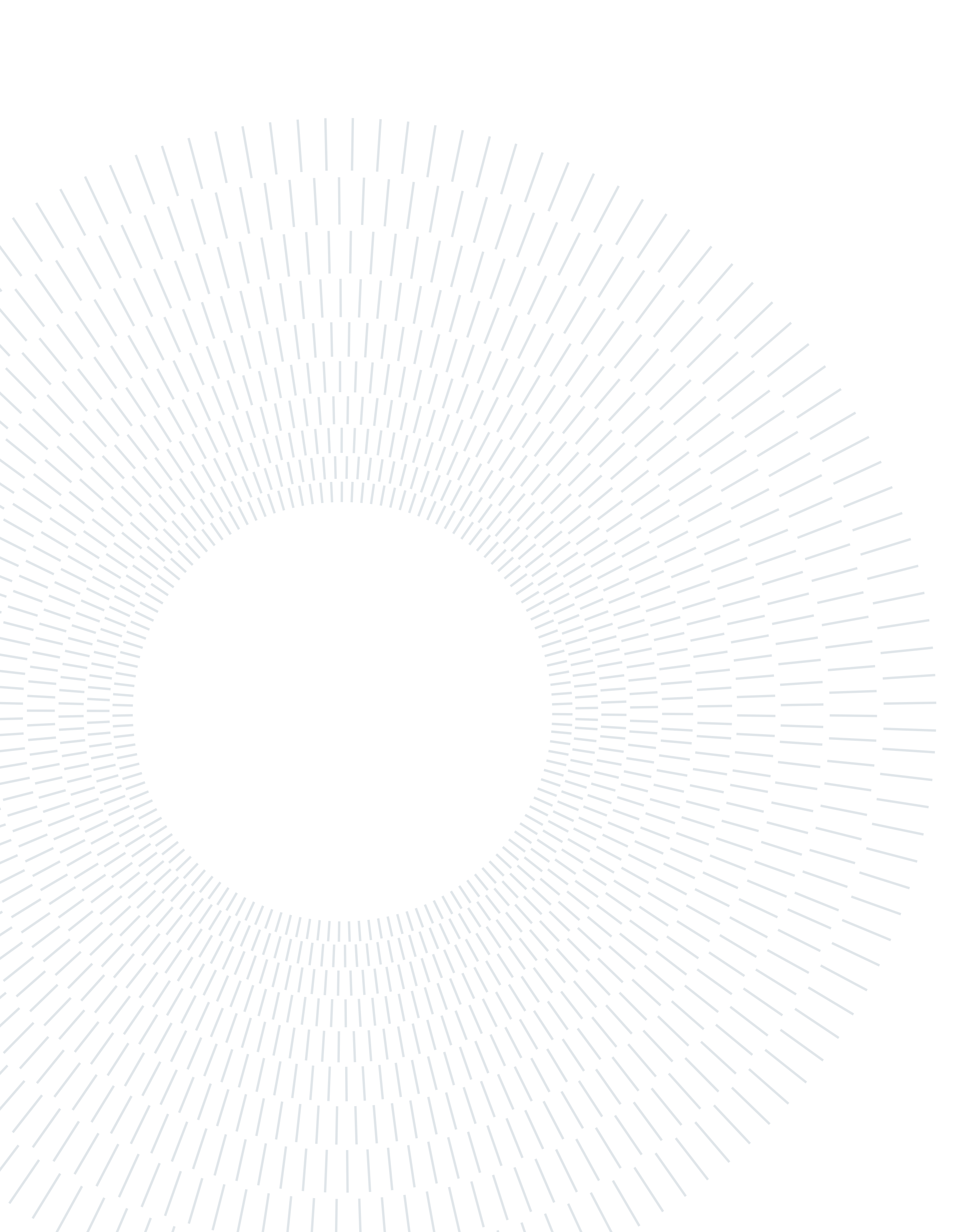
Semantic representation of a scene for robotic applications

Tesi di Laurea Magistrale in
Automation and Control Engineering - Ingegneria dell'Automazione

Author: **Alessandro Petazzi**

Student ID: 953628
Advisor: Prof. Paolo Rocco
Co-advisors: Prof. Andrea Maria Zanchettin, Ing. Isacco Zappa
Academic Year: 2021-22



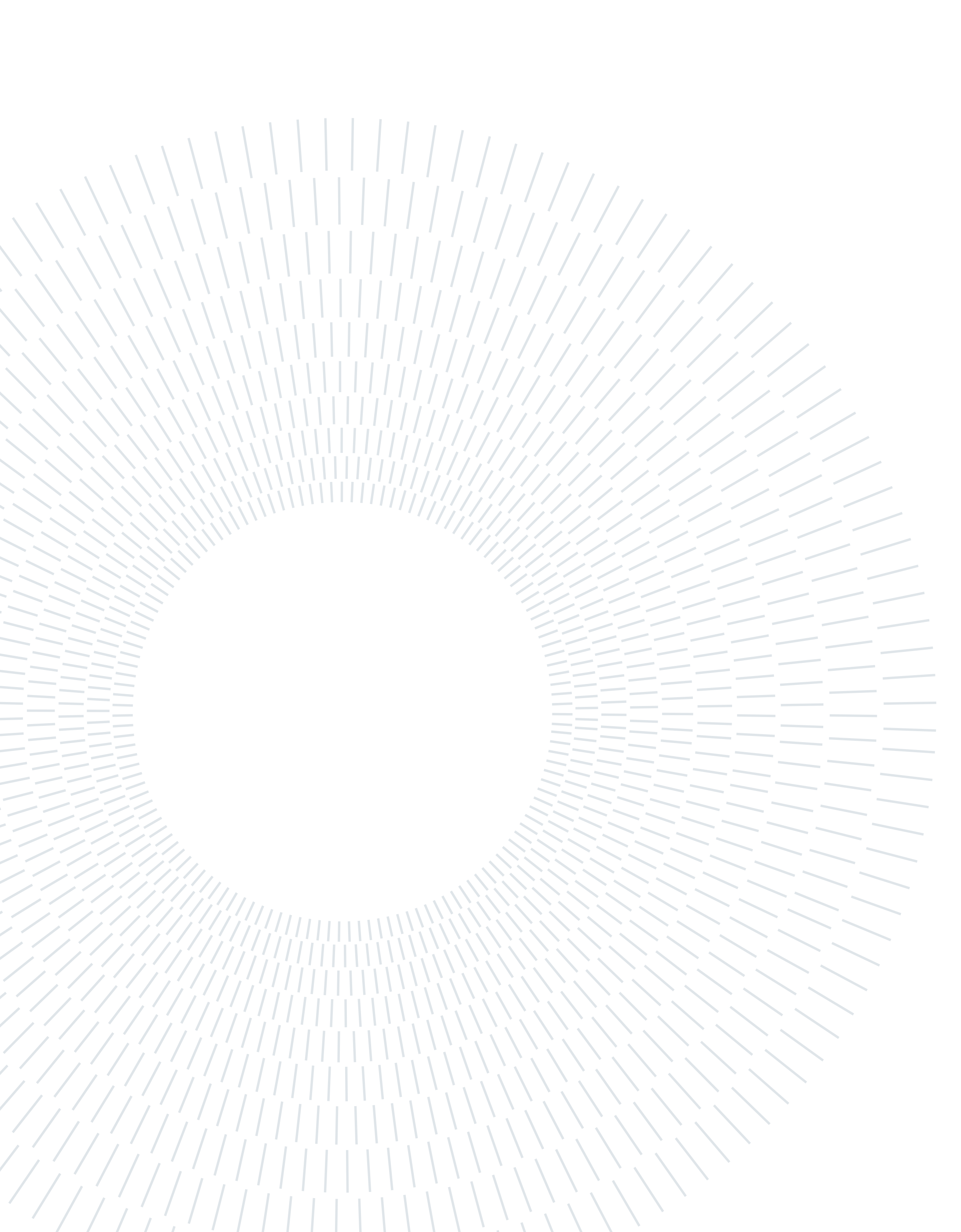
Abstract

The past decade has seen a growing drive for the simplification of cobot programming, aimed at making it within the reach of unskilled operators. Due to a higher demand for customized products, small and medium-sized enterprises require quicker and easier reconfiguration of the robots. Robot manufacturers need to look for more successful solutions, and academic research already offers some concepts, like *Semantic Programming by Demonstration*, that could be reformulated into new strategies for robot teaching.

Abstracting the meaning of actions, interpreted as the effect that they have on the scene and their purpose, is fundamental in semantic machines. The challenge is to make a machine capable of analyzing a scene to distinguish its salient features, such as the mutual relationships between the objects and the attributes of those objects. Such characteristics, specific to the operational context, are essential to subsequently being able to plan an action regardless of the circumstances of the case. The robot will interpret if the pre-conditions of an action are met, move accordingly, and finally verify the post-conditions that arise from the observation of the environment.

This thesis aims to develop an algorithm capable of extracting the relevant data of objects in a scene, checking for any relationships between instances, and constructing a *Scene Graph* whose structure is ordered and standardized according to past literature. The generation of the graph is complete, efficient, and real-time.

Keywords: robotics, semantics, scene, graph, recognition.



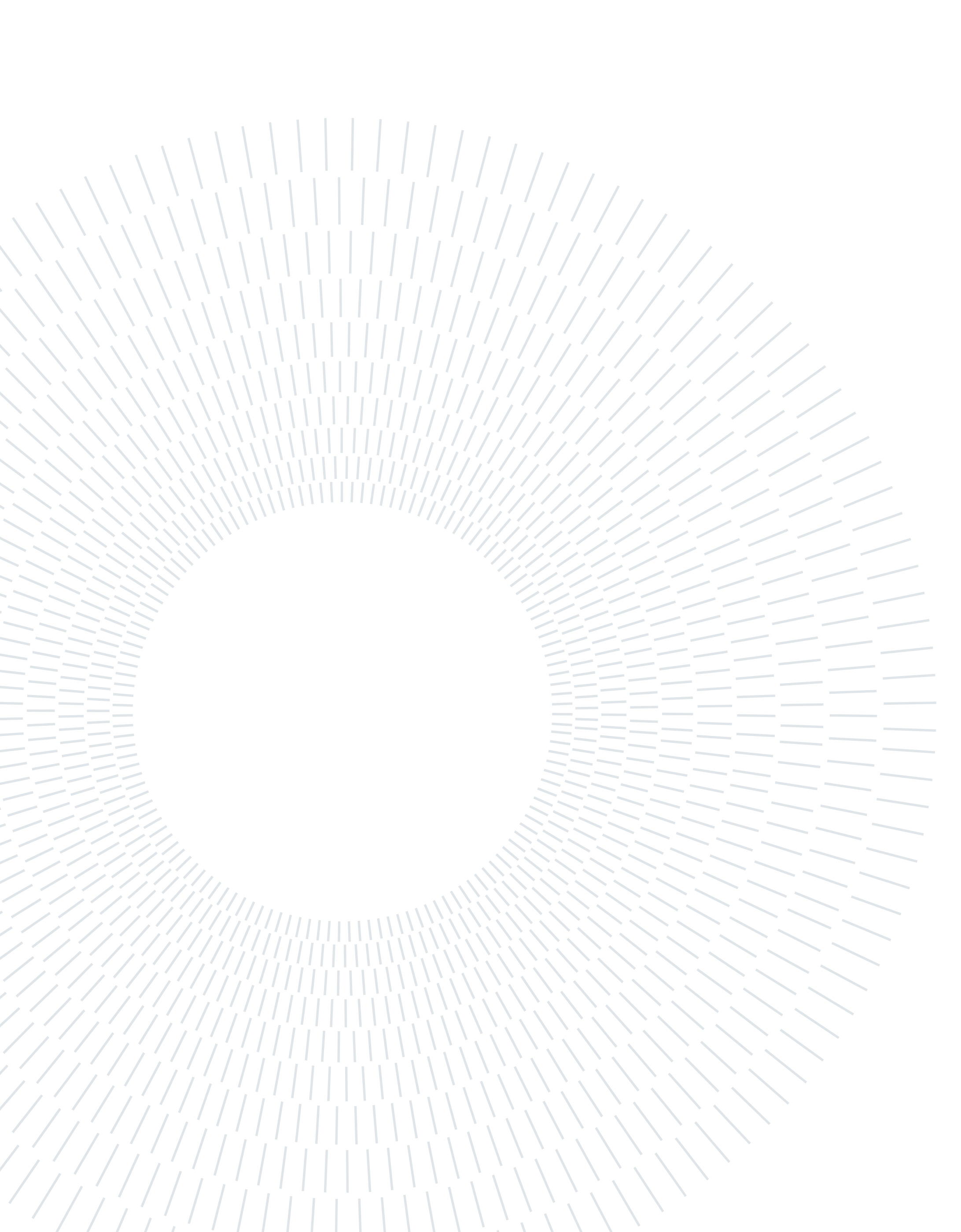
Abstract in lingua italiana

Negli ultimi dieci anni si è assistito ad una crescente spinta alla semplificazione della programmazione dei cobot, con l'obiettivo di renderla accessibile anche a operatori non qualificati. A causa della maggiore domanda di prodotti personalizzati, le piccole e medie imprese richiedono una riconfigurazione più rapida e semplice dei robot. I produttori di robot devono cercare soluzioni più efficaci, e la ricerca accademica offre già alcuni concetti, come la *Programmazione Semantica per Dimostrazione*, che potrebbero essere riformulati in nuove strategie per l'insegnamento dei robot.

L'astrazione del significato delle azioni, interpretate come l'effetto che hanno sulla scena e il loro scopo, è fondamentale nelle macchine che impiegano strumenti semantici. La sfida consiste nel rendere una macchina capace di analizzare una scena per distinguere le sue caratteristiche salienti, come le relazioni reciproche tra gli oggetti e gli attributi degli stessi. Tali caratteristiche, specifiche del contesto operativo, sono essenziali per poter successivamente pianificare un'azione indipendentemente dalle circostanze del caso. Il robot interpreterà se le pre-condizioni dell'azione sono soddisfatte, si muoverà di conseguenza e infine verificherà le post-condizioni che derivano dall'osservazione dell'ambiente.

Questa tesi mira a sviluppare un algoritmo in grado di estrarre i dati rilevanti degli oggetti in una scena, verificare eventuali relazioni tra le istanze e costruire uno *Scene Graph* la cui struttura è ordinata e standardizzata in base alla letteratura precedente. La generazione del grafo è completa, efficiente e in tempo reale.

Parole chiave: robotica, semantica, scena, grafo, riconoscimento.



Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**
- 1.1 Industrial automation trends 1
- 1.2 Semantics in robotics 2
- 1.3 Problem formalization 3
- 1.4 State of the art — Scene graphs 5
- 1.5 Thesis purpose and statement 7
- 1.6 Thesis achievements 8

2 Methodology **9**
- 2.1 Simulation of the 3D environment 9
- 2.2 Scene graph generation algorithm 10
  - 2.2.1 Basic programming concept 10
  - 2.2.2 Prior knowledge and data tables 11
  - 2.2.3 Algorithm overview 12
  - 2.2.4 Updating the scene graph 15
  - 2.2.5 Displaying the scene graph and other configurations 18

3 Experimental validation **21**
- 3.1 Test #1 — Stacking 22
- 3.2 Test #2 — Unstacking and tilting 32

4 Conclusions **41**

Bibliography **43**

A Semantic vocabulary of the program **45**
A.1 List of predicates 45
A.2 List of attributes 46
A.3 List of objects 46

List of Figures **49**

List of Tables **51**

Acknowledgements **53**

1 | Introduction

1.1. Industrial automation trends

Industry 4.0 has led to rapid progress in automation, in particular artificial intelligence, action planning, and control. The nature of these technologies — revolving around the concept of intercommunicability between machines (*Internet of Things*), self-diagnostics, simplification of the deployment process, and intelligent data manipulation — determined a large success among both large-capital firms and *SMEs* (small and medium enterprises).

