRF), IMUs and others. A number of sensors already have simulation models for the ROS/Gazebo environment. Adding a sensor to a robot simulation model means adding it to a robot description in a XACRO or URDF file. Every sensor type has its own description pattern incapsulated into a corresponding plugin [26]. An example of a sensor description in Fig.12 corresponds to a mono camera of the Servosila Engineer robot.

```
<gazebo reference="camera${number}_link"> <sensor type="camera"
name="camera_${number}"> <update_rate>${fps}</update_rate>
<camera name="head_${number}">
<horizontal_fov>1.3962634</horizontal_fov> <image>
<width>${width}</width>
<height>${height}</height>
<format>R8G8B8</format>
</image>
<clip>
<near>0.02</near>
<far>300</far>
</clip>
<noise>
<type>gaussian</type>
<mean>0.0</mean>
<stddev>0.007</stddev>
</noise>
</camera>
<plugin name="camera${number}_controller"
filename="libgazebo_ros_camera.so">
<alwaysOn>true</alwaysOn>
<updateRate>0.0</updateRate>
<cameraName>camera${number}</cameraName>
<imageTopicName>image${number}_raw</imageTopicName>
<cameraInfoTopicName>camera_info</cameraInfoTopicName>
<frameName>camera${number}_link_optical</frameName>]
<hackBaseline>0.0</hackBaseline>
<distortionK1>0.0</distortionK1>
<distortionK2>0.0</distortionK2>
<distortionK3>0.0</distortionK3>
<distortionT1>0.0</distortionT1>
<distortionT2>0.0</distortionT2>
<CxPrime>0</CxPrime>
<Cx>0.0</Cx>
<Cy>0.0</Cy>
<focalLength>0.0</focalLength>
</plugin>
</sensor>
</gazebo>
```

Fig. 12: Code listing of the file *engineer_arm.xacro*.

An example of working sensors is demonstrated in Fig. 13. Data from working cameras and LRF are presented in RViz. LRF scan data are shown in the left subfigure with red dots, corresponding to a ball and a cube obstacles of the simulated Gazebo world (right subfigure). Visual data are captured by the four cameras of the robot, and a user could switch between video streams of the cameras by going through the corresponding tabs in the bottom of the camera window (left subfigure); in the figure a dynamically updated frame from the right camera demonstrates the ball and the cube obstacles.

5. Conclusions

This paper presented a tutorial for creating a simulation model within the Gazebo simulator using Robot Operating System (ROS). It described an entire process starting from the model construction and physics setup and ending up with sensory and Robot Operating System based control. The process is illustrated with an example of a step-by-step modelling of a crawler-type robot Servosila Engineer. Examples of implementation with code, detailed comments, explanations, and corresponding video files are available as open source supporting files [24].

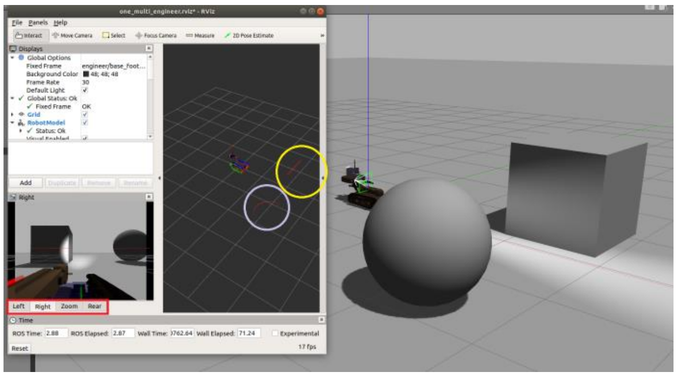

Fig. 13: Example of the working cameras and the LRF in the simulation. Right: side view of the robot and the environment in the Gazebo simulation. Left: RViz window with data from the right camera in simulation; the red rectangular emphases tabs of switchable camera views; the yellow circle emphases LRF data that corresponds to the cube obstacle; the magenta circle emphases LRF data that corresponds to the ball obstacle.

6. Acknowledgements

This paper has been supported by the Kazan Federal University Strategic Academic Leadership Program (”PRIORITY-2030”).

7. References

[1] S. Smys and G. Ranganathan. Robot assisted sensing control and manufacture in automobile industry. *Journal of ISMAC*. 2019, **1**(3): 180–187.

[2] V. Voronin, M. Zhdanova, E. Semenishchev, A. Zelenskii, Y. Cen, and S. Again. Action recognition for the robotics and manufacturing automation using 3-d binary micro-block difference. *Int J Adv Manuf Techno*. 2021, **17**: 2319–2330.

