



KANDIDAT

6

PRØVE


IE502414 1 Masteroppgave i Simulering og Visualisering

Emnekode	IE502414
Vurderingsform	Oppgave
Starttid	01.06.2018 12:00
Sluttid	29.06.2018 12:00
Sensurfrist	29.09.2018 02:00
PDF opprettet	05.11.2018 09:38
Opprettet av	Anne Lise Grande




1 Ny oppgave

Please upload your master thesis here (PDF file).

We refer to the e-mail dated 30.05.18 from Anne Lise Grande regarding: 1. The Template for the Front Page, 2. Mandatory Statement and 3. Publication Agreement. These three files must be included in your PDF file together with the thesis.



Din fil ble lastet opp og lagret i besvarelsen din.

 Last ned Fjern Erstatt

Filnavn:	Torstein_Sundnes_Lenerand_Master_Thesis.pdf
Filtype:	application/pdf
Filstørrelse:	3.57 MB
Opplastingstidspunkt:	11.06.2018 17:44
Status:	Lagret

Besvart.

Master's degree thesis

Component-Based Simulator for Modelling the Design and Dynamics of Modular Robots

Field of study: (880MVS) Simulation and Visualization

University: NTNU Ålesund

Number of pages including this page: 84

Rev.	Place, date:	Author:
1.0	Aalesund, 10.06.2018	Torstein Sundnes Lénérand

Mandatory statement

Each student is responsible for complying with rules and regulations that relate to examinations and to academic work in general. The purpose of the mandatory statement is to make students aware of their responsibility and the consequences of cheating. **Failure to complete the statement does not excuse students from their responsibility.**

Please complete the mandatory statement by placing a mark in each box for statements 1-6 below.		
1.	I/we hereby declare that my/our paper/assignment is my/our own work, and that I/we have not used other sources or received other help than is mentioned in the paper/assignment.	<input checked="" type="checkbox"/>
2.	<p>I/we hereby declare that this paper</p> <p>1. Has not been used in any other exam at another department/university/university college</p> <p>2. Is not referring to the work of others without acknowledgement</p> <p>3. Is not referring to my/our previous work without acknowledgement</p> <p>4. Has acknowledged all sources of literature in the text and in the list of references</p> <p>5. Is not a copy, duplicate or transcript of other work</p>	<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
3.	I am/we are aware that any breach of the above will be considered as cheating, and may result in annulment of the examination and exclusion from all universities and university colleges in Norway for up to one year, according to the Act relating to Norwegian Universities and University Colleges, section 4-7 and 4-8 and Examination regulations at NTNU.	<input checked="" type="checkbox"/>
4.	I am/we are aware that all papers/assignments may be checked for plagiarism by a software assisted plagiarism check.	<input checked="" type="checkbox"/>
5.	I am/we are aware that The Norwegian University of Science and Technology (NTNU) will handle all cases of suspected cheating according to prevailing guidelines.	<input checked="" type="checkbox"/>
6.	I/we are aware of the University's rules and regulations for using sources.	<input checked="" type="checkbox"/>

Publication agreement

ECTS credits: 30

Supervisors: Arne Styve, Dr. Guoyuan Li.

Agreement on electronic publication of master thesis

Author(s) have copyright to the thesis, including the exclusive right to publish the document (The Copyright Act §2).

All theses fulfilling the requirements will be registered and published in Brage with the approval of the author(s).

I/we hereby give NTNU the right to, free of charge, make the thesis available for electronic publication: yes

Is there an agreement of confidentiality? no

(A supplementary confidentiality agreement must be filled in)

Date: 10.06.18

Acknowledgments

This project marks the end of 2 years on the Simulation and Visualization masters study at NTNU Ålesund. I would like to thank my supervisors, Dr. Guoyuan Li and Assistant Professor Arne Styve, for their guidance throughout the project and for discussions which always pushed me to see things from different perspectives. With extensive knowledge about their respective fields, they have given better guidance and feedback than I could ever hope for. The people at Algoryx support have also been very helpful in aiding with troubleshooting and guidance. Lastly, I would like to thank my study program coordinator Siebe Bruno Van Albada for continually finding ways to improve the sim&vis course, and for helping me (together with my supervisors) get the permission to write this specific thesis.

i. Abstract

This project presents the design of a component-based simulator used for modelling the design and movement of chain-based modular robots. This work is in collaboration with NTNU Ålesund and implemented in the Unity® game engine with Algorix® Dynamics for physics calculations. The focus is on Modular robots, along with techniques for simulator creation and software development such as Component-Based Software Engineering and Design. The Unified Process is used for prototyping and research, while the finished design is verified using tests, reviews, and use-case studies. This thesis discusses the impact of using Component-Based Design in a relatively small project, and the advantages/disadvantages of this decision. The goal is to provide the optimum tool for students to learn about, and researchers to develop, customized modular robots.

ii. Contents

I.	ABSTRACT	1
II.	CONTENTS	1
III.	LIST OF FIGURES	3
IV.	LIST OF TABLES	4
V.	DOCUMENT HISTORY	5
VI.	ABBREVIATIONS AND EXPLANATIONS	6
1.	INTRODUCTION	7
1.1.	INTRODUCTION AND MOTIVATION.....	7
1.1.1.	<i>Problem</i>	7
1.1.2.	<i>Motivation</i>	7
1.1.3.	<i>Scope</i>	8
1.1.4.	<i>Objective</i>	9
1.1.5.	<i>Research Questions</i>	9
1.2.	PREVIOUS WORK.....	10
1.2.1.	<i>Literature</i>	10
1.2.1.1.	Webots [4].....	10
1.2.1.2.	Unity [5].....	10
1.2.1.3.	VSPARC [6].....	10
1.2.1.4.	MECABOT [8, 9].....	11
1.2.1.5.	Screw-less Solution for Snake-like Robot Assembly & Sensor Integration [10].....	11
1.2.1.6.	Modular Robot Systems (Self-assembly) [1].....	11
1.2.1.7.	A Light-Weight Robot Simulator for Modular Robotics [11].....	12
1.2.1.8.	Component-based Development Process & Component Lifecycle [13].....	12
1.2.1.9.	Twenty-eight years of component-based software engineering [14].....	12
1.2.2.	<i>Literature discussions</i>	13
1.2.2.1.	Advantages of CBSE.....	13
1.2.2.2.	Development focus for various modular robots.....	13
2.	BACKGROUND	15
2.1.	PROJECT FRAMEWORK.....	15
2.1.1.	<i>Unified Process</i>	15
2.1.2.	<i>Phases</i>	17
2.1.3.	<i>Risk</i>	18
2.2.	MODULAR ROBOTICS.....	19
2.2.1.	<i>Design</i>	19
2.2.2.	<i>Dynamics</i>	20
2.2.3.	<i>Optimization algorithms viable for modular robotics</i>	21
2.2.3.1.	Genetic algorithm.....	21
2.2.3.2.	Simulated Annealing.....	24
2.3.	COMPONENT-BASED DEVELOPMENT.....	25
2.3.1.	<i>Introduction to CBSE</i>	25
2.3.2.	<i>Business use-case</i>	26
2.3.3.	<i>Component-based optimization</i>	27
3.	METHODOLOGY	28
3.1.	DEVELOPMENT.....	28
3.1.1.	<i>Project methods</i>	28
3.1.1.1.	Unified Process.....	29
3.1.2.	<i>Research and planning</i>	30
3.1.3.	<i>Modular robotics</i>	30
3.2.	VERIFICATION AND VALIDATION.....	32
3.2.1.	<i>Simulator Specifications</i>	32
3.2.1.1.	Requirements.....	32
3.2.1.2.	Test and Verification.....	33

3.2.1.3. System validation.....	34
3.2.2. Framework and simulator verification	34
3.3. TOOLS, PLATFORMS, LIBRARIES.....	35
3.3.1. Hardware.....	35
3.3.2. Software components.....	35
3.3.3. Algoryx Dynamics	37
3.3.4. Unity	38
4. RESULTS AND FINDINGS.....	39
4.1. DESIGN OF THE MODULAR ROBOT SIMULATOR.....	39
4.1.1. Setting up the software environment.....	39
4.1.2. Component-based design	40
4.1.2.1. Simulation Core	41
4.1.2.2. Algoryx Interface.....	42
4.1.2.3. Visualization.....	42
4.1.2.4. Dynamics.....	43
4.1.2.5. Optimization	43
4.1.2.6. Main	44
4.1.3. Software design	45
4.1.3.1. Simulation Core	46
4.1.3.2. AgX Interface	53
4.1.3.3. Unity_Visualization	59
4.1.3.4. Dynamics.....	61
4.1.3.5. Optimization	62
4.1.3.6. Scene Designer.....	63
4.1.4. Core Framework usage (Robot design)	65
4.1.4.1. Robot assembly.....	65
4.1.4.2. Object creation	65
4.1.4.3. XML functionality.....	68
4.2. VERIFICATION AND VALIDATION.....	69
4.3. CASE STUDIES	70
4.3.1. Creating and running a scenario	70
4.3.1.1. Design	70
4.3.1.2. Result	71
4.3.2. Dynamics test	73
4.3.3. Framework case-study	74
5. DISCUSSION	77
5.1. CBD DECISIONS	77
5.2. STAKEHOLDER NEEDS (NTNU).....	78
5.3. ISSUES.....	78
5.4. SUMMARY	79
5.5. FURTHER WORK	80
6. CONCLUSION	81
7. REFERENCES.....	82

iii. List of figures

FIGURE 1: MAIN PROJECT COMPONENTS AND PROJECT SCOPE.....	8
FIGURE 2: ITERATIVE DEVELOPMENT MODEL.....	15
FIGURE 3: PROPOSED PROJECT PLAN.....	17
FIGURE 4: MODULAR ROBOT; PITCH, YAW, AND PITCH-YAW CONFIGURATIONS.....	19
FIGURE 5: EXAMPLE FITNESS FUNCTION.....	22
FIGURE 6: THE MUTATION AND SELECTION PROCESS IN A GA.....	22
FIGURE 7: "CUT AND SPLICE" CROSSOVER OPERATOR.....	23
FIGURE 8: EXAMPLES OF MUTATION OPERATORS SOURCE: HTTPS://WWW.RESEARCHGATE.NET/272093243	23
FIGURE 9: SIMULATED ANNEALING PATHFINDING EXAMPLE [FROM ANOTHER PROJECT].....	24
FIGURE 10: PROJECT DEVELOPMENT PLAN.....	29
FIGURE 11: ALGORYX SIMULATION OF A MODULAR SNAKE-LIKE ROBOT.....	37
FIGURE 12: UNITY EDITOR.....	38
FIGURE 13: SETTING UP THE ENVIRONMENT VARIABLES ON THE TEST COMPUTER.....	39
FIGURE 14: MODULAR ROBOT SIMULATOR COMPONENTS.....	41
FIGURE 15: CORE FRAMEWORK SOFTWARE ARCHITECTURE.....	45
FIGURE 16: SEQUENCE OF ALGORYX OBJECT HANDLING.....	53
FIGURE 17: UNITY_VISUALIZATION OBJECT LIFECYCLE.....	60
FIGURE 18: SCENE DESIGNER INTERFACE.....	63
FIGURE 19: THE SCENE DESIGNER'S MOST IMPORTANT FEATURES.....	64
FIGURE 20: ROBOT ASSEMBLY STRUCTURE.....	65
FIGURE 21: CREATION AND DESTRUCTION OF SIMULATION OBJECTS.....	66
FIGURE 22: XML FILE SCENARIO CLASS REPRESENTATION.....	68
FIGURE 23: CASE STUDY - DESIGN OVERVIEW.....	70
FIGURE 24: CASE STUDY - SENSOR MODULE AND SCENE OBJECT DESIGN.....	70
FIGURE 25: ROBOT MOVING FORWARD TO PUSH THE BALL.....	71
FIGURE 26: DISTANCE SENSOR MEASUREMENTS.....	71
FIGURE 27: FORCE SENSOR VS Y-POSITION MEASUREMENTS.....	72
FIGURE 28: ROBOT FORWARD MOTION.....	73
FIGURE 29: ROBOT FORWARD MOTION WITH INCREASED AMPLITUDE.....	73
FIGURE 30: ROBOT WIDE TURN.....	73
FIGURE 31: ROBOT SHARP TURN.....	73
FIGURE 32: CUSTOM PROJECT, SCENARIO.....	75
FIGURE 33: CUSTOM PROJECT, MODULE MOVEMENT.....	76

iv. List of tables

TABLE 1: DOCUMENT HISTORY.....	5
TABLE 2: RISK MATRIX	18
TABLE 3: EXAMPLE OF CHROMOSOMES IN A GA SYSTEM.....	21
TABLE 4: REQUIREMENT IMPORTANCE	32
TABLE 5: REQUIREMENT EXAMPLE.....	33
TABLE 6: VERIFICATION CRITERIA.....	33
TABLE 7: TEST SPECIFICATION EXAMPLE, FULLY VERIFIED	33
TABLE 8: TEST COMPUTERS.....	35
TABLE 9: CONTENTS OF THE SCENARIO CLASS	46
TABLE 10: CONTENTS OF THE ROBOT CLASS	47
TABLE 11: ROBOT CLASS FUNCTIONS	48
TABLE 12: CONTENTS OF THE MODULE CLASS	48
TABLE 13: MODULE CLASS FUNCTIONS.....	48
TABLE 14: CONTENTS OF THE FRAME CLASS	49
TABLE 15: FRAME CLASS FUNCTIONS	49
TABLE 16: CONTENTS OF THE JOINT CLASS	50
TABLE 17: JOINT CLASS FUNCTIONS	50
TABLE 18: CONTENTS OF THE SCENEOBJECT CLASS.....	51
TABLE 19: SCENEOBJECT CLASS FUNCTIONS	51
TABLE 20: CONTENTS OF THE SCENE CLASS	52
TABLE 21: SCENE CLASS FUNCTIONS	52
TABLE 22: CONTENTS OF THE AGX_ASSEMBLY CLASS:	54
TABLE 23: AGX_ASSEMBLY CLASS FUNCTIONS.....	54
TABLE 24: CONTENTS OF THE AGX_FRAME CLASS.....	55
TABLE 25: AGX_FRAME CLASS FUNCTIONS	55
TABLE 26: CONTENTS OF THE AGX_JOINT CLASS.....	56
TABLE 27: AGX_JOINT CLASS FUNCTIONS	56
TABLE 28: CONTENTS OF THE AGX_PRIMITIVE CLASS.....	56
TABLE 29: AGX_PRIMITIVE CLASS FUNCTIONS	57
TABLE 30: CONTENT OF AGX_SCENE CLASS	57
TABLE 31: AGX_SCENE CLASS FUNCTIONS	57
TABLE 32: CONTENT OF AGX_SIMULATION CLASS.....	58
TABLE 33: AGX_SIMULATION CLASS FUNCTIONS.....	58
TABLE 34: MAIN CONTENT OF THE UNITY_VISUALIZATION CLASSES	59
TABLE 35: MAIN CONTENT OF THE DYNAMICS CLASS	61
TABLE 36: MAIN CONTENT OF THE OPTIMIZATION CLASS.....	62
TABLE 37: FRAMEWORK IMPLEMENTATION IN A NEW PROJECT	75

v. Document history

Rev.	Date	Author	Description
0.1	16.04.2018	TS	Document created Project sub-documents assembled
0.2	04.05.2018	TS	Methodology reviewed Results part 1 finished
0.3	03.06.2018	TS	Results part 2 finished Discussion and conclusion finished
1.0	10.06.2018	TS	Published

Table 1: Document history

vi. Abbreviations and explanations

.Net	Software framework developed by Microsoft
.png	Short for Portable Network Graphics, a graphics file format.
AgX	Algorix
CBD	Component-Based Development
CBSE	Component-Based Software Engineering
dll	Dynamic-link library (.dll file extension)
Enum	Enumerated type, numbers (constants) represented by names (values)
EQ	Equation
exe	Executable file (.exe file extension)
GA	Genetic Algorithm
Gait	Pattern of movement of limbs of animals
JSON	JavaScript Object Notation
MRSim	Modular Robot Simulator
NTNU	Norwegian University of Science and Technology
Porting	Translating code from one language to another
PSO	Particle Swarm Optimization
SA	Simulated Annealing
UI	User Interface
UP	Unified Process
XML	eXtensible Markup Language
XPath	XML Path Language

1. Introduction

1.1. Introduction and motivation

The field of modular robotics has been in development since the late eighties, starting with the creation of CEBOT in 1988, up until all the different research projects that are being worked on today [1]. Hereafter, the technology has continually been improved over the years, with the implementation of more flexibility and functionality in modular robot systems. Modular robots present an interesting and useful field of study, especially when it comes to executing tasks humans are unable to perform. By creating robots with different designs, the application areas can be anything from subsea operations, to deep space exploration.

Since the concept was presented, humans have always been interested in creating robots that can change shape as seen in the movie industry, with examples like the robots in “Terminator” [2], and “Transformers” [3]. The portrayal of modular robotic technology in movies may very well be the inspiration that research groups need to come up with matching solutions in real-life applications. Solutions that were once considered science fiction, which are now actual proven science.

1.1.1. Problem

Humans can solve plenty of problems and are flexible enough to do relatively precise operations. However, there are certain actions humans are not able to do and must thus rely on other means of completing tasks. For this purpose, researchers are creating a wide range of robots (with the focus of this thesis being on modular robots). However, because of the wide range of use-cases and implementation techniques, together with a limited amount of people knowing how to design modular robots or how to use modelling programs to design them, many potentially great ideas may never be explored. If there existed a program that could allow users to go from an early concept to finished simulation in a fraction of the time with relative ease, it should be easier for seasoned researchers in addition to students and non-programmers, to test out their ideas and possibly extend the number of possible uses for modular robotics.

1.1.2. Motivation

NTNU currently has an ongoing research project involving different forms of modular robots. Currently they are being manually designed from scratch or a template by the researchers, before being created and tested in self-coded simulations. Better and faster results should be achieved if there existed a fast prototyping platform for robot assembly, scene construction and control algorithm testing. This way users could create and optimize robots for

different scenarios, using only a fraction of the time and resources as with the approach implemented today, no matter how much experience the user has in creating modular robots from before. Another motivation for this project is to create a learning platform where students may study modular robotics and create their own designs and algorithms, without the need for specialized knowledge of programming or physics simulations.

1.1.3. Scope

There are five main fields that will be used in this project to create a simulator for modular robots:

- Modular robot research
- Implementation of visualization capable of realizing the project goals
- A realistic physics engine for prototyping (in this case, Algorix)
- Component-based development for a flexible and expandable system
- The iterative based project framework “Unified Process” for reliable prototyping

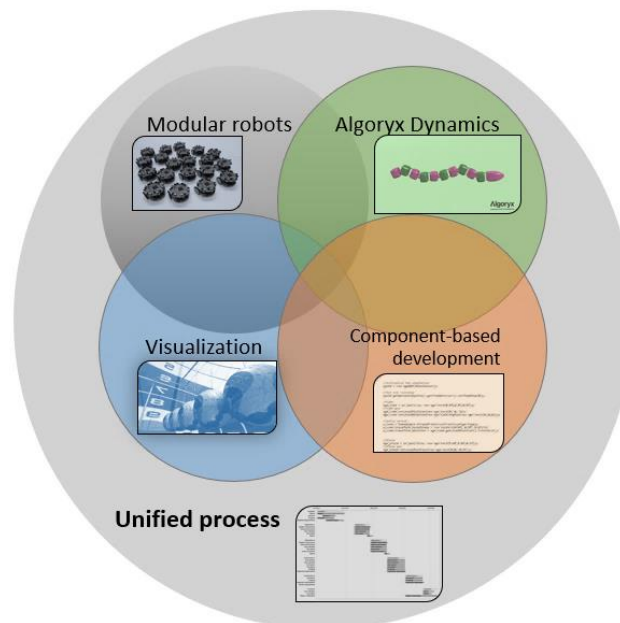


Figure 1: Main project components and project scope

The scope of the project is limited to the snake/caterpillar configurations of modular robots. As the project's estimated time-frame is around 4 months, there are limits to the amount of time spent on different tasks, making the simulation's core functionality the focus as it should be modular enough for integration with other potential visual interfaces. Based on this, the component-based model is a large part of the project, as it facilitates separation of concerns and incremental improvements to the simulator functionality. The visualizations shall be

realistically rendered by interface with the stored physics objects and provide intuitive simulation controls. When the modular robot and its environment is created, the user should be able to apply optimization algorithms increasing the efficiency of the created robot prototypes.

1.1.4. Objective

The objective of this project is to enable realistic and easy design of snake-based modular robots and exploring the benefits of component-based development in the simulation architecture of a relatively small-scale project. The tool should give the user, no matter the level of their previous knowledge, the ability to design modular robots and implement their own ideas and creative solutions to realize their goals. It is hoped that the simulator will motivate students to learn about modular robotics and allow researchers to develop their robots, or future simulation systems, more quickly and efficiently. Having a simulator with realistic physics, a professional degree of functionality and high quality visual effects should also make the design of modular robots more appealing. The core simulation framework should be usable in custom projects, requiring the component-based development method to be implemented correctly.

1.1.5. Research Questions

- Is the proposed simulation platform better for developing modular robots than manually programming simulations from scratch?
- Will the component-based software development method notably improve the flexibility of a simulation platform and ease its further development?
- Is the AgX physics library a stable choice for a simulator platform, and how efficient is it in providing realistic simulations while allowing for the effective use of optimization algorithms?
- Will the benefits of Component-Based Development outweigh the disadvantages of the implementation in an independent, small-scale project?

1.2. Previous work

Following section contains literature on key topics in the project. Specifics related to the project are in focus, with the two main themes being modular robot design and Component-Based Software Engineering (CBSE). Sections (1.2.1.1-1.2.1.3) feature a game engine and simulator systems, where features related to the project are highlighted.

1.2.1. Literature

1.2.1.1. Webots [4]

Webots is a simulator designed for the creation of general robots. The application allows users to create robots with functionality ranging from simple motorized tasks, to complex intelligent behavior. It can simulate flying, rolling, and legged robots to mention a few. Designing robots can be done by using pre-defined components, such as sensors, wheels, and geometries. Webots runs on the ODE physics engine, which provides fast real-time physics simulations. As Webots is a simulator for general modelling of robots, the specifications come from the users themselves, requiring some knowledge of programming and robot design. Implementation of optimization algorithms is done manually, mostly via supervisor controllers which are required for reading positions, distances, and general simulation properties (Webots PRO functionality).

1.2.1.2. Unity [5]

The game engine Unity uses components assigned to objects to handle the game-engine architecture, which is an intuitive extension of the object-oriented way of designing systems. The game object is in itself a component and may have components added to it for increased functionality, such as a rigid body for physics calculations or a mesh for visualization. The components have no knowledge of its surroundings or other components, as they are only required to do their intended tasks parallel to whatever else is happening in the environment. This method for defining objects automatically ensures that the created objects are independent and can be used in different scenarios with no need for modification of the code.

1.2.1.3. VSPARC [6]

VSPARC is a more specific simulator focused on modular robots. The simulator enables the creation of one specific type of module, called SMORES [7], which may be rearranged to fit pre-determined configurations. There is only one type of module, but the joints may be configured for speed or position control, adjusting the motion and design of the robot. The user interface allows for “drag and drop” functionality, where the user clicks on the side of an already

created module to add a new one. This results in a very user-friendly and visual-based robot construction tool. VSPARC is created in Unity, whose physics engine is designed for quick and simple simulations and is thus well-suited for quick prototyping and proof of concepts.

1.2.1.4. MECABOT [8, 9]

The MECABOT is a modular robotics system developed by the Militar Nueva Granada University (Colombia). It is a modular robotic system focused on chain-based architecture, with snake and caterpillar locomotion. The robot is created with several different configurations both physically at the research locations, and virtually in the Webots simulation system. Possible robot configurations are many, including prototypes for space exploration, self-balancing tables, and hexapods. The specifications for the MECABOT research topics also contain details about robot dynamics with relation to velocity, module size and sine functions, highlighting differences in performance based on varying variables.

1.2.1.5. Screw-less Solution for Snake-like Robot Assembly & Sensor Integration [10]

The focus of NTNU's research project regarding modular robots is to develop snake-inspired robots that can be reproduced via fast prototyping with easy to configure sensors. The modules are 3D-printed and connected to each other by sliding connectors, thus screw-less. Sensors are placed in an intermediate module featuring the same sliding connectors. Several solutions for connections are proposed, such as wired and wireless communication, the latter removing the need for cables between modules. The paper also describes the researchers' selected control behavior of the modular robot as being bio-inspired and allowing for sidewinding behavior, as seen in snakes.

1.2.1.6. Modular Robot Systems (Self-assembly) [1]

A proposed approach from K.Gilpin and D.Rus uses induction coils in the modules to perform the assembly connections to neighboring modules. The robots are as small as $12mm^2$, and contain no moving parts, making them less expensive and more durable. The algorithm for self-assembly assumes one root node, to which the neighboring modules will try to attach themselves. The newly attached modules will then calculate their X and Y location relative to the root node, and thus completing some part of the overall structure. If this location is not appropriate relating to the neighbor modules, which should have two neighbors itself, the module will power off, and detach itself from the structure.

1.2.1.7. A Light-Weight Robot Simulator for Modular Robotics [11]

The CTU (Czech Technical University) in Prague, faculty of electrical engineering, has created a simulator named “Sim” within an EU project called Symbion [12]. The project focuses on behavior and design of modular robots based on biological and evolutionary computational approaches. CTU’s simulator is based on a component-based design enabling the physics simulation and the visualization to be separate, allowing non-GUI machines such as computation grids to perform the physics calculations when visualization is not required. The simulator is used for a variety of robot types, including modular robots and mobile (wheel-based) robots. Lastly, locomotion examples are described together with optimization examples (PSO), which can be implemented in the simulator.

1.2.1.8. Component-based Development Process & Component Lifecycle [13]

Component-Based Software Engineering is a relatively new discipline, with no specific processes or workflow standards. However, the main principles featured in CBSE are proven to be advantageous in multiple use-cases and projects. Mainly, the concept of developing systems from pre-made components is one of the main ideas in CBSE. This reduces the amount of work from the development process, as the components should already have been developed in other projects. Furthermore, the journal defines Component Assessment as a process with high focus, which consists of labeling, testing and validating each component to ensure stability and usability. The last process introduced is the act of designing each component for reuse, which is not an integral focus in traditional software engineering.

1.2.1.9. Twenty-eight years of component-based software engineering [14]

This study is set on mapping the motivations behind CBSE, and the rewards gained from implementing the concept. It acknowledges that it is a large field of study, and that questions arise on whether advantages of CBSE have been clearly defined or if there are still a multitude of research topics that have not been concluded regarding the field. To perform the study, researchers have set up 5 research questions, which relate to the intensity of CBSE projects, main rewards, most investigated topics, different domains in which CBSE is applied, and the most frequent applied research methods. The book states that the main objectives highlighted in most of the journals concerning CBSE are increased productivity together with cost savings, with quality and reusability concerns also being large contributors. The main feature of CBSE is to create reusable components which may be used in future projects, and design systems to be able to implement already verified components.

1.2.2. Literature discussions

This section contains an analysis of the literature review. The literature found in section (1.2.1) is analyzed and compared against each other to highlight viable and relevant knowledge in the field, which may be of importance to this project.

1.2.2.1. Advantages of CBSE

An emerging trend in software development seems to be the implementation of Component-Based Software Engineering, CBSE, (or Component-Based Development, CBD) in simulator systems. Sections (1.2.1.1, 1.2.1.2 and 1.2.1.7) all reference technology modelled with this principle in mind and is a working demonstration on how well the concept is in use.

Even though CBSE is more rewarding in large-scale projects or businesses, many of the advantages should be viable for relatively small-scale projects too. As an example, in section (1.2.1.7), the importance of CBSE is shown when executing the simulations on computer grids, when visualization of the 3D scenario is not required (which may be the case in pure analytic and optimization-based simulations). This use-case is important in modern simulations, as users require different functionality such as 3D visualization versus graphing, to either observe the system or optimize components.

(Conclusions): Use of Component-Based Software Engineering should not only be viable for large projects, but should also improve stability, code readability, and effectiveness of the proposed simulator. It is expected that the advantages of using CBSE (gained from modifiability of project, separation of responsibility, processing efficiency and reusability of components) will outweigh the challenges (relating to time cost, complexity and version control). It is also believed that CBSE should increase the number of possibilities regarding expansion of the project when it comes to inclusion of further functionality.

1.2.2.2. Development focus for various modular robots

The physical models of modular robots have different design implementations depending on the various use-cases. The most recurring theme in most of the studies being performed on modular robots is the advantage of designing one module which is replicated, reducing cost of manufacturing and maintenance.

The MECABOT study's approach (1.2.1.4) uses modules containing a structural base (body) connected to a motion joint. This allows the modules to be configured for quadruped configurations as well as hexapod and snake configurations. The modules are designed for

reusability and sturdiness in a variety of scenarios including space exploration and maneuvering complex terrain structures.

The modules used in the Symbion (1.2.1.7) simulator project are more compact and customized, being a result of the research project's commitment to adapting bio-inspired behavior for their modular robots. This enables more flexibility but may increase costs relative to the MECABOT system.

NTNU Ålesund's research project (1.2.1.5) involves easily modifiable and simply designed snake-like modular robots, that can be developed with fast prototyping and are easy configurable. The design of the modules is less robust, with thinner components than in the MECABOT and Symbion projects, but with focus on modifiability there is a good tradeoff between cost and sturdiness. The sliding mechanisms for locking modules together is also relatively simple in design, again lowering the cost and complexity of assembly/disassembly.

(Conclusions): Studies regarding the optimal design of robot modules is an integral part of the proposed Modular Robot Simulator featured in this project. Being able to design different modules with varying materials and geometry for different terrain and obstacles will help researchers reduce the time and cost of physically testing proposed prototypes, and possibly implement more effective control algorithms for different use-cases and robot properties.