This thesis project was conducted in the context of the implementation of collaborative robots (also known as cobots) in industrial scenarios, therefore the focus will not be on anything directly concerning service automation. Specifically, we are interested in an increasingly popular approach that aims to simplify cobot programming in order to make it more within the reach of a non-specialized end user, or operator. One of the end goals of this strategy is to ease the commissioning of robotic arms in the context of highly customized products, so as to increase flexibility while reducing non-operative time. Among the most famous paradigms, we cannot but mention the so-called *Programming by Demonstration* (PbD), which today constitutes the basis of various industrial solutions.

PbD is a way of programming that allows the end user to instruct the machine directly, without resorting to "classical" coding methods — which often coincide with the writing of a series of instructions — but rather by demonstrating in a series of physical movements the task to be performed. The procedure will therefore be learnt through the recognition of the robot's movements instructed by the operator through a kinesthetic or "teleoperated" approach.

However, one of the major shortcomings of traditional PbD techniques lies in the fact that the teaching phase is often limited to instructing the machine to follow a series of waypoints; thus, the machine is unable to generalize the *meaning*, in human terms, of the action, nor is it able to abstract it and apply it to even slightly different scenarios. *Skill reusability* is therefore not permitted.

In order to overcome this limitation, more suitable models are used to represent the "meaning" of actions and entities, i.e. models that take into account the *semantics* of the elements in question.

1.2. Semantics in robotics

If, on the one hand, in linguistics semantics is used to indicate the discipline that studies the meaning of the utterances of a language as the relationship between the signifier and the meaning of each element, as well as the meaning of relationships between the various meanings of a given sentence; on the other hand, the same term is used in philosophy to indicate a similar concept, namely the complex of theories of meaning [10]. In mathematics, the term takes on a further dimension, complementary to syntax, as a system of analytical tools for the interpretation — hence still an acknowledgment of meaning — of formulae expressed through conventional symbols; interpretations that are in turn evaluated as true or false depending on how the logical conditions apply.

Similarly, in robotics, the term semantics refers to «the meaning of places, objects, other entities that occupy the environment, or even the language used in communication between robots and humans or between robots themselves» [3], but also to the actions, which are to be interpreted by the machine as abstractions of a series of coordinated movements with the aim of changing the state of the working environment. As with semantics in the mathematical sense, the fundamental tool for semantic systems in robotics is logic. It is therefore evident that a better understanding of the meaning of what surrounds the robot and the actions to be performed allows a certain level of generalization of the problems faced by the machine, thus enhancing learning capabilities through manual demonstration. For these reasons, a more groundbreaking development of the research on Programming by Demonstration methods focuses on the so-called *Semantic PbD*.

To better understand the bigger framework in which the thesis lies, it is interesting to consider the kinds of applications that involve the concept of semantics in robotics. According to [3], the main areas of employment can be categorized as follows:

- SLAM (Simultaneous Localization And Mapping) algorithms;
- segmentation;
- object recognition.

For what concerns the taxonomy of the methods usually implemented for semantic robotics, the same survey subdivides it into two areas:

- Learnt semantics – the robot acquires the information on its own, which is further split into:
  - supervised;
  - unsupervised;
  - semi-supervised methods.
- Provided semantics – the robot is given knowledge beforehand.

1.3. Problem formalization

In order to enable the techniques inherent to Semantic Programming by Demonstration (or S. PbD), it is necessary to represent the environment in an adequate and, once again, semantic manner. That is to say, to offer to the teaching and action planning programs, data concerning the state of the scene, interpreted according to the useful meanings which depend on the specific work objectives, i.e. the hypotheses made concerning the kind of objects, events, actions and relations between entities that may occur. To this end, appropriate methods are needed to capture this information via sensors, extract the meanings and translate them into a semantic model, functional to the S. PbD.

Since we are focused on robotic arms in industrial environments, we must only deal with a certain range of scenarios, but its generalization is not straightforward. There might be many different entities of interest, in particular:

1. manipulable and non-manipulable objects;
2. the current state of the robot — e.g. the degree to which the gripper is open;
3. the environment surrounding the scene, such as walls, ceilings, and surfaces;
4. or even “groups of objects”, like a deck of cards or “clutter” — i.e. sets of elements that can be understood as a single individual to simplify their description [7].

Furthermore, not only it is useful to recognize the identity of the objects (the *class* to which each instance belongs) in the scene, but it is also beneficial, for the purposes of an adequate semantic representation, to acknowledge the properties specific to each instance and the relationships between them. Objects can be identified by their class, as well as by their current *pose* (position and orientation) in the scene, which directly affects the

portion of space they occupy depending on the specific geometry. Other attributes may be the dimensions, the *affordances* (features functional to a specific set of actions — e.g. the handle of a cup is used to grasp, the bowl part is to hold liquids), the color, the material, other physical properties (e.g. weight, opacity, elasticity, etc.), or even other states regarding the configuration (hovering, standing, etc.).

Yet, above all, the power of semantics in the representation of a scene mostly lies in the ability to identify and represent the relationships between instances of objects, e.g. proximity to, relative positions (x on top of y), comparisons (x bigger than y), or descriptions (x is holding y).

Once the "meaning" of the scene is understood, analysis strategies can be applied, such as the verification of the *preconditions* required for the robot to perform a given action and the verification of the *post-conditions* achieved as an effect of the same action. Consequently, the machine will be equipped with tools that will allow it to learn the transition model and reapply the same kind of action in different scenarios, enabling skill reusability.

It is indeed insightful to illustrate the classical object representation hierarchy quoted by [6], [1] and [8]. This model is composed of three main layers: from the most detailed to the most abstract object description.

- *Point level* (point cloud/pixel/voxel):
  - each point has material properties, color, etc.;
  - each point can be labeled to the object they belong to;
  - contact points can be recognized.
- *Part level* (set of features, e.g. shape, pose, material, surface, etc.):
  - to represent relations and interactions between parts;
  - to recognize the class of the object;
  - often, to define the affordances of the object.
- *Object level* ("separate" entities):
  - set of features is associated with each object;
  - object-specific properties (mass, pose, other uniform properties);
  - simplified representation in single entities.

A fundamental step that researchers must also take into consideration is the method applied to perceive the objects and collect data, as stated by [7] and [2].

There are two broad categories of sensors: passive and interactive. *Passive* sensors — like cameras — can perceive the state of the environment avoiding physical interactions; while in *interactive* perception — e.g. estimations of weight from the power required to drive a robotic arm, LiDARs, and RGB-D cameras — the machine has to interact with its surroundings in order to obtain information. Both technologies have advantages and disadvantages, regarding energy efficiency, speed, accuracy, and practicality. This means that depending on the context, it will be appropriate to use the most suitable perception method.

Regarding the extraction of position data, the available computer vision algorithms are countless, as well as beyond the scope of this thesis. It is rather interesting to note the existence of local features-based methods (like *RoPS* [5][4]) for object recognition, meaning that there are indeed solutions to make reasonable assumptions about the pose of the objects in cluttered scenes, where entities may be partially hidden by other objects.

Chapter 1.4 will answer two questions: what kind of semantic data representation models were defined by past literature? how, given a scene, do we generate a representation that is effective, efficient, and standardized?

1.4. State of the art — Scene graphs

According to [9], there are multiple methods for organizing the semantic data of a given scene: *scene graphs*, affordance-, syntactic-, and knowledge-based methods. Syntactic-based strategies offer a rich structure of syntax and semantics that represents domain-free rules and contexts.

While the last three have their advantages — respectively, the recognition of complementary features between the environment and the agents, the syntactical interpretation of subsequential activities, and the inference of relationships with partially observable information —, scene graphs (SGs) are definitely more organized, compact, and potentially less redundant.

Scene graphs are used for several activities [11]:

- SG generation — the process of extracting semantic data from a scene to compose a corresponding SG;
- image captioning — automatic assignment of a title to an image;

- image generation — creation of an image given the semantic description in the form of an SG;
- referring expression — highlighting of the region of an image corresponding to the given expression, which in turn corresponds to a certain SG segment;
- image retrieval — research of an image within a database corresponding to the semantic description;
- Visual Question Answering (VQA) — a tool that can answer questions that require complex reasoning about the scene, which works in parallel with automatic analysis of the SG.

To better understand what scene graphs are, it is worthwhile to introduce their mathematical definition [11].

Given any scene $\mathcal{S}$, a scene graph is a set of visual triplets $T_S \subseteq \mathcal{O}_S \times P_S \times (\mathcal{O}_S \bigcup A_S)$; where:

- $\mathcal{O}_S$ is the set of objects in the scene, each of which is defined as a two-element tuple $o_{S,k} = (l_{S,k}, b_{S,k})$, where $l_{S,k}$ is the label, or name, of the object, and $b_{S,k}$ is the area it occupies in the scene (bounding box, or BB);
- A_S is the set of attributes, i.e., the properties that can be achieved by objects in the scene;
- P_S is the set of relations or predicates, including the relation "is" (to be).

Additionally, there is an ulterior condition that binds each visual triplet t_S to assume one of the two following configurations:

- Relational triplet: $t_{S,i} = (s_{S,i}, p_{S,i \to j} o_{S,j}) \Leftarrow$ (Subject, predicate, object)
- Descriptive triplet: $t_{S,i} = (s_{S,i}, p_{S,is}, a_{S,k}) \Leftarrow$ (Subject, "is", attribute)

From this definition, it is clear that this method allows the complexity of a wide variety of scenes to be adequately recorded by a visual triplet list in an efficient and convenient, and — above all — in a standardized and generalized manner. Indeed, it not only lends itself well to the representation of objects, attributes, and relationships, but it is also sufficiently ordered to allow the implementation of complete graph search and analysis systems. This is especially useful because it allows the creation of query mechanisms in which the machine itself automatically investigates the scene and verifies whether a certain action is enabled by the existence of the relevant precondition. This potentiality of SGs becomes quite obvious once we visualize the scene graph mathematically defined

above as being graphically represented by nodes (objects) connected by oriented edges, each of which is labeled according to the type of predicate involved. It thus implies that it will lend itself well to the application of traditional graph analysis tools.

In anticipation of the upcoming explanation of the work that was done for this thesis project, it must be noted that the definition of scene graphs given by Guagming Zhu et al. [11] reported above was not applied in this work, even though it is still a good explanation of the concept. Further details about the mathematical definition of the version of SG that was developed for the project will follow in subsection 2.2.1.

1.5. Thesis purpose and statement

Once the techniques for multi-modal scene data collection have been defined and implemented, this thesis aims to generate a complete and dynamic SG.

- *Complete* in the sense that the data represented in this way must be sufficient and appropriate to the working conditions and objectives selected as the hypothesis; for example, the force measured by the load cell at the end effector is generally not of interest for grasping operations involving constant-weight objects.
- *Dynamic* because the system must be able to update the set of visual triplets that make up the SG in real-time, to analyze or verify the effects of an action.

A further traditional target of these kinds of algorithms is *non-redundancy*, which aims to obtain a minimal representation, to reduce the dimensionality of the problem: for example, it is obvious that if A is above B, and B is above C, it will be superfluous to specify that A is above C. While avoiding redundancy could be considered an interesting objective in the development of a scene graph generation algorithm, it was not taken into account in this thesis. This means that, for example, an object A which is positioned on top of another object B will be featured in two triplets: the first one representing the relationship between object A and object B; and the second (extra) one describing the top object A as “stacked”. This will make further computations of queries like “which objects are stacked in the scene?” faster and more trivial since it is a straightforward search problem (match with pattern “x - is - stacked”) within the SG.

Our approach falls in the category of the provided semantics methods discussed in Chapter 1.2, as we will see during the explanation of the algorithm, in particular in Chapter 2.2.2.

1.6. Thesis achievements

In this thesis the following objectives have been achieved:

- Translation of the data collected from the scene into a semantic entity-properties structure format.
- Modelization of an SG architecture arranged according to a hierarchical class framework.
- Development of an original generation algorithm that takes into account prior knowledge (in particular, limitations to the range of scenarios or interactions that can occur in that kind of environment) and produces the SG in an intuitive, efficient, reliable, complete way.

The algorithm was implemented in a C++ script that updates the SG with a configurable cycle time. Subsequent analysis revealed that the algorithm is extremely resource-efficient, taking up only a small amount of time to finish the required computations. Tests were conducted for pick-and-place operations such as stacking, unstacking, and tilting of an object. The program is easily scalable to account for more instances, object types, robots, predicates, and attributes.

The thesis is structured in the following way:

- Chapter 2 — Methodology.
  - Section 2.1 — Simulation of the 3D environment: where we explain how the 3D simulation was set up and how the communication to the SG generation program happens.
  - Section 2.2 — Scene graph generation algorithm: the design of the program is illustrated in detailed sections about the class model, the prior knowledge structures, the algorithm itself, and the output image generation.
- Chapter 3 — Experimental validation: here we show the data gathered from two pick-and-place test cases and we confront them to the expected outcome. A brief analysis of the computational times is included.
- Chapter 4 — Conclusions: where we summarize the results of our work and the achievements.

2 | Methodology

2.1. Simulation of the 3D environment

As previously mentioned, to replicate a physical scene featuring a robotic arm and several objects, we relied on a simulation of a virtual 3-dimensional environment. This accomplishment was achieved through the use of a simulation environment, *CoppeliaSim*, which is interfaced with a Python script that implements the functions provided by the library *PyRep*. The script is used to control the simulated robot movement and retrieve data on the state of the entities in the scene.

The first phase consists of the instantiation of the objects in the scene. During this stage, we specify some immutable characteristics, e.g. geometry, size, color, and variable features, like position and orientation. The second step requires us to specify the actions or events taking place during the simulation. Here we instruct the robot to follow a series of move and gripper commands to perform a pick-and-place operation.

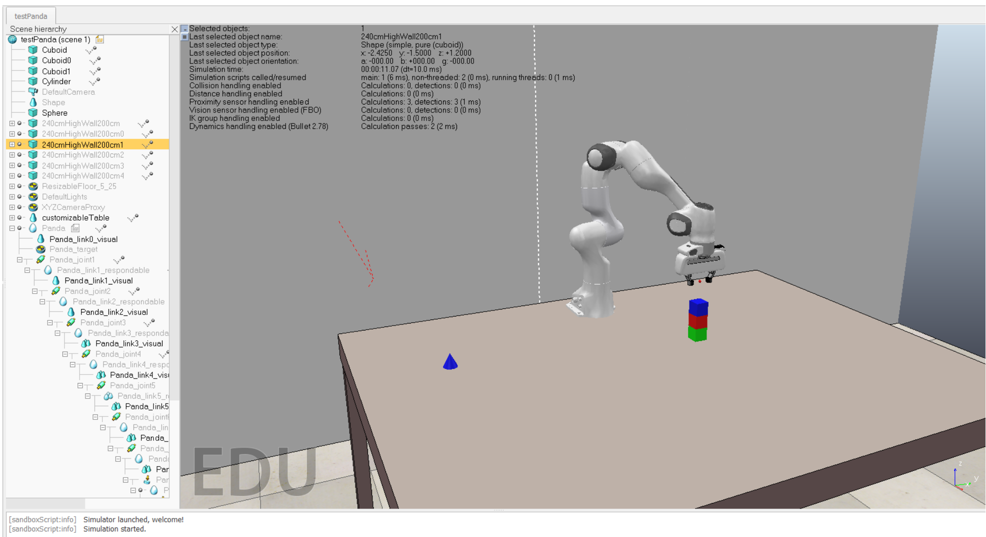


Figure 2.1: A screenshot from the CoppeliaSim environment

While the simulation of a virtual 3D environment is running, we need to send the information about the pose of the involved objects as a stream of data that feeds into the SG

generation algorithm. To do so, the Python script also had to manage a socket communication, which allows the creation of inter-process communication (IPC). The process running the Python script will act as a server in the TCP communication. It will accept the connection to a client — i.e. the process of the SG generation algorithm written in C++ —, then, while the simulation is still running, the server process will collect the data from the instances in the scene, pack them into a list of numeric values and send them to the client through the socket.

At the other end of the communication, the client C++ script applies a class object, `WorldObserver`, and a method, `UpdateObjects()`, to translate the values received through the socket and store them into the corresponding `ObjectNode` instances. Further computations to dynamically construct a scene graph based on these data are explained in the next section.

