Move the tip to the right – A language based computeranimation system in Box2D

Frank Schröder

November 26, 2017

Germany, unpublished manuscript

Abstract

Not only “robots need language”, but sometimes a human-operator too. To interact with complex domains, he needs a vocabulary to init the robot, let him walk and grasping objects. Natural language interfaces can support semi-autonomous and fully-autonomous systems on both sides. Instead of using neural networks, the language grounding problem can be solved with object-oriented programming. In the following paper a simulation of micro-manipulation under a microscope is given which is controlled with a C++ script. The small vocabulary consists of init, pregrasp, grasp and place.

Keywords: Robotics

Contents

1 Introduction

The most dominant problem in robotics is the optimal control problem:

> “Optimal control theory has received a lot of attention in the last 50 years, and has found numerous applications in both science and engineering” [20, page 1]

Many attempts in literature were started to solve this problem. The reason why most of them fail is because the problem was tried to solve with algorithmic methods. In most cases a reward function is used, not because this is the right way, but because this can described easier in mathematical formulas. Scientists belief, that the optimal control problem can be handled inside the area of mathematics, but they forget that the problem space is too big.

In the following thesis a better approach is presented which utilizes natural language and a symbolic planner for storing domain knowledge into normal C++ sourcecode. This approach has nothing to do with neural networks, reinforcement learning or genetic algorithms instead the human-machine interface gain the attention.

2 Theoretical background

2.1 Interactive control

The normal and most easiest way of controlling a robot is interactive manual control.[14] In most cases this is done with a remote control on which the human operator presses buttons. From the artificial intelligence point of view this paradigm is not the right one because the robot doesn’t react autonomously instead he is dependent on the control signals from a person. But from an engineering perspective this kind of interface is the best way to develop more complex control algorithms.

For a toy rc-car the remote control has no more then 4 buttons: forward, backward, left and right. If the robot is more complicated for example an UAV drone or a biped robot, the number of buttons increases up to 20 or even more. This kind of remote control is very difficult to handle. The aim for the engineers is to build a more easier remote control with less buttons.

The best-practice method in reaching that goal is to use natural language. Instead of using technical oriented buttons like “joint #1 left”, the buttons get a nametag like “move left leg up” or “init”.The new question is, how to connect the tags with concrete actions. The most simple approach is to use joint patterns. A action like “move leg right” will trigger the command “joint(0,2) for 1 seconds”. The connection between a button and a servo-command has to be programmed manually in a software engineering process. After that, the user has a interface like a midi-keyboard which is used by musicians who overlay the keys with

```
standup()
walkright()
walkright()
handdown()
graspapple()
```

Figure 2.1: macro “grasp apple”

```
event: robot is left
action: walkright
event: robit is middle
action: walkright
event: robot is right
```

Figure 2.2: action and effects

a dedicated sample. With this set-up, the operator can play with the robot, he can press buttons in every order and see the immediate result.

In academic literature this concept is sometimes called language grounding, because words are connected to action. A more suitable way is the term “language interface”, because the system works like a mouse as a connection between a natural person and a machine. For the robot it is the same, if the operator knows the command-name or not. For the robot all actions are only numbers which are send to the servo motors. But for the human operator natural language action-names makes the interaction much easier. He can create more advanced movement if hew knows the meaning of each button.

2.2 Symbolic planning

The second part of a robot-control-system after the language-based animation patterns is a symbolic planner. This is well known under the Term PDDL and is able to bring a symbolic system in a goal state. In literature the “monkey and banana” problem is widely discussed,[18] but I want to use for clearness another example. Suppose, the robot should stand up, walk to the chair and grasp the apple. The naive technique would be that the human-operator uses his midi-like keyboard and press the right buttons in sequence. After some trials he will fulfil the task.

The next better approach would be, that the human operator not directly presses keys but writing a small script. This could be the macro in figure 2.1. It consists of textual commands which will trigger animation actions. The problem with this script is, that it is not flexible enough. The robot must be at the right place to start the macro. If the robot is standing now in front of the table, the script would again do walkright to the table, even if makes no sense.