2. Background

2.1. Project framework

2.1.1. Unified Process

The Unified Process (UP) is a software development framework, which ensures iterative development by varying the workload of each process over the course of a project. With some of the processes being, as an example; research, implementation, and testing; the workload of these will be varied over the course of the project. In the beginning, there will be much more focus on research than on implementation, as research must be performed for the implementation to be started. However, where other models may complete the research phase before starting with the implementation (ex. Waterfall model), the iterative UP model encourages research during the implementation phase. Thus, the individual processes have more influence over the process currently in focus. The workload of each process varies over the course of iterations: Inception, Elaboration, Construction and Transition. Each of these iterations ensure an increment to the system, resulting in increased or improved functionality from the previous iteration.

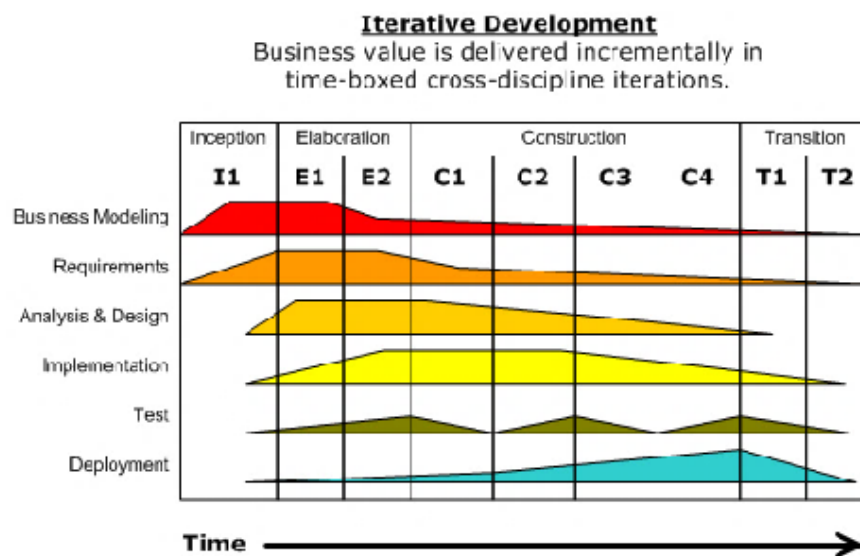


Figure 2: Iterative development model

This model is selected because it is designed for software development, ensuring progress for every iteration. UP also enforces project deadlines and milestones, making it easier to reach the project goals, and fulfill requirements. Another benefit of this model is that additional parts of the project may be added without much complication, since the system's base architecture will be implemented after the Elaboration phase. This should allow for testing new implementations or figuring out theories for other modules alongside the implementation of further functionality. As an example, if a new method for file transfer is proposed, this may be implemented alongside the current work in the analysis and design process. Since the architecture should be up and running, it will also be possible to implement the functionality, and evaluate the effectiveness of the new method. Another way to see this is that since the basic environment is created, additional features may be researched and tested.

In the beginning of the project, focus will be on creating requirements, while assigning priority values to each entry. These requirements must be formulated such that they may be tested later and show progress for the project. The requirements will be designed with input from the project supervisor and verified/validated through the Test and Verification specification.

Secondly, a RISK matrix and table should be created, to clearly identify which parts of the project are susceptible to failure or unacceptable behavior.

After each iteration, it may be feasible to create short iteration reports, highlighting the changes and goals reached for the specific iterations. This helps in giving an overview of the completeness of the project and serves to present the project progress. However, sprint reviews from meetings as mentioned in the agile methods (3.1.1) may be more feasible to use as iteration reports.

2.1.2. Phases

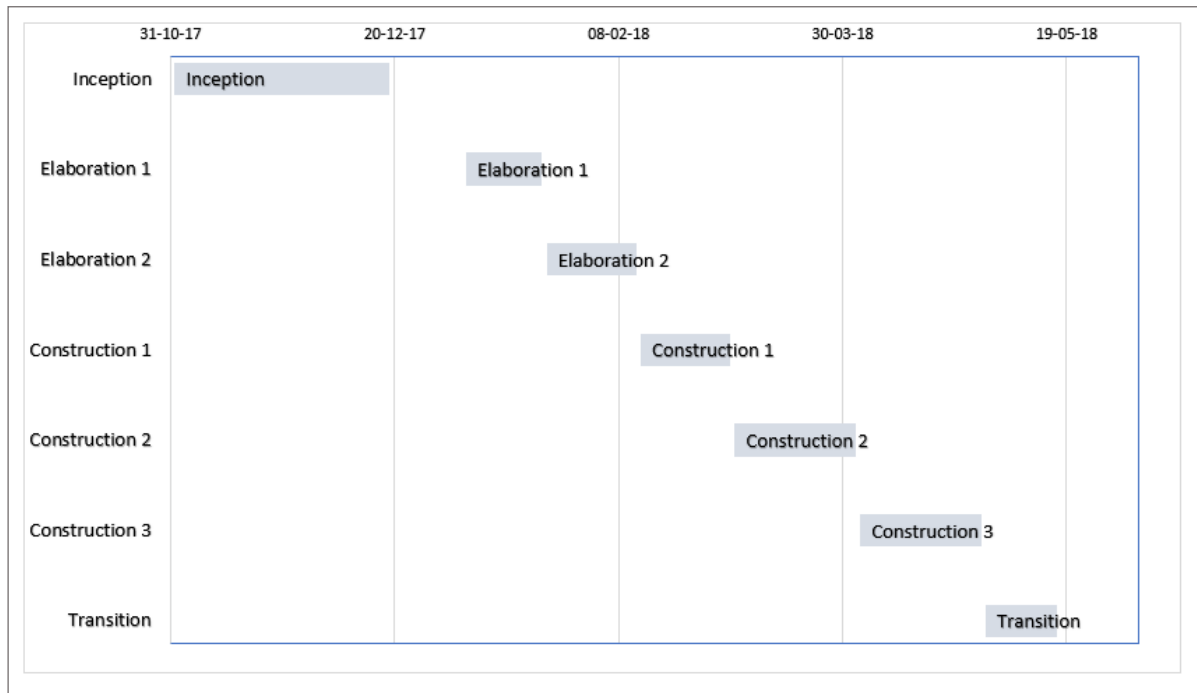


Figure 3: Proposed project plan

The four phases of the project contain different processes to be worked on during the specific iterations. Figure 3 shows how these are set up.

The inception phase, which serves as the initialization of the project, is performed before the official start date of the project. It can also be seen as an introduction to the project. Here, initial literature will be reviewed, and key concepts will be researched before the start of the thesis. The project scope and schedule will be defined in this phase.

In the elaboration phases, the focus is on setting up the working environment, in this case the basic functionality of the simulator. There will also be research on the other components of the system, and possible extensions to the currently proposed topics. There are two iterations of Elaboration in this project.

In the construction phases, all the theories and design proposals will be implemented in the project, as designed in the previous phases. There is also room for research. This phase is focused on the actual implementation, compared to elaboration which is focused on the preliminary designs theories. There are two iterations of Construction in this project.

The last phase is the Transition phase. This is where the project is refined, bugs are fixed, and the documentations are finished. This phase serves to prepare the project for analysis, review of results, and verification/validation. Thus, this phase's main workload lies on the report and evaluation aspect of the project.

2.1.3. Risk

UP has a big focus on the architecture of a system. One of the core tasks to be completed in the elaboration phase is an up-and-running base architecture for the program or system to be designed, so that requirements and the architecture may be tested and validated. Many projects using the UP are also heavily Risk-focused, serving to address the most critical risks early in the project. Thus, when the architecture is up, the preliminary RISK analysis may be finalized. As shown in Table 2, a RISK matrix considers the probability of an impact together with the impact effect. If something has a low probability, but a high impact, it will still have a medium risk value, as the project most likely cannot afford to ignore the element simply because it has a low probability for impact. Each risk entry will have an impact and probability value. The preliminary risk matrix is shown in Appendix A.

Impact	High	Medium	High	High
	Medium	Low	Medium	High
	Low	low	Low	Medium
		Low	Medium	High
	Probability of impact			

Table 2: Risk matrix

2.2. Modular robotics

2.2.1. Design

Modular robots are composed of several parts, moving in a specific manner depending on the current scenario to fulfill their goals. There are many ways of designing these robots, but as this project is focused on the chain architecture, the design of these will be the focus. Chain based modular robots form single or multi branched links, giving rise to multiple different configurations when creating the modular robot. They are mainly created with certain problem scenarios in mind and are thus modelled for these specific tasks.

Configurations:

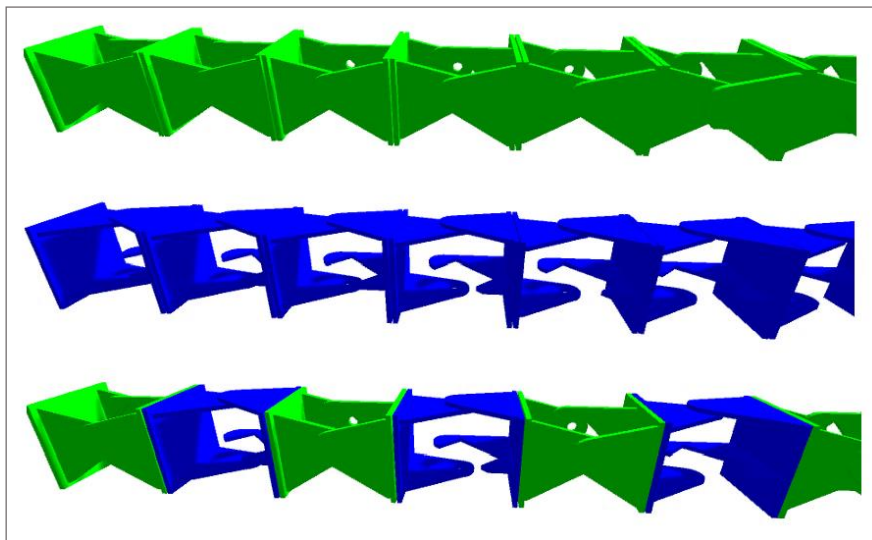


Figure 4: Modular robot; pitch, yaw, and pitch-yaw configurations

Using the chain architecture, there are two main axes in which rotational motion is performed: pitch and yaw. The robots may be connected in any manner that is suitable for the user, but the three standard configurations are: Pitch, Yaw, and Pitch-Yaw-configuration. Configurations outside of these three proposals do exist, with certain systems allowing for adjustment of module axis during movement. The modules are connected as required depending on the various use-cases of the robots, with configurations often sporting quadruplets or hexapods, most notably for use in space-based operations or traversal of complex terrain [9].

2.2.2. Dynamics

The motion of modular robots is mainly inspired from nature, with chain-based robots often modelled after snakes and caterpillars. The movement is often modelled as a sinusoidal function, with the rotation of each joint modelled as a function of time. An example of programming the movement for a modular robot is shown from the research paper provided by Dr. Guoyuan Li, from NTNU Ålesund [10], with a pitch-yaw configuration with 5 modules as shown in EQ (1) and EQ (2):

$$\theta_p(i, t) = A_p \cdot \sin\left(\omega t + \frac{(i-1)\phi}{2}\right), i \in \{1,3,5\} \quad (1)$$

$$\theta_y(i, t) = A_y \cdot \sin(\omega t + \frac{(i-2)\phi}{2} + \varphi_{py}), i \in \{2,4\}, \quad (2)$$

where θ is the reference angle for joint number “ i ”, A is the amplitude, ω is the angular frequency, ϕ is the phase difference for the selected configuration, and φ is the phase difference between the configuration connections. As these equations describe motion for two separate axes, the patterns of motion will be different depending on the configurations. One motion (turn) consists of forcing an offset angle on the Yaw-modules and only dynamically controlling the Pitch modules, allowing the robot to perform a right or left turn, depending on the direction of the offset. There is no limit for how the dynamics of modular robots may be implemented, and the example provided is just one instance. For this project, it is important to select a method that is easily replaceable or modifiable as the user should be able to implement custom scripting. This model for robot dynamics allows for a specific set of parameters, making it an interesting potential method for movement control.

The four main motion behaviors for chain-based robots are the following:

- Linear progression
- Rolling
- Sidewinding
- Turning

These models will be the focus for the robot dynamics and will be used to create the basis of a modifiable script for movement generation.

2.2.3. Optimization algorithms viable for modular robotics

The implementation of optimization algorithms is crucial for giving developers the tools they need to model modular robots. There are several different approaches to optimization, thus the Modular Robot Simulator should accommodate for implementation of the most common ones.

2.2.3.1. Genetic algorithm

Utilizing genetic algorithms for optimization of parameters is not something new, and has been around since 1950, when Alan Turing proposed a machine that could simulate the properties of evolution [15]. Since then, plenty of algorithms have been designed for different use cases, including for modular robots. To implement a genetic algorithm, one must first define a set of variables that may be modified for the selected system to better perform certain tasks. Examples of these variables in a modular robot system may be module weight and size, materials and joint lengths. However, variables may also be connected to the dynamics of the system, controlling the gait of the robot.

The variables for the genetic algorithm are often stored in an array or a list, in a specific order. These arrays/lists are referred to as chromosomes.

Population	Chromosomes					
Entity 1	a_1	b_1	c_1	d_1	e_1	f_1
Entity 2	a_2	b_2	c_2	d_2	e_2	f_2
Entity 3	a_3	b_3	c_3	d_3	e_3	f_3
Entity n	a_n	b_n	c_n	d_n	e_n	f_n

Table 3: Example of chromosomes in a GA system

A GA has 4 main functions: Fitness function, Selection operator, crossover operator and mutation operator. More operations may be implemented, but these are regarded as part of the standard GA algorithm.

Fitness function

$$Fit(s, d, e) = u(s) + v(d) + w(e)$$

Figure 5: Example fitness function

The fitness function defines the goal of the algorithm, and how well an entity has performed for each iteration. It is important to select a fitness function with the right variables, as this directly influences the time it takes to find the optimum solution, and how good this solution is. A good output for fitness can be some weights of speed, distance and energy used to get to the target. Having these as a fitness function with equal weights will in theory optimize these three parameters primarily.

Selection Operator

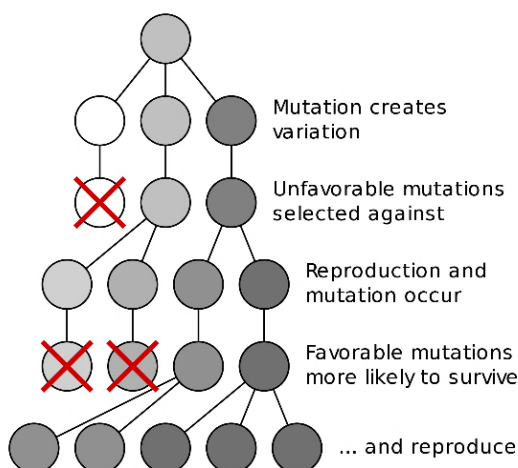


Figure 6: The mutation and selection process in a GA

The selection operator chooses which of the entities that will “survive” to create offspring, and thus pass their genes on. Often, the best solutions from the fitness function are selected, and the worst performing entities are killed off. Only the best solutions go on to breed, and the algorithm produces on average better results. Another method is to use probability to select chromosomes all over the spectrum, with a higher probability for entities with higher fitness to be selected. There are several ways to select entities, and specific solutions may be tailored to different applications.

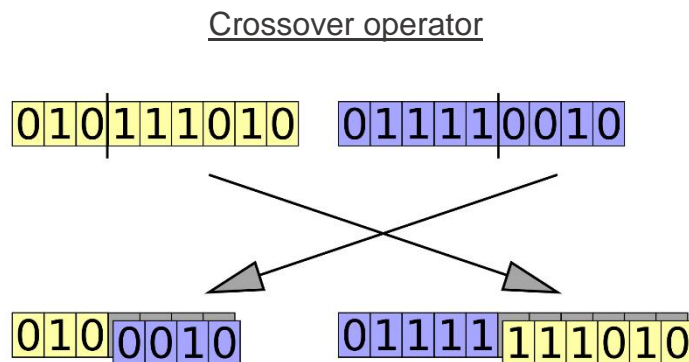


Figure 7: "Cut and splice" crossover operator

The crossover operator takes the chromosomes from the selection operator as input and produces children chromosomes as outputs. The function usually creates as many children as the amount of entities killed in the selection process, unless the mutation operator creates children on its own instead of modifying existing chromosomes. There are several different methods for crossover, including "cut and splice" and "uniform" crossover. "cut and splice" cuts off a part of two chromosomes and creates two children with one part of each parent chromosome each. The "uniform" crossover method assigns chromosomes to children randomly from each of the parents.

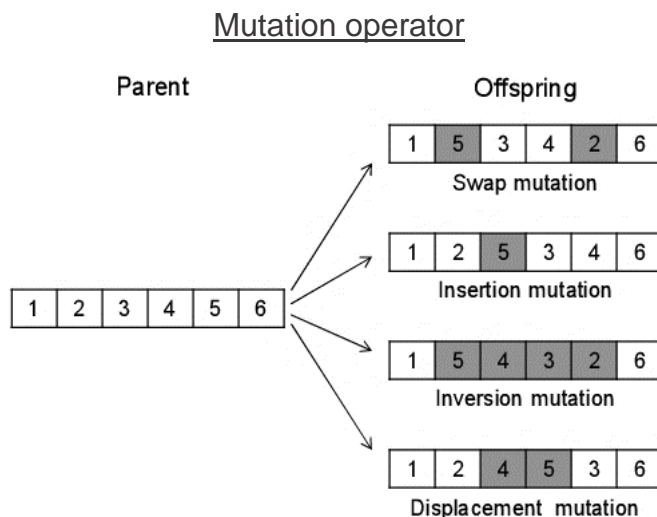


Figure 8: Examples of mutation operators
source: <https://www.researchgate.net/272093243>

The mutation operator applies changes to the chromosome, to prevent the formation of local minimums in the algorithm. By accepting completely new values to be added to the chromosomes, it is possible to find new unexplored combinations that would never have been found with just a crossover function. There are several ways to apply mutation, as seen in Figure 8, with the insertion mutation being the approach most similar to nature.

2.2.3.2. Simulated Annealing

An alternate approach to optimization for modular robots is the Simulated Annealing (SA) algorithm. The principle is based on annealing in metallurgy, where a metal is heated and slowly cooled to reduce defects in the material. In simulated annealing, the probability of accepting worse solutions as a function of the temperature “T” is compared to the cooling function in metallurgy annealing and is crucial to explore a wide area of possible solutions, reducing the risk of being stuck in a local optimum.

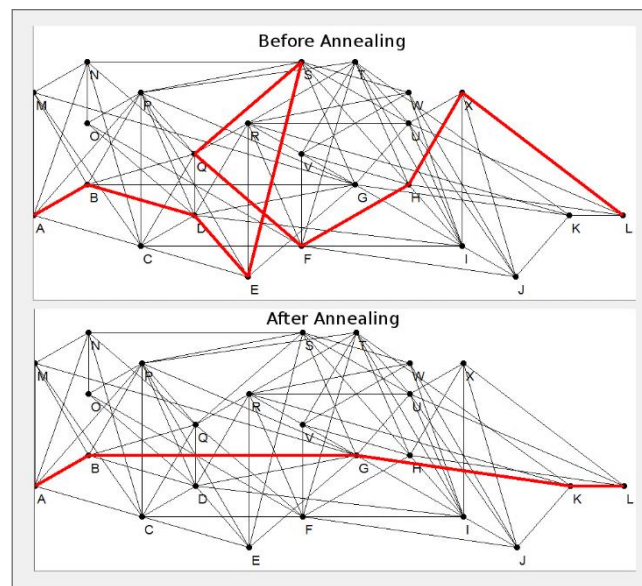


Figure 9: Simulated Annealing pathfinding example [from another project]

The SA algorithm starts with an initial solution, which can either be a list of nodes to be traversed or a list of variables to be modified, among others. The temperature “T” also starts with a certain value, and is decreased based on a custom function, which is often based on time. EQ (1.2.4) shows an example of a probability function:

$$P(S, T) = e^{\frac{-(S_1 - S_2)}{T}} > R, \quad (1.2.4)$$

where S contains an evaluation of the previous and the suggested solution, T is the temperature and R is a random number between 0 and 1. As the temperature decreases, the probability function will consider less and less values that go uphill, and mostly favor solutions that go downhill, making the start of the algorithm search for global solutions, while finding local optimums nearing the end. When the temperature has reached 0, the algorithm finishes while returning the final solution from the probability function.

In this project, SA may be used for finding the optimal robot parameters much like in the GA. By initializing the robot with a set of variables, and specifying which parameters that should not be changed, it is possible to perform annealing on the remaining variables, running

a simulation for each step, finding optimal solutions. It would also be interesting to research the algorithm's ability to modify the robot dynamic behavior. Lastly, it may be possible to create a pathfinding algorithm for the robot with the use of SA. This requires a fully observable overhead view of the environment.

2.3. Component-Based Development

CBD is the top-level design of a system, while CBSE is the Software Engineering aspect. This chapter contains both.

2.3.1. Introduction to CBSE

Component-Based Software Engineering involves creating components focused on reusability with possibility for implementing them again in other systems with little to no modification required. These components exist in the form of pre-compiled libraries or sections of code. CBSE is closely tied to modular programming and both provide similar functionality, with the main exception being that modules require no specific interfaces and may be viewed as additions, rather than features. Specific characterizations of the two terms vary based on developers' definitions. In this project the mentioned distinctions will be referred to as the standard.

As described in [16]: "A software component can be defined as an executable unit of code that provides a set of services through specified interfaces". The book describes the main goals of implementing CBSE as:

Cost Reduction:

Cost reduction is a top motivation for any project be it business or research based [14]. The implementation of CBSE in the correct way ensures development, testing, validation, and verification time is shortened by re-using the already tested components which have been through the necessary processes in previous projects.

Ease of assembly:

Segmentation of functionality facilitates the assembly process by assigning specific features to individual components. Correct use of the CBSE process ensures all components have a simple interface to the system, facilitating intuitive and quick implementations.

Reusability:

Reusability applies to both the programming level; re-use of functions and class frameworks, and the design level; re-use of architecture and design concepts. A popular tool

for modelling component-based systems is UML, which allows for representations of the system with class and flow diagrams.

Customization/flexibility:

When components are made available for the developers, the functionality of the system depends on the different assembly of components. If new requirements are discovered, modules may be replaced according to the developers' needs.

Maintainability:

By designing with components, systems become divided into sections, which enables troubleshooting specific parts of the code, rather than following the software flow. Bugs stay local, rather than system-wide issues. Deprecated modules may be replaced instantly when necessary.

Components are expected to offer certain services in a system. This creates different functionality for the system based on the assembly of components rather than changes or additions to individual code segments. By supplying independent component functionality, they can be ported to other projects with ease, and likewise receive new or modified components to extend their own functionality. The functionality of a component should be the same, whichever system it is deployed to.

2.3.2. Business use-case

Larger businesses often receive the most advantages of CBD, as they have bigger projects which may require more repetitive implementation than in smaller projects and they may already have, or plan to use components in several systems. If a company designs with modules/components, the modules may be re-used in other projects, shortening development time and cost. The cost-reduction goal is likely the most important factor for a company when deciding frameworks or development methods. However, it is a well-known fact that most software developers do not trust in code created by others [16]. This is one of the main challenges facing the CBD methods, as it reduces the probability for a component to be re-used.

Benefits of CBD mainly arise after extensive collections of components have been created, thus the great potential for large businesses. However, the benefits regarding ease of assembly, reusability, and maintainability are expected to be valid for projects, regardless of size. These three goals will be the focus in evaluating CBD in this small-scale project, and research will be conducted accordingly.

2.3.3. Component-based optimization

The customization and flexibility featured in the CBD process enables components to be removed if obsolete. A scenario where this is required is during the use of optimization algorithms, for optimizing movement or design performance. Since visualizing scenarios takes a lot of processing power, a more favorable solution is to cut out the visualization altogether, and only perform calculations in a closed system. Such a system is described in [11], where simulations are performed in a computational grid. Since optimization algorithms rarely need user interaction after initialization, the only visualization that will be needed is for showing the finished result after optimization has been performed. Still using CBD principles, the visualization module may be re-activated upon program completion, or shown in another setting such as a professional rendering software, for presentation- or advertisement-based uses.

3. Methodology

As described in the introduction (section 1.1), the focus of this research is to enable realistic and easy design of snake-based modular robots, in addition to studying the component-based development methods in relation to the simulator architecture and small-scale projects in general. Several methods have been used to study the proposed research problems, while at the same time answering or discussing the research questions provided in Section (1.1.5). The methods are as follows:

Section 3.1 contains methods for the development of the simulator, such as the project framework and work schedules, in addition to research performed relating to key aspects of the project such as the modular robot domain, and software methodology.

Section 3.2 describes the verification and validation part of the project which involves the project requirements, tests and verification specification of the system and core framework, and validation of the complete modular robot simulator.

Section 3.3 lists the tools, platforms and libraries used, including the hardware used for development and testing, software components and the development environment/libraries.

3.1. Development

This section covers the methods for developing the Modular Robot Simulator including research, the unified process, component-based software engineering and general research strategies.

3.1.1. Project methods

The project development method, Unified Process, has been used in tandem with an agile development method to ensure the project going according to plan. Unified process has been selected for the system development and prototyping, and the agile method for status updates, input from supervisors and short-term goals.

The agile method consisted of sprints every two weeks, with sprint reviews and retrospective meetings with supervisors after each sprint to keep everybody up to date and evaluate the quality of supervision. These sprints complimented the iterations in the UP, allowing for feedback more often than just after iterations, and potential modifications to be performed in good time before deadlines. In the elaboration phase, the increased feedback aided in the planning of all the project phases and refining the required features of the simulator. In the construction phases it aided with increased feedback and troubleshooting, by letting supervisors be regularly updated on the progress and developed features of the system.

3.1.1.1. Unified Process

A roadmap has been created to show the design phases of the project, as shown in Figure 10. Even though selective content is the focus for a specified phase, the other parts of the system are worked on simultaneously, as defined in the UP (2.1.1).

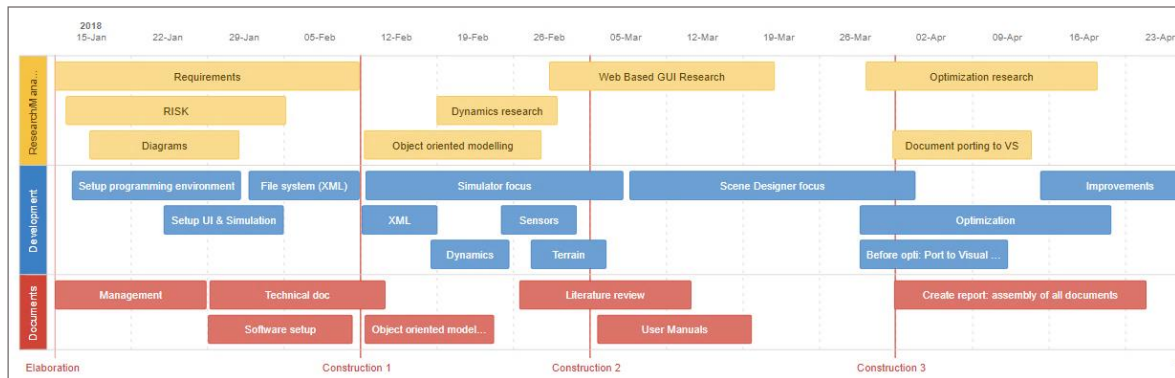


Figure 10: Project development plan

In the Elaboration phases, the focus has been the requirements, setup of the environment, file system and relevant documents. Also, as is a standard in the UP, a prototype architecture was finished by the end of the Elaboration 2 phase.

In the first construction phase, the focus was on the functional simulator framework, including all the main features the simulator has. In the research and management field, theories about object-oriented modelling were the first topics, with dynamics research the second, to prepare for the implementation of the robot dynamics. In the second construction phase, the focus was on the scene designer and further improvement of the work from phase 1, and in the third construction phase, the optimization algorithms were the focus, along with research relating to the topic.

The transition phase focused on finishing the main project components and ensured a stable system, together with the documentation of findings and finalizing the technical specifications.

3.1.2. Research and planning

Research was performed with regards to software frameworks to provide the most robust and flexible solution for the simulator system. A comparison study on traditional vs. Component-Based Development [17] highlights the advantages and disadvantages of the two approaches, with the segments comparing the easy to implement- (traditional) versus easy to re-use- (component-based) systems being the most relevant for this project. Research into other modular robot simulators, as in section (1.2.1.8), also showed motivating advantages of CBD.

There are multiple ways of designing simulators and programs in general but using a straight-forward development method did not seem like the correct way to approach this problem when thinking long-term solutions. Thus, the focus shifted towards a component-based engineering approach, which would not only enable easy modification of features but also potentially improve reusability of components, improve efficiency of operations such as optimization algorithms, and enable extraction of key-parts of the software to be used in other projects.

Use of the CBD method in turn led to the separation of concerns in the core-architecture, resulting in the Algoryx Interface class, and the Simulation Core class. Rough sketches of the class structure were designed, and later revised into the class diagrams shown in section (4.1.3). Towards the end of the second elaboration phase, technical documents were created as documentation and concept plans of the simulator.

Throughout the entire project, the focus has been on finding new methods and expansions to the project, to ensure the quality of the finished work. Thus, various research phases have been included whenever a new possibility for improvement has been discovered. This includes both research that has been implemented and conceptual research that has not been included in the final prototypes but might be interesting for future work. Information has been gathered from literature on the current state of the art regarding technologies and features implemented in the final system, but also from community-based forums, with questions related to simulator development and software implementations.

