Distributed algorithms for programmable matter : target shape description and self-assembly planning

Thadeu Knychala Tucci

▶ To cite this version:

Thadeu Knychala Tucci. Distributed algorithms for programmable matter : target shape description and self-assembly planning. Data Structures and Algorithms [cs.DS]. Université Bourgogne Franche-Comté, 2018. English. NNT : 2018UBFCD028 . tel-02927207

HAL Id: tel-02927207
https://theses.hal.science/tel-02927207

Submitted on 1 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

UFC
UNIVERSITÉ
DE FRANCHE-COMTÉ

DOCTORAL THESIS OF THE INSTITUTION UNIVERSITY BOURGOGNE FRANCHE-COMTÉ

PREPARED AT FRANCHE-COMTÉ UNIVERSITY

Doctoral school n°37

Engineering Sciences and Microtechnologies

Computing Dissertation

by

THADEU KNYCHALA TUCCI

Distributed Algorithms for Programmable Matter: Target Shape Description and Self-Assembly Planning

Thesis presented and publicly defended in Montbéliard, on November 12, 2018

Composition du Jury :

GLEIZES MARIE-PIERRE	Professor at the University of Toulouse	Reviewer
SIMONIN OLIVIER	Professor at the University of Lyon	Reviewer
PICARD GAUTHIER	Professor at the Ecole Nationale Supérieure des Mines of Saint-Etienne	Examiner
BOURGEOIS JULIEN	Professor at the University of Bourgogne Franche-Comté	Supervisor
PIRANDA BENOÎT	Associate Professor at the University of Bourgogne Franche-Comté	Co-supervisor

N° X X X X

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

UFC
UNIVERSITÉ
DE FRANCHE-COMTÉ

THÈSE DE DOCTORAT DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

École doctorale n°37

Sciences Pour l'Ingénieur et Microtechniques

Doctorat d'Informatique

par

THADEU KNYCHALA TUCCI

Algorithmes Distribués pour la Matière Programmable: Description de la Forme Cible et Planification de l'Auto-Assemblage

Thèse présentée et soutenue à Montbéliard, le 12 novembre 2018

Composition du Jury :

GLEIZES MARIE-PIERRE	Professeur des Universités à l'Université de Toulouse	Rapporteur
SIMONIN OLIVIER	Professeur des Universités à l'Université de Lyon	Rapporteur
PICARD GAUTHIER	Professeur des Universités à l'Ecole Nationale Supérieure des Mines de Saint-Etienne	Examinateur
BOURGEOIS JULIEN	Professeur des Universités à l'Université de Bourgogne Franche-Comté	Directeur de thèse
PIRANDA BENOÎT	Maître de conférences à l'Université de Bourgogne Franche-Comté	Codirecteur de thèse

N° X X X X

Acknowledgements

I would like to thank first my supervisors Benoît Piranda and Julien Bourgeois for their kindness, support and motivation. Our regular meetings was always very fruitful with insights for the progression of my thesis. I had been inspired in their hard work.

During the period of this thesis I was supported by The Brazilian National Council for Scientific and Technological Development, in Portuguese CNPq, with grant 202835/2014-6 that I am thankful.

A special thanks to Anna-Sofia Shloida Tucci, my loved daughter that is the bigger motivation of my life and my work.

I am also thankful to my friends Leonardo Alt and Enrique Fynn, my dear friends and climbing partners who were doing their PhD at Lugano in Switzerland and was always there for constructive comments over my thesis work and overall computer science.

In addition, I would like to thank my friendly office mates, André Naz and Pierre Thalamy for all their jokes and relax time letting me always beat them in table soccer during our coffee breaks.

Furthermore, I would love to thank all the friends I made living in the lovely city of Montbéliard, they were my replacement family during my PhD journey. Many thanks to all of you.

Last but foremost, my sincere thanks to all my family members that always supported me and delightful talks in video conferences even with an ocean that separated us during all these years.

Contents

1 Introduction 1

1.1 Modular Robots . . . 1
1.2 Challenges . . . 3
1.3 Problem Statement . . . 3
1.4 Contributions . . . 4
1.5 Organization of the Dissertation . . . 5

2 Programmable Matter 7

2.1 Modular Robots . . . 8
2.1.1 Architecture . . . 8
2.1.2 Motion . . . 10
2.1.3 Communication . . . 11
2.1.4 Programmable Matter Project . . . 12
2.1.4.1 Blinky Blocks . . . 13
2.1.4.2 Planar Catoms . . . 14
2.1.4.3 2D Catoms . . . 15
2.1.4.4 3D Catoms . . . 16
2.2 Simulators . . . 17
2.2.1 VisibleSim . . . 19
2.2.1.1 Classical Algorithms . . . 21
2.2.2 VisibleSim Online . . . 23

3 Scene Encoding for Programmable Matter 25

3.1 Related Works . . . 26
3.1.1 Bitmap Representation . . . 27
3.1.2 Constructive Solid Geometry . . . 27
3.1.3 Boundary Representation . . . 28
3.1.4 Triangle Mesh . . . 29
3.1.5 Overlapping Bricks . . . 31
3.1.6 Representation by Rules . . . 32

3.1.7 L-System Representation 32
3.2 Contribution 32
3.2.1 Evaluating Coordinate Status in Model 34
3.2.2 How to Encode the Model 35
3.3 Simulation 37
3.3.1 Fidelity to the Original Format 40
3.3.2 Comparison of Code Sizes 41
3.3.3 Decoding Process Time 42
3.4 Conclusion 42

4 Self-Assembly Planning 45

4.1 Introduction 46
4.2 Related Works 48
4.2.1 Message based 48
4.2.2 Centralized Solution 48
4.2.3 Transition Rules 49
4.2.4 Porous Scaffold of Modular Robots 50
4.2.5 Other methods 50
4.3 Introduction to Motion Strategy 51
4.3.1 Stochastic Movements 51
4.3.2 Local Rules 51
4.3.3 Coordinate Attractors 51
4.3.4 Gradient Attractors 51
4.3.5 Recruitment 52
4.4 Model and definition 52
4.4.1 Module background 52
4.4.2 Problem definition 53
4.5 Distributed Self-Assembly Algorithm 54
4.5.1 2D Self-Assembly Algorithm 54
4.6 Experimental Evaluation 61
4.6.1 Messages Evaluation 62
4.6.2 Available Docking Positions Evaluation 63
4.7 3D Self-Assembly 64
4.7.1 3D Self-Assembly Layer by Layer 65
4.7.2 Multilayer 3D Self-Assembly 67

4.7.3 Experimental Evaluation 68
4.7.3.1 Cube 69
4.7.3.2 Sphere 72
4.7.3.3 Cylinder 73
4.7.3.4 Pyramidal Object 73
4.7.3.5 Locomotive 3D 74
4.7.4 3D Self-Assembly with Merging Layers 78
4.8 Conclusion 80

5 Conclusion 83
5.1 Summary 83
5.2 Future Work 84

I Annexes 99

A 2D Self-Assembly Implementation Details 101

B Demonstrations of LMR Network Properties 103

1

Introduction

1.1/ Modular Robots

Some few decades ago, computers barely fit in an entire room and nowadays, thanks to technology evolution, an advanced computer can fit in a pocket. Smartphones and smartwatches are successful examples of this miniaturization present in our daily lives. The world continues to evolve into smaller and faster computer systems. Today, world's smallest computer is the Michigan Micro Mote (M3) (Lee et al. (2013)) and its small scale can be seen in the Figure 1.1. Taking into consideration that miniaturization tends to reduce a computer to a small piece of matter, new applications can be now designed.

At the same time, robots are essential in the development of human production and are increasingly occupying human places in the production chain. But robots as we know them today are not flexible, they are built for a certain specific task in mind, they can break, and they have few interchangeable parts.

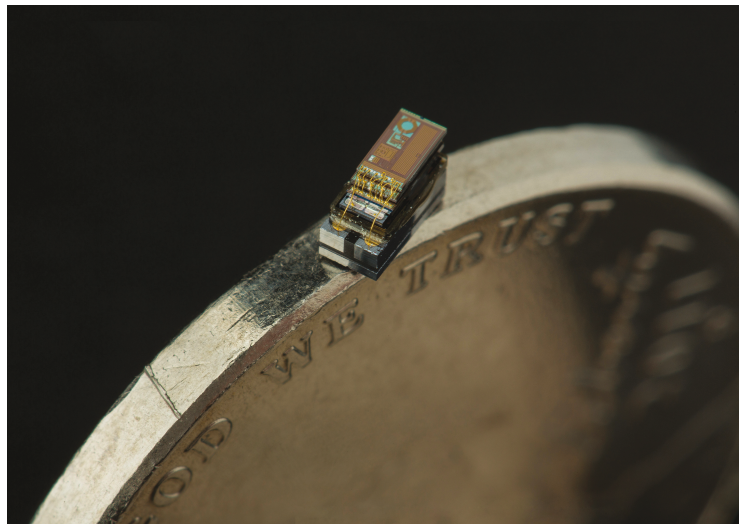


Figure 1.1: World's actual smallest computer.

Summing up the small scale devices with robots, a new paradigm comes true: modular robots. An entire robot made of many other small modules that would be capable of adapting to different morphologies. A single module has computing capabilities but alone it does not have much function. If there are two modules they can be reorganized in a different structure, but if they are many, they have the potential to create a myriad

of structures, from adding more arms to a robot, to adapting its legs to wheels when necessary, illustrating the potential for modular robots .

A single module is a simple robot that contains a computing unit, sensors, actuators and power. It can communicate with other modules sending messages to synchronize and converge to a common task, its sensors can detect environmental details as its orientation or the presence of another module in any direction, and its actuators are responsible for the movement between a pair of modules. Actual devices using current Microelectromechanical Systems (MEMS) technology can already be very small and consume low energy.

Modular robots make robots, as we are used to, adaptive to fulfill different tasks. One entity of robot can transform from one object to another, creating many possible applications. It can be, for instance, useful to use modular robots to align modules in a snake-shape to pass under a door gap in a rescue mission and to transform into something else on the other side. Also, it can create specific tools easily using the same components or, going beyond this, it can, as one adaptable robot can fulfill several requirements, minimize the number of robots to send in a space mission.

The expected properties of modular robots as pointed by Yim et al. (2002) are: (1) versatility, modules can self-assemble into many morphologies and can be used to fulfill different tasks, (2) robustness, as a faulty module can be replaced by another one, and (3) affordable price, as the mass production of identical modules is likely to reduce the overall cost.

One of the most interesting application of modular robot is the self-reconfiguration, that means, the modules can dynamically change their position over the structure, doing it by themselves with the help of actuators and by this, changing the overall shape of the robot. A definition from Østergaard (2004) exemplifies well the properties necessary for self-reconfigurable robots:

- *Modular:* The robot is built from many physically independents modules
- *Reconfigurable:* The modules can be arranged in different ways producing a multitude of robots in terms of shape and functionality.
- *Dynamically reconfigurable:* The modules are able to connect and disconnect from the overall robot while the robot is active.
- *Self-reconfigurable:* The modules are able to connect and disconnect by themselves.

Robots have always fascinated and fed the imagination of authors and creators. They are present in many movies where they are used to representing a futuristic view of the world, in which robots are capable of a multitude of tasks. Specifically, modular robots can be a solution for these types of robots, and they are already present in many shows, with the fiction movie “Terminator 2” (1991) in which the “Terminator T-1000” is an android that has the capability to transform his body in liquid and to assume the appearance he wants, the series “Transformers” (2007) with the notion of modular robots where, being each module a car, they can dock with others to build a big robot, “Big Hero 6” (2014) that uses swarm robots to design a nano-robotic system that can self-reconfigure into many shapes and objects as walls, bridges and pass through narrow spaces, and also in “Avengers: Infinity War” (2018) where the Iron Man armor is made of modular robots that adapt to his body

shape when activated. It is possible to see some application in these movies, but we can either go further citing cases as for military defense, surgery where small scale robots can find a way into the human body and, once inside, transform to perform any function, and also instant-prototyping to have a touchable representation of an object.

1.2/ Challenges

Self-reconfiguration requires many strategies in order to be fully functional. For instance, planning all the movements in advance for a self-reconfiguration is not feasible in polynomial time, as the number of unique configurations is huge: $(c \times w)^n$ where n is the number of modules, c the number of possible connections per module and w the ways of connecting the modules together as detailed in Park et al. (2008). If modules can move simultaneously, the number of possible configuration grow at the rate of $O(m^n)$ with m the number of possible movements and n the number of modules free to move (Barraquand et al. (1991)).

Ideally, modular robots would make advantage of a decentralized organization. In the nature, it is possible to find examples of coordination by decentralization, one example are bees that converge to a common goal without a centralized order. Determining the rules that coordinate them remains a challenge. Instead of having centralized decisions, as one leader deciding the action of each other module in the system, a decentralized organization where modules make their own choices based on local communication and, at the same time, converging to a common goal, is sought. Also, a decentralized organization could be beneficial for a network with a huge number of modules, as one module could not have enough resources to manage the entire group.

Summarizing, modular robots algorithms would take advantage if they are designed with the following aspects:

- *Decentralization:* A central authority would require information about every start and end of action in the network causing long delays and bottleneck in the communication. The use of a central authority should be minimized when possible.
- *Robustness:* The algorithms should be generic enough to overcome faulty modules. When possible, no particular task should be given to a specific agent.
- *Parallelism:* Modules acting independently and in parallel can optimize the general time to converge to a common goal.
- *Stateless:* Knowing the state of each module in a huge network of modular robot can be an expensive task.

1.3/ Problem Statement

The contributions of this thesis are motivated over the will to have better strategies for self-reconfiguration algorithms.

In addition to path planning, where the optimal path for modules in a robot with many modules may not be feasible, other methods should be employed to find a practical solution.

All these solutions should be aware of some specific characteristics of modular robots, where each module has limited resources, for example memory, avoiding collisions and avoiding hollow configurations where a module could not fit by means of rotation. Distributed algorithms for this specific class of network are required to be developed as they would have a better performance for thousands of modules, having faster decisions constructed locally and avoiding bandwidth, energy and memory waste.

Furthermore, the self-reconfiguration process can be divided in many sub-problems. For modules to identify their final position during a self-reconfiguration they need some additional data as a neighboring recognition, the target configuration and their own relative position in this set. One of these sub-problems we study here is how to store and how to encode the target configuration optimally in order to have a small encoding memory footprint and fast access to the information, considering that single modules have limited memory and resources.

Another self-reconfiguration sub-problem tackled in this thesis is how to avoid modules to get blocked in hollow positions. For example, a path for a module must avoid narrow tunnels as modules need space to conclude their rotation. The same is valid for a position where it should dock, as a narrow space between two different modules cannot allow a module to dock in it due to kinematic restrictions. Many solutions to overcome these sub-problems have been proposed in the past, and, in this thesis, we will review them and propose a new solution, evaluate it using many simulated contexts and show its advantages compared to the others.

1.4/ Contributions

In this thesis, I propose a collection of algorithms that can be used as tools for a range of self-reconfiguration systems. These contributions have been reviewed, published and presented in international conferences.

This thesis makes the following contributions:

- **Target shape description for modular robots systems:** I propose the Constructive Solid Geometry for Programmable Matter (CSG4PM) model, that is a description scene method using a tree which is adapted to the modular robots requirements. It consists in describing a scene using a tree made by boolean operators, as difference, union and intersection, in internal nodes, and simple primitives as cubes, spheres and cylinders in the tree leaves. As the resources of a single module are scarce, CSG4PM is a compact description that, at the same time, offers an efficient way of checking if a position is inside the object or not. CSG4PM is strongly inspired by Construct Solid Geometry, a well-know method in image synthesis domain, proposed in Requicha (1980) and here we present all the pre-processing and runtime algorithms associated. Other methods to represent a target shape are explained in detail as well. An evaluation is presented comparing this method to other existing solutions, and the results indicate that CSG4PM offers a good trade-off between fidelity, code size and decoding time.
- **Distributed self-assembly planning algorithm:** Self-assembly and self-reconfiguration work by moving modules from their initial position to a final position based on the target shape. A first requirement to find this path is identifying the

module final position that will not prevent other modules to dock in the system. If modules dock in random positions, they can create narrow spaces in the structure that makes impossible a module to fit in. An order of docking should be guaranteed to avoid these holes in the final structure. The algorithms proposed generate this order in a distributed fashion, modules can identify using local rules a neighbor position that can be filled and send a signal to attract another module to one of its connectors. At the same time, these algorithms propose high parallelism where many positions could be filled simultaneously. Our observations indicate that the number of messages is highly predictable and linear compared to the number of modules.

1.5/ Organization of the Dissertation

The structure of my thesis is organized as follows:

- **Chapter II** presents a brief introduction to programmable matter and categorizes different modular robots. Then, the distributed programming environment and the modular robots simulator used for the evaluations of this work are presented.
- **Chapter III** contains CSG4PM model, my first contribution to the domain of programmable matter suggesting an efficient method to represent the goal scene.
- **Chapter IV** presents my second contribution that uses a distributed algorithm to discover available positions where modules can dock safely while avoiding any other position to get blocked by kinematic constraints.
- **Chapter V** summarizes the contributions of this thesis and proposes some directions for future works.

2
PROGRAMMABLE MATTER

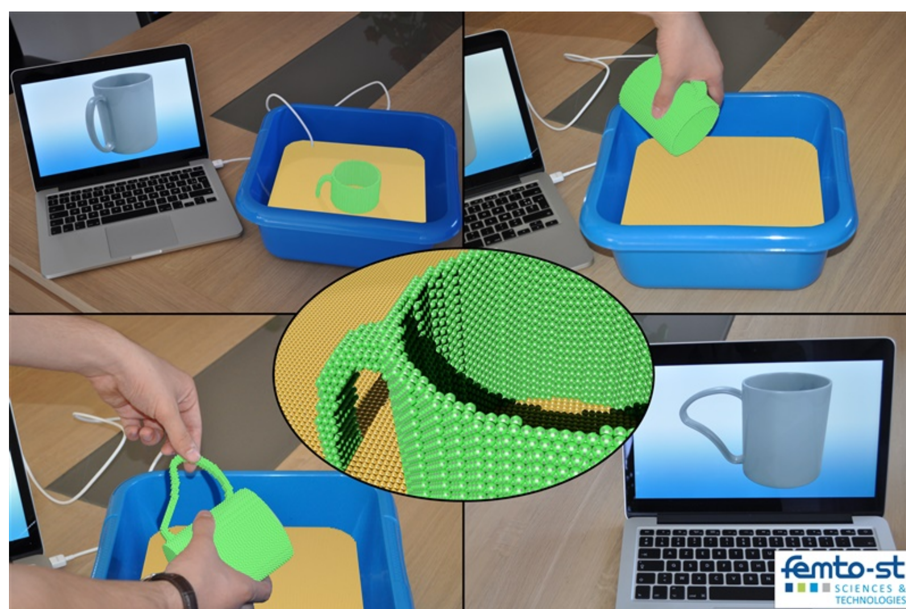


Figure 2.1: Programmable matter applied to create an intelligent mug which can be modeled.

Programmable matter is a concept to model intelligent and adaptive objects from a set of sub-millimeter computing units. It makes possible to create computing 3D models that we can touch and interact with.

The idea of an ensemble of millimeter scale computing elements arranged in space, was first coined in 1991 by Toffoli et al. (1991).

Later in 2002, a project called Claytronics at Carnegie Mellon University introduced the concept of programmable matter in the paper written by Goldstein et al. (2005) as a set of millions of millimeter modular robots with limited computer resources that can be reconfigured to another shape creating a synthetic reality. This project has focused on research and development of both hardware and software for programmable matter since then.

Programmable matter project has been based on The Moore's law, which is an observation made in 1965 which says that, for every two years, the number of transistors in a dense circuit doubles. New observations made by Intel in 2018 for this same law adjust this period to a range of two and half to three years. Anyway, today it is possible to create

milliliter scale circuits and oversee a future with even smaller circuits that make a support for programmable matter. Some other challenges remain as energy transmission, communication, actuators and coordination, but with the advances of the overall technology several research groups have begun to take an interest in this domain.

Programmable matter can be seen as a new way to use 3D printers. Current 3D printers are capable to print static objects that cannot be reused, while programmable matter is capable to print a dynamic object with computing capabilities that can be reused to print another object which also reduces the environmental footprint. Figure 2.1 shows an example of programmable matter use case. In the figure, an object is modeled in a computer and then it is transferred to a bunch of robots that forms the physical object. This matter can be modeled manually and the modifications can be reflected on the computer model. It can act as an interface between real world objects and virtual models.

2.1/ Modular Robots

Modular robots, a type of robot construction composed of many pieces that can be easily interchanged, allow the creation of systems that can be adapted to needs that were not even thought before. One aspect of this adaptability is the capacity to change its own morphology, what have been called self-reconfiguration, creating new features for these robots.

Shape formation can be achieved by different methods using modular robots. Self-assembly is the ability to modules, from an initial position, move by themselves until obtaining a morphology based on the goal shape, while for self-reconfiguration, modules can dynamically change from one goal shape to another. Other existing possibilities for modular robotic systems are manual formation, self-disassembly (Gilpin et al. (2008)) and self-duplication (Gilpin et al. (2012)).

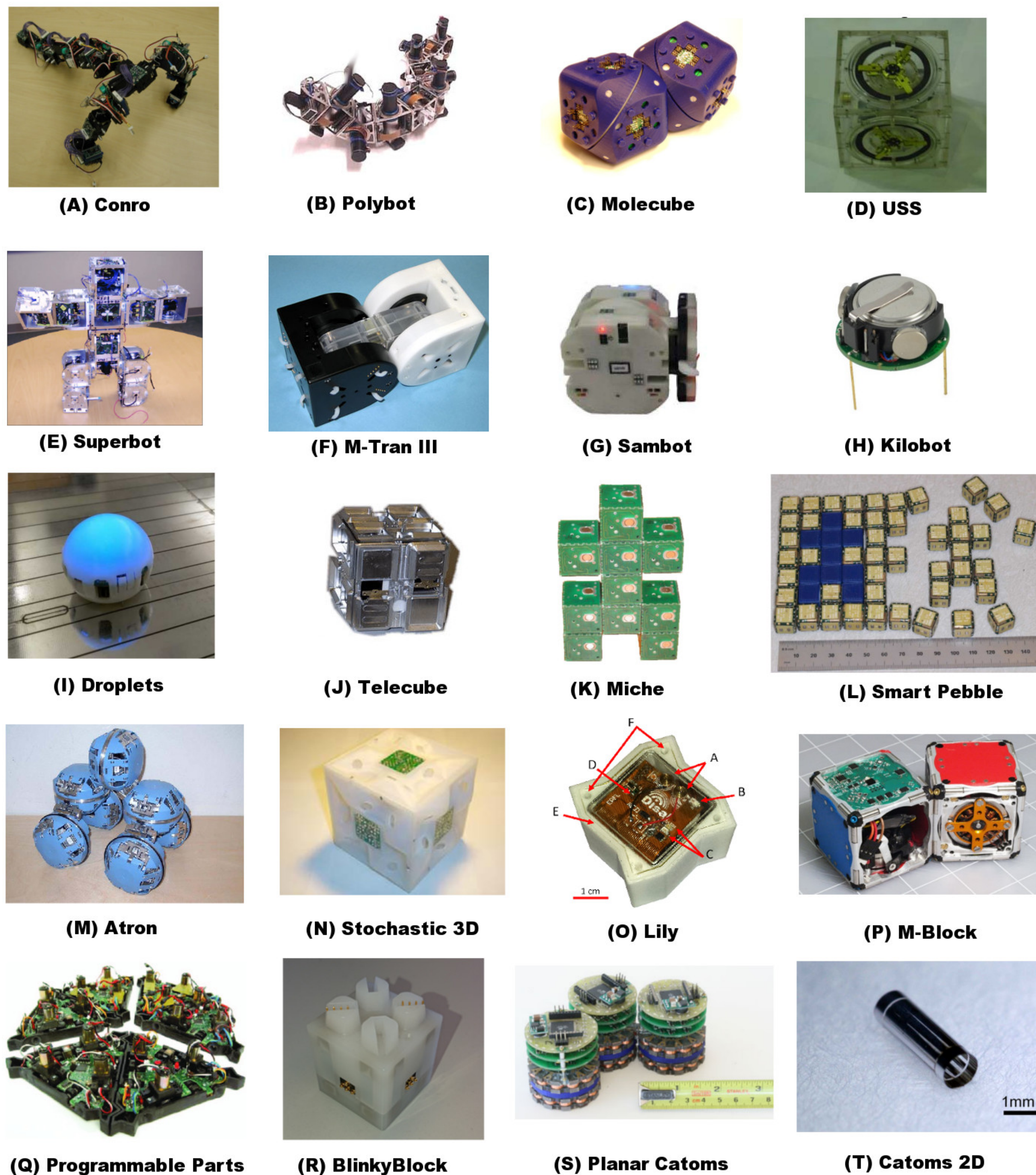
Aside from being able to shape formation, some other types of modular robots with different architecture can be better adapted for tasks as locomotion or balancing. In Table 2.1, many modular robotic systems that can be related with this work are represented and grouped into three general categories: architecture, type of motion, and type of communication. The reader can also check the survey made by Ahmadzadeh et al. (2015) which uses other categories to classify several modular robotic systems.

2.1.1/ Architecture

Architecture for modular robots can be grouped into three different categories which are *Chain*, *Lattice* and *Mobile*. Some implementations of modular robots can fit in more than one architecture, which can be found in the literature under specific names, but here we specify more than one category when it is necessary.

In *Chain* architecture, modular robots are made up of linear, tree or looped connections which form complex structures. They are usually used for locomotion, balancing and supporting, taking advantage of their graph representation. Modular robots are defined with this architecture when modules do not need a regular lattice to align for docking in another place, what makes them able to reach any point in the space.

Lattice architecture consists of modules that are located in discrete positions on a regular



(A) Conro (B) Polybot (C) Molecube (D) USS

(E) Superbot (F) M-Tran III (G) Sambot (H) Kilobot

(I) Droplets (J) Telecube (K) Miche (L) Smart Pebble

(M) Atron (N) Stochastic 3D (O) Lily (P) M-Block

(Q) Programmable Parts (R) BlinkyBlock (S) Planar Catoms (T) Catoms 2D

Figure 2.2: Implementation of Modular robots. Modules **(R)**, **(S)** and **(T)** are part of the Claytronics project.

grid, making use of these positions to align its connectors in order to dock. As modules are connected to each other, it is always possible to define a relative position using integer coordinates. This type of representation may be suitable for shape formation with a big number of modules due to its coordinate system.

Lastly, *Mobile* architecture is composed of robots that can move independently without being attached to other. They are known as swarm robots and for some implementations with docking capability, they can form lattice or chain structures.

Table 2.1: Table comparing different types of hardware specification for modular robot.

Architecture	Movement Type	Module	Reference	Conn. Mechanism	Comm. Type
Chain	Manual	CONRO	Castano et al. (2002)	Magnetic	Infra-Red
	Mechanical	Polybot	Yim et al. (2003)	Mechanical	Contact
		Molecube	Zykov et al. (2007)	Mechanical	Serial
		USS	Wu et al. (2008)	Mechanical	Serial
Chain and Lattice		Superbot	Salemi et al. (2006)	Mechanical	Infra-Red
		M-TRAN III	Kurokawa et al. (2008)	Mechanical	Bluetooth wireless and Serial
Chain and Mobile	Driving wheels	Sambot	Wei et al. (2011)	Mechanical	Wireless and Serial
Mobile	Vibration Motor	Kilobot	Rubenstein et al. (2012)	-	Infrared
		Droplet	Klingner et al. (2014)	-	Infrared
Lattice	Manual	Telecubes	Suh et al. (2002)	Magnetic	Infra-Red
		BlinkyBlocks	Kirby et al. (2011)	Magnetic	Serial
	Disassembly	Miche	Gilpin et al. (2008)	Magnetic	Infra-Red
		Smart Pebble	Gilpin et al. (2010a)	Magnetic	Serial (EP magnets)
	Mechanical	ATRON	Jorgensen et al. (2004)	Mechanical	Infra-Red
	Stochastic	Programmable Parts	Bishop et al. (2005)	Magnetic	Infra-Red
		Stochastic 3D	White et al. (2004)	Hydraulic	Serial and Global
		Lily	Haghighat et al. (2015)	Magnetic	Serial and Wireless
	Magnetic	Planar Catoms	Kirby et al. (2007)	Magnetic	Infra-Red
	Electrostatic	Catoms 2D	Karagozler et al. (2009)	Magnetic	Serial
Lattice and Mobile	Pivoting	M-Block	Romanishin et al. (2013)	Magnetic	Infra-Red

2.1.2/ MOTION

Another criteria which modular robots can be group is by its type of motion capabilities. The *Chain* architecture is less focused in shape formation and uses a type of locomotion based on its chain properties with mechanical actions. For others, the type of locomotion is important to attract robots into determined positions and form a specific shape. Several

types of locomotion used in modular robots will be analyzed here.

Chain robots moves by interaction of its modules, usually using mechanics to uplift and rotate some parts. Sambot is a type of robot that combines *chain* motions with a *mobile* architecture by aid of its wheels, what make its modules to move independently and able to dock into the ensemble.

Miche and its successor Pebbles use a motion by self-disassembly and the gravity force in order to construct their models. The goal shape is transferred to all modules of the system and, for the modules located outside of the target shape, they disable its magnets and detach from the group.

Also, a number of robots use stochastic movements to reach their final position. In a stochastic situation, modules can be moving in the environment and dock when in contact with other. After docking, modules can decide whether to remain connected or separate again based on their rules. Single-material is an example made of underwater robots that moves in a fluid tank. The attraction of these stochastic robots can be aided by valves in the system that control the fluid flow.

Catom 2D is another group of modules that uses a different motion control. This type of modules is able to move using electrostatic forces assisted by their neighbors and completing a rotation over them. To complete a roll, two actuators works simultaneously in order to allow one module to turn around the other one.

M-Block uses an original method to move in many directions. Their modules use a flywheel that rapidly decelerate which gives a torque that makes the robots pivot over others. This movement is interesting as modules can jump to arrive in higher positions, but can be hard to have a good precision about its final position.

Modular robots of the *Mobile* architecture represent modules that can move independently. An example is the Kilobots, a type of swarm robots which modules move by a vibration motor over a table to reach their final position. They are equipped with two motors that rotate the module to left or right, and, when both motors are activated at the same time, the swarm can move in a straight trajectory. Another widely used example in mobile architecture are modules based on flying drones.

Finally, some other types of modular robots does not have motion in themselves and can be manually reconfigured.

2.1.3/ Communication

Communication is essential for modular robots in order to coordinate their actions. As there is no global knowledge, information is shared with messages through a channel of communication. This channel of communication can be of many types, depending on the vision of the modular robot project.

Communication can be grouped into two more general types: With contact, communicating only to its direct neighbors or without contact (Wireless).

Contact communication allows data transfer when modules are touching and connectors aligned. The most common of these contact connection is called Controlled Area Network (CANbus), which is a standard used in micro-controllers. That can be useful as, using the same connector, it is possible to implement communication and energy transfer to neighbor modules at the same time.

Wireless connection is usually used as a form of infra-red communication and, although it is often used for neighbor to neighbor communication, it can be extended to communicate over some distance. Another advantage of this communication type is that it does not need perfect alignment of connectors to work.

Other types of modular robots, for example the Smart Pebbles modular robot, use electro-permanent magnets that can serve the functions of latching, communication and power transfer. Some modular robots can also implement both types of communication like M-Tran III that uses a bluetooth wireless transmitter and a local CANbus.