The third and most advanced problem solving technique is to use the above mentioned PDDL-like planner to take the apple. At first, the actionnames get two additional parameters: precondition and effect. With this additional specification, the high-level planner can bring the system to any state. He tries out the commands in different order and see if the conditions matches the rules. The implementation of the planner itself and the precondition / effects declaration must be done manually. It is not possible to use machine learning or similar technique. It is more a question of C++ programming and trial&error.

Classical planning In classical planning the usage of symbolic notation is preferred. Symbolic means, that there is no direct meaning behind the symbols and the planner acts more like a nonsense generator which combines words together. I want to give a short example.

The first step is to declare events which can happen in the simulation. The events could be: robot is left, robot is in the middle, robot is right. The event alone is only a string, it is a symbol which is nothing more than a variable in the program, perhaps this one “event=’robotisleft’”. The symbolic planner itself is not able to connect an event to a real situation, he only knows that a list of events is possible.

The action are operators in the game. An action could be “walkleft” and results in a certain effect. In figure 2.2 is a short logfile given which describes the transition of the system. It looks like a text-adventure game where the player has commands and the system reacts. In the given example the robot is at the beginning left on the screen, and the operator moves him to the right. To bring the system in a certain goal-state a PDDL-like planner is often used. In the simplest form the planner is based on brute-force technique. He tries out different actions and evaluate the results. The best action plan is presented as the solution.

This form of planning works reasonable well because of the symbolic nature of the rules. There is no physics-engine in the loop, the planner is focussed only on event-strings and actions which can be executed. And yes, to create such a symbolic planner, not reinforcement learning is the right way but it’s more the programming and understanding of the given domain. The possible actions of a biped robot are different from the actions of a UAV drone. The best method to specify the rules correctly is to control at first the system by hand with a remote control, note down what the operator has done and formalize the domain in computercode. That’s are very old but effective technique which works in every case. To improve the process, classical software-engineering techniques like bugtracking, versioncontrol systems and wikis are helpful. Sometimes it also helps to increase the number of people who are involved in the process. If more programmer are available the software can be developed faster.

Example [16] gives on page 5 an example how actions are formal described. The figure looks like a C++ class and consists of elements like: length, subaction, motion, involved agent, preconditions and so forth. A simple action can be seen as a instance in an array and the planner has to pickup the actions and bring them into right order. The overall system is called Jack-MOO and has many avatars which are controlled via language.

2.3 Domain knowledge

To program a robot-control-system some external knowledge has to utilize first. In the literature the storing of such knowledge is widely discussed [23][17][1] and most authors come to the conclusion that semantic networks, RDF-triples and frames

[1] on page 10 Schulz criticises RDF triple storage as too ambiguous and that different modelling techniques comes to different results.

are the best way to do that. I have a different opinion. Storing knowledge is not a big deal, normally knowledge is stored in sourcecode directly. And sourcecode is written in one of the major programming languages. RDF-triple is one possibility but simple C++ has the same capabilities. The more important problem is the question from where the knowledge come from. That can't be answered by robotics itself, even the computer science doesn't know it. Because knowledge comes in every case from outside. From normal academics books which describing language, art, physics and so on. This sort of knowledge is in most cases not formalized. Instead it is written in natural language and in some cases also images are used to describe it.

The most interesting subject from robotics knowledge can come from are the arts: painting, music and dance. Between art and Artificial Intelligence is a huge gap. On the one hand we have the traditional history of painting which is more then 5000 years old on the other hand we have the advanced robotics research which is a spin-off from mathematics and is not older than 50 years. To connecting both disciplines is a motivating task.

The resulting machine would be an art creating robot. A robot that can handle oil painting and sheet of paper. That domain is interesting especially because an art-doing machine has no dedicated purpose. He draws lines and colours on a white grid and is not economical productive like a robot who makes cars. A robot who is devoted to art, is focusing the attention away from robotics into the external domain where the knowledge is stored in reality. For painting an image, some kind of knowledge about impressionism is necessary. That domain-specific knowledge is in most cases available, otherwise it would not be possible for humans to paint. All what the robotics programmers have to do is search for art-related information and transfer it into a computer program.[7, 8]

On the formal way, art and artificial intelligence has nothing in common. .The first is oriented on creativity and the second is based on abstract mathematics. But in one aspect both subjects are equal. They want to create something new. The aim of an artist is to paint something which was never drawn before, and robotics engineers tries to build machine which are also innovative. So the general direction in which the disciplines are focused is the same.