3.1.3. Modular robotics

Research regarding modular robots has been performed by studying relevant and new literature, as described in (1.2.1.4-1.2.1.7), regarding the performance, use-cases, and dynamics of these. The main inspiration for the modelling has come from the NTNU research project; both from simulation code, simulation visualizations and physical 3D-printed models of the

robot modules. The dynamics research comes mainly from biologic research on snakes and their movement patterns, together with analysis and redesign of the NTNU project's dynamics code. Relating to the scene and terrain, most research comes from exploring the possibilities of simulations and image-based heightmaps. Lastly, a lot of research has been performed on Unity and Microsoft's libraries to create custom class structures for variable types, and functions for features not covered by the Algorix library, like distance measuring in the scene and image-loading for terrain.

3.2. Verification and validation

3.2.1. Simulator Specifications

3.2.1.1. Requirements

There must be thorough requirements describing the necessary functionality of the simulator before creating the test specifications. The requirements have been based on the needs of the professors working in the current research group at NTNU and the capabilities of the group's existing modular robot simulations, together with requirements derived from research and functionality issues. When the requirement specification was finished, the test specification was created to enable actual testing of the components and ensuring the requirements were met. When testing the system, text fields in the test-specification were filled in with the results of the tests, and the fully filled-out document became the Test and Verification specification.

As mentioned, requirements came from both the NTNU research group and also from the general functionality which is required of simulators. Some requirements were also defined from analyzing the manually-created modular robot simulations received from the research group.

The requirements have been divided into two main topics based on different aspects of the requirements; Simulator requirements, and CBD related requirements. Under these topics, there are sub-topics with requirements regarding specific parts of the system/sections. To rank the requirements, a three-letter system has been implemented, as shown in Table 4:

A	The requirement shall be met to ensure a stable component
B	The requirement should be met to ensure an efficient component
C	The requirement is optional or flagged as further work

Table 4: Requirement importance

The actual requirements consist of table entries, as shown in Table 5. Individual requirements reside within the sub-topics, and requirements within these are often tested together in the test and verification specification.

Nr:	Requirement	Category	Originator	Verified?
REQ-1.1.1T	The user shall be able to save scenario configurations	A	NTNU	Verified T-1.1.1
REQ-1.1.2T	The user shall be able to load scenario configurations	A	NTNU	Verified T-1.1.1
REQ-1.1.3T	The user should be able to load a scenario	B	TS	Not verified T-1.1.2

	stopped mid-execution.			
REQ-1.1.4R	The saved robot values should be represented in a format facilitating potential prototyping	B	TS	Partly Verified R-1.1.1
REQ-1.1.5R	All aspects of the simulation shall be transferrable in one single XML file	A	TS	Partly verified R-1.1.1

Table 5: Requirement example

3.2.1.2. Test and Verification

The test specification is the document describing the various tests which are performed on the simulator when the final prototype scheduled for the project is complete. The main goal of the test and verification specification has been to ensure the system meets the requirements. To verify the requirements, the test specification has been filled out with the results from the test-procedures. Thus, the name “Test and Verification Specification”. Each test will be based on at least one requirement.

There are three different categories of tests, as shown in Table 6:

T	Test verification
R	Review verification
U	Use-case analysis verification

Table 6: Verification criteria

T-1.1.1	REQ-1.1.1T	REQ-1.1.2T
Pass criteria	Configurations are saved to file, then loaded in a new simulation.	
Method	5 different configurations are created and saved. The program is shut down, and configurations are loaded to the simulator. Repeat 5 times with the different configurations. Analyze the ease-of-use.	
Result	Verified	
Comment	Performed in build application.	
Discussion	A file browser instead of typing file name directly as in this prototype scene designer would be more user-friendly.	

Table 7: Test specification example, fully verified

3.2.1.3. System validation

A comparison study has been performed to validate the requirements and test verifications, where the focus points of the simulator are compared with the current technology in the field. In this case, it was the current robot simulations created by NTNU Ålesund's research group on modular robots. A test and verification specification has been constructed for the study so that advantages and disadvantages could be evaluated against each other, comparing both overall results and individual features within the simulator. These studies are marked with the letter "U" in the Test and Verification Specification.

3.2.2. Framework and simulator verification

There is also a test case for some of the components in the simulation framework. Specifically, the Core Framework components as these are the central components in the project. Here, the focus is on the component-based development, verifying if there are any benefits of implementing this method while checking the difficulty of creating new custom projects with the developed framework.

The component-based focus in this project has led to some additional possibilities regarding testing, outside of the expected features of a simulator. Most notably, increased extendibility and modifiability is expected to emerge from the design process. As a result, the core framework of the simulator application is extracted in the form of "dll" files and used as a library for creating a new simulator project with other visualization tools or scene designers. Thus, the test scenario involves creating a custom project with the simulator framework and checking the feasibility regarding the development of new projects with it.

Case-studies have also been performed to verify the functionality of the simulator and the framework. These demonstrate scenario creation, robot dynamics and new simulator creation.

3.3. Tools, platforms, libraries

3.3.1. Hardware

The software has been developed mainly on two separate computers with different specifications. This has allowed for speedy development on the high-end computer with more powerful hardware whilst testing the overall usability with a the relatively lower-end computer. This enabled quick troubleshooting of parts of the system which were bottlenecking the simulator, since the low-end system was used regularly throughout the entire project for development. Code sections that were not well enough optimized for the lower-end system have been optimized where possible. It is expected that users of the simulator will not use hardware of lower grade than what the simulator has been tested with, thus validation and verification performed on the low-end system should be considered valid. However, lower memory might have been more suitable for the low-end system. Table 8 shows the test systems employed in this project.

System	Desktop (High-end)	Laptop (Low-end)
CPU	AMD FX-8350	Intel i7-4510U @ 2.0GHz
Cores	8	2(4)
GPU	Nvidia GTX 970 4GB	Nvidia GeForce 840M 2GB
Memory	8GB	8GB

Table 8: Test computers

3.3.2. Software components

Several software components needed for the project functionality were not included in the Unity and AgX collection of libraries and had to be implemented manually:

ObjImport

The “ObjImport” class was used to import the “.obj” files containing the meshes of the robot frames. Unity has no functionality for importing these from scripts and having non-modifiable meshes would severely limit the functionality of the simulator.

System.drawing

Separation of concerns is an important part of this project, to the extent that the core framework should be portable to external libraries, for use in multiple systems. To reduce file size and ensure usability over multiple platforms, the libraries utilized in these classes should be from Microsoft’s collections. However, some of the libraries are not automatically included in the standard “system” library and must be imported manually in the form of dlls. This was

the case of the “system.drawing” class which was needed in order to read pixel values from the heightmap image generating the terrain in the scenario.

Heightmap values

There are no officially available libraries that convert image files into heightmaps. Thus, a custom function had to be created to fulfill this requirement. An example solution was found on Unity’s community page [18], and modified to suit the needs of the terrain component. The original code created Unity meshes from an image by assigning to the mesh components (triangles, uvs and vertices), while for this project it was modified to store these in the scene class object, using the “system.drawing” class for bitmap storage, and ensuring the correct size of the terrain.

Additional build requirements

When building the .dll files representing the individual components it is important to build with .NET version 3.5.

Also, since “System.drawing” is a custom imported library, Unity does not recognize the dependencies when building the application (.exe project). Thus, the “System.dll” file found in the Unity install location (Unity\Editor\Data\Mono\lib\mono\2.0) must replace the file from the build folder: (\Managed\System.dll).

The Algoryx version used in the project is AGX-2.21.1.2. Users may have to upgrade their version to AGX-2.21.1.1 version+ to make all the functions work.

3.3.3. Algoryx Dynamics

<https://www.algoryx.se/products/agx-dynamics/>

To create a realistic simulation of the modular robots, and for correctly optimizing the required components, standard integrated physics found in most visualization- and game-engines will not be satisfactory. As these are designed to perform efficiently and in real-time primarily, they are actively programmed with reduced accuracy and functionality, in favor of speed. For a prototyping tool, it is more important to have accurate physics simulations, than real-time visualization. Thus, the need for a custom physics engine, like Algoryx Dynamics (AgX).

AgX is a highly realistic physics engine used for professional simulations. It is world leading when it comes to wire simulations and features much higher accuracy for computations than other conventional and open-source physics engines. The corresponding library consists of hundreds of C++ classes with intuitive and portable code, allowing for the use of virtually any programming language, with the use of correct references to the standard C++ code. To implement a custom physics engine into an already existing visualization platform, it is simply a matter of importing the required runtime libraries and launching the program with path variables referencing to the physics engine's file locations. When the programs are set up, simulations can be created as usual except for the different library structure of the custom physics engine, and the linking of visual objects to the corresponding physical objects. Also, where the integrated physics colliders often allow for behind-the-scenes updating of visuals, this must be done manually with a custom physics engine.



Figure 11: Algoryx simulation of a modular snake-like robot

3.3.4. Unity

<https://unity3d.com/unity>

AgX contains a simple visualization tool for outputting graphics of a simulation, with the company logo featured as a watermark on the screen. However, the graphics engine is limited when it comes to implementing user interfaces and modification, together with the relatively low visual quality of scenes. Since this project has a strong focus on user-friendly interaction, together with a high level of portability and modular design, it will be advantageous to use a game-engine such as Unity to visualize the simulation. Other solutions have been considered, such as the Java-based JMonkey-Engine. However, due to the lower visual quality and lower maintenance standards, these have been rejected.

Unity is one of the leading platforms for game development, due to its ease-of-use, combined with excellent visual quality. The software is free to use (assuming little to no income is generated from projects), and implementations are well documented, thanks to the large number of developers on the platform. The programming can be performed in JavaScript or C#, and the latter will be used for this project. Support for NET 3.5 gives access to most of the functionality that C# has to offer, which will be a great addition to AgX physics.

Both AgX and Unity's architecture is designed in such a way that porting the software to other systems should be possible, especially considering the programming language (C#). This has made it possible to import the code and library files to other C# based programs using Visual Studio, facilitating the ability to change to a DirectX-platform, or performing simulations on dedicated servers with no visualization. This is all assuming the code itself is created in a manner that allows for reusability, as was the intention with the CBD method.

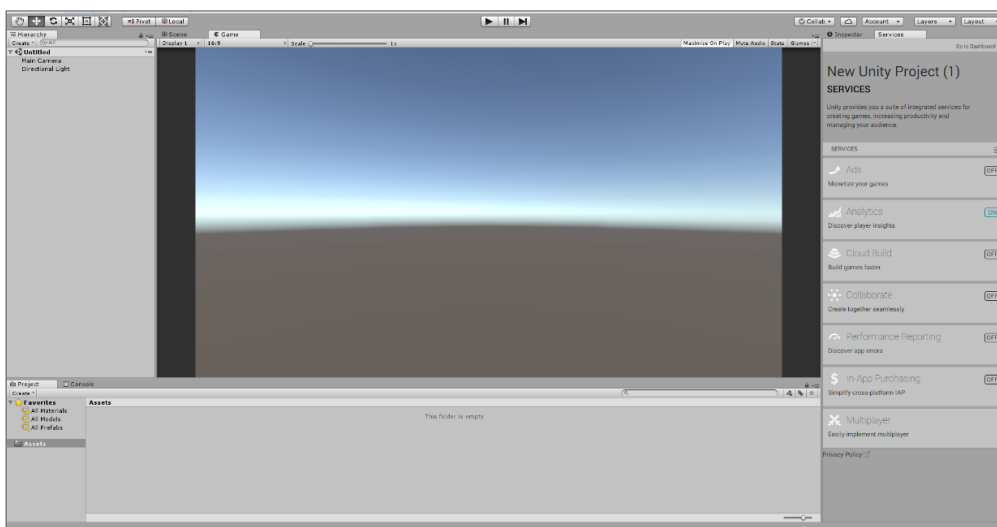


Figure 12: Unity editor

4. Results and findings

4.1. Design of the modular robot simulator

This section contains the design of the simulator. The first chapter details the steps to set up the editor and application environment, the second chapter contains the results of the component-based design process implemented in the project, the third chapter is the technical specification of the simulator framework, and the fourth chapter details the usage of the simulator framework and how robot/scenario creation has been implemented.

4.1.1. Setting up the software environment

1. To set up the platform/application environment, Algorix needs to be installed on the computer as detailed in the user-manual:
<https://www.algorix.se/documentation/complete/agx/tags/latest/UserManual/source/installation.html>
2. After the installation, the license file is pasted into the Algorix installation folder (Failure to do so will result in the physics engine not starting and producing errors), followed by adding the Algorix installation path to the system environment path variable.
3. Next, the agxDotNet.dll file is added from the “\bin\x64” folder to the “plugins” folder in the Unity project.
4. Visual studio 2017 must be installed on the computer, in order to run the environment setup in the next step.
5. Lastly, a command-window script is created as shown in Figure 13, to link the environment variables to the selected application in which Algorix physics will be running.

```
cd C:\Program Files\Algorix\AGX-2.21.1.2
call setup_env.bat
"C:\Program Files\Unity\Editor\Unity.exe"
```

Figure 13: Setting up the environment variables on the test computer.

The environment setup script must be executed every time the simulator application or editor application is started. An easy way of ensuring this is to start the setup_env.bat script and reference it directly to the executable file of the program. While developing the project, the setup is referenced to the Unity editor executable, while with a built application, the corresponding executable file must be referenced.

The next task was to find a suitable development environment for the simulator, which would enable quick prototyping without requiring too much effort to get familiar with. Two game-development platforms were reviewed: the java-based JMonkey Engine, and the C# based Unity engine. The latter was chosen because of previous experience with this software. In addition, the Unity platform contains more openly available documentation and a larger development community. However, any development platform featuring C# as the programming language should be able to use Core Framework and have the same results and features as this project. “.Net 3.5” versions of libraries from Microsoft are the ones implemented, as this is the version that Unity supports.

4.1.2. Component-based design

The main result of the component-based design is the trinity of task separation in the simulator, which is a result of dividing the functionality of the simulator into the three distinct components: Visualization, Simulation variables, and Physics calculations. The two latter (referred to as the Core Framework) are the most thoroughly designed aspects, as visualizations may be implemented later in various forms such as graphing, statistics, or 3D renderings, which all rely on a stable simulation to be effective.

The component-based architecture is a focus in designing the simulator, as the implementation also allows for more portability and modularity in the programming structure. As many program modules in the system may be changed in the future, having a component-based design ensures less programming is needed to introduce the new modules, and when it is implemented, the module should function flawlessly with the rest of the framework. Still, the concept of components is not limited to the individual objects.

The smallest components in this system are the classes containing information about individual objects. The largest component is the core framework itself, as it is usable without modification with custom developed Scene Designer applications, such as web-based solutions.

The main individual components of the simulator system referenced to in this paper are the classes and namespaces containing key functionality of the simulator, such as the “Simulation_Core”, “Agx_Interface”, “Dynamics” and “Optimization”. Top-level components in the Modular Robot Simulator are shown in Figure 14.

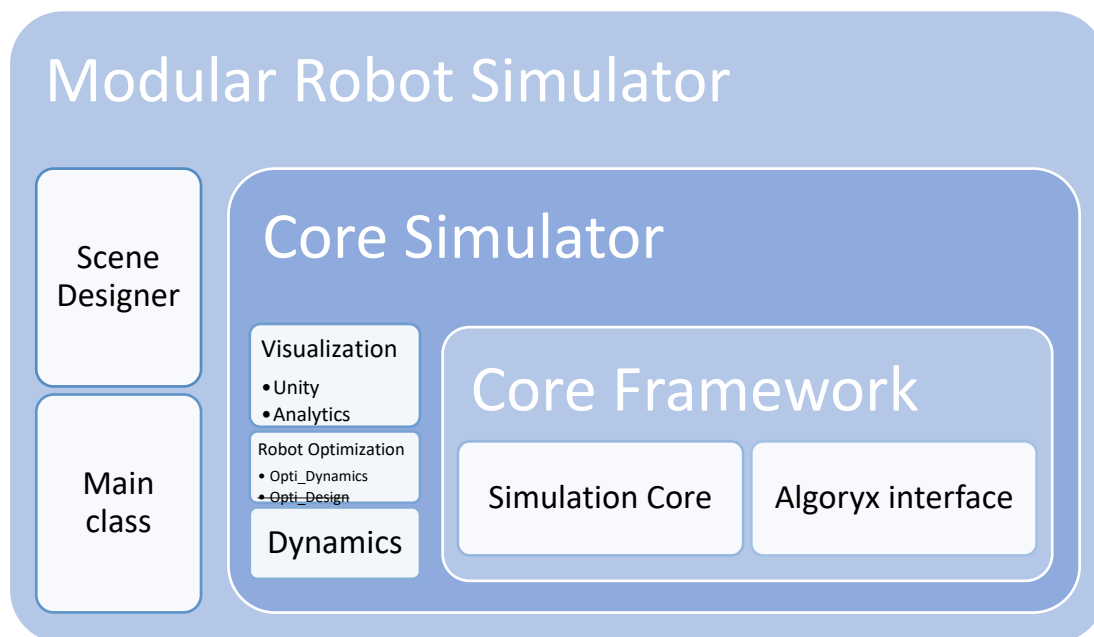


Figure 14: Modular Robot Simulator components

4.1.2.1. Simulation Core

The central functionality of the simulator is located in the data container namespace, “Simulation_Core” (4.1.3.1). This contains the classes that store all the variables required for the physics calculations and visualization of objects in the scenario. The objects created from these classes are in the center of the whole simulator system (core), and the objects from the physics handling (4.1.2.2) and visualization (4.1.2.3) namespaces will request information from these objects to perform their designated tasks. The core-objects will have the same functionality regardless of which technologies and methods are used within the other namespaces/system components. It is up to the Algoryx interface classes to have the correct attributes and function return values sent to the core-objects.

Components in the Simulation Core namespace are the classes containing object information, such as Robot, Modules, Joints, Frames, etc. these classes also contain components of other classes, such as the “Robot” objects which hold several modules, sensor modules and joints. The Simulation Core objects function regardless of the status of the Algoryx Interface or visualization objects, enabling visualizations without physics calculations, and vice versa. When the Initialization functions of the objects are called, the objects are added to physics calculations and are updated accordingly. Visualization is performed by retrieving component data.

4.1.2.2. Algoryx Interface

The physics handling namespace (4.1.3.2) is named based on the current physics library in use for the simulator. The name was chosen to represent the functionality of the classes, which is to handle the creation of physics objects and conveying the necessary information into C# based class objects. The library contains the physics objects from the Algoryx simulation, created by methods accessed by the Simulation Core, with return methods retrieving data from the physics objects. In the current design of this platform, the physics handling namespace is named “AgX_Interface”. However, in future versions it may be replaced with interfaces to other physics libraries, such as “ODE_Interface” (Open Dynamics Engine). Since the architecture is based on the component-based design, the only code requiring modification will be the namespace name and class names (In the Simulation Core namespace).

Although renaming the “AgX_Interface” namespace and its underlying classes to a less library-specific name might seem intuitive, this will reduce the readability in the system, as there will be no indication which physics library is in use. The amount of work required to rename these fields in the core-classes when libraries are changed will be minimal and will keep the code more readable.

The components in the library are the classes containing the required simulation objects; Joints, Frames, Scene, Simulation, etc. Based on which physics library is in use, the Simulation class may be static, as it only needs to be instantiated, and the objects added to the physics calculations of the Simulation. The rest of the classes reference the objects with the required syntax defined by the physics library. When the objects of the classes have been created, there should be two separate objects: an object residing in the physics calculations (again defined by the current library), and an object of the physics handling namespace “AgX_Interface”, linking to the physics object, which will contain all the information necessary to transmit relevant data to the Simulation Core objects.

4.1.2.3. Visualization

The visualization class (4.1.3.3) is the component which is designed using the current development environment's proprietary libraries, as it directly relates to the visualization program currently in use. As such, Unity's libraries and game object classes have been utilized to develop visualizations for the various objects created in the Simulator Core framework. Each object created in the Visualization, Simulation Core, and Algoryx Interface have a GUID (Globally Unique Identifier) which is used by the visualization classes to identify which object

is currently being created/updated, since the decentralized architecture does not encourage direct object referencing, in favor of modularity.

In the “Main” class (4.1.2.6), the robot objects are updated (following a simulation step forward), following the update of visualization objects with GUIDs corresponding to the ones of the core objects. Usage of the GUID variable is recommended throughout all implementations of visualization classes in further development or new projects using the MRSim framework.

4.1.2.4. Dynamics

The “Dynamics” class (4.1.3.4) is the component which controls the movement of the joints between each frame of a robot module. To separate the class as much as possible with regards to the component-based approach, it has been made to control all aspects of any movement of the robot. All robot attributes are passed to the dynamics class, such as number of modules and the joints to rotate. The dynamics class and its functions should be called from the main class, with the input parameters being the current robot being simulated. However, since the class is independent and not a reference in the robot assembly, the “Dynamics” class can be used to control the movement of several robots at the same time, if necessary. This assumes the availability of a list of robots, which is possible to implement as further work. If the goal of the simulator is to enable as quick and simple a simulation as possible, the “Dynamics” class has the functionality required to be called from the “Robot” object. However, this limits the user/developer’s different customization options regarding the movement variables.

4.1.2.5. Optimization

The “Robot_Optimization” class (4.1.3.5) is the component enabling the user to optimize the robot dynamic variables. As with the “Dynamics” class, the class is separated from the rest of the framework, allowing it to execute at any point in the simulation or not even be included at all if desired. The functions within the class have been implemented with focus on modifiability, allowing a developer to easily change the Genetic Algorithm’s properties, or even replace the GA with another method altogether. Optimization functions use a modified version of the “Dynamics” class, where the corresponding class object is being modified by the algorithm, and the genome of the GA is the dynamics parameters.

4.1.2.6. Main

Each project created with this framework require a centralized script or class to perform the operations required to set up and run the simulations. In this project, control has been realized with a “Main” class deriving from the Unity “MonoBehaviour” class. This class is the connection point between all the components described in the previous chapters and executes all the functions necessary for the simulation to initialize, run, and finish. A good example of the role of the “Main” class is the update of visualization objects, based on the “Guid” values of the corresponding “Simulation_Core” object. The reason for implementing a “MonoBehaviour” class is that it contains specific features for executing code at program start, and time-based repeat functions which update the simulation step and robot “Update” functions in Unity.

4.1.3. Software design

The core framework has been developed to create modular robots and put these in an environment within a physics simulation. The creation of these objects is performed separately in a Scene Designer script and serialized into the XML file, as shown in section (4.1.4.3), which is the only file/data that is transmitted before the startup of the simulator application. Thus, with the XML file correctly created, the simulator shall function independently with no additional requirements from outside of the simulation environment, except for a program initialization call (REQ-1.1.5T).

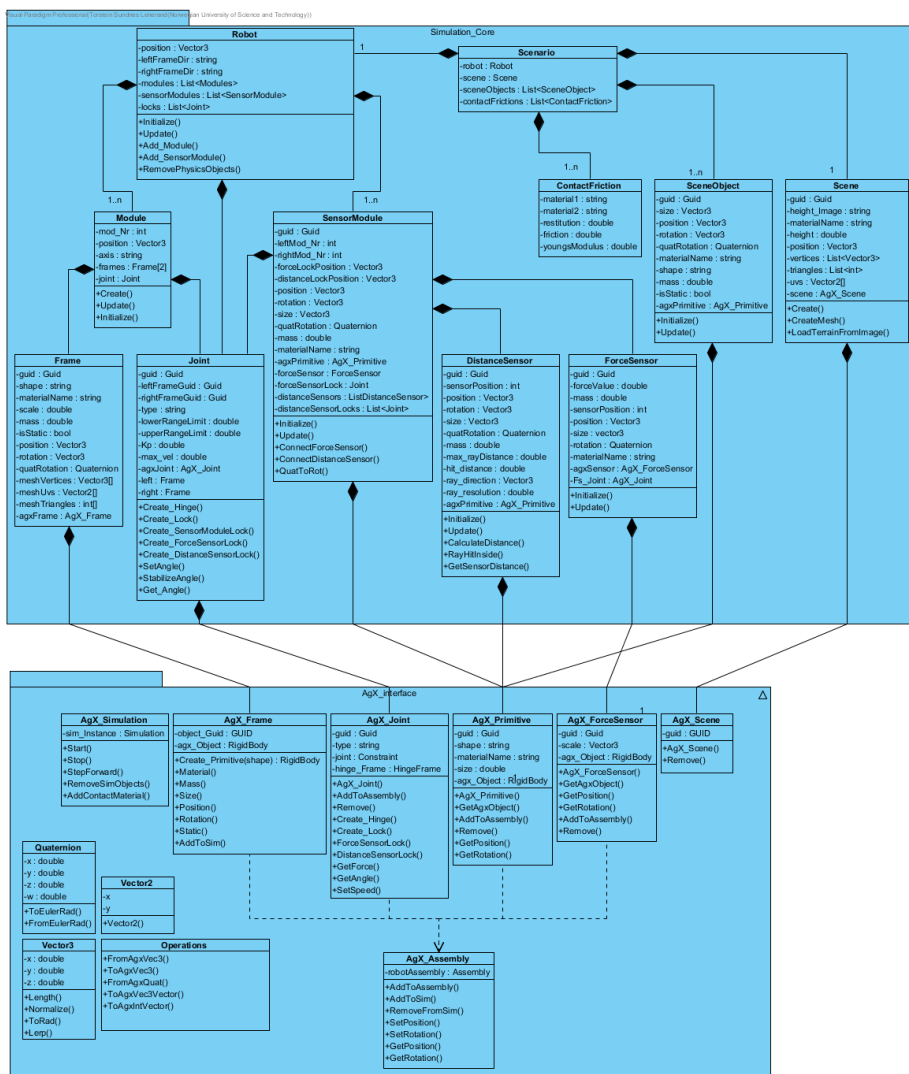


Figure 15: Core framework software architecture

The focus when designing the simulator was effectiveness and a structured architecture. The requirements for the system state that the program shall provide a stable physics simulation (REQ-1.2.4T), meaning that well-planned and structured code is of the essence, along with clearly defined system components. The Component-based design approach has facilitated a

structured architecture for the core framework, as shown in Figure 15, which makes object creation and connection with sub-components easy to implement. The full-size figure can be studied in Appendix D.

The usage of components improves ease of use for the class if it is to be implemented in other projects in the future and allows for separating the two namespaces into different libraries for further distribution. The “AgX_Interface” namespace is completely independent with no dependencies from external sources, except for the standard Microsoft C# libraries. Thus, the dll can be extracted and used as a quick prototyping tool using Algorix simulations in external projects.

Classes not shown in the following chapters are found in Appendix E.

4.1.3.1. Simulation Core

The “Simulation_Core” namespace contains all information about the scene and objects. Except for the “ContactFriction”, all classes contain at least one component in the form of “Simulation_Core” objects, or “AgX_Interface” objects. The classes in this namespace are the center of the simulation, by containing all the information about the scenario, and having no other dependencies than the Algorix wrapper namespace. Thus, any visualization may be used in tandem with these classes, even without using the “AgX_Interface” objects, as has been done when visualizing the creation in real-time from the Scene Designer (Section 4.1.3.6) (without physics interactions).

4.1.3.1.1. Scenario

The “Scenario” class is the top-level class containing all information about the current simulation scenario. With no attribute variables or functions, its sole purpose is to be a wrapper class for the “Robot”, “Scene”, “SceneObjects” and “ContactFriction” objects in the scenario. The object of the scenario class is the one which is serialized into the XML file, with all other sub-components being hierarchically listed inside of the respectable class fields.

Contents of the “Scenario” class are displayed in Table 9.

Attributes	Components
none	robot : Robot scene : Scene sceneObjects : List<SceneObject> contactFrictions : List<ContactFriction>
Functions	none

Table 9: Contents of the Scenario class

4.1.3.1.2. Robot

The “Robot” class represents the created robot containing all the modules in the current configuration to keep track of all the parameters required for simulation. This class is the main way of interacting with the robot from the main/control script, with access to modules and joints through the sub-objects in the specific object lists. The “Robot” class contains all the required functions for adding modules and sensor modules. This class also contains the file directories for the frame meshes, in case a developer requires these. The class contains a list of locks, which can take any form (Hinge or Lock), and a certain number are assigned after sub-component allocations, while they are initialized with actual “AgX_Joint” joints in the form of hinges or locks during the initialization function of this class.

Contents of the “Robot” class are displayed in Table 10, while the main functions of the class are described in Table 11.

Attributes		Components	
position : Vector3 leftFrameDir, rightFrameDir : string		modules : List<Module> sensorModules : List<SensorModule> locks : List<Joint>	
Functions			
Initialize()	Add_Module()	RemovePhysicsObjects()	
Update()	Add_SensorModule()		

Table 10: Contents of the Robot class

Function	Description	Return
Initialize	Calls the initialization function for all modules and sensor modules in the “Robot” object. For each module, depending on any sensor module having that module to the right or left of itself (defined in sensor module attributes), locks together the module and the sensor module. Adds the completely initialized robot with all its sub-components to the simulation via the “AgX_Assembly” class.	void
Update	Updates each “Module” object in the list of modules. Retrieves the position of all modules and uses it to calculate the overall position of the “Robot” object. Updates each “SensorModule” object in the list of sensor modules and force sensors on these.	void
Add_Module	Adds the function parameter’s module to this “Robot” object’s list of modules. If there is an additional parameter in the function for a “Joint” object, it is added to the list of locks in this “Robot” object. Gives the module an axis label based on its orientation in the global x-axis.	void
Add_SensorModule	Adds the function parameter’s sensor module to this “Robot” object’s list of sensor modules.	void

	Adds the function parameter's lock "Joint" object to the list of locks. If there are two "Joint" objects in the function parameters, both are added to the list of locks.	
RemovePhysicsObjects	Removes all physics objects from the Algoryx simulation, and all the "AgX_Interface" objects.	void

Table 11: Robot class functions

4.1.3.1.3. Module

The "Module" class contains the assembly of two frames and one joint object and builds up the shape of the robot. This class is included to better show the structure of the robot, and to modify a specific module, instead of individual frames and joints. Because of the modularity, it is easier to add multiple modules to a robot, which in turn makes modification of the code easier. The modules can be attached to other modules or sensor modules, by lock connection to the module's frames. Thus, the architectural hierarchy of the module class allows for easy access to the sub-components.