2.2. Scene graph generation algorithm

2.2.1. Basic programming concept

Taking now into consideration the SG generation program, it is necessary to first discuss the structure of the scene graph model itself. The definition of SG given by Zhu et al. [11] and reported in Chapter 1.4 was adapted to an equivalent representation that alters the descriptive triplet from a structure of type (object, is, attribute) to a structure of type (object, attribute, object). The advantage of this solution is the fact that we discarded the predicate of type "is" and that only object instances, not attributes, are the actual nodes of the graph. This allows a cleaner, more intuitive representation in which we avoid attribute nodes from being at the end point of multiple "is" relationships. In both cases, a descriptive triplet necessarily contains redundant information: either we have an attribute as the third element subsequent to an "is" predicate; or we must make sure that whenever the second element is an attribute, the third one is the same instance featured as the subject.

The mathematical definition would therefore become the following: a scene graph is a set of visual triplets $T_S \subseteq \mathcal{O}_S \times (P_S \bigcup A_S) \times \mathcal{O}_S$; where $\mathcal{O}_S$ is the set of objects in the scene, A_S is the set of attributes, P_S is the set of predicates. A visual triplet can only be of either type relational or descriptive:

- Relational triplet: $t_{S,i} = (s_{S,i}, p_{S,i\to j}, o_{S,j}) \Leftarrow$ (Subject, predicate, object)
- Descriptive triplet: $t_{S,i} = (s_{S,i}, a_{S,k}, s_{S,i}) \Leftarrow$ (Subject, attribute, subject)

From a coding perspective, this definition was implemented through a class object `SceneGraph` composed by a vector of `VisualTriplet`'s. Each `VisualTriplet`, in turn, features three elements: an `ObjectNode` called `subject`, an `Edge`, and a second `ObjectNode` referred to as `object` of the relationship.

The class `Edge` represents the connection between the two nodes and is graphically represented as a directed arrow. Two further classes, `Attribute` and `Predicate`, are extensions of `Edge`. They both feature a label string, demarcating the name of the relation, and a type identifier.

For what concerns `ObjectNode`, the class is once again characterized by a label string and a type that affects the geometry of the represented object, though it also features further data relative to the pose — represented by a 4x4 matrix —, and the top and bottom surfaces. This information is critical to enable subsequent computations regarding the verification of the conditions required for spacial relations such as "Object A is on top of object B" or "Object C is upright". While the algorithm runs, the SG generation process will make sure that the data regarding the pose and surfaces are updated at an adequate rate.

2.2.2. Prior knowledge and data tables

In order to achieve better computational performances, the SG generation algorithm was developed as a provided semantics method, meaning that the robot is given knowledge beforehand. This means that during the search for new potential visual triplets belonging to the SG, the program will only be considering plausible scenarios, i.e. certain acceptable combinations of relations. Therefore, the search is only conducted within a limited range of possibilities, mostly related to the type of objects involved with certain predicates or certain attributes.

During the initialization phase of the program, the runtime will read the data from selected `.csv` files and store them into the appropriate `InitializationTables` object, which is constituted by three elements:

- `AdmissibleAttributes` — a list of plausible (subject type, attribute type, subject type) descriptive triplets, e.g. `(cube, on_a_side, cube)` is possible, while `(cone, upside_down, cone)` is not;
- `AdmissiblePredicates` — a list of plausible (subject type, predicate type, object type) relational triplets, e.g. `(cube, on_top_of, cube)` is possible, while `(cube, on_top_of, cone)` is not;

- `CompatibleAttributes` — a list of compatible descriptors for the same object instance, e.g. `(cube, on_a_side, cube)` cannot be true while `(cube, upright, cube)` is true.

Further prior knowledge available to the program is the range of types and sizes regarding the objects, and the types of predicates and attributes, all represented through an enumerator value (see Appendix A).

2.2.3. Algorithm overview

Once the initialization phase is over and the connection to the 3D simulation data stream through the socket is established, the SG generation algorithm can begin its operations. The number of instances to track and their geometry class is already specified by the incoming data, so a vector of objects can be allocated.

Algorithm 2.1 Scene Graph generation

```
1:  while SimulationOngoing do
2:      StartTime ← GetStartTime( )
3:      SceneGraph.WasChanged ← false
4:      RawData ← ReceiveData( )
5:      Objects ← UpdateObjects(RawData, Objects)
6:      SceneGraph ← UpdateSceneGraph(SceneGraph, Objects)
7:      if SceneGraph.WasChanged then
8:          DisplaySceneGraph( )
9:      end if
10:     EndCycle(Objects, SceneGraph)
11:     SynchronizeCycle(StartTime)
12: end while
```

As we can see from the Algorithm 2.1, the SG update process is repeated cyclically for as long as the simulation lasts. On lines 2 and 11 of the pseudocode, the program ensures that the cycle time is maintained throughout the whole execution, setting the thread to sleep for the remaining time. This guarantees that the operations are conducted regularly at a predictable rate, which is an optimal condition for testing. The line 4 was implemented through the methods of the `WorldObserver` class mentioned in Chapter 2.1. It allows the algorithm to gain access to raw data read from the socket to the virtual environment. This same data is then translated to the corresponding `ObjectNode` variables on line 5 using the method `UpdateObjects()`. Finally, the function `UpdateSceneGraph()` proceeds with

the update of the scene graph itself using the new object information and the previous state of the graph to prune old triplets that are no longer valid and create new ones if the conditions apply. Since this step (line 6) also keeps track of any changes that were made to the SG, the operation of displaying a graphical representation of the updated scene graph is run only when requested, so to avoid resource-intensive procedures.

Algorithm 2.2 Update objects method

```
1:  function UpdateObjects(RawData, Objects)
2:      for i := 1 to |Objects| do
3:          Objects[i].WasUpdated ← false
4:          NewPoseMatrix ← TranslateToPoseMatrix(RawData, i)
5:          if IsDifferentEnough(NewPoseMatrix, Objects[i].Pose) then;
6:              Objects[i].PoseMatrix ← NewPoseMatrix
7:              Objects[i].TopSurface ← ComputeTopSurface(Objects[i].Pose)
8:              Objects[i].BottomSurface ← ComputeBottomSurface(Objects[i].Pose)
9:              Objects[i].WasUpdated ← true
10:         end if
11:         if Objects[i].Type = Gripper then
12:             NewState ← GetGripperState(RawData, i)
13:             if NewState ≠ Objects[i].State then
14:                 Objects[i].State ← NewState
15:                 Objects[i].WasUpdated ← true
16:             end if
17:         end if
```

Given the fact that efficiency is critical to have a scalable SG generation algorithm, the method `UpdateObjects()` had to be implemented in an appropriate way. In other words, the update should be performed only when strictly necessary for tracking the scene correctly. During the first execution of the method, the machine creates and allocates the corresponding memory for the `ObjectNode` elements featured in the scene, assigning to each one a label and the geometry type (in our tests, either a cube, a cone, or a gripper) according to the information gathered from the scene. Both at the first and any subsequent execution (see pseudocode 2.2), the algorithm cycles through every object instance and performs the following operations: it translates the socket data in x, y, z positions and x, y, z rotations; it constructs the corresponding 4x4 homogeneous transformation matrix; it updates the pose, and top/bottom surfaces data whenever the new matrix is detected to be different enough from the previously stored one; and it stores any addi-

tional information, e.g. in the case of a gripper, it saves the end-effector's state as `open` or `closed`. On top of the computations discussed, we also ensure that we keep track of whether or not the data relative to each object was modified during the cycle.

Consider two 4x4 matrices representing the position of the same object at two subsequent cycles, $\boldsymbol{A}$, the old one, and $\boldsymbol{B}$, the new one. $\boldsymbol{A}$ is constituted by the 3x3 rotation matrix $\boldsymbol{R}_A$ and the vertical position vector $\vec{p_A}$, while $\boldsymbol{B}$ is described by $\boldsymbol{R}_B$ and $\vec{p_B}$, therefore:

$$A = \begin{bmatrix} & \boldsymbol{R}_A & & \vec{p_A} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} & \boldsymbol{R}_B & & \vec{p_B} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Because of the fact that the pose signal could have been affected by random noise, and having new poses means that we will have further computations for the surfaces and the upcoming graph generation, it is a good thing to check the degree of variation in the pose before deciding if it is worth updating it. To determine it, the program implemented a condition check that is set to true in two scenarios. Either the distance $|\vec{p_A} - \vec{p_B}|$ is greater than a chosen threshold $\overline{\rho}$, which corresponds to a relevant shift in the position; or the angle between the two orientations is higher than a second threshold $\overline{\theta}$, which means that the rotation $\mathbf{R}_A{}^T\mathbf{R}_B$ between the two orientations was translated into an axis-angle representation — implemented through methods specific to the C++ *Eigen* library — whose angle must be sufficiently high in order to be considered relevant.

The next step in the update of the object's pose is the computation of the top and bottom surfaces. They will be significant during the validation of conditions like "Object A is on top of object B" to determine if the two objects are, indeed, in contact across a certain surface area. The process considers the spatial collocation of the features of each entity taking into account the specific geometry type and the current pose. It then refers to an array of surfaces, each represented by the orthogonal vector and by the set of ordered vertices that delimit it. To determine which surface is to be considered the top or the bottom one, the algorithm simply compares all surface directions to the up- and down-pointing vectors to check whether they are parallel enough. The surfaces are therefore recognized with a straightforward computation and labeled as *top* or *bottom* for future use.

2.2.4. Updating the scene graph

The method `UpdateSceneGraph()` is the core of the algorithm. It consists of two phases: graph pruning, where old triplets that are no longer valid are removed from the SG; and graph building, where new valid triplets are added.

Algorithm 2.3 Scene Graph update

1: **function** UPDATESCENEGRAPH($SceneGraph, Objects$)
2: $SceneGraph \leftarrow$ DESTROYOLDTRIPLETS($SceneGraph.Triplets, Objects$)
3: $SceneGraph \leftarrow$ CREATENEWTRIPLETS($SceneGraph, Objects$)

During the pruning phase (Algorithm 2.4), the program cycles through every triplet belonging to the SG and checks if it is of type relational or descriptive. In the first case (lines 3 to 11), if any of the two instances involved as subject or object of the relational triplet was not affected by the update of the object data, no other operation is performed. In the opposite case, the machine will prune the triplet from the graph as long as the objects are either too far apart to have any kind of relation, or if the predicate acting as the edge between the two nodes is no longer valid. A similar computation is done for descriptive triplets (lines 12 to 19), where we will only check whether it is true that the subject node was changed in the previous phase and if the attribute is still valid for the instance taken into consideration.

Algorithm 2.4 Graph pruning

```
1: function DestroyOldTriplets(Triplets, Objects)
2:     for i := 1 to |Triplets| do
3:         if Triplets[i].Type = Relational then
4:             (Subject, Predicate, Object) ← Triplets[i]
5:             if Subject.WasChanged or Object.WasChanged then
6:                 if not AreObjectsCloseEnough(Subject, Object) or
7:                 not IsRelationalTripletValid(Triplets[i], Destroying) then
8:                     Prune(Triplets[i])
9:                     SceneGraph.WasChanged ← true
10:                end if
11:            end if
12:        else if SceneGraph.Triplets[i].Type = Descriptive then
13:            (Subject, Attribute) ← Triplets[i]
14:            if Subject.WasChanged then
15:                if not IsDescriptiveTripletValid(Triplets[i], Destroying) then
16:                    Prune(Triplets[i])
17:                    SceneGraph.WasChanged ← true
18:                end if
19:            end if
20:        end if
```

Notice how the methods for validity checks at lines 7 and 15 require a second enumerator input which specifies that we are actually verifying if we can *destroy* the triplet, rather than *create* it. This detail will be explained later.

Finally, a similar sequence of operations is done during the generation phase. We first look for any new potential descriptive triplet (Algorithm 2.5) by cycling through every object that was changed and for each of them we consider every attribute compatible with it according to the prior data extracted from `AdmissibleAttributes`. For each combination, as long as the corresponding triplet is not already present in the SG, the program will verify whether or not the conditions for the validity of the descriptive triplet apply. If they do, the triplet will be created and added to the graph.

Algorithm 2.5 Graph building — descriptive triplets

```
1: function CreateNewDecriptiveTriplets(SceneGraph, Objects)
2:     for i := 1 to |Objects| do
3:         if not Objects[i].WasChanged then
4:             break
5:         end if
6:         PotentialAttributes ← AdmissibleAttributes(Objects[i])
7:         for k := 1 to |PotentialAttributes| do
8:             NewTriplet ← (Objects[i], PotentialAttributes[k], Objects[i])
9:             if not IsTripletAlreadyPresent(SceneGraph, NewTriplet) and
10:            IsDescriptiveTripletValid(NewTriplet, creating) then
11:                AddTriplet(SceneGraph, NewTriplet)
12:                SceneGraph.WasChanged ← true
13:            end if
```

Looking now for new potential relational triplets (Algorithm 2.6), we must consider every ordered object combination (o_i, o_j) where $i \neq j$ and at least one of the two objects was updated. For each ordered combination, there is an associated array of admissible predicates (prior knowledge, explained in 2.2.2) that could act as the edge of a potentially valid triplet. The algorithm then tests each potential triplet, verifying first if the objects are close enough to have any kind of interaction, if the triplet is already featured in the SG, and finally if the validity conditions apply.

Since we are not trying to generate a minimal representation of the scene, we introduce an additional method, `CreateRedundantDescriptiveTriplets()`, which makes sure that on top of the relational triplet, we also introduce redundant descriptive triplets which will make the pre- or post-condition check straightforward. For example, when dealing with a triplet of type `(cube_1, on_top_of, cube_2)`, we will also create `(cube_1, stacked, cube_1)` — because the object on top can be part of an unstacking action —, `(cube_2, blocked, cube_2)` — since the second objects can no longer be moved —, and `(cube_1, graspable, cube_1)` — because it can be manipulated. The same goes for `(gripper, holding, cube_3)` and `(cube_3, held, cube_3)`.

Algorithm 2.6 Graph building — relational triplets

```
1: function CreateNewRelationalTriplets(SceneGraph, Objects)
2:     for i := 1 to |Objects| do
3:         for j := 1 to |Objects| do
4:             if i ≠ j and not Objects[i].WasChanged and
5:             not Objects[j].WasChanged then
6:                 break
7:             end if
8:             PotentialPredicates ← AdmissiblePredicates(Objects[i], Objects[j])
9:             for k := 1 to |PotentialPredicates| do
10:                NewTriplet ← (Objects[i], PotentialPredicates[k], Objects[j])
11:                if AreObjectsCloseEnough(Objects[i], Objects[j]) and
12:                not IsTripletAlreadyPresent(SceneGraph, NewTriplet) and
13:                IsRelationalTripletValid(NewTriplet, creating) then
14:                    AddTriplet(SceneGraph, NewTriplet)
15:                    CreateRedundandDescriptiveTriplets(SceneGraph, NewTriplet)
16:                    SceneGraph.WasChanged ← true
17:                end if
```

While checking the validity of a triplet is particularly dependent on the specific implementation and the kind of predicates or attributes involved for the single application environment, it is quite important to consider different ranges when *creating* or *destroying* a triplet. If on one hand there might be different interpretations of which are the thresholds to be considered when checking, for example, when two objects are on top of each other; on the other hand, it is clear that the range of possibilities for which the relationship is corroborated when creating should be narrower than the cases in which we decide to destroy the relation. In other words, we must be absolutely sure that a triplet is true when we want to create it, while we must be more possibilistic when we want to destroy it. This is done to avoid constant retriggering of the condition due to small fluctuations in the measurements, either due to noise or real-world temporary instabilities.

2.2.5. Displaying the scene graph and other configurations

Since we were dealing with graphs, it was evident that a graphical representation could only improve the perception of the final results. The code features a set of functions used to generate a *DOT* — a standard graph description language — representation of the Scene Graph. The .dot file is then fed into a function from a library called *GraphViz*, which

is capable of producing a corresponding PNG image. It features the object instances as nodes of a directed graph, whose edges are represented by arrows and labeled accordingly.