In university day-life both subjects are separated from each other. Artificial intelligence and robotics is normally situated into the hard-core science department together with physics and mathematics, while art is often reached on art-schools outside normal universities buildings together with singing, dancing and writing. And in most cases robotics programmer and artists who paint images by hand are not talking to each other. They know from each other but they stay away from the others business. But makes this separation sense? Wouldn't it be bether to teach robotics in an art school and teach painting in a math-class?

There are working examples out there for example, fractals are art which are researched by hard-core mathematicians and sometimes artists are using a standard irobot Roomba robot for creating chaotic painting on the floor. Every discipline alone makes no sense. Computer-programming alone can not answer the question what to do with a computerprogram, and art alone have no knowledge about algorithms. But both disciplines combined together are a powerful new kind of science which could enrich the future.

Storing knowledge, but how exactly? The most common technique in AI literature for storing domain knowledge are:

- Rules
- Frames
- Semantic networks
- Ontologies
- predicate logic
- neural networks with word2vec

These techniques are often cited and are discussed over many years in the past. At first it is important to know, that non of them will work in practice. Behind every technique is a good idea, but real-life demonstration on a concrete example is not available. It is more an academic game to discussing the above mentioned ideas without ask if they are useful or not. The best term to describes this ideology is "academic artificial intelligence":

> "Definition of game artificial intelligence on the other hand is slightly different from that of the (academic) artificial intelligence and it follows other goals." [13, page 2]

The only form of storing knowledge, and which is widely used in robotics challenges and computergames is traditional object oriented programming. The nearest technique on the list is "frames", but object oriented programming can a lot more.

Here a short description of how the transfer of domain knowledge into a computer-program works. At first, some information are needed about the domain. In most cases these descriptions are available in books and in replays of human players who are involved in the domain. As an example the needed knowledge could be 20 books about football playing, some Videos of real football games, and perhaps a simulation as a computergame. This information describes the domain in detail via text, videos and images. The next step is, that a human programmer reads all the material and becomes an expert for this domain. After that, a first semi-autonomous prototype of a robot is constructed which is controlled manually. That means, a human operator with a joystick generates the control signals. The next step is the most important step in the overall process: A language to animation interface. That is the core of the AI system and it is written in C++. It consists of one or more C++ classes which provides functions for solving the game. In the football domain a class "shot the ball" is possible, and inside the class there are methods like "pass ball", "take ball" and so on.

This process which is called in the AI literature "knowledge engineering" is in reality that what ordinary C++ programmer do, who are programming the AI for a commercial computergame. They translate the given information into runnable C++ sourcecode. As tools for improve their efficiency they are using C++ IDEs, bugtracker, wikis, scrum-meetings, versioncontrol, testing of code and a lot of manpower.

Automation of this core process of programming an AI is not possible. So it's bit funny to compare the academic AI literature which talks about frames and semantic networks, with the reality which is 100% normal programming in a high level language. I

want to go a step further: it is not possible storing knowledge outside a C++ program, for example in vocabulary lists, RDF triples or as predicate logic. The reason is, that these alternatives are not powerful enough (a vocabulary list can not be executed in Linux) or they not support very well the adding of new knowledge like predicate logic which is programmed in Prolog. So, C++ sourcecode is the only way to store knowledge. The vocabulary of the given domain is reflected into the class names and method names, the logic of tasks are hand-coded as C++ statements and for-loops. As a consequence, knowledge engineering and C++ programming is exactly the same and other techniques likes Protege ontologies or Frames only support the development of C++ code.

To measure the progress of knowledge engineering the traditional "lines of code" is a good indicator. It means, if the domain is small like a pick&place robot, only a few lines are necessary, if the domain is complex like a football-playing robot much more program code is used for storing the knowledge.