Contents of the "Module" class are displayed in Table 12, while the main functions of the class are described in Table 13.

Attributes	Components
mod_Nr : int position : Vector3 axis : string	frames : Frame[2] joint : Joint
Functions	
Create()	Update()
	Initialize()

Table 12: Contents of the Module class

Function	Description	Return
Create	Assigns the left and right "Frame" objects. Assigns the "Joint" object which will connect the two "Frame" objects. Sets an initial position of the module in case position is requested before simulation time step.	void
Update	Calls the update function of all "Frame" objects in the module. Calls the update function of the "Joint" object in the module. Updates the position of the module based on the "Frame" object positions.	void
Initialize	Calls the initialization function of all "Frame" objects in the module. Updates the position of the module based on the "Frame" object positions. Specifies the "Joint" object of the module to be a hinge and connects the two "Frame" objects.	void

Table 13: Module class functions

4.1.3.1.4. Frame

The “Frame” class contains information about the frames of the robot, where two separate frames and a joint represent a module. The frames are created by uploading a specific mesh for each of the two frames in a module, together with the other attributes. The size of the meshes can be scaled by the “scale” attribute. The vertices, uvs and triangles stored as attributes are used by both the “AgX_Frame” class and the visualization class, maintaining the component-based separation which makes it possible to create frames either outside of the physics simulation, or outside of visualization environments.

Contents of the “Frame” class are displayed in Table 14, while the main functions of the class are described in Table 15.

Attributes		Components
guid : Guid shape, materialName : string scale, mass : double isStatic : bool position, rotation : Vector3	quatRotation : Quaternion meshVertices : Vector3[] meshUvs : Vector2[] meshTriangles : int[]	agxFrame : AgX_Frame
Functions		
Initialize() Update()	ScaleMesh() SetMesh()	GetQuatRot() QuatToRot()

Table 14: Contents of the Frame class

Function	Description	Return
Initialize	Calls the ScaleMesh function. Calls the QuatToRot function to get the Euler angles of the frame rotation (for checking if the frame is pitch or yaw-configured). Creates the “AgX_Frame” object with the attributes and mesh properties of this class.	void
Update	Updates the position and rotation of the object based on the corresponding values of the object in the Algoryx simulation instance.	void
ScaleMesh	Multiplies the mesh vertices with the “scale” attribute of this class to scale the mesh accordingly.	void
SetMesh	Assigns the mesh attributes to this class from the function parameters.	void
QuatToRot	Updates the Euler angle representation of the sensor module’s rotation. Returns the Euler angle representation of the sensor module’s rotation.	Vector3

Table 15: Frame class functions

4.1.3.1.5. Joint

The “Joint” class contains information about the joints of the robot, both locks and hinges. A “Joint” object can be attached between two frames, a frame, and a sensor module, and between sensor modules and connected sensors. The “Joint” object is also able to modify the angle between the two objects it is attached to. This class is one of the few classes that are not visualized, as the graphic would be mostly ignored. However, interpolating between two attached objects will give the joint position, if desired for visualizations or analytics. The function controlling the angle of the joint is interfaced through the “Dynamics” class.

Contents of the “Joint” class are displayed in Table 16, while the main functions of the class are described in Table 17.

Attributes		Components	
guid, leftFrameGuid, rightFrameGuid : Guid type : string lowerRangeLimit, upperRangeLimit : double Kp, max_vel : double		agxJoint : AgX_Joint left, right : Frame	
Functions			
Create_Hinge()	Create_SensorModuleLock()	SetAngle()	
Create_Lock()	Create_ForceSensorLock()	Stabilize_Angle()	
	Create_DistanceSensorLock()	GetAngle()	

Table 16: Contents of the Joint class

Function	Description	Return
Create_Hinge	Creates the “AgX_Joint” object in this class as a hinge. Adds hinge to simulation.	void
Create_Lock	Creates the “AgX_Joint” object in this class as a lock. Adds lock to simulation.	void
Create_SensorModuleLock	Creates a lock between a “Frame” and a “SensorModule” object. Adds lock to simulation.	void
Create_ForceSensorLock	Creates a lock between a “ForceSensor” and a “SensorModule” object. Adds lock to simulation.	void
Create_DistanceSensorLock	Creates a lock between a “SensorModule” and a “DistanceSensor” object. Adds lock to simulation.	void
SetAngle	Adjusts the joint velocity to make the joint reach the desired angle. Controlled by a P-value.	void
StabilizeAngle	Slowly resets the angle of the joint to return to 0.	void
GetAngle	Retrieves the current angle of the joint.	double

Table 17: Joint class functions

4.1.3.1.6. SceneObject

The “SceneObject” class contains the information about the objects the user may place in the simulation environment. The objects can be used for either obstacles or objects to be interacted with, such as balls or moving blocks. They can also be used as a static flat ground for the robot in the scene to move on.

Contents of the “SceneObject” class are displayed in Table 18, while the main functions of the class are described in Table 19.

Attributes	Components
guid : Guid size, position, rotation : Vector3 quatRotation : Quaternion materialName, shape : string mass : double isStatic : bool	agxPrimitive : AgX_Primitive
Functions	
Initialize()	Update()

Table 18: Contents of the SceneObject class

Function	Description	Return
Initialize	Creates the “AgX_Primitive” object with the attributes of this class.	void
Update	Updates the position and rotation of the object based on the corresponding values of the object in the Algoryx simulation instance.	void

Table 19: SceneObject class functions

4.1.3.1.7. Scene

The “Scene” class mainly contains the terrain used in the simulation. The robot will move on this terrain, and the individual heights of the terrain are specified in a heightmap file from which the terrain is generated using a custom function for terrain creation. The Scene class stores vertices, triangles and uvs as attributes, which are later used both for the “AgX_Scene” objects, and visualization classes.

Contents of the “Scene” class are displayed in Table 20, while the main functions of the class are described in Table 21.

Attributes		Components
guid : Guid height_Image : string materialName : string height : double position : Vector3	vertices : List<Vector3> triangles : List<int> uvs : Vector2[]	scene : AgX_Scene
Functions		
Create()	CreateMesh()	LoadTerrainFromImage()

Table 20: Contents of the Scene class

Function	Description	Return
Create	Calls the LoadTerrainFromImage function Creates a new “AgX_Scene” object with the attributes from the “Scene” class.	void
CreateMesh	Only calls the LoadTerrainFromImage function. This is required when only visualizing the terrain, and not including the physics.	void
LoadTerrainFromImage	Converts the height_Image string to a byte array and turns it into a bitmap. Bitmap color values are retrieved for every pixel to decide height of the specific point. All pixels are iterated through while adding the correct vertices and triangles. Uvs are calculated from vertices.	void

Table 21: Scene class functions

4.1.3.2. AgX Interface

The “AgX_Interface” namespace contains all functions for interacting with the AgX library. Objects of these classes are used to represent the corresponding AgX objects and have no relation to or knowledge of the Simulation core. Functions in these classes make Algoryx interfacing intuitive for other classes like the Simulation core or code created in custom projects. Several classes contain functions for retrieving the “agx.RigidBody” objects to enable more customization of a simulator if a developer wishes to implement this framework.

Figure 16 explains the “AgX_Interface” classes’ role and the task of handling data from the objects in the physics engine of Algoryx.

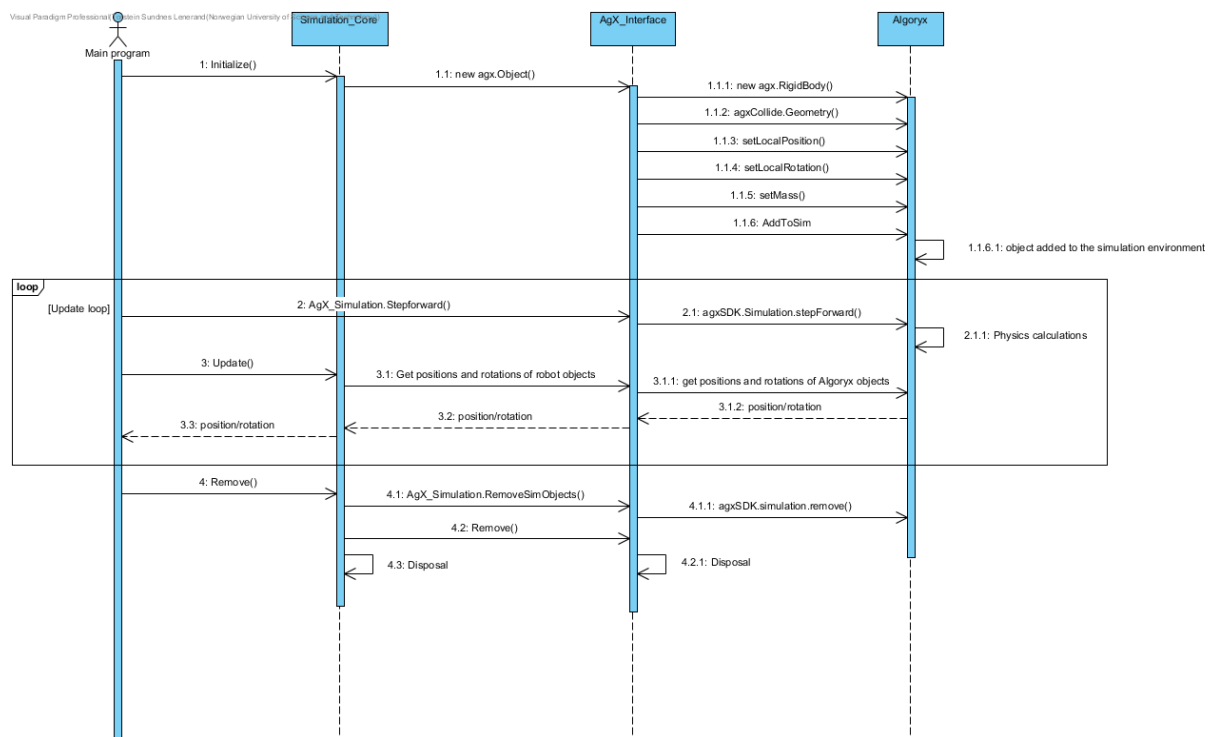


Figure 16: Sequence of Algoryx object handling

An Algoryx object is created when the corresponding “AgX_Interface” object is initialized. The “AgX_Interface” objects always have direct references to the Algoryx objects in the simulation environment. Thus, after the “agx.RigidBody” is created, it is modified with geometries, mass properties, etc. and added to the Agx simulation instance.

When the update loop of the simulator is called, the Agx simulation timestep is increased by the delta time each the update is called. All objects in the simulation are updated, and for the “Simulation_Core” objects to receive these values, they must call their corresponding “Update” functions which call “AgX_Interface” functions, which retrieve the attribute values from the “agx.RigidBody” objects.

Finally, when the application is closing, or a new robot is to be created, the removal functions are called, which first remove the rigid bodies from the simulation instance, and then ensures the “AgX_Interface” objects are also removed. After this, the “Simulation_Core” objects are cleared.

4.1.3.2.1. AgX_Assembly

The “AgX_Assembly” class is a container for the assembly class of Algoryx, which allows for the logical grouping of objects inside such an assembly. As such, the assembly is used for storing all the robot parts, such as the joints, frames, sensor modules and sensors.

Contents of the “AgX_Assembly” class are displayed in Table 22, while the main functions of the class are described in Table 23.

Attributes		Components	
none		robotAssembly : agxSDK.Assembly	
Functions			
AddToAssembly()	SetPosition()	GetPosition()	
AddToSim()	SetRotation()	GetRotation()	
RemoveFromSim()			

Table 22: Contents of the AgX_Assembly class:

Function	Description	Return
AddToAssembly	Adds a rigid-body or a joint to the assembly.	void
AddToSim	Adds the assembly to the Algoryx simulation instance.	void
RemoveFromSim	Removes the assembly from the Algoryx simulation instance.	void
SetPosition	Modifies the global position of the assembly.	void
SetRotation	Modifies the global quaternion rotation of the assembly.	void
GetPosition	Retrieves the global position of the assembly.	Vector3
GetRotation	Retrieves the global quaternion rotation of the assembly	Quaternion

Table 23: AgX_Assembly class functions

4.1.3.2.2. AgX_Frame

The “AgX_Frame” class contains the AgX objects that are created by mesh and constitute one of the two frames that make up a robot module. The class is dynamic and allows for multiple shapes and sizes of the required meshes. The center vector should remain the same for each frame that is created, as it is the center where the joint is located.

Contents of the “AgX_Frame” class are displayed in Table 24, while the main functions of the class are described in Table 25.

Attributes	Components	
guid : Guid shape : string materialName : string size : double	agx_Object : agx.RigidBody	
Functions		
AgX_Frame() Remove()	GetAgxObject() AddToSim()	GetPosition() GetRotation() GetQuatRotation()

Table 24: Contents of the AgX_Frame class

Function	Description	Return
AgX_Frame	Creates the physics version of the mesh received via vertices, uvs, triangles and other attribute parameters. Adds the frame to the robot assembly.	AgX_Frame
AddToAssembly	Adds the frame to the robot assembly.	void
Remove	Removes the frame from the simulation instance.	void
GetAgxObject	Returns the Algoryx rigid-body object.	agx.RigidBody
GetPosition	Returns the position of the object in the simulation	Vector3
GetRotation	Returns the rotation of the object in the simulation (radians).	agx.Vec3
GetQuatRotation	Returns the rotational matrix of the object in the simulation (Quaternion).	Quaternion

Table 25: AgX_Frame class functions

4.1.3.2.3. AgX_Joint

The “AgX_Joint” class contains AgX joint objects, and receives information from the C# simulation core, which defines each individual joint. It will also return values to the joint class in the simulation core regarding position, forces, and other parameters relating to the joint.

The joint is the connection between each robot module and sensor module and contain motor controllers which controls the movement of each joint. Since both lock-joints and hinge-joints come from the same class, agx.Constraints, the “AgX_Joint” class can contain them both.

Contents of the “AgX_Joint” class are displayed in Table 26, while the main functions of the class are described in Table 27.

Attributes	Components
guid : Guid type : string	joint : agx.Constraint hinge_Frame : agx.HingeFrame

Functions		
AgX_Joint()	Create_Hinge()	GetForce()
AddToAssembly()	Create_Lock()	GetAngle()
Remove()	ForceSensorLock()	SetSpeed()
	DistanceSensorLock()	

Table 26: Contents of the AgX_Joint class

Function	Description	Return
AgX_Joint	Adds the specified GUID to the object.	AgX_Joint
AddToAssembly	Adds the joint to the robot assembly.	void
Remove	Removes the joint from the simulation instance.	void
Create_Hinge	Creates a hinge with the given input variables. Assigns it to the agx.Constraint of the class. Locks two frames together.	void
Create_Lock	Creates a lock with the given input variables. Assigns it to the agx.Constraint of the class. Locks two frames or a frame and a primitive together.	void
ForceSensorLock	Creates a lock with the given input variables Assigns it to the agx.Constraint of the class. Locks a force sensor and a primitive together.	void
DistanceSensorLock	Creates a lock with the given input variables Assigns it to the agx.Constraint of the class. Locks two primitives together.	void
GetForce	Returns the force exerted on the joint.	double
GetAngle	Returns the current angle of the joint.	double
SetSpeed	Sets the desired velocity of the joint.	void

Table 27: AgX_Joint class functions

4.1.3.2.4. AgX_Primitive

The “AgX_Primitive” class allows for the creation of a primitive-shaped object such as spheres and cubes, to be placed into the simulation environment, or attached to the robot assembly. Specific function parameters determine how the object is attached (robot or scene). These objects may be static or dynamic.

Contents of the “AgX_Primitive” class are displayed in Table 28, while the main functions of the class are described in Table 29.

Attributes	Components	
guid : Guid shape, materialName : string size : double	agx_Object : agx.RigidBody	
Functions		
AgX_Primitive()	AddToAssembly()	GetPosition()
GetAgxObject()	Remove()	GetRotation()

Table 28: Contents of the AgX_Primitive class

Function	Description	Return
AgX_Primitive	Creates the Algoryx rigid-body with shape, position, rotation, size, mass, material, static/dynamic, and part of robot/standalone values for the input variables. Adds the object to the simulation instance if the AddToRobot variable is false, and adds to the robot assembly if true	AgX_Primitive
GetAgxObject	Retrieves the Algoryx object	agx.RigidBody
GetPosition	Retrieves the position of the object in the simulation	Vector3
GetRotation	Retrieves the Quaternion rotation of the object in the simulation	Quaternion
AddToAssembly	Adds the rigid-body to the current robot-assembly (if necessary)	void
Remove	Removes the rigid-body from the simulation instance.	void

Table 29: AgX_Primitive class functions

4.1.3.2.5. AgX_Scene

The “AgX_Scene” class contains the terrain information for the current scene. In the scene class, a function transforms the vertices, triangles and indices received from the “Scene” class and creates the physical terrain for the simulation. In addition, material name and position is adjusted by the function parameters.

There is no function for modifying the height of the terrain independently, as this is performed during the creation of the terrain variables in the “Scene” class and requires the whole process to be restarted.

Contents of the “AgX_Scene” class are displayed in Table 30, while the main functions of the class are described in Table 31.

Attributes	Components
guid : Guid	terrain : agx.RigidBody
Functions	
AgX_Scene() Remove()	

Table 30: Content of AgX_Scene class

Function	Description	Return
AgX_Scene	Receives several parameters for terrain creation, and constructs an Algoryx geometry which is used as the terrain. The created terrain rigid-body is added to the simulation instance.	AgX_Scene
Remove	Removes the terrain's rigid-body from the Algoryx simulation instance.	void

Table 31: AgX_Scene class functions

4.1.3.2.6. AgX_Simulation

“AgX_Simulation” is a static class which provides an intuitive interface to the algoryx simulation instance. By creating custom start and stop functions, the designer of programs using this framework have an easy tool for starting the physics environment, and safely disposing of it when necessary. All operations regarding the Algoryx simulation instance will be performed through this class.

Contents of the “AgX_Simulation” class are displayed in Table 32, while the main functions of the class are described in Table 33.

Attributes		Components	
none		sim_Instance : agxSDK.Simulation	
Functions			
Start()	StepForward()	AddContactMaterial()	
Stop()	RemoveSimObjects()		

Table 32: Content of AgX_Simulation class

Function	Description	Return
Start	Initializes the Algoryx C# environment (by invoking the agx.agxSWIG.init() function). Starts the simulation instance (by initializing a new agxSDK.Simulation instance). Sets gravity and time step values.	void
Stop	Removes all objects from the simulation. Clears the simulation instance. Shuts down the agxSWIG instance.	void
StepForward	Moves time forward in the simulation by accessing the “stepForward” function of the simulation instance.	void
RemoveSimObjects	Removes all the objects in the Algoryx simulation instance (by calling simulation.removeAllObjects()).	void
AddContactMaterial	Adds the contact friction between two input materials, defined by restitution, friction and Youngs modulus between the two materials. Adds the materials and the contact material info to the simulation instance.	void

Table 33: AgX_Simulation class functions

4.1.3.3. Unity_Visualization

This section describes the proposed (and currently used) visualization class for visualizing objects in the simulation core. As the visualization is a secondary concern in the project, the documentation will not be as in depth, but rather provide an example to the possibilities and modifiability that is included in the framework of this system. The “Unity_Visualization” objects are in no way referenced in the “Simulation_Core” objects, rather using the GUID system to identify components to visualize. Each class contains a certain method of visualizing a component, allowing the visualization class to display the terrain, frames, scene objects, sensor modules and sensors using the information stored in the “Simulation_core” classes. Table 34 shows how the individual visualization classes are structured, and Figure 17 shows the sequence of operations for a “Unity_Visualization” class object.

Scene_Vis	
Attributes	Components
guid : Guid mesh : UnityEngine.Mesh	terrain : UnityEngine.GameObject
Functions	
Scene_Vis()	
Frame_Vis/Primitive_Vis()	
Attributes	Components
guid : Guid	gameobject : UnityEngine.GameObject
Functions	
Frame_Vis()/Primitive_Vis()	Update() Remove()

Table 34: Main content of the Unity_Visualization classes

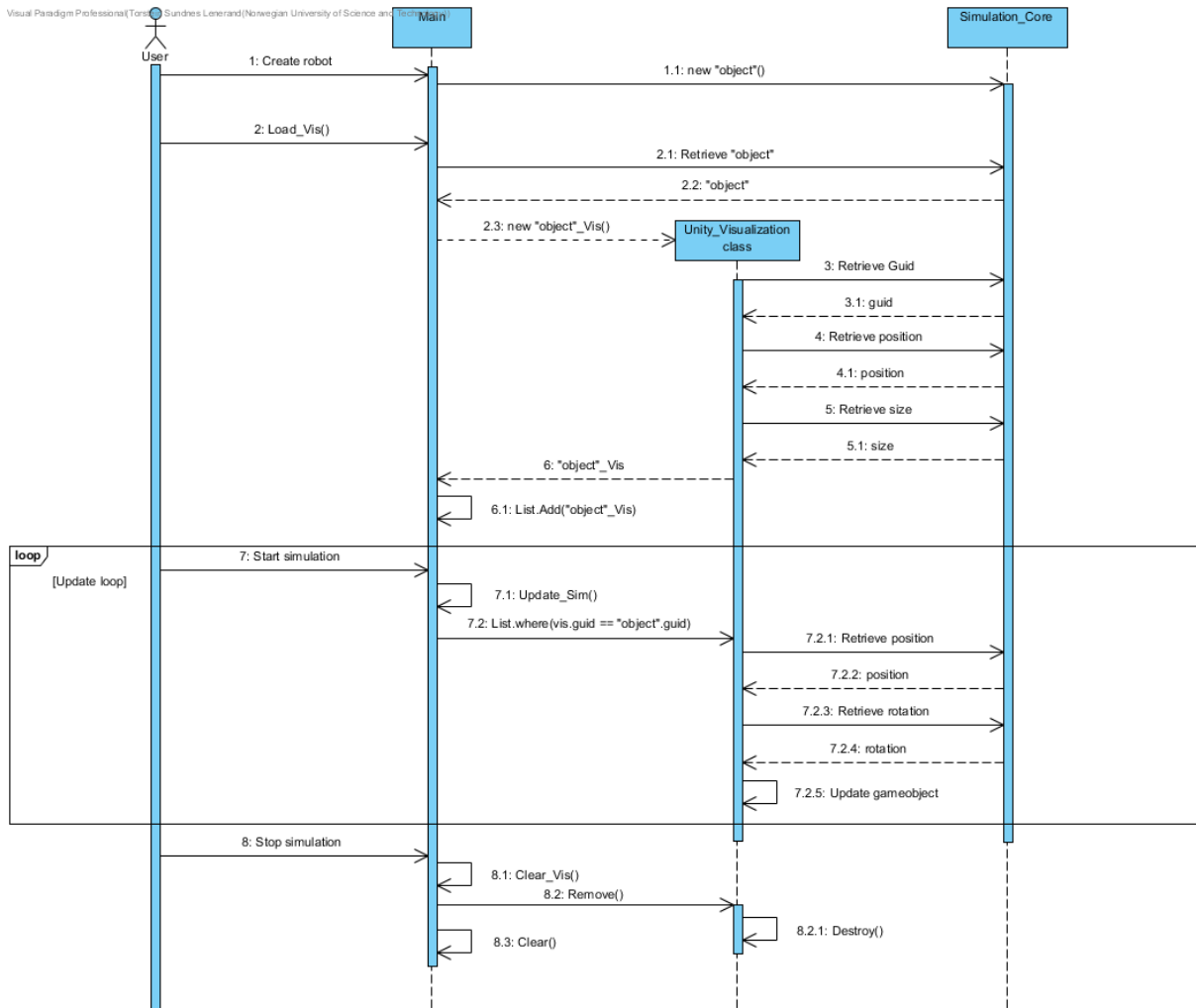


Figure 17: Unity_Visualization object lifecycle

A second component class for visualization (Analytics_Visualization) has been created as a demonstration of the simulator's visualization capabilities, which logs all essential information about the robot; such as positions, angles of joints, and force/distance measurements. This class is also a standalone component, which is called from the "Main" class (preferably in an update-loop) and takes a "Robot" object as input and reads its data. This component is not described in the thesis.

4.1.3.4. Dynamics

The “Dynamics” component class is used to modify the joint angles of a “Robot” object, producing different configurations of motion for the modular robot. There are several parameters relating to the phase, amplitude and offset of the angles of modules, in addition to movement variables which are used as direction controllers. These direction controllers assume values between “-1” and “1”, allowing input keys to change the direction of motion forward/backward, and left/right from the “Main” class. This is in addition to the option of changing the main parameters depicted in Table 35 under “Input parameters”. The values of the main parameters are added to their corresponding arrays which aid in calculating the angles of each joint, with the phase difference arrays being modified based on the phase offsets, giving a certain offset to the movement of each joint of the robot. The “Dynamics” functions should always be updated before the simulation timestep is advanced.

The desired angle of each joint is calculated as shown in EQ(3):

$$Angle[i] = Amplitude[i] \cdot \sin\left(\frac{2\pi t}{period[i] + phaseDiff[i]}\right) + offset[i]. \quad (3)$$

All “Dynamics” movement parameters are arrays (even though it is only necessary for the phase difference) to separate pitch and yaw configurations, in addition to enabling custom parameters for each individual joint. When overriding the class, only the “Initialize” function needs to be modified, as this is the one preparing parameter arrays.

Attributes						
angles : double[]		f_movementVars : double[7]				
amplitudes : double[]		t_movementVars : double[7]				
period : double[]		currentAction : string				
phaseDiff : double[]		nextAction : string				
offset : double[]						
Functions						
Initialize()		Control()			Forward()	
All_Movement()						
Input parameters						
Amplitudes pitch	Amplitudes yaw	phaseOffset pitch	phaseOffset yaw	period	Offset pitch	Offset yaw

Table 35: Main content of the Dynamics class

The “Forward” and “All_Movement” functions perform the angle calculations based on the movement parameters, using the equation shown in EQ(3). The forward and turn movement variables are pre-made for basic robot movement. The input variable “dyn_vars” contains the movement parameters given by the user. The current and next “action” variables define whether

a new movement pattern, such as forward or turn, has been selected. The corresponding initialization functions will be activated accordingly.

4.1.3.5. Optimization

The “Optimization” component class uses a Genetic Algorithm to improve the movement of a robot over time. The class uses a modified version of the “Dynamics” class to store the variables to optimize, called “Opti_Dynamics”. The genome featured in the algorithm is the movement parameters, as described in the “Dynamics” section (4.1.3.4). The functions of GA operations perform uniform crossovers and random mutations to better locate both global and local optimums. The fitness function is based on the Euclidean position of the robot. There are also upper and lower limits for the values that are to be optimized, so the robot will not behave in a manner that is disadvantageous though effective. Table 36 shows the main contents of the “Optimization” class:

Attributes		
started : bool		dynamics_List : List<Opti_Dynamics>
population : int		originalGenome : double[7]
quickOpti : bool		UpperLimit : double[7]
IterTime : int		LowerLimit : double[7]
currentGeneration : int		toggledForOptimization : bool[7]
Functions		
Load()	UpdatePopulation()	GetRandomNumber()
Reset()	UniformCrossover()	
Update()	Mutate()	

Table 36: Main content of the Optimization class

The optimization may be performed both while showing the visualization of the robot, and in a closed loop only performing the optimization within the physics engine allowing the user to step to a certain generation of species. The user may also customize the dynamics values that are to be optimized, reducing optimization time by excluding parameters not in use with the toggle array. These choices must be specified by the user and are in this project selected in the “Main” class.

4.1.3.6. Scene Designer

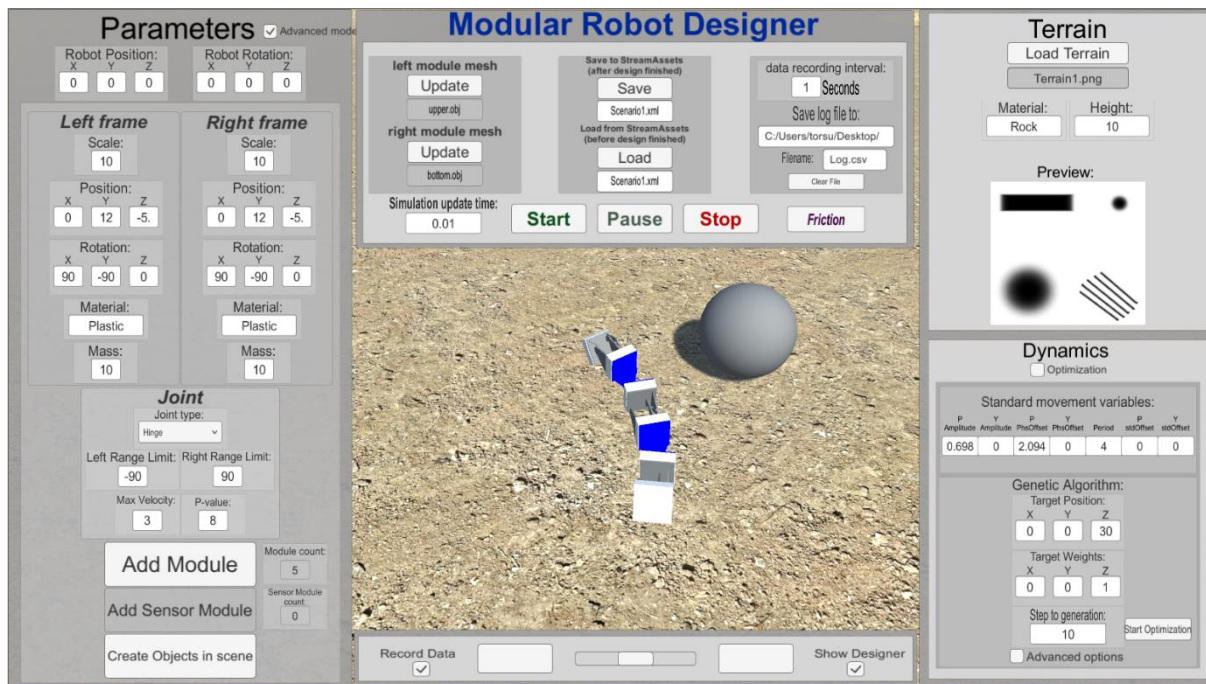


Figure 18: Scene designer interface

The “Scene Designer” is referred to as the GUI tool and its corresponding software, created for realizing the functionality of the simulator. The scene designer is not developed with regards to CBD, as the increased development time would possibly be problematic and that the designer is platform dependent (Unity visualizations). It has been designed to enable creation of scenarios in both an advanced manner with all parameters of the robot modifiable, and with an easy mode where the user may simply select if the next component is a pitch-, yaw-, or sensory-module, or if it’s a scene object. The details about the assembled robot are serialized to the XML file, following an initialization call to the “Main” class when the robot design is finalized, making the “Main” class take over the functionality of the program, and running the simulation. The “Scene Designer” GUI still appears on top of the simulation after it has started, to enable pausing/stopping, data recording, optimization value changes, etc. The controls on the bottom enable rotation and zooming of the simulation, while the toggles control the recording of data, and hiding the designer to show more of the scene.