Additionally, a communication type less used but still present in some relevant works as in Werfel et al. (2006b) is called communication by stigmergy. It is a type of communication that uses the environment to store information. It is inspired from nature as ants deposit pheromones in a path between the nest and the food and termites that use stigmergy to guide the construction of their nest. As it is a marker in the environment, it does not retain much information and deriving a rule that always produce the same structure can be challenging.

It is also important to note that there are researches that explore nano-wireless communication as in Boillot et al. (2014) that could fit the needs of programmable matter. Currently, the most part of modular robots use neighbor-to-neighbor communication and this thesis focus on the needs of it.

The next section is dedicated to analyze the Programmable Matter Project.

2.1.4/ Programmable Matter Project

A first project connected with programmable matter is the Claytronics project Goldstein et al. (2004) from the Carnegie Mellon University with a long-term vision to build 3D objects from nanoscale robots. Catom, for Claytronics atom, is the unit used for these programmable matter objects, which this group has developed different types and some of them were used as real parameter for this thesis progression.

As advanced algorithms are necessary for controlling the ensemble of modules, the group in a cooperation with Intel created a programming language for modular robots called Meld. Meld is a type of logical programming language, which gives abbreviated syntax and style of code than procedural programming languages. As a declarative language, the program is written thinking in the ensemble instead of one specific module. This simplifies the task of programming these huge distributed systems that programmable matter creates.

Nowadays, a consortium involving several international academics and industrial partners are working to continue the development of many fields in this area. The consortium can be seen in Figure 2.3 and each group is responsible for part of the development, as it can be divided in independent tasks for example miniaturization of internal components, assembling the product with actuators, energy management, specific algorithms and artwork applications.

Centimeter scale modular robots were created as a first step to test and elucidate the crucial effects of the physical and interaction of the forces that affects modules. A millimeter scale robot has been build to get a closer vision of programmable matter and its constraints working at a low scale. The next sections will deep present some modular

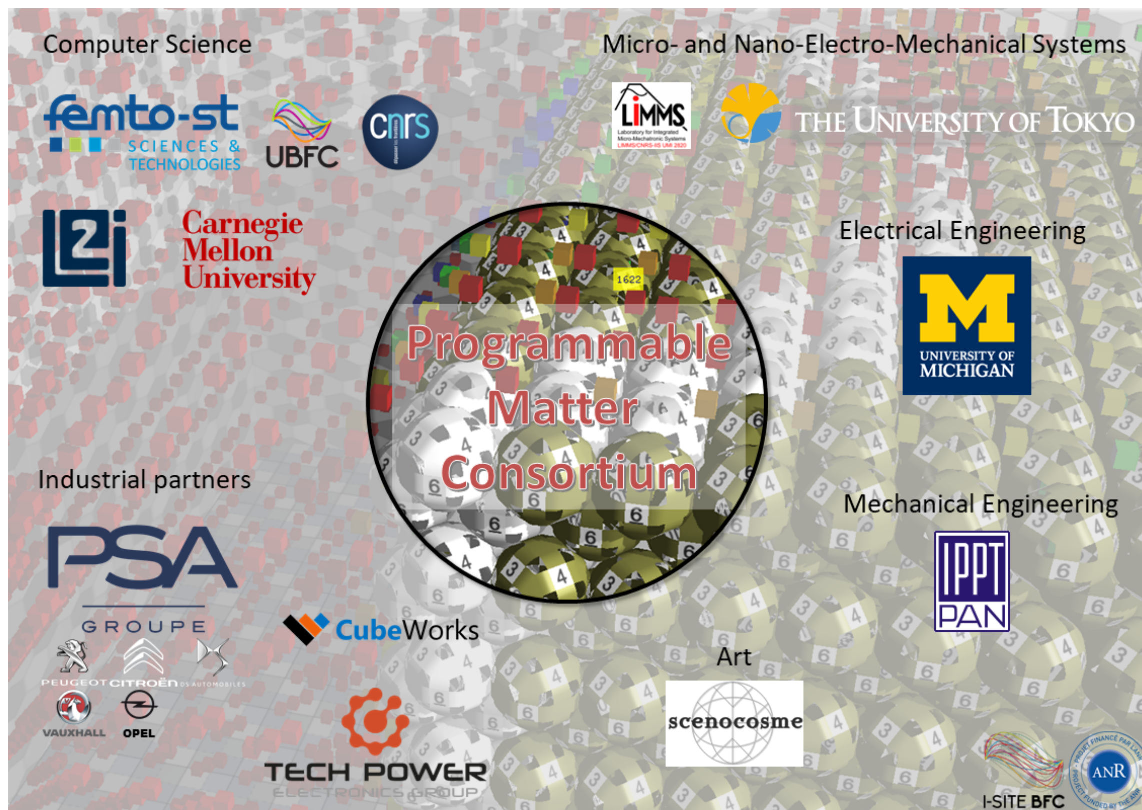


Figure 2.3: The actual consortium of partnership to develop the field of programmable matter and modular robots.

robots of the Claytronics project.

2.1.4.1/ Blinky Blocks

Blinky Blocks Kirby et al. (2011) are $4cm^3$ cubic modular robots that can be manually docked using magnets and are able to communicate with its direct neighbor blocks. A Blinky block can have up to 6 neighbors and communicate through a serial connection established with the docking. This serial connection is made with a full duplex communication controlled by Universal Asynchronous Receivers/Transmitter(UARTs) configured with a bit rate of 38.4 kBauds.

Hundreds of these modules have been manufactured with a purpose of education about distributed systems and, at the same time, working with the problems that arise when many of these modules are interacting at the same time. They are embedded with an internal storage and, to upload a new program to modules, one of them is connected to a computer that receives the new code. An internal spanning tree is created and the code is then distributed to all connected modules automatically.

Modules do not have batteries and energy comes from an external source and is distributed among the robots by their connectors that are also used as a channel of communication.

Blinky Blocks are a type of robot with manual reconfiguration and their main feedback is made using its led colors. Several other types of interactions with the system are possible

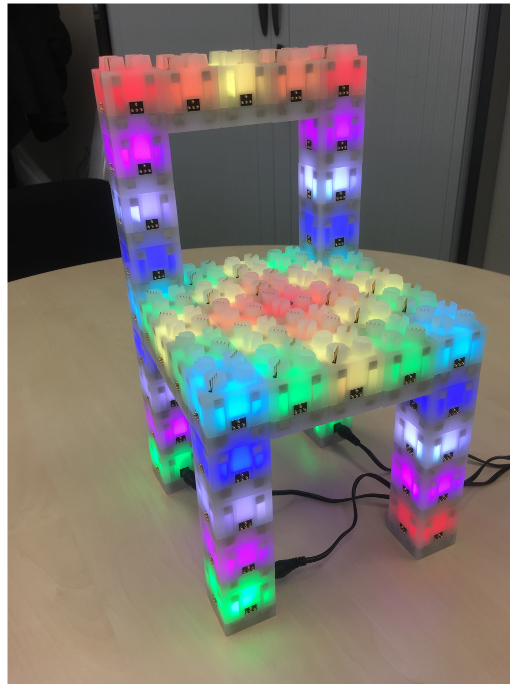


Figure 2.4: BlinkyBlocks running a distributed algorithm to find the approximative center module and apply rainbow colors based on the distance to the center.

as it contains an onboard microphone, speaker, and can detect shaking, taps and its own orientation. Figure 2.4 shows a configuration of Blinky Blocks where they use a distributed algorithm to find the centered module and their colors are based on the center distance.

Some works in distributed programming have used the Blinky Block modular robots as their main hardware to test their distributed algorithm. An example is the distributed algorithm to find the center of the structure, which is showed in Naz et al. (2016b) and a time synchronization protocol using these modules have been presented in the paper Naz et al. (2016c).

2.1.4.2/ Planar Catoms

Figure 2.5: Planar Catoms V8 with solid state electronic controls at the top of the stack, sensors presented inside and electro magnets in orange color.

The Planar Catom is a modular robot that can move without moving parts. It uses the force of its electromagnets and the cooperation of neighbors modules for its movements.

It has been developed to test the base concepts of a modular robot. One of these is the locomotion by electromagnets which can provide two-dimensional movements. To create motion the magnet rings change the poles of the electromagnetic forces, therefore movement is achieved by attraction and repulsion of magnets. Modules can converse the electrical to kinetic energy in order to turn in a clockwise or anticlockwise direction.

Aside the movements, these modules implement communication by Infra-Red with two separately multiplexed transmitters and receivers that allow simultaneous transmission.

The last version of the Planar Catom is presented in Figure 2.5 which has a 4.4 cm diameter, weights 100 grams and is composed of two rings of 24 electromagnets in total. Many versions has been developed to adjust mechanical misalignment of magnets which can cause breaks in the motion.

2.1.4.3/ 2D Catoms

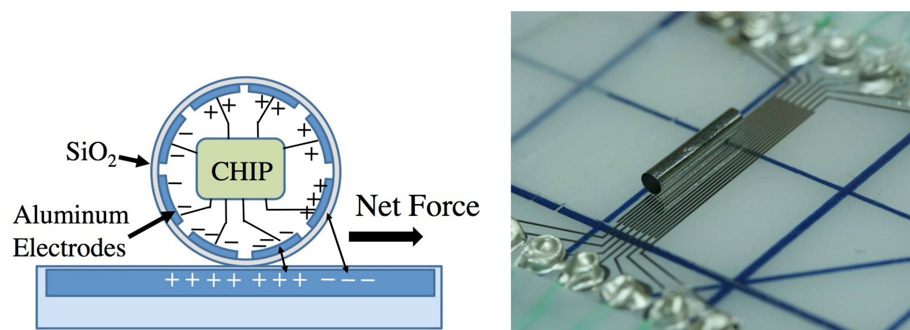


Figure 2.6: 2D Catom with the actuation scheme on left and, at right, a real prototype in a power grid. These robots have 1mm diameter.

2D Catoms presented in Kirby et al. (2007) are modules abstracted from the model of a Claytronics Atom, as CatomsGoldstein et al. (2005), that are a millimeter scale cylindrical robot validated prototype that consists of $6mm$ long and $1mm$ diameter. These robots can roll over their neighbors in Clockwise and Counter-Clockwise direction based on electrodes on the surface and on its shell. They are organized into a horizontal ponty-topped hexagonal lattice where modules can have up to six neighbors.

It is a simplified approach to a millimeter scale robot that is the base of programmable matter but instead of working with spheres they implemented cylinder based robots. This Catom is composed of a shell with an integrated chip, based on a high voltage CMOS die. The high voltage CMOS die is composed of a rectifier, a charge pump, a storage capacitor, a simple logic unit and high voltage drivers.

Figure 2.6 on right shows a Catom that can move on a power grid that contains rails which carry high voltage AC signals. The movement is produced by the forces represented on the left side of the figure. The powered chip generates voltage on the actuation electrodes sequentially creating electric fields that push the tube forward.

An interesting application of the 2D Catoms is presented in Naz et al. (2016a) where a fully decentralized distributed algorithm for self-reconfiguration is applied using these

modular robots. The algorithm presented can transform a set of initial modules into a goal shape with some restrictions over the possible target shapes. In this algorithm, modules roll over each other while constantly keeping a distance between non-stationary modules to avoid collision. They control this distance using messages thought intermediate stationary modules.

2.1.4.4/ 3D Catoms

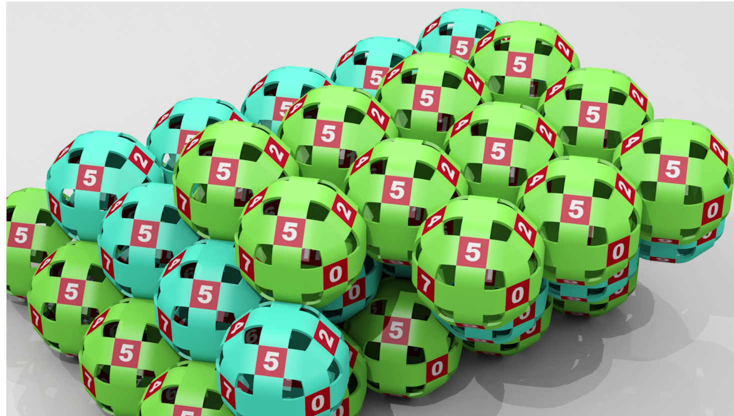


Figure 2.7: 3D Catoms in FCC lattice where modules can have up to 12 connected neighbors.

3D Catoms are quasi-spherical robots with 12 connectors that can move in three dimensions. Neighborhood of each module is placed in a Face Cubic Centered lattice (FCC) following an hexagonal close packing placement that corresponds to a dense organization of spheres. This theoretical model is currently under development in a consortium with international universities in order to create a $1cm$ diameter module prototype soon.

These robots assume the following capabilities:

- Change their color;
- Identify if there is a module in one of its connector interface;
- Communicate with their connected neighbors;
- Move to another position assisted by neighbor modules;
- Is able to get its orientation using embedded sensors.

The geometry of these robots are made to create a dense connectivity and their abilities to communicate in a Face Centered Cubic Lattice are also described in Piranda et al. (2016). With a big number of neighbors, messages can be transmitted with fewer number of hops.

Actually it exists only some two-dimensional Catoms manufactured and, although the three-dimensional Catom is currently under development, it is possible to take a series of strong assumptions to validate advanced algorithms for programmable matter as it have already a validated geometry.

Table 2.2: Coordinate of all possible neighbors in FCC lattice for Catoms3D.

Plane	Neighbors coordinates when z is even	Neighbors coordinates when z is odd
$z+1$	$(x-1, y-1)$ $(x-1, y)$ (x, y) $(x, y-1)$	(x, y) $(x, y+1)$ $(x+1, y)$ $(x+1, y+1)$
z	$(x-1, y)$ $(x+1, y)$ $(x, y-1)$ $(x, y+1)$	$(x-1, y)$ $(x+1, y)$ $(x, y-1)$ $(x, y+1)$
$z-1$	$(x-1, y-1)$ $(x-1, y)$ (x, y) $(x, y-1)$	(x, y) $(x, y+1)$ $(x+1, y)$ $(x+1, y+1)$

As writing algorithms for these programmable matter robots, we need to provide a relation between their FCC lattice space coordinates to their coordinates in the matrix. This space coordinate is described here as 3D catom is used to validated most of the algorithms presented in this document. As we can see in figure 2.7, modules are placed in regular grid in its $\vec{xy}$ plane, and its $\vec{xy}$ planes are interposed over the $\vec{z}$ axis by a ratio of $\frac{\sqrt{2}d}{2}$ where d is a module diameter.

In this way, a conversion from the matrix coordinates into a space coordinate is possible with the transformations described in the Equation 2.3. A module is connected to a maximum of 4 other modules for each of the planes z, $z+1$ and $z-1$ and these neighbors positions are described in Table 2.2. Using these data it is possible to use the expression *top module* to refer to a single position in the $z+1$ plane for FCC lattice.

The following formula, that has been demonstrated in Naz et al. (2016d) and is also present in appendix, gives the number of vertices in a sphere of radius ≥ 1 for the face centered cubic lattice:

Lemma [2.1.1].

$$n_{FCC\text{-}Sphere}(r) = 4r + 2(r+1)^2 + 2(r-1)4r \quad (2.1)$$

$$= 2(5r^2 + 1) \quad (2.2)$$

This formula demonstrates that FCC modular robots network form sparse and large-diameter networks.

$$(ix, iy, iz) = \left(d \times (x + 0.5\alpha), d \times (y + 0.5\alpha), \frac{\sqrt{2}d}{2} \times z\right) \quad (2.3)$$

A simple way to visualize the organization of these modules disposed in a FCC is that for one centered module it can have 4 modules connected in its bottom connectors, 4 modules connected in its middle connectors and 4 on modules on its top connectors with a total of 12 neighbors.

2.2/ Simulators

The use of software to simulate the real world is a fascinating way to push the development of specific areas. Simulation helps better understanding the real world without applying expensive time and money components. Think about space exploration cost and time to deploy a real mission and compare to software simulators that exists nowadays and implements real physics where scientists can use to test their models.

The same ideas of software simulation can be applied to modular robots. These robots are still in development but researchers can study their comportment in environments

Table 2.3: Comparison of Modular Robots simulators.

Simulator	Characteristics	Prog. Language	Open-source	Last activity
DPRSim	Claytronics	C++	Yes	Discontinued
USSR	ATRON, M-TRAN, Odin	Java, C	Yes	May 2012
Remod3d	M-TRAN III, ATRON	C++	Yes	October 2013
VREP	General robot simulator	C++, Python, Java, Lua, Matlab, and Octave	No	Up to date project
Webots		C++, Java, Python, Matlab, and URBI	No	
Gazebo		C++	Yes	
ARGoS	Heterogeneous swarm robots	C++	Yes	Up to date project
VisibleSim	Lattice based modular robots	C++, Meld, Javascript and Python	Yes	

with thousands of robots, analyzing which geometry and components are more suitable for the task. Asides of the hardware parts, the distributed system it creates involves many programming challenges that can already be considered while a real hardware is manufactured.

Simulators allow us to remove some constraints to target a specific goal. For example, it is easily possible to consider an environment without gravity or a perfect world with no faulty modules, what can simplify the first phase of a work and can be added in a postponed version.

Another advantaged of simulators for modular robots is the time to deploy the new algorithm into modules. Some modular robots deploy software using a wireless method Rubenstein et al. (2012), others, as BlinkyBlocks Kirby et al. (2011), sent the program to a gateway that creates a virtual spanning tree and distribute the code by direct communication and for Droplets Klingner et al. (2014) they use a platform with limited number of modules to redeploy the programs. All these methods can take some time and are exposed to error.

Modular robots come with some cost and, to test our algorithms with thousands of robots, a simulator makes it practical, which provides an easy way to deploy and test the algorithm over a big number of robots in a perfect world.

A simulator should receive an initial configuration and must evolve during the time to the goal shape meanwhile providing means to visualize the development of the application, that can be: printing text messages or with a graphical visualization directly in the 3D interface.

As for a big distributed system, which modular robots are intrinsically close, the simulation of messages is the main core of the system. One or many modules are responsible for

sending an initial message to their neighbors. Modules receiving these messages should process them and resend until there is no more messages in any module of the system. Therefore, there are two main functions for modular robots distributed systems, a startup function responsible for the initial message and an event function responsible to deal with all events, including messages, coming to this module.

Table 2.3 shows a list of simulators that have been used for different modular robots. Many specifics simulators were created as for ATRON and M-Tran. DPRSim is a simulator developed for Claytronics project with the purpose of performance evaluation with millions of modules. These three simulators have not received an update since a long time, but the research on modular robots has not stopped. Many teams use different general robot simulator as they are robust, have many options and are always receiving updates. The contrast of having many options is that they are not optimized for modular robots and most of them are not open-source for editing capabilities although these simulators presents many settings. Scalability with several thousands of modules can be a challenge with these simulators.

ARGoS Pinciroli et al. (2012) is a real-time simulator for large heterogeneous swarm robots and implements a physics engine. For lattice-based modular robots, VisibleSim Dhoutaut et al. (2013) is the most suitable simulator where almost one million robots have been simulated with a regular desktop computer.

I present in the next section the lattice-based simulator VisibleSim that was used for the evaluations realized during this work.

2.2.1/ VisibleSim

VisibleSim[1] is a C++ simulator for modular robots that allows users to run event-based simulations. An event is a task executed by the module simulating one of its actuator, for example sending or receiving a message, a tap detection, an interface that has been docked by other module and many others.

Many models of modular robotic systems with their particular characteristics are available in the simulator including Blinky Blocks, 2D Catoms and 3D Catoms models. Using this simulator, we can implement prototypes and extend them to a very large number of modules to get results about communications, number of movements and execution time. It implements also a GUI (Graphical User Interface) in OpenGL that makes possible to visualize the execution of algorithms by making modules move and change their color (see Figure 2.8).

Let analyze an example of a distributed algorithm using this simulator. Algorithm 1 shows an example of broadcasting the lower id of the network showing how this is different from a centralized version where it can be compared to a simple algorithm that finds a minimum number in a vector. Initially, each module only knows its own id, so for each module the minimum id is its own id. Each module sends a message to all its neighbors to inform them they have the minimal id. This message is then analyzed by neighbors, and if one of the ids just received is lower than the minimum it had stored, it broadcasts this new lowest id. As it is possible to imagine, this algorithm needs many interactions between modules depending on how these modules are disposed and the delay the message takes to reach the modules.

[1] https://github.com/claytronics/visiblesim

Algorithm 1: Broadcast the network minimum id using VisibleSim Simulator.

```
Input:
id // unique id number of the module
minimumId // lowest id number of the network
minimumIdMessageId // id for the minimum id message event
1  Function startup():
2      minimumId ← id;
3      message ← newMinimumIdMessage(lowestId);
4      sendMessageToAllNeighbors(message, trasmissionTime, deltaTime);
5  end
6  Function processLocalEvent(event):
7      if event.id = minimumIdMessageId then
8          if event.message.minimumId < minimumId then
9              minimumId ← event.message.minimumId;
10             message ← newMinimumIdMessage(lowestId);
11             sendMessageToAllNeighbors(message, transmissionTime, deltaTime);
12         end
13     end
14 end
```

In a distributed system, as one message can come first or later than another depending on the transmission time and the time a module takes to process this message, the program running in the same topology has a different execution for each run and the algorithm should be robust enough to deal with any of these executions.

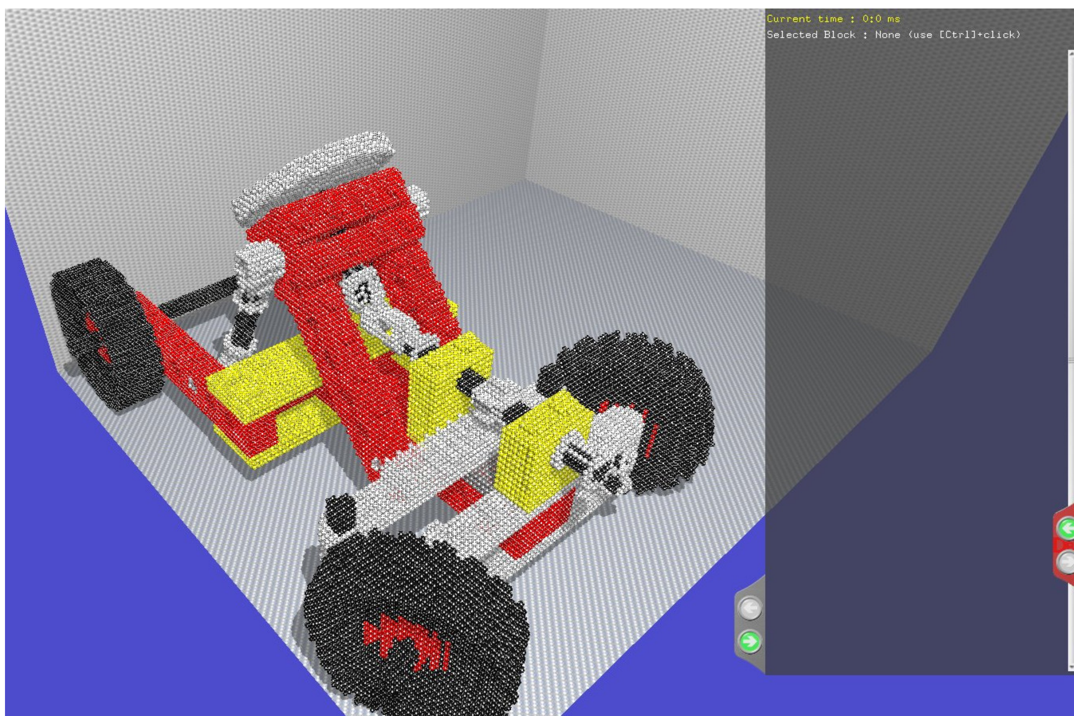


Figure 2.8: VisibleSim simulator executed with 127.583 Catoms 3D emulating a toy car.

Every module in the system loads the main code that is called BlockCode, that is exactly the same for all of them, and they do not have memory sharing. Each transference of data should be made by messages. Modules are internally assigned with an exclusive ID and, as they are placed in a regular grid, capable of associating a system of coordinates relative to a specific module.

A BlockCode is composed of two main functions: startup and processLocalEvent. Startup function is an initializing function responsible for assigning default values for modules or sending an initial message. This function is automatically called once for every existing module in the system when the module is turned on. ProcessLocalEvent is a function that receives as parameter an Event. The simulator does not guarantee an order for those events and the algorithm has to be prepared to handle all possibilities of message arrival order. Messages have an id to identify and differentiate them, for our example, the broadcast of minimum id message is called $minimumIdMessageId$ and is compared to the receiving event at line 7.

Messages and events are pushed to the scheduler system with a delay to be executed. TransmissionTime and DeltaTime are two information for the internal scheduler to adjust delays when calling the $sendMessageToAllNeighbors$ function. Events are then executed in the scheduler when the scheduler time matches an event execution time. One tip to have more stochastic algorithms, is using random delays when pushing new events in the scheduler, changing the order of the events and making each evaluation executed on the system unique, yet reproducible for a given seed.

2.2.1.1/ Classical Algorithms

Algorithm 2: Broadcast algorithm using VisibleSim Simulator.

Input:
id // unique id number of the module
$broadcastMessageId$ // id for the broadcast message event
$broadcastMessageReceived$ // boolean for broadcast message received

```
1  Function startup():
2      broadcastMessageReceived ← False;
3      if id = 1 then
4          message ← newBroadcastMessage();
5          sendMessageToAllNeighbors(message, trasmissionTime, deltaTime);
6      end
7  end
8  Function processLocalEvent(event):
9      if event.id = broadcastMessageId then
10         if broadcastMessageReceived = False then
11             broadcastMessageReceived ← True;
12             sendMessageToAllNeighbors(event.message, transmissionTime,
                 deltaTime);
13         end
14     end
15 end
```

Numerous classical distributed algorithms can be useful in a network of modular robots, for example, to broadcast an information to all modules in the network, a message from the module that wants to share the information sends a message to all its neighbors modules that replicate this request again to all their neighbors and so on. Using this algorithm, at some moment all modules will have received the information.

As modules can have a big number of neighbors, 12 for the case of quasi-spherical 3D Catoms, these *broadcast algorithms* should be aware that they can receive the same message from different sources and in any order.

Algorithm 2 shows a broadcast algorithm written using this simulator. It transmits an information to all neighbors and avoid retransmitting the same message twice. At the end, all modules in the same connectivity is guaranteed to have received the message.

If it is on the interest to be sure that all modules have received the information before advancing to a subsequent task, this algorithm needs to implement an acknowledge procedure. First of all, in order to have an acknowledge protocol, the notion of parent and child should be established, where a parent node can have many children and a child node must have just one parent, with an exception for the initial node that has no parent. It generates a tree without cycles that is called *spanning tree*. A node that has no children, when receiving the broadcast message, sends a response immediately to its parent and, after a node receiving a message from all its children, it resends the message to its parent until this message arrives to the initial node. When this initial node receives the confirmation from all its children, it means all modules have confirmed acknowledgment.

This algorithm depends on the confirmation of the message delivery and it is largely used in modular robot systems. As before starting any self-reconfiguration algorithm, the target shape and a relative position over this target shape should be distributed. Two types of message are used in this algorithm:

1. map: a message that contains the description code of the scene and the position of the receiver (in the set of modules relatively to the master block).

2. ack: acknowledgment that the entire subtree of neighbors linked to a module has received the data.

An arbitrary initial module starts sending the map message to its neighbors that will resend it to its neighbors and so on. When the ack message arrives to this initial module again, this module has the confirmation that all modules of the network have received the message and can continue to the next step of the self-reconfiguration.

The simulator presented here is continuously developed according to the prototypes we work with and our needs. As our prototypes are always placed in a regular grid, a swarm version has not yet been tackled. Although the simulator is very suitable for the actual stage of development for modular robots some options are still to be implemented and not considered in simulations as for physics, control of gravity and center of mass. The simulator is still in development and new functions are added whenever necessary.

2.2.2/ VISIBLESIM ONLINE

VisibleSim Online[2] is an extension of VisibleSim that executes in web browsers that support *WebGL*. It has been developed by myself with the idea to simplify and make accessible the distributed programming for modular robots with visual feedback. It has the potential to have a great impact on education, combining robots with distributed systems.

All an individual needs to start programming a modular robot system is a Web Browser. Starting from it, it is possible to modify manually the initial distribution of robots or import a file configuration that describes the initial emplacement of modules in the scene. It is also possible to modify the configuration during an execution. A robot face can be chosen and attach a new block at it, delete the selected robot and save the overall created scene in XML format for later import.

The first phase of this project was to allow submitting a C++ code for robots following the same structure and supporting the same functions as in the base version of VisibleSim without code duplication. To make the simulator work from a browser, a modified VisibleSim C++ version is executed in the server that listen for socket connections from a client. In order to execute the code, the scene configuration and the CodeBlock, the code that is executed in each module, is sent to the server.

When executing the local VisibleSim, if a modification over the CodeBlock occurs, a recompilation of the system is made necessary. This compilation and linking of files would be expensive for the online version with many clients accessing the same resources. A solution that I found was to use a dynamic shared library. Dynamic shared libraries make possible to load a library after the startup of a program, and our task is resumed to transform the given CodeBlock in a shared library. Every time a code is sent to the server, a shared library is created from this file with the following command on gcc compiler for GNU/Linux operating system:

```
g++ -fPIC -shared codeblock.cpp -o codeblock.so
```

From the server instance, the library of the CodeBlock can be loaded using the interface function *dlopen*. As we know by advance the name of the main functions, startup and processLocalEvent, they are linked using the interface for shared libraries called *dlsym* that requires the function name as an argument.

At this point, the code is ready to execute as a base VisibleSim instance. As the program executes, events as color change or messages are transmitted back through the socket connection along with the time in the simulator scheduler which they happened. When there are no more events left, the simulator sends a message of end of simulation and the animation starts.

An API (Application Programming Interface) that supports others programming languages has been created to this platform with the ambition to reach a greater public. Currently, *Python* and *JavaScript* programming languages are supported along with *C++*, and the implementation of other programming can now be done easily with this API. The base code is the same of *C++* VisibleSim to means of compatibility and code reusability. The system flow is the following:

- The desired code is sent to the server that executes the code;

[2] http://projects.femto-st.fr/projet-visiblesim/en

- If the code is written in C++, it loads the custom code as a dynamic library and sends each event by a socket connection to the browser.
- If the code is written in any other programming language, the API is used with the functions to control the robots. In the server side, a surrounding program in the specif programming language creates a connection to the C++ VisibleSim Server. They communicate over a channel using a structured *JSON* for data passing.

For example, the API consists in a formatted message that is transmitted to the Code-Block with all the information needed by the module to take any action. It comes with its identifier number, all variables saved on this component and, if it is the case, the event data. A function to send a message was created inside each programming language supported. Whenever a message is sent into a direction, the client saves its action and resends to the server using a formatted message defined in the API.

For the moment VisibleSim Online supports executing codes on the fly, saving and loading a XML configuration of the scene for BlinkyBlocks robots. This is a first version that I have worked to make modular robots programming easily accessible by the community. The aim is to extend this simulator to support other types of robots that implement rotations for example Catoms2D and Catoms3D and attract more people interested in algorithms for modular robots.

3

Scene Encoding for Programmable Matter

Programmable matter can be seen as a huge modular robot in which each module can communicate with its connected neighbors. All modules work together to achieve a common goal, more likely changing the shape of the whole robot. However, when the number of modules increases, the memory used in each module to store the target shape and the computation time to recreate this shape increases too.

