Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Planning for the MoleMOD system

Michaela Brejchová

Supervisor: RNDr. Miroslav Kulich, Rh.D.
Field of study: Cybernetics and Robotics
Subfield: Systems and Control
May 2018

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Brejchová** Jméno: **Michaela** Osobní číslo: **456985**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Kybernetika a robotika**

Studijní obor: **Systémy a řízení**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Plánování pro systém MoleMOD

Název bakalářské práce anglicky:

Planning for the MoleMOD system

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] Petrš, Jan, Havelka, Jan, Florián, Miloš and Novák, Jan, MoleMOD - On Design specification and applications of a self-reconfigurable constructional robotic system, ShoCK! - Sharing Computational Knowledge! - Proceedings of the 35th eCAADe Conference - Volume 2, Sapienza University of Rome, Rome, Italy, 20-22 September 2017, pp. 159-166
[2] Steven M. LaValle, Planning Algorithms, Cambridge University Press, 842 pages, 2006

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Miroslav Kulich, Ph.D., inteligentní a mobilní robotika CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **16.01.2018** Termín odevzdání bakalářské práce: **25.05.2018**

Platnost zadání bakalářské práce: **30.09.2019**

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studentky

Acknowledgements

I would like to thank my supervisor RNDr. Miroslav Kulich, Rh.D. for his guidance, patience and comments that helped me to completed the project.

Also, I would like to thank Ing. arch. Jan Petrš for the figures of the MoleMOD system and Collada files with models that he provided me.

Declaration

I declare that the presented work was written independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, May 2018

Abstract

The aim of this thesis is to design a planning algorithm for the MoleMOD system that is created at the Faculty of Architecture of the Czech Technical University in Prague. The system is designed to build various constructions of special modules. These modules have tunnels inside, where the worm-shaped robots can move, and shift modules.

The first part deals with a description of MoleMOD system, robots and building blocks, and possibilities of its use.

Then the theory of discrete planning and the design of the algorithm follow. An informed method for state-space search will be used, more precisely the A^* algorithm, adapted for the MoleMOD system's needs.

The last part is about the Gazebo simulator in which the algorithm output will be tested. For that, it is necessary to create models of individual parts of the system including their kinematics.

Keywords: MoleMOD, planning, A* algorithm, Gazebo simulator

Supervisor: RNDr. Miroslav Kulich, Rh.D.

Abstrakt

Cílem této práce je navrhnout plánovací algoritmus pro systém MoleMOD, který vzniká na Fakultě architektury ČVUT v Praze. Jde o systém navržený pro stavbu různých konstrukcí ze speciálních modulů. V těchto modulech jsou tunely, kterými mohou prolézat roboty tvarem připomínající červy, a moduly přemisťovat.

První část práce se zabývá popisem systému MoleMOD, robotů a stavebních bloků, a možnostmi jeho využití.

Následuje teorie diskrétního plánování a návrh samotného algoritmu. Použit bude algoritmus pro informované prohledávání stavového prostoru, přesněji algoritmus A^*, upravený pro potřeby systému MoleMOD.

Poslední část je o simulátoru Gazebo, ve kterém se bude výstup algoritmu testovat. K tomu je zapotřebí vytvořit modely jednotlivých částí systému včetně jejich kinematiky.

Klíčová slova: MoleMOD, plánování, algoritmus A*, simulátor Gazebo

Překlad názvu: Plánování pro systém MoleMOD

Contents

Figures

Tables

Chapter 1

Introduction

Currently, advanced technologies are used almost everywhere, even in architecture. Due to higher safety and speed requirements, the use of the robotic systems is more popular. Robots do not need to sleep; they are faster and more precise than humans. They also can work in locations, where it would be really dangerous for people.

The MoleMOD system is a new type of robotic system for usage in architecture to build static constructions as well as movable and adaptive. The system consists of two kinds of units - passive and active. The idea of this project is that the active elements (robots) move the passive elements (building units) and thus create various constructions.

The system as a whole and all its parts are described in more details in the following chapter. It also contains examples of possible practical applications in the future.

The third chapter deals with planning. It contains a description of the discrete planning and state-space search using the informed A_* algorithm followed by the design of the state-space for the MoleMOD system and the modification of the planning algorithm for the needs of the system.

The fourth chapter is about creating models in the robotics simulator called Gazebo. The plans returned by the planning algorithm will be tested in the simulator, so it is necessary to make models of active and passive components of the system. In the case of the active elements, it is needed to add also kinematics.

The summary of the thesis is in Chapter 5. Apart from that, it contains some ideas for the future enhancement of the algorithm to make the planning faster even for a higher number of modules and robots.

Chapter 2
MoleMOD

MoleMOD is a new revolutionary building robotic system. In fact, it is a kind of a modular robotic system (MRS). The system is cheaper than standard MRS because it does not need so many mechanical and electronic parts. It consists only of passive building modules (MOD) and active robots (Moles) that can move the modules from the inside.

Figure 2.1: The MoleMOD system (by Jan Petrš)

The system is inspired by colonies like termites, ants or bees, which permanently rebuild and adapt their "houses" to surroundings and current conditions. In the future, MoleMOD should be adaptive and movable or static, what will be at needed at the moment.[1][2]

2.1 Robots

The robot is made up of three parts: a body, a rotator and a head. Every part has a specific purpose. The head is used for rotating special screws that joint two modules. Each robot has two revolving heads, which are located on both ends of the body. They may also serve as wheels so the robot can ride.

Both heads are connected by a soft body. The body allows movement of the whole robot and control of the heads directions. It is split by rotator into

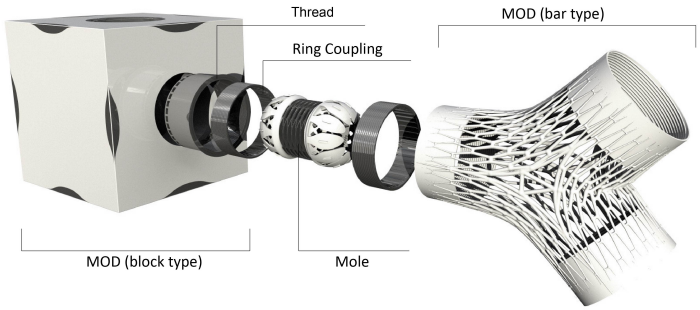


Figure 2.2: Structure of the system (by Jan Petrš)

two pieces consisting of three "vertebras" and three silicone pillows that are between them. The pillows can be inflated or deflated. Thanks to this it is possible to inflect, stretch or attract the whole body. The motion is similar to the movement of worms.

The last part of the robot is the rotator and is used for revolving with modules that are connected to robots.[1][2]

Figure 2.3: Pillows inflation (by Jan Petrš)

2.2 Modular building units

The final building consists only of the modular units. They are like bricks but can be adapted and moved by robots. The robots serve as workers or building machines.

The big advantage of the system is that the units can have several shapes. They only need to have same faces, which are connected by special joints. Each unit has at least one hole or tunnel, where robots can move. Thanks to that, the final construction is like a maze.

Another advantage of passive building units is that they do not need to be of a special material. They can be made of recycled plastics, wood, carbon, concrete or polystyrene and so on. Nevertheless, the weight is important, so the lighter materials are better.[1][2]

2.3 Application

The MoleMOD system is very adaptable and can be used in many various situations. There are lots of locations in the world that are not safe for people,

or there is a problem to build. Places like deserts, mountains or polar regions, which cannot be inhabited, but it can be necessary to build there. MoleMOD may not be used only for building houses, but also for bridges, pylons or research stations.

The system does not need cranes or other machines. Therefore it is quite easy to transport it to the building site. It will even be possible to transfer only robots; building modules will be created by a 3D printer from local materials. This way the system could be used for the colonization of another planet.

Other uses may be for example temporary constructions. Tribunes for sports events, such as the Olympic Games or the World Cup races, markets, exhibitions, festivals, events that last only a few days or weeks.