Figure 19 shows the most important functionality of the Scene Designer.

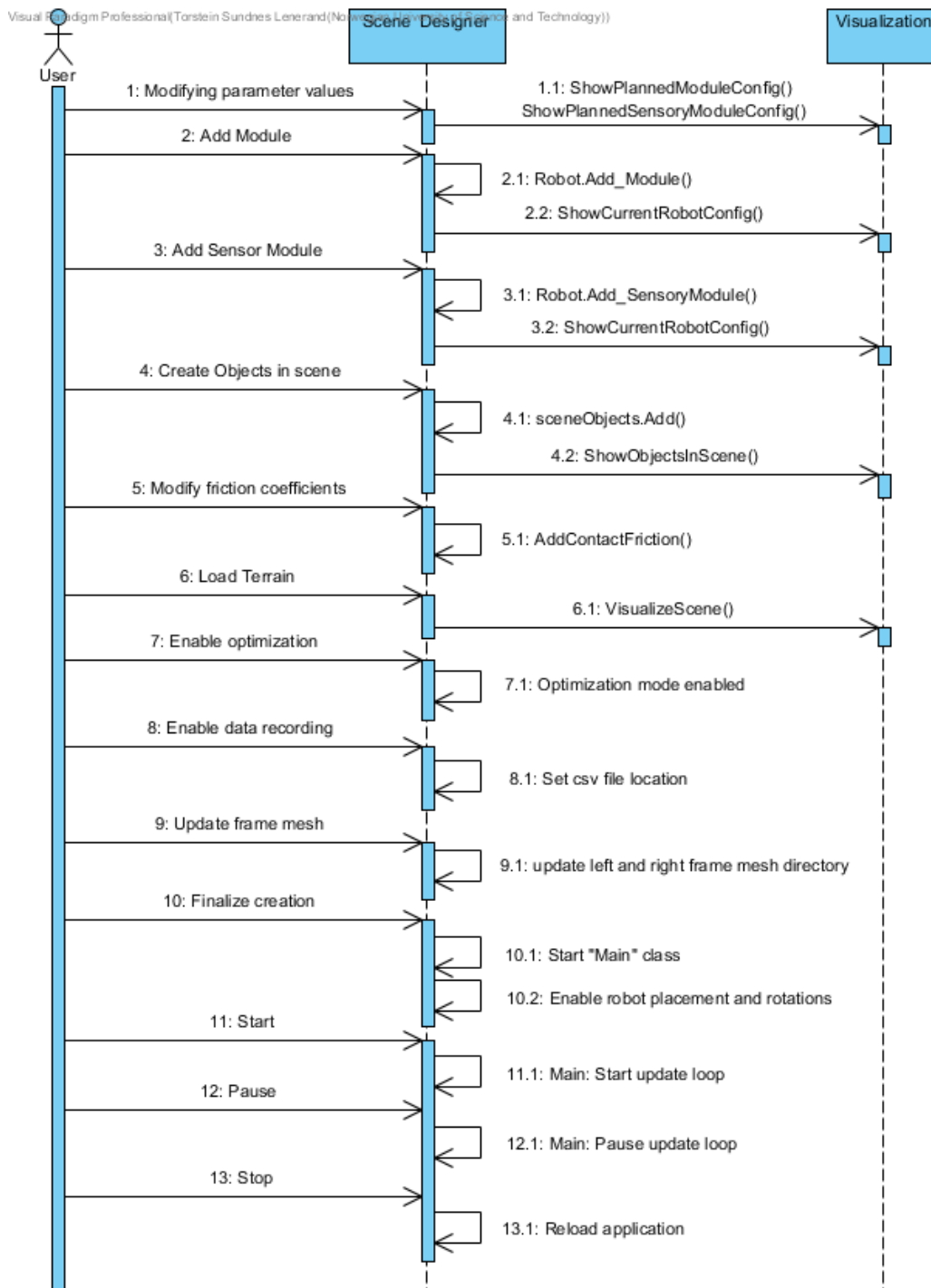


Figure 19: The Scene Designer's most important features

4.1.4. Core Framework usage (Robot design)

This chapter explains how the robot and the scenario is created, the simulation steps, and the components of a robot.

4.1.4.1. Robot assembly

The robot assembly, as shown in Figure 20, is created by several “Module” objects and “SensorModule” objects, with the amount depending on the choices of the designer. Each “Module” object always contains two “Frame” objects and one “Joint”. Each “SensorModule” may contain 1 “ForceSensor” object, or up to 6 “DistanceSensor” objects. In addition, there is a list of “Joint” objects in the robot, which are used to lock the modules and sensor modules together, and in the sensor module, to connect the sensors. Virtually infinite combinations may be created because of the module/component-based assembly structure.

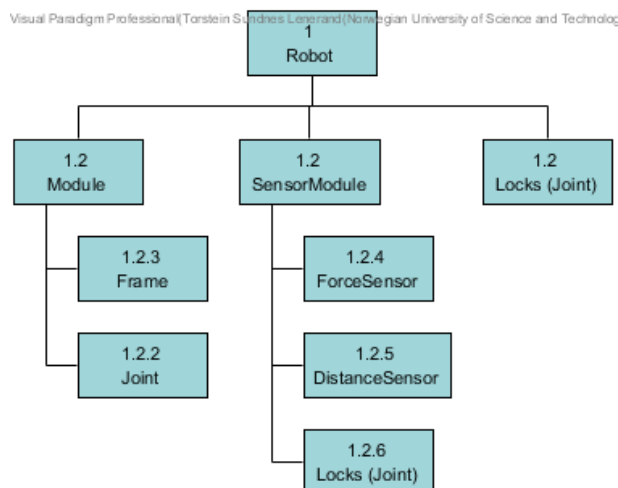


Figure 20: Robot assembly structure

4.1.4.2. Object creation

Figure 21 shows the sequences of operations on the robot's components in the simulator. General functions in the architecture mostly have the same function such as “Initialize” and “Update”, and all work in the same manner with relation to object creation and destruction.

The diagram shows the sequence in which objects are created as “Simulator_Core” objects, following the creation of their corresponding “AgX_Interface” objects, the update loop running, and finally the destruction of the object.

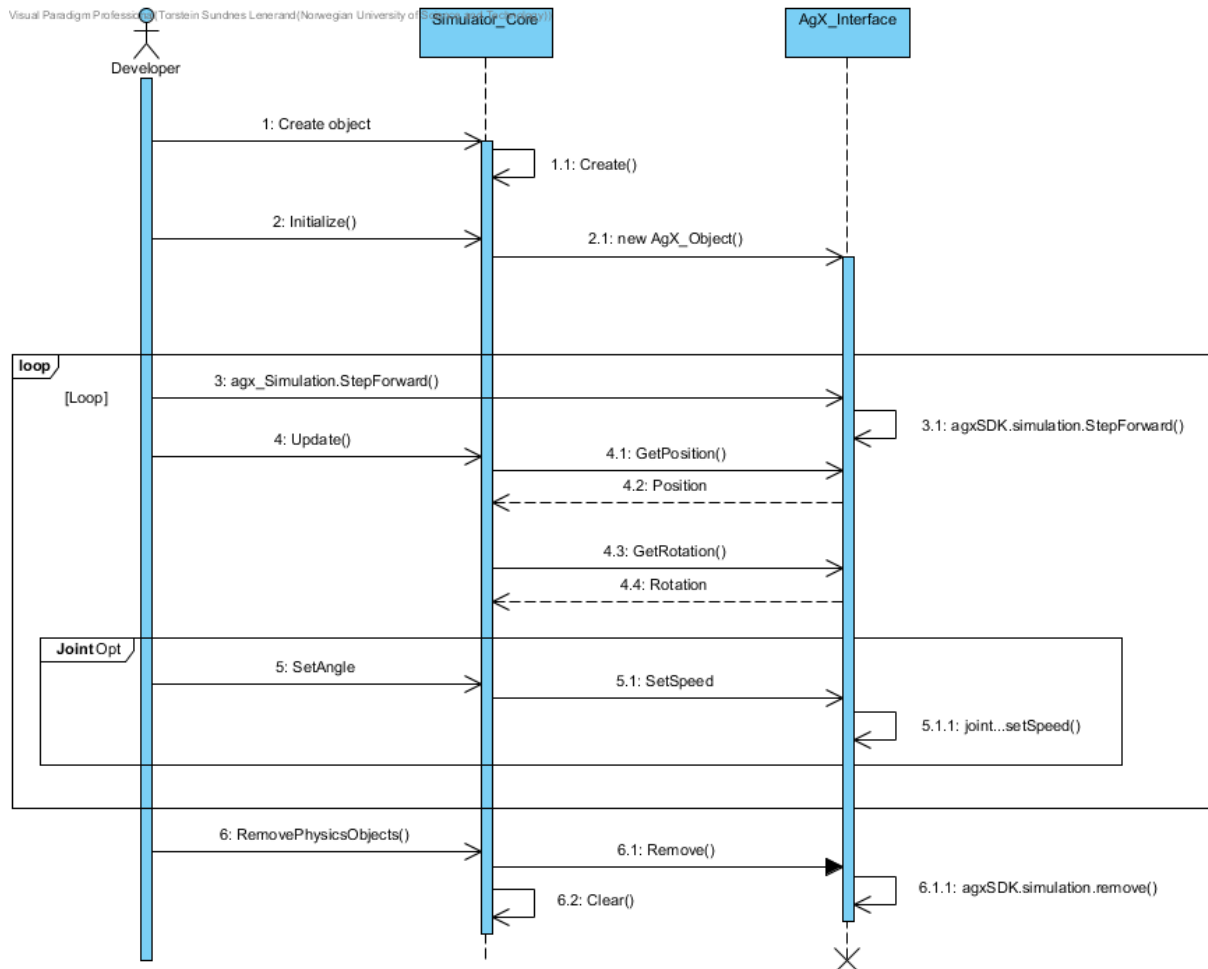


Figure 21: Creation and destruction of simulation objects

1.Create object

An object of a certain class is created, using a frame object as an example. The public variables of the frame are assigned, such as the unique GUID of the frame, scale, mass, and material. Some classes, such as the “Module” class, also have a “Create” function which performs more complex variable assignments, such as adding the correct frames and joints to the specific module. When all objects have been created, they can be serialized to an xml file.

2.Initialize

The “Initialize” functions create the “AgX_Interface” objects with the attributes in the corresponding “Simulation_Core” objects. In certain classes, such as the “Robot” and “Module” class, the initialization function calls the initialization functions for all its sub-components. In the “Robot” class, the initialization function also adds the robot to the simulation instance.

3.StepForward

The “StepForward” call goes straight to the static class “AgX_Simulation”, with no reference in the “Simulation_Core” namespace. There is no need for an extra wrapper for this class, as it is static and solely used in the Algoryx domain. This function increases the step of the Algoryx simulation instance by the delta time set in a startup call.

4.Update

The “Update” function is located in the “Robot” class, and updates all the positions, rotations, and sizes within the robot assembly, automatically. It is performed to get the “Simulation_Core” objects up to date with their corresponding objects residing in the Algoryx simulation instance.

5.SetAngle

Each “Joint” object has a “SetAngle” function, which takes an angle as input. The desired angle is sent to a P-regulator, which increases or decreases the speed of the joint motor for the error between requested and actual angle to become as close to zero as possible. This function is called from the “Dynamics” class which governs the movement of the robot. The function can also be called from the “Optimization” class.

6.Remove

The “RemovePhysicsObjects” class exists in the “Robot” class and removes all Algoryx objects in the “AgX_Interface” classes. Then, all the Algoryx interface objects are set to null, and the lists containing the sub-components of the “Robot” class are cleared.

4.1.4.3. XML functionality

The XML file contains the object representation of the scenario created. The “Scene_Designer” class serializes the created scenario, while the “Main” class deserializes it to create objects of the robots (all based on the Core Framework). Figure 22 shows the hierarchy of the XML tags, representing the class objects of the scenario. An example configuration of the XML file is shown in Appendix F.

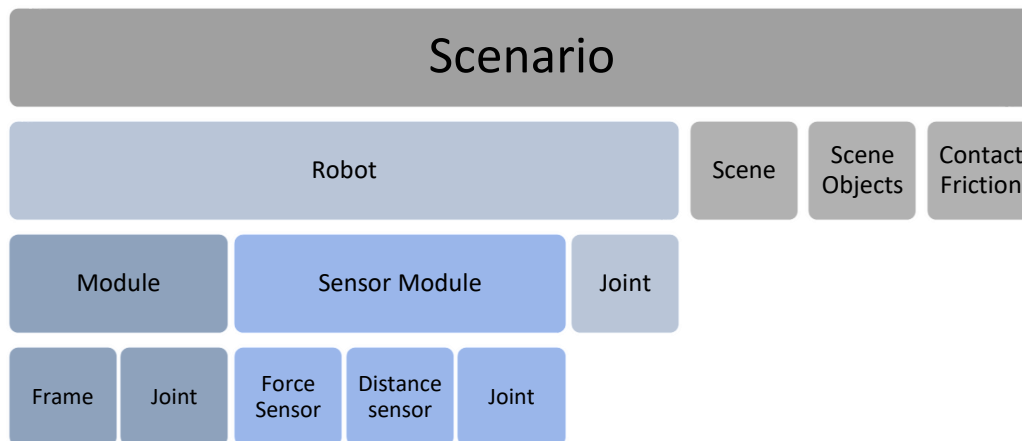


Figure 22: XML file scenario class representation

4.1.4.3.1. Scenario file structure and deserialization

The XML file is deserialized into an object containing the required parameters for the simulation. Two frames and one joint object are contained within a module object, and multiple module (or sensory module) objects together with corresponding joints are contained within the robot.

When the XML file is deserialized, objects in the file are created of the “Scenario” class. This class contains one instance of a robot assembly class, an instance of the scene, a list of contact frictions and a list of scene objects. It is deserialized into objects of all the classes within the “Simulation Core”, such as “Robot”, “Modules”, “Frames”, “Scene”, “ContactFriction”, “SceneObject”, etc.

When objects within the robot assembly are deserialized, they are put into their corresponding “parent” component, such as frames and joints put together to modules.

The “Scene” object is deserialized and initialized to create the AgX object which makes the terrain. The terrain height data is represented as a string in the XML file. This string is assigned to the Scene object and sent to AgX for terrain creation. The visualization class will use the same string to create the corresponding mesh.

Contact friction is calculated between all the objects specified in the “ContactFriction” list in the scenario object, and all scene objects from the “SceneObject” list are placed in the simulation.

4.2. Verification and validation

Full Requirement and Test&Verification Specifications are found in Appendix B and Appendix C.

The tests were performed mainly in the build application, and some in the editor (if code modification was necessary). Most tests passed, with a few exceptions which ended in “failed”, partly verified or Not Applicable:

- The B-level requirement regarding loading a scenario which has been saved mid-execution did not pass, due to a setting making robot construction easier (read more in T-1.1.2 comment section).
- Some tests (T-1.2.1, T-4.1.1) have been marked as partly verified because of the Algoryx library issues.
- Some tests have been marked as Not Applicable (N/A), due to functionality which has intentionally or due to time limits not been implemented.

The reviews were performed by going over the code to verify performance or functions. Some reviews were also performed by analysis of component functionality. By review, there is a higher standard for verification, so if the reviews uncover minor discrepancies between pass criteria and actual performance, the result will at most be partly verified.

Use-cases were performed along with Dr. Guoyuan Li, at NTNU Ålesund. The MRSim was compared to the current development methods for modular robots at NTNU, and general usefulness for teaching and research was distinctly reviewed.

4.3. Case studies

4.3.1. Creating and running a scenario

This is a test of the Modular Robot Simulator, demonstrating its functionality. A robot is created with 5 modules and 6 sensor modules which all have force sensors under them, weighing 1 gram. All components are made from plastic. The first sensor module has a distance sensor in the front. The mass of each module and sensor module is set to 100 grams.

The XML file created for this scenario is shown in Appendix F.

Lastly, the simulation is started, and the robot will move to push a scene object. Everything will be logged to a file for retrieving the values of sensors and modules.

4.3.1.1. Design

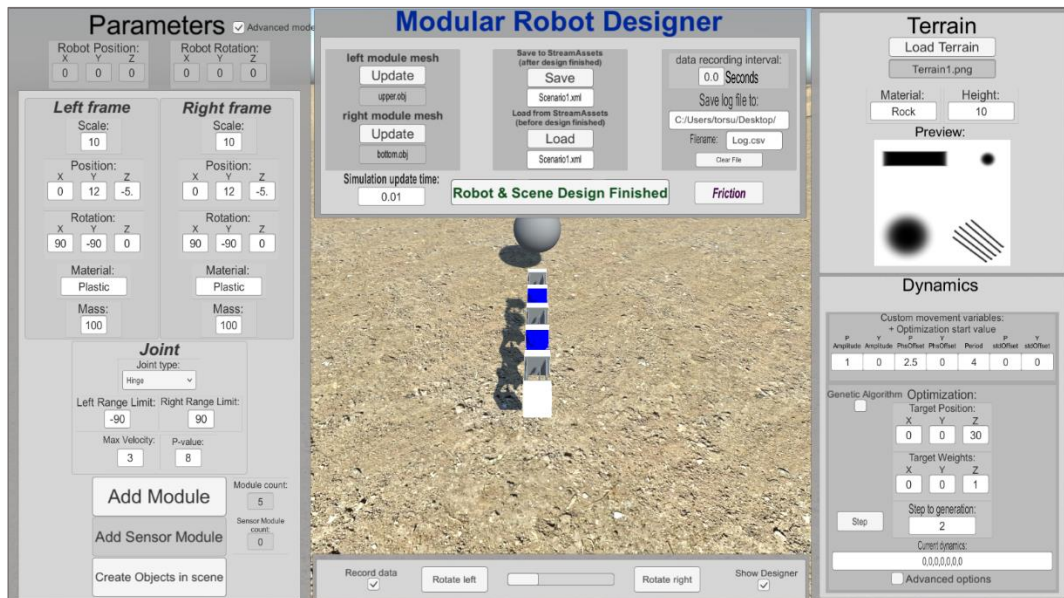


Figure 23: Case study - design overview

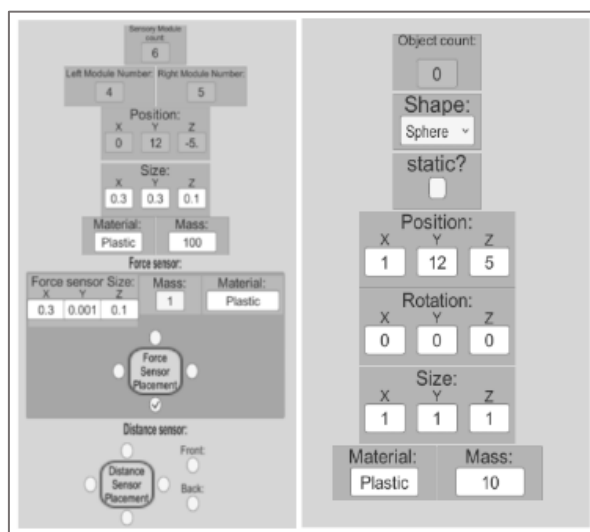


Figure 24: Case study - Sensor module and scene object design

4.3.1.2. Result

Visualization of scene

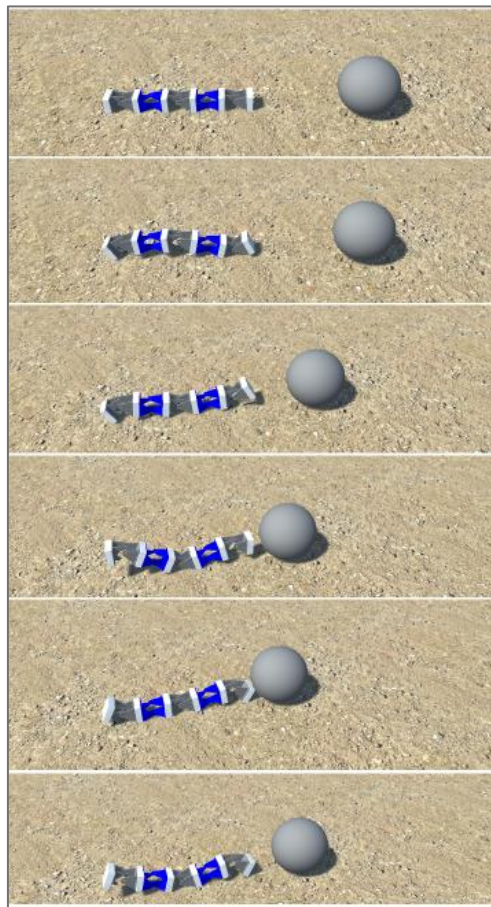


Figure 25: Robot moving forward to push the ball

Figure 25 shows the robot going from a neutral position, to moving forward to push the scene object, in this case modelled as a ball.

Analytics log

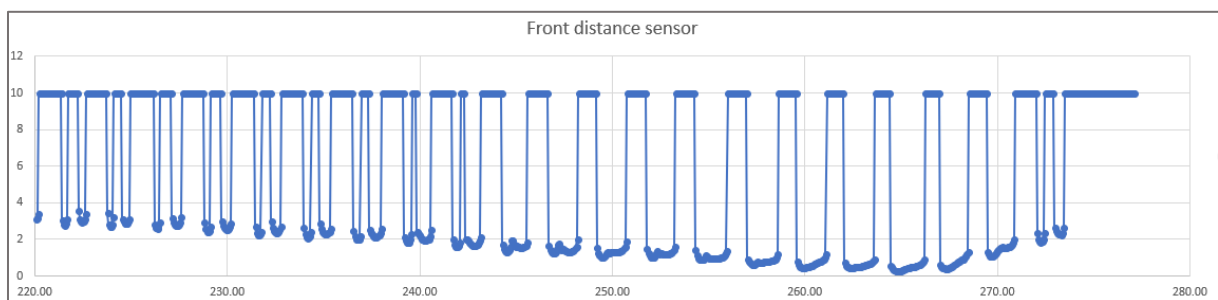


Figure 26: Distance sensor measurements

The chart in Figure 26 shows how the robot gets closer to the ball, before bumping into it, causing the ball to roll away from the robot. The top-values {10} indicate the maximum distance for the distance sensor measurements. Top-values occur when no objects intersect the distance ray. The x-values correspond to the time in seconds since the application started.

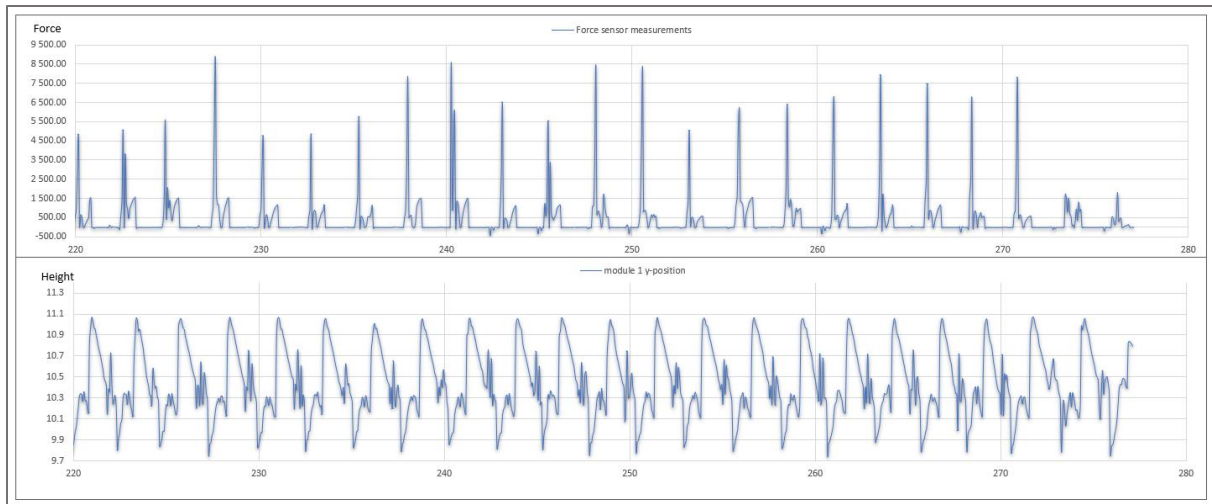


Figure 27: Force sensor vs y-position measurements

Figure 27 shows how the force exerted on the front sensor module correlates to the y-position of the front module. When the y-position is the lowest, the module is touching the ground, making the force sensor register the increased force of the sensor module pushing down.

4.3.2. Dynamics test

A robot with 11 modules is created. The dynamics are set to a normal forward motion, then a more extreme forward motion, then a wide turn, followed by a sharp turn. Additional dynamics tests are shown in Appendix G.