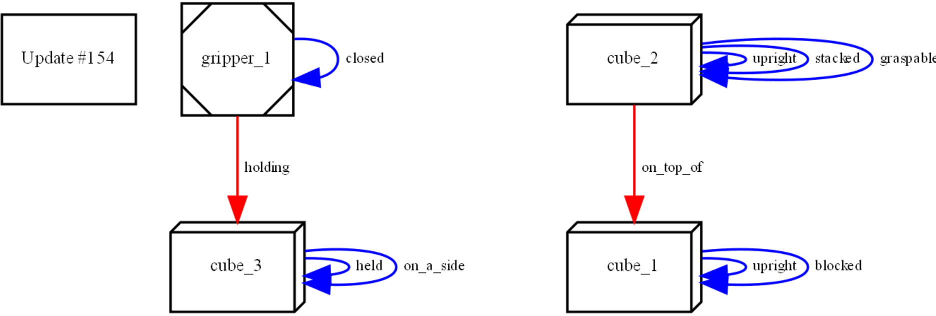
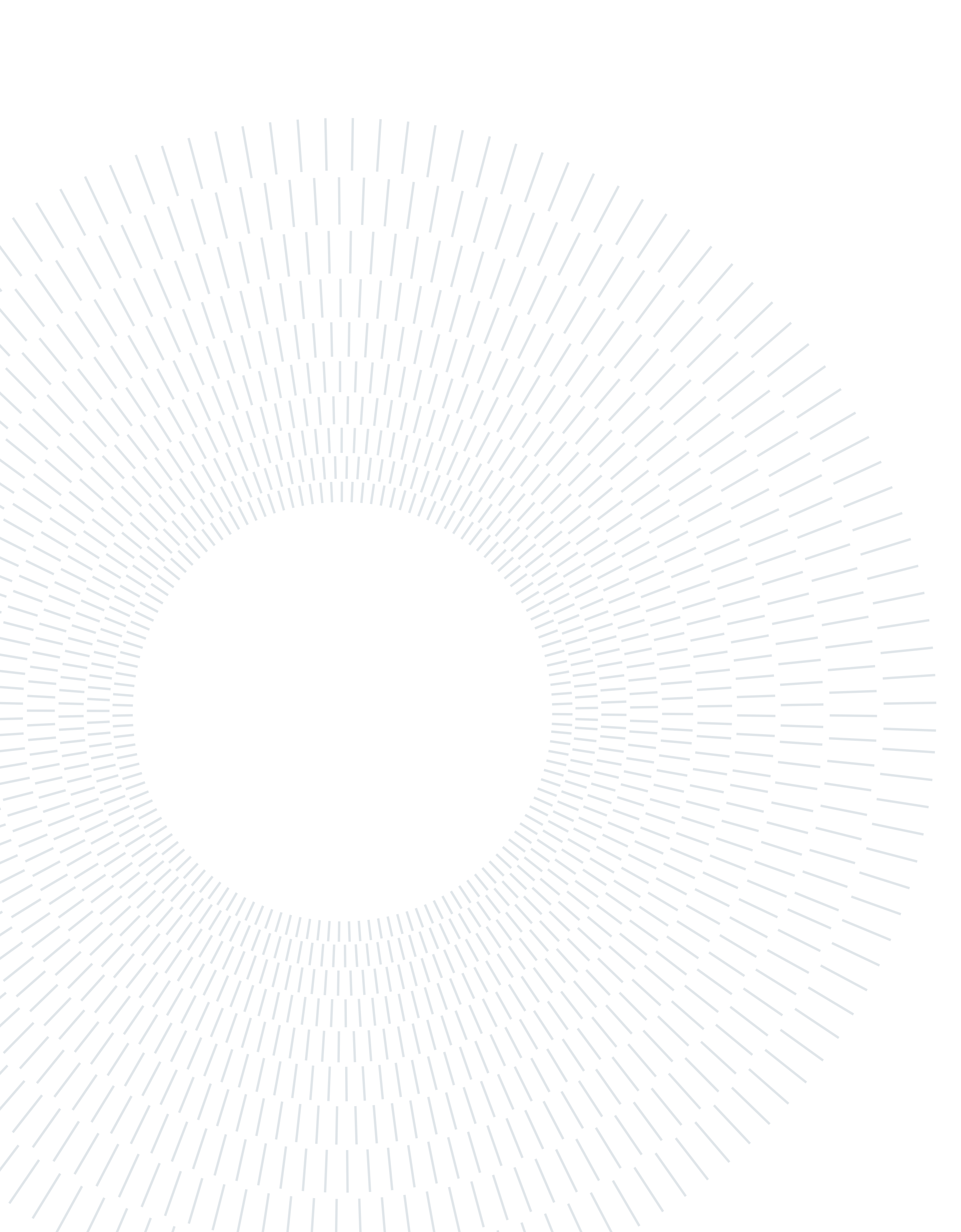


Figure 2.2: An example of graphical output

The image is then displayed in a separate window while the generation program and the simulation are running so that the graphical representation is always in sight and updated to the latest version of the SG. This was achieved through other methods of the *OpenCV* library.

Additionally, the code was developed taking into consideration the possibility of parameter tuning, particularly the ones used as thresholds in the validation check of triplets. The config header also contains print settings, file paths for input prior data tables and output drawings, socket definition, object dimensions, and rounding options.



3 | Experimental validation

The algorithm explained in Chapter 2.2 was implemented in C++ language, while the script to control the simulation from Chapter 2.1 was coded in Python. To prove our achievements, this section of the thesis will focus on the experimental validation of the system. In order to verify that the algorithm is working as intended, we must first remind ourselves about the objectives.

The machine should be able to gather from the scene simulated in the CoppeliaSim environment data about the objects' states and extract the meanings in terms of relations between objects — through predicates —, and in terms of descriptions of single instances — through attributes. The output representation should be configured according to the definition of Scene Graph reported in Chapter 2.2.1, which will then be translated into the corresponding image output (see Chapter 2.2.5).

First of all, we will have to verify that at each stage of the simulation, the resulting SG will be the one expected. Therefore, the output of the program must be backed up by the human interpretation of the scene within the same conceptual frame. In other words, the human interpretation must rely on the same semantic model, i.e. the SG, and the same set of available descriptors, i.e. list of predicates and attributes. This test ensures the *completeness* of the representation.

Secondly, the process must run efficiently and track the simulation in real-time. No lag beyond the duration of the cycle should be allowed, as that would mean that the algorithm is not efficient enough to follow the actual development of the scene. The *efficiency* of the system can be affected in multiple ways. Either we can have a lot of actions happening at the same time, or the machine has to keep track of a relatively large number of objects.

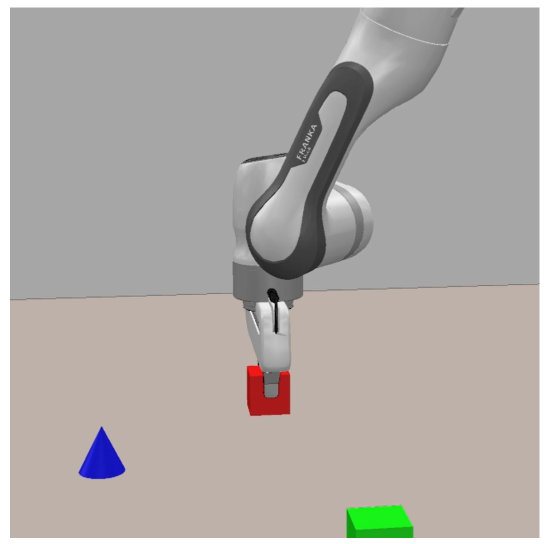
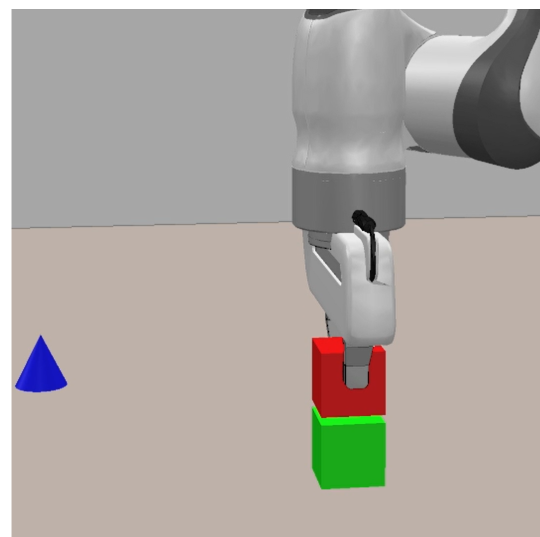
3.1. Test #1 — Stacking

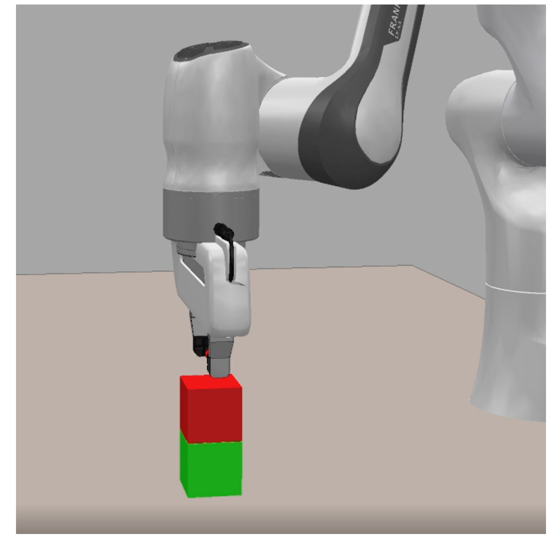
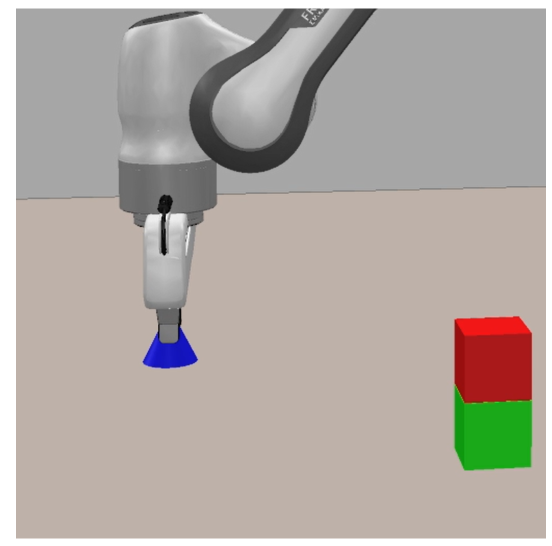
In our first test, we consider a stacking operation involving two cubes and a cone. In order to do it, we instruct the simulation with the corresponding steps that the robot arm should follow, and then we check the results.

Initially, the three objects are standing on the table surface too far apart to have any kind of interaction, therefore we won't expect any predicate edge between the corresponding nodes. The gripper is open and empty. The procedure begins, and the end-effector is moved above the first cube. The gripper closes and clutches the cube, then moves toward the second cube while holding the other. The robot brings the cubes into contact and finally releases the grip so that it can now leave cube 1 in the stacked position. The operation is then repeated for the cone, which will be positioned on top of the first cube.

Considering that we share the same conceptual framework (see Appendix A) of the robot, the expected outcome is the following:

Operation description	**Human interpretation**
1. Two cubes and a cone are standing on a table surface far from each other, the gripper is open.	*Only descriptive triplets. Gripper 1 is open. Cube 1, Cube 2, and Cone 1 are upright and graspable.*

2. The gripper reaches the first cube and holds it. 	*Gripper 1 is closed and is holding Cube 1. Cube 1 is no longer graspable, but instead held and upright. Cube 2 and Cone 1 are still upright and graspable.*
3. The robot brings the first cube on top of the second one, while the gripper still holds it. 	*Gripper 1 is still closed and holding Cube 1, while Cube 1 is held, upright, and stacked. Cube 1 is also on top of Cube 2, which is no longer graspable, but blocked and upright. Cone 1 is graspable and upright as before.*

4. The gripper releases the first cube. 	*Gripper 1 is now open and without any other relationship. Cube 1 is no longer held, but graspable, upright, and stacked. Cube 1 is also on top of Cube 2, which is upright and blocked. Cone 1 is graspable and upright.*
5. The gripper reaches the cone and grasps it. 	*Gripper 1 is closed and holding Cone 1. Cone 1 is upright and held. The two other cubes are unchanged: Cube 1 is upright, stacked, graspable, and on top of Cube 2. Cube 2 is upright and blocked.*

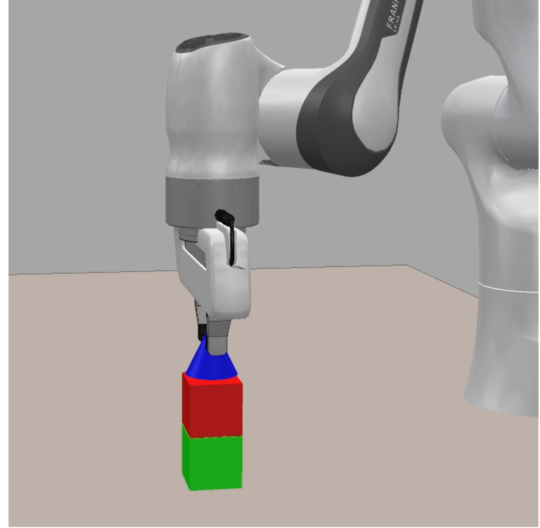
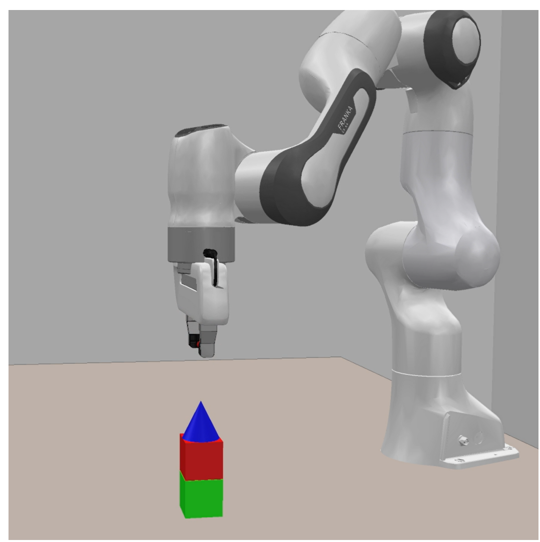
6. The robot brings the cone on top of the first cube, while the gripper still holds it. 	*Gripper 1 is closed and is holding Cone 1, while Cone 1 is held, upright, and stacked. Cone 1 is also on top of Cube 1, which is no longer graspable nor stacked, but blocked and upright. Cube 1 is on top of Cube 2, which is still blocked and upright.*
7. The gripper releases the cone. 	*The tower is now complete. Gripper 1 is now open and without any other relationship. Cone 1 is no longer held, but graspable, upright, and stacked. Cone 1 is on top of Cube 1, which is upright and blocked. Once again, Cube 1 is also on top of Cube 2, which is upright and blocked.*