No less important is the possibility of using the system in case of a disaster. It can be building of bridges after a flood, shelters for people who have lost their homes due to a catastrophe and so on. Also, it can be used after a nuclear disaster, when the presence of humans is not possible because of radiation.[1][2]

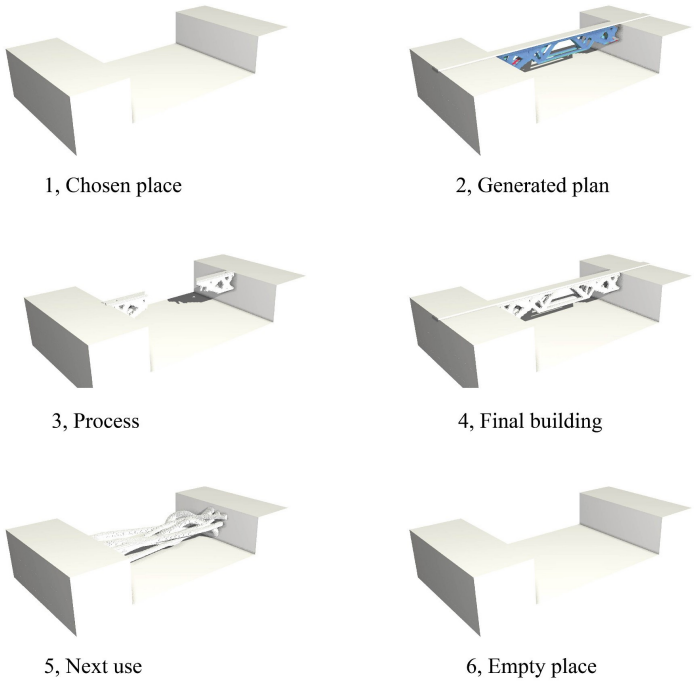


Figure 2.4: Building (by Jan Petrš)

Chapter 3

Planning

This chapter describes a planning algorithm for the MoleMOD system. The task is to put building modules into required construction by a sequential moving of models.

3.1 Discrete Planning

The basic idea of the discrete planning is that different situations are called *states* and set of all states, which are possible in the world, is called *state space*. The world can be changed by *actions*. When an action u is applied to the current state x, a new state x' is found according to *state transition function*:

$$x' = f(x, u).$$

The *action space* $U(x)$ represents all actions that can be applied to x. The same action can be used for several states. Then we define the set of all possible actions:

$$U = \bigcup_{x \in X} U(x).$$

The task of a planning algorithm is to find a series of actions that step by step transform the initial state to a *goal state* x_G or a set of goal states X_G.[3]

3.2 A* algorithm

The A^* algorithm count among informed methods for state-space search. It means that each state is somehow evaluated. In the case of the A^* algorithm it is the function:

$$f(x) = g(x) + h(x).$$

$g(x)$ is the cost of the entire way from the initial state to the current state x. It can be calculated incrementally in the algorithm, so there is no difference between the real cost $g^*(x)$ and the computed cost $g(x)$.

Function $h(x)$ is called *heuristic function*. Unfortunately, we are not able to calculate the real cost $h^*(x)$ of the way between state x and the goal state. However, we can estimate it. We need its value to be as close to the value of $h^*(x)$ as possible. If $0 \leq h(x) \leq h^*(x)$ for every x, we say that the heuristic function is *admissible* and the algorithm is guaranteed to find the cheapest way from initial to the goal state.

In many cases, we can make a reasonable underestimate of the cost. For example for planning of moving on a 2D grid, assuming that the cost of the way is the total number of steps, we can use the underestimate $h(x) = |i' - i| + |j' - j|$, where (i, j) are coordinates of one state and (i', j') are coordinates of another state. When there are some obstacles, the real cost can increase, but it never will be lower, because this is the least possible cost of the way between two states. If a longer way is taken, there is a possibility that the algorithm will not work correctly. For some states the computed cost might be higher than the real cost and the algorithm may choose a state that has a lower cost, but in fact, it has a higher cost.[3]

The entire A^* algorithm is shown in Listing 3.1. All open states are saved in an open list. In each cycle, the list is searched and the best state is selected (line 3). That means the algorithm computes $f(x)$ for every state in the open list and return the state with the lowest value. Then the state is inserted into a closed list (line 4) and removed from the open (line 5). If the state is the goal state, the algorithm terminates (line 7). Otherwise, the state expands (line 9) to other states that are stored in an expanded list. Every state e from this list, if it is not already in the closed, is inserted into the open and the cost of the way from the initial state $g(e)$ is computed. And then the cycle is repeated until a solution is found.

```
1  open.insert(initial)
2  while (open is not empty)
3      x = best(open)
4      closed.insert(x)
5      open.remove(x)
6      if (x is goal)
7          return SUCCESS
8      else
9          expanded = expand(x)
10         expanded = expanded - closed
11         open = open + expanded
12 return FAILURE
```

Listing 3.1: The A^* algorithm in the pseudocode

Operations at lines 10 and 11 need to be specified. Because some of the expanded states may already be in the open list or the closed list, it is necessary for the correct function of the algorithm to select just one of them. In case that x is already in the open list and its cost is better, then the state is not added into the list. If the cost of the state in the open list is worse, then the newly expanded state is added to the list and the original is removed.

The similar procedure is followed if x is already in the closed list. The better state is added to the open list and removes from the closed list. If it is worse than the state in closed, nothing happens.[8]

3.3 MoleMOD

To simplify the task, I will consider only two-dimensional space and cube-shaped building modules with tunnels crosswise that allow the robot to move up, down and to both sides. I will also not contemplate connecting blocks by special joints.

3.3.1 State

A state is represented by an arrangement of the buildings modules and robots positions. For the needs of the program, it is characterized by an array of values.

0	empty place
1	module
2	module with a robot

For example, states from Fig.3.1 are represented by the two-dimensional matrices below.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

state 1 state 2 state 3

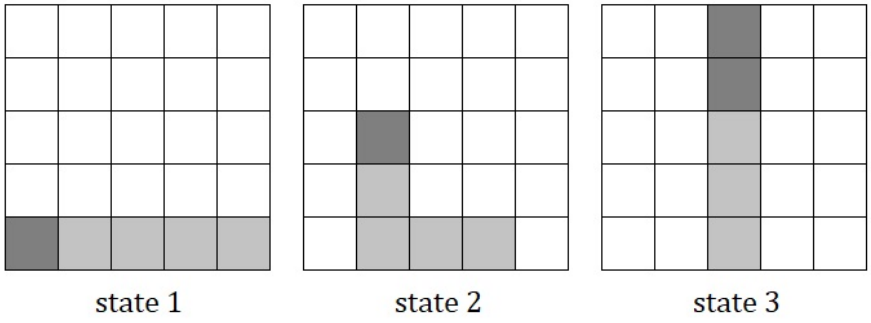


Figure 3.1: Examples of states

3.3.2 Actions

From a state to a next state it is passed by robots moving and shifts modules. All actions depend on the positions of robots and modules, the total number of robots and other conditions.

The simplest motion that the model can perform is moving from one block to another. The condition for this action is that the position, where the robot moves to, is not outside a given area and also there is a module. This way the model can move to the right, to the left, up and down.

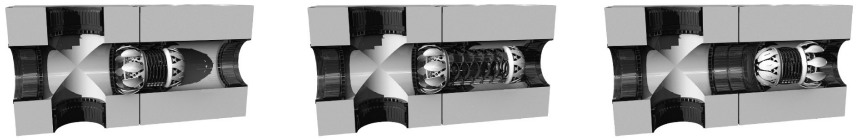


Figure 3.2: Moving in a tunnel (by Jan Petrš)

Another important movement is lifting modules. In reality, the model expands and partially inserts into the blocks beside. After that, the robot lifts one block up and place it on the top of the second one. Finally, the model retracts into one of the two blocks. To reduce the number of possibilities, I will require the robot picks up the module, where it originally was, and remains in the module after the movement. To the motion can be executed, there has to be a free space around the moving block.

Putting down is similar to lifting. The robot expands into the module under its position and then contracts with the top one. For simplicity, the model starts and finishes again in the moved block.

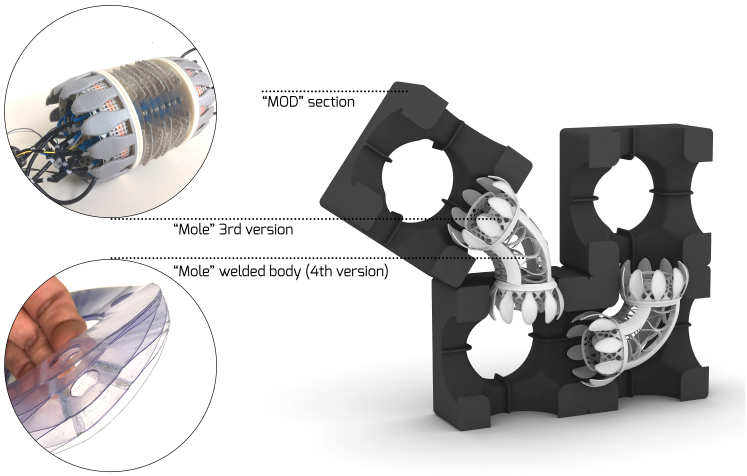


Figure 3.3: An example of movement (by Jan Petrš)

The last motion that is possible with only one robot is moving a module to the right or the left. The model is in the block that we want to shift. If the place next to the block is free and under it, there is another module, the

robot expands to this module. Then it moves the block to the empty position and contract. The moving finishes again in the shifted module.