This thesis studies different approaches to describe the shape of any object for a huge group of modular robots. The use of a good method for coding scene is a critical aspect that can reduce the memory, the time of transfer and the energy used of many distributed algorithms like self-reconfiguration. In this chapter, I propose a method called Constructive Solid Geometry for Programmable Matter (CSG4PM), that is composed of a compact description of an object and all the associated algorithms preprocessing and runtime.

The most interesting application of programmable matter is the ability to move each module in order to change the global shape or morphology of the whole, what is called self-reconfiguration. As a first step presented in the flow of Figure 3.1, the goal shape has to be transmitted to the system. There are actually three main solutions to deal with reconfiguration scenes on modular robots:

- Representation without coordinates proposed by Lakhlef et al. (2013). This method does not need data transfer but can only describe some simple geometrical shapes like squares.
- Shared model using a distributed shared memory system proposed in Bourgeois et al. (2016). It can drastically reduce the size of data memorized in each module.
- Compact representation is the solution explored here. Generally, the scene is first encoded and sent to a master module connected to a computer. Then, it is sent to all other modules of the network. The description is finally processed in each module in order to determine if the module is correctly placed or not. Compact representations are important in order to optimize memory space, reduce network bandwidth and to allow a faster broadcast of the scene.

Each robot is independent and must decide where and when to move using the knowledge of local neighborhood and distributed data only.

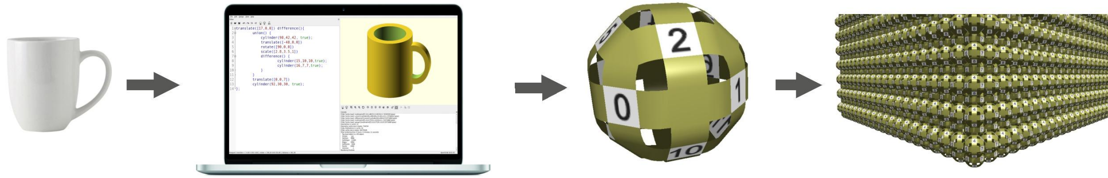


Figure 3.1: Process to define a target shape used before a reconfiguration algorithm. Firstly, a model is chosen and modeled. This model is then transfered to a master module that retransmit it to the entire network of modular robots.

A first module is assigned with a position in the coordinate system that is inside the object shape map. The following modules can be assigned with relative positions from this first module since modules can know their orientation and at which interface the neighbor is connected. This way, after a first flood of information, modules can have a relative coordinate system to the target shape, the current state of the configuration around their position and the final target shape. Reducing its description size can reduce significantly the time it takes for this information flooding.

For each step of a self-reconfiguration algorithm, every module needs to check if it is already in the goal map and check other geometrical information that allows identifying positions that it must reach in order to reduce the global distance of the current configuration to the final one. Thereby, a description that has a fast way to identify if positions are inside the form or not can help to reduce time and energy usage for modular robots systems.

In this chapter, we present the Constructive Solid Geometry for Programmable Matter (CSG4PM) which is based on a CSG representation of objects and provides all the algorithms necessary to get the state of a determined position. For example, the Toy Car, constructed using CSG4PM and presented Figure 3.2 is defined by 427,921 3D Catoms and occupies only 18.3 Kilobytes. We can note that with this description the memory size of the coding model does not depend on the final resolution of the object but on the details contained in the description.

This chapter is organized as follows. Section 3.1 discusses the related works and the different solutions proposed in the subject. In section 3.2, there is an explanation of the solution proposed followed with the algorithms to determine if a position is included in the model. Section 3.3 shows some experiments comparing the proposed method to others in means of size, time and fidelity comparison. Section 3.4 concludes about the proposed method and list some future works.

3.1/ Related Works

The next sub-sections present different types of scene representation that have been used in modular robots and also possible formats.

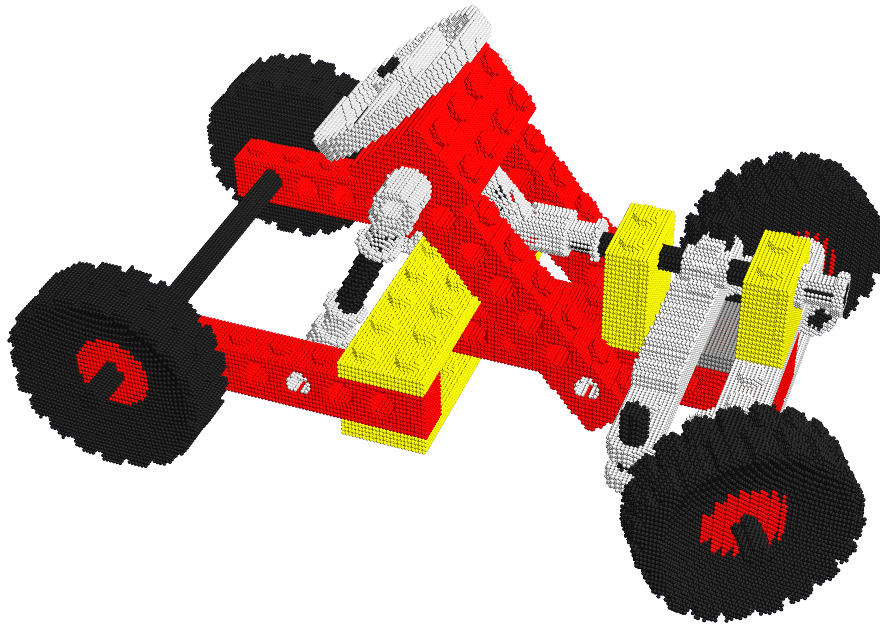


Figure 3.2: Huge set of 429,921 micro-robots defining a Toy Car.

3.1.1/ Bitmap Representation

In Butler et al. (2002), Butler et al. published a related work on mapping a configuration of modular robots. In their work, they represented the final configuration as a binary matrix, with 0 corresponding to empty spaces and 1 to occupied.

It is a well-known way to represent the goal configuration, and, furthermore, it is easy to implement without any loss of details. In addition, a simple operation could tell when the module is inside the model.

The negative point of this representation is that it depends on the number of modules and it will grow linearly with the size of the robots giving some restrictions on *scalability*. Also, the idea to have the same representation to different object size, as depending on the quantity of modules available, cannot be achieved in a simple way with this method.

The following related methods that will be presented in this section are all vectorial methods.

3.1.2/ Constructive Solid Geometry

Constructive Solid Geometry (CSG) proposed in Requicha (1980) is a classical method for describing scenes in image synthesis. It consists in defining a tree of simple geometrical objects, boolean operators and transformations that are combined in order to model the final scene. Leaves of the tree contain geometrical models and internal nodes are associated to geometrical transformations or boolean operators. Geometrical transformations can be used to apply displacements, rotations and scales over the subtree, they occupy internal nodes of the tree. Three boolean operators are usually used: union, intersection and difference.

Geometrical objects are defined by mathematical formulas that allow knowing if a point of the space is placed inside or outside the object. For example, a sphere S of radius r can check if point p is inside it with the following simply mathematics:

$$p \in S(r) \mid (|p_x| \wedge |p_y| \wedge |p_z|) < r$$

The union of many objects is the volume filled by at least one of the objects, the intersection of many objects is defined by the common volume of all objects, and the difference $A-B$ is the volume of A that is not in B. These n-ary operators are detailed in Equation 3.1.

$$\begin{aligned} Union(B_1, B_2, ..., B_n) = & \quad B_1 \cup B_2 \cup ... \cup B_n \\ Inter(B_1, B_2, ..., B_n) = & \quad B_1 \cap B_2 \cap ... \cap B_n \\ Diff(B_1, B_2, ..., B_n) = & \quad B_1 \cap \neg (B_2 \cup ... \cup B_n) \end{aligned} \tag{3.1}$$

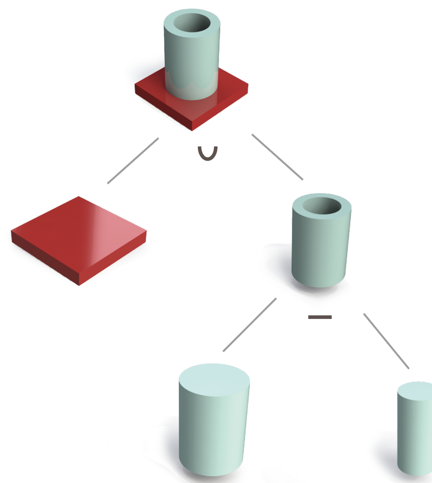


Figure 3.3: Operations to define a mug with CSG Tree.

Figure 3.3 shows an example of CSG tree constructing a simple “mug” scene with two different boolean operators (union and difference). Coding a scene using a CSG tree is very compact, because it consists in defining the volume occupied by the matter of the scene. Each object may be a simple geometric canonical object that is described by some intrinsic parameters and placed using a homogeneous transformation matrix. For example, a sphere placed in the origin of the coordinate system is just defined by its radius, a cylinder needs a radius and a height as we can see in Figure 3.4. However, describing a complex scene using CSG tree becomes harder when it contains small details. In our case, the smallest size of a detail is the size of a module.

In our experiments we use OpenSCAD[1] that is an open source software with the capabilities to compile and render CSG scripts where the code can be easily tested before being passed to our simulator.

3.1.3/ Boundary Representation

Boundary Representation presented in Foley et al. (1996), often abbreviated as B-Rep, is a geometric 3D modeling technique for solids using surfaces. This method consists in

[1] https://github.com/openscad/openscad

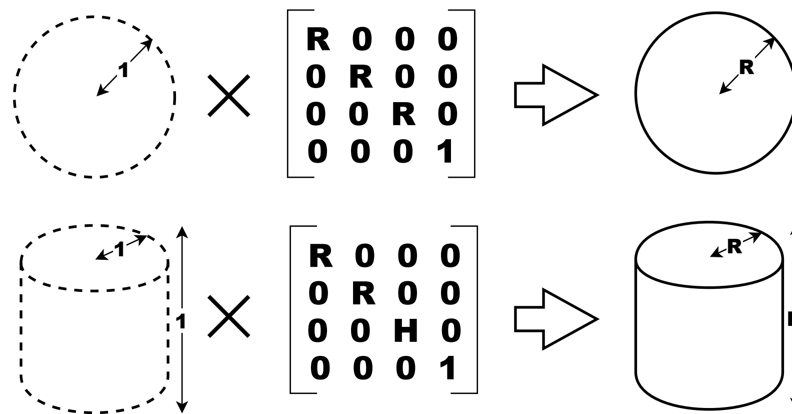


Figure 3.4: Canonical objects, sphere and cylinder, represented along with its transformation matrix.

representing the surface of geometric objects by a collection of connecting faces, represented by edges and vertices.

Boundary Representation can be used to describe solids and non-solid models. A solid is defined using this method by a closed surface that is the interface between exterior and interior regions.

3.1.4/ Triangle Mesh

Triangle Mesh is a specialization of B-Rep model and a very common representation of 3D objects. It consists in approximating the shape of an object by a set of small planar surfaces that define the border of the object, and then, interior and exterior spaces. The advantage of this description method is that we just have to describe a 2D surface in order to construct a 3D object, and it therefore needs less memory.

A wide number of 3D image software, for example 3D Studio Max and Blender, use Triangle Mesh despite the fact that this solution does not guarantee the final object to be a solid.

The representation of the surface is largely used to render objects on a screen but it may not be the best representation for our problem. Indeed, verifying if a module position is inside an object is complex as an object is described as a 2D surface.

To verify if a certain position is located inside the described object using Triangle Mesh an algorithm of Ray Tracing can be used. This idea is based on the Theorem 1 named Jordan Curve Theorem which apply in 2D and was later extended to higher dimensions by Jordan-Brouwer Separation Theorem.

Theorem [1]. *Any continuous simple closed curve in the plane, separates the plane into two disjoint regions, the inside and the outside.*

This theorem creates an interesting history in mathematics field as it can be seen obvious for simple curves when we think of circles and ellipses but was a challenge to many mathematicians trying to prove it for complex curves as fractals. A deeper discussion about this topic can be found in the interesting article Ross et al. (2011).

Ray Tracing, an algorithm to render scene dispatching rays from the camera presented in Whitted (1979), is a known technique and, to identify if points are inside or outside an object, it is possible to use a part of this idea. The main idea is to trace a single ray by selecting a point not in a border and trace a straight line from this point to outside the limits of the object. As exemplified by Figure 3.5, if the line meets borders an odd number of times it is supposed that the point is inside the object, otherwise, if the ray meets borders an even number of times it is outside.

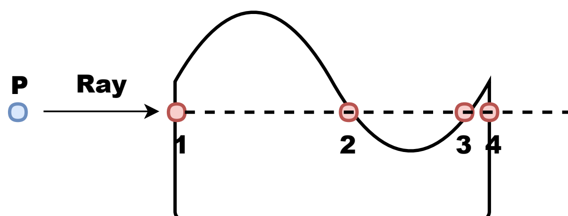


Figure 3.5: Checking if point P is inside the polygon. As the ray traverses the polygon an even number of times it is possible to assume it is located outside. Otherwise, if the ray would traverse the polygon an odd number of times, the point would be inside.

Simple analytics geometry can do this trick. First, from the position we want to verify its state, a ray with any vector direction is traced. This ray is projected over the plane of each triangle of the object. If it crosses the triangle plane, the point over the intersection ray-plane is picked and we get into a 2D analysis. This 2D analysis to check if a point on the plane is inside a triangle in the same plane can be solved in different ways.

A naive solution to check if a point is inside a triangle in a two-dimensional space is to sum the angles of the point with each of the triangle edges. The sum of these angles should be equal to 2π to a positive result. This method is simple but slow as we need to make this verification with many surfaces in a row.

Faster solutions exists, as is the case of the Same Side Technique which check if the point is on the correct side using the vectors of the triangle. This is a general method that can be applied to any convex polygon. Supposing that each vertice of the triangle is called A, B and C, if the point is in the correct side of the vector $\vec{AB}$, $\vec{AB}$ and $\vec{BC}$ it is inside the triangle. The correct side can be verified if the point is in the same side as of the remaining point. For example, in the vector $\vec{AB}$ the point P should be in the same side of the remaining point C of the triangle. Algorithm 3 exemplifies this solution.

A solution that requires less operations but restricts to triangle polygon is the case of barycentric coordinates. It is based on the equation 3.2 where A is the reference point, u is the distance walked in the direction $\vec{AC}$ and v the distance walked in the direction $\vec{AB}$. In this case, if u or v is less than 0, it is in the wrong direction and outside the triangle. If $u+v$ is bigger then 1, we can conclude it is out of the bounds of the triangle as well. Finally to be inside the triangle, u and v should be greater than 0 and the sum of both less than 1. Figure 3.6 helps to visualize this faster algorithm that was used by our experiments in section 3.3.

$$P = A + u \times (C - A) + v \times (B - A) \tag{3.2}$$

However, some problems arise with the ray tracing vector and makes it slight complex for use in modular robots.

Algorithm 3: Same Side Technique to check if the point P in the plane is inside the triangle formed by the vertices A, B and C.

```
1  Function isOnSameSide(p1, p2, a, b):
2      cp1 = CrossProduct(b-a, p1-a);
3      cp2 = CrossProduct(b-a, p2-a);
4      return DotProduct(cp1, cp2) >= 0;
5  end
6  Function PointInTriangle(P, A, B, C):
7      if isOnSameSide(P,A,B,C) ∧ isOnSameSide(P,B,C,A) ∧ isOnSameSide(P,C,A,B)
        then
8          return true;
9      return false;
10 end
```

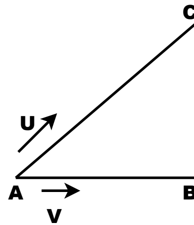


Figure 3.6: Triangle with u and v to solve the point inside a triangle problem.

1. The ray can cross over triangles borders, counting the same border twice.
2. The object represented with a Triangle Mesh may not be completely closed. This representation does not guarantee a solid object.

In the case 1, in mind that a border is shared by two triangles and could count twice the count twice for the same intersection, a solution is to avoid rays that get too close to a border, degenerating this ray and choosing another direction vector for a new ray.

In the case 2, an object may not be a solid, generating errors in this counting border algorithm presented. It may be a case of precision errors generating this objects or even a bad design causing unexpected results.

3.1.5/ Overlapping Bricks

An interesting idea was proposed in Stoy et al. (2007) and followed in Fitch et al. (2008), is to transform a CAD model, that is a largely used 3D format in industry that includes Triangle Mesh, into a set of overlapping bricks.

It is therefore less complex for modules to identify if their position is inside the bricks models than in a CAD model where we can bypass all the complexity described in the subsection 3.1.4.

Each brick, as pointed by Figure 3.7, can be represented by only two coordinates of its

most far points which reduces the size of the final model. It is a model that can be easily dimensioned for different size objects, multiplying each brick by a constant.

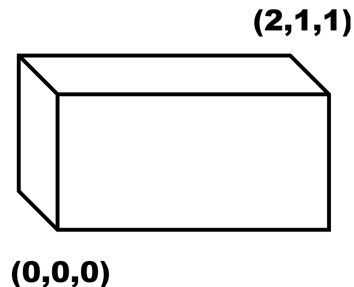


Figure 3.7: A brick that can represented using two coordinates of its extremities.

However, bricks does not produce a high quality representation of the object and, in order to increase fidelity, smaller bricks are generated which may increase dramatically the size of the model.

3.1.6/ Representation by Rules

Lakhlef et al. (2013) presents a self-reconfiguration algorithm for nodes that does not need to record any position and without a map of the target configuration reducing the memory usage. Besides, their work uses Meld programming language introduced in section 2.1.4 and the simulator DPRSim.

The mapless reconfiguration works by a set of states and rules that control the nodes. These states and rules are the hearth of the self-reconfiguration algorithm and no additional information is needed for the target shape. However, to have a new target shape, a new algorithm with specific rules should be deployed. This type of representation is useful for simple forms, but for complex shapes these rules can be hard to derive.

3.1.7/ L-System Representation

Recent work with L-System for modular robots is introduced in Bie et al. (2016) where it is used to provide a topological description for the target configuration mixed with cellular automata for motion planning.

Despite its small description that can converge to complex forms, these rules are, again, hard to be rewritten for specific tasks, what is called in fractal systems the inference problem.

3.2/ Contribution

Self-reconfiguration algorithms requires that, for each module of the set, they should be able to identify positions that must be filled. These positions are defined using the goal configuration (C_G). The objective of Constructive Solid Geometry for Programmable

Table 3.1: Summary of encoding scene methods for Programmable Matter

Encoding Name	Object Scalable	Easy to Represent a Variety of Scenes
Bitmap	No	Yes
L-System	Yes	No
By Rules Without Coordinate	Yes	No
Triangle Mesh	Yes	Yes
Overlapping Bricks	Yes	Yes
CSG	Yes	Yes

Matter (CSG4PM) is to provide the best trade-off between fidelity to the original shape, memory footprint and decoding processing time to fulfill this task.

The self-reconfiguration can be interpreted by three-step process as it can be seen in Figure 3.8.

The first step requires central computation to encode a 3D object into a set of modules. This lead to a kind of discretization which will imply in a loss of fidelity. The size and the number of modules are the two parameters that will influence the fidelity: the smaller the size and the higher the number of modules, better is the fidelity. However, the method chosen to represent the goal configuration also matters and have an impact in the fidelity.

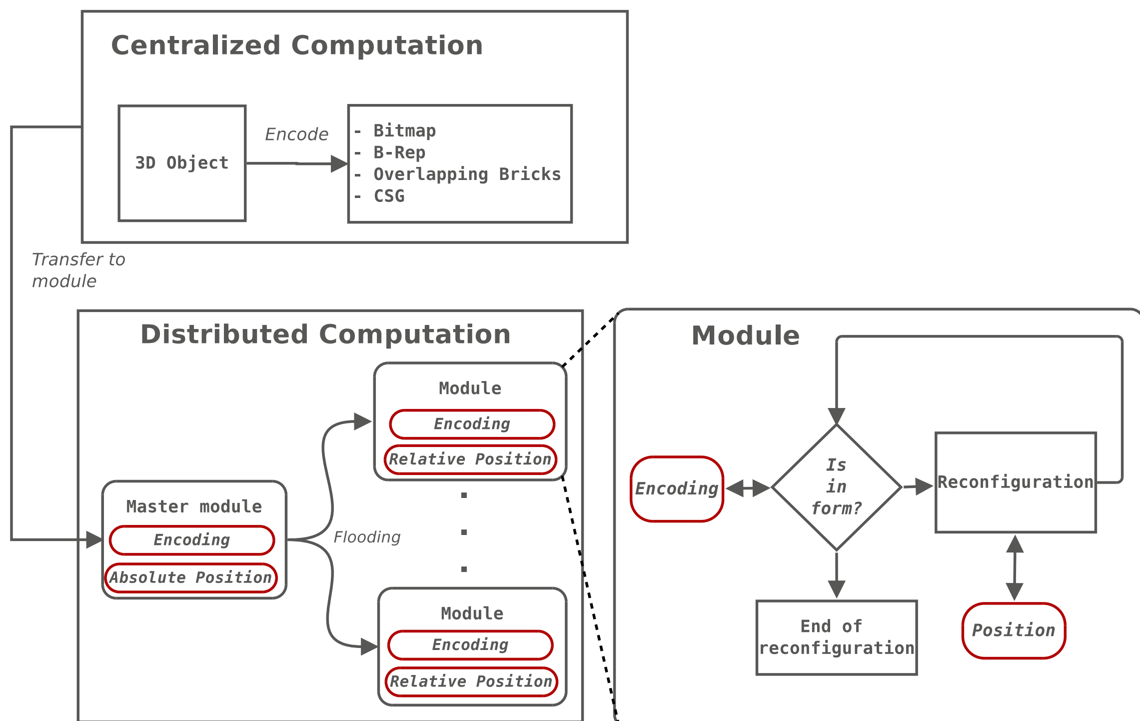


Figure 3.8: Global model flow used for reconfiguration.

To reduce the memory used in the goal configuration description while keeping a high level of detail, an encode in a vectorial compact format based on CSG Trees could fill the requirements. The CSG Tree is composed of four distinct nodes: primitive shapes, transformations, color and boolean operators.

- Primitive shapes are located in the leafs. Their parameters are the type of the shape (cube, sphere, cylinder, torus, etc.) and some associated intrinsic metrics, for example the ratio between small and large radius of the torus, the diameter of a sphere, the cylinder that can be a cone with different radius on its extremities etc.
- Geometrical transformations are placed in the tree inner nodes. Some examples are translation, rotation, and scale transformations. Statistically in our test models, geometrical transformations are more frequently used directly on primitive shapes what lead to make them accept only one child to reduce the representation size. Geometrical transformations are described using three float number parameters and internally all types if geometrical transformations are coded in one unique homogeneous matrix M.
- Color operators are unary inner nodes. They give the color of the subtree. The color of one node is given by the lowest color node in the tree.
- In order to reduce the height of our CSG trees, boolean operators are placed in inner nodes that can have $1 \dots n$ children.

A single Master Module is chosen, assigned with an arbitrary position inside the object of the goal scene and the model C_G is transmitted to it. Both information, coordinate and model, is then flooded to the entire network following the schema in Figure 3.8.

The information flood in a set of connected modules consists in sending neighbor-by-neighbor the data (coordinate and model) from the Master Module to all other modules and then waiting for an acknowledgment that means that every module has correctly received the information. For positioning considerations, the Master Module is empirically placed at a position inside the target shape, and as modules have the knowledge of which direction it is transmitting the data depending on the interface to which it is connected, it sends a relative coordinate in reference to the master module. Then, these modules send a relative coordinate to its other neighbors, and so on. This data flooding is used as prerequisite before self-reconfiguration process begins.

The next phase for self-reconfiguration consists in identifying which positions are inside the model C_G in order to define a strategy to converge to the goal configuration. Identifying positions inside the model is easily done with the CSG tree as it is described next.

3.2.1/ Evaluating Coordinate Status in Model

Considering a configuration coded by a CSG tree, we define an algorithm that allows each module to solve in/out problem as checking if the cell of position P is inside the model or not. The solution is obtained using a simple depth-first search algorithm in the CSG tree following the steps:

1. Traversing down, transmitted data are P coordinates.

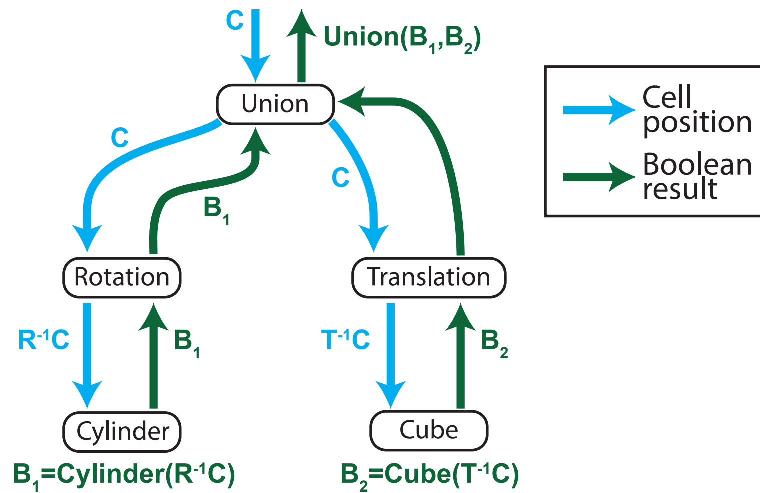


Figure 3.9: Example of traversing a CSG tree to determine if the point P is inside the described object

2. When visiting a geometrical transformation node, coordinates of P are converted into local coordinate of the subtree: $P_{subtree} = M^{-1}P$.
3. When visiting a leaf node, simple geometrical calculation allows to define if P is inside the geometrical shape coded in the leaf. For example, after the precedent transformations of the point P into $P_{subtree}$, a simple distance calculation allows checking if $P_{subtree}$ is in a sphere.
4. During back tracking, the transmitted data is a boolean value indicating if the subtree is inside or outside the model. Arriving in a boolean operator node, the combination rule of Equation 3.1 is applied once every subtree has answered its intersection state B_i.

Figure 3.9 shows an example of crossing a CSG Tree associating a rotated cylinder and a translated cube in order to deduce if a cell placed in P is included or not in the model. Blue arrows show cell coordinates transmitted when visiting a node for the first time, and green arrows are associated to boolean result (coordinate is inside or not) after visiting a subtree.

3.2.2/ How to Encode the Model

In order to send the model C_G to every module, it is necessary to transform the tree structure into a raw data that will be used by the network transport layer. This means transforming the tree into a simple array of bytes. The way this transformation occurs is important to define the final size of the model being transferred which, with a reduced size, can improve the transfer time specially because the same information is flooded to the entire network of modular robots.

This raw data is sent during initial flooding process, and then, each module must be able to decode the array of bytes to answer in/out problems. It can be obtained by a simple pre-order walk of the tree, each kind of node is coded in 1 byte followed by its intrinsic

Algorithm 4: Mug tree representation of the mug presented in Figure 3.10.

```
difference() {
    union() {
        color([1,1,1]) translate([10,10,1])
            cube([20, 20, 2.5]);
        color([0.2, 0.6, 0.8]) translate([10, 10, 12.5])
            cylinder(20, 10, 10);
    }
    translate([10, 10, 12.5])
        cylinder(20, 5, 5);
}
```

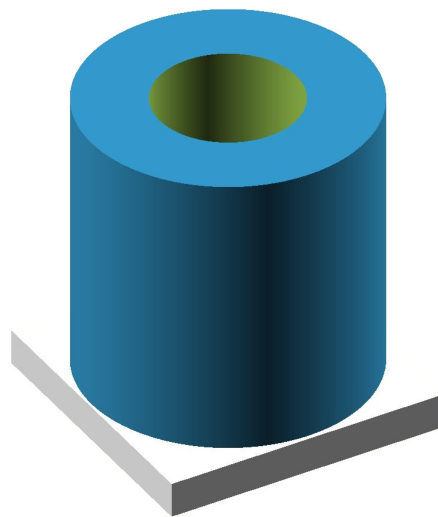


Figure 3.10: Simple colored mug used in the experiments.

parameters (here I used 4 bytes float for real values). For n-airy nodes an “end of child code” (1 byte) is added in order to stop the crossing.

The decoding process is obtained by a simple read of the array from the left to the right. This treatment is at least executed once during the flooding algorithm after reception of the map by each module, in order to know if the module is already well-placed in the goal configuration.

Figure 3.10 shows the model of a colored mug defined by a simple CSG Tree (detailed in Algorithm 4). The size of the raw data coding this CSG Tree is 65 bytes only.

The flow for self-reconfiguration is exemplified in Figure 3.11. The first step (A) is centralized in a computer and consists in discretization of the real object into a CSG tree. In the step B, a script takes the generated CSG tree and transform into raw data that is transferred to one single module (Master Module). The goal configuration is flooded from the Master Module to every module connected in the network in step C. Last step D, modules can identify coordinates that are inside or outside the goal configuration. Modules can identify if they are already well-placed according to the CSG tree (white modules). These that are not inside the goal configuration reorganize to displace to nearby positions included in the shape map (red dots).

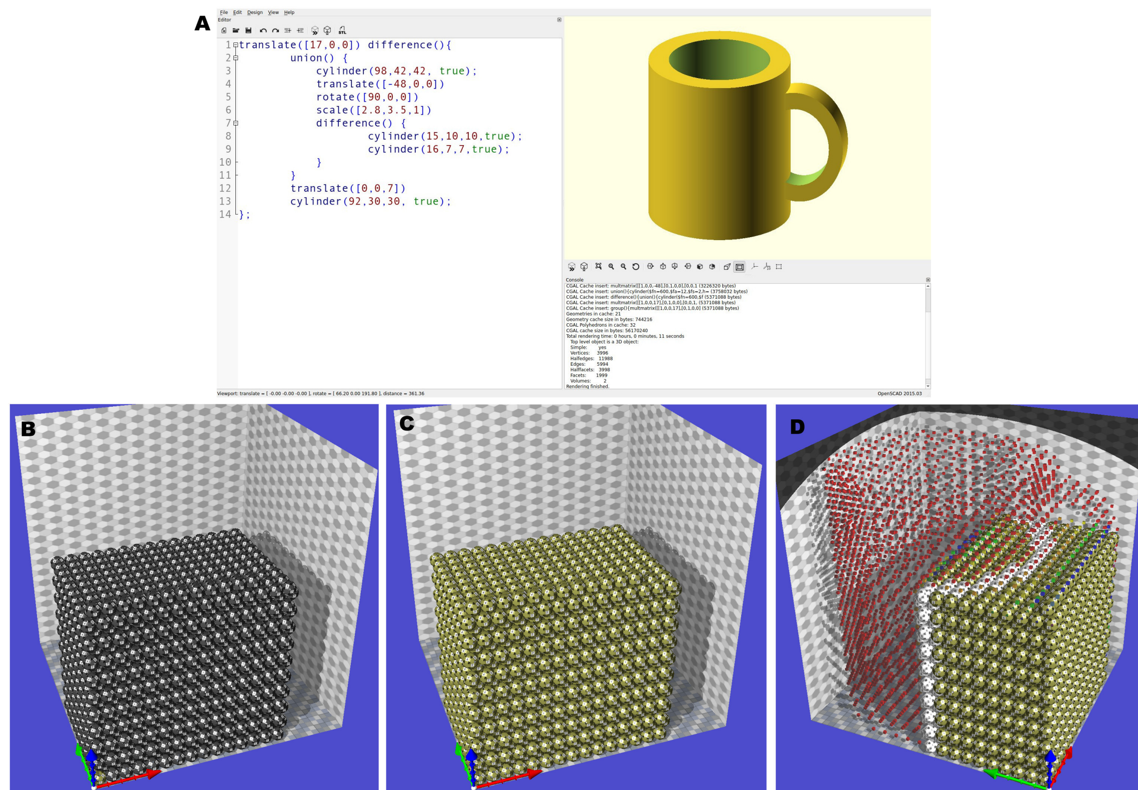


Figure 3.11: Flow of CSG4PM reconfiguration. At the top, the CSG tree is created and a binary version is transmitted to a master module (bottom left). This master module flood the goal configuration tree to the entire network of robots(modules become yellow after reception of goal configuration). These modules now can decide positions that are inside or outside the goal shape. White modules know they are inside the target shape and do not need to move. Other modules find a free position that is inside the target shape (red points) to occupy.