2.4 More about language grounding

Language grounding is the name for the problem of how to connect natural language to animation actions. More or less, until today the problem is unsolved [2]because the natural language describes a task on a knowledge level while lowlevel animation control is oriented on number values. To give a short example: A description in formal language of a task could be "stand up, go to left, sit down". For parsing the phrase the syntax or even grammar level is not enough. It gives no additional advantage if the software knows, that "go to" is a verb and that the comma separates the actions. It is not a question of the vocabulary of "sitting" instead the concept is situated in the domain of puppet theatre. That has to do with art of animation and how characters can move.

To solve the language grounding problem it is necessary to connect lowlevel servo commands with high-level domain knowledge[1], in this example with highlevel character animation knowledge. To store domain knowledge in formal way is also an unsolved problem. Languages like Prolog or RDF–triples are not powerful enough.

The trick is to find a way around the language grounding problem. And here comes interactive animation into the game. Interactive animation means, that a human operator is in the loop. With that trick his domain knowledge will be used for control the character. So the language grounding problem will be transformed into a interface problem. How should a man-machine interface looks like to control a character by words?

In most cases the interface consists of buttons which are named with words. The above phrase "stand up, go to left, sit down" can be executed because the human operator has three buttons which he presses after each other. It is comparable to a midi-keyboard which activates sound-samples but in this case the sample are motion pattern from the avatar.

With this little trick it is possible to create a language interface without formalize the domain knowledge into a database. Instead, in every situation a puppet master will be in the loop and he controls the robot. A second positive effect is, that a working language interface can used again for a symbolic planner. If the system has already buttons like "sit-down" and "go-left", it is easier to extend the buttons to a complete work-flow with actions, preconditions and reward values.

The baseline for advanced robotics system is a interface which connects two sides: on the one hand the interface is utilized by a human with normal words. On the other side the interface controls a robot, he need number values for the servo-motors. How exactly the interface has to look is depending on the domain. A language interface for controlling a soccer robot consists of different vocabulary than an interface for an industrial robot.

Technical realization From sourcecode point-of-view the implementation of a language interface is easy. A simple C++ method is enough which gets at a parameter a std::string with the name of an action, e.g.:

```
action("go to left");
```

Inside the method some lowlevel commands will send to the motor which drives the robot. A little bit more complicate is to implement a complete language interface for a domain. Therefore, the words have to be defined, the possible conflicts and the lowlevel-servo-commands. If the robot has additional dynamic aspects, that the case for biped robots which must balance the legs, than the programming task is a way more complicated. In most cases the domain-problem must be understand first and only as a second task can be implemented as sourcecode. To make it a little clearer: the domain knowledge has to be transferred from art-books about theatre and dance-motions into C++ code and natural language activates the subroutines.

This process itself can not be formalized. It is located in the software-engineering-layer and will be completed with the same tools, as computergames and operating system are programmed. At first, a developer team with skilled programmer is needed, than a concrete goal what the teams want to reach, a bugtracker, a shared programming language and a versioncontrol system [21] are used for communicating to each other.

I'm not sure, if the process of creating a language interface can be done easier with the help of neural networks. In the literature there are some example where instead of programming the system by hand, deeplearning was used to connecting words with actions. The problem with this idea is, that the domain-knowledge is lost, instead the neural networks is trained by abstract reward functions. So I'm in doubt if this will work.

Perhaps the domain-knowledge is stored into the examples which are demonstrated to the neural network. An action consists of a word and a servo pattern. But here is also the risk high, that important information can be lost and at the end the overall system will not work.

2.5 Action Primitives

In the literature the term Movement Primitive is used as least since the 1980s. The first one was [10] who runs "Task primitive" to get complex movement with Utah/Mit dexterous robotics hand. Later, [16] has the same idea and called it "Parameterized

[2]One author believes, that symbol grounding problem has been solved: "Each agent builds up a semiotic network relating sensations and sensory experiences to perceptually grounded categories and symbols for these categories" [19, page 26]

action representation (PAR)". In the 2000s Stefan Schaal introduced "Dynamic movement primitive" [15] for controlling a robot. What these concepts have in common is, that inside the software possible subactions are stored in a database. This database is utilized later by the planner to bring the system to a goal-state.