All movements, which were mentioned above, are valid also for the case with more robots. The advantage is that they can be performed at the same time, so the entire construction is done much faster.

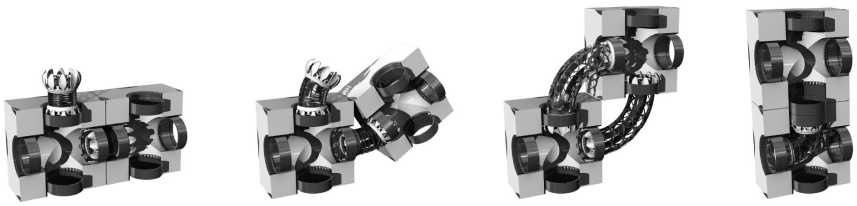


Figure 3.4: Cooperation of two robots (by Jan Petrš)

Besides, robots can work on a movement together. A module is lifted by one robot to a certain position, where a second robot takes it and completes the move as shown in Fig.3.4.

The planning algorithm applies one action, where cooperation of two robots is used. When the block is lifted by one robot, it is like stairs, the block moves not only up but also to a side. In some cases, however, it is necessary to move just upwards.

The movement starts identically as lifting. The robot in the module expands to the module beside and then elevates the block one position up. The second robot has to be in the module that lies on the block where the robot stretched to before. The second robot expands and "catches" the module. The first robot contract to the underlying block.

3.3.3 Cost calculation

Because the problem resembles a moving on a 2D grid, the cost of the way between two states is computed similarly. The distance of a model or a block moving is equal to $|i' - i| + |j' - j|$, where (i, j) are coordinates of the robot or the module in the first state and (i', j') are its coordinates in the second state. However, computing the distance in every step means to find the module or the robot that has just been moved, and calculate how far has shifted. That is senselessly complicated. Because the number of movements is limited and each has a specific distance that never changes, it is much simpler to assign a value d to every movement.

$$g(x_n) = g(x_{n-1}) + d(m).$$

In the case of one robot, it is obvious. The robot can perform only one movement in one step. But with more robots it is complicated. For example, two robots can shift two blocks at the same time or gradually in two steps. The second option does not have any advantages, it only prolongs the building, so it is necessary to obviate it. That can be done by adding 1 to the distance in every step.

$$g(x_n) = g(x_{n-1}) + d(m) + 1,$$

where $d(m)$ is the sum of costs of the motions that have been made to get from the state x_{n-1} to the state x_n.

Counting the distance to the goal state is more difficult. A robot moves to a block, shifts it, moves to another block, shifts it and so over and over again until the goal state is reached. It is almost impossible to calculate the real cost.

When a block moves between two states, the cost is equal to $|i' - i| + |j' - j|$. If more blocks are moved, the cost is $\sum_{n=1}^{N} |i'_n - i_n| + |j'_n - j_n|$, where N is the number of blocks that have been moved.

To the total cost have to be added costs of robots moving. But it is hard to reckon them in advance. It depends on the number of robots and modules and their positions. Every time a robot moves a block to the correct locations, it has to move to the next one. The minimal cost of the way from a module to another is one. The total cost of the way between blocks is equal to $n_{wrong} - n_{robot}$, where n_{wrong} is the number of the modules in the wrong positions and n_{robot} is the total count of robots. For example, one robot to shift five blocks needs to do at least four steps without modules. When there is a robot in every block, the total cost of the robots' movements can be zero.

Therefore, the estimation of the cost of the way from the current state x to the goal state x_G is:

$$h(x) = \sum_{n=1}^{n_{wrong}} (|i_{Gn} - i_n| + |j_{Gn} - j_n|) + n_{wrong} - n_{robot}.$$

3.4 An example of planning

Fig.3.5 shows a simple problem for the planning algorithm. The task is to build an L-shaped construction of five modules placed side by side on the ground using one robot.

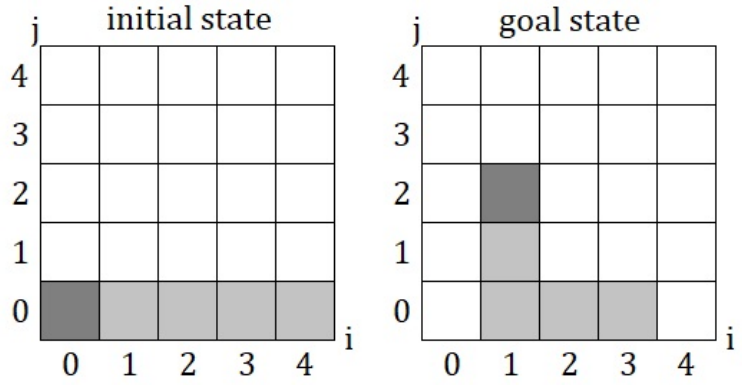


Figure 3.5: An example of planning

In the first planning cycle, the open list contains only one state, the initial state. It is expanded and new states are inserted into the list. In this case, it is possible to move to two different states (Fig.3.6).

Two of the five modules are in the wrong places. The total distance $h(x)$ to the correct positions is 7. $g(x)$ is for the initial state equal to 1.

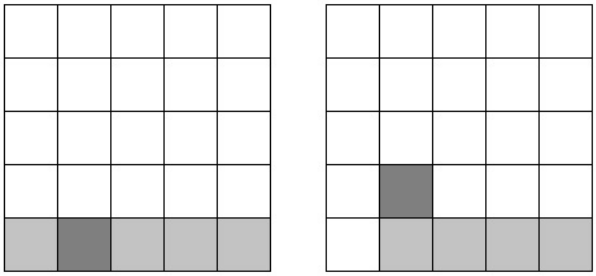


Figure 3.6: Possible next states for the initial state

The first option is that the robot will move one module to the right (Fig.3.6, the state on the left). This movement will increase $g(x)$, but it will not be closer to the target.

The second possibility (Fig.3.6, the state on the right) is to lift the block, where the robot is currently located, and put it on the module nearby. The motion will increase $g(x)$ to a higher value than the previous one, but also will reduce $h(x)$ by moving one module to the correct position.

$$f(x_1) = g(x_1) + h(x_1) = 1 + 7 = 8$$

$$f(x_2) = g(x_2) + h(x_2) = 2 + 5 = 7$$

The second state is better, so it is expanded (Fig.3.7) and removed from the open list. The first state (on the left) will not be inserted into the open, because it is already in the closed. The second and third states will be stored in the open, so the list now contains two state with $f(x) = 8$ and one state with $f(x) = 9$.

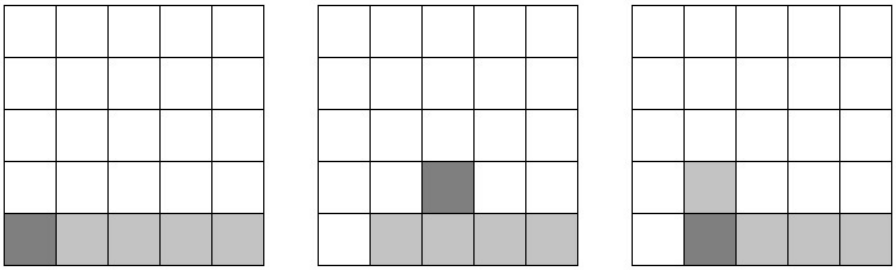


Figure 3.7: Possible next states

The rest of the planning is similar. From the open list, the state with the lower $f(x)$ is selected and expanded. Then all new convenient states are added to the list and the cycle is repeated.

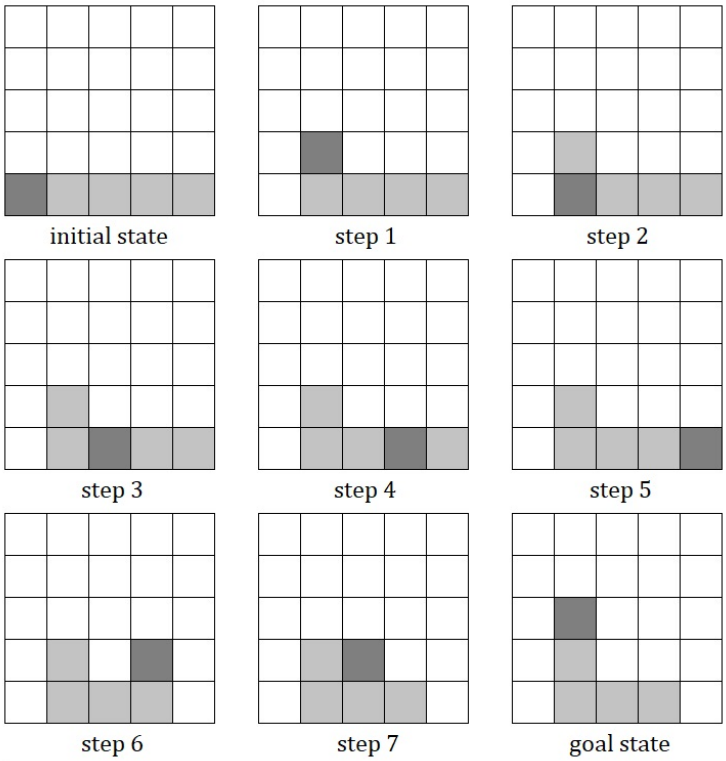


Figure 3.8: The building step by step

3.5 Planning time

The time that the algorithm needs to find a solution depends on several factors. The most important is the number of blocks and robots. The more blocks or robots, the longer time the search will take.