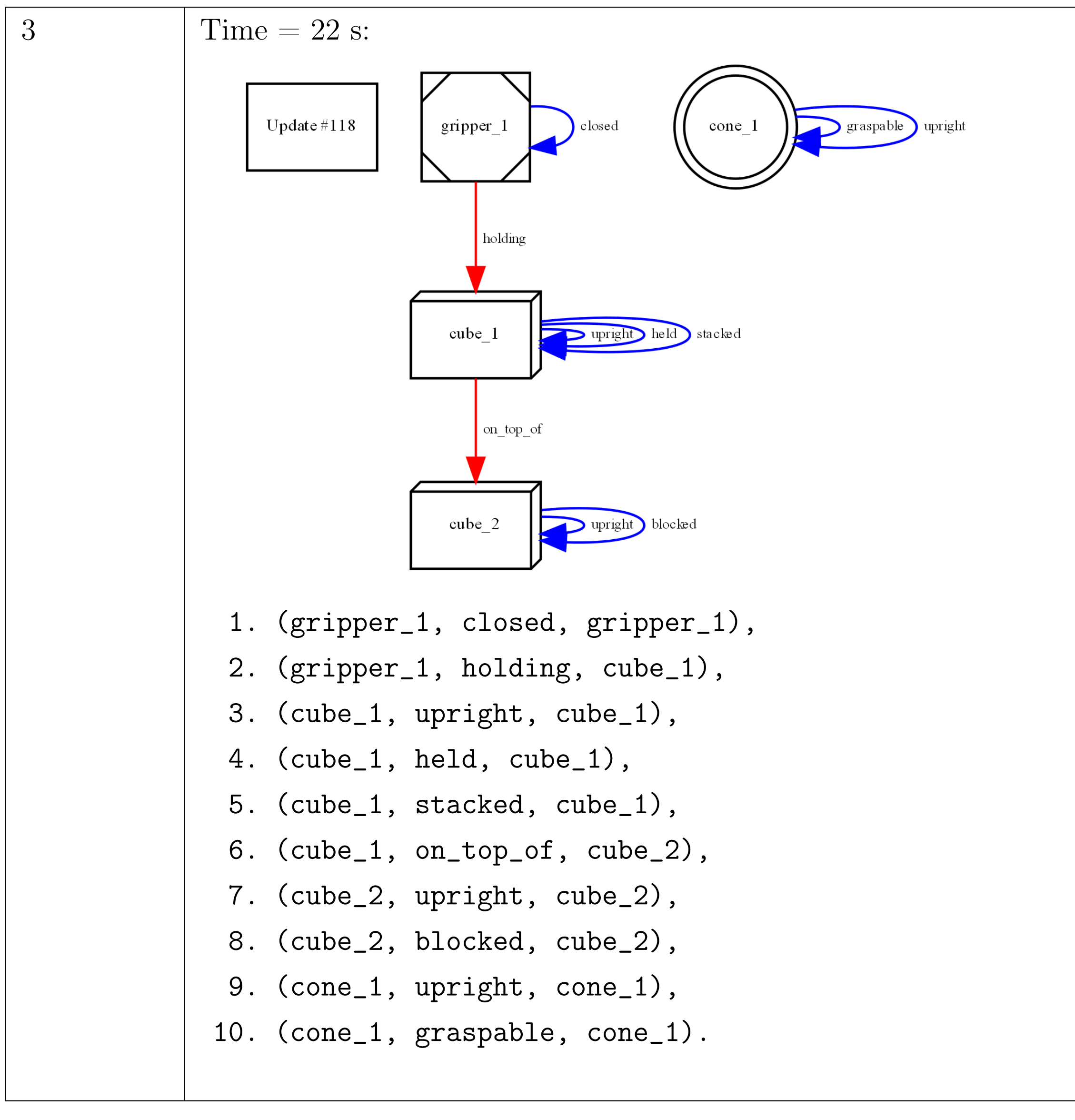
Table 3.1: Sequence of operations and corresponding semantics interpreted by a human observer (test 1).

The output from the program was the following:

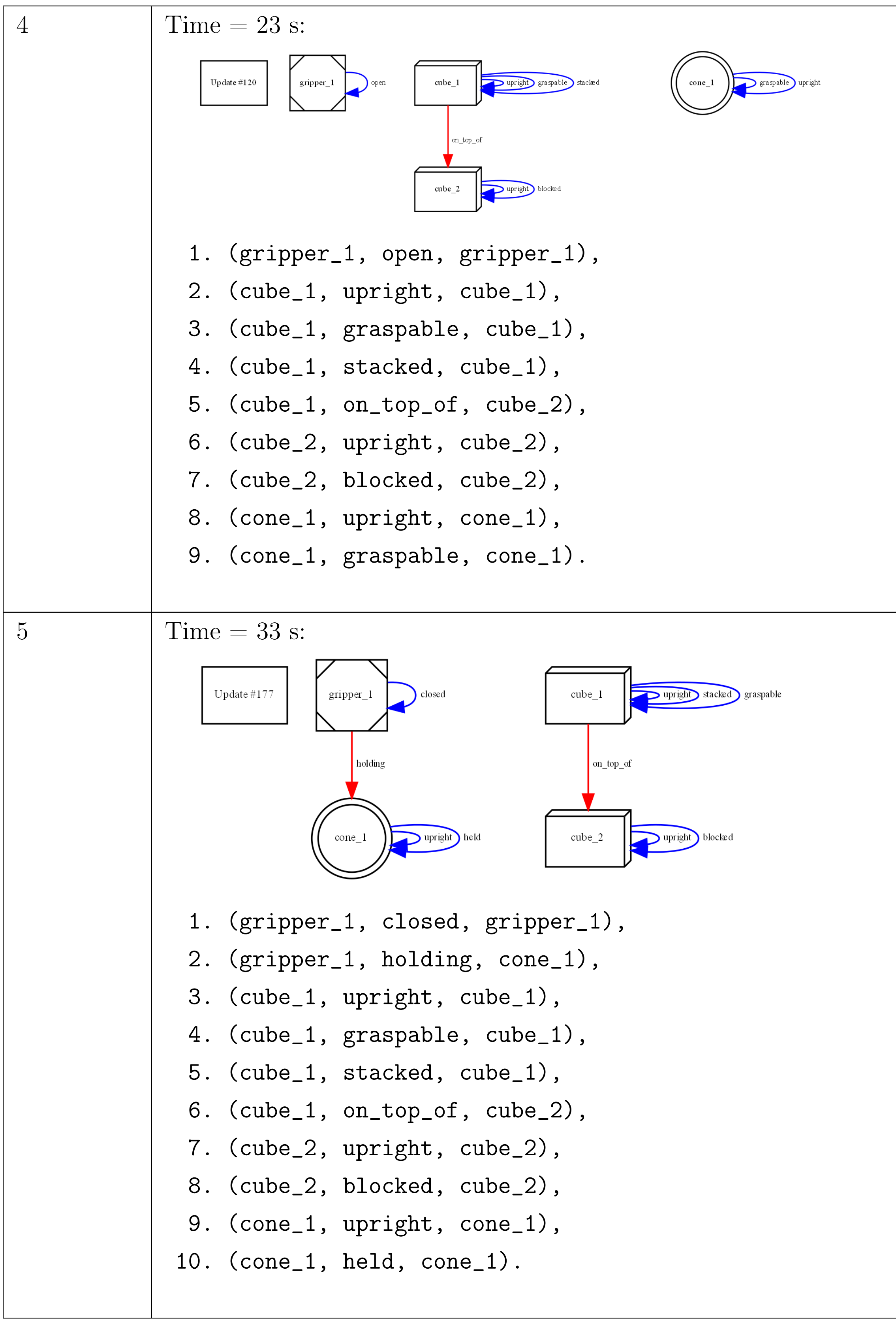
<table>
<tr><th>Operation</th><th>Generated Scene Graph</th></tr>
<tr><td>1</td><td>Time = 1 s:
Update #1 gripper_1 open cube_1 upright graspable cube_2 upright graspable cone_1 upright graspable

1. (gripper_1, open, gripper_1),
2. (cube_1, upright, cube_1),
3. (cube_1, graspable, cube_1),
4. (cube_2, upright, cube_2),
5. (cube_2, graspable, cube_2),
6. (cone_1, upright, cone_1),
7. (cone_1, graspable, cone_1).</td></tr>
<tr><td>2</td><td>Time = 11 s:
Update #51 gripper_1 closed cube_2 upright graspable cone_1 graspable upright holding cube_1 upright held

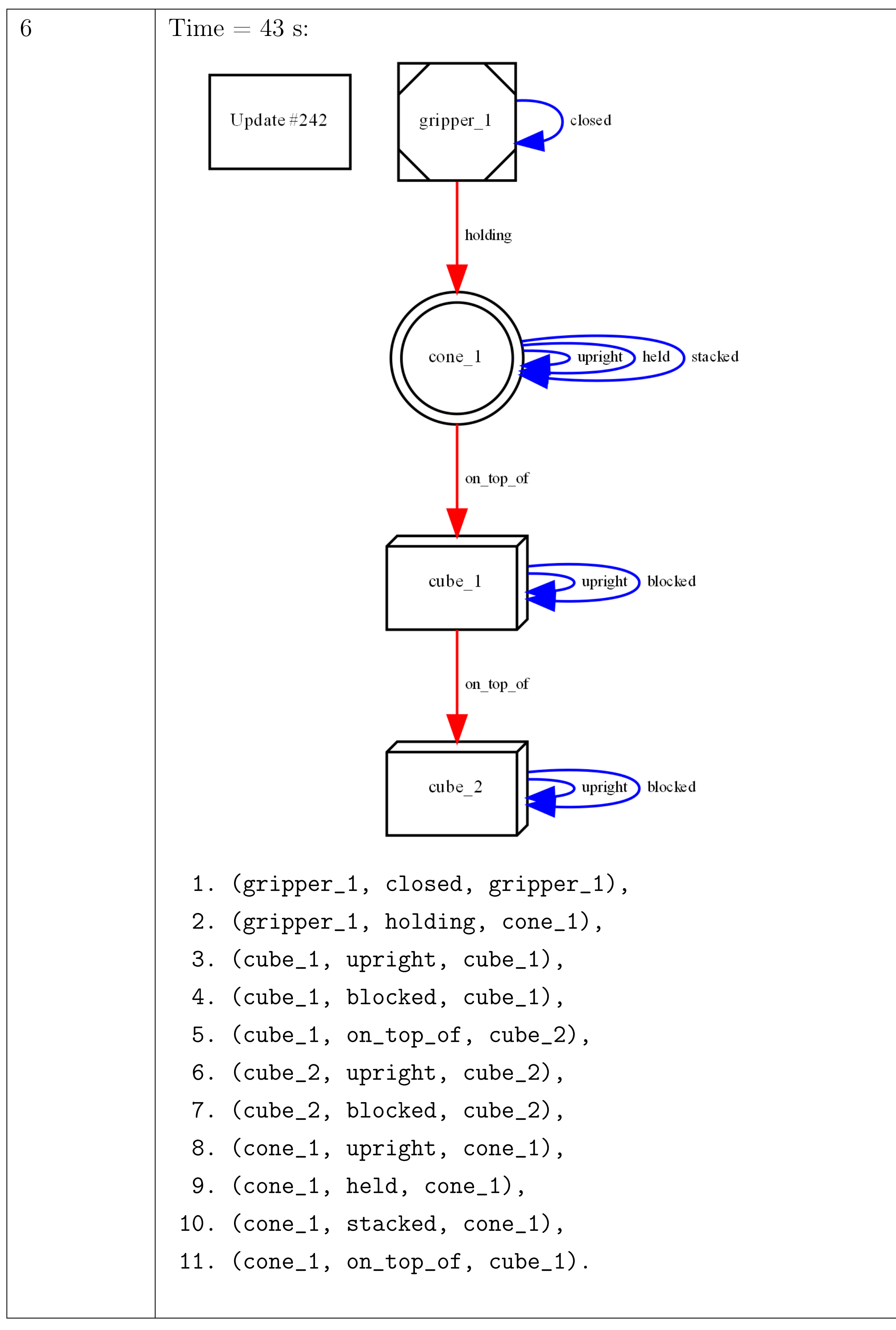
1. (gripper_1, closed, gripper_1),
2. (gripper_1, holding, cube_1),
3. (cube_1, upright, cube_1),
4. (cube_1, held, cube_1),
5. (cube_2, upright, cube_2),
6. (cube_2, graspable, cube_2),
7. (cone_1, upright, cone_1),
8. (cone_1, graspable, cone_1).</td></tr>
</table>



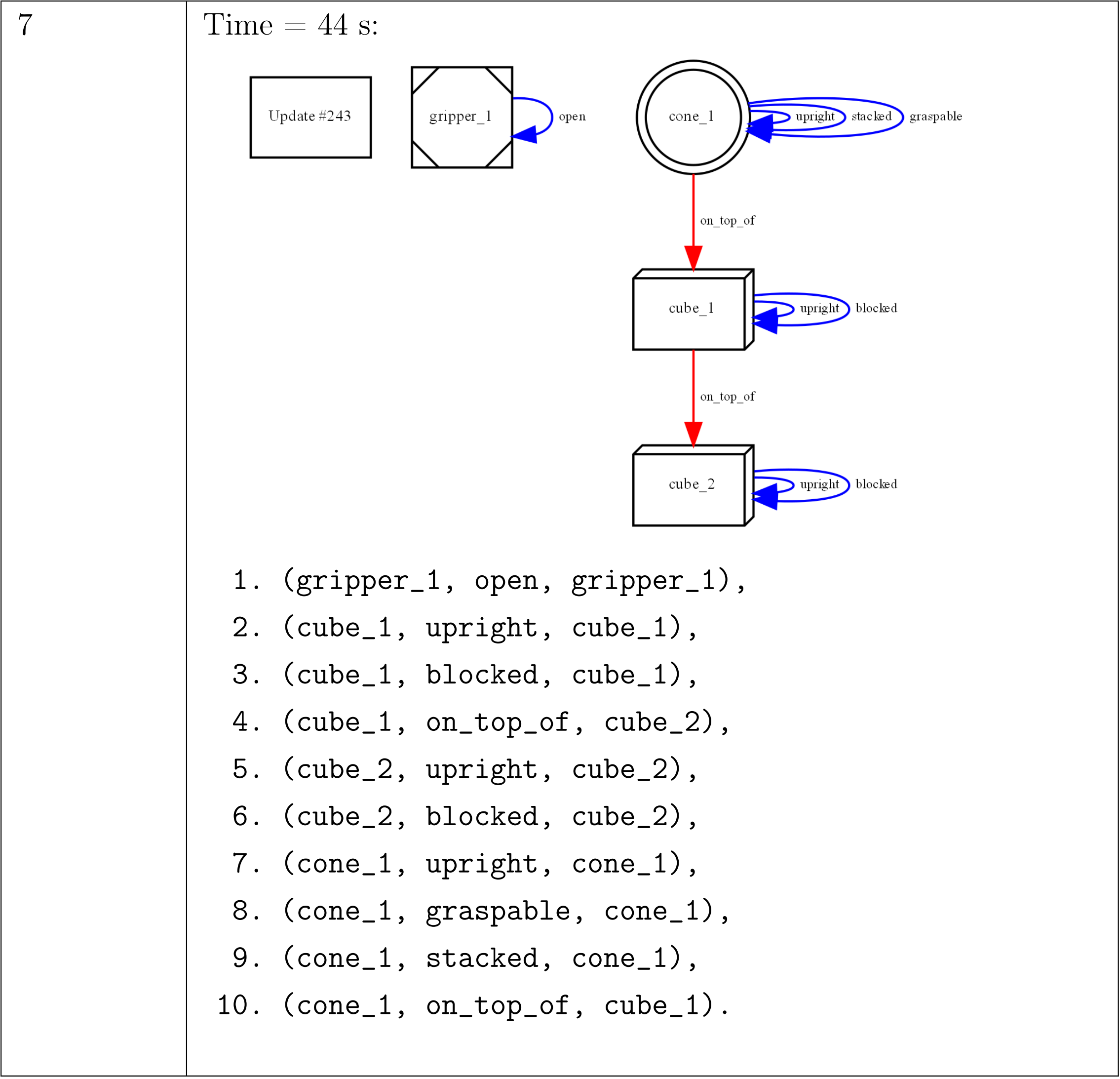
3	Time = 22 s: 1. (gripper_1, closed, gripper_1), 2. (gripper_1, holding, cube_1), 3. (cube_1, upright, cube_1), 4. (cube_1, held, cube_1), 5. (cube_1, stacked, cube_1), 6. (cube_1, on_top_of, cube_2), 7. (cube_2, upright, cube_2), 8. (cube_2, blocked, cube_2), 9. (cone_1, upright, cone_1), 10. (cone_1, graspable, cone_1).