Forward

P	Y	P	Y	Period	P	Y
Amplitude	Amplitude	PhsOffset	PhsOffset		stdOffset	stdOffset
1	0	2.5	0	4	0	0



Figure 28: Robot forward motion

Forward with increased amplitude

P	Y	P	Y	Period	P	Y
Amplitude	Amplitude	PhsOffset	PhsOffset		stdOffset	stdOffset
2	0	2.5	0	4	0	0



Figure 29: Robot forward motion with increased amplitude

Wide turn

P	Y	P	Y	Period	P	Y
Amplitude	Amplitude	PhsOffset	PhsOffset		stdOffset	stdOffset
0.5	0	2.5	0	4	0	0.3



Figure 30: Robot wide turn

Sharp turn

P	Y	P	Y	Period	P	Y
Amplitude	Amplitude	PhsOffset	PhsOffset		stdOffset	stdOffset
0.5	0	2.5	0	4	0	0.8

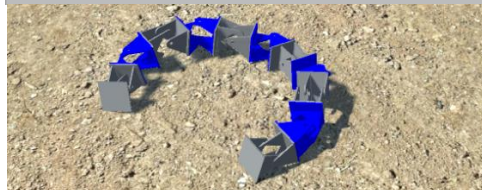


Figure 31: Robot sharp turn

4.3.3. Framework case-study

The Core Framework was built into two C# library files (dll), “Simulation_Core.dll” and “AgX_Interface.dll”. Along with the “AgxDotNet.dll”, they were put into a clear Unity project, for creating a simple simulation of two frames and a joint, assembled to a module, residing in a robot and placed on a scene object. The angle of the joint was set to a constant value, with the max velocity of the joint limited to $\frac{\pi}{6}$ degrees per second, to demonstrate that the joint movement is functional. Table 37 shows how one would go about creating a simple simulation with the framework and can be expanded to include more modules/objects, shown in the robot function that is commented out.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Xml.Serialization;
using AgX_Interface;
using Simulation_Core;
using System;

public class startSim : MonoBehaviour {

    Robot robot = new Robot();
    SceneObject sceneobj;
    void Start()
    {
        //Start simulation:
        Agx_Simulation.Start(0.01);

        //Setting position and rotations:
        var pos = new AgX_Interface.Vector3(0, 0, 0);
        var u_quat = UnityEngine.Quaternion.Euler(0, 90, 0);
        AgX_Interface.Quaternion frame_rot = new AgX_Interface.Quaternion(u_quat.x, u_quat.y, u_quat.z, u_quat.w);

        //Create Frames:
        Frame[] frames = new Frame[2];
        for (int i = 0; i < 2; i++)
        {
            frames[i] = new Frame()
            {
                guid = Guid.NewGuid(),
                position = pos,
                scale = 10,
                quatRotation = frame_rot,
                //rotation = rot,
                mass = 10,
                isStatic = false,
                materialName = "Plastic"
            };
        }

        //download mesh obj file:
        loadmesh();

        //Set mesh of frames:
        frames[0].SetMesh(AgxHelper(leftmesh.vertices), AgxHelper(leftmesh.uv), leftmesh.triangles);
        frames[1].SetMesh(AgxHelper(rightmesh.vertices), AgxHelper(rightmesh.uv), rightmesh.triangles);

        //Create Joint:
        Simulation_Core.Joint joint = new Simulation_Core.Joint()
        {
            guid = Guid.NewGuid(),
            leftFrameGuid = frames[0].guid,
            rightFrameGuid = frames[1].guid,
            type = "Hinge",
            lowerRangeLimit = -Math.PI/2,
            upperRangeLimit = Math.PI/2,
            max_vel = Math.PI/6,
            Kp = 3
        };
    }
}
```

```
//Create Module:
var module = new Module();
module.Create(frames[0],joint,frames[1]);

//Add module to robot:
robot.Add_Module(module);

//Add second module to robot:
//robot.Add_Module(module2,new Simulation_Core.Joint());

//Initialize robot:
robot.Initialize();

//Scene object:
sceneobj = new SceneObject()
{
    guid = Guid.NewGuid(),
    shape = "Box",
    size = new AgX_Interface.Vector3(5,1,10),
    position = new AgX_Interface.Vector3(0,-2,0),
    quatRotation = new AgX_Interface.Quaternion(0,0,0,1),
    materialName = "Rock",
    mass = 10,
    isStatic = true
};
sceneobj.Initialize();

//Load vis from mesh and robot + scene object:
Load_Vis();