Let us categorize the idea a little in the context of AI history in general. Widely known is Shakey the robot which at first time has a goal-planner, called STRIPS which worked on a symbolic level. Later, the STRIPS concept was extended to PDDL-planning. On the other hand in robotics, there is a question called optimal control problem, which searches for an algorithm to realize a certain movement. Optimal control is often solved with reinforcement learning, neural networks or even RRT state-space search. Bringing both ideas together is not easy. On the one hand in the PDDL context the items are expressed on an abstract symbolic level. They are nothing more than words. On the other low-end side were optimal control is located the domain is oriented on mathematical values and reward functions. So the idea is to connect both and in the middle are the Movement Primitives.

For getting into detail a look on page 5 in [16] helps. There is a figure given which shows the main elements of a "Parameterized action representation (PAR)". The description looks similar to pddl-like subactions. It is an object instance, consisting of fields like name, precondition, effect, duration, context, affected objects and so forth. It is not important to realize all fields in the own robot control system, the idea in general is remarkable. It can be described as some form of sql-database. A robot has some entries in his sql-database and with these patterns he plans on a abstract level the actions. The advantage is, that the in contrast to "behaviour trees" the order of the actions are no longer fixed, but can be rearranged in every direction. Colloquial this technique is called GOAP (goal oriented action planning)[9] and is widely used by the gaming-industry. According to the above cited literature it is possible to transfer it to the optimal control problem in the robotics domain.

Now I want to explain the background in detail. From gametheory it is known that a game consists of states and action. A state is a node in the gametree, and an action brings the game to new node in the tree. That is the basis for most chess engines, which are searching the gametree for a goalstate, where the opponent is beaten. Movement primitives are heuristics for reducing the possible actions in every gamestate. Only some of them which full-fill the precondition can be executed as a certain point. An example: If the robot's hand is not over the apple, he can't activate the grasp-action. To grasp the apple, the robot's hand must be at first over the apple. So, action primitives are some kind of gamerules which describes the possible movements. With that knowledge, a planner can search the gametree in a shorter amount of time. Like the search procedure in computerchess, the planner wants to reach a certain goalstate.

Movement primitives are a way for a structured storage of heuristics about the game. It is possible for the game designer to modify the movement library so, that the planner can bring the robot to a given state. That can be anything, from grasping an object up to walking to a point in the room. Oh I forget, there was another author in history of AI who used the same principle. [2] is a dissertation from the year 2013 which uses the same principle but this time it is called "diverse actions" and is related to planning with options.

Not all authors are using the same principle in their movement primitive. [15] is based on neural networks, while [16] is based on fixed values in a database. In general there are two possibilities for storing movement primitives. The easiest form is to store fixed patterns. A action is nothing more than joint-movement with a value. It consists of a duration and a parameter. More complexity can be reached if the action movement is seen as a computerprogram which is learned via neural networks or genetic algorithms. In that case (which is the concept of [15]) the action primitive consists of a parameter which fluctuates a chaotic function. And the function drives the servo-motor. In this case, some kind of movement compression is reached automatically and it is well suited for storing motion capture data.

2.6 Planning vs. language interface

For realization of a robot-control-system there are two core modules. First, the language-to-animation interface and second a high-level-symbolic planner. The second one is more popular in the computer-science-community because it works like a software should work. Because a planner consists of conditions which have to be satisfied and a solver which uses brute-force-searching to reach a goal. This kind of software-category is well understood and easy to implement. The bad news is, that a symbolic high-level-planner is the less important part of the overall system. Because a planner requires knowledge which is stored in a formal way. Knowledge about the domain, possible actions, the task-ontology, mathematical functions like inverse kinematics and so on. In case of doubt there is no specified domain-knowledge available so the planner is useless.

The better, but more difficult way for programming robots, is to ignore the symbolic planner and focus on the language-to-animation interface. Especially if the domain is complex it is the better approach for automating tasks. For realizing such systems modelling techniques like object oriented design will help. It is normally used for programming computergames and desktop applications but can be transferred to the area of agent-programming.[3]

A finished languge-animation-interface is equal to an API. In literature this is called a vocabulary list or an ontology and means, that there are actions and subactions defined which can be called from outside. An example action would be "manipulate.grasp(objectA)". This action accesses the class "Manipulation" and the subaction "grasp". Usually, the action-names depends on the domain-model. In the above example the robot is able to grasp and ungrasp objects. The mapping between language to actions is on the programming level equal to call a C++ method.

