Bachelor thesis
Computer Science

Simulating Lego Mindstorms EV3 robots using Unity and Python

Author: Leo Cornelissen – s4606566
Supervisor: prof. dr. J.J.M. Hooman
Second supervisor: drs. H.C.W. Kuppens

Date: 23-01-2019

Radboud University Nijmegen

ABSTRACT

The Lego Mindstorms line of ‘toys’ has been popular in computer science education for a while; with its highly configurable set-ups and support for all kinds of programming languages. The latest edition, called the EV3, is able to run custom Linux operating systems. Simulating these robots could save schools money and time. Furthermore, the idea of a programmable robot could be extended to offer even more benefits. This thesis handles the creation a framework to mirror the process of controlling a robot with custom code in simulation. Python is used to write programs for the robot/simulator, and the game engine Unity is used to create a digital environment for the robots to interact with. To start development, a list of requirements is created to narrow down the scope of the project. These requirements are then compared to other EV3 simulators are available online.

Contents

1 Introduction

1.1 Simulating embedded systems

An ‘embedded system’ is usually defined as a piece of hardware that is part of a bigger system or device. This hardware contains microprocessors and other chips that are focused on and optimized for a small number of tasks. Verifying and testing such an embedded system proves difficult: it can be impractical or time consuming for various reasons. Sometimes hardware and software for embedded systems are developed in parallel, making it impossible to debug and test before the physical hardware is finished. Therefore, it is often useful to create software that simulates (Engblom, 2008) the hardware to get information about the expected performance and behavior of an embedded system. Those simulations can range from accurate reimplementation of the actual hardware, to a more surface level reimagining of the system. This principle is also referred to as ‘virtual prototyping’ or ‘digital twins’.

However, reimplementing systems can be used for more than just industrial purposes. One example is using digital twins for entertainment and preservation: ‘emulators’ for old video game consoles read game code that is only executable by the original console, and convert them for use on modern hardware (Wikipedia contributors, n.d.). Furthermore, it is an increasingly popular concept in education: The principle of teaching students with assistance of simulation has applications among the fields of molecular science (Allen, 2007), medicine (Beer, Grit, Good, & Gravenstein, 2001) and construction (Purnus & Bodea, 2015).

1.2 Lego Mindstorms

This thesis addresses virtual prototyping for the latest Lego Mindstorms robot: the EV3. Lego Mindstorms is a platform that allows individuals to control a programmable and customizable ‘brick’ (Lego, 2018). This brick supports a collection of additional connectable modules for all kinds of objectives: examples include LED’s, motors, color sensors and buttons. These sensors and actuators can then be commanded by writing code and uploading it to the brick. EV3 is the third and latest entry in the Mindstorms line as of 2013, which includes an improved brick that locally runs Linux.

The Mindstorms series has proven to be popular in education (Klassner & Anderson, 2003), where it is used as a tool for teaching programming and computer science subjects. There are, however, several drawbacks that come with this method of teaching. Mindstorms robots are costly, which means it is hard to provide every student with their own robot. Frequent switching of the robot between students reduces the available time for each individual to actively work with it. Additionally, working with Mindstorms can be inefficient due to the significant effort it takes to repeatedly upload code to the brick.

Considering the popularity of the Mindstorms series in education and its drawback, simulation software for the EV3 series would be helpful. Because the main concept of these robots is the real-life interactivity (driving around, flashing LEDs, handling physical input, etc.) the simulation aspect would be different from usual virtual prototyping, as it requires a fully visual representation of the brick and its connected modules.

1.3 Goal

The main research question of this thesis is as follows:

"What is an effective method of developing an open-source simulation framework for the use of Lego Mindstorms EV3 in education?"

Sub questions handle different aspects of this enquiry:

a) "What are the primary requirements for educational use of this software, and how can they be sufficiently met?"
b) "What previous work exists for this problem field?"
c) "How is this realized, keeping extendibility and future developers in mind?"

2 Literature and approach

Answering sub question (*a)* will narrow the scope of what are considered necessary features for this thesis. The educational aspects will be researched to provide the base requirements.

Currently, Mindstorms are used in multiple levels of education with success: high school students perceived less difficulty in learning to program with the help of a robot (Mason, 2013), elementary school children show more attention when robots are introduced (Zygouris, et al., 2017) and undergraduate student have indicated that the robots helped them understand practical applications of coding (Gandy, Bradley, Arnold-Brookes, & Allen, 2015).

One - and perhaps the most important - benefit of educational simulation software is the 'hands-on' aspect, also referred to as experiential learning: "Experiential learning enhances the learner's critical thinking, problem-solving, and decision-making skills, all being aims of teaching" (Pasquale, 2015). This is in part because of direct application of learned knowledge in situations where it applies. A Mindstorms simulator would teach learners coding by showing a visual and practical example of how those skills can be used. This is similar to use of real robots in education. When considering features for this thesis, the importance of experiential learning should be central.

Experiential learning is described in four stages by David Kolb (McLeod, 2017): first, the learner has some kind of experience. Then, the learner reflects on and subsequently learns from that experience. Finally, the learner takes that new knowledge and applies it, looping back to the first stage. Some of these four stages can be made more accessible for learners in simulation software. A digital environment can boost the effectiveness of reflection by showing information that would be hidden in a real-life situation; e.g. internal values of a robot. Furthermore, applying new knowledge from previous experiences can be far more efficient when working entirely in simulation.

Availability is another aspect that is important when developing educational software. Considering the disadvantage of lower socio-economic circles when it comes to education technologies such as Mindstorms (Anderson, 2005), the software should be freely available on multiple platforms. Moreover, the target audience must be expanded by distributing the software in an accessible manner.

Answers to sub question *(b)* will be based on other work on the stated problem in this thesis, and the associated solutions to common problems. Analyzing previous work and their strong/weak points is can give insight in how and why certain choices are made in the development of robot simulation software. This will involve looking for similar software projects on the internet, and comparing them. Research conducted by Giel Besouw shows and compares various solutions for the NXT entry in the Mindstorms series (Besouw, 2018).

Finally, the requirements and choices made with the answers to sub question *(a)* and *(b)* are realized by answering sub question *(c)*. A custom solution to the stated problem will be developed by iterating prototypes of an EV3 simulation software. A demo of this software will be made available for intermediate evaluation. The set of features that would be implemented by the end of this thesis should be sufficient for educational use, and provide a straightforward skeleton structure for future developers to complete.

3 Requirements

Various categories of requirements are considered. First, the target platform of the software will be investigated. Next, the method of controlling and coding the simulation is considered. Aspects involving the simulation and its properties are finally narrowed down. All of these properties should conform to the educational nature of the simulation.

3.1 Platform

About 78% of all personal computers run Windows, while around 14% runs macOS. Various Linux distributions cover most of the remaining share (statcounter, 2018). However, among people with an interest in programming, this rate differs: Linux-based operating systems are frequently preferred for their openness. A cross-platform based solution is a strong requirement because of this.

3.2 Coding

The simulated robot would need some sort of input to be commanded by a user of the software. A real-life EV3 brick runs code that in a programming language that is supported by its installed operating system. Consider these two distinct methods to imitate this concept in simulation:

- The simulated robot acts according to a representation of a program that a user has created within a visual programming environment.
- The robot is controlled by a program written by a user in a textual programming language.

The latter method has a major benefit: a programming language can be used that is supported by an EV3 operating system. This makes interchangeability between code written for the simulation and a real-life EV3 brick possible.