Toy Car already presented in Figure 3.2 is our high definition and complex test model. The associated CSG Tree is composed of 1757 nodes including 628 primitive shapes in its leafs and with a depth of 16 nodes. Using our script to generate raw data we obtain a 18.757 bytes long array description.

3.3/ Simulation

In this section, we will be comparing the generated CSG tree model with other three main methods: Bitmap model, Mesh model and Stoy overlapping bricks model.

The size of each model depends on how they are coded. Following paragraphs introduce the way chosen to optimize the description size of the methods.

Bitmap model is the simplest solution consisting in storing the configuration bounding box of cells in a boolean three-dimensional array. Empty cells are coded by False and filled cells by True. It is the simplest way to encode a scene as one position can be represented by one single bit. However, to have a better comparison with our CSG encode that can represent objects with colors, it was chosen to encode the bitmap model using 3 bytes for

color representation. Each element of the array contains the color of the cell coded in 3 bytes, and, by convention, the color (0,0,0) is reserved to code an empty cell.

For the bitmap model, the size of the representation is the size of each position on the grid multiplied by the size of information in each cell, what gives $Size = (lx \times ly \times lz) \times D$ where lx, ly and lz are dimensions of lattice, and D is the space reserved for the cell data.

The advantage of bitmap model is that it makes easy to figure out if a module is inside C_G or not. However, it implies in transmitting a huge amount of memory in the flooding step if the number of modules composing the model is high and do not allow creating objects with different scale size. In order to generate a model with new dimensions, the configuration should be recreated and redeployed to all modules.

Mesh representation model consists in describing an object by its border in the world coordinate system, in this case using only triangles like border objects as show in Figure 3.12.

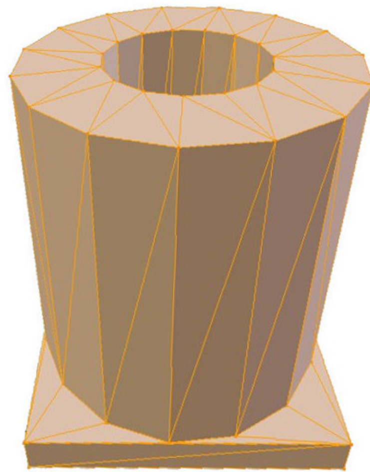


Figure 3.12: Triangle mesh used in the mug model.

As there is a great number of repeated vertexes, this representation first describes a list of all vertexes coordinates. Then a list of border faces is described specifying vertexes index and face color.

The simplest volume that can be described by this Triangle Mesh Representation is the tetrahedron that needs 4 points and 4 triangle faces. Besides, the complexity of this model depends on the precision of the mesh to approximate the shape of the object.

A script to generate the raw data for Triangle Mesh is also used. Thus, the raw data is composed of the number of vertexes followed by their descriptions that is three float numbers of 4 bytes each and then the number of faces followed by a list of faces described using the position in the list of vertex. In this work is used the convention of 4×3 floats per vertex, 4×3 integers plus 3 bytes for color per face.

One limit of this model as it describes only the boundary information, is to get data about the cells that are not exactly located on the border, for example which color to use for internal modules that can still be visible. One possible solution is getting the color of the face closest to the point, what would involve more computational and energy cost.

Therefore, mesh model implies in a harder computational process than the other methods in order to know if a module is inside or outside the model.

Definition [1]. *A point is inside a closed object mesh if each ray starting from this point intersects borders an odd number of times.*

According to the Definition 1, to identify if a module is inside an object, a ray is traced from the center of the cell in position P, with a random direction. Then, with the number of intersections between this ray and the list of faces in the mesh model, it is possible to conclude that the cell at position P is inside the object if we obtain an odd number of intersections. This method has to be re-executed with a ray in another direction if this ray cross over triangle intersections, which may give unpredictable results.

Overlapping bricks proposed by Stoy tries to simplify this approach pre-calculating bricks that are inside the Mesh model. A minimum brick size is defined depending on the demanding quality. The algorithm checks if the point in the middle of the brick is located inside the Mesh model using the previous ray algorithm and set a brick in-place when this check returns true. This calculation is done recursively for all points in the limits of the objects that will result in a set of minimal bricks. An optimization is later used to merge neighbor bricks whenever possible in order to have bigger and a less quantity of bricks. Thus, checking if a position is inside the set of overlapping models is straightforward, verifying the condition for each brick.

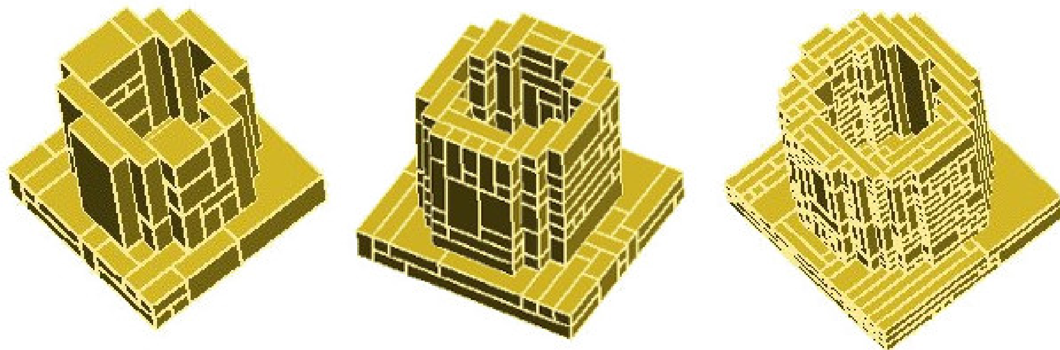


Figure 3.13: Mug representation using overlapping bricks. Three different minimum brick sizes are used.

Overlapping bricks with high resolution in our tests are defined with the smallest brick size having exactly the diameter of a module. The medium resolution is defined by one and half module diameter, and low resolution overlapping brick smallest size is twice the diameter of a module. Bricks are defined by two vertexes (using a float representations of 4 bytes per real number) and a color (coded in 1 byte). Summing it all, each brick uses 25 bytes in memory to be stored.

Experiments have been realized using our simulator, VisibleSim presented in details in section 2.2.1. The experiments aim to compare the four models of coding in terms of memory, time of treatments and time of transfer. As modules communicate sending a chain of bytes to each other thought the transport network layer, the transformation of all the models into a binary format was also necessary. This binary format is useful to

full compare the size of the encoding and the throughput of the modules communication network.

In this work two very different objects are used for experimentation. The first one is a mug shape that admits low level of detail, surfaces are smooth and of large size. The second one is a toy car in Figure 3.2, composed by complex elements with bumps and holes, presenting high level of detail.

These CSG models have been converted into Triangle Mesh presented in Figure 3.12, Overlapping Bricks (with 3 different resolutions, Figure 3.13), and Bitmap.

3.3.1/ Fidelity to the Original Format

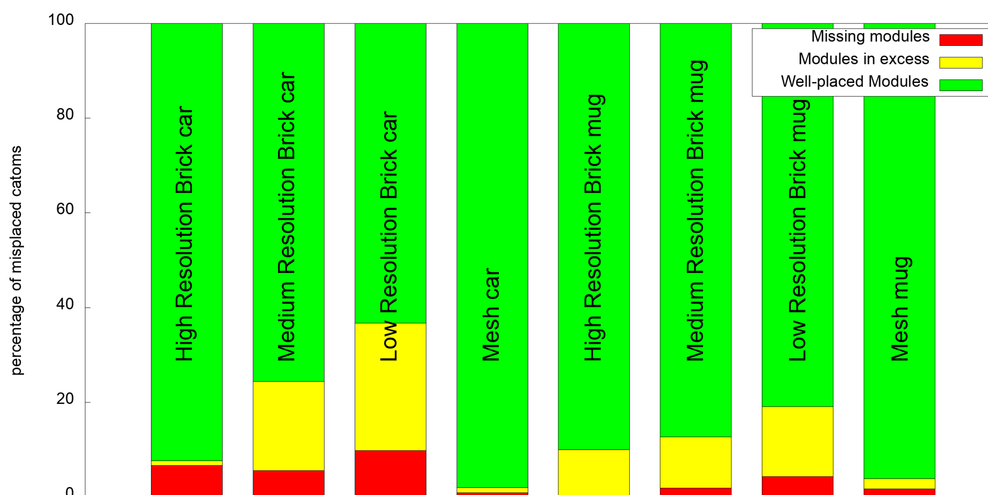


Figure 3.14: Fidelity of the Toy Car structure using overlapping bricks

Fidelity is a way to quantify how the described model compares to the real desired object. It is linked with how well we can represent the object using the available tools. To make a comparison between the representations, a first model has been created manually and enough effort was spent to create a CSG model that represents well the goal shape.

In this way, the bitmap model is created according to the resolution of the lattice and, as CSG is the reference model, they both can be considered perfect. Triangle meshes approach curved surfaces by small plane faces, that, indeed small, can produce loss of fidelity. Smaller faces give better quality, but the size of coding grows. Overlapping bricks representation can have a significant loss of fidelity depending on the minimum brick size chosen although its size can increase with smaller bricks to create a high fidelity model.

Figure 3.14 shows the quality of models for several resolutions over the two tested objects. For each representation type, all modules that are inside the model are compared how they differ from the modules placements for the CSG model.

With the chart of fidelity representation we can see that low resolution brick can generate important loss of fidelity compared to CSG model and that fidelity error for Mesh model may be neglected. Modules colors were not evaluated during this process.

Although overlapping bricks have a loss of fidelity, one advantage is that it can generate

models directly from existing mesh models, facilitating the task of checking if a module is inside the model or even facilitating the object scaling for use with the resource limited modular robot.

It can be considered that these models are enough similar to be used in the following experiments.

3.3.2/ Comparison of Code Sizes

The code size of the representation that is transmitted has a high impact in the time for distributing the goal map shape.

Bitmap model is different from the other methods discussed here as it does not scale with the size of the object. Thus, for bitmap model, the code size S of 3 bytes for the colored position depends directly on the size ($C_n = n^3$) of the cubic lattice: $S = 3n^3\ bytes$. In the case of the Mug Model, coded with $61\ bytes$ using CSG model, a cubic lattice C_3 of $n = 3$ cells per edge needs $S = 81\ bytes$, that is more than the corresponding CSG code size. For the second test model (the Toy Car, coded by $18,757\ bytes$) a only C_{19} cubic lattice needs more memory to be coded by a bitmap.

CSG, Mesh and overlapping bricks are vector methods, that is, the size of the code is invariant with the wanted resolution. For overlapping bricks method, it has been produced three descriptions corresponding to different level of subdivisions (called High, Medium and Low).

Figure 3.15 shows that CSG model gives very small size of code compared to other models. Comparing the small representation size with the fidelity proposed by the others data structures, CSG shows promising results.

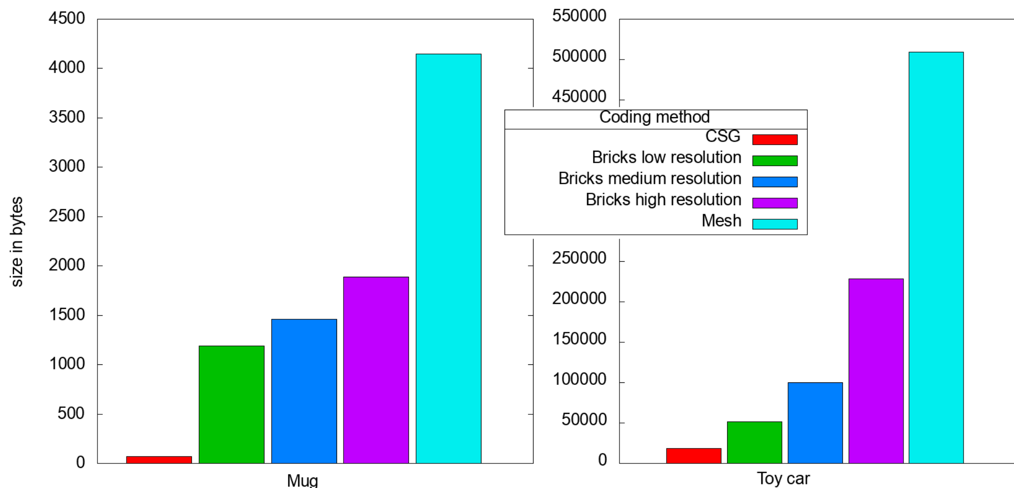


Figure 3.15: Size of codes using different vector encoding for 2 different structures

In order to compare the bitmap storage I evaluated the memory used representing the Toy car model presented in Figure 3.2. The lattice used is $145 \times 229 \times 167$ cells large, that represents $5,545,235$ cells to memorize. Then bitmap needs $15.8\ Mo$ to memorize the model, that is about 886 times more than the size of the code of the Toy Car using our CSG4PM method.

3.3.3/ Decoding Process Time

Vector models give great results in terms of the encoding size, but they need a decoding process that means being able to solve the problem of verifying whether a coordinate is inside or outside the model.

Each model has different properties and apply specific algorithms in order to execute the verification of a coordinate. Figure 3.16 shows a comparison of the average computation time of decoding task for the two studied models, mug and toy car.

It is possible to observe that computation time with bitmap model can be executed in few nanoseconds, where it consists simply in accessing a cell of a 3D array. However, CSG model drawn in red, that has the most compact encoding size, can also give a good time of decoding compared to Overlapping Bricks and specially Mesh representation. The complexity of decoding process for a CSG model is linear in the number of nodes in the tree.

In practice, the decoding time of CSG Tree may be neglected compared to communication time used to exchange data between modules. Latencies from 1 to 10 milliseconds are not unexpected in this type of network communication. However, if the decoding process is continually used to verify several coordinates, the right method can optimize resources and time.

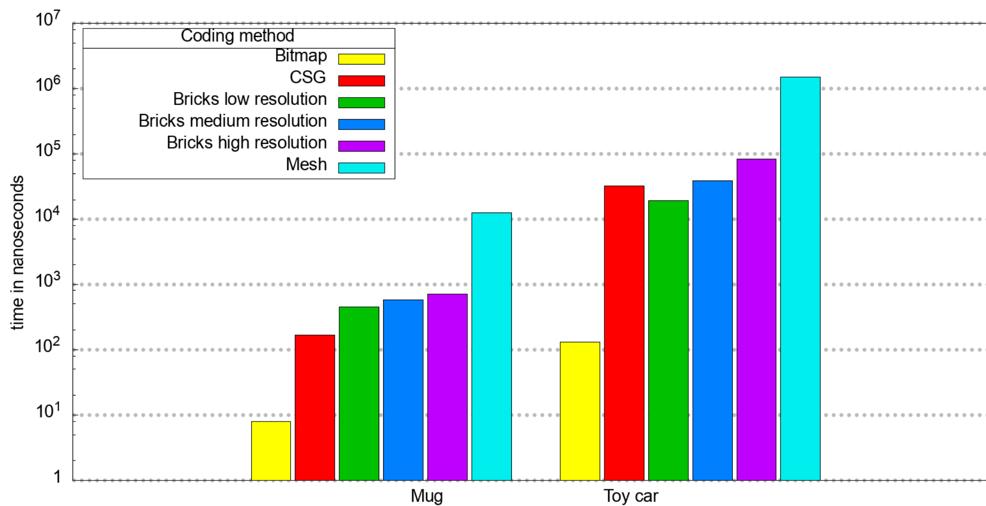


Figure 3.16: Time of decoding

3.4/ Conclusion

In this chapter, I presented CSG4PM, an efficient method to reduce the memory used in each module for storing the goal map needed by reconfiguration processes of Programmable Matter.

A comparison of the CSG4PM model with three existing methods for two very different objects both in size and in terms of complexity were made. It shows that the gain is significant compared to classical Bitmap models, and CSG4PM model need less decoding

time for better compression size compared to Mesh based method or overlapping bricks method.

The charts show also that overlapping bricks, as they can be considered a small set of CSG model where only cubes and unions exist, can be useful to automatically generate a model from an existing CAD model. However, as overlapping bricks is a subset of CSG, it implies that all models constructed by overlapping bricks can be constructed by CSG with equal or less number of primitives.

Graphic cards generally uses mesh to render objects making it the model with more research involved, but as we see it cannot be the best in the case of modular robots. CSG shows great results but it can be difficult to be automatically generated, although the method proposed by Stoy with overlapping bricks can be used in these cases. As demonstrated, a good choice of a representation method for modular robots is important and more research can be involved in this area looking for optimizations.

4

Self-Assembly Planning

A distributed modular robot is composed of many autonomous modules, capable of organizing the overall robot into a specific goal structure. There are two possibilities to change the morphology of such a robot. The first one, self-reconfiguration, moves each module to the right place, whereas the second one, self-assembly docks the modules at the right place.

Self-assembly is composed of two steps, (1) identifying the free positions that are available for docking and (2) controlling the movements and docking the modules to these positions. The work presented here focus on the first step.

This chapter presents a distributed planning algorithm that can decide which positions can be filled that is capable to create any 3D shape, including shapes with internal holes and concavities. Mechanically, a narrow space between two other modules makes it a difficult area to dock. We propose an idea that consider kinematic constraints and prevents positions from being blocked. Figure 4.1 presents positions that can be docked and that cannot be docked.

In our set of homogeneous modules presented here, distributed algorithms are proposed where each module embeds the exactly same algorithm and coordinates with others by means of neighbor-to-neighbor communication.

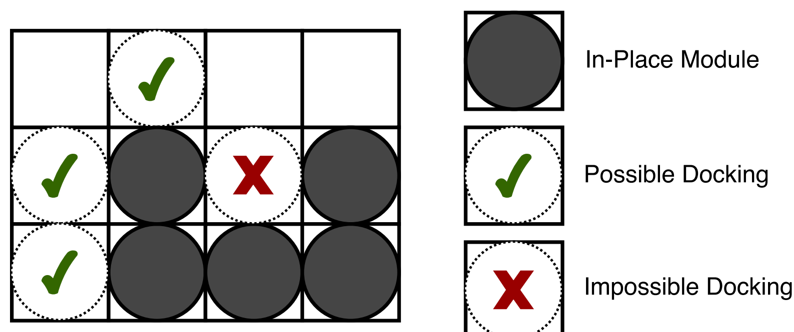


Figure 4.1: Example of possible and impossible docking positions due to kinematic restrictions.

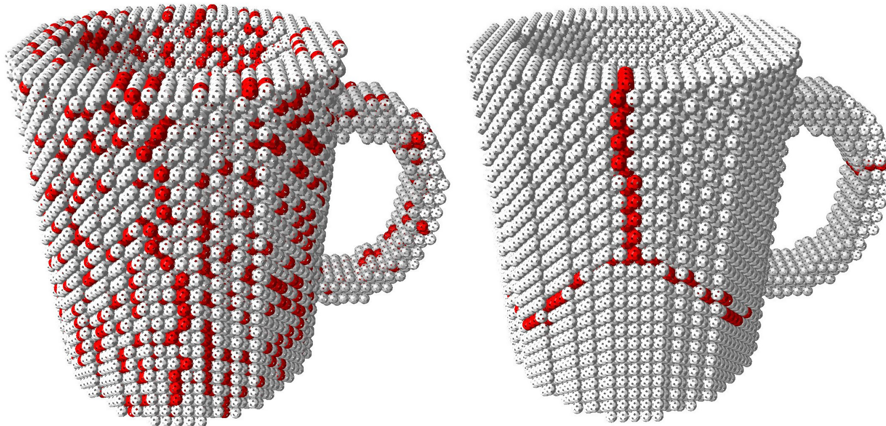


Figure 4.2: Importance of sequence planning shown on a Mug model made by 12,000 modules. Red modules represent positions that could not be docked using two simple planning algorithm. On the left, choosing a stochastic order produces 3,691 modules that does not verify the docking rule. On the right, filling regularly each module neighborhood in sequence results in 231 positions that could not be filled.

4.1/ Introduction

In this chapter, I propose a new self-assembly planning algorithm for modular robots with the following characteristics. The target is a set of modular robots composed of many homogeneous modules with limited computing resources that can change the way their modules are connected by releasing or docking some of them in order to create a given shape, thus creating smart objects. In nature, we have many examples of distributed organized constructions, for example the ants, termites and bees. Decomposing the rules that govern these beings represents a big challenge for modular robotic.

One of the most interesting capability of a system of modular robots is the ability of the modules to move towards a different position, that way changing the global shape or morphology of the whole. This is called self-assembly or even self-reconfiguration when modules can reorganize from any other initial configuration.

Self-reconfiguration and self-assembly are hard problems although the goal of it is pretty simple to understand, going from an initial configuration to a goal configuration. Many works have been done as a solution to self-reconfiguration, but always solving different subsets of it or taking of some criteria. Here we give three reasons justifying why it is complex. First, the number of possible unique configurations for a modular robot is huge: $(c.w)^n$ where n is the number of modules, c the number of possible connections per module and w the ways of connecting the modules together Park et al. (2008). In our locomotive example with 61,780 modules, there are $(12 \times 12)^{61,780}$ possible unique configurations for our 12 neighbors modular robot considering isomorphic configurations. Second, as modules can move or dock at the same time, the branching factor of the tree describing the configurations is $O(m^k)$ with m being the number of possible movements and k the number of modules free to move Barraquand et al. (1991). Third, as a consequence of the previous reason, the exploration space of a reconfiguration between two situations is exponential in n which prevents from finding a complete optimal planning. As it is possible to notice, the range of possibilities is huge, and until this moment the self-reconfiguration

algorithms presented are able to solve just some specific subsets of the problem.

Self-assembly can be divided in two steps:

1. Identifying the free positions that are available for docking
2. Controlling the movements and docking the modules to these positions

In this work, as self-assembly is a hard task we divided the problem and here we tackle mainly the first step, solving the problem of assembling 3D structures that may contain internal holes. While building the whole structure, modules are docking at available places but some positions may become impossible to reach, blocked by other modules as pointed by Figure 4.1.

That is caused for example, when executing a rotation a module needs space to conclude its movement. In a narrow tunnel robots need space to move and arrive to their destination.

There are two ways of designing the self-assembly algorithm: centralized on one module or distributed on every module. As we tackle robots composed of thousands of modules, a centralized algorithm would exceed the available memory on one module whereas a distributed algorithm would scale well. As the algorithm is distributed, each robot can be seen as an autonomous agent that gets only a partial knowledge of the system but complete their knowledge by exchanging information with connected neighbors.

Previous approaches over self-assembly planning focused on centralized planning or distributed planning for configurations *without* internal holes. Here, I present a distributed algorithm assisted by the shape description to create close-packed structures and assemble any morphology.

Defining how these modules come to their final position, which is the second step after locating where are these positions, has many possibilities. Underwater stochastic assembly has been described on Tolley et al. (2010). Robot attraction works by a system of valves that can be controlled. Thanks to the flow of water, free modules can come to the wanted position. Attraction gradient Stoy et al. (2007) is a method of attraction to avoid that all free modules move concurrently to the same position. These are just some of the methods that can be used in conjunction with the algorithm presented in this work. In both cases, local modules have the knowledge of their neighbor positions that can be filled, based on the representation of the goal configuration, and accelerate the convergence to the final structure by means of message passing or, as the example of Tolley et al., controlling valves to attract modules to dock on these positions.

Many other methods of agents multi path planning can be used combined with the proposed algorithm, although in our vision it should be a distributed path planning and an approximation as we are dealing with a huge number of modules.

The fundamental for the method proposed here consists of an initial seed module that knows the final shape coded into a string and is given a relative position in the target shape. This module waits for the connection of neighbors that are freely around the structure and allows its docking whenever possible based on the rules that will be described. When a neighbor is attached to a module already in place, it receives a position relative to the first one and the final string code of the map.

An efficient encoding of the goal shape description is essential for a good behavior on a modular robots system as the description size can grow linearly with the number of

robots already placed. As well as being memory efficient, it has to be efficient at checking whether positions are inside the goal shape as it is a recurrent task that involve energy consumption and time. The work presented in chapter 3 complements this distributed algorithm.

Our self-assembly algorithm consists of three steps. The first step is to create the description of the final scene in a spatially efficient way and transfer it to a seed module within its position on the representation. The second step is to structure the plane creation, a solution for a two-dimensional lattice is the first structural goal. Finally, the third step consists in synchronizing the two-dimensional solutions to get a three-dimensional representation and is described in section 6.

The module we use in this work is a 3D Catom developed in the Claytronics project Goldstein et al. (2004). This module, as presented before at section 2.1.4.4, is a quasi-spherical robot with 12 connectors Piranda et al. (2016). Neighborhood of each module is placed in a Face Cubic Centered lattice (FCC) that corresponds to a dense organization of spheres.

In the next section, we are going to review the related works and highlight the difference with our proposal. In section 3, I present the model, the particularities of the modular robot used, and the details about the problem we aim to address. Then, we describe how our idea and algorithm works on section 4. Section 5 is used to discuss the applications of the algorithm and the results of its performance. Section 6 introduces different ideas to pass from the 2D planning to a 3D planning with a series of different algorithms depending on the final shape. Finally, the conclusion and future works are discussed in section 7.

4.2/ Related Works

The Self-assembly planning proposed in this paper is complementary to previous research on self-assembly and self-reconfiguration systems. Previous methods were proposed although there is a lack of solutions that work in distributed systems and, at the same time, can handle internal holes as can be seen on Table 4.1, which presents an overview of existing methods.

4.2.1/ Message based

Werfel and Nagpal Werfel et al. (2006a, 2008, 2007) proposed an algorithm for distributed construction of structures without holes. These robots carry blocks and deploy them at the right position, avoiding empty spaces in the final object. They use three types of robots displacements to arrive to the proposed positions: random walking, systematic search and gradient-following. In our paper, we extend this idea to allow the construction of structures with holes.

4.2.2/ Centralized Solution

In Tolley et al. (2011b, 2010, 2011a), Tolley et al. worked around the kinematic restrictions and the importance of planning the self-assembly for their underwater robots. To have the order in which these robots should be connected, they start from the virtual assembled

Table 4.1: Robots Planning Algorithm Overview

Author	Architecture	Dimension	Distributed	Compact Representation	Internal Holes
Werfel et al. (2006a, 2008)	Heterogeneous Modular Robots	3D	Yes	Yes	No
Tolley et al. (2010, 2011b)	Underwater Modular Robots	3D	No	Yes	Yes
Stoy et al. (2007)	Porous Scaffold Modular Robots	3D	Yes	No	Yes
Seo et al. (2013, 2016)	Rectangular Modular Robots	2D	No	Yes	Yes
Jones et al. (2003)	Modular Robots	2D	Yes	Yes	Yes
Naz et al. (2016a)	Modular Robots	2D	Yes	Yes	No
Rubenstein et al. (2014)	Swarm Robots	2D	Yes	No	No
Our method	Modular Robots	3D	Yes	Yes	Yes

shape and remove the possible robots. Once there is no more robot left, the reverse order is used to assemble the modules without blockage. They also proposed an distributed approach to self-assembly layer by layer that limits the number of available positions for docking and the overall time for self-assembly. The robots they use do not have motion but move by the fluid flow they are in, with a control of their valves attracting robots to their allowed positions.

In Seo et al. (2013), Seo et al. present an assembly planning algorithm for constructing planar structure out of rectangular modular robots avoiding narrow corridors. Their approach is based on graph properties as topological sort and results in a specific centralized order that is deployed to robots before assembly begins. In Seo et al. (2016), they extend their idea to allow models with internal holes.

4.2.3/ Transition Rules

Jones and Matarić proposes in Jones et al. (2003) create a similar transition rule set to self-assembly agents to generate a consistent assembly of a desired goal structure. The decisions are made by the agent docking in the existing structure and not by the structure and, as a result, no attraction rule can be derived.

4.2.4/ Porous Scaffold of Modular Robots

Stoy et al. show in Stoy et al. (2004, 2007) an algorithm for 3D self-reconfiguration representing the final structure with overlapping bricks automatically generated from a CAD model. They adopted a porous scaffold of modular robots to prevent local minima and ensure that robots do not get stuck. That leads to a porous representation that allows modules to move freely in any direction. In their work they also analyze different methods of attraction that are interesting and can be also combined with algorithm being presented here.

In this work, we look for a technique to create a dense representation for the final robot module structure.

4.2.5/ Other methods

Gilpin et al. published in Gilpin et al. (2010a) about their programmable matter module system where each module has the size of 12mm per side and is capable of creating 2D shapes by self-disassembly. They start with a latched system and the modular robot detach unnecessary modules. In Gilpin et al. (2010b) they propose a 3D creation that rely on stochastic forces to self-assemble a close-packed crystalline lattice of modules and then self-disassemble into the specific shape. It is a functional approach that can reduce the complexity but it requires a bigger number of robots to assemble a specific shape as some robots will be discarded by the disassembly.

TERMES system is a decentralized autonomous construction team composed of swarm robots building structures with custom bricks in three dimensions. This system has similar constraints for their building. For example, robots cannot place bricks directly between two other bricks, robots can climb up or down at most brick height, and they must have an accessible start and end points in the structure. They have written compilers as by Deng (2018) to build the rules for this system with centralized and decentralized control. A decentralized solution presented in Hua (2018) works building the structure layer by layer although not dealing with concave structures or internal holes.

Naz et al. published in Naz et al. (2016a) a parallel, decentralized and asynchronous algorithm for self-reconfiguration using the 2D Catom lattice based robot modules. Their algorithm avoids collisions by having a gap of one empty cell between robots that are in transit using communications. Their algorithm restrict some shapes with local minima as it is a recurrent problem that we try to avoid with the proposed method.

It is important to cite Rubenstein et al. (2012), where Rubenstein et al. propose a parallel, decentralized and asynchronous algorithm to be used with Kilobot swarm system, a well know example of modular robot Rubenstein et al. (2014), to self-reconfigure two-dimensional robots in almost any shape. It has been applied on hardware system with more than a thousand swarm robots. Their system does not need to avoid collisions as it works on a lattice-free system and can construct sparse shapes using what they called collective artificial intelligence. Kilobot swarm system is used also to show error cascades Gauci et al. (2017) that is produced when using large systems and the need for robustness.

4.3/ Introduction to Motion Strategy

For the second step of the previous requirement for modular robots, that is, after identifying the available position how to control the movement of modules from their original position to its final position, many strategies can be applied. It is important to note the challenges that comes with these algorithms. It should be aware of any complication it can meet during its locomotion as barriers and conflicts between modules. Anyway, it is possible to find in the literature some ideas to this locomotion control and here we will list some strategies based in the book wrote by Stoy et al. (2010). All these strategies can be linked with the proposed algorithm in this chapter.

4.3.1/ Stochastic Movements

Depending on stochastic movements to achieve the goal representation is one of the simplest ways to implement a motion strategy but it can require an unpredictable amount of time and requires a big number of spare modules to converge to the final representation. Some works have used this type of strategy as in Murata et al. (1994); Rosa et al. (2006), where the randomness can be combined to other strategies as changing a module direction just when finding a barrier or a border.

4.3.2/ Local Rules

Local rules are closely related to automates where it depends on the state of its neighbors to take any action. A rule can be defined by which connectors are being used at the moment and execute a specific action based on this, for example disconnecting. It can be found in Butler et al. (2001); Bishop et al. (2005); Tolley et al. (2011b) where local rules are used. Local rules have a better performance, as of its monotonous rule creation, at regular structures or, as in chain architecture, to manage the locomotion with its repetitive task.