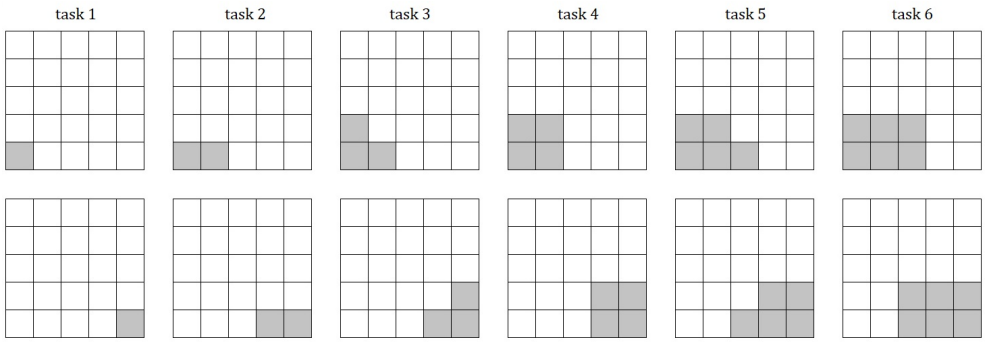


Figure 3.9: Planning tasks for time measurement

Fig.3.9 illustrates six planning tasks. In each pair, the upper state represents the initial layout of the blocks and the state below the goal arrangement. All tasks have been successively solved for different numbers of robots and for all cases the time of the planning was measured.

time [ms]						
	1 robot	2 robots	3 robots	4 robots	5 robots	6 robots
task 1	0.24					
task 2	1.87	2.02				
task 3	3.08	18.23	5.10			
task 4	9.00	75.60	117.95	9.91		
task 5	15.50	224.10	614.93	332.15	18.63	
task 6	35.64	1212.75	4390.14	4000.69	706.35	22.48

Table 3.1: Time of planning 1

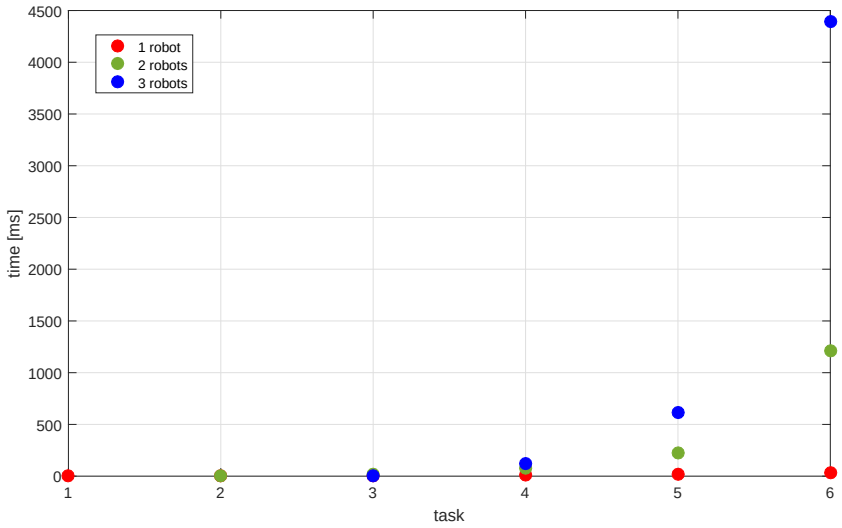


Figure 3.10: Time of planning dependent on the task (diverse numbers of robots)

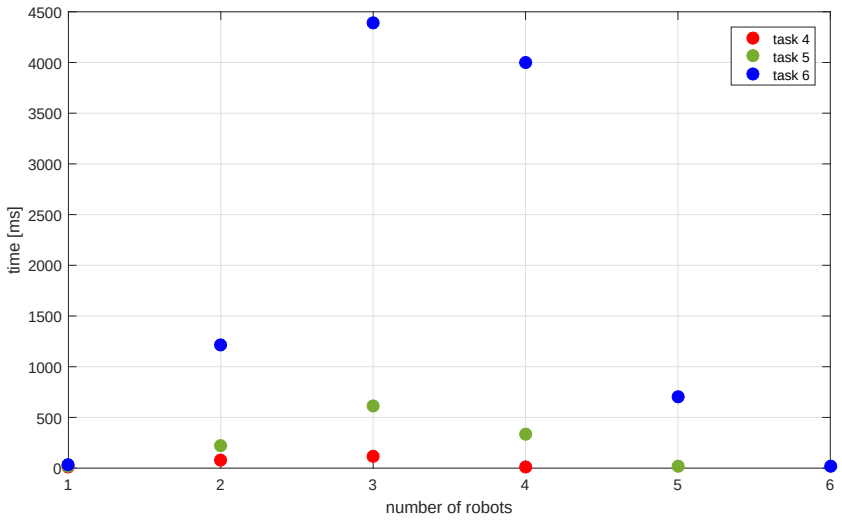


Figure 3.11: Time of planing dependent on the number of models (diverse types of tasks)

Fig.3.10 shows that the time needs to find a solution, if the system contains more robots, is significantly higher than the case of one robot.

However, Fig.3.11 demonstrates that the ratio of modules and robots is also important. Robots need sufficient amount of space to move, so in the case that most blocks are occupied, the number of options is reduced.

The other case is, if the quantity of modules is quite higher than the number of robots, so the robots have plenty of space for moving. Each robot can perform some movements, the number of moves depends on the specific conditions. One more robot adds its motions and combinations of its moves and moves of others. The larger space and the more blocks it contains, the more movements will be possible.

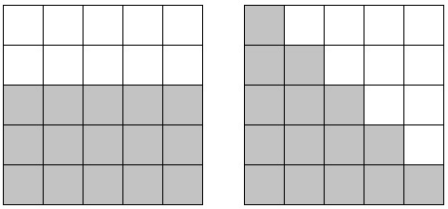


Figure 3.12: Another task for time measurment

number of robots	time [s]
1	0.048
2	13.373
3	24.002
4	156.559

Table 3.2: Time of planning 2

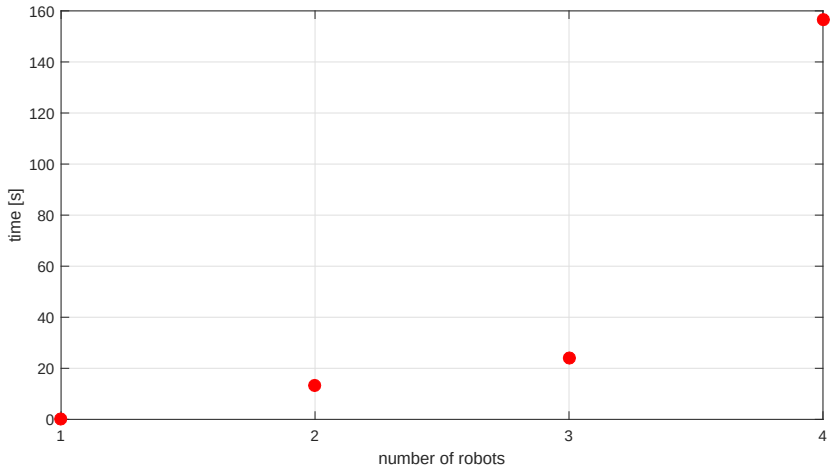


Figure 3.13: Time of planning dependent on the number of models

Chapter 4

Simulation environment for the MoleMOD system

This chapter provides a description of the Gazebo simulator and design of models for testing the MoleMOD system. The first part contains information about the simulator, its functionality and worlds and models creations. The second section describes models made for the MoleMOD system and some basic movements.