The technique to realize such systems is twofolded: one the programming side, skills in object-oriented programming with C++, and object-oriented modelling with UML diagrams are necessary. On the domain side, knowledge about vocabulary, domain-specific literature and important cases is utilized. To combine both sides a programming challange is required. For example:

The teams in a coding challenge get the task to program a robot who should search for objects in a maze, collect the objects and brings them to the trashcam. The task is described in normal english and a drawing of the robot, the map and the objects are

given. The transformation from the domain-specific mission to runnable C++ sourcecode is done in a certain amount of time by the programming team. Normally, they will talk to each other about the solution and then implementing the code with C++. At the end, in the evaluation phase, the results from different teams are compared and perhaps not all teams were successful with their robot.

3 Not working principles

3.1 Artificial Life and Learning

In the literature not only manual definition of knowledge via sourcecode is discussed but also a technique called artificial life.[12] The idea is to establish a challenging environment where simulated organisms like fishes must search for food. The fitness can be improved with acquired behaviours which are learnt through trial&error, from other species and with reasoning. Sometimes a learned vocabulary is used.

Can this concept improves the programming drastically? Unfortunately not. The possible actions of the simulated fished are too simple, it is not possible for the meta-algorithm to generate a high-performance path-planner from scratch or ask Wikipedia for knowledge about which food is right. So most of these experiments are not really demonstrations in artificial life, but there are agent-development-kits. The game-engine is preprogrammed and the player can write small code pieces for controlling the robots. Artificial life and vocabulary learning are not techniques for programming a robot-control-system, but it is a didactic concept for explaining game-programming and artificial intelligence to newbies. Perhaps it is comparable to the logo programming language which is also a teaching tool for the classroom.

3.2 DeepLearning

The roots of Cybernetics can be dated back to the science of psychology. The researchers in the 1950s tried to explain human behaviour. Even today, many artificial intelligence projects are not oriented on practical demonstration but in the simulation of the human brain and using psychological models for explaining behaviour. Instead of engineering a system to a certain goal, the researcher renounce to implement knowledge:

> "The system learns to parse and generate commentaries without any engineered knowledge about the English language." [5, page 1]

Another example where the aim is to use DeepLearning for recognizing images, videos and scene to connecting them with language is [22, 6]. Both are using the LSTM-neural network which is sometimes called "recurrent neural network". The reason why deeplearning is utilized is not because the result is so overwhelming good and the videos are correct grounded to the verbs, but the aim is to reproducing human-learning with a neural model. From a practical point of view, neural networks are useless. They acting as a blackbox, and can not fulfil any requirements in the software engineering process. The above cited authors are using them, because they are belief, that humans using also a neural network and they want to reproduce the human ability for language grounding. That was the core idea of Cybernetics, to reproduce the human brain.

Or to explain the strong belief in neural networks are bit colloquial: if a LSTM network is in-the-loop, ,there is no longer a software-engineering process needed. Instead of programming a software, which is able to understand images, the project is psychologically oriented and claims that the implemented recurrent network reproduces biological brain structure, and so it is normal, that the performance of the system is poor ...

4 Implementation

4.1 Simulation in Box2D

Robotics are most realized as physical system. Some engineers belief that a real robot consists of wires, electric current and a motor. This precondition makes it very hard to develop advanced control software because the edit-compile-run cycle is complicated. There are too much single point of failure. The overall system can broke because of the hardware itself, the lidar detection system or the software. To focus only on the control software a pure simulation in a controlled environment is the better alternative. In most cases the systems are programmed as game-like simulations. There is a graphic engine for the rendering, a physic engine for realistic torque calculation and a user-input for query the keyboard for input signals. In the centre of a agent-development-environment the most prominent part is the physics engine. Box2D is a good choice but there are also other engines available which support 3d space.