4	Time = 23 s: 1. (gripper_1, open, gripper_1), 2. (cube_1, upright, cube_1), 3. (cube_1, graspable, cube_1), 4. (cube_1, stacked, cube_1), 5. (cube_1, on_top_of, cube_2), 6. (cube_2, upright, cube_2), 7. (cube_2, blocked, cube_2), 8. (cone_1, upright, cone_1), 9. (cone_1, graspable, cone_1).
5	Time = 33 s: 1. (gripper_1, closed, gripper_1), 2. (gripper_1, holding, cone_1), 3. (cube_1, upright, cube_1), 4. (cube_1, graspable, cube_1), 5. (cube_1, stacked, cube_1), 6. (cube_1, on_top_of, cube_2), 7. (cube_2, upright, cube_2), 8. (cube_2, blocked, cube_2), 9. (cone_1, upright, cone_1), 10. (cone_1, held, cone_1).



6	Time = 43 s: 1. (gripper_1, closed, gripper_1), 2. (gripper_1, holding, cone_1), 3. (cube_1, upright, cube_1), 4. (cube_1, blocked, cube_1), 5. (cube_1, on_top_of, cube_2), 6. (cube_2, upright, cube_2), 7. (cube_2, blocked, cube_2), 8. (cone_1, upright, cone_1), 9. (cone_1, held, cone_1), 10. (cone_1, stacked, cone_1), 11. (cone_1, on_top_of, cube_1).



7	Time = 44 s: 1. (gripper_1, open, gripper_1), 2. (cube_1, upright, cube_1), 3. (cube_1, blocked, cube_1), 4. (cube_1, on_top_of, cube_2), 5. (cube_2, upright, cube_2), 6. (cube_2, blocked, cube_2), 7. (cone_1, upright, cone_1), 8. (cone_1, graspable, cone_1), 9. (cone_1, stacked, cone_1), 10. (cone_1, on_top_of, cube_1).

Table 3.2: Operations and visual triplets generated by the program (test 1).

By comparing the two tables, we notice that the outcome is the same as the one predicted, therefore the requirement of completeness was fulfilled. Taking now into consideration the efficiency of the algorithm, we can do a simple analysis focusing on the time that it takes for the computations. At some points during the execution, the machine could spare some time in an idle state.

The computation time is incredibly affected by the displaying and printing operations, which take up most of the resources:

Output image enabled			**Output image disabled**		
Operating t.	**Idle t.**	**Cycle t.**	**Operating t.**	**Idle t.**	**Cycle t.**
$\approx$ 20 ms	$\approx$ 80 ms	100 ms	$\approx$ 13 ms	$\approx$ 47 ms	60 ms

Table 3.3: Average operating and idle times, cycle times, in the cases of full print options and no print option (test 1).

In the first execution, the program was run with a cycle time of 100 ms, but this limit was overshot in some cycles due to the computations done by the *GraphViz* image generation tools. However, the algorithm was extremely efficient overall and it may even be tuned to have a lower cycle time. This would help decrease the average idle time and improve the tracking of the scene, making it even more up-to-date. It must be noted that these results mostly depend on the computational power of the machine running the algorithm.

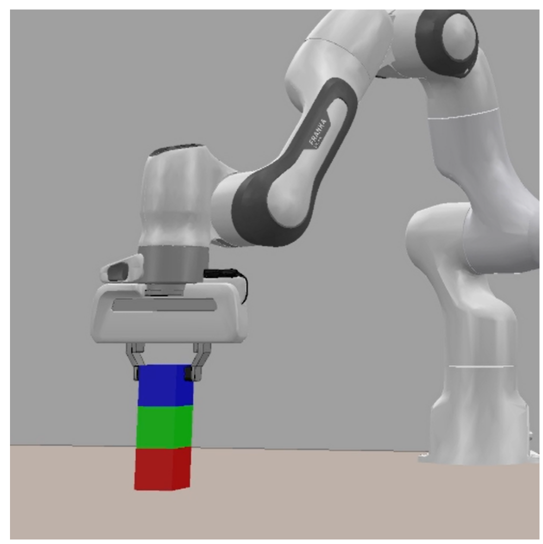
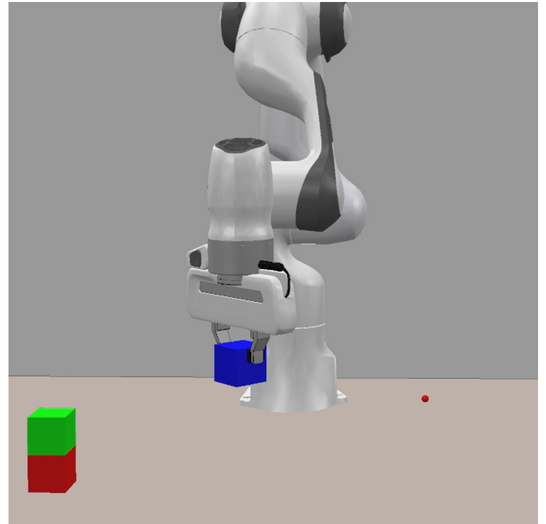
In the second scenario, we disabled the print options and we obtained better results. As we can see, the cycle time was reduced to 60 ms and is never overshot.

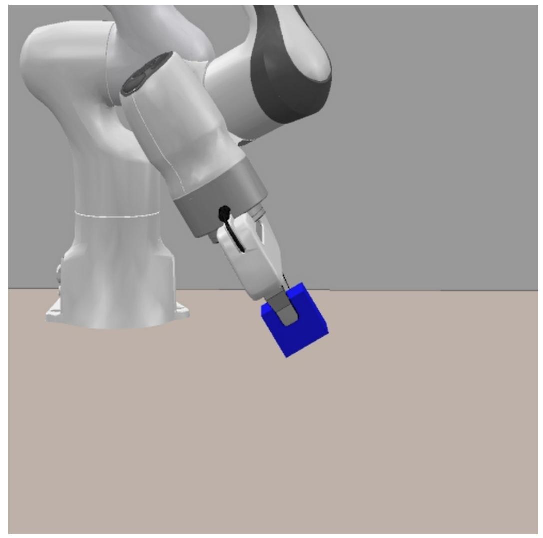
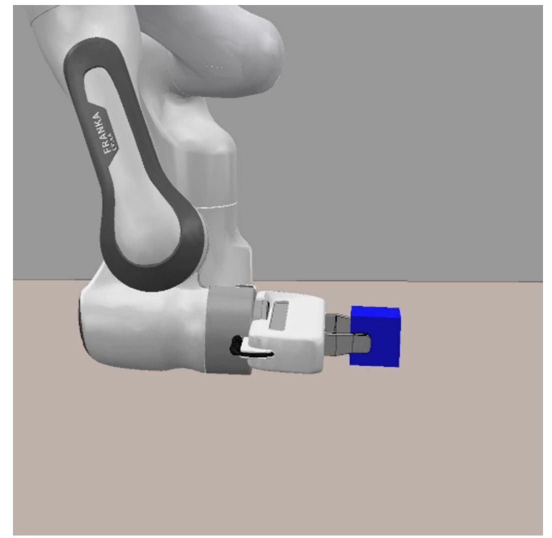
3.2. Test #2 — Unstacking and tilting

Finally, we consider a second, shorter, test case which involves three cubes. The robot performs an unstacking operation of the top cube, called Cube 3, but before laying it on the table it rotates the cube along the vertical axis and then along a horizontal axis. The outcome of the operation is a pile of two upright cubes and a single cube laying on one of its sides.

From a human point of view, the expected outcome is the following:

Operation description	Human interpretation
1. Three cubes are stacked on each other, and the gripper is open.	*Cube 1 is upright, graspable, and stacked. Cube 2 and 3 are upright and blocked. Cube 3 is on top of Cube 2, which is on top of Cube 1*

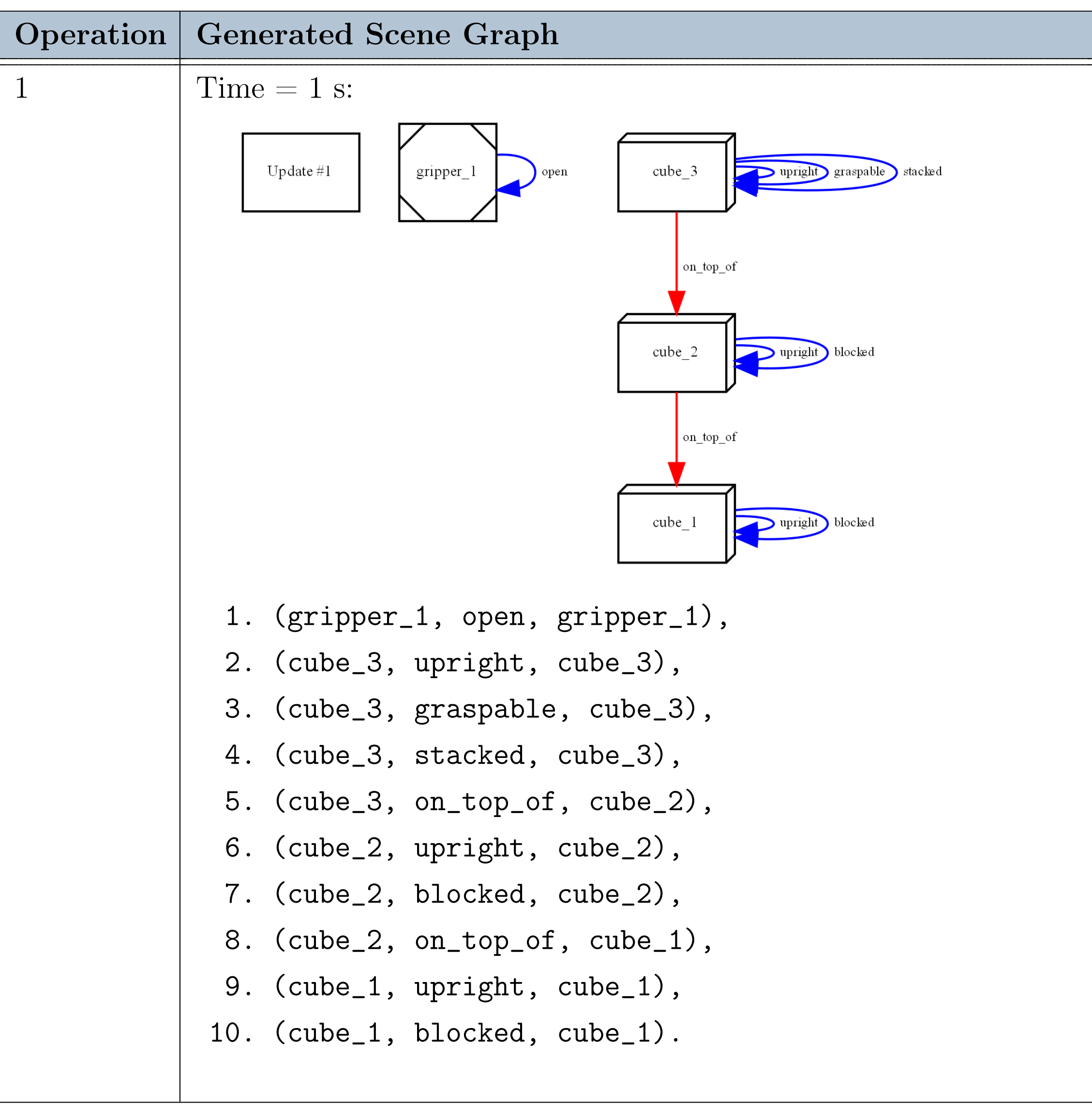
<table>
<tr>
<td>2. The gripper reaches the third cube on the top and holds it.
</td>
<td>Gripper 1 is closed and is holding Cube 3. Cube 3 is no longer graspable, but instead held, upright, and stacked. Cube 2 and 3 are still upright and blocked. Cube 3 is on top of Cube 2, which is on top of Cube 1.</td>
</tr>
<tr>
<td>3. The robot lifts the third cube while moving away and rotating the third cube around the vertical axis.
</td>
<td>Gripper 1 is still closed and holding Cube 3, while Cube 3 is held and upright. Yaw rotation does not affect its “upright” descriptor. Cube 2 is now upright, graspable, stacked, and on top of Cube 1, which is still upright and blocked.</td>
</tr>
</table>

4. The gripper starts rotating the third cube on a side. 	*Gripper 1 is still closed and holding Cube 3, while Cube 3 is held. At one point it has lost the requirements for being considered upright, but it is not on a side yet. Cube 2 and 3 are on top of each other as before: Cube 2 is upright, graspable, and stacked; Cube 1 is upright and blocked.*
5. The gripper finishes rotating the third cube on a side. 	*Gripper 1 is still closed and holding Cube 3, while Cube 3 is held and on a side. Cube 2 is upright, graspable, stacked, and on top of Cube 1, which is upright and blocked.*

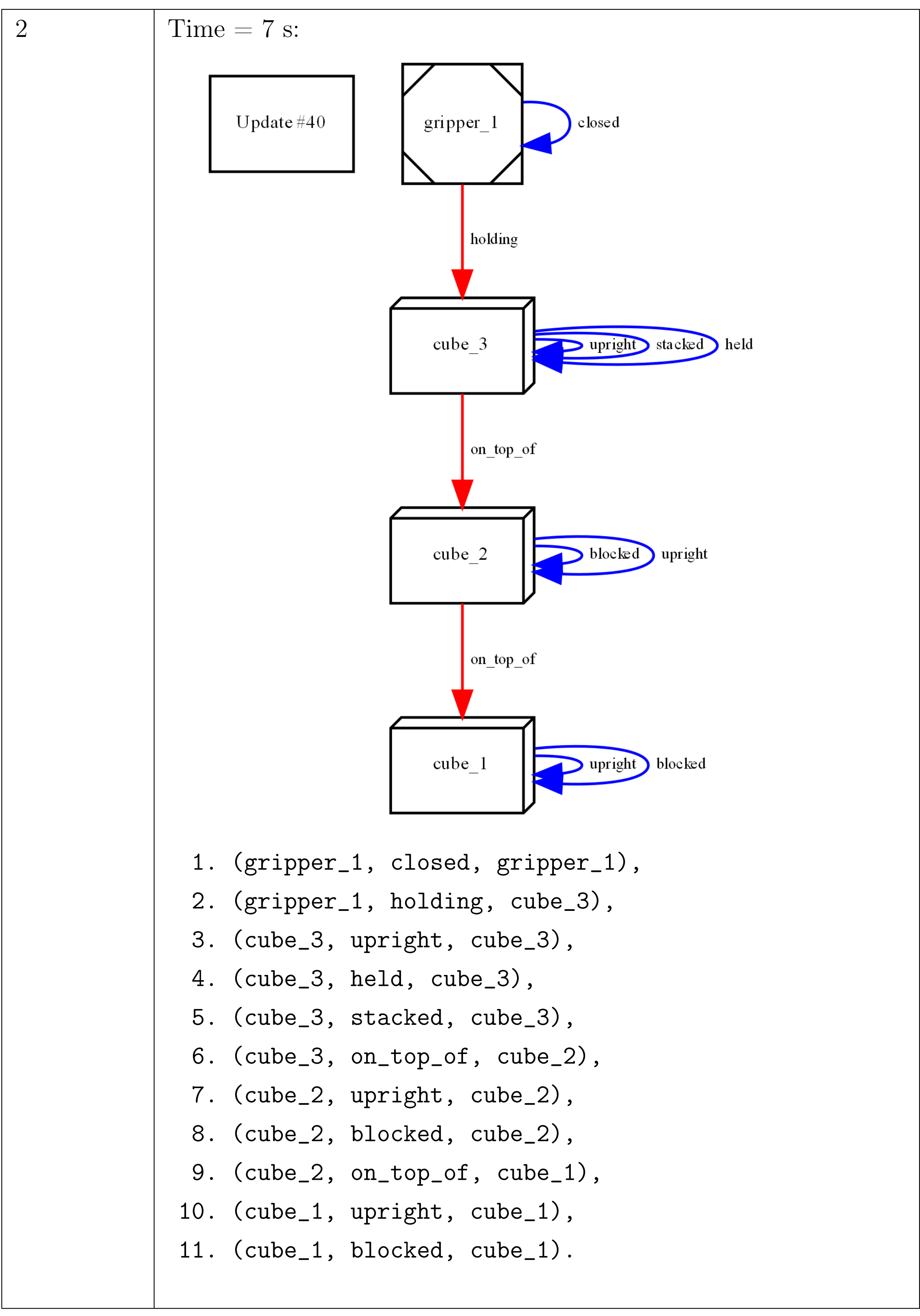
6. The gripper releases the cube.	*Gripper 1 is open. Cube 3 is graspable and on a side. Cube 2 and 1 as before: on top of each other. Cube 2 is stacked, graspable, and upright; while Cube 1 is upright and blocked.*

Table 3.4: Sequence of operations and corresponding semantics interpreted by a human observer (test 2).