Ideally, coding in combination with the simulation software should not require experienced users to learn a new programming language. That in turn allows users to apply their new skills in other areas, in accordance to the hands-on learning experience. The EV3 brick runs Linux locally, meaning hobbyists have free reign over their machine. This made it possible for various programming languages and operating systems to be ported for commanding the EV3 robot. The following are some of the most popular available:

- **`leJOS`** is a Java Virtual machine ported to a few robots including the EV3, making it possible to control the brick using Java code. **`leJOS`** is available for free.
- **`RobotC`** (Robomatter, 2005-2014) enables licensed users to write C-like code for various robot systems including the EV3.
- An open-source Debian-based operating system called **`ev3dev`** (ev3dev, 2018). This OS can be started from an SD card inserted in the EV3 brick. Its supported languages include Python, Java, C++ and more.

Python has a number of benefits for both the users and developers. It is one of the most widely used languages to teach programming, and a huge amount of information and support is available for the language. Combined with the huge number of devices and operating systems that support Python, these aspects are useful for the students using an educational piece of software. On the development side, Python development for the EV3 is made possible by using a library called `ev3dev2` (python-ev3dev2, 2018). Included in this library are different classes and methods that can be imported and called to in a Python script, representing the different modules connected to the EV3 brick.

3.3 Simulating

There are various tools to create a visual environment with objects moving around according to developed code, with differing levels of complexity and incorporated features. These tools either have two-dimensional or three-dimensional capabilities. A 2D environment in this would mean the simulation has a top-down view, while 3D allows for full 360-degree camera system. Another aspect that is necessary in this project is a support for physics and collision, meaning the simulated robot would be able to physically interact with other objects in the environment. Additionally, there is a need for the developed visual environment to be functional on multiple operating systems. Game engines are an example of such tools, usually containing all the features that are necessary for game development (such as the examples above).

Unity (Unity Technologies, 2018) is a well-documented game engine with 2D and 3D functionality, that includes an editor for multiple platforms. This editor contains all the necessary tools to make games and other interactive programs, such as physics and collision systems. Most developers write in C# or JavaScript in combination with Unity. Applications created in Unity can be exported to multiple platforms with identical functionality: Windows, macOS, Linux, Android, iPhone and many more.

Unity works with so-called `GameObject`s. These `GameObject`s have different components added to them. Components hold information on variables such as position, rotation, physics, 3D representation and much more. It is also possible to add scripts to a `GameObject`; making it possible for developers to control aspects of that `GameObject` and other variables through coding.

Unity applications run with a variable framerate, which changes depending on the CPU or GPU load. Normally this framerate hovers around 60 frames per second.

3.4 Customizability

The Mindstorms platform is in part successful because of the extendibility of both hardware and software. The Lego structure that all connectable Mindstorms modules adhere to facilitates personalized robots, further extending the practical uses. Standard Mindstorms modules and Lego pieces are combined to create a solver for Rubik's Cubes (Gilday, 2016), in addition to a fully automatic assembly line (Superlegosam, 2014) and a looming machine (Zając, 2013). In this spirit, customizability within and outside of the simulation should be an essential part of the software. This means granting users the ability to customize their simulated robot/environment and engineering the software in such a way that it can be extended upon in a clear way. The former requires a 'building mode' within the software, such that the users can connect modules in their chosen position on the brick and edit the environment that the robot exists in. The latter would at minimum involve creating a framework that facilitates the addition of new (possibly fictional) modules to be used in the simulation, without having to rework the entire code.

4 Other solutions

There are five EV3 simulators found on the internet:

- Virtual Robotics Toolkit (Cogmation Robotics, 2018), a Windows-based 3D simulator that requires users to buy a one-time license.
- Robot Virtual World (RoboMatter, n.d.), similar specifics as Virtual Robotics Toolkit.
- QEV3Bot (Simmons, 2018), a free 2D simulator for Windows.
- Open Roberta Lab (Fraunhofer IAIS, 2018), a web-based 2D simulator that is free to use.
- TRIK Studio (TRIK, 2014), a fully Russian 2D simulator software for Windows/Mac/Linux.

Since the TRIK Studio application and documentation is fully Russian, it is excluded the discussion below. Besides being cross-platform, it doesn't seem to have any outstanding features not present in other available simulators.

4.1 Coding

The four remaining EV3 simulators support the following programming solutions:

- Visual Robotics Toolkit implements a visual programming environment based on the official Lego Mindstorms programming software (Lego, 2018). This software allows customers to create pseudo-code to command their robot. Users can place different blocks that control the program flow, sensors, light, sound and motors. Additionally, there are advanced blocks which are capable of e.g. file transfer and Bluetooth connections.
- Robot Virtual Worlds also implements the official Lego visual programming software, besides allowing for textual and visual RobotC programming.
- QEV3Bot Simulator allows users to write full RobotC code manually.
- Open Roberta Lab has a more comprehensive visual programming compared to the official Lego Mindstorms software.

These simulators all allow users to save and load their created programs. Visual programs created in Open Roberta Lab can be saved to source code for a variety of robots and target operating systems. It supports two EV3 operating systems: leJOS EV3 and ev3dev. Open Roberta Lab can also export and import produced visual code as a *.xml file.

These source files can all be uploaded to an actual EV3 robot with the target OS. Excluding QEV3Bot, these simulators are capable of connecting to the real-life brick through a cable or a wireless connection to upload programs directly.

4.2 Platform

Virtual Robotics Toolkit, Robot Virtual Worlds and QEV3Bot only work on Windows. Open Roberta Lab functions in any modern browser. A web-based simulator has a few benefits: no installation is necessary, and as long as a operating system has a modern browser it is almost guaranteed to work without configuration. A major drawback is the difficulty of web development, especially when trying to create a fully dynamic (3D) graphical simulation.

4.3 Simulation

Virtual Robotics Toolkit and Robot Virtual Worlds have a fully three-dimensional simulation, while the others have only two-dimensional top-down views. Both of the 3D environments implement a physics engine, allowing the simulated robot to interact with various objects.

4.4 Customizability

Virtual Robotics Toolkit can import 'LDraw' models that are created using the Lego Digital Designer (Lego, n.d.). These models represent digital structures built using actual Lego bricks. Those imported models can then be used in the simulation as a replacement of the standard robot. Robot Virtual Worlds does not have this functionality; one can choose from a set of predefined robots. QEV3Bot has the ability to configure up to four sensors from a set of eight available sensors. Open Roberta Lab has no customizability options for the robot.

All of the available robot simulators come with various environments for the robot to work in. Robot Virtual Worlds can even important custom models for use as surroundings.

5 Design

The requirements and personal preferences lead to the choice of tools: Unity with C# is used on the virtual prototyping side, while Python programs are used to command both the actual EV3 bricks and the simulation. First, a general overview of the structure of these programs are given. A clarification of specific choices and their alternatives is provided in section 5.3.

5.1 Framework for communication

A real-life EV3 brick communicates with its hardware to set and get values according to the execution of a program. Figure 1 demonstrates the progress with various example modules connected to a brick.