The Box2D engine is documented online [4] and consists of moving parts (called dynamic bodies) which are connected to each other with joints. A joint is similar to a servo motor and can be driven to the left or to right with variable speed. For handling the real-time aspect of the overall system some kind of threading mechanism is needed. In C++ environment, the most dominant library is part of the language specification and is called "pthread". With threads it is possible to let the GUI run in normal speed and in the background move a joint for 1 seconds to a certain direction.

4.2 Programming

In the domain of micro-manipulation under a microscope there is the standard pick&place problem given. The tool consists of two tips, left and right which can be controlled separately by a human operator. In Figure 4.1 the Box2D physics engine is used as a simulater. The aim is to write a short macro which automates the manipulation.

After activating the macro the micromanipuulator goes at first into rest-position, then it grasps the object and brings it to the goal position. A look into the sourcecode in figure 4.2 shows, that a language-to-animation interface was used. I want to give some details.

At first the pick&place task was described with a vocabulary list:

- pregrasp

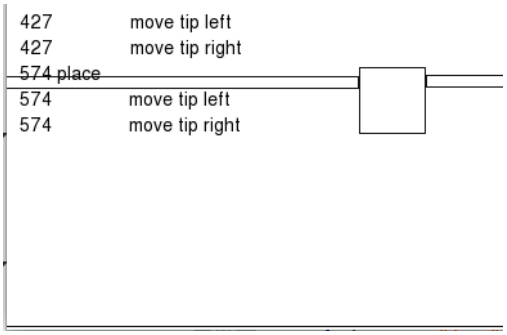

Figure 4.1: Micromanipulation with a tool

```
void action(std::string name) {
  if (name=="init-left") move("left",{200,80});
  if (name=="init-right") move("right",{500,80});
  if (name=="pregraspleft") move("left",{200,245});
  if (name=="pregraspright") move("right",{300,245});
  if (name=="graspleft") move("left",{235,245});
  if (name=="graspright") move("right",{265,245});
  if (name=="placeleft") move("left",{235+50,245-100});
  if (name=="placeright") move("right",{265+50,245-100});
}
void init() {
  std::thread t1,t2;
  myactionlist.add(frame,"init","");
  t1=std::thread(&Behavior::action,this,"init-left");
  t1.detach();
  t2=std::thread(&Behavior::action,this,"init-right");
  t2.join();
}
```

Figure 4.2: Sourcecode

- grasp
- place

Every command activates a certain control pattern of the servo motor inside the Box2d simulation. The overall task was done with executing the vocabulary words in a certain order. With the help of C++ std::threads two actions can be executed at the same time.

4.3 Threading

In the C++ programming language the out-of-the box functionality for running tasks in parallel is called “std::threading”. Instead of start a method of a class after the other in linear fashion, two methods are running at the same time. This is necessary if the left tip and right tip of a micromanipulator should move together while they a holding an object. “std::threading” is an advanced programming techniques which is not easy to realize, but it works stable. The user has many options, he can a task running in background with the “detach()” parameter, or he can wait in the main program until the task ends with “join()”. While running a task in background, the GUI in the foreground must run continuously and update the screen. So, in reality often threads are structured hierarchically: thread1 starts thread2 and thread3 and thread3 starts further threads.

The difficulty in implementation is not because a certain domain is so complex, or that a planning aspect is involved, in most cases the combination of Box2D, C++ programming, threading and executing of scripts was not understand well. In the setting of game-programming normally the game-engine handles the thread-mechanism. Because the most game-engines are undocumented and not available in sourcecode, it is comparable to alchemy.[11, page 263ff]

4.4 Scaling up

The most important question which remains open is how to scale up the shown solution. Currently the system can only pick&place an object, and that only sometimes (if the object is in the right position). Potential failures would be:

- tool is loosing the objects while moving
- object is at a different place
- objects starts to rotate
- other grasps are needed, for example from top to bottom and not from left to right

So there is some mechanism necessary to extend the functionality. It seems too easy, but scaling up works best with extending the sourcecode. The functionality depends in a linear fashion from the “lines of code” metric. With only 10 “lines of code” for move, init, pick and place the system has a limited scope. With 100 “lines of code” which is organized in classes more vocabulary with higher functions can be realized, including a failure recognition- and correction mechanism.