4.3.3/ Coordinate Attractors

Modules, specially in a lattice based architecture, can easily maintain a coordinate system and update its position after each movement. Using the algorithm proposed in this chapter, the available coordinate can be sent to the spare modules waiting them to come close and dock on these specific positions. The problem comes when modules cannot find a direct way from its position to the final position, as of a case of a hole in the structure, or a barrier. In these cases other strategies should be used in parallel as a path finding, that can demand too many messages in a modular robot network, or stochastic movements in these blocked situations as used in Yim et al. (2001).

4.3.4/ Gradient Attractors

One proposal used by Stoy et al. (2007) is to use gradient attractor. It acts as the previous coordinate attractor and one of its requirements is to know a position that a module can be attracted to.

There are two main difference between gradient attractors and coordinate attractors. The first one is that the gradient attractor uses a vector direction that is transmitted with the final goal position, by this way modules can simply follow the gradient and the vector direction over the system and avoid the use of path finding or stochastic movements for the cases the module finds itself in a border. At each step a module does, it verifies with its neighbors modules which has the strongest gradient and follows the direction of its vector.

The second difference is that it can use a gradient that expires which has the advantage of not recruiting every spare module in the system to the same positions. Even when, in a determined position, a module can be attracted to it, statistically other modules will be required nearby as well, although moving so many spare module of the system to the same location can cause conflicts. The gradient can start with some value and decrease with each step made until gradient is equal 0 and canceled, not attracting modules that are too far from the source signal of the gradient.

4.3.5/ Recruitment

Another possibility of attraction is using recruitment, that is, attracting the first module found to the desired position. A graph parallel between gradient attractor and recruitment can be made, where gradient attraction uses a distributed breath first search and recruitment uses a depth first search. In this way, one specific module is attracted to a determined position following the reverse way of the resulted distributed search tree.

This method, used in Butler et al. (2003), can take a big number of hops to find a spare module, but statistically have less number of messages. It is an interesting method where there will be exactly one module driving to the determined position avoiding collisions.

Finally, recruitment, gradient attractors and coordinate attractors are motions strategies that needs to know the position to where to attract and combines perfectly with the proposed work in this chapter.

4.4/ Model and definition

4.4.1/ Module background

The unit used in this work is also the Catom 3D previous presented. In this work these modules are firstly used in a 2D lattice and then extended to a 3D regular grid lattice oriented along the $\vec{x}$, $\vec{y}$ and $\vec{z}$ axes. The grid is defined by a large set of cells where each cell can only contain one block. Each cell of the grid has 2 different states: empty or filled with a block.

Resuming the modules used in this work, Catoms 3D can only communicate with their direct neighbors. Any communication to distant modules needs a certain number of hops. Modules also have sensors to detect, without communication, the presence of docked neighbors. The following properties are applied to each module:

- Same hardware executing the same program;

- Unique id;
- Is able to get its orientation relative to global referential using embedded sensors;
- Detect the presence of docked neighbors in the adjacent cells;
- Communicate with its connected neighbors.

The position P_c of the cell c in the grid is given by coordinates in $\mathbb{Z}^2$. Each block can be directly connected to up to 4 neighbors in a 2D lattice. We will generalize the method to 3D models at Section 4.7 considering connected overlapping layers using coordinates in $\mathbb{Z}^3$ where a single module can have up to 12 direct neighbors.

Catoms 3D is considered to move using electrostatic forces, that is, a module moves by the aid of another. In this manner, modules should be always connected to perform a move action. However, the algorithm presented here can go beyond module with this type of locomotion. It can be applied to modules that can move freely in different environments, as swarm or underwater robots, or connected and assisted by neighbors. In any case, the only condition for the idea presented here, is for systems in which modules can be blocked and forbidden to conclude a docking in a certain position as pointed by the Rule 1.

Rule [1]. *A module can dock on a cell only if there is no two adjacent cells in symmetrically opposed disjoint planes to that cell that are occupied.*

Rule 1 defines a kinematic restriction applied to 2D and 3D lattices. Indeed, we assume there must be enough space to allow modules to dock at a specific cell. Figure 4.3 shows an example of a case that can be restricted by Rule 1 if modules A and B are part of different contiguous rows.

From a practical point, being $\mathcal{G}$ the representation of the goal shape, we assume that each module B_i stores locally a copy of $\mathcal{G}$.

Considering the modules in a FCC 3D lattice which is made of a regular square grid on each horizontal layer $\overrightarrow{XY}$ and interleaved modules on $\overrightarrow{XZ}$ and $\overrightarrow{YZ}$ axes, it permits modules to have up to 12 neighbors creating a dense organization of quasi-spherical modules (Figure 2.7). In a FCC lattice, a module may have up to four candidates neighbors to its top position, and up to four candidates to its bottom position as well. To solve this ambiguity, we use the same coordinate system as defined in equation 2.3 in chapter 2, where a FCC lattice is translated in a regular grid and takes the module in its top position corresponding to $P_i(x_i, y_i, z_i + 1)$.

4.4.2/ Problem definition

Initially, a single seed module S_0 is assigned with the target shape $\mathcal{G}$ and a position in the target object. S_0 is responsible to attract other modules to dock in its connectors that must be filled according to $\mathcal{G}$ and must ensure that all remaining positions in $\mathcal{G}$ can still be reached. When a new neighbor is docked and connected, S_0 sends $\mathcal{G}$ and the neighbor relative coordinate. One after another, new modules in the structure attracts new neighbors to build a dense target object.

Modules should have enough memory to store a description of the goal shape $\mathcal{G}$. In this sequence, it can take advantage of the representation CSG4PM presented in chapter 3 that fits all requirements for a description shape for programmable matter.

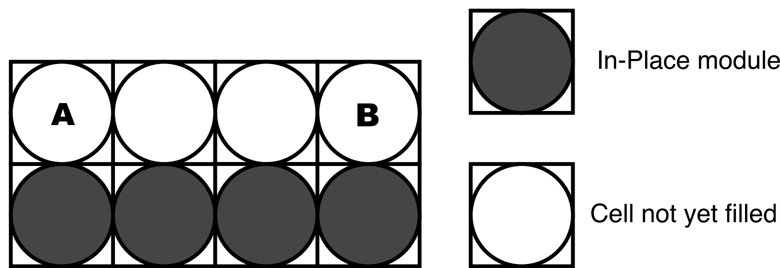


Figure 4.3: Modular robots and kinematic constraints in a row: only one contiguous line of blocks can be allowed. Modules A and B cannot be part of different contiguous line to avoid positions to get blocked.

The robots can attract others only when this action will not cause future positions to get blocked and lead to holes in the system. For example, in Figure 4.3 we show that if cells A and B becomes occupied before the cells between them, creating two contiguous rows, will lead to a future position getting blocked. At each row, just one contiguous line of robots should be possible to avoid blocked positions. In another general example, Figure 4.2 shows the construction of a mug without an algorithm that avoids these blocked cells and the catastrophic result it can generate.

There are two main conditions to avoid unwanted holes in the final object:

- only one line of robots is allowed by contiguous row;
- only one set of robots is allowed by contiguous plane.

4.5/ Distributed Self-Assembly Algorithm

An algorithm is necessary in order to have a construction order that creates structures without empty spaces. In this section we start presenting the idea for constructing a plane without these empty spaces, and later an extension of this same idea to different algorithms, depending on its goal morphology, to construct a 3D object.

4.5.1/ 2D Self-Assembly Algorithm

The general idea for self-assembly adopts the following steps: first, the goal object $\mathcal{G}$ is encoded using the CSG4PM method and transferred to a seed module. From the seed module, it can try to attract other modules on its connectors according to $\mathcal{G}$. For each of its connectors, it requires specific rules that can be solved using the goal shape description $\mathcal{G}$ and neighbor-to-neighbor communication.

First, lets introduce the rules for attracting modules in the same row, that means, in the positions $P_i(x_i + 1, y_i)$ and $P_i(x_i - 1, y_i)$ relative of B_i. Later, we will advance to the rules to create a new line with the notion of line seed that will reuse the concept of line attracting to east and west sides.

The rules responsible to attract a neighbor module on the same row of the current module are $WestAttraction$ and $EastAttraction$ rules:

$$
\begin{aligned}
WestAttraction(B_i) :- \; & \langle x_i - 1, y_i \rangle \in \mathcal{G} \\
& \wedge [\, \langle x_i - 1, y_i - 1 \rangle \notin \mathcal{G} \\
& \vee (\langle x_i, y_i - 1 \rangle \in C \wedge \langle x_i - 1, y_i - 1 \rangle \in C) \\
& \vee (\langle x_i, y_i - 1 \rangle \notin \mathcal{G} \wedge CWHole(Bi, \langle x_i - 1, y_i - 1 \rangle), C) \\
& \vee (\neg CWHole(Bi, \langle x_i - 1, y_i - 1 \rangle, \mathcal{G}))\,]
\end{aligned}
$$

$$
\begin{aligned}
EastAttraction(B_i) :- \; & \langle x_i + 1, y_i \rangle \in \mathcal{G} \\
& \wedge [\langle x_i + 1, y_i + 1 \rangle \notin \mathcal{G} \\
& \vee (\langle x_i, y_i + 1 \rangle \in C \wedge \langle x_i + 1, y_i + 1 \rangle \in C) \\
& \vee (\langle x_i, y_i + 1 \rangle \notin \mathcal{G} \wedge CWHole(Bi, \langle x_i + 1, y_i + 1 \rangle), C) \\
& \vee (\neg CWHole(Bi, \langle x_i + 1, y_i + 1 \rangle, \mathcal{G}))]
\end{aligned}
$$

$$CWHole(A, B, \mathcal{E}) :- \text{it exists a clockwise rotating path from A to B in } \mathcal{E}$$

These two rules depends on the definition of $CWHole$ that takes as argument two coordinates and the medium $\mathcal{E}$ which it is applied that can be:

- $\mathcal{G}$ when it is applied to the map of the goal configuration.
- C when it is applied in the current configuration, that means, only for modules that are already in place.

As modules do not have direct communication in its diagonals, as there is no physical contact between these modules to establish a communication, a way to verify if there is an internal or external hole is applying the rule $CWHole$ that can be also described in form of algorithm in Algorithm 5. This algorithm follows the positions that are in the border of the object and can be used in a distributed environment as it only depends on the direction that the rotation is driving to decide the next direction it will target. The angle of these previous and next directions can be summed up to verify if the clockwise rotation is part of internal or external hole.

The rule $WestAttraction$ is colored as it will be explained in detail. From this rule, the rule $EastAttraction$ is self-explanatory as it is the same but for the other direction.

Using Figure 4.4 as a guide image, lets check the rule $WestAttraction$ line by line. The first line is a direct condition that, to attract a module to its West side, this position should be contained in the goal configuration. Another requirement of the self-assembly when constructing to the West side is to be sure the module in the South-West position will not get blocked. Following these conditions, four cases can be applied:

- The green line means there is no module on the goal configuration in its South-West coordinate and in this manner attracting a module on its West side will not block this specific position. This condition depends only on the goal shape description and do not require any communication.

Algorithm 5: Algorithm input and function detailed for border following over module on position P.

Input:
$\mathcal{G}$ // global goal shape
P // position of B_i
$dir \in \{N, E, S, W\}$ // initial direction of following
$searchDir \in \{CW, CCW\}$ // direction of rotation

```
1  Function borderFollowing(P, dir, searchDir):
2      j ← PrecDir(dir, searchDir); // predecessor direction
3      for i ∈ [0...3] do
4          Q ← P + NextDir(j, searchDir);
5          if Q ∈ G then
6              borderFollowing(Q, j, searchDir);
7              return;
8          end
9          j ← NextDir(j, searchDir);
10     end
11 end
```

- The blue line depends on communication to verify if the same South-West is already in place in the case it exists in the goal configuration. To conclude this verification, the module responsible to attract in its West connector should use the module at its South side as a bridge for the communication with the South-West coordinate module. In this way, when both modules are in the goal configuration and both exist in the current configuration, the responsible module can be sure that attracting to its West connector will not block any other position.

- Red line of the $WestAttraction$ rule describes the case when the position in the South of the current module does not exist in the goal configuration and will not be able to act as a bridge to check the South-West coordinate. In this case, if there is a clockwise rotating path in the *current* configuration from the current module to its South-West position, it can attract to its West connector. Note that the rotating path should exists in the goal and current configuration, and a message should be sent through the modules existing in the border until arriving in the South-West coordinate. In the case of modules that are in the goal configuration and not yet in the current configuration, the modules create a queue of message waiting for the next module of the border to be docked in order to continue the border following. When this message arrive to the South-West module, it sends a response in the opposite direction for acknowledge and the self-assembly can continue its construction.

- Yellow line describes also the case the module in the South position does not exist to serve as a bridge to its South-West coordinate. However, the responsible module can verify it is a case of external hole and can immediately attract on its West connector without blocking the South-West position, as the South row will be filled from the current one.

Supposing all modules drawn in the Figure 4.4 try to attract on its West connector, the color of the module in the figure match the rule color. Only black modules have no color

associated as they do not have any module on its West side in the goal configuration.

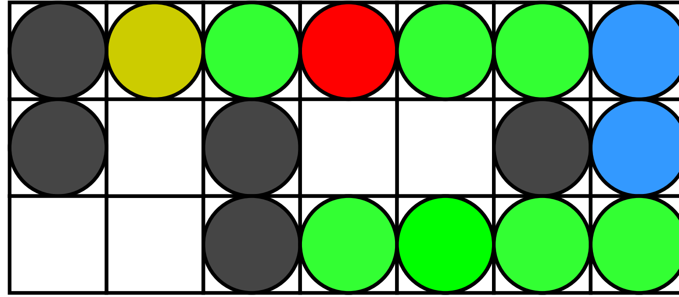


Figure 4.4: A set of modules where the colored modules use the definition of the Rule $WestAttraction$. The color is according to which condition of the rule that can be applied.

After verifying the basics to self-assembly modules in a single row, lets analyze how to start another row that will reuse the same rules.

A typical case of forbidden position is created if a row on the goal shape description has more than one contiguous row of modules on the system. To avoid two separated modules leading to an empty position between them, only one module, that is called seed, is responsible for attracting a module to a North or South line.

The $isNorthSeed$ rule is sufficient to a module verifies if it is the responsible module to attract on its North connector in order to start a new row.

The following rules define a module B_i in its position as a North or South seed:

$$
\begin{aligned}
isNorthSeed(B_i) :- \; & \langle x_i, y_i + 1\rangle \in \mathcal{G} \wedge \langle x_i, y_i + 1\rangle \notin C \\
& \wedge \left(\langle x_i + 1, y_i + 1\rangle \notin \mathcal{G} \vee \langle x_i + 1, y_i\rangle \notin \mathcal{G} \right) \\
& \wedge \neg CWHole(Bi, \langle x_i, y_i + 1\rangle, \mathcal{G})
\end{aligned}
$$

$$
\begin{aligned}
isSouthSeed(B_i) :- \; & \langle x_i, y_i - 1\rangle \in \mathcal{G} \wedge \langle x_i, y_i - 1\rangle \notin C \\
& \wedge (\langle x_i - 1, y_i - 1\rangle \notin \mathcal{G} \vee \langle x_i - 1, y_i\rangle \notin \mathcal{G}) \\
& \wedge \neg CWHole(Bi, \langle x_i, y_i - 1\rangle, \mathcal{G})
\end{aligned}
$$

Modules can decide if they are seed without any communication but using the shape description of $\mathcal{G}$. To decide if a module is a seed, it applies the Rules $isNorthSeed$ and $isSouthSeed$, for North and South seeds respectively. The first rule elects the rightmost module that has its North position inside the shape description as seed. A row can have more than one seed, for example seed modules B and C in Figure 4.6. Also, a module can decide if it is a seed if its North-East position is not inside the shape description and its North position is. The same can be done for their opposite direction, a module can be a seed for its previous row. In this case, they use their South and South-West positions as reference.

The general idea of $isNorthSeed$ is to select the East-most possible module of the row as a seed. Again, we will focus the analysis line by line only over the $isNorthSeed$ rule as the $isSouthSeed$ is symmetric opposed to the first.

- The first line of the rule makes the basic verification that the North coordinate have the presence of a module in the goal shape configuration and that this position is not already occupied. This last verification is due to avoid the case that a row has been created in the direction North-South and the North seed would not be a seed as the North row already exists.

- The second line of the rule has two main verification which look for an empty cell near its North coordinate. It means in blue that the North-East cell is empty or in green that the East cell is empty.

- In the last line, after verifying if the North or North-East cell is empty, it should also verify that is not in an internal hole. It uses the same Algorithm 5 seen before, but it does not need to send any message as it can make this verification only using the target description. Thereby, Figure 4.5 shows a case that apply the blue and green conditions. In this same Figure, the module with an X draw inside it would pass the green verification but is not a seed as consequence of being part of an internal hole.

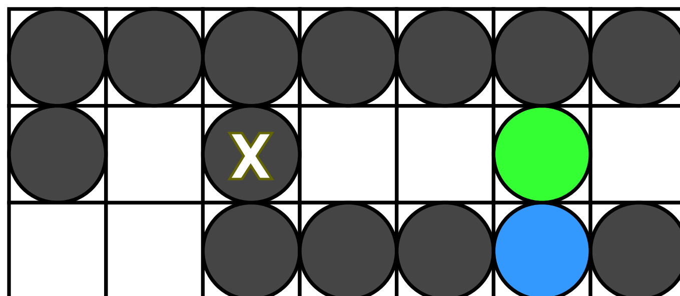


Figure 4.5: The image shows colored modules that used the rule $isNorthSeed$ and are responsible to attract a module on its North connector. Black modules cannot call others to their North connector as they are not seed.

Figure 4.6 shows another example of this algorithm seed selection. In light-gray are drawn seeds for their corresponding north position. Module marked as B is a seed due to the fact that its north position is in the shape description and their north-east position is not. Module C is seed as it is the rightmost module of the line and its north position is inside the shape.

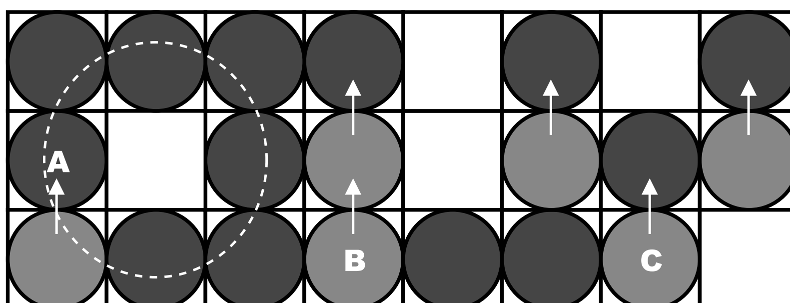


Figure 4.6: Light-gray modules represent seeds which are responsible for attracting a new module on its north row. Each contiguous row can have only one seed to avoid blocked positions. The rightmost module is alway elected, except for Module A, since it is part of an internal hole.

Module A on Figure 4.6 is not an elected seed as, in this case of internal hole, it would have two seeds for the same north row and may lead to a situation as foreseen in Figure 4.3. The algorithm verifies that is a point of row merge checking if it is a case of internal or external hole. It is done by running the Algorithm 5) and counting the number of turns it made to complete the hole. No communication is necessary as it only needs the scene description.

This algorithm creates a pattern that can fill diagonals in parallel as can be seen in Figure 4.7. The white positions can be filled with no dependency between them. Resuming the attraction law, a module can attract another to its West side when the cell located on its South-West is in its final state. Modules in a regular grid do not have direct communication with modules in their diagonal, as the case of the South-West location. A module on its South can be used as intermediate to communicate when this South module is present. Otherwise, if there is no module on its South coordinate, it can be a case of internal hole and the module sends a message over the inner border, using the Algorithm 5 to communicate and have the certitude that the module is already in place before sending an attraction signal to its West connector.

The same method is applied when attracting a module on its East side but with the symmetrically opposed verification. It is based on the presence of the module on its North-East location, when it is in $\mathcal{G}$. If the North-East cell is not inside the shape description it can attract a module immediately. The Algorithm 6 shows the rules to attract neighbors on all its four sides.

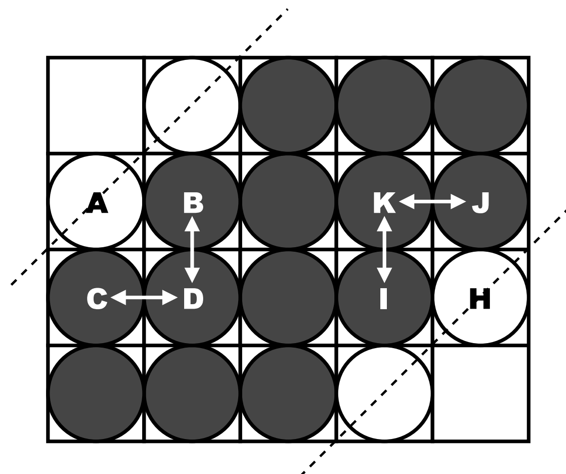


Figure 4.7: Cells that should be in place before attracting neighbors. Before attracting a module in cell A, modules B and C should already be in place. B and C are not direct neighbors, so to communicate they need a common neighbor module to know if both are already in place. The construction method fills space by adding modules along the diagonal line in both directions simultaneously (NW and SE).

When a module finally docks on its final position according to the target description, the following data is transmitted:

- Target scene description
- Relative position over the target scene
- Queue of messages waiting for the *BorderFollowing* algorithm

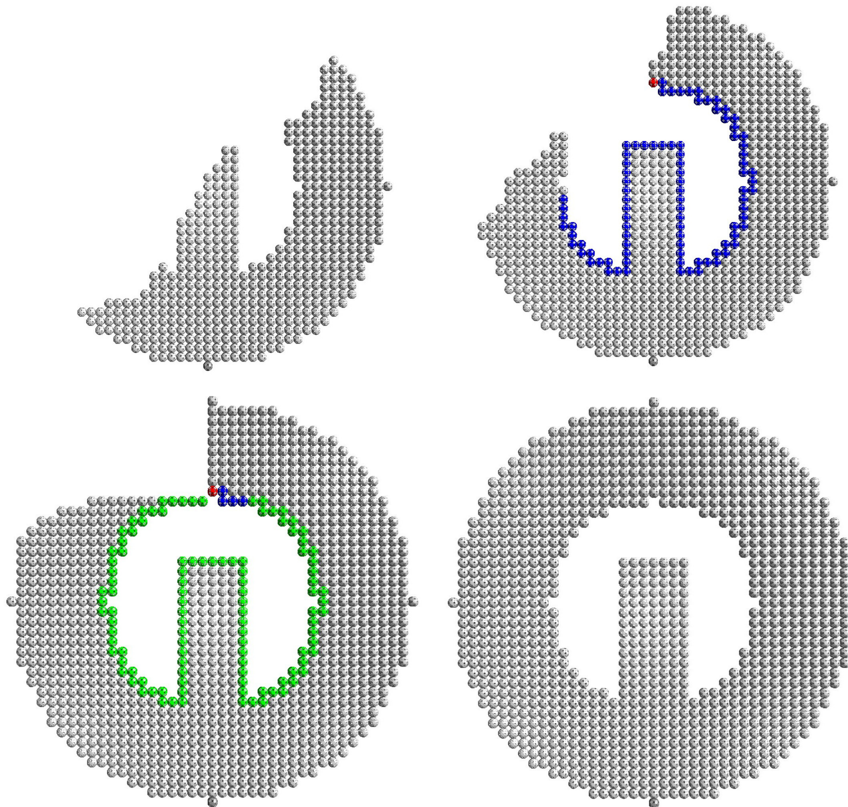


Figure 4.8: Example of constructing a model with internal hole in a two-dimensional square lattice. The task is parallelized as many positions can be filled at the same time. The modular robot drawn in red detects a section of line merge made by the internal hole. It sends a message following the inner border (blue modules) and waits for an unlocking message. At the moment the message arrives in the south-west position of the red module, the module sends a response in the reverse path (green). This verification is necessary before a module docks on the red module west side in order to have no position blocked.

Figure 4.8 shows the construction of an object with internal hole from the initial module which is the South-most position. In this example, it is possible to notice the diagonal pattern the algorithm creates. As North seeds are located in the East, they form a diagonal from West to North. The red module detects it is in a merge point and awaits to be sure the other side is in place before continuing. Modules change their color to blue to show the route of messages from East to West, and to green when they are transmitting a response. The message for the $BorderFollowing$ follows the internal border of the hole and requires only as extra information the previous direction of the message. After the message has returned, the system resumes its assembly.

Algorithm 6 annexed shows the rules of attraction described in algorithm form. There we can see how these rules goes from its logical state to its implementation and their difference.

4.6/ Experimental Evaluation

Figure 4.9: Three classes of target structure: Power Button, Letter C and Bumpy respective representations.

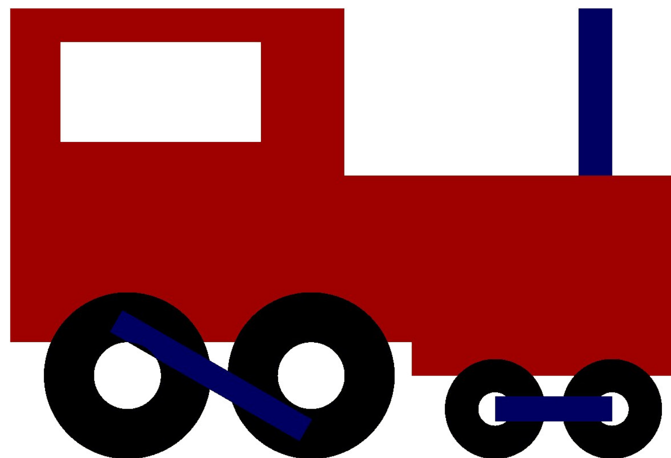


Figure 4.10: A side view of a locomotive used for the experimental in two dimensions.

We have implemented and evaluated the algorithm using VisibleSim Dhoutaut et al. (2013), a C++ simulator for modular robots. VisibleSim enables the users to run event-based simulations. An event is a task executed by the module simulating one of its actuator, for example attracting a new robot or sending a message. Messages and events can be pushed into the system with communication and actuator simulated delays and the random seed can be controlled in order to study one specific execution. In order to test our algorithm, we have used random delays with a large range for each message that change the orders of docking thus making each evaluation executed on the system unique.

We studied the number of messages by the scaling factor and the number of simultaneous attracting positions for four classes of target structure (called “power button”, “letter c”, “bumpy” and “2D locomotive”) presented in Figure 4.9. We made a more realistic experimentation using a locomotive 3D model shown in Figure 4.14, described by 61,780 Catoms. In order to have a proper comparison with the other target structures we used a lateral view of the locomotive for a 2D representation in Figure 4.10. Efficiency of our algorithm is affected by geometrical and topological characteristics of the goal shape:

- The “power button” model proposes a simple case of internal hole (homeomorphic

to a torus) with concave parts;

- The “letter c” model is a concave shape homeomorphic to a sphere;
- The “bumpy” model contains several internal holes at once;
- The “2D locomotive” groups many complex areas with holes, convex and concave parts.

For each representation three versions were used, each one scaling the size of the structure by two. As for the representation of the power button, we start with a structure using 815 robots, scale it to $3,318$ and $13,233$ modules. Through our experiments, we show the effectiveness of our method in terms of number of messages and number of available docking positions at the same time.

4.6.1/ Messages Evaluation

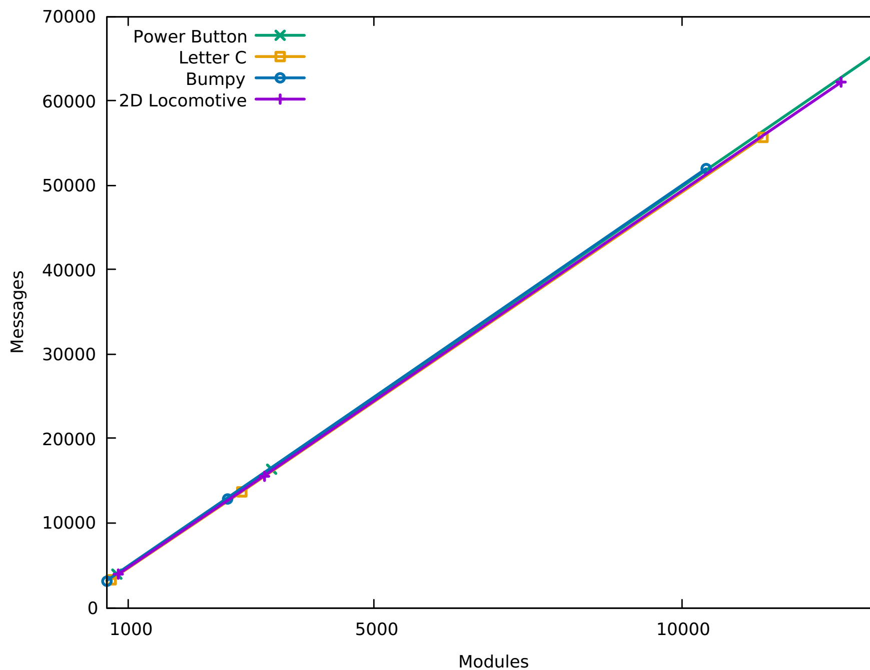


Figure 4.11: Number of messages for four topologically different representations on three different scales. The number of messages remains linear and similar across all representations.

The number of messages scales linearly with the number of modules as shown in Figure 4.11 with a proportion of 5 messages per module. The messages on the system are composed of an initial communication, where a recently docked module receives a message from its parent module. The parent module sends initial information as relative position, the state of neighbors, and a queue of messages for synchrony that have been waiting the docked module. This initial communication increases the number of messages by $2n$, where n is the number of modules, and is composed of one message when the module is ready and thus ask the initial information and a second message with the response.

A message is also needed to check if the West or East position can be filled, which uses a module as a bridge for the communication as showed in Figure 4.7. In the example, when module B wants to attract a module to position marked with A, a message is sent to module D, that should wait a message from C and send a response to B. In this way, more $3n$ messages are required.

Communication is used when the model contains internal holes. For example, the bumpy structure has more internal holes and requires a greater number of messages, but no significant difference was found between the four classes of the target structure. Therefore, by experimental results, the proposed algorithm requires an average of $5n$ number of messages as is verified by the chart in Figure 4.11.

4.6.2/ Available Docking Positions Evaluation

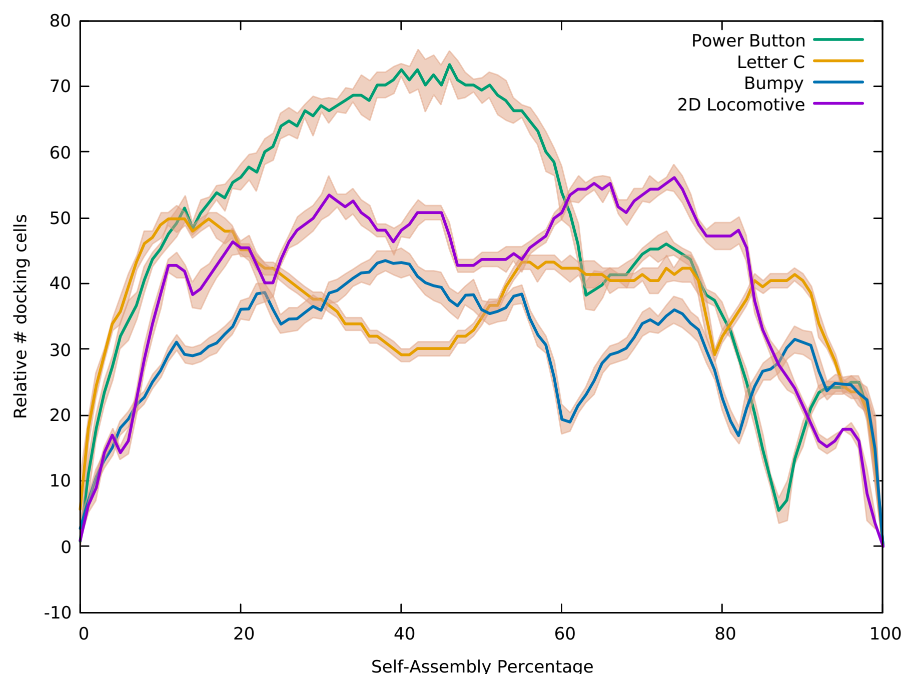


Figure 4.12: Relative number of available docking positions over the percentage of construction. It shows parallelism of the method by having as many docking positions as possible at the same time. The thick lines represent the standard deviation in a certain construction percentage.