4.1 Gazebo

Gazebo is an open-source robotics simulator. It can be used to design robots, test algorithms and artificial intelligence systems using realistic scenarios. The simulator offers indoor and outdoor environments with the possibility of setting several properties, such as wind, gravitation, friction and so on. Gazebo includes multiple physics engines (ODE, Bullet, Simbody and DART), a library of robot models and environments, several types of sensors and functional graphical and programmatic interfaces.[4]

4.1.1 World and model files

A world file includes all the elements, such as robots, lights, sensors or static object. The files use SDF (Simulation Description Format), an XML format that was originally developed as a part of the Gazebo simulator for description objects and environments.

Model files are similar to world files but contain only specifications for a model. The model created by this file can be included in a world file, so it is possible to use one model several times without rewriting the entire code. Also, there is the online model database.

SDF models can be just simple shapes but also complex robots. Basically, a model consists of links, joints, sensors, collision objects, visuals and plugins.[5]

link A link contains the physical properties. It is a body of the model or its part. It may have many collision and visual elements.

collision A collision element is a geometry that is used to check collisions. A link can contain many collisions.

visual A visual element visualize parts of a link. A link can have many visuals or none.

joint A joint connects two links. Each joint has a parent and a child, an axis of rotation and some other properties.

sensor A sensor collects data from the world and these are then used by a plugin. A link can have sensors but do not need them.

An example of a simple world is shown in Listing 4.1. The world contains two models included from the database - the sun and the flat ground. The third model consists of a cube-shaped link with a size of $1 \times 1 \times 1$ meter. It is located at the origin of the coordinates; just the z coordinate is non-zero to raise the link above the ground plane.

```
<sdf version="1.4">
    <world name='my_world'>
        <include>
            <uri>model://sun</uri>
        </include>
        <include>
            <uri>model://ground_plane</uri>
        </include>
        <model name='my_model'>
            <link name='cube'>
                <pose>0 0 0.5 0 0 0</pose>
                <collision name='collision'>
                    <geometry>
                        <box>
                            <size>1 1 1</size>
                        </box>
                    </geometry>
                </collision>
                <visual name='visual'>
                    <geometry>
                        <box>
                            <size>1 1 1</size>
                        </box>
                    </geometry>
                </visual>
            </link>
        </model>
    </world>
</sdf>
```

Listing 4.1: An example of SDF

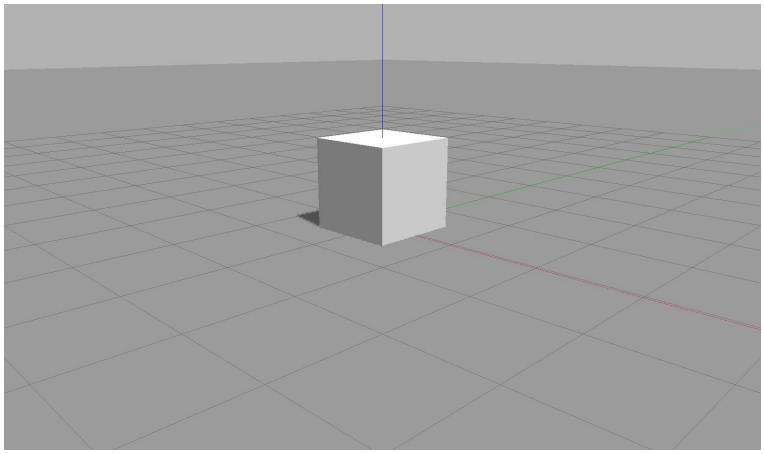


Figure 4.1: An example of SDF

4.1.2 Plugins

A plugin is a C++ code that is compiled as a shared library. Plugins can modify a simulation, for example, it can move models, change parameters of links, models or a world, respond to events, insert new models and so on.

There are six types of plugins:

- world
- model
- sensor
- system
- visual
- GUI

Each plugin type deals with a different part of the simulation. A world plugin is attached to a world and controls its properties. A model plugin is connected to a model, sensor plugin to a specific sensor etc.

A plugin type should be chosen according to its usage. For this thesis, I have opted a model plugin, because it allows changing the physical properties of the models when the simulation is running or set velocity to the joints.

4.2 Models

This section describes the particular steps of creating models for the Gazebo simulator. It contains a delineation of the concrete models of the robot, the parts of which the model is assembled and their capabilities.

4.2.1 Modules

Creating modules in the simulator would be unnecessarily complicated. A module is a passive element. It does not need to be able to move; this is done by a robot. Despite that, modules are quite complex. Even a simple cube-shaped block with only one straight tunnel would be set up of at least four parts.

That can be solved by using a triangle mesh, a collection of triangles that are connected to define the shape of a 3D object. A mesh can be saved as a Collada file (.dae). Collada means Collaborative Design Activity. It is an XML-based format that was developed to make it easier to transport 3D assets between applications.[6]

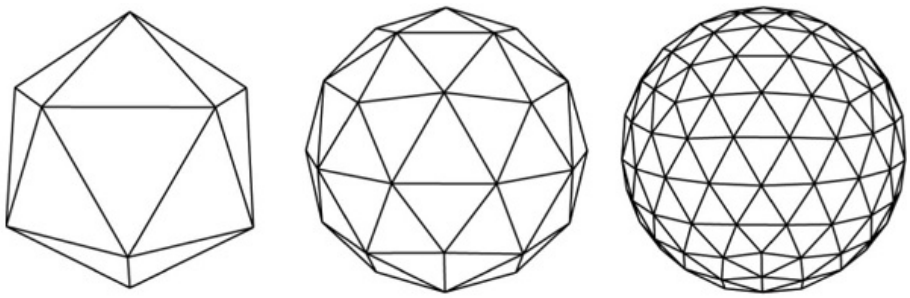


Figure 4.2: Triangle meshes[7]

As can be seen in Fig.4.2, to create a sphere of triangles is not a simple task. The more triangles, the more accurate it will be, but it will never look like a perfect sphere.

A cube is a much simpler shape than a sphere. The modules with angular tunnels are not nearly as complex as the modules with cylindrical tunnels. A robot made of cubes instead of cylinders or spheres also can move more precisely thanks to larger contact surfaces. Therefore, all models' links are cube-shaped.

```
1  <sdf version="1.4">
2      <world name="default">
3          <include>
4              <uri>model://sun</uri>
5          </include>
6          <include>
7              <uri>model://ground_plane</uri>
8          </include>
9          <model name="my_mesh">
10             <pose>0 0 .2 0 0 0</pose>
11             <link name='tunnel'>
12                 <visual name='visual'>
13                     <transparency>0.5</transparency>
14                     <geometry>
15                         <mesh>
```

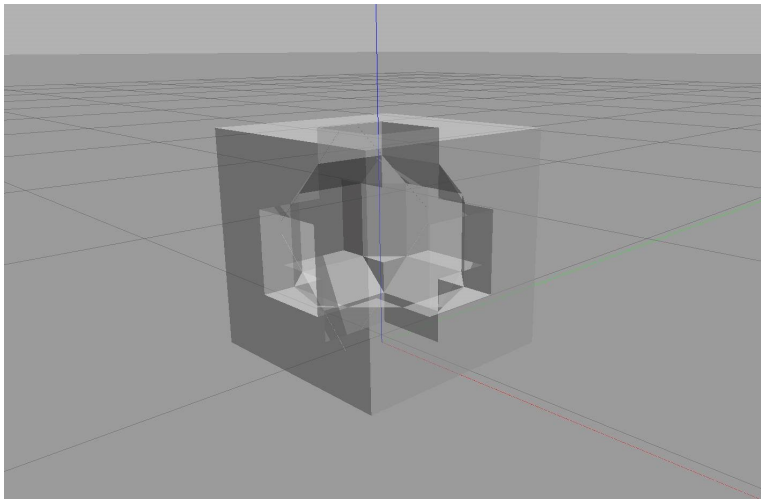


Figure 4.3: Importing mesh

```
16                              <uri>file://1.dae</uri>
17                              <scale>1 1 1</scale>
18                          </mesh>
19                      </geometry>
20                  </visual>
21                  <collision name='collision'>
22                      <geometry>
23                          <mesh>
24                              <uri>file://1.dae</uri>
25                              <scale>1 1 1</scale>
26                          </mesh>
27                      </geometry>
28                  </collision>
29              </link>
30          </model>
31      </world>
32  </sdf>
```

Listing 4.2: Importing mesh

4.2.2 Robots

The robot consists of three parts: a soft body, revolving heads and a rotator. But this is too complex. Therefore I will try to create a simplified model to use in a simulation.

The simplest model

The simplest model is made of two cubes connected by a prismatic joint. This model can move in one direction forwards or backwards by expanding and

contracting of the joint and changing the frictions of the links.