References

[1] Yiannis Aloimonos. Sensory grammars for sensor networks. *Journal of ambient intelligence and smart environments*, 1(1):15–21, 2009.

[2] Jennifer Lynn Barry. *Manipulation with diverse actions*. PhD thesis, Massachusetts Institute of Technology, 2013.

[3] Joanna Bryson and Brendan McGonigle. Agent architecture as object oriented design. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 15–29. Springer, 1997.

[4] Erin Catto. Box2d. *Available fro m: http://www. box2d. org*, 2010.

[5] David L Chen and Raymond J Mooney. Learning to sportscast: a test of grounded language acquisition. In *Proceedings of the 25th international conference on Machine learning*, pages 128–135. ACM, 2008.

[6] Lea Frermann, Shay B Cohen, and Mirella Lapata. Whodunnit? crime drama as a case for natural language understanding. *arXiv preprint arXiv:1710.11601*, 2017.

[7] Shunsuke Kudoh, Koichi Ogawara, Miti Ruchanurucks, and Katsushi Ikeuchi. Painting robot with multi-fingered hands and stereo vision. *Robotics and Autonomous Systems*, 57(3):279–288, 2009.

[8] Thomas Lindemeier, Jens Metzner, Lena Pollak, and Oliver Deussen. Hardware-based non-photorealistic rendering using a painting robot. In *Computer Graphics Forum*, volume 34, pages 311–323. Wiley Online Library, 2015.

[9] Edmund Long. *Enhanced NPC behaviour using goal oriented action planning*. PhD thesis, University of Abertay Dundee, 2007.

[10] Paul Michelman and Peter Allen. Forming complex dextrous manipulations from task primitives. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 3383–3388. IEEE, 1994.

[11] Druhin Mukherjee. *C++ Game Development Cookbook*. Packt Publishing Ltd, 2016.

[12] Marcio Lobo Netto, Marcos Antonio Cavalhieri, and Luciene Cristina Rinaldi Rodrigues. From genetic evolution of simple organisms to learning abilities and persuasion on cognitive characters. *RITA*, 12(2):31–60, 2005.

[13] Damijan Novak and Domen Verber. Real-time strategy games bot based on a non-simultaneous human-like movement characteristic. *GSTF Journal on Computing (JoC)*, 3(2):43, 2013.

[14] Edwin Olson, Johannes Strom, Ryan Morton, Andrew Richardson, Pradeep Ranganathan, Robert Goeddel, Mihai Bulic, Jacob Crossman, and Bob Marinier. Progress toward multi-robot reconnaissance and the magic 2010 competition. *Journal of Field Robotics*, 29(5):762–792, 2012.

[15] Stefan Schaal. Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In *Adaptive motion of animals and machines*, pages 261–280. Springer, 2006.

[16] William Schuler, Liwei Zhao, and Martha Palmer. Parameterized action representation for virtual human agents. *Embodied conversational agents*, 256, 2000.

[17] Stefan Schulz, Ludger Jansen, et al. Formal ontologies in biomedical knowledge representation. *Yearb Med Inform*, 8(1):132–46, 2013.

[18] Gerardo I Simari, Diego R García, and Gabriel R Filocamo. The monkey and bananas problem revisited: a situation calculus approach. In *IX Congreso Argentino de Ciencias de la Computación*, 2003.

[19] Luc Steels. The symbol grounding problem has been solved. so whats next. *Symbols and embodiment: Debates on meaning and cognition*, pages 223–244, 2008.

[20] Emanuel Todorov and Weiwei Li. A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *American Control Conference, 2005. Proceedings of the 2005*, pages 300–306. IEEE, 2005.

[21] Linus Torvalds and Junio Hamano. Git: Fast version control system. *URL http://git-scm. com*, 2010.

[22] Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond Mooney, and Kate Saenko. Translating videos to natural language using deep recurrent neural networks. *arXiv preprint arXiv:1412.4729*, 2014.

[23] Stefan Zander and Yingbing Hua. Utilizing ontological classification systems and reasoning for cyber-physical systems. In *Karlsruhe Service Summit Research Workshop, Feburary*, 2016.