The number of available positions attracting modules is an important parameter of the efficiency of the algorithm. It can speed up the convergence to the final object as many modules can arrive at the same time in different positions. The movement of free modules to their correct position has a large impact in the overall time of the self-assembly.

North seeds are located in the West-most possible location while *South seeds* are located in the East side. When a *North seed* attracts a module to its North connector, it can start attracting to its West and East connectors. Supposing a square, the remaining modules of the line are all located to its West, thus the algorithm has an effect to attract new modules in the diagonal of the target structure. Placing the initial seed at a position relative to the middle of the structure can make the convergence twice as fast as there are two

diagonals, with *North* and *South seeds* present, it can have one diagonal facing the West side and the other facing East. The number of available docking positions is related with the goal morphology and the position of the initial seed.

In order to have a proportion independent of the scale of the model, we define the relation over time $R(t) = \frac{D(t)}{\sqrt{N}}$, where D is the number of available positions and N the total number of modules. Figure 4.12 shows the relative number of available docking positions by the construction percentage. As power button can have up to 3 diagonals being filled at the same time, because of its three independent columns, it has between 10% to 60% more available docking positions.

Power button line in the chart has it first bottom, when the number of available docking positions decreases steeply, in around 60% of construction when the middle branch of the model has finished. The second bottom at around 85% is due to the wait caused by the internal hole.

In the points where number of available docking positions goes down, as for the Bumpy line, is when the self-assembly have two sides being constructed at the same time, mostly because of an internal hole that are visible seem at 60%, 80% and 90% construction points, and, in the end, one of these sides should wait the other and merge in just one. This waiting has as consequence to lower the number of available positions but then resuming to the same number quantity of available positions.

As for 2D locomotive, it has low percentage of internal holes size considering the total model. In this way, the synchrony for internal holes does not affect significantly the chart.

4.7/ 3D Self-Assembly

The Three-Dimensional Self-Assembly involves many new challenges than the previous work in two dimensions. The main point around the new challenges is that in two-dimensions the number of neighbors in a position is up to 4 neighbors instead of 12 in our three-dimensional FCC lattice. The solutions presented here are designed for FCC lattice but can be easily extended to other regular lattices.

In three-dimensions, the rule of blocked position remains the same of the two-dimensional Rule 1. Figure 4.13 can show how this rule is applied in three-dimensions where neighbor modules cannot be in two separate opposed planes at the same time.

Aside from having more neighbor modules that could block a determined position, internal holes in three-dimension objects fill a new category in self-assembly planning. For example, for two-dimensional objects with internal holes a walk through the internal border have just one way to follow but in three-dimensions there are myriads of possible ways.

Depending on the object, internal holes can be avoided as is the case of object presented in Figure 4.14. A construction oriented as it appears in the Figure, going in the direction bottom-up from the wheels until the top of the locomotive would need many synchronizations of planes since the construction starts from 4 different points (wheels) that merges and a point of merge at its window. In the specific case of the locomotive presented, constructing from its side, as the locomotive would be lying down, would avoid the need to synchronize separated planes and make the self-assembly simpler.

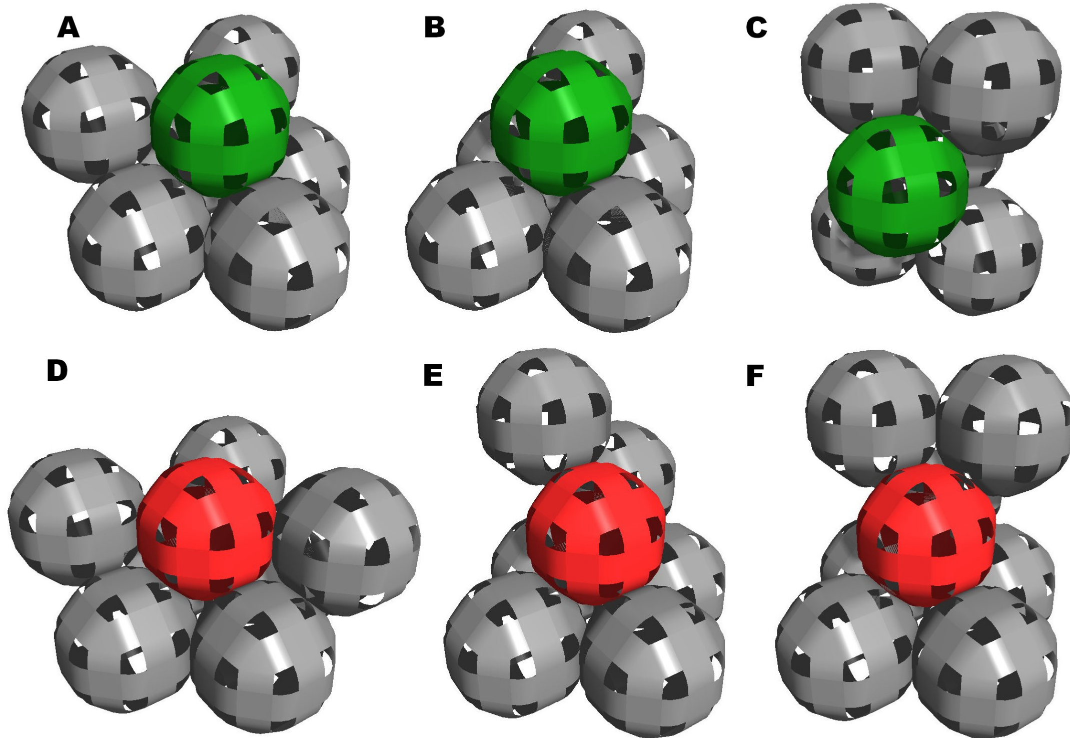


Figure 4.13: Some examples of modular robots configurations that express if the center position (colored module) can be docked or not. Modules in green represent positions that can be docked with the surrounding modules. Red modules represent positions where docking is not possible due the space needed for module movement.

However, many types of objects does not have holes at all or may contain intrinsic holes that it is not possible to avoid. In the following sections I will present different solutions to the 3D Self-Assembly Planning.

4.7.1/ 3D Self-Assembly Layer by Layer

In this subsection, a solution for a three-dimensional lattice self-assembly is proposed. The solution presented here is oriented for simple goal shape objects without merging planes, that means, without three-dimensional holes and starting from one single point.

Following the Rule 1, a given coordinate cannot have two different cells occupied in opposite planes. A module placed near a free position can block this free position and should be avoided. That means the construction should follow a certain order to avoid blocked coordinates, for example a bottom-up order. Let's define a layer as a plane in $\vec{xy}$ axis. The idea presented here is to assure all modules in one layer are already in place before starting the construction of the next layer.

A solution to this problem is to construct the object layer-by-layer in a bottom-up order. A secondary problem is, if a contiguous layer begins to be constructed from two different points, there may be a blockage somewhere in this same layer during the self-assembly. Based on this, modules have to agree with a unique initial point for one contiguous plane.

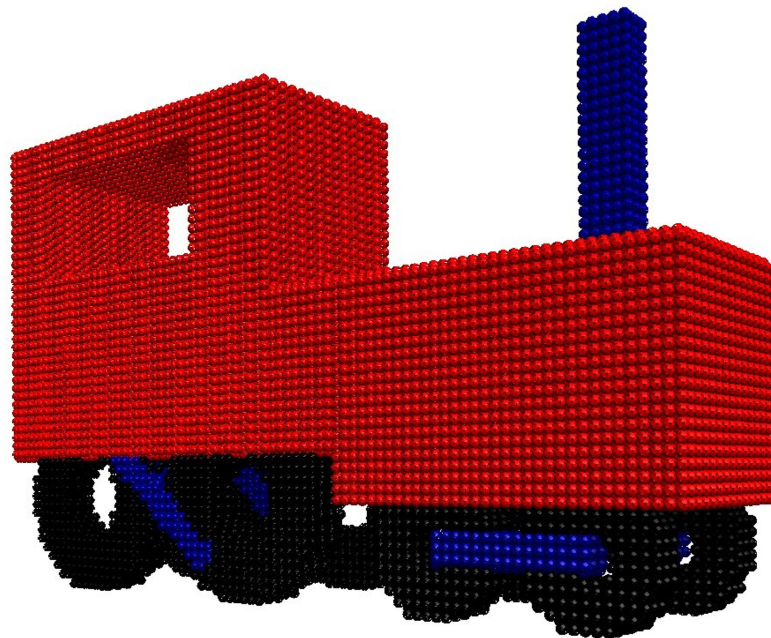


Figure 4.14: Locomotive example made by 61,780 Catoms constructed using the distributed 3D self-assembly algorithm. The self-assembly can be done without any position being blocked during the construction.

Let's call *3D seed* the module that is responsible for attracting another to the next plane. Each module can locally determine if it is a *3D seed* using only the $\mathcal{G}$ description, without neighbor-to-neighbor communication. The module located in the lowest x and y coordinate with a module in its next plane is the *3D seed* and it can be checked with the goal shape description.

Modules can use the same Algorithm 5 to follow the border and pick the module with the lowest value on $\vec{x}$ axis and then with the lowest value on $\vec{y}$ axis. There are two possibilities from this method, that the next layer is bigger or smaller than the current layer and with that choose to follow the outer border of the current plane or the outer border of the next plane. Note that this time the Border Following algorithm is applied to $\mathcal{G}$, that means, it does not send message to neighbors and treatments are local.

The first case is when next plane area is smaller than the current plane area, so there is a border of the next plane inside the current plane. For each module in the current plane they check if the respective next plane module is on a border and execute the Border Following algorithm in the border of the next plane to check if it is the lowest module with a filled position under it. With this verification, modules can decide if they are a *3D seed*.

If no *3D seed* module were found using the previous method, so this is a case that the next plane is bigger than actual plane, or explaining in other words, there is no edge of next plane inside the current plane. Thus, one module of the current plane border should be chosen. A Border Following over the current plane in G is executed and the lowest position with a module on its top is actually elected a *3D seed*.

After identifying the *3D seed* module of the plane, it should wait its current plane to have finish before attracting a module to its next plane, therefore a consensus on when the plane has finished is necessary. Using the given self-assembly algorithm, the order of

docking and the module that *attracts* generates an implicit spanning tree. This spanning tree is used to send data from the leaves to their parents nodes that do it recursively when all its child nodes have sent a confirmation. The last module receiving the confirmation is the layer parent module, which is the first module in the current layer. The layer parent module resends a message to all modules of the current layer to confirm the plane has finished.

When the *3D seed* module receives the confirmation that the plane has finished it can attract and allow docking on its next layer interface. This algorithm is executed recursively until all layers have been filled and the object is completed.

4.7.2/ Multilayer 3D Self-Assembly

Another approach to 3D Self-Assembly Planning is to rely on more communication to increase the number of available docking positions. In mind that the time to modules to move to a certain position will be certainly higher than the time of message passing, an idea to use more information exchange to have more available robots trying to dock in the free positions can reduce the overall self-assembly time .

This approach is similar from the previous identifying where a layer should have only one initial point and the *3D seed* module is chosen in the same way, but does not have to wait the entire layer to be constructed before the *3D seed* module allow docking in its top interface.

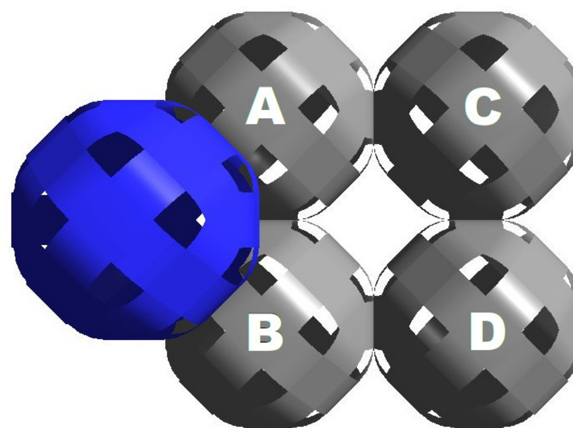


Figure 4.15: Neighborhood necessary for the blue module be able to attract a module to its right side connector.

This is done basically relying on the other connected layer and waiting for the modules in the same area to be in place. This means a layer saves the direction from where it has been initially constructed and Figure 4.15 exemplifies the necessary neighbors that must be in place before attracting a module to its East connector.

Sensors are enough to identify if modules with distance 1 (modules A and B) from the blue module are already in place, but the blue module does not have direct sensor and

communication to modules placed in distance 2 (modules C and D) from it. Therefore, the blue module should rely on some intermediate modules to get the necessary information.

In this method, following the example in Figure 4.15, the blue module waits modules A and B to confirm all modules in the region are done. For example, module A is responsible to confirm that module C is docked when it is present and module B responsible for D. In the case there is no intermediate module A or B and, in the shape description there should be modules in coordinate C or D, another path to communicate to C and D should be found. In these exception cases, the blue module waits the entire parent layer to be completed. The parent layer sends a confirmation to signal that it has ended and construction can continue in these blocked states.

A symmetric solution is done when attracting in the other module connectors as North, West and South connectors. We have seen so far how the algorithm works to create a distributed construction order in a layer and when and where start a next layer.

This method can improve the number of available docking positions and accelerate the self-reconfiguration in the cost of more communication as it does not need the previous layer to be completely completed. As the previous approach maintains the number of available docking position almost the same for every layer, this new approach tries to construct many layers at the same time and guarantees that all modules can fit in the structure.

4.7.3/ Experimental Evaluation

As the previous methods present a 3D self-assembly order that work on similar objects, a comparison of both methods in terms of number of messages, time steps and number of modules is presented to show the pros and cons of each solution. The experiments in this section were executed in different formats that we will analyze in detail. Primitive types as cube, cylinder and sphere objects are part of these tests as they are often included in more complex objects. A hollow version of these primitive objects were also tested as they need a special treatment for their internal holes, represents the possibility to create lighter models and how the algorithm would behave in these cases. Later, more complex objects were also analyzed as is the case of the 3D locomotive example in the Figure 4.14 and a pyramidal object.

In this section, there are charts comparing the number of modules by the number of messages and the number of time steps.

The intrinsic parallelism in modular robots, where each module moves and act independently, can be better expressed using time steps.

In this context, one time step is equal to the amount of time a module need to dock in a free position. If there are many free positions available in a certain time, in one single time step all modules dock in the current available positions. Normally, a module needs a considerable amount of time to dock in a given position, therefore every message is supposed to be sent with a negligible amount of time and do not interfere in the time step calculation.

4.7.3.1/ CUBE

The cube is a simple object where we can focus on the observation of some aspects of the construction in an FCC lattice.

Firstly, in the chart of Figure 4.17 let's start analyzing three equivalent solid cubes and the number of time steps needed to construct an object.

The first cube is analyzed using the Layer-by-Layer algorithm. The other two use the Multilayer algorithm with the same cube, but the bounding box of the cube is slightly displaced in the xy plane. This displacement has consequence in the self-assembly, as the module in top of the layer seed has an offset because of the FCC lattice, what lead to a difference in the choice of *top* module and can be visualized in Figure 4.16. In the chart of Figure 4.17, Multilayer Cube 1 represents the left cube in Figure 4.16 and Multilayer Cube 2 the right cube.

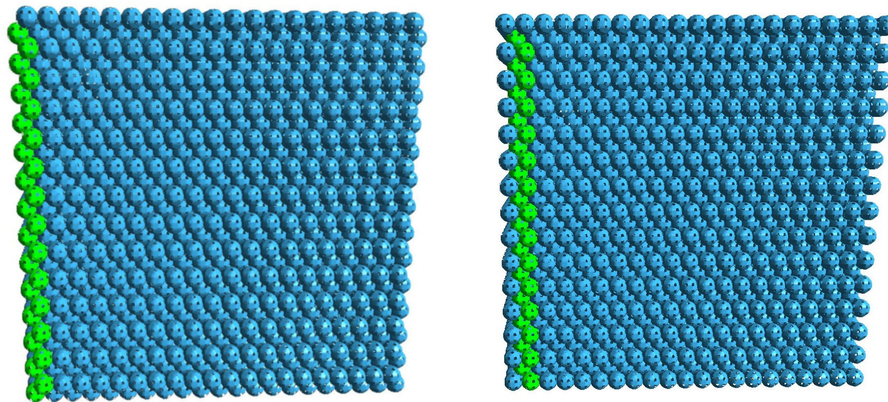


Figure 4.16: In green its represented the layer seed modules. FCC lattice can give different order of layer seed depending on the origin module placement in the coordinate system. This different order has consequence in the number of time steps needed to self-assembly a final object.

In this chart we can notice that in the worst scenario for the Multilayer Algorithm, of the Multilayer Cube 2, it is approximately 2 times faster than the Layer-by-Layer algorithm. At the best scenario, of Multilayer Cube 1, it can converge to the final cube with 16 times less time steps. The placement of the cube in the world coordinate, as for Cube 1 and Cube 2, plays an important role here. Using the FCC lattice, the *next layer seed* is chosen when there is a module in its top position. Depending on the coordinate assigned to the first module in the system, the top module, the one that contains the same *x* and *y* coordinates and its *z* incremented of one, can be different. This is a consequence of the difference of properties defined when the *z* coordinate is even or odd. In the case of Multilayer Cube 2, to the *next layer seed* be able to attract a module on its next layer, it should wait the module on its south position to be placed. This means waiting the entire row it is included to be filled and then its previous row, that starts in the West side filling in direction to the East side.

In any case, the Multilayer Algorithm shows to have a better performance in terms of time step compared to the Layer-by-Layer algorithm, but we should also check what is the impact of this algorithm in number of messages.

Figure 4.18 shows the number of messages required for each algorithm. In this chart the

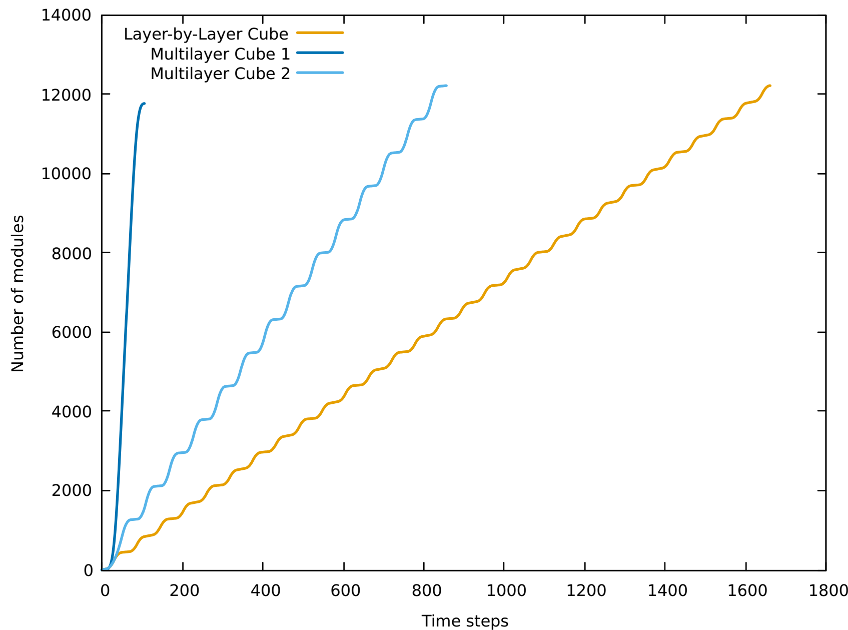


Figure 4.17: Number of available docking position during the progression of 3D self-assembly. Multilayer Algorithm presents a peak of efficiency at a regular phase in the construction.

number of modules is different from the Multilayer Cube 1 and 2. This is due to the fact that, as the bounding box is displaced and the position in the exact center of the module is taken to verify if it is in the target representation, a layer of modules can be missed.

The number of messages follows a pattern that grows and decreases in each layer of the Layer-by-Layer algorithm, that can be predicted as in the end of a layer construction, fewer positions are available to docking, therefore, fewer messages are exchanged. It correlates with the chart for time steps that also presents the same pattern.

A linear line with low deviation error can be traced to represent the overall number of messages of the algorithm. Layer-by-Layer algorithm requires approximately 10.5 messages per module for its self-assembly planning while Multilayer Algorithm needs an average of 23.5 messages per module.

A lighter version of a cube is also evaluated. It is composed of a hollow cube, that means an object with empty space inside. As internal modules are not visible, creating a hollow version of objects can be useful in specific tasks, for example, to create lighter modules. Figure 4.19 shows four steps of the hollow cube self-assembly using the Multilayer algorithm.

The chart of Figure 4.20 compares the time step of the hollow and solid cube representation using the Layer-by-Layer and Multilayer algorithm. The Multilayer algorithm for the hollow representation requires more time steps than the solid cube due to the need to synchronize the internal hole in many layers of the hollow cube. It is also important to notice that, although there are fewer number of modules, the time steps remains the same for the hollow and solid configuration using the Layer-by-Layer algorithm. It can be explained because each layer can fill a layer diagonal in parallel and the absence of modules in the middle of this diagonal would not affect the number of time steps.

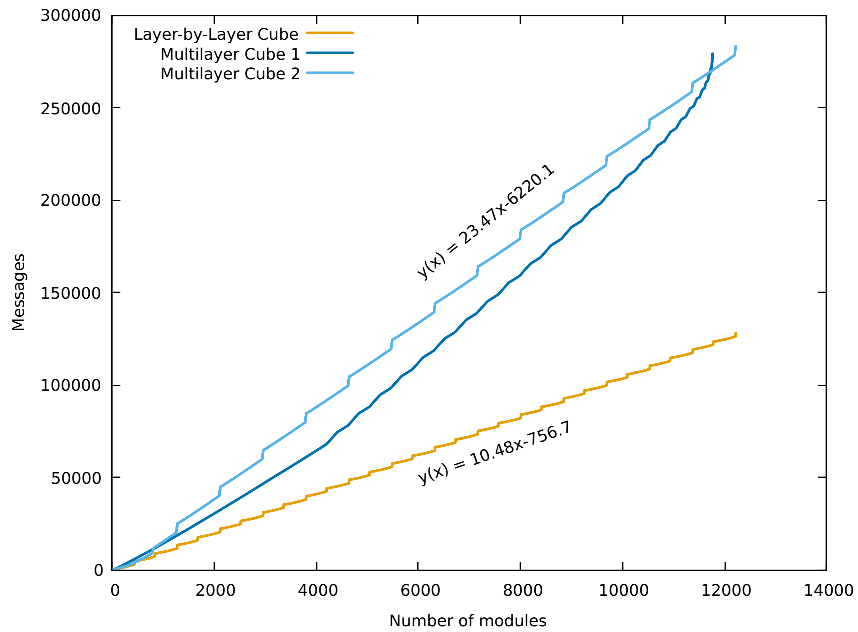


Figure 4.18: Number of messages required to self-assembly a cube using two different algorithms.

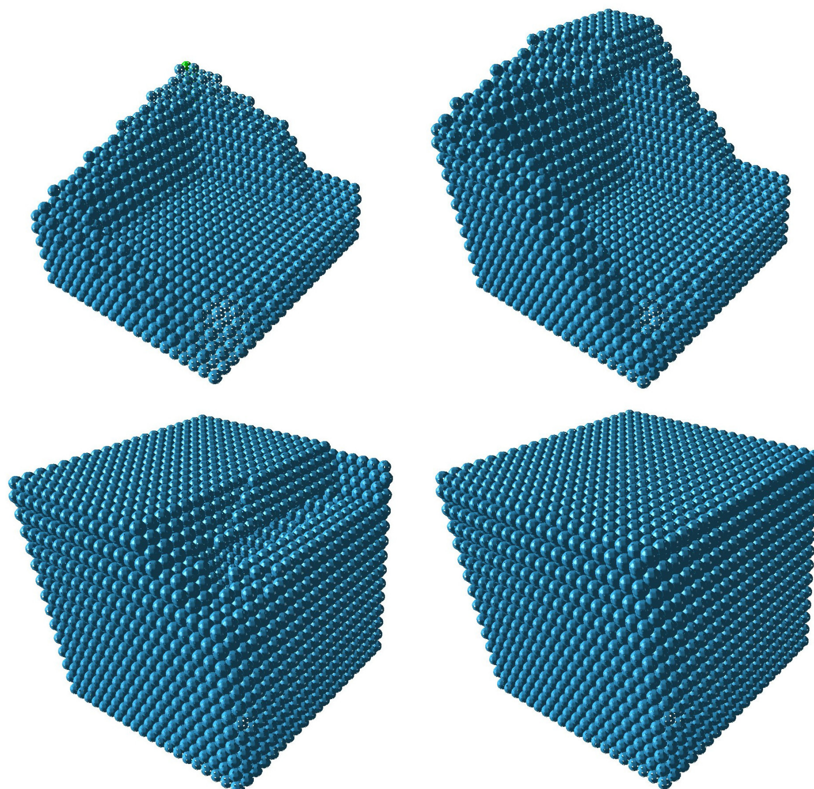


Figure 4.19: Number of available docking position during the progression of 3D self-assembly. Multilayer algorithm presents a peak of efficiency at a regular phase in the construction.

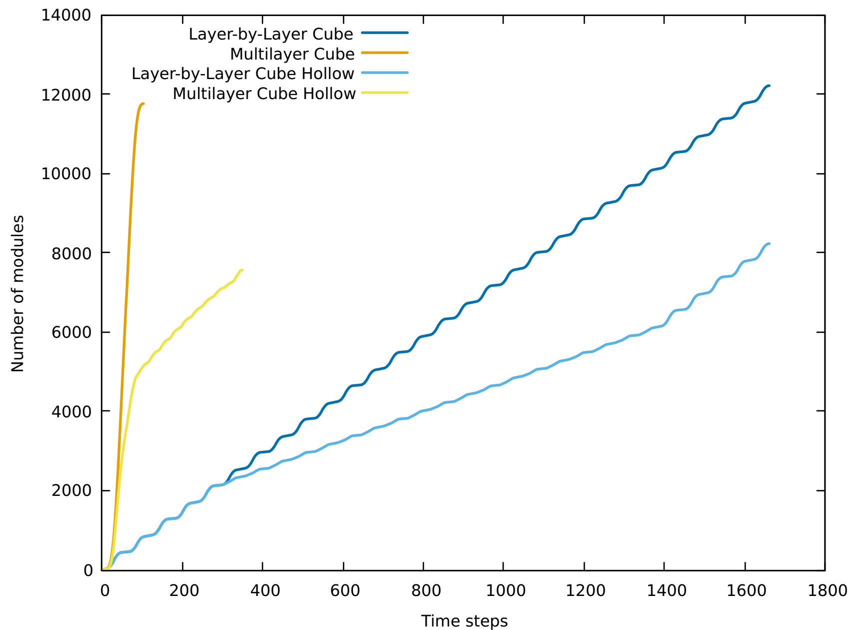


Figure 4.20: Comparing the number of time steps required to build a cube and a hollow version of a cube.

Figure 4.21 presents a chart with the number of messages by the number of modules present in the system. Layer-by-Layer algorithms produces a similar number of message for hollow and solid versions of the cube, although the solid versions requires slightly more number of messages due to the flood of a message indicating the plane has been finished to a layer. As there are fewer modules in some layers of the hollow version, there are fewer number of messages.

Multilayer algorithm also presents a similar number of message, but shows a small difference in the middle of the line of the hollow cube. Many synchronizations of layer, due to the internal hole, occur at the same time which lead to instantly increase the number of messages compared to the solid version. When 5000 modules are present in the system, the number of messages equalizes again. The number of message to synchronize the internal hole compensate the number of messages flooded with the indication the plane has finished and both lines grows in a similar proportion. Note also that the end layers of the cube are similar for hollow and solid versions.

The slight pump of messages in the end of both Multilayer algorithm is also explained with the indication that the plane has been finished to the superior layers. Even when all modules are already present, this message continues to be sent and the number of message continues to increase.

4.7.3.2/ SPHERE

In this subsection we evaluate a sphere as another base object and the behavior of the algorithm in this circumstance. A solid sphere model with 5931 and a hollow model with 3889 were used in this test scenario.

Again, the Multilayer algorithm in the charts of Figure 4.22 and Figure 4.23 shows to

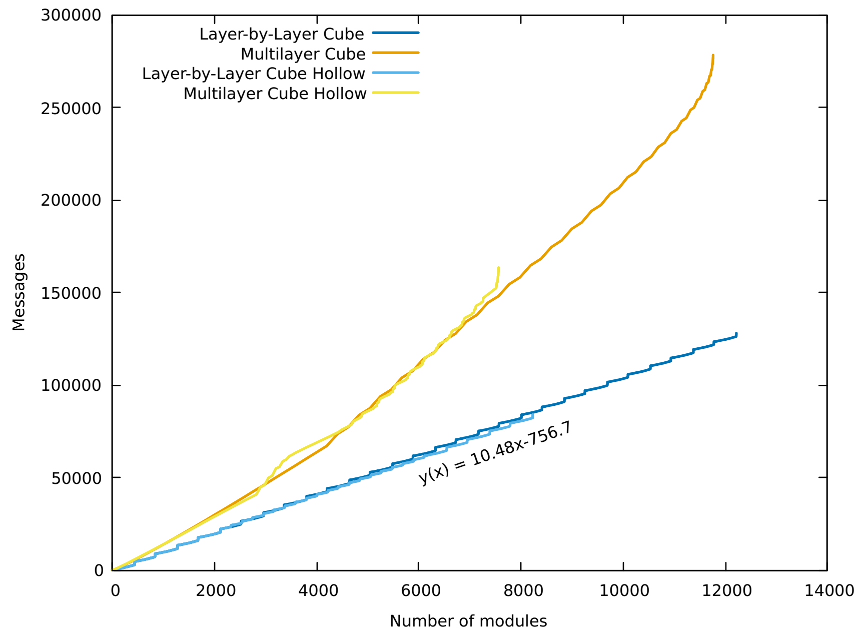


Figure 4.21: Number of messages used in a solid and hollow cube versions.

be efficient in terms of time steps for this type of object. Both show significant gains in terms of how they can parallelize the self-assembly attracting more modules in one single time step. In the evaluation, the solid version of the sphere takes 960 time steps to self-assembly using the Layer-by-Layer algorithm while the Multilayer takes 566 time steps. The hollow version takes 963 and 642 time steps for Layer-by-Layer and Multilayer algorithms respectively. The hollow version, although having less number of modules, takes slightly more time steps for the self-assembly than the solid sphere model due to its internal hole management.

The number of messages used during the self-assembling of a sphere follows again a linear value according to the number of modules according to the chart in Figure 4.24. Independent of the version of the sphere, the Multilayer algorithm requires 2 times more message than the Layer-by-Layer algorithm.