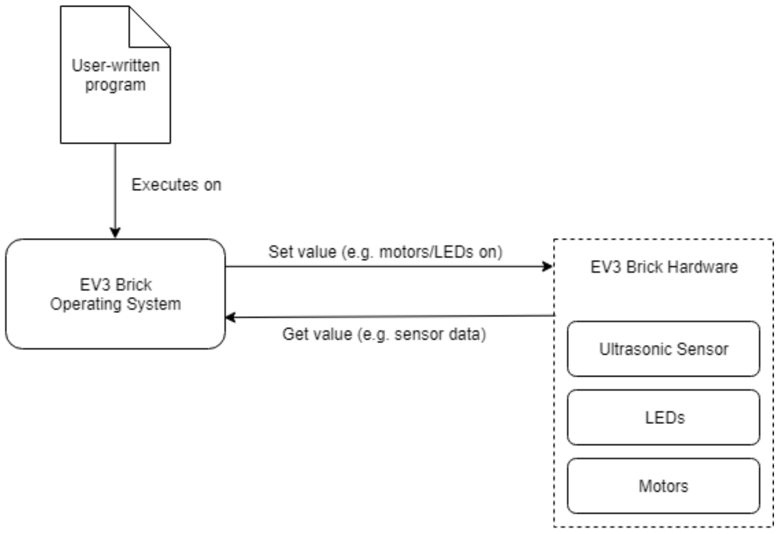

User-written program
Executes on
EV3 Brick Operating System
Set value (e.g. motors/LEDs on)
Get value (e.g. sensor data)
EV3 Brick Hardware
Ultrasonic Sensor
LEDs
Motors

Figure 1: a reduced representation of the EV3 hardware and software interaction.

This principle must be replicated in the simulation. A problem arises when programming languages are not compatible: the simulation software is developed in Unity using C# and Python is used to control the simulated robot. One way to overcome this problem is to implement an interpreter within the simulation software, allowing the simulated brick to natively run compatible code. A more attainable solution is to find a way to communicate between the simulation software and a user-written program that is executed on the same machine. This would function as displayed in figure 2.

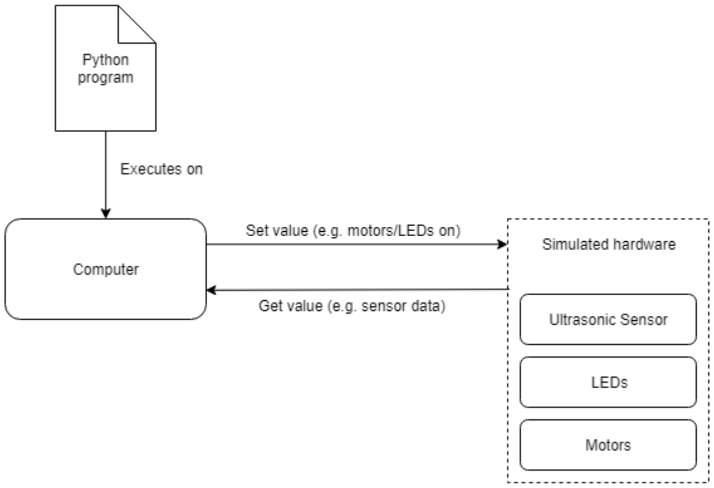


Figure 2: a reduced representation of the simulator and a user-written program interaction.

This approach has the two different processes running in tandem, with some method of communication. The most straight-forward way to implement this is to set up networking sockets on both ends. To send and receive information, data would need to be convertible to and from bytes. However, the communication between two different programming languages requires this specific method of encoding to be standard across platforms. JSON is a multiplatform solution that converts class objects to convenient string representations (and vice versa).

Since the Python programs in this scenario does not run on the actual brick with the **ev3dev** OS, the functionality of the **ev2dev2** Python package will need to be reimplemented for this purpose. Rewriting the **ev3dev2** package, using the same method and variable names, also conforms to the requirement of Python code executing identically on the simulation and the real-life EV3.

The hardware of the EV3 robot can be simplified as a collection of (sensor) modules. Every such module has a unique ID, type and list of values. For example, a brick with two LEDs and an Ultrasonic sensor connected to it can be described as demonstrated in figure 3.

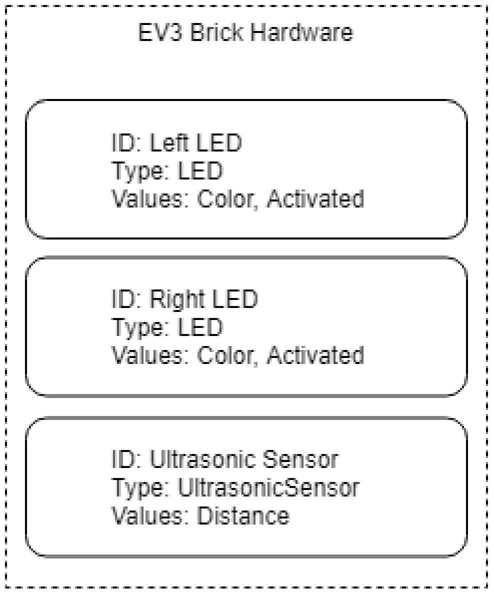


Figure 3: a simplified representation of EV3 hardware.

A distinction is made between 'input' and 'output' values. Input values are assigned by the Python program, while output values are retrieved from the simulation. The 'Color' value of an LED module counts as input, and the 'Distance' value of an ultrasonic sensor as output.

The goal is the reach structural parity between the simulation (in C#) and **ev3dev2** regarding the modeling of the hardware. This way, a correspondence can be achieved between the two processes beyond their language difference.

Implementing this model on the simulator side creates a 'virtual' brick representing the current state of the simulated hardware. The simulation can then run according to these variables. On the Python side, there is no locally stored representation of the brick. The re-implemented **ev3dev2** methods directly communicates with the simulation to get and set values.

A general structure must be constructed for communication between the two processes. The Python program being executed (essentially replacing the EV3 brick operating system) acts as the client in this relationship, whereas the simulated hardware acts as the server. First, the connection must be started. This is done through a simple handshake: the client sends its name, to which the simulator responds with its own. After this, the communication can start.

The getting and setting of variables is done by sending either a ‘request’ (for output values) or ‘command’ (for input values) to the simulated hardware. A request contains information about a certain value of a module, and expects an answer containing that requested value. A command tells the simulated hardware that some value should be updated. This is illustrated in figure 4. A complete overview of the framework is shown in figure 5.

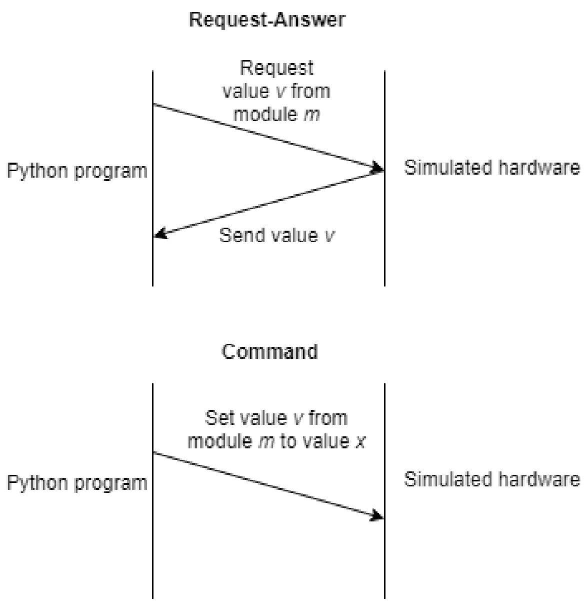


Figure 4: the messages that are used in communication between the simulator (server) and Python program (client).