The forward motion consists of four parts:

1. setting frictions
2. the joint expansion
3. setting frictions
4. the joint contraction

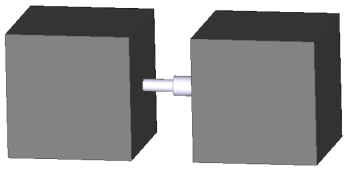


Figure 4.4: The simplest model

The frictions of the links can be set in a model plugin. Before moving itself, it is necessary to lower the friction of the front cube (the first cube in the direction of the movement). For expanding the joint, a positive velocity is set to the joint in the plugin. Then the front cube starts moving.

The simulator updates in certain intervals. After the set number of iterations, the plugin changes the velocity sign and swaps frictions – higher for the front cube and lower for the back cube. After repeating the same number of iterations, the joint is set to zero.

The rotating model

For turning, it is needed to add a revolute joint in the middle of the model. Because the simulator cannot connect two joints, it is necessary to add two links and one more prismatic joint. Now the model is made of two head cubes, two small middle cubes, two prismatic joints and one revolute joint.

The first prismatic joint connects the first head cube with the first middle cube. This cube is joined with the second middle cube by the revolute joint. These both cubes are immaterial and they are placed on themselves. The second middle cube is connected to the second cube with the second prismatic joint.

After these changes, the code of the plugin has to be adapted. For translation, both prismatic joints expand and contract at the same time.

Rotation is similar to translation. The four parts of motion are:

1. setting frictions
2. the joint rotation to the direction, where we want to turn
3. setting frictions
4. the joint rotation to the different direction

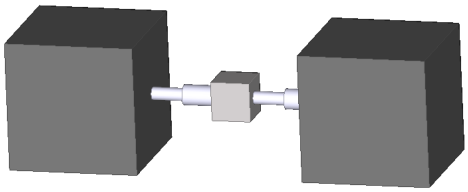


Figure 4.5: The rotating model

At first, the friction of the cube that we want to shift is reduced. Then the plugin set an input velocity to the revolute joint. This velocity can be positive or negative; it depends on the direction, where the robot has to turn. After that, frictions are swapped and velocity with an opposite sign is set to the joint.

In contrast to the moving ahead, this motion stops after one step and will never be repeated, because the model would spin on the same place.

Translation/rotation controller

Unfortunately, the movements we now have are not precise enough. The joints move for the same time, but it does not guarantee that their final position is the same. To make the motion more accurate, a simple controller has been designed. The output to control is the joint position – how the length of the prismatic joint has changed or the angle of rotation for the revolute joint.

```
1 if(position < required_value - accuracy) {
2     setVel(vel);
3 } else if(position > required_value + accuracy) {
4     setVel(-vel);
5 } else {
6     setVel(0.0);
7 }
```

Listing 4.3: A simple joint controller

Listing4.3 shows a primitive controller that sets a positive velocity to the joint, if its position is lower than the required one, negative if larger or zero if it is in the interval determined by the deviation.

For better control, we can divide the possible positions into more intervals. The result will be similar to the example above; it will only contain more conditions.

The controller used in the simulator is divided into five intervals. At the beginning of the motion, the joint is set to an initial speed. When the joint position is close to the required position, the speed is decreased to half the value.

The controller accuracy is 50 micrometres. In this range, the joint is set to zero velocity. If the position is larger or smaller, it depends on the direction

of the motion, the plugin sets a negative speed to the joint. Thanks to this condition there is no need to set the reverse speed for the contracting, the controller will solve it. Also, it is not necessary to count updates; one step consists of expanding to input length and contracting back to "zero".

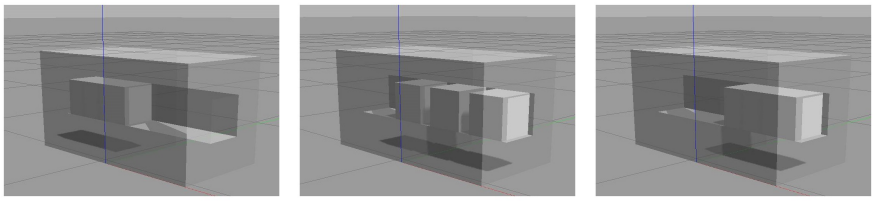


Figure 4.6: Moving in a tunnel - the final model

The second input for ahead motion is a count of iterations, but in this case, it is not the count of updates, but the number of uses of the controller (the total sum of all expanding and contracting). The distance the model moves is equal to the product of one step length and the number of iterations (in meters).

The final model

Unfortunately, the model with just one revolute joint is not sufficient. To turn in a tunnel or lift a building unit, at least two revolute joints are needed. It is also not convenient to use a simple revolute joint, because it can rotate only around one axis. With the joints, the robot could rotate sideways or up and down, but could not do both of these operations. A solution to this problem is to use another joint. The joint is called universal in the simulator and it can rotate around two axes.

The final model consists of three main cubes, four prismatic joints, two universal joints and four little immaterial cubes that are between joints.

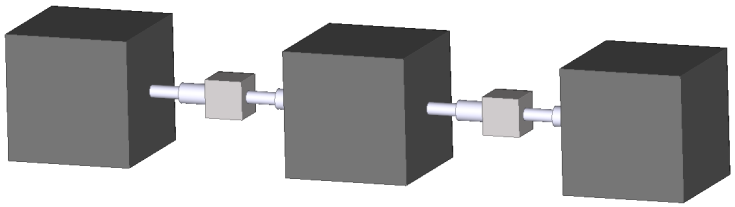


Figure 4.7: The final model

Chapter 5

Testing models in Gazebo

The models for the simulator have been made; the next step is to test them. In the previous chapter, the model of the MoleMOD robot was introduced and two simple moves (forward moving and rotation) were described.

5.1 Simple movements

Forward movement works on the same principle that has been described previously. The only difference is that the final model has four prismatic joints instead of two. The model could use all four joints when moving, but it is simpler to use only two, for example, the first and the fourth joint. That will spare us larger changes in the code and also the motion will be more accurate.

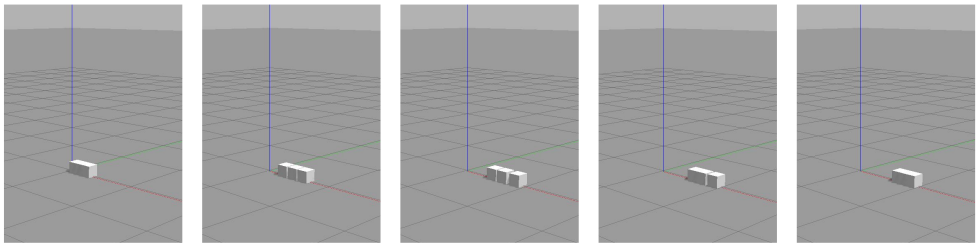


Figure 5.1: Forward moving

The rotation remains practically the same. Only two minor changes are necessary. The last model contains two universal joints, so it is needed to decide which joint will rotate. Then the rotation axis has to be set because the universal joint can rotate around two axes.

5.2 Movements for the planning

However, for simulating the work of the system, these two motions are not sufficient. For the basic version of the planning, it is essential to add lifting (putting a module on a next block or lifting a module just up), shifting modules and moves of robots in tunnels.

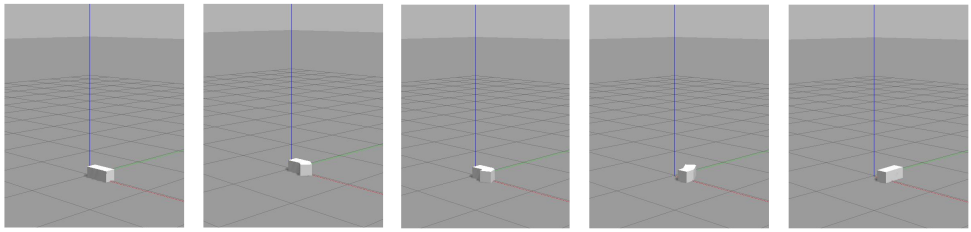


Figure 5.2: Turning right

We have already done the robot movement in a straight tunnel. It is exactly the same as the forward moving. The length of the motion is adjusted according to the size of the block.

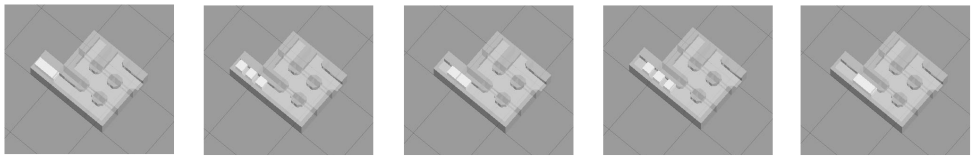


Figure 5.3: Moving in the straight tunnel

Turning in a tunnel is more complicated. Because the robot is only a little bit smaller than the tunnel, it is impossible to turn around at once. It is needed to combine both types of moves - rotation and translation. This can be achieved by using a joint controller.