//Start sim update loop:
InvokeRepeating("Update_Sim", 0.01f, 0.01f);
}
```

Table 37: Framework implementation in a new project

Visualization and update functions can be found in Appendix F1. The “AgxHelper” functions convert “AgX_Interface” structures to “UnityEngine” structures.

Figure 32 shows the created simulation:

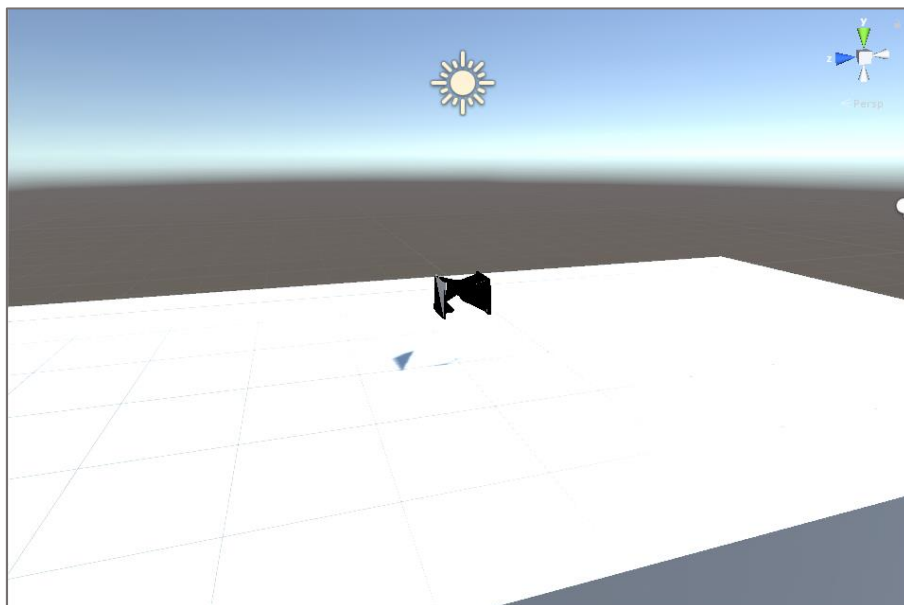


Figure 32: Custom project, scenario

Figure 33 shows the movement of the module over time:

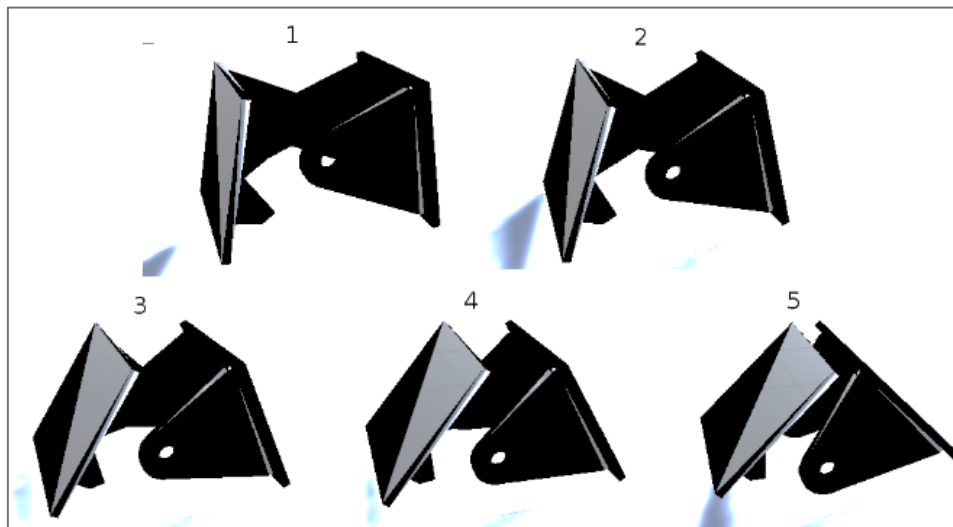


Figure 33: Custom project, module movement

The program took around 1 hour to design, with most of this time going to assigning the correct mesh. Each frame must have a mesh assigned to them, as well as to the Unity visualization objects. It is much easier than to begin from nothing, designing objects directly through the physics engine of both Unity or Algoryx, in addition to visualizing them.

5. Discussion

A modular robot simulator has been created for the NTNU research group focusing on modular robotics (1.2.1.5). It is based on the needs of the research group and improved by implementing CBD aspects originally intended for large scale projects, as partly demonstrated in previous projects (1.2.1.7), with visual inspiration from simulators like VSPARC (1.2.1.3). Though the system is designed for NTNU's design of robots, custom object models for the frames may be uploaded, specifically tailored to any type of chain-based modular robot.

Adopting the research questions in (1.2.1.9) have inspired several of the tests used to verify the impact of CBD in the project, and the concept of component assessment (1.2.1.8) have been the cornerstone for design and further improvements to modular robot simulators in the future.

Discussions regarding the specific tests are found in the Test and Verification Specification found in Appendix C

5.1. CBD decisions

Separating the simulation objects, simulation physics and simulation visualization has made the simulator more modular and portable, while at the same time enabling more effective optimization, by not showing visuals and only updating the robot variables for each timestep of the simulation. It is suspected to have delayed the development to some extent, but its benefits outweigh this development issue. Among these benefits are also the ability to use this framework in other projects, to modify individual components without disturbing the rest of the system, and the structure of the class system being easy to read and troubleshoot (because of the object-oriented component-based architecture).

The general impression that CBD is only viable in large-scale projects or businesses is contrasted in this project. Of course, this largely depends on the type of project, but for a master thesis with limited time it was expected that just creating a program would not be enough, but to create a framework and components that could be used in further teachings and research would be much more beneficial.

The use of CBD was one of the best decisions made for the project. It has improved the possibility for future work with the project, facilitated aspects like bug testing during development, and made it easy to implement new functionality. The Case study presented in

section (4.3.3) also proves the implementation has been successful, and provides easy, accurate and quick prototyping.

5.2. Stakeholder needs (NTNU)

The simulator has passed all the tests required by the research group, especially with regards to usability and effectiveness. For teaching purposes, the simulator allows for quick prototyping, with either simple pitch-yaw design or more specific design choices, and features that provides an easy entrance to the world of modular robots.

In research-use, some of the simulator's features are similar in customizability as hard-coded solutions offer. However, since custom dynamics scripting has not yet been implemented, optimization does not allow for multiple algorithms, and sensory placement options are limited for ease-of-use, it is decided that the simulator has not passed with regards to extensive research-use.

However, with the Core-Framework having more functionality without the Scene Designer, it is highly encouraged that researchers use this while developing new modular robot prototypes. By coding with the framework, robot creation is much faster, and provides customizability as with a hard-coded solution. Since the Core Framework is verified* there is no need for time-consuming development of simulation methods and robot component assembly. The Core Framework only uses Microsoft libraries, making it compatible with all C# development environments.

**note, the Core Framework is verified for this project. It may not be verified if the researcher needs different functionality than the robots and scenario provide in this project. I test is only partly verified, but only by customizability, and the functionality can be achieved by using the Core Framework directly, not through the Scene Designer.*

5.3. Issues

Many problems in the project have been related to the Algorix physics library. Because of the limited possibilities for troubleshooting potential errors, many bugs go unnoticed before a large section of code has been created. Switching out Algorix with another physics engine could have been a solution early in the project, where all the functions in the "Simulation_Core" would remain the same, but the "AgX_Interface" class would be replaced with similar functions from another physics library. Since many of the errors were related to memory access, there

was no way to get information about the failures before Unity force-closed. The solution was to construct a system-flow which ensured disposal of C# objects, and a reset of the Algoryx simulation as often as possible.

The Dynamics class should have been made more abstract, to enable custom dynamics more easily, without having to modify the class itself. Abstraction would have been a feature in the Core Framework too, but since the scenario data is serialized to an XML file, the classes cannot use inheritance implementations. If possible, this would make it possible to create modules and sensor modules as the same top-level entity, and the same with force and distance sensors. Nonetheless, the possibility of saving data to an XML file took precedence over the improved modularity this would result in.

Even though the “AgX_Interface” class was made with focus on a flow ensuring proper disposal, there are still bugs in the Optimization part of the MRSim which could not be fixed through AgX support or troubleshooting. There might be limitations in the AgX software itself, but this is not proven.

5.4. Summary

Based on the results and discussion, the research questions can now be reviewed and answered:

Is the proposed simulation platform better for developing modular robots than manually programming simulations from scratch?

Yes. As validated in (U-1.1.1, U-1.1.2 and U-1.1.3), the MRSim is much faster and easier to use than manually coding by far. The customization options are the same or higher than what is required for teaching, while for research purposes only certain parts are lacking in functionality.

Will the component-based software development method notably improve the flexibility and development of a simulation platform and ease further development of the simulation platform?

Both points of this research question are confirmed to be true. Flexibility and modifiability is partly verified in (R-5.3.1), with the reason for it being only partly being a wish for even more modularity from the “Main” class. Ease of further development is verified in (U-2.1.1 and U-2.2.1), but for the time being only in the Core Framework. Optimization and Dynamics has not yet been tested.

Is the AgX physics library a stable choice for a simulator platform, and how efficient is it in providing realistic simulations while allowing for the effective use of optimization algorithms?

Using AgX as a physics library has notably increased development time. This is described in section (5.3). This could be different if there were easier way of obtaining error information, or the simulator required less complex operations like position resets of the robot, or other optimization functionality. In conclusion: AgX is great for realistic physics simulations and hard-coded development, but when developing an application where the physics features are used as a reusable component interacting with other components, it may be better to use another technology.

Will the benefits of Component-Based Development outweigh the disadvantages of the implementation in an independent, small-scale project?

Yes. (Elaborated in section (5.1).

5.5. Further work

Following elements is suggested for further work, or did not fit into the time-schedule of the project:

- Testing and verifying the Core Framework and components in other environments than Unity, such as Visual studio with DirectX (C#).
- Verification of the Dynamics and Optimization component
- Modularizing the AgX_Interface even further, by eliminating all object references from the Simulation_Core and only associating the components with GUIDs. (while at the same time keeping the ease-of-use of the framework.)
- Eliminating all bugs from the optimization component related to AgX memory issues.
- Drag and drop functionality in the designer, for creating the modular robots.
- Scene designer with more options such as a string input field or .dll upload for custom robot dynamics.
- 3D-printing robot configurations designed in the MRSim, validating the prototyping feasibility.

6. Conclusion

The Modular Robot Simulator has been created for NTNU Ålesund's Modular Robot research group. It is tested and verified to be used for educational purposes and makes modular robot research, design and prototyping more accessible to potential developers/students with little to no programming background. The GUI gives suggestions on parameters to facilitate development, enabling quick prototyping. The scenarios can be customized on a level close to the manually-coded simulations, with uploads of custom 3D models and self-configured terrain.

By using the Component-Based approach, the core functionality of the simulator is contained in two linked libraries, allowing for reusability in other projects, and encouraging custom simulators or simulations to be created by both students and modular robot researchers. The CBD also allows for removal of components like visualization, robot dynamics or optimization, without impacting the rest of the simulator's functionality. New components can be added to projects without difficulty.

7. References

- [1] K. Gilpin and D. Rus, "Modular Robot Systems," *IEEE Robotics & Automation Magazine*, vol. 17, no. 3, pp. 38-55, 2010.
- [2] J. J. Cameron and W. Fisher, Director, *Terminator 2: Judgment Day*. [Film]. 1991.
- [3] B. Mantlo and B. Budiansky, *The Transformers*, New York: Marvel Comics, 1984.
- [4] Cyberobotics Ltd., "Webots User Guide," Cyberobotics, 2018. [Online]. Available: <https://www.cyberobotics.com/doc/guide/introduction-to-webots>. [Accessed 06 03 2018].
- [5] Unity Technologies, "Unity Manual," Unity Technologies, 2018. [Online]. Available: <https://docs.unity3d.com/Manual/index.html>. [Accessed 06 03 2018].
- [6] Autonomous Systems Lab, ModLab, "VSPARC," Cornell University, University of Pennsylvania, , 2012. [Online]. Available: vsparc.org. [Accessed 06 03 2018].
- [7] Modular Robotics Laboratory, "MODLAB UPENN," University of Pennsylvania, 2012. [Online]. Available: <http://www.modlabupenn.org/2012/10/19/smores/>. [Accessed 06 03 2018].
- [8] P. N. Lancheros, L. B. Sanabria and R. Castillo, "Simulation of Modular Robotic System MECABOT in Caterpillar and Snake Configuration Using Webots Software," in *Robotics and Automation (CCRA)*, Bogota, Colombia, 2016.
- [9] R. Castillo, Cotera Mateo and G. Vargas, "Simulation and Implementation of a Hexapod Configuration using Modular Robotics," in *de innovacion y Tendencias en Ingenieria (CONIITI)*, Bogota, Colombia, 2017.
- [10] G. Li, P. Verdru, W. Li and H. Zhang, "A Screw-less Solution for Snake-like Robot Assembly and Sensor Integration," Norwegian University of Science and Technology, Aalesund, Norway, 2017.
- [11] V. Vonasek, D. Fiser, K. Kosnar and L. Preucil, "A Light-Weight Robot Simulator for Modular Robotics," in *Modelling and Simulation for Autonomous Systems*, Switzerland, Springer International Publishing, 2014, pp. 206-216.
- [12] P. A. Winfield, "Symbiotic Evolutionary Robot Organisms," University of Stuttgart, 14 02 2017. [Online]. Available: <http://www.brl.ac.uk/research/researchthemes/swarmrobotics/symbrion.aspx>. [Accessed 06 03 2018].
- [13] I. Crnkovic, M. Chaudron and S. Larsson, "Component-based Development Process and Component Lifecycle," in *International Conference on Software Engineering Advances*, Tahiti, 2006.
- [14] T. Vale, I. Crnkovic, E. Santana, P. Anselmo, Y. Cerqueira and S. Rotaro, "Twenty-eight years of component-based software engineering," *Journal of Systems and Software*, vol. 111, pp. 128-148, 2016.
- [15] M. Burgin and E. Eberbach, "Evolutionary Turing in the Context of Evolutionary Machines," Hartford ; Los Angeles.
- [16] W. Hasselbring, "Component-Based Software Engineering," in *Handbook of Software Engineering and Knowledge Engineering*, Oldenburg, Germany, World Scientific Publishing Co., 2002, pp. 289-306.
- [17] M. Kaushik and M. S. Dulawat, "A Comparison Between Traditional and Component Based Software Development Process Models," in *Journal of Computer and Mathematical Sciences Vol. 3*, Udaipur, India, 2012, pp. 308-319.
- [18] cjdev, "Unity Answers," Unity, 21 08 2015. [Online]. Available: <https://answers.unity.com/questions/1033085/heightmap-to-mesh.html>. [Accessed 02 18].
- [19] W3schools, "W3schools.com," Refsnes Data, [Online]. Available: https://www.w3schools.com/xml/xpath_intro.asp. [Accessed 20 11 2017].

- [20] X. Cui, X. Zang, Y. Zhu, S. Tang and J. Zhao, "CPG based locomotion control of pitch-yaw connecting modular self-reconfigurable robots," *IEEE International conference on Information and Automation*, pp. 1410-1415, 2010.
- [21] Janalta, "Techopedia," Janalta Interactive, [Online]. Available: <https://www.techopedia.com/definition/25972/modular-programming>. [Accessed 21 11 2017].
- [22] R. Jha, "DotNetConcept," 29 11 2014. [Online]. Available: <http://www.dotnet-concept.com/Articles/2014/11/29/Understanding-Structured-Unstructured-and-Modular-programming-approach>. [Accessed 21 11 2017].
- [23] S. M. Sani and Y. K. Shokooh, "Minimalism in designing user interface of commercial websites based on Gestalt visual perception laws," in *Web Research (ICWR)*, Tehran, 2016.

Appendix A Preliminary RISK

Nr	RISK	Likelihood	Impact	Total	Mitigation action	Contingency plan	Mitigation date
1	Overall program	3.80	3.60	13.68			
1.1	Development risk	3.80	3.60	13.68			
1.1.1	bugs	5.00	1.00	5.00	Modular programming, structured coding for easy identification of bugs.	Ensure bugs can be easily identified and fixed.	E2,C1-3
1.1.2	Incompatible technologies	4.00	3.00	12.00	Ensure Elaboration phase is thorough, design with possibility of different implementations later	Allot extra time for modification when planning deadlines.	E1-2
1.1.3	Wrong technology decision	4.00	4.00	16.00	Design with possibility of different implementations later	Step back on project goals, focus on a simple but stable solution	E2
1.1.4	Misinterpretation of end result	2.00	5.00	10.00	Concurrent dialogue with supervisors	Create multiple areas of value in the project	E1
1.1.5	Program modules do not work together	4.00	5.00	20.00	Modular programming, Clearly defined class/functionality structure	Hardcoding	E2,C1-3
1.1.6	Missing functionality	4.00	4.00	16.00	Create technical documents for overview of functionality and ensurance of continued work	Ensure program can work without specific parts.	E1-2,C1-3

2	Scene Designer	3.23	3.67	11.86			
2.1	functionality	4.00	5.00	20.00			
2.1.1	User cannot create the desired robot configuration	4.00	5.00	20.00	Ensure enough parameters can be set.	Create alternative method for configuration (low-level code input)	E2,C1-2
2.2	XML issues	3.50	4.00	14.00			
2.2.1	Robot parameters cannot be sent	3.00	5.00	15.00	Evaluate additional technologies of file transfer	Create everything in one program	E2, C2-3
2.2.2	Robot movement script is not working	4.00	3.00	12.00	Ensure the method for setting robot dynamics is reliable, enable different implementation solutions	Set one specific movement pattern.	E2,C2-3
2.3	load/save issues	2.00	2.00	4.00			
2.3.1	Configurations cannot be saved	1.00	2.00	2.00	Ensure file is saved before Simulation is started	None	E2,C1-3
2.3.2	Select configurations cannot be loaded	3.00	2.00	6.00	Allow for saving of different revisions of configuration.	Only save one file	E2,C1-3
2.4	Dynamics issues	3.67	3.33	12.22			
2.4.1	Robot dynamics are incorrect	4.00	5.00	20.00	Dialogue with supervisor	Allot time for research on modulr robotics	E2,C1-2
2.4.2	Script input is too complex	3.00	2.00	6.00	Ensure UI is intuitive	Require user to upload a file instead of text input	E2,C3
2.4.3	Script functionality cannot be implemented	4.00	3.00	12.00	Design UI with possibility for other implementations	Don't allow for scripting	E2,C3

2.5	Scene creation issues	3.00	2.50	7.50			
2.5.1	Created heightmap has physics errors	4.00	4.00	16.00	Ensure height image has the correct values/dimensions, get programming guidance from AgX	Define other types of colliders, worst case: cube collider	E2,C1-2
2.5.2	Scene features are hard to correctly design	2.00	1.00	2.00	Make user choose features from list, feedback from test users	Create simple interface with minimal customization	E2,C2-3
2.5	Reliability	3.00	4.00	12.00			
2.5.1	Scene Designer is unstable	3.00	4.00	12.00	Modular programming, structured coding for easy identification of bugs.	Use another program for scene designer	E2,C1
3	Simulation	2.00	5.00	10.00			
3.1	functionality	2.00	5.00	10.00			
3.1.1	Program does not contain desired functionality	2.00	5.00	10.00	Keep high intervals for meetings with supervisor	Redefine requirements	E1-2,C1-3
3.2	XML issues	3.00	5.00	15.00			
3.2.1	Robot components are not created correctly	3.00	5.00	15.00	Technical document for object creations from XML, early conceptual models of classes		E1-2,C1-2
3.2.2	Robot is not moving correctly	3.00	5.00	15.00	Ensure the method for setting robot dynamics is reliable, enable different implementation solutions, alternatively, set static dynamics script	Abstract classes for further implementation later	E2,C2-3

3.3	Algoryx and Unity not working together	4.00	4.67	18.67			
3.3.1	Objects not behaving correctly	4.00	5.00	20.00	Technical document for object creations from XML, early conceptual models of classes	keep options open for other physics engines and visualization platforms	E1-2,C1-2
3.3.2	Visualization not matching physics	4.00	5.00	20.00	Research, thorough and clean code		E1-T1
3.3.3	Missing features	4.00	4.00	16.00	Regular contact with stakeholders, thorough and specific requirements	Component or modular system for easy implementation of additional options	E1-2,C1
3.4	Robot optimization	2.00	4.00	8.00			
3.4.1	Algorithm does not improve functionality	2.00	4.00	8.00	Research, evaluate multiple techniques, ensure simulation stability	Create an open optimization component so user can edit themselves	E2,C2-3
3.5	Reliability	3.00	3.50	10.50			
3.5.1	Simulation is lagging	3.00	3.00	9.00	Optimize code, create clean code, consult Algoryx	switch physics engine	C1-3
3.5.2	Physics are not accurate	3.00	4.00	12.00	Consult Algoryx, research documentation	switch physics engine	C1-3
4	Project	3.67	3.33	12.22			
4.1	Project not finished in time	4.00	3.00	12.00	Set clear goals/milestones, create thorough plan in inception/elaboration phases	set due dates several weeks earlier	I1,E1-2
4.2	Missing requirements	3.00	5.00	15.00	Meetings at the end of each iteration	set due dates several weeks earlier	E1-T1
4.4	Change in requirements	4.00	2.00	8.00	Sprint meetings every second week		E1-T1

Appendix B Requirement Specification

Project:

Component-Based Simulator for Modelling the Design and Dynamics of Modular Robots

University:

NTNU Ålesund

Document:

0003_Requirements

Rev.	Date:	Author:
1.0	29.05.2018	Torstein Sundnes Lénérand



i. Contents

I. CONTENTS	6
II. LIST OF TABLES	7
III. DOCUMENT HISTORY	8
I. REQUIREMENTS	9
1.1. INTRODUCTION	9
1.2. SIMULATOR REQUIREMENTS	10
1.2.1. <i>Simulator usage</i>	10
1.2.1.1. Save/Load	10
1.2.1.2. Performance	11
1.2.1.3. GUI	12
1.2.1.4. Scenario	13
1.2.2. <i>Core Framework</i>	14
1.2.2.1. Robot	14
1.2.2.2. Frame	14
1.2.2.3. Joint	15
1.2.2.4. Module	15
1.2.2.5. Sensors/sensor modules	16
1.2.2.6. Scene	17
1.2.3. <i>Dynamics</i>	18
1.2.4. <i>Optimization</i>	19
1.3. COMPONENT-BASED DESIGN	20
1.3.1. <i>Cost reduction</i>	20
1.3.2. <i>Ease of assembly</i>	20
1.3.3. <i>Reusability</i>	21
1.3.4. <i>Customization flexibility</i>	21
1.3.5. <i>Maintainability</i>	22

ii. List of tables

TABLE 38: DOCUMENT HISTORY.....	8
TABLE 39: REQUIREMENT CATEGORIES.....	9
TABLE 40: TEST CATEGORIES	9
TABLE 41: SAVE/LOAD.....	10
TABLE 42: PERFORMANCE.....	11
TABLE 43: GUI	12
TABLE 44: SCENARIO	13
TABLE 45: ROBOT	14
TABLE 46: FRAME	14
TABLE 47: JOINT.....	15
TABLE 48: MODULE	15
TABLE 49: SENSORS AND SENSOR MODULES.....	16
TABLE 50: SCENE.....	17
TABLE 51: DYNAMICS.....	18
TABLE 52: OPTIMIZATION.....	19
TABLE 53: COST REDUCTION	20
TABLE 54: EASE OF ASSEMBLY	20
TABLE 55: REUSABILITY.....	21
TABLE 56: CUSTOMIZATION FLEXIBILITY.....	21
TABLE 57: MAINTAINABILITY.....	22

iii. Document history

Rev.	Date	Author	Description
0.1	15.01.2018	TS	Document created
0.2	18.04.2018	TS	Updated
1.0	29.05.2018	TS	Updated with verification and added to report

Table 38: Document history

1. Requirements

1.1. Introduction

The requirements for this project are divided into categories depending on the necessity for the requirements, and the potential risk impact.

A	The requirement shall be met to ensure a stable component
B	The requirement should be met to ensure an efficient component
C	The requirement is optional or flagged as further work

Table 39: Requirement categories

The requirements are also divided into categories for verification methods, featured after the requirement number. T is standard verification based on performed tests, U is a verification based on a test case for the selected requirement/component, and R is verification based on review and analysis of components and features.

T	Test verification
R	Review verification
U	Use-case and comparison verification

Table 40: Test categories

1.2. Simulator requirements

These are the requirements for the Modular Robot Simulator application

1.2.1. Simulator usage

1.2.1.1. Save/Load

Nr:	Requirement	Category	Originator	Verified?
REQ-1.1.1T	The user shall be able to save scenario configurations	A	NTNU	Verified T-1.1.1
REQ-1.1.2T	The user shall be able to load scenario configurations	A	NTNU	Verified T-1.1.1
REQ-1.1.3T	The user should be able to load a scenario stopped mid-execution.	B	TS	Not verified T-1.1.2
REQ-1.1.4R	The saved robot values should be represented in a format facilitating potential prototyping	B	TS	Partly Verified R-1.1.1
REQ-1.1.5R	All aspects of the simulation shall be transferrable in one single XML file	A	TS	Partly verified R-1.1.1

Table 41: Save/load

1.2.1.2. Performance

Nr:	Requirement	Category	Originator	Verified?
REQ-1.2.2T	The simulator application shall be usable on mid-range+ workstations.	A	TS	Verified T-1.2.1
REQ-1.2.3T	The physics in the simulation shall be performed in real-time on mid-range+ workstations.	A	TS	Verified T-1.2.1
REQ-1.2.4T	The simulator application shall be stable.	A	TS	Partly verified T-1.2.1
REQ-1.2.5T	The simulation shall not stop mid-execution.	A	TS	Partly verified T-1.2.1
REQ-1.2.6R	Robot components which require so, shall exist in the physics environment.	A	TS	Verified R-1.2.1
REQ-1.2.7R	Information about robot components existing in the physics environment shall be retrievable.	A	TS	Verified R-1.2.1
REQ-1.2.8T	Physics objects shall not unintentionally overlap.	A	TS	Verified T-1.2.1
REQ-1.2.9T	Physics objects shall not pass through each other.	A	TS	Verified T-1.2.1

Table 42: Performance

1.2.1.3. GUI

Nr:	Requirement	Category	Originator	Verified?
REQ-1.3.1T	The simulator shall preview the proposed robot components during the design.	A	TS	Verified T-1.3.1
REQ-1.3.2T	The simulator shall show all robot components with correct physics representations.	A	TS	Verified T-1.3.1
REQ-1.3.3T	The simulator should enable modification of the scene view.	B	TS	Partly verified T-1.3.2
REQ-1.3.4R	The Scene Designer should realize all potential functionality in the Core Framework.	B	TS	Verified R-1.3.2

Table 43: GUI

1.2.1.4. Scenario

Nr:	Requirement	Category	Originator	Verified?
REQ-1.4.1U	The simulator shall make scenario creation easier than the current programming methods.	A	NTNU	Verified U-1.1.1
REQ-1.4.2U	The simulator shall enable intuitive and easy design of robots.	A	NTNU	Verified U-1.4.2
REQ-1.4.3U	The simulator should provide customizability of robots similar to hard-coded programs, for experienced users.	B	TS	Partly verified U-1.1.2
REQ-1.4.4U	The simulator shall enable faster design than hard-coded programming, regardless of user experience.	A	TS	Verified U-1.1.3
REQ-1.4.5R	The simulator shall not contain unused parameters	B	TS	Verified R-1.4.1

Table 44: Scenario

1.2.2. Core Framework

These are the requirements for the Core Framework. Some of the requirements are disconnected from the simulator application, as the Core Framework is designed to work as a standalone library.

1.2.2.1. Robot

Nr:	Requirement	Category	Originator	Verified?
REQ-2.1.1T	Robot modules shall be locked together	A	TS	Verified T-2.1.1
REQ-2.1.2T	Robot sensor modules shall be locked together	A	TS	Verified T-2.1.1
REQ-2.1.3T	Robot assembly transform values shall be readable	A	TS	Verified T-2.1.2
REQ-2.1.4R	Robot objects shall represent the entire robot assembly	A	TS	Verified R-2.1.1

Table 45: Robot

1.2.2.2. Frame

Nr:	Requirement	Category	Originator	Verified?
REQ-2.2.1U	The framework shall allow for customization of all necessary frame parameters.	A	NTNU	Verified U-1.1.2
REQ-2.2.2T	The framework shall allow for custom implementation of frame meshes.	A	TS	Verified T-2.2.1
REQ-2.2.3T	Frames shall have global transform properties.	A	TS	Verified T-2.2.2
REQ-2.2.4T	Frame transform values shall be readable	A	TS	Verified T-2.2.2
REQ-2.2.5T	Frames shall be visualized	A	NTNU	Verified T-2.2.2

Table 46: Frame

1.2.2.3. Joint

Nr:	Requirement	Category	Originator	Verified?
REQ-2.3.1U	The framework shall allow for customization of all necessary joint parameters	A	NTNU	Partly verified U-1.1.2
REQ-2.3.2T	Joint angle shall be modifiable during runtime of the physics environment.	A	TS	Verified T-2.3.1
REQ-2.3.3T	Joints shall connect frames.	A	TS	Verified T-2.3.1
REQ-2.3.4T	Joints shall connect modules.	A	TS	Verified T-2.3.1
REQ-2.3.5T	Joints shall connect sensor modules.	A	TS	Verified T-2.3.1
REQ-2.3.6T	Joints shall connect sensors.	A	TS	Verified T-2.3.1
REQ-2.3.7T	Joints should be detachable.	C	TS	N/A T-2.3.2
REQ-2.3.8T	Joint transform values should be readable.	B	TS	N/A T-2.3.3

Table 47: Joint

1.2.2.4. Module

Nr:	Requirement	Category	Originator	Verified?
REQ-2.4.1T	Each module shall contain one “Joint” object.	A	TS	Verified T-2.4.1
REQ-2.4.2T	Each module shall contain two “Frame” objects.	A	TS	Verified T-2.4.1
REQ-2.4.3R	There should be no limits on number of modules in a robot.	B	TS	Verified R-2.2.1
REQ-2.4.4T	Module transform values should be readable.	B	TS	Verified T-2.4.2
REQ-2.4.5U	The framework shall allow for customization of all necessary module parameters	A	NTNU	Verified U-1.1.2

Table 48: Module

1.2.2.5. Sensors/sensor modules

Nr:	Requirement	Category	Originator	Verified?
REQ-2.5.1T	The framework should allow for custom placements of sensor modules in the robot.	B	NTNU	Partly verified T-2.5.1
REQ-2.5.2T	The framework should allow for custom sensor placements on sensor modules.	B	NTNU	Verified T-2.5.1
REQ-2.5.3T	Sensor module transform values shall be readable.	A	TS	Verified T-2.5.2
REQ-2.5.4T	Force sensor values shall be readable.	A	NTNU	Verified T-2.5.2
REQ-2.5.5T	Distance sensor values shall be readable.	A	NTNU	Verified T-2.5.2
REQ-2.5.6T	Sensor modules shall be visualized.	A	NTNU	Verified T-2.5.2
REQ-2.5.7T	Sensors should be visualized.	B	TS	Verified T-2.5.2
REQ-2.5.8U	The framework shall allow for customization of all necessary sensor module parameters.	A	NTNU	Verified U-1.1.2
REQ-2.5.9U	The framework shall allow for customization of all necessary sensor parameters.	A	NTNU	Partly verified U-1.1.2

Table 49: Sensors and sensor modules

1.2.2.6. Scene

Nr:	Requirement	Category	Originator	Verified?
REQ-2.6.1U	The framework shall allow for customization of all necessary terrain parameters	A	NTNU	Verified U-1.1.2
REQ-2.6.2T	The created terrain shall be physically accurate based on design parameters.	A	TS	Verified T-2.6.1
REQ-2.6.3T	The terrain should be created using a heightmap image file.	B	TS	Verified T-2.6.1
REQ-2.6.4T	The terrain's visual representation shall match the physical representation.	A	TS	Verified T-2.6.1
REQ-2.6.5T	Scene should contain functionality for water fields.	B	TS	N/A T-2.6.2
REQ-2.6.6T	Scene should contain functionality for air resistance fields.	B	TS	N/A T-2.6.2

Table 50: Scene

1.2.3. Dynamics

Nr:	Requirement	Category	Originator	Verified?
REQ-3.1.1T	The dynamics component shall control the movable elements of the robot. (joints)	A	NTNU	Verified T-2.3.1
REQ-3.1.2R	The dynamics component shall be separate from the Core Framework	A	TS	Verified R-3.1.1
REQ-3.1.3T	The dynamics component shall enable robot forward/backwards movement	A	NTNU	Verified T-3.1.1
REQ-3.1.4T	The dynamics component shall enable robot left/right turns	A	NTNU	Verified T-3.1.1
REQ-3.1.5T	The dynamics component should enable robot sideways motion	B	TS	Partly verified T-3.1.1
REQ-3.1.6T	The dynamics component should enable modification of movement parameters to create custom motion commands for the robot.	B	TS	Verified T-3.1.1
REQ-3.1.7R	The dynamics component should enable custom robot movement scripts created by advanced users.	B	TS	N/A R-3.1.2

Table 51: Dynamics

1.2.4. Optimization

Nr:	Requirement	Category	Originator	Verified?
REQ-4.1.1R	The optimization component shall be separate from the Core Framework.	A	TS	Verified R-4.1.1
REQ-4.1.2R	The optimization component should contain functionality for multiple optimization algorithms.	B	TS	N/A R-4.1.2
REQ-4.1.3T	The optimization algorithms should be able to fast-forward to a specific simulation time-step.	B	TS	Partly verified T-4.1.1
REQ-4.1.4T	The user shall be able to optimize movement variables.	A	TS	Verified T-4.1.1
REQ-4.1.5T	The user shall be able to optimize robot design.	A	TS	Not verified T-4.1.1
REQ-4.1.6T	The user shall be able to select which parameters to not optimize throughout the optimization process.	A	TS	Verified T-4.1.1

Table 52: Optimization

1.3. Component-Based Design

These are the requirements for the CBD aspect of the Modular Robot Simulator and software components. Most requirements are abstract, prompting verification by review and analysis, rather than testing.

1.3.1. Cost reduction

Nr:	Requirement	Category	Originator	Verified?
REQ-5.1.1R	Future creation of modular robot simulators should be faster/cheaper using this framework.	B	TS	Verified R-5.1.1
REQ-5.1.2R	Time used on modifying components should be notably lower compared to modifying traditional systems.	B	TS	Verified R-5.1.2

Table 53: Cost reduction

1.3.2. Ease of assembly

Nr:	Requirement	Category	Originator	Verified?
REQ-5.2.1U	Namespace/class components shall be easier to assemble into a usable program, than to create a program from scratch.	A	TS	Verified U-2.1.1
REQ-5.2.2T	Custom variable structures shall be usable by all class/namespace components.	A	TS	Verified T-5.1.1
REQ-5.2.3R	CBS should make it easier to assemble a robot in various configurations.	B	TS	N/A R-5.2.1

Table 54: Ease of assembly

1.3.3. Reusability

Nr:	Requirement	Category	Originator	Verified?
REQ-5.3.1U	The Core Framework shall be reusable in new projects.	A	TS	Verified U-2.2.1
REQ-5.3.2U	The Dynamics component shall be reusable in other projects with the Core Framework.	A	TS	Not tested U-2.2.1
REQ-5.3.3U	The Optimization component shall be reusable in other projects with the Core Framework.	A	TS	Not tested U-2.2.1
REQ-5.3.4U	The Core Framework should enable creation of simple simulations using primitive shapes in this and other projects.	B	TS	Verified U-2.2.1
REQ-5.3.5U	Objects from classes such as Frames, Joints, SensorModules and SceneObjects should be usable without the creation of robot assemblies.	B	TS	Not verified U-2.2.1

Table 55: Reusability

1.3.4. Customization flexibility

Nr:	Requirement	Category	Originator	Verified?
REQ-5.4.1R	Implementation of new components shall be possible by default.	A	TS	Verified R-5.3.1
REQ-5.4.2R	Modification of the system features shall be possible by only changing the specific component.	A	TS	Partly verified R-5.3.1

Table 56: Customization flexibility

1.3.5. Maintainability

Nr:	Requirement	Category	Originator	Verified?
REQ-5.5.1R	Maintenance/fixes in the code shall be easier to perform compared to traditional software solutions.	B	TS	Verified R-5.4.1

Table 57: Maintainability

Appendix C Test and Verification

Project:

Component-Based Simulator for Modelling the Design and Dynamics of Modular Robots

University:

NTNU Ålesund

Document:

0009_Test&Verification

Rev.	Date:	Author:
2.0	30.05.2018	Torstein Sundnes Lénérand



i. Contents

I. CONTENTS	24
II. LIST OF TABLES	25
III. DOCUMENT HISTORY	26
I. TEST & VERIFICATION	27
1.1. TESTS.....	28
1.1.1. Simulator usage.....	28
1.1.1.1. Save/Load	28
1.1.1.2. Performance	29
1.1.1.3. GUI.....	29
1.1.2. Core Framework	30
1.1.2.1. Robot	30
1.1.2.2. Frame.....	30
1.1.2.3. Joint	31
1.1.2.4. Module.....	32
1.1.2.5. Sensors/sensor modules.....	32
1.1.2.6. Scene.....	33
1.1.3. Dynamics	33
1.1.4. Optimization.....	34
1.1.5. Component-based design	34
1.1.5.1. Ease of assembly.....	34
1.2. REVIEWS AND ANALYSIS	35
1.2.1. Simulator usage.....	35
1.2.1.1. Save/Loads.....	35
1.2.1.2. Performance	35
1.2.1.3. GUI	36
1.2.1.4. Scenario	36
1.2.2. Core Framework	36
1.2.2.1. Robot	36
1.2.2.2. Module.....	36
1.2.3. Dynamics	37
1.2.4. Optimization.....	37
1.2.5. Component-based design	38
1.2.5.1. Cost reduction.....	38
1.2.5.2. Ease of assembly.....	38
1.2.5.3. Customization flexibility.....	39
1.2.5.4. Maintainability.....	39
1.3. USE-CASES AND COMPARISONS	40
1.3.1. Simulator usage.....	40
1.3.1.1. Scenario	40
1.3.2. Component-Based Design	41
1.3.2.1. Ease of assembly.....	41
1.3.2.2. Reusability	41

ii. List of tables

TABLE 58: DOCUMENT HISTORY	26
TABLE 59: TEST CATEGORIES	27
TABLE 60: T-1.1.1	28
TABLE 61: T-1.1.2	28
TABLE 62: T-1.2.1	29
TABLE 63: T-1.3.1	29
TABLE 64: T-1.3.2	29
TABLE 65: T-2.1.1	30
TABLE 66: T-2.1.2	30
TABLE 67: T-2.2.1	30
TABLE 68: T-2.2.2	30
TABLE 69: T-2.3.1	31
TABLE 70: T-2.3.2	31
TABLE 71: T-2.3.3	31
TABLE 72: T-2.4.1	32
TABLE 73: T-2.4.2	32
TABLE 74: T-2.5.1	32
TABLE 75: T-2.5.2	32
TABLE 76: T-2.6.1	33
TABLE 77: T-2.6.2	33
TABLE 78: T-3.1.1	33
TABLE 79: T-4.1.1	34
TABLE 80: T-5.1.1	34
TABLE 81: R-1.1.1	35
TABLE 82: R-1.2.1	35
TABLE 83: R-1.3.2	36
TABLE 84: R-1.4.1	36
TABLE 85: R-2.1.1	36
TABLE 86: R-2.2.1	36
TABLE 87: R-3.1.1	37
TABLE 88: R-3.1.2	37
TABLE 89: R-4.1.1	37
TABLE 90: R-4.1.2	37
TABLE 91: R-5.1.1	38
TABLE 92: R-5.1.2	38
TABLE 93: R-5.2.1	38
TABLE 94: R-5.3.1	39
TABLE 95: R-5.4.1	39
TABLE 96: U-1.1.1	40
TABLE 97: U-1.1.2	40
TABLE 98: U-1.1.3	41
TABLE 99: U-2.1.1	41
TABLE 100: U-2.2.1	41

iii. Document history

Rev.	Date	Author	Description
0.1	15.01.2018	TS	Document created
0.2	09.03.2018	TS	Tests updated
1.0	18.04.2018	TS	Tests finalized
2.0	30.05.2018	TS	Tests performed, document added to report

Table 58: Document history

1. Test & Verification

The test & verification specification ensures the simulator meets the requirements defined in the requirement specification. There are three different categories of tests:

- Actual testing (T-X.X.X)
- Review and analysis of specifications (R-X.X.X)
- Testing a use-case and comparing to other solutions (U-X.X.X)

Tests are performed on the sub-topics specified in the requirement specification and will be verified by the means described in Table 59. Each test and verification table will contain a pass criterion, test execution (method) and a result, following a “comments” fields to elaborate on the test results if necessary. Some tests also have discussion fields.

T	Test verification
R	Review verification
U	Use-case and comparison verification

Table 59: Test categories

1.1. Tests

These are the tests performed within the project.

1.1.1. Simulator usage

1.1.1.1. Save/Load

T-1.1.1	REQ-1.1.1T	REQ-1.1.2T
Pass criteria	Configurations are saved to file, then loaded in a new simulation.	
Method	5 different configurations are created and saved. The program is shut down, and configurations are loaded to the simulator. Repeat 5 times with the different configurations. Analyze the ease-of-use.	
Result	Verified	
Comment	Performed in build application.	
Discussion	A file browser instead of typing file name directly as in this prototype scene designer would be more user-friendly.	

Table 60: T-1.1.1

T-1.1.2	REQ-1.1.3T
Pass criteria	Scenario in a started simulation is saved, then loaded in a new simulation.
Method	5 different configurations are created, and simulation is started. All 5 are saved during runtime. The 5 configurations are loaded and started. Analyze results and observations.
Result	Failed
Comment	Performed in build application. Robot can be saved mid-execution and loaded later. However, joint angles are incorrect, since the initialization of the robot assumes frame rotations as different configurations (pitch/yaw). This causes the joints to be wrongly attached.
Discussion	Solution may be to disable pitch/yaw separations, and let user manually specify configurations. However, this was not done due to increased complexity in designing the robots.

Table 61: T-1.1.2

1.1.1.2. Performance

T-1.2.1	REQ-1.2.2T	REQ-1.2.3T	REQ-1.2.4T	REQ-1.2.5T	REQ-1.2.8T	REQ-1.2.9T
Pass criteria	The simulator application shall be usable and perform real-time physics on a mid-range computer. There shall be no program crashes or mid-simulation crashes. Physics objects shall not overlap or pass through each other.					
Method	Run 10 simulations. For each simulation: <ul style="list-style-type: none"> • Create a unique scenario. • Run 1 minute. • Ensure physics are not updated too fast or too slow (observe). • Note any crashes. • Zoom in on robot if necessary to observe physics errors. 					
Result	Partly Verified					
Comment	Performed in build application. Optimization has been observed to crash. Rest of the simulator performing according to specification.					

Table 62: T-1.2.1

1.1.1.3. GUI

T-1.3.1	REQ-1.3.1T	REQ-1.3.2T
Pass criteria	Robots shall be previewed during the design phase, with enough detail to see the general shape of the robot. The robot shall also be correctly visualized during the simulation, according to physics and transforms.	
Method	Design a robot. Make sure robot components are representative in the way of shapes and orientations while designing. Start simulation. Observe and zoom in on robot to ensure visualizations match physics behavior and design.	
Result	Verified	
Comment	Robot is previewed with details matching the main simulation visualization.	

Table 63: T-1.3.1

T-1.3.2	REQ-1.3.3T
Pass criteria	Zoom, movement and rotation of camera shall be possible while the simulation is running.
Method	Start a simulation. Rotate around the created robot. Move camera around the created robot. Zoom in on the created robot.
Result	Partly verified
Comment	Performed in build application. Camera movement not working.

Table 64: T-1.3.2

1.1.2. Core Framework

1.1.2.1. Robot

T-2.1.1	REQ-2.1.1T	REQ-2.1.2T
Pass criteria	Robot modules and sensor modules are correctly locked together during simulation.	
Method	Create 5 different robot configurations with different module sizes. Observe module/sensor-module connections. Review Robot class code structure.	
Result	Verified	
Comment	Performed in build application Two sensor modules cannot be connected to each other based on current Core Framework code structure.	

Table 65: T-2.1.1

T-2.1.2	REQ-2.1.3T
Pass criteria	Position and rotation of the entire robot can be retrieved.
Method	In the update loop (or a similar function) retrieve the positional and rotational values of the robot.
Result	Verified
Comment	Performed in editor.

Table 66: T-2.1.2

1.1.2.2. Frame

T-2.2.1	REQ-2.2.2T
Pass criteria	Updating the mesh files shall update the mesh of the robot frames.
Method	Design robot. Start simulation. Observe. Upload new mesh to the StreamingAssets/Robot folder. Design robot. Start simulation. Observe. Analyze eventual discrepancies.
Result	Verified
Comment	Performed in build application.

Table 67: T-2.2.1

T-2.2.2	REQ-2.2.3T	REQ-2.2.4T	REQ-2.2.5T
Pass criteria	Frame objects have position, rotation and scale properties. These can be retrieved from outside of the class. They have correct values. Values received can be used for visualization		
Method	Modify update loop to print out transform values of a Frame object. Start simulation. Analyze values compared to visualization.		
Result	Verified		
Comment	Performed in editor.		

Table 68: T-2.2.2

1.1.2.3. Joint

T-2.3.1	REQ-2.3.2T	REQ-2.3.3T	REQ-2.3.4T	REQ-2.3.5T	REQ-2.3.6T	REQ-3.1.1T
Pass criteria	Joint movement is performed during simulation. Frames, modules, sensor modules and sensors are connected correctly with joints.					
Method	Run simulation with dynamics component. Ensure joint angles are modified by the component. Ensure all robot components are correctly attached and do not detach. Ensure different joints perform their specific task (locks vs. hinges).					
Result	Verified					
Comment	Performed in build application.					

Table 69: T-2.3.1

T-2.3.2	REQ-2.3.7T
Pass criteria	Joints detach.
Method	Modify dynamics component to enable detachment of joints. Design robot. Run simulation. Ensure joints detach correctly.
Result	N/A
Comment	Functionality not implemented.

Table 70: T-2.3.2

T-2.3.3	REQ-2.3.8T
Pass criteria	Joint position and rotation is readable.
Method	Print out a Joint objects position and rotation in the update loop. Ensure values are correct.
Result	N/A
Comment	Functionality not implemented.
Discussion	Can be approximated by interpolating between the two objects connected to a specific joint.

Table 71: T-2.3.3

1.1.2.4. Module

T-2.4.1	REQ-2.4.1T	REQ-2.4.2T
Pass criteria	A Module object contains one joint object and two frame objects. No more, no less.	
Method	Review class structure of a module. Design robot, run simulation. Observe all modules in simulation for errors in implementation.	
Result	Verified	
Comment	Performed in build application.	

Table 72: T-2.4.1

T-2.4.2	REQ-2.4.4T
Pass criteria	Module transform values can be retrieved.
Method	Print out a Module object's position from the update loop. Ensure values are correct.
Result	Verified
Comment	Performed in editor.

Table 73: T-2.4.2

1.1.2.5. Sensors/sensor modules

T-2.5.1	REQ-2.5.1T	REQ-2.5.2T
Pass criteria	User may decide where sensor modules are placed and their size. User may decide where sensors are placed on sensor modules.	
Method	While designing a robot, configure the sensor modules with different positions/sizes. While designing a robot, configure the sensors with different positions on the sensor modules. Run simulator to test design.	
Result	Partly verified	
Comment	Sensor modules may only be placed between any modules with a pre-set position in the Scene Designer, but using the Core Framework, Sensor Module position may be chosen manually. Force sensors can be placed in 1 of 4 locations on sensor modules. Distance sensors can be placed in 6 locations on sensor modules.	

Table 74: T-2.5.1

T-2.5.2	REQ-2.5.3T	REQ-2.5.4T	REQ-2.5.5T	REQ-2.5.6T	REQ-2.5.7T
Pass criteria	Sensor module position and rotation is readable. Force and distance sensor values are readable. Sensor module transform values can be used for visualization. Sensor transform values can be used for visualization.				
Method	Design robot. Start simulation. Ensure visualization of sensor modules and sensors is visualized according to standard physics behavior. Ensure the analytics log is saving to file with a reasonable interval. Analyze log-file and review values.				
Result	Verified				
Comment	Performed in build application.				

Table 75: T-2.5.2

1.1.2.6. Scene

T-2.6.1	REQ-2.6.2T	REQ-2.6.3T	REQ-2.6.4T
Pass criteria	The terrain height-mesh is created by an image file. The terrain matches the image. The visuals of the terrain match where the visuals of the robot interact.		
Method	Create a scenario. Check if terrain matches the image by robot interaction. Create a new scenario. Select a new image. Check if terrain matches the image.		
Result	Verified		
Comment	Performed in build application.		

Table 76: T-2.6.1

T-2.6.2	REQ-2.6.5T	REQ-2.6.6T
Pass criteria	Terrain can contain water and air resistance fields.	
Method	Create terrain with water field. Create scenario. Review robot performance in field. Create terrain with air field. Create scenario. Review robot performance in field.	
Result	N/A	
Comment	Functionality not implemented.	

Table 77: T-2.6.2

1.1.3. Dynamics

T-3.1.1	REQ-3.1.3T	REQ-3.1.4T	REQ-3.1.5T	REQ-3.1.6T
Pass criteria	Robot can move forwards and backwards. Robot can turn left and right. Robot can move sideways. Robot can move in a user-defined manner.			
Method	Create scenario. Move robot forward. move robot backward. Turn robot left, turn robot right. Move robot left, move robot right. Create 5 different custom movement patterns using the dynamics parameters.			
Result	Partly verified			
Comment	Performed in build application. Dynamics does not contain functionality for left and right. (REQ-3.1.5T)			
Discussion	Left and right movement can be achieved by creating a custom parameter selection for movement.			

Table 78: T-3.1.1

1.1.4. Optimization

T-4.1.1	REQ-4.1.3T	REQ-4.1.4T	REQ-4.1.5T	REQ-4.1.6T
Pass criteria	Simulation can jump to specific time-step while optimizing. Movement variables are optimized. Design variables are optimized. Specific variables can be selected/deselected for optimization.			
Method	Create scenario. Run simulation. Perform for movement and design: Select only 1 variable to optimize. Jump to 2 nd Generation. Observe for errors. Repeat, selecting all variables to optimize. Observe for error.			
Result	Partly verified			
Comment	Performed in build application. Design variables are not optimized. (REQ-4.1.5T) Algorix library errors cause the optimization to crash after too many GA population iterations.			

Table 79: T-4.1.1

1.1.5. Component-based design

1.1.5.1. Ease of assembly

T-5.1.1	REQ-5.2.2T
Pass criteria	Vector2, Vector3 and Quaternion structures are used successfully for all operations in the Core Framework and component classes.
Method	If the visualization is working, the terrain is correctly created and rotations seem correct, the structures are working. Run simulation and check for errors. Analyze code to see if Dynamics and Optimization components can successfully implement the structures.
Result	Verified
Comment	Performed in build application.

Table 80: T-5.1.1

1.2. Reviews and analysis

1.2.1. Simulator usage

1.2.1.1. Save/Loads

R-1.1.1	REQ-1.1.4R	REQ-1.1.5R
Pass criteria	A simulation can be created with all necessary data coming from the XML file (except for an initialization call). The XML class contents are representative of the data used when physically creating modular robots.	
Method	Review the data flow from the Scene Designer to the Main class. Check which variables are passed, and if any simulation properties are not coming from the XML file. Review the XML file, and find whether it contains all necessary info for creating a real-life modular robot, such as sizes, assembly info, joint angles, etc.	
Result	Partly verified	
Comment	All simulation properties relating to the scenario come through the XML file. However, due to the Scene Designer needing to be designed with the specific engine (Unity in this case), several buttons and values, such as simulation start/stop, local file system paths and dynamics/optimization values must be transmitted from Scene Designer to the main file, outside of the XML file.	
Discussion	The parameters stored in the XML file have no specific units, such as meters or centimeters, limiting the possibility of directly relating creation of real-life robots based on pure values. However, scale and mass can be set on all objects, so if the user makes all objects in the scenario relative to each other, with the correct friction coefficients etc. It should be easy to recreate the virtual robot as a physical 3D-printed model. This will of course be based on the actual size of the Frame meshes and sensor modules sizes. (REQ-1.1.4R)	

Table 81: R-1.1.1

1.2.1.2. Performance

R-1.2.1	REQ-1.2.6R	REQ-1.2.7R
Pass criteria	Specific robot components featured in the assembly shall all exist in the running physics environment. Information about these shall be retrievable.	
Method	Follow the flow of information in each of the frames, joints, sensor modules, force sensors and distance sensors. Ensure all code paths ends in the addition of a component to either the AgX simulation instance, or the robot assembly being added to the instance. Ensure all AgX_Interface classes contain correct return functions for object values.	
Result	Verified	
Comment	All scenario objects are added to the Algorix physics environment when their corresponding initialization functions are called. All objects can be removed from the environment by calling the "AgX_Simulation.RemoveSimObjects()" function.	

Table 82: R-1.2.1

1.2.1.3. GUI

R-1.3.2	REQ-1.3.4R
Pass criteria	All functionality contained in the Core Framework is being utilized by the Scene Designer.
Method	Review the functionality of the Scene Designer versus the possibilities in the Core Framework. Can the user do everything that is expected? Can this be done in the Core Framework in the first place?
Result	Verified
Comment	All functionality in the core Framework is used, but only implemented to various degrees. However, it is enough to get verified.

Table 83: R-1.3.2

1.2.1.4. Scenario

R-1.4.1	REQ-1.4.5R
Pass criteria	All the tweakable parameters and values in the Scene Designer shall have a purpose with impact on the scenario, and not be misleading or useless.
Method	Review all changeable values in the Scene Designer and analyze by robot design or functionality whether the values have purpose. Also check if the changed values have impact on the transmitted XML file.
Result	Verified
Comment	Not all input fields have impact on the scenario, but they all perform functions relating to either the scenario, dynamics, optimization or visualizations.
Discussion	Some of the parameters could be hidden behind other options or sub-containers.

Table 84: R-1.4.1

1.2.2. Core Framework

1.2.2.1. Robot

R-2.1.1	REQ-2.1.4R
Pass criteria	The robot in the scene has no components other than those defined in the robot class and its sub-components, except for any movement-controlling components.
Method	Review the robot class and its sub-components. Ensure all sub-components exist in the Robot class.
Result	Verified
Comment	

Table 85: R-2.1.1

1.2.2.2. Module

R-2.2.1	REQ-2.4.3R
Pass criteria	There are no limits to the number of modules in a robot.
Method	Review the class structure of Module objects.
Result	Verified
Comment	A list contains the modules in the robot, and lists can be theoretically infinite. No errors occur when adding a large amount of modules.

Table 86: R-2.2.1

1.2.3. Dynamics

R-3.1.1	REQ-3.1.2R
Pass criteria	The dynamics component is separate from the Core Framework
Method	Define the relationship between the Core Framework and the Dynamics component. Review how the component is connected to the simulator.
Result	Verified
Comment	The Dynamics component only takes a robot object as a parameter (from the Core Framework). The Core Framework has no relation to the Dynamics component. The Dynamics class is connected through the “Main” class, which is the class used to realize all component functionality.

Table 87: R-3.1.1

R-3.1.2	REQ-3.1.7R
Pass criteria	The user can script robot dynamics.
Method	Review how the user may implement a custom movement script through a string input-field.
Result	N/A
Comment	Functionality not implemented.

Table 88: R-3.1.2

1.2.4. Optimization

R-4.1.1	REQ-4.1.1R
Pass criteria	Optimization component is separate from the Core Framework.
Method	Define the relationship between the Core Framework and the Optimization component. Review how the component is connected to the simulator.
Result	Verified
Comment	The Optimization component only takes a robot object from the Core Framework. The Core Framework has no relation to the Optimization component. The Optimization class is connected through the “Main” class, which is the class used to realize all component functionality.

Table 89: R-4.1.1

R-4.1.2	REQ-4.1.2R
Pass criteria	Optimization component consists of more than one optimization algorithm.
Method	Define amount of optimization algorithms available. Test all optimization algorithms.
Result	N/A
Comment	Only a Genetic Algorithm is currently included in the Optimization component.
Discussion	Algorithms such as simulated annealing could be added easily, thanks to the similarities in execution operations such as iterative performance, goal functions and hierarchical fitness tracking.

Table 90: R-4.1.2

1.2.5. Component-based design

1.2.5.1. Cost reduction

R-5.1.1	REQ-5.1.1R
Pass criteria	Future creation of simulators is faster and cheaper to develop thanks to this framework
Method	Review the usage of the Core Framework and the other components.
Result	Verified
Comment	The Core Framework in addition to the accompanying components (Dynamics, Optimization, Visualization), can be re-used, and developers won't have to invest as much time just to create a framework for robot creation. There are aspects of the framework which may be improved (such as adding more joint types), but improvements as such will still be less effort than to redesign the entire framework. Because of the component-based design, new components can also be added, based on the developer's needs. As this verification specification is completed, future developers may review it to see if it contains the desired functionality for their projects.

Table 91: R-5.1.1

R-5.1.2	REQ-5.1.2R
Pass criteria	Time used on modification is shorter with components.
Method	Review whether less time will be used on modifications and additions to a modular robot simulator, by modifying components instead of whole traditional systems.
Result	Verified
Comment	Throughout the development process and in test scenarios, all modifications have been performed on the specific components only. If a new feature is added to the dynamics class, this is the only place where modification is performed. This has reduced development time and error buildup. It is highly likely that such simple development and maintainability functionality would not be present in a traditional system.

Table 92: R-5.1.2

1.2.5.2. Ease of assembly

R-5.2.1	REQ-5.2.3R
Pass criteria	CBD makes it easier to assemble a robot in various configurations.
Method	Review design choices in the CBD and assess whether robot creation has been facilitated versus using traditional design.
Result	N/A
Comment	There is no clear result.
Discussion	It is assumed that separating all robot-related classes (in the Core Framework) have made flexibility and observability easier. However, without a case-study on a traditional system to compare with, this statement cannot be backed up.

Table 93: R-5.2.1

1.2.5.3. Customization flexibility

R-5.3.1	REQ-5.4.1R	REQ-5.4.2R
Pass criteria	New components can be added to the Core Framework or the Modular Robot Simulator. Modification of components is the only necessity for changing system features.	
Method	Review the method for adding components to the system. Is it easy to just replace? Could it be easier? Can modifications be performed purely inside the specific components and thus change the functionality, or must modifications be performed also in the main class?	
Result	Partly verified	
Comment	There could be less code which assembles components in the “Main” class, but in order to not rely on visualization-specific functions, the components cannot contain update-loops related to the visualization platform. Thus, the “Main” class must contain this functionality. The “Main” class is the only class that needs modification when adding a component. Only the current component needs to be changed when modifying functionality or the component itself.	

Table 94: R-5.3.1

1.2.5.4. Maintainability

R-5.4.1	REQ-5.5.1R
Pass criteria	Code maintenance requires less effort in this component-based system than if it was traditionally designed.
Method	Review components. Are there many overlaps between classes? The more separated the functionality is, the less maintenance is required, and the easier it is to find bugs. How has bug-testing been solved during the system design?
Result	Verified
Comment	See R-5.1.2.

Table 95: R-5.4.1

1.3. Use-cases and comparisons

1.3.1. Simulator usage

1.3.1.1. Scenario

All scenario comparisons have been discussed with the supervisor from NTNU.

U-1.1.1	REQ-1.4.1U	REQ-1.4.2U
Pass criteria	Scenario is easier to create in this program than making a hard-coded simulation (both in easy and advanced mode). The simulator is intuitive to use.	
Method	Compare the Modular Robot Simulator with NTNU's method for creating simulations. Review the time and effort required to create a scenario, both from scratch and by modifying an existing simulation. Discuss the intuitiveness of the simulator.	
Result	Verified	
Comment	The simulator fulfills all of the necessary functionality while being easier to use than manual-coding it or modifying existing software.	

Table 96: U-1.1.1

U-1.1.2	REQ-1.4.3U	REQ-2.2.1U	REQ-2.3.1U	REQ-2.4.5U	REQ-2.5.8U	REQ-2.5.9U	REQ-2.6.1U
Pass criteria	Customization options are similar or better compared to hard-coded simulations, by using the advanced mode in robot creation.						
Method	Compare the Modular Robot Simulator with NTNU's method for creating robot assemblies. See if all required parameters are included, and if the customization options are better, worse or has exactly the required customizability. This will be performed by using the advanced design mode. Also analyze the Frame, Joint, Module, Sensor module, Sensors and Scene classes, to see if the Core Framework has enough customizability.						
Result	Partly verified						
Comment	Sensor attachment less customizable than hard-coding it. Distance sensor is currently not sensing terrain. Customization of joints, optimization and dynamics is enough for teaching, but lacks a little for research purposes. Should have more modularity regarding custom dynamics. Terrain and scene object creation has enough customizability for teaching and research purposes.						
Discussion	Sensor attachment is enough for teaching purposes. Currently, researchers are designing flat ground. Thus, the distance sensor will still work in the scenarios researchers use when developing.						

Table 97: U-1.1.2

U-1.1.3	REQ-1.4.4U
Pass criteria	Designing a robot is faster using the Modular Robot Simulator than with hard-coded simulations.
Method	Compare the Modular Robot Simulator with NTNU's method for creating simulations. Is it faster? Review and discuss the use and advantages of speed vs. customizability.
Result	Verified
Comment	Both the easy and advanced options provide faster results than the current methods used. The time it takes to design various modular robots have been significantly improved. There is a high level of customizability even though the design has been made significantly easier.

Table 98: U-1.1.3

1.3.2. Component-Based Design

1.3.2.1. Ease of assembly

U-2.1.1	REQ-5.2.1U
Pass criteria	Assembling a program is easier than to create it from scratch
Method	Use the framework to create a new project with a basic visualization. Review whether a designer will find it easier to use pre-made components and whether these components are usable enough, or if it would be easier to just create a new solution from scratch.
Result	Verified
Comment	Created in a new Unity project. See framework case-study.

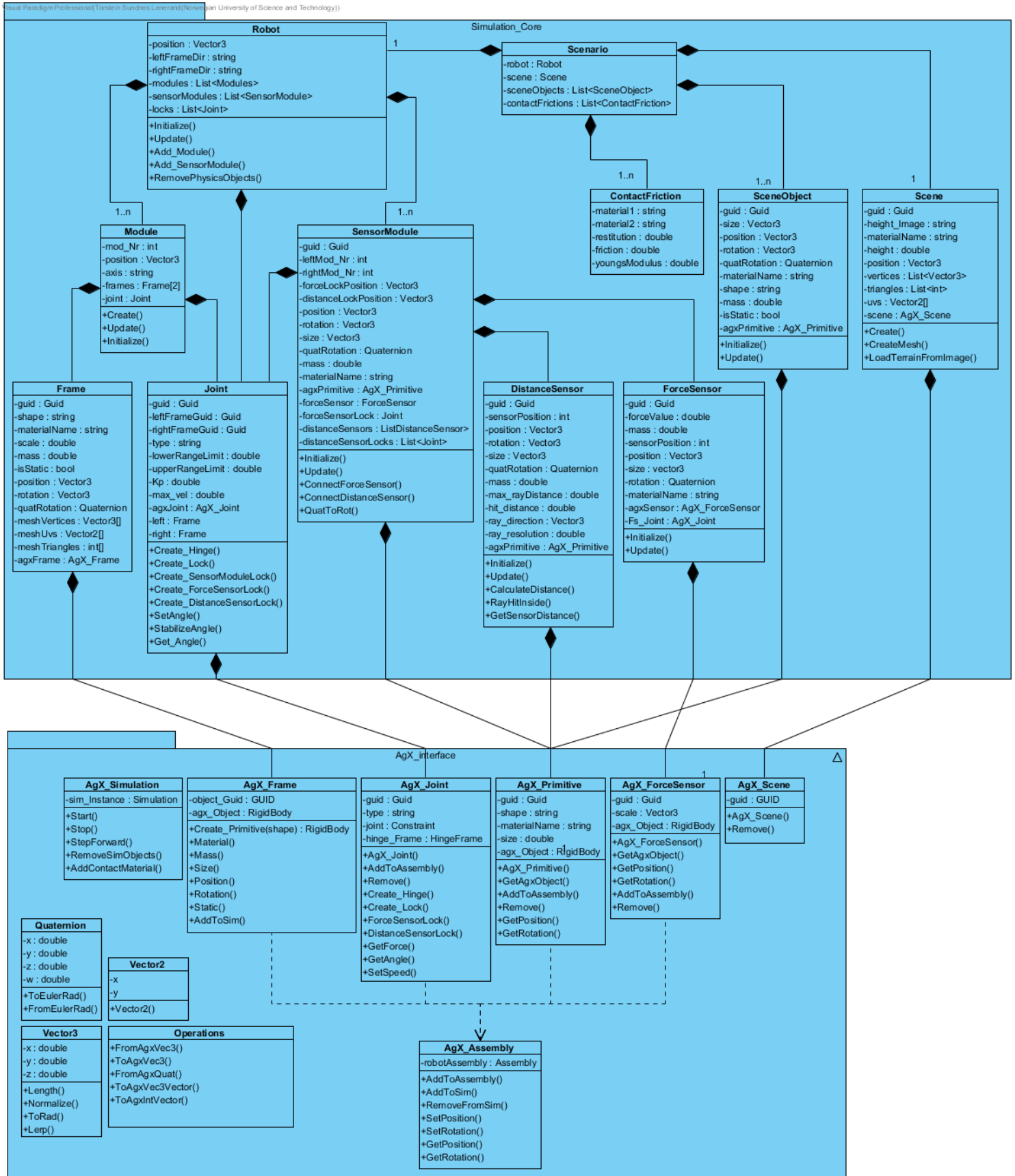
Table 99: U-2.1.1

1.3.2.2. Reusability

U-2.2.1	REQ-5.3.1U	REQ-5.3.2U	REQ-5.3.3U	REQ-5.3.4U	REQ-5.3.5U
Pass criteria	The Core Framework can be used to create a new project. The Dynamics component can be added to the project. The Optimization component can be added to the project. Primitive shapes can be created with the framework, resulting in a minimalistic physics simulator, with no other simulation elements. Sub-components of a robot can be included in the simulation without a robot.				
Method	Create a new project using the Core Framework. Test functionality/vis. Add Dynamics and optimization components. Test. Create simulation only with primitive shapes. Create a frame. Add to simulation. Test.				
Result	Partly verified				
Comment	Created in a new Unity project. See framework case-study in the result section of the report. Dynamics (REQ-5.3.2U) and Optimization (REQ-5.3.3U) not yet tested. REQ-5.3.5U not tested.				

Table 100: U-2.2.1

Appendix D Core Framework overview



Appendix E Remaining technical specification

Simulation_Core SensorModule

The “SensorModule” class contains modules that are connected between the Frame/Joint modules and can be outfitted with distance and force sensors to receive input from the surrounding area. The class is one of the most comprehensive ones, as it has specific functions for sensory attachments allowing for 1 force sensor, and up to 6 distance sensors. The module is a simple cube shape.

Contents of the “SensorModule” class are displayed in Table 101, while the main functions of the class are described in Table 102.

Attributes		Components	
guid : Guid leftMod_Nr, rightMod_Nr : int forceLockPosition : Vector3 distanceLockPosition : Vector3 position, rotation, size : Vector3 quatRotation : Quaternion mass : double materialName : string		agxPrimitive : AgX_Primitive forceSensor : ForceSensor forceSensorLock : Joint distanceSensors : List<DistanceSensor> distanceSensorLocks : List<Joint>	
Functions			
Initialize()	ConnectForceSensor()	QuatToRot()	
Update()	ConnectDistanceSensor()		

Table 101: Contents of the SensorModule class

Function	Description	Return
Initialize	Creates the “AgX_Primitive” object with the attributes of this class. If a force sensor is attached, the force sensor’s initialization function is called, and the corresponding lock is attached. If one of more distance sensors are attached, the initialization functions of these are called, and corresponding locks are attached.	void
Update	Updates the position and rotation of the object based on the corresponding values of the object in the Algoryx simulation instance. Also updates the force and distance sensor attached to the module.	void
ConnectForceSensor	Attaches an input “ForceSensor” to this sensor module and creates a new “Joint” as the sensor lock component	void

	Sets up the sensor and lock position based on the attribute “sensorPosition” in the “ForceSensor”.	
ConnectDistanceSensor	Attaches a list of “DistanceSensors” to the sensor module and creates a new list of “Joints” as distance sensor locks. Attaches each sensor at a position according to the “sensorPosition” attribute in the “DistanceSensor”.	void
QuatToRot	Updates the Euler angle representation of the sensor module’s rotation. Returns the Euler angle representation of the sensor module’s rotation.	Vector3

Table 102: SensorModule class functions

ForceSensor

The “ForceSensor” class defines the force sensors which may be attached to the sensor modules. The force calculations are not performed by this class, but in the “AgX_ForceSensor” class. The force sensor may only be attached to sensor modules, and in four different places: bottom, left, top and right.

Contents of the “ForceSensor” class are displayed in Table 103, while the main functions of the class are described in Table 104.

Attributes	Components
guid : Guid forceValue, mass : double sensorPosition : int position, size : Vector3 rotation : Quaternion materialName : string	agxSensor : AgX_ForceSensor Fs_Joint : AgX_Joint
Functions	
Initialize()	Update()

Table 103: Contents of the ForceSensor class

Function	Description	Return
Initialize	Creates the “AgX_ForceSensor” object with the attributes of this class.	void
Update	Updates the position, rotation and force value of the object based on the corresponding values of the object in the Algoryx simulation instance.	void

Table 104: ForceSensor class functions

DistanceSensor

The “DistanceSensor” class defines the distance sensors which may be attached to the sensor modules. The functionality of the actual sensor resides solely in this class, while the “AgX_DistanceSensor”-object only keeps track of the sensor’s position and rotation on the robot. This class has two main functions for the distance calculations, with one creating a ray, which is sent out by a set distance and angle based on the requested resolution and direction of the sensor. If the ray hits an object, the distance is saved in the class variable “hit_distance”, which will equal to the “max_rayDistance” value if no object is intersecting the ray.

Contents of the “DistanceSensor” class are displayed in Table 105, while the main functions of the class are described in Table 106.

Attributes		Components
guid : Guid sensorPosition : int position, rotation : Vector3 size : Vector3 quatRotation : Quaternion mass : double	max_rayDistance : double hit_distance : double ray_direction : Vector3 ray_Resolution : double	agxPrimitive : AgX_Primitive
Functions		
Initialize()	CalculateDistance()	GetSensorDistance()
Update()	RayHitInside()	

Table 105: Contents of the DistanceSensor class

Function	Description	Return
Initialize	Creates the “AgX_Primitive” object with the attributes of this class.	void
Update	Updates the position and rotation of the object based on the corresponding values of the object in the Algoryx simulation instance.	void
CalculateDistance	Shoots out a ray in the sensor direction. Checks if any scene objects are hit using the RayHitInside function. Retrieves distance of the first object hit within the max distance value.	void
RayHitInside	Checks if the ray intersects any of the scene objects. Checks for box and sphere collisions.	Boolean
GetSensorDistance	Returns the “hit_distance” value.	double

Table 106: DistanceSensor class functions

ContactFriction

The “ContactFriction” class contains all information about the friction coefficients between two individual materials. In each scenario there can be a near infinite number of these objects. The class contains no functions, as the contact frictions are added directly to the static “AgX_Simulation” class.

The attribute variables consist of two materials between which the friction coefficient will be calculated each time the corresponding objects interact in the simulation.

The restitution coefficient contains the ratio between the final to initial velocity of the objects after collision, the friction coefficient contains the resistance force of sliding motions, and the Young’s modulus is the relationship of stress (or deformity) in the contact points. These coefficients are used in the physics calculations of the AgX physics engine after being passed to a function in the “AgX_Simulation” class.

Contents of the “ContactFriction” class are displayed in Table 107.

Attributes	Components
material1 : string material2 : string restitution : double friction : double youngsModulus : double	none
Functions	none

Table 107: Contents of the ContactFriction class

AgX_Interface

AgX_ForceSensor

The “AgX_ForceSensor” class contains the information about a force sensor that may be added to a robot’s sensor module (“SensorModule” class). The module receives the force magnitude from the “Joint” object connecting the force sensor’s rigid-body with the sensor module rigid-body.

Contents of the “AgX_ForceSensor” class are displayed in Table 108, while the main functions of the class are described in Table 109.

Attributes		Components	
guid : Guid scale : Vector3		agx_Object : agx.RigidBody	
Functions			
AgX_ForceSensor()	GetPosition()	AddToAssembly()	
GetAgxObject()	GetRotation()	Remove()	

Table 108: Contents of the AgX_ForceSensor class

Function	Description	Return
AgX_ForceSensor	Creates the Algoryx rigid-body with material, position, rotation, scale, and mass values for the input variables. Adds the object to the simulation instance.	AgX_ForceSensor
GetAgxObject	Retrieves the Algoryx object	agx.RigidBody
GetPosition	Retrieves the position of the object in the simulation	Vector3
GetRotation	Retrieves the Quaternion rotation of the object in the simulation	Quaternion
AddToAssembly	Adds the rigid-body to the current robot-assembly (if necessary)	void
Remove	Removes the rigid-body from the simulation instance.	void

Table 109: AgX_ForceSensor class functions

Operations

Since the AgX library has other variable types than Unity, the “Operations” class is created to transform variables between AgX and Unity. These variables include vectors and quaternions, which must be decomposed into their individual axis-values before constructing the resulting output type.

Table 110 shows the operation functions.

Function	Description	Return
FromAgxVec3	Converts from agx.Vec3 to Vector3	Vector3
ToAgxVec3	Converts from Vector3 to agx.Vec3	agx.Vec3
FromAgxQuat	Converts from agx.Quat to Quaternion	Quaternion
ToAgxQuat	Converts from Quaternion to agx.Quat	agx.Quat
ToAgxVec3Vector	Converts from a Vector3 array to an agx.Vec3Vector	agx.Vec3Vector
ToAgxIntVector	Converts from an int array to an agx.UInt32Vector	agx.UInt32Vector

Table 110: Operations class functions

Types

As detailed in REQ-5.3.1U, the core architecture should be independent from any specific platform and usable in other projects. Thus, the inclusion of Unity3D-based vector and quaternion data types are not valid for use. Three distinct structures have been created to solve this problem, Vector2, Vector3 and Quaternion, with inspiration from the functionality of the corresponding Unity3D classes.

Vector2/Vector3

In the Vector3 structure, there are functions that perform certain operations on the variables such as retrieving the length of a vector, converting from degrees to radians, and interpolations. Operator overloads have also been created. The Vector2 structure contains two variables, “x” and “y” for a 2D representation, and the Vector3 variable contains an additional “z” variable for a 3D representation.

Table 111 shows the functions within the Vector3 structure (The Vector2 structure does not contain any custom functions).

Function	Description	Return
Length	Retrieves the length of the vector	Double
Normalize	Retrieves the direction of the vector with a length of 1	Vector3
ToRad	Transforms the vector from degrees to radians	Vector3
Lerp	Linearly interpolates between two vectors by a set amount	Vector3
Operators	Allows for addition, subtraction, multiplication and division between two vectors or a vector and a double	Vector3

Table 111: Vector3 structure functions

Quaternion:

Functions in the Quaternion structure have been influenced by Unity3D’s Quaternion class, but also general matrix calculus for rotational matrices. The functions perform Quaternion to vector operations and vice versa, in addition to some helper functions for the conversions. There is one operator overload to enable multiplication between a Quaternion and a Vector3. The Quaternion structure contains an “x”, “y”, “z”, and “w” variable, as it is a rotation matrix.

Table 112 shows the functions within the Quaternion structure.

Function	Description	Return
ToEulerRad	Converts Quaternion to Vector3 in radians, and uses two helper functions for the conversion	Vector3
FromEulerRad	Converts from Vector3 Euler angles to Quaternion	
Operator*	Multiplies a Vector3 with a Quaternion to modify the angles	Vector3

Table 112: Quaternion structure functions

Appendix F Case-study, XML file

Scenario, start of Robot, Module, Frames, Joints:

```
<robot>
  <modules xmlns="Assembly">
    <Module>
      <mod_Nr>0</mod_Nr>
      <position>
        <x>0</x>
        <y>12</y>
        <z>-0.48999998755753</z>
      </position>
      <axis>Pitch</axis>
      <frames>
        <Frame>
          <guid>5f9f543f-fc55-49ec-9c97-9944e0264b4b</guid>
          <shape>Box</shape>
          <scale>10</scale>
          <position>
            <x>0</x>
            <y>12</y>
            <z>-0.48999998755753</z>
          </position>
          <rotation>
            <x>0</x>
            <y>270.00000196115104</y>
            <z>0</z>
          </rotation>
          <quatRotation>
            <x>0</x>
            <y>-0.70710676908493042</y>
            <z>0</z>
            <w>0.70710676908493042</w>
          </quatRotation>
          <mass>100</mass>
          <isStatic>>false</isStatic>
          <materialName>Plastic</materialName>
        </Frame>
        <Frame>
          <guid>fe7a5aba-49bf-441b-8f4c-76089bb02d81</guid>
          <shape>Box</shape>
          <scale>10</scale>
          <position>
            <x>0</x>
            <y>12</y>
            <z>-0.48999998755753</z>
          </position>
          <rotation>
            <x>0</x>
            <y>0</y>
            <z>0</z>
          </rotation>
          <quatRotation>
            <x>0</x>
            <y>-0.70710676908493042</y>
            <z>0</z>
            <w>0.70710676908493042</w>
          </quatRotation>
          <mass>100</mass>
          <isStatic>>false</isStatic>
          <materialName>Plastic</materialName>
        </Frame>
      </frames>
      <joint>
        <guid>ead26d3b-e615-4e80-9464-a5e429e0ed02</guid>
        <leftFrameGuid>5f9f543f-fc55-49ec-9c97-9944e0264b4b</leftFrameGuid>
        <rightFrameGuid>fe7a5aba-49bf-441b-8f4c-76089bb02d81</rightFrameGuid>
        <type>Hinge</type>
        <lowerRangeLimit>-1.5707963267948966</lowerRangeLimit>
        <upperRangeLimit>1.5707963267948966</upperRangeLimit>
        <Kp>8</Kp>
        <max_vel>3</max_vel>
      </joint>
    </Module>
  </modules>
</robot>
```

Sensor module, force sensor, distance sensor, locks:

```
<sensorModules xmlns="Assembly">
  <SensorModule>
    <guid>49c95067-af5e-49cc-9f24-d523666bf3d6</guid>
    <leftMod_Nr>-1</leftMod_Nr>
    <rightMod_Nr>0</rightMod_Nr>
    <forceSensor>
      <guid>2a343a24-a8c9-4ac6-8b5e-638fd98647b3</guid>
      <forceValue>0</forceValue>
      <sensorPosition>0</sensorPosition>
      <position>
        <x>0</x>
        <y>11.6985</y>
        <z>0</z>
      </position>
      <rotation>
        <x>0</x>
        <y>0</y>
        <z>0</z>
        <w>1</w>
      </rotation>
      <materialName>Plastic</materialName>
      <mass>1</mass>
      <size>
        <x>0.3</x>
        <y>0.001</y>
        <z>0.1</z>
      </size>
    </forceSensor>
    <forceSensorLock>
      <guid>00000000-0000-0000-0000-000000000000</guid>
      <leftFrameGuid>00000000-0000-0000-0000-000000000000</leftFrameGuid>
      <rightFrameGuid>00000000-0000-0000-0000-000000000000</rightFrameGuid>
      <lowerRangeLimit>0</lowerRangeLimit>
      <upperRangeLimit>0</upperRangeLimit>
      <Kp>3</Kp>
      <max_vel>0</max_vel>
    </forceSensorLock>
    <forceLockPosition>
      <x>0</x>
      <y>11.6995</y>
      <z>0</z>
    </forceLockPosition>
    <distanceSensors>
      <DistanceSensor>
        <guid>e2e792af-bdaa-41ab-b851-541220f94b7b</guid>
        <sensorPosition>4</sensorPosition>
        <position>
          <x>0</x>
          <y>12</y>
          <z>0</z>
        </position>
        <rotation>
          <x>0</x>
          <y>0</y>
          <z>0</z>
        </rotation>
        <quatRotation>
          <x>0</x>
          <y>0</y>
          <z>0</z>
          <w>0</w>
        </quatRotation>
        <mass>1</mass>
        <size>
          <x>0.01</x>
          <y>0.01</y>
          <z>0.01</z>
        </size>
        <max_rayDistance>10</max_rayDistance>
        <ray_Direction>
          <x>0</x>
          <y>0</y>
          <z>1</z>
        </ray_Direction>
        <ray_Resolution>0.05</ray_Resolution>
      </DistanceSensor>
    </distanceSensors>
    <distanceLockPosition>
      <x>0</x>
      <y>12</y>
    </distanceLockPosition>
  </SensorModule>
</sensorModules>
```

```

    <z>0</z>
  </distanceLockPosition>
  <distanceSensorLocks>
    <Joint>
      <guid>00000000-0000-0000-0000-000000000000</guid>
      <leftFrameGuid>00000000-0000-0000-0000-000000000000</leftFrameGuid>
      <rightFrameGuid>00000000-0000-0000-0000-000000000000</rightFrameGuid>
      <lowerRangeLimit>0</lowerRangeLimit>
      <upperRangeLimit>0</upperRangeLimit>
      <Kp>3</Kp>
      <max_vel>0</max_vel>
    </Joint>
  </distanceSensorLocks>
  <position>
    <x>0</x>
    <y>12</y>
    <z>0</z>
  </position>
  <rotation>
    <x>0</x>
    <y>0</y>
    <z>0</z>
  </rotation>
  <size>
    <x>0.3</x>
    <y>0.3</y>
    <z>0.1</z>
  </size>
  <quatRotation>
    <x>0</x>
    <y>0</y>
    <z>0</z>
    <w>1</w>
  </quatRotation>
  <mass>100</mass>
  <materialName>Plastic</materialName>
</SensorModule>

```

Robot Locks (between modules)

```

<locks xmlns="Assembly">
  <Joint>
    <guid>00000000-0000-0000-0000-000000000000</guid>
    <leftFrameGuid>00000000-0000-0000-0000-000000000000</leftFrameGuid>
    <rightFrameGuid>00000000-0000-0000-0000-000000000000</rightFrameGuid>
    <lowerRangeLimit>0</lowerRangeLimit>
    <upperRangeLimit>0</upperRangeLimit>
    <Kp>0</Kp>
    <max_vel>0</max_vel>
  </Joint>

```

Scene

Height image string is 21848 characters long.

```

<scene>
  <guid>b2135ad0-9d9a-4c63-b341-09d001be0cad</guid>
  <height_Image>1VBORw0KGgoAAAANSUHEUgAAAPsAAAD7CAYAAACscuKmAAAgAE1EQVR4Ae3dV491y1E38H2Cc45g4hwTDDY4gEHGGCxx+wIu4I:
  <vertices />
  <triangles />
  <position>
    <x>-125</x>
    <y>0</y>
    <z>-125</z>
  </position>
  <materialName>Rock</materialName>
  <height>10</height>
</scene>

```

Scene Objects

```
<sceneObjects>
  <SceneObject>
    <guid>7b579252-58ce-4fb0-937f-57c24696363f</guid>
    <size>
      <x>1</x>
      <y>1</y>
      <z>1</z>
    </size>
    <position>
      <x>0</x>
      <y>12</y>
      <z>4</z>
    </position>
    <rotation>
      <x>0</x>
      <y>0</y>
      <z>0</z>
    </rotation>
    <quatRotation>
      <x>0</x>
      <y>0</y>
      <z>0</z>
      <w>1</w>
    </quatRotation>
    <materialName>Plastic</materialName>
    <shape>Sphere</shape>
    <mass>10</mass>
    <isStatic>false</isStatic>
  </SceneObject>
</sceneObjects>
```

Appendix F1 Case-study, visualization and update

Update loop:

```
//vis
GameObject sceneobjvis;
GameObject[] FrameDemoVis = new GameObject[2];

// Update is called once per frame
void Update_Sim ()
{
    //Update physics:
    Agx_Simulation.StepForward();
    //Update the robot:
    robot.Update();
    //Update the visualization:
    {
        //Frames:
        FrameDemoVis[0].transform.position = AgxHelper(robot.modules[0].frames[0].position);
        FrameDemoVis[0].transform.rotation = (AgxHelper(robot.modules[0].frames[0].quatRotation));
        FrameDemoVis[1].transform.position = AgxHelper(robot.modules[0].frames[1].position);
        FrameDemoVis[1].transform.rotation = (AgxHelper(robot.modules[0].frames[1].quatRotation));
        //Scene object:
        sceneobjvis.transform.position = AgxHelper(sceneobj.position);
    }
    robot.modules[0].joint.SetAngle(2);
}
```

Visualization:

```
Mesh leftmesh, rightmesh;
void loadmesh()
{
    ObjImporter import = new ObjImporter();
    leftmesh = import.ImportFile(Application.streamingAssetsPath + "/upper.obj");
    rightmesh = import.ImportFile(Application.streamingAssetsPath + "/bottom.obj");
}

void Load_Vis()
{
    //Set frame meshes:
    FrameDemoVis[0] = GameObject.CreatePrimitive(PrimitiveType.Cube);
    FrameDemoVis[1] = GameObject.CreatePrimitive(PrimitiveType.Cube);

    Mesh[] mesh = new Mesh[2];
    mesh[0] = FrameDemoVis[0].GetComponent<MeshFilter>().mesh;
    mesh[1] = FrameDemoVis[1].GetComponent<MeshFilter>().mesh;

    var robotsize = new UnityEngine.Vector3(10,10,10);
    for(int i = 0; i<2;i++)
    {
        mesh[i].vertices = AgxHelper(robot.modules[0].frames[i].meshVertices);
        mesh[i].uv = AgxHelper(robot.modules[0].frames[i].meshUvs);
        mesh[i].triangles = robot.modules[0].frames[i].meshTriangles;
    }

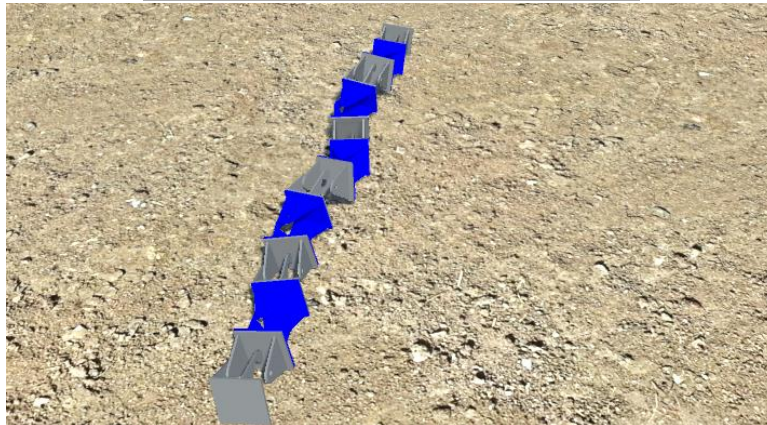
    UnityEngine.Vector3 AgxHelper(AgX_Interface.Vector3 vec)
    {
        var vector = new UnityEngine.Vector3();
        vector.x = (float)vec.x;
        vector.y = (float)vec.y;
        vector.z = (float)vec.z;

        return vector;
    }
}
```

Appendix G Custom dynamics case study

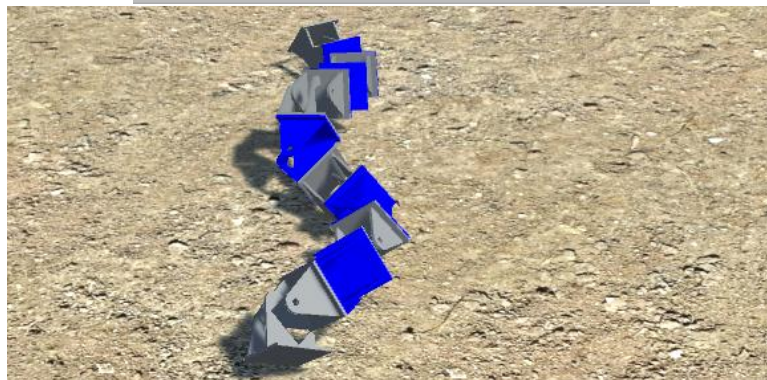
Sidewinding 1

P	Y	P	Y	Period	P	Y
Amplitude	Amplitude	PhsOffset	PhsOffset		stdOffset	stdOffset
0.5	0.5	2.5	2.5	4	0	0



Sidewinding 2

P	Y	P	Y	Period	P	Y
Amplitude	Amplitude	PhsOffset	PhsOffset		stdOffset	stdOffset
1	1	2	2	4	0	0



Rolling

P	Y	P	Y	Period	P	Y
Amplitude	Amplitude	PhsOffset	PhsOffset		stdOffset	stdOffset
.5	.5	1	.5	4	0	0