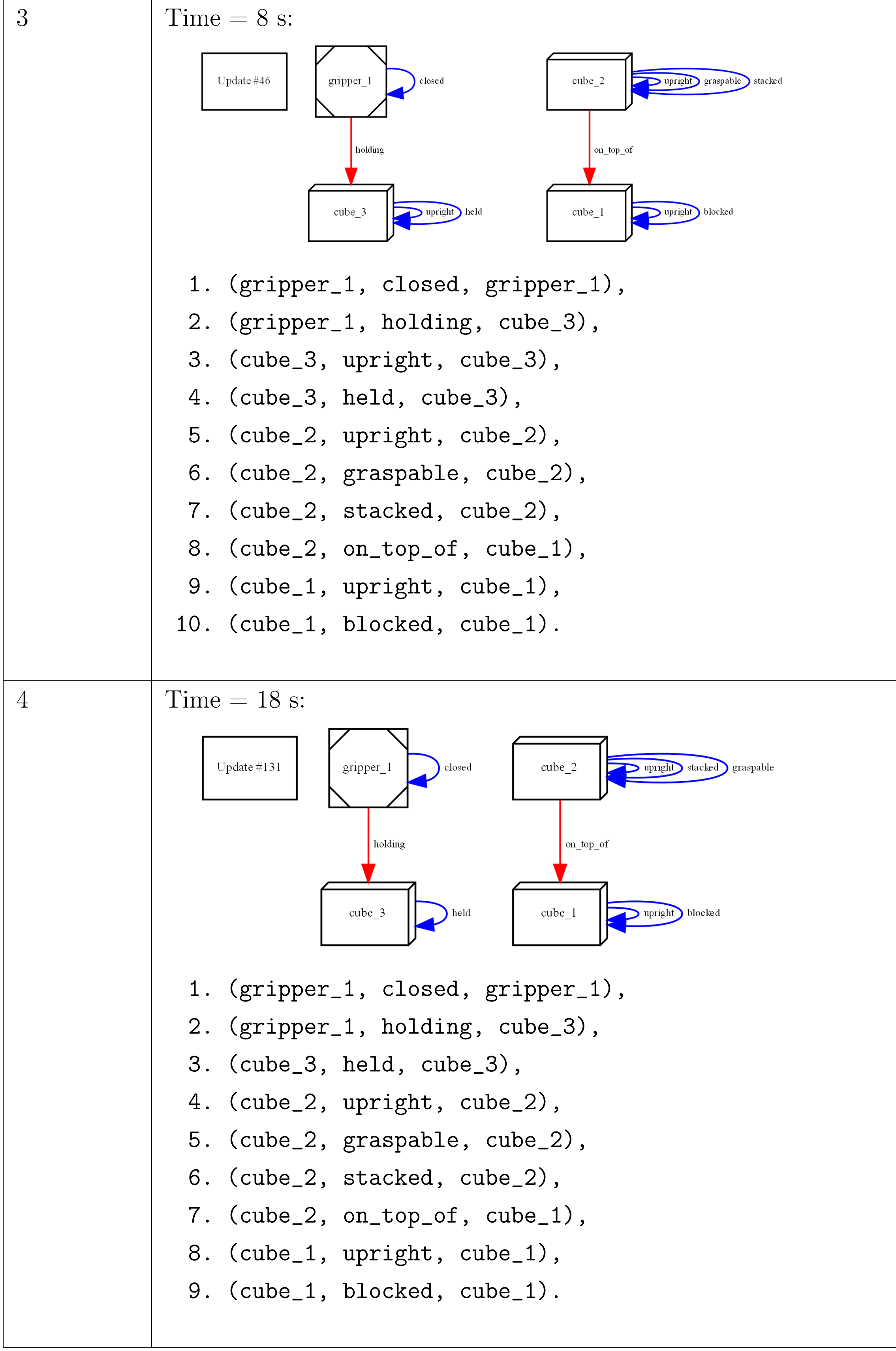
The output from the program was the following:



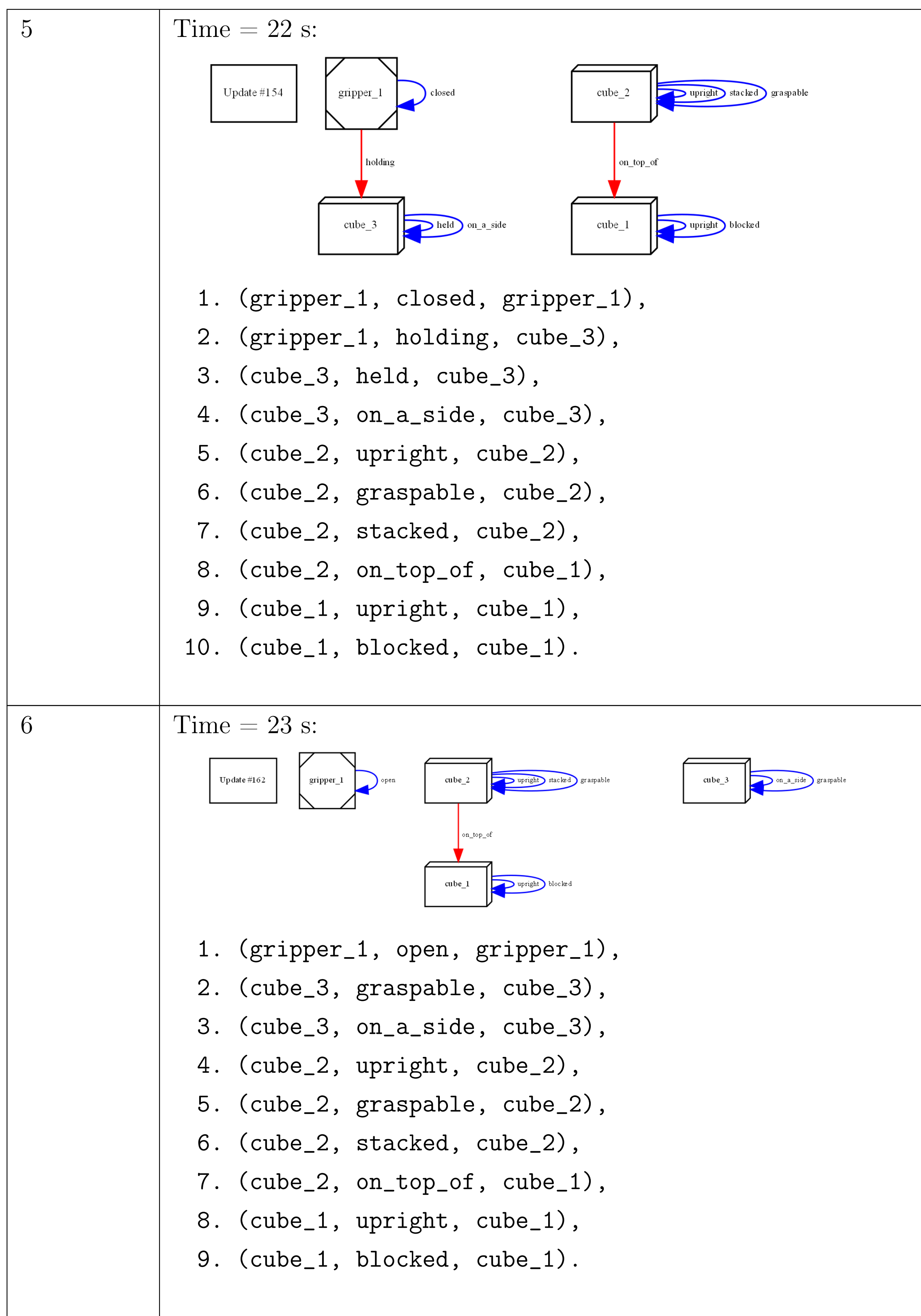
Operation	Generated Scene Graph
1	Time = 1 s: 1. (gripper_1, open, gripper_1), 2. (cube_3, upright, cube_3), 3. (cube_3, graspable, cube_3), 4. (cube_3, stacked, cube_3), 5. (cube_3, on_top_of, cube_2), 6. (cube_2, upright, cube_2), 7. (cube_2, blocked, cube_2), 8. (cube_2, on_top_of, cube_1), 9. (cube_1, upright, cube_1), 10. (cube_1, blocked, cube_1).



2	Time = 7 s: 1. (gripper_1, closed, gripper_1), 2. (gripper_1, holding, cube_3), 3. (cube_3, upright, cube_3), 4. (cube_3, held, cube_3), 5. (cube_3, stacked, cube_3), 6. (cube_3, on_top_of, cube_2), 7. (cube_2, upright, cube_2), 8. (cube_2, blocked, cube_2), 9. (cube_2, on_top_of, cube_1), 10. (cube_1, upright, cube_1), 11. (cube_1, blocked, cube_1).



3	Time = 8 s: 1. (gripper_1, closed, gripper_1), 2. (gripper_1, holding, cube_3), 3. (cube_3, upright, cube_3), 4. (cube_3, held, cube_3), 5. (cube_2, upright, cube_2), 6. (cube_2, graspable, cube_2), 7. (cube_2, stacked, cube_2), 8. (cube_2, on_top_of, cube_1), 9. (cube_1, upright, cube_1), 10. (cube_1, blocked, cube_1).
4	Time = 18 s: 1. (gripper_1, closed, gripper_1), 2. (gripper_1, holding, cube_3), 3. (cube_3, held, cube_3), 4. (cube_2, upright, cube_2), 5. (cube_2, graspable, cube_2), 6. (cube_2, stacked, cube_2), 7. (cube_2, on_top_of, cube_1), 8. (cube_1, upright, cube_1), 9. (cube_1, blocked, cube_1).



5	Time = 22 s: 1. (gripper_1, closed, gripper_1), 2. (gripper_1, holding, cube_3), 3. (cube_3, held, cube_3), 4. (cube_3, on_a_side, cube_3), 5. (cube_2, upright, cube_2), 6. (cube_2, graspable, cube_2), 7. (cube_2, stacked, cube_2), 8. (cube_2, on_top_of, cube_1), 9. (cube_1, upright, cube_1), 10. (cube_1, blocked, cube_1).
6	Time = 23 s: 1. (gripper_1, open, gripper_1), 2. (cube_3, graspable, cube_3), 3. (cube_3, on_a_side, cube_3), 4. (cube_2, upright, cube_2), 5. (cube_2, graspable, cube_2), 6. (cube_2, stacked, cube_2), 7. (cube_2, on_top_of, cube_1), 8. (cube_1, upright, cube_1), 9. (cube_1, blocked, cube_1).

Table 3.5: Operations and visual triplets generated by the program (test 2).

Once again, the outcome is the same as the one predicted. The generation algorithm produced a complete Scene Graph.

We now consider the same analysis on the efficiency of the computations:

Output image enabled			**Output image disabled**		
Operating t.	**Idle t.**	**Cycle t.**	**Operating t.**	**Idle t.**	**Cycle t.**
$\approx$ 19 ms	$\approx$ 81 ms	100 ms	$\approx$ 10 ms	$\approx$ 40 ms	50 ms

Table 3.6: Average operating and idle times, cycle times, in the cases of full print options and no print option (test 2).

We find a positive result in terms of efficiency. In the first case, with image generation enabled, the cycle time is sometimes overshot by the graphical computations required by *GraphViz*. Nonetheless, on average the operating time is still lower than the limit. When the print options are not enabled, the algorithm never exceeds the limit cycle time and can definitely run in real-time. It would be possible to tweak the program so that it runs the update cycle with a higher frequency.

4 | Conclusions

The objectives of this thesis were manifold. First, we considered the problem of cobot programming simplification through semantic techniques, which concerns the extraction of meaning from the scene. By meaning, we intend the set of relations and attributes involving the instances featured in the scene. The past literature offers a standard versatile tool for the representation of such data structure: the scene graph. Taking into consideration the classification of semantic systems, we focused on provided semantics methods, which require the machine to be given some knowledge about the environment rules beforehand.

After researching the topic, the methodology we followed was split into two steps: on one hand, we had to define the strategies for scene data collection, which was implemented through a 3-dimensional simulation and socket connection; on the other, we developed a SG generation algorithm that translates the socket data into the corresponding SG.

The algorithm was developed with two main objectives in mind, which were achieved successfully as demonstrated in Chapter 3. The SG generated had to be complete and the process had to be efficient enough to happen in real-time while the simulation was running. The first objective implies the fact that the SG created by the program had to be sufficient and appropriate to the conditions selected as the hypothesis of the working scenario. The second one indicates that the process had to run dynamically, avoiding useless or resource-intensive computations.

Redundancy was taken into account and incorporated as an intentional feature of the system. In practice, after the creation of non-redundant descriptive triplets, the algorithm considers relational triplets. Here we included an additional step for the creation of redundant descriptive triplets which will benefit SG analysis tools for the recognition of the pre- and post-conditions for actions. This means that, for example, it will be more straightforward to check if an object is available for an unstacking operation because the action planner will only need to know if the object is described as stacked. Computation is therefore faster since the search problem has become much simpler.

Our program also relies on prior knowledge data in the form of `InitializationTables`.

These tables contain all the relevant data regarding the triplet combinations that are allowed in the environment. They simplify the generation of the SG as they reduce the range of possibilities in a finite space that excludes impossible scenarios. Object update strategies are also relevant in order to minimize the time of computation for computing the top/bottom surfaces, as well as for reducing the number of objects considered for potential graph pruning or expansion.

The C++ script was developed with scalability in mind, so that it would be possible to easily expand the code with new methods to check the validity of different kinds of predicates and attributes, or alternatively new types of objects with other geometries. The validity check mostly relies on straightforward geometrical computations that involve hysteresis to evaluate the threshold differently depending on the kind of action: either building or destroying a triplet. Nonetheless, the system is not the smartest implementation and requires manual coding as it does not rely on fancier automatic techniques such as machine learning. This means that this algorithm, even if we proved it to be efficient and reliable, has a lot of potential for further improvements.

Bibliography

[1] H. Dang and P. K. Allen. Semantic grasping: Planning robotic grasps functionally suitable for an object manipulation task. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1311–1317. IEEE, 2012.

[2] G. Du, K. Wang, S. Lian, and K. Zhao. Vision-based robotic grasping from object localization, object pose estimation to grasp estimation for parallel grippers: a review. *Artificial Intelligence Review*, 54(3):1677–1734, 2021.

[3] S. Garg, N. Sünderhauf, F. Dayoub, D. Morrison, A. Cosgun, G. Carneiro, Q. Wu, T.-J. Chin, I. Reid, S. Gould, et al. Semantics for robotic mapping, perception and interaction: A survey. *Foundations and Trends® in Robotics*, 8(1–2):1–224, 2020.

[4] Y. Guo, F. A. Sohel, M. Bennamoun, J. Wan, and M. Lu. Rops: A local feature descriptor for 3d rigid objects based on rotational projection statistics. In *2013 1st International Conference on Communications, Signal Processing, and their Applications (ICCSPA)*, pages 1–6. IEEE, 2013.

[5] Y. Guo, M. Bennamoun, F. Sohel, M. Lu, and J. Wan. 3d object recognition in cluttered scenes with local surface features: A survey. *IEEE Transactions on pattern analysis and machine intelligence*, 36(11):2270–2287, 2014.

[6] E. Jang, S. Vijayanarasimhan, P. Pastor, J. Ibarz, and S. Levine. End-to-end learning of semantic grasping. *arXiv preprint arXiv:1707.01932*, 2017.