4.7.3.3/ Cylinder

A cylinder, as a last primitive evaluated, has been tested in its solid and hollow version as well. Figure 4.25 and Figure 4.26 present the charts for time steps and messages that follow the same pattern of the other primitives, cube and sphere analyzed previously.

4.7.3.4/ Pyramidal Object

A pyramidal object as presented in Figure 4.27 has also been evaluated. It is composed of a pyramid with an inverted pyramid in its top to check the behavior of the algorithm when layers shrinks and grows.

In Figure 4.28 we can note that both algorithms follow the same pattern, although the Multilayer algorithm shows to be useful to optimize the number of time steps. Both lines

	Layer-by-Layer Algorithm	Multilayer Algorithm
Solid	960	566
Hollow	963	642

Table 4.2: Total number of time steps for the self-assembly with two spheres of the same scale. The solid version contains 5931 modules and in its hollow version 3889 modules.

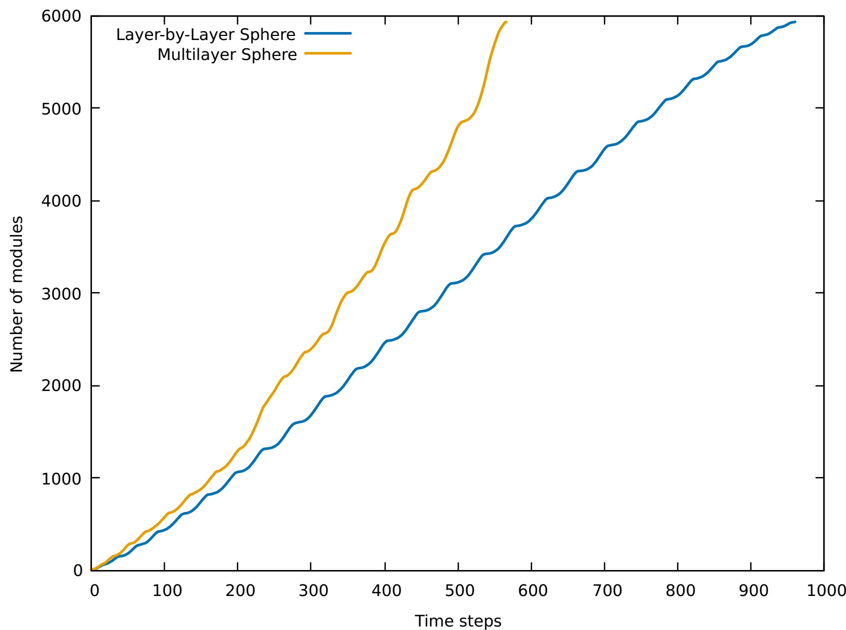


Figure 4.22: Number of time steps needed to self-assembly a solid sphere composed of 5.931 modules.

of the chart present as a pattern that the number of time steps is higher in the middle of the pyramidal object. As layers are smaller and the Multilayer algorithm parallelizes the self-assembly in a diagonal, the diagonal of the cubes in the middle of the pyramidal object contains fewer number of modules and slow down the time steps.

Also, the number of messages continues to follow the same linear growth seen in the other primitives as present in Figure 4.29.

4.7.3.5/ Locomotive 3D

Figure 4.30 shows the number of available positions in both Layer-by-Layer and Multilayer algorithms for the locomotive self-assembly presented in Figure 4.14. The Layer-by-Layer algorithm produces a result that oscillates, it peaks in the middle construction of a layer and reduces the number of available positions as few free positions remains in the end construction of the layer.

Here it is possible to verify the number of available docking positions in the Multilayer algorithm is mostly greater than in the Layer-by-Layer algorithm. It contains some oscillation as sometimes it has a behavior like Layer-by-Layer algorithm where the next layer should wait the confirmation that the previous has finished before continuing. Even in these cases some layers can be constructed at the same time.

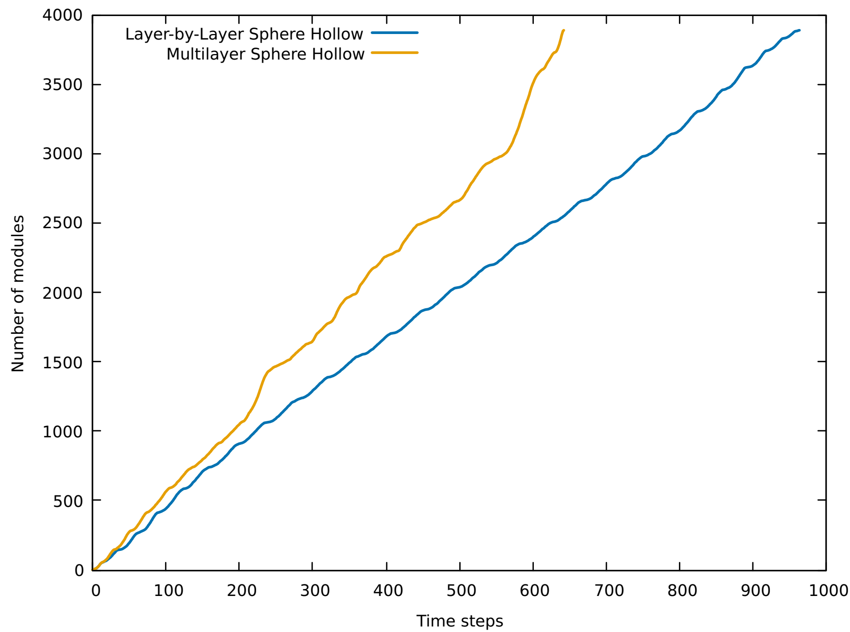


Figure 4.23: Number of time steps needed to self-assembly a hollow sphere composed of 3.889 modules.

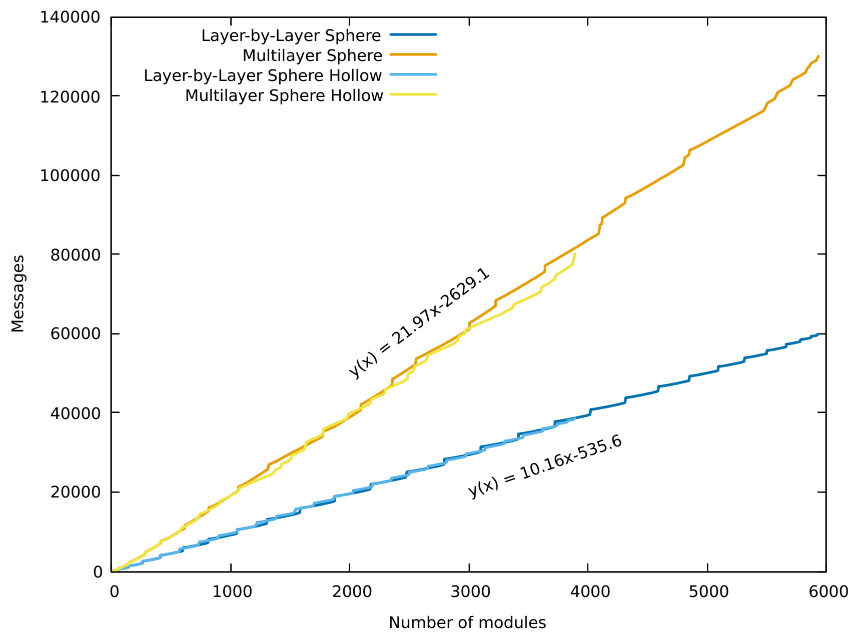


Figure 4.24: Messages required for the self-assembly of a sphere in a solid and hollow version.

It is possible to see there is no need to wait the previous layer when filling a big regular region of modules, as presented here in the 20% progression of self-assembly and represents the central body of the locomotive. In regular cases the number of available positions for Multilayer Algorithm can speed up considerably the construction of the object.

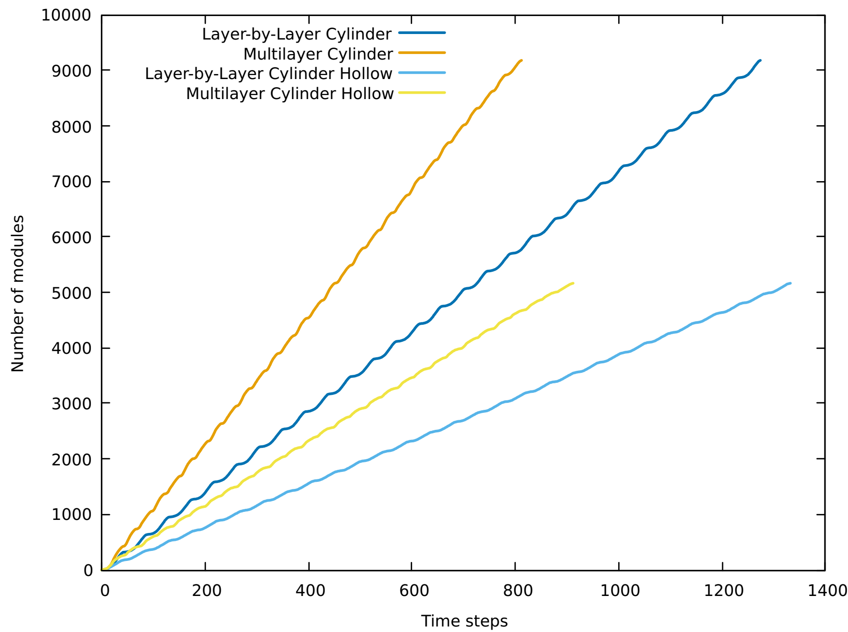


Figure 4.25: Messages required for the self-assembly of a cylinder in a solid and hollow version.

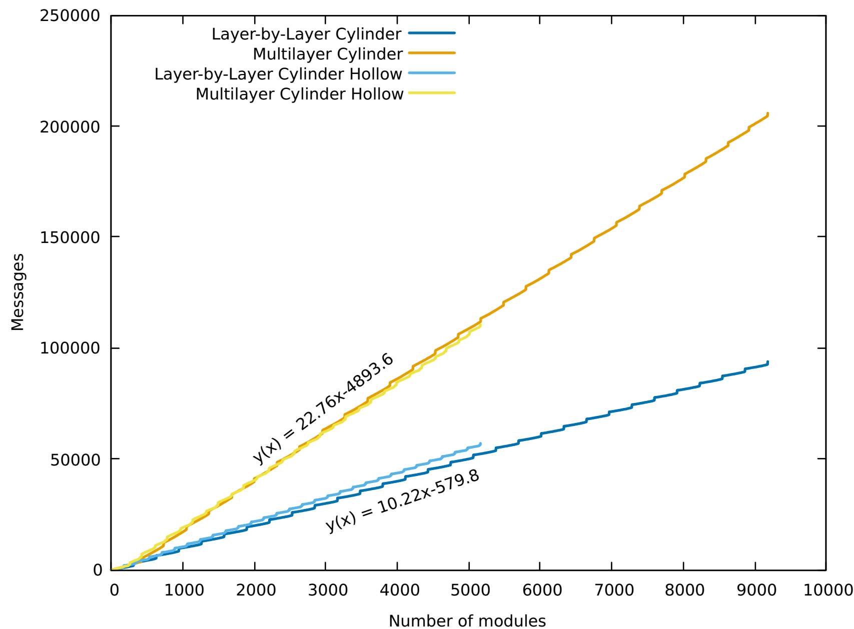


Figure 4.26: Messages required for the self-assembly of a cylinder in a solid and hollow version.

The number of messages in the Multilayer algorithm is essentially bigger than in the Layer-by-Layer algorithm due to its property of relying on more communication to be able to have many available positions at the same time. The comparison of how bigger is the cost of communication from one algorithm to another is presented in Figure 4.31.

As lines are almost linear, a certain equation with low error deviation can be expressed.

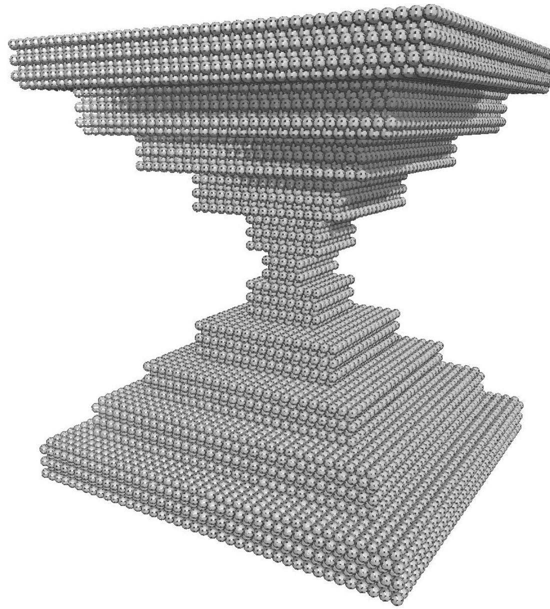


Figure 4.27: Pyramidal object used for evaluation of self-assembly algorithm that is made of 42.744 modules.

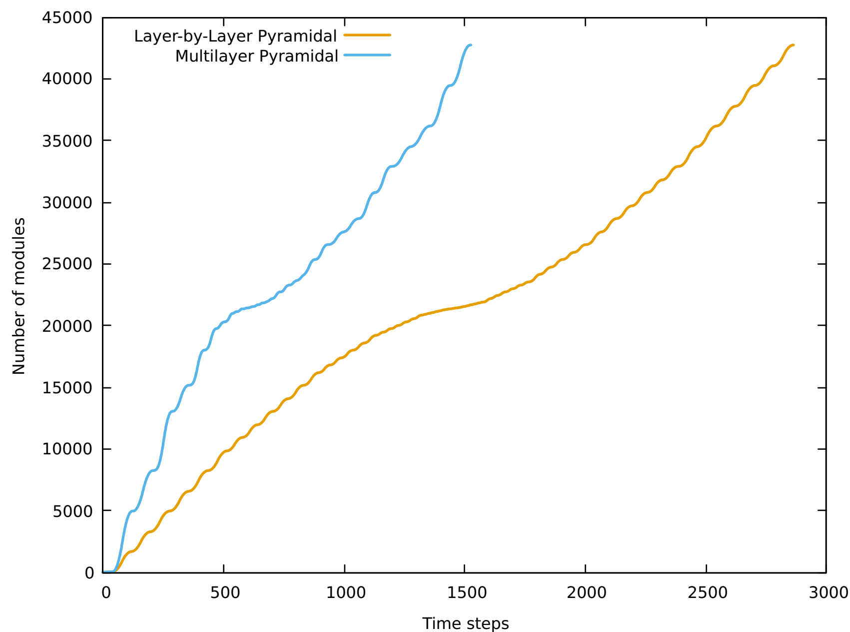


Figure 4.28: Time steps necessary to assembly a pyramidal object with 42.744 modules.

Analyzing these equations it is possible to see Layer-by-Layer algorithm using in average 10.16 messages per module while Multilayer algorithm uses approximately 21.97 messages per module. The number of messages for the locomotive representation shows to be similar to the other representations evaluated before.

According to the results in Figure 4.32, it is possible to conclude that with the trade off for more message usage in the Multilayer algorithm can be worth as it uses half the number of time step needed to assembly the object than in Layer-by-Layer algorithm.

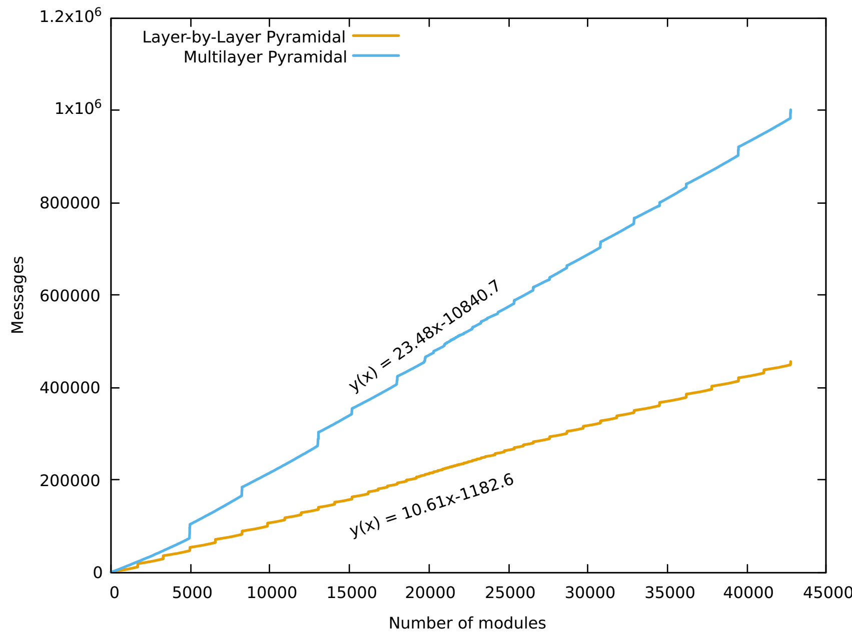


Figure 4.29: Messages used during the execution of the self-assembly planning algorithm for the pyramidal object.

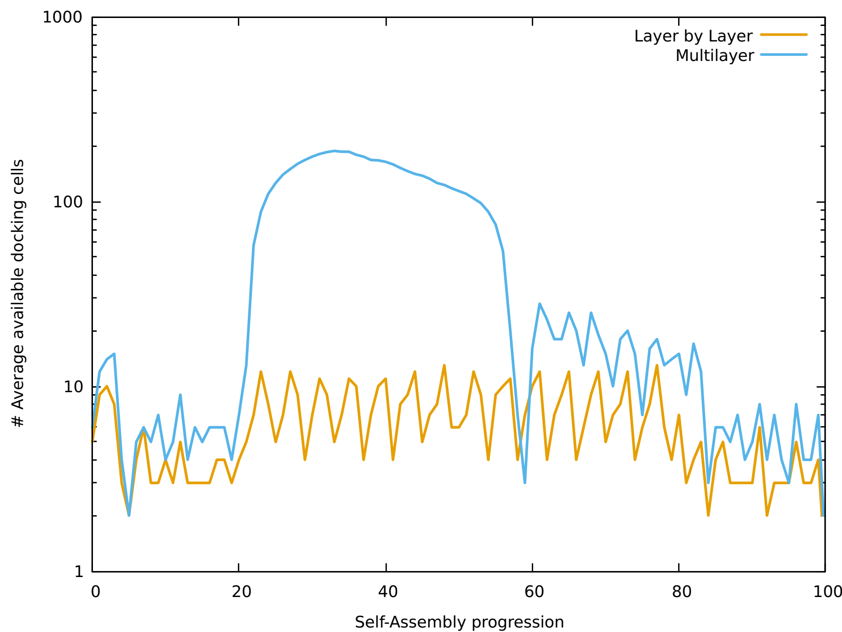


Figure 4.30: Number of available docking position during the progression of 3D self-assembly for the Locomotive 3D. Multilayer algorithm presents a peak of efficiency at a regular phase in the construction.

4.7.4/ 3D Self-Assembly with Merging Layers

The previous algorithms present an efficient solution for objects without merging layers, for example, they do not work for objects that divide in two columns that regroup later. The idea presented here has the main functionality to support two or more separate

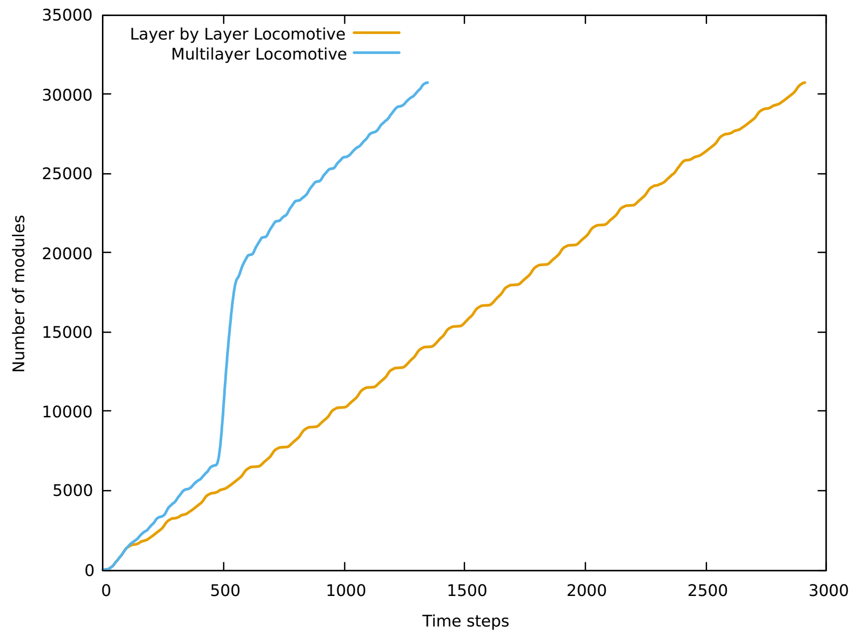


Figure 4.31: Neighborhood necessary for blue module be able to attract modules to its right side connector.

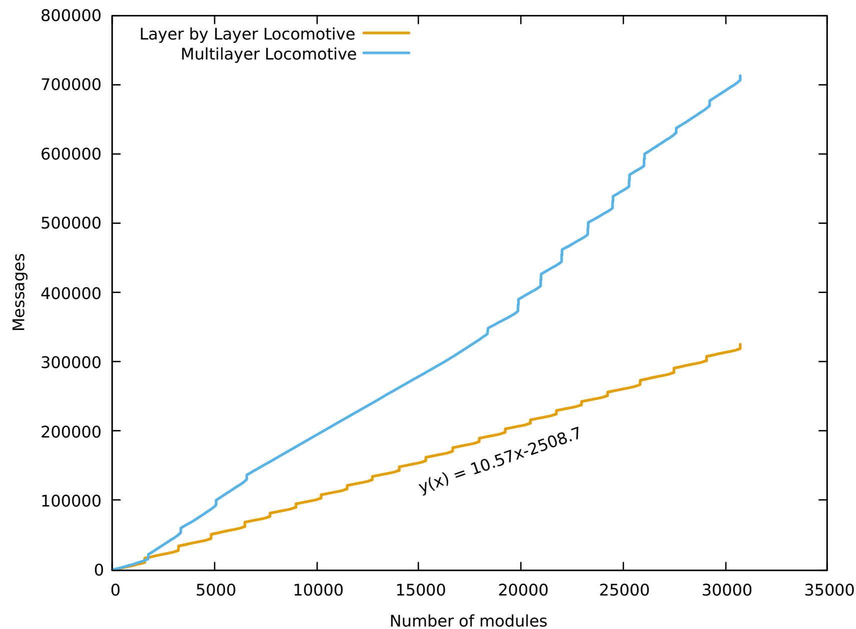


Figure 4.32: Number of messages by the number of modules in place in the 3D locomotive model. These messages are necessary to give a self-assembly planning that avoids blocked cells.

layers merging in one, presenting a solution for objects not supported with the previous algorithms.

As demonstrated before, if a layer starts the construction from two different points it will have one or more blocked positions. A solution to this problem is to have only one initial

module in a layer. Anyway, a problem continues to exist. How can a layer be sure there is no other layer constructing already the same next layer? How can a layer know they are the responsible to construct the next layer and the can proceed?

Let's take an example of two columns with a roof connecting both. If both columns are assembled and start constructing the roof, there will have certainly blocked positions when closing the layer of the roof. If one column is assembled before the other and constructs the roof before the other column has been completed, the last layer of the column will be blocked by the roof. To be able to perfectly construct the object, both columns should be totally constructed, and they have to decide to just one of them start the construction of the roof.

A self-assembly planning solution for *any* type of 3D object can be achieved centralizing the decisions in a leader *L*. It works by the leader L(assumed as the first module in the system) keeping the register of modules responsible to create the next layer (*3D seeds*) and giving permission to attract modules to their next layer one at a time. Once a layer has been completed, a message is sent to the central module to acknowledge. When the message is received it decides the next layer that can continue attracting for construction.

This idea to synchronize layer construction is modeled with a tree of seeds, linking *3D seeds* of connected layers. Then, when a local connected area has been completed, the layer parent sends a message ($EndOfLayer$) to the leader L with the id of the *3D seeds* in its layer, or no id when there is no *3D seed* indicating this branch has ended. Note that there is an implicit spanning tree when constructing the model and, as the leader module is the parent of the spanning tree, a route from any module to it is trivial. The leader waits for messages ($EndOfLayer$) from every branch of its internal seed tree to make an arbitrary choice of which *3D seed* can continue constructing, which can be the *3D seed* with minimum ID number. The leader floods the modular robot network with the id of the *3D seed* that have the authorization to attract a module on its next layer. Repeating this process leads to the construction of any model without holes in its structure.

The results of this algorithm in terms of time steps is similar to the Layer-by-Layer algorithm as they share many properties, they do the same checks in order of attracting new modules, and both works Layer-by-Layer and the main difference is that the algorithm presented in this section needs to wait for an authorization to start the next layer sending a message to the leader and waiting its response, while the other can start immediately. In terms of messages this algorithm sends more messages per new layer. For each new layer it sends one message to the leader following a route by the spanning tree, but the answer comes in the form of a flood in the network and is more expensive in terms of number of message.

Constructing an object with merging layers has a cost in terms of message, time steps and also in the decentralization that is diminished.

4.8/ Conclusion

We presented a set of distributed algorithms for a system of modular robots that is capable of generating an attraction list of available docking positions. Our algorithms prevent positions from becoming impossible to reach and is able to create close-packed structures with internal holes that are applicable to a variety of modular robotic systems. A time-

consuming task in a self-assembly system is moving modules to these placements. We show with our results that the algorithms can have many simultaneous docking positions and that it can speed up the creation of a model with a linear number of messages.

Four algorithms have been proposed so far:

- 2D Self-Assembly
- 3D Layer-by-Layer
- 3D Multilayer
- 3D Self-Assembly with Merging Layers

The two-dimensional self-assembly planning proved to have good results offering an order for any two-dimensional model without blocking points. All the 3D algorithms reuse the two-dimensional algorithm work in their layers. These 3D algorithms can be used for different type of models. The Layer-by-Layer algorithm has been proved to produce fewer possibilities of docking in parallel but also requires fewer number of messages. Multilayer algorithm presents a higher number of available positions during it construction, meaning a higher parallel activity, but it cannot build objects with reconnecting layers. A solution for models that cannot avoid merging layers has been proposed as well.

5
Conclusion

5.1/ Summary

In this thesis, I have presented solutions that bring self-assembly and self-reconfiguration using modular robots closer to reality. These contributions are related to module intelligence and distributed coordination.

One requirement for self-reconfigurable modular robots is to have a good description of the target shape. This description should be distributed to all robots and its size has an important role since flooding it to all neighbors have an impact on time, memory and energy resources of modular robotic systems. Beside of its size, a representation should present a high level of fidelity. The CSG4PM presented can fulfill all these requirements. This method shows high fidelity with low encoding size when compared to other methods. Also, the most used function for the target shape, that is to identify if a coordinate is located inside the object, has a simple and fast solution using CSG4PM which makes this method ready to be used in applications for programmable matter.

Another aspect for self-assembly and self-reconfiguration tackled in this thesis is how to order the modules docking in a distributed fashion to create the final configuration without holes in its structure. These holes could exist due to impossible docking positions. This happens when a docking position is located in a narrow space, too narrow to move the module inside this position, for example between two modules. A 2D distributed algorithm that can create any object based only in local rules has been presented. This algorithm requires a low number of messages and can have many available positions in order to have a higher parallel work and achieve the final object with fewer time steps.

Three 3D distributed solutions are also provided and compared between them. These 3D algorithms are an extension of the 2D algorithm presented and can be chosen between them depending on the target structure. The Multilayer algorithm is the most suitable for objects without merging layers that provides a higher number of available docking positions thus having a higher parallelization. Layer-by-Layer algorithm starts a new layer just when the current layer has been completed. One advantage over the Multilayer algorithm is that it requires fewer number of messages, an order of 10 messages per module while Multilayer requires 21 messages per module depending on the target structure. Another solution based on the Layer-by-Layer algorithm has been proposed that requires more messages per module and have fewer parallel activity but can create 3D objects with merging layers.

5.2/ Future Work

Scene Encoding As a future work, CSG tree has the advantage to be easily decomposed in different union operations. The target structure can be too big to fit in the memory of one single module. The tree structure can be divided before being sent to the master module, where these parts of the tree will be hold by a percentage of modules. This data sharing in a distributed network is known as *shard* and its behavior can be studied in modular robots.

Also, there is still place for improving the size of the representation using the CSG description, one of them is to reuse subtrees of the main tree. For example, in the case of the Toy Car presented in Figure 3.2, a number of subtrees could be reused reducing the overall size of the description as its wheels and some intrinsic structure of the pieces are similar.

Self-Assembly Planning In future works, we hope to extend the actual Multilayer algorithm to accept merging layers. An algorithm to find a path as the $BorderFollowing$ in three-dimensions could improve the number of parallel available cells and should be proved to be useful depending on the required number of messages.

We also would like to combine this planning algorithm, that gives the final position of the module, with algorithms to control module movements, collisions and path planning. Also, it would be interesting to have the algorithm implemented and tested on an existing hardware.

Publications

Thadeu Tucci, Benoît Piranda, and Julien Bourgeois.**A Distributed Self-Assembly Planning Algorithm for Modular Robots**. In 2018 International Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2018), Stockholm, Sweden, July 2018. ACM. Core Rank: A*.

Thadeu Tucci, Benoît Piranda, and Julien Bourgeois. **Efficient scene encoding for programmable matter self-reconfiguration algorithms**. In ACM SIGAPP Symposium on Applied Computing (SAC 2017), Marrakesh, Morocco, April 2017. ACM. Core Rank: B.

André Naz, Benoît Piranda, Thadeu Tucci, Seth Copen Goldstein, and Julien Bourgeois. **Network characterization of lattice-based modular robots with neighbor-to-neighbor communications**. In 2016 13th International Symposium on Distributed Autonomous Robotic Systems (DARS), London, UK, November 2016. Springer.

Julien Bourgeois, Benoît Piranda, André Naz, Nicolas Boillot, Hakim Mabed, Dominique Dhoutaut, Thadeu Tucci, and Hicham Lakhlef. **Programmable matter as a cyber-physical conjugation**. In 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2016), Budapest, Hungary, October 2016. IEEE. Core Rank: B.

Bibliography

Ahmadzadeh, H., and Masehian, E. (2015). **Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization**. *Artificial Intelligence*, 223:27–64.

Barraquand, J., and Latombe, J.-C. (1991). **Robot Motion Planning: A Distributed Representation Approach**. *The International Journal of Robotics Research*, 10(6):628–649.

Bie, D., Zhu, Y., Wang, X., Zhang, Y., and Zhao, J. (2016). **L-systems driven self-reconfiguration of modular robots**. *International Journal of Advanced Robotic Systems*, 13(5):1729881416669349.

Bishop, J., Burden, S., Klavins, E., Kreisberg, R., Malone, W., Napp, N., and Nguyen, T. (2005). **Programmable parts: a demonstration of the grammatical approach to self-organization**. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3684–3691.

Boillot, N., Dhoutaut, D., and Bourgeois, J. (2014). **Using Nano-wireless Communications in Micro-Robots Applications**. In *Proceedings of ACM The First Annual International Conference on Nanoscale Computing and Communication*, NANOCOM' 14, pages 10:1–10:9, New York, NY, USA. ACM.

Bourgeois, J., Piranda, B., Naz, A., Boillot, N., Mabed, H., Dhoutaut, D., Tucci, T., and Lakhlef, H. (2016). **Programmable matter as a cyber-physical conjugation**. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 002942–002947.

Butler, Z., Fitch, R., Rus, D., and Wang, Y. (2002). **Distributed goal recognition algorithms for modular robots**. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, pages 110–116. IEEE.