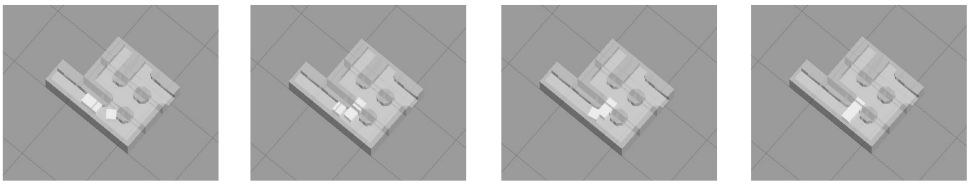


Figure 5.4: Turnning in the module

Similar to turning, also the lifting is a combination of translational and rotational motion.

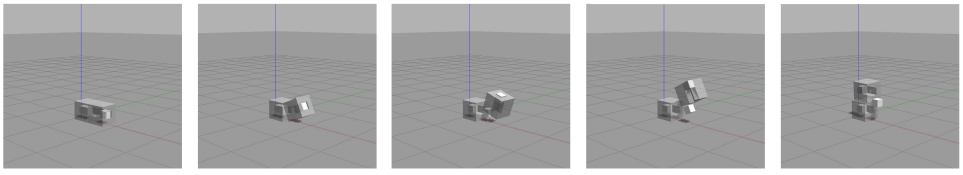


Figure 5.5: Lifting

Regrettably, the lifting is not completed and the shifting blocks does not work at all. The reason is the curious behaviour of block models in the simulator that occurs if two or more modules are close to each other.

Fig.5.6 shows two formations of the same shape. One is made of boxes, the other one of the meshes. The default setting is on the left; the running simulation is on the right. Both, boxes and meshes, have the same properties, but the result is different.

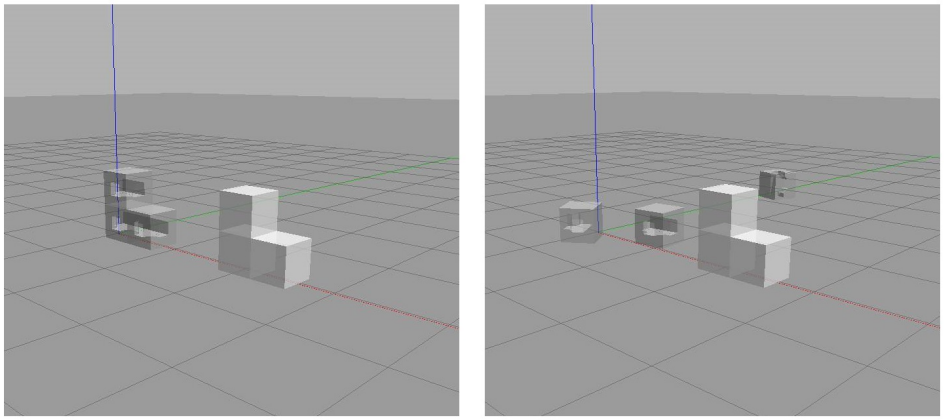


Figure 5.6: A problem with modules

Chapter 6

Conclusion

The planning algorithm for the MoleMOD system has been successfully designed. It can be used for planning in a two-dimensional space of specified size. The number of robots can be various, but cannot exceed the number of modules. Possible actions are robot moving up, down or sideways, putting a block up on the next block or putting it down next to the block below. It can also lift a block or take it down and catch that block by another robot and shift a module to sides.

In the future, it will be necessary to extend the planning algorithm into three-dimensional space. Then it is needed to incorporate special joints that keep the modules together. Also choosing a state has to be modified to speed up the search for solutions.

For example, the states that are closer to the goal may be preferred. As well the heuristic function can be improved. It should focus more on the movement of the robots. The current estimate of the cost of the way between two blocks may be quite inaccurate, especially for planning with a larger number of robots.

Another aim of the thesis was to design a simulation environment for the MoleMOD system and to test the planning algorithm in the Gazebo simulator. The models of robots and building modules are done, but to test the planning failed due to the strange behaviour of the models of modules. At first, it is necessary to solve this problem; then the simulation environment can be completed.

Finally, the joint controller can be modified to make moving in the simulator more accurate and faster.

References

[1] Jan Petrš, Jan Havelka, Miloš Florián and Jan Novák, MoleMOD - On Design specification and applications of a self-reconfigurable constructional robotic system, ShoCK! - Sharing Computational Knowledge! - Proceedings of the 35th eCAADe Conference - Volume 2, Sapienza University of Rome, Rome, Italy, 20-22 September 2017, pages 159-166

[2] MoleMOD | Jan Petrš. Studio Florián | FLOW [online]. Retrieved from http://www.studioflorian.com/projekty/347-jan-petrs-molemod

[3] Steven M. LaValle, Planning Algorithms, Cambridge University Press, 842 pages, 2006

[4] Gazebo [online]. Copyright ©2014 Open Source Robotics Foundation. Retrieved from http://gazebosim.org/

[5] SDF [online]. Copyright ©2014 Open Source Robotics Foundation. Retrieved from http://sdformat.org/

[6] The Khronos Group Inc [online]. Copyright ©2018 The Khronos. Retrieved from https://www.khronos.org/collada/

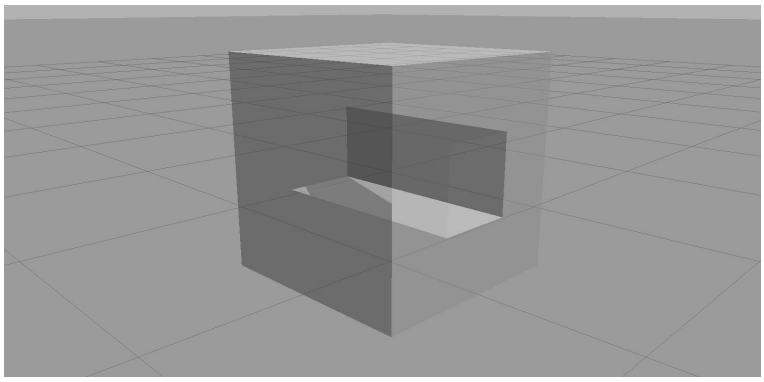
[7] Day 6: Drawing Primitives in OpenGL ES | Hessan Feghhi. Hessan Feghhi [online]. Retrieved from http://hessan.annahid.com/game-development-days/day-6/

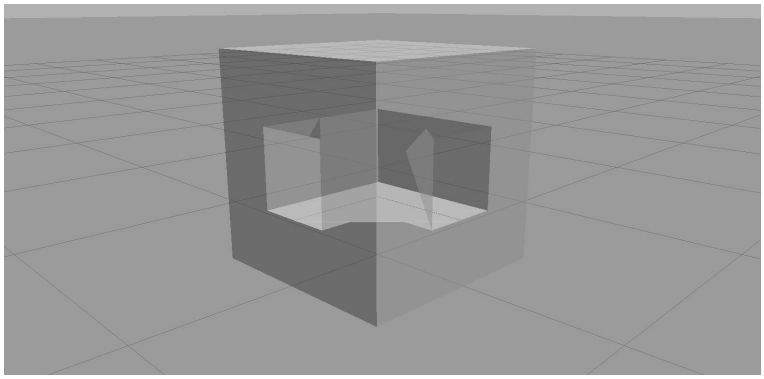
[8] Michal Pěchouček, Milan Rollo, Informed search algorithms [lecture]. In CourseWare Wiki [online]. Retrieved from https://cw.fel.cvut.cz/old/courses/ae3b33kui/lectures/start