[7] O. Kroemer, S. Niekum, and G. Konidaris. A review of robot learning for manipulation: Challenges, representations, and algorithms. *The Journal of Machine Learning Research*, 22(1):1395–1476, 2021.

[8] A. Myers, C. L. Teo, C. Fermüller, and Y. Aloimonos. Affordance detection of tool parts from geometric features. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1374–1381. IEEE, 2015.

[9] K. Ramirez-Amaro, Y. Yang, and G. Cheng. A survey on semantic-based methods

for the understanding of human movements. *Robotics and Autonomous Systems*, 119: 31–50, 2019.

[10] Treccani.it. Definizione di 'semantica'. `https://www.treccani.it/vocabolario/semantica/`.

[11] G. Zhu, L. Zhang, Y. Jiang, Y. Dang, H. Hou, P. Shen, M. Feng, X. Zhao, Q. Miao, S. A. A. Shah, et al. Scene graph generation: A comprehensive survey. *arXiv preprint arXiv:2201.00443*, 2022.

A | Semantic vocabulary of the program

A.1. List of predicates

Predicate type	Explanation
`on_top_of`	Used when the subject stands on top of the object. The following three conditions must be met: • Height difference of the centers of the two objects are in *contact range*, which varies depending on the geometrical solids involved. • Both objects have a defined top and a bottom surface respectively, therefore both are sufficiently horizontal to the ground. • The vertical projections of the two surfaces are intersecting.
`holding`	Only an object of type `gripper` can be the subject of this predicate. When the gripper is closed and the center of the gripper is sufficiently near to that of an object, then the gripper is holding it.
`close_to`	Simply checks when two objects are close enough. For the purposes of the experimental validation in Chapter 3, the closeness threshold was set to zero to avoid this relation, as it always builds two edges, one per direction.

Table A.1: List of the predicate types of the program

A.2. List of attributes

Attribute type	Explanation
open	Only an object of type gripper can be the subject of this attribute. It is true when the gripper state of the ObjectNode corresponding to the gripper is true.
closed	Opposite of open.
held	It is true whenever the subject is the object of a holding relational triplet.
upright	It is true whenever the local reference frame of the subject has its z^+ direction sufficiently parallel to the z^+ direction of the environment.
on_a_side	It is true whenever the local reference frame of the subject has its x^+ or its y^+ direction sufficiently parallel to the z^+ or z^- direction of the environment.
upside_down	It is true whenever the local reference frame of the subject has its z^+ direction sufficiently parallel to the z^- direction of the environment.
graspable	It is true whenever the subject is neither blocked nor held.
stacked	It is true whenever the subject is on_top_of another object and it is not the subject.
blocked	It is true whenever the subject is the object of an on_top_of relational triplet.

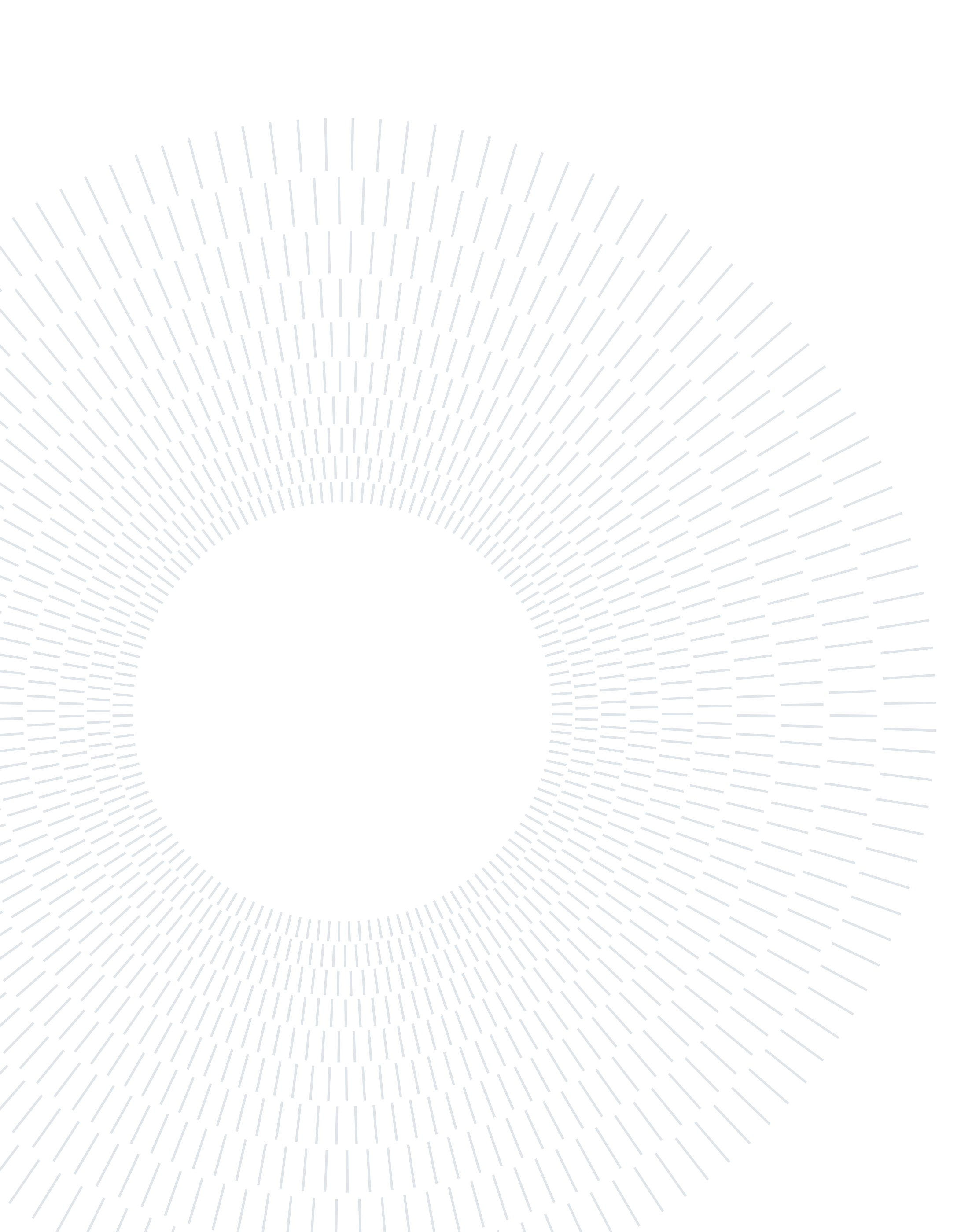
Table A.2: List of the attribute types of the program

A.3. List of objects

Object type	Explanation
gripper	It is the end effector of the robotic arm. It can perform movements and manipulate the pose of the other objects in the scene.

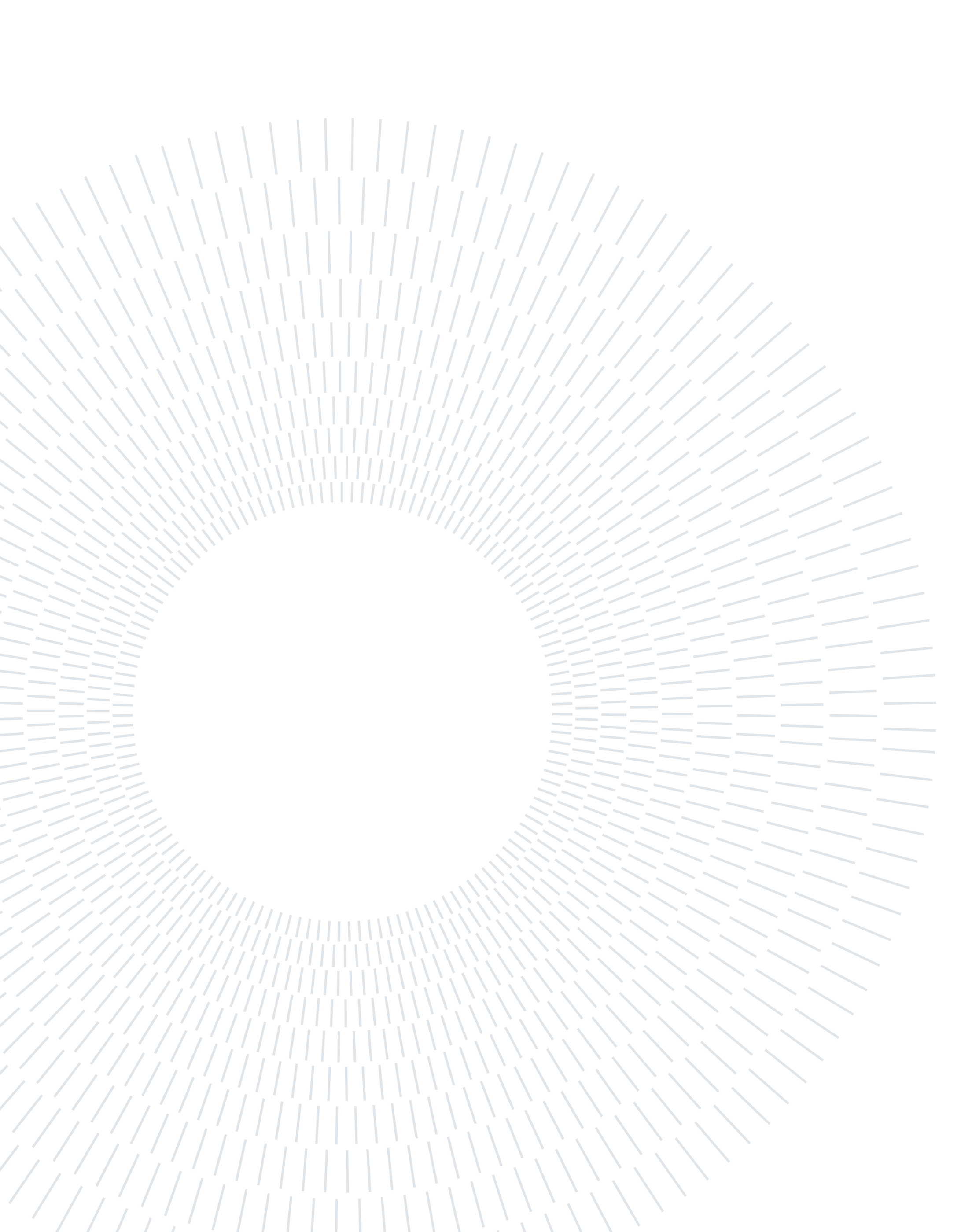
`cube`	A basic cube solid.
`cone`	A cone solid with a height equal to the base diameter.

Table A.3: List of the object types of the program



List of Figures

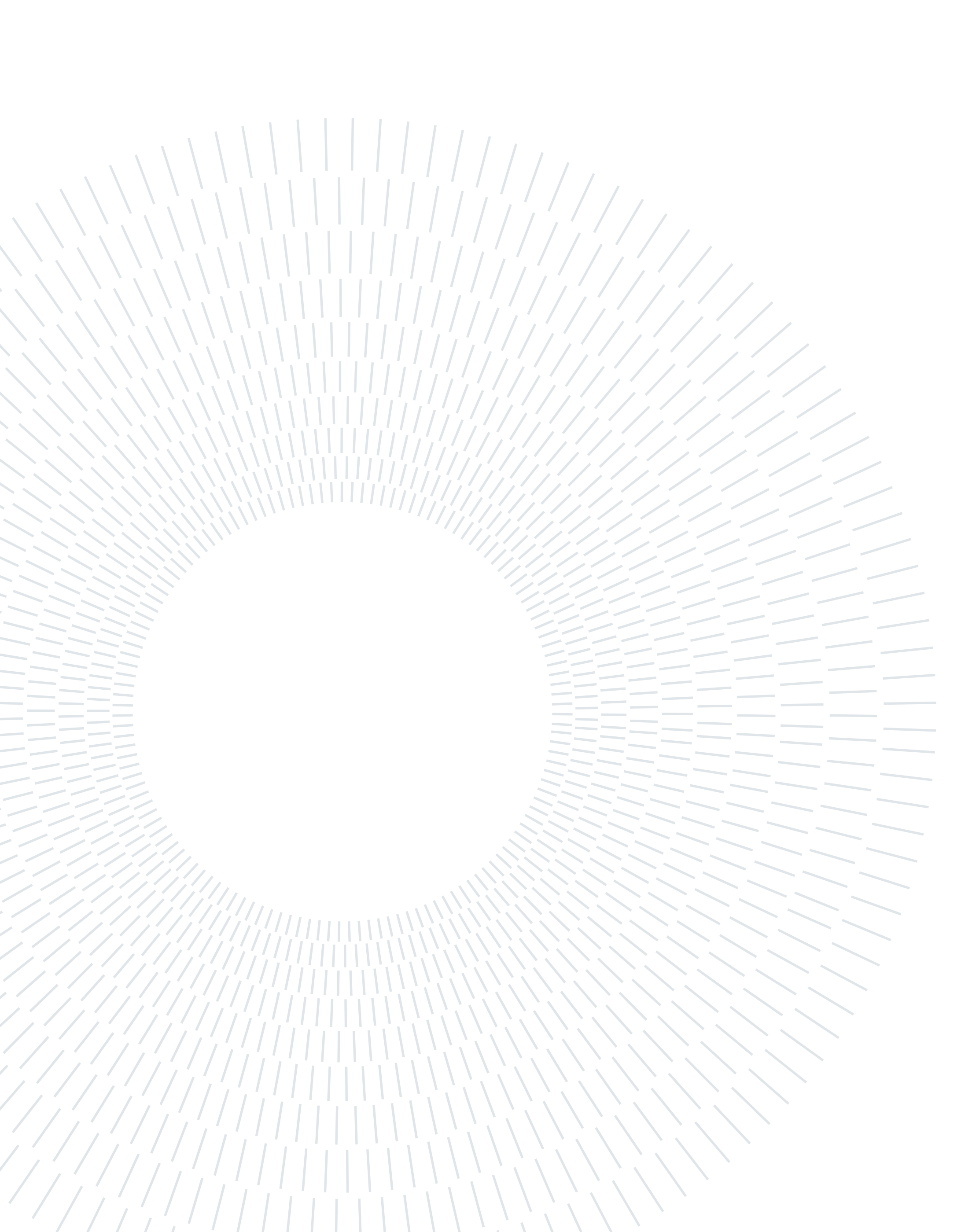
2.1 A screenshot from the CoppeliaSim environment 9
2.2 An example of graphical output 19



List of Tables

3.1 Sequence of operations and corresponding semantics interpreted by a human observer (test 1). 25
3.2 Operations and visual triplets generated by the program (test 1). 30
3.3 Average operating and idle times, cycle times, in the cases of full print options and no print option (test 1). 31
3.4 Sequence of operations and corresponding semantics interpreted by a human observer (test 2). 35
3.5 Operations and visual triplets generated by the program (test 2). 39
3.6 Average operating and idle times, cycle times, in the cases of full print options and no print option (test 2). 40

A.1 List of the predicate types of the program 45
A.2 List of the attribute types of the program 46
A.3 List of the object types of the program 47



Acknowledgements

I would like to take a moment to thank those who have played a part in completing this work and have been by my side during these years.

I express my appreciation to Isacco Zappa, my supervisor, for his prompt support, as well as for his valuable suggestions that greatly assisted me in writing and organizing my thesis. His contribution was crucial in achieving the success of this work. I thank Prof. Paolo Rocco and Andrea Maria Zanchettin for providing me with invaluable guidance on how to advance with my work and for giving me the opportunity to work on such an interesting topic.

My family deserves my sincere gratitude for their unwavering support throughout my education. To my mother, to my father, and to my sister Sara — thank you for your constant encouragement and love. Your belief in me has been invaluable, and I could not have accomplished this without you.

Finally, I am thankful for the inspiration of my friends and colleagues throughout my life. To my dear friends from my hometown of Novara, your presence has been a constant source of motivation. To my colleagues and friends at the Polytechnic University of Milan, I will always cherish the wonderful moments and experiences we shared. Your company and guidance have been invaluable in shaping my academic journey.

Alessandro Petazzi