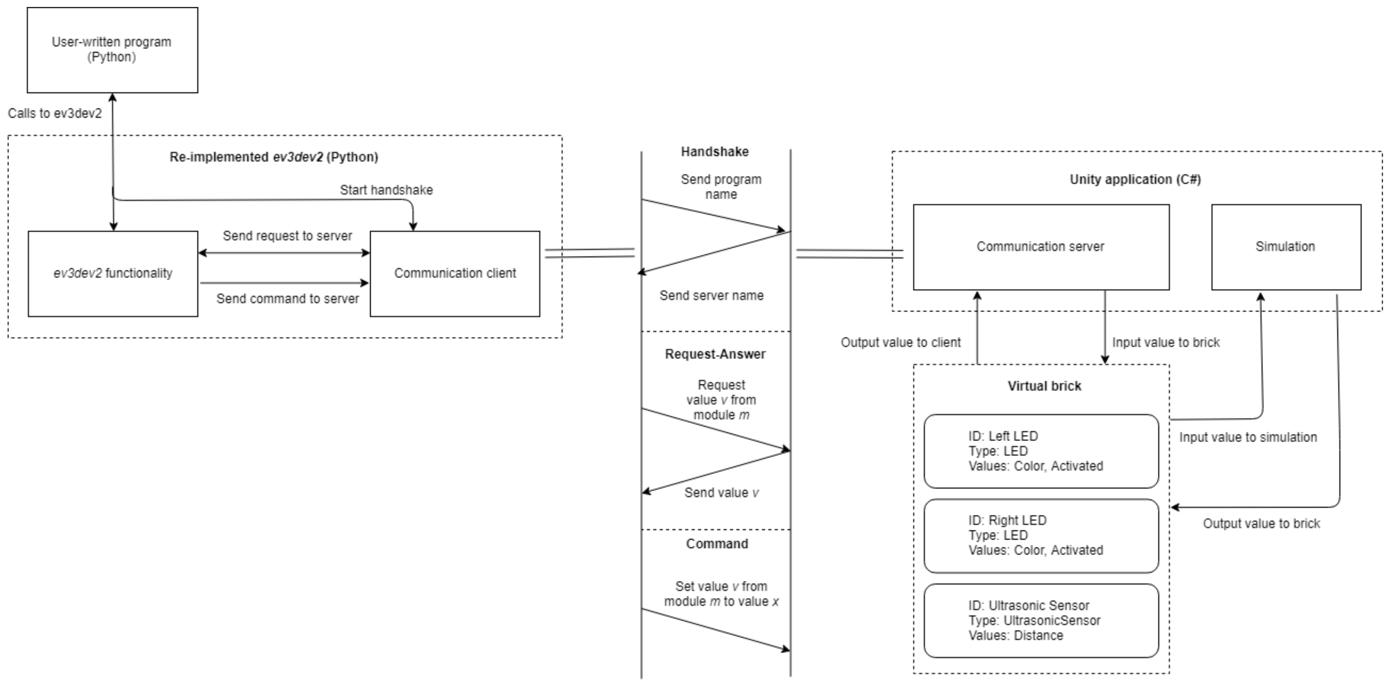


Figure 5: an overview of the EV3 simulation software.

5.2 Modes of operation

5.2.1 Building mode

To make the simulator adhere to the customizability requirements, a ‘building mode’ will be constructed. This mode makes it possible for users of the simulation software to construct their own robot with the implemented modules available. When the full robot is constructed, it should be able to be used in a subsequent simulation. The data that represents a certain robot should be convertible to a file that can be loaded.

5.2.2 Action mode

In the action mode, the simulation happens. This is where the communication framework starts to work with a created robot. Information on the current state of the connected modules should be shown on screen.

5.3 Design choices

There are two problems that need to be solved in order to have functional communication between two programs: the method of communication and serialization format need to be chosen.

The concept of inter-process communication is not a new one, and there are multiple approaches:

- One can create a local file that is being written to and read from. The biggest drawback of this form of communication is the dependence on ownership. If one party (for example, the simulator) claims a file for use, it will not be accessible by any other party. This can cause problems in a situation where different parties keep switching ownership. Furthermore, the precise mechanisms concerning file systems are different for every OS, causing more potential problems.
- Pipelines are somewhat similar to the current method. Data flows from one process (output) to another process (input). With some tinkering this can also work bidirectionally. This is the standard way of communicating between processes. However, due to the multiplatform and extendable nature of this framework, using this approach would be too obtuse. Like files, this methods behavior is OS dependent too.

Socket networking comes with some benefits. Sending and receiving values is practically instantaneous when hosting locally, and could even be extended to allow remote access. Multiple clients can be connected to one server; in this case possibly multiple robots to a single simulation software instance.

Using JSON is not the only way to tackle the serialization problem. There are various other formats freely available for developers to use, such as XML, YAML and CSV. There isn’t much difference between these formats, and only basic serialization is needed in this case. JSON support comes with Python in the form of the **`json`** library. An additional framework is needed for JSON integration with C#: Newtonsoft’s **`Json.NET`** (Newtonsoft, n.d.) is a solution that is fully open-source.

The structure of the messages send through networking is in line with the expected behavior concerning server-client communication. The client initiates the contact, and subsequently sends requests to the server. The server handles these actions appropriately.

6 IMPLEMENTATION

6.1 FRAMEWORK FOR COMMUNICATION

6.1.1 Python side

The first step to implementing the previously mentioned framework, is to re-implement the **ev3dev2** package. For every sensor/actuator for the EV3 that is supported by this library, a virtual counterpart is to be created. This makes it possible to set up the communication framework later on. These virtual modules are manipulated with calls to the re-implemented ev3dev2, using the original method and variable names.

The user-written program needs to be located in the same directory as a folder called **ev3dev2**. This folder then needs an empty file named **__init__.py** to indicate it is a Python package. Various files and folders (with their unique **__init__.py** file) can then be added to further add functionality and nesting. For example, the following imports are requested by a Python program called **Program.py**:

```
from ev3dev2.display import Display
from ev3dev2.led import Leds
from ev3dev2.sensor.lego import UltrasonicSensor
```

These imports correspond to the created file structure (where the ***.sensor.lego** import is located in a nested folder):

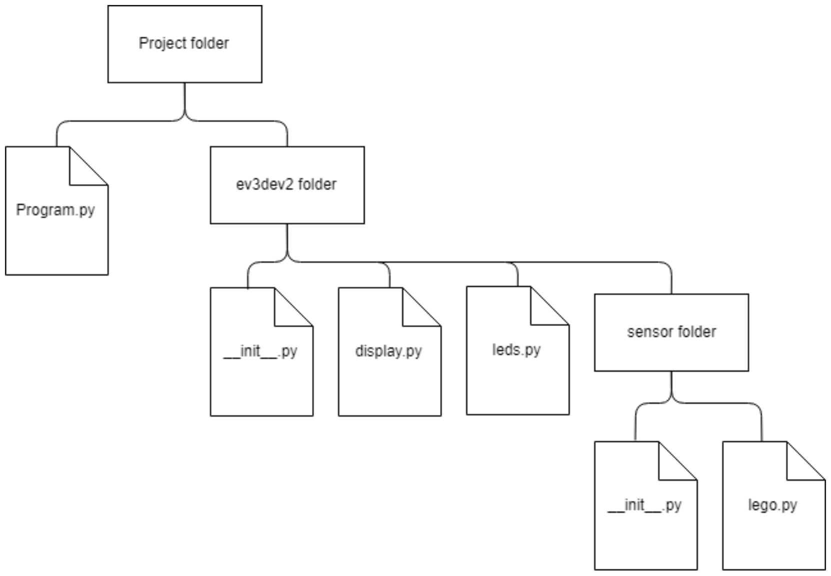


Figure 6: the recreated file system inherited from the 'ev3dev2' package.