Butler, Z., Kotay, K., Rus, D., and Tomita, K. (2001). **Cellular Automata for Decentralized Control of Self-Reconfigurable Robots**. In *In Proc. IEEE ICRA Workshop on Modular Robots*, pages 21–26.

Butler, Z., and Rus, D. (2003). **Distributed planning and control for modular robots with unit-compressible modules**. *The International Journal of Robotics Research*, 22(9):699–715.

Castano, A., Behar, A., and Will, P. M. (2002). **The Conro modules for reconfigurable robots**. *IEEE/ASME Transactions on Mechatronics*, 7(4):403–409.

Deng, Y. (2018). **SCALABLE COMPILER FOR TERMES DISTRIBUTED ASSEMBLY SYSTEM**. Master's thesis, Cornell University.

Dhoutaut, D., Piranda, B., and Bourgeois, J. (2013). **Efficient simulation of distributed sensing and control environments**. In *IEEE Internet of Things (iThings/CPSCom)*, pages 452–459. IEEE.

Fitch, R., and Butler, Z. (2008). **Million Module March: Scalable Locomotion for Large Self-Reconfiguring Robots**. *The International Journal of Robotics Research*, 27(3-4):331–343.

Foley, J. D., Dam, A. v., Feiner, S. K., and Hughes, J. F. (1996). **Computer Graphics (2nd edn in C): Principles and Practice**.

Gauci, M., Ortiz, M. E., Rubenstein, M., and Nagpal, R. (2017). **Error Cascades in Collective Behavior: A Case Study of the Gradient Algorithm on 1000 Physical Agents**. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, pages 1404–1412, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Gilpin, K., Knaian, A., and Rus, D. (2010a). **Robot pebbles: One centimeter modules for programmable matter through self-disassembly**. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2485–2492. IEEE.

Gilpin, K., Kotay, K., Rus, D., and Vasilescu, I. (2008). **Miche: Modular Shape Formation by Self-Disassembly**. *The International Journal of Robotics Research*, 27(3-4):345–372.

Gilpin, K., and Rus, D. (2010b). **Modular Robot Systems**. *IEEE Robotics Automation Magazine*, 17(3):38–55.

Gilpin, K., and Rus, D. (2012). **A distributed algorithm for 2d shape duplication with smart pebble robots**. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3285–3292. IEEE.

Goldstein, S., and Mowry, T. (2004). **Claytronics: A Scalable Basis For Future Robots**. *RoboSphere 2004*.

Goldstein, S. C., Campbell, J. D., and Mowry, T. C. (2005). **Programmable matter**. *Computer*, 38(6):99–101.

Haghighat, B., Droz, E., and Martinoli, A. (2015). **Lily: A Miniature Floating Robotic Platform for Programmable Stochastic Self-Assembly**. *Proceedings - IEEE International Conference on Robotics and Automation*, 2015:1941–1948.

Hua, Y. (2018). **BUILDING 3d-STRUCTURES WITH AN INTelLIGENT ROBOT SWARM**. Master's thesis, Cornell University.

Jones, C., and Mataric, M. J. (2003). **From local to global behavior in intelligent self-assembly**. In *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, volume 1, pages 721–726 vol.1.

Jorgensen, M. W., Ostergaard, E. H., and Lund, H. H. (2004). **Modular ATRON: modules for a self-reconfigurable robot**. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 2, pages 2068–2073 vol.2.

Karagozler, M. E., Goldstein, S. C., and Reid, J. R. (2009). **Stress-driven MEMS assembly + electrostatic forces = 1mm diameter robot**. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2763–2769.

Kirby, B. T., Aksak, B., Campbell, J. D., Hoburg, J. F., Mowry, T. C., Pillai, P., and Goldstein, S. C. (2007). **A modular robotic system using magnetic force effectors**. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2787–2793.

Kirby, B. T., Ashley-Rollman, M., and Goldstein, S. C. (2011). **Blinky Blocks: A Physical Ensemble Programming Platform**. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1111–1116, New York, NY, USA. ACM.

Klingner, J., Kanakia, A., Farrow, N., Reishus, D., and Correll, N. (2014). **A stick-slip omnidirectional powertrain for low-cost swarm robotics: Mechanism, calibration, and control**. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 846–851.

Kurokawa, H., Tomita, K., Kamimura, A., Kokaji, S., Hasuo, T., and Murata, S. (2008). **Distributed Self-Reconfiguration of M-TRAN III Modular Robotic System**. *The International Journal of Robotics Research*, 27(3-4):373–386.

Lakhlef, H., Mabed, H., and Bourgeois, J. (2013). **Distributed and Dynamic Map-less Self-reconfiguration for Microrobot Networks**. In *2013 12th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 55–60.

Lee, Y., Bang, S., Lee, I., Kim, Y., Kim, G., Ghaed, M. H., Pannuto, P., Dutta, P., Sylvester, D., and Blaauw, D. (2013). **A Modular 1 mm$^{3\$}$Die-Stacked Sensing Platform With Low Power I$^{2\$}$C Inter-Die Communication and Multi-Modal Energy Harvesting**. *IEEE Journal of Solid-State Circuits*, 48(1):229–243.

Murata, S., Kurokawa, H., and Kokaji, S. (1994). **Self-assembling machine**. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 441–448 vol.1.

Naz, A., Piranda, B., Bourgeois, J., and Goldstein, S. C. (2016a). **A distributed self-reconfiguration algorithm for cylindrical lattice-based modular robots**. In *Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on*, pages 254–263. IEEE.

Naz, A., Piranda, B., Goldstein, S. C., and Bourgeois, J. (2016b). **Approximate-Centroid Election in Large-Scale Distributed Embedded Systems**. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 548–556.

Naz, A., Piranda, B., Goldstein, S. C., and Bourgeois, J. (2016c). **A Time Synchronization Protocol for Modular Robots**. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 109–118.

Naz, A., Piranda, B., Tucci, T., Goldstein, S. C., and Bourgeois, J. (2016d). **Network Characterization of Lattice-Based Modular Robots with Neighbor-to-Neighbor Communications**. In *13th International Symposium on Distributed Autonomous Robotic Systems (DARS-2016), In pres*.

Østergaard, E. H. (2004). **Distributed control of the ATRON self-reconfigurable robot**. PhD thesis, University of Southern Denmark.

Park, M., Chitta, S., Teichman, A., and Yim, M. (2008). **Automatic Configuration Recognition Methods in Modular Robots**. *The International Journal of Robotics Research*, 27(3-4):403–421.

Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Caro, G. D., Ducatelle, F., Birattari, M., Gambardella, L. M., and Dorigo, M. (2012). **ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems**. *Swarm Intelligence*, 6(4):271–295.

Piranda, B., and Bourgeois, J. (2016). **Geometrical study of a quasi-spherical module for building programmable matter**. In *DARS 2016, 13th International Symposium on Distributed Autonomous Robotic Systems*.

Requicha, A. A. (1980). **Representations of rigid solid objects**. Springer.

Romanishin, J. W., Gilpin, K., and Rus, D. (2013). **M-blocks: Momentum-driven, magnetic modular robots**. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4288–4295.

Rosa, M. D., Goldstein, S., Lee, P., Campbell, J., and Pillai, P. (2006). **Scalable shape sculpting via hole motion: motion planning in lattice-constrained modular robots**. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1462–1468.

Ross, F., and Ross, W. T. (2011). **The Jordan curve theorem is non-trivial**. *Journal of Mathematics and the Arts*, 5(4):213–219.

Rubenstein, M., Ahler, C., and Nagpal, R. (2012). **Kilobot: A low cost scalable robot system for collective behaviors**. In *2012 IEEE International Conference on Robotics and Automation*, pages 3293–3298.

Rubenstein, M., Cornejo, A., and Nagpal, R. (2014). **Programmable self-assembly in a thousand-robot swarm**. *Science*, 345(6198):795–799.

Salemi, B., Moll, M., and Shen, W. m. (2006). **SUPERBOT: A Deployable, Multi-Functional, and Modular Self-Reconfigurable Robotic System**. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3636–3641.

Seo, J., Yim, M., and Kumar, V. (2013). **Assembly planning for planar structures of a brick wall pattern with rectangular modular robots**. In *2013 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1016–1021.

Seo, J., Yim, M., and Kumar, V. (2016). **Assembly sequence planning for constructing planar structures with rectangular modules**. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5477–5482.

Stoy, K., and Nagpal, R. (2004). **Self-repair through scale independent self-reconfiguration**. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, pages 2062–2067. IEEE.

Stoy, K., and Nagpal, R. (2007). **Self-Reconfiguration Using Directed Growth**. In Alami, R., Chatila, R., and Asama, H., editors, *Distributed Autonomous Robotic Systems 6*, pages 3–12. Springer Japan.

Stoy, K., Nagpal, R., and Shen, W.-M. (2010). **Modular Robots: The State of the Art**. *Proceedings of the IEEE*.

Suh, J. W., Homans, S. B., and Yim, M. (2002). **Telecubes: mechanical design of a module for self-reconfigurable robotics**. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, volume 4, pages 4095–4101 vol.4.

Toffoli, T., and Margolus, N. (1991). **Programmable Matter: Concepts and Realization**. In *Proceedings of the NATO Advanced Research Workshop on Lattice Gas Methods for PDE's : Theory, Applications and Hardware: Theory, Applications and Hardware*, pages 263–272, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

Tolley, M. T., Hiller, J. D., and Lipson, H. (2011a). **Evolutionary design and assembly planning for stochastic modular robots**. *New Horizons in Evolutionary Robotics*, pages 211–225.

Tolley, M. T., and Lipson, H. (2010). **Fluidic manipulation for scalable stochastic 3d assembly of modular robots**. In *2010 IEEE International Conference on Robotics and Automation*, pages 2473–2478.

Tolley, M. T., and Lipson, H. (2011b). **On-line assembly planning for stochastically reconfigurable systems**. *The International Journal of Robotics Research*, 30(13):1566–1584.

Wei, H., Chen, Y., Tan, J., and Wang, T. (2011). **Sambot: A self-assembly modular robot system**. *IEEE/ASME Transactions on Mechatronics*, 16(4):745–757.

Werfel, J., Bar-Yam, Y., Rus, D., and Nagpal, R. (2006a). **Distributed construction by mobile robots with enhanced building blocks**. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 2787–2794.

Werfel, J., Ingber, D., and Nagpal, R. (2007). **Collective construction of environmentally-adaptive structures**. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2345–2352.

Werfel, J., and Nagpal, R. (2006b). **Extended stigmergy in collective construction**. *IEEE Intelligent Systems*, 21(2):20–28.

Werfel, J., and Nagpal, R. (2008). **Three-Dimensional Construction with Mobile Robots and Modular Blocks**. *The International Journal of Robotics Research*, 27(3-4):463–479.

White, P. J., Kopanski, K., and Lipson, H. (2004). **Stochastic self-reconfigurable cellular robotics**. In *2004 IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04*, volume 3, pages 2888–2893 Vol.3.

Whitted, T. (1979). **An Improved Illumination Model for Shaded Display**. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, pages 14–, New York, NY, USA. ACM.

Wu, C., Ge, T., and Lian, L. (2008). **USS - An Underwater Self-reconfigurable System**. In *OCEANS 2008*, pages 1–7.

Yim, M., Roufas, K., Duff, D., Zhang, Y., Eldershaw, C., and Homans, S. (2003). **Modular Reconfigurable Robots in Space Applications**. *Autonomous Robots*, 14(2-3):225–237.

Yim, M., Zhang, Y., and Duff, D. (2002). **Modular robots**. *IEEE Spectrum*, 39(2):30–34.

Yim, M., Zhang, Y., Lamping, J., and Mao, E. (2001). **Distributed Control for 3d Metamorphosis**. *Autonomous Robots*, 10(1):41–56.

Zykov, V., Mytilinaios, E., Desnoyer, M., and Lipson, H. (2007). **Evolved and Designed Self-Reproducing Modular Robotics**. *IEEE Transactions on Robotics*, 23(2):308–319.

List of Figures

1.1 World's actual smallest computer. 1

2.1 Programmable matter applied to create an intelligent mug which can be modeled. 7

2.2 Implementation of Modular robots. Modules **(R)**, **(S)** and **(T)** are part of the Claytronics project. 9

2.3 The actual consortium of partnership to develop the field of programmable matter and modular robots. 13

2.4 BlinkyBlocks running a distributed algorithm to find the approximative center module and apply rainbow colors based on the distance to the center. . 14

2.5 Planar Catoms V8 with solid state electronic controls at the top of the stack, sensors presented inside and electro magnets in orange color. 14

2.6 2D Catom with the actuation scheme on left and, at right, a real prototype in a power grid. These robots have 1mm diameter. 15

2.7 3D Catoms in FCC lattice where modules can have up to 12 connected neighbors. 16

2.8 VisibleSim simulator executed with 127.583 Catoms 3D emulating a toy car. 20

3.1 Process to define a target shape used before a reconfiguration algorithm. Firstly, a model is chosen and modeled. This model is then transfered to a master module that retransmit it to the entire network of modular robots. . . 26

3.2 Huge set of 429,921 micro-robots defining a Toy Car. 27

3.3 Operations to define a mug with CSG Tree. 28

3.4 Canonical objects, sphere and cylinder, represented along with its transformation matrix. 29

3.5 Checking if point P is inside the polygon. As the ray traverses the polygon an even number of times it is possible to assume it is located outside. Otherwise, if the ray would traverse the polygon an odd number of times, the point would be inside. 30

3.6 Triangle with u and v to solve the point inside a triangle problem. 31

3.7 A brick that can represented using two coordinates of its extremities. 32

3.8 Global model flow used for reconfiguration. 33

3.9 Example of traversing a CSG tree to determine if the point P is inside the described object 35

3.10 Simple colored mug used in the experiments. 36

3.11 Flow of CSG4PM reconfiguration. At the top, the CSG tree is created and a binary version is transmitted to a master module (bottom left). This master module flood the goal configuration tree to the entire network of robots(modules become yellow after reception of goal configuration). These modules now can decide positions that are inside or outside the goal shape. White modules know they are inside the target shape and do not need to move. Other modules find a free position that is inside the target shape (red points) to occupy. 37

3.12 Triangle mesh used in the mug model. 38

3.13 Mug representation using overlapping bricks. Three different minimum brick sizes are used. 39

3.14 Fidelity of the Toy Car structure using overlapping bricks 40

3.15 Size of codes using different vector encoding for 2 different structures . . . 41

3.16 Time of decoding 42

4.1 Example of possible and impossible docking positions due to kinematic restrictions. 45

4.2 Importance of sequence planning shown on a Mug model made by 12,000 modules. Red modules represent positions that could not be docked using two simple planning algorithm. On the left, choosing a stochastic order produces 3,691 modules that does not verify the docking rule. On the right, filling regularly each module neighborhood in sequence results in 231 positions that could not be filled. 46

4.3 Modular robots and kinematic constraints in a row: only one contiguous line of blocks can be allowed. Modules A and B cannot be part of different contiguous line to avoid positions to get blocked. 54

4.4 A set of modules where the colored modules use the definition of the Rule $WestAttraction$. The color is according to which condition of the rule that can be applied. 57

4.5 The image shows colored modules that used the rule $isNorthSeed$ and are responsible to attract a module on its North connector. Black modules cannot call others to their North connector as they are not seed. 58

4.6 Light-gray modules represent seeds which are responsible for attracting a new module on its north row. Each contiguous row can have only one seed to avoid blocked positions. The rightmost module is alway elected, except for Module A, since it is part of an internal hole. 58

4.7 Cells that should be in place before attracting neighbors. Before attracting a module in cell A, modules B and C should already be in place. B and C are not direct neighbors, so to communicate they need a common neighbor module to know if both are already in place. The construction method fills space by adding modules along the diagonal line in both directions simultaneously (NW and SE). 59

4.8 Example of constructing a model with internal hole in a two-dimensional square lattice. The task is parallelized as many positions can be filled at the same time. The modular robot drawn in red detects a section of line merge made by the internal hole. It sends a message following the inner border (blue modules) and waits for an unlocking message. At the moment the message arrives in the south-west position of the red module, the module sends a response in the reverse path (green). This verification is necessary before a module docks on the red module west side in order to have no position blocked. 60

4.9 Three classes of target structure: Power Button, Letter C and Bumpy respective representations. 61

4.10 A side view of a locomotive used for the experimental in two dimensions. . 61

4.11 Number of messages for four topologically different representations on three different scales. The number of messages remains linear and similar across all representations. 62

4.12 Relative number of available docking positions over the percentage of construction. It shows parallelism of the method by having as many docking positions as possible at the same time. The thick lines represent the standard deviation in a certain construction percentage. 63

4.13 Some examples of modular robots configurations that express if the center position (colored module) can be docked or not. Modules in green represent positions that can be docked with the surrounding modules. Red modules represent positions where docking is not possible due the space needed for module movement. 65

4.14 Locomotive example made by 61,780 Catoms constructed using the distributed 3D self-assembly algorithm. The self-assembly can be done without any position being blocked during the construction. 66

4.15 Neighborhood necessary for the blue module be able to attract a module to its right side connector. 67

4.16 In green its represented the layer seed modules. FCC lattice can give different order of layer seed depending on the origin module placement in the coordinate system. This different order has consequence in the number of time steps needed to self-assembly a final object. 69

4.17 Number of available docking position during the progression of 3D self-assembly. Multilayer Algorithm presents a peak of efficiency at a regular phase in the construction. 70

4.18 Number of messages required to self-assembly a cube using two different algorithms. 71

4.19 Number of available docking position during the progression of 3D self-assembly. Multilayer algorithm presents a peak of efficiency at a regular phase in the construction. 71

4.20 Comparing the number of time steps required to build a cube and a hollow version of a cube. 72

4.21 Number of messages used in a solid and hollow cube versions. 73

4.22 Number of time steps needed to self-assembly a solid sphere composed of 5.931 modules. . . . 74

4.23 Number of time steps needed to self-assembly a hollow sphere composed of 3.889 modules. . . . 75

4.24 Messages required for the self-assembly of a sphere in a solid and hollow version. . . . 75

4.25 Messages required for the self-assembly of a cylinder in a solid and hollow version. . . . 76

4.26 Messages required for the self-assembly of a cylinder in a solid and hollow version. . . . 76

4.27 Pyramidal object used for evaluation of self-assembly algorithm that is made of 42.744 modules. . . . 77

4.28 Time steps necessary to assembly a pyramidal object with 42.744 modules. 77

4.29 Messages used during the execution of the self-assembly planning algorithm for the pyramidal object. . . . 78

4.30 Number of available docking position during the progression of 3D self-assembly for the Locomotive 3D. Multilayer algorithm presents a peak of efficiency at a regular phase in the construction. . . . 78

4.31 Neighborhood necessary for blue module be able to attract modules to its right side connector. . . . 79

4.32 Number of messages by the number of modules in place in the 3D locomotive model. These messages are necessary to give a self-assembly planning that avoids blocked cells. . . . 79

B.1 Diameter bounds by the numer of vertices in the network graph for difference LMR network. S-Lattice and H-Lattice stands for Square and Hexagonal lattices, which are two-dimensional lattices. SC-Lattice stands for 3D Simple Cubic Lattice. The terms “LB” and “UB” respectively stand for “lower bound” and “upper bound”. . . . 103

B.2 An $FCC\text{-}Ball(2)$ of 3D Catoms and its decomposition into horizontal layers with color gradient from the center of the ball. . . . 104

List of Tables

2.1 Table comparing different types of hardware specification for modular robot. 10
2.2 Coordinate of all possible neighbors in FCC lattice for Catoms3D. 17
2.3 Comparison of Modular Robots simulators. 18

3.1 Summary of encoding scene methods for Programmable Matter 33

4.1 Robots Planning Algorithm Overview 49
4.2 Total number of time steps for the self-assembly with two spheres of the same scale. The solid version contains 5931 modules and in its hollow version 3889 modules. 74

ANNEXES

A

2D Self-Assembly Implementation Details

Algorithm 6: Algorithm functions for 2D self-assembly planning - Part 1.

```
1  Function northAttraction(B_i):
2      if isNorthSeed(B_i) then
3          sendAttractSignalTo(⟨x_i, y_i + 1⟩);
4  end

5  Function southAttraction(B_i):
6      if isSouthSeed(B_i) then
7          sendAttractSignalTo(⟨x_i, y_i − 1⟩);
8  end

9  Function westAttraction(B_i):
10     if West(B_i) ∈ G then
11         if SouthWest(B_i) ∈ G ∧ hasModuleOnSouthConnector(B_i) then
12             getAuthorizationToAttract(South(B_i), WEST);
13         else if isOnInternalHole(B_i) then
14             borderFollowingToGetAuthorizationToAttract();
15         else
16             sendAttractSignal(⟨x_i, y_i − 1⟩);
17         end
18 end

19 Function eastAttraction(B_i):
20     if East(B_i) ∈ G then
21         if NorthEast(B_i) ∈ G ∧ hasModuleOnNorthConnector(B_i) then
22             getAuthorizationToAttract(North(B_i), EAST);
23         else if isOnInternalHole(B_i) then
24             borderFollowingToGetAuthorizationToAttract();
25         else
26             sendAttractSignal(⟨x_i, y_i + 1⟩);
27         end
28 end
```

Algorithm 7: Algorithm functions for 2D self-assembly planning - Part 2.

```
29 Function getAuthorizationToAttract(B_i, direction):
30   if direction = WESTt then
31     if isConnected(West(B_i)) then
32       sendAuthorizationToAttract(North(B_i), WEST) ;
33   else if direction = EASTt then
34     if isConnected(East(B_i)) then
35       sendAuthorizationToAttract(South(B_i), EAST) ;
36 end

37 Function sendAuthorizationToAttract(B_i, direction):
38   if direction = WESTt then
39     sendAttractSignal(⟨x_i − 1, y_i⟩);
40   else if direction = EASTt then
41     sendAttractSignal(⟨x_i + 1, y_i⟩);
42   end
43 end
```

B

Demonstrations of LMR Network Properties

Lattice-based Modular Robots (LMR) network has some specific properties and are different from Internet network. One of these difference is that LMRs network form sparse and large-diameter networks as we demonstrate. Bounds of the radius and the diameter of these network work are also provided.

In the next section we analyze the FCC lattice-based, which has been subject of a paper in cooperation with my office mate Andre Naz, who went beyond and also calculated the diameter bounds for other lattice-based modular robot networks as show in Figure B.1.

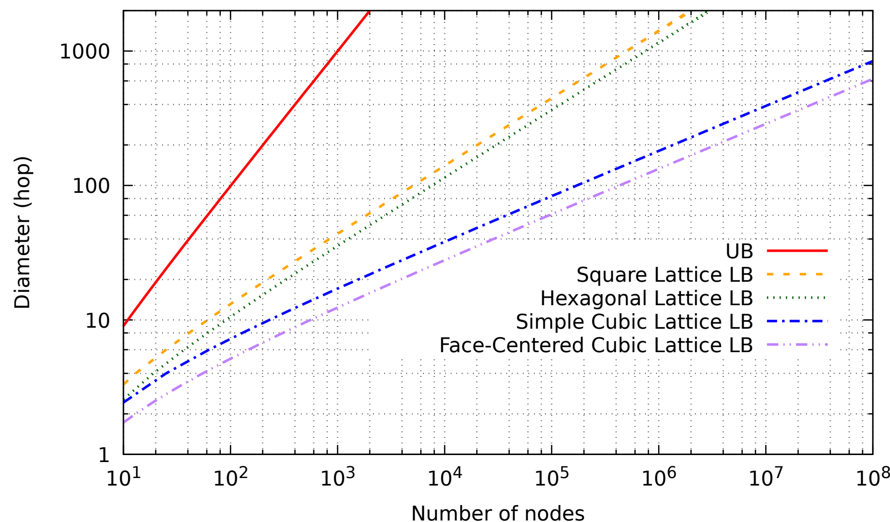


Figure B.1: Diameter bounds by the numer of vertices in the network graph for difference LMR network. S-Lattice and H-Lattice stands for Square and Hexagonal lattices, which are two-dimensional lattices. SC-Lattice stands for 3D Simple Cubic Lattice. The terms “LB” and “UB” respectively stand for “lower bound” and “upper bound”.

Three Dimension System: Face-Centered Cubic Lattices In this section, we compute the exact radius of an *L-Ball*, given the number of vertices it contains, for the case of three-dimensional systems embedded in Face-Centered Cubic (FCC) lattices. Figure B.2 depict the *FCC-Ball* of radius 2, composed of 3D Catoms. This system can be

decomposed into horizontal layers.

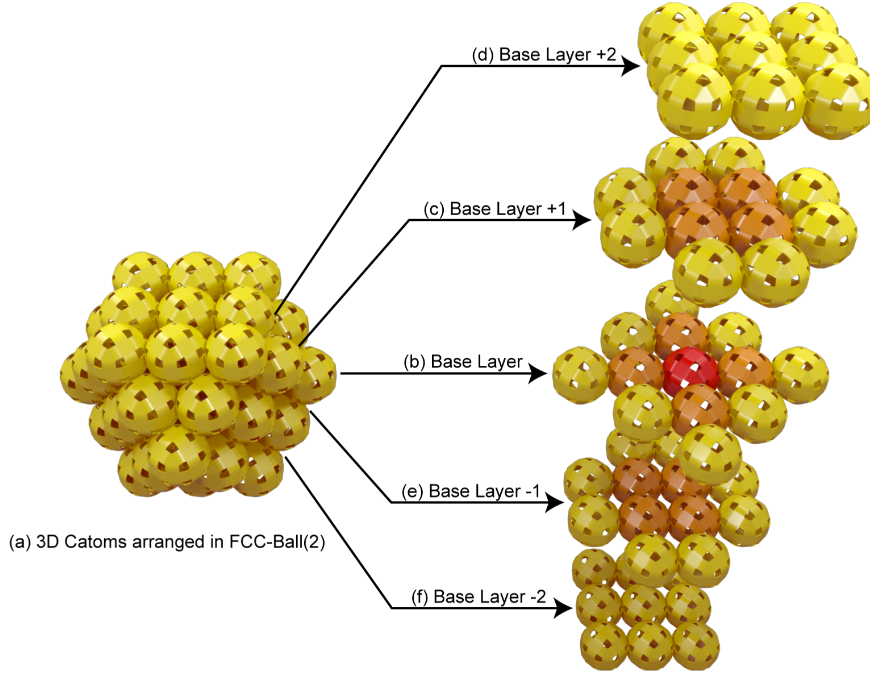


Figure B.2: An $FCC\text{-}Ball(2)$ of 3D Catoms and its decomposition into horizontal layers with color gradient from the center of the ball.

Lemma B.0.1. *In the face-centered cubic lattice, the number of vertices in a sphere of radius $r \geq 1$, $n_{FCC\text{-}Sphere}(r)$, can be computed by:*

$$n_{FCC\text{-}Sphere}(r) = 4r + 2(r+1)^2 + 2(r-1)4r \quad \text{(B.1)}$$
$$= 2(5r^2 + 1) \quad \text{(B.2)}$$

Proof. As shown in Figure B.2, a sphere of radius r in the face-centered cubic lattice can be decomposed into $2r + 1$ horizontal layers. The base layer is an $S\text{-}Sphere(r)$ and contains $4r$ vertices. The bottom and the top layers both contain $(r+1)^2$ vertices. The $2(r-1)$ other layers contain $4r$ vertices each. Equation (B.1) is obtained by summing up the number or vertices of each layer. □

Theorem B.0.1. *In the face-centered cubic lattice, the radius of a ball of $n \geq 1$ vertices, $r_{FCC\text{-}Ball}(n)$, can be computed by:*

$$r_{FCC\text{-}Ball}(n) = \frac{1}{2}\left(\frac{(\sqrt{15}\sqrt{4860n^2+343}+270n)^{\frac{1}{3}}}{15^{\frac{2}{3}}} - \frac{7}{15^{\frac{1}{3}}(\sqrt{15}\sqrt{4860n^2+343}+270n)^{\frac{1}{3}}} - 1\right) \quad \text{(B.3)}$$

Proof. By definition, $L\text{-}Ball(r)$ is the union of all the $L\text{-}Sphere(i)$ for i ranging from 0 to r. Thus, for $r \geq 1$, the number of vertices in an $FCC\text{-}Ball(r)$, $n_{FCC\text{-}Ball}(r)$, can be computed

as follows:

$$n_{FCC\text{-}Ball}(r) = \sum_{i=0}^{r} n_{FCC\text{-}Sphere}(i) \tag{B.4}$$

$$= 1 + \sum_{i=1}^{r} 2(5i^2 + 1) \tag{B.5}$$

$$= \frac{10}{3}r^3 + 5r^2 + \frac{11}{3}r + 1 \tag{B.6}$$

To obtain Equation B.3, we solve Equation (B.6) for r and keep only the real root. □

Document generated with LaTeX and:
the LaTeX style for PhD Thesis created by S. Galland — http://www.multiagent.fr/ThesisStyle
the tex-upmethodology package suite — http://www.arakhne.org/tex-upmethodology/

école doctorale sciences pour l'ingénieur et microtechniques

Title: Distributed Algorithms for Programmable Matter: Target Shape Description and Self-Assembly Planning

Keywords: modular robots, programmable matter, distributed programming, self-assembly, self-reconfiguration, large scale distributed robots

Abstract:

Programmable matter can be seen as a huge modular robot in which each module can communicate to its connected neighbors and work all together to achieve a common goal, more likely changing the shape of the whole robot and adapting it with new functionalities. In order to achieve coordination between a group with many thousand robots, local rules and distributed algorithms would take advantage in this environment. In the same way, small modules means there is also small resources and algorithms should be designed to reflect these needs. This thesis provide algorithms and solutions to solve some parts of the self-reconfiguration problem with each module embedding the same algorithm and coordinating with the others by means of neighbor-to-neighbor communication. One of them is a study and proposal of a representation for the goal structure that reduces the footprint memory. Also, the self-assembly like self-reconfiguration is composed of two steps: (1) identifying the free positions that are available for docking and (2) moving and docking modules to these positions. In this thesis, distributed solutions for the first step are presented which can decide positions that can be filled and can create any 3D shape, including shapes with internal holes and concavities.

Titre : Distributed Algorithms for Programmable Matter: Target Shape Description and Self-Assembly Planning

Mots-clés : robots modulaires, matière programmable, programmation distribuée, auto-assemblage, auto-reconfiguration, robots distribués à grande échelle

Résumé :

La matière programmable peut être vue comme un énorme robot modulaire dans lequel chaque module peut communiquer avec ses voisins connectés et travailler tous ensemble pour atteindre un objectif commun, modifiant la forme du robot et l'adaptant à nouvelles fonctionnalités. Afin de parvenir à une coordination entre un groupe de plusieurs milliers de robots, des règles locales et des algorithmes distribués profiteraient à ce type d'environnement. De la même manière, utiliser de petits modules signifient qu'il existe également des ressources restreintes et que les algorithmes doivent être conçus pour intégrer ces contraintes. Cette thèse fournit des algorithmes et des solutions pour résoudre certaines parties du problème d'auto-reconfiguration, chaque module intégrant le même algorithme et se coordonnant avec les autres au moyen d'une communication de voisin à voisin. L'une d'entre elles propose une représentation de la structure cible qui réduit la mémoire utilisée. De plus, l'auto-assemblage comme l'auto-reconfiguration se compose de deux étapes: (1) identifier les positions libres disponibles pour l'installation des modules et (2) déplacer et connecter les modules à ces positions. Dans cette thèse sont présentées des solutions distribuées pour la première étape qui peuvent décider des positions pouvant être remplies et peuvent créer n'importe quelle forme 3D, y compris des formes avec des trous internes et des concavités.

UBFC UNIVERSITE BOURGOGNE FRANCHE-COMTE
Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besançon, France