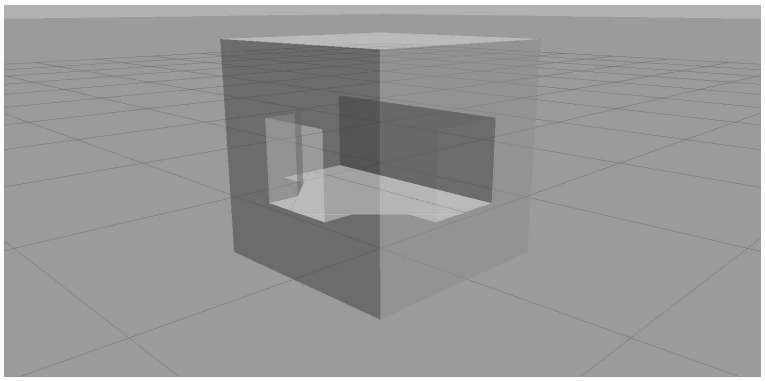
Appendix A

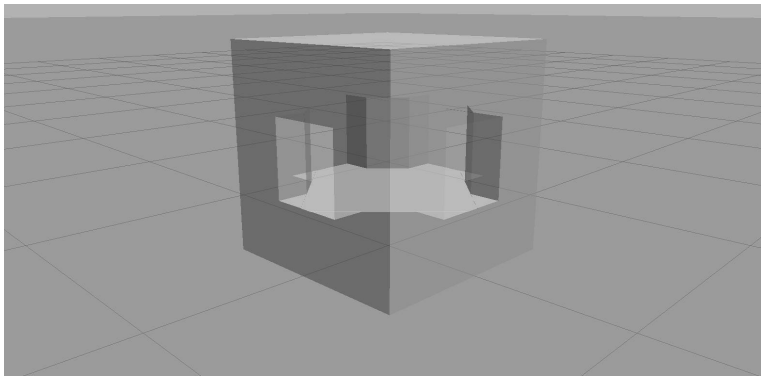
Models of modules

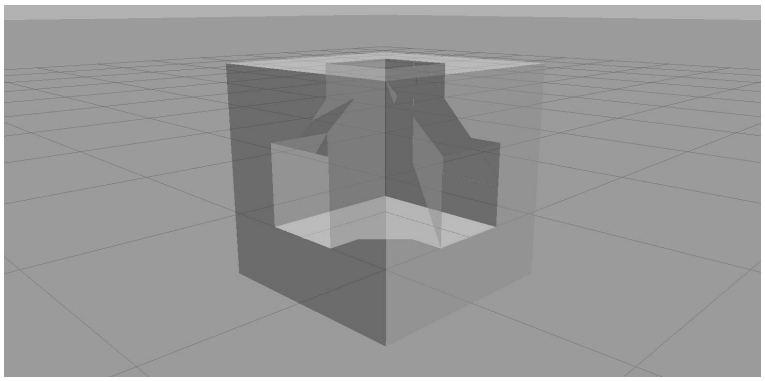
The following figures show some types of tunnel that may be inside building units.

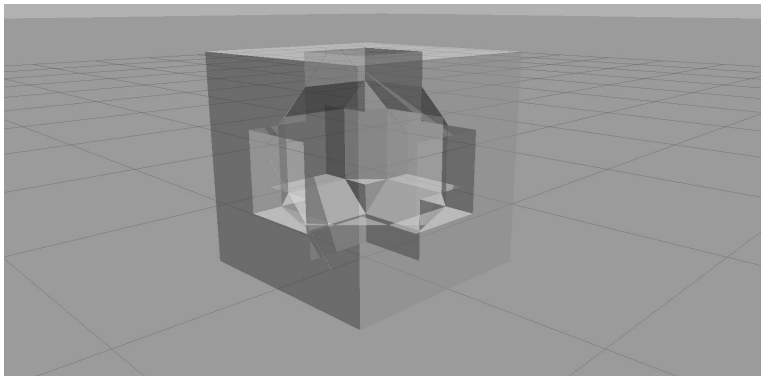








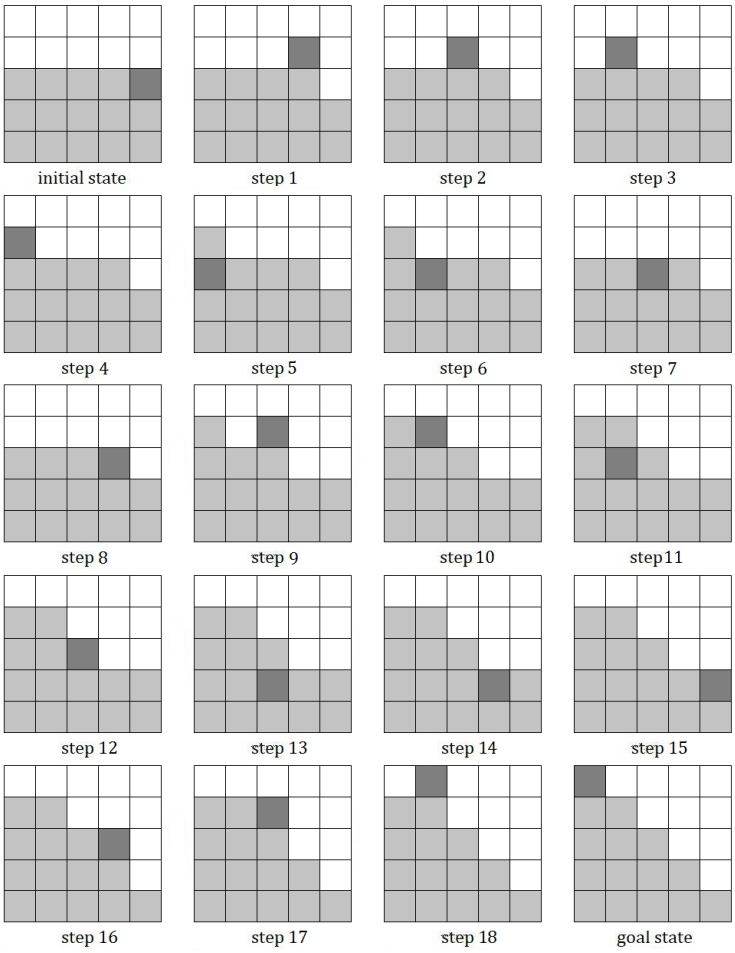


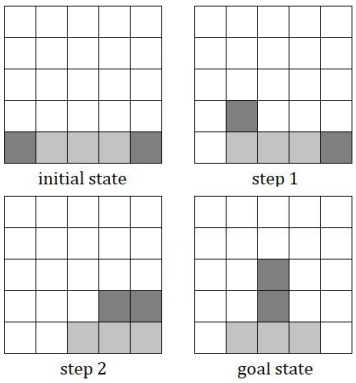


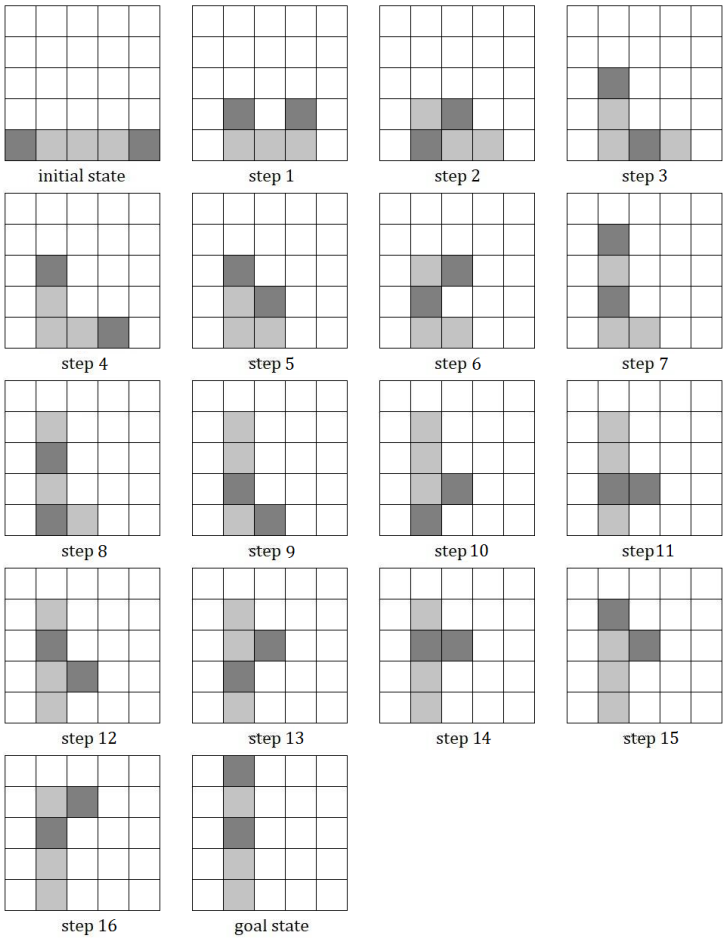
Appendix B

Planning tasks

The following figures show some planning tasks solved by the designed algorithm.




initial state
step 1
step 2
goal state


initial state
step 1
step 2
step 3
step 4
step 5
step 6
step 7
step 8
step 9
step 10
step11
step 12
step 13
step 14
step 15
step 16
goal state

Appendix C

CD content

- **codes**
  - **planning** - all files needed for planning algorithm
  - **collada** - testing meshes
  - **simulation** - world files and plugins for testing models
- **thesis** - the entire thesis in pdf
- **video** - videos from the Gazebo simulator
- **pictures** - pictures of the MoleMOD system and models from the Gazebo simulator