A file, such as **`lego.py`**, contains at least one class that represents a module (in this case an ultrasonic sensor). To facilitate the communication framework, this 'module class representation' inherit from a single class: the **`ModuleClass`**. The **`ModuleClass`** contains properties to identify the specific module (ID and type) and two functions that handle the getting and setting of values (**`get_value`** and **`set_value`**). This inheritable class is stored in the **ev3dev2** folder in a file called **`module_class.py`**. Coding a specific module class is straightforward from that point:

```
from ev3dev2.module_class import ModuleClass

class UltrasonicSensorModule(ModuleClass):
    def __init__(self, id, module_type):
        ModuleClass.__init__(self, id, module_type)

    def get_distance(self):
        return float(self.get_value("Distance"))
```

This class can be initialized (with the ID and type parameters) and its functions can be called. The initialization and handling of classes inheriting from **`ModuleClass`** is hidden in the re-implementation of **ev3dev2** to become synonymous with the normal usage of **ev3dev2**. Combined, this is the re-implementation of the ultrasonic sensor module contained in **`lego.py`**:

```
class UltrasonicSensor:
    sensor = UltrasonicSensorModule("Sensor 1", "UltrasonicSensor")

    def value(self):
        return self.sensor.get_distance()
```

The ultrasonic sensor is now ready for use in a user-written Python program with the same syntax as the original **ev3dev2**:

```
from ev3dev2.sensor.lego import UltrasonicSensor

us = UltrasonicSensor()
distance = us.value()
print(distance)
```

The next step is providing functionality to the **`get_value`** and **`set_value`** functions contained in **`ModuleClass`**. These functions depend on the C# simulator, for which we first need to establish a communication channel. For this purpose, a separate file (**`unityev3`**) and class (**`ClientInfo`**) that handle communication are created. This class has a static function **`get_main`** that returns the currently running client socket. If the socket is not yet up and running, it will be set up and returned.

This piece of codes starts a socket client looking to connect to the local host on port 24197:

```
port = 24197
host = "127.0.0.1"
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print ("Waiting for simulator..")

s.connect((host,port))

print ("Found simulator!")
```

This code waits until another party (the simulator) accepts the connection. If no such party can be found, execution will be stalled indefinitely to prevent code running with no simulator attached. After a connection is formed, the handshake (as stated previously) will be started.

For timing purposes, the Python program should only start executing when the simulator is connected and ready. The **`__init__.py`** file in the root folder of ev3dev2 can be edited to execute code upon import. Because the imports are stated at the start of every Python program, we can use this method to stall the program until a socket is correctly set up:

```
from ev3dev2.unityev3 import ClientInfo

ClientInfo.get_main()
```

After the connection with the simulation is established, **`get_server_value`** and **`set_server_value`** functions of **`ClientInfo`** can be used to communicate. The former expects a **`Request`** to send and returns the **`Answer`** it received, while the latter expects a **`Command`** to send. These classes represent the messages used for communication as previously stated and are defined as such:

```
class Request:
    def __init__(self, id, module_type, value_name):
        self.ModuleID = id
        self.ModuleType = module_type
        self.ValueName = value_name

class Answer:
    def __init__(self, value):
        self.Value = value

class Command:
    def __init__(self, id, module_type, value_name, value):
        self.ModuleID = id
        self.ModuleType = module_type
        self.ValueName = value_name
        self.Value = value
```

Additional functions are added to those classes to convert between JSON strings and bytes. These bytes are then sent to the server alongside some information about the message. If the message was a **`Request`**, the client will then start listening to receive an **`Answer`** JSON string.

If the (C#) server is not ready to receive a message by the client, the Python program will (deliberately) stall until it is. No Python code should be executed when the simulator is not responding, preventing timing and simulation errors. This makes the server responsible for the client program execution flow. This has an interesting property: it can be used to model the delay between an actual EV3 brick and its hardware. This delay is necessary due to the way game engines execute code every frame, instead of continuously (more on this in the Unity Side subsection).

Finally, the **`get_value`** and **`set_value`** functions of **`ModuleClass`** can be defined:

```
def get_value(self, value_name):
        request = Request(self.id, self.module_type, value_name)
        return ClientInfo.get_main().get_server_value(request)

    def set_value(self, value_name, value):
        command = Command(self.id, self.module_type, value_name, value)
        ClientInfo.get_main().set_server_value(command)
```

6.1.2 Unity Side

On the Unity side, counterparts of the implemented **`ModuleClass`** Python modules should be created to facilitate communication. Since Unity is a game engine, some things need to be set up before this step can be completed.

First, a new project is created in Unity. Unity has both 2D and 3D functionality, selected at startup. For this simulator, a 3D environment is needed. Upon creation, the Unity editor will make a new scene. Then, GameObjects can be added. GameObjects can represent any type of object. For example, the camera and light sources in a scene are instances of a GameObject. These objects have a set of components that control its behavior. What all GameObjects have in common is the Transform component: A Transform is responsible for the objects position, orientation and size. Furthermore, it's possible to add scripts to a GameObject. These scripts (written in C#) make it possible for developers to code their own GameObject behavior and directly interact with other objects or components.

To understand how to code with a game engine like Unity, it must first be clear how that code will run. Scripts attached to a **`GameObject`** must inherit from **`MonoBehaviour`**, which provides a set of methods that can be overridden. **`MonoBehaviour`** is the base class responsible for much of the program execution flow. The **`Awake`** and **`Update`** methods make it possible to code according to the Unity execution flow: **`Awake`** runs when the GameObject is activated for the first time (i.e. at the start of execution) and **`Update`** runs every frame. A frame is a fraction of time in which (among other things) the **`Update`** method of all activated scripts is ran.

```
public class CustomScript : MonoBehaviour {
      void Awake() {
            transform.position = Vector3.zero;
      }

      void Update() {
            transform.position += Vector3.up * Time.deltaTime;
      }
}
```

At the end of each frame, the current state of the scene is drawn to the screen. If the framerate is 60 Hz, the GameObjects and graphics will thus be updated 60 times per second. The framerate is dependent on the specifications of the device the application is running on. For example, slower computers will need more time to execute one frame, resulting in a lower framerate. The **`Time.deltaTime`** variable shown above makes sure the position translation is scaled to the time of a frame, so that the application runs correctly independent of the frame rate.

This does not mean conventional C# coding is not useful here: custom classes that do not inherit from **`MonoBehaviour`** can be created and called to within those functions. Sometimes it is necessary to execute some code with a different rate than the applications framerate. Coroutines exist for this purpose. Using a coroutine is similar to threading, as it can be running in parallel without stalling the Unity execution loop. A coroutine can be halted for any number of seconds and used to create an infinite loop:

```
private IEnumerator CustomCoroutine(){
        while (true) {
            PrintTime();
            yield return new WaitForSeconds(0.01f);
        }
}
```

An overview of the different timings using these two examples is presented in figure 7.