[3] E. A. Martinez-Garcia, O. Akihisa et al. Crowding and guiding groups of humans by teams of mobile robots. I*EEE Workshop on Advanced Robotics and its Social Impacts*. 2005, pp. 91–96.

[4] D. Ryumin, I. Kagirov, A. Axyonov, N. Pavlyuk, A. Saveliev, I. Kipyatkova, M. Zelezny, I. Mporas, A. Karpov. A Multimodal User Interface for an Assistive Robotic Shopping Cart. *Electronics*. 2020, **9**(12): 2093.

[5] R. R. Murphy. Rescue robotics for homeland security. *Communications of the ACM*. 2004, **47**(3): 66–68.

[6] D. Koung, O. Kermorgant, I. Fantoni and L. Belouaer. Cooperative multi-robot object transportation system based on hierarchical quadratic programming. *IEEE Robotics and Automation Letters*. 2021, **6**(4): 6466-6472.

[7] E. Chebotareva, R. Safin, K.-H. Hsia, A. Carballo, E. Magid. Person-Following Algorithm Based on Laser Range Finder and Monocular Camera Data Fusion for a Wheeled Autonomous Mobile Robot. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 2020, **12336**: 21-33.

[8] E. Magid, A. Zakiev, T. Tsoy, R. Lavrenov, and A. Rizvanov. Automating pandemic mitigation. *Advanced Robotics*. 2021, **35**(9): 572–589.

[9] D. Kolpashchikov, O. Gerget, and R. Meshcheryakov. Robotics in healthcare. In: *Handbook of Artificial Intelligence in Healthcare*. Springer. 2022, pp. 281–306.

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng et al. Ros: an open-source robot op-erating system. In: *ICRA workshop on open-source software*. Kobe, Japan: 2009, **3**(3.2): 5.

[11] A. Dobrokvashina, R. Lavrenov, E. A. Martinez-Garcia, and Y. Bai. Improving model of crawler robot servosila engineer for simulation in ros/gazebo. *13th International Conference on Developments in eSystems Engineering (DeSE), IEEE*. 2020, pp. 212–217.

[12] A. Dobrokvashina, R. Lavrenov, T. Tsoy, E. A. Martinez-Garcia, and Y. Bai. Navigation stack for the crawler robot servosila engineer. In: *16th Conference on Indus-trial Electronics and Applications (ICIEA), IEEE*. 2021, pp. 1907–1912.

[13] Servosila official site. https://www.servosila.com/en/index. shtml, [Online; accessed 30-March-2020].

[14] J. Pages, L. Marchionni, and F. Ferro. Tiago: the modular robot that adapts to different research needs. In: *International workshop on robot modularity, IROS*. 2016.

[15] C. N. Thai. Robotis'robot systems. *Exploring Robotics with ROBOTIS Systems*. Springer. 2017, pp. 5–21.

[16] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2004, pp. 2149–2154.

[17] H. R. Kam, S.-H. Lee, T. Park, and C.-H. Kim. Rviz: a toolkit for real domain data visualization. *Telecommunication Systems.* 2015, **60**(2): 337–345.

[18] J. Van Gumster. *Blender for dummies*. John Wiley & Sons. 2020.

[19] B. R. Kent. 3D scientific visualization with blender. *Morgan & Claypool Publishers San Rafael*. CA, 2015.

[20] K. Murdock. *Autodesk Maya 2019 Basics Guide*. SDC Publications, 2018.

[21] R. Hess. *Blender Foundations: The Essential Guide to Learning Blender 2.5*. Routledge, 2013.

[22] S.Tickoo. *Autodesk 3Ds Max 2021: A comprehensive guide*. Cadcim Technologies, 2020.

[23] M. Sokolov, I. Afanasyev, R. Lavrenov, A. Sagitov, L. Sabirova, E. Magid. Modelling a crawler-type UGV for urban search and rescue in Gazebo environment. In: Artificial Life and Robotics (ICAROB). 2017, pp. 360-362.

[24] Support files for the tutorial. Laboratory of Intelligent Robotic Systems, Intelligent Robotics Department, Institute of Information Technology and Intelligent Systems, Kazan Federal University - https://gitlab.com/LIRS_Projects/public_examples/ModellingInstructionsExample.

[25] P. Cignoni, G. Ranzuglia, M. Callieri, M. Corsini, F. Ganovelli, N. Pietroni, M. Tarini. *MeshLab*. 2011.

[26] Gazebo plugins in ros. http://gazebosim.org/tutorials?tut=ros_gzplugins, [Online; accessed 28-March-2022].