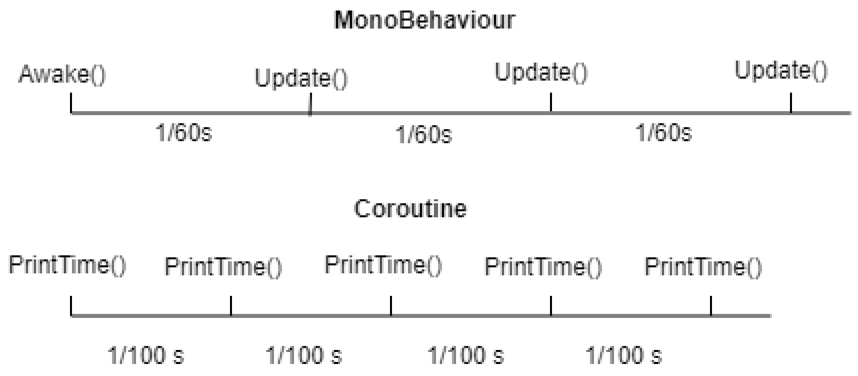


Figure 7: the timing differences between the regular Unity update loop and coroutines.

Since the **MonoBehaviour** framerate is variable, it's good practice to use coroutines for secondary processes (Unity, n.d.). The static rate of a coroutine is more suitable to communicate with the Python client. Furthermore, it's beneficiary when the communication slows down or stops: the **MonoBehaviour** update loop needs to finish executing all code in that frame before it can be rendered (thus slowing down its variable framerate), while coroutines are exempt from this.

A new GameObject is created with the **ServerInfo** script. This script contains the code that mirrors the networking part of the Python program: first, the handshake is completed. Then, a coroutine is started that handles a message receiver from the Python client according to a set rate. The simulated robot seems to function fine with a rate of 100Hz. If the coroutine wants to handle a message but can't receive any at that moment, the coroutine will immediately (after 1 millisecond) stop trying to receive and wait according to the set rate. When a message is received, it will need to interact with the requested module.

The C# equivalent of the Python **ModuleClass** will function somewhat differently in Unity. The individual modules should be updated according to the **MonoBehaviour** structure, as to properly make the simulation work. Modules are represented as **GameObject**s with a unique **ModuleScript** component. **ModuleScript** inherits from **MonoBehaviour**, to keep access to the **Awake** and **Update** functions. A **ModuleScript** will have the same identifying data as the Python class: ID and type. Added to this is a dictionary of values. This specific dictionary links a key (string) to a value (object). **GetValue** and **SetValue** functions are also added, similarly to the Python class:

```
public object GetValue(string valueName) {
      if (Values.ContainsKey(valueName)) {
            return Values[valueName];
      }
      return null;
}

public void SetValue(string valueName, object value) {
      if (Values.ContainsKey(valueName)) {
            Values[valueName] = value;
      } else {
            Values.Add(valueName, value);
      }
}
```

Every type of module has a script that inherits from **`ModuleScript`**, providing specific behavior. For example, the ultrasonic sensor **`ModuleScript`** gets the distance using a raycasting. Raycasting is a technique where a ray is shot according to a given point and direction to check if it hits an object. If it does, information regarding this path is stored in a **`RaycastHit`** object.

```
public class UltrasonicSensorScript : ModuleScript {
      void Update() {
            float distance = 0;
            RaycastHit hit = new RaycastHit();
            bool hitSomething = Physics.Raycast(transform.position,
transform.forward, out hit);

            if (hitSomething) {
                  distance = hit.distance;
            } else {
                  distance = float.MaxValue;
            }
            SetValue("Distance", distance);
      }
}
```

The values dictionary acts as a buffer between the rate of message handling and updating the Unity execution loop. If a value is set by the server in the middle of one frame, its effects will only be visible on the next frame.

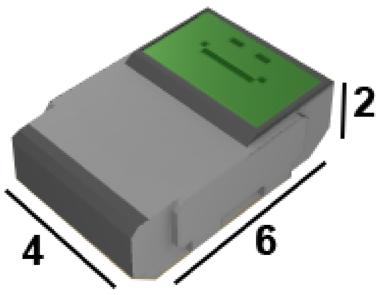
The collection of **`ModuleScript`**s which are active in a scene make up the so-called 'virtual brick'. One more problem remains: how are the Python modules linked to the C# modules? This is solved by matching the IDs and module types. First, the IDs must be set within the simulation software when creating a robot. Afterwards, when the Python client asks or sets a value of a module with a unique ID, the C# server will search for connected modules of the same ID. For cases where no ID is set, the server will try to find any connected module with the same type and an empty ID. When it finds that module, the ID will automatically be assigned for further use.

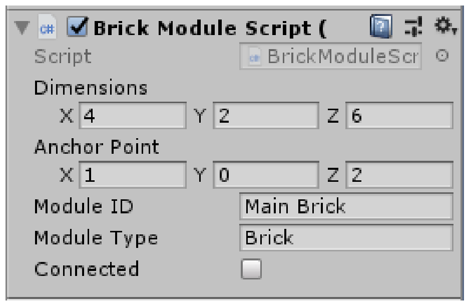
6.2 Modes of operation

6.2.1 Building mode

The building mode will first need to have access to all implemented **`ModuleScript`**s. However, every module has its own dimensions, 3D model and GameObject quirks. Distances in Unity are measured in no specific unit; making the developer responsible for choosing the scale. Most commonly, these units are set to be equal to meters. Unity allows developers to create **`Prefab`**s. A **`Prefab`** is a copy of a GameObject that can be instantiated (added to the scene) at any time at runtime. Every implemented **`ModuleScript`** is combined with a model and some measurement information to make a **`Prefab`**. These are then placed in a specific folder. A major benefit of the Prefabs is the convenient finetuning that is available from the Unity editor.

Figure 8 shows the standard EV3 brick with its dimensions, as well as its **`Prefab`**'s **`ModuleScript`** component.





Brick Module Script (
Script BrickModuleScr
Dimensions
X 4 Y 2 Z 6
Anchor Point
X 1 Y 0 Z 2
Module ID Main Brick
Module Type Brick
Connected ☐

Figure 8: the dimensions of the 'Brick' module with its script representation.

The building mode is started in a new scene. At startup, all **`Prefab`**s are loaded into the scene and shown to the user to choose from. These objects can then be placed on the (initially empty) robot. To accommodate this placing, a **`BuildModule`** is created from the **`ModuleScript`** attached to a chosen **`Prefab`**. This newly created class contains all the functions and variables needed to correctly place and rotate a multidimensional object. The modules are placed on a grid to reduce the complexity of this editor and the created robots.

A script is added to the scene to handle this process: **`PointsHandlerScript`**. Every point in the three-dimensional grid can either be free or occupied. If a point is an immediate neighbor of an occupied point, this point is legible for placement of a new module (as everything should be connected directly to each other). To compensate for the scale of Unity units, the size of every module **`Prefab`** is divided by ten. The brick in figure 8 would thus end up being about 40 x 20 x 60 cm. The **`PointsHandlerScript`** creates a grid of ten points per unit, or 10 cm between every point.

When the user has completed the robot, a blueprint of the entire robot (including all connected modules) is converted to a JSON string and stored. The robot can then be reconstructed with this string. The current version of the building mode scene when running is presented in figure 9.

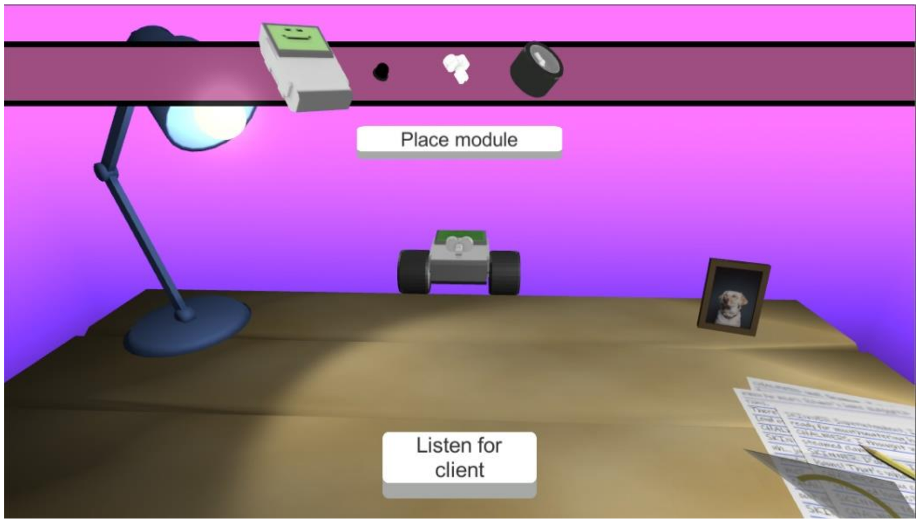


Figure 9: the building mode scene, with buttons to place the currently selected module and to start communication.

6.2.2 Action Mode

On startup, the currently stored robot will be loaded into the action mode scene. This scene also contains the standard environment to drive around in. The robot will then activate its **`RigidBody`** component. A **`RigidBody`** makes an object adhere to the rules of physics, as long as these objects have a collider. Colliders describe the bounds of an object, allowing them to interact with each other instead of intersecting. These components cause the robot to immediately drop to the ground, as expected.

The physics system of Unity has many more features to more closely model the real world. The center of mass can be directly set or accessed. Every **`RigidBody`** has its customizable settings for mass, bounciness, friction and other physical properties. Collision can be configured to find the balance between performance and accuracy.

Now the communication between the Python program can start. The **`ModuleScript`**s start executing their **`Awake`** and **`Update`** functions. A list of the currently connected modules is shown on screen, indicating whether these are being communicated with or not. If so, the latest values of its variables are displayed.

The bare-bones version of the present action mode scene is shown in figure 10.

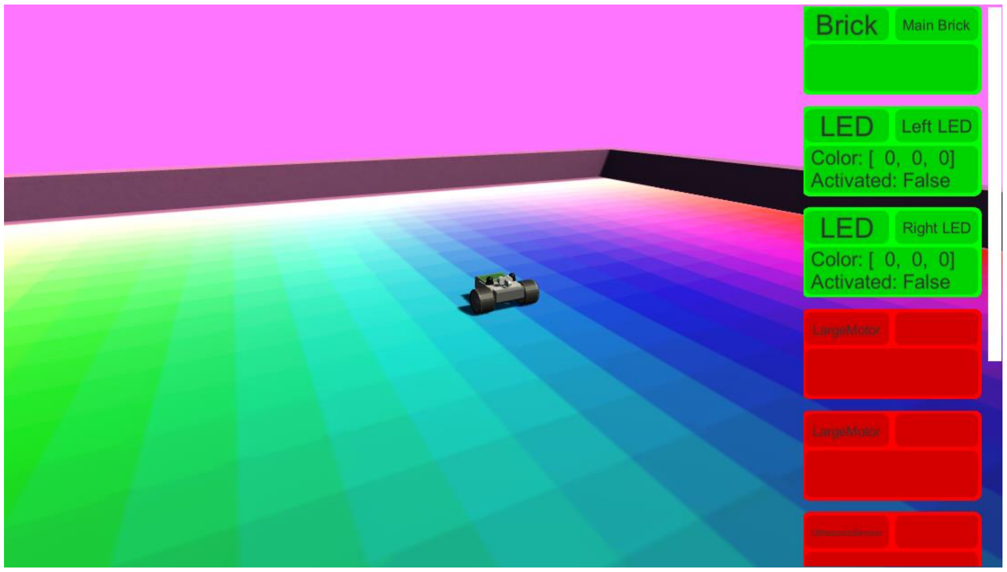


Figure 10: the action mode scene with the list of connected modules exhibited on the right.

7 CONCLUSION

7.1 EASE OF USE

Installation of this software should be intuitive. Luckily, the design choices lead to a convenient setup: a user only needs the built Unity application (a single executable provided for download) and a folder containing the re-implemented **ev3dev2**. Furthermore, the user should have Python 3 installed. Unity executables are completely standalone; the Unity Editor will not have to be installed. The Unity application comes with a settings menu on launch. This menu allows the user to finetune the graphical fidelity of the 'game'. Since the application is not visually demanding, most computers should run it fine at the least graphically-demanding settings without losing performance. Issues might occur when running the simulator on a computer with no dedicated graphics card or outdated/no drivers. However, choosing a smaller output resolution might resolve this problem.

Gaining access to the source code (for further development) should be as easy. The Python side of the project is customizable out of the box, as it is an interpreted programming language. Developing the Unity application requires installing the latest version of the Unity Editor, freely available online. Then, the project files can be loaded into the editor. These files will be shared on GitHub. After a developer is finished, it can distribute the application for any supported platform with a few clicks. Currently only one environment is supported for the action mode.

7.2 REQUIREMENTS

The educational aspects of the simulation software have taken a back seat in comparison to the produced framework. These ideas need to be implemented in the next step of the development cycle, where the user interface and functionality is added to the simulator. The four requirements categories stated before have been implemented as follows:

7.2.1 Platform

Because of the multiplatform nature of both Unity and Python, this simulation framework should be able to run on most platforms. It has been fully developed and tested in Windows, but might need some operating system specific fixes or changes to work on other platforms. For example, different operating systems or computer setups might have different rules concerning the use of networking sockets and the GPU, resulting in unexpected behavior and timing.

7.2.2 Coding

Choosing Python as the programming language on the user-end, keeps its educational benefits intact. The Python programs needed for the real-life EV3 program and simulator are identical with this framework.

New programs can be written in the root folder containing the **ev3dev2** folder. These programs can call to the re-implemented ev3dev2 package, causing them to stall on execution waiting to connect.

7.2.3 Simulating

Unity as the development environment has reduced the difficulty of implementing certain features such as 360-degree cameras and storing the module prefabs, in addition to its built-in rendering and graphical features. The support for the engine and its efficiency makes it a long-term solution too.

Currently, no significant effort has been put in the visualization of the simulation. However, due to the Unity UI system and the accessibility of the framework this should not be too difficult. More visualization methods can be used to show for example the frequency of request/commands or the change of the module values over time. Furthermore, a more robust camera system should be implemented for the action mode.

7.2.4 Customizability

This framework has customizability at its roots. In particular, the implementation of the modules on both the Python and Unity side (including the **`ModuleScript`** component) is built to be easily extendable. With this solution, it would also be possible to implement features that are not present in real-life Mindstorms sets.

The workflow of creating a new module starts with the implementation of a new class inheriting from **`ModuleClass`**, stored in a Python file in the ev3dev2 folder. This class needs to call the **`ModuleClass`** initialization method correctly. Methods can be added to the class to get and set values with the use of the inherited **`get_value`** and **`set_value`** methods. This class can then be imported and initialized in the EV3 program.

In the Unity project, a prefab must be made for the new module. This prefab is a **`GameObject`** with a newly created script component, inheriting from **`ModuleScript`**. This new script can call to its 'hardware' by using the **`GetValue`** and **`SetValue`** methods, with the same value names as the Python definitions. This prefab should then be dragged in to the 'Prefabs' assets folder.

Now, if everything is set up correctly, the new module is available for use in the building mode. After switching to action mode, the module will appear in the 'connected modules' list and interacts with the Python program.

8 DISCUSSION

8.1 SIMULATION ACCURACY

The simulation is not very accurate in its current iteration; various ‘essential’ sensors/actuators (such as the wheels) are not yet implemented. However, due to the tools that Unity provide and the created framework, most of these would be fairly easy to add. Unity has a preset solution for wheels in its physics system, with support for various real-life concepts (dampening, motors, braking). However, the miniature nature of the Mindstorms wheels makes it a bit different to implement. For example, the wheels are practically locked in place when no torque is applied, instead of freely rolling around. That and the instant acceleration of the electric motor result in a unorthodox motion path. More research should be done on how this works exactly.

Accuracy can be provided through finetuning, provided a comparison of execution on the simulator and a real-life robot. It is necessary to implement certain features to not be perfect; as the real world is not perfect either. This comes in the form of adding random noise to sensors and slightly randomizing the module behaviors. Simulation of the (real-life) delay of a call to hardware can also be further improved. It’s possible that this delay is different for every unique module. In this case, the **`ModuleClass`** representation would need a slight modification to acknowledge this.

8.2 FUTURE WORK

The conceptual framework for these two programs can be ported to other programming languages and/or simulators. The basic functionality in Python and C# used in this implementation that is available for (almost every) other language. For example, Python can be replaced by Java to communicate with the simulator. The **`ModuleClass`**, messages, and networking (including JSON) would need to be correctly implemented in Java for this to work. No major changes should be necessary to get this working.

Several features are still missing. Most importantly, not all of ev3dev2 is re-implemented. Besides this, the simulation software currently has no support for environment editing. Only the standard surroundings are available for the robot to work with. The current system for building a robot could be extended for this purpose.

Disconnecting and connecting is somewhat slow currently, networking errors are not sufficiently handled in both applications.

Beyond these basic features, here are some ideas for new functionality:

- Allow for multiple robots in one environment. Due to the server-client nature of the communication framework, it would be possible to make multiple concurrent connections. For example: two different programs each running its own robot in the simulator.
- Non-local connections. The communication now rests on the concept of hosting a local server. This could be changed to facilitate outside connections: e.g. running the simulator on a phone, with the Python program running from a PC. Timing would certainly be different in this scenario, possibly breaking some of the communication.

- An easier way to import new modules into the Unity program. In the current structure, developers would need to download both the Unity editor and the source project file to add functionality in the simulator. Perhaps there is some method of importing these custom modules at runtime.
- A way to easily program a Python ev3dev2 program without having to be connected to a simulator. This can be achieved by storing the simulated hardware values locally, so the user can test the program against default values.

9 Bibliography

Allen, M. P. (2007). Educational aspects of molecular simulation. *Molecular Physics*, 157-166.

Anderson, N. (2005). ‘Mindstorms’ and ‘Mindtools’ Aren’t Happening: streaming of students via socio-economic disadvantage. *E-Learning*, 144-152.

Beer, N. A., Grit, M. B., Good, M. L., & Gravenstein, D. (2001). Educational simulation of the electroencephalogram (EEG). *Technology and Health Care*, 237-256.

Besouw, G. (2018, April 16). Simulating NXT-robots. Retrieved from https://www.cs.ru.nl/bachelors-theses/2018/Giel_Besouw___4483898___Simulating_NXT_robots.pdf

Cogmation Robotics. (2018). Retrieved from Visual Robotics Toolkit: https://www.virtualroboticstoolkit.com/

Engblom, J. (2008). Using Simulation Tools for Embedded Development. *Embedded Systems Conference, Silicon Valley*, 1.

ev3dev. (2018). Retrieved from ev3dev: https://www.ev3dev.org/

Fraunhofer IAIS. (2018). Retrieved from Open Roberta Lab: https://lab.open-roberta.org/

Gandy, E. A., Bradley, S., Arnold-Brookes, D., & Allen, N. R. (2015, December 15). The use of LEGO Mindstorms NXT Robots in the Teaching of Introductory Java Programming to Undergraduate Students. *Innovation in Teaching and Learning in Information and Computer Sciences*, pp. 2-9.

Gilday, D. (2016). Retrieved from MindCub3r: http://mindcuber.com/

Klassner, F., & Anderson, S. D. (2003, July 22). LEGO MindStorms: Not Just for K-12 Anymore. *IEEE Robotics & Automation Magazine*, pp. 12-18.

Lego. (n.d.). Retrieved from Lego Digital Designer: https://www.lego.com/en-us/ldd

Lego. (2018). Retrieved from Mindstorms: https://www.lego.com/en-us/mindstorms/learn-to-program

Lego. (2018). *Learn to Program*. Retrieved from Mindstorms: https://www.lego.com/en-us/mindstorms

Mason, R. (2013, August 30). Mindstorms robots and the application of cognitive load theory in introductory programming. *Computer Science Education*, pp. 296-314.

McLeod, S. A. (2017). *Simply Psychology*. Retrieved from Kolb - learning styles: https://www.simplypsychology.org/learning-kolb.html

Newtonsoft. (n.d.). *Json.NET*. Retrieved from https://www.newtonsoft.com/json

Pasquale, S. J. (2015, March). Educational science meets simulation. *Best Practice & Research Clinical Anaesthesiology*, pp. 5-12.

Purnus, A., & Bodea, C.-N. (2015). Educational Simulation in Construction Project Financial Risks Management. *Procedia Engineering*, 449-461.

python-ev3dev2. (2018). Retrieved from python-ev3dev2 2.0.0b2: https://pypi.org/project/python-ev3dev2/

RoboMatter. (n.d.). Retrieved from Robot Virtual Worlds: http://www.robotvirtualworlds.com

Robomatter. (2005-2014). Retrieved from RobotC: http://www.robotc.net/

Simmons, S. (2018). Retrieved from QEV3Bot Simulator: https://sites.google.com/site/qev3bot/qev3bot-simulator

statcounter. (2018). Retrieved from Desktop Operating System Market Share Worldwide: http://gs.statcounter.com/os-market-share/desktop/worldwide

Superlegosam. (2014, 8 26). *LEGO car factory - ONLY ONE NXT*. Retrieved from YouTube: https://www.youtube.com/watch?v=4Bp0GJSqfso

TRIK. (2014). Retrieved from TRIK Studio: http://blog.trikset.com/p/trik-studio.html

Unity. (n.d.). *Coroutines*. Retrieved from https://docs.unity3d.com/Manual/Coroutines.html

Unity Technologies. (2018). Retrieved from Unity: https://unity3d.com/

Wikipedia contributors. (n.d.). *Emulator*. Retrieved from Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Emulator&oldid=866614331

Zając, T. (2013, 1 31). *LEGO Mindstorms NXT Loom Machine*. Retrieved from YouTube: https://youtu.be/IPIJsdvDjsc

Zygouris, N. C., Striftou, A., Dadaliaris, A. N., Stamoulis, G. I., Xenakis, A. C., & Vavougios, D. (2017). The use of LEGO mindstorms in elementary schools. *IEEE Global Engineering Education Conference (EDUCON)*, (pp. 514-516). Athens.